ENGENHARIA DE SOFTWARE

Licenciatura em Engenharia Informática

Freecol
Final delivery

Miguel Matos Barreto, nº61891

Índice

Introduction

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

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

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

Our group develop the 1st user story funcionality:

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

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

1st Phase

User Stories

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

1st User Story

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

2nd User Story

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

3rd User Story

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

2nd Phase

Code Metrics

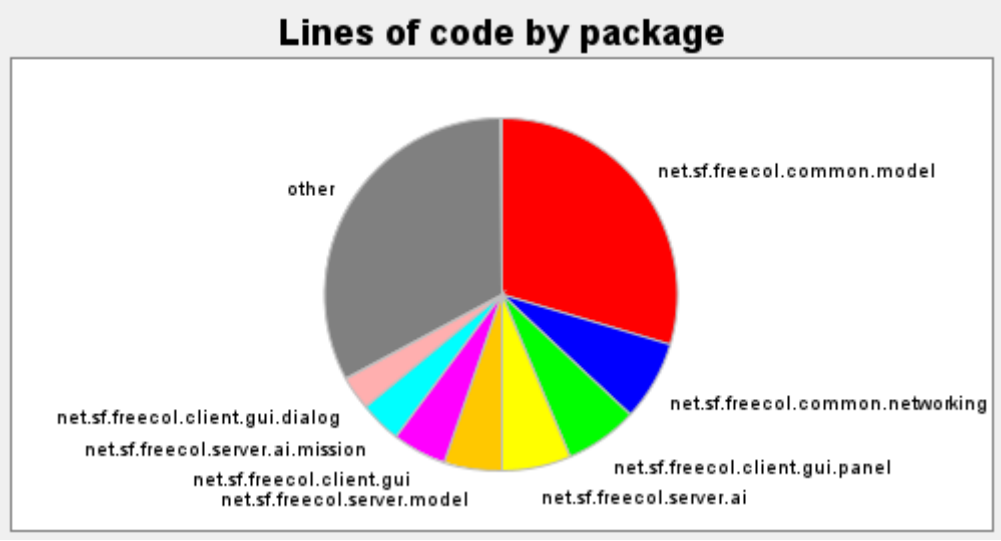
Miguel Barreto, nº 61891, mm-barreto

Lines of Code Metric

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

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

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



Upon inspecting the pie chart, we can observe that **net.sf.freecol.common.model** has a larger segment, suggesting that it contains more lines of code compared to other packages.
This could potentially be a trouble spot.
Packages with large amounts of code can be harder to maintain and understand and be more prone to bugs.
It relates to code smells like Long Methods and Long Classes.

Attribute Hiding Factor (AHF):

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

Attribute Inheritance Factor (AIF):

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

Coupling Factor (CF):

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

Method Hiding Factor (MHF):

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

Method Inheritance Factor (MIF):

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

Polymorphism Factor (PF):

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

Design Patterns

Miguel Barreto, nº 61891, mm-barreto

GoF Patterns

Singleton Pattern

Location:

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

Code snippet:

```
5 usages  Stian Grenborgen +2
public class FSGConverter {

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


    /**
     * Creates an instance of {@code FSGConverter}
     */
    1 usage  Stian Grenborgen
    private FSGConverter() {
        // Nothing to initialize;
    }


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

Explanation:

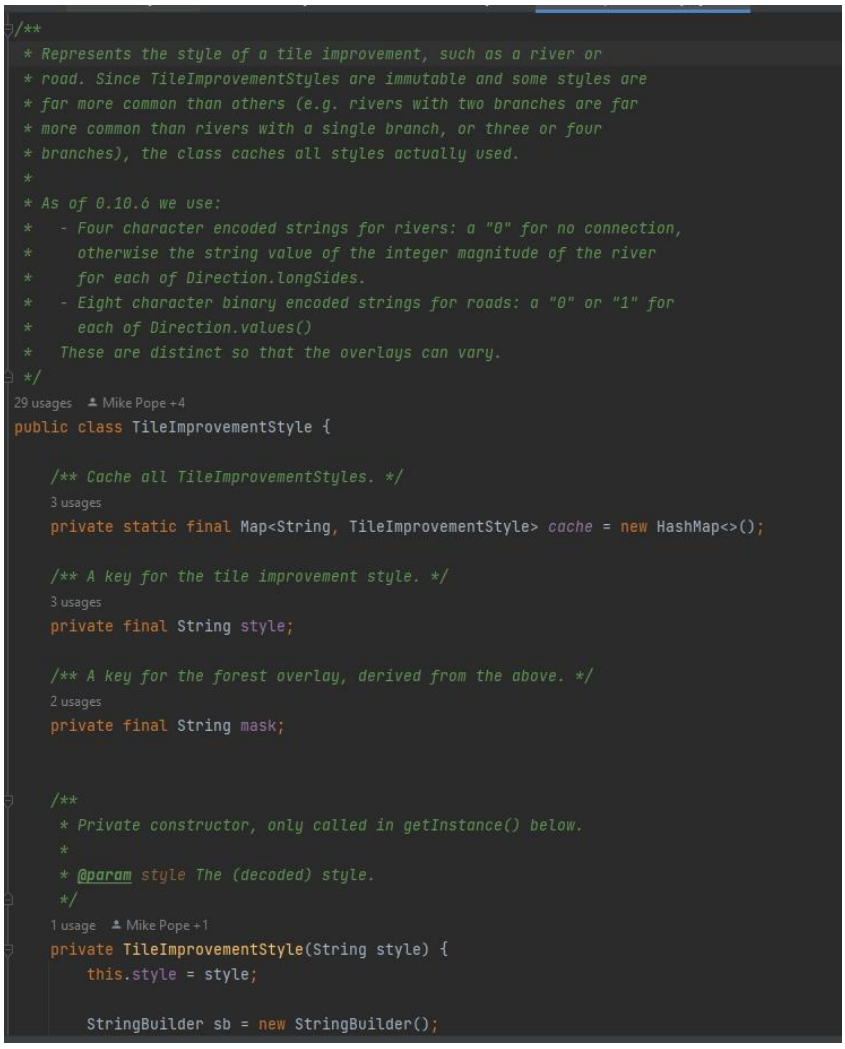
- The identification of this pattern is quite straightforward, we can easily identify the unique instance of the class and the lazy constructor, assuring that there is always only one instance of this class.

Decorator Pattern

Location:

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

Code snippet:



