CSE 331

Mutable Heap State

Katherine Murphy
## Correctness Levels

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
<th>Testing</th>
<th>Tools</th>
<th>Reasoning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>small # of inputs</td>
<td>exhaustive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>straight from spec</td>
<td>heuristics</td>
<td>type checking</td>
<td>code reviews</td>
</tr>
<tr>
<td>2</td>
<td>no mutation</td>
<td>“”</td>
<td>libraries</td>
<td>calculation induction</td>
</tr>
<tr>
<td>3</td>
<td>local variable mutation</td>
<td>“”</td>
<td>“”</td>
<td>Floyd logic</td>
</tr>
<tr>
<td>4</td>
<td>array mutation</td>
<td>“”</td>
<td>“”</td>
<td>for-any facts</td>
</tr>
<tr>
<td>5</td>
<td>heap state mutation</td>
<td>“”</td>
<td>“”</td>
<td>alias tracking rep invariants</td>
</tr>
</tbody>
</table>
Mutable Heap State

• “With great power, comes great responsibility”

• With arrays:
  – gain the ability to easily access any element
  – must keep track of information about the whole array

• Additional references to the same object are “aliases”

• With mutable heap state:
  – gain efficiency in some cases
  – must keep track of every alias that could mutate that state
    any alias, anywhere in the entire program could cause a bug
Heap State

- "Heap state" = lives on after the call stack finishes
  - after current function and those calling it all return
  - state could be arrays or records

- No different from before when immutable
  - we don’t care when the garbage collectors gets rid of it

- Vastly more complex when mutable...
Scary Bugs

• **Do not** fear crashes
  – those are easy to spot and fix
    get a stack trace that tells you exactly where it went wrong

• **Do** fear unexpected mutation
  – failure will give you no clue what went wrong
    will take a long time to realize the BST invariant was violated by mutation
  – *bug could be almost anywhere in the code*
    anyone who mutates an object could have caused it
  – could take *weeks* to track it down
Easy Ways to Stay Safe

1. Do not use mutable state
   – don’t need to think about aliasing at all
   – any number of aliases is fine

2. Do not allow aliases
   a) do not hand out aliases yourself
   b) make a copy of anything you want to keep

   ensures only one reference to the object (no aliases)

• For 331, allowing aliases is a bug! (“rep exposure”)
  – gives the client the ability to break your code
  – we will stick to these simple strategies for avoiding it
An Advanced (Two-Stage) Approach

• Mutable object has only one reference (**owner**)
  – one reference that is allowed to use & mutate it

• Object is eventually “frozen”, making it immutable
  – no longer necessary to track ownership

• **Example:** Java’s **StringBuilder** vs **String**
  – StringBuilder is mutable (be careful!)
  – StringBuilder.toString returns the value as a String
  – String is immutable
Language Features & Aliasing

• Most recent languages have some answer to this...

• Java chose to make \texttt{String} immutable
  – most keys in maps are strings
  – hugely controversial at the time, but great decision

• Python chose to only allow immutable keys in maps
  – only numbers, strings, and tuples allowed
  – surprisingly, not that inconvenient

• Rust has built-in support for tracking ownership
  – ownership can be “borrowed” and returned
  – type system ensures there is only one usable alias
Avoiding Representation Exposure

• Prevent aliasing of mutable state
  – otherwise, code outside your class can break it

• Options for avoiding representation exposure:

  1. Use immutable types
     lists are immutable, so you can freely accept and return them

  2. Copy In, Copy Out
     store copies of mutable values passed to you
     return copies of not aliases to mutable state
     don’t take their word that they haven’t kept an alias

• Professionals are untrusting about aliases
Need for Mutable Heap State

• Saw that mutable heap state is complex
  – better to avoid when possible

• Cannot be avoided in some cases
  1. server-side data storage (HW Chatbot - Final)
  2. client-side UI (HW Squares - Final)

• In both cases, we try to constrain its use
  – including coding conventions to keep ourselves sane
Stateful UI in React
(React Components)
• **UI so far was static**
  – `index.tsx` calls `render` to show a fixed UI
    UI was different based on query params
    but never changed once rendered

• **Made the UI change by reloading the page**
  – change the query params, so it renders something different
UI in HW Fib - Weave

