# Gang of Four Design Patterns Part 2

**M19COM Software Development & Design**

**Week 9**

**Dr Faiyaz Doctor**

# More Gang-of-Four (GoF) Design Patterns

Reminder

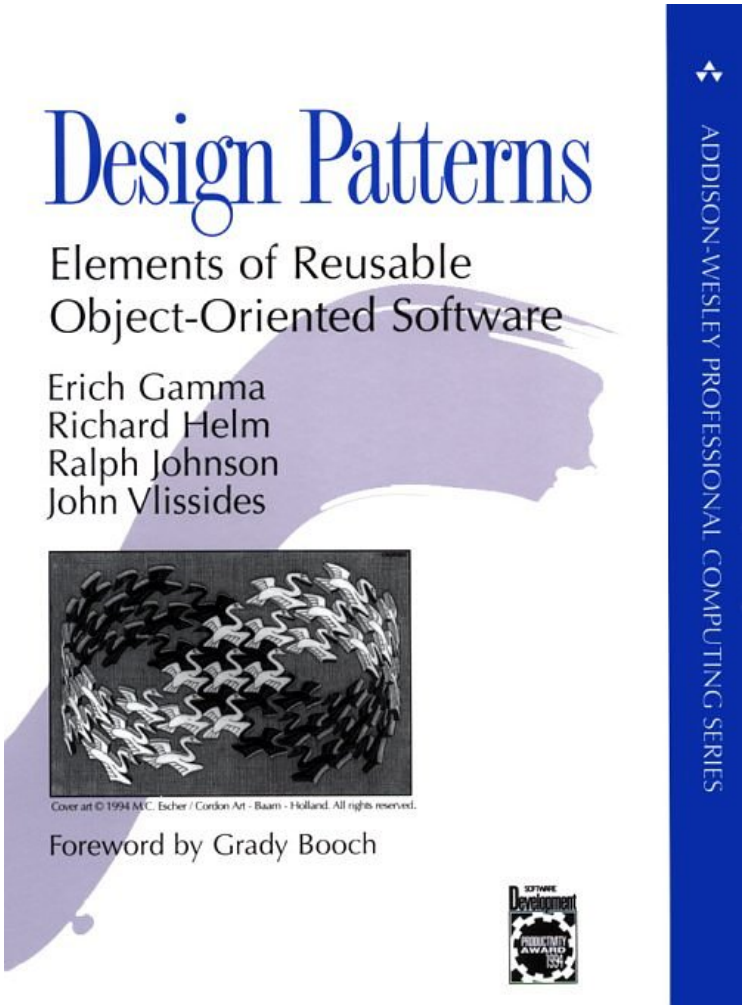
- There are 23 patterns
- 10-15 are widely used
- We are looking at just a few:

**In previous session we covered**

- Adapter ✓
- Singleton ✓
- Factory ✓
- Strategy ✓

**In this session we will cover:**

- Observer
- MVC



# THE OBSERVER PATTERN

- Problem:

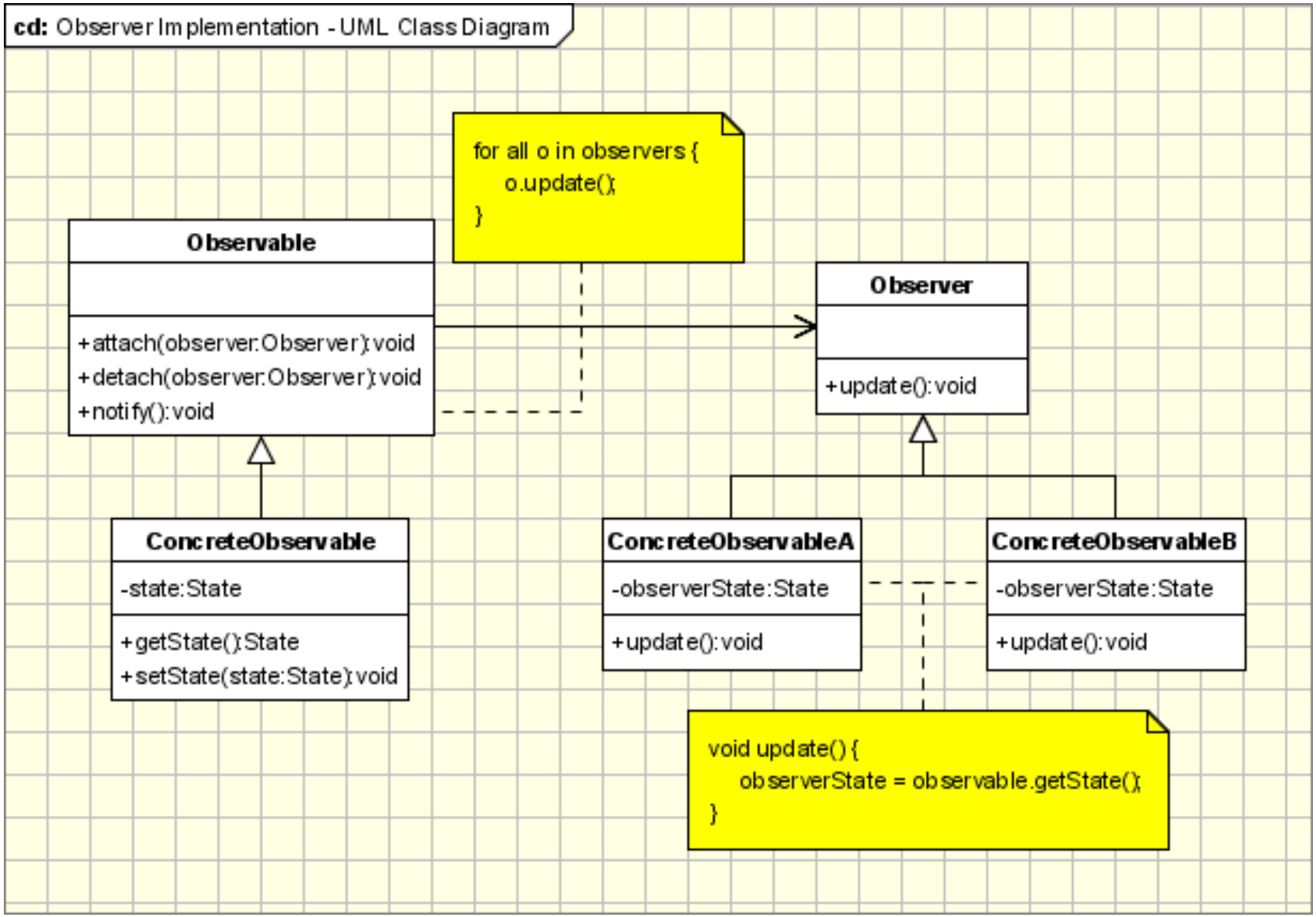
  Different kinds of observer objects are interested in the state changes or events of a publisher or subject object, and want to react in their own way when the publisher/subject generates an event. Moreover, the publisher wants to maintain **low coupling** to the observers. What to do?

