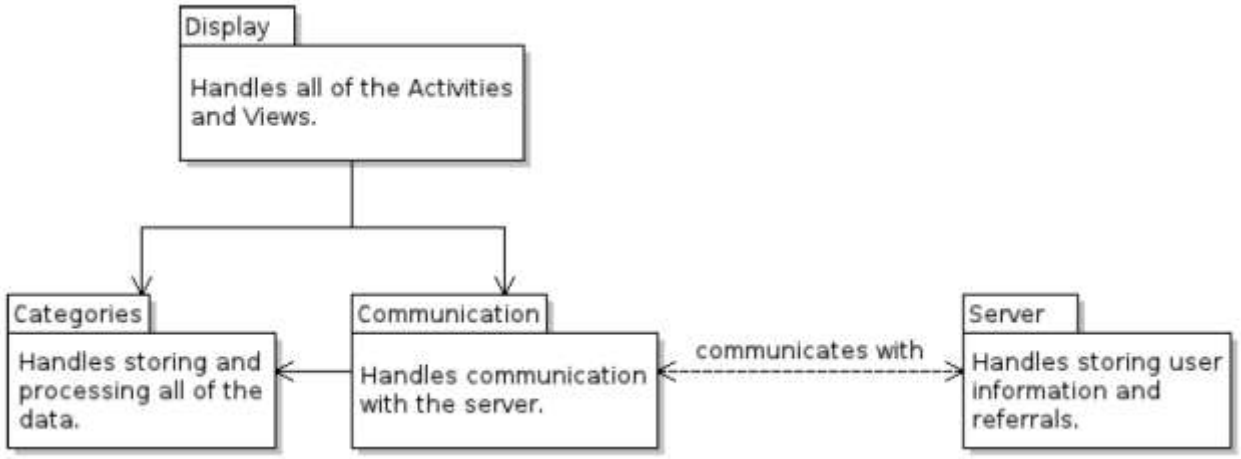
System Architecture

WeShould

Colleen Ross ◊ Troy Schuring ◊ Davis Shepherd ◊ Lawrence Tjok ◊ Will Pitts

The *WeShould* system consists of a handful of major, top-level modules: the Categories module, the Display module, the Communication module, and the Server module.



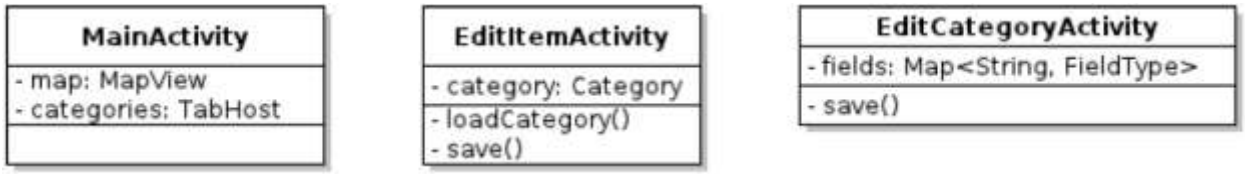
- Categories: The Categories module deals with the categories, tags, and items that the user wants to remember. It handles finding all items in a given category or with a given tag, as well as any computation that must be done for item data (such as returning any Redbox kiosks holding a movie when asked for that movie's location). Categories does not have knowledge of any other module.

- Display: The Display module deals with displaying the data stored in the Categories module, as well as adding new items. In short, the Display module holds all of the UI elements, and thus has knowledge of both Categories and Communication.

