MSO
MVC & Observer

Hans Philippi

October 19, 2018

This lecture

- The Model-View-Controller pattern
- The Observer pattern

Software architecture – definition

According to Wikipedia:

> *Software architecture describes the high-level structure of a software system*

Software architecture – definition

Software architecture encompasses the set of significant decisions about the organization of a software system including the selection of the structural elements and their interfaces by which the system is composed; behavior as specified in collaboration among those elements; composition of these structural and behavioral elements into larger subsystems; and an architectural style that guides this organization. Software architecture also involves functionality, usability, resilience, performance, reuse, comprehensibility, economic and technology constraints, tradeoffs and aesthetic concerns.

(According to Kruchten, Booch, Bittner, and Reitman)

Software architecture – definition

> *The highest-level breakdown of a system into its parts;* **the decisions that are hard to change***; there are multiple architectures in a system; what is architecturally significant can change over a system's lifetime; and, in the end, architecture boils down to whatever the important stuff is.*

(According to Fowler, emphasis is mine)

Architectural decisions

- How will the users be using the application?
- Where is data stored?
- How will the application be deployed?
- How will it be maintained?

Architectural patterns

Just as we have seen design patterns, there are numerous architectural patterns:

- Component-based
- Data-centric
- Event-driven (or Implicit invocation)
- Layered
- Peer-to-peer
- Pipes and filters
- Plug-ins

Architectural patterns

We will mention one pattern that is more architectural in flavour than many of the design patterns we have seen so far:

Model-View-Controller

- It is very useful for the final lab project
- It helps you understand that patterns appear on different levels during software design
- It might motivate you to study such patterns further
- MVC is actually discussed in the first pages of the Gang of Four book – it's a classic pattern that predates many of the design patterns we've seen so far

Problem: a board and pieces game

Suppose you need to (re)implement a board and pieces game. What do you need to do?

- Logic for handling the moves
- Graphics for the board and pieces
- Interaction with the users

How should you assign these aspects to classes?

Model-View-Controller

The MVC is a triple of classes or components:

- The **Model** is responsible for storing the data
- The **View** is responsible for visualizing data
- The **Controller** is responsible for user interaction

Model-View-Controller on the web

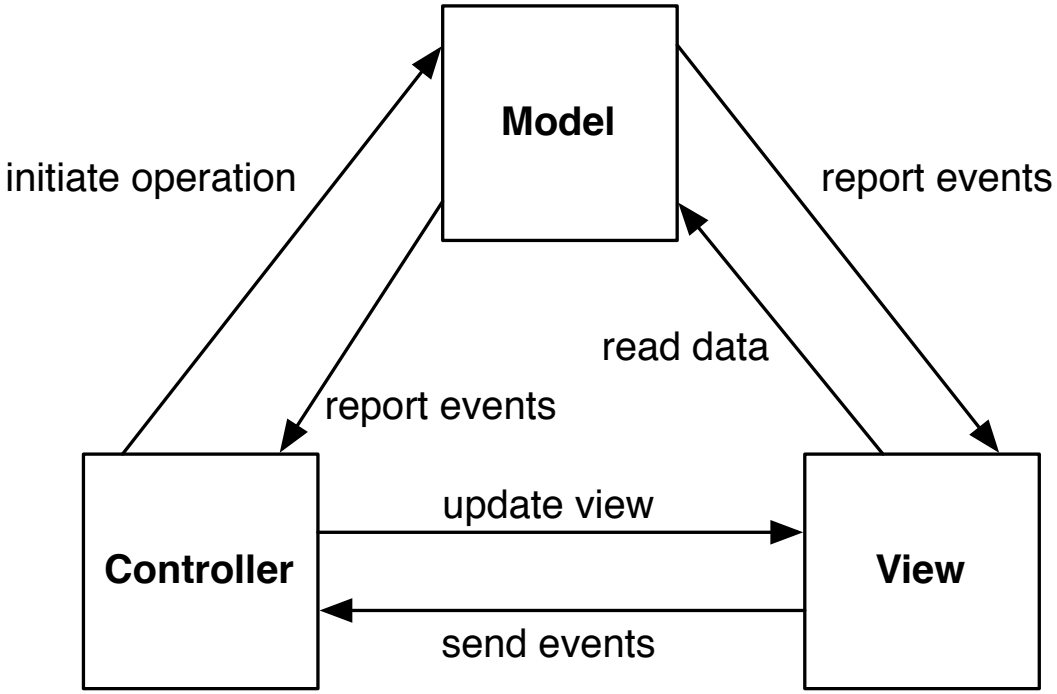
If you are familiar with web programming, you may find the following analogy useful:

- The **Model** is the data in an HTML page
- The **View** is the CSS file describing how it should be displayed
- The **Controller** is your browser that lets you view and navigate a webpage

Model-View-Controller in a database application

- The **Model** is the database: schema and data
- The **View** defines the GUI's for data entry and displaying query results
- The **Controller** defines the control flow and assembles/launches the SQL code

Model-View-Controller



(This is not an UML diagram!)

Why use MVC?

- Strong cohesion – responsibilities are clear
- Decoupled different responsibilities:
  - Allows for multiple views of the same data
  - or completely different controllers
  - or change the way a view responds to the controller
- Makes communication clear
- Most of the 'domain logic' should be in the Model

Exercise: Have a look at some of the code examples from IMP, GP, MOP, such as the calculator, TickTick or the drawing program. Can you recognize MVC structures?

Communication

- How do the Model, View, and Controller communicate?
- They are good candidates for using the *Observer pattern*

Observer pattern: case study

Suppose I get yet another new requirement for my online webstore. Whenever a new customer registers, we should:

- send a welcome e-mail to the customer
- verify the customer's address with the post office
- update our data base
- ...

Observer pattern: case study

We could hard-code this:

```
class Customer
{
    public void addCustomer()
    {
        emailServer.WelcomeCustomer();
        postOffice.VerifyAddress();
        database.AddCustomer();
    }
}
```

Starting to lose cohesion . . . , but it can get worse

Observer pattern: games

Alternatively, what if you're writing a simple mobile game with a bunch of *achievements* that can be unlocked:

- when you defeat 100 enemies
- when you die 10 times
- when you have scored 1 million points
- ...

Who checks when you earn an achievement?

- You want to keep the achievement checks separated from the main game code
- But when certain events happen (enemy defeated, death, etc.) you may want to *inform* another class to check whether a new badge has been earned
- How can you set up this communication effectively?

