The Model View Controller: a Composed Pattern.

Toni Sellarès
Universitat de Girona

The Model-View-Controller (MVC) Pattern

The MVC pattern separates the modeling of the domain, the presentation, and the actions based on user input into three separate classes:

Model:

- Contains data.
- Encapsulates application state.
- Responds to state queries/updates.
- Exposes application functionality.

View:

- Renders the model.
- Allows Controller to select view.
- Sends user input to the Controller.

Controller:

- Defines application behavior.
- Maps user actions to Model updates.

Views and Controllers conform the User Interface.

A change-propagation mechanism ensures consistency between the user interface and the model.

In general all GUI toolkits are developed according to MVC principles.

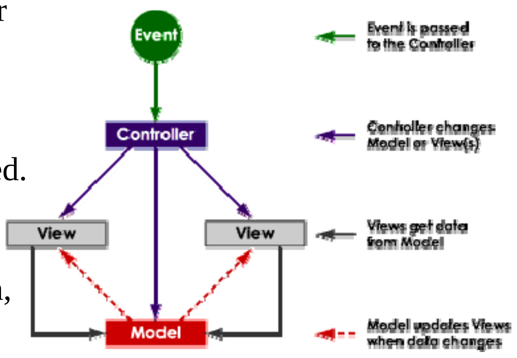
MVC: Flow

The user manipulates a view and, as a result, an event is generated.

Events typically cause a controller to change a model, or view, or both.

Whenever a controller changes a model’s data or properties, all dependent views are automatically updated.

Similarly, whenever a controller changes a view, for example, by revealing areas that were previously hidden, the view gets data from the underlying model to refresh itself.


Event
Controller
View
View
Model
Event is passed to the Controller
Controller changes Model or View(s)
Views get data from Model
Model updates Views when data changes

Observe that:

- Both the view and the controller depend on the model.
- The model depends on neither the view nor the controller.
- The separation between view and controller is secondary in many rich-client applications, and, in fact, many user interface frameworks implement the roles as one object.

MVC: Benefits

- Supports multiple views. Because the view is separated from the model and there is no direct dependency from the model to the view, the user interface can display multiple views of the same data at the same time.
- Accommodates change. User interface requirements tend to change more rapidly than business rules. Because the model does not depend on the views, adding new types of views to the system generally does not affect the model. As a result, the scope of change is confined to the view.
- Complexity. The MVC pattern introduces new levels of indirection and therefore increases the complexity of the solution slightly. It also increases the event-driven nature of the user-interface code, which can become more difficult to debug.
- Cost of frequent updates. Decoupling the model from the view does not mean that developers of the model can ignore the nature of the views. If the model undergoes frequent changes, it could flood the views with update requests. As a result, the view may fall behind update requests. Therefore, it is important to keep the view in mind when coding the model.

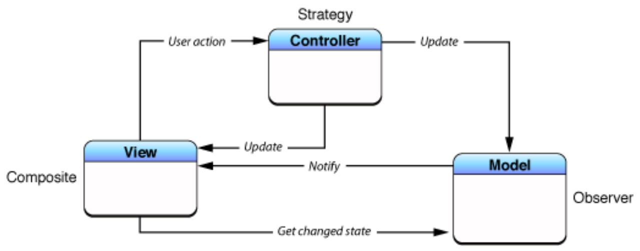
Looking at MVC through other patterns

MVC is set of patterns together in the same design:

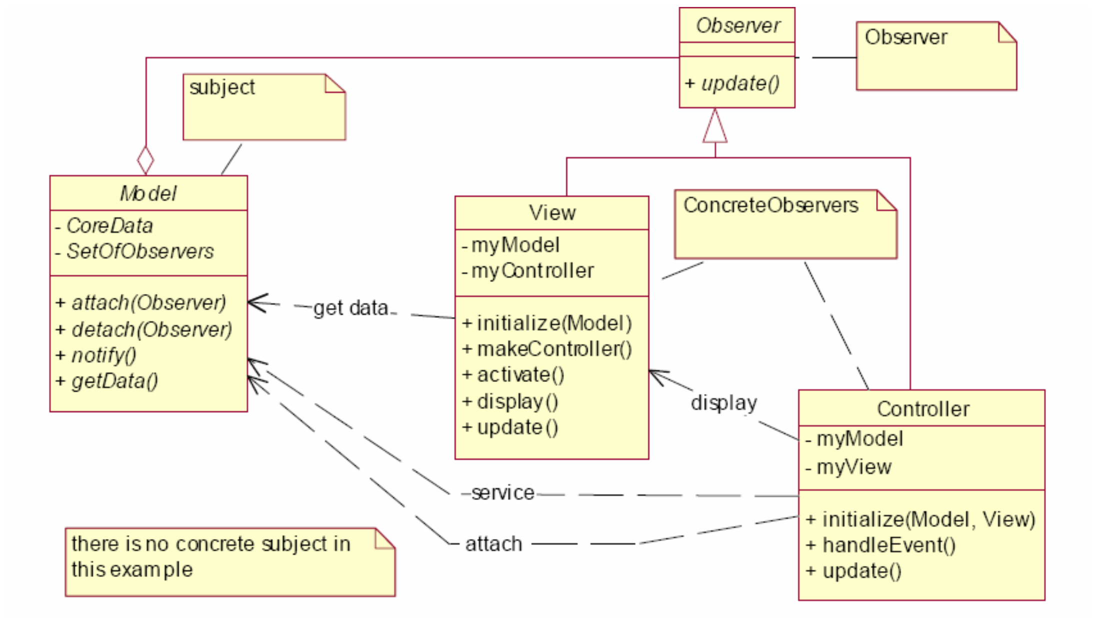
- *Model* uses Observer to keep views and controllers updated on the latest state changes.

- *View* and *Controller* implement Strategy Pattern Controller is the behavior of the view and can be easily exchanged with another controller if you want different behavior.

- *View* also uses a pattern internally to manage the windows buttons and other components of the display: the Composite Pattern



The Observer Pattern in the MVC



Meet the MVC

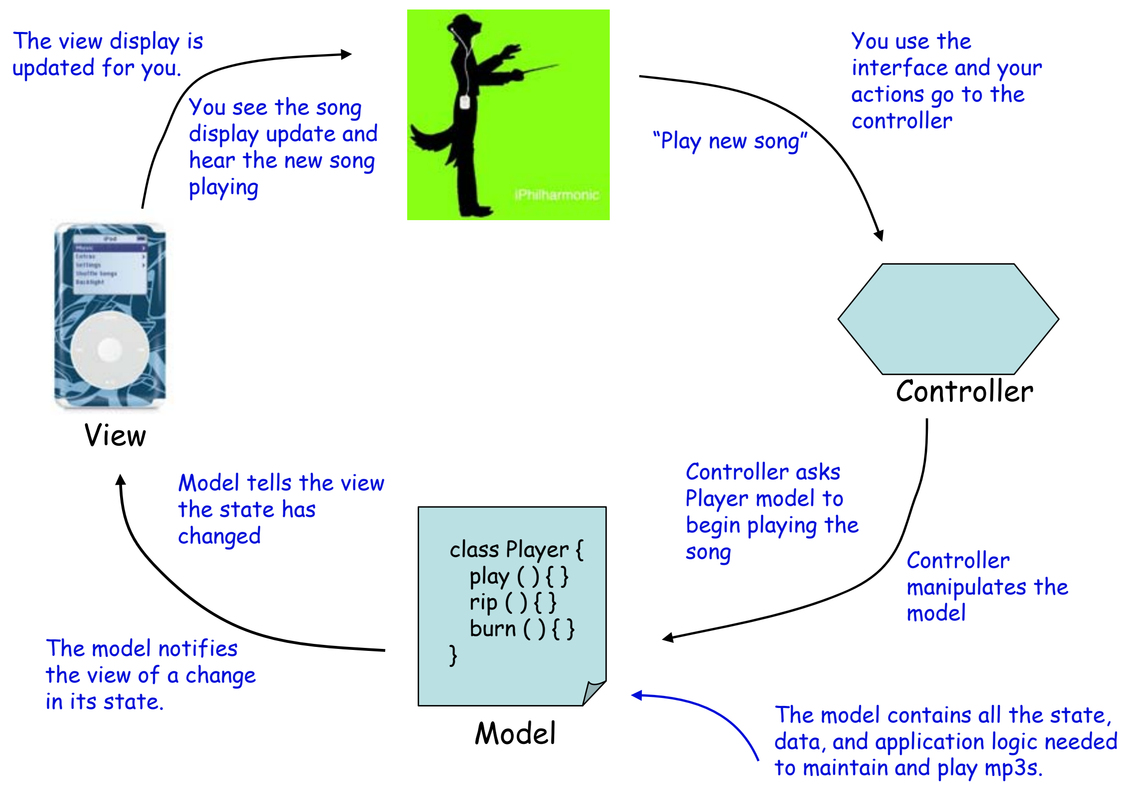
Picture your favorite MP3 player:

– You can use its interface to add new songs, manage play lists and rename tracks.

– The player maintains:

- Database of all your songs along with associated names and data.
- Plays the song.
- Updates the interface constantly with current song, running time and so on

Meet the MVC


The view display is updated for you.
You see the song display update and hear the new song playing
iPhilharmonic
"Play new song"
You use the interface and your actions go to the controller
View
Controller
Model tells the view the state has changed
Controller asks Player model to begin playing the song
Controller manipulates the model
class Player {
play ( ) { }
rip ( ) { }
burn ( ) { }
}
The model notifies the view of a change in its state.
Model
The model contains all the state, data, and application logic needed to maintain and play mp3s.

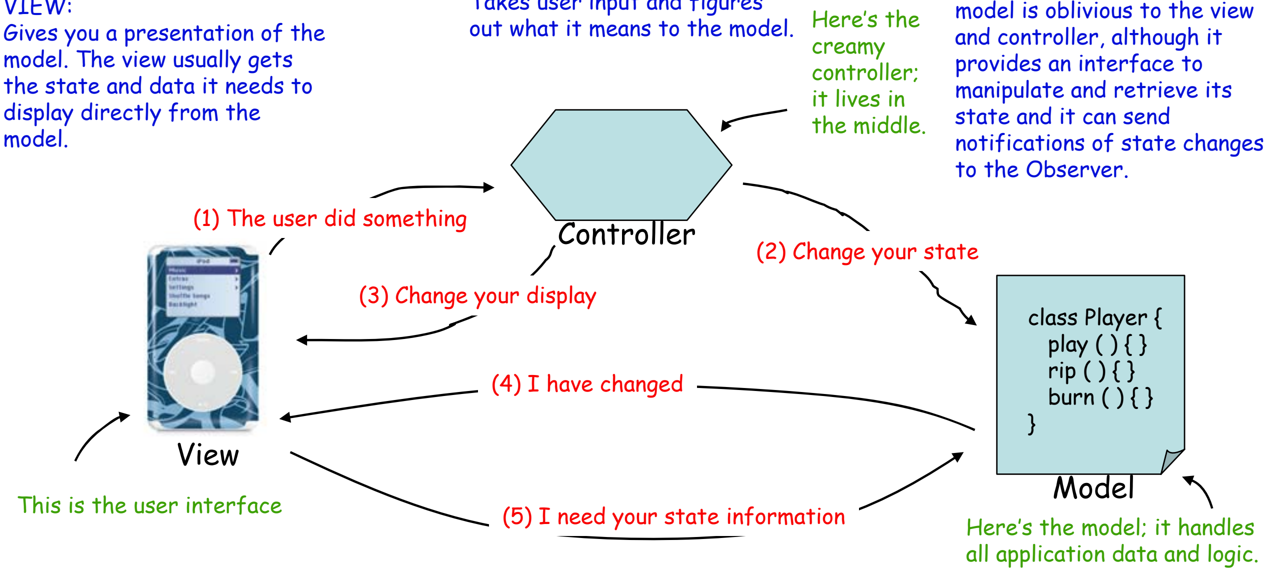
A Closer Look….

