ROLE

Design and Planning Document
02/23/10, Version 1.0
Adam Clay, Karl He, Glen Kim, Saung Li, and Tian Wang

Meta-specifications: http://inst.eecs.berkeley.edu/~cs169/sp10/doku.php?id=proj3

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*	First Iteration?
155		**As a new player I can create an account.**	
	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**.	
	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 a account to my installation, the application should log in when I start it.**	
	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.**	✓
	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.**	✓
	25	Android app writes to server - XML over TCP	
	25	Android app reads from server - XML over TCP	
	125	Creation of the server - Rails	
		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.**	✓

	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.**	
	10	Send request for data to server	
	15	Server retrieves and sends data	
	50	Display data for user	
180		**With a character selected, I can quickly initiate a battle.**	✓
	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**	✓
	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		**Players want to view information on their character's status.**	
	10	Send request for data to server	
	10	Server retrieves and sends data	
	50	Display 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	

Total number of units: 2690
Minimum units per iteration = 2690 / 4 = 672
Maximum units per iteration: 800

Number of units chosen for first iteration = 765

Allocation - Iteration 1

For each of the tasks allocated to a member, that member will write test cases for that task before carrying out the task.

For this first iteration, one member, who is most experienced with Ruby, will focus on developing the server backend, setting it up to accept notifications and save data sent by the Android app, as well as allowing it to send information back to the Android app. This will allow for character data to be persistent, which is one of the user stories. What information is sent back and forth between the app and server depends on the tasks, listed above, that need to be done by the app. The app assumes that the server reliably saves data and that the app will be able to retrieve the data whenever requested.

The user interface for the SelectClass view, character information view, and Battle view are quite modular so they can each be designed and developed independently by one of the members. Ultimately, all user stories are dependent on server functionality. Accordingly, a team member has been allocated tasks comprising the Character Data Persistence user story (which represents the core of our server implementation) as well as participation in server-related tasks in other user stories.

Team Members	**Unit Cost**	**Story in Bold**, Tasks indented below

--	**205**	**I want to create a character with distinct advantages and disadvantages.**
Saung	70	Display the SelectClass view
Saung	10	Ask the user to pick the character's class
Saung, Tian	100	Display information for the class selected
Saung	10	Ask user to confirm selection or make another selection
Karl	15	Create the new character
--	**175**	**I want my character data to be persistent.**
Karl	25	Android app writes to server - XML over TCP
Karl	25	Android app reads from server - XML over TCP
Karl, Tian	125	Creation of the server - Rails
--	**70**	**I can select another character from a list of characters.**
Glen, Tian	15	Retrieve collection of players from server
Glen	25	Display CharacterList view
Glen	25	Process touch coordinates
Glen	5	Store selection
--	**180**	**With a character selected, I can quickly initiate a battle.**
Adam	100	Display battle view
Adam, Tian	10	Notify server of battle status
Adam, Tian	15	Server sends battle notification to selected player
Adam	45	Selected player's device asks player to choose to battle or ignore
Adam	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**
Glen, Tian	15	Receive opponent action data from server
Glen	100	Display opponent action in battle view
Glen	10	Receive player input action
Glen,Tian	10	Send player action data to server

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.

There is also an argument to be made about how many interfaces are needed. We have currently designed for a periodic update interface which deals with the player status, locations, and battles. The reason is that these are all things that need to be updated regularly. The rest of the interfaces will be split off, such as the player information query interface.

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

In this section, provide a brief description of how you plan to test the system. This section lays the foundation for test-driven development, an extreme programming practice we expect you to use. Each *user story* selected for the iteration should be translated into an acceptance test. A user story has to pass its acceptance test at the end of the iteration. **Also describe how acceptance testing can be automated so as to maximize the limited number of human hours you will have for testing your system** (e.g., write a function that will input test data to your code and check the result rather than entering the test data interactively). You should also produce a unit test for each *task*. Tests should be written before implementation of the corresponding features begins. The purpose of this section is a reality check that your design will in fact be testable with a reasonable effort. **You should specifically discuss design decisions that affect testing and describe any test interfaces built into the system.**

Android Application:

To test our android application, we will write a suite of unit tests and acceptance tests using Monkey and Positron (http://code.google.com/p/autoandroid/), an Android story runner, for behavior-based testing. Positron will run a set of pre-defined test cases and perform the necessary clicks and other actions, and check that the view is displayed correctly. Communication to the server is easily testable by using Positron to view the packets that will be sent or received and making sure those are correct against predetermined values. GPS-testing will receive faked input at first, but may require manual testing in the field to ensure that our application works. When testing other components dependent on GPS, the input will be mocked.

