Software Architectures

From Design Patterns to Enterprise Architecture

Design Patterns I

Last week on “Software Architectures”

- Unit tests & Refactoring
- Notion of responsibility
- Architecture evaluation
- Single-Responsibility Principle
- Interface Segregation Principle
- Abstraction & Layers
- Dependency Inversion Principle
- Command & Query Separation Principle
- Coding for robustness: Exceptions, Assertions, Checks

Design Patterns

Motivation

What is a design pattern?

- A pattern describes…
  - a problem which occurs over and over again in our environment
  - 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

Design Patterns in Software Architecture

- A Design Pattern describes a solution for a problem in a context
- A pattern has a name
- The problem has to reoccur to make the solution relevant in situations outside the immediate one
- It has to be possible to tailor the solution to a variant of the problem

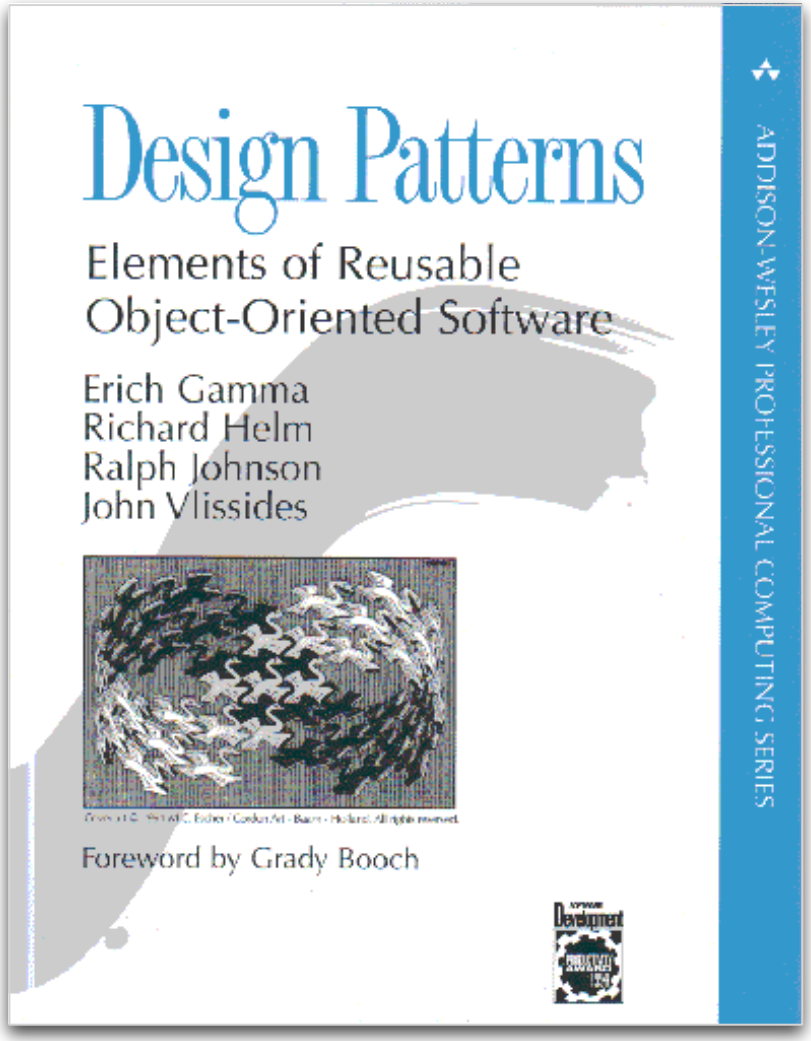
Software Design Pattern

- A software design pattern describes…
  - a commonly recurring structure of interacting software components
  - that solves a general software design problem
- within a particular context.

Structure of a pattern

According to Gamma, et el.

1.	▶ Name ▶ Intent
2.	▶ Motivation ▶ Applicability
3.	▶ Structure ▶ Participants ▶ Collaboration ▶ Implementation
4.	▶ Consequences
5.	▶ Known Uses ▶ Related Patterns

