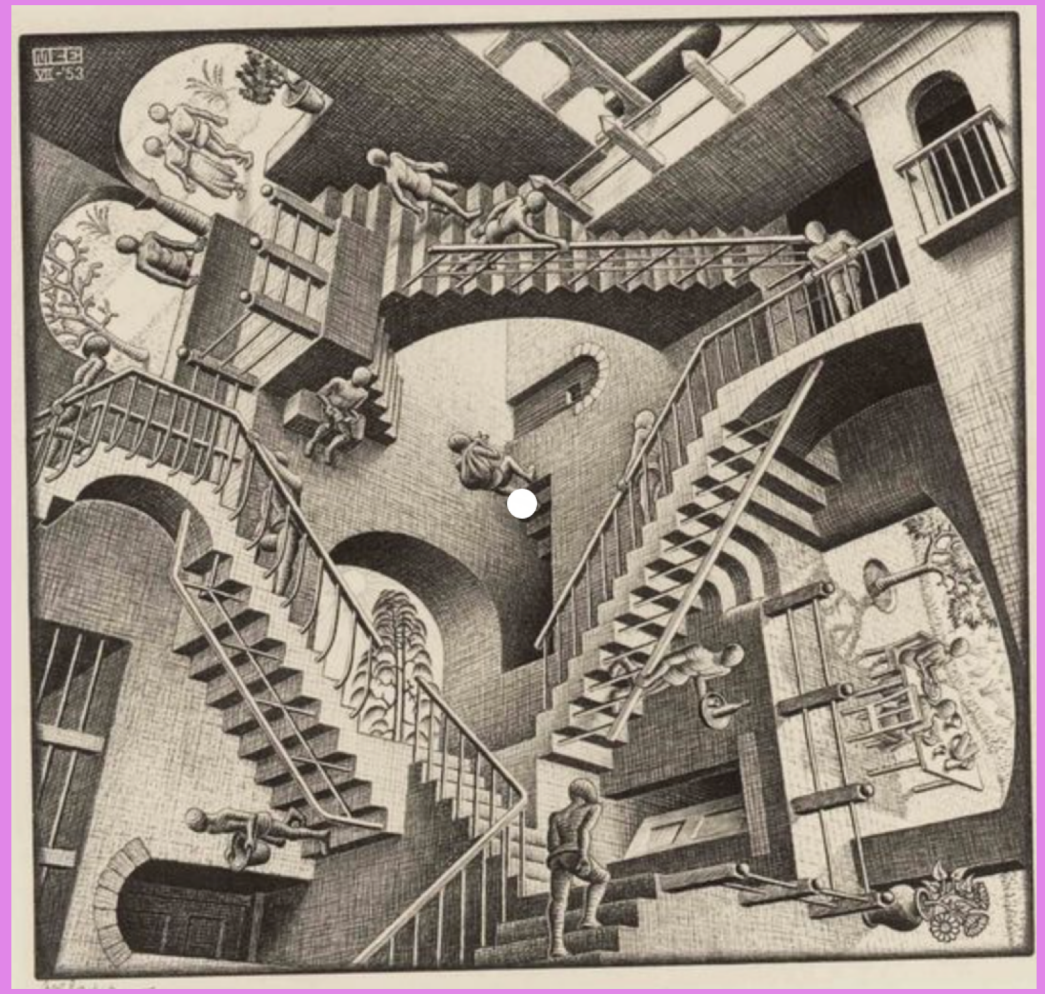


07

Strukturální design patterny

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

Next >

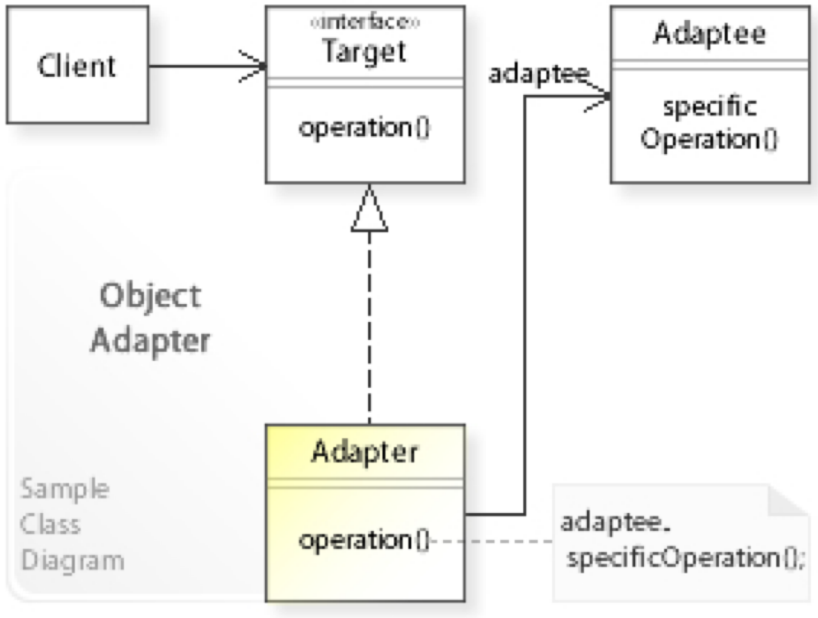
Objektové Modelování - podzim 2022

david.kadlecek@cvut.cz, www.czm.fel.cvut.cz

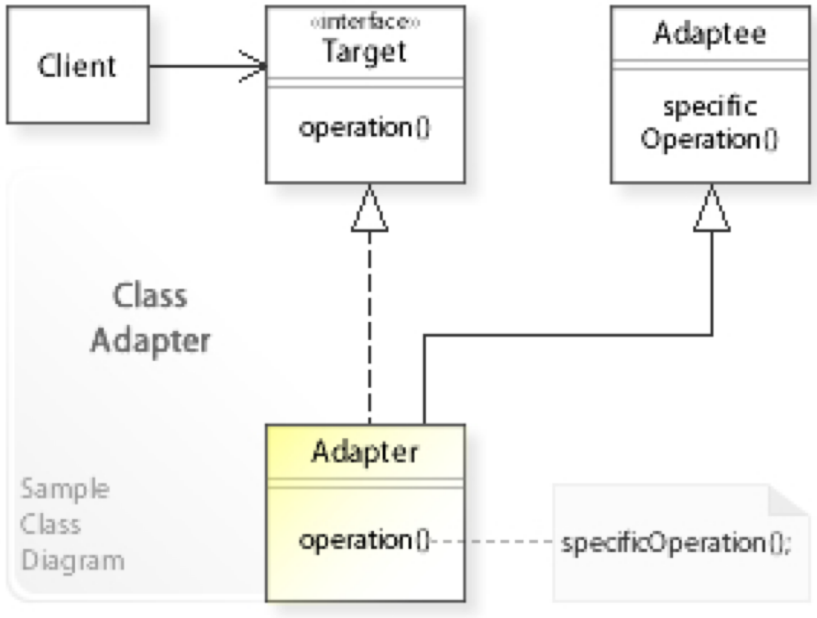
07 ADAPTÉR

- 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 přes její rozhraní

Připojení adaptéru v run time



Připojení adaptéru v compile time

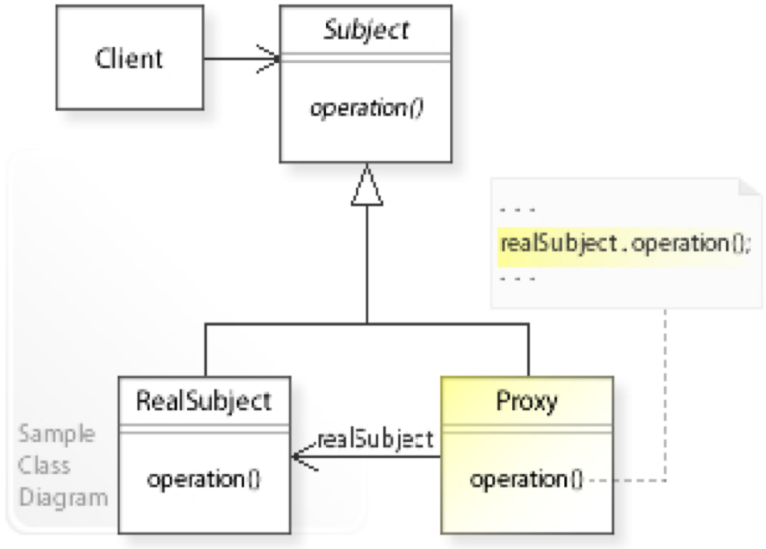


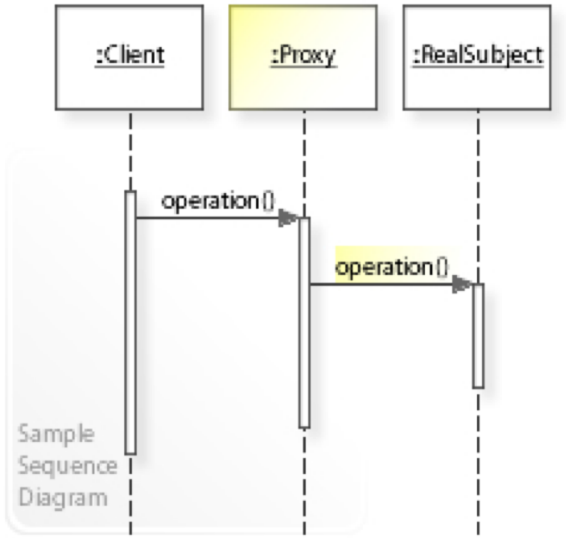
07 PROXY

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





07 PROXY

```java
public interface WizardTower {

  void enter(Wizard wizard);
}

@Slf4j
public class IvoryTower implements WizardTower {

  public void enter(Wizard wizard) {
    LOGGER.info("{} enters the tower.", wizard);
  }
}
```

```java
public class Wizard {

  private final String name;

  public Wizard(String name) {
    this.name = name;
  }

  @Override
  public String toString() {
