UCLA Samueli School of Engineering

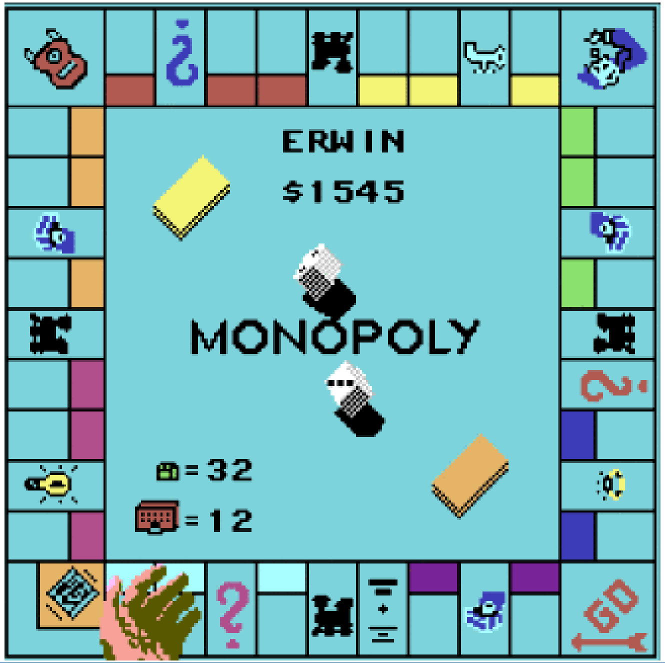
CS 35L Software Construction

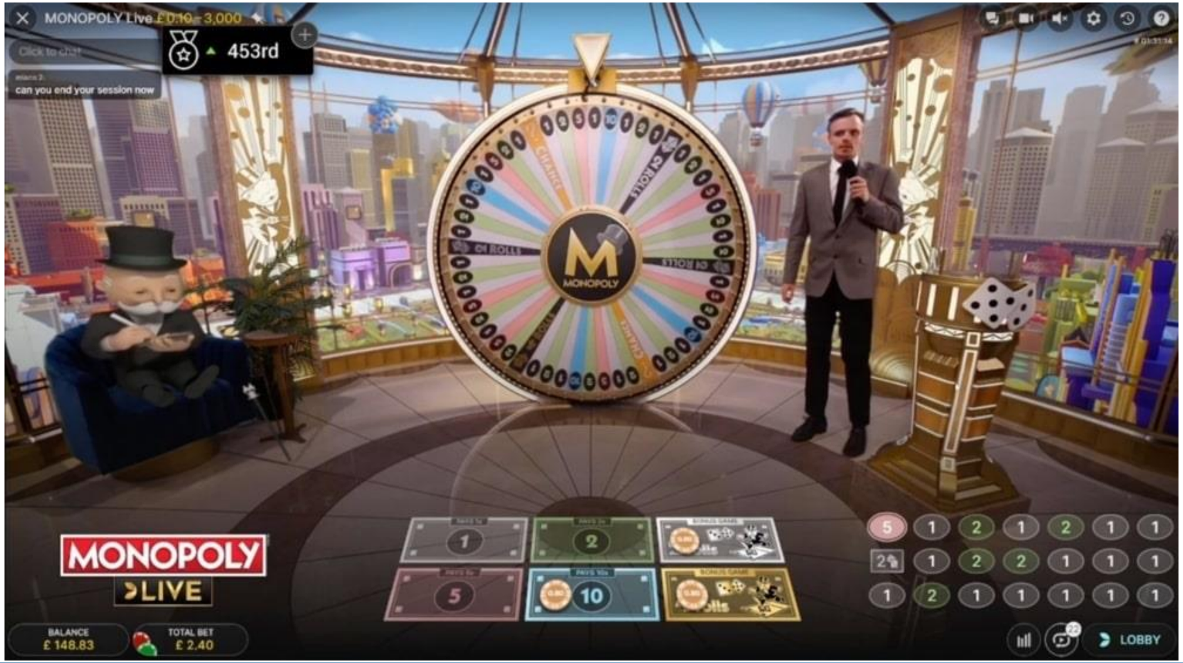
Lecture 5 – React, UI Development & Design Pattern Introduction