- Made the UI change by reloading the page
  - change the query params, so it renders something different

```
const word = params.get("word");
if (word === null) {
    root.render(<MakeForm/>);
} else {
    root.render(<ShowResults word={word} ../>);
}
```
Client-Side State

- Client needs to update the UI after getting response
  - don’t want to reload the whole page to redraw
    reloading is slow and can lose user data (e.g., contents of text fields)
  - need a way to update the UI without a reload

```
GET /new
{text: "Your fav color is?"}
```

```
GET /check?answer=blue
{correct: false}
```
React Functions

- React let us create custom tags
  - e.g., from HW Quilt
    
    ```
    root.render(<QuiltElem quilt={q}/>);
    ```
  
  - acts like the call
    
    ```
    root.render(QuiltElem({quilt: q}));
    ```
  
  - where `QuiltElem` is function taking a record argument
    
    ```
    const QuiltElem = (props: {quilt: Quilt}): JSX.Element => { .. }; 
    ```

- Render spots `<QuiltElem>` and calls `QuiltElem`
  
  - replaces `<QuiltElem>` with HTML returned by `QuiltElem`
React Functions

- React let us create custom tags
  - e.g., from HW Quilt

    ```js
    root.render(<QuiltElem quilt={q}/>);
    ```

  - acts like the call

    ```js
    root.render(QuiltElem({quilt: q}));
    ```

  - where `QuiltElem` is function taking a record argument

    ```js
    const QuiltElem = (props: {quilt: Quilt}): JSX.Element => { .. };
    ```

- Gives modularity but UI cannot change
  - need *mutable* state to allow the UI to update after events
React also let us create custom tags with classes

- e.g., from HW Quilt

```javascript
root.render(<QuiltElem quilt={q}/>);
```

- acts like the call

```javascript
root.render(new QuiltElem({quilt: q}).render());
```

- where `QuiltElem` is class that takes a record in constructor

```javascript
class QuiltElem extends Component<{quilt: Quilt}, {}> {
  constructor(props: {quilt: Quilt}) { ... /* store props */ }
  render = (): JSX.Element => { ... /* return HTML */ };
};
```
Simplest React Component

- Component that prints a Hello message:

  ```javascript
  type HiProps = {name: string};

  class HiElem extends Component<HiProps, {}> {
    constructor(props: HiProps) {
      super(props);
    }

    render = (): JSX.Element => {
      return <p>Hi, {this.props.name}</p>;
    }
  }

  - Used as `<HiElem name="Fred"/>`:
Simplest React Component

• Component that prints a Hello message:

```javascript
type HiProps = {name: string};

class HiElem extends Component<HiProps, {}> {
    constructor(props: HiProps) {
        super(props);
    }

    render = (): JSX.Element => {
        return <p>Hi, {this.props.name}</p>;
    }
}
```

• Component is a generic type
  – first type parameter is the type of “props”
  – second type parameter is for “state”...

No sensible reason to make Components without state
Simplest Stateful React Component

type HiProps = {name: string};
type HiState = {greeting: string};

class HiElem extends Component<HiProps, HiState> {
  constructor(props: HiProps) {
    super(props);
    this.state = {greeting: "Hi"};
  }
}

• Component is a generic type
  – first component is type of this.props (readonly)
  – second component is type of this.state

• Initialize this.state in the constructor
  – never directly modified after that
React Components

type HiProps = {name: string};
type HiState = {greeting: string};

class HiElem extends Component<HiProps, HiState> {
  render = (): JSX.Element {
    return <p>{this.state.greeting},
       {this.props.name}!</p>;
  }
}

• render can use both this.props and this.state
  – difference 1: caller give us props, but we set our state
  – difference 2: we can change our state
  – React will automatically re-render when state changes
    re-render happens shortly after the state change
React Components

type HiProps = {name: string};
type HiState = {greeting: string};

class HiElem extends Component<HiProps, HiState> {
  setGreeting = (newGreeting: string): void => {
    this.setState({greeting: newGreeting});
  };
}

- Must call setState to change the state
  - directly modifying this.state is a (painful) bug
    our linter will prevent this, thankfully

- React will automatically re-render when state changes
  - this is the (only) reason to use a Component
React Components

