# Es Project

# Merge Doc

**Github Repository:**

https://github.com/jmamorim/SE2324_57409_47994_53175_59457

**Link Vídeo: https://youtu.be/DW8JPIMI4HI**

**Authors:**

**João Amorim 57409**

**João Esteves 47994**

**Nádia Mendes 53175**

**José Morgado 59457**

# Index:

**User Stories: 3**

**Code Smells:**

**- João Amorim: 4-6**

**- João Esteves: 7-12**

**- Nádia Mendes: 13-18**

**- José Morgado: 19-20**

**Gof Patterns:**

**- João Amorim: 21-25**

**- Nádia Mendes: 26-30**

**- João Esteves: 31-34**

**- José Morgado: 35-40**

**Code Metrics:**

**- João Amorim: 41-43**

**- João Esteves: 44-51**

**- Nádia Mendes: 52-58**

**- José Morgado: 59-62**

**Use Case Diagram:**

**- João Amorim: 63-65**

**- Nádia Mendes: 66-68**

**- João Esteves: 69-71**

**- José Morgado: 72-75**

**Use cases for the User Stories: 76**

# User Stories:

**1º User Story**

As a user, I wish the game to include special tiles with unique effects to make the gameplay more varied and strategic like when entering a forest a event happens.

**2º User Story**

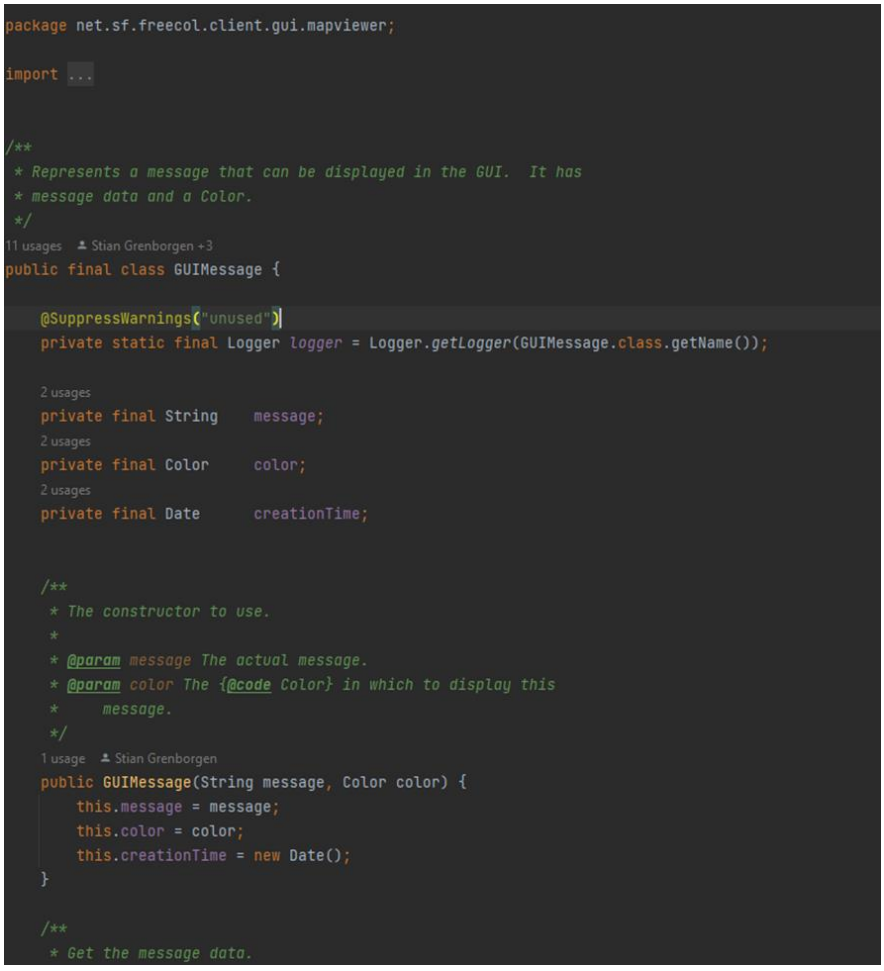
As a new player, I want a set of starting missions to provide me with essential

information and tips, so I can quickly grasp the basic gameplay concepts without feeling overwhelmed like missions that document milestones like first time moving, create a settlement, etc.

# Code Smells:

## João Amorim:

1º - Data Class - net.sf.freecol.client.gui.mapviewer.GUIMessage



```
package net.sf.freecol.client.gui.mapviewer;

import ...


/**
 * Represents a message that can be displayed in the GUI.  It has
 * message data and a Color.
 */
11 usages  Stian Grenborgen +3
public final class GUIMessage {

    @SuppressWarnings("unused")
    private static final Logger logger = Logger.getLogger(GUIMessage.class.getName());

    2 usages
    private final String    message;
    2 usages
    private final Color     color;
    2 usages
    private final Date      creationTime;


    /**
     * The constructor to use.
     *
     * @param message The actual message.
     * @param color The {@code Color} in which to display this
     *     message.
     */
    1 usage  Stian Grenborgen
    public GUIMessage(String message, Color color) {
        this.message = message;
        this.color = color;
        this.creationTime = new Date();
    }

    /**
     * Get the message data.
```

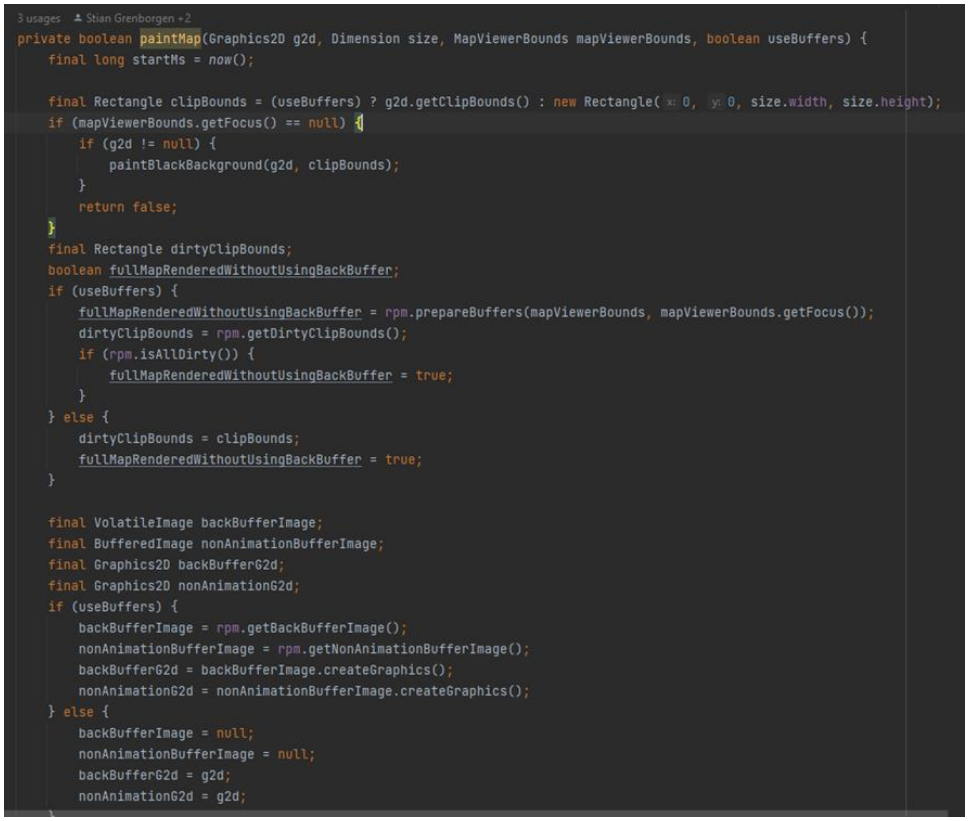
This class does not serve much purpose but to represent a message in a chat having tree variables and 3 getters for each not having any other functionality tackling strictly only data.

The way it looks a good idea for refactoring would be to develop this class more give more of a purpose then just contain data maybe there are

behaviours that are outside this class, like methods or even variables, that should be moved to this class.

## 2º - Long Method - net.sf.freecol.client.gui.mapviewer

## Method Paint Map



```
3 usages  Stian Grenborgen +2
private boolean paintMap(Graphics2D g2d, Dimension size, MapViewerBounds mapViewerBounds, boolean useBuffers) {
    final long startMs = now();

    final Rectangle clipBounds = (useBuffers) ? g2d.getClipBounds() : new Rectangle( x: 0,  y: 0, size.width, size.height);
    if (mapViewerBounds.getFocus() == null) {
        if (g2d != null) {
            paintBlackBackground(g2d, clipBounds);
        }
        return false;
    }
    final Rectangle dirtyClipBounds;
    boolean fullMapRenderedWithoutUsingBackBuffer;
    if (useBuffers) {
        fullMapRenderedWithoutUsingBackBuffer = rpm.prepareBuffers(mapViewerBounds, mapViewerBounds.getFocus());
        dirtyClipBounds = rpm.getDirtyClipBounds();
        if (rpm.isAllDirty()) {
            fullMapRenderedWithoutUsingBackBuffer = true;
        }
    } else {
        dirtyClipBounds = clipBounds;
        fullMapRenderedWithoutUsingBackBuffer = true;
    }

    final VolatileImage backBufferImage;
    final BufferedImage nonAnimationBufferImage;
    final Graphics2D backBufferG2d;
    final Graphics2D nonAnimationG2d;
    if (useBuffers) {
        backBufferImage = rpm.getBackBufferImage();
        nonAnimationBufferImage = rpm.getNonAnimationBufferImage();
        backBufferG2d = backBufferImage.createGraphics();
        nonAnimationG2d = nonAnimationBufferImage.createGraphics();
    } else {
        backBufferImage = null;
        nonAnimationBufferImage = null;
        backBufferG2d = g2d;
        nonAnimationG2d = g2d;
```

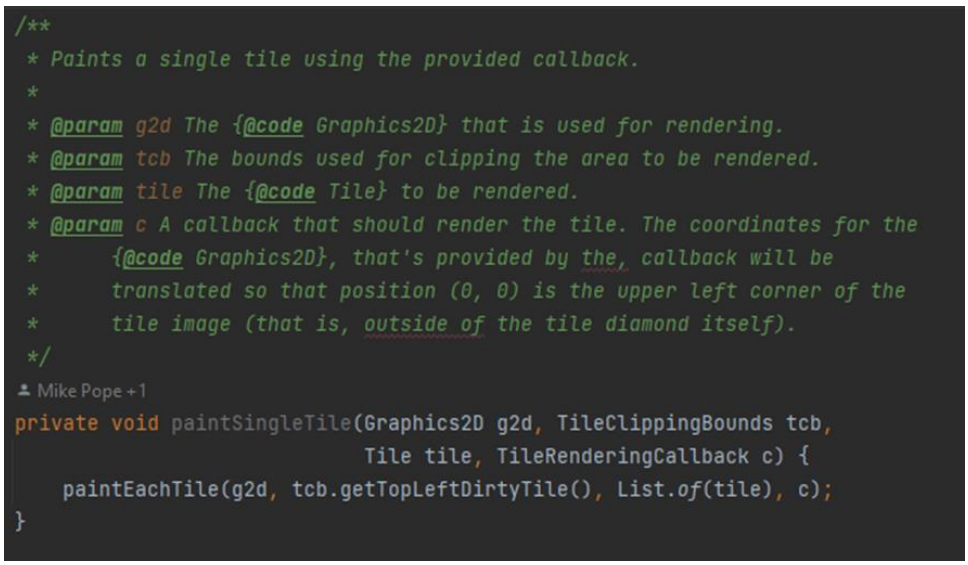
This method is just way to long making it way more complex, although it may be a method that tackles the graphical part of the game and its normal for a method with that job to be extensive and complex, the way this method looks with about 174 lines of code and to add insult to injury is not documented make it a smell.

My suggestion for a refactor would be to make it simpler and to better document the method itself and each step done in it, this should not be too arduous because there are already some comments in the method that document the steps being done. Now when I mean make it simpler this would be the steps that I mentioned before each should be turned into their own

method that is called inside the parent method and as I said already all these changes should be carefully documented having in mind the complexity of the method that we are looking at.

## 3º - Duplicated Code- net.sf.freecol.client.gui.mapviewer

## method paintSingleTile



```
/**
 * Paints a single tile using the provided callback.
 *
 * @param g2d The {@code Graphics2D} that is used for rendering.
 * @param tcb The bounds used for clipping the area to be rendered.
 * @param tile The {@code Tile} to be rendered.
 * @param c A callback that should render the tile. The coordinates for the
 *      {@code Graphics2D}, that's provided by the, callback will be
 *      translated so that position (0, 0) is the upper left corner of the
 *      tile image (that is, outside of the tile diamond itself).
 */
 Mike Pope +1
private void paintSingleTile(Graphics2D g2d, TileClippingBounds tcb,
                             Tile tile, TileRenderingCallback c) {
    paintEachTile(g2d, tcb.getTopLeftDirtyTile(), List.of(tile), c);
}
```

Although its well documented and the purpose of it is well understood, this method is not used anywhere and there is even alternative to it in the same class, so with what was just mentioned this method is useless there are no comments justifying the “why” it’s not used or even if there is an “when” is going to be used making redundant and just unnecessary complexity to the code base.

A good way to refactor this would be firstly to understand if there is actually a use for it, and this can be achieved but exploring the code base or even making a pull request with the changes and seeing the opinions of other collaborators, if there isn’t an use I would just remove it.

# João Esteves 47994:

**1-Primitive Obsession:**

In this class, we can see that primitive types (such as integers) are used to represent specific concepts. One possible solution would be to create specific classes for these concepts, making the code more expressive. For example, consider the int *saveGamePeriod* variable within the *autoSaveGame* method on line 886 of the *InGameController.java* class in the *src.net.sf.freecol.client.control* package. This variable is used to represent information like save game periods. By using an object type that encapsulates this information, such as a *SaveGamePeriod* class that we could create, we can make the code clearer and enable more robust validations.

```
    // conditional save after user-set period
    int saveGamePeriod = options.getInteger(ClientOptions.AUTOSAVE_PERIOD);
    int turnNumber = game.getTurn().getNumber();
    if (saveGamePeriod >= 1 && turnNumber % saveGamePeriod == 0) {
        String fileName = prefix + "-" + getSaveGameString(game);
        saveGame(FreeColDirectories.getAutosaveFile(fileName));
    }
}
```

Pic. 1. Part of the autoSaveGame method code where the entire saveGamePeriod is called.

**2-Long Method:**

The moveDirection method in the *InGameController.java* class located in the *src.net.sf.freecol.client.control* package, which starts at line 1315 and extends all the way to line 1496, is evidently a lengthy method. It's apparent that this method contains numerous conditional checks and performs various actions. Dividing this method into smaller, more specific methods would be a possible improvement, both in terms of code readability and code maintenance.

```
public boolean moveDirection(Unit unit, Direction direction,
boolean interactive) {
    // Is the unit on the brink of reaching the destination with
    // this move?
    final Location destination = unit.getDestination();
    final Tile oldTile = unit.getTile();
    boolean destinationImminent = destination != null
            && oldTile != null
            && Map.isSameLocation(oldTile.getNeighbourOrNull(direction),
            destination);

    // Consider all the move types.
    final Unit.MoveType mt = unit.getMoveType(direction);
    boolean result = mt.isLegal();
    switch (mt) {
        case MOVE_HIGH_SEAS:
            // If the destination is Europe (and valid) move there,
            // if the destination is null, ask what to do,
            // otherwise just move on the map.
            result = (destination instanceof Europe
                    && getMyPlayer().getEurope() != null)
                    ? moveTowardEurope(unit, (Europe)destination)

                    : (destination == null)
                    ? moveHighSeas(unit, direction)
                    : moveTile(unit, direction);
            break;
```

```
case MOVE:
    result = moveTile(unit, direction);
    break;
case EXPLORE_LOST_CITY_RUMOUR:
    result = moveExplore(unit, direction);
    break;
case ATTACK_UNIT:
    result = moveAttack(unit, direction);
    break;
case ATTACK_SETTLEMENT:
    result = moveAttackSettlement(unit, direction);
    break;
case EMBARK:
    result = moveEmbark(unit, direction);
    break;
case ENTER_INDIAN_SETTLEMENT_WITH_FREE_COLONIST:
    result = moveLearnSkill(unit, direction);
    break;
case ENTER_INDIAN_SETTLEMENT_WITH_SCOUT:
    result = moveScoutIndianSettlement(unit, direction);
    break;
case ENTER_INDIAN_SETTLEMENT_WITH_MISSIONARY:
    result = moveUseMissionary(unit, direction);
    break;
case ENTER_FOREIGN_COLONY_WITH_SCOUT:
    result = moveScoutColony(unit, direction);
    break;
```

```
case ENTER_SETTLEMENT_WITH_CARRIER_AND_GOODS:
    result = moveTrade(unit, direction);
    break;

// Illegal moves
case MOVE_NO_ACCESS_BEACHED:
    if (interactive || destinationImminent) {
        sound("sound.event.illegalMove");
        StringTemplate nation = getNationAt(unit.getTile(), direction);
        showInformationPanel(unit, StringTemplate
                .template("move.noAccessBeached")
                .addStringTemplate("%nation%", nation));
    }
    break;
case MOVE_NO_ACCESS_CONTACT:
    if (interactive || destinationImminent) {
        sound("sound.event.illegalMove");
        StringTemplate nation = getNationAt(unit.getTile(), direction);
        showInformationPanel(unit, StringTemplate
                .template("move.noAccessContact")
                .addStringTemplate("%nation%", nation));
    }
    break;
case MOVE_NO_ACCESS_GOODS:
    if (interactive || destinationImminent) {
        sound("sound.event.illegalMove");
        StringTemplate nation = getNationAt(unit.getTile(), direction);
        showInformationPanel(unit, StringTemplate
```

```
                .template("move.noAccessGoods")
                .addStringTemplate("%nation%", nation)
                .addStringTemplate("%unit%",
                        unit.getLabel(Unit.UnitLabelType.NATIONAL)));
    }
    break;
case MOVE_NO_ACCESS_LAND:
    if (!moveDisembark(unit, direction)) {
        if (interactive) {
            sound("sound.event.illegalMove");
        }
    }
    break;
case MOVE_NO_ACCESS_MISSION_BAN:
    if (interactive || destinationImminent) {
        sound("sound.event.illegalMove");
        StringTemplate nation = getNationAt(unit.getTile(), direction);
        showInformationPanel(unit, StringTemplate
                .template("move.noAccessMissionBan")
                .addStringTemplate("%unit%",
                        unit.getLabel(Unit.UnitLabelType.NATIONAL))
                .addStringTemplate("%nation%", nation));
    }
    break;
case MOVE_NO_ACCESS_SETTLEMENT:
    if (interactive || destinationImminent) {
        sound("sound.event.illegalMove");
```

```
                sound("sound.event.illegalMove");
                StringTemplate nation = getNationAt(unit.getTile(), direction);
                showInformationPanel(unit, StringTemplate
                        .template("move.noAccessSettlement")
                        .addStringTemplate("%unit%",
                                unit.getLabel(Unit.UnitLabelType.NATIONAL))
                        .addStringTemplate("%nation%", nation));
            }
            break;
        case MOVE_NO_ACCESS_SKILL:
            if (interactive || destinationImminent) {
                sound("sound.event.illegalMove");
                showInformationPanel(unit, StringTemplate
                        .template("move.noAccessSkill")
                        .addStringTemplate("%unit%",
                                unit.getLabel(Unit.UnitLabelType.NATIONAL)));
            }
            break;
        case MOVE_NO_ACCESS_TRADE:
            if (interactive || destinationImminent) {
                sound("sound.event.illegalMove");
                StringTemplate nation = getNationAt(unit.getTile(), direction);
                showInformationPanel(unit, StringTemplate
                        .template("move.noAccessTrade")
                        .addStringTemplate("%nation%", nation));
            }
            break;
```

```
            case MOVE_NO_ACCESS_WAR:
                if (interactive || destinationImminent) {
                    sound("sound.event.illegalMove");
                    StringTemplate nation = getNationAt(unit.getTile(), direction);
                    sh
case MOVE_NO_MOVES:
    // The unit may have some moves left, but not enough
    // to move to the next node.  The move is illegal
    // this turn, but might not be next turn, so do not cancel the
    // destination but set the state to skipped instead.
    destinationImminent = false;
    changeState(unit, UnitState.SKIPPED);
    break;
case MOVE_NO_TILE:
    if (interactive || destinationImminent) {
        sound("sound.event.illegalMove");
        showInformationPanel(unit, StringTemplate
            .template("move.noTile")
            .addStringTemplate("%unit%",
                unit.getLabel(Unit.UnitLabelType.NATIONAL)));
    }
    break;
default:
    if (interactive || destinationImminent) {
        sound("sound.event.illegalMove");
    }
    result = false;
    break;
}
```

```
    if (destinationImminent && !unit.isDisposed()) {
        // The unit either reached the destination or failed at
        // the last step for some reason.  In either case, clear
        // the goto orders because they have failed.
        if (!askClearGotoOrders(unit)) result = false;
    }
    // Force redisplay of unit information
    if (unit == getGUI().getActiveUnit()) {
        /*
         * The unit might have been disposed as a result of the move
         * when we get here. For example after vanishing when exploring
         * a lost city rumour.
         */
        changeView(unit, true);
    }

    return result;
}
```

Pic. 2 to 8. Complete representation of the *moveDirection* method.