Lets see the nitty gritty details of how this MVC pattern works.

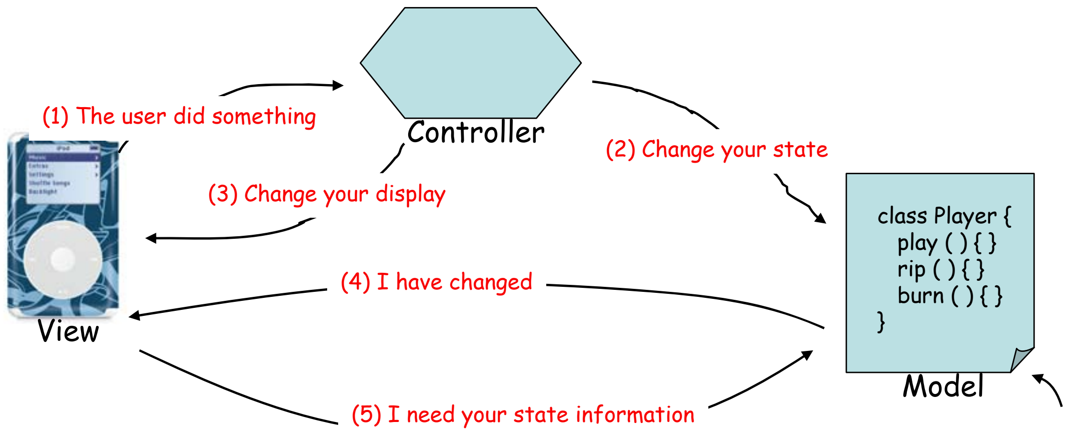
VIEW:
Gives you a presentation of the model. The view usually gets the state and data it needs to display directly from the model.

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

MODEL:
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 the Observer.


Here's the creamy controller; it lives in the middle.
Controller
(1) The user did something
(2) Change your state
(3) Change your display
(4) I have changed
(5) I need your state information
View
This is the user interface
class Player {
play ( ) { }
rip ( ) { }
burn ( ) { }
}
Model
Here's the model; it handles all application data and logic.

Observer



Observer: The model implements the Observer Pattern to keep the interested objects updated when the state changes occur. Using the Observer Pattern keeps the model completely independent of the views and the controllers. Its allows us to use different views with the same model, or even multiple views at once.

Observer

(1) You are the user: you interact with the view:

- The view is your window to the model. When you do something to the view (like click the Play button), then the view tells the controller what you did. It's the controller's job to handle that.

(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.

