Análise e Projeto Orientados por Objetos

Aula 08 – Padrões GoF (Observer e Builder)

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

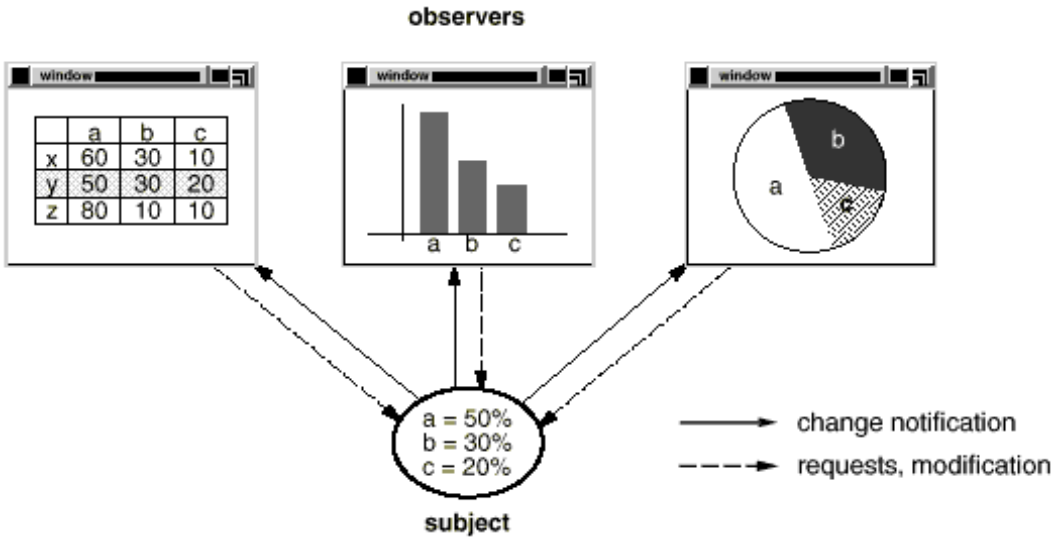
Padrões GoF

- 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

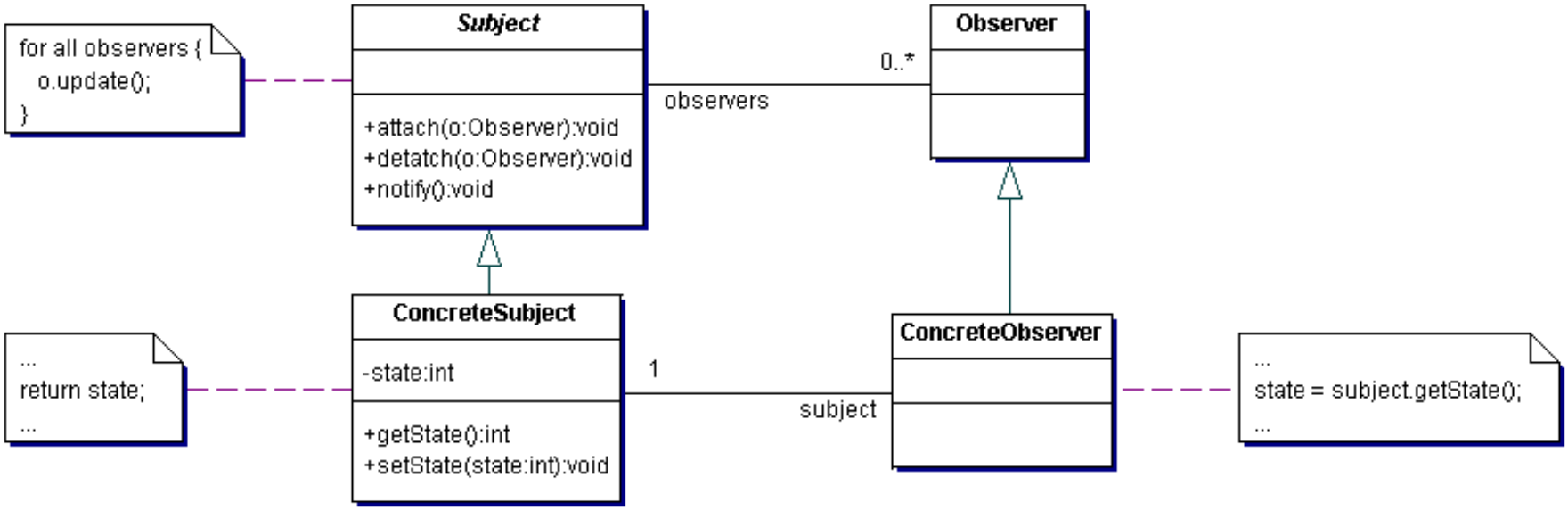
Observer

- Existem situações onde diversos **objetos mudam seu estado de acordo com a mudança de estado de outro objeto**.

