Phase 4 Design Report

Name: Liwen Tian
UTeid: lt22968
Group 8
Link:https://github.com/Lena-Liwen-Tian/PumpkinMovie1

Information Hiding

To apply the Information Hiding to my project, I divided my app into different modules. Since I used React to write the front end, I divided my app into different components. In this Phase, I divided my app into even more components. For instance, I created a new component for http fetch. I feel that http fetch usually depends on the stability of the network so that it has a higher chance to be affected. Encapsulating it can make the further change easier. When some components want to use the http fetch, it only need to take care of the url and then sendRequest(url),
Http fetch module can help handle all other things, including setIsLoading and setError. Besides, I created some environment variables to store the API keys and the backend url. I also separated the front end and back end for the sake of information hiding since I feel that both of them may require some changes in the future. As a result , other components except http fetch can happily enjoy the result even without knowing what happened. There are a lot of other examples related to information hiding in my app. I created the form module responsible for form submit, input module taking responsibility of handling user input, auth module specifically used for authentications and etc. Each small module has their own small task and ready to be combined for a larger task.

This is my first time to do modularization in my app, I think it undoubtedly makes the change easider in the future. I can use my back end as an example. My backend codes are separated into a lot of controllers and routes. For this app, I used one route for a model. So if more routes are added, the other routes will not be affected. Different routes are executed by their controllers. Inside the controllers, one method handles one type of request. In my Cinema controller, I have getCinema and getCinemabyId, if one of them fails, the other one will not be affected. The controller also allows you to add more requests type you want to handle in the future. So general speaking, the app is prepared for the change later.

However, my app also has some disadvantages. To be honest, I still feel that my front end code can be improved more. I divided my movie page into different subclasses like filter, search and etc. But I feel that they still depend on the same variable LoadedMovie. If the LoadedMovie is not fetched successfully, a lot of things will fail. I do not have a specific solution now (if I have, this will not be the problem left). I guess, maybe I can use some abstract classes or abstract methods. I am not good at using abstract things, I hope I can handle it after more practice.

Design Pattern

Since I used React as my front end language, I picked up two design patterns within my application. The first one is Decorator Design Pattern. Some people called it the Higher-Order-Component Design Pattern in React. It is very popular for organizing the codes written in React. The main purpose of Decorator Design Pattern is to "Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality." In my original front end code, I created a Button component, I used it as a shared component in different places within my app. Though I have already implemented it on the basis of the plain <button> tag. I still wanted more variations and more features. For instance, when using Button in the login page, I want all texts inside the Button to be upper case letters. I also wanted to use different colors in different places. If I change them in place, I need to use a lot of styles which lead to too many duplicated codes. So I think this is a good chance to apply the Decorator Design Pattern. I passed the Button component as a parameter to different functions and converted it into different styles ahead of time. The advantage is that it saves me a lot of time when inserting them to different places since they are customized ahead of time. It also reduces the duplicated codes. The disadvantage is that I need to remember to import this new component every time. Sometimes, I may misuse the <Button> without decorators.

Below are the codes of my new Button components, it can either link to internal web pages or web pages outside.

```
const upper = (Button) =>(props)=>{
  class UpperButton extends React.Component{
    render(){
      return (
        <Button {...props}>
          {props.children.toString().toUpperCase()}
        </Button>
      )
    }
  }
  return <UpperButton/>;
}
```

```
const lower = (Button) =>(props)=>{

  class LowerButton extends React.Component{
    render(){
      return (
        <Button {...props}>
          {props.children.toString().toLowerCase()}
        </Button>
      )
    }
  }
  return <LowerButton/>;
}
```

```
const color = (Button) =>(props)=>{
 const mystyle = {backgroundColor:"DodgerBlue"}
 class BlueButton extends React.Component{
   render(){
     return (
       <Button style={mystyle} {...props}>
         {props.children}
       </Button>
     )
   }
 }
 return <BlueButton/>;
}

const BlueButton = color(Button);
```

When putting this new design into use. My front-end codes have some changes.

