# Compound patterns

Model View Controller

- Patterns are often used together, but…
- ***do not force*** it
  - A simpler solution is always better.
    - do not strive to use a design pattern

- Use a pattern when
  - the need arises naturally
  - the flexibility the pattern introduces will be used in the future

## Compound patterns

- Compound pattern
  - Combines two or more patterns into a solution that solves a ***recurring*** *or a* ***general*** *problem*
- We can combine patterns to achieve a particular solution but
  - Not every combination is a compound pattern ☺

## Model View Controller

- MVC is the most famous compound pattern
  - an architectural pattern for applications with many User interfaces
- First described in 1979 and published in 1987
  - *"Applications Programming in Smalltalk-80: How to use Model-View-Controller"*
- Separating View from domain concerns

## Life without MVC

- A huge GUI class that
  - Displays data and
  - captures user interactions and
  - interprets/decides what to do and
  - holds data structures and manipulates them

- Developers used to create a View using window and then write all logical code
  - View classes: button, panel, text area,….
  - Logical code: Event handling, initialization and data model, …

## Life without MVC

- A huge GUI class that
  - Displays data and
  - captures user interactions and
  - interprets/decides what to do and
  - holds data structures and manipulates them

- Everything is in one place is always a **bad idea**

## Developing UI w/o separation

- Everything is in one place is always a **bad idea**
  - **Maintainability:** A bug in one part breaks everything.
    - one developer's changes might break the other code.
  - **Collaboration:** Two developers cannot work on it.
  - **Extendibility:** How do you add a new feature?
  - **Testability:** Cannot test logic without the UI.

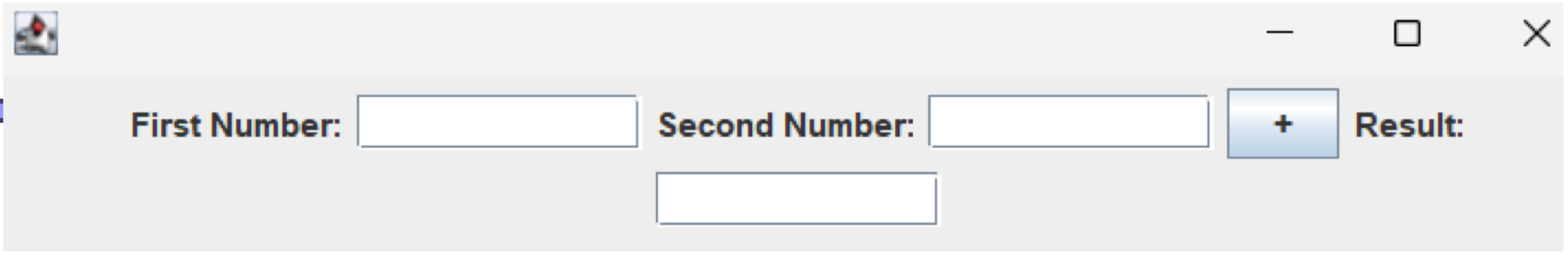
- Reason: there is a **very tight** coupling between **visualization**, handling interaction, data and business logic

# Separate Model and View

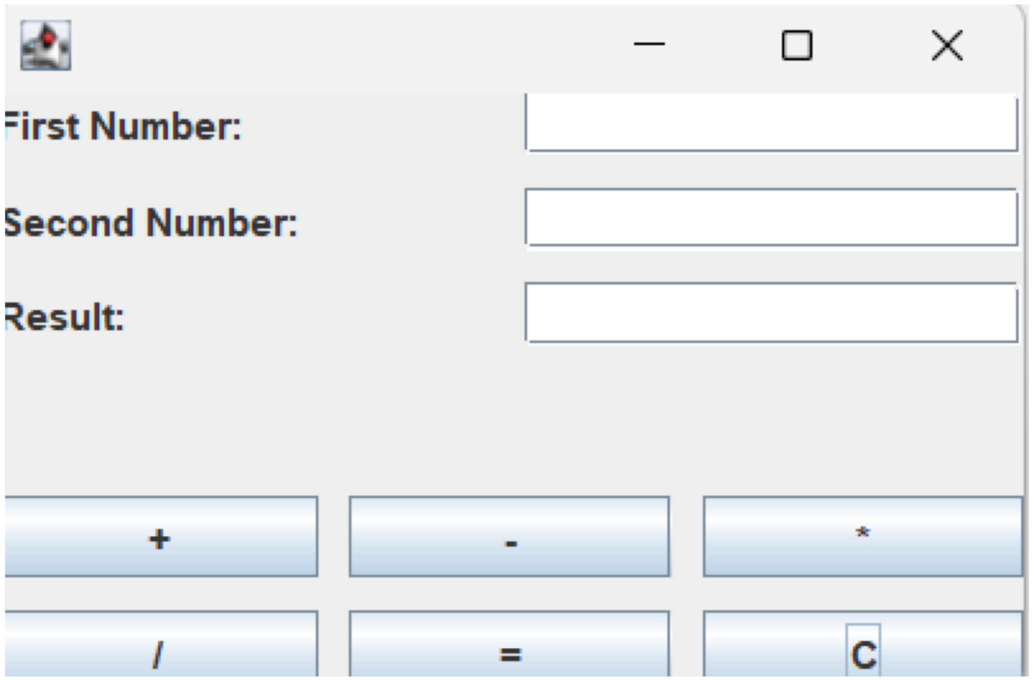
```
class CalculatorModel {
    private int result;
    public void add(int x, int y) {
        result = x + y;
    }
    public void subtract(int x, int y) {
        result = x - y;
    }
    public void multiply(int x, int y) {
        result = x * y;
    }
    public void divide(int x, int y) {
        result = x / y;
    }
    public int getResult() {
        return result;
    }
}
```

