Design patterns/1 (Gang of Four)

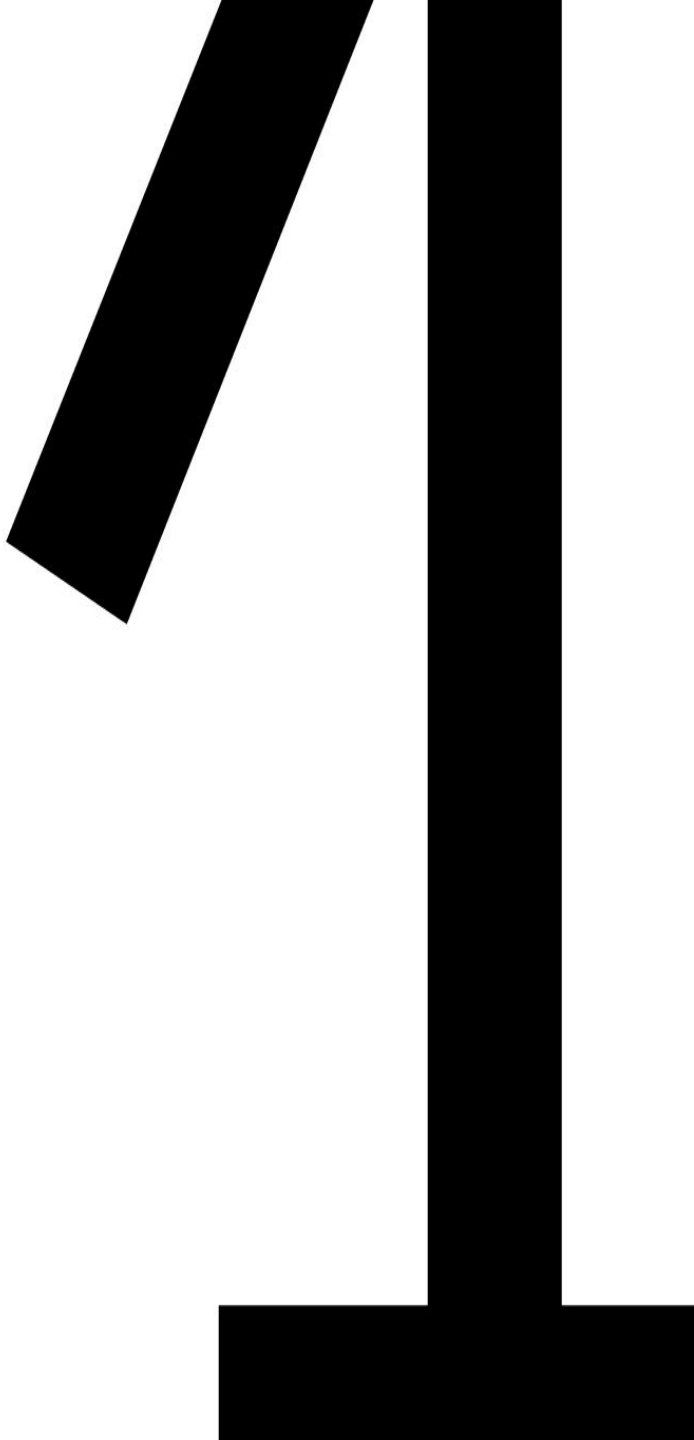
Programming 4

KdG Karel de Grote Hogeschool

Agenda

1. Code: values and principles
2. Design patterns introduction
3. State
4. Strategy
5. Observer

Code: values and principles



Programming values

- Values are general characteristics of code that are always appliccable and technology agnostic
- Simple
  - Easy to read
    - Clear structure
    - Clear naming
  - Easy to write
  - Easy to use
  - Single Responsibility Principe (SOLID)

Programming values

- Flexibility
  - Easy to change
  - Usable in many contexts
- Sometimes you have to strike a balance (example between simplicity and flexibility)

Programming principles

- Guidelines to write code that conforms to programming values. Technology agnostic.
- Keep impact of changes local
  - impact of a change should largely be limited to the method, class, package that is changed
  - Related to: encapsulation, low coupling

try to see the diff in 2 pieces of similar code and try to make it 1

– Avoid repetition
  - Do not copy/paste code

Programming principles

put things together that change together

- Put things together that change together
  - example: put user interface code together
  - Business logic within one package is often manage by the same person(s)

users are responsible for certain things - usually put them together

- Put logic and data it operates on together
  - Put methods in the class containing the data they use
  - Related to: infomation expert (GRASP)
- Favour composition over inheritance
  - Composition is dynamic, loose coupling (but requires more work)
- Symmetry
  - Are the methods in a class complete? Get/set, add/remove, enable/disable
  - Similar methods should have similar signatures, names…

combine composition with inheritance

are all actions supported

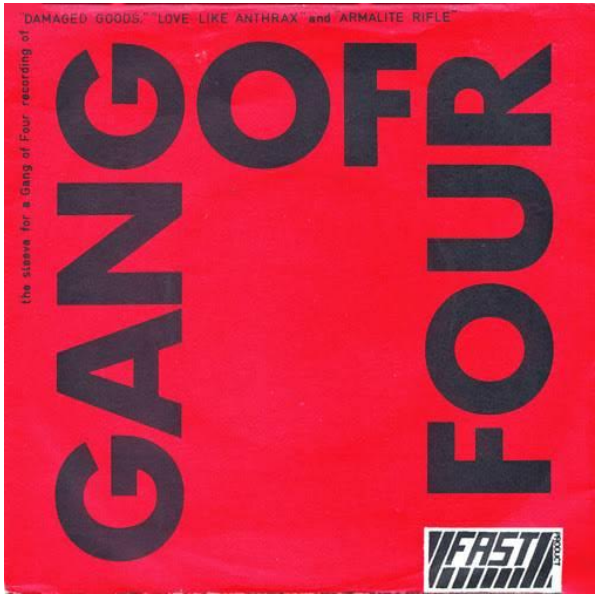
Programming principles

– prefer declarative expressions over imperative
  - examples: annotations, functional style (method chaining, builders), Domain Specific Languages (DSL: SQL, CSS, rule engines…), configuraton files
  - imperative

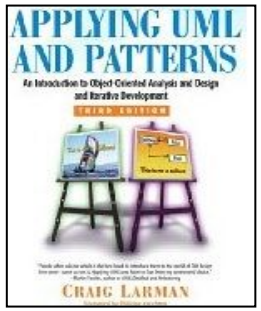
```
List numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
int result = 0;
for(int i = 0; i < numbers.size(); i++) {
  if(numbers.get(i) > 5 && isEven(numbers.get(i)) && numbers.get(i) < 9 && numbers.get(i) * 2 > 15) {
    result = numbers.get(i);
    break;
  }
}
```

  - declarative

```
List numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
int result = numbers.stream()
                    .filter(number -> number > 5)
                    .filter(number -> isEven(number))
                    .filter(number -> number < 9)
                    .filter(number -> number * 2 > 15)
                    .findFirst()
                    .get();
```



Design patterns introduction



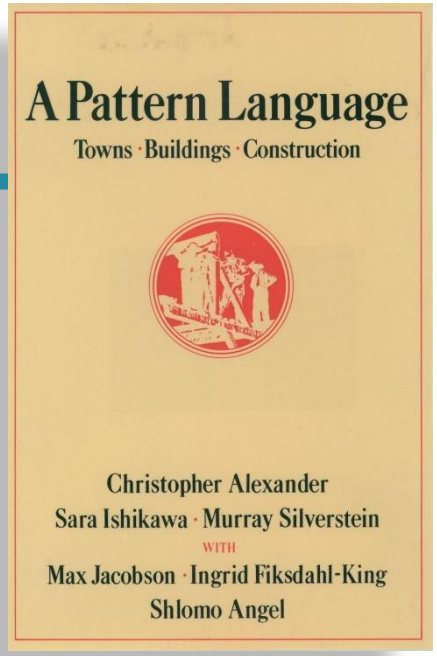
17.6

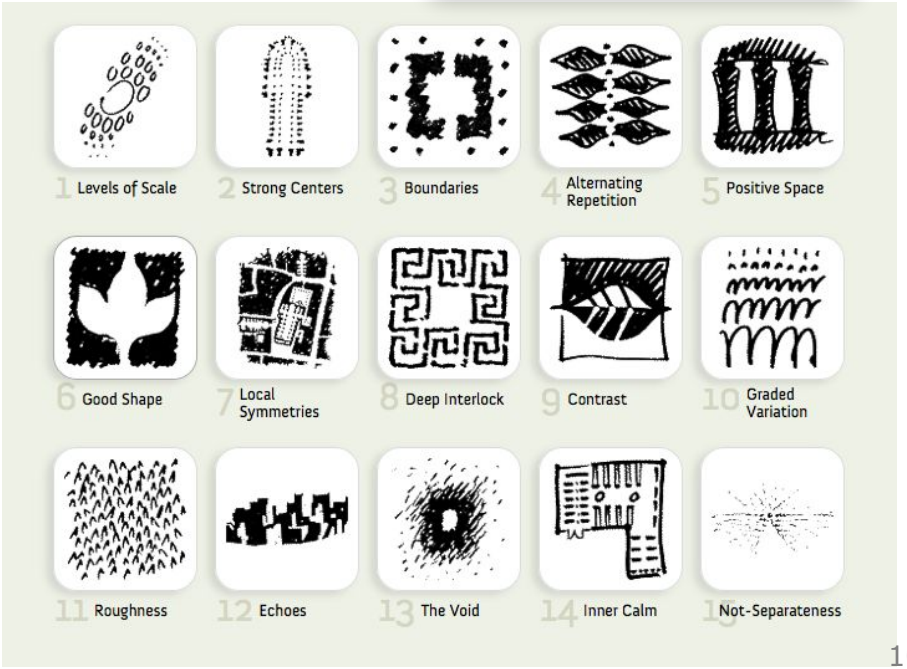
KdG
Karel de Grote Hogeschool

What is a design pattern?

- **A proven general solution for a common design problem.**

  Each pattern describes a context, a returning problem and the essence of its solution, so that it can be used again and again, without ever having to undertake the same action twice





