

ENGENHARIA DE SOFTWARE

Licenciatura em Engenharia Informática



Freecol
Final delivery

Miguel Matos Barreto, nº61891

Índice

Introduction	4
1st Phase	5
User Stories	5
1st User Story	5
2nd User Story	5
3rd User Story	5
2nd Phase	6
Code Metrics	6
Lines of Code Metric	6
Design Patterns	9
Code Smells	12
Use Case Diagram.....	19
Final Phase.....	20

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 functionality:

- 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 functionality 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

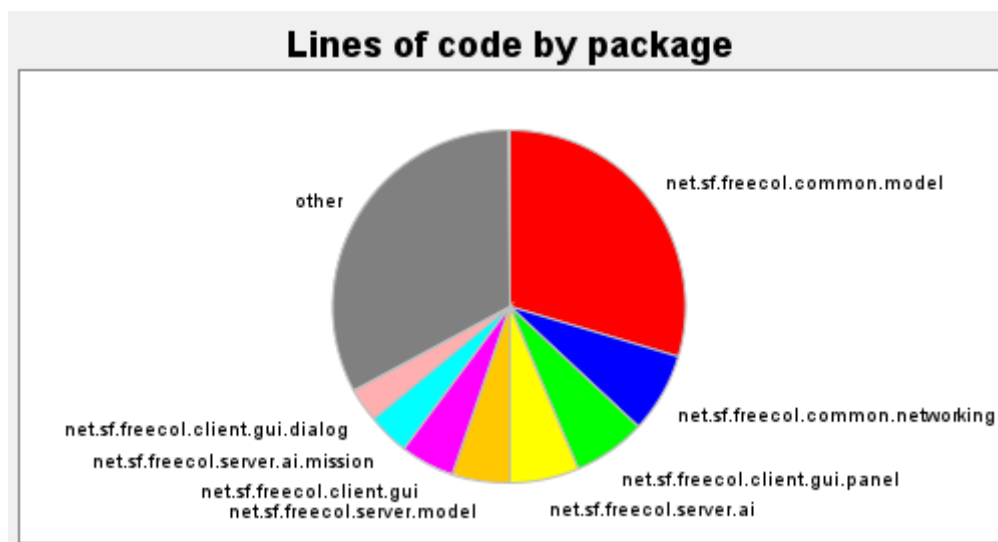
Lines of Code Metric

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 Grenborg +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 Grenborg
    private FSGConverter() {
        // Nothing to initialize;
    }

    /**
     * Gets an object for converting FreeCol Savegames.
     * @return The singleton object.
     */
    3 usages  Stian Grenborg +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.

