Es Project

Merge Doc

Github Repository:

https://github.com/jmamorim/SE2324_57409_47994_53175_59457_56837_61891

Authors:

João Amorim 57409

João Esteves 47994

Nádia Mendes 53175

José Morgado 59457

Diogo Correia 62475

Miguel Barreto 61891

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.

2º User Story

As a player, I want the ability to deepen my interactions with the native

characters in the game, to enrich the narrative, obtain valuable information and influence the story arc.

3º User Story

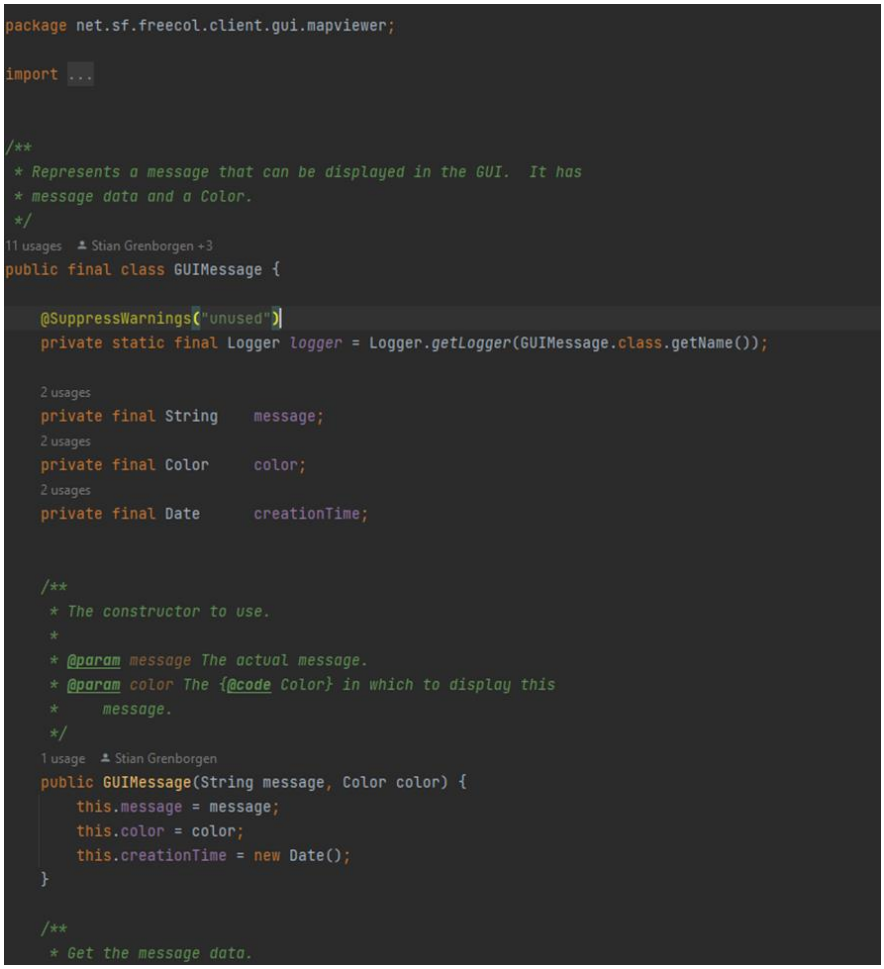
As a new player, I want a set of starting missions to provide me with essential

information and tips, so I can quickly grasp the basic gameplay concepts without feeling overwhelmed.

Code Smells:

João Amorim:

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



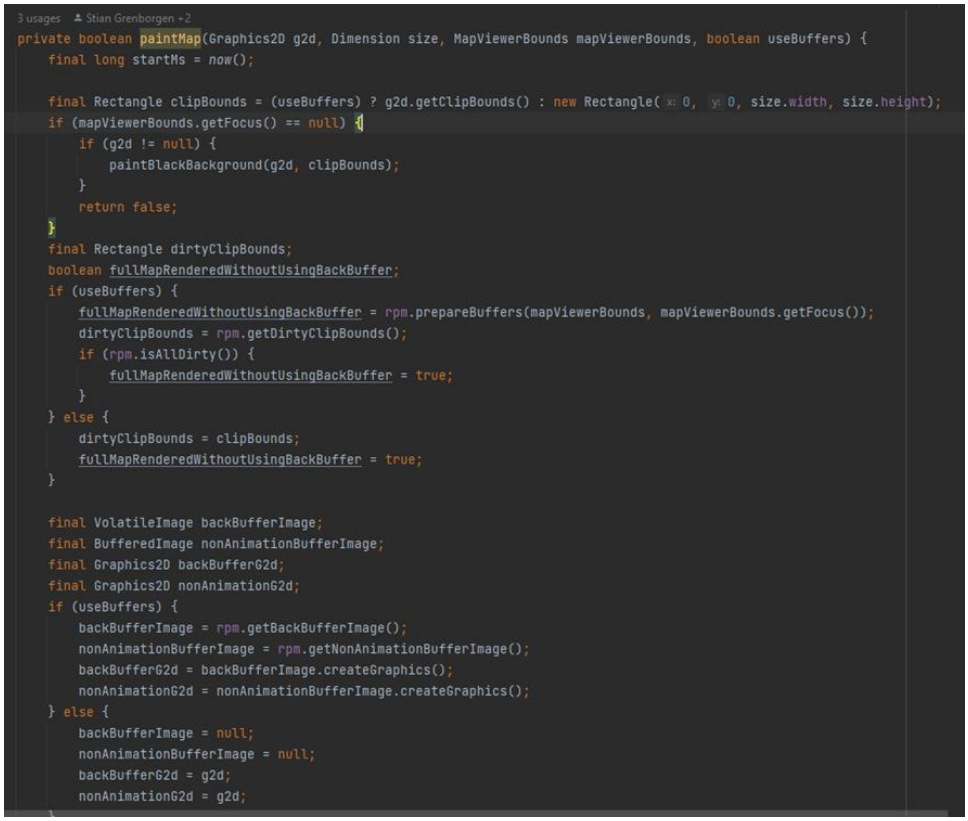
```
package net.sf.freecol.client.gui.mapviewer;

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



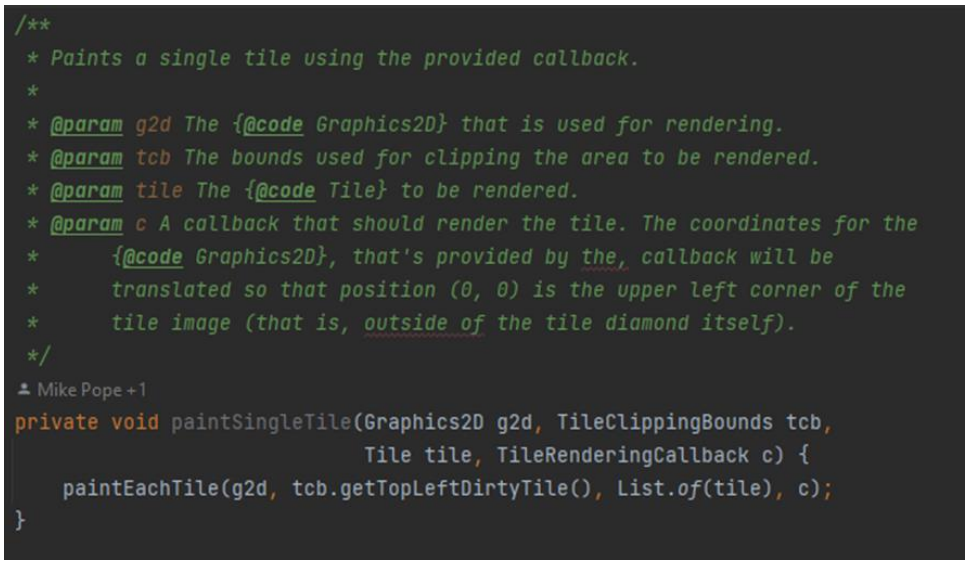
```
3 usages  Stian Grenborgen +2
private boolean paintMap(Graphics2D g2d, Dimension size, MapViewerBounds mapViewerBounds, boolean useBuffers) {
    final long startMs = now();

    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.

