Lecture:
Software Engineering, Winter Semester 2011/2012

some more

# Design Patterns

it‘s about the **Observer** pattern,
the **Command** pattern,
**MVC**, and some **GUI**

Andreas Naderlinger

UNIVERSITÄT SALZBURG

# Design Pattern

- “[...] **describes a problem** which occurs over and over again […], and then **describes the core of the solution** to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice“.
  *Christopher Alexander: Pattern Language for <u>architecture</u> (late 70s)*

- Kent Beck, W. Cunningham: applied ideas to <u>software</u> (late 80s)
- Erich Gamma et al.: Design Patterns: Elements for Reusable OO Software (90s) [GOF]

- *In 70s: MVC pattern (Smalltalk @Xerox Parc)*

# SO WHAT?

Do I need it?

# Why is it good to know them (or some of them at least)?

- Hmmm.. It’s required for the exam, isn’t it.
- But also:
  - Common language; facilitates discussion
  - Identify patterns (design ideas) in existing code.
  - Software structure
    - loose coupling
    - composition vs. inheritance
  - …
- Popular topic in job interviews

# Danger

- Complexity !

Applying DP may lead to unnecessarily complex designs.

# Example:

- A small weather station app.

# CODE

- weatherdefault
  - Add a View (Temperature)
  - Add also humidity to the view
  - Want to have a separate Humidity View

# What we want (in general)

- We want to
  - develop different classes independently
  - keep existing classes unchanged when adding new features
  - reuse components
- What we don’t want:
  - Tight coupling between two components that could also be used without one another.
- What helps?
  - Separation of concerns
  - Encapsulation
  - Loose coupling
  - “Program against an interface, not an implementation”

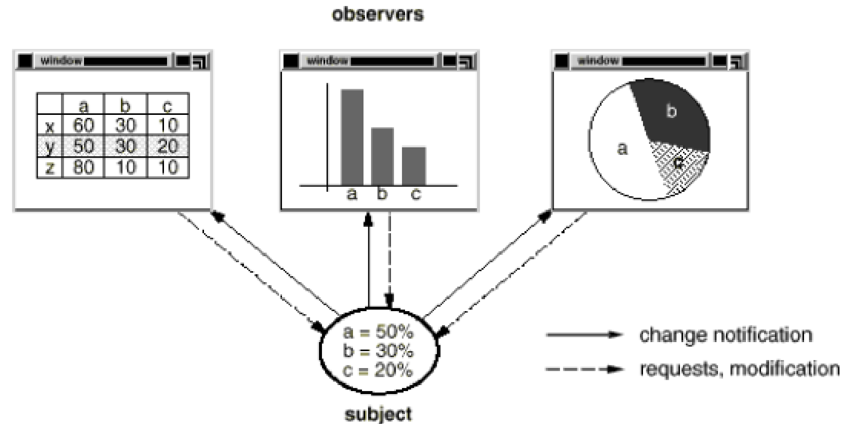
# Now what's the problem?

- What if we want a second view?
- What if we do not want a GUI at all?
- What if …?

- The model should never know about its views.