**3-Duplicated Code:**

In the Flag class located in the *src.net.sf.freecol.client.gui.dialog* package, there are multiple sections of code that repeat. For example, both the *drawStripes* and *drawQuarters* methods contain repeated lines of code for *g.setColor* and *rectangle.setRect*. One possible solution would be to create helper methods to eliminate this duplicated code.

```
private void drawStripes(Graphics2D g, Alignment alignment, int stripes) {
    int colors = backgroundColors.size();
    double stripeWidth = getStripeWidth(alignment);
    double stripeHeight = getStripeHeight(alignment);
    double x = (alignment == Alignment.VERTICAL)
        ? stripeWidth : 0;
    double y = (alignment == Alignment.HORIZONTAL)
        ? stripeHeight : 0;
    Rectangle2D.Double rectangle = new Rectangle2D.Double();
    for (int index = 0; index < stripes; index++) {
        g.setColor(backgroundColors.get(index % colors));
        rectangle.setRect(index * x, index * y, stripeWidth, stripeHeight);
        g.fill(rectangle);
    }
}
```

Pic. 9. Method *drawStripes*

```
private void drawQuarters(Graphics2D g) {
    int colors = backgroundColors.size();
    int[] x = { 0, 1, 1, 0 };
    int[] y = { 0, 0, 1, 1 };
    double halfWidth = WIDTH / 2;
    double halfHeight = HEIGHT / 2;
    double offset = (decoration == Decoration.SCANDINAVIAN_CROSS)
        ? CROSS_OFFSET : 0;
    Rectangle2D.Double rectangle = new Rectangle2D.Double();
    for (int index = 0; index < 4; index++) {
        g.setColor(backgroundColors.get(index % colors));
        rectangle.setRect(x[index] * halfWidth - offset, y[index] * halfHeight,
                          halfWidth + x[index] * offset, halfHeight);
        g.fill(rectangle);
    }
}
```

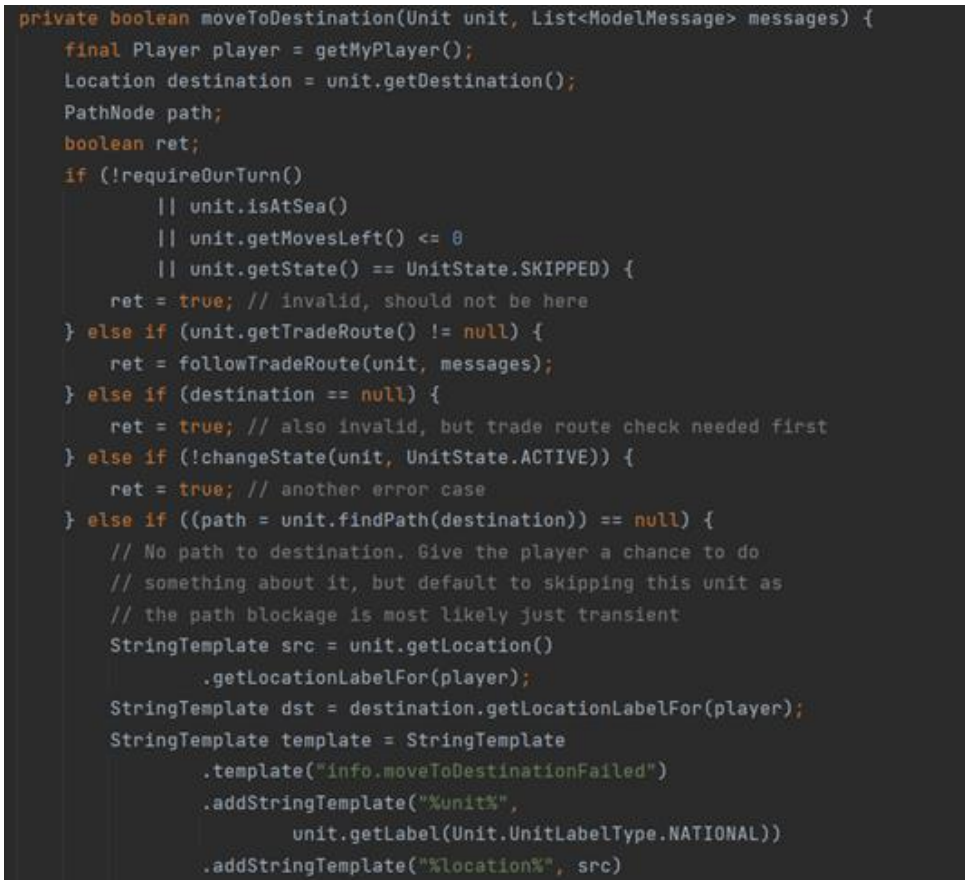
Fig10. Method drawQuarters

# Nádia Mendes 53175:

**1-Long Method:**

The method *moveToDestination* present in *the InGameController.java* class contained in the *src.net.sf.freecol.client.control* package is an extremely long method, it starts on line 1196 and ends on line 1256, containing 60 lines of code. The code, in addition to being extensive, also contains a lot of logic. Therefore, the Long Method code smell is verified.

A possible solution to this problem would be to divide this method into smaller and simpler submethods. This would clearly make the code easier to read and maintain.



```
private boolean moveToDestination(Unit unit, List<ModelMessage> messages) {
    final Player player = getMyPlayer();
    Location destination = unit.getDestination();
    PathNode path;
    boolean ret;
    if (!requireOurTurn()
            || unit.isAtSea()
            || unit.getMovesLeft() <= 0
            || unit.getState() == UnitState.SKIPPED) {
        ret = true; // invalid, should not be here
    } else if (unit.getTradeRoute() != null) {
        ret = followTradeRoute(unit, messages);
    } else if (destination == null) {
        ret = true; // also invalid, but trade route check needed first
    } else if (!changeState(unit, UnitState.ACTIVE)) {
        ret = true; // another error case
    } else if ((path = unit.findPath(destination)) == null) {
        // No path to destination. Give the player a chance to do
        // something about it, but default to skipping this unit as
        // the path blockage is most likely just transient
        StringTemplate src = unit.getLocation()
                .getLocationLabelFor(player);
        StringTemplate dst = destination.getLocationLabelFor(player);
        StringTemplate template = StringTemplate
                .template("info.moveToDestinationFailed")
                .addStringTemplate("%unit%",
                        unit.getLabel(Unit.UnitLabelType.NATIONAL))
                .addStringTemplate("%location%", src)
```

```
        } else {
            // If the unit has moves left, select it
            ret = unit.getMovesLeft() == 0;
        }
    } else { // Still in transit, do not select
        ret = true;
    }
    return ret;
}
```

```
            .addStringTemplate("%destination%", dst);
    showInformationPanel(unit, template);
    changeState(unit, UnitState.SKIPPED);
    ret = false;
} else if (!movePath(unit, path)) {
    ret = false; // ask the player to resolve the movePath problem
} else if (unit.isAtLocation(destination)) {
    final Colony colony = (unit.hasTile()) ? unit.getTile().getColony()
            : null;
    // Clear ordinary destinations if arrived.
    if (!askClearGotoOrders(unit)) {
        ret = false; // Should not happen.  Desync?  Ask the user.
    } else if (colony != null) {
        // Always ask to be selected if arriving at a colony
        // unless the unit cashed in (and thus gone), and bring
        // up the colony panel so something can be done with the
        // unit
        if (checkCashInTreasureTrain(unit)) {
            ret = true;
        } else {
            showColonyPanelWithCarrier(colony, unit);
            ret = false;
        }
```

Fig. 1 to 3 Represent the *moveToDestination* method in its entirety.

**2-Large Class**

When we analyze the *InGameController* class present in the *src.net.sf.freecol.client.control* package, we see that the class has many responsibilities and methods, being an extremely long class, containing a total of 5387 lines. This way we verify that we are in the presence of the large class code smell. One solution would be to split this class into smaller classes, each with a single responsibility. For example, the methods that in this class deal with the movement of units, such as *moveToDestination*, *movePath* and *moveDirection*, could be placed in a new class, which only deals with the movement of units.

```
private boolean moveToDestination(Unit unit, List<ModelMessage> messages) {
    final Player player = getMyPlayer();
    Location destination = unit.getDestination();
    PathNode path;
    boolean ret;
    if (!requireOurTurn()
            || unit.isAtSea()
            || unit.getMovesLeft() <= 0
            || unit.getState() == UnitState.SKIPPED) {
        ret = true; // invalid, should not be here
    } else if (unit.getTradeRoute() != null) {
        ret = followTradeRoute(unit, messages);
    } else if (destination == null) {
        ret = true; // also invalid, but trade route check needed first
    } else if (!changeState(unit, UnitState.ACTIVE)) {
        ret = true; // another error case
    } else if ((path = unit.findPath(destination)) == null) {
        // No path to destination. Give the player a chance to do
        // something about it, but default to skipping this unit as
        // the path blockage is most likely just transient
        StringTemplate src = unit.getLocation()
                .getLocationLabelFor(player);
        StringTemplate dst = destination.getLocationLabelFor(player);
        StringTemplate template = StringTemplate
                .template("info.moveToDestinationFailed")
                .addStringTemplate("%unit%",
                        unit.getLabel(Unit.UnitLabelType.NATIONAL))
                .addStringTemplate("%location%", src)
```

Fig.4 Represents part of the *moveToDestination* method.

```
public boolean moveDirection(Unit unit, Direction direction,
boolean interactive) {
    // Is the unit on the brink of reaching the destination with
    // this move?
    final Location destination = unit.getDestination();
    final Tile oldTile = unit.getTile();
    boolean destinationImminent = destination != null
            && oldTile != null
            && Map.isSameLocation(oldTile.getNeighbourOrNull(direction),
            destination);

    // Consider all the move types.
    final Unit.MoveType mt = unit.getMoveType(direction);
    boolean result = mt.isLegal();
    switch (mt) {
        case MOVE_HIGH_SEAS:
            // If the destination is Europe (and valid) move there,
            // if the destination is null, ask what to do,
            // otherwise just move on the map.
            result = (destination instanceof Europe
                    && getMyPlayer().getEurope() != null)
                    ? moveTowardEurope(unit, (Europe)destination)

                    : (destination == null)
                    ? moveHighSeas(unit, direction)
                    : moveTile(unit, direction);
            break;
        case MOVE:
```

Fig.5 Partial representation of the *moveDirection* method.

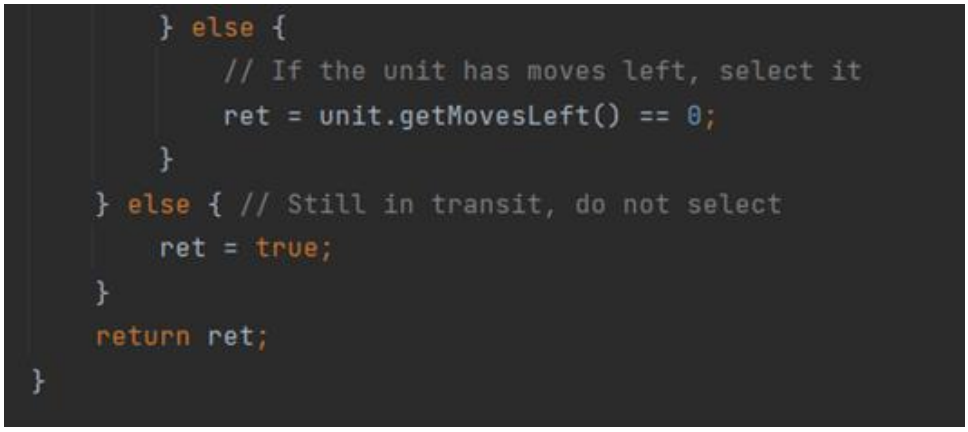
**3-Data Clumps**

There is evidence of data groupings, in the *InGameController.java* class contained in the *src.net.sf.freecol.client.control* package, there are methods to receive many parameters, such as the *moveToDestination* method almost starts on line 1196, where the unit is passed as a parameter to almost all unit-related method calls such as *followTradeRoute*, *moveTile*, *moveAttack*, etc. This can be considered a data clump since the unit object is always related to these movement operations, and the same Parcels are repeated past.

A possible solution would be to create objects to group related data and make the code more readable.

```
private boolean moveToDestination(Unit unit, List<ModelMessage> messages) {
    final Player player = getMyPlayer();
    Location destination = unit.getDestination();
    PathNode path;
    boolean ret;
    if (!requireOurTurn()
            || unit.isAtSea()
            || unit.getMovesLeft() <= 0
            || unit.getState() == UnitState.SKIPPED) {
        ret = true; // invalid, should not be here
    } else if (unit.getTradeRoute() != null) {
        ret = followTradeRoute(unit, messages);
    } else if (destination == null) {
        ret = true; // also invalid, but trade route check needed first
    } else if (!changeState(unit, UnitState.ACTIVE)) {
        ret = true; // another error case
    } else if ((path = unit.findPath(destination)) == null) {
        // No path to destination. Give the player a chance to do
        // something about it, but default to skipping this unit as
        // the path blockage is most likely just transient
        StringTemplate src = unit.getLocation()
                .getLocationLabelFor(player);
        StringTemplate dst = destination.getLocationLabelFor(player);
        StringTemplate template = StringTemplate
                .template("info.moveToDestinationFailed")
                .addStringTemplate("%unit%",
                        unit.getLabel(Unit.UnitLabelType.NATIONAL))
                .addStringTemplate("%location%", src)
```

```
                .addStringTemplate("%destination%", dst);
        showInformationPanel(unit, template);
        changeState(unit, UnitState.SKIPPED);
        ret = false;
    } else if (!movePath(unit, path)) {
        ret = false; // ask the player to resolve the movePath problem
    } else if (unit.isAtLocation(destination)) {
        final Colony colony = (unit.hasTile()) ? unit.getTile().getColony()
                : null;
        // Clear ordinary destinations if arrived.
        if (!askClearGotoOrders(unit)) {
            ret = false; // Should not happen.  Desync?  Ask the user.
        } else if (colony != null) {
            // Always ask to be selected if arriving at a colony
            // unless the unit cashed in (and thus gone), and bring
            // up the colony panel so something can be done with the
            // unit
            if (checkCashInTreasureTrain(unit)) {
                ret = true;
            } else {
                showColonyPanelWithCarrier(colony, unit);
                ret = false;
            }
```



```
        } else {
            // If the unit has moves left, select it
            ret = unit.getMovesLeft() == 0;
        }
    } else { // Still in transit, do not select
        ret = true;
    }
    return ret;
}
```

Fig. 6 to 8 Representation of the *moveToDestination* method.

# José Morgado:

## 1º - Comments

The FreeColClient.java class contained in the src/net/sf/freecol/client package appears to have an excessive number of comments. Well-defined methods, such as those shown in Figure 1 and Figure 2, do not require redundant comments. One possible solution to this problem would be to remove comments from functions that clearly specify their behavior solely through their names.

Fig. 1 – isLoggedIn Method

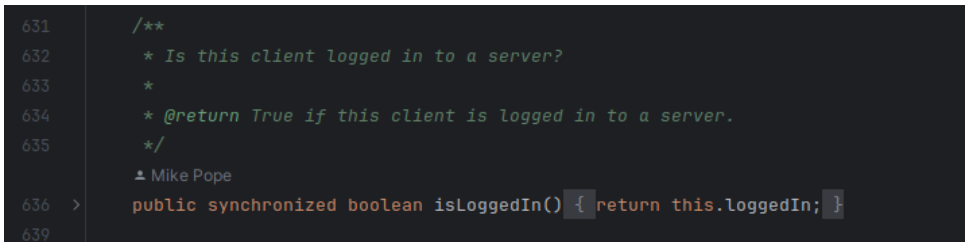
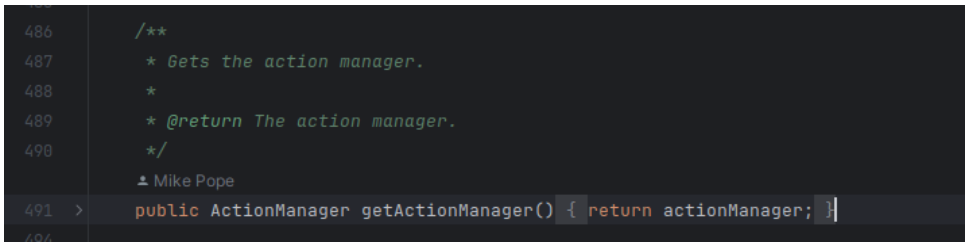



Fig. 2 – ActionManager Method




## 2º - Magic Numbers code smell

The code within the IndianSettlement.java class in the src/net/sf/freecol/common/model directory may exhibit the "magic numbers" code smell, indicated by the presence of hard-coded numeric values lacking clear explanation or context.

Refactoring this code by replacing such magic numbers with named constants or variables having descriptive names can significantly enhance code readability and maintainability. By introducing named constants like PREMIUM_WANTED_PRICE = 150 and similar, the purpose of these numbers becomes explicit, improving the code's comprehensibility for future developers.

Addressing the reliance on magic numbers in the IndianSettlement.java class can lead to clearer and more maintainable code, facilitating easier comprehension and future modifications.

Fig. 3 – Some Magical numbers

```
846
847         // Apply wanted bonus
848         final int wantedBase = 100; // Granularity for wanted bonus
849         final int wantedBonus // Premium paid for wanted goods types
850             = (type == getWantedGoods(0)) ? 150
851             : (type == getWantedGoods(1)) ? 125
852             : (type == getWantedGoods(2)) ? 110
853             : 100;
854         // Do not simplify with *=, we want the integer truncation.
855         price = wantedBonus * price / wantedBase;
856
857         logger.finest("Full price(" + amount + " " + type + ")"
858                       + " → " + price);
859         return price;
860     }
861
```

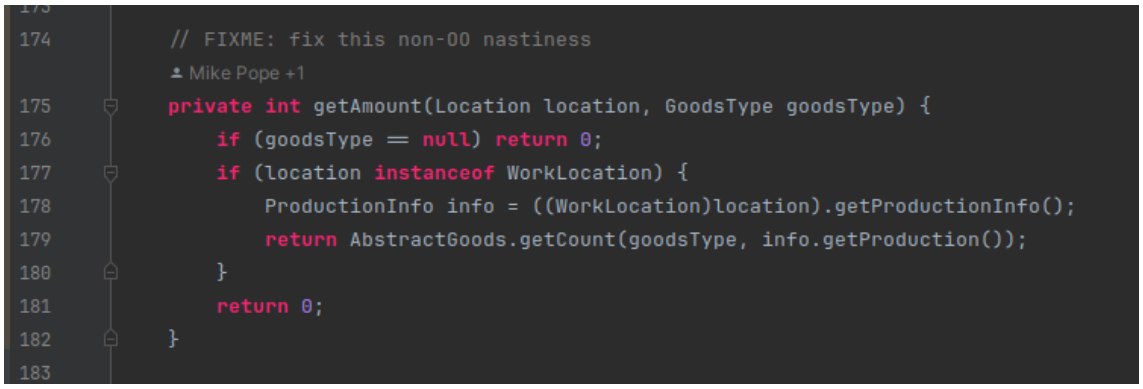
## 3º - Instance Type Checking code smell

The class UnitWas in the src/net/sf/freecol/common/model directory appears to present the “Instance Type Checking” code smell, as the multiple chained ternary operators perform type checking (instanceof) for various types of FreeColGameObject to determine a specific change type. This practice violates the principles of polymorphism and object-oriented design by centralizing decision-making logic based on the concrete type of objects.

