Whatsloo

Private Github repository link: https://github.com/646-group8/mobile-app Branch: for-merge-tmp

Mengyao Zhang m473zhan@uwaterloo.ca 20912885
Zishuo Xu z463xu@uwaterloo.ca 20900288
Yitao Hou y39hou@uwaterloo.ca 20615100
Lu Wang l647wang@uwaterloo.ca 20896426
Yaowen Mei y5mei@uwaterloo.ca 20470193
Ting Gu t39gu@uwaterloo.ca 20924386

1 Architectural Style

1.1 Functional Properties

Our system supports the following functional requirements as follows:

1) **Overview pattern.** Users can click "Overview Pattern" to enter the corresponding functionality. Our app displays an overview pattern of Google map of University of Waterloo campus. On the map, the campus is outlined, buildings, benches, lakes and other meaningful spots are marked. Users can zoom in, zoom out and move the map to discover more interesting spots on campus. By long click of the map, user can change the map type to normal/satellite/hybrid.

2) **Following pattern.** Our app supports online following the pattern of Google map. This pattern utilizes the GPS function of mobile devices to do more things than the overview pattern. First, there are some Easter Eggs spots which can only be shown when the user is less than 20 meters away from them. Secondly, the camera view of the map changes as the user moves. Finally, the user's location is displayed as a blue dot. By clicking it, the user can upload a new spot in this position. This pattern is useful when the user is located on campus since the user can walk and visit some spots that attract him/her.

3) **Read stories.** The name and an info shadow are shown when users click the marked spot. When users click the shadow, the page jumps to the list of stories related to the spot. Each item in the list shows the title of the story, and users can click on whatever they want to view. In the story's detail page, it shows moving or interesting content with vivid pictures.

4) **Data contribution.** Users can click "Upload a New Story" in the main interface to upload their own stories and views related to spots. Before uploading, the app will check and require three permissions from users: the permission to obtain the user's real-time GPS information, the permission to read a picture from the album and extract GPS information from the picture, and the permission to take a picture via the system's camera app. If the user decides to choose a picture with GPS information attached from the album, our app can extract the longitude and latitude and display the picture; whereas if the picture chosen has no GPS information, the app prompts the user to manually input the longitude and latitude. If the user decides to take a picture, the app can display the user's current GPS information in a related bar. After the user has input all required information such as the name of the place, descriptions or stories, the user can click the "Submit" button to share it with others.

1.2 Non-Functional Properties

1) **Efficiency**. Our app responds quickly. The time our app takes for a picture to be taken through the camera or chosen from the album and then uploaded will be under 5 seconds. The time it takes for submitting new stories after clicking the submit button will be under 2 seconds. The time it takes for showing marked spots on the map after entering overview and following pattern will be under 2 seconds. The time it takes for displaying the story's list of one spot after clicking will be under 2 seconds. The time it takes for displaying the detailed story after clicking will be under 2 seconds. Different pages or activities take less than 2 seconds to jump.

2) **Readability**. The system is easy to understand and the project is comprehensible to new developers. The architecture of the project is clear and the code is readable. Our app has 6 activities and each activity is responsible for single functionality. Each method contains no more than 100 lines of code and in most cases, each method has less than 50 lines of code. Each class has no more than 20 methods. In addition, developers name activities, methods and variables sensibly.

3) **Usability**. Our app is usable. A user can upload pictures within 2 clicks (one click for choosing the picture from the camera or the album and another click for taking the picture or clicking the picture in the album). A user can submit a new story within 5 clicks. On the map, a user can view the story within 3 clicks. Related information is kept together, users can scroll the whole story list within 2 times.

4) **Safety**. When users open the app, the home page will prompt 2 security reminders: “Please pay attention to safety when walking on campus using the APP” and “Please keep a safe social distance”. It enables users to gain more insight into the history and anecdotes of UW with a strong sense of participation and immersion. Users will have a chance to discover the campus in depth and experience the joy of “treasure hunting”.

1.3 Repository

The Fig. 1 shows the repository architecture. In our system, we store basic spots information such as building name, description, image urls, position information (latitude and longitude) and stories in the Firebase Realtime Database. In addition, images are stored in the Firebase Storage. So, Firebase acts as a repository that stores data long term and let other components access the data.

