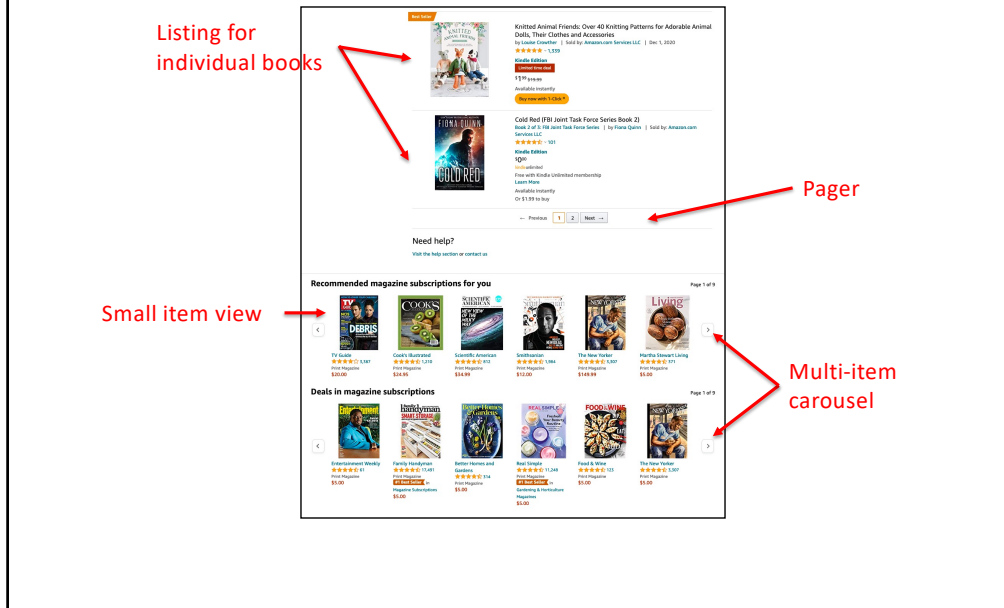


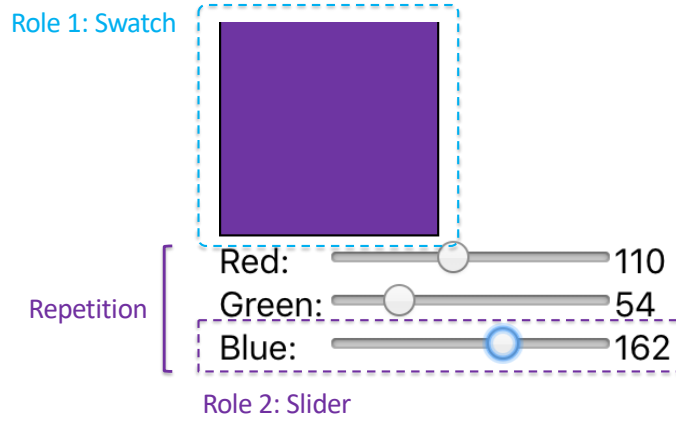
# Component based web design



Here is an example from Amazon. If we opened the developer tools to view the page, it would look like one large "chunk" of HTML, but that is not how the page is designed or implemented. Instead, it is constructed as a set of components that can be composed together to create the final page (application). There are all kinds of components on this page: individual book listings, the pager, the multi-item carousels, the small item views. Each one of these entities is a component. They all have some "generic" form (i.e., show cover, title, reviews), some associated data that changes how each instance appears, and potentially some interactive functionality.

This is largely how modern web development is done, by composing "smart" modular components rather than hand coding everything in raw HTML.

## “Color picker” component example



Consider a Color Picker... It has a swatch that shows the current color and three sliders for changing the RGB values. What are some possible sub-components? The color swatch and the slider (for each color component). What are we looking for when thinking about components? We are looking for distinct roles, and repetition.

