Technische Universität München
Fakultät für Informatik

# Exercise Sheet 4: Anti-Patterns, Software Reuse, and React

Welcome to the fourth week of Advanced Topics of Software Engineering! This week we will deepen the discussion on *good* software design by revisiting anti-patterns on the code level and different strategies for software reuse. In particular, for the latter, we will cover software product lines, and distinguish between libraries and frameworks in the theoretical exercises.

In the practical exercises, we will get to know an open-source client-side JavaScript library, React, which helps you to build interactive user interfaces with its component-based architecture.

## Theoretical Exercise

### Exercise 1: Anti-Patterns

In the lecture you have learned about high level anti-patterns such as *Blob* or *Design by Committee*. This exercise now aims to give you an introduction to language level anti-patterns.

Take a look at the following Java source code examples. These examples are syntactically correct, but exhibit slight problems. **Relate** these problems to the seven deadly sins from the lecture **and provide** an improved solution (multiple solutions might be reasonable).

1. A junior developer is tasked with parsing a guestbook and outputting all greetings. At the end of the business day he proudly presents you his solution:

```java
int start = xml.indexOf("<content>") + "<content>".length();
int end = xml.indexOf("</content>");
String content = xml.substring(start, end);
```

2. Browsing your company's code base you come across these lines:

```java
File tmp = new File("C:\\Temp\\1.tmp");
File exp = new File("export-2013-02-01T12:30.txt");
File f = new File(path +'/'+ filename);
```

3. An inexperienced programmer gives you a prototypical implementation that you have to improve. It contains this try-catch block:

```java
Query q = ...
Person p;
try {
  p = (Person) q.getSingleResult();
} catch(Exception e) {
  p = null;
}
```

4. A network application that you maintain sometimes crashes silently. You already spend hours trying to find the cause. Suddenly you spy these lines of code:

```java
try {
  ... do risky stuff ...
} catch(SomeException e) {
  // never happens
}
... do some more ...
```

5. Maintaining an e-commerce website you receive an angry call from a customer. He just sold 100 different chewing gums for €0.30 each and the bill to the buyer totaled to only €29.999971 instead of €30. After inspecting the code you start to write a letter of apology to your customer.

```
float total = 0.0f;
for (OrderLine line : lines) {
  total += line.price * line.count;
}
```

## Exercise 2: Software Reuse

In the lecture on software reuse you have been introduced to the different levels of (structured) reuse and how different concepts try to cope with the challenges of managing variability. This exercise builds on top of this discussion by providing an example for software product lines and by distinguishing libraries from frameworks as reusable software artifacts.

Reference architectures, as another approach for reuse, are discussed in the practical and project exercises below.

1. **Software Product Lines**

   *Mockia* is a rising star in the highly competitive smartphone market. Their smartphone *MeiFon* is the first smartphone that allows almost full customization of the hardware configuration. Although the new concept already found wide reception and pre-orders by far exceed the immediate production capabilities, the CTO of Mockia raised concerns about the flexibility of their current development approach. Currently the entire software stack of Mockia's smartphones is custom built; reuse is, if at all, entirely done opportunistically (i.e. *ad-hoc reuse*). There is an enormous mass of configuration possibilities: a user can choose between 4 CPU models, 6 persistent memory and 3 different camera modules, and 2 GPS chips. A closer look at the pre-order data revealed that more than 92% of the customers pre-ordered a configuration that included the cheapest CPU and the second-smallest memory module. You are hired to propose a new development concept to make the software development more efficient. From previous projects you know that developing a new controller software for a CPU takes about 6 person months (PM), for a memory module 2PM, for a camera module 4PM, and for a GPS chip 2PM. Furthermore, your experience taught you that writing a SOLID platform that a-priori is designed for reuse typically yields 3 times the effort of the normal development effort of the respective components.

   Discuss

   (a) what would be the *core features* (i.e. commonalities) of a (economically reasonable) software product line,

   (b) what would be the *variability points*, and

   (c) when would you expect the *break-even point*.