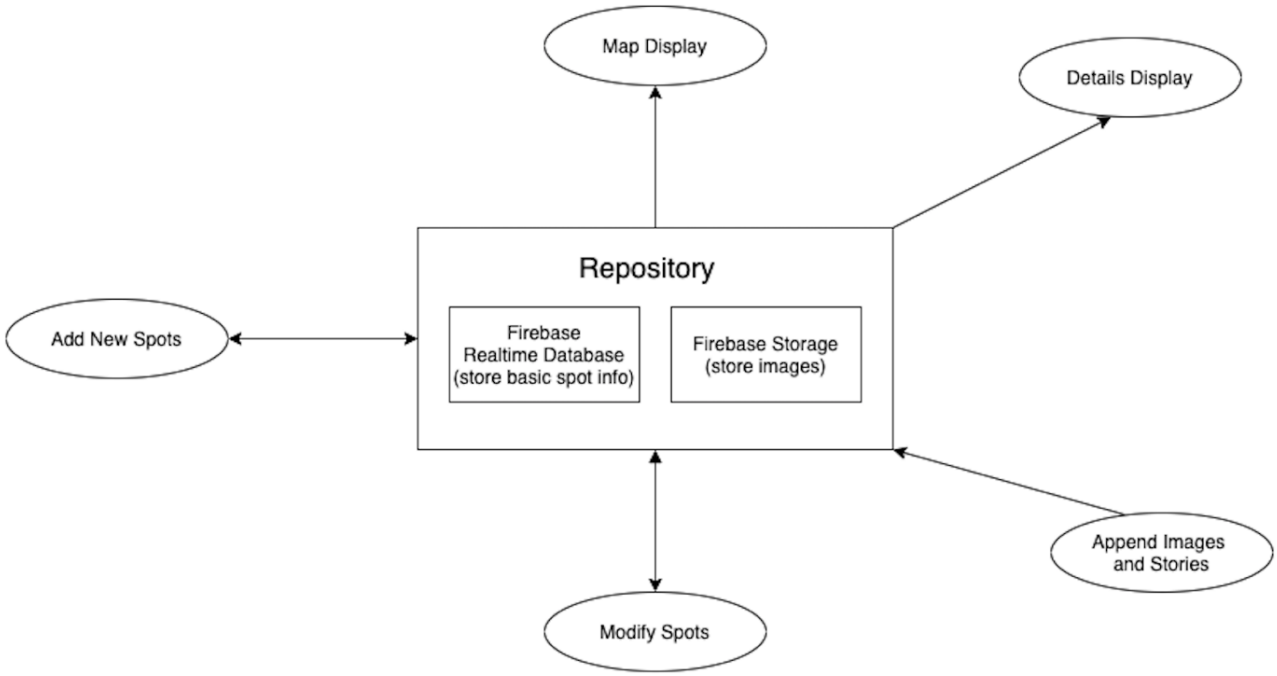

Fig. 1 Repository

As you can see in Fig. 2, it shows that the component diagram of our system. The repository architecture supports readability because other components can easily access data from repository (Firebase Realtime Database and Firebase Storage) via APIs which are located in DataOperation.java. For Map Display, “MapsActivity” and “MapsFollowActivity” use the method “readInfoFromFirebase” to retrieve basic spots information that stores in the Firebase Realtime Database. For Spot Details, “DetailsActivity” and “SpotActivity” retrieves the details of one spot from Firebase easily. For upload and update new spots, “UploadNewPlaceActivity” uploads and updates spots information in the Firebase. The structure and component are very clear and code is easy to understand.

