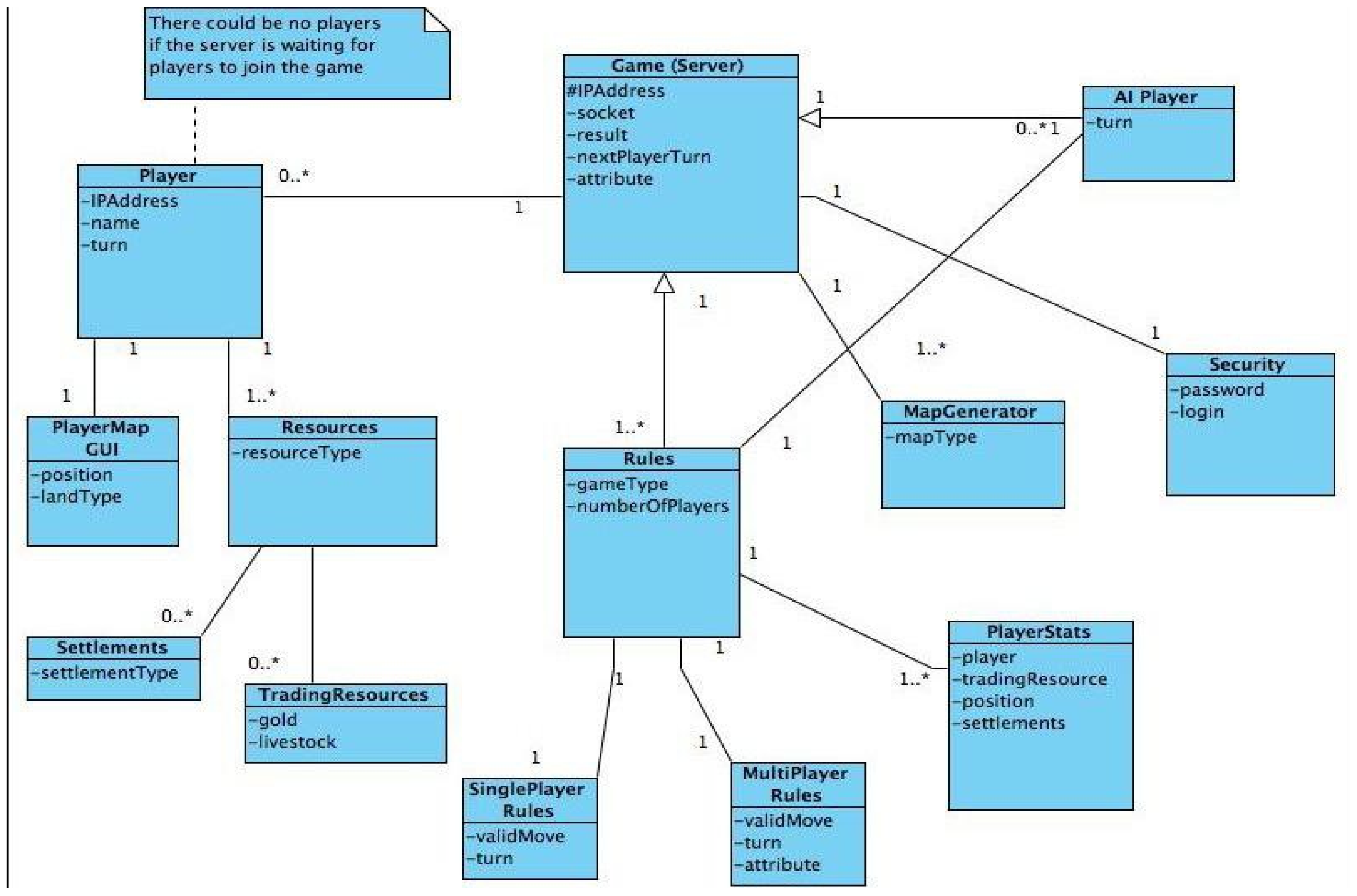
The FreeCol Server Architecture

SOEN 6471 Milestone 3

Ede Cameron 2952270

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

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 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.

