MVC and MVP

AARHUS UNIVERSITY
AARHUS UNIVERSITY SCHOOL OF ENGINEERING

MICHAEL LOFT
ML@ASE.AU.DK

Agenda

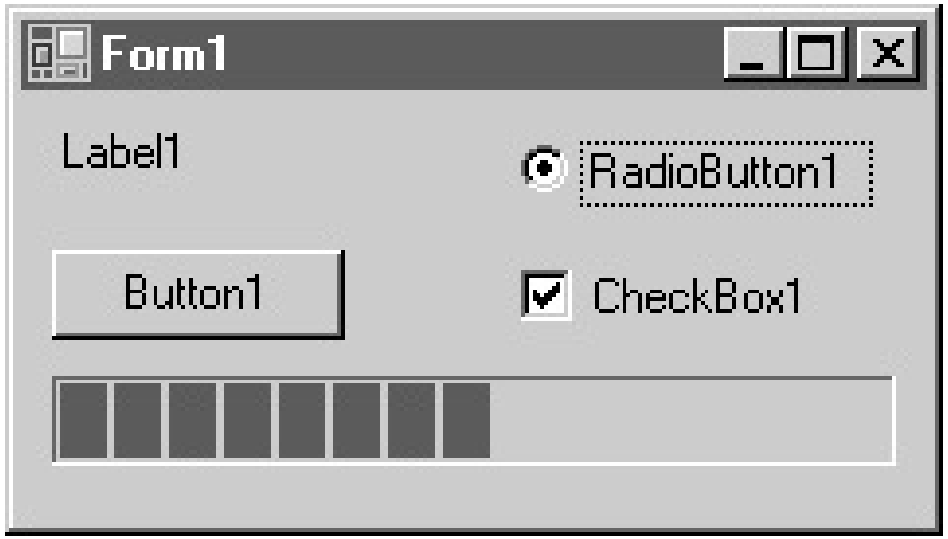
Forms and Controls

Model-View-Controller

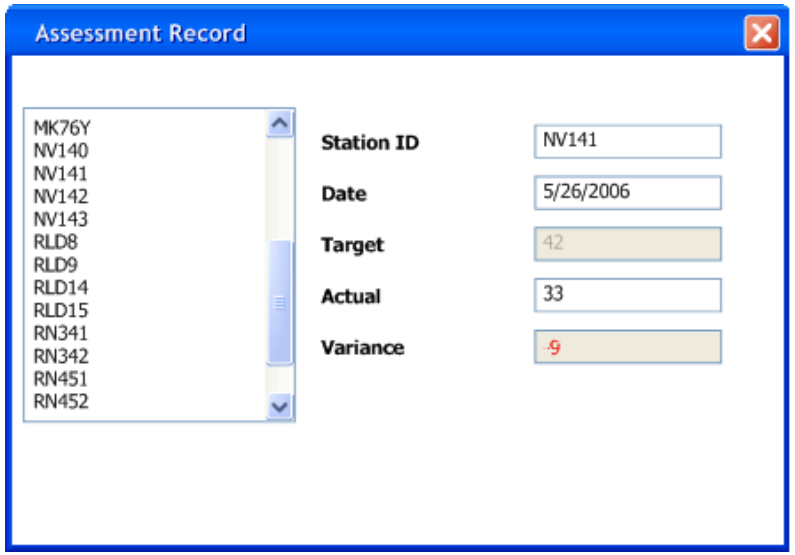
Model-View-Presenter

N-layer architecture and MVC/MVP

Forms and Controls



Forms and Controls Architecture



AKA: Designer Architecture

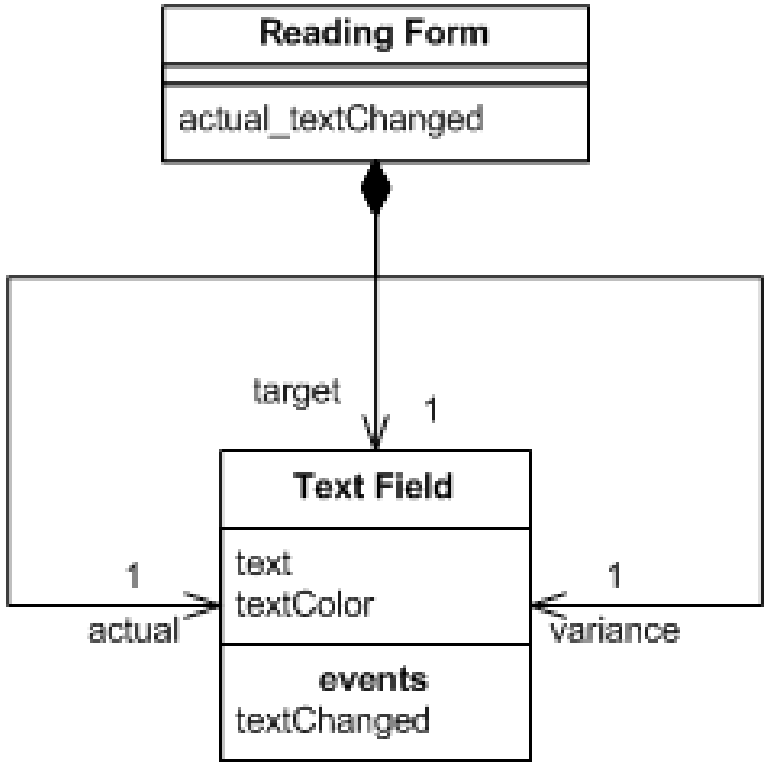
The form is specific to our application, but it uses generic controls.

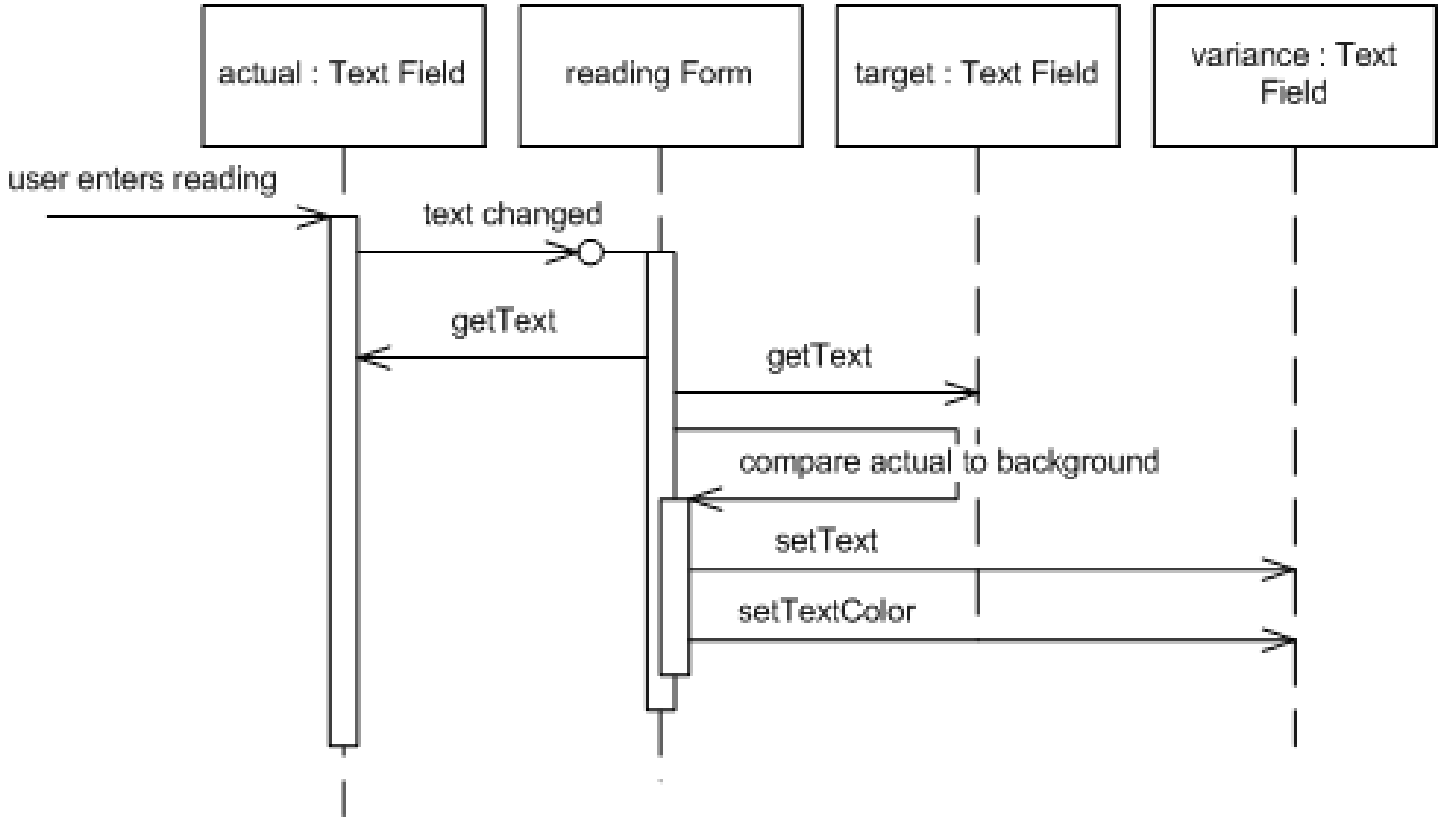
The form has two main responsibilities:

- Screen layout
- Form Logic

[Fowler]

F&C Class and Sequence Diagram





[Fowler]

Forms and Controls

1. Draw the GUI
2. Double-click buttons etc.
3. Implement the event handlers.

Easy peasy!

What is the problem?

Forms and Controls Summary

- Developers write application specific forms that use generic controls
- The form describes the layout of controls on it
- The form observes the controls and has handler methods to react to interesting events raised by the controls
- Simple data edits are handled through data binding
- Complex changes are done in the form's event handling methods

- Advantage: simple structure – easy to understand
- **Disadvantage: Domain/business logic is in the form class which makes it impossible to reuse with other UI frameworks! And it is also difficult to unit test the logic in the event handlers!** [Fowler]

Why GUI patterns?

Dealing with complexities of Graphical User Interfaces that work on/with data.

