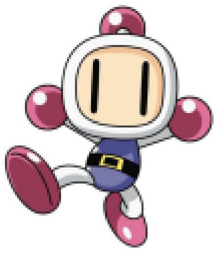


JBOMBERMAN

Project for the Methods of Programming course, 2nd semester, academic year 2022/2023.

AUTHOR
Valerio Gregori

SUMMARY

1 PROJECT SPECIFICATIONS

Firstly, below are the general project specifications chosen:

a) Design decisions related to each specification

b) Adopted design patterns, where and why

c) Use of streams

d) Other design and development notes.

Subsequently, the specifications related to the game in particular, in this case, those related to the project for a team composed of one person are:

1) User profile management, nickname, avatar, games played, won, lost, level...

2) Management of a complete game with at least two playable levels, two types of enemies with different graphics and gameplay behavior, with score management, lives, 3 power-ups, you won screen, game over, continue.

3) Appropriate use of MVC [1,2], Observer Observable, and other Design Patterns.

4) Adoption of Java Swing [2] or JavaFX [3] for the GUI.

5) Appropriate use of streams.

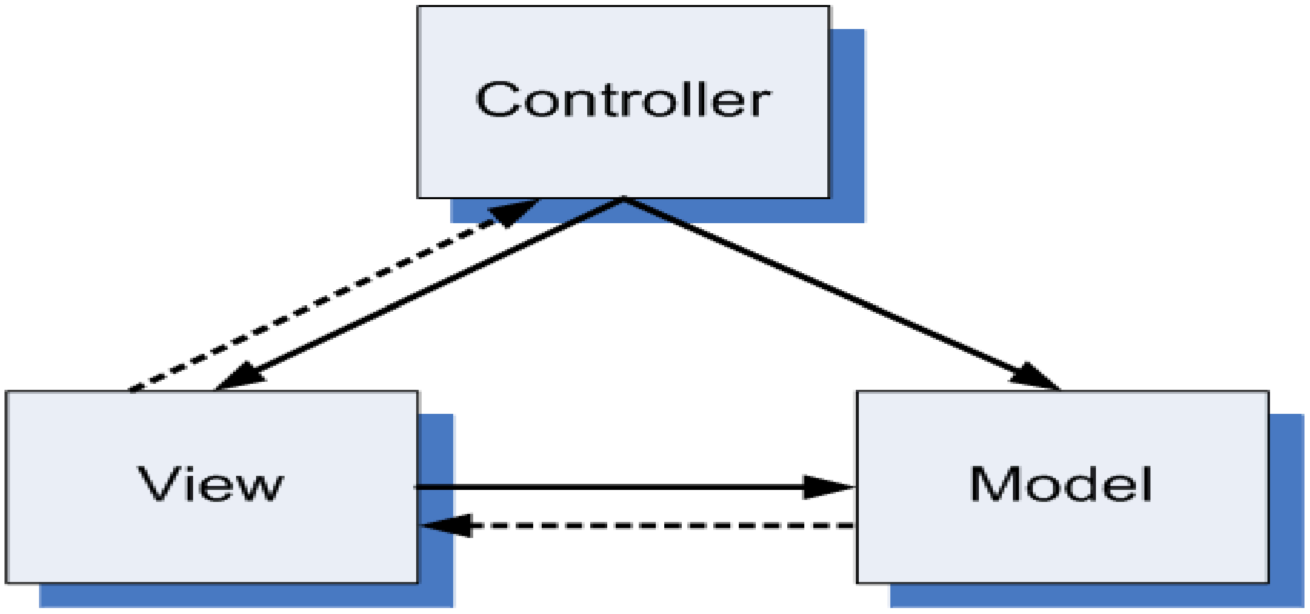
6) Reproduction of audio samples (see appendix AudioManager.Java).

7) Animations and special effects.

Regarding the latter design aspects, an overview of the game will be given first, then the project details will be discussed and each point will be commented on individually. As for the streams, they will only be described and commented on when necessary. Java Swing has been chosen for point 4, without any specific reason, since the development of the project is aimed only at the course held (therefore various risks such as unsupported libraries are not taken into consideration).

2 Initial Design and Use of MVC

In accordance with the MVC pattern, we immediately begin by defining the role of the three components that make up the basic pattern schema, as shown in the following figure:

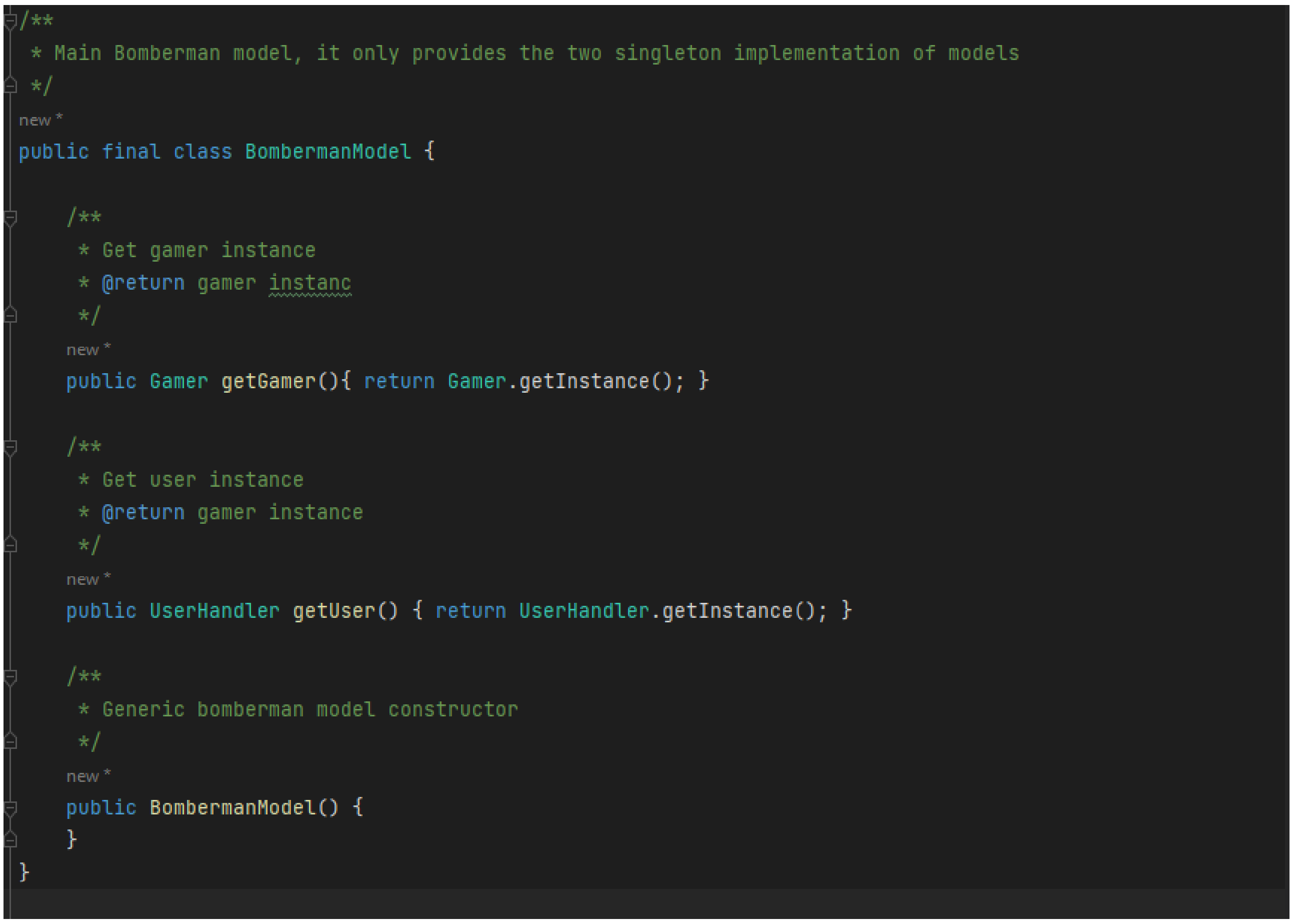


Let us begin by focusing on the Model, which as indicated by the arrow, will not depend on any other component. Therefore, in the model, we will find all the logic of the game, such as the callbacks associated with player movements during gameplay, or the creation of a new player in the menu. These callbacks, however, should not have any references to the controller or view, as only the latter will depend on the model to a certain extent, as they will use the callbacks described later.

Regarding the model, we have chosen to create a BombermanModel class that contains the two only instances of the class related to the game model and the user data class, respectively. These two classes can only be instantiated once at the beginning of the game (implementation of the Singleton pattern).

The decision to have a class that contains the two instances of the two main models instead of having a single class that inherits the methods of both models is partly due to the lack of multiple inheritance in Java. Furthermore, in general, one of the best practices in software design is to prefer composition over inheritance.

Below, we will describe both classes that make up the BombermanModel class. For simplicity, the UML diagram of the class is not provided, but we have included a screenshot of the class code. The class is declared as 'final' as it should not be extended. It has an empty constructor (added only for the completeness of JavaDoc), and two methods that return the two unique instances of the model.

