FreeCol Project Management
Full Report

Group composed of:

(Student Name, Student Number)

André Santos, 62331

António Palmeirim, 63667

Duarte Inácio, 62397

Luís Serrano, 60253

Rafael Tavares, 60608

Tiago Santos, 63390

Work carried out as part of:

Engenharia de Software

(Software Engineering)

FCT-NOVA

Index

Return to Index.

Report by student:

André Santos, 62331: Code Smells, Code Metrics, Design Patterns, Use Case.

António Palmeirim, 63667: Code Smells, Code Metrics, Design Patterns, Use Case.

Duarte Inácio, 62397: Code Smells, Code Metrics, Design Patterns, Use Case.

Luís Serrano, 60253: Code Smells, Code Metrics, Design Patterns, Use Case.

Rafael Tavares, 60608: Code Smells, Code Metrics, Design Patterns, Use Case.

Tiago Santos, 63390: Code Smells, Code Metrics, Design Patterns, Use Case.

Note: All information related to the author of a segment of this report will be mentioned in *italic*. Each page also contains a quick return button to the Index page, for facilitated navigation!

Return to Index.

User Stories

(Duarte Inácio 62397)

1. As a FreeCol player, I want the game to offer better security measures at the IP layer when connecting to a server, to ensure the protection of my personal information and provide a safer gaming experience.

Description User Story 1: To take part in a multiplayer game, players have to connect directly to other players' IP addresses, which poses a significant security risk, as IPs can be significant security risk, as IPs can be collected and exploited by malicious people to collect personal information and potentially attack

Extra information: We consider this User Story to be quite complex, so we decided to have 4 US in case we couldn't implement this US with the tools and knowledge we have.

(Tiago Santos 63390)

2. As a FreeCol player, I want camera and unit movements (troops, settlers, ...) easier to use and differentiate, to move easily and not have any accidental movements.

Description User Story 2: The movement of the player's view camera is difficult to differentiate from the movement of the units, so unit movements occur accidentally. The player view could be done by dragging the mouse.

(Tiago Santos 63390)

3. As a FreeCol player, I want to center the map directly on the units I have available, so I don't have to manually search for them on the map.

Description User Story 3: After browsing the map, you have to manually search for the unit you want to use. It should be possible to go directly to the units you want at the click of a button (instead of having to search the map).

Return to Index.

(Rafael Tavares 60608)

4. As a competitive player, I want a leaderboard that tracks the players' scores/statistics.

Description User Story 4: When playing, I find it useful to have a leaderboard that tracks the scores/statistics of each participant in both single player and multiplayer games, allowing me to compare my performance with that of other players.

Video Demonstration

The following link is a video demonstration of all FreeCol implementations that were done: *https://www.youtube.com/watch?v=qx76yd-Nw74 (posted as unlisted on YouTube).*

Return to Index.

Implementation Adaptation

Information related to any adaptation to the implementation of the user stories made due to certain User Stories being complex and time consuming to be implemented. Each adapted implementation also has a use case associated, with respective name, description and actors.

User Story 1: Not implemented as explained previously. Our team opted to dedicate our time to implementation of User Stories 2,3 and 4.

Use Case: Not implemented.

(António Palmeirim 63667, Duarte Inácio 62397)

User Story 2: As implementing mouse feedback in-game was very time consuming and complex, we opted to create a button that will re-center and modify the zoom to the default, in-case of various zooming out and in actions and moving around the map, it is easier to refocus on the unit currently in control.

Use Case:
Name: Full Re-center
Description: Involves player pressing a keyboard key or a button on the View area of the navigation bar that resets the zoom and re-centers to the main unit.
Actors: Player

Return to Index.

(Tiago Santos 63390, André Santos 62331)

User Story 3: No adaptations were made to this user story.

Use Case:
Name: Locate Units
Description: Involves player pressing an available unit from the units list on the canvas and the map automatically centers the screen view on the selected unit (so that the player doesn´t need to manually drag through the map).
Actors: Player

(Luís Serrano 60253, Rafael Tavares 60608)

User Story 4: When implementing it, we realized that obtaining the information of all the players present in the game was very difficult and we couldn't find a way to do it, so we ended up implementing the statistics only for the current player.

Use Case:
Name: Game Statistics
Description: The player can open a panel where he can see a list of his resources, gold and score.
Actors: Player

Code Review

The following code review will be divided by code smells, code metrics, design patterns and use case diagrams. Each type of code review will be divided independently by member, i.e., for Code Smells, member António Palmeirim will have a dedicated part in this report with the code smells found and discussed. The same structure will be used for all other parts of the team's code review.

At the end of each report, there will be a Code Report Review Log, with all the reviews submitted regarding the respective report, with the information of the reviewer, the date and the review itself.

Return to Index.

Code Smells

Author: André Santos(62331)

Code Smell 1:

Large Class

Path: net/sf/freecol/client/gui/dialog/Flag.java

```
45  public class Flag {
46
        The alignment of stripes or other design elements on the flag. Might be extended to handle diagonal alignment
        (Friesland, for example).
52      public enum Alignment {
53          NONE,
54          HORIZONTAL,
55          VERTICAL
56      };
57
        The "background layer" of the flag, generally one or several squares or triangles of different color. The
        alignment of the background influences the size of the canton, if there is one.
63      public enum Background {
64              /** A plain background. */
65          PLAIN(Alignment.NONE),
66              /** Quartered rectangularly. */
67          QUARTERLY(Alignment.NONE),
68              /** Vertical stripes. */
69          PALES(Alignment.VERTICAL),
70              /** Horizontal stripes. */
71          FESSES(Alignment.HORIZONTAL),
72              /** Diagonal top left to bottom right. */
73          PER_BEND(Alignment.NONE),
74              /** Diagonal bottom left to top right. */
75          PER_BEND_SINISTER(Alignment.NONE),
76              /** Quartered diagonally. */
77          PER_SALTIRE(Alignment.NONE);
78
79          public final Alignment alignment;
```

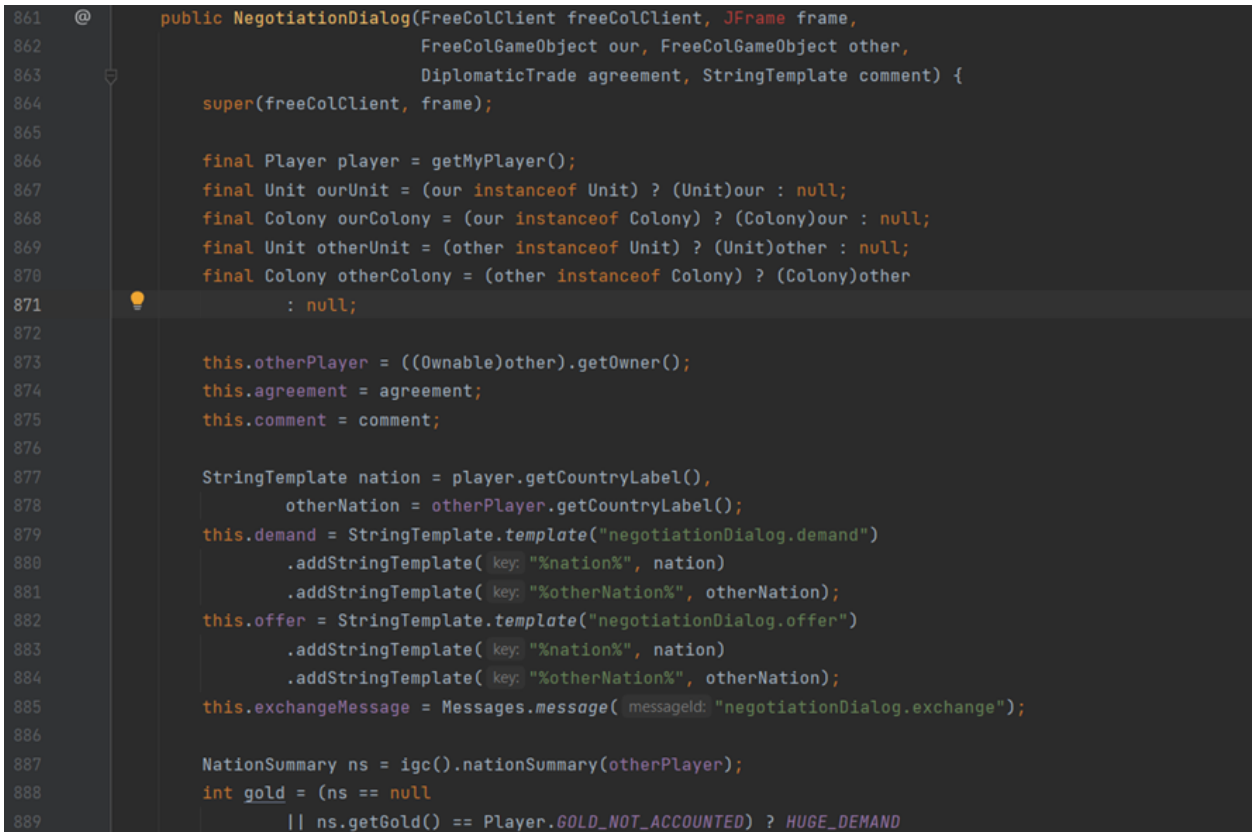
Here we can see the "Flag" class with more than 1200 lines of code, which is an example of the "large class code smell," a type of code problem that falls into the group of "code smells" called "bloaters." This problem occurs when a single class takes on a large number of responsibilities. To try to solve this problem, we can subdivide the Flag class into smaller classes, each in charge of different functionalities that are all present in the same class. For example, we could separate the calculation of the different figures needed to draw a "flag" into different classes associated with the Flag class.

Return to Index.

Code Smell 2:

Long Method

Path: net/sf/freecol/client/gui/dialog/NegotiationDialog.java



```
861  @   public NegotiationDialog(FreeColClient freeColClient, JFrame frame,
862                               FreeColGameObject our, FreeColGameObject other,
863                               DiplomaticTrade agreement, StringTemplate comment) {
864          super(freeColClient, frame);
865
866          final Player player = getMyPlayer();
867          final Unit ourUnit = (our instanceof Unit) ? (Unit)our : null;
868          final Colony ourColony = (our instanceof Colony) ? (Colony)our : null;
869          final Unit otherUnit = (other instanceof Unit) ? (Unit)other : null;
870          final Colony otherColony = (other instanceof Colony) ? (Colony)other
871                  : null;
872
873          this.otherPlayer = ((Ownable)other).getOwner();
874          this.agreement = agreement;
875          this.comment = comment;
876
877          StringTemplate nation = player.getCountryLabel(),
878                  otherNation = otherPlayer.getCountryLabel();
879          this.demand = StringTemplate.template("negotiationDialog.demand")
880                  .addStringTemplate( key: "%nation%", nation)
881                  .addStringTemplate( key: "%otherNation%", otherNation);
882          this.offer = StringTemplate.template("negotiationDialog.offer")
883                  .addStringTemplate( key: "%nation%", nation)
884                  .addStringTemplate( key: "%otherNation%", otherNation);
885          this.exchangeMessage = Messages.message( messageId: "negotiationDialog.exchange");
886
887          NationSummary ns = igc().nationSummary(otherPlayer);
888          int gold = (ns == null
889                  || ns.getGold() == Player.GOLD_NOT_ACCOUNTED) ? HUGE_DEMAND
```

This method with almost two hundred lines is an obvious example of the "code smell" called "Long Method." This code smell points to the excessive length and complexity of a method, making it difficult to understand, modify and debug. The solution involves refactoring the code, i.e. dividing this method into smaller methods, each with a specific function and a descriptive name.

Return to Index.

Code Smell 3:

Message Chain

Path: net/sf/freecol/client/gui/dialog/VictoryDialog.java

Line 51-53: panel.add(Utility.localizedHeader(Messages.message("victory.text"), Utility.FONTSPEC_TITLE), "align center, wrap 20");

As we can see in the code excerpt above, we find a code smell from the "Couplers" category called "Message Chain". This code smell occurs when there are several chained method calls in a single line of code. This message chain indicates a strong dependency between objects, making the code less readable and fragile. One way to solve this "code smell" would be to store the intermediate result in local variables, thus dividing the calls into several lines and avoiding this chaining of methods on a single line.

Return to Index.

Code Report Review Log

Reviewer: Luís Serrano 60253

Review 1 (4/11):
Concordo com todos os code smells apresentados, apenas acho que a print no primeiro smell poderia ter sido mais explícita, colapsando alguns métodos para verificarmos melhor a extensão do código. No entanto, o path em questão é-nos fornecido, portanto foi fácil verificar a veracidade!

Reviewer: Tiago Santos 63390

Review 1 (5/11):
Estou de acordo com todas as indicações de problemas de código mencionadas. No entanto, também penso que a descrição do primeiro problema poderia ter sido mais detalhada.

Review 2 (12/11):
Agora estando com mais detalhe em todos, não tenho nada a apontar!

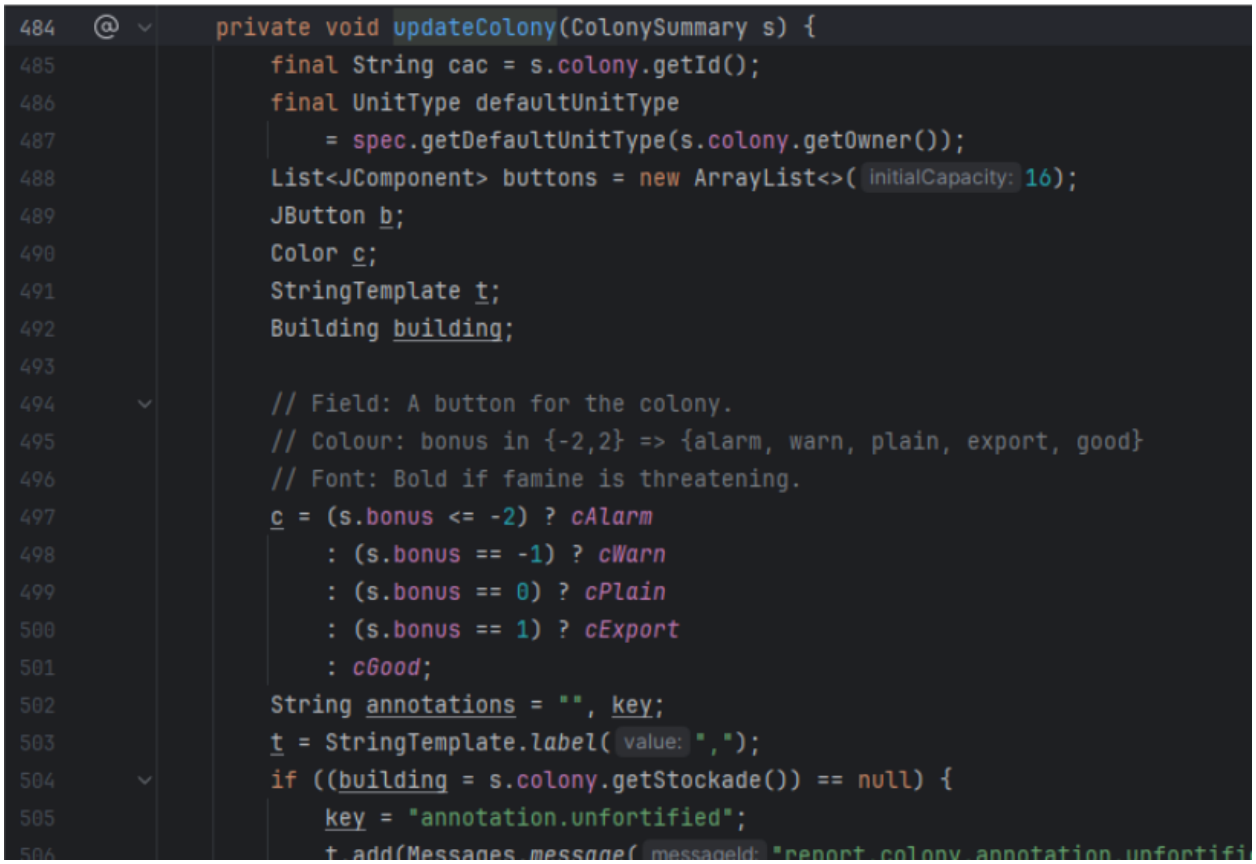
Return to Index.

Author: António Palmeirim (63667)

Code Smell 1:

Long Method

Path: net.sf.freecol/client/gui/panel/report/ReportCompactColonyPanel.java (line 484



```
484  @   private void updateColony(ColonySummary s) {
485          final String cac = s.colony.getId();
486          final UnitType defaultUnitType
487              = spec.getDefaultUnitType(s.colony.getOwner());
488          List<JComponent> buttons = new ArrayList<>( initialCapacity: 16);
489          JButton b;
490          Color c;
491          StringTemplate t;
492          Building building;
493
494          // Field: A button for the colony.
495          // Colour: bonus in {-2,2} => {alarm, warn, plain, export, good}
496          // Font: Bold if famine is threatening.
497          c = (s.bonus <= -2) ? cAlarm
498              : (s.bonus == -1) ? cWarn
499              : (s.bonus == 0) ? cPlain
500              : (s.bonus == 1) ? cExport
501              : cGood;
502          String annotations = "", key;
503          t = StringTemplate.label( value: ",");
504          if ((building = s.colony.getStockade()) == null) {
505              key = "annotation.unfortified";
506              t.add(Messages.message( messageId: "report.colony.annotation.unfortifi
```

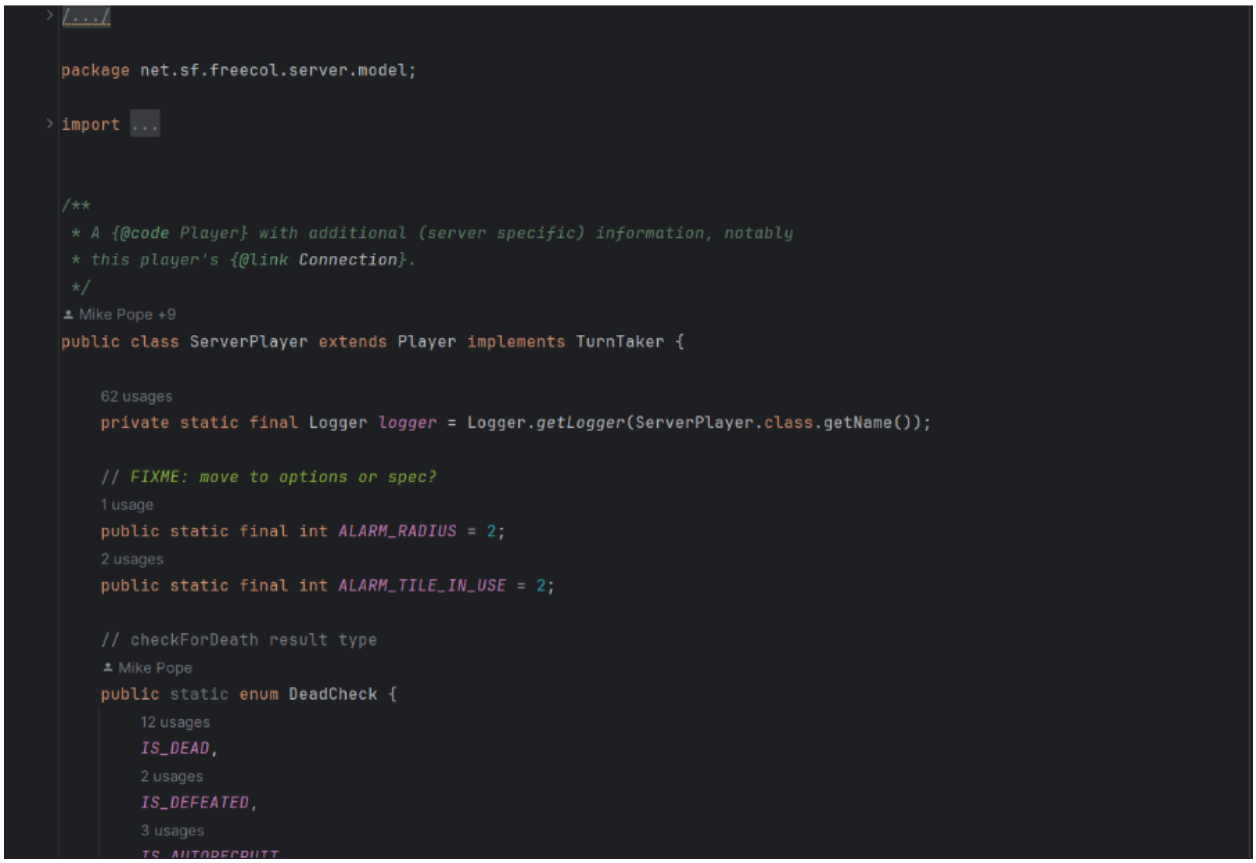
Here we have an extensive method that consists of 484 lines, surpassing the threshold for lines of code in a single method. This is considered to be the long method code smell, which is a bloater code smell. This code smell can create difficulty for any future refactoring and can make the code hard to read. To solve this problem we can divide this method into smaller methods. In our case, where we are updating information of a single colony, we can define the different parts of the colony that are updated on the interface in different methods, making the code easier to read and easier to maintain.

Return to Index.

Code Smell 2:

Large Class

Path: net.sf.freecol/server/model/ServerPlayer.java



```
/.../

package net.sf.freecol.server.model;

import ...


/**
 * A {@code Player} with additional (server specific) information, notably
 * this player's {@link Connection}.
 */
Mike Pope +9
public class ServerPlayer extends Player implements TurnTaker {

    62 usages
    private static final Logger logger = Logger.getLogger(ServerPlayer.class.getName());

    // FIXME: move to options or spec?
    1 usage
    public static final int ALARM_RADIUS = 2;
    2 usages
    public static final int ALARM_TILE_IN_USE = 2;

    // checkForDeath result type
    Mike Pope
    public static enum DeadCheck {
        12 usages
        IS_DEAD,
        2 usages
        IS_DEFEATED,
        3 usages
        IS_AUTORECRUIT
```

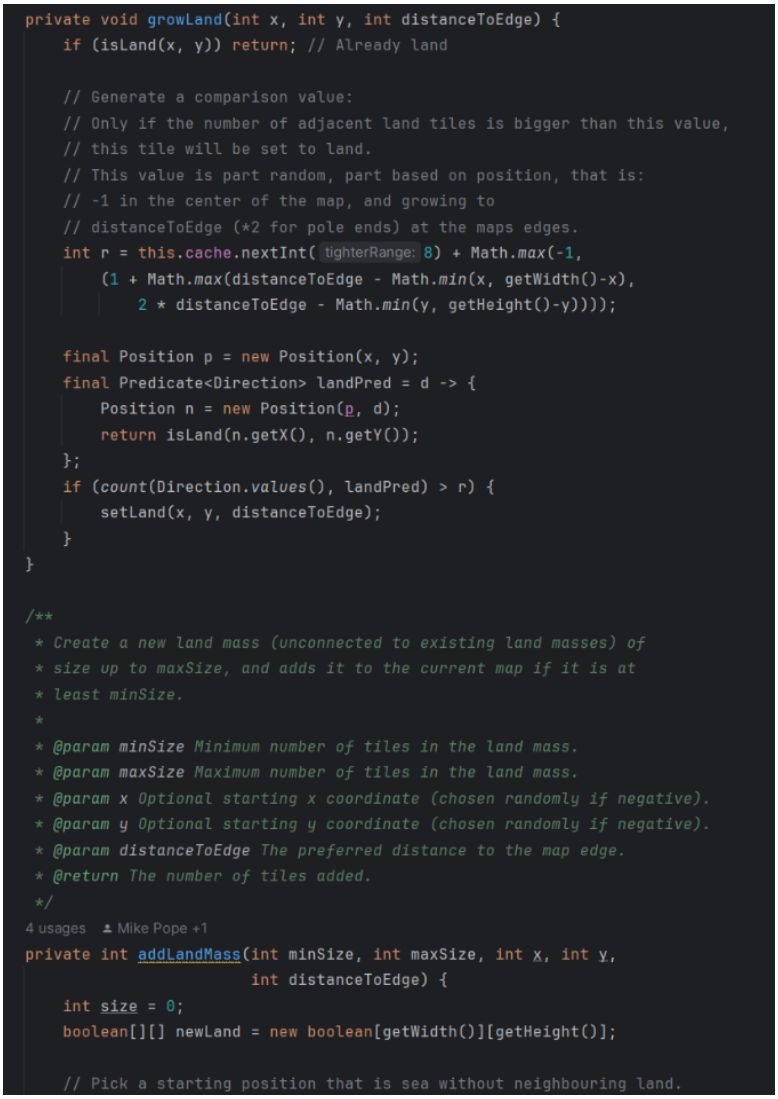
Here we have an extensive class with over 4000 lines of code. This code smell is the large class code smell, part of the bloater group of code smells, and is due to a single class having too many responsibilities. To resolve this problem, in our case, we can divide the ServerPlayer class into smaller classes, for different parts of the player. In other words, we can have different modifications or actions of a player divided into other classes, to dissolve the responsibilities of the larger class.

Return to Index.

Code Smell 3:

Data Clumps

Path: net.sf.freecol/common/model/LandMap.java



```
private void growLand(int x, int y, int distanceToEdge) {
    if (isLand(x, y)) return; // Already land

    // Generate a comparison value:
    // Only if the number of adjacent land tiles is bigger than this value,
    // this tile will be set to land.
    // This value is part random, part based on position, that is:
    // -1 in the center of the map, and growing to
    // distanceToEdge (*2 for pole ends) at the maps edges.
    int r = this.cache.nextInt( tighterRange: 8) + Math.max(-1,
        (1 + Math.max(distanceToEdge - Math.min(x, getWidth()-x),
            2 * distanceToEdge - Math.min(y, getHeight()-y))));

    final Position p = new Position(x, y);
    final Predicate<Direction> landPred = d -> {
        Position n = new Position(p, d);
        return isLand(n.getX(), n.getY());
    };
    if (count(Direction.values(), landPred) > r) {
        setLand(x, y, distanceToEdge);
    }
}

/**
 * Create a new land mass (unconnected to existing land masses) of
 * size up to maxSize, and adds it to the current map if it is at
 * least minSize.
 *
 * @param minSize Minimum number of tiles in the land mass.
 * @param maxSize Maximum number of tiles in the land mass.
 * @param x Optional starting x coordinate (chosen randomly if negative).
 * @param y Optional starting y coordinate (chosen randomly if negative).
 * @param distanceToEdge The preferred distance to the map edge.
 * @return The number of tiles added.
 */
4 usages  Mike Pope +1
private int addLandMass(int minSize, int maxSize, int x, int y,
                        int distanceToEdge) {
    int size = 0;
    boolean[][] newLand = new boolean[getWidth()][getHeight()];

    // Pick a starting position that is sea without neighbouring land.
```

Here we have a data clump code smell, which is part of the bloater code smell group. In my opinion it is a code smell, because the parameters of int x, int y and int distanceToEdge which both addLandMass() and growLand() methods share, could be put into one separate class. To avoid creating a new code smell (data classes) we would need to give the new class certain responsibilities. This new class could be responsible for growing new land adjacent to itself, and to identify itself, this is, returning its coordinates or its distance to the edge. Any additional land would be created by creating a new instance of the class.

Return to Index.

Code Report Review Log

Reviewer: Duarte Inácio 62397

Review 1 (4/11):
Parecem me code smells bastante pertinentes já que em todo o código encontramos métodos e classes bastante extensas, tornando assim o código mais difícil de compreender e modificar.
Code Smell 3 review - Parece uma code smell bastante interessante e faz bastante sentido.
Nada a apontar!!

Reviewer: Rafael Tavares 60608

Review 1 (4/11):
Code smell 2 - Ao apontar este code smell, penso que se ressalta um code smell importante tanto na eficiência, como escalabilidade e mantimento do código. Pela descrição dada conseguimos entender de forma mais profunda a presença e o efeito do code smell em questão. No geral, achei bem estruturado e de fácil compreensão.

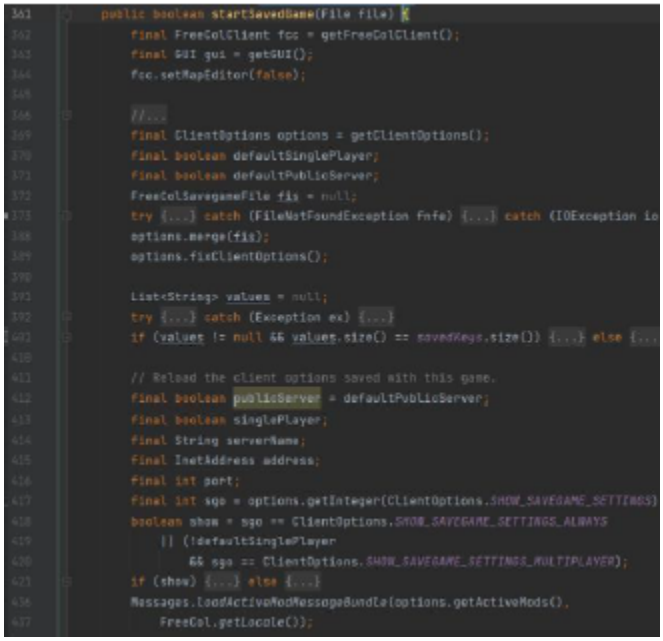
Return to Index.

