

SOEN 6471 Professor Peter Rigby

FREECOL GAME

(Milestone 4)

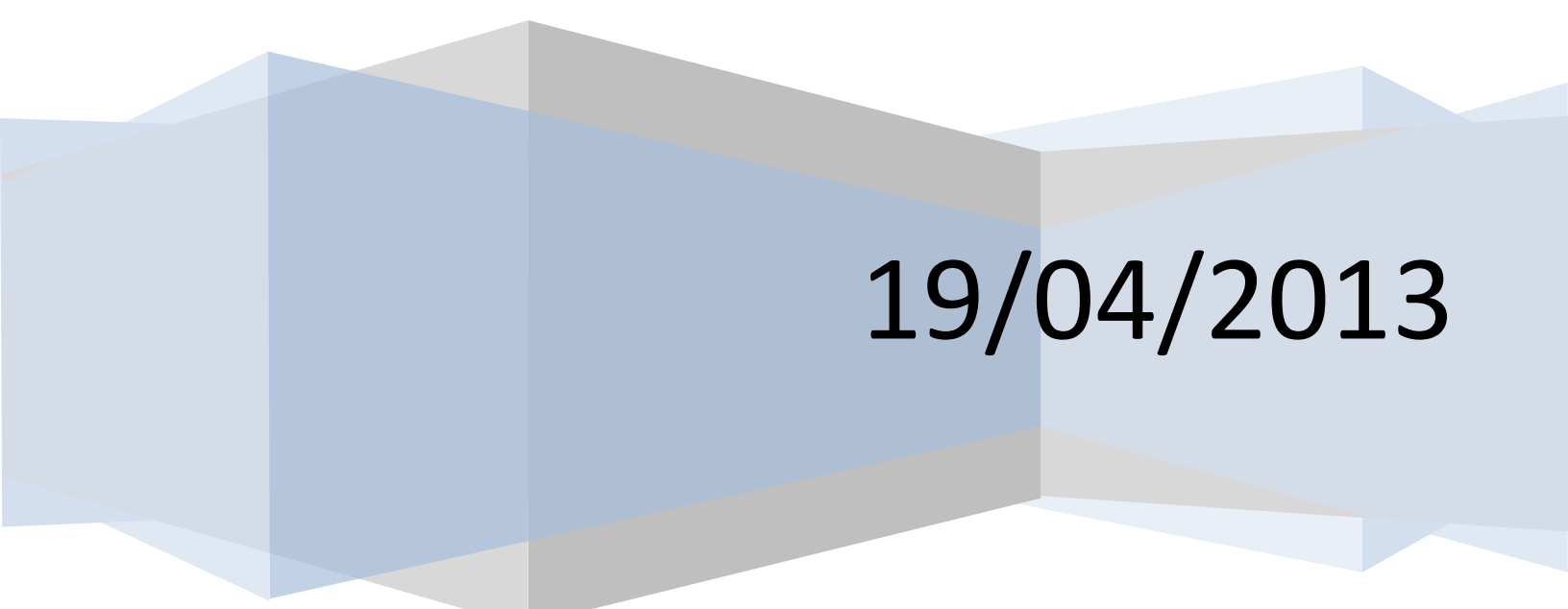
Refactoring & Design Patterns

Ronak Patel [6483607]

Hardev Goraya [6446019]

Navrang Singh [6447430]

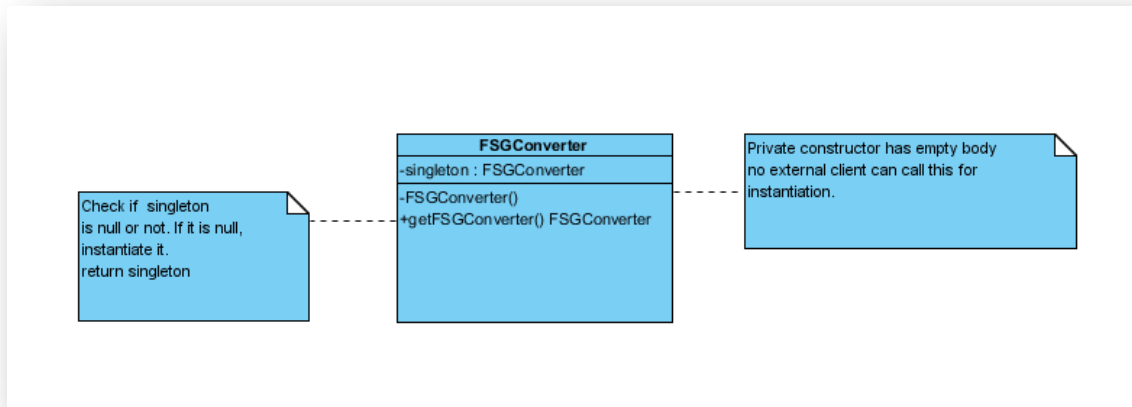
Vishal Mittal [6425240]