```javascript
type HiProps = { name: string };

type HiState = { greeting: string };

class HiElem extends Component<HiProps, HiState> {
    setGreeting = (newGreeting: string): void => {
        this.setState({ greeting: newGreeting });
    }
}
```

- **Must call** `setState` **to change the state**
  - directly modifying `this.state` is a *(painful)* bug
    our linter will prevent this, thankfully

- **Only need to supply the fields that have changed**
  - all the other fields will stay as they were before
type HiProps = {name: string};
type HiState = {greeting: string};

class HiElem extends Component<HiProps, HiState> {
  constructor(props: HiProps) {
    super(props);
    this.state = {greeting: "Hi"};
  }

  render = (): JSX.Element {
    return <p>{this.state.greeting},
               {this.props.name}!</p>;
  }

  setGreeting = (newGreeting: string): void => {
    this.setState({greeting: newGreeting});
  }
}

React Components

type HiProps = {name: string};
type HiState = {curName: string};

class HiElem extends Component<HiProps, HiState> {
    ...
    setGreeting = (newGreeting: string): void => {
        this.setState({greeting: newGreeting});
    };
}

• How could setGreeting be called?
  – typically happens in a handler for an HTML event

Hi, Fred. ➔ Hola, Fred.
React Component with an Event Handler

- **Pass method to be called as argument (a “callback”)**
  - **value of** onClick **attribute** is our **makeSpanish** method

    ```jsx
    render = (): JSX.Element {
        return (<div>
            <p>{this.state.greeting}, {this.props.name}!</p>
            <button onClick={this.doEspClick}>Espanol</button>
        </div>);
    }
    ```

- **Browser will invoke that method when button is clicked**

    ```jsx
    doEspClick = (evt: MouseEvent<HTMLButtonElement>) => {
        this.setState({greeting: "Hola"});
    }
    ```

    - **Call to** setState **causes a re-render (in a bit)**
React Component with an Event Handler

```javascript
type HiProps = {name: string};
type HiState = {greeting: string};

class HiElem extends Component<HiProps, HiState> {
    constructor(props: HiProps) {
        super(props);
        this.state = {greeting: "Hi"};
    }

    render = (): JSX.Element {
        return (<div>
            <p>{this.state.greeting}, {this.props.name}!</p>
            <button onClick={this.doEspClick}>Espanol</button>
        </div>);
    }

    doEspClick = (evt: MouseEvent<HTMLButtonElement>) => {
        this.setState({greeting: "Hola"});
    }
}
```
React Component with an Event Handler

- Pass method to be called as argument (a “callback”)
  - value of `onClick` attribute is our `makeSpanish` method

  ```jsx
  render = (): JSX.Element {
    return (<div>
      <p>{this.state.greeting}, {this.props.name}!</p>
      <button onClick={this.doEspClick}>&nbsp;Espanol</button>
    </div>);
  };
  ```

- Including parentheses here is a (painful) bug!
  - that would call the method inside render
    passing its return value as the value of the `onClick` attribute
  - we want to pass the method to the button, and have it called when the click occurs
Event Handler Conventions

• We will use this convention for event handlers

  doMyCompMyEvent

  component name  event name

  – e.g., doAddClick, doNewNameChange

• Reduces the need to explain these methods
  – method name is enough to understand what it is for
  – method name is the only thing you know they read

• Components should be just rendering & event handlers
React Components are Like Java Classes

```javascript

type HiProps = {name: string};
type HiState = {greeting: string};

• “Props” are part of the specification (arguments)
  – public interface, used by clients

    root.render(<Hi name={"Fred"}/>);   // pass in name

• “State” is like the concrete representation
  – private choice of data structures, hidden from clients