```
/**
 * Represents the style of a tile improvement, such as a river or
 * road. Since TileImprovementStyles are immutable and some styles are
 * far more common than others (e.g. rivers with two branches are far
 * more common than rivers with a single branch, or three or four
 * branches), the class caches all styles actually used.
 *
 * As of 0.10.6 we use:
 *   - Four character encoded strings for rivers: a "0" for no connection,
 *     otherwise the string value of the integer magnitude of the river
 *     for each of Direction.longSides.
 *   - Eight character binary encoded strings for roads: a "0" or "1" for
 *     each of Direction.values()
 *   These are distinct so that the overlays can vary.
 */
29 usages  Mike Pope +4
public class TileImprovementStyle {

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

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

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


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

        StringBuilder sb = new StringBuilder();
```

Explanation:

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

Diogo Correia, nº 62475, Correia21

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

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

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

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

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

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

Code Smells

Miguel Barreto, nº 61891, mm-barreto

net.sf.freecol.FreeCol:

Large Class:

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

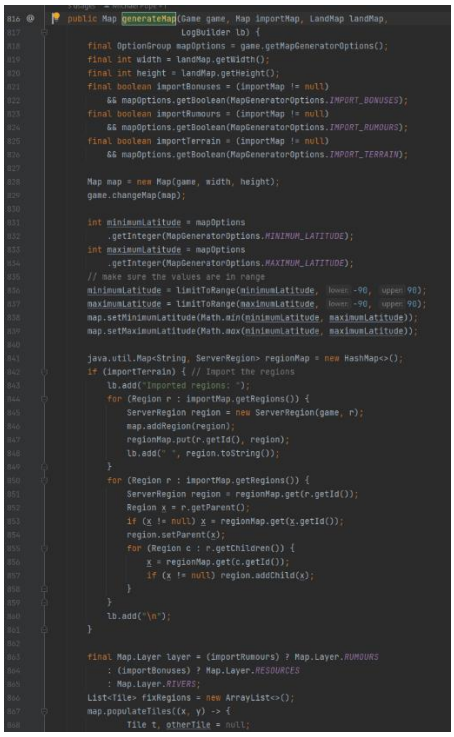
Duplicated Code:

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

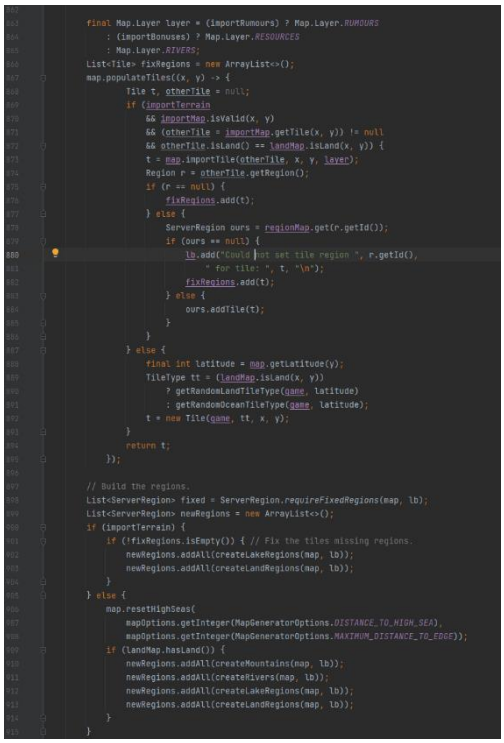
Net.sf.freecol.server.generator.TerrainGenerator:

Long Method:

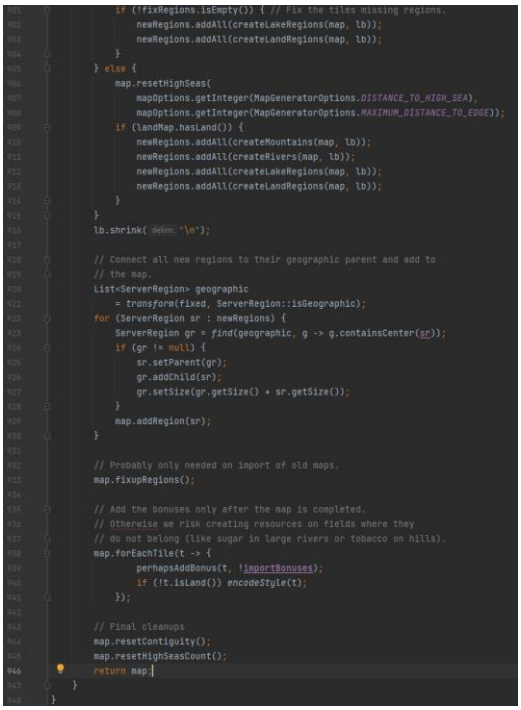
The generateMap method is quite lengthy, performing multiple tasks such as importing tiles, setting regions, creating mountains, rivers, lakes, and bonuses.