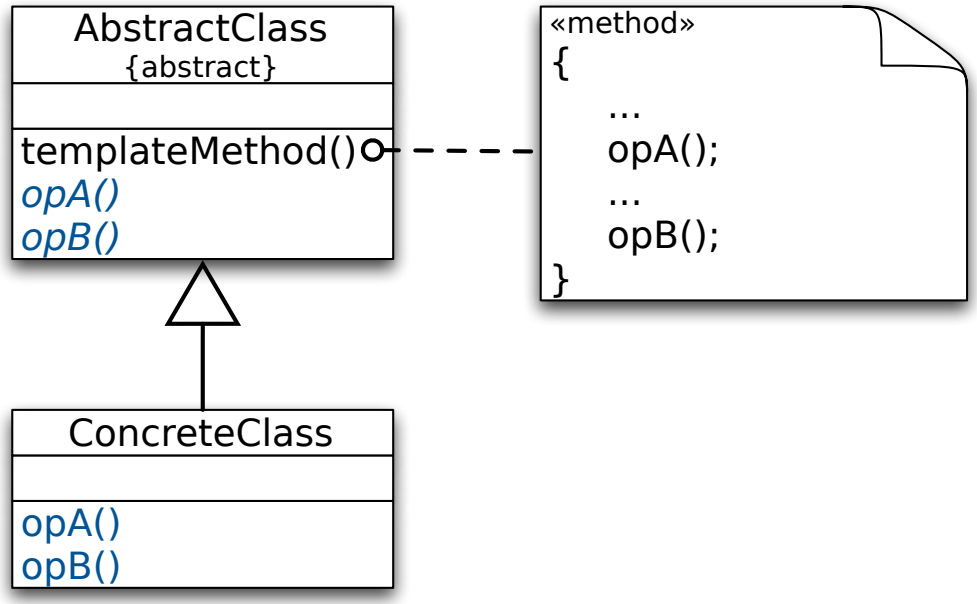


Patterns

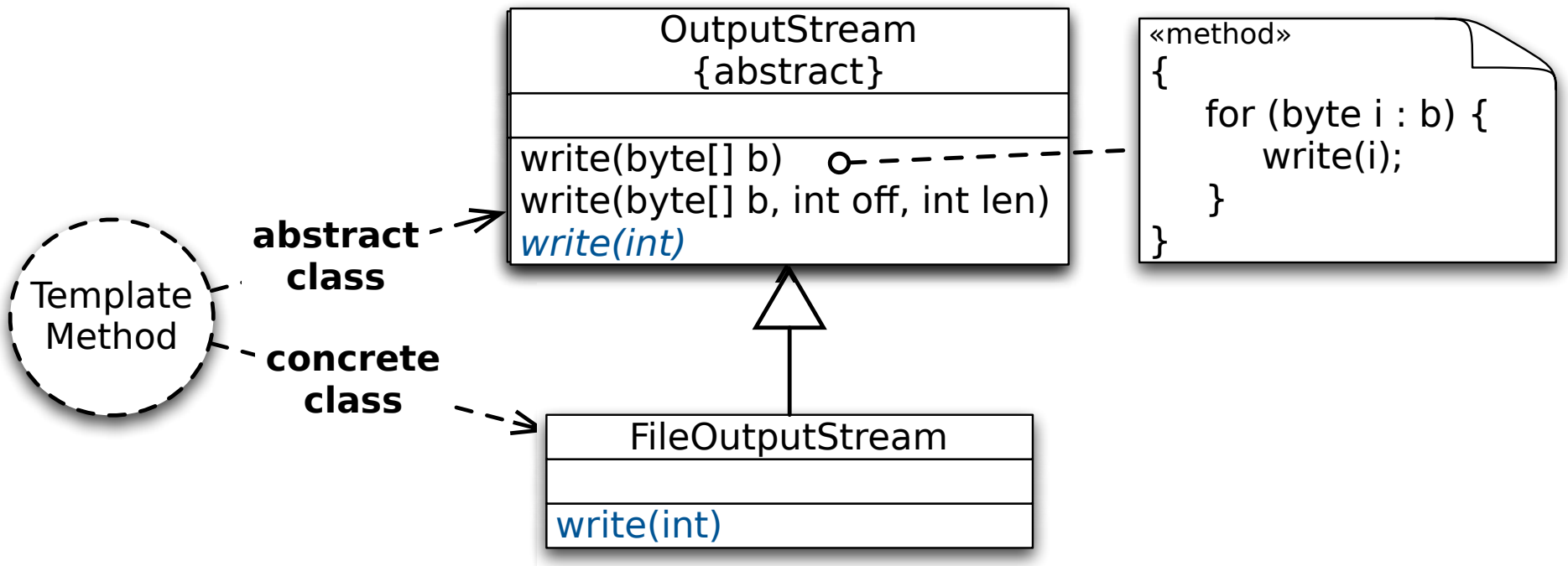
Template Method (325)

Template Method
Overview

- Intent
  - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.
  - Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.