Tobias Dürschmid
Assistant Teaching Professor
Computer Science Department

Motivating Case Study: Digital Monopoly Game

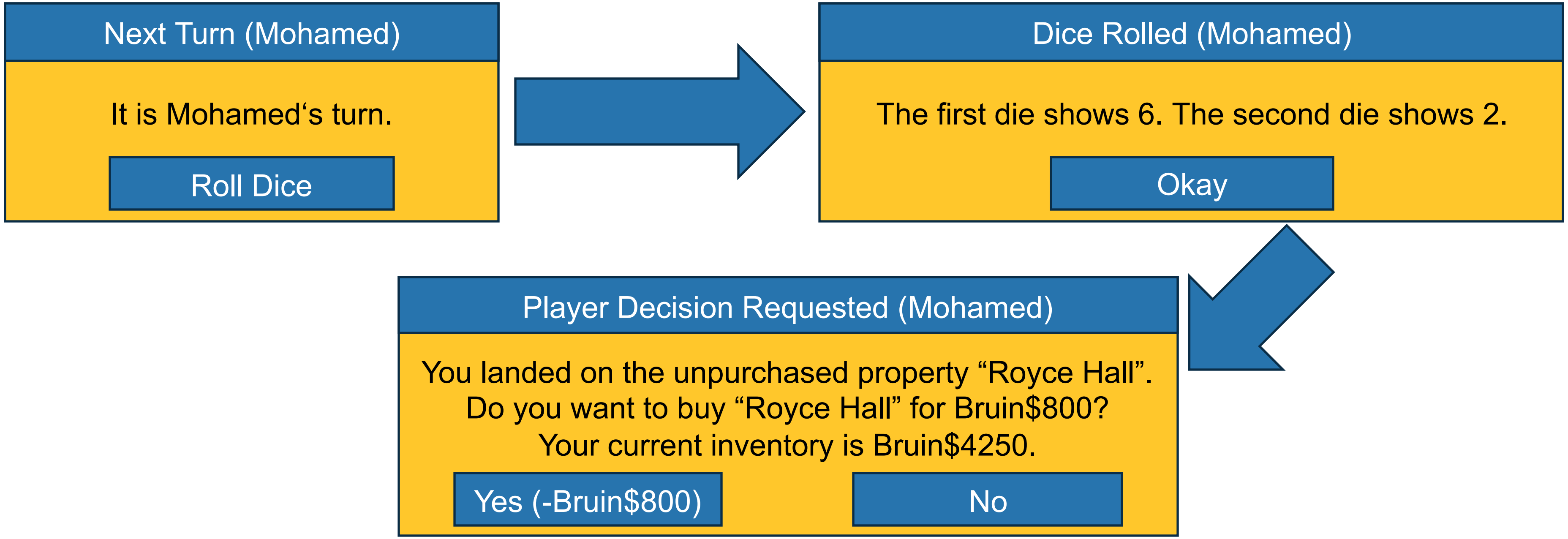
There are dozens of implementations of the game “Monopoly” with vastly different User Interfaces





Motivating Case Study: Digital Monopoly Game

There are dozens of implementations of the game “Monopoly” with vastly different User Interfaces. Even with just plain text UI!



Next Turn (Mohamed)

It is Mohamed‘s turn.

Dice Rolled (Mohamed)

The first die shows 6. The second die shows 2.

Player Decision Requested (Mohamed)

You landed on the unpurchased property “Royce Hall”.
Do you want to buy “Royce Hall” for Bruin$800?
Your current inventory is Bruin$4250.

What Should They All Have In Common?

All of them should implement the exact same game logic!

So at least in theory they could all run the exact same code just with a different UI (ideally even with multiple UIs simultaneously)

Next Turn (Mohamed)

It is Mohamed's turn

Roll Dice

Dice Rolled (Mohamed)

The first dice shows 6.
The second dice shows 2.

Okay

Player Decision Requested (Mohamed)

You landed on the unpurchased property "Royce Hall".
Do you want to buy "Royce Hall" for Bruin$800?
Your current inventory is Bruin$4250.

Yes (-Bruin$800) No

Separation of Concerns

Systems should be **divided** into **distinct** sections, or **concerns**, where each section addresses a separate, specific goal, purpose, or responsibility.
The goal is to make the system **easier to develop, maintain, and evolve.**

Applies to **all software** development. Not just UI

Important General Design Principle

Read more here: https://www.geeksforgeeks.org/software-engineering/separation-of-concerns-soc/

Front-End
Back-End

Separate the UI Implementation from the Domain Logic of your Application

Presentation Layer

Displays information to the user and collects **input** (e.g., positions of players on the board, style of the boards, buttons, …)

↓

Application Layer

Implements the **domain logic** and **behavior** (e.g., effects of community chest cards, player turns, interactions between players)

Doesn't have a clue that there even is a UI. I mean who really needs a UI? Can't we just have command line interfaces with the shell???

Separate the UI Implementation from the Domain Logic of your Application

Presentation Layer

Application Layer

- Allows presentation later to register a **callback** for state changes (e.g., `onBalanceChanged`)
- Allows presentation layer to retrieve information using **getter functions** (e.g., `getCurrentBalance()`)
- Allows presentation layer to forward user events (e.g., `buyProperty(name, user)`)

Doesn't have a clue that there even is a UI.
I mean who really needs a UI? Can't we just have command line interfaces with the shell???

The Observer Design Pattern

Pattern descriptions have many **different formats**. At minimum it should describe **problem**, **context**, and **solution.**

Context

Your application should separate classes from different parts

Problem

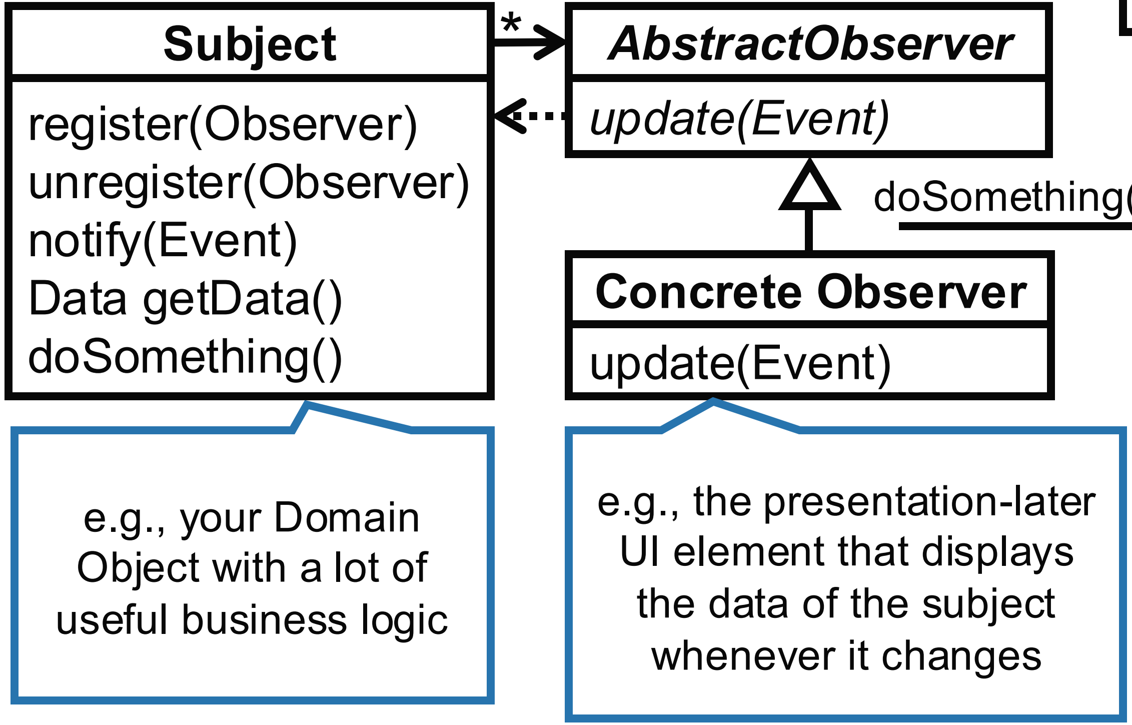
How to get updates from one class to another?