Author: Duarte Inácio (62397)

Code Smell 1:

Long Method

There are methods that are too large that make code difficult to understand. These methods make maintenance and code legibility difficult. An example of this code smells in the game's code is in the class net/sf/freecol/client/control/ConnectController.java , the method startSavedGame (this method has almost 100 lines of code, demonstrating the gigantic complexity of this code).



Resolution Code Smell 1:

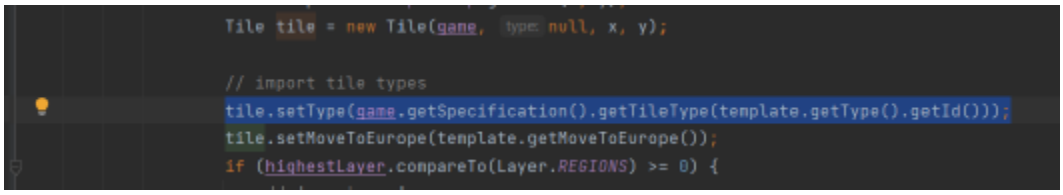
To solve this problem of methods that are too long, I would recommend separating them into smaller, more specific methods, i.e. having smaller methods in which each one performs a single, well-defined task. In the example you have above, I would separate this method into two smaller ones, since the initial part is to get a suggestion of the player's name, single/multiplayer ... and the final part refers to loading the client options saved in this game.

Return to Index.

Code Smell 2:

Message Chain

There are several chained method calls in a single line of code, for example in the net/sf/freecol/server/generator/FreeColMapLoader.java class in the loadMap method on the line: tile.setType(game.getSpecification().getTileType(template.getType().getId());(line 95)



Resolution Code Smell 2:
To solve this problem, I would choose to separate the method calls that are made, storing the intermediate results in variables.

Code Smell 3:

Large Class

As we can see, the net/sf/freecol/client/control/InGameController.java class has around 5387 lines of code, which is a sign that this class is doing too many things, making the code difficult to understand and read.

Code Smell 3 resolution:

I would start by dividing this class into smaller classes in which each one would have a specific responsibility within this InGameController class.One way of thinking could be, creating different classes to deal with the different types (players, colonies, units, etc), this would greatly reduce the complexity of the InGameController class and it would become easier to manage and understand.

Return to Index.

Code Report Review Log

Reviewer: António Palmeirim 63667

Review 1 (14/11):
Os code smells estão bem explicados e a sua localização e resolução também. A única grelha que encontro é na especificação do path para o code smell,
onde tem pontos (".") mete barras ("/"). De resto está ótimo.

Review 2 (19/11):
Já vi que corrigiste a grelha em relação aos pontos e barras. O resto continua ótimo e bem explícito.

Reviewer: Rafael Tavares 60608

Review 1 (Code Smell 2 on 8/11):
Através da explicitação deste code smell conseguimos perceber o seu efeito no código bem como a sua notória presença. Acho que o code smell foi bem apontado.

Return to Index.

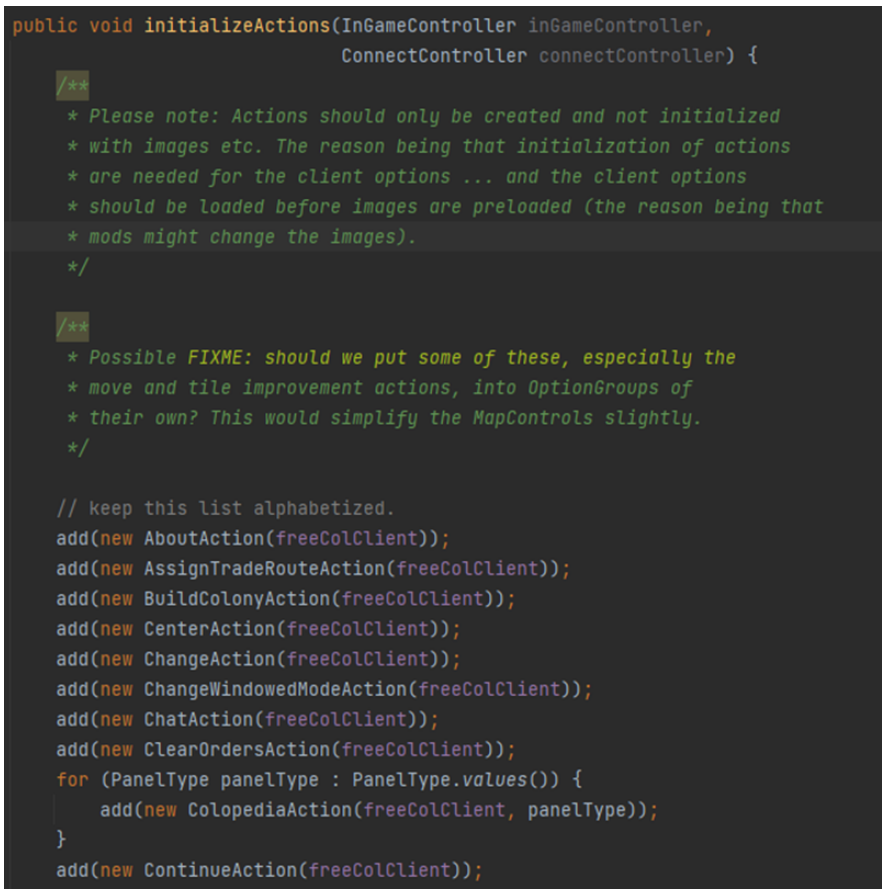
Author: Tiago Santos (63390)

Code Smell 1:

Long Method

Path: net.sf.freecol/client/gui/action/ActionManager.java

The "initializeActions" method shown below has more than 100 lines of code, which means it is very complex and difficult for programmers to read.



```
public void initializeActions(InGameController inGameController,
                              ConnectController connectController) {
    /**
     * Please note: Actions should only be created and not initialized
     * with images etc. The reason being that initialization of actions
     * are needed for the client options ... and the client options
     * should be loaded before images are preloaded (the reason being that
     * mods might change the images).
     */

    /**
     * Possible FIXME: should we put some of these, especially the
     * move and tile improvement actions, into OptionGroups of
     * their own? This would simplify the MapControls slightly.
     */

    // keep this list alphabetized.
    add(new AboutAction(freeColClient));
    add(new AssignTradeRouteAction(freeColClient));
    add(new BuildColonyAction(freeColClient));
    add(new CenterAction(freeColClient));
    add(new ChangeAction(freeColClient));
    add(new ChangeWindowedModeAction(freeColClient));
    add(new ChatAction(freeColClient));
    add(new ClearOrdersAction(freeColClient));
    for (PanelType panelType : PanelType.values()) {
        add(new ColopediaAction(freeColClient, panelType));
    }
    add(new ContinueAction(freeColClient));
```

…

To solve the code smell, the method could be separated into several smaller methods, making it easier to read and understand.

Return to Index.

Code Smell 2:

Large Class

Path:net.sf.freecol/common/model/Tile.java

The following class contains more than 2800 lines, which makes it very complex and difficult to read. It also violates the single responsibility rule by doing too many actions for just one class.

```
/**
 * Represents a single tile on the {@code Map}.
 *
 * @see Map
 */
Michael Pope +15
public final class Tile extends UnitLocation implements Named, Ownable {

    4 usages
    private static final Logger logger = Logger.getLogger(Tile.class.getName());

    public static final String TAG = "tile";

    /** Comparator to sort tiles by increasing distance from the edge. */
    1 usage
    public static final Comparator<Tile> edgeDistanceComparator
        = Comparator.comparingInt(Tile::getEdgeDistance);

    /** Comparator to find the smallest high seas count. */
    1 usage
    public static final Comparator<Tile> highSeasComparator
        = Comparator.comparingInt(Tile::getHighSeasCount);

    /** Predicate to identify ordinary sea tiles. */
    1 usage
    public static final Predicate<Tile> isSeaTile = t ->
        !t.isLand() && t.getHighSeasCount() >= 0;
```

...

To solve the code smell, the class could be divided into smaller classes, separating the responsibilities for each new class. That way, there wouldn't be a class that does too many actions.

Return to Index.

Code Smell 3:

Magic Number

Path: net.sf.freecol/server/model/ServerPlayer.java

On line 606, the number 7 is called the "magic number". It has no direct explanation in the code and can be difficult to understand its usefulness.

```
// OK, the situation is poor, but the REF hangs on if it has
// a minimal number of land and naval units
final int landREFUnitsRequired = 7; // FIXME: magic number
if (land < landREFUnitsRequired) {
    logger.info( msg: "REF surrenders due to land army collapse ("
        + land + " < " + landREFUnitsRequired + ")");
    return true;
}
```

To solve the code smell, the variable "landREFUnitRequired" should have been initialized with "private static final int", earlier.

Return to Index.

Code Report Review Log

Reviewer: Duarte Inácio 62397

Review 1 (4/11):
Tenta meter os code smells com imagens para ser mais fácil de visualizar.

Review 2 (17/11):
Agora sim parece-me um bom documento de code smells!

Reviewer: Rafael Tavares 60608

Review 1 (Code Smell 1 on 4/11):
A explicação sobre o code smell faz sentido e ao analisar os dados recolhidos pelo Tiago conseguimos também ver o code deodorant utilizado, presente no snippet, que ajuda na confirmação da presença do code smell.

Return to Index.

Author: Luís Serrano (60253)

Code Smell 1:

Long Method

Path: net/sf/freecol/FreeCol.java

This try catch block has 300 lines of code, which is pretty extensive for a method, so we can define it as a Long Method.

```
691         CommandLineParser parser = new DefaultParser();
692         boolean usageError = false;
693         try {...} catch (ParseException e) {
939             System.err.println("\n" + e.getMessage() + "\n");
940             usageError = true;
941         }
```

One possible solution to this code smell is breaking this piece of code into different methods so that it wouldn't be as extensive as this.

Return to Index.

Code Smell 2:

Large Class

Path: net/sf/freecol/common/model/Unit.java

This class has around 5000 lines of code, which becomes really hard to analyze and “read”, so we can define this as a Large Class.

One possible solution would be to turn this class into an abstract class, in order to get specific methods for each different unit inside their respective type, making this class a lot smaller.

Return to Index.

Code Smell 3:

Data Class

Path: net/sf/freecol/client/gui/dialog/Parameters.java

This consists of an entire class, which is only setting certain parameters, so it's a Data Class, because it doesn't do anything relevant apart from storing values (and few).

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

Paolo Bizzarri +1
public class Parameters {

    public final int distToLandFromHighSeas;

    public final int maxDistanceToEdge;


    1 usage   Paolo Bizzarri
    Parameters(int distToLandFromHighSeas, int maxDistanceToEdge) {
        this.distToLandFromHighSeas = distToLandFromHighSeas;
        this.maxDistanceToEdge = maxDistanceToEdge;
    }
}
```

A possible solution would be to define these parameters as global variables of the class where they are used, since they are also used in few classes (less than 5).

Return to Index.

Review log

Reviewer: Antonio Palmeirim 63667

Review 1 (4/11):
Acho que os code smells encontrados estão certos. Além disso, podias por
uma informação a indicar a razão destes code smells influenciarem o futuro funcionamento do código.

Return to Index.

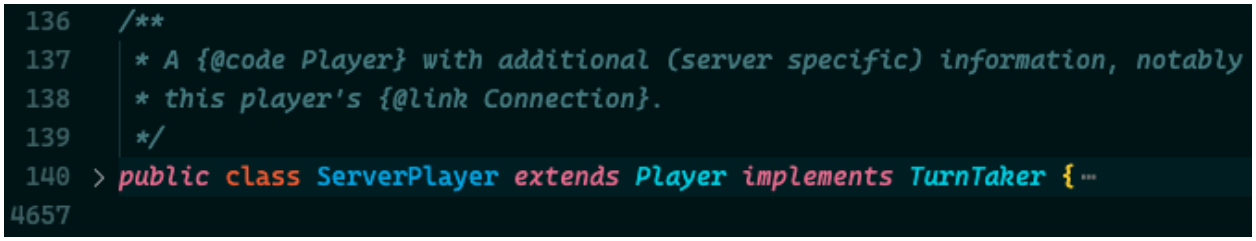
Author: Rafael Tavares (60608)

Code Smell 1:

Large Class

Path: src/net/sf/freecol/server/model/ServerPlayer.java

We can see this is a "Large Class" code smell because the class is excessively large, and it seems like it's trying to do too many things at once, this can lead to low cohesion, making the code harder to understand. It also makes testing challenging since it's difficult to isolate specific behaviors. Furthermore, such large classes are often less reusable, limiting their versatility.



…

To fix this, we can break down this class into smaller, more focused ones, each with a single clear responsibility, improving code organization and maintainability.

Return to Index.

Code Smell 2:

Long Method

Path:src/net/sf/freecol/client/control/InGameController.java Ln. 4229

Here a "Long Method" code smell can be clearly identified as this method can't even fit on screen. After analyzing the method we can conclude it's a "Long Method" code smell which is composed by a method that has grown excessively long and complex. This can make the code harder to understand, maintain, and reuse.

```
4237    private void nativeTrade(NativeTrade nt, TradeAction act,
4238                             NativeTradeItem nti, StringTemplate prompt) {
4239        final IndianSettlement is = nt.getIndianSettlement();
4240        final Unit unit = nt.getUnit();
4241        final StringTemplate base = StringTemplate
4242            .template(value:"trade.welcome")
4243            .addStringTemplate(key:"%nation%", is.getOwner().getNationLabel())
4244            .addStringTemplate(key:"%settlement%", is.getLocationLabelFor(unit.getOwner()));
4245
4246        // Col1 only displays at most 3 types of goods for sale.
4247        // Maintain this for now but consider lifting in a future
4248        // "enhanced trade" mode.
4249        nt.limitSettlementToUnit(n:3);
4250
4251        final Function<NativeTradeItem, ChoiceItem<NativeTradeItem>>
4252            goodsMapper = i -> {
4253            String label = Messages.message(i.getGoods().getLabel(sellable:true));
4254            return new ChoiceItem<>(label, i);
4255        };
4256        while (!nt.getDone()) {
4257            if (act == null) {
4258                if (prompt == null) prompt = base;
4259                act = getGUI().getIndianSettlementTradeChoice(is, prompt,
4260                    nt.canBuy(), nt.canSell(), nt.canGift());
4261                if (act == null) break;
4262                prompt = base; // Revert to base after first time through
4263            }
4264            switch (act) {
4265            case BUY:
4266                act = null;
4267                if (nti == null) {
4268                    nti = getGUI().getChoice(unit.getTile(),
4269                        StringTemplate.key(value:"buyProposition.text"),
4270                        is, cancelKey:"nothing",
4271                        transform(nt.getSettlementToUnit(),
```

...

To improve the situation, we can refactor the long method into smaller, more focused sub-methods, each with a specific role.

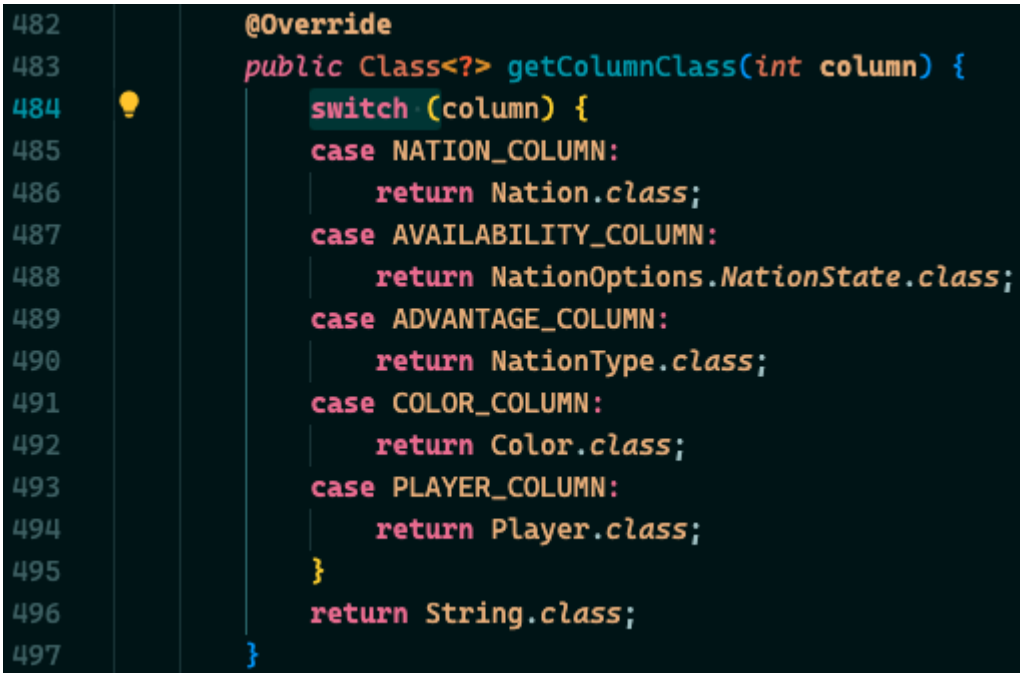
Return to Index.

Code Smell 3:

Switch Statements

Path: src/net/sf/freecol/client/gui/panel/PlayersTable.java Ln. 482 and Ln. 537

In this class where both snippets come from we can notice multiple similar switch statements, indicating a "Switch Statement" code smell. This can lead to inflexible code and makes future modifications challenging and potentially causing code duplication.



```
482     @Override
483     public Class<?> getColumnClass(int column) {
484         switch (column) {
485         case NATION_COLUMN:
486             return Nation.class;
487         case AVAILABILITY_COLUMN:
488             return NationOptions.NationState.class;
489         case ADVANTAGE_COLUMN:
490             return NationType.class;
491         case COLOR_COLUMN:
492             return Color.class;
493         case PLAYER_COLUMN:
494             return Player.class;
495         }
496         return String.class;
497     }
```

Return to Index.