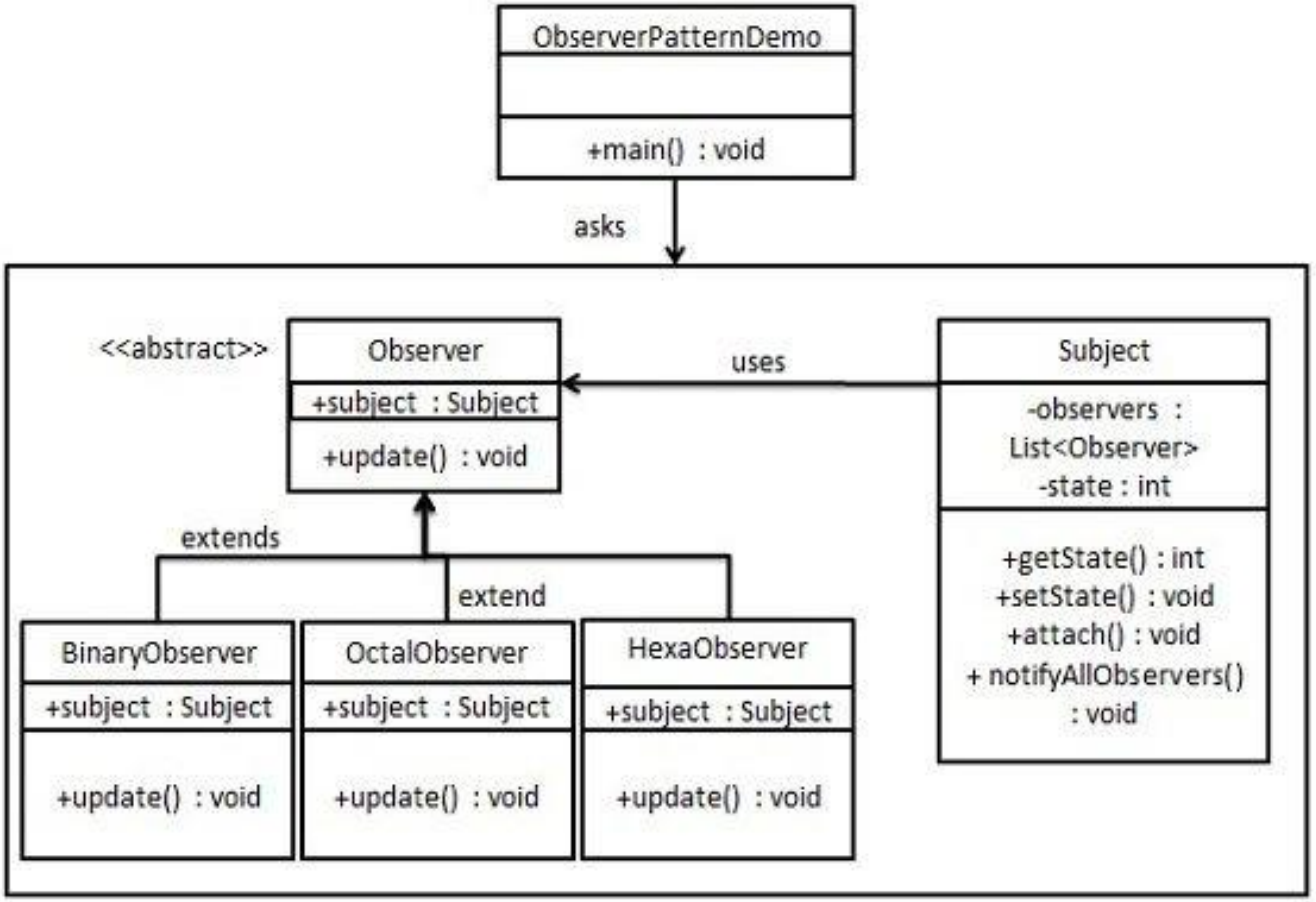
- Define uma relação de dependência 1:N entre objetos, de tal forma que, quando um objeto (assunto) tem seu estado modificado, os seus objetos dependentes (observadores) são notificados.



