Hometown Heroes

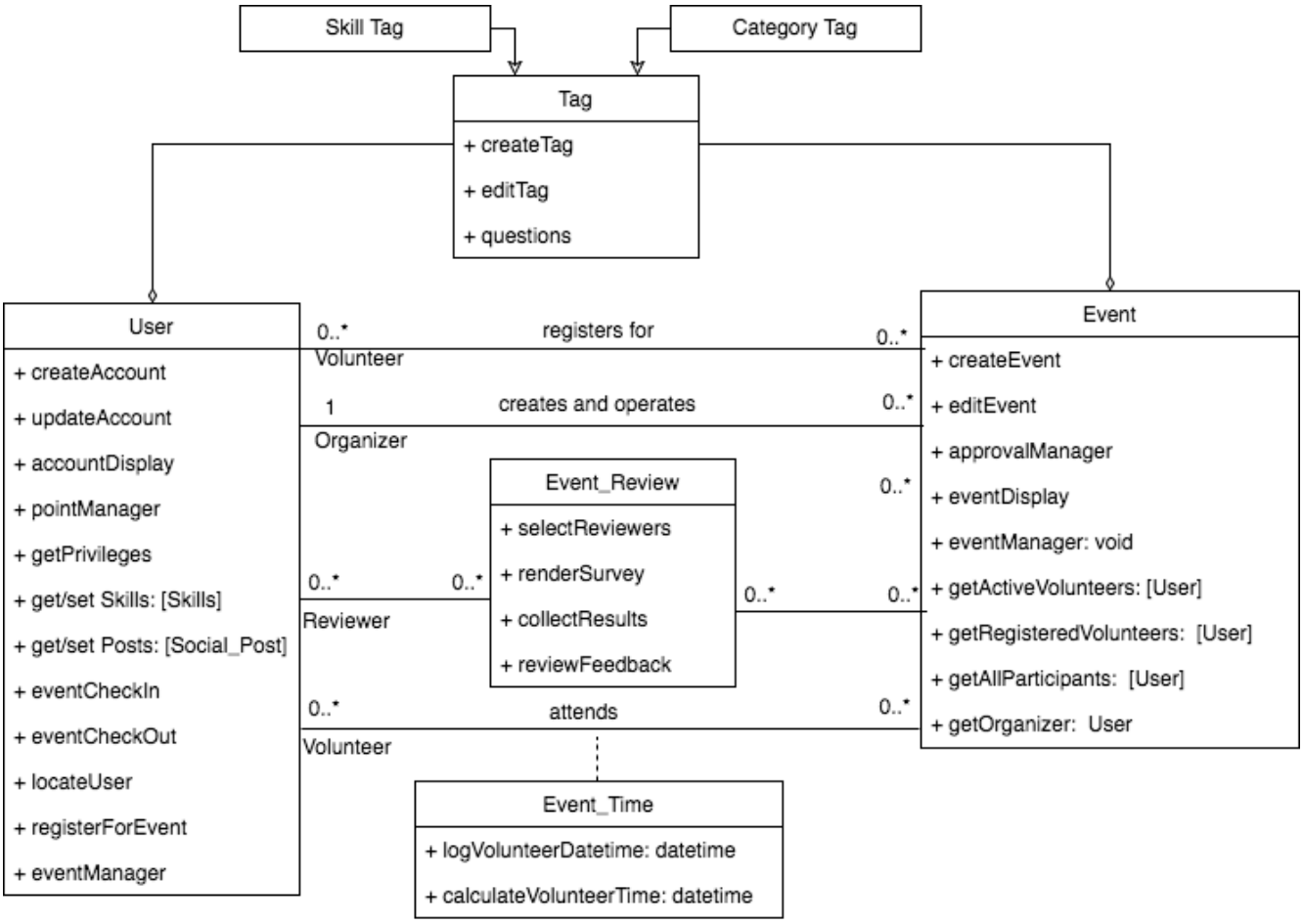
CS 361 - 400, Group 6
Homework 4
July 26, 2017

Client:
Benjamin Rodarte

Developers:
Jon Austin, Valerie Chapple, Kenny Lew, Gregory Niebanck, and Charlotte Murphy

1 UML Class Diagram

_:



The above diagram captures the key aspects of Hometown Hero: Users and Events. Additionally, basic properties of the entities are omitted as they were previously included in the UML diagram from Homework 1. The UML diagram above focuses on essential methods of each entity as they relate to other entities. There are additional entities not listed in the UML diagram. Those entities include details about event approval management, social media updates, administrative aspects for moderating events, and the rendering of event views.

