Engenharia de Software
2023/2024

Merge Document

Docentes:
João Araújo
Vasco Amaral

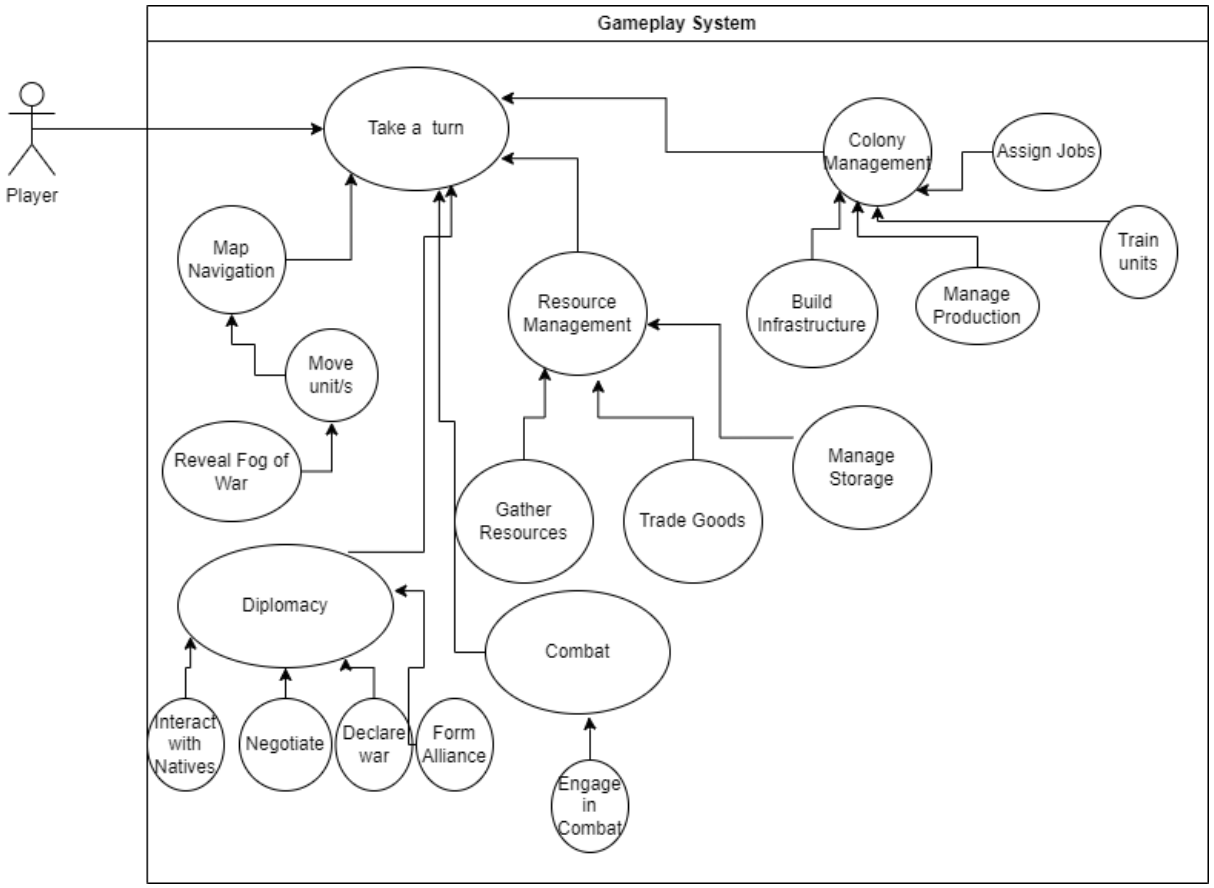
Discentes:
Diogo Lemos 56837
Gustavo Silva 59472
José Trigueiro 58119
José Pereira 55204
Liliane Correia 58427
Wilker Martins 58535

Table of Contents

José Pereira - 55204

Use Case Diagram:

Gameplay System Use Case



- **Use Case Name:** Gameplay System
- **Description:** For each turn, the player has the liberty to engage in a variety of strategic activities: Exploration, Diplomacy, Combat, Trade, Colony Management, etc.
- **Actors**: Player

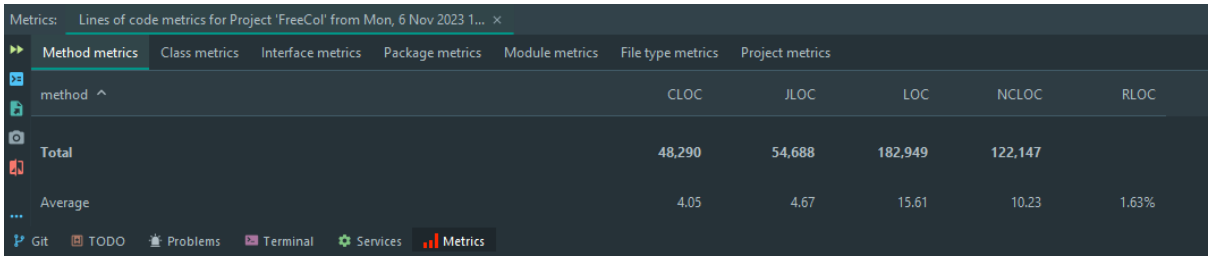
Review(s): Reviewed by Gustavo Silva 59472 (08/11/2023):

- Use case Diagram seems to represent the correct information.

Code Base Metrics:

Lines of Code Metrics

Method metrics:



Explanation of Method Metrics:

- **CLOC (Comment Lines of Code):** Indicates the number of lines that are comments within method bodies. This metric helps assessing the level of documentation and comments in the code.
- **JLOC (Java Lines of Code):** Refers to the number of Java lines of code, excluding comments and blank lines, within method bodies.
- **LOC (Lines of Code):** Total number of lines of code, including comments and blank lines within method bodies.
- **NCLOC (Non-Comment Lines of Code):** The number of lines of actual code without comments within method bodies.
- **RLOC (Relative Lines of Code):** Percentage of the lines of code relative to the total lines of code within method bodies. It's shown as 1.63%, which might indicate the relative size of method bodies to the entire codebase.

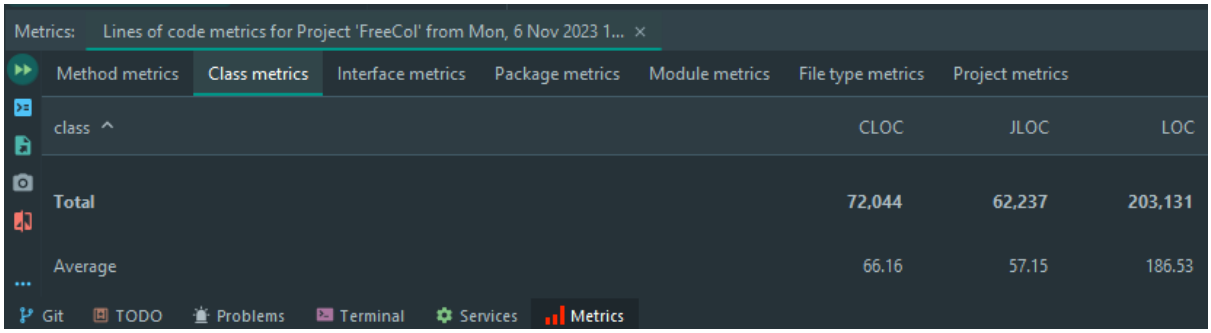
Example of a method with high LOC and NCLOC

- This means that perhaps the method is too complex, doing too much (God Method code smell) and could benefit from refactoring.

Example of a method with high RLOC

- This could also mean that refactoring is necessary since 88.10% of the lines of code in the method's class are within that method.

Class metrics:



- **CLOC (Comment Lines of Code):** Reflects the number of lines of comments within classes, indicating how well-documented the classes are.
- **JLOC (Java Lines of Code):** Total number of Java statement lines in classes, not including comments and blank lines.
- **LOC (Lines of Code):** Total number of lines that make up the classes, including code, comments, and blank lines.

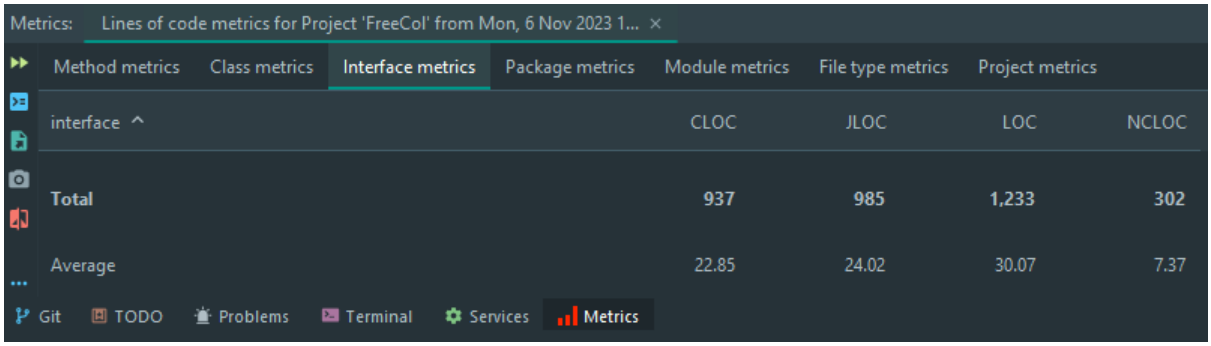
- Similar to method metrics, high LOC in a class may suggest that a class is doing too much, potentially violating the Single Responsibility Principle.
- An average CLOC that is significantly lower than the average LOC might suggest under-documentation, whereas a very high average might imply over-commenting which could also be a maintenance issue if comments are not kept up-to-date with changes in code.
- High JLOC can indicate complex classes that may be difficult to maintain or extend.
- Classes with high LOC could be symptomatic of 'Large Class' or 'God Class' code smells, where a class has too many responsibilities.
- Excessive commenting might be indicative of 'Comment Smells,' where the code is not self-explanatory and relies too much on comments for clarity.

These class-level metrics help in assessing the complexity and maintainability of classes.

They can provide insights into the need for refactoring at the class level, such as splitting up classes that are too large or enhancing those with inadequate documentation.

Tracking these metrics over time can help in identifying trends, such as a gradual increase in class size, which could lead to maintainability issues down the line.

Interface metrics:



CLOC (Comment Lines of Code): The number of lines of comments within interfaces, giving an insight into the documentation level of the interfaces.

JLOC (Java Lines of Code): Represents the number of effective Java statement lines in interfaces, not including comments or blank lines.

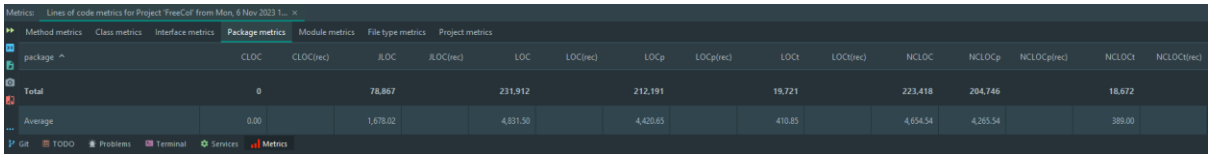
LOC (Lines of Code): The total line count for interfaces, encompassing code, comments, and blank lines.

NCLOC (Non-Comment Lines of Code): The count of actual code lines within interfaces, excluding comments.

- Interfaces with an unusually high LOC may be attempting to define too much functionality, potentially violating interface segregation principles.
- A high CLOC relative to LOC could indicate that the interfaces are well-documented, which is usually positive unless the comments are excessively verbose or not meaningful.
- An average NCLOC of 7.37 suggests that the interfaces are quite slim in terms of actual code, which is generally a good practice in interface design.
- Large interfaces (indicated by high LOC) might be a sign of 'Interface Bloat', where an interface contains more methods than its implementers actually require.
- If the CLOC is very low, it could suggest that the interfaces lack adequate documentation, which can make them difficult to understand and implement correctly.
- Interface metrics can be particularly useful in identifying the need for interface segregation or splitting up complex interfaces.
- The balance between CLOC and NCLOC can guide developers on how well the interfaces are explained, which is critical since interfaces define core contracts in the codebase.

These metrics, when tracked over time, can also indicate whether the complexity of interfaces is increasing, which might lead to a more rigid and less maintainable system.

Package metrics:



CLOC (Comment Lines of Code): It shows zero comment lines of code at the package level, which could be due to the packaging structure not containing comments or it not being the typical place where comments are expected or extracted from.

LOC (Lines of Code): The total lines of code in packages, which include everything (comments, whitespace, and actual code).

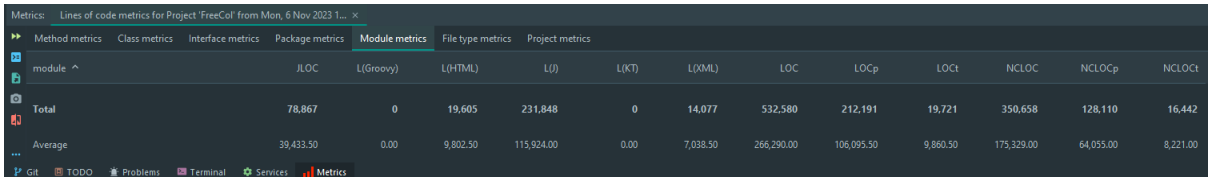
NCLOC (Non-Comment Lines of Code): Represents the number of executable or logical lines of code, excluding comments and empty lines.

- The absence of CLOC at the package level isn't necessarily an issue, but it could suggest a lack of package-level documentation. This might be a point to consider for improvement.
- The average NCLOC at the package level suggest sizable packages. It might indicate that the packages are doing too much or are too large, which could be a sign of poor modularity or low cohesion.
- If certain packages have a particularly high NCLOC compared to others, it might indicate a "Large Class" smell at the package level, meaning that the package could be trying to do too much and might benefit from being broken.

These metrics provide insight into the size and complexity of packages. Large packages can be more difficult to maintain and understand. They can also indicate the scope of changes when packages undergo revisions, affecting testing and integration efforts.

Tracking the evolution of these metrics over time can help in assessing whether the project's modularity and structure are improving or deteriorating, which can guide refactoring efforts.

Module metrics:



JLOC: This stands for Java Lines of Code. Here, we see 78,867 which likely indicates the number of lines of Java code across the module.

L(Groovy): This would represent lines of Groovy code, which is another JVM language. It shows 0, indicating there's no Groovy code in the module.

L(HTML): Lines of HTML code, which are 19,605. This indicates that there's a substantial amount of HTML, possibly suggesting that this module contains web interface elements.

L(J): This stands for lines of Java code including comments and blank lines.

L(KT): Represents lines of Kotlin code, which there aren't.

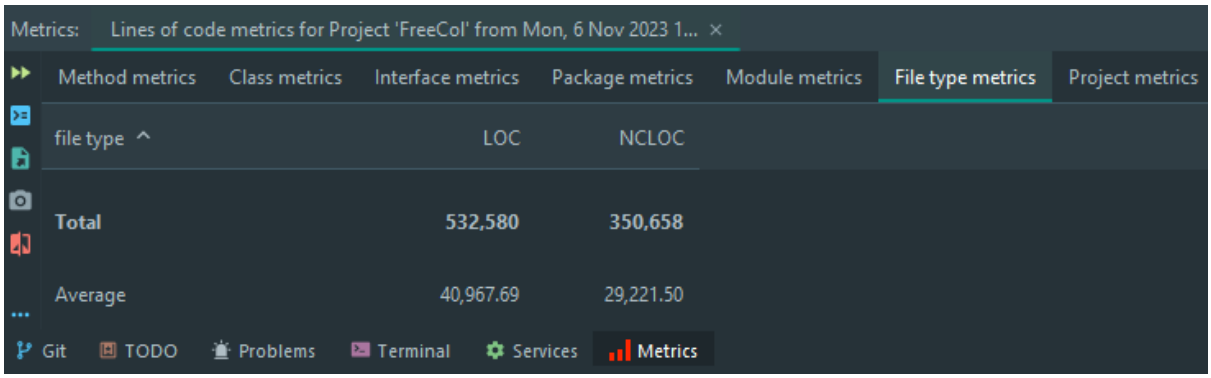
L(XML): Represents lines of XML code.

LOC: This is the total Lines of Code and is very high at 532,580, suggesting a large module with a lot of code.

NCLOC: Non-Comment Lines of Code are at 350,658, indicating the actual lines of code minus comments, which is still quite substantial.

- With such a high count of lines of code, this module may be quite complex, potentially housing a diverse range of functionalities. This could make maintenance and understanding more difficult.
- If the module has this much code, it's worth examining whether it adheres to the Single Responsibility Principle. Modules should ideally have one clear purpose.
- With 19,721 lines of test code, we can infer that there is a substantial amount of testing, but we would need to compare the amount of test code to the amount of production code to determine if test coverage is adequate.
- The difference between LOC and NCLOC could give an indication of how much the module is documented. Proper documentation is crucial for maintainability, especially in large codebases.
- Potential for Code Smells: Large modules can sometimes be indicative of "God Objects" or classes that have too many responsibilities. They can become difficult to manage and are prone to errors.

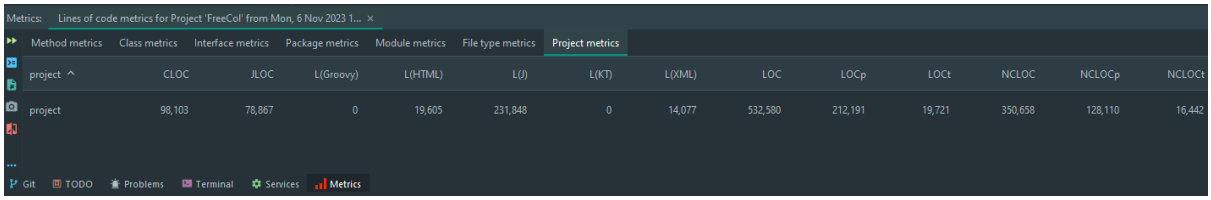
File type metrics:



LOC (Lines of Code): This indicates the total number of lines of code in the project, including comments, whitespace, and actual code lines. The total LOC here is 532,580.

NCLOC (Non-Comment Lines of Code): This represents the number of lines that are not comments or blank lines, which typically equates to the amount of "actual" code. The total NCLOC is 350,658.

Project metrics:



CLOC (Comment Lines of Code): This metric, showing 98,103 lines, suggests how much of the codebase is documented through comments.

JLOC: Stands for Java Lines of Code. Here, we see 78,867 which indicates the number of lines of Java code across the project.

L (Groovy): Since this is at 0, it indicates that there are no lines of Groovy code.

LOC (Lines of Code): The total lines, 532,580, give you a rough measure of the project's size.

LOCp (Productive Lines of Code): The 212,191 lines here are directly contributing to the functionality of the software product.

LOCt (Test Lines of Code): With 19,721 lines of test code, it shows a commitment to testing in the project. A high ratio of test code to productive code can be a good sign, indicating a dedication to software quality and stability.

NCLOC (Non-Comment Lines of Code): 350,658 lines of executable code without comments. This figure helps in understanding the volume of the code that actually performs the operations and functionalities of the project.

NCLOCp (Non-Comment Lines of Code for the Product): The 128,110 lines here shows how much of the codebase is non-comment and non-test code. This helps identify the core of the project's functionality.

NCLOCt (Non-Comment Lines of Code for Tests): Finally, the 16,442 lines indicate how much test code there is excluding comments. It can be used to assess the quality and depth of testing in relation to the productive code.

Review(s): Reviewed by Gustavo Silva 59472 (08/11/2023):

- Method Metrics analysis is valid, that method might need to be refactored.

- With the numbers given we can see that the average lines of code aren´t high which in fact means the code is easier to maintain.

- Interface Metrics conclusions are valid.

- Package Metrics suggest the lack of documentation about the packages.

GOF Patterns:

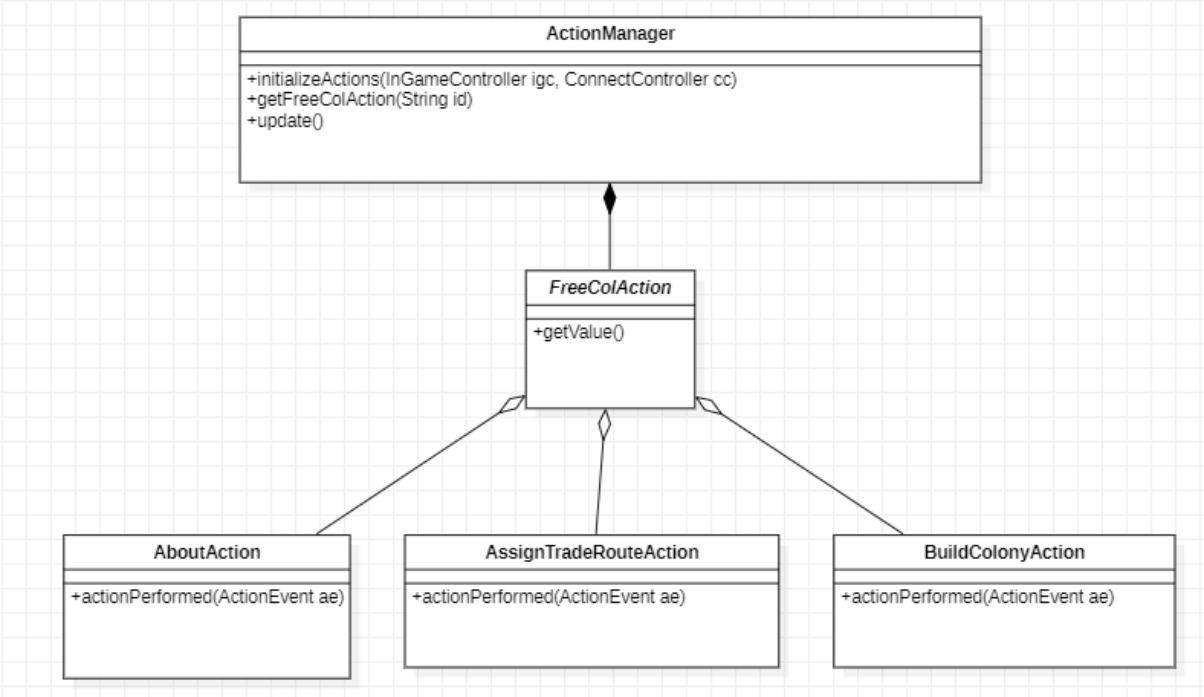
Design Patterns

1. **Command Pattern** (src/net/sf/freecol/client/gui/action/ActionManager.java)

```
73      public void initializeActions(InGameController inGameController,
74                                    ConnectController connectController) {
75          /**
76           * Please note: Actions should only be created and not initialized
77           * with images etc. The reason being that initialization of actions
78           * are needed for the client options ... and the client options
79           * should be loaded before images are preloaded (the reason being that
80           * mods might change the images).
81           */
82
83          /**
84           * Possible FIXME: should we put some of these, especially the
85           * move and tile improvement actions, into OptionGroups of
86           * their own? This would simplify the MapControls slightly.
87           */
88
89          // keep this list alphabetized.
90          add(new AboutAction(freeColClient));
91          add(new AssignTradeRouteAction(freeColClient));
92          add(new BuildColonyAction(freeColClient));
93          add(new CenterAction(freeColClient));
94          add(new ChangeAction(freeColClient));
95          add(new ChangeWindowedModeAction(freeColClient));
96          add(new ChatAction(freeColClient));
97          add(new ClearOrdersAction(freeColClient));
```

Abstract class location: src/net/sf/freecol/client/gui/action/FreeColAction.java

```
/**
 * The super class of all actions in FreeCol.  Subclasses of this
 * object is stored in an {@link ActionManager}.
 */
80 inheritors  Mike Pope + 10
public abstract class FreeColAction extends AbstractAction
    implements Option<FreeColAction> {
```



Since every action extends this abstract class and with every new action added we have the keyword "new" such as in "new AboutAction(freeColClient)", it showcases a concrete implementation of a command. The calls to "add" are encapsulating a request as an object.

2. **Strategy Pattern** (src/net/sf/freecol/client/gui/action/ActionManager.java)

```
add(new AboutAction(freeColClient));
add(new AssignTradeRouteAction(freeColClient));
add(new BuildColonyAction(freeColClient));
add(new CenterAction(freeColClient));
add(new ChangeAction(freeColClient));
add(new ChangeWindowedModeAction(freeColClient));
add(new ChatAction(freeColClient));
add(new ClearOrdersAction(freeColClient));
```

The ActionManager class also uses glimpses of the Strategy pattern through its interaction with FreeColAction objects.

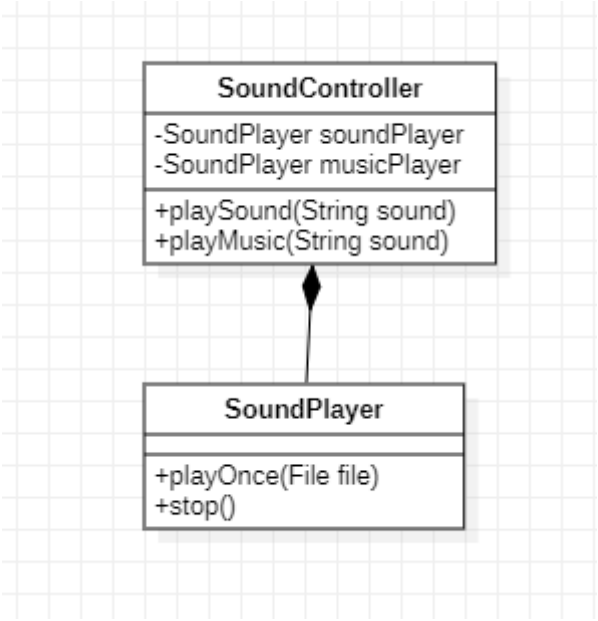
Different actions encapsulate different behaviors that can be executed.

It manages a collection of strategies (FreeColAction objects) and allows invoking them based on certain conditions or inputs.

3. **Facade Pattern** (src/net/sf/freecol/client/control/SoundController.java)

```
8 usages  Stian Grenborgen +4
public class SoundController {

    3 usages
    private static final Logger logger = Logger.getLogger(SoundController.class.getName());

    /** The internal sound player for sound effects. */
    7 usages
    private SoundPlayer soundPlayer;

    /** The internal sound player for music. */
    4 usages
    private SoundPlayer musicPlayer;
```

The **SoundController** class simplifies the interface for playing sounds and music by providing methods like **playSound** and **playMusic**. It hides the complexities of the **SoundPlayer** class and the underlying audio system from the rest of the application - a key aspect of the Facade Pattern. Clients of SoundController don't need to deal with mixers, audio files, or sound resources directly.