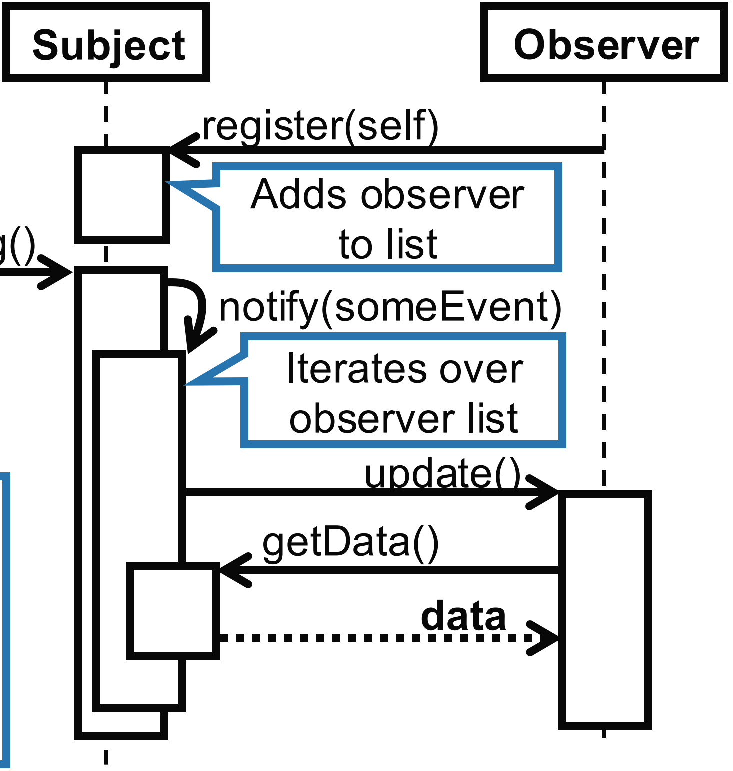
Solution

Define observers that can register for updates in the subject

The Observer Design Pattern

Read more here: https://refactoring.guru/design-patterns/observer


Subject
register(Observer)
unregister(Observer)
notify(Event)
Data getData()
doSomething()
*
AbstractObserver
update(Event)
Concrete Observer
update(Event)
e.g., your Domain Object with a lot of useful business logic
e.g., the presentation-later UI element that displays the data of the subject whenever it changes



Design Patterns

found in **many instances**

It is a **good** solution, but **not always the best** one

“A *pattern* is a common, acceptable **solution** to a reoccurring **problem** that arises in a specific **context**”

The problem is generic enough so that the it **generalizes** beyond a few concrete cases

Patterns always refer to a specific **situation**, **goal**, or **trade-off**. Patterns are **not universally** good

Design Exercise: Talk to Your Neighbor(s)!

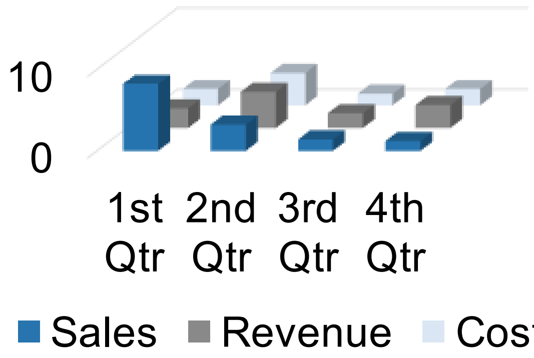
We want to design an **interactive application** that represent the same information across **different views** and that should **update immediately** when the information changes.

Table View

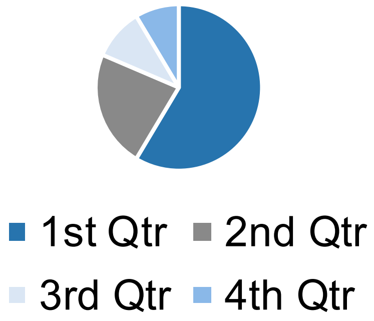
	Sales	Revenue	Cost
1st Qtr	8.2	2.4	2
2nd Qtr	3.2	4.4	4
3rd Qtr	1.4	1.8	1.4
4th Qtr	1.2	2.8	2

Changes here trigger updates in other views

Bar Chart View



Sales Pie Chart View



Read more here: https://www.geeksforgeeks.org/software-engineering/mvc-framework-introduction/

Model View Controller

Role*:* Model	***Collaborators***
Responsibilities - Provides core functionality (main business logic) - Registers views and controllers - Notifies components about data changes	- View - Controller

Role*:* View	***Collaborators***
Responsibilities - Displays information to user - Creates controller - Retrieves data from model - Implements update	- Model - Controller

Role*:* Controller	***Collaborators***
Responsibilities - Accepts user input - Translates evens to service requests for the model or display request to the view	- View - Model

Example Pattern: Model-View Controller

Context

Designing an **interactive application**.

