

5 Design Patterns (deel 1)

Programmeren 2 – Java

2017 - 2018

KdG Karel de Grote
Hogeschool

Kris Behiels
Jan De Rijke
Mark Goovaerts

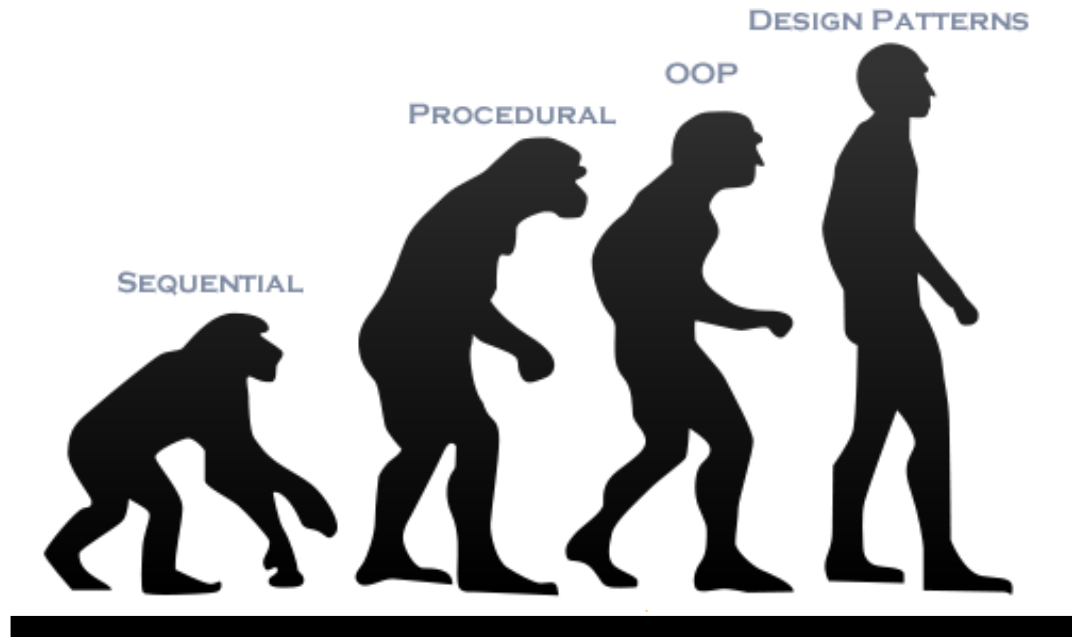
Programmeren 2 - Java

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





Inleiding patterns

Agenda

1. Inleiding patterns

- Wat is een design pattern?
- Soorten patterns
- Voordelen



2. Singleton

- Kenmerken
- Singleton (Lazy initialization, klassiek, enum)

3. Observer

- Kenmerken en vergelijking MVP
- Voorbeeld
- Probleem (overerving/delegatie)

4. Static Factory

- Kenmerken
- Voorbeelden
- Voordelen / nadelen

Wat is een design pattern?

- Een beschrijving van een **algemene oplossing** voor een ontwerpprobleem.
- Los van de programmeertaal.
- Bevat gewoonlijk meer dan één klasse, geeft aan welke methoden de klassen moeten bieden en hoe de objecten samenhangen en samenwerken.
- Bij gebruik van een pattern moet de ontwerper de algemene oplossing vertalen in de context van het specifieke probleem en de implementatie-omgeving.
- Patterns hebben de evolutie van talen en bibliotheken beïnvloed

Hoe ziet een pattern eruit?

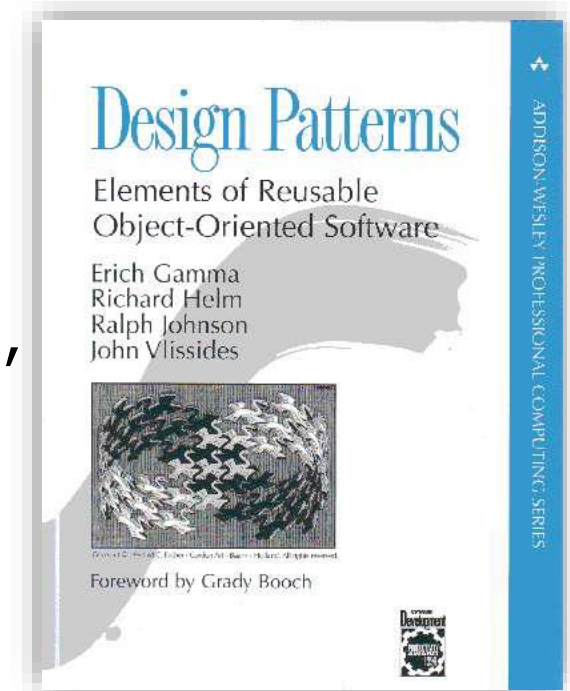
- Elk pattern heeft een **naam** en is beschreven volgens een vast schema:
 - ✓ de **formulering** van het probleem;
 - ✓ de **context** waarin het pattern relevant is;
 - ✓ de **oplossing**, met argumentatie en overwegingen voor de uitwerking;
 - ✓ de **gevolgen** van het pattern (met verwijzingen naar andere relevante patterns).

Standaardwerk

Design Patterns

Elements of Reusable Object-Oriented Software

- Gamma, Helm, Johnson, Vlissides
(GoF = Gang of Four)
- 1995 (voorbeelden in C++)
- Geeft een beschrijving van 23 patterns,
ingedeeld in 3 categorieën:
 - Creational (o.a. Singleton, Abstract Factory)
 - Structural (o.a. Adapter, Composite, Proxy)
 - Behavioral (o.a. Observer, State)



Soorten patterns

Creational

Behavioral

Structural

107 FM Factory Method							139 A Adapter
117 PT Prototype	127 S Singleton				123 CR Chain of Responsibility	163 CP Composite	175 D Decorator
87 AF Abstract Factory	325 TM Template Method	233 CD Command	173 MD Mediator	293 O Observer	243 IN Interpreter	207 PX Proxy	185 FA Façade
97 BU Builder	315 SR Strategy	283 MM Memento	305 ST State	257 IT Iterator	331 V Visitor	195 FL Flyweight	151 BR Bridge

Voordelen

- Bepaalde onderdelen van een ontwerp kan je aanduiden met een pattern-naam
→ **hoger abstractie-niveau**
- Patterns vergemakkelijken en bevorderen de **communicatie** tussen software-ontwikkelaars.
- Patterns worden opzettelijk "vaag", onvolledig gespecificeerd en **taalafhankelijk** beschreven, toch hebben ze betrekking op concrete software-problemen.

Voordelen (vervolg)

- Patterns zijn geen vervanging voor reguliere ontwerp- en programmeervaardigheden. Ze bieden een **mogelijke oplossing** voor een probleem, maar de feitelijke uitwerking blijft aan de ontwikkelaar.
- Patterns ontstaan uit de **ervaringen** van ontwikkelaars. Zij herkennen in probleemstellingen en specificaties deelproblemen die regelmatig terugkeren. Een eerder bedachte, goed werkende oplossing levert vaak een bruikbaar pattern op.

Behandelde patterns:



Singleton



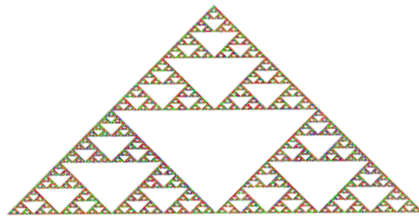
Observer



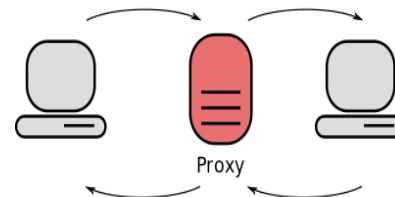
Static Factory



Adapter



Composite



Proxy



State

Agenda

1. Inleiding patterns

- Wat is een design pattern?
- Soorten patterns
- Voordelen



2. Singleton

- Kenmerken
- Singleton (Lazy initialization, klassiek, enum)