Review(s): Liliane Correia Review

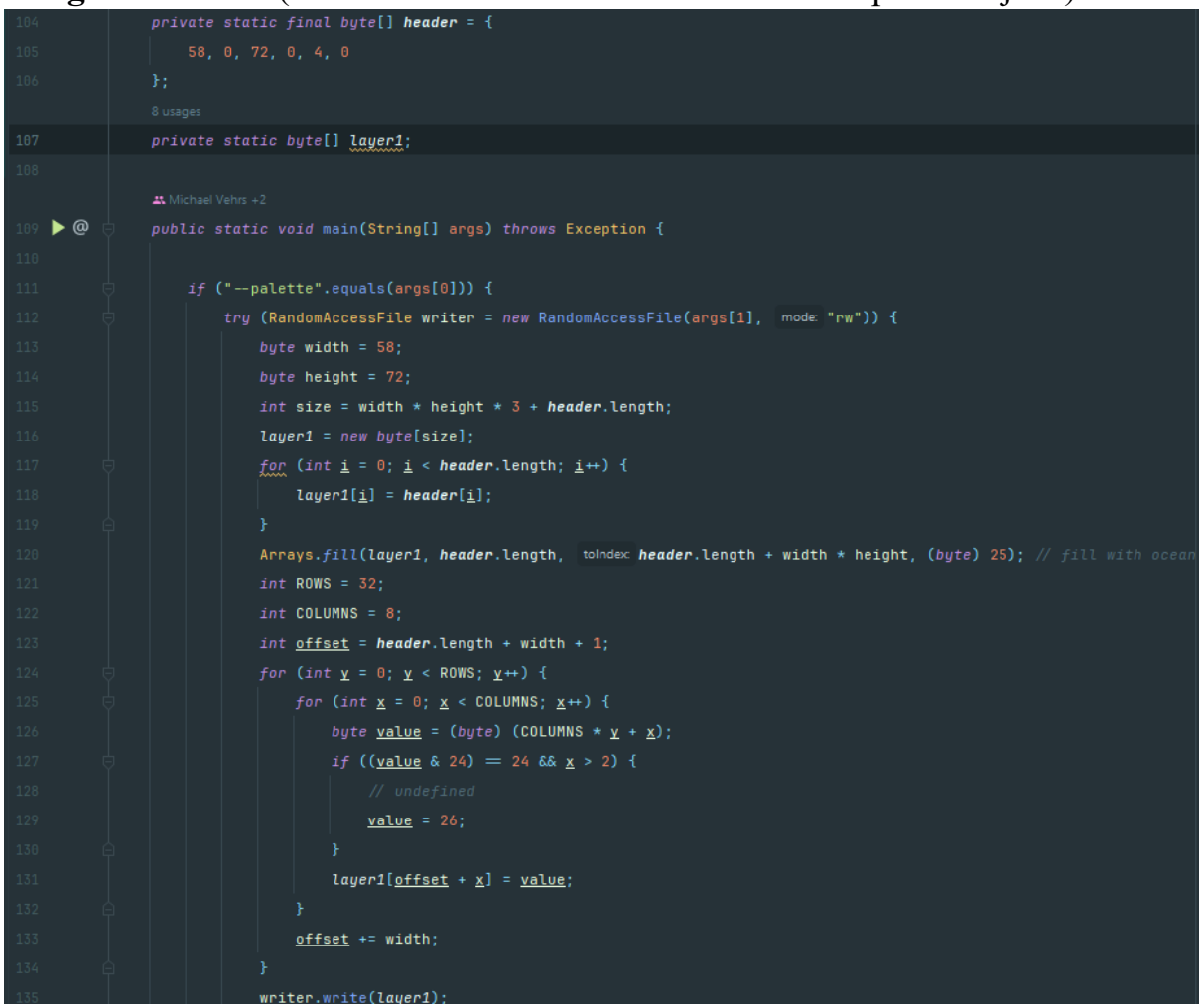
- The report includes code snippets and UML diagrams, making it easier to comprehend the design pattern being demonstrated

- It might be beneficial to double-check some of the patterns to ensure accuracy. FOR example: one of the pattern appears to be more closely related to Factory Design pattern than the Command pattern.

- Nonetheless, the explanations of the patterns are clear and comprehensible

Code Smells:

1. **Magic Numbers** (src/net/sf/freecol/tools/ColonizationMapReader.java)



```
104     private static final byte[] header = {
105         58, 0, 72, 0, 4, 0
106     };
        8 usages
107     private static byte[] layer1;
108
        Michael Vehrs +2
109     public static void main(String[] args) throws Exception {
110
111         if ("--palette".equals(args[0])) {
112             try (RandomAccessFile writer = new RandomAccessFile(args[1],  mode: "rw")) {
113                 byte width = 58;
114                 byte height = 72;
115                 int size = width * height * 3 + header.length;
116                 layer1 = new byte[size];
117                 for (int i = 0; i < header.length; i++) {
118                     layer1[i] = header[i];
119                 }
120                 Arrays.fill(layer1, header.length,  toIndex: header.length + width * height, (byte) 25); // fill with ocean
121                 int ROWS = 32;
122                 int COLUMNS = 8;
123                 int offset = header.length + width + 1;
124                 for (int y = 0; y < ROWS; y++) {
125                     for (int x = 0; x < COLUMNS; x++) {
126                         byte value = (byte) (COLUMNS * y + x);
127                         if ((value & 24) == 24 && x > 2) {
128                             // undefined
129                             value = 26;
130                         }
131                         layer1[offset + x] = value;
132                     }
133                     offset += width;
134                 }
135                 writer.write(layer1);
```

There are a lot of "magic numbers" in the code, like 58, 72, 3, 26, 32, 8, etc. These numbers are used without context, making it difficult to understand their purpose. They should be replaced with named constants.

2. **Long Method** (src/net/sf/freecol/tools/ColonizationMapReader.java)

```
public static void main(String[] args) throws Exception {

    if ("--palette".equals(args[0])) {
        try (RandomAccessFile writer = new RandomAccessFile(args[1], mode: "rw")) {
            byte width = 58;
            byte height = 72;
            int size = width * height * 3 + header.length;
            layer1 = new byte[size];
            for (int i = 0; i < header.length; i++) {
                layer1[i] = header[i];
            }
            Arrays.fill(layer1, header.length, toIndex: header.length + width * height, (byte) 25); // fill with ocea
            int ROWS = 32;
            int COLUMNS = 8;
            int offset = header.length + width + 1;
            for (int y = 0; y < ROWS; y++) {
                for (int x = 0; x < COLUMNS; x++) {
                    byte value = (byte) (COLUMNS * y + x);
                    if ((value & 24) == 24 && x > 2) {
                        // undefined
                        value = 26;
                    }
                    layer1[offset + x] = value;
                }
                offset += width;
            }
            writer.write(layer1);
        }
    } else {
        try (RandomAccessFile reader = new RandomAccessFile(args[0], mode: "r")) {
            try {
                reader.readFully(header);
            } catch (EOFException ee) {
                System.err.println("Unable to read header of " + args[0]
                    + ": " + ee);
                System.exit( status: 1);
            }
            System.out.println(String.format("Map width:  %02d", (int) header[WIDTH]));
            System.out.println(String.format("Map height: %02d", (int) header[HEIGHT]));

            int size = header[WIDTH] * header[HEIGHT];
            layer1 = new byte[size];
            try {
                reader.readFully(layer1);
            } catch (EOFException ee) {
                System.err.println("Unable to read data of " + args[0]
                    + ": " + ee);
                System.exit( status: 1);
            }
```

The main method is quite long and does multiple things: it handles both reading and writing of map files, and within those operations, it includes low-level operations such as file handling and byte manipulation. This method could be broken down into smaller, more focused methods.

3. **Data Clump** (src/net/sf/freecol/tools/ColonizationMapReader.java)

```
private static final char[] tiletypes = {
    't', // 0x00 tundra
    'd', // 0x01 desert
    'p', // 0x02 plains
    'r', // 0x03 prairie
    'g', // 0x04 grassland
    'v', // 0x05 savannah
    'm', // 0x06 marsh
    's', // 0x07 swamp

    'B', // 0x08 boreal (tundra with forest)
    'S', // 0x09 scrub (desert with forest)
    'M', // 0x0a mixed (plains with forest)
    'L', // 0x0b broadleaf (prairie with forest)
    'C', // 0x0c conifer (grassland with forest)
    'T', // 0x0d tropical (savannah with forest)
    'W', // 0x0e wetland (marsh with forest)
    'R', // 0x0f rain (swamp with forest)

    'B', // 0x10 boreal (tundra with forest)
    'S', // 0x11 scrub (desert with forest)
    'M', // 0x12 mixed (plains with forest)
    'L', // 0x13 broadleaf (prairie with forest)
    'C', // 0x14 conifer (grassland with forest)
    'T', // 0x15 tropical (savannah with forest)
    'W', // 0x16 wetland (marsh with forest)
    'R', // 0x17 rain (swamp with forest)
```

The repetition of characters in the *tiletypes* array is a form of code smell, specifically a "Data Clump," where similar data is grouped together without a clear purpose, which can lead to

confusion and potential errors. The repeated characters from 0x08 to 0x0f and from 0x10 to 0x17 indicate different tile types, but the characters used to represent them are the same.

Review(s): Review (08/11/2023 - Diogo Lemos):

The first pattern seems to be correctly identified. It is a good example of Magic Numbers, since it does compromise the readability of the method.

The second pattern seems to be correctly identified and the explanation behind it seems very accurate.

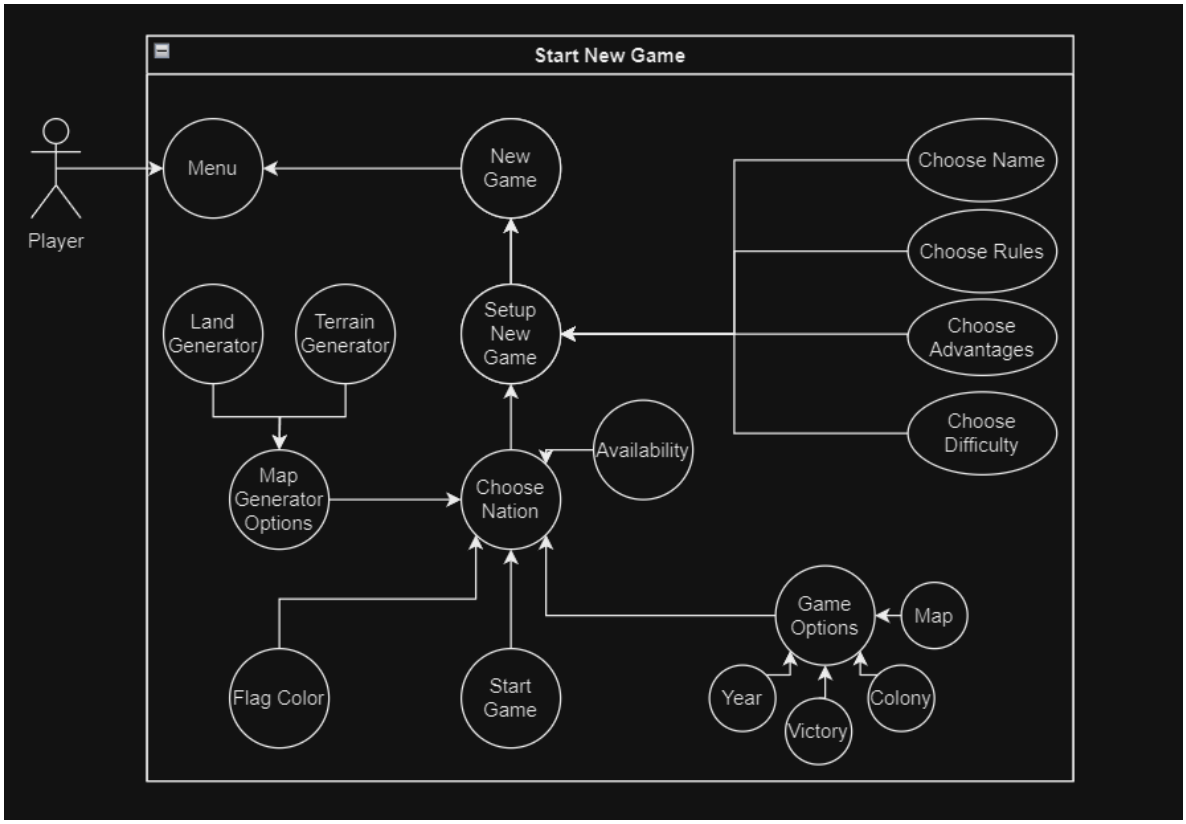
It is true that it is a bad practice to keep that many characters grouped together without much context other than comments, especially repeated ones, I don’t believe Data Clump to be the most accurate Code Pattern in this situation, since it usually applies to a group of data that would rather be together inside its own class to be used as a single instance.

Diogo Lemos - 56837

Use Case Diagram:

“Game Setup”

USE CASE



Name: Start New Game

Description: The Player wants to start a fresh new game.

Main Actor: Player

Secondary Actors: None

Pre-condition: None

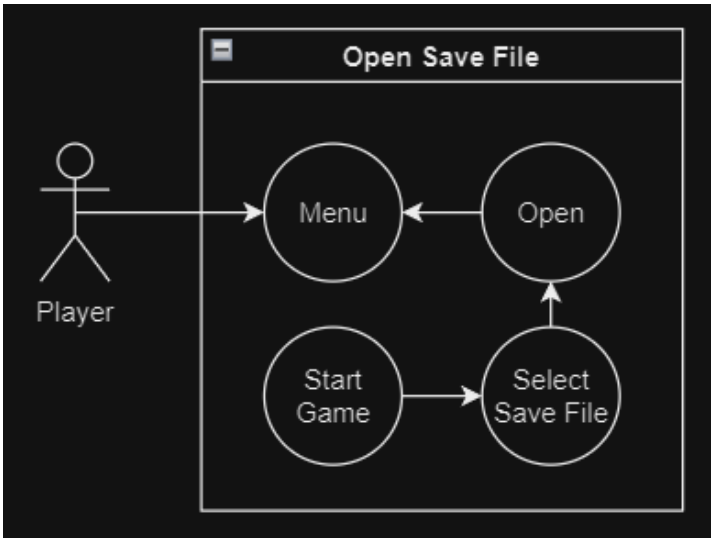
Main Flow:

1. Reach FreeCol’s Main Menu.
2. Select *New Game* option.
3. Choose *Player Name*.
4. Select *Setup New Game* option.
5. Choose the intended Nation.
6. Select the *Start Game* option.

Post-Condition: None

Alternative Flows: Player chooses to change other features.

USE CASE



Name: Open Save File

Description: The Player wants to load a save file.

Main Actor: Player

Secondary Actors: None

Pre-condition: There must exist at least a save file.

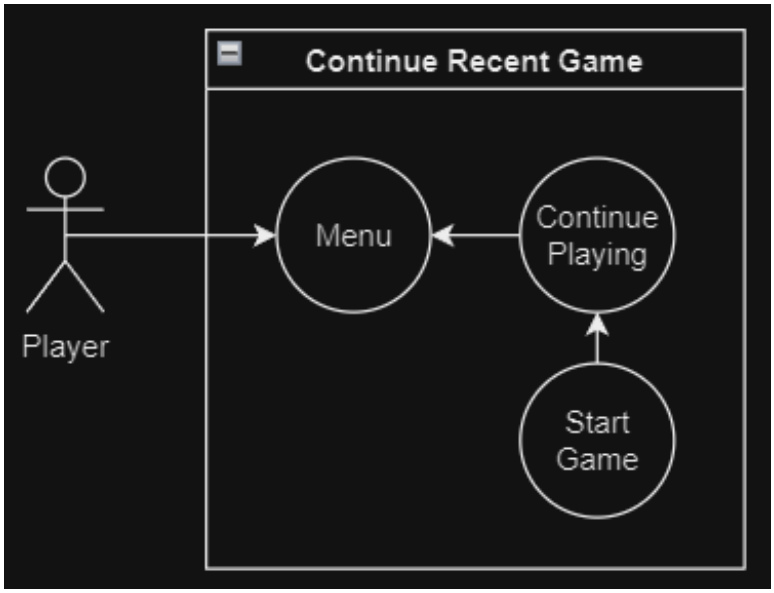
Main Flow:

1. Reach FreeCol's Main Menu.
2. Select *Open* option.
3. Select the intended save file.
4. Select the *Ok* option.

Post-Condition: None

Alternative Flows: None.

USE CASE



Name: Open Save File

Description: The Player wants to continue a recent save file.

Main Actor: Player

Secondary Actors: None

Pre-condition: The Player must have played the game before.

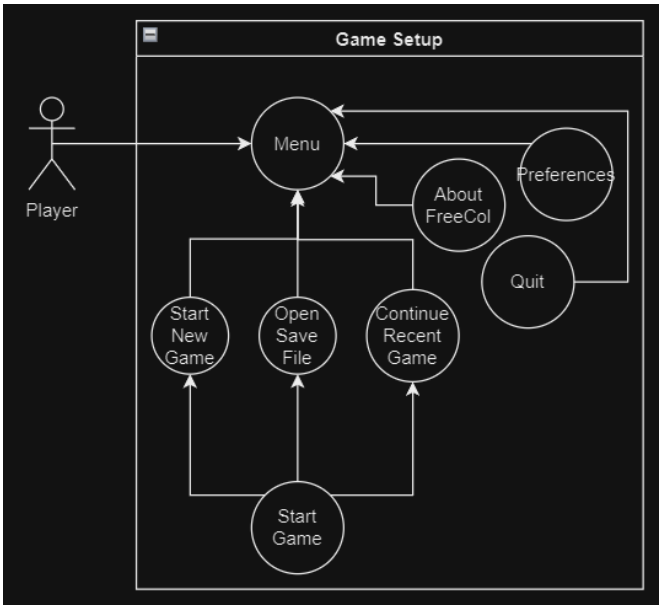
Main Flow:

1. Reach FreeCol's Main Menu.
2. Select *Continue Playing* option.
3. The game will start.

Post-Condition: None

Alternative Flows: None.

USE CASE



Name: Game Setup

Description: The Player wants to setup and start a game.

Main Actor: Player

Secondary Actors: None

Main Flow:

1. Reach FreeCol's Main Menu.
2. Select either *Continue Playing,* or *Open Save File* or *Start New Game* option.
3. Start the game.

Post-Condition: None

Alternative Flows:

- The Player accesses the *Preferences* option.
- The Player accesses the *About FreeCol* option.
- The player accesses the *Quit* option.

Review(s): REVIEWER: José Trigueiro 58119 -

I'm not sure if the arrows direction is correct. I would need help from another person to check on it.

The Doc has some information that was not ask by the teachers but it's a nice addition, like the pre/post condition and main/alternative flow.

I like how you divided the Use Cases in different subjects.

Overall, it seems to be ok.

Code Base Metrics:

"MOOD Metrics"

Understanding the MOOD Metrics

The **MOOD (Metrics for Object Oriented Design)** metrics set includes 6 metrics:

1. **MHF (Method Hiding Factor) –** MHF measures how variables are encapsulated in a class. It represents the average amount of hiding among all classes in the system.
   A private method is fully hidden. Basically, hiding decreases in the following order: Protected, Friend, Protected Friend, Public.
   It is calculated using the following formula:

   MHF = 1 – MethodsVisible
   MethodsVisible = sum (MV) / (C − 1) / Number of methods

   MV = number of other classes where method is visible
   C = number of classes

   If all methods are private, then MHF=100%. That would imply very little functionality as methods wouldn't be able to be reused by other classes.

   If all methods are public, MHF=0%. That would imply insufficiently abstracted implementation. A large proportion of methods would be unprotected and the probability of errors high.

   So, the ideal value is somewhere in between.

2. **AHF (Attribute Hiding Factor) –** AHF works similarly to MHF, but it is applied to attributes instead.
   It is calculated using a similar formula as well:

   AHF = 1 – AttributesVisible
   AttributesVisible = sum (AV) / (C − 1) / Number of attributes

   AV = number of other classes where attribute is visible
   C = number of classes

   If all attributes are private, then AHF=100% and that would be the ideal value.
   If all attributes are public, AHF=0%. Very low values of AHF should trigger attention.

3. **MIF (Method Inheritance Factor) –** MIF measures the average between inherited and total methods in classes.
   It is calculated using the following formula:

   MIF = inherited methods / total methods available in classes

   A class that inherits lots of methods from its ancestor classes contributes to a high MIF. A child class that redefines its ancestors' methods and adds new ones contributes to a lower MIF. An independent class that does not inherit and has no children contributes to a lower MIF.

4. **AIF (Attribute Inheritance Factor) –** AIF works similarly to MIF, but it is applied to attributes instead. It is calculated using a similar formula as well:

   AIF = inherited attributes / total attributes available in classes

   The ideal value for AIF would be 0% since all attributes should be private.

5. **PF (Polymorphism Factor) –** PF measures the degree of method overriding in the class inheritance tree. It equals the number of actual method overrides divided by the maximum number of possible method overrides.
   It is calculated using the following formula:

   PF = overrides / sum for each class (new methods * descendants)

   PF varies between 0% and 100%. As mentioned above, when PF=100%, all methods are overridden in all derived classes. A PF value of 0% may indicate one of the following cases:

   - project uses no classes or inheritance.
   - project uses no polymorphism.
   - full class hierarchies have not been analyzed (child Overrides unknown)

6. **CF (Coupling Factor) -** measures the actual couplings among classes in relation to the maximum number of possible couplings.
   It is calculated using the following formula:

   CF = Actual couplings / Maximum possible couplings

   Class A is *coupled to* class B if A calls methods or accesses variables of B. In turn, B is coupled to A only if B calls methods or accesses variables of A. B is not coupled to A if there is no call/access from B to A.
   If no classes are coupled, CF = 0%. If all classes are coupled to all other classes, CF=100%.
   Couplings due to the use of the Inherits statement are not included in CF.

   Coupling relations increase complexity, reduce encapsulation and potential reuse, and limit understandability and maintainability. Very high values of CF should be avoided.
   However, classes must cooperate somehow, and CF is expected to be lower bounded.

MOOD Results

Project metrics

project	AHF	AIF	CF	MHF	MIF	PF
project	69,05%	46,26%	3,34%	26,03%	72,78%	7,06%

Figure 1. Project Metrics using IntelliJ's extension - MetricsReloaded.

Analyzing the results, we can assume certain aspects about the project's code:

- **MHF = 26,03% -** The value seems reasonable. We can assume that most methods must be private or maintain an acceptable degree of encapsulation.
- **AHF = 69,05% -** The target value is 0%, so the presented result is quite above the intended. It is alarming that most attributes are visible to other classes other than their own. Code smells associated with this behavior might be *Feature Envy* or *Inappropriate Intimacy.*
- **MIF = 72,78% -** The value is acceptable, although quite high. We can assume that many methods are being inherited by subclasses, which will raise the project's overall complexity.
- **AIF = 46,46% -** Although the value is acceptable it is still quite high. We can take into consideration that while ideally having AHF at 0%, AIF should also tend to lower values to keep encapsulation in check.
- **PF = 7,06% -** We can expect a decent usage of overridden methods. A higher value of such methods could increase the code's clarity but would also increase the project's overall complexity.
- **CF = 3,34% -** The value seems reasonable since it should be a lower end number. Higher coupling induces a bigger complexity and lower encapsulation which compromise other factors such as understandability and maintainability.

References

MOOD Metrics - https://www.aivosto.com/project/help/pm-oo-mood.html

(consulted in 07/11/2023)

Review(s): Liliane Correia Review:

- A very detailed and well-presented report

- Analysed each metric percentage in the MOOD metrics, providing an overall understanding of the system's complexity.

- Offered improvement tips, especially regarding AHF, where it could be beneficial to reduce attributes visibility

GOF Patterns:

1. Template Method (src/net/sf/freecol/client/gui/action)

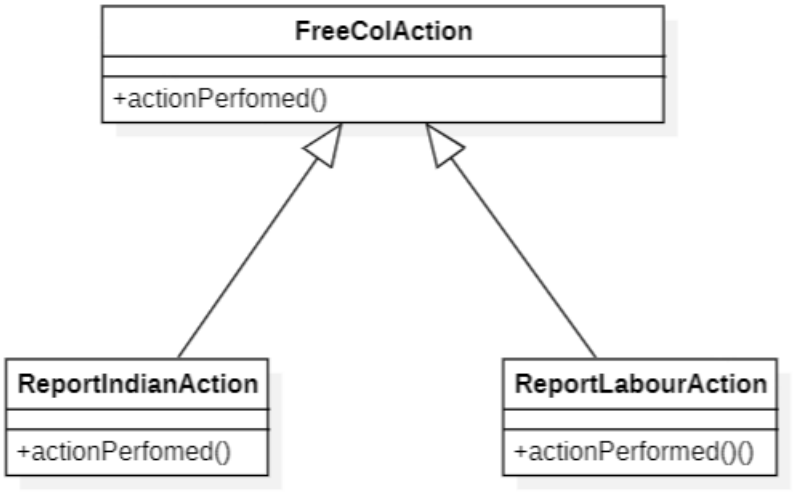


Figure 2. StarUML Diagram of the Template Method Pattern

- Both ReportIndianAction and ReportLabourAction share similar methods, for example, actionPerfomed(). FreeColAction is then used as a basis for both subclasses.

2. Iterator Pattern (src/net/sf/freecol/common/model/UnitIterator.java)

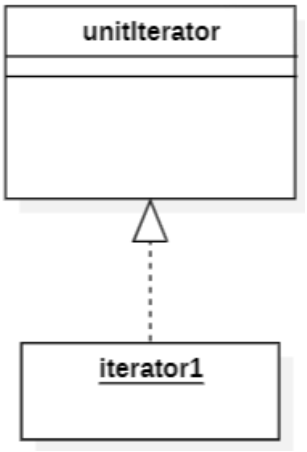


Figure 3. unitIterator

The unitIterator class enables the possibility of iterating through the Unit objects without compromising their implementation.

3. Singleton

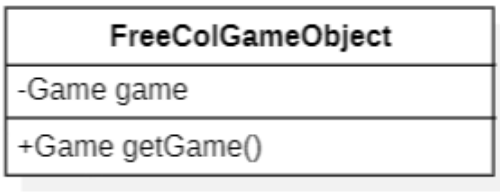


Figure 4. FreeColGameObject class's diagram

- FreeColGameObject's class has only one instance of the attribute game, which is hidden from all other outside classes. The only way to access it is through the public method *getGame* found in the same class.

Review(s): Reviewed by Gustavo Silva 59472 (08/11/2023):

- Everything seems to be alright.

