Mergeddoc Deliverable

Group ID: 13

André Branco, 62482, Aser28860d
Beatriz Machado, 62551, BeatrizCaiola
João Silveira, 62654, Joao-Pedro-Silveira
Rodrigo Fernandes 63191, LemosFTW
Rodrigo Suzana, 63069, suzana2314
Tiago Sousa, 63324, tiago-cos

https://www.youtube.com/watch?v=jbbP0DNPo7E

https://github.com/LemosFTW/SE2324_63191_63324-62654_63069_62551_62482

Índice

Índice **1**
User Stories **4**
Code Metrics **5**
Beatriz Machado 62551 (BeatrizCaiola) - Chidamber and Kemerer Metrics 5
Coupling Between Objects (CBO) 5
Depth of Inheritance Tree (DIT) 5
Lack of Cohesion of Methods (LCOM) 5
Number of Children (NOC) 5
Response for Class (RFC) 6
Weighted Methods Per Class (WMC) 6
Relation to Code Smells 6
Rodrigo Monteiro Suzana 63069 (suzana2314) - Martin Packaging Metrics 7
Efferent Coupling (Ce) 7
Afferent Coupling (Ca) 7
Instability 7
Abstractness 8
Distance from the main sequence 8
Rodrigo Fernandes 63191 (LemosFTW) - Javadoc Coverage 10
Potenciais Pontos de Problema: 10
João Pedro Silveira 62654 (Joao-Pedro-Silveira) - Complexity Metrics 11
Metrics Definitions 11
Project Metrics: 11
Module Metrics: 11
Package Metrics: 11
Class Metrics: 11
Method Metrics: 11
Potencial trouble spots 11
Relation with code smells 12
Tiago Sousa 63324 (tiago-cos) - Lines of Code Metrics 13
Potential trouble spots 13
Relation to code smells 14
André Branco 62482 (Aser28860d) - Dependency Metrics 15
Number of Cyclic Dependencies 15
Number of Dependencies 15
Number of Transitive Dependencies 15
Number of Dependents 15
Number of Transitive Dependents 15
Number of Package Dependencies 15
Number of Dependent Packages: 16
Potencial Trouble Spots 16
Code Smells **17**
Beatriz Machado 62551 (BeatrizCaiola) 17

Duplicated Code 17
Message Chains 19
Long Method 22
Rodrigo Monteiro Suzana 63069 (suzana2314) 33
Speculative Generality 33
Long methods 34
Data Class 36
Rodrigo Fernandes 63191 (LemosFTW) 37
Speculative Generality 37
Long Method 37
Data Class 45
João Pedro Silveira 62654 (Joao-Pedro-Silveira) 46
Data Clumps 46
Message chains 54
Divergent Class 54
Tiago Sousa 63324 (tiago-cos) 55
Long Method 55
Data Class 62
Switch Statement 63
André Branco 62482 (Aser28860d) 65
Long Method 65
Speculative Generality 68
Duplicate Code 69
Design Patterns **71**
Beatriz Machado 62551 (BeatrizCaiola) 71
Template Pattern 71
Memento Pattern 73
Composite Pattern 75
Rodrigo Monteiro Suzana 63069 (suzana2314) 77
Template Pattern 77
Factory pattern 77
Observer Pattern 78
Rodrigo Fernandes 63191 (LemosFTW) 81
Proxy Pattern 81
Singleton Pattern 83
Template Method Pattern 84
João Pedro Silveira 62654 (Joao-Pedro-Silveira) 86
Composite Pattern 86
Strategy Design Pattern 90
Template Pattern 91
Tiago Sousa 63324 (tiago-cos) 95
Singleton Pattern 95
Template Pattern 95
State Pattern 97

André Branco 62482 (Aser28860d) 100
Singleton Pattern 100
Proxy Pattern 101
Template Pattern 102
Use Case **105**
Beatriz Machado 62551 (BeatrizCaiola) - Trade and Economy 105
Rodrigo Monteiro Suzana 63069 (suzana2314) - Colony Management 107
Rodrigo Fernandes 63191 (LemosFTW) - Government and Independence 108
João Pedro Silveira 62654 (Joao-Pedro-Silveira) - Combat 110
Tiago Sousa 63324 (tiago-cos) 112
André Branco 62482 (Aser28860d) 114
Implementation Use Cases **116**
André Branco 62482 (Aser28860d) e João Pedro Silveira 62654 (Joao-Pedro-Silveira) - Cave Exploration 116
Beatriz Machado 62551 (BeatrizCaiola) e Tiago Sousa 63324 (tiago-cos) - Gunpowder and Ammunition 119
Rodrigo Monteiro Suzana 63069 (suzana2314) e Rodrigo Fernandes 63191 (LemosFTW) - Capture Ship 124
Postmortem **126**
Rodrigo Fernandes 63191 (LemosFTW) 126
Use Case Diagram 126
MOOD Code Metrics 127
Tiago Sousa 63324 (tiago-cos) 128
Use Case Diagram 128
Beatriz Machado 62551 (BeatrizCaiola) 130
Use Case: Trade and Economy 130

User Stories

As an experienced player, I want to be able to invade and claim enemy ships, so that I can grow my colony even faster.

As an experienced player, I want to see caves as a new tile square that can have either a treasure or a trap, so that I can have more risk/reward situations.

As a history enthusiast, I want there to be gunpowder in the game, used in firearms as a way to make the combat feel more authentic.

Code Metrics

Beatriz Machado 62551 (BeatrizCaiola) - Chidamber and Kemerer Metrics

The metric set Chidamber & Kemerer is an object-oriented metrics suite that presents 6 metrics. The metrics exhibited by this metric suit are:

Coupling Between Objects (CBO)

This metric indicates the number of classes to which a class is coupled. The instances of coupling considered for this metric are method calls, field accesses, inheritance, arguments, return types and exceptions. This metric can help identify eventual cases of the Code Smell Inappropriate Intimacy as classes with high coupling are more likely to be dependent on others and of establishing a cycle of dependency.

This metric locates eventual trouble spots, such as the class FreeColObject with whom there might be many strong dependencies.

Depth of Inheritance Tree (DIT)

This metric presents the maximum inheritance path from a given class to its root. Therefore, this metric can aid the identification of the Code Smell of Refused Request, as classes with large inheritance paths might be, for instance, inheriting unnecessary methods. A large inheritance path may also indicate a problem of Speculative Generality, as a too-general parent class would lead to the need of more children classes with implementations that actually tackle the problem at hand.

Classes with high inheritance paths that could, therefore, be identified through this metric as potential trouble spots are, for example, the classes ReportCargoPanel, ReportMilitaryPanel and ReportNavalPanel.

Lack of Cohesion of Methods (LCOM)

This metric counts the sets of methods in a class that aren't related through the sharing of some of the class' fields. This metric has since suffered many alterations after being met with much critique, so it wasn't employed in the identification of potential trouble spots.

Number of Children (NOC)

This metric presents the number of immediate descendants of a given class. A high number of children indicates a heavy reuse of the class in question which, on the one hand, might be beneficial as it allows for code reuse, but it also may indicate problems in the abstraction of the parent class. The latter might result in the Code Smell of Speculative Generality, as the parent class could have been made to tackle future problems, instead of the immediate ones, which then result in the need for more children classes.

Classes with very high NOCs may be identified as potential trouble spots, such as the class FreeColAction, which might be overly general in order to then warrant such a necessary number of children.

Response for Class (RFC)

This metric presents the number of distinct methods that can be executed when an object of that class receives a message, that is, when a method is invoked for that object. This metric can assist in the location of the Code Smells of Feature Envy and Inappropriate Intimacy. In the case of Feature Envy various instances of communication between methods of different classes might indicate that this code should be together. A case of Inappropriate Intimacy might arise from a dependency established through the communication between the classes' methods.

Classes such as InGameController from the Client package, which holds a very high RFC (700) could be considered potential trouble spots for the above mentioned Code Smells.

Weighted Methods Per Class (WMC)

This metric counts the number of methods in a given class. The high number of methods suggest that some classes might present the Code Smell of Divergent Class, as they are taking on many responsibilities that then warrant the presence of so many methods. This metric can also indicate the Code Smell of Large Classes, as classes with many methods tend to be longer.

Through analysing this metric we can identify potential trouble spots, such as the class InGameController which holds a very high number of methods (786), therefore signifying an eventual divergence in the present methods as the class holds too much responsibility.

Relation to Code Smells

Duplicated Code

By analysing the metric WMC we were able to locate the Code Smell Duplicated Code in class Colony, as the number it displayed for this metric was quite high (396). The elevated number of methods indicated the eventual presence of unnecessary code, such as the duplicated methods we identified.

Message Chains

Through examining the metric RFC we were able to detect the Code Smell Message Chains in class ServerColony, as the number it exhibited for this metric was quite high (209). The elevated number of methods being executed as a method is invoked for that object indicated the eventual occurrence of chaining.

Long Method

By inspecting the metric WMC we were able to identify the Code Smell Long Method in class ServerPlayer, as the number shown for this metric was quite high (601). As the class with the 3rd highest number for this metric, there was a high likelihood of locating Code Smells, making this class trouble spot.

Rodrigo Monteiro Suzana 63069 (suzana2314) - Martin Packaging Metrics

By calculating this metric, using the plugin in Idea IntelliJ, we obtain an csv file with the following columns:

- Abstractness;
- Afferent Couplings;
- Efferent Couplings;
- Distance from the main sequence;
- Instability;

Efferent Coupling (Ce)

This metric measures the relationship between classes. I.e., the number of classes in a package that depend on classes located in other packages.

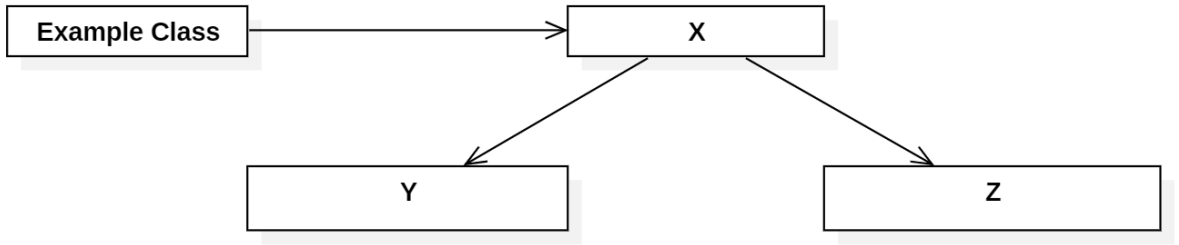
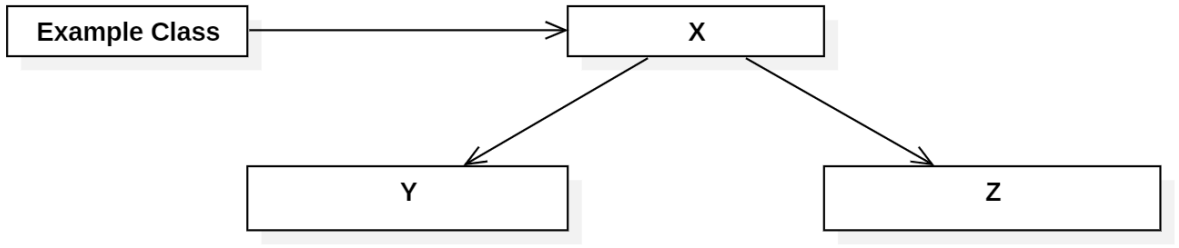


Figure 1

The metric Ce of this example would be 2, because class X has 2 dependencies to other 2 classes.

Afferent Coupling (Ca)



This metric measures incoming dependencies between classes. It enables the measurement of the sensitivity in the remaining packages to changes in the analyzed package.

The metric Ca of this example would be 1, because class X has 1 incoming dependency (from *Example Class*).

Instability

This metric measures the susceptibility of class to changes. This metric is defined according to the formula:

$$I = \frac{Ce}{Ce + Ca}$$

In our example shown in Figure 1, the instability would be around 0.67. This metric helps identify two types of components:

1. Packages with lots of connections going out and few coming in (value closer to 1) are less stable because they can be easily changed.
2. Packages with lots of connections coming in and few going out (value closer to 0) are harder to change because they have more responsibilities.

Abstractness

This metric measures how abstract a package is, somewhat similar to instability. Abstractness refers to the ratio of abstract classes to all classes within the package, and can be easily calculated using the following formula:

$$A = \frac{N_{Abstract}}{N_{Abstract} + N_{Concrete}}$$

Where $N_{Abstract}$ is the number of abstract classes in a package and $N_{Concrete}$ is the number of concrete classes in a package.

Distance from the main sequence

This metric measures the balance between stability and abstractness, using the following formula:

$$D = |A + I - 1|$$

The goal is to keep this metric as low as possible to keep components close to the main sequence. There are two extreme, unfavorable cases:

1. When A = 0 and I = 0, a package is extremely stable and concrete. This is undesirable because the package is rigid and cannot be extended.
2. When A = 1 and I = 1, it's a nearly impossible situation. A completely abstract package must have some connection to the outside so that an instance implementing the functionality from the abstract classes in this package could be created.

The data extracted from the code base using the plugin makes it easy for us to spot certain code issues. For instance, when we analyze the instability column and notice values closer to 1, it suggests a high number of "connections to the outside," which indicates strong

interdependencies among classes. This may imply that classes within a package are deviating from their core purpose.

Conversely, when we encounter an instability value closer to 0, indicating a package with low visibility (fewer connections to the "outside"), it might signify that some classes within the package serve as simple data classes or have little relevance to the codebase.

However, it's essential to exercise caution when relying solely on instability values. For instance, there's a utils package with an instability value of 0. This doesn't necessarily imply issues with the code; it simply means that the classes within the package lack external connections.

Rodrigo Fernandes 63191 (LemosFTW) - Javadoc Coverage

Foram analisados 4 valores contidos nesta métrica:

- **JF** - Javadoc Field Coverage
- **JLOC** - Javadoc Lines of Code
- **JM** - Javadoc Method Coverage
- **JC** - Javadoc Class Coverage

Nas seguintes instancias: *Method Metrics, Class Metrics, Interface Metrics, Package metrics, Module Metrics e Project Metrics.*

O que representam para o código:

1. Na área do Method Metrics o campo JLOC, indica quais métodos foram comentados de acordo com o Javadoc.
2. Na área do Class Metrics temos os seguintes campos: JF, JLOC, JM. JF nos diz a percentagem de campos comentados no código, já o JLOC, nos diz quantas linhas de comentários foram escritas e por fim o JM nos diz quantos métodos foram comentados.
3. A área do Interface Metrics e similar a área do Class Metrics portanto JF, JLOC, JM significam a mesma coisa só que aplicada a interfaces.
4. Na área do Package Metrics temos os seguintes campos: JF, JLOC, JM, JC. Os valores de JF, JLOC, JM significam a mesma coisa só que aplicada as instancias anteriores, porém o JC nos diz a percentagem de classes que tem ao menos um comentário.
5. Na área do Module Metrics temos os seguintes campos: JF, JLOC, JM, JC. Os valores de JF, JLOC, JM, JC significam a mesma coisa só que aplicada as instancias anteriores. Porém temos um campo que faz a média geral desses valores, possibilitando ter uma visão geral dos módulos.
6. Na área do Project Metrics temos os seguintes campos: JF, JLOC, JM, JC. Os valores de JF, JLOC, JM, JC significam a mesma coisa só que aplicada as instancias anteriores.

Potenciais Pontos de Problema:

Nessas métricas conseguimos observar alguns problemas referente a métodos e classes. Algumas das classes e métodos não tem comentários, assim dificultando o entendimento de terceiros sobre o código. Os code smells achados estão contidos nessas métricas:

- DataClass – BuildPlan : JF = 0%, JLOC = 4, JM = 33,3%
- Long Method – UpdateColony: JLOC = 5
- Speculative Generality – getWorkerWishes: JLOC = 7

João Pedro Silveira 62654 (Joao-Pedro-Silveira) - Complexity Metrics

Metrics Definitions

The collected metrics were essentially the complexity of the code, as well as how hard it is to perceive.
In order to represent these metrics we use the Cyclomatic complexity, which is a measurement of the number of linearly-independent paths through a program module.

Project Metrics:

- Average cyclomatic complexity
- Total cyclomatic complexity

Module Metrics:

- Average cyclomatic complexity
- Total cyclomatic complexity

Package Metrics:

- Average cyclomatic complexity
- Total cyclomatic complexity

Class Metrics:

- Average operation Complexity
- Maximum operation Complexity (maximum Cyclomatic complexity of any non abstract method in the class)
- Weighted method complexity (total cyclomatic complexity of the methods in the class)

Method Metrics:

- Cognitive complexity (similar to Cyclomatic complexity but more focused on understandability)
- Cyclomatic complexity
- Design Complexity (how interlinked the method is with calls to other methods)
- Essential Cyclomatic complexity (graph-theoretic measure of just how ill-structured a method's control flow is)

Potencial trouble spots

The parts of the code that seem to have the most troublesome spots are parts like:

- InGameControler, which receives its complexity from the large amount of switches, if’s and cycles to deal with all the possibilities the player might make

- Unit and Player classes suffer from a large complexity due to a fact that dese classes try to do many things in the same class, resulting in gigantic classes of more than 4000 lines of code
- Classes that deal with AI.
- The class Map also has the method with the second highest complexity (searchMap), this is due to the fact the method deals with algorithmic search.

Relation with code smells

A high complexity usually indicates the code Smells for large methods and large Classes, which become hard to read due to their size. A high complexity also might indicate divergent Classes, like what happens with the class Map.

Tiago Sousa 63324 (tiago-cos) - Lines of Code Metrics

The collected metrics were the number of lines of code in our project, along with information derived from those lines.
There were 7 different types of metrics collected:

- Interface metrics:
    - Lines of code (LOC), which include comments, documentation and code.
    - Comment lines of code (CLOC), which only include comments.
    - Javadoc lines of code (JLOC), which only include javadoc lines.
    - Non-comment lines of code (NCLOC), which only include non-commented lines of code.
- Package metrics:
    - CLOC
    - CLOC(rec)
    - JLOC
    - JLOC(rec)
    - LOC
    - LOC(rec)
    - LOCt
    - LOCt(rec)
    - NCLOC
    - NCLOCp
    - NCLOCp(rec)
      NCLOCt
    - NCLOCt(rec)

In the above metrics, (rec) indicates that the counting was done recursively, “t” indicates that the lines of code counted were test code and “p” indicates that the lines counted were product code.

- Module metrics, which calculate the same data as above, for each module, with the added distinction of counting the lines of code of each programming language separately.
- File type metrics, which calculate the LOC and NCLOC for each file type in the project.
- Project metrics, which calculate the same data as the module metrics, but for the whole project instead.
- Class metrics, which calculate the CLOC, JLOC and LOC for each class.
- Method metrics, which calculate the CLOC, JLOC, LOC. NCLOC and RLOC metrics for each method. The RLOC metric is the relation between the lines of code of the method and the total lines of code in the class where the method is located.

Potential trouble spots

In the collected metrics, we can observe some potential trouble spots, namely in the method and class metrics. For example, by analyzing the lines of code in our classes, we can see that the InGameController holds the most lines of code, possibly indicating some issues like, for example, code that isn’t distributed correctly and that should be in another class. Another

way we can find potential trouble spots is by analyzing methods. If we check the methods that have the highest amount of comment and documentation lines, we'll usually arrive at methods that are too complex and confusing, requiring the use of many comments in order to understand what is happening. This can be observed in the assignWorkers method in the ColonyPlan class and in the travelToTarget method in the Mission class. Furthermore, we can also take a look at the relative lines of code in order to find potential trouble methods and classes. For example, the method generateAttackResult occupies the entirety of the HitpointsCombatModel class, and has a considerable size. This could indicate that this class is not entirely necessary, or that the method can be subdivided into smaller, less complex methods in order to increase readability.

Relation to code smells

The code smells identified were located with the help of these metrics. The CombatResult class is a data class that was found using the lines of code metrics, by searching for classes with very little lines of code. The csCombat was found by searching through the methods with the most lines of code, and the assignWorkers method was found by searching through the methods with the most comment lines.

André Branco 62482 (Aser28860d) - Dependency Metrics

Dependency metrics are used to measure and analyze the relationships between classes and packages in a software system (interfaces are also measured). These metrics help software developers and architects to understand the complexity and maintainability of their codebase.

The following values are shown at the class level:

Number of Cyclic Dependencies

- This metric indicates the number of cyclic dependencies within a class. Cyclic dependencies occur when two or more classes depend on each other in a circular way, creating a cycle.

Number of Dependencies

- This metric measures the total number of classes or components that depend on a specific class. It reflects how interconnected a class is with other parts of the codebase.

Number of Transitive Dependencies

- Transitive dependencies are dependencies that are inherited through other classes. This metric counts the total number of classes, both direct and indirect, that depend on a specific class.

Number of Dependents

- This metric represents the total number of classes that a specific class depends on. It shows how many other classes are needed for the proper functioning of the analyzed class.

Number of Transitive Dependents

- Similar to the number of transitive dependencies, this metric counts the total number of classes, both direct and indirect, that a specific class depends on.

Number of Package Dependencies

- This metric measures the number of other packages or modules that a class depends on.

Number of Dependent Packages:

- This metric represents the total number of packages or modules that depend on a specific class. It indicates how many different packages rely on the functionality provided by the analyzed class.

At the Interface level, we have the same 7 metrics as the class level, maintaining the same definition but related to interfaces and their relationships.

At the package level, we have 4 package related metrics, (“Number of Cyclic Package Dependencies”, “Number of Package Dependencies”, “Number of Dependent Packages”, “Number of Transitively Dependent Packages”), respectively equivalent to the metrics “Number of Cyclic Dependencies”, “Number of Dependencies”, “Number of Dependents”, “Number of Transitive Dependents”, in class and interface level’s, but related to packages.

Potencial Trouble Spots

In the FreeCol code these metrics in some cases show high values in classes, interfaces and packages. Those suggest that the code might be complex, hard to maintain and in some cases over-used, being that it should be reviewed and adjusted according to the class’s responsibilities, method complexity, efficient and appropriate use of dependencies etc.

An example is the class TileType (scr.net.sf.freecol.common.model.TileType), with 922 transitive dependents, which means that it is a potentially complex and heavily used class.
Another example is the BaseProduction interface (scr.net.sf.freecol.common.model.BaseProduction), with 804 cyclic dependencies, it means that many classes depend on this interface as well as the interface depends on those classes, which leads to maintainability issues and makes the codebase difficult to work with.
A solution for this case might be refactoring the interface to reduce its dependencies, and if possible, even split the interface into smaller interfaces, each serving a specific purpose.

Code Smells

Beatriz Machado 62551 (BeatrizCaiola)

Duplicated Code

net.sf.freecol.common.model.Colony.getsTurnsToComplete(BuildableType buildable)

```
/**
* Returns how many turns it would take to build the given
* {@code BuildableType}.
*
* @param buildable The {@code BuildableType} to build.
* @return The number of turns to build the buildable, negative if
*     some goods are not being built, UNDEFINED if none is.
*/
public int getTurnsToComplete(BuildableType buildable) {
   return getTurnsToComplete(buildable, null);
}

/**
* Returns how many turns it would take to build the given
* {@code BuildableType}.
*
* @param buildable The {@code BuildableType} to build.
* @param needed The {@code AbstractGoods} needed to continue
*     the build.
* @return The number of turns to build the buildable (which may
*     be zero, UNDEFINED if no useful work is being done, negative
*     if some requirement is or will block completion (value is
*     the negation of (turns-to-blockage + 1), and if the needed
*     argument is supplied it is set to the goods deficit).
*/
public int getTurnsToComplete(BuildableType buildable,
                              AbstractGoods needed) {
   final List<AbstractGoods> required = buildable.getRequiredGoodsList();
   int turns = 0, satisfied = 0, failing = 0, underway = 0;

   ProductionInfo info = productionCache.getProductionInfo(buildQueue);
   for (AbstractGoods ag : required) {
       final GoodsType type = ag.getType();
       final int amountNeeded = ag.getAmount();
       final int amountAvailable = getGoodsCount(type);
       if (amountAvailable >= amountNeeded) {
           satisfied++;
           continue;
       }
       int production = productionCache.getNetProductionOf(type);
       if (info != null) {
           AbstractGoods consumption = find(info.getConsumption(),
```

```
                    AbstractGoods.matches(type));
            if (consumption != null) {
                // add the amount the build queue itself will consume
                production += consumption.getAmount();
            }
        }
        if (production <= 0) {
            failing++;
            if (needed != null) {
                needed.setType(type);
                needed.setAmount(amountNeeded - amountAvailable);
            }
            continue;
        }

        underway++;
        int amountRemaining = amountNeeded - amountAvailable;
        int eta = amountRemaining / production;
        if (amountRemaining % production != 0) eta++;
        turns = Math.max(turns, eta);
    }

    return (satisfied + underway == required.size()) ? turns // Will finish
            : (failing == required.size()) ? UNDEFINED // Not even trying
            : -(turns + 1); // Blocked by something
}
```

The above code from the class Colony qualifies as the Code Smell Duplicated Code as they consist of two methods that perform essentially the same functionality. The main difference between the methods resides in the amount of parameters, as the first instance of getTurnsToComplete() only has one parameter, whilst the second occurrence has 2. A way of fixing this code smell would be to converge these methods, resulting in a single method, as the first instance could be written as a circumstance covered by the second in which the object AbstractGoods equals null.

Message Chains

net.sf.freecol.server.model.ServerColony.csNewTurnWarnings(Random random, LogBuilder lb, ChangeSet cs)

```
/**
* Do the checks for user warnings that must wait for all player
* settlements, units and whatever to stabilize.  Along the way,
* throw away excess goods.
*
* For example, a pioneer might clear a colony tile and change the
* lumber amount.
*
* @param random A {@code Random} number source.
* @param lb A {@code LogBuilder} to log to.
* @param cs A {@code ChangeSet} to update.
*/
public void csNewTurnWarnings(Random random, LogBuilder lb, ChangeSet cs) {
   final Specification spec = getSpecification();
   final BuildQueue<?>[] queues = new BuildQueue<?>[] {
       this.buildQueue, this.populationQueue };
   final GoodsContainer container = getGoodsContainer();

   for (WorkLocation wl : getCurrentWorkLocationsList()) {
       if (wl instanceof ServerBuilding) {
           // FIXME: generalize to other WorkLocations?
           ((ServerBuilding)wl).csCheckMissingInput(getProductionInfo(wl),
                                                    cs);
       }
   }

   for (BuildQueue<?> queue : queues) {
       ProductionInfo info = getProductionInfo(queue);
       if (info == null) continue;
       if (info.getConsumption().isEmpty()) {
           BuildableType build = queue.getCurrentlyBuilding();
           if (build != null) {
               AbstractGoods needed = new AbstractGoods();
               int complete = getTurnsToComplete(build, needed);
               // Warn if about to fail, or if no useful progress
               // towards completion is possible.
               if (complete == -2 || complete == -1) {
                   cs.addMessage(owner,
                       new ModelMessage(MessageType.MISSING_GOODS,
                                        "model.colony.buildableNeedsGoods",
                                        this, build)
                           .addName("%colony%", getName())
                           .addNamed("%buildable%", build)
                           .addAmount("%amount%", needed.getAmount())
                           .addNamed("%goodsType%", needed.getType()));
               }
           }
```

```
    }
}

// Throw away goods there is no room for, and warn about
// levels that will be exceeded next turn
final int limit = getWarehouseCapacity();
final int adjustment = limit / GoodsContainer.CARGO_SIZE;
for (Goods goods : transform(getCompactGoodsList(),
                             AbstractGoods::isStorable)) {
    final GoodsType type = goods.getType();
    final ExportData exportData = getExportData(type);
    final int low = exportData.getLowLevel() * adjustment;
    final int high = exportData.getHighLevel() * adjustment;
    final int amount = goods.getAmount();
    final int oldAmount = container.getOldGoodsCount(type);

    if (amount < low && oldAmount >= low
        && type != spec.getPrimaryFoodType()) {
        cs.addMessage(owner,
            new ModelMessage(MessageType.WAREHOUSE_CAPACITY,
                             "model.colony.warehouseEmpty",
                             this, type)
                .addNamed("%goods%", type)
                .addAmount("%level%", low)
                .addName("%colony%", getName()));
        continue;
    }
    if (type.limitIgnored()) continue;
    String messageId = null;
    int waste = 0;
    if (amount > limit) {
        // limit has been exceeded
        waste = amount - limit;
        container.removeGoods(type, waste);
        messageId = "model.colony.warehouseWaste";
    } else if (amount == limit && oldAmount < limit) {
        // limit has been reached during this turn
        messageId = "model.colony.warehouseOverfull";
    } else if (amount > high && oldAmount <= high) {
        // high-water-mark has been reached this turn
        messageId = "model.colony.warehouseFull";
    }
    if (messageId != null) {
        cs.addMessage(owner,
            new ModelMessage(MessageType.WAREHOUSE_CAPACITY,
                             messageId, this, type)
                .addNamed("%goods%", type)
                .addAmount("%waste%", waste)
                .addAmount("%level%", high)
                .addName("%colony%", getName()));
    }
```

```
            // No problem this turn, but what about the next?
            if (!(exportData.getExported()
                  && hasAbility(Ability.EXPORT)
                  && owner.canTrade(type, Market.Access.CUSTOM_HOUSE))
                && amount <= limit) {
                int loss = amount + getNetProductionOf(type) - limit;
                if (loss > 0) {
                    cs.addMessage(owner,
                        new ModelMessage(MessageType.WAREHOUSE_CAPACITY,
                                         "model.colony.warehouseSoonFull",
                                         this, type)
                            .addNamed("%goods%", goods)
                            .addName("%colony%", getName())
                            .addAmount("%amount%", loss));
                }
            }
        }

        // If a build queue is empty, check that we are not producing
        // any goods types useful for BuildableTypes, except if that
        // type is the input to some other form of production.  (Note:
        // isBuildingMaterial is also true for goods used to produce
        // role-equipment, hence neededForBuildableType).  Such
        // production probably means we forgot to reset the build
        // queue.  Thus, if hammers are being produced it is worth
        // warning about, but not if producing tools.
        if (any(queues, BuildQueue::isEmpty)
            && any(spec.getGoodsTypeList(), g ->
                (g.isBuildingMaterial()
                    && !g.isRawMaterial()
                    && !g.isBreedable()
                    && getAdjustedNetProductionOf(g) > 0
                    && neededForBuildableType(g)))) {
            cs.addMessage(owner,
                new ModelMessage(MessageType.BUILDING_COMPLETED,
                    "model.colony.notBuildingAnything", this)
                .addName("%colony%", getName()));
        }
    }
```

The above code has various instances of the Code Smell Message Chains as there are many subsequent calls in order to get another object, for example:

```
ModelMessage(MessageType.WAREHOUSE_CAPACITY,
                                 messageId, this, type)
                .addNamed("%goods%", type)
                .addAmount("%waste%", waste)
                .addAmount("%level%", high)
                .addName("%colony%", getName()))
```

A way of fixing this Code Smell is to allow direct access to the required object instead of chaining it through various method calls.