Example: Template Method in the JCL

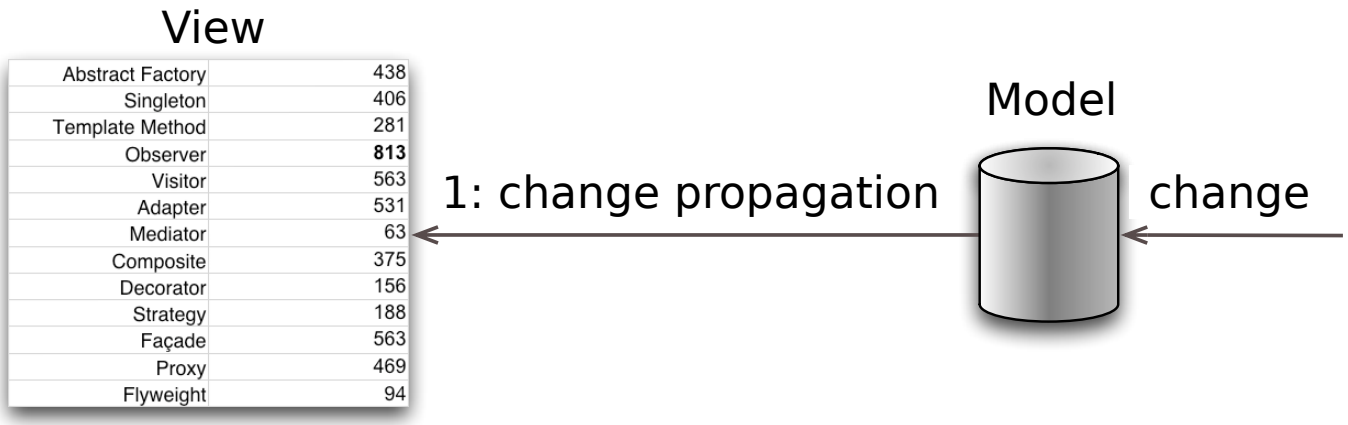


Patterns

Model-View-Controller (MVC)

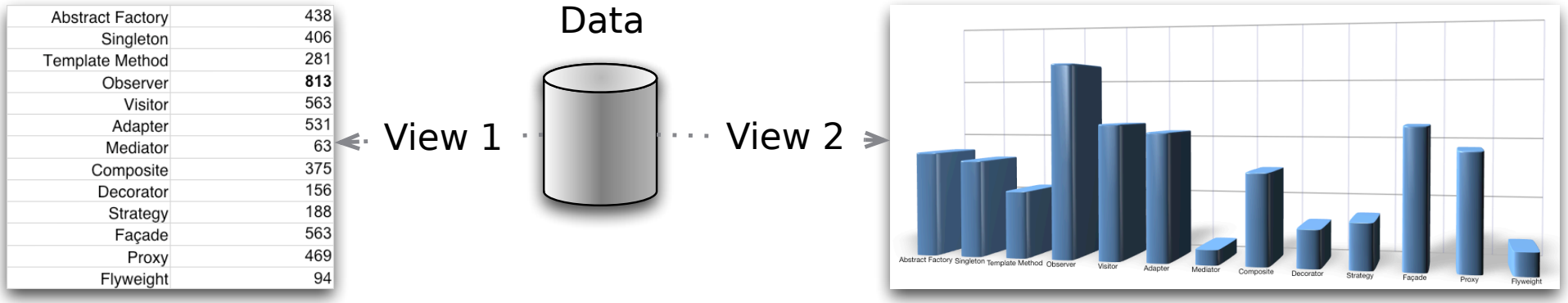
Model-View-Controller
Intent

- Separates the representation of information from the user's interaction with it