2 Packaging Implementations

The implementations of the entities in the UML Class diagram are to be packaged based on their main concern. Certain methods and sub-entities primarily involve the user such as account creation, user authentication, management of social media updates, calculation of participation, and the GPS location manager for event participation. Likewise, methods that manage events, such as event creation and organizer management, event approval, and display of events, are to be handled by the Event System package. Note that all the components of the package are only accessible through the package interface.

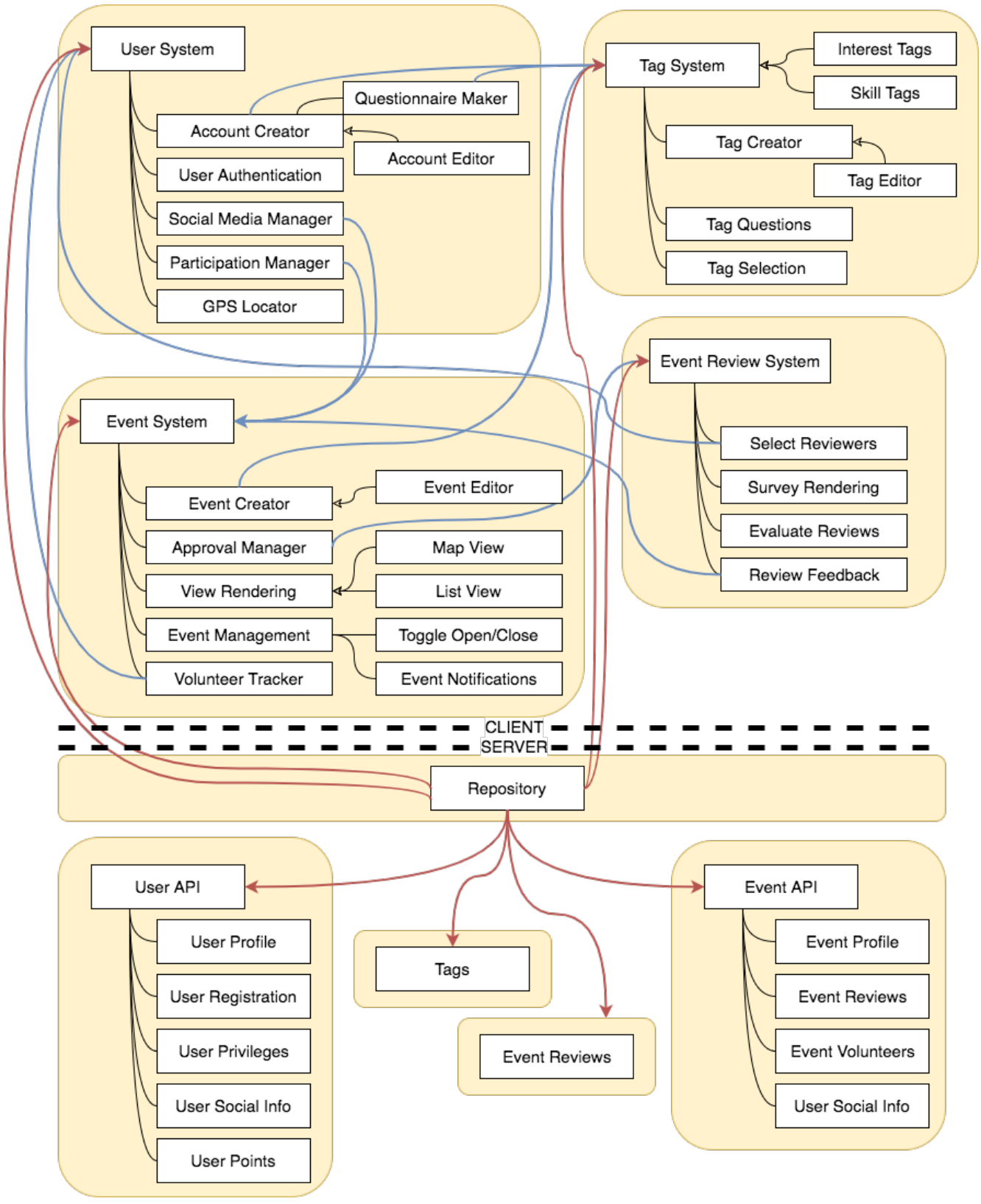
There are two additional systems on the client side that are worth mentioning: the Tag System and Event Review System. The Tag system is required to filter event and user results based on the needs of the users and events. It is also necessary to include relevant questions about each tag to include in the user questionnaire. The Event Review System is quite a bit more complicated as it must join an event and multiple users together in an approval system and then evaluate if an event is approved for publishing.

