FreeCol

Refactoring the Strategy Game FreeCol

SOEN 6471 - Concordia University

2013

Ede Cameron [2952270]

I have compiled and assessed this project a believe it to be of a reasonable size for a term project.

Ede Cameron

Project Proposal - Milestone 1

Project Description

FreeCol is a turn based strategy game. Its design is similar to Civilization, where players start with a few colonizers and have to build up a colony, the larger the colony obviously the more powerful it becomes. The game can be played in singleplayer or multiplayer mode. In singleplayer mode, a player plays against a computer, in multiplayer mode a player joins, resumes or starts a game with players online. Games can also be run by players using their own computer as the client as well as the server allowing other players to connect to the game through a specified port. Games can be public or private. There are different map choices terrain can be redefined ie mountain, forest, as well as the size and basic shape of the map.[1]

The game is written in Java and is designed using the client- server model. It is for this reason that the project interests us. The internet and internet based applications, “Web Apps” have become one of the predominant business models for software design today. Online Gaming been more predominant today than ever before.[2]

FreeCol was originally released in 2003, using C++ for the server and Java for the client, but during later releases became a Pure Java Application. The Freecol website lists five primary developers P. Bizzarri, S. Grenborgen, M. Pope, P. Rodriguez and M. Vehers. There are however numerous other credits on the FreeCol website for musical score, graphic design and translations for the gameplay.[3] FreeCol is still under development and is now at version 0.10.7.[4]

Project Size and Scope

The FreeCol website gives all relevant statistics for the game on the web page "Current Status" the number of lines of code in FreeCol is 75128. Moreover, it contains 665 classes, 30 interfaces and 6821 methods. The number of lines of code is quite large, too large to be taken as a term project, but has been designed with many small modules to separate the different aspects of the game. It would not be possible to refactor the whole program, in the time given. It is perhaps more realistic to take a specific module and focus on it for the term. After reading the source code I have decided to refactor some aspects of the server module. Specifically; The Generator the module which builds the map, The Model which defines the design, and the Net Module which generates and maintains socket (internet) communication. The total lines of code for these modules is 10, 480 which I feel is a good size for my Term Project

Group Members

Ede Cameron - Presently enrolled in Concordia's Masters Program (thesis) in Computer Science, specializing in Parallel Programming. My Thesis focuses on parallelization techniques for real-time audio specifically written to run on heterogeneous parallel systems, using both the CPU and GPU processor elements.

Use Cases and the Domain

Milestone 2

Personas, Actors, and Stakeholders

Main Persona - Judy the avid Strategy Game player and has joined a small community of players that are serious about their strategy games. She wants to play not only single player games, to hone her skills but also wants to play online with friends in private online games or on public servers provided by the Community of game players. FreeCol is such a game, it is a turn based strategy game, similar to Colonization and the famous Civilization. It differs from these because it can be played in single player mode or multiplayer mode, online, which is what attracts Judy to FreeCol. Judy also wants to introduce this game to her other friends. She can do this now with FreeCol by inviting people to play with her online in Private games where there is less pressure to compete. In other words games can be played for fun.

The primary actors in FreeCol are the players and the online hosts of the game. It is the Players that play the game, report bugs [5] and express to the developers concerns about the game. The game play and rules of the game are thoroughly tested by the players. The Host of a game deploys a game and invites other players to join the game. The game is deployed on the hosts computer. It is the hosts computer that will build the map and keep track of the overall game play, including the validation of player moves and keeping track of all the players statistics during game play. The Players play the game, these players may also include computer AI Players the game is played in SIngle Player Mode. These AI Players are programmed to play as close to a human player as possible. There are really no other stakeholders of FreeCol, the players and the game hosts are the people involved in playing the game.

Use Cases

Start New Game Single Player.

Player launches FreeCol by clicking on FreeCol Icon
System displays welcome message and prompts the player to make the choice : Single or Multiplayer game mode.
Player selects for Single Player.
Server prompts user to choose Create New Game or Continue Saved Game.
The player selects to start a new game
System returns success message that a new game has been created

Start New Private Game - Multiplayer

Use Case assumes that Freecol is already launched.

Game prompts user to choose single/multiplayer mode.
Player chooses multiplayer option.
Game prompts user to choose Create New Game or Join Existing Game.
User selects Create New Game.
Game prompts host to choose public or private game.
Host chooses private game.
Game prompts user to create a password for the game.
Host enters the password.
Game prompts user for IP address.
User enters IP address.
Server authenticate/validates the host.
Server opens defualt port with IP Address supplied by the user and returns success.

Join Private Game - Multiplayer

Assumes Player has already launched FreeCol and has selected multiplayer mode

Game prompts player to Create a New Game or Join Existing Game
Player chooses to Join Existing Game option
Game prompts Player for the IP Address of the Multiplayer game
User enters the IP Address
Game searches for Server IP Address - on success
Game prompts player for the password.
Player enters the password.
Server validates the password
Server opens the port with IP address and validates the user to play a new private multiplayer game.

Take Turn - Attack Resource

Assumes Player has already launched FreeCol a has started game play
Player clicks on his Soldier Icon
Game highlights Soldier Icon
Player clicks on opposing players resource
Game highlights opposing players resource
Player selects attack resource form action menu
Server validates that player can attack the resource and informs defender of the opponents attack
Sends message back to Attacker informing them that the attack has begun.
The Game gets information of both Players Statistics and calculates the outcome of the attack
Returns the result of the attack to both players.