# The **Observer** pattern [GOF]

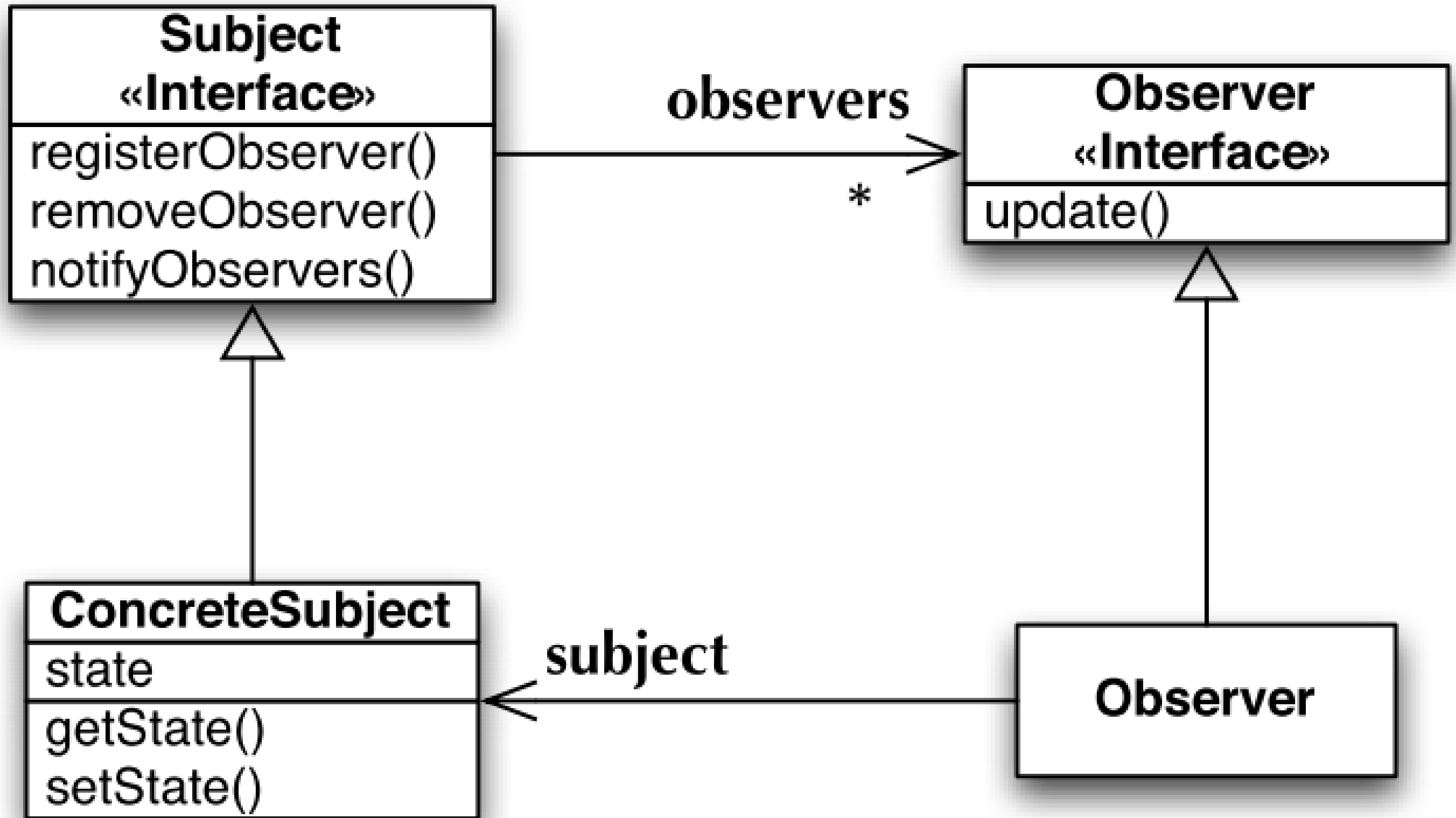
- **Motivation**



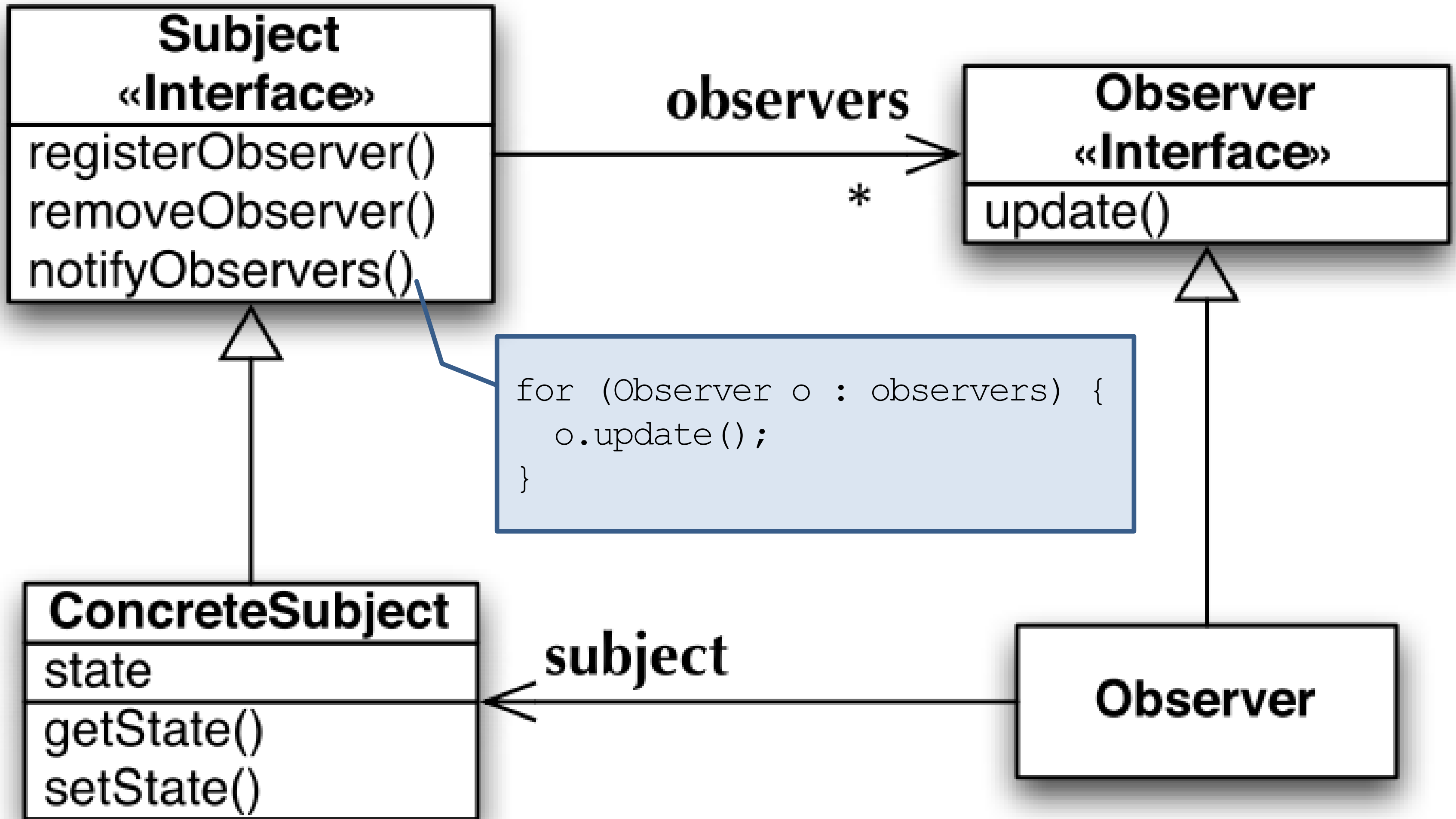

- **Intent**
  Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

- **Applicability**
  - When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
  - When a change to one object requires changing others, and you don't know how many objects need to be changed.
  - When an object should be able to notify other objects without making assumptions about who these objects are. → In other words, you don't want these objects tightly coupled.

# UML



# UML



# CODE

- weatherobserver
  - Change tempView to temp+humView
  - Add a new HumView

```java
import java.util.*;

public class WeatherData {

  private List<WeatherObserver> observers
    = new ArrayList<WeatherObserver>();

  private double temperature;
  private double humidity;

  public double getTemperature() {    return temperature;  }
  public double getHumidity() {    return humidity;  }

 public void setData(double temperature, double humidity) {
    this.temperature = temperature;
    this.humidity = humidity;
    notifyObservers();
  }

  public void registerObserver(WeatherObserver o) {
    observers.add(o);
  }

  public boolean removeObserver(WeatherObserver o) {
    return observers.remove(o);
  }

  private void notifyObservers() {
    for (WeatherObserver o : observers) {
      o.update();
    }
  }
}
```

```java
public interface WeatherObserver {
  public void update();
}
```

```java
//import awt & swing

public class TempView extends Jframe
  implements WeatherObserver {

  private JLabel label = new JLabel();
  private WeatherData wd;

  public TempView(WeatherData wd) {
    super("Temperature");
    this.wd = wd;
    //…
  }

  public void update() {
    label.setText(""
     + wd.getTemperature());
  }
}
```

…this is the “**pull**” approach

# Pull vs. Push

- Two approaches:
  - **Push**:
    - The subject sends detailed information about the change.
    - E.g., all information is encapsulated in an object ("Event")
  - **Pull**
    - The subject only informs the observes that there was a change. Observers need to request details themselves.