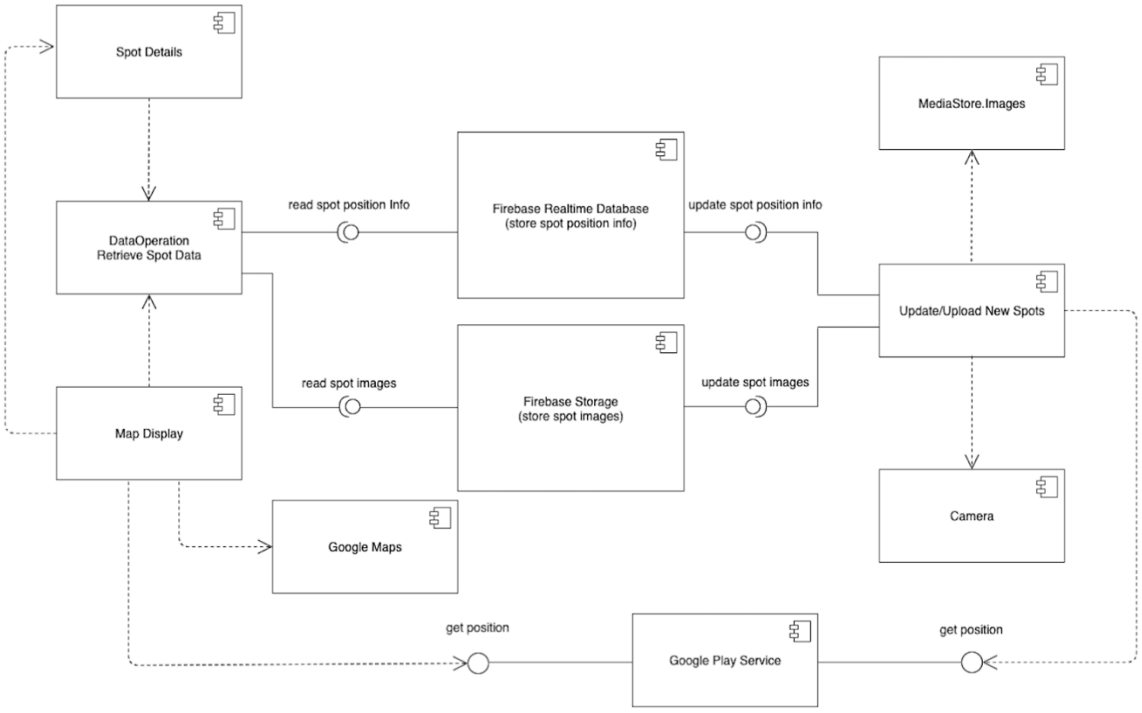

Fig. 2 Component Diagram

1.4 Pipe and Filter

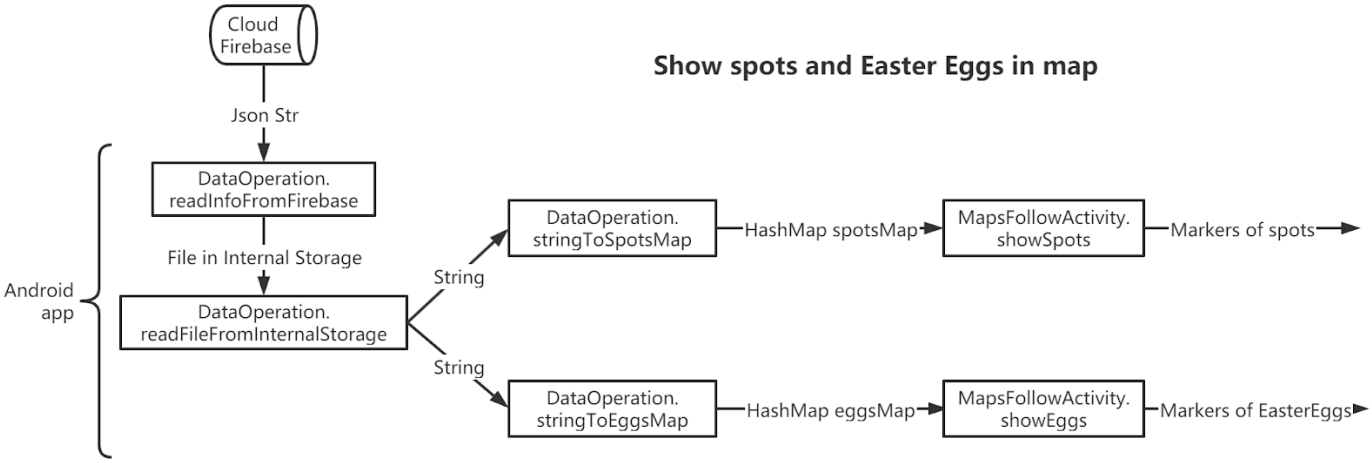

Fig. 3 Pipe and Filter

The Fig. 3 is an example of Pipe and Filter used in our app. This is about the process of showing spots and Easter eggs in following mode, which is one of the most significant function in the app. The final goal is to show the markers of normal spots and special Easter eggs spots. In this case, different phases of data processing are pipelined.

At the beginning, the data is stored in Cloud Firebase. The method “readInfoFromFirebase” in “DataOperation.java” get Json Str from cloud and output file in internal storage. Then the method “readFileFromInternalStorage” takes the file as input and produces Strings as output. The next two components are to convert these Strings correspondingly to HashMaps we want. Finally, in “MapsFollowActivity”, these two methods take the corresponding Hashmap as input, and add markers on the map to show the result. Thus, all spots are displayed properly in the map. You can verify these methods in “DataOperation.Java” and “MapsFollowActivity.java”.