Take Turn - Player Trading Turn

Assumes Player has already launched FreeCol and has started game play

Player requests to trade resources with another Player
The Server validates the user has the resources to trade and forwards trade message to other Player.
Player Two responds - agreeing to the trade.
The server forwards response to Player One

Player One validates the trade
The Game updates each Players individual resources and sends success message to both Players

Generate Random Map - Single Player

Assumes Player has already launched FreeCol and player has selected to play a new single player game

Game prompts player to build a map with the options;
manually creating a map,
choosing a default map,
creating a random map
Player chooses to create a random map
Game returns success to player when the map is created stating map was created and prompts player to click on “continue” button to start the game
Player presses continue button to go to the Main GUI.

Reload Single Player Saved Game

Assumes Player has already launched FreeCol and player has selected to play a single player gam

Game Prompts Player to start New Game or Continue Saved Game
Player chooses to Continue Saved Game
Game server locates the data of Saved Game and loads game onto player Console starting the game

Take Turn- Build Port

Assumes Player is already playing FreeCol - Single or Multiplayer

Game Prompts player to take turn
Player selects Build from Drop down Menu
Game returns Types of Build Options
Player Selects “Build Port”
Game validates players resources to make sure player has enough resources to build a Port
Returns success message to player removes resources from player stats and reflects this in the

Player info.
Updates the GUI to reflect building port completed.

Take Turn- Build Building

Assumes Player is already playing FreeCol - Single or Multiplayer
Player clicks on empty map tile.
Game highlights empty map tile
Player selects option Build Fort
Game validates players resources to make sure player has enough resources to build a Fort.
Returns success message to player removes resources from player stats and reflects this in the Players info.
Updates the GUI to reflect building completed.

Multiplayer : Quit Game

Assumes that player is playing a Multiplayer game.

Player selects the option Quit Game from drop down menu bar
Game responds prompting Player Player to Save and Quit Game or Cancel
Player Selects Quit Game Option
The system successfully saves the game posts success message to player, but at the same time posts status of player to other connected players and exits player from the game.

UML Domain Diagram

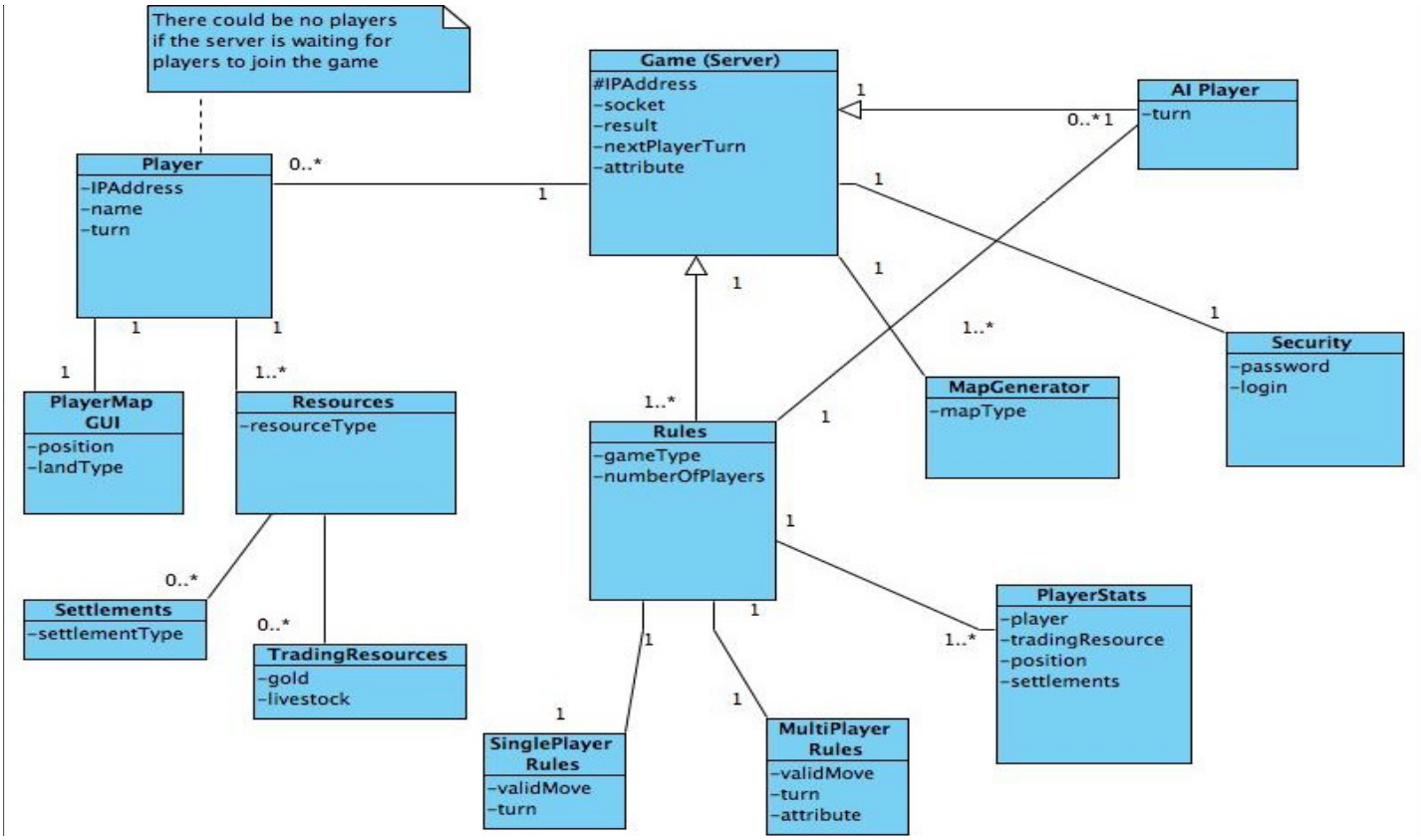

Figure 2.1 Conceptual Domain Diagram of the Game FreeCol

FreeCol is based of the Client Server Model and separates the concerns of Player and host in these regards. The Player will have a View of the game represented by a Graphical User Interface (GUI) and all game play is done through this GUI. The Player also

