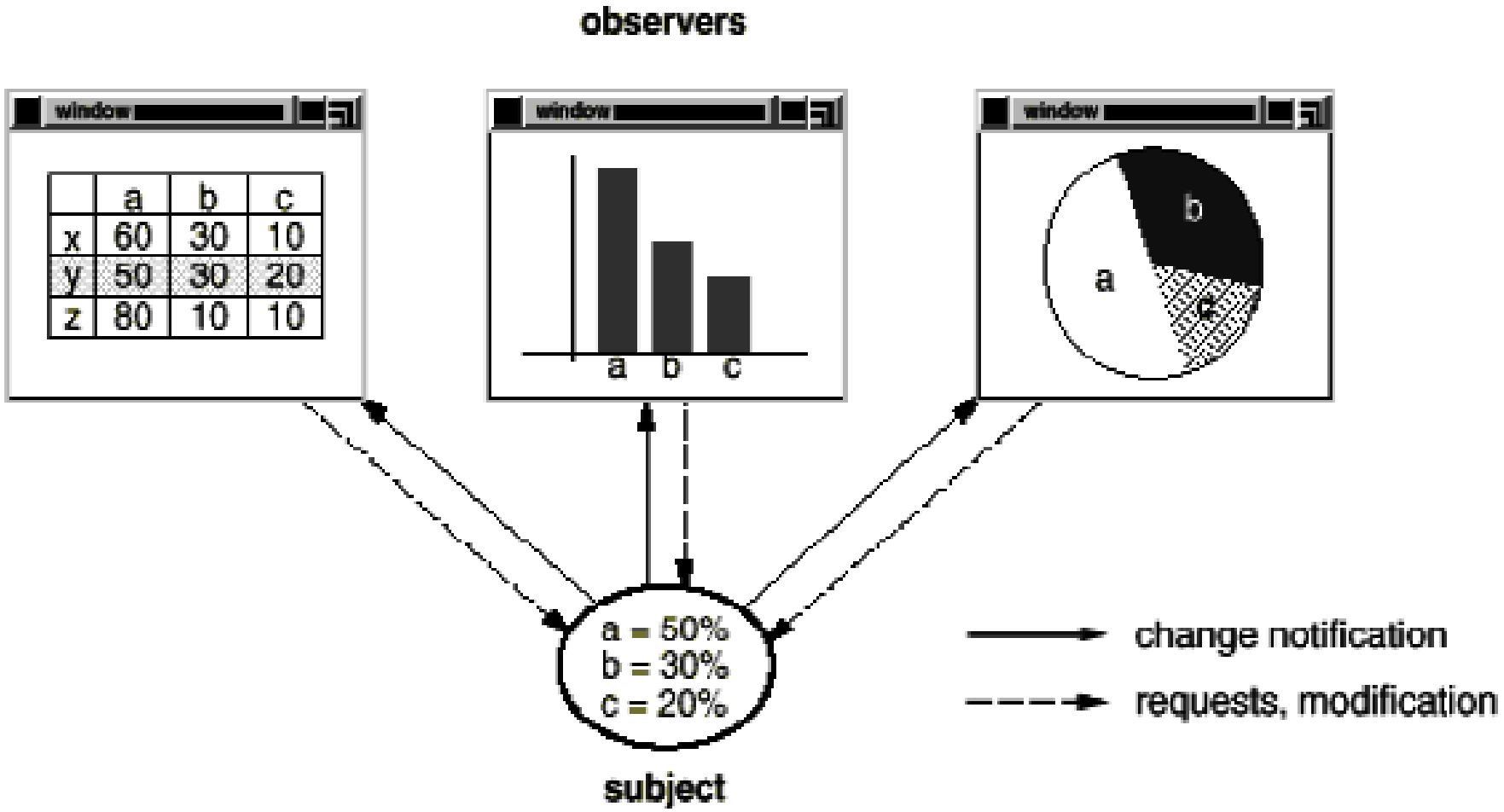
Tecniche di Progettazione: Design Patterns

