ROLE

Design and Planning Document
03/8/10, Version 2.0
Adam Clay, Karl He, Glen Kim, Saung Li, and Tian Wang

Meta-specifications: http://inst.eecs.berkeley.edu/~cs169/sp10/doku.php?id=proj4

Note: the Requirements and Specification Document has been updated to version 1.1 and can be used as a reference for this document.

Implementation Plan

User Stories with Tasks

Note: Most User Stories require prior setup of the back-end server. This is not noted in the table below for each story, to eliminate redundancy of presentation.

Story Cost	Task Cost	**Story in Bold**, Tasks indented below, *Notes in Italics*	Iteration
155		**As a new player I can create an account.**	2
	70	Display the CreateAccount view	
	10	Ask the user the account name	
	15	Check to see if the name is taken	
	10	Ask user for password	
	15	Check password against requirements	
	15	Create account	
	5	Retrieve and link the account to the new installation	
	15	Ask for permission to use user's location	
		Dependency: Character Data Persistence (Server setup) must be implemented first	
70		**As an existing player I can link my account to a new installation**.	2
	30	Display UseAccount view	
	10	Ask the user for the account name and password	
	15	Check login credentials	
	10	Retrieve and link the account to the new installation	
	5	Ask for permission to use user's location	

		Dependency: Account Creation functionality must be implemented first.	
35		**After linking an account to my installation, the application should log in when I start it.**	2
	20	Device sends login data to server	
	15	Verify login credentials	
		Limitation: The user can only use one game account from each phone.	
205		**I want to create a character with distinct advantages and disadvantages.**	1
	70	Display the SelectClass view	
	10	Ask the user to pick the character's class	
	100	Display information for the class selected	
	10	Ask user to confirm selection or make another selection	
	15	Create the new character	
175		**I want my character data to be persistent.**	2
	25	Android app writes to server - XML over TCP	
	25	Android app reads from server - XML over TCP	
	125	Creation of the server - Rails	(1)
		Limitation: TCP requires Android app to poll server.	
230		**I can open the map view and see my location and the location of other players near me.**	
	25	Send current location data to the server (GPS)	
	30	Retrieve Google map data for current location	
	50	Retrieve locations of nearby players	
	125	Display map data with player-location data	
75		**I can select another character by touching his/her icon on the map view.**	
	10	Process touch coordinates	
	20	If coordinates are near more than one object, resolve with dialog box	
	5	Store selection	
	40	Indicate selection on Map view	
		Dependency: Map functionality must be implemented first.	
		Dependency: Character List Selection functionality must be implemented first.	
70		**I can select another character from a list of characters.**	1

	15	Retrieve collection of players from server	
	25	Display CharacterList view	
	25	Process touch coordinates	
	5	Store selection	
		Limitation: This story does not allow for combat range considerations, and will be superseded when Map View is implemented.	
80		**With a character selected, I can send another player a message.**	
	30	Receive message text from user	
	10	Send message to server	
	15	Server send message to selected player	
	25	Selected player receives message for viewing	
75		**With another character selected, I can inspect that character.**	2
	10	Send request for data to server	
	15	Server retrieves and sends data	
	50	Display up-to-date data for user	
180		**With a character selected, I can quickly initiate a battle.**	1
	100	Display battle view	
	10	Notify server of battle status	
	15	Server sends battle notification to selected player	
	45	Selected player's device asks player to choose to battle or ignore	
	10	Selected player's device displays battle view if appropriate	
135		**In battle, I can see what my opponent is attempting and respond in real-time**	1
	15	Receive opponent action data from server	
	100	Display opponent action in battle view	
	10	Receive player input action	
	10	Send player action data to server	
210		**In battle, I can make special attacks or defensive moves with broad physical gestures, such as swinging a blade.**	
	200	Process gesture input and interpret as action	
	10	Send action data to server	
115		**When I close ROLE, my character should stay in the game world at my current location, and defend itself if necessary.**	
	10	Send logout notification to server	

	5	Server sets character status as away/offline	
	100	Server chooses player actions when engaged in battle	
70		**I want to view information on my character's status.**	2
	10	Send request for data to server	
	10	Server retrieves and sends data	
	50	Display up-to-date data for player	
65		**I can select some skill(s) to advance while I'm offline.**	
	40	Receive player selection of skill to advance	
	10	Send selection data to server	
	15	Server gives updates on skill advancement at each login	
45		**I can advance faster by actively playing.**	
	35	At the end of battle, server calculates advancement/regression for characters	
	10	Server sends advancement info to players	
45		**I can advance fastest by battling more difficult opponents**	
	35	At the end of battle, server calculates ranking information for characters	
	10	At the end of battle, server calculates ranking information for characters	
		Dependency: Advancement for actively playing must be implemented first	
270		**I can select my offline response to attacks.**	
	10	User selects to customize AI control from status view	
	250	User selects options from list-boxes on AIControl view and confirms	
	10	Server receives and stores AI control data	
135		**I can equip special gear won in battle.**	
	10	User selects to equip gear from the Status view	
	85	From a list of gear, user can select an item and see relevant information	
	15	After selecting an item, user can choose to equip it	
	25	Server recieves and stores date on equipment configuration	
120		**I will get advantages to staying in the locations that I frequent in my daily routine.**	
	30	Server records time of character transitions between map zones	
	20	Server records total time spent in each map zone	