This leads to less maintainable code as any modification or addition of types “FreeColGameObject” will require changing the existing logic and thus violate the Open-Closed Principle (of the SOLID Principles).

We could fix this by adding the change function to each of the child objects and calling it directly from the given object, instead of checking the type in the parent function.

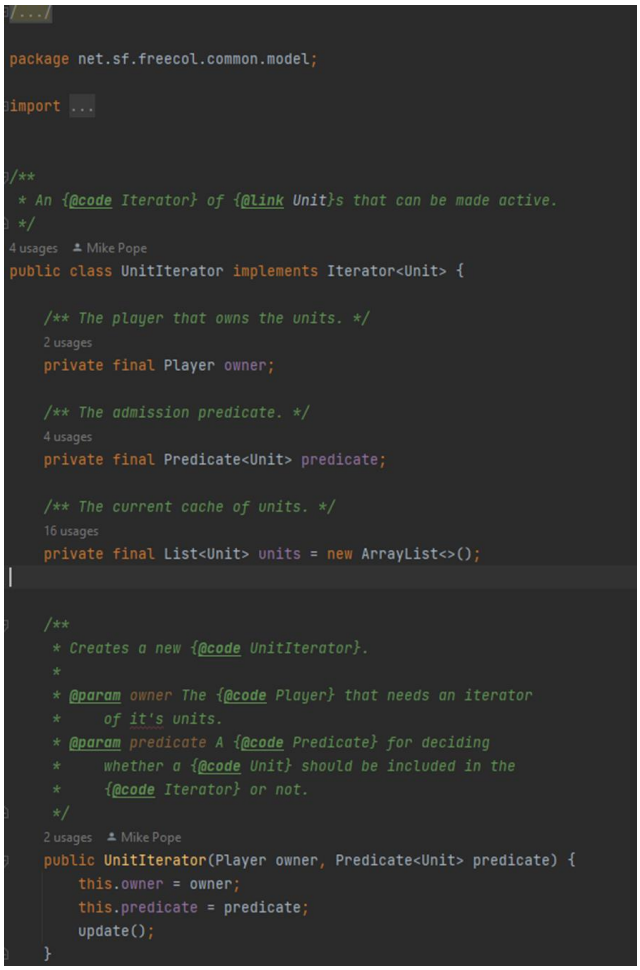
Fig. 4 - “Non-OO nastiness”



# Gof Patterns:

## João Amorim:

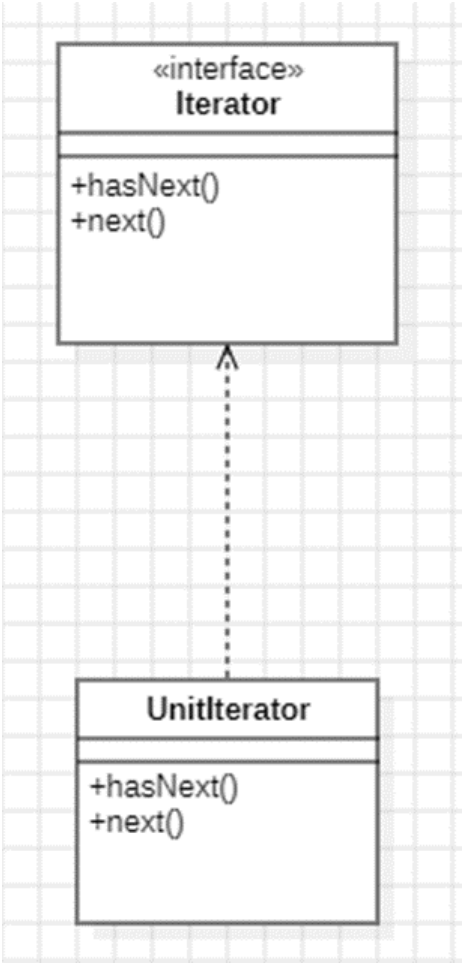
### 1 – Iterator - net.sf.freecol.common.model.UnitIterator



```
/.../

package net.sf.freecol.common.model;

import ...


/**
 * An {@code Iterator} of {@link Unit}s that can be made active.
 */
4 usages  Mike Pope
public class UnitIterator implements Iterator<Unit> {

    /** The player that owns the units. */
    2 usages
    private final Player owner;

    /** The admission predicate. */
    4 usages
    private final Predicate<Unit> predicate;

    /** The current cache of units. */
    16 usages
    private final List<Unit> units = new ArrayList<>();


    /**
     * Creates a new {@code UnitIterator}.
     *
     * @param owner The {@code Player} that needs an iterator
     *     of it's units.
     * @param predicate A {@code Predicate} for deciding
     *     whether a {@code Unit} should be included in the
     *     {@code Iterator} or not.
     */
    2 usages  Mike Pope
    public UnitIterator(Player owner, Predicate<Unit> predicate) {
        this.owner = owner;
        this.predicate = predicate;
        update();
    }
```

In this example, the iterator pattern is being implemented directly by the means of a custom iterator in this case.

## Class Diagram –

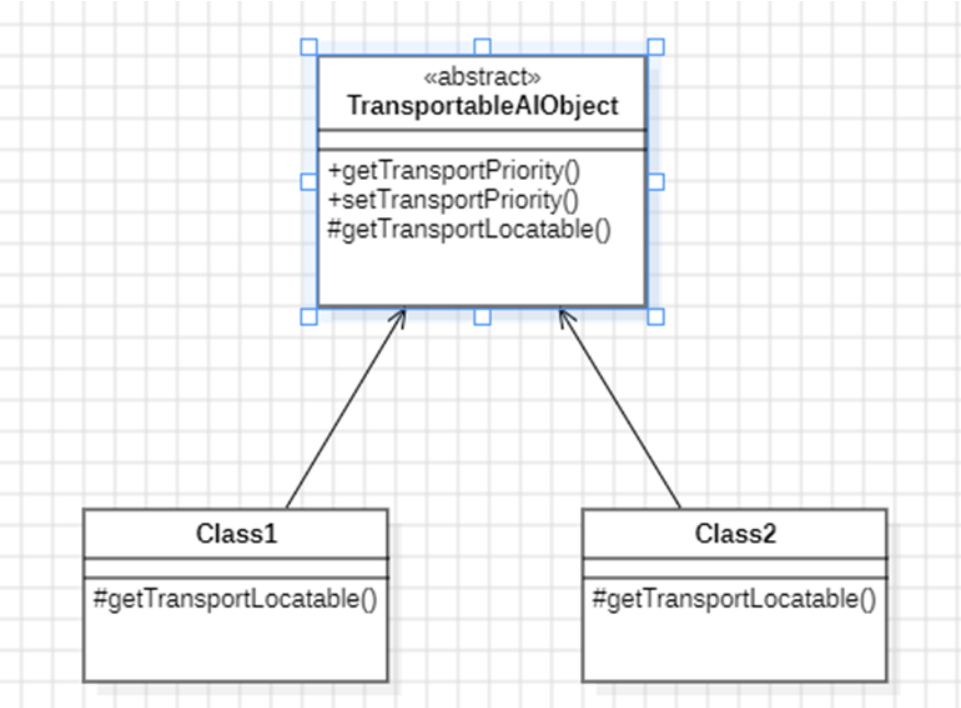



## 2 – Template - net.sf.freecol.server.ai.TrasportableAIObject

```java
/**
 * A single item in a carrier's transport list.  Any {@link Locatable}
 * which should be able to be transported by a carrier using the
 * {@link net.sf.freecol.server.ai.mission.TransportMission},
 * should extend this class.
 *
 * @see net.sf.freecol.server.ai.mission.TransportMission
 */
3 inheritors  Michael Pope +2
public abstract class TransportableAIObject extends ValuedAIObject {

    /**
     * The priority for a goods that are hitting the warehouse limit.
     */
    1 usage
    public static final int IMPORTANT_DELIVERY = 110;

    /**
     * The priority for goods that provide at least a full cargo load.
     */
    1 usage
    public static final int FULL_DELIVERY = 100;

    /**
     * The priority of tools intended for a Colony with none stored
     * at the present (and with no special needs).
     */
    3 usages
    public static final int TOOLS_FOR_COLONY_PRIORITY = 10;

    /**
     * The extra priority value added to the base value of
     * {@link #TOOLS_FOR_COLONY_PRIORITY}
     * for each ColonyTile needing a terrain improvement.
     */
    public static final int TOOLS_FOR_IMPROVEMENT = 10;

    /**
     * The extra priority value added to the base value of
```

We can see the template pattern here because in this case this class serves has an “template” for objects in the game having methods that are common for all of them and having abstract methods that have different behaviours in the classes that implement them.

## Class Diagram –

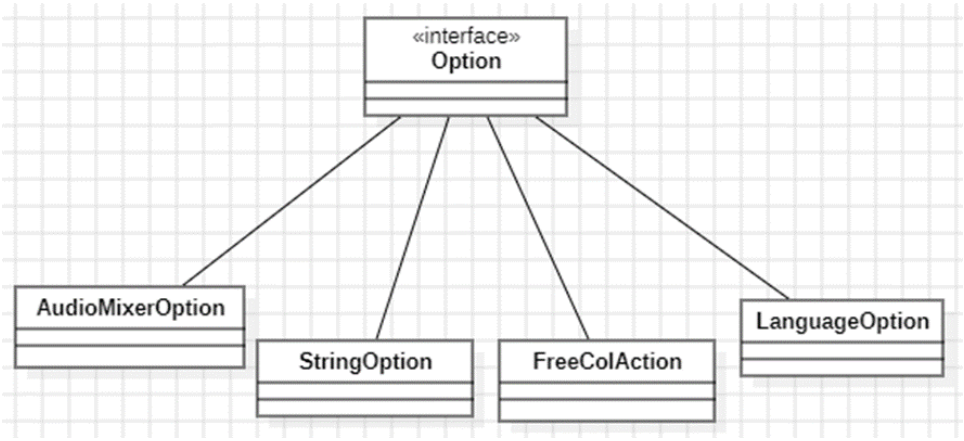



## 3 – Facade - net.sf.freecol.client.gui.action.Option

```
/**
 * An option describes something which can be customized by the user.
 */
101 implementations  Mike Pope +6
public interface Option<T> extends Cloneable, ObjectWithId {

    /**
     * Clone this option.
     *
     * @return A clone of this option.
     * @exception CloneNotSupportedException if we can not clone.
     */
    15 implementations  Mike Pope
    public Option<T> cloneOption() throws CloneNotSupportedException;

    /**
     * Gets the option group identifier for this option.
     *
     * @return The option group identifier.
     */
    2 implementations  Mike Pope
    public String getGroup();

    /**
     * Set the option group for this option.
     *
     * @param group The identifier for the option group.
     */
    2 implementations  Mike Pope
    public void setGroup(String group);

    /**
     * Gets the value of this option.
```

This interface provides a simplified, unified interface to a complex subsystem.

## Class Diagram –



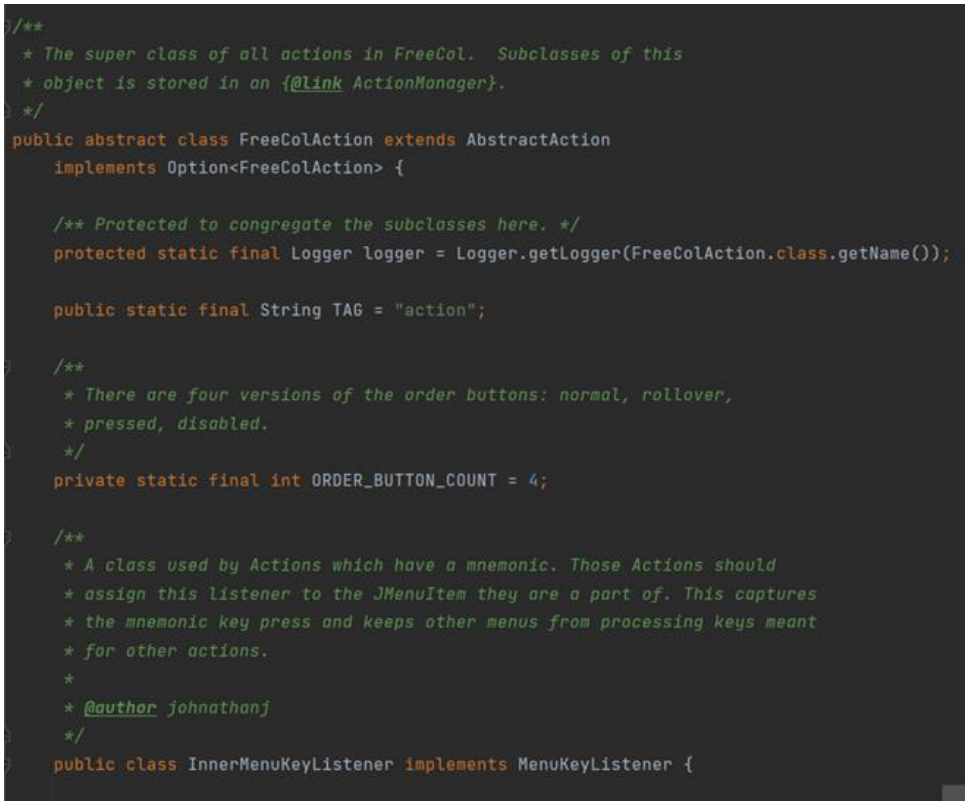

There are more classes that implement methods of Option but for the sake of simplicity and size of the diagram I selected 4.

# Nádia Mendes 53175:

## 1 - **Template method pattern:** package net.sf.freecol.client.gui.action.FreeColAction

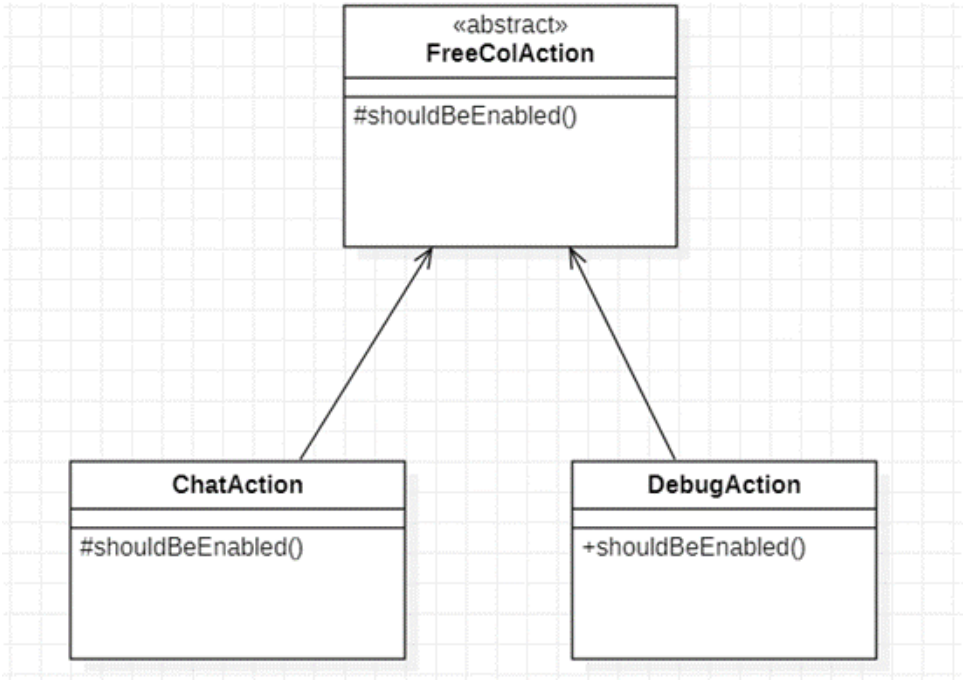
In this exemple we can see that FreeColAction serves as an abstract base class that defines a skeleton of behavior for various actions. It implements common methods and defines a shouldBeEnabled method as a hook that is overridden by its subclasses to provide specific behaviors, as is the case with the ChatAction and DebugAction classes.

In the ChatAction class, which is a subclass of the FreeColAction class, this is exactly what happens, it overrides the shouldBeEnabled method to provide specific logic for the chat action.



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

    /** Protected to congregate the subclasses here. */
    protected static final Logger logger = Logger.getLogger(FreeColAction.class.getName());

    public static final String TAG = "action";

    /**
     * There are four versions of the order buttons: normal, rollover,
     * pressed, disabled.
     */
    private static final int ORDER_BUTTON_COUNT = 4;

    /**
     * A class used by Actions which have a mnemonic. Those Actions should
     * assign this listener to the JMenuItem they are a part of. This captures
     * the mnemonic key press and keeps other menus from processing keys meant
     * for other actions.
     *
     * @author johnathanj
     */
    public class InnerMenuKeyListener implements MenuKeyListener {
```

## Class Diagram –




## 2 - **Observer pattern:** net.sf.freecol.client.gui.dialog.CaptureGoodsDialog

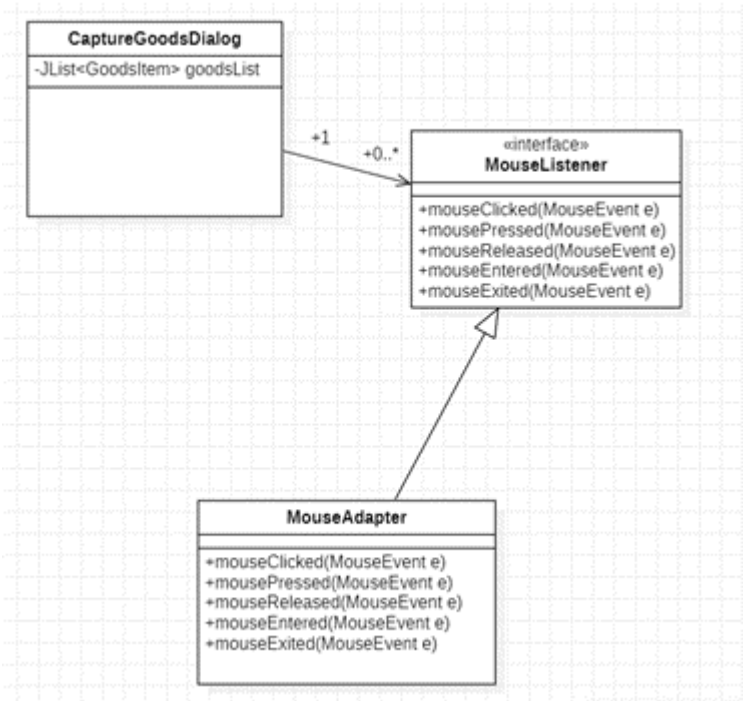
In this CaptureGoodsDialog class we check the existence of an observer pattern, where the subject of the Observer pattern is the goodsList, which is an instance of JList<GoodsItem>. However, there is no explicit interface called Observer or Observable, because in context the interaction between the subject (such as the goodsList) and the observers (or "listeners") is handled through specific methods and interfaces provided by Java Swing itself.

The interaction between goodsList and observers is carried out through the addMouseListener(MouseListener listener) and removeMouseListener(MouseListener listener) methods. These methods add or remove specific observers that implement the MouseListener interface. The goodsList notifies these observers when mouse events occur.

Therefore, the goodsList acts as the subject and the observers are classes that implement the MouseListener interface.

