Litterbug – A Software Approach to Littering

DAVID MEDNIKOV, Oregon State University
HITESH VARMA, Oregon State University
KEVIN ALLEN, Oregon State University
ALEXANDER YFRAIMOV, Oregon State University

1. UML CLASS DIAGRAM

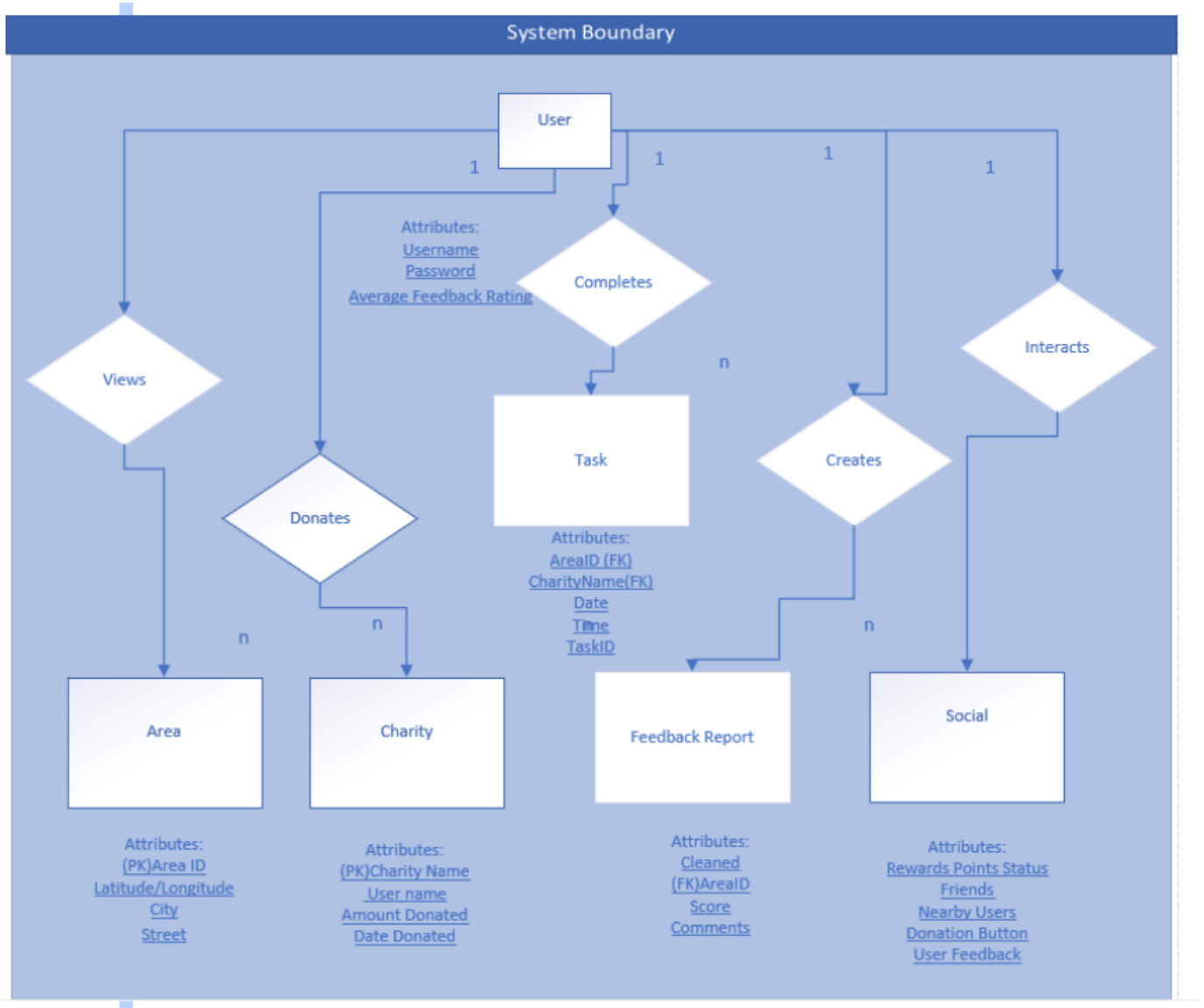

Fig. 1. UML Class Diagram

This work is supported by Michael Johnson of Oregon State University.