```
537     @Override
538     public Object getValueAt(int row, int column) {
539         if (row ≥ 0 && row < getRowCount()
540             && column ≥ 0 && column < getColumnCount()) {
541             Nation nation = nations.get(row);
542             switch (column) {
543             case NATION_COLUMN:
544                 return nation;
545             case AVAILABILITY_COLUMN:
546                 return nationOptions.getNationState(nation);
547             case ADVANTAGE_COLUMN:
548                 return (nationMap.get(nation) == null) ? nation.getType()
549                     : nationMap.get(nation).getNationType();
550             case COLOR_COLUMN:
551                 return nation.getColor();
552             case PLAYER_COLUMN:
553                 return nationMap.get(nation);
554             }
555         }
556         return null;
557     }
```

To address this, we can employ design patterns like the Strategy Pattern. Instead of switches, we create separate classes or functions for each branch, improving extensibility and maintainability while adhering to object-oriented principles.

Review Log

Review Antonio Palmeirim 63667 (4/11):

Acho que a identificação dos code smells está correta, e as explicações do seu raciocínio e resolução estão bastante desenvolvidos e bem estruturados.

Review André Santos (62331) (4/11):

Documento bem estruturado e organizado de forma clara e concisa, tudo bem explicado sem deixar nenhuma dúvida.

Return to Index.

Code Metrics

Author: André Santos (62331)

Mood Metrics

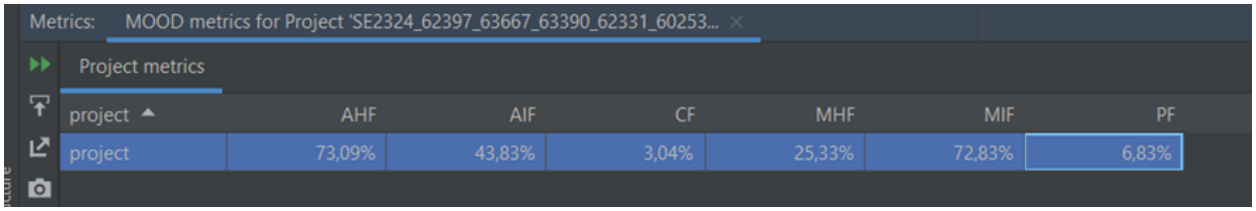


***Figure 1**- Valores dos diferentes fatores*

AHF (Attribute hiding factor): 73,09%

The AHF is relatively high (73.09%), which indicates that a significant part of the attributes in the classes are encapsulated or hidden, indicative of good data encapsulation and maintaining the integrity of the class data.

AIF (Attribute Inheritance Factor): 43,83%

The AIF is moderately high (43.83%), suggesting that there is a reasonable amount of attribute inheritance within the class hierarchy. This indicates that attributes are inherited between classes, promoting code reuse and reducing redundancy.

CF (Coupling Factor): 3,04%

The CF is quite low (3.04%), indicating relatively low coupling between the classes in the project, which contributes to better maintenance and understanding of the code.

Return to Index.

MHF (Method Hiding Factor): 25,33%

The MHF is 25.33%, which suggests a reasonable level of method hiding and encapsulation. This means that the methods are adequately hidden within the classes, contributing to a well-structured design.

MIF (Method Inheritance Factor): 72,83%

The MIF is relatively high at 72.83%, indicating a significant degree of method inheritance within the class hierarchy. That is, methods are inherited and reused between classes, leading to greater code efficiency and reduced redundancy.

PF (Polymorphism Factor): 6,83%

The FP is relatively low at 6.83%, indicating that there is limited use of method substitution in the class hierarchy. That is, this low percentage may be indicative of the fact that child classes rarely override the methods of parent classes.

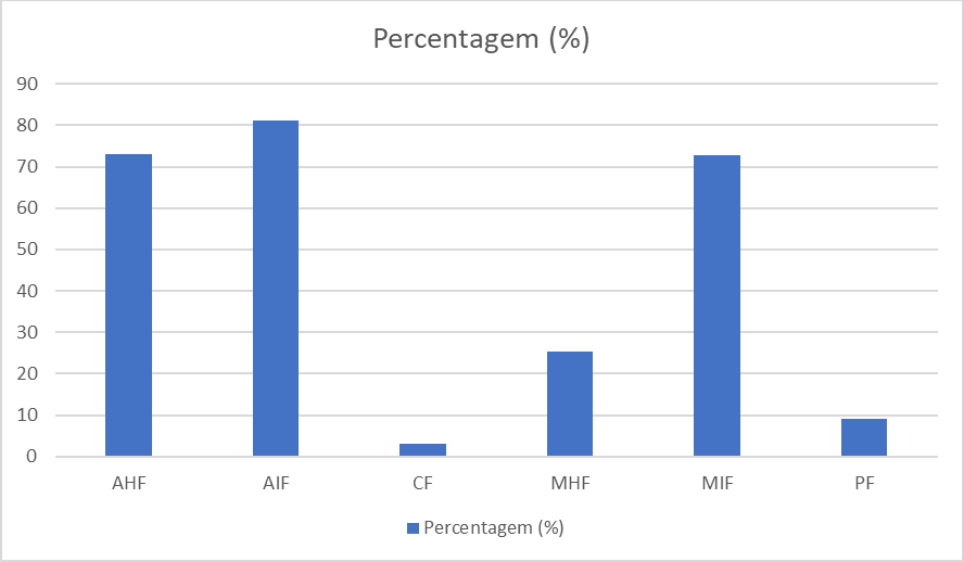


Gráfico 1- Valores dos diferentes fatores

Return to Index.

Relation with Code Smells:

In general, the project's "MOOD metrics" indicate a generally well-structured and encapsulated code, with a reasonable level of attribute and method hiding, as well as method inheritance. However, the relatively low polymorphism factor (PF) may indicate that method substitution is not extensively used in the project.

Code Report Review Log

Reviewer: Duarte Inácio 62397

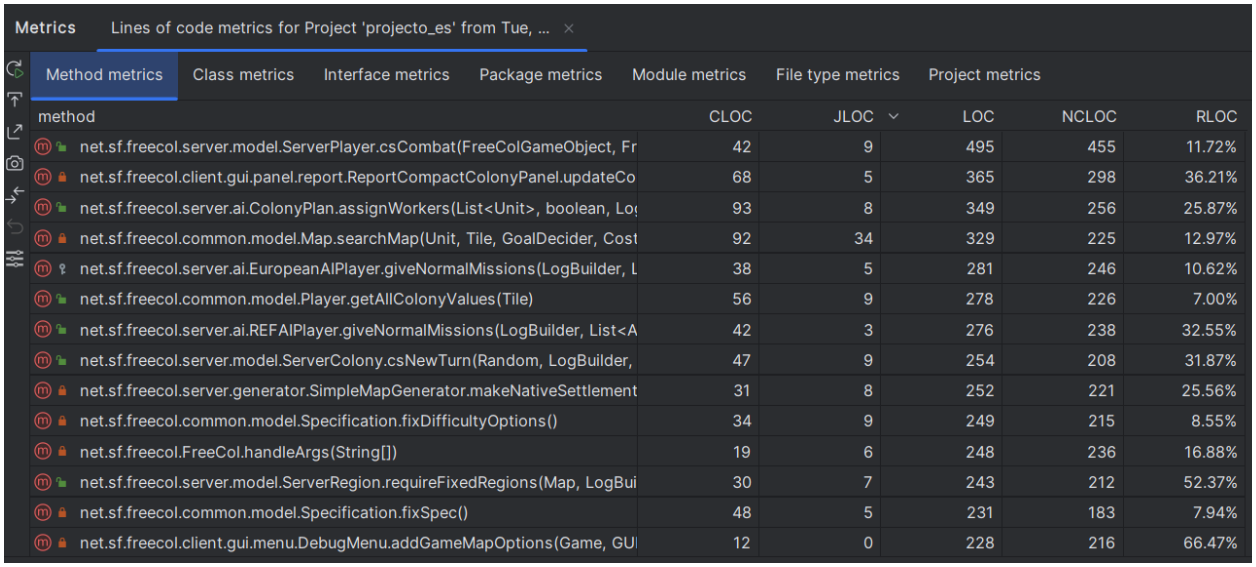
Review 1(5/11):
Parece-me uma análise bastante interessante e concordo com a análise feita dos fatores. Nada a apontar!

Return to Index.

Author: António Palmeirim (63667)

Line of Code (LOC) Metrics

Line of code metrics gives us information about the amount of lines of code that exist in our program. The information is subdivided into sub-metrics that include, packages, methods, classes, interfaces, modules and the lines of codes in the whole project. The lines of codes are divided into different types as well: comments (CLOC), Javadoc (JLOC), non-comment (NCLOC), the percentage of relative code lines (RLOC), and the total lines of code (LOC)



method	CLOC	JLOC	LOC	NCLOC	RLOC
net.sf.freecol.server.model.ServerPlayer.csCombat(FreeColGameObject, Fr	42	9	495	455	11.72%
net.sf.freecol.client.gui.panel.report.ReportCompactColonyPanel.updateCo	68	5	365	298	36.21%
net.sf.freecol.server.ai.ColonyPlan.assignWorkers(List<Unit>, boolean, Lo	93	8	349	256	25.87%
net.sf.freecol.common.model.Map.searchMap(Unit, Tile, GoalDecider, Cost	92	34	329	225	12.97%
net.sf.freecol.server.ai.EuropeanAIPlayer.giveNormalMissions(LogBuilder, L	38	5	281	246	10.62%
net.sf.freecol.common.model.Player.getAllColonyValues(Tile)	56	9	278	226	7.00%
net.sf.freecol.server.ai.REFAIPlayer.giveNormalMissions(LogBuilder, List<A	42	3	276	238	32.55%
net.sf.freecol.server.model.ServerColony.csNewTurn(Random, LogBuilder,	47	9	254	208	31.87%
net.sf.freecol.server.generator.SimpleMapGenerator.makeNativeSettlement	31	8	252	221	25.56%
net.sf.freecol.common.model.Specification.fixDifficultyOptions()	34	9	249	215	8.55%
net.sf.freecol.FreeCol.handleArgs(String[])	19	6	248	236	16.88%
net.sf.freecol.server.model.ServerRegion.requireFixedRegions(Map, LogBui	30	7	243	212	52.37%
net.sf.freecol.common.model.Specification.fixSpec()	48	5	231	183	7.94%
net.sf.freecol.client.gui.menu.DebugMenu.addGameMapOptions(Game, GUI	12	0	228	216	66.47%

After some research to find the rule of thumb when it comes to knowing how many lines of code is too much for a certain method or class, I come to the conclusion that over 200 lines for a class is too much and 50-60 lines is the limit for methods. This is to ensure correct usage of certain philosophies that come with programming in OOP.

With this information we can find many cases where these rules are broken and values go way over the supposed limit.

Return to Index.

Classes

Method metrics | Class metrics | Interface metrics | Package metrics | Module metrics | File type metrics | Project metrics

class	CLOC	JLOC	LOC
net.sf.freecol.client.control.InGameController	1,638	1,297	4,806
net.sf.freecol.common.model.Unit	1,962	1,766	4,263
net.sf.freecol.server.model.ServerPlayer	1,164	777	4,217
net.sf.freecol.common.model.Player	1,921	1,768	3,892
net.sf.freecol.server.control.InGameController	1,073	738	3,451
net.sf.freecol.common.model.Colony	1,223	1,088	2,818
net.sf.freecol.common.model.Specification	883	676	2,740
net.sf.freecol.server.ai.EuropeanAIPlayer	731	545	2,645
net.sf.freecol.common.model.Tile	1,172	1,025	2,504
net.sf.freecol.common.util.CollectionUtils	1,445	1,443	2,374
net.sf.freecol.client.gui.SwingGUI	742	645	2,321
net.sf.freecol.client.gui.GUI	1,457	1,420	2,220
net.sf.freecol.common.model.Map	899	731	2,099
net.sf.freecol.server.control.InGameControllerTest	125	21	1,833

Here we can see that the code has an overwhelming amount of classes with over 2,000 lines of code. This can cause reduced readability and maintainability. It can also be more difficult for debugging and code review.

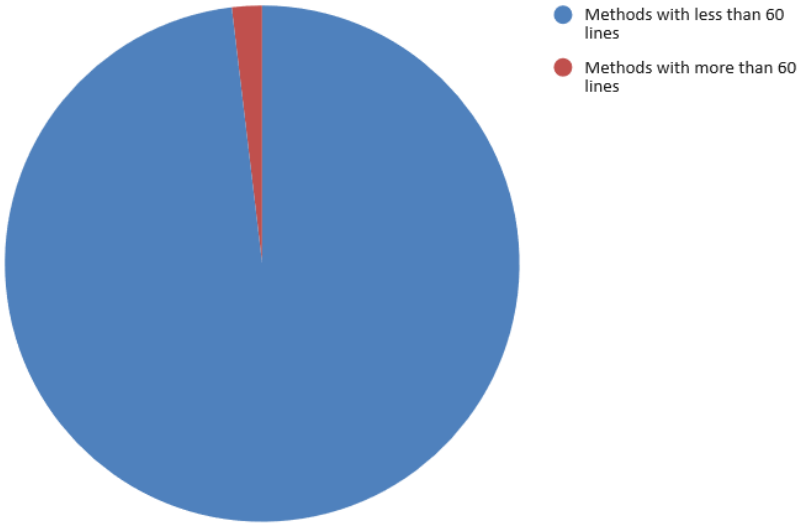
Methods

Method metrics | Class metrics | Interface metrics | Package metrics | Module metrics | File type metrics | Project metrics

method	CLOC	JLOC	LOC	NCLOC	RLOC
net.sf.freecol.server.model.ServerPlayer.csCombat(FreeColGameObject, Fr	42	9	495	455	11.72%
net.sf.freecol.client.gui.panel.report.ReportCompactColonyPanel.updateCo	68	5	365	298	36.21%
net.sf.freecol.server.ai.ColonyPlan.assignWorkers(List<Unit>, boolean, Lo	93	8	349	256	25.87%
net.sf.freecol.common.model.Map.searchMap(Unit, Tile, GoalDecider, Cost	92	34	329	225	12.97%
net.sf.freecol.server.ai.EuropeanAIPlayer.giveNormalMissions(LogBuilder, L	38	5	281	246	10.62%
net.sf.freecol.common.model.Player.getAllColonyValues(Tile)	56	9	278	226	7.00%
net.sf.freecol.server.ai.REFAIPlayer.giveNormalMissions(LogBuilder, List<A	42	3	276	238	32.55%
net.sf.freecol.server.model.ServerColony.csNewTurn(Random, LogBuilder,	47	9	254	208	31.87%
net.sf.freecol.server.generator.SimpleMapGenerator.makeNativeSettlement	31	8	252	221	25.56%
net.sf.freecol.common.model.Specification.fixDifficultyOptions()	34	9	249	215	8.55%
net.sf.freecol.FreeCol.handleArgs(String[])	19	6	248	236	16.88%
net.sf.freecol.server.model.ServerRegion.requireFixedRegions(Map, LogBui	30	7	243	212	52.37%
net.sf.freecol.common.model.Specification.fixSpec()	48	5	231	183	7.94%
net.sf.freecol.client.gui.menu.DebugMenu.addGameMapOptions(Game, GUI	12	0	228	216	66.47%

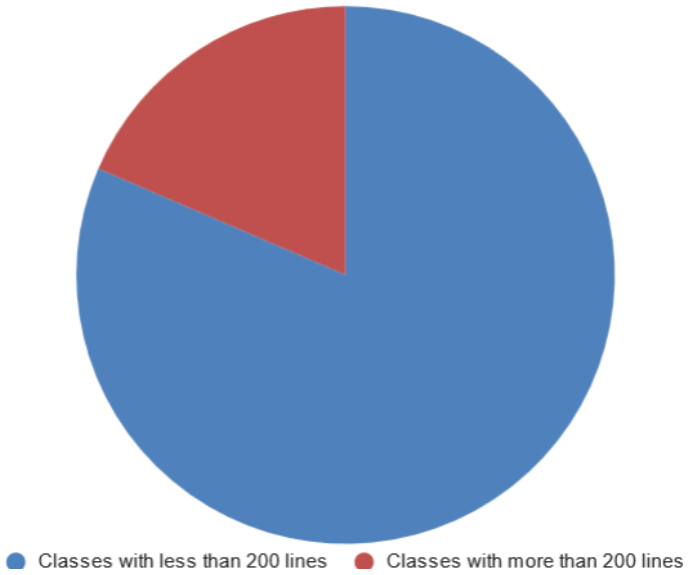
Return to Index.

In the screenshot above, we can see that we have many methods with over 200 lines of code, and most of them have the majority with non-comment lines of code, meaning that most of it is actual code and not simply commenting. Some methods will have many lines of code, such as setting up user interfaces, etc.. But in our example, the method cs.Combat, consisting of 495 total lines of codes, is just to set up combat between an attacker and defender.



Using the metrics to create a pie chart, we can see that there is only a small amount of methods in the full project with more than the ideal 60 lines, but even so, the value is still noticeable.

Using the same metrics to create a pie chart, but this time for the classes. After evaluation we can see that there is a substantial amount of classes with more than the 200 ideal amount, which can cause future problems in debugging and code maintenance



Return to Index.

Relation to Code Smells

The values we find can help us identify code smells that have relation to methods or classes that are too large. This can affect future code refactoring or debugging for future developers. Documenting larger methods or classes can also be more difficult and more susceptible to errors in understanding between documenter and coder.

Code Report Review Log

Reviewer: Rafael Tavares 60608

Review 1 (5/11):
O relatório está bem elaborado e oferece uma compreensão e visão geral abrangentes sobre as métricas em questão. As métricas analisadas fornecem informações valiosas sobre as linhas de código do programa e são uma mais valia para a otimização do código escrito.

Return to Index.

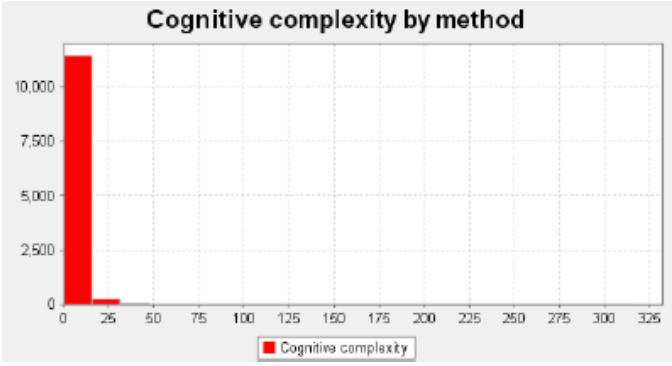
Author: Duarte Inácio (62397)

Complexity metrics

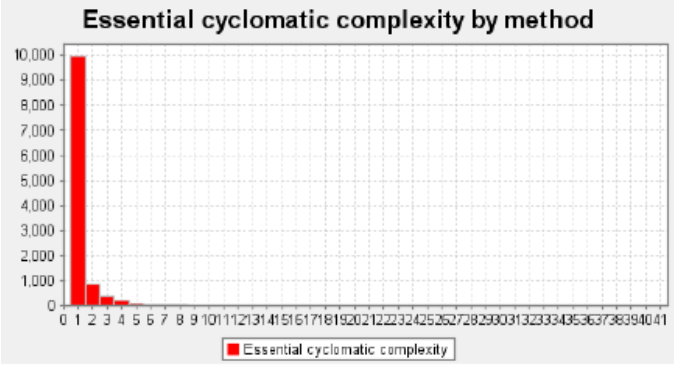
Complexity metrics are tools that help us assess the complexity of a project and identify parts of the code that may be difficult to understand.

Method metrics

The **cognitive complexity** CogC evaluates how difficult it is to understand a unit of code, i.e. it measures the cognitive load that a programmer needs to understand a unit of code. The lower the CogC the better, so that the code is easy to understand.

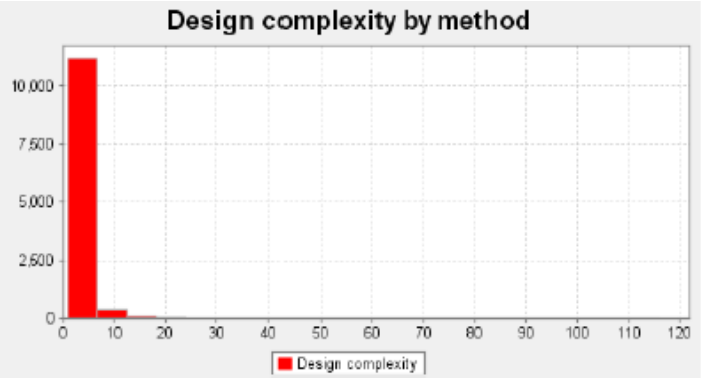


Essential complexity (ev(G)) is a metric that focuses on the structural complexity of the code and the quality of the code. ev(G) is an extension of the cyclomatic complexity metric that calculates the number of independent paths that can be traversed during the execution of that method. This metric takes into account all possible forms of flow control, such as loops, conditional decision structures (like if-else, switch-case, etc).

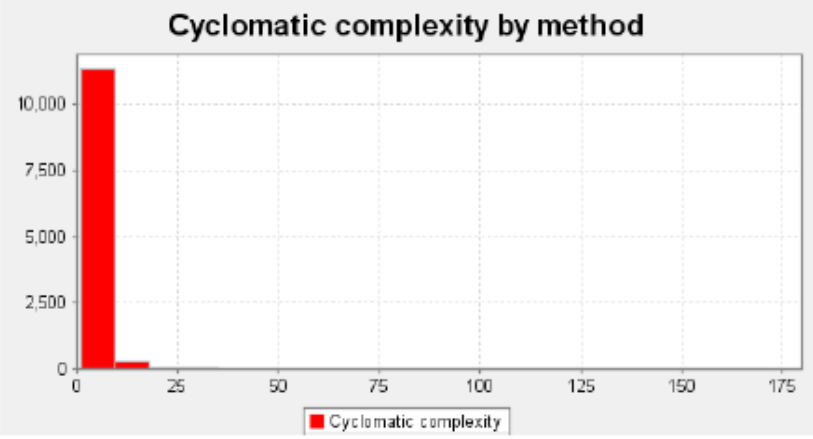


Return to Index.

Design complexity (iv(G)) is a qualitative metric that considers the organization and structure of the code.



Cyclomatic complexity (v(G)) evaluates the complexity of the code, taking into account all the elements that affect the flow of the method. In practical terms, cyclomatic complexity measures the number of independent paths that can be taken in the code, i.e. each conditional structure, each loop and each switch-case increases the cyclomatic complexity, as it introduces new possible paths into the method.

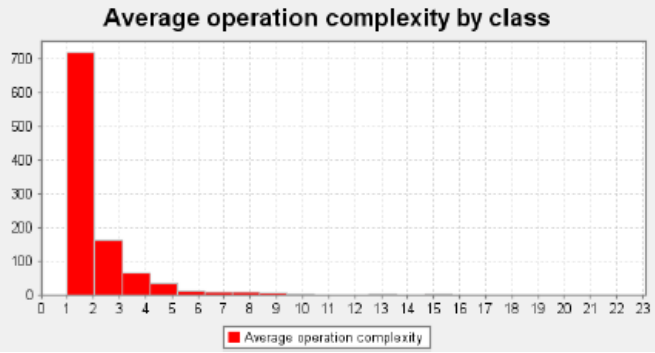


Return to Index.

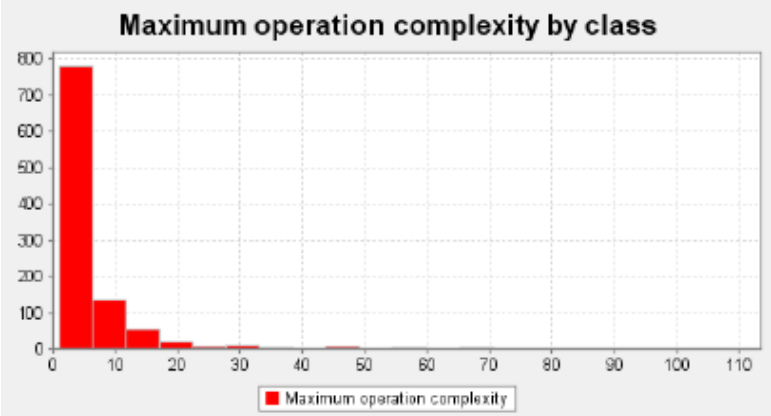
Class metrics

Class metrics are metrics used to evaluate the characteristics and quality of classes in a project/system.

The **average complexity of operations** (OCavg) measures the complexity of operations in relation to the logic contained in each method.

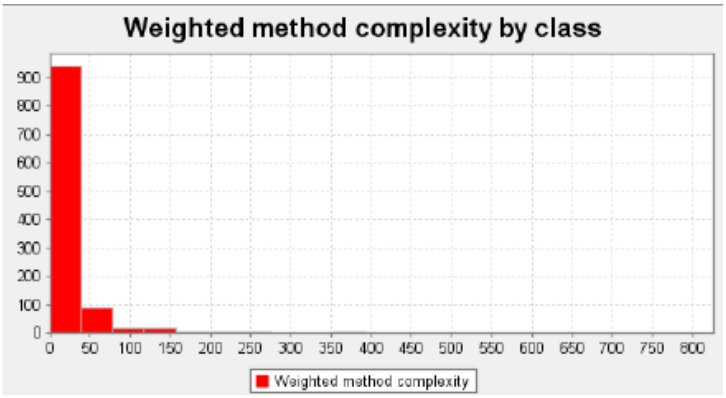


The **maximum complexity of operations** (OCmax) is the metric that evaluates the highest complexity of all the methods in a system.



Return to Index.

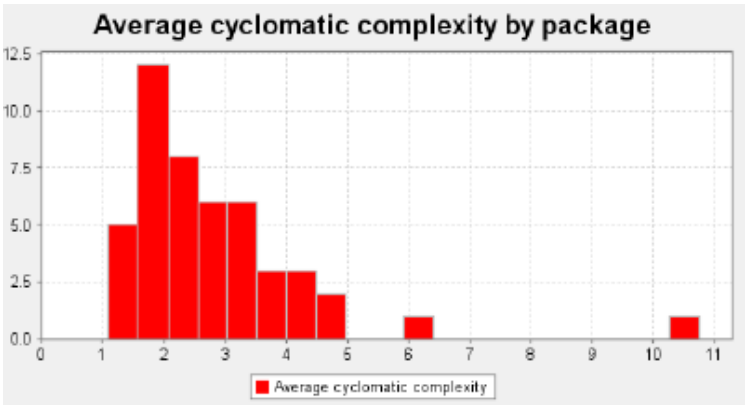
Weighted method complexity (WMC) is a metric that evaluates the complexity of methods or operations in a system, assigning different weights to the various elements found in a method.



Package metrics

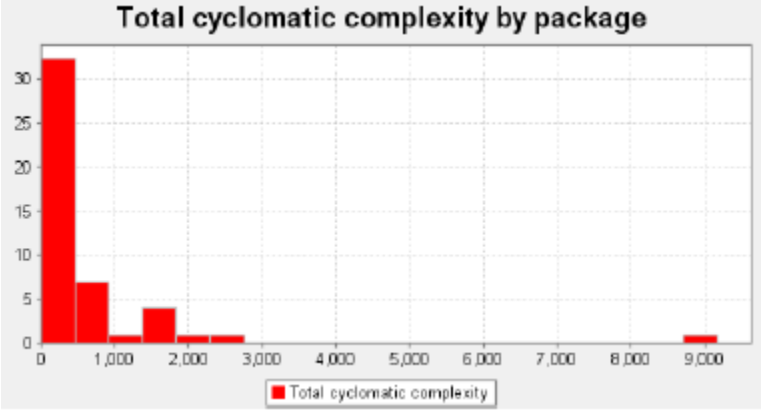
Package metrics are particularly important in large-scale projects, where the system structure is complex and maintenance is critical.

Average cyclomatic complexity (v(G)avg) is a metric that calculates the average complexity of each method in a system, with the package with the highest v(G)avg being net/sf/freecol/client/gui/tooltip.



Return to Index.

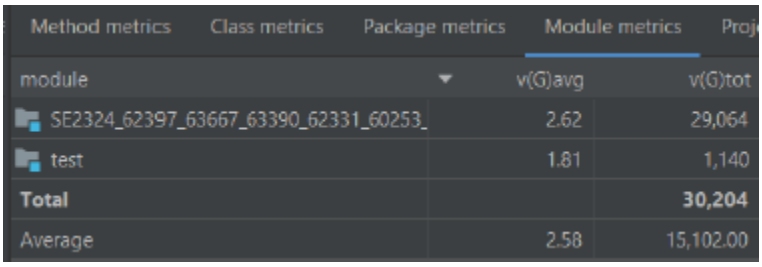
The **total cyclomatic complexity** (v(G)tot) is a metric that evaluates the overall complexity of a system, taking into account all the methods present. The package with the highest v(G)tot is the net/sf/freecol/common/model package



Module Metrics

Module metrics are metrics that evaluate the characteristics, quality and complexity of modules.

In our work, we can see that the module with the highest v(G)avg and v(G)tot is the main module of the project. On the other hand, the module with the lowest v(G)avg and v(G)tot is the test module.



Return to Index.

Project metrics

Project metrics are metrics used to evaluate the quality and performance of the project.

The project has a v(G)avg of 2.58, which is relatively low, meaning that the project's methods have low complexity on average. As for v(G)tot, we can see that it has a value of 30.204, which means that the system as a whole has relatively high complexity, which can make the code more difficult to manage as a whole.

Relationship between metric complexities and Code Smells

Long Method - Cyclomatic complexity and cognitive complexity are related to very long methods, because methods with high cyclomatic complexity can have many independent paths, making them longer and more difficult to understand, and in the case of cognitive complexity, this is related to long methods because if the cognitive load that a programmer needs to understand a unit of code is greater, it means that this method is not simple to understand.

Large Class - In addition to cyclomatic complexity, the weighted complexity of methods is also related to this code smell, since a large number of methods in a class usually indicates that the class is large.

Message Chain – The weighted complexity of methods helps to identify methods that make several chained method calls in a single line of code, as this code smell involves calling methods serially.

Code Report Review Log

Reviewer: Antonio Palmeirim 63667

Review 1: (5/11):
A avaliação dos code metrics e a explicação está bastante explícita e bem estruturada.

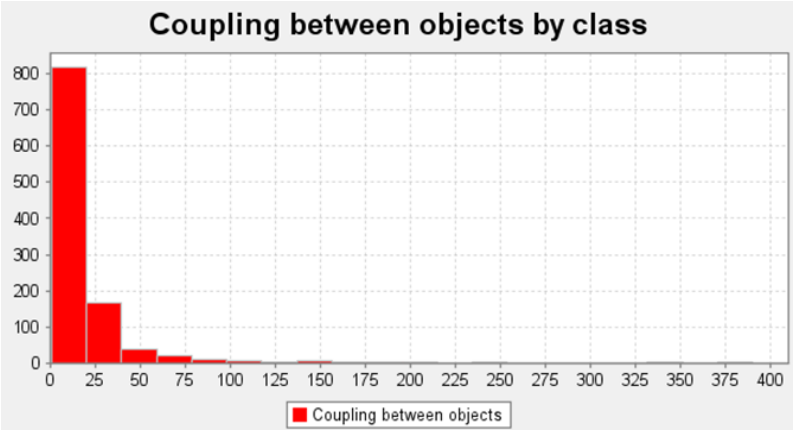
Return to Index.

Author: Tiago Santos (63390)

Chidamber and Kemerer

CBO (Coupling Between Objects):

The BOD value represents the degree of coupling of the class, counting the number of other classes with which the class interacts. A high BOD indicates strong dependency. Although the project has several classes with low BODC, there are quite a few classes with high BODC. This strong coupling can make the code less flexible.

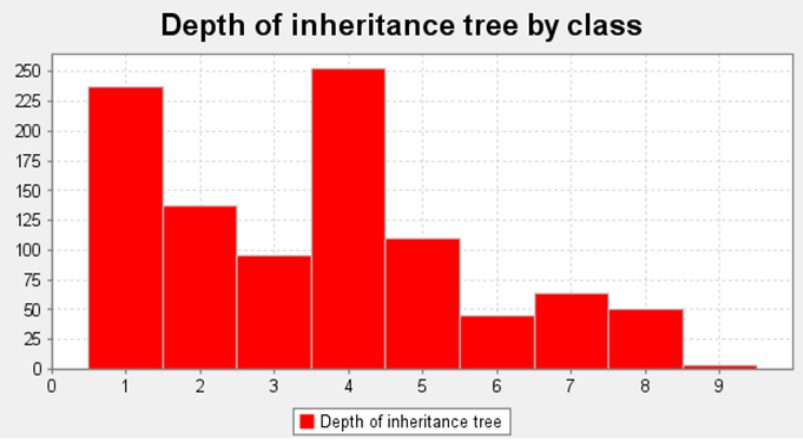


DIT (Depth of Inheritance Tree):

The DIT represents the level of inheritance of a class. The higher the DIT value, the deeper the inheritance hierarchy in which the class is involved.

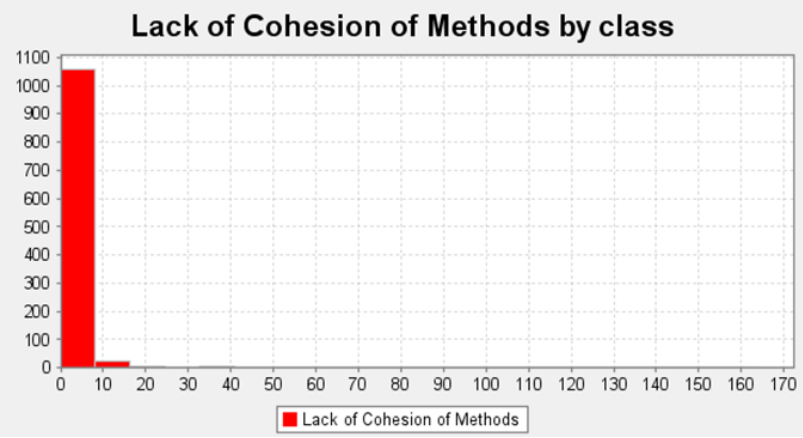
In the project, there is a balanced number of classes with high and low DIT, so there are not many possibilities for hierarchies that are too deep, or classes that are not reused enough.

Return to Index.



LCOM (Lack of Cohesion in Methods):

LCOM evaluates the cohesion of methods in a class. A low LCOM value indicates poorly related methods, which can affect the maintenance and understanding of the code. In the project, there are many classes with low LCOM.

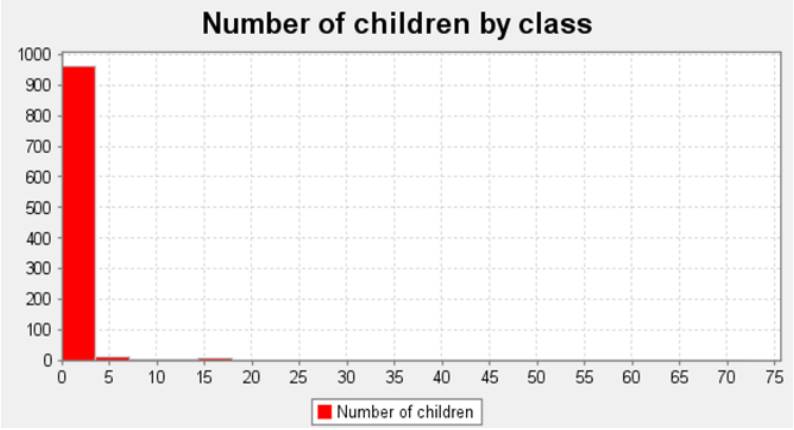


Return to Index.

NOC (Number of Children):

The NOC indicates how many direct subclasses inherit from the class in question.

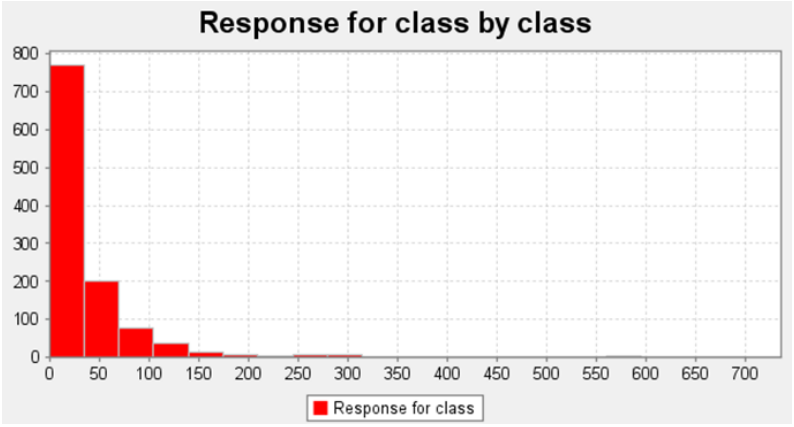
In the project, there are several classes with a high NOC value, which may indicate a parent class with a large number of direct subclasses. This can make the code less flexible.



RFC (Response For a Class):

The RFC calculates the number of methods in the class (inherited and local methods). And it helps to assess the responsibility of the class and its potential impact on other classes.

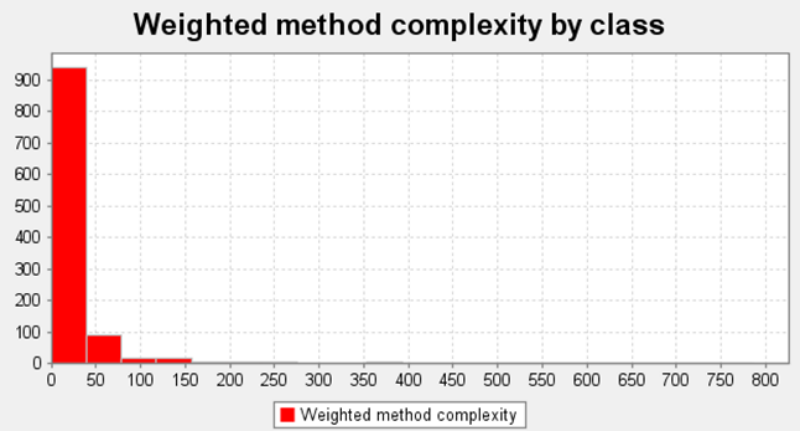
The project contains several classes with high RFC values, which may indicate classes with too many responsibilities.



Return to Index.

WMC (Weighted Methods per Class):

The WMC represents the complexity of the class. The higher the WMC value, the more complex the class. This is because the WMC measures complexity by adding up the number of methods in the class, giving each method a weight based on its complexity. In the project, there are several classes with very high WMC. This can be indicative of a violation of the single responsibility principle and can be difficult to understand and maintain.



Code Report Review Log

Reviewer: André Santos 62331

Review 1 (5/11):
A análise das métricas de código identificou preocupações com alto acoplamento, baixa coesão, hierarquias complexas e classes com muitas responsabilidades.
Concordo que refatorações são necessárias para melhorar a qualidade do código. De acordo com tudo presente no documento, nada a acrescentar.

Return to Index.

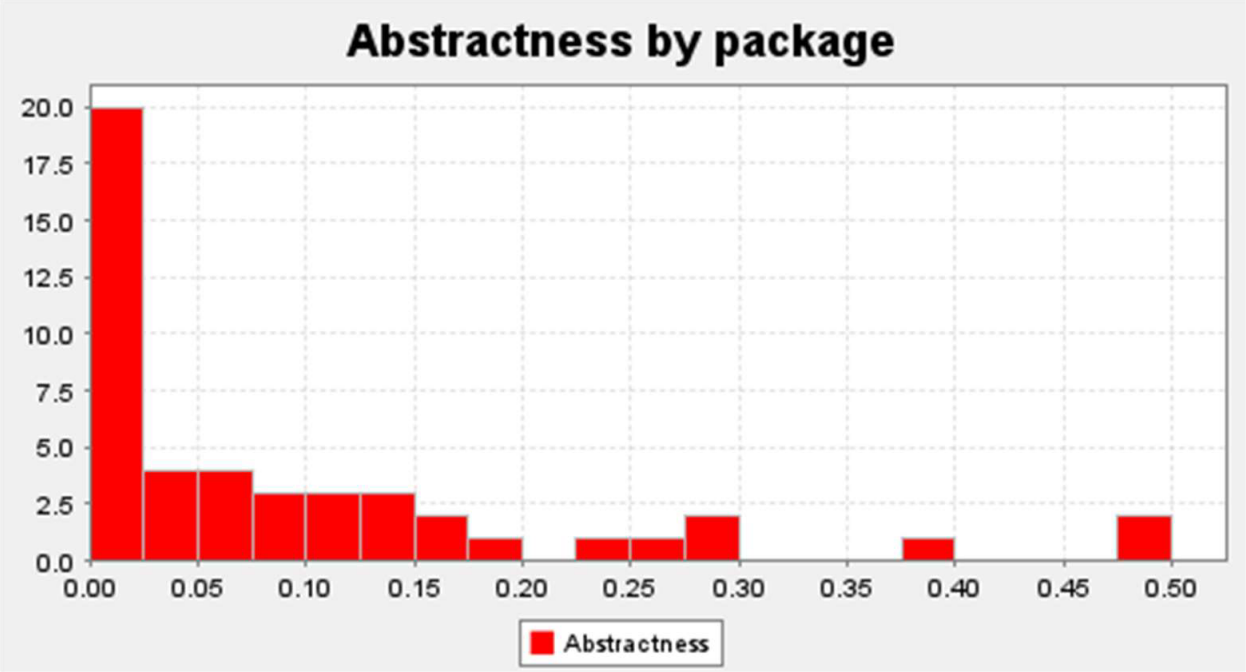
Author: Luís Serrano (60253)

Martin Packaging Metrics

The Martin Packaging Metrics aim to provide more details about the packages in a project, in this case FreeCol. They give us details about the modularity, stability and coupling of different packages within the project.

Abstractness (A)

Abstraction measures the proportion of abstract classes and interfaces in a package. It ranges from 0.0 (no abstract classes/interfaces) to 1.0 (fully abstract).

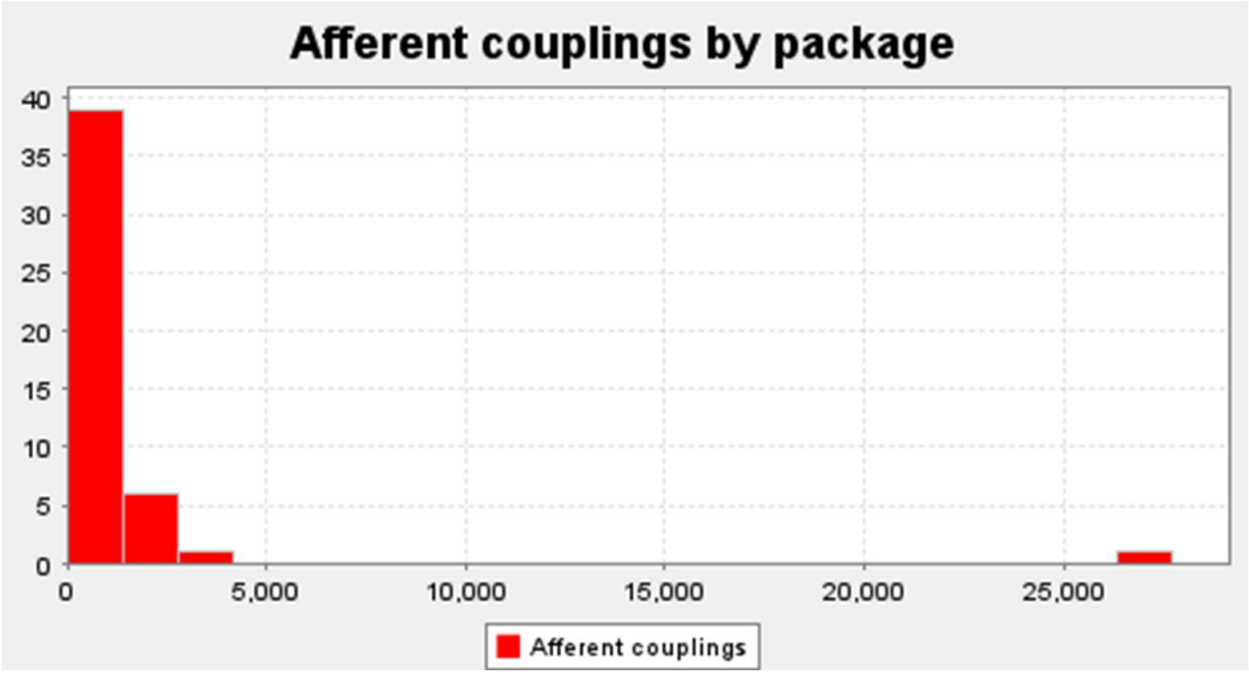


Observations: The highest abstraction value is found in the package net.sef.freecol.client.gui.video and net.sf.freecol.common.io.sza with a value of 0.5.
Packages such as net.sef.freecol.client.gui.animation and net.sf.freecol.client.gui.label also display relatively high abstraction values (0.4 and 0.3).

Return to Index.

Afferent Couplings (Ca)

The afferent couplings represent the number of dependencies that go into a package.
It indicates how many other packages depend on a specific package.

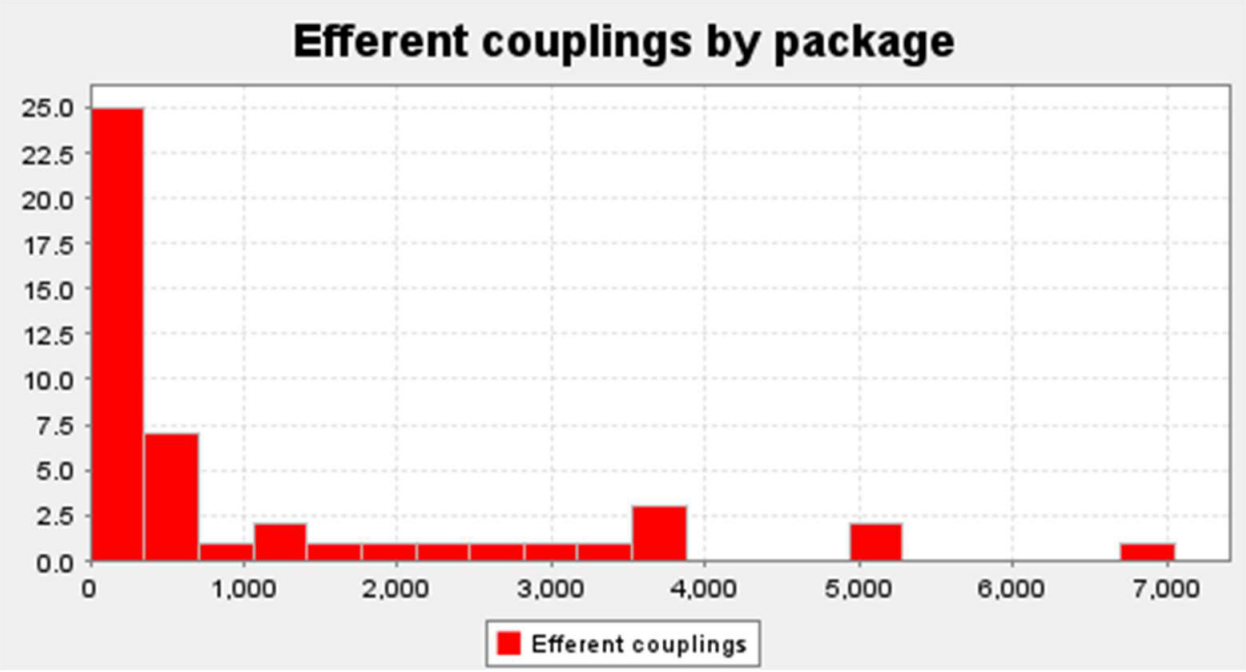


Observations: The net.sf.freecol.common.model package has the highest number of efferent couplings with 32537 dependencies. This number is incredibly different from the second package with the highest number of efferent couplings (net.sf.freecol.common.util with 3492 dependencies). The packages net.sf.freecol.common.model.mission, net.sf.freecol.metaserver and net.sf.freecol.tools have the lowest number of efferent couplings with 0 dependencies.

Return to Index.

Efferent Couplings (Ce)

The efferent couplings represent the number of dependencies coming out of a package. It indicates how many packages a specific package depends on.

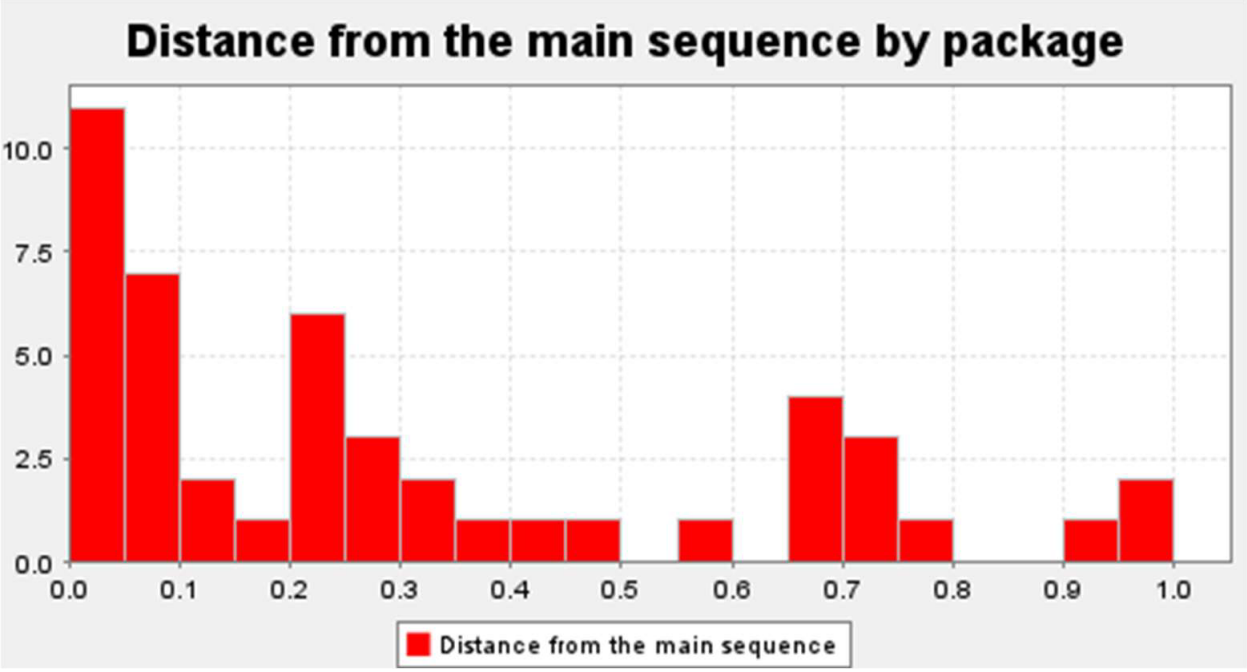


Observations: The net.sf.freecol.server.model package has the highest number of efferent couplings with 8509 dependencies. The net.sf.freecol.client.gui.video package has the lowest number of efferent couplings with 4 dependencies.

Return to Index.

Distance from the Main Sequence (D)

The distance from the main sequence measures the balance between abstraction and stability of a package. A value close to 0.0 indicates a well-balanced package.

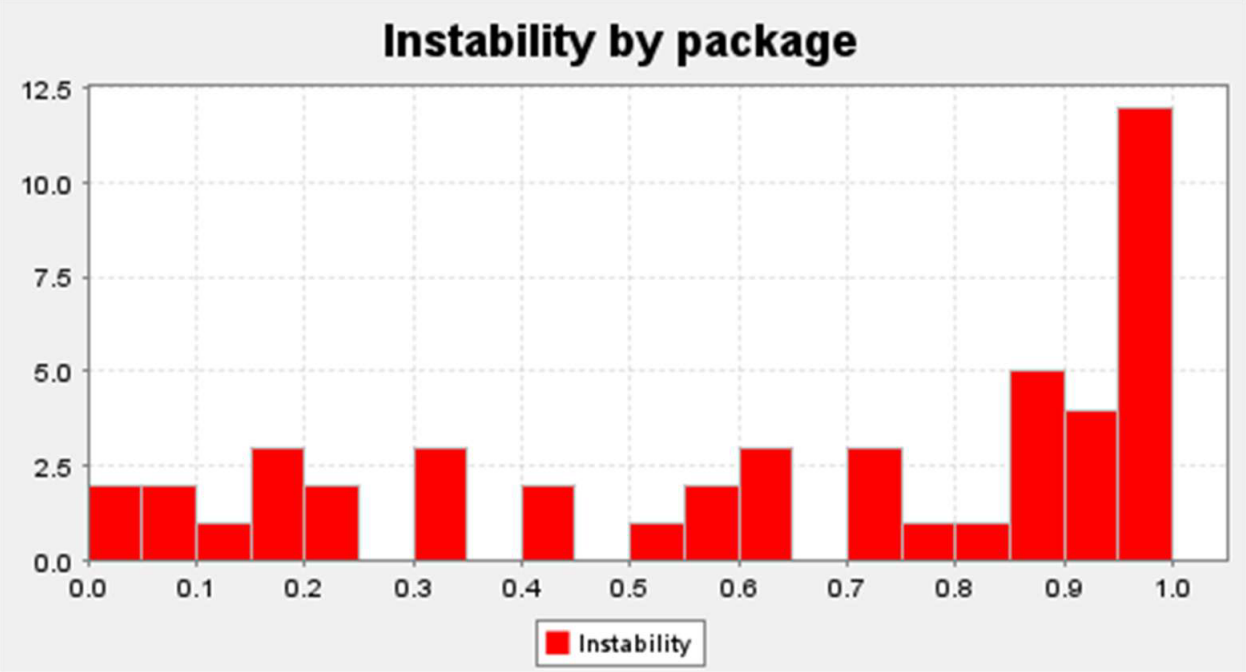


Observations: Packages such as net.sf.freecol.metaserver, net.sf.freecol.tools and net.sf.freecol.server.model have low distance values (0.0), indicating a balanced mix of abstraction and stability. The net.sf.freecol.common.util package has a fairly high distance value (1.0), suggesting that it may need further refinement.

Return to Index.

Instability (I)

Instability measures the tendency of a package to change. A value closer to 0.0 indicates stability, while closer to 1.0 indicates instability.



Observations: The net.sf.freecol.common.util package exhibits the highest level of stability, with a value of 0.0. On the other hand, packages such as net.sf.freecol.model.mission, net.sf.freecol.metaserver and net.sf.freecol.tools have the highest instability values.

Return to Index.

Conclusion

The Martin Packaging Metrics offer valuable insights into the structural characteristics of the FreeCol project. It is essential to find a balance between abstraction, coupling and stability to ensure maintainability and flexibility in the code base.

Review log

Reviewer: André Santos 62331

Review 1 (5/11):
Concordo com tudo o que foi analisado pelo Luis, tudo bem explicado e bem desenvolvido.

Return to Index.

Author: Rafael Tavares (60608)

Dependency Metrics

Introduction & Metrics data

As required, we will be analyzing the code metrics for the FreeCol codebase, in this report, more specifically, the Dependency Metrics, extracted using the MetricsReloaded plugin for IntelliJ IDEA. The extensive dataset of metrics collected can be found in the spreadsheet sent alongside the report, however, to make this metrics overview simpler, we will only take a look at the average values for each field.

Metrics Fields

To better comprehend what these metrics mean, and why are they valuable, we will first understand the purpose of each field:

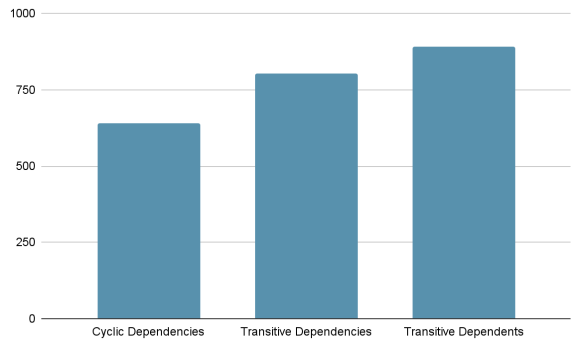
1. **Number of Cyclic Dependencies:** Measures how many times classes or packages participate in cycles with other classes or packages.
2. **Number of Dependencies:** Counts the total number of direct relationships where a class or package relies on other classes or packages to function.
3. **Number of Transitive Dependencies:** Tallies the dependencies that are not direct but are through a chain of other classes or packages. For instance, if A depends on B, and B depends on C, then A has a transitive dependency on C.
4. **Number of Dependents:** Indicates the count of other classes or packages that directly rely on a given class or package.
5. **Number of Transitive Dependents:** Similar to transitive dependencies, this counts how many classes or packages indirectly rely on a given class or package through a chain of dependencies.
6. **Number of Package Dependencies:** The specific count of direct dependencies at the package level, reflecting how many other packages a single package depends on.
7. **Number of Dependent Packages:** This metric shows how many other packages directly depend on a given package, indicating its level of responsibility and importance within the application's package structure.

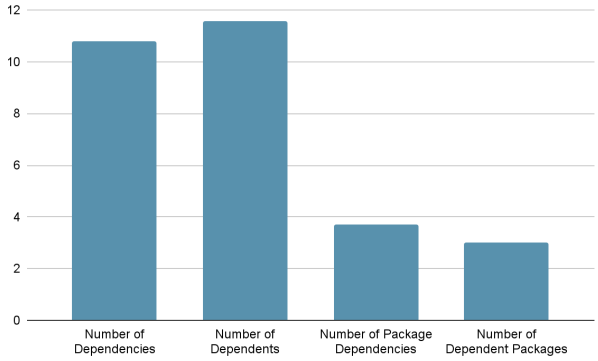
Return to Index.

Class Values

Average values for each field, in regards to Classes:

Number of Cyclic Dependencies	639.131
Number of Dependencies	10.778
Number of Transitive Dependencies	804.308
Number of Dependents	11.574
Number of Transitive Dependents	890.477
Number of Package Dependencies	3.725
Number of Dependent Packages	3.007





From these average values, it's evident that the codebase exhibits a high degree of interconnectivity, particularly in terms of cyclic dependencies among classes. With an average of 639.131 cyclic dependencies, there's a strong indication that the codebase could suffer from tight coupling and poor separation of concerns, making it difficult to maintain and evolve.

A significant number of transitive dependencies (804.308 on average) also suggests that classes are not only depending on many other classes directly but also through chains of dependencies, this could lead to challenges in understanding the impact of changes, potential for ripple effects across the codebase, and difficulties in ensuring the isolation of components for testing and maintenance.

The number of dependents and transitive dependents (11.574 and 890.477, respectively) are indicative of certain classes being heavily relied upon, either directly or indirectly, which may point towards classes that are too central or hold too much responsibility. Such classes could become problematic "Large Class", with an excessive number of responsibilities, making them critical points of failure.

Return to Index.

On the packaging side, while the direct package dependencies and dependents (3.725 and 3.007, respectively) don't seem to be as extreme, they still suggest a moderate level of coupling between packages. This could potentially lead to package coupling, where changes in one package may have a cascading effect on others, indicating that refactoring towards more modular and loosely coupled packages might be beneficial.

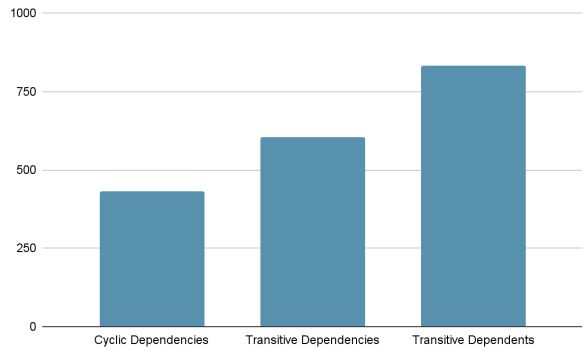
Overall, the extreme values in cyclic and transitive dependencies strongly suggest the presence of code smells related to class design and package structure. Refactoring to address these issues could greatly improve the maintainability and robustness of the codebase.

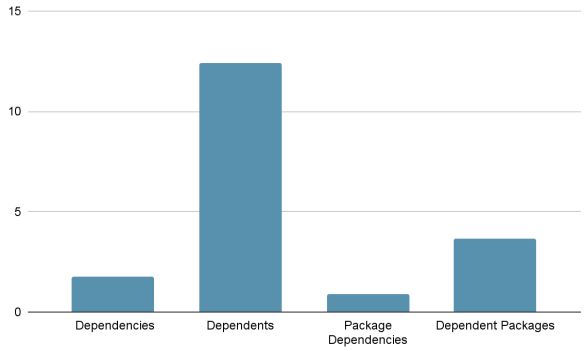
Return to Index.

Interface Metrics

Average values for each field, in regards to Interfaces:

Number of Cyclic Dependencies	431.415
Number of Dependencies	1.756
Number of Transitive Dependencies	603.780
Number of Dependents	12.439
Number of Transitive Dependents	832.171
Number of Package Dependencies	0.878
Number of Dependent Packages	3.683





For interfaces, the average number of cyclic dependencies is significant at 431.415, though not as high as with classes, suggesting that interfaces also contribute to the cyclic complexity of the codebase, which might imply an overuse or misuse of interfaces in the system.

The average number of dependencies for interfaces is relatively low at 1.756, indicating that interfaces tend to have fewer direct relationships compared to classes. However, just like in classes, the number of transitive dependencies remains high (603.780), which points towards a complex network of indirect interface relationships that could lead to hidden dependencies and make changes more difficult to manage.

The number of dependents and transitive dependents is higher for interfaces (12.439 and 832.171, respectively) compared to classes, this reflects interfaces' purpose in the system but also flags a potential over-reliance on interfaces, which could suggest some interfaces are not cohesively defined or are carrying too much responsibility.

Return to Index.

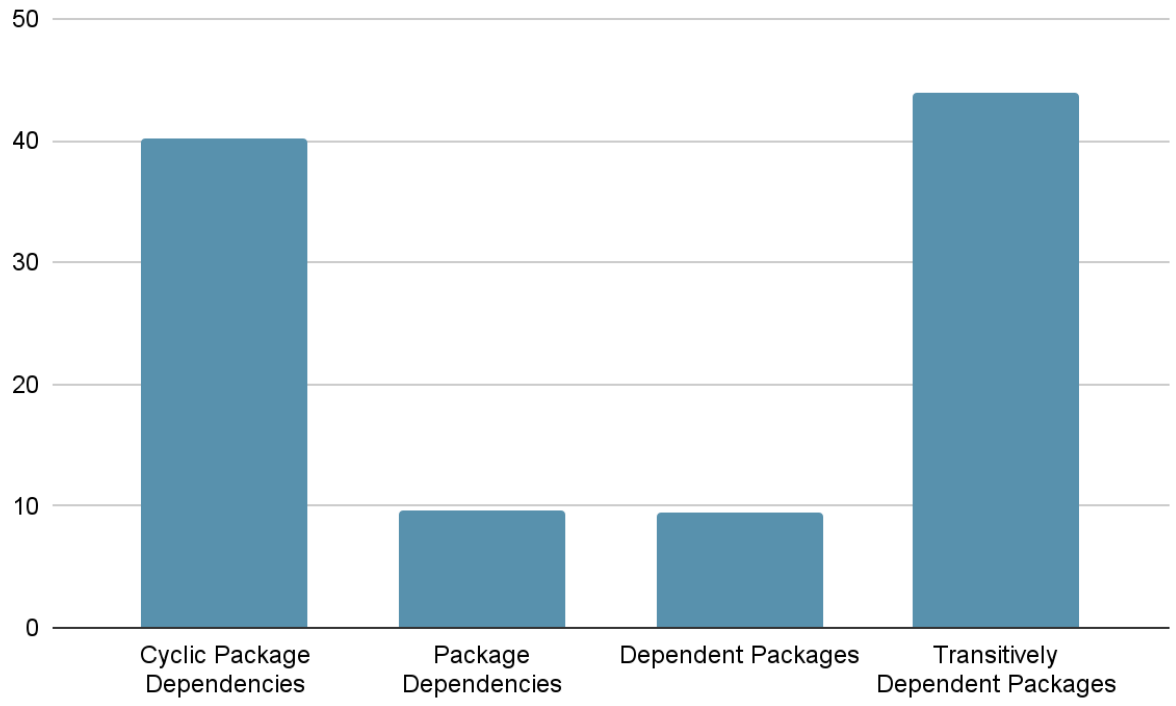
Unlike in classes, the number of package dependencies and dependent packages for interfaces are considerably lower (0.878 and 3.683, respectively), implying that interfaces are less tied to the structure of packages and may be more reusable across the codebase. This is a positive sign in terms of package design, but given the other metrics, it still requires careful management to prevent scenarios where interfaces contain methods that not all implementers actually need, violating the interface segregation principle.

Return to Index.

Package Metrics

Average values for each field, in regards to Packages:

Number of Cyclic Package Dependencies	40.133
Number of Package Dependencies	9.556
Number of Dependent Packages	9.444
Number of Transitively Dependent Packages	43.933



For packages, the metrics suggest a relatively contained level of cyclic package dependencies, with an average of 40.133, this is a concern as it indicates some level of tight coupling at the package level, potentially complicating the modularity of the system.

The average number of direct package dependencies and dependent packages is roughly equal (9.556 and 9.444, respectively), which suggests a balanced but possibly intricate web of inter-package relationships, this could be a sign that packages are not as independent as they could be, potentially leading to difficulties in isolating changes and impacts across the codebase.

The number of transitively dependent packages stands at 43.933 on average, which indicates that the dependency chain can extend significantly beyond immediate connections. While this

Return to Index.

may point to a robust system of interactions, it also raises the risk of unintended side effects when changes are made in one package.

Overall, the metrics related to the packages suggest a complexity in the package structure with a noticeable number of cyclic dependencies and inter-package relationships, this reflects a scenario where packages may be too closely tied to each other, leading to challenges in maintaining and evolving the codebase due to the potential for cascading changes and the difficulty in isolating package responsibilities.

Review Log

Review feita por Luís Serrano, 60253:

A review está muito bem detalhada e concordo com todos os pontos apresentados!

Review feita por Tiago Santos, 63390:

A análise apresentada é bastante interessante, e estou de acordo com a avaliação dos elementos. Não tenho críticas a fazer.

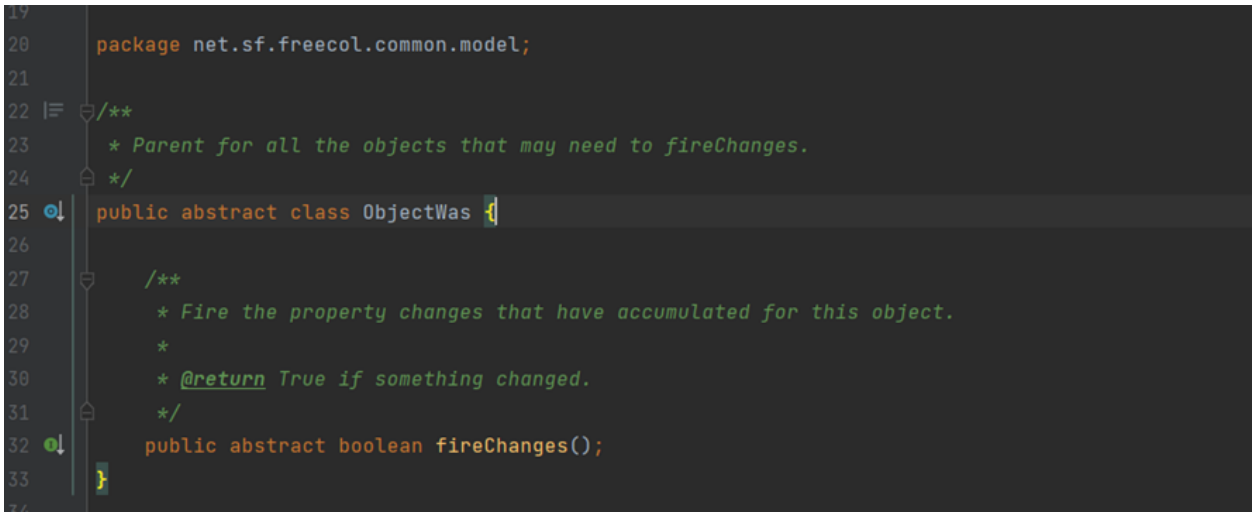
Return to Index.

Design Patterns

Author: André Santos (62331)

Template Method

The main idea behind the Template Method pattern is to create a structure that defines the sequence of steps in an algorithm, but leaves the detailed implementation of some of these steps to the subclasses. These steps can be represented as methods, and the pattern allows subclasses to provide their own implementations for these methods, keeping the overall structure of the algorithm consistent.



