

# 6 Design Patterns (deel 2)

Programmeren 2 – Java

2017 - 2018

**KdG** Karel de Grote  
Hogeschool

Kris Behiels

Jan De Rijke

Mark Goovaerts

# Programmeren 2 - Java

---

1. Herhaling en Collections
2. Generics en documenteren
3. Annotations en Reflection
4. Testen en logging
5. Design patterns (deel 1)

## **6. Design patterns (deel 2)**

7. Lambda's en streams
8. Persistentie (JDBC)
9. XML en JSON
10. Threads
11. Synchronization
12. Concurrency



# Behandelde patterns:



Singleton



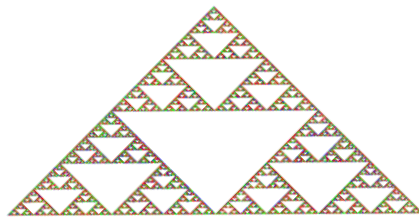
Observer



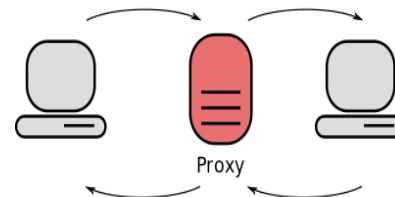
Static Factory



Adapter



Composite



Proxy



State

# Agenda

---

## 1. Adapter pattern

- Kenmerken en context
- Object adapter
- Class adapter



## 2. Composite pattern

- Kenmerken
- Voorbeelden

## 3. Proxy pattern

- Kenmerken en oplossingsmodel
- Remote Proxy
- Virtual Proxy
- Protection Proxy

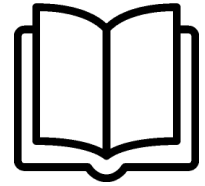
## 4. State pattern

- Kenmerken en context
- Voorbeelden



---

# Adapter



- E-book: "Adapter" p.288 ev
  - Uit: "Applied Java Patterns", First Edition (Stephen Stelting and Olav Maassen)

# Adapter: kenmerken

---



- Naam: **Adapter** pattern [GoF95]  
(ook wel het Wrapper pattern genoemd.)
- Familie: **Structural** patterns
- Samenvatting:  
*GoF: "Convert the interface of a class into another interface clients expect. Adapter let classes work together that couldn't otherwise because of incompatible interfaces."*
- Context:  
Het adapter pattern werkt als een **tussenpersoon** tussen twee klassen, waarbij de interface van de ene klasse geconverteerd wordt zodat die door de andere klasse kan gebruikt worden.
- Zie ook: *Decorator, Bridge, Proxy*

# Adapter: context

---



- Gebruik het **adapter pattern** als:
  - je een object wilt gebruiken in een omgeving die een interface verwacht die verschilt van de **interface** (publieke methoden) van het object
  - een object werkt als **tussenpersoon** voor één klasse uit een groep, en je weet pas op het moment van aanroep welke klasse je uit die groep moet kiezen.



# Adapter: oplossingmodel

---

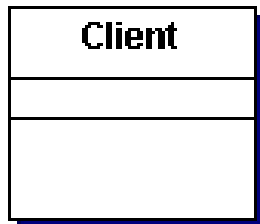


Oplossingsmodel: 2 varianten:

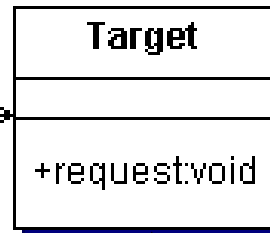
1. een **object adapter** die via **delegatie** naar een adaptee object verwijst.
2. een **klasse adapter** die gebruikt maakt van **overerving** om de ene interface aan de andere aan te passen.

# Object-adapter (delegatie)

**clientklasse** die een bepaalde interface aanroept



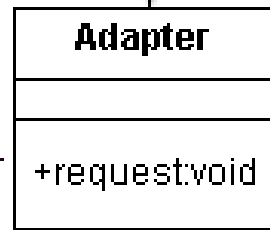
**interface** die door de client gebruikt wordt en de signature van de adapterklasse bepaalt



**Adaptee:** oorspronkelijke klasse (mag niet gewijzigd worden)



implements



return adaptee.specificRequest()

**Adapterklasse:** doorgeefluik tussen client en adaptee object

adaptee

# Object-adapter (delegation)

---

```
public class Adaptee {  
    public String specificRequest() {  
        return "Specific request on Adaptee";  
    }  
}
```

```
public interface Target {  
    public String request();  
}
```

```
public class Adapter implements Target {  
    private Adaptee adaptee;  
    public String request() {  
        if (adaptee == null) { adaptee = new Adaptee(); }  
        return adaptee.specificRequest();  
    }  
}
```

# Object-adapter (delegatie)

---

```
public class Client {  
    public static void main(String[] args) {  
  
        Target target = new Adapter();  
        System.out.println(target.request());  
  
    }  
}
```

/\*

**AFDRUK:**

*Specific request on Adaptee*

\*/

# Object-adapter (concreet voorbeeld)

de basisklasse

```
public class HelpDeskItem {
    private LocalDateTime localDateTime;
    private int priority;
    private String description;

    public HelpDeskItem(String description) {
        this.localDateTime = LocalDateTime.now();
        this.priority = 1; // normal priority
        this.description = description;
    }

    public HelpDeskItem(int priority, String description) {
        this.localDateTime = LocalDateTime.now();
        this.priority = priority;
        this.description = description;
    }

    @Override
    public String toString() {
        return String.format("%-10s: %2d %s", localDateTime,
                               priority, description);
    }
}
```



# Object-adapter (concreet voorbeeld)

de interface  
van de  
Adapterklasse

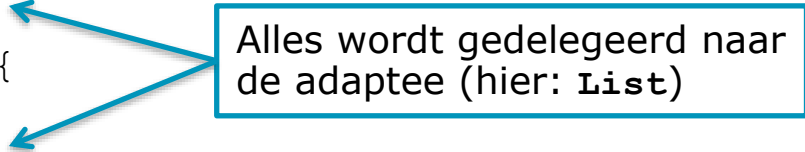
```
public interface HelpDeskQueue {  
    void enqueue(HelpDeskItem helpDeskItem);  
    HelpDeskItem dequeue();  
    void overviewByPriority();  
    void overviewNatural();  
}
```

# Object-adapter (concreet voorbeeld)

de Adapter-  
klasse

```
public class Queue2ListAdapter
    implements HelpDeskQueue {
    private List<HelpDeskItem> adaptee = new ArrayList<>();

    public void enqueue(HelpDeskItem helpDeskItem) {
        adaptee.add(helpDeskItem); // FIFO: achteraan toevoegen
    }
    public HelpDeskItem dequeue() {
        if(adaptee.size() > 0) {
            return adaptee.remove(0); //FIFO: vooraan verwijderen
        }
        return null;
    }
    public void overviewByPriority() { // toon volgens priority
    }
    public void overviewNatural() { // toon in FIFO-volgorde
    }
```



Alles wordt gedelegeerd naar de adaptee (hier: List)

# Object-adapter (concreet voorbeeld)

de Client-  
klasse

```
public class DemoAdapter {  
    static void main(String[] args) {  
        HelpDeskQueue myQueue = new Queue2ListAdapter();  
        myQueue.enqueue(new HelpDeskItem("Kan niet inloggen"));  
        myQueue.enqueue(new HelpDeskItem(5, "Server crash"));  
        myQueue.enqueue(new HelpDeskItem("Muis doet het niet"));  
        myQueue.enqueue(new HelpDeskItem(5, "Laptop in de fik!"));  
        myQueue.overviewNatural();  
        myQueue.overviewByPriority();  
    }  
}
```

De clientklasse gebruikt de  
interface van de adapter

Queue in natural order:

```
2016-10-18T13:55:24.146: 1 Kan niet inloggen  
2016-10-18T13:55:24.146: 5 Server crash  
2016-10-18T13:55:24.146: 1 Muis doet het niet  
2016-10-18T13:55:24.146: 5 Laptop in de fik!
```

Queue by priority:

```
2016-10-18T13:55:24.146: 5 Server crash  
2016-10-18T13:55:24.146: 5 Laptop in de fik!  
2016-10-18T13:55:24.146: 1 Kan niet inloggen  
2016-10-18T13:55:24.146: 1 Muis doet het niet
```