```
public Map generateMap(Game game, Map importMap, LandMap landMap,
                       LogBuilder lb) {
    final OptionGroup mapOptions = game.getMapGeneratorOptions();
    final int width = landMap.getWidth();
    final int height = landMap.getHeight();
    final boolean importBonuses = (importMap != null)
        && mapOptions.getBoolean(MapGeneratorOptions.IMPORT_BONUSES);
    final boolean importRumours = (importMap != null)
        && mapOptions.getBoolean(MapGeneratorOptions.IMPORT_RUMOURS);
    final boolean importTerrain = (importMap != null)
        && mapOptions.getBoolean(MapGeneratorOptions.IMPORT_TERRAIN);

    Map map = new Map(game, width, height);
    game.changeMap(map);

    int minimumLatitude = mapOptions
        .getInteger(MapGeneratorOptions.MINIMUM_LATITUDE);
    int maximumLatitude = mapOptions
        .getInteger(MapGeneratorOptions.MAXIMUM_LATITUDE);
    // make sure the values are in range
    minimumLatitude = limitToRange(minimumLatitude,  lower: -90,  upper: 90);
    maximumLatitude = limitToRange(maximumLatitude,  lower: -90,  upper: 90);
    map.setMinimumLatitude(Math.min(minimumLatitude, maximumLatitude));
    map.setMaximumLatitude(Math.max(minimumLatitude, maximumLatitude));

    java.util.Map<String, ServerRegion> regionMap = new HashMap<>();
    if (importTerrain) { // Import the regions
        lb.add("Imported regions: ");
        for (Region r : importMap.getRegions()) {
            ServerRegion region = new ServerRegion(game, r);
            map.addRegion(region);
            regionMap.put(r.getId(), region);
            lb.add(" ", region.toString());
        }
        for (Region r : importMap.getRegions()) {
            ServerRegion region = regionMap.get(r.getId());
            Region x = r.getParent();
            if (x != null) x = regionMap.get(x.getId());
            region.setParent(x);
            for (Region c : r.getChildren()) {
                x = regionMap.get(c.getId());
                if (x != null) region.addChild(x);
            }
        }
        lb.add("\n");
    }