3. Observer

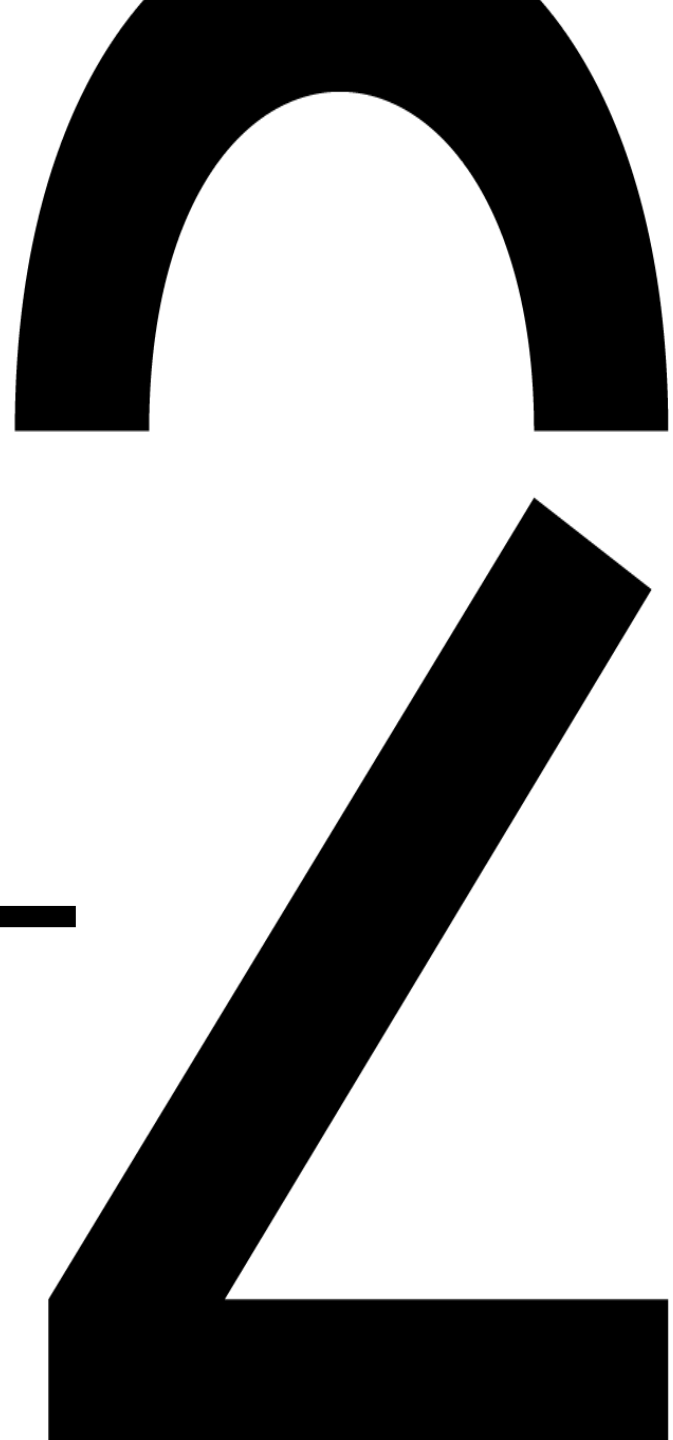
- Kenmerken en vergelijking MVP
- Voorbeeld
- Probleem (overerving/delegatie)

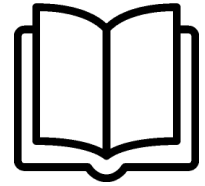
4. Static Factory

- Kenmerken
- Voorbeelden
- Voordelen / nadelen



Singleton





- E-book: "Singleton" p.264 ev
 - Uit: "Applied Java Patterns", First Edition (Stephen Stelting and Olav Maassen)

Singleton: kenmerken



- Naam: **Singleton** pattern [GoF95]
- Familie: Creational patterns
- Samenvatting:

GoF: *"Ensure a class only has one instance, and provide a global point of access to it."*

- Context:

Wanneer er slechts één instantie van een klasse mag gemaakt worden.

- Voorbeeld:

- ✓ connection, session
- ✓ een specifieke modelklasse
- ✓ `Java.lang.Runtime#getRuntime()`

- Zie ook: *Factory*

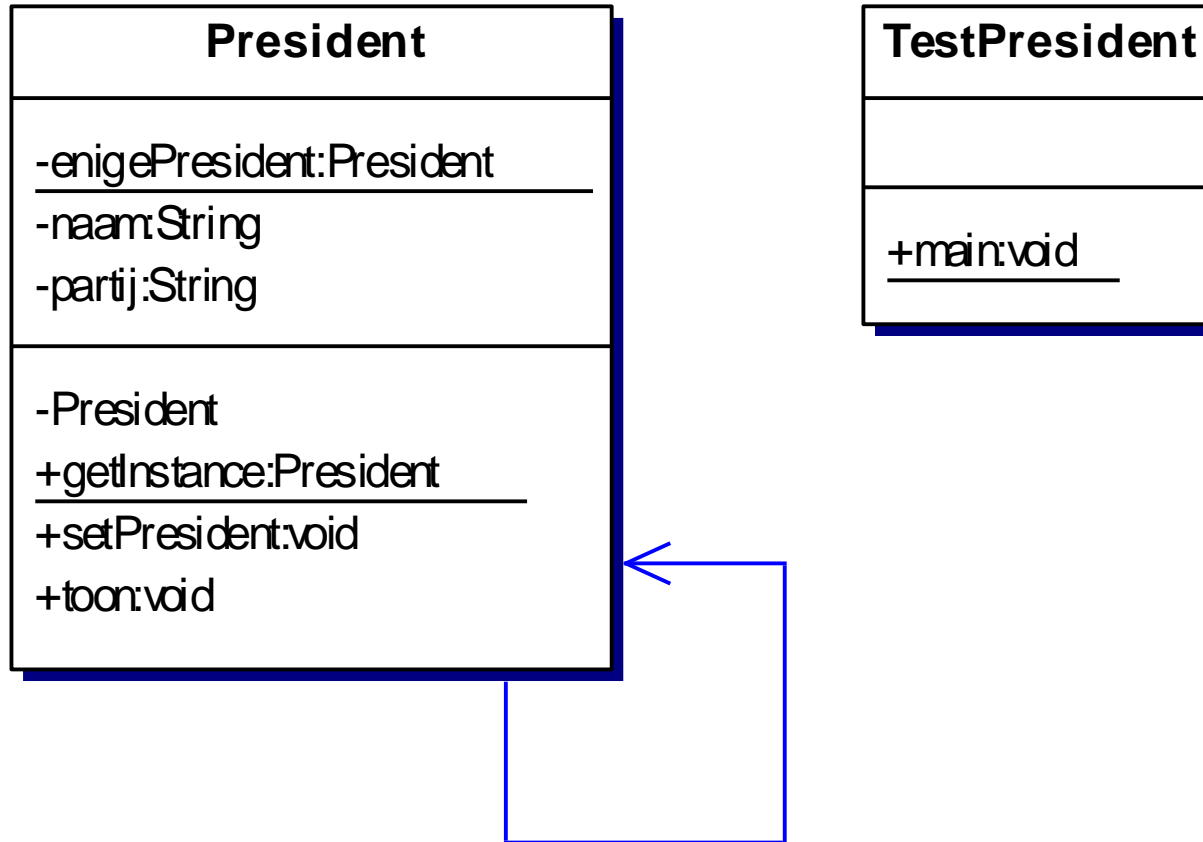


Singleton: kenmerken



- Oplossingsmodel:
 - Declareer het enige object als **klassevariabele**;
 - Maak de constructor **private** zodat hij voor de gebruiker van de klasse onzichtbaar wordt;
 - Maak een klassenmethode met de naam **getInstance** die het object zelf als returnwaarde teruggeeft.
- Lazy initialization:
 - Bij **lazy initialization** wordt de constructor pas opgeroepen bij de eerste oproep van `getInstance`

Singleton: UML



Singleton (lazy initialization)

```
public final class President {  
    private static President enigePresident; ← klassevariabele  
    private String naam;  
    private String partij;  
  
    private President(String naam, String partij) {  
        this.naam = naam;  
        this.partij = partij;  
    }  
  
    public static synchronized President getInstance() {  
        if (enigePresident == null) {  
            enigePresident = new President("onbekend",  
                                           "onbekend");  
        }  
        return enigePresident;  
    }  
}
```

← private constructor!

← getter (met lazy initialization)

← **Synchronized** komt later aan bod (W11: multithreading)

Singleton (lazy initialization - vervolg)

```
public void setPresident(String naam, String partij) {  
    this.naam = naam;  
    this.partij = partij;  
}
```

```
@Override  
public String toString() {  
    return "President: " + naam + ", partij: " + partij;  
}
```


```
@Override  
public Object clone() throws CloneNotSupportedException {  
    throw new CloneNotSupportedException();  
}
```

← Override van de
clone methode



Singleton (klassiek - voorkeur)

```
public final class President {  
    private static President enigePresident = new President();  
    private String naam;  
    private String partij;  
  
    private President() {  
        this.naam = "onbekend";  
        this.partij = "onbekend";  
    }  
  
    public static synchronized President getInstance() {  
        return enigePresident;  
    }  
}
```



onmiddellijke instantiëring!
(dus niet "lazy")

Singleton (klassiek - vervolg)

```
public void setPresident(String naam, String partij) {  
    this.naam = naam;  
    this.partij = partij;  
}
```

```
@Override  
public String toString() {  
    return "President: " + naam + ", partij: " + partij;  
}
```

```
@Override  
public Object clone() throws CloneNotSupportedException {  
    throw new CloneNotSupportedException();  
}
```

← Override van de
clone methode

Singleton (Voorbeeld gebruik)

```
public class Demo1 {  
    public static void main(String[] args) {  
        //President test = new President();  
        //President test2 = new President("Barack Obama",  
        //                                "Democrats");  
  
        President p = President.getInstance();  
        p.setPresident("Barack Obama", "Democrats");  
        System.out.println(p);  
  
        President np = President.getInstance();  
        np.setPresident("Donald Trump", "Republicans");  
        System.out.println(p);  
        System.out.println(np);  
    }  
}
```

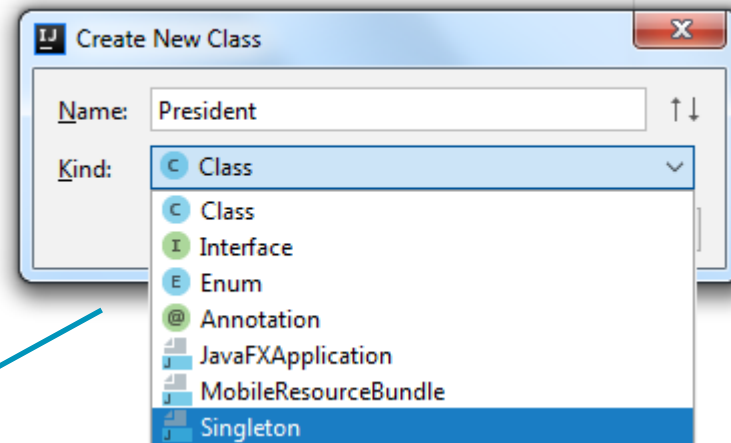
Onmogelijk, want **private** constructor!

```
President: Barack Obama, partij: Democrats  
President: Donald Trump, partij: Republicans  
President: Donald Trump, partij: Republicans
```

