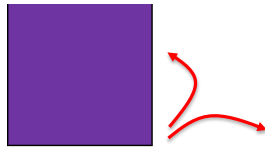


In the beginning...

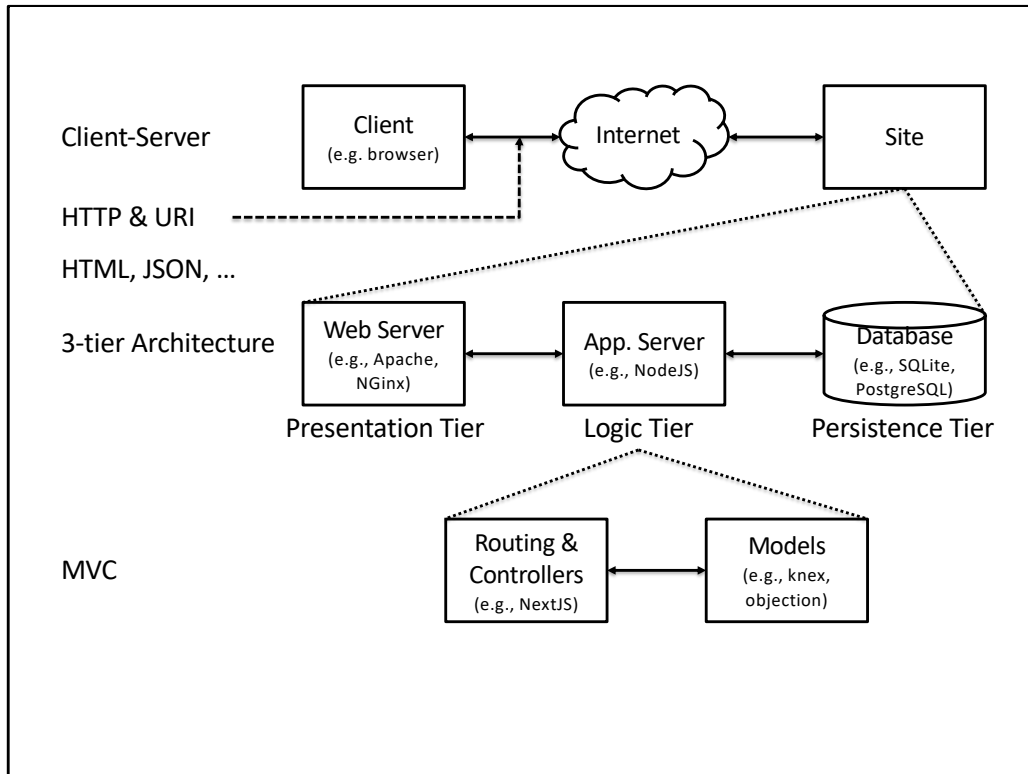


Red: 110
Green: 54
Blue: 162

```
<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);  
};
```

In the beginning, I suspect this was new to many of you... But we have now implemented this kind of interactivity and much more. In fact,...



Over the semester, we have learned about, used and often implemented components in every one of these boxes, from the JavaScript running on the front-end (i.e., the client) to the route handlers on the server, the database schema and more...

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

Callbacks and more callbacks!

```
const wrapValue = (n) => { // function(n) {  
  let local = n;  
  return () => local; // function () { return local; }  
}  
  
let wrap1 = wrapValue(1);    // () => 1  
let wrap2 = wrapValue(2);    // () => 2  
console.log(wrap1()); // What will print here?  
console.log(wrap2()); // What will print here?
```

Along the way we have gained familiarity with JavaScript (perhaps a new language) and its emphasis on closures and callbacks. The idea of functions as values (functions as 1st class objects), and asynchronous execution hopefully now feels familiar.

Asynchronous execution in action

<pre>function do(time, since) { return wait(time).then(() => { console.log(Date.now() - since); }); } const start = Date.now(); do(2, start); do(1, start).then(() => { do(4, start); }); do(3, start); console.log("end");</pre>	<pre>async function do(time, since) { await wait(time); console.log(Date.now() - since); } const start = Date.now(); do(2, start); do(1, start); do(4, start); await do(3, start); console.log("end");</pre>
<p>end 1000 2000 3000 5000</p>	<p>1000 2000 3000 end 4000</p>

Recall: Assume the function `wait(sec)` returns a promise that resolves after `sec` seconds have elapsed. Assume that every other operation is instantaneous (e.g., takes 0 milliseconds). Remember that `Date.now()` returns the number milliseconds elapsed since 12:00AM January 1, 1970, UTC.