GoF: MVC e Observer

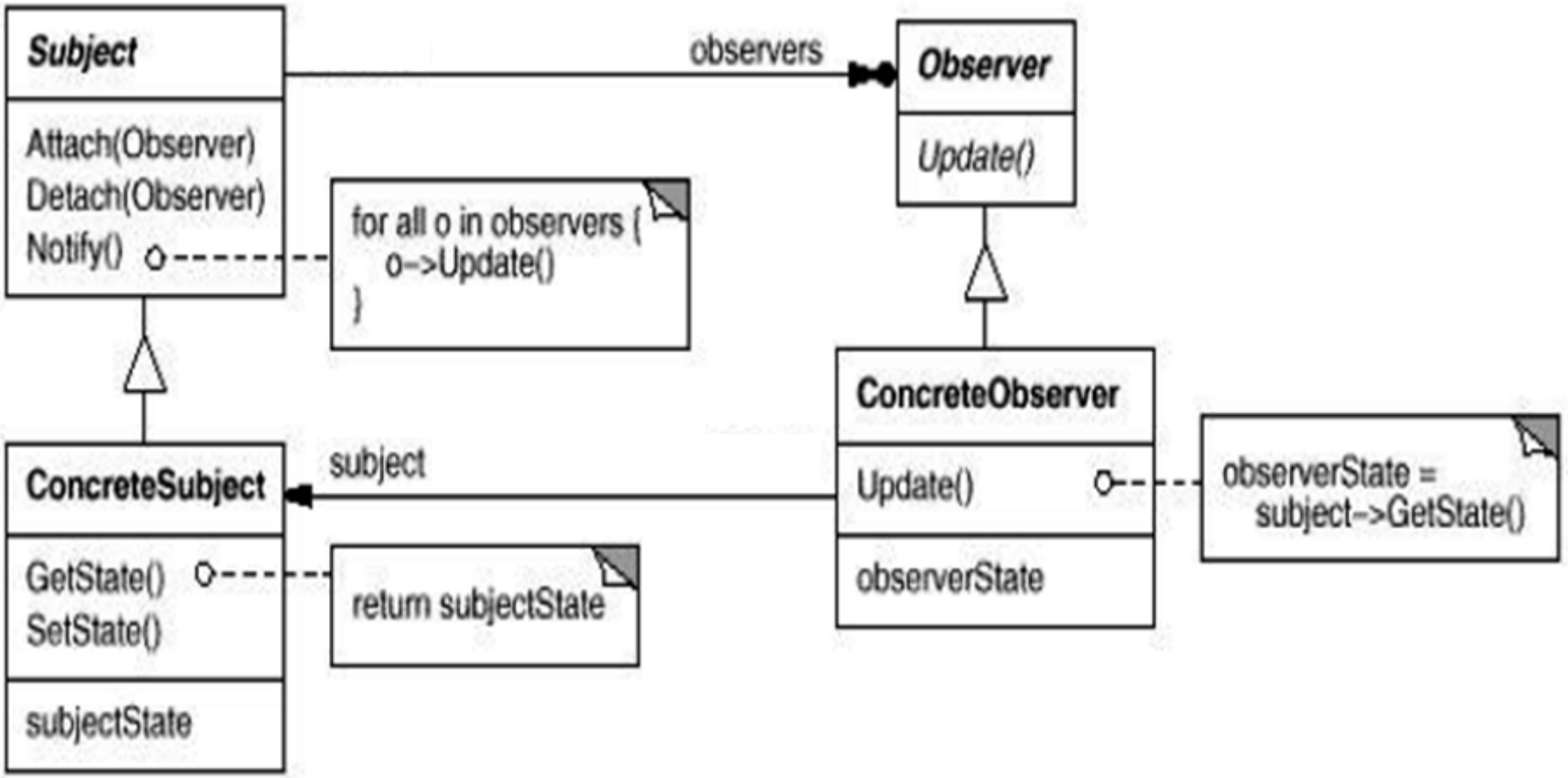
The Observer Pattern

- Intent
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- AKA
  - Dependents, Publish-Subscribe, Model-View
- Motivation
  - The need to maintain consistency between related objects without making classes tightly coupled

Example


observers
window
window
window
a b c
x 60 30 10
y 50 30 20
z 80 10 10
b
a
c
a = 50%
b = 30%
c = 20%
subject
change notification
requests, modification

Structure


Subject
Attach(Observer)
Detach(Observer)
Notify()
observers
Observer
Update()
for all o in observers {
o->Update()
}
ConcreteObserver
Update()
observerState
observerState =
subject->GetState()
ConcreteSubject
GetState()
SetState()
subjectState
subject
return subjectState

MVC

- Model view controller
  - More architectural than design pattern
  - How the structure changes?
  - What is the controller for?