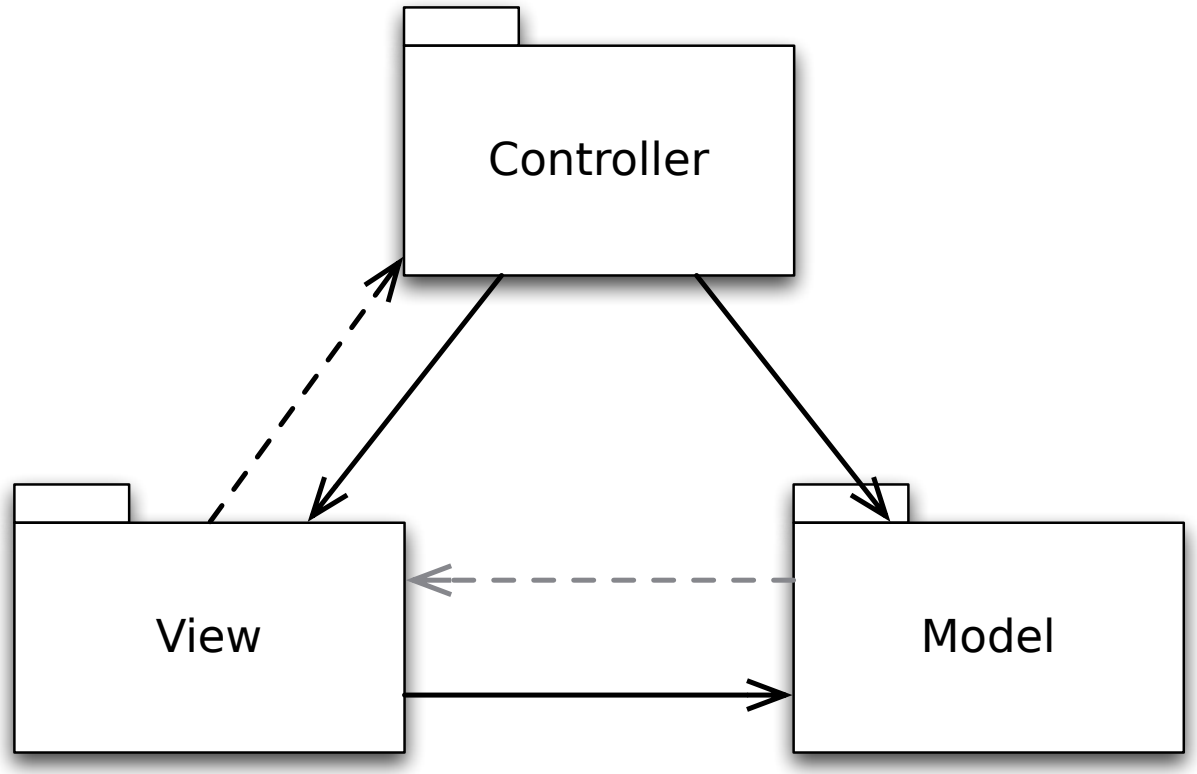
Model-View-Controller
Benefits

- Multiple views on the same data
- Separation of responsibilities allows for flexibility