```java
private final JList<GoodsItem> goodsList;


/**
 * Creates a new CaptureGoodsDialog.
 *
 * @param freeColClient The {@code FreeColClient} for the game.
 * @param frame The owner frame.
 * @param winner The {@code Unit} that is looting.
 * @param loot The {@code Goods} to loot.
 */
public CaptureGoodsDialog(FreeColClient freeColClient, JFrame frame,
        Unit winner, List<Goods> loot) {
    super(freeColClient, frame);

    this.maxCargo = winner.getSpaceLeft();

    GoodsItem[] goods = new GoodsItem[loot.size()];
    for (int i = 0; i < loot.size(); i++) {
        goods[i] = new GoodsItem(loot.get(i));
    }
    this.goodsList = new JList<>();
    this.goodsList.setListData(goods);

    this.allButton = Utility.localizedButton("all");
    this.allButton.addActionListener((ActionEvent ae) -> {
            JList<GoodsItem> gl = CaptureGoodsDialog.this.goodsList;
```

## Class Diagram –



## 3 - **Abstract Factory Pattern:**

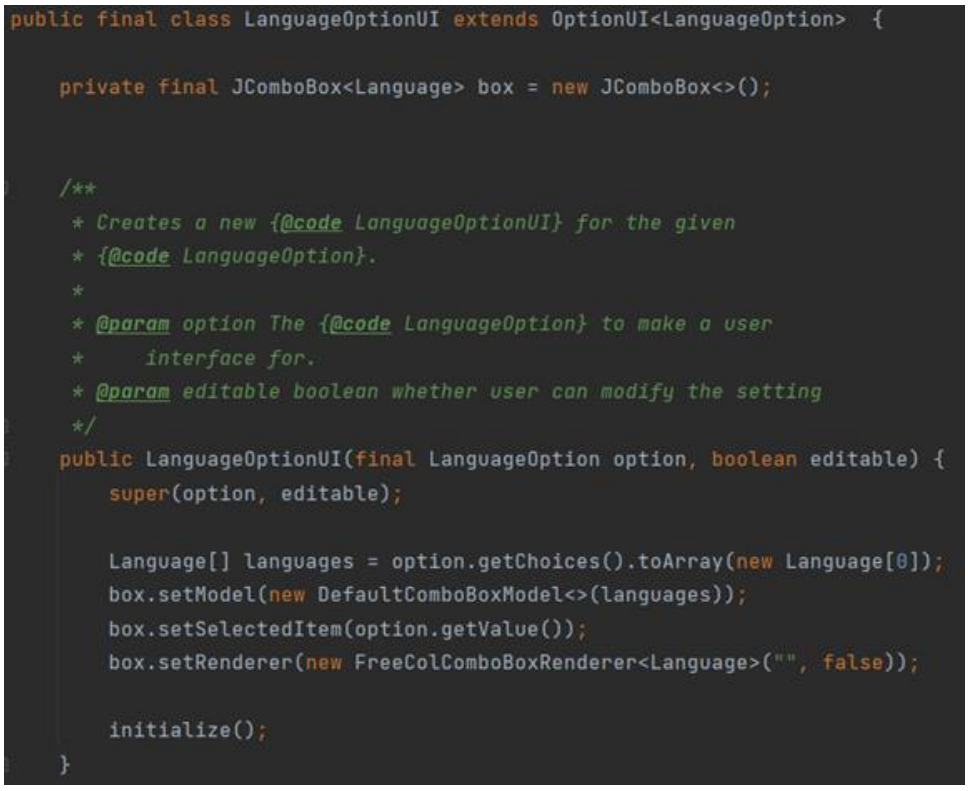
src.net.sf.freecol.client.gui.option.LanguageOptionUi

In the LanguageOptionUI class, contained in the src.net.sf.freecol.client.gui.option package, it appears that it acts as an abstract factory to create objects related to the language option (LanguageOption)

The LanguageOptionUI class creates and returns an instance of JComboBox <Language> which is a part of the language option-related family of UI objects.

The LanguageOption class represents the language option, while the Language class represents the available languages.

Therefore, the use of the Abstract Factory Pattern is used to create related objects according to the language choice, and this allows the creation of a family of coherent objects related to the language choice.

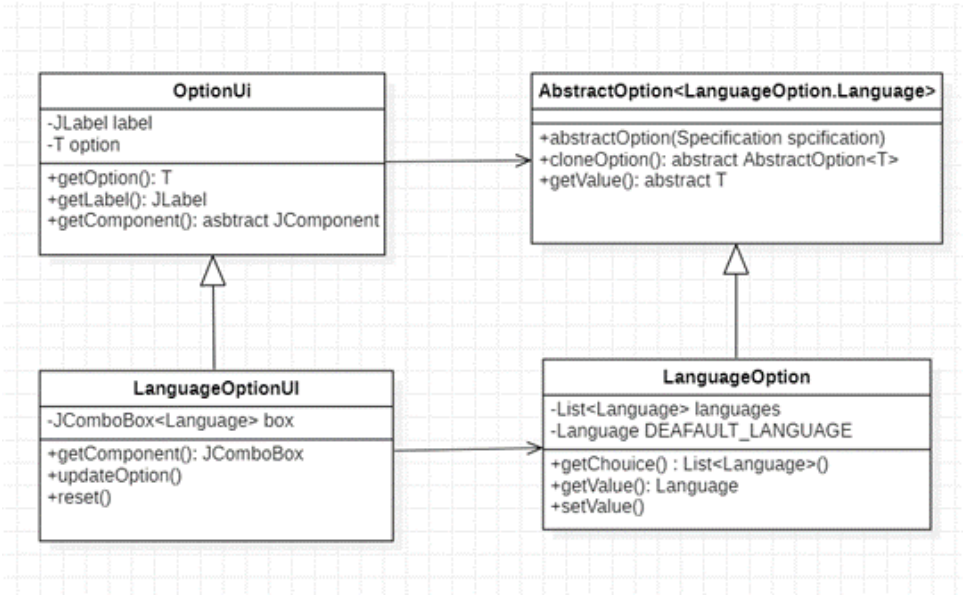


```
public final class LanguageOptionUI extends OptionUI<LanguageOption>  {

    private final JComboBox<Language> box = new JComboBox<>();


    /**
     * Creates a new {@code LanguageOptionUI} for the given
     * {@code LanguageOption}.
     *
     * @param option The {@code LanguageOption} to make a user
     *     interface for.
     * @param editable boolean whether user can modify the setting
     */
    public LanguageOptionUI(final LanguageOption option, boolean editable) {
        super(option, editable);

        Language[] languages = option.getChoices().toArray(new Language[0]);
        box.setModel(new DefaultComboBoxModel<>(languages));
        box.setSelectedItem(option.getValue());
        box.setRenderer(new FreeColComboBoxRenderer<Language>("", false));

        initialize();
    }
```

```java
/**
 * {@inheritDoc}
 */
@Override
public JComboBox getComponent() { return box; }

/**
 * {@inheritDoc}
 */
@Override
public void updateOption() { getOption().setValue((Language)box.getSelectedItem()); }

/**
 * {@inheritDoc}
 */
@Override
public void reset() { box.setSelectedItem(getOption().getValue()); }
```

Representation of code of the *LangugeOptionUI* class.

## Class Diagram –



# João Esteves 47994:

## 1-Template Method Pattern:

The *loadGame(File file)* method in the *MapEditorController* class located in the *net.sf.freecol.client.control* package is an example of a method that follows the Template Method pattern. It establishes a general structure for loading a game but delegates the implementation of specific details to derived classes.

```
private void loadGame(File file) {
    final FreeColClient fcc = getFreeColClient();
    final GUI gui = getGUI();

    fcc.setMapEditor(true);
    gui.showStatusPanel(Messages.message("status.loadingGame"));

    final File theFile = file;
    new Thread(FreeCol.CLIENT_THREAD + "Loading-Map") {
        @Override
        public void run() {
            final FreeColServer freeColServer = getFreeColServer();
            try {
                Specification spec = getDefaultSpecification();
                Game game = FreeColServer.readGame(new FreeColSavegameFile(theFile),
                                                   spec, freeColServer);
                fcc.setGame(game);
                requireNativeNations(game);
                SwingUtilities.invokeLater(() -> {
                        gui.closeStatusPanel();
                        gui.setFocus(game.getMap().getTile(1,1));
                        gui.updateMenuBar();
                        gui.refresh();
                    });
            } catch (FileNotFoundException fnfe) {
                gui.showErrorPanel(fnfe,
                    FreeCol.badFile("error.couldNotFind", theFile));
            } catch (IOException | FreeColException ioe) {
                gui.showErrorPanel(fnfe,
                    FreeCol.badFile("error.couldNotFind", theFile));
            } catch (IOException | FreeColException ioe) {
                gui.showErrorPanel(ioe,
                    StringTemplate.key("server.initialize"));
            } catch (XMLStreamException xse) {
                gui.showErrorPanel(xse,
                    FreeCol.badFile("error.couldNotLoad", theFile));
            }
        }
    }.start();
  }
}
```

Pic. 1 to 2. Representation of method LoadGame

## 2-Command Pattern

In the code of the GUI class located in the *net.sf.freecol.client.gui* package, there are various actions such as "buy," "sell," "negotiate," "attack," and others. These are represented as choice objects (ChoiceItem) and passed to the *getChoice* method. In this way, there is an application of the Command Pattern principle.

```
public TradeAction getIndianSettlementTradeChoice(Settlement settlement,
                                                   StringTemplate template,
                                                   boolean canBuy,
                                                   boolean canSell,
                                                   boolean canGift) {
    String msg;
    ArrayList<ChoiceItem<TradeAction>> choices = new ArrayList<>();
    if (canBuy) {
        msg = Messages.message("tradeProposition.toBuy");
        choices.add(new ChoiceItem<>(msg, TradeAction.BUY, canBuy));
    }
    if (canSell) {
        msg = Messages.message("tradeProposition.toSell");
        choices.add(new ChoiceItem<>(msg, TradeAction.SELL, canSell));
    }
    if (canGift) {
        msg = Messages.message("tradeProposition.toGift");
        choices.add(new ChoiceItem<>(msg, TradeAction.GIFT, canGift));
    }
    if (choices.isEmpty()) return null;

    return getChoice(settlement.getTile(), template,
                     settlement, "cancel", choices);
}
```

Pic. 3. Partial representation of the method *getIndianSettlementTradeChoice.*

## 3-Proxy Pattern:

In the GUI class within the *net.sf.freecol.client.gui* package, intermediate methods are used for user interactions. For example, methods like *getBoycottChoice* and *getBuyChoice* serve as intermediaries to obtain user choices.

```
public BoycottAction getBoycottChoice(Goods goods, Europe europe) {
    int arrears = europe.getOwner().getArrears(goods.getType());
    StringTemplate template = StringTemplate
        .template("boycottedGoods.text")
        .addNamed("%goods%", goods)
        .addNamed("%europe%", europe)
        .addAmount("%amount%", arrears);

    List<ChoiceItem<BoycottAction>> choices = new ArrayList<>();
    choices.add(new ChoiceItem<>(Messages.message("payArrears"),
                                 BoycottAction.BOYCOTT_PAY_ARREARS));
    choices.add(new ChoiceItem<>(Messages.message("boycottedGoods.dumpGoods"),
                                 BoycottAction.BOYCOTT_DUMP_CARGO));

    return getChoice(null, template,
                     goods.getType(), "cancel", choices);
}
```

Pic. 4. Partial representation of the method *getBoycottChoice.*

# José Morgado:

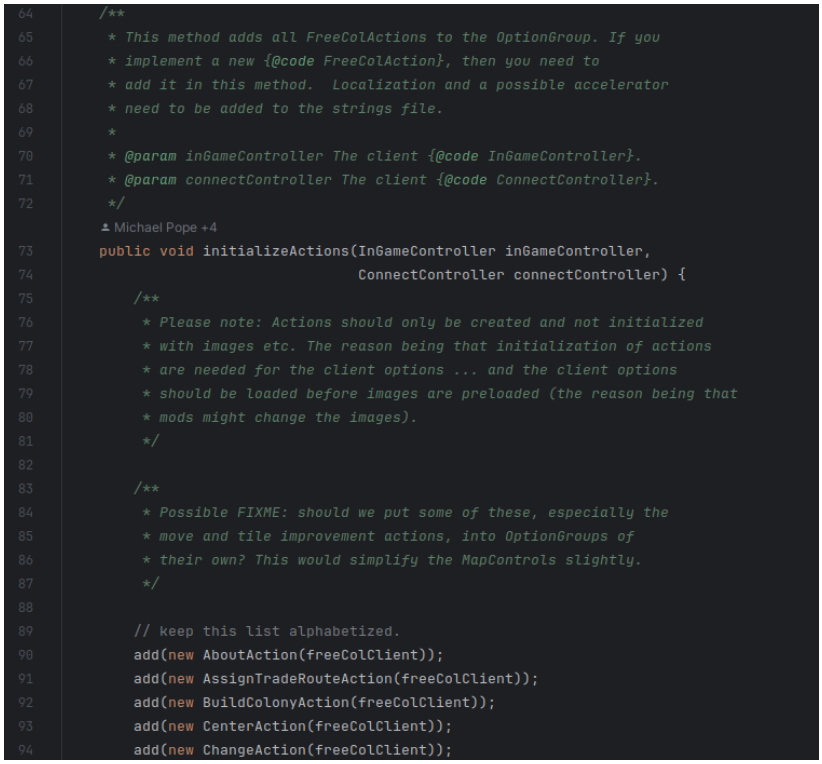
## 1-Command Pattern

The ActionManager.java class in the src/net/sf/freecol/client/gui/action package appears to exhibit the Command pattern. This pattern is designed to encapsulate requests as objects, allowing clients to parameterize them, queue them, and record their history (though it is unused in this method, even though it's implemented). In the context of the ActionManager:

Actions are represented as FreeColAction objects, which encapsulate specific requests or commands that can be executed in the game.

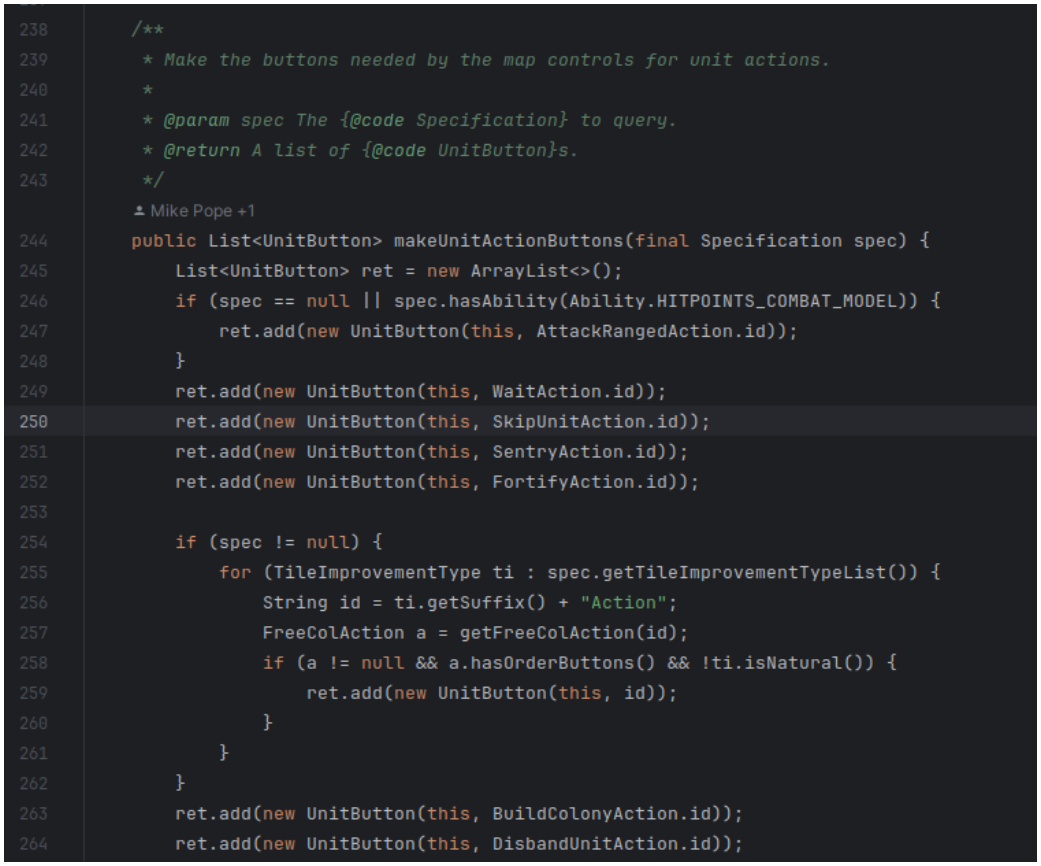
These actions are then mapped to buttons, enabling the user to request actions without needing to know the specific details of how they are executed or how the commands are processed.

Fig. 1 – Some implemented actions



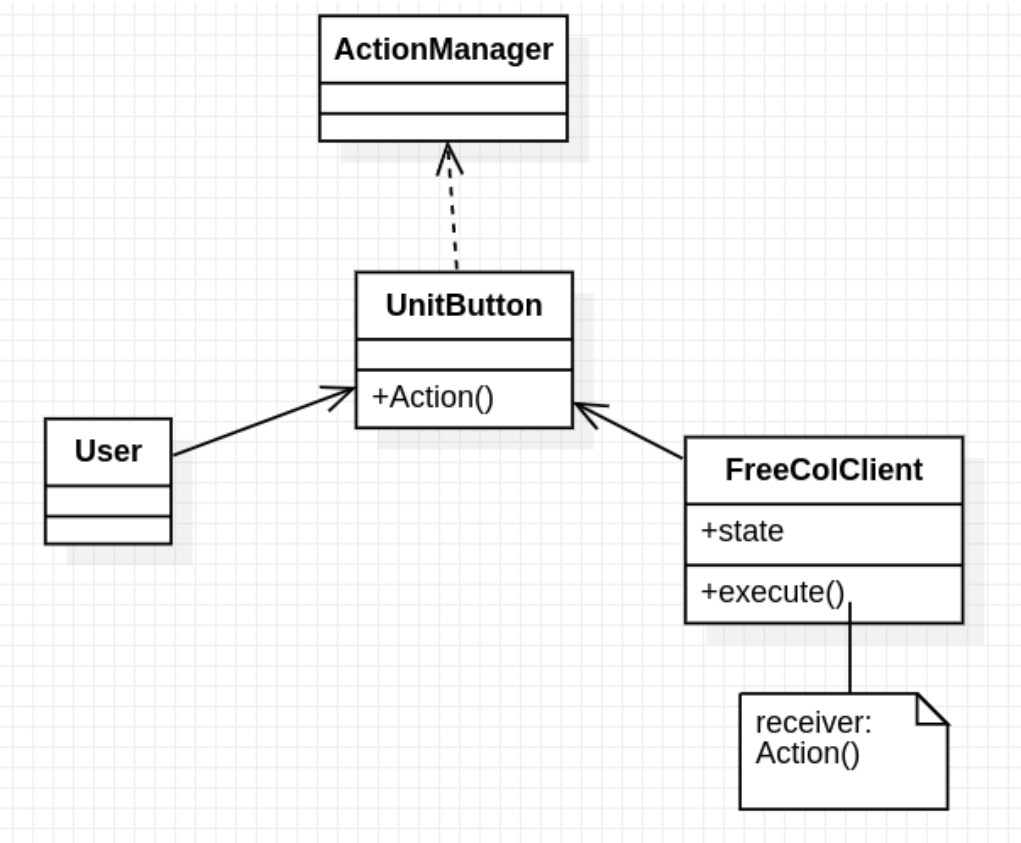
```
64      /**
65       * This method adds all FreeColActions to the OptionGroup. If you
66       * implement a new {@code FreeColAction}, then you need to
67       * add it in this method.  Localization and a possible accelerator
68       * need to be added to the strings file.
69       *
70       * @param inGameController The client {@code InGameController}.
71       * @param connectController The client {@code ConnectController}.
72       */
         Michael Pope +4
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));
```

These actions are then mapped to buttons, enabling the user to request actions without knowing the specific details of how they are executed or how the commands are processed.



