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

Previously the code for the creation of various UI elements in the App was done by a "new" instantiation with its styles and behaviour specified inline at the time of rendering the page (See Figure 1). This violates the Open-Close SOLID principle because it needs to be recompiled every time there is a deployment change. Implementing the Factory method for button creation is one way we can solve this problem. 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.

```
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>
```

Figure 1. *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. 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.*

Buttons in the ARGeocache app have three main behaviours. They could display messages in the form of alerts, trigger navigation to different screens in the app, and/or perform an action (such as logging in the user). Our buttons naturally fall into these classes:

- A parent **Button** class which takes an action and styling and returns a button.
- A **MessageButton** class. This button simply displays a message upon being clicked. Useful for buttons who's actions have not been implemented yet.
- An **ActionButton**. This button performs an action, and then may display a message about the result of the action.
- A **NavigationButton**. This button triggers navigation through the app. The navigation may be contingent on a certain action succeeding, (for example, only navigate to the home screen if the "Log In" action was successful).
- A **BackButton.** This button simply returns to the previous screen. It does not perform any other action or display any messages.

As shown in the UML in Appendix A, we have created an abstract product button with multiple concrete products such as NavigationButton, ActionButton, MessageButton, and BackButton. We've then created an abstract factory method ButtonFactory and its concrete implementations for each of their respective concrete products. 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:

```
1  import React from 'react';
2  import {TouchableOpacity} from 'react-native';
3  import ButtonStyle from './button-styles/ButtonStyle';
4
5  export default class Button {
6    buttonStyle: ButtonStyle
7    action: Function
8
9    constructor(action: Function, buttonStyle: ButtonStyle) {
10     this.buttonStyle = buttonStyle;
11     this.action = action;
12   }
13
14   get component(): JSX.Element {
15     return (
16       <TouchableOpacity style={this.buttonStyle.style} onPress={this.action}>
17         {this.buttonStyle.graphic}
18       </TouchableOpacity>
19     );
20   }
21 }
```

Figure 1. 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 2. 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 3. 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 4. 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 5. 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 6. 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 7. 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 8. 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 9. NavigationButton.ts

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

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

Figure 10. NavigationButtonFActory.ts

Prototype Pattern:

Within the composite pattern, we also 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.

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 11: 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 12: Leaf Node geocache Class Containing Prototype Pattern

Structural Design Patterns

Bridge Pattern

Previously in the app, UI elements in the app were all created and styled in whichever page they were being used. 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 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.

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. This allows us to have various button styles and graphics for different types of buttons. This makes both the button style and 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:

```
1  export default class ButtonStyle {
2    _style: Object
3    _graphic: JSX.Element
4
5    constructor(style: Object, graphic: JSX.Element) {
6      this._style = style;
7      this._graphic = graphic;
8    }
9
10   get style(): Object {
11     return this._style;
12   }
13
14   get graphic(): JSX.Element {
15     return this._graphic;
16   }
17 }
```

Figure 13. ButtonStyle.ts

```tsx
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 14. 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 15. 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 16. 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.

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:

```
1  + import displayItems from "../DataModels/displayItems";
2  + import geocache from "../DataModels/geocache";
3  + import otherUser from "../DataModels/otherUser";
4  + import ItemIterator from "../DataModels/ItemIterator";
5  +
6  + class GeocacheScreenServices {
7  +     items: displayItems;
8  +     iterator: ItemIterator;
9  +
10 +     constructor(){
11 +         this.items = new displayItems();
12 +         this.items.add(new geocache("Geocache 1", 100, "Avatar 1"));
13 +         this.items.add(new geocache("Geocache 2", 200, "Avatar 1"));
14 +         this.items.add(new geocache("Geocache 3", 250, "Avatar 1"));
15 +         this.items.add(new geocache("Geocache 4", 400, "Avatar 1"));
16 +         this.items.add(new geocache("Geocache 5", 790, "Avatar 1"));
17 +         this.items.add(new otherUser("User 2", 150, "Avatar 2"));
18 +         this.items.add(new otherUser("User 3", 400, "Avatar 3"));
19 +         this.items.add(new otherUser("User 4", 650, "Avatar 4"));
20 +         this.items.add(new otherUser("User 5",1080, "Avatar 6"));
21 +         this.iterator = this.items.createIterator();
22 +     }
23 +
24 +
25 +     renderItems(userLocation: number){
26 +         while(this.iterator.hasMore()){
27 +             this.iterator.getNext(userLocation).renderItem();
28 +         }
29 +     }
30 + }
```

Figure 17: 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 18: 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 19: Leaf Node otherUser Class

```
+ 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 20: 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 differentiating 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.

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 iterator 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.

