OMO

5 - Structura design patterns

- Adapter, Proxy, Facade
- Decorator
- Bridge
- Flyweight

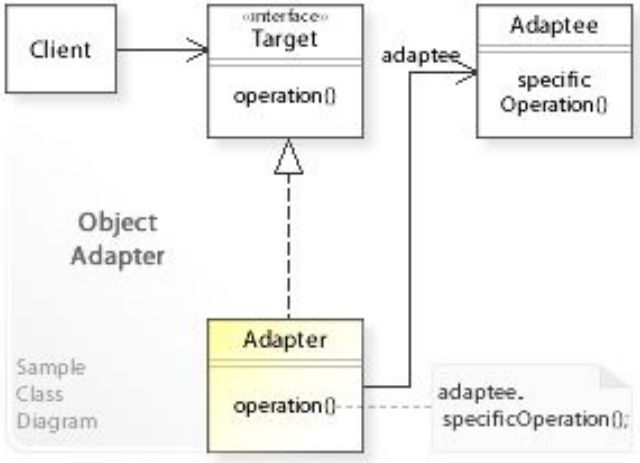
Ing. David Kadleček, PhD.
kadlecd@fel.cvut.cz, david.kadlecek@cz.ibm.com

Adapter

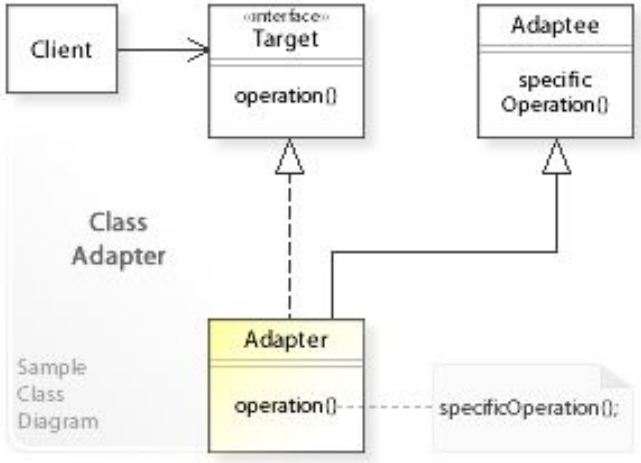
Potřebujeme propojit dvě komponenty, které mají rozdílné rozhraní

- Aplikace, která používá volání přes již definované rozhraní
- Potřebujeme zapojit novou komponentu, která má odlišné rozhraní
- Vytvoříme třídu (adaptér), která vystavuje původní rozhraní a provádí převolání do nové knihovny přes její rozhraní

Připojení adaptéru v run time



Připojení adaptéru v compile time

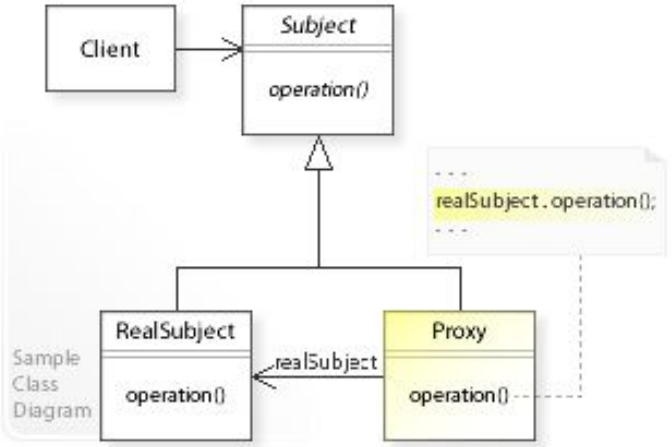


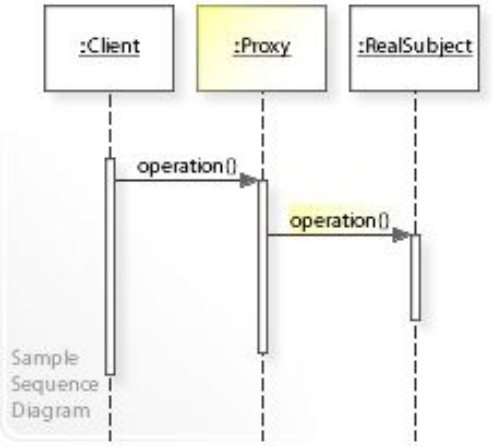
Proxy

Potřebujeme obalit volání komponenty nějakou (obvykle sdílenou) funkcionalitou aniž bychom museli modifikovat původní objekt. Proxy má stejné rozhraní jako objekt, který proxuje. Říkáme tomu i jinak aspekt.

