INFORMATION HIDING

In order to apply information hiding to our project, we first considered the design decisions that we anticipate changing, and then encapsulated each of them into separate modules. This allowed us to achieve the goal of anticipating changes and defining interfaces that capture the stable aspects of our system and implementations that capture the unstable aspects of our system.

One area where we anticipate a lot of changes is the data that we have on our models. For our models, we used a Restful API to get data into our MongoDB database. As the product keeps developing, we anticipate that we will be adding fields to each instance of each model. The addition of fields to our database means that we will now have to display more information about each model and will have to add filters to account for these new fields.

As we add fields to our MongoDB database, we will have to make changes to instance pages for each model's instances. We would need to do this to display the new information on our webpage. For instance, if we were to add social media links to every artist, we would have to add that to every artist's instance page. Thus, we have encapsulated each instance page in its own module. We have 3 models, meaning we have 3 classes for instance pages: ArtistsPage, SongsPage, and Article (for the News model). This makes adding a new field to the instance page easy, as you go to the module for that model and add it.

We also anticipate having to make changes to the cards for each model as we get more fields. For example, if we wanted to add an artist's Twitter handle to each of their cards, we would have to add that to the splash page. Thus, we have split the model splash pages into different components, so that it is easy to edit different parts. For the Artists splash page, it is made up of an ArtistsForm class and 10 instances of the ArtistsCard class. In order to add to each card, you would simply have to add a field to the ArtistsCard class and it would be populated by the information that the HTTP call to MongoDB makes. The pseudocode for this implementation in the Artists class can be seen below (it is similar for both the Songs and News classes):

```
Class Artists:
        API call to query MongoDB Artists collection based on search & filter parameters
        Return {
                new ArtistsForm(currentParams)
                for(Artist i in API call results):
                        new ArtistsCard(i)
        }
```

As we had mentioned in the previous paragraph, we have also created an ArtistsForm module. If fields are added to each model in MongoDB, then we also anticipate adding fields to filter by. In order to do so, the only place we would have to make a change is the ArtistsForm class, because that controls the form for searching, sorting, and filtering the Artists model. The same would be true for other models as well, albeit with different names.

Lastly, adding fields to the MongoDB database could also mean we have to add endpoints to the back-end of our code, so that the front-end can do operations based on the new field. During earlier phases of implementation itself, we divided our back-end routes up by model. All Artists endpoints go in artistsRoutes.js, Songs endpoints go in songsRoutes.js, and

News endpoints go in newsRoutes.js. Thus, we encapsulate each model's endpoints in its own file. This way, if changes need to be made or an endpoint needs to be added for a certain model, we know we have to go to that model's routes file to make the change.

By dividing each page up into React components, our program is robust to changes and additional features can be added easily. Depending on what change you want to make, you will know exactly what component to modify to add the feature. For example, to add a filter due to a new field being added to the artists model, you would know that you have to modify the ArtistsForm class, because that controls the form that handles filtering for the artists model. As another example, to add social media links to each artist's instance page, you know that you have to modify the ArtistsPage class, because that is what creates the artist instance pages. If you had wanted to add the social media links to the card for each artist, then you would need to edit the ArtistCard class. On the back-end, if we want to add an endpoint for a certain model, we know exactly which file to add it in because our back-end is modularized by model. So, to add an endpoint that updates a song name, we know we have to add it to songsRoutes.js. Our modularization scheme and naming convention makes it easy to add features to both the front-end and the back-end, fulfilling the extensibility capability that we learned about in class.

One disadvantage of our modularization approach is that we could not modularize where we make API calls. That is, we could not create one central location per model where API calls are made. This is because the needs of the model splash page and the model instance pages differ, since the instance pages display more content. Thus, during development, if we need to add an API call to add information, we will have to do so separately in both the model splash and model instance pages. In order to mitigate this, we could embed the call to a new API in an endpoint that is already called, so that we don't have to worry about having to call it in both the model splash and model instance page. For example, the getSongByName endpoint is already called in the SongsPage class (the instance page for Songs). If we want to add a link to the music video for this song from the YouTube API, we could simply add the query to the YouTube API in the getSongByName endpoint in songsRoutes.js, instead of having to put it in the SongsPage class itself. This way, it becomes part of the response object of the API call, so if any other page needs the music video link too, a call to this endpoint suffices, rather than having to make a call to this endpoint as well as the YouTube API endpoint.