19/04/2013

DESIGN PATTERNS

Singleton Pattern (Hardev Singh 6446019)



Singleton pattern ensure that a class has one instance and provide global point of access to that instance. It is present in FSGConverter class of `net.sf.freecol.tools` package which contain tools directly related to Freecol. This class is responsible for converting freecol Save games (fig- files). Here a Static method FSGConverter gets an object for converting Freecol savegames and return Singleton object. The Singleton pattern here ensures that only one instance is made for the class FSGConverter. In Freecol game at game saving stage in addition to creating single Instance, it also provide a globally available method to get it. Since the Singleton is initialized only when it's first accessed, it won't be instantiated at all if the game never asks for it.

Description:

1. Private static attribute in the "single instance" class:

```
public class FSGConverter {  
    private static FSGConverter singleton;  
    .....  
}
```

Singleton object of this class. Which is responsible for creation, initialization, access, and enforcement.

2. Public static accessor function in the class:

```
public static FSGConverter getFSGConverter() {  
    if (singleton == null) {  
        singleton = new FSGConverter();  
    }  
}
```

```
return singleton; }
```

Public static member function that encapsulates all initialization code, and provides access to the instance.

The “static member function accessor” approach will not support subclassing of the FSGConverter class. Here lazy initialization is done i.e. creation on first use.

3. Constructors to be protected or private.

```
private FSGConverter() {....}
```

4. Clients may only use the accessor function to manipulate the Singleton.

```
public static void main(String[] args) {.....  
try {  
    FSGConverter fsgc = FSGConverter.getFSGConverter();  
    fsgc.convertToXML(in, out);  
} catch (IOException e) {.....  
}
```

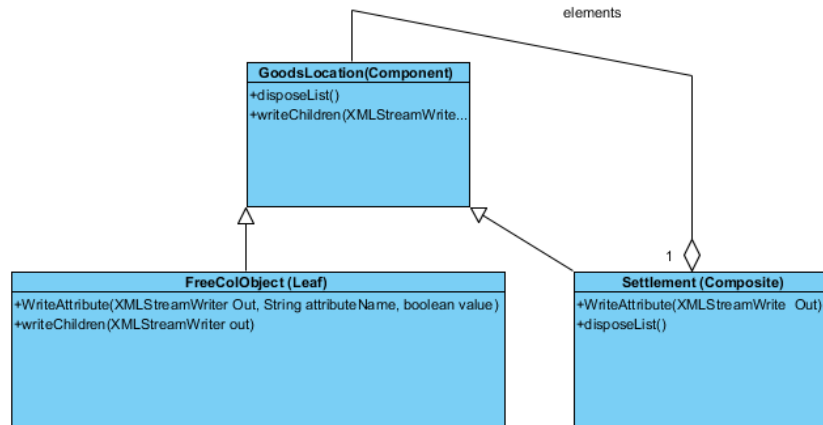
Main class is an entry point for converting FreeCol Savegames.

The Reverse Engineering tool used for detection is web of patterns plug-in of eclipse. It is a toolkit that facilitates the sharing of knowledge about software design.