On the server side, there is a repository, which is in charge of accessing data on behalf of the app, as well as notifying users and events of appropriate changes. The repository has access to various sets of data: User, Event, Tags, and Event Reviews.

Unfortunately, this system comes with a certain amount of coupling, but it seems that this cannot be avoided with a repository system. The systems and system modules that create and modify users and events must always interact with the repository in some way. Our system uses mainly stamp coupling; user, event, tag, and review objects will be requested by modules that require access to data, and they will be provided back in a structured package like JSON. This way, our system avoids violating the Law of Demeter. Similarly, objects that are modified by system packages need to be updated in the repository. A JSON object is not required here, though; we can have “setter” methods that update database attributes individually.

There are a few areas where system packages may need to interact, though. The volunteer tracker module will need to call on the social media manager in the user system in order to make posts about users who volunteer. The account creator and questionnaire systems will need to call on tag system methods to be able to display and apply the appropriate tags. For example, based on a user’s questionnaire results, skill and interest tags must be retrieved from the tag system in order to be suggested to the user. Also, the event review system must interact with the user system in order to make users into reviewers and to check and see if a user is able to review events. We feel that these control and stamp couples are necessary to maintain the level of cohesion we wish to achieve with our design.

Our ultimate goal with this implementation is to keep cohesion to a maximum. Every system is organized based on its concern. User modules such as the account creator, the authenticator, and the participation manager are all grouped with the user system. Similarly, the event display, event creator, and event manager modules are all grouped in the event system. Also, within each system, each of these modules is designed to perform as close to one function as possible. Organizing modules this way creates a high amount of functional cohesion, which will increase the reliability and reusability of each system. Below is a high-level diagram of our system, couplers, and its cohesive parts.


User System
Account Creator
Questionnaire Maker
Account Editor
User Authentication
Social Media Manager
Participation Manager
GPS Locator
Tag System
Interest Tags
Skill Tags
Tag Creator
Tag Editor
Tag Questions
Tag Selection
Event Review System
Select Reviewers
Survey Rendering
Evaluate Reviews
Review Feedback
Event System
Event Creator
Event Editor
Approval Manager
Map View
View Rendering
List View
Event Management
Toggle Open/Close
Volunteer Tracker
Event Notifications
CLIENT
SERVER
Repository
User API
User Profile
User Registration
User Privileges
User Social Info
User Points
Tags
Event Reviews
Event API
Event Profile
Event Reviews
Event Volunteers
User Social Info

3 Development Strategy

The Hometown Heroes design primarily supports iterative development, though there are some opportunities for incremental development. For the app to work, the whole system must work to some extent. That is, the users, events, and database must have some basic functionality for the system to hold value. Therefore, once these basic modules are functional, changes and improvements can be iteratively made to them to improve the system.

When a user creates an account, the requirements state that they may choose to manually input their skill set or walk through a guided questionnaire. This questionnaire may change to improve usability and identify skill set more accurately. The social media interaction can be improved over time to add new kinds of social media posts as well as new kinds of functionality, like earning points for recruiting new users, to improve the user experience. The requirements for events include creating events, but event management tools can be added to the event such as updating the event information and sending event notifications.

Though the main system is comprised of the user, event, and database, we can utilize incremental development to add additional functionality once those modules are complete. For example, event reviews can be added after the event creation and user participation is built into the system. Furthermore, a ride sharing subsection can be included into the system for users to find rides to an event so that they can participate. The completion of the associated modules is necessary to implement a ride sharing functionality.