	15	On initiation of batlle or entry into Status view, Server receives location information	
	25	Server calculates effective character stats and sends them to device	
	30	Device displays relevant data on Battle and Status views	
130		**I want to be able to see listings of the top-ranked players by a variety of indices, and view my own position in those rankings.**	
	25	From Status view, user chooses to enter Rank view	
	10	Server sends Ranking data to device	
	50	Device displays relevant data	
	25	User may select different listing criteria	
	20	Device reformats display as needed	

Analysis of Iteration 1

After the initial planning of iteration 1, some changes were made, requiring an update of this document to version 1.1. Because of unforeseen challenges in creating an interface between UI components of the client application and components of the client-server interface, the user story, "**I want my character data to be persistent.**" was moved from iteration 1 to iteration 2. However, one task of that user story, "Creation of the server" was largely completed in iteration 1. These on-the-fly changes to the first iteration yielded repercussions in other elements of iteration 1, as some included user stories were not completed as planned; tasks involving client-server communication were not fully implemented. To address these shortcomings in the last iteration, the current iteration includes selected tasks from iteration 1, detailed below.

The estimation of task difficulty in iteration 1 also evinced some (not entirely unexpected) imprecision. The challenges inherent in working with one transfer protocol, another markup protocol, a server-implementation language unfamiliar to some team members, and a platform-specific development structure new to some team members, introduced variance into the estimation process. Notably, the differences between estimation and actual difficulties were not uniform, nor did they adhere to any specific pattern. Instead, the above challenges added difficulty to some tasks, in widely varying degree. Because most of these challenges have been met with the near-completion of iteration 1's original plan, the team believes that the existing estimations are valid for work going forward. New estimations have been made for the iteration 1 "leftovers," but a more general review of our approach to estimation will have to wait on the completion of an iteration more representative of the work still undone -- that is, an iteration without the once-new challenges listed above. Iteration 2 should provide this basis.

Fixes and Improvements - Iteration 2

The table below reflects plans for fixing bugs and improving code in some user stories from iteration 1. The associated unit costs are included in the totals for this iteration.

Team Members	Unit Cost	**Story in Bold**, Tasks indented below
--	**40**	**I want to create a character with distinct advantages and disadvantages.**
Karl, Tian	25	Retrieve information from server for selected class
Karl	15	Create the new character
--	**25**	**I can select another character from a list of characters.**
Glen, Tian	25	Retrieve collection of players from server
--	**15**	**With a character selected, I can quickly initiate a battle.**
Adam, Tian	15	Server sends battle notification to selected player
--	**35**	**In battle, I can see what my opponent is attempting and respond in real-time**
Glen, Tian	15	Receive opponent action data from server
Glen	10	Receive player input action
Glen,Tian	10	Send player action data to server

New for Iteration 2

Team Members	Unit Cost	**Story in Bold**, Tasks indented below
--	**155**	**As a new player I can create an account.**
Adam	70	Display the CreateAccount view
Adam	10	Ask the user the account name
Adam, Tian	15	Check to see if the name is taken
Adam	10	Ask user for password
Adam	15	Check password against requirements
Adam, Tian	15	Create account
Adam, Karl	5	Retrieve and link the account to the new installation
Adam	15	Ask for permission to use user's location
--	**70**	**As an existing player I can link my account to a new installation**.