Code Smells:

1. Long Method (src/net/sf/freecol/server/ai/ColonyPlan.java)

```
// Exempt defence and export from the level check.
if (type.hasModifier(Modifier.DEFENCE)) {
    double factor = 1.0;
    if ("conquest".equals(advantage)) factor = 1.1;
    prioritize(type, weight: FORTIFY_WEIGHT * factor,
        1.0/*FIXME: 0 if FF underway*/);
}
if (type.hasAbility(Ability.EXPORT)) {
    double factor = 1.0;
    if ("trade".equals(advantage)) factor = 1.1;
    prioritize(type, weight: EXPORT_WEIGHT * factor,
        1.0/*FIXME: weigh production v transport*/);
}

// Skip later stage buildings for smaller settlements.
if (type.getLevel() > maxLevel) continue;
```

```
// Scale docks by the improvement available to the food supply.
if (type.hasAbility(Ability.PRODUCE_IN_WATER)) {
   double factor = 0.0;
   if (!colony.hasAbility(Ability.PRODUCE_IN_WATER)
       && colony.getTile().isShore()) {
       int landFood = 0, seaFood = 0;
       for (Tile t : transform(colony.getTile().getSurroundingTiles( rangeMin: 1, rangeMax: 1),
               t2 -> (t2.getOwningSettlement() == colony || player.canClaimForSettlement(t2)))) {
           for (AbstractGoods ag : t.getSortedPotential()) {
               if (ag.isFoodType()) {
                   if (t.isLand()) {
                       landFood += ag.getAmount();
                   } else {
                       seaFood += ag.getAmount();
                   }
               }
           }
       }
       factor = (seaFood + landFood == 0) ? 0.0
           : seaFood / (double)(seaFood + landFood);
   }
   prioritize(type, FISH_WEIGHT, factor);
}
```

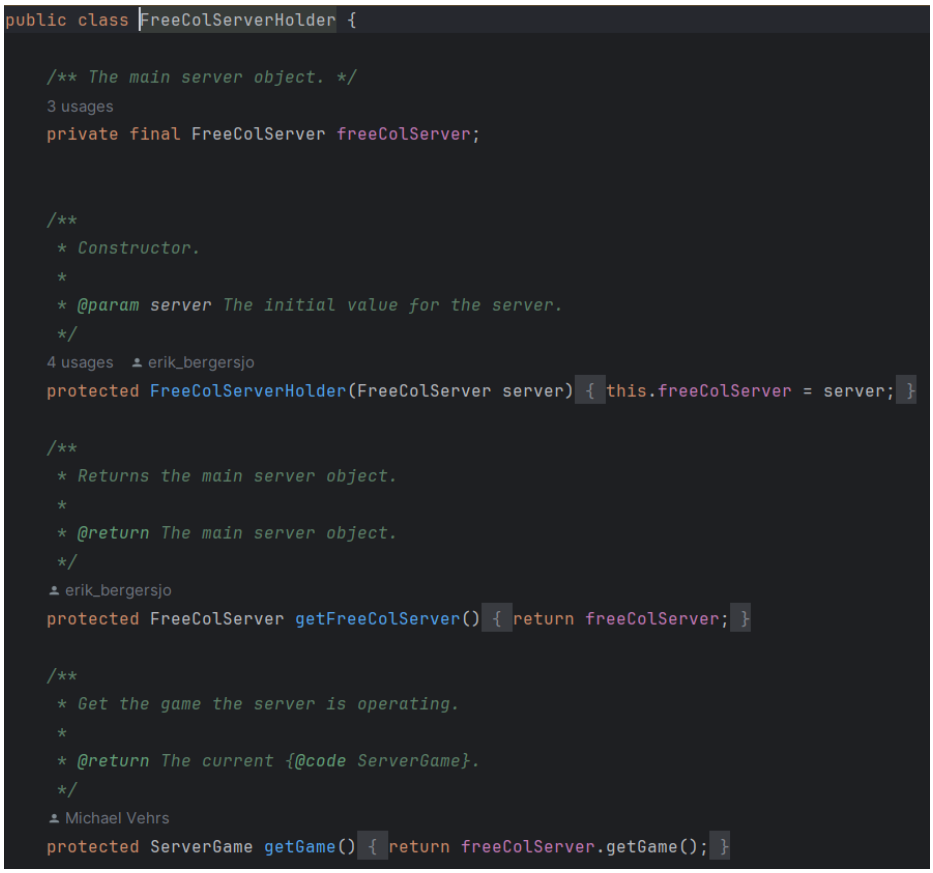
```
if (type.hasAbility(Ability.BUILD)) {
    double factor = ("building".equals(advantage)) ? 1.1 : 1.0;
    double support = (any(type.getAbilities(Ability.BUILD),
                          Ability::hasScope)) ? 0.1 : 1.0;
    prioritize(type, weight: BUILDING_WEIGHT * factor,
        support/*FIXME: need for the thing now buildable*/);
}

if (type.hasAbility(Ability.TEACH)) {
    prioritize(type, TEACH_WEIGHT,
        1.0/*FIXME: #students, #specialists here, #wanted*/);
}

if (type.hasAbility(Ability.REPAIR_UNITS)) {
    double factor = 1.0;
    if ("naval".equals(advantage)) factor = 1.1;
    prioritize(type, weight: REPAIR_WEIGHT * factor,
        1.0/*FIXME: #units-to-repair, has-Europe etc*/);
}
```

```
GoodsType output = type.getProducedGoodsType();
if (output != null) {
    if (!prioritizeProduction(type, output)) {
        // Allow failure if this building can not build.
        expectFail = true;
    }
} else {
    for (GoodsType g : spec().getGoodsTypeList()) {
        if (type.hasModifier(g.getId())) {
            if (!prioritizeProduction(type, g)) {
                expectFail = true;
            }
        }
    }
    // Hacks.  No good way to make this really generic.
    if (type.hasModifier(Modifier.WAREHOUSE_STORAGE)) {
        double factor = 1.0;
        if ("trade".equals(advantage)) factor = 1.1;
        prioritize(type, weight: STORAGE_WEIGHT * factor,
            1.0/*FIXME: amount of goods*/);
    }
```

Figure 1. ColonyPlan's if statements.

- This method has various substeps with multiple *if* statements and *for* cycles making the method extremely long and hard to comprehend. A possible solution could be to turn some of it's *if* statements into other methods to enhance the method's readability and lower it's complexity.

2. Data Class (src/net/sf/freecol/server/control/FreeColServerHolder.java)



```
public class FreeColServerHolder {

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


    /**
     * Constructor.
     *
     * @param server The initial value for the server.
     */
    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.
     *
     * @return The current {@code ServerGame}.
     */
    Michael Vehrs
    protected ServerGame getGame() { return freeColServer.getGame(); }
```

Figure 2. FreeColServerHolder class' methods.

- This class only has getter methods. It serves no other purpose other than holding the server object. A solution would be to give it more meaning, for example by adding more complex methods apart from the simple getters.

3. Large Class (src/net/sf/freecol/client/control/InGameController.java)

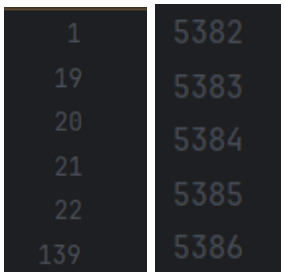


Figure 3. InGameController class's first and last lines of code

- This class has over five thousand lines of code. It is very hard to read it thoroughly. A solution would be to split it into multiple classes to better enhance it's readability and lower it's complexity.

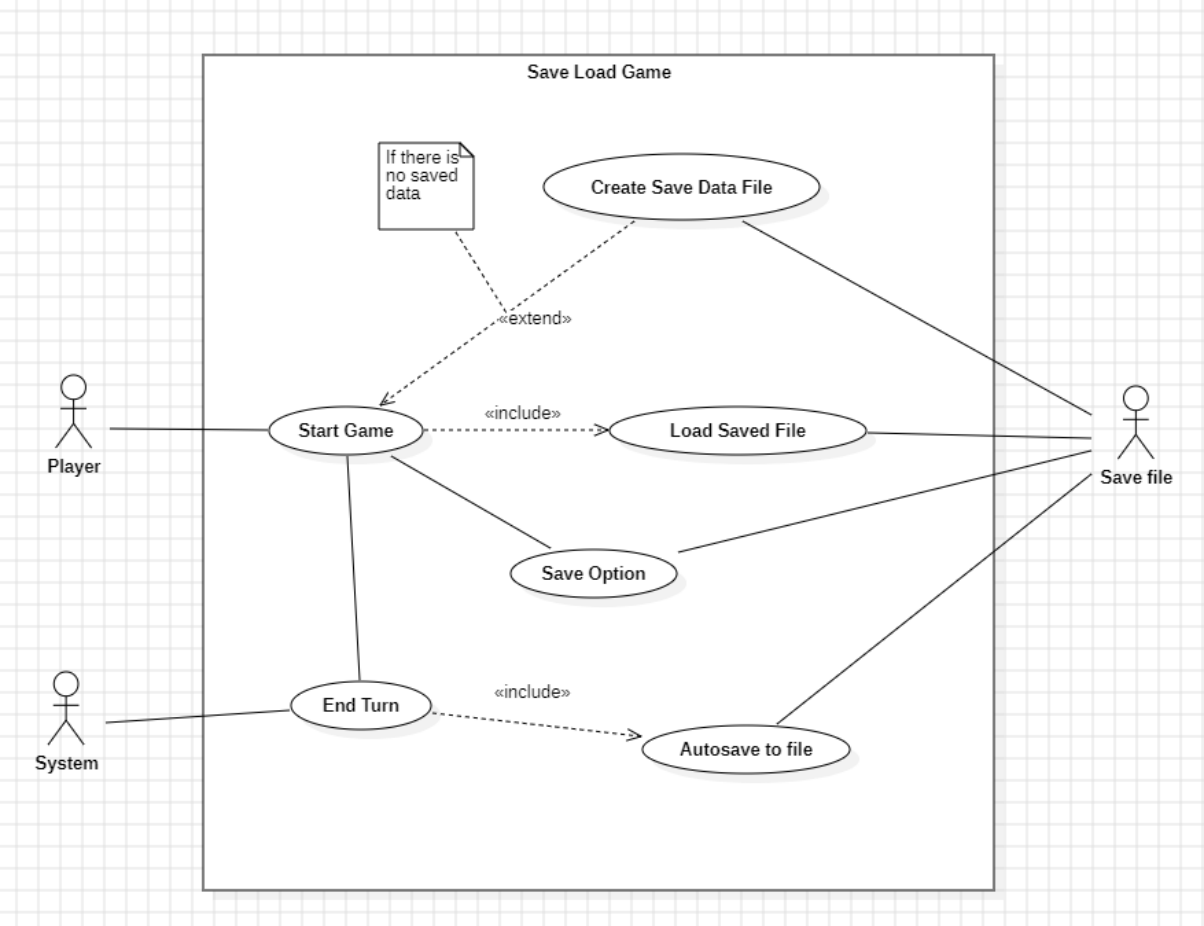
Review(s): Reviewed by Gustavo Silva 59472 (08/11/2023):
- All code smells (Long Method, Data Class and Large Class) appear to be correctly identified.

Reviewed by José Pereira –

- The first proposed smell is indeed a Long Method since it could clearly be broken down into multiple auxiliary methods to prevent such clutter of information.
- The second proposed smell is the classic Data Class code smell, with simple "getters" and "setters", however this is not necessarily a bad thing.
- Well, the last proposed smell is hard to analyze but if we consider the number of lines from the screenshot presented it is indeed a very large class that surely could be broken down into simpler parts, and it could imply bad code management as well.

Gustavo Silva - 59472

Use Case Diagram:



- Use Case Name: Save or Load game file.
- Description: When starting a game, the game is loaded from the save file if it exists, otherwise a new save data file is created. After starting a game, the player has the option to turn on autosave feature which saves the game automatically after a few turns. The player can also manually save the game.
- Actors: Player, System

Review(s): Liliane Correia Review Comments:

- The UML are well-designed and highly detailed

- The descriptions are straightforward and concise

Code Base Metrics:

CHIDAMBER-KEMERER METRICS

Chidamber-Kemerer metrics, often referred to as CK metrics, are a set of software complexity metrics designed to assess various aspects of object-oriented software code. These metrics focus on the whole class itself instead of parts of it.

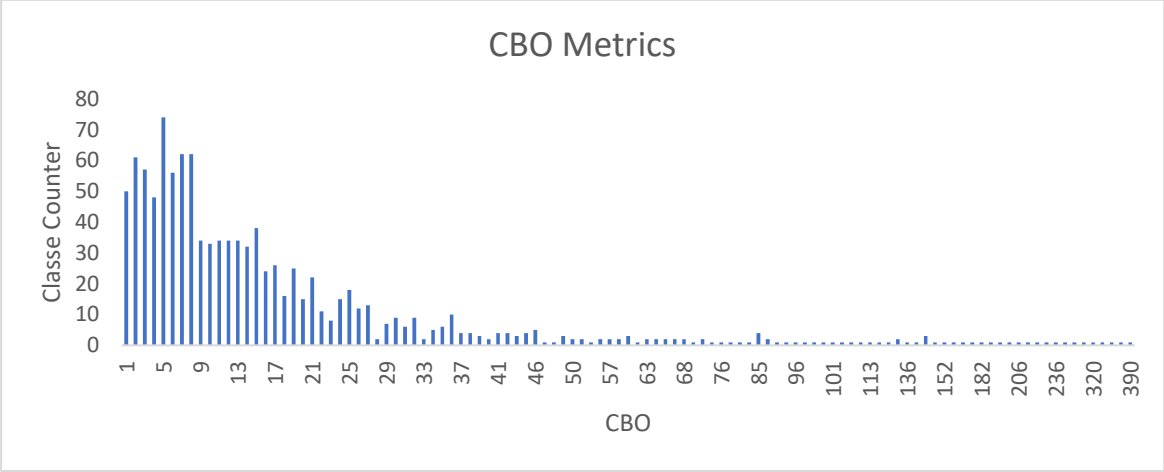
class	CBO	DIT	LCOM	NOC	RFC	WMC
Total						**26 666**
Average	20,99	3,53	2,65	0,64	39,82	24,49

I. CBO – Coupling Between Objects

CBO counts how many other classes a given class is directly linked to. A class with a high CBO value depends on many other classes, which means it has a higher degree of coupling.

A. Data

Interfaces are not considered.



B. Analysis

The program indicates a high level of coupling, with the average being 21. That means the code might be difficult to understand, teste, and modify. The most extreme case for this metric are:

class	CBO
net.sf.freecol.common.model.FreeColObject	390
net.sf.freecol.common.model.Player	382

One being an abstract class that represents the FreeCol root class and the other being the class that represents the Player itself. Values are expected to be high on these 2 classes since both must interact with many aspects of the program.

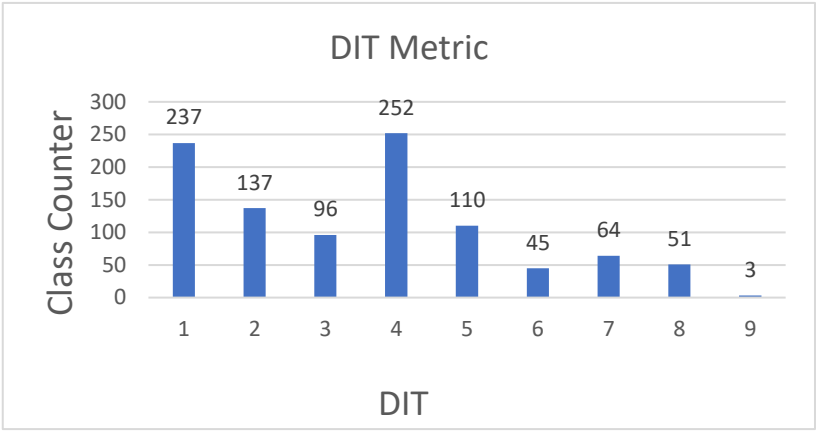
II. DIT – Depth of Inheritance Tree

DIT determines the depth of the inheritance hierarchy for a class. It computes how many ancestor classes a class has. A class with a high DIT value has a deeper class hierarchy, which can increase the code complexity.

A. Data

Interfaces and enumerators are not considered.

DIT	Class Counter
1	237
2	137
3	96
4	252
5	110
6	45
7	64
8	51
9	3



B. Analysis

The average DIT for this project is 3.5. Which could be considered a good value given the project scope.

The 3 classes with the highest DIT are:

class	DIT
net.sf.freecol.client.gui.panel.report.ReportCargoPanel	9
net.sf.freecol.client.gui.panel.report.ReportMilitaryPanel	9
net.sf.freecol.client.gui.panel.report.ReportNavalPanel	9

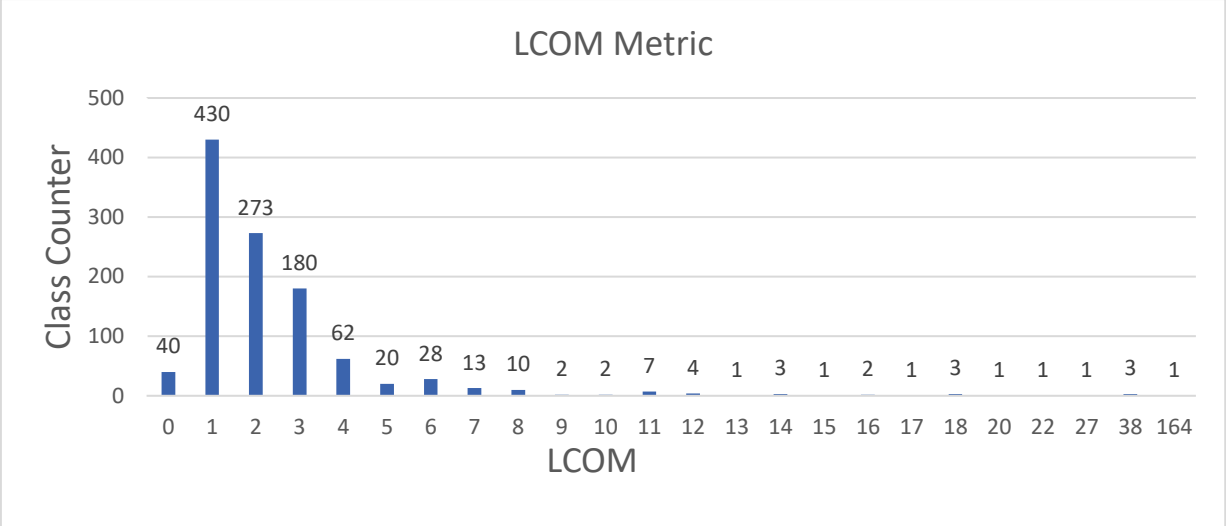
All 3 classes are related a GUI that displays reports to the player.

III. LCOM – Lack of Cohesion in Methods

LCOM evaluates the cohesion within a class. It identifies how many groups of methods do not share attributes. A class with a higher LCOM value has lower cohesion, which can make the code more difficult to maintain.

A. Data

Interfaces are not considered.



B. Analysis

The metric indicates that the program has a good level of cohesion (average of 2.65), that is an excellent characteristic given the open-source nature of the code. That means that the code is easier to maintain and reuse.

The extreme case of lack of cohesion is in the following class:

class	LCOM
net.sf.freecol.client.gui.GUI	164

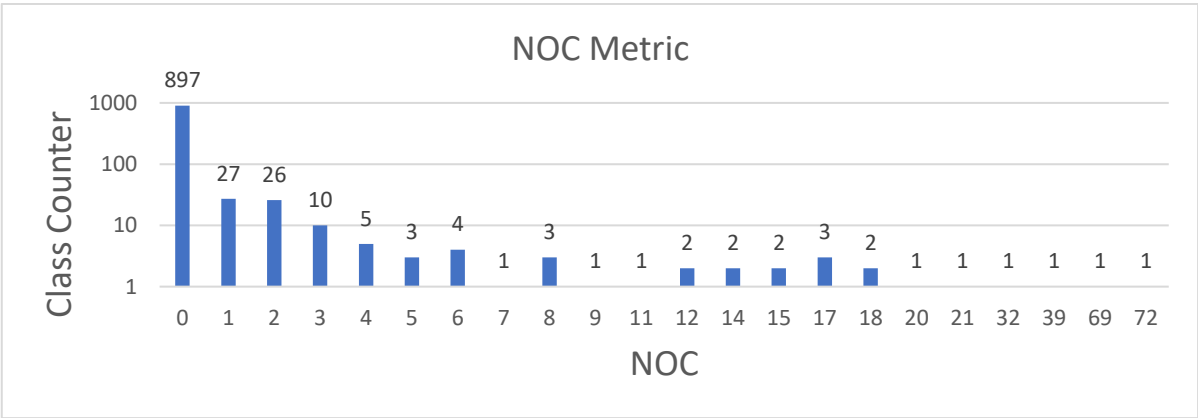
This class is the API and common reusable functionalities of the overall GUI. For that reason it is expected that the class would not have a high cohesion.

IV. NOC – Number of Children

NOC measures the number of immediate subclasses a class has. A class with a high NOC value has many child classes, which can affect its maintainability and complexity.

A. Data

Interfaces and enumerators are not considered.



B. Analysis

Most of the Classes does not have children, that means that the classes are likely to be less complex and serve a specific role or functionality within the software.

The most extreme case is found in the following class:

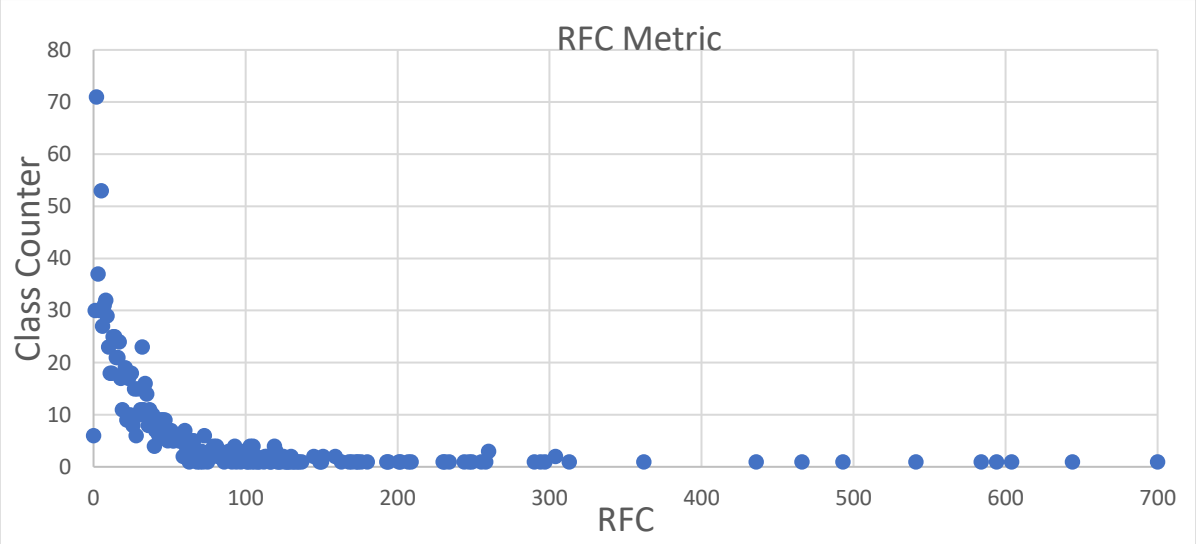
class	NOC
net.sf.freecol.common.networking.AttributeMessage	72

This class implements the basic functionalities of messages. Since the game has a lot of messages throughout the gameplay, its expected that this class would have a higher number of immediate subclasses.

V. RFC – Response For a Class

RFC estimates the total number of methods that can be invoked in response to a message to an object of a class. A class with a high RFC value responds to many different messages and can be harder to understand and maintain.

A. Data



B. Analysis

The average value for this metric is 40 RFC. The highest value is found in the following class:

class	RFC
net.sf.freecol.client.control.InGameController	700

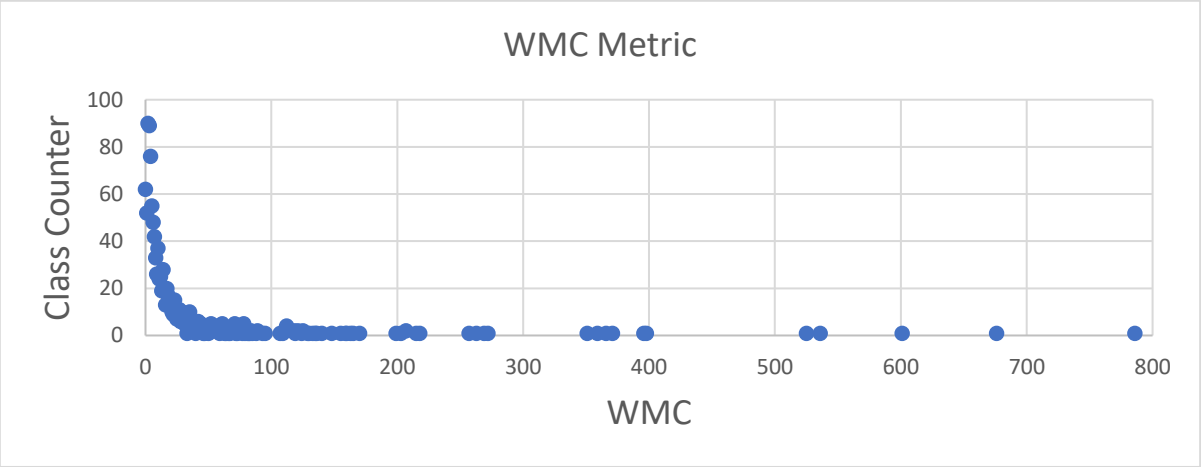
Being the game controller class, its expected that the value of RFC would be extremely high since it must respond to every single command in game.

VI. WMC – Weighted Methods per Class

WMC calculates the complexity of a class by counting the number of methods it contains. Methods with more complexity receive higher weight. A class with a high WMC value has complex functionality, which can make it more challenging to maintain.

A. Data

Interfaces are not considered.



B. Analysis

The average value of this metric is 24.5, which is not a high value. The class with the highest value of this metric is once again the Game Controller Class, also something expected because of the high number of controller inputs that the class must manage.

class	WMC
net.sf.freecol.client.control.InGameController	786

Review(s): José Trigueiro 58119:

The document seems to be well-structured and informative. It provides a detailed explanation of the Chidamber-Kemerer metrics, including Coupling Between Objects (CBO), Depth of Inheritance Tree (DIT), Lack of Cohesion in Methods (LCOM), Number of Children (NOC), Response for a Class (RFC), and Weighted Methods per Class (WMC). Each metric is explained with its definition, data considerations, and an analysis of its application.

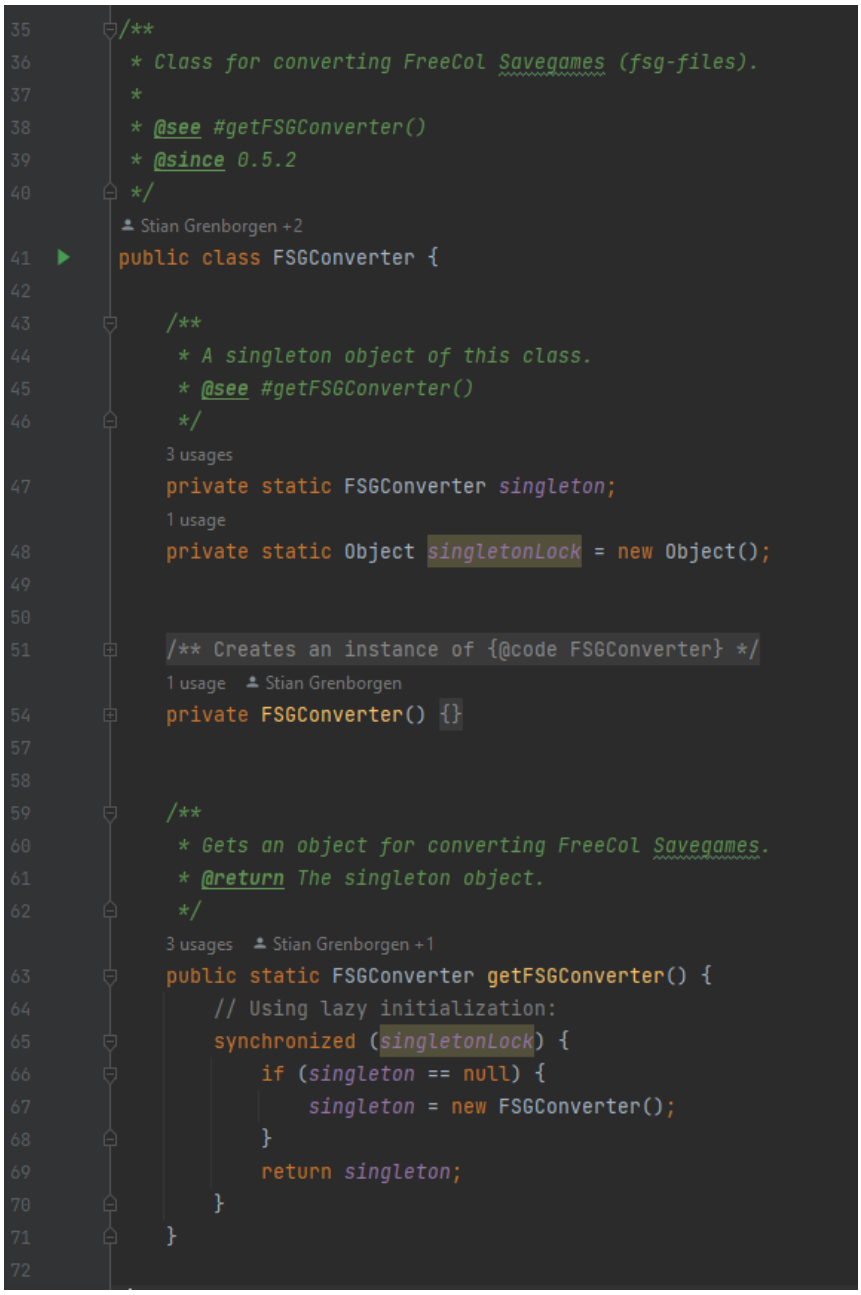
The graphs help a lot visualizing the data and it's a nice addition.