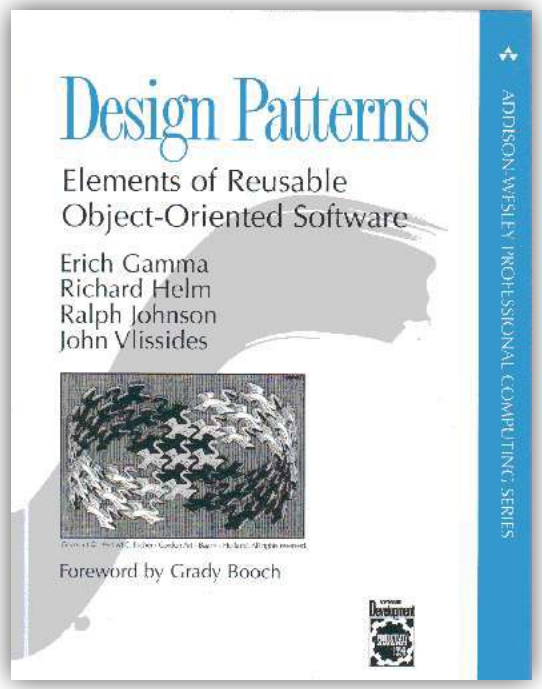
KdG
Karel de Grote Hogeschool

Design Patterns

Elements of Reusable Object-Oriented Software

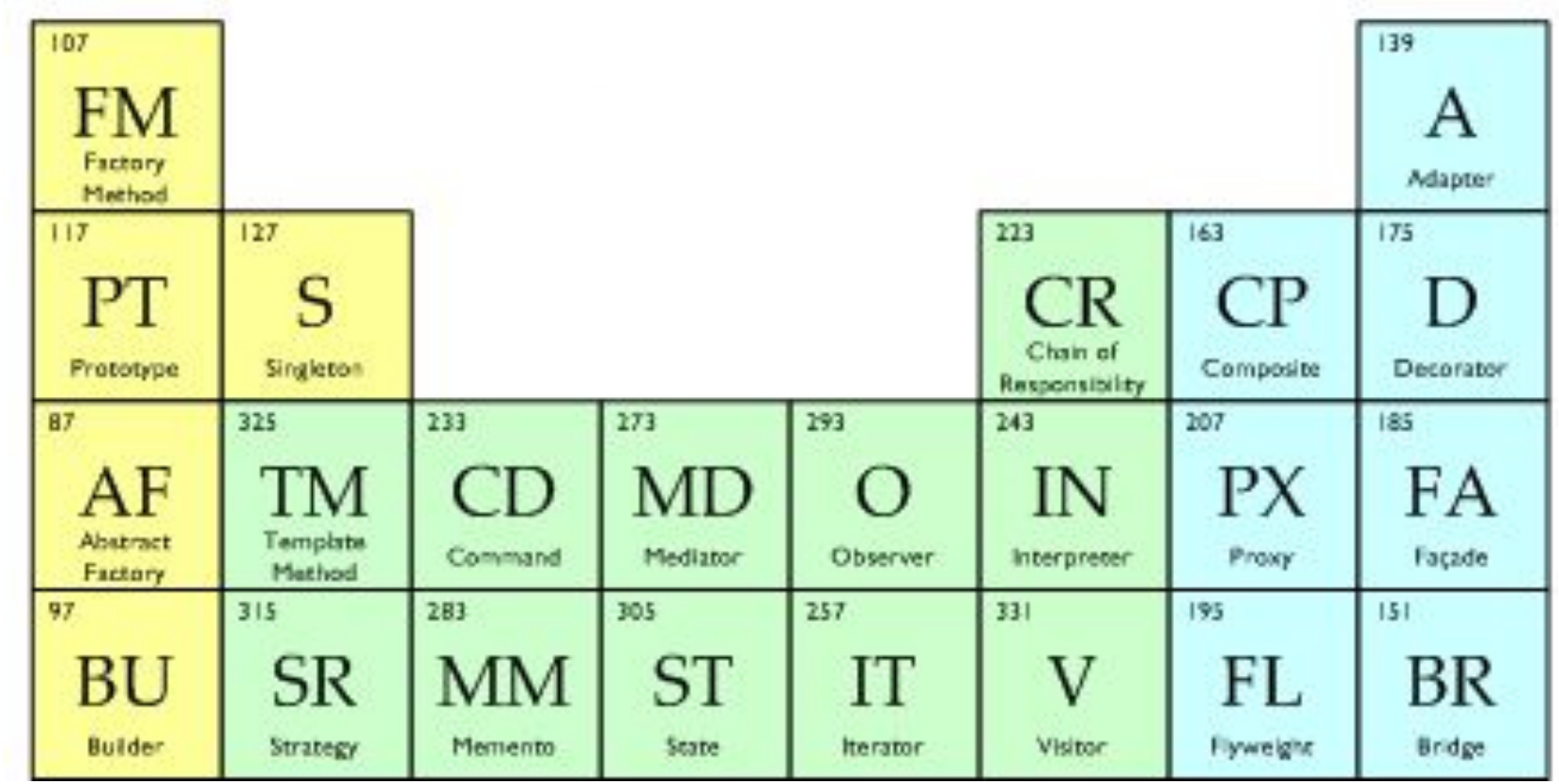
- Gamma, Helm, Johnson, Vlissides

  (GoF = Gang of Four)
- 1995 (exmples in C++)
- 23 patterns in 3 families:
  - Creational (e.g. Singleton, Abstract Factory)
  - Structural (e.g. Adapter, Composite, Proxy)
  - Behavioural (e.g. Observer, State)



Pattern categories

Creational | Behavioural | Structural



Pattern format

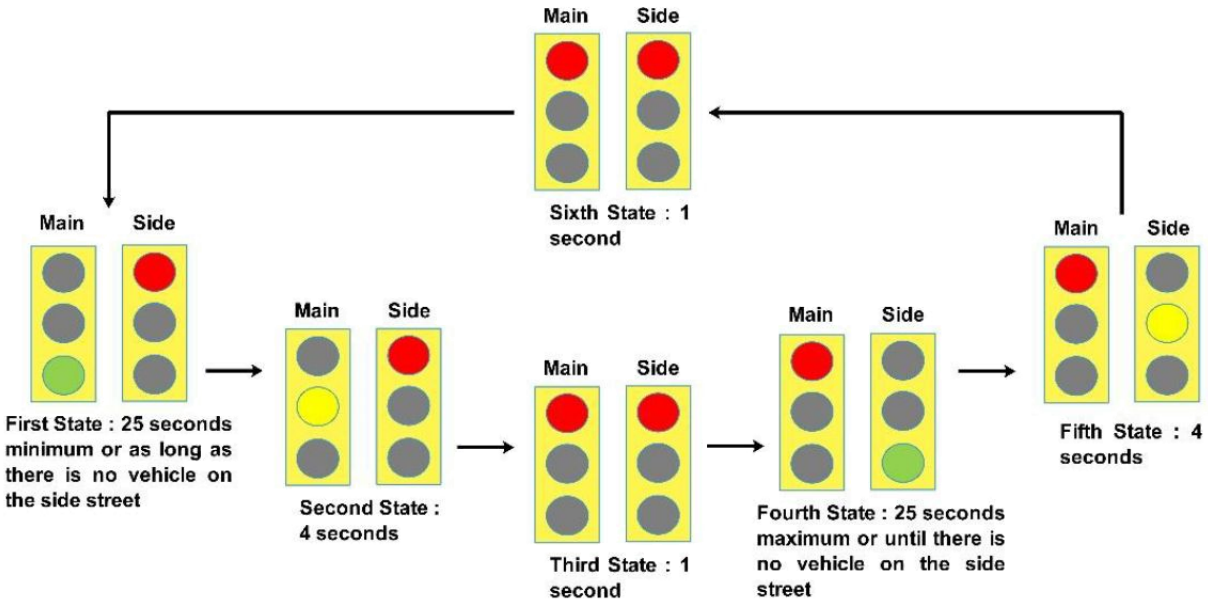
- Each pattern has a **name** and is documented in a fixed structure:
  - ✔ **intent**
  - ✔ **context** in which the pattern can be applied
  - ✔ **solution**, including rationale, trade offs and different strategies for implementing the pattern
  - ✔ **consequences** of the pattern
    - ✔ + references to related patterns (we will not discuss all GoF patterns)

separated from coding

OO design patterns characteristics

inheritance will be part of the solution

- Solution for a problem type based on OO programming values and principles
- Independant of programming language
- Solution often includes multiple classes working together. Specifies methods and collaborations.
- Allows communicating about solutions at a higher level
- Generic solution must be transposed to the context of a specific problem en working environment
- Patterns have influenced the evulution of languages, tools and libraries
  - Sometimes already built-in



State

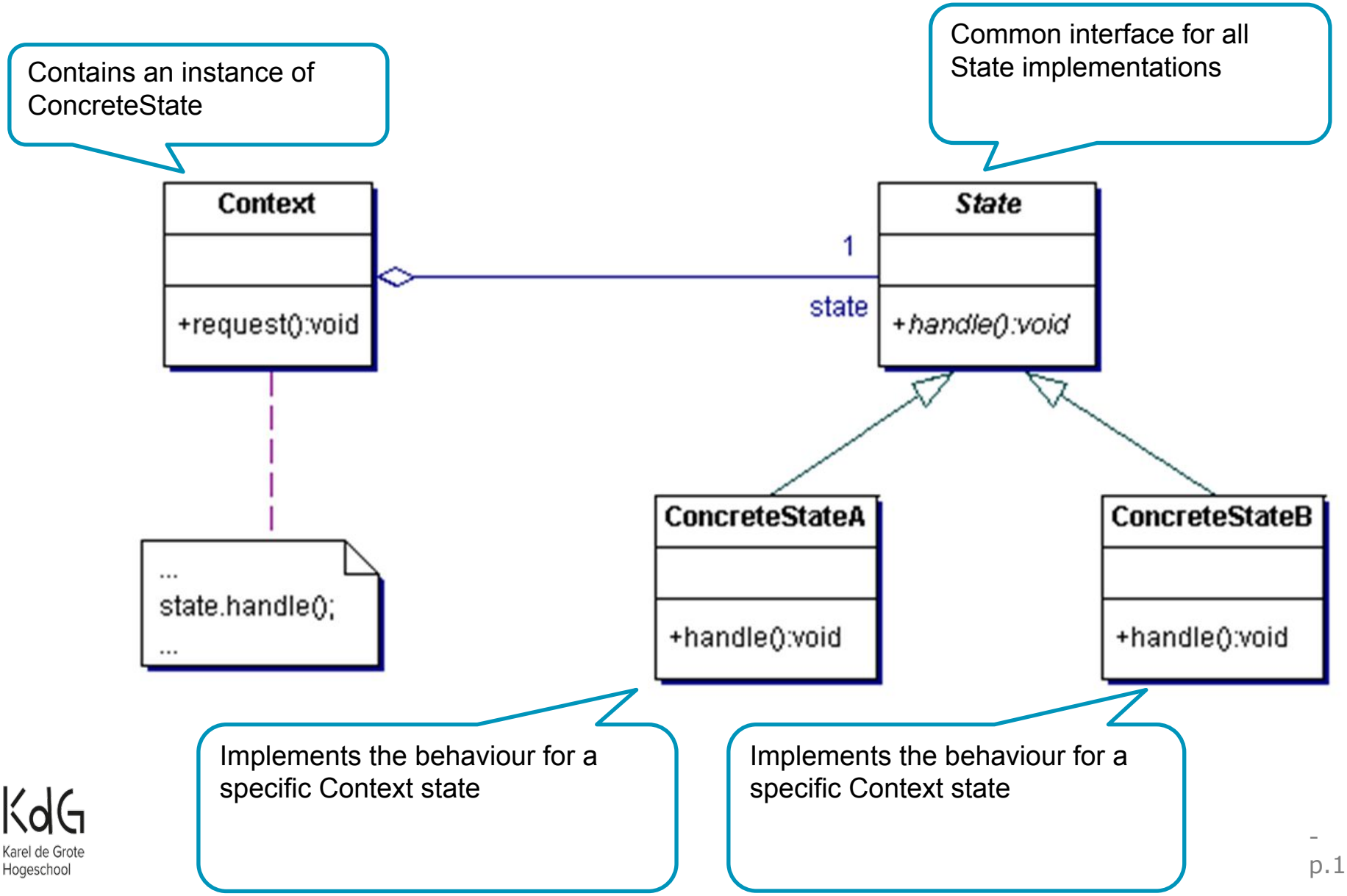
https://gitlab.com/kdg-ti/acs-programming-2.2/gof2