Left (each on their own line): end 1000 2000 3000 5000

Right (each on their own line): 1000 2000 3000 end 4000

The key idea is to think about what sequencing relationships are introduced. Both `then` and `await` introduce a "before than" relationship, i.e., some code only executes when a promise resolves. On the left-hand side the `do` before relationship occurs between `do(1)` and `do(4)`, on the right-hand side it is between `do(3)` and `console.log("end")`.


Single source of truth!

```

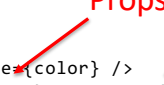
function ColorPicker() {
  const [red, setRed] = useState(0);
  const [green, setGreen] = useState(0);
  const [blue, setBlue] = useState(0);

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


```




Props down!



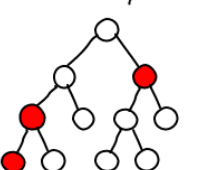
Callbacks up!



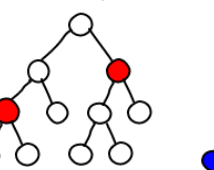
setState



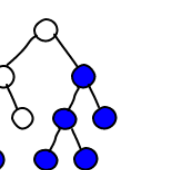
Dirty



Dirty



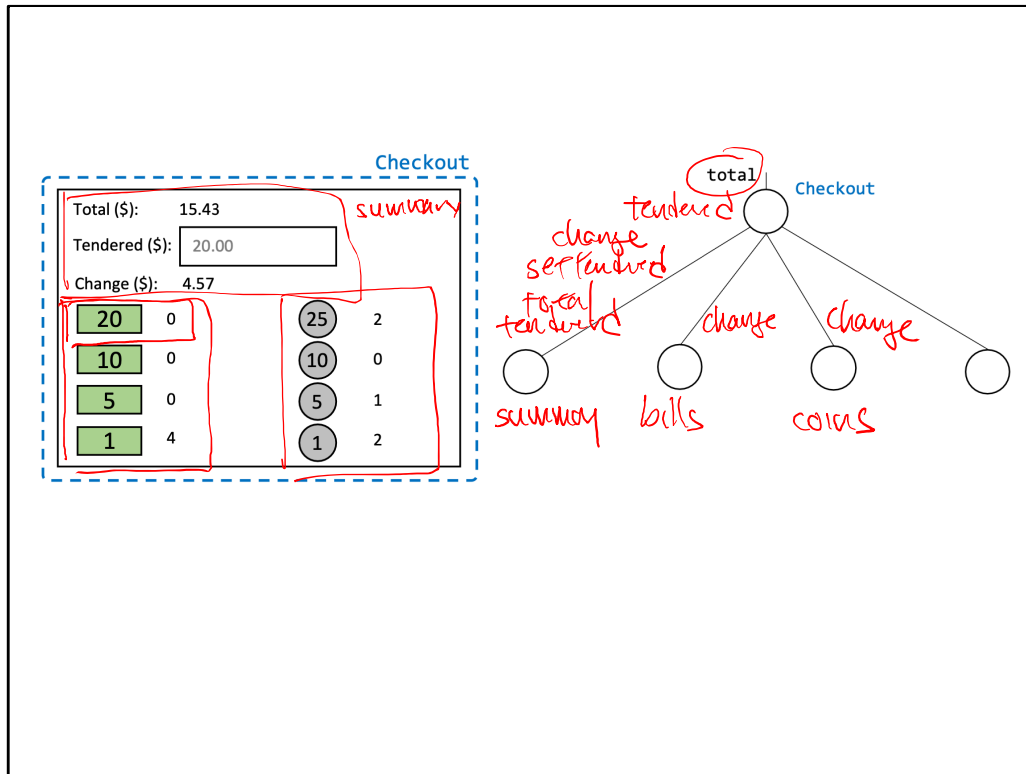
Re-rendered



We have made extensive use of React, and some its key principles:

- Maintain a single source of truth
- Props "flow" down
- Callbacks "flow" up

React implements a design pattern for highly interactive UIs. Your responsibility as the programmer is to define what you want rendered on the screen for a given state of the application and how you want to update that state in response to user actions. React fills in the "last piece" of the cycle by efficiently re-rendering the UI as the state changes.