Problem

How to represent the **same information** across **different simultaneous views** that should **update immediately** when the information changes?

Solution

Divide your application into **model**, **view**, and **controller**.

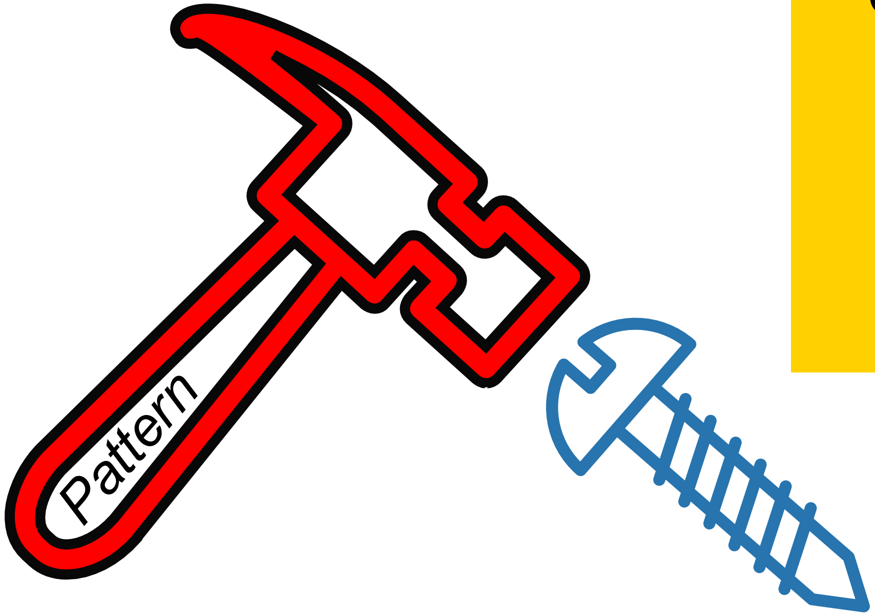
Lesson Learned: Start By Considering Existing Solutions

- Most problems have been **solved already** and described in a **well-documented** way
- **Knowing existing solutions and patterns in your field** can save you a lot of time and effort

Consequences of MVC

- **Extensibility** of **Views**: Adding new views requires little effort
- **Changeability** of **Views**: Changing a view does not require changing other parts of the software
- **Performance of Updates**: Many calls are made between *models* and *views*
- **Extensibility of Model**: Adding new features to the model are likely to require changes in *controllers* and *views*.

If you have a Hammer, everything looks like a Nail



Software quality is **not** measured by the number of used patterns

User Input Validation

Context

Your application requires a significant amount of user input.

Problem

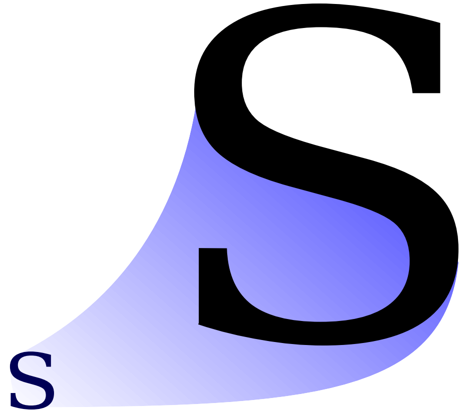
How to ensure invalid user input does not corrupt the data in our downstream application

Solution

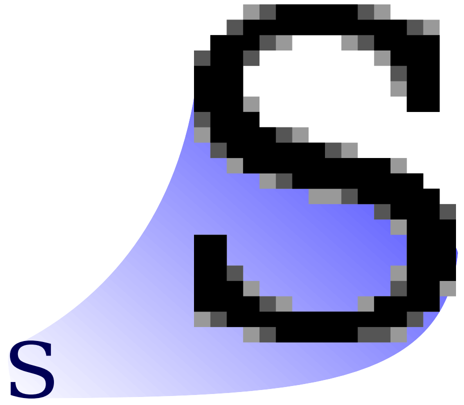
Instantly validate user input (for example with RegEx) and show a meaningful error message to users.

Use SVGs Whenever You Can

Scalable Vector Graphics (SVG) are vector-based graphics defined by mathematical equations for shapes. They scale to any size and often have smaller file sizes.

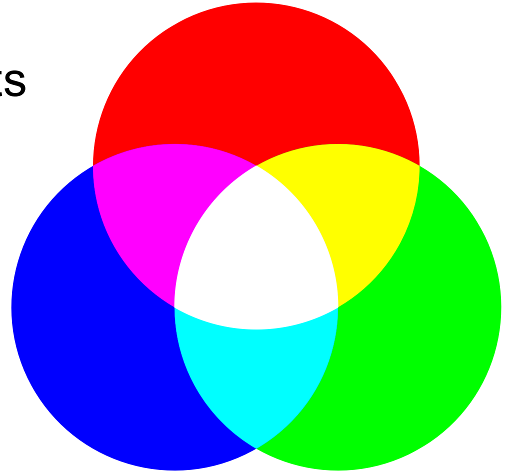


Bitmaps (e.g., JPEG, PNG, BMP, GIF, …) are pixel-based graphics composed of a fixed grid of colored squares. Due to their general-purpose representation, they are supported by more editing software.



