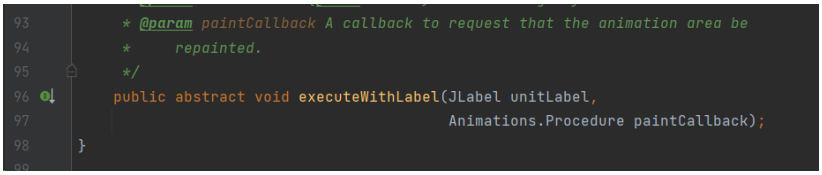
Design Patters

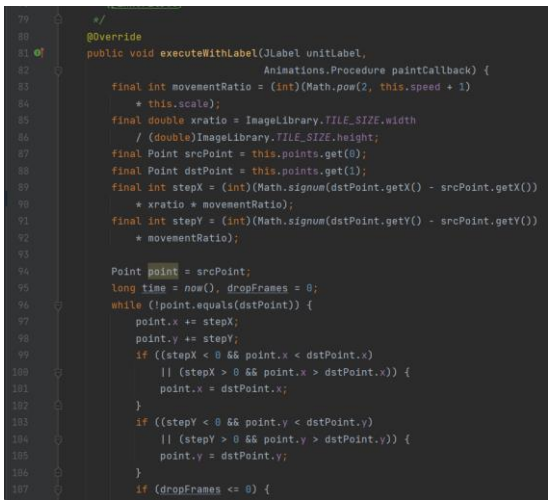
Design Pattern 1 – Template Method Pattern

Este padrão de design permite definir a estrutura de um algoritmo numa classe base, enquanto as subclasses façam as implementações específicas dos métodos dessa interface. Um exemplo no código do freeCol para este design pattern é na classe base Animation(net/sf/freecol/client/gui/animation/Animation) e nas sublclasses (net/sf/freecol/client/gui/animation/UnitImageAnimation e net/sf/freecol/client/gui/animation/UnitMoveAnimation) já que cada uma destas subclasses implementa o método executeWithLabel() de forma distinta nas diferentes subclasses.



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

Animation Class

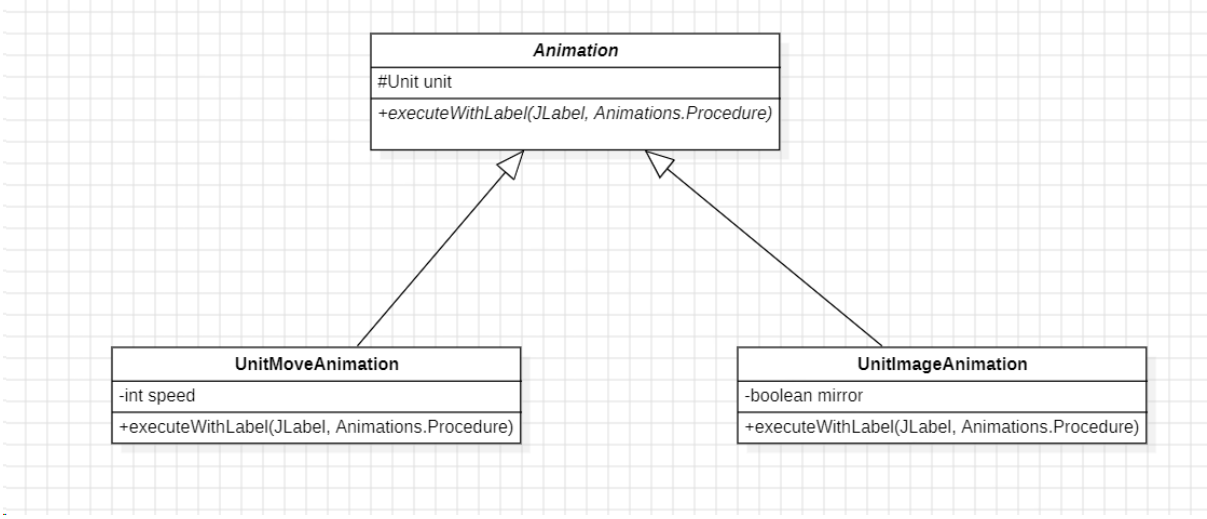


```
    */
    @Override
    public void executeWithLabel(JLabel unitLabel,
                                 Animations.Procedure paintCallback) {
        final int movementRatio = (int)(Math.pow(2, this.speed + 1)
            * this.scale);
        final double xratio = ImageLibrary.TILE_SIZE.width
            / (double)ImageLibrary.TILE_SIZE.height;
        final Point srcPoint = this.points.get(0);
        final Point dstPoint = this.points.get(1);
        final int stepX = (int)(Math.signum(dstPoint.getX() - srcPoint.getX())
            * xratio * movementRatio);
        final int stepY = (int)(Math.signum(dstPoint.getY() - srcPoint.getY())
            * movementRatio);

        Point point = srcPoint;
        long time = now(), dropFrames = 0;
        while (!point.equals(dstPoint)) {
            point.x += stepX;
            point.y += stepY;
            if ((stepX < 0 && point.x < dstPoint.x)
                || (stepX > 0 && point.x > dstPoint.x)) {
                point.x = dstPoint.x;
            }
            if ((stepY < 0 && point.y < dstPoint.y)
                || (stepY > 0 && point.y > dstPoint.y)) {
                point.y = dstPoint.y;
            }
            if (dropFrames <= 0) {
```