Long Method

net.sf.freecol.server.model.ServerPlayer.csCombat(FreeColGameObject attacker, FreeColGameObject defender, List<CombatEffectType> crs, Random random, ChangeSet cs)

```
/**
* Combat.
*
* @param attacker The {@code FreeColGameObject} that is attacking.
* @param defender The {@code FreeColGameObject} that is defending.
* @param crs A list of {@code CombatResult}s defining the result.
* @param random A pseudo-random number source.
* @param cs A {@code ChangeSet} to update.
*/
public void csCombat(FreeColGameObject attacker,
                     FreeColGameObject defender,
                     List<CombatEffectType> crs,
                     Random random,
                     ChangeSet cs) {
   CombatModel combatModel = getGame().getCombatModel();
   boolean isAttack = combatModel.combatIsAttack(attacker, defender);
   boolean isBombard = combatModel.combatIsBombard(attacker, defender);
   Unit attackerUnit = null;
   Settlement attackerSettlement = null;
   Tile attackerTile = null;
   Unit defenderUnit = null;
   Player defenderPlayer = null;
   Tile defenderTile = null;
   if (isAttack) {
       attackerUnit = (Unit)attacker;
       //attackerPlayer = attackerUnit.getOwner();
       attackerTile = attackerUnit.getTile();
       defenderUnit = (Unit)defender;
       defenderPlayer = defenderUnit.getOwner();
       defenderTile = defenderUnit.getTile();
       boolean bombard = attackerUnit.hasAbility(Ability.BOMBARD);
       cs.addAttribute(See.only(this), "sound",
           (attackerUnit.isNaval()) ? "sound.attack.naval"
           : (bombard) ? "sound.attack.artillery"
           : (attackerUnit.isMounted()) ? "sound.attack.mounted"
           : "sound.attack.foot");
       if (attackerUnit.getOwner().isIndian()
           && defenderPlayer.isEuropean()
           && defenderUnit.getLocation().getColony() != null
           && !defenderPlayer.atWarWith(attackerUnit.getOwner())) {
           StringTemplate attackerNation
               = attackerUnit.getApparentOwnerName();
```

```
            Colony colony = defenderUnit.getLocation().getColony();
            cs.addMessage(defenderPlayer,
                new ModelMessage(ModelMessage.MessageType.COMBAT_RESULT,
                                 "combat.raid.ours", colony)
                    .addName("%colony%", colony.getName())
                    .addStringTemplate("%nation%", attackerNation));
        }
    } else if (isBombard) {
        attackerSettlement = (Settlement)attacker;
        attackerTile = attackerSettlement.getTile();
        defenderUnit = (Unit)defender;
        defenderPlayer = defenderUnit.getOwner();
        defenderTile = defenderUnit.getTile();
        cs.addAttribute(See.only(this), "sound", "sound.attack.bombard");
    } else {
        throw new RuntimeException("Bogus combat: " + attacker
            + " v " + defender);
    }
    assert defenderTile != null;

    // If the combat results were not specified (usually the case),
    // query the combat model.
    CombatResult combatResult = null;
    if (crs == null) {
            combatResult  =  combatModel.generateAttackResult(random,  attacker,
defender);
        crs = combatResult.getEffects();
    }
    if (crs.isEmpty()) {
        throw new RuntimeException("empty attack result: " + this);
    }

    // Extract main result, insisting it is one of the fundamental cases,
    // and add the animation.
    // Set vis so that loser always sees things.
    // FIXME: Bombard animations
    See vis; // Visibility that insists on the loser seeing the result.
    CombatEffectType result = crs.remove(0);
    switch (result) {
    case NO_RESULT:
        vis = See.perhaps();
        break; // Do not animate if there is no result.
    case WIN:
        vis = See.perhaps().always(defenderPlayer);
        if (isAttack) {
            if (attackerTile == null
                || attackerTile == defenderTile
                || !attackerTile.isAdjacent(defenderTile)) {
                logger.warning("Bogus attack from " + attackerTile
                    + " to " + defenderTile);
            } else {
```

```
                cs.addAttack(vis, attackerUnit, defenderUnit, true);
            }
        }
        break;
    case LOSE:
        vis = See.perhaps().always(this);
        if (isAttack) {
            if (attackerTile == null
                || attackerTile == defenderTile
                || !attackerTile.isAdjacent(defenderTile)) {
                logger.warning("Bogus attack from " + attackerTile
                    + " to " + defenderTile);
            } else {
                cs.addAttack(vis, attackerUnit, defenderUnit, false);
            }
        }
        break;
    default:
        throw new IllegalStateException("generateAttackResult returned: "
                                        + result);
    }

    // Now process the details.
    boolean attackerTileDirty = false;
    boolean defenderTileDirty = false;
    boolean moveAttacker = false;
    boolean burnedNativeCapital = false;
    Settlement settlement = defenderTile.getSettlement();
    Colony colony = defenderTile.getColony();
    IndianSettlement natives = (settlement instanceof IndianSettlement)
        ? (IndianSettlement) settlement
        : null;
    int attackerTension = 0;
    int defenderTension = 0;

    if (combatResult != null && combatResult.isAttackerHitpointsAffected()) {
        attackerUnit.setHitPoints(combatResult.getAttackerHitpointsAfter());
        attackerTileDirty = true;
    }

    if (combatResult != null && combatResult.isDefenderHitpointsAffected()) {
        defenderUnit.setHitPoints(combatResult.getDefenderHitpointsAfter());
        defenderTileDirty = true;
    }

    for (CombatEffectType cr : crs) {
        boolean ok;
        switch (cr) {
        case AUTOEQUIP_UNIT:
            ok = isAttack && settlement != null;
            if (ok) {
```

```
                csAutoequipUnit(defenderUnit, settlement, cs);
            }
            break;
        case BURN_MISSIONS:
            ok = isAttack && result == CombatEffectType.WIN
                && natives != null
                && isEuropean() && defenderPlayer.isIndian();
            if (ok) {
                defenderTileDirty |= natives.hasMissionary(this);
                csBurnMissions(attackerUnit, natives, cs);
            }
            break;
        case CAPTURE_AUTOEQUIP:
            ok = isAttack && result == CombatEffectType.WIN
                && settlement != null;
            if (ok) {
                csCaptureAutoEquip(attackerUnit, defenderUnit, cs);
                attackerTileDirty = defenderTileDirty = true;
            }
            break;
        case CAPTURE_COLONY:
            ok = isAttack && result == CombatEffectType.WIN
                && colony != null
                  && (isEuropean() || isUndead()) && (defenderPlayer.isEuropean()
|| defenderPlayer.isUndead());
            if (ok) {
                csCaptureColony(attackerUnit, (ServerColony)colony,
                                random, cs);
                attackerTileDirty = defenderTileDirty = false;
                moveAttacker = true;
                defenderTension += Tension.TENSION_ADD_MAJOR;
            }
            break;
        case CAPTURE_CONVERT:
            ok = isAttack && result == CombatEffectType.WIN
                && natives != null
                && isEuropean() && defenderPlayer.isIndian();
            if (ok) {
                csCaptureConvert(attackerUnit, natives, random, cs);
                attackerTileDirty = true;
            }
            break;
        case CAPTURE_EQUIP:
            ok = isAttack && result != CombatEffectType.NO_RESULT;
            if (ok) {
                if (result == CombatEffectType.WIN) {
                    csCaptureEquip(attackerUnit, defenderUnit, cs);
                } else {
                    csCaptureEquip(defenderUnit, attackerUnit, cs);
                }
                attackerTileDirty = defenderTileDirty = true;
```

```
        }
        break;
    case CAPTURE_UNIT:
        ok = isAttack && result != CombatEffectType.NO_RESULT;
        if (ok) {
            if (result == CombatEffectType.WIN) {
                csCaptureUnit(attackerUnit, defenderUnit, cs);
            } else {
                csCaptureUnit(defenderUnit, attackerUnit, cs);
            }
            attackerTileDirty = true;
            defenderTileDirty = false; // Added in csCaptureUnit
        }
        break;
    case DAMAGE_COLONY_SHIPS:
        ok = isAttack && result == CombatEffectType.WIN
            && colony != null;
        if (ok) {
            csDamageColonyShips(attackerUnit, colony, cs);
            defenderTileDirty = true;
        }
        break;
    case DAMAGE_SHIP_ATTACK:
        ok = isAttack && result != CombatEffectType.NO_RESULT
            && ((result == CombatEffectType.WIN) ? defenderUnit
                : attackerUnit).isNaval();
        if (ok) {
            if (result == CombatEffectType.WIN) {
                csDamageShipAttack(attackerUnit, defenderUnit, cs);
                defenderTileDirty = true;
            } else {
                csDamageShipAttack(defenderUnit, attackerUnit, cs);
                attackerTileDirty = true;
            }
        }
        break;
    case DAMAGE_SHIP_BOMBARD:
        ok = isBombard && result == CombatEffectType.WIN
            && defenderUnit.isNaval();
        if (ok) {
            csDamageShipBombard(attackerSettlement, defenderUnit, cs);
            defenderTileDirty = true;
        }
        break;
    case DEMOTE_UNIT:
        ok = isAttack && result != CombatEffectType.NO_RESULT;
        if (ok) {
            if (result == CombatEffectType.WIN) {
                csDemoteUnit(attackerUnit, defenderUnit, cs);
                defenderTileDirty = true;
            } else {
```

```
                csDemoteUnit(defenderUnit, attackerUnit, cs);
                attackerTileDirty = true;
            }
        }
        break;
    case DESTROY_COLONY:
        ok = isAttack && result == CombatEffectType.WIN
            && colony != null
            && isIndian() && defenderPlayer.isEuropean();
        if (ok) {
            csDestroyColony(attackerUnit, colony, random, cs);
            attackerTileDirty = defenderTileDirty = true;
            moveAttacker = true;
            attackerTension -= Tension.TENSION_ADD_NORMAL;
            defenderTension += Tension.TENSION_ADD_MAJOR;
        }
        break;
    case DESTROY_SETTLEMENT:
        ok = isAttack && result == CombatEffectType.WIN
            && natives != null
            && defenderPlayer.isIndian();
        if (ok) {
            burnedNativeCapital = settlement.isCapital();
            csDestroySettlement(attackerUnit, natives, random, cs);
            attackerTileDirty = defenderTileDirty = true;
            moveAttacker = true;
            attackerTension -= Tension.TENSION_ADD_NORMAL;
            if (!burnedNativeCapital) {
                defenderTension += Tension.TENSION_ADD_MAJOR;
            }
        }
        break;
    case EVADE_ATTACK:
        ok = isAttack && result == CombatEffectType.NO_RESULT
            && defenderUnit.isNaval();
        if (ok) {
            csEvadeAttack(attackerUnit, defenderUnit, cs);
        }
        break;
    case EVADE_BOMBARD:
        ok = isBombard && result == CombatEffectType.NO_RESULT
            && defenderUnit.isNaval();
        if (ok) {
            csEvadeBombard(attackerSettlement, defenderUnit, cs);
        }
        break;
    case LOOT_SHIP:
        ok = isAttack && result != CombatEffectType.NO_RESULT
            && attackerUnit.isNaval() && defenderUnit.isNaval();
        if (ok) {
            if (result == CombatEffectType.WIN) {
```

```
                csLootShip(attackerUnit, defenderUnit, cs);
            } else {
                csLootShip(defenderUnit, attackerUnit, cs);
            }
        }
        break;
    case LOSE_AUTOEQUIP:
        ok = isAttack && result == CombatEffectType.WIN
            && settlement != null;
        if (ok) {
            csLoseAutoEquip(attackerUnit, defenderUnit, cs);
            defenderTileDirty = true;
        }
        break;
    case LOSE_EQUIP:
        ok = isAttack && result != CombatEffectType.NO_RESULT;
        if (ok) {
            if (result == CombatEffectType.WIN) {
                csLoseEquip(attackerUnit, defenderUnit, cs);
                defenderTileDirty = true;
            } else {
                csLoseEquip(defenderUnit, attackerUnit, cs);
                attackerTileDirty = true;
            }
        }
        break;
    case PILLAGE_COLONY:
        ok = isAttack && result == CombatEffectType.WIN
            && colony != null
            && isIndian() && defenderPlayer.isEuropean();
        if (ok) {
            csPillageColony(attackerUnit, colony, random, cs);
            defenderTileDirty = true;
            attackerTension -= Tension.TENSION_ADD_NORMAL;
        }
        break;
    case PROMOTE_UNIT:
        ok = isAttack && result != CombatEffectType.NO_RESULT;
        if (ok) {
            if (result == CombatEffectType.WIN) {
                csPromoteUnit(attackerUnit, cs);
                attackerTileDirty = true;
            } else {
                csPromoteUnit(defenderUnit, cs);
                defenderTileDirty = true;
            }
        }
        break;
    case SINK_COLONY_SHIPS:
        ok = isAttack && result == CombatEffectType.WIN
            && colony != null;
```

```
        if (ok) {
            csSinkColonyShips(attackerUnit, colony, cs);
            defenderTileDirty = true;
        }
        break;
    case SINK_SHIP_ATTACK:
        ok = isAttack && result != CombatEffectType.NO_RESULT
            && ((result == CombatEffectType.WIN) ? defenderUnit
                : attackerUnit).isNaval();
        if (ok) {
            if (result == CombatEffectType.WIN) {
                csSinkShipAttack(attackerUnit, defenderUnit, cs);
                defenderTileDirty = true;
            } else {
                csSinkShipAttack(defenderUnit, attackerUnit, cs);
                attackerTileDirty = true;
            }
        }
        break;
    case SINK_SHIP_BOMBARD:
        ok = isBombard && result == CombatEffectType.WIN
            && defenderUnit.isNaval();
        if (ok) {
            csSinkShipBombard(attackerSettlement, defenderUnit, cs);
            defenderTileDirty = true;
        }
        break;
    case SLAUGHTER_UNIT:
        ok = isAttack && result != CombatEffectType.NO_RESULT;
        if (ok) {
            if (result == CombatEffectType.WIN) {
                csSlaughterUnit(attackerUnit, defenderUnit, cs);
                defenderTileDirty = true;
                attackerTension -= Tension.TENSION_ADD_NORMAL;
                defenderTension += getSlaughterTension(defenderUnit);
            } else {
                csSlaughterUnit(defenderUnit, attackerUnit, cs);
                attackerTileDirty = true;
                attackerTension += getSlaughterTension(attackerUnit);
                defenderTension -= Tension.TENSION_ADD_NORMAL;
            }
        }
        break;
    default:
        ok = false;
        break;
    }
    if (!ok) {
        throw new IllegalStateException("Attack (result=" + result
                                        + ") has bogus subresult: "
                                        + cr);
```

```
        }
    }

    // Handle stance and tension.
    // - Privateers do not provoke stance changes but can set the
    //     attackedByPrivateers flag
    // - Attacks among Europeans imply war
    // - Burning of a native capital results in surrender
    // - Other attacks involving natives do not imply war, but
    //     changes in Tension can drive Stance, however this is
    //     decided by the native AI in their turn so just adjust tension.
    if (attacker.hasAbility(Ability.PIRACY)) {
        if (!defenderPlayer.getAttackedByPrivateers()) {
            defenderPlayer.setAttackedByPrivateers(true);
            cs.addPartial(See.only(defenderPlayer), defenderPlayer,
                "attackedByPrivateers", Boolean.TRUE.toString());
        }
    } else if (defender.hasAbility(Ability.PIRACY)) {
        ; // do nothing
    } else if (burnedNativeCapital) {
        defenderPlayer.getTension(this).setValue(Tension.SURRENDERED);
        // FIXME: just the tension
        cs.add(See.perhaps().always(this), defenderPlayer);
        csChangeStance(Stance.PEACE, defenderPlayer, true, cs);
                                        for     (IndianSettlement     is     :
transform(defenderPlayer.getIndianSettlements(),
                                                is -> 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 (hasExplored(is.getTile())) {
                cs.add(See.perhaps().always(this), is);
            } else {
                cs.add(See.only(defenderPlayer), is);
            }
        }
    } else if (isEuropean() && defenderPlayer.isEuropean()) {
        csChangeStance(Stance.WAR, defenderPlayer, true, cs);
    } else { // At least one player is non-European
        if (isEuropean()) {
            csChangeStance(Stance.WAR, defenderPlayer, true, cs);
        } else if (isIndian()) {
            if (result == CombatEffectType.WIN) {
                attackerTension -= Tension.TENSION_ADD_MINOR;
            } else if (result == CombatEffectType.LOSE) {
                attackerTension += Tension.TENSION_ADD_MINOR;
            }
        }
        if (defenderPlayer.isEuropean()) {
            ((ServerPlayer)defenderPlayer).csChangeStance(Stance.WAR, this, true,
cs);
```

```
    } else if (defenderPlayer.isIndian()) {
        if (result == CombatEffectType.WIN) {
            defenderTension += Tension.TENSION_ADD_MINOR;
        } else if (result == CombatEffectType.LOSE) {
            defenderTension -= Tension.TENSION_ADD_MINOR;
        }
    }
    if (attackerTension != 0) {
        this.csModifyTension(defenderPlayer,
                             attackerTension, cs);//+til
    }
    if (defenderTension != 0) {
        ((ServerPlayer)defenderPlayer).csModifyTension(this,
            defenderTension, cs);//+til
    }
}

// Move the attacker if required.
if (moveAttacker) {
    attackerUnit.setMovesLeft(attackerUnit.getInitialMovesLeft());
    ((ServerUnit) attackerUnit).csMove(defenderTile, random, cs);
    attackerUnit.setMovesLeft(0);
    // Move adds in updates for the tiles, but...
    attackerTileDirty = defenderTileDirty = false;
    // ...with visibility of perhaps().
    // Thus the defender might see the change,
    // but because its settlement is gone it also might not.
    // So add in another defender-specific update.
    // The worst that can happen is a duplicate update.
    cs.add(See.only(defenderPlayer), defenderTile);
} else if (isAttack) {
    // The Revenger unit can attack multiple times, so spend
    // at least the eventual cost of moving to the tile.
    // Other units consume the entire move.
    if (attacker.hasAbility(Ability.MULTIPLE_ATTACKS)) {
        int movecost = attackerUnit.getMoveCost(defenderTile);
        attackerUnit.setMovesLeft(attackerUnit.getMovesLeft()
                                  - movecost);
    } else {
        attackerUnit.setMovesLeft(0);
    }
    if (!attackerTileDirty) {
        cs.addPartial(See.only(this), attacker,
            "movesLeft", String.valueOf(attackerUnit.getMovesLeft()));
    }
}

// Make sure we always update the attacker and defender tile
// if it is not already done yet.
if (attackerTileDirty) {
    if (attackerSettlement != null) cs.remove(attackerSettlement);
```

```
            cs.add(vis, attackerTile);
        }
        if (defenderTileDirty) {
            if (settlement != null) cs.remove(settlement);
            cs.add(vis, defenderTile);
        }
    }
```

This method qualifies as the Code Smell Long Method as it undertakes too much and requires many lines of comments in order to understand what is being done. A way to fix this Code Smell would be to subdivide this method into smaller methods with their own comments that would then be called by the main method in order to achieve the necessary outcome, resulting in more comprehensible code.

Rodrigo Monteiro Suzana 63069 (suzana2314)

Speculative Generality

This smell is located in: *net.sf.freecol.server.generator.ColonizationMapLoader*

Looking specifically at the *ColonizationMapLoader* class, we can pinpoint the speculative class smell, notably in the methods ColonizationMapLoader(File file) throws IOException and Layer loadMap(Game game, Layer layer). This is due to no usage in the code at all, probably it was supposed to be an implementation for future updates. To address this issue, deleting this class would be the right choice.

Bellow is a code snippet of both methods:

```
public ColonizationMapLoader(File file) throws IOException {
    try (RandomAccessFile reader = new RandomAccessFile(file, "r")) {
        reader.readFully(header);
        int size = header[WIDTH] * header[HEIGHT];
        layer1 = new byte[size];
        reader.readFully(layer1);
    } catch (EOFException ee) {
        logger.log(Level.SEVERE, "File (" + file + ") is too short.", ee);
    } catch (FileNotFoundException fe) {

        logger.log(Level.SEVERE, "File (" + file + ") was not found.", fe);
    } catch (IOException e) {
         logger.log(Level.SEVERE, "File (" + file + ") is corrupt and cannot be
read.", e);
    }
public Layer loadMap(Game game, Layer layer) {
        Specification spec = game.getSpecification();
        Tile[][] tiles = new Tile[header[WIDTH]][header[HEIGHT]];
        Layer highestLayer = layer.compareTo(getHighestLayer()) < 0
            ? layer : getHighestLayer();
        int index = 0;
        TileType tileType = null;
        if (highestLayer == Layer.LAND) {
            // import only the land / water distinction
            for (int y = 0; y < header[HEIGHT]; y++) {
                for (int x = 0; x < header[WIDTH]; x++) {
                    int decimal = layer1[index] & 0xff;
                    int terrain = decimal & 0b11111;
                    tileType = (terrain == OCEAN || terrain == HIGH_SEAS) ?
                        TileType.WATER : TileType.LAND;
                    index++;
                }
            }
        } else {
                                            TileImprovementType    riverType    =
spec.getTileImprovementType("model.improvement.river");
            for (int y = 0; y < header[HEIGHT]; y++) {
                for (int x = 0; x < header[WIDTH]; x++) {
```

```
                    int decimal = layer1[index] & 0xff;
                    int terrain = decimal & 0b11111;
                    int overlay = decimal >> 5;
                    if (terrain < tiletypes.length) {
                                    tileType = spec.getTileType("model.tile." +
tiletypes[terrain]);
                    } else if (overlay == 1 || overlay == 3) {
                        tileType = spec.getTileType("model.tile.hills");
                    } else if (overlay == 5 || overlay == 7) {
                        tileType = spec.getTileType("model.tile.mountains");
                    }
                    tiles[x][y] = new Tile(game, tileType, x, y);
                    if (highestLayer == Layer.RIVERS
                            && (overlay == 2 || overlay == 3 || overlay == 6 ||
overlay == 7)) {
                                                TileItemContainer container = new
TileItemContainer(game, tiles[x][y]);
                        TileImprovement river =
                            new TileImprovement(game, tiles[x][y], riverType,

TileImprovementStyle.getInstance(TileImprovement.EMPTY_RIVER_STYLE));//TODO:
connections!
                        river.setMagnitude(overlay <= 3 ? 1 : 2);
                        container.tryAddTileItem(river);
                        tiles[x][y].setTileItemContainer(container);
                    }
                    index++;
                }
            }
        }
        return highestLayer;
    }
```

Long methods

This smell is located in: *net.sf.freecol.server.generator.ColonizationMapLoader*

Moreover, within the same class (*ColonizationMapLoader*), there's another code smell: long methods, particularly in the method Layer loadMap(Game game, Layer layer). As the name suggests this method is really long, the nested if-else conditions based on "overlay" and "terrain" might become overly complex and challenging to maintain as the code grows, also because this method isn't really finished because there is a TODO statement. Extracting some of this logic into separate methods or even utilizing switch cases could significantly enhance readability.

Bellow is a code snippet of the method in question:

```
public Layer loadMap(Game game, Layer layer) {
        Specification spec = game.getSpecification();
        Tile[][] tiles = new Tile[header[WIDTH]][header[HEIGHT]];
        Layer highestLayer = layer.compareTo(getHighestLayer()) < 0
```

```
            ? layer : getHighestLayer();
        int index = 0;
        TileType tileType = null;
        if (highestLayer == Layer.LAND) {
            // import only the land / water distinction
            for (int y = 0; y < header[HEIGHT]; y++) {
                for (int x = 0; x < header[WIDTH]; x++) {
                    int decimal = layer1[index] & 0xff;
                    int terrain = decimal & 0b11111;
                    tileType = (terrain == OCEAN || terrain == HIGH_SEAS) ?
                        TileType.WATER : TileType.LAND;
                    index++;
                }
            }
        } else {
                                            TileImprovementType    riverType    =
spec.getTileImprovementType("model.improvement.river");
            for (int y = 0; y < header[HEIGHT]; y++) {
                for (int x = 0; x < header[WIDTH]; x++) {
                    int decimal = layer1[index] & 0xff;
                    int terrain = decimal & 0b11111;
                    int overlay = decimal >> 5;
                    if (terrain < tiletypes.length) {
                                    tileType = spec.getTileType("model.tile." +
tiletypes[terrain]);
                    } else if (overlay == 1 || overlay == 3) {
                        tileType = spec.getTileType("model.tile.hills");
                    } else if (overlay == 5 || overlay == 7) {
                        tileType = spec.getTileType("model.tile.mountains");
                    }
                    tiles[x][y] = new Tile(game, tileType, x, y);
                    if (highestLayer == Layer.RIVERS
                            && (overlay == 2 || overlay == 3 || overlay == 6 ||
overlay == 7)) {
                                                TileItemContainer  container  =  new
TileItemContainer(game, tiles[x][y]);
                        TileImprovement river =
                            new TileImprovement(game, tiles[x][y], riverType,

TileImprovementStyle.getInstance(TileImprovement.EMPTY_RIVER_STYLE));//TODO:
connections!
                        river.setMagnitude(overlay <= 3 ? 1 : 2);
                        container.tryAddTileItem(river);
                        tiles[x][y].setTileItemContainer(container);
                    }
                    index++;
                }
            }
        }
        return highestLayer;
    }
```

Data Class

This smell is located in: *net.sf.freecol.common.model.CombatModel.CombatOdds*

This class is associated as a *data class* code smell primarily because it acts as a container for data with minimal functionality. It only stores a single value (win) and doesn't have any additional methods or behaviors associated with it. To address this, a simple solution involves moving the only variable to another class, or alternatively, either removing the class or adding more functionality to it.