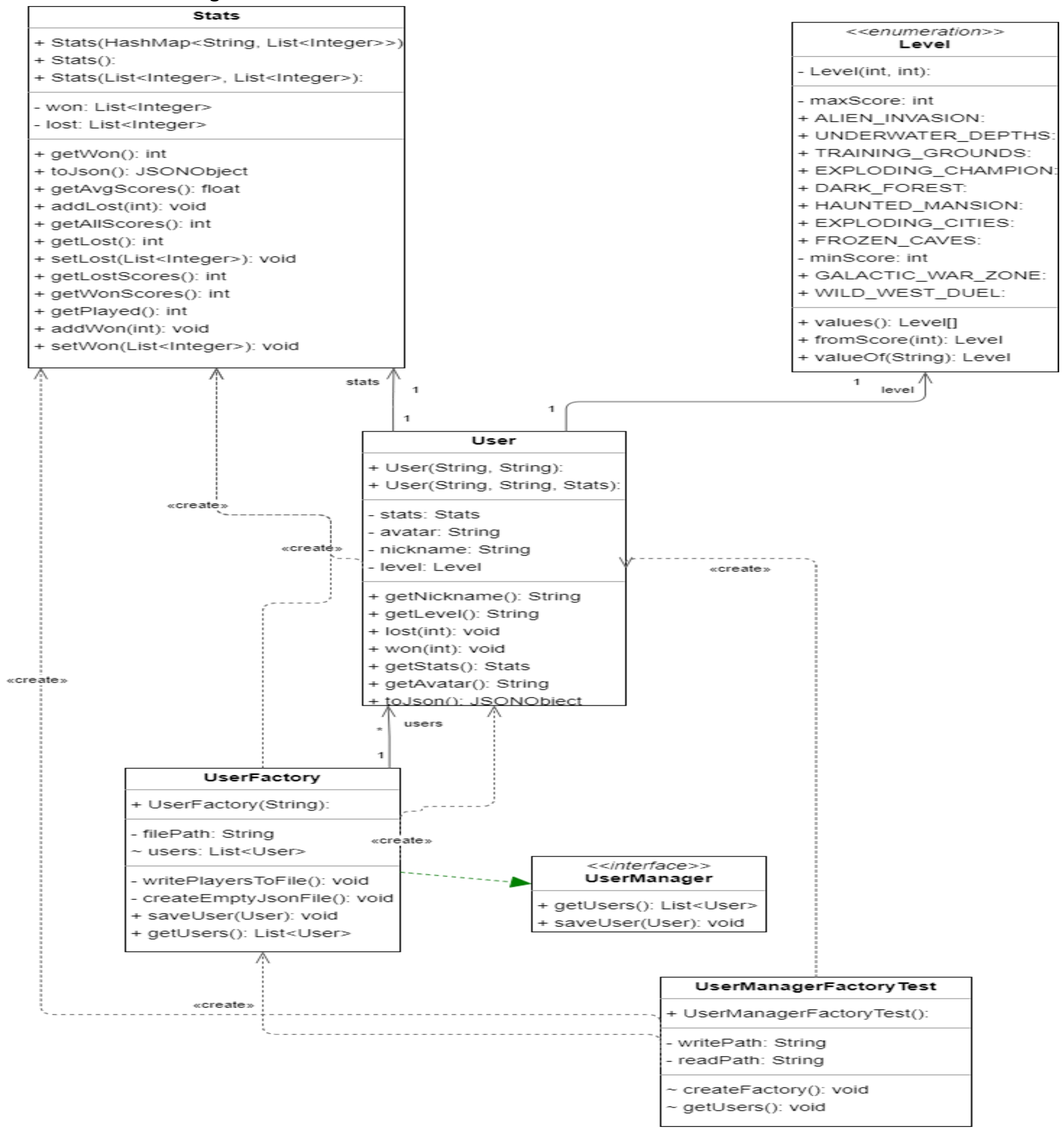


```
/**
 * Main Bomberman model, it only provides the two singleton implementation of models
 */
new *
public final class BombermanModel {

    /**
     * Get gamer instance
     * @return gamer instanc
     */
    new *
    public Gamer getGamer(){ return Gamer.getInstance(); }

    /**
     * Get user instance
     * @return gamer instance
     */
    new *
    public UserHandler getUser() { return UserHandler.getInstance(); }

    /**
     * Generic bomberman model constructor
     */
    new *
    public BombermanModel() {
    }
}
```

2.1 USER MODEL

Regarding the management of player data, the task is assigned to the model, except for the display and manipulation of data on the GUI, which are handled by the controller and view, respectively, as discussed later on.

Below is the class diagram for the user-related model:



We can immediately notice that this part already implements some of the proposed specifications, such as:

- The management of the avatar (the avatar can be chosen from four different colors, and during the game, the player will have a character of the chosen color).

- The level (10 different levels that can be achieved based on the score obtained throughout the games played).

- The nickname (the chosen name of the player; default names are also included and will be displayed randomly during character creation, but of course, players can choose any name they want).

- The save of the number of games won and lost.

The User class represents the chosen player and has fields for the nickname, level, and avatar, as per the specifications, as well as statistics represented by another class that stores won or lost games in a list. We have chosen to opt for a separate class for statistics for two reasons. Firstly, for a proper design, a class should have only one responsibility. Secondly, in subsequent versions, it may be useful to have convenience methods, such as average games or exporting statistics online in case of multiplayer, etc.

Both the User and Stats classes implement multiple constructors because when creating a new player, default parameters will be used, but when loading previously created players, a mechanism will be needed to create objects at runtime from existing data. To create this mechanism, there is a design pattern that defines its behavior, called the factory pattern, a creational pattern whose structure will be used frequently in the project.

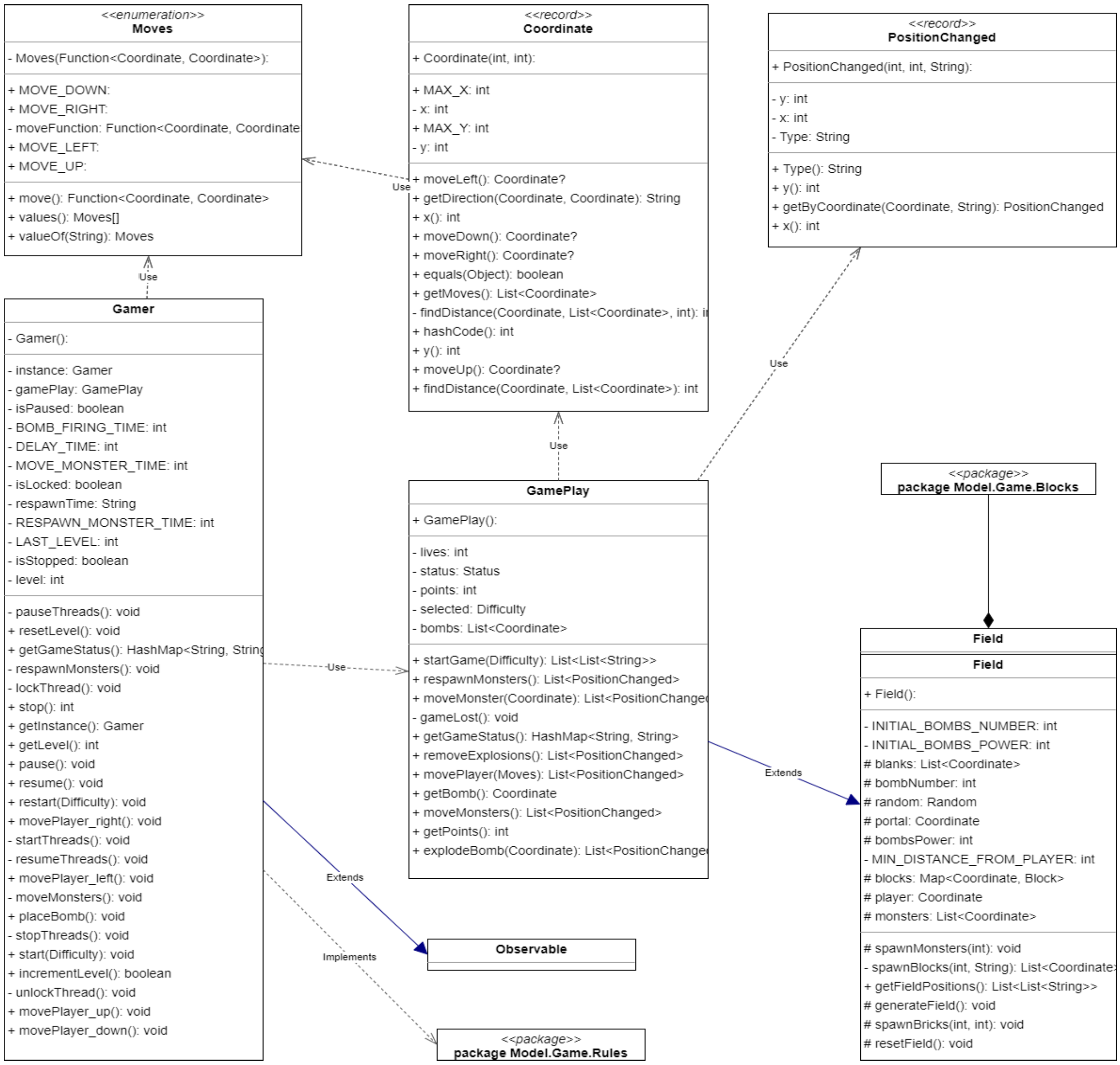
The factory is implemented by the User Factory class, which, to fulfill this task, implements the User Manager interface, which has only two methods (getUsers and saveUsers). These two methods represent a "contract" that must be signed by any class that manages data. In this way, with an eye to expanding the online game, where more significant quantities of data will be managed, data management can be migrated from a simple JSON file, as it is now, to an SQL database, while maintaining the same methods. By the way, the use of JSON can be maintained to have a local copy of the files.

To define the levels, an enum with several convenience methods was used.

Finally, the UserManagerFactoryTest class is only a test class that, with the help of the JUnit library, allowed testing the code related to data management. It would have been expensive to debug afterward during GUI execution.