    return name;
  }
}
```

07 PROXY

```java
@Slf4j
public class WizardTowerProxy implements WizardTower {

  private static final int NUM_WIZARDS_ALLOWED = 3;

  private int numWizards;

  private final WizardTower tower;

  public WizardTowerProxy(WizardTower tower) {
    this.tower = tower;
  }

  @Override
  public void enter(Wizard wizard) {
    if (numWizards < NUM_WIZARDS_ALLOWED) {
      tower.enter(wizard);
      numWizards++;
    } else {
      LOGGER.info("{} is not allowed to enter!", wizard);
    }
  }
}
```

```java
var proxy = new WizardTowerProxy(new IvoryTower());
proxy.enter(new Wizard("Red wizard"));
proxy.enter(new Wizard("White wizard"));
proxy.enter(new Wizard("Black wizard"));
proxy.enter(new Wizard("Green wizard"));
proxy.enter(new Wizard("Brown wizard"));
```

=>

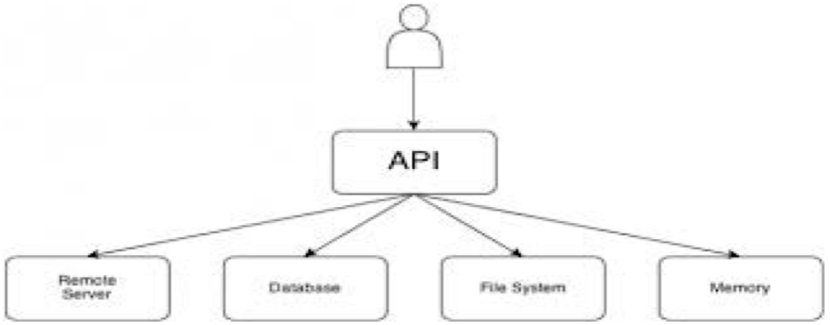
Red wizard enters the tower.
White wizard enters the tower.
Black wizard enters the tower.
Green wizard is not allowed to enter!
Brown wizard is not allowed to enter!