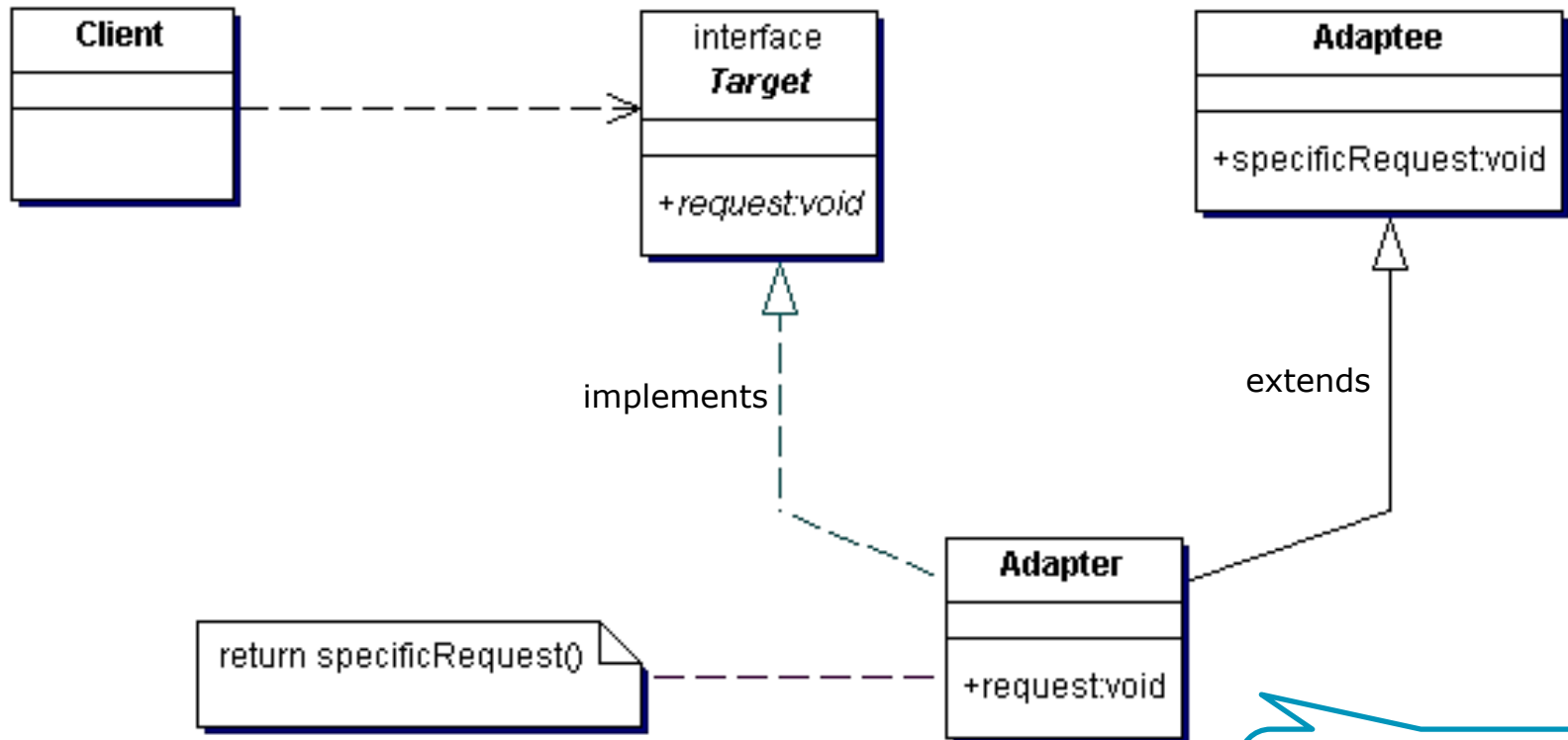


# Klasse-Adapter (overerving)

**clientklasse** die een bepaalde interface gewoon is

**interface** die door de client gebruikt wordt

**Adaptee:** oorspronkelijke klasse (mag niet gewijzigd worden)



**Adapterklasse:** doorgeefluik tussen client en adaptee

# Klasse-Adapter (overerving)

---

```
public class Adaptee {  
    public String specificRequest() {  
        return "Specific request on Adaptee";  
    }  
}
```

```
public interface Target {  
    public String request();  
}
```

```
public class Adapter extends Adaptee implements Target {  
    public String request() {  
        return super.specificRequest();  
    }  
}
```

# Klasse-Adapter (overerving)

---

```
public class Client {  
    public static void main(String[] args) {  
        Target target = new Adapter();  
        System.out.println(target.request());  
    }  
}
```

/\*

*AFDRUK:*

*Specific request on Adaptee*

\*/

# Klasse-adapter (concreet voorbeeld)

de interface  
van de  
Adapterklasse

- Zelfde basisklasse (HelpDeskItem)
- Zelfde clientklasse (DemoAdapter)
- Zelfde interface (HelpDeskQueue):

```
public interface HelpDeskQueue {  
  
    void enqueue(HelpDeskItem helpDeskItem);  
  
    HelpDeskItem dequeue();  
  
    void overviewByPriority();  
  
    void overviewNatural();  
}
```



# Klasse-adapter (concreet voorbeeld)

de Adapter-  
klasse

```
public class Queue2ListAdapter
    extends ArrayList<HelpDeskItem>
    implements HelpDeskQueue {

    public void enqueue(HelpDeskItem helpDeskItem) {
        super.add(helpDeskItem); // achteraan toevoegen volgens FIFO
    }

    public HelpDeskItem dequeue() { //FIFI: vooraan verwijderen
        if(super.size() > 0) {return super.remove(0); }
        return null;
    }

    public void overviewByPriority() { // toon volgens priority
    }

    public void overviewNatural() { // toon in FIFO-volgorde
    }
}
```

Alles wordt vertaald naar de  
superklasse (hier: **ArrayList**)

# Object-adapter (concreet voorbeeld)

de Client-  
klasse

```
public class DemoAdapter {  
    static void main(String[] args) {  
        HelpDeskQueue myQueue = new Queue2ListAdapter();  
        myQueue.enqueue(new HelpDeskItem("Kan niet inloggen"));  
        myQueue.enqueue(new HelpDeskItem(5, "Server crash"));  
        myQueue.enqueue(new HelpDeskItem("Muis doet het niet"));  
        myQueue.enqueue(new HelpDeskItem(5, "Laptop in de fik!"));  
        myQueue.overviewNatural();  
        myQueue.overviewByPriority();  
    }  
}
```

De clientklasse gebruikt de  
interface van de adapter

Queue in natural order:

```
2016-10-18T13:55:24.146: 1 Kan niet inloggen  
2016-10-18T13:55:24.146: 5 Server crash  
2016-10-18T13:55:24.146: 1 Muis doet het niet  
2016-10-18T13:55:24.146: 5 Laptop in de fik!
```

Queue by priority:

```
2016-10-18T13:55:24.146: 5 Server crash  
2016-10-18T13:55:24.146: 5 Laptop in de fik!  
2016-10-18T13:55:24.146: 1 Kan niet inloggen  
2016-10-18T13:55:24.146: 1 Muis doet het niet
```

# Agenda

---

## 1. Adapter pattern

- Kenmerken en context
- Object adapter
- Class adapter



## 2. Composite pattern

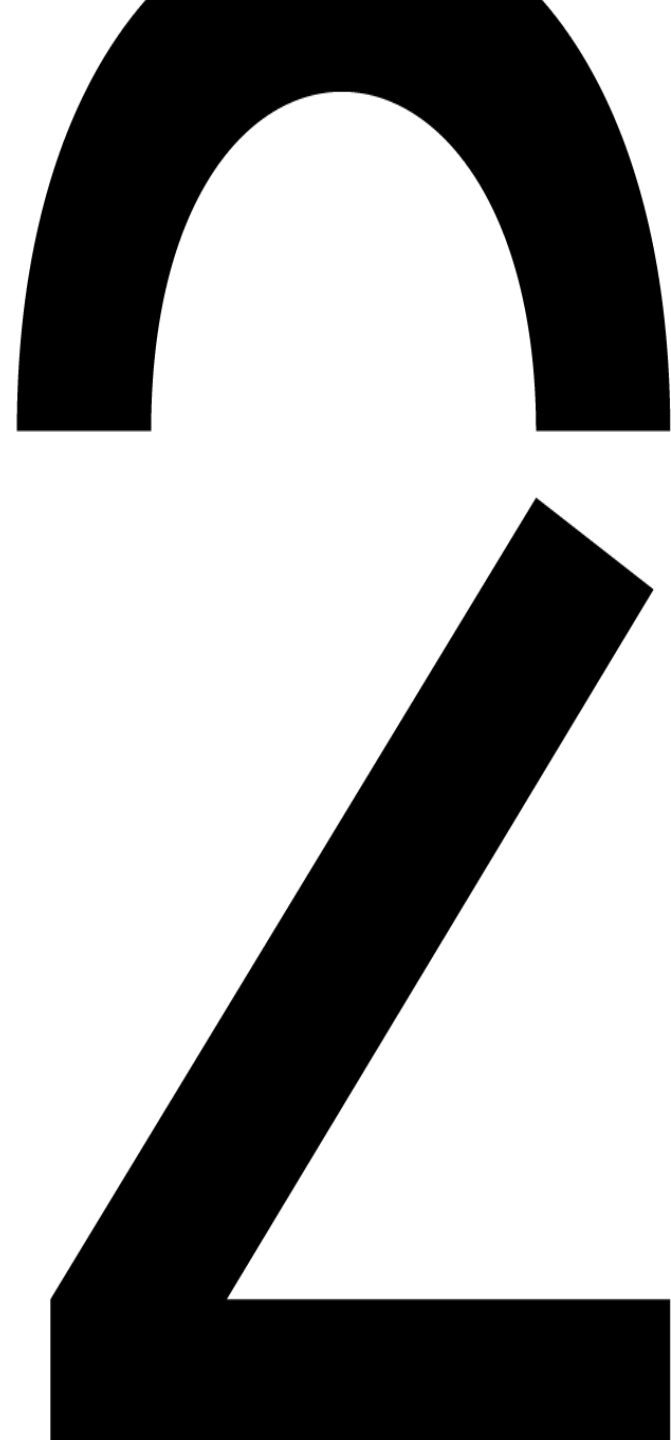
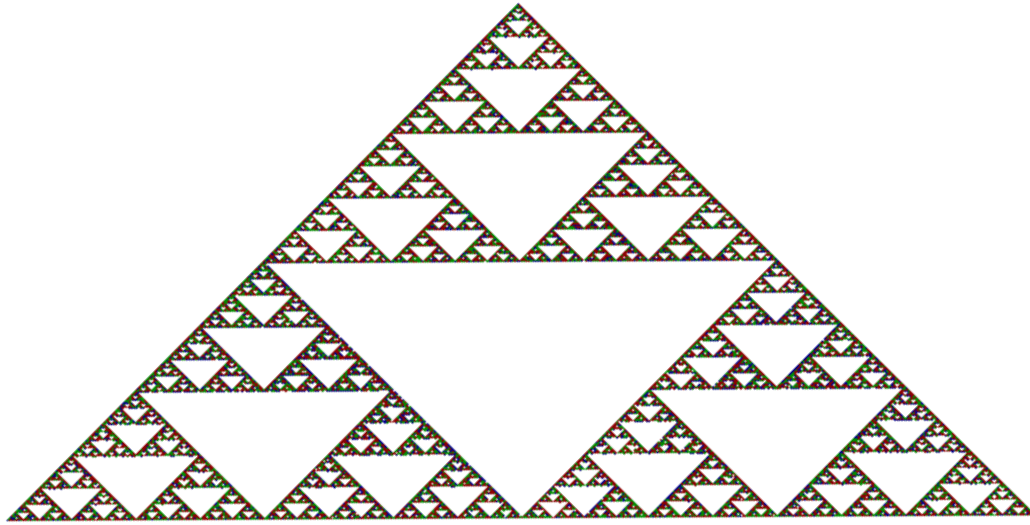
- Kenmerken
- Voorbeelden