References:

1. http://sourcemaking.com/design_patterns/singleton
2. <http://www.youtube.com/watch?v=NZaXM67fxbs>
3. <http://www.headfirstlabs.com/books/hfdp/>

Composite Pattern (Ronak Patel 6483607)



A composite design pattern is a pattern that allows you to treat individual objects and composition of objects uniformly. They are used to represent part-whole hierarchies. It is present in the net.sf.freecol.common.model package where GoodsLocation.java acts as Component, Settlement.java act as Composite and FreeColObject.java as the Leaf. The parts of source code that explains this design pattern is shown below:

Description:

Component: Goodslocation

Component: It is basically the abstraction for all type of components which also includes the composite and the leaf ones. It declares the interface or abstraction for objects in the composite classes.

```
public abstract class GoodsLocation extends UnitLocation {

    public List<FreeColGameObject> disposeList() {
        List<FreeColGameObject> objects = new ArrayList<FreeColGameObject>();
        if (goodsContainer != null) {...}

    }

    protected void writeChildren(XMLStreamWriter out, Player player,
                                boolean showAll, boolean toSavedGame)
        throws XMLStreamException {...}

}
```

Composite: Settlement.java

Composite: It represents components or elements which are composite. It implements the methods from the component class which here is GoodsLocation.

```
abstract public class Settlement extends GoodsLocation
    implements Named, Ownable {

    public List<FreeColGameObject> disposeList() {
        List<FreeColGameObject> objects = new
            ArrayList<FreeColGameObject>();
        .....
    }

    protected void writeAttributes(XMLStreamWriter out)
        throws XMLStreamException {.....
    }
```

Leaf: FreeColObject.java

Leaf: represents leaf objects in the composite class which is Settlement.java. It also implements all Component methods from GoodsLocation.java

```
public abstract class FreeColObject {

    public void writeAttribute(XMLStreamWriter out, String attributeName,
        boolean value) throws XMLStreamException {
        .....}

    protected void writeChildren(XMLStreamWriter out)
        throws XMLStreamException {...}
```

Here the methods in Component class i.e. `disposeList()`, `writeChildren()` are overridden in the Composite class and the Leaf class, which does not extend any class.

References:

1. http://sourcemaking.com/design_patterns/composite
2. <http://www.oodeesign.com/composite-pattern.html>
3. <http://java.dzone.com/articles/design-patterns-composite>

Reverse Engineering Tools:

As we already know, the Web of Patterns is an efficient tool that helps us to identify various design patterns such as Abstract Factory, Composite Pattern, Proxy Pattern, Adapter, Bridge pattern, etc.

The software is Open source and was downloaded from source forge. The web link for this tool is:

<http://www-ist.massey.ac.nz/wop/>

PROXY PATTERN [VISHAL MITTAL 6425240]

In the Freecol Project there are number of Pattern for example is proxy, adapter, singleton etc. The most interesting that I found in the freecol open source project is the proxy. The main idea of this pattern is to provide a placeholder for an object to control references.

Implementation:

The proxy is implemented in the freecol project in the Common.model package. These are described as below

RealSubject: It is the real object that proxy represents for example in the freecol project the settlement class in the common.model package work as realsubject in the proxy pattern. It also implement the Ownable interface that work as the subject.

```
/**
 * The super class of all settlements on the map (that is colonies and
 * indian settlements).
 */
abstract public class Settlement extends GoodsLocation
    implements Named, Ownable {
```

Subject:- It is the Interface implemented by the realsubject. The interface must be implemented by the proxy as well so that the proxy can be used in any location where realsubject can be used. In the freecol project Subject is made as Owner interface having an method getOwner(). Is is basically use for getting the Who is Owner at a particular time when game is start of a particular colony