Role of the controller

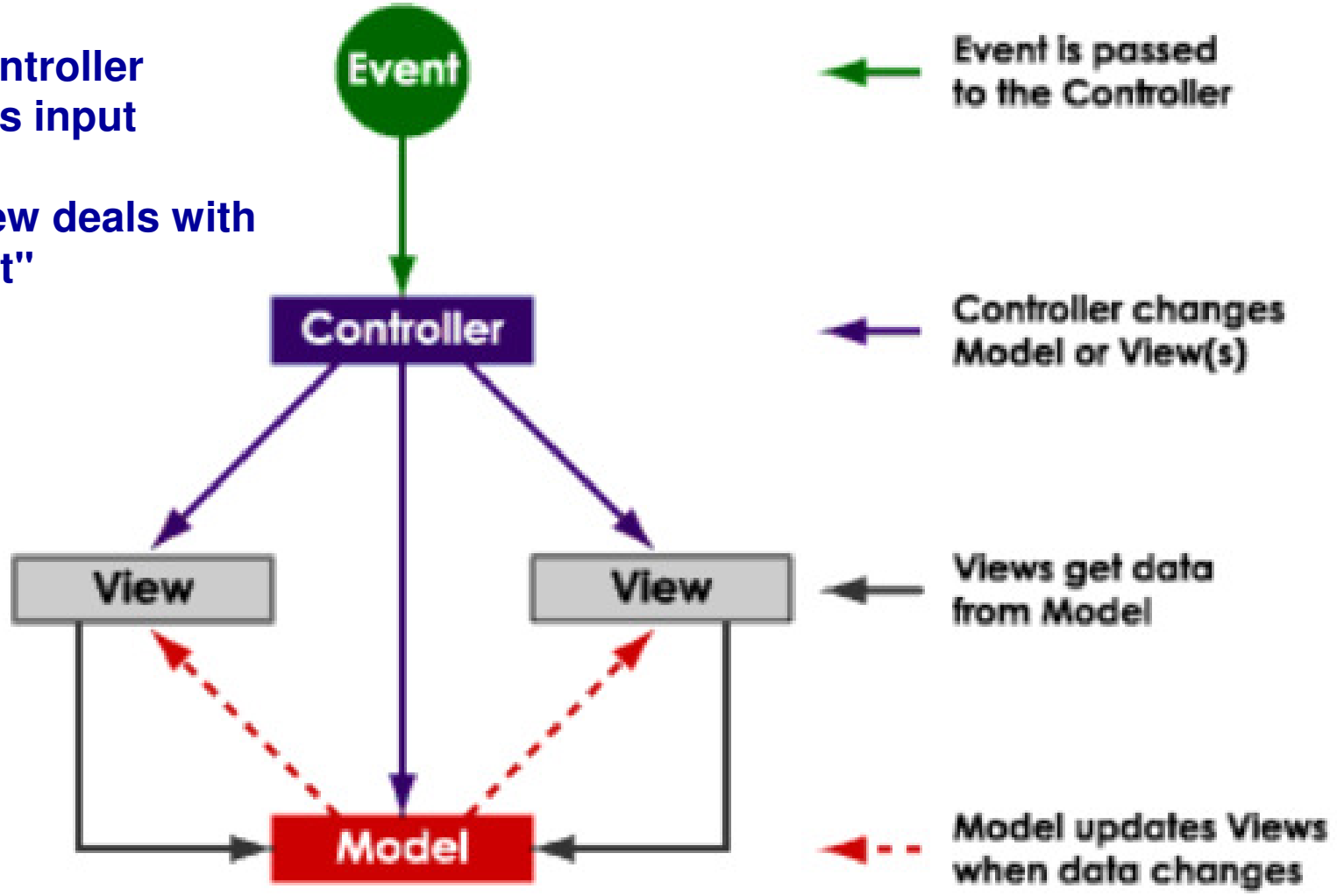
- In MVC, the controller is a strategy for handling events.
- The events come directly to it, rather than to the view.
- The controller changes the model, but it is not notified by the model.
- When the user presses a key or moves the mouse, the controller receives the event. It checks with the view to map mouse locations into model coordinates, then interacts directly with the model.
- If it changes the model then the model notifies all dependents (observers), which notifies the view, which redisplays.

