Software Architecture and Design Patterns Project Documentation for ARGeocache

Authors: Sam MacPhee, Jared Soehner, Haniya Ahmed, and Alyssa Wilcox

Table of Contents

Introduction

ARGeocache is an app created in the 2021-2022 academic year for one of our member's senior design project in software engineering. It is an Android app built in react native that is a game meant to allow players to search for and place geocaches digitally in their local area. The app displayed a need for improvement, therefore it was a perfect candidate for this project. The design patterns we decided to implement were the Factory pattern, the Prototype Pattern, the Bridge pattern, the Composite pattern, and the Iterator pattern. Below you will find the justification for each design pattern, the source code for the change, and the UML diagram for the system changes.

Creational Design Patterns

Factory Pattern

We implemented the Factory Pattern to instantiate different types of Buttons and assign standardized behaviours.

Previous Implementation

- The code for the creation of various UI elements in the App was done by a "new" instantiation with the styles and behaviour specified inline at the time of rendering the page (See Figure 1.1). This violates the Open-Close SOLID principle because it needs to be recompiled every time there is a deployment change.

- Button action methods defined in the views are long and complex, and violate the single responsibility principle as they may be responsible for multiple things at once (performing a request, displaying error/success messages, navigating to another screen) (See onPressSignUp in Figure 1.1). Other buttons in the app perform similar steps in their actions, and it would be nice to not have to re-write the same behaviour in different action methods for every button, possibly introducing inconsistencies in the ways the Buttons behave and confusing the user. Therefore, we want to introduce some different types of Button classes which will standardize the different types of behaviours our Buttons can have, and allow us to pass in parameters to customize the behaviour such as backend calls, success and error messages, and navigation locations. However we do not want to instantiate the concrete Button classes directly, as this would increase coupling in our code, and our Views do not really need to know the concrete button class, just the Button interface. We also want to be able to add new Button types in the future, and have them seamlessly integrate into our existing code without breaking our existing Views.

```
const onPressSignUp = async () => {
```

```
  try {
    if (password === confirmPassword) {
      if (password.length > 7) {
        await signUp(username, password);
        signIn(username, password);
        await user.functions.insertUser(username, password);
        navigation.navigate('MainMenu');
        alert('New Profile Created, you are now signed in');
      } else {
        Alert.alert('Password must be at least 8 characters long');
      }
    } else {
      Alert.alert('Passwords do not match, please try again');
    }
  } catch (error) {
    Alert.alert(`Failed to sign up: ${error.message}`);
  }
};
                                                    :
                                                    :
                                                    :
<TouchableOpacity onPress={navigation.goBack}>
  <Image
    source={require('../../data/images/back.png')}
    style={{width: 35, height: 35, marginLeft: 2}}
  />
</TouchableOpacity>
                                                    :
                                                    :
                                                    :
<TouchableOpacity style={styles.SubmitButton} onPress={onPressSignUp}>
  <Text style={{fontWeight: 'bold', color: 'black'}}>Submit</Text>
</TouchableOpacity>
```

Code Excerpt 1.1 (Register.js). *Examples of Buttons created in the Register.js file before refactoring. The onPressSignUp action needs to handle performing the sign up action, displaying error messages, and navigating to the Main Menu. Our app has several buttons which need to perform similar sequences of steps, so it would be nice to abstract away some of the common steps of creating these actions, so that we do not accidentally introduce inconsistencies in the ways our buttons behave.*

Implementing the Factory method for button creation is one way we can solve these problems. Instead of creating a new button from scratch everytime we need a button, we will use the Factory method to defer button instantiation to a helper class.

Justification for the Pattern

How is the Factory Method useful for solving the above problems? Buttons in the ARGeocache app have three main behaviours, which are sometimes combined in a single action. They could display messages in the form of alerts, trigger navigation to different screens in the app, and/or perform a backend call (such as user log in). Our buttons naturally fall into the following classes:

1. A parent **Button** class which takes in an action and styling as parameters and builds a button which will perform that action upon being clicked.
2. A **MessageButton** class. This button simply displays a message upon being clicked. This could be useful for buttons who's actions have not been implemented yet. We only need to pass in the message, and the **MessageButton** constructor can create an action to display that message and pass it to the superclass.
3. An **ActionButton** class. This button performs some method, and then may display an alert regarding the result of the method. We pass in a method which returns null on success, and an error message if it fails. The **ActionButton** constructor builds an action which runs the method and displays the error message if it failed, and displays the success message if the method succeeded.
4. A **NavigationButton** class. This button triggers navigation through the app. The navigation may be contingent on a certain method succeeding, (for example, only navigate to the home screen if the "Log In" method was successful). We pass in the method to be called and the location to navigate to on success, and the **NavigationButton** constructor builds the action.
5. A **BackButton** class. This button simply returns to the previous screen. It does not perform any other action or display any messages, so its only parameter is the React-Native's navigation object, on which it will call the goBack method in its action.

We can see that all button classes use a subset of these main building blocks:

1. A style and graphic.
2. A method that could succeed or fail, returning an error message if it fails.
3. A location to navigate to on success.
4. A message to display on success.