//Miguel Barreto

net.sf.freecol.FreeCol:

Long Method:

startClient() and startServer() contain a significant amount of logic.

Long methods are hard to understand, debug, and maintain.

Breaking them down into smaller, focused methods can improve readability and maintainability.

Primitive Obsession:

The use of primitive types (int, boolean, etc.) to represent configuration settings (europeanCount, headless, etc.)

could be replaced with domain-specific classes or enums for better expressiveness and type safety.

Large Class:

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

Duplicated Code:

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

Duplicated code can be a hard to mantain as it requires updates in multiple places if logic changes. My suggestion is to extract the duplicated code and handle the situation with a more general solution.

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

Long Method:

The generateMap method is quite lengthy, performing multiple tasks such as importing tiles, setting regions, creating mountains, rivers, lakes, and bonuses. Long methods can be harder to understand, debug, and maintain.

Breaking down this method into smaller, more focused methods, each responsible for a specific task, would improve readability and make the code easier to manage.

Data Clumps:

There are multiple groups of related parameters used across methods, such as latitude-related parameters.

These data clumps suggest that certain parameters might be better organized into objects or data structures, creating classes or structures to encapsulate related parameters, would make the code more organized and self-explanatory. For example, latitude is passed to methods like getRandomLandTileType and getRandomOceanTileType.

Feature Envy:

Methods like createMountains, createRivers, and createLakeRegions heavily depend on properties and methods of the Map class.

While the logic is encapsulated within these methods, the heavy reliance on external class properties may indicate a form of feature envy.

Feature envy occurs when a method accesses properties or methods of another class more than its own class.

To address this, i would move some of the logic into the Map class itself, promoting encapsulation and better object-oriented design principles.

Methods like createMountains, createRivers, and createLakeRegions heavily use properties and methods of the Map class, such as getGame(), getSpecification(), and getType().

Net.sf.freecol.server.generator.SimpleMap:

Long Method:

The createEuropeanUnits method is quite long and performs multiple tasks, including handling different types of units, selecting starting positions, and checking various conditions.

Long methods can be hard to understand, maintain, and test.

My suggestion is refactoring this method into smaller, more focused methods that handle specific tasks.

Data Clumps:

There are instances of using groups of related data as method parameters, such as generateSkillForLocation taking Map, Tile, and NationType as parameters.

This indicates a data clump, where certain groups of parameters are frequently passed together.

Encapsulating related parameters into a class or structure to improve code readability and maintainability.

Feature Envy:

The createDebugUnits method seems to be more interested in the Player and Unit objects' data than its own class's data.

It may be a sign that the method belongs to a different class or that the responsibilities need to be redistributed.

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*

```java
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

Há evidencia de agrupamentos de dados, na classe *InGameController.java* contida no package *src.net.sf.freecol.client.control,* nesta existem métodos a receber muitos parâmetros, tais como o método *moveToDestination* quase inicia na linha 1196, em que o unit é passado como um parâmetro para quase todas as chamadas de métodos relacionados a unidades, como *followTradeRoute*, *moveTile, moveAttack, etc.* Isto pode ser considerado um data clump uma vez que o objecto unit está sempre relacionado a essas operações de movimento, e os mesmos parâmetros são repetidamente passados.

Uma possível solução seria criar objetos para agrupar dados relacionados e tornar o código mais legível.

```
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:

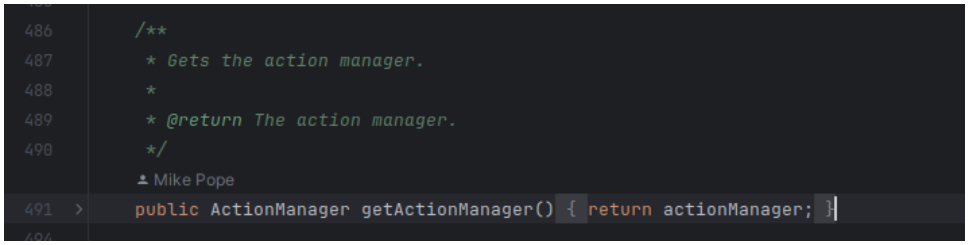
1º - Comments

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

```
631     /**
632      * Is this client logged in to a server?
633      *
634      * @return True if this client is logged in to a server.
635      */
        Mike Pope
636     public synchronized boolean isLoggedIn() { return this.loggedIn; }
639
```

Fig. 2 – ActionManager Method



2º - Magical Numbers (or Strings, in this case)

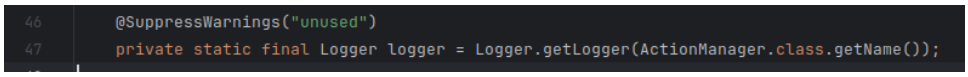
When analyzing the FreeColClient.java class located in the src/net/sf/freecol/client package, it becomes evident that it contains several "magic strings." These are literal strings directly embedded in the source code and are not defined as constants or enums. These magic strings are used in various parts of the code and can make the code less readable, more challenging to maintain, and prone to errors when modification or translation of the software is required. Therefore, it is a good practice to replace these strings with well-documented and named constants or enums to make the code clearer and more maintainable.

Fig. 3 – Some Magical Strings

3º - Speculative Generality

The ActionManager.java class located in the src/net/sf/freecol/client/gui/action package contains an instance of unused code, as indicated by the use of the line with @SuppressWarnings("unused"). This code was added with the intention of potentially logging the actions performed in the future but was never implemented. It is a good practice to remove unused variables to keep the code clean and easy to maintain.

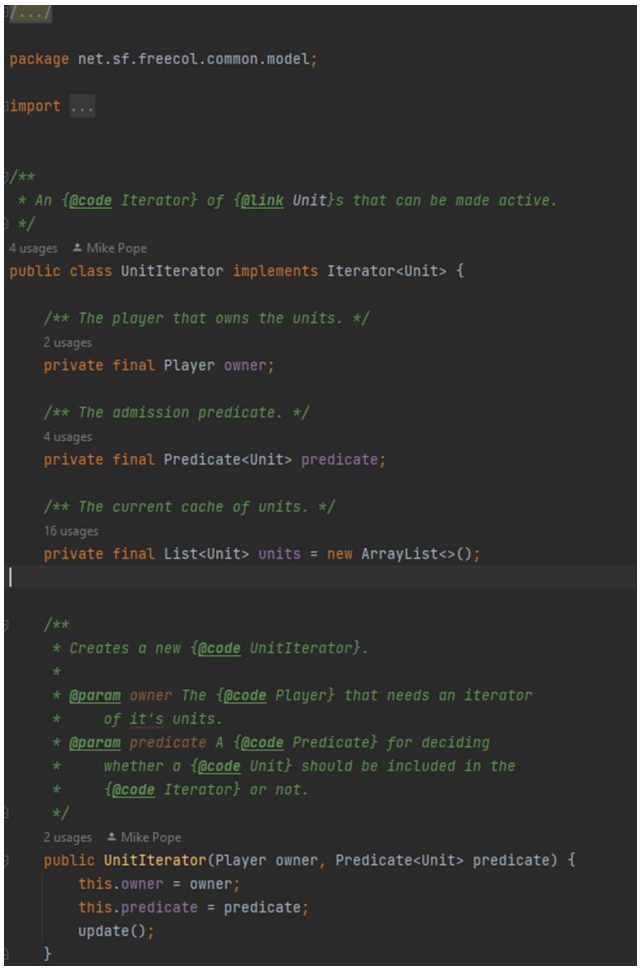
Fig.4 - Useless Logger



Gof Patterns:

João Amorim:

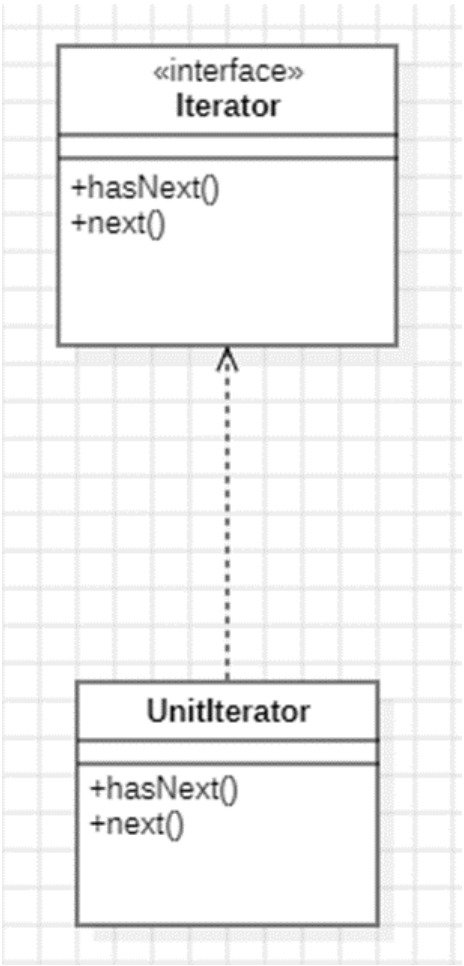
1 – Iterator - net.sf.freecol.common.model.UnitIterator



```
/.../

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 –

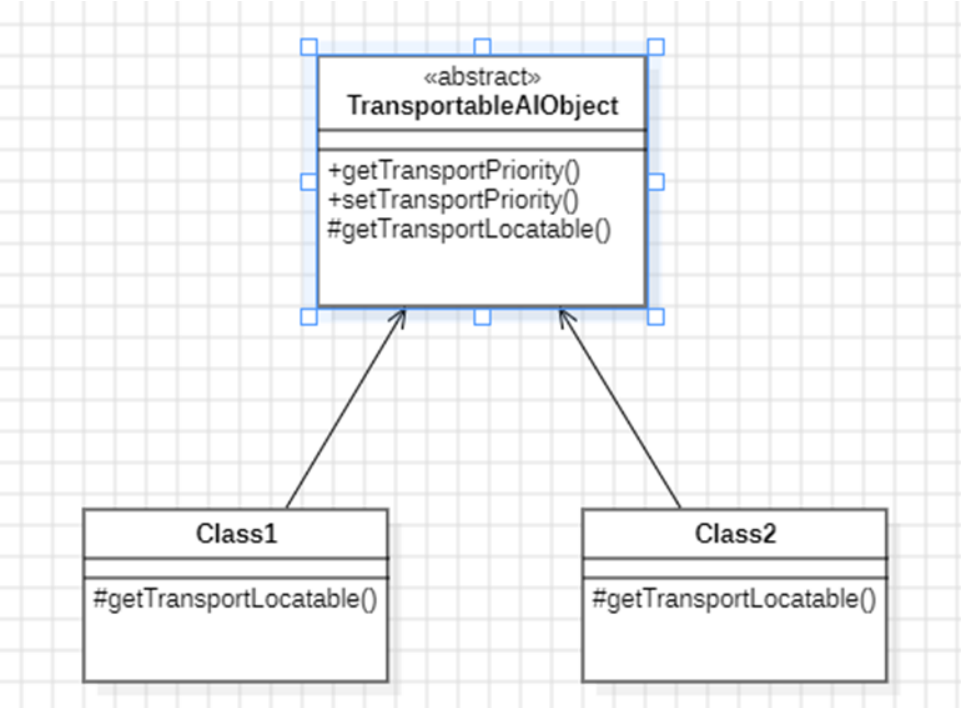


2 – Template - net.sf.freecol.server.ai.TrasportableAIObject

```
/**
 * 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 –

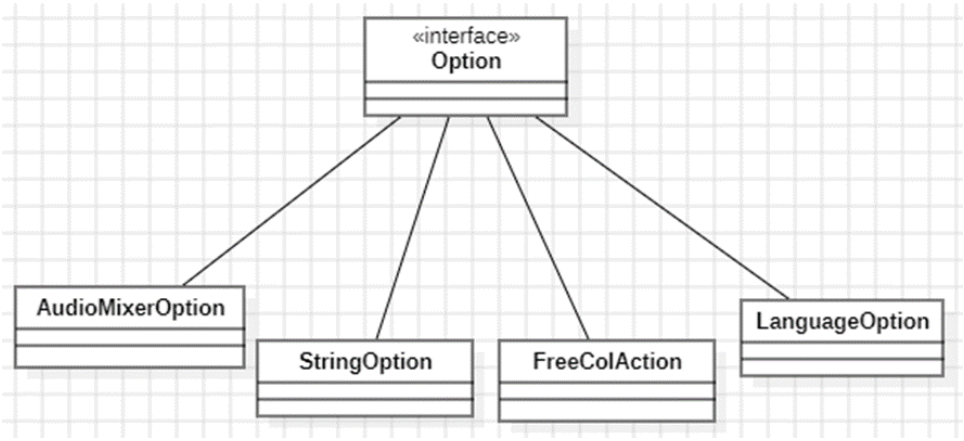


3 – Facade - net.sf.freecol.client.gui.action.Option

```
/**
 * 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:**

The *followTradeRoute* method appears to present the template method pattern, as it defines a skeleton of the algorithm with concrete steps defined in concrete methods, such as *unloadUnitAtStop*, *loadUnitAtStop*, among others. It can thus be seen that the method itself is more concerned with the attribution of responsibilities. The subclasses (or in this case the private methods) extend and override these steps as needed.

This method starts on line 2145 of the InGameController.java class contained in the *src.net.sf.freecol.client.control* package and ends on line 2283, as it is an extensive method the complete representation of this was divided from figure 3 to figure 5.

```
/**
 * Follows a trade route, doing load/unload actions, moving the unit,
 * and updating the stop and destination.
 *
 * @param unit The {@code Unit} on the route.
 * @param messages An optional list in which to retain any
 *     {@code ModelMessage}s generated.
 * @return True if automatic movement can proceed.
 */
private boolean followTradeRoute(Unit unit, List<ModelMessage> messages) {
```

Fig.1 start of the *followTradeRoute* method.

```
// At the stop, do the work available.
lb.mark();
unloadUnitAtStop(unit, lb); // Anything to unload?
loadUnitAtStop(unit, lb); // Anything to load?
lb.grew("\n", Messages.message(stop.getLabelFor("tradeRoute.atStop",
                                                player)));
```

Fig.2 part of the followTradeRoute method code where the call to the private unloadUnitAtStop and loadUnitAtStop methods takes place.

```
private boolean followTradeRoute(Unit unit, List<ModelMessage> messages) {
    final Player player = unit.getOwner();
    final TradeRoute tr = unit.getTradeRoute();
    final boolean detailed = getClientOptions()
        .getBoolean(ClientOptions.SHOW_GOODS_MOVEMENT);
    final boolean checkProduction = getClientOptions()
        .getBoolean(ClientOptions.STOCK_ACCOUNTS_FOR_PRODUCTION);
    final List<TradeRouteStop> stops = unit.getCurrentStops();
    boolean result = true;

    // If required, accumulate a summary of all the activity of
    // this unit on its trade route.
    LogBuilder lb = new LogBuilder((detailed && !tr.isSilent()) ? 256
        : -1);
    lb.mark();

    // Validate the whole route.
    boolean valid = true;
    for (TradeRouteStop trs : stops) {
        if (!TradeRoute.isStopValid(unit, trs)) {
            lb.add(" ", Messages.message(trs.invalidStopLabel(player)));
            valid = false;
        }
    }
    if (!valid) {
        clearOrders(unit);
        stops.clear();
        result = false;
```

Fig.3 Representation of the initial part of the *followTradeRoute* method

```
}

// Try to find work to do on the current list of stops.
while (!stops.isEmpty()) {
    TradeRouteStop stop = stops.remove(0);

    if (!unit.atStop(stop)) {
        // Not at stop, give up if no moves left or the path was
        // exhausted on a previous round.
        if (unit.getMovesLeft() <= 0
            || unit.getState() == UnitState.SKIPPED) {
            lb.add(" ", Messages.message(stop
                    .getLabelFor("tradeRoute.toStop", player)));
            break;
        }

        // Find a path to the stop, skip if none.
        Location destination = stop.getLocation();
        PathNode path = unit.findPath(destination);
        if (path == null) {
            lb.add("\n", Messages.message(stop
                    .getLabelFor("tradeRoute.pathStop", player)));
            changeState(unit, UnitState.SKIPPED);
            break;
        }
```

Fig4. Continued representation of the *followTradeRoute* method.