Saung, Adam	30	Display UseAccount view
Saung, Adam	10	Ask the user for the account name and password
Saung, Tian	15	Check login credentials
Saung, Tian	10	Retrieve and link the account to the new installation
Adam, Saung	5	Ask for permission to use user's location
--	**35**	**After linking an account to my installation, the application should log in when I start it.**
Karl, Glen	20	Device sends login data to server
Karl, Glen	15	Verify login credentials
--	**50**	**I want my character data to be persistent.**
Karl, Tian	25	Android app writes to server - XML over TCP
Karl, Tian	25	Android app reads from server - XML over TCP
	75	**With another character selected, I can inspect that character.**
Saung, Tian	10	Send request for data to server
Saung, Tian	15	Server retrieves and sends data
Saung	50	Display up-to-date data for user
	75	**I want to view information on my character's status.**
Saung, Glen	10	Send request for data to server
Saung, Glen	15	Server retrieves and sends data
Saung	50	Display up-to-date data for user

Total number of units: 2805
Average units per iteration = 2805 / 4 = 702
Minimum units per iteration: 600
Maximum units per iteration: 800

Number of units chosen for current iteration = 675

Solution Alternatives and Trade-offs

Communication

Bluetooth can be used as an alternative communication method. Its advantage over **Wi-Fi** communication would be the filling of dead-spots in connectivity, and would not be reliant on **GPS** positioning to determine the location. Skipping the server middle-man would also speed up interactions. However, this makes it nearly impossible to have a map view of the players around you, as the range of bluetooth isn't significant in light of GPS resolution. In

addition, player-to-player interactions that skip interaction with the server would not have guarantees against client-side foul-play. We would also need to store more information on the clients such as damage formulas that would otherwise be stored on the server.

A possibility is to use bluetooth as a supplement to Wi-Fi and GPS. We decided against this as we are already using two types of communication and do not want to complicate the system any further, at least not until we have a large enough player-base that this would be deemed useful.

Server Interface

JSON can be used as an alternative interface to **XML**. Its advantages are that it is more compact, while providing the same level of information. The downside of JSON would be that we would likely need to build or otherwise include a JSON library into both the Android application and the server. In the interest of simplifying development, we will be using XML.

Player Authentication

We have the option of using the **phone**, **phone number**, **or sim card** (or something similar along that line of thought) as the method of identifying the player when communicating with the server. This has the immediate advantage of not needing the player to ever log on, and would make trying to steal accounts a fruitless effort. It also has the immediate downside of problems when the phone is lost or the phone number is changed.

Our implementation relies on a **login**- and **password**-based system. This design decision was made mainly because of the account-transferring problem. To solve it, we would need the players to make an account on a website or system of some sort, then give them the option to associate a phone number with it. Since the user would have to have a login and password anyway, and since it simplifies development, we decided to use accounts exclusively. We also keep the convenience of the phone or sim validation by storing the user account information on the application after the first time, so that users do not have to login every time they start the app.

Risks

Some risks exists for the application. The game requires periodic GPS location pinging, which may be taxing on battery life. There are many dead spots (e.g., inside buildings) where GPS does not work, so the player wouldn't be able to directly participate in battles even if the player wanted to because he/she would be in AI mode. The game also requires periodic server pinging, so if the server crashes or has some similar problems, the game would not function properly. It would be best for the data to be backed up somewhere. As for network speed, there needs to be a reasonable latency for enjoyable gameplay, and, similarly, the server needs to have a reasonable response time for the game to work well.

Security problems also exist. A player could attempt a denial-of-service attack by sending requests to the server over and over again until the server is overloaded and crashes. Log-in information such as passwords might be intercepted by a third party when they are sent back and forth between the server and mobile phone. Some players might try to hack the software to gain a competitive advantage.

Limitations

Because of our chosen Player Authentication system and our decision to automatically log players in when they launch ROLE, users will be limited to one game account per telephone number, at least for the first few iterations. GPS functionality and implementation of a Map view are delayed until a future iteration; instead this iteration will allow characters to be selected from a list of all characters, regardless of the corresponding players' locations. As a result, range considerations in battle interactions will be ignored for this iteration; no distinction may be made between ranged skills and melee skills.

These limitations will be addressed in future iterations.

Testing Plan

Analysis of Iteration 1

Many of the tests planned for iteration 1 were not implemented in the manner originally anticipated, for a variety of reasons. Our plan to use Positron, and Android story runner, for behavior testing that was to include UI finctionality was delayed. Positron's current release predates Android version 1.5 (Cupcake), and is no longer supported by its developer. We still hope to include Positron tests in our future testing suite, but inclusion of an unsupported framework in iteration 1, in addition to the several frameworks already necessary for iteration 1, was unrealistic. Going forward, where we identify still-functional Positron testing modules applicable to our project, we will ad them to our testing suite.

Similarly, our plan to use Android Monkey for acceptance and unit testing was ill-founded. Monkey allows testing by sending a pseudo-random series of events to an Android emulator instance, which, though useful for overall functionality testing, is not suited for the task-oriented nature of acceptance and user testing.

Lastly, JUnit testing has not proved to be very useful in testing client application components, which as yet largely consist of activities, or client-server interface components. Our implemented activities consist largely of UI elements, with a dearth of value-oriented methods. Because most data transactions and calculations are in the domain of the client-server interface or the server code itself, where the JUnit library is unhelpful, our testing suite for iteration 1 consisted of RSpec, detailed below, and non-automated simulator interaction.