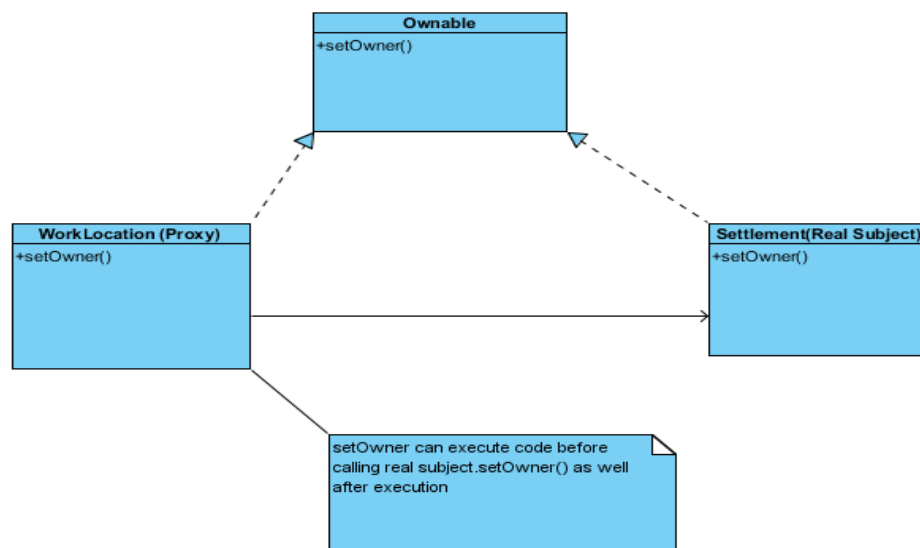
```
/**
 * Interface for objects which can be owned by a <code>Player</code>.
 * @see Player
 */
public interface Ownable {

    /**
     * Gets the owner of this <code>Ownable</code>.
     *
     * @return The <code>Player</code> controlling this
     *         {@link Ownable}.
     */
}
```

Proxy:

This class having a lot of responsibility for example It maintains a reference that allow the proxy to access the real subject and implement the same interface by the realsubject so that the proxy can be substituted for the realsubject. In the freecol Project Work location is the class in the common.model package work as the proxy it control access to the realsubject so that the proxy can be substituted for the realsubject.

```
*/  
public abstract class WorkLocation extends UnitLocation implements Ownable {  
  
    /**  
     * The colony that contains this work location.  
     */  
    private Colony colony;
```



Tool: The tool used for finding the design pattern is the web of pattern. It is the plugin that basically detect the design pattern in any java project.

References:

- 1) http://sourcemaking.com/design_patterns/proxy
- 2) <http://www.oodeesign.com/proxy-pattern.html>
- 3) Head first "Design pattern"

Composite Pattern:(Navrang 6447430)

In the Freecol project there are only few patterns found. The most interesting pattern that I found is composite pattern. The main intent of this pattern is to compose objects into tree structures to represent part – whole hierarchies.

Implementation:

Freecol composite pattern is structured as follows:

Component: Component is the abstraction for leafs and composites. It defines the Interface that must be implemented by the object in the composition. For example: AIObjct is the abstract class in the *Server.ai* package.

```
/**
 * An <code>AIObjct</code> contains AI-related information and methods.
 * Each <code>FreeColGameObject</code>, that is owned by an AI-controlled
 * player, can have a single <code>AIObjct</code> attached to it.
 */
public abstract class AIObjct extends FreeColObject {
```

Leaf: Leafs are objects that have no children. They implement services described by the component interface. For example FreeColObject class in the Common.model package overrides the methods with writeAttribute() and getspecification() method.

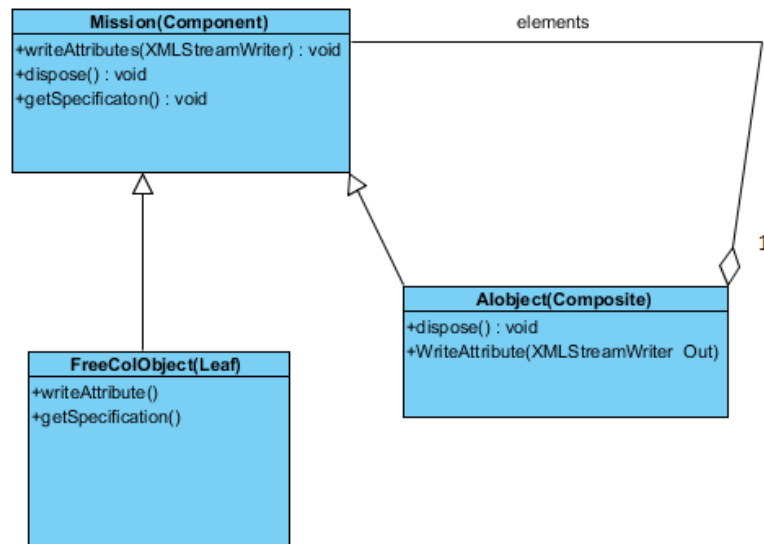
```
public abstract class FreeColObject {
```

```
/**
 * Describe <code>getSpecification</code> method here.
 *
 * @return a <code>Specification</code> value
 */
public Specification getSpecification() {
    return specification;
}
```