You are implementing the Checkout component shown below with React. It receives a numeric prop representing the total purchase. Entering an amount paid (tendered) by the customer in cash computes the necessary amount of change and how the change should be provided, i.e., the number of bills and coins of the denominations shown. Outline and label the wireframe (below, top) with a possible set of components. Label the tree (below, bottom) with components to show the hierarchy. Label the tree nodes with state implemented in that component and label the tree edges with props passed to each component (similar to the figure in programming assignment 2). Repeated components can be labeled once in the tree. The top-level component and its prop(s) are labeled for you. Any implementation reflecting good React practices will be accepted. You may not need all the nodes in the tree or may need to add nodes depending on your design; cross out any unused nodes. Your component, state and prop names should be sufficiently descriptive that their role is clear..

Recall our design process:

- Identify components, with particular attention to repetition
- Identify the *minimum* state needed for the application. In this cases, we only need the tendered amount. Why don't so we need change? Because it can be derived from the total and the amount tendered.
- Where does that state live? Nearest common ancestor of the components that need the information, i.e., Checkout.

- Data flows down as props, information flows up as callbacks. Tendered flows down as a prop to the controlled input in the form, a callback “flows” that information back up.

Route	Controller Action
POST /api/films	Create new movie from request data
GET /api/films/:id	Read data of movie with id == :id
PUT /api/films/:id	Update movie with id == :id from request data
DELETE /api/films/:id	Delete movie with id == :id
GET /api/films	List (read) all movies



```
router.get(async (req, res) => {  
  const films = Film.query().withGraphFetched('genres');  
  res.status(200).send(films);  
});
```

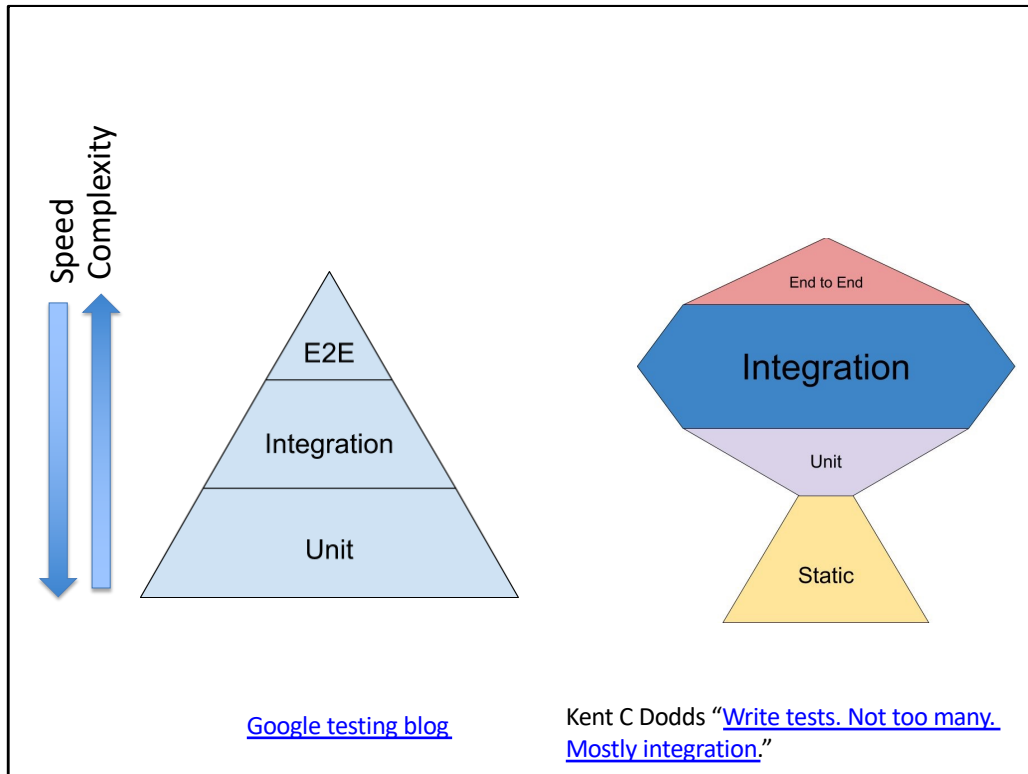
We learned about writing backend servers using Javascript to create RESTful APIs. RESTful APIs are built around the idea of actions on resources, i.e., the film resource in the film explorer example. The routes are typically implemented as queries to our persistence layer, in this example to a RDBMS via the Objection.js ORM. This particular query is using the "one-to-many" relation defined on the Film Objection.js model to automatically fetch the Genres associated with each of the films.