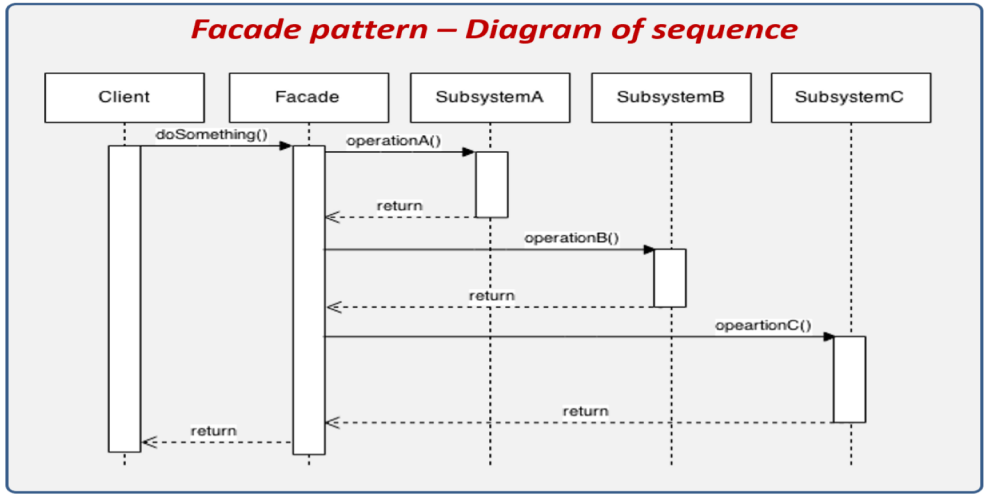
07 FACADE

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.

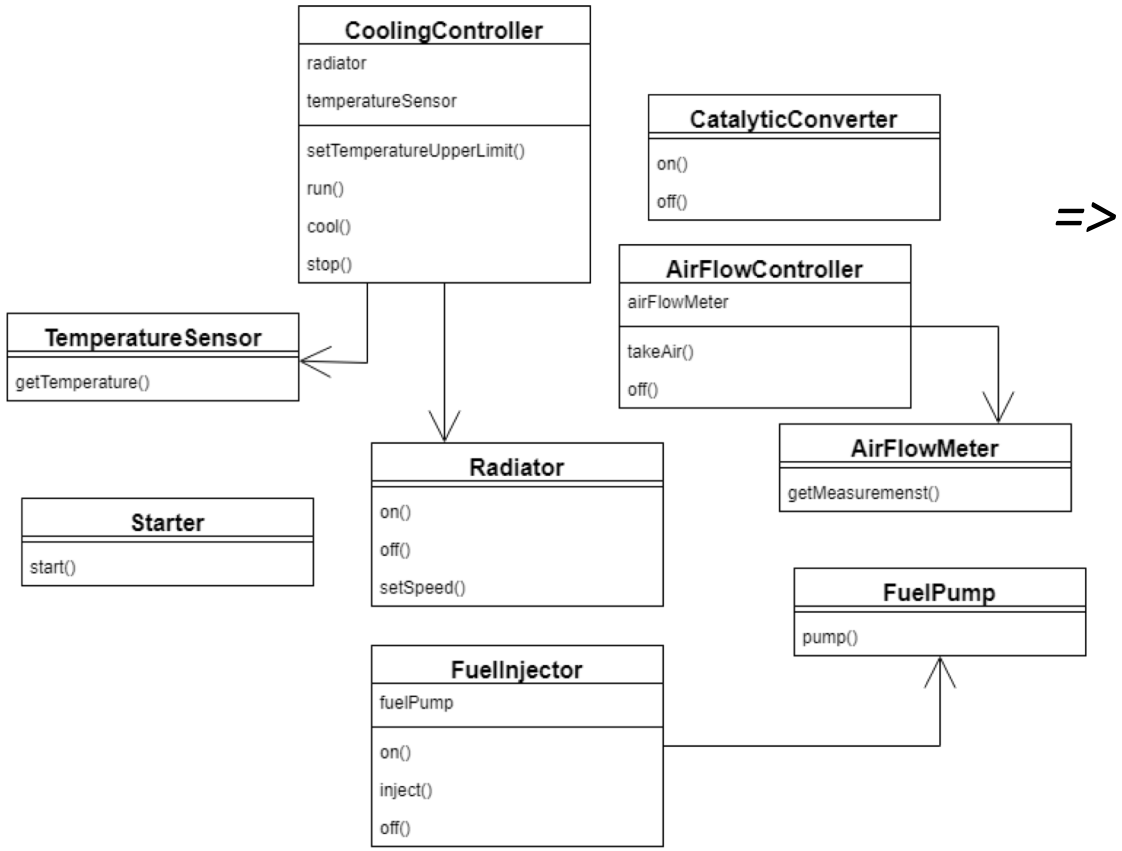
Tedy konzument nemusí:

- opravovat svůj kód při každé úpravě našeho program
- Rozumět našemu kódu





07 FACADE



=>

```
//Start engine
airFlowController.takeAir()
fuelInjector.on()
fuelInjector.inject()
starter.start()
coolingController.setTemperatureUpperLimit(DEFAULT_COOLING_TEMP)
coolingController.run()
catalyticConverter.on()

//Stop engine
fuelInjector.off()
catalyticConverter.off()
coolingController.cool(MAX_ALLOWED_TEMP)
coolingController.stop()
airFlowController.off()
```