has resources such as settlements and money represented by the Resources Module. The Server contains all other information concerning the game, although the player may understand the rules the rules are validated on the server side. The Server also generates a map, though this is a conceptual map, not a map represented by a GUI, the map is passed to the player as the player moves in the game. The Server also validates moves and contains the rules of the game, as well as controlling the AI Players during single player games. The Server of course is also responsible for security making sure that no incoming connections or actions are unsolicited.

The FreeCol Server Architecture

Milestone 3

The FreeCol Server

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.

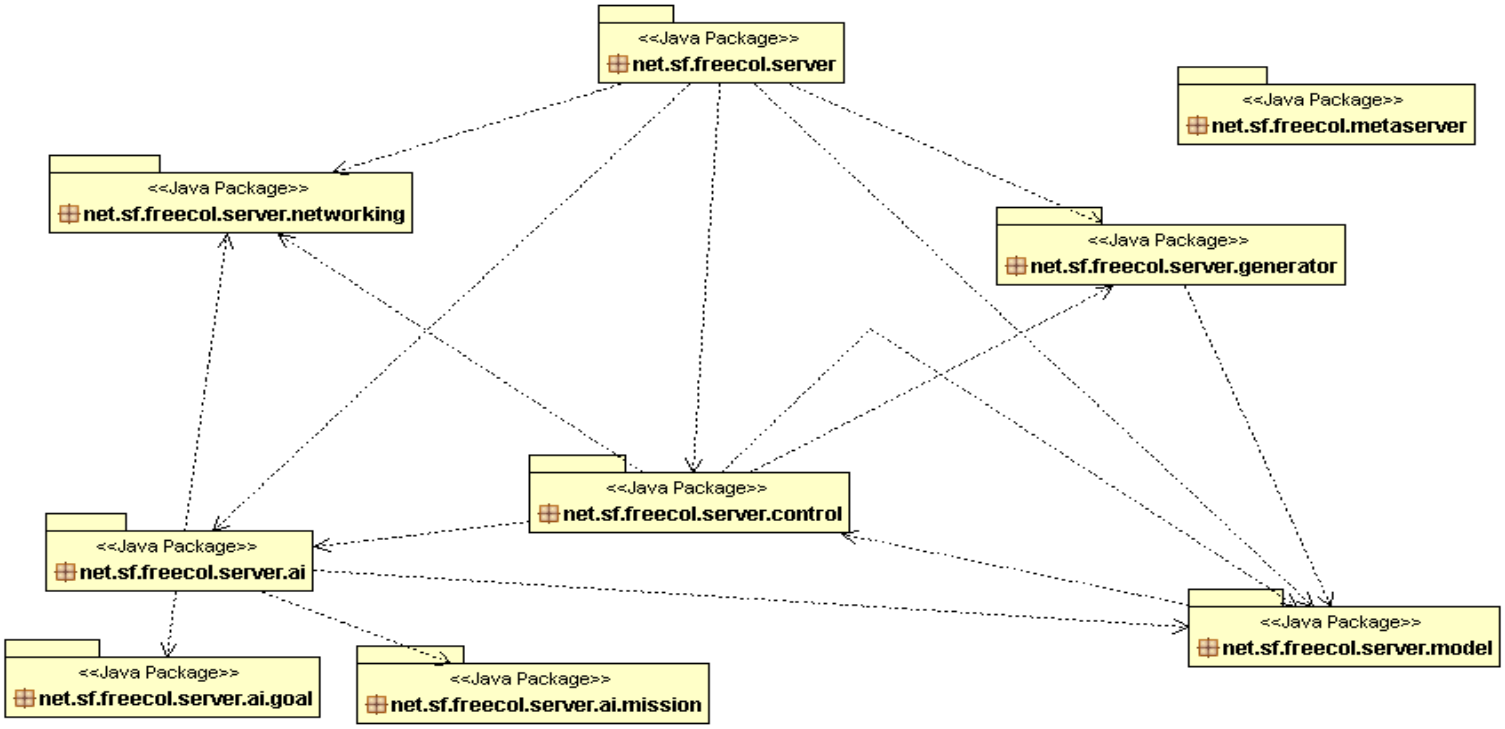

Figure 3.1 The Package Diagram for the server aspects of FreeCol derived from the Eclipse Plugin ObjectAid.