DESIGN PATTERN 1: Singleton Design Pattern

We implement the Singleton Design Pattern in our back-end code in order to get rid of duplicated code throughout our back-end routes. Since we implemented our back-end using Node.js and Express.js, we had 3 different JavaScript scripts for each model: artistsRoutes.js, songsRoutes.js, and newsRoutes.js. In each of these files, we had numerous API endpoints. Most of these endpoints accessed the respective MongoDB collection for that model to find certain documents. We copy-pasted the code to connect to MongoDB in each endpoint's implementation. We felt we could improve the design of our code to make it more readable and more efficient if we created a connection to each collection only once. Thus, we decided to move all of this code into a Singleton class that returned a MongoDB collection.

The code in Figure 1 shows a snippet of our code before the refactoring. The section in red shows the code to connect to MongoDB. This code was repeated in all of our functions that access MongoDB across all models. This smells of duplicated code and is also highly inefficient, as connecting to MongoDB is a time-intensive procedure.

```
async function getArtistByName(inputname) {
  var returnedArtist;
  const dbName = "MuseNewsDatabase";
  const collectionName = "artists";
  const MongoClient = require("mongodb").MongoClient;
  const uri =
    "mongodb+srv://musenews:musenew5@musenewsdatabase-cbkjn.gcp.mongodb.net/test?retryWrites=true&w=majority";
  const client = await MongoClient.connect(uri, { useNewUrlParser: true });
  // .then(function(db) {
  console.log("Connected...");
  const collection = client.db(dbName).collection(collectionName);
  returnedArtist = collection.findOne({ name: inputname });
  console.log(returnedArtist);
  console.log("Done looking");
  client.close();
  return returnedArtist;
}
```

Figure 1: MongoDB Accesses Prior to Applying Singleton Design Pattern

This problem fits the general statement of the pattern given in lecture very well. The MongoClient connection is an example of a Thread pool, as MongoDB implements thread pooling internally. The connection to MongoDB is memory and time intensive, meaning we want to minimize the number of times we have to connect. One way to do this is to ensure that a class (or in this case our script) has only one instance of the connection with a global point of access. We want a global point of access so that all functions and endpoints that access MongoDB can use the same connection. Although it may be beneficial to use a global variable, we know that a Singleton gives the advantages of a global variable without the downsides.

In order to achieve this pattern, we followed the classic Singleton approach shown in class, with lazy instantiation. We make it thread-safe by ensuring that we wait for the collection object to be returned before continuing our work, as seen in Figure 2.

```
async function getArtistByRankRanges(start, end) {
  const collection = await ArtistsMongoSingleton.getInstance();
```

Figure 2: Use of Async and Await Keywords in JavaScript to Ensure Thread Safety

Our implementation of the Singleton pattern is slightly different in JavaScript than it would be in Java, as we use the JavaScript Module Design Pattern, as shown in Figure 3.

```
var ArtistsMongoSingleton = (function () {
  var instance;

  async function createInstance() {
    const dbName = "MuseNewsDatabase";
    const collectionName = "artists";
    const MongoClient = require("mongodb").MongoClient;
    const uri =
      "mongodb+srv://musenews:musenew5@musenewsdatabase-cbkjn.gcp.mongodb.net/test?retryWrites=true&w=majority";
    const client = await MongoClient.connect(uri, { useNewUrlParser: true });
    // .then(function(db) {
    console.log("Connected...");
    const collection = client.db(dbName).collection(collectionName);
    return collection;
  }

  return {
    getInstance: function () {
      if (!instance) {
        instance = createInstance();
      }
      return instance;
    },
  };
})();
```

Figure 3: Implementation of Singleton Class in JavaScript Using Module Design Pattern

The implementation is very similar to that in Java- the top line represents the class name with one instance field- instance. The function createInstance() is the private constructor, which creates the MongoDB collection. The return statement creates a getInstance() function that serves the same purpose as the getInstance() function in the Java implementation- to return the unique instance. We then use the Singleton instance in all the functions that need the connection by calling ArtistsMongoSingleton.getInstance(), as was seen in Figure 2.

There are numerous advantages to implementing the Singleton design pattern to only connect to MongoDB once. First of all, it makes the back-end code much cleaner. Second of all, it greatly enhanced the efficiency of our back-end code. Prior to implementing this design pattern, the time to call our queryArtists endpoint (our most-used endpoint) took an average of 879ms. After implementing this pattern, it took an average of 83ms, a vast improvement.