2. IMPLEMENTATION OF ENTITIES

2.1 Coupling

- **Definition** – Coupling is defined as when one module is involved in another module's concern. Coupling reduces maintainability.
- **Entities** – The entities we have involved are Task, Area, Charity, Feedback Report and Social.
- **Task** is coupled with Area and Charity, since it has foreign keys from both of these places. This is known as "stamp coupling".
- **Area** provides structured data to Task in the form of a Area ID, this is known as "stamp coupling" since the ID is derived from latitude and longitudinal data.
- **Charity** provides unstructured data in the form of the Charity Name. This is referred to as "data coupling" and is a more desirable form of coupling than stamp coupling.
- **Feedback Report** received structured data from the Area entity in the form of Area ID. This would qualify as stamp coupling.

2.2 Cohesion

- **Definition** – Cohesion happens when a module is devoted to its concern, this is essentially the opposite of Coupling. Cohesion increases maintainability.
- **Entities** – The entities we have involved are Task, Area, Charity, Feedback Report and Social.
- Due to the reasons in the "Coupling" section above, Task and Feedback Report, because of their dependency on Foreign Keys, lend themselves to low Cohesion.
- Temporal Cohesion exists between Area and Charity, they both can contribute to Task, but are unrelated.
- Procedural Cohesion is defined as: A executes, then B executes, and A and B have a vaguely related purpose. Procedural Cohesion exists when methods in Feedback Report get executed and they help with the points attribute in the Social Aspect.
- Task and Social have Temporal cohesion, Task executes first and then later on, the Social Method entity is influenced by the user getting a report that a task was completed.

3. SUPPORT FOR INCREMENTAL OR ITERATIVE DEVELOPMENT

3.1 Which One is Better?

We believe that our design would support incremental development better than iterative development. The customer already came up with a well-defined software concept, so if we were to practice iterative development, we might run out of things to work on after several cycles. It's hard to say that the software system is complete as currently defined, but we do not foresee enough new features or added capabilities to make iterative development a good idea.

3.2 Why Incremental Development?

Incremental development makes a lot of sense for our product. The product can be split into many modular pieces pretty well, and each piece can be thoroughly tested before building on top of that. For example, if we were to utilize incremental development for our product, we would probably develop components in the following order:

1. Create a database to store info about users, areas, tasks, etc.
2. Develop user profiles, badges, points, etc.
3. Create a list of charities that donations can be sent to and build the functionality to select a charity and send money.
4. Create a map server that pulls map data from the Google Maps API and provides it to the app.
5. Overlay green/red/colorless areas that are stored in the database onto the map in the app.
6. Create the ability to submit a task where a user reports that they cleaned an area.
7. Build functionality for another user to confirm whether or not the first user did in fact clean the area.
8. Add the ability to challenge users after submitting a task.
9. Develop a way to determine when a user is fraudulently reporting areas as cleaned despite them not being cleaned.

Each of these components is clearly defined and can be developed independently or on top of another component. Likewise, testing each individual component should not be too difficult to test. For this reason, we believe that our system design would be more favorable to an incremental approach as opposed to an iterative approach.

4. USEFUL DESIGN PATTERNS

When determining which design patterns would be useful for implementing the system, we had to take a lot of details into consideration. How would our various components interact? How complex are the interactions between our server, database, and map and payment APIs? How many attributes and pieces will each individual component require?

4.1 Which Design Patterns Would be Useful for Implementing the System?

The questions above us helped us determine that the Builder, Adapter, Façade, Memento, and Observer design patterns would all be useful for implementing the system.