1. The swatch and the sliders have distinct roles. <click>
2. And the sliders for each color component are repetitive. <click>

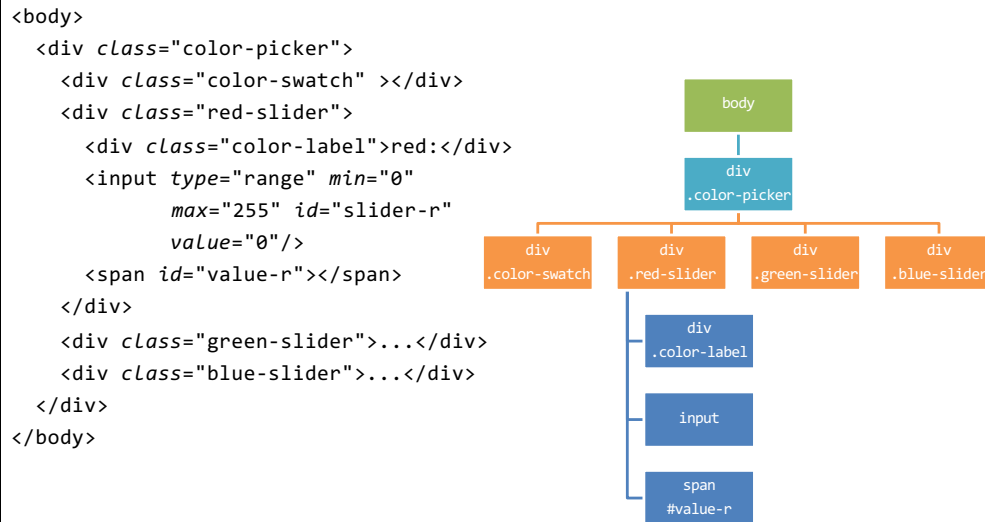
As we start to think about our applications in terms of components, it is not just thinking about how we might “split” up the way the page is displayed, it also thinking about data, or specifically, state. What information do our components maintain and how might it change as the user interacts with the application.

Consider the red channel of the color picker:

- What data do we have? The color value.
- How many views of that data do we have? Three: 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 the red value.

Implementing that kind of interactivity is the role for JavaScript in the browser.

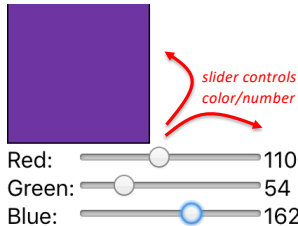
# The DOM and HTML



I will use the term the DOM frequently. At a high-level the “DOM” is what the browser renders and I will use it (“DOM”) as a catch-all term for what is shown on the screen.

More specifically, a web page (or document) is a set of (many) nested boxes, i.e., nested elements. The Document Object Model (DOM) is the tree data structure representing the nested structure of the page. The boxes (HTML tags in our context) are nodes in the tree. [The DOM properties and methods \(the API\) provide programmatic access to the tree to access or change the document’s structure, style or content.](#)

## Use JS to create interactivity



Red: 110  
Green: 54  
Blue: 162

slider controls color/number

```
<div class="blue-slider">  
  <div class="color-label">blue: </div>  
  <input type="range" id="slider-b" .../>  
  <span id="value-b"></span>  
</div>
```

```
// Set oninput callback for each slider  
sliders.forEach((slider) =>  
  slider.addEventListener("input", update));  
  
const update = function() {  
  colorBox.style.background =  
    `rgb(${sliders[0].value}, ${sliders[1].value}, ${sliders[2].value})`;  
  sliders.forEach((slider, index) =>  
    labels[index].innerHTML = slider.value);  
};
```

Update swatch and numeric text

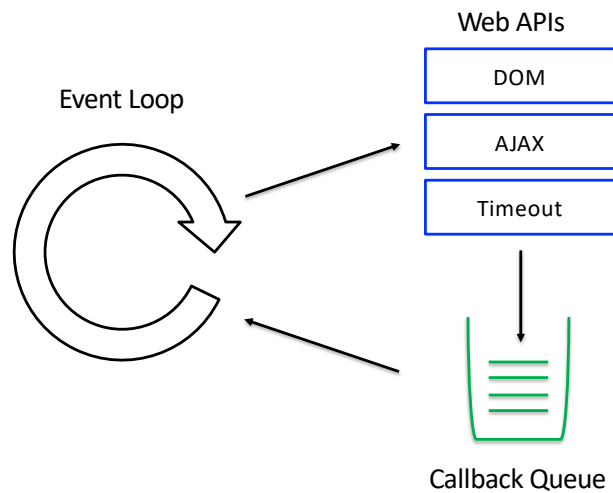
Here is an excerpt from the JS implementation for our color picker, showing both the HTML/DOM and the key function that updates the color swatch and the numeric value each time a slider is moved, i.e., keeps everything in sync.

<click> This function is provided as callback for the "input" event on the slider (an event defined in the DOM), that it is triggered moving the slide input.

