Beknopte samenvatting OO Ontwerpen III

geschreven door

jakoblierman

OO ONTWERPEN III

Academiejaar 2017 - 2018

HoGent

Jakob Lierman
HoGent

Inhoud

Inleiding

Creational Patterns

- Factory method
- Abstract factory
- Builder
- Singleton

Behavior Patterns

- Template method
- Command
- Iterator

Structural Patterns

- Composite
- Adaptor
- Proxy

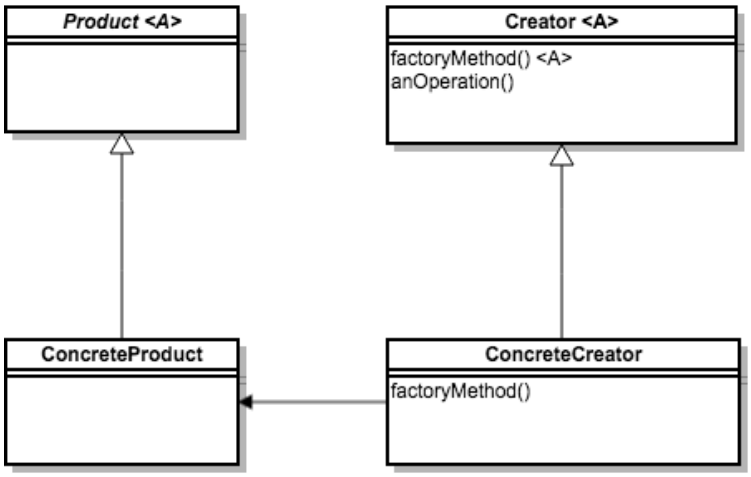
Factory Pattern

Voorbeeld: pizzeria.

Factory Method

Het factory method pattern definieert een interface voor het creëren van een object, maar laat de subklassen beslissen welke klasse er geïnstantieerd wordt. De factory method draagt de instantie over aan de subklassen.

UML Diagram



Ontwerpprincipe

Dependency inversion principe:

- Wees afhankelijk van abstracties
- Wees niet afhankelijk van concrete klassen

Het principe suggereert dat onze high-level componenten niet afhankelijk mogen zijn van onze low-level componenten. Beiden zouden moeten afhangen van abstracties.

Code

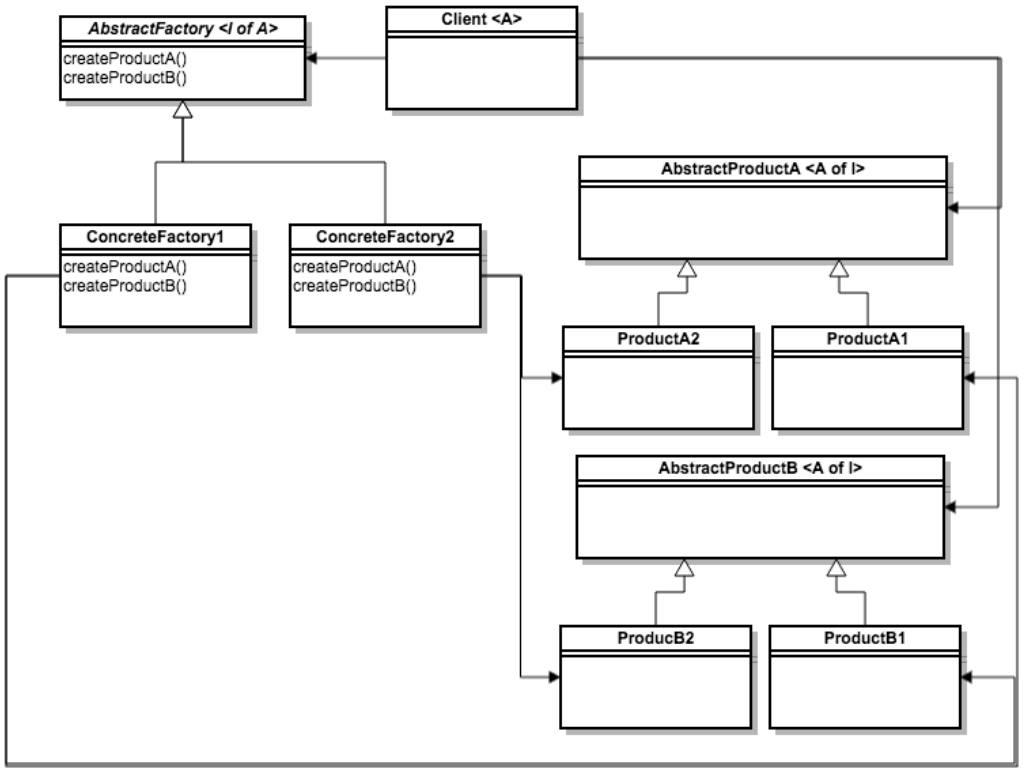
```
public interface PizzaIngredientFactory {
    public Dough createDough();
    public Sauce createSauce();
    public Cheese createCheese();
}

public class BinfPizzaIngredientFactory implements PizzaIngredientFactory {
    public Dough createDough() {
        return new ThinCrustDough();
    }

    public Sauce createSauce() {
        return new MarinaraSauce();
    }

    public Cheese createCheese() {
        return new ReggianoCheese();
    }
}

public abstract class Pizza {
    private Dough dough;
```

```
    private Sauce sauce;
    private Cheese cheese;

    private PizzaIngredientFactory ingredientFactory;

    public Pizza(PizzaIngredientFactory ingredientFactory) {
        this.ingredientFactory = ingredientFactory;
    }

    public abstract void prepare();

    public void bake() {
        System.out.println("Bake for 25 minutes at 350");
    }
}

public class CheesePizza extends Pizza {
    public PepperoniPizza(PizzaIngredientFactory ingredientFactory) {
        super(ingredientFactory);
    }

    public void prepare() {
        setDough(getPizzaIngredientFactory().createDough());
        setSauce(getPizzaIngredientFactory().createSauce());
        setCheese(getPizzaIngredientFactory().createCheese());
    }
}

public abstract class PizzaStore {
    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }

    protected abstract Pizza createPizza(String type);
}

public class BinfPizzaStore extends PizzaStore {
    protected Pizza createPizza(String item) {
        Pizza pizza = null;
        PizzaIngredientFactory ingredientFactory = new BinfPizzaIngredientFactory();

        switch(item.toLowerCase()) {
            case "cheese":
                pizza = new CheesePizza(ingredientFactory);
                pizza.setName("BINF Style Cheese Pizza");
                break;
        }

        return pizza;
    }
}
```