- Solution:

  Define an “observer” interface. Observers need to implement this interface. The publisher/subject can dynamically register observers who are interested in an event and notify them when an event occurs.

# The Observer Pattern

- Let's assume we have a stock system which provides data for several types of client. We want to have a client implemented as a web based application but in near future we need to add clients for mobile devices, Palm or Pocket PC, or to have a system to notify the users with sms alerts. Now it's simple to see what we need from the observer pattern: we need to separate the subject(stocks server) from it's observers(client applications) in such a way that adding new observer will be transparent for the server.


cd: Observer Implementation - UML Class Diagram
for all o in observers {
o.update();
}
Observable
+attach(observer:Observer):void
+detach(observer:Observer):void
+notify():void
Observer
+update():void
ConcreteObservable
-state:State
+getState():State
+setState(state:State):void
ConcreteObservableA
-observerState:State
+update():void
ConcreteObservableB
-observerState:State
+update():void
void update() {
observerState = observable.getState();
}

The participants classes in this pattern are:

**Observable** - interface or abstract class defining the operations for attaching and de-attaching observers to the client. In the GOF book this class/interface is known as **Subject**.

**ConcreteObservable** - concrete Observable class. It maintain the state of the object and when a change in the state occurs it notifies the attached **Observers**.

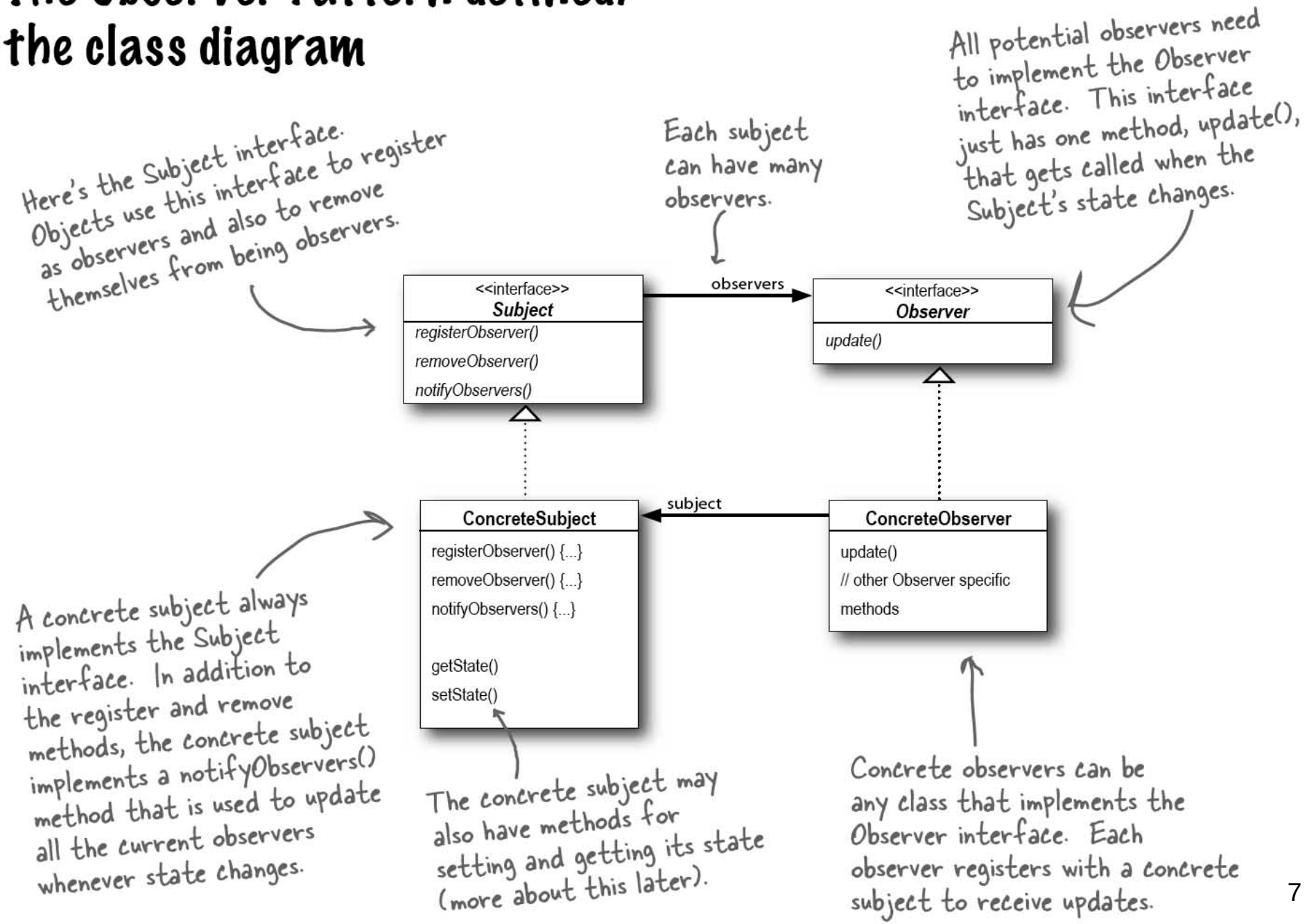
**Observer** - interface or abstract class defining the operations to be used to notify this object.

