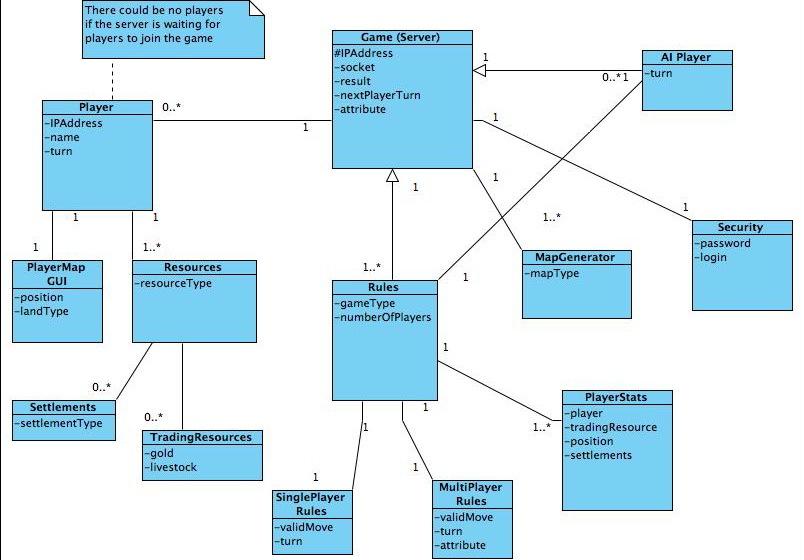
**Project Name**

*Student Name and Number (one per line)*

*Date (Due March 25th @ 5:30pm)*

You must submit this milestone through **GitHub** and have uploaded the project's source code (-2 marks if you don’t use GitHub)

# **Class Diagram of Actual System (5 points)**



1.1 Conceptual Domain Diagram - FreeCol

The Domain Diagram and conceptual architecture for the initial design of FreeCol was based on the client-server model. The reason for this was specifically to support multiplayer games that were to be played over the internet. The communication of the Player and the server was limited to a single control flow between the player and the server (the game), and knowledge was either contained on the server side or the client side. The designers of FreeCol however chose a different and perhaps unique approach of creating an additional layer called Common.

The Common layer contains aspects of the game that are shared by both the client and the server. It doesn’t contain details like GUI but the model of the game common to both client and server. The model could be described as the conceptual or imaginary aspects of the game. for example the concept of “building”. A building has a type and a builder, these aspects award the player different scores depending on different aspects of the building. Both Server and Client need this information in order to play the game. On the other hand the Server needs no information about the GUI since it is the player who needs a visual representation of the game. Creating the Common model for both Client and Server will obviously lead to much less code repetition, since both Client and Server utilize the same classes for common features of the game, this in turn reduces the size of the Server and Client side Classes. FreeCol is however a large program and the rest of this document will focus on the Server SIde aspects of the game. It is important however to get an overview of the whole game to see how the game runs.

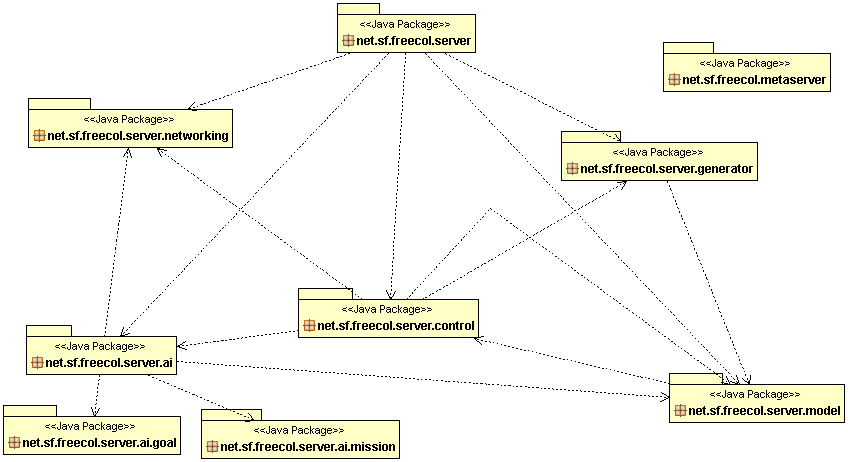


Figure 1.2 Package Diagram for the server aspects of FreeCol

The Server packages of FreeCol that this project focuses on are shown in the Package Diagram, figure 1.2 We will not discuss the AI packages, but focus on the Model, Control, Networking and Generator packages. Each package but performs specific tasks related to game play.

The Control package contains the classes responsible for the control of the game. There are nine classes each responsible for different aspects of the controller. In the Conceptual Domain Model we assume there is a controller that controls the whole server side activity. These activities include making connections, checking all networking activities and validating the rules of game. In the conceptual model, validating Rules was by defined by the Rule Class but in FreeCol this work is done by the Controller class in the server Control Package.

InputHandler: - Handles the network messages.

Controller: - The control object that is responsible for making changes to the internal model and for communicating with the clients.

FreeColServerHolder: - This base class provides thread-safe access to a FreeCol server for several subclasses.

PreGameInputHandler: - Handles the network messages that arrive before the game starts.

InGameInputHandler: - Handles the network messages that arrive while [in game](#IN_GAME).

PreGameController: - The control object that is responsible for setting parameters and starting a new game.

InGameController: - The main server controller.

UserConnectionHandler: - Handles a new client connection.

ChangeSet: - Changes to be sent to the client.

**Package net.sf.freecol.server.model):**

The server package contains ‘model’ classes with server specific information. The package diagram showing relationships, dependencies and associations can be seen below:

***Interface:***

**ServerModelObject:** It is an interface for server-side objects which stores extra information to save game. It details the conceptual aspects of the game.