- Model holds data and logic.
- It knows *nothing* about buttons or text fields.
- It is pure, reusable logic.

# Reuse is easy

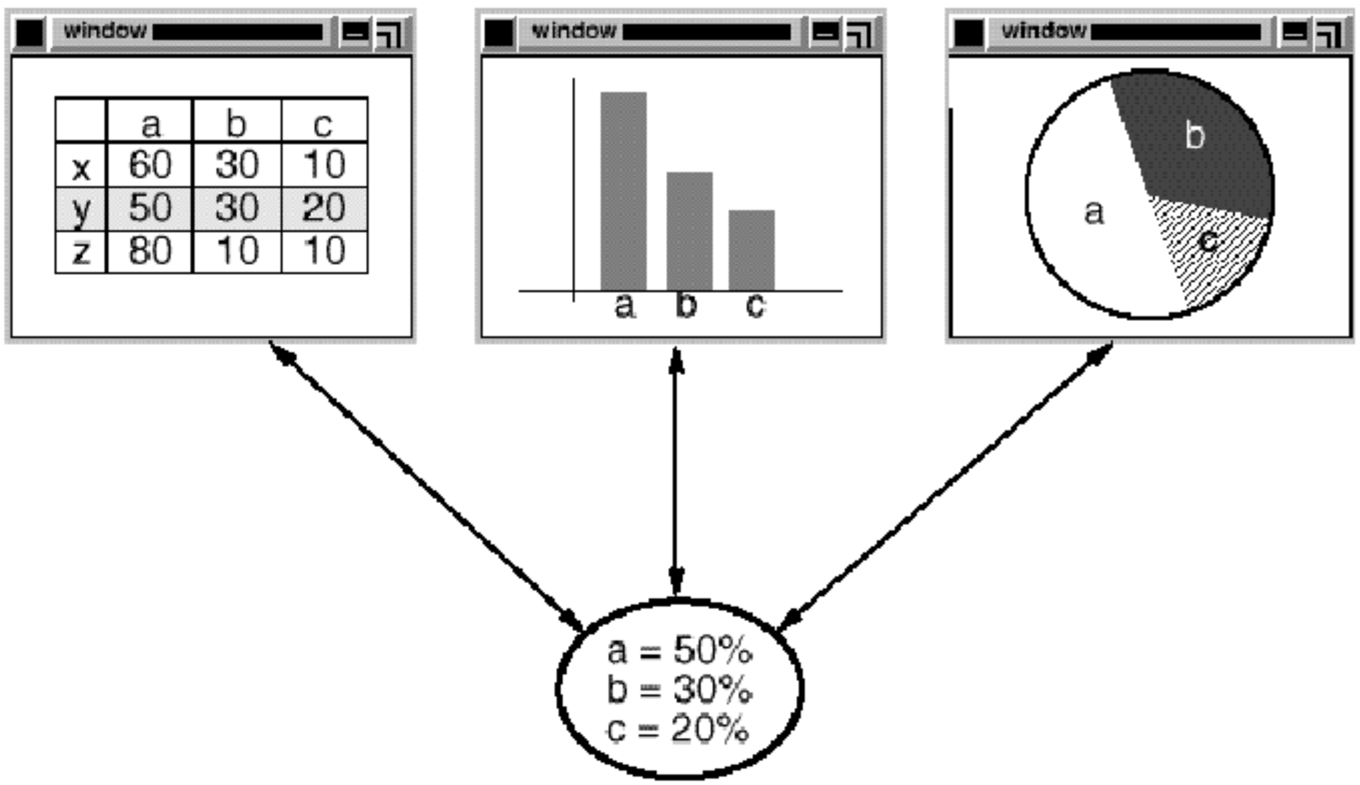


```
class CalculatorModel {
    private int result;
    public void add(int x, int y) {
        result = x + y;
    }
    public void subtract(int x, int y) {
        result = x - y;
    }
    public void multiply(int x, int y) {
        result = x * y;
    }
    public void divide(int x, int y) {
        result = x / y;
    }
    public int getResult() {
        return result;
    }
}
```



