13 The MVC model

Main concepts to be covered

- Design patterns
- The Observer design pattern
- The Model View Controller architecture

Using design patterns

- Inter-class relationships are important, and can be complex.
- Some relationship recur in different applications.
- Design patterns help clarify relationships, and promote reuse.

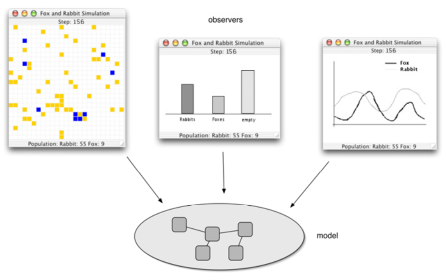
Pattern structure

- A pattern name.
- The problem addressed by it.
- How it provides a solution:
  - Structures, participants, collaborations.
- Its consequences.
  - Results, trade-offs.

Design pattern: Observer

- Supports separation of internal model from a view of that model.
- Observer defines a one-to-many relationship between objects
  - *publisher - subscriber*
- The object-observed notifies all Observers of any state change.
- Example `SimulatorView` in the *foxes-and-rabbits project*.

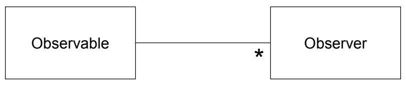
Observers


Fox and Rabbit Simulation
Step: 156
Population: Rabbit: 55 Fox: 9
observers
Fox and Rabbit Simulation
Step: 156
Rabbits
Foxes
empty
Population: Rabbit: 55 Fox: 9
Fox and Rabbit Simulation
Step: 156
Fox
Rabbit
Population: Rabbit: 55 Fox: 9
model

Main classes of interest

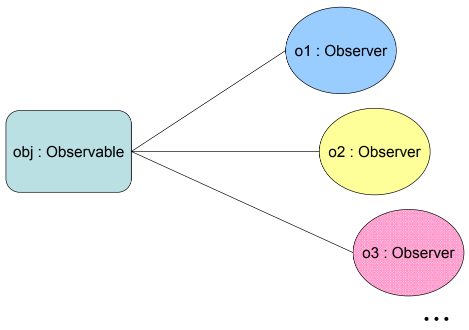
- `class Observable`
  - Subclasses inherit basic functionality for reporting state changes to observing objects.
  - Independent of the observer’s logic
- `interface Observer`
  - Subclasses implement update funtionality.
  - Many objects can connect to the same observable object.

Class relationships



- No particular owner-owned relationship
  - Observers do not own the observed objects.
  - Observed objects are unaware of observers.
  - The relation is *navigable* in both directions
    - *Observers know what they observe.*
    - *Observables must be able to update observers (weak dependency).*

Typical configuration



`class Observable`

```
public class Observable {

- Add observer o to the set of observers for this object
    public void addObserver(Observer o)

- Mark this object as changed
    public void setChanged()

- If this object has changed, then notify all of it’s observers
    public void notifyObservers()

}
```

Typical Observable class

```
public class Obsrvbl extends Observable {
    private SomeType x;

    public void someMutator() {
       ...
       x = ...; // x has changed, inform observers
       setChanged();
       notifyObservers(x.clone());
       ...
    }
}
```

Pass some information to the observers.
Maybe a copy of x, or something else.

`interface Observer`

```
public interface Observer {

An observable object calls it’s inherited notifyObservers method
to have all the object's observers notified of a state change.
notifyObservers then calls update for each observer.

    Parameters:
        o - the observable object who initiated the call.
        arg - the argument that was passed to the
              notifyObservers method by the observable object.
              notifyObservers forwards this argument to update.

    void update(Observable o, Object arg);
}
```

Typical Observer class

```
public class Obsrvr implements Observer {
    ...
    public void update(Observable o,Object arg) {
        if ( o instanceof Obsrvbl &&
             arg instanceof SomeType) {
            SomeType x = (SomeType)arg;
            // take some appropriate action
            // based on the value of x
        } else
            ...
    }
}
```

Several objects of different types may be observed by the same observer. Moreover, each observed object may, depending on the situation, pass arguments of different types to update. Hence a case analysis may be necessary.

