Tecniche di Progettazione: Design Patterns

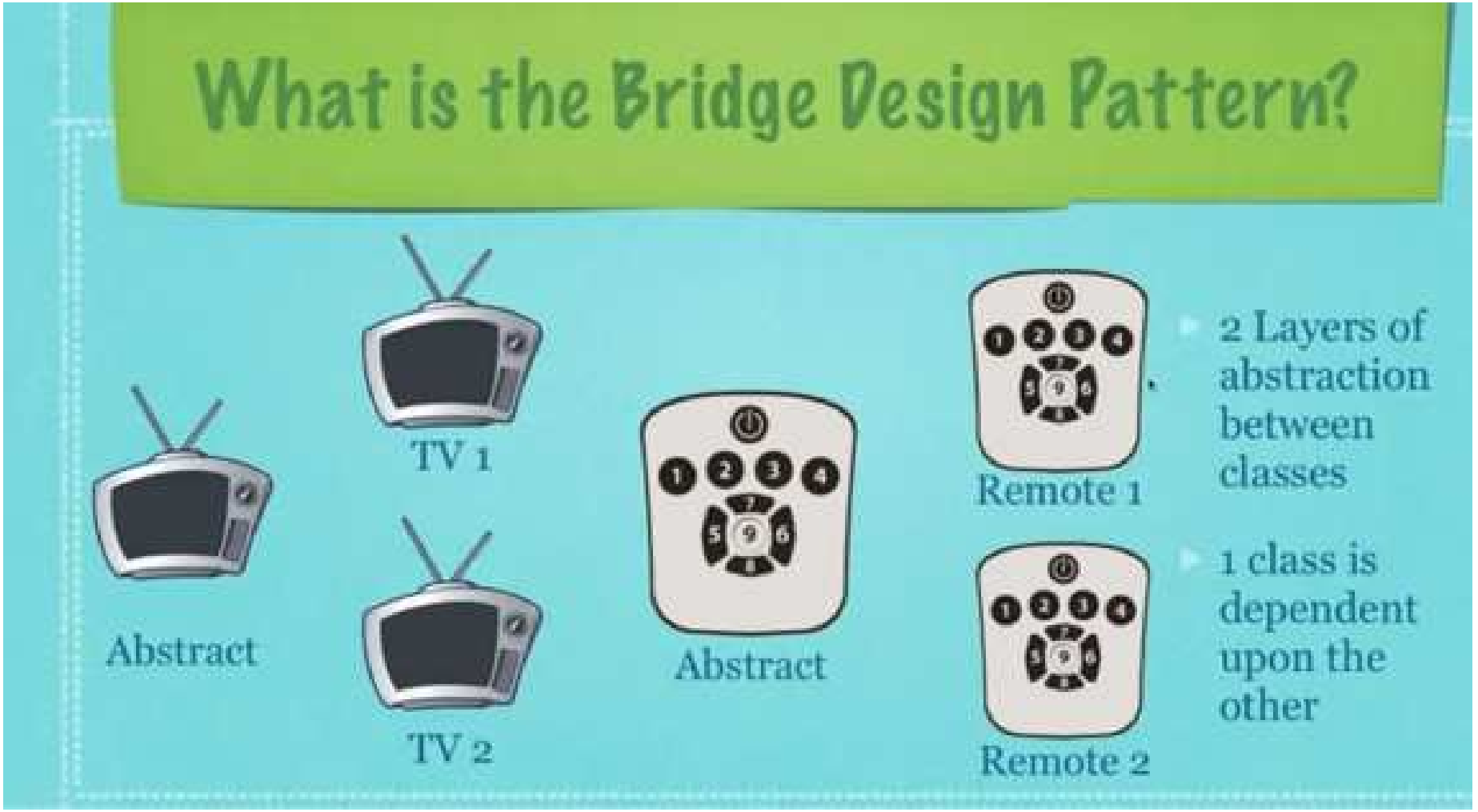
GoF: Bridge

The Bridge Pattern

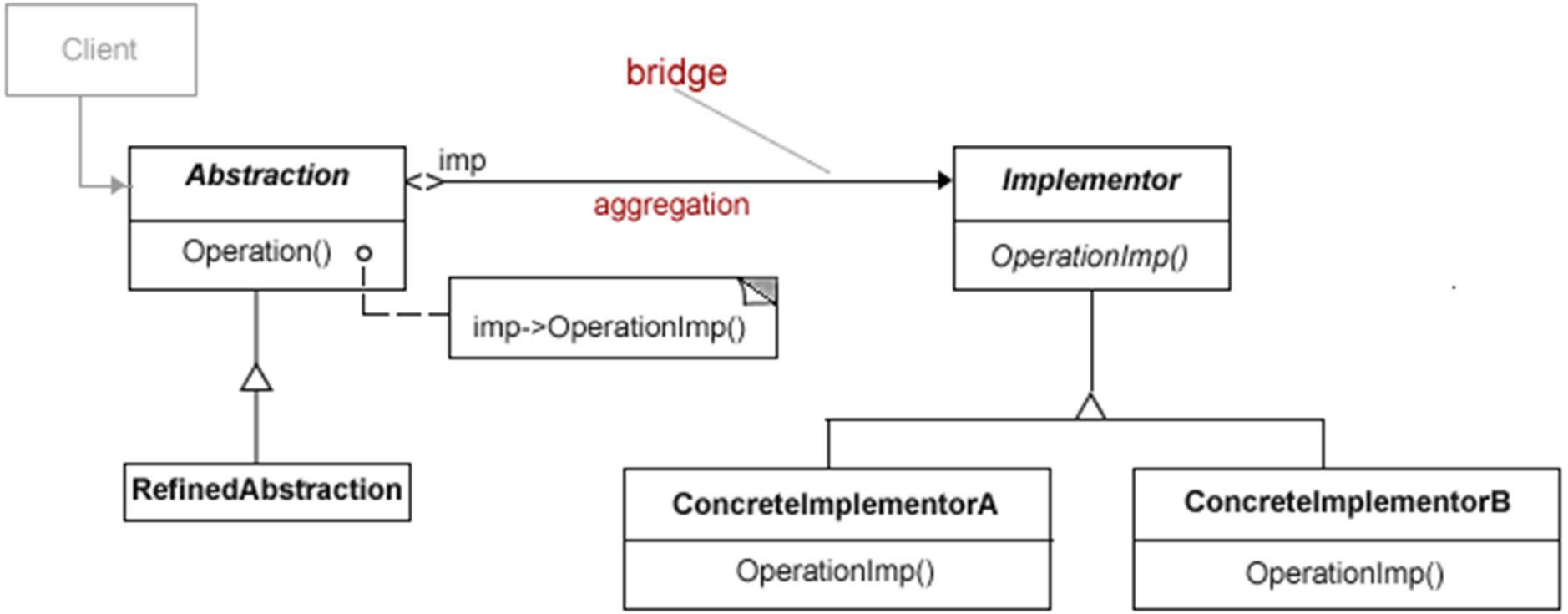
- The Bridge Pattern permits to vary the implementation and abstraction by placing the two in seperate hierarchies.

- Decouple an abstraction or interface from its implementation so that the two can vary independently.