    final Map.Layer layer = (importRumours) ? Map.Layer.RUMOURS
        : (importBonuses) ? Map.Layer.RESOURCES
        : Map.Layer.RIVERS;
    List<Tile> fixRegions = new ArrayList<>();
    map.populateTiles((x, y) -> {
            Tile t, otherTile = null;
```



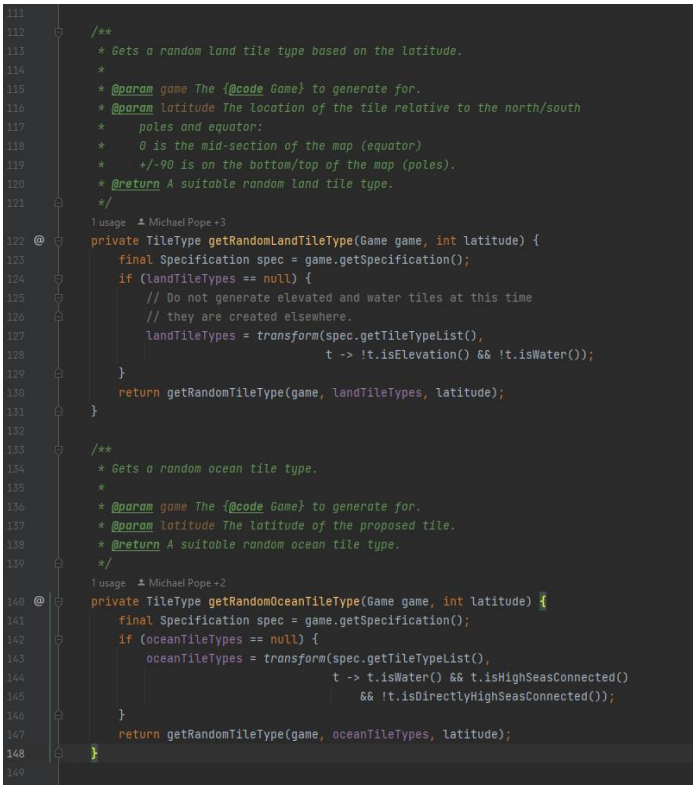
```
    final Map.Layer layer = (importRumours) ? Map.Layer.RUMOURS
        : (importBonuses) ? Map.Layer.RESOURCES
        : Map.Layer.RIVERS;
    List<Tile> fixRegions = new ArrayList<>();
    map.populateTiles((x, y) -> {
            Tile t, otherTile = null;
            if (importTerrain
                && importMap.isValid(x, y)
                && (otherTile = importMap.getTile(x, y)) != null
                && otherTile.isLand() == landMap.isLand(x, y)) {
                t = map.importTile(otherTile, x, y, layer);
                Region r = otherTile.getRegion();
                if (r == null) {
                    fixRegions.add(t);
                } else {
                    ServerRegion ours = regionMap.get(r.getId());
                    if (ours == null) {
                        lb.add("Could not set tile region ", r.getId(),
                               " for tile: ", t, "\n");
                        fixRegions.add(t);
                    } else {
                        ours.addTile(t);
                    }
                }
            } else {
                final int latitude = map.getLatitude(y);
                TileType tt = (landMap.isLand(x, y))
                    ? getRandomLandTileType(game, latitude)
                    : getRandomOceanTileType(game, latitude);
                t = new Tile(game, tt, x, y);
            }
            return t;
        });

    // Build the regions.
    List<ServerRegion> fixed = ServerRegion.requireFixedRegions(map, lb);
    List<ServerRegion> newRegions = new ArrayList<>();
    if (importTerrain) {
        if (!fixRegions.isEmpty()) { // Fix the tiles missing regions.
            newRegions.addAll(createLakeRegions(map, lb));
            newRegions.addAll(createLandRegions(map, lb));
        }
    } else {
        map.resetHighSeas(
            mapOptions.getInteger(MapGeneratorOptions.DISTANCE_TO_HIGH_SEA),
            mapOptions.getInteger(MapGeneratorOptions.MAXIMUM_DISTANCE_TO_EDGE));
        if (landMap.hasLand()) {
            newRegions.addAll(createMountains(map, lb));
            newRegions.addAll(createRivers(map, lb));
            newRegions.addAll(createLakeRegions(map, lb));
            newRegions.addAll(createLandRegions(map, lb));
        }
    }
```



```
            if (!fixRegions.isEmpty()) { // Fix the files missing regions.
                newRegions.addAll(createLakeRegions(map, lb));
                newRegions.addAll(createLandRegions(map, lb));
            }
        } else {
            map.resetHighSeas(
                mapOptions.getInteger(MapGeneratorOptions.DISTANCE_TO_HIGH_SEA),
                mapOptions.getInteger(MapGeneratorOptions.MAXIMUM_DISTANCE_TO_EDGE));
            if (landMap.hasLand()) {
                newRegions.addAll(createMountains(map, lb));
                newRegions.addAll(createRivers(map, lb));
                newRegions.addAll(createLakeRegions(map, lb));
                newRegions.addAll(createLandRegions(map, lb));
            }
        }
        lb.shrink( delim: "\n");

        // Connect all new regions to their geographic parent and add to
        // the map.
        List<ServerRegion> geographic
            = transform(fixed, ServerRegion::isGeographic);
        for (ServerRegion sr : newRegions) {
            ServerRegion gr = find(geographic, g -> g.containsCenter(sr));
            if (gr != null) {
                sr.setParent(gr);
                gr.addChild(sr);
                gr.setSize(gr.getSize() + sr.getSize());
            }
            map.addRegion(sr);
        }

        // Probably only needed on import of old maps.
        map.fixupRegions();