There are also a couple of disadvantages to implementing the Singleton design pattern. Scalability of the site could become an issue- if a lot of people start accessing the site quickly, the few connections to MongoDB could become a bottleneck. This could be addressed in various ways to pool accesses on our back-end code but that was beyond the scope of this exercise, as it was focused on refactoring the code. A class diagram of our implementation of Singleton classes in the back-end is shown in Figure 4.

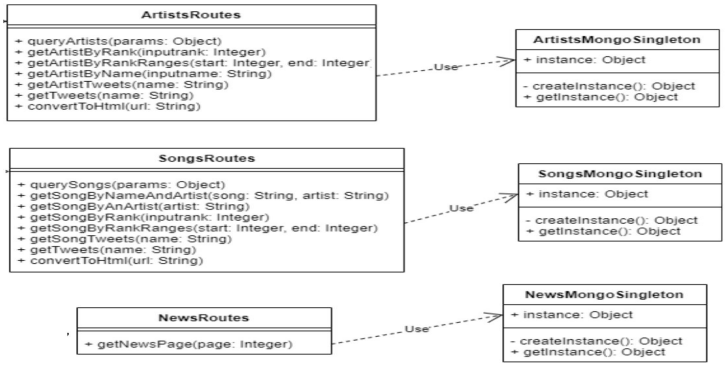

Figure 4: UML Class Diagram of Back-End with Singleton

DESIGN PATTERN 2: Model-View-Controller Pattern

Throughout our development process, we were implementing something similar to the Model-View-Controller (MVC) pattern. We use the MERN stack for our application (MongoDB, Express.js, React.js, and Node.js). This stack closely resembles the MVC pattern, although not exactly. We had to make a few changes in this phase to completely implement this design pattern, but once we did, our code became a lot more organized, meaning our software would be more extensible, maintainable, and reusable.

The MVC pattern divides an application up into 3 parts: model, view, and controller. The view is what the user sees- in this case, it is the cards on each model splash page. The controller is what the user uses to manipulate what they see- in this case, it is the form for searching, sorting, and filtering on each model splash page. The model is the data on the back-end. It is manipulated by the controller and updates the view. In our case, the model is our back-end code that connects with MongoDB. When a user changes the fields in the form and submits it, the URL is updated to reflect these changes and an HTTP call is sent to our back-end code, the model, to query MongoDB. The model then returns the response, which contains a JSON Object of the updated model instances that should be shown on the splash page. The view is updated with the information from the JSON object, and the user now sees the updated cards. Thus, our application fits the general statement of the MVC design pattern.

To implement the MVC pattern, we made a few changes to modularize the components and split the front-end into controller and view sections. For the front-end, we originally just had an Artists class and a Songs class. As can be seen in Figure 5, for the Artists class, we extract the controller into a ArtistsForm component and the view into a ArtistsCard component. The Artists class contains the HTTP call to the API, based on which it creates 10 ArtistCard objects. It also contains an ArtistsForm component. The same architecture was used for the Songs model splash page, where the Songs class contains 10 SongsCard objects and one SongsForm component. Thus, the front-end was refactored so that each model splash page contained a view and controller aspect. When a card is clicked, it redirects to that specific instance's page, which is just an extension of the view since there are no form components on that page.

```
function Artists({ match }) {

  const fetchItems = async () => {
    const data = await fetch("/api/artists/queryArtists/" , {
      method: "GET",
      // mode: "no-cors",
      headers: {
        "Content-Type": "application/json",
        Accept: "application/json",
      },
    });
  };
  }
  return (
    <div>
      <div class="container-fluid">
        <h1 class="pageHeader">America's Top Artists</h1>
        <h2 class="sectionHeader">Top Artists: Page {match.params.page}</h2>
        <ArtistsForm passedInParams={passedInParams}></ArtistsForm>
      </div>
      {items.map((item) => (
        <ArtistsCard item={item}></ArtistsCard>
      ))}
    </div>
  );
```

Figure 5: Implementation of Artists Class Including ArtistsForm and ArtistsCard

The News module front-end is organized in a similar manner. The NewsContainer class is the controller, including the form for searching, sorting, and filtering. The Articles class is the view, as it represents the cards on each page. The NewsGrid class makes all API calls to both Google News API and our server, using the back-end code found in the News Routes class.

