Code Smell and Refactoring

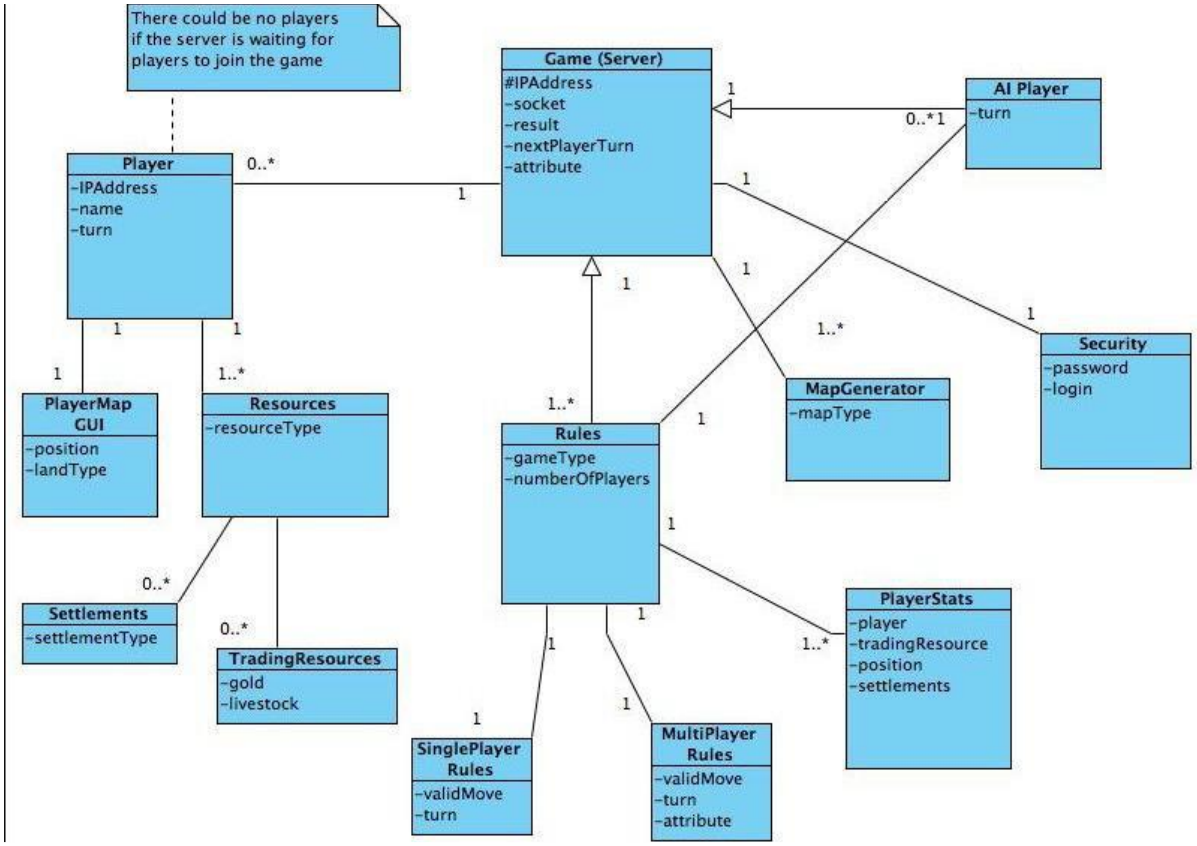
Freecol Server

SOEN 6471

By:

Ronak Patel [6483607]
Vishal Mittal [6425240]
Hardev Goraya [6446019]
Navrang Singh [6447430]

FreeCol server Conceptual Diagram:

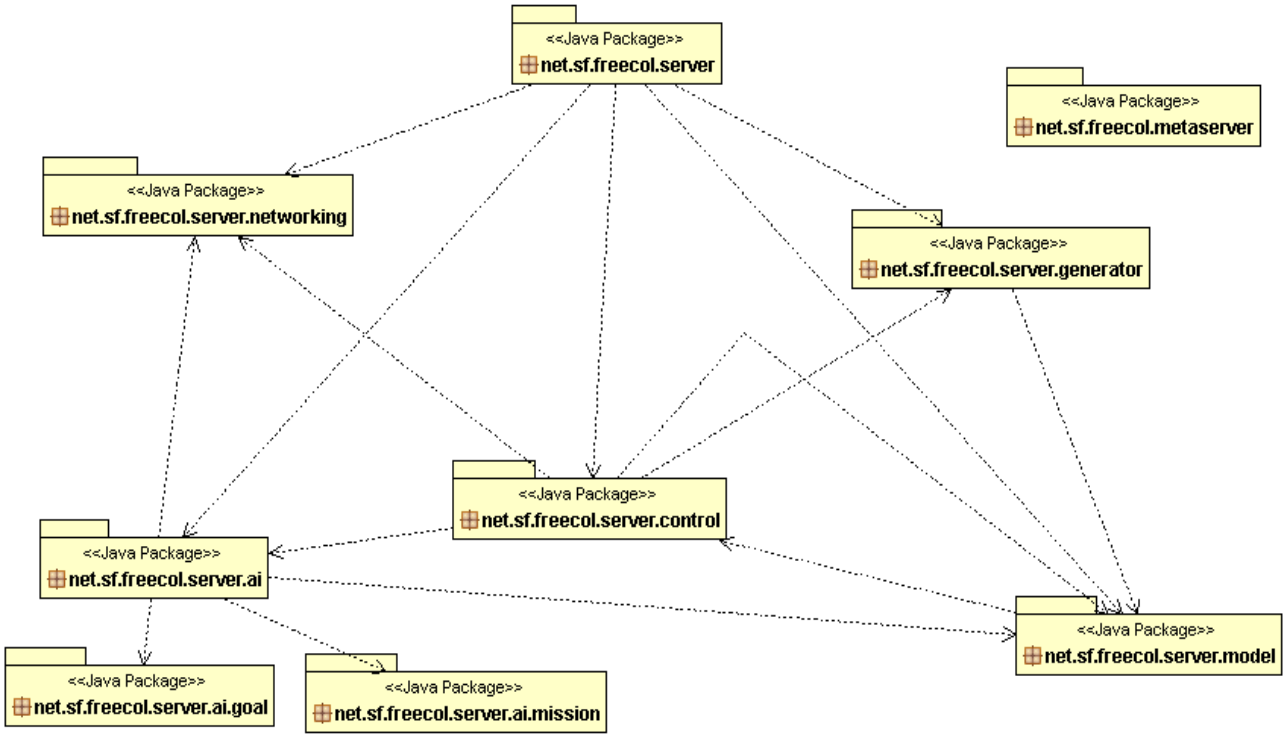


The Domain Diagram and conceptual architecture for our initial design of FreeCol was based on the client-server model. The reason for this was specifically to support multiplayer games that were to be played over the internet. The communication of the Player and the server was limited to a single control 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
obviously leads 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.

Package Diagram of server



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.

