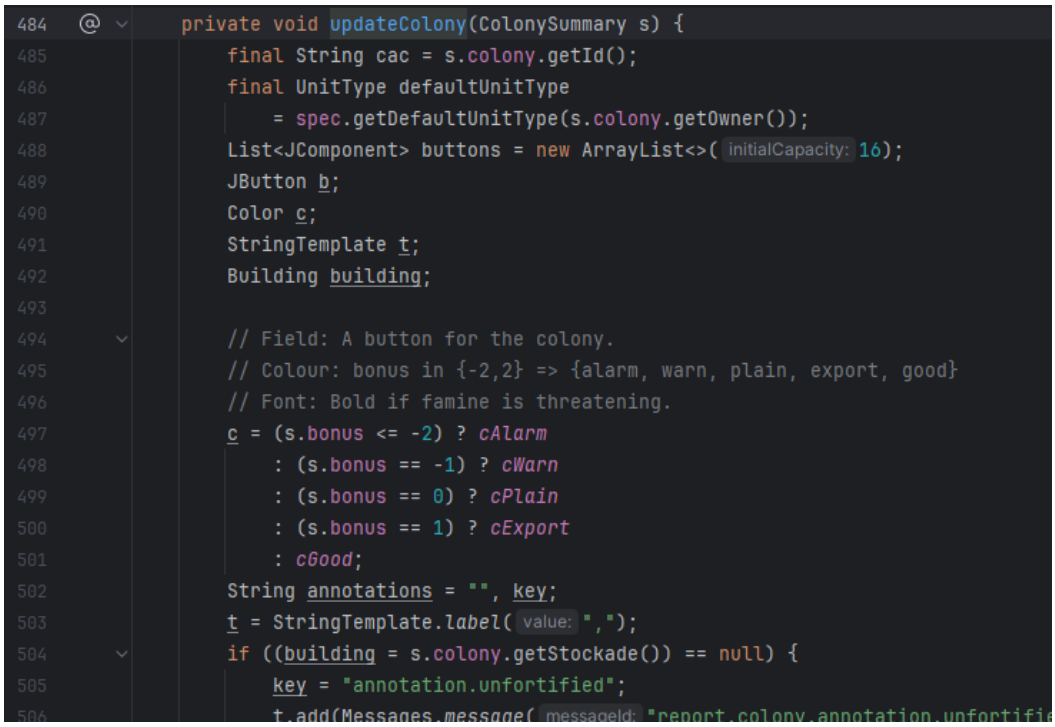
Code Smells

Long Method

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

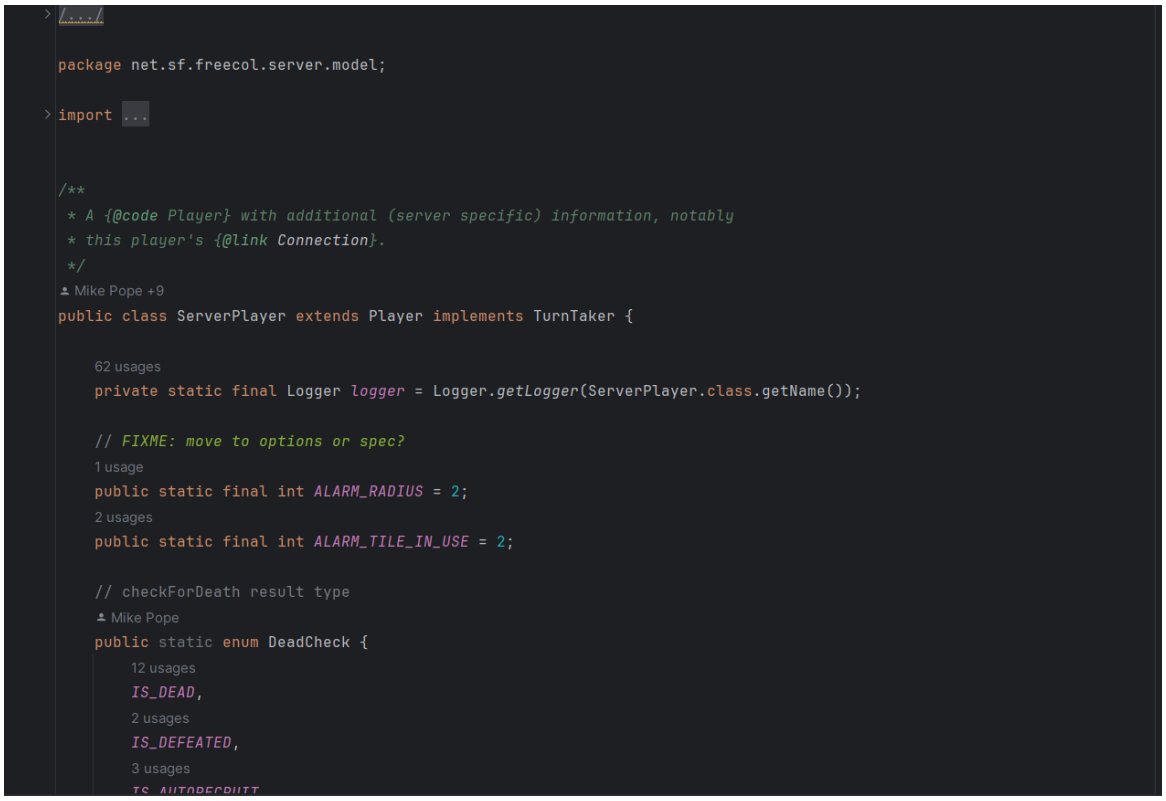


```
484  @  private void updateColony(ColonySummary s) {
485         final String cac = s.colony.getId();
486         final UnitType defaultUnitType
487             = spec.getDefaultUnitType(s.colony.getOwner());
488         List<JComponent> buttons = new ArrayList<>( initialCapacity: 16);
489         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.

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.

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.