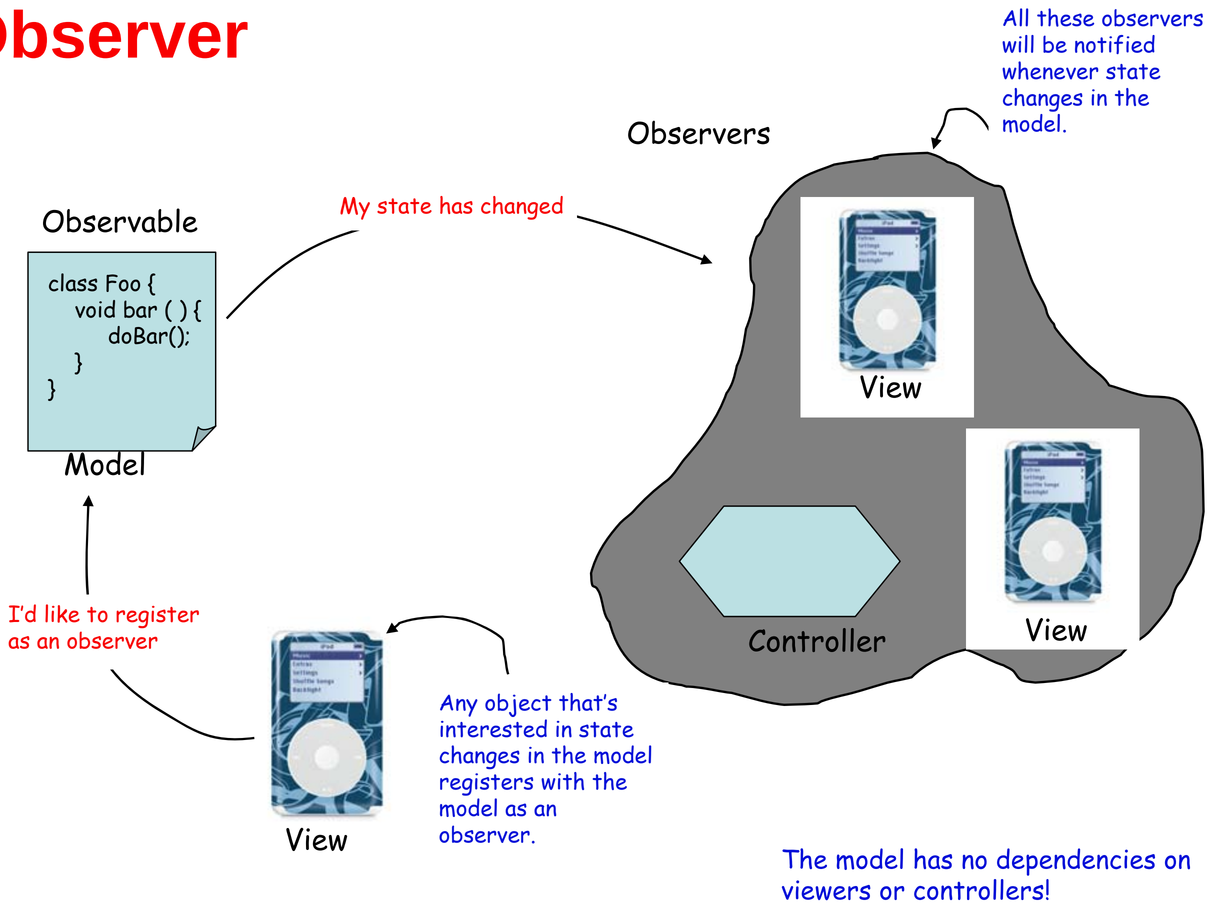
(4) The model notifies the view when its state has changed:

- When something changes in the model, based on either 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.