2.2 GAME MODEL

The game model is, of course, more complex than player management, and will be described starting from the superclass. Below is the general UML of the game model:



<<enumeration>> Moves
- Moves(Function<Coordinate, Coordinate>):
+ MOVE_DOWN:
+ MOVE_RIGHT:
- moveFunction: Function<Coordinate, Coordinate
+ MOVE_LEFT:
+ MOVE_UP:
+ move(): Function<Coordinate, Coordinate>
+ values(): Moves[]
+ valueOf(String): Moves
<<record>> Coordinate
+ Coordinate(int, int):
+ MAX_X: int
- x: int
+ MAX_Y: int
- y: int
+ moveLeft(): Coordinate?
+ getDirection(Coordinate, Coordinate): String
+ x(): int
+ moveDown(): Coordinate?
+ moveRight(): Coordinate?
+ equals(Object): boolean
+ getMoves(): List<Coordinate>
- findDistance(Coordinate, List<Coordinate>, int): i
+ hashCode(): int
+ y(): int
+ moveUp(): Coordinate?
+ findDistance(Coordinate, List<Coordinate>): int
<<record>> PositionChanged
+ PositionChanged(int, int, String):
- y: int
- x: int
- Type: String
+ Type(): String
+ y(): int
+ getByCoordinate(Coordinate, String): PositionChanged
+ x(): int
Use
Use
Use
Use
Gamer
- Gamer():
- instance: Gamer
- gamePlay: GamePlay
- isPaused: boolean
- BOMB_FIRING_TIME: int
- DELAY_TIME: int
- MOVE_MONSTER_TIME: int
- isLocked: boolean
- respawnTime: String
- RESPAWN_MONSTER_TIME: int
- LAST_LEVEL: int
- isStopped: boolean
- level: int
- pauseThreads(): void
+ resetLevel(): void
+ getGameStatus(): HashMap<String, String
- respawnMonsters(): void
- lockThread(): void
+ stop(): int
+ getInstance(): Gamer
+ getLevel(): int
+ pause(): void
+ resume(): void
+ restart(Difficulty): void
+ movePlayer_right(): void
- startThreads(): void
- resumeThreads(): void
+ movePlayer_left(): void
- moveMonsters(): void
+ placeBomb(): void
- stopThreads(): void
+ start(Difficulty): void
+ incrementLevel(): boolean
- unlockThread(): void
+ movePlayer_up(): void
+ movePlayer_down(): void
Use
GamePlay
+ GamePlay():
- lives: int
- status: Status
- points: int
- selected: Difficulty
- bombs: List<Coordinate>
+ startGame(Difficulty): List<List<String>>
+ respawnMonsters(): List<PositionChanged>
+ moveMonster(Coordinate): List<PositionChange
- gameLost(): void
+ getGameStatus(): HashMap<String, String>
+ removeExplosions(): List<PositionChanged>
+ movePlayer(Moves): List<PositionChanged>
+ getBomb(): Coordinate
+ moveMonsters(): List<PositionChanged>
+ getPoints(): int
+ explodeBomb(Coordinate): List<PositionChange
<<package>> package Model.Game.Blocks
Extends
Field
Field
+ Field():
- INITIAL_BOMBS_NUMBER: int
- INITIAL_BOMBS_POWER: int
# blanks: List<Coordinate>
# bombNumber: int
# random: Random
# portal: Coordinate
# bombsPower: int
- MIN_DISTANCE_FROM_PLAYER: int
# blocks: Map<Coordinate, Block>
# player: Coordinate
# monsters: List<Coordinate>
# spawnMonsters(int): void
- spawnBlocks(int, String): List<Coordinate
+ getFieldPositions(): List<List<String>>
# generateField(): void
# spawnBricks(int, int): void
# resetField(): void
Extends
Observable
Implements
<<package>> package Model.Game.Rules

Let's start with the "Gamer" class, which will manage game threads, a further reason to implement the singleton pattern. Game threads will serve to:

- Make bombs explode after a defined interval,

- Respawn monsters when a certain time expires,

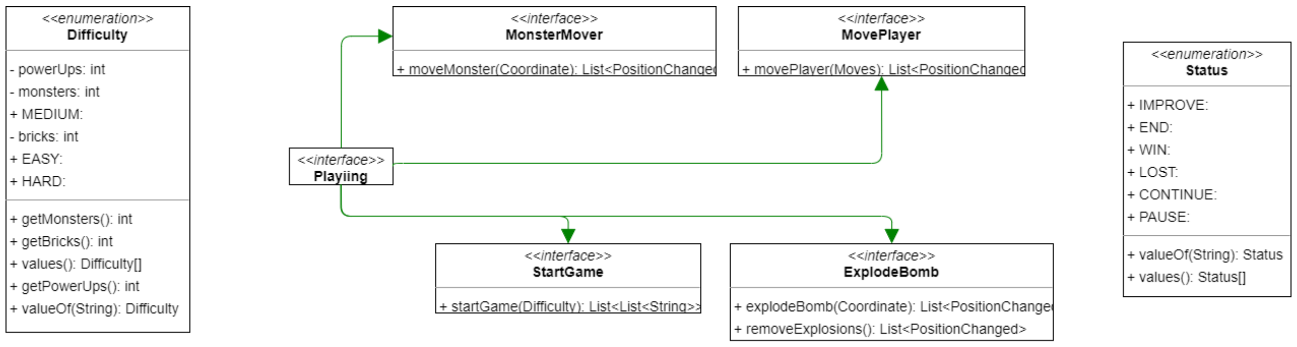
- Move monsters,

In addition, it will stop threads when the game is paused and reactivate them when the game resumes.

It will also manage the game, move to the next level, declare victory upon completion of the last level, and declare defeat when the lives run out.

The private methods lockThread and unlockThread implement the semaphore for thread management. When a thread is modifying a list, such as moving monsters or exploding bombs, it is essential that they are executed one at a time and not temporarily to prevent difficult-to-handle exceptions for the Java Virtual Machine (such as "concurrent modification exception" in the case of simultaneous modifications of lists by two threads).

Furthermore, the gamer will implement a series of rules that will be the APIs used by the controller to "stimulate" the model. Below is the interface diagram present in the "Rules" package:

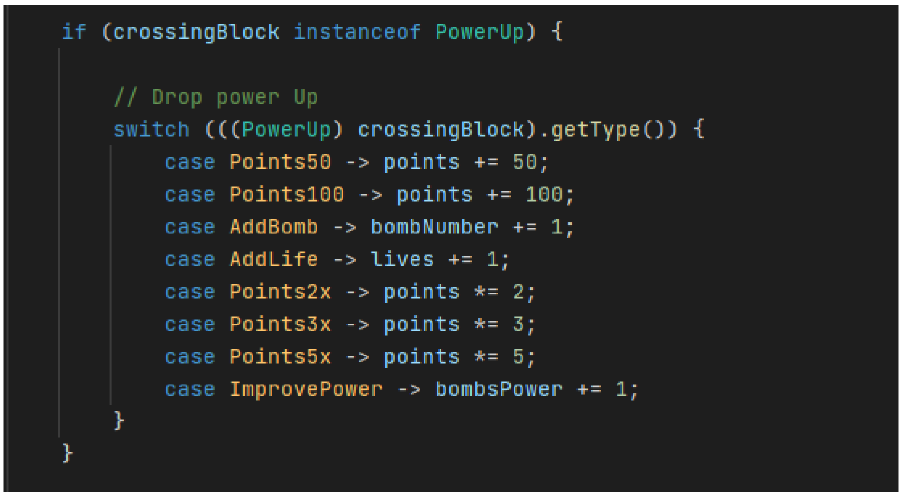


The "playing" interface describes the above-mentioned APIs, specifically the functions for starting the game, exploding bombs and moving monsters and the player.

The "Difficulty" enumeration describes the selected difficulty during the start of the game (which could be progressively increased during subsequent levels; currently it remains unchanged), and the difficulty management strategy. Here, a "DifficultyStrategy" class could be implemented (implementation of the Strategy design pattern) to define different behaviors for managing the game difficulty. This is not currently necessary, as only one behavior has been defined. Specifically, increasing difficulty levels correspond to a greater number of monsters and blocks to be destroyed. A smaller number of blocks will contain power-ups.

The "Status" enumeration represents the "protocol" towards the observer (implementation of the Observer pattern/Observable as required), which will react to the received status.

The gamer has an instance of the "GamePlay" class, which extends "Field". Starting from the top, we have "GamePlay", which implements the mid-level methods of the game (between the APIs towards the observer and the game grid, which represents the lowest layer of the game). Among these, it returns the game grid, returns changes in case of external stimuli (explosions, player moves, etc.), maintains the number of bombs, points, lives and power-ups. As shown in the following photo, it manages power-ups:

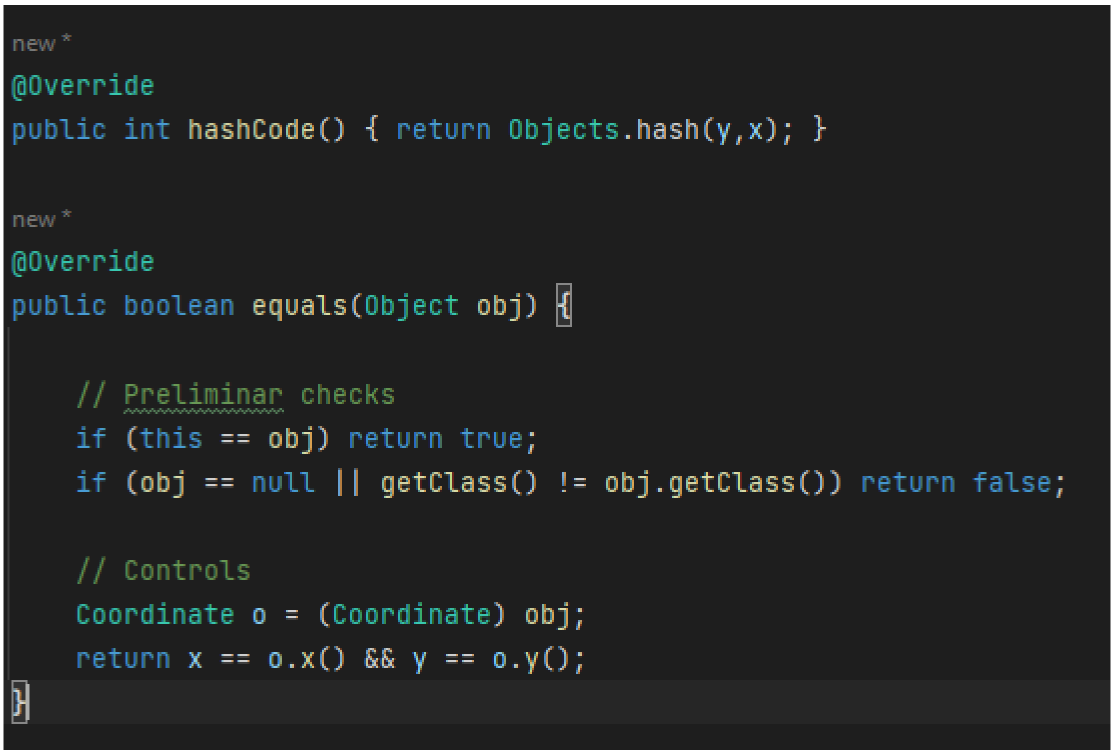


```
if (crossingBlock instanceof PowerUp) {

    // Drop power Up
    switch (((PowerUp) crossingBlock).getType()) {
        case Points50 -> points += 50;
        case Points100 -> points += 100;
        case AddBomb -> bombNumber += 1;
        case AddLife -> lives += 1;
        case Points2x -> points *= 2;
        case Points3x -> points *= 3;
        case Points5x -> points *= 5;
        case ImprovePower -> bombsPower += 1;
    }
}
```

In the above figure, the code that manages the power-ups is shown. There are eight power-ups (as required), although five are related to points, while the others grant an additional bomb, an extra life, and an increased bomb power during explosions.