For iteration 2 our testing strategy will remain largely the same, while we will look for new opportunities to automate acceptance and unit tests, and use Android Monkey for overall testing.

Server Testing:

The server backend will be developed using Ruby on Rails. The server will be tested using a Rails gem called RSpec, which is based upon many Java utilities including JUnit, JBehave, and JMock. RSpec is widely used by Rails developers. Because our server is mainly only backend, most of our tests will consist of unit tests that will mock inputs and test that the outputs are correct. Server acceptance tests, if we have more complicated processes that need

to be tested, will be tested using a user story runner called Cucumber. Cucumber will go through the set of pre-defined actions and make sure the outcomes match along the way. Although Rails has a behavior-based testing system called WebRat, a human-interface is not needed for our website (at least not yet), so it will be unnecessary.

Connection Testing:

Some simple cross-Android testing should be done to evaluate the communication as a whole. Although testing of input and output will be done on both the Android application and the server, additional testing based on the data sent and received by a second Android application would make our tests more solid.

This type of testing would be harder to automate than testing on a single device. The best solution would likely be to have a pre-programmed test suite on one Android applicaiton, then a comparison to expected values on the other Android's received data.

Acceptance and Unit Tests:

- **I want to create a character with distinct advantages and disadvantages**
    - [Acceptance Test] If I select "Assassin" in the SelectClass view on application, then I will see information about the class in an information box nearby ("The ninja is a master of stealth and cunning. [Beginner Stats Here]"). I will then see a button below that asks to confirm my selection. If I confirm my selection for "Assassin" by pressing the button in the SelectClass view, it will send the information in a packet to the server and create my character as an "Assassin" on the server. I will then be able to go into my Status display and see that I am an "Assassin" from the information taken from the server.
    - Ask the user to pick the character's class and display class-specific information.
        - [Unit Test - App] Classes should be selectable. Select a class, and the correct class is actually what is selected. If I select "Assassin" on view, the Assassin image and information is displayed
        - [Unit Test - App] After selecting a class, select another one. The first class is deselected and the correct class is now the newly selected one. If I select "Brawler" on view, the Brawler image and information is displayed.
    - Create the new character
        - [Unit Test - App] When Assassin has been confirmed, verify that the packet that is sent out contains Assassins selected.
        - [Unit Test - Server] When Assassin has been confirmed, check the newly created character and verify it is Assassin.
- **I can select another character from a list of characters.**
    - **[**Acceptance Test] I view a list of characters that are pre-defined, and they all appear on CharacterList view. I touch one, and the character is selected.
    - Retrieve collection of players from server
        - [Unit Test - App] Activate CharacterList view. A request for players should be sent out.

            - [Unit Test - Connection] Compare collection received from server to collection stored on server
- **With a character selected, I can quickly initiate a battle.**
    - [Acceptance Test] As a logged in player ("Tian"), with another player ("Adam") selected, I can initiate a battle by pressing the "Attack" button. This will send a request to battle to the server. The server should send the request notification to Adam, and he will see the battle notification screen. I will enter the battle screen and await responses from Adam.
    - [Acceptance Test] As a logged in player ("Adam"), when another player requests to battle me, I will receive a notification about the request and see the battle notification screen. If I press the accept button, I will enter the battle screen. If I ignore it or press the ignore option, I will go back to my regular view. The server will take over the fight and give responses back from a battle AI.
    - Server sends battle notification to selected player.
        - [Unit Test - Server] Mock a battle initiation. The server sends a notification to the selected player.
        - [Unit Test - App] Mock battle initiation input. The player receives an alert message that he or she is being challenged to a battle.
        - [Unit Test - Connection] Verify that when one side starts battle, the correct opposite side receives alert.
    - Selected player's device asks player to choose to battle or ignore.
        - [Unit Test - App] Mock battle initiation input. Show a battle initiation screen, and ask to accept or ignore.
        - [Unit Test - App] Mock accept. A packet should be sent out to server to accept battle. There should be a redirect to battle screen.
        - [Unit Test - Server] Mock "ignore". Server should register for AI mode (in our current case, nothing will happen).
        - [Unit Test - Server] Mock "accept". Server should now set both status in combat, and prepare to receive battle notifications.
    - Selected player's device displays battle view if appropriate.
        - [Unit Test] Mock battle accept. The correct view (Battle View) should be shown for the selected player with all of its relevant fields.
- **In battle, I can see what my opponent is attempting and respond in real-time.**
    - [Acceptance Test] If I ("Adam") select to attack, the other player ("Tian") will see a display that says that the other player is attacking.
    - [Acceptance Test] If another player ("Tian") selects to defend, I ("Adam") will see a display that says that the other player is defending.
    - Receive opponent action data from server
        - [Unit Test - App] Check that sent packet's action matches the action chosen.
        - [Unit Test - Server] Mock an action. Check that the received action matches the action that was sent in.
        - [Unit Test - Connection] Double-check that the action received matches the server's action.
    - Display opponent action in battle view
        - [Unit Test - App] Mock received action. Check that action shown matches mocked action.