# One Model and Many Views

views



|   | a | b | c |
|---|---|---|---|
| x | 60 | 30 | 10 |
| y | 50 | 30 | 20 |
| z | 80 | 10 | 10 |

a = 50%
b = 30%
c = 20%

Model

# More decoupling…

- View should be concerned only with the visual aspects of the application
- What about the responsibilities for interacting with the model to carry out the user requests?
  - Including the decisions about the interface behavior
- The view knows nothing about how this gets done

## MVC – the Controller

- Controller is responsible for interacting with the model to carry out the user request
  - View delegates the request to the controller and the controller *translates* them into actions to be performed by the model
  - Controller *decides* what model operations to call
  - Controller decides how the view should change
- Now view and model are totally decoupled
  - View is concerned only with the visual aspects of the application and delegates to the controller for any *decisions* about the interface behavior

## Separate the concerns

- VIEW : displaying data
- Controller: user event handling, reaction to user interaction
- MODEL: data and application logic

VIEW
Handles the display of information

CONTROLLER
Handles interactions

MODEL
Manages data, state, and business logic

# MVC–Responsibilities

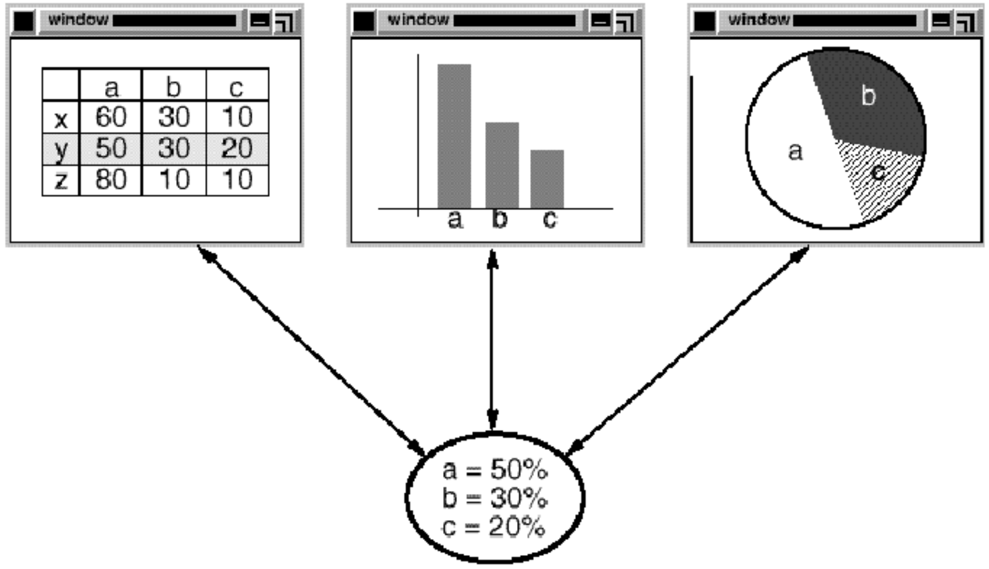
- Model
  - Holds data, state, and application logic
  - Unaware of view and controller classes
- View
  - display model’s data
    - gets the state and data from the model
  - No application logic, no interaction logic –it is dumb
- Controller
  - Interacts with the model to carry out user requests coming from view.
    - Calls the model operations to perform actions

## MVC enables

- Attach multiple presentation for a domain model
  - a GUI, a command line, a web presentation
  - Reuse the domain
- Reuse the visual part for many domains
  - Reuse View
- Ability to change the way a view responds to user inputs
  - change controllers

# Problem1 -- Pattern #1

- A view must ensure that its appearance reflects the state of the model
  - Change the view whenever the model's data changes
  - Remember, the Model **cannot** and **should not** know about the View classes.