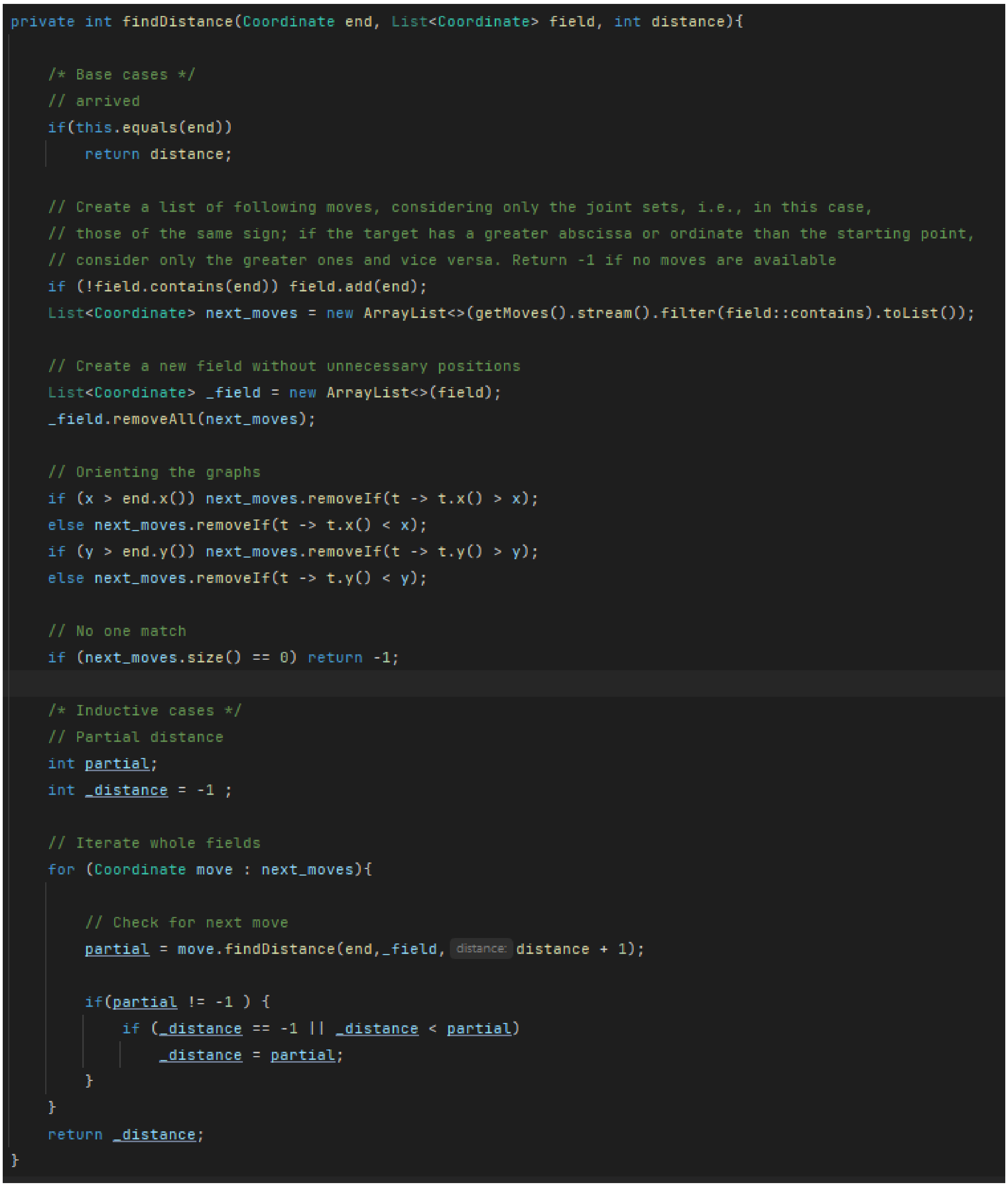
The "GamePlay" class has two "dependency" arrows, excluding the one that extends "Field". One points towards "Coordinate" and the other towards "PositionChanged". The two classes have a very common structure and behavior, but they have been kept separate so that the "Coordinate" class remains confined in the Model world while the "PositionChanged" class is used to notify the Observers.

Let's briefly discuss the "Coordinate" class, which provides a hash mechanism calculated based on the x and y coordinates:



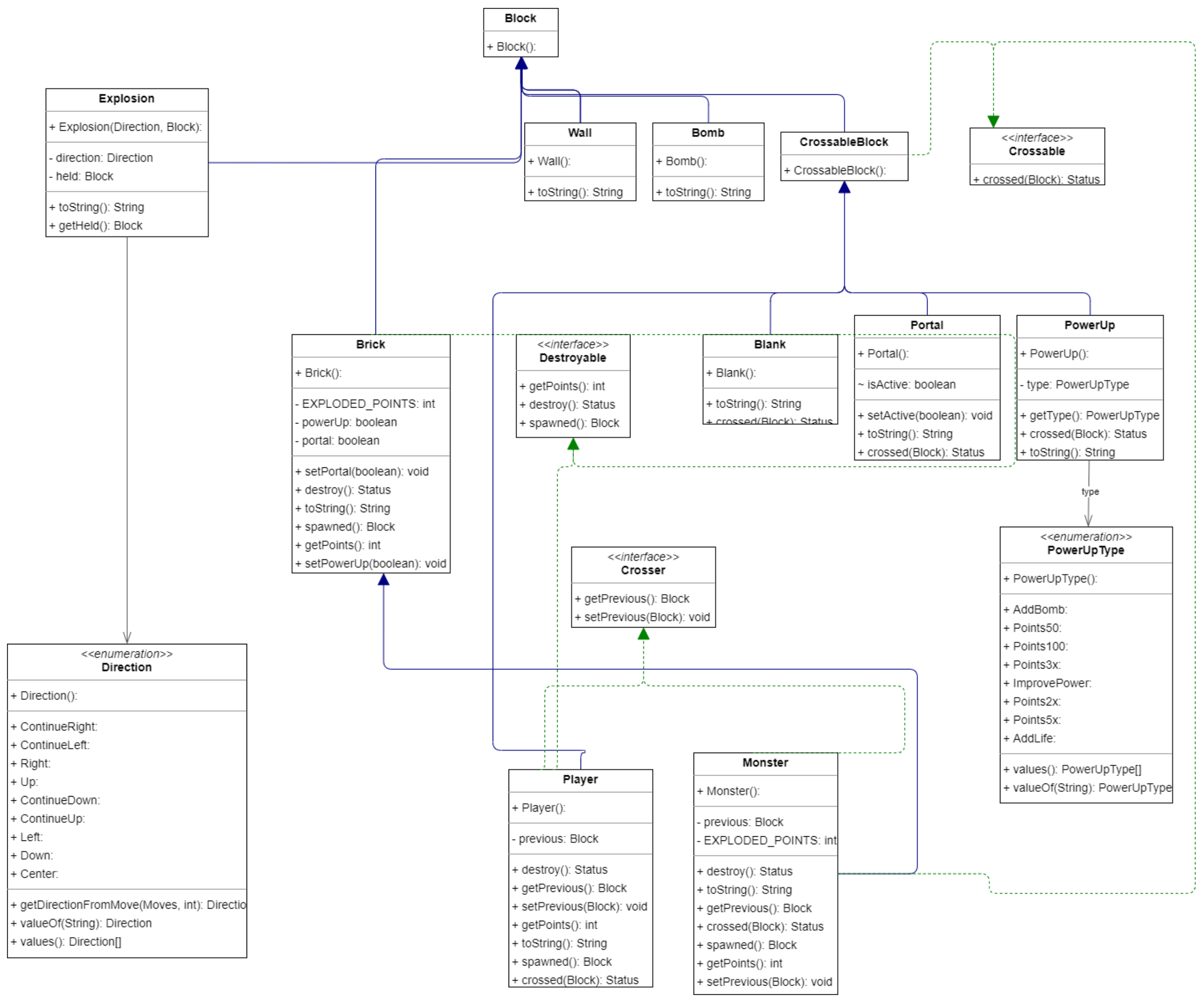
```
new *
@Override
public int hashCode() { return Objects.hash(y,x); }

new *
@Override
public boolean equals(Object obj) {

    // Preliminar checks
    if (this == obj) return true;
    if (obj == null || getClass() != obj.getClass()) return false;

    // Controls
    Coordinate o = (Coordinate) obj;
    return x == o.x() && y == o.y();
}
```