Composite: A composite Stores child components in addition to implementing methods defined by the component interface. Composites implement methods defined in the component interface by delegating to child components. In addition composites provide additional methods for adding,

removing, as well as getting components for example Mission class in the server.ai.mission package works as a composite for this pattern.

```
/**
 * A mission describes what a unit should do; attack, build colony,
 * wander etc. Every {@link AIUnit} should have a mission. By
 * extending this class, you create different missions.
 */
public abstract class Mission extends AIObject {
```



In this composite pattern, freecol developer use the tree structure for the war in the game. They set a mission for their particular objects that define in the freecolobject and Aobject class and later they get this description by using `getSpecification()` method and even to write the specification of this war using the `writeAttribute()` method in the **FreeColObject** class.

Tool: The Tool that is used to find the design pattern is "WebOfPatterns". The WebOfPatterns (WOP) is a toolkit that facilitates the sharing of knowledge about software design.

References:

- 1) <http://www.oodeesign.com/composite-pattern.html>
- 2) Head First "Design Pattern" Book.
- 3) http://sourcemaking.com/design_patterns/composite

REFACTORING IMPLEMENTATION

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. With this we decrease the responsibility ,complexity and increased the cohesion of the original class and make new class as *ServerConnection*.

Mechanics:

1. Decide how to split the responsibilities:

In this class we can separate fields like Socket, Connection, Boolean, List<ServerPlayer> into separate class named as *ServerConnection.java*

2. New Class *ServerConnection.java* appear as :

In this class we defined data members as private.The getter setters are defined here.

```
public class ServerConnection{
    private Socket socket;
    private Connection connection;
    private boolean connected;
    private List<ServerPlayer> stanceDirty;
    public ServerPlayerData(boolean connected, int remainingEmigrants,
        List<ServerPlayer> stanceDirty) {
        this.connected = connected;
        this.remainingEmigrants = remainingEmigrants;
        this.stanceDirty = stanceDirty;
    }
    .....Add getters and setters{....
}
```

3. Make a link from the old to the new class.

In this we make link from *ServerPlayer* class to *ServerConnection* class.

```
private ServerConnection data = new ServerConnection (false, 0,
    new ArrayList<ServerPlayer>());
```

4.In the *ServerPlayer* class we remove fields and replace with read and write accesses.

5. compile both *ServerPlayer* and *ServerConnection* class.

Class diagram :

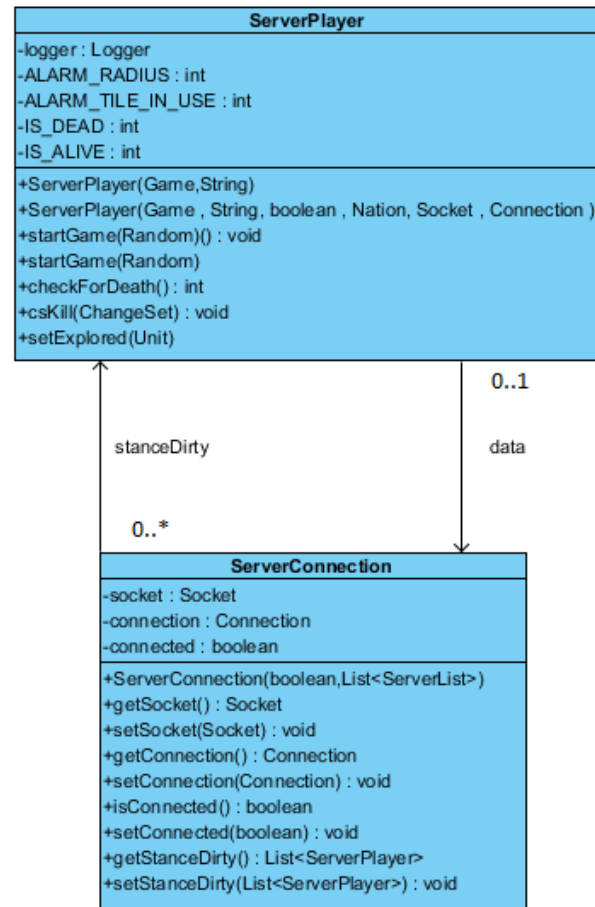


Fig: - Image shown above is the class diagram after refactoring Extract class.