    constructor(props: HiProps) {
        super(props);
        this.state = {greeting: "Hi"};   // initial state
    }
```
React Components are Like Java Classes

HTML on screen = render(this.state)

<table>
<thead>
<tr>
<th>Component</th>
<th>React</th>
</tr>
</thead>
<tbody>
<tr>
<td>t = 10</td>
<td>this.state = s₁</td>
</tr>
<tr>
<td>t = 20</td>
<td>this.setState(s₂)</td>
</tr>
<tr>
<td>t = 30</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

React updates this.state to s₂ and doc to HTML₂ simultaneously
React Components are Like Java Classes

HTML on screen = render(this.state)

• Don’t want to be in a state where that is not true
  ...unless you enjoy painful debugging

1. Do not mutate this.state (call setState)
   React will update this.state and HTML on screen at the same time

   Easy way to ensure this: disallow mutation in the client

   We’ll use that rule this quarter.
React Components are Like Java Classes

HTML on screen = render(this.state)

• Don’t want to be in a state where that is not true
  ...unless you enjoy painful debugging

1. Do not mutate this.state (call setState)
   React will update this.state and HTML on screen at the same time

2. Make sure no data on screen would disappear on re-render
   More on this later...
React Components have Mutable Heap State

• Like Java Classes, methods are sharing state
  – change in one method is read in other methods

• Error in one method (writing) fails in another (reading)
  – debugging will be harder!

• HW Squares - Final are the debugging assignments
  – necessary to understand all the parts of the code
React Components have Mutable Heap State

• Hard debugging makes correctness more important

• Move complex parts into separate functions
  – test and reason carefully through those functions
  – class is ideally just be rendering and event handlers
    move everything complex into helper functions
    e.g., calculation of new state can be a helper function
  – harder to reason about and test with mutable heap state,
    so keep it simple

• Write code to check your invariants
  – ensure the new state is valid before calling setState
  – practice defensive programming
Example: To-Do List (v1)
TodoApp – State

// Represents one item in the todo list.

```typescript
// Represents one item in the todo list.

type Item = {
    name: string;
    completed: boolean;
};
```

// Client gives us the initial (complete) list of items.

```typescript
// Client gives us the initial (complete) list of items.

type TodoProps = {
    initialItems: Item[];  // items to show initially
};
```

// State of the app is the current list of items,
// which will be the initial list with some possibly removed.

```typescript
// State of the app is the current list of items,
// which will be the initial list with some possibly removed.