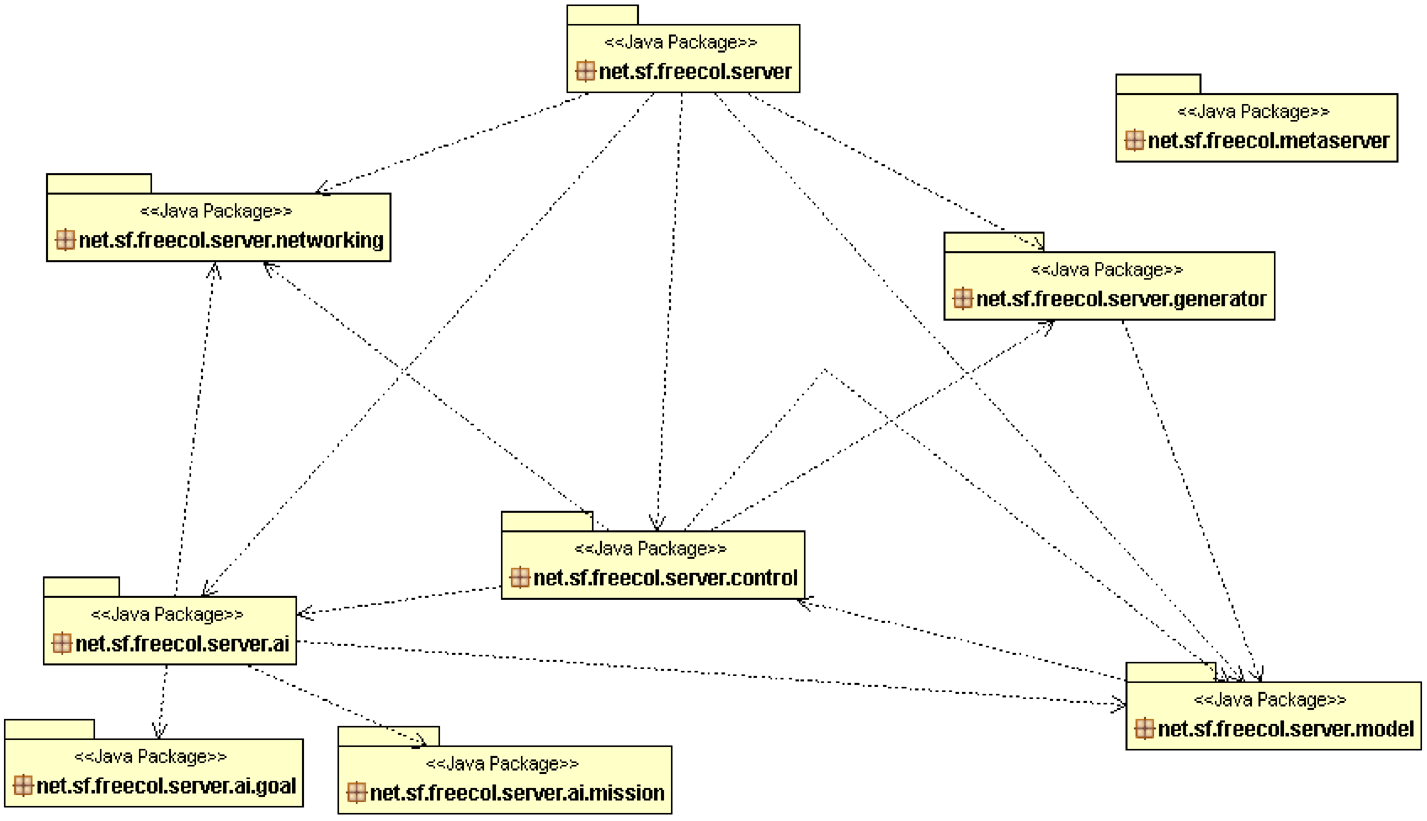

<<Java Package>>
net.sf.freecol.server
<<Java Package>>
net.sf.freecol.metaserver
<<Java Package>>
net.sf.freecol.server.networking
<<Java Package>>
net.sf.freecol.server.generator
<<Java Package>>
net.sf.freecol.server.control
<<Java Package>>
net.sf.freecol.server.ai
<<Java Package>>
net.sf.freecol.server.model
<<Java Package>>
net.sf.freecol.server.ai.goal
<<Java Package>>
net.sf.freecol.server.ai.mission

Figure 1.2 The Package Diagram for the server aspects of FreeCol derived from the Eclipse Plugin ObjectAid.