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

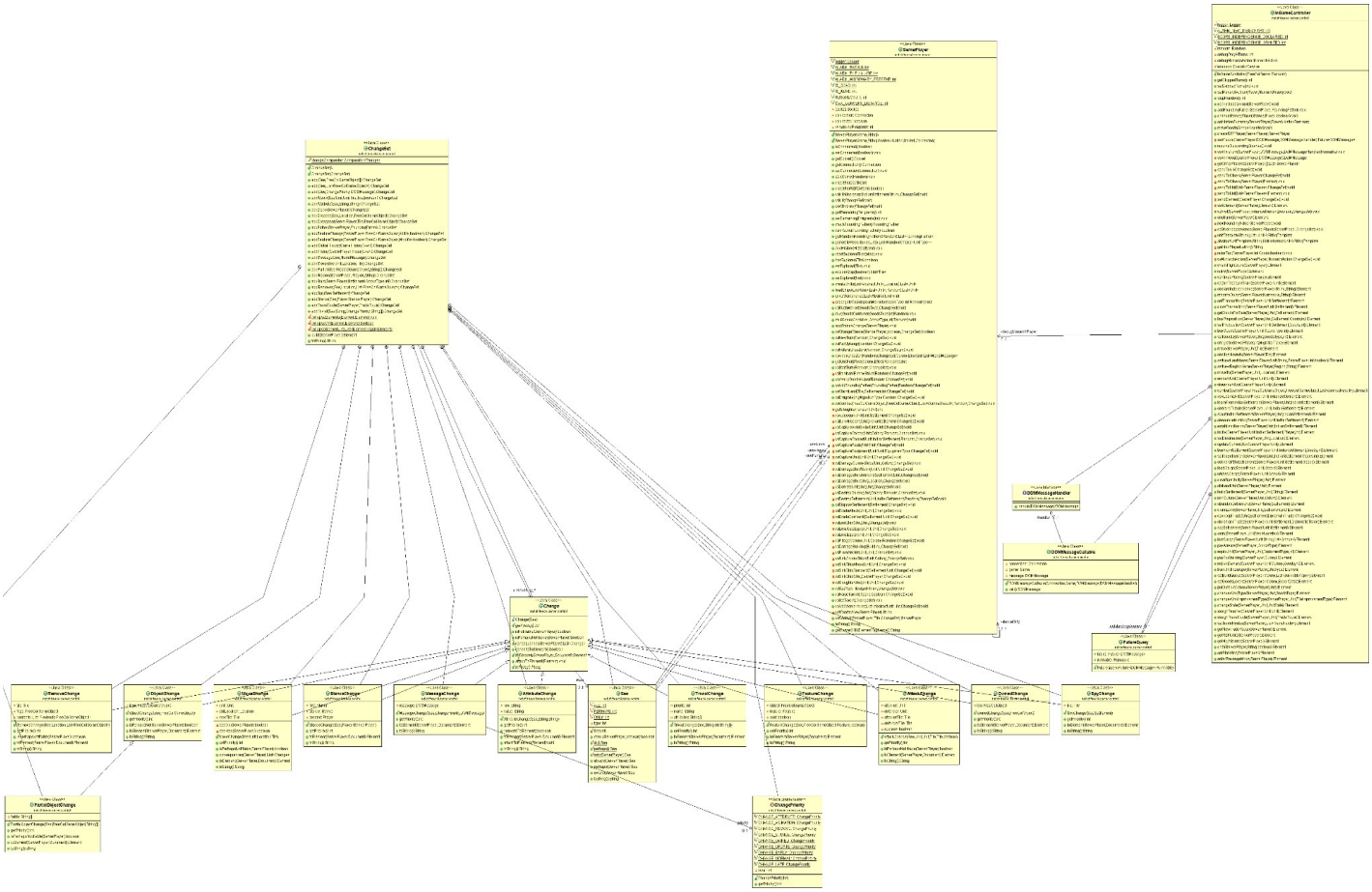

Figure 3.2 Three Large Classes found in the model and Control packages. From Left to right ChangeSet, ServerPlayer and the InGameController. Notice the ChangeSet Inner Classes.

In first versions of FreeCol, the server was written in C++ and was later ported to Java[5]. 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 may be justified, where a class is instantiated only through the parent class, or the inner classes depends heavily on the variables of the parent.[6] 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. Extracting these classes, using polymorphism and inheritance would make the code structure 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 where 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 ServerPlayerExplored 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 can be refactored.

The method csCombat 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 be rewritten 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 whether 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.

Patterns and Refactoring

MileStone 4

Patterns

There is one Pattern within FreeCol that is apparent just from the names given to the FreeCol Packages. In the Server Packages there is the Model Package and the Control Package. In the Common Packages there is also a Model Package, and in the Client there is a Control Package. It can be assumed that these Packages are intentionally named to suggest the use of the Design Pattern, Model-Control-View (MVC) which is a widely used Design Pattern for Client - Server based applications.

The View is the visual aspect of an application accessible to the user and generally refers to the Graphical User Interface (GUI). Within the MVC Pattern is also the Controller Pattern which delegates diverse commands from the View to the different implementation rules of an application aspects contained within the Model.