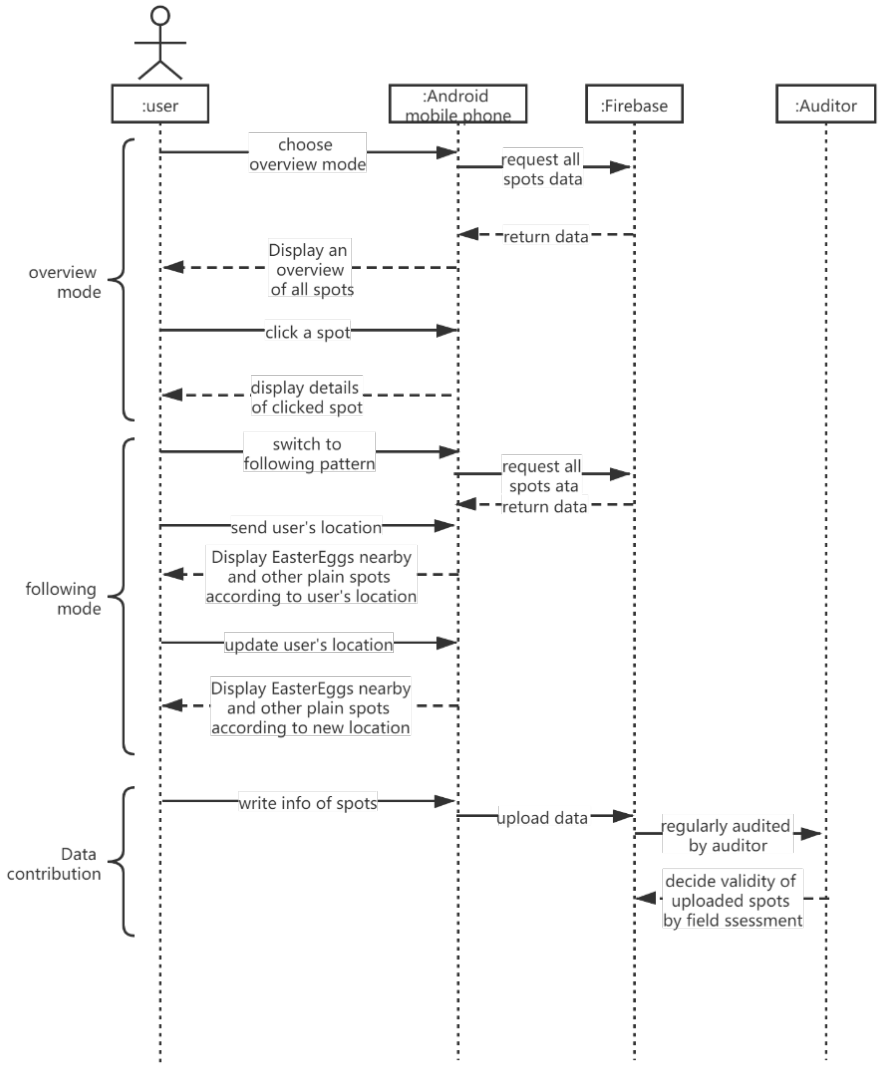

Fig. 4 Sequence Diagram

The sequence diagram shows the three scenarios in the initial proposal, which is also the three main functions.

1) Overview mode
In this case, user choose the overview mode, the Android mobile phone receives the choice and request all spot data from Firebase, which is a cloud database. Then the database return requested data to app, and the app display an overview of all spots. Furthermore, if the user is interested in certain spot, he/she can click it to get more info, the app will jump to a new display interface to show more details of clicked spot.

2) Following mode

When use switch to following mode, the app requests all spot data from Firebase again to get the latest data. The firebase also returns these data again. Besides, in this mode, user's location is needed to show easter eggs nearby, so user also sends the location to app. Then the app display easter eggs nearby and other plain spots according to the location. Each time the location is changed, user will update it, and app will also update the information shown in the interface.

3) Data Contribution

As the initial data is collected by our developer team, they are inevitably inadequate. However, after a period living on campus, everyone has his/her own experiences. This scenario is designed for user to upload his/her own experience. User can write info of certain spots they want to create or update in app and choose to upload. The app will upload data to Firebase. To avoid false or offensive content from appearing in the app, all these data are regularly audited by us, we decide their validities by field assessment.

2 System Design

2.1 Class Diagram

The class diagram of our app is shown in Fig 5, including all key classes and their public methods. First, there is a "MainActivity" when the user opens our app, he/she will come to this activity first, and from here, he/she will be able to navigate to all other activities by choosing from the sidebar. The user may jump to four possible activities (four user interfaces) from the "MainActivity", namely to be: the "MapsActivity", the "MapsFollowActivity", the "UploadNewPlaceActiviry", and the "InstructionActivity". These 4 activities are correspondingly designed for the function of overview mode, following mode, and data contribution, showing how to use this app to users. Furtherly, when a user clicks a certain spot on the map, the app will display more details about this spot to the user, these functionalities are implemented with the "SpotActivity" and "DetailActivity". All the information of spots is loaded from cloud Firebase by "DataOperation". The "DataUploadAdapter" is used in the process of uploading new place's data. Note that this is only a simplified diagram of all classes, and the actual internal implementation is more complicated.

The justification of why we choose to design our app in this way is given by the features of our app. The basic function of our app is a map showing data for different interesting spots inside of campus. Based on this idea, we need a cloud database and make data interaction (CRUD operations) to be available to our app users, so the Firebase and its reading/writing properties are our best choice. Considering all these requirements above, we have designed the app in this way.