The reason will be explained in detail when discussing the "Field" class. The "Coordinate" class also has functions associated with movements (which return the coordinates above, below, or to the side of another coordinate) and it has a public and a private recursive method to calculate the distance between two coordinates. This method is actually used in generating blocks at a certain distance from the player. This prevents the player from finding themselves in unpleasant situations, such as games with monsters and blocks too close together, making it impossible to throw bombs and continue the game. Below is a screenshot of the private recursive method (with the public method initializing the distance parameter to 0). This method represents one of several points (also within the "Coordinate" class) where streams are used as required by the specifications. The "getMoves()" method also makes use of this functionality.



```java
private int findDistance(Coordinate end, List<Coordinate> field, int distance){

    /* Base cases */
    // arrived
    if(this.equals(end))
        return distance;

    // Create a list of following moves, considering only the joint sets, i.e., in this case,
    // those of the same sign; if the target has a greater abscissa or ordinate than the starting point,
    // consider only the greater ones and vice versa. Return -1 if no moves are available
    if (!field.contains(end)) field.add(end);
    List<Coordinate> next_moves = new ArrayList<>(getMoves().stream().filter(field::contains).toList());

    // Create a new field without unnecessary positions
    List<Coordinate> _field = new ArrayList<>(field);
    _field.removeAll(next_moves);

    // Orienting the graphs
    if (x > end.x()) next_moves.removeIf(t -> t.x() > x);
    else next_moves.removeIf(t -> t.x() < x);
    if (y > end.y()) next_moves.removeIf(t -> t.y() > y);
    else next_moves.removeIf(t -> t.y() < y);

    // No one match
    if (next_moves.size() == 0) return -1;

    /* Inductive cases */
    // Partial distance
    int partial;
    int _distance = -1 ;

    // Iterate whole fields
    for (Coordinate move : next_moves){

        // Check for next move
        partial = move.findDistance(end,_field, distance: distance + 1);

        if(partial != -1 ) {
            if (_distance == -1 || _distance < partial)
                _distance = partial;
        }
    }
    return _distance;
}
```

Lastly, let's talk about the "Field" class, which extends the "GamePlay" class so that it can modify the "Blocks". Let's first talk about the "Blocks" contained in the Blocks package, whose UML diagram is depicted below:



Before discussing the above-mentioned diagram, it is important to make a note. To create the game field, it was chosen to create a matrix of each individual block, whether a player, wall, bomb, explosion, etc. Alternatively, and in hindsight, it would have been the better choice in terms of efficiency and user experience to only draw some blocks over the static game field. This would certainly provide better graphics and consumption of fewer resources, but it would be less flexible and customizable in managing levels. However, the chosen path, while not entirely correct, has brought many more benefits for learning purposes than the better option (in this context, it can be said that it was worth it).

So, the game field is a matrix of generic blocks, which will be saved not on a matrix but on a map where, for the key, we have a certain coordinate and for the value, the associated block (hence the reason for implementing the "hashcode" and "equals" methods of the "Coordinate" record).

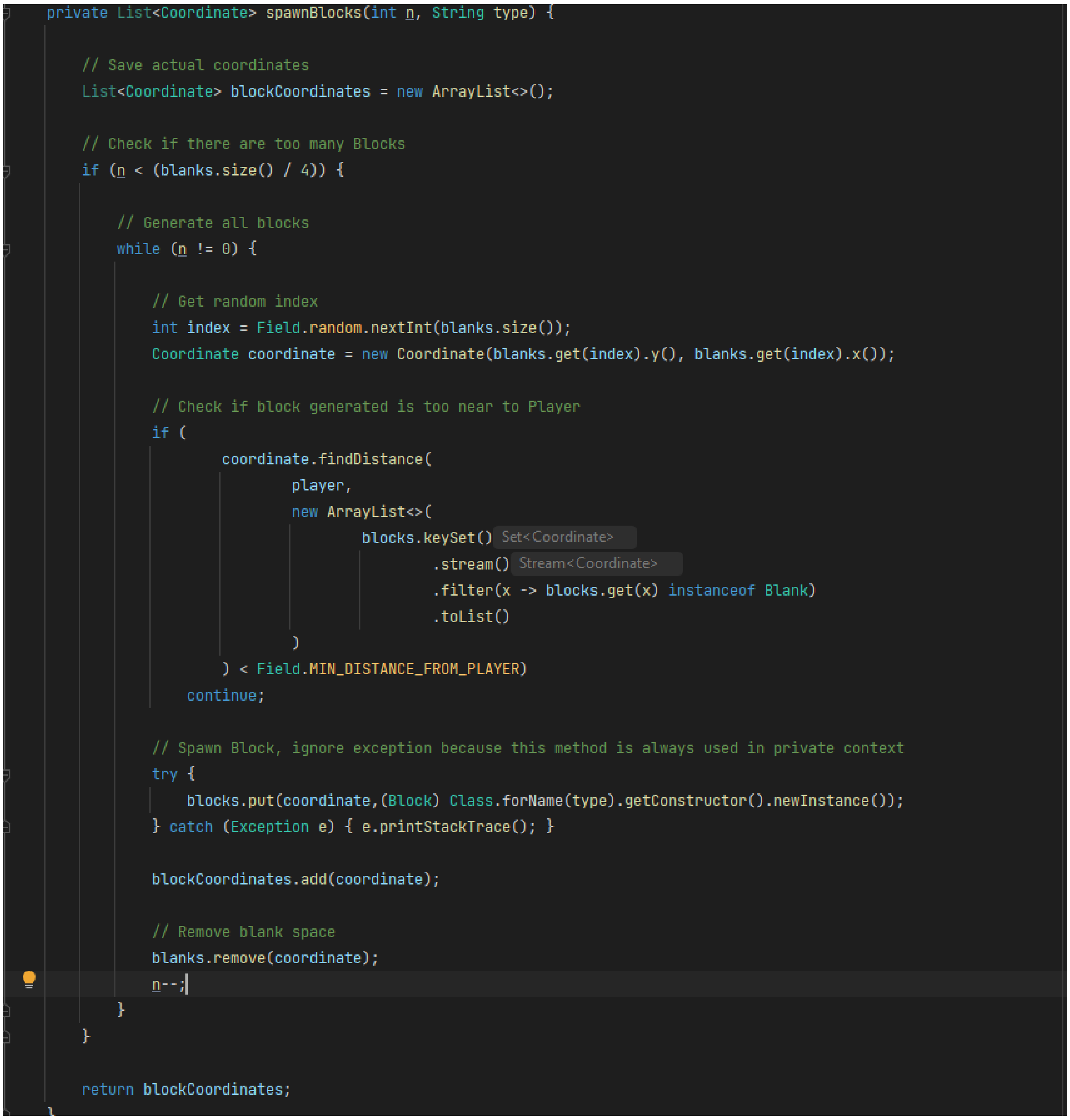
We have the following classes and interfaces for the blocks:

- "Block": the generic abstract class
- "Wall": representing the walls of the game
- "Bomb": representing bombs
- "Explosion": depicting explosions, which can have a direction (an enum inserted with the sole purpose of avoiding extra code for direction management in the View and redundant code)
- "Brick": blocks that can be destroyed and contain power-ups or the portal that determines the end of the game if all monsters have been killed
- "Monster"
- "Player"
- "Portal"
- "Power-up"
- "Blank": representing the empty block.

Additionally, there are several interfaces that specialize the block. For example, the block can be destructible (which can respawn another block, such as a power-up), navigable, and can go through another block.

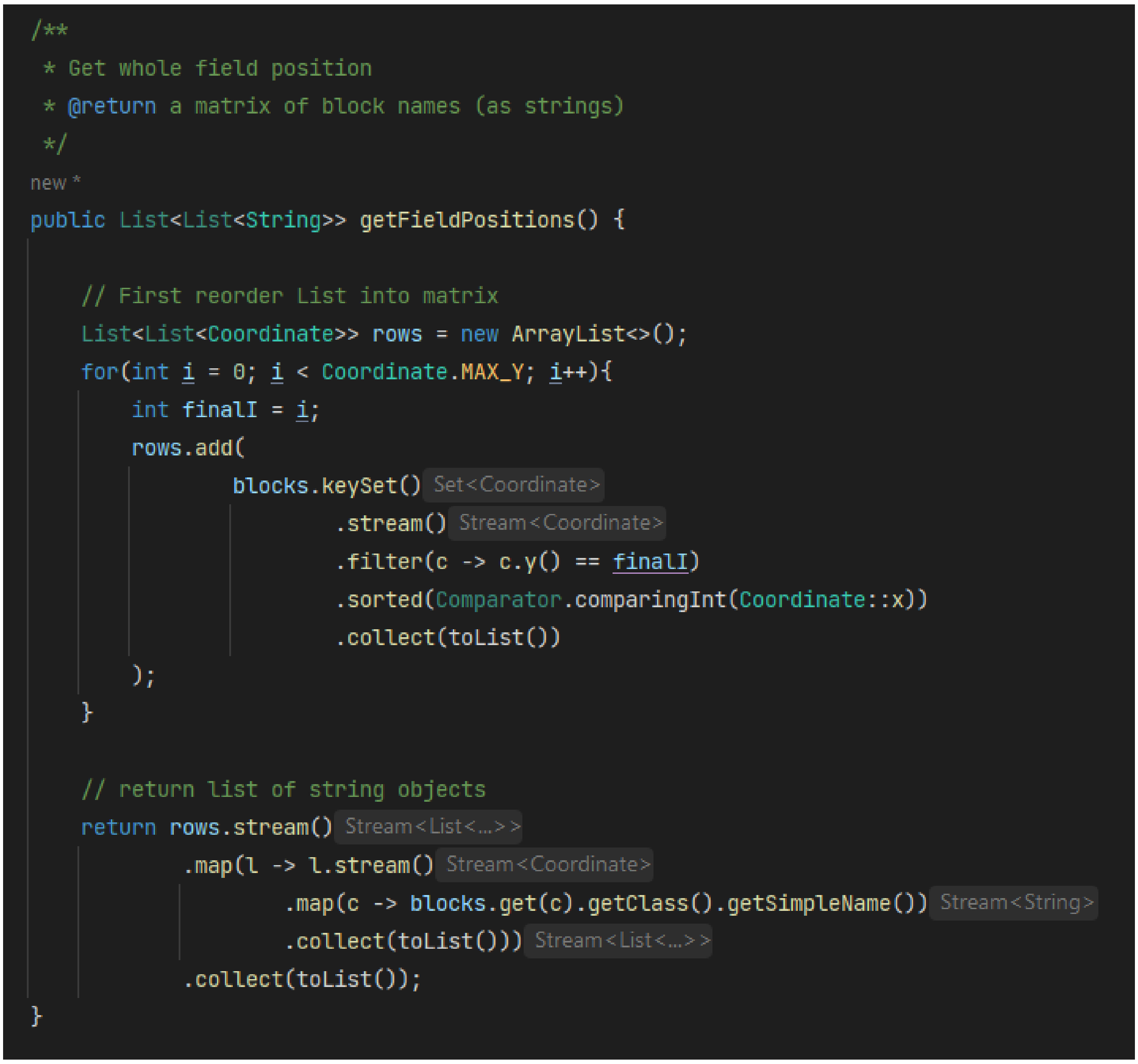
The creation of blocks also uses the Factory pattern and reflection, as a single method is defined to create blocks of the specified type. With reflection, it is possible to instantiate objects created from the class name, and in this case, the pattern is referred to as Abstract Factory.