There are many Different Color Representations

- **RGB** (**Red**, **Green**, **Blue**) is the most common format to represent digital colors. The resulting color is a mix of red, green, and blue light.
  - The Hex-Color representation uses two hexadecimal digits for each color (**#RRGGBB** e.g., **#2774AE**, **#FFD100**)
- **HSL/BSH** (Hue, Saturation, Luminance/Brightness) lets you represent colors based on their color tone (hue), and color intensity (satuation) and brightness (luminance)
- **CMYK** (**Cyan**, **Magenta**, **Yellow**, **Keycolor** (Black)) represents colors based on the subtractive color model used in printing and is less common for screen-based rendering



See https://www.w3schools.com/tags/default.asp

Important HTML Tag

Content Tags

- **Headings** (`<h1>` to `<h6>`)
  e.g., `<h1>`Top Heading`</h1>`
- **Paragraphs** (`<p>`) e.g., `<p>`Example`</p>`
- **Lists** (`<ul>` **unordered list**
  and `<ol>` **ordered lists**)
  e.g.,

```
<ol>
      <li>first item</li>
      <li>second item</li>
</ol>
```

Top Heading

Example

1. first item
2. second item

- First item
- Second item

See https://www.w3schools.com/tags/default.asp

Important HTML Tag

Text Formatting Tags

- **Bold** (`<strong>` previously: `<b>` or `<B>`)
  e.g., `<strong>`Important Text`</strong>`
- **Italics** (`<em>` previously: `<i>` or `<I>`)
  e.g., `<em>`emphasis`</em>`
- **Underline** (`<ins>` previously: `<U>`)
  e.g., `<ins>`inserted`</ins>`
- **Strikethrough** (`<del>`)
  e.g., `<del>`deleted`</del>`

Important Text

emphasis

inserted

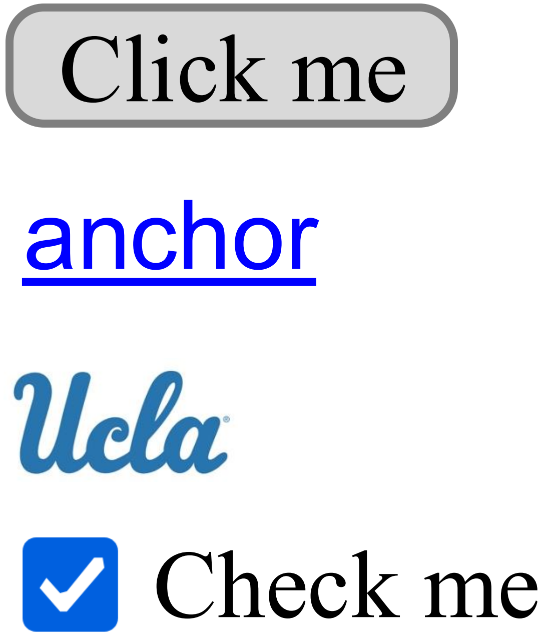
~~deleted~~

See https://www.w3schools.com/tags/default.asp

Important HTML Tag

Text Formatting Tags

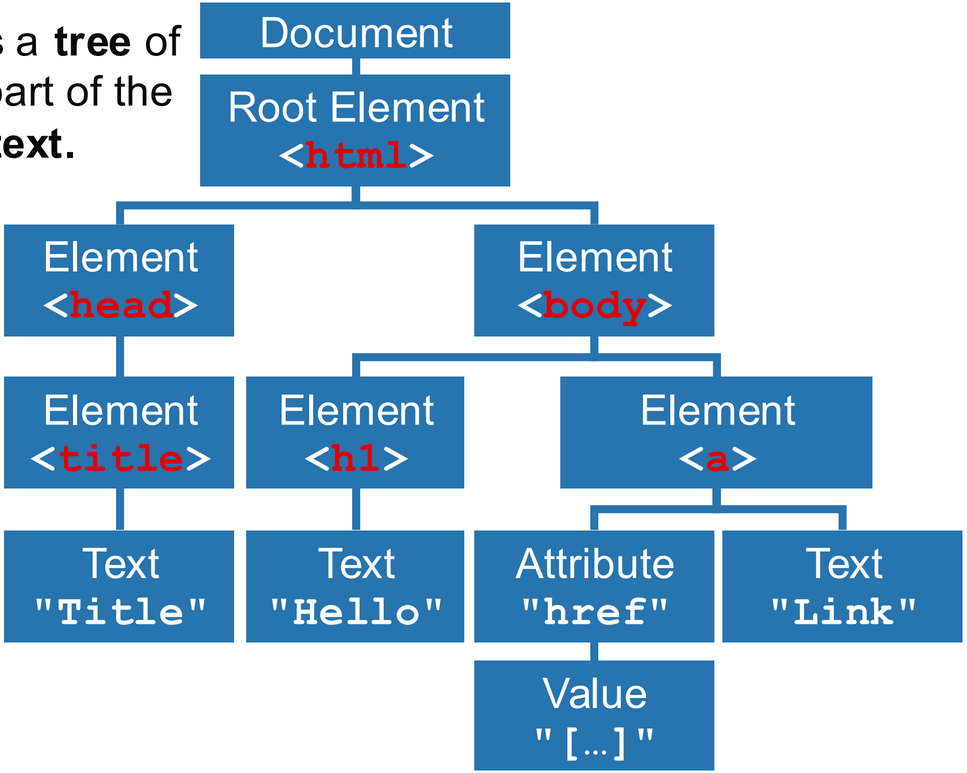
- **Button** (**`<button>`**)
  e.g., **`<button>`**Click me**`</button>`**
- **Links** (**`<a>`**)
  e.g., **`<a href="ucl.edu">`**anchor**`</a>`**
- **Images**(**`<img>`**)
  e.g., **`<img src="https://u.edu/l">`**
- **Inputs** (**`<input>`** with various types)
  e.g., **`<input type="checkbox">`**check me**`</input>`** (or text, file, email, radio, …)



HTML DOM (Document Object Model) is a programming interface for HTML documents.