Managing data/state at various levels [Fowler]:

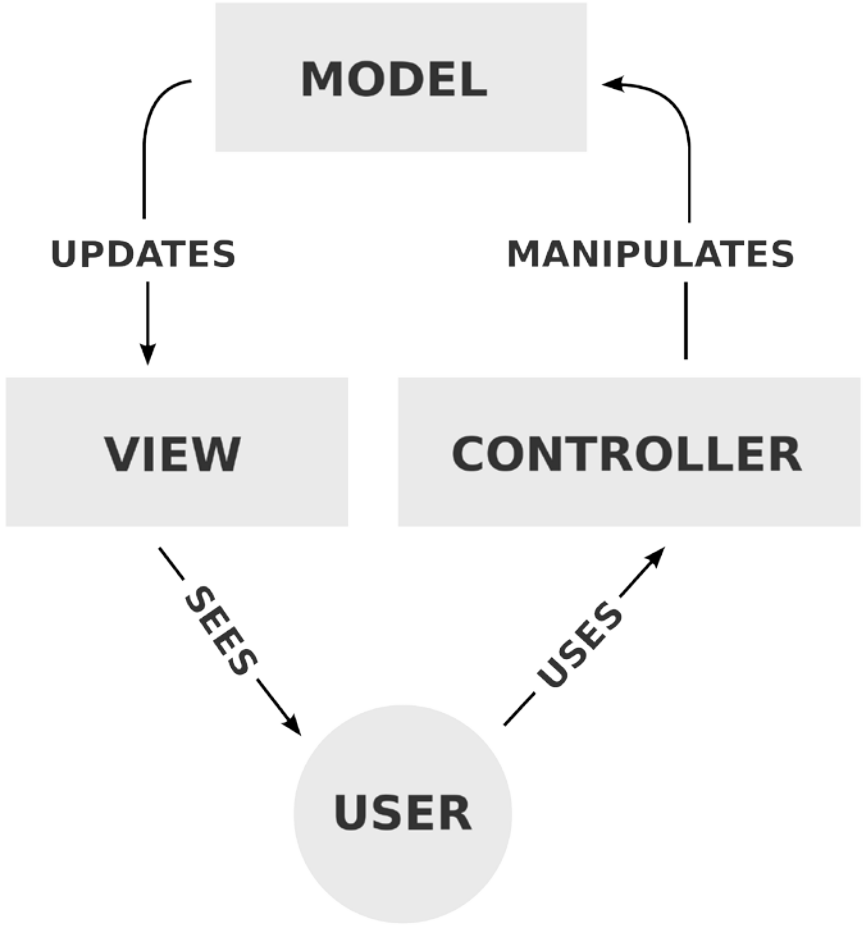
- GUI state
- Session state
- Record state