Below the code that also uses streams to create blocks using reflection:



```java
private List<Coordinate> spawnBlocks(int n, String type) {

    // Save actual coordinates
    List<Coordinate> blockCoordinates = new ArrayList<>();

    // Check if there are too many Blocks
    if (n < (blanks.size() / 4)) {

        // Generate all blocks
        while (n != 0) {

            // Get random index
            int index = Field.random.nextInt(blanks.size());
            Coordinate coordinate = new Coordinate(blanks.get(index).y(), blanks.get(index).x());

            // Check if block generated is too near to Player
            if (
                    coordinate.findDistance(
                            player,
                            new ArrayList<>(
                                    blocks.keySet() Set<Coordinate>
                                            .stream() Stream<Coordinate>
                                            .filter(x -> blocks.get(x) instanceof Blank)
                                            .toList()
                            )
                    ) < Field.MIN_DISTANCE_FROM_PLAYER)
                continue;

            // Spawn Block, ignore exception because this method is always used in private context
            try {
                blocks.put(coordinate,(Block) Class.forName(type).getConstructor().newInstance());
            } catch (Exception e) { e.printStackTrace(); }

            blockCoordinates.add(coordinate);

            // Remove blank space
            blanks.remove(coordinate);
            n--;
        }
    }

    return blockCoordinates;
```

The blocks are created at the beginning of the game to generate the field. The latter is passed, initially, as a matrix of strings to the observer that creates the game field (subsequently, only changes to the newly created field will be notified). Please check the screenshot of the method that uses streams below:

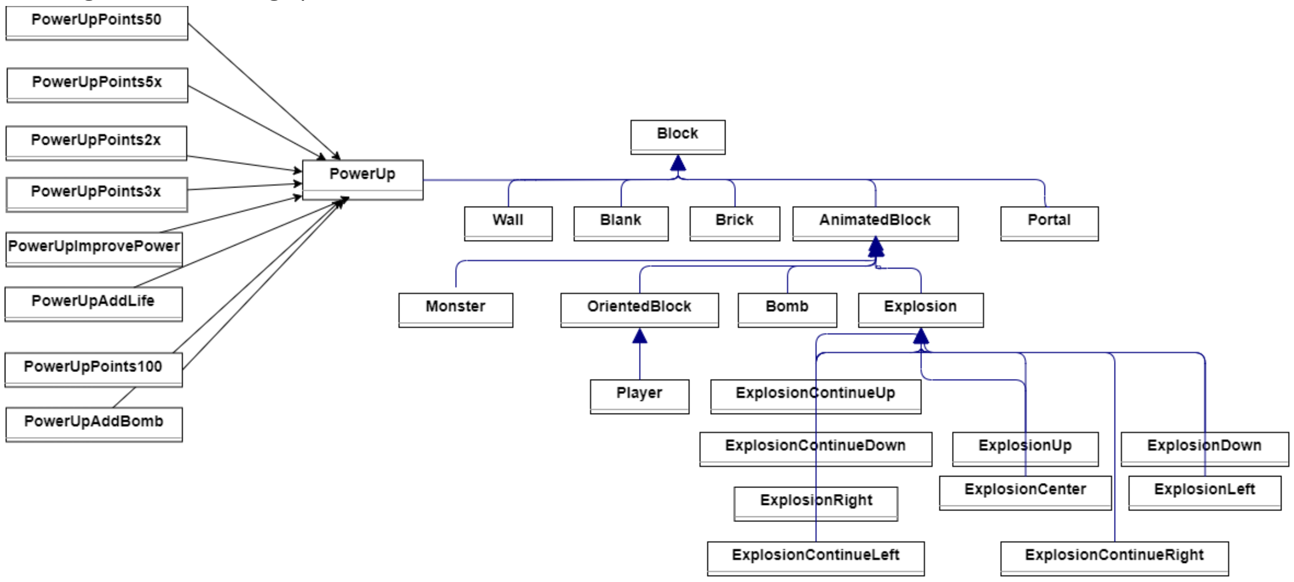


```java
/**
 * Get whole field position
 * @return a matrix of block names (as strings)
 */
new *
public List<List<String>> getFieldPositions() {

    // First reorder List into matrix
    List<List<Coordinate>> rows = new ArrayList<>();
    for(int i = 0; i < Coordinate.MAX_Y; i++){
        int finalI = i;
        rows.add(
                blocks.keySet() Set<Coordinate>
                        .stream() Stream<Coordinate>
                        .filter(c -> c.y() == finalI)
                        .sorted(Comparator.comparingInt(Coordinate::x))
                        .collect(toList())
        );
    }

    // return list of string objects
    return rows.stream() Stream<List<...>>
            .map(l -> l.stream() Stream<Coordinate>
                    .map(c -> blocks.get(c).getClass().getSimpleName()) Stream<String>
                    .collect(toList())) Stream<List<...>>
            .collect(toList());
}
```

3 VIEW

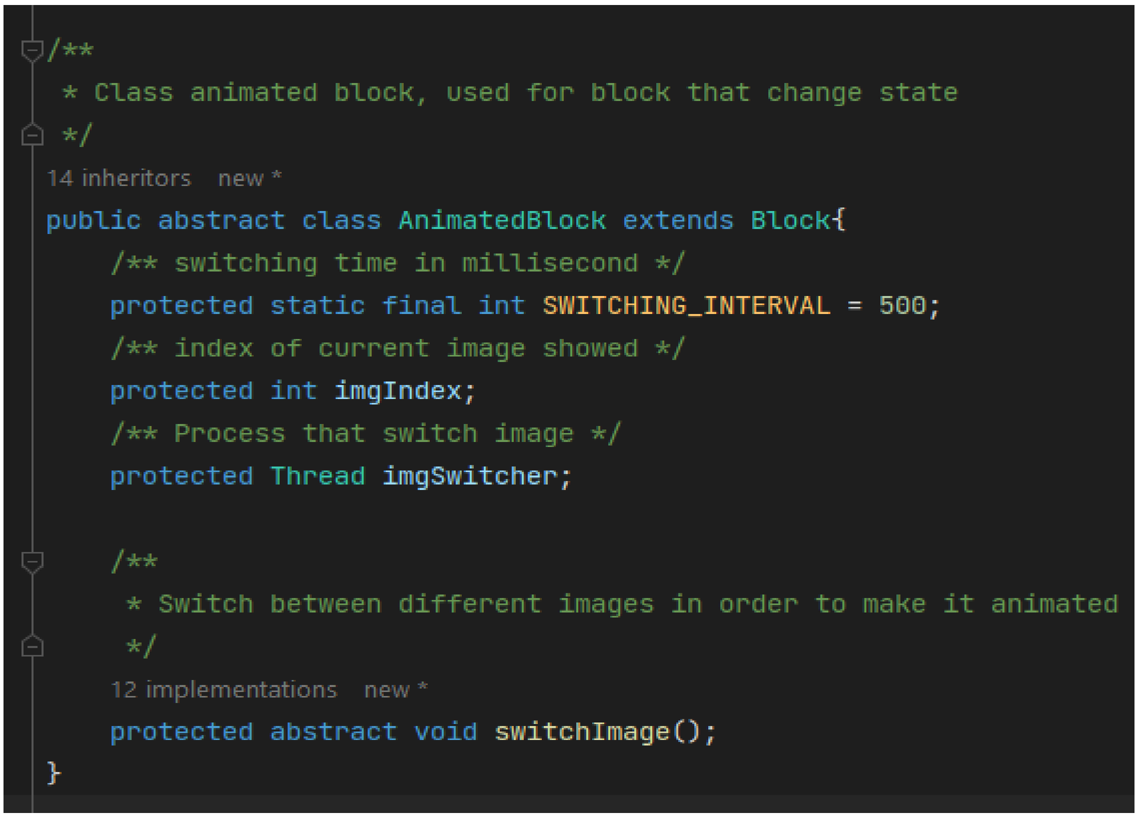
3.1 GAME VIEW (FIELD)

As with the Model, for convenience, the View, which depends heavily on it, will also have the same blocks as classes with the same name (of course, in a different package). In this way, they can be reconstructed starting from the strings passed from the last method seen in the Model section.