Singleton ondersteuning



Als je een nieuwe klasse maakt kan je voor een Singleton kiezen:



```
public class President {  
    private static President ourInstance = new President();  
  
    public static President getInstance() {  
        return ourInstance;  
    }  
  
    private President() {  
    }  
}
```

Singleton (via enum)

```
public enum President {  
    INSTANCE;
```

← Een **enum** met slechts één member!

```
    private String naam;  
    private String partij;
```

```
    public void setPresident(String naam, String partij) {  
        this.naam = naam;  
        this.partij = partij;  
    }
```

De constructor van een Enum is automatisch al private, dus hoeft niet zelf geschreven te worden,

```
@Override
```

```
public String toString() {  
    return "singleton.President: " + naam  
        + ", partij: " + partij;  
}
```

```
}
```



Democode: 3_Singleton_Enum

Singleton (via enum)

```
public class Demo3 {  
    public static void main(String[] args) {  
        President p = President.INSTANCE;  
        p.setPresident("Barack Obama", "Democrats");  
        System.out.println(p);  
  
        President np = President.INSTANCE;  
        np.setPresident("Donald Trump", "Republicans");  
        System.out.println(p);  
        System.out.println(np);  
    }  
}
```

```
President: Barack Obama, partij: Democrats  
President: Donald Trump, partij: Republicans  
President: Donald Trump, partij: Republicans
```

Singleton bespreking



- Hebben static methoden (`Klasse.methode`) op de klasse niet dezelfde eigenschappen als een singleton?
 - Voor beiden is er slechts één centraal punt: de klasse
 - Voor beiden is er Globale toegang
- Dit klopt, maar static methoden zijn niet OO
 - Bij static methoden is elke methodcall aan de klasse gebonden, je hebt bv geen polymorfisme
 - Singleton heeft maar één static methode: `President.getInstance` voor het maken van het object.
Daarna werk je steeds OO, met het gemaakte object.
- Globale simultane toegang kan ook een nadeel zijn
 - Geen inkapseling, globale afhankelijkheid
 - Opletten bij aanpassing in multithreaded omgeving (synchronized)

Agenda

1. Inleiding patterns

- Wat is een design pattern?
- Soorten patterns
- Voordelen



2. Singleton

- Kenmerken
- Singleton (Lazy initialization, klassiek, enum)

3. Observer

- Kenmerken en vergelijking MVP
- Voorbeeld
- Probleem (overerving/delegatie)

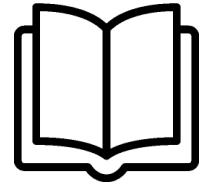
4. Static Factory

- Kenmerken
- Voorbeelden
- Voordelen / nadelen



Observer



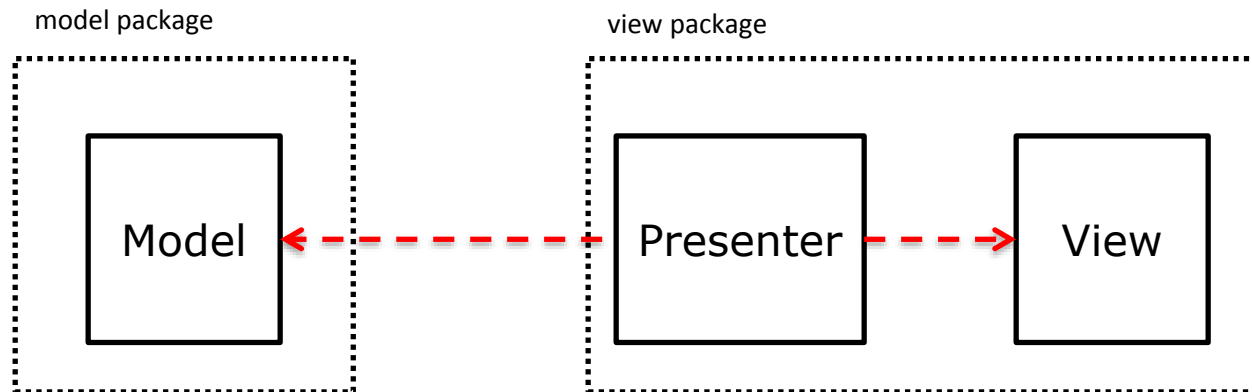


- E-book: "Observer" p.268 ev
 - Uit: "Applied Java Patterns", First Edition (Stephen Stelting and Olav Maassen)

Model View Presenter

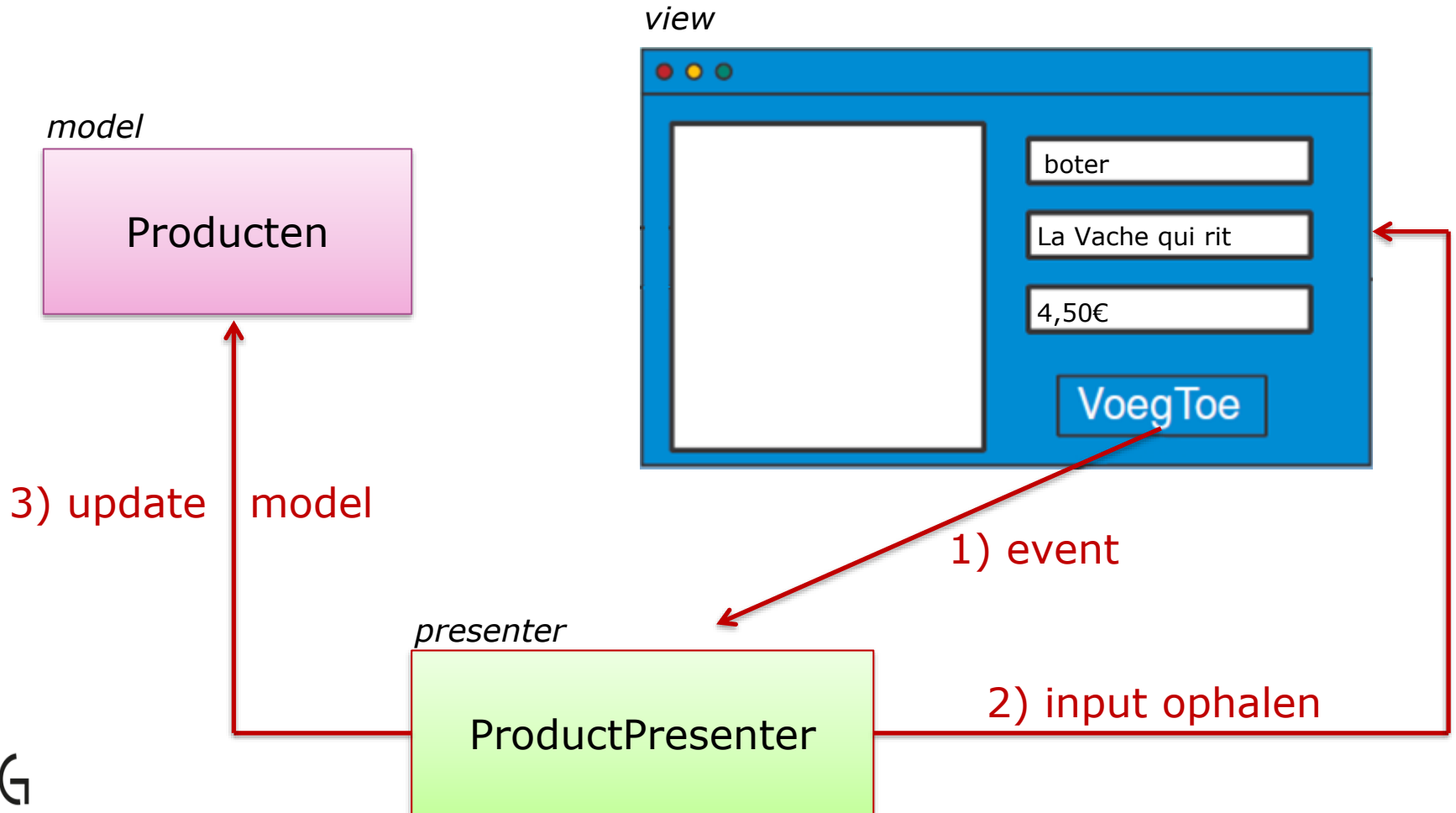
Even herhalen: het **MVP** pattern:

- **Model:** implementeert de logica, geen UI code (slim maar lelijk)
- **View:** tekent de UI, communiceert met de gebruiker, (bijna) geen logica (dom maar mooi)
- **Presenter:** de koppelaar tussen model en view;
 - ontvangt via de view input van de gebruiker en geeft die door naar model
 - vraagt de gewijzigde data op aan model en presenteert die via de view



