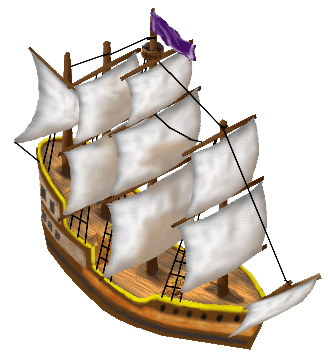
ENGENHARIA DE SOFTWARE

Licenciatura em Engenharia Informática



Freecol

Final delivery

Miguel Matos Barreto, nº61891

Índice

[Introduction 4](#_Toc152366543)

[1st Phase 5](#_Toc152366544)

[User Stories 5](#_Toc152366545)

[1st User Story 5](#_Toc152366546)

[2nd User Story 5](#_Toc152366547)

[3rd User Story 5](#_Toc152366548)

[2nd Phase 6](#_Toc152366549)

[Code Metrics 6](#_Toc152366550)

[Lines of Code Metric 6](#_Toc152366551)

[Design Patterns 7](#_Toc152366552)

[Code Smells 9](#_Toc152366553)

[Use Case Diagram 16](#_Toc152366554)

[Final Phase 17](#_Toc152366555)

# Introduction

The goal of this project is to work as a team and develop good working environment to improve an open source project called Freecol.

Our group had a split After the 2nd deliverable was delivered.

We forked the repository and continued our work, our could not develop the 3 user stories that were defined while we were in the previous group, the tutorial mission that is implemented in our project is code that belongs to the previous group came when we forked the code, it has the beggining of the tutorial logic, and works for the first mission.

Our group develop the 1st user story funcionality:

* As a user, I wish the game to include special tiles with unique effects to make the gameplay more varied and strategic.

We implemented a funcionality where when you get a unit move into a river tile it triggers an event, the event has 85% chance to fail and 15% chance to give you a random amount of gold between 0 and 145.

# 1st Phase

## User Stories

The user stories were made in the previous group and when we forked the project we kept them.

### 1st User Story

As a user I wish the game to include special tiles with unique effects to make the gameplay more varied and strategic.

### 2nd User Story

As a player I want the ability to deepen my interactions with the native characters in the game to enrich the narrative

### 3rd User Story

As a new player I want a set of starting missions to provide me with essential information and tips, so I can so I can quickly grasp the basic gameplay concepts without feeling overwhelmed.

# 2nd Phase

## Code Metrics

#### Miguel Barreto, nº 61891, mm-barreto

### Lines of Code Metric

The Lines of Code metric measures the total number of lines within different parts of a codebase. It's a quantitative measure used to express the size of a codebase.

It includes the following metrics:

* CLOC (Comment Lines of Code): The number of lines of code which are comments. Used to assess documentation quality within the code.
* JLOC (Java Lines of Code): Specific to Java, this metric counts the lines of Java code.
* LOC (Lines of Code): The total number of lines in a code segment, including comments and whitespace.
* NCLOC (Non-Comment Lines of Code): The number of lines of actual code, excluding comments and blank lines.
* RLOC (Relative Lines of Code): Indicates the proportion of the total lines of code that are actual, non-comment code.

A pie chart with different colors

Description automatically generatedAfter running the plugin on our codebase and looking specifically at the Non-Comment Lines of Code (NCLOC) metric by package, we can visualize the data with the following pie chart:

Upon inspecting the pie chart, we can observe that **net.sf.freecol.common.model** has a larger segment, suggesting that it contains more lines of code compared to other packages.

This could potentially be a trouble spot.

Packages with large amounts of code can be harder to maintain and understand and be more prone to bugs.

It relates to code smells like Long Methods and Long Classes.

#### Diogo Correia, nº 62475, Correia21

Attribute Hiding Factor (AHF):

So, the Attribute Hiding Factor (AHF) is like the undercover boss of your class attributes. It checks how well your class keeps its private stuff, like instance variables, away from the nosy external classes. A top-notch AHF, aiming for that cool 100%, means your class is the master of disguise, hiding its attributes like a pro. It's all about encapsulation, making sure your code stays tidy and easy to build upon.

Attribute Inheritance Factor (AIF):

Alright, let's talk about the Attribute Inheritance Factor (AIF). This one's all about the family ties between parent and child classes. A high AIF, cruising up to 48%, suggests a bit of a family drama with lots of attributes being passed down. But hold on, too much drama isn't always good. Keeping AIF on the down-low is the trick, lessening the dependency between classes for a smoother coding experience.