UnitMoveAnimation subclasse

```java
*/
public void executeWithLabel(JLabel unitLabel,
                             Animations.Procedure paintCallback) {
    final ImageIcon icon = (ImageIcon)unitLabel.getIcon();

    // Step through the animation, chaning the image
    for (AnimationEvent event : animation) {
        long time = System.nanoTime();
        if (event instanceof ImageAnimationEvent) {
            final ImageAnimationEvent ievent = (ImageAnimationEvent)event;
            Image image = ievent.getImage();
            if (mirror) {
                // FIXME: Add mirroring functionality to SimpleZippedAnimation
                image = createMirroredImage(image);
            }
            icon.setImage(image);
            paintCallback.execute(); // paint now

            // Time accounting
            time = ievent.getDurationInMs()
                - (System.nanoTime() - time) / 1000000;
            if (time > 0) Utils.delay(time, warning: "Animation delayed.");
        }
    }
}
```

UnitImageAnimation subclasse

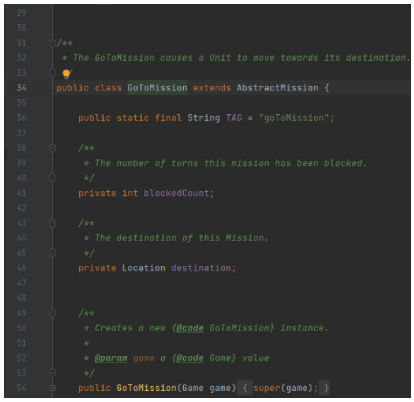


Design Pattern 2 – State Pattern

Este padrão de design é usado para modelar objetos que podem ter diferentes estados e permite que que esses objetos mudem de estado durante a execução do programa. Um exemplo deste padrão é a interface Mission (net.sf.freecol/common/model/mission/Mission) e os diferentes estados, nomeadamente, GoToMission(net.sf.freecol/common/model/mission/GoToMission) , CompoundMission (net/sf/freecol/common/model/mission/CompoundMission) e ImprovementMission (net/sf/freecol/common/model/mission/ImprovementMission).

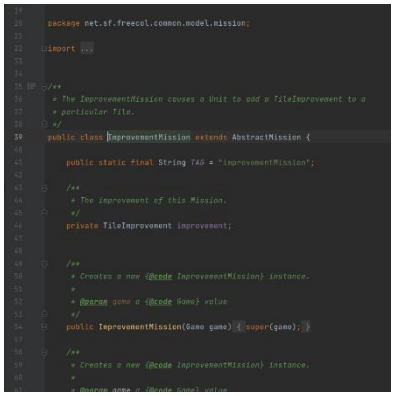
```
 * given to a {@link Unit}, such as the order to move to a certain
 * Tile, attack a certain Unit, or build a TileImprovement, for
 * example. Missions can be atomic, or combine a number of simpler
 * Missions.
 */
public interface Mission {

    public static enum MissionState {
        /**
         * Mission is in progress.
         */
        OK,
        /**
         * Mission has been completed.
         */
        COMPLETED,
        /**
         * Mission is temporarily blocked, e.g. by another Unit.
         */
        BLOCKED,
        /**
         * Mission has been aborted, e.g. because a target or
         * destination has been destroyed.
         */
        ABORTED
    };


    /**
     * Attempts to carry out the mission and returns an appropriate
     * MissionState.
```