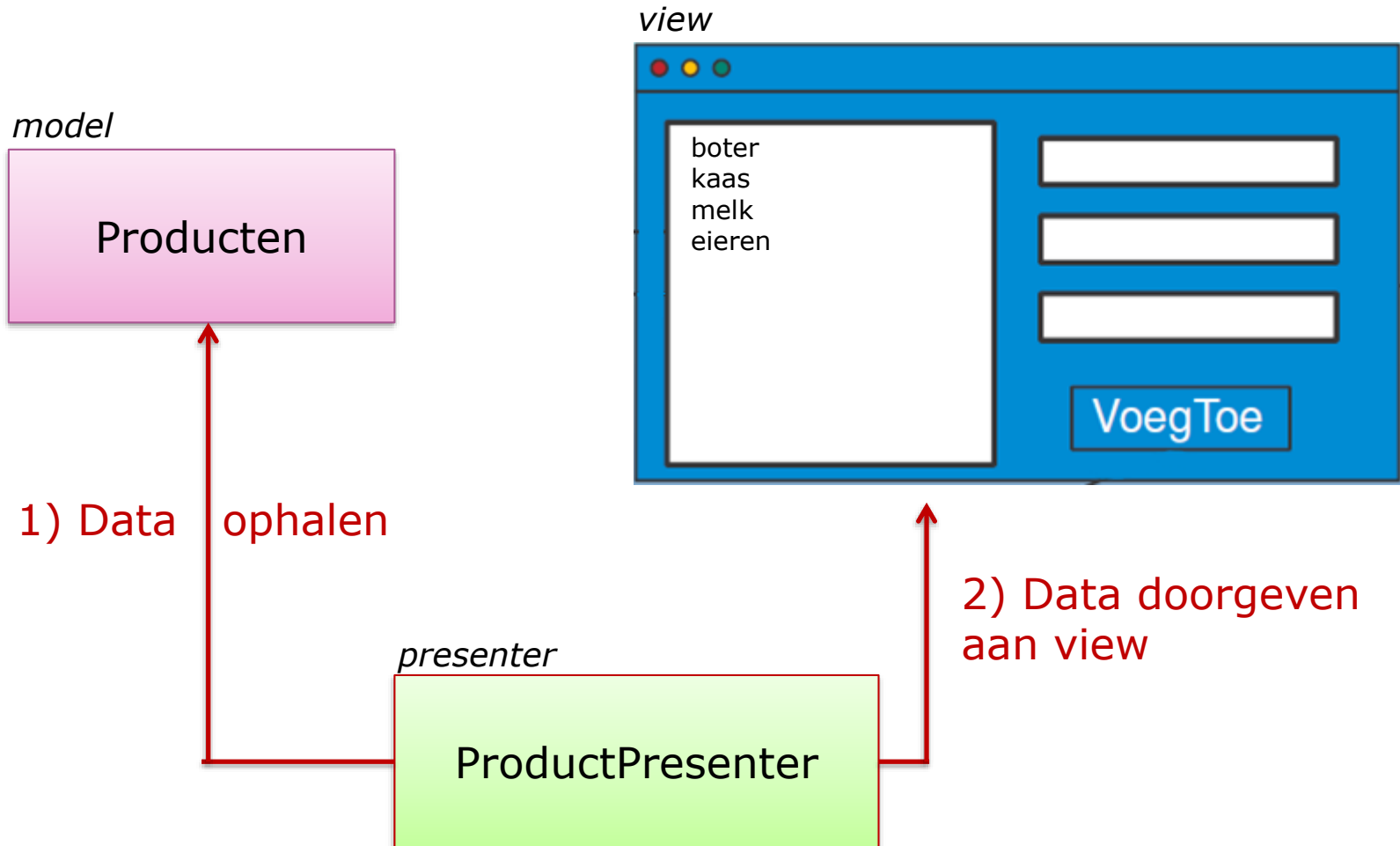
Model View Presenter

- User input toevoegen (MVP):



Model View Presenter

- Data uit model tonen (MVP):



MVP?



- Hoe lossen we deze situatie op?
meerdere views gekoppeld aan zelfde model:

Data input

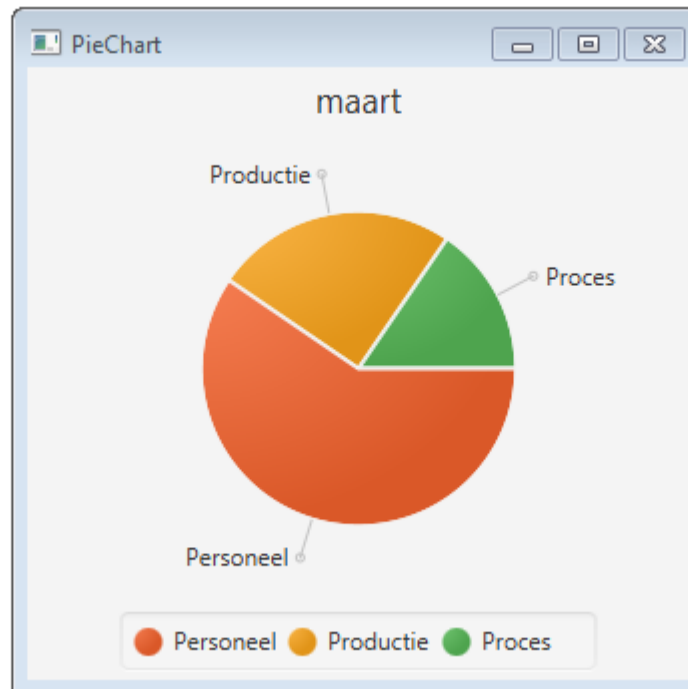
Maand: maart

Personeel: 100

Productie: 42

Proces: 26

Voeg toe



Logging

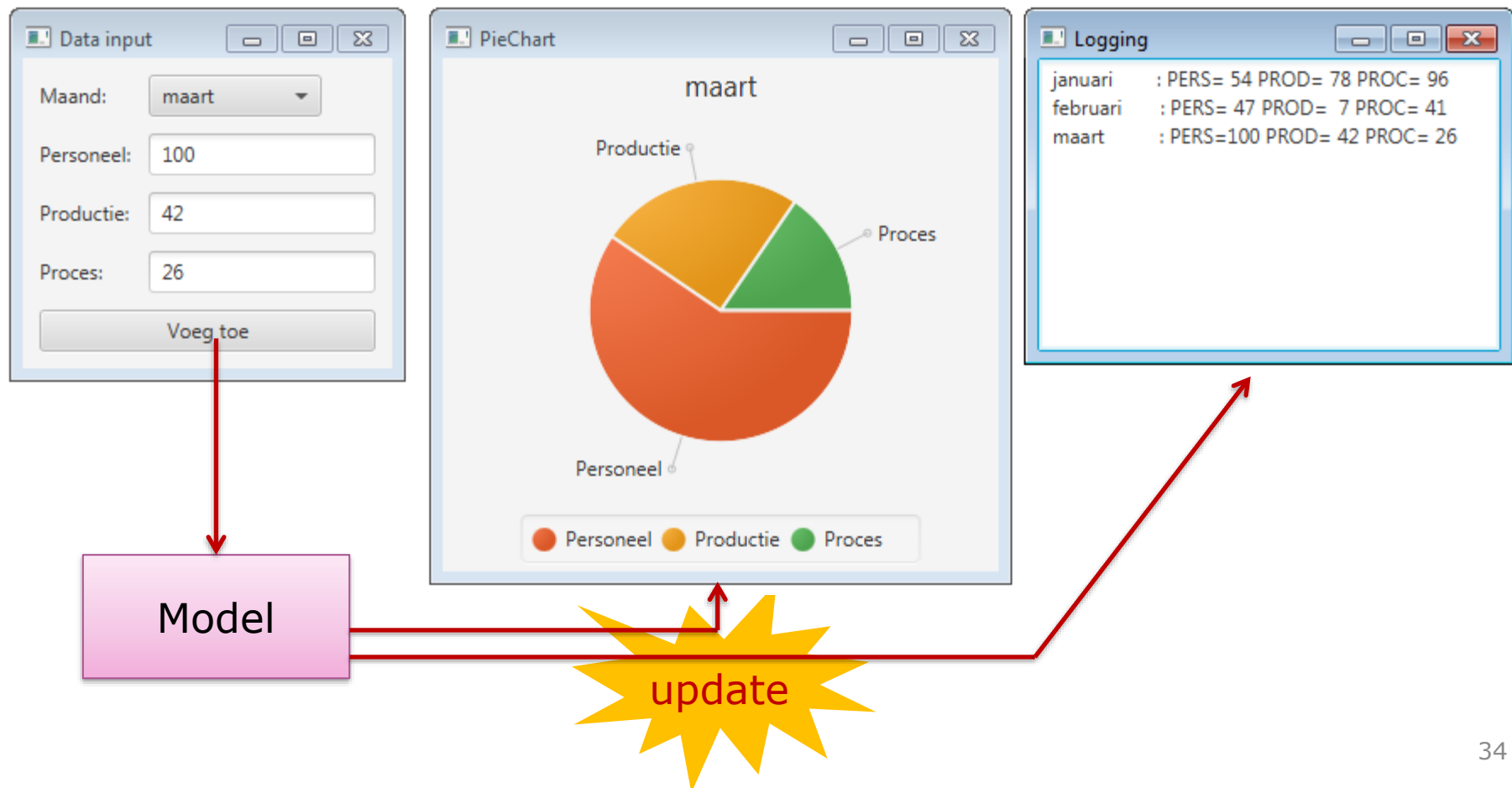
januari	: PERS= 54 PROD= 78 PROC= 96
februari	: PERS= 47 PROD= 7 PROC= 41
maart	: PERS=100 PROD= 42 PROC= 26

Model

Observer pattern!