# Interaction

- **Registration**: some component registers as an observer (right at the program initialization or later)
- **Notification**: subject notifies all observers about some change
- **Update**: observer decide on itself how to react on the change. Depending on Push/Pull: observer may fetch information from the subject
- **Deregistration**: if an observer does not want to get further notifications

# Observers in Swing: **Listeners**

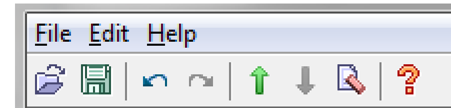
- CODE: swingobserver
  - Button: add actionListener

- Use built-in Java classes: Yes/No?
- Update order?
- Java Beans?

# The **Command** pattern [GOF]

- **Motivation**

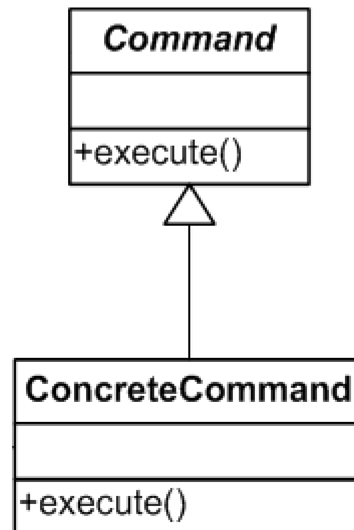
  Issue requests to objects without knowing anything about the operation being requested or the receiver of the request. E.g., UI toolkits




- **Intent**

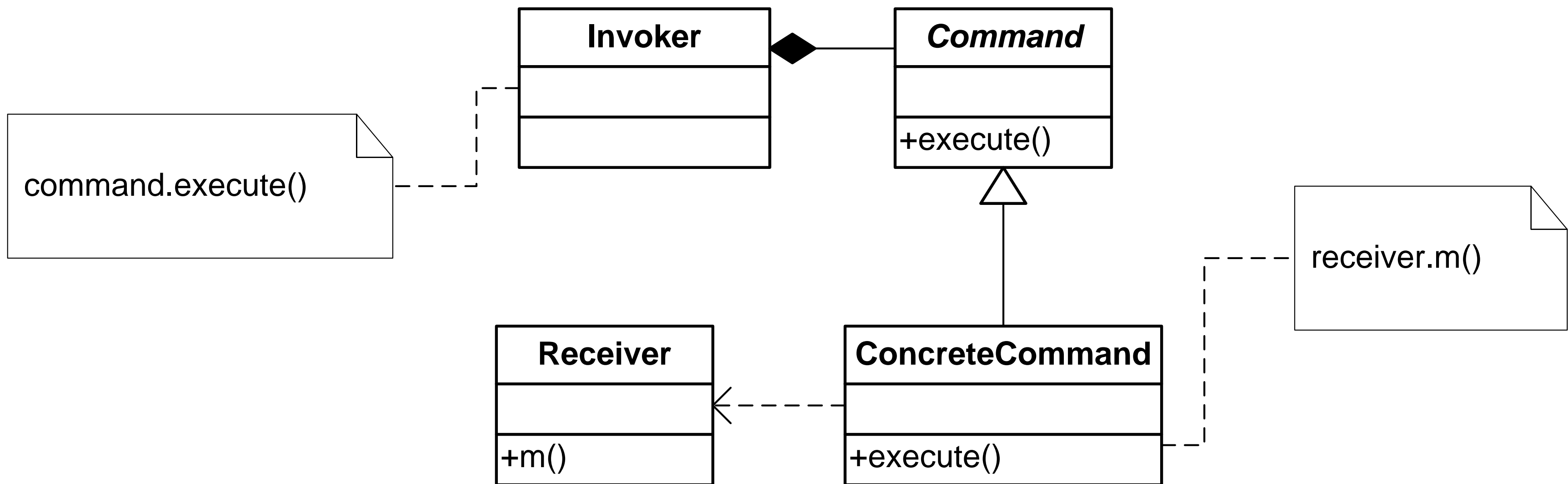
  Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

# The **Command** pattern [GOF]




Why an object? It's about a function, right?
*Callback function*: A function that is registered somewhere and called at a later time.
In C/C++: function pointer
In Java: *Functor* (usually a class with a single method to mimic function pointers)
In C#: Delegates

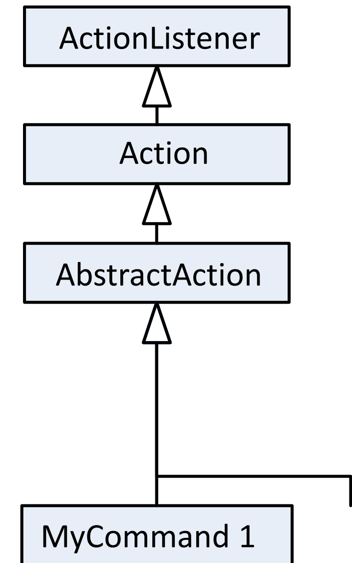
# The **Command** pattern [GOF]



- **Applicability**
  - Queue requests, execute them at different times.
  - Log commands. Reapply them after a system crash.
  - Compose Macro-Commands (Composite pattern).
  - Transactions.
  - Undo?