07 FACADE

```java
public class CarEngineFacade {
    private static int DEFAULT_COOLING_TEMP = 90;
    private static int MAX_ALLOWED_TEMP = 50;
    private FuelInjector fuelInjector = new FuelInjector();
    private AirFlowController airFlowController = new AirFlowController();
    private Starter starter = new Starter();
    private CoolingController coolingController = new CoolingController();
    private CatalyticConverter catalyticConverter = new CatalyticConverter();

    public void startEngine() {
        fuelInjector.on();
        airFlowController.takeAir();
        fuelInjector.on();
        fuelInjector.inject();
        starter.start();
        coolingController.setTemperatureUpperLimit(DEFAULT_COOLING_TEMP);
        coolingController.run();
        catalyticConverter.on();
    }

    public void stopEngine() {
        fuelInjector.off();
        catalyticConverter.off();
        coolingController.cool(MAX_ALLOWED_TEMP);
        coolingController.stop();
        airFlowController.off();
    }
```

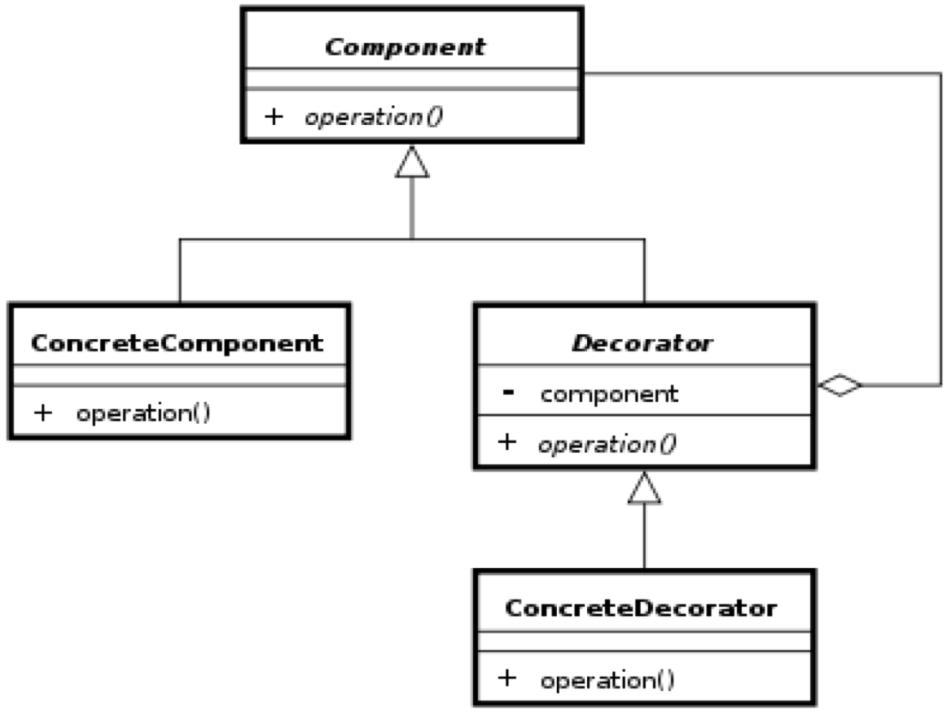
=>

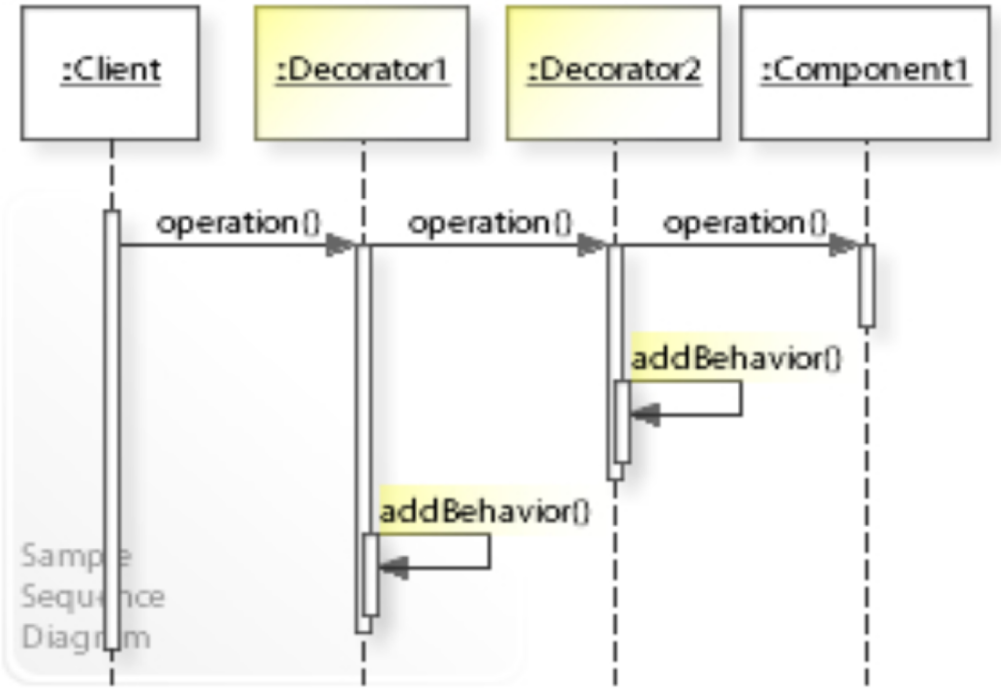
Po zapouzdření (“schování”) complexity za fasádu vypadá volání takto:

```java
facade.startEngine();
// ...
facade.stopEngine();
```

07 DEKORÁTOR

Přidání funkcionality do existujícího objektu bez jeho modifikace





07 DEKORÁTOR

```java
public interface DataSource {
    void writeData(String data);

    String readData();
}
```

```java
public class FileDataSource implements DataSource {
    private String name;

    public FileDataSource(String name) {
        this.name = name;
    }

    public void writeData(String data) {
        File file = new File(name);
        try (OutputStream fos = new FileOutputStream(file)) {
            fos.write(data.getBytes(), 0, data.length());
        } catch (IOException ex) {
            System.out.println(ex.getMessage());
        }
    }

    public String readData() {
        char[] buffer = null;
        File file = new File(name);
        try (FileReader reader = new FileReader(file)) {
            buffer = new char[(int) file.length()];
            reader.read(buffer);
        } catch (IOException ex) {
            System.out.println(ex.getMessage());
        }
        return new String(buffer);
    }
}
```