We can pass these parameters into a Factory Method which can decide which type of button to instantiate depending on which values were provided, therefore the Factory Method pattern is a natural fit for creating the buttons while avoiding coupling our views to several different button classes. In Code Excerpt 1.2, we can see how the onPressSignUp action was simplified, and it is easier to read in the code whether or not a button performs navigation or displays a message on success, without having to trace through the large "onPressSignUp" method. This will allow us to easily add new Button types which have different behaviours without changing our existing View code, making it easily extendable.

```
const buttonFactory = new ButtonFactory();
                                                    ⋮
                                                    ⋮
                                                    ⋮
const onPressSignUp = async () => {
```

```
  if (password !== confirmPassword)
    return 'Passwords do not match, please try again';
  if (password.length < 8)
    return 'Password must be at least 8 characters long';

  try {
    await signUp(username, password);
    if (!(await signIn(username, password)))
      return 'User was not able to be signed in';
  } catch (error) {
    return `Failed to sign up: ${error.message}`;
  }
};
                                                    ⋮
                                                    ⋮
                                                    ⋮
{buttonFactory.createButton({navigation, navTo: -1}).component}
                                                    ⋮
                                                    ⋮
                                                    ⋮
{
  buttonFactory.createButton({
    action: onPressSignUp,
    message: 'New Profile Created, you are now signed in',
    navigation,
    navTo: 'MainMenu',
    buttonStyle: new CenterButtonStyle('Submit'),
   }).component
}
```

Code Excerpt 1.2 (Register.js). *Examples of Buttons created in the Register.js file after refactoring using the Factory Method pattern. The onPressSignUp action is no longer concerned with performing navigation or displaying alerts. It simply needs to sign up and log in the user, and if any problems occur, return an error message. The concrete Button class that ends up being used to instantiate the Button will create the full action in it's constructor, which will perform the onPressSignUp method and then display alerts and navigate depending on whether onPressSignUp returned an error message or not.*

Drawbacks of the Pattern

While this pattern makes the button creation code extremely convenient, it does in some ways decrease the maintainability of the code. For example, if when testing the app, I come across a Button which is not behaving how I expect, investigating the problem becomes more complicated since it is not clear on a first glance at the View file which Button class is actually being instantiated, so it will take more effort to track down the bug. Also, in order to implement

the pattern, many classes were needed, when it would have been possible to simply implement four button classes. However, we believe the usage of the pattern for this use case outweighs the drawbacks, since it is now a lot easier to write button actions, and we should find less bugs in the button actions since the common steps have been abstracted into the Button classes themselves to be reused as many times as necessary.

Pattern Implementation

As shown in the UML in Appendix A, we have created

1. An abstract product Button
2. Multiple concrete products such as NavigationButton, ActionButton, MessageButton, and BackButton.
3. An abstract factory method ButtonFactory
4. Concrete implementations of the ButtonFactory for each of the respective concrete products.
5. The factory method itself is the createButton method, defined in the ButtonFactory and overridden by each of the concrete factories.

The app view classes each instantiate a ButtonFactory and call the createButton method instead of defining buttons with JSX inline (as shown in Code Excerpt 1.2). This implementation allows the functionality of the app to be far more extensible by being able to easily create a new type of button for future features. This solves the problem of the application violating the open-close principle.

Source Code

```tsx
import React from 'react';
import {TouchableOpacity} from 'react-native';
import ButtonStyle from './button-styles/ButtonStyle';

export default class Button {
  buttonStyle: ButtonStyle
  action: Function

  constructor(action: Function, buttonStyle: ButtonStyle) {
    this.buttonStyle = buttonStyle;
    this.action = action;
  }

  get component(): JSX.Element {
    return (
      <TouchableOpacity style={this.buttonStyle.style} onPress={this.action}>
        {this.buttonStyle.graphic}
      </TouchableOpacity>
    );
  }
}
```

Figure 1.3. Button.tsx

```
import ActionButtonFactory from './ActionButtonFactory';
import BackButtonFactory from './BackButtonFactory';
import Button from './Button';
import ButtonStyle from './button-styles/ButtonStyle';
import MessageButtonFactory from './MessageButtonFactory';
import NavigationButtonFactory from './NavigationButtonFactory';

export default class ButtonFactory {
  createButton({
    action,
    navigation,
    navTo,
    message,
    buttonStyle,
  }: {
    action: Function;
    navigation: any;
    navTo: String | number | null;
    message: String | null;
    buttonStyle: ButtonStyle;
  }): Button {
    let factory: ButtonFactory;
    if (navigation && navTo === -1) {
      factory = new BackButtonFactory();
    } else if (navigation) {
      factory = new NavigationButtonFactory();
    } else if (action) {
      factory = new ActionButtonFactory();
    } else {
      // message must not be null
      factory = new MessageButtonFactory();
    }

    return factory.createButton({
      action,
      navigation,
      navTo,
      message,
      buttonStyle,
    });
  }
}
```

Figure 1.4. ButtonFactory.ts

```
import Button from './Button';
import IconButtonStyle from './button-styles/IconButtonStyle';

export default class BackButton extends Button {
  constructor(navigation: any) {
    super(
      navigation.goBack,
      new IconButtonStyle(require('./../../../data/images/back.png'), null, 35),
    );
  }
}
```

Figure 1.5. BackButton.ts

```
import BackButton from './BackButton';
import Button from './Button';
import ButtonStyle from './button-styles/ButtonStyle';

export default class BackButtonFactory {
  createButton({
    action,
    navigation,
    navTo,
    message,
    buttonStyle,
  }: {
    action: Function;
    navigation: any;
    navTo: String;
    message: String;
    buttonStyle: ButtonStyle;
  }): Button {
    return new BackButton(navigation);
  }
}
```

Figure 1.6. BackButtonFactory.ts

```
import {Alert} from 'react-native';
import Button from './Button';
import ButtonStyle from './button-styles/ButtonStyle';

export default class ActionButton extends Button {
  constructor(
    action: Function,
    successMessage: String,
    buttonStyle: ButtonStyle,
  ) {
    super(async () => {
      const errorMessage = await action();
      if (errorMessage) {
        Alert.alert(null, errorMessage);
      } else {
        Alert.alert(successMessage);
      }
    }, buttonStyle);
  }
}
```

Figure 1.7. ActionButton.ts

```
import ActionButton from './ActionButton';
import Button from './Button';
import ButtonStyle from './button-styles/ButtonStyle';

export default class ActionButtonFactory {
  createButton({
    action,
    navigation,
    navTo,
    message,
    buttonStyle,
  }: {
    action: Function;
    navigation: any;
    navTo: String;
    message: String;
    buttonStyle: ButtonStyle;
  }): Button {
    return new ActionButton(action, message, buttonStyle);
  }
}
```