   Note: Even though MeiFon is known for its full customization, it could be the case that from an economic perspective, it does not make sense to give full flexibility regarding the customization.

2. **Libraries vs. Frameworks**

   Copying contiguous fragments of source code with common functionality or design patterns from existing systems to new systems is an ad-hoc approach for software reuse which has been called *design* or *code scavenging*[1]. The approaches have evolved since these first rather unstructured attempts and reusable artifacts, e.g. libraries and frameworks, are inherently important to modern software development (especially in business information systems).

   Try to differentiate between the notions of a library and of a framework and discuss potential problems with using frameworks (some software engineers even argue that *frameworks are evil*).

[1] Krueger (1992). Software reuse.

## Practical Exercise

In this week's practical exercise, you will learn how to create a new React app, build components, design views, and utilize a UI framework. We chose React for this task due to its:

- developer- friendly architecture that gives you the ability to create *reusable* UI components (buttons, tables, grids,...)
- VirtualDOM rendering mechanism that efficiently updates and renders just the right components when data changes (no need for page reloads!)
- scalability of large-scale web apps
- flexibilty: React can be used on various number of platforms such as *Electron*[2] for buildiing desktop applications, *Next.js*[3] for server-side rendered (SSR)[4] components, or *ReactVR*[5] for building virtual reality (VR) applications that run in browser

"Learn React Once and Write Everywhere" - *Reactjs.org*

### Exercise 3: Introduction to React

1. In this exercise, you are going to use the de-facto way to start a new single-page application in React, the *create-react-app*[6] package. To use this package, you have to first make sure that you have Node.js and node package manager (npm)[7] installed in your system. To check this, you can use the following commands which would output the version of your Node.js and npm builds:

```
      $ node -v
      v16.7.0 # Node.js version
      $ npm -v
      7.20.3 # Node package manager (npm) version
```

   If you get a `command not found` output when you run these version checks, then visit https://nodejs.org/en/download/ for installing the latest verion of Node.js (comes with npm).

2. Run `npx create-react-app <your_app_name>` to create a React app.
   *(npx is just a package runner tool that comes with npm 5.2+)*

3. Once the setup is over, open the app directory in your favorite editor (VS Code, Sublime Text, Atom, ...). You should be able to locate your *src*, *public*, and *node_modules* folders and your *package.json* file.

4. Open a terminal inside your application's directory and run `npm start` to start your application. On the page that opens in your browser, you should be able to see a rotating React logo with some text under it.

5. Before jumping into creating a component, let's take a closer look at how the application works. Under your *public* folder, you have the *index.html* file. This is a standard HTML file that serves as an entry point for the application. Inside this file, you can see that there is a single *<div>* element with an attribute `id="root"`. Your React app injects its code inside this div.
   **Tip**: *For better search engine optimization (SEO), locate the* <title> *tag (controls the title of your HTML document) and change it to something that fits to your application.*

6. Now, open the *index.js* file which is located under *src* folder. This is where your React app begins. As you can see, a component called "App" is injected into the root div element that is defined in the index.html file. Thus, our top-level component is App. This is standard for almost all React applications.

[2] https://www.electronjs.org/
[3] https://nextjs.org/
[4] https://tsh.io/blog/what-is-next-js-used-for/
[5] https://www.pluralsight.com/guides/getting-started-with-react-vr
[6] https://reactjs.org/docs/create-a-new-react-app.html
[7] https://www.npmjs.com/

**What are React Components?**
Components are building blocks of React applications that are responsible for generating UI views. Each component returns/renders some JSX[8] code and defines which HTML code React should render to the real DOM in the end.

**Component Types**
The two main components types in React are **functional** and **class** components. While the former is basically a JavaScript function that accepts props as an argument and returns a React element, the latter requires you to extend React and create a render function that returns a React element.