Example

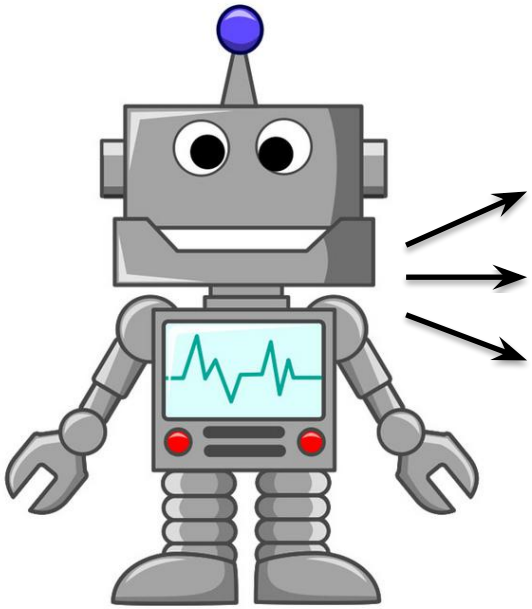
State

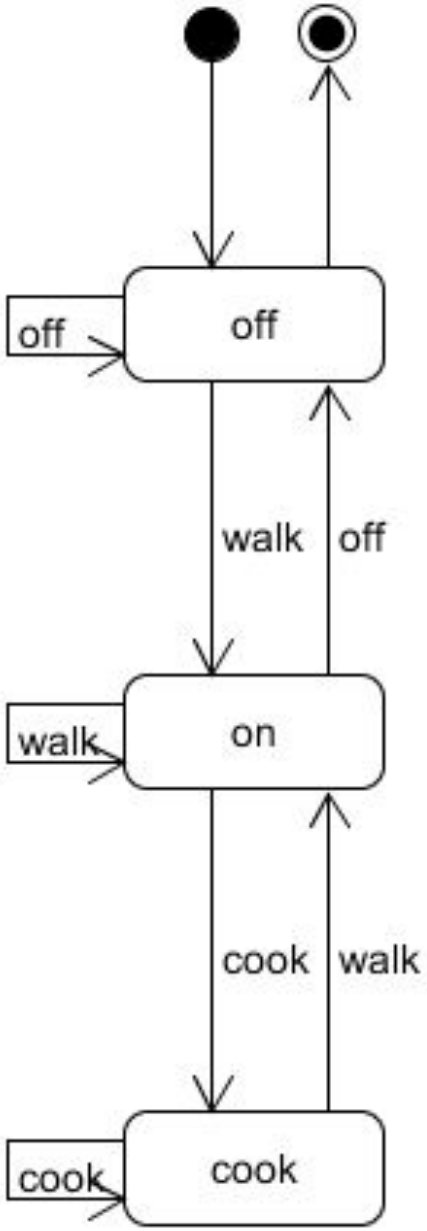
- Family: **Behavioural**
- Intent: Changes the behaviour of an object when its state changes. The object appears to change its class.
- Related patterns: *Composite, Singleton*

State: collaborations



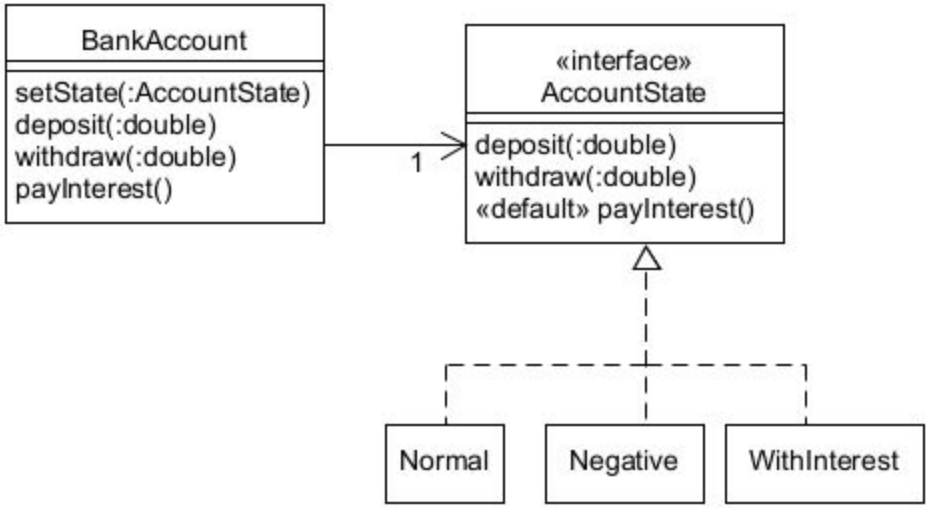
Demo: Robot





Democode: S1_State_Robot

Example: BankAccount



Example: BankAccount

```
public class BankAccount {
    private final String holder;
    private double balance;
    private double interest;
    private double minLimit;
    private double maxLimit;
    private AccountState state;
   //…
     public void deposit(double amount) {
       state.deposit(amount);
     }

     public void withdraw(double amount) {
         state.withdraw(amount);
     }

     public void payInterest() {
         state.payInterest();
    }
}
```

Example: BankAccount / State interface

```
public interface AccountState {
    void deposit(double amount);

    void withdraw(double amount);

    default void payInterest() {
        // default no interest is payed
    };
}
```

can put methods in an interface -> default method (most common implementation), just override it

can use an abstract class to avoid using a default method in an interface

Interface default method (since Java 8)

Example: BankAccount/State implementation 1

```
public class Negative implements AccountState {
    private final BankAccount account;
    private double balance;

    public Negative(BankAccount account) {
        balance = account.getBalance();
        this.account = account;
        initialise();
    }

    void initialise() {
        account.setInterest(0.0);
        account.setMinLimit(-500.0);
        account.setMaxLimit(1000.0);
    }

 public void deposit(double amount) {
        balance += amount;
        account.setBalance(balance);
        testStateChange();
    }
```

Example: BankAccount/State implementation 2

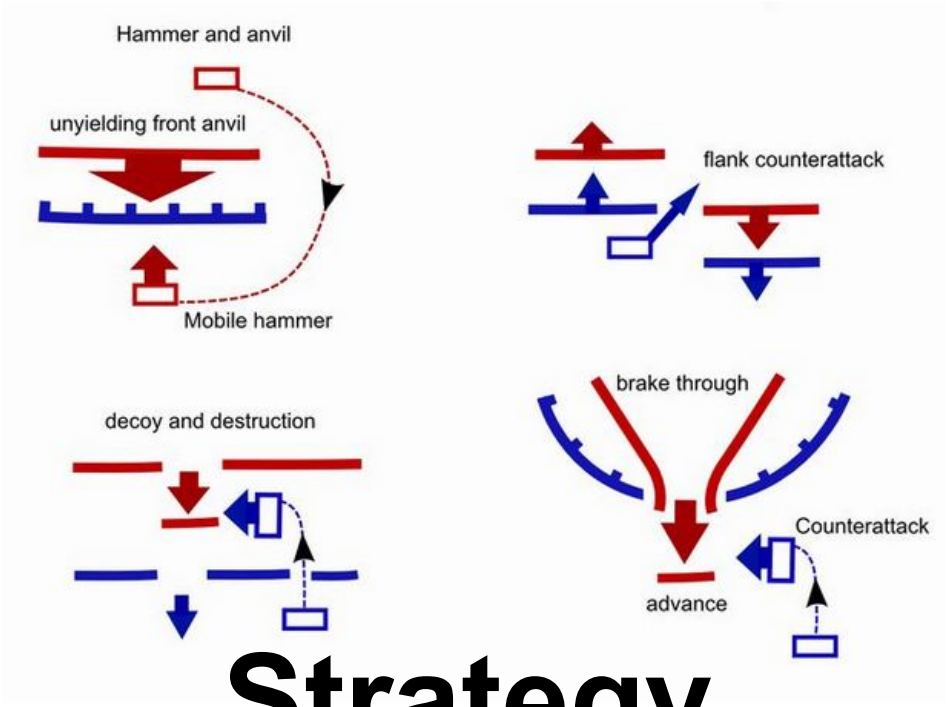
```
    public void withdraw(double amount) {
        System.out.println("Withdrawal refused!");
    }

    public void testStateChange(){
        if (balance > 0.0 &&
                balance < account.getMaxLimit()) {
            account.setState(new
              Normal(account));
        } else if (balance >
                account.getMaxLimit()) {
            account.setState(new
              WithInterest(account));
        }
    }

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

Example: BankAccount/main

```
public class DemoState {
    public static void main(String[] args) {
        BankAccount account = new BankAccount("Jos The Boss");
        System.out.printf("Bankaccount of %s%n",
account.getHolder());
        account.deposit(500);
        System.out.println(account);
        account.deposit(850);
        System.out.println(account);
        account.payInterest();
        System.out.println(account);
        account.withdraw(1100);
        System.out.println(account
        account.withdraw(500);
        System.out.println(account
        account.withdraw(500);
        System.out.println(account
    }
}
```