Separation of concerns

Better software solutions

- Manageable
- Testable
- Reusable
- Team workflow

Model – View -Control



MVC and MVP are compound patterns

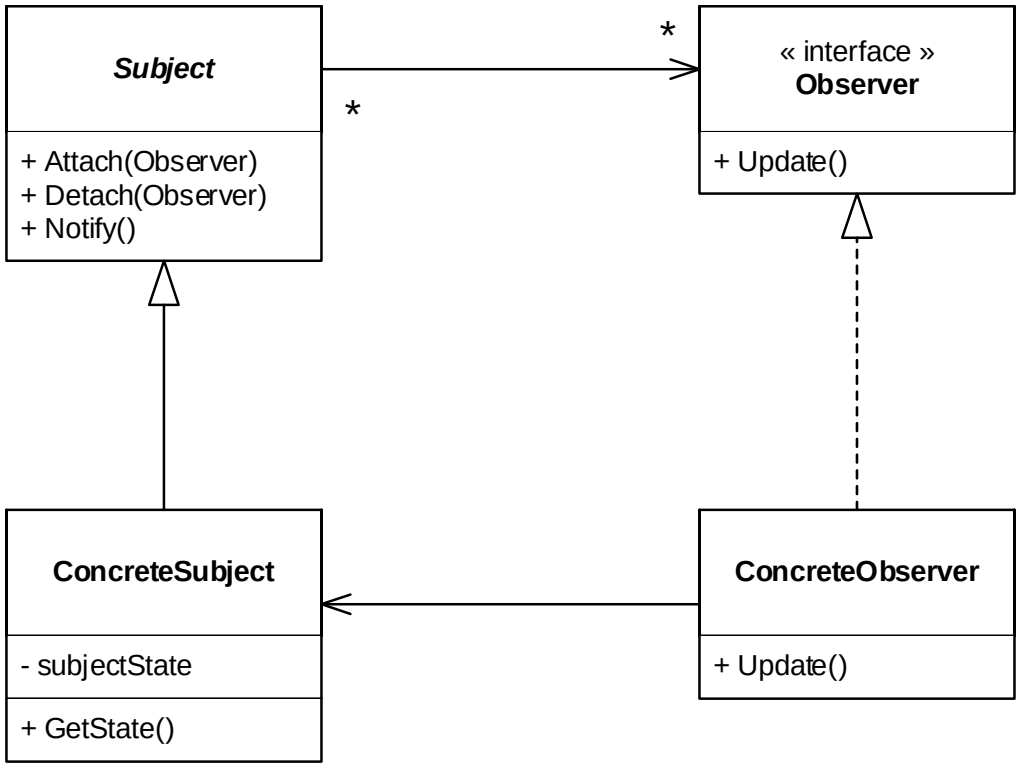
MVC/MVP is based on

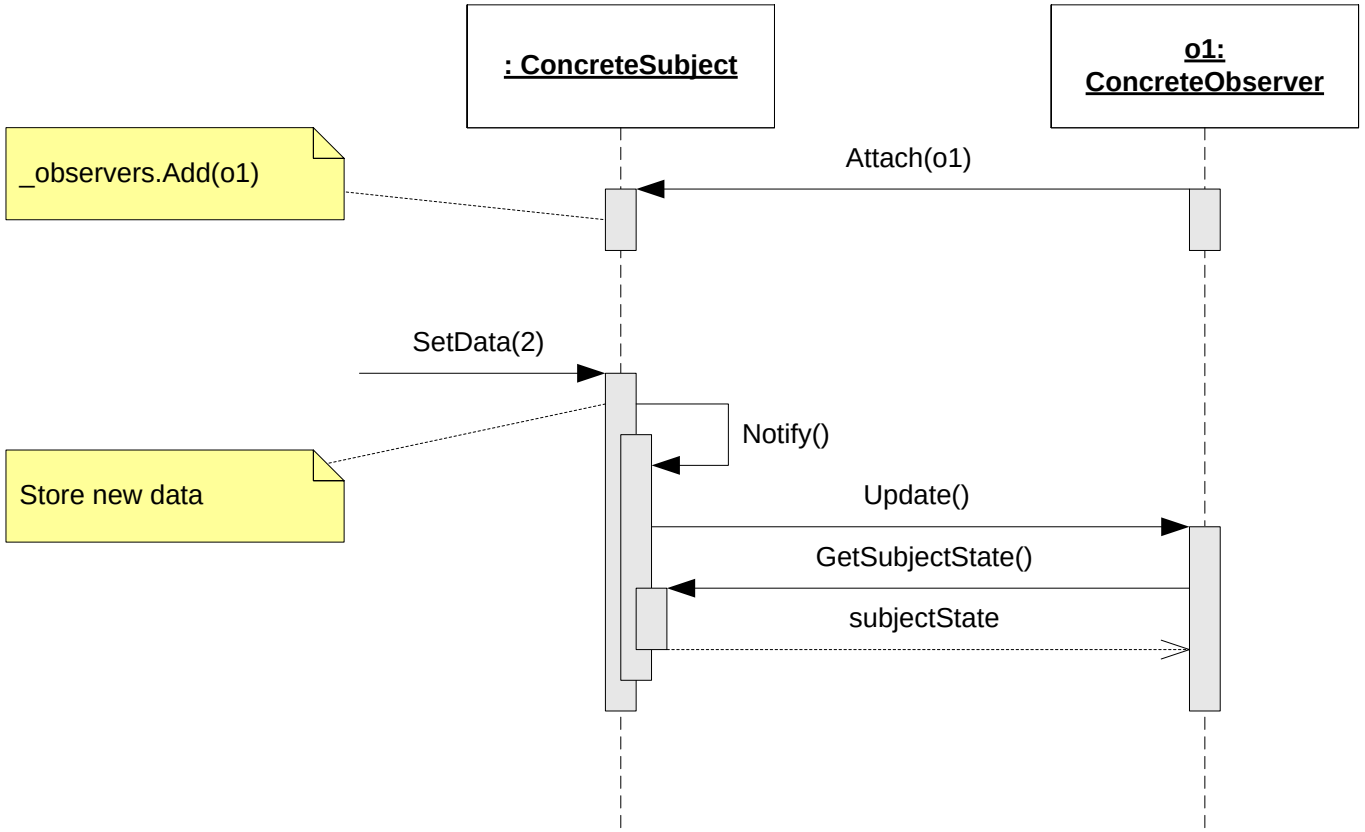
Observer

Strategy

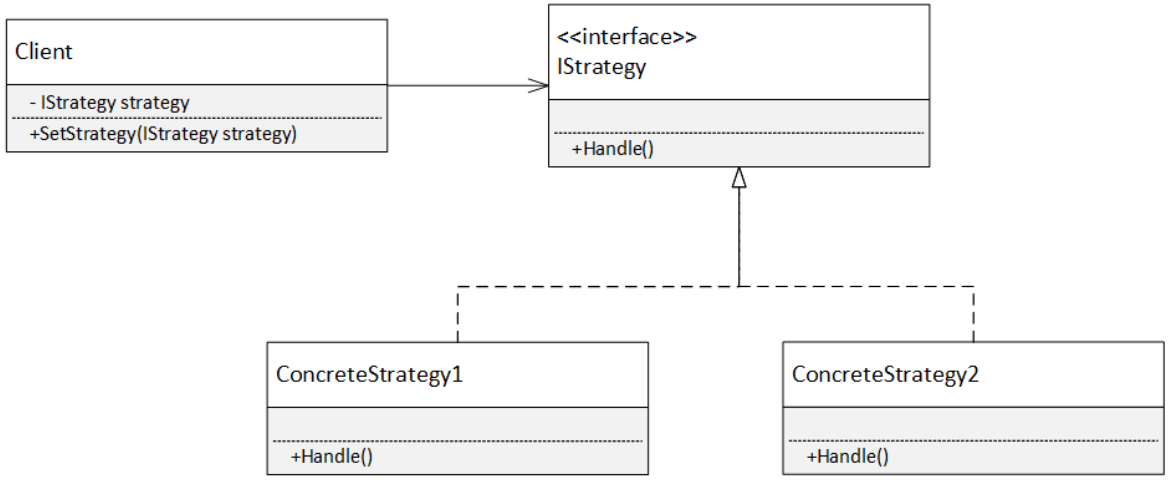
Composite

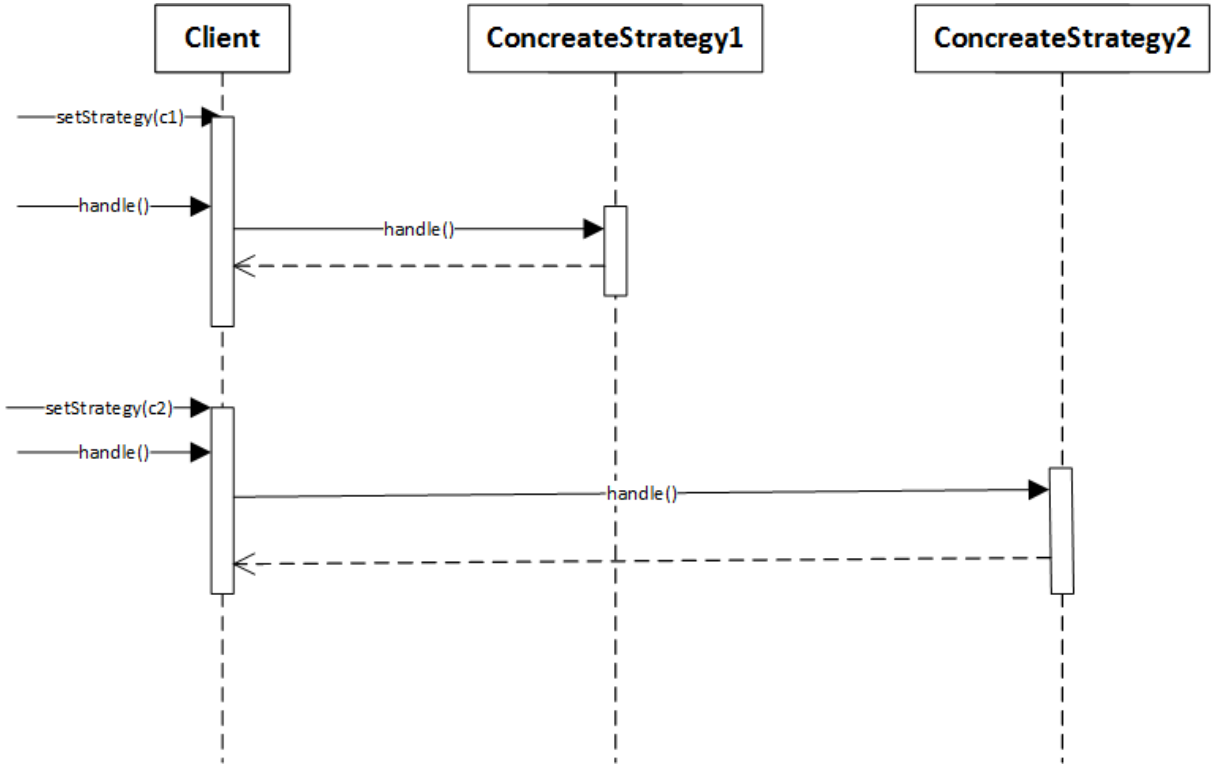
Observer recap



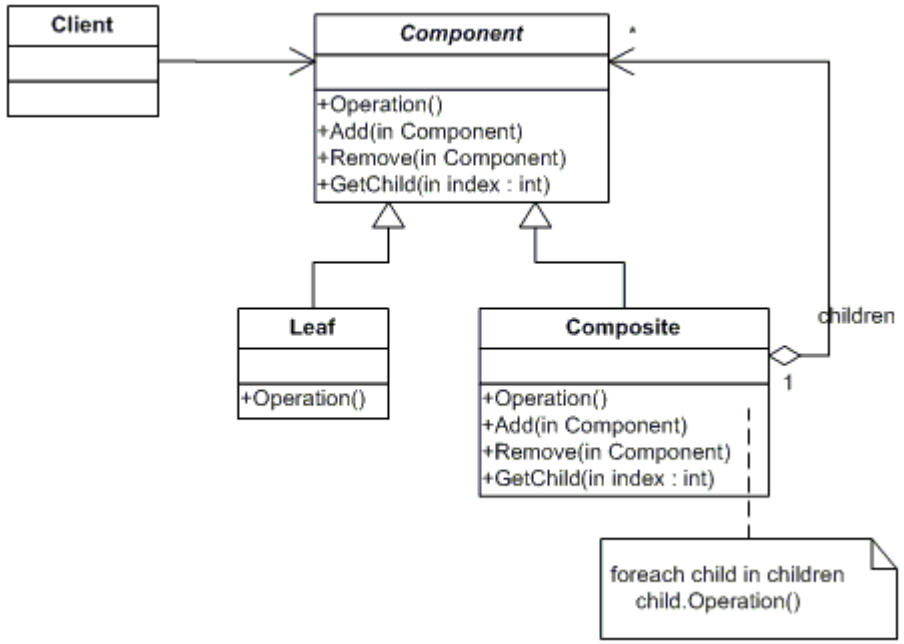


Strategy recap





Composite design pattern (very brief)



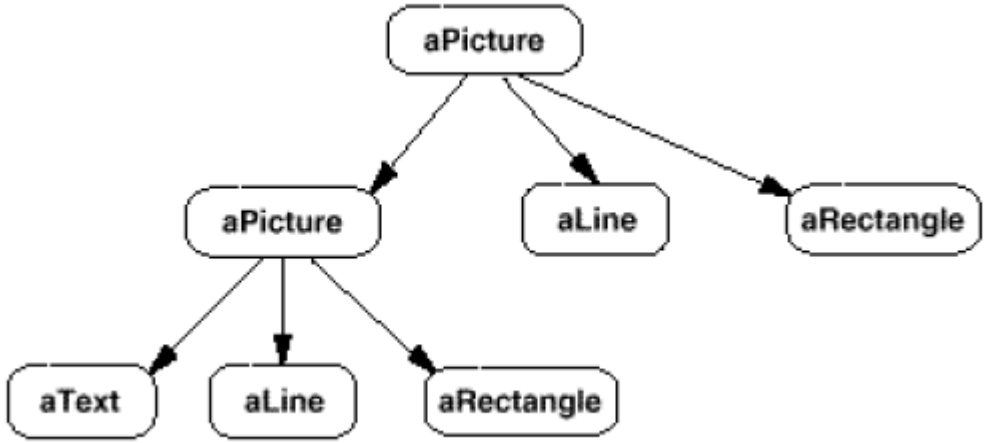
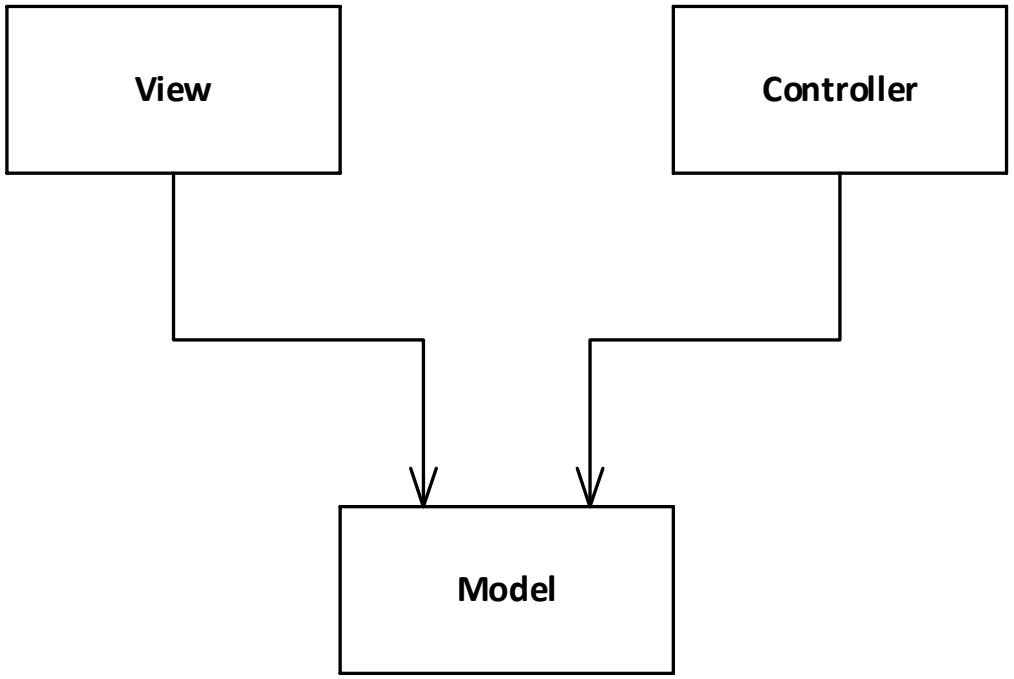


Image sources:
Design Patterns - Elements of Reusable Object-Oriented Software & http://www.dofactory.com/net/composite-design-pattern

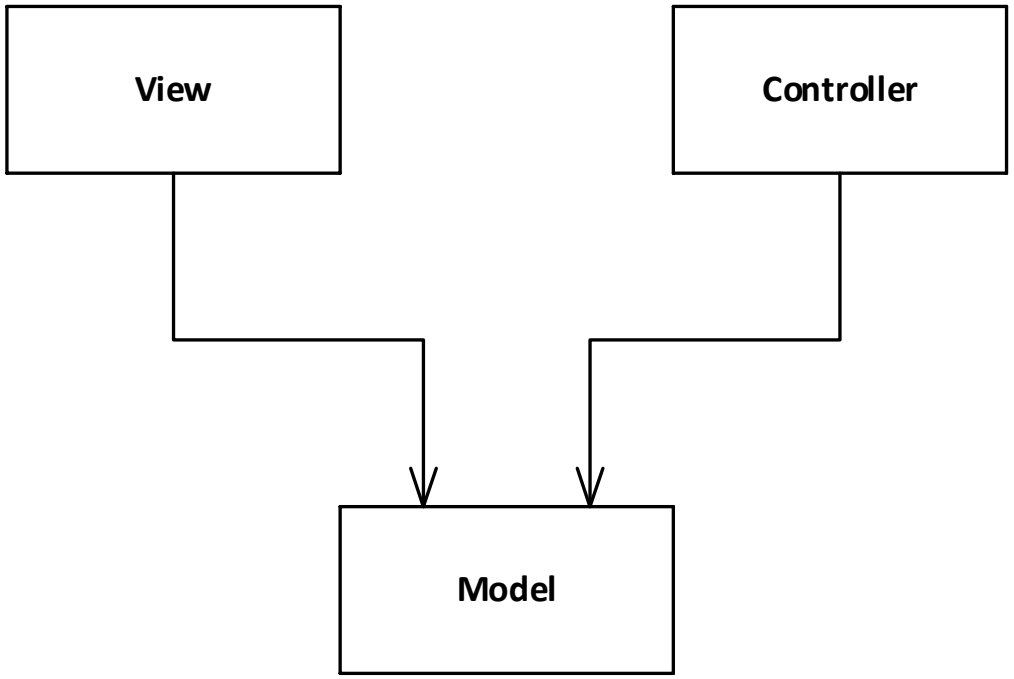
Original MVC



Clear division between domain objects that **model** our perception of the real world, and presentation objects that are the GUI elements we see on the screen.

The model is self contained domain objects, which do not depend on anything else.