- Umožňuje zastupovat objekt, aby bylo možné například řídit přístup k původnímu objektu
- Proxy objekt má stejný interface jako původní objekt
- Proxy dodává další logiku do volání původní služby

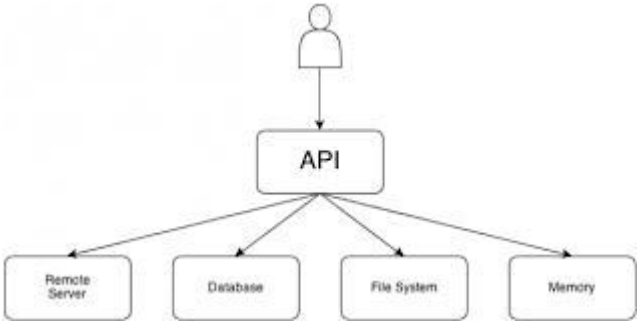
Příklady:
Logování - při jakékoliv volání metody chci toto volání zalogovat
Security - při jakémkoliv volání metody, chci ověřit zda má volající právo provolat metodu

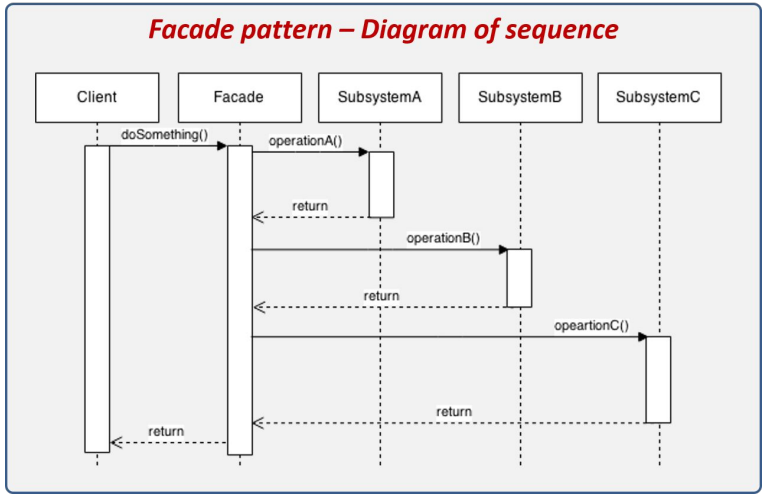




Facade

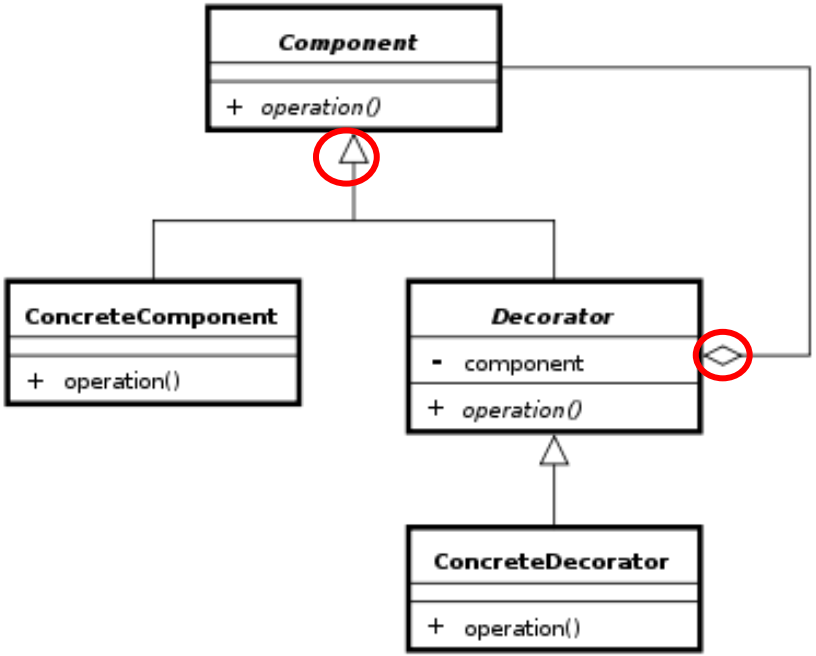
Potřebujeme skrýt komplexitu svého programu za tzv. fasádu. Jedná se o princip abstrakce, kdy z našeho systému vystavíme API, kterým zprostředkováváme pouze funkcionalitu, kterou potřebuje klient a ještě způsobem, aby používání pro něj bylo co nejjednodušší a nejstabilnější. Např. aby nemusel při každé úpravě našeho programu opravovat svůj kód.