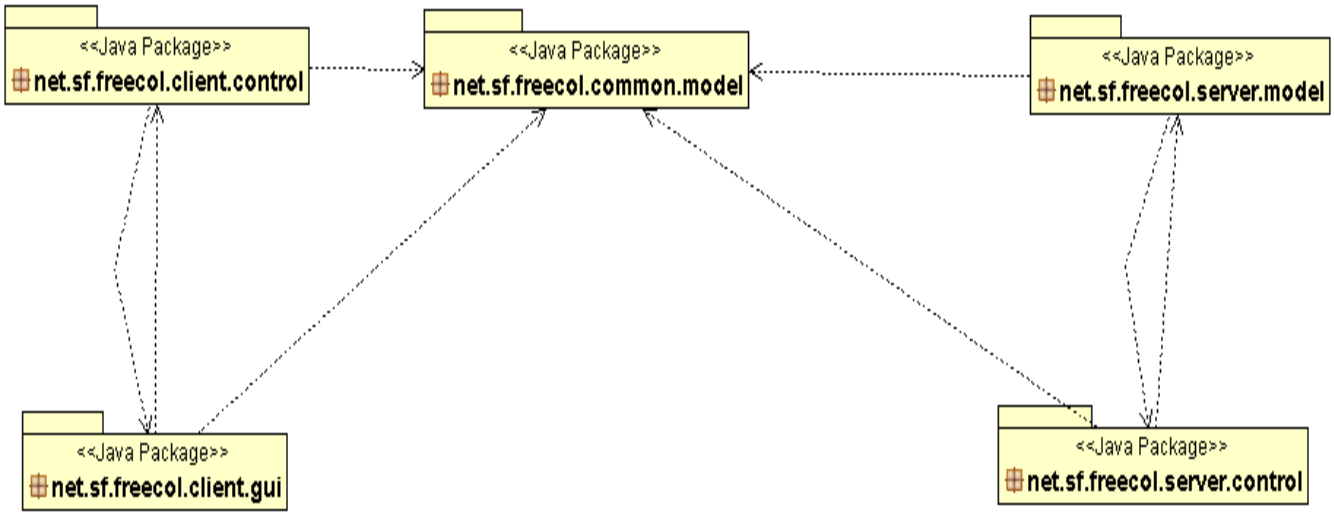

Figure 4.1 FreeCol Model Control and GUI (View) Packages - Showing dependencies

between the Packages.

The Diagram above shows how the developers have implemented the MCV Pattern. There are some discrepancies between the Pattern description and the implementation, there is a relationship between the GUI and the Common Model that shouldn't be there, but this dependency is to set up international language features for the game, but this seems to be the only noticeable dependency, the rest of the implementation seems to follow the Model Control View Pattern.

The main advantage to using an MCV Pattern is that the View can be decoupled from the Model and the Controller. In other words the Model should have no knowledge of the View. This means that any change to the GUI (View) should never affect the model and any change to the Model should have no affect on the user's view of an application though will affect behavior which is the responsibility of the model.

In general terms this means that a button represented in the GUI could initially be set to add one to an already set value and the new value is displayed on the screen.The model is written at some point to add two instead of one, because the view and model are decoupled the only code that needs to be changed will be in the model the view will stay the same but display a different number.

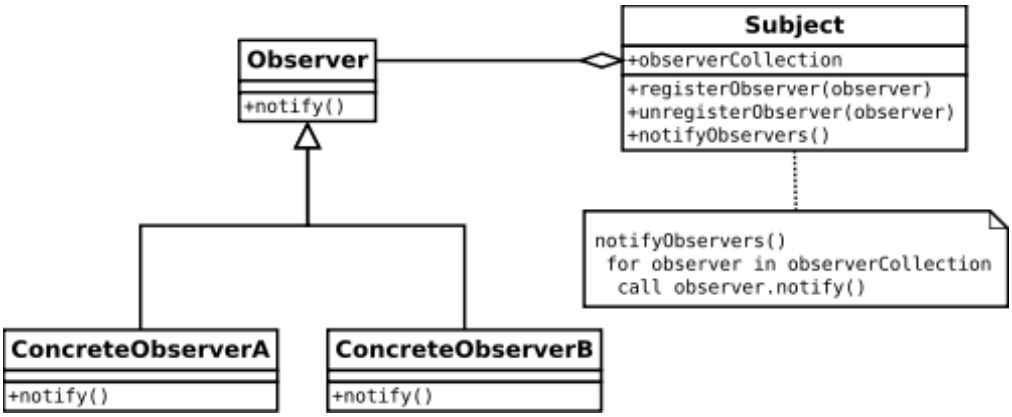

Figure 4.2 The Observer Pattern [8]

"The Observer pattern defines and maintains a dependency between objects. The classic example of Observer is in Smalltalk Model/View/Controller, where all views of the model are notified whenever the model's state changes." [9]

There is however another Pattern which is integral to the MCV Pattern and is also used in FreeCol, and that is the Observer Pattern. in fact the Java GUI Library is based partially on

the Observer Pattern because the Java GUI is Action Event based, in other words the GUI provides listeners (Observers) that react to events that are triggered through user-GUI interaction. This is an integral aspect of how the events triggered in the View are translated by the Controller to invoke methods and pass data to the Model which in turn will update the view. These listeners are called Action- Event Listeners and are generally window event listeners triggered by a mouse or keyboard in the case of the FreeCol user interface.

```
public void fireActionEvent(ActionEvent event){
    ActionListener[] listeners = listenerList.getListeners(ActionListener.class);
    for(int i=0; i < listenerList.getListenerCount(); i++){
       listeners[i].actionPerformed(event);
    }
```

The Code above is one one of FreeCol's implementations of the Java Event Listeners.
In this case from the TerrainCursor Class in the Client GUI Package. The cursor represents the mouse of the Player, is then added to the MapViewer Class. The cursor when it receives input from the game (there are several other Classes involved in triggering an event) will fire events that are in the ActionListener List this very much the Observer Pattern, the listeners been the subscribers and the Terrain GUI been the publisher. These events are separated from the publisher because of the Model Control View architecture the publisher in this case controls when the events are published, but has no idea of what these events may be.