## 3. Proxy pattern

- Kenmerken en oplossingsmodel
- Remote Proxy
- Virtual Proxy
- Protection Proxy

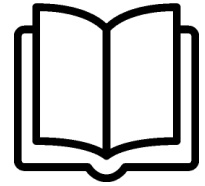
## 4. State pattern

- Kenmerken en context
- Voorbeelden



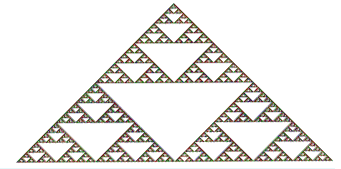
Composite



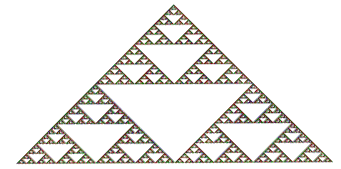


- E-book: "Composite" p.297 ev
  - Uit: "Applied Java Patterns", First Edition (Stephen Stelting and Olav Maassen)

# Composite: kenmerken



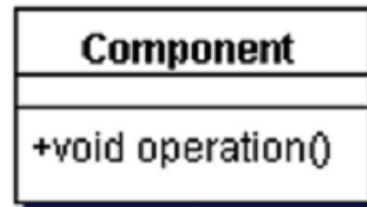
- Naam: **Composite** pattern [GoF95]  
(ook wel het Part-whole pattern genoemd.)
- Familie: **Structural** patterns
- Samenvatting:  
*GoF: "Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly."*
- Context:  
Deel-geheel hiërarchie waarin elk object:
  - ofwel een **verzameling** van elementen is
  - ofwel een **individueel** element is
- Voorbeeld:
  - tree structures, recursive compositions
  - `Javafx.scene.Node`
- Zie ook: *Chain of Responsibility, Flyweight*



- **Component:**
  - definieert gemeenschappelijke methoden voor alle elementen van de boomstructuur
  - = interface of abstracte klasse
- **Composite:**
  - implements of extends de Component
  - bevat een collection van componenten met methoden om toe te voegen, verwijderen, ...
- **Leaf:**
  - implements of extends de Component
  - bevat geen verwijzen naar andere componenten

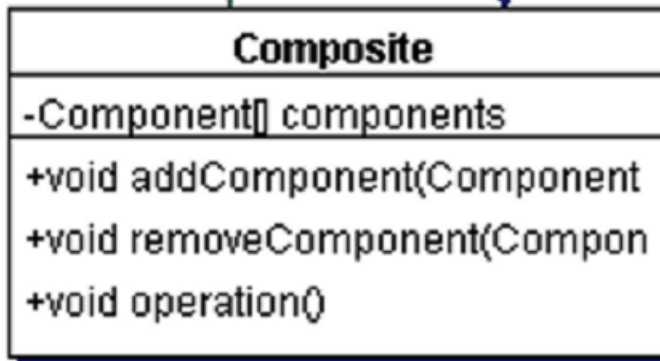
# Composite: oplossingsmodel

**abstracte klasse** of **interface** die de signature bepaalt voor zowel Composite als Leaf



implements of  
extends Component

Eindcomponent  
(**leaf**)



for every child in components  
child.operation()

bevat een **collection** van  
Component -  
objecten

# Voorbeeld (theoretisch)

---

```
public interface Component {  
    void operation();  
}
```

```
public class Part implements Component {  
    private String identifier;  
    public Part(String info) { this.identifier = info; }  
    public void operation() {  
        System.out.println("Part.operation() [info: "  
            + identifier "]" );  
    }  
}
```

## Voorbeeld (theoretisch) 2

---

```
public class Composite implements Component {
    private List<Component> parts = new ArrayList<>();

    public void operation() {
        for (Component component : parts) {
            component.operation();
        }
    }

    public void add(Component component) {
        parts.add(component);
    }

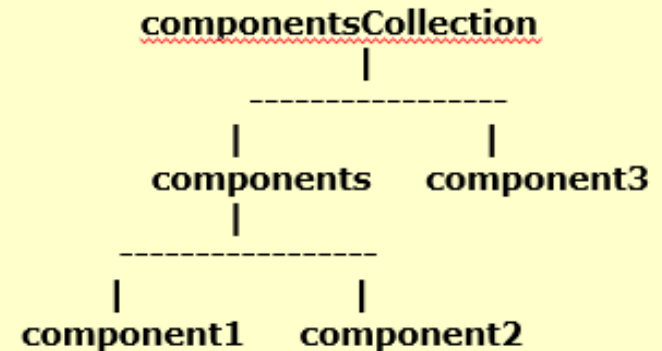
    public void remove(Component component) {
        parts.remove(component);
    }

    public Component getChild(int index) {
        return parts.get(index);
    }
}
```

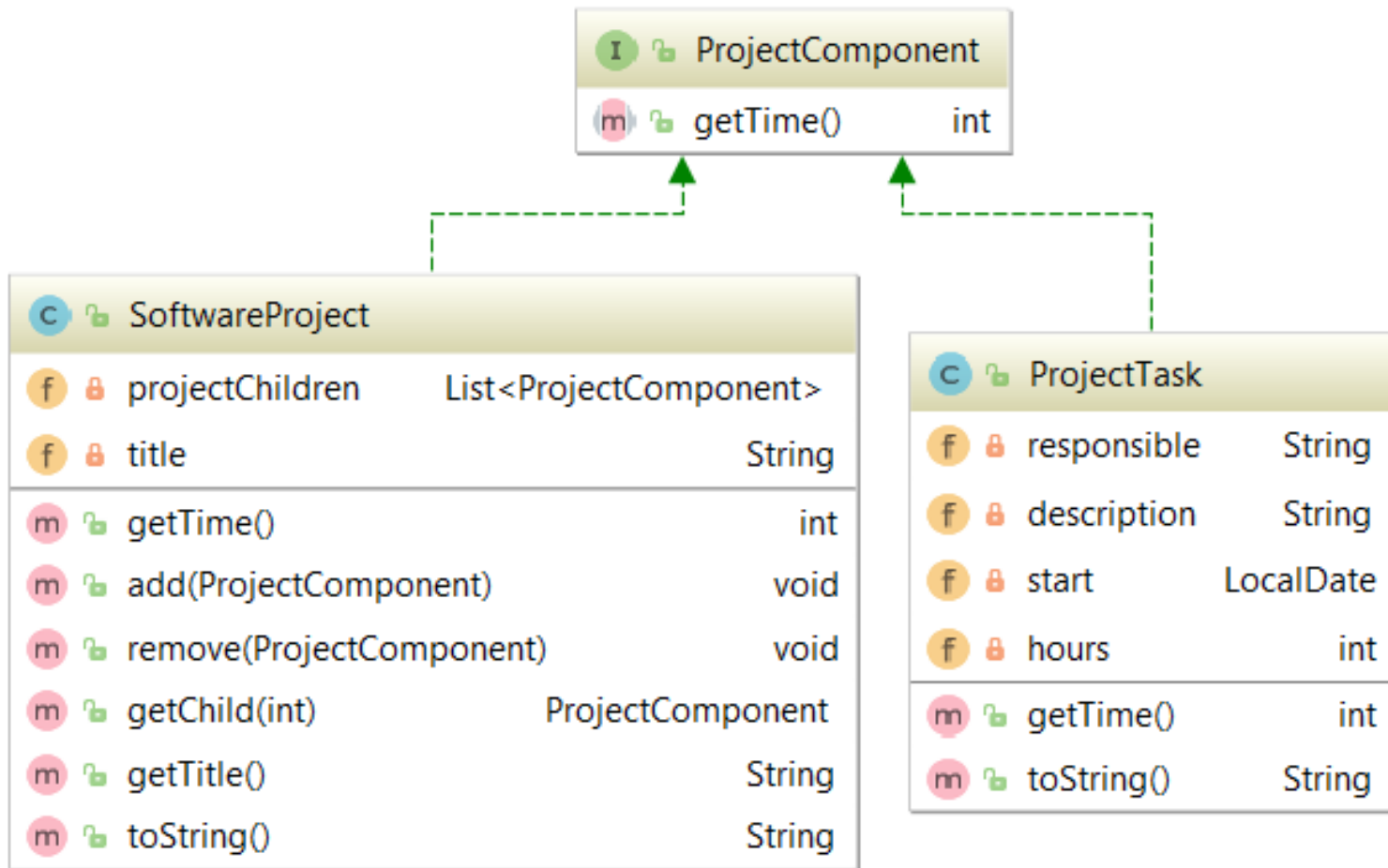
## Voorbeeld (theoretisch) 3

```
public class Client {  
    public static void main(String[] args) {  
        Component component1 = new Part("leaf 1");  
        Component component2 = new Part("leaf 2");  
        Component component3 = new Part("leaf 3");  
        Composite components = new Composite();  
        Composite componentsCollection = new Composite();  
        components.add(component1);  
        components.add(component2);  
        componentsCollection.add(components);  
        componentsCollection.add(component3);  
        componentsCollection.operation();  
    }  
}
```

