Code-Layout,Readability And Reusability

Team Id	NM2023TMID04410
Project Name	Project-Drug Traceability

Code-Layout

Readability:

Readability is the simplest way of assessing code quality and it's the most straightforward to fix. It is the most obvious thing you see right when you open up a piece of code, and it generally consists of:

* Formatting

* Variable names

* Function names

* Amount of function arguments

* Function length (number of lines)

* Nesting levels

These aren't the only things to consider, but they are immediate red flags. Fortunately, there are a few easy rules to follow to fix problems associated with those above:

* Invest in an automatic formatter. Find one your team agrees on and integrate it into your build process. There's nothing that wastes more time and money during code reviews than formatting arguments. Get a formatter and never look back! In this project we will use Prettier.

* Use meaningful and pronounceable variable/function names. Code is for people, and only incidentally for computers. Naming is the biggest thing that communicates the meaning behind your code.

* Limit your function arguments to between 1-3. 0 arguments implies you're mutating state or relying on state coming from somewhere else other than your caller. More than 3 arguments is just plain hard

to read and refactoring it is difficult because there are so many paths your function can take the more arguments it has.

* There is no set limit of lines for a function, as this depends on what particular language you are coding in. The main point is that your function should do ONE thing, and ONE thing only. If your function, which calculates the price of an item after taxes, first has to connect to the database, look up the item, get the tax data, and then do the calculation, then it's clearly doing more than one thing. Long functions typically indicate too much is happening.

* More than two levels of nesting can imply poor performance (in a loop), and it can be especially hard to read in long conditionals. Consider extracting nested logic into separate functions.

Let's take a look at this first piece of our shopping cart application, to see what bad readability looks like:

```javascript
// src/1-readable/bad/index.js
import React, { Component } from 'react';

// Inventory
class inv extends Component
{
 constructor()
 {
  super();

  // State
  this.state =
  {
   c: 'usd', // currency
   i: [ // inventory
    {
     product: 'Flashlight',
```

```
      img: '/flashlight.jpg',
      desc: "A really great flashlight",
      price: 100,
      id: 1,
      c: 'usd'
    },
    {
      product: 'Tin can',
      img: '/tin_can.jpg',
      desc: "Pretty much what you would expect from a tin can",
      price: 32,
      id: 2,
      c: 'usd'
    },
    {
      product: 'Cardboard Box',
      img: '/cardboard_box.png',
      desc: "It holds things",
      price: 5,
      id: 3,
      c: 'usd'
    }
  ]
 }
}

render () {
 return (
  <table style={{width: '100%'}}>
```

```
<tbody>
<tr>
<th>
 Product
</th>

<th>
 Image
</th>

<th>
 Description
</th>

<th>
 Price
</th>
</tr>
 {this.state.i.map(function(i, idx) {
   return (
     <tr key = {idx}>
       <td>
         {i.product}
       </td>

       <td>
       <img src={i.img} alt=""/>
       </td>
```

```
            <td>
            {i.desc}
            </td>

            <td>
              {i.price}
            </td>
          </tr>
        );
      })}
   </tbody>
   </table>
   );
 }
}

export default inv;
```

There are a number of problems we can see right away:

* Inconsistent and unpleasant formatting
* Poorly named variables
* Disorganized data structures (inventory not keyed by IDs)
* Comments that are either unnecessary or serve the job of what a good variable name would

Let's take a look at how we could improve it:

```javascript
// src/1-readable/good/index.js
import React, { Component } from 'react';
```

```
export default class Inventory extends Component {
  constructor() {
    super();
    this.state = {
      localCurrency: 'usd',
      inventory: {
        1: {
          product: 'Flashlight',
          img: '/flashlight.jpg',
          desc: 'A really great flashlight',
          price: 100,
          currency: 'usd',
        },
        2: {
          product: 'Tin can',
          img: '/tin_can.jpg',
          desc: 'Pretty much what you would expect from a tin can',
          price: 32,
          currency: 'usd',
        },
        3: {
          product: 'Cardboard Box',
          img: '/cardboard_box.png',
          desc: 'It holds things',
          price: 5,
          currency: 'usd',
        },
      },
    };
```

```
}

render() {
  return (
    <table style={{ width: '100%' }}>
      <tbody>
        <tr>
          <th>
            Product
          </th>

          <th>
            Image
          </th>

          <th>
            Description
          </th>

          <th>
            Price
          </th>
        </tr>
        {Object.keys(this.state.inventory).map(itemId => (
          <tr key={itemId}>
            <td>
              {this.state.inventory[itemId].product}
            </td>
```

```
            <td>
              <img src={this.state.inventory[itemId].img} alt="" />
            </td>

            <td>
              {this.state.inventory[itemId].desc}
            </td>

            <td>
              {this.state.inventory[itemId].price}
            </td>
          </tr>
        ))}
      </tbody>
    </table>
  );
 }
}
```

This improved code now exhibits the following features:

* It is consistently formatted using the automatic formatter Prettier

* Names are much more descriptive

* Data structures are properly organized. In this case the Inventory is keyed by ID. Bad readability can mean bad performance. If we had wanted to get an item from our inventory in our bad code example we would have had an O(n) lookup time but with Inventory keyed by ID we get an O(1) lookup, which is MUCH faster with large inventories.