Refactoring FreeCol - Large Class

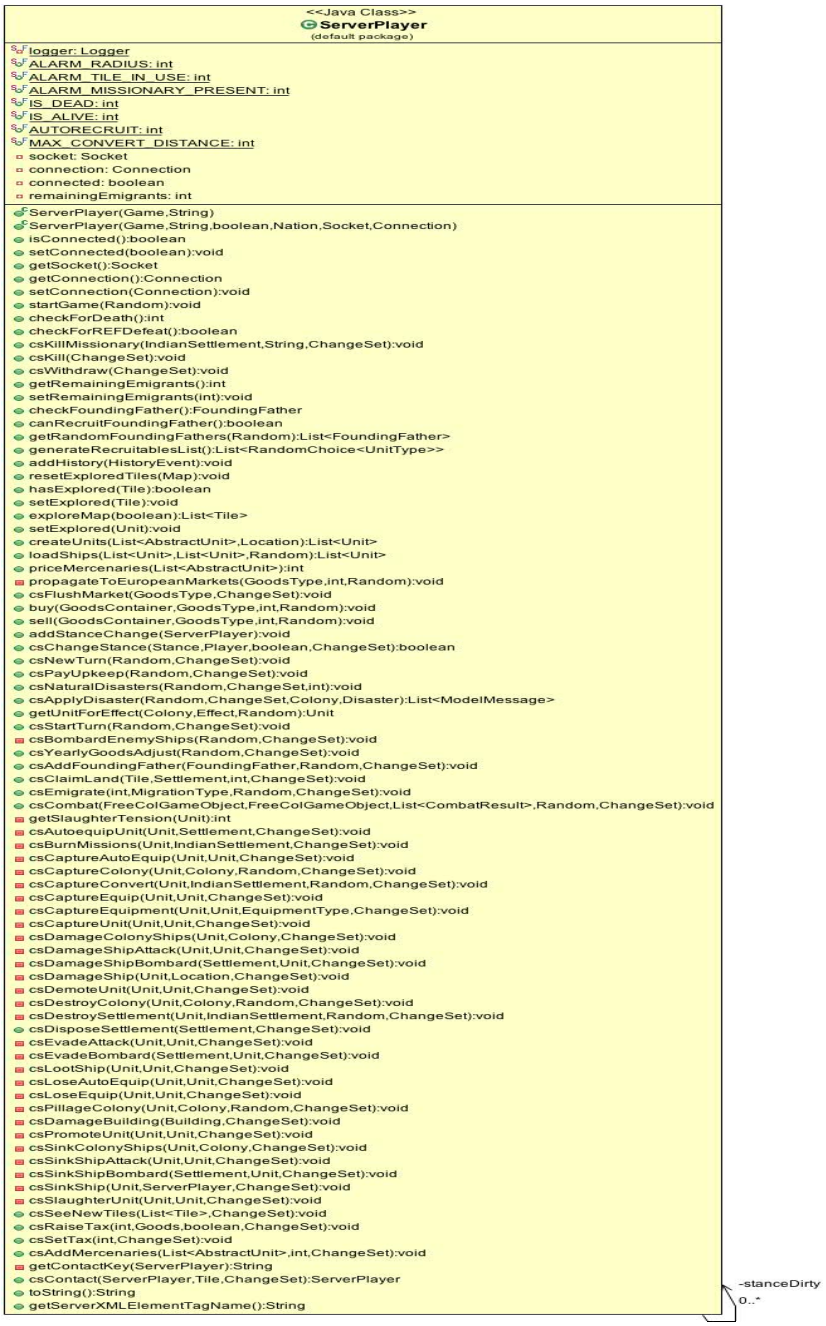

Figure 4.3 The ServerPlayer Class in the FreeCol Server Model package.

The Server Packages of FreeCol from a design perspective are from a Package view

well organized, but within these packages there are numerous problems. First and foremost are the overly large Classes, from Milestone Three we saw several Classes that were well over 1500 lines of code. One Class specifically the ServerPlayer Class is over 3000 lines and contains over 70 methods. These methods are all connected to the Player Object, but are very diverse in nature, and some of these methods are over 400 lines long. Although Extract Class and Extract method could rectify this problem, there rests the question of where to start.

It should be noted that this refactoring is quite long there were numerous reasons to do all these refactorings but the primary reason was to show that although daunting cleaning up this class was possible. There are three basic refactorings here each one done once or twice to show its relevancy. Extract Class from ServerPlayer, move related methods into new class, ServerPlayerCombat and within this class extract methods from large methods and finally extract classes from ServerPlayerCombat into polymorphic subclasses.It should also be noted that when starting these refactorings the possible refactorings in Milestone 3 were set aside in order to pursue the refactoring methods found during this milestone

Patchset 1/15 Refactor Class ServerPlayer

Create new Class ServerPlayerCombat

There are themes within the ServerPlayer Class such as combat, or trading. The ServerPlayer Class has several long methods. one method specifically is csCombat. This method has more than 500 lines of code and seems a good place to start the refactoring in that it represents the combat theme. In order to move this method into its own Class the Class needs to be created. The first step was to add the Class ServerPlayerCombat.java to the Package server.model, the same location as ServerPlayer.java

Patchset 2/15 Extract Method

Move Method csCombat into the ServerPlayerCombat Class

We were told to think small but moving the method csCombat out of the ServerPlayer Class broke numerous dependencies. This commit moves the method into the new Class. Eclipse fortunately auto generated most of the import statements but the errors in the New Class method were numerous. The commit reflects the move and the broken dependencies