Bellow is a code snippet of the class in question:

```
public static class CombatOdds {
        public static final double UNKNOWN_ODDS = -1.0;
        public final double win;
        public CombatOdds(double win) {
            this.win = win;
        }
    }
```

Rodrigo Fernandes 63191 (LemosFTW)

Speculative Generality

net.sf.freecol.server.ai.AIColony.getWorkerWishes()

Probably they created this method thinking about a future version, but they do not use this method, so you could remove this code Smell by removing this method, because it is never used.

```
public List<WorkerWish> getWorkerWishes() {
    return transform(wishes, w -> w instanceof WorkerWish,
                     w -> (WorkerWish)w);
}
```

Long Method

net.sf.freecol.client.gui.panel.report.ReportCompactColonyPanel.updateColony(ColonySummary)

This method is too long, it could be better to divide him in sub-methods dividing the complexity to other methods, i.e. there the key specific for each functionality in the if else, they could make a specific method that handles this key for specific cases.

```
private void updateColony(ColonySummary s) {
    final String cac = s.colony.getId();
    final UnitType defaultUnitType
        = spec.getDefaultUnitType(s.colony.getOwner());
    List<JComponent> buttons = new ArrayList<>(16);
    JButton b;
    Color c;
    StringTemplate t;
    Building building;

    // Field: A button for the colony.
    // Colour: bonus in {-2,2} => {alarm, warn, plain, export, good}
    // Font: Bold if famine is threatening.
    c = (s.bonus <= -2) ? cAlarm
        : (s.bonus == -1) ? cWarn
        : (s.bonus == 0) ? cPlain
        : (s.bonus == 1) ? cExport
        : cGood;
    String annotations = "", key;
    t = StringTemplate.label(",");
    if ((building = s.colony.getStockade()) == null) {
        key = "annotation.unfortified";
        t.add(Messages.message("report.colony.annotation.unfortified"));
    } else {
```

```
        key = "annotation." + building.getType().getSuffix();
        t.add(Messages.message(building.getLabel()));
    }
    if (ResourceManager.getStringResource(key, false) != null) {
        annotations += ResourceManager.getString(key);
    }
    if (!s.colony.getTile().isCoastland()) {
        key = "annotation.inland";
        t.add(Messages.message("report.colony.annotation.inland"));
                            }         else         if         ((building         =
s.colony.getWorkLocationWithAbility(Ability.PRODUCE_IN_WATER,   Building.class))
== null) {
        key = "annotation.coastal";
        t.add(Messages.message("report.colony.annotation.coastal"));
    } else {
        key = "annotation." + building.getType().getSuffix();
        t.add(Messages.message(building.getLabel()));
    }
    if (ResourceManager.getStringResource(key, false) != null) {
        annotations += ResourceManager.getString(key);
    }
    /* Omit for now, too much detail.
       for (GoodsType gt : spec.getLibertyGoodsTypeList()) {
              if  ((building  =  s.colony.getWorkLocationWithModifier(gt.getId(),
Building.class)) != null) {
      key = "annotation." + building.getType().getSuffix();
      t.add(Messages.message(building.getLabel()));
      if (ResourceManager.hasResource(key))
      annotations += ResourceManager.getString(key);
      }
      }*/
    /* Omit for now, too much detail.
       for (GoodsType gt : spec.getImmigrationGoodsTypeList()) {
              if  ((building  =  s.colony.getWorkLocationWithModifier(gt.getId(),
Building.class)) != null) {
      key = "annotation." + building.getType().getSuffix();
      t.add(Messages.message(building.getLabel()));
      if (ResourceManager.hasResource(key))
      annotations += ResourceManager.getString(key);
      }
      }*/
    /* Font update needed
            if  ((building  =  s.colony.getWorkLocationWithAbility(Ability.TEACH,
Building.class)) != null) {
      key = "annotation." + building.getType().getSuffix();
      t.add(Messages.message(building.getLabel()));
                    if    (ResourceManager.hasResource(key))    annotations    +=
ResourceManager.getString(key);
      }*/
        if   ((building   =   s.colony.getWorkLocationWithAbility(Ability.EXPORT,
Building.class)) != null) {
```

```
            annotations += "*";
            t.add(Messages.message(building.getLabel()));
        }
        b = newButton(cac, s.colony.getName() + annotations, null, c,
            StringTemplate.label(": ").add(s.colony.getName())
            .add(Messages.message(t)));
        if (s.famine) b.setFont(b.getFont().deriveFont(Font.BOLD));
        reportPanel.add(b, "newline");

        // Field: Size
        c = cGood;
        t = stpld("report.colony.size");
        reportPanel.add(newButton(cac, Integer.toString(s.unitCount), null, c, t));

        // Field: The number of colonists that can be added to a
        // colony without damaging the production bonus
        if (s.unitsToAdd > 0) {
            c = cGood;
            t = stpld("report.colony.growing")
                .addName("%colony%", s.colony.getName())
                .addAmount("%amount%", s.unitsToAdd);
            b = newButton(cac, Integer.toString(s.unitsToAdd), null, c, t);
        } else {
            b = null;
        }
        reportPanel.add((b == null) ? new JLabel() : b);

        // Field: the number of colonists to remove to fix the inefficiency.
        // Colour: Blue if efficient/Red if inefficient.
        if (s.unitsToRemove > 0) {
            c = s.bonus < 0 ? cAlarm : cGood;
            t = stpld("report.colony.shrinking")
                .addName("%colony%", s.colony.getName())
                .addAmount("%amount%", s.unitsToRemove);
            b = newButton(cac, Integer.toString(s.unitsToRemove), null, c, t);
        } else {
            b = null;
        }
        reportPanel.add((b == null) ? new JLabel() : b);

        // Field: The number of potential colony tiles that need
        // exploring.
        // Colour: Always cAlarm
        int n = count(s.tileSuggestions,
            TileImprovementSuggestion::isExploration);
        if (n > 0) {
            t = stpld("report.colony.exploring")
                .addName("%colony%", s.colony.getName())
                .addAmount("%amount%", n);
            b = newButton(cac, Integer.toString(n), null, cAlarm, t);
        } else {
```

```
    b = null;
}
reportPanel.add((b == null) ? new JLabel() : b);

// Fields: The number of existing colony tiles that would
// benefit from improvements.
// Colour: Always cAlarm
// Font: Bold if one of the tiles is the colony center.
for (TileImprovementType ti : spec.getTileImprovementTypeList()) {
    if (ti.isNatural()) continue;
    n = 0;
    boolean center = false;
    for (TileImprovementSuggestion tis : s.tileSuggestions) {
        if (tis.tileImprovementType == ti) {
            n++;
            if (tis.tile == s.colony.getTile()) center = true;
        }
    }
    if (n > 0) {
        c = cAlarm;
        if (n == 1) {
            TileImprovementSuggestion tis = first(s.tileSuggestions);
            if (any(tis.tile.getUnits(),
                    u -> (u.getState() == Unit.UnitState.IMPROVING
                        && u.getWorkImprovement() != null
                        && u.getWorkImprovement().getType()
                        == tis.tileImprovementType))) {
                c = cWarn; // Work is underway
            }
            t = stpld("report.colony.tile." + ti.getSuffix()
                + ".specific")
                .addName("%colony%", s.colony.getName())
                .addStringTemplate("%location%",
                    tis.tile.getColonyTileLocationLabel(s.colony));
        } else {
            t = stpld("report.colony.tile." + ti.getSuffix())
                .addName("%colony%", s.colony.getName())
                .addAmount("%amount%", n);
        }
        b = newButton(cac, Integer.toString(n), null, c, t);
        if (center) b.setFont(b.getFont().deriveFont(Font.BOLD));
    } else {
        b = null;
    }
    reportPanel.add((b == null) ? new JLabel() : b);
}

// Fields: The net production of each storable+non-trade-goods
// goods type.
// Colour: cAlarm if too low, cWarn if negative, empty if no
// production, cPlain if production balanced at zero,
```

```
// otherwise must be positive, wherein cExport
// if exported, cAlarm if too high, else cGood.
for (GoodsType gt : this.goodsTypes) {
    final ColonySummary.GoodsProduction gp = s.production.get(gt);
    switch (gp.status) {
    case FAIL:
        c = cAlarm;
        t = stpld("report.colony.production.low")
            .addName("%colony%", s.colony.getName())
            .addNamed("%goods%", gt)
            .addAmount("%amount%", -gp.amount)
            .addAmount("%turns%", gp.extra);
        break;
    case BAD:
        c = cWarn;
        t = stpld("report.colony.production")
            .addName("%colony%", s.colony.getName())
            .addNamed("%goods%", gt)
            .addAmount("%amount%", gp.amount);
        break;
    case NONE:
        c = null;
        t = null;
        break;
    case ZERO:
        c = cPlain;
        t = stpld("report.colony.production")
            .addName("%colony%", s.colony.getName())
            .addNamed("%goods%", gt)
            .addAmount("%amount%", gp.amount);
        break;
    case GOOD:
        c = cGood;
        t = stpld("report.colony.production")
            .addName("%colony%", s.colony.getName())
            .addNamed("%goods%", gt)
            .addAmount("%amount%", gp.amount);
        break;
    case EXPORT:
        c = cExport;
        t = stpld("report.colony.production.export")
            .addName("%colony%", s.colony.getName())
            .addNamed("%goods%", gt)
            .addAmount("%amount%", gp.amount)
            .addAmount("%export%", gp.extra);
        break;
    case EXCESS:
        c = cWarn;
        t = stpld("report.colony.production.high")
            .addName("%colony%", s.colony.getName())
            .addNamed("%goods%", gt)
```

```
                .addAmount("%amount%", gp.amount)
                .addAmount("%turns%", gp.extra);
            break;
        case OVERFLOW:
            c = cAlarm;
            t = stpld("report.colony.production.waste")
                .addName("%colony%", s.colony.getName())
                .addNamed("%goods%", gt)
                .addAmount("%amount%", gp.amount)
                .addAmount("%waste%", gp.extra);
            break;
        case PRODUCTION:
            c = cWarn;
            t = stpld("report.colony.production.maxProduction")
                .addName("%colony%", s.colony.getName())
                .addNamed("%goods%", gt)
                .addAmount("%amount%", gp.amount)
                .addAmount("%more%", gp.extra);
            break;
        case CONSUMPTION:
            c = cWarn;
            t = stpld("report.colony.production.maxConsumption")
                .addName("%colony%", s.colony.getName())
                .addNamed("%goods%", gt)
                .addAmount("%amount%", gp.amount)
                .addAmount("%more%", gp.extra);
            break;
        default:
            throw new IllegalStateException("Bogus status: " + gp.status);
        }
        reportPanel.add((c == null) ? new JLabel()
            : newButton(cac, Integer.toString(gp.amount), null, c, t));
    }

    // Field: New colonist arrival or famine warning.
    // Colour: cGood if arriving eventually, blank if not enough food
    // to grow, cWarn if negative, cAlarm if famine soon.
    if (s.newColonist > 0) {
        t = stpld("report.colony.arriving")
            .addName("%colony%", s.colony.getName())
            .addNamed("%unit%", defaultUnitType)
            .addAmount("%turns%", s.newColonist);
        b = newButton(cac, Integer.toString(s.newColonist), null,
            cGood, t);
    } else if (s.newColonist < 0) {
        c = (s.famine) ? cAlarm : cWarn;
        t = stpld("report.colony.starving")
            .addName("%colony%", s.colony.getName())
            .addAmount("%turns%", -s.newColonist);
        b = newButton(cac, Integer.toString(-s.newColonist), null,
            c, t);
```

```
    if (s.famine) b.setFont(b.getFont().deriveFont(Font.BOLD));
} else {
    b = null;
}
reportPanel.add((b == null) ? new JLabel() : b);

// Field: What is currently being built (clickable if on the
// buildqueue) and the turns until it completes, including
// units being taught, or blank if nothing queued.
// Colour: cWarn if no construction is occurring, cGood with
// turns if completing, cAlarm with turns if will block, turns
// indicates when blocking occurs.
// Font: Bold if blocked right now.
final String qac = BUILDQUEUE + cac;
if (s.build != null) {
    int turns = s.completeTurns;
    String bname = Messages.getName(s.build);
    if (turns == UNDEFINED) {
        t = stpld("report.colony.making.noconstruction")
            .addName("%colony%", s.colony.getName());
        b = newButton(qac, bname, null, cWarn, t);
    } else if (turns >= 0) {
        t = stpld("report.colony.making.constructing")
            .addName("%colony%", s.colony.getName())
            .addNamed("%buildable%", s.build)
            .addAmount("%turns%", turns);
        b = newButton(qac, bname + " " + Integer.toString(turns), null,
            cGood, t);
    } else { // turns < 0
        turns = -(turns + 1);
        t = stpld("report.colony.making.blocking")
            .addName("%colony%", s.colony.getName())
            .addAmount("%amount%", s.needed.getAmount())
            .addNamed("%goods%", s.needed.getType())
            .addNamed("%buildable%", s.build)
            .addAmount("%turns%", turns);
        b = newButton(qac, bname + " " + Integer.toString(turns),
            null, cAlarm, t);
        if (turns == 0) b.setFont(b.getFont().deriveFont(Font.BOLD));
    }
    buttons.add(b);
}

// Field: What is being trained, including shadow units for vacant
// places.
// Colour: cAlarm if completion is blocked, otherwise cPlain.
int empty = 0;
Building school = s.colony.getWorkLocationWithAbility(Ability.TEACH,
    Building.class);
if (school != null) empty = school.getType().getWorkPlaces();
for (Entry<Unit, Integer> e
```

```
                : mapEntriesByValue(s.teachers, descendingIntegerComparator)) {
            Unit u = e.getKey();
            ImageIcon ii = new ImageIcon(this.lib.getTinyUnitImage(u));
            if (e.getValue() <= 0) {
                t = stpld("report.colony.making.noteach")
                    .addName("%colony%", s.colony.getName())
                    .addStringTemplate("%teacher%",
                        u.getLabel(Unit.UnitLabelType.NATIONAL));
                b = newButton(cac, Integer.toString(0), ii, cAlarm, t);
            } else {
                t = stpld("report.colony.making.educating")
                    .addName("%colony%", s.colony.getName())
                    .addStringTemplate("%teacher%",
                        u.getLabel(Unit.UnitLabelType.NATIONAL))
                    .addAmount("%turns%", e.getValue());
                b = newButton(cac, Integer.toString(e.getValue()), ii,
                    cPlain, t);
            }
            buttons.add(b);
            empty--;
        }

        if (empty > 0) {
            final ImageIcon emptyIcon
                    = new ImageIcon(this.lib.getTinyUnitTypeImage(defaultUnitType,
true));
            t = stpld("report.colony.making.educationVacancy")
                .addName("%colony%", s.colony.getName())
                .addAmount("%number%", empty);
            for (; empty > 0; empty--) {
                buttons.add(newButton(cac, "", emptyIcon, cPlain, t));
            }
        }
        addTogether(buttons);

        // Field: The units that could be upgraded, followed by the units
        // that could be added.
        if (s.improve.isEmpty() && s.want.isEmpty()) {
            reportPanel.add(new JLabel());
        } else {
            buttons.clear();
            buttons.addAll(unitButtons(s.improve, s.couldWork, s.colony));
            buttons.add(new JLabel("/"));
            // Prefer to suggest an improvement over and addition.
            for (UnitType ut : s.improve.keySet()) s.want.remove(ut);
            buttons.addAll(unitButtons(s.want, s.couldWork, s.colony));
            addTogether(buttons);
        }
    }
```

Data Class

net.sf.freecol.server.ai.ColonyPlan.BuildPlan

This class only contains data and no real functionality, only the getter method. This class may have more functionalities inside, that are now in responsibility of other classes.

```
private static class BuildPlan {

    public final BuildableType type;
    public double weight;
    public double support;
    public double difficulty;
    public BuildPlan(BuildableType type, double weight, double support) {
        this.type = type;
        this.weight = weight;
        this.support = support;
        this.difficulty = 1.0f;
    }
    public double getValue() {
        return weight * support / difficulty;
    }
```

João Pedro Silveira 62654 (Joao-Pedro-Silveira)

Data Clumps

The methods:

```
net.sf.freecol.common.model.Map.search((final Unit unit, Location start,
                            final GoalDecider goalDecider,
                            final CostDecider costDecider,
                            final int maxTurns, final Unit carrier,
                            LogBuilder lb))
```

and

```
net.sf.freecol.common.model.Map.searchMap(final Unit unit, final Tile start,
                                final GoalDecider goalDecider,
                                final CostDecider costDecider,
                                final int maxTurns, final Unit carrier,
                                final SearchHeuristic searchHeuristic,
                                final LogBuilder lb)
```

have the code smell DataClumps since both of them share the fields unit, start, goalDecider, costDecider, maxTurns, carrier, lb, this could be fixed by adding a new class containing all of these fields. In it we could create a
method to get the current unit

```
( Unit currentUnit = (start.isLand())
            ? ((unit != null && unit.getLocation() == carrier
                    && start.hasSettlement()
                    && start.getSettlement().isConnectedPort())
                ? carrier : unit)
            : offMapUnit;
)
```

, this will also decrease the complexity of the method searchMap.

```
public PathNode search(final Unit unit, Location start,
                       final GoalDecider goalDecider,
                       final CostDecider costDecider,
                       final int maxTurns, final Unit carrier,
                       LogBuilder lb) {
    if (traceSearch) lb = new LogBuilder(1024);

    final Unit offMapUnit = (carrier != null) ? carrier : unit;

    PathNode p, path;
    if (start instanceof Europe) {
        // Fail fast if Europe is unattainable.
        if (offMapUnit == null
            || !offMapUnit.getType().canMoveToHighSeas()) {
            path = null;

            // This is suboptimal.  We do not know where to enter from
            // Europe, so start with the standard entry location...
        } else if ((p = searchMap(unit,
                    offMapUnit.getFullEntryLocation(),
                    goalDecider, costDecider, maxTurns, carrier,
                    null, lb)) == null) {
            path = null;

            // ...then if we find a path, try to optimize it.  This
            // will lose if the initial search fails due to a turn limit.
            // FIXME: do something better.
        } else {
            path = this.findPath(unit, start, p.getLastNode().getTile(),
                                 carrier, costDecider, lb);
        }

    } else {
        path = searchMap(unit, start.getTile(), goalDecider,
                         costDecider, maxTurns, carrier, null, lb);
    }

    finishPath(path, unit, lb);
    return path;
 }
```

```
private PathNode searchMap(final Unit unit, final Tile start,
                           final GoalDecider goalDecider,
                           final CostDecider costDecider,
                           final int maxTurns, final Unit carrier,
                           final SearchHeuristic searchHeuristic,
                           final LogBuilder lb) {
        final HashMap<String, PathNode> openMap = new HashMap<>();
        final HashMap<String, PathNode> closedMap = new HashMap<>();
        final HashMap<String, Integer> f = new HashMap<>();
        final PriorityQueue<PathNode> openMapQueue = new PriorityQueue<>(1024,
            Comparator.comparingInt(p -> f.get(p.getLocation().getId())));
        final SearchHeuristic sh = (searchHeuristic == null)
            ? trivialSearchHeuristic : searchHeuristic;
        final Unit offMapUnit = (carrier != null) ? carrier : unit;
        Unit currentUnit = (start.isLand())
            ? ((unit != null && unit.getLocation() == carrier
                    && start.hasSettlement()
                    && start.getSettlement().isConnectedPort())
                ? carrier : unit)
            : offMapUnit;
        if (lb != null) lb.add("Search trace(unit=", unit,
            ", from=", start,
            ", max=", ((maxTurns == INFINITY)?"-":Integer.toString(maxTurns)),
            ", carrier=", carrier, ")",
                                                                    "\n",
net.sf.freecol.common.debug.FreeColDebugger.stackTraceToString());

        // Create the start node and put it on the open list.
        final PathNode firstNode = new PathNode(start,
            ((currentUnit != null) ? currentUnit.getMovesLeft() : -1),
            0, carrier != null && currentUnit == carrier, null, null);
        f.put(start.getId(), sh.getValue(start));
        openMap.put(start.getId(), firstNode);
        openMapQueue.offer(firstNode);

        PathNode best = null;
        int bestScore = INFINITY;
 ok:     while (!openMap.isEmpty()) {
            // Choose the node with the lowest f.
            final PathNode currentNode = openMapQueue.poll();
            final Location currentLocation = currentNode.getLocation();
            openMap.remove(currentLocation.getId());
            if (lb != null) lb.add("\n  ", currentNode);

            // Reset current unit to that of this node.
            currentUnit = (currentNode.isOnCarrier()) ? carrier : unit;

            // Check for success.
            if (goalDecider.check(currentUnit, currentNode)) {
                if (lb != null) lb.add(" ***goal(",
                    currentNode.getCost(), ")***");
```

```
            best = goalDecider.getGoal();
            if (best == null || !goalDecider.hasSubGoals()) break ok;
            bestScore = best.getCost();
            continue;
        }

        // Valid candidate for the closed list.
        closedMap.put(currentLocation.getId(), currentNode);
        if (lb != null) lb.add(" closing");

        // Skip nodes that can not beat the current best path.
        if (bestScore < currentNode.getCost()) {
            if (lb != null) lb.add("...goal cost wins(",
                bestScore, " < ", currentNode.getCost(), ")...");
            continue;
        }

        // Ignore nodes over the turn limit.
        if (currentNode.getTurns() > maxTurns) {
            if (lb != null) lb.add("...out-of-range");
            continue;
        }

        // Collect the parameters for the current node.
        final int currentMovesLeft = currentNode.getMovesLeft();
        final int currentTurns = currentNode.getTurns();
        final boolean currentOnCarrier = currentNode.isOnCarrier();

        final Tile currentTile = currentNode.getTile();
        if (currentTile == null) { // Must be in Europe.
            // FIXME: Do not consider tiles "adjacent" to Europe, yet.
            // There may indeed be cases where going to Europe and
            // coming back on the other side of the map is faster.
            if (lb != null) lb.add("...skip Europe");
            continue;
        }

        // Try the tiles in each direction
        PathNode closed;
        for (Tile moveTile : currentTile.getSurroundingTiles(1)) {
            // If the new tile is the tile we just visited, skip it.
            if (lb != null) lb.add("\n    ", moveTile);
            if (currentNode.previous != null
                && currentNode.previous.getTile() == moveTile) {
                if (lb != null) lb.add(" !prev");
                continue;
            }

            // Skip neighbouring tiles already too expensive.
            closed = closedMap.get(moveTile.getId());
            if (closed != null) {
```

```
                    int cc = closed.getCost();
                    if (cc <= currentNode.getCost()) {
                        if (lb != null) lb.add(" !worse ", cc);
                        continue;
                    }
                }

                // Prepare to consider move validity
                Unit.MoveType umt = unit.getSimpleMoveType(currentTile,
                                                           moveTile);
                boolean unitMove = umt.isProgress();
                boolean carrierMove = carrier != null
                                    &&  carrier.getSimpleMoveType(carrier.getTile(),
moveTile).isProgress();
                if (lb != null) lb.add(" ", ((unitMove) ? "U"
                        : ((carrierMove) ? "C" : "")));
                MoveCandidate move;
                String stepLog;

                // Is this move to the goal?  Use fake high cost so
                // this does not become cached inside the goal decider
                // as the preferred path.
                boolean isGoal = goalDecider.check(unit,
                    new PathNode(moveTile, 0, INFINITY/2, false,
                                 currentNode, null));
                if (isGoal) {
                    if (lb != null) lb.add(" *goal*", umt);
                    if (unitMove) {
                              if (moveTile.hasSettlement() && currentOnCarrier &&
carrierMove) {
                            // If the goal has a settlement and the
                            // unit is travelling by carrier, dock the
                            // carrier.
                            move = new MoveCandidate(carrier, currentNode,
                                moveTile, currentMovesLeft, currentTurns,
                                true, CostDeciders.tileCost());
                        } else {
                            // Otherwise let the unit complete the path.
                            int left = (currentOnCarrier)
                                ? ((currentNode.embarkedThisTurn(currentTurns))
                                    ? 0
                                    : unit.getInitialMovesLeft())
                                : currentMovesLeft;
                            move = new MoveCandidate(unit, currentNode,
                                moveTile, left, currentTurns, false,
                                CostDeciders.tileCost());
                        }
                    } else {
                        // Handle some special cases where the move may not
                        // necessarily progress, but still can do useful work.
                        switch (umt) {
```

```
case ATTACK_UNIT:
case ATTACK_SETTLEMENT:
case ENTER_FOREIGN_COLONY_WITH_SCOUT:
case ENTER_INDIAN_SETTLEMENT_WITH_SCOUT:
case ENTER_INDIAN_SETTLEMENT_WITH_FREE_COLONIST:
case ENTER_INDIAN_SETTLEMENT_WITH_MISSIONARY:
case ENTER_SETTLEMENT_WITH_CARRIER_AND_GOODS:
    // Can not move to the tile, but there is
    // a valid interaction with the unit or
    // settlement that is there.
    move = new MoveCandidate(unit, currentNode,
        moveTile, currentMovesLeft, currentTurns,
        false, CostDeciders.tileCost());
    unitMove = true;
    break;
case EMBARK:
    move = new MoveCandidate(unit, currentNode,
        moveTile, currentMovesLeft, currentTurns,
        true, CostDeciders.tileCost());
    unitMove = true;
    break;
case MOVE_NO_ATTACK_CIVILIAN:
    // There is a settlement in the way, this
    // path can never succeed.
    if (moveTile.hasSettlement()) {
        if (lb != null) lb.add(" !FAIL-SETTLEMENT");
        if (!goalDecider.hasSubGoals()) break ok;
        continue;
    }
    // There is a unit in the way.  Unless this
    // unit can arrive there this turn, assume the
    // condition is transient as long as the tile
    // is not in a constrained position such as a
    // small island or river.
    if (currentNode.getTurns() <= 0
        || moveTile.getAvailableAdjacentCount() < 3) {
        if (lb != null) lb.add(" !FAIL-ATTACK");
        if (!goalDecider.hasSubGoals()) break ok;
        continue;
    }
    if (lb != null) lb.add(" blocked");
    unitMove = true;
    move = new MoveCandidate(unit, currentNode,
        moveTile, currentMovesLeft, currentTurns,
        false, CostDeciders.tileCost());
    break;
default:
    // Several cases here, these are understood:
    // MOVE_NO_ATTACK_EMBARK:
    //   Land unit trying to use water, which
    //   can not work without a ship there
```

```
                // MOVE_NO_ACCESS_WATER:
                //   The unit can not disembark directly to
                //   the goal along this path
                // MOVE_NO_ATTACK_MARINE:
                //   Ampibious attack disallowed, disembark to
                //   reach the goal
                // There will be more.
                // We used to do---
                //   if (!goalDecider.hasSubGoals()) break ok;
                // --- here, like in the transient failure case
                // above but we should not truncate other
                // surrounding tiles.
                if (lb != null) lb.add(" !FAIL-", umt);
                continue;
            }
        }
        assert move != null;
        stepLog = "@";
    } else {
        // Ordinary non-goal moves.
        //
        // Check for a carrier change at the new tile,
        // creating a MoveCandidate for each case.
        //
        // Do *not* allow units to re-embark on the carrier.
        // Note that embarking can actually increase the moves
        // left because the carrier might be not have spent
        // any moves yet that turn.
        //
        // Note that we always favour using the carrier if
        // both carrier and non-carrier moves are possible,
        // which can only be true moving into a settlement.
        // Usually when moving into a settlement it will be
        // useful to dock the carrier so it can collect new
        // cargo.  OTOH if the carrier is just passing through
        // the right thing is to keep the passenger on board.
        // However, see the goal settlement exception above.
        MoveStep step = (currentOnCarrier)
            ? ((carrierMove) ? MoveStep.BYWATER
                : (unitMove) ? MoveStep.DISEMBARK
                : MoveStep.FAIL)
            : (carrierMove && !usedCarrier(currentNode))
            ? MoveStep.EMBARK
            : (unitMove) ? ((unit.isNaval())
                ? MoveStep.BYWATER
                : MoveStep.BYLAND)
            : MoveStep.FAIL;
        switch (step) {
        case BYLAND:
            move = new MoveCandidate(unit, currentNode, moveTile,
                currentMovesLeft, currentTurns, false, costDecider);
```

```
                        break;
                    case BYWATER:
                                move = new MoveCandidate(offMapUnit, currentNode,
moveTile,
                            currentMovesLeft, currentTurns, currentOnCarrier,
                            costDecider);
                        break;
                    case EMBARK:
                                move = new MoveCandidate(offMapUnit, currentNode,
moveTile,
                            currentMovesLeft, currentTurns, true,
                            costDecider);
                        move.embarkUnit(carrier);
                        break;
                    case DISEMBARK:
                        move = new MoveCandidate(unit, currentNode, moveTile,
                            0, currentTurns, false, costDecider);
                        break;
                    case FAIL: default: // Loop on failure.
                        if (lb != null) lb.add("!");
                        continue;
                    }
                    stepLog = " " + step + "_";
                }
                if (move.cost >= INFINITY) {
                    continue;
                }
                assert move.getCost() >= 0;
                // Tighten the bounds on a previously seen case if possible
                if (closed != null) {
                    if (move.canImprove(closed)) {
                        closedMap.remove(moveTile.getId());
                        move.improve(openMap, openMapQueue, f, sh);
                        stepLog += "^" + Integer.toString(move.getCost());
                    } else {
                        stepLog += "v";
                    }
                } else {
                    if (move.canImprove(openMap.get(moveTile.getId()))){
                        move.improve(openMap, openMapQueue, f, sh);
                        stepLog += "+" + Integer.toString(move.getCost());
                    } else {
                        stepLog += "-";
                    }
                }
                if (lb != null) lb.add(stepLog);
            }
        }
```

Message chains

The method net.sf.freecol.server.ai.EuropeanAIPlayer.startWorking() has two very long and convoluted message chains:

```
final                List<AIUnit>                normalAiUnits                =
etAIUnits().stream().filter(MilitaryCoordinator.isUnitHandledByMilitaryCoordinat
or().negate()).collect(Collectors.toList());
final                Set<AIUnit>                militaryUnits                =
getAIUnits().stream().filter(MilitaryCoordinator.isUnitHandledByMilitaryCoordina
tor()).collect(Collectors.toSet());
```

which add to the complexity of the code making it hard to read and making the code rigid. One way of solving this smell would be to explain each variable by separating them or by creating methods that solve them separately.