	Relational (RDBMS)	Non-Relational
Data	Table-oriented	Document-oriented, key-value, graph-based, column-oriented, ...
Schema	Fixed schema	Dynamic schema
Joins	Used extensively	Used infrequently
Interface	SQL	Custom query language
Transactions	ACID	CAP

`SELECT * FROM people
WHERE age > 25;`

`db.people.find(
 { age: { $gt: 25 } }
)`

We learned about different approaches to data persistence on the backend with a focus on relational databases, e.g., SQLite and PostgreSQL.



We learned about (but maybe didn't always practice...) test driven development (TDD) and concepts like unit testing and integration testing. We also talked about the use of behavior driven development (BDD) to move us from user story to scenario to test to implementation.

```

describe("Word game", () => {
  beforeEach(() => {
    Mock time
  });
  afterEach(() => {
    Clear mocks
  });
  test("Shows correct end of game after one incorrect guess", () => {
    Render WordGame - "hidden" as the secret prop
    guess list is empty
    Enter text guess1 and click Guess
    assert guess list contains "guess1 x"
    Advance mock time 3 s
    hidden
    Assert guess list and correct time

  });
});

```

You are developing a React component named WordGame for playing a word guessing game. The component takes the hidden word as a prop. The component provides a text input where the user can enter their guess and button "Guess" to check their guess. Incorrect guesses are displayed in a list below the input with an x appended. A correct guess is displayed at the end of the list with a ✓ appended. When the user guesses correctly the component displays the time elapsed since the game started (the component was mounted), e.g., "You guessed in 5.2s!". Using the skeleton below, implement pseudo-code for a F.I.R.S.T. unit test to verify that game correctly handles a correct guess after one incorrect guess. You do not need to provide executable Javascript, instead describe the steps of your test as pseudo-code.

A first step is to think about what you need to test:

- A satisfactory tests would assert an incorrect guess showed the expected result (guess with X) and then a correct guess showed the guess with a ✓ and the elapsed time.

Next, what ways do you need to isolate the component under test from the outside world (Independent and Repeatable in FIRST). This is where you will create mocks. Two common situations are mocking any callbacks that the React components expect as props and mocking interfaces to the outside world (e.g., network requests or time). Setting values for value props, form fields, etc. are not mocks. Instead, mocks are used to intercept or monitor interactions with other code/systems. We use mocks

to set known response values, etc.

- The design does not imply that this component expects a callback as a prop, instead as indicated, it is provided the secret word as a prop
- But it does rely on time and so we need some way to measure or control time. The most robust approach would be to control time by mocking Date (or more specifically using Jest's functions for setting and advancing time). We also need to clear any mocks to ensure repeatable tests.

Our test typically involves:

- Rendering the component with the relevant props, mock functions, etc. (Arrange)
- Finding and setting any inputs, e.g., guess input. (Act)
- Find and clicking any buttons, e.g., Guess button (Act)
- Make assertions about the UI, mocks. Here we want to assert the result is displayed with the expected following character. (Assert)

And we want to do that for an incorrect guess and then a correct guess, also asserting on the correct time after the correct guess.