In that function, it uses properties defined in the DOM (value, innerHTML) to retrieve the current values for each slider and uses those values to update the swatch and the labels.

Interactive demo: <https://codepen.io/mlinderm/pen/NWqPbeq?editors=1010>

Recall that the browser is  
*asynchronous*



That is when our JavaScript code runs the first time it doesn't actually change the color. Instead, it registered a callback to be invoked when the user moves the slider (the "input" event). Whenever the user moves the slider, that callback, the update function, is placed in the callback queue to be executed at the next opportunity.

The update function closed over references to the colorBox, sliders and labels so that it can access and update those elements even though the corresponding variables are no longer in scope when that function executes.

## How to extend the color picker?

RGB Calculator

rgb(255, 25, 0)

#ff1900

hsl(6, 100%, 50%)

R: 255

G: 25

B: 0

255

25

0

Input & Output

[https://www.w3schools.com/colors/colors\\_rgb.asp](https://www.w3schools.com/colors/colors_rgb.asp)

We can imagine we would want to add more inputs `<click>` and outputs, `<click>`, e.g., the color in hex, etc. Our color picker just got a lot more complicated! How do we keep all these inputs and outputs in sync with each other (a change to any input should update all outputs)? If we continued with the “vanilla JS” approach, each time we add a new input or output we would need to update the callbacks to update all other inputs/outputs. Simple enough at the start, but you can imagine it gets complicated quickly. Each of the input elements, the numeric input, the slider, the text box, all maintain their own state and the data needs to flow back and forth between them.

## Different “design patterns” for the same problem



- Event based (e.g., Backbone)  
Changing the data triggers an event  
Views register event handlers
- Two-way binding (e.g., Angular)  
Assigning to a value propagates to dependent components and vice versa
- Unidirectional dataflow (e.g., React)  
Efficiently re-render all subcomponents when data changes

The challenge of keeping everything in sync is not a new problem... Some solutions that have been/are in use. In class we are going to focus on React (again one choice amongst many...)

## Philosophy of React

1. Render the UI as it should appear for any given state of the application
2. Update the state in response to user actions
3. Repeat (i.e., re-render UI with new state)

*The key conceptual idea is that those two steps are now decoupled and so simpler*

*The key technical enabler was efficient re-rendering when the data changes*

React is a framework (library) designed to help us solve this exact problem. That is build highly interactive and “reactive” UIs. The key idea is to decouple the render of the current state from the updates to that state. You just need to answer those different questions and do so separately. What do I want the UI to look like at any given moment – more formally for any given state of the application – and how do I update that state based on user actions. We don’t have to answer the much trickier question of how do we want to update the UI in response to user actions. React takes care of efficiently propagating those state changes to (and throughout) the UI.

Another way to think about it: in React, data only flows one way. That is there is a single source of truth, the application state and the data flows to all views of that data. To change the view, we change the state and propagate those changes throughout the UI. There are not two pieces of state that need to stay in sync, i.e., data doesn't flow back and forth.

[https://medium.com/@dan\\_abramov/youre-missing-the-point-of-react-a20e34a51e1a](https://medium.com/@dan_abramov/youre-missing-the-point-of-react-a20e34a51e1a)



What is the state in the “extended” color picker, i.e., what information do we need to uniquely specify the UI?

- A. The current color components
- B. *A and* the slider positions
- C. *B and* the numeric text inputs
- D. *C and* the outputs, e.g., hex output

Answer: A

By state we mean what information/data do I need to uniquely specify the UI. In this case there is only one piece of information needed to uniquely specify the UI – the RGB color components. All the inputs and outputs are determined by that information (that is the positions of the sliders, the swatch color, etc.) and should all show the same information.

## What is the state in our “enhanced” color picker?

Recall state in React is the answer to the question: *What information do I need to uniquely specify the UI?*

```
const [red, setRed] = useState(0);  
const [green, setGreen] = useState(0);  
const [blue, setBlue] = useState(0);
```

JS Note: De-structuring assignment splits array into distinct variables

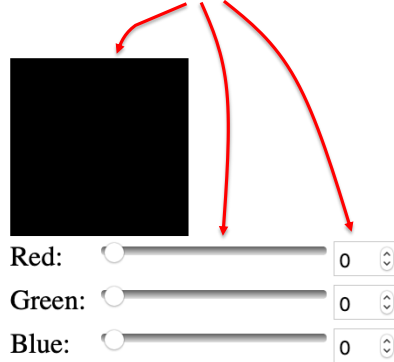
That’s it! Even in the most complex color picker, the only state is the 3 current color components. Every aspect of the UI depends on those three values. Here we implement that state using Hooks. At its simplest, we can create state with the `useState` function. This returns an array with the value (the current value for our state object, initialized to the value we pass into `useState`) and a setter function for updating the state (e.g., `setRed`). We shouldn’t (and by declaring it as const can’t) change the value (by reassigning red, etc.), instead we use the setter function. When we call the setter function that signals to React to re-render to update the UI based on the new state value.