* Comments are no longer needed because good naming serves to clarify the meaning of the code. Comments are needed when business logic is complex and when documentation is required.

Reusability:

Reusability is the sole reason you are able to read this code, communicate with strangers online, and even program at all. Reusability allows us to express new ideas with little pieces of the past.

That is why reusability is such an essential concept that should guide your software architecture. We commonly think of reusability in terms of DRY (Don't Repeat Yourself). That is one aspect of it -- don't have duplicate code if you can abstract it properly. Reusability goes beyond that though. It's about making clean, simple APIs that make your fellow progammer say, "Yep, I know exactly what that does!" Reusability makes your code a delight to work with, and it means you can ship features faster.

We will look at our previous example and expand upon it by adding a currency converter to handle our inventory's pricing in multiple countries:

```javascript
// src/2-reusable/bad/index.js
import React, { Component } from 'react';

export default class Inventory extends Component {
  constructor() {
    super();
    this.state = {
      localCurrency: 'usd',
      inventory: {
        1: {
          product: 'Flashlight',
          img: '/flashlight.jpg',
          desc: 'A really great flashlight',
          price: 100,
          currency: 'usd',
        },
        2: {
          product: 'Tin can',
```

```
      img: '/tin_can.jpg',
      desc: 'Pretty much what you would expect from a tin can',
      price: 32,
      currency: 'usd',
    },
    3: {
      product: 'Cardboard Box',
      img: '/cardboard_box.png',
      desc: 'It holds things',
      price: 5,
      currency: 'usd',
    },
  },
};

this.currencyConversions = {
  usd: {
    rupee: 66.78,
    yuan: 6.87,
    usd: 1,
  },
};

this.currencySymbols = {
  usd: '$',
  rupee: 'â1',
  yuan: 'å',
};
}
```

```
onSelectCurrency(e) {
  this.setState({
    localCurrency: e.target.value,
  });
}

convertCurrency(amount, fromCurrency, toCurrency) {
  const convertedCurrency = amount *
    this.currencyConversions[fromCurrency][toCurrency];
  return this.currencySymbols[toCurrency]  convertedCurrency;
}

render() {
  return (
    <div>
      <label htmlFor="currencySelector">Currency:</label>
      <select
        className="u-full-width"
        id="currencySelector"
        onChange={this.onSelectCurrency.bind(this)}
        value={this.state.localCurrency}
      >
        <option value="usd">USD</option>
        <option value="rupee">Rupee</option>
        <option value="yuan">Yuan</option>
      </select>
      <table style={{ width: '100%' }}>
        <tbody>
```

```
<tr>
  <th>
    Product
  </th>

  <th>
    Image
  </th>

  <th>
    Description
  </th>

  <th>
    Price
  </th>
</tr>

{Object.keys(this.state.inventory).map(itemId => (
  <tr key={itemId}>
    <td>
      {this.state.inventory[itemId].product}
    </td>

    <td>
      <img src={this.state.inventory[itemId].img} alt="" />
    </td>

    <td>
```

```
              {this.state.inventory[itemId].desc}
            </td>

            <td>
              {this.convertCurrency(
                this.state.inventory[itemId].price,
                this.state.inventory[itemId].currency,
                this.state.localCurrency,
              )}
            </td>
          </tr>
        ))}
      </tbody>
    </table>
  </div>
 );
 }
}
```

This code works, but merely working is not the point of code. That's why we need to look at this with a stronger lens than just analyzing if it works and it's readable. We have to look if it's reusable. Do you notice any issues?

Think about it!

Alright, there are 3 main issues in the code above:

* The Currency Selector is coupled to the Inventory component

* The Currency Converter is coupled to the Inventory component

* The Inventory data is defined explicitly in the Inventory component and this isn't provided to the component in an API.

Every function and module should just do one thing, otherwise it can be very difficult to figure out what is going on when you look at the source code. The Inventory component should just be for displaying an inventory, not converting and selecting currencies. The benefit of making modules and functions do one thing is that they are easier to test and they are easier to reuse. If we wanted to use our Currency Converter in another part of the application, we would have to include the whole Inventory component. That doesn't make sense if we just need to convert currency.