Render the WordGame component with prop "HiddenWord"

Find the guesses list and assert it is empty

Find text input and enter "input1"

Find and click the "Guess" button

Find the guesses list and assert it is ["input1\$\times\$"]

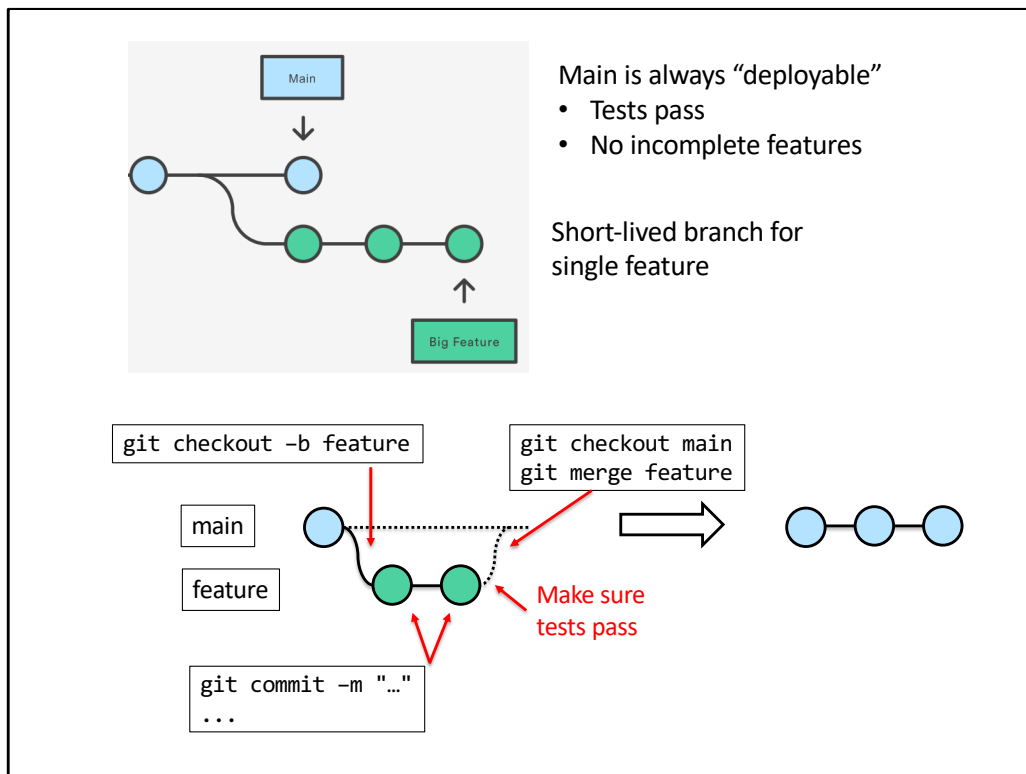
Advance mock time 3s

Find text input and enter "HiddenWord"

Find and click the "Guess" button

Find the guesses list and assert it is ["input1\$\times\$", "HiddenWord\$\checkmark\$"]

Find and assert the presence of "You guessed in 3s!" message



We learned about using git to manage development, and the value of continuous integration (CI). CI rigorously tests (we hope) every integration in production-like environment, the motivation is to:

- Prevent development-production mismatch
- Test multiple browsers, etc.
- “Stress test” code for performance, fault-tolerance, etc.