Divergent Class

The class net.sf.freecol.common.model.Map is a divergent class since it does many more things than just defining the map, as it is also used to make searches directly in it (like in the method net.sf.freecol.common.model.Map.searchMap(final Unit unit, final Tile start,
final GoalDecider goalDecider,
final CostDecider costDecider,
final int maxTurns, final Unit carrier,
final SearchHeuristic searchHeuristic,
final LogBuilder lb),

and

```
net.sf.freecol.common.model.Map.search((final Unit unit, Location start,
                            final GoalDecider goalDecider,
                            final CostDecider costDecider,
                            final int maxTurns, final Unit carrier,
                            LogBuilder lb))).
```

This makes the class more complex than it needs to be.
One way of solving this code smell would be to create other classes that would take care of making such searches using the Map.
(*The code for the methods mentioned has been already shown in the beginning of this doc*)

Tiago Sousa 63324 (tiago-cos)

Long Method

The following Long Method code smell was identified:

```
/**
     * Tries to apply a colony plan given a list of workers.
     *
     * @param workers A list of {@code Unit}s to assign.
     * @param preferScout Prefer to make scouts rather than soldiers.
     * @param lb A {@code LogBuilder} to log to.
     * @return A scratch colony with the workers in place.
     */
    public Colony assignWorkers(List<Unit> workers, boolean preferScout,
                                LogBuilder lb) {
        final GoodsType foodType = spec().getPrimaryFoodType();

        // Collect the work location plans.  Note that the plans are
        // pre-sorted in order of desirability.
        final List<GoodsType> produce = getPreferredProduction();
        List<WorkLocationPlan> foodPlans = getFoodPlans();
        List<WorkLocationPlan> workPlans = getWorkPlans();

        // Make a scratch colony to work on.
        Colony col = colony.copyColony();
        Tile tile = col.getTile();

        // Replace the given workers with those in the scratch colony.
        List<Unit> otherWorkers = new ArrayList<>(workers);
        workers.clear();
        for (Unit u : otherWorkers) workers.add(col.getCorresponding(u));

        // Move all workers to the tile.
        // Also remove equipment, which is safe because no missionaries
        // or active pioneers should be on the worker list.
        for (Unit u : workers) {
            u.setLocation(tile);
            col.equipForRole(u, spec().getDefaultRole(), 0);
        }

        // Move outdoor experts outside if possible.
        // Prefer scouts in early game if there are very few.
        Role[] outdoorRoles = {
            spec().getRoleWithAbility(Ability.IMPROVE_TERRAIN, null),
            null,
            spec().getRoleWithAbility(Ability.SPEAK_WITH_CHIEF, null)
        };
        if (preferScout) {
            Role tmp = outdoorRoles[1];
            outdoorRoles[1] = outdoorRoles[2];
```

```
            outdoorRoles[2] = tmp;
        }
        for (Role outdoorRole : outdoorRoles) {
            for (Unit u : new ArrayList<>(workers)) {
                if (workers.size() <= 1) break;
                Role role = outdoorRole;
                if (role == null) {
                    for (Role r : u.getSortedMilitaryRoles()) {
                                    if (u.getType() == r.getExpertUnit() &&
fullEquipUnit(spec(), u, r, col)) {
                            workers.remove(u);
                            lb.add(u.getId(), "(", u.getType().getSuffix(),
                                ") -> ", r.getSuffix(), "\n");
                            break;
                        }
                    }
                           } else if (u.getType() == role.getExpertUnit() &&
fullEquipUnit(spec(), u, role, col)) {
                    workers.remove(u);
                    lb.add(u.getId(), "(", u.getType().getSuffix(),
                        ") -> ", role.getSuffix(), "\n");
                }
            }
        }

        // Consider the defence situation.
        // FIXME: scan for neighbouring hostiles
        // Favour low-skill units for defenders, then order experts
        // in reverse order of their production on the produce-list,
        // and finally by least experience.
        final Comparator<Unit> soldierComparator
            = Comparator.<Unit>comparingInt(Unit::getSkillLevel)
                .thenComparingInt(u ->
                    (u.getType().getExpertProduction() == null) ? 1 : 0)
                .thenComparingInt(u ->
                    produce.indexOf(u.getType().getExpertProduction()))
                .reversed()
                .thenComparingInt(Unit::getExperience);
        /*
         * Defence is now handled by the military coordinator.
         *
        for (Unit u : sort(workers, soldierComparator)) {
            if (workers.size() <= 1) break;
            if (!col.isBadlyDefended()) break;
            if (u.getSkillLevel() > 0) {
                // Stops experts from being used as soldiers
                continue;
            }
            for (Role role : u.getSortedMilitaryRoles()) {
                if (role != null && fullEquipUnit(spec(), u, role, col)) {
                    workers.remove(u);
```

```
                lb.add(u.getId(), "(", u.getType().getSuffix(), ") -> ",
                        u.getRoleSuffix(), "\n");
                break;
            }
        }
    }
    */

    // Greedy assignment of other workers to plans.
    List<AbstractGoods> buildGoods = new ArrayList<>();
    BuildableType build = col.getCurrentlyBuilding();
    if (build != null) buildGoods.addAll(build.getRequiredGoodsList());
    List<WorkLocationPlan> wlps;
    WorkLocationPlan wlp;
    boolean done = false;
    while (!done && !workers.isEmpty()) {
        // Decide what to produce: set the work location plan to
        // try (wlp), and the list the plan came from so it can
        // be recycled if successful (wlps).
        wlps = null;
        wlp = null;
        if (col.getAdjustedNetProductionOf(foodType) > 0) {
            // Try to produce something.
            wlps = workPlans;
            while (!produce.isEmpty()) {
                if ((wlp = findPlan(produce.get(0), workPlans)) != null) {
                    break; // Found a plan to try.
                }
                produce.remove(0); // Can not produce this goods type
            }
        }

        // See if a plan can be satisfied.
        Unit best;
        WorkLocation wl;
        GoodsType goodsType;
        for (;;) {
            if (wlp == null) { // Time to use a food plan.
                if (foodPlans.isEmpty()) {
                    lb.add("    Food plans exhausted\n");
                    done = true;
                    break;
                }
                wlps = foodPlans;
                wlp = wlps.get(0);
            }

            String err = null;
            goodsType = wlp.getGoodsType();
            wl = col.getCorresponding(wlp.getWorkLocation());
            best = null;
```

```
                lb.add("    ", LogBuilder.wide(2, col.getUnitCount()),
                       ": ", LogBuilder.wide(-15, goodsType.getSuffix()),
                       "@", LogBuilder.wide(25, locationDescription(wl)),
                       " => ");

                if (!wl.canBeWorked()) {
                    err = "can not be worked";
                } else if (wl.isFull()) {
                    err = "full";
                } else if ((best = ColonyPlan.getBestWorker(wl, goodsType,
                                                            workers)) == null)
{
                    err = "no worker found";
                }
                if (err != null) {
                    wlps.remove(wlp); // The plan can not be worked, dump it.
                    lb.add(err, "\n");
                    break;
                }

                // Found a suitable worker, place it.
                best.setLocation(wl);

                // Did the placement break the production bonus?
                if (col.getProductionBonus() < 0) {
                    best.setLocation(tile);
                    done = true;
                    lb.add("    broke production bonus\n");
                    break;
                }

                // Is the colony going to starve because of this placement?
                if (col.getAdjustedNetProductionOf(foodType) < 0) {
                    int net = col.getAdjustedNetProductionOf(foodType);
                    int count = col.getGoodsCount(foodType);
                    if (count / -net < PRODUCTION_TURNOVER_TURNS) {
                        // Too close for comfort.  Back out the
                        // placement and try a food plan, unless this
                        // was already a food plan.
                        best.setLocation(tile);
                        wlp = null;
                        if (goodsType.isFoodType()) {
                            lb.add("    starvation (", count, "/", net, ")\n");
                            done = true;
                            break;
                        }
                        lb.add("    would starve (", count, "/", net, ")\n");
                        continue;
                    }
                    // Otherwise tolerate the food stock running down.
                    // Rely on the warehouse-exhaustion code to fire
```

```
    // another rearrangement before units starve.
}

// Check if placing the worker will soon exhaust the
// raw material.  Do not reduce raw materials below
// what is needed for a building--- e.g. prevent
// musket production from hogging the tools.
GoodsType raw = goodsType.getInputType();
int rawNeeded = sum(buildGoods, ag -> ag.getType() == raw,
                    AbstractGoods::getAmount);
if (raw == null
    || col.getAdjustedNetProductionOf(raw) >= 0
    || (((col.getGoodsCount(raw) - rawNeeded)
            / -col.getAdjustedNetProductionOf(raw))
        >= PRODUCTION_TURNOVER_TURNS)) {
    // No raw material problems, the placement
    // succeeded.  Set the work type, move the
    // successful goods type to the end of the produce
    // list for later reuse, remove the worker from
    // the workers pool, but leave the successful plan
    // on its list.
    best.changeWorkType(goodsType);
    workers.remove(best);
    lb.add("    ", best.getId(), "(",
           best.getType().getSuffix(),")\n");
    if (!goodsType.isFoodType() && produce.remove(goodsType)) {
        produce.add(goodsType);
    }
    break;
}

// Yes, we need more of the raw material.  Pull the
// unit out again and see if we can make more.
best.setLocation(tile);

WorkLocationPlan rawWlp = findPlan(raw, workPlans);
if (rawWlp != null) {
    // OK, we have an alternate plan.  Put the raw
    // material at the start of the produce list and
    // loop trying to satisfy the alternate plan.
    if (produce.remove(raw)) produce.add(0, raw);
    wlp = rawWlp;
    lb.add("    retry with ", raw.getSuffix(), "\n");
    continue;
}

// No raw material available, so we have to give up on
// both the plan and the type of production.
// Hopefully the raw production is positive again and
// we will succeed next time.
wlps.remove(wlp);
```

```
                produce.remove(goodsType);
                lb.add("    needs more ", raw.getSuffix(), "\n");
                break;
            }
        }

        // Put the rest of the workers on the tile.
        for (Unit u : workers) {
            if (u.getLocation() != tile) u.setLocation(tile);
        }

        // Check for failure to assign any workers.  This happens when:
        // - there are no useful food plans
        //   - in which case look for a `harmless' place and add one worker
        // - food is low, and perhaps partly eaten by horses, and no
        //   unit can *improve* production by being added.
        //   - find a place to produce food that at least avoids
        //     starvation and add one worker.
        if (col.getUnitCount() == 0) {
            if (getFoodPlans().isEmpty()) {
             for (WorkLocation wl : col.getAvailableWorkLocationsList()) {
                    for (Unit u : new ArrayList<>(workers)) {
                        for (GoodsType type : libertyGoodsTypes) {
                            if (wl.canAdd(u)
                                && wl.getPotentialProduction(type,
                                    u.getType()) > 0) {
                                u.setLocation(wl);
                                u.changeWorkType(type);
                                workers.remove(u);
                                break locations;
                            }
                        }
                    }
                }
            } else {
              for (WorkLocationPlan w : getFoodPlans()) {
                    GoodsType goodsType = w.getGoodsType();
                                                        WorkLocation   wl   =
col.getCorresponding(w.getWorkLocation());
                    for (Unit u : new ArrayList<>(workers)) {
                        GoodsType oldWork = u.getWorkType();
                        u.setLocation(wl);
                        u.changeWorkType(goodsType);
                        if (col.getAdjustedNetProductionOf(foodType) >= 0) {
                            lb.add("    Subsist with ", u, "\n");
                            workers.remove(u);
                            break plans;
                        }
                        u.setLocation(tile);
                        u.changeWorkType(oldWork);
                    }
```

```
            }
        }
    }

    // The greedy algorithm works reasonably well, but will
    // misplace experts when they are more productive at the
    // immediately required task than a lesser unit, not knowing
    // that a requirement for their speciality will subsequently
    // follow.  Do a cleanup pass to sort these out.
    List<Unit> experts = new ArrayList<>();
    List<Unit> nonExperts = new ArrayList<>();
    for (Unit u : col.getUnitList()) {
        if (u.getType().getExpertProduction() != null) {
            if (u.getType().getExpertProduction() != u.getWorkType()) {
                experts.add(u);
            }
        } else {
            nonExperts.add(u);
        }
    }
    int expert = 0;
    Iterator<Unit> expertIterator = experts.iterator();
    while (expertIterator.hasNext()) {
        Unit u1 = expertIterator.next();
        Unit other = u1.trySwapExpert(experts);
        if (other != null) {
            lb.add("    Swapped ", u1.getId(), "(",
                u1.getType().getSuffix(), ") for ", other, "\n");
            expertIterator.remove();
        } else if ((other = u1.trySwapExpert(nonExperts)) != null) {
            lb.add("    Swapped ", u1.getId(), "(",
                u1.getType().getSuffix(), ") for ", other, "\n");
            expertIterator.remove();
        }
    }
    for (Unit u : new ArrayList<>(workers)) {
        GoodsType work = u.getType().getExpertProduction();
        if (work != null) {
            Unit other = u.trySwapExpert(col.getUnitList());
            if (other != null) {
                lb.add("    Swapped ", u.getId(), "(",
                    u.getType().getSuffix(), ") for ", other, "\n");
                workers.remove(u);
                workers.add(other);
            }
        }
    }

    // Rearm what remains as far as possible.
    for (Unit u : sort(workers, soldierComparator)) {
        if (u.getSkillLevel() > 0) continue;
```

```
            for (Role role : u.getSortedMilitaryRoles()) {
                if (fullEquipUnit(spec(), u, role, col)) {
                    lb.add("    ", u.getId(), "(", u.getType().getSuffix(),
                           ") -> ", u.getRoleSuffix(), "\n");
                    workers.remove(u);
                    break;
                }
            }
        }
        for (Unit u : transform(col.getUnits(), u -> !u.hasDefaultRole())) {
            logger.warning("assignWorkers bogus role for " + u);
            u.changeRole(spec().getDefaultRole(), 0);
        }

        // Log and return the scratch colony on success.
        // Otherwise abandon this rearrangement, disposing of the
        // scratch colony and returning null.
        for (Unit u : workers) {
            lb.add("    ", u.getId(), "(", u.getType().getSuffix(),
                   ") -> UNUSED\n");
        }
        if (col.getUnitCount() <= 0) col = null;
        return col;
    }
```

This code is located at *net.sf.freecol.server.ai.ColonyPlan.assignWorkers(List<Unit>, boolean, LogBuilder)*, and was selected based on size and complexity. As you can see, this method is extremely long and deals with multiple things that should not be addressed in the same method. For example, in the code we have a section where we try to assign outdoors experts to outdoors work, a section where we consider the defense of the colony, a huge section where we do a greedy assignment of the remaining workers (which in itself contains checks to verify if a plan can be satisfied and checks to verify if a placement breaks the production bonus, among many other verifications), a cleanup pass in the end of the assignments and code to rearm any remaining unit in the end.
A possible refactoring to remove this smell would be breaking up this method into several other smaller methods that each deal with a small issue. For example, all the verifications inside of the greedy assignment could be their own separate methods. There is also a part of the code inside the greedy assignment that deals with the exhaustion of raw materials. This code can be dealt with separately, inside their own methods.

Data Class

The following Data Class code smell was identified:

```
public static final class CombatResult {

        private final List<CombatEffectType> effects;
        private final int attackerHitpointsAfter;
        private final int defenderHitpointsAfter;
```

```
        public CombatResult(List<CombatEffectType> effects) {
            this.effects = Objects.requireNonNull(effects);
            this.attackerHitpointsAfter = -1;
            this.defenderHitpointsAfter = -1;
        }

                    public  CombatResult(List<CombatEffectType>  effects,  int
attackerHitpointsAfter, int defenderHitpointsAfter) {
            this.effects = Objects.requireNonNull(effects);
            this.attackerHitpointsAfter = attackerHitpointsAfter;
            this.defenderHitpointsAfter = defenderHitpointsAfter;
        }

        public List<CombatEffectType> getEffects() {
            return effects;
        }

        public boolean isAttackerHitpointsAffected() {
            return attackerHitpointsAfter >= 0;
        }

        public boolean isDefenderHitpointsAffected() {
            return defenderHitpointsAfter >= 0;
        }

        public int getAttackerHitpointsAfter() {
            return attackerHitpointsAfter;
        }

        public int getDefenderHitpointsAfter() {
            return defenderHitpointsAfter;
        }
    }
```

This class can be found in *net.sf.freecol.common.model.CombatModel.CombatResult*, and was selected because it only contains data and no functionality (only getter and setter methods). A possible refactoring to remove this smell would be to make this an abstract class and to add the combat logic code to the extensions of this class, one extension per combat model.

Switch Statement

The following Switch Statement code smell was identified:

```
(more code...)
for (CombatEffectType cr : crs) {
    boolean ok;
    switch (cr) {
    case AUTOEQUIP_UNIT:
        ok = isAttack && settlement != null;
        if (ok) {
```

```
                csAutoequipUnit(defenderUnit, settlement, cs);
            }
            break;
        case BURN_MISSIONS:
            ok = isAttack && result == CombatEffectType.WIN
                && natives != null
                && isEuropean() && defenderPlayer.isIndian();
            if (ok) {
                defenderTileDirty |= natives.hasMissionary(this);
                csBurnMissions(attackerUnit, natives, cs);
            }
            break;
        case CAPTURE_AUTOEQUIP:
            ok = isAttack && result == CombatEffectType.WIN
                && settlement != null;
            if (ok) {
                csCaptureAutoEquip(attackerUnit, defenderUnit, cs);
                attackerTileDirty = defenderTileDirty = true;
            }
            break;
        case CAPTURE_COLONY:
            ok = isAttack && result == CombatEffectType.WIN
                && colony != null
                 && (isEuropean() || isUndead()) && (defenderPlayer.isEuropean() ||
defenderPlayer.isUndead());
            if (ok) {
                csCaptureColony(attackerUnit, (ServerColony)colony,
                                random, cs);
                attackerTileDirty = defenderTileDirty = false;
                moveAttacker = true;
                defenderTension += Tension.TENSION_ADD_MAJOR;
            }
            break;
```

This code is a snippet of the method csCombat, located at *net.sf.freecol.server.model.ServerPlayer* (which can also serve as a long method code smell, but we will focus on the switch statement). This switch statement is a code smell because it will lead to issues when we want to make changes or add new combat effects. We can also observe code repetition in some cases. A possible refactoring would be to use polymorphism to take care of combat effects. For example, we could have an abstract combat effect class that in turn would be extended by each combat effect (and some effects could extend other effects), and the csCombat method would simply call a method of the abstract class. This would fix both the repeated code issues and the issues that arise due to the switch case.

André Branco 62482 (Aser28860d)

Long Method

(net.sf.freecol.server.ai.mission.Mission.travelToTarget(Location, CostDecider, LogBuilder))

```
protected MoveType travelToTarget(Location target, CostDecider costDecider,
                                  LogBuilder lb) {
    if (target == null) return MoveType.MOVE_ILLEGAL;
    final Tile targetTile = target.getTile();
    if (!(target instanceof Europe) && targetTile == null) {
        throw new RuntimeException("Target neither Europe nor Tile: "
            + target);
    }
    final Unit unit = getUnit();
    AIUnit aiCarrier = aiUnit.getTransport();
    final Map map = unit.getGame().getMap();
    PathNode path = null;
    boolean useTransport = false;
    target = Location.upLoc(target);
    // Consider where the unit is starting.
    if (unit.isAtSea()) {
        // Wait for carrier to arrive on the map or in Europe.
        lb.add(", at sea");
        return MoveType.MOVE_HIGH_SEAS;
    } else if (unit.isOnCarrier()) {
        // Transport mission will disembark the unit when it
        // arrives at the drop point.
        lb.add(", on carrier");
        return MoveType.MOVE_NO_ACCESS_EMBARK;
    } else if (unit.isAtLocation(target)) {
        // Arrived!
        return MoveType.MOVE;
    } else if (unit.isInEurope()) {
        // Leave, or require transport.
        if (!unit.getOwner().canMoveToEurope()) {
            lb.add(", impossible move from Europe");
            return MoveType.MOVE_ILLEGAL;
        }
        if (unit.getType().canMoveToHighSeas()) {
            unit.setDestination(target);
            if (AIMessage.askMoveTo(aiUnit, map)) {
                lb.add(", sailed for ", target);
                return MoveType.MOVE_HIGH_SEAS;
            } else {
                lb.add(", failed to sail for ", target);
                return MoveType.MOVE_ILLEGAL;
            }
        }
        useTransport = true;
    } else if (!unit.hasTile()) {
```

```
        // Fail!
        return MoveType.MOVE_ILLEGAL;
    } else {
        // On map.  Either find a path or decide to use transport.
        if (target instanceof Europe) {
            // Going to Europe.
            if (!unit.getOwner().canMoveToEurope()) {
                lb.add(", impossible move to Europe");
                return MoveType.MOVE_ILLEGAL;
            }
            if (!unit.getType().canMoveToHighSeas()
                || aiCarrier != null) {
                useTransport = true;
            } else {
                path = unit.findPath(unit.getLocation(), target,
                                     null, costDecider, null);
            }
        } else if (aiCarrier != null) {
            // Transport already allocated.
            useTransport = true;
        } else if (!unit.getType().canMoveToHighSeas()
            && !Map.isSameContiguity(target, unit.getLocation())) {
            // Transport necessary.
            useTransport = true;
        } else {
            // Should not need transport within the same contiguity.
            path = unit.findPath(unit.getLocation(), target,
                                 null, costDecider, null);
        }
    }
    if (useTransport) {
        if (aiCarrier != null) {
            // A carrier has been assigned.  Try to go to the
            // collection point.
            Location pick;
            TransportMission tm;
            boolean waiting = false;
            PathNode ownPath;
            int pathTurns, ownTurns;
            if ((tm = aiCarrier.getMission(TransportMission.class)) == null) {
                // Carrier has no transport mission?!?  Bogus.
                lb.add(", had bogus carrier ", aiCarrier.getUnit());
                logger.warning(unit + " has transport " + aiCarrier
                    + " without transport mission");
                aiUnit.dropTransport();
                aiCarrier = null;
            } else if ((pick = tm.getTransportTarget(aiUnit)) == null) {
                // No collection point for this unit?  Bogus.
                lb.add(", had bogus transport on ", aiCarrier.getUnit());
                logger.warning(unit + " has transport " + aiCarrier
                    + " with transport mission but null transport target\n"
```

```
            + tm.toFullString());
        aiUnit.dropTransport();
        aiCarrier = null;
    } else if (Map.isSameLocation(pick, unit.getLocation())) {
        // Waiting for the carrier at the collection point.
        waiting = true;
    } else if ((path = unit.findPath(unit.getLocation(), pick,
                null, costDecider, null)) == null) {
        // No path to the collection point.
        lbAt(lb);
        lb.add(", no path to meet ", aiCarrier.getUnit(),
               " at ", pick);
        path = unit.findPath(unit.getLocation(), target,
            null, costDecider, null);
        if (path == null) {
            // Unable to fall back to going direct.
            // Return failure in the hope that it is a
            // transient blockage.
            return MoveType.MOVE_NO_TILE;
        }
        // Fall back to going direct to the target.
        lb.add(", dropped carrier");
        aiUnit.dropTransport();
        aiCarrier = null;
        useTransport = false;
    } else if ((ownPath = unit.findPath(unit.getLocation(),
                target, null, costDecider, null)) == null
        || (ownTurns = ownPath.getTotalTurns())
        > (pathTurns = path.getTotalTurns())) {
        // Either there is no direct path to the target or
        // a path exists but takes longer than using the
        // carrier.  This confirms that it is not only
        // possible to travel to the collection point, it
        // is also the best plan.
        MoveType ret = followMapPath(path.next, lb);
        if (ret != MoveType.MOVE) return ret;
        waiting = true; // Arrived for collection.
    } else {
        // It is quicker to cancel the transport and go to
        // the target directly.
        lb.add(", dropping carrier", aiCarrier.getUnit(),
            " as it is faster (", ownTurns, "<", pathTurns,
            " without it");
        aiUnit.dropTransport();
        aiCarrier = null;
        path = ownPath;
        useTransport = false;
    }
    if (waiting) {
        // If waiting for the carrier, signal that this
        // unit can be reexamined if the carrier is still
```

```
                // moving.
                lbAt(lb);
                lb.add(", wait for ", aiCarrier.getUnit());
                return (aiCarrier.getUnit().getMovesLeft() > 0)
                    ? MoveType.MOVE_NO_ACCESS_EMBARK
                    : MoveType.MOVE_NO_MOVES;
            }
        }
        if (useTransport && aiCarrier == null) {
            // Still interested in transport but no carrier.
            lb.add(", needs transport to ", target);
            return MoveType.MOVE_NO_ACCESS_EMBARK;
        }
    }
    // Follow the path to the target.  If there is one.
    if (path == null) {
        lbAt(lb);
        lb.add(", no path to ", target);
        return MoveType.MOVE_NO_TILE;
    }
    if (path.next == null) {
        // This should not happen, the isAtLocation() test above
        // should have succeeded.
        throw new IllegalStateException("Trivial path found "
            + path.fullPathToString()
            + " from " + unit.getLocation() + " to target " + target
            + " result=" +  unit.isAtLocation(target));
    }
    return followMapPath(path.next, lb);
}
```

As mentioned above, this method is located in net.sf.freecol.server.ai.mission.Mission.travelToTarget(Location, CostDecider, LogBuilder). This method is a bit too extensive and more complex than it needs to be. The various if and else statements make it difficult to understand with values changing throughout the method and being carried until the end (with almost 200 lines can be confusing to keep track), or even full functionalities, inside the if statement, that deserved their own method.

As said, a possible solution for this smell would be separating into their own methods the if statements functionalities and conditions, leaving a clean space and reorganized if and else statements with each method and condition being a lot more clear and easy to read.