4.2 Why Would These Design Patterns be Useful for Implementing the System?

BUILDER DESIGN PATTERN

The builder design pattern will help us develop the user profile and task entities. The user profile has many different aspects to it – a name, profile picture, attached social media accounts, points balance, favorite charities, completed tasks, sent challenges, medals and badges, etc. The user profile object will be quite complex in nature, especially since many of the attributes of the user entity are references to another entity in our class model. The builder design pattern makes it easier to develop and keep track of the user profile without getting lost in the complexity.

Similarly, the task entity has several components as well: the user that submitted the task, the date and time, location, attached photos, the user that confirmed the area as cleaned, etc. Once again, several of these are references to other entities on our system. Using the builder design pattern helps us split the entities up into smaller pieces and build from the ground up.

ADAPTER DESIGN PATTERN

The adapter design pattern will be useful when developing the interaction between our server and various payment APIs. Our product will accept payment in the form of credit/debit cards, PayPal, and

Google/Apple Pay. These are all individual services that our system will need to be integrated with. Using the adapter design pattern allows us to send and receive data from these external components and do it in an efficient manner.

FACADE DESIGN PATTERN

The façade design pattern will be used for a different API integration – the connection between our server and the maps API that we use to obtain map data. Getting the map from the API and converting it into the proper format to be viewed in the app will require a lot of code, certainly more than the payment integration. This integration will require information about the user's location, nearby landmarks and areas, as well as all the map metadata that isn't immediately visible to the user. For this reason, the façade design pattern makes the most sense for this component.

MEMENTO DESIGN PATTERN

The memento design pattern will be useful to remember the state of the map when the user sees it and preserve it for some amount of time, which is yet to be determined. This amount of time may be dependent on various factors, such as app popularity in that area, how strong the user's connection to the Internet is, how many areas are located nearby, etc. Refreshing the map every second would be very inefficient and resource-heavy, so for this reason, having a way to maintain the state of the map would be very useful, and the memento design pattern helps us achieve that.

OBSERVER DESIGN PATTERN

Last, the observer design pattern will help us implement two different functionalities – one where users have to report another area as cleaned or not cleaned, and another about the map. The map should trigger an update if there is a change to an area in a user's immediate vicinity. We do not want users to clean an area that has just been cleaned, so the map ought to update in scenarios where the status of an area in the immediate vicinity was changed. If there isn't a change nearby, then the map should continue to update on an interval as mentioned in the paragraph above.

The observer design pattern will also be useful so that a user is notified when there is a nearby area that needs to be confirmed for cleanliness, or if a user reported an area and is waiting for feedback from another user. When a second user confirms that the first user's cleaned area was actually cleaned, the first user should be notified and the system will add the task to the user's history and add badges/medals to their account. This sequence of events requires a trigger, which is why the observer design pattern will be useful.

5. MESSAGE SEQUENCE DIAGRAM FOR USE CASE 1

5.1 Use Case 1 Summary

User can use map to see areas near them and (on mobile only) mark them as complete before/during/after they have picked up litter in their area.