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 ship and pass the parameters like attacker colony, random, cs to it i.e. ship(attacker colony, random, cs) .In this way, the damaged or illegally obtained ships will be dealt by this method.

Mechanics :

1. Create a new method, and name it after the intention of the method .

Here we create a new method **ship** with parameters: attacker, colony, random, cs .

```
private void ship(Unit attacker, Colony colony, Random random, ChangeSet cs)
{.....
}
```

2. Copy the extracted code from the source method into the new target method.

We have three fields of type Lists i.e. buildingList, shipList, goodsList which collect the damageable buildings, ships, movable goods information and variable of type int i.e. pillage which is used for storing information regarding stealing gold as an extra choice.

```
private void ship(Unit attacker, Colony colony, Random random, ChangeSet cs)
{
    if (pillage < buildingList.size() + shipList.size()) {
        Unit ship = shipList.get(pillage - buildingList.size());
        if (ship.getRepairLocation() == null) {
            csSinkShipAttack(attacker, ship, cs);
        } else {
            csDamageShipAttack(attacker, ship, cs);
        }
    }
}
```

3. Scan the extracted code for reference to any variable that are are local iin scope to the source method.

```
private void ship(Unit attacker, Colony colony, Random random, ChangeSet cs)
{
    List<Building> buildingList = colony.getBurnableBuildingList();
    List<Unit> shipList = colony.getShipList();
```

```

        List<Goods> goodsList = colony.getLootableGoodsList();
        int pillage = Utils.randomInt(logger, "Pillage choice", random,
        buildingList.size() + shipList.size() + goodsList.size()+
        ((colony.canBePlundered()) ? 1 : 0));
    if (pillage < buildingList.size() + shipList.size()) {
        Unit ship = shipList.get(pillage - buildingList.size());
        if (ship.getRepairLocation() == null) {
            csSinkShipAttack(attacker, ship, cs);
        } else {
            csDamageShipAttack(attacker, ship, cs);
        }
    }
}

```

4. Calling the new method in the old class :

```

Ship (attacker , colony , random , cs);

```

5. Compile and run refracted ServerPlayer.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 *settleAgreement*. 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 *settleAgreement* 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.

Mechanics :

1. Create a new method, and name it after the intention of the method .

```

private ModelMessage settleAgreement (Unit unit, IndianSettlement settlement,
int gold, int year)

```

2. Copy the extracted code from the source method into the new target method.

```

private ModelMessage settleAgreement (Unit unit, IndianSettlement settlement,
int gold, int year) {
    settlement.setLastTribute(year);
}

```

```

ModelMessage m;
if (gold > 0) {
m = new ModelMessage (ModelMessage.MessageType.FOREIGN_DIPLOMACY,
    "scoutSettlement.tributeAgree", unit, settlement)
    .addAmount("%amount%", gold);
} else {
m = new ModelMessage (ModelMessage.MessageType.FOREIGN_DIPLOMACY,
    "scoutSettlement.tributeDisagree", unit, settlement);
}
unit.setMovesLeft(0);
return m;
}

```

3. Scan the extracted code for reference to any variable that are local in scope to the source method.

4. Calling the new method in the old class :

```
ModelMessage settleAgreement = settleAgreement(unit, settlement, gold , year);
```

5. Compile and run refactored *InGameController.java* class.

Refactoring of ServerUnit class (Extract Superclass) to ServerUnitCurrentStatus :

The *ServerUnit* class in the *Server.Control.model* Package has a code smell of God class. The major functionalities of this class is determining the Server Unit status, location, equipment, unit type etc. It has more than 1600 lines of code and having responsibility of maintaining current status as well as the taking care of other unit. Every unit is owned by a player and it has a location. So the idea is to separate the responsibility such as one class maintain the current status of Server Unit and the remaining responsibilities are taken care by the previous class *ServerUnit*. The *ServerUnit* class has bunch of methods; we move those methods into superclass that belongs to maintaining the current status of server unit into *ServerUnitCurrentStatus*. We move methods like *csNewTurn*, *collectNewTiles*, *csMove* into the superclass called *ServerUnitCurrentStatus*.