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.

```
140 public Map generateMap(Game game, Map importMap, LandMap landMap,
141                          LogBuilder lb) {
142     final OptionGroup mapOptions = game.getMapGeneratorOptions();
143     final int width = landMap.getWidth();
144     final int height = landMap.getHeight();
145     final boolean importBonuses = (importMap != null)
146         && mapOptions.getBoolean(MapGeneratorOptions.IMPORT_BONUSES);
147     final boolean importRumours = (importMap != null)
148         && mapOptions.getBoolean(MapGeneratorOptions.IMPORT_RUMOURS);
149     final boolean importTerrain = (importMap != null)
150         && mapOptions.getBoolean(MapGeneratorOptions.IMPORT_TERRAIN);
151
152     Map map = new Map(game, width, height);
153     game.changeMap(map);
154
155     int minimumLatitude = mapOptions
156         .getInteger(MapGeneratorOptions.MINIMUM_LATITUDE);
157     int maximumLatitude = mapOptions
158         .getInteger(MapGeneratorOptions.MAXIMUM_LATITUDE);
159     // make sure the values are in range
160     minimumLatitude = limitToRange(minimumLatitude, -90, upper 90);
161     maximumLatitude = limitToRange(maximumLatitude, -90, upper 90);
162     map.setMinimumLatitude(Math.min(minimumLatitude, maximumLatitude));
163     map.setMaximumLatitude(Math.max(minimumLatitude, maximumLatitude));
164
165     java.util.Map<String, ServerRegion> regionMap = new HashMap<>();
166     if (importTerrain) { // Import the regions
167         lb.add("Imported regions:");
168         for (Region r : importMap.getRegions()) {
169             ServerRegion region = new ServerRegion(game, r);
170             map.addRegion(region);
171             regionMap.put(r.getId(), region);
172             lb.add(" ", region.toString());
173         }
174         for (Region r : importMap.getRegions()) {
175             ServerRegion region = regionMap.get(r.getId());
176             Region x = r.getParent();
177             if (x != null) x = regionMap.get(x.getId());
178             region.setParent(x);
179             for (Region c : r.getChildren()) {
180                 x = regionMap.get(c.getId());
181                 if (x != null) region.addChild(x);
182             }
183         }
184         lb.add("\n");
185     }
186
187     final Map.Layer layer = (importRumours) ? Map.Layer.RUMOURS
188         : (importBonuses) ? Map.Layer.RESOURCES
189         : Map.Layer.RIVERS;
190     List<Tile> fixRegions = new ArrayList<>();
191     map.populateTiles(x, y) -> {
192         Tile t, otherTile = null;
```

```
193         final Map.Layer layer = (importRumours) ? Map.Layer.RUMOURS
194             : (importBonuses) ? Map.Layer.RESOURCES
195             : Map.Layer.RIVERS;
196         List<Tile> fixRegions = new ArrayList<>();
197         map.populateTiles(x, y) -> {
198             Tile t, otherTile = null;
199             if (importTerrain)
200                 && importMap.isValid(x, y)
201                 && (otherTile = importMap.getTile(x, y)) != null
202                 && otherTile.isLand() == landMap.isLand(x, y) {
203                 t = map.importTile(otherTile, x, y, layer);
204                 Region r = otherTile.getRegion();
205                 if (r != null) {
206                     fixRegions.add(t);
207                     if (r == null) {
208                         ServerRegion ours = regionMap.get(r.getId());
209                         if (ours == null) {
210                             lb.add("Could not set tile region ", r.getId(),
211                                 " for tile: ", t, "\n");
212                             fixRegions.add(t);
213                         } else {
214                             ours.addTile(t);
215                         }
216                     }
217                 } else {
218                     final int latitude = map.getLatitude(y);
219                     TileType tt = (landMap.isLand(x, y))
220                         ? getRandomLandTileType(game, latitude)
221                         : getRandomOceanTileType(game, latitude);
222                     t = new Tile(game, tt, x, y);
223                 }
224                 return t;
225             }
226         });
227
228         // Build the regions.
229         List<ServerRegion> fixed = ServerRegion.requireFixedRegions(map, lb);
230         List<ServerRegion> newRegions = new ArrayList<>();
231         if (importTerrain) {
232             if (fixRegions.isEmpty()) { // Fix the tiles missing regions.
233                 newRegions.addAll(createLakeRegions(map, lb));
234                 newRegions.addAll(createLandRegions(map, lb));
235             }
236         } else {
237             map.resetHighSeas(
238                 mapOptions.getInteger(MapGeneratorOptions.DISTANCE_TO_HIGH_SEA),
239                 mapOptions.getInteger(MapGeneratorOptions.MAXIMUM_DISTANCE_TO_EDGE));
240             if (landMap.hasLand()) {
241                 newRegions.addAll(createMountains(map, lb));
242                 newRegions.addAll(createRivers(map, lb));
243                 newRegions.addAll(createLakesRegions(map, lb));
244                 newRegions.addAll(createLandRegions(map, lb));
245             }
246         }
247     }
```

```

901         if (firstRegions.isEmpty()) { // fix the tiles missing regions.
902             newRegions.addAll(createLakeRegions(map, lb));
903             newRegions.addAll(createLandRegions(map, lb));
904         }
905     } else {
906         map.resetHighSeas(
907             mapOptions.getInteger(MapGeneratorOptions.DISTANCE_TO_HIGH_SEA),
908             mapOptions.getInteger(MapGeneratorOptions.MAXIMUM_DISTANCE_TO_EDGE));
909         if (landMap.hasLand()) {
910             newRegions.addAll(createMountains(map, lb));
911             newRegions.addAll(createRivers(map, lb));
912             newRegions.addAll(createLakeRegions(map, lb));
913             newRegions.addAll(createLandRegions(map, lb));
914         }
915     }
916     lb.shrink(30000L, "ln");
917
918     // Connect all new regions to their geographic parent and add to
919     // the map.
920     List<ServerRegion> geographic
921         = transform(fixed, ServerRegion::isGeographic);
922     for (ServerRegion sr : newRegions) {
923         ServerRegion gr = find(geographic, g -> g.containsCenter(sr));
924         if (gr != null) {
925             sr.setParent(gr);
926             gr.addChild(sr);
927             gr.setSize(gr.getSize() + sr.getSize());
928         }
929         map.addRegion(sr);
930     }
931
932     // Probably only needed on import of old maps.
933     map.fixupRegions();
934
935     // Add the bonuses only after the map is completed.
936     // Otherwise we risk creating resources on fields where they
937     // do not belong (like sugar in large rivers or tobacco on hills).
938     map.forEachTile(t -> {
939         perhapsAddBonus(t, !importBonuses);
940         if (!t.island()) encodeStyle(t);
941     });
942
943     // Final cleanups
944     map.resetContiguity();
945     map.resetHighSeasCount();
946     return map;
947 }