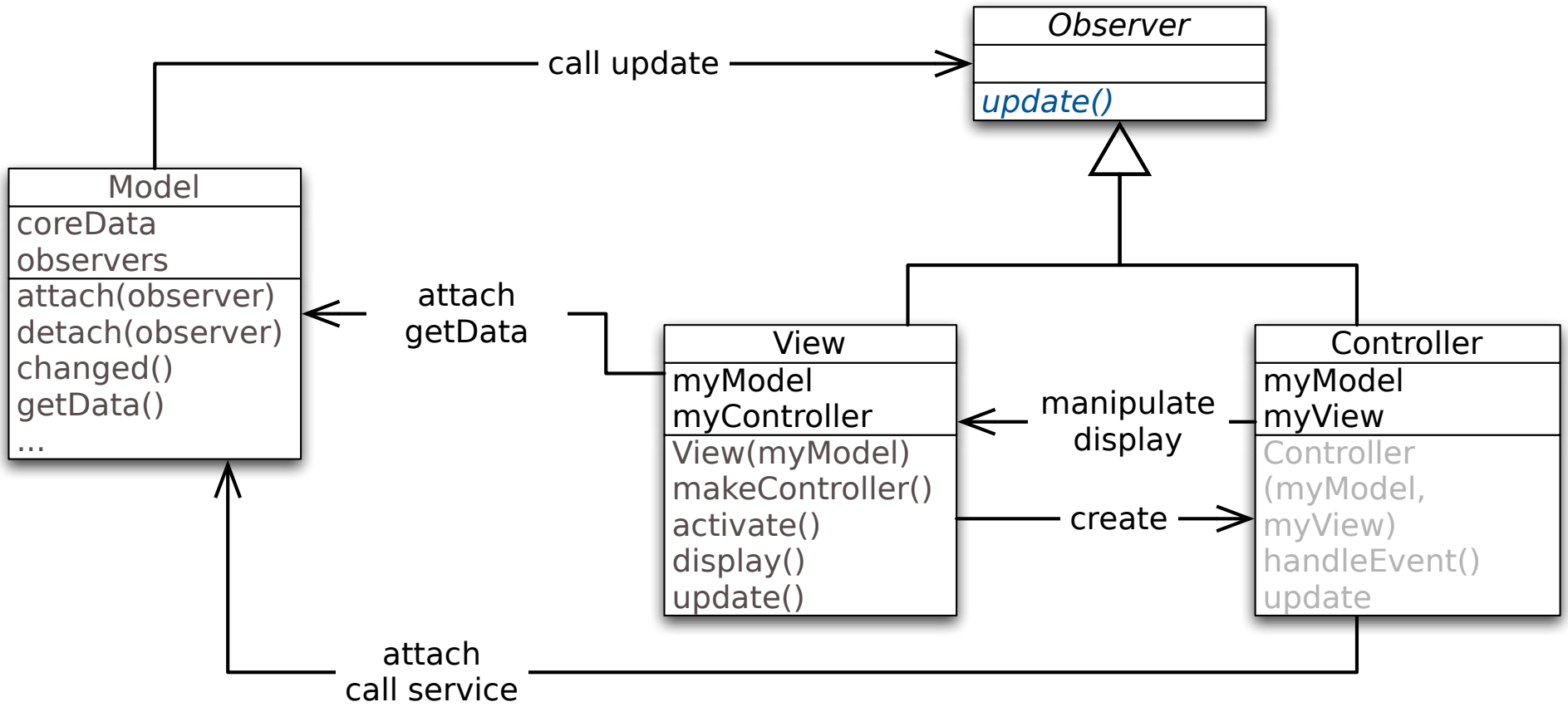


Model-View-Controller

Structure

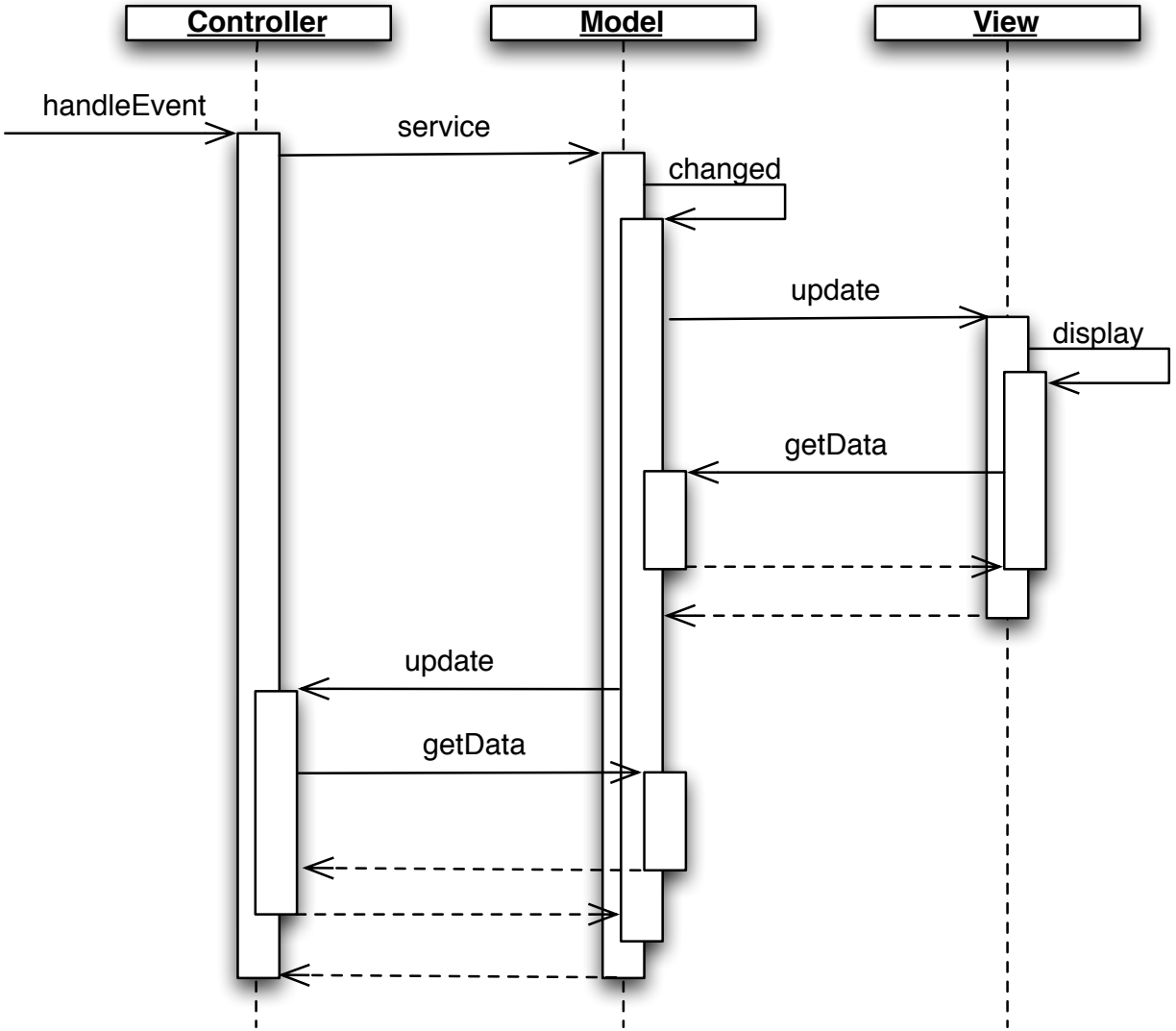


Model-View-Controller
Implementation



Model-View-Controller
Interaction

- Controller handles events and changes the model
- The Model updates the View
- Model triggers update of the Controller
- Which returns control back to the user