Speculative Generality

(3 methods from the class, scr.net.sf.freecol.common.networking.Message)

```
/**
* Sets an attribute in this message with a boolean value.
*
* @param key The attribute to set.
* @param value The value of the attribute.
```

```
*/
protected void setBooleanAttribute(String key, Boolean value) {
   if (value != null) setStringAttribute(key, Boolean.toString(value));
}
/**
* Sets an attribute in this message with an enum value.
*
* @param key The attribute to set.
* @param value The value of the attribute.
*/
protected void setEnumAttribute(String key, Enum<?> value) {
   if (value != null) setStringAttribute(key, downCase(value.toString()));
}
/**
* Sets an attribute in this message with an integer value.
*
* @param key The attribute to set.
* @param value The value of the attribute.
*/
protected void setIntegerAttribute(String key, int value) {
   setStringAttribute(key, Integer.toString(value));
}
```

This code smell usually happens when programmers are preparing themselves for possible future functionalities or needs and preemptively code methods before they have any use. In this case, these 3 ended up not having any influence in the final project, therefore the possible and logical solution for this smell is simply removing the methods.

Duplicate Code

(3 more methods from the class, scr.net.sf.freecol.common.networking.Message)

```
/**
* Sets an attribute in this message with a boolean value.
*
* @param key The attribute to set.
* @param value The value of the attribute.
*/
protected void setBooleanAttribute(String key, Boolean value) {
   if (value != null) setStringAttribute(key, Boolean.toString(value));
}
/**
* Sets an attribute in this message with an enum value.
*
* @param key The attribute to set.
* @param value The value of the attribute.
*/
protected void setEnumAttribute(String key, Enum<?> value) {
               if     (value     !=     null)     setStringAttribute(key,
downCase(value.toString()));
}
```

```
/**
* Sets an attribute in this message with an integer value.
*
* @param key The attribute to set.
* @param value The value of the attribute.
*/
protected void setIntegerAttribute(String key, int value) {
   setStringAttribute(key, Integer.toString(value));

}
```

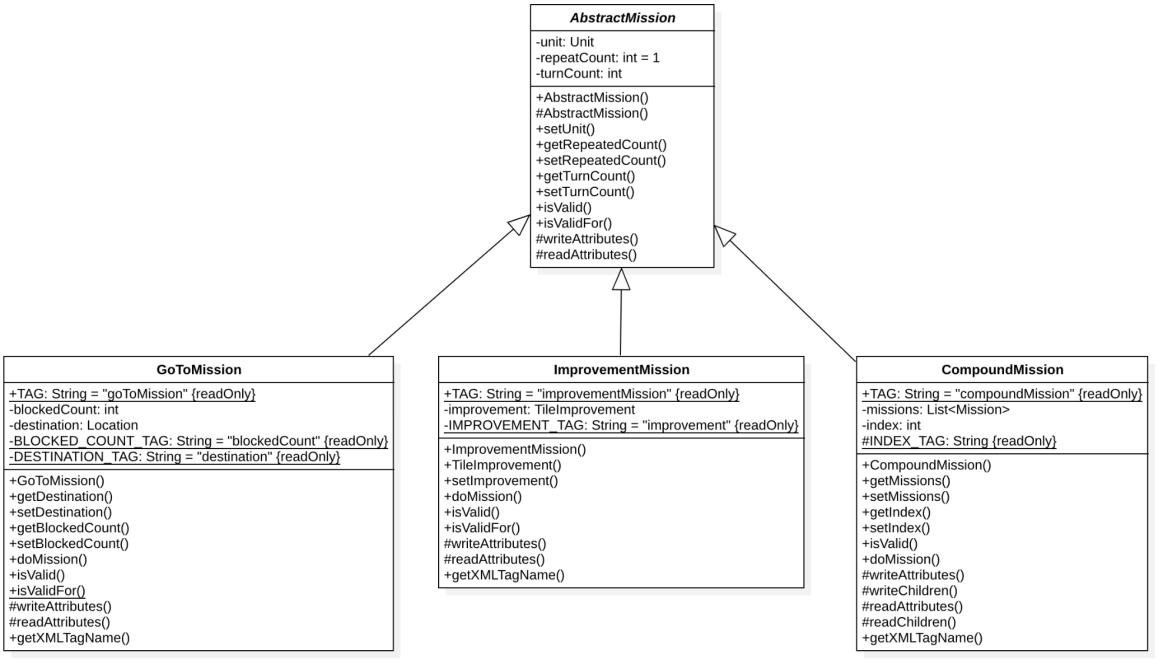
These methods follow a similar pattern for different types of attributes (Boolean, Enum, Integer), but the core functionality of converting these values to strings and setting attributes is duplicated with minor variations. This qualifies as Duplicate code.

A possible solution for this particular case would be a single method for example “setAttribute”, that can handle various data types, using an Object argument as the value, and executes the setStringAttribute accordingly.

Design Patterns

Beatriz Machado 62551 (BeatrizCaiola)

Template Pattern



In the class net.sf.freecol.common.model.mission.AbstractMission and in its subclasses net.sf.freecol.common.model.mission.GoToMission, net.sf.freecol.common.model.mission.ImprovementMission and net.sf.freecol.common.model.mission.CompoundMission we can identify an instance of the Design Pattern Template Pattern. The abstract class represents a general mission, providing methods prevalent for all missions upon which the subclasses build through their specific methods.

net.sf.freecol.common.model.mission.AbstractMission

```
/**
* The AbstractMission provides basic methods for building Missions.
*/
public abstract class AbstractMission extends FreeColGameObject implements
Mission {

/**
* Returns true if the Unit this mission was assigned to is
* neither null nor has been disposed, and the repeat count of the
* mission is greater than zero.
*
* @return a {@code boolean} value
*/
```

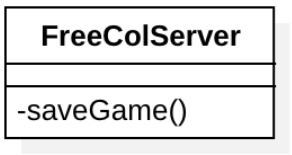
```
@Override
public boolean isValid() {
    return repeatCount > 0
        && unit != null && !unit.isDisposed();
}

/**
* {@inheritDoc}
*/
@Override
protected void writeAttributes(FreeColXMLWriter xw) throws XMLStreamException {
    super.writeAttributes(xw);

    xw.writeAttribute(UNIT_TAG, unit);

    xw.writeAttribute(TURN_COUNT_TAG, turnCount);

    xw.writeAttribute(REPEAT_COUNT_TAG, repeatCount);
}

/**
* {@inheritDoc}
*/
@Override
protected void readAttributes(FreeColXMLReader xr) throws XMLStreamException {
    super.readAttributes(xr);

    unit = xr.makeFreeColObject(getGame(), UNIT_TAG, Unit.class, true);

    turnCount = xr.getAttribute(TURN_COUNT_TAG, 0);

    repeatCount = xr.getAttribute(REPEAT_COUNT_TAG, 1);
}
```

net.sf.freecol.common.model.mission.GoToMission

```
/**
* The GoToMission causes a Unit to move towards its destination.
*/
public class GoToMission extends AbstractMission {

/**
* Returns true if the mission is still valid.
*
* @return a {@code boolean} value
*/
@Override
public boolean isValid() {
    // FIXME: check for disposed destinations
    return destination != null && destination.canAdd(getUnit())
        && super.isValid();
```

```
}
/**
* {@inheritDoc}
*/
@Override
protected void writeAttributes(FreeColXMLWriter xw) throws XMLStreamException {
    super.writeAttributes(xw);

    xw.writeAttribute(DESTINATION_TAG, destination);

    xw.writeAttribute(BLOCKED_COUNT_TAG, blockedCount);
}

/**
* {@inheritDoc}
*/
@Override
protected void readAttributes(FreeColXMLReader xr) throws XMLStreamException {
    super.readAttributes(xr);

    destination = xr.getLocationAttribute(getGame(), DESTINATION_TAG,
                                          false);

    blockedCount = xr.getAttribute(BLOCKED_COUNT_TAG, 0);
}
```

Memento Pattern



In the class net.sf.freecol.server.FreeColServer we find an example of the Design Pattern Memento Pattern. The private method saveGame() from the class FreeColServer proceeds to save a game, storing a state in which the necessary data is kept so that this instance of the game may be returned to if so wished.

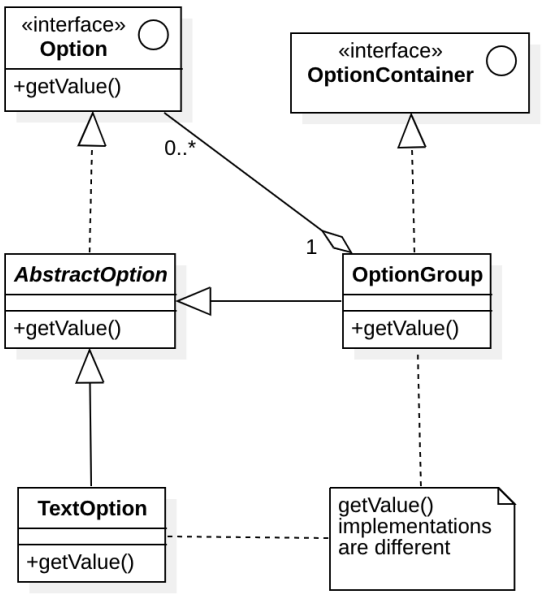
net.sf.freecol.server.FreeColServer

```
/**
* Saves a game.
*
* @param file The file where the data will be written.
* @param owner An optional name to use as the owner of the game.
* @param options Optional client options to save in the game.
* @param active An optional active {@code Unit}.
* @param image A thumbnail {@code Image} value to save in the game.
* @exception IOException If a problem was encountered while trying
```

```
*     to open, write or close the file.
*/
private void saveGame(File file, String owner, OptionGroup options,
                      Unit active, BufferedImage image) throws IOException {
   // Try to GC now before launching into the save, as a failure
   // here can lead to a corrupt saved game file (BR#3146).
   // Alas, gc is only advisory, but it is all we have got.
   garbageCollect();
   try (JarOutputStream fos = new JarOutputStream(Files
           .newOutputStream(file.toPath()))) {
       if (image != null) {
           fos.putNextEntry(new JarEntry(FreeColSavegameFile.THUMBNAIL_FILE));
           ImageIO.write(image, "png", fos);
           fos.closeEntry();
       }

       if (options != null) {
           fos.putNextEntry(new JarEntry(FreeColSavegameFile.CLIENT_OPTIONS));
           options.save(fos, null, true);
           fos.closeEntry();
       }

       Properties properties = new Properties();
       properties.setProperty("map.width",
           Integer.toString(this.serverGame.getMap().getWidth()));
       properties.setProperty("map.height",
           Integer.toString(this.serverGame.getMap().getHeight()));
       fos.putNextEntry(new JarEntry(FreeColSavegameFile.SAVEGAME_PROPERTIES));
       properties.store(fos, null);
       fos.closeEntry();

       // save the actual game data
       fos.putNextEntry(new JarEntry(FreeColSavegameFile.SAVEGAME_FILE));
       try (
           // throws IOException
           FreeColXMLWriter xw = new FreeColXMLWriter(fos,
               FreeColXMLWriter.WriteScope.toSave(), false)) {
           xw.writeStartDocument("UTF-8", "1.0");

           xw.writeComment(FreeCol.getConfiguration().toString());
           xw.writeCharacters("\n");

           xw.writeStartElement(SAVED_GAME_TAG);

           // Add the attributes:
           xw.writeAttribute(OWNER_TAG,
                             (owner != null) ? owner : FreeCol.getName());

           xw.writeAttribute(PUBLIC_SERVER_TAG, this.getPublicServer());

           xw.writeAttribute(SINGLE_PLAYER_TAG, this.singlePlayer);
```

```
            xw.writeAttribute(FreeColSavegameFile.VERSION_TAG,
                              SAVEGAME_VERSION);

            xw.writeAttribute(RANDOM_STATE_TAG,
                              getRandomState(this.random));

            xw.writeAttribute(DEBUG_TAG, FreeColDebugger.getDebugModes());

            if (active != null) {
                this.serverGame.setInitialActiveUnitId(active.getId());
            }

            this.serverGame.toXML(xw); // Add the game

            if (this.aiMain != null) { // Add the AIObjects
                this.aiMain.toXML(xw);
            }

            xw.writeEndElement();
            xw.writeEndDocument();
        }
        fos.closeEntry();
    } catch (XMLStreamException e) {
        throw new IOException("Failed to save (XML): " + file.getName(), e);
    }
}
```

Composite Pattern



In the interface net.sf.freecol.comon.option.Option and class net.sf.freecol.common.option.OptionGroup we can identify an instance of the Design Pattern

Composite Pattern. The class OptionGroup extends the abstract class AbstractOption, which in turn implements the interface Option, therefore, the class OptionGroup also implements this interface, granting it the method getValue(). This method will produce different results in the circumstance where we're only dealing with a single Option and when we're handling an OptionGroup, despite both implementing the same interface. From the perspective of the program, there isn't a way of discerning whether we're handling an Option or an OptionGroup, consequently, we can label this as a Composite Pattern.

net.sf.freecol.comon.option.Option

```
/**
* Gets the value of this option.
*
* @return The value of this {@code Option}.
*/
public T getValue();
```

net.sf.freecol.common.option.OptionGroup

```
/**
* {@inheritDoc}
*/
@Override
public OptionGroup getValue() {
   return this;
}
```

Rodrigo Monteiro Suzana 63069 (suzana2314)

Template Pattern

This pattern is located in the following package: *package net.sf.freecol.common.model*

```
public abstract class NationType extends FreeColSpecObjectType
public class IndianNationType extends NationType
public class EuropeanNationType extends NationType
```

The rationale for identifying this as a template method pattern instantiation lies in the clear structure of having an abstract class (NationType) providing the template or skeleton for behavior and concrete subclasses (IndianNationType, EuropeanNationType) implementing specific details.

Factory pattern

This pattern is located in: *package net.sf.freecol.common.resources.ResourceFactory*

This pattern allows for the creation of different types of objects (resources in this case) without explicitly specifying their classes. The method createResource acts as a factory by creating and returning the appropriate subclass of the Resource class based on the type of URI provided. Bellow is the referenced method:

```
public Resource createResource(String key, String cachingKey, URI uri) {
        final Resource r = resources.get(uri);
        if (r != null) {
            return r;
        }
        final String pathPart;
        if (uri.getPath() != null) {
            pathPart = uri.getPath();
        } else if (uri.toString().indexOf("!/") >= 0) {
              pathPart = uri.toString().substring(uri.toString().indexOf("!/") +
2);
        } else {
            pathPart = null;
        }
        try {
            final Resource resource;
            if ("urn".equals(uri.getScheme())) {
                                                                        if
(uri.getSchemeSpecificPart().startsWith(ColorResource.SCHEME)) {
                    resource = new ColorResource(cachingKey, uri);
                                                              }    else    if
(uri.getSchemeSpecificPart().startsWith(FontResource.SCHEME)) {
                    resource = new FontResource(cachingKey, uri);
                } else {
                                logger.log(Level.WARNING, "Unknown urn part: " +
uri.getSchemeSpecificPart());
                    return null;
                }
```

```
                } else if (pathPart.endsWith("\"") && pathPart.lastIndexOf('"',
pathPart.length()-1) >= 0) {
                    resource = new StringResource(cachingKey, uri);
                } else if (pathPart.endsWith(".faf")) {
                    resource = new FAFileResource(cachingKey, uri);
                } else if (pathPart.endsWith(".sza")) {
                    resource = new SZAResource(cachingKey, uri);
                } else if (pathPart.endsWith(".ttf")) {
                    resource = new FontResource(cachingKey, uri);
                } else if (pathPart.endsWith(".wav")) {
                    resource = new AudioResource(cachingKey, uri);
                } else if (pathPart.endsWith(".ogg")) {
                    if (pathPart.endsWith(".video.ogg")) {
                        resource = new VideoResource(cachingKey, uri);
                    } else {
                        resource = new AudioResource(cachingKey, uri);
                    }
                } else if (key.startsWith("sound.")) {
                    resource = new AudioResource(cachingKey, uri);
                } else {
                    resource = new ImageResource(cachingKey, uri);
                }
                resources.put(uri, resource);
                return resource;
            } catch (IOException ioe) {
                logger.log(Level.WARNING, "Failed to create " + uri, ioe);
                return null;
            }
        }
```

Observer Pattern

This pattern is located in the following package: *net.sf.freecol.client.gui.panel*
Respectively in the following classes: TransactionListener, EuropePanel (TransactionLog is inside this class in line 509) and Market.
The TransactionListener interface serves as a contract, providing a way for the TransactionLog class to be notified when changes occur in the class Market. When relevant changes take place within the Market class, it triggers notifications to all subscribed TransactionListeners. This mechanism allows the observers to react to changes in the subject's state.

Market

```
    /**
     * Adds a transaction listener for notification of any transaction
     *
     * @param listener The {@code TransactionListener} to add.
     */
    public void addTransactionListener(TransactionListener listener) {
```

```
        transactionListeners.add(listener);
    }
    /**
     * Removes a transaction listener
     *
     * @param listener The {@code TransactionListener} to remove.
     */
    public void removeTransactionListener(TransactionListener listener) {
        transactionListeners.remove(listener);
    }
    /**
     * Gets the listeners added to this market.
     *
     * @return An array of all the {@code TransactionListener}s
     *     added, or an empty array if none found.
     */
    public TransactionListener[] getTransactionListener() {
        return transactionListeners.toArray(new TransactionListener[0]);
    }
```

TransactionListener

```
public interface TransactionListener {
    /**
     * Logs a purchase
     *
     * @param goodsType The type of goods which have been purchased
     * @param amount The amount of goods which have been purchased
     * @param price The unit price of the goods
     */
    public void logPurchase(GoodsType goodsType, int amount, int price);
    /**
     * Logs a sale
     *
     * @param goodsType The type of goods which have been sold
     * @param amount The amount of goods which have been sold
     * @param price The unit price of the goods
     * @param tax The tax which has been applied
     */
    public void logSale(GoodsType goodsType, int amount, int price, int tax);
}
```

TransactionLog

```
 /**
         * Initializes this TransactionLog.
         */
        public void initialize() {
            getMyPlayer().getMarket().addTransactionListener(this);
            setText("");
        }
        /**
         * Cleans up this TransactionLog.
```

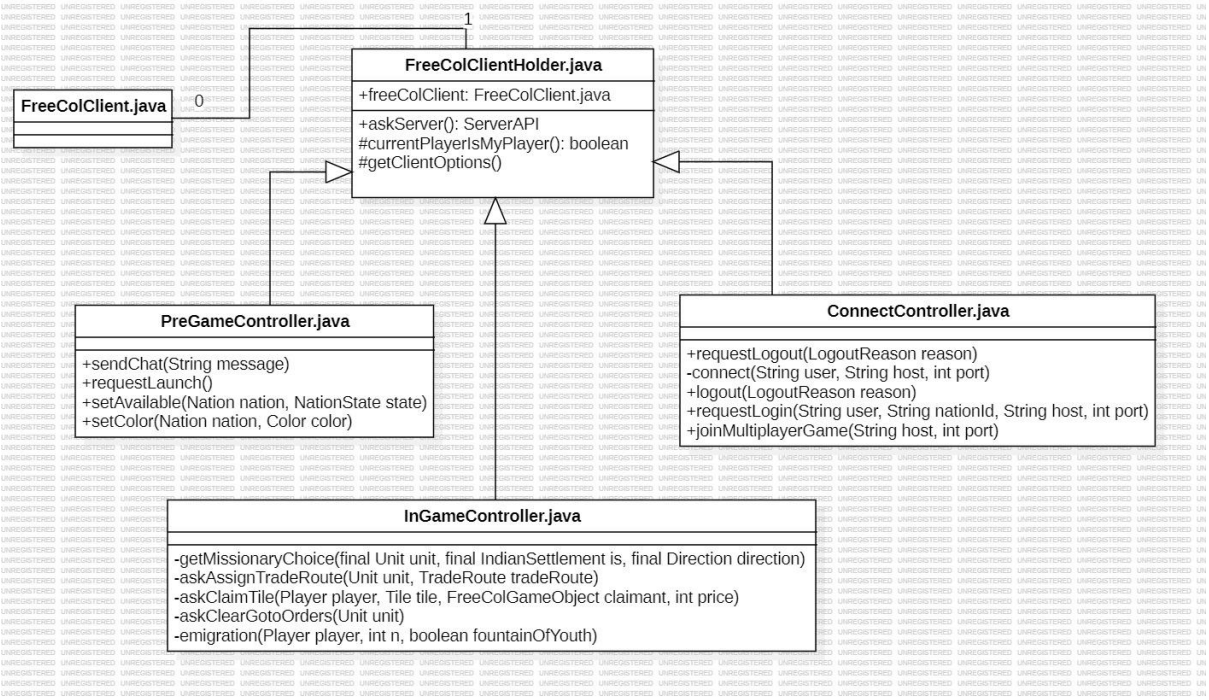
```
         */
        public void cleanup() {
            getMyPlayer().getMarket().removeTransactionListener(this);
        }
// Implement TransactionListener
        @Override
        public void logPurchase(GoodsType goodsType, int amount, int price) {
            int total = amount * price;
                                                    StringTemplate    t1    =
StringTemplate.template("europePanel.transaction.purchase")
                .addNamed("%goods%", goodsType)
                .addAmount("%amount%", amount)
                .addAmount("%gold%", price);
                                                    StringTemplate    t2    =
StringTemplate.template("europePanel.transaction.price")
                .addAmount("%gold%", total);
            add(Messages.message(t1) + "\n" + Messages.message(t2));
        }
        @Override
        public void logSale(GoodsType goodsType, int amount,
                            int price, int tax) {
            int totalBeforeTax = amount * price;
            int totalTax = totalBeforeTax * tax / 100;
            int totalAfterTax = totalBeforeTax - totalTax;
                                                    StringTemplate    t1    =
StringTemplate.template("europePanel.transaction.sale")
                .addNamed("%goods%", goodsType)
                .addAmount("%amount%", amount)
                .addAmount("%gold%", price);
                                                    StringTemplate    t2    =
StringTemplate.template("europePanel.transaction.price")
                .addAmount("%gold%", totalBeforeTax);
                                                    StringTemplate    t3    =
StringTemplate.template("europePanel.transaction.tax")
                .addAmount("%tax%", tax)
                .addAmount("%gold%", totalTax);
                                                    StringTemplate    t4    =
StringTemplate.template("europePanel.transaction.net")
                .addAmount("%gold%", totalAfterTax);
            add(Messages.message(t1) + "\n" + Messages.message(t2)
                + "\n" + Messages.message(t3) + "\n" + Messages.message(t4));
        }
    }
```

Rodrigo Fernandes 63191 (LemosFTW)

Proxy Pattern

FreeColClientHolder.java

FreecolClientHolder é um Proxy pois simplifica a complexidade da classe FreecolClient usando outra classe para manipular os métodos.



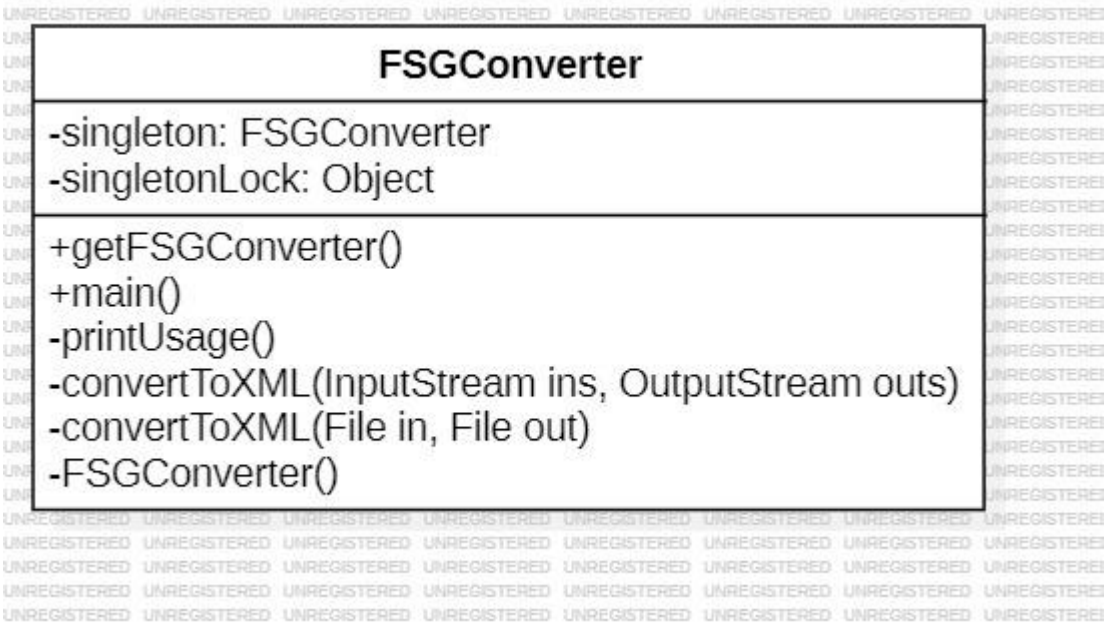

```
public class FreeColClientHolder {

    /** The main client object. */
    private final FreeColClient freeColClient;
    /**
     * Simple constructor.
     *
     * @param freeColClient The {@code FreeColClient} to hold.
     */
    protected FreeColClientHolder(FreeColClient freeColClient) {
        this.freeColClient = freeColClient;
    }
    /**
     * Meaningfully named access to the server API.
     *
     * @return The {@code ServerAPI}.
     */
    public ServerAPI askServer() {
        return this.freeColClient.askServer();
    }
    /**
```

```
     * Check if the current player is the client player.
     *
     * @return True if the client player is current.
     */
    protected boolean currentPlayerIsMyPlayer() {
        return this.freeColClient.currentPlayerIsMyPlayer();
    }
    /**
     * Get the client options.
     *
     * @return The {@code ClientOptions} held by the client.
     */
    protected ClientOptions getClientOptions() {
        return this.freeColClient.getClientOptions();
    }
    /**
     * Get the connect controller.
     *
     * @return The {@code ConnectController} held by the client.
     */
    protected ConnectController getConnectController() {
        return this.freeColClient.getConnectController();
    }
    /**
     * Get the main client object.
     *
     * @return The {@code FreeColClient} held by this object.
     */
    protected FreeColClient getFreeColClient() {
        return this.freeColClient;
    }
    /**
     * Get the server.
     *
     * @return The {@code FreeColServer} held by the client.
     */
    protected FreeColServer getFreeColServer() {
        return this.freeColClient.getFreeColServer();
    }
    /**
     * Get the game.
     *
     * @return The {@code Game} held by the client.
     */
    protected Game getGame() {
        return this.freeColClient.getGame();
    }
```

