1. Proxy Pattern

Principle

In this game implementation of Catan, one issue we must resolve is how a client communicates with the server. The server's job is to store synchronized data for all the clients to access. The server ensures that all clients can see the exact same data/game, so that the clients can speak to one another in an indirect sense.

In the actual design of Catan, there are many, many different parts that come together to create the client-side of the program. The GUI consists of several views, which show a visual representation of the game data. All of these individual parts are necessary to divide the game into smaller, efficient sub-pieces.

However, we run into an issue, of how each one of these parts can communicate its valuable data to the server for all clients to access. One solution is to let each part communicate across the network, and manually send and receive this data. However, this is quickly shown to be a sub-optimal solution. In this case, every single class must know about HTTP protocols, know how to package data, and several other methods which would violate the single-responsibility principle.

To solve this problem, we created a class called ServerProxy, whose job is to be a proxy (facade) the all classes client-side. This class implements IServer, an interface, in order to ensure that all calls across the network can be quickly and easily mapped to another function server side. The overall purpose of this class is to let other classes call functions in the Server Proxy, and pretend that it is the server. The rest of the classes in the client will be able to assume that the ServerProxy is the server, and make all their calls to it, without having to know about protocols and data transfer over the internet.

Implementation

We have implemented an IServer, so that our Server Proxy can have the exact same methods we will need server-side. Some of these methods are:

- `public Session login(String username, String password)`
- `public GameHeader createGame(String name, boolean randomTiles, boolean randomNumbers, boolean randomPorts)`
- `public JSONObject getModel(int version)`

These methods, as one can tell, are the very methods that a client will need to call in order to allow for functionality. Though the actual login will occur server side, any client object can access the server through the ServerProxy, which implements the login function. Here, it uses the ClientCommunicator to package up the information, send it to the actual Server (which also implements login server-side), and return the result as a Session object, which contains useful information about the current session.

Overall, this allows for controlled access to the server through one class, allowing for other classes to simply ignore the functionality of how the call is actually being performed.

2. Data Integrity Enforcement

Principle

As data is placed in our model, we need a way to ensure that it follows the guidelines we have set for it. To guarentee that a system can work properly, we must set limits for what certain values can be set to, what methods can be called at what times, etc. The most basic way to ensure this is to create checks at critical points (when variables are being set/updated) to ensure that the data follows the rules we have created. If we get a call trying to change a variable to an illegal amount, it means someone is making a bad call, either intentionally (cheating), or another

method is not properly handling these cases. Overall, we must continually ensure that our data is not corrupted or invalid (such as roads in invalid locations, players having too many cards, etc.).

Implementation

I will show a couple of examples to demonstrate how our group ensures valid data. In our implementation, we have checks to guarantee only valid data is used. For example, in the Player class, we want to make sure that only 19 cards of any one type can be out at a time. This is checked in the setResources function:

```
public void setResources(ResourceList resources) throws
InsufficientResourcesException {
	for (ResourceType type : ResourceType.values()) {
		int numResourceCardsForPlayer = resources.count(type);
		if (game.getBank().getResources().count(type) +
numResourceCardsForPlayer > 19)
			throw new InsufficientResourcesException();
	}
	this.resources = resources;
}
```

This will ensure that a player will always have a valid amount of cards in their hand.

Other examples are shown as follows:

```
public boolean canMoveRobberTo(HexLocation location) {

if (location.equals(robber)) {
	return false;
}
if (location.getDistanceFromCenter() > radius) { // Robber must stay on the board.
	return false;
}
if (getHexAt(location).getResource() == null) { // Can't move to the desert
	return false;
}

return true;

}
```

This will ensure that the robber is only moved to valid positions on the board.

```
// make sure the city is on the board.
	VertexLocation location = town.getLocation();
	if (location.getDistanceFromCenter() > radius) {
		throw new IndexOutOfBoundsException();
	}
	// enforce Distance Rule
	for (VertexLocation neighbor : location.getNeighbors()) {
		if (municipalities.containsKey(neighbor)) {
			throw new GameInitializationException("Distance Rule
violation!");
		}
	}
	if (municipalities.containsKey(location)) {
		throw new DuplicateKeyException();
	}

	municipalities.put(location, town);
```

This is called from the initializeMunicipalitiesFromList function in the Board class, to ensure that any municipality is in a valid location on the board, is not bordering another municipality, and is in a vacant spot.

```
EdgeLocation location = road.getLocation();
if (location.getDistanceFromCenter() > radius) {
	throw new IndexOutOfBoundsException();
}
if (roads.containsKey(location)) {
	throw new DuplicateKeyException();
}

roads.put(location, road);
```

This is called from the initializeRoadsFromList, also inside the Board class, to guarantee that all roads are placed in valid positions.

3. Can Do? methods

Principle

To provide visual cues to the player, can do methods are needed so that a player can know whether or not a certain action is legal or not. This includes things like knowing whether or not a certain location is ok to place a road, or whether or not they can make a trade with another player. Without this functionality, it is extremely difficult for the player to make informed decisions in the game.

As the controllers and views should not have direct access to the data with our model-view-presenter pattern we are following, we provide canDo methods, so that each controller may make an easy call to determine if a certain action is legal or not.

Implementation

Here follows a list of all our canDo methods, and the methods to which they are precondtions:

Can Do Method	**Is Precondition to:**
`public synchronized boolean canRoll(PlayerReference player)`	`public synchronized void doRoll(PlayerReference player)`
`public synchronized boolean canRob(HexLocation hexLoc)`	`public synchronized void doRob()`
`public synchronized boolean canFinishTurn()`	`public synchronized boolean doFinishTurn()`
`public synchronized boolean canBuyDevelopmentCard()`	`public synchronized boolean doBuyDevelopmentCard()`
`public synchronized boolean canBuildRoad(EdgeLocation edgeLoc)`	`public synchronized boolean doBuildRoad()`
`public synchronized boolean canBuildSettlement(VertexLocation vertexLoc)`	`public synchronized boolean doBuildSettlement(VertexLocation vertexLoc)`
`public synchronized boolean canBuildCity(VertexLocation vertexLoc)`	`public synchronized boolean doBuildCity(VertexLocation vertexLoc)`
`public synchronized boolean canYearOfPlenty()`	`public synchronized boolean doYearOfPlenty()`
`public synchronized boolean canRoadBuildingCard()`	`public synchronized boolean doRoadBuildCard()`
`public synchronized boolean canSoldier()`	`public synchronized boolean doSoldier()`
`public synchronized boolean canMonopoly()`	`public synchronized boolean doMonopoly()`
`public synchronized boolean canMonument()`	`public synchronized boolean doMonument()`

`public synchronized boolean canOfferTrade()`	`public synchronized boolean doOfferTrade()`
`public synchronized boolean canAcceptTrade()`	`public synchronized boolean doAcceptTrade()`
`public synchronized boolean canMaritimeTrade()`	`public synchronized boolean doMaritimeTrade()`

Note: Most of these do methods are to be fully implemented server side, which will actually change the model data, update all necessary members, and send the response back when the ServerPoller gets the new model.

TEAM REPORT

I, Jordan Chipman, spent about 20 hours in the implementation of this phase.

Team Evaluation:

Steve Pulse: 4.8

Justin Snyder: 5

Grant Lubeck: 4.8

Jordan Chipman: 4.8

This group has been a real pleasure to work with, and I have enjoyed learning from different coding styles, and how to work with a group. Its been a great time!