07 DEKORÁTOR

```java
public class DataSourceDecorator implements DataSource {
    private DataSource wrappee;

    DataSourceDecorator(DataSource source) {
        this.wrappee = source;
    }

    @Override
    public void writeData(String data) {
        wrappee.writeData(data);
    }

    @Override
    public String readData() {
        return wrappee.readData();
    }
}
```

```java
public class EncryptionDecorator extends DataSourceDecorator {

    public EncryptionDecorator(DataSource source) {
        super(source);
    }

    @Override
    public void writeData(String data) {
        super.writeData(encode(data));
    }

    @Override
    public String readData() {
        return decode(super.readData());
    }

    private String encode(String data) {
        byte[] result = data.getBytes();
        for (int i = 0; i < result.length; i++) {
            result[i] += (byte) 1;
        }
        return Base64.getEncoder().encodeToString(result);
    }

    private String decode(String data) {
        byte[] result = Base64.getDecoder().decode(data);
        for (int i = 0; i < result.length; i++) {
            result[i] -= (byte) 1;
        }
        return new String(result);
    }
}
```

07 DEKORÁTOR

```
public class CompressionDecorator extends DataSourceDecorator {
    private int compLevel = 6;
    public CompressionDecorator(DataSource source) {
        super(source);
    }
    public int getCompressionLevel() {
        return compLevel;
    }
    public void setCompressionLevel(int value) {
        compLevel = value;
    }
    @Override
    public void writeData(String data) {
        super.writeData(compress(data));
    }

    @Override
    public String readData() {
        return decompress(super.readData());
    }
    private String compress(String stringData) {
        byte[] data = stringData.getBytes();
        try {
            ByteArrayOutputStream bout = new ByteArrayOutputStream(512);
            DeflaterOutputStream dos = new DeflaterOutputStream(bout, new Deflater(compLevel));
            dos.write(data);
            dos.close();
            bout.close();
            return Base64.getEncoder().encodeToString(bout.toByteArray());
        } catch (IOException ex) {
            return null;
        }
    }
    private String decompress(String stringData) {
        byte[] data = Base64.getDecoder().decode(stringData);
        try {
            InputStream in = new ByteArrayInputStream(data);
            InflaterInputStream iin = new InflaterInputStream(in);
            ByteArrayOutputStream bout = new ByteArrayOutputStream(512);
            int b;
            while ((b = iin.read()) != -1) {
                bout.write(b);
            }
            in.close();
            iin.close();
            bout.close();
            return new String(bout.toByteArray());
        } catch (IOException ex) {
            return null;
        }
    }
}
```

07 DEKORÁTOR

```java
public class Demo {
    public static void main(String[] args) {
        String salaryRecords = "Name,Salary\nJohn Smith,100000\nSteven Jobs,912000";
        DataSourceDecorator encoded = new CompressionDecorator(
                                        new EncryptionDecorator(
                                            new FileDataSource("out/OutputDemo.txt")));
        encoded.writeData(salaryRecords);
        DataSource plain = new FileDataSource("compressed_encrypted_data.txt");

        System.out.println("- Input ----------------");
        System.out.println(salaryRecords);
        System.out.println("- Compressed and encoded ");
        System.out.println(plain.readData());
        System.out.println("- Decoded --------------");
        System.out.println(encoded.readData());
    }
}
```

```
- Input ----------------
Name,Salary
John Smith,100000
Steven Jobs,912000
- Encoded --------------
Zkt7e1Q5eU8yUm1Qe0ZsdHJ2VXp6dDBKVnhrUHtUe0sxRUYxQkJIdjVLTVZ0dVI5Q2IwOXFISmVUMU5rcENCQmdxRlByaD4+
- Decoded --------------
Name,Salary
John Smith,100000
Steven Jobs,912000
```

07 FLYWEIGHT

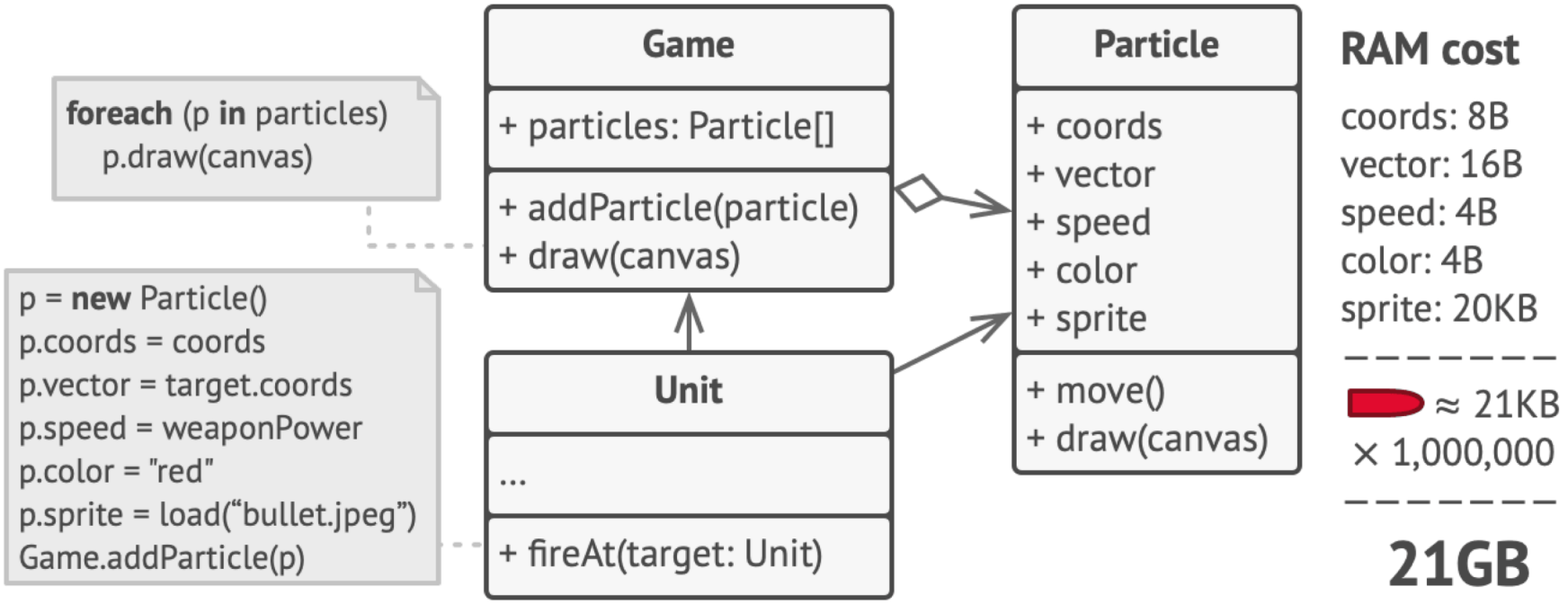
Flyweight je strukturní návrhový vzor, který vám umožňuje vměstnat více objektů do dostupného množství paměti RAM sdílením společných částí stavu mezi více objekty namísto uchování všech dat v každém objektu.