JS note: This is an example de-structuring assignments, that is assigning the elements in the array returned by `useState` to individual variables.

# 1. Render the UI for a given state

```
const [red, setRed] = useState(0);
```

...



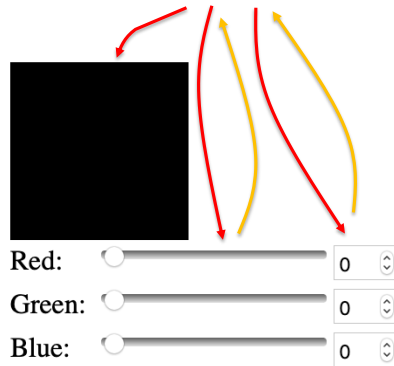
Let's revisit the steps of the philosophy of React in the context of the color picker.

Step1 : The state – the 3 color components – determines the color of the color swatch, the position of the slider and the value in the numeric output. We first make sure we can render those values in our UI. By render that UI we mean render components corresponding to the desired HTML, e.g. a `<div>` for the swatch, “range” `<input>`s for the sliders, etc.

## 2. Update the state, then re-render

```
const [red, setRed] = useState(0);
```

...



Step 2: We will then connect these components such that changing the slider bar changes the state (e.g., via the `setRed` setter) – i.e., the orange arrows.

Step 3: Setting the state will trigger React to re-render, but with a new value of state. That new value of state will propagate (via the red arrows again) to create the new view (with the updated color value).

Is the slider aware of the numeric input (or the swatch)? No. Each component only needs to know the state and how to update that state where relevant! That is how React helps us minimize the complexity when building complex UIs. The complexity grows linearly with the components we add, instead of exponentially with the interactions between those components.

## “Thinking in React”

1. Break the UI into a component hierarchy
2. Build a static version in React
3. Identify the minimal (but complete) representation of state
4. Identify where your state should live
5. Add “inverse” data flow (data flows down, callbacks flow up)

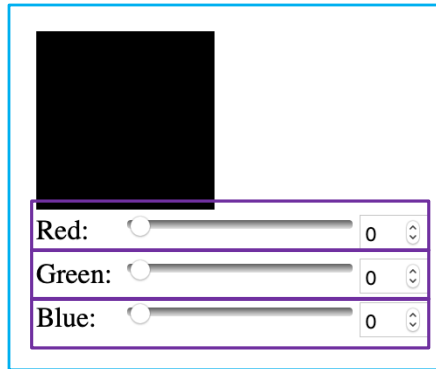
<https://react.dev/learn/thinking-in-react>

We can translate that philosophy of React into a recipe for developing React applications/components...

# Break the UI into components

ColorPicker

LabeledSlider



*A component is function that takes a single argument, the **props**, and returns a hierarchy of components*

The fundamental unit of React is the component. In React, we can implement components as either *classes* or *functions*. For our purposes, we will exclusively use function-based components, but many older examples you might find online will use classes (as did previous versions of this course).

A function-based component is a function that takes a single object argument, termed the **props**, and returns a hierarchy of components (think of these child components like a nested tree, similar to the DOM itself) with a single root. The returned hierarchy is the view, and specifically what is added to the virtual DOM.

The first step in building a React app is break down the UI (the view) into a hierarchy of components and sub-components. Here we will implement the color picker with one main component (the color picker itself, with the swatch) and the 3 sliders with their corresponding value display/inputs.