The User System is highly reusable, as it has a few features that are common in social applications. That is, social applications often require a user authentication system and social media publishing, which are very modularized. The Questionnaire Maker is also reusable; however it is limited to a survey instance, not a quiz instance (with correct answers).

Additionally, the repository itself is highly reusable. The basic skeleton of developing key functions to insert, read, update, and delete data is universal to all the objects. While some objects will be more specialized, each can be inherited from a base API class on the server. This allows for common functions to be created for all child classes but also add functions for specific children. Additionally, each API, such as the User API and the Event API, will communicate with the repository, not each other. This allows for more modularization, which in turn creates code that is more reusable.

4 Design Patterns

The most applicable design patterns for the app are builder, facade, observer, and adapter. The Builder design pattern should be used for new profiles and events. Profiles and Events are both comprised of complex data and associations. A builder would walk through the elements of each object and populate them with the appropriate data and relationships.

The Façade design pattern should be used for the event map view. The map view displays events within a geographical area that may be related by date or category tag. The façade will make calls to the event database and the GPS subsystems and return the results in a single, unified image.

The Observer design pattern could be used for new user (volunteer) event registration notifications. When a user (volunteer) registers for an event, the app automatically sends the user (organizer) a notification that alerts the organizer that a volunteer has registered for the event.

In addition to implementing event notifications, the Observer design pattern could also be used for user (volunteer) event participation. The user participant will be checked in and out of the event based on the device geolocation so an observer design pattern would watch for when the user GPS location enters/exits the event location area. When this occurs, the app would automatically respond by firing off instructions to the server to check the user in/out of the event.

The Adapter design pattern would be useful for social media integration. Because social media accounts are external interfaces, an adapter would be useful to for providing accessibility to social media APIs that allow the app to integrate the social media functionality into the system.