        // Add the bonuses only after the map is completed.
        // Otherwise we risk creating resources on fields where they
        // do not belong (like sugar in large rivers or tobacco on hills).
        map.forEachTile(t -> {
                perhapsAddBonus(t, !importBonuses);
                if (!t.isLand()) encodeStyle(t);
            });

        // Final cleanups
        map.resetContiguity();
        map.resetHighSeasCount();
        return map;
    }
}
```

Data Clumps:

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



```
111
112     /**
113      * Gets a random land tile type based on the latitude.
114      *
115      * @param game The {@code Game} to generate for.
116      * @param latitude The location of the tile relative to the north/south
117      *     poles and equator:
118      *     0 is the mid-section of the map (equator)
119      *     +/-90 is on the bottom/top of the map (poles).
120      * @return A suitable random land tile type.
121      */
         1 usage  Michael Pope +3
122 @   private TileType getRandomLandTileType(Game game, int latitude) {
123         final Specification spec = game.getSpecification();
124         if (landTileTypes == null) {
125             // Do not generate elevated and water tiles at this time
126             // they are created elsewhere.
127             landTileTypes = transform(spec.getTileTypeList(),
128                                       t -> !t.isElevation() && !t.isWater());
129         }
130         return getRandomTileType(game, landTileTypes, latitude);
131     }
132
133     /**
134      * Gets a random ocean tile type.
135      *
136      * @param game The {@code Game} to generate for.
137      * @param latitude The latitude of the proposed tile.
138      * @return A suitable random ocean tile type.
139      */
         1 usage  Michael Pope +2
140 @   private TileType getRandomOceanTileType(Game game, int latitude) {
141         final Specification spec = game.getSpecification();
142         if (oceanTileTypes == null) {
143             oceanTileTypes = transform(spec.getTileTypeList(),
144                                        t -> t.isWater() && t.isHighSeasConnected()
145                                            && !t.isDirectlyHighSeasConnected());
146         }
147         return getRandomTileType(game, oceanTileTypes, latitude);
148     }
149
```

For example, latitude is passed to

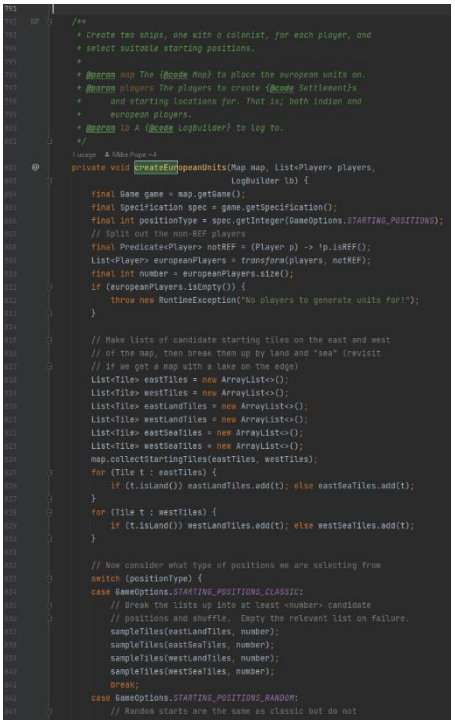
methods like getRandomLandTileType

and getRandomOceanTileType.

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

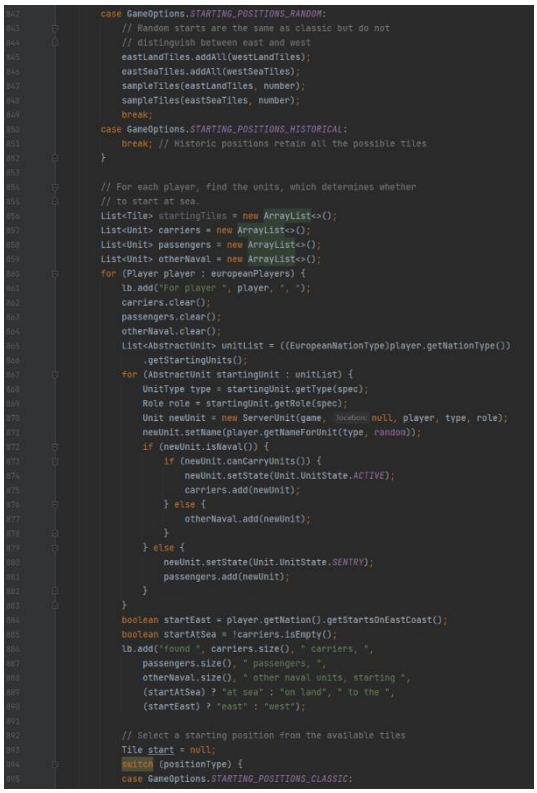
Long Method:

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