```
238     /**
239      * Make the buttons needed by the map controls for unit actions.
240      *
241      * @param spec The {@code Specification} to query.
242      * @return A list of {@code UnitButton}s.
243      */
        Mike Pope +1
244     public List<UnitButton> makeUnitActionButtons(final Specification spec) {
245         List<UnitButton> ret = new ArrayList<>();
246         if (spec == null || spec.hasAbility(Ability.HITPOINTS_COMBAT_MODEL)) {
247             ret.add(new UnitButton(this, AttackRangedAction.id));
248         }
249         ret.add(new UnitButton(this, WaitAction.id));
250         ret.add(new UnitButton(this, SkipUnitAction.id));
251         ret.add(new UnitButton(this, SentryAction.id));
252         ret.add(new UnitButton(this, FortifyAction.id));
253 
254         if (spec != null) {
255             for (TileImprovementType ti : spec.getTileImprovementTypeList()) {
256                 String id = ti.getSuffix() + "Action";
257                 FreeColAction a = getFreeColAction(id);
258                 if (a != null && a.hasOrderButtons() && !ti.isNatural()) {
259                     ret.add(new UnitButton(this, id));
260                 }
261             }
262         }
263         ret.add(new UnitButton(this, BuildColonyAction.id));
264         ret.add(new UnitButton(this, DisbandUnitAction.id));
```

Fig. 2 - Mapping of actions to buttons

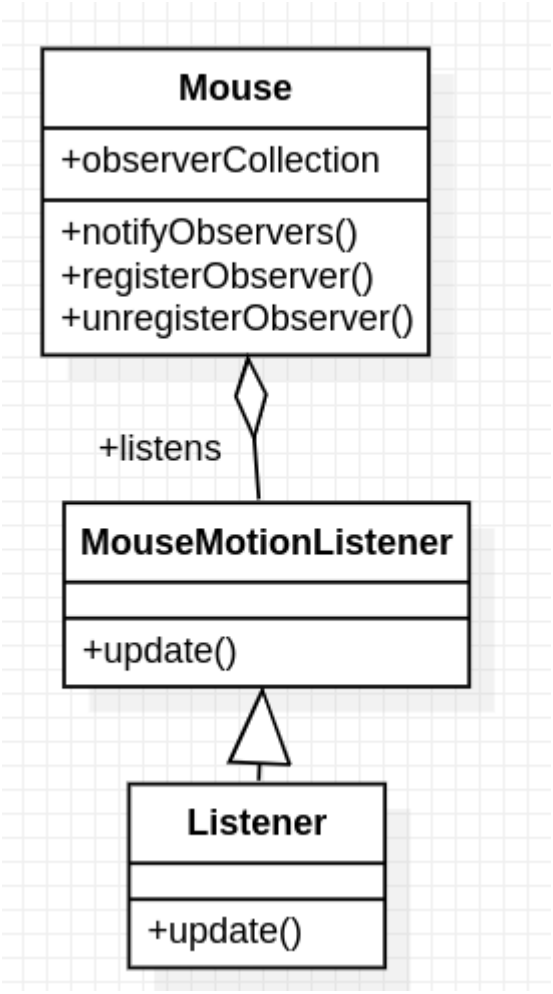



## 2-Observer Pattern

The class FreeColMenuBar.java, located in the package src/net/sf/freecol/client/gui/menu, contains listener elements that exhibit the Observer design pattern. One of these elements is the MouseMotionListener, which observes user-controlled mouse movement and provides a way of decoupling between the subject (in this case, the menu bar) and the observers, allowing observers to be notified and respond to events independently.