Pozn:. Obecný předpis ve formě class diagramu neexistuje

source: https://refactoring.guru/

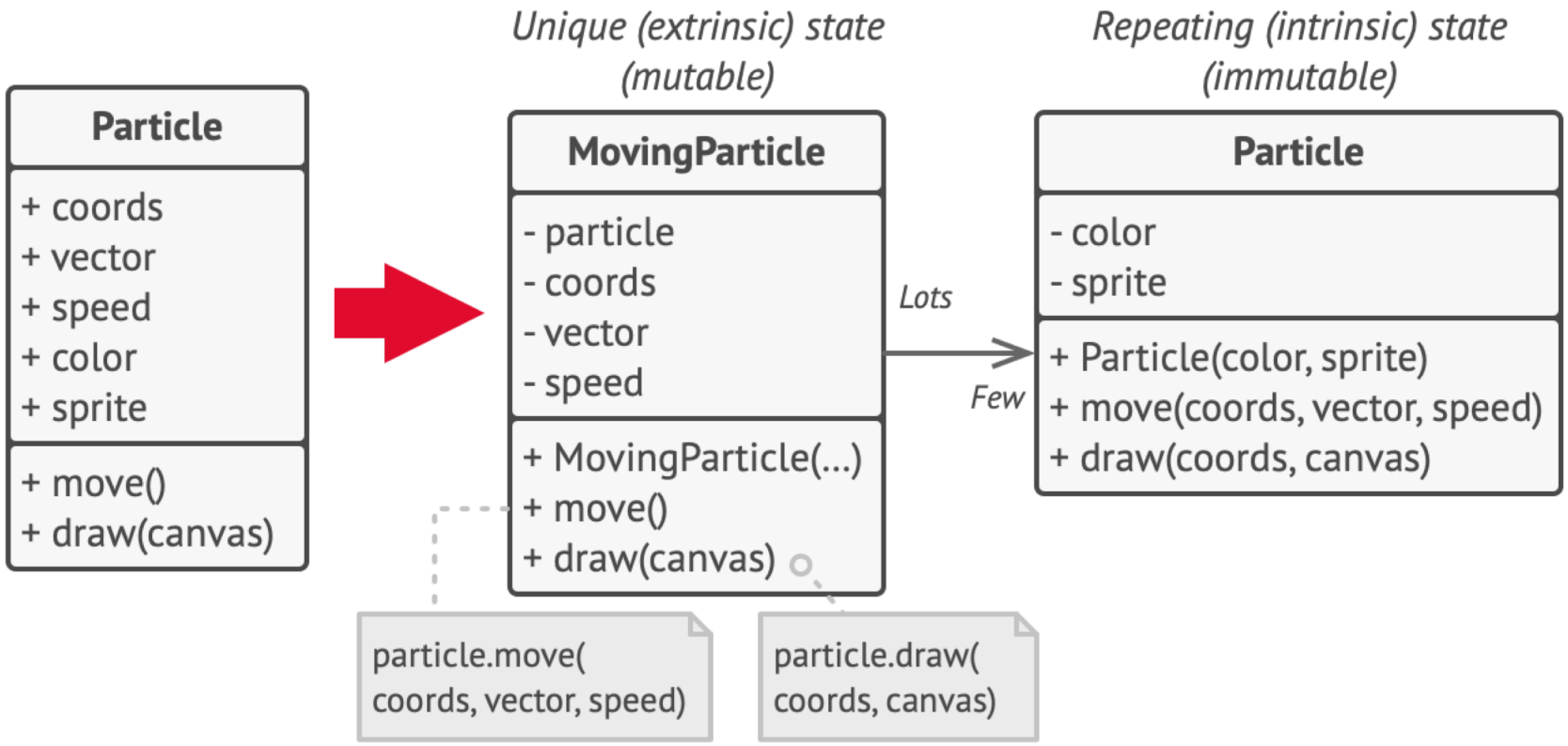
07 FLYWEIGHT

Střílečka: Objekt Particle ve hře reprezentuje střely, střepiny
Minecraft: Objekt particle reprezentuje slepice, které si můžete rozmnožovat