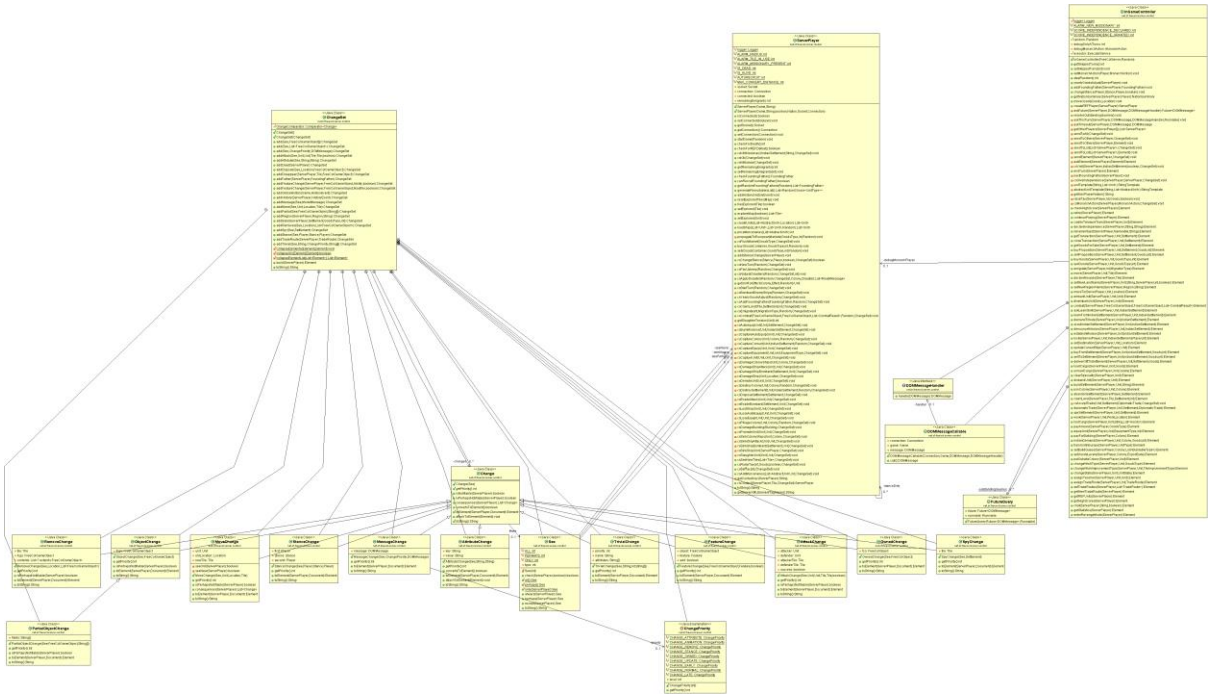
The Generator Package **c**ontains the classes that create maps and sets the starting locations for the players. There are three interfaces. The MapGenerator which creates maps and sets the starting locations for the players. The MapLayerGenerator which creates a map layer, consisting of Land, River 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.

Refactoring the server Package in the FreeCol game:

In the Freecol the server package having bunch of classes and all those classes having many code smells like god class, long method and type checking for example ChangeSet in the server.control package, inGameController class in the same package. And ServerUnit class in the server.model package respectively.



As seen in the diagram there is three big classes changeSet,ServerPlayer and InGamecontroller.Change set class and ServerPlayer class is very huge class having 1671 and 3750 number of line of code. It use large number of methods and lot of IF and switch Statements. Moreover in the change set there is 15 numbers of inner classes and all of these are public static. And the dependency between these classes is too much these all classes using a number of variable from their parent classes .And also it import an ArrayList that also make code very complicated.

Import java.utill.Arraylist;

And after that it makes an object of array list called changes its attribute is private that make much dependency.

```
19
20 package net.sf.freecol.server.control;
21
22 import java.util.ArrayList;
54
55
57  * Changes to be sent to the client.
59 public class ChangeSet {
60
61     // Convenient way to specify the relative priorities of the fixed
62     // change types in one place.
63     public static enum ChangePriority {
85
86     private ArrayList<Change> changes;
```

Furthermore ServerPlayer is a Player with Additional (server specific) information and incuses some fields i.e. Socket, connection, Boolean, List<serverplayer> which can be moved to another separate class in order to split the responsibility of the class.

```
107
108     // How far to search for a colony to add an Indian convert to.
109     public static final int MAX_CONVERT_DISTANCE = 10;
110
111
112     /** The network socket to the player's client. */
113     private Socket socket;
114
115     /** The connection for this player. */
116     private Connection connection;
117
118     private boolean connected = false;
119
120     /** Remaining emigrants to select due to a fountain of youth */
121     private int remainingEmigrants = 0;
122
123     /** Players with respect to which stance has changed. */
124     private List<ServerPlayer> stanceDirty = new ArrayList<ServerPlayer>();
125
```

By using extract class refactoring we can reduce some dependency between the inner classes in the changeSet classes and ServerPlayer class to make the code much readable and maintainable.

