Settlers of Catan® Phase 1

Objective

In this phase you will design, implement, and test the following subsystems:
1. Catan Model
2. Server Proxy
3. Server Poller

Catan Model

Design

The Catan Model will be at the heart of both your client and server designs. The types in this subsystem should model the core domain concepts of the Catan game (players, games, map, resources, development cards, etc.). These types should encapsulate all of the core data structures and algorithms for the system. Many of your model classes will be derived from the conceptual model created in Phase 0. However, additional implementation-oriented classes that are not part of the conceptual model will also be needed. For example, your design might need a UserManager class that manages the collection of all User objects, and provides operations for creating, enumerating, and authenticating users. While the User class appears in the conceptual model, the UserManager class probably does not.

The Java code provided on the course web site already contains a number of data types that you will need in your design. Review the code in the shared.definitions and shared.locations packages so that you don’t waste time creating data types that are already provided.

Make sure that your model classes support all of the necessary operations required by the program. Thoroughly analyze the Functional Specification, and make a list of the data and operations needed to support the program’s functionality. For example, what kind of information will the Map need to store? What operations will need to be performed on that data? Perform such an analysis for all parts of the system. Examples of necessary operations are: createUser, authenticateUser, listGames, createGame, placeRoad, placeCity, etc.

The classes in your model should be inherent to the application domain, and independent of the particular user interface we are using. That is, if the user interface were to be substantially redesigned, the classes in the model should be reusable in implementing the new interface. It is inappropriate to include user interface notions in the model.
However, your model should provide operations that the user interface can call to determine what operations the user is currently allowed to perform. This will allow the user interface to enable or disable user actions based on what is currently allowed. Examples of such operations are: canBuyRoad, canPlaceRoadAtLoc, canPlayDevCard, canAcceptTrade, etc. We will refer to such operations as the “can do?” operations.

Your model will also need the ability to initialize itself from the JSON data returned by the server from the /game/model web service API. The format of this JSON data is described in the document titled “Client Model JSON Documentation” on the course web site. You are encouraged to use the Gson open-source library for converting Java objects to JSON format, and vice versa.

Your model classes should actively reject invalid operations and data by throwing exceptions. In other words, methods should throw exceptions when their preconditions are violated.

**Implementation and Testing**

Write some JUnit tests for the “can do?” methods in your model.

Write some JUnit tests verifying that you can successfully initialize your model from the JSON model data returned by the server.

**NOTE:** For this phase you do not need to implement the “longest road” and “largest army” algorithms in your model. These will come later. For now, all your model needs to do is store which players have the “longest road” and “largest army”, and allow this information to be queried.

**Server Proxy**

**Design**

Design a Server Proxy interface that the client can use to communicate with the server. This interface should provide methods for invoking all of the server’s web service APIs. (The web service API is described in the server’s Swagger page see the README.txt file with the downloaded source code to know how to get to this page, and also the document titled “Server Web API Documentation” on the course web site). It is also a good idea to look at the Functional Spec on the website so you can see how the GUI is designed to work.
Provide two concrete implementations of your Server Proxy interface. The first implementation should be “real” in that it actually calls the server and returns any results from the server. The second implementation should be a “mock” implementation that does not actually call the server, but instead returns canned, hard-coded results. (The “mock” implementation will be used to do local unit testing of the Server Poller, which is described later.) The canned results returned by the “mock” proxy can be created by going to the server’s Swagger page, calling the server methods, and copy-and-pasting the output from the Swagger page into your testing code.

Your “real” server proxy implementation should use Java’s HttpURLConnection class to send HTTP requests to the server, and to receive HTTP responses in return. It should also manage the HTTP cookies that are returned by the server to track the local player’s identity and which game they are participating in (i.e., the catan.user and catan.game cookies). These cookies are described in the document titled “How the Catan Server Uses HTTP Cookies”. When the proxy calls the server’s /user/login method, the server will return a value for the catan.user cookie which contains the player’s identity. Your proxy should extract the value of this cookie from the HTTP response’s “Set-cookie” headers, and cache it for later use. Thereafter, whenever the proxy calls a method on the server, it should include the catan.user cookie in the HTTP request by setting the “Cookie” header in the HTTP request. The server will read this Cookie header to identify the calling player. Similarly, when the proxy calls the /games/join method, the server response will include a value for the catan.game cookie. The proxy should also cache this cookie’s value, and include it on all subsequent calls to the server. The server will read this cookie to determine which game the caller is participating in.

**Implementation and Testing**

Write some JUnit tests to verify that your Server Proxy is able to successfully communicate with the server. Test that each of the proxy methods can successfully connect to the server, make its request, and receive any results. For these tests you are only trying to verify the communication between proxy and server, not the functionality of the server itself.

NOTE: There are several methods supported by the server that are only for testing and debugging purposes. These methods will never be called through your server proxy, and will only ever be called through the server’s Swagger page. Therefore, you need not implement or test these methods on your server proxy. These methods are:

- /games/save
- /games/load
- /game/reset
- /game/commands [GET]
- /game/commands [POST]
You should have tests for each move endpoint that get a 200 response from the server using your real Server Proxy. In order to get a 200, all you have to do is give the server a properly formatted request. The server will not check to make sure it is a valid move.

Example: If you send the server a soldier request and that player doesn’t technically have a soldier, the server doesn’t care and will return a 200 as long as the request is formatted properly.

**Server Poller**

*Design*
Create a Server Poller that is responsible for:
1. Polling the server at regular intervals (every few seconds) to download the current model state
2. Updating the state of the client’s model with the JSON data returned by the server

The Server Poller should be fairly simple to design and implement. It should call the Server Proxy to retrieve the current model state, and then update the local model’s state with the returned JSON data.

Design your Server Poller to use *dependency injection* so that it can be easily configured to use either the “real” or “mock” Server Proxy implementation. This means that rather than calling “new” internally to create a proxy object, the poller should instead have a constructor parameter or setter that can be used to pass in the proxy it should use.

*Implementation and Testing*
Write some JUnit tests to verify that your Server Poller is working properly. Use the “mock” Server Proxy instead of the “real” one so you can run these tests without needing to run the server. (The tests for your Catan model should already contain some code to verify that the model can properly initialize itself from JSON data. You should be able to reuse that code here to ensure that the model has been properly updated by the poller.)

*Deliverables*

*Design*
Your design should include the following documentation:
1. UML class diagrams for your Catan Model, Server Proxy, and Server Poller. These diagrams are intended to provide a high-level overview of your design, and need not contain a lot of attributes and operations. Details about each class's method interface will be provided in the Javadocs.


All team members should help create both the UML diagrams and Javadocs. Do not underestimate the amount of work required to create the Javadocs for this phase. Since many new classes will be created in this phase, substantial work is required to create the Javadocs. Therefore, all team members should help.

Make sure that all team members use consistent naming conventions for packages, classes, and methods.

*Implementation and Testing*

1. Fully implement your Catan Model, Server Proxy, and Server Poller.

2. Implement JUnit tests cases for your Catan Model, Server Proxy, and Server Poller (as previously described). Your test cases should successfully compile and run.

3. Write an ANT target named “test” that will compile and run your JUnit test cases.

4. Check all source files into Git.

5. At the end of the phase, submit a zip file containing your source tree to the TAs.