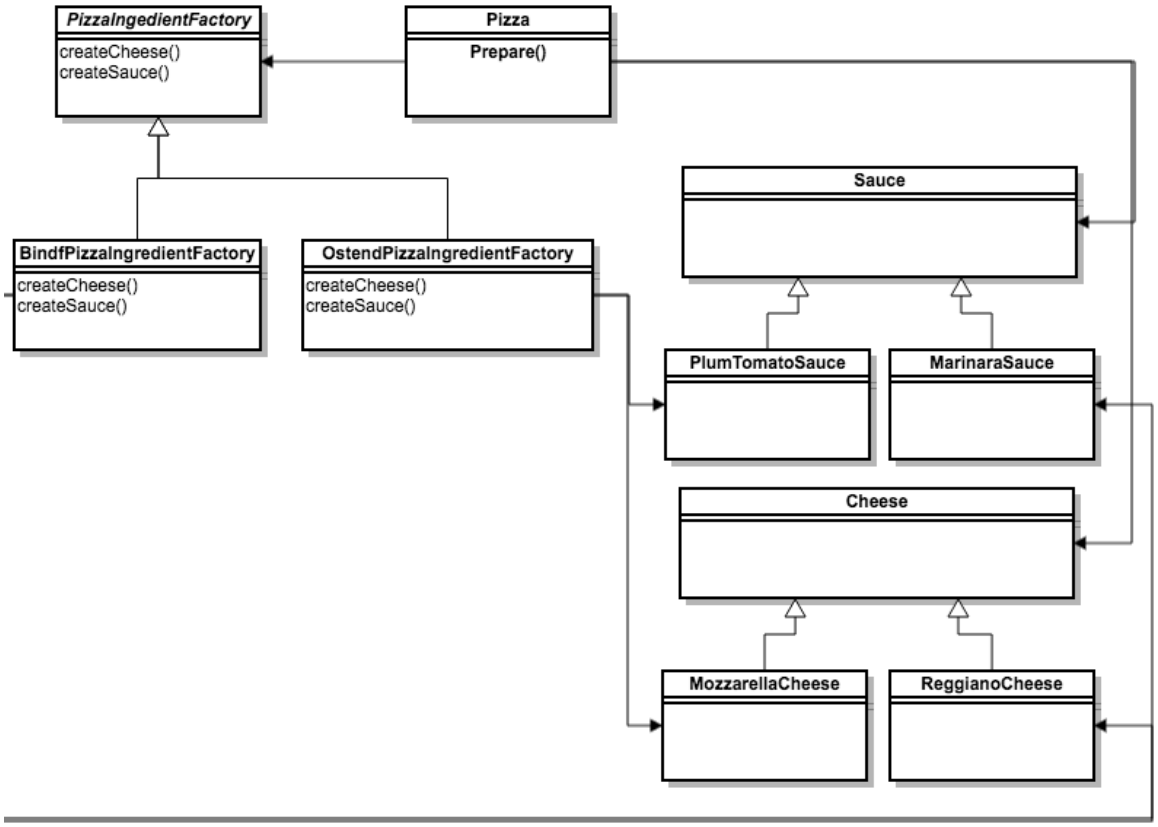
Abstract Factory

Het abstract factory pattern levert een interface voor de vervaardiging van reeksen gerelateerde of afhankelijke objecten zonder hun concrete klassen te specifieren.

UML Diagram



Pizza voorbeeld:

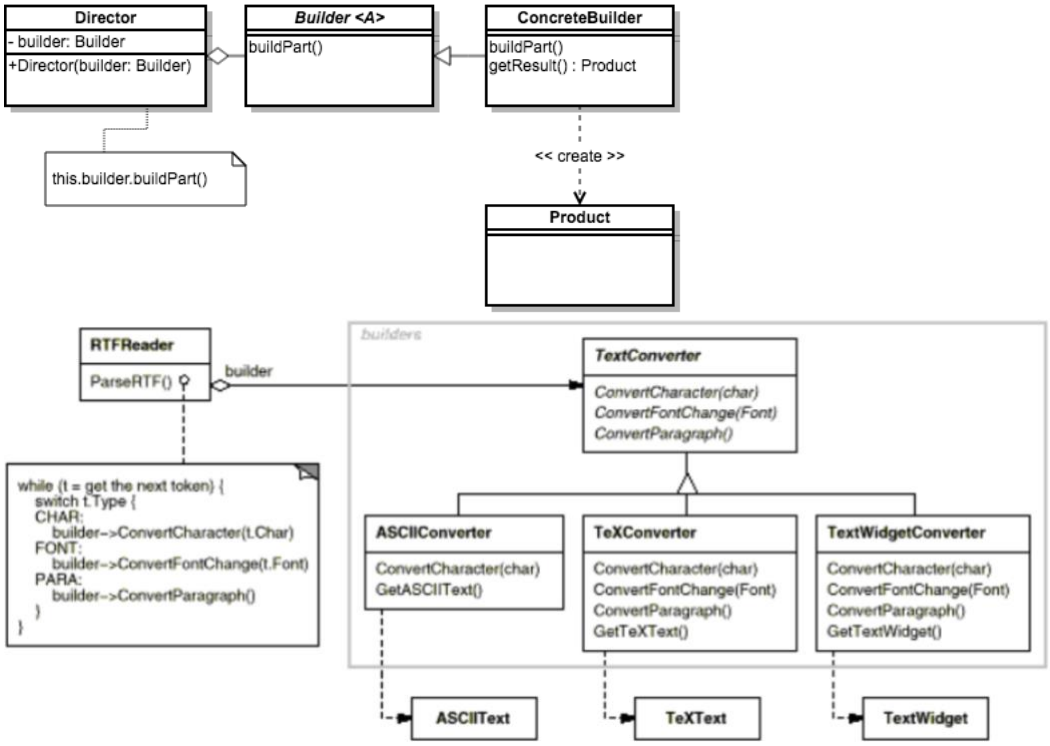


Builder Pattern

Vervangen van veel te grote constructors op een interessante manier.

Gebruik het builder pattern om de constructie van een produvt af te scherm en zorg dat je het in stappen kan construeren.

UML Diagram



De **Builder** klasse specifieert een abstracte interface voor de creatie van de onderdelen van het Product object.

De **ConcreteBuilder** bouwt de onderdelen van het complexe object en gooit deze samen door implementatie van de Builder interface. Het houdt een representaite van het object bij en biedt een interface voor het opvragen van het product.

De **Director** class bouwt het complexe object gebruik makend van de interface van de Builder.

De **Product** stelt het complexe object voor dat gebouwd wordt.