Original MVC

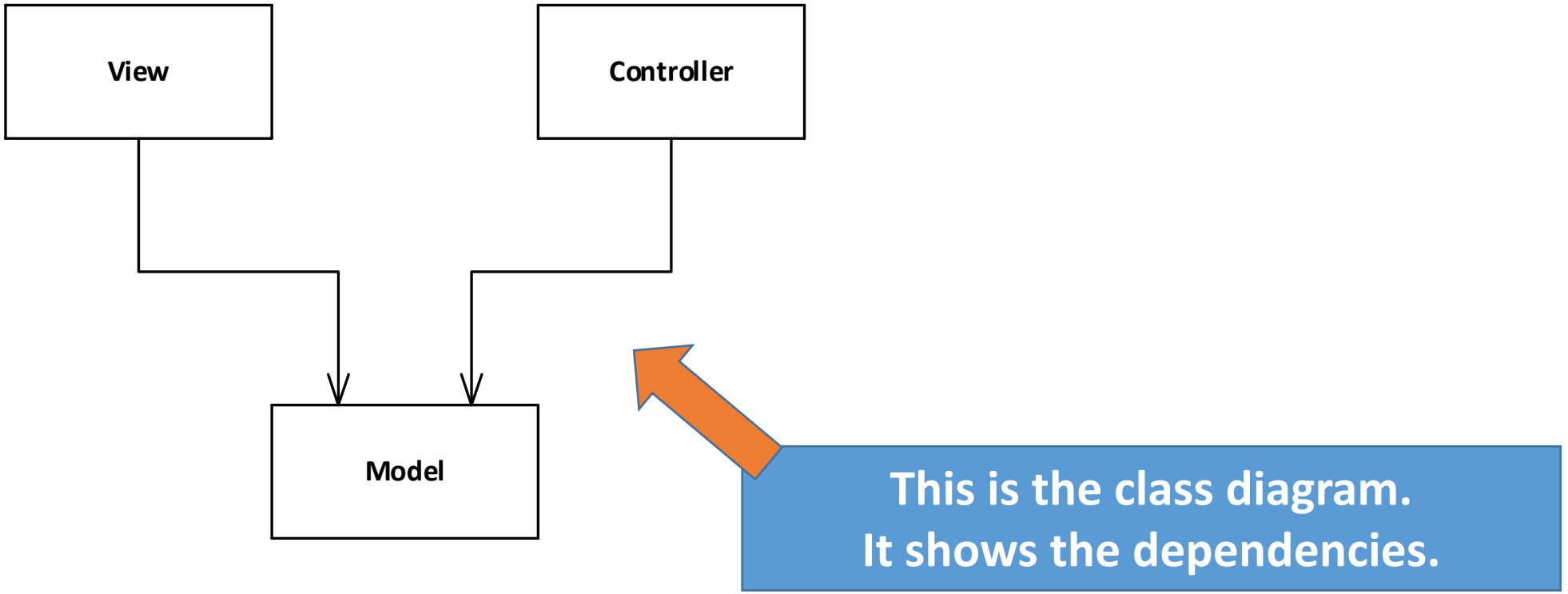


The presentation part of MVC is made of the two remaining elements: **view** and **controller**.

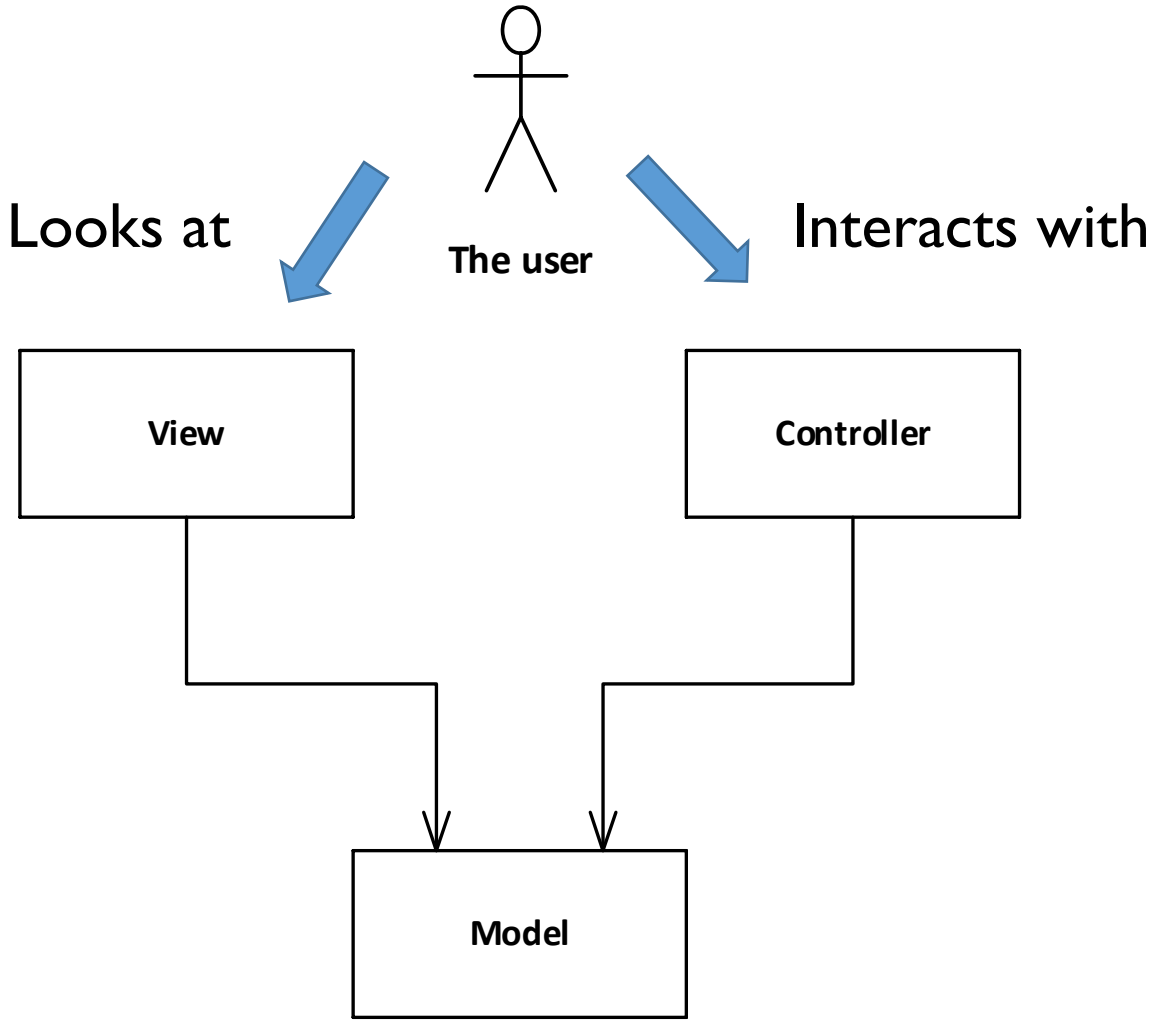
The controller's job is to take the user's input and figure out what to do with it.

The view is how the model is displayed to the user.

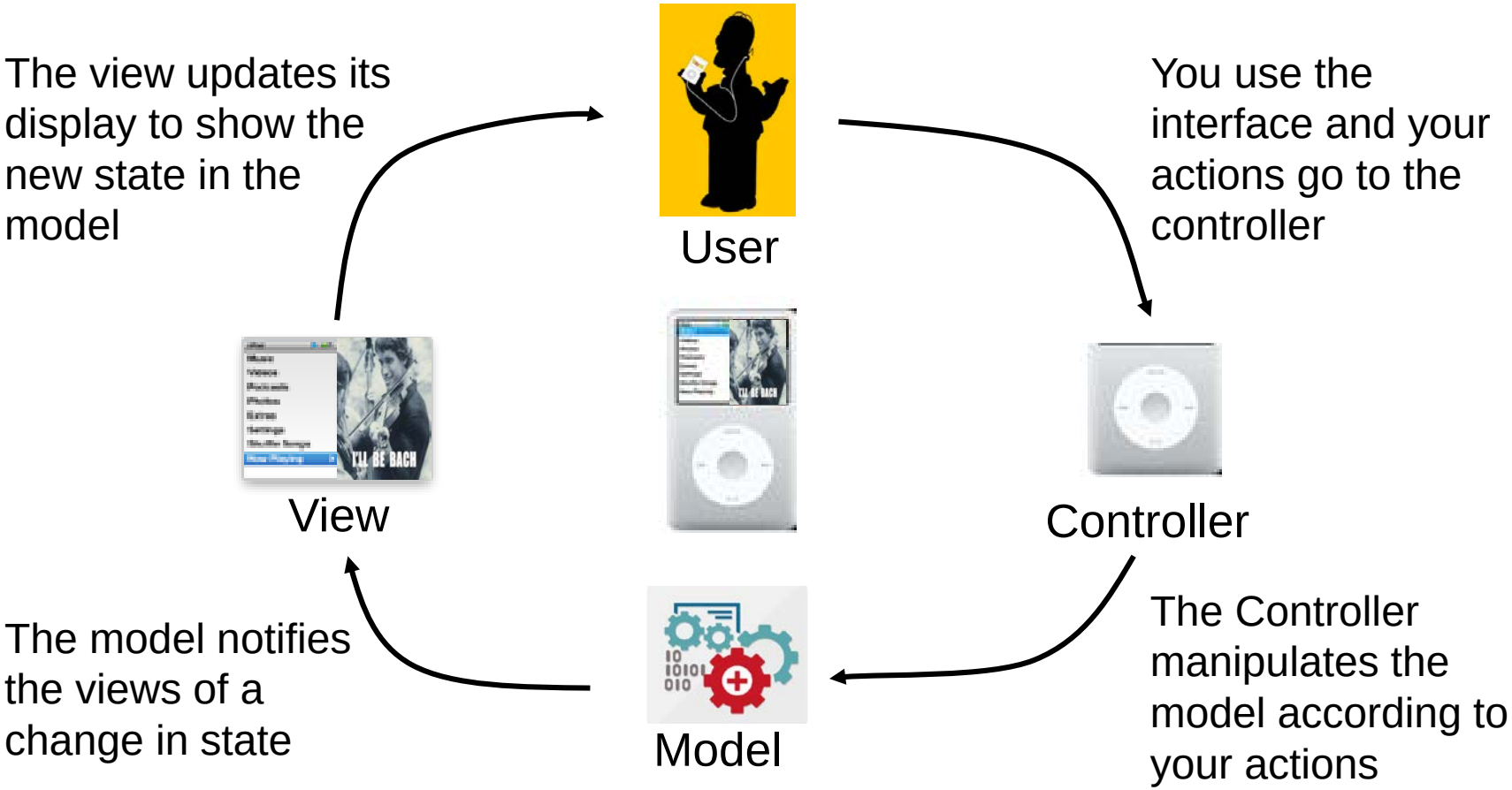
Original MVC



Original MVC



MP3 player seen as a Model View Controller



The model contains all the state, data and logic needed to store and play mp3s

[Head First Design Patterns]

Where is the view logic?