Typical setup

```
Observable obj = new Obsrvbl();

Observer o1 = new Obsrvr();
Observer o2 = new Obsrvr();
Observer o3 = new Obsrvr();

obj.addObserver(o1);
obj.addObserver(o2);
obj.addObserver(o3);
```

Observer registration

Alternative Observer class

```
public class Obsrvr implements Observer {

    public Obs(Observable x) {
       ...
       x.addObserver(this);
       ...
    }

    public void update(Observable o,Object arg)  {
        ...
    }
}
```

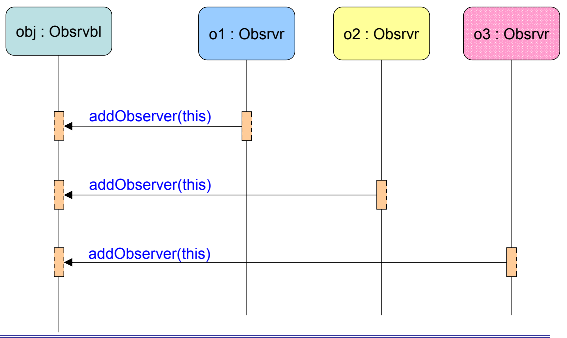
Observer registration

Alternative setup

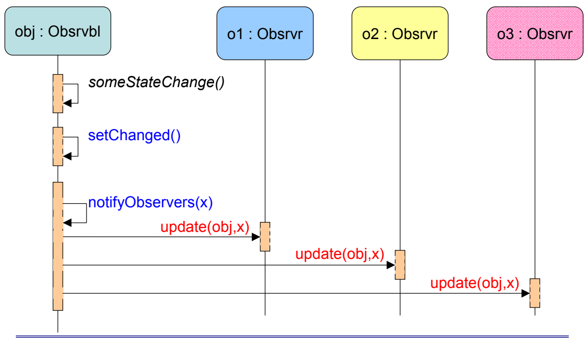
```
Observable obj = new Obsrvbl();

Observer o1 = new Obsrvr(obj);
Observer o2 = new Obsrvr(obj);
Observer o3 = new Obsrvr(obj);
```

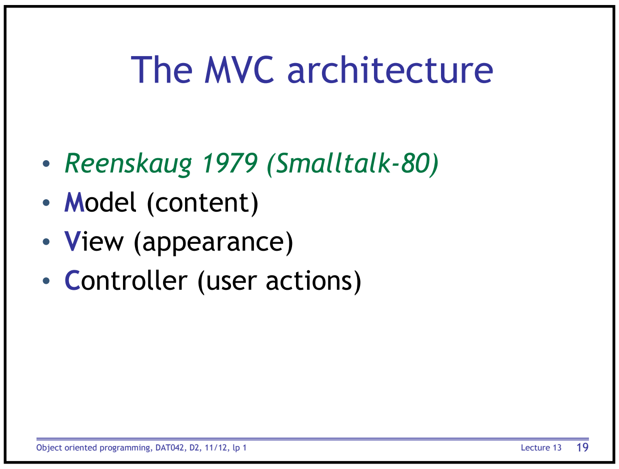
Observer registration

A setup scenario



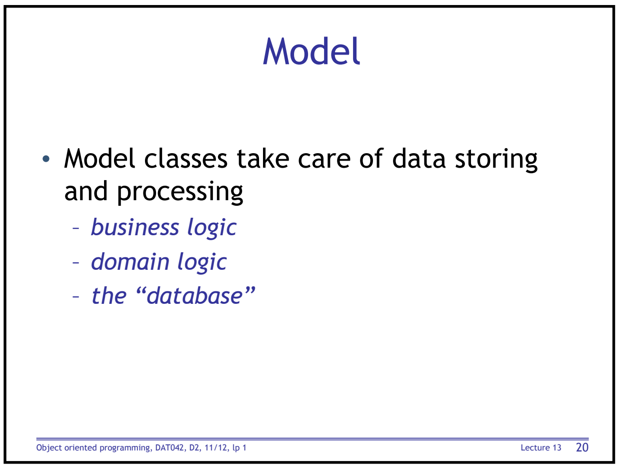
An update scenario





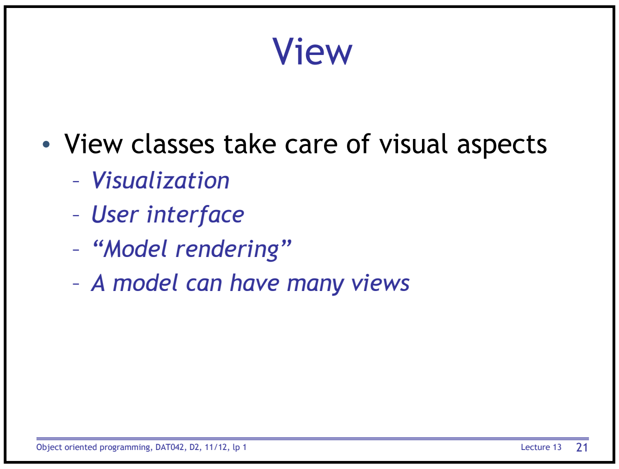
The MVC architecture

- *Reenskaug 1979 (Smalltalk-80)*
- Model (content)
- View (appearance)
- Controller (user actions)