5.2 Use Case 1 Sequence Diagram

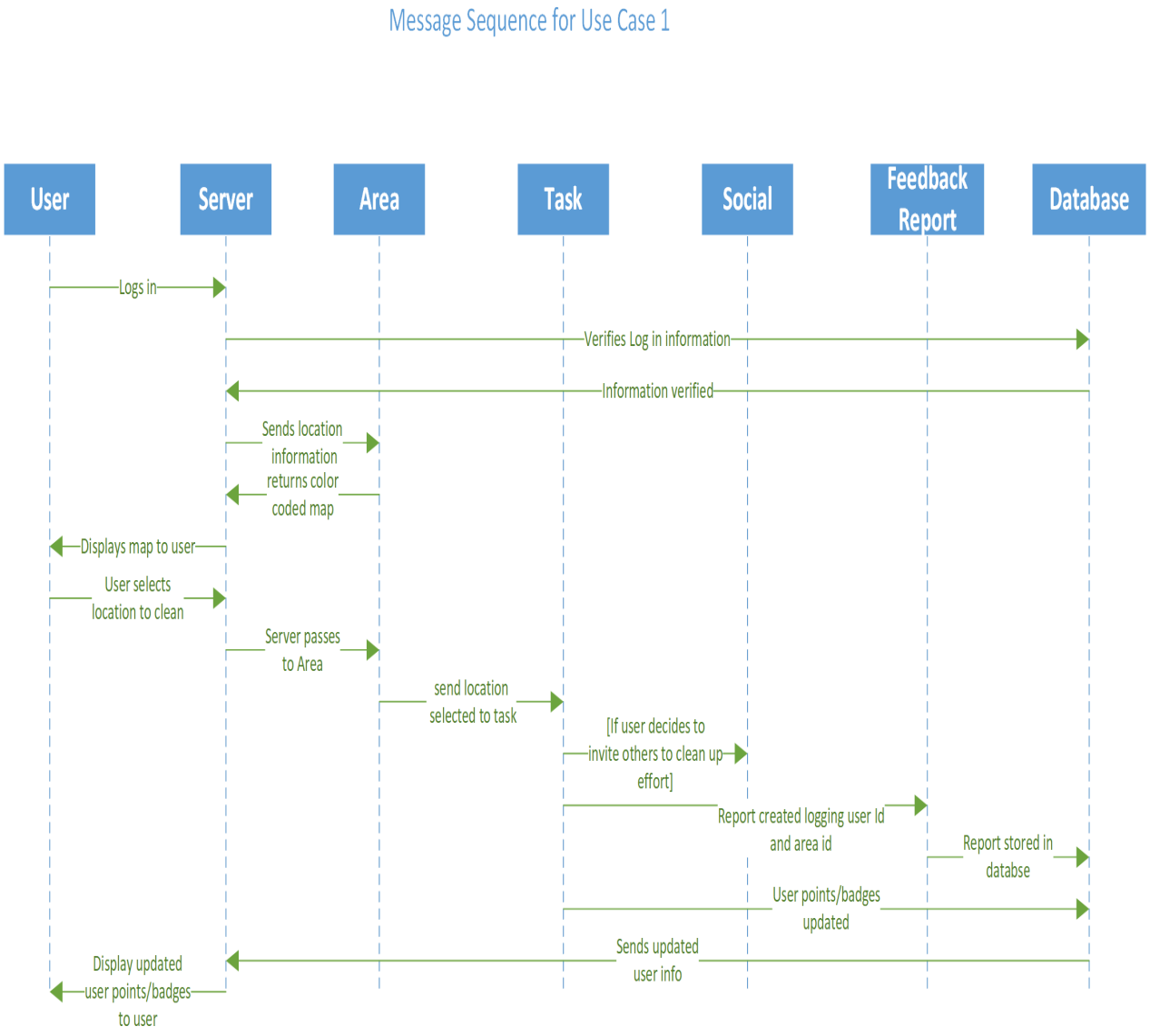

Fig. 2. Use Case 1 Message Sequence Diagram

6. WHICH INTERFACES ARE NEEDED?

6.1 Interface Connecting Map Server and Google Maps API

One key interface lies between our server and Google Maps API. When the server receives GPS coordinates from a user's mobile device reflecting their current position it needs to be able to provide this data to the Google Maps API (fulfilling the precondition). The contract in this case states that if that particular precondition is met, in exchange, Google Maps API will return the proper portion of a map. This map will show the region the latitude and longitude values supplied to it by our server pinpoints as well as some of the surrounding territory. Upon reception of this map, the server will need to exchange data through another interface that will apply the color-coding to the map.