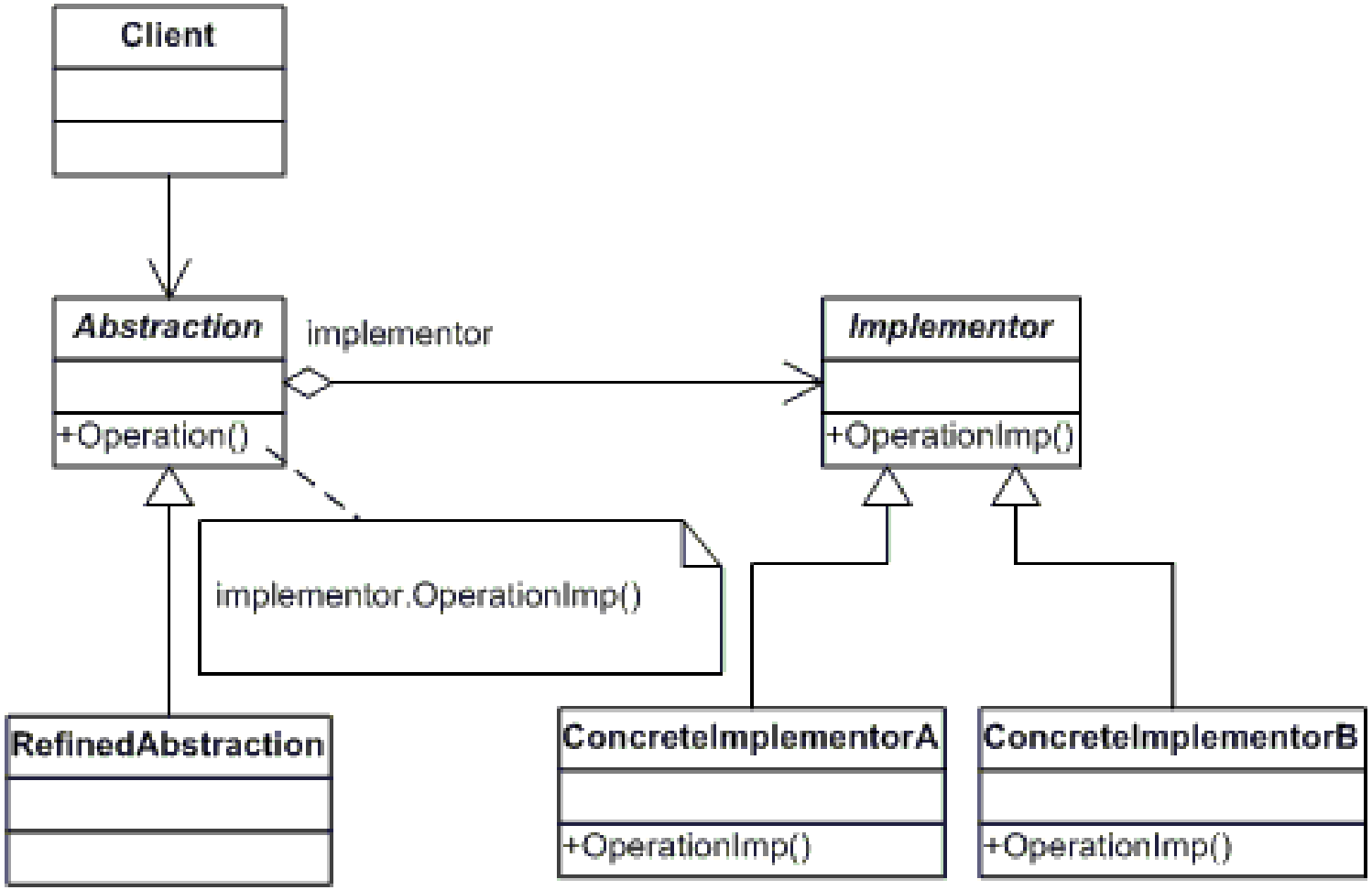
- The bridge uses encapsulation, aggregation, and can use inheritance to separate responsibilities into different classes.


What is the Bridge Design Pattern?
TV 1
Abstract
TV 2
Abstract
Remote 1
Remote 2
2 Layers of abstraction between classes
1 class is dependent upon the other

Pattern structure



UML


Client
Abstraction
implementor
Implementor
+Operation()
+OperationImp()
implementor.OperationImp()
RefinedAbstraction
ConcreteImplementorA
+OperationImp()
ConcreteImplementorB
+OperationImp()