Patchset 3/15 Handling Dependencies

Start recognizing and handling dependencies between ServerPlayer and ServerPlayerCombat

Although there are three steps in this refactoring they follow the logic of calling the new class from ServerPlayer and what the new class requires from the old class, in the hierarchy chain. The next refactorings recognize some interesting dependencies and try to rectify these problems

The first step in the refactoring was to add a constructor to the ServerPlayer Class that would call the ServerPlayerCombat class. There was never the intention of trying to separate the two Classes completely but have the csCombat method in ServerPlayer call the delegated method in ServerPlayerCombat. The inheritance hierarchy was then added so ServerPlayerCombat would extend the ServerPlayer and implement the ServerModelObject, like the parent class. Then the two methods getServerXMLElementTagName and csNewTurn were overridden after implementing the ServerModelObject in the new class.

PatchSet 4/15 Changing method visibility

The next refactoring was changing the methods called from ServerPlayerCombat that were still in the ServerPlayer Class from private to protected. After this was done a build was attempted and succeeded, but on close inspection of this refactoring, it seemed that this fix didn't resolve the inherent Large Class problem. It also was apparent after looking at the code in ServerPlayerCombat that the classes called from ServerPlayerCombat were only called from ServerPlayerCombat.

Patchset 5/15 Small fixes

Several small fixes were made to the ServerPlayerCombat Class including making a global instance of the ServerPlayerCombat Class in the ServerPlayer instead of a local instance in the method body, removing and adding some comments. It was at this stage that a build of FreeCol was made and succeeded. It was at this point that removing other methods from ServerPlayer commenced.

Patchset 6/15 Reducing the size of the ServerPlayer Class

Extract Method GetSlaughterTension from ServerPlayer to ServerPlayerCombat.

If the idea of this refactoring process was to reduce the size of the ServerPlayer Class then changing methods from private to protected doesn't really solve the problem and in fact creates a new code smell Feature Envy. The now protected classes which for the most part were now been called from ServerPlayerCombat.

This Patch represents the initial test to extract these methods. The method GetSlaughterTension was extracted from ServerPlayer to ServerPlayerCombat. The build was tested and succeeded.

The methods now in ServerPlayer whose visibilty was changed from private to protected really belong either in the ServerPlayerCombat Class or in Classes extended from ServerPlayerCombat. In fact the refactoring process will be to extract the classes into ServerPlayerCombat and then using polymorphism move them into related Classes extended from ServerPlayerCombat

This one method move of GetSlaughterTension was to see if there were dependencies involved when moving the method into the ServerPlayerCombat Class. There were no errors reported, so decided to extract all relevant methods from ServerPlayer into ServerPlayerCombat.

Patchset 7/15 Massive Moves

Since the the refactoring in 6/17 worked with few problems all other protected methods changed during the refactoring of 4/17 were moved into the ServerPlayerCombat Class, and although this commit has over 1400 lines in the patch file (Patch7-15.patch), the moves were done one by one and tested but the commit condenses what would have been numerous repetitious commits into one large but tested commit. The ServerPlayer Class is reduced by over 500 lines of code but this now means that ServerPlayerCombat has increased by 500 lines.

Patchset 8/15 JUnit testing

Change constructor in ServerPlayerCombat to resolve JUnit Test Failure.

Although FreeCol was successfully building with what seemed to be no errors, the JUnit tests that are included in the FreeCol Source were failing and up until this point there seemed to be little point in using these tests, until after the massive move of methods the JUnit tests seemed warranted. Of course on first run there were several errors and the JUnit Tests became a required step in building and testing after this and I admit should have been done from the start.

Error thrown from logger duplicate ID. Solved by changing the constructor for ServerPlayerCombat to only pass the game and not the ID (each class needs a unique Id, but couldn't be passed directly). Retested and recompiled

Patchset 9/15 Delete Old Method csCombat From ServerPlayer

After Unit Testing the new class ServerPlayerCombat, the old method csCombat and other methods already moved into ServerPlayerCombat (although already commented out) were deleted from ServerPlayer - again passed JUnit test and recompiled.

Patchset 10/15 - Extract Remaining Methods

All other protected methods moved to ServerPlayerCombat
Final move of all remaining protected methods called in ServerPlayerCombat from ServerPlayer into ServerPlayerCombat. Tested, build succeeded

Patchset 11/15 Extract Method from csCombat in ServerPlayerCombat

Create new smaller method handleStanceAndTension to handle same method body code.

These next two refactorings move some of the code from the method csCombat into their own methods, in an attempt to reduce the size of csCombat. The object is just to show the refactorings needed within the long method.the visibility changed of several booleans and integer values for tension from local to method to private global methods in ServerPlayerCombat Class. Again JUnit tested.

Patchset 12/15 Extract Method

moveAttacker Method created - same process as handleStanceAndTension

Patchset 13/15 Create Class

Start replacing enumerated types with polymorphism
Created Class SlaughterUnit

These final refactorings are to now start reducing the size of the class ServerPlayerCombat. There are numerous calls to methods within the ServerPlayerCombat Class form enumerated types these enums are been sent from the client side classes of FreeCol and would be complex to refactor but each method call within the enumerations can be moved into separate classes to reduce the size of ServerPlayerCombat.

Patchset 14/15 Create Class

Created Class BurnMissions
Same process as Create Class SlaughterUnit with the same justifications

Patchset 15/15 Delete Method

Delete method burnMissions from ServerPlayerCombat - run JUnit tests and compile.