We consider the whole back-end server to be our model in this case. Interactions with our model occur through HTTP requests. Since our controller is a form, when the form is submitted, it performs logic to create an HTTP request to an endpoint on the server. This logic occurs in the Artists, Songs, and NewsGrid classes. The server, our model, queries MongoDB and returns the new results in a JSON object. This JSON object is returned as a response to the webpage, and the view component is updated with its contents. Our server contains endpoints for artists, songs, and news. These endpoints provide the operation of our model component, as they interact with MongoDB. The class diagram for our implementation of the MVC pattern is shown in Figure 6, depicting how our classes are divided into view, controller, and model.

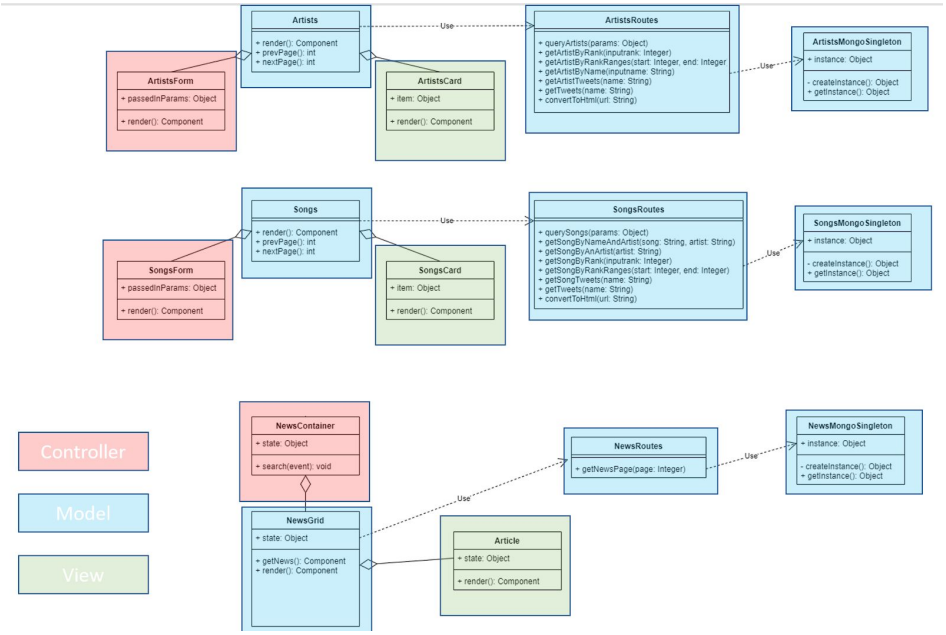

Figure 6: UML Diagram With Annotations for MVC Pattern

Using the MVC pattern is advantageous in the design of our web application because it provides a uniform design that everyone follows. Throughout the development process, we had an idea of how our front-end and back-end would interact, as we knew exactly what components do what. Going forward, this will make it easier for us to add new software such as new data models (extensibility) and make changes to the existing software (maintainability) because each aspect of the pattern is independent.

There are few disadvantages to the MVC pattern. One disadvantage is that separating the front-end into controller and view means that the elements in the view cannot manipulate the model. If we want to implement a feature where we click on an instance's card to search for similar instances, that may be hard to implement without turning the view into a controller itself.

REFACTORING

REFACTOR No. 1: Extract class to split up Artists class into Artists, ArtistsForm, ArtistsCard classes and Songs class into Songs, SongsForm, SongsCard classes. (Used 2 times)

One smell that we noticed was that our implementation of the Artists and Songs model pages were just one file that was unnecessarily long and could be split up into multiple modules that each had their own responsibility. As our development process had evolved, we had to add more HTML and React elements to our webpage, meaning the complexity of our model splash pages grew. We identified this as a Large Class smell, and decided that the best way to refactor this was to split up the responsibilities into different classes. For instance, our Songs.js file, which is the splash page for the songs model, was 364 lines prior to refactoring and is now 207 lines. The best way to make the code prettier was to use the Extract class refactoring technique and split up the responsibilities of the class, specifically the HTML. This would also further the Model-View-Controller pattern by separating the page into controller and view components and increase information hiding by decreasing the number of responsibilities in the Artists and Songs classes.

We use the Artists class as an example of our refactor, because the changes to the Songs class were made in parallel. The two responsibilities we decided to extract to a new class were the form React component and the card React component. The form React component consists of all searching, sorting, and filtering inputs that the user can enter. This component was extracted to the ArtistsForm class. The card React component represents one card for an artist that displays their name, picture, and ranking. There are 10 such cards on the model splash page. Each card serves as a link to that artist's instance page. This component was extracted to the ArtistsCard class. The final resultant UML Class Diagram can be seen in Figure 7.