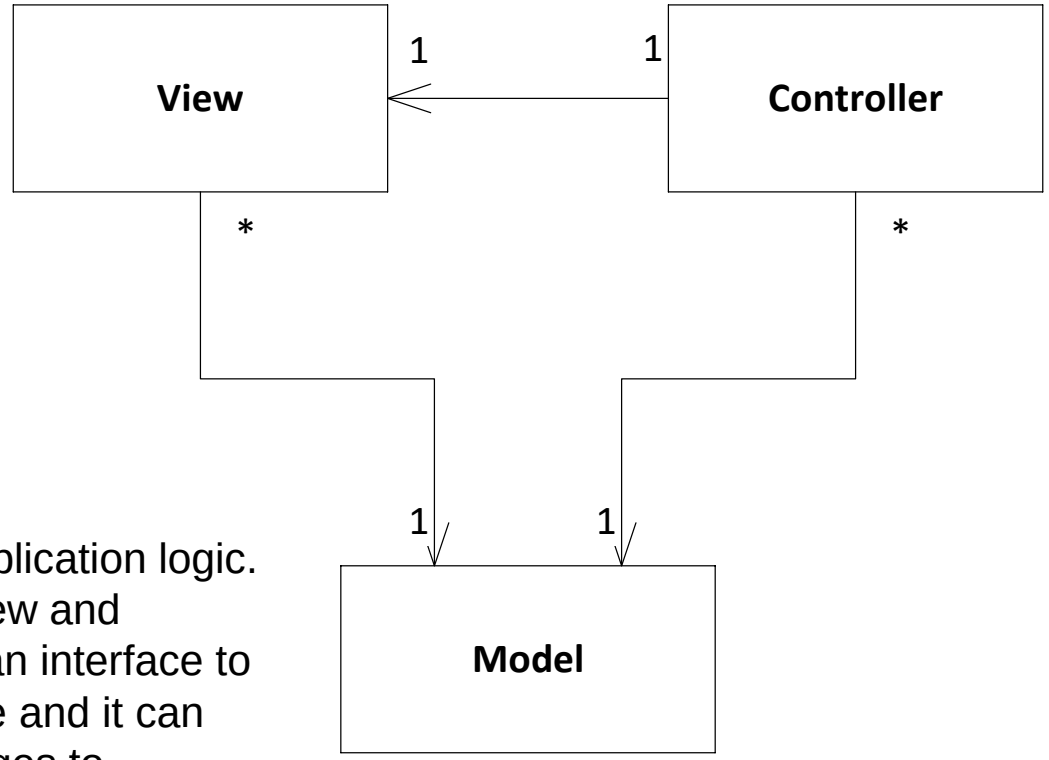
We may want to:

- Change the color of a widget.
- Go to a new screen.
- Enable/disable parts of the GUI.
- ...

MVC Class Diagram

View

Gives you a presentation of the model. The view usually gets the state and data it needs to display directly from the model.



Controller

Takes user input and figures out what it means to the model

Model

Holds all the data, state and application logic. The model is unaware of the view and controller, although it provides an interface to manipulate and retrieve its state and it can send notifications of state changes to observers

Model-View-Controller Summary

Make a strong separation between presentation (view & controller) and domain (model)

Divide GUI widgets into a controller (for reacting to user stimulus) and view (for displaying the state of the model). Controller and view should (mostly) not communicate directly but through the model

Have views (and controllers) observe the model to allow multiple widgets to update without needed to communicate directly - Observer Synchronization

Advantage: a strong separation of model and UI, makes the application easier to update and maintain.

Disadvantage: requires more code than Forms & Controls [Fowler]

MVC in modern GUIs (.Net, Java, etc.)

MVC is probably the widest quoted pattern in UI development

- but it's also the most misquoted!
- the reason for this is that parts of classic MVC don't really make sense for rich clients these days (.Net, Java, etc.)

In modern GUI frameworks, the View handles the initial interaction, but immediately delegates to the controller.

Model View Presenter

Two variants:

- Supervising Controller
- Passive View

The **view** in MVP is a structure of widgets. No behavior!

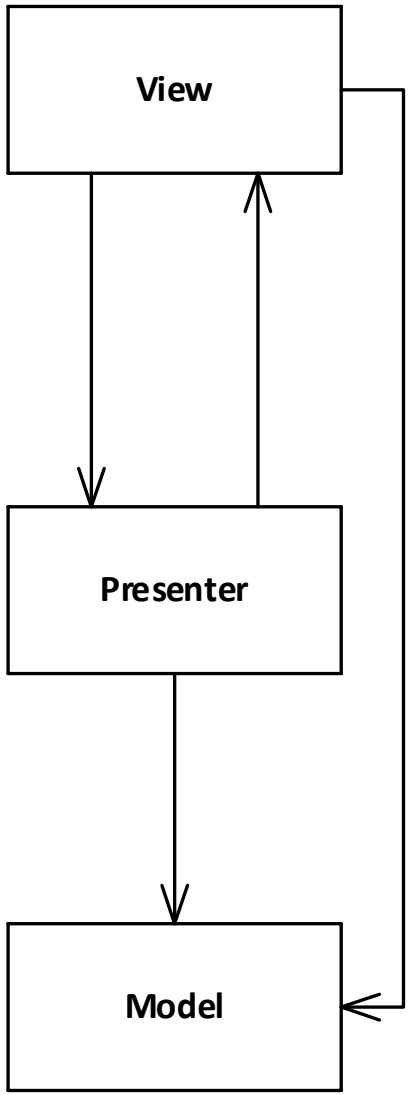
The reaction to user input is placed in a separate **presenter** object

- The fundamental handlers for user gestures still exist in the widgets, but these handlers pass control to the presenter.

The presenter then decides how to react to the event

MVP – Supervising Controller

MVP – Supervising Controller

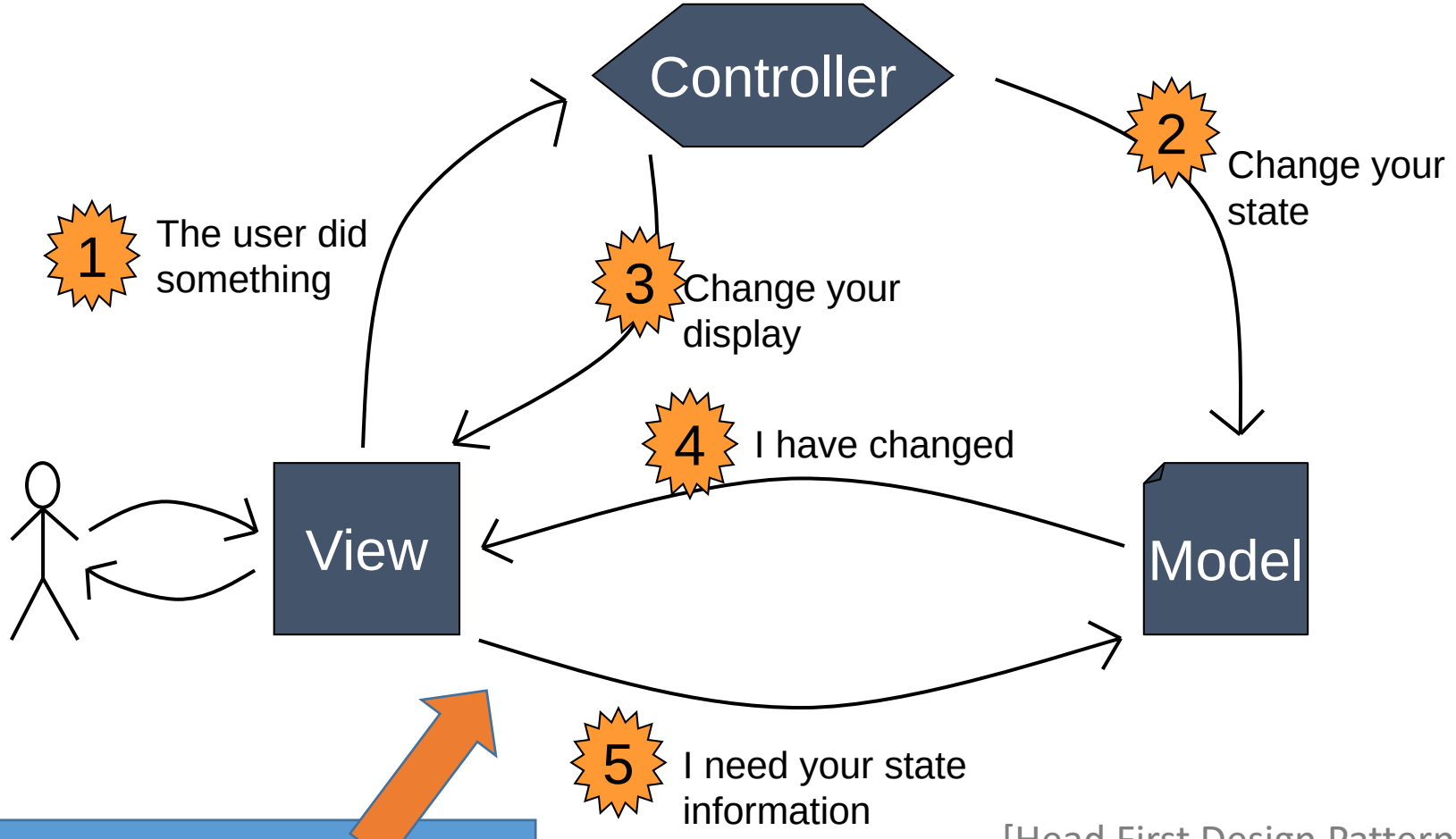


Factor the UI into a view and controller where the view handles simple mapping to the underlying model and the controller handles input response and complex view logic.

Let the view handle as much as possible and only step in when there's more complex logic involved.

This is a class diagram.
It shows the dependencies.