```
Bankaccount of Jos The Boss
Balance: 500.0  State: Normal
Balance: 1350.0  State: With
Interest
Balance: 1356.75  State: With
Interest
Balance: 256.75  State: Normal
Balance: -243.25  State: Negative
Withdrawal refused!
Balance: -243.25  State: Negative
```

emocode: s2_State_Bankrekening



Strategy

4

Example

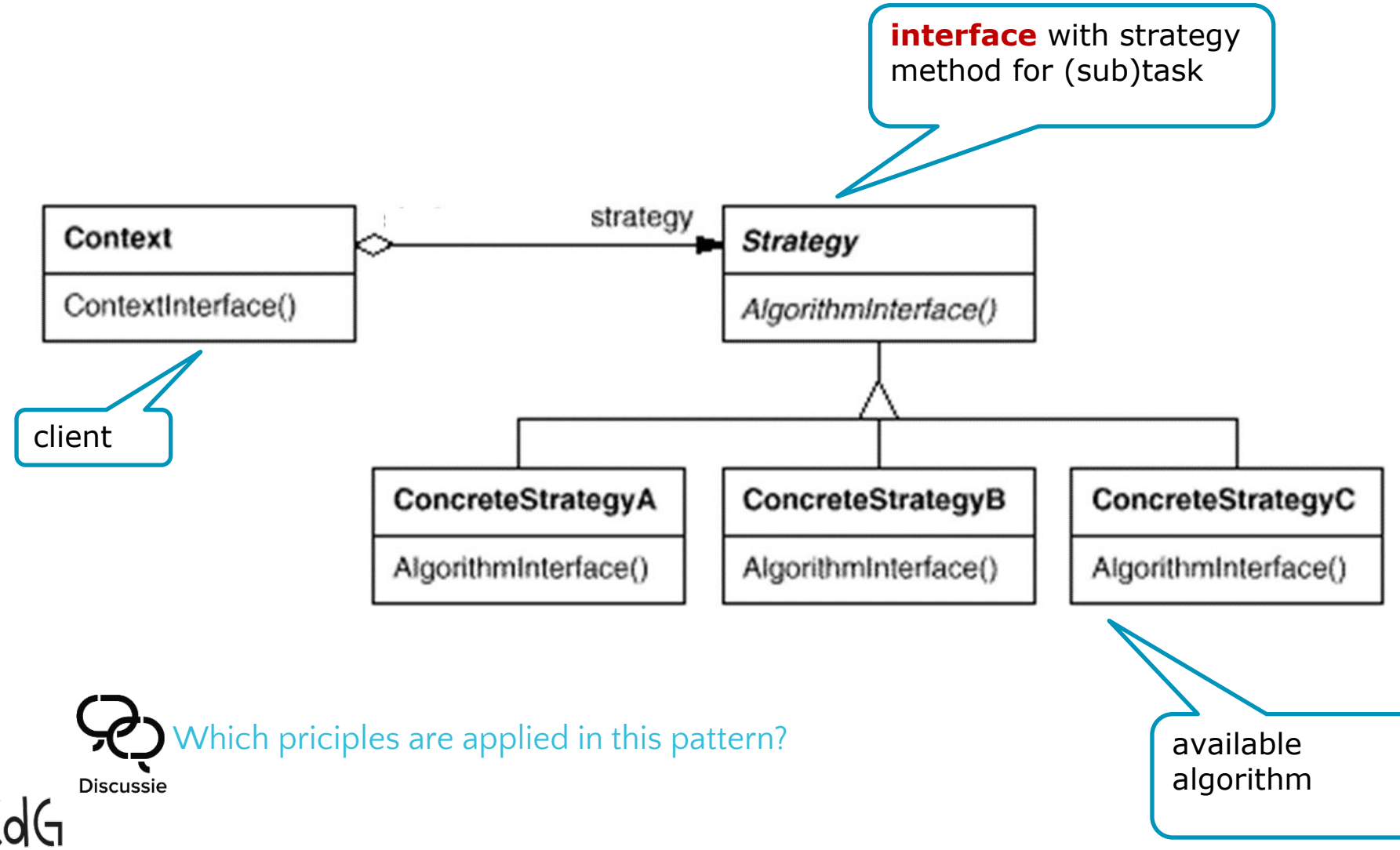
https://gitlab.com/kdg-ti/acs-programming-2.2/gof1

Strategy

- Also known as: Policy
- Family: behavioural
- Intent: *Define a group of algorithms for the same (sub)task, with the same interface. Algorithms can be replaced without changing the client.*
- Context:

  You want to (dynamically) choose the algorithm to use.
- Example:
  - Pass a `java.util.Comparator` to a sort method
- Related patterns: s*tate, template method, command*

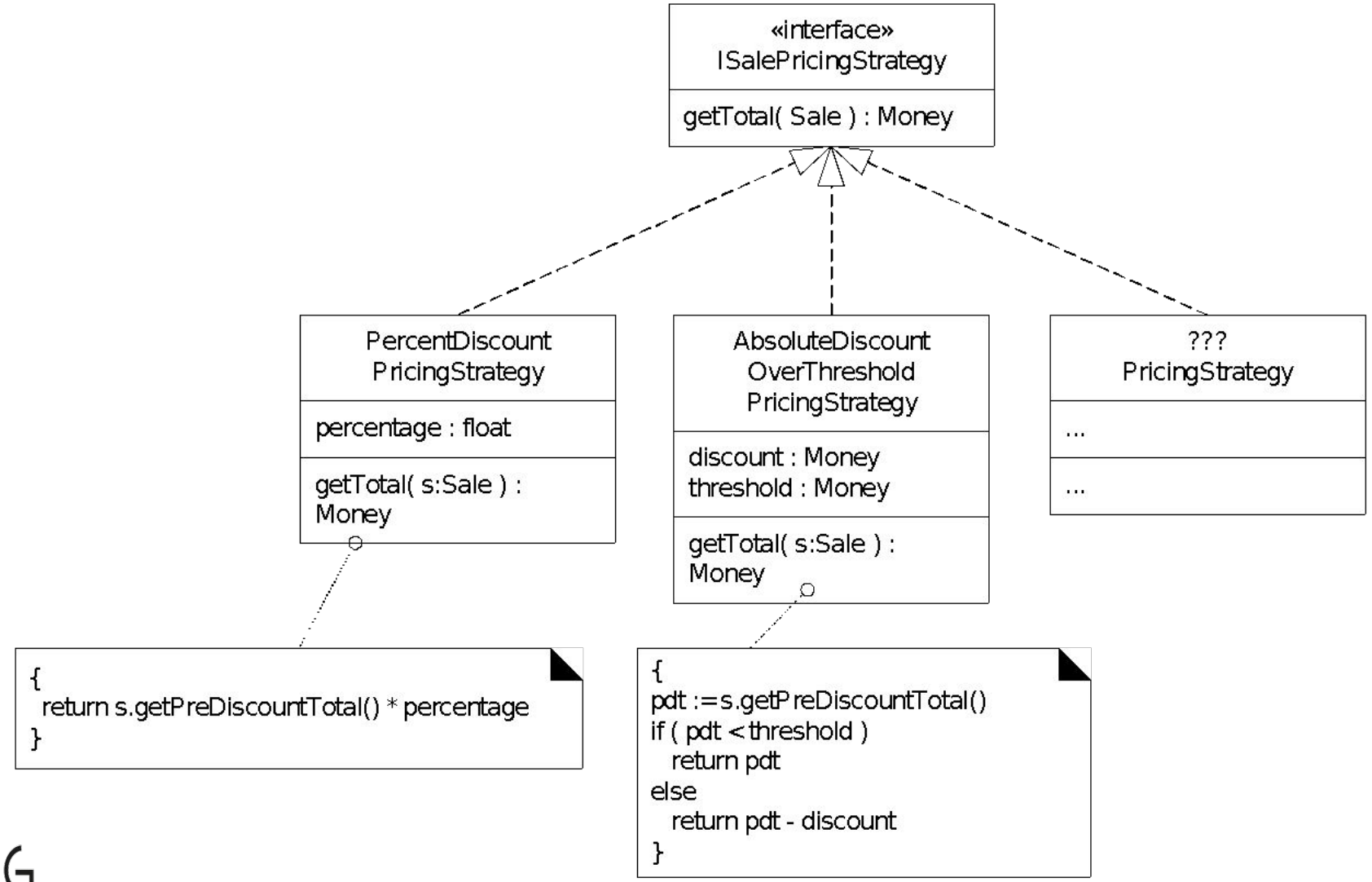
Strategy: collaborations



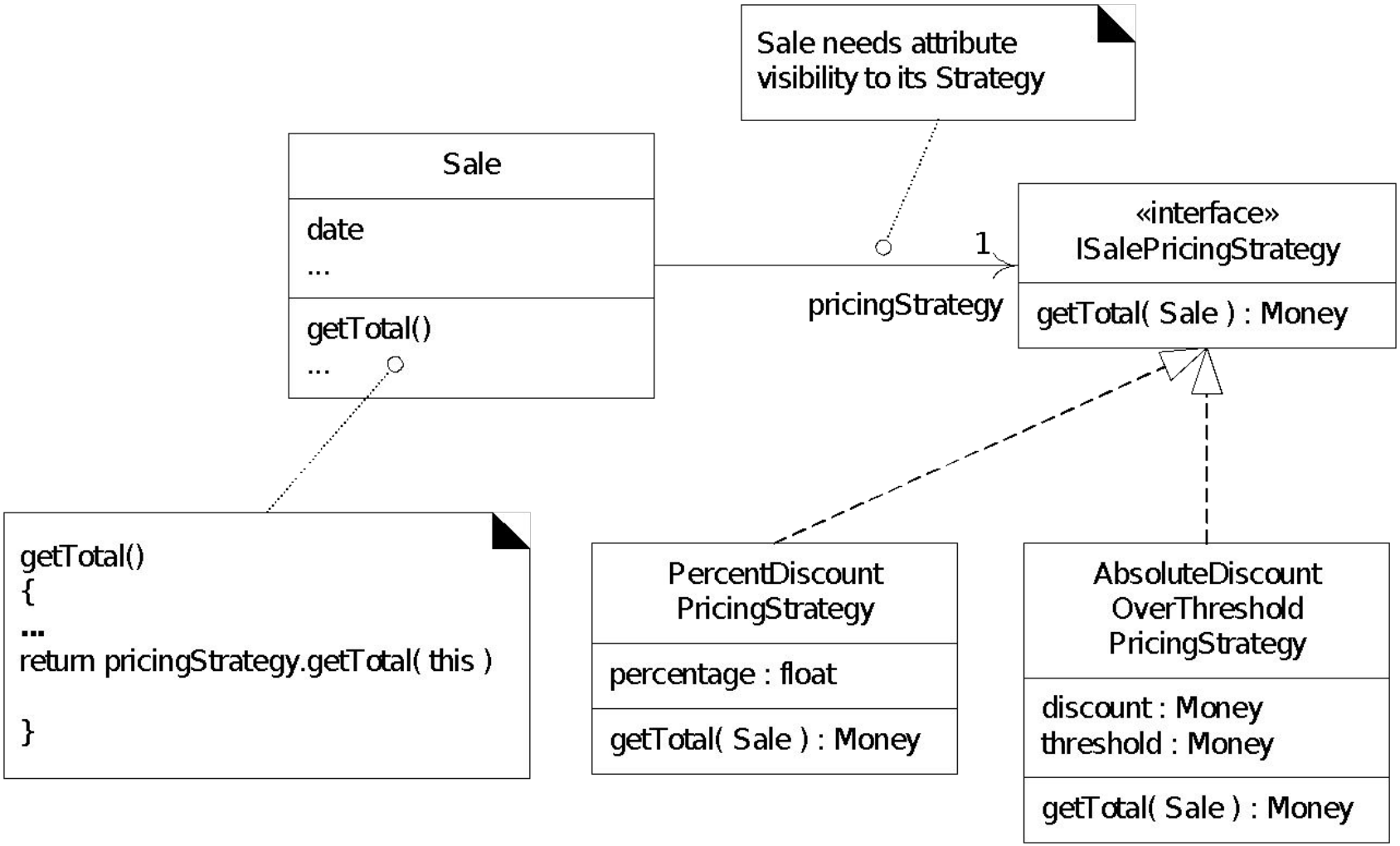
Discussie

Which priciples are applied in this pattern?

POS example: pricing strategies

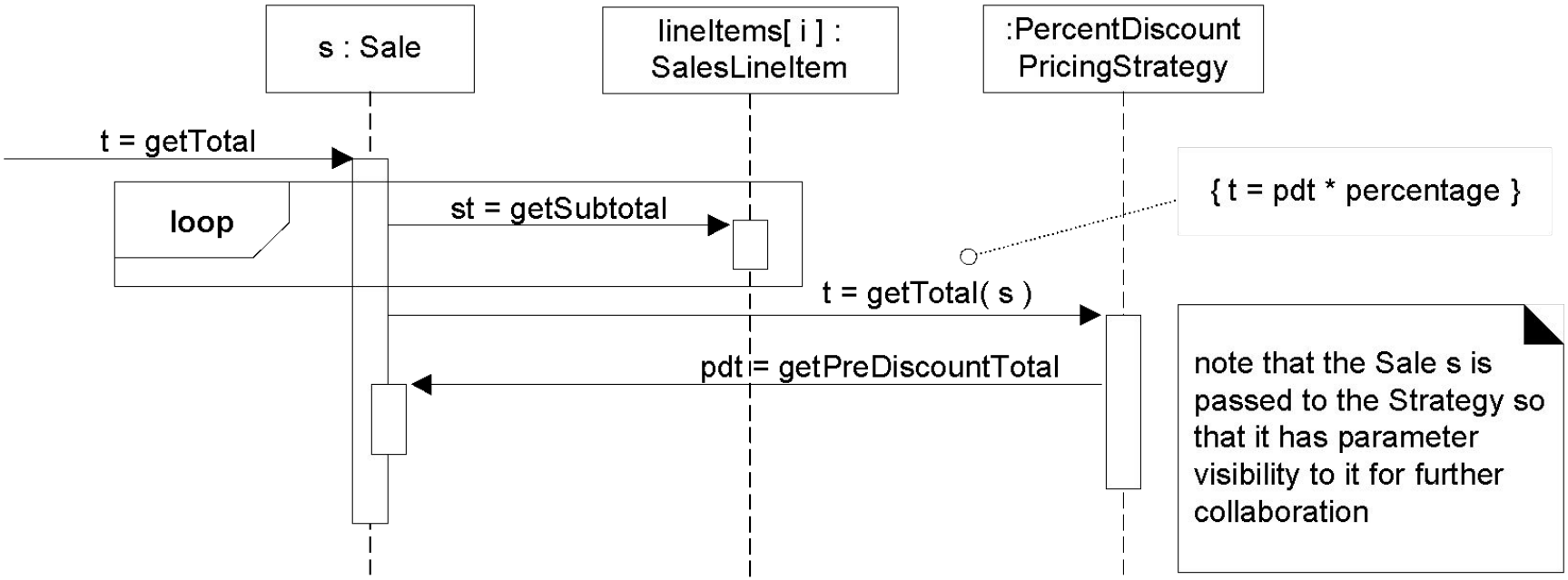


POS example: pricing strategies



Democode: strategy_discount_pos

POS example: pricing strategies



Strategy: discussion

- Sometimes you can use a lambda expression for the strategy
  - example Stream::filter(:Predicate)
- The client must be aware of available strategies
- Different algorithms may need different parameters
  - You could pass different information to the constructor
  - You can pass all information in the strategy interface and ignore part of the information in some of the algorithms (in the example, there will be PriceStrategy implementations that don't need Sale)

Related pattern: Template Method

- A product price is composed of different elements, each of which can vary (each price component can be a strategy)