Mission interface



```
/**
 * The GoToMission causes a Unit to move towards its destination.
 */
public class GoToMission extends AbstractMission {

    public static final String TAG = "goToMission";

    /**
     * The number of turns this mission has been blocked.
     */
    private int blockedCount;

    /**
     * The destination of this Mission.
     */
    private Location destination;


    /**
     * Creates a new {@code GoToMission} instance.
     *
     * @param game a {@code Game} value
     */
    public GoToMission(Game game) { super(game); }
```

GoToMission Classe

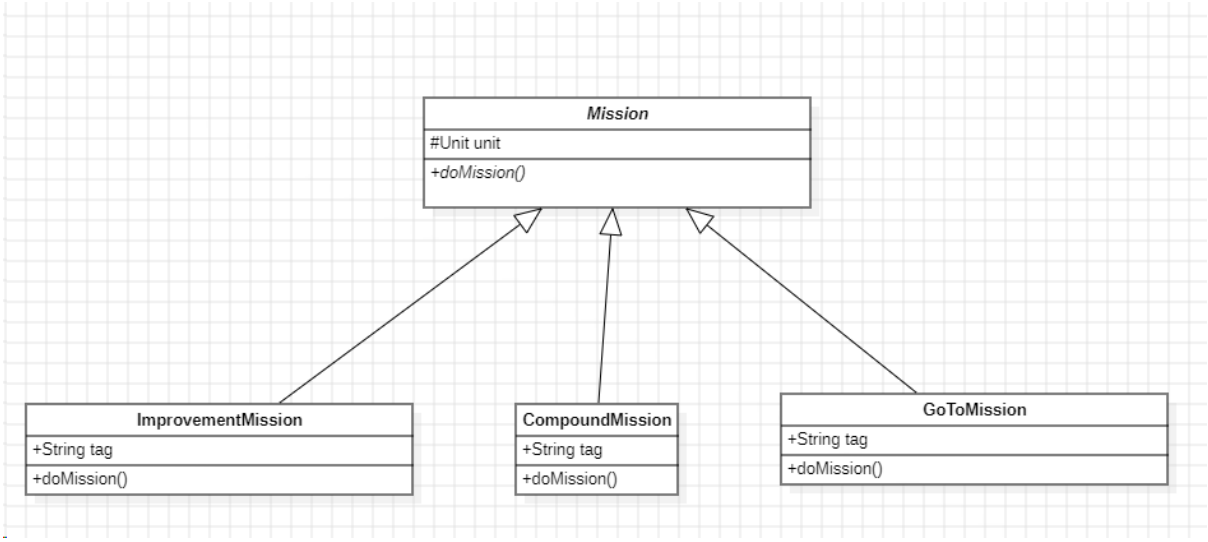


```
package net.sf.freecol.common.model.mission;

import ...


/**
 * The ImprovementMission causes a Unit to add a TileImprovement to a
 * particular Tile.
 */
public class ImprovementMission extends AbstractMission {

    public static final String TAG = "improvementMission";

    /**
     * The improvement of this Mission.
     */
    private TileImprovement improvement;


    /**
     * Creates a new {@code ImprovementMission} instance.
     *
     * @param game a {@code Game} value
     */
    public ImprovementMission(Game game) { super(game); }

    /**
     * Creates a new {@code ImprovementMission} instance.
     *
```

ImprovementMission Classe

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

import ...


/**
 * The CompoundMission provides a wrapper for several more basic
 * Missions that will be carried out in order.
 */
public class CompoundMission extends AbstractMission {

    public static final String TAG = "compoundMission";

    /**
     * The individual missions this CompoundMission wraps.
     */
    private List<Mission> missions;

    /**
     * The index of the current mission.
     */
    private int index;


    /**
     * Creates a new {@code CompoundMission} instance.
     *
     * @param game a {@code Game} value
     */
    public CompoundMission(Game game) { super(game); }
```

CompoundMission Classe



Design Pattern 3 – Façade pattern

Este padrão de design fornece uma interface unitária para um conjunto de classes, simplificando a interação com o subsistema, ocultando a complexidade interna e fornecendo uma única entrada para o cliente.O exemplo deste padrão é a interface Ownable (net/sf/freecol/common/model/Ownable), que é implementada pelas classes Colony (net/sf/freecol/common/model/Colony), Europe (net/sf/freecol/common/model/Europe) e TradeRoute(net/sf/freecol/common/model/TradeRoute) por exemplo. Depois na classe Player(net/sf/freecol/common/model/Player) temos 0 ou mais owners.