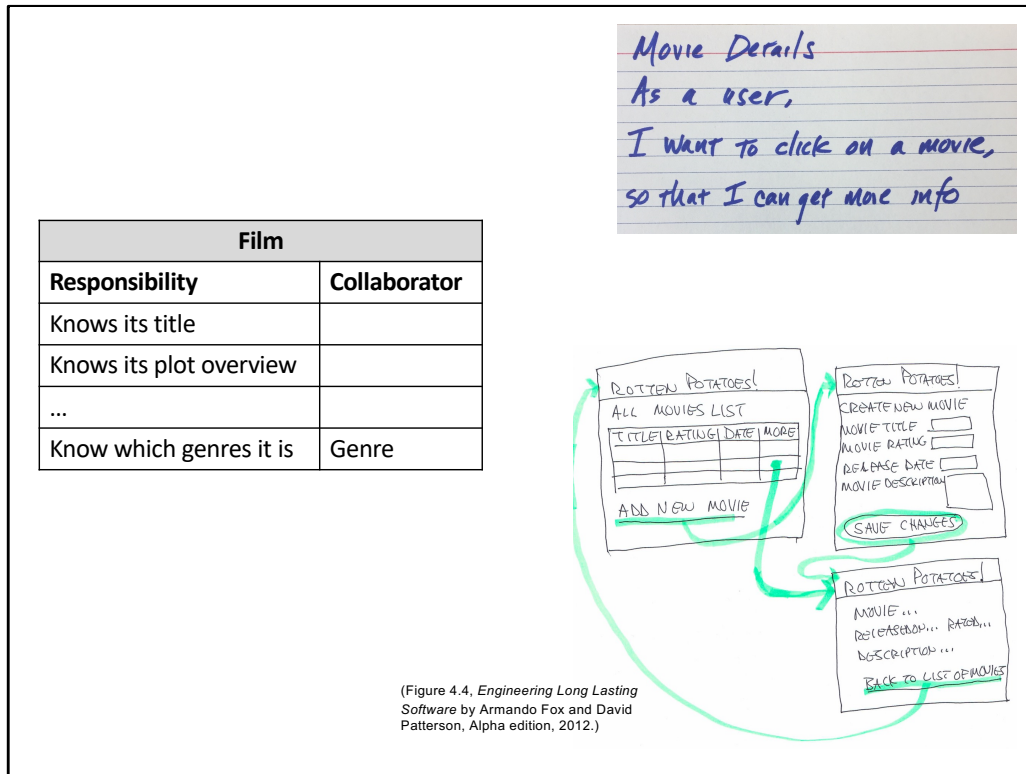
CI is part of a larger DevOps approach.

Recall the DevOps principles:

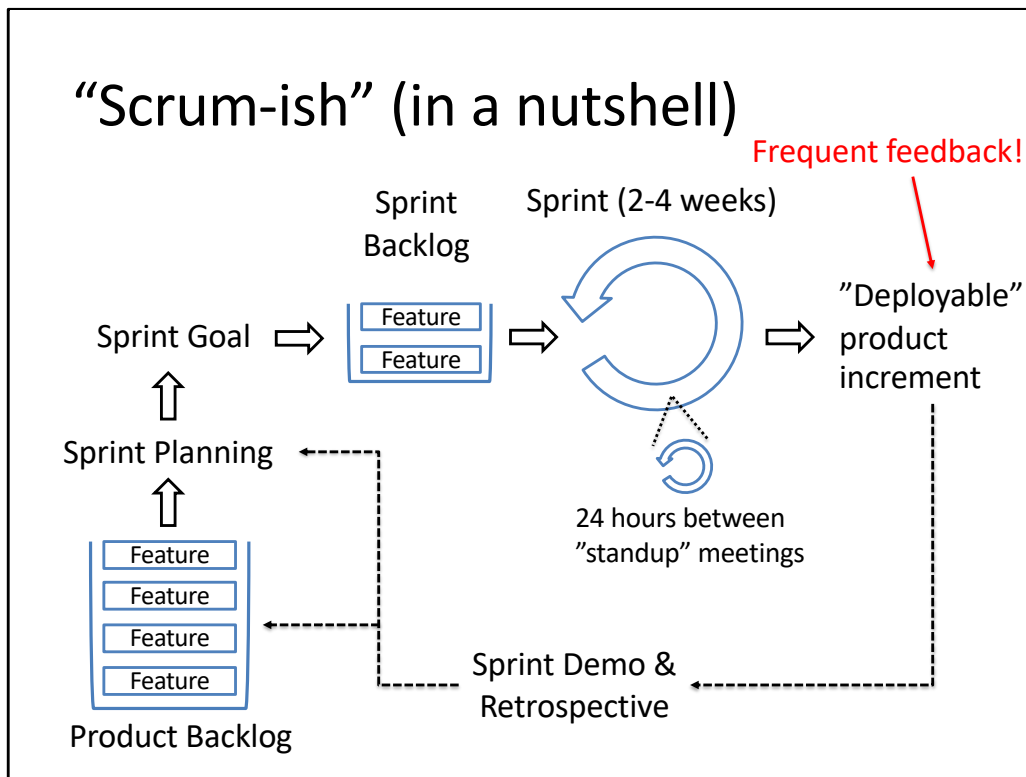
- Involve operations in each phase of a system’s design and development,
- Heavy reliance on automation versus human effort,
- The application of engineering practices and tools to operations tasks

This manifested for us in our use of GitHub actions to test our builds, and automated preparation (e.g. running migrations/seeding) and ultimately deployment with csci312.dev.

