Even our simple color picker starting getting complex. As we tackle more sophisticated applications we will clearly need approaches to manage/mitigate SW complexity. One approach is *design patterns*.

Effectively, a design pattern describes those aspects of a problem and solution that are the same every time (and thus can be DRY’d up!).

A design pattern is not a particular class or library, it is a template. You will build up a library of these templates over time.

---

The elements of this language are entities called patterns. Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

Christopher Alexander
There are also “anti-patterns”: some bad practices out there or techniques that people have found don’t work well

These are some signs that you are not doing it right
A well known example of design patterns comes from this very influential book on writing object oriented software. A variety of techniques for dealing with common issues. An example would be the factory method, where you create a static method for creating and initializing new objects instead of a constructor. This allows you to swap in different subclasses depending on need.
The three tiered architecture is an architectural design pattern, and the way most web apps are designed.
The Model-View-Controller pattern is used for interactive graphical applications like web applications.

Note the similar structure to the three-tier architecture...

MVC separates the data/resource (Model) from the presentation (View) with the Controller. Generally the controller manipulates the model in response to user actions and presents the resulting model(s) for rendering by the view. I say generally because there are many different implementations of MVC, all of which have slightly different MVC roles.
Rails is an example of the MVC architecture

The model would be an entity (user, movie, etc)
The controller fetches data from the model and gives it to the view
The view renders the interface and handles user interaction
These lines can get blurry, models can get smarter. The views, which are already accepting user input can take on more of the controller’s job.

In Backbone, for example, the controller has been absorbed by the view in many web apps, there is little for the controller to do other then pass information back and forth and this cuts out the middle man

Model-view-presenter
the controller assumes more responsibilities and formats that data for viewing, while the viewer becomes more passive and mostly just routs data to the presenter
Model-view-vm
similar to MVP, but here the VM doesn’t actively manipulate the view, the view is grabbing formatted data from the VM.

The whole class of approaches is called MV* and mostly you can gloss over the nitpicking differences between these approaches
Which of the following is a benefit of the Model-View-Controller (MVC) design pattern?

<table>
<thead>
<tr>
<th></th>
<th>HOW WE DO</th>
<th>0/0 students answered</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>MVC supports multi-user access and updating of the model data, with different views for each user</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Ensures there's a one-to-one mapping from each model to a single view</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Provides a window into the model and controller for debugging purposes</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>Like most design patterns, it results in more concise code</td>
<td></td>
</tr>
</tbody>
</table>

SHOW EXPLANATION
Frameworks such as these provide the functionality of architectural patterns. They help us know where to put our data, and provide the communication “glue” to connect the model to potentially multiple views. They tend to be “opinionated” as they support a particular design philosophy that you need to buy into to use them. If you do, they lubricate the path, if you don’t...
Consider the Color Picker... for each color component, the state, we have three "views": 1) the position of the slider itself, 2) the numeric value label, and 3) the color swatch. All three need to be updated when we change that component. How can we do so? What patterns could be relevant?

Ideally, you will pick the framework that follows the design patterns that match the goals of your application. More often, the choice is made by other concerns like what is hot now, the last article you read, or because someone is forcing you to use it...

Along those lines, we will be using React for the rest of the semester.
Speaking of which, we will be using React for the rest of the semester. Because it is the best? No. Because it isn’t horrible, it isn’t overly complex to learn, and because it is very “now”.

React is a framework for just the View in MVC (although not all would agree with that characterization). And in particular it is designed to solve a very specific problem in client side applications: implementing updates to multiple views of the same data.

A key innovation in React is making that re-rendering process very fast. React maintains a virtual DOM that represents the ideal state of the UI. Changing the application state triggers re-rendering, which changes the virtual DOM (those changes are fast since only the "virtual" DOM is changing). Any differences between the virtual DOM and actual DOM are then reconciled to the bring the actual DOM to the desired state. But only those elements that changed are updated making this process more efficient.

Those operations on the (virtual) DOM are the "part that is the same" and occur entirely "behind the scenes" within React. As a developer your focus is just on rendering the desired UI.