Model-View-Controller
Evolution

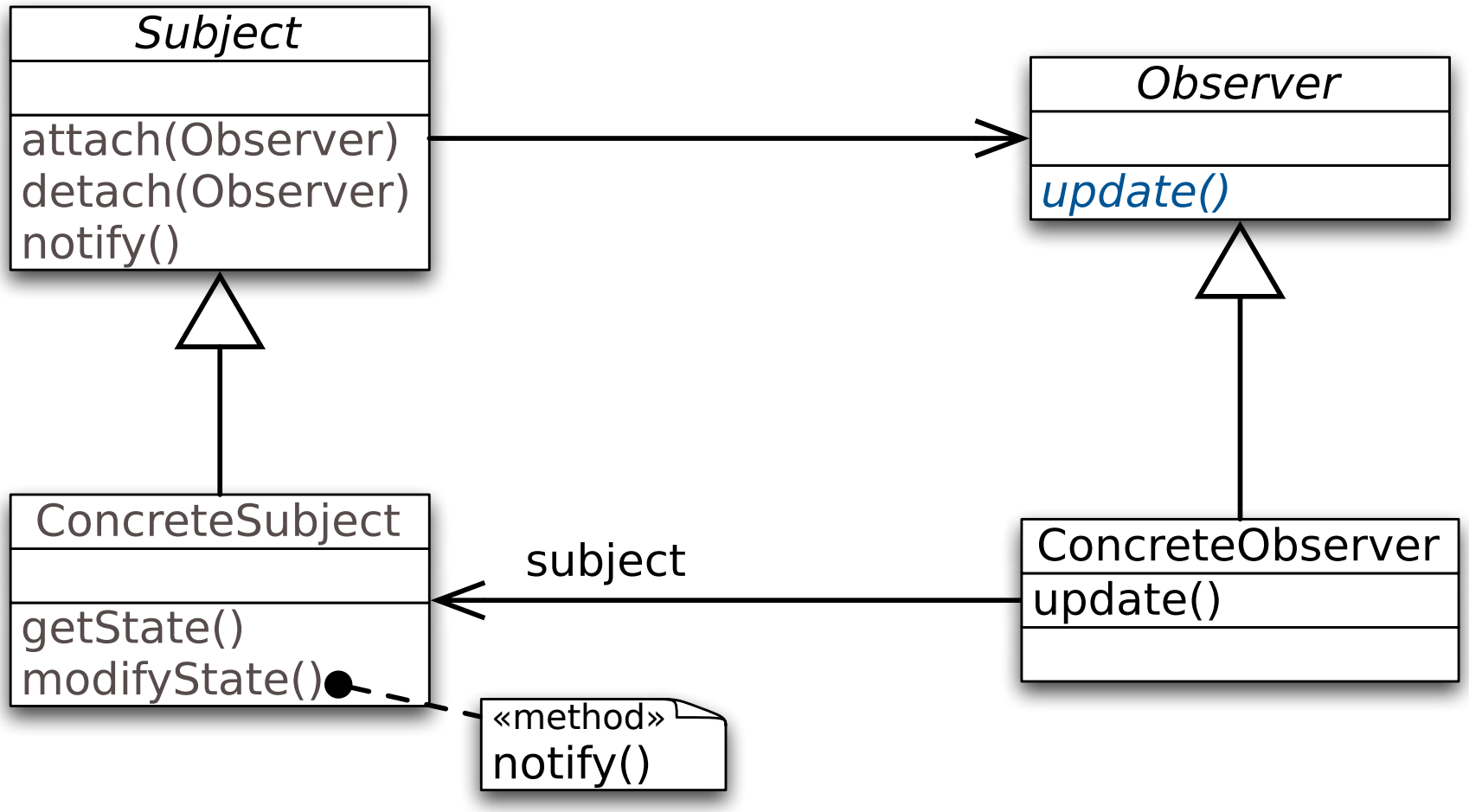
- MVC is quite old
- It might not be entirely suitable for modern architectures
- Newer patterns are:
  - Hierarchical model–view–controller (HMVC)
  - Model-View-Adapter (MVA)
  - Model-View-Presenter (MVP)
  - Model-View-ViewModel (MVVM)

Patterns

Observer (293)