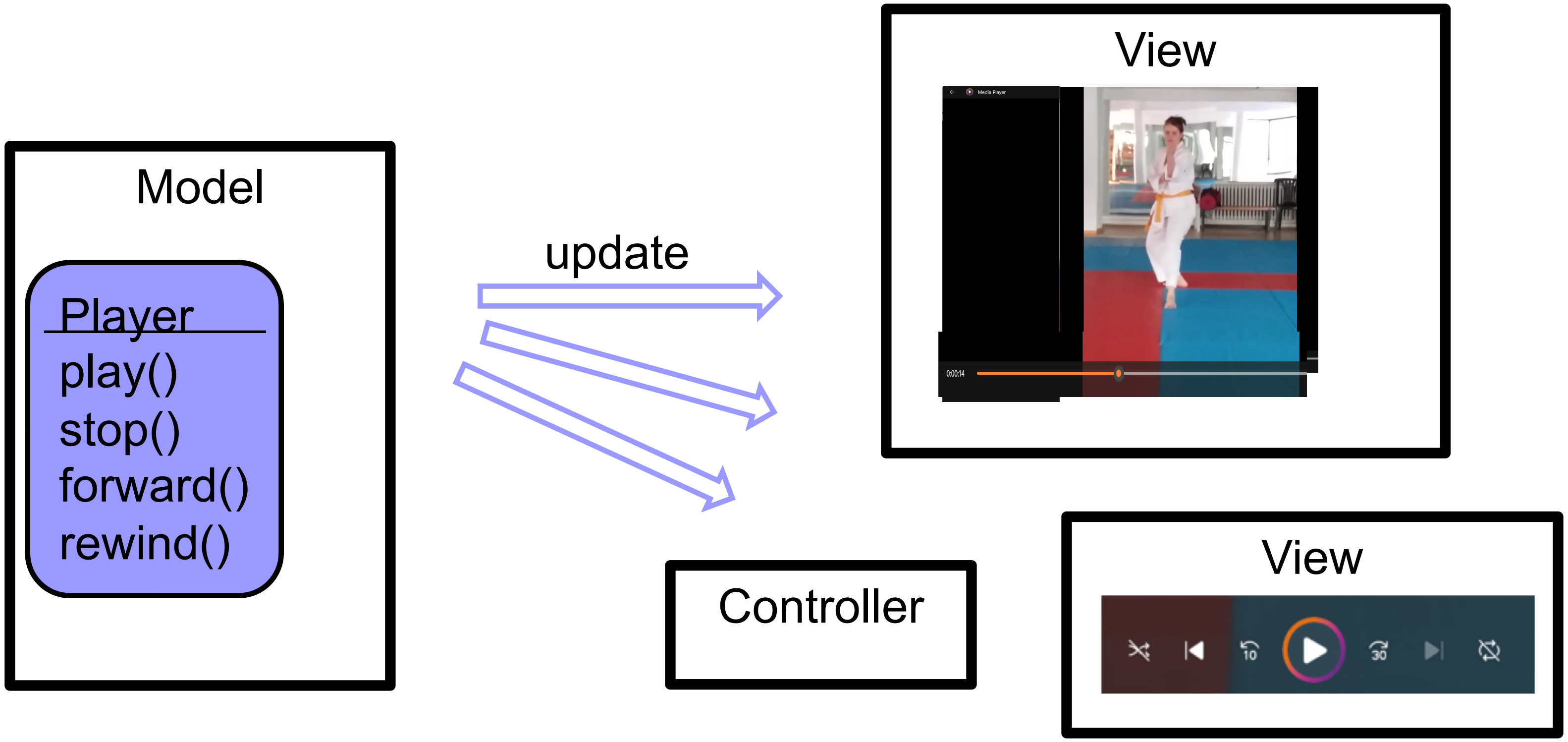


- Ability to attach multiple views and new ones
- Each view gets an opportunity to update its

## Pattern #1

- Pattern name?
  - General problem: Decoupling objects so that changes to one can affect any number of others without requiring the changed object to know details of the others.
- Participants?