Let's see what this looks like with more reusable components:

```javascript
// src/2-reusable/good/currency-converter.js
export default class CurrencyConverter {
  constructor(currencyConversions) {
    this.currencyConversions = currencyConversions;
    this.currencySymbols = {
      usd: '$',
      rupee: 'â',
      yuan: 'å',
    };
  }

  convert(amount, fromCurrency, toCurrency) {
    const convertedCurreny = amount *
      this.currencyConversions[fromCurrency][toCurrency];
    return this.currencySymbols[toCurrency]  convertedCurreny;
  }
}
```

```javascript
// src/2-reusable/good/currency-selector.js

import React, { Component } from 'react';

class CurrencySelector extends Component {
  constructor(props) {
    super();
    this.props = props;
    this.state = {
      localCurrency: props.localCurrency.currency,
    };

    this.setGlobalCurrency = props.setGlobalCurrency;
  }

  onSelectCurrency(e) {
    const currency = e.target.value;

    this.setGlobalCurrency(currency);

    this.setState({
      localCurrency: currency,
    });
  }

  render() {
    return (
      <div>
```

```
      <label htmlFor="currencySelector">Currency:</label>
      <select
        className="u-full-width"
        id="currencySelector"
        onChange={this.onSelectCurrency.bind(this)}
        value={this.state.localCurrency}
      >
        <option value="usd">USD</option>
        <option value="rupee">Rupee</option>
        <option value="yuan">Yuan</option>
      </select>
    </div>
  );
 }
}

CurrencySelector.propTypes = {
 setGlobalCurrency: React.PropTypes.func.isRequired,
 localCurrency: React.PropTypes.string.isRequired,
};

export default CurrencySelector;
```

```javascript
// src/2-reusable/good/inventory.js

import React, { Component } from 'react';
```

```
class Inventory extends Component {
  constructor(props) {
    super();
    this.state = {
      localCurrency: props.localCurrency,
      inventory: props.inventory,
    };

    this.CurrencyConverter = props.currencyConverter;
  }

  componentWillReceiveProps(nextProps) {
    this.setState({
      localCurrency: nextProps.localCurrency,
    });
  }

  render() {
    return (
      <div>
        <table style={{ width: '100%' }}>
          <tbody>
            <tr>
              <th>
                Product
              </th>

              <th>
                Image
```

```
  </th>

  <th>
    Description
  </th>

  <th>
    Price
  </th>
</tr>

{Object.keys(this.state.inventory).map(itemId => (
  <tr key={itemId}>
    <td>
      {this.state.inventory[itemId].product}
    </td>

    <td>
      <img src={this.state.inventory[itemId].img} alt="" />
    </td>

    <td>
      {this.state.inventory[itemId].desc}
    </td>

    <td>
      {this.CurrencyConverter.convert(
        this.state.inventory[itemId].price,
        this.state.inventory[itemId].currency,
```

```
                this.state.localCurrency,
              )}
            </td>
          </tr>
        ))}
      </tbody>
    </table>
  </div>
 );
 }
}

Inventory.propTypes = {
 inventory: React.PropTypes.object.isRequired,
 currencyConverter: React.PropTypes.object.isRequired,
 localCurrency: React.PropTypes.string.isRequired,
};

export default Inventory;
```

```javascript
// src/2-reusable/good/index.js

import React, { Component } from 'react';
import CurrencyConverter from './currency-converter';
import Inventory from './inventory';
import CurrencySelector from './currency-selector';
```

```
export default class ReusableGood extends Component {
  constructor() {
    super();

    this.inventory = {
      1: {
        product: 'Flashlight',
        img: '/flashlight.jpg',
        desc: 'A really great flashlight',
        price: 100,
        currency: 'usd',
      },
      2: {
        product: 'Tin can',
        img: '/tin_can.jpg',
        desc: 'Pretty much what you would expect from a tin can',
        price: 32,
        currency: 'usd',
      },
      3: {
        product: 'Cardboard Box',
        img: '/cardboard_box.png',
        desc: 'It holds things',
        price: 5,
        currency: 'usd',
      },
    };

    // Most likely we would fetch this from an external source if this were a real app
```

```
  this.currencyConversions = {
    usd: {
      rupee: 66.78,
      yuan: 6.87,
      usd: 1,
    },
  };

  this.state = {
    localCurrency: 'usd',
  };

  this.setGlobalCurrency = (currency) => {
    this.setState({
      localCurrency: currency,
    });
  };
}

render() {
  return (
    <div>
      <CurrencySelector
        setGlobalCurrency={this.setGlobalCurrency}
        localCurrency={this.state.localCurrency}
      />
      <Inventory
        inventory={this.inventory}
        currencyConverter={new CurrencyConverter(this.currencyConversions)}
```

```
        localCurrency={this.state.localCurrency}
      />
    </div>
  );
 }
}
```

This code has improved a great deal. Now we have individual modules for currency selection and conversion. Moreover, we can now provide the inventory data to our Inventory component. That means that we could download the inventory data, for example, and provide it to the Inventory component without ever having to modify its source code. This decoupling is the Dependency Inversion Principle, and it's a powerful way of creating reusable code.