Singleton Pattern

FSGConverter.java

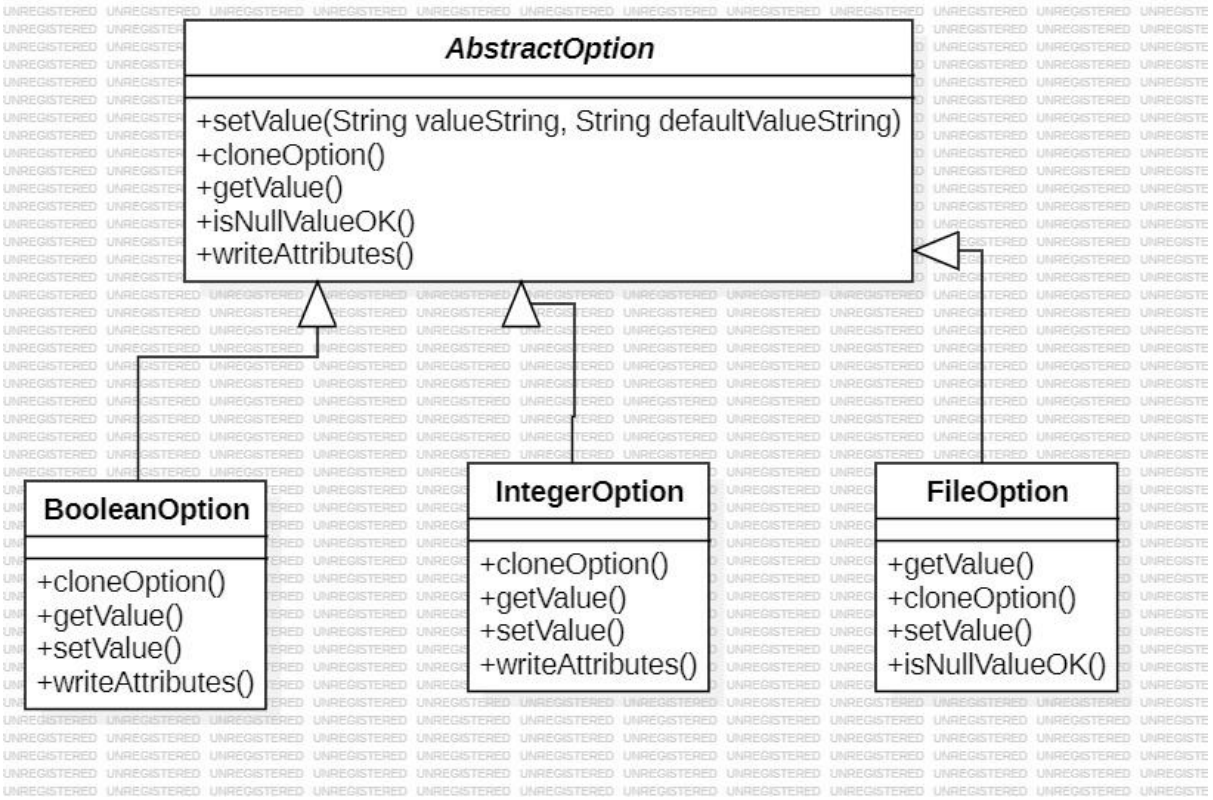
FSGConverter é um singleton pattern pois tem seu construtor privado e um método, que só permite a criação de apenas uma instância do objeto dessa classe.




```java
public class FSGConverter {
    /**
     * A singleton object of this class.
     * @see #getFSGConverter()
     */
    private static FSGConverter singleton;
    private static Object singletonLock = new Object();
    /**
     * Creates an instance of {@code FSGConverter}
     */
    private FSGConverter() {
        // Nothing to initialize;
    }
    /**
     * Gets an object for converting FreeCol Savegames.
     * @return The singleton object.
     */
    public static FSGConverter getFSGConverter() {
        // Using lazy initialization:
        synchronized (singletonLock) {
            if (singleton == null) {
                singleton = new FSGConverter();
            }
            return singleton;
        }
    }
```

Template Method Pattern

AbstractOption.java

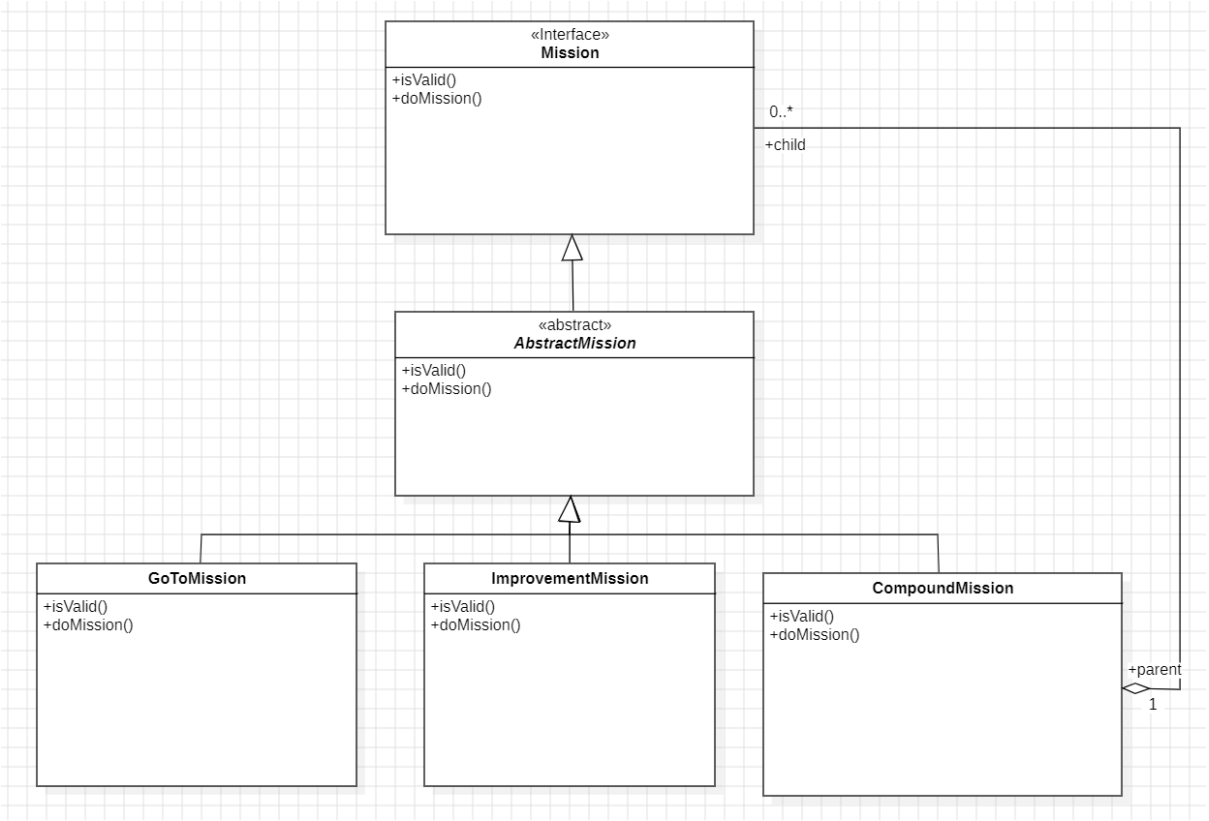



```
public abstract class AbstractOption<T> extends FreeColSpecObject
    implements Option<T> {

                    private      static      final      Logger      logger      =
Logger.getLogger(AbstractOption.class.getName());
    private static final String ENABLED_BY_TAG = "enabledBy";
    /** The option group prefix. */
    private String optionGroupId = "";
    /**
     * Determine if the option has been defined.  When defined an
     * option won't change when a default value is read from an XML file.
     */
    protected boolean isDefined = false;
    private String enabledBy = null;
public boolean isNullValueOK() {
    return false;
}
. . .

// Interface Option
/**
```

```
* {@inheritDoc}
*/
@Override
public abstract AbstractOption<T> cloneOption()
   throws CloneNotSupportedException;
/**
* {@inheritDoc}
*/
public String getGroup() {
   return this.optionGroupId;
}
/**
* {@inheritDoc}
*/
public void setGroup(String group) {
   this.optionGroupId = (group == null) ? "" : group;
}
/**
* {@inheritDoc}
*/
@Override
public abstract T getValue();
/**
* {@inheritDoc}
*/
@Override
public abstract void setValue(T value);
/**
* {@inheritDoc}
*/
@Override
public String getEnabledBy() {
   return enabledBy;
}
```

João Pedro Silveira 62654 (Joao-Pedro-Silveira)

Composite Pattern



Localização: src/net/sf/freecol/common/model/mission
This class diagram represents a composite pattern because the class CompoundMisson has a List of Mission, where it stores other instances of the type Mission and in the method isValid() all the objects in the list are checked for validity. The method doMission() works the same way.

src/net/sf/freecol/common/model/mission/Mission.java

```java
4 implementations   Michael Vehrs +2
public interface Mission {

    10 usages   Michael Vehrs
    public static enum MissionState {...};


    /** Attempts to carry out the mission and returns an appropriate ...*/
    3 implementations   Michael Vehrs
    public MissionState doMission();


    /**
     * Returns true if the mission is still valid. This might not be
     * the case if its target or destination have been destroyed, or
     * if the Unit this mission was assigned to was destroyed or
     * changed owner, for example.
     *
     * @return a {@code boolean} value
     */
    4 implementations   Michael Vehrs
    public boolean isValid();


    /** Return the Unit this mission was assigned to. ...*/
    1 implementation   Michael Vehrs
    public Unit getUnit();

    /** This method writes an XML-representation of this object to ...*/
    1 implementation   Mike Pope
    public void toXML(FreeColXMLWriter xw) throws XMLStreamException;
}
```

src/net/sf/freecol/common/model/mission/AbstractMission.java

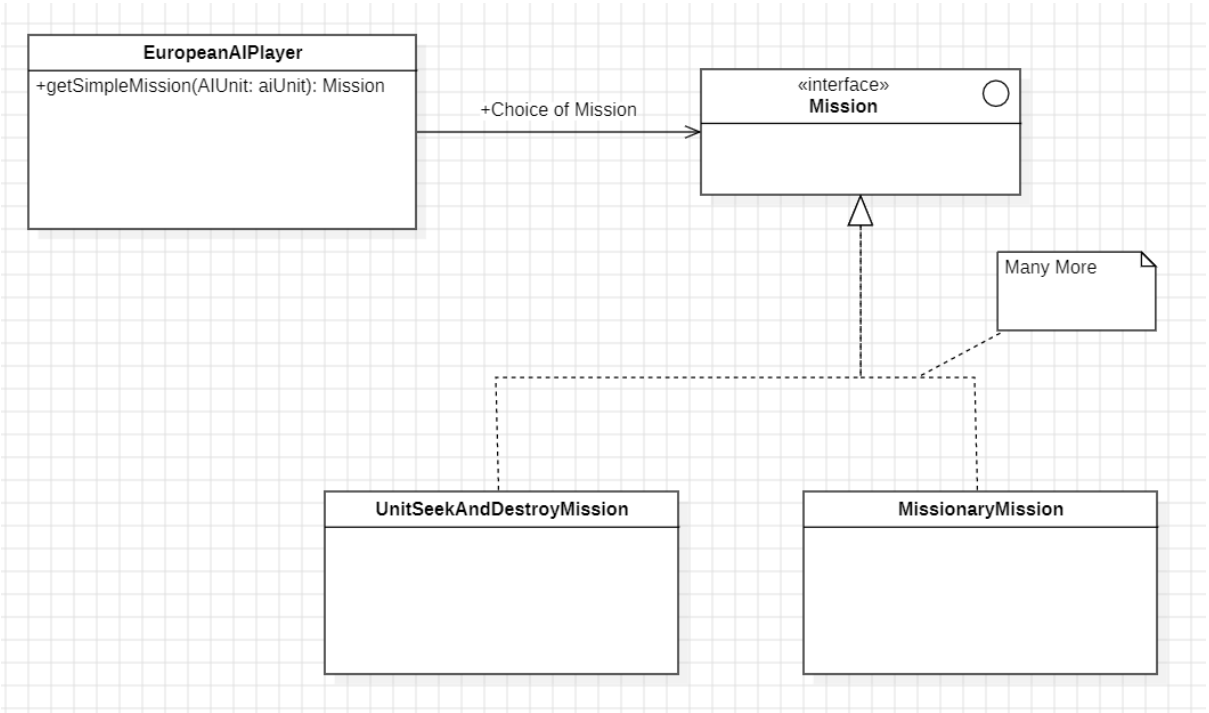
```java
/**
 * Returns true if the Unit this mission was assigned to is
 * neither null nor has been disposed, and the repeat count of the
 * mission is greater than zero.
 *
 * @return a {@code boolean} value
 */
3 overrides  Mike Pope +1
@Override
public boolean isValid() {
    return repeatCount > 0
        && unit != null && !unit.isDisposed();
}
```

src/net/sf/freecol/common/model/mission/CompoundMission.java

```java
/**
 * The CompoundMission provides a wrapper for several more basic
 * Missions that will be carried out in order.
 */
2 usages  Michael Vehrs +4
public class CompoundMission extends AbstractMission {

    public static final String TAG = "compoundMission";

    /** The individual missions this CompoundMission wraps. */
    9 usages
    private List<Mission> missions;

    /** The index of the current mission. */
    8 usages
    private int index;
```

```java
/**
 * {@inheritDoc}
 */
Michael Vehrs +1
@Override
public MissionState doMission() {
    while (true) {
        MissionState state = missions.get(index).doMission();
        if (state == MissionState.COMPLETED) {
            index++;
            if (index == missions.size()) {
                setRepeatCount(getRepeatCount() - 1);
                if (getRepeatCount() > 0) {
                    index = 0;
                } else {
                    return MissionState.COMPLETED;
                }
            }
            if (getUnit().getMovesLeft() > 0) {
                continue;
            }
        }
        return state;
    }
}
```

Strategy Design Pattern



Localização: src/net/sf/freecol/server/ai/EuropeanAIPlayer.java
This relation is a Strategy Pattern because it has a class called getSimpleMission where, considering several conditions in game and of the unit itself, a unit is assigned a mission to perform which can be of many different types like UnitSeekAndDestroyMission or MissionaryMission. This will then change the behavior of the unit in game.

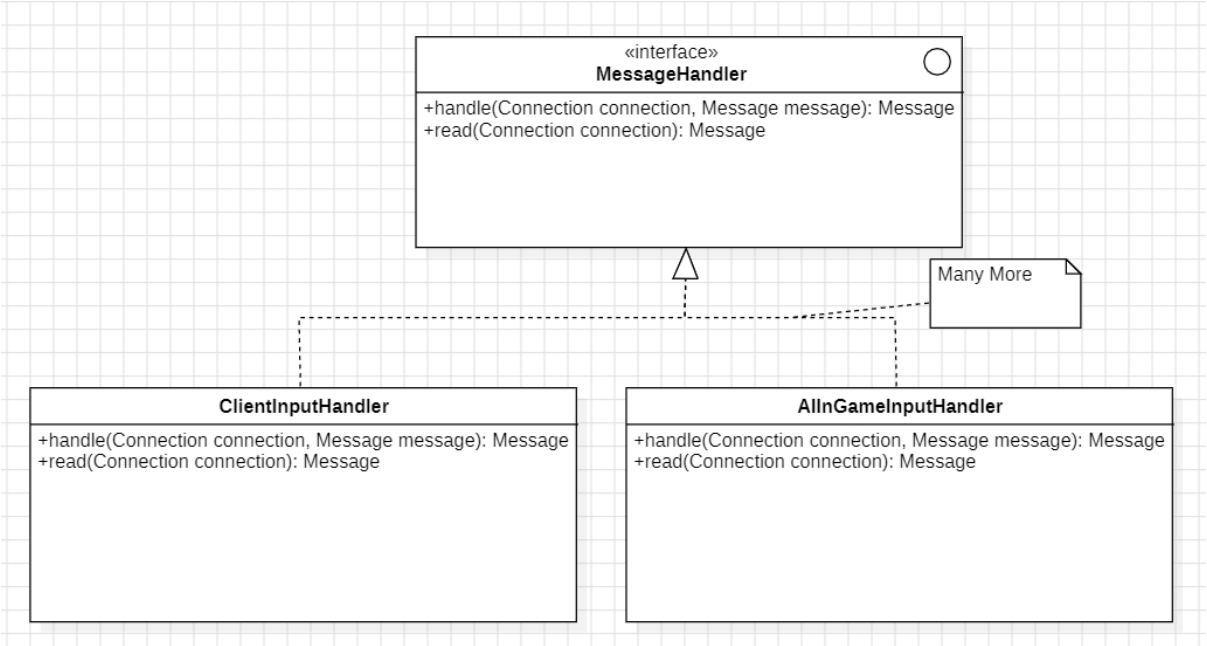
src/net/sf/freecol/server/ai/EuropeanAIPlayer.java

```
/**
 * Choose a mission for an AIUnit.
 *
 * @param aiUnit The {@code AIUnit} to choose for.
 * @return A suitable {@code Mission}, or null if none found.
 */
 private Mission getSimpleMission(AIUnit aiUnit) {
    final Unit unit = aiUnit.getUnit();
    Mission m, ret;
    final Mission old = ((m = aiUnit.getMission()) != null && m.isValid())
        ? m : null;
```

src/net/sf/freecol/server/ai/EuropeanAIPlayer.java

```java
/**
 * Gets a new MissionaryMission for a unit.
 *
 * @param aiUnit The {@code AIUnit} to check.
 * @return A new mission, or null if impossible.
 */
2 usages  Mike Pope +1
public Mission getMissionaryMission(AIUnit aiUnit) {
    if (MissionaryMission.prepare(aiUnit) != null) return null;
    Location loc = MissionaryMission.findMissionTarget(aiUnit,
        missionaryRange,  deferOK: true);
    if (loc == null) {
        aiUnit.equipForRole(getSpecification().getDefaultRole());
        return null;
    }
    return new MissionaryMission(getAIMain(), aiUnit, loc);
}
```

Template Pattern



Location: src/net/sf/freecol/common/networking

This Relation is a template pattern because all the classes that implement the Interface MessageHandler perform different actions when called to do the same methods.

src/net/sf/freecol/common/networking/MessageHandler.java

```java
public interface MessageHandler {

    /**
     * Handle an incoming message.
     *
     * @param connection The {@code Connection} the message arrived on.
     * @param message The {@code Message} to handle.
     * @return A reply message, if any.
     * @exception FreeColException if the message is malformed.
     */
    6 implementations  Mike Pope
    public Message handle(Connection connection, Message message)
        throws FreeColException;

    /**
     * Read an incoming Message.
     *
     * @param connection The {@code Connection} to read from.
     * @return The {@code Message} found, or null if none.
     * @exception FreeColException if the message can not be instantiated.
     * @exception XMLStreamException if there is a problem reading the
     *     message.
     */
    6 implementations  Mike Pope
    public Message read(Connection connection)
        throws FreeColException, XMLStreamException;
}
```

src/net/sf/freecol/client/control/ClientInputHandler.java

```java
public final class ClientInputHandler extends FreeColClientHolder
    implements MessageHandler {

    no usages
    private static final Logger logger = Logger.getLogger(ClientInputHandler.class.getName());

    /** The constructor to use. ...*/
    1 usage   Mike Pope
    public ClientInputHandler(FreeColClient freeColClient) { super(freeColClient); }


    // Implement MessageHandler

    /** {@inheritDoc} */
    Mike Pope
    public Message handle(/unused/ Connection connection,
                          Message message) throws FreeColException {
        message.clientHandler(getFreeColClient());
        return null;
    }

    /**
     * {@inheritDoc}
     */
    Mike Pope
    public Message read(Connection connection)
        throws FreeColException, XMLStreamException {
        return Message.read(getGame(), connection.getFreeColXMLReader());
    }
}
```

src/net/sf/freecol/server/ai/AIInGameInputHandler.java

```
    /**
     * {@inheritDoc}
     */
    Mike Pope
    public Message handle(Connection connection, Message message)
        throws FreeColException {
        message.aiHandler(getFreeColServer(), getMyAIPlayer());
        return null;
    }

    /**
     * {@inheritDoc}
     */
    Mike Pope
    public Message read(Connection connection)
        throws FreeColException, XMLStreamException {
        return Message.read(getGame(), connection.getFreeColXMLReader());
    }
}
```

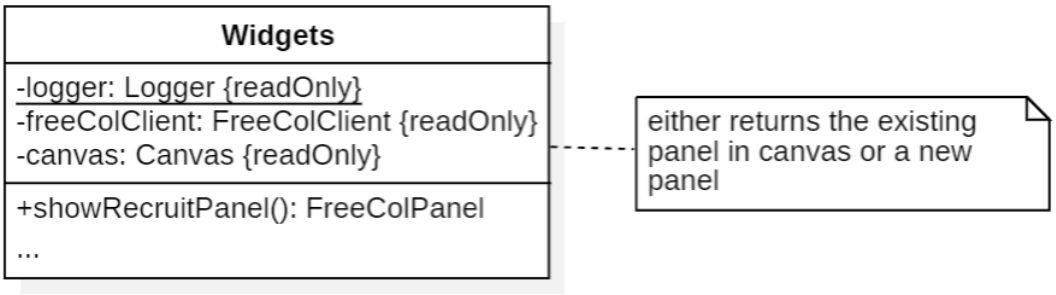
Tiago Sousa 63324 (tiago-cos)

Singleton Pattern

```
public FreeColPanel showPurchasePanel() {
     PurchasePanel panel
         = this.canvas.getExistingFreeColPanel(PurchasePanel.class);
     if (panel == null) {
         panel = new PurchasePanel(this.freeColClient);
         panel.update();
         this.canvas.showFreeColPanel(panel, PopupPosition.CENTERED, false);
     }
     return panel;
 }
```

This GoF design pattern is found implemented in the method net.sf.freecol.client.gui.Widgets.showPurchasePanel.
As we can see, this method restricts the initialization of the PurchasePanel class, ensuring that only one instance of the class is created. It starts by searching for an instance in the canvas, and only creates the object if it isn't found.

The Class diagram for this pattern is as follows:



Template Pattern

```
public abstract class TradeItem extends FreeColGameObject {

 ...

    /**
     * Is this trade item valid?  That is, is the request well formed.
     *
     * @return True if the item is valid.
     */
    public abstract boolean isValid();

    /**
     * Is this trade item unique?
     * This is true for the StanceTradeItem and the GoldTradeItem,
```

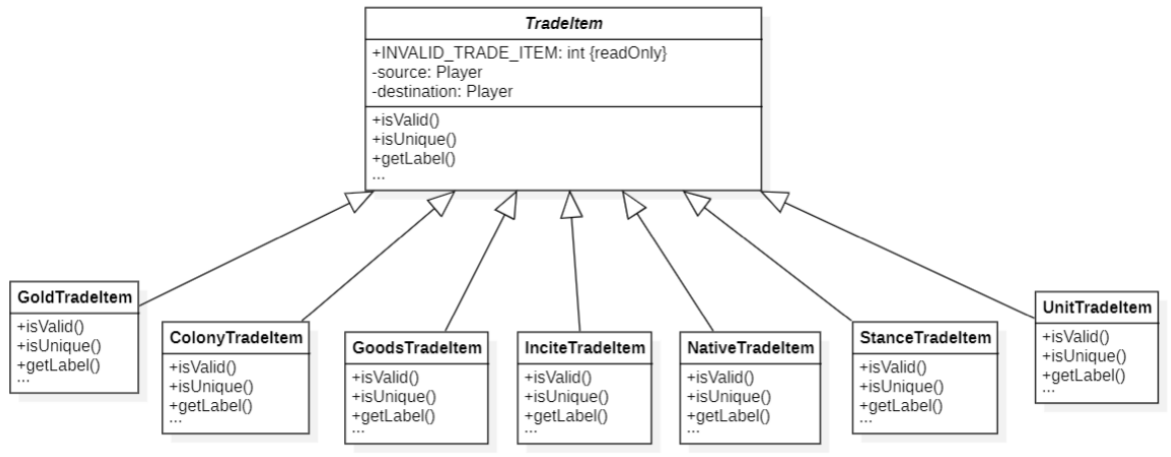
```
     * and false for all others.
     *
     * @return True if the item is unique.
     */
    public abstract boolean isUnique();

    /**
     * Get a label for this item.
     *
     * @return A {@code StringTemplate} describing this item.
     */
    public abstract StringTemplate getLabel();

    /**
     * Get the colony to trade.
     *
     * @param game A {@code Game} to look for the colony in.
     * @return The {@code Colony} to trade.
     */
    public Colony getColony(Game game) { return null; }

    ...

    /**
     * Get the gold to trade.
     *
     * @return The gold to trade.
     */
    public int getGold() { return 0; }

...

}

public class ColonyTradeItem extends TradeItem {

...

    /**
     * {@inheritDoc}
     */
    @Override
    public boolean isUnique() {
        return false;
    }

    /**
     * {@inheritDoc}
     */
    @Override
```

```
    public StringTemplate getLabel() {
                                                                    return
StringTemplate.template(Messages.descriptionKey("model.tradeItem.colony"))
            .addName("%colony%", colonyName);
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public Colony getColony(Game game) {
        return game.getFreeColGameObject(colonyId, Colony.class);
    }

...

}
```

This GoF design pattern is found in the class net.sf.freecol.common.model.TradeItem and its subclasses.

This class acts as a general trade item class, where some of the more generic trade item methods are implemented and some of the specific methods are left to be implemented in the subclasses.

The Class diagram for this pattern is as follows:



State Pattern

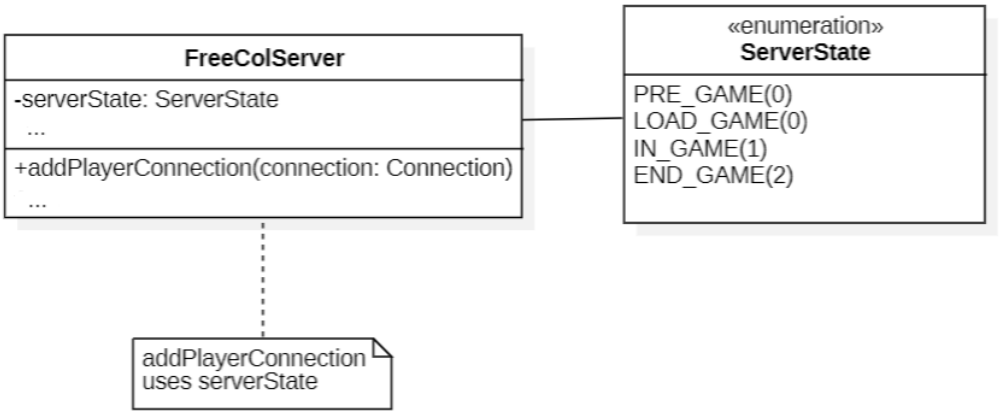
```
public final class FreeColServer {
```

```
    ...

    /** The current state of the game. */
    private ServerState serverState = ServerState.PRE_GAME;

    ...

   /**
    * Add player connection.  That is, a user connection is now
    * transitioning to a player connection.
    *
    * @param connection The new {@code Connection}.
    */
   public void addPlayerConnection(Connection connection) {
       switch (this.serverState) {
       case PRE_GAME: case LOAD_GAME: case IN_GAME:
           connection.setMessageHandler(this.inputHandler);
           break;
       case END_GAME: default:
           return;
       }
       getServer().addConnection(connection);
       updateMetaServer();
    }

...

}
```

This GoF design pattern is found in the class net.sf.freecol.server.FreeColServer.

This class has a private attribute called serverState which stores an enumeration indicating the state of the server. We can see that it has some methods that change their behavior according to the state of the class.

The Class diagram for this pattern is as follows:


FreeColServer
-serverState: ServerState
...
+addPlayerConnection(connection: Connection)
...
«enumeration»
ServerState
PRE_GAME(0)
LOAD_GAME(0)
IN_GAME(1)
END_GAME(2)
addPlayerConnection
uses serverState

André Branco 62482 (Aser28860d)

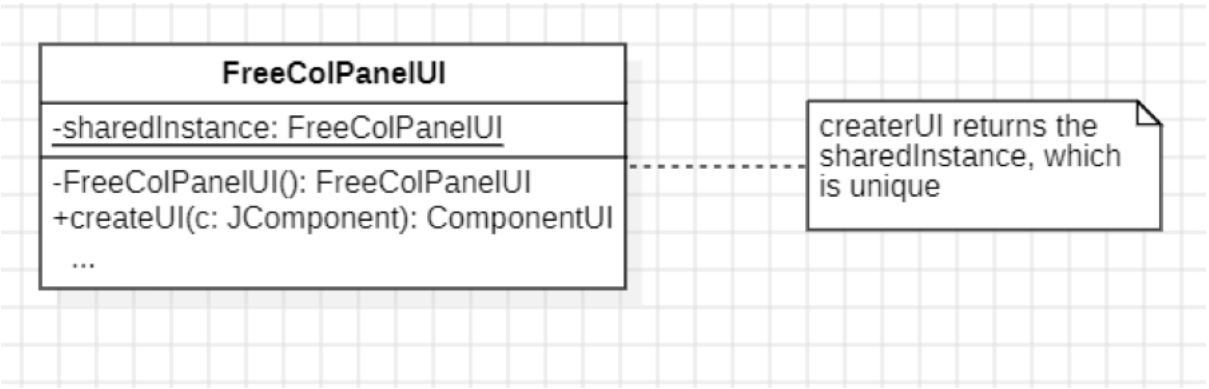
Singleton Pattern

```
public class FreeColPanelUI extends BasicPanelUI {

    1 usage
    private static final FreeColPanelUI sharedInstance = new FreeColPanelUI();


    1 usage   Mike Pope
    private FreeColPanelUI() {}

    Mike Pope +1
    public static ComponentUI createUI(/unused/ JComponent c) { return sharedInstance; }

    Stian Grenborgen +2
    @Override
    public void paint(java.awt.Graphics g, javax.swing.JComponent c) {
        if (c.isOpaque()) {
            ImageUtils.drawTiledImage(ImageLibrary.getPanelBackground(c.getClass()),
                                      g, c,  insets: null);
        }
    }
}
```

This design pattern is implemented on the class, net.sf.freecol.client.gui.plaf.FreeColPanelUI.java.