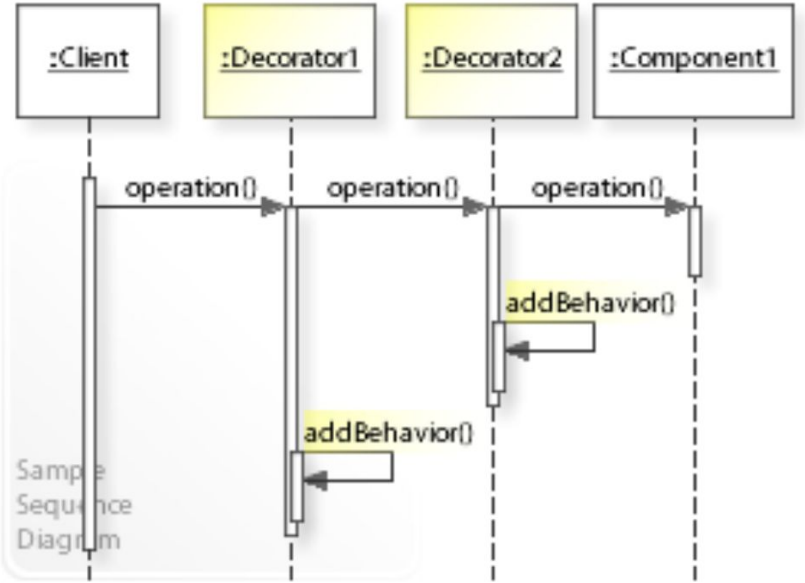




Decorator

Přidání funkcionality do existujícího objektu bez jeho modifikace





```java
public interface Shape {
   void draw();
}

public class Rectangle implements Shape {
   public void draw() {
      ... vykreslení boxu
   }
}

public class Circle implements Shape {
   public void draw() {
      ... vykreslení kruhu
   }
}

public abstract class ShapeDecorator implements
Shape {
   protected Shape decoratedShape;
   public ShapeDecorator(Shape decoratedShape){
      this.decoratedShape = decoratedShape;
   }
   public void draw(){
      decoratedShape.draw();
   }
}
```

```java
public class RedFillDecorator extends ShapeDecorator {
  public RedFillDecorator(Shape decoratedShape) {
      super(decoratedShape);
   }
   public void draw() {
      decoratedShape.draw();
      setRedBorder(decoratedShape);
   }
   private void fillRed(Shape decoratedShape){
      ... vybarvení červeně
   }
}

public class LabelDecorator extends ShapeDecorator {
  private String label;
  public LabelDecorator(Shape decoratedShape, String lbl) {
      super(decoratedShape);
   }
   public void draw() {
      decoratedShape.draw();
      setLabel(decoratedShape);
   }
    public void setLabel(Shape decoratedShape) {
         ...nastavení názvu na this.label
   }
}
```