# Pattern #1: Observer



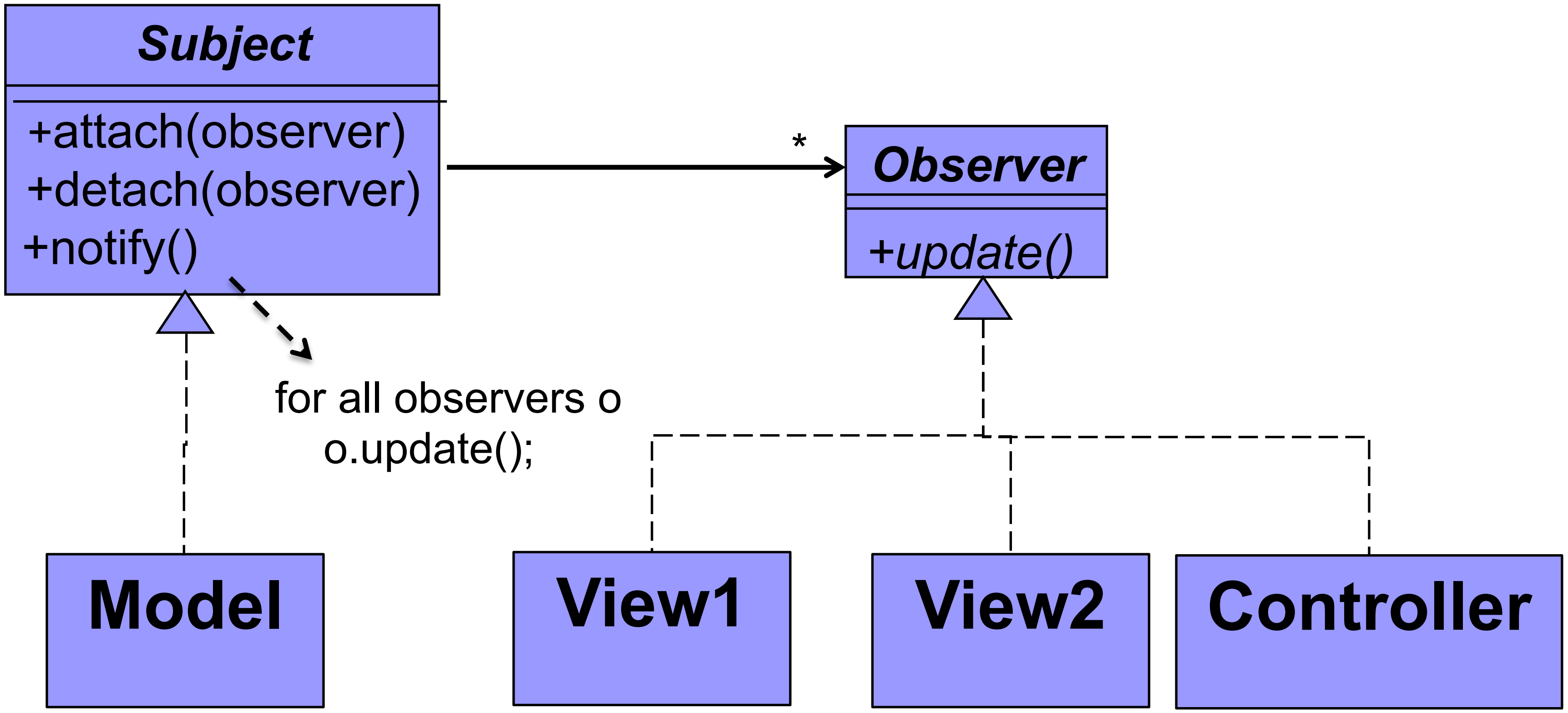
Subject

Observers

## Observer pattern

- Multiple views and controller **observe** the Model
  - View gets the state/data from model
    - Push or pull?
  - Controller observes model so that it can change the view
    - Disable widgets, open a dialog,…
- Model is independent of views and controllers
  - Reusable model
  - Multiple views is possible

# Pattern #1: Observer



# Problem2 --Pattern #2

- We need to change a View's response *behavior* without changing its visual presentation
  - E.g. change respond to a user action, for example, use a pop-up menu instead of a new page
- MVC encapsulates the response mechanism in a separate object: the Controller
  - View (JButton) knows *when* a click happens, but not *what* to do.
  - View delegates it to the response policy: the Controller

## Pattern #2

- A class hierarchy of controllers,
  - making it easy to create a new controller as a variation on an existing one.
- A view uses an instance of a Controller subclass to implement a particular response policy
  - to implement a different response mechanism, simply replace the instance with a different kind of controller.
- Change a view's controller at run-time to change the way it responds to user input.
  - For example, a view can be disabled so that it doesn't accept input simply by giving it a controller that ignores input events.

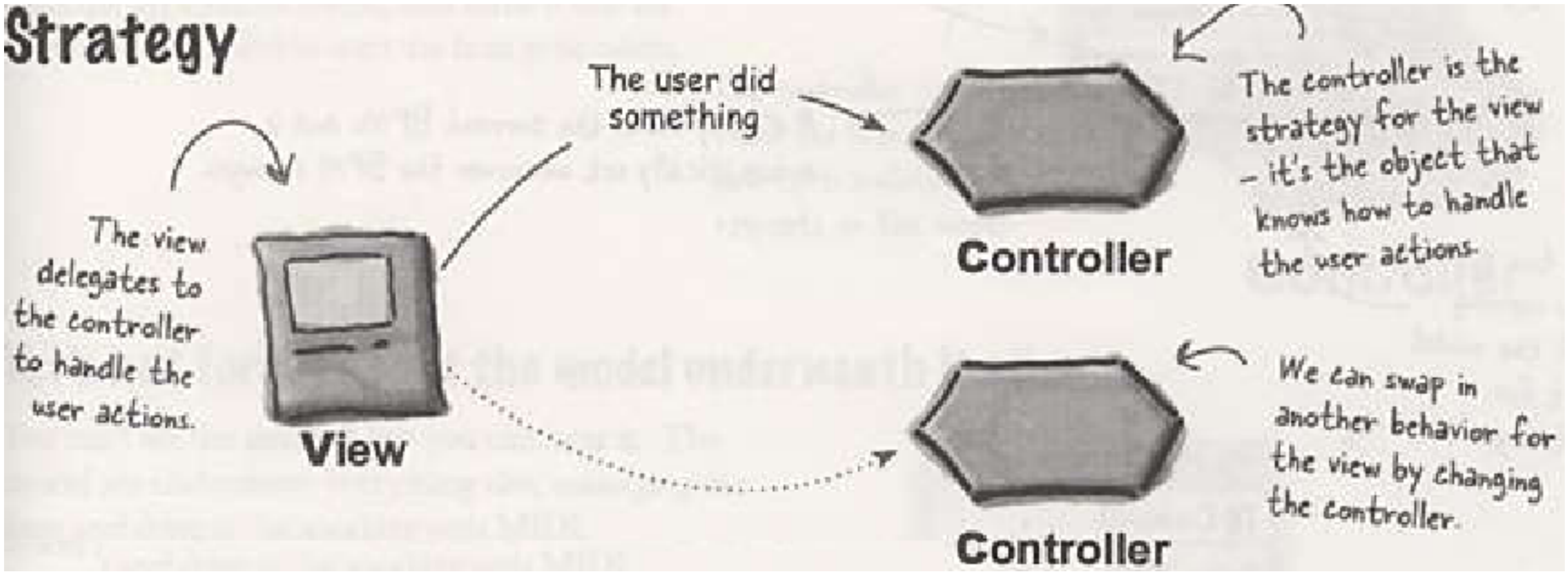
## Pattern #2

- The View-Controller relationship is an example of which design pattern?