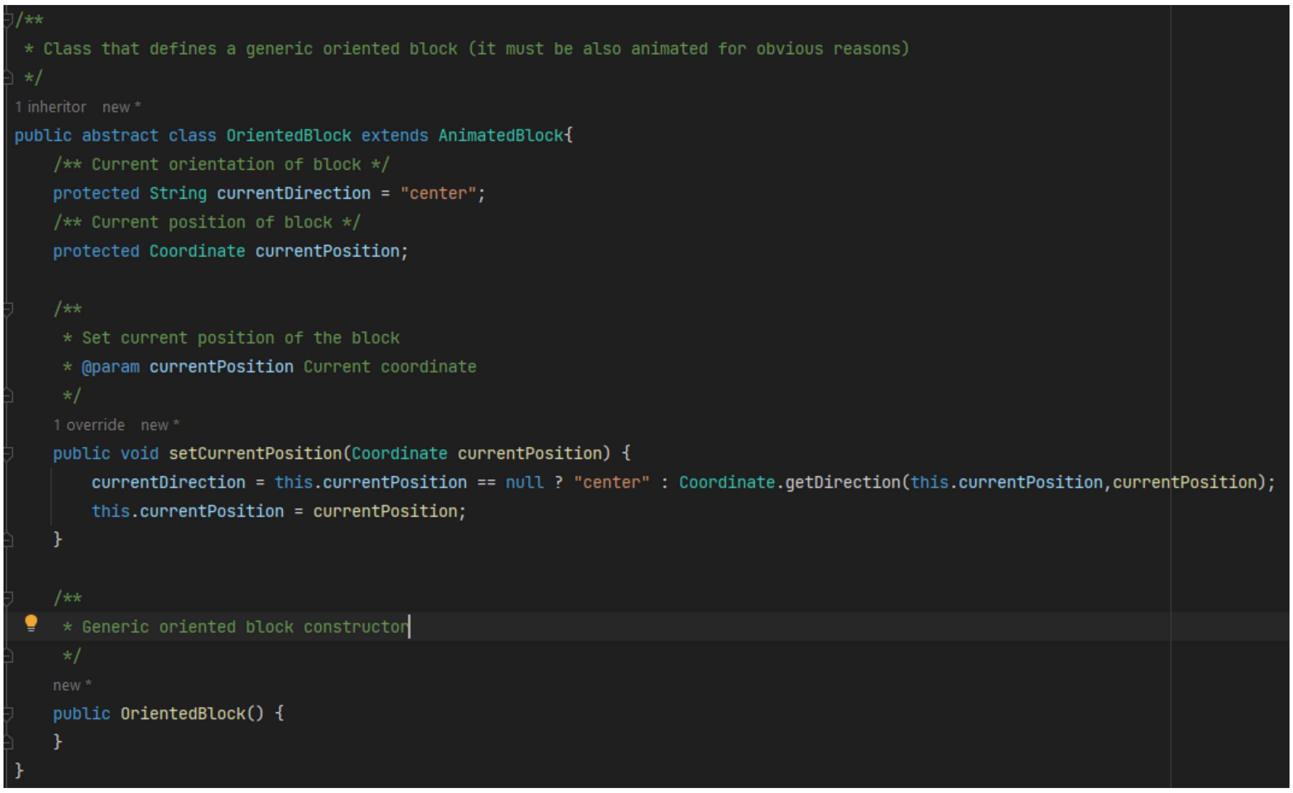


In this case, the differences are that, in addition to having a class for each type of block (associated with a specific image for ease of creation), some blocks can also be animated or oriented. This results in the changing images of the individual objects. Of course, AnimatedBlock and OrientedBlock are abstract classes, as are GenericPowerUp and GenericExplosion.

Here are the implementations of these two abstract classes:

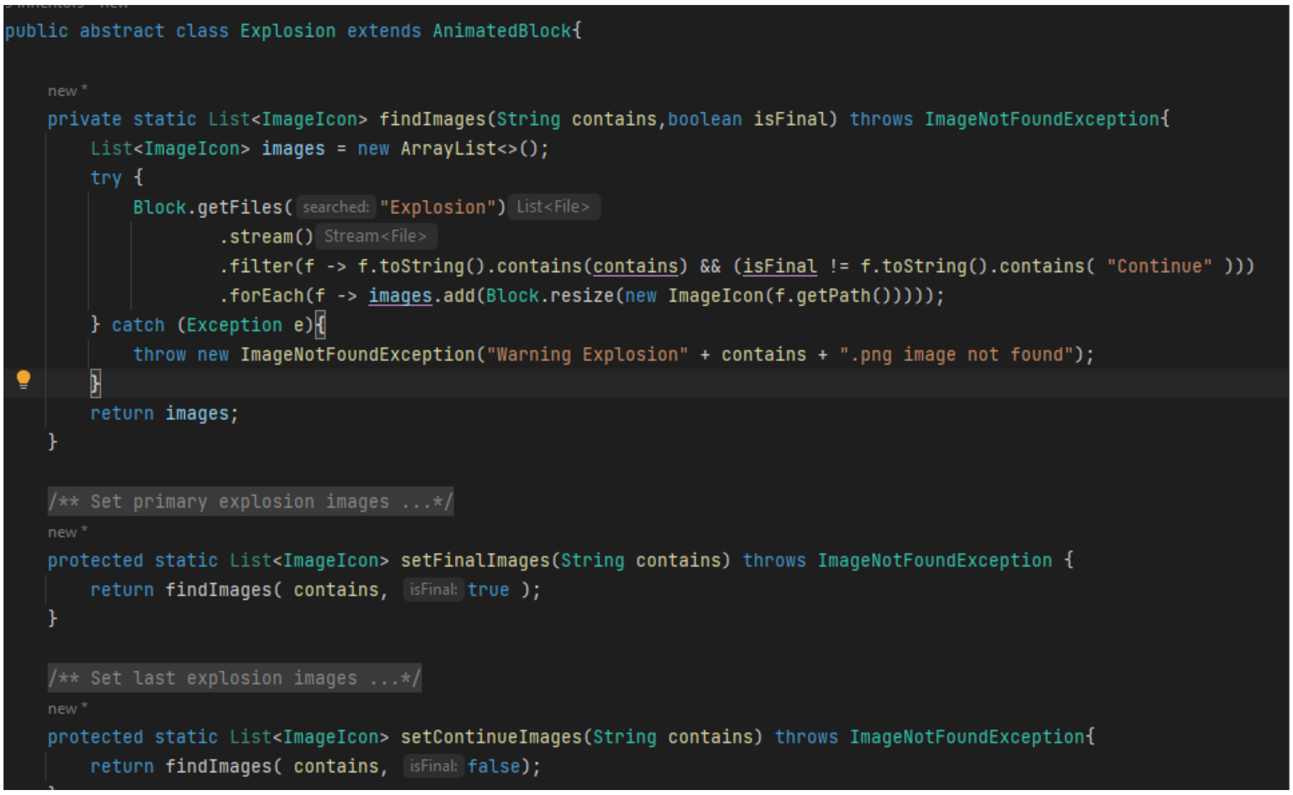


```
/**
 * Class animated block, used for block that change state
 */
14 inheritors   new *
public abstract class AnimatedBlock extends Block{
    /** switching time in millisecond */
    protected static final int SWITCHING_INTERVAL = 500;
    /** index of current image showed */
    protected int imgIndex;
    /** Process that switch image */
    protected Thread imgSwitcher;

    /**
     * Switch between different images in order to make it animated
     */
    12 implementations   new *
    protected abstract void switchImage();
}
```



```
/**
 * Class that defines a generic oriented block (it must be also animated for obvious reasons)
 */
1 inheritor   new *
public abstract class OrientedBlock extends AnimatedBlock{
    /** Current orientation of block */
    protected String currentDirection = "center";
    /** Current position of block */
    protected Coordinate currentPosition;

    /**
     * Set current position of the block
     * @param currentPosition Current coordinate
     */
    1 override   new *
    public void setCurrentPosition(Coordinate currentPosition) {
        currentDirection = this.currentPosition == null ? "center" : Coordinate.getDirection(this.currentPosition,currentPosition);
        this.currentPosition = currentPosition;
    }

    /**
     * Generic oriented block constructor
     */
    new *
    public OrientedBlock() {
    }
}
```