- We maken het model **observable** en de views **observers**



Observer: kenmerken



- Naam: **Observer** pattern [GoF95]
(ook wel het **Publish/Subscribe**-pattern genoemd.)
- Familie: Behavioral patterns
- Samenvatting:
 - GoF: *"Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."*
 - Wanneer een bepaald object **van toestand verandert**, worden andere geïnteresseerde objecten **automatisch ingelicht**.
- Voorbeelden:
 - ✓ EventHandlers bij event-driven programming (JavaFX)
 - ✓ mailing list

Observer pattern: conceptueel



2 rollen: **Observer** \leftrightarrow **Observable**

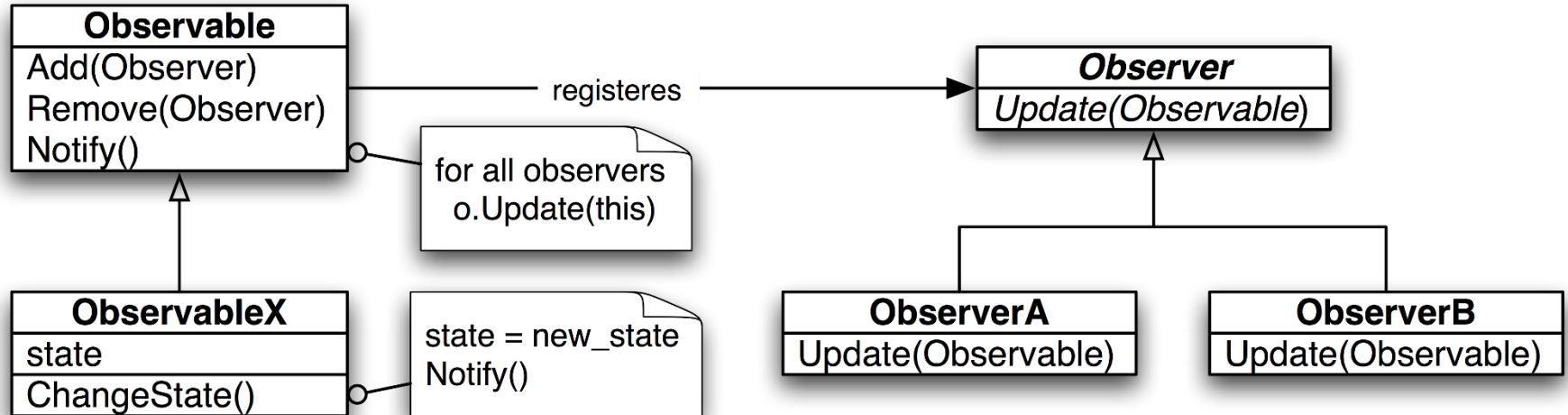
1. **Observer** (= de view)

- registreert zich bij de observable (komt op een lijst met geïnteresseerden te staan)

2. **Observable** (= het model)

- Bij elke wijziging:
 - de lijst van geregistreerde observers wordt overlopen
 - elke observer wordt verwittigd
 - de observer doet een refresh (update)

Observer pattern: UML

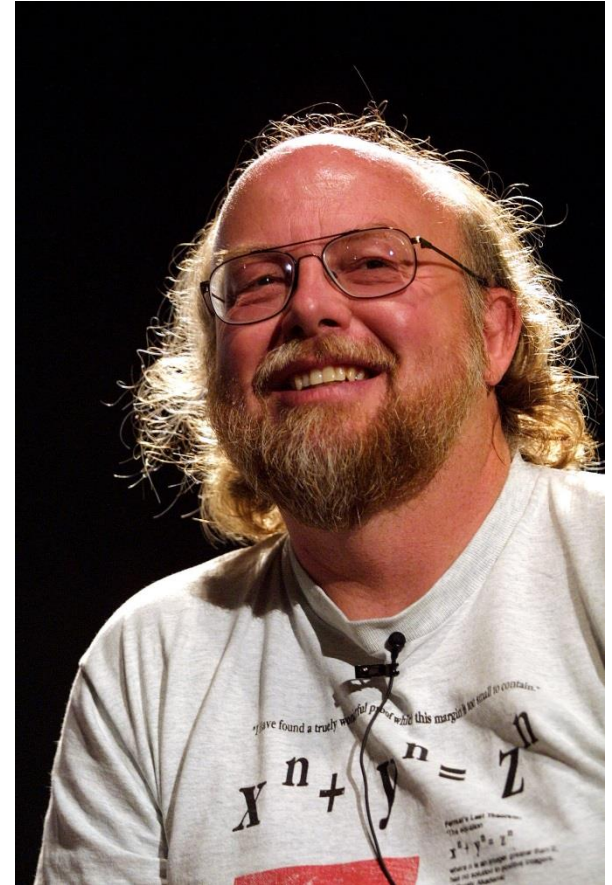


Observer pattern in Java

- Het patterns boek en Java verschenen ongeveer gelijktijdig op de markt
- Toch is een pattern als Observer van in het begin in de taal ingebouwd
- Meer weten over deze man?



http://en.wikipedia.org/wiki/James_Gosling



James Gosling

Observer in Java

Observable

observers

changed: boolean

addObserver()

deleteObserver()

setChanged()

notifyObservers()

notifyObservers(arg)

Observer

update(observable, arg)

De interface **Observer**
bestaat al

De klasse **Observable**
bestaat al

De klasse Observable

Naam: `java.util.Observable`

Constructor:

```
public Observable();
```

Methoden:

```
public void addObserver(Observer observer);  
public int countObservers();  
public void deleteObserver(Observer object);  
public void deleteObservers();  
public boolean hasChanged();  
public void notifyObservers();  
public void notifyObservers(Object object);  
protected void clearChanged();  
protected void setChanged();
```



Extra argument in de
vorm van een object

De interface Observer


Naam: `java.util.Observer`

Methoden:

```
public interface Observer {  
    void update(Observable observable, Object object);  
}
```



De **Observable** die de update oproept



Facultatief: **extra argument** in de vorm van een Object

Samengevat in 3 stappen

1. Modelklasse **extends Observable**

– Waar data wijzigt:

- **setChanged**
- **notifyObservers**



In deze
volgorde!



2. Viewklasse(n) **implements Observer**

- **update** (daarin de data opnieuw opvragen bij model)

3. Alle observers moeten zich registreren bij het model

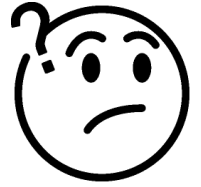
- **modelklasse.addObserver(viewKlasse)**



Wordt vaak
vergeten!

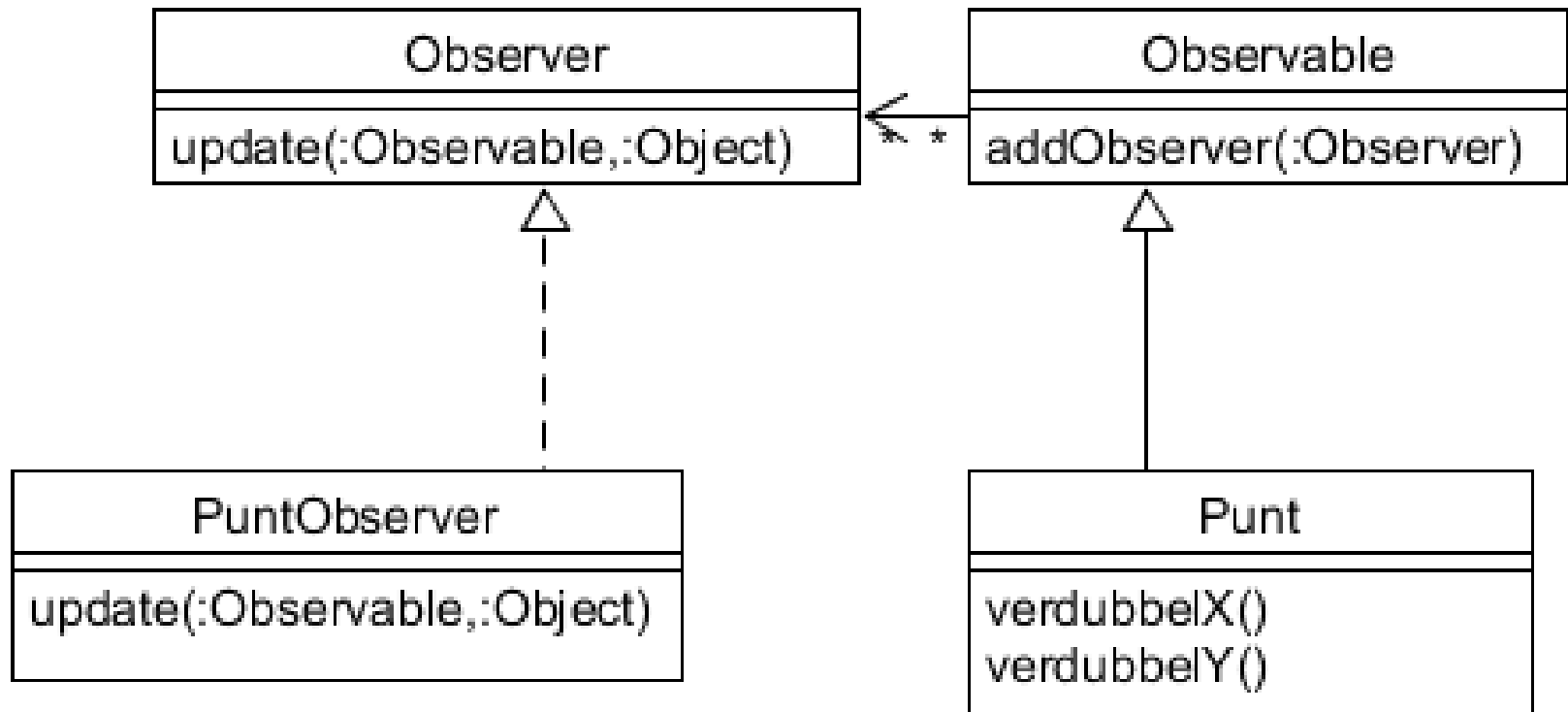


Conclusie MVP - Observer



- Gebruiken we dus vanaf nu geen MVP meer?
Alles met Observer-pattern?
- **Nee!**
- Enkel als een view zich automatisch moet aanpassen als het model verandert (meestal in een context met **meerdere views**)