Explore this starting point at: <https://codepen.io/mlinderm/pen/YzXwNdd>

## Interlude: Expressing the view w/ JSX

```
// Example HTML
var heading = React.createElement("h1", null, "Hello, world!");
// Example component
var person = React.createElement(Person, {
  name: p.name,
  address: p.addr
});
```

Rendering with “pure” JS

```
// Example HTML
const heading = <h1>Hello, world!</h1>;
// Example component
const person = <Person name={p.name} address={p.addr} />;
```

Rendering with JSX

{embedded JS}

Props

```
function Person(props) { // props is { name: ..., address: ... }
```

Here we render a `<h1>` HTML components, and a custom components that displays information about a person. This is the JS that is actually executed to create our view. But as you might already sense, it is awkward to write. Since we are ultimately producing HTML, we would like a representation that is closer to that eventual product. For us that is JSX. `<click>`

JSX is an extension to Javascript for efficiently describing the UI, including both React components (like `Person`) and HTML (like `h1`). Since JSX is an extension to JavaScript, we will need a compiler, really a transpiler, to convert it to standard JavaScript. The online sandboxes do that for us (as an option) and the tool we will use for setting up React application (e.g., `Next`) integrates the necessary compiler to *transpile* JSX (and support features of ES6). We will use JSX in our components (as it is much more concise and clearer). However, you should realize that it is being translated directly into “plain” JavaScript functions like shown above (i.e., it is just “syntactic sugar”).

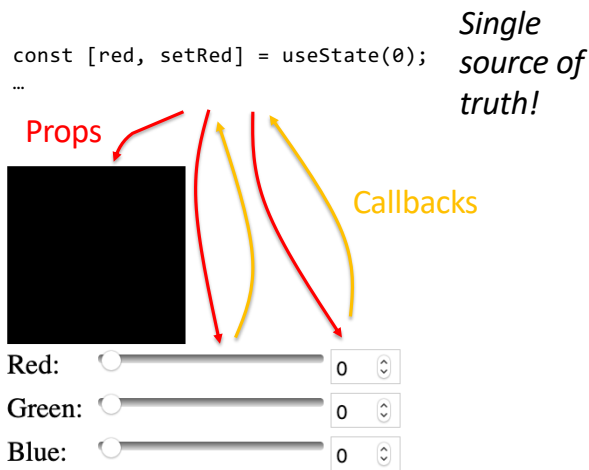
The names of the props, the LHS of the `=` becomes the keys/properties in the props input, and the RHS (in the curly brackets), becomes the values. `<draw line connecting those>`. Inside the curly brackets are Javascript expressions and typically references to variables defined in our component function. `<click>`

The names for our props, “name” and “address” in this example, are our choice (excluding some reserved names). But the props we pass must match the props

expected by the component. This JSX implies that there is a Person component, i.e., a `Person` function, and it expects its props to contain a name and address property. Note that the convention is to start component names with upper case letters.



## Props flow down, callbacks flow up



Recall that our state is just the 3 color components. Where should this state live (step 4 in “Thinking in React”)? We need this information in the sliders, i.e., in LabeledSlider, but also in ColorPicker to set the swatch color. Per the React documentation: “Often, several components need to reflect the same changing data. We recommend lifting the shared state up to their closest common ancestor.” Thus, we will implement the state in the ColorPicker component (the closest common ancestor).

That state then “flows down” to the labeled sliders as props to those components. React components must act like pure functions with respect to their props. That is a component can't modify its props (this enables efficient updates). To communicate updates “back up” we supply a callback to the child that modifies the state in the parent (the “inverse” data flow or step 5 in Thinking in React).

Some important notes about modifying state:

- Do not modify state directly, instead use the setter.
- State updates may be asynchronous. React may batch updates, and so you shouldn't assume the state has immediately changed after the call to the setter.

## Putting it all together: the ColorPicker

```
function ColorPicker() {  
  const [red, setRed] = useState(0);  
  const [green, setGreen] = useState(0);  
  const [blue, setBlue] = useState(0);  
  
  const color = {  
    background: `rgb(${red}, ${green}, ${blue})`  
  };  
  return (  
    <div>  
      <div className="color-swatch" style={color} />  
      <LabeledSlider label="Red" value={red} setValue={setRed} />  
      <LabeledSlider label="Green" value={green} setValue={setGreen} />  
      <LabeledSlider label="Blue" value={blue} setValue={setBlue} />  
    </div>  
  );  
}
```

JS Note: Template literal

Passing state as prop

Passing a Callback as prop

Here is our ColorPicker component. When we talk about what a component renders, we are talking about the components that are returned. Here we see an enclosing `<div>` (a standard HTML component) with a child `<div>` for the swatch, and 3 child React components, the LabeledSliders. At the top of the ColorPicker we define the three pieces of state, which are then passed to their corresponding sliders as props (along with the callback function to update the value). Every time React renders the color picker it will execute this function to generate (updated) DOM. In the process it will set the background color style based on the current values of the color components.