The class has a private constructor, “private FreeColPanelUI() {}”, which means that instances of this class can only be created from within the class itself. Then also there is a private static instance variable named “sharedInstance” that holds the single instance of the FreeColPanelUI class. It is final, so it cannot be reassigned once it's set.

The Class Diagram of this pattern is:



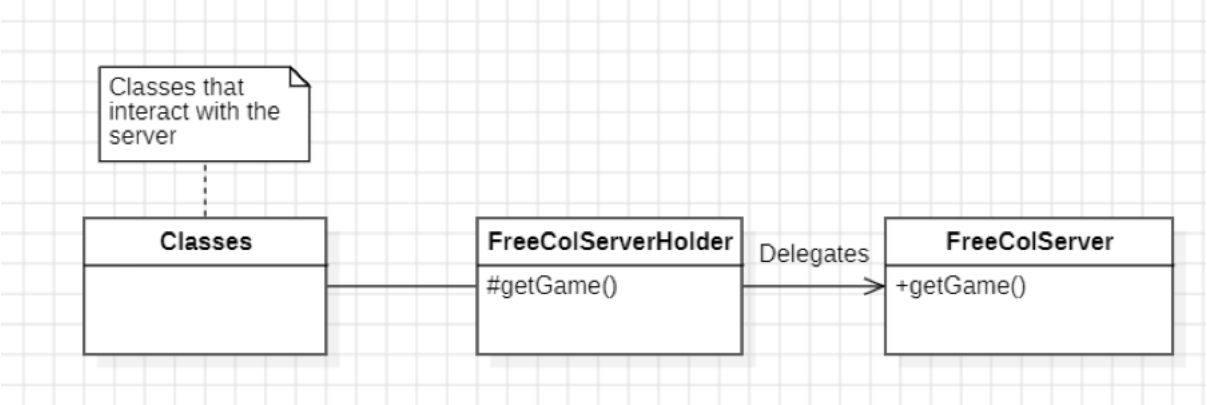
Proxy Pattern

```java
/**
 * This base class provides thread-safe access to a
 * {@link net.sf.freecol.server.FreeColServer} for several subclasses.
 */
5 usages  6 inheritors  erik_bergersjo +2
public class FreeColServerHolder {

    /** The main server object. */
    3 usages
    private final FreeColServer freeColServer;


    /** Constructor. ...*/
    4 usages  erik_bergersjo
    protected FreeColServerHolder(FreeColServer server) { this.freeColServer = server; }

    /**
     * Returns the main server object.
     *
     * @return The main server object.
     */
    erik_bergersjo
    protected FreeColServer getFreeColServer() {
        return freeColServer;
    }

    /** Get the game the server is operating. ...*/
    Michael Vehrs
    protected ServerGame getGame() { return freeColServer.getGame(); }
}
```

This design pattern is implemented in scr.net.sf.freecol.server.control.FreeColServerHolder

The class FreeColServerHolder acts as a proxy, and to be more specific, as a protection proxy.
This is when a proxy class is used to control access to the real subject class, which in this case is FreeColServer.

The Class Diagram of this pattern is:



Template Pattern

```
public abstract class Message {
```

…

```
    abstract public boolean currentPlayerMessage();

    /**
     * Get the priority of this type of message.
     *
     * @return The message priority.
     */
89 implementations   Mike Pope
    abstract public MessagePriority getPriority();


    /**
     * Does this message consist only of mergeable attributes?
     *
     * @return True if this message is trivially mergeable.
     */
3 usages   1 override   Mike Pope
    public boolean canMerge() { return false; }

    /**
     * AI-side handler for this message.
```

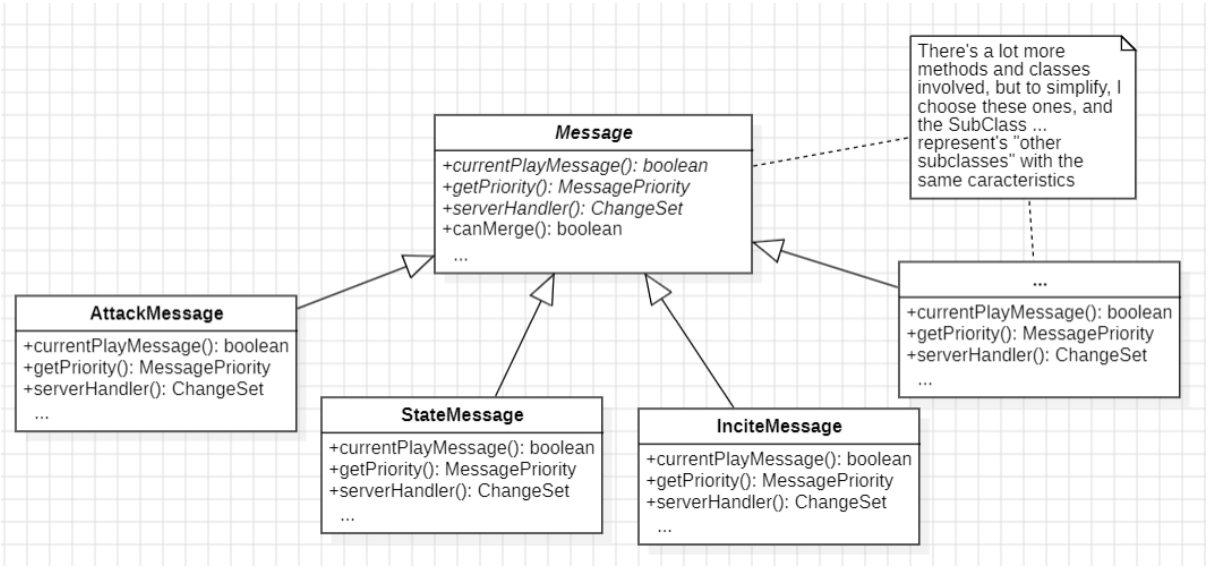
… (the class keeps going but we'll focus on a simpler example)
From ChangeStateMessage.java

```java
@Override
public boolean currentPlayerMessage() {
    return true;
}

/**
 * {@inheritDoc}
 */
Mike Pope
@Override
public MessagePriority getPriority() { return Message.MessagePriority.NORMAL; }

/**
 * {@inheritDoc}
 */
Mike Pope +2
@Override
public ChangeSet serverHandler(FreeColServer freeColServer,
                               ServerPlayer serverPlayer) {
    final String unitId = getStringAttribute(UNIT_TAG);
    final String stateString = getStringAttribute(STATE_TAG);

    Unit unit;
    try {
        unit = serverPlayer.getOurFreeColGameObject(unitId, Unit.class);
    } catch (Exception e) {
        return serverPlayer.clientError(e.getMessage());
```

This design pattern is found in scr.net.sf.freecol.common.networking.Message and it's subClasses (for example scr.net.sf.freecol.common.networking.ChangeStateMessage.

The class Message is the general class that has both generic message related methods, and abstract methods to be implemented by the subclasses.

The Class Diagram for this pattern is (the real diagram would be too big, so I made a simpler and understandable version):


There's a lot more methods and classes involved, but to simplify, I choose these ones, and the SubClass ... represent's "other subclasses" with the same caracteristics
Message
+currentPlayMessage(): boolean
+getPriority(): MessagePriority
+serverHandler(): ChangeSet
+canMerge(): boolean
...
AttackMessage
+currentPlayMessage(): boolean
+getPriority(): MessagePriority
+serverHandler(): ChangeSet
...
StateMessage
+currentPlayMessage(): boolean
+getPriority(): MessagePriority
+serverHandler(): ChangeSet
...
InciteMessage
+currentPlayMessage(): boolean
+getPriority(): MessagePriority
+serverHandler(): ChangeSet
...
...
+currentPlayMessage(): boolean
+getPriority(): MessagePriority
+serverHandler(): ChangeSet
...

Use Case

Beatriz Machado 62551 (BeatrizCaiola) - Trade and Economy

In terms of actors, the Use Case diagram illustrating Trade and Economy has the actor Player. This actor, which represents a human player, may perform all the actions stated in the diagram and some of the actions in question might involve more than one Player actor.

The actions that may be performed by the player actor are the following: Assign Trade Routes, Trade with Other Colonies, Trade with Europe, Trade with Natives, Gather Resources and Be Taxed.

Assign Trade Routes

In this Use Case the primary actor (Player) may assign Routes in order to establish trade, therefore, this case is included in the actions of Trading with Other Colonies, Trading with Europe and Trading with Natives. This action might be performed by one or more Players, as the trading relationships can be established with other Players.

Trade with Other Colonies

This Use Case represents the abstract case in which the primary actor (Player) establishes trading relationships with other colonies. This Use Case has 3 extensions: Buy Goods, Sell Goods and Trade Gold. In the extension Buy Goods, the primary actor (Player) acquires goods from other colonies, whilst in the Sell Goods extension the primary actor (Player) may put their goods up for sale. In the Trade Gold extension, the primary actor (Player) can receive from or give gold to other colonies.

Trade with Europe

This Use Case represents the abstract case in which the primary actor (Player) establishes trading relationships with European nations. This Use Case has 4 extensions: Sell Foodstuff, Buy Foodstuff, Buy Goods from Europe and Export Goods. In the extension Sell Foodstuff, the primary actor (Player) may put their produce up for sale, whilst in the Buy Foodstuff extension the primary actor (Player) can acquire food products. In the Buy Goods from Europe, the primary actor (Player) may purchase goods, whereas in the Export Goods extension, the primary actor (Player) can sell goods in Europe.

Trade with Natives

This Use Case represents the abstract case in which the primary actor (Player) establishes a trading relationship with the Native population. This Use Case has 4 extensions: Buy Goods, Sell Goods, Receive Gifts and Give Gifts. In the extension Buy Goods, the primary actor (Player) acquires goods from the Native people, whilst in the Sell Goods extension the primary actor (Player) may sell goods to them. In the Receive Gifts extension, the primary actor (Player) is given presents from the Native people, whereas in the Give Gifts extension the primary actor (player) may give them offerings.

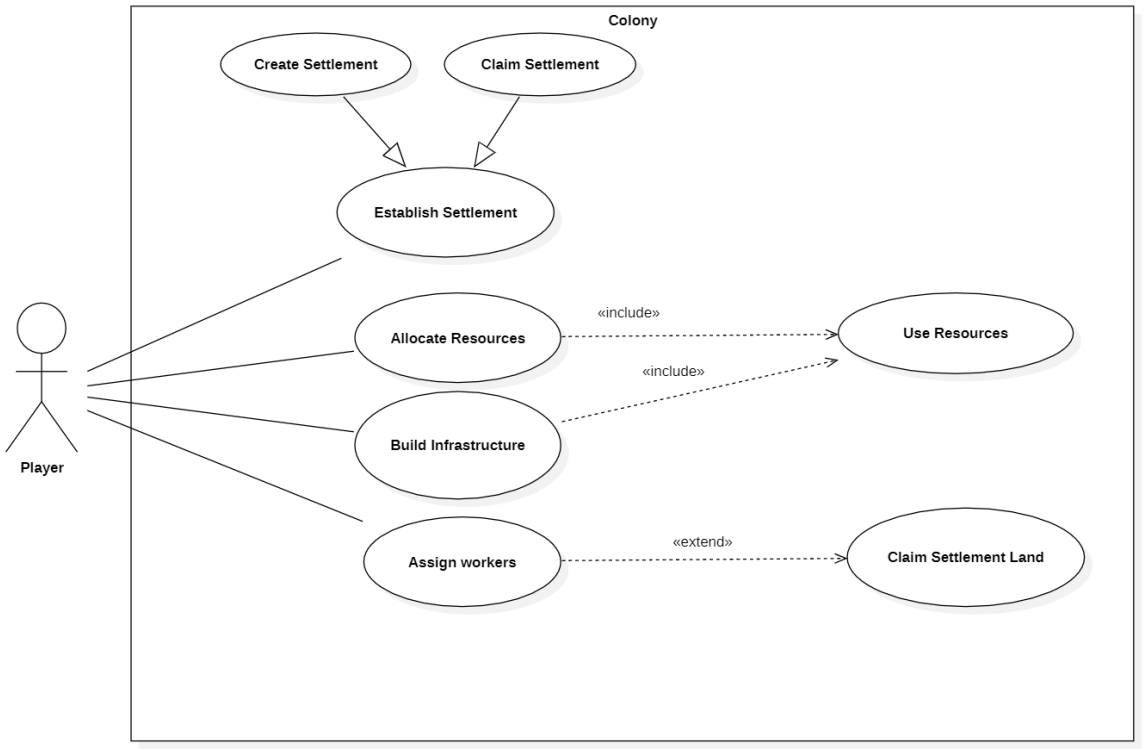
Gather Resources

This Use Case represents the abstract case in which the primary actor (Player) gathers resources that will then be traded through the established trade routes. This Use Case has 2 extensions: Mining and Gather Colony Resources. In the extension Mining, the primary actor (Player) gathers mining resources that will then be traded. In the extension Gather Colony Resources, the primary actor (Player) gathers raw materials that may then be traded as is, or traded after being used to produce other items, such as luxury goods, for example.

Be Taxed

This Use Case represents the abstract case in which the primary actor (Player) is taxed. This Use Case has two extensions: Pay Taxes and Refuse Taxes. In the extension Pay Taxes, the primary actor (Player) accepts the appointed taxes and pays the according amount. In the extension Refuse Taxes, the primary actor (Player) refuses the payment of the due taxes, which may, for instance, lead to a boycott of goods.

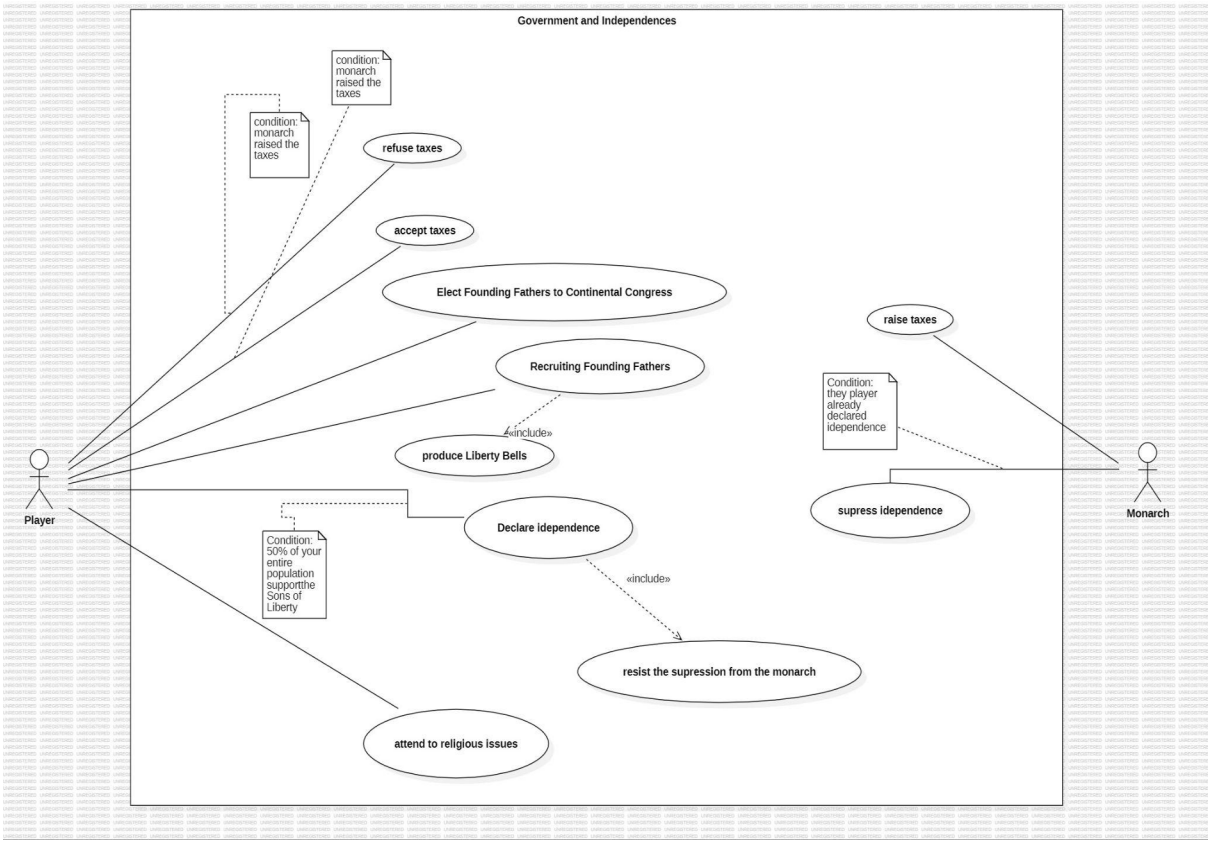
Rodrigo Monteiro Suzana 63069 (suzana2314) - Colony Management



The main actor in this use case diagram is the "Player", like the name suggests this actor is a real human, who engages with the colony to perform the following actions:

1. **Establish Settlement:** The player can either create a settlement or claim settlement. This action involves initiating or taking over an area for further development.
2. **Allocate Resources:** This action involves the management and distribution of resources within the settlement. The player decides how resources are distributed among various needs or areas within the settlement. Like for example allocation of resources for different purposes, such as construction or production.
3. **Build Infrastructure:** The player can construct various structures or elements within the settlement to enhance its functionality or capabilities. This action also involves the use of resources for construction purposes.
4. **Assign Workers:** Extends from "Claim Settlement Land" signifying the player's ability to allocate workers to claimed land by those workers, likely for specific tasks or resource gathering.

Rodrigo Fernandes 63191 (LemosFTW) - Government and Independence



This Diagram talks about the government and Independence actions. The actors of this Diagram are Time and Player, where Time represents the time that is being handled by the game system, and the player is the current player in the game.

Refusing the taxes

- Primary actor: Player
- Refusing the taxes that are imposed by the Monarch, you can only do it if the monarch raises the taxes

Accepting the taxes

- Primary actor: Player
- Accepting the taxes that are imposed by the Monarch,you can only do it if the monarch raises the taxes.

Elect Founding Fathers to Continental Congress

- Primary actor: Player
- Elect Founding Fathers to Continental Congress, they will represent you and give benefits for your nation.

Recruiting Founding Fathers

- Primary actor: Player

- Recruiting Founding Fathers, when you produce Liberty Bells you will be allowed to recruit founding fathers. They can assist you in different ways, and each of them have different benefits.

Declare Independence

- Primary actor: Player
- Declare Independence, opposing the Monarch, and being free to not pay the taxes but initiating a conflict. This implies that the Monarch will try to suppress your rebelion, if so you will fight for your freedom as an independent territory.

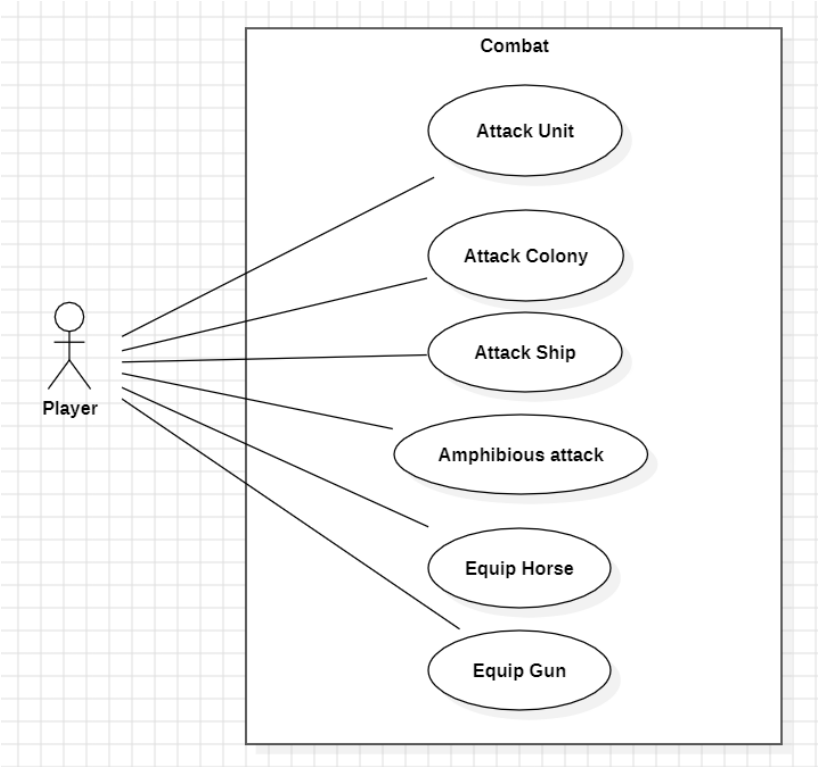
Attending religious Issues

- Primary actor: Player
- Attending religious Issues may let your colony members be satisfied with your administration.It could be a good idea if the player wants to maintain the peace in your territory.

Make Monarch raise taxes

- Primary actor: Time
- From time to time, the Monarch may decide to raise the taxes from his colonies.

João Pedro Silveira 62654 (Joao-Pedro-Silveira) - Combat



The use case diagram describing the combat in FreeCol has the following actor:

- Player, which represents a player capable of engaging in combat.

Combat:

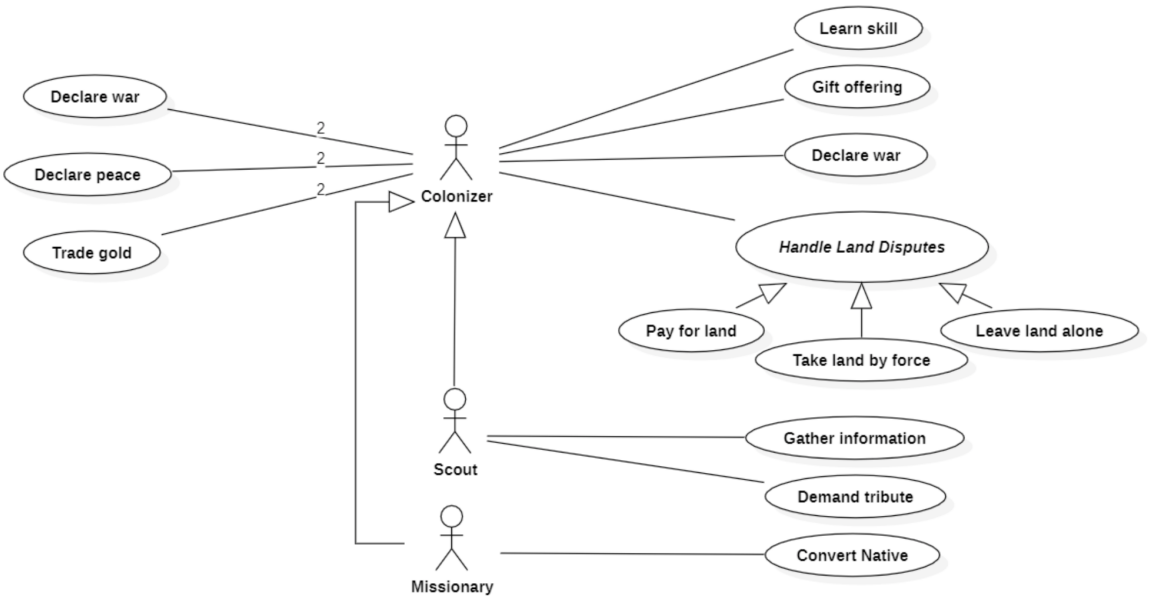
- Attack Unit:
  - Primary actor: Player
  - The player uses one of his ground combat units to attack an enemy ground unit by moving to its tile.
- Attack Colony:
  - Primary actor: Player
  - The player uses one of his ground combat units to attack an enemy colony by moving to the tile where the colony is.
- Attack Ship:
  - Primary actor: Player
  - The player uses one of his combat capable ships to attack an enemy ship by moving to its tile.
- Amphibious attack:
  - Primary actor: Player
  - The player uses a ground combat unit on board of a ship to attack an enemy coastal unit or colony.
- Equip Horse:
  - Primary actor: Player
  - The player equips 50 units of horses to a ground unit raising its combat strength.
- Equip Gun:

- Primary actor: Player
- The player equips 50 units of guns to a ground unit raising its combat strength and allowing it to start combat engagements, making it a ground combat unit.

Glossary:

- ground combat unit – a unit able to engage in ground combat, which can be a normal unit equipped with a gun or an artillery piece.
- combat capable ships – a ship capable of starting combat, which can be of three different types: Frigate, Man of War and Privateer.

Tiago Sousa 63324 (tiago-cos)



The use case diagram concerning diplomacy and interaction in FreeCol has the following actors:

- Colonizer, which represents a player controlling an European unit.
- Scout, which represents an extension of the Colonizer actor that has access to extra actions.
- Missionary, which represents an extension of the Colonizer actor that has access to extra actions.

Diplomacy With Europeans Subject

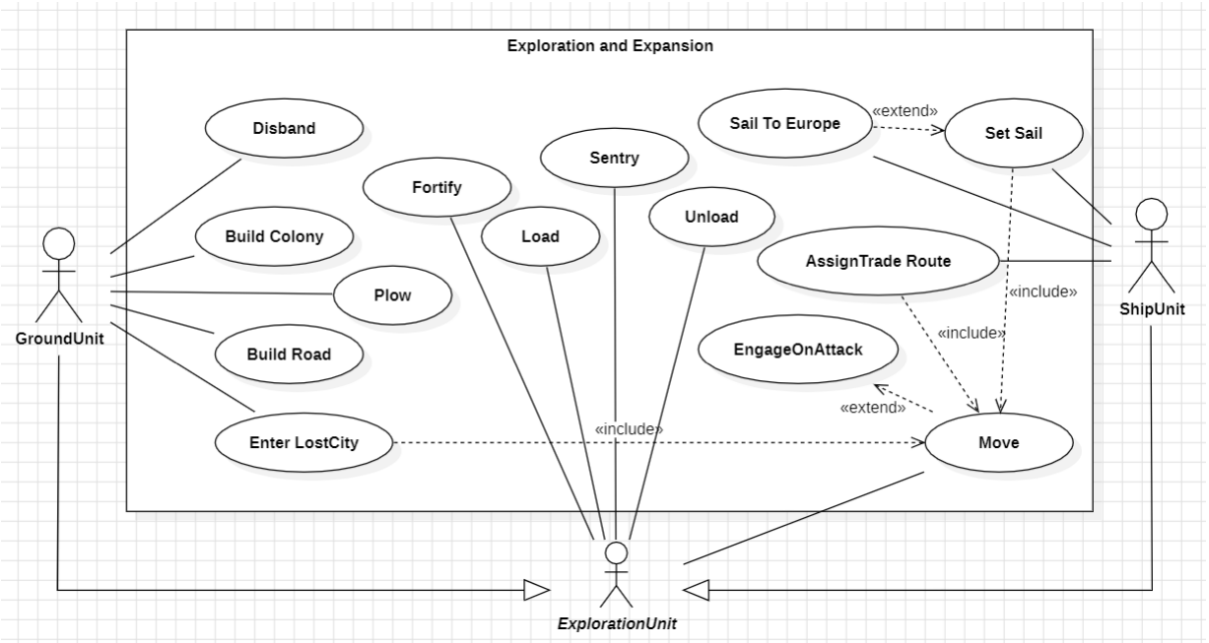
- Declare war:
    - Primary actor: Colonizer
    - A colonizer declares war against another colonizer.
- Declare peace:
    - Primary actor: Colonizer
    - A colonizer declares peace with another colonizer.
- Trade gold:
    - Primary actor: Colonizer
    - A colonizer trades their gold with another colonizer.

Diplomacy With Natives Subject

- Learn skill:
    - Primary actor: Colonizer
    - Allows a unit to learn a skill from a native settlement, becoming a specialized unit.
- Gift offering:
    - Primary actor: Colonizer

        - A unit gives an offering to another settlement.
- Declare war:
    - Primary actor: Colonizer
    - A unit declares war against another settlement.
- Handle land disputes:
    - Primary actor: Colonizer
    - Abstract use case regarding land disputes caused by trying to claim land that is already claimed
- Pay for land:
    - Primary actor: Colonizer
    - Extension of the Hand land disputes use case that solves the dispute by paying for the land.
- Take land by force:
    - Primary actor: Colonizer
    - Extension of the Hand land disputes use case that solves the dispute by taking the land by force.
- Leave land alone:
    - Primary actor: Colonizer
    - Extension of the Hand land disputes use case that solves the dispute by giving up the land.
- Gather information:
    - Primary actor: Scout
    - A scout unit talks with a native settlement leader and gathers information.
- Demand tribute:
    - Primary actor: Scout
    - A scout unit demands a tribute from a native settlement leader.
- Convert native:
    - Primary actor: Missionary
    - A missionary unit converts a native unit, allowing the colonizer player to control the converted unit.