Part.operation() [info:leaf 1]  
Part.operation() [info:leaf 2]  
Part.operation() [info:leaf 3]



# Composite: concreet voorbeeld





# Composite (concreet voorbeeld)

Gemeenschap-  
pelijke interface

```
public interface ProjectComponent {  
    int getTime();  
}
```

Gemeenschappelijke interface voor zowel de leaf-klasse (**ProjectTask**) als voor de composite-klasse (**SoftwareProject**)



# Composite (concreet voorbeeld)

de leaf-klasse

```
public class ProjectTask implements ProjectComponent {
    private String responsible;
    private String description;
    private LocalDate start;
    private int hours;

    public ProjectTask(String responsible, String description,
                       LocalDate start, int hours) {
        this.responsible = responsible;
        this.description = description;
        this.start = start;
        this.hours = hours;
    }

    public int getTime() {
        return hours;
    }

    @Override
    public String toString() {
        return String.format("%s (%s) --> %d hrs",
                             description, responsible, hours);
    }
}
```



# Composite (concreet voorbeeld)

de composite-  
klasse

```
public class SoftwareProject implements ProjectComponent {
    private List<ProjectComponent> projectChildren = new
    ArrayList<>();
    private String title;
    public SoftwareProject(String title) {this.title = title;}
    @Override
    public int getTime() {
        int totalTime = 0;
        for (ProjectComponent projectChild : projectChildren) {
            totalTime += projectChild.getTime();
        }
        return totalTime;
    }

    public void add(ProjectComponent component) {
        projectChildren.add(component);
    }

    public void remove(ProjectComponent component) {
        projectChildren.remove(component);
    }

    public ProjectComponent getChild(int index) {
        return projectChildren.get(index);
    }
    //enz...
```

Extra methoden voor  
beheer children

# Composite (concreet voorbeeld)

```
public class DemoComposite {
    public static void main(String[] args) {
        ProjectTask task1=new ProjectTask("Mark",
            "Schrijf JUnit testen", LocalDate.of(2016, 10, 10), 16);
        ProjectTask task2 = new ProjectTask("Linda", "Webservices"
, LocalDate.of(2016, 10, 5), 4);
        ProjectTask task3 = new ProjectTask("Freddy",
            "Website frontend", LocalDate.of(2016, 10, 15), 64);
        SoftwareProject subProject =
            new SoftwareProject("Web applicatie Stad Antwerpen");
        subProject.add(task1);
        subProject.add(task2);
        subProject.add(task3);
        SoftwareProject masterProject =
            new SoftwareProject("Project Stad Antwerpen");
        masterProject.add(subProject);
        masterProject.add(new ProjectTask("Nancy", "Offerte opmaken"
, LocalDate.of(2016, 11, 5), 4));
        System.out.println(masterProject);
        System.out.println("Totaal begroot: " + masterProject.getTime()
                                + " uren");
    }
}
```

# Composite (concreet voorbeeld)

```
public class DemoComposite {  
    public static void main(String[] args) {  
        ProjectTask task1=new ProjectTask("Mark",  
            "Schrijf JUnit testen", LocalDate.of(2016, 10, 10), 16);  
        ProjectTask task2 = new ProjectTask("Linda", "Webservices"  
        , LocalDate.of(2016, 10, 5), 4);  
        ProjectTask task3 = new ProjectTask("Freddy",  
            "Website frontend", LocalDate.of(2016, 10, 15), 64);  
        SoftwareProject subProject =  
            new SoftwareProject("Web applicatie Stad Antwerpen");  
        subProject.add(task1);  
        subProject.add(task2);  
        subProject.add(task3);  
        SoftwareProject masterProject =  
            new SoftwareProject("Project Stad Antwerpen");  
        masterProject.add(subProject);  
        masterProject.add(new ProjectTask("Nancy", "Offerte opmaken",  
        , LocalDate.of(2016, 11, 1), 4);  
        System.out.println(masterProject);  
        System.out.println("Totaal begroot: 88 uren");  
    }  
}
```

Project Stad Antwerpen:  
[Web applicatie Stad Antwerpen:  
[JUnit tests schrijven (Mark) --> 16 hrs,  
Webservices (Linda) --> 4 hrs,  
Website frontend (Freddy) --> 64 hrs],  
Offerte opmaken (Nancy) --> 4 hrs]  
Totaal begroot: 88 uren

# Opdrachten

---



- Opdrachten op BB:

➔ Adapter-Lichamen

➤ Adapter-Duo

➔ Composite-Bestanden

➤ Composite-Car

Vermits het groeiproject deze week enkel over Proxy gaat, raad ik je sterk aan om ook andere oefeningen van BB te maken!