# The Command pattern in Java

- Callbacks
  - The AWT (framework) calls back into the application code.
- What about exceptions?
  - General problem with callbacks!
- Undo/Redo



# CODE

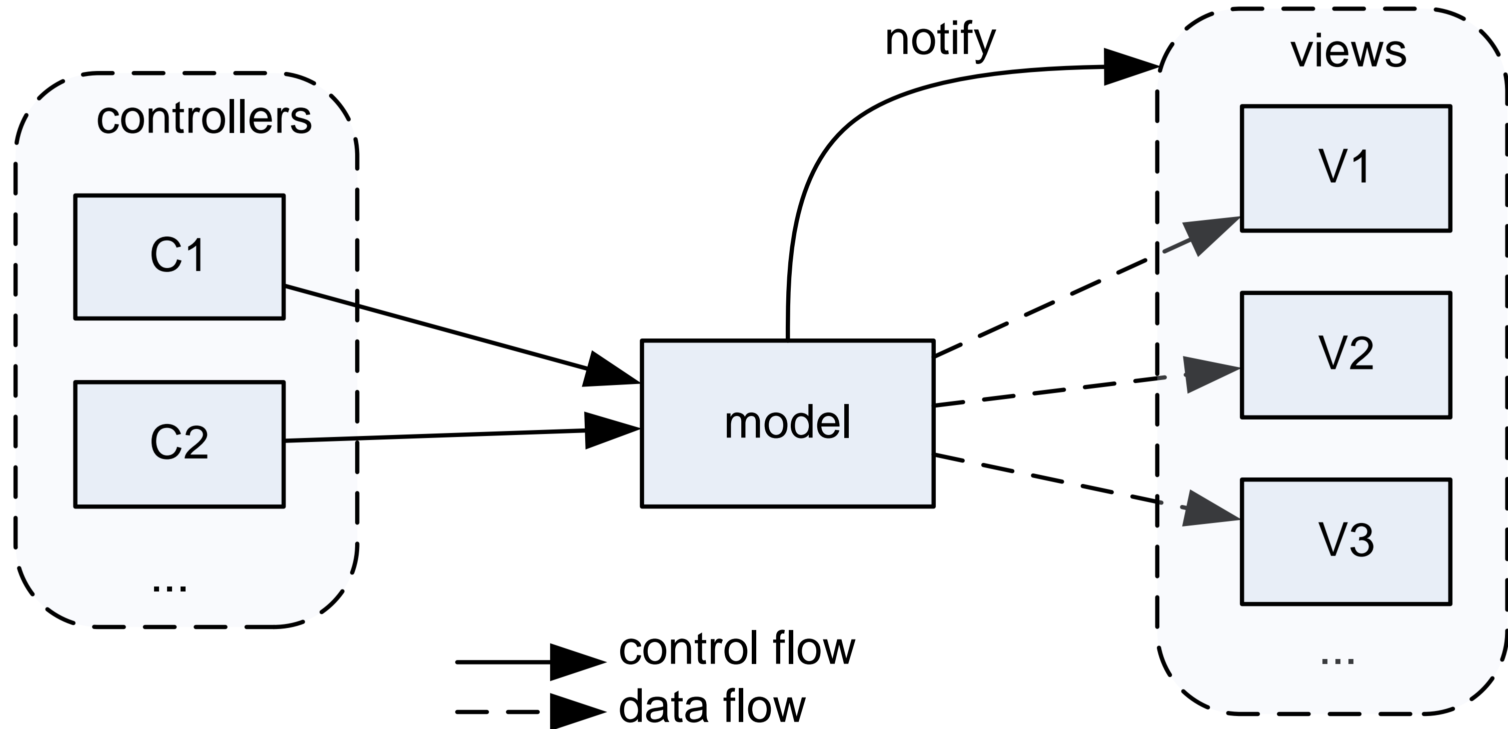
- commandpattern

# Model-View-Controller

- What is it? – Ask 5 people, get 5+ answers
- Architectural pattern that combines
  - **Observer**
  - Strategy
  - Composite
  - Decorator
  - Factory Method
  - …?
- was introduced for building user interfaces in Smalltalk (70s)