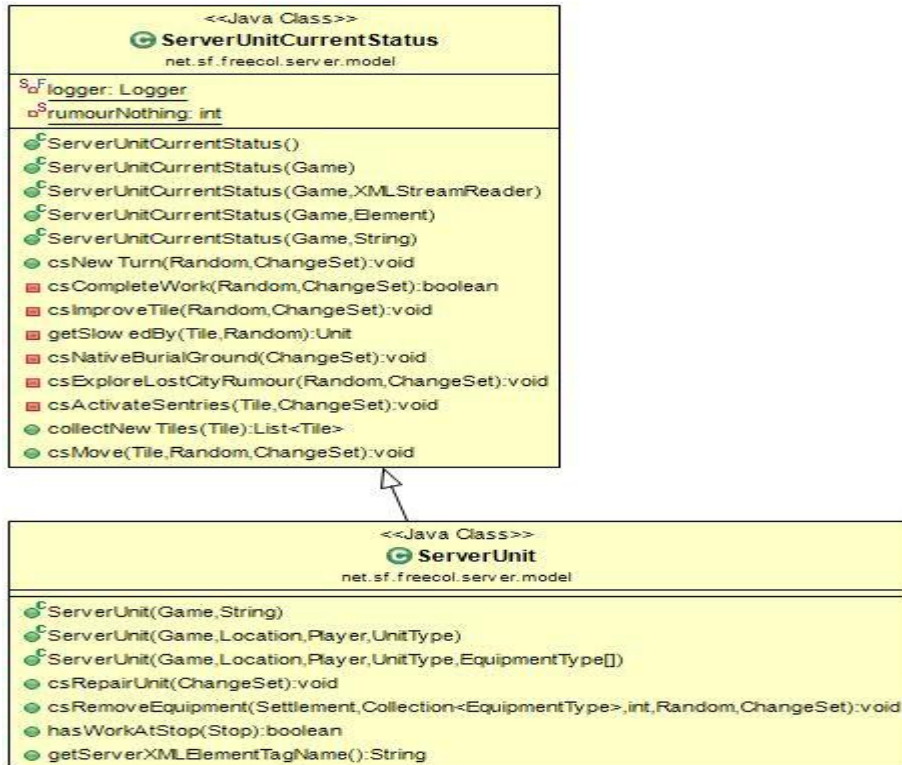


Fig: - Image shown above is the class diagram after refactoring Extract Super class.

Mechanics :

1. Create a blank abstract superclass; make the original classes subclasses of this superclass.

```
public class ServerUnitCurrentStatus extends Unit {
```

2. Next use Pull Up Method to move common elements to the superclass as shown below :

```
public void csNewTurn(Random random, ChangeSet cs) {

public List<Tile> collectNewTiles(Tile tile) {

public void csMove(Tile newTile, Random random, ChangeSet cs) {
```

3. Compile and test after each pull.
4. Then we examine the methods left on the subclasses to see if there are common parts.

Conclusion:

JUnit is a unit testing framework for the Java programming language. JUnit has been important in the development of test-driven development and is one of the family of unit testing frameworks which is collectively known as [xUnit](#) that originated with [SUnit](#).

JUnit is linked as a [JAR](#) at compile-time; the framework resides under packages junit.framework for JUnit 3.8 and earlier and under org.junit for JUnit 4 and later.

Reference: <http://en.wikipedia.org/wiki/JUnit>

We have tested the Freecol game application after making changes and fixing bugs. Initially, we encountered many problems due to the humungous size of the project but later on, we were able to cope up with all of them. We were able to do this with the help of JUnit framework and 'ANT' builder to compile build.xml and we were successful in our attempts.

Also, it was indeed a great experience to learn about Github, refactorings and design patterns in this course. Thanks to Professor Peter Rigby for his dedicated and distinct approach to teach students.