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 21: Service class for Iterator/Composite Pattern

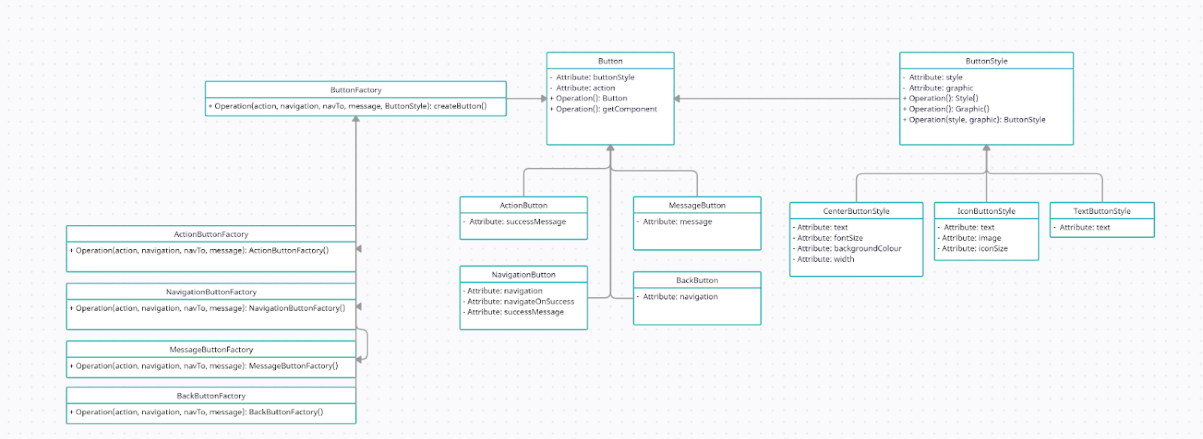
```
+ import DisplayItem from "./displayItem";
+ import ItemIterator from "./ItemIterator";
+
+ class displayItemsIterator implements ItemIterator {
+     items: DisplayItem[];
+     index: number = 0;
+
+     constructor(items: DisplayItem[]){
+         this.items = items;
+     }
+