Server:

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.
    - Display the SelectClass view.
        - [Unit Test - App] Select the option to create a character. The correct view (SelectClass) is shown with all of its relevant fields. There should be an option for each class available, and it should be rendered with either a menu for all classes or a list of select buttons for each class.
    - Ask the user to pick the character's class
        - [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 internal ClassSelected should be "Assassin"
        - [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 internal ClassSelected should now be "Brawler."
    - Display information for the class selected
        - [Unit Test - App] A view with the information is displayed. The information needs to be of the class that is selected. If "Assassin" is mocked, the information requested from server and received back should be Assassin information ("The assassin is a master of stealth and cunning.") and it should be displayed in the information box.
        - [Unit Test - Server] Mock a request for assassin information. The information sent from server should be Assassin information ("The assassin is a master of stealth and cunning.").
    - Ask user to confirm selection or make another selection
        - [Unit Test - App] Mock the user clicks accept. The value passed out should include that the class selected is "Assassin"
        - [Unit Test - App] Mock the user does not confirm. Nothing should be sent from the app to server.
    - Create the new character
        - [Unit Test - App] Mock that Assassin has been confirmed. Verify that the packet that is sent out contains Assassins selected.
        - [Unit Test - Server] Mock that a packet for character creation with Assassin as class has been received on server end. Check the newly created character and verify it is Assassin.
- **I want my character data to be persistent.**
    - [Acceptance Test] I log off my character, and when I come back on my stats are the same as before I logged off in the Status display.
    - [Acceptance Test] I press the home button, and when I come back on my stats are the same as before I pressed the home button in the Status display.
    - [Acceptance Test] I lose internet connection, and when I come back on my stats are the same as before I lost internet connection in the Status display.
    - Android app writes to server
        - [Unit Test - App] Activate a send function on app. Make sure the packet sent out matches a pre-defined correct packet.
        - [Unit Test - Server] Mock a packet received. Make sure the server saves that packet correctly, in that values stored match packet mocked.
        - [Unit Test - Connection] App sends data to the server, and the server saves these values for future retrieval. The app will then read values from server, and the result should be same as data sent out.
    - Android app reads from server
        - [Unit Test - App] Mock input being sent to app. Make sure app takes in the input correct and that the values taken in by app match the pre-defined input sent in.
        - [Unit Test - Server] Mock request for information sent to server. Make sure server returns the correct information regarding the correct player.
        - [Unit Test - Server] Mock request for information sent to server for non-existent player. Make sure server returns an error message.

        - [Unit Test - Connection] App sends request to the server, and the server returns a pre-defined value. The app will then pull values from server, and the result should be same as data sent out.
    - Creation of the server
        - [Unit Test] For each model in the server, we will have RSpec tests that ensure integrity of our data.
        - [Unit Test - Server] Classes should contain a ClassName.
        - [Unit Test - Server] Classes should contain stats.
- **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 - Server] Mock a player list request. All players should be included in packet sent out.
        - [Unit Test - Connection] Compare collection received from server to collection stored on server
    - Display CharacterList view
        - [Unit Test - App] Mock a list of players. Ensure that each player is present in the list on the view.
    - Process touch coordinates
        - [Unit Test - App] Touch a list entry and ensue that the corresponding character is selected
    - Store selection
        - [Unit Test - App] Touch a list entry and ensure character variable references corresponding character
- **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.
    - Display battle view.
        - [Unit Test - App] Press attack character option. Make sure correct view (Battle View) is shown with all of its relevant fields. Make sure combat opponent is same as character selected.
    - Notify server of battle status.
        - [Unit Test - App] Make sure information that the user is starting a battle along with information about the selected target character is sent to the server.