# Agenda

---

## 1. Adapter pattern

- Kenmerken en context
- Object adapter
- Class adapter



## 2. Composite pattern

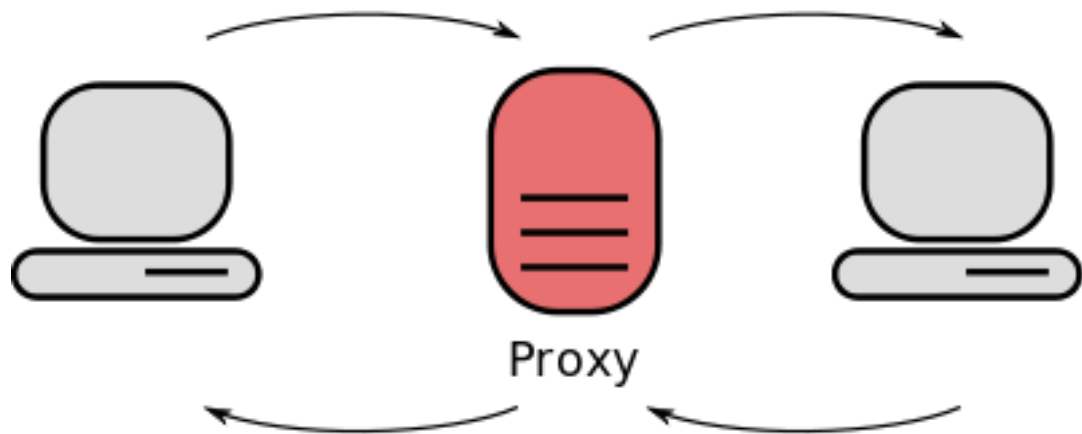
- Kenmerken
- Voorbeelden

## 3. Proxy pattern

- Kenmerken en oplossingsmodel
- Remote Proxy
- Virtual Proxy
- Protection Proxy

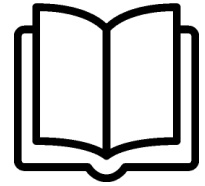
## 4. State pattern

- Kenmerken en context
- Voorbeelden



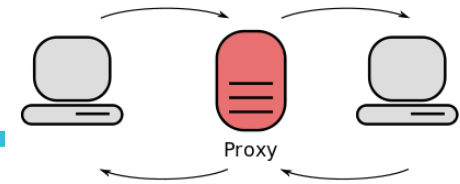
**Proxy**





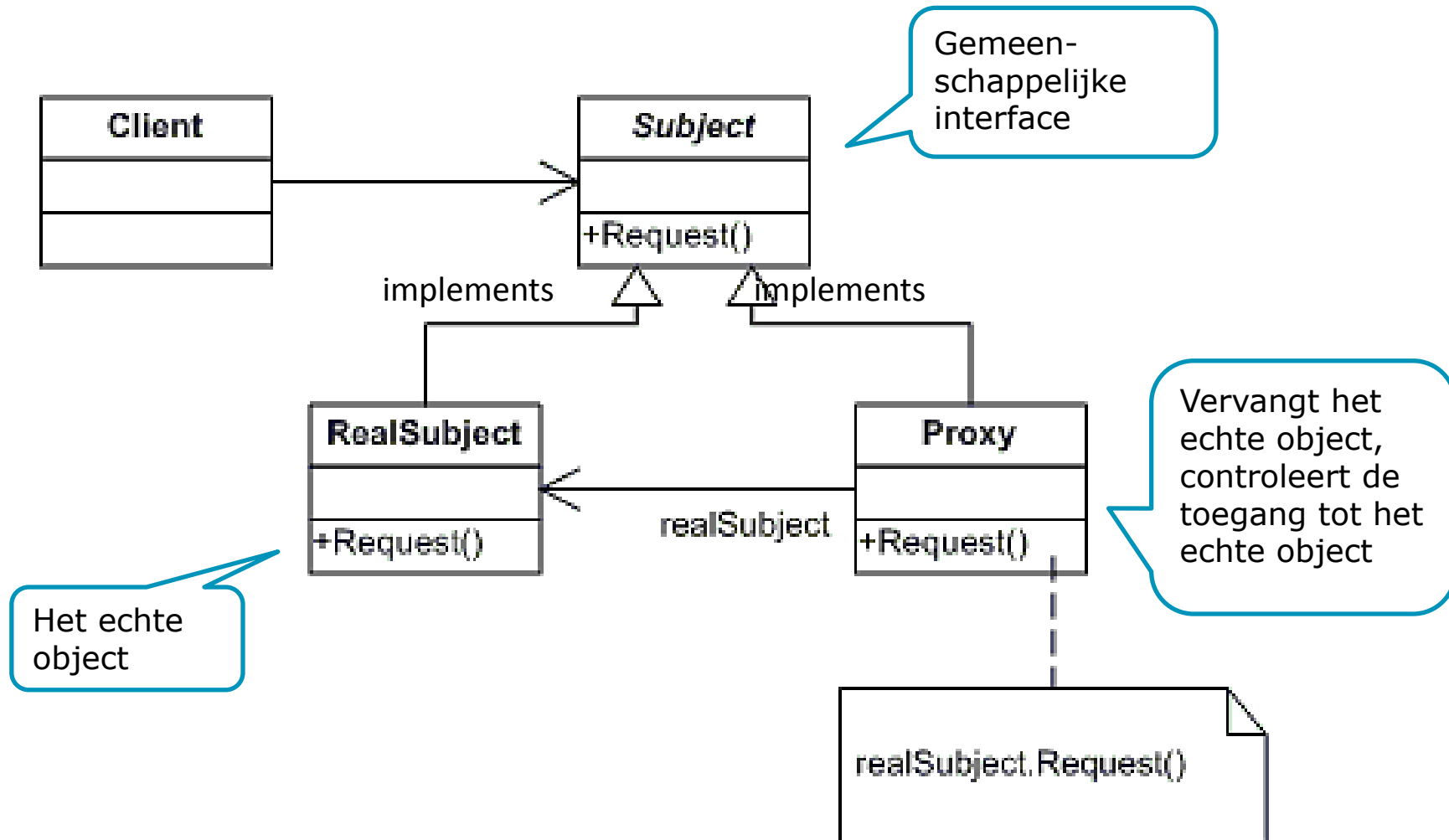
- E-book: "Proxy" p.307 ev
  - Uit: "Applied Java Patterns", First Edition (Stephen Stelting and Olav Maassen)

# Proxy: kenmerken

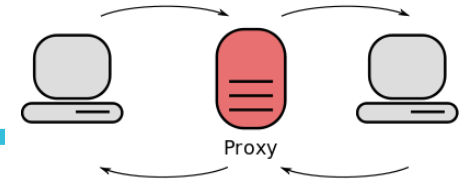


- Naam: **Proxy** pattern [GoF95]  
(ook wel het Surrogate pattern genoemd.)
- Familie: **Structural** patterns
- Samenvatting:  
GoF: *"Provides a surrogate or placeholder for another object to control access to it."*
- Voorbeelden:
  - Om toegang tot een object te krijgen zonder dat het object zelf al moet bestaan
  - Om bepaalde acties te controleren of af te schermen
  - Om logging op een object te voorzien
- Zie ook:  
Adapter, LazyInstantiation, Caretaker, Decorator

# Proxy: oplossingsmodel

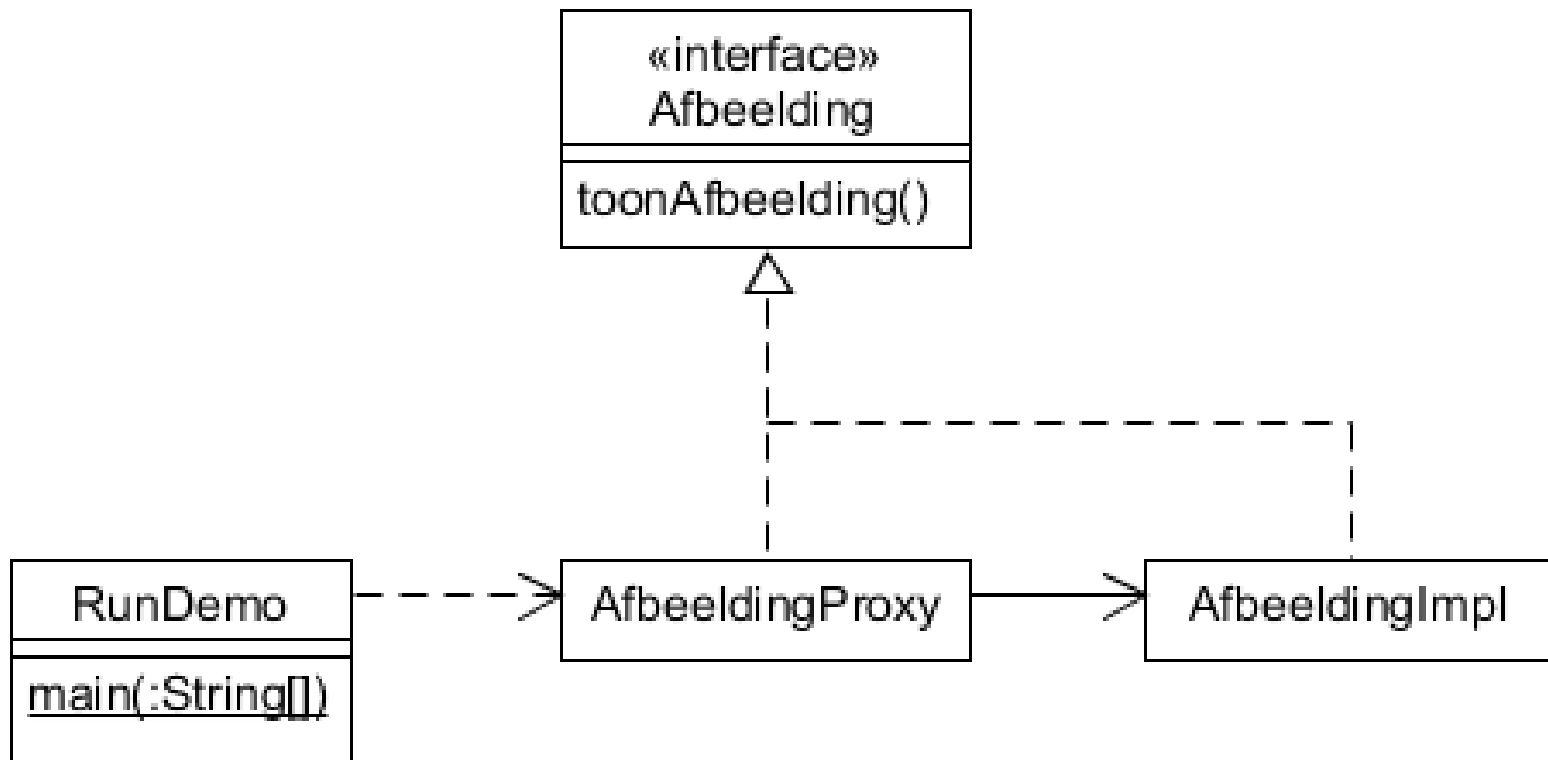


# Proxy: context



- Toepasbaar als:
  - **remote** proxy  
→ als je een lokale representatie wil voor een remote object
  - **virtual** proxy  
→ gedraagt zich als een dubbelganger en stelt de creatie van complexe (expensive) objects uit
  - **protection** proxy  
→ controleert de toegang tot het werkelijke object
- Voorbeelden:
  - cache proxy
  - firewall proxy
  - synchronization proxy
  - ...

# Virtual Proxy: voorbeeld



# Virtual Proxy: voorbeeld

---

```
public interface Afbeelding {  
    public void toonAfbeelding();  
}
```

```
public class AfbeeldingProxy implements Afbeelding {  
    private String filename;  
    private Afbeelding afbeelding; { // real subject caching  
    public AfbeeldingProxy(String filename) {  
        this.filename=filename;  
    }  
    public void toonAfbeelding() {  
        if (afbeelding == null) { // alleen ophalen indien nodig  
            afbeelding = new AfbeeldingImpl(filename);  
        }  
        afbeelding.toonAfbeelding();  
    }  
}
```

# Virtual Proxy: voorbeeld

---

```
public class AfbeeldingImpl implements Afbeelding {
    private String bestandsnaam;
    public AfbeeldingImpl(String bestandsnaam) {
        this.bestandsnaam = bestandsnaam;
        haalAfbeeldingOp();
    }
    private void haalAfbeeldingOp() {
        // Foto van harde schijf lezen of downloaden van server
        // ...
        System.out.println("Ophalen van " + bestandsnaam);
    }
    public void toonAfbeelding() {
        System.out.println("Tonen van " + bestandsnaam);
    }
}
```