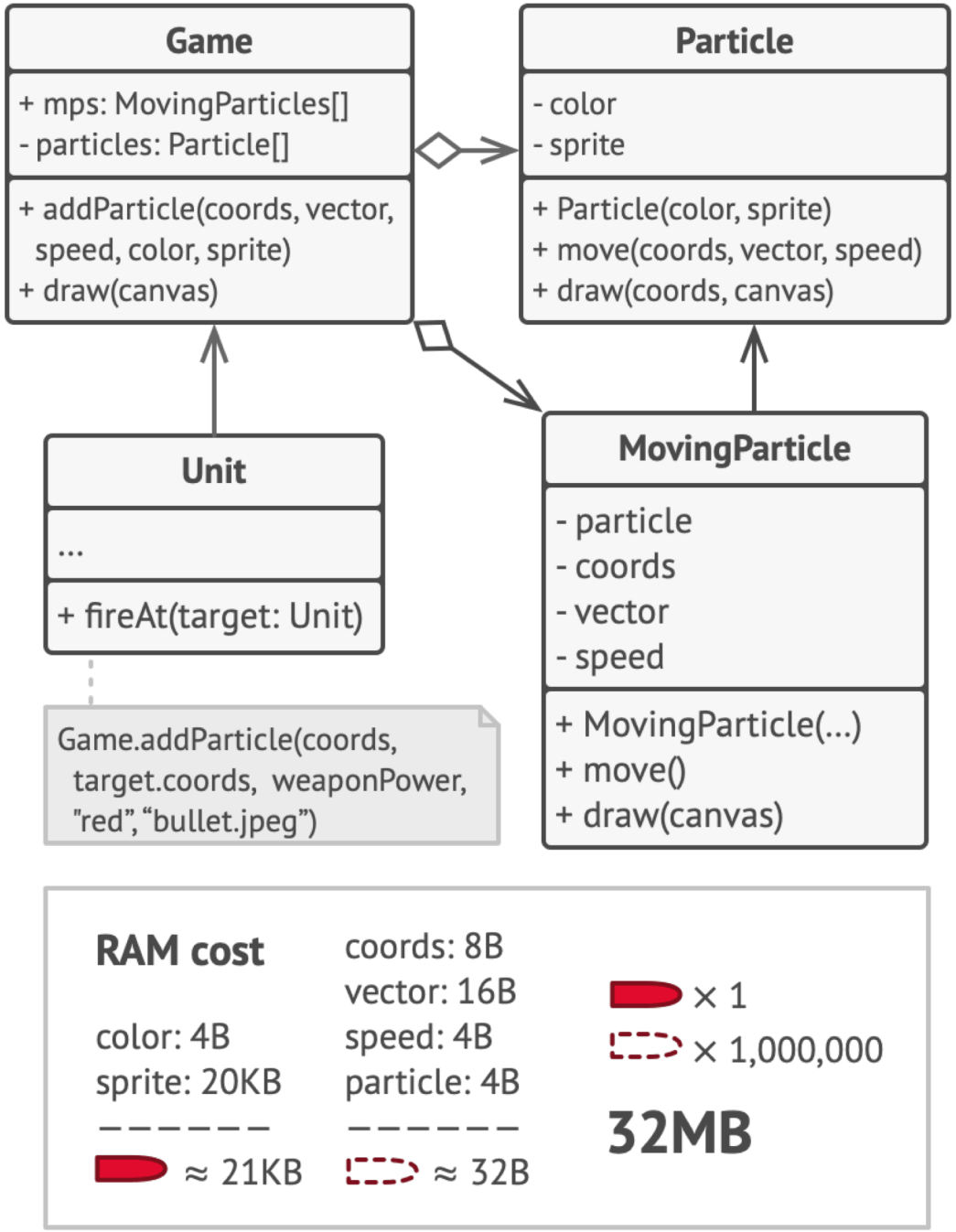
source: https://refactoring.guru/

07 FLYWEIGHT



source: https://refactoring.guru/

07 FLYWEIGHT



07 BRIDGE

… aneb mixujeme hierarchii tříd s kompozicí

=> přidáváme k původní hierarchii tříd metody a atributy jinak než dědičností

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

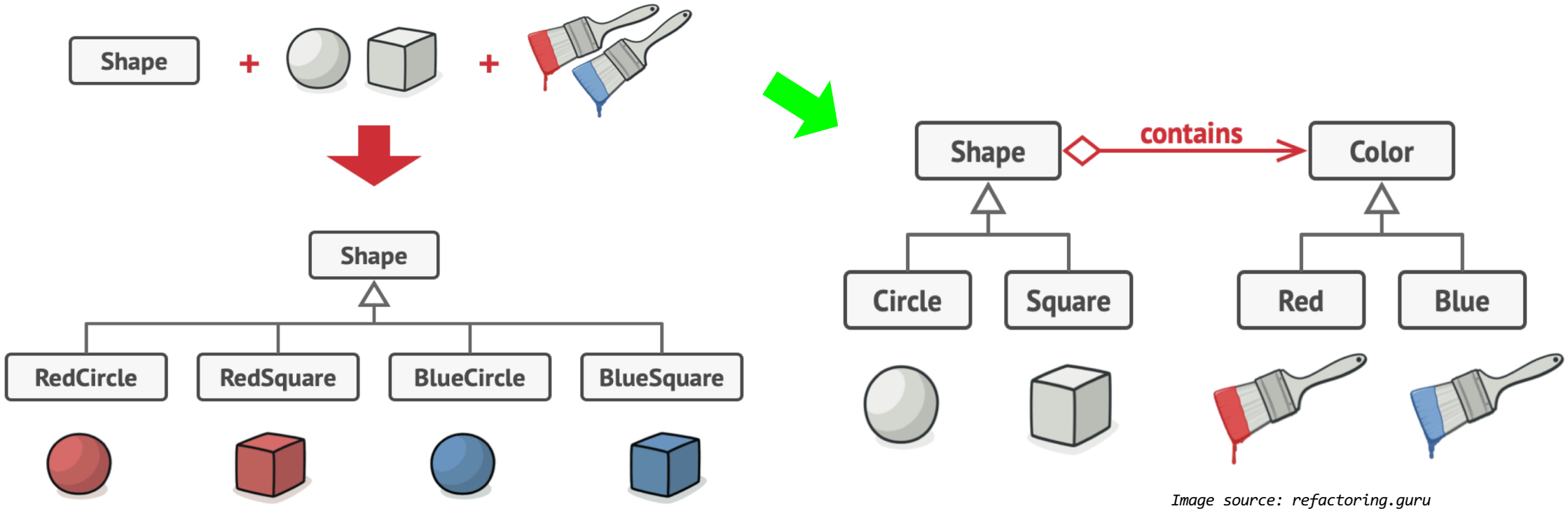