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

Discussion

- 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
    }
}
```

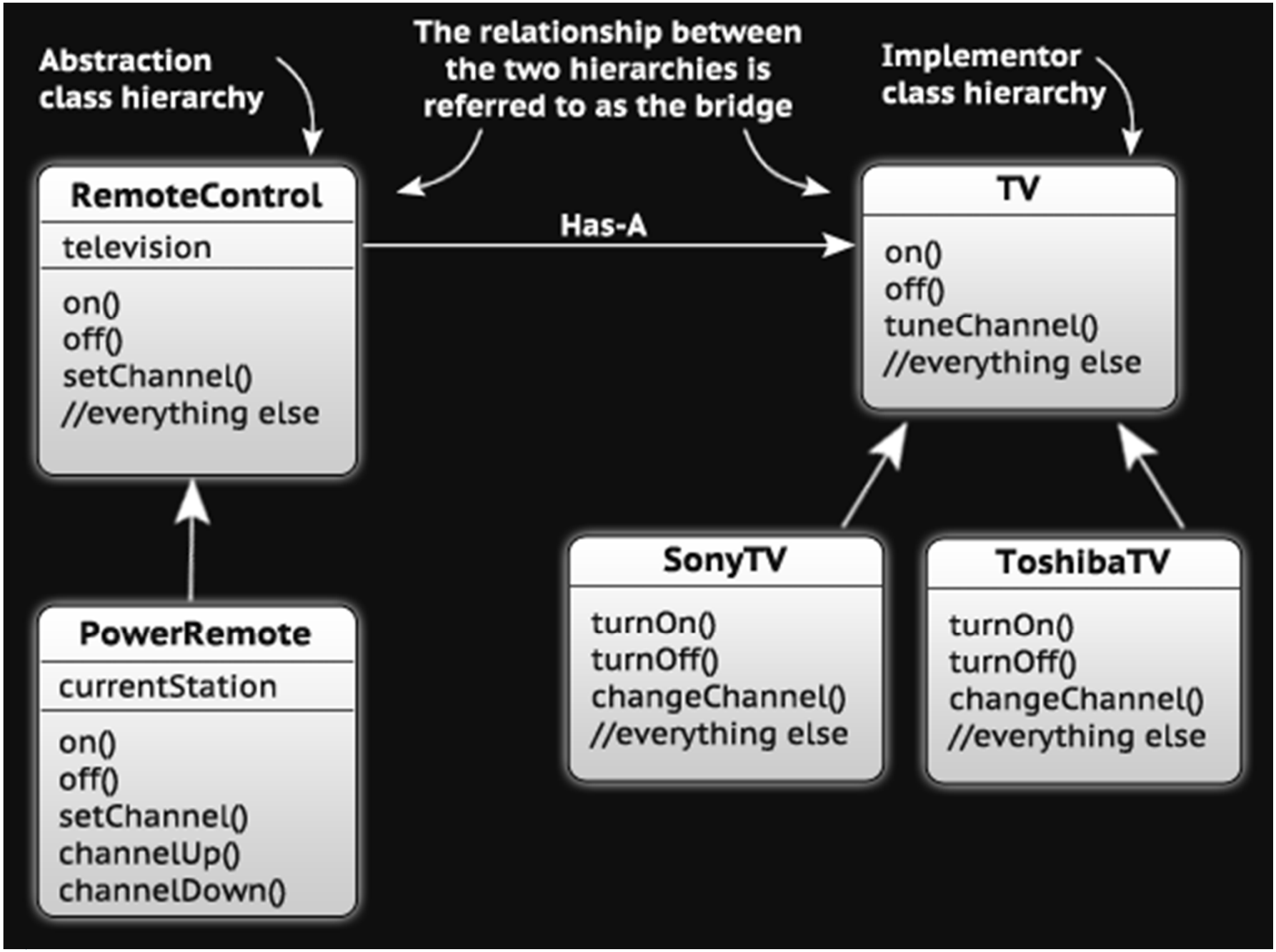
Now, we create a remote control abstraction to control the TV