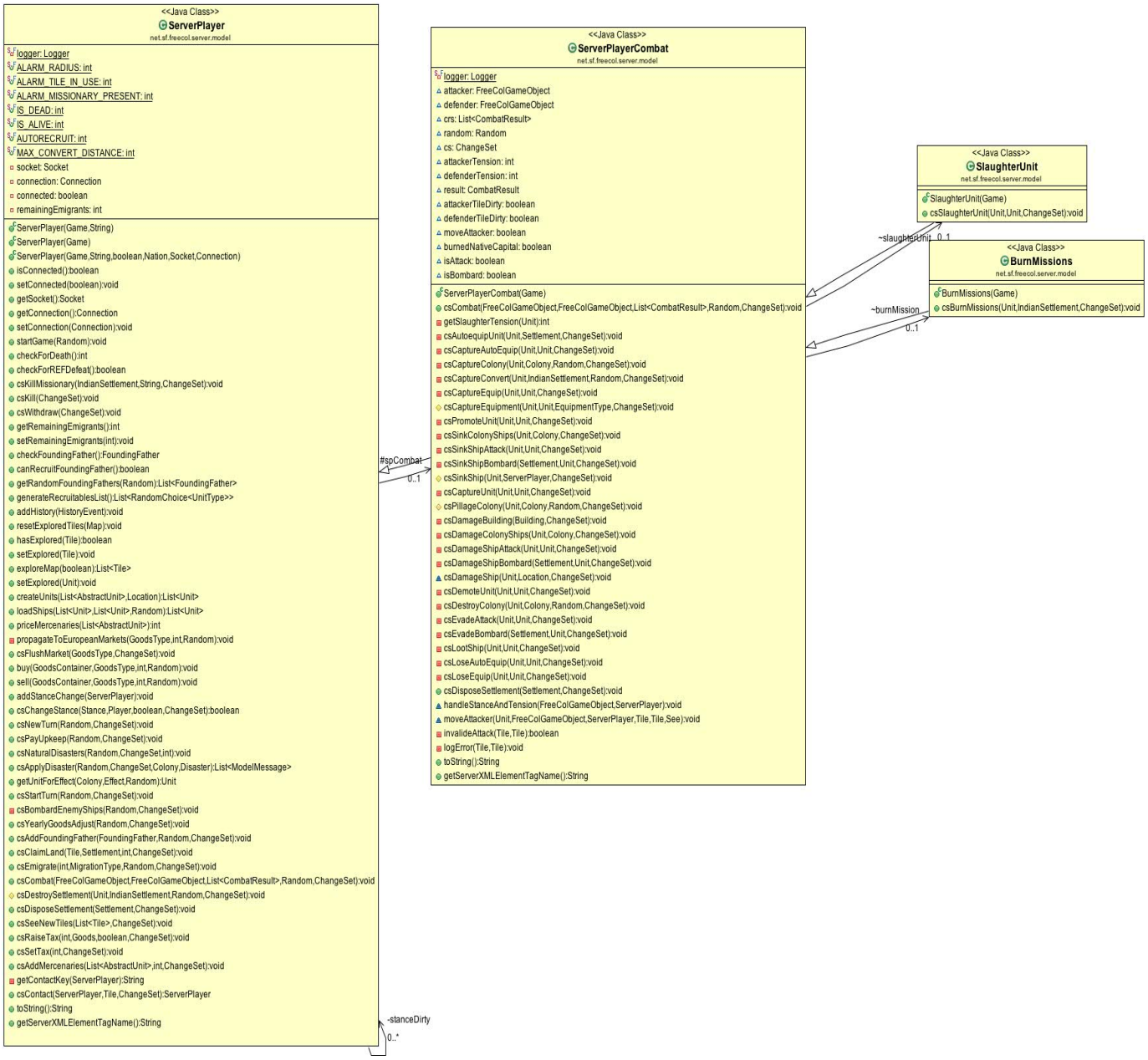

Figure 4.3 - The Refactored Class ServerPlayer. If the refactoring logic was continued we would see more smaller classes in the hierarchy but the principles of this refactoring process have already reduced the size of both classes

The refactoring would continue this way until there would be numerous small classes refactored from ServerPlayerCombat and more methods extracted from csCombat. The

refactoring would then start again in the ServerPlayer to remove more common code into another Classes. These refactoring techniques would not only make the Server side code more readable it would also allow for an more extensible code where new rules could be added as classes and not affect the core code base of the ServerPlayer.

References

[1] FreeCol Manual
http://www.freecol.org/docs/FreeCol.html#x1-40001.2

[2]TechWench - Online Gaming Increasing In Popularity. September 3, 2012

http://www.techwench.com/online-gaming-increasing-in-popularity/

[3]FreeCol Website - History
http://www.freecol.org/history.html

[4]FreeCol Website - Current Status
http://www.freecol.org/status.html

[5] There is a need in the Open Source Community for users to report bugs as there is little resources to do thorough tests, and FreeCol is no exception FreeCol Players can report bugs to SourceForge at the address below

http://sourceforge.net/p/freecol/bugs/

see also SuperCollider Issue Tracker
https://github.com/supercollider/supercollider/issues

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

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

[8] http://en.wikipedia.org/wiki/Observer_pattern

[9] Erich Gamma.. [et al.]. Design Patterns: Elements of Reusable Object-Oriented Software,

Boston, Addison-Wesley, 1994