Observer


for all observers {
o.update();
}
Subject
+attach(o:Observer):void
+detatch(o:Observer):void
+notify():void
0..*
observers
Observer
...
return state;
...
ConcreteSubject
-state:int
+getState():int
+setState(state:int):void
1
subject
ConcreteObserver
...
state = subject.getState();
...

Observer – Exemplo



Observer – Implementação

```
public class Subject
{
   private List<Observer> observers = new ArrayList<Observer>();
   private int state;

   public int getState(){
      return state;
   }
   public void setState(int state){
      this.state = state;
      notifyAllObservers();
   }
   public void attach(Observer observer){
      observers.add(observer);
   }
   public void notifyAllObservers(){
      for (Observer observer : observers){
         observer.update();
      }
   }
}
```

Observer – Implementação

```
public abstract class Observer
{
   protected Subject subject;
   public abstract void update();
}
```

```
public class BinaryObserver extends Observer
{
   public BinaryObserver(Subject subject)
   {
      this.subject = subject;
      this.subject.attach(this);
   }

   @Override
   public void update()
   {
      System.out.println("Binary String: " +
                         Integer.toBinaryString(subject.getState()));
   }
}
```

Observer – Implementação

```
public class OctalObserver extends Observer
{

   public OctalObserver(Subject subject)
   {
      this.subject = subject;
      this.subject.attach(this);
   }

   @Override
   public void update()
   {
     System.out.println("Octal String: " +
                        Integer.toOctalString( subject.getState()));
   }
}
```

Observer – Implementação

```
public class HexaObserver extends Observer
{

   public HexaObserver(Subject subject)
   {
      this.subject = subject;
      this.subject.attach(this);
   }

   @Override
   public void update()
   {
      System.out.println("Hex String: " + Integer.toHexString(
                           subject.getState()).toUpperCase());
   }
}
```

Observer – Implementação

```
public static void main(String[] args)
{
   Subject subject = new Subject();

   Observer ho = new HexaObserver(subject);
   Observer ob = new OctalObserver(subject);
   Observer bo = new BinaryObserver(subject);

   System.out.println("First state change: 15");
   subject.setState(15);
   System.out.println("Second state change: 10");
   subject.setState(10);
}
```

Observer – Aplicabilidade

- Quando **uma mudança em um objeto requer uma mudança em outros**, e não se sabe como esses outros objetos efetivamente fazem suas mudanças;

- Quando um objeto deve poder **notificar outros objetos** sem assumir nada sobre eles. Dessa forma evita-se que os objetos envolvidos fiquem fortemente acoplados;

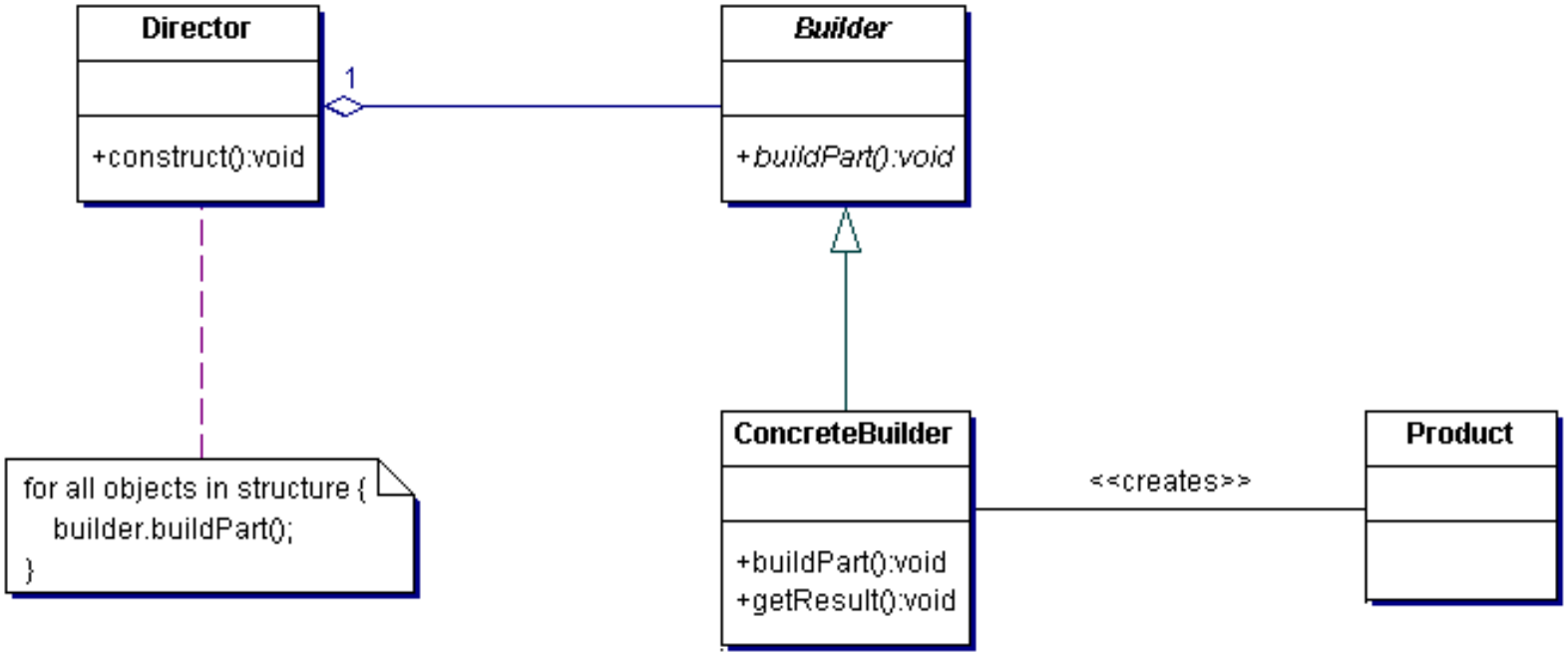
- Possibilita baixo acoplamento entre os objetos dependentes (os observadores) e o assunto.