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

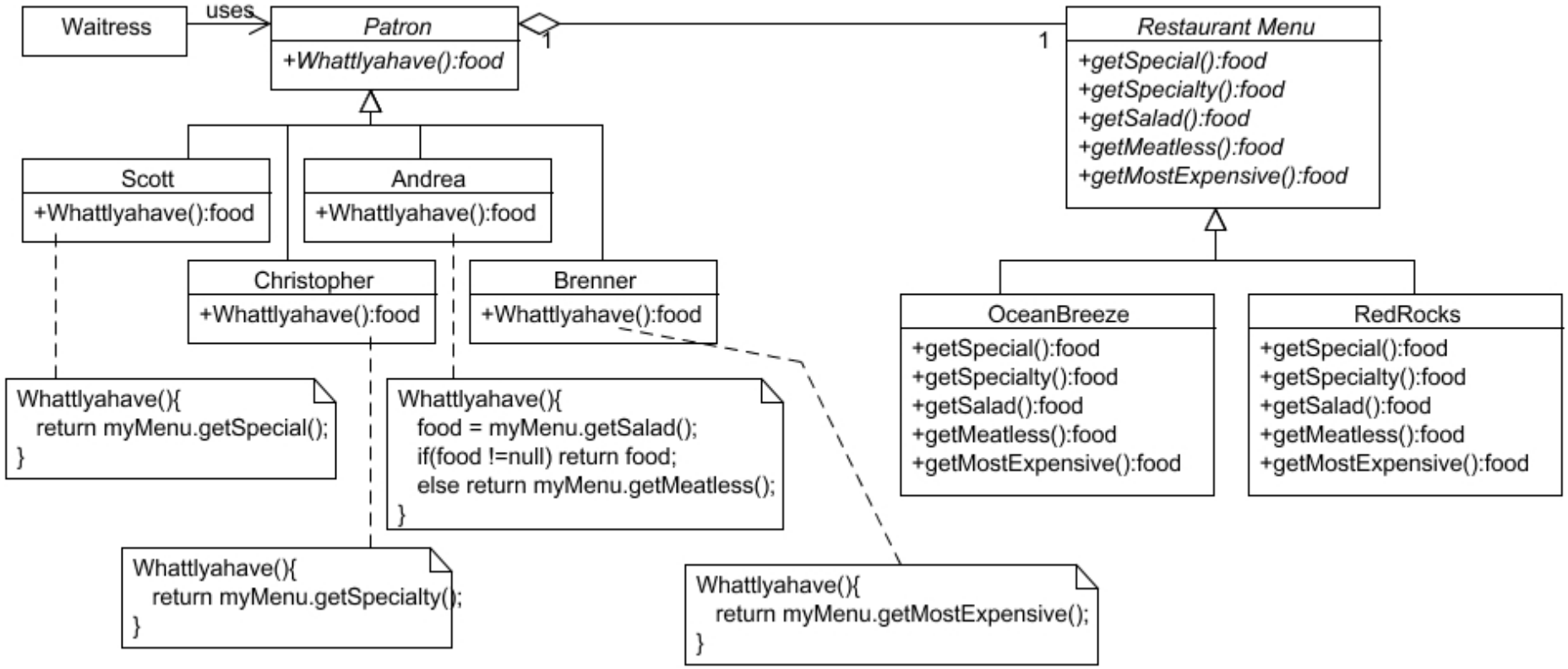
But what if we want a more specific remote control - one that has the + / - buttons for moving through the channels?