Figure 1.8. ActionButtonFactory.ts

```
1  import {Alert} from 'react-native';
2  import Button from './Button';
3  import ButtonStyle from './button-styles/ButtonStyle';
4
5  export default class MessageButton extends Button {
6    constructor(message: String, buttonStyle: ButtonStyle) {
7      super(() => Alert.alert(message), buttonStyle);
8    }
9  }
```

Figure 1.9. MessageButton.ts

```
1  import Button from './Button';
2  import ButtonStyle from './button-styles/ButtonStyle';
3  import MessageButton from './MessageButton';
4
5  export default class MessageButtonFactory {
6    createButton({
7      action,
8      navigation,
9      navTo,
10     message,
11     buttonStyle,
12   }: {
13     action: Function;
14     navigation: any;
15     navTo: String;
16     message: String;
17     buttonStyle: ButtonStyle;
18   }): Button {
19     return new MessageButton(message, buttonStyle);
20   }
21 }
```

Figure 1.10. MessageButtonFactory.ts

```
import {Alert} from 'react-native';
import Button from './Button';
import ButtonStyle from './button-styles/ButtonStyle';

export default class NavigationButton extends Button {
  constructor(
    action: Function,
    navigation: any,
    navigateOnSuccess: String,
    successMessage: String | null,
    buttonStyle: ButtonStyle,
  ) {
    super(async () => {
      if (action) {
        const errorMessage = await action();
        if (errorMessage) {
          Alert.alert(errorMessage);
          return;
        }
      }
      navigation.navigate(navigateOnSuccess);
      if (successMessage) {
        Alert.alert(successMessage);
      }
    }, buttonStyle);
  }
}
```

Figure 1.11. NavigationButton.ts

```
1  import Button from './Button';
2  import ButtonStyle from './button-styles/ButtonStyle';
3  import NavigationButton from './NavigationButton';
4
5  export default class NavigationButtonFactory {
6    createButton({
7      action,
8      navigation,
9      navTo,
10     message,
11     buttonStyle,
12   }: {
13     action: Function;
14     navigation: any;
15     navTo: String;
16     message: String;
17     buttonStyle: ButtonStyle;
18   }): Button {
19     return new NavigationButton(
20       action,
21       navigation,
22       navTo,
23       message,
24       buttonStyle,
25     );
26   }
27 }
```

Figure 1.12. NavigationButtonFActory.ts

Prototype Pattern

Within the composite pattern, we implemented the prototype pattern in the leaf nodes. This helps to efficiently create complex objects simply by copying existing ones. Which ultimately leads to saving time and resources. It also helps to promote more flexibility and adaptability in the design promoting better reusability as the prototype could be used in different parts of the code.

Justification for the Pattern

The clone method would be very useful for placing new geocaches. Geocaches could have many different configuration options, and it would be nice to be able to create new geocaches which have the same attributes, with only the coordinates differing. So, we could call the clone method to get a new geocache and then simply change the coordinates. This will look a lot cleaner in the code than instantiating a new geocache and passing lots of configuration options to the constructor.

Source Code

```
+ import DisplayItem from "./displayItem";
+ import React from 'react'
+
+ class otherUser implements DisplayItem {
+     name: string;
+     location: number;
+     avatar: string;
+     type: "USER";
+
+     constructor (name: string, location: number, avatar: string){
+         this.name = name;
+         this.location = location;
+         this.avatar = avatar;
+     }
+
+     clone(){
+         return new otherUser(this.name, this.location, this.avatar);
+     }
+
+     renderItem = () => {
+         //return name, location, avatar
+         return <><h4>{this.name}{this.location}{this.avatar}</h4> </>
+     };
+
+ }
+
+ export default otherUser;
```

Figure 2.1: Leaf Node otherUser Class Containing Prototype Pattern

```
+ import DisplayItem from "./displayItem";
+ import React from 'react'
+
+ class geocache implements DisplayItem {
+     name: string;
+     location: number;
+     avatar: string;
+     type: "GEOCACHE";
+
+     constructor (name: string, location: number, avatar: string){
+         this.name = name;
+         this.location = location;
+         this.avatar = avatar;
+     }
+
+     clone(){
+         return new geocache(this.name, this.location, this.avatar);
+     }
+
+     renderItem = () => {
+         //return name, location, avatar
+         return <><h4>{this.name}{this.location}{this.avatar}</h4> </>
+     };
+ }
+
+ export default geocache;
```

Figure 2.2: Leaf Node geocache Class Containing Prototype Pattern

Structural Design Patterns

Bridge Pattern

We implemented the Bridge Pattern to abstract Button styles into their own class hierarchy separate from the Button classes. The Button classes contain a ButtonStyle object and use it to apply graphics and layouts to themselves, so that Button Styles can be mixed and matched with different types of Buttons, without having to create a new class for each combination of Button Behaviour and Style.