Observer

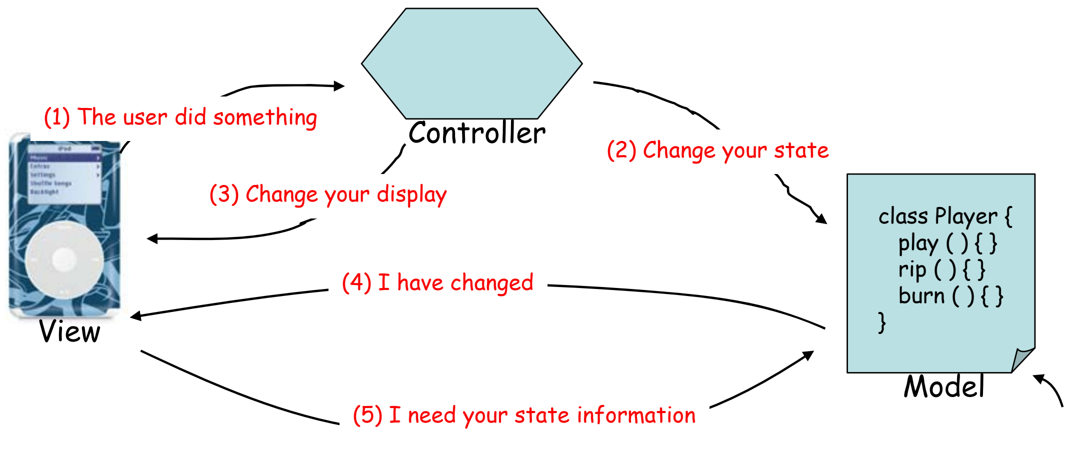


All these observers will be notified whenever state changes in the model.
Observers
Observable
My state has changed
class Foo {
void bar ( ) {
doBar();
}
}
Model
View
View
Controller
View
I'd like to register as an observer
Any object that's interested in state changes in the model registers with the model as an observer.

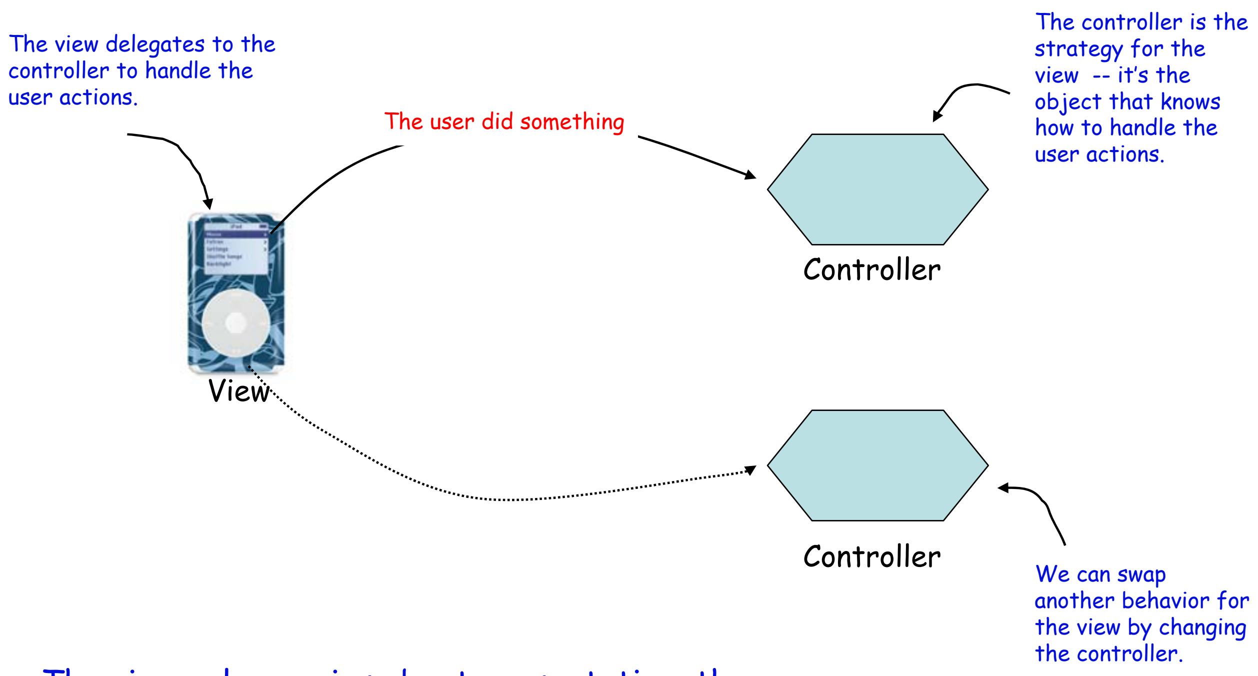
The model has no dependencies on viewers or controllers!