Code

```
// Dit is maar voor 1 specifieke constructor/soort sandwich
public class MySandwhichBuilder {
    private Sandwich sandwich;
    public Sandwich getSandwich() {
        return sandwich;
    }

    public void createSandwich() {
        // Opeenvolgende stappen bij het maken van een sandwich
        createNewSandwich();
        prepareBread();
        applyMeatAndCheese();
```

```
        applyVegetables();
        addCondiments();
    }

    private void createNewSandwich() {
        // Een van de stappen
        sandwich = new Sandwich();
    }

    private void prepareBread() {
        sandwich.setbread(BreadType.Whear);
    }

    private applyMeatAndCheese() {
        sandwich.setCheeseType(CheeseType.American);
        sandwich.setMeatType(MeatType.Turkey);
    }

    private void applyVegetables() {
        List<String> vegetables = new ArrayList<>();
        vegetables.add("Tomato");
        vegetables.add("Lettuce");
        sandwich.setVegetables(vegetables);
    }

    private void addCondiments() {
        sandwich.setHasMayo(false);
        sandwich.setIsToasted(true);
        sandwich.setHasMustard(true);
    }
}
// De builder: de abstracte klasse
public abstract class SandwichBuilder {
    private Sandwich sandwich;
    public Sandwich getSandwich {
        return sandwich;
    }

    public void createNewSandwich() {
        sandwich = new Sandwich();
    }

    public abstract void prepareBread();
    public abstract void applyMeatAndCheese();
    public abstract void applyVegetables();
    public abstract void addCondiments();
}

// De builder klassen: concrete klassen
public class MySandwhichBuilder extends SandwichBuilder {
    public void prepareBread() {
        Sandwich sandwich = getSandwich();
        sandwich.setbreadType(BreadType.Wheat);
    }
    public void applyMeatAndCheese() {
        // ...
    }
    public void applyVegetables() {
        // ...
    }
    public void addCondiments() {
        // ...
    }
}

public class ClubSandwichBuilder extends SandwichBuilder {
```

```
    public void prepareBread() {
        Sandwich sandwich = getSandwich();
        sandwich.setbreadType(BreadType.White);
    }
    public void applyMeatAndCheese() {
        // ...
    }
    public void applyVegetables() {
        // ...
    }
    public void addCondiments() {
        // ...
    }
}

// De Director
public class SandwichDirector {
    private SandwichBuilder builder;

    public SandwichDirector(SandwichBuilder builder) {
        this.builder = builder;
    }

    public void buildSandwich() {
        builder.createNewSandwich();
        builder.prepareBread();
        builder.applyMeatAndCheese();
        builder.applyVegetables();
        builder.addCondiments();
    }

    public Sandwich getSandwich() {
        return builder.getSandwich();
    }
}

public static void main(String[] args) {
    SandwichDirector director = new SandwichDirector(new MySandwhichBuilder());

    director.buildSandwich();
    Sandwich sandwich = director.getSandwich();
    sandwich.display();
}
```

Voordelen

- Schermt de manier waarop een complex object gebouwd wordt af
- Geeft de mogelijkheid om objecten in meerdere stappen en wisselende processen te maken (in tegenstelling tot een éénstapsfactory)
- Verbergt de interne represantatie van het priduct voor de client
- Productimplementaties kunnen steeds wisselen, omdat een client alleen een abstracte interface ziet

Gebruik en nadelen

- Wordt vaak gebruikt voor samengestelde objecten
- Het maken van een object vereist meer domeinkennis van de client (tenzij je een Director klasse kan voorzien) dan wanneer je een Factory gebruikt

Fluent variant

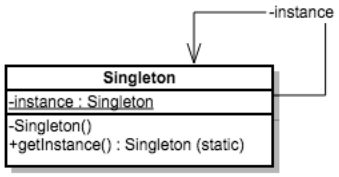
```
public class Rectangle {
```

```
    private final double opacity;
    private final double height;
    private final double width;
    // ...
    private final Color color;

    private static class Builder {
        private double opacity;
        private double height;
        private double width;
        private Color color;
        // ...
        public Builder opacity(double opacity) {
            this.opacity = opacity;
            return this;
        }

        public Builder height(double height) {
            this.height = height;
            return this;
        }

        public Builder width(double width) {
            this.width = width;
            return this;
        }

        public Builder color(Color color) {
            this.color = color;
            return this;
        }
        // ... meer methodes eventueel

        public Rectangle build() {
            return new Rectangle(this);
        }
    };

    private Rectangle(Builder builder) {
        this.opacity = builder.opacity;
        this.height = builder.height;
        this.width = builder.width;
        this.color = builder.color;
        // ... validatie
    }
}


public static void main(String[] args) {
    Rectangle r = new Rectangle.Builder()
        .height(250)
        .width(300)
        .opacity(0.5)
        .color(Color.PINK)
        .build();
}
```

Singleton Pattern

Het singleton pattern garandeert dat een klasse slechts één instantie heeft, en biedt een globaal toegangspunt ernaartoe.

UML Diagram



Code

```
// Eager loading
public class Singleton {

    // Eager het is er van in begin, het kan zijn dat je het zelf niet nodig hebt...
    private static final Singleton instance = new Singleton();

    private Singleton() {
    }

    public static Singleton getInstance() {
        return instance;
    }
}

// Lazy loading
public class Singleton {
    private static Singleton instance;

    private Singleton() {
    }

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }

        return instance;
    }
}
```

- **Lazy loading**: enkel laden vanaf je het nodig hebt
- **Eager loading**: vooraf laden, tegenovergestelde van lazy loading

Multithreading oplossingen

Zonder lazy loading

```
public class Singleton {

    // Maak meteen een instanties
    private static final Singleton instance = new Singleton();

    private Singleton() {
    }

    public static Singleton getInstance() {
```

```
        return instance;
    }
}
```

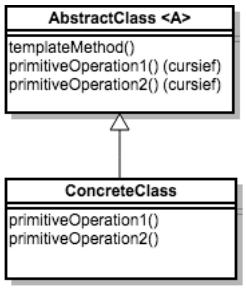
Met lazy loading

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {
    }

    // synchronized keyword
    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }

        return instance;
    }
}
```

Template method

Het template method pattern definieert het skelet van een algoritme in een methode, waarbij sommige stappen aan subklassen worden overgelaten. De template method laat subklassen bepaalde stappen in een algoritme herdefiniëren zonder de structuur van het algoritme te veranderen.

UML Diagram



Code

```
public abstract class AbstractClass {
    public final void templateMethod() {
        primitiveOperation1();
        primitiveOperation2();
        concreteOperation();
        hook();
    }

    // Deze zijn abstract en worden geimplementeerd door concrete subklassen
    protected abstract void primitiveOperation1();
    protected abstract void primitiveOperation2();

    // Een concrete implementatie, gedefinieerd als final zodat geen override mogelijk is
    protected final void concreteOperation() {
        // implementatie...
    }

    // een concrete methode maar ze doet niets. Een hook.
    // Deze kan, maar moet niet override worden in een subklasse
    protected void hook() {}
}
```

Voorbeeld:

```
public abstract class CaffeineBeverage {

    // MAG NIET OVERRIDEN WORDEN, vandaar de "final"
    public final void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();

        if (customerWantsCondiments()) addCondiments();
    }

    protected void boilWater() {
        System.out.println("Boiling water");
    }

    protected void pourInCup() {
```

```
        System.out.println("Boiling water");
    }

    protected abstract void brew();
    protected abstract void addCondiments();

    // Dit is een hook
    protected boolean customerWantsCondiments() {
        return true;
    }
}

public class Coffee extends CaffeineBeverage {
    @Override
    protected void brew() {
        System.out.println("Dripping coffee through filter");
    }

    @Override
    protected void addCondiments() {
        System.out.println("Adding sugar and milk");
    }
}

public class Tea extends CaffeineBeverage {
    @Override
    protected void brew() {
        System.out.println("Steeping the tea");
    }