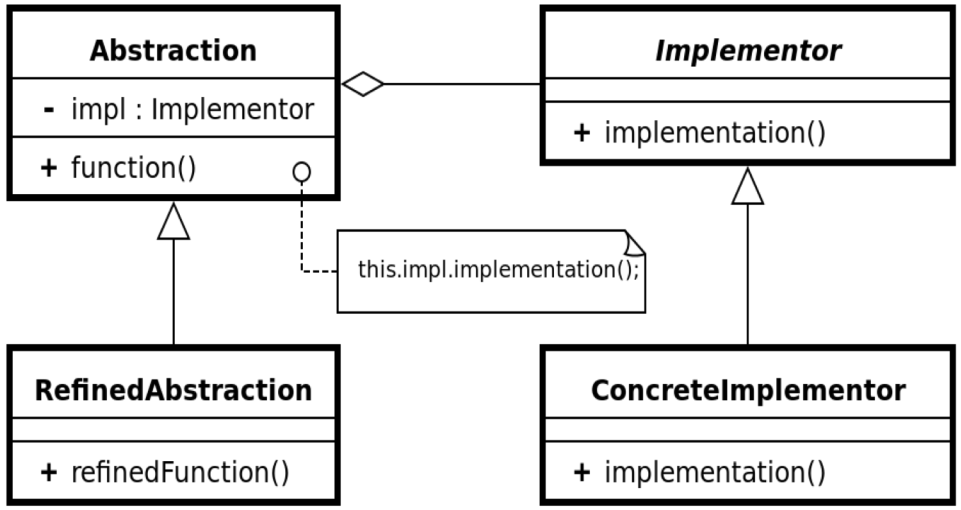
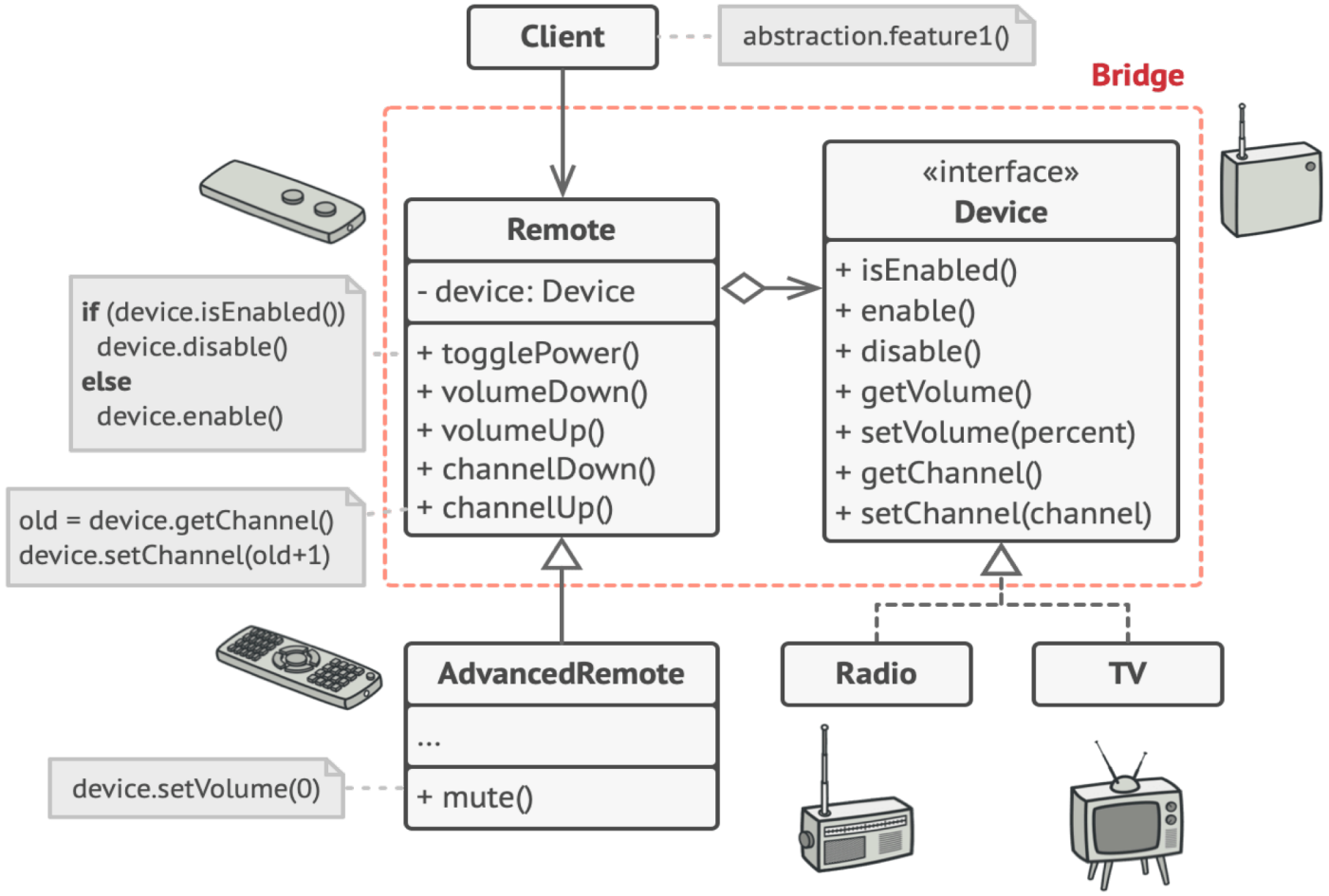


Image source: refactoring.guru

07 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

07 POROVNÁNÍ

- 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 Dekorátoru je řízena klientem

- Facade definuje nové rozhraní, Adapter spojuje existující rozhraní