André Branco 62482 (Aser28860d)



Actors :

- ExplorationUnit - represents a player controlling a (playable) unit in the game.
- GroundUnit - represents a unit that interacts on land(troops, workers, etc.)
- ShipUnit - represents a unit that interacts on sea (ships)

Exploration and expansion:

- Fortify:

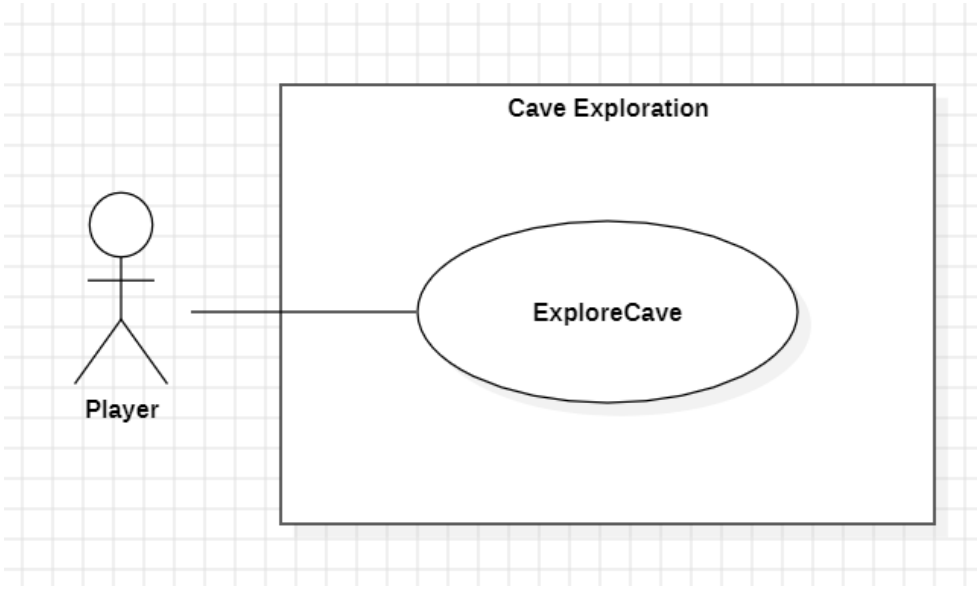
  Primary actor : ExplorationUnit
  Unit stands in place (fortifies) until new order.

- Load:
  - Primary actor : ExplorationUnit
  - Unit loads available goods (used for example, on ships and wagons)
- Unload:
  - Primary actor : ExplorationUnit
  - Unit unloads cargo (used for example, on ships and wagons)
- Sentry:
  - Primary actor : ExplorationUnit
  - Unit stands in place “waiting for something to happen
- Move:
  - Primary actor : ExplorationUnit
  - Unit moves to one of its surrounding tiles (chosen by theplayer)
- Set Sail:
  - Primary actor : ShipUnit

        - Unit sets sail to a chosen location (includes Move)
- Sail to Europe:
    - Primary actor : ShipUnit
    - Unit sets sail to Europe (extends Set Sail)
- Assign Trade Route:
    - Primary actor : ShipUnit
    - Unit starts moving following the assigned trade route (includes Move)
- Enter Lost City:
    - Primary actor : GroundUnit
    - Unit moves to the lost city (includes Move)
- Disband:
    - Primary actor : GroundUnit
    - Unit is disbanded
- Build Colony:
    - Primary actor : GroundUnit
    - Unit builds colony on it’s current map tile
- Build Road:
    - Primary actor : GroundUnit
    - Unit builds road in an adjacent map tile
- Plow:
    - Primary actor : GroundUnit
    - Unit plows it’s current map tile

Implementation Use Cases

André Branco 62482 (Aser28860d) e João Pedro Silveira 62654 (Joao-Pedro-Silveira) - Cave Exploration



Use case: CaveExploration
ID: 1
Description: A player moves one of his units to a Cave tile with a CaveExploration, yielding a good, bad or neutral result
Primary Actors: **Player**
Secondary Actors: None
Preconditions: 1. The unit to be moved is adjacent to a Cave tile with a CaveExploration.

Main flow:

1. The use case starts when the player moves one of his units to a Cave tile with a CaveExploration.
2. The system will then check the probabilities of **Good**, **Bad** and **Neutral** results according to the game's current difficulty.
3. If the system deems that the CaveExploration yields a **Good** result.
    1. The system will choose a reward.
    2. If the unit can learn and the reward is **Learn** then the unit will become of the type Seasoned Scout.
    3. If the reward is **Colonist** then a new unit of one of the types that are found in caves will be generated on the tile where the CaveExploration is and given to the player.
    4. If the reward is **Treasure** then a treasure cart holding between 100 and 500 gold will be generated on the tile where the CaveExploration is and given to the player.
    5. If the reward is **Resources** then a wagon train holding between 40 and 140 units of a certain Good that is found in caves is generated on the tile where the CaveExploration is and given to the player.
4. If the system deems that the CaveExploration yields a **Bad** result.
    1. The system will choose a penalty.
    2. If the penalization is **Trap** the unit performing the CaveExploration will take damage.
        1. If the unit is mounted it will be dismounted.
        2. If the unit is dismounted and is armed or has tools equipped it will be disarmed or lose the tools.
        3. If the unit is dismounted and is neither armed nor has tools equipped it will die.
    3. If the penalization is **Lethal Trap** the unit will die.
5. If the system deems that the CaveExploration yields a **Neutral** result nothing will happen.
6. The CaveExploration will have another one of its floors explored.
7. If the CaveExploration runs out of floors it will disappear.

PostConditions: None

Alternative flows:
UnitCannotExploreCave

Alternative flow: CaveExploration: UnitCannotExploreCave

ID: 1.1

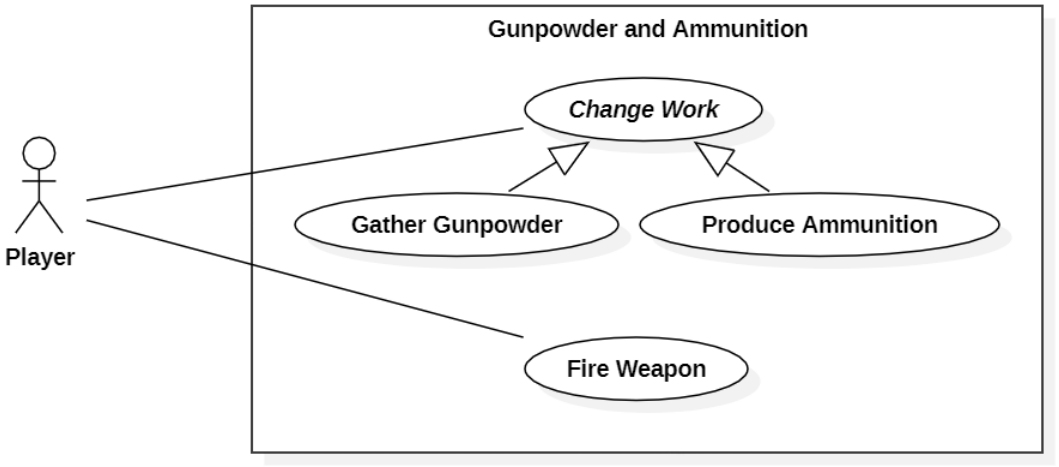
Description:
A player moves one of his units to a Cave tile with a CaveExploration, but the unit is unable to explore the cave.

Primary Actors: **Player**
Secondary Actors: None
Preconditions: 1. The player has chosen a unit of type **Artillery**, **Wagon Train** or **Treasure Cart**.
Main flow: 1. The alternative flow begins after step 1 of the main flow. 2. The CaveExploration is not explored. Nothing happens.
PostConditions: None

Glossary:

- Goods Found in caves: tobacco, cotton, fur, lumber, ore, silver, rum, cigars, cloth, coats, tradeGoods, tools, hammers.
- Colonists Found in caves: Free Colonist, Expert Silver miner, Expert Ore miner, Expert Gunpowder gatherer.

Beatriz Machado 62551 (BeatrizCaiola) e Tiago Sousa 63324 (tiago-cos) - Gunpowder and Ammunition



Use case: *Change Work*
ID: 1
Brief description: The Player changes the job of a unit in a colony.
Primary actors: Player
Secondary actors: None
Preconditions: 1. The Player has units in a colony. 2. The colony has tiles that can produce goods adjacent to it or it has buildings which can refine goods.
Main flow: 1. The use case starts when the Player selects the “Change Work” option while managing a unit inside of a colony. 2. The system gathers all possible occupations based on nearby tiles and colony buildings and displays them to the Player. 3. The Player selects an occupation from the list presented by the system. 4. The system sends a message that alters the unit workType in order to create the goods produced by the occupation.

Postconditions: 1. The unit's workType matches their occupation.
Alternative flows: None.

Use case: Gather Gunpowder
ID: 2
Specializes: *Change work*
Brief description: The Player changes the job of a unit in a colony to Gunpowder Gatherer.
Primary actors: Player
Secondary actors: None
Preconditions: 1. The Player has units in a colony. 2. The colony is adjacent to a cave tile. 3. The cave tile does not have any tile items on it.
Main flow: 1. The use case starts when the Player selects the "Change Work" option while managing a unit inside of a colony. 2. The system gathers all possible occupations based on nearby tiles and colony buildings and displays them to the Player. 3. **(o3.)** The Player selects the "Gunpowder Gatherer" option from the list presented by the system. 4. **(o4.)** The system sends a message that alters the unit workType in order to create gunpowder.
Postconditions: 1. The unit's workType is Gunpowder.
Alternative flows: None.

Use case: Produce Ammunition
ID: 3
Specializes: *Change work*
Brief description: The Player changes the job of a unit in a colony to Armourer.
Primary actors: Player
Secondary actors: None
Preconditions: 1. The Player has units in a colony. 2. The colony has built an ammunition building.
Main flow: 1. The use case starts when the Player selects the “Change Work” option while managing a unit inside of a colony. 2. The system gathers all possible occupations based on nearby tiles and colony buildings and displays them to the Player. 3. **(o3.)** The Player selects the “Armourer” option from the list presented by the system. 4. **(o4.)** The system sends a message that alters the unit workType in order to create ammunition.
Postconditions: 1. The unit’s workType is Ammunition.
Alternative flows: None.

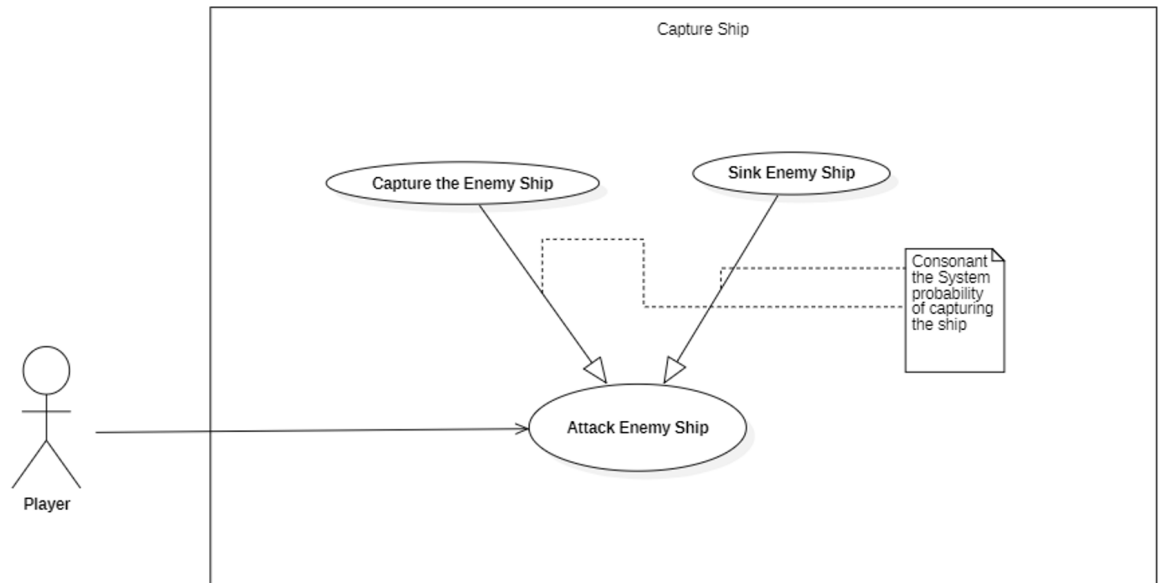
Use case: Fire Weapon
ID: 4
Brief description: The Player orders a soldier unit to attack another unit by firing their weapon.
Primary actors: Player
Secondary actors: None
Preconditions: 1. The unit controlled by the Player has a musket. 2. The unit controlled by the Player has ammunition. 3. The unit controlled by the Player has remaining moves. 4. The unit the Player wants to attack belongs to another nation. 5. The unit the Player wants to attack is adjacent to the unit the Player is controlling.

Main flow:
 	
1. The use case starts when the Player orders a soldier unit to move to the same location as a unit belonging to another nation.
    1. The system requests confirmation from the Player on whether they want to attack the unit or not.
2. If the Player refuses
    1. The system closes the confirmation window and no combat happens.
3. If the Player accepts
    1. The system sends a message that causes a combat interaction between the two units.

Postconditions: 1. The unit has one less ammunition than what it had before. 2. Combat has occurred between the two units.
Alternative flows: No Ammunition Left

Alternative flow: Fire Weapon:No Ammunition Left
ID: 4.1
Brief description: The unit that the Player ordered has his role changed to the “no ammo” variant by the system.
Primary actors: Player
Secondary actors: None
Preconditions: 1. The unit controlled by the Player has run out of ammunition.
Alternative flow: 1. The alternative flow begins after step 3.1 of the main flow. 2. The system verifies that the unit has no ammunition left due to his combat. 3. The system changes the unit role to its “no ammo” variant.
Postconditions: 2. The unit’s role is a “no ammo” variant.

Rodrigo Monteiro Suzana 63069 (suzana2314) e Rodrigo Fernandes 63191 (LemosFTW) - Capture Ship



Use Case: Capture Ship
ID: 5
Brief Description: When attacking an enemy ship, the player can either capture it or sink it. This is determined randomly.
Primary actors: Player
Secondary actors: None
Preconditions: Attacking ship is naval and the defending ship is naval and must exist.
Main Flow: 1. The use case begins when the player orders a ship to attack an enemy ship; 2. The player wins the battle; 3. The system decides if the player can capture the ship or if the ship is sunk;
Postconditions: If the winning player captured the ship it must be owned by him.
Alternative Flows: The alternative flow begins after step 1 of the main flow. The player loses the battle.

Alternative Flow: Player Loses
ID: 5.1
Brief Description: The player loses the battle
Primary actors: Player
Secondary actors: None
Preconditions: Attacking ship is naval and the defending ship is naval and must exist.
Alternative Flows: 1. The alternative flow begins after step 1 of the main flow. 2. The player loses the battle.
Postconditions: Loser player either has it’s ship destroyed or damaged.

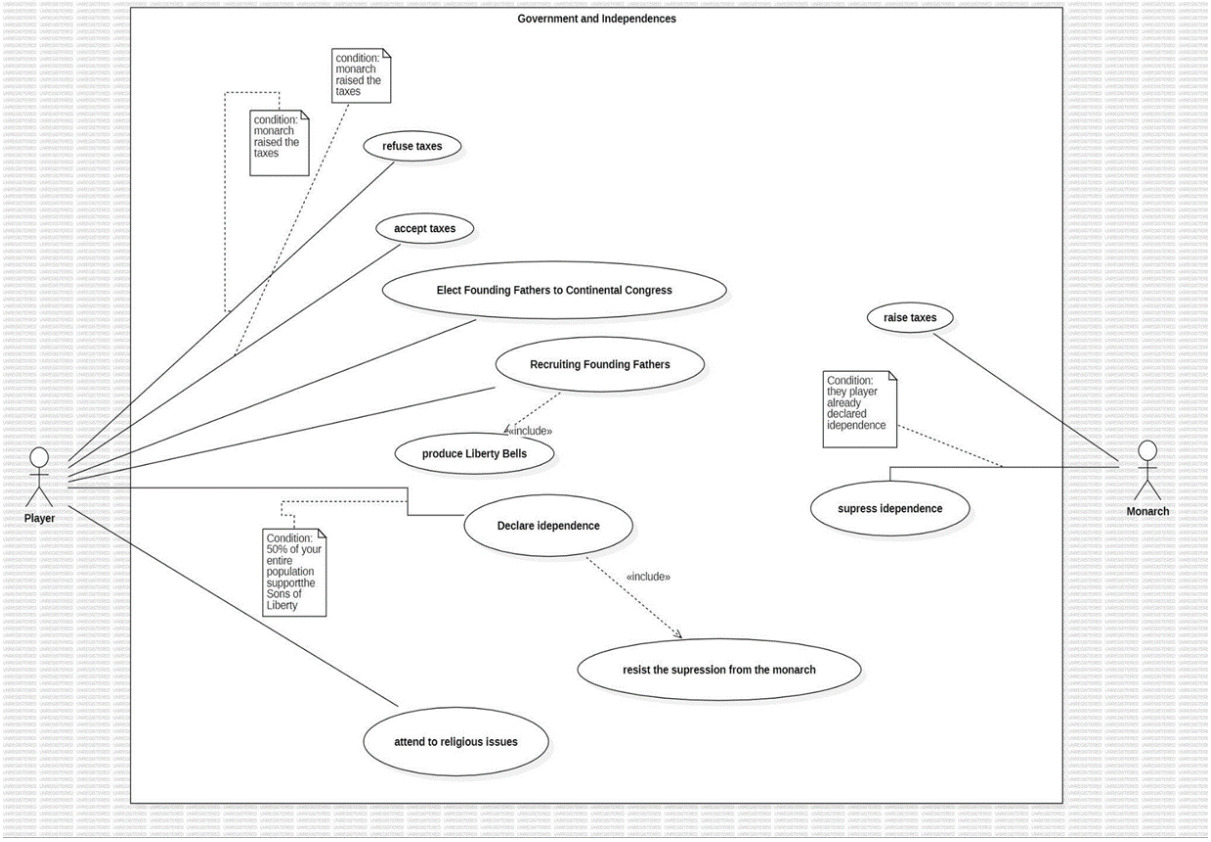
Postmortem

Rodrigo Fernandes 63191 (LemosFTW)

Use Case Diagram

The main actor in this use case diagram is the “Player”, i.e., the human Player who is playing the game. We also have a Secondary actor, Monarch, that is an AI.

The main actor can interact with this context by:



1. **Refusing the taxes** that are imposed by the Monarch, you can only do it if the monarch raises the taxes.
2. **Accepting the taxes** that are imposed by the Monarch,you can only do it if the monarch raises the taxes.
3. **Elect Founding Fathers to Continental Congress,** they will represent you and give benefits for your nation.
4. **Recruiting Founding Fathers,** when you produce Liberty Bells you will be allowed to recruit founding fathers. They can assist you in different ways, and each of them have different benefits.

5. **Declare Independence,** opposing the Monarch, and being free to not pay the taxes but initiating a conflict. This implies that the Monarch will try to suppress your rebelion, if so you will fight for your freedom as an independent territory.
6. **Attending religious Issues** may let your colony members be satisfied with your administration.It could be a good idea if the player wants to maintain the peace in your territory.

Justification: We found a better way to represent the Use Case.

MOOD Code Metrics

All the values are rounded in 2 decimals cases.

Attribute hiding factor 73,09%

Attribute inheritance factor 81,15%

Coupling factor 3,06%

Method hiding factor 25,33%

Method inheritance factor 72,83%

Polymorphism factor 09,01%

As a good project should be, the COUPLING factor is low, this indicates that there is not a high dependency between the classes, also the METHOD INHERITANCE factor is high, this indicates that the classes are well designed and the inheritance is used in a good way, but the ATTRIBUTE INHERITANCE factor is high too, this can be a bad thing to the project, because it indicates that most of the attributes are not private.

The POLYMORPHISM factor is low, this indicates that the classes are not using polymorphism, this is not a bad thing to have in a big project.

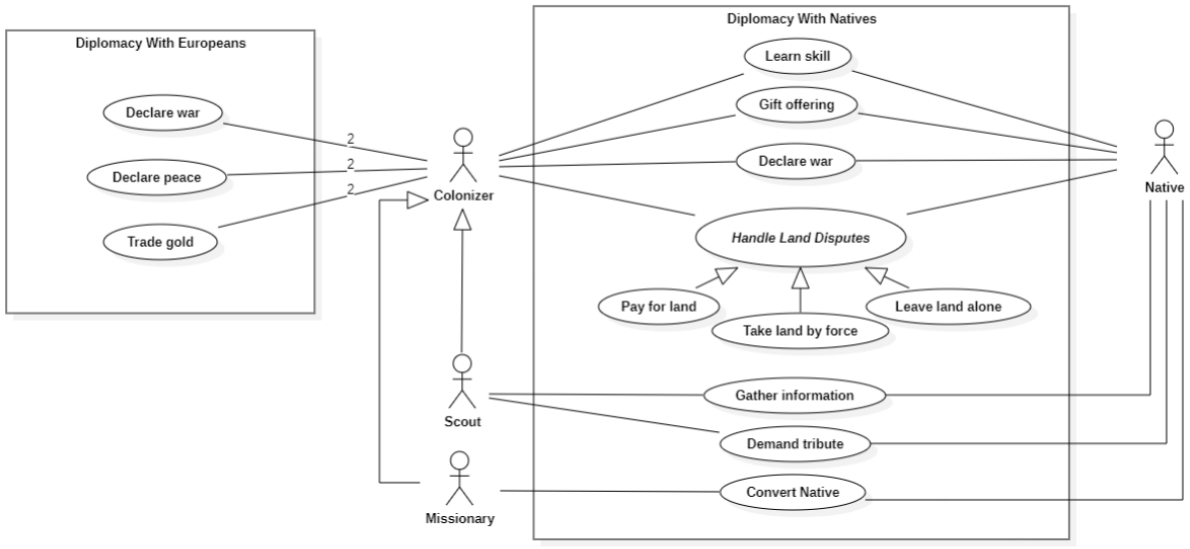
The METHOD HIDING factor is on average(8% to 25%), a low MHF indicates insufficiently abstracted implementation, and a high MHF indicates very little functionality.

The ATTRIBUTE HIDING factor is high, this indicates that the attributes are hidden, this is good, allowing other classes to see only the attributes that are needed.

Justification: MOOD metrics plugin only shows the global data analysis, then it is not possible to find code smells with these metrics. In the next deliverable I will use new metrics and find code smells using these metrics.

Tiago Sousa 63324 (tiago-cos)

Use Case Diagram



The use case diagram concerning diplomacy and interaction in FreeCol has the following actors:

- Colonizer, which represents a player (human or otherwise) controlling an European unit.
- Scout, which represents an extension of the Colonizer actor that has access to extra actions.
- Missionary, which represents an extension of the Colonizer actor that has access to extra actions.
- Native, which represents a player (human or otherwise) controlling a Native unit.

Diplomacy With Europeans Subject

- Declare war:
    - Primary actor: Colonizer
    - A colonizer declares war against another colonizer.

- Declare peace:
    - Primary actor: Colonizer
    - A colonizer declares peace with another colonizer.

- Trade gold:
    - Primary actor: Colonizer
    - A colonizer trades their gold with another colonizer.

Diplomacy With Natives Subject

- Learn skill:
    - Primary actor: Colonizer
    - Secondary actor: Native
    - Allows a unit to learn a skill from a native settlement, becoming a specialized unit.

- Gift offering:
    - Primary actor: Colonizer
    - Secondary actor: Native
    - A unit gives an offering to another settlement.

- Declare war:
    - Primary actor: Colonizer
    - Secondary actor: Native
    - A unit declares war against another settlement.

- Handle land disputes:
    - Primary actor: Colonizer
    - Secondary actor: Native
    - Abstract use case regarding land disputes caused by trying to claim land that is already claimed

- Pay for land:
    - Primary actor: Colonizer
    - Secondary actor: Native
    - Extension of the Hand land disputes use case that solves the dispute by paying for the land.

- Take land by force:
    - Primary actor: Colonizer
    - Secondary actor: Native
    - Extension of the Hand land disputes use case that solves the dispute by taking the land by force.

- Leave land alone:
    - Primary actor: Colonizer
    - Secondary actor: Native
    - Extension of the Hand land disputes use case that solves the dispute by giving up the land.

- Gather information:
    - Primary actor: Scout
    - Secondary actor: Native
    - A scout unit talks with a native settlement leader and gathers information.

- Demand tribute:
    - Primary actor: Scout
    - Secondary actor: Native
    - A scout unit demands a tribute from a native settlement leader.

- Convert native:
    - Primary actor: Missionary
    - Secondary actor: Native
    - A missionary unit converts a native unit, allowing the colonizer player to control the converted unit.

Justification: We found a better way to represent the Use Case.

Beatriz Machado 62551 (BeatrizCaiola)

Use Case: Trade and Economy

In terms of actors, the Use Case diagram illustrating Trade and Economy has the actor Player. This actor, which represents either a human or a non-human player, may perform all the actions stated in the diagram and some of the actions in question might involve more than one Player actor.

The actions that may be performed by the player actor are the following: Assign Trade Routes, Trade with Other Colonies, Trade with Europe, Trade with Natives, Gather Resources and Be Taxed.

Assign Trade Routes

In this Use Case the primary actor (Player) may assign Routes in order to establish trade, therefore, this case is included in the actions of Trading with Other Colonies, Trading with Europe and Trading with Natives. This action might be performed by one or more Players, as the trading relationships can be established with other Players, may they be human or not.

Trade with Other Colonies

This Use Case represents the abstract case in which the primary actor (Player) establishes trading relationships with other colonies. This Use Case has 3 extensions: Buy Goods, Sell Goods and Trade Gold. In the extension Buy Goods, the primary actor (Player) acquires goods from other colonies, whilst in the Sell Goods extension the primary actor (Player) may put their goods up for sale. In the Trade Gold extension, the primary actor (Player) can receive from or give gold to other colonies.

Trade with Europe
This Use Case represents the abstract case in which the primary actor (Player) establishes trading relationships with European nations. This Use Case has 4 extensions: Sell Foodstuff, Buy Foodstuff, Buy Goods from Europe and Export Goods. In the extension Sell Foodstuff,

the primary actor (Player) may put their produce up for sale, whilst in the Buy Foodstuff extension the primary actor (Player) can acquire food products. In the Buy Goods from Europe, the primary actor (Player) may purchase goods, whereas in the Export Goods extension, the primary actor (Player) can sell goods in Europe.

Trade with Natives

This Use Case represents the abstract case in which the primary actor (Player) establishes a trading relationship with the Native population. This Use Case has 4 extensions: Buy Goods, Sell Goods, Receive Gifts and Give Gifts. In the extension Buy Goods, the primary actor (Player) acquires goods from the Native people, whilst in the Sell Goods extension the primary actor (Player) may sell goods to them. In the Receive Gifts extension, the primary actor (Player) is given presents from the Native people, whereas in the Give Gifts extension the primary actor (player) may give them offerings.

Gather Resources

This Use Case represents the abstract case in which the primary actor (Player) gathers resources that will then be traded through the established trade routes. This Use Case has 2 extensions: Mining and Gather Colony Resources. In the extension Mining, the primary actor (Player) gathers mining resources that will then be traded. In the extension Gather Colony Resources, the primary actor (Player) gathers raw materials that may then be traded as is, or traded after being used to produce other items, such as luxury goods, for example.

Be Taxed

This Use Case represents the abstract case in which the primary actor (Player) is taxed. This Use Case has two extensions: Pay Taxes and Refuse Taxes. In the extension Pay Taxes, the primary actor (Player) accepts the appointed taxes and pays the according amount. In the extension Refuse Taxes, the primary actor (Player) refuses the payment of the due taxes, which may, for instance, lead to a boycott of goods.

Justification: Removed AI from textual description, as it is part of the system.