    - Receive player input action
        - [Unit Test - App] Press the attack button. Make sure packet formed contains attack action. Similar for rest of commands.
    - Send player action data to server
        - [Unit Test - App] Mock attacking. Make sure packet sent out is the attack action. Same for rest of commands.
        - [Unit Test - Server] Mock reception of attack button. Make sure server does corresponding attack internally on server. Similar for rest of commands.
        - [Unit Test] Check that the action received on server matches action chosen
- **I want to view information on my character's status.**
    - [Acceptance Test] If I click on the status button, I should be brought to the Status View, which should show information about my character. Such information includes character name, character class, stats, experience, level, skills, and health.
    - [Acceptance Test] After looking at the character status, if I start a battle and look at the character status again, the information shown should be updated.
    - Send request for data to server
        - [Unit Test - App] Send RSpec request to server, confirm response.
    - Server retrieves and sends data
        - [Unit Test - App] Mock response from server and validate UI behavior
    - Display up-to-date data for user
        - [Unit Test - App] Select the option to view player status. The correct Status View should be shown with all of its relevant fields.
        - [Unit Test - App] Participate in a battle and select the Status View again. The character information should be updated to reflect changes that occurred in the battle.
- **With another character selected, I can inspect that character.**
    - [Acceptance Test] If I select another player on the player list view, I should be brought to a new view that shows public information about that player. Such information includes character name, character class, stats, experience, level, and health.
    - Send request for data to server
        - [Unit Test - App] Send RSpec request to server, confirm response.
    - Server retrieves and sends data
        - [Unit Test - App] Mock response from server and validate UI behavior
    - Display up-to-date data for user
        - [Unit Test - App] Select a player from the list view. The correct inspect view should be shown with all of its relevant fields about the selected player.
        - [Unit Test - App] Inspect the player. Fight the player and inspect him/her again. The information should be different and should reflect changes that occurred in the battle.
- **As a new player I can create an account.**

    - [Acceptance Test] If I install the app for the first time and open it, I should have the choice to create a new account that will store all my player information.
    - [Acceptance Test] If I select to create a new account, a new view should display where I can input relevant data such as account name and password and confirm my input. I should be brought into the actual game after creating the account.
    - Display the CreateAccount view
        - [Unit Test - App] Select the create account option. Make sure the correct view (CreateAccount View) is shown with all of its relevant fields. The view should include save and cancel buttons.
    - Ask the user the account name
        - [Unit Test - App] Input text into the textbox. The text shown should reflect the input characters.
    - Check to see if the name is taken
        - [Unit Test - App] Create an account named "wootcakes." Reinstall the app or install the app to another phone and try to create another account with the same name. An error message should appear saying "This name has already been taken. Please selected a different one."
    - Ask user for password
        - [Unit Test - App] Input text into the textbox. The text should show the same number of asterisks (*) as the number of input characters.
    - Check password against requirements
        - [Unit Test - App] Passwords must be at least six characters long. Try to create an account with a password 5 characters long. An error message should appear saying "Passwords must be at least 6 characters long."
    - Create account
        - [Unit Test] Create an account. Lookup that account in the server database and it should be found with the correct information specified by the player.
    - Retrieve and link the account to the new installation
        - [Unit Test - Server] Send RSpec login request to server to authenticate login. Confirm account linkage.
    - Ask for permission to use user's location
        - [Unit Test - App] Compare android manifest to expected text.
- **As an existing player I can link my account to a new installation.**
    - [Acceptance Test] If I reinstall the app or install the app onto another phone, I should have the option to link my existing account to the new installation.
    - [Acceptance Test] If I select to link an existing account, I should be brought to a new view in which I can input my login data such as account name and password. If I confirm my input my account should be linked to the app and I should be brought to the main view of the game.
    - Display UseAccount view
        - [Unit Test - App] Select the link to an existing account option. Make sure the correct view (UseAccount View) is shown with all of its relevant fields. The view should include save and cancel buttons.
    - Ask the user for the account name and password