    @Override
    protected void addCondiments() {
        System.out.println("Adding lemon");
    }
}

public class CoffeeWithHook extends CaffeineBeverage {
    private boolean wantsCondiments;
    public CoffeeWithHook(boolean wantsCondiments) {
        this.wantsCondiments = wantsCondiments;
    }
    @Override
    protected void brew() {
        System.out.println("Dripping coffee through filter");
    }

    @Override
    protected void addCondiments() {
        System.out.println("Adding sugar and milk");
    }

    protected boolean customerWantsCondiments() {
        return wantsCondiments;
    }
}

public class Template {
    public static void main(String[] args) {
        System.out.println("Making coffee");
        CaffeineBeverage beverage = new Coffee();
        beverage.prepareRecipe();

        System.out.println("Making tea");
        beverage = new Tea();
        beverage.prepareRecipe();
```

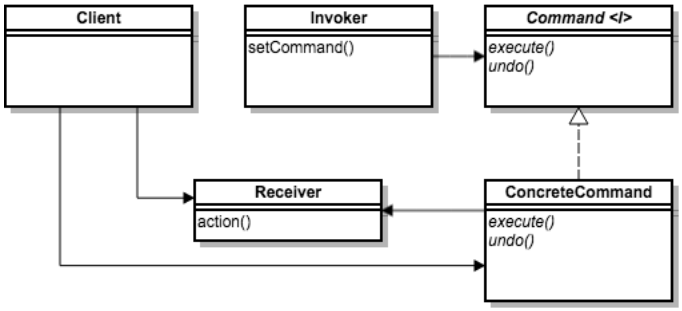
```
        System.out.println("Making coffee with a hook");
        boolean answer = getUserInputForCoffee();
        beverage = new CoffeeWithHook(answer);
        beverage.prepareRecipe();
    }

    public static boolean getUserInputForCoffee() {
        String answer = null;
        System.out.println("Would you like milk and sugar with your coffee (y/n)?");
        Scanner in = new Scanner(System.in);

        return in.next().equalsIgnoreCase("y");
    }
}
```

Command Pattern

Het command patterns schermt een aanroep af door middel van een object, waarbij je verschillende aanroepen in verschillende objecten kunt opbergen, in een queue kunt zetten of op schijf bewaren; ook undo-operaties kunnen worden ondersteund.

- Een actie wordt voorgesteld als een object
- De commands zijn “self-contained”
  Ze bevatten alle info die nodig is om de actie uit te voeren
- Nieuwe command’s kunnen eenvoudig worden toegevoegd (open/closed principe)

UML Diagram



Code

```
public interface Command { // Command
    void execute();
    void undo();
}

// Zorgt er voor dat we geen if-statements moeten zetten met de vraag of we wel execute()
kunnen aanroepen
public class NoCommand implements Command { // ConcreteCommand
    public void execute() {}
    public void undo() {}
}


// Voorbeelden
public class LightOnCommand implements Command { // ConcreteCommand
    private Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.on();
    }

    public void undo() {
        light.off();
    }
}

public class LightOffCommand implements Command { // ConcreteCommand
    private Light light;

    public LightOffCommand(Light light) {
        this.light = light;
```

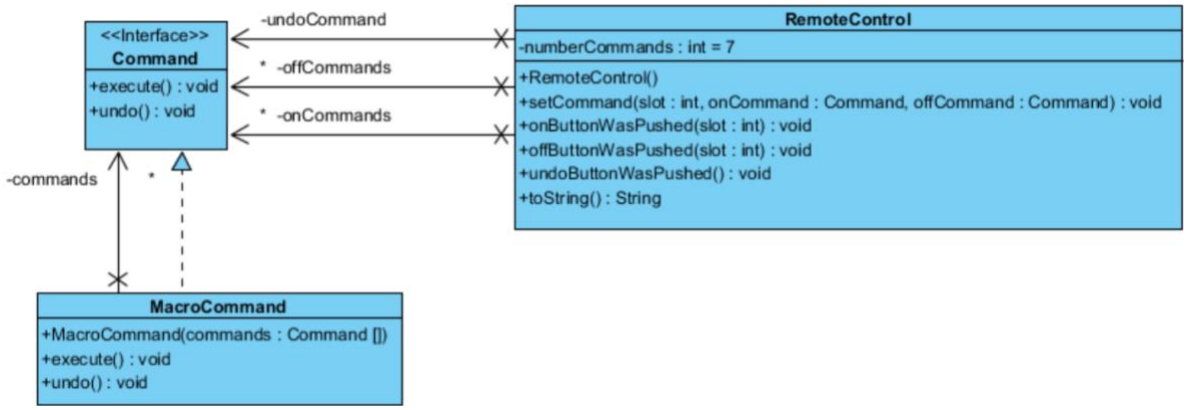
```
    }

    public void execute() {
        light.off();
    }

    public void undo() {
        light.on();
    }
}

public class StereoOnWithCDCommand implements Command { // ConcreteCommand
    private Stereo stereo;

    public StereoOnWithCDCommand(Stereo stereo) {
        this.stereo = stereo;
    }

    public void execute() {
        stereo.on();
        stereo.setCD();
        stereo.setVolume(11);
    }
}

// Hoe runnen:
public class RemoteControl { // Invoker, invoked commands
    private Command[] onCommands;
    private Command[] offCommands;
    private Command undoCommand;
    private final int numberCommands = 7;

    public RemoteControl() {
        onCommands = new Command[numberCommands];
        offCommands = new Command[numberCommands];
        Command noCommand = new NoCommand();

        for (int i = 0; i < numberCommands; i++) {
            onCommands[i] = noCommand;
            offCommands[i] = noCommand;
        }

        undoCommand = noCommand;
    }

    public void setCommand(int slot, Command onCommand, Command offCommand) {
        onCommands[slot] = onCommand;
        offCommands[slot] = offCommand;
    }

    public void onButtonWasPushed(int slot) {
        onCommands[slot].execute();
        undoCommand = onCommands[slot];
    }

    public void undoButtonWasPushed() {
        undoCommand.undo();
        undoCommand = new NoCommand();
    }

}

public class RemoteControlApp { // Client
    public static void main(String[] args) {
        RemoteControl remoteControl = new RemoteControl();
```

```
        Light livingRoomLight = new Light("Living Room Lighting");
        Light kitchenLight = new Light("Kitchen lighting");
        Stereo stereo = new Stereo("Stereo");
        // ...