```
68      protected FreeColMenuBar(FreeColClient f, MouseMotionListener listener) {
69          // FIXME: FreeColClient should not have to be passed in to
70          // this class.  This is only a menu bar, it doesn't need a
71          // reference to the main controller. The only reason it has
72          // one now is because DebugMenu needs it. And DebugMenu needs
73          // it because it is using inner classes for ActionListeners
74          // and those inner classes use the reference.  If those inner
75          // classes were in seperate classes, when they were created,
76          // they could use the FreeColClient reference of the
77          // ActionManger. So DebugMenu needs to be refactored to remove
78          // inner classes so that this MenuBar can lose its unnecessary
79          // reference to the main controller.  See FreeColMenuTest.
80          //
81          // Okay, I lied.. the update() and paintComponent() methods in
82          // this MenuBar use freeColClient, too. But so what.  Move
83          // those to another class too. :)
84          super();
85
86          setOpaque(false);
87
88          this.freeColClient = f;
89          this.listener = listener;
90
91          this.am = f.getActionManager();
92
93          // Add a mouse listener so that autoscrolling can happen here
94          this.addMouseMotionListener(listener);
95
96          setBorder(FreeColImageBorder.menuBarBorder);
97      }
```

Fig. 3 - Initialization of the MouseMotionListener

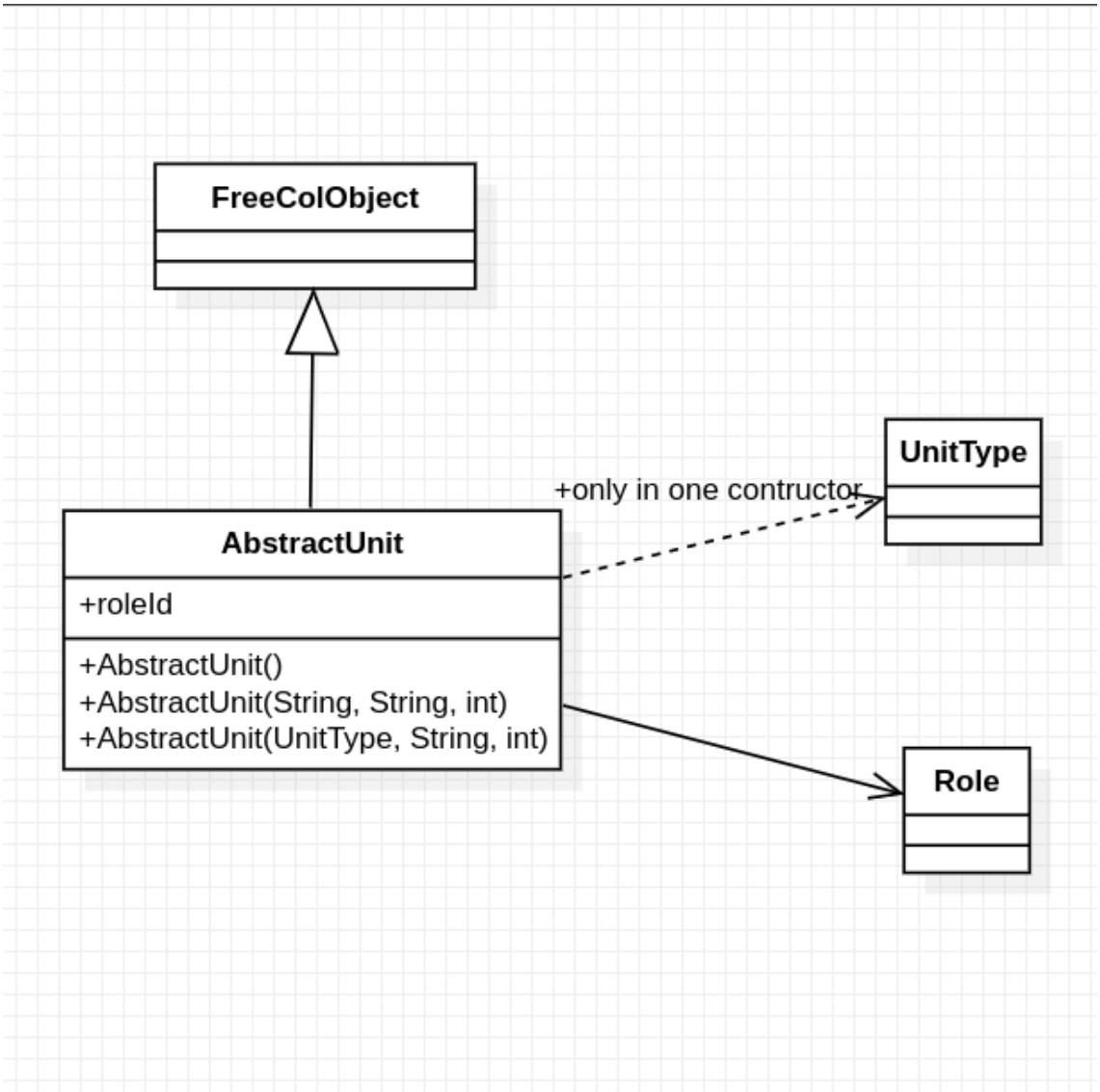



## 3-Factory Method Pattern

The class AbstractUnit.java in src/net/sf/freecol/common/model seems to exhibit the Factory Method pattern, the existence of different constructors like AbstractUnit(), AbstractUnit(String id, String roleId, int number) and AbstractUnit(UnitType unitType, String roleId, int number) provides various ways to instantiate AbstractUnit objects with different parameters and initialization approaches.

This versatility in object creation points to a Factory Method Pattern, where multiple factory methods or constructors exist in a class to create instances of objects, allowing flexibility in how these objects are created and initialized based on different criteria or parameters.


Fig. 4 - Different constructors


FreeColObject
UnitType
+only in one contructor
AbstractUnit
+roleId
+AbstractUnit()
+AbstractUnit(String, String, int)
+AbstractUnit(UnitType, String, int)
Role

# Metrics:

## João Amorim:

### Summary:

- LOC (Lines of Code) metrics are a quantitative measure used to assess the size and complexity of a software program. They count the number of lines of source code within a program or software project.

- CLOC (Comment Lines of Code): CLOC represents the number of lines in the class that are comments.

- JLOC (Javadoc Lines of Code): JLOC specifically counts lines of code that are part of Javadoc comments in a class.

- LOC (Lines of Code): LOC, as mentioned earlier, represents the total number of lines of code in a class.

## Data Visualization -

**Top 5 CLOC -**

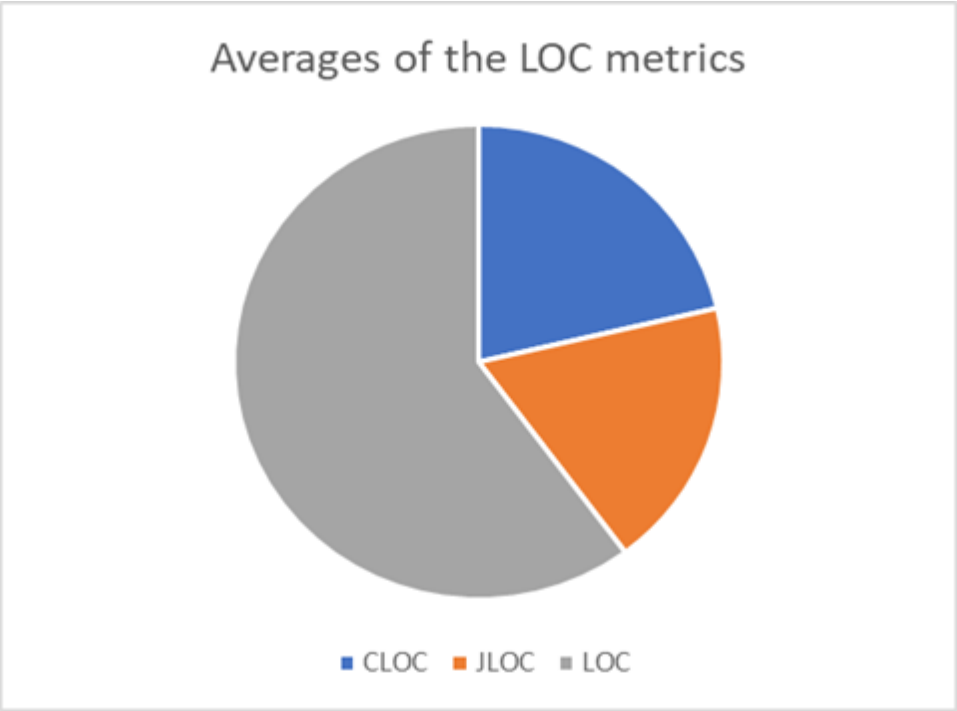
| class | CLOC | JLOC | LOC |
|---|---|---|---|
| net.sf.freecol.common.model.Unit | 1962.0 | 1766.0 | 4263.0 |
| net.sf.freecol.common.model.Player | 1921.0 | 1768.0 | 3892.0 |
| net.sf.freecol.client.control.InGameController | 1638.0 | 1297.0 | 4806.0 |
| net.sf.freecol.client.gui.GUI | 1457.0 | 1420.0 | 2220.0 |
| net.sf.freecol.common.util.CollectionUtils | 1445.0 | 1443.0 | 2374.0 |

**Top 5 JLOC -**

| class | CLOC | JLOC | LOC |
|---|---|---|---|
| net.sf.freecol.common.model.Player | 1921.0 | 1768.0 | 3892.0 |
| net.sf.freecol.common.model.Unit | 1962.0 | 1766.0 | 4263.0 |
| net.sf.freecol.common.util.CollectionUtils | 1445.0 | 1443.0 | 2374.0 |
| net.sf.freecol.client.gui.GUI | 1457.0 | 1420.0 | 2220.0 |
| net.sf.freecol.client.control.InGameController | 1638.0 | 1297.0 | 4806.0 |

**Top 5 LOC –**

| class | CLOC | JLOC | LOC |
|---|---|---|---|
| net.sf.freecol.client.control.InGameController | 1638.0 | 1297.0 | 4806.0 |
| net.sf.freecol.common.model.Unit | 1962.0 | 1766.0 | 4263.0 |
| net.sf.freecol.server.model.ServerPlayer | 1164.0 | 777.0 | 4217.0 |
| net.sf.freecol.common.model.Player | 1921.0 | 1768.0 | 3892.0 |
| net.sf.freecol.server.control.InGameController | 1073.0 | 738.0 | 3451.0 |




## Discussion-

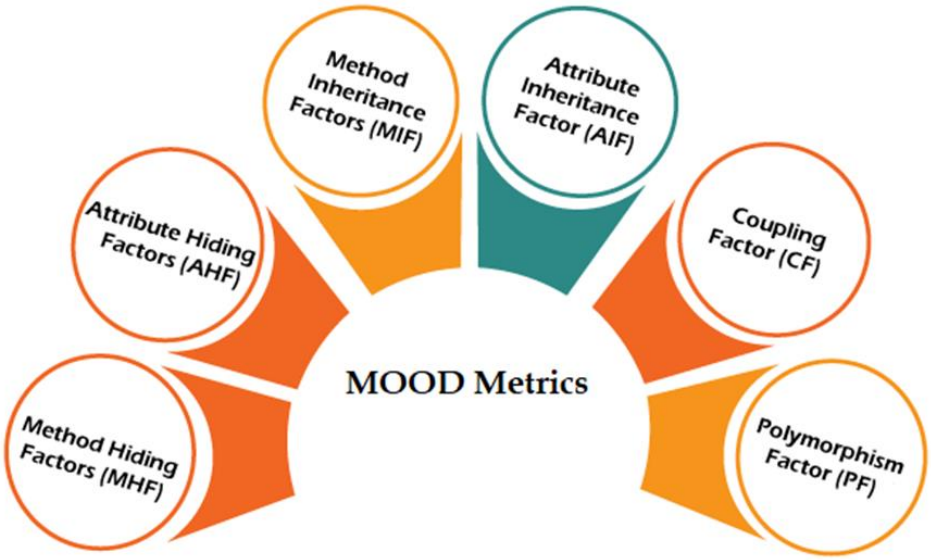
As it can be seen in the tables our top 5 tables are dominated by the same 5 classes in exception for LOC which means these classes are the most well documented classes, for example we can see this class net.sf.freecol.client.control.InGameController in all of the tables which mean its class of high importance in the code space having the most LOC and being documented with java doc and comments. Now looking at the pie chart which reflects the averages we can that the number of lines of java doc and comments are pretty much the same.

Now these results can be associated with the code smells reported, like the large method and Duplicated code, the class that I mentioned that had the most LOC can be a target of having these code smells because trough out the exploration of the code base I saw big use of java doc in large methods and to

explain the steps inside the method was used a lot of comments so that class can be a super class per say, that is doing more than it should be.

# MOOD Metrics

João Miguel Lopes Romão Esteves - 47994




- Attribute Hiding Factor (AHF)
- Attribute Inheritance Factor (AIF)
- Coupling Factor (CF)
- Method Hiding Factor (MHF)
- Method Inheritance Factor (MIF)
- Polymorphism Factor (PF)

## Attribute Hiding Factor (AHF)

Attribute Hiding Factor (AHF) measures the degree to which the attributes (instance variables) of a class are encapsulated and hidden from external classes. The higher the AHF, the better the encapsulation and hiding of attributes, which is generally considered beneficial for code maintenance and extensibility. This factor can be calculated using the following formula:

$$AHF = \frac{\sum_{i=1}^{TC} A_h(C_i)}{\sum_{i=1}^{TC} A_d(C_i)}$$

$A_h(C_i)$ = Hidden attributes in the class C

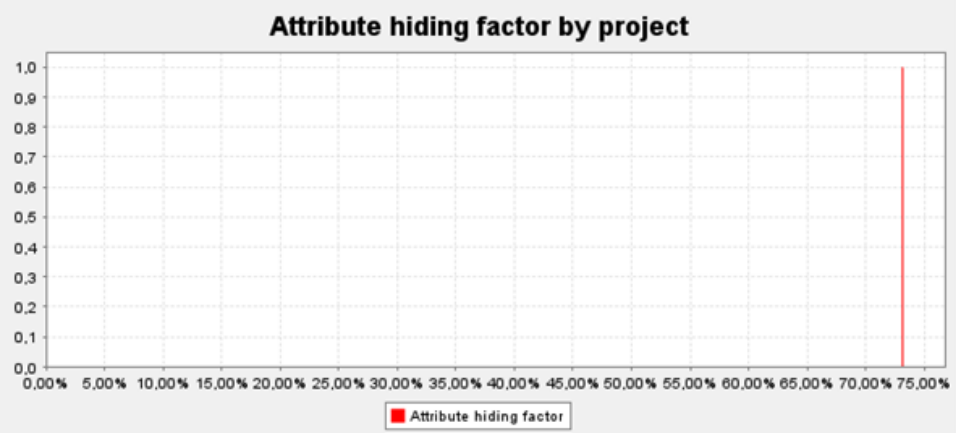
$A_d(C_i)$ = $A_v(C_i) + A_h(C_i)$: Attributes defined in C

$A_v(Ci)$: Attributes visible in the class $C_i$

TC: Total number of Classes.

Project metrics

| project | AHF | AIF | CF | MHF | MIF | PF |
|---|---|---|---|---|---|---|
| project | 73,09% | 43,83% | 2,71% | 25,32% | 73,03% | 6,83% |




Generally, a high AHF value is advisable, as the attributes of a class should be hidden from other classes, making 100% the ideal AHF value. Regarding our project, we have an Attribute Hiding Factor (AHF) of 73.09%, which indicates that the classes in this project follow a good practice of encapsulation. This metric suggests that the majority of attributes (instance variables) in the classes are well protected and not directly accessible by external classes.

# Attribute Inheritance Factor (AIF)

The Attribute Inheritance Factor (AIF) assesses the inheritance of attributes from a parent class to a child class. A high AIF indicates a high inheritance of attributes, which can increase complexity and coupling between classes. A low AIF is generally preferable as it reduces the dependency between classes. This factor can be calculated using the following formula:

$$AIF = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}$$

$A_h(C_i)$ **=** Inherited attributes

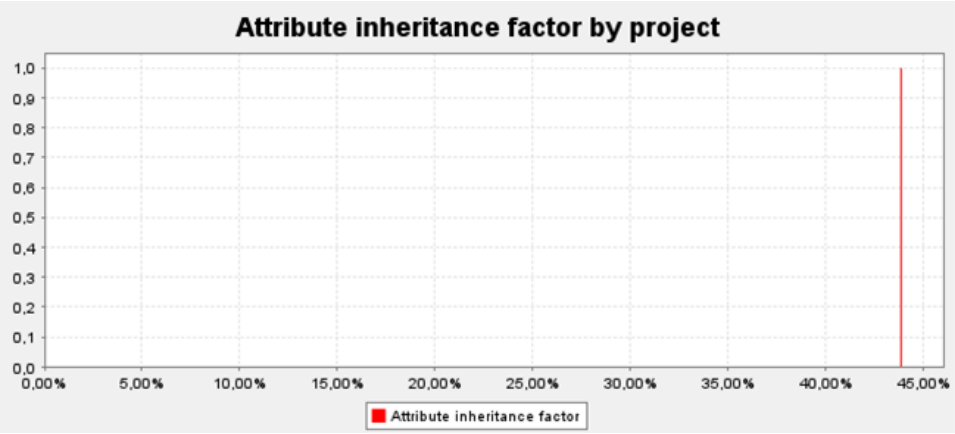
**$A_a(C_i)$** = $A_d(C_i) + A_h(C_i)$: Attributes defined in $C_i$

**$A_d(Ci)$:** Attributes defined in the class $C_i$

**TC:** Total number of Classes.

Project metrics

| project | AHF | AIF | CF | MHF | MIF | PF |
|---|---|---|---|---|---|---|
| project | 73,09% | 43,83% | 2,71% | 25,32% | 73,03% | 6,83% |




Generally, the range for AIF is between 0% and 48%. According to our program where we have a percentage of 43.83%, we can conclude that the inheritance of attributes between classes in the project is not very extensive, resulting in lower coupling between classes and reduced complexity.

# Coupling Factor (CF)

The Coupling Factor (CF) measures the dependency between classes in the source code. A low CF indicates that classes are loosely coupled, which is desirable to facilitate code maintenance and reusability. A high CF indicates that classes are tightly coupled and may be difficult to modify without affecting other parts of the system. This factor can be calculated using the following formula:

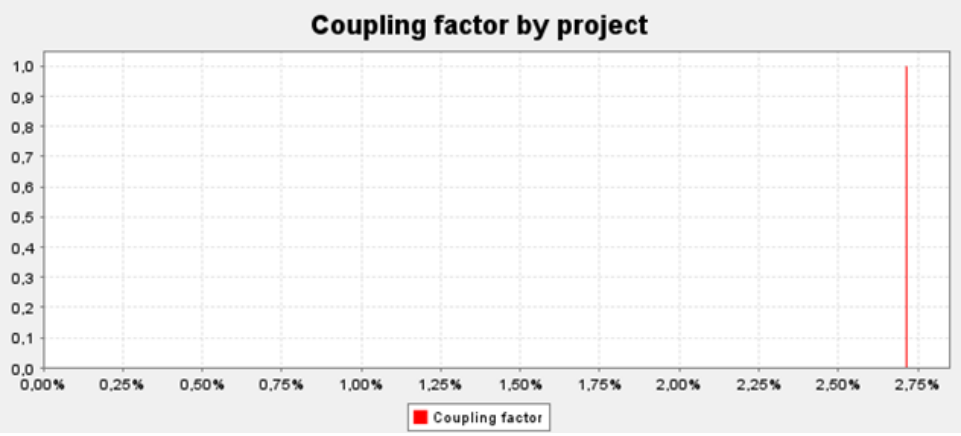
$$COF = \frac{\sum_{i=1}^{TC}\left[\sum_{j=1}^{TC} is\_client\ (C_i, C_j)\right]}{TC^2 - TC}$$

**is_client**($C_c$,$C_s$) =| 1 **if** ($C_i \Rightarrow C_j$)^($C_i \neq C_j$) , **else** 0

**TC:** Total number of Classes.

Project metrics

| project | AHF | AIF | CF | MHF | MIF | PF |
|---|---|---|---|---|---|---|
| project | 73,09% | 43,83% | 2,71% | 25,32% | 73,03% | 6,83% |




A high CF value indicates that the classes in the system are more interconnected and interdependent, leading to the problem that sometimes it's very difficult to change or fix the system in case of any bug or issue because the functionality where the bug resides could be implemented by more than two classes, and we have to make changes in all related classes. In our program analysis, the CF value is only 2.71%, and with such a low CF, the classes in the project are independent from each other, meaning that changes in one class tend to have minimal or no impact on other classes. This is positive as it facilitates code maintenance and modification.

# Method Hiding Factor (MHF)

The Method Hiding Factor (MHF) assesses the degree of encapsulation and hiding of methods (functions) within a class. A high MHF indicates that methods are well encapsulated, which is generally preferable to prevent external classes from accessing and modifying methods inappropriately. This factor can be calculated using the following formula:

**$M_h(C_i)$ =** Hidden methods in the class $C_i$

$$MHF = \frac{\sum_{i=1}^{TC} M_h(C_i)}{\sum_{i=1}^{TC} M_d(C_i)}$$

$M_d(C_i) = M_v(C_i) + M_h(C_i)$: Methods defined in $C_i$

**$M_v(Ci)$:** Visible methods in the class $C_i$

**TC:** Total number of Classes.

Project metrics

| project | AHF | AIF | CF | MHF | MIF | PF |
|---|---|---|---|---|---|---|
| project | 73,09% | 43,83% | 2,71% | 25,32% | 73,03% | 6,83% |


A low MHF indicates an insufficiently abstract implementation. A large proportion of methods are unprotected, and the likelihood of errors is high. A high MHF indicates too little functionality. It may also indicate that the design or model includes a high proportion of specialized methods that are not available for reuse. An acceptable MHF value ranges from 8% to 25%. In alignment with our program, a Method Hiding Factor (MHF) of 25.32% in a software project indicates a moderate level of method encapsulation and hiding, allowing us to use and reuse a substantial number of methods while maintaining a sufficiently abstract implementation, resulting in good program functionality.

# Method Inheritance Factor (MIF)

The Method Inheritance Factor (MIF) measures the inheritance of methods from parent classes to child classes. A high MIF indicates a high inheritance of methods, which can increase complexity and coupling between classes. A low MIF is generally preferable to reduce the dependency between classes. This factor can be calculated using the following formula:

$$MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)}$$