Previous Implementation

- Previously in the app, UI elements were all created and styled inline in the JSX of whichever View they were being used in. This violated the single responsibility principle because functionality and style were both being done in the same instantiation. Implementing the Bridge pattern to abstract the button style into its own class hierarchy solves this problem.
- The open-close principle was also being violated in this case due to you not being able to extend the style of a button without modifying the functionality as well.
- Since button creation was moved into the factory method, we needed to decide where to store the style info. The TouchableOpacity style, and inner graphic of the button now needed to be passed into the factory method. We could store the graphic and the style directly in the Button class. However, out of all the buttons in the app, we had about four very similar styles, so it would be nice to be able to create these styles in one place, and only need to be able to specify the parameters that change between them (for example the image file or text).

```
<TouchableOpacity onPress={onPressSignOut}>
  <Text style={{fontWeight: 'bold', color: 'black'}}>Logout</Text>
</TouchableOpacity>
<TouchableOpacity
    style={styles.ProfileStyle}
    activeOpacity={0.5}
    onPress={() => navigation.navigate('Profile', {username})}>
  <Image
    source={require('../../data/images/Profile.png')}
    style={styles.ImageIconStyle}
  />
  <Text style={{fontWeight: 'bold', color: 'black'}}>Profile</Text>
</TouchableOpacity>
```

```
<TouchableOpacity
  style={styles.MapStyle}
  activeOpacity={0.5}
  onPress={() => navigation.navigate('Geocaching', {username})}>
  <Image
    source={require('../../data/images/Map.png')}
    style={styles.ImageIconStyle}
  />
  <Text style={{fontWeight: 'bold', color: 'black'}}>Map</Text>
</TouchableOpacity>
<TouchableOpacity style={styles.SearchStyle} activeOpacity={0.5}>
  <Image
    source={require('../../data/images/Search.png')}
    style={styles.ImageIconStyle}
  />
  <Text style={{fontWeight: 'bold', color: 'black'}}>Search</Text>
</TouchableOpacity>
```

Code Excerpt 3.1 (MainMenu.js) *The original implementation of button styling in the View. Note that each button has two components which affect the look and feel of the button. There is a style attribute on the TouchableOpacity component, and an inner graphic component, such as Text, Image, or Text and Image. Throughout the app, the many buttons fall into similar types of styles. Note how three of the buttons have similar inner graphics, with the only differences being the image file and the string of text. It would be nice to abstract away the creation of the styles and graphics, and only need to specify those varying attributes.*

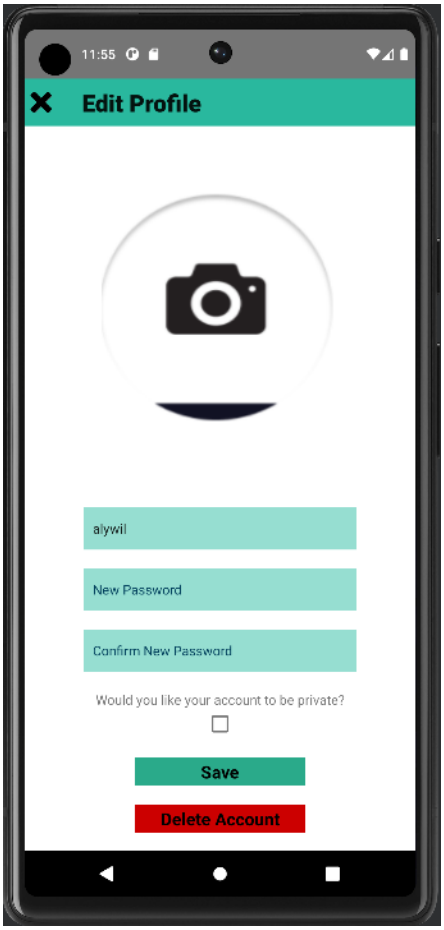


Figure 3.2 *"Save" and "Delete" are examples of buttons which have similar styles. The parameters that are different between them are the text and the color. In the top left corner, there is a back button (the X). Back buttons are found on nearly every page throughout the app, and are always styled the same way. Instead of redefining that style on every page where it is needed, it should be defined in one location in the code.*

Justification for the Pattern

Since we are creating our buttons using the Factory Method, we might be tempted to pass in the varying style parameters to the Factory Method as well, and have the style/graphic creation done in the Button classes. However, not every button of the same type will look the same. Some action buttons may have text on them, and some may have only an icon. In this case we would need to split our ActionButton class into an ActionTextButton and an ActionIconButton, and we would also need a new factory class for each of these new buttons, increasing the complexity of the code. Aside from this requiring many extra classes, we can realize that there is no reason to couple the button actions and styles in this way, since changing the style has no effect on the action of the button.

With the need for different types of button styles, it made sense to use the Bridge Pattern to develop Button behaviour and styles as separate hierarchies of classes. We can pass in a ButtonStyle to the Button constructor and the button can get it's outer style and inner graphic

from the ButtonStyle class. The button class itself is not concerned with whether it is going to display an image or text, because this does not affect the functionality of the button.

Figure 3.3 *Examples of icon buttons with text. The search button is a message button (for now as the functionality has not been implemented yet), and the profile and map buttons are navigation buttons. This illustrates how buttons that behave differently can look the same, justifying the need to split apart the Button and ButtonStyle class hierarchies.*

Drawbacks of the Pattern

Implementing this pattern requires coupling our Views to the different ButtonStyle classes, because they need to instantiate the concrete ButtonStyle subclasses to pass to the Button Factory on button creation. Therefore, we are taking the risk that we might need to make changes to the ButtonStyle class in the future, and require changing all of our Views to maintain compatibility, violating the Open-Close principle. However we believe that overall the maintainability of the code is increased, because we are guaranteed to have consistent button appearances, and if we want to make a change like making all icon buttons a different size, we only need to make that change in one place.

Pattern Implementation

As shown in the UML in Appendix A, we have abstracted the ButtonStyle component from the rest of the Button functionality, creating two separate hierarchies.

1. The Button class is the abstraction in this pattern,
2. The concrete button classes (NavigateButton, etc.) are the refined abstractions.
3. The ButtonStyle is the implementation,
4. The subclasses of ButtonStyle are the concrete implementations.

Buttons have a ButtonStyle field, and use the style and graphic attributes of the ButtonStyle to set the style of the component created in the Button class. This allows us to have various button styles and graphics for different types of buttons. This makes both the button style and button functionality extensible without impacting the other. Doing so solves the single-responsibility principle and the open-close principle by having the functionality of the button and the style of the button separate.