type TodoState = {
    items: Item[];  // current list of items
};
```
TodoApp – Class

// Application that displays a to-do list.
export class TodoApp extends Component<TodoProps, TodoState> {

    constructor(props: TodoProps) {
        super(props);

        this.state = {items: props.initialItems.slice(0)};
    }

    ...

TodoApp – Render

// Return a UI with all the items and elements that allow them to
// add a new item with a name of their choice.
render = (): JSX.Element => {
  return (  
    <div>
      <h2>To-Do List</h2>
      {this.renderItems()}
    </div>);  
}
TodoApp – Render Items (abbreviated)

renderItems = () => { JSX.Element[] => {
    const items: JSX.Element[] = [];
    for (let i = 0; i < this.state.items.length; i++) {
        if (!this.state.items[i].completed) {
            items.push(
                <div className="form-check" key={i}>
                    <input className="form-check-input" type="checkbox"
                        id="check" + i checked={false}
                        onChange={evt => this.doItemChange(evt, i)} />
                    <label className="form-check-label" htmlFor="check" + i>
                        {this.state.items[i].name}
                    </label>
                </div>);
        } else { /* read-only once completed */ }
    }
    return items;
};
// Called when the user clicks on an item.
// Removes it from the list and redraws

doItemChange = (evt: ChangeEvent<HTMLInputElement>,
               index: number): JSX.Element => {
    const newItems = this.state.items.slice(0, index).concat(
        this.state.items.slice(index + 1));
    this.setState({items: newItems});
}
Example: To-Do List (v2)
// Represents one item in the todo list.
type Item = {
    name: string;
    completed: boolean;
};

// State of the app is the list of items and the text that the
// the user is typing into the new item field.
type TodoState = {
    items: Item[];  // existing items
    newName: string;  // mirrors text in the field to add a new name
        // (need this for two reasons...)
};
TodoApp – Class

// Application that displays a to-do list.
export class TodoApp extends Component<{}, TodoState> {

    constructor(props: {})
    {
        super(props);

        this.state = {items: [], newName: ""};
    }

    ...

// Return a UI with all the items and elements that allow them to
// add a new item with a name of their choice.
render = (): JSX.Element => {

  return (
    <div>
      <h2>To-Do List</h2>
      {this.renderItems()}
      <p className="instructions">Check an item to mark it…</p>
      <p className="more-instructions">New item:
        <input type="text" className="new-item"
          value={this.state.newName}
          onChange={this.doNewNameChange} />
        <button type="button" className="btn btn-link"
          onClick={this.doAddClick}>Add</button>
    </p>
    </div>);
}
// Return a UI with all the items and elements that allow them to
// add a new item with a name of their choice.
render = (): JSX.Element => {
    return (
        <div>
            <h2>To-Do List</h2>
            {this.renderItems()}
            <p className="instructions">Check an item to mark it... </p>
            <p className="more-instructions">New item:
                <input type="text" className="new-item"
                    value={this.state.newName}
                    onChange={this.doNewNameChange} />
                <button type="button" className="btn btn-link"
                    onClick={this.doAddClick}>Add</button>
            </p>
        </div>);
}
TodoApp – Add Click

// Called when the user clicks on the button to add the new item.
doAddClick = (_: MouseEvent<HTMLButtonElement>): void => {
  // Ignore the request if the user hasn't entered a name.
  const name = this.state.newName.trim();
  if (name.length == 0) return;

  // Cannot mutate this.state.items! Must make a new array.
  const items = this.state.items.concat([ {name: name, completed: false} ]);  
  this.setState({items: items, newName: ""}); // clear input box
};
// Return a UI with all the items and elements that allow them to
// add a new item with a name of their choice.
render = (): JSX.Element => {
  return (  
    <div>
      <h2>To-Do List</h2>
      {this.renderItems()}
      <p className="instructions">Check an item to mark it…</p>
      <p className="more-instructions">New item:
        <input type="text" className="new-item"
          value={this.state.newName}
          onChange={this.doNewNameChange} />
        <button type="button" className="btn btn-link"
          onClick={this.doAddClick}>Add</button>
      </p>
    </div>);
}
TodoApp – New Name Change

// Called each time the text in the new item name field is changed.
doNewNameChange = (evt: ChangeEvent<HTMLInputElement>): void => {
  this.setState({newName: evt.target.value});
}

• Most event handlers are passed an event object
  – field “evt.target” stores the object that fired the event
  – hence, “evt.target.value” is the text in that input box

• Make sure no data on screen would disappear on re-render
  – must record the text the user typed into the field
goes into the value={} attribute of the input box
  – otherwise, render would produce an input box with no text
Other Events

• Components should be just rendering & event handlers
  – our linter will enforce this

• Timers have events that fire after a given time
  – call to `setTimeout` invokes callback after a delay
Example: Auctions
More Complex UI

- To-Do List UI is basic
  - all of it easily fits in a single component (`TodoApp.tsx`)

  **To-Do List**

  - laundry
  - wash dog

  Check the item to mark it completed.

  New item: [Enter new item] [Add]

- More complex UI can be too much code for one file
  - necessary to split it into multiple components
Recall: Other Properties of High-Quality Code

• Professionals are expected to write high-quality code

• Correctness is the most important part of quality
  – users hate products that do not work properly

• Also includes the following:
  – easy to understand
  – easy to change via abstraction
  – modular
Component Modularity

• Poor design to put all the app in one Component
  – it works, but is lacks properties of high-quality code
  – better to break it into smaller pieces (modular)

• Two ways to the UI into separate components:

  1. Separate parts that are next to each other on screen
  2. Separate parts on the screen at different times
Component Modularity

- Separate parts that are next to each other

```javascript
class App extends Component<..> {
    render = (): JSX.Element {
        return (<div>
            <TitleBar title={"My App"}/>
            <SideBar/>
            <MainBody/>
        </div>);
    }
}
```
Component Modularity

- Separate parts on the screen at different times
- App is always on the screen
  - App chooses which child component to display
    - sometimes it has an Editor child and sometimes not
Component Modularity

• Separate parts on the screen at different times