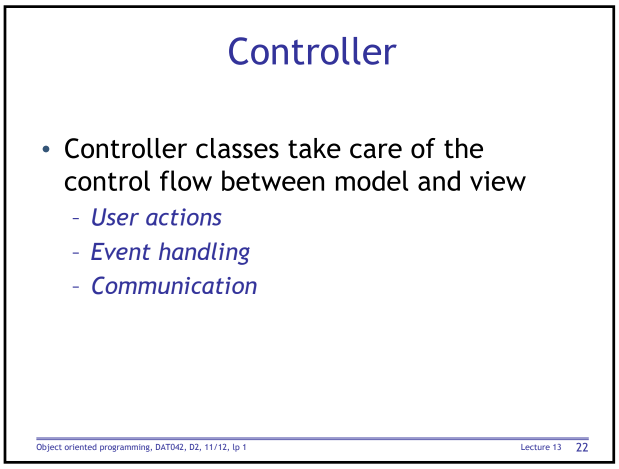
Model

- Model classes take care of data storing and processing
  - *business logic*
  - *domain logic*
  - *the "database"*



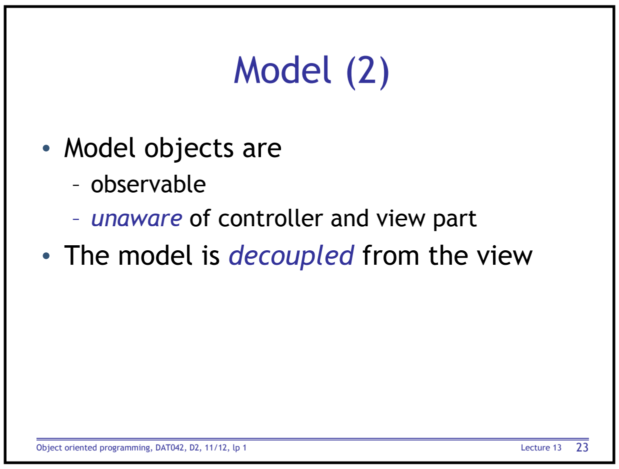
View

- View classes take care of visual aspects
  - *Visualization*
  - *User interface*
  - *"Model rendering"*
  - *A model can have many views*



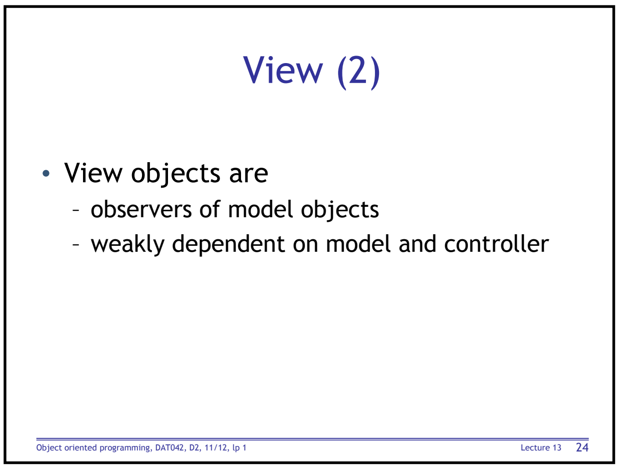
Controller

- Controller classes take care of the control flow between model and view
  - *User actions*
  - *Event handling*
  - *Communication*



Model (2)

- Model objects are
  - observable
  - *unaware* of controller and view part
- The model is *decoupled* from the view



View (2)

- View objects are
  - observers of model objects
  - weakly dependent on model and controller

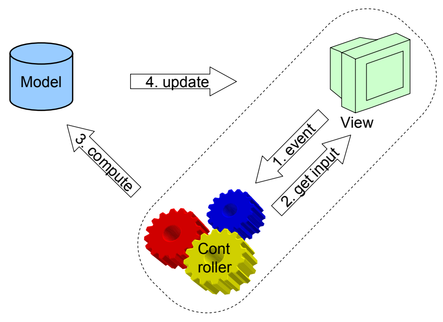
Controller (2)

- Controller objects
  - update the model with information from the view
- Example: Action control objects in a GUI

