Project Report : All About Austin
Phase 4 - Team Amber

Team Members

Name, E-mail, Github User
Justin DuPont, justindpnt@gmail.com, justindpnt
Canyon Evenson, canyon@utexas.edu, cpe342
John Koelling, john.k.koelling@gmail.com, lucundus
Yixing Wang, yixing.wang@utexas.edu, AlienEdith
Grayson Watkins, graysonwatkins@gmail.com, Graysless
Zach Wempe, zdwempe@gmail.com, zachwempe

Project Leader, Phase 4

Grayson Watkins

URL to Website, Git Repository, Google Docs

Deployed Website:
http://www.allaboutaustin.info/
GitHub:
https://github.com/Iucundus/AustinData
Google Docs:
https://docs.google.com/document/d/1R-suefPzFPDNJkwbQ2wc0wpQ5aZYFyTxBlKfVDbHA-4

Part IA: Information Hiding

- We have maximized cohesion between our front end and back end systems in addition to minimizing coupling between the raw data and presentation of our calculated averages. Our RESTful API acts as a handshake between the raw data from our scraping, and presents the Front End with information to present to the user. The “secret” in this case is the raw data points such as restaurant scores, but the information hiding lies in that the calculation of the finalized average and ranking of the zip codes based on user metrics is completely detached from the presentation itself. This setup reduces coupling in the sense that the front end is able to make calls to our API without worrying about how the information from the backend is being calculated. In a hypothetical example, if we were to rewrite our entire methodology for computing traffic data, the front end does not care about the updated implementation, because the API call for getTrafficScore() will remain the same. This level of abstraction and information hiding is essential not only for longevity of our code, but also provides a means for us to diversify our datasets in the future.

- On the individual zip codes page, we have a section for the respective food, traffic and education categories that present more detailed information to the user. Although this feature may seem to be in contrast with the information hiding previously discussed, decreased coupling with maximum cohesion is maintained through API calls. Although the Front End is not entirely blind to the actual data in terms of restaurants, schools, and speeds, these filtered categories provide a high level view into our average computation methodology. If we were to completely refactor our system for obtaining food scores, we would need to modify our API call to display the relevant data, but we ultimately decided that this slight degree of coupling proved essential to establish credibility of our data calculations.

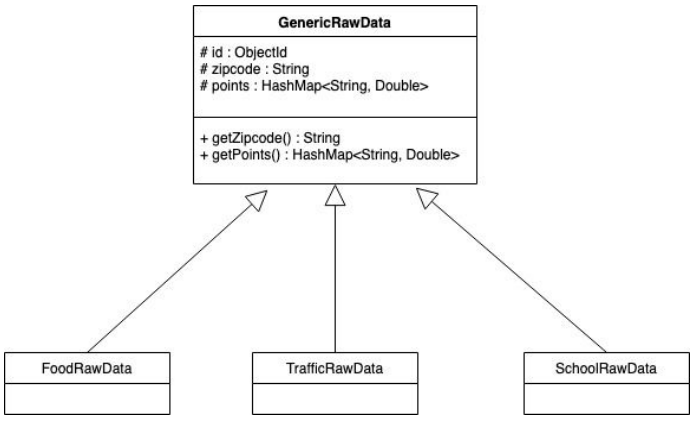
- We encapsulated separate UI components into different React Components, each component only need to access specific small piece of data, for example, the *HeatMap* component just need an geojson file as an input. This feature supports us to use reusable components across different pages, for example, *ZipcodeCard* and *HeatMap* component are used in different category (food, traffic, education) page. Also, Each page is made up by combination of components, which makes it easier for us to add or delete components.

Part IB: Design Patterns and Refactorings

Design Patterns

Backend raw data: Template Design Pattern

The GenericRawData class defines the functionality of all raw data objects. Subclasses (FoodRawData, SchoolRawData, TrafficRawData) override the functions to store and retrieve the correct information. Other classes can assume the template type of GenericRawData and use any subclass. Each of the data types are very similar in nature, and so it is advantageous to have them corresponding to each other in some way. Using the template design pattern, we have a abstract pattern to follow with the implementation of each raw data type, but we can customize the specifics of each implementation with each data type.