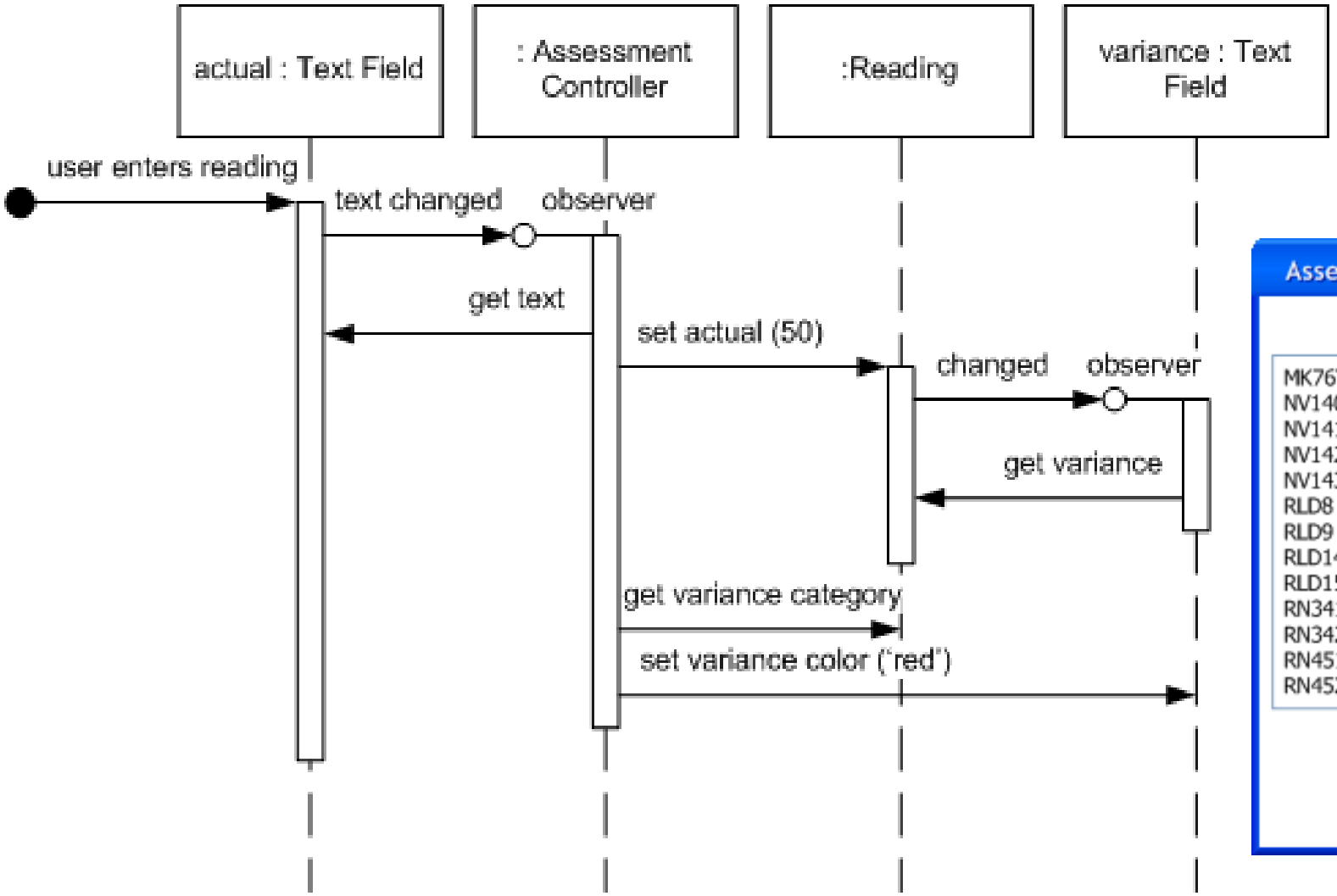
MVP the Collaboration

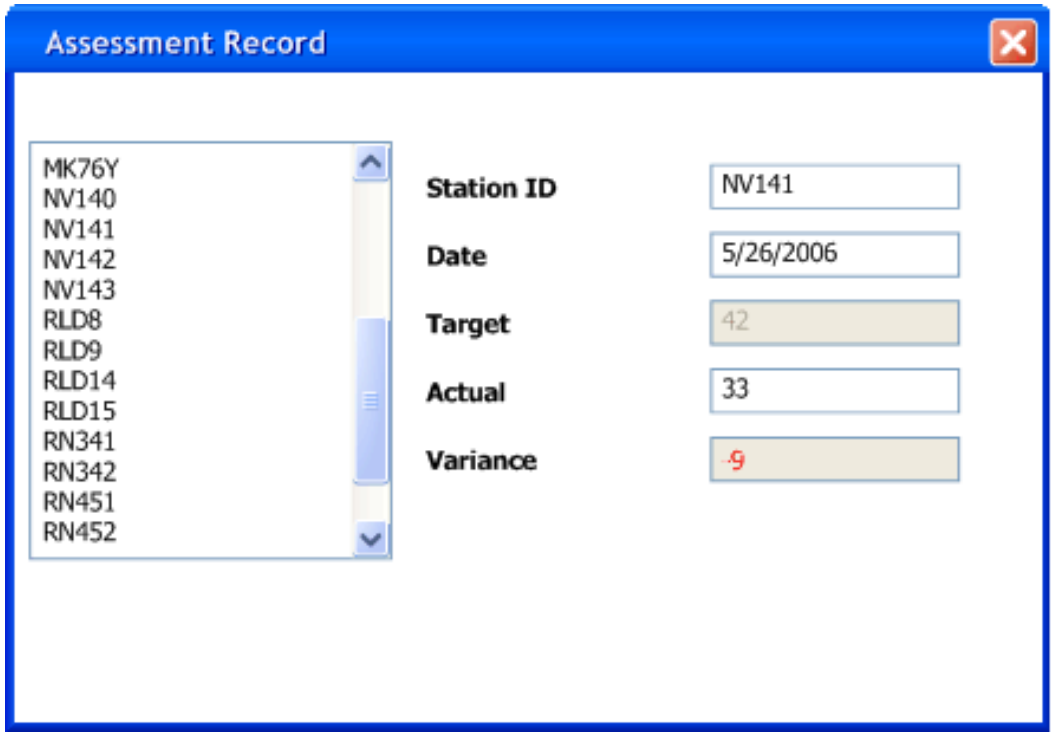


[Head First Design Patterns]

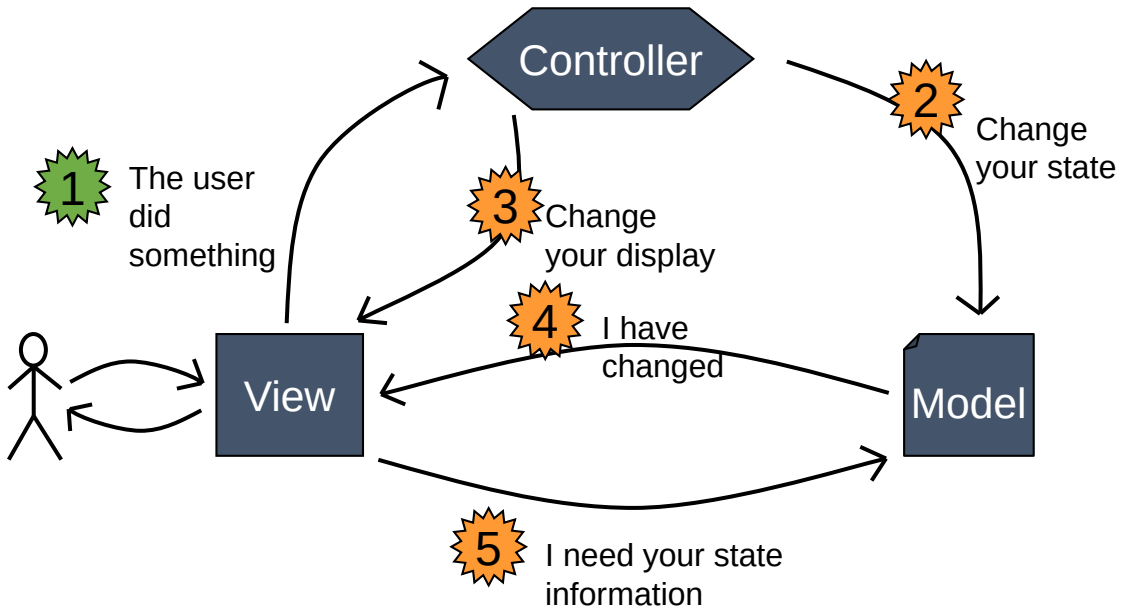
This diagram shows the data flow.

MVP – Supervising Controller sequence





A compound Pattern



Composite

The *view* consists of a nested set of windows, panels and controls.
Each display component is a composite (inherits from Panel) or a leaf (inherits from Control)

Strategy

The view and controller may implements the Strategy pattern. The *view* is the object that is configured with a strategy. The *controller* provides the strategy

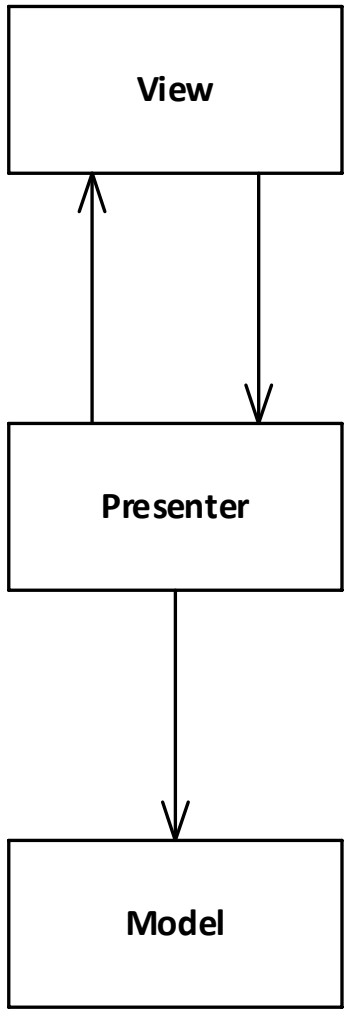
Observer

The *model* implements the Observer Pattern to keep interested objects updated when state changes occur.
Use of the Observer Pattern keeps the *model* completely independent of the *views* and *controllers*

[Head First Design Patterns]

MVP – Passive View

MVP – Passive View



The View has no direct association to the Model.