Before: (I write it a lot of times)

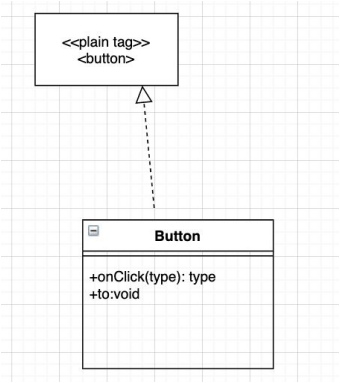
<Button style={{backgroundColor: “DodgerBlue”}} to
={`/${props.id}/theaterinfo`>More</Button>
<Button className= “lowercase” >VIEW ON MAP</Button>
<Button className = “uppercase” inverse onClick = {switchModeHandler}>
SWITCH TO {isLoginMode ? ‘SIGNUP’ : ‘LOGIN’}
</Button>

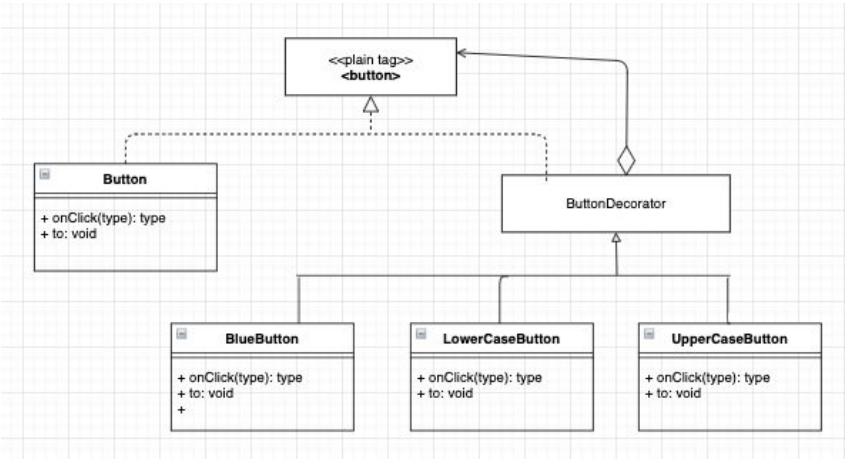
After:

<BlueButton to ={`/${props.id}/theaterinfo`}More</BlueButton>
<LowerButton>VIEW ON MAP</LowerButton>
<UpperButton inverse onClick = {switchModeHandler}>
SWITCH TO {isLoginMode ? ‘SIGNUP’ : ‘LOGIN’}
</UpperButton>

UML Diagrams:

Change from top to bottom





I used the Observer Pattern in React from the beginning of building my application, so I will not compare the before and after change for this Design Pattern. As is known to us all, the Observer Pattern is aimed at “defining a one - to - many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically”. In my applications there are some components changes that require the dependents to change at the same time. A good example is that when the sort method changes, both the current page contents and the title of the sort button will be changed together. So I created a SortSubject, this is the sort method used now. I also created some observers for it. One observer is the current content, this is used to update the current page content. The other observer is the button title, this is used to update the appearance of the sort button when different sort methods are used. I used attach and detach to register and unsubscribe the subject’s notifications. The advantage of this is that the change will be easier to track since the dependencies are clear. The disadvantage is that we have to remember to detach the dependent if it is not in use anymore. Actually, I think Redux can save even more codes than this Observer Pattern, so I change to Redux in the end.