```
public class Product {

  public double getPrice(TaxStrategy taxes,
    DepositStrategy deposit) {
    return basePrice*taxes.getTax(this)
      + deposit.getDeposit(this);
  }
…
}
```

- Een template method contains a fixed main algorithm. Each part of the algorithm (subalgorithm) can vary.
  - Example: a sales proce is always multiplied by a tax factor and has a deposit (for recyclable packing) added. The exact tax and deposit calculation is different for different products

Observer

Observer

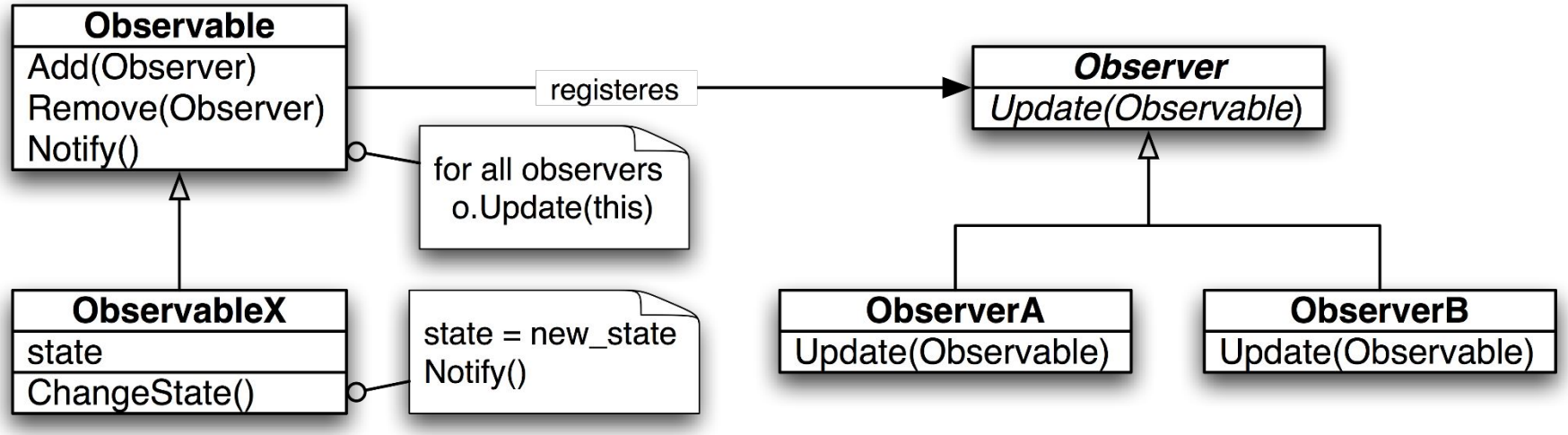
- Also Known as: Publish/Subscribe
  Hollywood principe (don't call us, we'll call you)
- Family: **behavioural**
- Intent: When an object (observable) changes state, other interested objects (observers) are automatically notified. There is no direct coupling between observables and observers.
- Exmples:
  - ✔ EventHandlers in event-driven programming (JavaFX)
  - ✔ mailing list

Observer: collaborations

2 roles: `Observer` ☐☐ `Observable`

1. `Observer` (subscriber)
   - ☐ registers with observable (is added to a list of interested objects)
2. `Observable` (publisher)
   - ☐ For each change, observable:
     - Iterates over the list of registered observers
       - Notifies each observer
         - observer refreshes (updates)

Observer patroon: UML



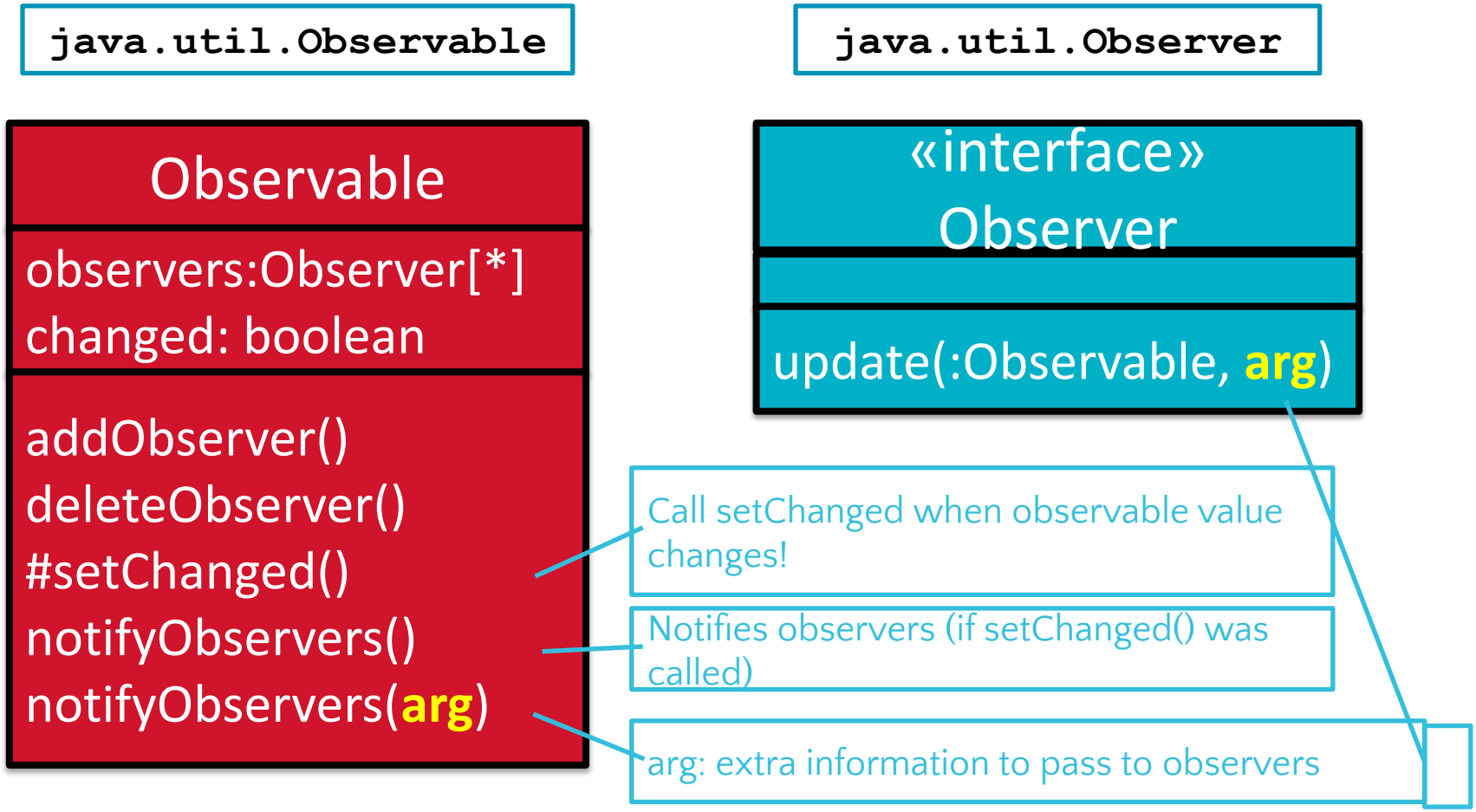
Observer in JavaFX

- Event handling in **JavaFX** uses the observer pattern:
  - The observable (publisher) is the JavaFX-component (e.g. **`Button`**)
  - Avoid coupling between Button component and action(s) to be executed when it is pressed
    - observer (subscriber) is the **`EventHandler`**
    - **`Eventhandler`** is an interface with a **`handle`** method
    - Each **`EventHandler`** must register with the Observable (e.g. **`button.setOnAction(myEventHandler)`**)
  - When the button is pressed, it calls the **`handle`** method on the **`EventHandlers`** (=observers)

Observer pattern in JDK

- **`java.util Observer/Observable`** is an example implementation of the pattern
  - Not for production use (see larman p471, deprecated in Java 9)
    - **`java.beans PropertyChangeSupport / PropertyChangeListener`** is a better alternative for production use. (online voorbeeld)
  - **`Observer/Observable`** is simpler and fine to demonstrate the pattern

Observer pattern in JDK



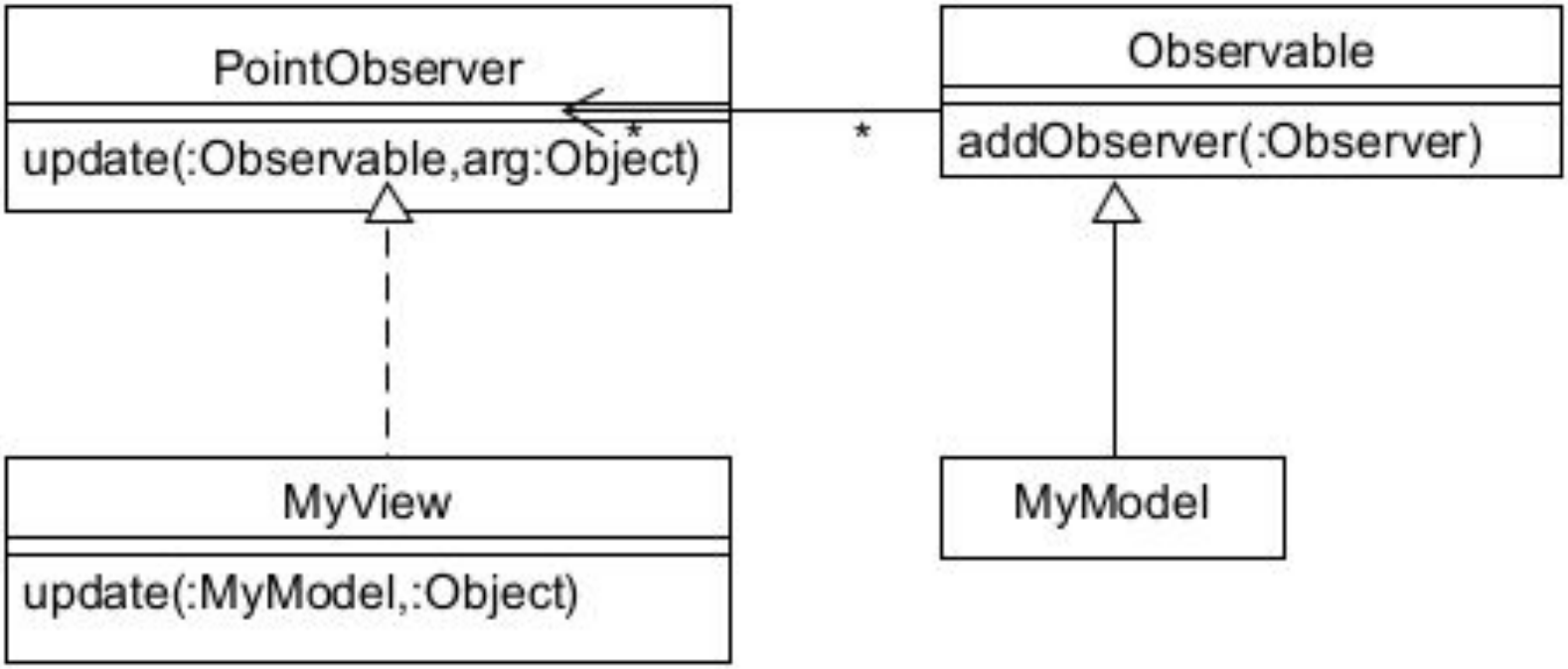
`java.util.Observer` interface

```
public interface Observer {

    void update(Observable observable, Object object);