***Class Description:***

**DemandSession:**

A session type that demands tribute from a colony. It extends the *ServerPlayer* class (contains additional server specific information) and the *Transaction Session* class which is the root class for all sessions.

**DiplomacySession:**

* A type of session to handle diplomacy. It also extends the *Transaction Session* class.

**LootSession:**

* It is again a subclass of *Transaction Session* which handles the looting of a cargo. It basically contains the goods that are available for capture.

**ServerBuilding, ServerColony, ServerColonyTile:**

It is the server version of building, colony and colony tile. All these classes implement the *ServerModelObject* interface, which in turn contains the extra information to save a particular game between the client and the server.

**ServerEurope:**

* It is the server version of Europe. It extends the *ServerPlayer* and the *ServerUnit* class and also implements the *ServerModelObject* interface. The *ServerUnit* class contains server version of units such as consumer, location, movable objects, etc.

**ServerGame:**

* It contains the Server representation of the game. Unlike client side, the server side doesn’t contain the GUI details.

**ServerIndianSettlement:**

* The server version of Indian Settlement. It extends the *IndianSettlement* class and implements the *ServerModelObject* interface.

**ServerPlayer:**

* *A Player with additional (server specific) information. It points to player's Connection* and *Socket.*

**TradeSession:**

* It is a session which handles trading. It allows actions such as buying, selling or gifting during a session. It also extends the *TransactionSession* class.

**Relation between Domain diagram and Actual class diagram:**

The Domain model describes resources which are essential to add information to the client side of the game. The actual class diagram contains more details through the server side via “ServerPlayer” class and the “ServerModelObject” interface. The ServerPlayer contains control over most the classes for storing additional server specific information about the player of the game. Eg: server resources such as building, colony, colonytile. Moreover, the actual class diagram does not need to have GUI in the server as functionality.

**Generator package** (Package net.sf.freecol.server.generator)

**Package net.sf.freecol.server.generator c**ontains the map generator that creates maps and sets the starting locations for the players and includes five classes and three interfaces classes i.e

*Interfaces* : MapGenerator, MapLayerGenerator and MapLoader.

*Classes* : LandGenerator, River, RiverSection , SimpleMapGenerator, TerrainGenerator.

**Description of classes and interfaces and associations** :

* **LandGenerator** : Class for creating a land map. LandGenerator class is a subclass of Map class.
* **River** : A river for the map generator. River class depends on River section class and SimpleMapGenerator class which further depends on the Map class.
* **RiverSection**  : This class facilitates building, editing the TileImprovement style for rivers. Rivers on the Map are composed of many individual TileImprovements displayed on each Tile the river flows through.The river TileImprovement on a Tile has a style which represents the inputs/outputs of water to/from neighboring Tiles .This class allows manipulation of individual stream(s) to neighboring Tiles .
* **SimpleMapGenerator** : Creates random maps and sets the starting locations for the players. SimpleMapGenerator implements the MapGenerator.
* **TerrainGenerator**  : Class for making a Map based upon a land map. TerrainGenerator depends on River and RiverSection class.

**Interfaces** :

**MapGenerator** : Creates maps and sets the starting locations for the players

**MapLayerGenerator** : Generates a map layer.

**MapLoader** : Load a map into the given game, copying all layers up to the given layer.

**Networking Package (**Package net.sf.freecol.server.networking**)**

Contains the server networking classes. The main server networking class is Server

**Classes** :

**DummyConnection** : A dummy connection used for AI players.

**Server** : The networking server in which new clients can connect and methods like sendToAll are kept.

**Domain vs Actual class description:**

The MapGenerator class as described in the domain diagram is quite similar to the actual class diagram. However, it contains some specific details which are not mentioned in the domain model. The MapLayerGenerator and the MapLoader are important classes which are not mentioned in the domain diagram. They are necessary for a complete system architecture.

For the networking package, the class diagram contains a separate functionality for establishing a connection between AI players and the client through the networking class module. This was done through the main Server(Game) in the domain diagram.

We all should know what we are working on for Sunday but get some stuff done for Friday/Saturday so Sunday we can review what has been done

Hardev upload the package diagram for the server..

Ede - client common server - Overall Description of the Game.Done!!

Hardev Mishal - Control Package

Navrang - networking and Generator packages

Ronak - Model

1 to 2 pages of text -> Describe the diagram and the relationships between the classes. Which conceptual classes map to which actual classes? Is there a discrepancy between the concepts and the actual classes? What does it mean to the architecture of the system? Describe any reverse engineering tools used (e.g., Diver, ObjectAid UML Explorer, Enterprise Architect - Sparx Systems, ArgoUML).

# **Code Smell and Possible Refactoring (5 marks)**

Be ambitious! You will get no marks if all you suggest is a rename of a class. (Note: in the M4 you will implement a small part of the refactoring you suggest here)

Describe the code smell(s) and how you will combine together a series of refactorings to fix the smells. Describe how the refactorings are interrelated and how they correct the problems you identified in the above. For example, “First, I moved the methods X and fields Y and Z, to increase cohesion and to reduce feature envy. Second I renamed the class to reflect its new limited responsibilities. Third I removed the coupling to …”

1 to 2 pages of text + 1 to 2 UML diagrams (diagrams should be at most one additional page).

In maximum of one page, for one of the refactorings you suggest: Copy-and-paste the class, method, and attributes **declarations** (and anything else that is necessary) directly from the source code. Do not include code that is unnecessary (be selective, you will lose marks for large dumps of source code.)