Class components are also known as *stateful* components as they implement logic and state. The functional components on the other hand are known as *stateless* components since by default they only accept some arguments and display them in some form. However, with React 16.8, **hooks** are introduced to React. Hooks gives you the ability to use state logic and other React features without writing classes. Thus, we are going to be using hooks for managing states when writing functional components.

**React Hooks**
Hooks are JavaScript functions that let you "hook" into states from functional components. For this exercise, we are only going to be using the built-in hooks *useState* and *useEffect*. However, you can also come up with your own hooks.[9]

The useState hook lets you add a local state to a functional component and React preserves this state between re-renders. A call to the useState hook returns:

- the current state value,
- function that lets you update the state value.

You can then call this function from an event handler or somewhere else to update the state. However, as a rule-of-thumb, *never* call hooks inside loops, conditions, or nested functions.

The useEffect hook is utilized when you want to do operations like data fetching, subscriptions, or manually changing the DOM. These operations are called *side effects* since they affect other components and can't be executed during rendering. In class components, the role of the effect hook is carried out by different functions such as componentDidMount, componentDidUpdate, and componentWillUnmount.

## Exercise 4: Creating a React Component

Now that you have learned what are React components and why you need them, let's move on to creating your own component. In your current application, you only have a single component, App, which is by default your top-level component.

1. What is the type of App as a component? Is it allowed to manage states with its default implementation?
2. You can see that the current display on your browser is actually returned by your App component. Go ahead and update the returned element such that it only shows a welcome text instead of the React logo and other texts.
   **Warning**: *Inside the return, there can only be a single top level element.*
   **Tip**: *Remove all the "className" attributes if you want to clear the default styling.*
3. Create a new file called *Quote.js* under your src folder. Copy and paste the content of App.js into this new file. Now, update all the necessary fields (function name, exported element name, . . . ) and remove any unnecessary imports. Finally, update the return such that it only returns a paragraph element with your favorite quote in it. Congratulations, you have written your first functional component! However, we are not done yet. . .
4. You can see that although you have created a new component that returns your favorite quote, it is not shown on the browser yet. This is because it is not rendered by any other component. Open App.js and update it with the following code piece:

[8] https://reactjs.org/docs/introducing-jsx.html
[9] https://reactjs.org/docs/hooks-custom.html

```
 1  import Quote from './Quote';
 2
 3  function App() {
 4    return (
 5      <div>
 6        <Quote/>
 7      </div>
 8    );
 9  }
10
11  export default App;
```

5. Once you save the updated App.js file, you will see that your favorite quote is now displayed on the browser. This is because we have returned your Quote component at line 6.
   **Warning**: *Make sure that the components that you return are imported in the first place (e.g. Line 1 in App.js).*

## Exercise 5: Building a Simple UI with React and Material UI

Although you have got your favorite quote displayed on your screen with your new React skills, it is not fun to look at the same thing which doesn't even have a proper display in the first place. In this exercise, you are going to design an interface that displays a random quote out of the quotes you have added. To make everything look prettier, you will use *Material UI* (MUI)[10] which is one of the most popular React UI frameworks. To get a better understanding of MUI components visit https://mui.com/components.

1. Open a terminal inside your application's directory and run the following command to add MUI packages.
   ```
   npm install @mui/material @emotion/react @emotion/styled
   ```
2. To test if the installation worked, open your Quote.js file and copy the following code. Upon saving, you should be able to see a button under your quote.

```
import Button from "@mui/material/Button";

function Quote() {
  return (
    <div>
      <p>My favorite quote.</p>
      <Button variant="contained">My MUI Button</Button>
    </div>
  );
}

export default Quote;
```

3. Now that MUI is working, you can move on to designing the interface. The interface should:
   - Display a quote
   - Have a button to add a new quote (no functionality needed)
   - Have a button to change the displayed quote (no functionality needed)

   An example design is shown in Figure 1.
   **Hint**: *You can use Container, Paper, Typography, and TextField components of MUI for a cleaner design.*