Decorator

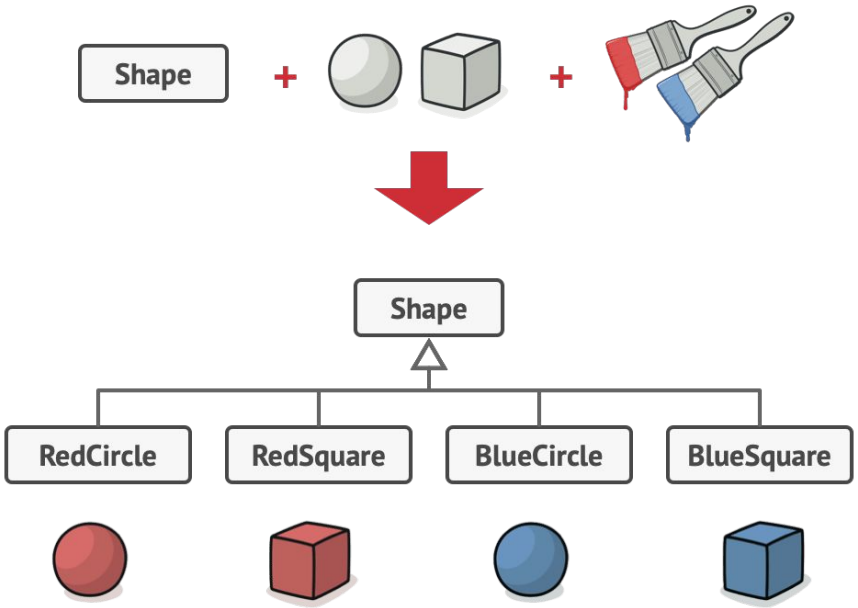
```
public class DecoratorPatternDemo {
   public static void main(String[] args) {

      Shape redCircle = new RedShapeDecorator(new Circle());

      Shape redRectangle = new RedShapeDecorator(new Rectangle());

      Shape redLabeledRectangle = new LabelDecorator(redRectangle, “Vymazlenej čtverec”);

      redCircle.draw();
      redLabeledRectangle.draw();

   }
}
```

Bridge

Mám problém, kde potřebuji namodelovat strukturu, která má více - navzájem ortogonálních vlastností. Mám variantu, že z root třídy budu dědit podtřídy, které odpovídají všem kombinacím vlastností nebo do root třídy připojím druhou hiearachii tříd.

Výhoda je více zřejmá, kdyby v hierarchiích nebyly dvojice tříd, ale např. tři (*Circle, Square, Dot*) a nebo že bych měl např. tři ortogonální hierarchie (*Shape, Color, Sound*)

Implementace jednotlivých hierarchií jsou oddělené, takže se o jejich implementaci a rozvoj mou starat různé týmy