Refactoring the changeSet class(GOD Class):

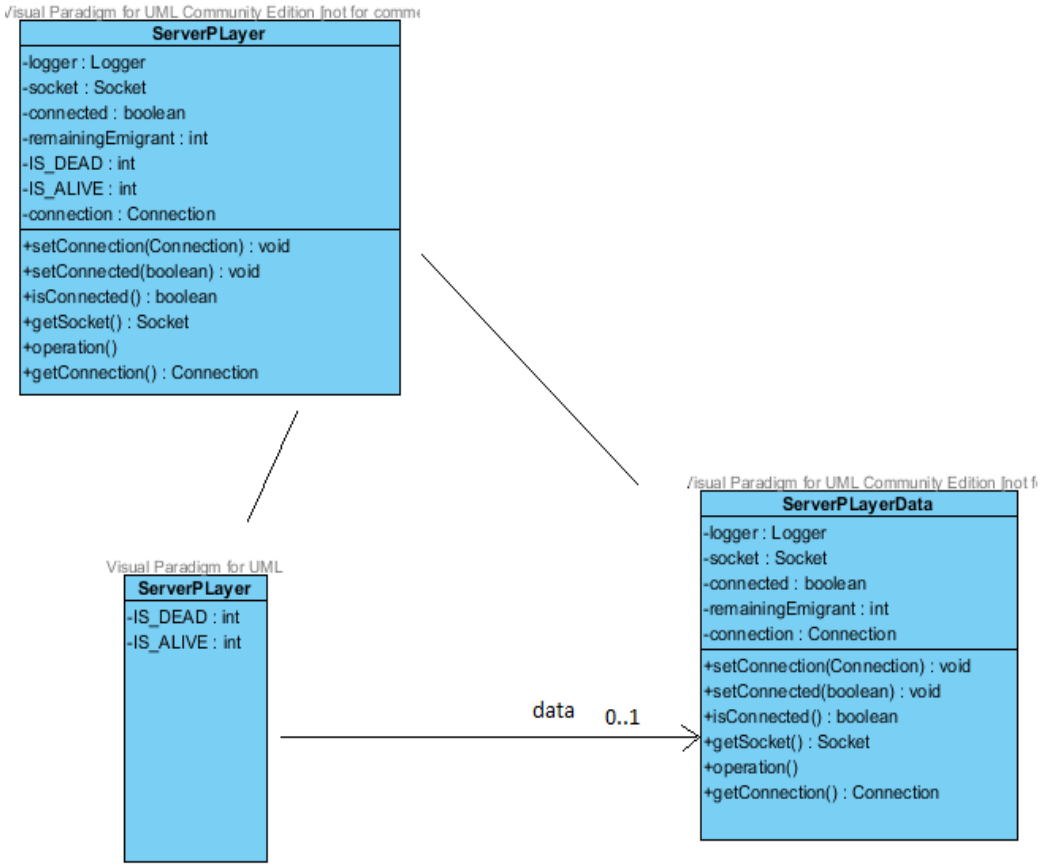
In this case we can separate "Array_List" behaviour into its own nested class by inserting the field class in the original class to limit the responsibly to the original class. Secondly we can remove the Fields i.e. ArrayList<change> and replace by the link from old class to new class in order to make connect among them. Lastly replace a set of fields (write accesses and read accesses) with new container object. All references to the fields are updated to access the new container object.

Refactoring the ServerPlayer(GOD Class):

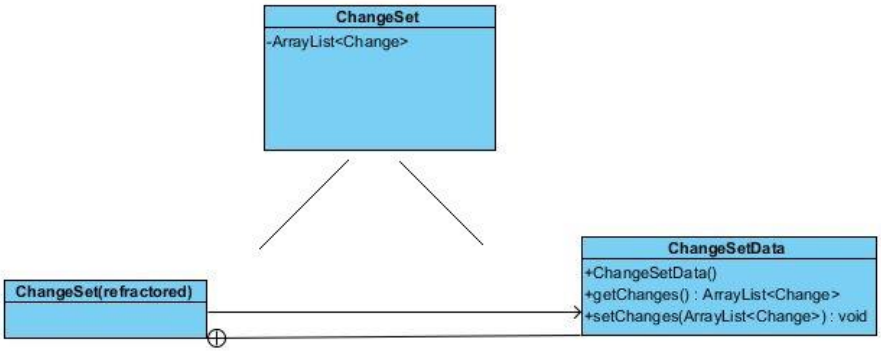
As we already know the ServerPlayer is the huge class having lot of dependency between them. To make it less coupled we extract some of the members to put into another class and then just call from the Serverplayer class.

we can extract the body of these fields into the another class called ServerPlayerProduct in the server.model.package define their getters seeters and instead of doing all these stuff in one class we just call the method and return the value to the ServerPlayer method class.