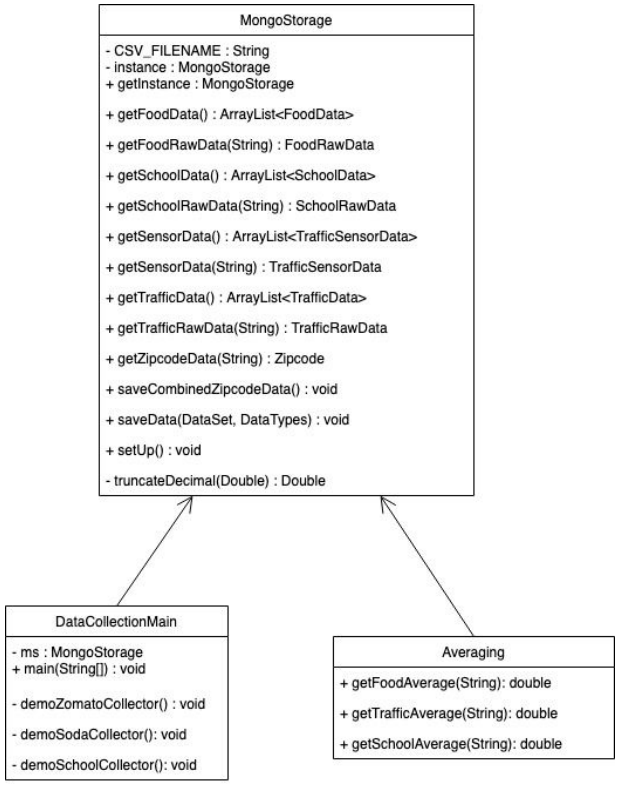


Code snippets:

```
public class GenericRawData {
...
    public GenericRawData() {};
    public GenericRawData(String zipcode, HashMap<String, Double>
points)...
    public String getZipcode()...
    public HashMap<String, Double> getPoints()...
}
...
public class TrafficRawData extends GenericRawData {
    public TrafficRawData()...
    public TrafficRawData(String str, HashMap<String, Double> props)...
}
```

MongoStorage: Singleton Design Pattern

The MongoStorage class will be refactored into a singleton. Whenever the MongoDB database needs to be accessed, the class will be accessed with MongoStorage.getInstance() which instantiates the MongoStorage singleton if it is not already prepared. This is efficient because the connection setup for MongoDB only needs to be done once, and it should not have to happen multiple times.



Code Snippets:

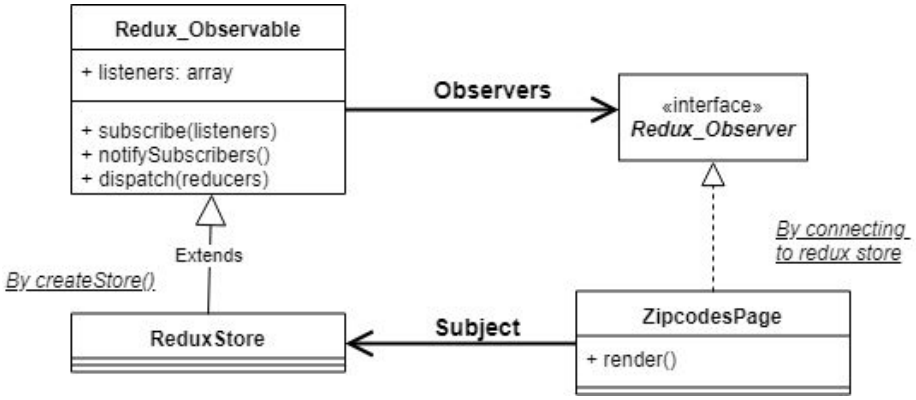
```
public class MongoStorage {
    static MongoStorage instance = null;
...
    public MongoStorage() {
       // Connect to Mongo DB
       MongoClientURI uri = new
MongoClientURI("mongodb://aaa:allaboutaustin...");
...
    public static MongoStorage getInstance() {
       if (instance == null) {
             synchronized (MongoStorage.class) {
                   if (instance == null) {
                         instance = new MongoStorage();
                   }
             }
       }
       return instance;
    }
...
}
```

```
public class DataCollectionMain {
    private static MongoStorage ms;
    public static void main(String[] args) throws IOException {
       ms = MongoStorage.getInstance();
...
       ms.saveCombinedZipcodeData();
    }
...
    private static void demoSchoolCollector() {
...
       DataSet ds = demo.getNewData();
       ds.printDataSet();
       ms.saveData(ds, MongoStorage.DataTypes.EDUCATION_RAW_DATA);
...
    }
...
}
```

Observer Pattern with Redux

For Front End(React), we used a React bindings package for Redux (react-redux). The Redux is an implementation of Observer Pattern. Using Observer Pattern with Redux benefits us with rendering our UI components dynamically and managing global state. However, the implementation is less straightforward and required more code.

1. The basic idea of observer pattern applied in Redux:
   - **State**: The Data that changed by Reducers (changes are triggered by calling actions).
   - **Subject/Observable**: A Redux Store created by calling the function *createStore()*, which take reducers as arguments. The Redux Store holds a list of listeners/subscribes of components. When an action is called, Redux Store will dispatch it to all the reducers and corresponding state will be changed. Once the state is changed, Redux Store will notify all the subscribers (similar to *notifyObserver()* in general observer pattern)
   - **Observers**: All the components that are connected to the Redux Store. If we connect the component to redux store, the component will subscribe to the Redux Store when it's mounted (similar to *registerObserver()* in general observer pattern). The component could call the actions to trigger the state changing and then trigger the re-render of itself or other components (similar to *update()* in general observer pattern). Since the components are basically UI components, rerender means the the update of UIs here.
2. UML diagram: We use ZipcodesPage as an example of the observer components