Strategy

The View and the Controller implement the classic *Strategy* Pattern: the view is an object that is configured with a strategy. The view is concerned only with the visual aspects of the application, and delegates to the controller for any decisions about the interface behavior. Using the Strategy pattern also keeps the view decoupled from the model, because it is the controller that is responsible for interacting with the model to carry out user requests. The view knows nothing about how it gets done.

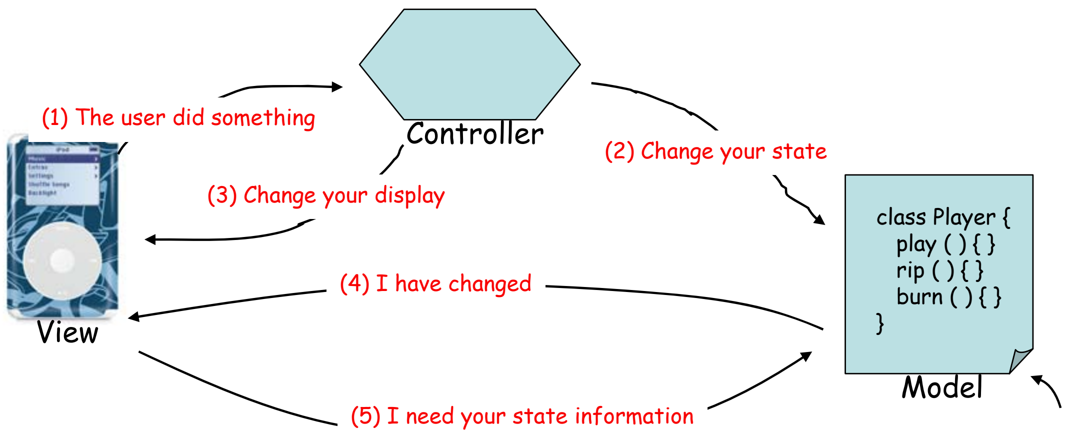


Strategy


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

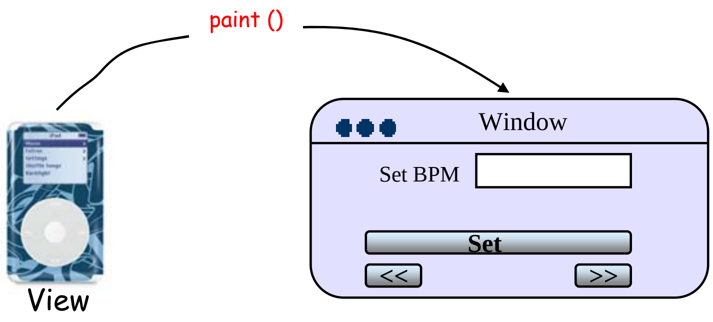
The view only worries about presentation, the controller worries about translating user input to actions on the model.

Composite



The display consists of a nested set of windows, panels, buttons, text labels and so on. Each display component is a composite (like a window) or a leaf (button). When the controller tells the view to update, it only has to tell the top view component, and the Composite takes care of the rest.

Composite



The view is a composite of GUI components (labels, buttons, text entry, etc.). The top level component contains other components, which contain other components, and so on until you get to the leaf nodes.

Summary

- The MVC Pattern is a compound pattern consisting of the Observer, Strategy and Composite patterns.
- The model makes use of the Observer pattern so that it can keep observers updated, yet stay decoupled from them.
- The controller is the strategy for the view. The view can use different implementations of the controller to get different behavior.
- The view uses Composite Pattern to implement the user interface, which usually consists of nested components like panels, frames, and buttons.
- These patterns work together to decouple the three players in the MVC model, which keeps designs clear and flexible.
- The Adapter pattern can be used to adapt a new model to an existing view and controller.