**$M_i$:** Inherited methods

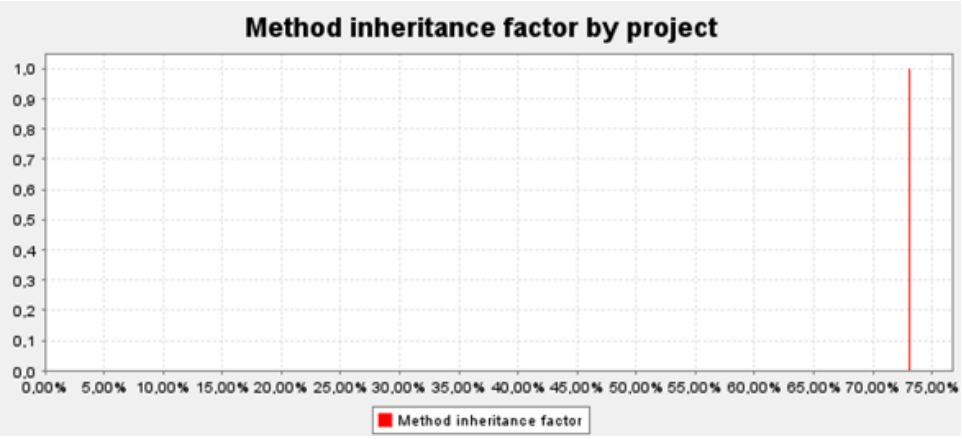
**$M_a(C_i)$** = $M_d(C_i)$ + $M_i(C_i)$: Methods defined in $C_i$

**$M_d(Ci)$:** Methods defined in the class $C_i$

**TC:** Total number of Classes.

Project metrics

| project | AHF | AIF | CF | MHF | MIF | PF |
|---|---|---|---|---|---|---|
| project | 73,09% | 43,83% | 2,71% | 25,32% | 73,03% | 6,83% |




At first glance, we might be tempted to think that inheritance should be used extensively. However, composing multiple inheritance relationships builds a directed acyclic graph (a hierarchy tree of inheritance) whose depth and width can quickly erode comprehensibility and testability. Generally, the MIF range falls between 20% to 80%. According to our values, a Method Inheritance Factor (MIF) of 73.03% in a software project indicates a high inheritance of methods from parent classes to child classes. This suggests strong functionality reuse, providing us with good program comprehensibility and testability.

# Polymorphism Factor (PF)

The Polymorphism Factor (PF) assesses the use of polymorphism in the code. Polymorphism allows objects of different classes to be treated uniformly, which can make the code more flexible and extensible. In polymorphism, the child class can implement the method differently. The same method can be implemented differently in the child class and the parent class. It is defined by the ratio between the actual number of method substitutions and the maximum total number of method substitutions. This factor can be calculated using the following formula:

$$POF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC} [M_n(C_i) \times DC(C_i)]}$$

$M_o(C_i)$**:** Overridden methods in the class $C_i$

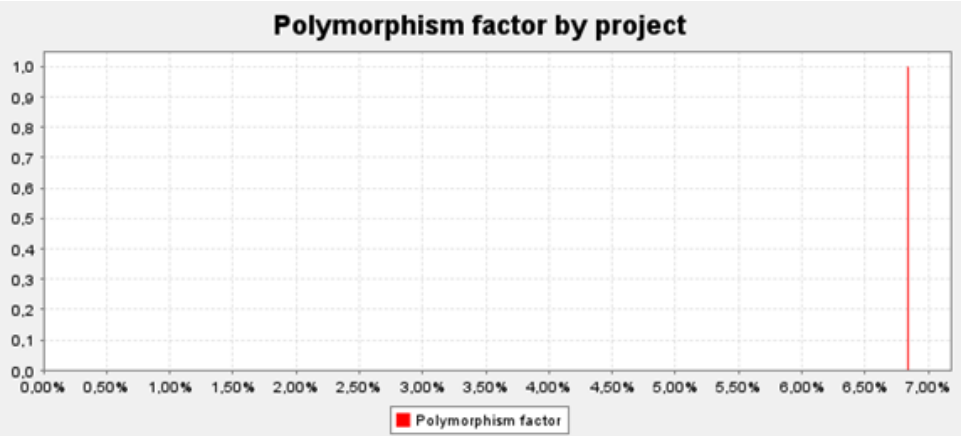
**$M_n(Ci)$:** New Methods in $C_i$

**$D_C(C_i)$:** Number of descendants of class Ci (derived classes)

**TC:** Total number of Classes.

Project metrics

| project | AHF | AIF | CF | MHF | MIF | PF |
|---|---|---|---|---|---|---|
| project | 73,09% | 43,83% | 2,71% | 25,32% | 73,03% | 6,83% |




Polymorphism arises from inheritance and has its pros and cons. Intuitively, we might expect that polymorphism (overrides) can be used to a reasonable extent to keep the code clear, but excessively polymorphic code can be very complex to understand (as several alternative methods can be executed for a single method call). The PF should be within a reasonable range with both lower and upper limits. When analyzing the PF in our program, we have a Polymorphism Factor (PF) of 6.83%, indicating low use of polymorphism in the project. We can conclude that we have a system that is sufficiently clear and clean with reasonable complexity, allowing for better understanding.

# Dependency Metrics Nádia Mendes 53175

## Summary:

- CYCLIC (Number of cyclic dependencies) measures, for each class c, the number of classes c directly depends on, and that in turn depend on c.

-DCY (Number of dependencies) measures, for each class c, the number of classes c directly depends on.

-DCY* (Number of transitive dependencies) measures, for each class c, the number of classes c directly or indirectly depends on.

-DPT (Number of dependants) measures, for each class c, the number of classes that directly depend on c.

-DPT* (Number of transitive dependants) measures for each class c, the number of classes that directly or indirectly depend on c.

-PDCY (Parse distance or Parse depth) this variable generally measures the distance or depth in the parse tree between two elements in a sentence. This metric is used to evaluate how far apart or close together elements are in the sentence structure.

-PDPT (Parse tree depth) measures the depth of an element in the parse tree of a sentence.

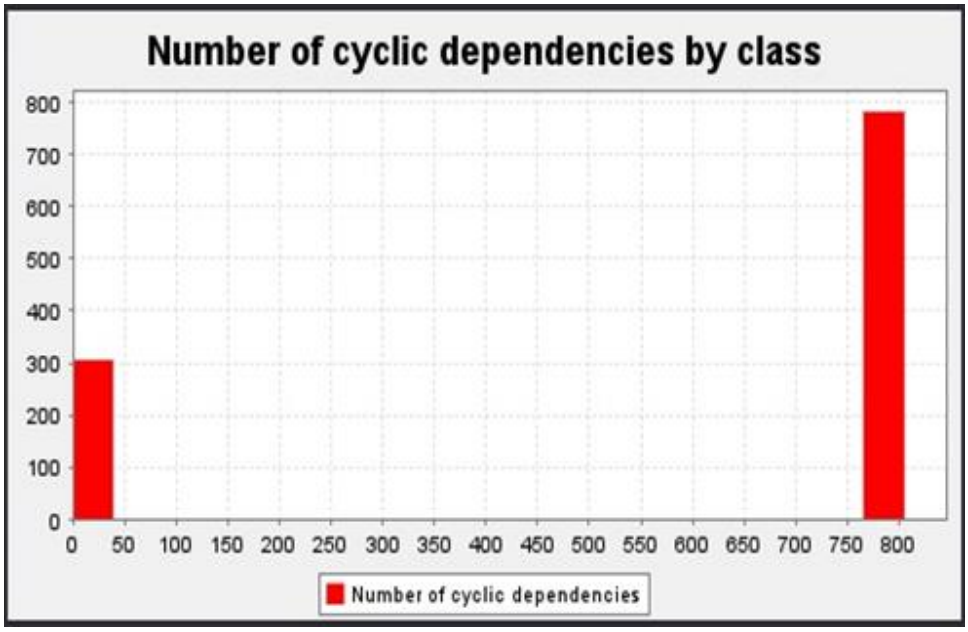
## Data Visualization –

Class metrics | Interface metrics | Package metrics

| class | Cyclic | Dcy | Dcy* | Dpt | Dpt* | PDcy | PDpt |
|---|---|---|---|---|---|---|---|
| net.sf.freecol.server.model.NativeTradeSession | 805 | 5 | 989 | 1 | 923 | 3 | 1 |
| net.sf.freecol.server.model.ServerBuilding | 805 | 23 | 989 | 13 | 923 | 4 | 6 |
| net.sf.freecol.server.model.ServerColony | 805 | 47 | 989 | 12 | 923 | 5 | 6 |
| net.sf.freecol.server.model.ServerColonyTile | 805 | 20 | 989 | 2 | 923 | 4 | 2 |
| net.sf.freecol.server.model.ServerEurope | 805 | 26 | 989 | 4 | 923 | 6 | 3 |
| net.sf.freecol.server.model.ServerGame | 805 | 46 | 989 | 12 | 923 | 7 | 7 |
| net.sf.freecol.server.model.ServerIndianSettlement | 805 | 35 | 989 | 9 | 923 | 5 | 6 |
| net.sf.freecol.server.model.ServerPlayer | 805 | 96 | 989 | 124 | 923 | 9 | 12 |
| net.sf.freecol.server.model.ServerRegion | 805 | 18 | 989 | 7 | 923 | 4 | 3 |
| net.sf.freecol.server.model.ServerUnit | 805 | 63 | 989 | 61 | 923 | 6 | 12 |
| net.sf.freecol.server.model.Session | 805 | 3 | 989 | 10 | 923 | 2 | 3 |
| net.sf.freecol.server.model.TimedSession | 805 | 2 | 989 | 2 | 923 | 2 | 1 |
| net.sf.freecol.server.networking.DummyConnection | 805 | 3 | 989 | 2 | 923 | 2 | 2 |
| net.sf.freecol.server.networking.Server | 805 | 5 | 989 | 3 | 923 | 3 | 3 |
| Total | | | | | | | |
| Average | 579,56 | 10,95 | 817,49 | 10,48 | 807,06 | 3,61 | 2,78 |

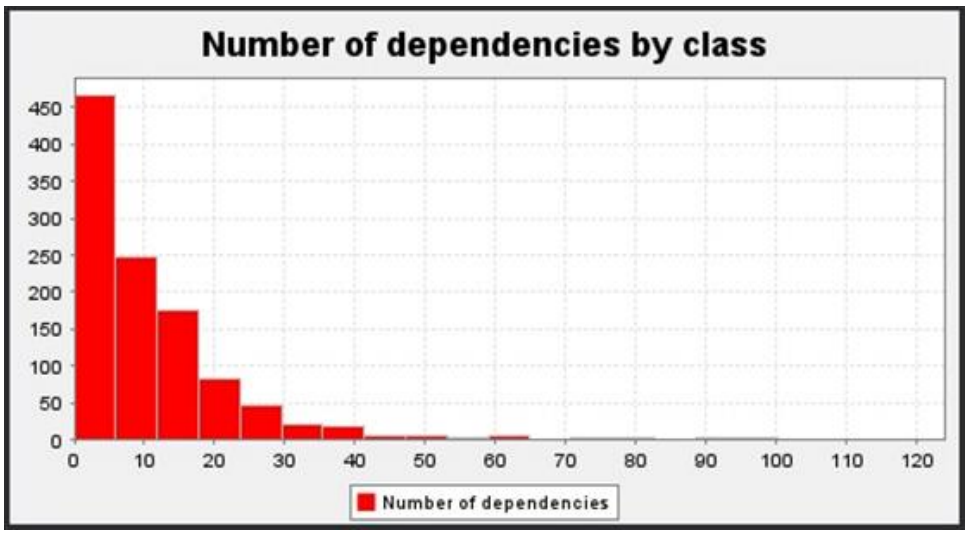
**Top 5 CYCLIC –**

| | CYCLIC | DCY | DCY* | DPT | DPT* | PDCY | PDPT |
|---|---|---|---|---|---|---|---|
| net.sf.freecol.server.networking.Server | 805 | 63 | 989 | 61 | 923 | 6 | 12 |
| net.sf.freecol.server.networking.DummyConnection | 805 | 3 | 989 | 10 | 923 | 2 | 3 |
| net.sf.freecol.server.model.TimedSession | 805 | 2 | 989 | 2 | 923 | 2 | 1 |
| net.sf.freecol.server.model.Session | 805 | 3 | 989 | 2 | 923 | 2 | 2 |
| net.sf.freecol.server.model.ServerUnit | 805 | 5 | 989 | 3 | 923 | 3 | 3 |




**Top 5 DCY –**

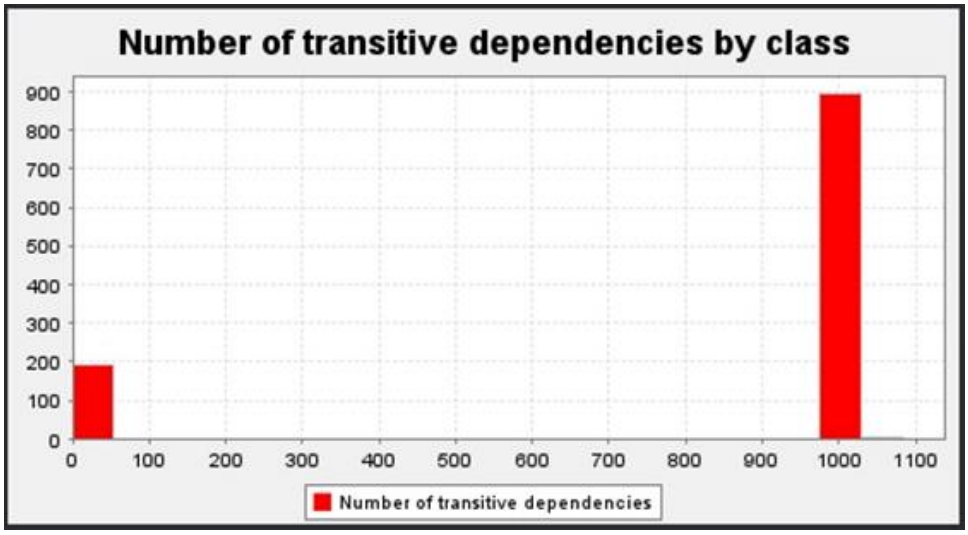
| | CYCLIC | DCY | DCY* | DPT | DPT* | PDCY | PDPT |
|---|---|---|---|---|---|---|---|
| net.sf.freecol.server.control.InGameController | 805 | 118 | 989 | 83 | 923 | 11 | 1 |
| net.sf.freecol.client.gui.Widgets | 805 | 111 | 989 | 2 | 923 | 10 | 12 |
| net.sf.freecol.common.networking.ServerAPI | 805 | 104 | 989 | 10 | 923 | 3 | 5 |
| net.sf.freecol.server.model.ServerPlayer | 805 | 96 | 989 | 124 | 923 | 9 | 1 |
| net.sf.freecol.client.gui.SwingGUI | 805 | 95 | 989 | 1 | 923 | 20 | 9 |




**Top 5 DCY* –**

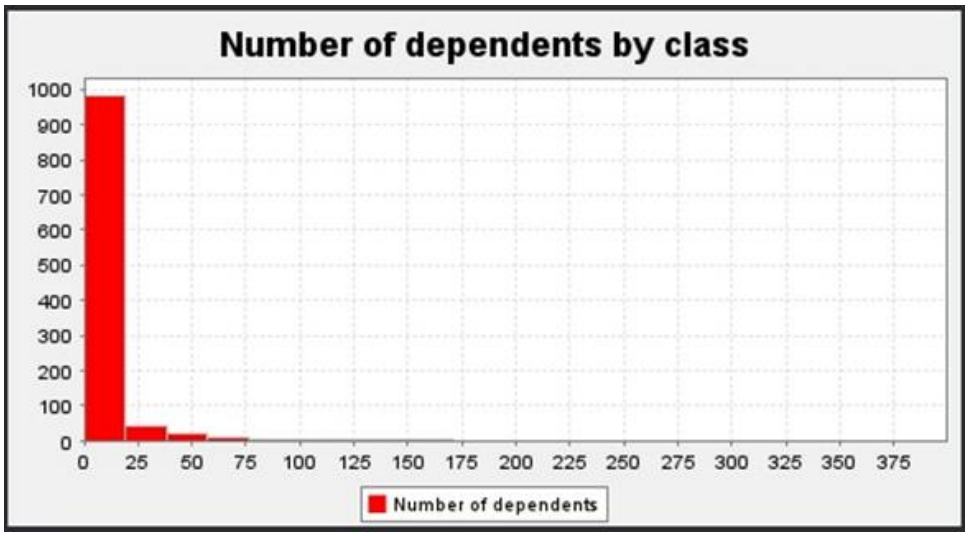
| | CYCLIC | DCY | DCY* | DPT | DPT* | PDCY | PDPT |
|---|---|---|---|---|---|---|---|
| net.sf.freecol.AllTests | 0 | 5 | 1083 | 0 | 0 | 5 | 0 |
| net.sf.freecol.common.AllTests | 0 | 5 | 1050 | 1 | 1 | 5 | 1 |
| net.sf.freecol.common.model.AllTests | 0 | 38 | 1040 | 1 | 2 | 1 | 1 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| net.sf.freecol.server.AllTests | 0 | 4 | 1022 | 1 | 1 | 4 | 1 |
| net.sf.freecol.server.ai.AllTests | 0 | 8 | 1011 | 1 | 2 | 2 | 1 |




**Top 5 DPT –**

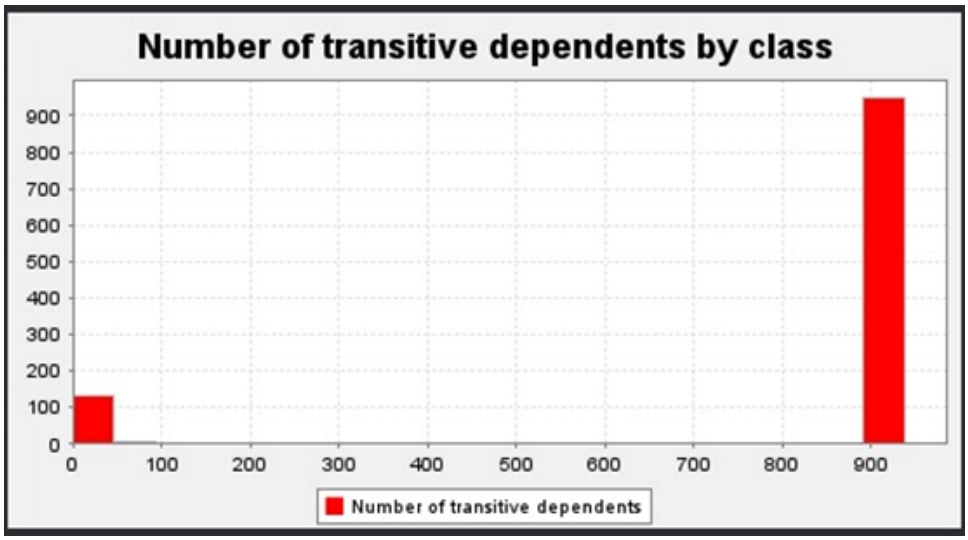
| | CYCLIC | DCY | DCY* | DPT | DPT* | PDCY | PDPT |
|---|---|---|---|---|---|---|---|
| net.sf.freecol.common.model.FreeColObject | 805 | 5 | 1083 | 380 | 923 | 4 | 34 |
| net.sf.freecol.common.model.Player | 805 | 5 | 1050 | 335 | 923 | 7 | 26 |
| net.sf.freecol.common.model.Game | 805 | 38 | 1040 | 328 | 923 | 7 | 27 |
| net.sf.freecol.common.model.Unit | 805 | 4 | 1022 | 297 | 923 | 6 | 25 |
| net.sf.freecol.client.FreeColClient | 805 | 8 | 1011 | 294 | 923 | 15 | 16 |




**Top 5 DPT* –**

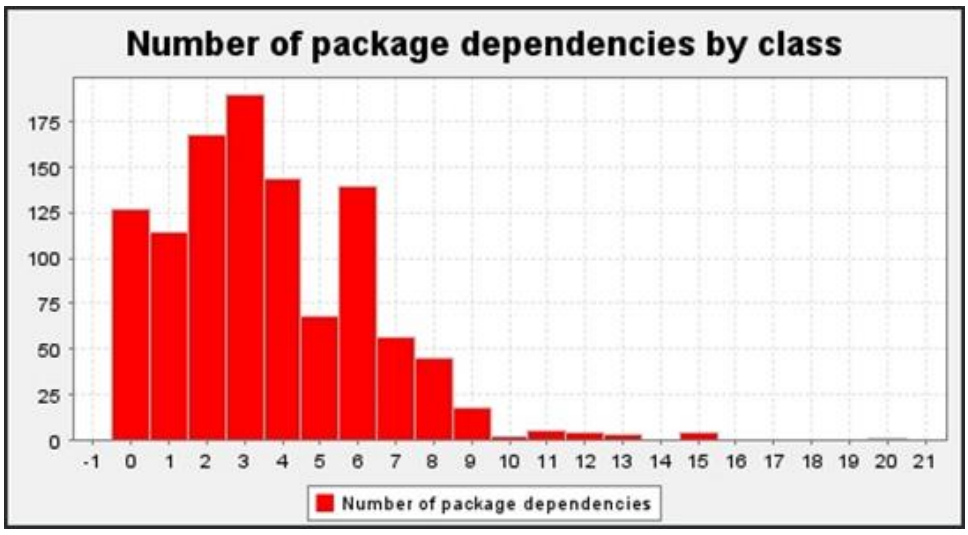
| | CYCLIC | DCY | DCY* | DPT | DPT* | PDCY | PDPT |
|---|---|---|---|---|---|---|---|
| net.sf.freecol.common.util.Utils | 0 | 1 | 1 | 67 | 937 | 0 | 20 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| net.sf.freecol.common.util.StringUtils | 0 | 1 | 1 | 53 | 936 | 0 | 17 |
| net.sf.freecol.common.resources.Resource | 0 | 0 | 0 | 16 | 935 | 0 | 3 |
| net.sf.freecol.common.util.CachingFunction | 0 | 0 | 0 | 3 | 933 | 0 | 2 |
| net.sf.freecol.common.util.CollectionUtils | 0 | 3 | 3 | 165 | 932 | 1 | 32 |




**Top 5 PDCY –**

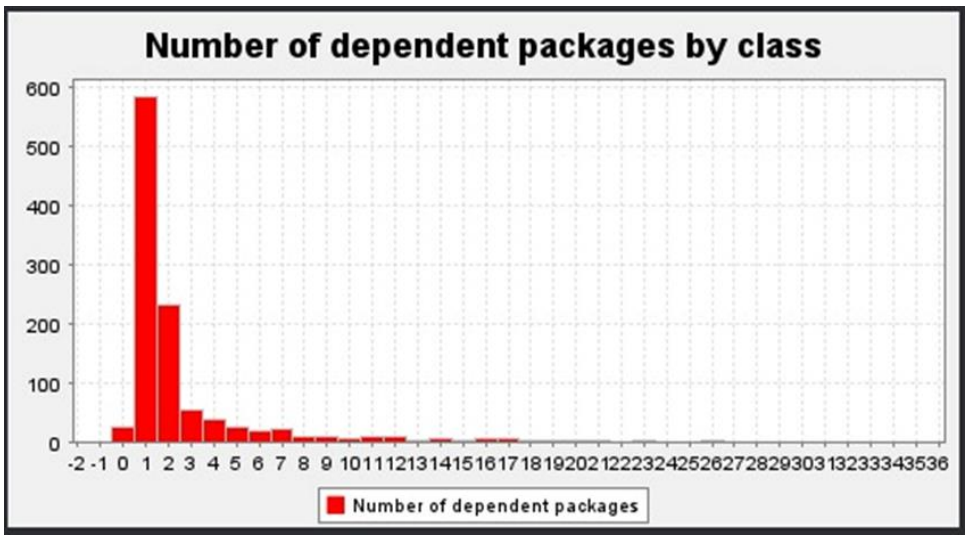
| | CYCLIC | DCY | DCY* | DPT | DPT* | PDCY | PDPT |
|---|---|---|---|---|---|---|---|
| net.sf.freecol.client.gui.SwingGUI | 805 | 95 | 989 | 1 | 923 | 20 | 1 |
| net.sf.freecol.server.FreeColServer | 805 | 59 | 989 | 138 | 923 | 15 | 15 |
| net.sf.freecol.common.debug.DebugUtils | 805 | 64 | 989 | 8 | 923 | 15 | 5 |
| net.sf.freecol.client.gui.GUI | 805 | 71 | 989 | 153 | 923 | 15 | 15 |
| net.sf.freecol.client.gui.FreeColClient | 805 | 37 | 989 | 294 | 923 | 15 | 16 |




**Top 5 PDPT –**

| | CYCLIC | DCY | DCY* | DPT | DPT* | PDCY | PDPT |
|---|---|---|---|---|---|---|---|

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| net.sf.freecol.common.model.FreeColObject | 805 | 17 | 989 | 338 | 923 | 4 | 34 |
| net.sf.freecol.common.util.CollectionUtils | 0 | 3 | 3 | 165 | 923 | 1 | 32 |
| net.sf.freecol.common.model.Specification | 805 | 65 | 989 | 262 | 923 | 5 | 29 |
| net.sf.freecol.common.model.Game | 805 | 50 | 989 | 328 | 923 | 7 | 27 |
| net.sf.freecol.common.model.Player | 805 | 81 | 989 | 325 | 923 | 7 | 26 |




## Discussion-

As can be seen in the first figure 2, the 5 classes with the highest Number of cyclic dependencies (CYCLIC) are contained in the net.sf.freecol.server.networking package and the net.sf.freecol.server.model package, these present a high number of cyclic dependencies, indicating that the project is complex, with many dependency relationships between its components. Which makes the code more difficult to understand and modify.

As can be seen in the first figure 2, the 5 classes with the highest Number of cyclic dependencies (CYCLIC) are contained in both the net.sf.freecol.server.networking package and the net.sf.freecol.server.model package, These present a high number of cyclic dependencies, indicating that the project is complex, with many dependency relationships between its components. Which makes the code more difficult to understand and modify.

It can also be seen that due to the existence of so many cyclical dependencies, the project becomes more difficult to test, since it is more difficult to isolate the components for unit testing. Which can impact the quality of the software.

A possible solution would be to refactor the project, which would involve restructuring the code to reduce or eliminate cyclical dependencies, thus making the project easier to modulate and maintain.

Regarding the number of dependencies by class (DCY), we can conclude that there are classes that present very high values, which suggests that there is a high level of coupling between the project components, as is the case with the classes identified above, however it is verified that there is also the existence of classes whose value is relatively low or even null. The average is 10.95, which suggests that it is a relatively good value.

When we analyze the number of transitive dependencies (DCY*) we see that in this project there is a high number of classes with a high value, and on average each class has around 817.49 indirect dependencies in relation to other components, thus it appears that in these classes the code is more difficult to understand and manage. We also verified that once again due to this relatively high value the project becomes difficult to maintain since with many transitive dependencies changes to an indirect dependency can affect many components, requiring extensive testing and validation, and this high value can also affect project performance, since more resources may be required to load and manage all dependencies.