- General problem: we want to replace the algorithm either statically or dynamically;
  - we have a lot of variants of the algorithm;
  - or the algorithm has complex data structures that we want to encapsulate.

# Pattern #2 -Strategy

- View is an object configured with a strategy


Strategy
The user did something
The view delegates to the controller to handle the user actions.
View
Controller
The controller is the strategy for the view - it's the object that knows how to handle the user actions.
We can swap in another behavior for the view by changing the controller.
Controller

We can swap in a *different* Controller to change the program's behavior

without touching the View's code.

Figure from Head First Design Patterns

# Pattern#2 - alternative

- In the classical MVC, the view is **directly** accessing the model to get data
  - Does **<u>not</u>** happen in a mediator behavior

- **MVP**: model-view-presenter
  - **Presenter acts as Mediator**
  - Model data change notifies Presenter, then Presenter reflects the change in the view.
    - View does not access model at all.
  - View sends user interactions to Presenter, then Presenter invokes the model accordingly.

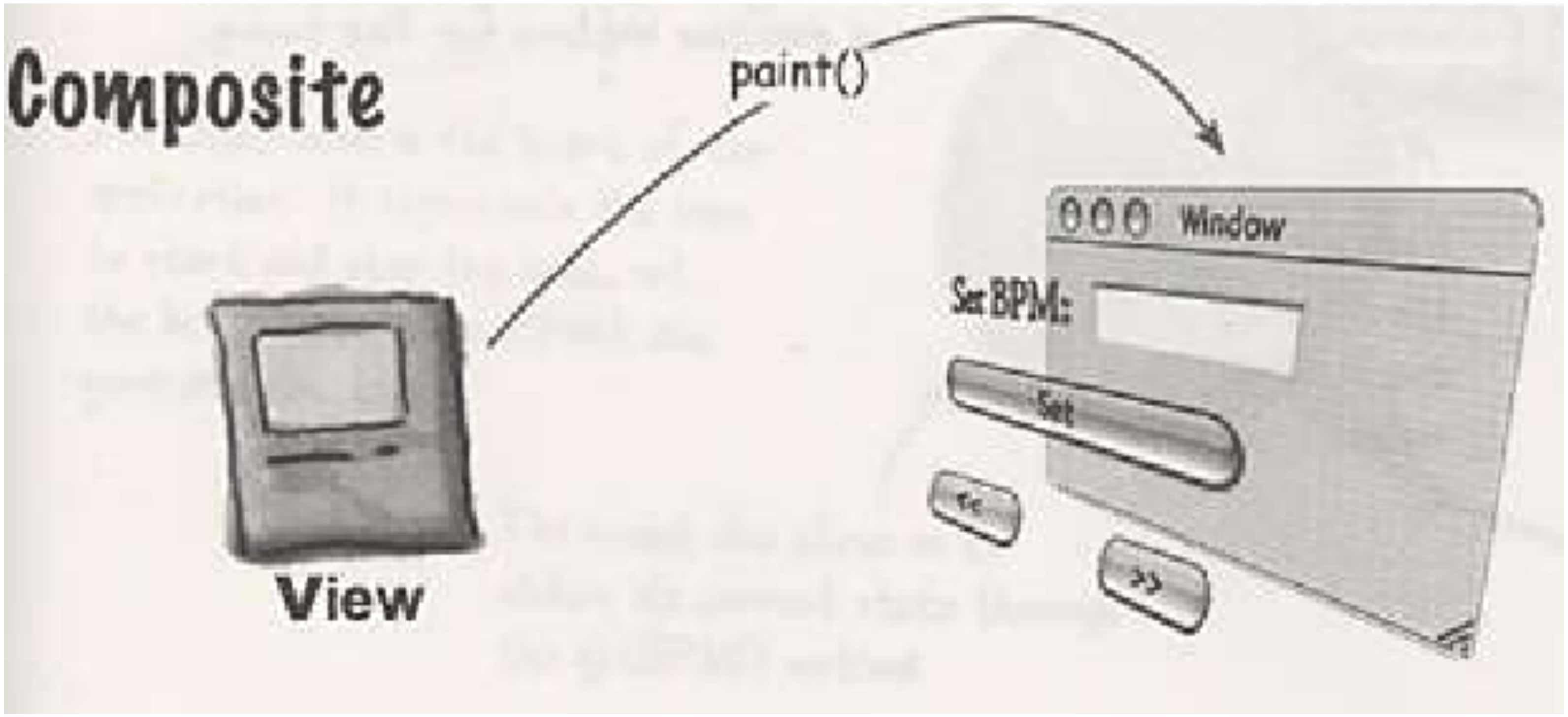
# MVC – The View

- Views can be nested
  - E.g. a control panel of buttons containing nested button views.
  - E.g. The user interface for an object inspector can consist of nested views that may be reused in a debugger.
- This is the standard now with GUI frameworks.
  - It is the norm that we don’t realize the pattern anymore ☺
- When the controller tells the view to update, it only needs to tell the top view component
  - E.g When a Frame is told to update, it tells its children, like Panel`s, to update, and so on.