```

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.

```
113
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  */
122 1 usage  Michael Pope +3
123 private TileType getRandomLandTileType(Game game, int latitude) {
124     final Specification spec = game.getSpecification();
125     if (landTileTypes == null) {
126         // Do not generate elevated and water tiles at this time
127         // they are created elsewhere.
128         landTileTypes = transform(spec.getTileTypeList(),
129                                 t -> !t.isElevation() && !t.isWater());
130     }
131     return getRandomTileType(game, landTileTypes, latitude);
132 }
133
134 /**
135  * Gets a random ocean tile type.
136  *
137  * @param game The {@code Game} to generate for.
138  * @param latitude The latitude of the proposed tile.
139  * @return A suitable random ocean tile type.
140  */
141 1 usage  Michael Pope +2
142 private TileType getRandomOceanTileType(Game game, int latitude) {
143     final Specification spec = game.getSpecification();
144     if (oceanTileTypes == null) {
145         oceanTileTypes = transform(spec.getTileTypeList(),
146                                 t -> t.isWater() && t.isHighSeasConnected()
147                                     && !t.isDirectlyHighSeasConnected());
148     }
149     return getRandomTileType(game, oceanTileTypes, latitude);
150 }
```

*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.

```
793  //
794  // Create two ships, one with a colonist, for each player, and
795  // select suitable starting positions.
796  //
797  // @param map The (Hedge Map) to place the european units on.
798  // @param players The players to create (Hedge Settlements)
799  // and starting locations for. That is, both indian and
800  // european players.
801  // @param lb A (Hedge Logistics) to log to.
802  //
803  // Usage: A HedgeGame->createEuropeanUnits(Map map, List<Player> players,
804  //                                         Logistics lb) {
805  //
806  //     final Game game = map.getGame();
807  //     final Specification spec = game.getSpecification();
808  //     final int positionType = spec.getInteger(GameOptions.STARTING_POSITIONS);
809  //     // Split out the non-REF players
810  //     final Predicate<Player> notRef = (Player p) -> !p.isREF();
811  //     List<Player> europeanPlayers = transform(players, notRef);
812  //     final int number = europeanPlayers.size();
813  //     if (europeanPlayers.isEmpty()) {
814  //         throw new RuntimeException("No players to generate units for!");
815  //     }
816  //
817  //     // Make lists of candidate starting tiles on the east and west
818  //     // of the map, then break them up by land and "sea" (revisit
819  //     // if we get a map with a lake on the edge)
820  //     List<Tile> eastTiles = new ArrayList<>();
821  //     List<Tile> westTiles = new ArrayList<>();
822  //     List<Tile> eastLandFiles = new ArrayList<>();
823  //     List<Tile> westLandFiles = new ArrayList<>();
824  //     List<Tile> eastSeaFiles = new ArrayList<>();
825  //     List<Tile> westSeaFiles = new ArrayList<>();
826  //     map.collectStartingFiles(eastTiles, westTiles);
827  //     for (Tile t : eastTiles) {
828  //         if (t.isLand()) eastLandFiles.add(t); else eastSeaFiles.add(t);
829  //     }
830  //     for (Tile t : westTiles) {
831  //         if (t.isLand()) westLandFiles.add(t); else westSeaFiles.add(t);
832  //     }
833  //
834  //     // Now consider what type of positions we are selecting from
835  //     switch (positionType) {
836  //     case GameOptions.STARTING_POSITIONS_CLASSIC:
837  //         // Break the lists up into at least <number> candidate
838  //         // positions and shuffle. Empty the relevant list on failure.
839  //         sampleFiles(eastLandFiles, number);
840  //         sampleFiles(eastSeaFiles, number);
841  //         sampleFiles(westLandFiles, number);
842  //         sampleFiles(westSeaFiles, number);
843  //         break;
844  //     case GameOptions.STARTING_POSITIONS_RANDOM:
845  //         // Random starts are the same as classic but do not
846  //
847  //     }
848  //
849  //     switch (positionType) {
850  //     case GameOptions.STARTING_POSITIONS_CLASSIC:
851  //         // Classic mode respects coast preference, the lists
852  //         // are pre-sampled and shuffled
853  //         start = ((startAtSea)
854  //             ? ((startEast) ? eastSeaFiles : westSeaFiles)