```
        // Try to follow the path.  If the unit does not reach
        // the stop it is finished for now.
        movePath(unit, path);
        if (!unit.atStop(stop)) {
            changeState(unit, UnitState.SKIPPED);
            break;
        }
    }

    // At the stop, do the work available.
    lb.mark();
    unloadUnitAtStop(unit, lb); // Anything to unload?
    loadUnitAtStop(unit, lb); // Anything to load?
    lb.grew("\n", Messages.message(stop.getLabelFor("tradeRoute.atStop",
                                                    player)));

    // If the un/load consumed the moves, break now before
    // updating the stop.  This allows next turn to retry
    // un/loading, but this time it will not consume the moves.
    if (unit.getMovesLeft() <= 0) break;

    // Find the next stop with work to do.
    TradeRouteStop next = null;
    List<TradeRouteStop> moreStops = unit.getCurrentStops();
    if (unit.atStop(moreStops.get(0))) moreStops.remove(0);
    for (TradeRouteStop trs : moreStops) {
        if (trs.hasWork(unit, (!checkProduction) ? 0
                        : unit.getTurnsToReach(trs.getLocation()))) {
```

Fig5. Continued representation of the followTradeRoute method.

```
            next = trs;
            break;
        }
    }
    if (next == null) {
        // No work was found anywhere on the trade route,
        // so we should skip this unit.
        lb.add(" ", Messages.message("tradeRoute.wait"));
        changeState(unit, UnitState.SKIPPED);
        unit.setMovesLeft(0);
        break;
    }
    // Add a message for any skipped stops.
    List<TradeRouteStop> skipped
        = tr.getStopSublist(stops.get(0), next);
    if (!skipped.isEmpty()) {
        StringTemplate t = StringTemplate.label("")
            .add("tradeRoute.skipped");
        String sep = " ";
        for (TradeRouteStop trs : skipped) {
            t.addName(sep)
                .addStringTemplate(trs.getLocation()
                    .getLocationLabelFor(player));
            sep = ", ";
        }
        t.addName(".");
        lb.add(" ", Messages.message(t));
```

Fig6. Continued representation of the followTradeRoute method.

1. command Pattern:

Os métodos como *moveTile*, *moveScountColony, moveSpymoveTrade, moveTribute*, *moveUseMissionary* contidos na classe *InGameController.java* do package *src.net.sf.freecol.client.control* representam a encapsulação de uma solicitação como um objeto, o que se assemelha ao command Pattern, uma vez que estes encapsulam uma unidade de ação e os parâmetros necessários para realizar essa ação.

Fig8. Método *moveScountColony.*

2. Abstract Factory Pattern:

Na classe *LanguageOptionUI,* contida no package *src.net.sf.freecol.client.gui.option* verifica-se que esta atua como uma fábrica abstrata para criar objetos relacionados à opção de idioma(LanguageOption)

A classe *LanguageOptionUI* cria e retorna uma instância de JComboBox <Language> que é uma parte da família de objetos de interface do usuário relacionados à opção de idioma.

A classe *LanguageOption* representa a opção de idioma, enquanto a classe Language representa os idiomas disponíveis.

Portanto, verifica-se o uso do *Abstract Factory Pattern,* este é usado para criar objetos relacionados de acordo com a escolha do idioma, e isso permite criar uma família de objetos coerentes relacionados à opção do idioma.

Fig.9 Representação da parte inicial da classe *LangugeOptionUI*.

Fig.10 Representação do código restante da classe *LangugeOptionUI*.

```
        }
        // Bring the next stop to the head of the stops list if it
        // is present.
        while (!stops.isEmpty() && stops.get(0) != next) {
            stops.remove(0);
        }
        // Set the new stop, skip on error.
        if (!askServer().setCurrentStop(unit, tr.getIndex(next))) {
            changeState(unit, UnitState.SKIPPED);
            break;
        }
    }
    if (lb.grew()) {
        ModelMessage m = new ModelMessage(MessageType.GOODS_MOVEMENT,
                                          "tradeRoute.prefix", unit)
            .addName("%route%", tr.getName())
            .addStringTemplate("%unit%",
                unit.getLabel(Unit.UnitLabelType.NATIONAL))
            .addName("%data%", lb.toString());
        if (messages != null) {
            messages.add(m);
        } else {
            player.addModelMessage(m);
            turnReportMessages.add(m);
        }
    }
    return result;
}
```

Fig7. Finalization of the representation of the followTradeRoute method.

1. command Pattern:

Os métodos como *moveTile*, *moveScountColony, moveSpymoveTrade, moveTribute*, *moveUseMissionary* contidos na classe *InGameController.java* do package *src.net.sf.freecol.client.control* representam a encapsulação de uma solicitação como um objeto, o que se assemelha ao command Pattern, uma vez que estes encapsulam uma unidade de ação e os parâmetros necessários para realizar essa ação.

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.

