**The FreeCol Server Architecture**

**SOEN 6471 - Milestone 3**

*Student Name and Number (one per line)*

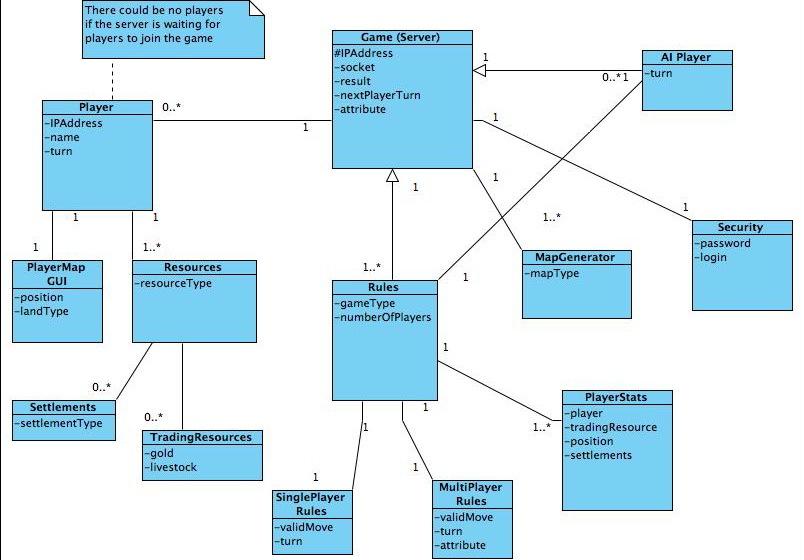
*Ede Cameron 2952270*

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

# 

# 

# **The FreeCol Server**



1.1 Conceptual Domain Diagram - FreeCol

The Domain Diagram and conceptual architecture for our 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 approach and incorporated 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.

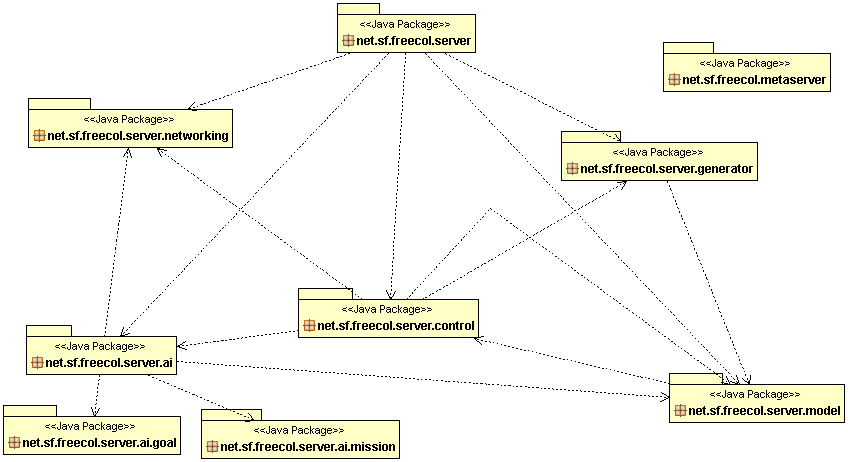


Figure 1.2 Package Diagram for the server aspects of FreeCol

The Server packages of FreeCol that this project focuses on are shown in the Package Diagram. We will not discuss the AI Packages, but focus on the Model, Control, Networking and Generator Packages.

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. The primary tasks of the controller are handling network messages, making changes to the internal model and communicating these messages to clients. Handling network messages before and during game play. The primary classes been the Controller, the InGameController and ChangeSet. ChangeSet is responsible for updating the Client side model.

The Model Package contains the model classes but with server specific information. As mentioned above the model of the game is the conceptual world that makes up the game. The Model Classes are divided into Session classes, Building and Colony classes as well as Settlement and Player Classes.

The ServerGame Class contains the Server's representation of the game. Unlike the client side, the server side doesn’t contain the GUI details. The ServerPlayer Class represents a Player but with additional server specific information and points to the Players Connections and Sockets. The ServerModelObject is the interface for server-side objects. It stores extra information in order to save a game and details the conceptual aspects of the game such as Building, Colony and Tile. The turns of the games in the model are described as transaction sessions and involve transactions between players such as diplomacy and trade. These classes extend the TransactionSession Interface. 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 players of the game.

The Generator Package **c**ontains the classes that create maps and sets the starting locations for the players. There are three interfaces. The MapGenerator which creates maps and sets the starting locations for the players. The MapLayerGenerator which creates a map layer, consisting of Land, River and other types of terrain and the MapLoader which loads the map into a given game. The SimpleMapGenerator **c**reates random maps and sets the starting locations for the players.

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.

The Networking Package contains the server networking classes. The main server networking class is the Server Class which is where players make their initial connection, and server related functionality like Broadcast are implemented. As well as starting and shutting down the server. The DummyConnection Class is used to connect AI players. This package is small compared to the others and is an implementation of a skeleton Server with little functionality. The networking package, in the game contains 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.

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

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.)

Hey I have chosen my code smell and will do the one page description of the refactorings involved.

In the ServerPlayer the method combatAction(...)

lines 1890 > 2300 should be a class to start with?? Then figure out the other code smells..

Super refactoring!! Long method switch statements etc just a complex mess extract class then extract method etc Replace case statements with poly morphism blah blah blah

ede

Code smell easy long method

InGameController

endTurn.. lines 652> 842 : unreadable

csMonarchAction lines 994>1141 long method SWITCH Statments from Monarch Class in Common.Model

enum of different types of actions >> switch statements could be converted to classes scince they have attributes and actions in csMonarchAction

declareIndependence 1279>1353

notice the for statements at end of the code could be methods again long method

Harder

Another Possible refactoring is moving common aspects out of InGameController and into individual classes

for example there are classes with the word "proposition" or Transaction Buy and sell (lines 1426> 1636) in them these all have something in common they are transactions between players. Maybe remove them from the InGameController and into a separate class called InGameTransactionController some one could write a bit about this and other refactorings about common functionalities that could be grouped in unique classes?? To solve the large class problem with the InGameController class ??? anotherexample is all the Control aspects for land like move setNewLandName remove into a cless called InGameLandController I assume you get the idea...

The same could be said for ChangeSet there are numerous common methods that could be moved to individual classes to create a more readable code base..

I will look into this for the ServerPlayer Class