```java
package net.sf.freecol.common.model;

/**
 * Parent for all the objects that may need to fireChanges.
 */
public abstract class ObjectWas {

    /**
     * Fire the property changes that have accumulated for this object.
     *
     * @return True if something changed.
     */
    public abstract boolean fireChanges();
}
```

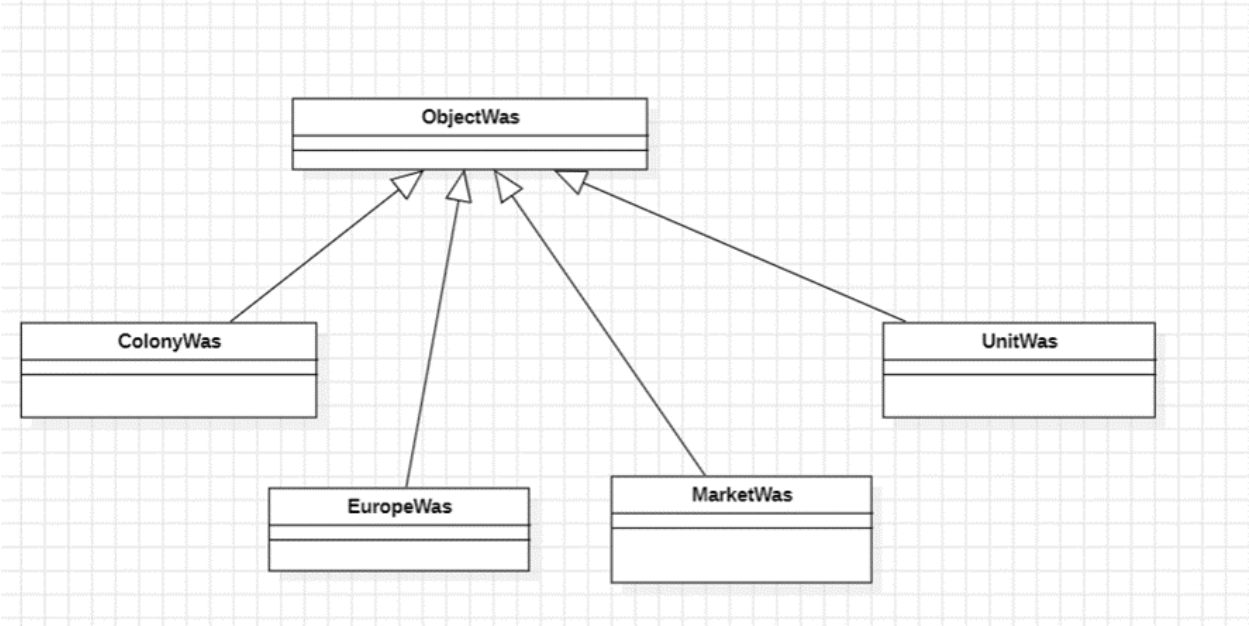
Path: net/sf/freecol/common/model/ObjectWas.java

Return to Index.