Overall, it seems to be ok, except the page 6 that has a minor formation problem with the text "Interface" leaving the word page.

GOF Patterns:

1. Singleton Pattern – FSGConverter.java (src/net/sf/freecol/tools/FSGConverter.java)

The FSGConverter class exemplifies the Singleton design pattern due to its embodiment of essential Singleton characteristics. Its private constructor, private static instance, lazy initialization, synchronized access, and global access point all align with the pattern's key principles.



```
/**
 * Class for converting FreeCol Savegames (fsg-files).
 *
 * @see #getFSGConverter()
 * @since 0.5.2
 */
Stian Grenborgen +2
public class FSGConverter {

    /**
     * A singleton object of this class.
     * @see #getFSGConverter()
     */
    3 usages
    private static FSGConverter singleton;
    1 usage
    private static Object singletonLock = new Object();


    /** Creates an instance of {@code FSGConverter} */
    1 usage  Stian Grenborgen
    private FSGConverter() {}


    /**
     * Gets an object for converting FreeCol Savegames.
     * @return The singleton object.
     */
    3 usages  Stian Grenborgen +1
    public static FSGConverter getFSGConverter() {
        // Using lazy initialization:
        synchronized (singletonLock) {
            if (singleton == null) {
                singleton = new FSGConverter();
            }
            return singleton;
        }
    }

```

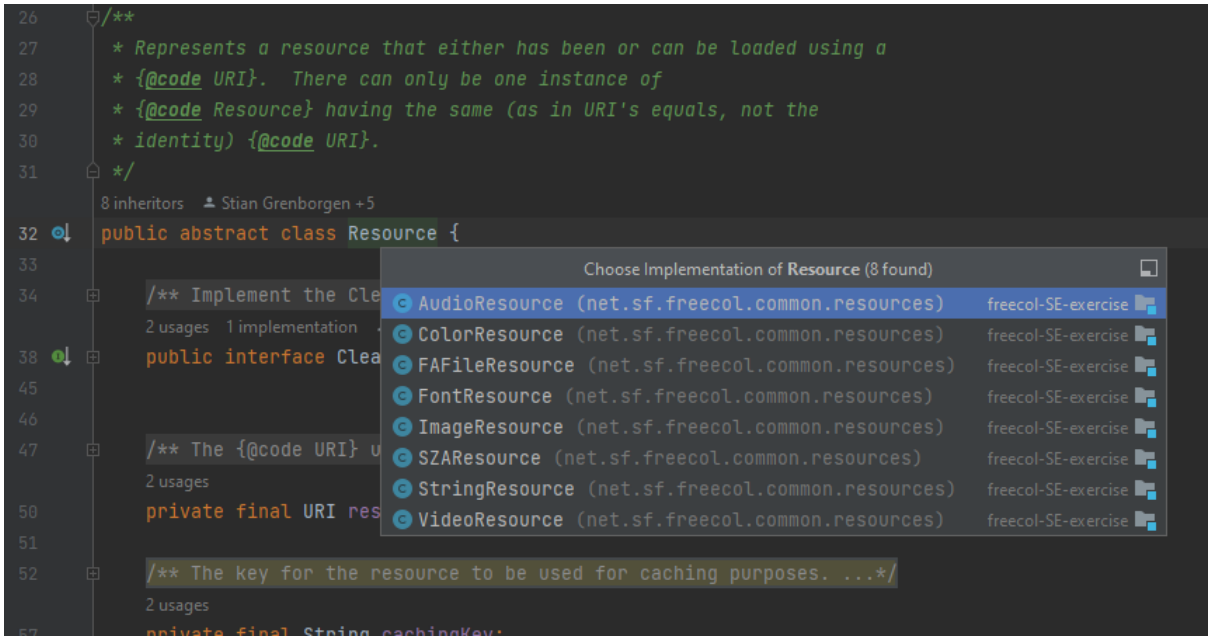
2. Factory Pattern – ResourceFactory.java (src/net/sf/freecol/common/resources/ResourceFactory.java)

The ResourceFactory class is a central factory that creates different Resource objects based on the given parameters. This way, the creation process is hidden, and the code is organized. This pattern also increases flexibility, hides resource creation, and simplifies resource management in the FreeCol project.

```
/**
 * Returns an instance of {@code Resource} with the
 * given {@code URI} as the parameter.
 *
 * @param key The key part of the resource mapping.
 * @param cachingKey The caching key.
 * @param uri The {@code URI} used when creating the instance.
 * @return The <code>Resource</code> if created.
 */
2 usages  Stian Grenborgen +4
public Resource createResource(String key, String cachingKey, URI ur
    final Resource r = resources.get(uri);
    if (r != null) {...}

    final String pathPart;
    if (uri.getPath() != null) {...} else if (uri.toString().indexOf

    try {
        final Resource resource;
        if ("urn".equals(uri.getScheme())) {...} else if (pathPart.e
            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;
```

3. **Template Pattern** – Resource.java (src/net/sf/freecol/common/resources/Resource.java)

The Resource Class is implemented by 8 other classes, as seen in the image. All the 8 implementations utilizes the common structure set by the Resource Class and make changes to the methods if needed. That's a clear use of the template pattern.



Review(s): Wilker martins 58535

It's worth noting that the design patterns utilized in the codebase appear to have been well chosen for their respective contexts.

The Singleton Pattern, as applied in FSGConverter.java, effectively embodies the essential characteristics of a Singleton, ensuring a single instance with private constructor, synchronized access, and global access point.

The Factory Pattern, exemplified by ResourceFactory.java, is a suitable choice for creating Resource objects based on parameters, offering a structured and flexible approach while simplifying resource management.

The Template Pattern, employed in Resource.java and its eight implementations, provides a clear and effective way to establish a common structure and make necessary method adjustments, promoting code reuse and consistency.

Overall, the use of these design patterns appears to be well-matched with the specific requirements and objectives of the respective classes.

Code Smells:

1. Long Method: rearrangeColony(LogBuilder lb) – src/net/sf/freecol/server/ai/AIColony.java

```
229     /**
230      * Rearranges the workers within this colony using the {@link ColonyPlan}.
231      *
232      * FIXME: Detect military threats and boost defence.
233      *
234      * @param lb A {@code LogBuilder} to log to.
235      * @return A set of {@code AIUnit}s that changed their work state.
236      */
        Mike Pope +4
237 @   public Set<AIUnit> rearrangeColony(LogBuilder lb) {...}
427
428     /**
```

This method has almost 200 lines of code. This could have been avoided if the method was fragmented into smaller methods that did a specific thing. For example, in the following part of the code, we can extract a method to “assignPioneers()” and another to “logChanges()”.

```
390
391         // Allocate pioneers if possible.
392         int tipSize = tileImprovementPlans.size();
393         if (tipSize > 0) {
394             List<Unit> pioneers
395                 = transform(tile.getUnits(), u -> u.getPioneerScore() >= 0,
396                             Function.<~>identity(), pioneerComparator);
397             for (Unit u : pioneers) {
398                 final AIUnit aiu = getAIUnit(u);
399                 if (aiu.tryPioneeringMission(lb)) {
400                     if (--tipSize <= 0) break;
401                 }
402             }
403         }
404
405         for (Unit u : tile.getUnitList()) {
406             final AIUnit aiu = getAIUnit(u);
407             if (!aiu.trySomeUsefulMission(colony, lb)) result.add(aiu);
408         }
409
410         // Log the changes.
411         build = colony.getCurrentlyBuilding();
412         String buildStr = (build != null) ? build.toString()
413             : ((build = colonyPlan.getBestBuildableType()) != null)
414             ? "unexpected-null(" + build + ")"
415             : "expected-null";
416         lb.add(", building ", buildStr, ", population ", colony.getUnitCount(),
417             ", rearrange ", nextRearrange, ".\n");
418         lb.add(aw.toString());
419         lb.shrink( delim: "\n");
420         for (UnitWas uw : was) lb.add("\n  ", uw);
421
422         // Set the next rearrangement turn.
423         rearrangeTurn = new Turn(turn + nextRearrange);
424
425         return result;
426     }
```

2. Long Class: Map.java - src/net/sf/freecol/common/model/Map.java

```
2850        /**
2851         * {@inheritDoc}
2852         */
             Mike Pope
2853        public String getXMLTagName() { return TAG; }
2854    }
2855
```

This class is extremely big. It might be a good option to compartmentalize the code into smaller and less responsible classes.

3. Magic Number: main( ) - src/net/sf/freecol/tools/ForestMaker.java

```
128  public static void main(String[] args) throws IOException {
129
130      if (args.length == 0) {
131          System.out.println("Usage: ForestMaker <directory>...");
132          System.out.println("Directory name should match a directory in");
133          System.out.println("    " + DESTDIR);
134          System.exit( status: 1);
135      }
136
137      String riverName = "data/rules/classic/resources/images/terrain/"
138              + "ocean/center0.png";
139      BufferedImage river = ImageIO.read(new File(riverName));
140      // grab a rectangle completely filled with water
141      river = river.getSubimage( x: 44,  y: 22,  w: 40,  h: 20);
142      Rectangle2D rectangle = new Rectangle( x: 0,  y: 0, river.getWidth(), river.getHeight());
143      TexturePaint texture = new TexturePaint(river, rectangle);
```

These numbers appears from nowhere making it not clear their purpose. Number like that should be defined as a constant and only after that utilized. That makes the code more readable and easier to understand.

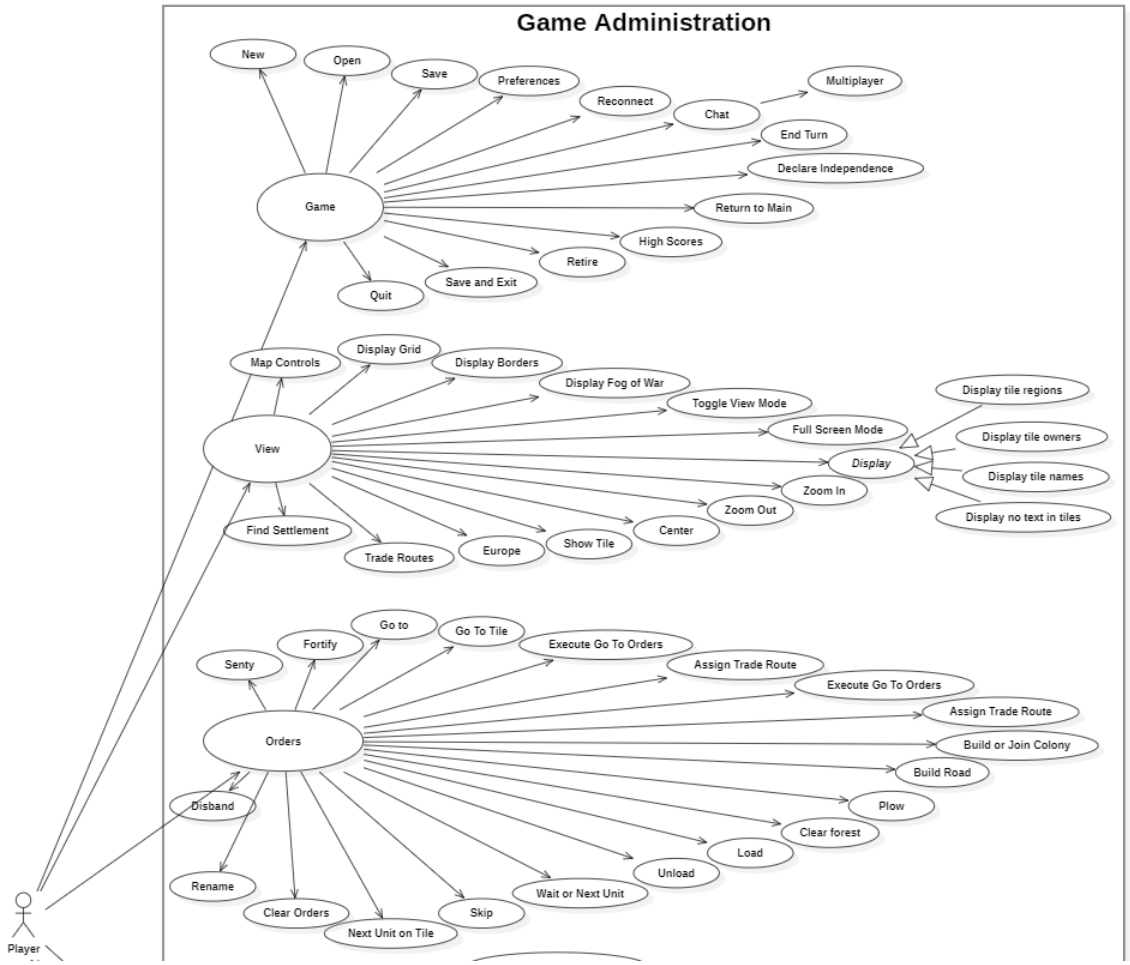
Review(s): Review (08/11/2023 - Diogo Lemos):

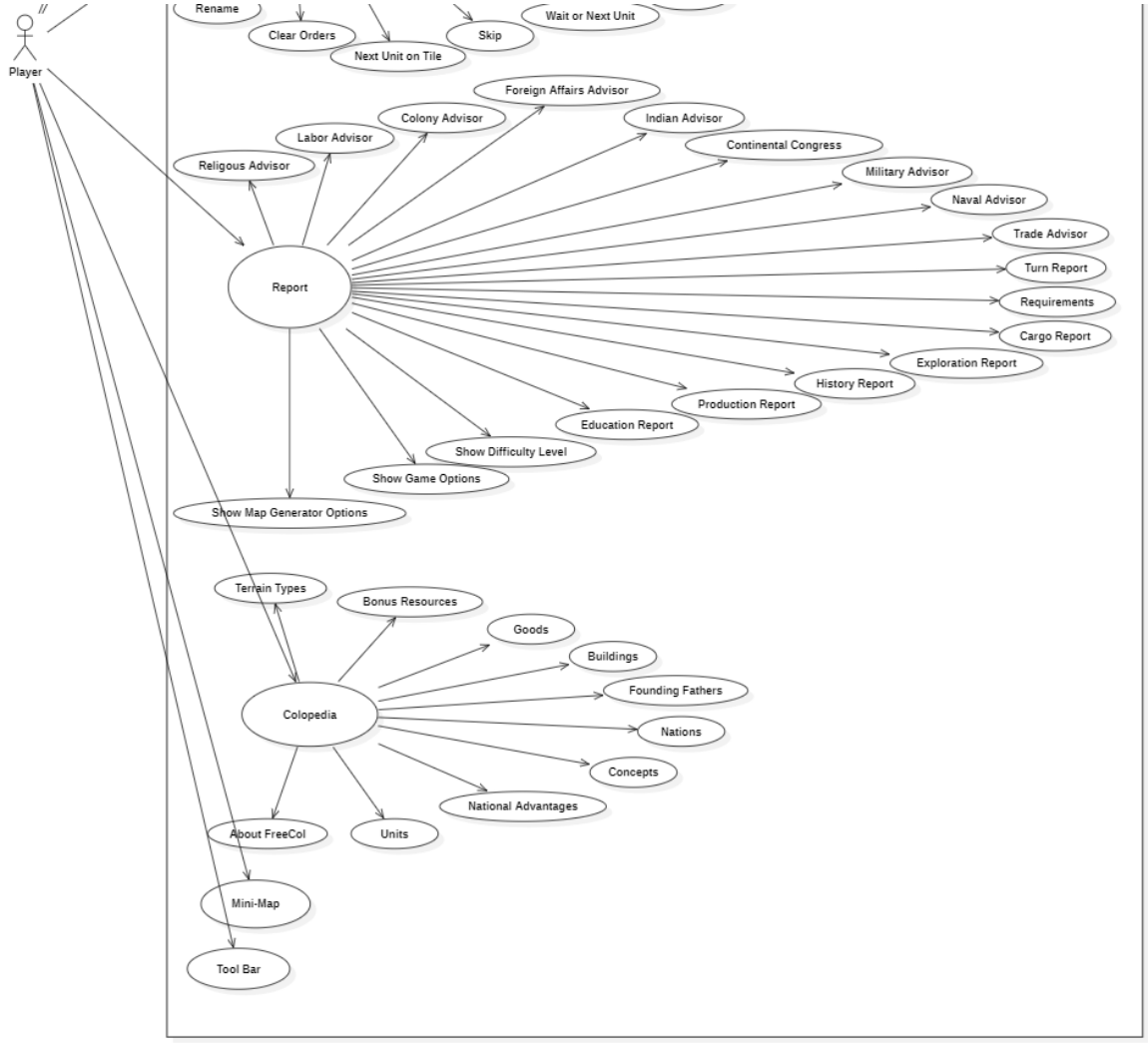
All code smells (Long Method, Long Class and Magic Number) appear to be correctly identified.

José Trigueiro - 58119

Use Case Diagram:

Use Case Diagram: Game Administration





Name: Game Administration.

Primary/Main Actor: Player.

Description: The Player wants to select a **Game** option or **View** something in the game or hide it or have a **Report** of something done or finally check the **Colopedia** to learn about the game. Additionally, can use the **Mini-Map** to view and **Tool-Bar** to manage the game.

Review(s): Reviewed by José Pereira:

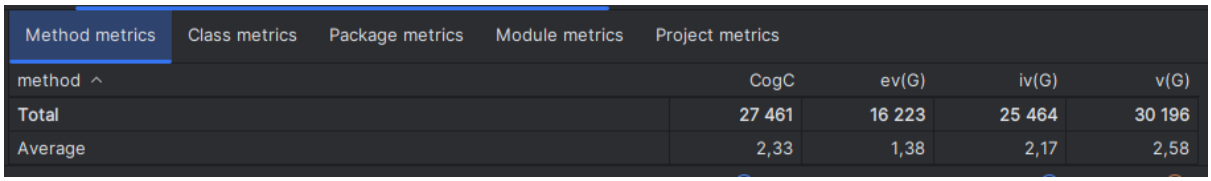
The Use Case diagram appears to be well constructed, highly detailed, and easy to understand. The Actors were correctly identified, and the description matches the Use Case perfectly.

Code Base Metrics:

Code Metric: Complexity Metrics

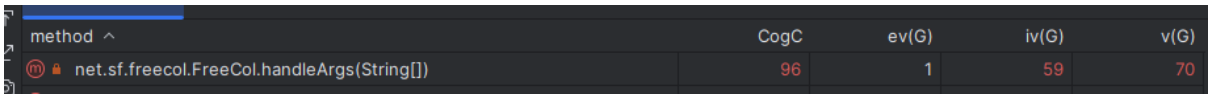
The Complexity Metric is the level of difficulty in understanding and maintaining a piece of software code. It measures how much effort is required to comprehend the code, make modifications or fix bugs.

- **Method metrics:**



method	CogC	ev(G)	iv(G)	v(G)
Total	27 461	16 223	25 464	30 196
Average	2,33	1,38	2,17	2,58

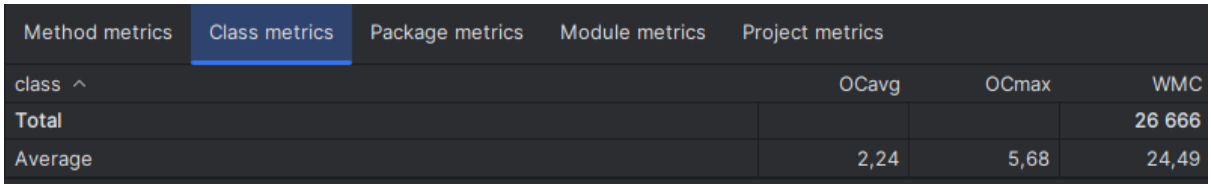
- **CogC** (Cognitive Complexity): is a measure of how difficult a unit of code is to intuitively understand;
- **ev(G)** (Essential Cyclomatic Complexity): is a version of Cyclomatic Complexity that ignores structures that allow multiple statements to be sequentially executed as a group. It provides a measure of the number of decision points in a method, plus one for the method entry;
- **iv(G)** (Design Complexity): it's a measure of the intricacy of the structure of a software system, the metric quantifies the level of difficulty in managing and evolving a particular design of software, for example a high design complexity often leads to a higher likelihood of errors, as it can be more difficult to understand the interaction between different components of the software;
- **v(G)** (Cyclomatic Complexity): it indicates the complexity of a program and the quantitative measure of the number of linearly independent paths through a program's source code.



method	CogC	ev(G)	iv(G)	v(G)
net.sf.freecol.FreeCol.handleArgs(String[])	96	1	59	70

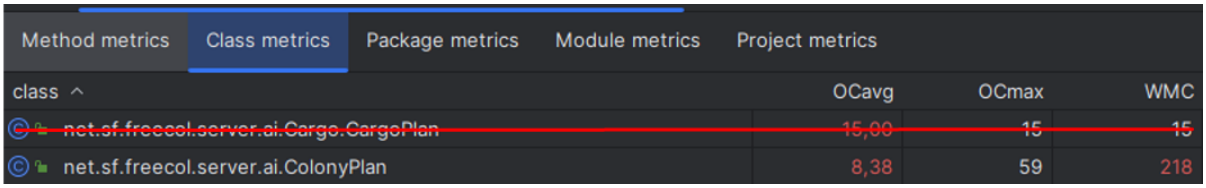
The code smell, Large Method I found on "handleArgs()" relates to the fact that it has a large impact on the average value, due to its size and complexity of understanding, affecting heavily its Cognitive Complexity, Design Complexity and Cyclomatic Complexity metrics.

- **Class metrics:**



Method metrics	Class metrics	Package metrics	Module metrics	Project metrics		
class				OCavg	OCmax	WMC
Total						26 666
Average				2,24	5,68	24,49

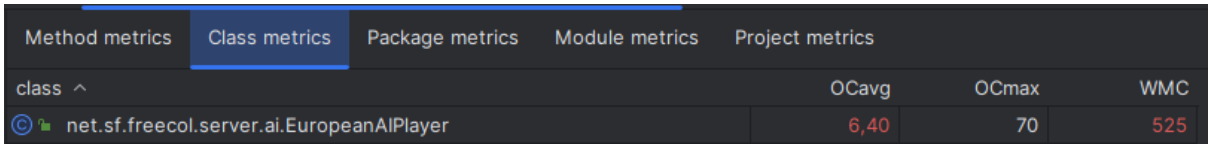
- **OCavg** (Average Operation Complexity): it measures the average complexity of operations (methods) in a class. It's calculated by summing the complexities(for ex. Cyclomatic Complexity) of all operations in the class and dividing by the number of operations. A high Average Operation Complexity might indicate that the operations in the class are doing too much and could be broken down into smaller, more manageable methods. This can make the code easier to understand, test, and maintain;
- **OCmax** (Maximum Operation Complexity): it measures the complexity of the most complex operation (method) in a class. It's calculated by determining the complexities of all operations in the class and selecting the highest one. A high Maximum Operation Complexity might indicate that there is a method in the class that is doing too much and could be broken down into smaller, more manageable methods. This can make the code easier to understand, test, and maintain;
- **WMC** (Weighted Method Complexity): it measures the total complexity of all methods in a class. It's calculated by summing the complexities(for example Cyclomatic Complexity) of all methods in the class. A high WMC indicates that a class may be doing too much and could be broken down into smaller, more manageable classes. It also suggests that the class may be harder to maintain and more prone to errors.



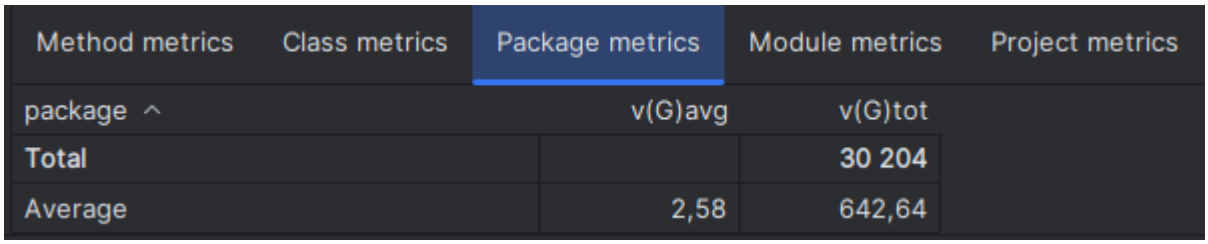
Method metrics	Class metrics	Package metrics	Module metrics	Project metrics		
class				OCavg	OCmax	WMC
net.sf.freecol.server.ai.Cargo.CargoPlan				15,00	15	15
net.sf.freecol.server.ai.ColonyPlan				8,38	59	218

The code smell, Large Class I found on "ColonyPlan.java" relates to the fact that it has a large impact on the average value, due to its size and complexity of understanding, affecting heavily its Average Operation Complexity, Maximum Operation Complexity and Cyclomatic Complexity metrics.