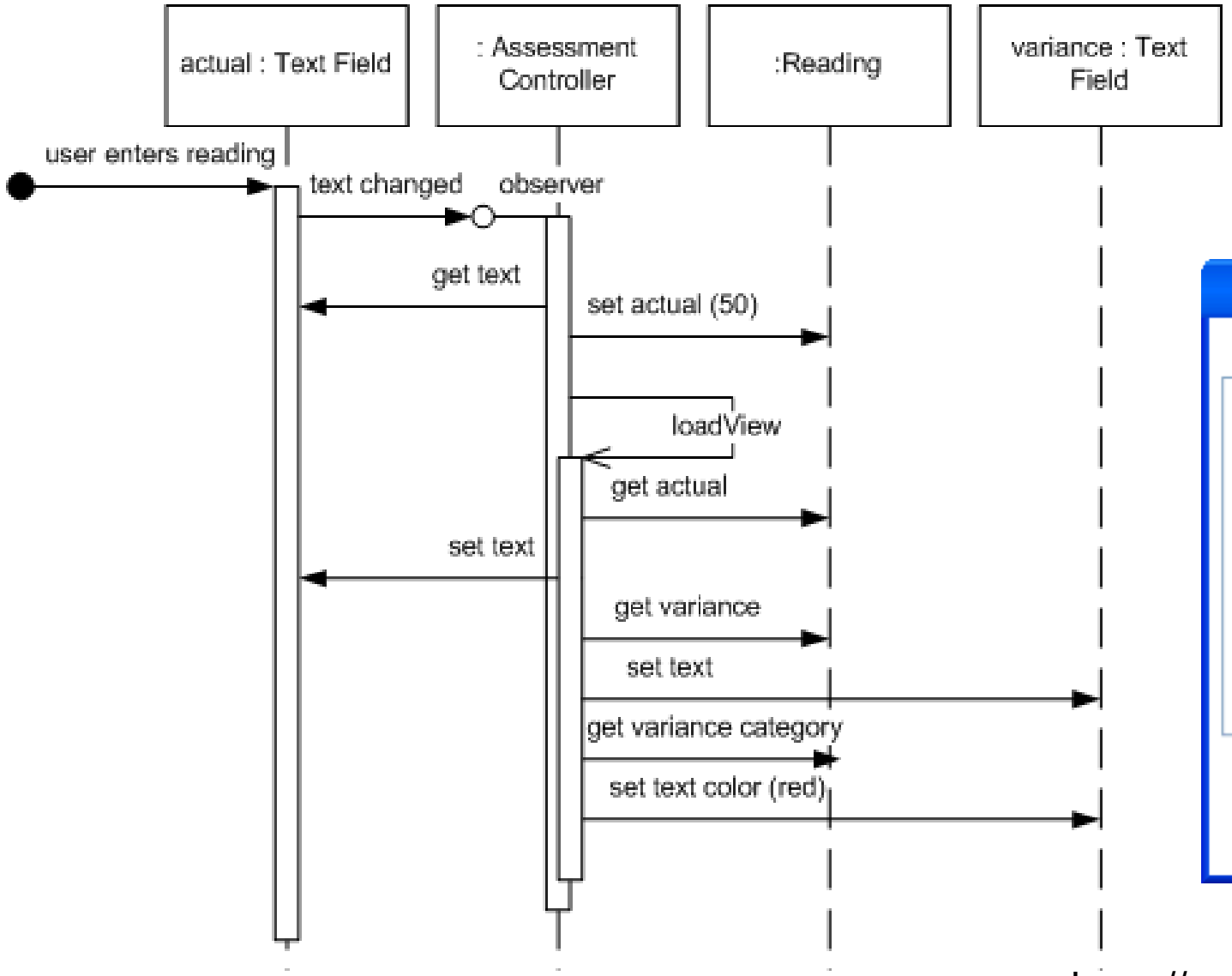
The View only contains the widgets that shows the data on the screen, and it will hand over all user input to the Presenter (Fowler calls it "Controller")

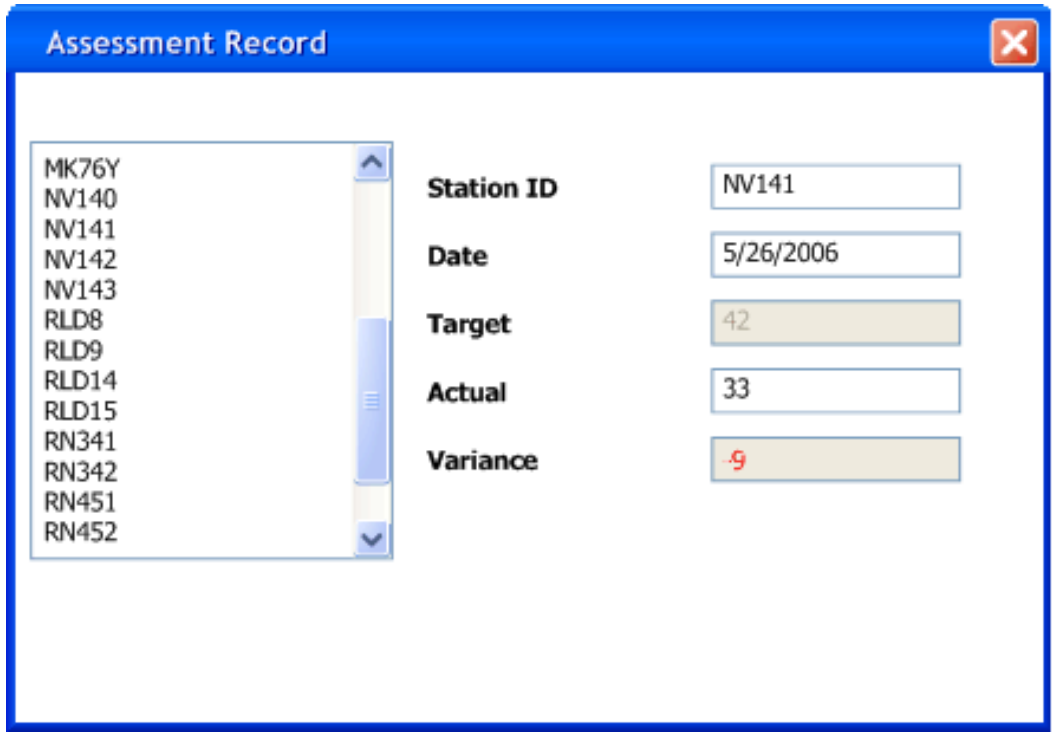
The Presenter contains all the GUI logic and is responsible for both updating the Model and updating the View

- the Presenter populates the Widgets in the View with data from the Model.

The Presenter can be notified of changes to the Model.

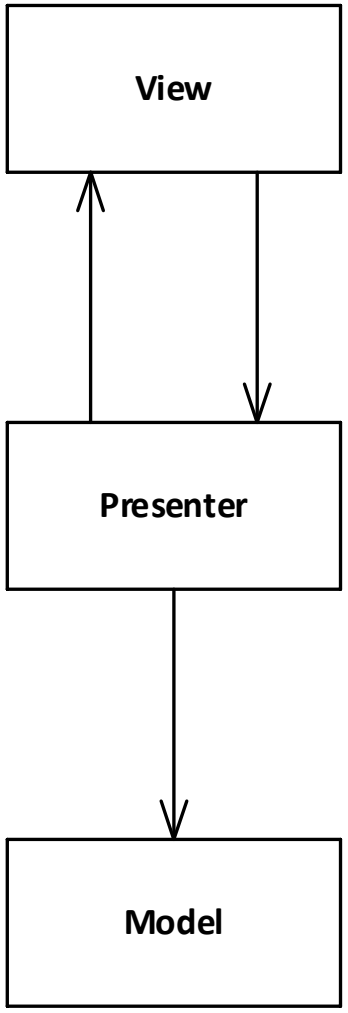
MVP – Passive View sequence diagram



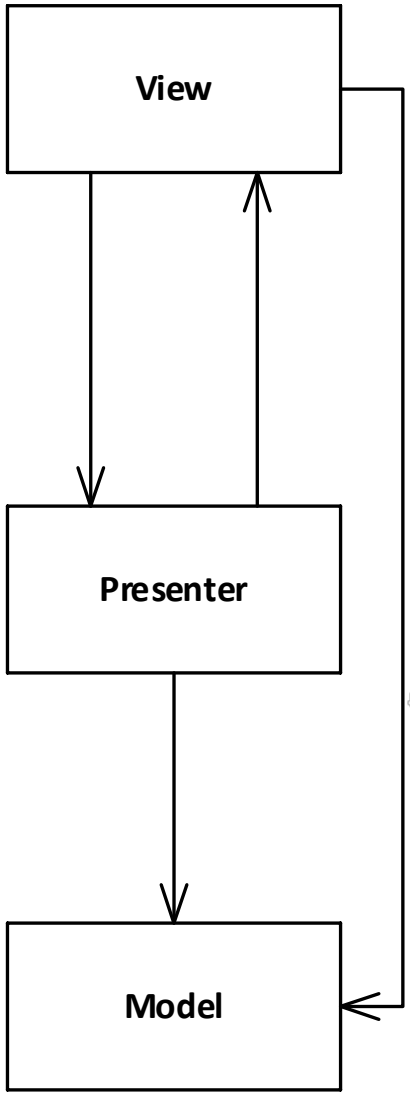


https://www.martinfowler.com/eaaDev/PassiveScreen.html

Passive View versus Supervising Controller



VS



Model-View-Presenter Summary

User gestures are handed off by the widgets to the presenter

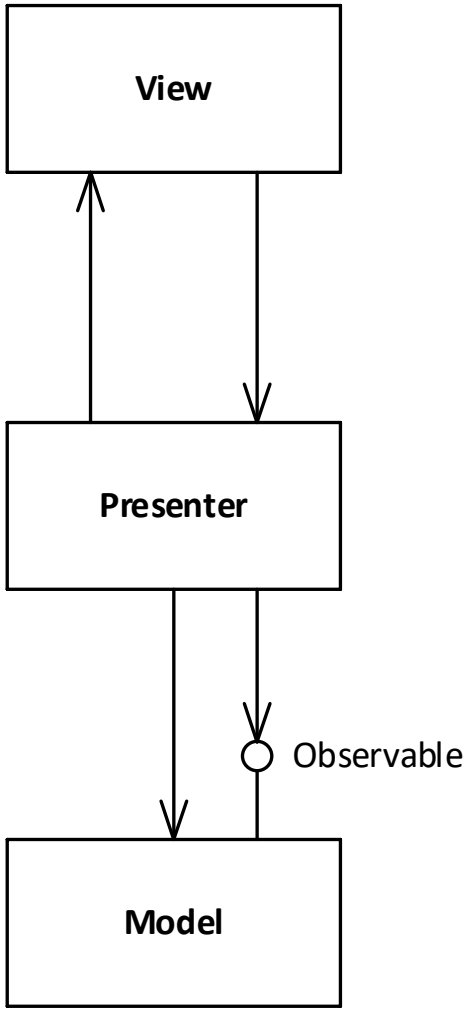
The presenter coordinates changes in a domain model

Different variants of MVP handle view updates differently

- These vary from using Observer Synchronization
  - **Supervising Controller**
- to having the presenter doing all the updates
  - **Passive View**