```java
/**
 * {@inheritDoc}
 */
public boolean fireChanges() {
    boolean ret = false;
    int newPopulation = colony.getUnitCount();
    if (newPopulation != population) {
        String pc = ColonyChangeEvent.POPULATION_CHANGE.toString();
        colony.firePropertyChange(pc, population, newPopulation);
        ret = true;
    }
    int newProductionBonus = colony.getProductionBonus();
    if (newProductionBonus != productionBonus) {
        String pc = ColonyChangeEvent.BONUS_CHANGE.toString();
        colony.firePropertyChange(pc, productionBonus,
            newProductionBonus);
        ret = true;
    }
    List<BuildableType> newBuildQueue = colony.getBuildQueue();
```

Path: net/sf/freecol/common/model/ColonyWas.java

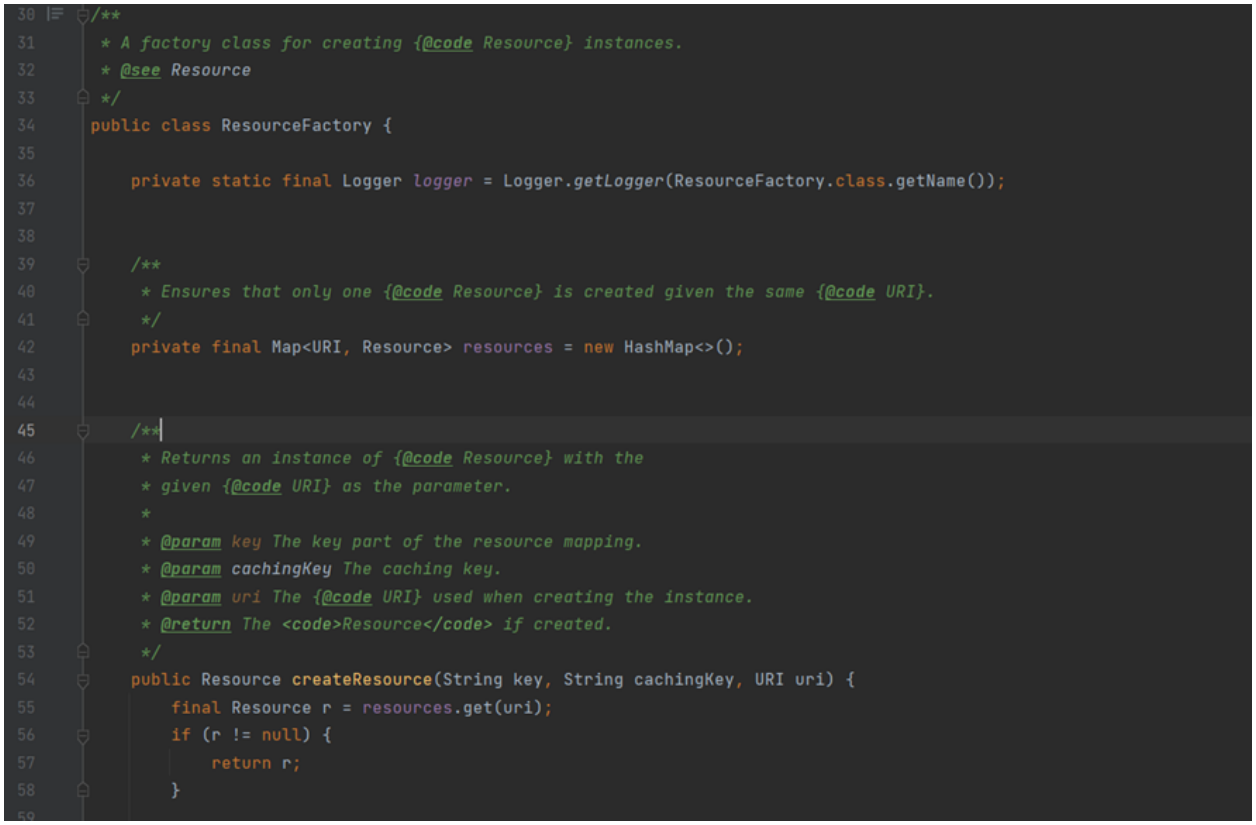
As we can see above, the "fireChanges()" method is redefined in the "ColonyWas" subclass, implementing it differently compared to other subclasses.



Return to Index.

Factory method

The "Factory Method" is a creation pattern that defines an interface for creating objects, but allows subclasses to decide which concrete class to instantiate. It can be identified when a class contains a creation method that returns objects from subclasses.



```java
/**
 * A factory class for creating {@code Resource} instances.
 * @see Resource
 */
public class ResourceFactory {

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


    /**
     * Ensures that only one {@code Resource} is created given the same {@code URI}.
     */
    private final Map<URI, Resource> resources = new HashMap<>();


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

Path: net/sf/freecol/common/resources/ResourceFactory.java

An example of this pattern is the "ResourceFactory" class, the purpose of which is to create "Resource" type factories.

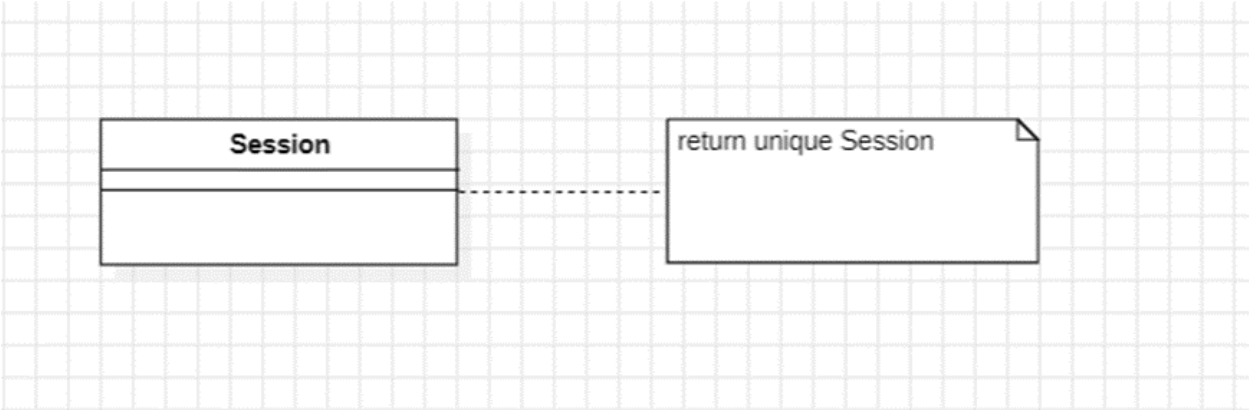
Return to Index.

Singleton Method

This pattern ensures that a class has only one instance and provides a global access point to that instance. It is often identified by the presence of a static method to access this single instance.

```java
public abstract class Session {

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

    /** A map of all active sessions. */
    private static final Map<String, Session> allSessions = new HashMap<>();

    /** The key to this session. */
    private String key;

    /** Has this session been completed? */
    private boolean completed = false;


    /**
     * Protected constructor, we only really instantiate specific types
     * of transactions.
     *
     * @param key A unique key to lookup this transaction with.
     */
    protected Session(String key) {
        Session s = getSession(key);
        if (s != null) {
            throw new IllegalArgumentException("Duplicate session: " + key
                                               + " -> " + s);
        }
        this.key = key;
        this.completed = false;
        logger.finest( msg: "Created session: " + key);
    }
```

Path: net/sf/freecol/server/model/Session.java



Return to Index.

We can tell that this really is a "Singleton Method" by these two parameters that are characteristic of this "design pattern":
Private Static Variable: The "private static final Map<String, Session> allSessions" variable is a private and static instance that holds all Session instances. Access is done statically through the class, which is typical of a "Singleton" implementation.

Private Constructor: The constructor of the "Session" class is defined as "protected", which means that it cannot be accessed directly by external code. This prevents new instances of the class from being created outside the class.

Code Report Review Log

Review feita por Tiago Santos, 63390:
Review 1 (4/11)
Concordo com os design patterns 2 e 3. No caso do primeiro, acho que está mal fundamentado, o que dificulta a perceção de se pode ou não ser considerado pattern.

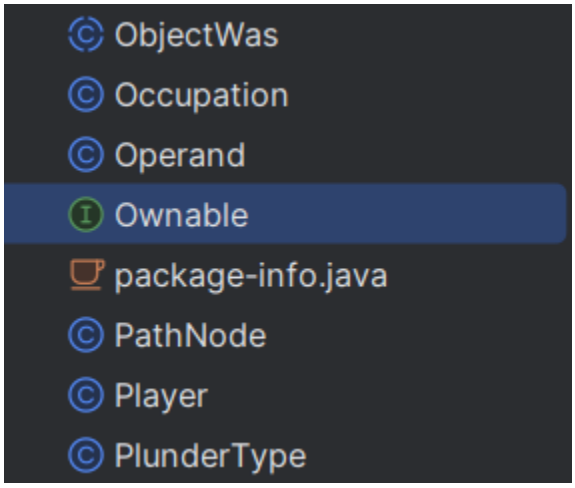
Review 2 (12/11)
Na minha opinião, o documento está bem fundamentado. Todos os problemas mencionados anteriormente foram resolvidos.

Return to Index.

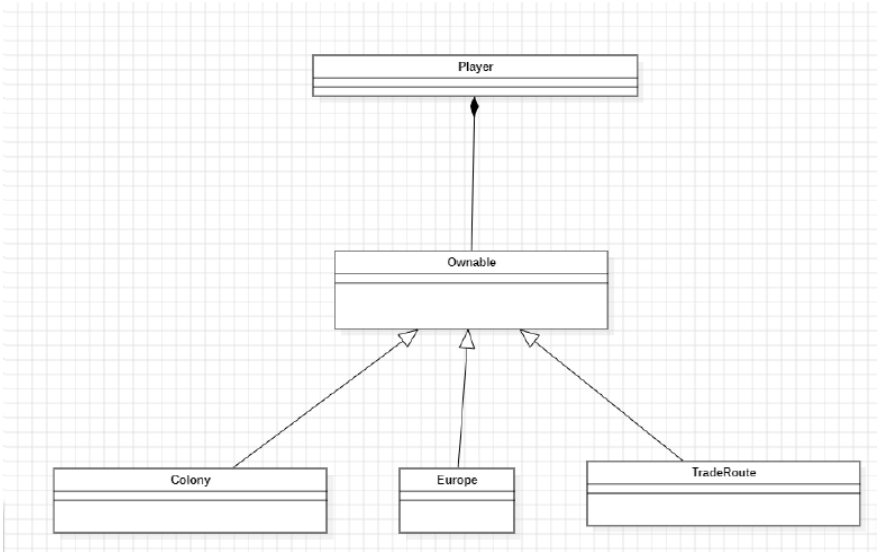
Author: António Palmeirim (63667)

Facade Pattern

Path: net/sf/freecol/common/model/Ownable.java



The design pattern that I consider here is the Facade design pattern, which is used so that the user can only access certain parts of a complex system, only the ones that are needed. The Ownable interface is the interface that the client accesses. Through the Ownable interface the client can set and get the owner of a certain object without having to access the object directly.



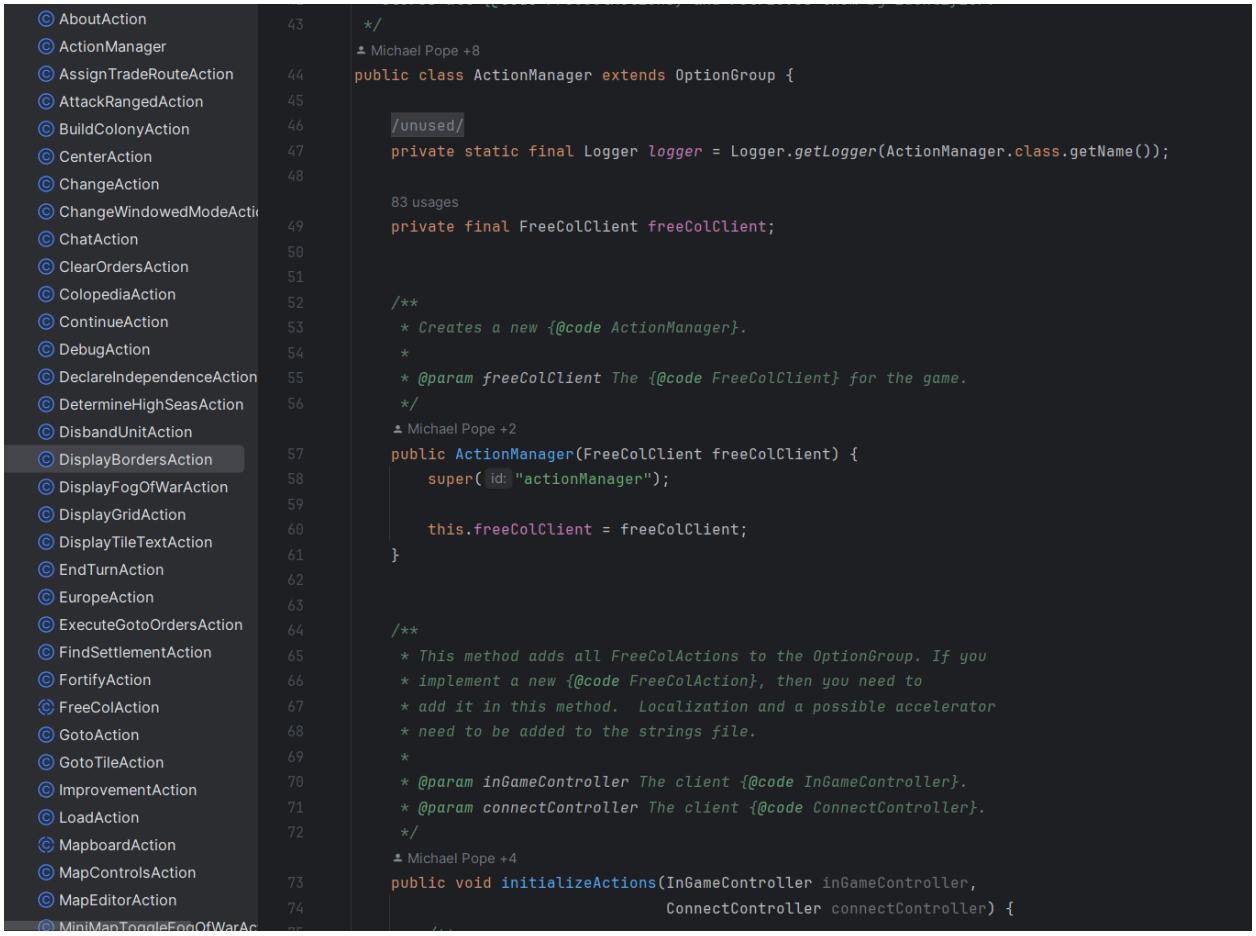
With the UML diagram I represent that the Ownable is the interface that the

Return to Index.

player accesses to access the objects, like Colony, Europe and TradeRoute. For simplicity reasons only these three objects are represented, but there are more classes that implement the Ownable interface.

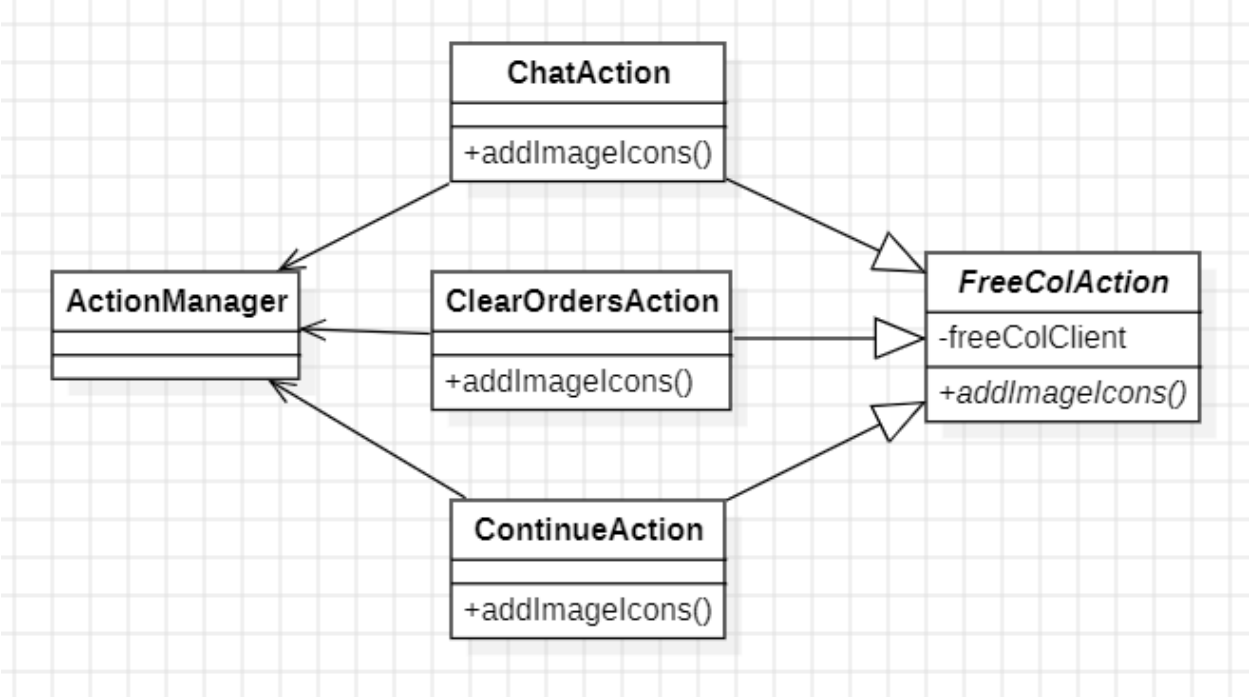
Command Pattern

Path: net/sf/freecol/client/gui/action/ActionManager.java



The design pattern found here is the command pattern. I consider this a command pattern in code because we have the ActionManager, which is the class that calls the different actions possible. In our case the different types of actions listed in the package (ChatAction, EndTurnAction, EuropeAction, etc.) are our various actions that ActionManager can use. These actions are then extending other abstract classes. These work as subclasses, as all of them extend FreeColAction. Compared to the editor and command example taught in class, the editor in our example is the ActionManager and the commands are the actions, while the main command abstract class is FreeColAction.

Return to Index.



The operations and attributes used are minimal and not all extensions of abstract classes are shown.

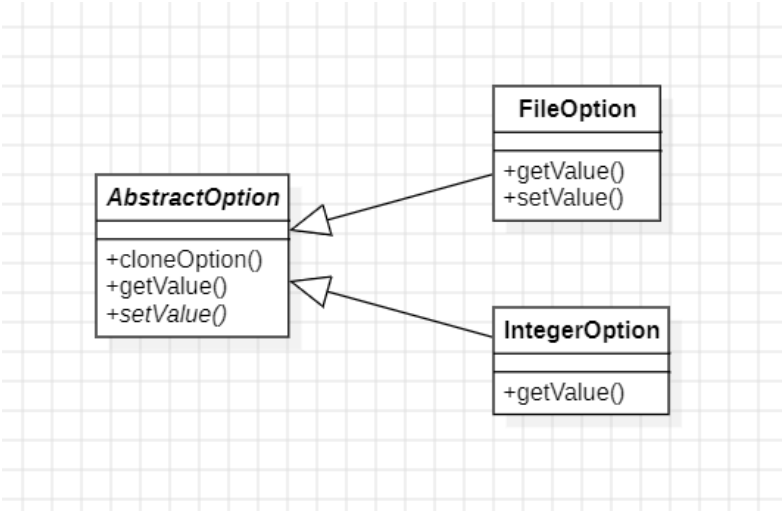
In the Class Diagram above we can see that the FreeColAction is the main abstract class while ChatAction, ClearOrdersAction and ContinueAction are the classes that extend, while the ActionManager is the class that calls the actions.

Return to Index.

Template Method

Path: net/sf/freecol/common/option/AbstractOption.java

Here we can find the Template Method being used. The “skeleton” is the AbstractOption class while other classes such as FileOption, TextOption, IntegerOption, etc. are the various branches of the skeleton. This is because the AbstractOption class has methods that are base for the rest, such as cloneOption() and getValue(), and other methods to be overridden or implemented by the subclasses, such as, setValue() and toString().



The operations and attributes used are minimal. In the class diagram we can see that the methods used in FileOption and IntegerOption are overriding the method in AbstractOption (getValue()) and are also implementing the abstract methods of AbstractOption (setValue()).

Return to Index.

Code Report Review Log

Reviewer: Duarte Inácio 62397

Review 1 (4/11):
Todos os design patterns parecem me ser adequados, bem estruturados e explicados, excepto o facade pattern já que o FreeColClientHolder teria de ser uma interface, por isso esse exemplo não me parece o mais adequado.

Return to Index.

Author: Duarte Inácio (62397)

Template Method Pattern

This design pattern allows you to define the structure of an algorithm in a base class, while the subclasses do the specific implementations of the methods of that interface. An example in freeCol's code of this design pattern is in the Animation (net/sf/freecol/client/gui/animation/Animation) base class and the subclasses (net/sf/freecol/client/gui/animation/UnitImageAnimation and net/sf/freecol/client/gui/animation/UnitMoveAnimation) since each of these subclasses implements the executeWithLabel() method differently in the different subclasses.

```
93      * @param paintCallback A callback to request that the animation area be
94      *     repainted.
95      */
96     public abstract void executeWithLabel(JLabel unitLabel,
97                                           Animations.Procedure paintCallback);
98 }
```

Animation Class

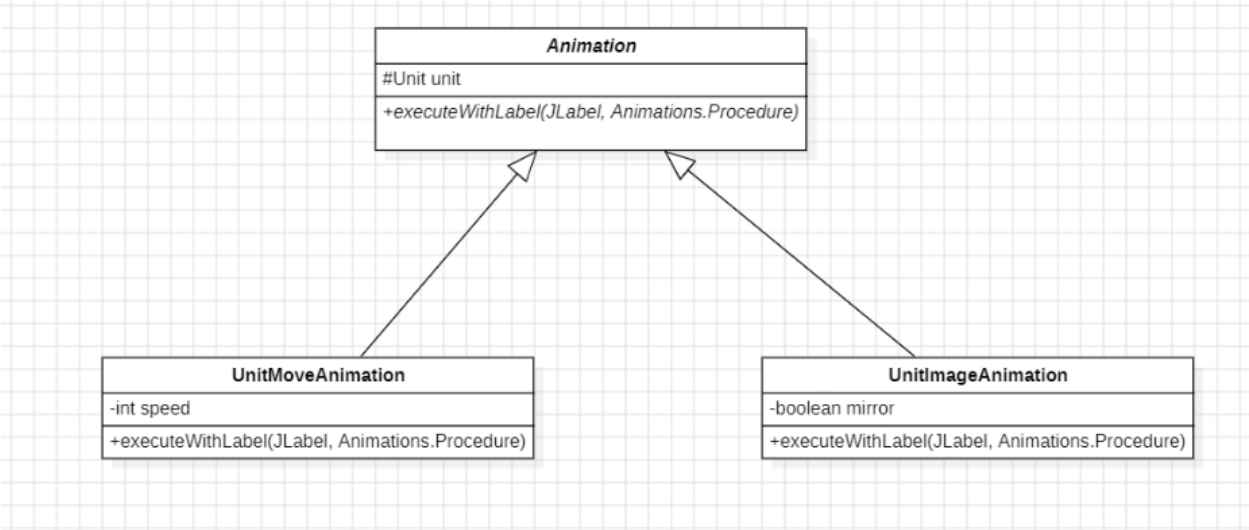
```
79      */
80     @Override
81     public void executeWithLabel(JLabel unitLabel,
82                                  Animations.Procedure paintCallback) {
83         final int movementRatio = (int)(Math.pow(2, this.speed + 1)
84             * this.scale);
85         final double xratio = ImageLibrary.TILE_SIZE.width
86             / (double)ImageLibrary.TILE_SIZE.height;
87         final Point srcPoint = this.points.get(0);
88         final Point dstPoint = this.points.get(1);
89         final int stepX = (int)(Math.signum(dstPoint.getX() - srcPoint.getX())
90             * xratio * movementRatio);
91         final int stepY = (int)(Math.signum(dstPoint.getY() - srcPoint.getY())
92             * movementRatio);
93
94         Point point = srcPoint;
95         long time = now(), dropFrames = 0;
96         while (!point.equals(dstPoint)) {
97             point.x += stepX;
98             point.y += stepY;
99             if ((stepX < 0 && point.x < dstPoint.x)
100                 || (stepX > 0 && point.x > dstPoint.x)) {
101                 point.x = dstPoint.x;
102             }
103             if ((stepY < 0 && point.y < dstPoint.y)
104                 || (stepY > 0 && point.y > dstPoint.y)) {
105                 point.y = dstPoint.y;
106             }
107             if (dropFrames <= 0) {
```

Return to Index.

UnitMoveAnimation subclasse

```
169 of @    public void executeWithLabel(JLabel unitLabel,
                                         Animations.Procedure paintCallback) {
171         final ImageIcon icon = (ImageIcon)unitLabel.getIcon();
172
173         // Step through the animation, chaning the image
174         for (AnimationEvent event : animation) {
175             long time = System.nanoTime();
176             if (event instanceof ImageAnimationEvent) {
177                 final ImageAnimationEvent ievent = (ImageAnimationEvent)event;
178                 Image image = ievent.getImage();
179                 if (mirror) {
180                     // FIXME: Add mirroring functionality to SimpleZippedAnimation
181                     image = createMirroredImage(image);
182                 }
183                 icon.setImage(image);
184                 paintCallback.execute(); // paint now
185
186                 // Time accounting
187                 time = ievent.getDurationInMs()
188                     - (System.nanoTime() - time) / 1000000;
189                 if (time > 0) Utils.delay(time,  warning: "Animation delayed.");
190             }
191         }
192     }
```

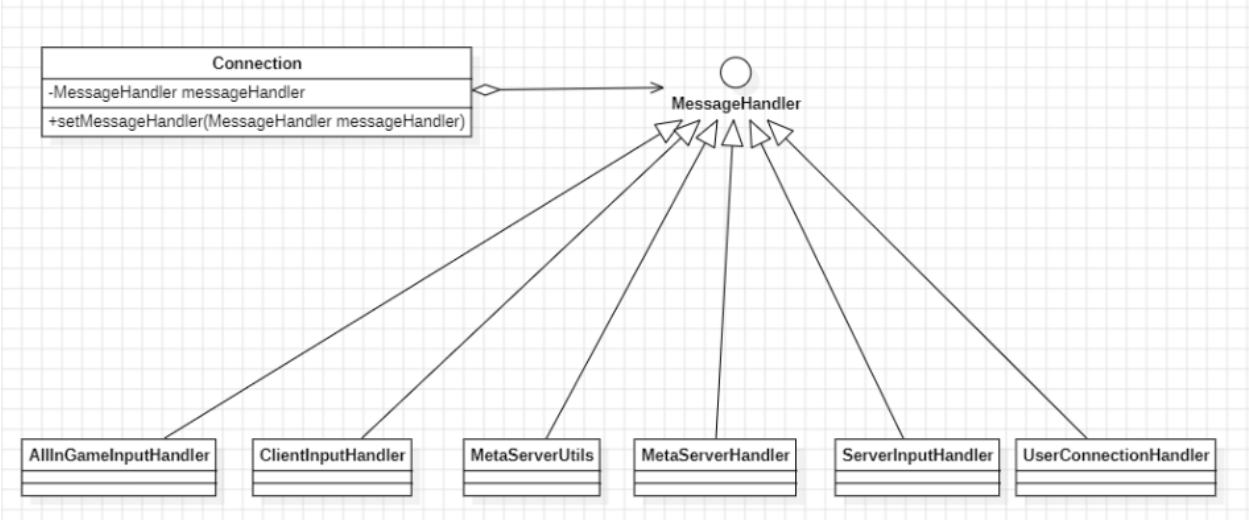
UnitImageAnimation subclasse



Return to Index.

