

# Tecniche di Progettazione: Design Patterns

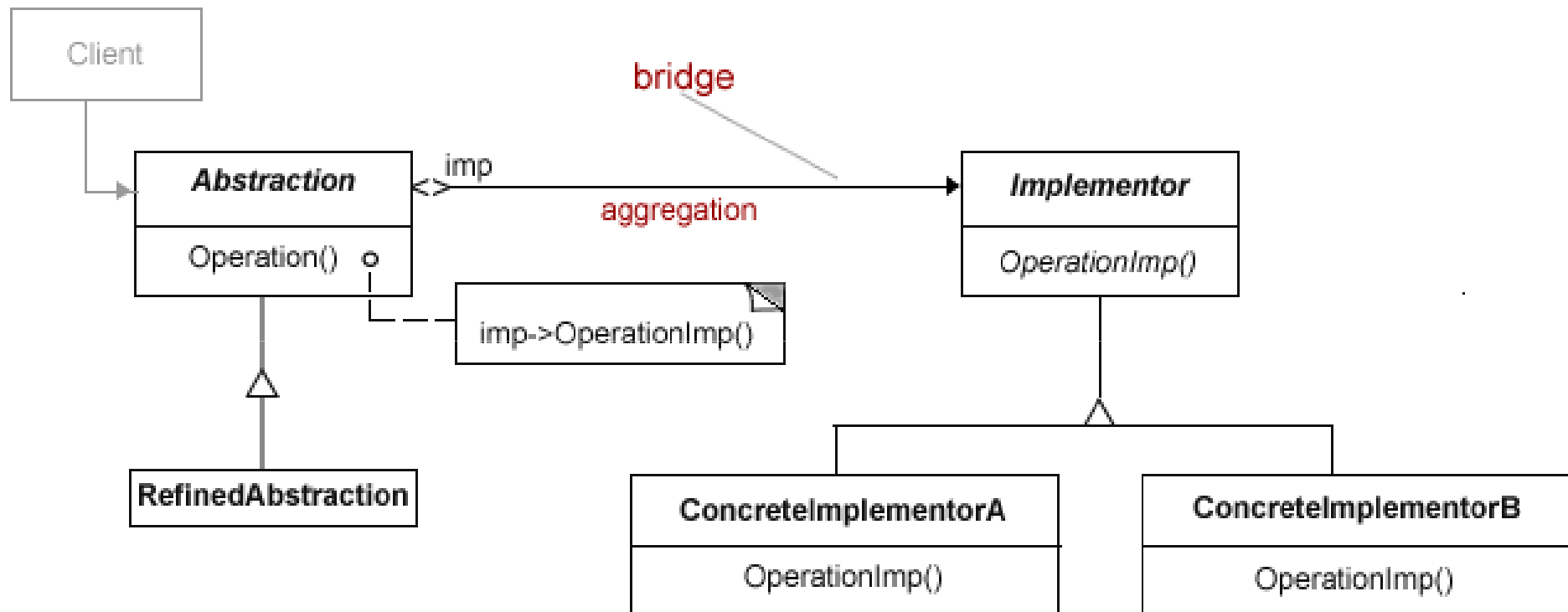
GoF: Bridge

# The Bridge Pattern

---

- ▶ The Bridge Pattern permits to vary the implementation and abstraction by placing the two in separate hierarchies.
- ▶ Decouple an abstraction or interface from its implementation so that the two can vary independently.
- ▶ The bridge uses encapsulation, aggregation, and can use inheritance to separate responsibilities into different classes.

# Pattern structure



# Participants

---

- ▶ **Abstraction**

  - defines the abstract interface
  - maintains the Implementor reference

- ▶ **Refined Abstraction**

  - extends the interface defined by Abstraction

- ▶ **Implementor**

  - defines the interface for implementation classes

- ▶ **ConcreteImplementor**

  - implements the Implementor interface

# Uses and Benefits

---

- ▶ Want to separate abstraction and implementation permanently
- ▶ Share an implementation among multiple objects
- ▶ Want to improve extensibility
- ▶ Hide implementation details from clients

- 
- ▶ Bridge might be a situation where the programmer thought it would be best to isolate the handling of the system-dependent stuff from the handling of the system-independent stuff.

The collections class framework in the Java API provides several examples of use of the bridge pattern. Both the `ArrayList` and `LinkedList` concrete classes implement the `List` interface. The `List` interface provides common, abstract concepts, such as the abilities to add to a list and to ask for its size. The implementation details vary between `ArrayList` and `LinkedList`, mostly with respect to when memory is allocated for elements in the list.

# First, we have our TV implementation interface

---

```
//Implementor
```

```
public interface TV {  
    public void on();  
    public void off();  
    public void tuneChannel(int channel);  
}
```

# And then we create two specific implementations.

---

```
//Concrete Implementor
public class Sony implements TV{
    public void on(){
        //Sony specific on
    }
    public void off(){
        //Sony specific off
    }
    public void tuneChannel(int
channel) {
        //Sony specific tuneChannel
    }
}
```

```
//Concrete Implementor
public class Philips implements TV{
    public void on(){
        // Philips specific on
    }
    public void off(){
        // Philips specific off
    }
    public void tuneChannel(int
channel) {
        // Philips specific tuneChannel
    }
}
```



Now, we create a remote control abstraction to control the TV

---

//Abstraction

```
public abstract class RemoteControl {  
    private TV implementor;  
    public void on() { implementor.on(); }  
    public void off() { implementor.off(); }  
    public void setChannel(int channel) {  
        implementor.tuneChannel(channel); }  
}
```

But what if we want a more specific remote control - one that has the + / - buttons for moving through the channels?

---

//Refined abstraction

```
public class ConcreteRemote extends RemoteControl {
    private int currentChannel;
    public void nextChannel() {
        currentChannel++;
        setChannel(currentChannel); }
    public void prevChannel() {
        currentChannel--;
        setChannel(currentChannel); }
}
```

# Bridge vs Strategy

## Bridge vs Adapter

---

- ▶ Often, the Strategy Pattern is confused with the Bridge Pattern. Even though, these two patterns are similar in structure, they are trying to solve two different design problems. Strategy is mainly concerned in encapsulating algorithms, whereas Bridge decouples the abstraction from the implementation, to provide different implementation for the same abstraction.
- ▶ The structure of the Adapter Pattern (object adapter) may look similar to the Bridge Pattern. However, the adapter is meant to change the interface of an existing object and is mainly intended to make unrelated classes work together.