Settlers of Catan® Phase 2: Implementation Notes

1. Become very familiar with the JSON Model that comes from the server. You will need a way to translate this into your own Java GameModel.

2. Get server running, log in to one of the existing games.

3. The default game would be the place to start to get things going, because it bypasses the initial setup stage.

4. The controllers have everything they need passed in to them through their constructors. You can add in other references as needed to suit your design. The Base Controller class will save the main view and the client model when you call the superclass constructor. You can access them by calling this.getView() and this.getClientModel(). Anything else needs to be saved within the child constructor.

5. Each view is either a normal view or a modal view. When you want to show a modal, call showModal() on the view object, and when you want to close it, call closeModal(). You don’t really want to try and have multiple modals up at once, or the page will freak out a little.

6. The methods the controllers need to call on the views are mostly documented through comments in the controllers. We tried name the methods reasonably so it is clear what they do.

   1. Often a string needs to be passed to the view method so it knows which object to hide/show/update. Passing in catan.definitions.CatanColor.Red or catan.definitions.ResourceTypes.Wood, the views will know what to do for the accompanying part.

      If you pass in an invalid argument (like trying to update a “road” part for discarding resources, which isn’t possible), the views usually throw an error because it’s trying to access an undefined object, but sometimes will silently fail, especially if you pass in an undefined Boolean because of the lack of type-checking.

7. The first thing you will want to work on is populating the views from a model. With only a little added code you should be able join a game and then request the game Model from the server. This is done through the start methods in each of the controllers. It is difficult add functionality until this is done.

8. You have to be able to roll to start your turn at all. We have an automatic timer (which the students will have to implement) for rolling, as well as the rolling button. Generate the dice roll in the controller (add the results of 2 random number generators so it simulates the real world) and send that to the server. Once the command is sent, close the modal
and display the “result modal” with the roll number set. Clicking “okay” will call a method that should allow the game to proceed.

9. The next thing you’ll probably want to get working is the turn tracker, which is in charge of saying who’s turn it is and the colored “finish turn” button. It’s pretty simple, and that will allow you to start having some control over the game. While you’re at it, the points controller is trivial: just set the total player points, and bring up the endgame overlay if there is a winner (our clientModel stores the winner Index).

10. Once you can roll and rotate turns, discarding is probably next, since rolling a seven could require you to discard cards no matter whose turn it is. We use a ResourceList object to store the cards, and send that to the server. Just increase the number of cards in the hand any time a button is clicked. When you have the exact amount of cards you need to discard, then enable to button and hide all the “increase” buttons. Also, make sure you check whether a person has discarded yet this turn before bringing up the modal—you only need to discard once per turn.

11. The chat and log are probably the next easiest thing to implement. These are found in the “communication” folder. The only difference between them is that the chat view has an input element, so there is a function on the chat controller for adding a line, which is just a matter of sending the command. We’ve set this up so that you pass in an array of objects that represent a line, and the view will update. This array is supposed to contain all the messages to log (we don’t have an incremental way of doing it). Each line object should look like this: \{ source: “Pete”, message: “my message”, className: “red”\}

12. The resource bar is the next easiest. Just iterate through the items in the catan.definitions.ResourceTypes and catan.definitions.Buyables arrays, and update the amounts based on what is in the player’s hand. a. With our model, you can call the two following methods with a string to find the amounts: i. this.getClientModel().getClientPlayer().getResources().getItem(STRING); ii. this.getClientModel().getClientPlayer()\”get\+STRING\”(); if the value is the DEV_CARD option, calling [ii] may return the card hand instead of an amount, so you want to make sure the result is a number or undefined, and pass that into the view method.

13. You’ll probably want to do the map actions next.
   a. The doSoldierAction() and startDoubleRoadBuilding() can stay unimplemented until you implement development cards—that’s the only place they’re called. The doubleRoadBuilding function should let you place two free roads (without closing the modal between). You’ll want to track the two locations and then send them together in the RoadBuilding command.
   b. Robbing takes place within the map as well. When it is time to rob someone (i.e., the Turn tracker status is “robbing”) create the robbing map modal. On choosing a location, the onDrop function is called, which in this case should set the “players to rob”
on the rob overlay and then show that modal.

c. startMove() should bring up the overlay by calling the
this.getOverlayView().showModal(String titleToUse) function. This is called by the
resource bar when you click on the roads, settlements, and cities. i. You need to track
somehow whether you are already trying to drag/drop something because otherwise the
modal can get locked up(?) ii. You need to also track whether the item to send is free or
not so resources are removed appropriately. iii. And whether a piece is
“disconnected”—that should only happen in the setup round and on playing a road build
card.

d. onDrag() is supposed to do the validity check for whether a piece can be
placed there. e. onDrop should close the modal and send the appropriate command or
bring up the next display.

f. cancelMove should close everything without sending a command.

14. Probably now is a good time for the setup controller. It should automatically bring up the
road overlay and send a (free) road command, then do the same for settlement, then
finish your turn. And redirect as soon as the setup phase is over.

15. *Now you can basically play a whole game, so we can do the extra game features:

16. Trading

a. Both trading things are fairly straightforward conceptually, they’re just very
state-based, so that can complicate things.

b. Maritime trade—I used a PortList to track the player’s available ports, so that I
always knew what amount to send (initialized at 4 for everything, then decreased if a
player build on a port). I then just had a “give” and “get” variable to track which two
resources I wanted to exchange.

c. Domestic trade-- use a resourceList to store the cards you want to exchange:
receiving is negative, sending is positive.

i. Only enable the trade button when everything is set up. When the
button is pressed, send the offer command to the server.

ii. Once that happens, the model stores a “tradeObject” which we use to
determine who should be shown the trade.

iii. The person to trade with (track by id) should have the accept overlay
show up with the button enabled/disabled depending on whether they can
accept. iv. Clicking cancel or accept calls the accept(Boolean) function,
which should send the corresponding command to the server.

17. Cards can only be played the turn after they are received. To track this, we have an
“old” and “new” hand of cards in the model. You can only enable the cards in the old
hand. These hands are combined at the end of each turn in the server. Each card has a
specific function that is called, which should close the modal and complete some action.
For most of these, you’ll send a command, but the soldier and road building are different.
Right now we've set it up so that the controller receives as some of its parameters the soldier and road actions, since both of those pertain to the map. You should store those, and call them from the useSolider() and useRoadBuild() methods (the view doesn't call them directly).