<https://www.atlassian.com/git/tutorials/using-branches>



You also learned about ways to approach design from user stories to CRC cards to lo-fi prototypes. Our goal is to be able to iterate on our design quickly and cheaply. These "lo-tech" tools are intended to facilitate conversations with our stakeholders, e.g., customers.



Agile development processes, Scrum in particular, played an important role for us. In Scrum the short sprints provide frequent opportunities to update our approach in response to what we have learned about the problem or the application. Recall the key idea of lower-case "a" agility:

1. Find out where you are,
2. Take a small step towards your goal,
3. Adjust your understanding based on what you learned, and
4. Repeat

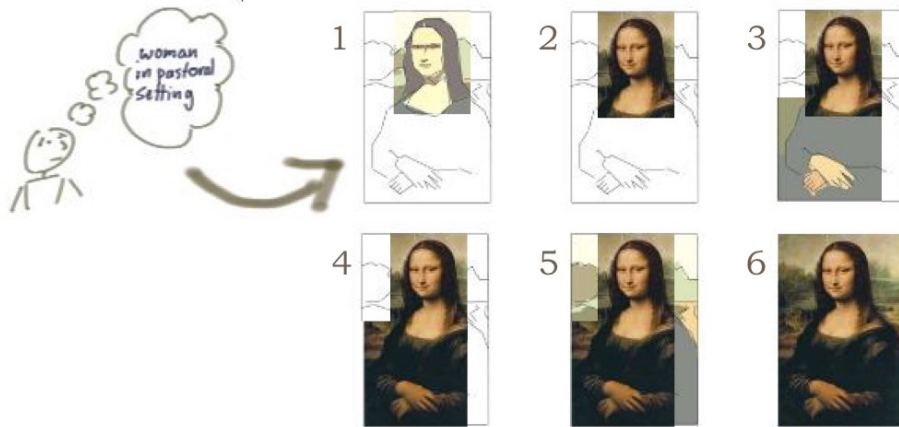
And when faced with two or more alternatives that deliver roughly the same value, choose the path that makes future change easier

Adapted from Mountain Goat Software

<https://www.mountaingoatsoftware.com/uploads/presentations/Getting-Agile-With-Scrum-Norwegian-Developers-Conference-2014.pdf>

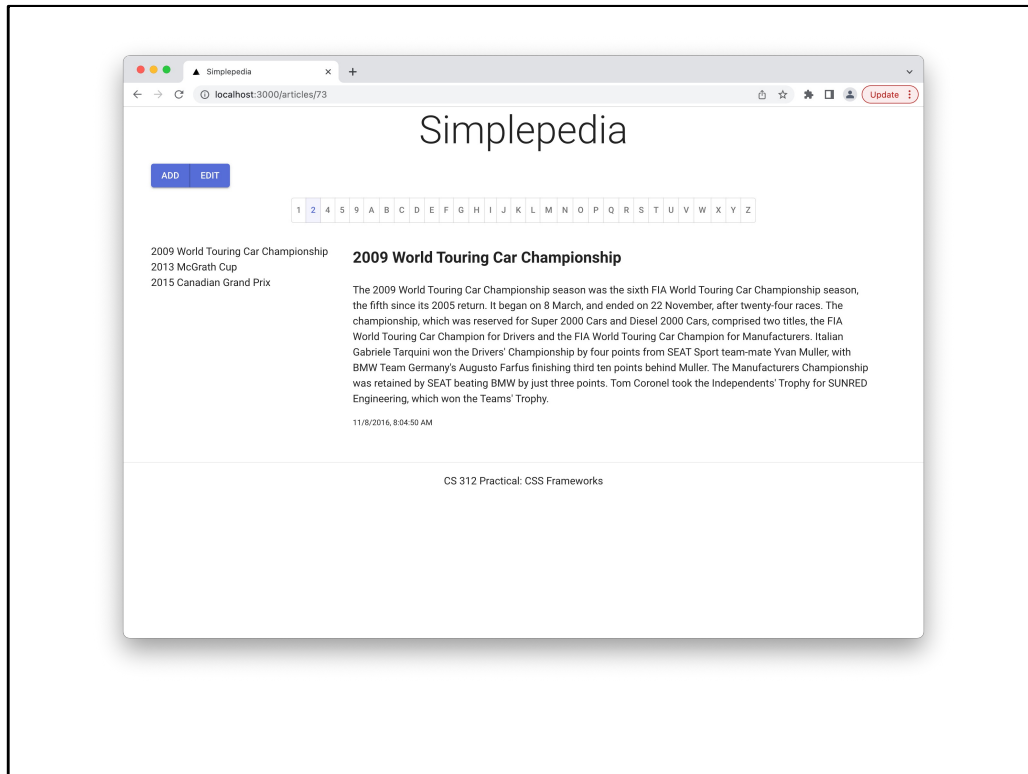
Adapted from Dave Thomas (<https://www.youtube.com/watch?v=a-BOSpxYJ9M>)