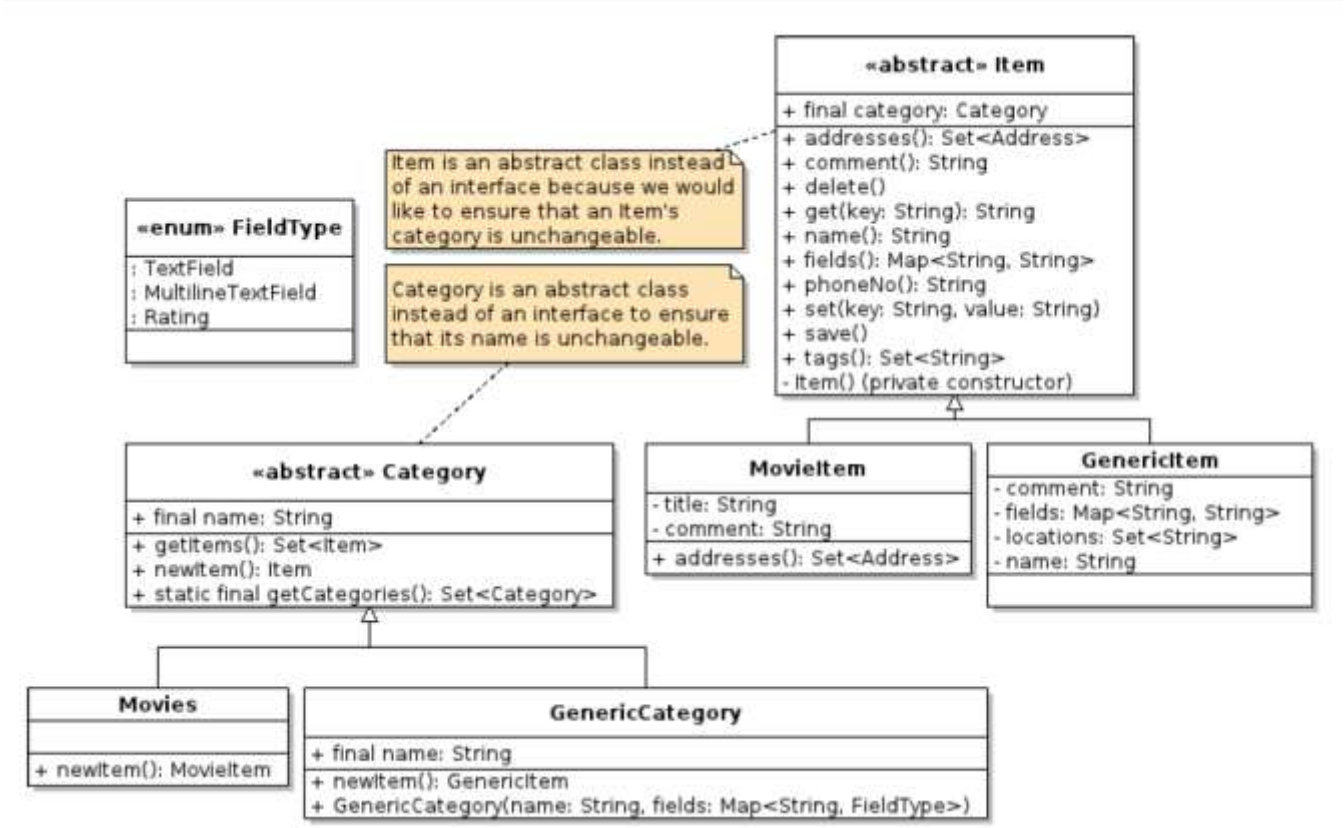
- Communication: The Communication module handles communication with the Server module in order to allow users to refer places to their friends. It has knowledge of the Categories module, as it needs to send and receive items, and knows the API of the Server module.

- Server: The Server module runs on a centralized web server, and backs up user data and stores pending referrals. Server does not have any knowledge of any of the other modules, as it simply communicates through HTTP. From the customer's viewpoint, there are only the Display and Categories modules, in that they have some form of stored data and some way of viewing it. The customer will see the map view, the add item view, and the add category view.

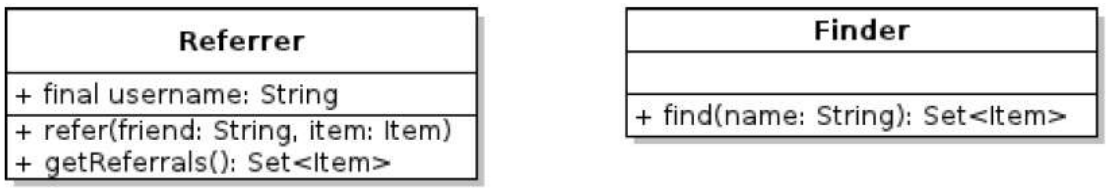
Display:



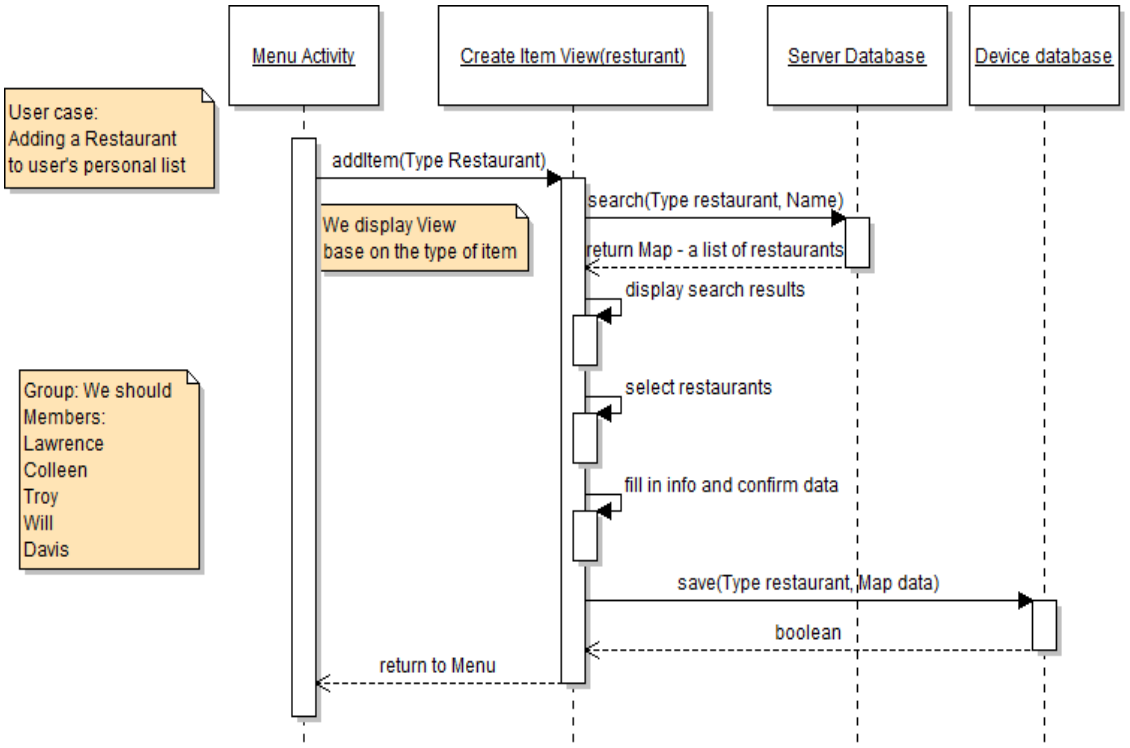
Categories:



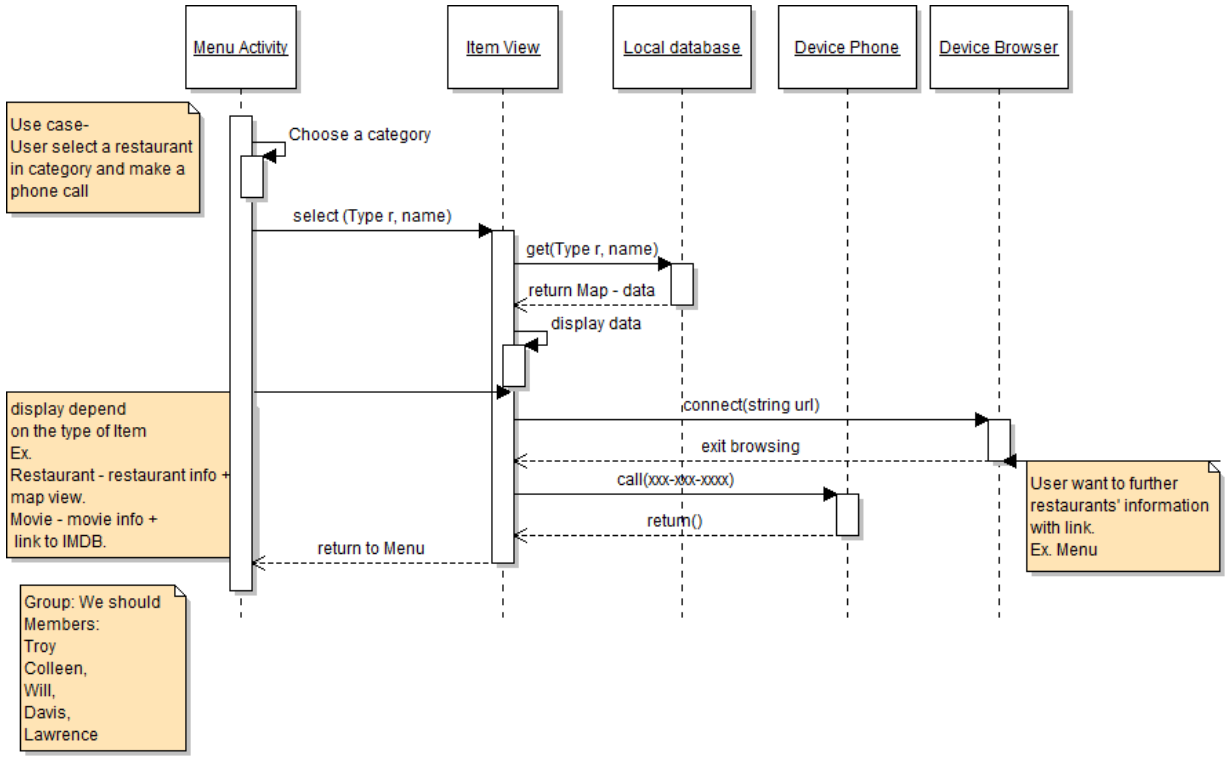
Communication:



Use Case Diagram: Add a Restaurant



Use Case Diagram: Select a Restaurant



Risk Assessment

Since the SRS, we have had more time to look into what libraries we are going to be using, and to get a better idea of our team members' skill levels with various product components, so that we have a better idea of how long it will take to do things and where we'll struggle.

We've also strongly defined the core components of our product, made a commitment to not overexert ourselves beyond those components, and had time to think about what could go wrong with the product now that we have a better idea of what our architecture is.

**See Table 1 for a view of our Risk Analysis

Project Schedule

**See Table 2 for a view of our Projected Schedule

Team Structure

We have chosen the following team members to be responsible for the listed portions of the project.

Will Pitts – *UI* – activities, page & menu design
Colleen Ross – *Cloud* – Server DB, account, device-cloud communication
Lawrence Tjok –*Google API* –mapping, searching, location, login
Davis – *Item Classes* – class design & implementation
Troy Schuring – *PM*, *Internal DB* – device database implementation, code & test review

As workloads will vary throughout the project, we are all willing to jump in and assist where needed or work on extra pieces that fall onto the 'Other' list. Without having specific requirements on the scheduled releases, a tentative schedule has been put together. It is noted that some tasks require rely on the completion of others before being started. The database is not dependent on any other portion, so Troy will complete the majority of that piece and load some test data so that others can do queries/updates to test their code. Lawrence will need a space in the list view page to begin integrating the Google map, and Colleen will need a login page to test the user login and authentication. Will is going create the required pages with the title displayed and minimum layout and functions to get them started. The device database will not require much time, so Troy will help out wherever he can. The team is communicating by email as needed and meeting every Tuesday from 9:30 – 10:20 in CSE025 All code we push to the repository will be tested prior to the push and reviewed after by Troy or another team member depending on current workloads.

Test Plan

Unit Tests will be written by those writing the sections to be tested (i.e. Implementers will write their own tests). Tests will be written prior to any implementation to ensure true black box testing of the specification, and will be run, and must pass, prior to a repository PUSH to the origin remote. These tests are designed to ensure that the individual modules/classes fulfill their specification prior to system integration/visibility.

System tests will be written by those responsible for their system components, these will test the integration of the various components of each module. Complete system tests will be written by all members of the team prior to system integration, but will not run until after full system integration. This is done in an effort to ensure that tests will be written to test spec and not implementation. System tests will run automatically once per day upon system integration, and any test failures will flag for repair. Complete system tests will primarily consist of Android Activity JUnit tests that interact solely with the GUI, to test the user side specification.

Both unit tests and system tests will utilize the JUnit framework, but usability tests will consist of user surveys and questions. We will attempt to get user feedback on the functionality and usability of the various features of WeShould to see if there are UI or functionality tweaks that need to be made. The results from these tests will translate into specification changes and/or test introductions. Usability tests will be performed once after each release of WeShould

Bugs will be tracked with gitHub's issue tracker. This allows for bugs to be tagged and assigned to individuals, and subsequently checked off if fixed. This will integrate well with our existing architecture, and allow for rapid communication of bugs to other team members. Anytime a team member detects a bug that they suspect is in another module outside of their own, they will submit a bug report to the issue tracker. The PM will then delegate repair of that bug. This same system allows for users to submit bug reports as well as requested features and other general issues with WeShould.