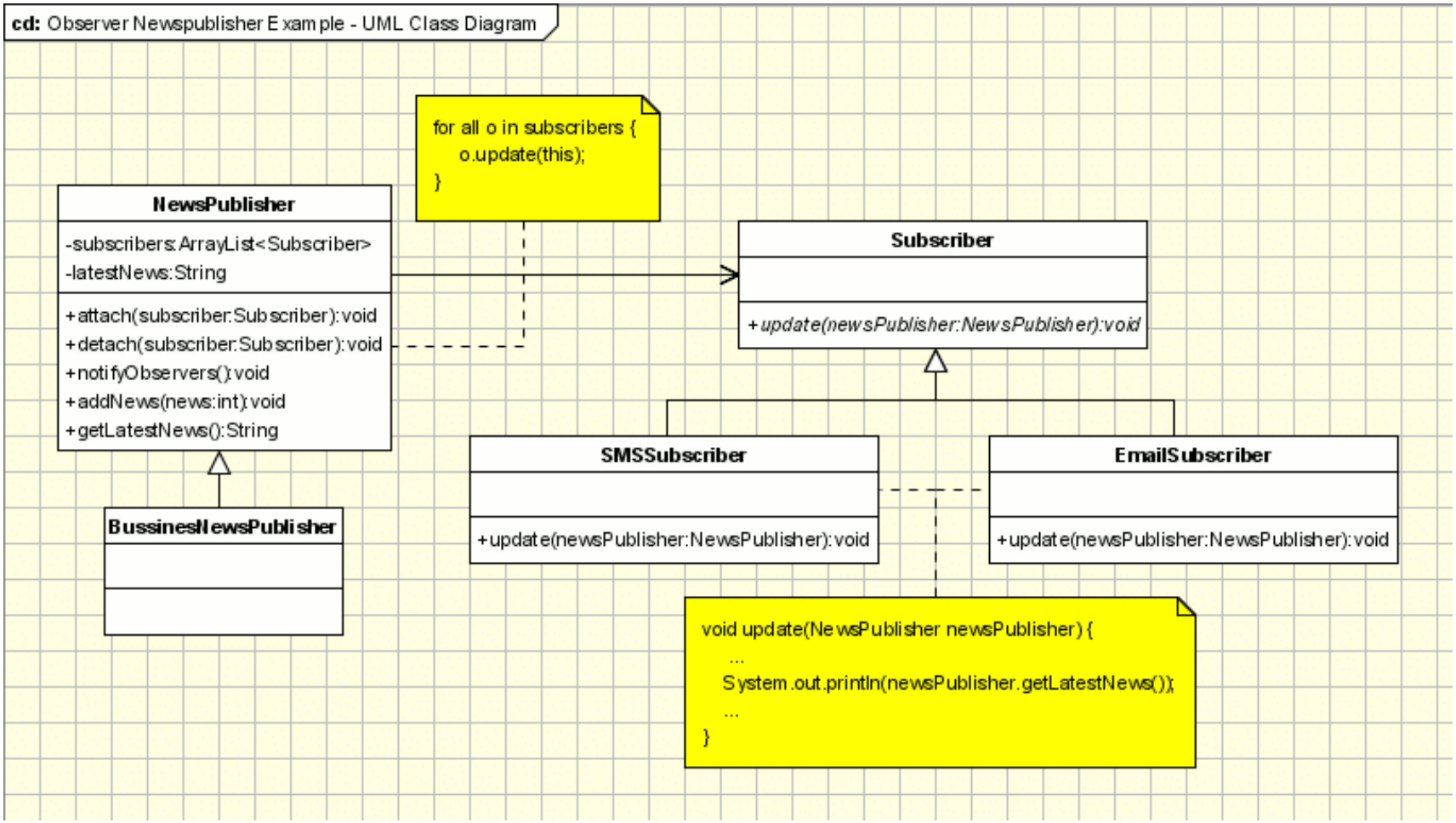
**ConcreteObserverA, ConcreteObserver2** - concrete **Observer** implementations.