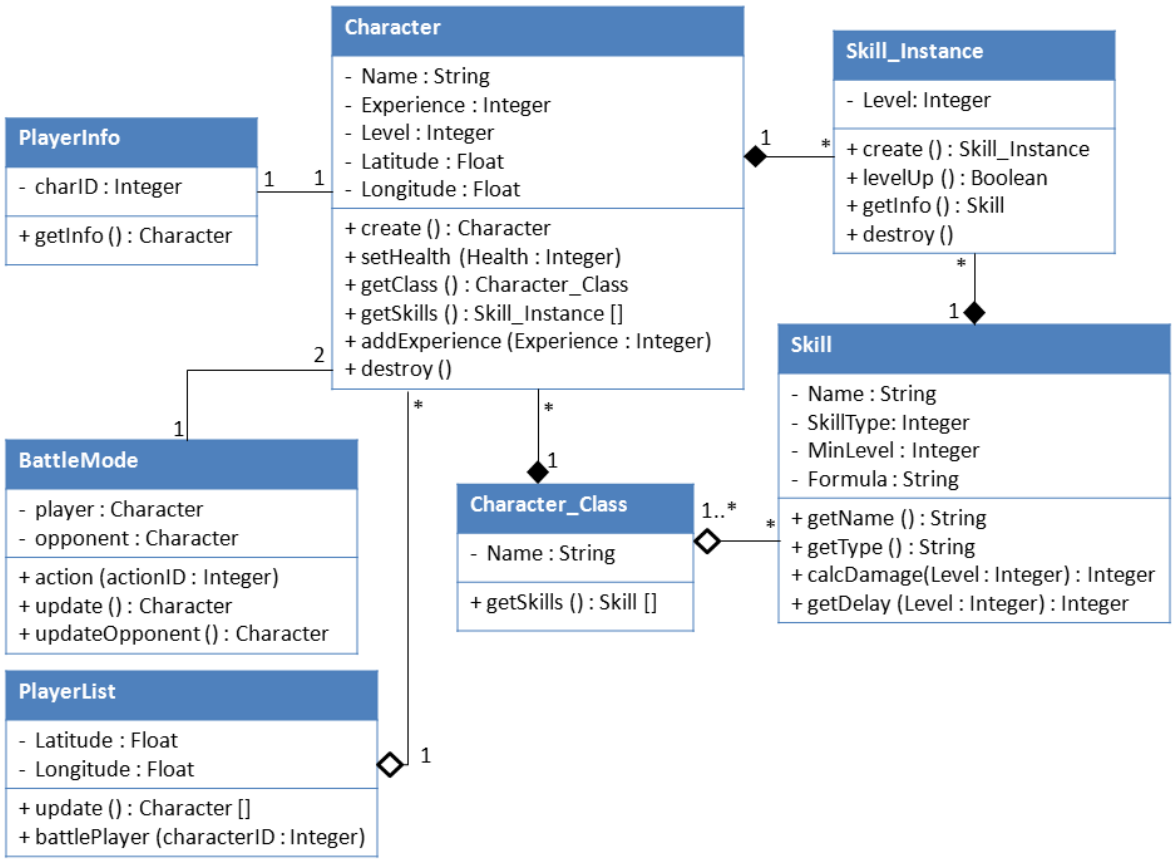
            - [Unit Test - App] Input text into the name and password fields. The text displayed should reflect what the player inputted, except that the password will be the same number of asterisks as there are input characters.
        - Check login credentials
            - [Unit Test - App] Enter an existing account name with the wrong password. An error message should occur mentioning that the password is incorrect.
            - [Unit Test - App] Enter a non-existing account name and any password. An error message should occur mentioning that the account name is non-existing.
        - Retrieve and link the account to the new installation
            - [Unit Test - Server] Send RSpec login request to server to authenticate login. Confirm account linkage.
        - Ask for permission to use user's location
            - [Unit Test - App] Compare android manifest to expected text.
- **After linking an account to my installation, the application should log in when I start it.**
    - [Acceptance Test] After linking an account to the app, if I close and open the app again I should automatically be logged in and be brought to the main view of the game.
    - Device sends login data to server
        - [Unit Test - App] Send RSpec request to server to authenticate login. Confirm character login status.
    - Verify login credentials
        - See Check login credentials for "As an existing player I can link my account to a new installation"
- **Uncategorized tests**
    - Client-side unit tests for sending requests and attached data to server
        - [Unit Test] Call sendRequest on a ServerLink object with a params object. It should translate the params object into the appropriate XML.
        - [Unit Test] Call sendRequest on a ServerLink object. It should create an HTTP connection requesting the appropriate address with the appropriate data.
    - Client-side unit tests for receiving response data from server
        - [Unit Test] Call sendRequest on a ServerLink object. Given a correct response from the server to the proper request, it should translate the received XML to the appropriate object.
        - [Unit Test] Call sendRequest on a ServerLink object. Given a correct response from the server to the proper request, it should call its registered component's callback function.