```java
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));
```

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 -

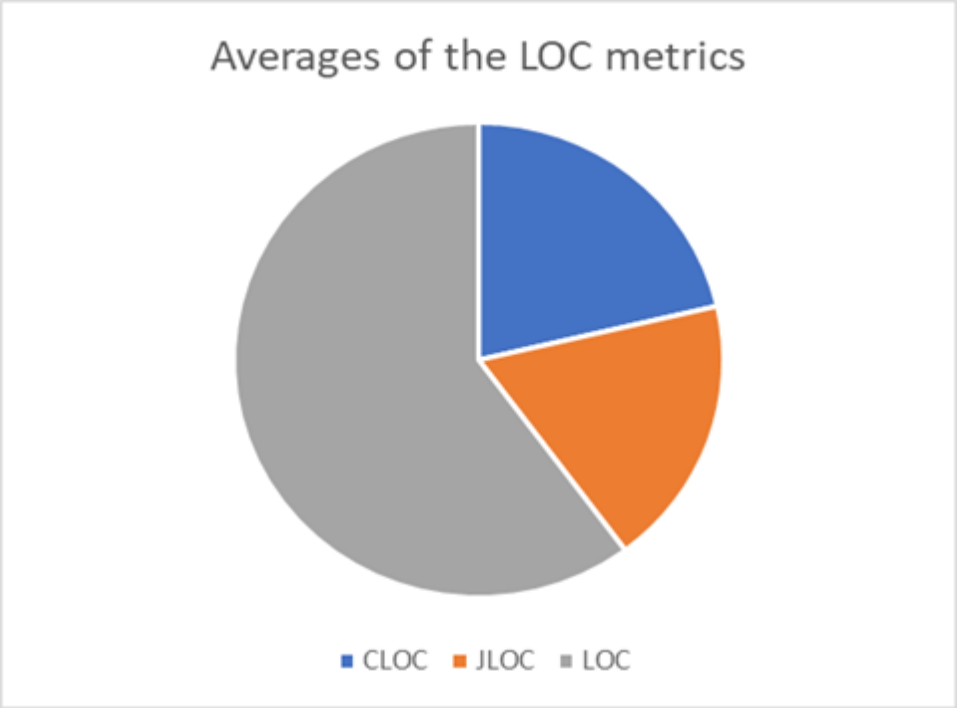
class	CLOC	JLOC	LOC
net.sf.freecol.common.model.Unit	1962.0	1766.0	4263.0
net.sf.freecol.common.model.Player	1921.0	1768.0	3892.0
net.sf.freecol.client.control.InGameController	1638.0	1297.0	4806.0
net.sf.freecol.client.gui.GUI	1457.0	1420.0	2220.0
net.sf.freecol.common.util.CollectionUtils	1445.0	1443.0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



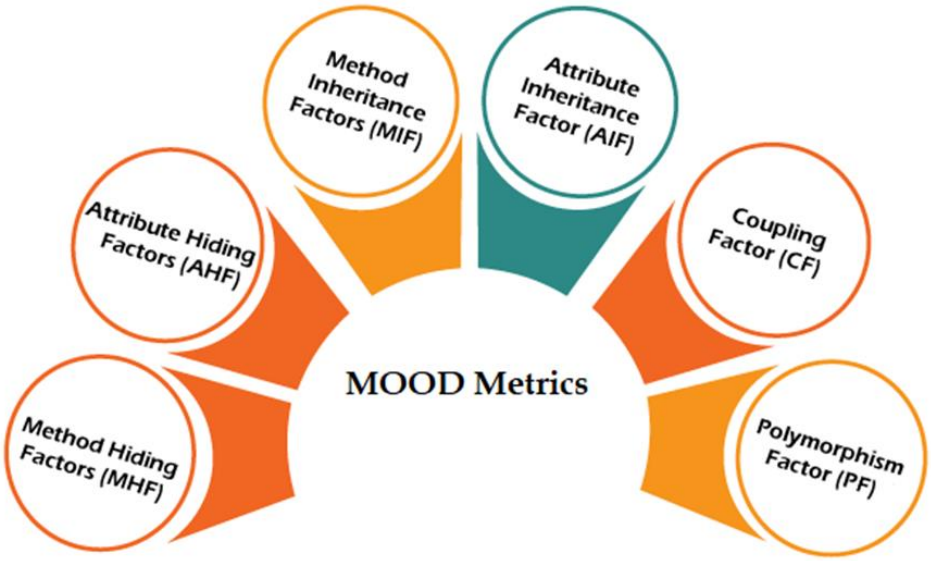
Discussion-

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

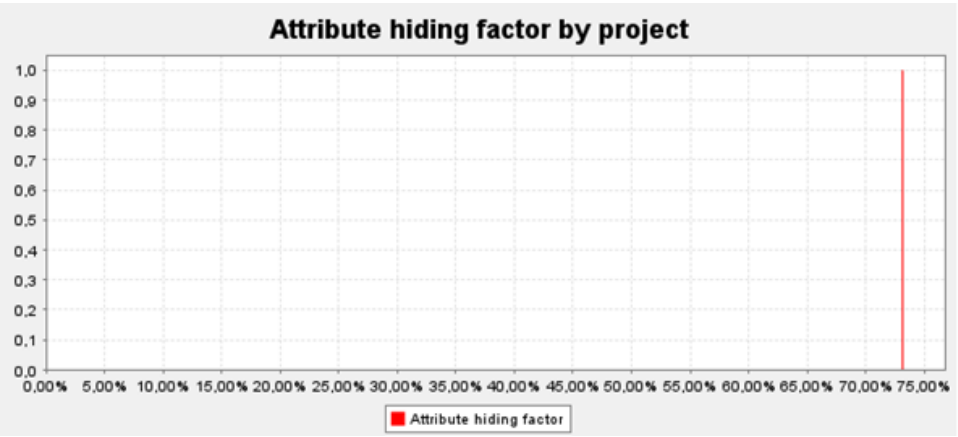
$A_d(C_i)$ = $A_v(C_i) + A_h(C_i)$: Attributes defined in C

$A_v(C_i)$: 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

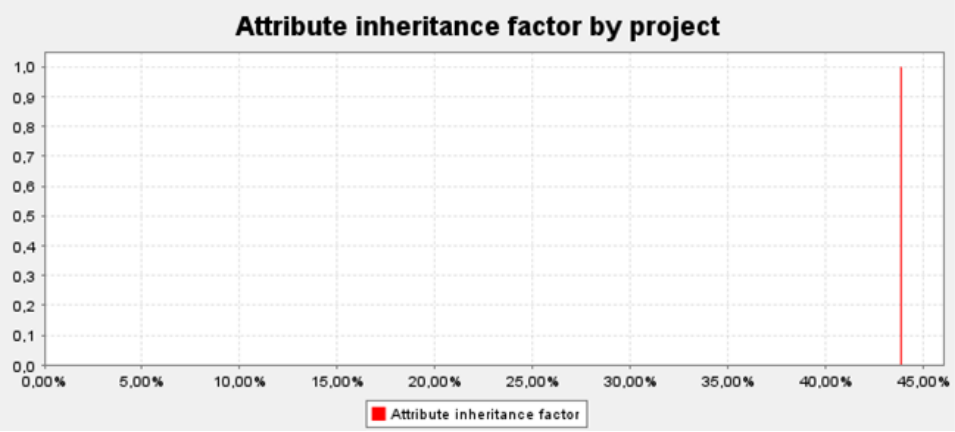