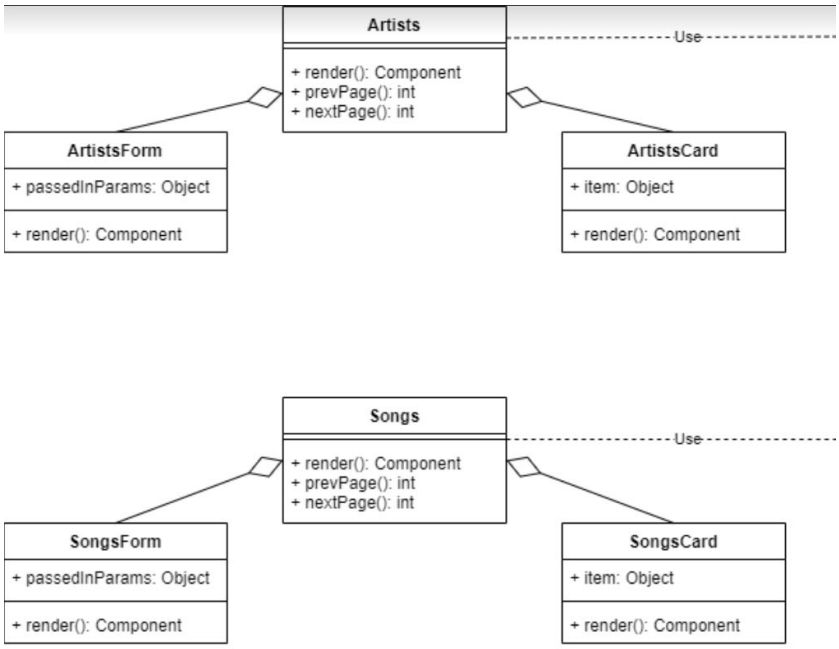

Figure 7: Selection from UML Class Diagram Showing Refactor of Artists and Songs

There are many advantages to implementing this refactor. For starters, it makes our code much more readable, as the Artists and Songs classes are much shorter. It also encapsulates responsibilities better, because now, changes to the form will occur in ArtistsForm instead of Artists, and changes to the cards will occur in ArtistsCard. This refactor also helps us commit to the Model-View-Controller design pattern, as described in the Design Patterns section of the report. There are no significant downsides to this refactor.

REFACTOR No. 2: Preserve whole object on queryArtists() and querySongs() methods in artistsRoutes.js and songsRoutes.js (Used 2 times)