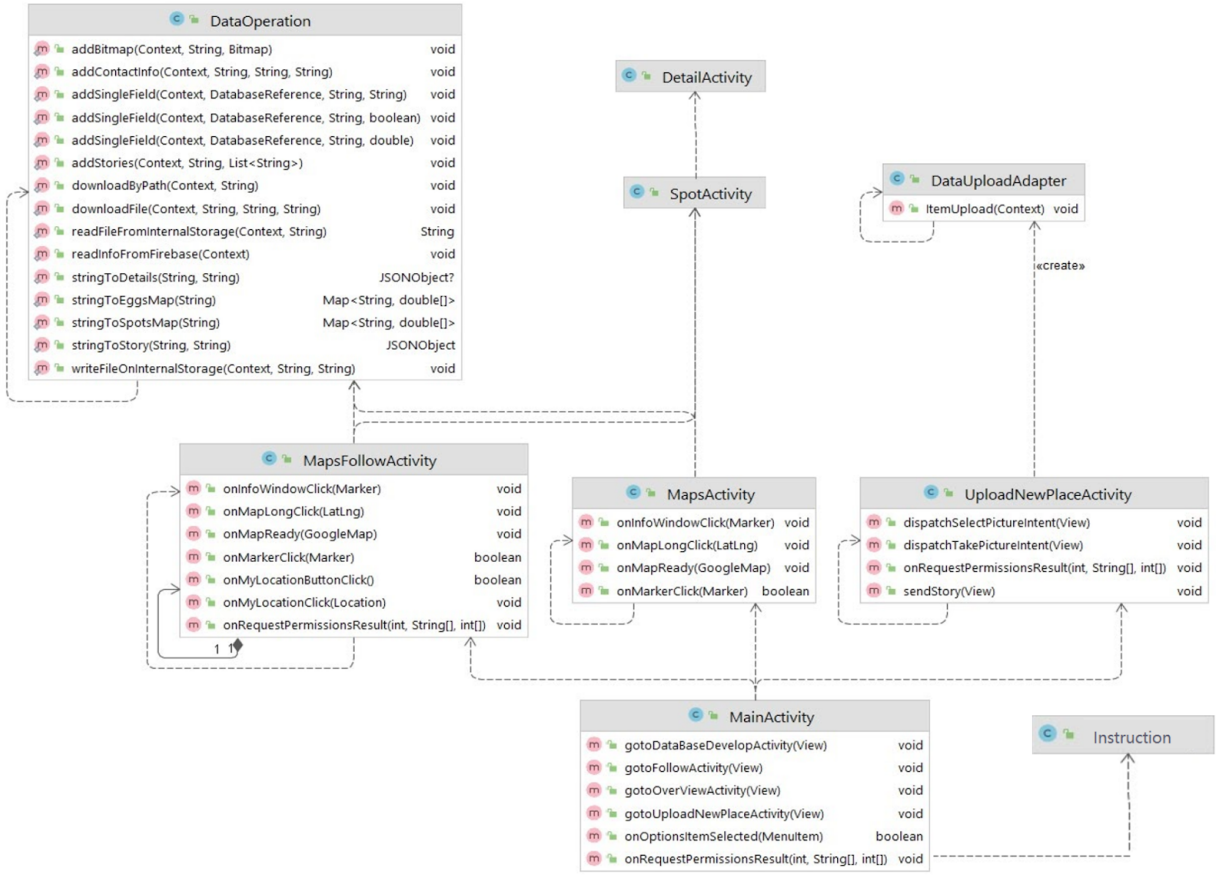

Fig. 5 The Class Diagram

Mapping between design level and components described in architecture:

MapsActivity&MapsFollowActivity-->Map display, Google maps, Google Play Services,

SpotActiviry&DetailActivity-->Spot Details

UploadNewPlaceActivity-->Upload New Spots, Camera, MediaStore, GooglePlayServices

DataOperation&DataUploadAdapter-->Firebase Realtime Database, Firebase Storage

2.2 How Our Design Minimizes Coupling

The whole system can be decomposed into three main subsystems: map module, data module, and item-submission module.

1) The map module is responsible for displaying items and manipulating position information. The data module provides data abstraction and utilities for data upload/retrieval operations.

2) The item-submission module takes charge of the submission process, through which users upload new spots and their stories.

3) The data module acts as an adapter that helps the map module to read data from the cloud database, and helps the item-submission module upload data to the cloud database

On the one hand, this decomposition improves cohesion because each subsystem focuses on a group of logically closely related behaviors. On the other hand, the subsystems hide the implementation details from other subsystems and communicate with each other by invoking public APIs, which reduces coupling and makes it possible for the subsystems to evolve relatively dependently according to their own requirements changes.

Only data coupling is involved in the system. There is no control coupling, such as passing a control flag or returning error codes. Besides, there is no stamp coupling; namely, no redundant data is passed to a module, which avoids unnecessary dependencies and prevents possible security risks.

Notably, methods in the DataOperation class require context information because it is needed in the toast message prompted to use when the data operations fail. Therefore, there is no stamp coupling. We have also strived to eliminate code smells. For example, in order to avoid the data clump problem, we encapsulated items that users upload as the Item class and use Item objects in methods rather than separate fields, which improves the readability of the code. Moreover, during the development process, we found code duplication on adding a single field to the database, so we factored it out to a helper method, addSingleField() in our DataOperation class, which is beneficial for bug fixing and code maintenance.