JS note: This includes an example of a template literal where we dynamically construct a string from JS variables.

Check out a demo of the complete implementation at:  
<https://codepen.io/mlinderm/pen/JjdYmOK>

## Forms (inputs) are implemented as “controlled components”

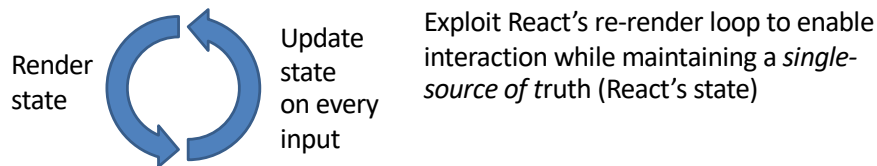
JS note: De-structure props object into variables

```
function LabeledSlider({ label, value, setValue }) {  
  return (  
    <div>  
      <span>{label}</span>  
      <input type="range" min="0" max="255"  
        value={value}  
        onChange={event => setValue(parseInt(event.target.value, 10))}</input>  
      <span>{value}</span>  
    </div>  
  );  
}
```

Input value determined by React state (or props derived from state)

“Event Object” with event data

Any change invokes callback to update state (in this case via callback passed as a prop)



Here is the labeled slider implementation: Note that we use “controlled” `<input>` components. Controlled components are form elements with state controlled by React. Uncontrolled components maintain their own state. The latter is the way `<input>` elements naturally work (recall the “vanilla JS” color picker). The former, “controlled”, is the recommended approach as it ensures there is only one source of truth, the React state. `<click>` We set the `<input>` element’s value from state and provide an `onChange` (or other relevant) callback to update that state in response to user input `<click>`. Each state change triggers a re-rendering that shows the changes the user just initiated. `<click>`

There are many of these potential “callback” props, like `onChange`, that are triggered in response to specific events in the browser, like changing an inputs, hovering, etc. We can learn more about them in the React documentation for the specific HTML component: <https://react.dev/reference/react-dom/components/input>

That’s it! That is all we need to build the fully interactive color picker! Our more complex applications are just these same ideas, this same “Thinking in React” process, applied again.

JS note: This includes an example of destructuring the single props object argument into individual variables.

## Which of the following are equivalent and correct callback implementations?

1.	<code>&lt;button onClick={() =&gt; setValue("Clicked")} &gt;</code> <i>event</i>
2.	<code>&lt;button onClick={setValue("Clicked")} &gt;</code> <i>setValue</i>
3.	<code>const handleClick = () =&gt; setValue("Clicked"); ... &lt;button onClick={handleClick} &gt;</code> <i>event</i>

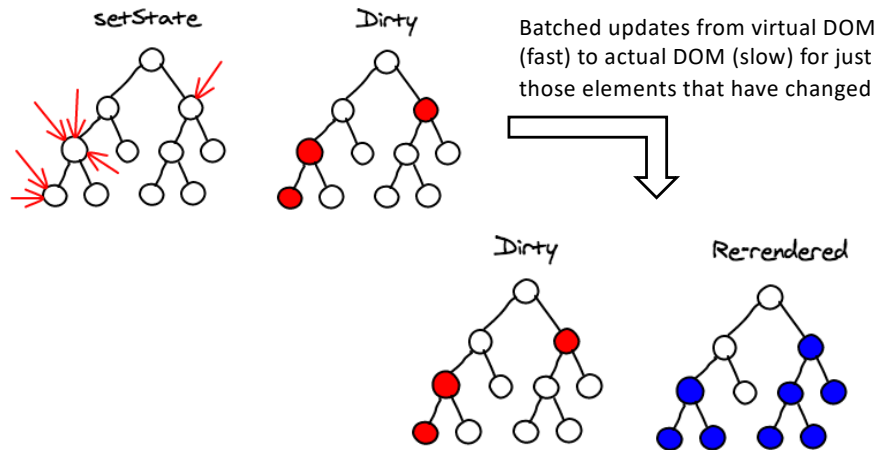
- A. 1 only
- B. 2 only
- C. 3 only
- D. 1 and 3
- E. All

*= { setValue }*

Answer: D

What's different about 2? The other two are passing a function to be invoked in the future when the button is clicked. While 2 is invoking the `setValue` function during render and passing its result as the prop (not later when the click occurs). We ultimately want to pass a function to the event handlers like onClick, onChange, etc. This is a common error and often tricky to find (because the syntax is so similar). Keep an eye out for messages suggesting something is happening before your application is ready. That suggests a function intended as a callback is getting invoked during (the initial) render, instead of later when an event occurs.