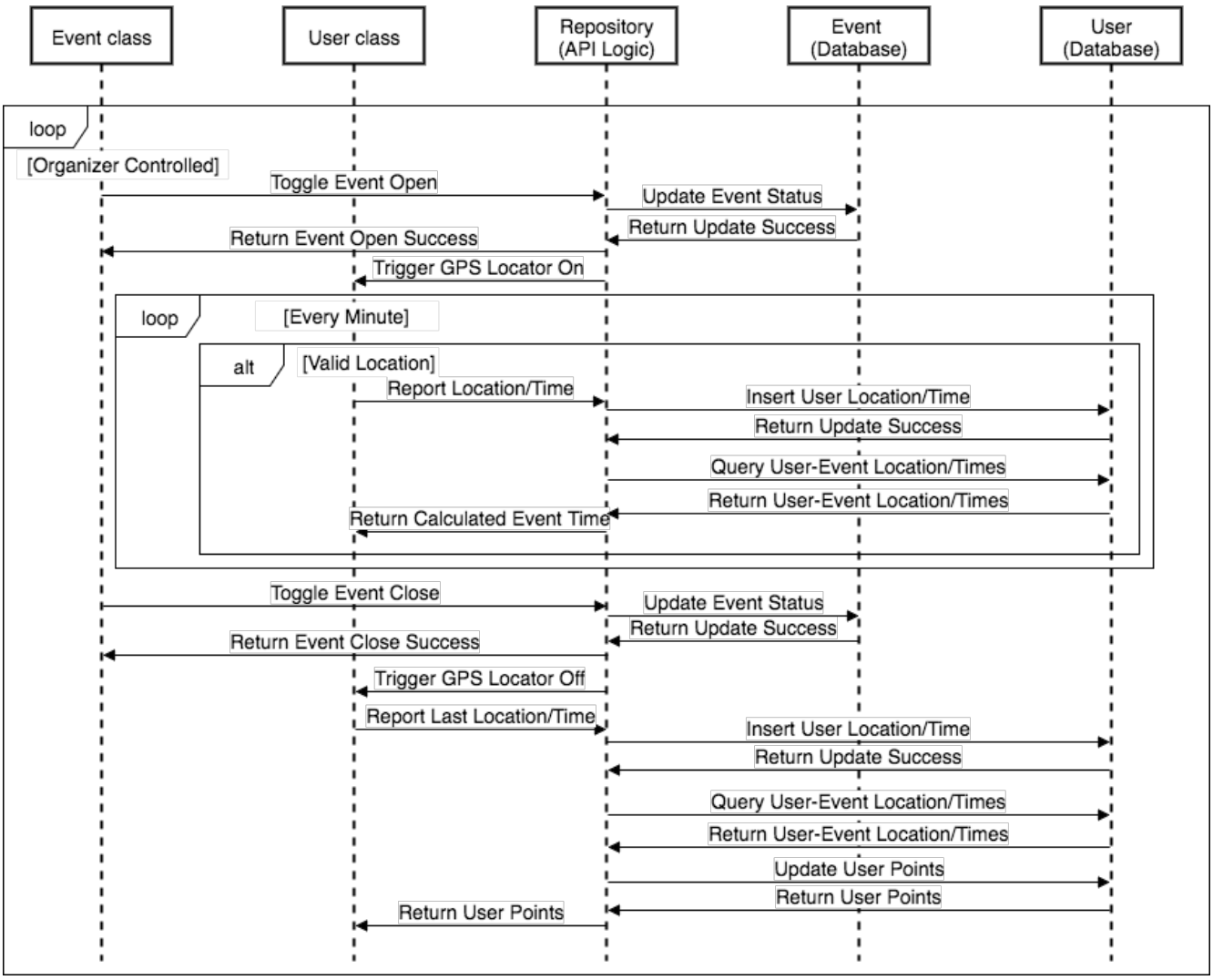
5 Sequence Diagram

5.1 Event Participation

Users who attend events will need to have their location tracked once the Organizer (User) opens the event on the day of the event. There are five classes of interest: Event class (client-side), User class (client-side), Repository (server-side), Event API (server-side), and User API (server-side). The classes on the client side will need to access information from each other during event participation; however, the information will be on multiple user devices. This requires constant communication to the server so that each device can have the most up-to-date information.

The repository is in charge of communicating with the server side architecture (Event and User databases). The repository is also responsible for notifying the User classes to turn on the GPS location. Consider the repository as the glue that connects client classes together, server classes together, as well as bridges the gap between client-side and server-side.

On the sequence diagram for the User Participation tracking, the Organizer (a user) can open and close an event as he or she deems fit. Also, once the event is open, the User class on the client-side will loop a GPS location test every minute. Once the user is in a valid GPS location, the client-app will report the user's location and timestamp to the server, where it will be saved in a database. This repeats until the organizer closes the event.



6 Interface Contracts

The Interface Contracts below address the core functionality of the app. The contracts identify four operations, the requirements they satisfy, and the results of each operation. [1]

6.1 Create a User Account Contract

Operation: createAccount()	
Preconditions	Postconditions

None	- A User was created - A Level was created - A pointManager was created - One or more Social_Post(s) were created - One or more Skill_tags(s) were created - User account was associated with a Level - User was associated with a pointManager - User account was associated with one or more Social_Post(s) - User account was associated with one or more Skill_Tag(s)

6.2 Create an Event Contract

Operation: createEvent()	
Preconditions	Postconditions
- A User exists - One or more Category_Tags exist (optional - see post conditions)	- An Event was created - One or more Category_Tag(s) were created (if the tags were not previously created) - An Event_review was created - An Event was associated with an Event_Review - An Event was associated with one or more Category_Tag(s) - Event.num_vol_req >= 1 and Event.num_vol_req <= maxVolunteersAllowable

6.3 Register for an Event Contract

Operation: registerEvent()	
Preconditions	Postconditions
- A User exists - An Event exists - Event.num_vol_reg <= Event.num_vol_req - 1	- A User was associated with an Event - Event.num_vol_reg <= Event.num_vol_req

6.4 Participate in an Event Contract

Operation: participateEvent()	
Preconditions	Postconditions
- A User exists - An Event exists - A User is associated with an Event	- An Event_Time was created - An Event_Time was associated with a User and an Event - Event.activeVolunteers >= 1 - User.points is now >= (minutes participated * User.pointMultiplier)

7 Exceptions and Handlers

7.1 Exception Handling

Important nonfunctional requirements for this application are encouraging participation and ease of use. Our system requires organizers and volunteers to make some key interactions with the application surrounding event participation. The point system is the main specification that promotes and encourages participation. Several exceptions could occur during participation that could affect score keeping. Handlers need to be in place to maintain the integrity without sacrificing usability during an event. Device local caching needs to be employed for various data to ensure participation is not interrupted.