Builder

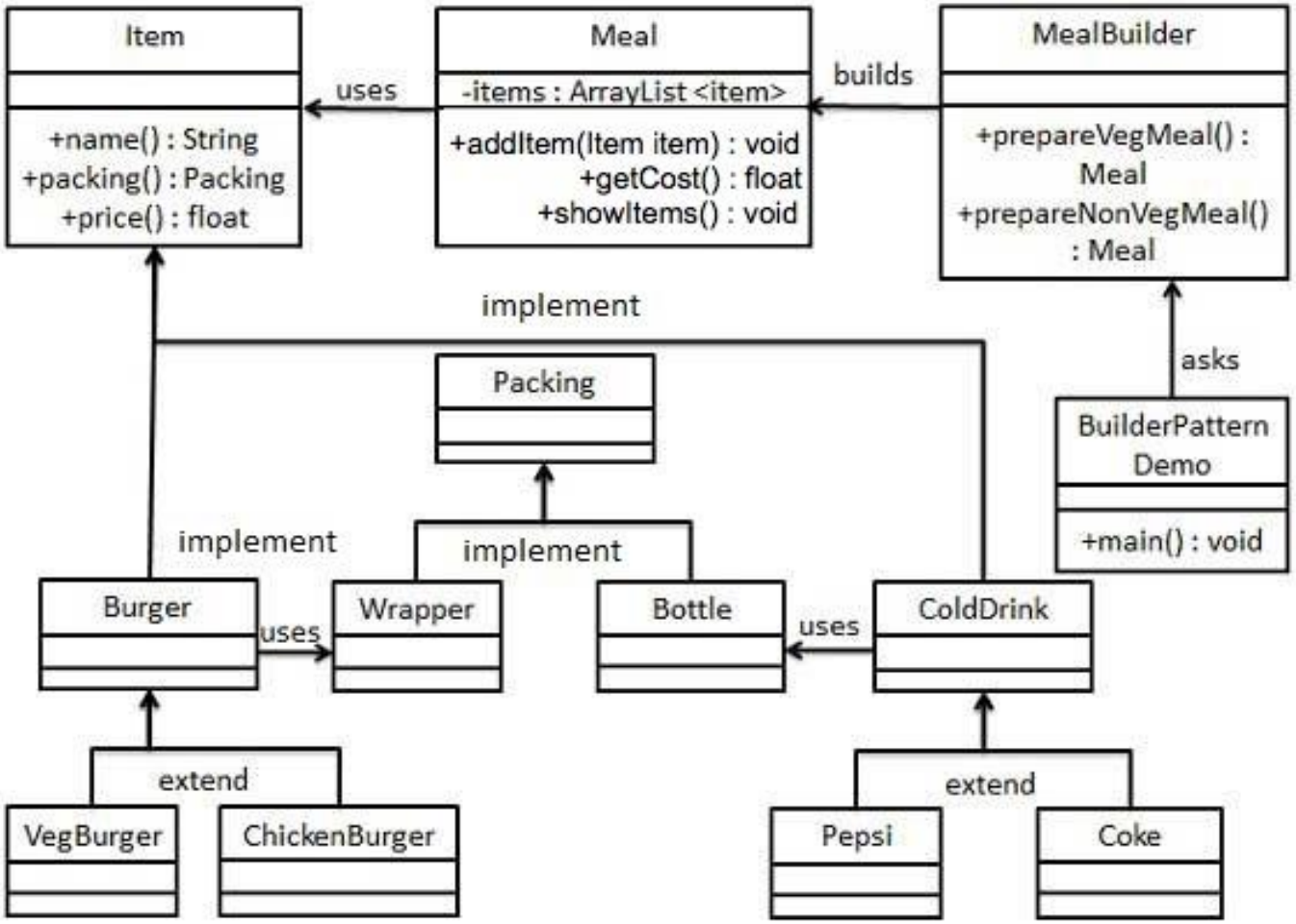
- **Intenção:** separar a construção de objetos complexos da sua representação, de modo que o mesmo processo de construção possa criar diferentes representações.