It represents the structure of an HTML page as a **tree** of objects, where each **object** corresponds to a part of the document, such as an **element, attribute, or text.**

```
<html>
<head>
  <title>My Title</title>
</head>
<body>
  <h1>Hello</h1>
  <a href="[…]">Link</a>
</body>
</html>
```



CSS (Cascading Style Sheets)

- **Declarative-style** computer programming language used to **define the visual presentation and layout** of web pages.
- Works in conjunction with markup languages like **HTML** (which **provide the structure and content of a webpage**)

CSS
Visual presentation & Layout

HTML
Content & Structure

Separation of Concerns ✓

More details here:
https://www.w3schools.com/css/

CSS can define style classes for HTML `divs` (divisions, used as containers)

example.html

```
<!DOCTYPE html>
<html>
<head>
  <title>CSS Example</title>
  <link rel="stylesheet"
        href="example.css">
</head>
<body>
  <h1>CSS Example</h1>
  <div class="rounded-box">
    <p>This will be styled
       with CSS.</p>
  </div>
</body>
</html>
```

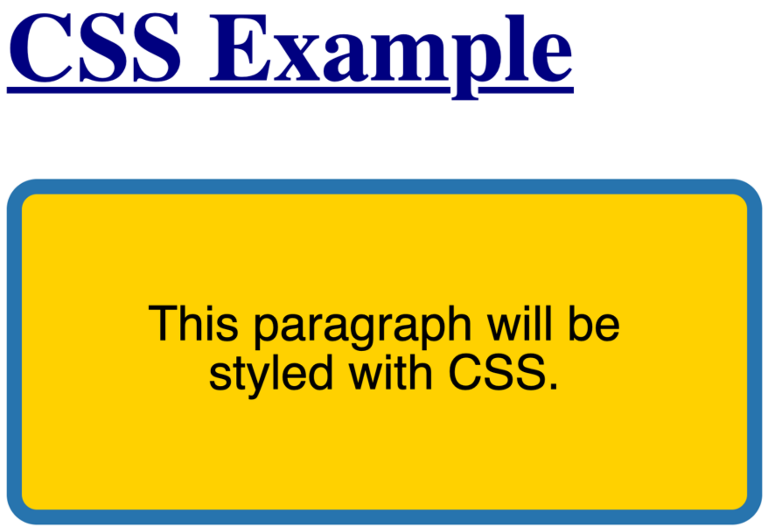
example.css

```
h1 { /* Heading 1 will be navy colored
and underlined */
    color: navy;
    text-decoration: underline;
}
.rounded-box {/* Will become a class
                 for divs*/
    width: 200px;
    background-color: #FFD100;
    border: 5px solid #2774AE;
    border-radius: 10px; /* corners */
    text-align: center;
    font-family: sans-serif;
}
```

CSS can define style classes for HTML `divs` (divisions, used as containers)

example.html

```
<!DOCTYPE html>
<html>
<head>
  <title>CSS Example</title>
  <link rel="stylesheet”
        href="example.css">
</head>
<body>
  <h1>CSS Example</h1>
  <div class="rounded-box">
    <p>This will be styled
       with CSS.</p>
  </div>
</body>
</html>
```



CSS Example

This paragraph will be styled with CSS.

React.js (aka. React, ReactJS)

- React is a free and open-source front-end JavaScript library for building user interfaces
- maintained by Meta (formerly Facebook)
- React has a **declarative nature**
  - you describe ***what*** you want the UI to look like for a given state
  - React handles the steps to achieve that UI and **keep it updated efficiently when the state changes**.
  - In contrast, **imperative** programming focuses on describing *how* to achieve a result by manually manipulating the UI elements

React uses *JSX* (JavaScript XML)

- JSX is a **syntax extension** that allows developers to **write HTML-like code within JavaScript**. This makes it easier to define the structure and content of UI components.

```
React.createElement('h1',null,'Hello, ',name,'!');
```

```
const name = "Lisa”
[…]
```

JavaScript constant (or variable) defined somewhere

Hello, Lisa

```
return <h1>Hello, {name}!</h1>;
```

Use JavaScript within HTML-like tags via curly braces.

Map function

```
const names = ["Lisa", "Seoyoung", "Mengjun"]

[…]


return (<ul>
  {names.map((name, index) => (
    <li key={index}>{name}</li>
  ))}
</ul>);
```

- Lisa
- Seoyoung
- Mengjun

React Applications use *Components*

- Components are self-contained, reusable pieces of UI that manage their own rendering logic.
- This modularity promotes code **reusability**, **simplifies development**, and **improves maintainability.**
- React Component are **Defined as Functions** with a **Single Return Element**
  - **Inputs to the functions are called pr**

Can be referenced via:
`<WelcomeMessage name="Lisa"/>`

Component Parameters

Component Parameter Reference

```
function WelcomeMessage(props) {
    return <h1>Hello, {props.name}!</h1>;
}
export default WelcomeMessage;
```

Makes the component visible to outside modules

React Components manage their own *State*

```
import React, { useState } from 'react';
function Counter(props) {
  const [count, setCount] = useState(props.initialCount);
```

Initial value of the state

State variable

State change function name

```
  const increment = () => {
    setCount(count + 1);
  };
  return (
    <>
      <p>Current Count: {count}</p>
      <button onClick={increment}>
        Increment
      </button>
    </>
  );
} export default Counter;
```