        LightOnCommand livingRoomLightOn = new LightOnCommand(livingRoomLight);
        LightOffCommand livingRoomLightOff = new LightOffCommand(livingRoomLight);
        LightOnCommand kitchenLightOn = new LightOnCommand(kitchenLight);
        LightOffCommand kitchenLightOff = new LightOffCommand(kitchenLight);
        StereoOnWithCDCommand stereoOnWithCD = new StereoOnWithCDCommand(stereo);

        // Execution
        remoteControl.setCommand(1, livingRoomLightOn, livingRoomLightOff);
        remoteControl.setCommand(2, kitchenLightOn, kitchenLightOff);

        remoteControl.onButtonWasPushed(1);
        remoteControl.onButtonWasPushed(2);
        remoteControl.undoButtonWasPushed();
        remoteControl.undoButtonWasPushed();
    }
}
```

Macro-command

Een command, die een verzameling van commands bevat. Deze commands kan je dan één voor één uitvoeren (execute).

UML Diagram



Code

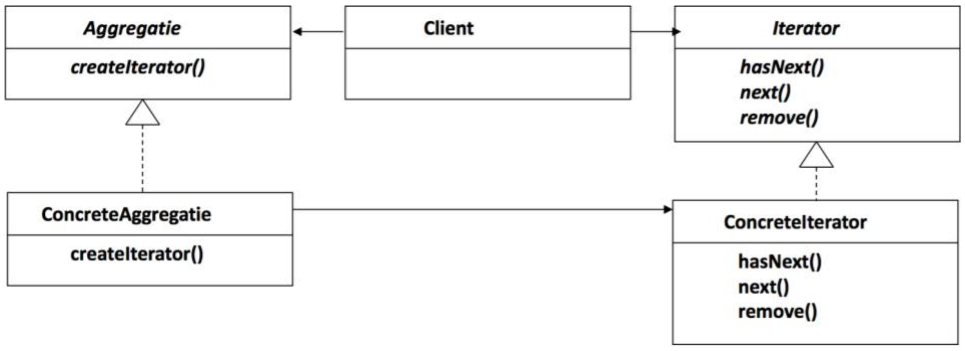
```
public class MacroCommand implements Command { // ConcreteCommand
    private Command[] commands;
    public MacroCommand(Command[] commands) {
        this.commands = commands;
    }

    public void execute() {
        Arrays.stream(commands).forEach(Command::execute);
    }

    public void undo() {

    }
}

public class RemoteLoader { // Client
    public static void main(String[] args) {
        RemoteControl remotecontrol = new RemoteControl();

        Light light = new Light("Living Room");
```

```
        Tv tv = new Tv("Living Room");
        Stereo stereo = new Stereo("Living Room");
        Hottub hottub = new Hottub();

        LightOnCommand lightOn = new LightOnCommand(light);
        StereoOnCommand stereoOn = new StereoOnCommand(stereo);
        TvOnCommand tvOn = new TvOnCommand(tv);
        HottubOnCommand hottubOn = new HottubOnCommand(hottub);

        LightOffCommand lightOff = new LightOffCommand(light);
        StereoOffCommand stereoOff = new StereoOffCommand(stereo);
        TvOffCommand tvOff = new TvOffCommand(tv);
        HottubOffCommand hottubOff = new HottubOffCommand(hottub);

        Command[] partyOn = { lightOn, stereoOn, tvOn, hottubOn };
        Command[] partyOff = { lightOff, stereoOff, tvOff, hottubOff };

        MacroCommand partyOnMacro = new MacroCommand(partyOn);
        MacroCommand partyOffMacro = new MacroCommand(partyOff);

        remoteControl.setCommand(0, partyOnMacro, partyOffMacro);

        System.out.println(remoteControl);

        System.out.println("--- Pushing Macro On---");
        remoteControl.onButtonWasPushed(0);

        System.out.println("--- Pushing Macro Off---");
        remoteControl.offButtonWasPushed(0);
    }
}
```

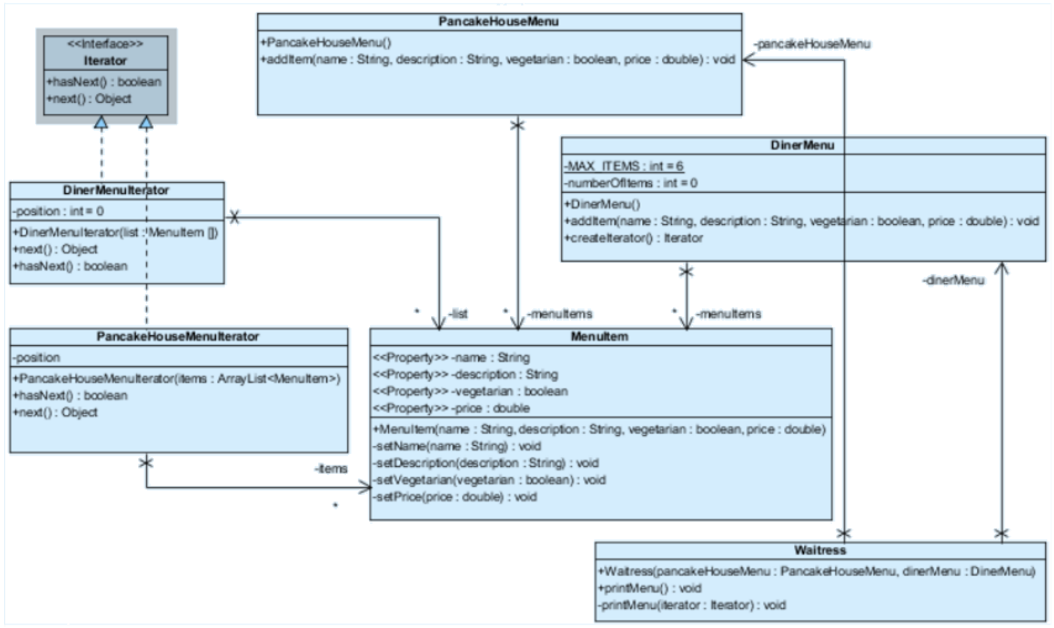
Iterator

Het iterator pattern voorziet ons van een manier voor sequentiële toegang tot de elementen van een aggregaat-object zonder de onderliggende representatie weer te geven.

UML Diagram



Voorbeeld:



Code

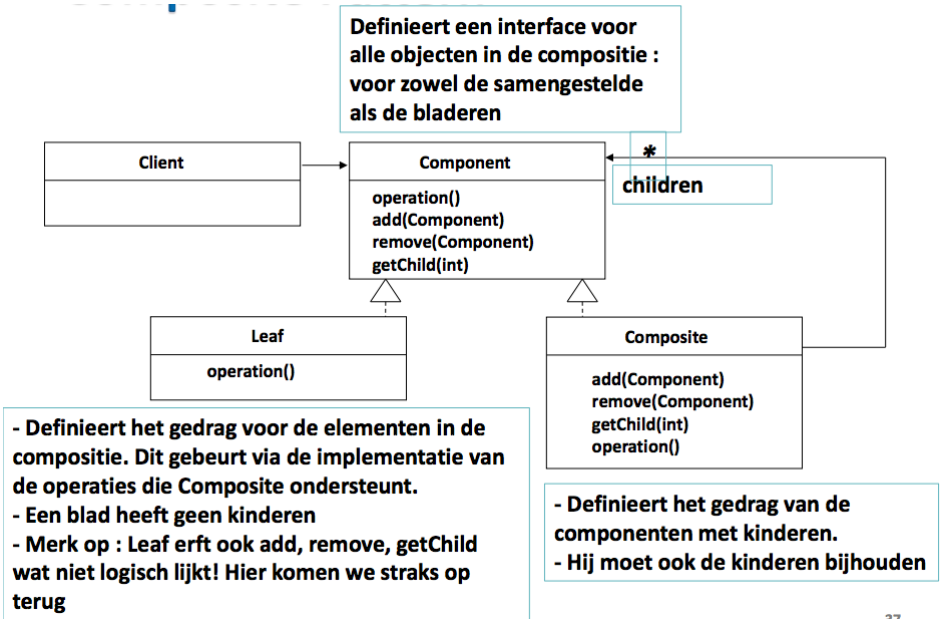
```
public class PancakeHouseMenu {
    private ArrayList<Menuitem> menuItems;
    public PancakeHouseMenu() {
        menuItems = ArrayList<>();

        addItem("K&B's Pancake Breakfast", "Pancakes with scrambled eggs, and toest",
        true, 2.99);
        // meer items toevoegen...
    }

    public void addItem(String name, String description, boolean vegetarian, double price){
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        menuItem.add(menuItem);
    }