2.3 How Our Design Could Accommodate Future Requirement Change

As mentioned above, low coupling of our design enables possible future changes described below:

1) Change of Database. We might choose to use a different database because it ensures better throughput, latency, scalability, availability, etc. Ideally, the data module keeps its public APIs as before and only changes the implementations by invoking different external database services. The other two modules could remain unchanged.

2) Upload Videos. Users may request to upload videos for spots because it is more interesting and serves as a time capsule. In this case, we need to create new APIs in the data module to upload and download the videos to/from the storage. Other two modules only need to invoke these APIs.

3) Add Search Function. Future versions of this application may include the search function: users type in the query, and the app returns spots with properties related to the queries. For example, if a freshman is eager to find out where geese might appear and types in "geese," the app will return spots whose name, description, or stories includes the word "geese." In this case, the data module needs to maintain an inverted index and update it every time a new item or story is uploaded. Besides, it needs to implement a retrieval method to generate query results, for instance, boolean retrieval or BM25 ranking algorithm. And it also needs to provide a public query interface for the map module. The map module only needs to add the layout for the query and invoke the API.

4) Change of Display Style. It is more user-friendly if the appearances of item display and item upload interface change to different styles on festivals to create a festive atmosphere. To achieve this, we only need to change the related layout of the map module and the item-submission module. The data module could remain unchanged.

2.4 Design Pattern

1) Decorator Design Pattern

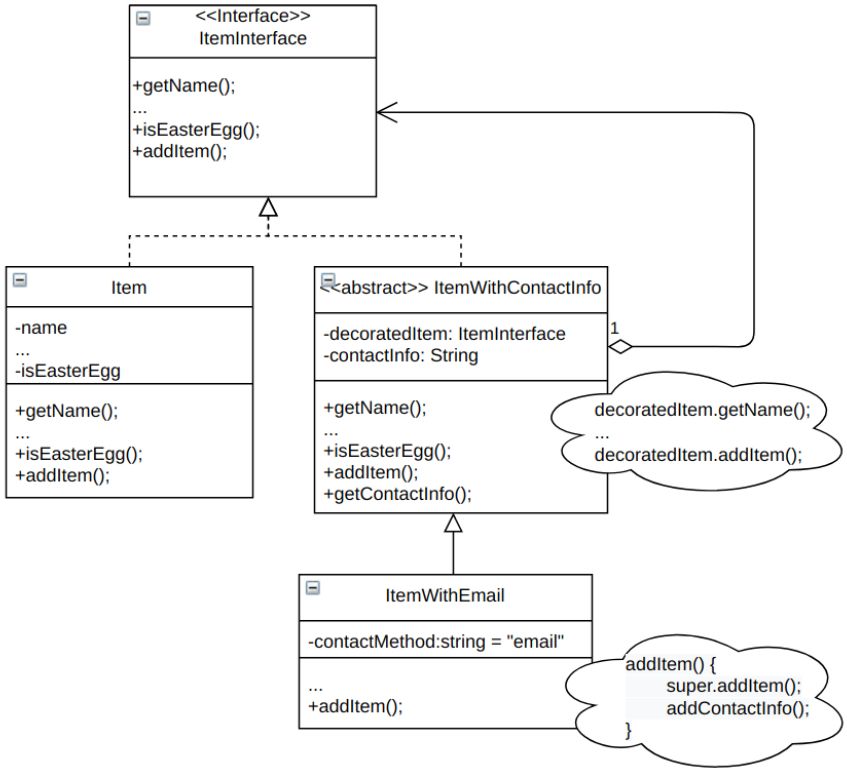

Fig. 6 Decorator Design Pattern

We apply the decorator pattern for the creation of different types of items that users upload. We created the ItemInterface interface to abstract the common basic operations. The Item concrete class implements these operations. We also created the decorator class ItemWithContactInfo that implements ItemInterface and wraps an ItemInterface reference. It has an extra contactInfo field. All the override methods forward the execution logic to the wrapee. The concrete class ItemWithEmail extends ItemWithContactInfo and adds its email-related implementation logic to the addItem method.

This pattern has several advantages. First, it avoids subclass explosion and allows responsibilities to be added at run time. Second, the interface inheritance of Item and ItemWithContactInfo decreases coupling compared with

implementation inheritance. Besides, it has the flexibility to allow for new decorators. For example, we may introduce ItemWithFacebook, or a new abstract decorator ItemWithVideo.