Reference to the state variable. Changes whenever the state variable is changed via the state change function

Click event handler (the name of the function to be called when the user clicks on the button)

Makes the component visible to outside modules

React Components manage their own *State*

```
import React, { useSta                  t';
function Counter({initialCount}) {
  const [count, setCount] = useState(initialCount);

  const increment = () => {
    setCount(count + 1);
  };
  return (
    <>
      <p>Current Count: {count}</p>
      <button onClick={increment}>
        Increment
      </button>
    </>
  );
} export default Counter;
```

Unwrap props

Initial value of the state

State variable

State change function name

Reference to the state variable. Changes whenever the state variable is changed via the state change function

Click event handler (the name of the function to be called when the user clicks on the button)

Makes the component visible to outside modules

React Components manage their own *State*

```
import React, { useState
function Counter({initialCount, countName}) {
  const [count, setCount] = useState(initialCount);
```

You can add props

Initial value of the state

State variable

State change function name

```
  const increment = () => {
    setCount(count + 1);
  };
  return (
    <>
      <p>{countName}: {count}</p>
      <button onClick={increment}>
        Increment
      </button>
    </>
  );
} export default Counter;
```

Reference to the state variable. Changes whenever the state variable is changed via the state change function

Click event handler (the name of the function to be called when the user clicks on the button)

Makes the component visible to outside modules

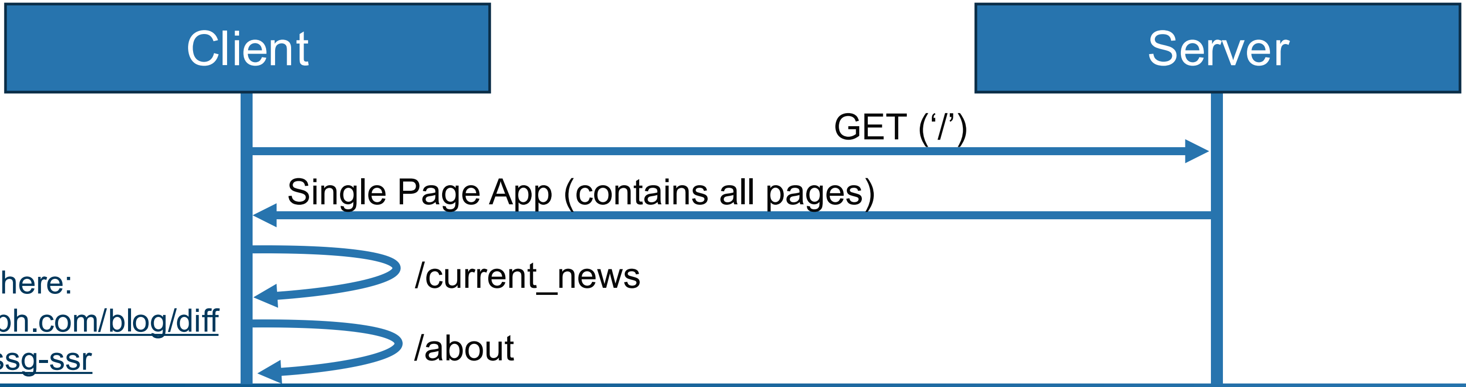
React employs a *Virtual DOM*

- Virtual DOM is a **lightweight representation** of the actual DOM.
- When data changes, React **first updates the Virtual DOM**, then efficiently **calculates the differences** with the real DOM and **applies only the necessary changes** to the real DOM.
- Minimizes direct DOM manipulation and **improves performance**

Single Page Application (SPA)

Most common in React

- SPAs are structured as a **single HTML page** that has no preloaded content.
- **Content is loaded dynamically via JavaScript** for the entire application and housed within a single HTML page.
- The JavaScript code houses all the data relating to the application logic, UI, and communication with the server

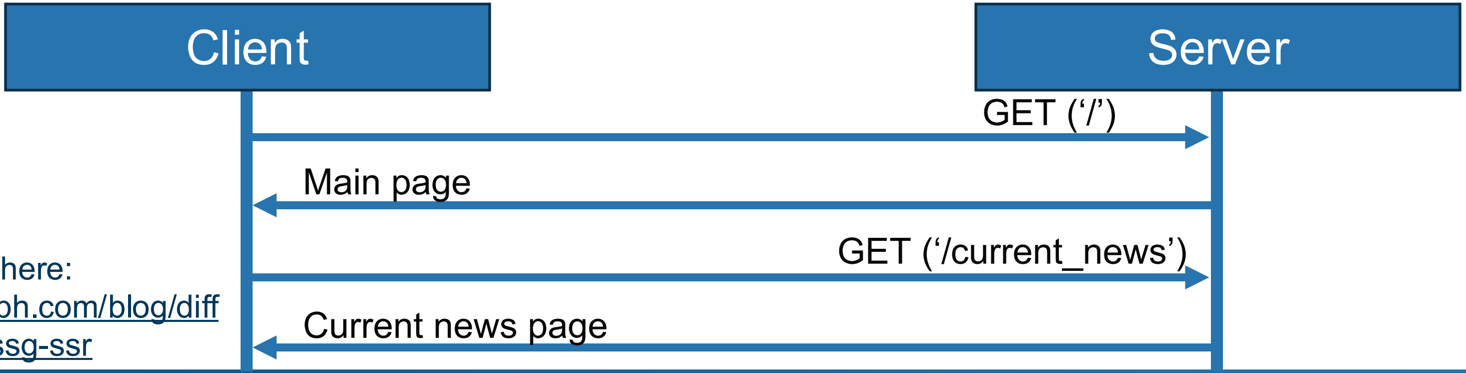


More details here: https://hygraph.com/blog/difference-spa-ssg-ssr

Server-Side Rendering (SSR)