```
    /**
     * Create two ships, one with a colonist, for each player, and
     * select suitable starting positions.
     *
     * @param map The {@code Map} to place the european units on.
     * @param players The players to create {@code Settlement}s
     *      and starting locations for. That is; both indian and
     *      european players.
     * @param lb A {@code LogBuilder} to log to.
     */
    1 usage   Mike Pope +4
    private void createEuropeanUnits(Map map, List<Player> players,
                                     LogBuilder lb) {
        final Game game = map.getGame();
        final Specification spec = game.getSpecification();
        final int positionType = spec.getInteger(GameOptions.STARTING_POSITIONS);
        // Split out the non-REF players
        final Predicate<Player> notREF = (Player p) -> !p.isREF();
        List<Player> europeanPlayers = transform(players, notREF);
        final int number = europeanPlayers.size();
        if (europeanPlayers.isEmpty()) {
            throw new RuntimeException("No players to generate units for!");
        }

        // Make lists of candidate starting tiles on the east and west
        // of the map, then break them up by land and "sea" (revisit
        // if we get a map with a lake on the edge)
        List<Tile> eastTiles = new ArrayList<>();
        List<Tile> westTiles = new ArrayList<>();
        List<Tile> eastLandTiles = new ArrayList<>();
        List<Tile> westLandTiles = new ArrayList<>();
        List<Tile> eastSeaTiles = new ArrayList<>();
        List<Tile> westSeaTiles = new ArrayList<>();
        map.collectStartingTiles(eastTiles, westTiles);
        for (Tile t : eastTiles) {
            if (t.isLand()) eastLandTiles.add(t); else eastSeaTiles.add(t);
        }
        for (Tile t : westTiles) {
            if (t.isLand()) westLandTiles.add(t); else westSeaTiles.add(t);
        }

        // Now consider what type of positions we are selecting from
        switch (positionType) {
        case GameOptions.STARTING_POSITIONS_CLASSIC:
            // Break the lists up into at least <number> candidate
            // positions and shuffle.  Empty the relevant list on failure.
            sampleTiles(eastLandTiles, number);
            sampleTiles(eastSeaTiles, number);
            sampleTiles(westLandTiles, number);
            sampleTiles(westSeaTiles, number);
            break;
        case GameOptions.STARTING_POSITIONS_RANDOM:
            // Random starts are the same as classic but do not
```

```
            switch (positionType) {
            case GameOptions.STARTING_POSITIONS_CLASSIC:
                // Classic mode respects coast preference, the lists
                // are pre-sampled and shuffled
                start = ((startAtSea)
                    ? ((startEast) ? eastSeaTiles : westSeaTiles)
                    : ((startEast) ? eastLandTiles : westLandTiles)).remo
                break;
            case GameOptions.STARTING_POSITIONS_RANDOM:
                // Random mode is as classic but ignores coast
                // preference, the east lists already contain the west
                start = ((startAtSea) ? eastSeaTiles : eastLandTiles).re
                break;
            case GameOptions.STARTING_POSITIONS_HISTORICAL:
                start = (startAtSea)
                    ? findHistoricalStartingPosition(player, map,
                        eastSeaTiles, westSeaTiles)
                    : findHistoricalStartingPosition(player, map,
                        eastLandTiles, westLandTiles);
                break;
            }
            if (start == null) {
                throw new RuntimeException("Failed to find start tile "
                    + ((startAtSea) ? "at sea" : "on land")
                    + " for player " + player);
            }
            player.setEntryTile(start);
            lb.add(" at starting tile ", start);

            final Europe europe = player.getEurope();
            if (startAtSea) { // All aboard!
                for (Unit u : carriers) {
                    u.setLocation(start);
                    ((ServerPlayer)player).exploreForUnit(u);
                }
                passengers: for (Unit unit : passengers) {
                    for (Unit carrier : carriers) {
                        if (carrier.canAdd(unit)) {
                            unit.setLocation(carrier);
                            continue passengers;
                        }
                    }
                    // no space left on carriers
                    unit.setLocation(europe);
                }
            } else { // Land ho!
                for (Unit u : passengers) {
                    u.setLocation(start);
                    ((ServerPlayer)player).exploreForUnit(u);
                }
            }
            for (Unit u : otherNaval) {
                u.setLocation(europe);
            }
```



```
        case GameOptions.STARTING_POSITIONS_RANDOM:
            // Random starts are the same as classic but do not
            // distinguish between east and west
            eastLandTiles.addAll(westLandTiles);
            eastSeaTiles.addAll(westSeaTiles);
            sampleTiles(eastLandTiles, number);
            sampleTiles(eastSeaTiles, number);
            break;
        case GameOptions.STARTING_POSITIONS_HISTORICAL:
            break; // Historic positions retain all the possible tiles
        }