## Pattern #3

- MVC supports nested views
  - A composite view can be used wherever a view can be used, but it also contains and manages nested views.
- When the controller tells the view to update, it only needs to tell the top view component
- The pattern is ….
  - <u>general problem:</u> We want to group objects and treat the group like an individual object and give a part-whole hierarchy.

# Pattern #3



The paint() or update() travels down the tree, and each object knows how to draw itself.

Figure from Head First Design Patterns

## Patterns in MVC

- Model
  - Observer: model notifies views and controllers
- View
  - Composite: view elements in a hierarchy
- Controller
  - Strategy: controller is the action strategy of the view
  - Adapter also comes along (adapt a new model to an existing view & controller)

# Collaborations in MVC

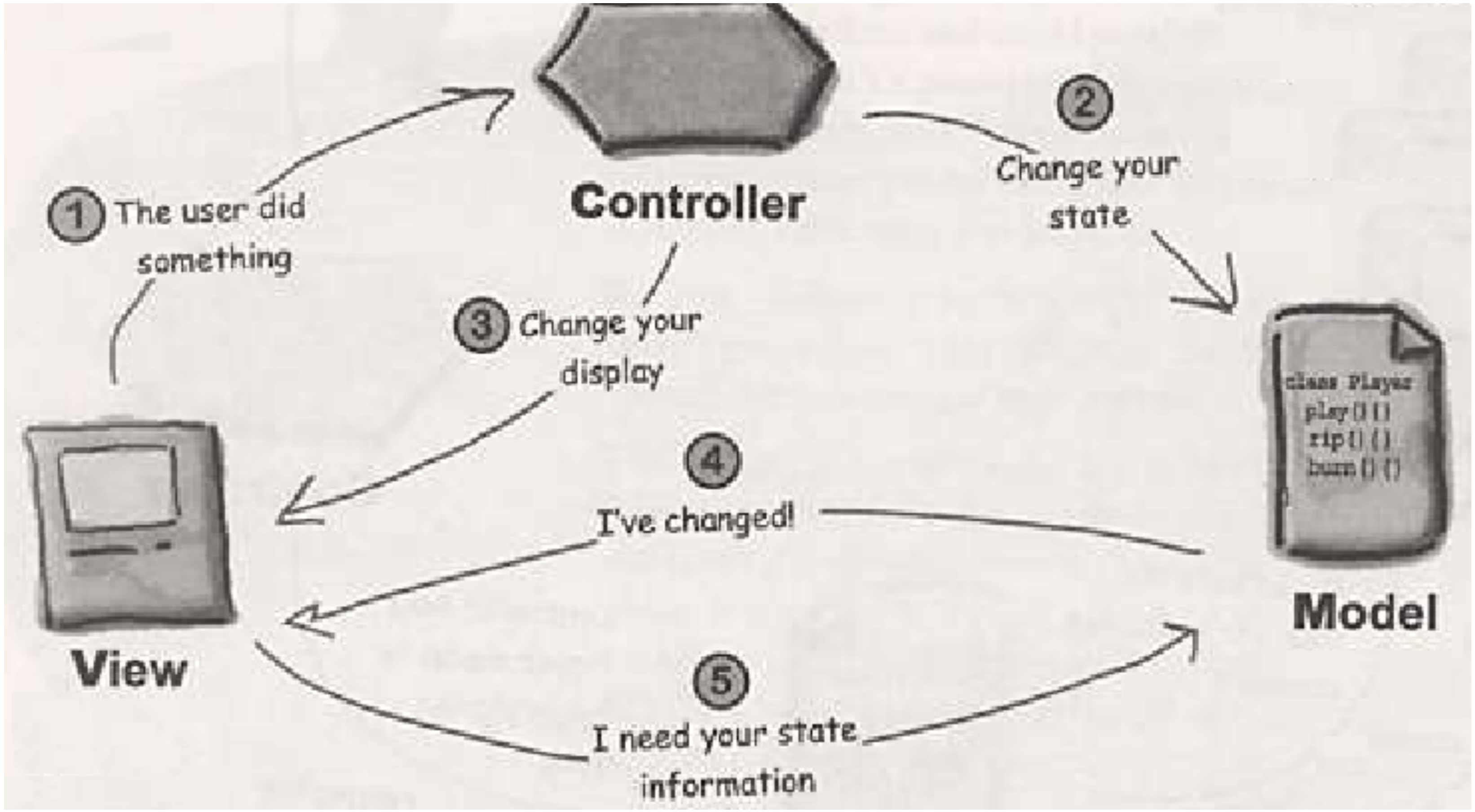


Figure from Head First Design Patterns

# Passive Model Collaboration

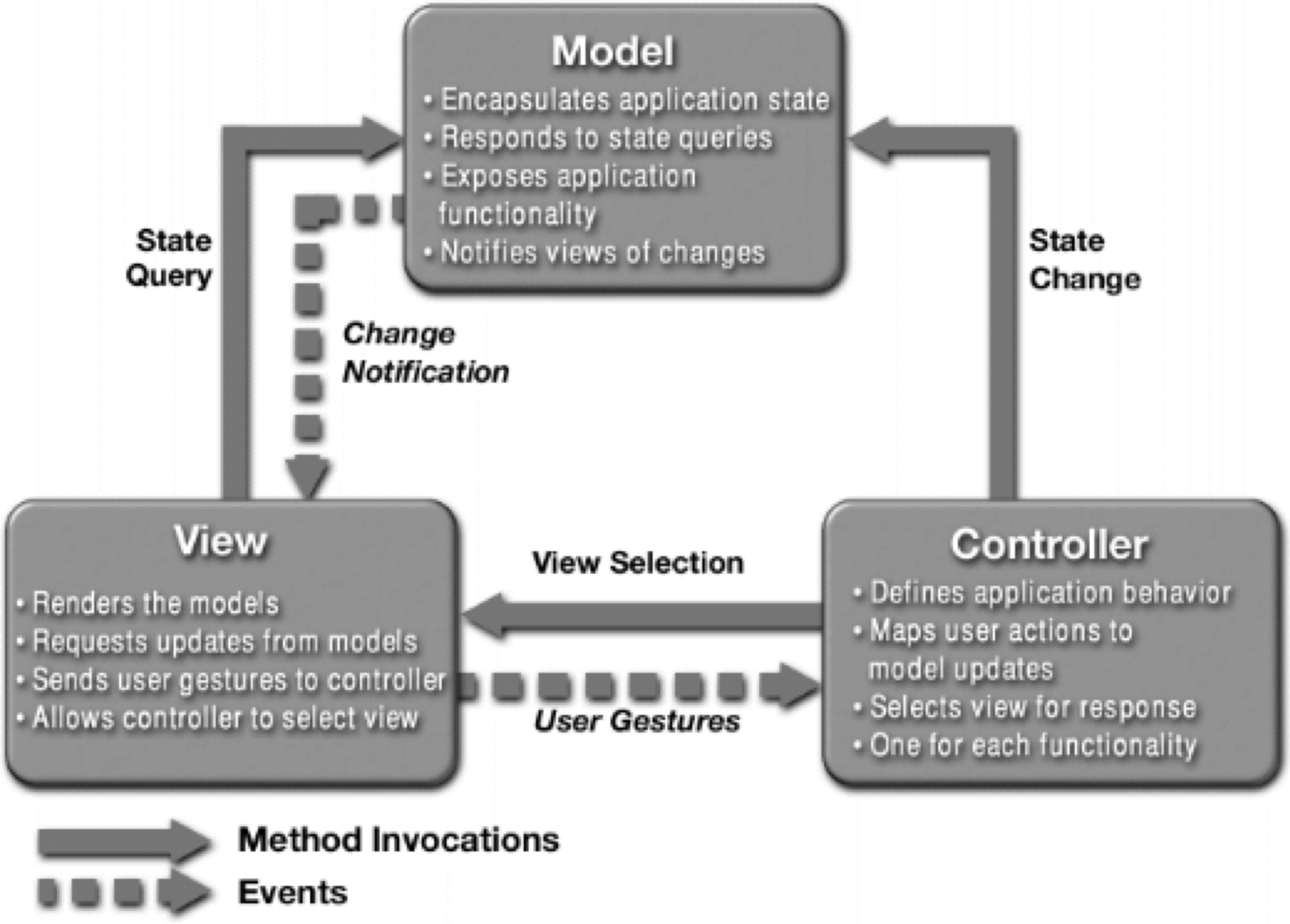

Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
Change Notification
State Change
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• One for each functionality
Method Invocations
Events

Figure from http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/app-arch/app-arch2.html

## Variations

- MVC is an architectural pattern for GUI based software
- Model2, or MVC2 for web development
  - Servlet as the controller
  - JSP is the View producing HTML
- MVP: Presenter acts as Mediator
  - When model changes, presenter updates the view
  - Presenter implements the UI logic
- MVVP: Model-View-ViewModel
  - "Passive Model" is its direct ancestor.
  - ViewModel is the state of the View.
  - Changes in the ViewModel automatically update the View, and vice versa.
    - Two-way data binding between View and ViewModel
    - https://learn.microsoft.com/en-us/dotnet/architecture/maui/mvvm

# With MVC

- **Separation of Concerns**: The application is divided into three components: Model, View, and Controller.
- **Easier to Maintain**: Changes in one component do not affect the others.
- **Improved Testability**: Each component can be tested independently.