2) Strategy Design Pattern

Strategy design pattern is one of the 10 behavioral design patterns as per the famous GoF book. In general, behavioral design patterns provide software developers a mechanism for switching and diversifying algorithms and assigning responsibilities between different components. The strategy design pattern defines a family of interchangeable algorithms and then encapsulates the algorithms under an interface, so that the client code that needs to use these algorithms does not need to know the implementation details of these algorithms, and the client code can also change the algorithms dynamically through a setter method at runtime. In our app, the strategy design pattern is used as the following: In our app, we have an activity (UploadNewPlaceActivity.java) that need to display the latitude and longitude on the screen; however, there are many different methods that the app could get the latitude and longitude, and this create a chaos in the Activity's code. For example:

1) The app could get the user's current latitude and longitude GPS value from the google map service (a combination signal from GPS sensor and Wife Sensor);

2) The app could also start this activity from another activity via an intent

3) Or users can select a picture from the album, and then, the latitude and longitude from this picture's EXIF information should be used,

As we can see from the discussion above, these different algorithms we are using indeed introduced a lot of chaos in the activities code, a lot of conditions and helper functions which are in-relevant with the general purpose of uploading new place is coupled together, and this makes the activity's code very hard to read and difficult to maintain. Therefore, we used the strategy design pattern to minimize the coupling here. The strategy is implemented as the following:

1) We Introduced an interface, UpdateGPSBehavior, to encapsulate the abstract behavior of updating the GPS information, which is representing a family of algorithms that people can get an updated GPS information. This behavior/strategy has two abstract methods: the updataLat() method that will be able to update the client's latitude information, and the updateLog() method that will be able to update the client's longitude information; this behavior needs some concrete classes (concrete strategies) to implement these two methods.

2) At this stage, we have 3 concrete classes (strategy classes) implemented in our application:

1. The UpdateGPSWithGPSValue class, which represents the algorithm of obtaining the latitude and longitude from two string values which is presented by Google Map service.

2. The UpdateGPSWithIntent class, which represents the algorithms of obtaining the latitude and longitude from an Intent which has the pre-defined longitude and latitude value attached by another activity already.

3. The UpdateGPSWithPictureEXIF class which represents the algorithms of obtaining the the latitude and longitude from a picture's EXIF information, and convert type from GPS-DMS to GPS-DD;

3) The GPSUpdateManager is a "context" which has a private field pointing to one of these three concrete strategies, and communicates with this strategy only through the UpdateGPSBehavior interface.

4) At this stage, we have 3 clients using this design pattern, namely to be the GPSUpdateWithGPSValueClient, GPSUpdateWithIntentClient, and the GPSUpdateWithPictureClient. Each of these clients creates a strategy class, and the

specific strategy is assembled in the system via the context's set behavior (setUpdateGPSBehavior()) method, and is executed via the the perform behavior (performUpdateGPSBehavior()) method.

As we can see from the discussion above, by taking advantage of the strategy design pattern, the implementation details of the strategies/algorithms are separated from the client code, which makes our code nice and clean. Instead of letting the client inherit different algorithms, the strategy design pattern helps us to aggregate the strategies/algorithms into the client, then the client code will be able to use the set behavior method to change strategies/algorithms at run time. Last but not the least, following the Open/Close Principle, our UpdateGPSBehavior interface is open to all new strategies that will be introduced to our app in the future (open for extensions), but the GPSUpdateManger does not need to be modified when use strategy is introduced (closed for modification).

In the future, if we want to bring new features to our app, say, if we want the latitude and longitude value inside of the UploadNewPlaceActivity.java has the "magnetic property", that is to say, if a user is taking a picture within 1 meter range of an existing place, instead of using the user's current GPS to creating a new place, the app should automatically use and display the latitude and longitude value of the place which is already existed in our database. The already existing places are behaving like big magnets that will attract all user's GPS value within 1 meter range. To achieve this purpose, with the help of the strategy design pattern, we just need to create a new concrete strategy class, UpdateGPSWithMagnetic.java. The advantage of the strategy design pattern is that the GPSUpdateManger is not affected by this change, and a new client can just simply use this new strategy through the set behavior and perform behavior methods.