        // For each player, find the units, which determines whether
        // to start at sea.
        List<Tile> startingTiles = new ArrayList<>();
        List<Unit> carriers = new ArrayList<>();
        List<Unit> passengers = new ArrayList<>();
        List<Unit> otherNaval = new ArrayList<>();
        for (Player player : europeanPlayers) {
            lb.add("For player ", player, ", ");
            carriers.clear();
            passengers.clear();
            otherNaval.clear();
            List<AbstractUnit> unitList = ((EuropeanNationType)player.getNationType())
                .getStartingUnits();
            for (AbstractUnit startingUnit : unitList) {
                UnitType type = startingUnit.getType(spec);
                Role role = startingUnit.getRole(spec);
                Unit newUnit = new ServerUnit(game, location: null, player, type, role);
                newUnit.setName(player.getNameForUnit(type, random));
                if (newUnit.isNaval()) {
                    if (newUnit.canCarryUnits()) {
                        newUnit.setState(Unit.UnitState.ACTIVE);
                        carriers.add(newUnit);
                    } else {
                        otherNaval.add(newUnit);
                    }
                } else {
                    newUnit.setState(Unit.UnitState.SENTRY);
                    passengers.add(newUnit);
                }
            }
            boolean startEast = player.getNation().getStartsOnEastCoast();
            boolean startAtSea = !carriers.isEmpty();
            lb.add("found ", carriers.size(), " carriers, ",
                passengers.size(), " passengers, ",
                otherNaval.size(), " other naval units, starting ",
                (startAtSea) ? "at sea" : "on land", " to the ",
                (startEast) ? "east" : "west");

            // Select a starting position from the available tiles
            Tile start = null;
            switch (positionType) {
            case GameOptions.STARTING_POSITIONS_CLASSIC:
```

```
                }
            } else { // Land ho!
                for (Unit u : passengers) {
                    u.setLocation(start);
                    ((ServerPlayer)player).exploreForUnit(u);
                }
            }
            for (Unit u : otherNaval) {
                u.setLocation(europe);
            }

            if (FreeColDebugger.isInDebugMode(FreeColDebugger.DebugMode.INIT)) {
                createDebugUnits(map, player, start, lb);
                spec.setInteger(GameOptions.STARTING_MONEY, value: 10000);
            }

            // Start our REF player entry tile somewhere near to our
            // starting tile, but do not place their units.
            // They are assumed to always be on naval transport.
            final Player ourREF = player.getREFPlayer();
            if (ourREF == null) continue; // Some tests do not generate REFs
            final int threshold = 10;
            final int startY = start.getY();
            final Predicate<Tile> closeSeaTile
                = t -> !t.isLand() && Math.abs(t.getY() - startY) < threshold;
            List<Tile> refTiles = (startEast) ? eastTiles : westTiles;
            start = getRandomMember(logger, logMe: ourREF + " start",
                                    transform(refTiles, closeSeaTile), random);

            ourREF.setEntryTile(start);
            lb.add(" with REF at ", start, ".\n");
        }
    }

```

Data Clumps:

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