Role of the controller
In a conversation form

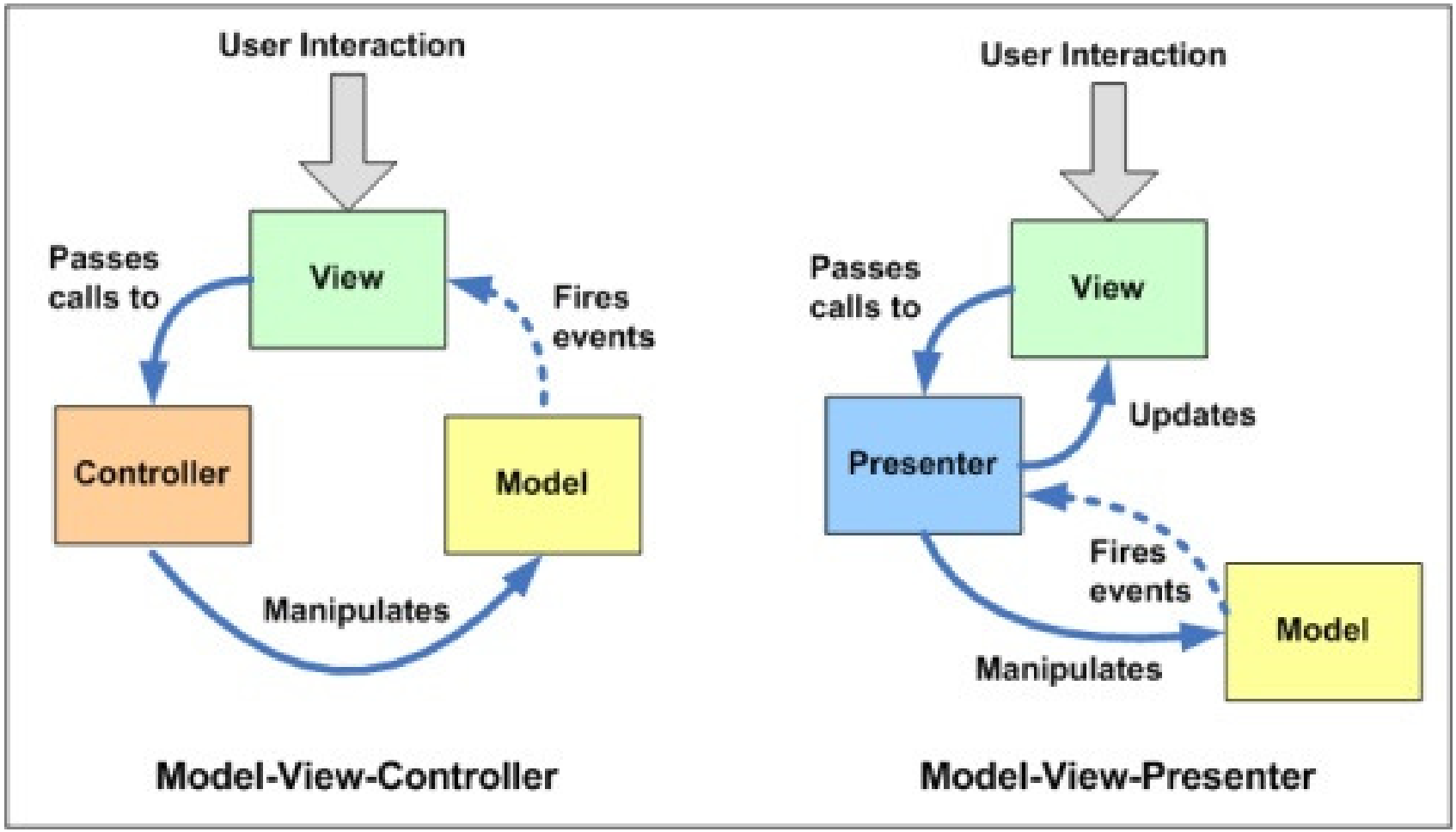
- **View**: "Hey, controller, the user just told me he wants item 4 deleted.“
- **Controller**: "Hmm, having checked his credentials, he is allowed to do that... Hey, model, I want you to get item 4 and do whatever you do to delete it.“
- **Model**: "Item 4... got it. It's deleted. Back to you, Controller.“
- **Controller**: "Here, I'll collect the new set of data. Back to you, view.“
- **View**: "Cool, I'll show the new set to the user now."