}
```

Observable that was updated

optional: **extra information**

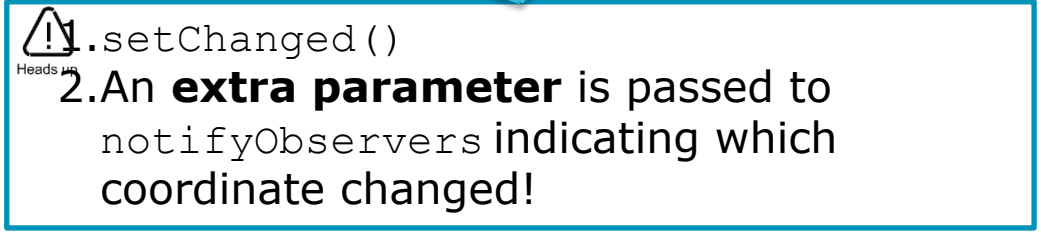
Example PointObserver



Example PointObserver (1)

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

     public Point(int x, int y) {
          this.x = x;
          this.y = y;
     }
     public void doubleX() {
          x *= 2;
          setChanged();
          notifyObservers("X");
     }
```

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

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



Example PointObserver (2)

extra argument

```
public class PointObserver implements Observer {

  public void update(Observable observable, Object object) {
    System.out.println(object
   + " changed, new values: "
       + observable);
    }
}
```

```
X changed, new values: (x,y) = (2,2)
Y changed, new values : (x,y) = (2,4)
```

KdG
Karel de Grote Hogeschool

Example PointObserver (3)

```
public class Demo {
    public static void main(String[]
args) {
        ObservablePoint point = new
ObservablePoint(1, 2);
        PointObserver observer = new
PointObserver();
        point.addObserver(observer);

        point.doubleX();
        point.doubleY();
    }
}
```

Democode: Observer

Observable without changing original class

What if you want to make a class Observable , but can't change the class. Possible strategies:

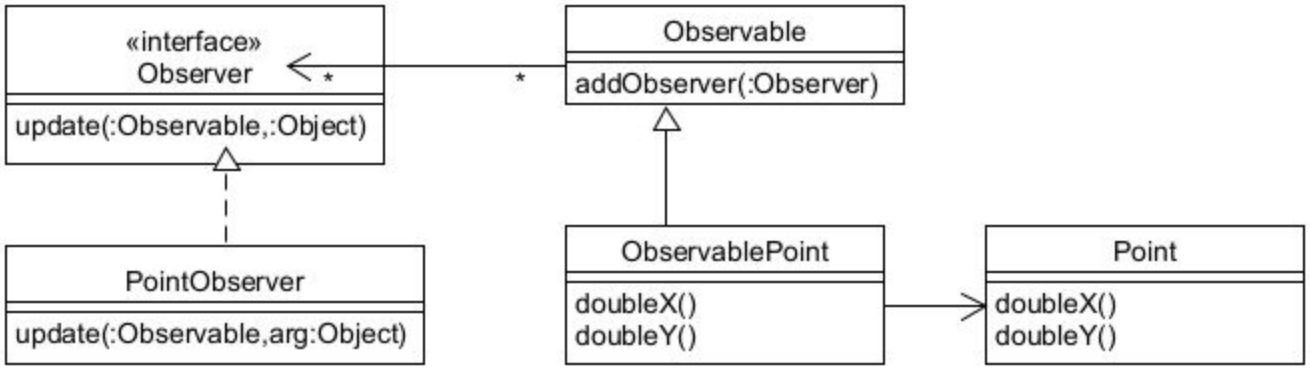
- Use **delegation**
  - New class (ObservablePoint) has original class (Point) as an attribute
  - ObservablePoint inherits from Observable
- Gebruik **overerving**
  - New class ObservablePoint inherits from original class (Point)
  - ObservablePoint has Observable (subclass) as an attribute

Original class (unchanged)

```
public class Point {
    private int x;
    private int y;

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

    public void doubleX() {
        x *= 2;
    }
```

```
public void doubleY() {
    y *= 2;
}

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

Solution with delegation (UML)



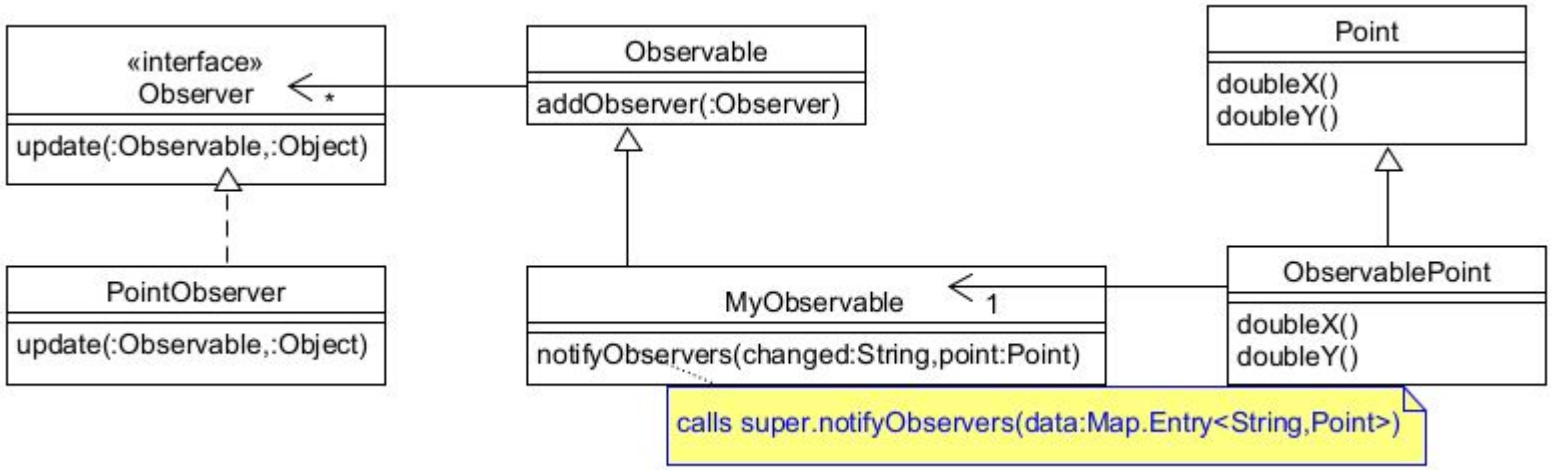
Solution with delegation (code)

```
public class ObservablePoint extends
Observable {
  private Point point;

  public ObservablePoint(int x,int y){
        point = new Point(x, y);
    }

  public void doubleX() {
    point.doubleX();
    setChanged();
    notifyObservers("X");
  }

  public void doubleY() {
    point.doubleY();
    setChanged();
    notifyObservers("Y");
  }

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

Delegates to encapsulated Point

Democode: delegation

Solution with inheritance (UML)



Remark: MyObservable can only inherit from one class (Observable) and thus can not inherit from Point.
This implies that the MyObservable passed as the ***first*** parameter in MyObservable##notifyObservers
PointObserver##update(:Observable,:Object) does not contain Point data. To make these data available they are passed in the ***second*** parameter.

Solution with inheritance (code)

```
public class ObservablePoint extends Point {
    private MyObservable notifier = new MyObservable();

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

    public void doubleX() {
        super.doubleX();
        notifier.notifyObservers("X",this);
    }

    public void doubleY() {
        super.doubleY();
        notifier.notifyObservers("Y",this);
    }

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

inheritance, uses super-methods

setChanged() is protected and cannot be called from Point

Solution with inheritance (code, continued)

```
public class MyObservable extends Observable {
  public void notifyObservers(String changed, Point point) {
        setChanged();
        notifyObservers( Map.entry(changed,point));
    }
}
```

This Observable does not have Point information. It receives Point information as a parameter and passes it into the notifyObservers method. It also receives and passes which attribute of Point was changed ("X" or "Y")

Democode: inheritance

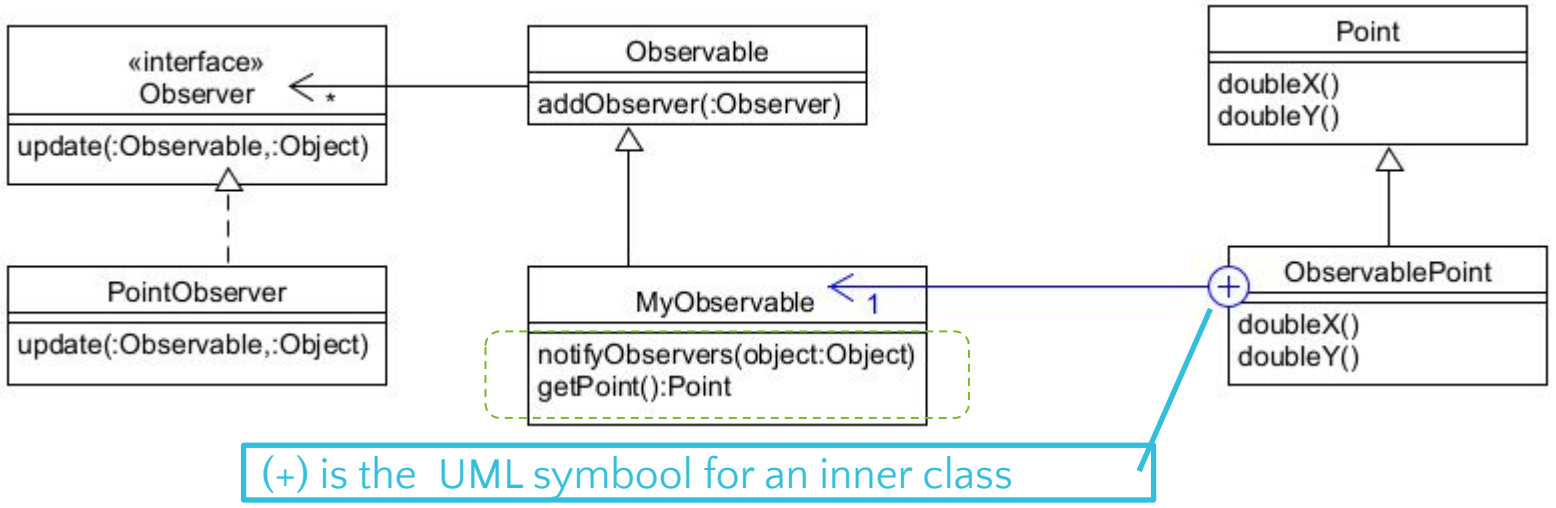
Solution with inheritance (code, continued)

extra argument

```
public class PointObserver implements Observer{

  public void update(Observable observable, Object o) {
    Map.Entry entry = (Map.Entry ) o;
    System.out.println(entry.getKey() +
      " changed, new values: " + entry.getValue());
  }
}
```

```
X changed, new values: (x,y) = (2,2)
Y changed, new values : (x,y) = (2,4)
```

Solution with inheritance/ inner class (UML)



By making MyObservable an inner class of ObervablePoint it gets access to the outer class data.
Inner classn can inherit from one class AND have access to data of the outer class (which can inherit from a different class)

Democode: inner_inheritance

Solution with inheritance/ inner class (code)

```
public class ObservablePoint extends Point {
    private MyObservable notifier = new MyObservable();

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

        public Point getPoint(){
            return ObservablePoint.this;
        }
    }// end inner class
  //… continued on next page
}
```

Inner class extends Observable

outerClass.this

Solution with inheritance / inner class (code)

```
//… continued from previous page
 public ObservablePoint(int x, int y) {
        super(x, y);
    }

    public void doubleX() {
        super.doubleX();
        notifier.notifyObservers("X");
    }

    public void doubleY() {
        super.doubleY();
        notifier.notifyObservers("Y");
    }

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

Solution with inheritance / inner class (code)

```
package observer;

import java.util.*;

public class PointObserver implements Observer {

 public void update(Observable observable, Object object) {
  ObservablePoint.MyObservable myObservable =
      (ObservablePoint.MyObservable) observable;
  System.out.println(object +
     "  changed, new values:  " + myObservable.getPoint());
 }
}
```

Cast Observable to Inner class

Observer Pattern in JavaFX

- Event handling in JavaFX uses the observer pattern:
  - The observable is the JavaFX-component (e.g. **`Button`**)
  - De observer is an **`EventHandler`**
  - When the component state changes, the **`EventHandlers`** (=observers) are norified using the **`handle`** method
  - Each **`EventHandler`** needs to registrer first with a component (e.g. **`button.setOnAction`**)

Observer pattern

- **Observable**

  = model-class (e.g. **`Game`**)

  - When the model changes, registered observers are notified

    - **`setChanged()`**
    - **`notifyObservers();`**

JavaFX event handling

- **Observable**

  = component (e.g. **`Button`**)

  - When the button is pressed, registered eventHandlers are notified.