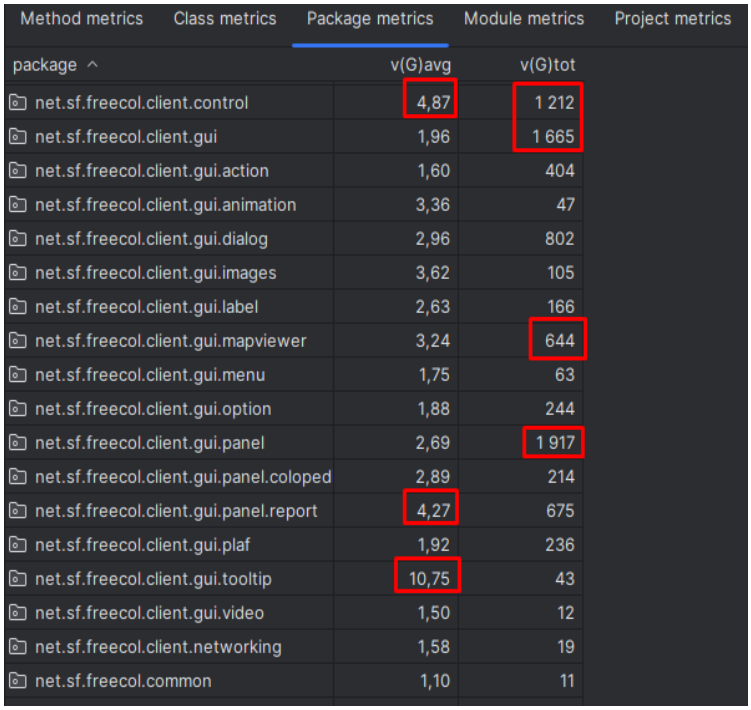
The code smell, Message Chains found in the methods isAggressive()" & "isLikesAttackingNatives()" & "needsMoreDragoons()" & "reallyNeedsMoreDragoons()" & "reallyNeedsMoreArtillery()" & "needsMoreArtillery()" in "EuropeanAIPlayer.java" also affect the same metrics.



- **Package metrics:**



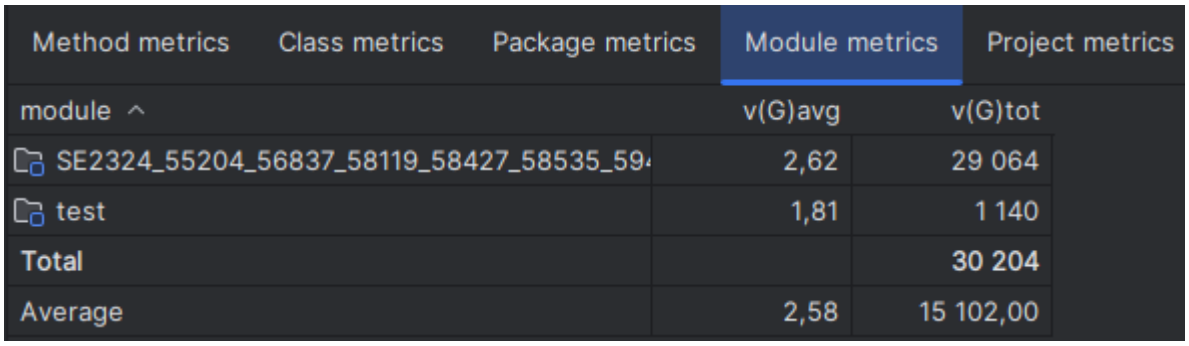
- **v(G)avg** (Average Cyclomatic Complexity): it's a metric that measures the average complexity of all the classes in a package and is calculated by summing the complexities of all classes in the package and dividing by the number of classes. A high Average Cyclomatic Complexity might indicate that the classes in the package are doing too much and could be broken down into smaller, more manageable classes. This can make the code easier to understand, test, and maintain;
- **v(G)tot** (Total Cyclomatic Complexity): it measures the sum of the cyclomatic complexity of all the classes in a package. It's calculated by summing the cyclomatic complexities of all classes in the package. A high Total Cyclomatic Complexity might indicate that the classes in the package are doing too much and could be broken down into smaller, more manageable classes. This can make the code easier to understand, test, and maintain.



Method metrics	Class metrics	Package metrics	Module metrics	Project metrics
package		v(G)avg	v(G)tot	
net.sf.freecol.client.control		4,87	1 212	
net.sf.freecol.client.gui		1,96	1 665	
net.sf.freecol.client.gui.action		1,60	404	
net.sf.freecol.client.gui.animation		3,36	47	
net.sf.freecol.client.gui.dialog		2,96	802	
net.sf.freecol.client.gui.images		3,62	105	
net.sf.freecol.client.gui.label		2,63	166	
net.sf.freecol.client.gui.mapviewer		3,24	644	
net.sf.freecol.client.gui.menu		1,75	63	
net.sf.freecol.client.gui.option		1,88	244	
net.sf.freecol.client.gui.panel		2,69	1 917	
net.sf.freecol.client.gui.panel.coloped		2,89	214	
net.sf.freecol.client.gui.panel.report		4,27	675	
net.sf.freecol.client.gui.plaf		1,92	236	
net.sf.freecol.client.gui.tooltip		10,75	43	
net.sf.freecol.client.gui.video		1,50	12	
net.sf.freecol.client.networking		1,58	19	
net.sf.freecol.common		1,10	11	

As for bad package metrics we have a mix of bad values all throughout the packages compared to the average but it doesn't seem that bad.

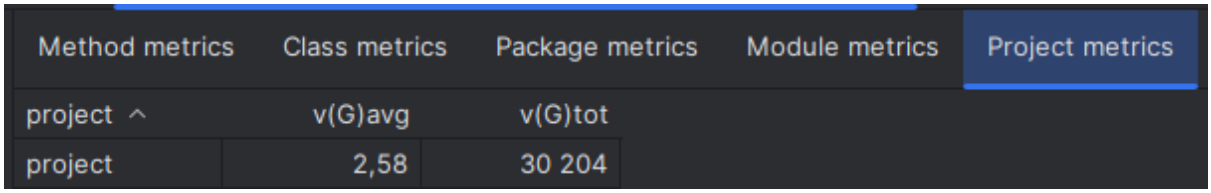
- **Module metrics:**



Method metrics	Class metrics	Package metrics	Module metrics	Project metrics
module			v(G)avg	v(G)tot
SE2324_55204_56837_58119_58427_58535_59			2,62	29 064
test			1,81	1 140
Total				**30 204**
Average			2,58	15 102,00

- **v(G)avg** (Average Cyclomatic Complexity): it's a metric that measures the average complexity of all the classes in a module. It's calculated by summing the complexities of all classes in the module and dividing by the number of classes. A high Average Cyclomatic Complexity might indicate that the classes in the module are doing too much and could be broken down into smaller, more manageable classes. This can make the code easier to understand, test, and maintain;
- **v(G)tot** (Total Cyclomatic Complexity): it measures the sum of the cyclomatic complexity of all the classes in a module. It's calculated by summing the cyclomatic complexities of all classes in the module. A high Total Cyclomatic Complexity might indicate that the classes in the module are doing too much and could be broken down into smaller, more manageable classes. This can make the code easier to understand, test, and maintain.

- **Project Metrics:**



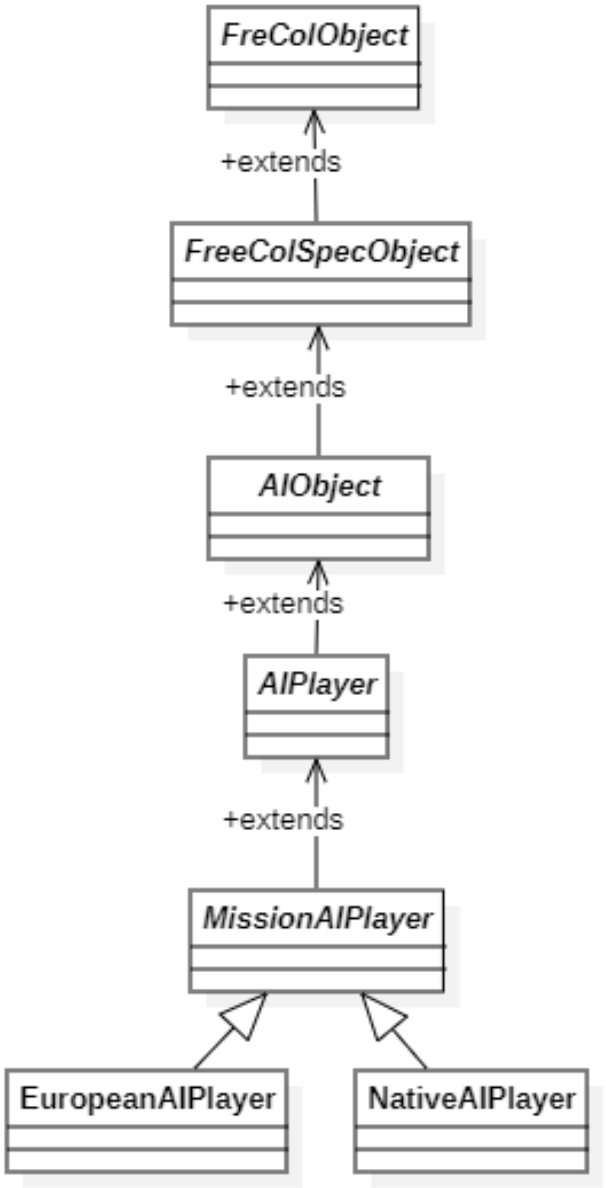
- **v(G)avg** (Average Cyclomatic Complexity): it's a metric that measures the average complexity of all the modules in a project. It's calculated by summing the complexities of all modules in the project and dividing by the number of modules. A high Average Cyclomatic Complexity might indicate that the modules in the project are doing too much and could be broken down into smaller, more manageable modules. This can make the code easier to understand, test, and maintain;
- **v(G)tot** (Total Cyclomatic Complexity): it measures the sum of the cyclomatic complexity of all the modules in a project. It's calculated by summing the cyclomatic complexities of all modules in the project. A high Total Cyclomatic Complexity might indicate that the modules in the project are doing too much and could be broken down into smaller, more manageable modules. This can make the code easier to understand, test, and maintain.

Review(s): Reviewed by José Pereira:

The Project Metrics appear to touch every crucial detail about the metrics in question, and my colleague made sure to analyze the variations from the Average Complexity's values and explained with great detail its consequences.

GOF Patterns:

1. Template method pattern - The pattern has a skeleton of operations, and the details are implemented by the child classes. This means that the overall structure and sequence of the algorithm are preserved by the parent class. The pattern encapsulates the algorithms in separate classes.



\src\net\sf\freecol\server\ai\AIObject.java

```
50  public abstract class FreeColObject
51      implements Comparable<FreeColObject>, ObjectWithId {
52
```

\src\net\sf\freecol\common\model\FreeColSpecObject.java

```
public abstract class FreeColSpecObject extends FreeColObject {
```

\src\net\sf\freecol\server\ai\AIObject.java

```
public abstract class AIObject extends FreeColObject {
```

\src\net\sf\freecol\server\ai\AIPlayer.java

```
68   */
69  public abstract class AIPlayer extends AIObject {
70
```

\src\net\sf\freecol\server\ai\MissionAIPlayer.java

```
public abstract class MissionAIPlayer extends AIPlayer {
```

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

```
public class EuropeanAIPlayer extends MissionAIPlayer {
```

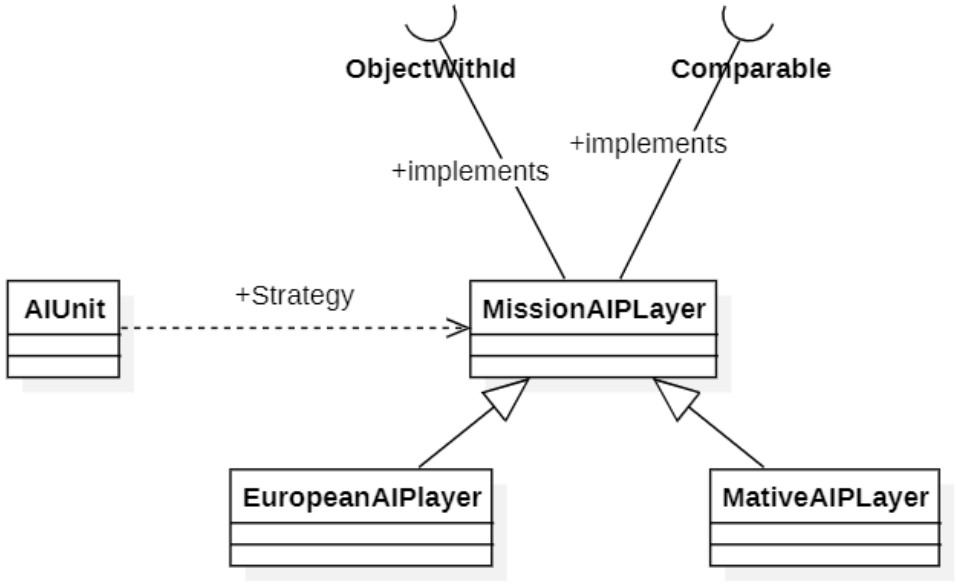
\src\net\sf\freecol\server\ai\NativeAIPlayer.java

```
3  public final class NativeAIPlayer extends MissionAIPlayer {
```

2. Strategy Design Pattern -

It provides a flexible way to encapsulate and swap behavior of an object.

The behavior of the object's behavior can change dynamically.



\src\net\sf\freecol\server\ai\MissionAIPlayer.java

```
56    */
57  public abstract class MissionAIPlayer extends AIPlayer {
58
```

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

```
public class EuropeanAIPlayer extends MissionAIPlayer {
```

\src\net\sf\freecol\server\ai\NativeAIPlayer.java

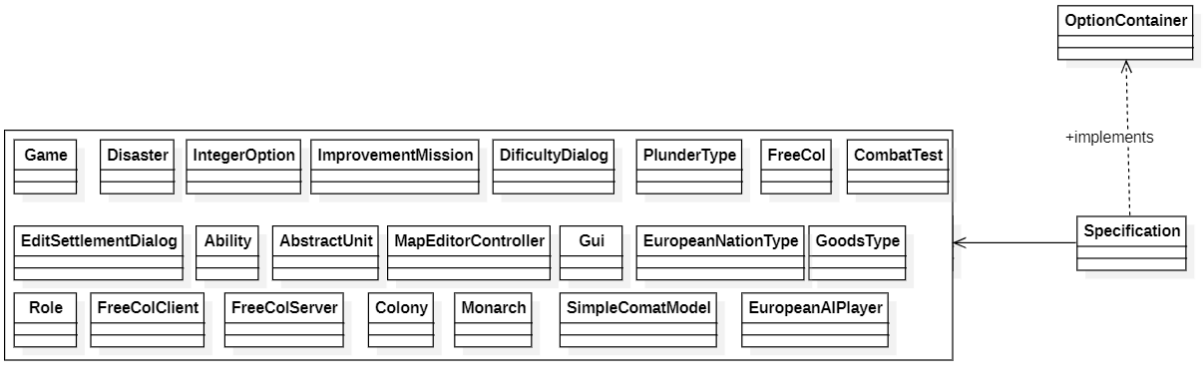
```
public final class NativeAIPlayer extends MissionAIPlayer {
```

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

```
public final class AIUnit extends TransportableAIObject {
```

```
Sebastian Zhorel +1
public boolean tryWorkInsideColonyMission(AIColony aiColony, LogBuilder lb) {
    WorkInsideColonyMission wic
        = getMission(WorkInsideColonyMission.class);
    if (wic == null) {
        AIPlayer aiPlayer = getAIOwner();
        if (!(aiPlayer instanceof EuropeanAIPlayer) ||
                ((EuropeanAIPlayer)aiPlayer)
                .getWorkInsideColonyMission( aiUnit: this, aiColony) == null) {
            return false;
        }
        lb.add(", ", getMission());
        dropTransport();
    }
    return true;
}

Sebastian Zhorel +1
public boolean tryPioneeringMission(LogBuilder lb) {
    Mission m = getMission();
    Location oldTarget = (m == null) ? null : m.getTarget();
    AIPlayer aiPlayer = getAIOwner();

    if (aiPlayer instanceof EuropeanAIPlayer) {
        EuropeanAIPlayer euaiPlayer = (EuropeanAIPlayer)aiPlayer;
        if (euaiPlayer.getPioneeringMission( aiUnit: this,  target: null) != null) {
            lb.add(", ", getMission());
            euaiPlayer.updateTransport( aiu: this, oldTarget, lb);
            return true;
        }
    }
    return false;
}
```

3. Factory Method Pattern - It has an interface “OptionContainer.java” for creating objects, but allows subclasses like “Specification” to decide which class to instantiate. It encapsulates the logic required to instantiate several complex objects.



src/net/sf/freecol/common/option/OptionContainer.java

```
public interface OptionContainer {
```

src/net/sf/freecol/common/model/Specification.java

```
public final class Specification implements OptionContainer {
```

```
2513        private boolean fixDifficultyOptions() {
2514            boolean ret = false;
2515            String id;
2516            AbstractOption op;
```

```
2588        if (ulo != null) {
2589            AbstractUnitOption regulars
2590                = new AbstractUnitOption( id: id + ".regulars", specification: this);
2591            regulars.setValue(new AbstractUnit( id: "model.unit.kingsRegular",
2592                                               roleId: "model.role.infantry", number: 31));
2593            ulo.getValue().add(regulars);
2594            AbstractUnitOption dragoons
2595                = new AbstractUnitOption( id: id + ".dragoons", specification: this);
2596            dragoons.setValue(new AbstractUnit( id: "model.unit.kingsRegular",
2597                                               roleId: "model.role.cavalry", number: 15));
2598            ulo.getValue().add(dragoons);
2599            AbstractUnitOption artillery
2600                = new AbstractUnitOption( id: id + ".artillery", specification: this);
2601            artillery.setValue(new AbstractUnit( id: "model.unit.artillery",
2602                                                DEFAULT_ROLE_ID, number: 14));
2603            ulo.getValue().add(artillery);
2604            AbstractUnitOption menOfWar
2605                = new AbstractUnitOption( id: id + ".menOfWar", specification: this);
2606            menOfWar.setValue(new AbstractUnit( id: "model.unit.manOWar",
2607                                               DEFAULT_ROLE_ID, number: 8));
2608            ulo.getValue().add(menOfWar);
2609            ret = true;
2610        }
2611        id = GameOptions.IMMIGRANTS;
2612        ulo = checkDifficultyUnitListOption(id, GameOptions.DIFFICULTY_IMMIGRATION, lb);
2613        if (ulo != null) {
2614            AbstractUnitOption i1
2615                = new AbstractUnitOption( id: id + ".1", specification: this);
2616            i1.setValue(new AbstractUnit( id: "model.unit.masterCarpenter",
2617                                         DEFAULT_ROLE_ID, number: 1));
2618            ulo.getValue().add(i1);
2619            ret = true;
2620        }
```

```
        id = GameOptions.INTERVENTION_FORCE;
        ulo = checkDifficultyUnitListOption(id, GameOptions.DIFFICULTY_MONARCH, lb);
        if (ulo != null) {
            AbstractUnitOption regulars
                = new AbstractUnitOption( id: id + ".regulars", specification: this);
            regulars.setValue(new AbstractUnit( id: "model.unit.colonialRegular",
                                               roleId: "model.role.soldier", number: 2));
            ulo.getValue().add(regulars);
            AbstractUnitOption dragoons
                = new AbstractUnitOption( id: id + ".dragoons", specification: this);
            dragoons.setValue(new AbstractUnit( id: "model.unit.colonialRegular",
                                               roleId: "model.role.dragoon", number: 2));
            ulo.getValue().add(dragoons);
            AbstractUnitOption artillery
                = new AbstractUnitOption( id: id + ".artillery", specification: this);
            artillery.setValue(new AbstractUnit( id: "model.unit.artillery",
                                                DEFAULT_ROLE_ID, number: 2));
            ulo.getValue().add(artillery);
            AbstractUnitOption menOfWar
                = new AbstractUnitOption( id: id + ".menOfWar", specification: this);
            menOfWar.setValue(new AbstractUnit( id: "model.unit.manOWar",
                                               DEFAULT_ROLE_ID, number: 2));
            ulo.getValue().add(menOfWar);
            ret = true;
        }
```

```
        id = GameOptions.MERCENARY_FORCE;
        ulo = checkDifficultyUnitListOption(id, GameOptions.DIFFICULTY_MONARCH, lb);
        if (ulo != null) {
            AbstractUnitOption regulars
                = new AbstractUnitOption( id: id + ".regulars", specification: this);
            regulars.setValue(new AbstractUnit( id: "model.unit.veteranSoldier",
                                               roleId: "model.role.soldier", number: 2));
            ulo.getValue().add(regulars);
            AbstractUnitOption dragoons
                = new AbstractUnitOption( id: id + ".dragoons", specification: this);
            dragoons.setValue(new AbstractUnit( id: "model.unit.veteranSoldier",
                                               roleId: "model.role.dragoon", number: 2));
            ulo.getValue().add(dragoons);
            AbstractUnitOption artillery
                = new AbstractUnitOption( id: id + ".artillery", specification: this);
            artillery.setValue(new AbstractUnit( id: "model.unit.artillery",
                                                DEFAULT_ROLE_ID, number: 2));
            ulo.getValue().add(artillery);
            AbstractUnitOption menOfWar
                = new AbstractUnitOption( id: id + ".menOfWar", specification: this);
            menOfWar.setValue(new AbstractUnit( id: "model.unit.manOWar",
                                               DEFAULT_ROLE_ID, number: 2));
            ulo.getValue().add(menOfWar);
            ret = true;
        }
```

Review(s): Review (08/11/2023 - Diogo Lemos):

There may be more classes than what is required in the first pattern (Template Method) diagram, but it seems to be correctly identified.
While the inheritance does enable the possibility of having more than one implementation per method, I believe the pattern identified might not be the most accurate one, since the object's behaviour doesn’t really change dynamically.
The third pattern (Factory Method) seems to be correctly identified.

Code Smells:

1. Long Method - “handleArgs()” in (src/net/sf/freecol/tools/FreeCol.java)

The method is 273 lines long, to mitigate this we could extract chunks of code into separate methods. Each of these methods should do one thing and do it well. For example, have separate methods for handling each command line option like “handleAdvantagesOption”, “handleCheckSavegameOption”, “handleClientOptions”, etc and have generic method to handle them all, the use of polymorphism could be used. Also the string literals(like “advantages”, “check-savegame”, “clientOptions”, etc) that are used to check the command line options, could be replaced with constants or enumerations.

```
664     /**
665      * Processes the command-line arguments and takes appropriate
666      * actions for each of them.
667      *
668      * @param args The command-line arguments.
669      */
670     private static void handleArgs(String[] args) {
671         Options options = new Options();
672         for (String[] o : optionsTable) {
673             String arg = o[3];
674             Option op = new Option(o[0], o[1], arg != null,
675                 ((o[2].startsWith(prefix:"cli.")) ? Messages.message(o[2]) : o[2]));
676             if (arg != null) {
677                 boolean optional = false;
678                 if (arg.startsWith(prefix:"!")) {
679                     optional = true;
680                     arg = arg.substring(beginIndex:1, arg.length());
681                 }
682                 if (arg.startsWith(argDir)
683                     || arg.startsWith(argFile)) op.setType(File.class);
684                 if (arg.startsWith(prefix:"cli.")) arg = Messages.message(arg);
685                 op.setArgName(arg);
686                 op.setOptionalArg(optional);
687             }
688             options.addOption(op);
689         }
690 
691         CommandLineParser parser = new DefaultParser();
692         boolean usageError = false;
693         try {
694             CommandLine line = parser.parse(options, args);
695             if (line.hasOption("help") || line.hasOption("usage")) {
696                 printUsage(options, status:0);
697             }
698 
699             // Ignore "default-locale", "freecol-data", which are
700             // already handled in main()
701 
702             if (line.hasOption("advantages")) {
703                 String arg = line.getOptionValue("advantages");
704                 Advantages a = selectAdvantages(arg);
705                 if (a == null) {
706                     fatal(StringTemplate.template(value:"cli.error.advantages")
707                         .addName(key:"%advantages%", getValidAdvantages())
708                         .addName(key:"%arg%", arg));
709                 }
710             }
```

```
710         }
711
712         if (line.hasOption("check-savegame")) {
713             String arg = line.getOptionValue("check-savegame");
714             if (!FreeColDirectories.setSavegameFile(arg)) {
715                 fatal(StringTemplate.template(value:"cli.err.save")
716                     .addName(key:"%string%", arg));
717             }
718             checkIntegrity = true;
719             standAloneServer = true;
720         }
721
722         if (line.hasOption("clientOptions")) {
723             String fileName = line.getOptionValue("clientOptions");
724             if (!FreeColDirectories.setClientOptionsFile(fileName)) {
725                 // Not fatal.
726                 gripe(StringTemplate.template(value:"cli.error.clientOptions")
727                     .addName(key:"%string%", fileName));
728             }
729         }
730
731         if (line.hasOption("debug")) {
732             // If the optional argument is supplied use limited mode.
733             String arg = line.getOptionValue("debug");
734             if (arg == null || arg.isEmpty()) {
735                 // Let empty argument default to menus functionality.
736                 arg = FreeColDebugger.DebugMode.MENUS.toString();
737             }
738             if (!FreeColDebugger.setDebugModes(arg)) { // Not fatal.
739                 gripe(StringTemplate.template(value:"cli.error.debug")
740                     .addName(key:"%modes%", FreeColDebugger.getDebugModes()));
741             }
742             // Keep doing this before checking log-level option!
743             logLevels.add(new LogLevel(name:"", Level.FINEST));
744         }
745         if (line.hasOption("debug-run")) {
746             FreeColDebugger.enableDebugMode(FreeColDebugger.DebugMode.MENUS);
747             FreeColDebugger.configureDebugRun(line.getOptionValue("debug-run"));
748         }
749         if (line.hasOption("debug-start")) {
750             debugStart = true;
751             FreeColDebugger.enableDebugMode(FreeColDebugger.DebugMode.MENUS);
752         }
753
```

```
754         if (line.hasOption("difficulty")) {
755             String arg = line.getOptionValue("difficulty");
756             String difficulty = selectDifficulty(arg);
757             if (difficulty == null) {
758                 fatal(StringTemplate.template(value:"cli.error.difficulties")
759                     .addName(key:"%difficulties%", getValidDifficulties())
760                     .addName(key:"%arg%", arg));
761             }
762         }
763
764         if (line.hasOption("europeans")) {
765             int e = selectEuropeanCount(line.getOptionValue("europeans"));
766             if (e < 0) {
767                 gripe(StringTemplate.template(value:"cli.error.europeans")
768                     .addAmount(key:"%min%", EUROPEANS_MIN));
769             }
770         }
771
772         if (line.hasOption("fast")) {
773             fastStart = true;
774             introVideo = false;
775         }
776
777         if (line.hasOption("font")) {
778             fontName = line.getOptionValue("font");
779         }
780
781         if (line.hasOption("full-screen")) {
782             windowSize = null;
783         }
784
785         if (line.hasOption("headless")) {
786             headless = true;
787         }
788
789         if (line.hasOption("load-savegame")) {
790             String arg = line.getOptionValue("load-savegame");
791             if (!FreeColDirectories.setSavegameFile(arg)) {
792                 fatal(StringTemplate.template(value:"cli.error.save")
793                     .addName(key:"%string%", arg));
794             }
795         }
796
797         if (line.hasOption("log-console")) {
798             consoleLogging = true;
799         }
800
```

```
        if (line.hasOption("log-file")) {
            FreeColDirectories.setLogFilePath(line.getOptionValue("log-file"));
        }

        if (line.hasOption("log-level")) {
            for (String value : line.getOptionValues("log-level")) {
                String[] s = value.split(regex:":");
                logLevels.add((s.length == 1)
                    ? new LogLevel(name:"", Level.parse(upCase(s[0])))
                    : new LogLevel(s[0], Level.parse(upCase(s[1]))));
            }
        }

        if (line.hasOption("meta-server")) {
            String arg = line.getOptionValue("meta-server");
            if (!setMetaServer(arg)) {
                gripe(StringTemplate.template(value:"cli.error.meta-server")
                    .addName(key:"%arg%", arg));
            }
        }

        if (line.hasOption("name")) {
            setName(line.getOptionValue("name"));
        }

        if (line.hasOption("no-intro")) {
            introVideo = false;
        }
        if (line.hasOption("no-java-check")) {
            javaCheck = false;
        }
        if (line.hasOption("no-memory-check")) {
            memoryCheck = false;
        }
        if (line.hasOption("no-sound")) {
            sound = false;
        }
        if (line.hasOption("no-splash")) {
            splashStream = null;
        }

        if (line.hasOption("private")) {
            publicServer = false;
        }

        if (line.hasOption("server")) {
            standAloneServer = true;
        }
```

```
        if (line.hasOption("server-name")) {
            serverName = line.getOptionValue("server-name");
        }
        if (line.hasOption("server-port")) {
            String arg = line.getOptionValue("server-port");
            if (!setServerPort(arg)) {
                fatal(StringTemplate.template(value:"cli.error.serverPort")
                    .addName(key:"%string%", arg));
            }
        }
        if (line.hasOption("server-ip")) {
            String arg = line.getOptionValue("server-ip");
            if (!setServerAddress(arg)) {
                fatal(StringTemplate.template(value:"cli.error.serverIp")
                    .addName(key:"%string%", arg));
            }
        }

        boolean seeded = (line.hasOption("seed")
            && FreeColSeed.setFreeColSeed(line.getOptionValue("seed")));
        if (!seeded) FreeColSeed.generateFreeColSeed();

        if (line.hasOption("splash")) {
            String splash = line.getOptionValue("splash");
            try {
                InputStream fis = Files.newInputStream(Paths.get(splash));
                splashStream = fis;
            } catch (IOException ioe) {
                gripe(StringTemplate.template(value:"cli.error.splash")
                    .addName(key:"%name%", splash));
            }
        }

        if (line.hasOption("tc")) {
            setTc(line.getOptionValue("tc")); // Failure is deferred.
        }

        if (line.hasOption("rules")) {
            setRules(line.getOptionValue("rules")); // Failure is deferred.
        }