The flow is simple: the main framework instantiate the ConcreteObservable object. Then it instantiate and attaches the concrete observers to it using the methods defined in the Observable interface. Each time the state of the subject it's changing it notifies all the attached Observers using the methods defined in the Observer interface. When a new Observer is added to the application, all we need to do is to instantiate it in the main framework and to add attach it to the Observable object. The classes already created will remain unchanged.

# The Observer Pattern defined: the class diagram

Head first design patterns, Eric Freeman et al




cd: Observer Newspublisher Example - UML Class Diagram
for all o in subscribers {
o.update(this);
}
NewsPublisher
-subscribers:ArrayList<Subscriber>
-latestNews:String
+attach(subscriber:Subscriber):void
+detach(subscriber:Subscriber):void
+notifyObservers():void
+addNews(news:int):void
+getLatestNews():String
Subscriber
+update(newsPublisher:NewsPublisher):void
BussinesNewsPublisher
SMSSubscriber
+update(newsPublisher:NewsPublisher):void
EmailSubscriber
+update(newsPublisher:NewsPublisher):void
void update(NewsPublisher newsPublisher) {
...
System.out.println(newsPublisher.getLatestNews());
...
}

The agency is represented by an Observable(Subject) class named **NewsPublisher**.

This one is created as an abstract class because the agency want to create several types of Observable objects: in the beginning only for business news, but after some time sport and political new will be published.

The concrete class is **BusinessNewsPublisher**.

The observer logic is implemented in **NewsPublisher**. It keeps a list of all it subscribers and it informs them about the latest news.

The subscribers are represented by some observers (SMSSubscriber, EmailSubscriber). Both the observers mentioned above are inherited from the Subscriber.

The subscriber is the abstract class which is known to the publisher.

The publisher doesn't know about concrete observers, it knows only about their abstraction.

In the main class a publisher(Observable) is built and a few subscribers(Observers). The subscribers are subscribed to the publisher and they can be unsubscribed.

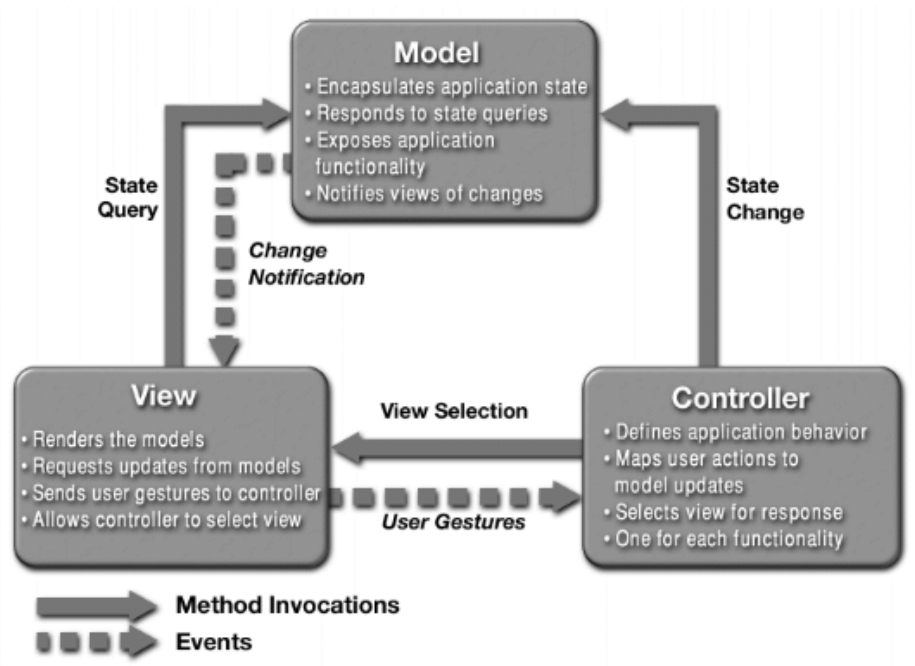
In this architecture new types of subscribers can be easily added(instant messaging, ...) and new types of publishers(Weather News, Sport News, ...).