Source Code

```
export default class ButtonStyle {
  _style: Object
  _graphic: JSX.Element

  constructor(style: Object, graphic: JSX.Element) {
    this._style = style;
    this._graphic = graphic;
  }

  get style(): Object {
    return this._style;
  }

  get graphic(): JSX.Element {
    return this._graphic;
  }
}
```

Figure 3.4. ButtonStyle.ts

```
import React from 'react';
import {Text} from 'react-native';
import ButtonStyle from './ButtonStyle';

export default class CenterButtonStyle extends ButtonStyle {
  constructor(text: String, backgroundColor: String = '#2AAA8A', fontSize: number = 18, width: number = 180) {
    super(
      {
        width,
        height: (fontSize * 5) / 3,
        alignItems: 'center',
        justifyContent: 'center',
        backgroundColor,
      },
      <Text style={{fontWeight: 'bold', fontSize, color: 'black'}}>
        {text}
      </Text>,
    );
  }
}
```

Figure 3.5. CenterButtonStyle.tsx

```tsx
import React from 'react';
import {Image, Text} from 'react-native';
import ButtonStyle from './ButtonStyle';

export default class IconButtonStyle extends ButtonStyle {
  constructor(image: String, text: String | null, iconSize: number = 40) {
    super(
      {...{alignItems: 'center'}},
      <>
        <Image
          source={image}
          style={{
            width: iconSize,
            height: iconSize,
            aspectRatio: 1,
            resizeMode: 'stretch',
          }}
        />
        {text ? (
          <Text style={{fontWeight: 'bold', fontSize: 18, color: 'black'}}>
            {text}
          </Text>
        ) : null}
      </>,
    );
  }
}
```

Figure 3.6. IconButton.tsx

```
import React from 'react';
import {Text} from 'react-native';
import ButtonStyle from './ButtonStyle';

export default class TextButtonStyle extends ButtonStyle {
  constructor(text: String) {
    super(
      {
        padding: 10,
      },
      <Text style={{fontWeight: 'bold', fontSize: 18, color: 'black'}}>
        {text}
      </Text>,
    );
  }
}
```

Figure 3.7. TextButton.tsx

Composite Pattern

For this project, when we started taking a deeper look into the underlying functionality of the geocache map we noticed that overall it needed to exhibit better structural modularity and reusability and was far more complicated than necessary. The map itself was just a combination of objects needing to be rendered without distinguishing between the things. Which given the composite pattern is based on the need to create a hierarchical structure of objects and is extremely helpful when dealing with objects that consist of different behaviors and attributes. We saw this as an excellent fit to be implemented.

Justification for the Pattern

The composite pattern makes sense here because the Geocache Screen needs to interact with several different types of objects in the same way. We need to show other users and geocaches on the screen. Clearly these are very different types of objects with very different behaviours. However, as far as the map object is concerned, these objects only need to have two attributes: a location, and a graphic to be displayed. The composite pattern can be used to define an interface that all objects that will be displayed on the map will follow. The map can then iterate over a list of elements without knowing what their concrete classes are, as it only needs to work with them through the DisplayItem interface.

Pattern Implementation

As shown with a UML in Appendix C, we have leaf nodes consisting of OtherUser and Geocache. These are both components of DisplayItem which is the interface that regulates the

leaf notes with the requirement to be able to render any leaf node when instantiated. We then have a composite of items denoted as DisplayItems which takes consistent of various leaf notes to add to the composite. As for the overall functionality, it takes place within a service screen characterized as GeoCacheScreenService.

Source Code

```
+ import displayItems from "../DataModels/displayItems";
+ import geocache from "../DataModels/geocache";
+ import otherUser from "../DataModels/otherUser";
+ import ItemIterator from "../DataModels/ItemIterator";
+
+ class GeocacheScreenServices {
+     items: displayItems;
+     iterator: ItemIterator;
+
+     constructor(){
+         this.items = new displayItems();
+         this.items.add(new geocache("Geocache 1", 100, "Avatar 1"));
+         this.items.add(new geocache("Geocache 2", 200, "Avatar 1"));
+         this.items.add(new geocache("Geocache 3", 250, "Avatar 1"));
+         this.items.add(new geocache("Geocache 4", 400, "Avatar 1"));
+         this.items.add(new geocache("Geocache 5", 790, "Avatar 1"));
+         this.items.add(new otherUser("User 2", 150, "Avatar 2"));
+         this.items.add(new otherUser("User 3", 400, "Avatar 3"));
+         this.items.add(new otherUser("User 4", 650, "Avatar 4"));
+         this.items.add(new otherUser("User 5",1080, "Avatar 6"));
+         this.iterator = this.items.createIterator();
+     }
+
+
+     renderItems(userLocation: number){
+         while(this.iterator.hasMore()){
+             this.iterator.getNext(userLocation).renderItem();
+         }
+     }
+ }
```

Figure 4.1: Service Class

```
+ import DisplayItem from "./displayItem";
+ import displayItemsIterator from "./displayItemIterator";
+
+ class displayItems implements DisplayItem {
+     name: "";
+     location: 0;
+     avatar: "";
+     type: "COMPOSITE";
+
+     items: DisplayItem[] = [];
+
+     add(item: DisplayItem){
+         this.items.push(item);
+     }
+
+     remove(item: DisplayItem){
+         this.items = this.items.filter(i => i !== item);
+     }
+
+     renderItem(){
+         this.items.forEach(i => i.renderItem());
+     }
+
+     getChildren(){
+         return this.items;
+     }
+
+     createIterator(){
+         return new displayItemsIterator(this.items);
+     }
+ }
+
+ export default displayItems;
```