System Architecture

ROLE is essentially comprised of two elements: the device application running on users' mobile phones, and the server back-end. The device application and the server both employ the Model-View-Controller architecture. The interface between the two, quite naturally, is of a client-server nature.

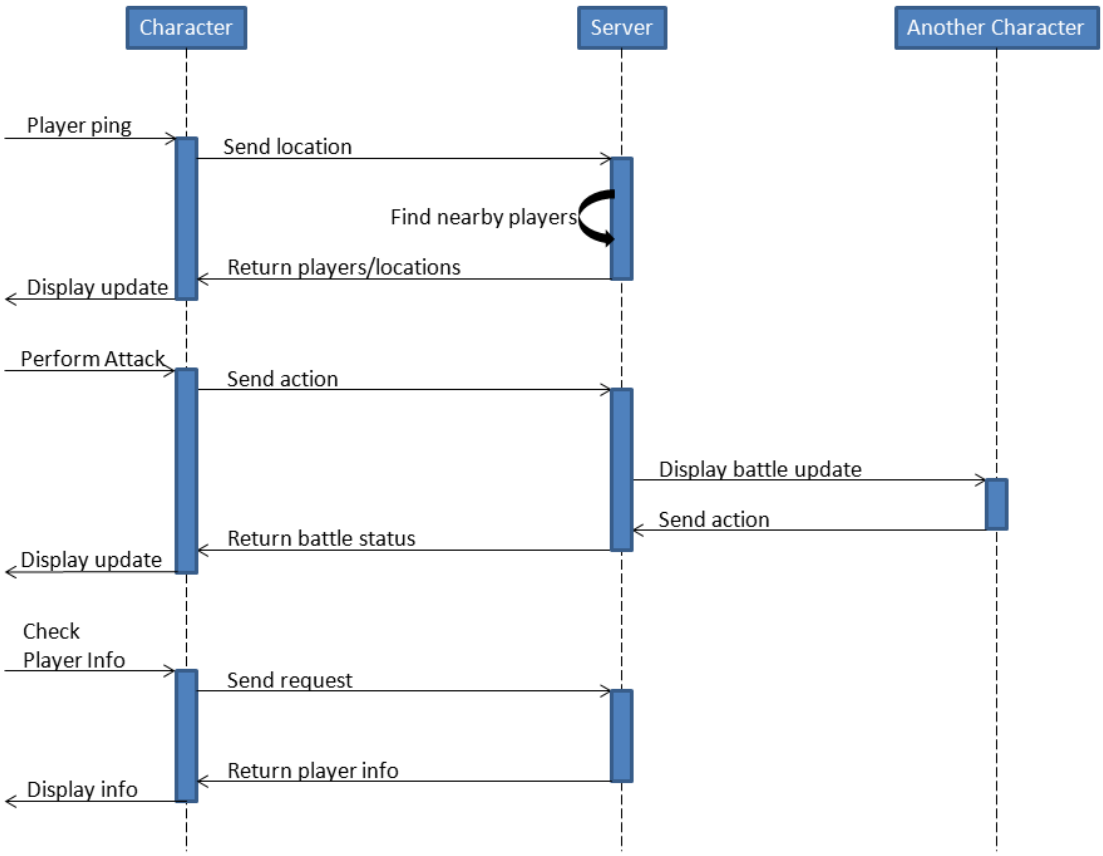
Although the Android client is sending the requests to the server, the application itself is event-driven. In response to requests from the client, the server will occasionally send back "events" such as other players challenging the client's player. The server itself--as is the general nature of servers--runs in an event-driven style, only changing its state when queried by a client.

Server-Side Class Diagram:



Much of the information being requested is presentable in the form of Character objects. A character possesses both a Class and Skill_Instances, which reference Skill objects.

Server-Communication Sequence Diagram:

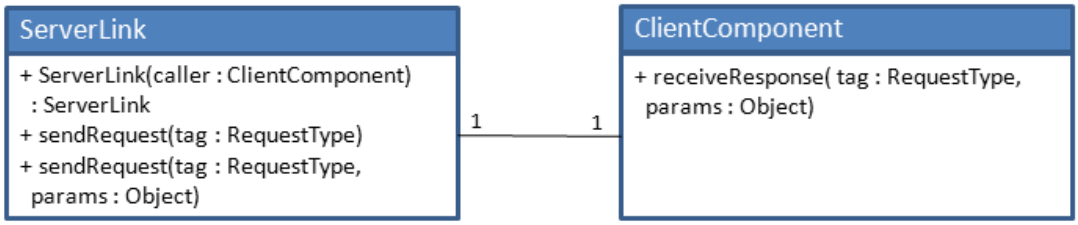


The server can be seen as being always waiting for requests. The clients never directly communicate with each other, instead information they send changes the state of the server, which is then funneled to the clients the next time they ping.