ServerPlayer.java



ChangeSet.java



Refactoring for the demand tribute (long Method)

The *InGameController* class is inherently too big with more than 3000 lines of code. Moreover, it can be observed that the method named "*demandTribute"* is large enough. The demandTribute method has a ServerPlayer and unit that is demanding the tribute from IndianSettlement. Hence, we have ServerPlayer, unit and IndianSettlement as method parameters. We will extract the method *ModelMessage* into a new method *m*. The new method will have parameters such as unit, settlement, gold and the year when the settlement is made. So, we will create a new private method *m* and call this method to access its properties. The additional gold attribute added in the method is also demanded as tribute. In this way, we can achieve code comprehension by extracting a long method and the code is indeed maintainable due to this. A snapshot for the code is given below:

InGameController.java

Original Source

```
2082                    Tension.TENSION_ADD_NORMAL));
2083            settlement.setLastTribute(year);
2084            ModelMessage m;
2085            if (gold > 0) {
2086                indianPlayer.modifyGold(-gold);
2087                serverPlayer.modifyGold(gold);
2088                cs.addPartial(See.only(serverPlayer), serverPlayer, "gold", "score");
2089                m = new ModelMessage(ModelMessage.MessageType.FOREIGN_DIPLOMACY,
2090                                     "scoutSettlement.tributeAgree",
2091                                     unit, settlement)
2092                    .addAmount("%amount%", gold);
2093            } else {
2094                m = new ModelMessage(ModelMessage.MessageType.FOREIGN_DIPLOMACY,
2095                                     "scoutSettlement.tributeDisagree",
2096                                     unit, settlement);
2097            }
2098            cs.addMessage(See.only(serverPlayer), m);
2099            Tile tile = settlement.getTile();
2100            tile.updatePlayerExploredTile(serverPlayer, true);
2101            cs.add(See.only(serverPlayer), tile);
2102            unit.setMovesLeft(0);
2103            cs.addPartial(See.only(serverPlayer), unit, "movesLeft");
2104
2105            // Do not update others, this is all private.
```

Refactoring of the Ship Method (long method) in ServerPlayer class:

The *csPillageColony* method in the *ServerPlayer* class has a code smell of long method. The method works in order to damage the building or ship or steal some resources such as any goods or gold. It has method parameters such as unit, colony, random and changeset. Every unit is owned by a player and it has a location. The colony is the colony which will be pillaged by the attacker. The method consists of almost 90 lines of code. Hence, we will extract a new ship method and pass these parameters in this method. In this way, the damaged or illegally obtained ships will be dealt by this method. A snippet of the code is given below:

```
3248    private void csPillageColony(Unit attacker, Colony colony,
3249                                 Random random, ChangeSet cs) {
3250        ServerPlayer attackerPlayer = (ServerPlayer) attacker.getOwner();
3251        StringTemplate attackerNation = attacker.getApparentOwnerName();
3252        ServerPlayer colonyPlayer = (ServerPlayer) colony.getOwner();
3253        StringTemplate colonyNation = colonyPlayer.getNationName();
3254
3255        // Collect the damagable buildings, ships, movable goods.
3256        List<Building> buildingList = colony.getBurnableBuildingList();
3257        List<Unit> shipList = colony.getShipList();
3258        List<Goods> goodsList = colony.getLootableGoodsList();
3259
3260        // Pick one, with one extra choice for stealing gold.
3261        int pillage = Utils.randomInt(logger, "Pillage choice", random,
3262            buildingList.size() + shipList.size() + goodsList.size()
3263            + ((colony.canBePlundered()) ? 1 : 0));
3264        if (pillage < buildingList.size()) {
3265            Building building = buildingList.get(pillage);
3266            csDamageBuilding(building, cs);
3267            cs.addMessage(See.only(colonyPlayer),
3268                new ModelMessage(ModelMessage.MessageType.COMBAT_RESULT,
3269                    "model.unit.buildingDamaged", colony)
3270                .add("%building%", building.getNameKey())
3271                .addName("%colony%", colony.getName())
```