- A classe Builder constroi os objetos complexos passo a passo.

Builder


Director
+construct():void
1
Builder
+buildPart():void
for all objects in structure {
builder.buildPart();
}
ConcreteBuilder
+buildPart():void
+getResult():void
<<creates>>
Product

Builder – Exemplo



Builder – Implementação

```
public interface Item
{
   public String name();
   public Packing packing();
   public float price();
}
```

```
public interface Packing
{
   public String pack();
}
```

Builder – Implementação

```
public class Wrapper implements Packing
{
   @Override
   public String pack()
   {
      return "Wrapper";
   }
}
```

```
public class Bottle implements Packing
{
   @Override
   public String pack()
   {
      return "Bottle";
   }
}
```

Builder – Implementação

```
public abstract class Burger implements Item {

   @Override
   public Packing packing() {
      return new Wrapper();
   }
   @Override
   public abstract float price();
}
```

```
public abstract class ColdDrink implements Item {

   @Override
   public Packing packing() {
      return new Bottle();
   }
   @Override
   public abstract float price();
}
```

Builder – Implementação

```
public class VegBurger extends Burger {
   @Override
   public float price() {
      return 25.0f;
   }
   @Override
   public String name() {
      return "Veg Burger";
   }
}
```

```
public class ChickenBurger extends Burger {
   @Override
   public float price() {
      return 50.5f;
   }
   @Override
   public String name() {
      return "Chicken Burger";
   }
}
```

Builder – Implementação

```
public class Coke extends ColdDrink {
   @Override
   public float price() {
      return 30.0f;
   }
   @Override
   public String name() {
      return "Coke";
   }
}
```

```
public class Pepsi extends ColdDrink {
   @Override
   public float price() {
      return 35.0f;
   }
   @Override
   public String name() {
      return "Pepsi";
   }
}
```

Builder – Implementação

```
public class Meal {
   private List<Item> items = new ArrayList<Item>();

   public void addItem(Item item){
      items.add(item);
   }
   public float getCost(){
      float cost = 0.0f;
      for (Item item : items) {
         cost += item.price();
      }
      return cost;
   }
   public void showItems(){
      for (Item item : items) {
         System.out.print("Item : " + item.name());
         System.out.print(", Packing : " + item.packing().pack());
         System.out.println(", Price : " + item.price());
      }
   }
}
```

Builder – Implementação

```
public class MealBuilder {

   public Meal prepareVegMeal()
   {
      Meal meal = new Meal();
      meal.addItem(new VegBurger());
      meal.addItem(new Coke());
      return meal;
   }

   public Meal prepareNonVegMeal()
   {
      Meal meal = new Meal();
      meal.addItem(new ChickenBurger());
      meal.addItem(new Pepsi());
      return meal;
   }
}
```

Builder – Implementação

```
public static void main(String[] args)
{

   MealBuilder mealBuilder = new MealBuilder();

   Meal vegMeal = mealBuilder.prepareVegMeal();
   System.out.println("Veg Meal");
   vegMeal.showItems();
   System.out.println("Total Cost: " + vegMeal.getCost());

   Meal nonVegMeal = mealBuilder.prepareNonVegMeal();
   System.out.println("\n\nNon-Veg Meal");
   nonVegMeal.showItems();
   System.out.println("Total Cost: " + nonVegMeal.getCost());
}
```

Builder vs Abstract Factory

- O padrão **Builder** é muitas vezes comparado com o padrão **Abstract Factory** pois ambos podem ser utilizados para a construção de objetos complexos.

- A principal diferença entre eles é que o Builder constrói objetos complexos **passo a passo** e o Abstract Factory constrói **famílias de objetos**, simples ou complexos, de uma só vez.