Regarding the number of dependents (DPT), with an average of 10.48, we can see that in terms of direct dependencies, the project has a moderate level of coupling.

Regarding the number of transitive dependencies (DPT*), we found an average of 807.06, this value is significantly high, which suggests that, in addition to direct dependencies, there are many transitive dependencies (i.e., indirect dependencies, dependency dependencies). This indicates that changes in one component can potentially affect many other components, including those that indirectly depend on the component in question. The value of Parse distance or Parse depth has an average of 3.61, which indicates that on average the syntactic analysis of the code reaches a depth of 3.61 levels, therefore we can consider it as a moderate depth, which suggests that the code does not present excessive complexity in terms of analysis structure.

Finally, we verified that the Parse tree deth (PDPT) value presented a value of 2.78, which indicates that the parse tree structure is not very deep, which is positive, as excessive depth would make the code more difficult to understand.

# Martin Packaging Metrics José Morgado 59457

## Summary:

- Efferent Coupling (Ce): This metric quantifies the number of classes outside the module that directly depend on classes within the module. It measures the outgoing dependencies from a module.
- Afferent Coupling (Ca): Afferent Coupling counts the number of classes outside the module that depend directly on classes within the module. It measures the incoming dependencies to a module.
- Instability (I): Instability is a metric that represents the module's tendency to change. It's calculated as the ratio of efferent coupling to the sum of efferent and afferent couplings (Ce / (Ce + Ca)). A higher instability suggests the module is more prone to change.
- Abstractness (A): Abstractness measures the ratio of the number of abstract classes/interfaces in a module to the total number of classes/interfaces in that module. It indicates how abstract or concrete a module is.

- Normalized Distance from Main Sequence (D): This metric determines how far a module is from the optimal balance between abstraction and stability, represented by the Main Sequence. It's calculated as |A + I - 1|, where A is abstractness, and I is instability. Modules closer to the Main Sequence are considered more ideal in terms of design.

## Data Visualization:

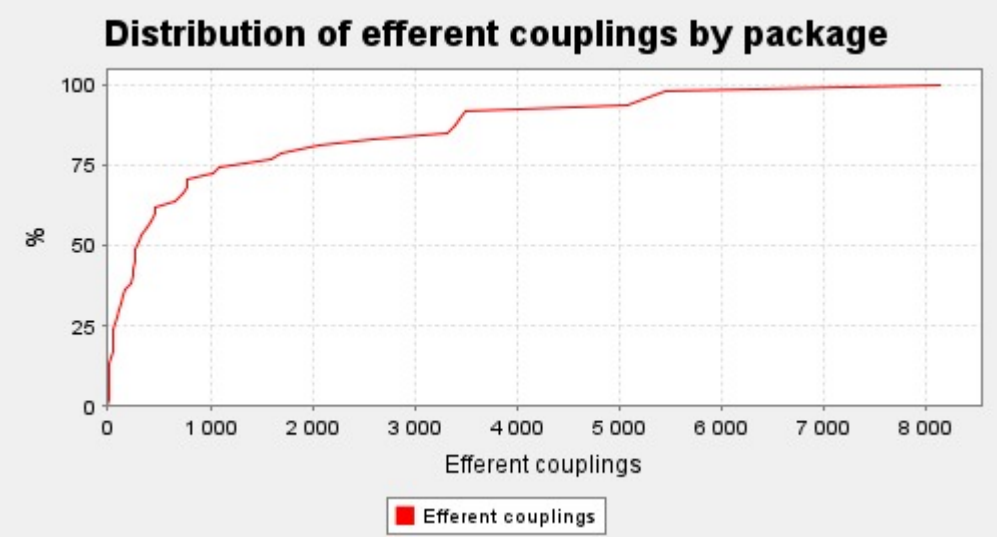



Fig. 1 - Efferent Coupling (Ce)

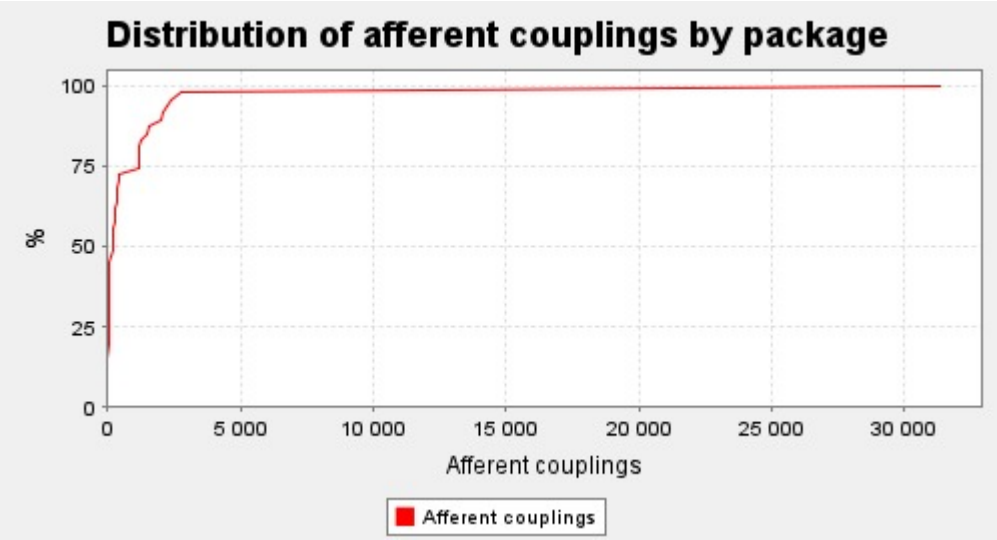



Fig. 2 - Afferent Coupling (Ca)

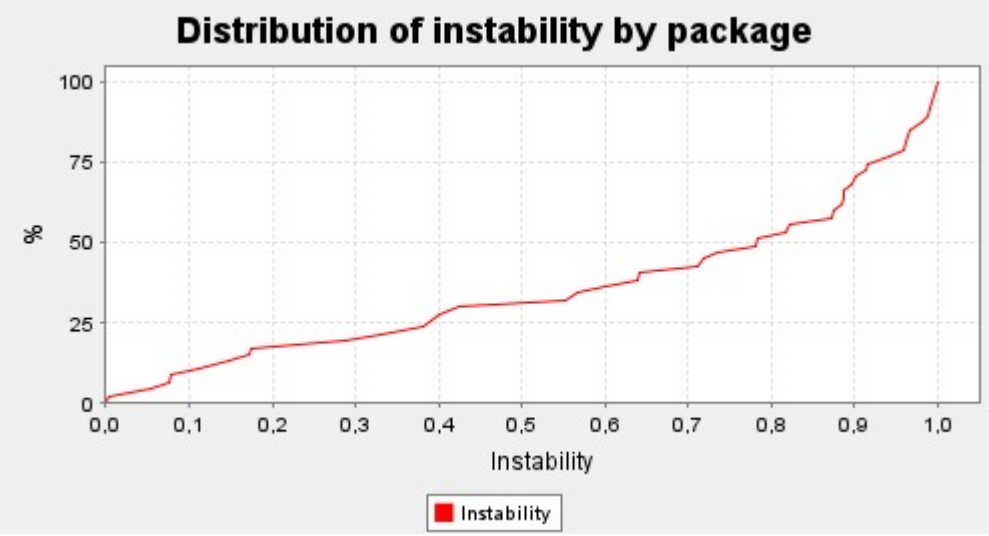


Fig. 3 - Instability (I)

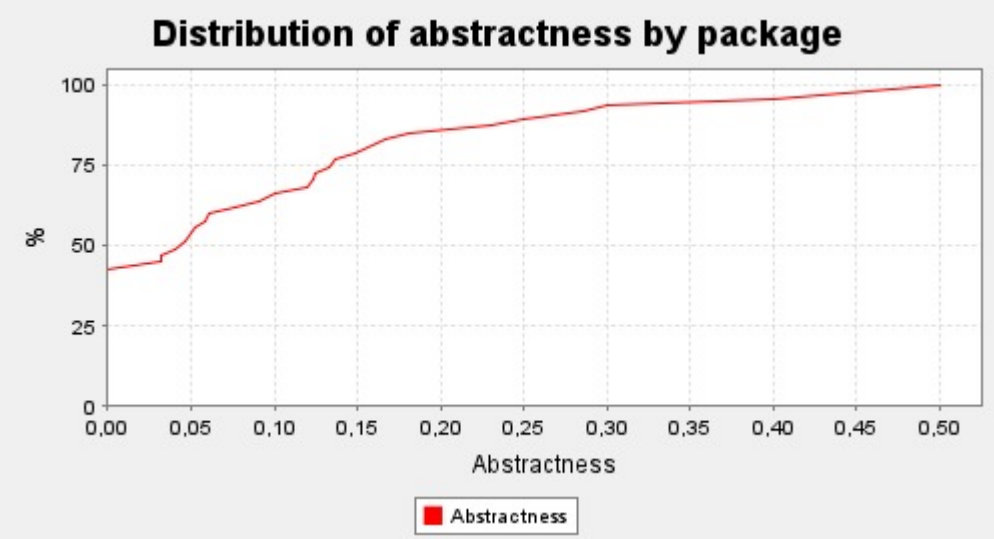


Fig. 4 - Abstractness (A)

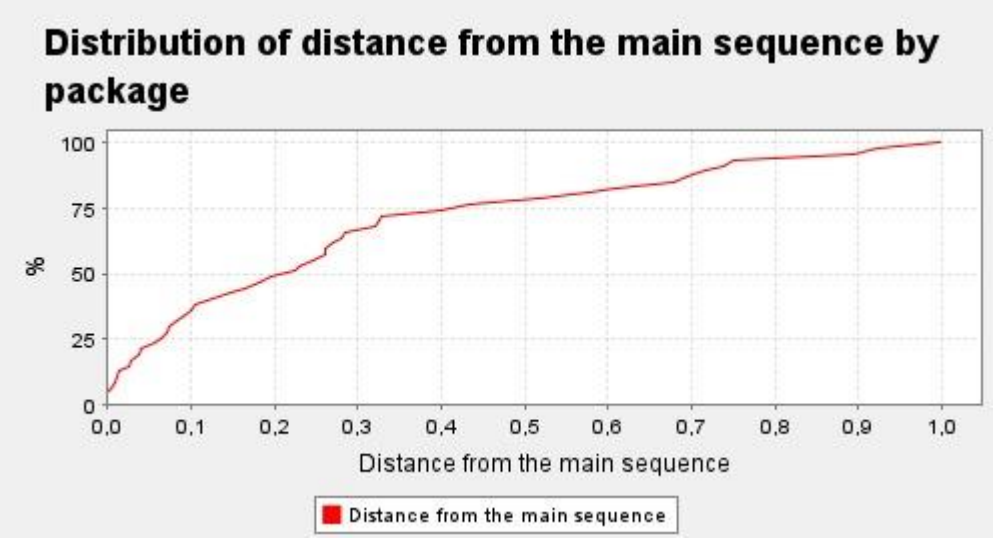


Fig. 5 - Instability (I)

## Discussion:

**Average Efferent Couplings (ce = 1178.09):**

The average efferent couplings suggest that, on average, the classes in the project depend on many external classes. This indicates a high degree of class dependency on other parts of the system.

**Average Afferent Couplings (ca = 1178.09):**

The average afferent couplings indicate that, on average, the classes in the project have many external dependencies. This might suggest an architecture where various parts of the system heavily rely on a central set of classes.

**Average Instability (i = 0.51):**

The average instability indicates that, on average, the classes in the project are moderately unstable, with a moderate propensity for changes. A value close to 0.5 suggests a reasonable balance between stability and instability.

**Average Abstractness (a = 0.08):**

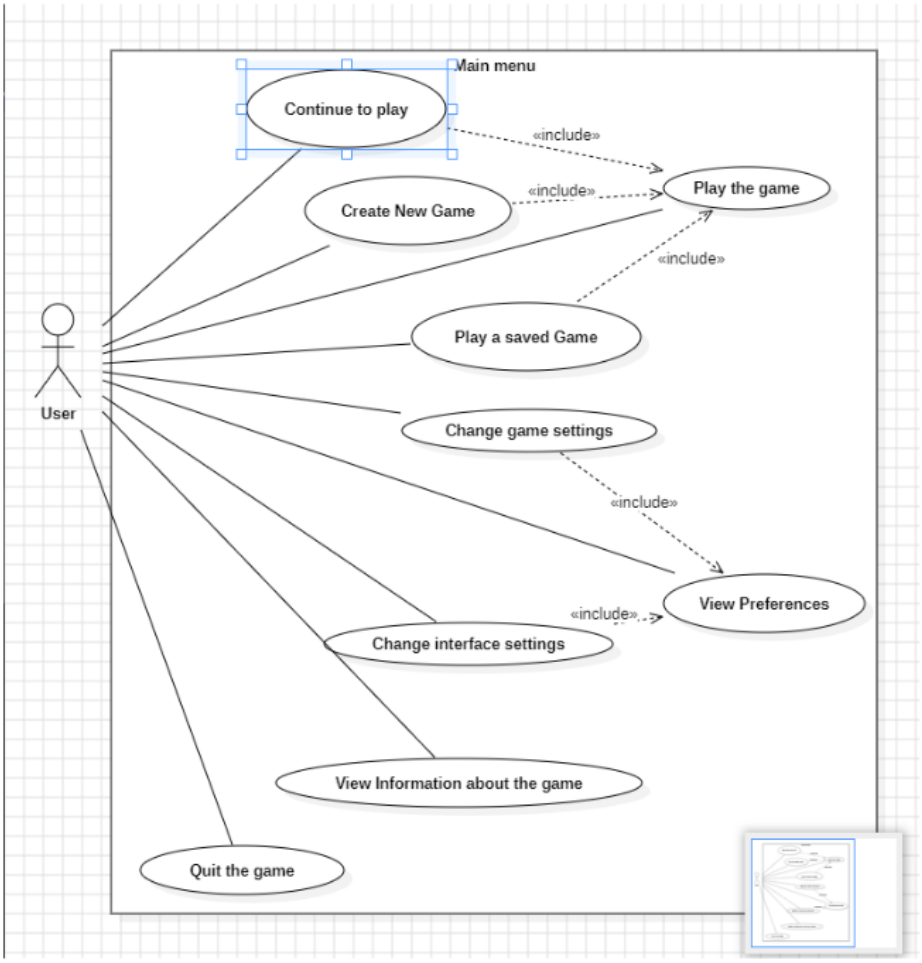
The average abstractness suggests that, on average, the classes in the project are relatively non-abstract, with more concrete implementations than abstract methods.

**Average Distance (d = 0.29):**

The average distance suggests that, on average, the classes in the project are in a reasonable balance between abstraction and coupling, although there might be room for improvements.

# Use Case Diagram:

## João Amorim:




--Main Menu—

Use Cases:

Name: Continue to play

Description: The user can continue playing from the last autosave.

Primary actor: User

--

Name: Create New Game

Description: Create a new game, being able to select from a lot of options like if the user wants

to play multiplayer or even what kingdom they wish to play as.

Primary actor: User

--

Name: Play a saved game

Description: The user can select from the saved games he has one to resume playing.

Primary actor: User

--

Name: Play the game

Description: The act of playing freecol.

Primary actor: User

--

Name: Change game settings

Description: The user can view a list of game settings and change them.

Primary actor: User

--

Name: Change interface settings

Description: The user can see a list of interface settings and change them.

Primary actor: User

--

Name: View preferences

Description: The user can see a list of interface settings and change them.

Primary actor: User

--

Name: View information about the game

Description: The user can see all his preferences and settings that it can change.

Primary actor: User

--

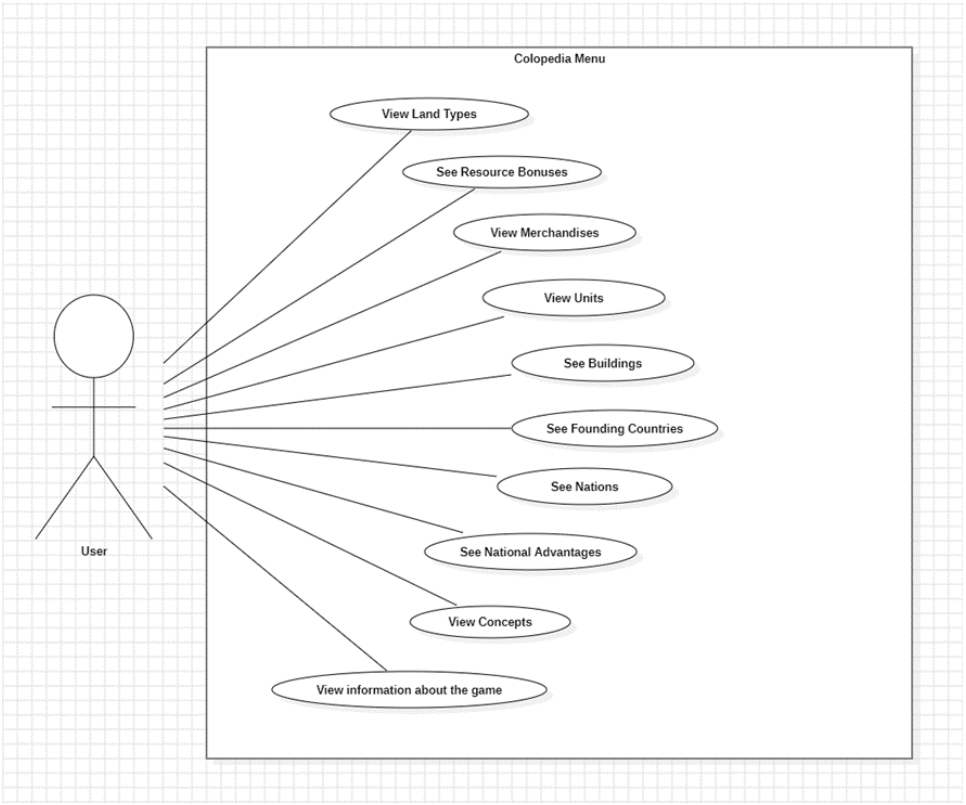
Name: Quit the game

Description: The user can quit the game by pressing a button.

Primary actor: User


# João Esteves 47994


# Colopedia Menu




# Use Cases

Name: View Land Types

Description: The User can consult the types of terrain available in the game, as well as their information.

Primary Actor: User

--

Name: See Resource Bonuses

Description: The User can consult resource bonuses as well as their information.

Primary Actor: User

--

Name: View Merchandises

Description: The User can consult the goods available in the game and have access to their information.

Primary Actor: User

--

Name: View Units

Description: The User can consult the units available in the game and have access to their information.

Primary Actor: User

--

Name: See Buildings

Description: The User can consult the buildings available in the game and have access to their information.

Primary Actor: User

--

Name: See Founding Countries

Description: The User can consult information about the founding countries.

Primary Actor: User

--

Name: See Nations

Description: The User can consult the nations available in the game, as well as their information.

Primary Actor: User

--

Name: See National Advantages

Description: The User can consult information about national advantages.

Primary Actor: User

--

Name: View Concepts

Description: The User can consult information about different concepts present in the game.

Primary Actor: User

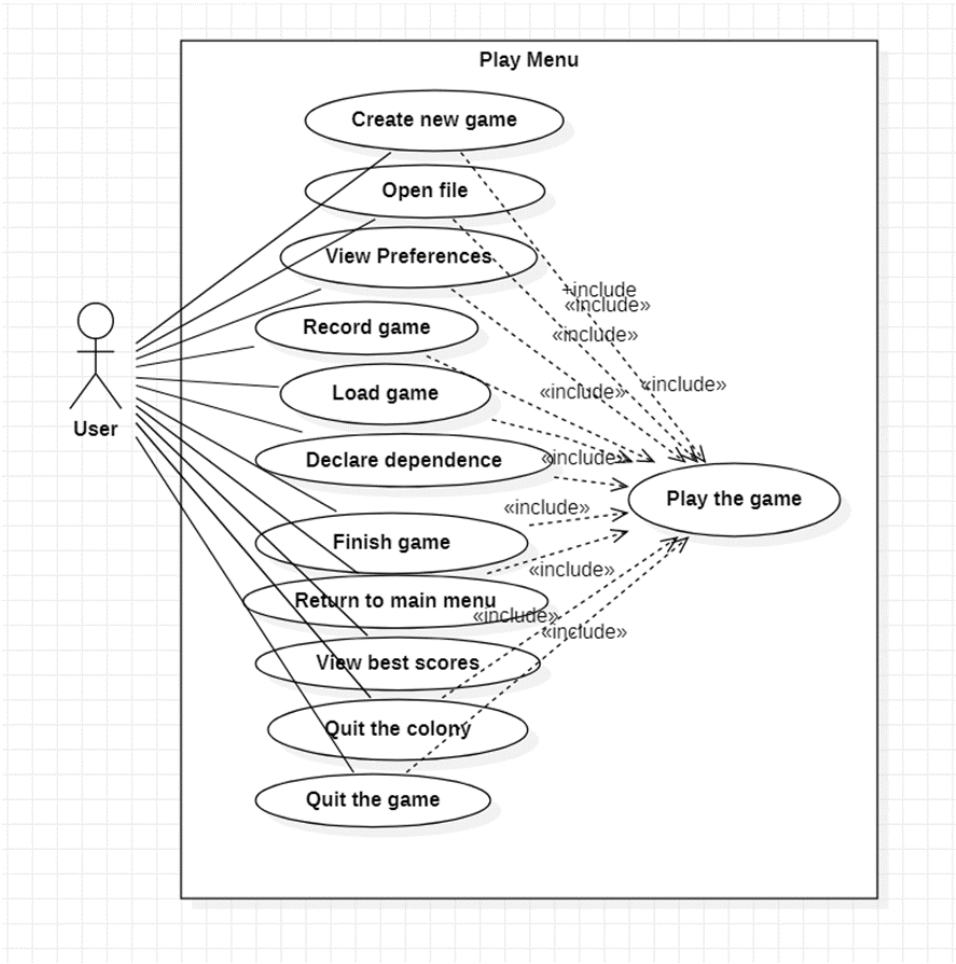
--

Name: View information about the game

Description: The User can check the game credits.

Primary Actor: User

# Nádia Mendes 53175




# Use cases:

Name: Create new game

Description: The user can create a new game.

Primary actor: User

--

Name: Open file

Description: The user can open a file chosen from their computer.

Primary actor: User

--

Name: View preferences

Description: The user can see the preferences.

Primary actor: User

--

Name: Record game

Description: The user can record his game.

Primary actor: User

--

Name: Load game

Description: The user reconnects the game from the last autosave.

Primary actor: User

--

Name: Declare dependence

Description: The user can choose declare dependence.

Primary actor: User

--

Name: Finish game

Description: The user can finish the game.

Primary actor: User

--

Name: Return to main menu

Description: The user can return to the main menu.

Primary actor: User

--

Name: View best scores

Description: The user can see the best scores of the game.

Primary actor: User

--

Name: Quit the colony

Description: The user can quit from his colony.

Primary actor: User

--

Name: Quit the game

Description: The user can quit from his game.

Primary actor: User

actor: User

--

Name: Play the game Description:

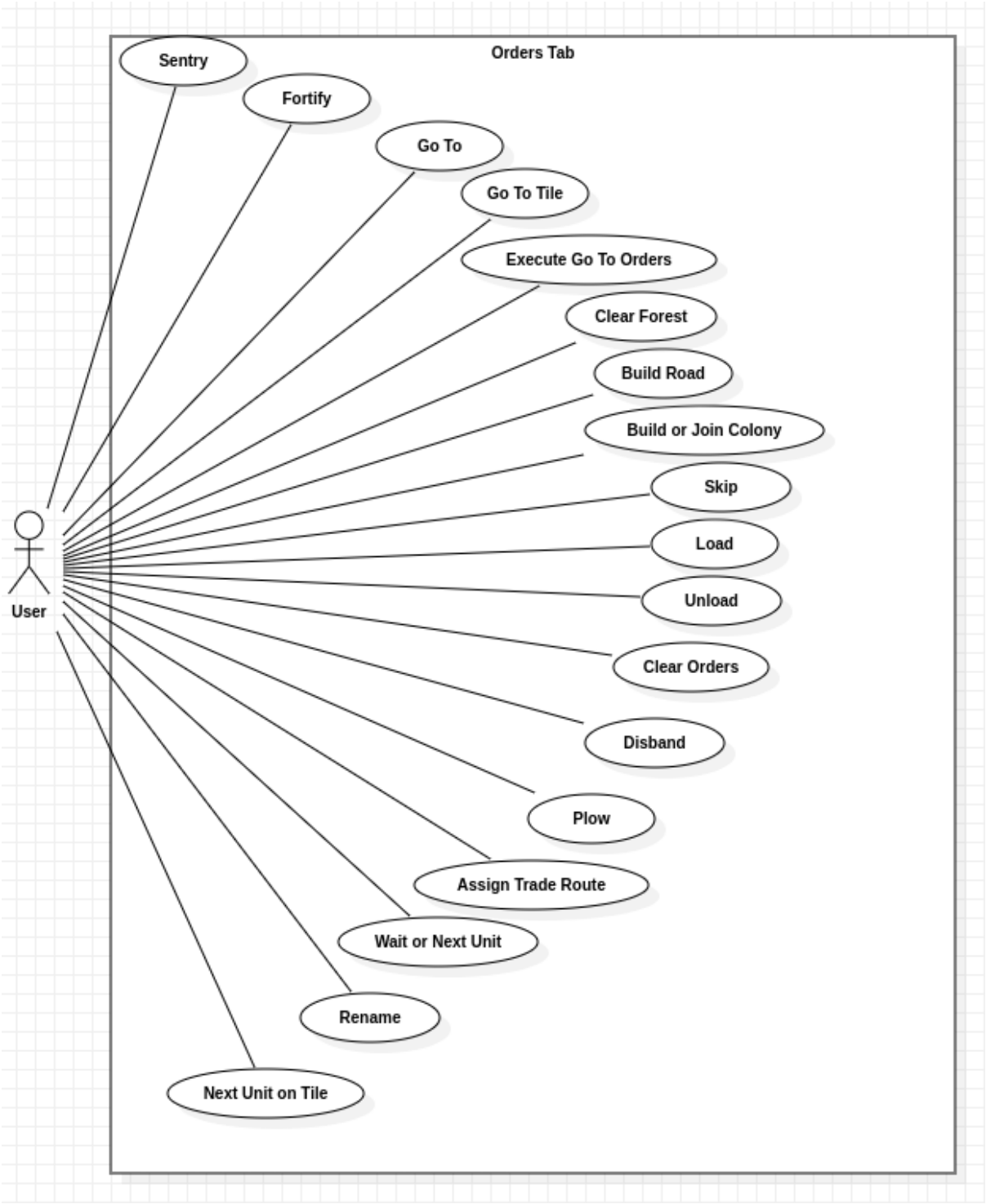
Description: The act of playing freecol.

Primary actor: User

--

# José Morgado 59457

# Use Cases - Orders Tab




**Name:** Sentry

**Description:** The user commands the unit to enter “sentry” mode, waiting for something to happen to it.

**Primary Actor:** User

–

**Name:** Fortify

**Description:** The user directs the unit to establish a defensive stance, reinforcing its position to repel potential threats.

**Primary Actor:** User

–

**Name:** Go To

**Description:** The user instructs the unit to move to a specified destination.

**Primary Actor:** User

–

**Name:** Go To Tile

**Description:** The user commands the unit to navigate to a particular tile on the map.

**Primary Actor:** User

–

**Name:** Execute Go To Orders

**Description:** The user prompts the unit to execute a series of movement commands, following the designated path.

**Primary Actor:** User

–

**Name:** Clear Forest

**Description:** The user commands the unit to clear a forested area.

**Primary Actor:** User

–

**Name:** Build Road

**Description:** The user directs the unit to construct a road

**Primary Actor:** User

–

**Name:** Build or Join Colony

**Description:** The user commands the unit to engage in the establishment or reinforcement of a colony.

**Primary Actor:** User

–

**Name:** Skip

**Description:** The user instructs the unit to skip its turn

**Primary Actor:** User

–

**Name:** Load

**Description:** The user directs the unit to load items or individuals (in the case of ships, for example) onto itself

**Primary Actor:** User

–

**Name:** Unload

**Description:** The user commands the unit to offload its cargo.

**Primary Actor:** User

–

**Name:** Clear Orders

**Description:** The user directs the unit to clear all pending orders.

**Primary Actor:** User

–

**Name:** Disband

**Description:** The user orders the unit to disband.

**Primary Actor:** User

–

**Name:** Plow

**Description:** The user instructs the unit to prepare the land for cultivation.

**Primary Actor:** User

–

**Name:** Assign Trade Route

**Description:** The user directs the unit to establish or modify a trade route.

**Primary Actor:** User

–

**Name:** Wait or Next Unit

**Description:** The user commands the unit to wait for its next turn or for the next unit's action.

**Primary Actor:** User

–

**Name:** Rename

**Description:** The user changes the name of the unit.

**Primary Actor:** User

–

**Name:** Next Unit on Tile

**Description:** The user directs the unit to focus on the next unit present on the same tile.

**Primary Actor:** User

# Use cases for the User Stories:

## 1st user story: Tutorial missions:

Name: report tab

Description: The Player consults the report tab

Primary actor: Player

---

Name: tutorial missions

Description: The Player consults the tutorial missions.

Primary actor: Player

## 2nd user story: Forest Event:

Name: move unit to forest

Description: The Player moves a unit to a forest.

Primary actor: Player

---

Name: gain a move

Description: The Player, after moving a unit to a forest, has a chance to gain a move.

Primary actor: Player

---

Name: gain gold

Description: The Player, after moving a unit to a forest, has a chance to gain a random amount of gold.

Primary actor: Player

---

Name: end turn

Description: The Player, after moving a unit to a forest, has a chance of his turn forcibly end.

Primary actor: Player

---

Name: Nothing happens

Description: The Player, after moving a unit to a forest, has a chance of nothing to happen.

Primary actor: Player