## What is React doing behind the scenes?



<https://calendar.perfplanet.com/2013/diff/>

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 (and those changes are fast since only the "virtual" DOM is changing and it is just objects in memory). Any differences between the virtual DOM and actual DOM (on the screen) are then reconciled to bring the actual DOM to the desired state. But only those elements that changed are updated making this process more efficient.

Further, state updates may be asynchronous. React may batch updates. Thus, you shouldn't assume the state has immediately changed after the call to the state setter.

Often re-rendering is sufficiently fast that we don't need to worry about when and how components are re-rendering. And that should be the starting point. If, however, we observe performance problems (and only if we observe problems) we can optimize the rendering behavior (avoid unnecessary re-renders). See the "read more" links on the course web page for additional information about how re-rendering works and how we could optimize when components re-render.

## Why React: Design patterns

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.

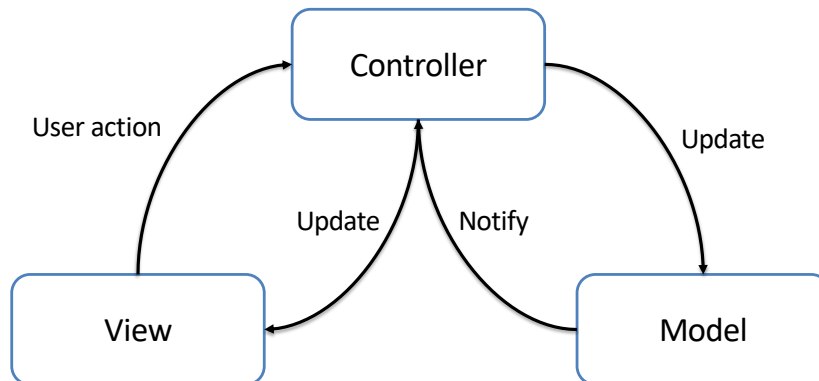
Christopher Alexander

Even our simple color picker started getting complex. As we tackle more sophisticated applications, we will 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 "mental library" of these templates over time. React is an implementation of design pattern for building interactive UIs. The operations on the (virtual) DOM are the "part that is the same everytime" and occur entirely "behind the scenes" within React. As a developer your focus is just on rendering the desired UI. That is, you can focus on the part that is different each time instead of the parts that are the same.

\* DRY is an acronym for "Don't Repeat Yourself", i.e., don't duplicate code/work

## Design pattern: Model-View-Controller



In an even more general sense, React is trying to solve the problem of what do we show the user (what should appear on the screen) in a graphical application and how does that “view” change in response to user actions. A more general design pattern for that problem is the Model-View-Controller pattern (widely used in web applications, but also GUIs). 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(s). I say generally because there are many different implementations of MVC, all of which have slightly different MVC roles. There are also other related patterns like MVVM – Model View ViewModel that divide up responsibilities slightly differently.

Where does React fit into this pattern? At a high-level, it is just the “V” (the view) (although not all would agree with that characterization), with the server (something we will talk about in subsequent classes) responsible for the C and M. The reality is a little less clear cut. Our React components will have elements of V and C (we have already seen that...). As with other design patterns, the value is in the “core” of the solution, that is thinking about how to separate/decoupling the roles of storing the state of the application (the model), how we visualize that state (the view) and how we modify the state of the application in response to user actions (the controller). If we think about those roles explicitly, we are more likely to produce “beautiful” code.

In some frameworks, that decomposition is not so “optional”, that is the framework is really built around this design pattern, and so we need to explicitly identify those components of our application.



## Symptoms of anti-patterns, i.e., tactical programming or a sign it's going awry

- Viscosity  
Easier to do a hack than do the “Right Thing”
- Immobility  
Can't DRY out functionality
- Needless repetition
- Needless repetition
- Needless complexity from generality

There are also anti-patterns, that is code that looks like it should probably follow some design pattern but doesn't. Such code is both the cause and result of “technical debt”. Some symptoms of anti-patterns...

These are more specific manifestations of the tactical programming we discussed previously and signs that complexity is winning (recall those signs were: change is hard, high-cognitive local and unknown unknowns).