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 Elements of FreeCol

“The game has been developed incrementally without any formal design document”

http://www.freecol.org/history.html

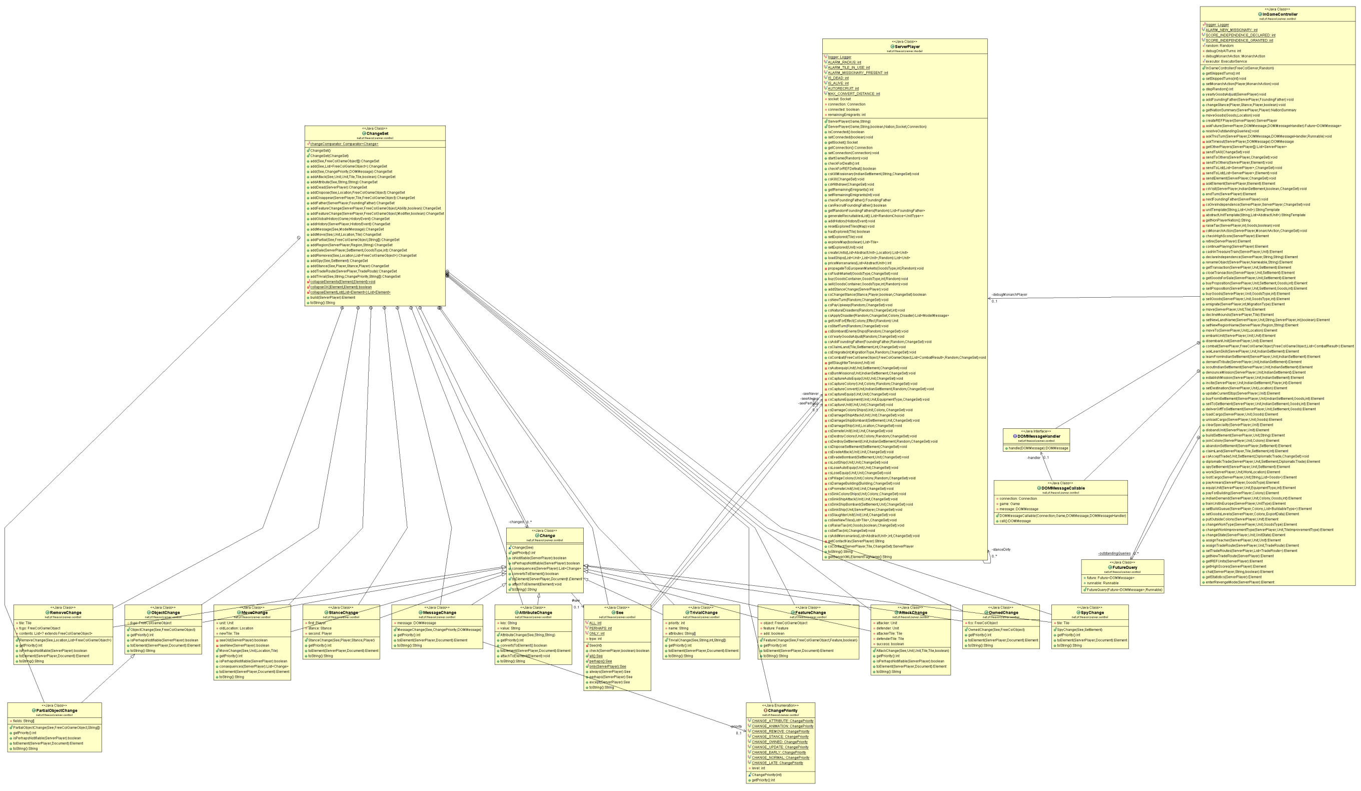


Figure 2.1 Three Large Classes found in the model and Control packages. From Left to right ChangeSet, ServerPlayer and the InGameController. Notice the ChangeSet Inner Classes.