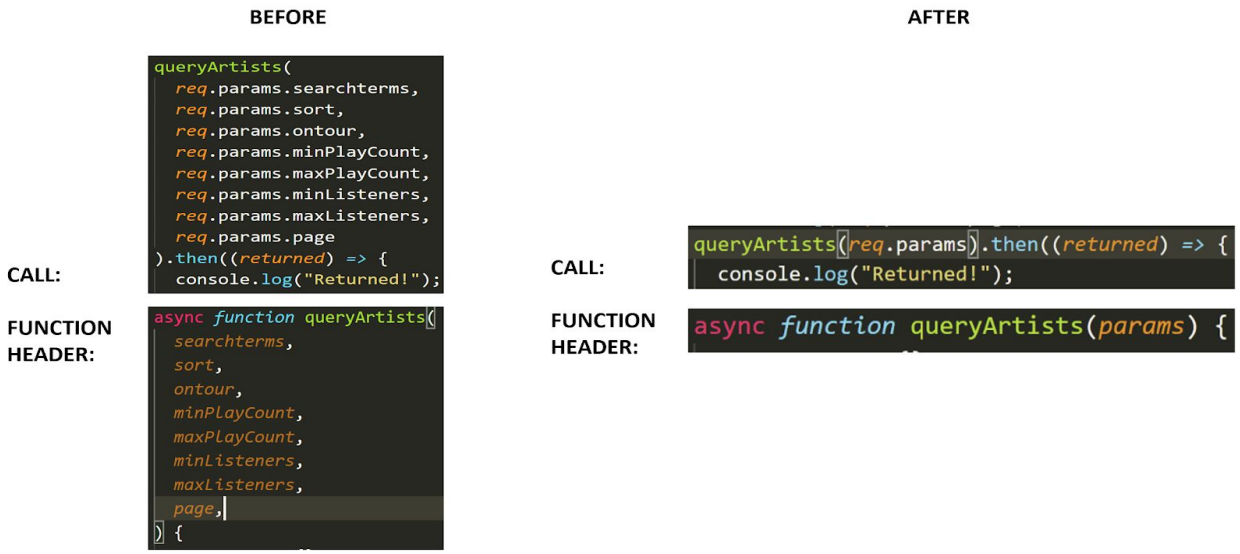

Figure 8: Before and After Code Snippets of queryArtists Function

Another smell that we noticed was that the main back-end query we used to search, sort, and filter artists and songs, queryArtists() in artistsRoutes.js and querySongs() in songsRoutes.js, called a function that had a long parameter list. This issue can be seen in the “BEFORE” section of the code snippet in Figure 8. We were passing in specific fields of the HTTP request parameter object, but it made more sense to just pass in the whole HTTP request parameter (req.params) and let the function access the specific fields it needs to. It was easy to see that we should implement the Preserve whole object refactoring technique to fix this issue. The easiest way to do this was to pass in the whole request parameter object, which is what we did. In the queryArtists() and querySongs() functions, we replaced all calls to the fields (e.g. searchterms, sort) with calls to the fields in the parameter object (e.g. params.searchterms, params.sort). This can be seen in the “AFTER” section of Figure 8. The long list of parameters is replaced instead with a single parameter, params. This made the code much easier to read, as we didn’t have the parameters spanning multiple lines. This was very advantageous because now, if parameters were added, we would not have to change the function signature or the call- we could just add code in the queryArtists() or querySongs() function to handle the new

parameter. There aren't many downsides to this refactor, because it increases readability and flexibility, enhancing our software's extensibility, maintainability, and reusability.

REFACTOR No. 3: Extract method on RedirectPages and RedirectSongsPages render() methods to take out code creating URL to redirect to (Used 2 times)

BEFORE

```
render() {
  console.log(this.props.match.params.sort);
  const values = queryString.parse(this.props.location.search);
  console.log(values.search);
  console.log(values.search === "");
  console.log(values.source);
  var urlString = "/" + values.source;
  if (values.search === "") {
    console.log("search null");
    urlString += "/splash/none";
  } else {
    var encodedSearch = encodeURIComponent(encodeURIComponent(values.search));
    console.log("encoded url is" + encodedSearch);
    urlString += "/search/" + encodedSearch;
  }

  if (values.sort === "rank") {
    urlString += "/rank";
  } else if (values.sort === "nameAsc") {
    urlString += "/nameAsc";
  } else if (values.sort === "nameDesc") {
    urlString += "/nameDesc";
  } else {
    console.log("Illegal sort");
    urlString += "/rank";
  }
...
```

AFTER

```
render() {
  console.log(this.props.match.params.sort);
  const values = queryString.parse(this.props.location.search);
  console.log(values.search);
  console.log(values.search === "");
  console.log(values.source);

  var urlString = createURLFromProps(values);

  return (
    <div>
      <h1>Redirection</h1> <Redirect to={urlString} />{" "}
    </div>
  );
}
```

Figure 9: render() Method Before and After Refactoring

Another smell we noticed was that in RedirectArtiststPages and RedirectSongsPages, we had a very long render method. These two classes are what the forms on the Songs and Artists splash pages redirect to whenever a search filter is updated and the form is submitted. These classes then process all of the data in the forms and redirect back to the respective splash page, but with updated parameters. The render() method serves to create the Redirect React element, which redirects back to the splash page. We had a lot of lines of code to process all of the form inputs and handle edge cases prior to redirection. This can be seen in the "BEFORE" section of Figure 9. We display two out of many if statements that handle the different cases for each form input. With all of the fields that we had to parse, this made the render() function unnecessarily long, exhibiting a Long Method smell. We decided it would be best to split the responsibilities of the render() function so that the render() function just took care of rendering the Redirect element. This means that we had to move the code for creating the URL to a separate method that was just responsible for that action. This problem prompted us to use the Extract method refactoring technique. We moved the URL-creation code to a separate method (createURLFromProps()) and then called that in the render() method. The benefit of this was that it cleaned up our code and made it more readable. This refactor also enhances our program's extensibility, maintainability, and reusability because we now know we only have to edit createURLFromProps() to handle the edge cases associated with processing the form input. There aren't any downsides to this refactor, as it only helps our program look cleaner.