Model View Controller with a different perspective

The controller handles input events
The view deals with "output"

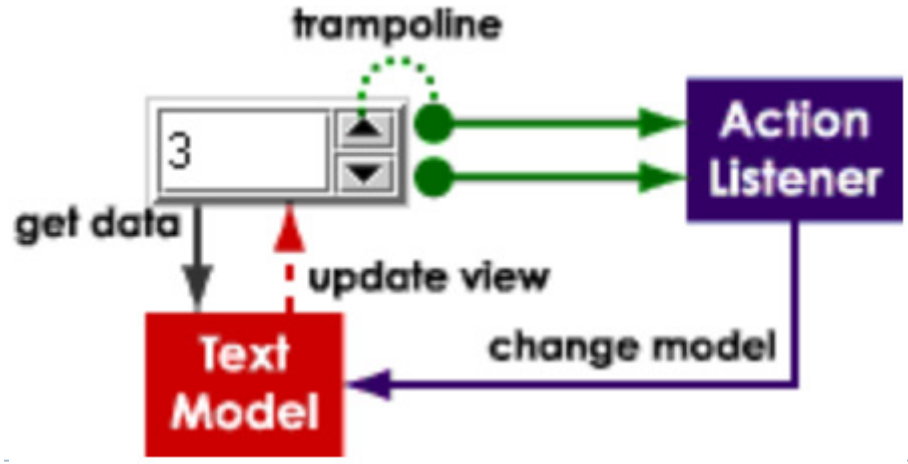


Yet another picture



MVC: counter

- Counter component which consists of a text field and two arrow buttons that can be used to increment or decrement a numeric value shown in the text field.
- The counter's data is held in a model that is *shared* with the text field. The text field provides a view of the counter's current value. Each button is an event source, that spawns an action event every time it is clicked. The buttons can be hooked up to trampolines that receive action events, and route them to an action listener that eventually handles that event. Recall that a trampoline is a predefined action listener that simply delegates action handling to another listener.



CONTROLLER

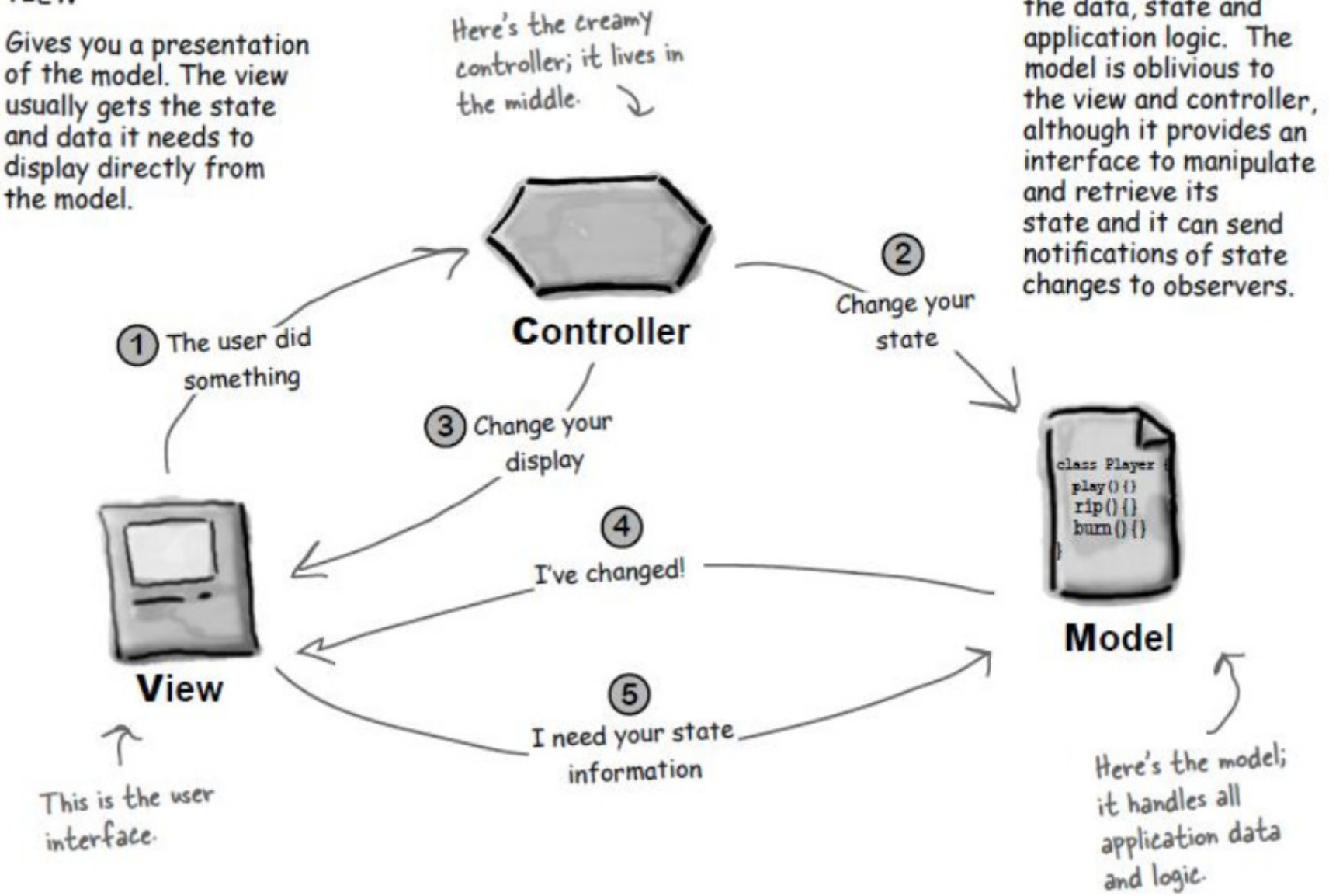
Takes user input and figures out what it means to the model.