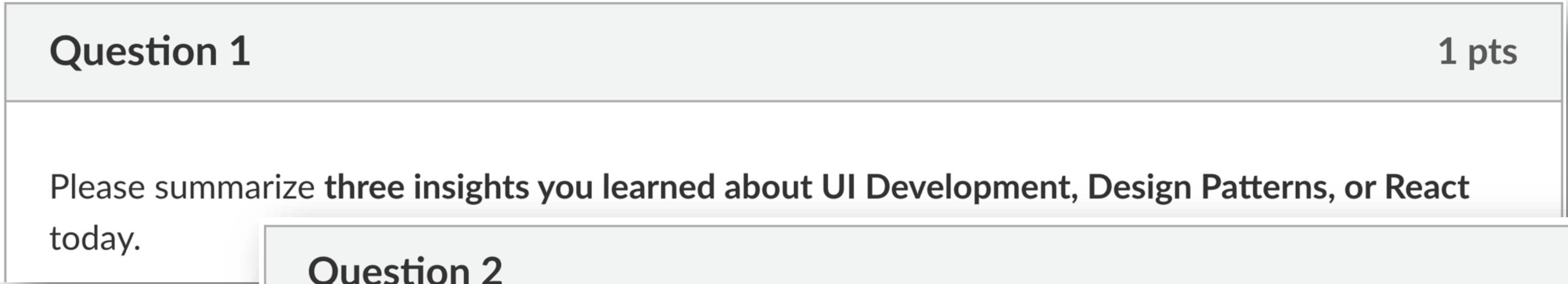
Frameworks like Next.js (which use React) do this

- Clients receive a fully rendered page (rendered on the server)
- Requires more **server compute**
- Faster **initial load time** (since only the current page is sent)
- Longer response time for **user interaction**
- Better search **engine optimization**



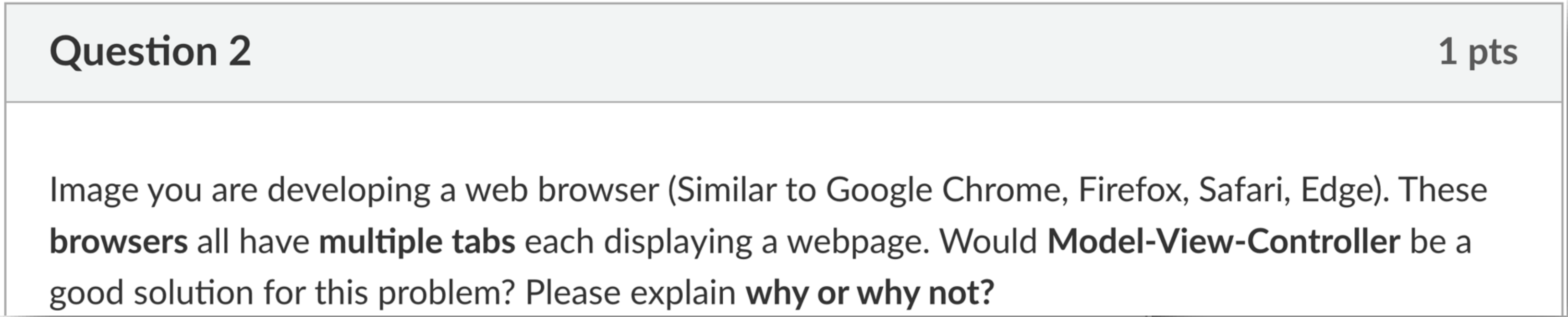
More details here: https://hygraph.com/blog/difference-spa-ssg-ssr

Please fill out your Exit Tickets on Bruin Learn!



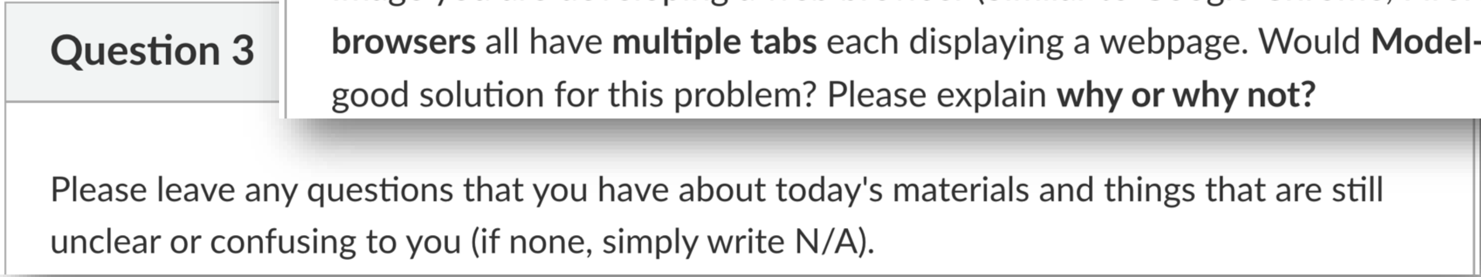
Question 1 1 pts

Please summarize **three insights you learned about UI Development, Design Patterns, or React** today.



Question 2 1 pts

Image you are developing a web browser (Similar to Google Chrome, Firefox, Safari, Edge). These **browsers** all have **multiple tabs** each displaying a webpage. Would **Model-View-Controller** be a good solution for this problem? Please explain **why or why not?**



Question 3

Please leave any questions that you have about today's materials and things that are still unclear or confusing to you (if none, simply write N/A).