Observer pattern: intent

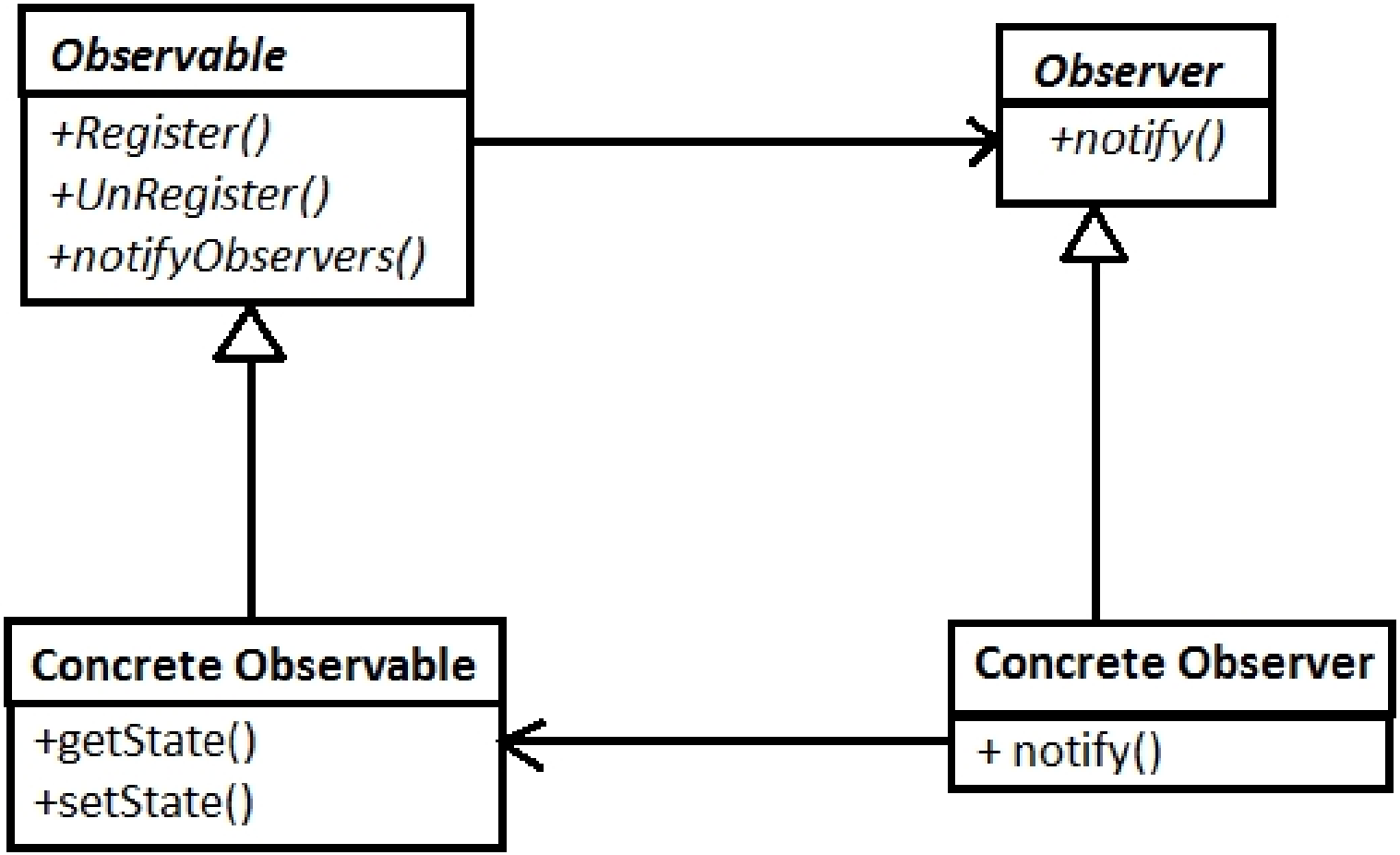
> *To define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated.*

It's also known as the *Publish-Subscribe* model or *Dependents*

Terminology issues: book vs. C# IObserver

- The terms `update()`, `attach()`, `detach()` are used in the book, where the authors define their own interface
- `update()` corresponds to `Notify()` here
- `attach()` corresponds to `Register()` here
- `detach()` corresponds to `UnRegister()` here
- The equivalent terms `Notify()`, `Register()`, `UnRegister()` match the C#-interface
- Some other sources use `Notify()` for `NotifyObservers()`

General Observer pattern



Step 1: Establish a common interface

- In order to treat all the interested objects, or Observers, we need to establish a common interface
- We prefer using interfaces above subclasses
- C# has a special IObserver interface for precisely this purpose:

```
public interface IObserver
{
   void Notify(object anObject);
}
```

- In our example, the email server and post office would need to define Notify methods

Step 1: Establish a common interface

```
public interface IObserver
{
   void Notify(object anObject);
}

...

class PostOffice : IObserver
...
void Notify (Customer myCustomer)
// do the required stuff here
```

- In our example, the email server and post office would need to define Notify methods

Step 2: Register the observers

- Let anyone interested in knowing about new customers register themselves
- To do that, we add two methods to the Customer class, implementing this interface:

```
public interface IObservable
{
  void Register(IObserver anObserver);
  void UnRegister(IObserver anObserver);
}
```

- Now anyone interested in new customers can Register

Step 3: Notification

Now when a new customer is added, os the status of the customer has been changed, we can notify any observers interested in that event:

```
void notifyObservers ()
{
    foreach(IObserver anObserver in myObservers)
    {
        anObserver.Notify(anObject);
    }
}
```

Step 4: Exposing further information

- Perhaps the observers need further information about the Customers: `getState()`
- You may want to revisit your design and make sure they have access to the necessary information, like addresses

Pitfalls

The Observer pattern is particularly useful when the list of Observers can vary:

- not all requirements are known yet
- the observers change dynamically
- ...

Material covered

- Design Patterns explained: chapter 18
- Model-View-Controller: https://en.wikipedia.org/wiki/Model-view-controller