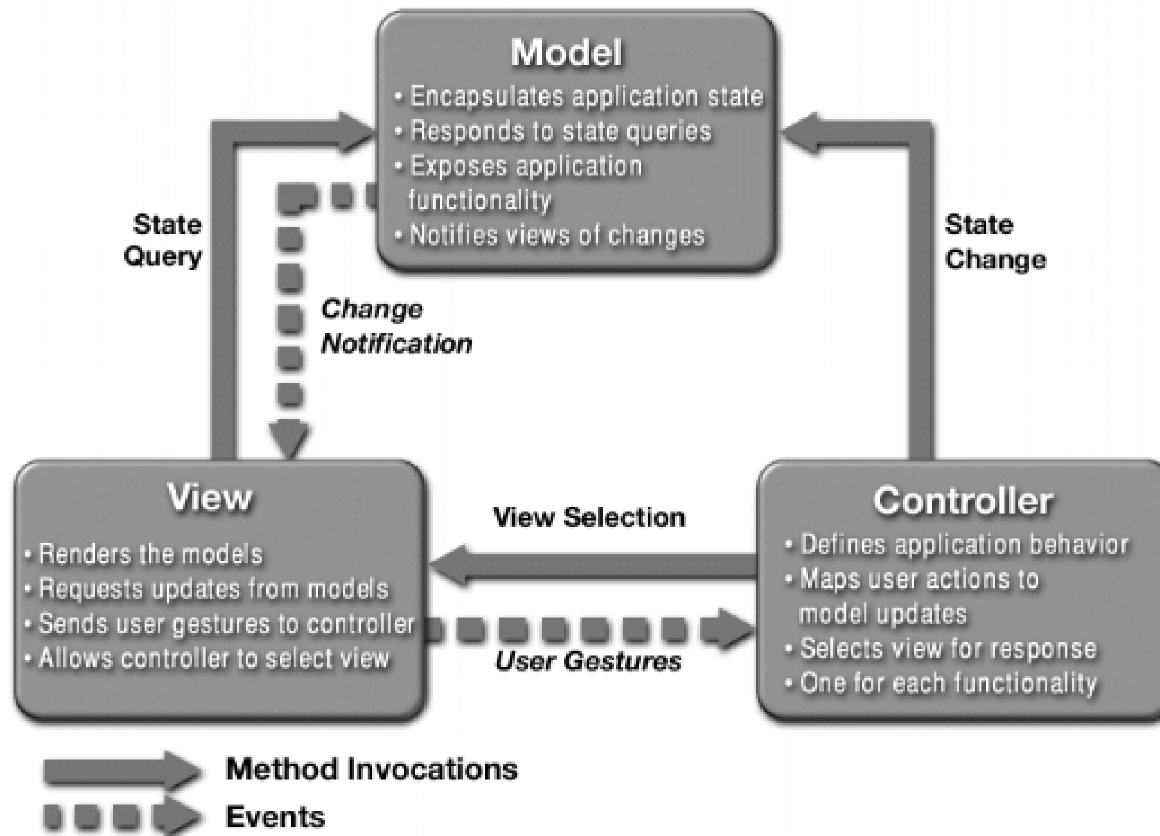
- **M – model**
  - data and methods to manipulate the data; *business logic*
  - may interact with databases, …
  - typically consists of multiple classes
- **V – view**
  - visual representation of the model (e.g. GUI)
- **C – controller**
  - takes user input
  - modifies the model

# MVC – in principle …




with the **Observer** pattern at the core

# X variants



# M-V-C vs. M-VC

- Sometimes V and C are combined (Swing: Model and UI component)

- A dedicated controller class
  - Flexible, extensible design
  - Simplifies View
  - Overcome gap between model-API and user interface
    - E.g. setTemperature(int value) vs. incTempByOne/decTempByOne
  - Web applications
  - …

# Conclusion

- Observer
  - When a change to one object requires changing others
- Command
  - Encapsulates a request in an object
- MVC
  - Always separate your business logic (model) from the GUI (view)!

Thanks!