Coupling Factor (CF):

Now, the Coupling Factor (CF) is like the relationship status between your classes. A low CF is like saying, "We're just hanging out, no strings attached." Loose coupling means easy maintenance and reusable code – a coder's dream. But beware, a high CF screams "It's complicated," signaling tightly woven classes that could make changes a headache. Keeping it low is the key to a drama-free coding life.

Method Hiding Factor (MHF):

Picture the Method Hiding Factor (MHF) as your class's security guard for methods. A high MHF, hitting that sweet spot between 8% and 25%, means your methods are VIPs protected from unauthorized access. On the flip side, a low MHF is like leaving your methods out in the open – not the best move. Striking that balance is crucial, ensuring a safe yet functional method party.

Method Inheritance Factor (MIF):

Let's chat about the Method Inheritance Factor (MIF). It's the measure of how much your methods like to travel from parent to child classes. High MIF can bring in some complexity and dependencies between classes. Going for a lower MIF is like keeping things chill – less inheritance, less drama. It's all about finding that sweet spot for a codebase that's easy on the eyes.

Polymorphism Factor (PF):

Lastly, the Polymorphism Factor (PF) is like the cool factor of your code. Embracing polymorphism is like letting different classes party together, and that's awesome for flexibility and extensibility. But hold up, too much polymorphic action can make your code a bit confusing. So, finding that sweet spot between clear code and polymorphic fun is the name of the game. Keep it cool, not chaotic.

## Design Patterns

#### Miguel Barreto, nº 61891, mm-barreto

GoF Patterns

**Singleton Pattern**

**Location:**

* **src/net/sf/freecol/tools/FSGConverter.java**

**Code snippet:**

A screenshot of a computer program

Description automatically generated

**Explanation:**

* The identification of this pattern is quite straightforward, we can easily identify the unique instance of the class and the lazy constructor, assuring that there is always only one instance of this class.

**Decorator Pattern**

**Location:**

* **src/net/sf/freecol/common/model/TileImprovementStyle.java**

**Code snippet:**

Uma imagem com texto, eletrónica, captura de ecrã, computador

Descrição gerada automaticamente

**Explanation:**

* The comments help us understand this pattern, it represents an additional feature of a tile, and it allow us to create different types of new features.

#### Diogo Correia, nº 62475, Correia21

Template Method Pattern in Action: net.sf.freecol.client.gui.action.FreeColAction

Delving into the realms of design patterns, the FreeColAction package unveils an instance of the Template Method Pattern. Serving as an abstract base class, FreeColAction meticulously defines a behavioral skeleton for diverse actions. It judiciously implements common methods while leaving the shouldBeEnabled method as a hook, inviting its subclasses to override and provide tailor-made behaviors. The ChatAction and DebugAction classes adeptly showcase this pattern in action, with ChatAction, a subclass, ingeniously overriding shouldBeEnabled to articulate specific logic for the chat action.

Observer Pattern Dynamics: net.sf.freecol.client.gui.dialog.CaptureGoodsDialog

Navigating through the intricacies of the CaptureGoodsDialog class, an implicit Observer Pattern surfaces. The linchpin of this pattern is the goodsList, a JList<GoodsItem> serving as the subject. While the absence of an explicit Observer or Observable interface might seem conspicuous, the magic unfolds through Java Swing's native methods and interfaces. The goodsList orchestrates the interaction by employing addMouseListener(MouseListener listener) and removeMouseListener(MouseListener listener) methods. These maneuvers seamlessly add or remove specific observers implementing the MouseListener interface. Ergo, the goodsList elegantly dons the role of the subject, orchestrating the symphony of notifications to observers (listeners) whenever mouse events dance across its domain.

Abstract Factory Harmony: src.net.sf.freecol.client.gui.option.LanguageOptionUi

The LanguageOptionUI class, ensconced within the src.net.sf.freecol.client.gui.option package, emerges as an embodiment of the Abstract Factory Pattern. Functioning as an abstract factory, it masterfully crafts objects intertwined with the language option domain. Witness the creation of a JComboBox<Language>, a pivotal component within the language option's UI repertoire. As LanguageOption symbolizes language preferences and Language encapsulates available languages, the Abstract Factory Pattern seamlessly orchestrates the creation of a cohesive family of objects aligned with the chosen

## Code Smells

#### Miguel Barreto, nº 61891, mm-barreto

*net.sf.freecol.FreeCol:*

Large Class:

The FreeCol class contains a large number of methods and properties, which might indicate that it's doing too many things.