In first versions of FreeCol, the server was written in C++ and was later ported to Java[1]. There are numerous code smells throughout the Server Packages of FreeCol. Perhaps most notably the use of Large Classes and within these Classes, Long Methods. There is also an abundant use of Switch Statements and heavy use of Enumerated Types. Examples of Large Classes are the InGameController, the ServerPlayer and the ChangeSet Class. The ChangeSet Class exceeds 1500 lines of code and contains 15 Inner Classes. The InGameController Class and the ServerPlayer Class both have more than 3000 lines. The InGameController and ChangeSet are part of the Control Package and are responsible for game play. The ServerPlayer Class is in the Model Package and references each player in the game and the refactoring discussed below will be focused in these three classes.

ChangeSet has 15 inner classes most been labelled as “public static”. Although there are times where an inner class maybe justified, where a class is instantiated only through the parent class, or the inner classes depends heavily on the variables of the parent.[2] The use of inner classes leads to a large class that doesn't reveal its full functionality to programmers, and a search of the code itself is necessary in order to fully comprehend its functionality. By Extracting these classes and using polymorphism and inheritance the code structure could be more easily understood and reduce the size of the class considerably.

Switch statements and Enumerate types occur in all these classes. An example in the InGameController Class. The method csMonarchAction, not only uses switch statements but is also a Long Method (147 lines of code). The reason for this is that in the method each switch statement involves numerous complex rule changes, reflecting another code smell Divergent Change. The solution for this is of course to replace Switch Statements with Polymorphism. In order to do this one would have to access the Monarch Class in the Common Model Package which is were the enumerated variables are declared and replace these enumerated types with Polymorphism.

This is not the only method in this InGameController Class where code smells are apparent. There are several long methods declareIndependence, endTurn and equipUnit are all over 100 lines of code.

The ServerPlayer Class in the Model Package has again the same problems with Long Methods. The method combat has close to 400 lines of code and also includes a series of switch statements again coming from the Common Model Package. The solution again is to extract methods and replace switch statements with polymorphism. This would not however solve the primary code smell these Classes reflect, Large Class. In the control classes this is sometimes acceptable but when a Class has almost 3000 lines of code it is time to refactor.

Refactoring the ServerPlayer Class

A large class such as the ServerPlayer Class in the Model Package seems almost impossible to refactor. There are numerous long methods in this class, and the model package itself has numerous dependencies within the FreeCol Server Packages and the Common Packages between Player and Server. The problem however if reduced to its parts, might be a less daunting task.

The refactoring strategy for Extract Class recommends finding the methods that have some common function and move them to another class. Finding the similarities among the methods in the ServerPlayer Class is not too difficult, examples would be the methods that have something to do with exploration, exploreMap, setExplored, hasExplored, resetExploredTiles, there are more but the idea would be to move these into a separate class called PlayerExplored but keep this within the Server Model Package. The combat methods could be moved to the ServerPlayerCombat class. These combat methods have name like combat, csBurnMissions, csCaptureColony and csCaptureEquipment and all reflect turns of the game that involve combat between players. Once these Methods are moved to new Classes the real work begins. As mentioned above the code smells are still not really resolved by moving the methods for combat into a separate class. Now the methods themselves cam be refactored.

The method combat contains more than 300 lines of code and utilizes a complex switch statement which is instantiated in the Common Model Package through the object CombatResult, and is a public static enum called CombatResult. Each one reflecting a different combat action or result for example CAPTURE_COLONY or EVADE_ATTACK.

The code below is for the case where a colony is captured.

```
case CAPTURE_COLONY:
        ok = isAttack && result == CombatResult.WIN
        && colony != null
        && isEuropean() && defenderPlayer.isEuropean();
      if (ok) {
   csCaptureColony(attackerUnit, colony, random, cs);
   attackerTileDirty = defenderTileDirty = false;
   moveAttacker = true;
   defenderTension += Tension.TENSION_ADD_MAJOR;
          }
      break;
```

This one switch statement accesses numerous control changes that may be better represented using polymorphism and again access another enumeration Tension (TENSION_ADD_MAJOR) located in the Common Model Package in the Class Tension where the level of tension between players is measured as an integer, this enum could easily be eliminated for a simple static method that adds to a primitive type integer instead of using an enumeration. The long conditional expression could also be replaced with something more descriptive as to the nature of the condition like the method name EuropeanColonyCaptured() so the code could be more readable, The method could look like this

```
Boolean EuropeanColonyCaptured(ServerPlayer defenderPlayer){
      ok = isAttack && result == CombatResult.WIN
              && colony != null
              && isEuropean() && defenderPlayer.isEuropean();
        return ok;
    }
```

and in the primary method be replaced with

```
ok = EuropeanColonyCaptured(defenderPlayer);
```