        - [Unit Test - Server] Mock a battle initiation. The server reads this information and looks up the selected player in its database.
    - 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

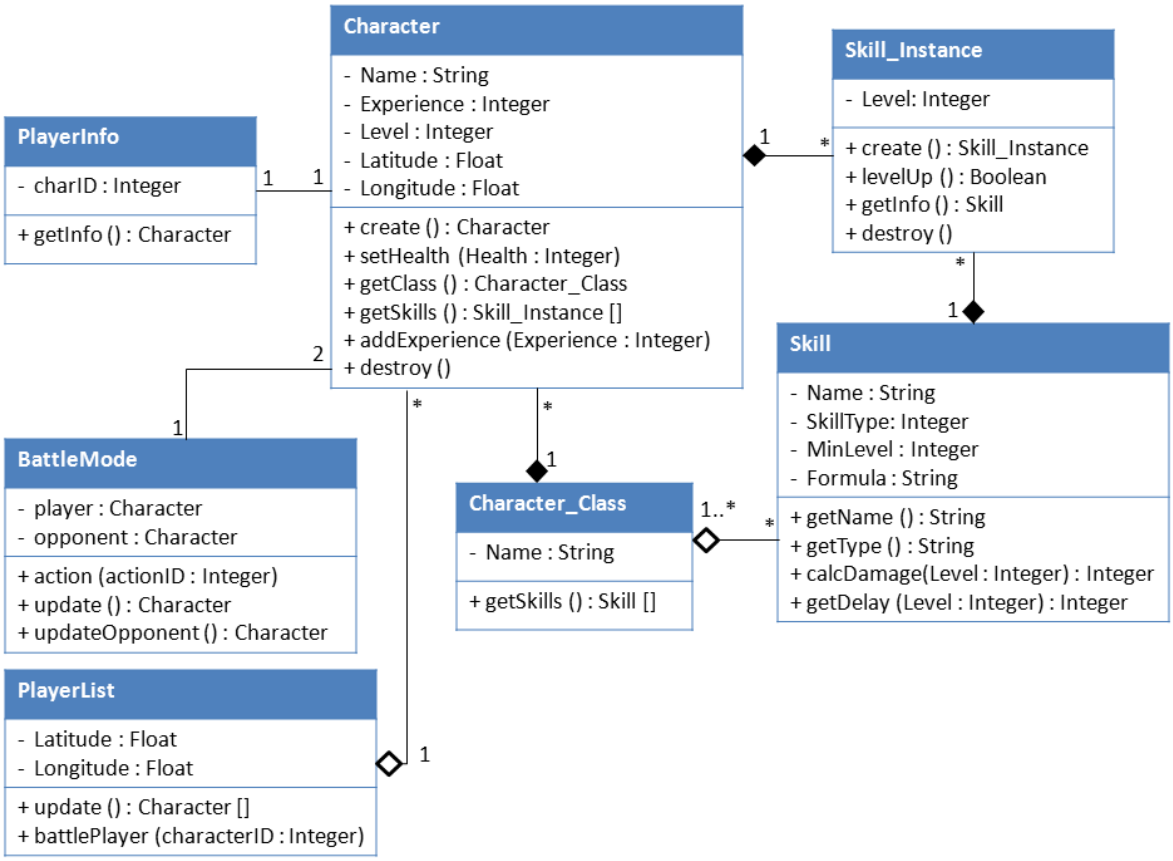
System Architecture

Here you should describe the high-level architecture of your system: the major pieces and how they fit together. Use graphical notations as much as possible in preference to English sentences. UML class diagrams (to describe system components at the class level) and sequence diagrams (to describe interactions between the components) should be included in this section. Try to use standard architectural elements (e.g., pipe-and-filter, client-server, event-based).

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.