Now, it's time for a bit of caution. Before diving in and making everything reusable, it's important to realize that reusability requires that you have a good API for others to consume. If you don't, then whoever uses your API could be hurt when you go to update it because you realize it wasn't thought out well enough. So, when should code NOT be reusable?

* If you can't define a good API yet, don't make a separate module. Duplication is better than a bad foundation.

* You don't expect to reuse your function or module in the near future.

This flow makes it possible to be sure that the state of your application can only be updated in one way, and that's through the _action_ -> _reducer_ -> _store_ -> _component_ pipeline. There's no global state to modify, no messages to pass and keep track of, and no uncontrolled side effects that our modules can produce. The best part is, we can keep track of the entire state of our application so debugging and QA can become much easier, because we have an exact snapshot in time of our entire application.

One caveat to note: you might not need Redux in this project's example application, but if we were to expand this code it would become easier to use Redux as the state management solution instead of putting everything in the top-level controller `index.js`. We could have isolated the state of our app there and passed the appropriate data-modifying action functions down through each module. The issue with that is that at scale, we would have a lot of actions to pass down and a lot of data that would

live in one massive `index.js` controller. By committing to a proper centralization of state early, we won't need to change much as our application develops.

The last thing we need to look at is tests. Tests give us confidence that we can change a module and it will still do what it was intended to do. We will look at the tests for the Cart and Inventory components:

```javascript
// src/test/cart.test.js
import React from 'react';
import { shallow } from 'enzyme';
import Cart from '../src/3-refactorable/good/components/cart';

const props = {
  localCurrency: 'usd',
  cart: [1, 1],
  inventory: {
    1: {
      product: 'Flashlight',
      img: '/flashlight.jpg',
      desc: 'A really great flashlight',
      price: 100,
      currency: 'usd',
    },
  },
  currencyConverter: {
    convert: jest.fn(),
  },
};
```

```
it('should render Cart without crashing', () => {
  const cartComponent = shallow(<Cart {...props} />);
  expect(cartComponent);
});

it('should show all cart data in cart table', () => {
  props.currencyConverter.convert = function () {
    return `$${props.inventory[1].price}`;
  };

  const cartComponent = shallow(<Cart {...props} />);
  let tr = cartComponent.find('tr');
  expect(tr.length).toEqual(3);

  props.cart.forEach((item, idx) => {
    let td = cartComponent.find('td');
    let product = td.at(2 * idx);
    let price = td.at(2 * idx  1);

    expect(product.text()).toEqual(props.inventory[item].product);
    expect(price.text()).toEqual(props.currencyConverter.convert());
  });
});
```

```javascript
// src/test/inventory.test.js
import React from 'react';
import { shallow } from 'enzyme';
```

```
import Inventory from '../src/3-refactorable/good/components/inventory';

const props = {
  localCurrency: 'usd',
  inventory: {
    1: {
      product: 'Flashlight',
      img: '/flashlight.jpg',
      desc: 'A really great flashlight',
      price: 100,
      currency: 'usd',
    },
  },
  addToCart: jest.fn(),
  changeCurrency: jest.fn(),
  currencyConverter: {
    convert: jest.fn(),
  },
};

it('should render Inventory without crashing', () => {
  const inventoryComponent = shallow(<Inventory {...props} />);
  expect(inventoryComponent);
});

it('should show all inventory data in table', () => {
  props.currencyConverter.convert = function () {
    return `$${props.inventory[1].price}`;
  };
```

```
  const inventoryComponent = shallow(<Inventory {...props} />);
  let tr = inventoryComponent.find('tr');
  expect(tr.length).toEqual(2);

  let td = inventoryComponent.find('td');
  let product = td.at(0);
  let image = td.at(1);
  let desc = td.at(2);
  let price = td.at(3);

  expect(product.text()).toEqual('Flashlight');
  expect(image.html()).toEqual('<td><img src="/flashlight.jpg" alt=""/></td>');
  expect(desc.text()).toEqual('A really great flashlight');
  expect(price.text()).toEqual('$100');
});

it('should have Add to Cart button work', () => {
  const inventoryComponent = shallow(<Inventory {...props} />);
  let addToCartBtn = inventoryComponent.find('button').first();
  addToCartBtn.simulate('click');
  expect(props.addToCart).toBeCalled();
});
```