State Pattern

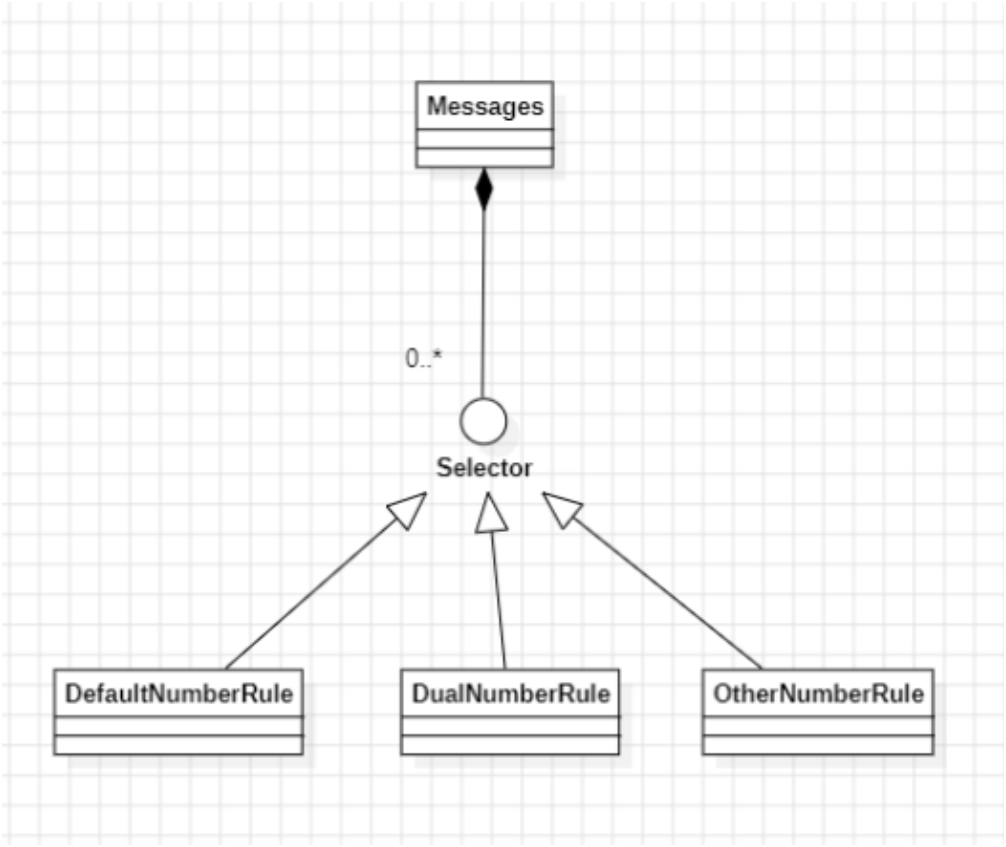
This design pattern is used to model objects that can have different states and allows these objects to change state during program execution. An example of this pattern is the MessageHandler interface (net/sf/freecol/common/networking/MessageHandler) and the different states, namely AllInGameInputHandler(net/sf/freecol/server/ai/AIInGameInputHandler), ClientInputHandler(net/sf/freecol/client/control/ClientInputHandler), MetaServerUtils(net/sf/freecol/common/metaserver/MetaServerUtils), MetaServerHandler(net/sf/freecol/metaserver/MetaServerHandler), ServerInputHandler(net/sf/freecol/server/control/ServerInputHandler) and UserConnectionHandler(net/sf/freecol/server/control/UserConnectionHandler). This messageHandler state is modified in Connection (net/sf/freecol/common/networking/Connection).



Return to Index.

Facade Pattern

This design pattern provides a unitary interface for a set of classes, simplifying interaction with the subsystem, hiding internal complexity and providing a single input to the client. An example of this pattern is the Selector(net/sf/freecol/common/i18n/Selector) interface, which is implemented by the following classes: DefaultNumberRule (net/sf/freecol/common/i18n/DefaultNumberRule), DualNumberRule (net/sf/freecol/common/i18n/DualNumberRule) and OtherNumberRule (net/sf/freecol/common/i18n/OtherNumberRule), for example. Then in the Messages(net/sf/freecol/common/i18n/Messages) class you have 0 or more Selectors.



Return to Index.

Code Report Review Log

Reviewer: Antonio Palmeirim 63667

Review 1 (4/11):
Os design patterns encontrados parecem me aceitáveis e explícitos, exceto o último caso. No use case diagram no final do Facade Pattern, poderia pôr as diferentes operações que são escondidas ao utilizador, e que são só acessíveis a partir da interface que liga o utilizador e o sistema.

Review 2 (19/11):
Notei que modificou o facade pattern. Continua correto, e no meu último review referi a inexistencia dos métodos no UML diagram, mas com o texto e a explicação da para perceber bem o pattern e a sua utilidade.

Reviewer: Rafael Tavares 60608

Review 1 (4/11):
Acho que todos os design patterns estão bem ilustrados e descritos, é possível compreender a forma como estão implementados e a sua implicação na codebase.

Review 2 (25/11):
A informação continua sólida, no entanto penso que em termos do documento poderia ser melhorado para melhor interpretar o que transmite.

Return to Index.

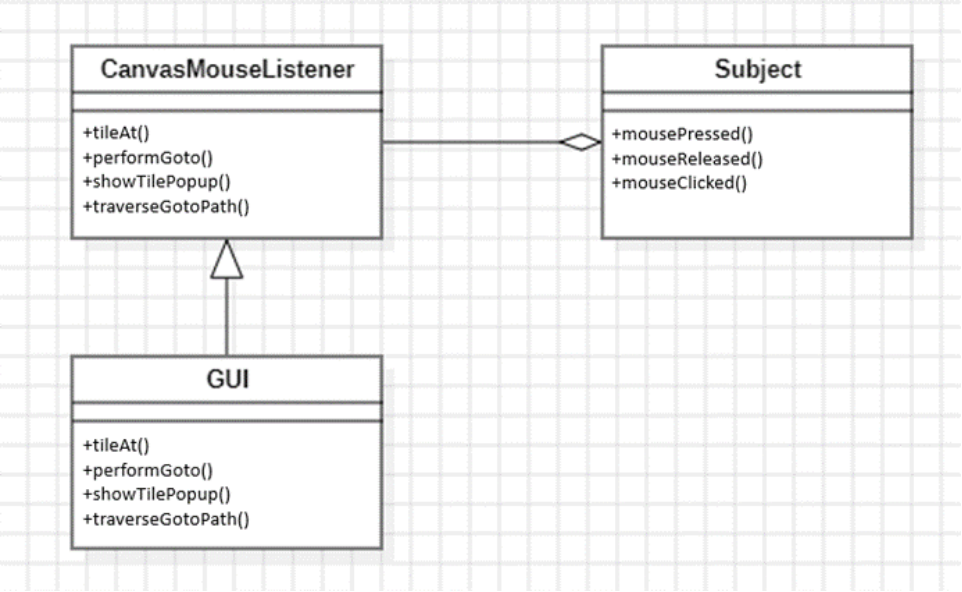
Author: Tiago Santos (63390)

Observer Pattern

Path: net.sf.freecol/client/gui/CanvasMouseListener.java

```java
/**
 * Listens to mouse buttons being pressed at the level of the Canvas.
 */
Mike Pope +6
public final class CanvasMouseListener extends FreeColClientHolder
    implements MouseListener {

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


    /**
     * Create a new mouse listener.
     *
     * @param freeColClient The enclosing {@code FreeColClient}.
     */
    Mike Pope +1
    public CanvasMouseListener(FreeColClient freeColClient) { super(freeColClient); }
```

The CanvasMouseListener class acts as a mouse event observer, responding dynamically to user interactions in the Canvas area. The one-to-many relationship between the Canvas component and the CanvasMouseListener reflects the corresponding design pattern, allowing other components to be notified and updated automatically when mouse events occur during the game.



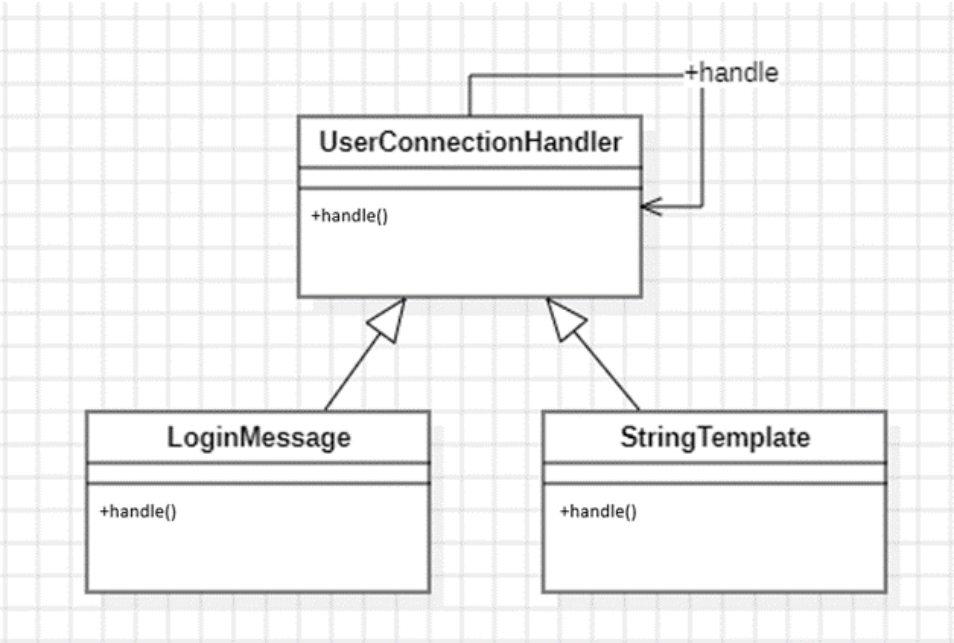
Return to Index.

Chain of Responsibility Pattern

Path: net.sf.freecol/server/control/UserConnectionHandler.java

```java
/**
 * {@inheritDoc}
 */
Mike Pope
public Message handle(Connection connection, Message message)
    throws FreeColException {
    final FreeColServer freeColServer = getFreeColServer();
    ChangeSet cs = null;
    switch (message.getType()) {
    case DisconnectMessage.TAG:
        break;
    case LoginMessage.TAG:
        cs = ((LoginMessage)message).loginHandler(freeColServer, connection);
        break;
    default:
        cs = ChangeSet.clientError((Player)null,
            StringTemplate.template( value: "server.couldNotLogin"));
        break;
    }
    return (cs == null) ? null
        : cs.build(freeColServer.getPlayer(connection));
}
```

This design pattern manages messages for new connections, using the "handle" method to handle different types of messages (such as LoginMessage). This provides flexibility and scalability in the server's message processing.



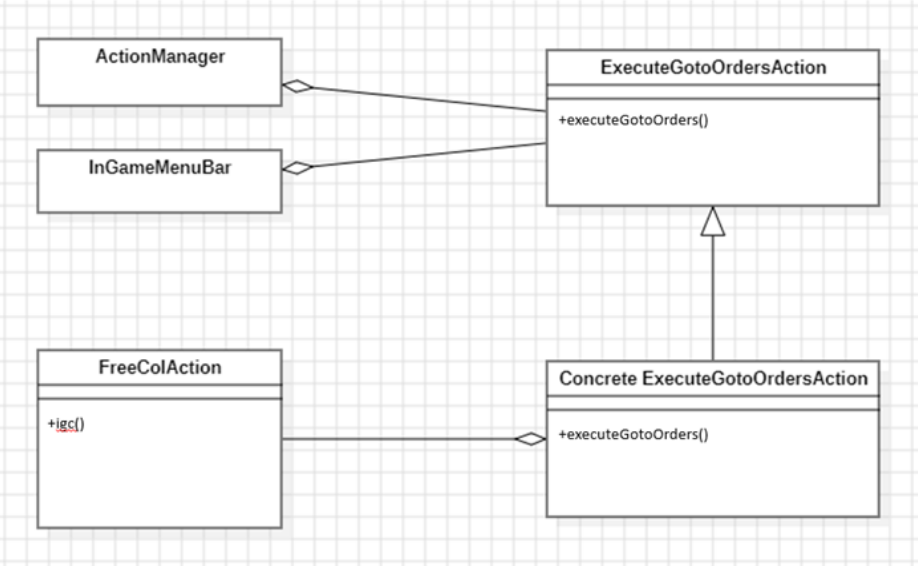
Return to Index.

Command Pattern

Path: net.sf.freecol/client/gui/action/ExecuteGotoOrdersAction.java

```java
/**
 * An action for executing goto orders immediately.
 */
public class ExecuteGotoOrdersAction extends MapboardAction {

    public static final String id = "executeGotoOrdersAction";


    /**
     * Creates a new {@code ExecuteGotoOrdersAction}.
     *
     * @param freeColClient The {@code FreeColClient} for the game.
     */
    public ExecuteGotoOrdersAction(FreeColClient freeColClient) {
        super(freeColClient, id);
    }


    // Interface ActionListener

    /**
     * {@inheritDoc}
     */
    @Override
    public void actionPerformed(ActionEvent ae) {
        igc().executeGotoOrders();
    }
}
```

The Command design pattern stores a command as an object, so it can be used in several classes. It offers flexibility by parameterizing clients with different commands. This makes the code more flexible and reusable.



Return to Index.

Code Report Review Log

Reviewer: António Palmeirim 63667

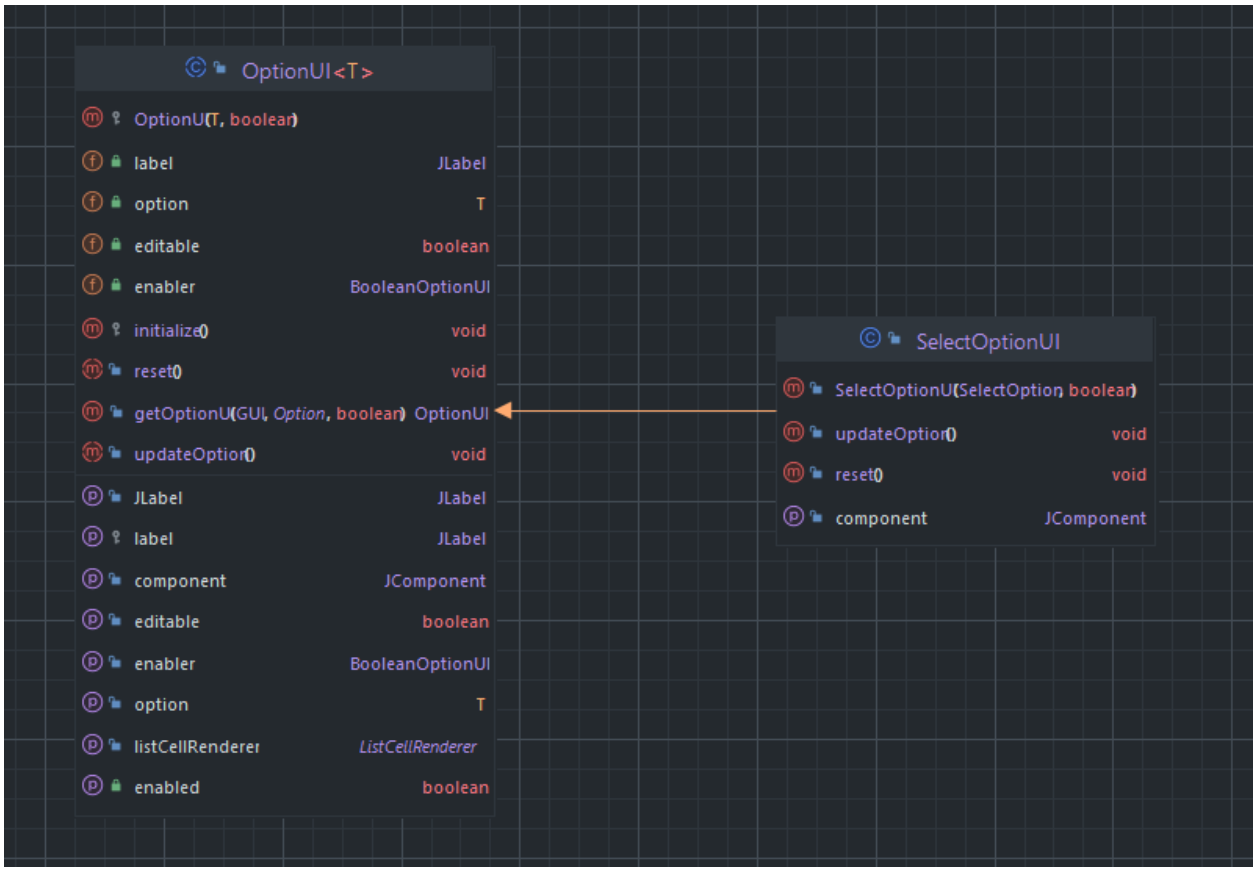
Review 1 (23/11):
Os patterns encontrados parecem me corretos. O documento está bem estruturado
e todos os patterns estão explicados de forma explícita e sucinta. De resto, nada a apontar!

Return to Index.

Author: Luís Serrano (60253)

Strategy Pattern

Path: net/sf/freecol/client/gui/option

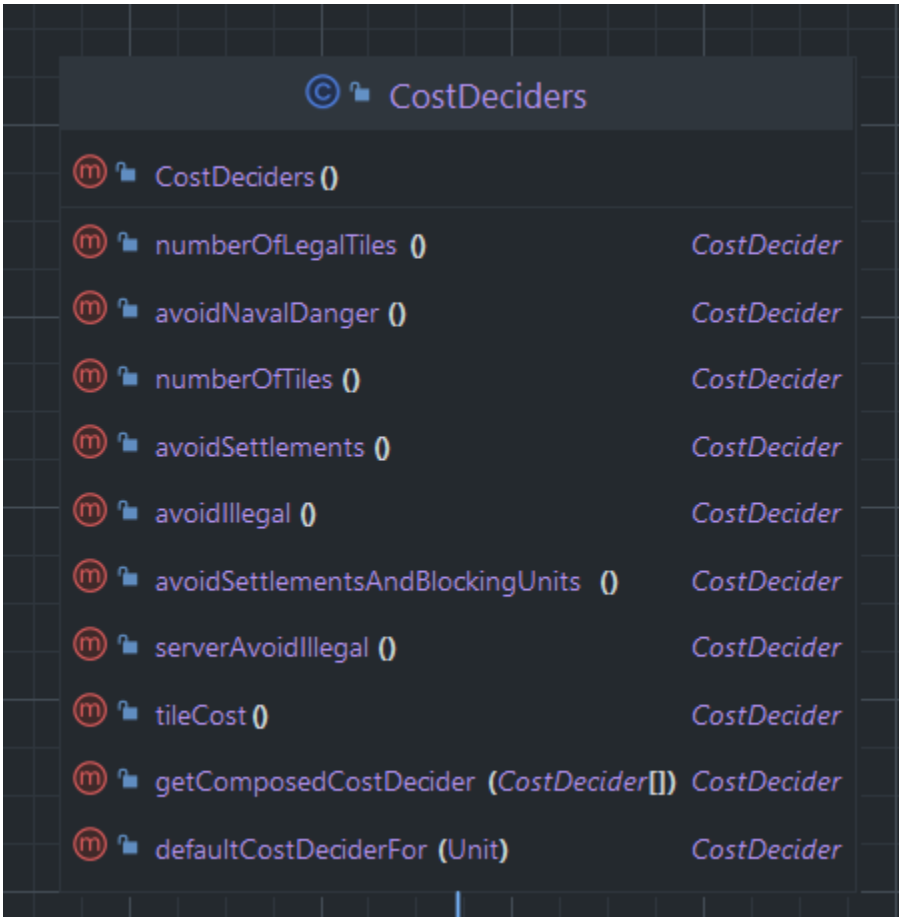


These classes represent a Strategy Pat tern, as SelectOptionUI is a concrete class that implements the OptionUI class, indicating that it provides a specific implementation for dealing with selection options.

Return to Index.

Composite Pattern

Path: net/sf/freecol/common/model/pathfinding/CostDeciders.java



This class has a Composite Pattern. We can conclude this because the method getComposedCostDecider (CostDecider[]) suggests that this class can compose several objects of the Cost Decider type to create a more complex CostDecider. This means that CostDeciders can create a new CostDecider that combines the behavior of other CostDecider objects. This ability to compose objects in a hierarchical way is a feature of the Composite Pattern.

Return to Index.

Return to Index.

State Pattern

Path: net/sf/freecol/common/model/Stance.java

The diagram represents a class that presents a State Pattern, since it encapsulates different states to manage state transitions (badTransition(Stance)) and obtain state-specific behaviors (badStance()). In addition, the presence of static methods such as values() and valueOf(String) for state management also supports this pattern.

Return to Index.

Review log

Reviewer: Antonio Palmeirim

Review 1 (15/11):

Os design patterns estão bem explícitos e explicados, entretanto só meteria uns class diagrams para todos os patterns que encontraste, para perceber melhor o funcionamento dos designs com o código do FreeCol.

Return to Index.

Reviewer: André Santos 62331

Review 1 (15/11):

Documento bem estruturado e argumentos organizados de forma coesa e clara. Concordo com a review do Antonio no intuito em que faltam class diagrams para os design patterns que foram encontrados. De resto nada a dizer.

Return to Index.

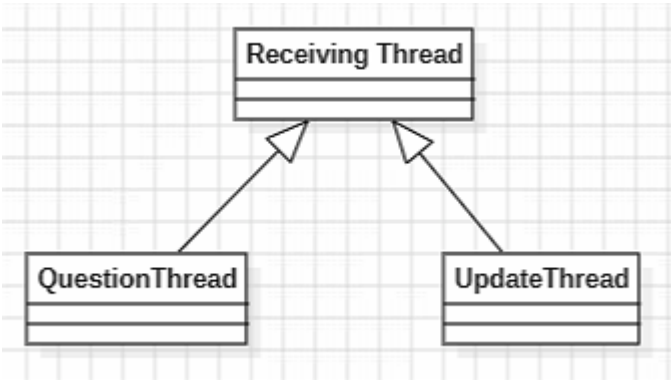
Author: Rafael Tavares (60608)

Template Method Pattern

Path: src/net/sf/freecol/common/networking/ReceivingThread.java

```
37  /**T
38   * The thread that checks for incoming messages.
39   */
40  final class ReceivingThread extends Thread {
41
42      private static final Logger logger = Logger.getLogger(ReceivingThread.class.getName());
43
44      /** A class to handle questions. */
45  >   private static class QuestionThread extends Thread {…
98
99  >   private static class UpdateThread extends Thread {…
145
```

In the class ReceivingThread we can see two other classes being defined inside of it, which change very little in terms of implementation, slightly differing, which indicates a sign of the Template Method Pattern. Not only here, but, there are other classes extending the Thread class which are also very similar.



Return to Index.

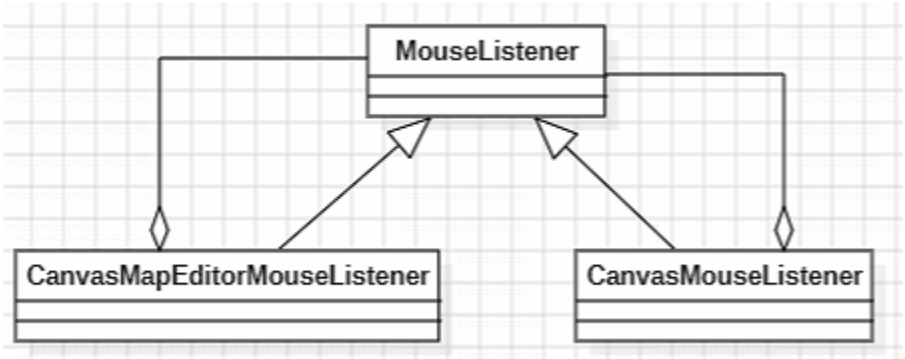
Composite Pattern

Path: src/net/sf/freecol/client/gui/CanvasMapEditorMouseListener.java
and
src/net/sf/freecol/client/gui/CanvasMouseListener.java

```java
/**
 * Listens to mouse buttons being pressed at the level of the Canvas.
 */
public final class CanvasMouseListener extends FreeColClientHolder
    implements MouseListener {
```

```java
/**
 * Listens to the mouse being moved at the level of the Canvas.
 */
public final class CanvasMapEditorMouseListener extends FreeColClientHolder
    implements MouseListener, MouseMotionListener {
```

By examining the relations between MouseListener with CanvasMapEditorMouseListener and CanvasMouseListener, we can see their dependency through the implementation of the same interface, which can indicate the Composite Pattern. Moreover, we can conclude our rationale by understanding that both CanvasMapEditorMouseListener and CanvasMouseListener are MouseListeners themselves.



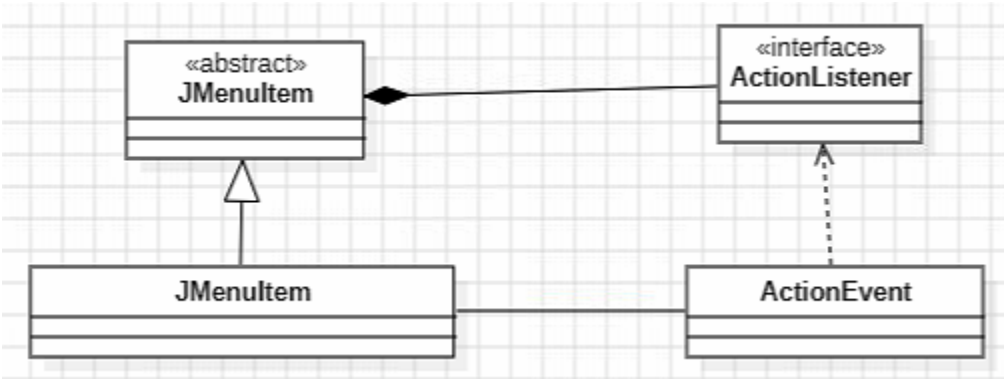
Return to Index.

Observer Pattern

Path: src/net/sf/freecol/client/gui/TilePopup.java

```
        ji.addActionListener((ActionEvent ae) -> {
                igc.changeState(activeUnit, Unit.UnitState.ACTIVE);
            });
        add(ji);
```

The method addActionListener was a very strong indication of an Observer pattern, which was later confirmed by examining the code. We can see here, an observer being added to an object, which when triggered will produce a state change.



Review Log

Review Luís Serrano (60253) (4/11):

O documento é muito fácil de interpretar e apelativo, percebe-se bem cada exemplo utilizado e está bem explicado.

Review Antonio Palmeirim (63667) (4/11):

Os patterns estão bem definidos e bem explicados, a única coisa que mudava e por as diferentes operações nos class diagrams para perceber que operações e que são partilhadas entre as classes.

Return to Index.

Use Case Diagrams

Colonies Use Case Diagram

Author: André Santos (62331)

The following use cases encapsulates the pivotal moments and a few basic mechanics that players face when managing their colony. From the critical task of selecting a suitable site for a new colony to intricacies involving government efficiency, colony buildings, and unit construction, players must take into consideration numerous factors to thrive in the landscape of FreeCol.