Voorbeeld PuntObserver



Voorbeeld PuntObserver (1)

```
public class Punt
    extends Observable {
    private int x;
    private int y;

    public Punt(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public void verdubbelX() {
        x *= 2;
        setChanged();
        notifyObservers("X");
    }
}
```

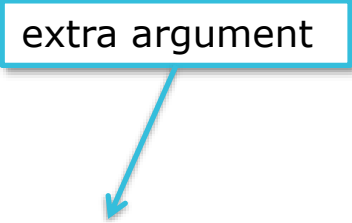
```
public void verdubbelY() {
    y *= 2;
    setChanged();
    notifyObservers("Y");
}
```

```
@Override
public String toString() {
    return "(x,y) = ("
        + x + "," + y + ")";
}
```

Er wordt een **extra argument** meegegeven naar de observers!

Voorbeeld PuntObserver (2)

```
public class PuntObserver implements Observer {  
    private Punt punt;  
  
    public PuntObserver(Punt punt) {  
        this.punt = punt;  
    }  
  
    public void update(Observable observable, Object object) {  
        System.out.println(object  
            + " waarde gewijzigd, nieuwe waarden: " + punt);  
    }  
}
```



```
X waarde gewijzigd, nieuwe waarden: (x,y) = (2,2)  
Y waarde gewijzigd, nieuwe waarden: (x,y) = (2,4)
```

Voorbeeld PuntObserver (3)

```
public class Demo {  
    public static void main(String[] args) {  
        Punt punt = new Punt(1, 2);  
        PuntObserver observer = new PuntObserver(punt);  
        punt.addObserver(observer);  
  
        punt.verdubbelX();  
        punt.verdubbelY();  
    }  
}
```

X waarde gewijzigd, nieuwe waarden: (x,y) = (2,2)
Y waarde gewijzigd, nieuwe waarden: (x,y) = (2,4)



Probleem



Je wenst een klasse **Observable** te maken,
maar je mag/kunt de klasse zelf niet wijzigen.
Wat nu?

Mogelijke oplossingen:

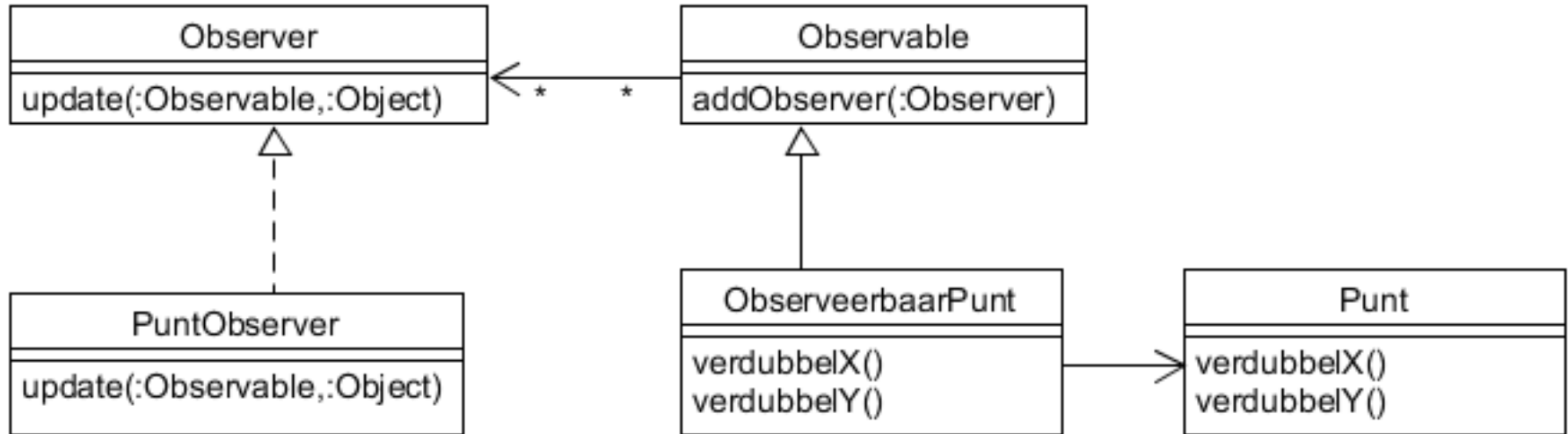
- Gebruik **delegatie** (object van de klasse is attribuut van de nieuwe klasse)
- Gebruik **overerving** (de nieuwe klasse wordt van de originele klasse afgeleid)

Originele klasse

```
public class Punt {  
    private int x;  
    private int y;  
  
    public Punt(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public void verdubbelX() {  
        x *= 2;  
    }  
}
```

```
public void verdubbelY()  
  
    y *= 2;  
}  
  
@Override  
public String toString() {  
    return "(x,y) = (" +  
        + "," + y + ")";  
}
```

Oplossing Delegatie



Oplossing delegatie

```
public class ObserveerbaarPunt
    extends Observable {
    private Punt punt;

    public ObserveerbaarPunt(
        int x, int y) {
        punt = new Punt(x, y);
    }

    public void verdubbelX() {
        punt.verdubbelX();
        setChanged();
        notifyObservers("X");
    }
}
```

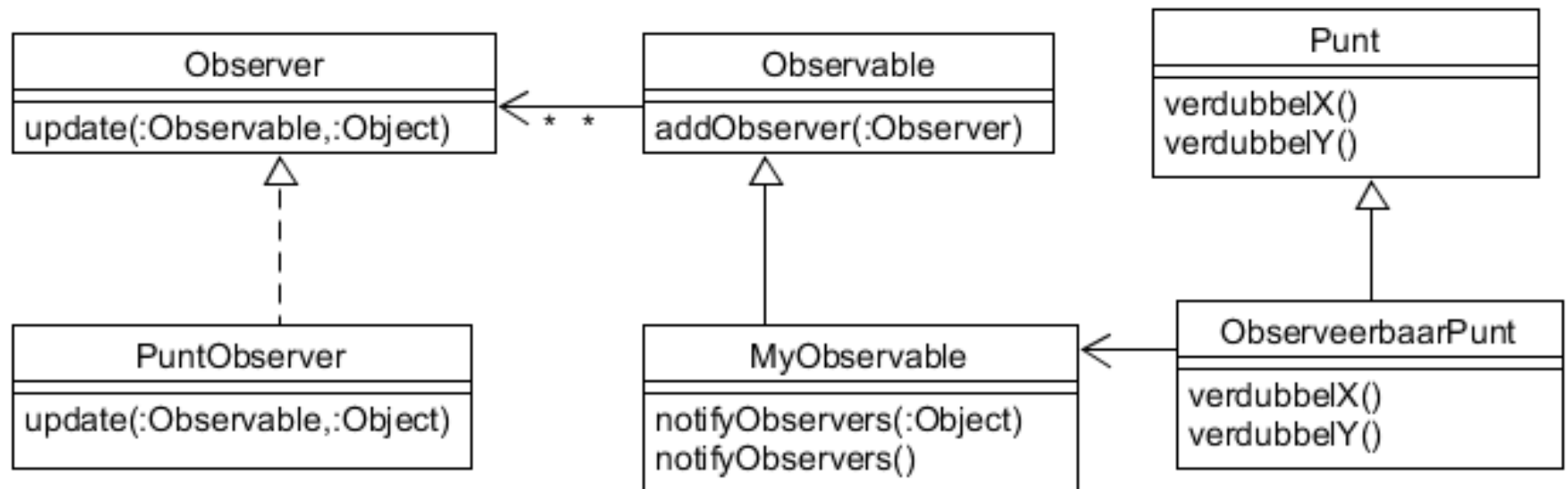
```
public void verdubbelY() {
    punt.verdubbelY();
    setChanged();
    notifyObservers("Y");
}

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

Delegatie naar
het ingekapselde
Punt-object



Oplossing Overerving



Oplossing overerving

```
public class ObserveerbaarPunt extends Punt {
    private MyObservable notifier = new MyObservable();

    public ObserveerbaarPunt(int x, int y) {
        super(x, y);
    }

    public void verdubbelX() {
        super.verdubbelX();
        notifier.notifyObservers();
    }

    public void verdubbelY() {
        super.verdubbelY();
        notifier.notifyObservers();
    }

    public void addObserver(Observer observer) {
        notifier.addObserver(observer);
    }
}
```

Overerving, dus
gebruik maken van
super-methoden

Oplossing overerving (vervolg)