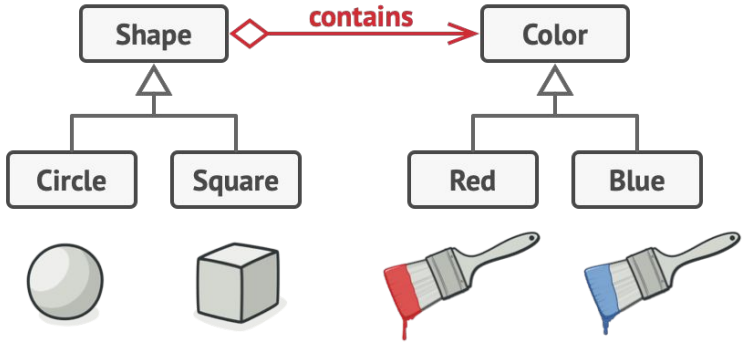
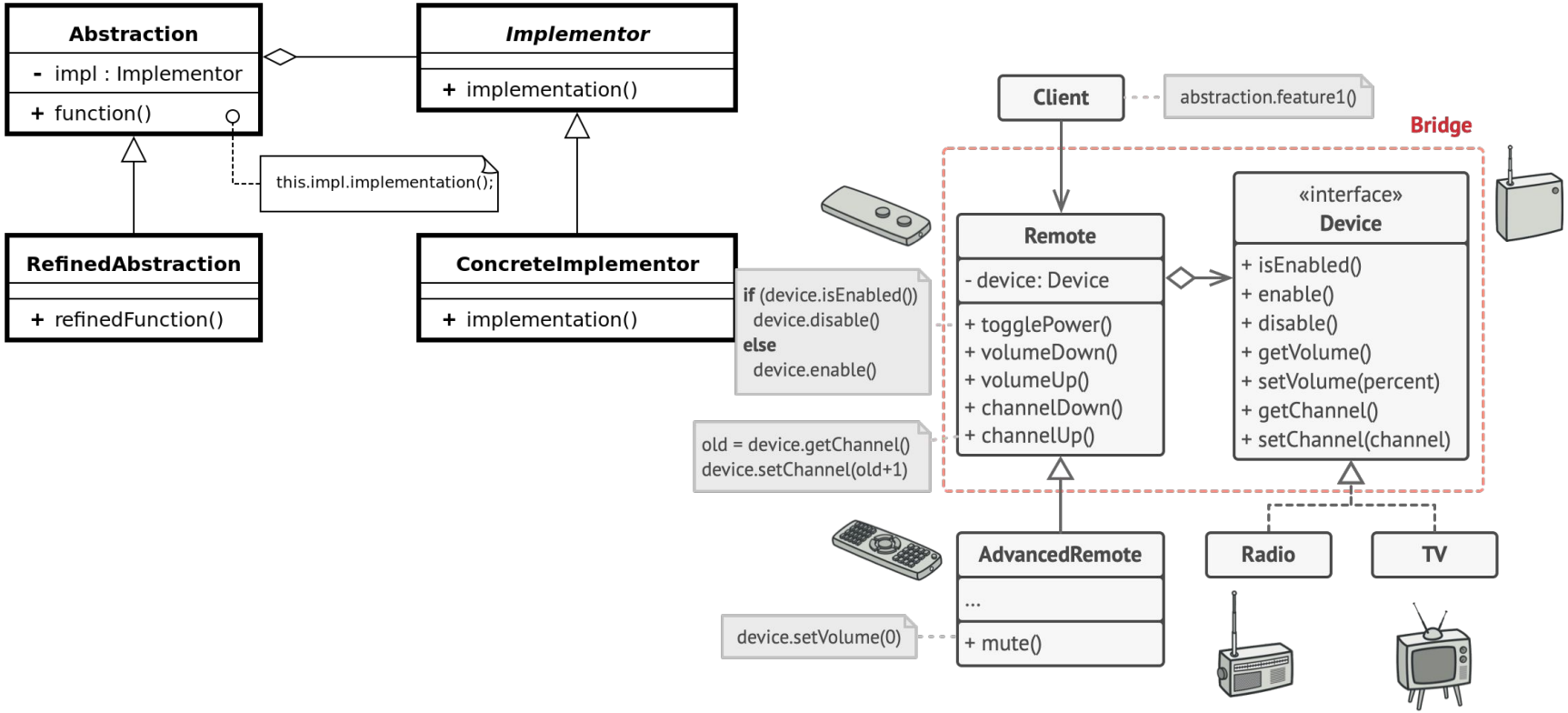


Image source: refactoring.guru

Bridge


Abstraction
- impl : Implementor
+ function()
Implementor
+ implementation()
this.impl.implementation();
RefinedAbstraction
+ refinedFunction()
ConcreteImplementor
+ implementation()
Client
abstraction.feature1()
Bridge
Remote
- device: Device
+ togglePower()
+ volumeDown()
+ volumeUp()
+ channelDown()
+ channelUp()
«interface»
Device
+ isEnabled()
+ enable()
+ disable()
+ getVolume()
+ setVolume(percent)
+ getChannel()
+ setChannel(channel)
if (device.isEnabled())
device.disable()
else
device.enable()
old = device.getChannel()
device.setChannel(old+1)
AdvancedRemote
...
+ mute()
device.setVolume(0)
Radio
TV

Vzájemné vztahy

- Adapter poskytuje rozdílný interface oproti objektu, který zpřístupňuje, Proxy proskytuje shodný interface
- Facade a Proxy
  - podobné v tom, že inicializují a poskytují odstínění klienta od komplexních využívaných komplexních objektů
  - Rozdíl v tom, že **PROXY** poskytuje shodný interface jako servisní objekt
- Decorator a Proxy
  - Podobná struktura, ale rozdílný účel
    - Proxy spravuje životní cyklus servisního objektu
    - Struktura Decoratoru je řízena klientem
- Facade definuje nové rozhraní, Adapter spojuje existující rozhraní