# Virtual Proxy: voorbeeld

```
public class RunDemo {  
    public static void main(String[] args) {  
        List<Afbeelding> fotos = new ArrayList<Afbeelding>();  
        fotos.add(new AfbeeldingProxy("50MB_Foto1"));  
        fotos.add(new AfbeeldingProxy("50MB_Foto2"));  
        fotos.add(new AfbeeldingProxy("50MB_Foto3"));  
        // foto3 nooit opgehaald (tijd bespaard)  
        fotos.get(0).toonAfbeelding(); // ophalen noodzakelijk  
        fotos.get(1).toonAfbeelding(); // ophalen noodzakelijk  
        fotos.get(0).toonAfbeelding(); // gecached (tijd bespaard)  
    }  
}
```

```
Ophalen van 50MB_Foto1  
Tonen van 50MB_Foto1  
Ophalen van 50MB_Foto2  
Tonen van 50MB_Foto2  
Tonen van 50MB_Foto1
```





# Protection Proxy in Collections framework

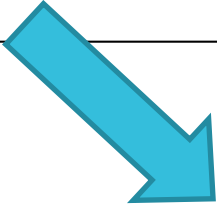
- Collections.**unmodifiableXXX** methoden
  - Er wordt intern een proxy gecreëerd die de originele collection afschermt zodat de inhoud ervan niet meer kan wijzigen.
- Codevoorbeeld:

```
List<Integer> getallen = new ArrayList<>();  
    for (int i = 0; i < 10; i++) {  
        getallen.add(i);  
    }  
  
List<Integer> cijfers = Collections.unmodifiableList(getallen);  
    getallen.set(3, 10); // Ok  
    cijfers.set(3, 10); // Hier loopt het mis!
```

**UnsupportedOperationException**

# Protection Proxy in Collections framework

```
//...  
public List getList() {  
    return Collections.unmodifiableList(myList);  
}
```



```
public class UnmodifiableList implements List {  
    final List<Object> list;  
    UnmodifiableList(List<Object>list){this.list = list;}  
    public Object get(int index) {return list.get(index);}  
    public Object set(int index, Object element) {  
        throw new UnsupportedOperationException();  
    }  
    public void add(int index, Object element) {  
        throw new UnsupportedOperationException();  
    }  
    public int indexOf(Object o) {return list.indexOf(o);}  
    //...
```



# Protection Proxy: voorbeeld

---

```
public interface Klant {  
    String getRekeningNummer();  
}
```

```
public class KlantImpl implements Klant {  
    private String rekeningNummer = "BE97-0123-4567-8901";  
  
    public KlantImpl() {  
        System.out.println("KlantImpl: creatie klant");  
    }  
  
    public String getRekeningNummer() {  
        System.out.println("KlantImpl: rekeningNummer =" + rekeningNummer);  
        return rekeningNummer;  
    }  
}
```


# Protection Proxy: voorbeeld

---

```
public class ProtectionProxy implements Klant {
    private String wachtwoord;
    private KlantImpl klant;

    public ProtectionProxy(String wachtwoord) {
        this.wachtwoord = wachtwoord;
        klant = new KlantImpl();
        System.out.println("ProtectionProxy: creatie proxy");
    }

    public String getRekeningNummer() {
        System.out.print("Wachtwoord: ");
        Scanner scanner = new Scanner(System.in);
        String temp = scanner.nextLine();
        if(wachtwoord.equals(temp)){return klant.getRekeningNummer();}
        else{System.out.println("ProtectionProxy: fout wachtwoord");}
        return "";
    }
}
```



# Protection Proxy: voorbeeld

```
public class ProtectionDemo {  
  
    public static void main(String[] args) {  
  
        Klant klant = new ProtectionProxy("bla");  
  
        System.out.println("Main ontvangt: "  
            + klant.getRekeningNummer());  
  
    }  
  
}
```

Nieuwe run

```
KlantImpl: creatie klant  
ProtectionProxy: creatie proxy  
Wachtwoord: bla  
KlantImpl: rekeningNummer = BE97-0123-4567-8901  
Main ontvangt: BE97-0123-4567-8901
```

```
KlantImpl: creatie klant  
ProtectionProxy: creatie proxy  
Wachtwoord: boe  
ProtectionProxy: fout wachtwoord  
Main ontvangt:
```



# Agenda

---

## 1. Adapter pattern

- Kenmerken en context
- Object adapter
- Class adapter



## 2. Composite pattern

- Kenmerken
- Voorbeelden

## 3. Proxy pattern

- Kenmerken en oplossingsmodel
- Remote Proxy
- Virtual Proxy
- Protection Proxy

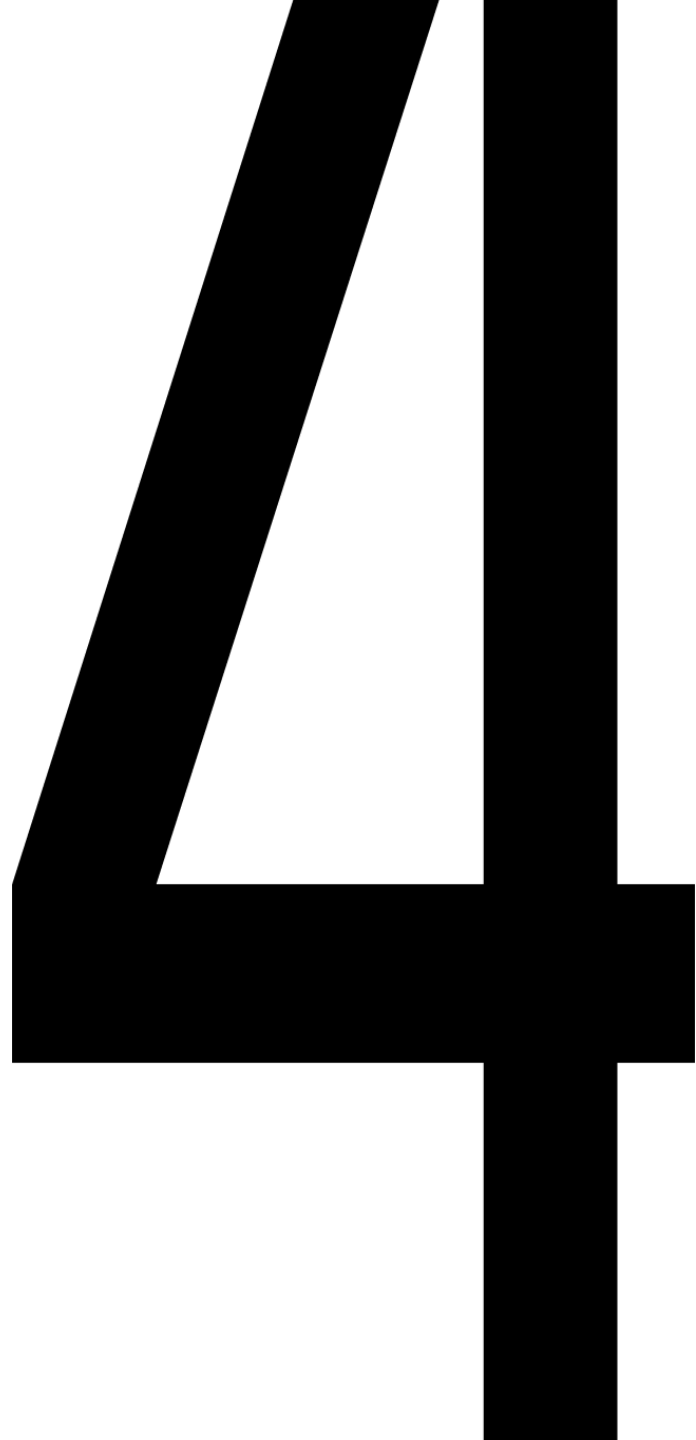
## 4. State pattern

- Kenmerken en context
- Voorbeelden

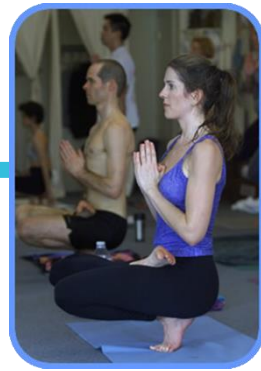


---

State



# State: kenmerken



- Naam: **State** pattern[GoF95]  
(Object for states)
- Familie: **Behavioral** patterns
- Samenvatting:

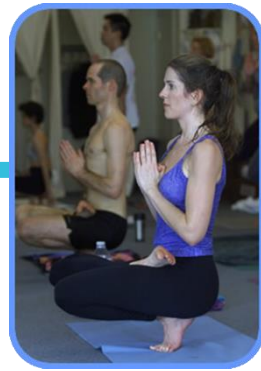
GoF: *"Allow an object to alter its behavior when its internal state changes. The object will appear to change its class."*

- Het State pattern laat toe om het volledige gedrag van een object te wijzigen afhankelijk van de toestand van één of meer attributen.