$A_a(C_i)$ = $A_d(C_i) + A_h(C_i)$: Attributes defined in C_i

$A_d(Ci)$: 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:

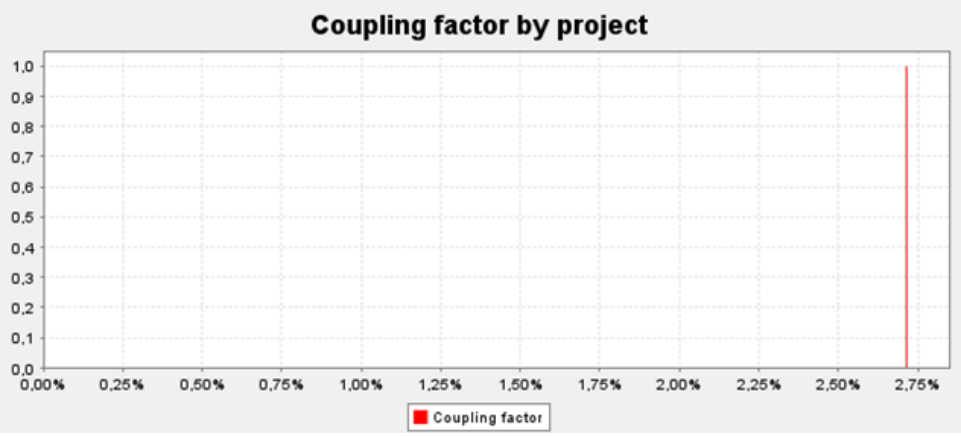
$$COF = \frac{\sum_{i=1}^{TC}\left[\sum_{j=1}^{TC} is\_client\ (C_i, C_j)\right]}{TC^2 - TC}$$

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

$M_v(C_i)$: 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

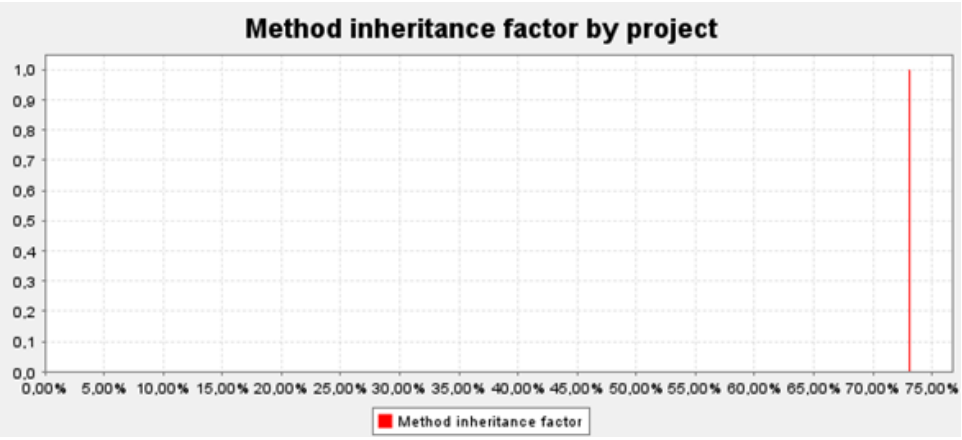
$M_a(C_i)$ = $M_d(C_i)$ + $M_i(C_i)$: Methods defined in C_i

$M_d(Ci)$: 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

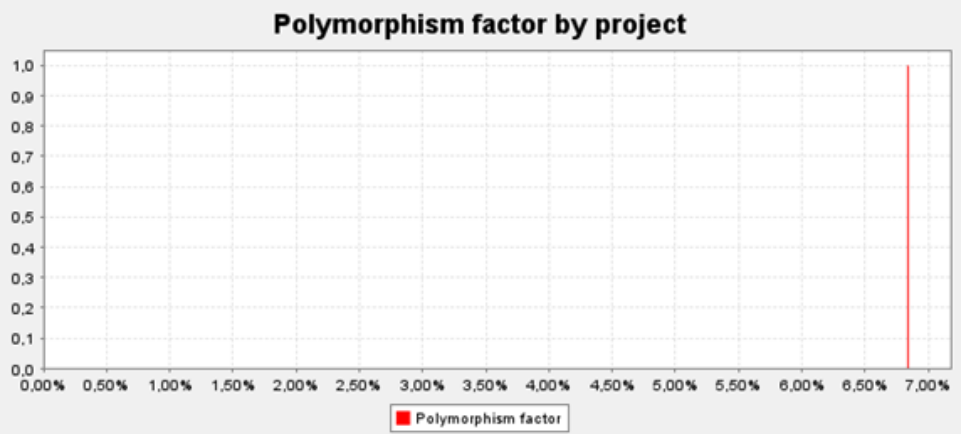
$M_n(Ci)$: New Methods in C_i

$D_C(C_i)$: 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.

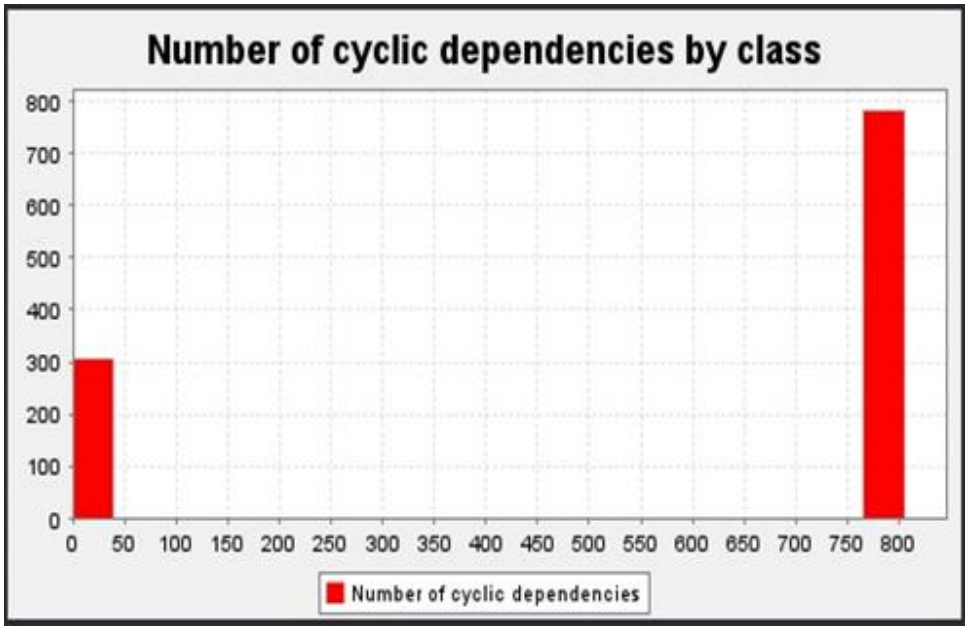
Data Visualization –

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

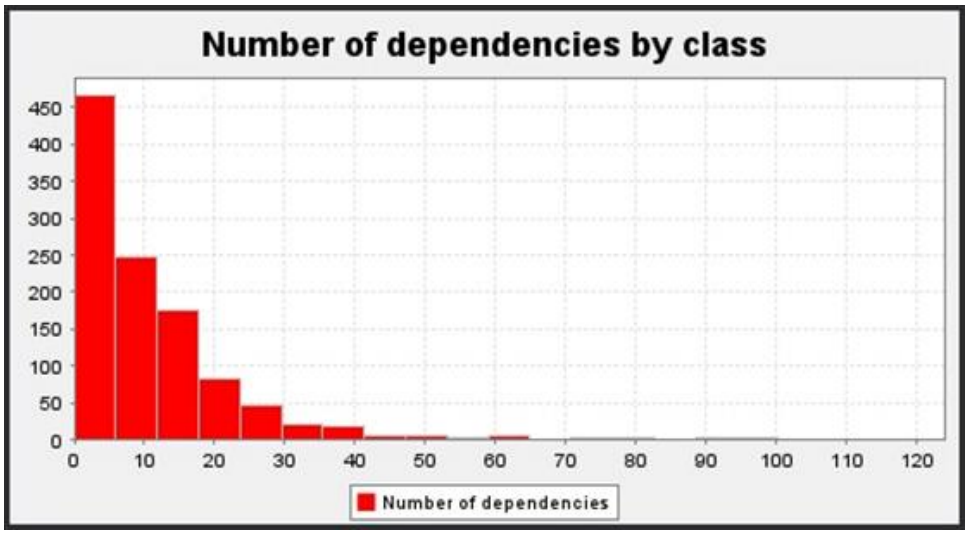