Figure 4.2: Composite Class

```
+ import DisplayItem from "./displayItem";
+ import React from 'react'
+
+ class otherUser implements DisplayItem {
+     name: string;
+     location: number;
+     avatar: string;
+     type: "USER";
+
+     constructor (name: string, location: number, avatar: string){
+         this.name = name;
+         this.location = location;
+         this.avatar = avatar;
+     }
+
+     clone(){
+         return new otherUser(this.name, this.location, this.avatar);
+     }
+
+     renderItem = () => {
+         //return name, location, avatar
+         return <><h4>{this.name}{this.location}{this.avatar}</h4> </>
+     };
+
+ }
+
+ export default otherUser;
```

Figure 4.3: Leaf Node otherUser Class

```
1  + import DisplayItem from "./displayItem";
2  + import React from 'react'
3  +
4  + class geocache implements DisplayItem {
5  +     name: string;
6  +     location: number;
7  +     avatar: string;
8  +     type: "GEOCACHE";
9  +
10 +     constructor (name: string, location: number, avatar: string){
11 +         this.name = name;
12 +         this.location = location;
13 +         this.avatar = avatar;
14 +     }
15 +
16 +     clone(){
17 +         return new geocache(this.name, this.location, this.avatar);
18 +     }
19 +
20 +     renderItem = () => {
21 +         //return name, location, avatar
22 +         return <><h4>{this.name}{this.location}{this.avatar}</h4> </>
23 +     };
24 + }
25 +
26 + export default geocache;
```

Figure 4.4: Leaf Node geocache Class

Behavioral Design Patterns

Iterator Pattern

In a previous example, we used the composite pattern which provides a hierarchy of objects with different behaviors. Given we are looking to implement a map that will showcase all of these items within the map we saw this as a good opportunity to implement the iterator behavioral pattern.

Justification for the Pattern

The iterator pattern itself is a great way to encapsulate the collection of objects being traversed making the reusability of the code far more easy. This makes the code far more flexible by being able to iterate without changing the collection itself and provides flexibility in accessing and processing the objects within the collection. This ultimately leads to a more efficient design residing the memory and processing time needed to traverse large collections.
For a geocaching app such as this, iterators could be useful for iterating over a collection of objects to find the one that is nearest to the user, or sort a list in order of distance of objects to the user. It could also be used, to iterate over the list of all objects in the user's city, skipping over objects that are not close enough to be displayed on the map, to create a new list of just the objects we need to display. We could have different settings for the geocache screen, such as one that allows you to view only geocaches placed by friends. In that case we could define an iterator to find objects that need to be displaced, skipping over geocaches which were not placed by friends. We can see from all these suggestions how we may need to traverse the same list of items in different ways, so by implementing the iterator pattern we can ensure that all these traversals can be implemented easily by following the common iterator interface.

Pattern Implementation

The iterator pattern as shown in the UML in Appendix D was implemented by having the component of the composite pattern act as the iterator as shown in the DisplayItem class. Along with the DisplayItems composite class being the concrete iterator. We then added in the ItemIterator to futile the duties of an iterableCollection then finally DisplayItemIterator as a ConcreateCollection.

Source Code

```
+ import displayItems from "../DataModels/displayItems";
+ import geocache from "../DataModels/geocache";
+ import otherUser from "../DataModels/otherUser";
+ import ItemIterator from "../DataModels/ItemIterator";
+
+ class GeocacheScreenServices {
+     items: displayItems;
+     iterator: ItemIterator;
+
+     constructor(){
+         this.items = new displayItems();
+         this.items.add(new geocache("Geocache 1", 100, "Avatar 1"));
+         this.items.add(new geocache("Geocache 2", 200, "Avatar 1"));
+         this.items.add(new geocache("Geocache 3", 250, "Avatar 1"));
+         this.items.add(new geocache("Geocache 4", 400, "Avatar 1"));
+         this.items.add(new geocache("Geocache 5", 790, "Avatar 1"));
+         this.items.add(new otherUser("User 2", 150, "Avatar 2"));
+         this.items.add(new otherUser("User 3", 400, "Avatar 3"));
+         this.items.add(new otherUser("User 4", 650, "Avatar 4"));
+         this.items.add(new otherUser("User 5",1080, "Avatar 6"));
+         this.iterator = this.items.createIterator();
+     }
+
+
+     renderItems(userLocation: number){
+         while(this.iterator.hasMore()){
+             this.iterator.getNext(userLocation).renderItem();
+         }
+     }
+ }
```

Figure 5.1: Service class for Iterator/Composite Pattern

```
+ import DisplayItem from "./displayItem";
+ import ItemIterator from "./ItemIterator";
+
+ class displayItemsIterator implements ItemIterator {
+     items: DisplayItem[];
+     index: number = 0;
+
+     constructor(items: DisplayItem[]){
+         this.items = items;
+     }
+
+     getNext(from: number){
+         // return this.items[this.index++];
+         if(this.items[this.index].location < from){
+           return  this.index++;
+         }
+     }
+
+     hasMore(): boolean {
+         return this.index < this.items.length;
+     }
+ }
+
+ export default displayItemsIterator;
```

Figure 5.2: displayItemsIterator Implementing Functionality of ConcreteeCollection

```
+ interface ItemIterator{
+     getNext(from:number);
+     hasMore(): boolean;
+ }
+
+ export default ItemIterator;
```

Figure 5.3: ItemIterator Implementing Functionality of IterableCollection

Other Design Patterns

Mediator Pattern

The main geocaching screen violates the single responsibility principle because it handles both the view and the backend logic for picking up and placing geocaches. A solution to this problem would be to separate the geocaching screen view from the backend logic, and create a mediator class to handle communication between the different components and functionality of the app.

This would solve the problem of the single-responsibility principle and would be a great addition to the application at a future date.