Iterative Incremental



<http://itsadeliverything.com/revisiting-the-iterative-incremental-mona-lisa>

And we thought about how to approach building our own "Mona Lisa"s. Recall that an incremental approach calls for building a fully formed idea a bit at time, and thus requires having a fully formed idea. In contrast, iterating allows you to move from vague idea to realization. The catch is we have to address the entire scope at one time, i.e., we are working on the entire image (a tricky and risky approach). Thus, we sought the of the best of both where in each sprint we both add new features (incremental) and refine existing functionality (iterative) with strong focus the highest priority features (the "face").



Along the way, you have completed four assignments and 10 practical exercises yielding a Wikipedia-like platform with database persistence and user authentication. And took a midterm with 8 learning targets and reviewed 443+ slides!

Between the assignments, practical and the project, you/we have created 491 repositories in our GitHub organization. On the just the main branches of your projects (as of yesterday) you have made 736 commits (and I suspect there are many more in total) and many, many, branches (I didn't even want to try to count!)

328+159+48+118+83

SMALL (So Many Acronyms Littering the Lectures)

- F.I.R.S.T.
- I.N.V.E.S.T.
- R.A.S.P.
- DRY
- SoC
- SOFA
- SOLID
- SaaS
- TDD, BDD
- MVC
- WISBNWIW
- ACID
- CRUD(L)
- HTML / DOM / CSS / JSX
- CI / CD
- UI
- AJAX
- REST API
- URI / URL
- TCP/IP
- JSON
- CRC
- ORM
- POJO
- SQL / RDBMS
- VCS
- SLO / SLA

And the 30+ different acronyms... I hope these (and especially those on the left) will help you remember the keys ideas when writing a test, or a user story or debugging.

The one I struggle with always is RASP. Read the error message, Ask a colleague an informed question, Search using a keyword, Post on Stack Overflow, etc.

Recall ACID is set of properties for database transaction: Atomicity, Consistency, Isolation, and Durability

Commandments for being a bad SW team player (and some alternatives)

- | | |
|--|--|
| 1. Those fails don't matter | 1. Never push failing tests |
| 2. My branches, my sanctuary | 2. Have short-lived branches by integrating frequently |
| 3. It's just a simple change | 3. Test everything |
| 4. I am a special snowflake | 4. One coding style |
| 5. Cleverness is impressive | 5. Transparency is humble |
| 6. Just change it quickly on the production server | 6. Make every change automatable |
| 7. Time spent looking stuff up is wasted time (not coding) | 7. Spend 5 minutes searching for less or better code |
| 8. "Green fever": Catch it! | 8. More tests \neq higher quality |
| 9. Weeks of coding can save hours of planning & thought | 9. Work through your design |
| 10. When blocked, I am stuck | 10. I unblock myself, or move on to the next task |

I suspect these "anti-patterns" resonate now as we are in the thick of the project and than they did before!

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

Take-aways

- Behind every design decision there should be a user story (a stakeholder and a motivation!)
- Testing, not just a class requirement, it's a good idea
- Develop iteratively *and* incrementally
- There should be one source of truth
- Don't repeat yourself
- Don't mutate props or state
- Do really read error messages (and the docs!)
- Automate all the things
- Don't break the "Build"
- Program strategically, not tactically



earthcam.com

Write beautiful code, do the Right Thing, give your Statue of Liberty hair!

[go/crf](#)
[go/cshelp-evals](#)