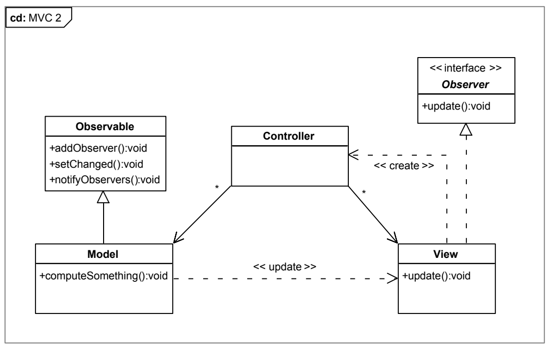
Variations

- Variations of the MVC pattern are possible.
- More or less coupling between model, view and controller:
  - View observes model directly.
  - or: Controller mediates all communication between model and view.

MVC architecture



MVC class diagram

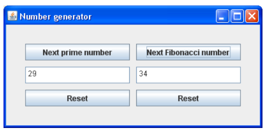


Consequences

+ Model is completely independent of view.
- View is more or less dependent on model
  - the view must often have some *domain knowledge*. Eg. Syntax checking in forms.
- Controllers are dependent of both model and view.

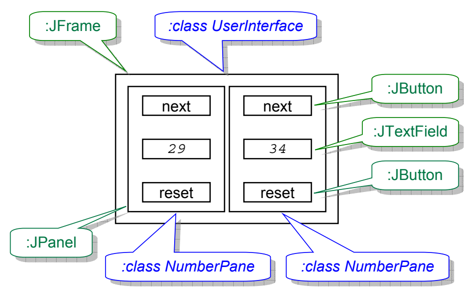
Example: Number series calculator

- A (very) simple calculator for exploring the prime number and Fibonacci number series.

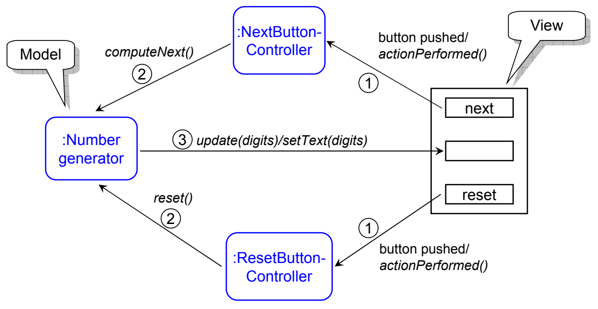


- Program design based on the MVC pattern.
- Explore the *mvc1* project!

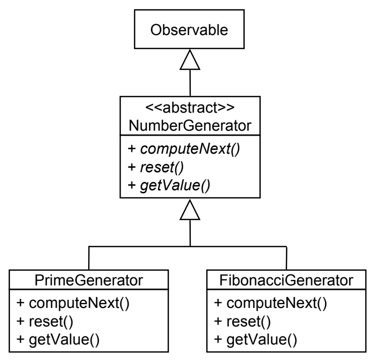
The calculator GUI



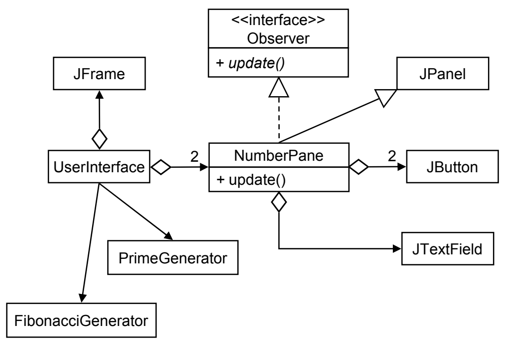
Control flow



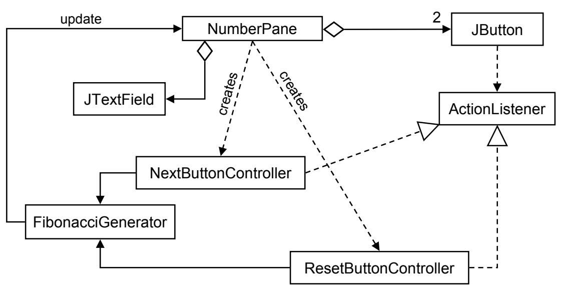
Class design



Class design (cont.)



Class design (cont.)



Review

- The degree of dependency between components is called *coupling*.
- Aim for less coupling!
- The *observer* design pattern decreases coupling.
- The MVC architectural pattern decouples the business logic from GUI issues
  - *thus easy to modify or replace GUI!*