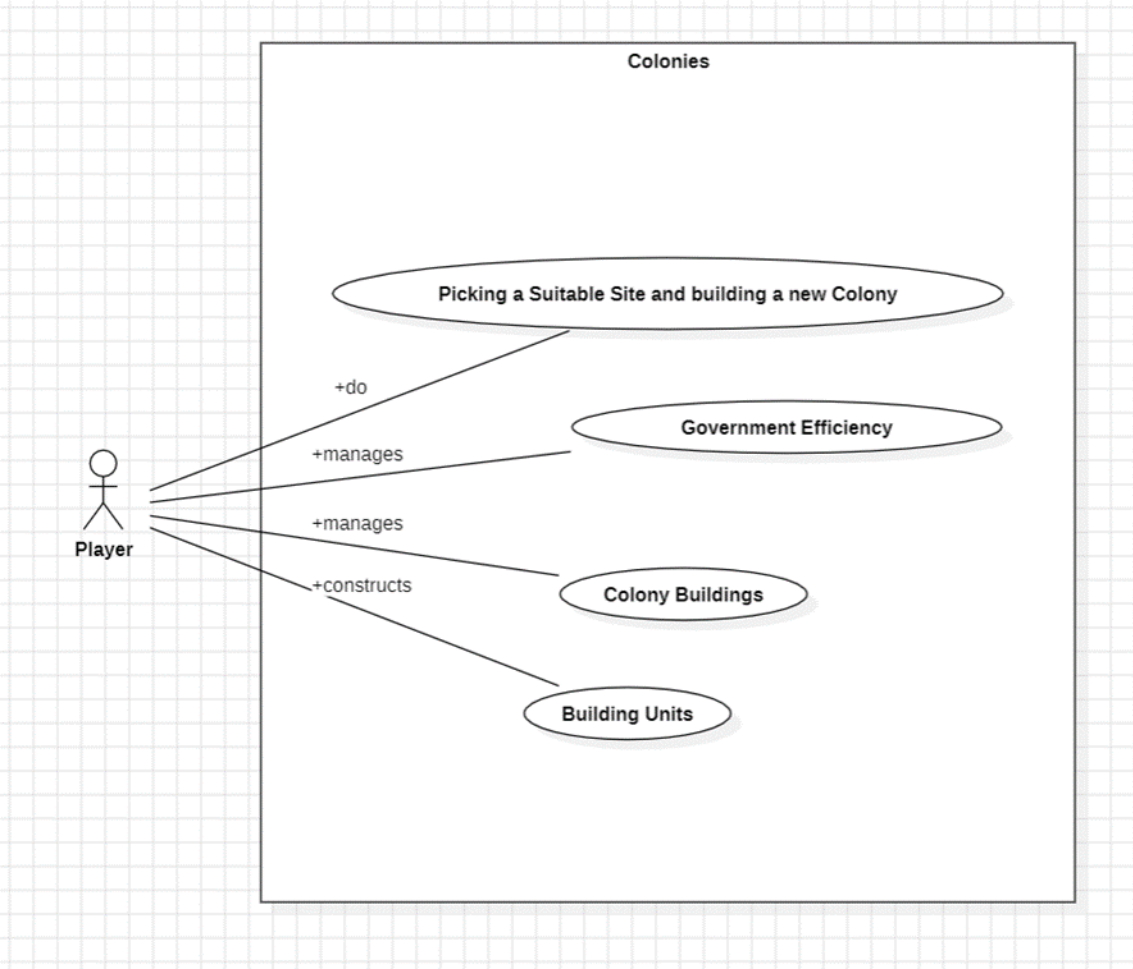


Figure 1- *Colonies Use Case Diagram*

Return to Index.

Use Case 1

Name: Picking a Suitable Site and building a new Colony

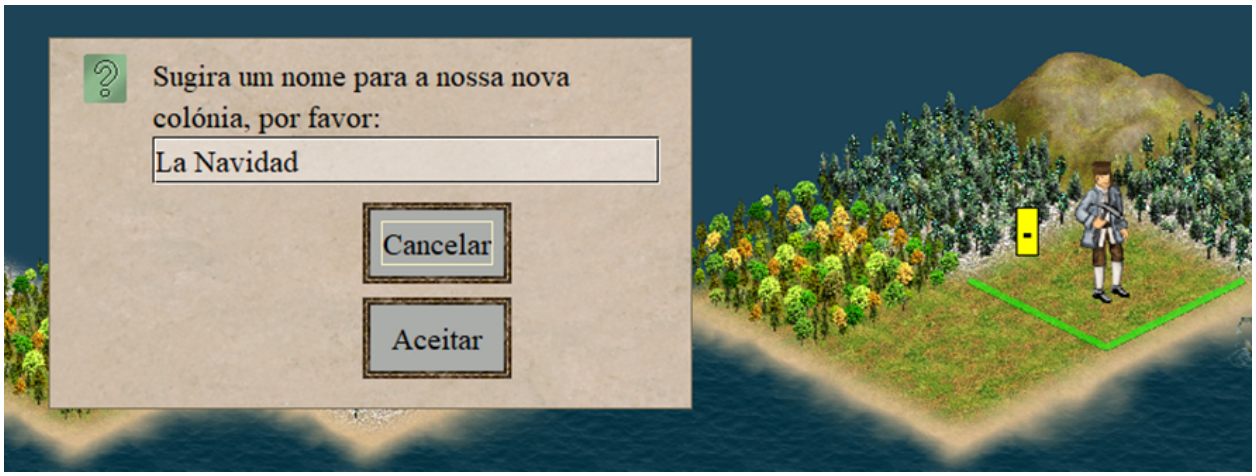


***Figure 2**- Before laying the foundation for the new Colony*

***Figure 3** -Information about an adjacent tile to the colony*

Description: The player evaluates and selects an appropriate location for a new colony, considering factors such as the colony tile, adjacent tiles, terrain types, potential for improvement and lays the foundations for his new Colony in selected tile.

Primary Actor: Player

Secondary Actors: None

Return to Index.

Use Case 2

Name: Manage Government Efficiency

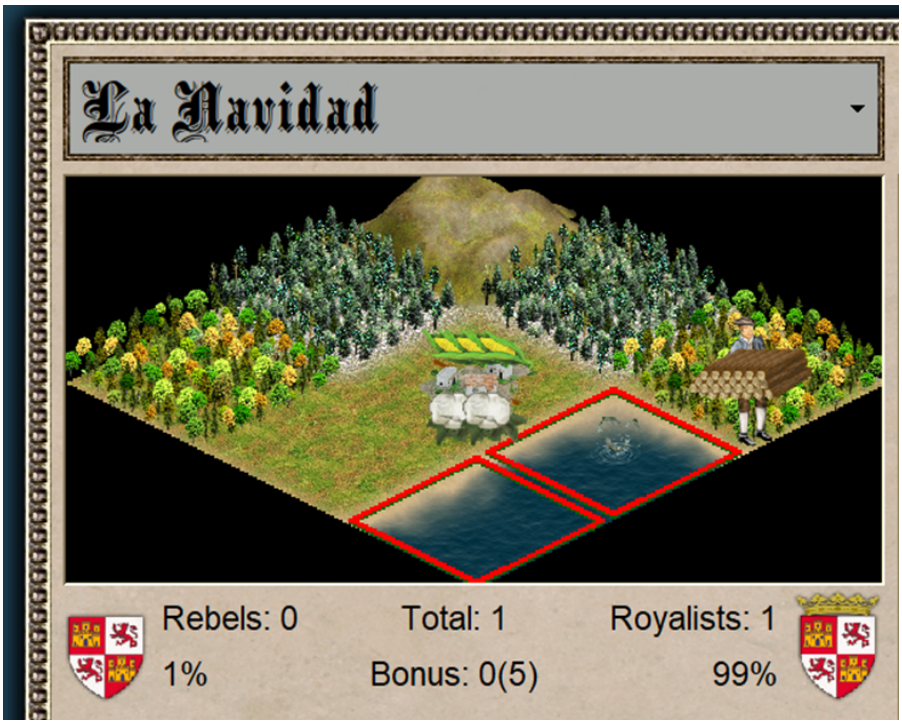


***Figure 4**- The relation between the number of rebels and royalists*

Description: The player takes actions to influence the efficiency of the local government, including monitoring support for the Sons of Liberty and taking measures to increase or maintain it.

Primary Actor: Player

Secondary Actors: None

Return to Index.

Use Case 3

Name: Colony Buildings

Figure 5- *Currently building the new docks for the colony*

Figure 6- *The player has two units working as carpenters in order to obtain more hammers in fewer rounds*

Description: The player manages, upgrades, and builds new various buildings within the colony, considering factors such as population size, available resources granting the player boosts in production, higher storage limits, the ability to train units, etc…The player can also manage the units in each building producing more or less products according to their needs.

Primary Actor: Player

Secondary Actors: None

Return to Index.

Use Case 4

Name: Building Units

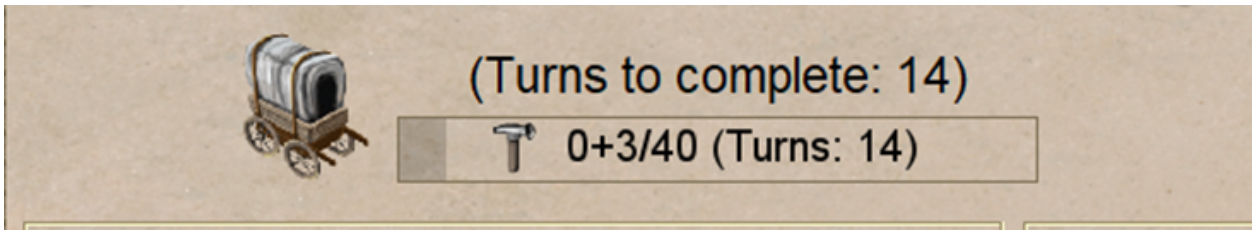


Figure 7*- Currently building a Wagon Train unit which requires 40 hammers*

Description: The player produces hammers, tools, and other resources needed to construct new units also taking into consideration the conversion of raw materials into finished products.

Primary Actor: Player

Secondary Actors: None

Return to Index.

Code Report Review Log

Review feita por Tiago Santos, 63390: (5/11)
Na minha opinião, o caso de uso está bem apresentado, porém o diagrama podia estar mais desenvolvido.

Review Duarte Inácio, 62397: (5/11)
Tenta colocar imagens do jogo para se entender melhor que parte do jogo estás a abordar

Review Duarte Inácio, 62397: (19/11)
Falta-te os IDs em cada um dos use cases.

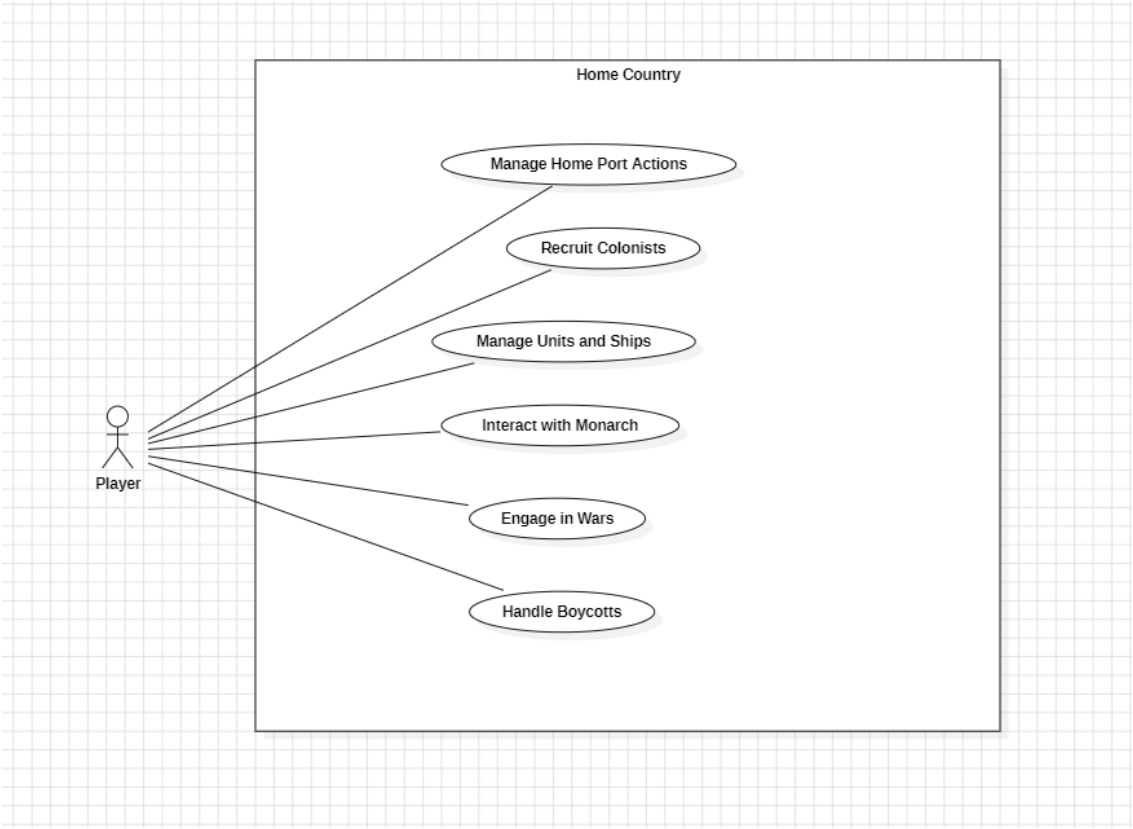
Return to Index.

Author: António Palmeirim (63667)

The use case diagrams that I will be covering are the ones associated with the Home Country functionality of the game and the Unit functionality.

Home Country Use Case Diagram

To give context, the Home Country that is associated with the player when playing the game is a European colonial and monarchy power. The player is associated with one out of eight of the nations in the game: Spain, France, Netherlands, England, Portugal, Sweden, Denmark and Russia. Each of these nations have different abilities and different starting units depending on the respective nation that is associated with the player.



In the use case diagram above, we have the various interactions the player has with the Home Country that is associated when initially starting the FreeCol adventure. Here are the various Use Cases and their respective description:

Return to Index.

Use Case: Manage Home Port Actions
Description: Allows the player to trade goods, train, recruit and buy units in the Home Port. Also involves repairing damaged ships at the Home Port if no Drydock is built in colonies.
Primary Actor: Player

Use Case: Recruit Colonists
Description: Involves recruiting colonists from the recruitment list by offering gold incentives. Players can also train colonists at the Royal Academy for emigration to the New World.
Primary Actor: Player

Use Case: Manage Units and Ships
Description: Allows players to purchase, build and manage ships and artillery in the Home Port. Includes conditions for building units in colonies
Primary Actor: Player

Use Case: Interact with Monarch
Description: Represents interactions with the Monarch, including decisions on taxes, responses to boycotts, declarations of war and the addition of units to the Royal Expeditionary Force.
Primary Actor: Player

Use Case: Handle Boycotts
Description: Covers the actions related to handling boycotts, such as refusing taxes, staging protests, paying tax arrears, and the availability of Custom Houses.
Primary Actor: Player

Use Case: Engage in Wars
Description: Involves declaring war on other nations and dealing with the consequences, including changes in relations, possible Mercenary offers and the impact on taxes.
Primary Actor: Player

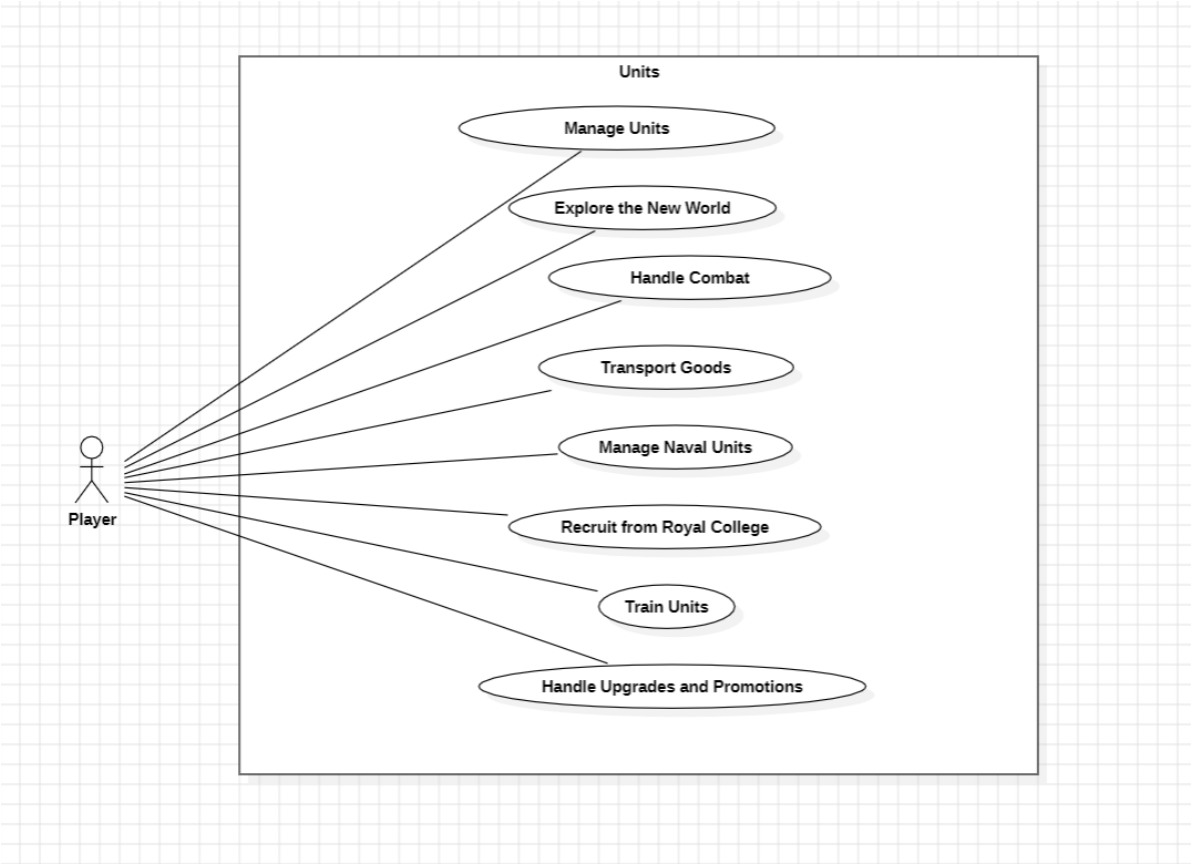
Drydock: An upgraded dock used for repairing damaged ships.

Units Use Case Diagram

The Units in the game FreeCol are used as non-playable characters that have certain skills dependent on where they originate from and can be beneficiary to the player and their

Return to Index.

colony. There are multiple types of units available to the player depending on which continent the player chooses to start at.



In the use case diagram above, we have the various interactions the player has with the Units. Here are the various Use Cases and their respective description:

Use Case: Manage Units
Description: Allows the player to recruit, train and upgrade units. The player can also equip units with tools, horses, muskets or a bible to enhance their abilities.
Primary Actor: Player

Use Case: Explore the New World
Description: Involves sending units, such as Scouts, to explore the New World. The player can also encounter and handle the dangers of the New World.
Primary Actor: Player
Return to Index.

Use Case: Handle Combat
Description: The player has access to the combat system, including battles between different units and the outcomes. They also can engage in combat with units from other players or with native units.
Primary Actor: Player

Use Case: Transport Goods
Description: Involves using various modes of transport, to transport goods, treasures and units between colonies and trade with native settlements. The player must consider the limitations on transport capacity and potential dangers.
Primary Actor: Player

Use Case: Manage Naval Units
Description: The player can deploy and manage naval units. The player can also engage in naval battles, capture enemy goods and explore coastal colonies.
Primary Actor: Player

Use Case: Recruit Units from Royal College
Description: The process of recruiting units directly from the Royal College in Europe by spending gold.
Primary Actor: Player

Use Case: Train Units
Description: Involves training units in various skills and professions by placing them in the Schoolhouse, College or University. Units can learn new skills, and the training duration is affected by the colony's production bonus or penalty.
Primary Actor: Player

Use Case: Handle Upgrade and Promotions
Description: Covers the process of upgrading units based on experience or promotions gained through battles.
Primary Actor: Player

Return to Index.

Code Report Review Log

Reviewer: Luís Serrano 60253

Review 1 (5/11)
Cada use case está muito bem descrito, o que torna muito simples a compreensão do diagrama. Não mudaria nada, acho que será este tipo de descrição o ideal.

Reviewer: Duarte Inácio 62397

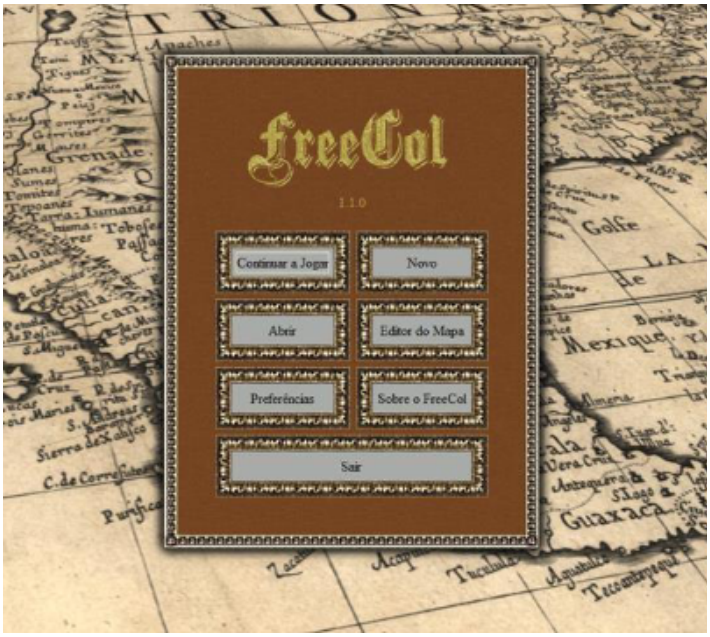
Review 1 (22/11)
Os 2 use cases diagrams parecem-me bem explicados. Nada a apontar!

Reviewer: André Santos, 62331

Review 1 (24/11)
Agora sim já está de acordo com o que o professor pediu, nada a acrescentar!

Return to Index.

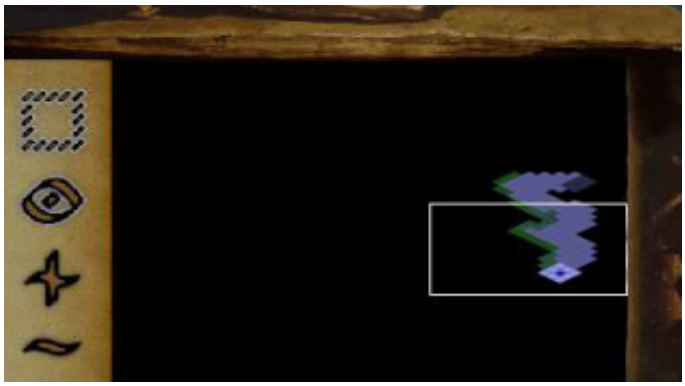
Author: Duarte Inácio (62397)



The use case diagram I decided to implement was that of the main menu, namely where we can open different tabs (continue playing, new, open, map editor, preferences, about freeCol and exit) and where in each one the behavior is different.

In the case of Continue playing we can see that there is a menu bar with different options, namely: Game, View, Orders, Reports, Colopedia. Each of these options opens a tab with different functions.

Continuing in the Continue playing option, we have a mini-map where we can modify it with the 4 different options provided.

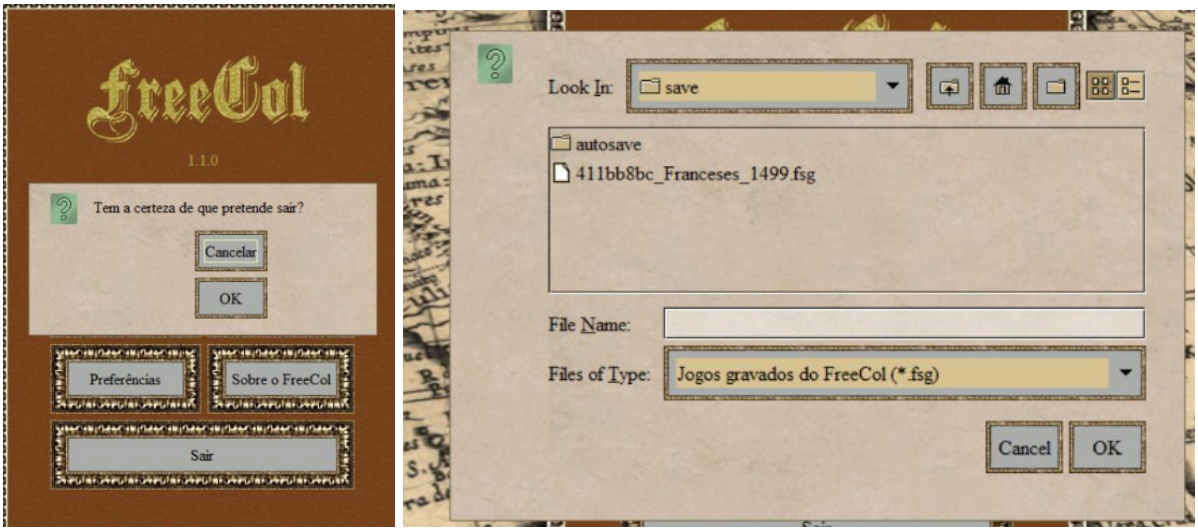


Return to Index.

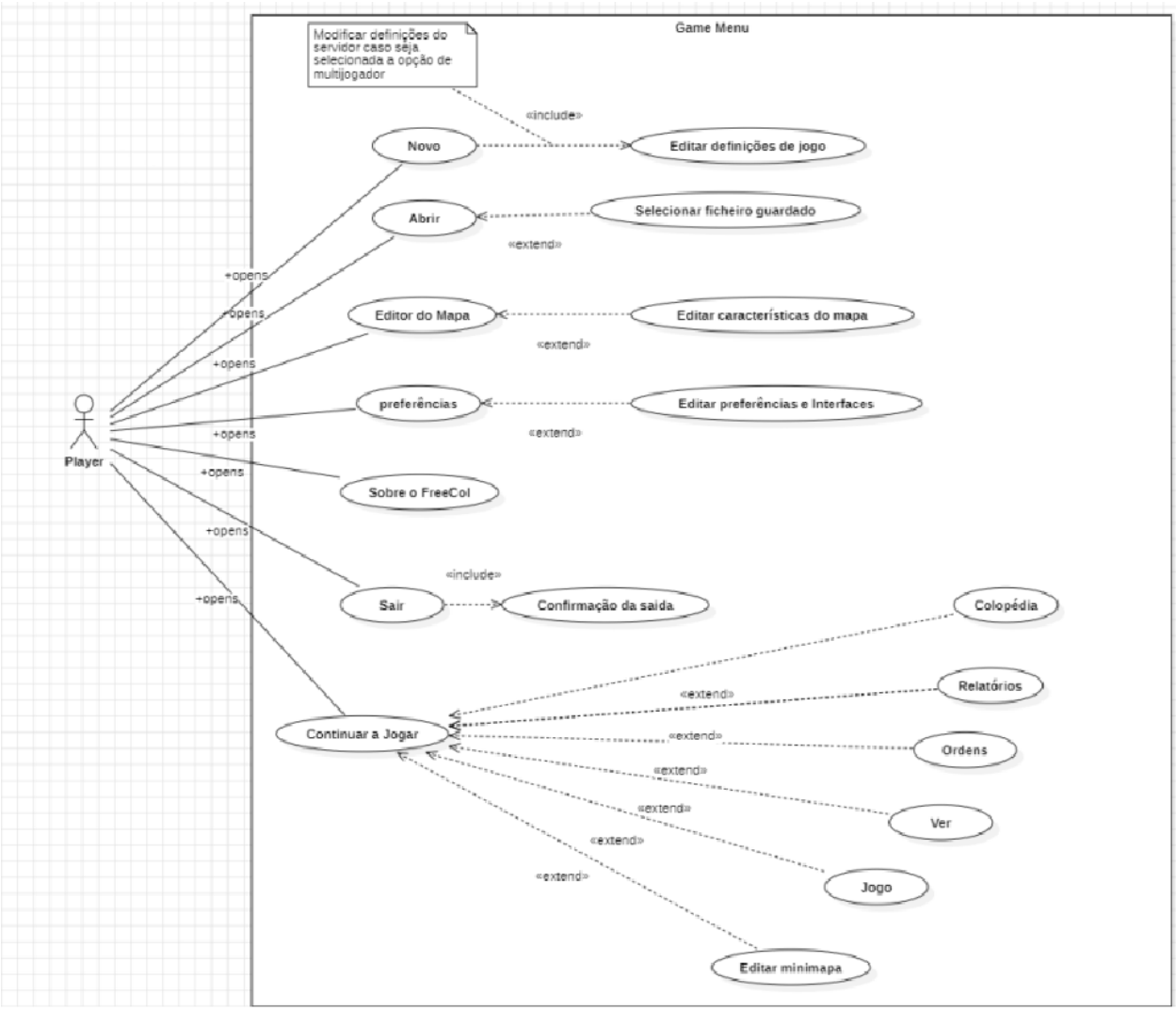
On the other hand, we have the option of starting a new game, where we can choose between a singleplayer or multiplayer game and modify the game settings and even the colony settings.

On the other hand, there are preferences where you can change the game settings in general. There is also the exit option in my main menu, which follows a confirmation that the player really wants to leave the game.

There is also an Exit option on my main screen, which follows a confirmation that the player really wants to leave the game. The Open option only allows the player to select an already saved file. Finally, there's the About FreeCol option, which just gives a brief description of the game.



Return to Index.



Use cases templates

Use case 1
Name: Novo jogo
ID: 1
Description: When starting a new game, the player has the option of choosing to play in single-player or multiplayer mode. If they choose multiplayer mode, they have to modify certain settings relating to the server, etc. In both cases, the player can edit the game settings.
Primary Actors: Player

Return to Index.