+     getNext(from: 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 22: displayItemsIterator Implementing Functionality of ConcreteeCollection

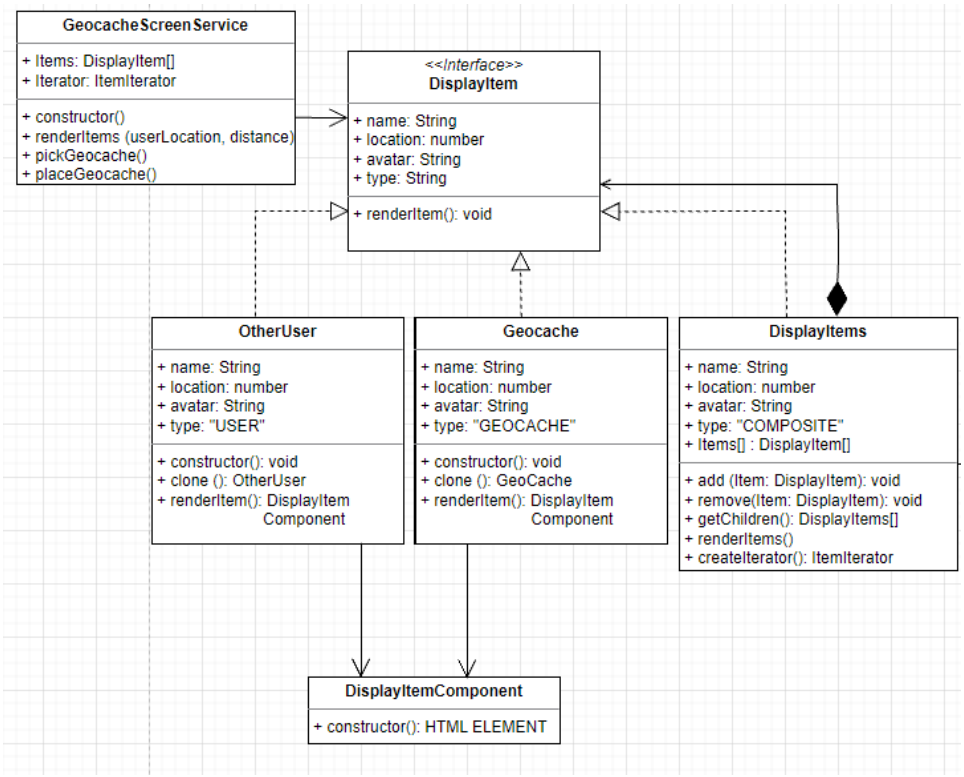
```
+ interface ItemIterator{
+     getNext(from:number);
+     hasMore(): boolean;
+ }
+
+ export default ItemIterator;
```

Figure 23: ItemIterator Implementing Functionality of IterableCollection

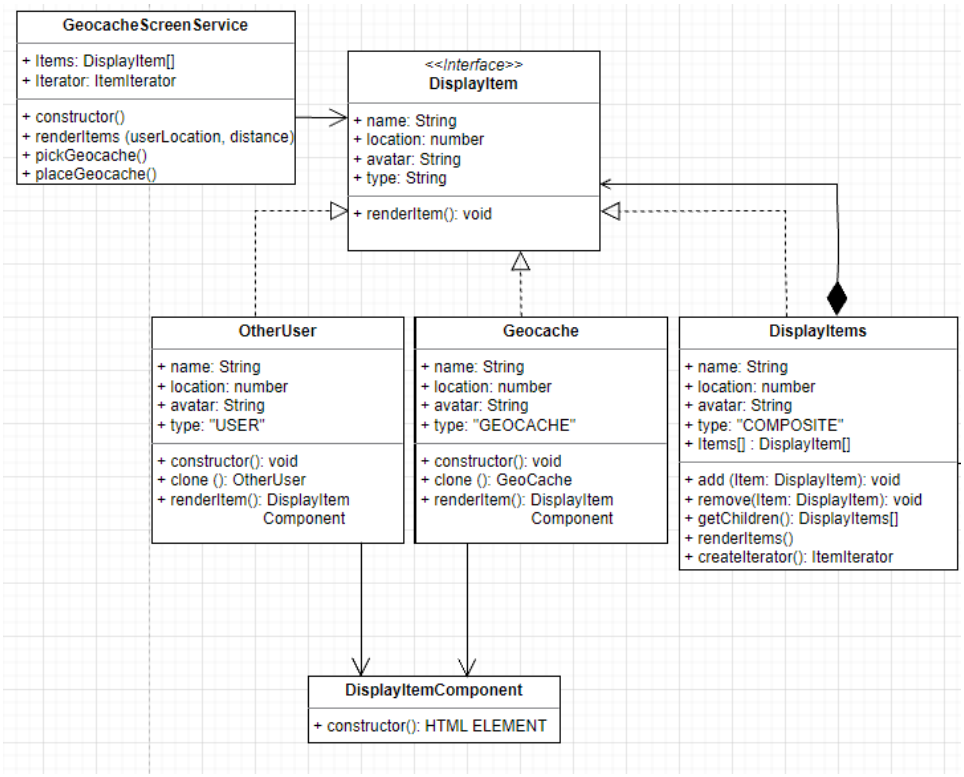
Appendices



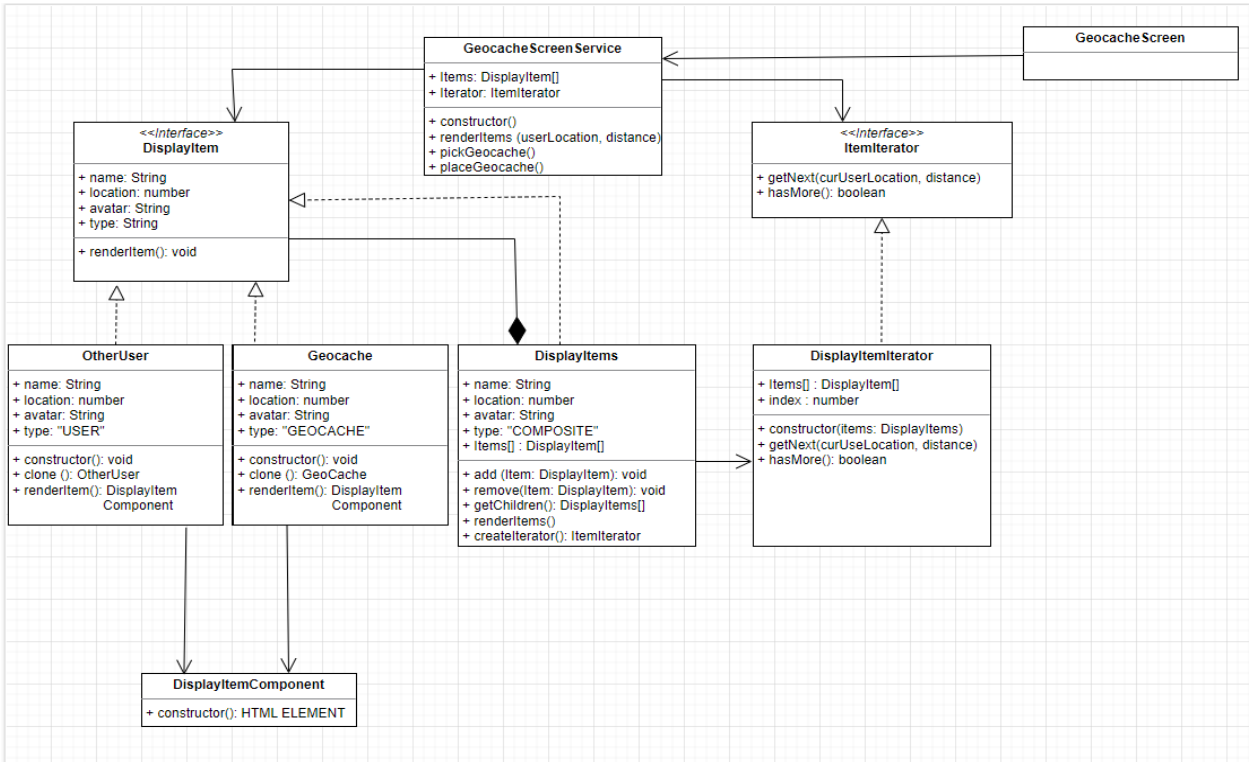
Appendix A



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