```
699
700     /**
701      * Generates a skill that could be taught from a settlement on the
702      * given tile.
703      *
704      * @param map The {@code Map}.
705      * @param tile The {@code Tile} where the settlement will be located.
706      * @param nationType The {@code NationType} to generate a skill for.
707      * @return A skill that can be taught to Europeans.
708      */
         1 usage   Michael Pope +2
709 @   private UnitType generateSkillForLocation(Map map, Tile tile,
710                                               NationType nationType) {
711         List<RandomChoice<UnitType>> skills
712             = ((IndianNationType)nationType).getSkills();
713         java.util.Map<GoodsType, Integer> scale
714             = transform(skills, alwaysTrue(),
715                         Function.identity(),
716                         Collectors.toMap(rc ->
717                             rc.getObject().getExpertProduction(), rc -> 1));
718
719         for (Tile t: tile.getSurroundingTiles( range: 1)) {
720             forEachMapEntry(scale, e -> {
721                     GoodsType goodsType = e.getKey();
722                     scale.put(goodsType, e.getValue()
723                         + t.getPotentialProduction(goodsType,  unitType: null));
724                 });
725         }
726
727         final Function<RandomChoice<UnitType>, RandomChoice<UnitType>> mapper
728             = rc -> {
729                 UnitType unitType = rc.getObject();
730                 return new RandomChoice<>(unitType,  probability: rc.getProbability()
731                     * scale.get(unitType.getExpertProduction()));
732             };
733         UnitType skill = RandomChoice.getWeightedRandom( logger: null,  logMe: null,
734             transform(skills, alwaysTrue(), mapper), random);
735         final Specification spec = map.getSpecification();
736         return (skill != null) ? skill
737             : getRandomMember(logger,  logMe: "Scout",
738                 spec.getUnitTypesWithAbility(Ability.EXPERT_SCOUT), random);
739     }
740
```

Enhancing Expressiveness:

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

Refactoring Long Methods:

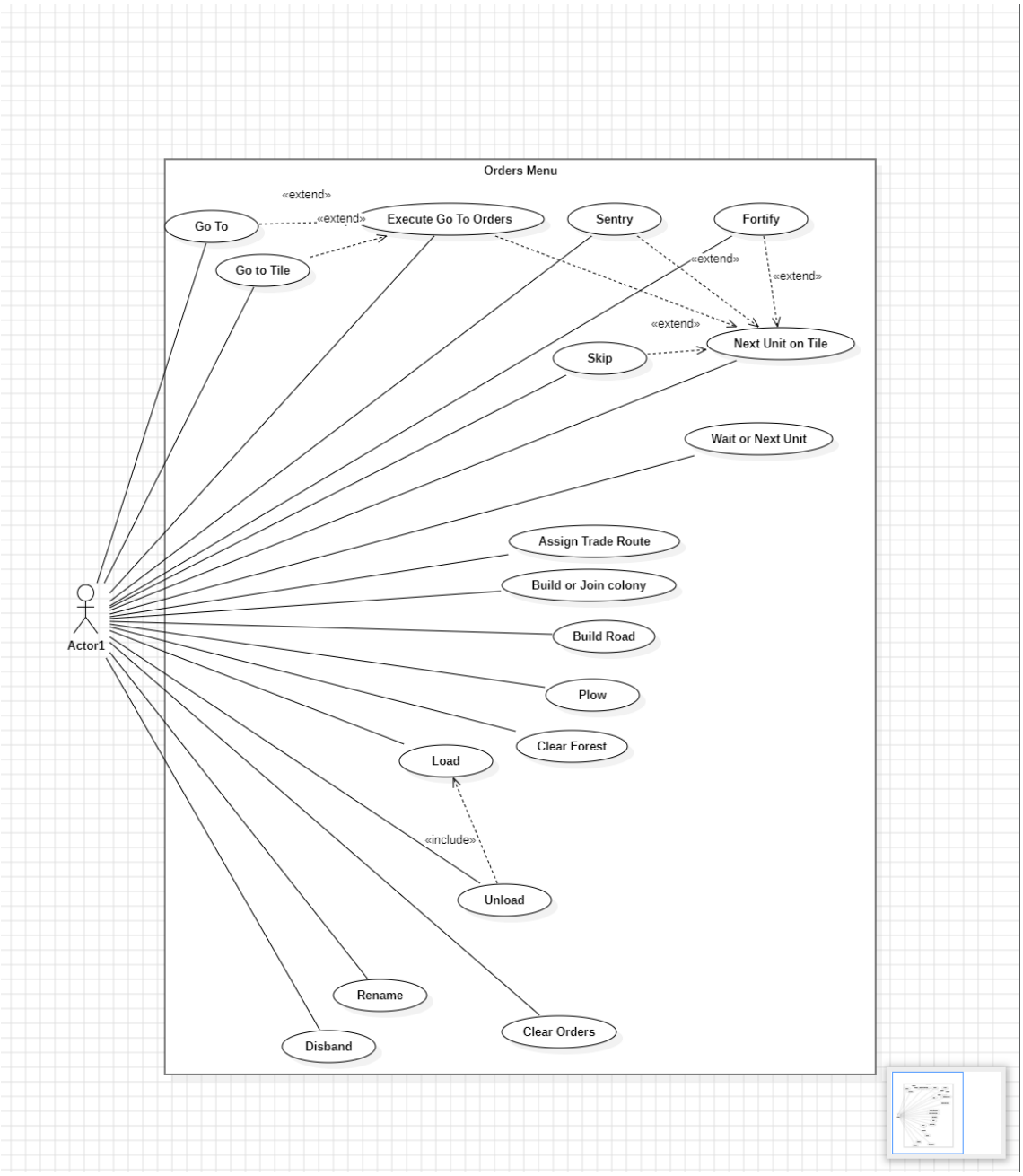
Nestled within the labyrinth of code, a lengthy method named moveDirection in InGameController.java has surfaced. This method weaves through an array of conditional checks and diverse actions. A judicious approach here involves the surgical division of this behemoth into smaller, more specialized methods.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

Mitigating Duplicated Code:

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

Use Case Diagram

Miguel Barreto, nº 61891, mm-barreto



Final Phase

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

1st User Story

We succeeded on implementing the 1st user story.
We developed a funcionality where when a unit crosses a river has 15% chance to trigger the event “you found gold in the river”.

Demo video

2nd User Story

Our group tried to implemented a functionality where we could speak with natives that were close to a unit tile, we implemented A NativeRecruit class that would work like NativeTrade but to recruit natives, a UnitRecruitable that extends Recruitable (an abstract class created to make natives a recruitable Object. The abstract class extends FreeColGameObject and would work like the class TradeItem (that makes Goods become tradeable)) the UnitRecruitable Class would work like NativeTradeItem but to make units recruitable.
We could not make this work due to time shortage.

3rd User Story

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