Observer patroon

- **Observer**

  = view-class (e.g. **`GameView`**)

  - Observer registers with model:
    ```
    myGame.addObserver(myView);
    ```
  - Must implement interface:
    ```
    implements Observer
    ```
  - Event method in interface:
    ```
    public void update(
            Observable obs,
            Object arg);
    ```

JavaFX event handling

- **Observer**

  = EventHandler-class

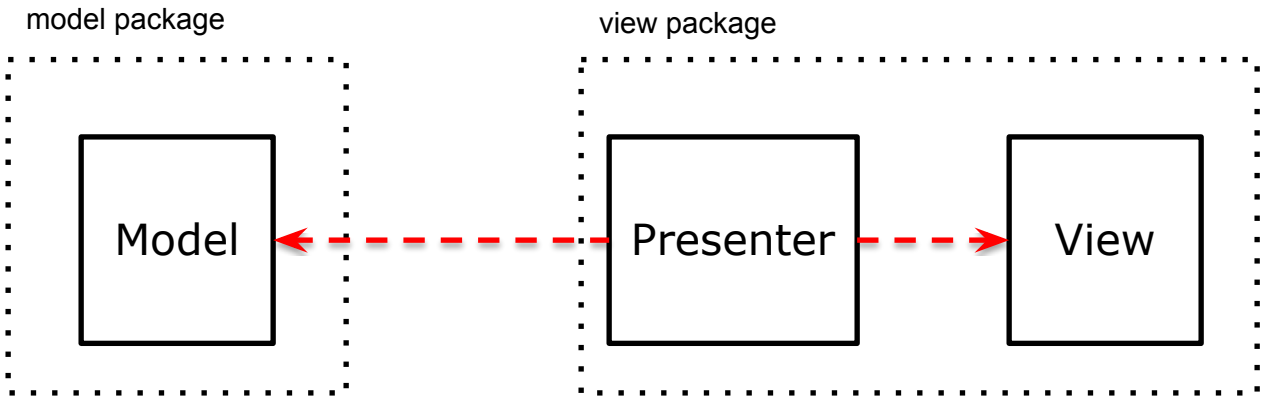
  - EventHandler registers with component:
    ```
    myButton.setOnAction(...);
    ```
  - Must implement interface:
    ```
    implements EventHandler
    ```
  - Event method in interface:
    ```
    public void handle(
         ActionEvent event);
    ```

Model View Presenter

Refresher

Recap: the **MVP** pattern:

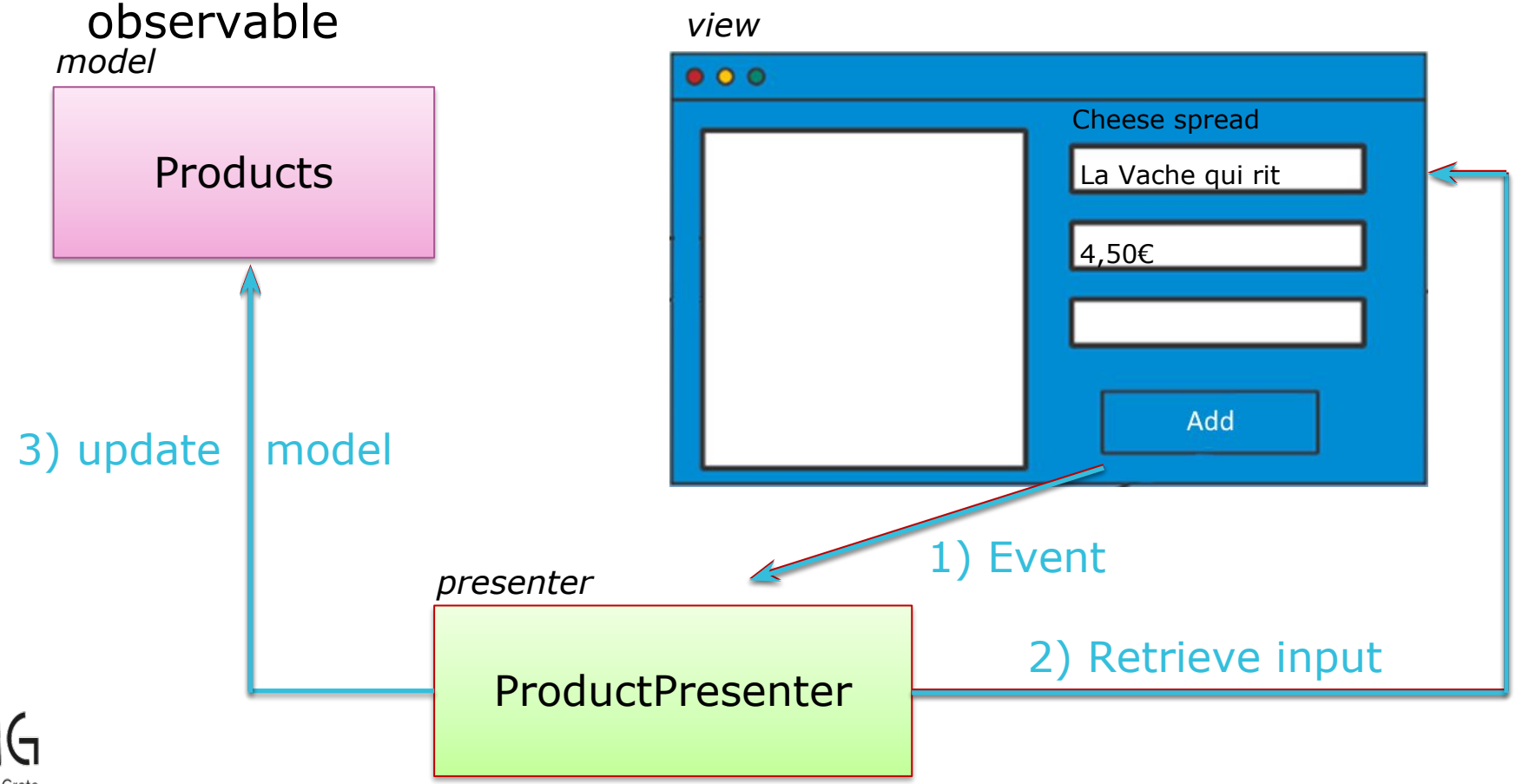
- **Model:** implements logic, no UI code (smart but ugly)
- **View:** draws UI, interacts with the user, (almost) no logic (stupid but beautiful)
- **Presenter:** links model and view;
  - Receives view (user) actions and forwards to model
  - Retrieves changed data from model and presents them through the view



Model View Presenter

Refresher

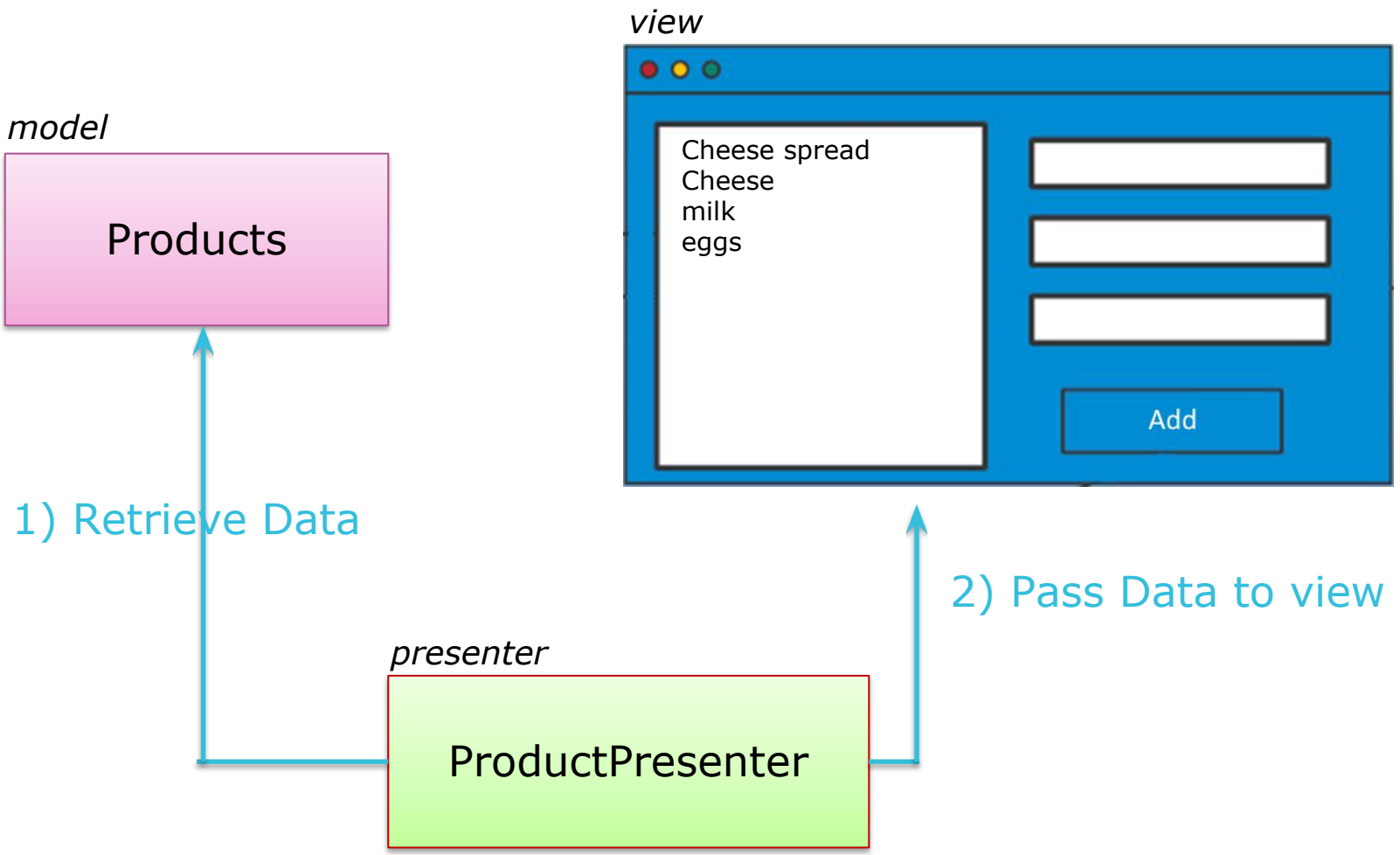
- Process user input(MVP):
  - In step 1 the presenter is the observer and the button is the observable