Observer Pattern

Structure

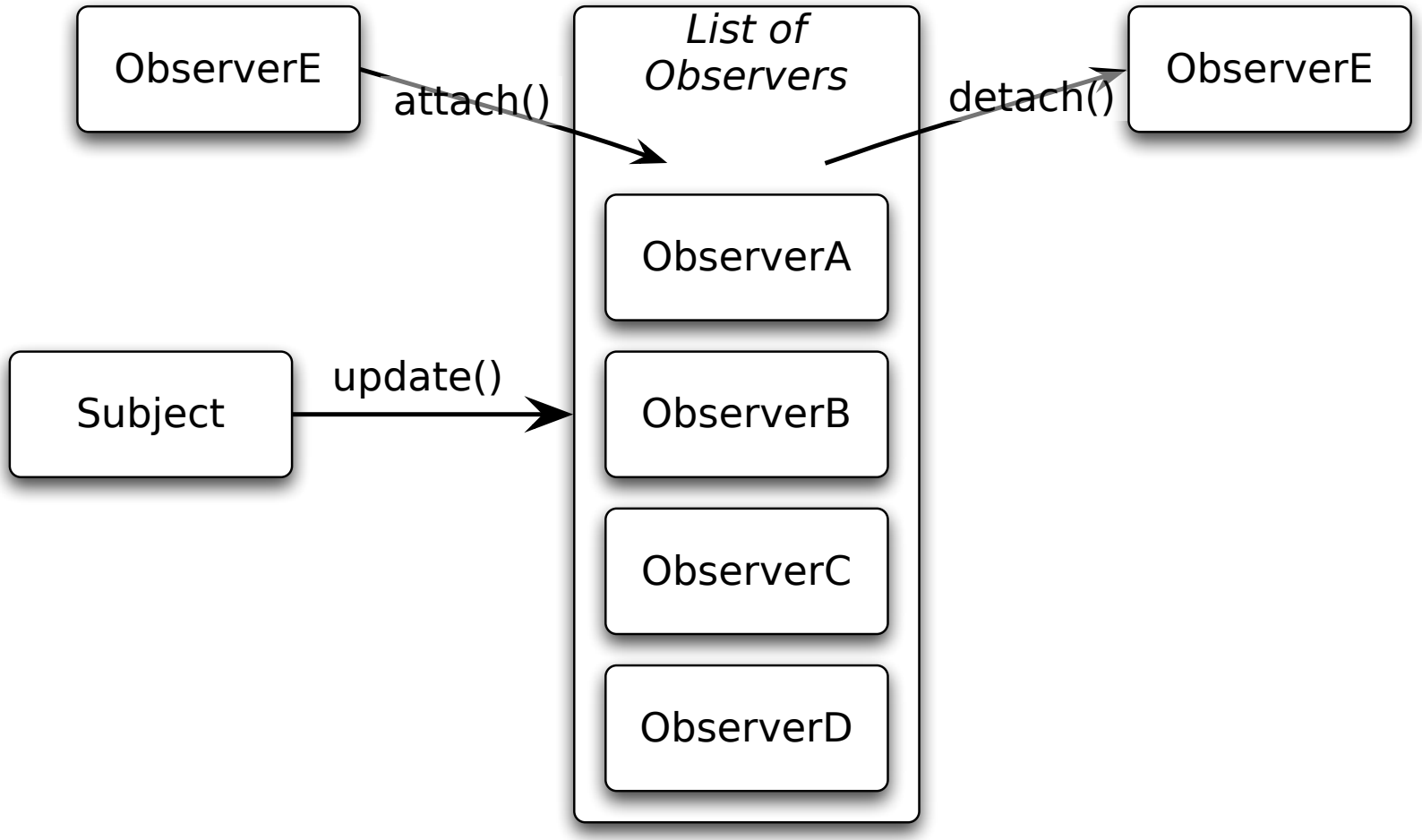


Observer Pattern
Intent

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

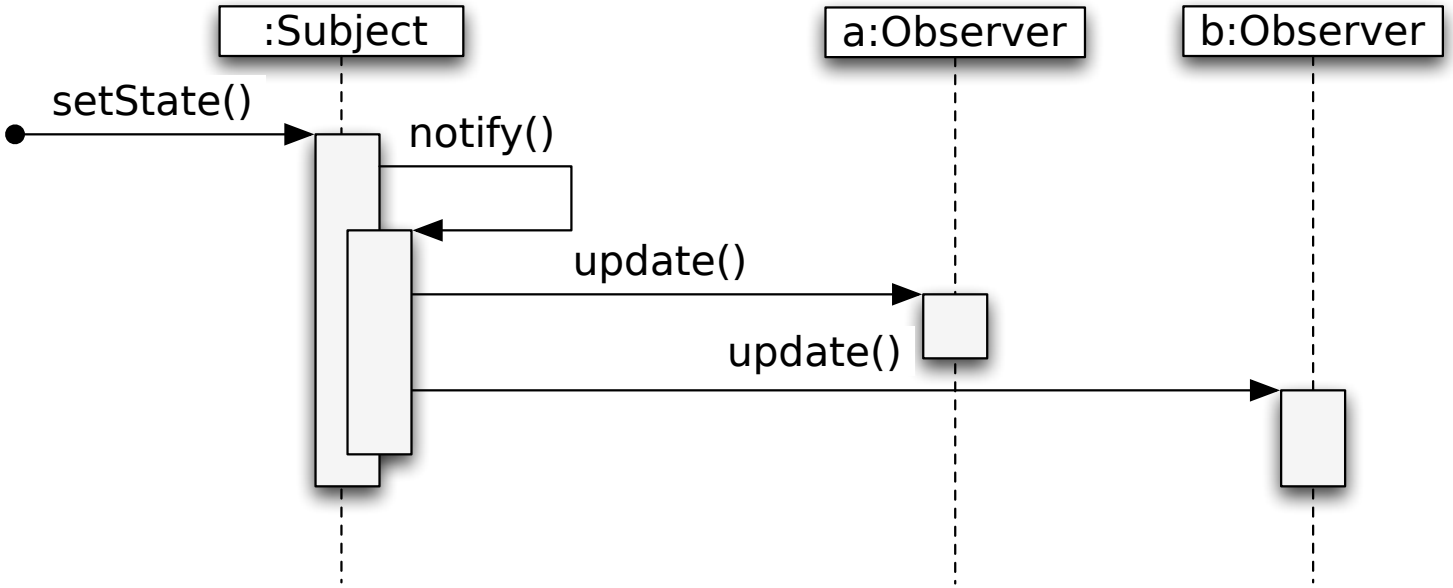
Observer Pattern

Composite Observers



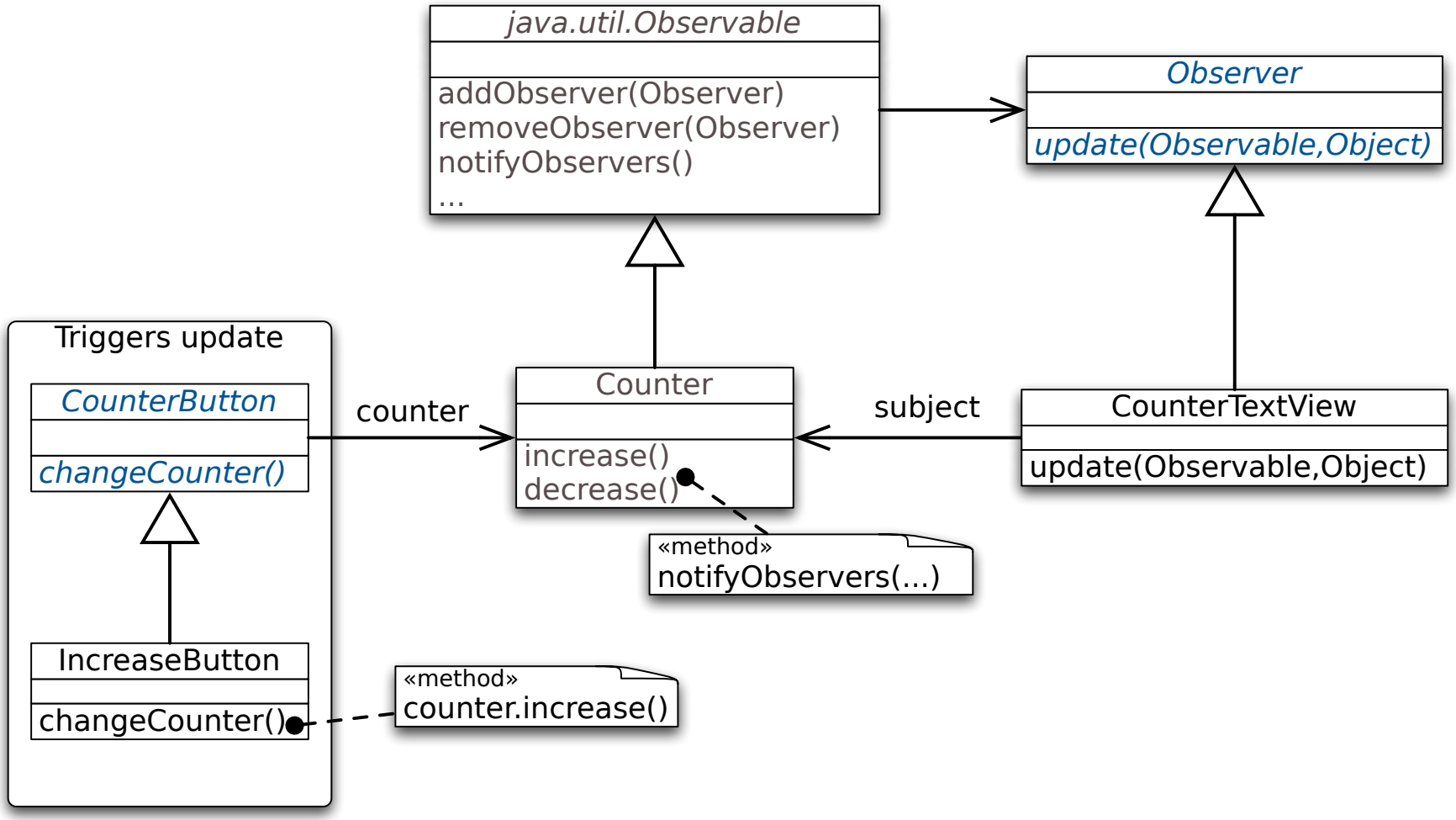
Observer Pattern

Notifying multiple observers



Observer Pattern

Implementing an Observer



Observer Pattern
Implementing an Observer


```
class Counter extends java.util.Observable{
    public static final String INCREASE = "increase";
    public static final String DECREASE = "decrease";
    private int count = 0; private String label;

    public Counter(String label) { this.label= label; }
    public String label() { return label; }
    public int value() { return count; }
    public String toString(){ return String.valueOf(count); }
    public void increase() {
        count++;
        setChanged(); notifyObservers(INCREASE);
    }
    public void decrease() {
        count--;
        setChanged(); notifyObservers(DECREASE);
    }
}
```

Observer Pattern
Implementing an Observer


```
abstract class CounterButton extends Button {

    protected Counter counter;

    public CounterButton(String buttonName, Counter counter) {
        super(buttonName);
        this.counter = counter;
    }

    public boolean action(Event processNow, Object argument) {
        changeCounter();
        return true;
    }

    abstract protected void changeCounter();
}
```

Observer Pattern
Implementing an Observer


```
class IncreaseButton extends CounterButton{
    public IncreaseButton(Counter counter) {
        super("Increase", counter);
    }
    protected void changeCounter() { counter.increase(); }
}
class DecreaseButton extends CounterButton{
    // correspondingly...
}
```

Wrap up

- Design Patterns - Why are we using them?
- Template Method
- Model-View-Controller
- Observer