[10]https://mui.com/

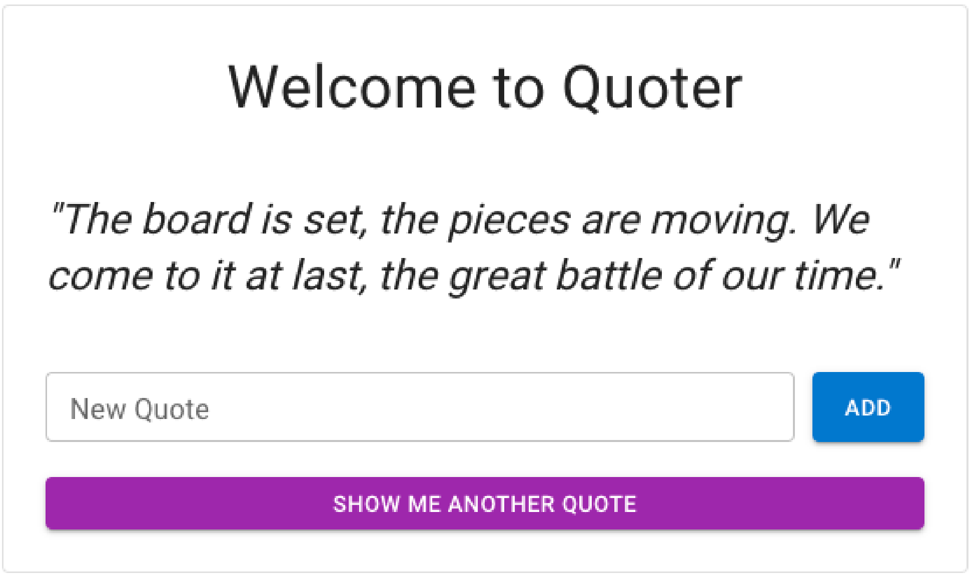
Welcome to Quoter
"The board is set, the pieces are moving. We come to it at last, the great battle of our time."
New Quote
ADD
SHOW ME ANOTHER QUOTE

Figure 1: An example interface for the Quoter application.

4. The next step is listening to the changes in the *New Quote* field. To do so, go ahead and add "onChange" event to your text field and pass a function to it in order to handle the event.
   An example implementation:

```
import TextField from "@mui/material/TextField";

function Quote() {
  const handleChange = (e) => {
    // e is a synthetic event
    // visit https://reactjs.org/docs/events.html to learn more
    console.log(e.target.value)
  }

  return (
    <div>
      <TextField fullWidth id="quote" size="small" label="New Quote" name="quote"
          onChange={handleChange}/>
    </div>
  );
}

export default Quote;
```

   After implementing the onChange event, you should see that every time you type a new character, the handler function is getting called.

5. To store the value of the *New Quote* field, we need a state variable. As explained previously, to use states in functional components, we need hooks. For this purpose, go ahead and import the *useState* hook and define a state variable called "newQuote" (should be set to an empty string initially). You can use the following lines for this task:
   `import React, { useState } from 'react';`
   `const [newQuote, setNewQuote] = useState("");`

6. Now, call *setNewQuote* function inside your on change event handler with the new value of the text field. Also, add a new attribute to your text field component called "value" and set it to newQuote. This way, your text field is binded to the state variable.

7. Finally, let's add functionality to our buttons. When *Add* button is clicked, *newQuote* should be added to a list of existing quotes (and set back to an empty string) and when *Show Me Another Quote* button is clicked, a random quote from the quotes list should be picked and displayed. To handle the clicking event, you can use the "onClick" event in your buttons.
   **Hint**: *You need two more state variables; one for storing the displayed quote and the other for storing the list of existing quotes. Ideally, the displayed quote variable should be set to some quote initially. This way, when the app first begins, the display won't appear empty. Don't forget to include this initial quote in your quotes list!*