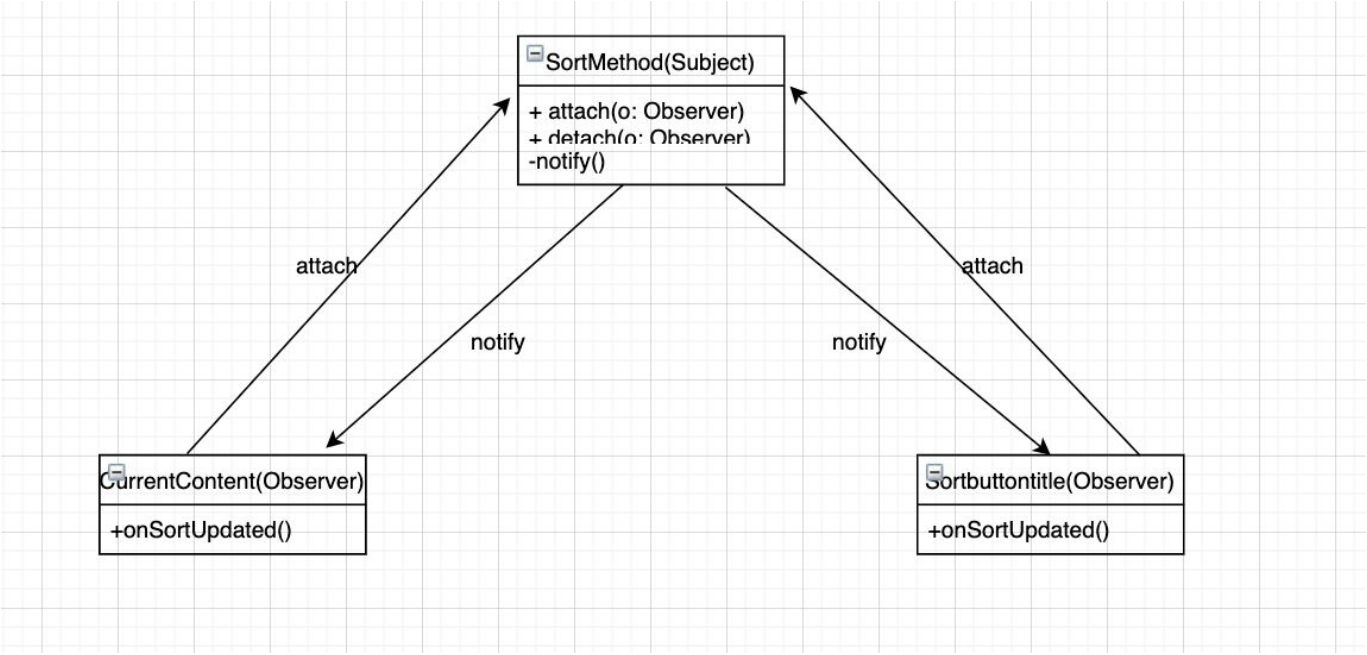
```
//the SortSubject
export type SortObserver = (method:String) => void;
class SortSubject {
   private observers: SortObserver[] = [];
   public attach(observer: SortObserver){
      this.observers.push(observer);
   }
   public detach(observer: SortObserver){
      this.observers = this.observers.filter(observers => observerToRemove!== observer)
   }
   public method(method: String){
      this.notify(method)
   }
   private notify(method:String){
      this.observers.forEach(observer => observer(s))
   }
}
const SortSubject = new SortSubject()
export default SortSubject

//observer1 current content
export const currentcontent: React.FC = () =>{
   const[currentsort,setcurrentsort] = useState<String>()
   const onSortUpdated: SortObserver = (method:String) =>{setcurrentsort(method)}
}
useEffect(() =>{
```

```
    SortSubject.attach(onSortUpdated)
    return() => SortSubject.detach(onSortUpdated);},[]
)
return (
    <LoadedMovies currentsort = {currentsort}/>
)

//observer two sort button title
export const buttontitle: React.FC = () =>{
    const[currentsort,setcurrentsort] = useState<String>()
    const onSortUpdated: SortObserver = (method:String) =>{setcurrentsort(method)}
}
useEffect(() =>{
    SortSubject.attach(onSortUpdated)

    return() => SortSubject.detach(onSortUpdated);},[]
)
return (
    <Sortbutton title = {currentsort}/>
)
```



Refactorings

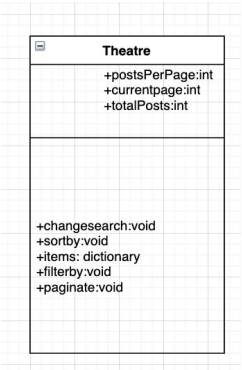
I also used some refactoring methods in this phase. I have three examples here.