Client Component / ServerLink Interface

Components of the client application communicate with the server using an instance of the ServerLink class, to whose constructor they pass a reference to themselves. All such client components implement the ClientComponent java interface, which specifies one method to facilitate the response from the server: receiveResponse(tag: RequestType, params: Object).

Server communication API Class Diagram:



XML Interfaces:

The interfaces with the server will be accomplished using get/put requests on RESTful resources. This information will be formatted using XML. Different interfaces will be needed for different types of requests and updates.

On the Android side, we plan to construct a handler for processing and sending XML requests. This will take care of timing and using the correct URL and the such, so that the rest of the application does not have to deal with it.

The XML interfaces have been overhauled from iteration 1. The previous iteration planned to send/receive most information via the location updating interface. This made the interface very complicated, and thus we decided to split off the interface for different actions.

There are currently planned interfaces for character creation/inspection, location sending/receiving, and battle sending/receiving.

Send (Location) Update Interface: => /androids/update

```
<?xml version="1.0" encoding="utf-8"?>
<update>
  <client>[integer: client version]</client>
  <mode>[boolean: active/passive]</mode>
  <location>
    <latitude>[float]</latitude>
    <longitude>[float]</longitude>
  </location>
</update>
```

Receive (Location) Updates Interface: => /androids/fetch

```
<?xml version="1.0" encoding="utf-8"?>
<updates>
  <!-- Updates on current player's metrics -->
  <health>[integer]</health>
  <experience>[integer]</experience>
  <level>[integer]</level>
```

```
  <!-- Locations of "nearby" players -->
  <locations>
    <character>
      <id>[integer: character id]</id>
      <name>[string: character name</name>
      <location>
        <latitude>[float]</latitude>
        <longitude>[float]</longitude>
      </location>
    </character>
  </locations>
  <!-- A notification of a battle request -->
  <battle>
    <id>[integer: battle id]</id>
    <initiator>[integer: character id]</initiator>
  </battle>
</updates>
```

Battle Send Interface: => /battles/update

```
<?xml version="1.0" encoding="utf-8"?>
<update>
  <opponent>[integer: opponent id]</opponent>
  <action>
    <type>[string: initiate/accept/reject/fight]</type>
    <!-- Skill you are employing if you choose to fight -->
    <ability>[integer]</ability>
  </action>
</update>
```

Battle Receive Interface: => /battles/fetch

```
<?xml version="1.0" encoding="utf-8"?>
<battle>
  <status>[string: initiated/started/rejected/won/lost]</status>
  <!-- Experience only sent when battle status is ended -->
  <experience>[integer: experience gained/lost]</experience>
  <player>
    <health>[integer]</health>
    <action>
      <type>[string: initiate/accept/reject/fight]</type>
      <ability>[integer: ability id]</ability>
      <status>[string: pending/blocked/complete]</status>
      <effect>[integer]</effect>
      <delay>[integer]</delay>
    </action>
  </player>
  <opponent>
    <id>[integer: opponent id]</id>
```

```
    <health>[integer]</health>
    <action>
      <type>[string: initiate/accept/reject/fight]</type>
      <ability>[integer: ability id]</ability>
      <status>[string: pending/blocked/complete]</status>
      <effect>[integer]</effect>
      <delay>[integer]</delay>
    </action>
  </opponent>
</battle>
```

Character Creation Interface: => /characters/create

```
<?xml version="1.0" encoding="utf-8"?>
<character>
  <name>[string: character name]</name>
  <class>[string: class name]</class>
</character>
```

Character Inspection Interface: => /characters/1

```
<!-- /character/1 -->
<?xml version="1.0" encoding="utf-8"?>
<character>
  <id>[integer: character id]</id>
  <status>[string: active/passive/battle]</status>
  <name>[string]</name>
  <level>[integer]</level>
  <class>[string: class name]</class>
  <health>[integer]</health>
  <location>
    <latitude>[float]</latitude>
    <longitude>[float]</longitude>
  </location>

  <!-- The following are only shown if you inspect yourself -->
  <experience>[integer: experience]</experience>
  <abilities>
    <ability>
      <id>[integer: ability id]</id>
      <name>[string: skill name]</name>
      <level>[integer: ability level]</level>
      <delay>[integer: skill delay]</level>
    </ability>
  </abilities>
</character>
```