Documentation Plan

Integrated help text: On each page we will have a menu with a help option, which will pop up a page with instructions for common operations for that page, a link to an index of such pages, and a contact email

Help Index: As mentioned above, an index of all help pages will be made available to the user

Contact email: A contact email will be made available to the user via the help page, the help index, and as its own option in the menu. This email address will be checked regularly by one or more members of the team and will be used to assist users with questions and as a means for users to identify bugs to the team

User Guide/Tutorial: We will create and link to an online user guide/tutorial with instructions (including screenshots), links to our help index pages, and the contact email

Coding Style Guidelines

The language we will be primarily using is Java. It will be used for both the local and cloud database communication also. We will be using Javadoc for documentation, JUnit for testing, and XML will be used for some page layout. We have adopted the Google Android style guide. It follows the Oracle Java guidelines plus a few extras. We are also using Oracle Javadoc guidelines and Google XML style. JUnit test guidelines are borrowed from Geosoft. Our team has agreed to the following code style guidelines. We will write our own JUnit tests for the code we are responsible for. The PM will review code as it is pushed for consistency.

- Android - https://sites.google.com/a/android.com/opensource/submit-patches/code-style-guide
- Java - http://www.oracle.com/technetwork/java/codeconvtoc-136057.html
- Javadoc - http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html#styleguide
- XML - http://google-styleguide.googlecode.com/svn/trunk/xmlstyle.html
- JUnit - http://geosoft.no/development/unittesting.html

Table 1: Risk Assessment

#	Risk	Likelihood	Impact	Estimate Evidence	Detection Plan	Mitigation Plan
1	List data will be corrupted/lost on phone	Medium	High	Since there are many ways that the data can be corrupted or lost, there is a reasonably good chance that it will happen to one or more of our users	If we get an exception when reading/writing the data, or if we are unable to load the list from the device database	Keep a backup of the list on an external server and restore if corrupted, watch for I/O errors and reattempt if one occurs
2	List data will be corrupted/lost on server	Medium	Medium	Since we are using a storage service, we should be able to count on having backups, but there is a reasonably good chance that at some point we will encounter an I/O error reading or writing to the remote database	If we get an exception when reading/writing the data, or if we are unable to load the list from the remote database	Watch for I/O errors and reattempt if one occurs
3	Phone loses access to internet	Medium	Medium	Based on general phone usage and coverage, there will be times and places where internet access is unavailable	Check on app use/startup for internet access, or catch errors	Allow users to view their lists in a list view from the device database, but disable mapping if necessary
4	Underestimate time needed to learn APIs and Android programming (the most likely of which are general Android development and interactions with the remote db)	High	High	No member of the team has ever dealt with interfacing with a remote database or storing things on Google Cloud SQL, a d few members of our team have any experience writing for Android	We check to see in our weekly meetings and when assignments are due that we are on track	Rearrange the workload, eliminate extra features, get help, or switch to easier APIs
5	Database integration may fail (May be too expensive, or we may not be able to learn how to interact with Google Cloud SQL in a timely way)	Medium	High	No member of the team has ever dealt with interfacing with a remote database or storing things on Google Cloud SQL	Check in early on in the project to ensure we are able to connect and write to the database, and that storage costs are not too high	try using another cloud storage service such as Amazon RDS

Table 2: Schedule

Task	Subtasks	Time (hrs)	Deadline
Complete			
	Eclipse w/Android & Google setup		
	Empty project in repository		
	Splash Screen		
Zero Feature Release			5/3
	UI - login page, display test list items & map, menu options displayed, pages available with title & description. *REQUIRES* - database with queries, test data	18	
	Device DB - tables complete, add, delete, update, query, insert test data	6	
	Cloud - connect, login, tables complete. REQUIRES - Login Page	12	
	Item Classes - basics for demo	18	
	Google API - Login functions, search demo, simple map. *REQUIRES* - List view page with map area allocated.	12	
	Testing - All completed portions tested	12	
	Documentation - Code documented with Javadoc & inline comments as written. Application help menu available to guide user through current functionality. All knows bugs recorded.	6	
	Other - apk w/ documentation	1	
Beta Release			5/15
	UI - insert list items, send/receive	12	
	DB - complete	4	
	Cloud - send/receive recommendation, backup list	12	
	Item Classes - complete functionality	12	
	Google API - search, complete map display	12	
	Testing - All completed portions tested	12	
	Documentation - code documented with Javadoc & inline comments as written. Application help menu complete. Online documentation draft	6	
	Other - click web link to browse, copy paste url from browser, click phone# to call	12	
Final Release			5/30
	UI - user review modifications, enhancements, bug fixes	12	
	DB - bug fixes	2	
	Cloud - restore from backup, bug fixes	12	
	Items - enhancements & bug fixes	12	
	Google API - user review modifications, enhancements	12	
	Testing - All completed portions tested	12	
	Documentation - Application help menu complete. Online documentation	6	