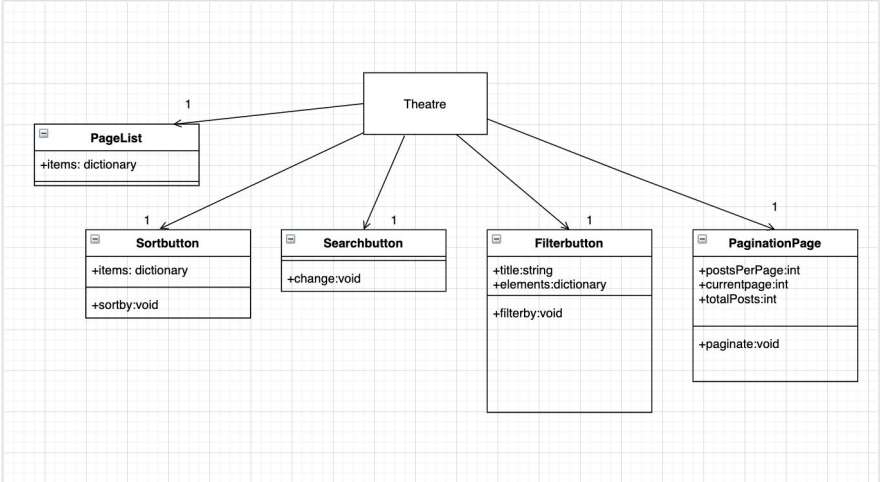
1.The first one is extract class. At the very beginning, I have very long class contents for the main page of each model. Then I decide to divide the contents into different classes. And each subclass complements its own task. My code snippets and UML diagram are as follows.

```
  <td>
<Sortbutton className="sortbutton" items={["Name (A-Z)","Distance (nearby)"]} sortby={sortby} />
</td>
  <td className="tablecontent2">
 <Searchbutton placeholder="Search by cinema name or zipcode" change={searchtheatre}/>
  </td>
  <td className="tablecontent3">
  <Filterbutton title= "Distance" elements={elements.Distance} filterby={changeFilter}/>
  </td>


<td className="tablecontent">
  <PaginationPage className="page" postsPerPage={postsPerPage} currentpage={currentPage} totalPosts={LoadedTheatres.length} paginate={paginate}/>
  </td>
  </tr>
</table>
```

UML Diagram change from top to bottom



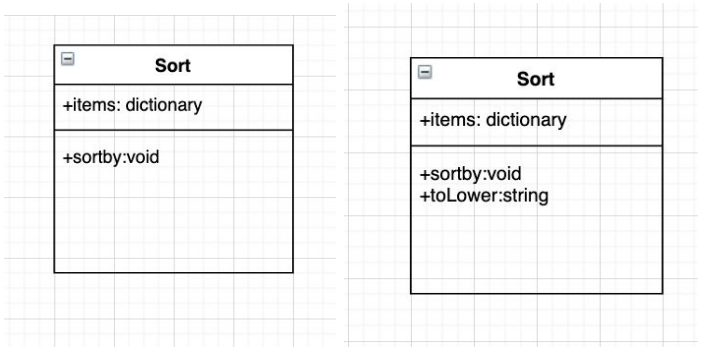


2.The second factoring method I used was extract methods. I found the codes used for converting the letters into lowercase inside the sort function were too duplicated. So I wrote a new method to combine some codes together.

```
 function toLower(a){
    return a.Title.toLowerCase();
  }
const sortby = method => {
    let movies;
    if(method === "Title (A-Z)"){
       movies = LoadedMovies.sort((a,b)=>{
       setSort(method);
       if(toLower(a) < toLower(b)) return -1;
       if(toLower(a) > toLower(b)) return 1;
       return 0;
     })
```

Before this change, the code looks like
If (a.Title.toLowerCase() < b.Title.toLowerCase()) return -1;
If (a.Title.toLowerCase() > b.Title.toLowerCase()) return -1;

The UML Diagram is changed from left to right



3. I also used extract method inside the search method in order to standardize the search contents.

```
function standard(character){
    return character.replace(/\s/g,'').toLowerCase();
  }
const searchmovie = e =>{
   setSearch(e);
   let title;
```

```
   let description;
   title = originalmovies.filter((contact) =>{
     return standard(contact.Title).indexOf(standard(e)) !== -1 ;
   });
   description = originalmovies.filter((contact) =>{
   return standard(contact.Actors).split(',').join('').indexOf(standard(e)) !== -1 ;
  });
   setLoadedMovies(title.length > 0 ? title:description);
 }
```

At the very beginning,
The search function looks like:

```
const searchmovie = e =>{
   setSearch(e);
   let title;
   let description;
   title = originalmovies.filter((contact) =>{
     return
contact.Title.replace(/\s/g,'').toLowerCase().indexOf(e.replace(/\s/g,'').toLowerCase(
)) !== -1 ;
   });
   description = originalmovies.filter((contact) =>{
   return
contact.Actors.replace(/\s/g,'').toLowerCase()split(',').join('').indexOf(e.replace(/\
s/g,'').toLowerCase()) !== -1 ;
  });
   setLoadedMovies(title.length > 0 ? title:description);
 }
```

The UML Diagram changed from left to right