# The Model-View-Controller (MVC) Pattern

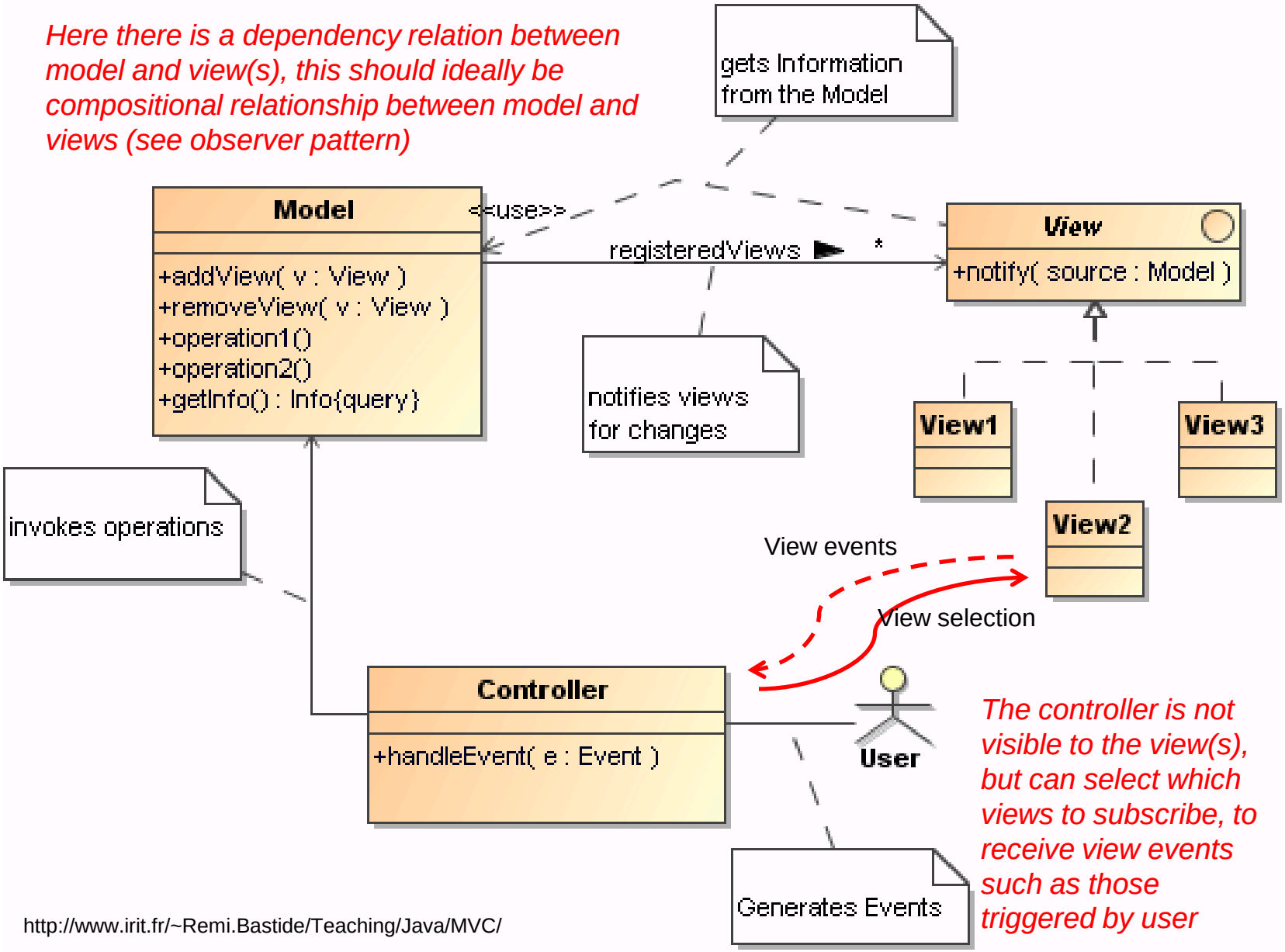
- In the Observer pattern it was implied that events would generally be *input* events, e.g. mouse-clicks.
- However in a GUI we often have “views” (external representations) of the “model” (domain data maintained by the system).
- We may have *external or internal* events in response to which a “controller” class will make changes to the model data.
- All relevant views then need to be updated. This is the situation addressed by the MVC pattern.
- The following diagram is from the Sun J2EE website:

# The Model-View-Controller (MVC) Pattern

- The dotted arrows are *event notifications*, using the Observer pattern. That is,
  - the Controller subscribes to View events (e.g. mouse-clicks, keystrokes)
  - the View similarly subscribes to Model change events. *There may be multiple Views (Observers) running.*
- A View event is notified to the Controller, so that it can update the Model (e.g. setting pageOrient = landscape).
- The resulting model change event is notified to the View, so that it can update its display (the radio-button selection).
- *Wrong for the View to update itself directly, as the user request might not be acceptable to the system.*