```
public class MyObservable extends Observable {  
    public void notifyObservers() {  
        super.setChanged();  
        super.notifyObservers();  
    }  
  
    public void notifyObservers(Object object) {  
        super.setChanged();  
        super.notifyObservers(object);  
    }  
}
```



Event handling

- Event handling in **JavaFX** is gebaseerd op observer pattern:
 - De observable is de JavaFX-component (vb: **Button**)
 - De observer is de gekoppelde **EventHandler**
 - Als de toestand van de component verandert, dan verwittigt hij de **EventHandlers** (=observers) via de methode **handle**)
 - Elke **EventHandler** moet zich ook eerst registreren bij de component (vb: **button.setOnAction**)

Observer pattern

- **Observable**

= model-klasse (vb: **Game**)

- Wanneer het model verandert, worden de bijhorende observerklassen verwittigd.

→ `setChanged()`

→ `notifyObservers()` ;

Event handling in JavaFX

- **Observable**

= component (vb: **Button**)

- Wanneer de knop wordt ingedrukt, worden de bijhorende eventHandlers verwittigd.

Observer pattern

- **Observer**

= view-klasse (vb: **GameView**)

- Observer wordt aan model gekoppeld:
`myGame.addObserver(myView);`
- verplichte interface:
`implements Observer`
- bijhorende eventmethode:
`public void update(
 Observable obs,
 Object arg);`

Event handling in JavaFX

- **Observer**

= EventHandler-klasse

- EventHandler wordt aan component gekoppeld:
`myButton.setOnAction(...);`
- verplichte interface:
`implements EventHandler`
- bijhorende eventmethode:
`public void handle(
 ActionEvent event);`

Opdrachten



- Groeiproject
 - module 5
(deel 1 en 2: "Observer pattern")



- Opdrachten op BB
 - Singleton: Film opgave
 - Observer: 2 views met observer



Het wordt aangeraden om, naast het groeiproject, ook andere oefeningen van BB te maken!

Agenda

1. Inleiding patterns

- Wat is een design pattern?
- Soorten patterns
- Voordelen



2. Singleton

- Kenmerken
- Singleton (Lazy initialization, klassiek, enum)

3. Observer

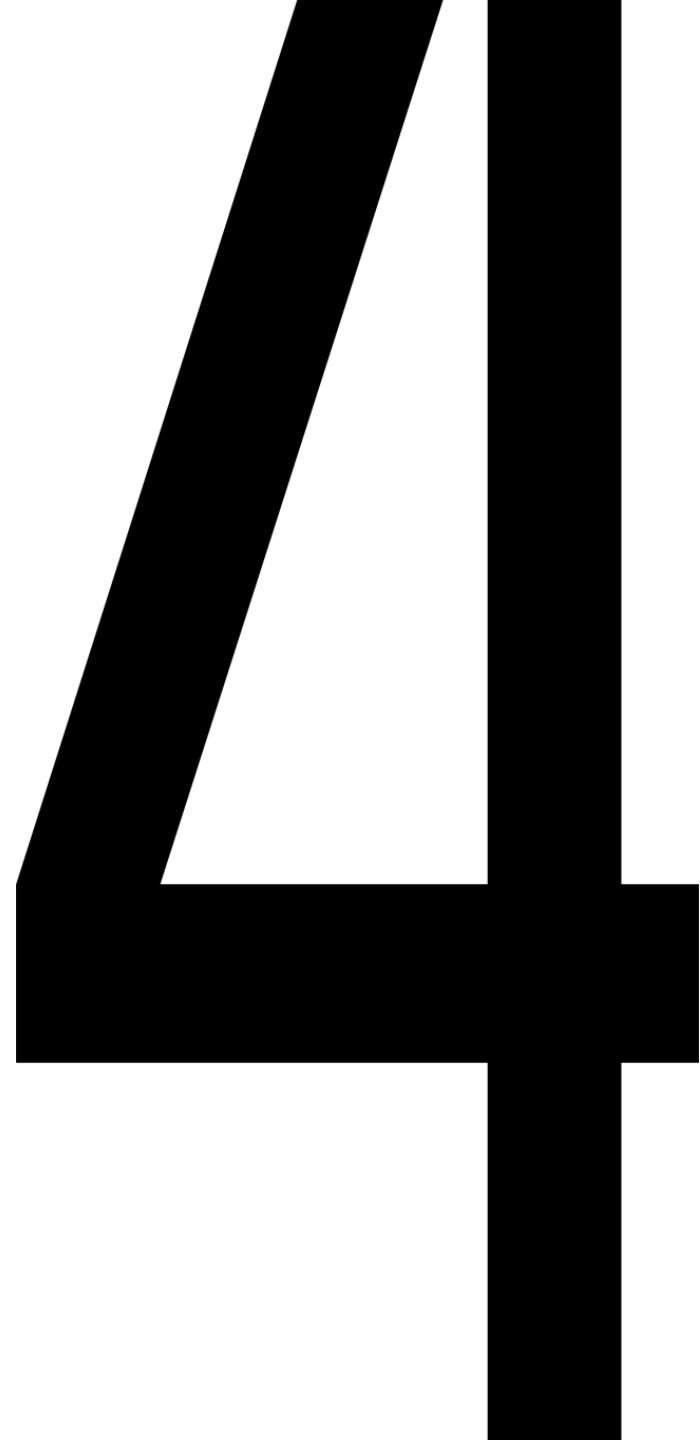
- Kenmerken en vergelijking MVP
- Voorbeeld
- Probleem (overerving/delegatie)

4. Static Factory

- Kenmerken
- Voorbeelden
- Voordelen / nadelen



Static Factory





- E-book: "Abstract Factory" p.257 ev
 - Uit: "Applied Java Patterns", First Edition (Stephen Stelting and Olav Maassen)
 - In het e-book vind je nog een ander (verwant) creational pattern. Lees dit als je een voorbeeld wil zien van andere flexibele manieren om objecten aan te maken.

Static Factory: kenmerken



- Naam: **Static Factory Method** pattern
- Familie:
 - Creational patterns
- Samenvatting:
 - Geen GoF pattern, beschreven door Joshua Bloch
 - Factory patterns gebruik je om nieuwe objecten te maken, het zijn alternatieven voor **new**.
 - static factory-method(s) i.p.v. constructor(s)
- Zie ook: Abstract Factory, Factory Method, Singleton



Static Factory Method: motivatie



Waarom gebruiken?

- Je hebt meer **controle** over wat je factory methode maakt.
- **Constructors** kunnen enkel verschillen in parameter types. Je kan echter meerdere factory methoden hebben met dezelfde parameter types als ze verschillende namen hebben.
- De **namen** van de factory methoden geven de gebruiker informatie over hoe iets gemaakt wordt.



One advantage of static factory methods is that, unlike constructors, they have names.

— Joshua Bloch —

AZ QUOTES

Static Factory Method: voorbeeld

- De klasse **LocalDate** heeft een *private constructor*
- Creatie van LocalDate-object enkel mogelijk via een aantal *static methods*:
 - `now()`
 - `of(int year, int month, int dayOfMonth)`
 - `ofEpochDay(long epochDay)`
 - `ofYearDay(int year, int dayOfYear)`
 - `parse(CharSequence text)`
 - `parse(CharSequence text, DateTimeFormatter formatter)`

Static Factory Method: voorbeelden

- Andere Voorbeelden:

- JavaFX: **SpinnerValueFactory**

`SpinnerValueFactory.IntegerSpinnerValueFactory`

`SpinnerValueFactory.DoubleSpinnerValueFactory`

`SpinnerValueFactory.ListSpinnerValueFactory`

`SpinnerValueFactory.LocalDateSpinnerValueFactory`

De `SpinnerValueFactory` is het model achter de JavaFX `Spinner`.




Meer weten over `SpinnerValueFactory`?

<http://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/SpinnerValueFactory.html>

Static Factory Method: wanneer gebruiken?

- Als je zelf toezicht wil houden over de creatie van objecten.
- In plaats van een constructor gebruik je een **static factory methode** die een object teruggeeft.
- Een voorbeeld (uit de klasse Boolean):

```
public static Boolean valueOf(boolean b) {  
    return (b ? Boolean.TRUE : Boolean.FALSE) ;  
}
```



Er wordt niet telkens een nieuw `Boolean`-object gecreëerd, maar wel een referentie geretourneerd naar één van de 2 bestaande `Boolean`-objecten.

Probleemstelling



–De klasse Punt

- Ik wil 2 constructors voorzien:
 - op basis van x en y coördinaten
 - op basis van afstand van x tot nulpunt, afstand van y tot het nulpunt

–Probleem?