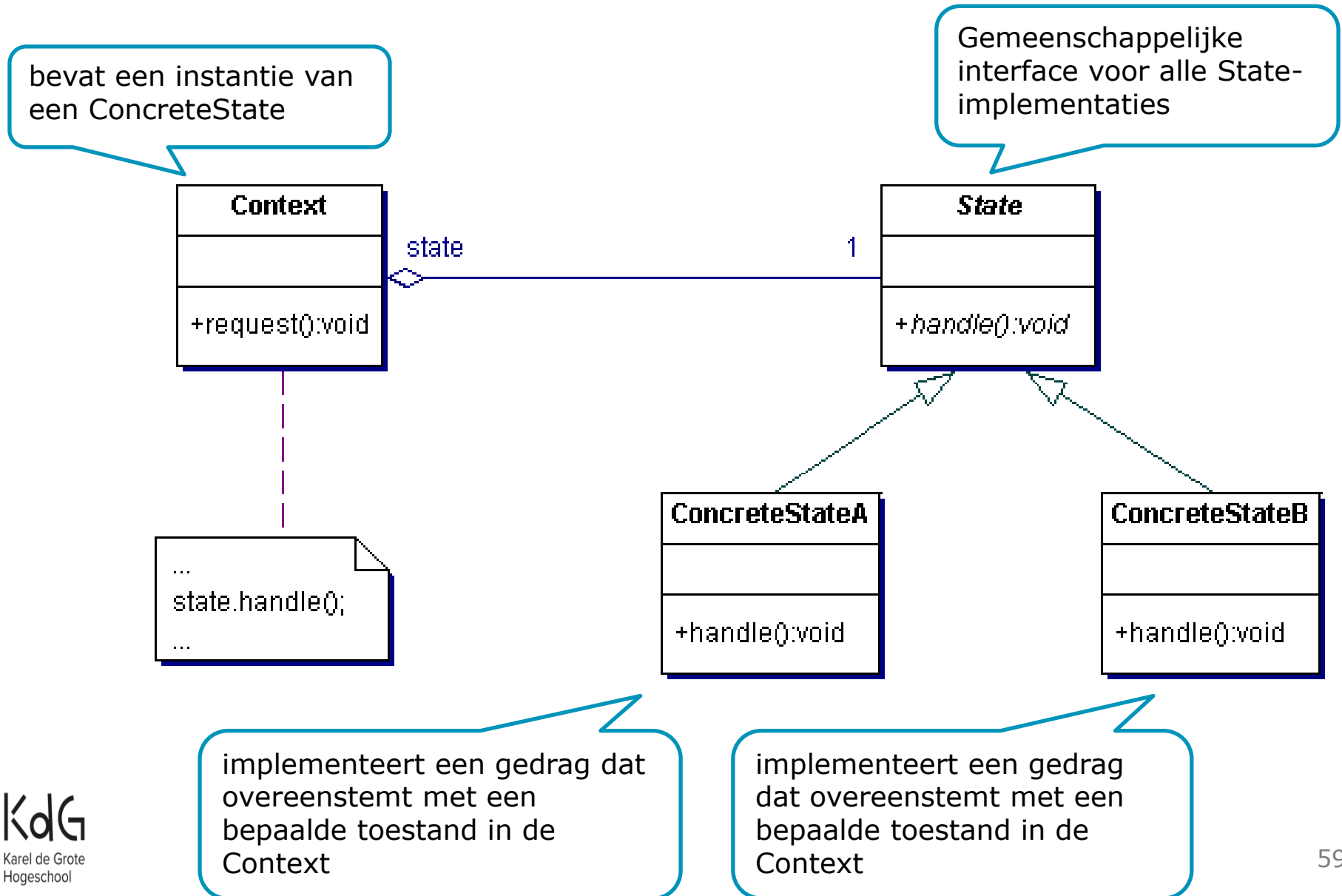
## State: context

---



- Gebruik het **state pattern** als:
  - ... het gedrag van een object afhankelijk is van zijn toestand en de toestand regelmatig wijzigt
  - ... je code sterk afhankelijk is van de toestand van het object
  - ... je veel conditional statements gebruikt om die toestand te testen (switch cases)

# State: Oplossingsmodel



# Voorbeeld (theoretisch)

---

```
public class Context {  
    public static final int STATE_ONE = 0;  
    public static final int STATE_TWO = 1;  
    private State currentState = new ConcreteState1();  
    public void request() { currentState.handle(); }  
    public void changeState(int state) {  
        switch (state) {  
            case STATE_ONE:  
                currentState = new ConcreteState1();  
                break;  
            case STATE_TWO:  
                currentState = new ConcreteState2();  
        }  
    }  
}
```

# Voorbeeld (theoretisch)

```
public interface State {  
    public void handle();  
}
```

```
public class ConcreteState1 implements State {  
    public void handle() {  
        System.out.println("ConcreteState1.handle() executing");  
    }  
}
```

```
public class ConcreteState2 implements State {  
    public void handle() {  
        System.out.println("ConcreteState2.handle() executing");  
    }  
}
```

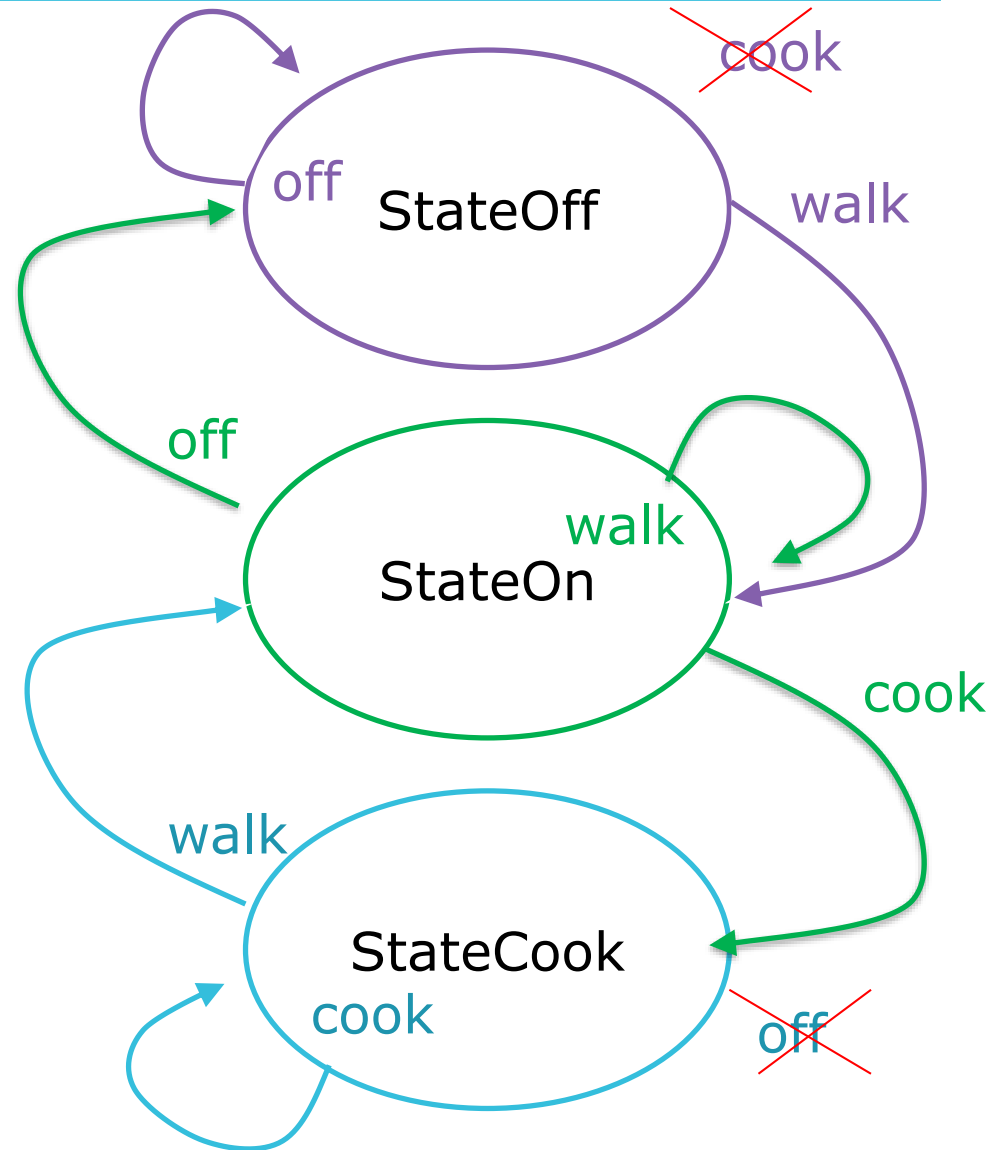
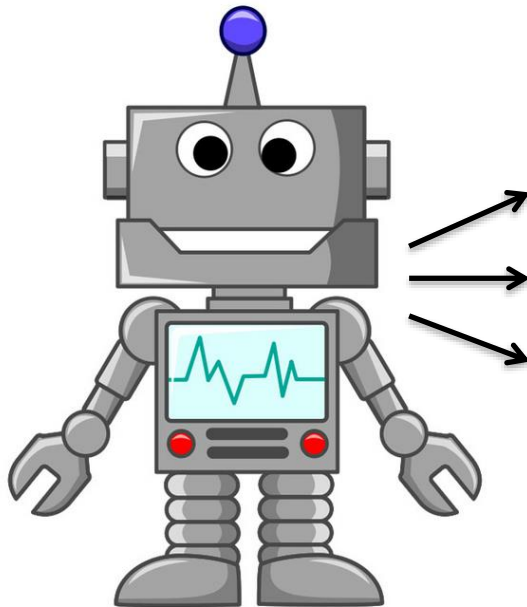
## Voorbeeld (theoretisch)