    // Wrap whatever list/collection in an iterator
    public Iterator createIterator() {
```

```
        return new PancakeHouseMenuIterator(menuItems);
    }
}

public class Waitress {
    private PancakeHouseMenu pancakeHouseMenu;
    private DinerMenu dinerMenu;

    public Waitress(PancakeHouseMenu pancakeHouseMenu, DinerMenu dinerMenu) {
        this.pancakeHouseMenu = pancakeHouseMenu;
        this.dinerMenu = dinerMenu;
    }

    public void printMenu() {
        Iterator pancakeIterator = PancakeHouseMenuIterator.createIterator();

        printMenu(pancakeIterator);
    }

    private void printMenu(Iterator iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = iterator.next();

            System.out.print(menuItem.getName() + ",");
            System.out.print(menuItem.getPrice() + "--");
            System.out.println(menuItem.getDescription());
        }
    }
}

public static void main(String[] args) {
    PancakeHouseMenu pcm = new PancakeHouseMenu();
    DinerMenu dm = new DinerMenu();

    Waitress waitress = new Waitress(pcm, dm);

    waitress.printMenu();
}
```

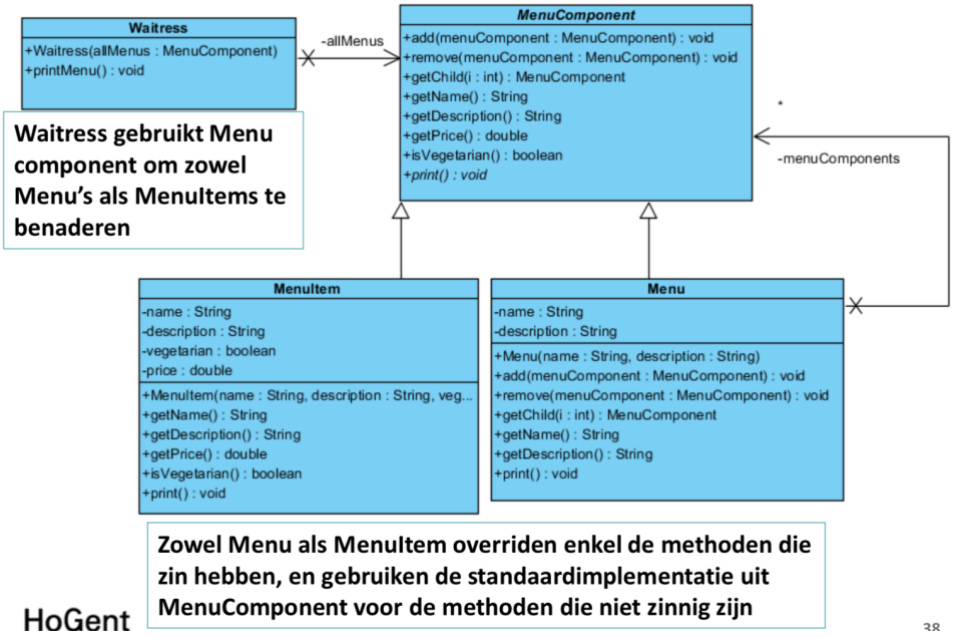
Composite Pattern

Het composite pattern stelt je in staat om objecten in boomstructuren samen te stellen om partwhole hiërarchiën weer te geven. Composite laat clients de afzonderlijke objecten of samengestelde objecten op uniforme wijze behandelen.

UML Diagram



Voorbeeld:



Code

```
public abstract class MenuComponent {
    public void add(MenuComponent menuComponent) {
        throw new UnsupportedOperationException();
    }

    public void remove(MenuComponent menuComponent) {
```

```
        throw new UnsupportedOperationException();
    }

    public MenuComponent getChild(int i) {
        throw new UnsupportedOperationException();
    }

    public String getName() {
        throw new UnsupportedOperationException();
    }

    public String getDescription() {
        throw new UnsupportedOperationException();
    }

    public double getPrice() {
        throw new UnsupportedOperationException();
    }

    public boolean isVegetarian() {
        throw new UnsupportedOperationException();
    }

    public abstract void print();
}

public class MenuItem extends MenuComponent {
    private String name;
    private String description;
    private boolean vegetarian;
    private double price;

    public MenuItem(String name, String description, boolean vegetarian, double price) {
        this.name = name;
        this.description = description;
        this.vegetarian = vegetarian;
        this.price = price;
    }

    public String getName() { return name; }
    public String getDescription() { return description; }
    public double getPrice() { return price; }
    public boolean isVegetarian() { return vegetarian; }

    public void print() {
        System.out.print("  " + getName());
        if (isVegetarian()) {
            System.out.print("(v)");
        }
        System.out.println("," + getPrice()):
        System.out.println("  --" + getDescription()):
    }
}

public class Menu extends MenuComponent {
    private List<MenuComponent> menuComponents;
    private String name;
    private String description;

    public Menu(String name, String description) {
        this.name = name;
        this.description = description;
    }

    public void add(MenuComponent menuComponent) (
        menuComponents.add(menuComponent);
```

```
    )

    public void remove(MenuComponent menuComponent) (
        menuComponents.remove(menuComponent);
    )

    public MenuComponent getChild(int i) {
        return (MenuComponent) menuComponents.get(i);
    }