Top 5 CYCLIC –	CYCLIC	DCY	DCY*	DPT	DPT*	PDCY	PDPT
net.sf.freecol.server.networking.Server	805	63	989	61	923	6	12
net.sf.freecol.server.networking.DummyConnection	805	3	989	10	923	2	3

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 –

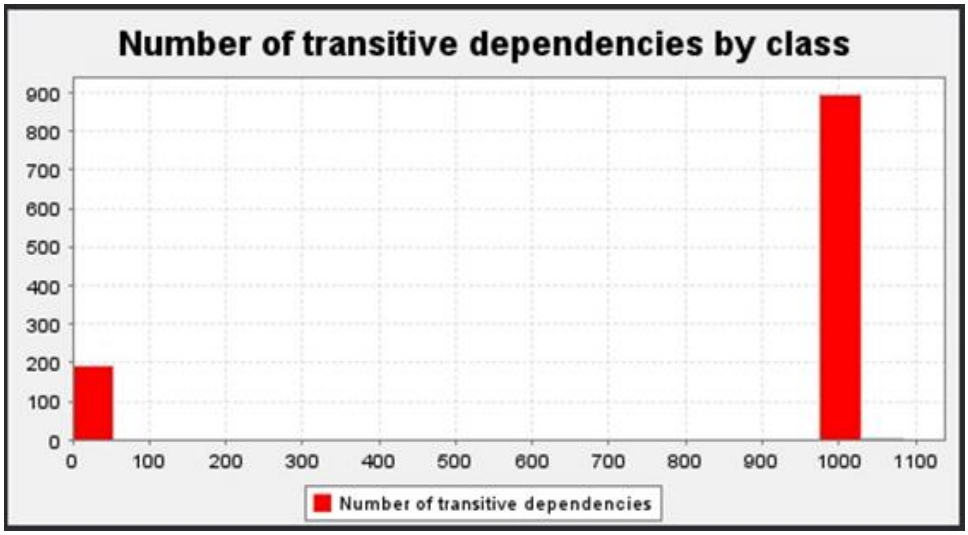
	CYCLIC	DCY	DCY*	DPT	DPT*	PDCY	PDPT
net.sf.freecol.server.control.InGameController	805	118	989	83	923	11	1
net.sf.freecol.client.gui.Widgets	805	111	989	2	923	10	12
net.sf.freecol.common.networking.ServerAPI	805	104	989	10	923	3	5
net.sf.freecol.server.model.ServerPlayer	805	96	989	124	923	9	1
net.sf.freecol.client.gui.SwingGUI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



Top 5 DCY* –

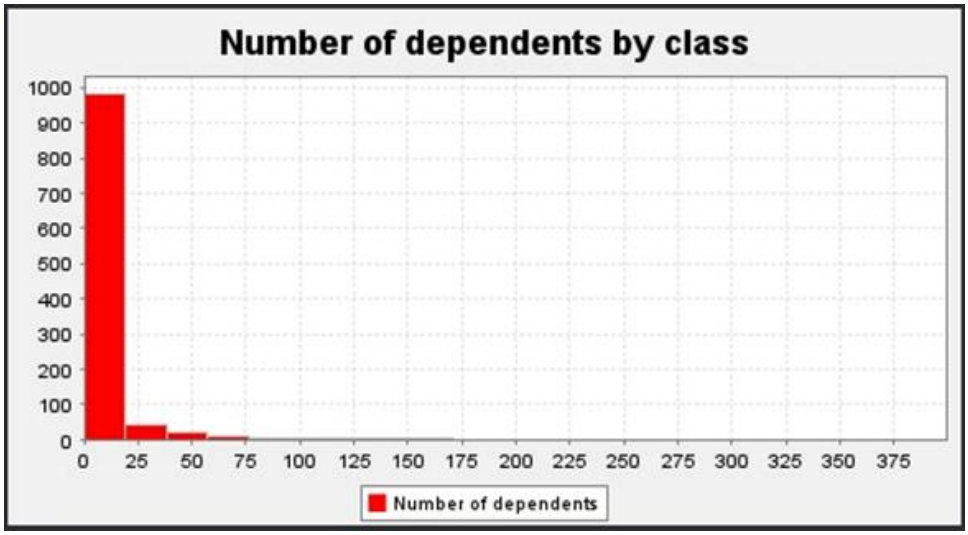
	CYCLIC	DCY	DCY*	DPT	DPT*	PDCY	PDPT

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



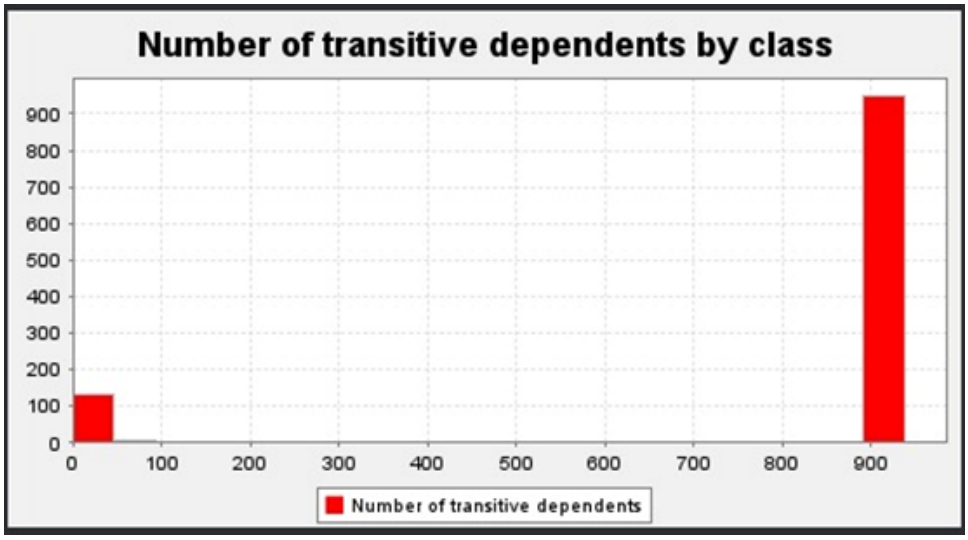
Top 5 DPT –