---

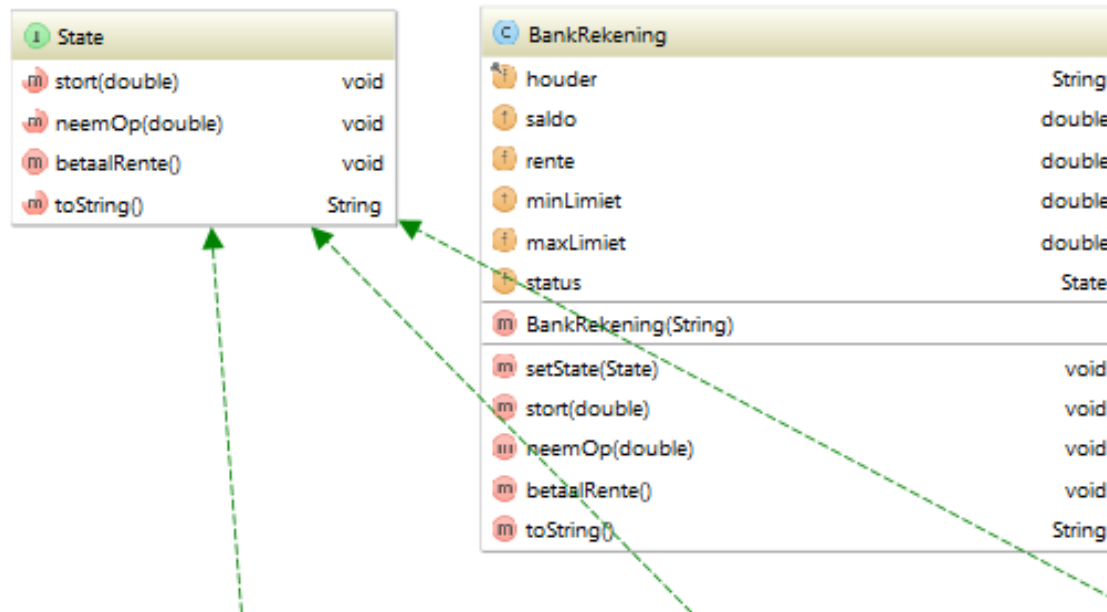
```
public class Client {  
    public static void main(String[] args) {  
        // construct context  
        Context context = new Context();  
  
        // call request  
        context.request();  
  
        // change the state  
        context.changeState(Context.STATE_TWO);  
  
        // call request  
        context.request();  
    }  
}
```



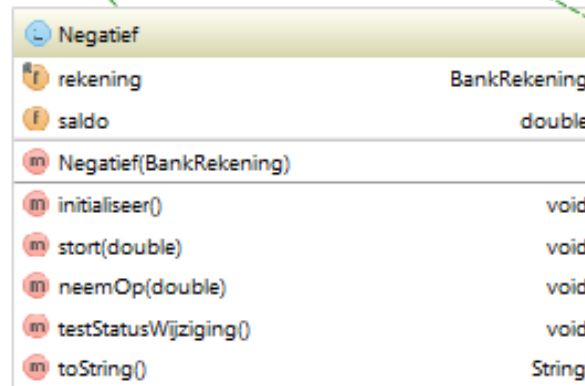
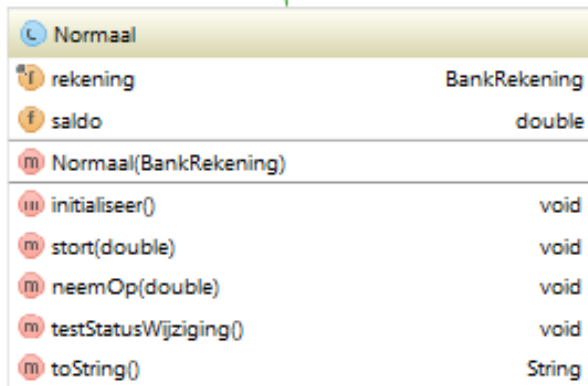
# Demo: Robot



# Voorbeeld (Praktisch)



Een aantal getters en setters in de klasse BankRekening werden in de volgende slides weggelaten...



## Voorbeeld (Praktisch)

```
public class BankRekening {
    private final String houder;
    private double saldo;
    private double rente;
    private double minLimiet;
    private double maxLimiet;
    private State status;
    public BankRekening(String
        houder) {
        this.houder = houder;
        status = new Normaal(this);
    }
    // ... + getters en setters
    // o.a. setState(State status)
}
```

```
public void stort(double bedrag) {
    status.stort(bedrag);
}
public void neemOp(double bedrag) {
    status.neemOp(bedrag);
}
public void betaalRente() {
    status.betaalRente();
}
public String toString() {
    return "Saldo: " + status.saldo
        + " Status: " + status;
}
}
```



# Voorbeeld (Praktisch)

---

```
public interface State {  
    void stort(double bedrag);  
  
    void neemOp(double bedrag);  
  
    default void betaalRente() {  
        // Er wordt geen rente betaald  
    };  
  
    String toString();  
}
```

# Voorbeeld (Praktisch)

```
public class Normaal implements State {
    private final BankRekening rekening;
    private double saldo;

    public Normaal(BankRekening
                    rekening) {
        saldo = rekening.getSaldo();
        this.rekening = rekening;
        initialiseer();
    }

    void initialiseer() {
        rekening.setRente(0.0);
        rekening.setMinLimiet(0.0);
        rekening.setMaxLimiet(1000.0);
    }

    public void stort(double bedrag) {
        saldo += bedrag;
        rekening.setSaldo(saldo);
        testStatusWijziging();
    }
}
```

```
    public void neemOp(double bedrag) {
        saldo -= bedrag;
        rekening.setSaldo(saldo);
        testStatusWijziging();
    }

    private void testStatusWijziging() {
        if (saldo <
            rekening.getMinLimiet()) {
            rekening.setState(
                new Negatief(rekening));
        } else if (saldo >
            rekening.getMaxLimiet()) {
            rekening.setState(
                new MetRente(rekening));
        }
    }

    public String toString() {
        return "Normaal";
    }
}
```

# Voorbeeld (Praktisch)

```
public class Negatief implements State {
    private final BankRekening rekening;
    private double saldo;

    public Negatief(BankRekening rekening) {
        saldo = rekening.getSaldo();
        this.rekening = rekening;
        initialiseer();
    }

    void initialiseer() {
        rekening.setRente(0.0);
        rekening.setMinLimiet(-500.0);
        rekening.setMaxLimiet(1000.0);
    }

    public void stort(double bedrag) {
        saldo += bedrag;
        rekening.setSaldo(saldo);
        testStatusWijziging();
    }
}
```

```
public void neemOp(double bedrag) {
    System.out.println("Geen opname
                        mogelijk!");
}

private void testStatusWijziging() {
    if (saldo > 0.0 &&
        saldo < rekening.getMaxLimiet()) {
        rekening.setState(
            new Normaal(rekening));
    } else if (saldo >
               rekening.getMaxLimiet())
    {
        rekening.setState(
            new MetRente(rekening));
    }
}

public String toString() {
    return "Negatief";
}
}
```

# Voorbeeld (Praktisch)

```
public class MetRente implements State {
    private final BankRekening rekening;
    private double saldo;

    public MetRente(BankRekening rekening) {
        saldo = rekening.getSaldo();
        this.rekening = rekening;
        initialiseer();
    }

    void initialiseer() {
        rekening.setRente(0.005);
        rekening.setMinLimiet(1000.0);
        rekening.setMaxLimiet(1e10);
    }

    public void stort(double bedrag) {
        saldo += bedrag;
        rekening.setSaldo(saldo);
    }
}
```

```
    public void neemOp(double bedrag) {
        saldo -= bedrag;
        rekening.setSaldo(saldo);
        testStatusWijziging();
    }

    @Override
    public void betaalRente() {
        saldo += rekening.getRente() *
                                   saldo;
        rekening.setSaldo(saldo);
    }

    private void testStatusWijziging(){
        if (saldo < 0.0) {
            rekening.setState(new
                               Negatief(rekening));
        } else if (saldo <
                    rekening.getMinLimiet())
            rekening.setState(new
                               Normaal(rekening));
        }
    }

    public String toString() {
        return "Met Rente";
    }
}
```

# Voorbeeld (Praktisch)

```
public class DemoState {  
    public static void main(String[] args) {  
        BankRekening rekening = new BankRekening("Jos The Boss");  
        System.out.printf("Bankrekening van %s%n", rekening.getHouder());  
        rekening.stort(500);  
        System.out.println(rekening);  
        rekening.stort(850);  
        System.out.println(rekening);  
        rekening.betaalRente();  
        System.out.println(rekening);  
        rekening.neemOp(1100);  
        System.out.println(rekening);  
        rekening.neemOp(500);  
        System.out.println(rekening);  
        rekening.neemOp(500);  
        System.out.println(rekening);  
    }  
}
```

```
Bankrekening van Jos The Boss  
Saldo: 500.0   Status: Normaal  
Saldo: 1350.0  Status: Met Rente  
Saldo: 1356.75 Status: Met Rente  
Saldo: 256.75  Status: Normaal  
Saldo: -243.25 Status: Negatief  
Geen opname mogelijk!  
Saldo: -243.25 Status: Negatief
```



Democode: 10\_StateVoorbeeld

# Opdrachten

---



- Groeiproject

- module 5  
(deel 4: "Proxy pattern")



- Opdrachten op BB

- ➔ Adapter-Lichamen
  - Adapter-Duo
- ➔ Composite-Bestanden
  - Composite-Car
- ➔ Proxy-Factuur
  - State-Simple
- ➔ Craps met Proxy – State – Factory



Vermits het groeiproject deze week enkel over Proxy gaat, raad ik je sterk aan om ook andere oefeningen van BB te maken!