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }

    public void print() {
        System.out.print("\n" + getName());
        System.out.print("," + getDescription());
        System.out.print("--------------------");

        menuComponents.forEach(MenuComponent::print);
    }
}

public class Waitress {
    private MenuComponent allMenus;

    public Waitress(MenuComponent allMenus) {
        this.allMenus = allMenus;
    }

    public void printMenu() {
        allMenus.print();
    }
}
```

Null iterator

```
public class NullIterator implements Iterator<MenuComponent> {
    public MenuComponent next() {
        return null;
    }

    public boolean hasNext() {
        return false;
    }
}
```

Composite iterator

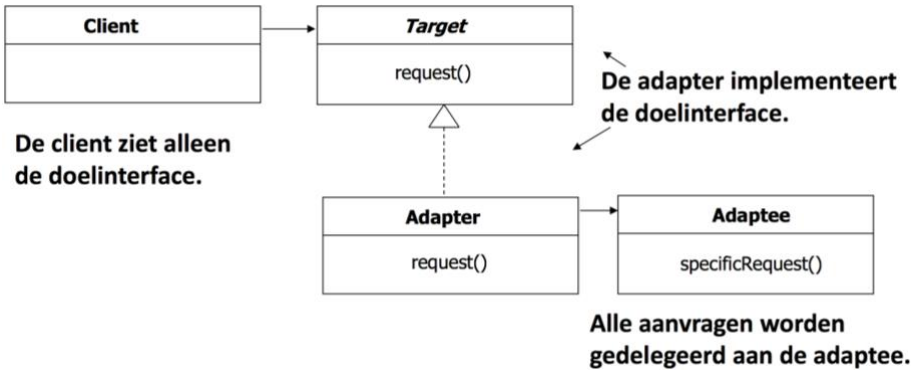
```
public class CompositeIterator implements Iterator<MenuComponent> {
    private Stack<Iterator<MenuComponent>> stack = new Stack<>();

    public CompositeIterator(Iterator<MenuComponent> iterator) {
        stack.push(iterator);
    }

    public MenuComponent next() {
        if (hasNext()) {
            Iterator<MenuComponent> iterator = stack.peek();
            MenuComponent component = iterator.next();

            // It is not a leaf, it has children
```

```
            if (component instanceof Menu) {
                stack.push(component.createIterator());
            }
        }

        return null;
    }

    public boolean hasNext() {
        if (stack.empty()) return false;

        Iterator<MenuComponent> iterator = stack.peek();

        if ( ! iterator.hasNext()) {
            stack.pop();
            return hasNext();
        }

        return true;
    }
}
```

Adapter Pattern

Het adapter pattern converteert de interface van een klasse naar een andere interface die de client verwacht. Adapters zorgen ervoor dat klassen samenwerken. Zonder de adapters lukt dit niet vanwege incompatibele interfaces.

UML Diagram



Code

```
public interface Duck {
    void quack();
    void fly();
}

public class MallardDuck implements Duck {
    public void quack() {
        System.out.println("Quack");
    }

    public void fly() {
        System.out.println("I'm flying");
    }
}

public interface Turkey {
    void gobble();
    void fly();
}

public class WildTurkey implements Turkey {
    public void gobble() {
        System.out.println("Gobble gobble");
    }

    public void fly() {
        System.out.println("I'm flying a short distance");
    }
}

public class TurkeyAdapter implements Duck {
    private Turkey turkey;

    public TurkeyAdapter(Turkey turkey) {
        this.turkey = turkey;
    }

    public void quack() {
        turkey.gobble();
```

```
    }

    public void fly() {
        for (int i = 0; i < 5; i++) {
            turkey.fly();
        }
    }
}

public class Adapter {
    public static void main(String[] args) {
        MallardDuck duck = new MallardDuck();
        testDuck(duck);

        WildTurkey turkey = new WildTurkey();
        Duck turkeyAdapter = new TurkeyAdapter(turkey);
        testDuck(turkeyAdapter);
    }

    static void testDuck(Duck duck) {
        duck.quack();
        duck.fly();
    }
}
```

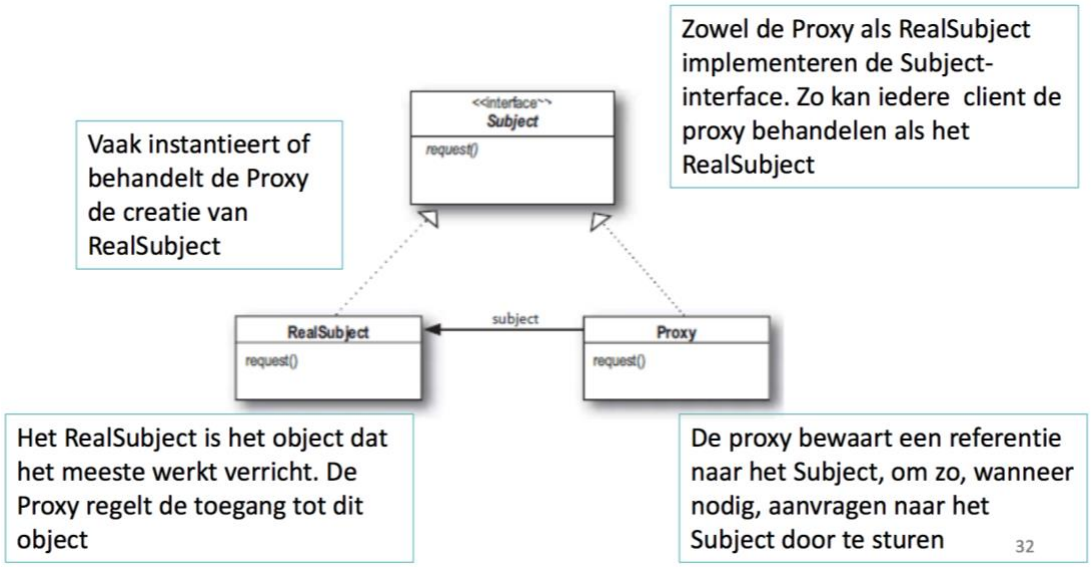
Proxy Pattern

Een proxy is structureel gezien gelijk aan een decorator, maar hun doelstellingen verschillen.

- Decorator voegt gedrag toe aan een object terwijl proxy de toegang regelt.

Het proxy pattern zorgt voor een surrogaat of plaatsvervanger voor een ander object om de toegang hiertoe te controleren.

UML Diagram



Code

```
public interface Image {
    void display();
}

public class RealImage implements Image {

   private String fileName;

   public RealImage(String fileName){
      this.fileName = fileName;
      loadFromDisk(fileName);
   }

   @Override
   public void display() {
      System.out.println("Displaying " + fileName);
   }

   private void loadFromDisk(String fileName){
      System.out.println("Loading " + fileName);
   }
}

public class ProxyImage implements Image{

   private RealImage realImage;
   private String fileName;

   public ProxyImage(String fileName){
```

```
      this.fileName = fileName;
   }

   @Override
   public void display() {
      if(realImage == null){
         realImage = new RealImage(fileName);
      }
      realImage.display();
   }
}

public class ProxyPatternDemo {

   public static void main(String[] args) {
      Image image = new ProxyImage("test_10mb.jpg");

      //image will be loaded from disk
      image.display();
      System.out.println("");

      //image will not be loaded from disk
      image.display();
   }
}
```

Remote Proxy (RMI)

```
public interface GumballMachineRemote extends Remote {
    // Wat de client zal kunnen opvragen
    int getCount() throws RemoteException;
    String getLocation() throws RemoteException;
    String getState() throws RemoteException;
}

// Alle argumenten & returnwaarden moeten serialiseerbaar zijn
public abstract class GumballMachineState implements Serializable {

    // Dit object willen we niet serialiseren en dus negeren
    transient protected GumballMachine gumballMachine;