Duplicated Code:

There are instances of duplicated code, such as similar error handling patterns found in different methods.

*Net.sf.freecol.server.generator.TerrainGenerator:*

Long Method:

Uma imagem com texto, captura de ecrã

Descrição gerada automaticamenteThe generateMap method is quite lengthy, performing multiple tasks such as importing tiles, setting regions, creating mountains, rivers, lakes, and bonuses.

Uma imagem com texto, captura de ecrã, software

Descrição gerada automaticamente

Uma imagem com texto, captura de ecrã, software

Descrição gerada automaticamente

Data Clumps:

There are multiple groups of related parameters used across methods, such as latitude-related parameters. These data clumps suggest that certain parameters might be better organized into objects or data structures, creating classes or structures to encapsulate related parameters, would make the code more organized and selfexplanatory.

Uma imagem com texto, captura de ecrã, software

Descrição gerada automaticamente

*For example, latitude is passed to*

*methods like getRandomLandTileType*

*and getRandomOceanTileType.*

*Net.sf.freecol.server.generator.SimpleMapGenerator:*

Long Method:

The createEuropeanUnits method is quite long and performs multiple tasks, including handling different types of units, selecting starting positions, and checking various conditions. Long methods can be hard to understand, maintain, and test. My suggestion is refactoring this method into smaller, more focused methods that handle specific tasks.

Uma imagem com texto, captura de ecrã, menu

Descrição gerada automaticamente

Uma imagem com texto, captura de ecrã, menu

Descrição gerada automaticamente

Uma imagem com texto, captura de ecrã, menu

Descrição gerada automaticamente

Uma imagem com texto, captura de ecrã, software

Descrição gerada automaticamente

Data Clumps:

There are instances of using groups of related data as method parameters, such as generateSkillForLocation taking Map, Tile, and NationType as parameters. This indicates a data clump, where certain groups of parameters are frequently passed together. Encapsulating related parameters into a class or structure to improve code readability and maintainability.

Uma imagem com texto, captura de ecrã, software

Descrição gerada automaticamente

#### Diogo Correia, nº 62475, Correia21

Enhancing Expressiveness:

Within the confines of the codebase, there's an intriguing instance of primitive obsession. Take, for example, the utilization of a primitive type like an integer to encapsulate the concept of save game periods within the autoSaveGame method in InGameController.java. A more nuanced approach would involve crafting dedicated classes, like a SaveGamePeriod class, to encapsulate such information. By doing so, not only does the code become more lucid, but it also opens avenues for robust validation mechanisms.

Refactoring Long Methods:

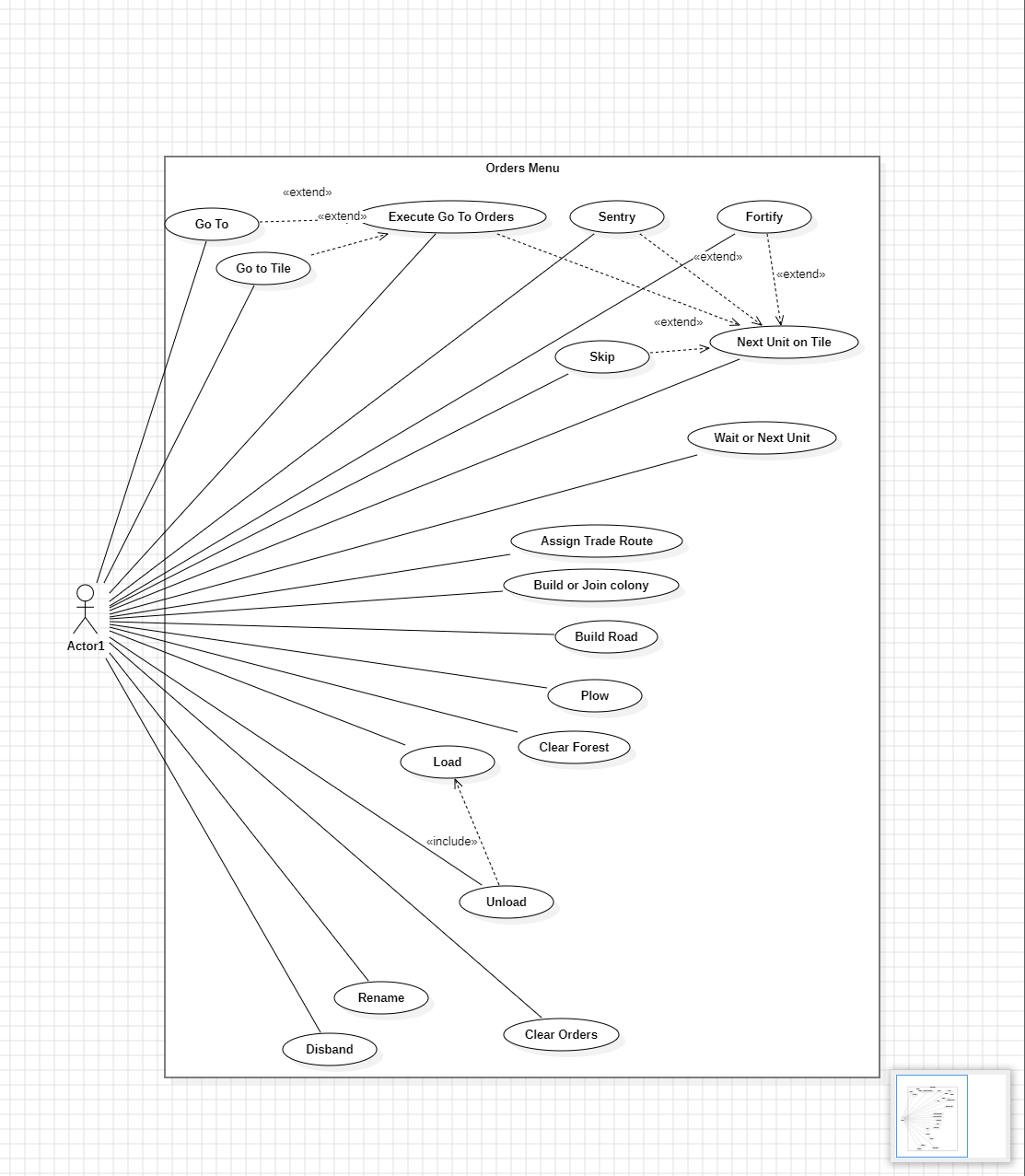
Nestled within the labyrinth of code, a lengthy method named moveDirection in InGameController.java has surfaced. This method weaves through an array of conditional checks and diverse actions. A judicious approach here involves the surgical division of this behemoth into smaller, more specialized methods. Such a stratagem not only elevates the readability of the code but also facilitates easier maintenance down the line.

Mitigating Duplicated Code:

An intriguing case of duplicated code manifests within the Flag class in the src.net.sf.freecol.client.gui.dialog package. Specifically, both the drawStripes and drawQuarters methods exhibit repetitive lines of code pertaining to g.setColor and rectangle.setRect. A pragmatic antidote to this redundancy conundrum is to fashion handy helper methods, effectively excising the duplicated snippets and fostering a more streamlined and maintainable codebase.

## Use Case Diagram

#### Miguel Barreto, nº 61891, mm-barreto



# Final Phase

On the 3rd phase our group was challenged with a group split, we split from the group and continued our group with only 2 members.

##### 1st User Story

We succeeded on implementing the 1st user story.

We developed a funcionality where when a unit crosses a river has 15% chance to trigger the event “you found gold in the river”.

[Demo video](https://www.youtube.com/watch?v=mbb_MfRe2C0&ab_channel=MiguelBarreto)

##### 2nd User Story

For the 2nd User Story

Our group tried to implemented a functionality where we could speak with natives that were close to a unit tile, we implemented A NativeRecruit class that would work like NativeTrade but to recruit natives, a UnitRecruitable that extends Recruitable (an abstract class created to make natives a recruitable Object. The abstract class extends FreeColGameObject and would work like the class TradeItem (that makes Goods become tradeable)) the UnitRecruitable Class would work like NativeTradeItem but to make units recruitable.

We could not make this work due to time shortage.

##### 3rd User Story

For the 3rd User Story

The 3rd user story was started in the old group by a non member of the new group, and all code made for the tutorials was not made by us, we kept the code because it existed at the moment of the fork.