3. Code Snippets: Most of the observer and subject functions are already implemented in the package. Thus, we just show the customized part here.

```
const store = createStore(
    // reducers controll state data
    reducers
);

// Component: Observer
class ZipcodesPage extends Component{
    componentDidMount(){
        // subscribe to redux store here, automatically done by connect() function
    }

    // connect() will register the component to the redux store
    export default connect(mapStateToProps, {
        GetAllZipcodes, GetFilteredZipcodes // actions
    })(ZipcodesPage)
}
```

Extract Class Refactoring (Front end tests):

- Prior to refactoring, our front end test functioned as just a simple script that ran through all of the tests. We realized that it would be more expandable and modifiable if, rather than keeping this as a script, we broke it out into an object-oriented design with a TestRunner object, which would allow us to extend and expand the functionality of our testing suite for any potential future development. The code itself did not change much here, rather just the structure of the file. What were once simple functions became class methods, and rather than passing a driver object from function to function we simply had a self.driver field for the TestRunner object.
- Here is an example of a method before the refactor:

```
def verifyHomePageButton(driver):
    '''
    This method tests that the homepage button in the top left corner
    is a link to the homepage
    '''

    btn = driver.find_element_by_partial_link_text('AAA')
    btn.click()
    url = driver.current_url
    assert url == 'http://allaboutaustin.info/' , 'HomePage URL incorrect!'
```

- And here is an example of a method following the refactor:

```
def verifyHomePageButton(self):
    '''
    This method tests that the homepage button in the top left corner
    is a link to the homepage
    '''

    btn = self.driver.find_element_by_partial_link_text('AAA')
    btn.click()
    url = self.driver.current_url
    assert url == 'http://allaboutaustin.info/' , 'HomePage URL incorrect!'
```

- Visually, it is not a particularly significant refactor, but the transition to an object-oriented design presents a significant upgrade in terms of modularity, reusability, and maintainability.

- **Rename Refactoring**
    - React for a frontend framework was amazing to work with, but the heavy amount of modularization also led us to somewhat over modularize components of the front end. The functionality was superb for this model, but the understandability somewhat suffered as a result. In order to ensure the longevity of the project, we heavily utilized Rename Refactoring. In order to get a fresh set of eyes on the problem, the front end team employed the back end team to help with renaming. After giving them a short amount of time to examine file and component names, we asked them to describe the high level functionality of the code snippets. If the code review took longer than a few minutes, we marked down the file and focused on clarifying functionality. The following are our refactorings:

        - SurveyModal → Survey Popup
        - Stack → Development Stack
        - ZipcodesMap → AllZipcodesMap
        - CategorySwtich → ChangeCategory
        - RankingCard → ZipcodeRankingCard
        - ZipCodeComponent → ZipCodeCardComponent
        - MoreFilters → AdvancedFilters
        - Pagination → ZipCodePageChangeButton
        - CarouselComponent → PictureCarouselComponent

Extract Class Refactoring (Front End Components)

- Upon evaluation of our HeatMap component, we realized that it reeked of the large class and duplicate code smells. In order to combat these problems, we sought to further modularize our heatmap implementation by separating the big chunks of code into react components. By creating the NonHoveredZipCode and MapLegend components, we were not only able to clean up our implementation of our heatmap component but also increased readability of the HeatMap.js file upon first glance.

Heatmap.js before refactoring

```
else{
  return(
        <div className="row-bl-2">
            <ul className="list-group list-group-flush">
              <li className="list-group-item active"><strong>Zipcode:</strong> </li>
              <li className="list-group-item"><strong>Food Score:</strong> </li>
              <li className="list-group-item"><strong>Traffic Score:</strong></li>
              <li className="list-group-item"><strong>Education Score:</strong></li>
            </ul>

            <h4 className="mt-2">Map Legend</h4>
            <div className="row">
            <div className="col-5">
            <ul className="list-unstyled">

              <li><img src="https://i.ibb.co/WHH8fHc/3288bd.png" alt=""/> 10%</li>
              <li><img src="https://i.imgur.com/lSMtmKA.png" alt=""/> 20%</li>
              <li><img src="https://i.imgur.com/xx72TDD.png" alt=""/> 30% </li>
              <li><img src="https://i.imgur.com/sIlIoNX.png" alt=""/> 40% </li>
              <li><img src="https://i.imgur.com/Wrac0Mc.png" alt=""/> 50% </li>

            </ul>
            </div>
            <div className="col-5">



            <ul className="list-unstyled">
              <li><img src="https://i.imgur.com/LpUYGQO.png" alt=""/> 60% </li>
              <li><img src="https://i.imgur.com/pY6qMwT.png" alt=""/> 70% </li>
              <li><img src="https://i.ibb.co/Sm5c0mG/col9.png" alt=""/> 80% </li>
              <li><img src="https://i.imgur.com/uFyOFkC.png" alt=""/> 90%</li>
              <li><img src="https://i.ibb.co/txKQTK1/col11.png" alt=""/> 100%</li>
            </ul>
            </div>
            </div>

      </div>
```

Same Section After Refactoring

```
else{
  return(
      <div>
          <NonHoveredZipCode/>
          <MapLegend/>


      </div>
  )
}
```

NonHoveredZipCode.js and MapLegend.js reflect the refactored components