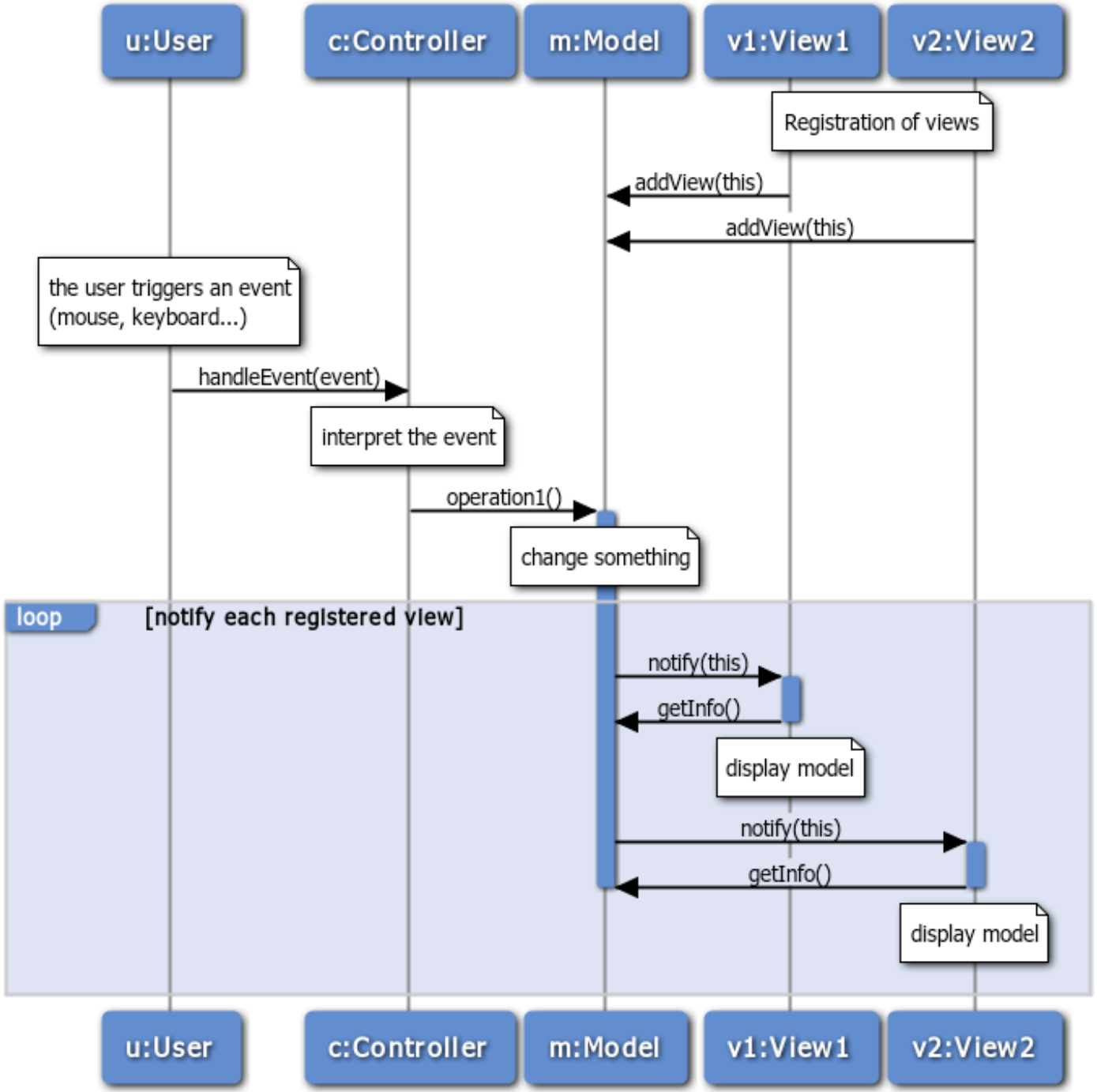


*Here there is a dependency relation between model and view(s), this should ideally be compositional relationship between model and views (see observer pattern)*