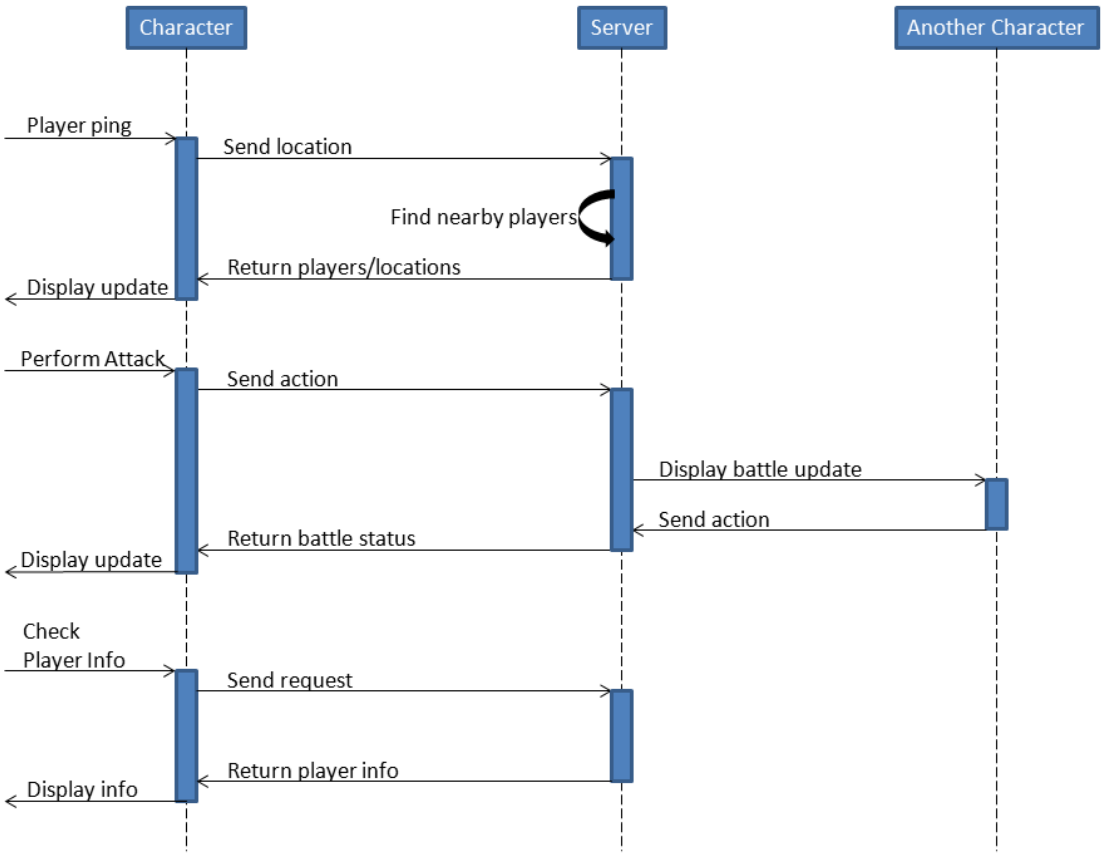
Although the Android client is sending the requests to the server, the application itself is event-driven. In response to requests from the client, the server will occasionally send back "events" such as other players challenging the client's player. The server itself--as is the general nature of servers--runs 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.

XML Interfaces:

The interfaces with the server will be accomplished using get/put requests on a RESTful resource. This information will be formatted using XML. On the Android side, this can be facilitated via the java.net.URLConnection class (alternative, we can use the HTTPClient class). Different interfaces will be needed for different types of requests and updates.

Send Update Interface:

```
<?xml version="1.0" encoding="utf-8"?>
<update
    version="[integer]"
    packetID="[integer]"
    characterID="[integer]">
  <mode>[boolean: active/passive]</mode>
  <status>[string: idle/battle/dead]</status>
  <!-- Updated location of the player -->
  <location>
    <latitude>[float]</latitude>
    <longitude>[float]</longitude>
  </location>
  <!-- This section will be missing if no battle is occurring -->
```

```
  <battle>
    <!-- ID of player you are battling with -->
    <opponent>[integer]</opponent>
    <!-- Used if the battle is being initiated -->
    <initiate>[string: initiate/reject]</initiate>
    <!-- Skill you are employing if battle is in progress -->
    <action>[integer]</action>
  </battle>
</update>
```

Receive Updates Interface:

```
<?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]">
      <name>[string]</name>
      <location>
        <latitude>[float]</latitude>
        <longitude>[float]</longitude>
      </location>
    </character>
  </locations>
  <battle>
    <player>
      <health>[integer]</health>
      <!-- The effects of your action -->
      <action>
        <id>[integer]</id>
        <status>[string: pending/blocked/complete]</status>
        <effect>[integer]</effect>
        <delay>[integer]</delay>
      </action>
    </player>
    <opponent id="[integer]">
      <health>[integer]</health>
      <!-- The opponent's action -->
      <action>
        <id>[integer]</id>
        <status>[string: pending/blocked/complete]</status>
        <effect>[integer]</effect>
        <delay>[integer]</delay>
      </action>
    </opponent>
  </battle>
</updates>
```

Inspection Interface:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Querying a character's information -->
<character id="[character id]">
  <name>[string]</name>
  <level>[integer]</level>
  <class>[integer]</level>
  <health>[integer]</health>
  <location>
    <latitude>[float]</latitude>
    <longitude>[float]</longitude>
  </location>
</character>
```