Another example within combat method is this series if if then else statements.

```
if (attacker.hasAbility(Ability.PIRACY)) {
      if (!defenderPlayer.getAttackedByPrivateers()) {
        defenderPlayer.setAttackedByPrivateers(true);
        cs.addPartial(See.only(defenderPlayer), defenderPlayer,
            "attackedByPrivateers");
      }
} else if (defender.hasAbility(Ability.PIRACY)) {
      ; // do nothing
   } else if (burnedNativeCapital) {
        defenderPlayer.getTension(this).setValue(Tension.SURRENDERED);
       cs.add(See.perhaps().always(this), defenderPlayer); // TODO: just the tension
       csChangeStance(Stance.PEACE, defenderPlayer, true, cs);
    for (IndianSettlement is : defenderPlayer.getIndianSettlements()) {
            if (is.hasContacted(this)) {
```

```
            is.getAlarm(this).setValue(Tension.SURRENDERED);
           // Only update attacker with settlements that have
          // been seen, as contact can occur with its members.
          if (is.getTile().isExploredBy(this)) {
          cs.add(See.perhaps().always(this), is);
        } else {
              cs.add(See.only(defenderPlayer), is);
          }...
```

This code continues for another 70 lines (approximately) and is almost impossible to follow logically, creating obvious problems for debugging. The solution for both these long methods is extract method so that at least the logic of the control flow could be understood, for example

```
    if (attacker.hasAbility(Ability.PIRACY)) {
            if (!defenderPlayer.getAttackedByPrivateers()) {
               defenderPlayer.setAttackedByPrivateers(true);
               cs.addPartial(See.only(defenderPlayer), defenderPlayer,
              "attackedByPrivateers");
```

could be changed to

```
    if (attacker.hasAbility(Ability.PIRACY)) {
         setDefenderAttackedByPirates(defenderPlayer, cs);
       }
```

and a new method could be created called setDefenderAttackedByPirates for example

```
void setDefenderAttackedByPirates(ServerPlayer defenderPlayer, ChangeSet cs) {
            if (!defenderPlayer.getAttackedByPrivateers()) {
                  defenderPlayer.setAttackedByPrivateers(true);
                  cs.addPartial(See.only(defenderPlayer), defenderPlayer,
                 "attackedByPrivateers");
}
```

The TODO comment implies that this code reflects some sort of Shotgun Surgery that was never resolved. There are other lines of code where even the comments don't make sense.

```
  // Only update attacker with settlements that have
  // been seen, as contact can occur with its members.
  if (is.getTile().isExploredBy(this)) {
  cs.add(See.perhaps().always(this), is);
} else {
      cs.add(See.only(defenderPlayer), is);
  }...
```

Moving this code into a method with a more descriptive name would help a lot because deciphering this code is incredibly difficult. But its saying more or less that if the attacker sees a defenders tile update the GUI of the attacker. The method name could be “setDefenderTilesVisible(is, this)”.

Another problem with this code structure is the logic of the control flow. The if, then, else statements seem to be randomly placed and range from wether a pirate is involved to discerning if the attack was on a Native Capital? Refactoring and then rearranging common control behavior would ensure more readable code.

The Refactorings would continue along these lines. Removing common methods into new classes and then refactoring the methods themselves hopefully achieving a more reasonable Class size and method body size, while also removing as many enumerated types as possible. It should be noted that there are more code smells than what has been mentioned. There is a lot of duplicate code, especially in conditional expressions, isEuropean() && defenderPlayer.isEuropean() seems to be repeated numerous times. During the 'Macro' Refactoring (Large Class, Long Method) the Micro Code Smells should become more apparent, but the initial refactoring would have to focus on the Macro Level where the control and logic should be separated into smaller more cohesive classes in order to create a more readable and easier server structure to debug, update and maintain.

References

[1] http://www.freecol.org/history.html

[2]http://stackoverflow.com/questions/70324/java-inner-class-and-static-nested-class