855  //             : ((startEast) ? eastLandFiles : westLandFiles)).remove();
856  //         break;
857  //     case GameOptions.STARTING_POSITIONS_RANDOM:
858  //         // Random mode is as classic but ignores coast
859  //         // preference, the east lists already contain the west
860  //         start = ((startAtSea) ? eastSeaFiles : eastLandFiles).remove();
861  //         break;
862  //     case GameOptions.STARTING_POSITIONS_HISTORICAL:
863  //         start = (startAtSea)
864  //             ? findHistoricalStartingPosition(player, map,
865  //                 eastSeaFiles, westSeaFiles)
866  //             : findHistoricalStartingPosition(player, map,
867  //                 eastLandFiles, westLandFiles);
868  //         break;
869  //     }
870  //     if (start == null) {
871  //         throw new RuntimeException("Failed to find start tile "
872  //             + ((startAtSea) ? "at sea" : "on land")
873  //             + " for player " + player);
874  //     }
875  //     player.setEntryTile(start);
876  //     lb.add("at starting tile ", start);
877  //
878  //     final Europe europe = player.getEurope();
879  //     if (startAtSea) { // All aboard!
880  //         for (Unit u : carriers) {
881  //             u.setLocation(start);
882  //             ((ServerPlayer)player).exploreForUnit(u);
883  //         }
884  //         passengers: for (Unit unit : passengers) {
885  //             for (Unit carrier : carriers) {
886  //                 if (carrier.canAdd(unit)) {
887  //                     unit.setLocation(carrier);
888  //                     continue passengers;
889  //                 }
890  //             }
891  //             // no space left on carriers
892  //             unit.setLocation(europe);
893  //         }
894  //     } else { // Land ho!
895  //         for (Unit u : passengers) {
896  //             u.setLocation(start);
897  //             ((ServerPlayer)player).exploreForUnit(u);
898  //         }
899  //         for (Unit u : otherNaval) {
900  //             u.setLocation(europe);
901  //         }
902  //     }
903  // }
```

```

932 case GameOptions.STARTING_POSITIONS_RANDOM:
933     // Random starts are the same as classic but do not
934     // distinguish between east and west
935     eastLandTiles.addAll(westLandTiles);
936     eastSeaTiles.addAll(westSeaTiles);
937     sampleTiles(eastLandTiles, number);
938     sampleTiles(eastSeaTiles, number);
939     break;
940 case GameOptions.STARTING_POSITIONS_HISTORICAL:
941     break; // Historic positions retain all the possible tiles
942 }
943
944 // For each player, find the units, which determines whether
945 // to start at sea.
946 List<Tile> startingTiles = new ArrayList<>();
947 List<Unit> carriers = new ArrayList<>();
948 List<Unit> passengers = new ArrayList<>();
949 List<Unit> otherNaval = new ArrayList<>();
950 for (Player player : europeanPlayers) {
951     lb.add("For player ", player, ", ");
952     carriers.clear();
953     passengers.clear();
954     otherNaval.clear();
955     List<AbstractUnit> unitList = ((EuropeanNationType)player.getNationType())
956         .getStartingUnits();
957     for (AbstractUnit startingUnit : unitList) {
958         UnitType type = startingUnit.getType(spec);
959         Role role = startingUnit.getRole(spec);
960         Unit newUnit = new ServerUnit(game, location, null, player, type, role);
961         newUnit.setName(player.getNameForUnit(type, random));
962         if (newUnit.isNaval()) {
963             if (newUnit.canCarryUnits()) {
964                 newUnit.setState(Unit.UnitState.ACTIVE);
965                 carriers.add(newUnit);
966             } else {
967                 otherNaval.add(newUnit);
968             }
969         } else {
970             newUnit.setState(Unit.UnitState.SENTRY);
971             passengers.add(newUnit);
972         }
973     }
974     boolean startEast = player.getNation().getStartOnEastCoast();
975     boolean startAtSea = carriers.isEmpty();
976     lb.add("found ", carriers.size(), " carriers, ",
977         passengers.size(), " passengers, ",
978         otherNaval.size(), " other naval units, starting ",
979         (startAtSea ? "at sea" : "on land", " to the ",
980         (startEast ? "east" : "west"));
981
982     // Select a starting position from the available tiles
983     Tile start = null;
984     choose (positionType) {
985 case GameOptions.STARTING_POSITIONS_CLASSIC:

```

```

938     }
939 } else { // Land ho!
940     for (Unit u : passengers) {
941         u.setLocation(start);
942         ((ServerPlayer)player).exploreForUnit(u);
943     }
944 }
945 for (Unit u : otherNaval) {
946     u.setLocation(europe);
947 }
948
949 if (FreeColDebugger.isInDebugMode(FreeColDebugger.DebugMode.INIT)) {
950     createDebugUnits(map, player, start, lb);
951     spec.setInteger(GameOptions.STARTING_MONEY, value: 10000);
952 }
953
954 // Start our REF player entry tile somewhere near to our
955 // starting tile, but do not place their units.
956 // They are assumed to always be on naval transport.
957 final Player ourREF = player.getREFPlayer();
958 if (ourREF == null) continue; // Some tests do not generate REFs
959 final int threshold = 10;
960 final int startY = start.getY();
961 final Predicate<Tile> closeSeaTile
962     = t -> !t.isLand() && Math.abs(t.getY() - startY) < threshold;
963 List<Tile> refTiles = (startEast ? eastTiles : westTiles);
964 start = getRandomMember(logger, logMe: ourREF + " start",
965     transform(refTiles, closeSeaTile), random);
966
967 ourREF.setEntryTile(start);
968 lb.add(" with REF at ", start, ".\n");
969 }
970 }
971

```


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   */
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:

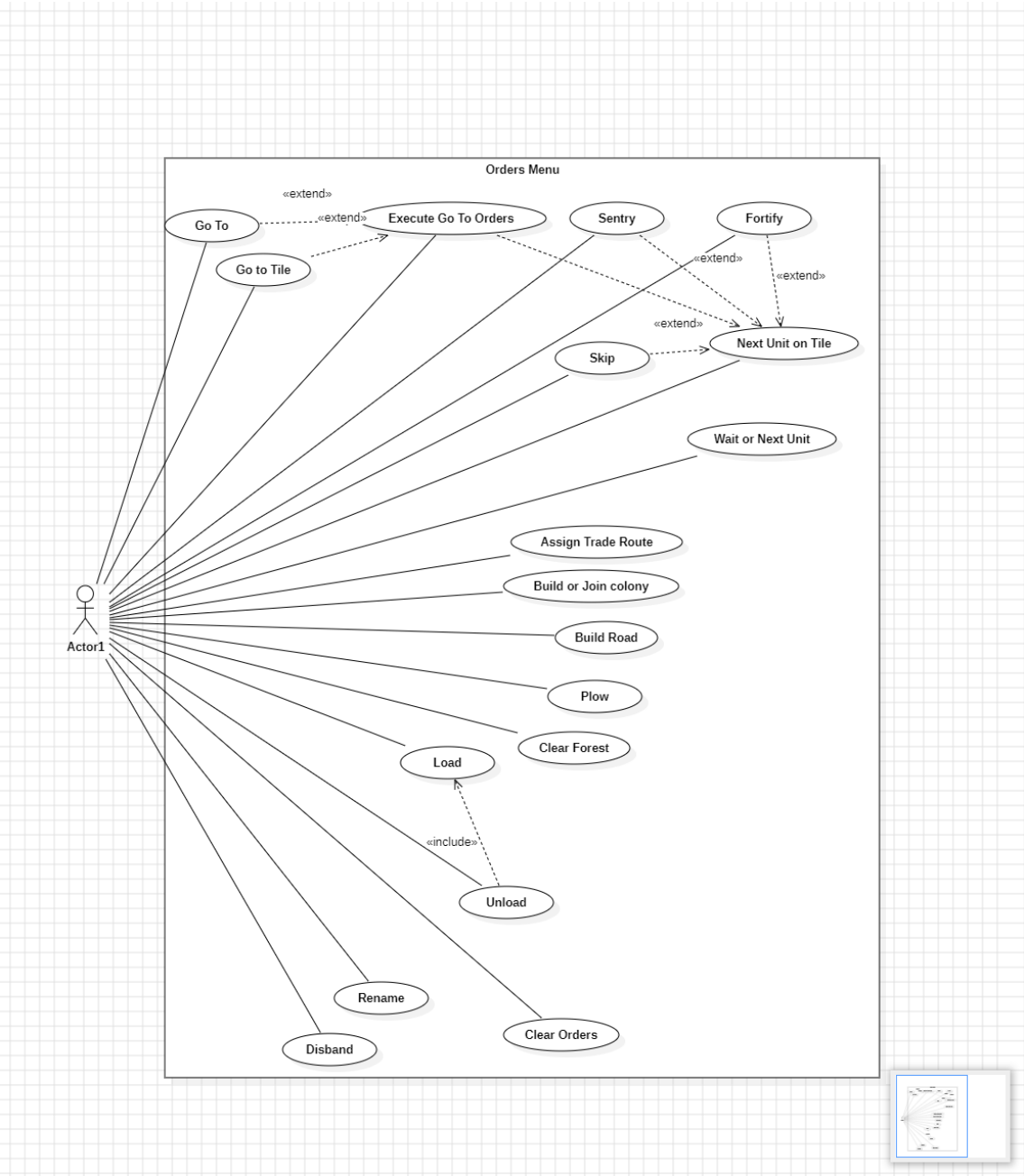
Within the confines of the codebase, there's an intriguing instance of primitive obsession. Take, for example, the utilization of a primitive type like an integer to encapsulate the concept of save game periods within the `autoSaveGame` method in `InGameController.java`. A more nuanced approach would 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.



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 functionality 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

For the 2nd User Story

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

We could not make this work due to time shortage.

3rd User Story

For the 3rd User Story

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