7.2 Organizer ‘Post hoc’ Exception Handler: Organizer fails to open or close an event

The user with event organizer permissions is required to open and close an event for scoring. If the organizer neglects to do so, the event management module needs to initiate a push notification and ask the organizer to open an event and initiate scoring. Organizer negligence should not penalize participants. This reveals a need for a ‘post hoc’ exception handler to process the scoring of events that either never get opened or start late. The initial push notification would be handled by the basic event management >> event notifications module. Notifications that go unanswered for a predetermined amount of time will initiate a ‘post hoc’ exception which will ask the organizer about the event. Did it start on time? Did it end on time? Was it canceled? In this fashion, participants can receive points after an event when the organizer answers those questions.

7.3 ‘Connectivity’ Exception Handler

Let’s say a certain volunteer activity involves cleaning up garbage and debris along a stretch of highway that has spotty cellular coverage. This exception is an important reason to employ local caching. Under normal circumstances the user>>participation manager and user >> GPS module

interact with the repository to update the user profile through the user API. The data collected about a participating user is packaged locally and periodically updated in the repository. If connectivity fails during an attempt to update a user profile, a ‘connectivity’ exception occurs which causes the application to cache the update payload. This behavior continues until the connectivity returns, at which point the ‘connectivity’ exception handler creates a cumulative update package spanning the outage gap.

7.4 Offline Mode

If the user lacks internet connectivity at application launch, the application should still be functional to some degree. Under normal circumstances, the application would access the repository to access the most recent details for the information a user is attempting to access. If the application is offline at launch, the offline mode exception handler needs to limit some application functionality and rely on some information from local caching. The application still needs to be able to push reminders to the user about upcoming events, and the user still needs to be able to check in to an event without having internet access. For participation services, the offline mode event handler would pass the GPS logging to the ‘connectivity‘ exception handler, which would cache update packages as discussed above. The offline exception handler’s main purpose is to notify the user that event data is not up to date and that they should connect to the internet for the most recent information. A user could still attempt to register for an event through update package caching, but should be notified that the event registration will not complete until the application has internet access. Registration, in this case, would occur through the cumulative update package. Users would be notified of the success or failure of registrations that occur during connectivity outages.

7.5 Volunteer ‘Post hoc’ Exception Handler: Device Failures

Imagine a user volunteering at a soup kitchen. The user checks into the event by tapping the check in button on the interface. Perhaps the excitement involved in participating in an event causes the user to tap so hard, that the phone flies across the room, and lands in the cauldron of soup that was about to be distributed at the event. This is one example of how a mobile device could fail during participation. As discussed above, under normal circumstances the participation manager would periodically update the repository. During outages, the connectivity handler would employ update package caching. In this scenario, the device would not be caching anything but soup. The main function of this handler should be to detect the lack of a definitive end to user participation. These endpoints would be the GPS log data at the closing of the event, the user checking out of the event, or the user leaving the proximity of the event. This handler could employ a post hoc exception to which the user could be offered the chance to explain the device failure. This needs to be discussed with the client, but there are many ways to respond to a device failure. Among these are Option A: Too bad, so sad. Device failures cause scoring to stop. Option B: Honor System, upon next successful login the user is prompted to indicate when they stopped volunteering. These changes would then be reflected when updating user points in the repository. Option C: Conditional update opportunities. In this scenario, users of a certain level would be granted the honor system, but also keep track and deny honor system privileges under suspected abuse.

References

[1] Bultan, Tevfik. 2008. Lecture 2: Design by Contract. Retrieved from https://web.njit.edu/~gblank/cs602/Operation%20Contracts.ppt.

Team Contributions

Jon Austin

- Participated in team meeting, 7/25
- Package Implementations
- Coupling and Cohesion

Valerie Chapple

- Participated in team meeting, 7/25
- Collaborated on UML, Package, and Message Sequence diagrams
- Wrote descriptions of diagrams and implications on quality attributes

Kenny Lew

- Participated in team meeting, 7/25
- Collaborated on Development Strategy
- Collaborated on Design Patterns

Gregory Niebanck

- Participated in team meeting, 7/25
- Exceptions and Handlers

Charlotte Murphy

- Participated in team meeting, 7/25
- Interface contracts
- Collaborated on Development Strategies
- Collaborated on Design Patterns