```

```
890         if (line.hasOption("timeout")) {
891             String arg = line.getOptionValue("timeout");
892             try {
893                 setTimeout(arg); // Not fatal
894             } catch (NumberFormatException nfe) {
895                 gripe(StringTemplate.template(value:"cli.error.timeout")
896                     .addName(key:"%string%", arg)
897                     .addName(key:"%minimum%", Long.toString(TIMEOUT_MIN)));
898             }
899         }
900 
901         if (line.hasOption("user-cache-directory")) {
902             String arg = line.getOptionValue("user-cache-directory");
903             String errMsg = FreeColDirectories.setUserCacheDirectory(arg);
904             if (errMsg != null) { // Not fatal.
905                 gripe(StringTemplate.template(errMsg)
906                     .addName(key:"%string%", arg));
907             }
908         }
909 
910         if (line.hasOption("user-config-directory")) {
911             String arg = line.getOptionValue("user-config-directory");
912             String errMsg = FreeColDirectories.setUserConfigDirectory(arg);
913             if (errMsg != null) { // Not fatal.
914                 gripe(StringTemplate.template(errMsg)
915                     .addName(key:"%string%", arg));
916             }
917         }
918 
919         if (line.hasOption("user-data-directory")) {
920             String arg = line.getOptionValue("user-data-directory");
921             String errMsg = FreeColDirectories.setUserDataDirectory(arg);
922             if (errMsg != null) { // Fatal, unable to save.
923                 fatal(StringTemplate.template(errMsg)
924                     .addName(key:"%string%", arg));
925             }
926         }
927 
928         if (line.hasOption("version")) {
929             System.out.println("FreeCol " + getVersion());
930             quit(status:0);
931         }
932 
933         if (line.hasOption("windowed")) {
934             String arg = line.getOptionValue("windowed");
935             setWindowSize(arg); // Does not fail
936         }
```

```
937 
938         } catch (ParseException e) {
939             System.err.println("\n" + e.getMessage() + "\n");
940             usageError = true;
941         }
942         if (usageError) printUsage(options, status:1);
943     }
```

2. **<u>Long Class</u>- "ColonyPlan.java"** in **(src/net/sf/freecol/server/ai/ColonyPlan.java)**
The class is 1523 lines long! To solve this problem each responsibility should have its own class and only after be called in this class that manages the colony plan.

```
1521            return lb.toString();
1522        }
1523    }
1524
```

3. **<u>Message Chains</u> - "isAggressive()" & "isLikesAttackingNatives()" & "needsMoreDragoons()" & "reallyNeedsMoreDragoons()" & "reallyNeedsMoreArtillery()" & "needsMoreArtillery()"** in **(src/net/sf/freecol/server/ai/EuropeanAiPlayer.java)**
These methods have really long message chains to get an object back due to rigidity and complexity in the design. There are several ways to solve this code smell. We could hide all the calls under a single method or reduce coupling between objects by moving behavior to the object that has the data or instead of querying objects for data and making decisions based on that data, instruct objects to perform actions.

```
310     public boolean isAggressive() {
311         /*
312          * TODO: We need to decide where to put AI behavior parameters so that mod
313          *       authors can customize the AI feel. Perhaps just in the specification?
314          */
315         return getPlayer().getNation().getType().getId().equals(anObject:"model.nationType.conquest")
316                 || getPlayer().getNation().getType().getId().equals(anObject:"model.nationType.immigration");
317     }
318
319     /**
320      * Checks if this player should be attacking the natives.
321      *
322      * @return {@code true} if native settlements should be targeted by this player.
323      */
324     public boolean isLikesAttackingNatives() {
325         return getPlayer().getNation().getType().getId().equals(anObject:"model.nationType.conquest");
326     }
```

```
private boolean needsMoreDragoons() {
    return 2 * (getAIColonies().size() + 1) > getAIUnits().stream().filter(au -> au.getUnit().isMounted() && au.getUnit().isArmed()).count();
}

private boolean reallyNeedsMoreDragoons() {
    return getAIColonies().size() + 1 > getAIUnits().stream().filter(au -> au.getUnit().isMounted() && au.getUnit().isArmed()).count();
}

public boolean reallyNeedsMoreArtillery() {
    return getAIColonies().size() / 2 + 1 > getAIUnits().stream().filter(au -> au.getUnit().hasAbility(Ability.BOMBARD)).count();
}

public boolean needsMoreArtillery() {
    return getAIColonies().size() + 1 > getAIUnits().stream().filter(au -> au.getUnit().hasAbility(Ability.BOMBARD)).count();
}
```

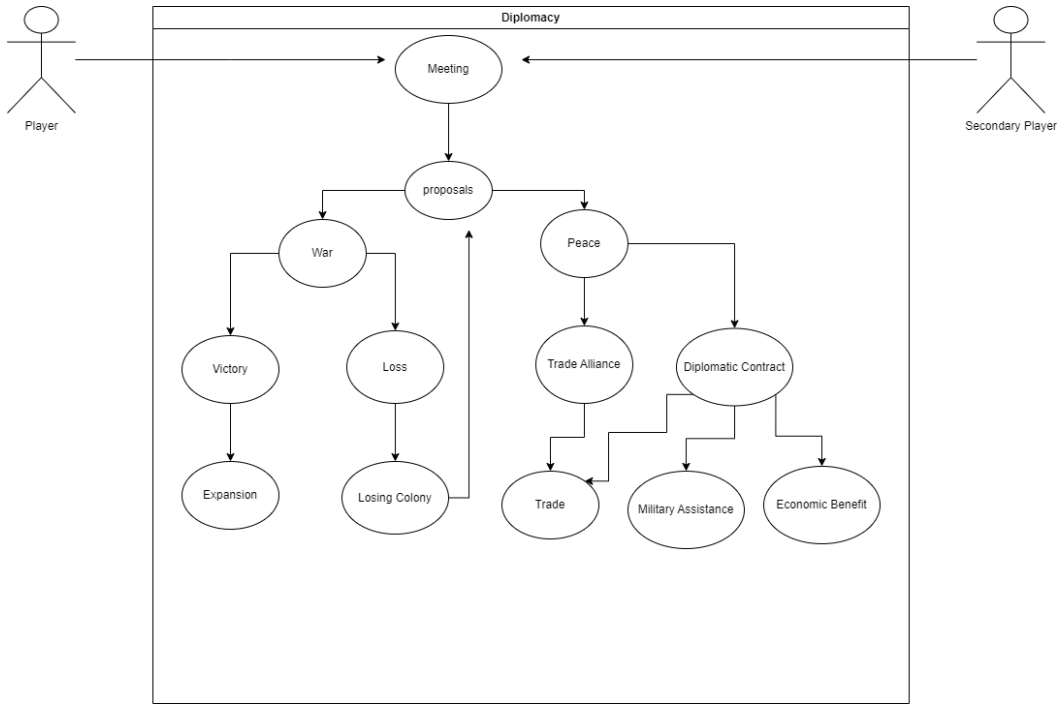
Review(s): Liliane Correia Review
Comments:
- Perhaps a reduction of the code presented in "handleArgs" method should be considered, providing only a snippet of it to illustrate the issue.
- The overall report is concise in demonstrating the code smells present in the program
- The analysis was well-done and even included solutions for the identified code smells.

Liliane Correia - 58427

Use Case Diagram:

Use Cases

Diplomacy



Some use cases:

Name: Expansion

Description: the player wants to expand the colony by war, Conquer.

Main: player

Secondary: Natives, Pirates, Other nations

Pre-condition: Have a boat and/or people on land

Main flow:

1. Meeting the secondary players
2. Receiving proposals for peace or war
3. Declare war
4. Attack
5. Checks if military force is bigger than the secondary
6. Win the war

Post-condition: Colony Expansion

Alternative flows: Does not declare war, military force is smaller

Alternative flow: Smaller Military force

Description: The player declares war, but military force is smaller than the opposition

Main: player

Secondary: Natives, Pirates, other nations

Pre-condition: Military force smaller than secondary

Main Flow:

1. Start from 4
2. Loses war
3. Loses parts of colony/power
4. returns to 2

Post-condition: lost property/people,

Alternative flow: Accept peace (does not declare war)

Description: the player accepts peace treaty from other players and goes for diplomatic contract

Main: player

Secondary: Natives, Pirates, other nations

Pre-condition: Receives proposal

Main Flow:

1. Start from 2
2. Accept peace proposal
3. Makes diplomatic contract

Post-condition: can benefits from trade and military assistance from allies

Alternatives flows: Make trade treaty

Alternative flow: Make trade treaty

Description: the player accepts peace treaty from other players

Main: player

Secondary: Natives, Pirates, other nations

Pre-condition: Accepts peace proposal

Main Flow:

1. Starts from 2
2. Makes trade treaty

Post-condition: can benefits from trade

Review(s): Review (08/11/2023 - Diogo Lemos):

Some grammar mistakes.

Very complete Use Case. All alternative flows scenarios were explored.

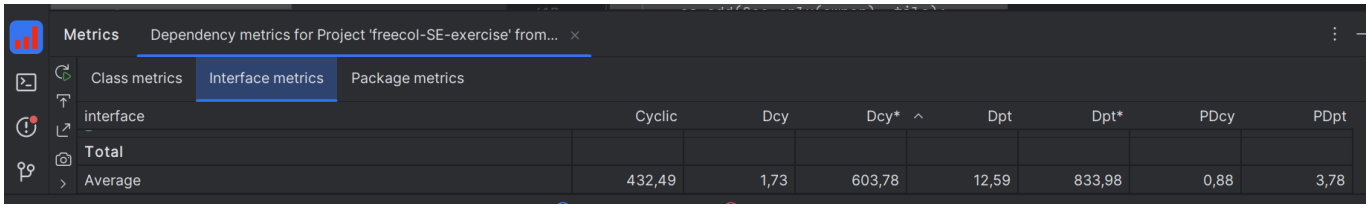
Code Base Metrics:

Code Metrics

Dependency metrics

Dependency metrics are quantitative measures. They are used in software engineering to analyze the dependencies between various components, modules, classes, or packages within a system.

Design Patterns like Factory or Adapter can be useful for high dependencies to help with coupling (interconnectedness/interdependences between components/classes)



Description:

- **Cyclic (Number of cyclic dependency) –** are situations where there is interdependency of two or more components/modules/packages in a system that form a closed loop. Does not apply, usually to interfaces.

- **Dcy (number of dependency)** – is the count of other classes/components that a particular class/module/component depends on. Measures interconnections with other parts of the software. High dependencies lead to high coupling. Does not apply, usually to interfaces but the class that implements the interface might have dependencies.

- **Dcy* (number of transitive dependency)** – is the count of classes/components that a particular class/module/package indirectly depends on. In other words, the class is dependent of another indirectly, through an intermediate class Does not apply, usually to interfaces but the class that implements the interface might have dependencies

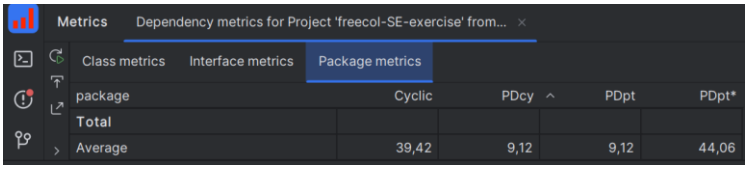
- **Dpt (number of dependents)** – is the count of other classes/components that directly depend on a specific class/component/package. Denotes how many other parts of the code are dependent on the functionality provided by that class. If we consider a class that implements a certain interface, then that class is dependent on that interface.

- **Dpt* (number of transitive dependents)** – is the count of other classes/components that indirectly depend on the specific class/component/package in question. In other words, it measures how many other parts of the code are indirectly dependent on the functionalities provided by that class. If we consider a class that implements a certain interface, then that class is dependent on that interface.

- **PDcy (number of package dependencies)** – is the count of packages/modules that the class/component/package directly depends on.

- **PDpt (number of dependent packages)** – is the count of packages/modules that depend on or have a direct dependency on the specific class/component/package in question. In other words, it measures how many other packages rely on the functionality provided by that

class/component/package. This metric helps assess how widely it is used across different parts of a software.

- 

For packages we have more:

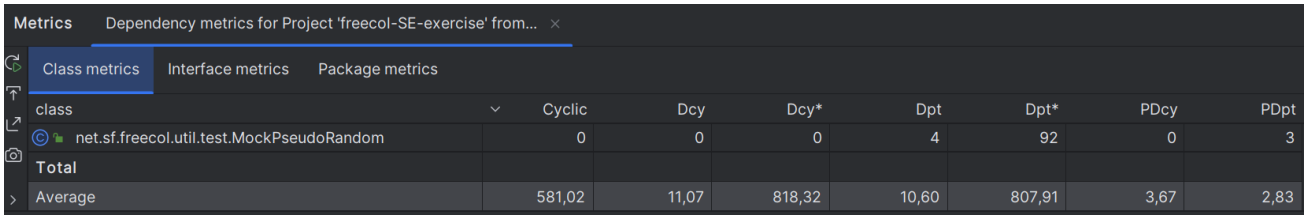
- **PDpt* (number of transitively dependent packages)** – it's the number of packages that are indirectly dependent on a specific package. They rely on the functionality of a certain package.

Note:

Average value – it represents the average value of the dependencies of a type (cyclic, Dcy, PDpt, etc) in each metric (Class, Interface, Package). Having **a high average** can indicate a significant degree of interdependence between the parts of the system.

FreeCol Dependency metrics

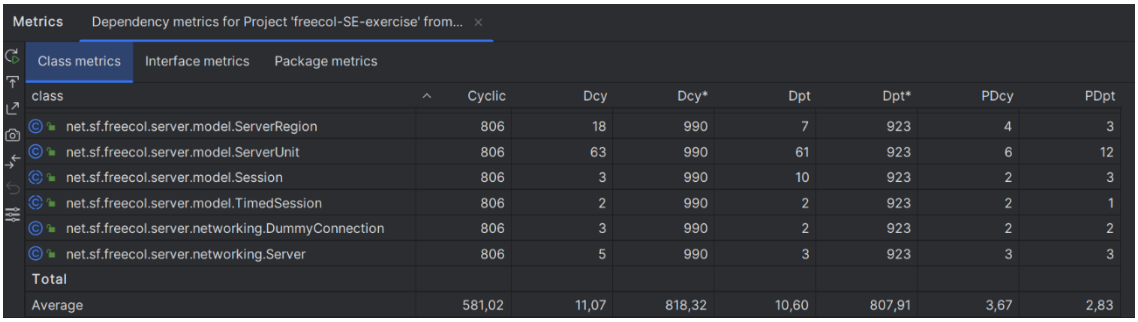
Class metrics:



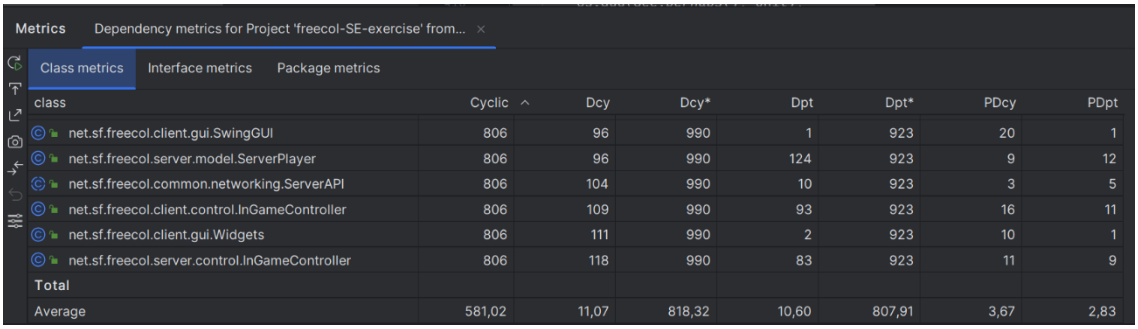
Classes in FreeCol have a high transitive dependency average – many classes depend indirectly on others

Examples:

- **High number of cyclic dependencies (Cyclic Collum)** – it means that multiple classes depend on each other in a circular/cyclical manner: component A depends on component B; B depends on component C and C depends on A.



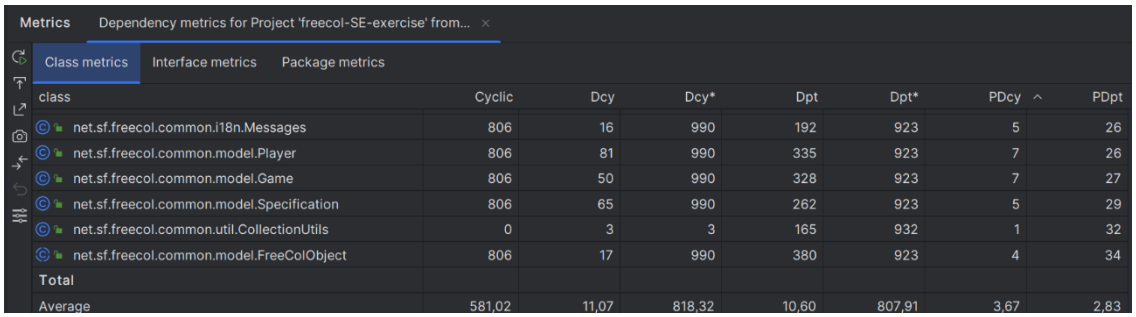
- **High number of dependencies (Dcy Collum)** – a high number of dependencies indicates high coupling, which can be a code smell. It means that a class relies on many other parts of



class	Cyclic	Dcy	Dcy*	Dpt	Dpt*	PDcy	PDpt
net.sf.freecol.client.gui.SwingGUI	806	96	990	1	923	20	1
net.sf.freecol.server.model.ServerPlayer	806	96	990	124	923	9	12
net.sf.freecol.common.networking.ServerAPI	806	104	990	10	923	3	5
net.sf.freecol.client.control.InGameController	806	109	990	93	923	16	11
net.sf.freecol.client.gui.Widgets	806	111	990	2	923	10	1
net.sf.freecol.server.control.InGameController	806	118	990	83	923	11	9
Total							
Average	581,02	11,07	818,32	10,60	807,91	3,67	2,83

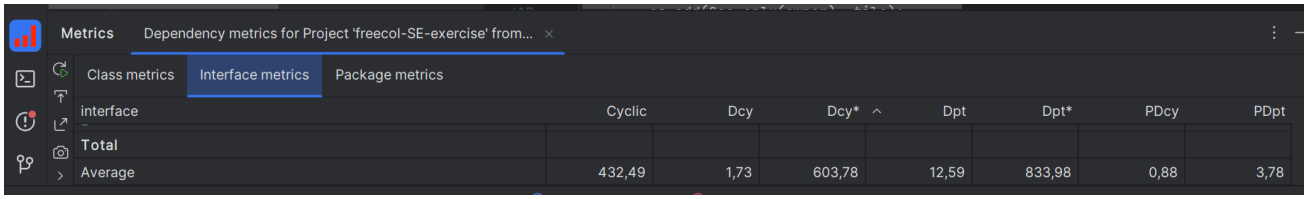
the code.

- **High number of dependent packages (PDpt Collum)** – means that the class is being used extensively by various parts of the system



class	Cyclic	Dcy	Dcy*	Dpt	Dpt*	PDcy	PDpt
net.sf.freecol.common.i18n.Messages	806	16	990	192	923	5	26
net.sf.freecol.common.model.Player	806	81	990	335	923	7	26
net.sf.freecol.common.model.Game	806	50	990	328	923	7	27
net.sf.freecol.common.model.Specification	806	65	990	262	923	5	29
net.sf.freecol.common.util.CollectionUtils	0	3	3	165	932	1	32
net.sf.freecol.common.model.FreeColObject	806	17	990	380	923	4	34
Total							
Average	581,02	11,07	818,32	10,60	807,91	3,67	2,83

Interface metrics:

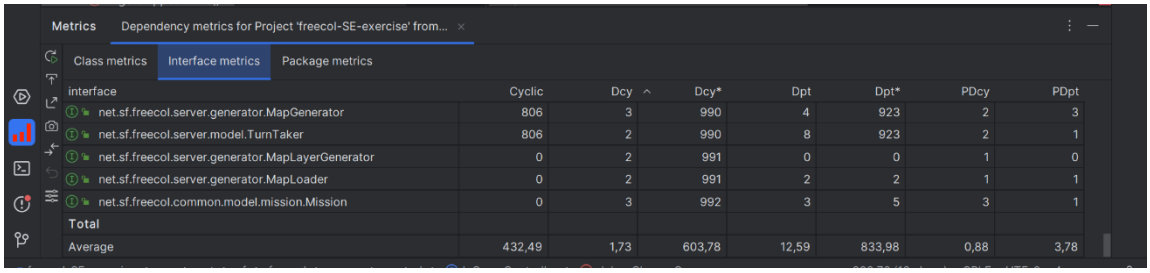


interface	Cyclic	Dcy	Dcy*	Dpt	Dpt*	PDcy	PDpt
Total							
Average	432,49	1,73	603,78	12,59	833,98	0,88	3,78

Interfaces in FreeCol have a high transitive dependent average; means that many other parts of the system depend indirectly on these interfaces. This is likely due to the classes that are implement the interfaces.

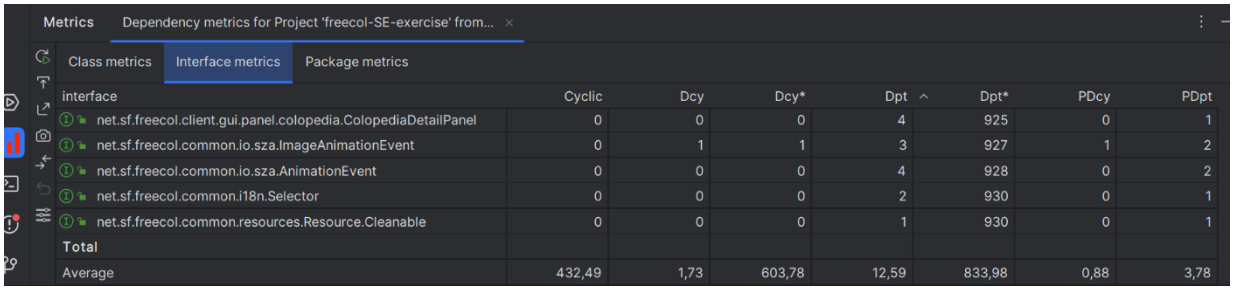
Examples:

- **High number of transitive dependencies (Dcy* Collum)** – it means that the interface is indirectly connected to many other components.



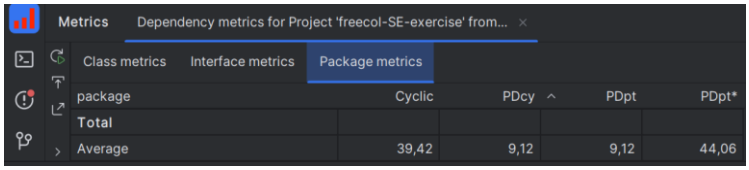
interface	Cyclic	Dcy	Dcy*	Dpt	Dpt*	PDcy	PDpt
net.sf.freecol.server.generator.MapGenerator	806	3	990	4	923	2	3
net.sf.freecol.server.model.TurnTaker	806	2	990	8	923	2	1
net.sf.freecol.server.generator.MapLayerGenerator	0	2	991	0	0	1	0
net.sf.freecol.server.generator.MapLoader	0	2	991	2	2	1	1
net.sf.freecol.common.model.mission.Mission	0	3	992	3	5	3	1
Total							
Average	432,49	1,73	603,78	12,59	833,98	0,88	3,78

- **High number of transitive dependents (Dpt* Collum)** – indicates that the interface indirectly affects a large portion of the software system.



interface	Cyclic	Dcy	Dcy*	Dpt	Dpt*	PDcy	PDpt
net.sf.freecol.client.gui.panel.colopedia.ColopediaDetailPanel	0	0	0	4	925	0	1
net.sf.freecol.common.io.sza.ImageAnimationEvent	0	1	1	3	927	1	2
net.sf.freecol.common.io.sza.AnimationEvent	0	0	0	4	928	0	2
net.sf.freecol.common.i18n.Selector	0	0	0	2	930	0	1
net.sf.freecol.common.resources.Resource.Cleanable	0	0	0	1	930	0	1
Total							
Average	432,49	1,73	603,78	12,59	833,98	0,88	3,78

Package metrics:

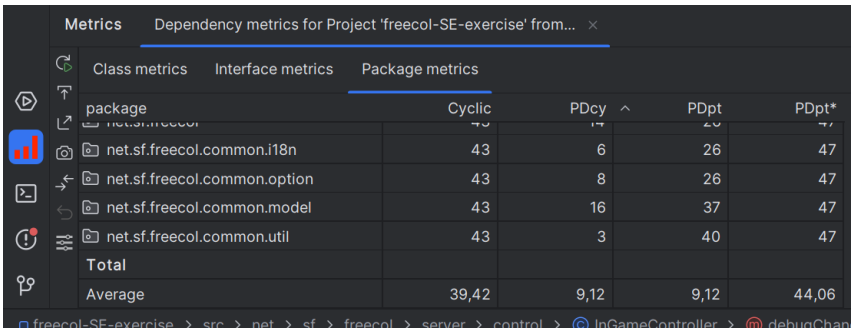


package	Cyclic	PDcy	PDpt	PDpt*
Total				
Average	39,42	9,12	9,12	44,06

Packages in FreeCol have a high transitively dependent package average. In other words, many packages rely indirectly on other packages

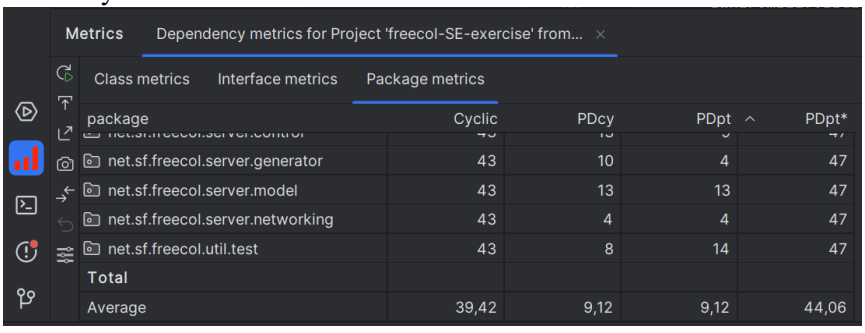
Examples:

- **High number of package dependents (PDpt Collum)** – it signifies that a certain package plays a critical role in the software. In other words, many other parts of the code rely on functionalities provided by the package. Utility libraries are used very common because they have general-purposes functions/classes/methods



package	Cyclic	PDcy	PDpt	PDpt*
net.sf.freecol.common.i18n	43	6	26	47
net.sf.freecol.common.option	43	8	26	47
net.sf.freecol.common.model	43	16	37	47
net.sf.freecol.common.util	43	3	40	47
Total				
Average	39,42	9,12	9,12	44,06

- **High number of transitively dependent packages (PDpt* Collum)** – packages that are indirectly relied on.



package	Cyclic	PDcy	PDpt	PDpt*
net.sf.freecol.server.generator	43	10	4	47
net.sf.freecol.server.model	43	13	13	47
net.sf.freecol.server.networking	43	4	4	47
net.sf.freecol.util.test	43	8	14	47
Total				
Average	39,42	9,12	9,12	44,06

Review(s): - Wilker Martins 58535 review

Comment: The comprehensive explanation of dependency metrics in the field of software engineering is both informative and insightful. It effectively dissects various metrics, including cyclic dependency, Dcy, Dcy*, Dpt, Dpt*, PDcy, PDpt, and PDpt*, providing a clear understanding of their significance and practical applications.