    // States...
}

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

// Extend van URO & implement interface van hierboven
public class GumballMachine extends UnicastRemoteObject implements GumballMachineRemote {
    // Vergeet de throws niet
    public GumballMachine(String location, int numberGumballs) throws RemoteException {
        this.location = location;
        this.count = numberGumballs;
    }
}

public class GumballMonitor {
    private GumballMachineRemote machine;

    public GumballMonitor(GumballMachineRemote machine) {
        this.machine = machine;
    }

    public void report() {
        // doe dingen..
    }
```

```
}

public class ClientApp {
    public void blabla() {
        try {
            // Get remote registry object on port 1099 (rmi nameservice)
            Registry myRegistry = LocateRegistry.getRegistry("127.0.0.1", 1099);
            // dit is de default ==> LocateRegistry.getRegistry();

            // Search for remote object GumballMachineRemote (via rmi nameservice)
            GumballMachineRemote machine = (GumballMachineRemote)
myRegistry.lookup("gumballmachine");

            // Geef remote object door aan GumballMonitor
            GumballMonitor monitor = new GumballMonitor(machine);

            monitor.report();
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}

public class RemoteServiceApp { // Not sure though
    // Meld de service aan bij de RMI registry
    private void registerRemoteGumballMachine() {
        try {
            Registry registry = LocateRegistry.createRegistry(1099);
            machine = new GumballMachine(location, count);
            registry.rebind("gumballmachine", machine);
        } catch(RemoteException e) {
            e.printStackTrace();
        }
    }
}
```

Virtual Proxy

```
// Een tijdelijke afbeelding laden tot de echte afbeelding geladen is
public class ImageProxy implements Icon {
    private ImageIcon imageIcon;
    private URL imageUrl;
    private Thread retrievalThread;
    public ImageProxy(URL url) {
        this.imageUrl = url;
    }

    @Override
    public int getIconWidth() {
        if (imageIcon != null) {
            return imageIcon.getIconWidth(); // Doordelegeren
        }
        return 800; // Default
    }

    @Override
    public int getIconHeight() {
        if (imageIcon != null) {
            return imageicon.getIconHeight();
        }

        return 600;
    }

    public void paintIcon(final Component c, Graphics g, int x, int y) {
        if (imageIcon != null) {
```

```
            imageIcon.paintIcon(c, g, x, y); // Toon de echte afbeelding
        } else {
            g.drawString("cd cover wordt geladen, wachten aub...", x + 300, y + 190); //
tijdelijke string tonen

            if (retrievalThread == null) {
                retrievalThread = new Thread(new Runnable() {
                    @Override
                    public void run() {
                        try {
                            imageicon = new ImageIcon(imageUrl, "cd-cover"); // Nu pas
ophalen
                            c.repaint(); // Zal opnieuw uitvoeren zodat het eerste blokje
wordt uitgevoerd
                        } catch(Exception e) { e.printStackTrace(); }
                    }
                });
                retrievalThread.start();
            }
        }
    }
}

public class SomethingApp {
    public static void main(String[] args) {
        Icon icon = new ImageProxy("http://d.pr/i/1kx77+");

        // Echte gui zeker?
        imageComponent = new ImageComponent(icon);
        frame.getContentPane().add(imageComponent);
    }
}
```

Dynamic Proxy

```
public interface PersonBean {
    public String getName();
    public int getHotOrNotRating();

    public void setName(String string);
    public void setHotOrNotRating(int i);
}

public class PersonBeanImpl implements PersonBean {
    private String name;
    private int rating;
    private int ratingCount;

    public String getName() { return name; }
    public int getHotOrNotRating() {
        if (ratingCount == 0) return 0;

        return rating / ratingCount;
    }

    public void setName(String string) { this.name = string; }
    public void setHotOrNotRating(int i) { this.rating += rating; ratinCount++; }
}

// Nu zou de persoon zichzelf een hotOrNot rating kunnen geven: Das ziek ze manne!

import java.lang.reflect.*;

public class OwnerInvocationHandler implements InvocationHandler {
    private PersonBean person;
    public OwnerInvocationHandler(PersonBean person) { this.person = person; }
```

```
    public Object invoke(Object proxy, Method method, Object[] args) throws
IllegalAccessException
    {
        try {
            // Getter is altijd toegelaten
            if (method.getName().startsWith("get")) {
                return method.invoke(person, args);
            } else if (method.getName().equals("setHotOrNotRating")) {
                // Da mag niet omdat het een "owner" InvocationHandler is
                throw new IllegalAccessException();
            } else if (method.getName().startsWith("set")) {
                // andere setters zijn wel toegelaten, kijk eens aan!
                return method.invoke(person, args);
            } catch(InvocationHandler e) { e.printStackTrace(); throw e; }

            return null;
        }
    }
}

public class NonOwnerInvocationHandler implements InvocationHandler {
    private PersonBean person;
    public NonOwnerInvocationHandler(PersonBean person) { this.person = person; }

    public Object invoke(Object proxy, Method method, Object[] args) throws
IllegalAccessException
    {
        try {
            // Getter is altijd toegelaten
            if (method.getName().startsWith("get")) {
                return method.invoke(person, args);
            } else if (method.getName().equals("setHotOrNotRating")) {
                // Dit is de enige setter die wel mag gebeuren
                return method.invoke(person, args);
            } else if (method.getName().startsWith("set")) {
                // We willen niet dat andere personen, ons object kunnen wijzigen
                throw new IllegalAccessException();
            } catch(InvocationHandler e) { e.printStackTrace(); throw e; }

            return null;
        }
    }
}

public class SomethingApp {
    public Personbean getOwnerProxy(PersonBean person) {
        return (PersonBean) Proxy.newProxyInstance(
            person.getClass().getClassLoader(),
            person.getClass().getInterfaces(),
            new OwnerInvocationHandler(person)
        );
    }
    public Personbean getNonOwnerProxy(PersonBean person) {
        return (PersonBean) Proxy.newProxyInstance(
            person.getClass().getClassLoader(),
            person.getClass().getInterfaces(),
            new NonOwnerInvocationHandler(person)
        );
    }

    public static void main(String[] args) {
        test1();
        test2();
    }
```

```
    private void test1()
    {
        PersonBean joe = getPersonFromDatabase("Joe Javabean");
        PersonBean ownerProxy = getOwnerProxy(joe);
        System.out.println("Name is " + ownerProxy.getName());

        try {
            ownerProxy.setHotOrNotRating(10);
        } catch(Exception e) {
            System.out.println("Jezelf een hot or not rating geven is zoals u eigen
facebook afbeelding liken. Da doe ge niet!");
        }

        System.out.println("Rating is " + ownerProxy.getHotOrNotRating());
    }

    private void test2()
    {
        PersonBean joe = getPersonFromDatabase("Joe Javabean");
        PersonBean nonOwnerProxy = getNonOwnerProxy(joe);
        System.out.println("Name is " + nonOwnerProxy.getName());

        nonOwnerProxy.setHotOrNotRating(10); // Nu gaat da welke
        // Andere setters gaan niet..

        System.out.println("Rating is " + nonOwnerProxy.getHotOrNotRating());
    }
}
```