Use case 2
Name: Abrir um jogo
ID: 2
Description: Open an existing game, if it exists.
Primary Actors: Player

Use case 3
Name: Editar o Mapa
ID: 3
Description: Allows you to edit/remove the different existing objects, i.e. edit their characteristics
do mapa
Primary Actors: Player

Use case 4
Name: Mudar preferências
ID: 4
Description: Allows you to edit the different preferences/interface of the game.
Primary Actors: Player

Use case 5
Name: Sobre o FreeCol
ID: 5
Description: A brief description of the game, as well as the website, the project, the game's github and the freeCol manual.
Primary Actors: Player

Use case 6
Name: Sair
ID: 6
Description: Allows you to leave the game/end the game, and a confirmation is always sent if the player really wants to leave the game.
Primary Actors: Player

Return to Index.

Use case 7:
Name: Jogo
ID: 7
Description: After clicking on the button to open a new game, there are a number of options to choose from in the Game option. These are the options related to the game.
Primary Actors: Player

Use case 8:
Name: Ver
ID: 8
Description: After clicking on the button to open a new game, there is a set of options that we can choose from in the View option. These are options related to preferences/interface and some options that make it easier to play the game.
Primary Actors: Player

Use case 9:
Name: Ordens
ID: 9
Description: After clicking on the button to open a new game, there are a number of options to choose from in the Orders option. These are the options related to the Orders that the user is allowed to make.
Primary Actors: Player

Use case 10:
Name: Relatórios
ID: 10
Description: After clicking on the button to open a new game, there are a number of options to choose from in the Reports option. These are the options related to the reports/queries that the player can make.
Primary Actors: Player

Use case 11:
Name: Colopédia
ID: 11
Description: After clicking on the button to open a new game, there are a number of options to choose from in the Colopedia option. These are the options related to the different information in the game, i.e. basically like a "tutorial".
Primary Actors: Player

Return to Index.

Use case 12:
Name: Editar Minimapa
ID: 12
Description: After clicking on the button to open a new game, there is a mini-map where you can change the way it looks, i.e. you have 4 options ("+", "-", an eye and a square) where each of these options performs a different function on the mini-map.
Primary Actors: Player

Code Report Review Log

Reviewer: Antonio Palmeirim 63667

Review 1 (5/11):
Na minha opinião o use case diagram está bem estruturado com todas as opções possíveis descritas. A descrição do use case é único, o que neste caso faz sentido pois o jogador tem o mesmo comportamento com todas as opções do menu.

Review 2 (17/11):
Com as modificações feitas, e depois de um pouco de auto-aprendizagem, considero agora que é mais correto os use cases agora, com tudo dividido, isto é, um use case template para cada interação entre o sistema e jogador.

Review 3 (22/11):
Após as modificações, vi que aprofundas-te mais na área do ecrã principal e também adicionaste a parte da barra de menus, o que abrange mais do jogo. De resto nada a apontar!

Return to Index.

Reviewer: André Santos 62331

Review 1 (5/11):
Na minha opinião o diagrama parece me um bocado confuso e acho que poderia ser simplificado.

Review 2 (17/11):
O diagrama já está bastante mais simples. De resto nada a apontar.

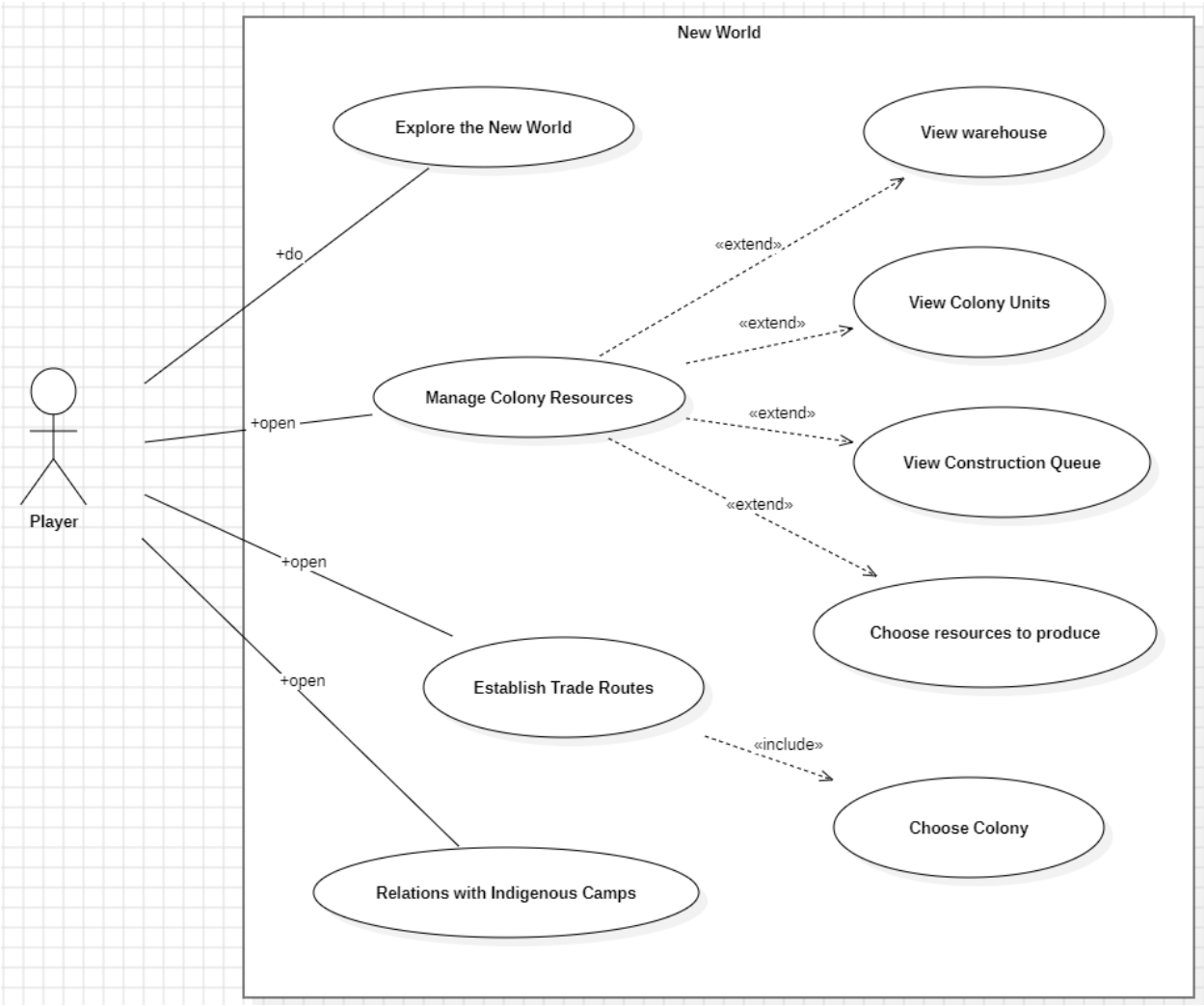
Reviewer: Tiago Santos

Review 1 (5/11):
A meu ver, o use case template deveria ser para cada uma das opções do menu inicial. De resto concordo com o mencionado.

Review 2 (17/11):
Agora sim, tendo 1 use case template para cada opção do menu inicial, considero que o documento está como esperado.

Return to Index.

Author: Tiago Santos (63390)



Return to Index.

Use Case: Explore the New World

ID: 1

Description: The player explores the New World, starting with a ship with two colonists. The aim is to discover lands to establish colonies and produce goods to send back to Europe. To move the ship, you drag the mouse from the ship to the direction you want to move.

Primary actor: Player

Secondary actor: None

Return to Index.

Use Case: Managing Colony Resources

ID: 2

Description: The player checks and manages the different types of terrain and resources available in the colony. He can view the resources in the warehouse; view the units present in the colony and can assign them positions to increase the colony's resource production; and view the construction queue (he can also add more buildings to the queue).

Primary actor: Player

Secondary actor: None

Return to Index.

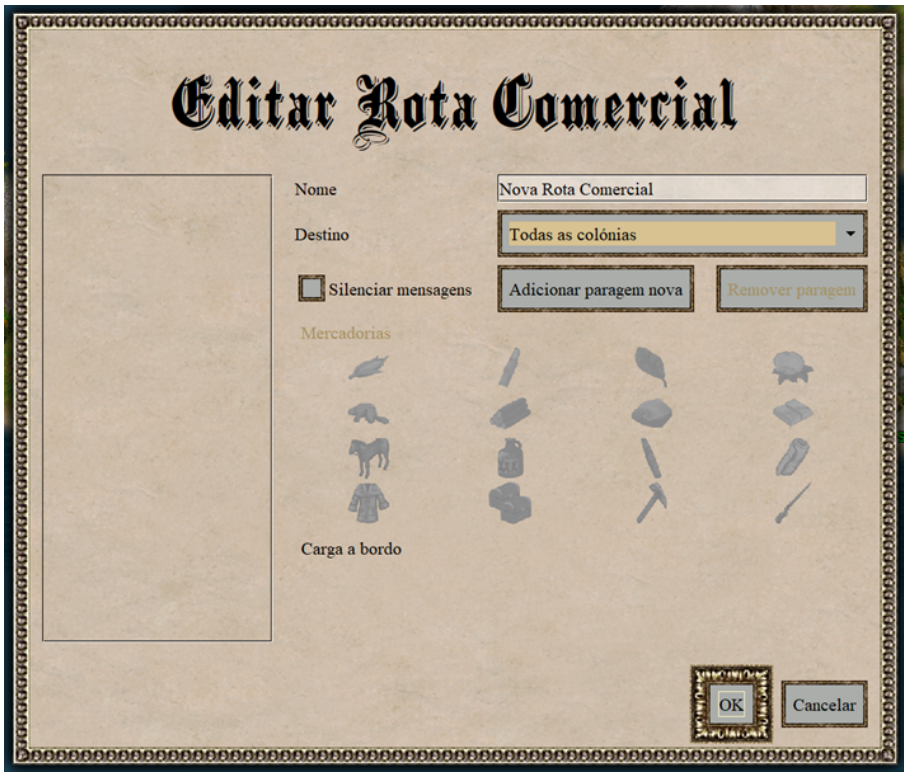
Use Case: Establishing Trade Routes

ID: 3

Description: The player creates and manages trade routes to transport goods between the New World and Europe. Access the orders menu to assign a trade route to an available vessel. Select at least 2 destinations and choose the goods to be transported between the stops on the route.

Primary actor: Player

Secondary actor: None



Return to Index.

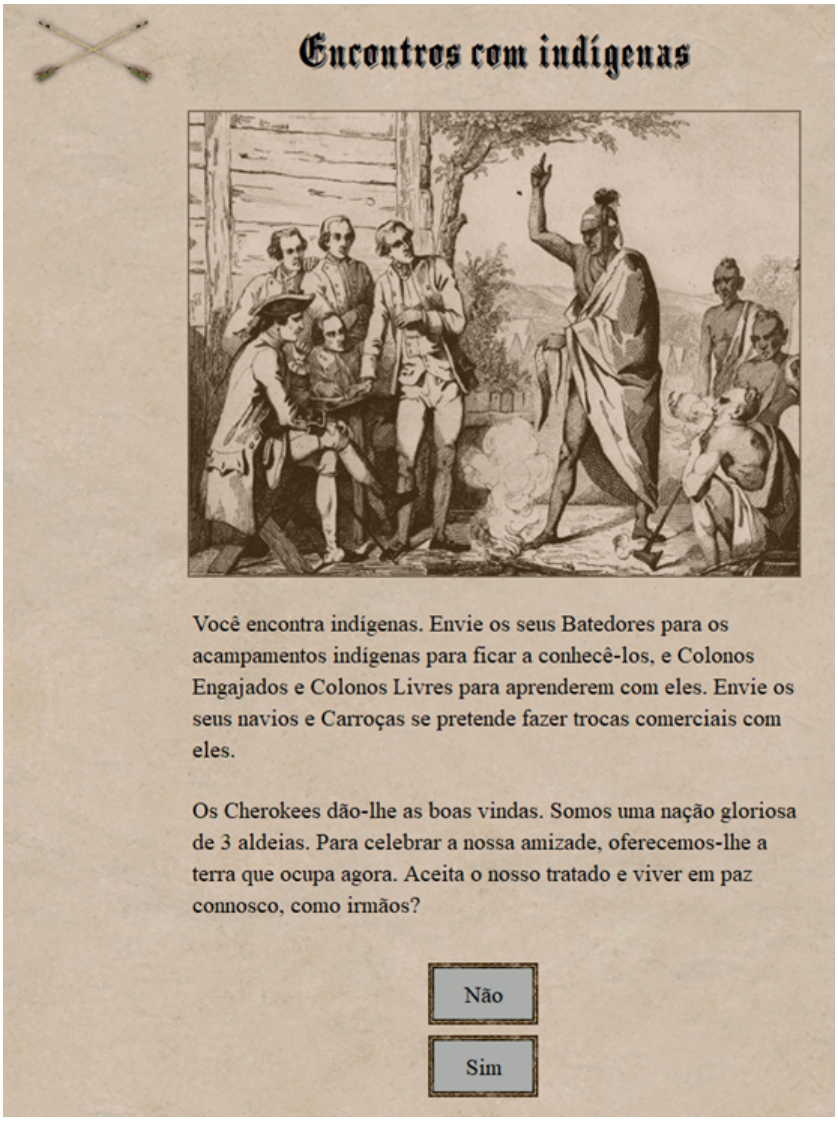
Use Case: Relations with Indigenous Camps

ID: 4

Description: The player decides how to interact with the native camps, choosing between coexisting peacefully or declaring war. Later, if they are at peace, when he interacts with the indigenous camps he can trade with them.

Primary actor: Player

Secondary actor: None



Return to Index.

Code Report Review Log

Reviewer: André Santos 62331

Review 1 (5/11)
Documento bem ilustrado facilitando assim a compreensão do ficheiro, documentação clara e concisa. Faltam apenas as pré e pós condições.

Reviewer: Duarte Inácio 62397

Review 1 (5/11)
Coloca os IDs em cada um dos use case templates.

Review 2 (19/11)
Agora sim parece-me tudo correto, nada a apontar!

Reviewer: António Palmeirim 63667

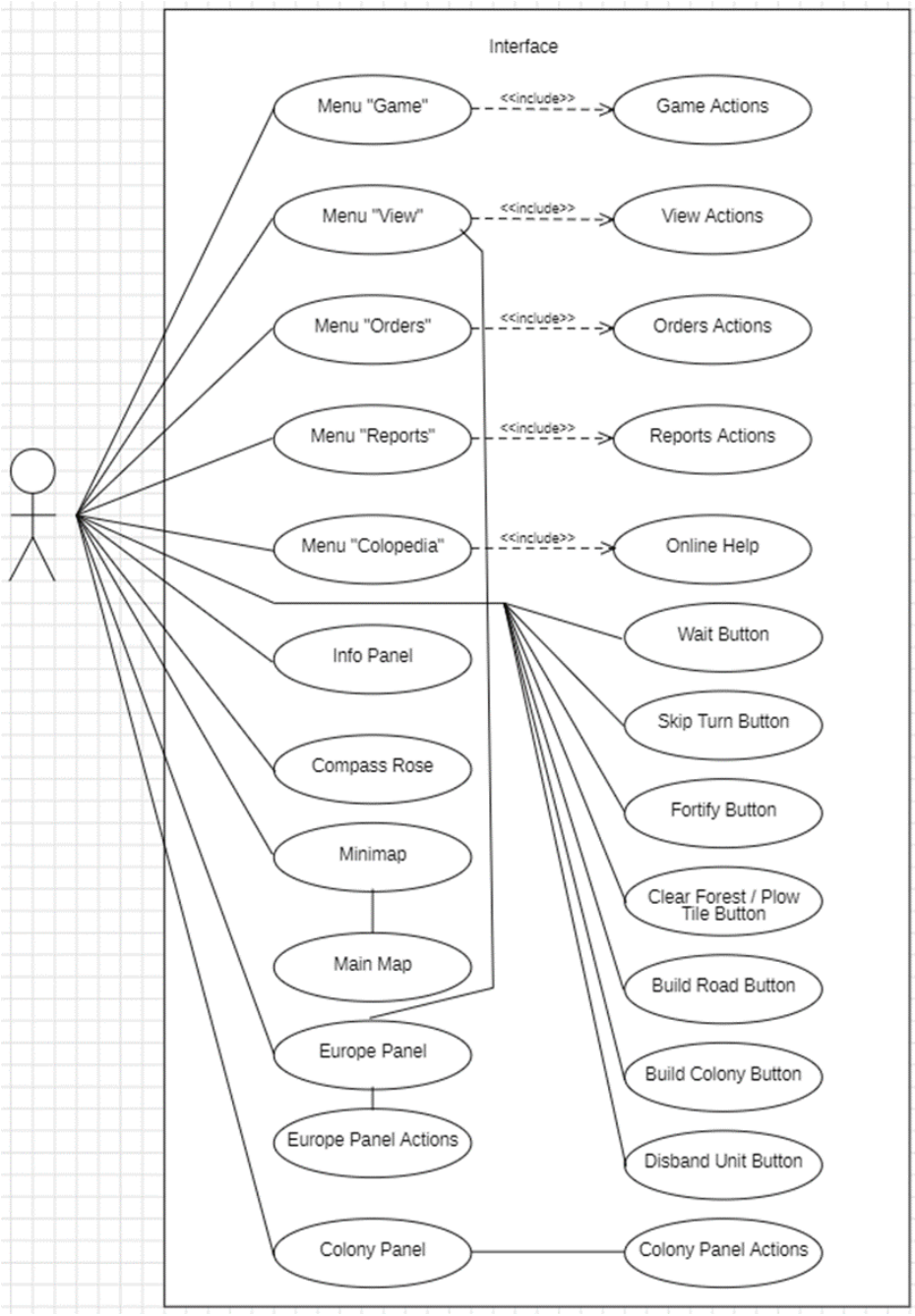
Review 1 (19/11)
O use case diagrama está correto e completo, só falta meter as explicações de cada use case à parte num PDF próprio.

Return to Index.

Author: Luís Serrano (60253)

Use Case: Interface

The Diagram for this use case is the following:



Return to Index.

Use cases template

- **Use Case 1**

Name: Menu “Game”
ID: 1
Description: Button that opens the game options menu.
Actors:
Primary: Player

- **Use Case 2**

Name: Game Actions
ID: 2
Description: These actions consist of starting a new game, opening a saved game, saving the current game, changing preferences, reconnecting to the server, chatting with another player, declaring independence, ending the turn, returning to the main menu, viewing the high scores, leaving the game and exiting the game.
Actors:
Primary: Player

- **Use Case 3**

Name: Menu “View”
ID: 3
Description: Button that opens the game options menu.
Actors:
Primary: Player

Return to Index.

- **Use Case 4**

Name: View Actions
ID: 4
Description: These actions consist of turning on and off the map controls, the grid and the borders. We can also switch between unit and terrain view, full screen or window. We can show the names of tiles, owners, regions or none of these, change the zoom of the main map, switch to the Europe panel, show trade routes and center the map on a known settlement.
Actors:

Primary: Player

- **Use Case 5**

Name: Menu “Orders”
ID: 5
Description: Button that opens a menu of command options.
Actors:

Primary: Player

- **Use Case 6**

Name: Orders Actions
ID: 6
Description: These actions consist of switching to sentry mode, fortifying, going to a selected location, going to a selected tile, executing goto orders, assigning trade routes, building or joining a settlement, plowing the tile where the unit is, building a road in the tile where the unit is.
Actors:

Primary: Player

Return to Index.

- **Use Case 7**

Name: Menu “Reports”
ID: 7
Description: Button that opens a menu of report options.
Actors:
Primary: Player

- **Use Case 8**

Name: Reports Actions
ID: 8
Description: Several advisors for each area give a report on how each of the areas is doing, respective to the advisor's area.
Actors:
Primary: Player

- **Use Case 9**

Name: Menu “Colopedia”
ID: 9
Description: Menu that opens the game's online help.
Actors:
Primary: Player

Return to Index.

- **Use Case 10**

Name: Online Help
ID: 10
Description: Game help, divided into 8 sections - terrain, bonus resources, goods, units, skills, buildings, Founding Father, nations and national advantages.
Actors:
Primary: Player

- **Use Case 11**

Name: Info Panel
ID: 11
Description: Panel in the bottom right-hand corner showing information about the selected unit. If there is no unit selected, there is a button to end the shift.
Actors:
Primary: Player

- **Use Case 12**

Name: Minimap
ID: 12
Description: Mini-map in the bottom left-hand corner showing a more abstract version of the map. It can be used to navigate the map more quickly and also allows you to see a larger area of the map.
Actors:
Primary: Player

Return to Index.

- **Use Case 13**

Name: Wait Button
ID: 13
Description: Button that allows you to wait until other units have moved.
Actors:
Primary: Player

- **Use Case 14**

Name: Skip Turn Button
ID: 14
Description: Button to skip the shift.
Actors:
Primary: Player

- **Use Case 15**

Name: Fortify Button
ID: 15
Description: Button that allows you to fortify.
Actors:
Primary: Player

- **Use Case 16**

Name: Clear Forest / Plow Tile Button
ID: 16
Description: Button to clear the forest or plow the tile.
Actors:
Primary: Player

Return to Index.

- **Use Case 17**

Name: Build Road Button
ID: 17
Description: Button that allows you to build a road.
Actors:
Primary: Player

- **Use Case 18**

Name: Build Colony Button
ID: 18
Description: Button to build a colony.
Actors:
Primary: Player

- **Use Case 19**

Name: Disband Unit Bytton
ID: 19
Description: Button that allows us to dissolve a unit.
Actors:
Primary: Player

- **Use Case 20**

Name: Compass Rose
ID: 20
Description: The compass rose in the top right corner allows you to give movement orders to your units by clicking on the desired direction.
Actors:
Primary: Player

Return to Index.

- **Use Case 21**

Name: Main Map
ID: 21
Description: The game's main map is the default view.
Actors:
Primary: Player

- **Use Case 22**

Name: Europe Panel
ID: 22
Description: Europe Panel.
Actors:
Primary: Player

- **Use Case 23**

Name: Europe Panel Actions
ID: 23
Description: In this panel we can control the ships sailing between America and Europe and the ships docked in Europe. You can also buy resources and purchase, recruit and train units.
Actors:
Primary: Player

- **Use Case 24**

Name: Colony Panel
ID: 24
Description: Colony Panel.
Actors:
Primary: Player

Return to Index.

- **Use Case 25**

Name: Colony Panel Actions
ID: 25
Description: This panel contains information about the selected colony. We can also unload the active boat or train. We can fill up all the parts that aren't there. We can open the Warehouse Dialog to change the export level of all goods. Finally, we can select which units or buildings we want to build.
Actors:
Primary: Player

Review log

Use Case Diagram review(Duarte Inácio 62397)
Review 1 (5/11):
Parece-me um Use Case diagram simples e de fácil compreensão. Tenta só meter algumas imagens para se entender melhor o use case.

Review 2 (27/11):
Já me parece bem com a imagem do use Case diagram! Nada a apontar!
Reviewer: Rafael Tavares 60608

Review 1 (5/11):
O use case diagram apresentado bem como a sua descrição são úteis para compreender a mecânica em questão. De um modo geral, o diagrama está claro e fácil de interpretar.

Return to Index.

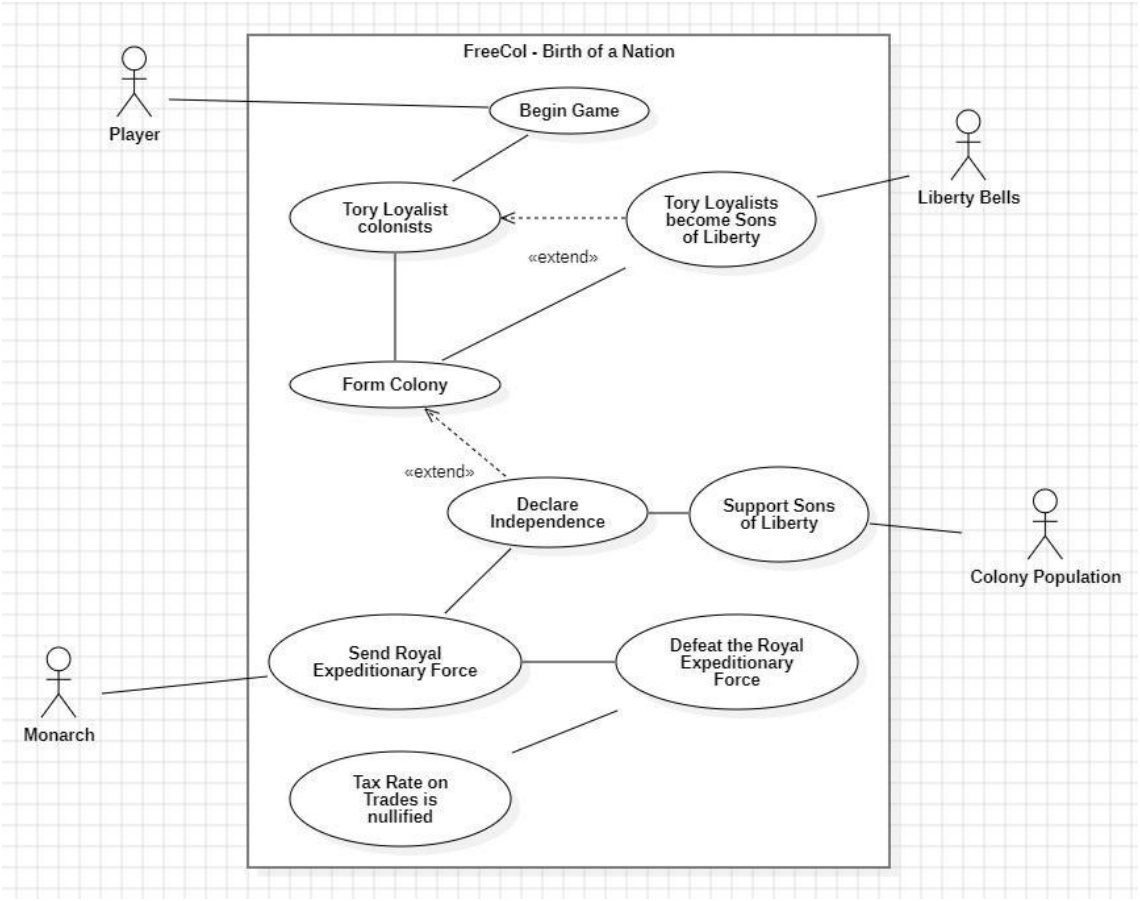
Author: Rafael Tavares (60608)

Use Case: Birth of a Nation

Description: The game commences with all colonists as Tory Loyalists, supporting the Monarch and opposing player policies. Upon the generation of Liberty Bells, the player can optionally convert Tory Loyalists into Sons of Liberty. Players form new colonies and, if desired, declare independence when 50% of the population supports the Sons of Liberty, triggering a conflict with the Monarch, who sends the Royal Expeditionary Force. Success in defeating the Royal Expeditionary Force leads to the establishment of a free nation, where the Custom House operates without external tax threats.

Primary actor: Player

Secondary actors: Monarch, Liberty Bells, Colony Population



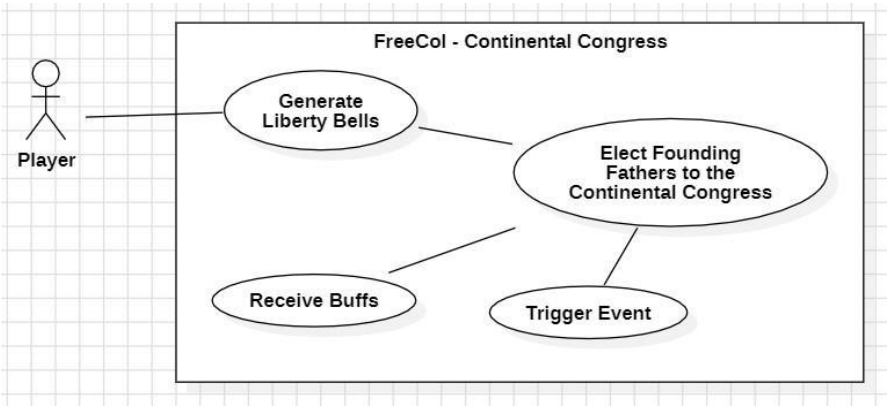
Return to Index.

Use Case: The Continental Congress

Description: As the player produces Liberty Bells in the game, they have the opportunity to elect Founding Fathers to the Continental Congress, these Founding Fathers, historical figures integral to the New World's conquest, provide the player with unique bonuses or trigger specific events. Initially, only a small number of Liberty Bells is required to elect a Founding Father, but as the game advances, the threshold for election may rise significantly, potentially requiring many hundreds of Liberty Bells.

Primary actor: Player

Secondary actors: None



Return to Index.

Review Log

Review Tiago Santos (63390) (4/11):

Acho que o Use Case está bem apresentado e explicado. Apenas acrescentaria imagens do jogo quando é feita cada ação explicada.

Return to Index.