```
//Refined abstraction
public class ConcreteRemote extends RemoteControl { private
   int currentChannel;
   public void nextChannel() {
         currentChannel++;
         super.setChannel(currentChannel); }
public void prevChannel() {
         currentChannel--;
         super.setChannel(currentChannel); }
public void setChannel(int channel) {
         super.setChannel(channel);
         currentChannel=channel;}
}
```

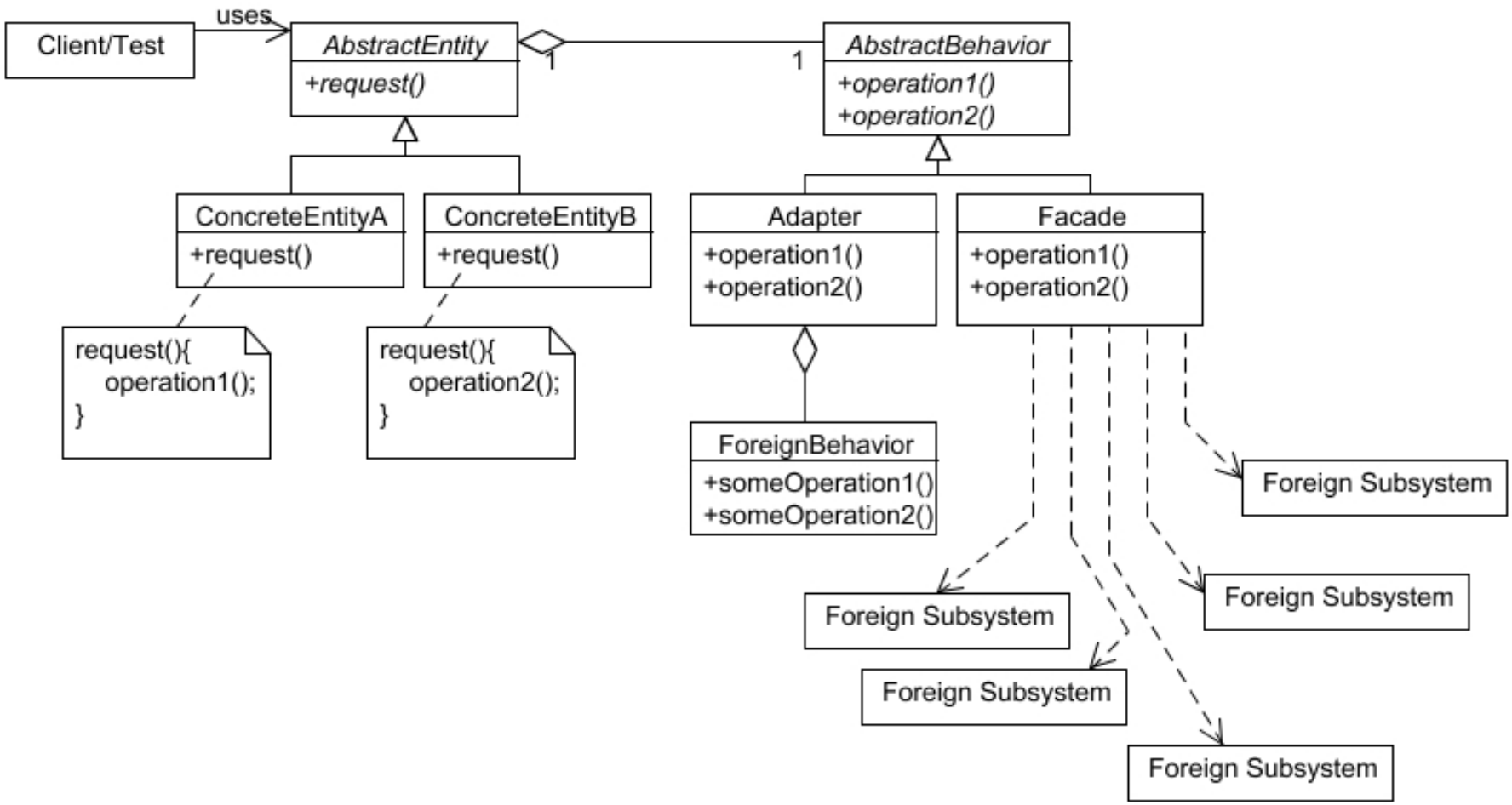
nextChannel definito chiamando metodi dell'astrazione, non dell'implementazione


Abstraction class hierarchy
The relationship between the two hierarchies is referred to as the bridge
Implementor class hierarchy
RemoteControl
television
on()
off()
setChannel()
//everything else
Has-A
TV
on()
off()
tuneChannel()
//everything else
PowerRemote
currentStation
on()
off()
setChannel()
channelUp()
channelDown()
SonyTV
turnOn()
turnOff()
changeChannel()
//everything else
ToshibaTV
turnOn()
turnOff()
changeChannel()
//everything else

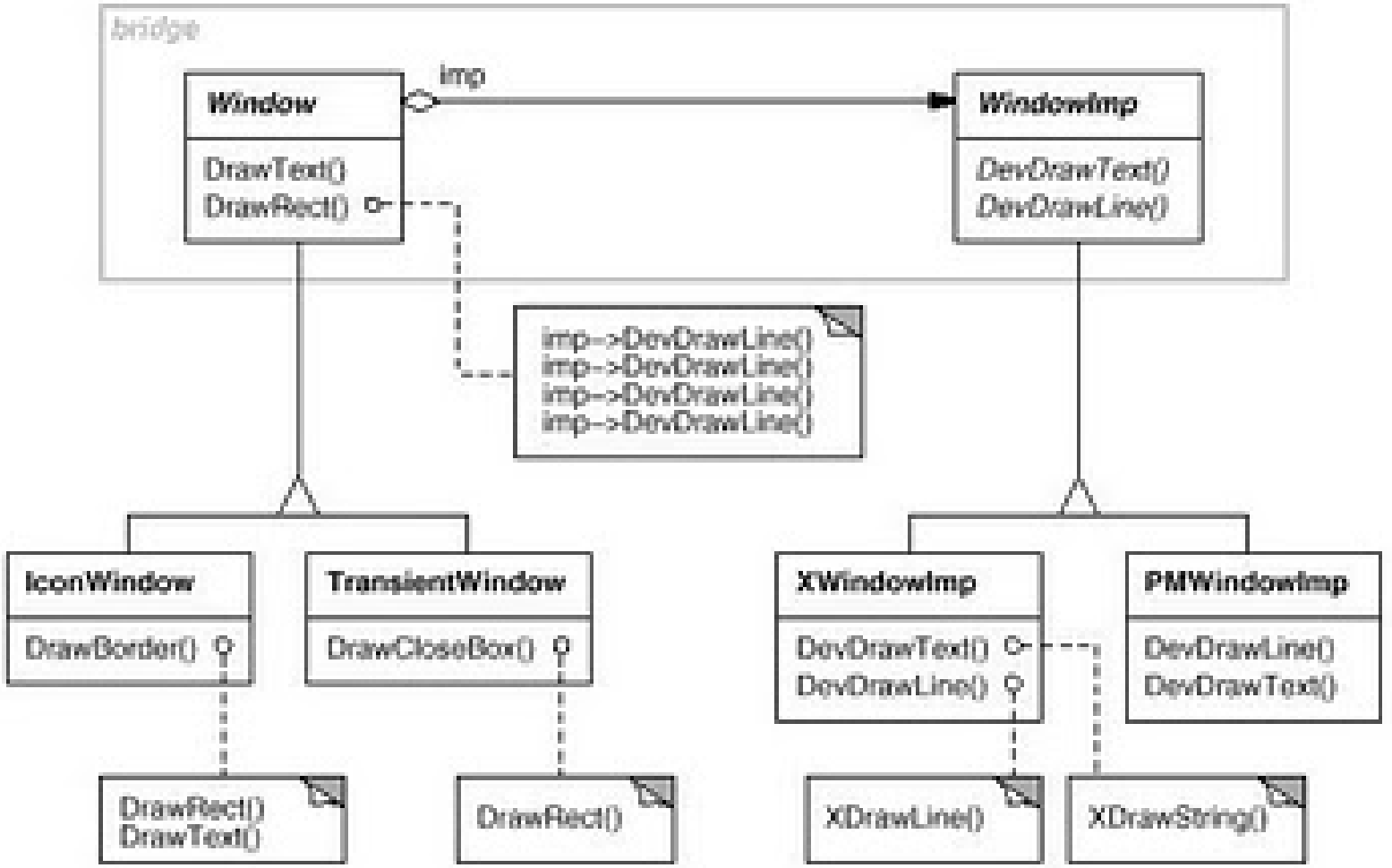
Example


Waitress
uses
Patron
+Whattlyahave():food
1
1
Restaurant Menu
+getSpecial():food
+getSpecialty():food
+getSalad():food
+getMeatless():food
+getMostExpensive():food
Scott
+Whattlyahave():food
Andrea
+Whattlyahave():food
Christopher
+Whattlyahave():food
Brenner
+Whattlyahave():food
OceanBreeze
+getSpecial():food