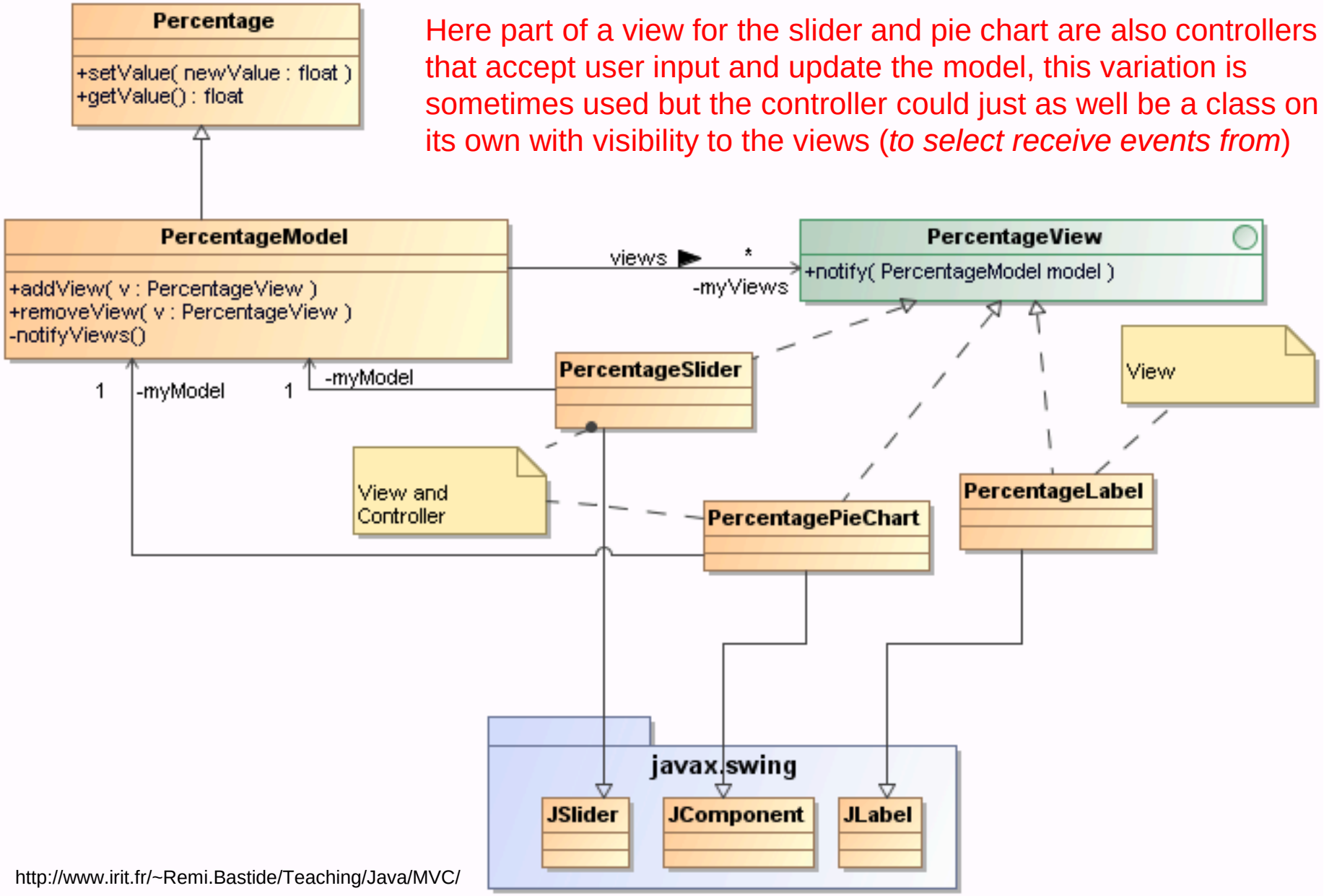

*The controller is not visible to the view(s), but can select which views to subscribe, to receive view events such as those triggered by user*

http://www.irit.fr/~Remi.Bastide/Teaching/Java/MVC/



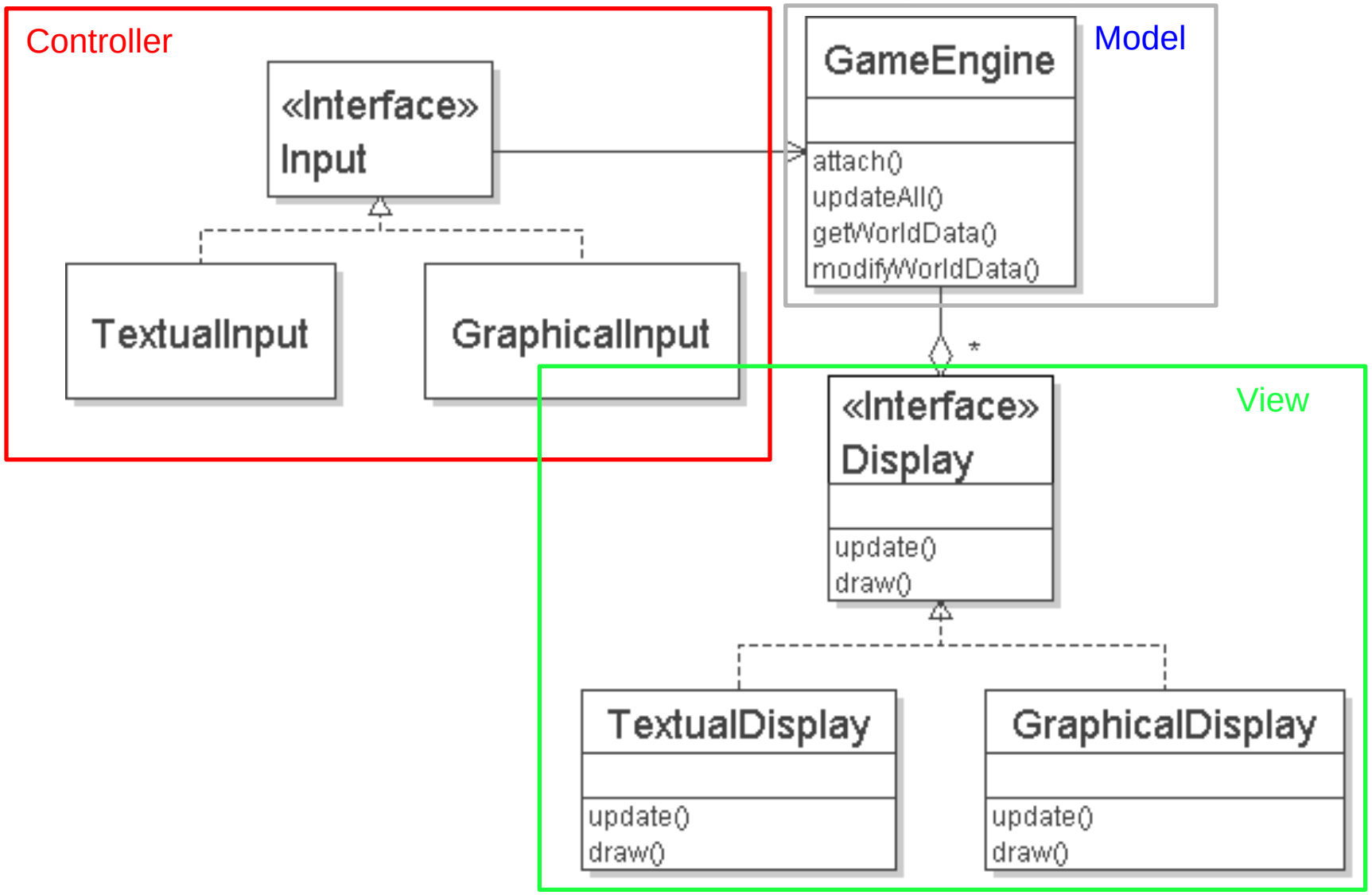

www.websequencediagrams.com

Here part of a view for the slider and pie chart are also controllers that accept user input and update the model, this variation is sometimes used but the controller could just as well be a class on its own with visibility to the views (*to select receive events from*)