	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



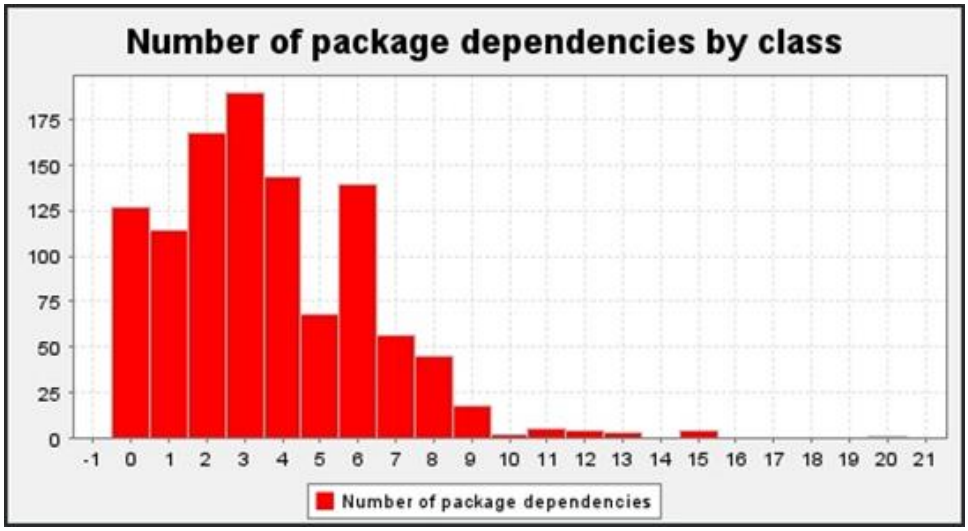
Top 5 DPT* –

	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

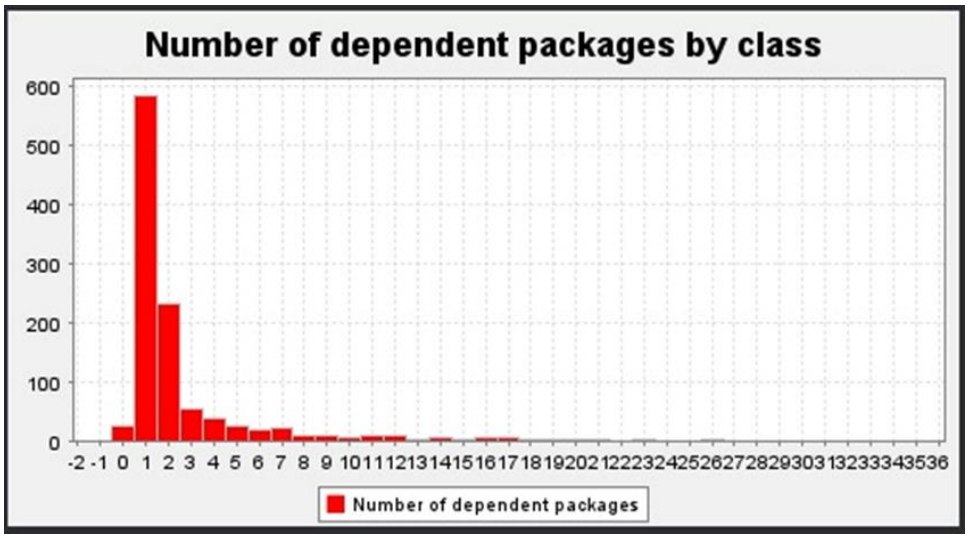


Top 5 PDCY –

	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.

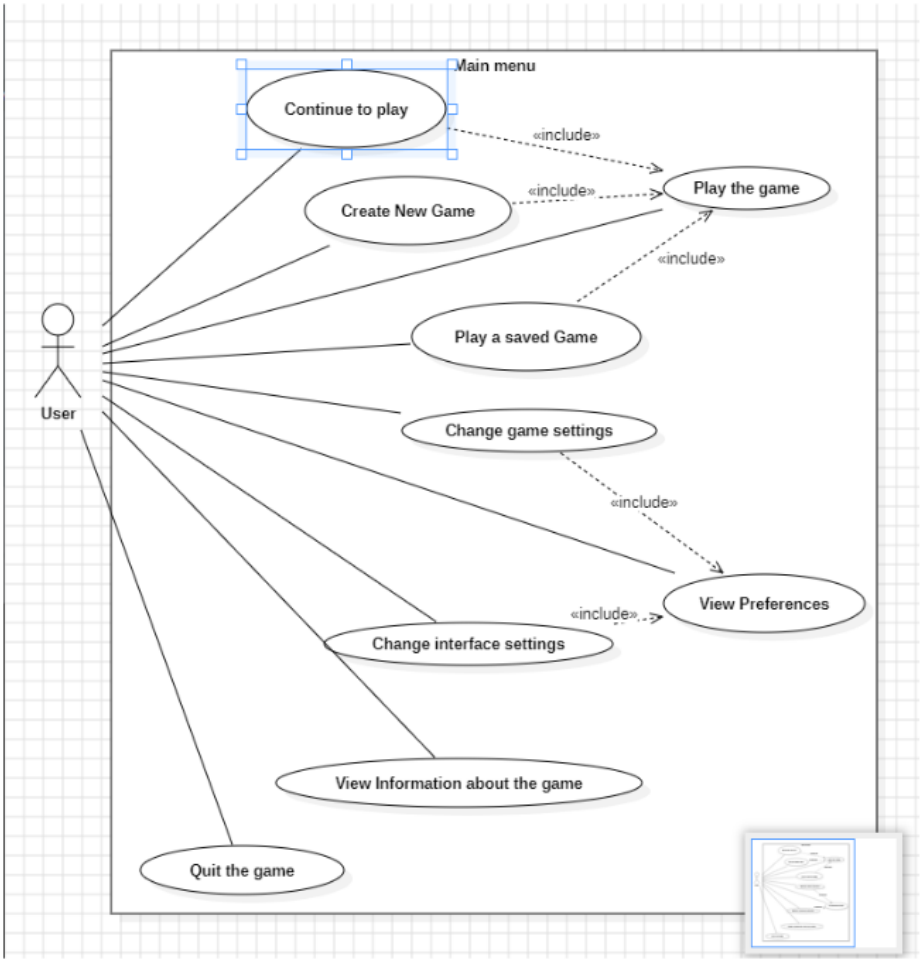
Regarding the number of dependents (DPT), with an average of 10.48, we can see that in terms of direct dependencies, the project has a moderate level of coupling.

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.

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

--

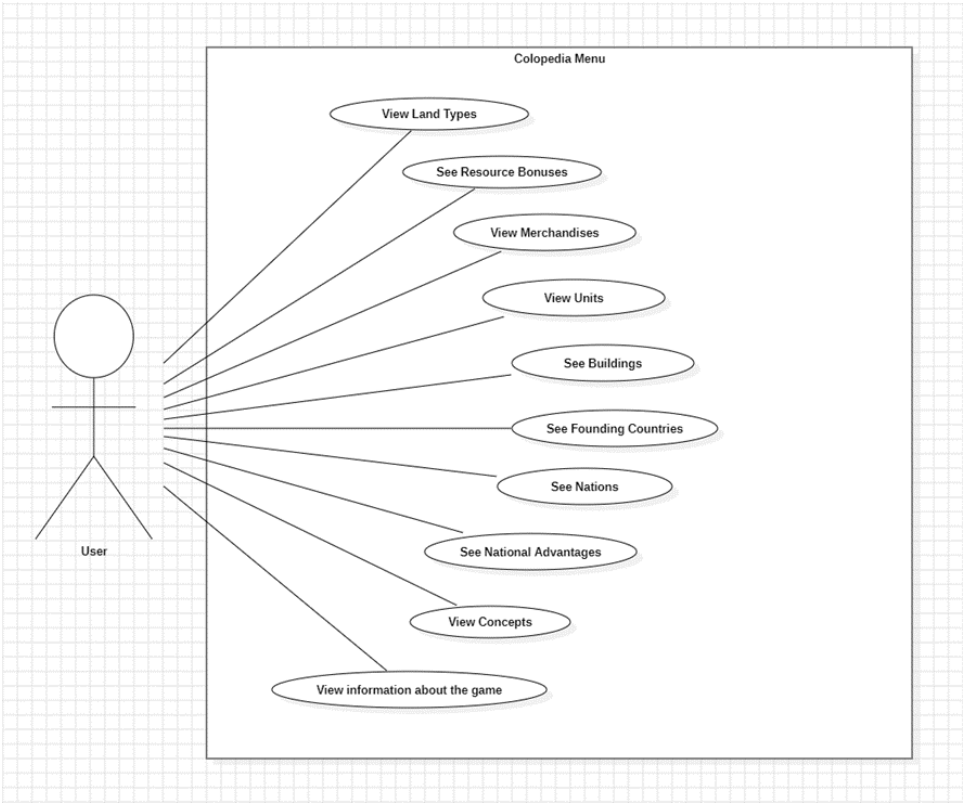
Name: Quit the game

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

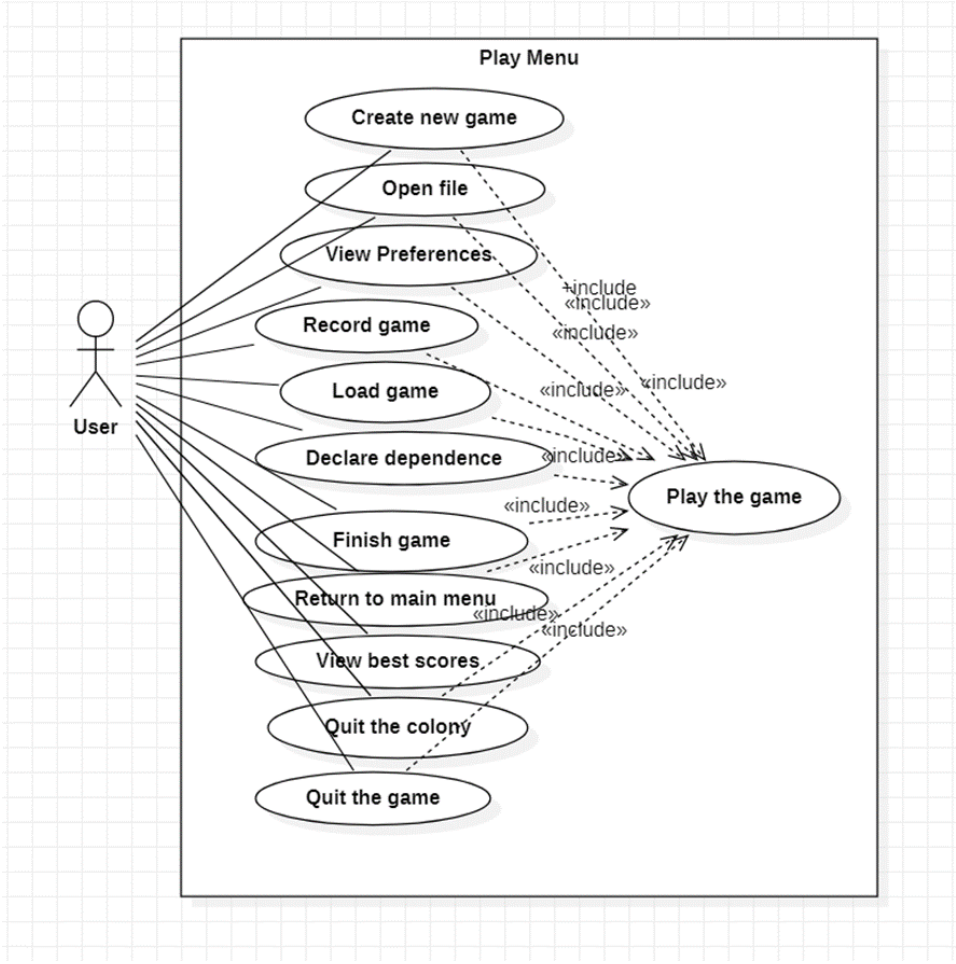
--

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:

The act of playing freecol.

Primary actor: User

--