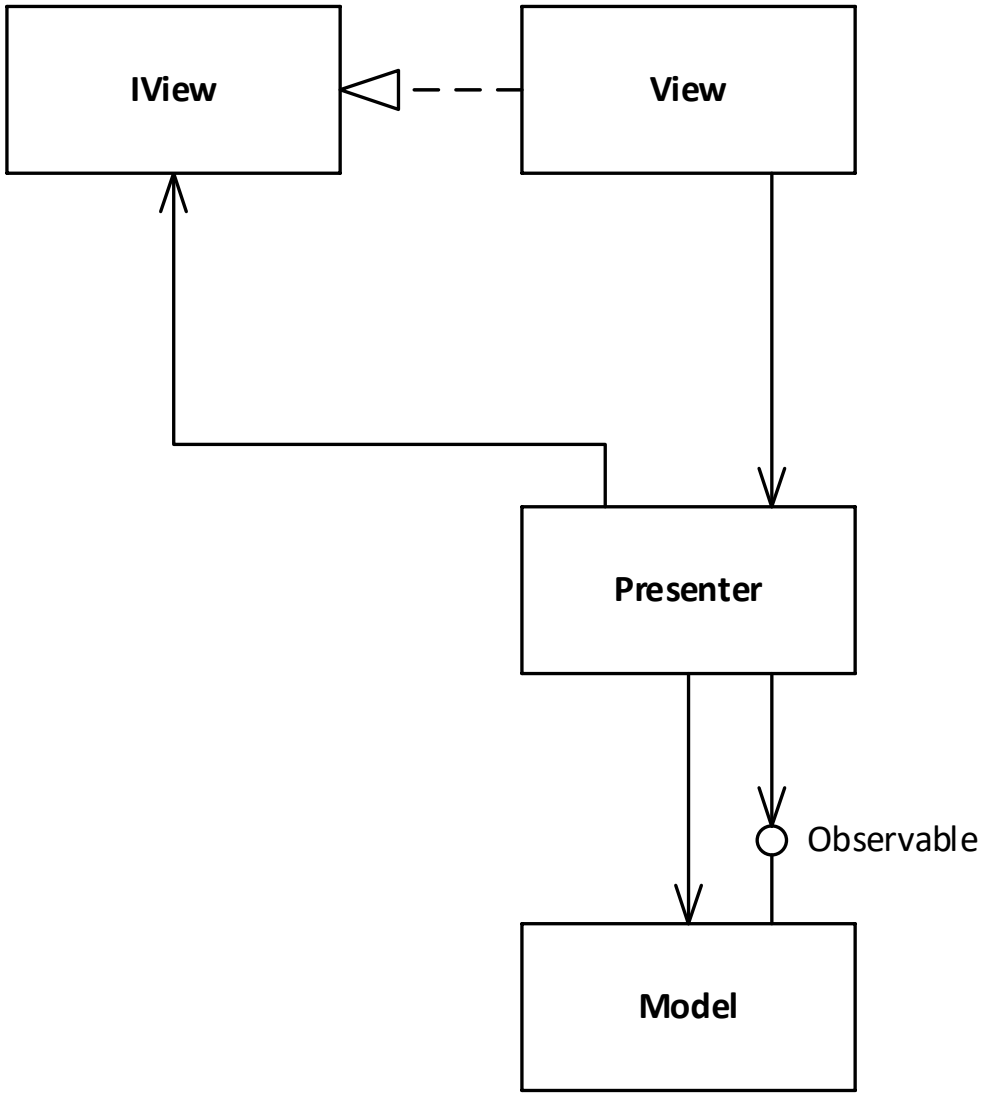
[Fowler]

Minimizing coupling



High coupling from presenter to view.

A view interface

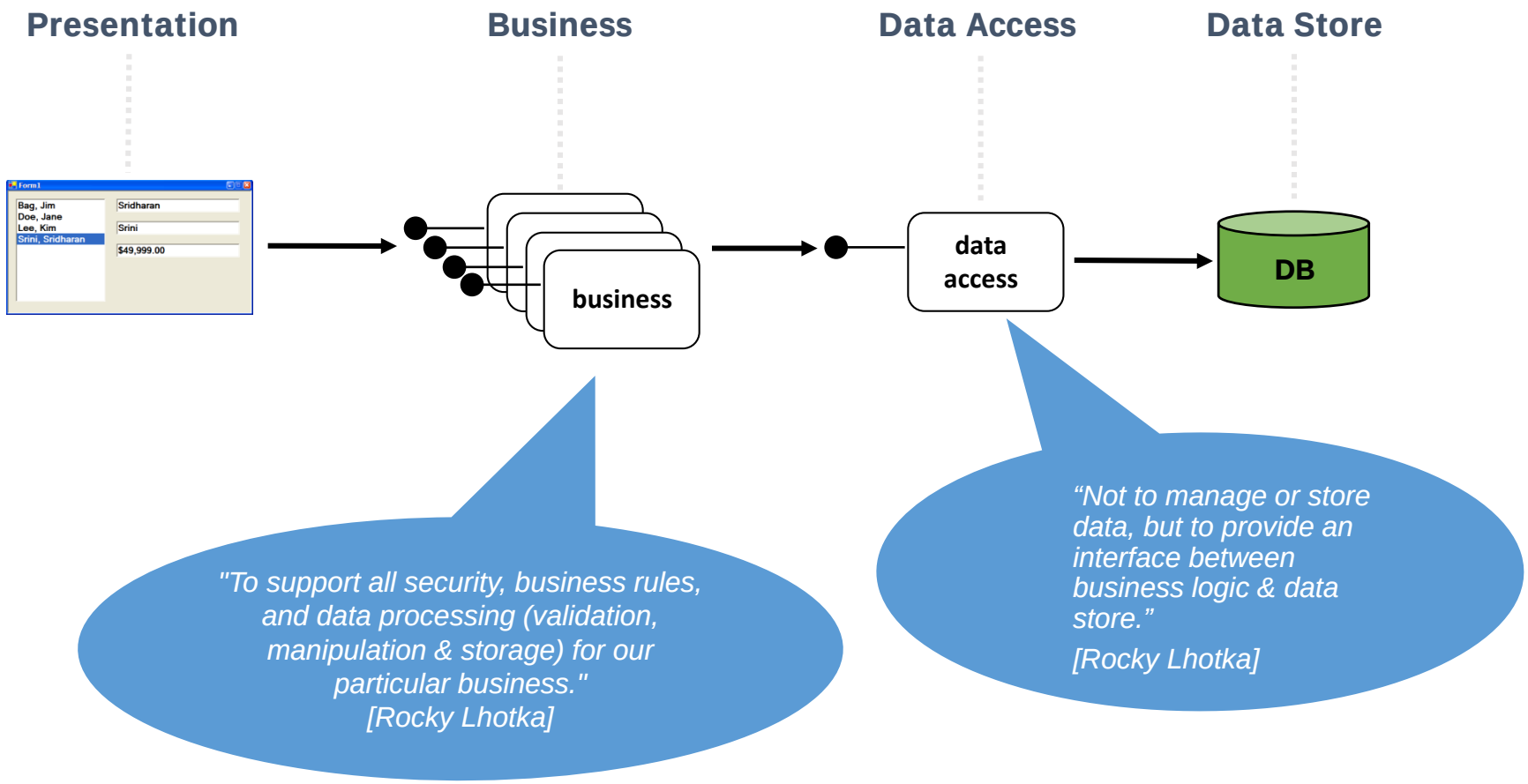


High coupling from presenter to view.

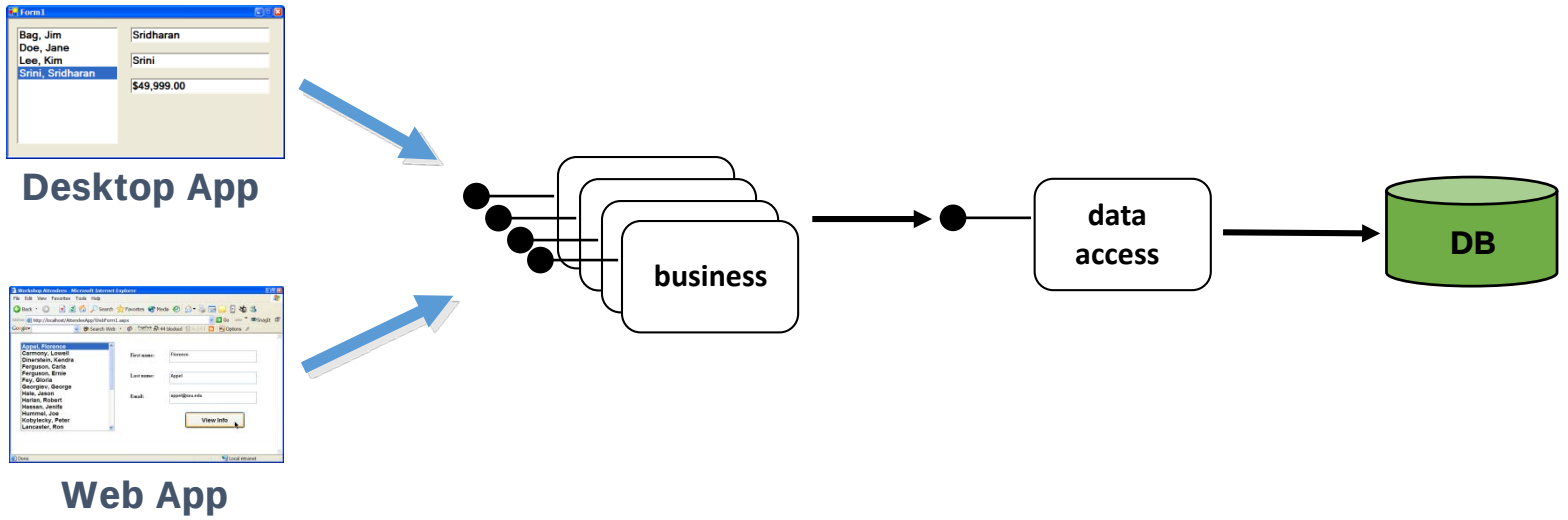
We can add an interface to the view.

N-layer architecture and MVC/MVP

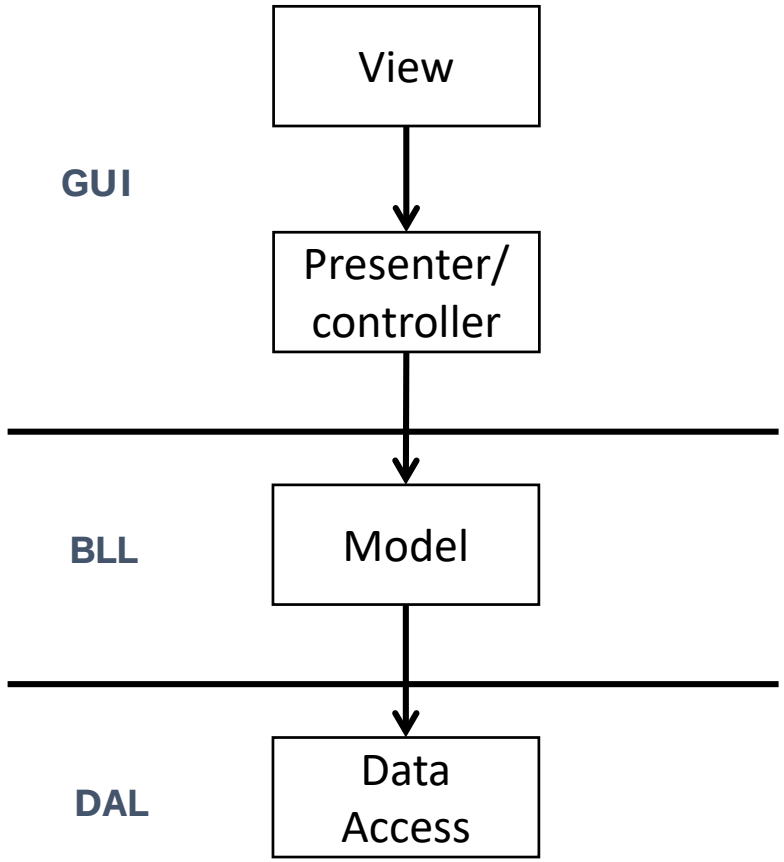
Typical 3-Layer Design



Reusable Model



Mapping MVC/MVP to 3-layer Architecture



Your turn

Solve the exercise

AARHUS UNIVERSITY

References and image sources