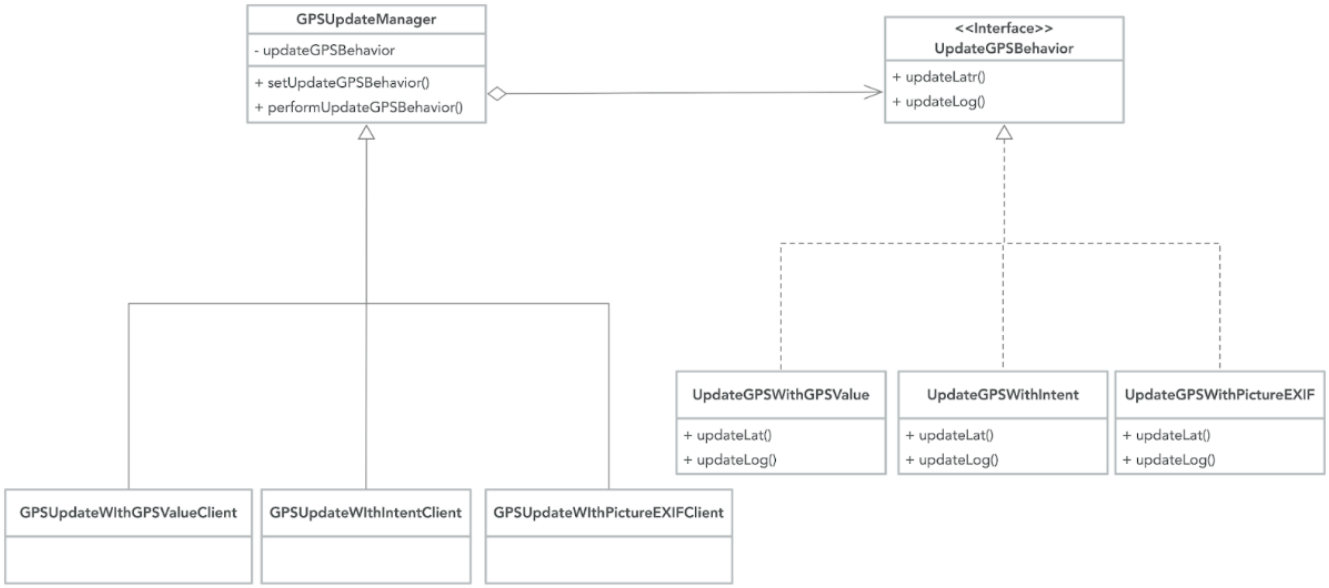

Fig. 7 Strategy Design Pattern

3 Mitigation to the harms

For mitigating the potential harms, two measurements are taken. First, a series of warnings will be given to the users when they open Watsloo. Second, the data (including stories, location, and images) uploaded by users are subject to review before being published to the public.

When the users open this app, a "Things to know" interface will be shown prior to the main interface. A series of warnings are shown in this interface to caution the users about the potential harms. And the users must read all the warnings and hit the "I acknowledge" button to access this app. The warnings are listed below, followed by a brief explanation.

- "Please do not use this app while walking and pay attention to the surrounding environment!"
  Similar to other tour guide applications, Watsloo may cause accidents when users are walking and looking at their phones simultaneously. This warning can remind users to be cautious about the surrounding environment and mitigate the potential harms caused by inattentive walking while using Watsloo.
- "Please do not access any destination within temporarily closed/under construction buildings or other inaccessible places!"
  The destinations recorded in Watsloo may be temporarily closed or under construction at some time in the future. This warning can mitigate the harm caused by walking into inaccessible places accidentally.
- "Please keep social distance while using this app for the health of others and yourself."
  During the COVID-19 pandemic, social distance is required at any public place for public health concerns. The reminder of safety distance can mitigate the harms caused by neglecting the social distance while using Watsloo.
- "All the information you upload will be reviewed; no violent, sexual, or disrespectful content is allowed!"
  The users are informed about the review mechanism and warned not to post any illegal information. Posting violent graphs or disrespectful descriptions can threaten the mental health of operation team members. This warning is able to eliminate the harm caused by attempting to post illegal information.
- "This application will not be liable to any loss caused by the violations to the warnings above."
  If accidents happen, the developers of Watsloo might be sued for the losses. The warning declares that users should take their own responsibility to avoid any pre-cautioned danger.

When users attempt to upload new stories, the uploaded data will be marked with the label "subject to review". Then it will be reviewed by the operational team to filter out the illegal and inauthentic information. Besides, postings on controversial or disrespectful topics will be rejected since they can cause harm to the reputation of the university. The posting with an inaccessible location will be rejected as well for safety reasons. The quality of the uploaded image is also an important factor in the review. The users can only see the reviewed stories while viewing the map interfaces. The review mechanism prevents users from viewing the unfiltered stories directly. Therefore, Watsloo protects the users from sexual, violent, or other harmful information, and provides meaningful and authentic stories and a safe user environment.

All the information uploaded, including image, location, descriptions, and contact email, is encapsulated into a package before being sent to the database, reducing the vulnerability subject to SQL injection or other kinds of attacks. Besides, Watsloo does not offer a searching function for the destination in the map/main interface, which means the users cannot input strings to query the database directly. That can also increase the reliability of Watsloo. And the access to the database is limited to the administrators. Therefore, the design of Watloo can also mitigate the harm of security threads.

4 Resposibility

Members	Classes
Mengyao Zhang	Maps, MapsFollow
Lu Wang	Spot, Detail
Ting Gu	DataOperation(read data from Firebase), Position
Zishuo Xu	DataOperation(write data into Firebase), Item*
Yaowen Mei	MainActivity, Upload New Spots (GPS info, get pictures from camera/album)
Yitao Hou	Upload New Spots (Description, Submission), Instruction