MODEL

The model holds all the data, state and application logic. The model is oblivious to 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.

VIEW

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



Brief Summary

1. You're the user — you interact with the view.
   - The controller takes your actions and interprets them. If you click on a button, it's the controller's job to figure out what that means and how the model should be manipulated based on that action.
2. The controller asks the model to change its state.
   - The controller takes your actions and interprets them. If you click on a button, it's the controller's job to figure out what that means and how the model should be manipulated based on that action
3. The controller may also ask the view to change.
   - When the controller receives an action from the view, it may need to tell the view to change as a result. For example, the controller could enable or disable certain buttons or menu items in the interface.

Brief Summary

4. The model notifies the view when its state has changed.
   - When something changes in the model, based either on some action you took (like clicking a button) or some other internal change (like the next song in the playlist has started), the model notifies the view that its state has changed.
5. The view asks the model for state.
   - The view gets the state it displays directly from the model. For instance, when the model notifies the view that a new song has started playing, the view requests the song name from the model and displays it. The view might also ask the model for state as the result of the controller requesting some change in the view.

MVC1: CounterGui

```
/**
* Class CounterGui demonstrates having the model and view in the same class
*/

import java.awt.*;
import java.awt.event.*;

public class CounterGui extends Frame
{
    //the counter (the model!)
    private int counter = 0;

    //the view/
    private TextField tf = new TextField(10);

//…… see project
```

MVC2: CounterView & Counter

- This example shows the model and the view in separate classes. First the view class:
- /**
- * Class CounterView demonstrates having the model and view
- * in the separate classes. This class is just the view.
- */
- public class CounterView extends Frame {
- // The view.
- private TextField tf = new TextField(10);
- // A reference to our associated model.
- private Counter counter;

MVC2: CounterView & Counter

- public CounterView(String title, Counter c) {
- super(title);
- counter = c;
- Panel tfPanel = new Panel();
- tf.setText(counter.getCount()+ "");
- tfPanel.add(tf);
- add("North",tfPanel);
- Panel buttonPanel = new Panel();
- Button incButton = new Button("Increment");
- incButton.addActionListener(new ActionListener() {
  - public void actionPerformed(ActionEvent e) {counter.incCount();
    - tf.setText(counter.getCount() + "");

  } });
- buttonPanel.add(incButton);

MVC3: applying Observer

```
/**
* Class ObservableCounter implements a simple observable
* counter model.
*/
public class ObservableCounter extends Observable {

/**
* Class ObservableCounterView demonstrates having the model
* and view in the separate classes. This class is just the view.
*/
 ….. counter.addObserver(new Observer() {
                public void update(Observable src, Object obj) { …….
```