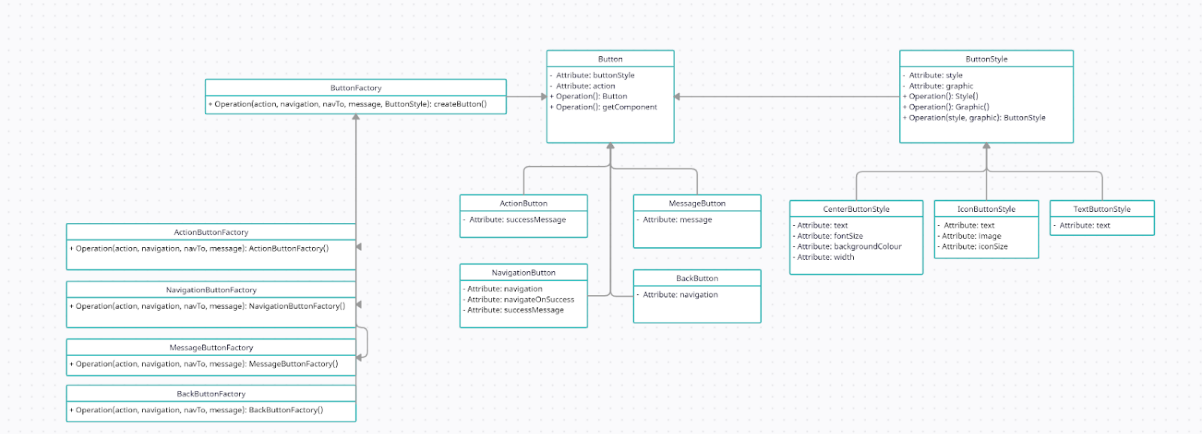
Singleton Pattern

For this project, we initially decided we were going to implement the singleton pattern to instantiate the database. This would strongly benefit the system because it would mean all the data for the system would have a global access point. After further discussion and research, we came to the decision that this pattern does not solve any real problem within the application as the mongodb client is already acting as a singleton. The mongodb client will only have one instance unless otherwise specified, and the client creates a single point of access that the entire application can access.

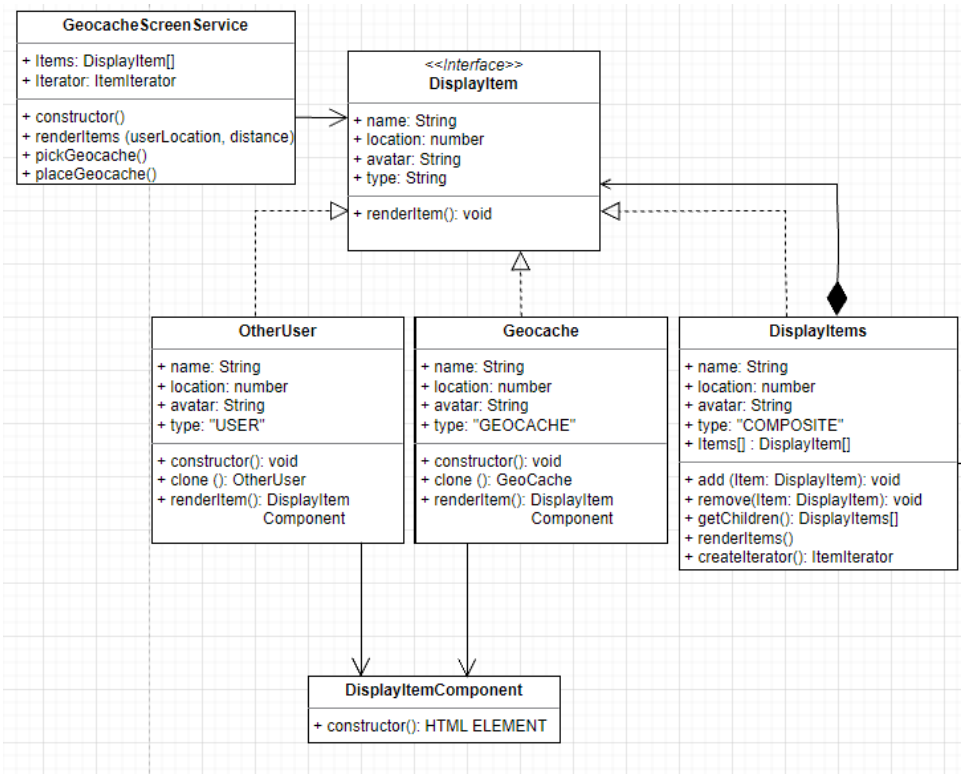
Factory Pattern

There is a problem with how geocaches are created. Similarly to the button creation, the current implementation violates the single responsibility principle and the open-close principle by instantiating each geocache with all of its details with the "new" instantiation when it is required. We decided not to implement this pattern in this part of the code because there is currently only one type of geocache and therefore doesn't provide as much value to the system as it does for the buttons which have multiple different types implemented in the code. For the purposes of this project, the second implementation of this pattern would be redundant and our time would be spent better implementing other patterns, however, it should be implemented in the future.