```
public Punt(double x, double y) {...}  
public Punt(double xDistance, double yDistance) {...}
```

Constructor overloading kan hier niet! Zie je ook waarom?

–Oplossing met **Static Factory Method** pattern:

- Maak een aantal static methoden voor de creatie van Punt-objecten (en maak de constructor eventueel private)

Static Factory Method: voordelen



- **Naam** van de methode zelf kiezen (<> constructor)
- **Zelfde signature** mogelijk (<> constructor)

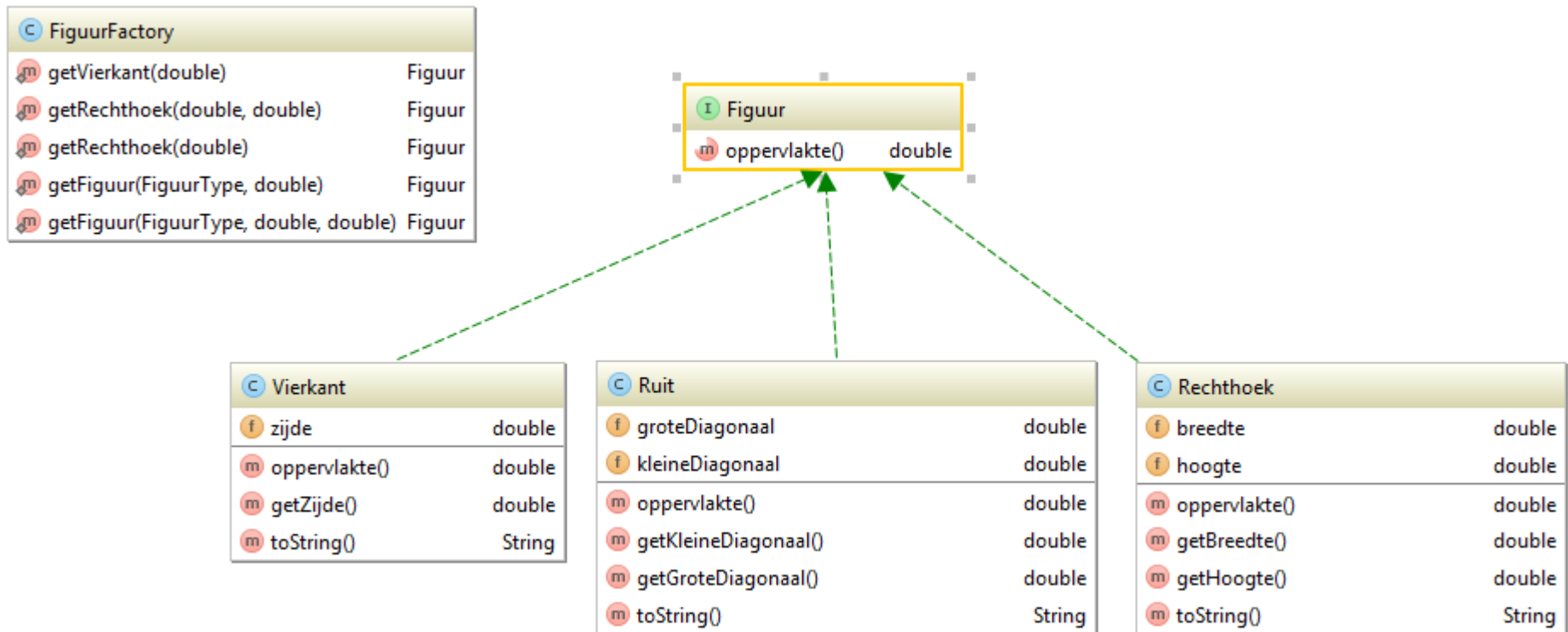
```
public static Complex valueOf(double re, double im) { ... }  
public static Complex valueOfPolar(double r, double theta)  
{ ... }
```

- Controle mogelijk over **wanneer** (uitgesteld?) het object aangemaakt wordt (<> constructor)

Static Factory Method: voordelen



- Objecten van om het even welke **subklasse** maken (<>constructor)



Static Factory Method: voordelen



- Objecten van om het even welke **subklasse** maken (<>constructor)

Voorbeeld:

```
public class FiguurFactory {  
    public static Figuur getVierkant(double zijde) {  
        return new Vierkant(zijde);  
    }  
    public static Figuur getRechthoek(double b, double h){  
        return new Rechthoek(b, h);  
    }  
    public static Figuur getRuit(double ld, double kd) {  
        return new Ruit(ld, kd);  
    }  
}
```

Static Factory Method: voordelen



- **Controle** houden over het **aantal** objecten dat gemaakt wordt (<> constructor)
 - **Hergebruik** van immutable objecten → betere performantie!
 - Singleton is een speciaal geval van Static Factory Method
- Voorbeelden:
 - `Boolean.valueOf(boolean b)`
 - Je wil controle houden over het aantal schaakstukken: 1 zwarte koning, 1 witte koning, 2 zwarte paarden, 2 witte paarden, ... Dus je maakt een klasse **SchaakStukkenFactory** die bijhoudt hoeveel keer een bepaald schaakstuk al is aangemaakt.

(Zie ook oefening "*Quarto*")

Static Factory Method: **nadelen**



- **private constructor**, dus geen overerving meer mogelijk!
 - Protected constructor is een alternatief
 - Static methoden worden niet overgeërfd
- Static factory methoden zijn niet van andere static methoden te **onderscheiden**.
- Daarom naamconventies:
 - **valueOf** – voor type conversie
 - **getInstance** / **newInstance** – in frameworks en ook bij singletons

ImageReader Voorbeeld/1

```
public interface ImageReader {  
    public DecodedImage getDecodedImage();  
}
```

```
public class GifReader implements ImageReader {  
    public GifReader( InputStream in ) {  
        // check that it's a gif, throw exception if it's not, then decode it.  
    }  
  
    public DecodedImage getDecodedImage() {  
        return decodedImage;  
    }  
}
```

```
public class JpegReader implements ImageReader {  
    //....  
}
```

ImageReader Voorbeeld/2

```
public class ImageReaderFactory {  
    private ImageReaderFactory() {  
        //not available  
    }  
  
    public static ImageReader getImageReader(InputStream is) {  
        int imageType = figureOutImageType(is);  
  
        switch( imageType ) {  
            case ImageReaderFactory.GIF:  
                return new GifReader(is);  
            case ImageReaderFactory.JPEG:  
                return new JpegReader(is);  
            // etc.  
        }  
    }  
}
```

Static Factory Method refactoring

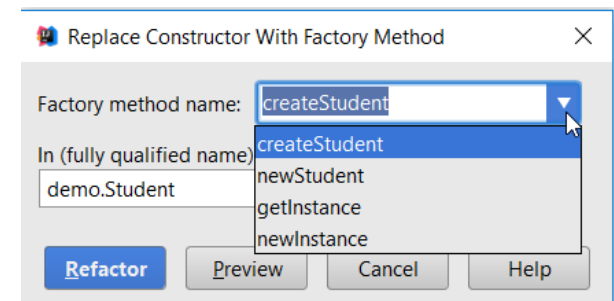
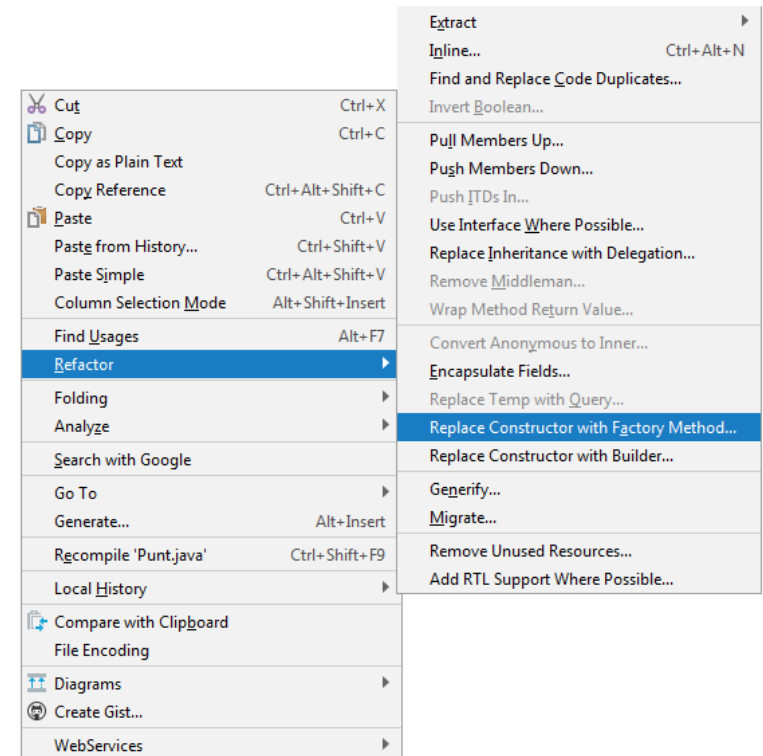
- Voor:

```
public Student(int id,
String naam, double score) {
    this.id = id;
    this.naam = naam.trim();
    this.score = score;
}
```

- Na:

```
private Student(int id, String naam,
double score) {
    this.id = id;
    this.naam = naam.trim();
    this.score = score;
}
```

```
public static Student createStudent(int id,
String naam, double score) {
    return new Student(id, naam, score);
}
```



Evolutie

- Alle creational patterns (factories, singleton) geven, zoals de naam het zegt, alternatieven voor een constructor
- We zagen een ander alternatief voor een constructor: reflection (`Class#newInstance`).
 - Rond reflection zijn een aantal creational frameworks gebouwd, vaak in combinatie met annotaties
 - Deze populaire techniek noemt men **Dependency Injection**
 - Voorbeelden in Java Enterprise Edition:
 - `@javax.inject.Inject`, `@javax.ejb.Singleton`...

Opdrachten



- Groeiproject
 - module 5
(deel 3: "Static Factory pattern"
deel 4: "Proxy pattern" nog niet!)



- Opdrachten op BB
 - Singleton: Film opgave
 - Observer: 2 views met observer
 - Factory: Quarto, Figuur



- Zelftest!

Het wordt aangeraden om, naast het groeiproject, ook andere oefeningen van BB te maken!