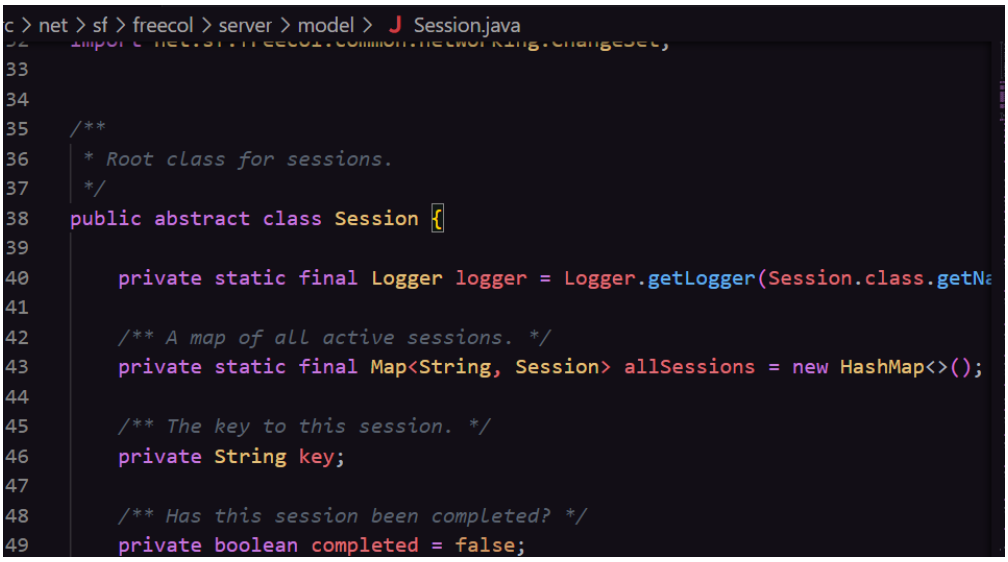
The incorporation of practical examples and scenarios, alongside the note on calculating average values, enhances the depth and real-world applicability of the discussion. This enables readers to not only comprehend the metrics' theoretical aspects but also to envision their practical use within their own software projects.

The organization of the content is commendable, with each metric succinctly explained and presented in a logical order. Furthermore, the suggestion to leverage design patterns like Factory or Adapter to address high dependencies is a pragmatic approach to resolving potential issues. This practical perspective is valuable for software developers and engineers seeking to improve the maintainability and efficiency of their software systems.

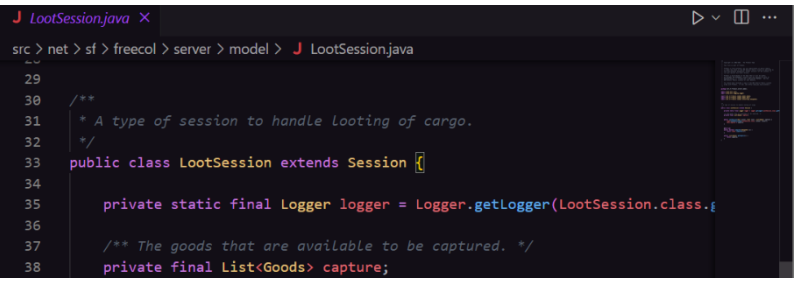
GOF Patterns:

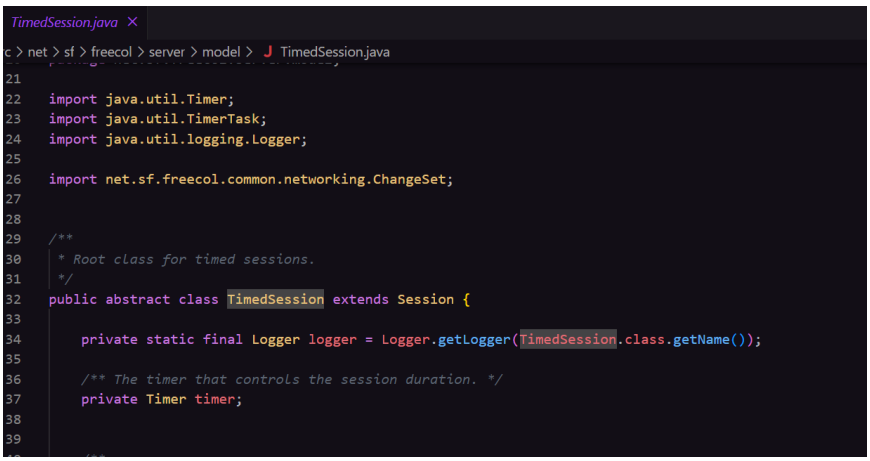
Design Patterns

1- **Template Method Pattern** – We use the generalized algorithm Session into LootSession, MonarchSession and TimedSession.
In: Src/sf/freecol/server/model
Session Class:

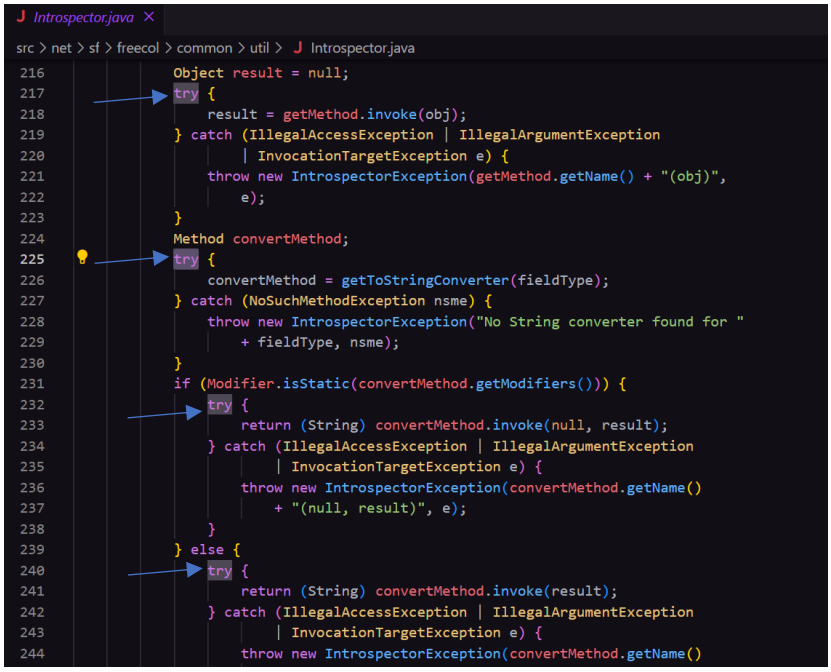


Extends to:





2- **Chain of Responsibility** – There is a chain of objects that are responsible for handling requests. In this case, the Try/Catch method is used.
In: src/net/freecol/common/util/introspector.java



3- **Singleton** - it has only one instance, specification. This class is responsible for only this object and provides a global point of accessing it, getSpecification.
In: src/net/freecol/common/model/FreeColSpecObject.java

```
public abstract class FreeColSpecObject extends FreeColObject {


    /**documentation */
    private Specification specification;


        /**documentation */
    public FreeColSpecObject(Specification specification) {
        this.specification = specification;
    }

    public Specification getSpecification() {
        return this.specification;
    }

    /**documentation */
    @Override
    protected void setSpecification(Specification specification) {
        this.specification = specification;
    }

    /**documentation */
    @Override
    public <T extends FreeColObject> boolean copyIn(T other) {
        FreeColSpecObject o = copyInCast(other,
FreeColSpecObject.class);
        if (o == null || !super.copyIn(o)) return false;
        this.specification = o.getSpecification();
        return true;
    }
```

Review(s): Wilker Martins 58535

Comment: It appears that the Singleton pattern might not have been the most suitable choice for this class. One of the key characteristics of a Singleton is to restrict the instantiation of the class to a single instance and ensure that access to that instance is controlled via a global point of access method. However, in this code, it seems that the class's methods are not private, potentially allowing multiple instances or direct instantiation, which can undermine the intended behavior of a Singleton. Reevaluating the use of Singleton or ensuring proper access control for the class's methods may be necessary to align with the Singleton pattern's principles. However, the other design patterns (Template Method and Chain of Responsibility) appear to have been well-suited for their respective use cases.

José Trigueiro 58119 -

1- The Template Method was well identified and it's correct.

2- I don't think that is a chain of responsibility code pattern, the try catches go nowhere, they simply throw exceptions.

3 - I'm not sure if it's a Singleton Pattern the variable "specification" should be "static" for the class to be considered a Singleton.

Code Smells:

Code Smells

1. **Data Class –** contains only data and no real functionality. It inherits certain data just to return it again. Perhaps it can be used as another component in the session class instead of a class.
   In: src/net/sf/freecol/server/model/LootSession.java

```
public class LootSession extends Session {

    private static final Logger logger = Logger.getLogger(LootSession.class.getName());

    /** The goods that are available to be captured. */
    private final List<Goods> capture;


    public LootSession(Unit winner, Unit loser, List<Goods> capture) {
        super(makeSessionKey(LootSession.class, winner, loser));
        this.capture = capture;
    }


    @Override
    public boolean complete(ChangeSet cs) {
        return super.complete(cs);
    }

    public List<Goods> getCapture() {
        return capture;
    }
}
```

2. **Long method** – the main is too long. It has 9202 lines. It takes up most of the class and it is more complex than needs to be. We could do the Extract Method and put certain functionalities in other methods, simplifying the method and making it easier to understand. **In**: src/net/sf/freecol/tools/ForestMaker.java – main method

```
public static void main(String[] args) throws IOException {

    if (args.length == 0) {
        System.out.println("Usage: ForestMaker <directory>...");
        System.out.println("Directory name should match a directory in");
        System.out.println("   " + DESTDIR);
        System.exit(1);
    }

    String riverName = "data/rules/classic/resources/images/terrain/"
            + "ocean/center0.png";

(… more code …)

                // sort by y, x coordinate
                trees.sort(Comparator.naturalOrder());
                for (ImageLocation imageLocation : trees) {
                    g.drawImage(imageLocation.image, imageLocation.x, imageLocation.y, null);
                }

            }
            g.dispose();

            ImageIO.write(base, "png", new File(destinationDirectory,
                                    sourceDirectory.getName() + counter + ".png"));

        }
    }
```

3- **Speculative Generality** – the code isn't used, and it occupies 1/3 of the class. In Agile Development, there should only be the code we currently use. Removing it will improve code clarity

In: src/net/sf/freecol/tools/GenerateDocumentation.java

```
/* Currently unused

  private static void generateTMX() {

      Map<String, Map<String, String>> translations = new HashMap<>();

      for (String name : sourceFiles) {

          System.out.println("Processing source file: " + name);

          String languageCode = name.substring(15, name.length() - 11);
          if (languageCode.isEmpty()) {
              languageCode = "en";
          } else if ('_' == languageCode.charAt(0)) {
              languageCode = languageCode.substring(1);
          } else {
              // don't know what to do
              continue;
          }
(…more commented code)
              out.write("  <tu tuid=\"" + tu.getKey() + "\">\n");
              for (Map.Entry<String, String> tuv : tu.getValue().entrySet()) {
                  out.write("    <tuv xml:lang=\"" + tuv.getKey() + "\">\n");
                  out.write("      <seg>" + tuv.getValue() + "</seg>\n");
                  out.write("    </tuv>\n");
              }
              out.write("  </tu>\n");
          }
          out.write("</body>\n");
          out.write("</tmx>\n");
          out.flush();
      } catch (Exception e) {
          e.printStackTrace();
      }
  }
  */
```

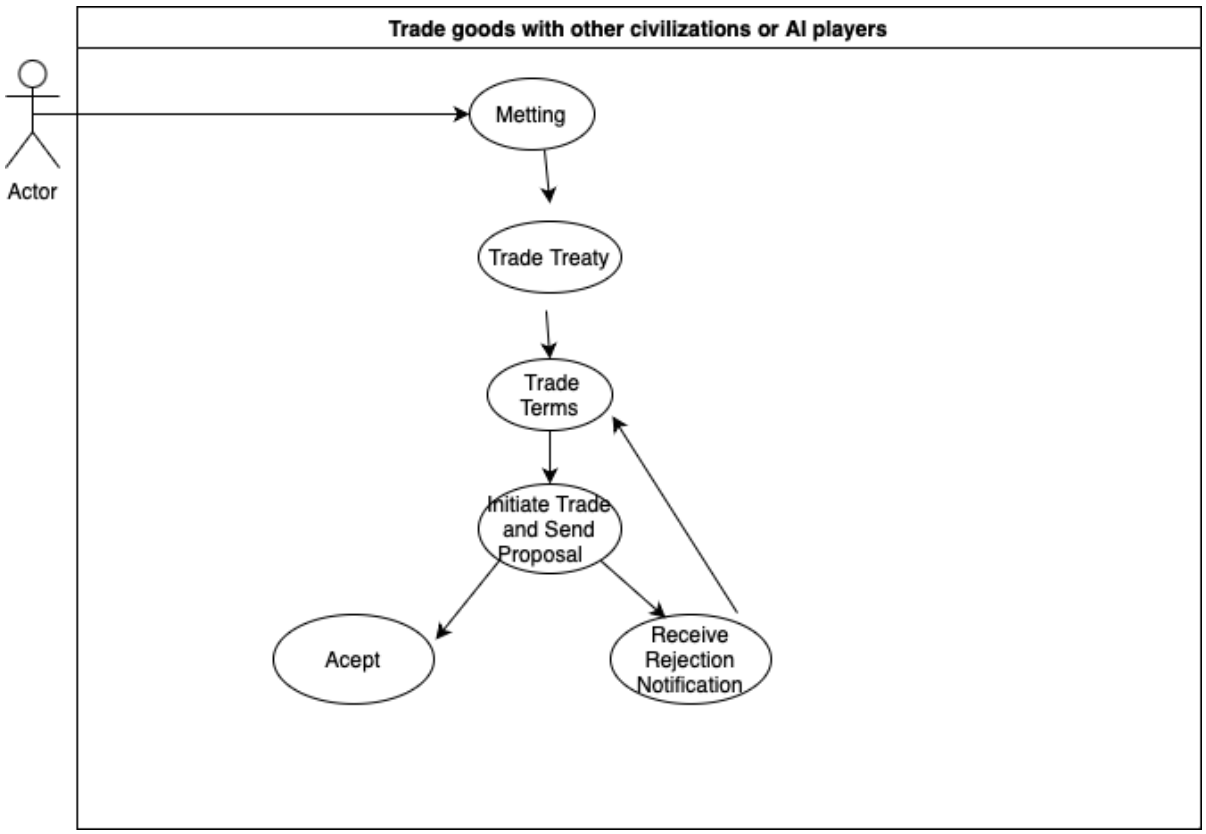
Review(s): Wilker Martins 58535

Comment: The assessment provides valuable insights into the codebase. The observation that the LootSession class may be categorized as a "Data Class" is accurate. This classification highlights that the class primarily holds data without significant functionality, which can lead to a more streamlined and efficient design.

Similarly, the recognition of a "Long Method" issue in the main method of ForestMaker.java is crucial. Long methods can indeed introduce challenges in code maintenance and comprehension, ultimately affecting the overall code quality. The recommendation to address this issue through code refactoring aligns with best practices for code organization and readability.

Wilker Martins - 58535
Use Case Diagram:

Use case: trade goods with other civilizations or AI players



Primary Actor: Player

Summary: This use case describes the process of trading goods between the player and other civilizations or AI-controlled players in the FreeCol game.

Preconditions:

The player must have settlers or units that have established contact with other civilizations or AI players on the map.

Main Flow:

The player selects a colony or unit that has established contact with another civilization or AI player. The player opens the trade window. The player views two sections in the trade window: their inventory and the other party's inventory. The player selects the goods they want to offer in the "Offer" section and the goods they want to request in the "Request" section. Goods have assigned values. The player adjusts the quantities and types of goods offered and requested to reach an agreement on the total trade value. The player initiates the

trade and sends the proposed terms to the other party. The other party receives the trade proposal and has the option to accept it, reject it, or make a counteroffer. If the other party accepts the proposal, the goods are exchanged, and the trade is successfully completed. If the other party rejects the proposal, the player can adjust the terms and try again.

Alternative Flow:

6a. The other party rejects the trade proposal.

The player receives a notification that the proposal was rejected. The player can adjust the trade terms and try again or abandon the trade.

Postconditions:

If the trade is successful, the exchanged goods are transferred between the player and the other party. The player can use the acquired goods in the trade to improve their colonies, expand their territory, and achieve their goals in the game. This use case describes the process of trading goods with other civilizations or AI players in FreeCol, highlighting the steps and possible alternative scenarios that may occur during the trade.

Review(s): Review (08/11/2023 - Diogo Lemos):

Spelling error in diagram: "Meeting".

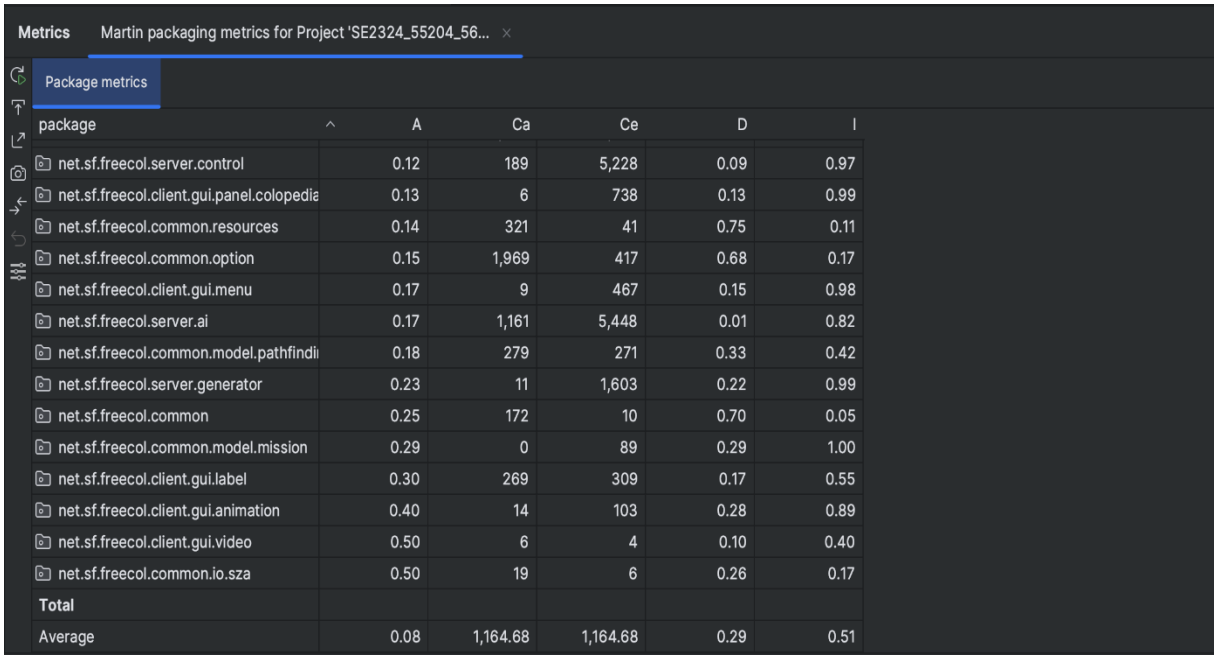
Missing secondary actor (Player2 or AI).

Everything else seems alright.

Code Base Metrics:

Martin Packaging Metrics

Project metrics:



Metrics | Martin packaging metrics for Project 'SE2324_55204_56...

Package metrics

package	A	Ca	Ce	D	I
net.sf.freecol.server.control	0.12	189	5,228	0.09	0.97
net.sf.freecol.client.gui.panel.colopedia	0.13	6	738	0.13	0.99
net.sf.freecol.common.resources	0.14	321	41	0.75	0.11
net.sf.freecol.common.option	0.15	1,969	417	0.68	0.17
net.sf.freecol.client.gui.menu	0.17	9	467	0.15	0.98
net.sf.freecol.server.ai	0.17	1,161	5,448	0.01	0.82
net.sf.freecol.common.model.pathfindi	0.18	279	271	0.33	0.42
net.sf.freecol.server.generator	0.23	11	1,603	0.22	0.99
net.sf.freecol.common	0.25	172	10	0.70	0.05
net.sf.freecol.common.model.mission	0.29	0	89	0.29	1.00
net.sf.freecol.client.gui.label	0.30	269	309	0.17	0.55
net.sf.freecol.client.gui.animation	0.40	14	103	0.28	0.89
net.sf.freecol.client.gui.video	0.50	6	4	0.10	0.40
net.sf.freecol.common.io.sza	0.50	19	6	0.26	0.17
Total					
Average	0.08	1,164.68	1,164.68	0.29	0.51

- **A (Abstractness)**: Abstractness is a metric used in software design and architecture to measure the degree to which a module or component is abstract or high-level. Modules with high abstractness contain more general, reusable code, while those with low abstractness are more concrete and specific to a particular implementation. High abstractness is often associated with better design and maintainability.
- **Ca (Afferent Couplings)**: Afferent couplings, also known as incoming dependencies, measure the number of other modules that depend on a particular module. High afferent couplings indicate that many other modules rely on the module in question, which can lead to increased complexity and potential challenges during maintenance.
- **Ce (Efferent Couplings)**: Efferent couplings, or outgoing dependencies, measure the number of other modules that a particular module depends on. High efferent couplings suggest that a module relies on many other modules, which can make it less cohesive and more tightly coupled with the system.
- **D (Distance from the Main Sequence)**: The distance from the main sequence is a metric used to assess the balance between abstractness (A) and instability (I) in software design. It helps determine whether a module is well-placed in terms of its level of abstraction and its dependencies on other modules. Modules that are too far from the main sequence are often considered problematic in terms of design quality.
- **I (Instability)**: Instability is a metric that quantifies the balance between a module's incoming dependencies (afferent couplings, Ca) and outgoing dependencies (efferent couplings, Ce). It helps to evaluate whether a module is more stable (few

dependencies and many dependents) or more volatile (many dependencies and few dependents). An ideal module should strike a balance between these two factors.

Example of a method with high Afferent Couplings

An elevated number of Afferent Couplings (Ca) can be a source of code smells because it signifies that a particular module or component is heavily depended upon by other modules in the codebase

Example of a method with high Efferent Couplings

A high number of Efferent Couplings indicates that a module is heavily dependent on other modules, potentially leading to code smells related to low cohesion, a ripple effect of changes, dependency management issues, and reduced encapsulation.

Example of a method with high Distance from the main Sequence

A significant distance from the main sequence suggests an imbalance between abstractness and instability, which can lead to code smells associated with overly abstract modules, fragility, complex dependencies, and violations of design principles

Review(s): Review (08/11/2023 - Diogo Lemos):

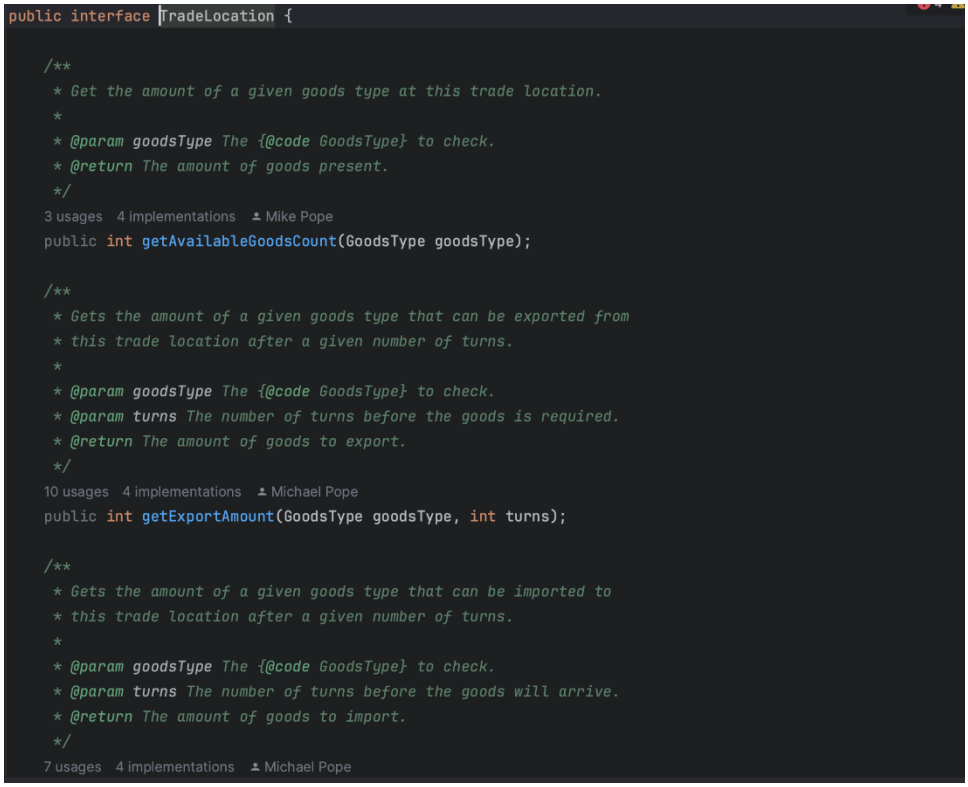
The metrics of the Martin Packaging are well explained and the relations between unacceptable metric values and code smells are well executed.