+getSpecialty():food
+getSalad():food
+getMeatless():food
+getMostExpensive():food
RedRocks
+getSpecial():food
+getSpecialty():food
+getSalad():food
+getMeatless():food
+getMostExpensive():food
Whattlyahave(){
return myMenu.getSpecial();
}
Whattlyahave(){
food = myMenu.getSalad();
if(food !=null) return food;
else return myMenu.getMeatless();
}
Whattlyahave(){
return myMenu.getSpecialty();
}
Whattlyahave(){
return myMenu.getMostExpensive();
}

Example


Client/Test
uses
AbstractEntity
+request()
1
1
AbstractBehavior
+operation1()
+operation2()
ConcreteEntityA
+request()
ConcreteEntityB
+request()
Adapter
+operation1()
+operation2()
Facade
+operation1()
+operation2()
request(){
operation1();
}
request(){
operation2();
}
ForeignBehavior
+someOperation1()
+someOperation2()
Foreign Subsystem
Foreign Subsystem
Foreign Subsystem
Foreign Subsystem
Foreign Subsystem

Example


bridge
Window
DrawText()
DrawRect()
imp
WindowImp
DevDrawText()
DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
IconWindow
DrawBorder()
TransientWindow
DrawCloseBox()
XWindowImp
DevDrawText()
DevDrawLine()
PMWindowImp
DevDrawLine()
DevDrawText()
DrawRect()
DrawText()
DrawRect()
XDrawLine()
XDrawString()

Bridge vs Strategy

- 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.

Bridge vs Adapter

- 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.