```typescript
type AppState = {editItem: string | undefined};

class App extends Component<{}, AppState> {
  ...
  render = () : JSX.Element {
    if (this.state.editItem !== undefined) {
      return <ItemEditor item={this.state.editItem}/>
    } else {
      return <ItemList/>
    }
  }
  ...
}
```
Example: Auctions
Example: Auction UI

- Auction site has three different “pages”

**Current Auctions**

- **Oak Cabinet** ends in 10 min
- **Red Couch** ends in 15 min
- **Blue Bicycle**

**Oak Cabinet**

A beautiful solid oak cabinet. Perfect for any bedroom. Dimensions are 42” x 60”.

Current Bid: $250

<table>
<thead>
<tr>
<th>Name</th>
<th>Bid</th>
<th>Submit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fred</td>
<td>251</td>
<td></td>
</tr>
</tbody>
</table>

**New Auction**

<table>
<thead>
<tr>
<th>Name</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>Table Lamp</td>
</tr>
</tbody>
</table>
Example: Auction UI

- Auction site has three different “pages”

- Need four different components:
  - Auction List: shows all the auctions (and Add button)
  - Auction Details: shows details on the auction (w Bid button)
  - New Auction: lets the user describe a new auction
  - **App**: decides which of these pages to show
Auction Client: App.tsx

- state needs to indicate which page to be showing

```typescript
// RI: 0 <= index < auctions.length
type Page = {kind: “list”} | {kind: “new”} |
    {kind: “details”, index: number};

type AppState = {page: Page, auctions: Auction[]};

class App extends Component<{}, AppState> { … }
```

- What is Page an example of?
  it is an inductive data type (of the “enum” variety)

```typescript
type Page := list | new | details(n: N)
```
render = (): JSX.Element => {
  if (this.state.page.kind === "list") {
    return <AuctionList auctions={this.state.auctions}
      onNewClick={this.doNewClick}
      onAuctionClick={this.doAuctionClick}/>
  }
  else if (this.state.page.kind === "new") {
    return <NewAuction onStartClick={this.doStartClick}
      onBackClick={this.doBackClick}/>
  }
  else { // kind: "details"
    const auction = this.state.auctions[this.state.page.index];
    return <AuctionDetails auction={auction}
      onBidClick={this.doBidClick}
      onBackClick={this.doBackClick}/>
  }
};
Example: Auction UI

**Current Auctions**

- **Oak Cabinet** ends in 10 min
- **Red Couch** ends in 15 min
- **Blue Bicycle**

**Oak Cabinet**

A beautiful solid oak cabinet. Perfect for any bedroom. Dimensions are 42” x 60”.

Current Bid: $250

<table>
<thead>
<tr>
<th>Name</th>
<th>Bid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fred</td>
<td>251</td>
</tr>
</tbody>
</table>

**New Auction**

- **Name**: Bob
- **Item**: Table Lamp

Buttons:
- **Submit**
- **Back**
- **Start**
- **Back**

Event handlers:
- `on_auc_onAuctionClick`
- `on_auc_onNewClick`
- `on_auc_onBidClick`
- `on_auc_onStartClick`
- `on_auc_onBackClick`
– event handlers change what is shown

```javascript
doNewClick = (): void => {
    this.setState({page: "new"}); // show new auction page
};

doBackClick = (): void => {
    this.setState({page: "list"}); // show auction list page
};

doAuctionClick = (index: number): void => {
    // show details list page for the given auction
    this.setState({page: {kind: "details", index: index}});
};
```
Auction Client: `App.tsx`

- the `App` component stores the auction list easy to pass it down to subcomponents in their props
- subcomponents cannot mutate the auction list!
  they must invoke `callbacks` to have the `App` update the auction list

```typescript
doStartClick = (name: string, seller: string, ...): void => {
  const auction = {name, seller, ...};
  const auctions = this.state.auctions.concat([auction]);
  this.setState({page: "list", auctions: auctions});
};

doBidClick = (index: number, bidder: string, amount: number) => {
  const newVal = ...; // update the auction to have a new high bidder
  const auctions = this.state.auctions.slice(0, index)
                   .concat([newVal])
                   .concat(this.state.auctions.slice(index+1));
  this.setState({auctions: auctions,
                 page: {kind: "details", index: index}});
};
```
Next Up: “Full Stack” (Client & Server)

- **Stateful client:** error in one method fails in another
  - bug in writing new state shows up when reading it

- **Client-server:** error in one part can fail in the other
  - bug in client shows up as server crash
  - bug in server shows up as client crash

- **HW Squares - Final** are the **debugging** assignments
  - necessary to **understand** all the parts of the code