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license

## Design patterns (or abstractions): Moderation in all things

Two key questions:

1. What is benefit of this pattern (abstraction)?
2. What is the cost of this pattern (abstraction)?



If we can't answer those questions we don't know if we are coming out ahead. What if we are paying a cost, but not getting any benefit, that is paying for a solution to a problem we don't have.

Using a particular design pattern or abstraction (we can think of React abstracting away the details of updating our UI in response to state changes) doesn't automatically make our software better, it only does so if it solves (and is a good fit for) a problem we actually have. Our course webpage doesn't use React. Why? Because it is almost all static (no state to interactively update!)

How do we know if our problem is a good fit? As a start, we want to keep an eye out for the anti-patterns we just discussed. A positive sign is that we are successfully abstracting away unimportant details. The word unimportant is crucial (and sometimes hard to define). In this context, we would describe the mechanics of efficiently updating the DOM as unimportant. Another sign is that a good abstraction will have a simple interface, but the functionality behind that interface is "deep". The interface for managing state with React is very simple, but the functionality behind the rendering process is substantial ("deep").

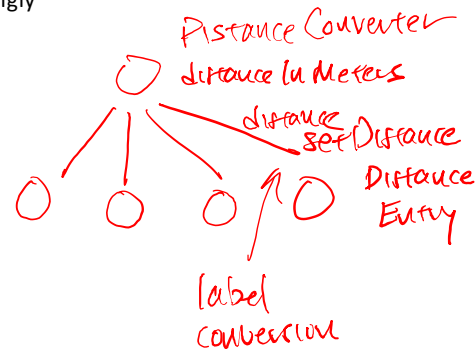
<https://kentcdodds.com/blog/how-to-react>

Ousterhout, John K. . A Philosophy of Software Design, 2nd Edition

## How would you decompose this unit converter component?

miles	1
km	1.609
feet	5280
meters	1609

Changing one distance should change all the others accordingly



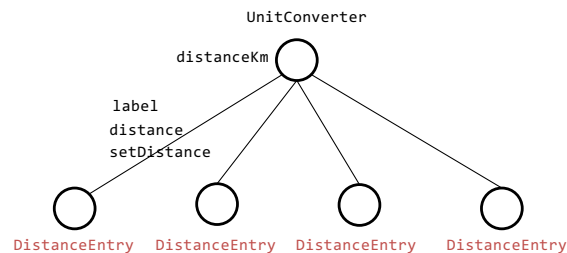
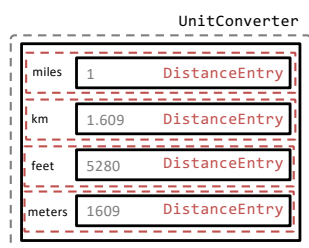
- What are repeated elements? The DistanceEntry. There are 4 such components that are very similar. These are composed within the parent UnitConverter.
- What is the state? There is just one piece of state, the distance (in some units). All other distances reflect that same underlying information.
- Where does that state live? Since it is needed in all the DistanceEntry components, it needs to live in the nearest common ancestor, UnitConverter. It would then flow down as a prop to each DistanceEntry. UnitConverter would pass a callback for the child DistanceEntry components to update that state.
- Are these 4 distant components, or 4 instances of the same component? Likely the latter. No need to repeat the functionality. We can pass the label as a prop (like we did for labeled slider), along with appropriate expressions/functions for updating the distance. Why would that be preferable? Fewer components to develop and test!

Example implementation: <https://codepen.io/mlinderm/pen/WbeaLqX?editors=0010>

## How would you decompose this unit converter component?

miles	<input type="text" value="1"/>
km	<input type="text" value="1.609"/>
feet	<input type="text" value="5280"/>
meters	<input type="text" value="1609"/>

Changing one distance should change all the others accordingly



- What are repeated elements? The `DistanceEntry`. There are 4 such components that are very similar. These are composed within the parent `UnitConverter`.
- What is the state? There is just one piece of state, the distance (in some units). All other distances reflect that same underlying information.
- Where does that state live? Since it is needed in all the `DistanceEntry` components, it needs to live in the nearest common ancestor, `UnitConverter`. It would then flow down as a prop to each `DistanceEntry`. `UnitConverter` would pass a callback for the child `DistanceEntry` components to update that state.
- Are these 4 distant components, or 4 instances of the same component? Likely the latter. No need to repeat the functionality. We can pass the label as a prop (like we did for labeled slider), along with appropriate expressions/functions for updating the distance. Why would that be preferable? Fewer components to develop and test!