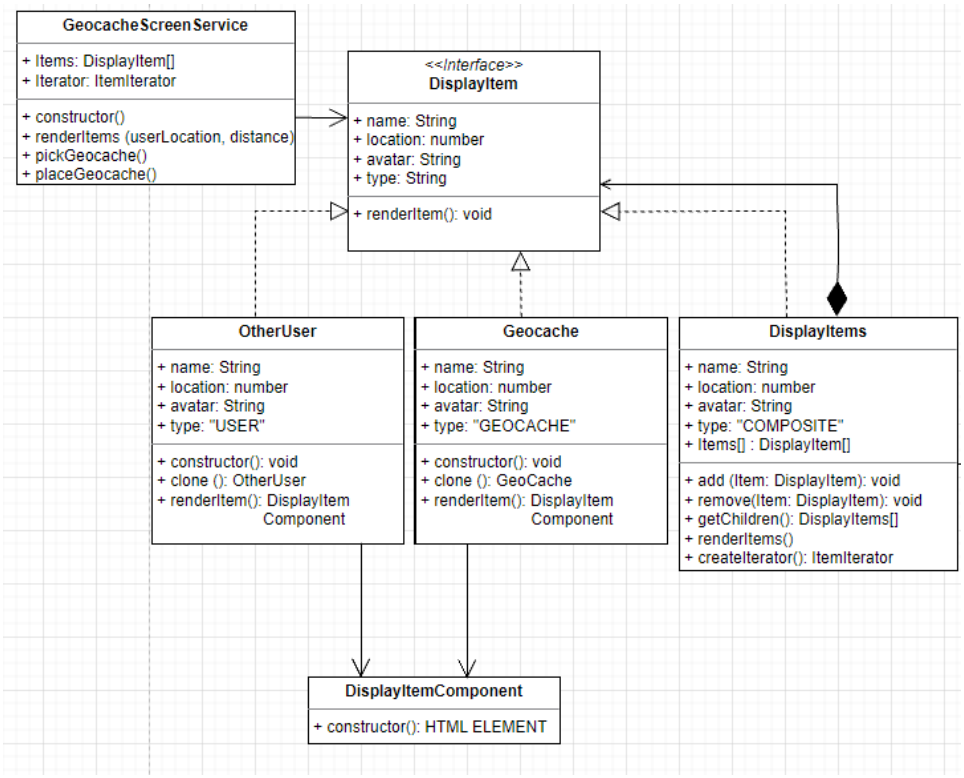
Appendices



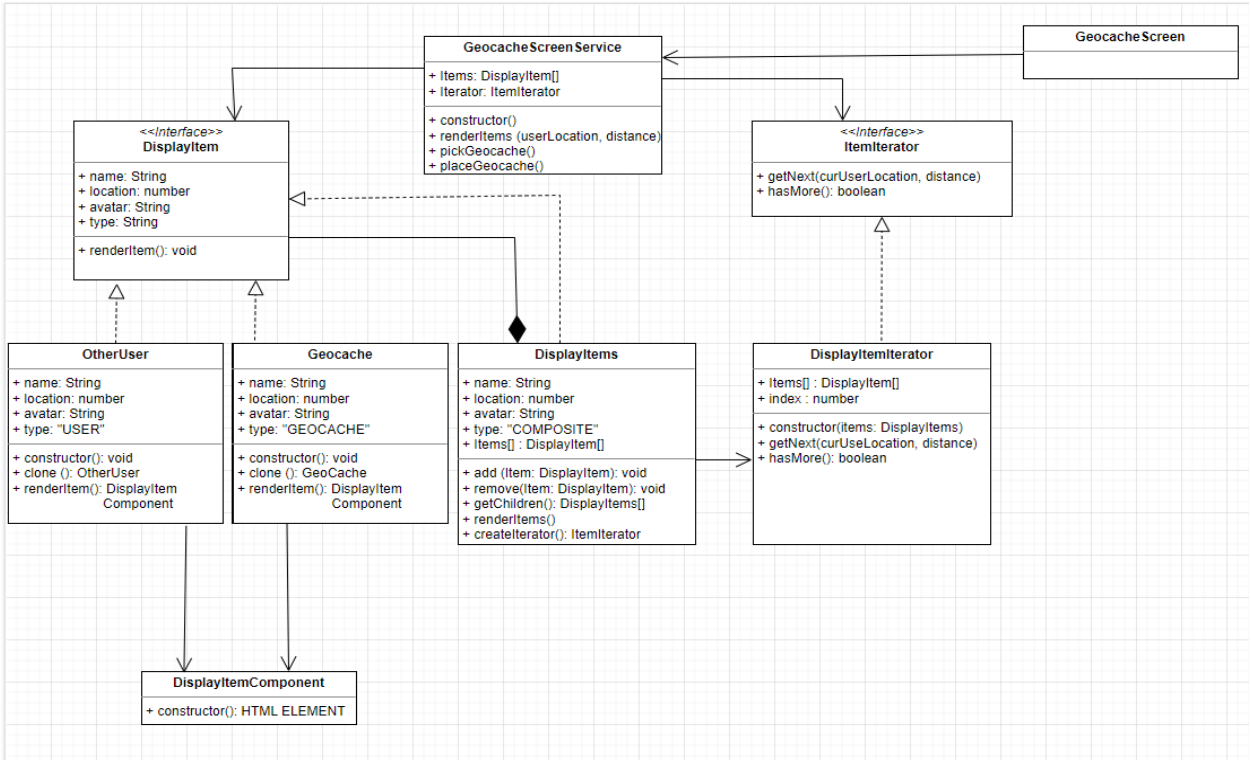
Appendix A



Appendix B - Implementation of Prototype Pattern within Composite Pattern



Appendix C - Implementation of Composite Pattern on GeocacheScreen



Appendix D - Implementation of Iterator Pattern used on Prototype Pattern within Composite Pattern

Contributions

Our group has met every week on Tuesday morning for an hour before class to discuss the application and how it can be improved. These were both brainstorming sessions of which patterns would be best to implement for the project, as well as discussing the practical application of how the pattern would be implemented. We created UML diagrams and planned out how the project would look after the patterns were implemented. We had divided the patterns between the four of us to implement based on where they were located in the app. There were two main parts of the code that ended up having the refactor changes, the UI buttons and the geocaching screen. Haniya implemented the code in the geocaching screen including the Prototype, composite, and iterator patterns. Jared wrote the documentation for the geocaching screen including the Prototype, composite, and iterator patterns. Alyssa implemented the code for the UI buttons including the Factory and Bridge patterns and Sam wrote the documentation and UML diagrams for the UI Buttons including the Factory and Bridge patterns. Alyssa also added some further justification and ideas for extending the patterns to each section of the documentation. Sam attempted to implement the singleton pattern and factory pattern for the database, however after research and attempting to create a local version of the MongoDB, determined it would not be a useful change to the system. We decided it was

best to have the patterns that were closely related in the application to be implemented and demoed by one person to keep consistency within the code in the app.