Model View Presenter

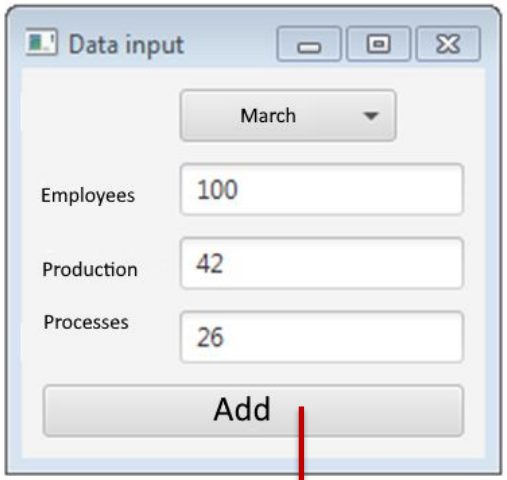
Refresher

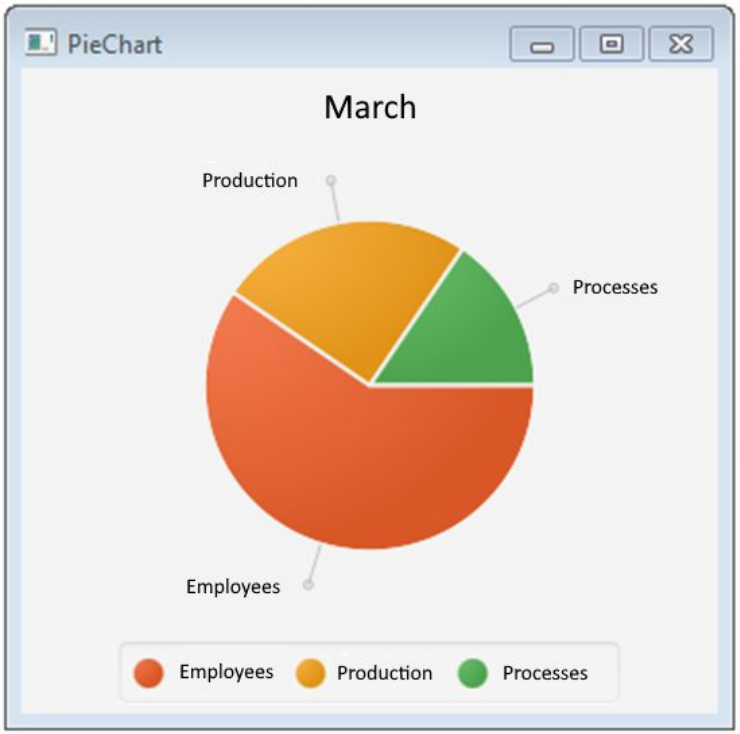
- Show model data (MVP):

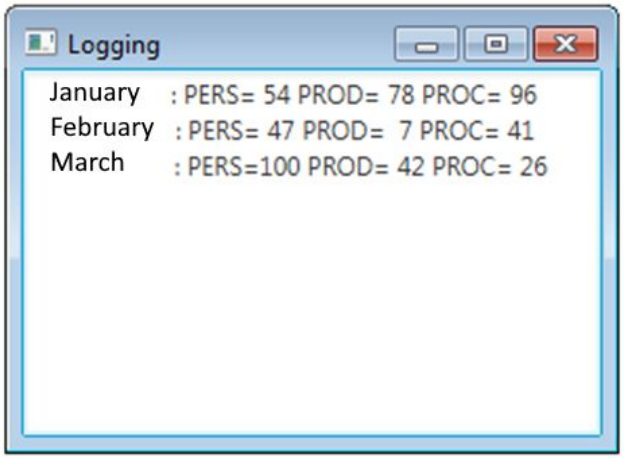


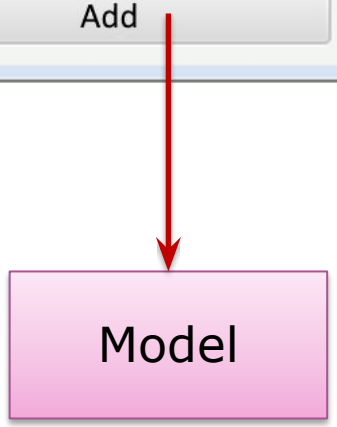
MVP?

1. When the input view (left) changes, the model is updated
   - View is an observable (== MVP)



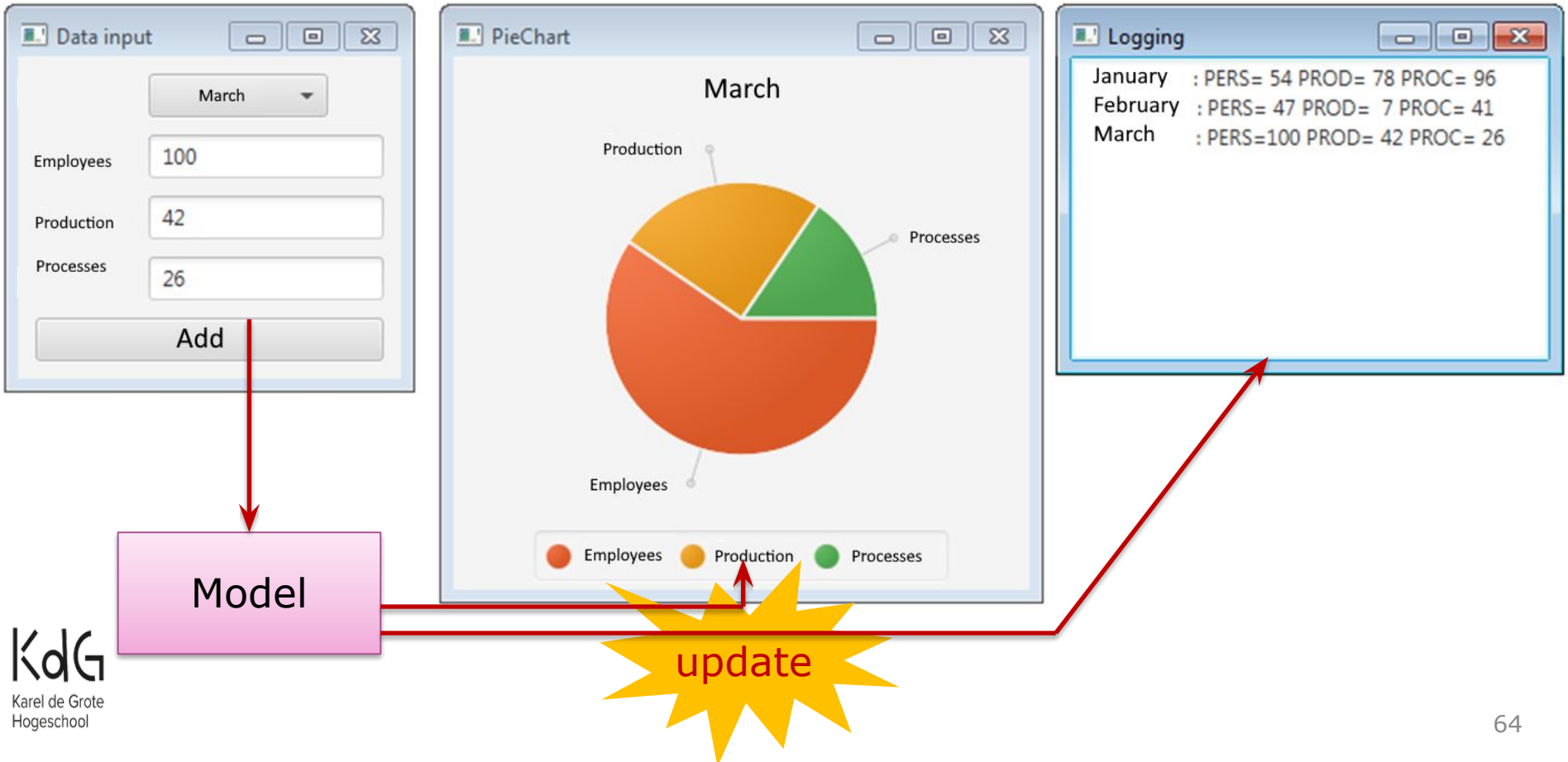






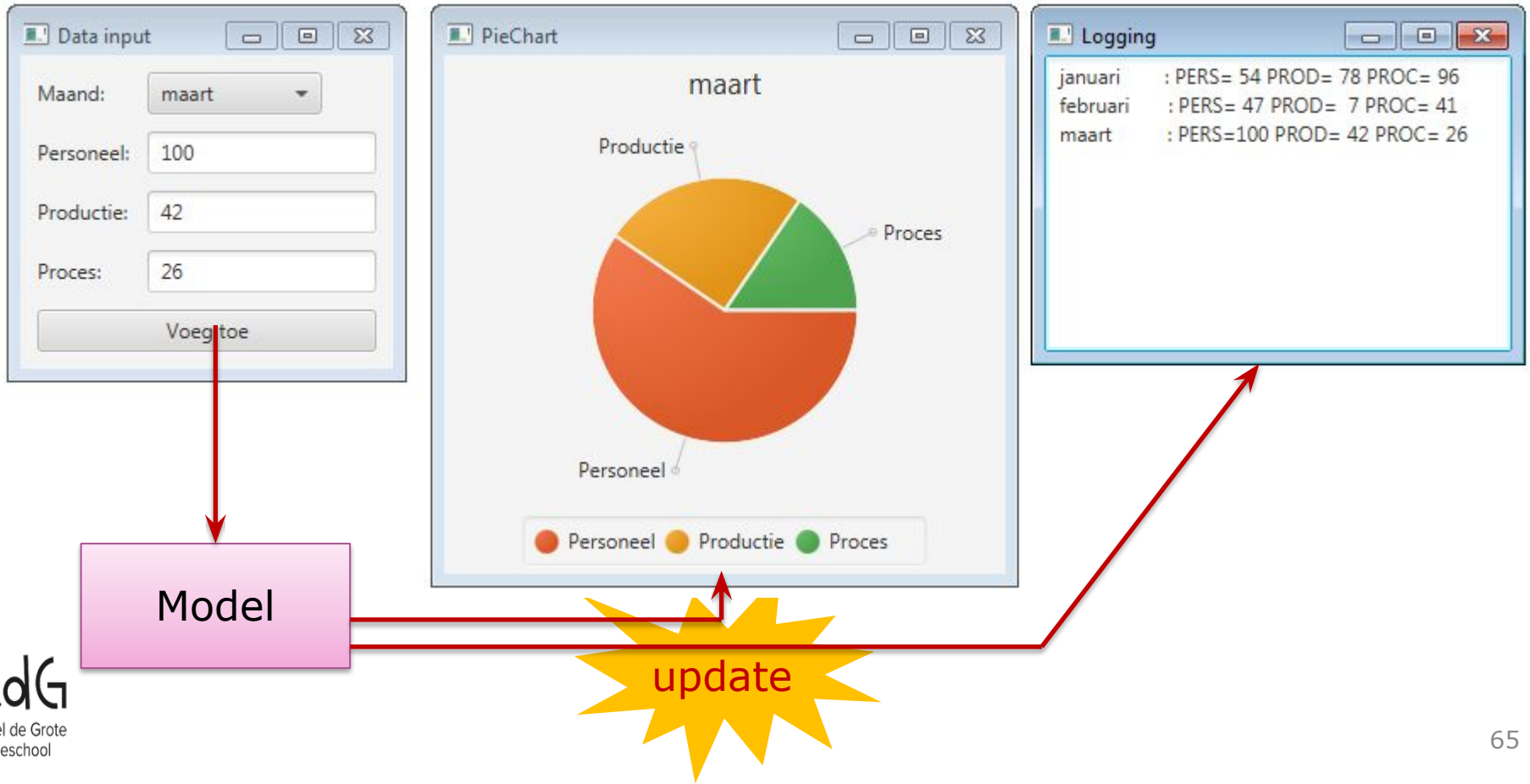
MVP?

- Upon a model change, update both output views



MVP?

- Upon a model change, update both output view
  - Model is also an observable, output views are observers



ConclusiON MVP - Observer

- An observable/observer relation between model and view allows multiple views to be updated when the model changes
- Requires the model to have access to the views
  - Presenter normally is a strict boundary between model and view
  - Technically not always possible (web appliciation)
- This interaction structure is often called MVC (Model View Controller)

Summary

1. Code: values and principles
2. Design patterns introduction
3. State
4. Strategy
5. Observer