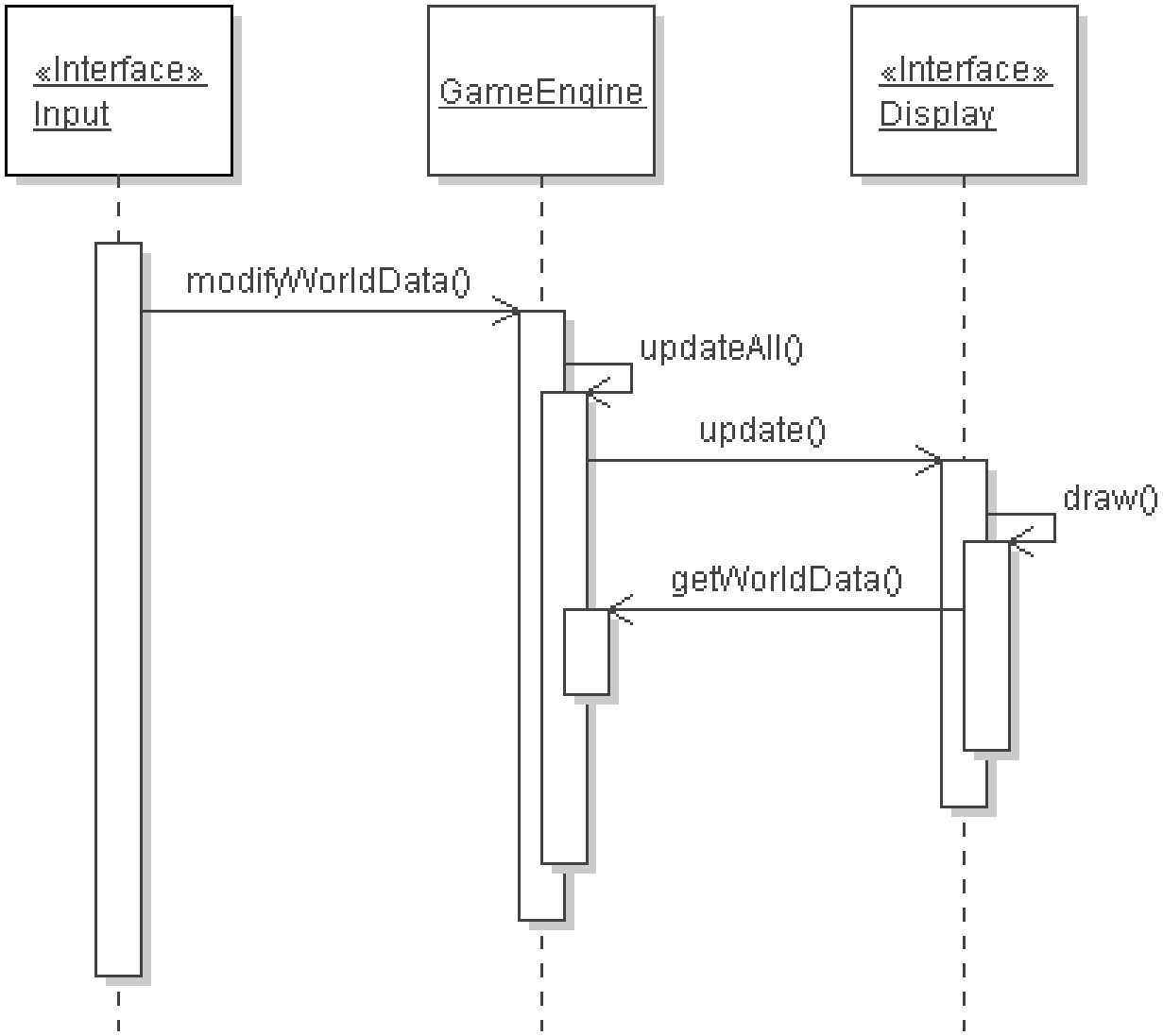



http://www.irit.fr/~Remi.Bastide/Teaching/Java/MVC/

# The Model-View-Controller (MVC) Pattern



# The Model-View-Controller (MVC) Pattern



# The Model-View-Controller (MVC) Pattern

- The Controller can also “select views” (but not change display contents).
- The View can obtain current Model status directly if needed.
- This visibility between the classes does not need to be known at compile time – it can be achieved entirely via references to objects implementing known interfaces.
- The main benefit of MVC is the controlled separation of the *presentation (UI)* and *application* (domain) layers and their responsibilities, while facilitating their interaction.
- This enhances the reusability of both sides (e.g. GUI classes used in other applications, or domain classes used with a new GUI).

# Model View Presenter