The results from the FreeCol code metrics are well interpreted.

GOF Patterns:

Design Patterns

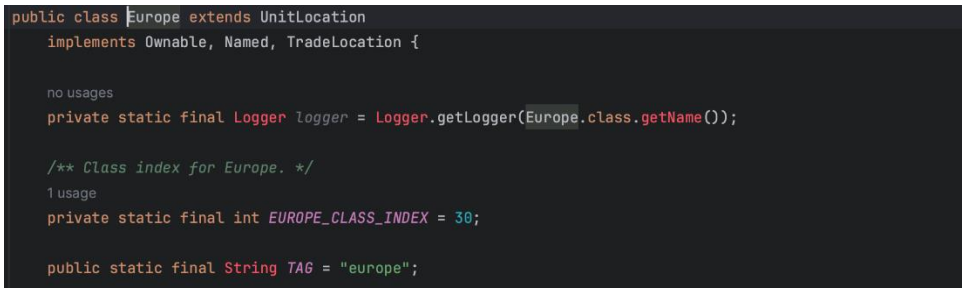
1-Factory Method (src/net/sf/freecol/common/model/TradeLocation.java)



```java
public interface TradeLocation {

    /**
     * Get the amount of a given goods type at this trade location.
     *
     * @param goodsType The {@code GoodsType} to check.
     * @return The amount of goods present.
     */
    3 usages  4 implementations  Mike Pope
    public int getAvailableGoodsCount(GoodsType goodsType);

    /**
     * Gets the amount of a given goods type that can be exported from
     * this trade location after a given number of turns.
     *
     * @param goodsType The {@code GoodsType} to check.
     * @param turns The number of turns before the goods is required.
     * @return The amount of goods to export.
     */
    10 usages  4 implementations  Michael Pope
    public int getExportAmount(GoodsType goodsType, int turns);

    /**
     * Gets the amount of a given goods type that can be imported to
     * this trade location after a given number of turns.
     *
     * @param goodsType The {@code GoodsType} to check.
     * @param turns The number of turns before the goods will arrive.
     * @return The amount of goods to import.
     */
    7 usages  4 implementations  Michael Pope
```

Classes that implements the interface

(src/net/sf/freecol/common/model/Europe.java)



```java
public class Europe extends UnitLocation
    implements Ownable, Named, TradeLocation {

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

    /** Class index for Europe. */
    1 usage
    private static final int EUROPE_CLASS_INDEX = 30;

    public static final String TAG = "europe";
```

(src/net/sf/freecol/common/model/IndianSettlement.java)

```java
public class IndianSettlement extends Settlement implements TradeLocation {

    5 usages
    private static final Logger logger = Logger.getLogger(IndianSettlement.class.getName());

    public static final String TAG = "indianSettlement";

    /** The level of contact between a player and this settlement. */
    22 usages  Mike Pope
    public static enum ContactLevel {
        UNCONTACTED,     // Nothing known other than location?
        2 usages
        CONTACTED,       // Name, wanted-goods now visible
        4 usages
        VISITED,         // Skill now known
        4 usages
        SCOUTED          // Scouting bonus consumed
    };
```

In the **TradeLocation** interface, you're abstracting the process of creating **TradeLocation** objects. This abstraction allows you to decouple the client code from the concrete classes (**European** and **IndianSettlement**) that implement the interface.

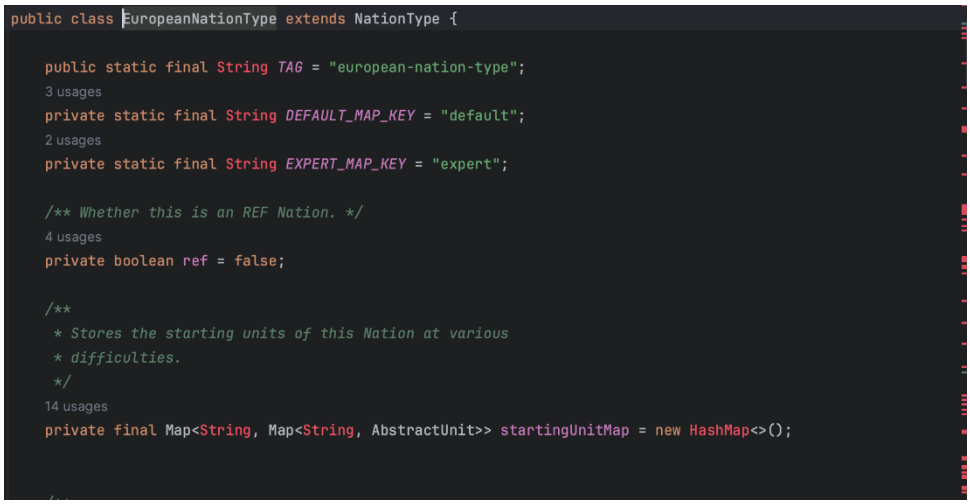
2-Template Method Pattern

(src/net/sf/freecol/common/model/NationType.java)

```java
public abstract class NationType extends FreeColSpecObjectType {

    4 usages  Mike Pope
    public static enum SettlementNumber {
        2 usages
        LOW, AVERAGE, HIGH;

        /**
         * Get a message key for this settlement number.
         *
         * @return A message key.
         */
        Mike Pope
        public String getKey() { return "settlementNumber." + getEnumKey( value: this); }
    }

    4 usages  Mike Pope
    public static enum AggressionLevel {
        no usages
        LOW, AVERAGE, HIGH;

        /**
         * Get a message key for this aggression level.
```

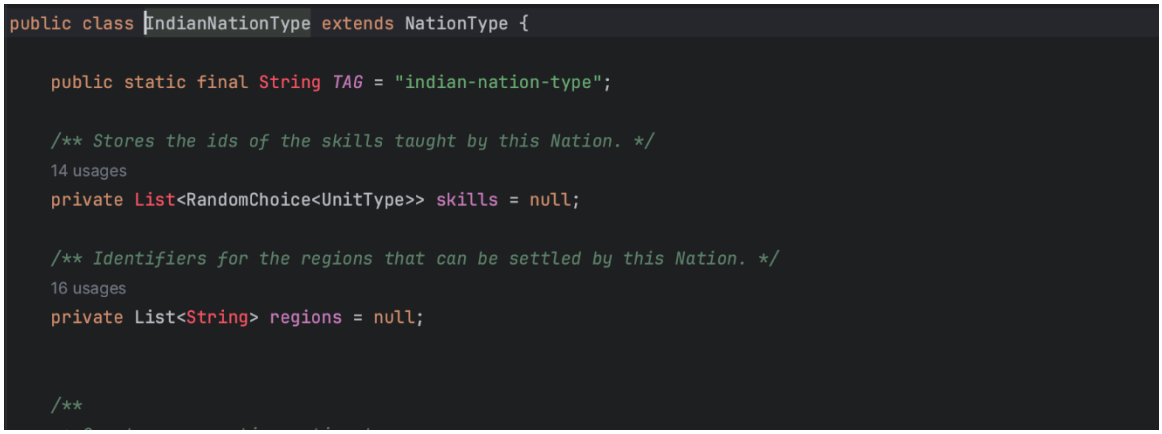
Classes that Extend the Super Class NationType

(src/net/sf/freecol/common/model/EuropeanNationType.java)



```
public class EuropeanNationType extends NationType {

    public static final String TAG = "european-nation-type";
    3 usages
    private static final String DEFAULT_MAP_KEY = "default";
    2 usages
    private static final String EXPERT_MAP_KEY = "expert";

    /** Whether this is an REF Nation. */
    4 usages
    private boolean ref = false;

    /**
     * Stores the starting units of this Nation at various
     * difficulties.
     */
    14 usages
    private final Map<String, Map<String, AbstractUnit>> startingUnitMap = new HashMap<>();
```

(src/net/sf/freecol/common/model/IndianNationType.java)



```
public class IndianNationType extends NationType {

    public static final String TAG = "indian-nation-type";

    /** Stores the ids of the skills taught by this Nation. */
    14 usages
    private List<RandomChoice<UnitType>> skills = null;

    /** Identifiers for the regions that can be settled by this Nation. */
    16 usages
    private List<String> regions = null;


    /**
```

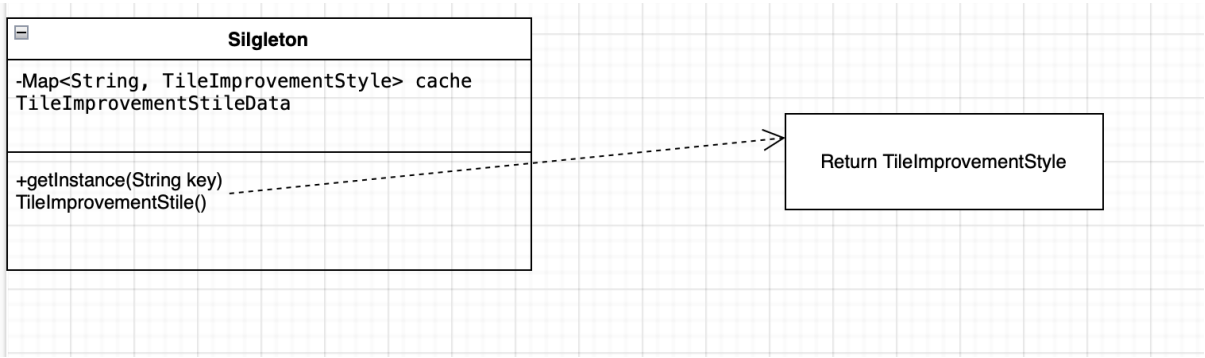
By defining a template method in the abstract class **NationType**, you provide a common structure for creating different nation types. This common structure contains the algorithm's skeleton, including the sequence of steps required to create a nation type. This structure is reused by all subclasses, eliminating the need to duplicate code.

3-Singleton

(src/net/sf/freecol/common/model/TileImprovementStyle.java)

```java
public class TileImprovementStyle {

    /** Cache all TileImprovementStyles. */
    3 usages
    private static final Map<String, TileImprovementStyle> cache = new HashMap<>();

    /** A key for the tile improvement style. */
    3 usages
    private final String style;

    /** A key for the forest overlay, derived from the above. */
    2 usages
    private final String mask;


    /**
     * Private constructor, only called in getInstance() below.
     *
     * @param style The (decoded) style.
     */
    1 usage   Mike Pope
    private TileImprovementStyle(String style) {
        this.style = style;

        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < style.length(); i++) {
            char c = style.charAt(i);
            if (Character.digit(c, Character.MAX_RADIX) < 0) break;
            sb.append((c == '0') ? "0" : "1");
```

The **TileImprovementStyle** class has a private constructor, which means it cannot be instantiated directly from outside the class. The class maintains a cache (represented as the **Map** named **cache**) to store already created instances of the **TileImprovementStyle** class.



Review(s): Reviewed by José Pereira:

The first design pattern appears to be correctly assigned as a Factory Design Pattern, since we have a generic interface from which its subclasses redefine the object, in this case the unit's location, each subclass manages its object (location) independently from the other subclasses.

The second design pattern also seems to be correctly identified as a Template Method, since we have an abstract class that defines a general skeleton and then we specify the "algorithm", in this case a particular Nation in each subclass.

The third and last pattern, is also correctly identified as a Singleton since we have a private constructor implying that the class can only be instantiated by itself.

Code Smells:

Code Smells:

1-Message Chain (src/net/sf/freecol/common/util/Utils.java)

```
final int highestScreenSize = Arrays.stream(displayModes).map(dm -> dm.getHeight()).max(Integer::compare).get();
```

There are a series of method calls or property accesses chained together, typically separated by dots. To refactor the code and avoid the message chain, you can break it down into separate steps using meaningful variable names. This will make the code more readable and easier to understand.

2-Large Classe (src/net/sf/freecol/common/util/CollectionUtils.java)

```
private static final double PRODUCT_DEFAULT = 1.0;

/** Trivial integer accumulator. */
4 usages
public static final BinaryOperator<Integer> integerAccumulator
    = (i1, i2) -> i1 + i2;

/** Trivial double accumulator. */
1 usage
public static final BinaryOperator<Double> doubleAccumulator
    = (d1, d2) -> d1 + d2;

/** Useful comparators for mapEntriesBy* */
3 usages
public static final Comparator<Integer> ascendingIntegerComparator
    = Comparator.comparingInt(Integer::intValue);
5 usages
public static final Comparator<Integer> descendingIntegerComparator
    = ascendingIntegerComparator.reversed();
2 usages
public static final Comparator<Double> ascendingDoubleComparator
    = Comparator.comparingDouble(Double::doubleValue);
2 usages
public static final Comparator<Double> descendingDoubleComparator
    = ascendingDoubleComparator.reversed();
2 usages
public static final Comparator<List<?>> ascendingListLengthComparator
    = Comparator.comparingInt(List::size);
2 usages
public static final Comparator<List<?>> descendingListLengthComparator
    = ascendingListLengthComparator.reversed();
```

There are a situation where a class has grown too large and complex, making it difficult to understand, maintain, and modify. By breaking down a large class into smaller, focused, and cohesive classes, we can improve code maintainability, readability, and reusability.

3-Long Method (src/net/sf/freecol/common/resources/FAFile.java)

```
private void load(InputStream is) throws IOException {
    letters.clear();

    BufferedReader in = new BufferedReader(new InputStreamReader(is,
            StandardCharsets.UTF_8));
    String line = in.readLine();
    if (line == null || !line.startsWith("FontAnimationFile")) {
        throw new RuntimeException("Not a FAF: " + this);
    }

    line = in.readLine();
    if (line == null) {
        throw new RuntimeException("Max height expected: " + this);
    }
    StringTokenizer st = new StringTokenizer(line);
    maxHeight = Integer.parseInt(st.nextToken());

    line = in.readLine();
    while (line != null && !line.startsWith("[Chars]")) {
        String name = line;
        if ((line = in.readLine()) == null) break;
        st = new StringTokenizer(line);
        int width = Integer.parseInt(st.nextToken());
        int height = Integer.parseInt(st.nextToken());
        int numberOfPoints = Integer.parseInt(st.nextToken());
        int[] xs = new int[numberOfPoints];
        int[] ys = new int[numberOfPoints];

        if ((line = in.readLine()) == null) break;
        st = new StringTokenizer(line);
        for (int i=0; i<numberOfPoints; i++) {
```

...

The method is overly long and complex. This can make the code harder to read, understand, and maintain. To address this code smell, we can break down the method into smaller, more focused methods, each responsible for a specific part of the overall process.

For example:

```
private void load(InputStream is) throws IOException {
    letters.clear();
    BufferedReader in = new BufferedReader(new InputStreamReader(is, Standar
    String line = in.readLine();

    if (line == null || !line.startsWith("FontAnimationFile")) {
        throw new RuntimeException("Not a FAF: " + this);
    }

    line = readMaxHeight(in);
    line = readCharacters(in, line);
    readLetters(in, line);
}
```

```
private String readMaxHeight(BufferedReader in) throws IOException {
    String line = in.readLine();
    if (line == null) {
        throw new RuntimeException("Max height expected: " + this);
    }

    StringTokenizer st = new StringTokenizer(line);
    maxHeight = Integer.parseInt(st.nextToken());
    return in.readLine();
}
```

```
private FAName createFAName(int width, int height, int numberOfPoints, int[]
    FAName newLetter = new FAName();
    newLetter.width = width;
    newLetter.height = height;
    newLetter.points = new Point[numberOfPoints];
    for (int i = 0; i < numberOfPoints; i++) {
        newLetter.points[i] = new Point(xs[i], ys[i]);
    }
    return newLetter;
}

private FALetter createFALetter(int advance, int numberOfPoints, int[] xs, i
    FALetter newLetter = new FALetter();
    newLetter.advance = advance;
    newLetter.points = new Point[numberOfPoints];
    for (int i = 0; i < numberOfPoints; i++) {
        newLetter.points[i] = new Point(xs[i], ys[i]);
    }
    return newLetter;
}
```

The **load** method has been split into smaller methods with clear responsibilities. This makes the code more organized, easier to read, and simpler to maintain.

Review(s): REVIEWER: José Trigueiro 58119 -

1- Message Chains: Yes, it's a Message Chain code smell, not the longest I have seen haha but it's one. And yes, the code should be broken down into separate steps as said. The Code smell is ok, but it only happens once in that class so it's not that bad, I guess?

2- Large Class: Yes, it's a Large Class code smell, it has 2606 lines!!! I agree to all you have said but you could give more tips on how to break it down by showing examples.

3- Long Method: Yes, it can be considered a Long Method but it's not the worst I have seen, it's 80 lines long, there are methods way bigger needing refactoring. I like and agree with the suggestion of code break down as given in the example.

User Stories

- ## User Story 1

  As a new user I want a brief tutorial to get me started so that I don't waste a lot of time trying to figure out how to play.

- ## User Story 2

  As a more experienced player, I want the gameplay to feel more realistic by introducing seasons to the game, so it affects the gameplay depending on both the current season and also the game’s difficulty.

- ## User Story 3

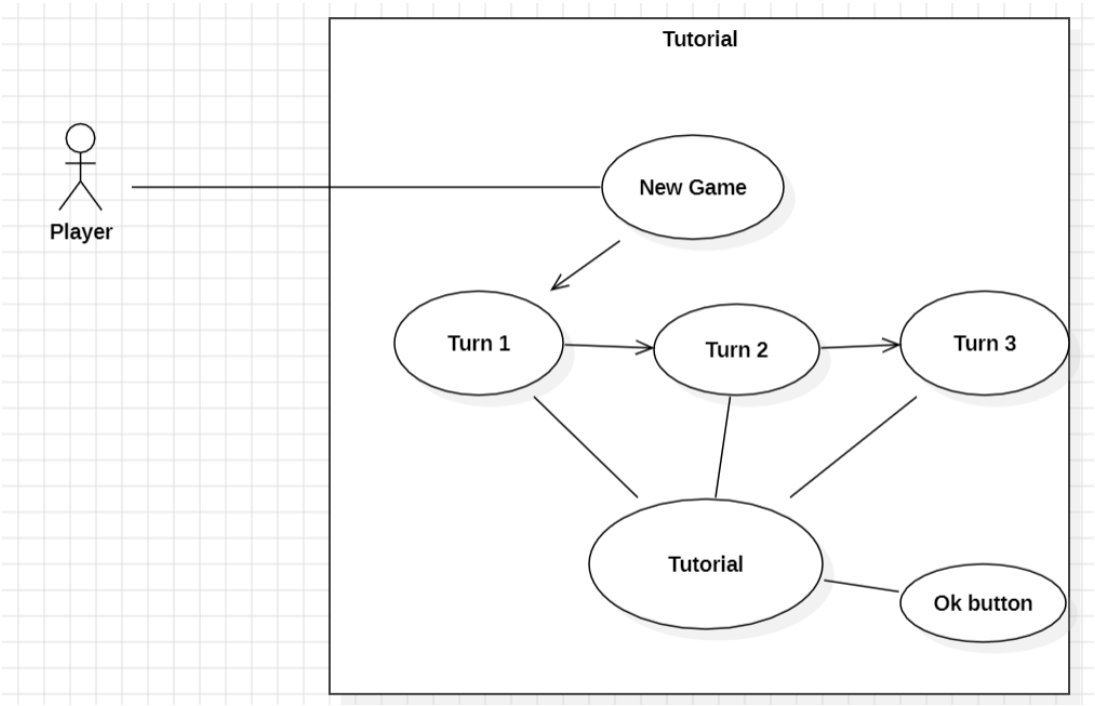
  As a user I want the transition between turns to be faster and smoother so that the game feels more enjoyable to play.

User Stories Use Case:

- ## User Story 1 (Tutorial)

Liliane Correia - 58427

Wilker Martins - 58535



Primary Actor: Player.

Secondary Actor: None.

Description:

Upon starting the game, the player encounters a brief tutorial guiding them through the basics of navigating either the boat or the character, depending on the starting point. This tutorial provides essential information on movement controls and basic interactions.

As the game progresses to turn 2, a tutorial emerges to elucidate the functionalities of the map. The Player receives guidance on how to explore the map, understand key locations, and utilize navigation tools effectively.

Moving into turn 3, an informative tutorial unfolds, shedding light on each icon present in the bottom bar of the screen. The Player gains insights into the significance and functionality of each icon, empowering them to make informed decisions during gameplay.

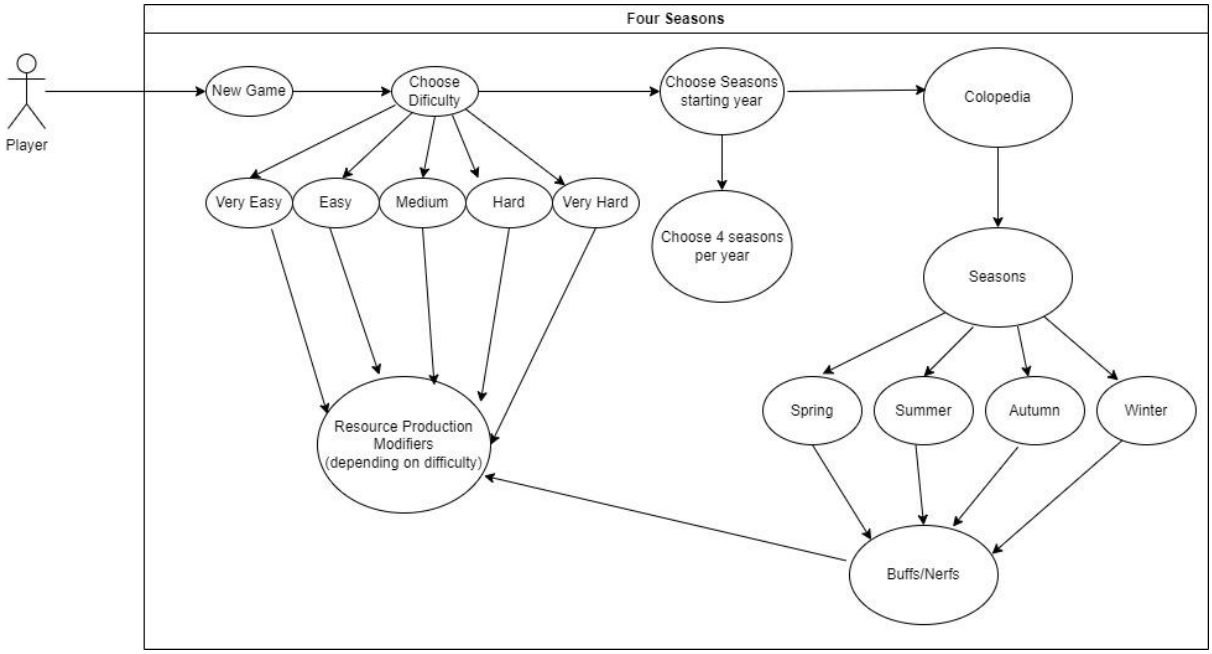
This sequential tutorial approach ensures that the Player gradually acquires a comprehensive understanding of the game mechanics, enabling a smoother and more enjoyable gaming experience.

- # User Story 2 (Four Seasons):

José Ricardo M. Pereira - 55204

Gustavo Silva – 59472

Use Case Diagram:



Primary Actor: Player.

Secondary Actor: None.

Description:

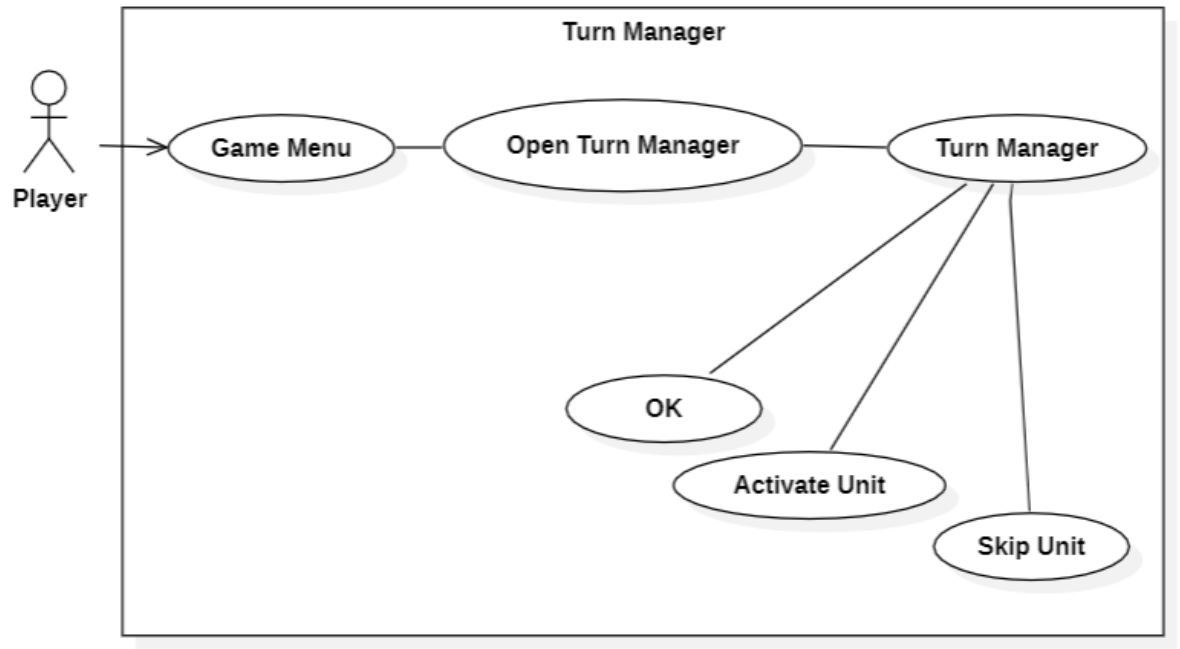
1. The Player, when starting a new game, can opt to choose between five game difficulties in the Game Settings.
2. The Player can then choose to alter the year where seasons are introduced to the game. In order to play with our complete feature, the player also has to select 4 seasons per year (Spring, Summer, Autumn, Winter).
3. The player can then check inside Colopedia's tab a new entry called "Seasons" where it showcases the effects each season of the year has when it comes to Resource Production.

- # User Story 3 (Turn Manager):

Diogo Lemos -56837

José Trigueiro – 58119

Use Case Diagram:



Primary Actor: Player.

Secondary Actor: None.

Description:

1. The Player opens the *Turn Manager* tab in the Game Menu Bar in the top left corner of the screen.
2. The Player selects the *Open Turn Manager* option that surges from the Turn Manager tab.
3. Then it will be presented with the *Turn Manager* window where it will have its deployed units listed, each one having its name, description and move count displayed on the screen.
4. The Player can then select one of the listed units to either *Skip* or *Activate* it, based on its current state. The active units are displayed with colors, while the skipped units are in shades of gray.
5. The Player can close the *Turn Manager* window by pressing the *OK* button.

Demo Video:

https://youtu.be/6GWg7HczET8