6.2 Interface Connecting Map Server to Color-Coding Module

The interface between our server and the color-coding module is another extremely important interface as the usability of our application depends heavily on this exchange functioning as intended. The color-coding module expects that the server will send it a portion of a map that shows the position of the user as well as some of the surround area. In return it will return a color coded version of the map to the server demarcating the areas which are cleaned and areas which have yet to be cleaned (or previously cleaned areas where too much time has elapsed since the last cleaning and are presumed to be in need of cleaning). In order for the color-coding to even occur however, another interface must be crossed, the one between the color-coding module and our database.

The database stores up-to-date information on which areas are clean and which are in need of cleaning. The color-coding module must provide the user's current location and use that information to retrieve the pertinent cleanliness information from the database. To state it explicitly, the promise is that upon reception of the current location of the user, the database will return the proper information to the color coder so that it will be able to overlay the map appropriately and in turn return this color coded map to the server. The server upon reception of this finalized color-coded map can send it back to the client application running on the user's mobile device. This return to the client represents the post conditions of another interface which initiated all of the transaction of information described in the paragraphs above however, we still have not discussed the most important interface in the entire system, the user interface.

6.3 User Interface

The user interface represents an exchange between the user themselves and the client application. When the user opens the app they must fulfill the precondition of providing valid login credentials. Once that is met, the expectation is that the client application will display a color coded map of the surrounding area displaying clean and unclean areas in their immediate vicinity. However, there is another promise between the client application and the user that is somewhat abstract. It's the promise that the app will be structured/designed in such a way as to be intuitive, visually appealing and will react to user inputs as fast as possible. The post condition here is that users will actually be compelled to use the application, even frequently, if those preconditions were met. That's why we say this is the most important interface in the entire system - we could construct the most efficient, well thought out, etc. backend interfaces however, if the user interface is poorly constructed the user will reject the application outright and all the background work will have been for nothing.

7. WHICH EXCEPTIONS ARE LIKELY TO OCCUR?

7.1 Issues with the Google Maps API

One exception could be caused by Google Maps API being down or nonresponsive. If the client application is not able to get a map of their current location due to this then the exception handler would pull up a previously cached version of the area. Since the database that stores data about which areas have been cleaned is not directly dependent on Google Maps API its information would still be accessible. This would mean that the cached version of the map the user had could be overlaid with the appropriate color-coding. If however, the user is visiting a brand new area they have never been before then the exception handler would need to handle this situation by displaying a message to the user that Google Maps appears to be down and to try again later.

7.2 Issues Connecting to the Database

Another potential exception we need to handle is if communication with our database fails or is temporarily not functioning. Since the time a user marks an area as cleaned is important (as we require confirmation from another user that the area marked by the first user is in fact clean and this confirmation must come within a certain time window) we need a way to store the event and time stamp in the event that it cannot be logged in our database immediately. In this scenario our exception handler on the client application would need to store this data and periodically attempt to resend it until it receives confirmation from the server that the event has been transmitted and logged in the database. Since the app uses medals/badges as the primary method of reward (which in turn helps drive our users to continually use the app) it's important that a users work does not go unrecognized by the system just because a particular back end component has failed. The exception handler should alert the user that there is an issue on the backend but as soon as that's resolved the fact that they cleaned an area will be recognized, and once verified, they will receive any rewards they are due.

8. SUMMARY OF GROUP MEMBER CONTRIBUTIONS

- David Mednikov
  - Assess how well our design would support incremental or iterative development
  - Explain which design patterns would be useful for implementing the system
  - Compile the Google doc into the ACM format
- Hitesh Varma
  - Draw a UML class diagram showing the key OO entities
  - Explain how we would package the implementations of these entities, coupling and cohesion in particular
- Alexander Yfraimov
  - Identify several interfaces that would be needed
  - Identify several exceptions that are likely to occur
- Kevin Allen
  - Select a use case from HW2 and draw a sequence diagram showing how that use case would play out in the system