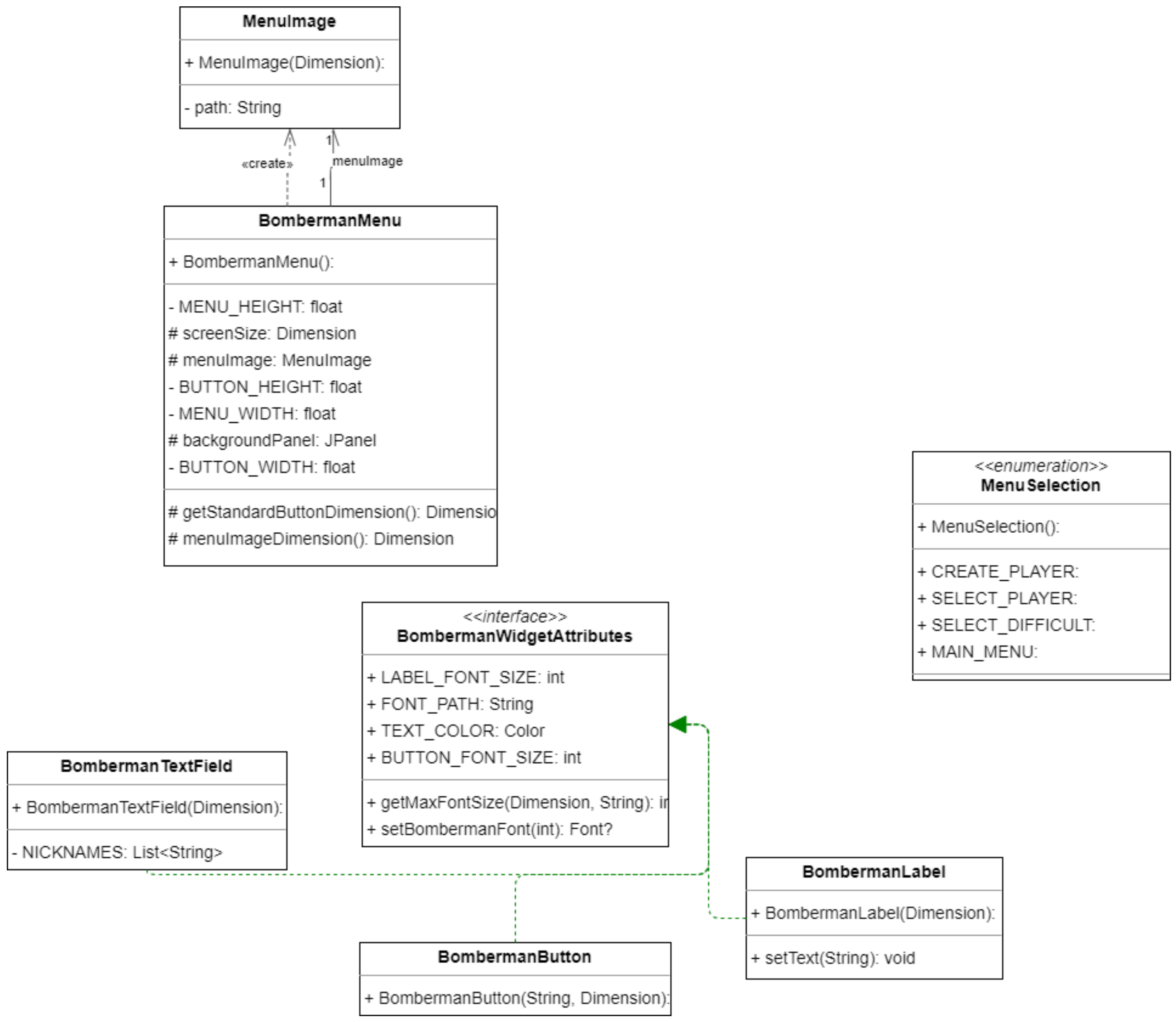
Below is the generic Explosion class that has a generic static method for file search that utilizes streams:



```
public abstract class Explosion extends AnimatedBlock{

    new *
    private static List<ImageIcon> findImages(String contains,boolean isFinal) throws ImageNotFoundException{
        List<ImageIcon> images = new ArrayList<>();
        try {
            Block.getFiles( searched: "Explosion") List<File>
                    .stream() Stream<File>
                    .filter(f -> f.toString().contains(contains) && (isFinal != f.toString().contains( "Continue" )))
                    .forEach(f -> images.add(Block.resize(new ImageIcon(f.getPath()))));
        } catch (Exception e){
            throw new ImageNotFoundException("Warning Explosion" + contains + ".png image not found");
        }
        return images;
    }

    /** Set primary explosion images ...*/
    new *
    protected static List<ImageIcon> setFinalImages(String contains) throws ImageNotFoundException {
        return findImages( contains,  isFinal: true );
    }

    /** Set last explosion images ...*/
    new *
    protected static List<ImageIcon> setContinueImages(String contains) throws ImageNotFoundException{
        return findImages( contains,  isFinal: false);
```

3.2 MENU VIEW

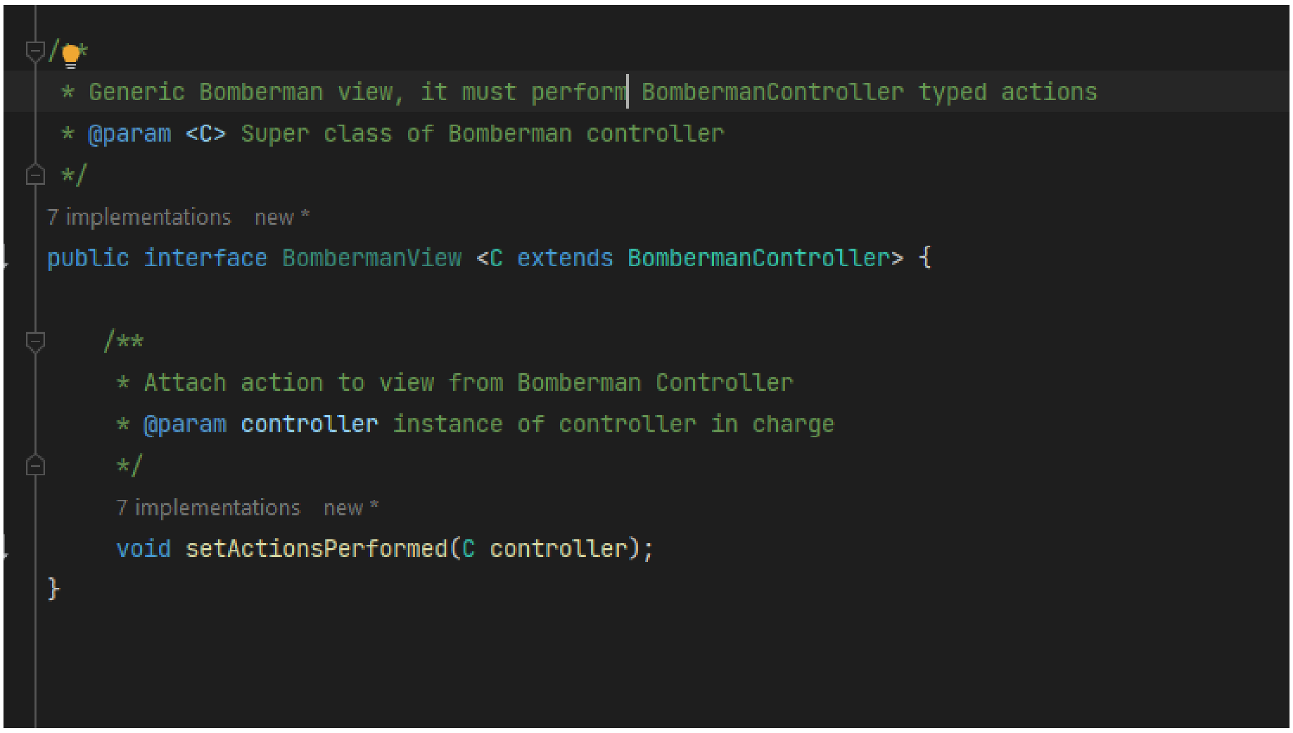
As a first step in creating the menu in the view, we chose to define some generic project components, widgets, and the game's image, of which I present the UML:



The classes defined here extend the components of the Java Swing library, customizing them for the specific use of the project. On the other hand, the BombermanWidgetAttributes interface defines the font used for creating labels, buttons, and others, creating a unified interface and a unique style for all game graphics (at least for the menu). The enumeration, instead, represents the different types of possible menus.

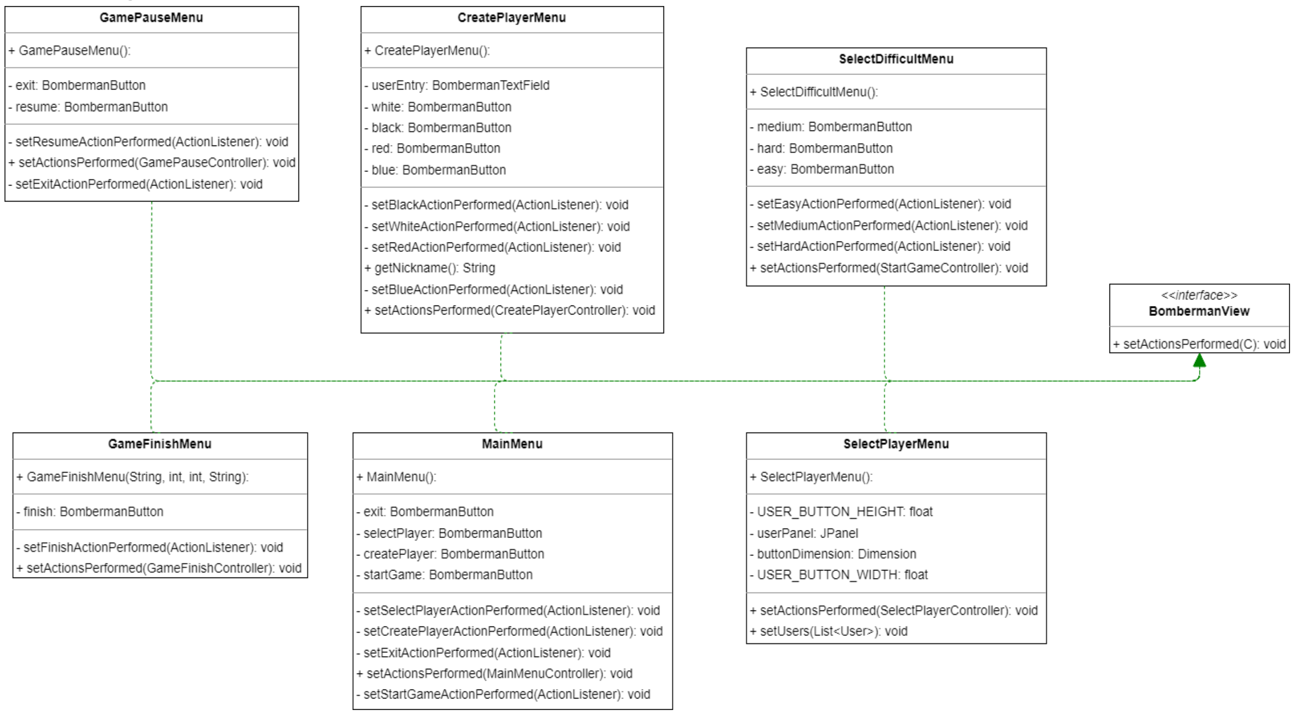
Lastly, BombermanMenu is a generic frame that only contains the bomberman image at the top, which will later be populated with the various elements that implement the previously described interface with their corresponding attributes.

Before discussing the screens that extend the BombermanMenu class, let me briefly describe the interface that all of the superclasses of BombermanMenu implement. Note that the implementation of this interface is not mandatory, but it is necessary and preferable in the case of widgets that require action listeners. Here is the code for the BombermanView interface.



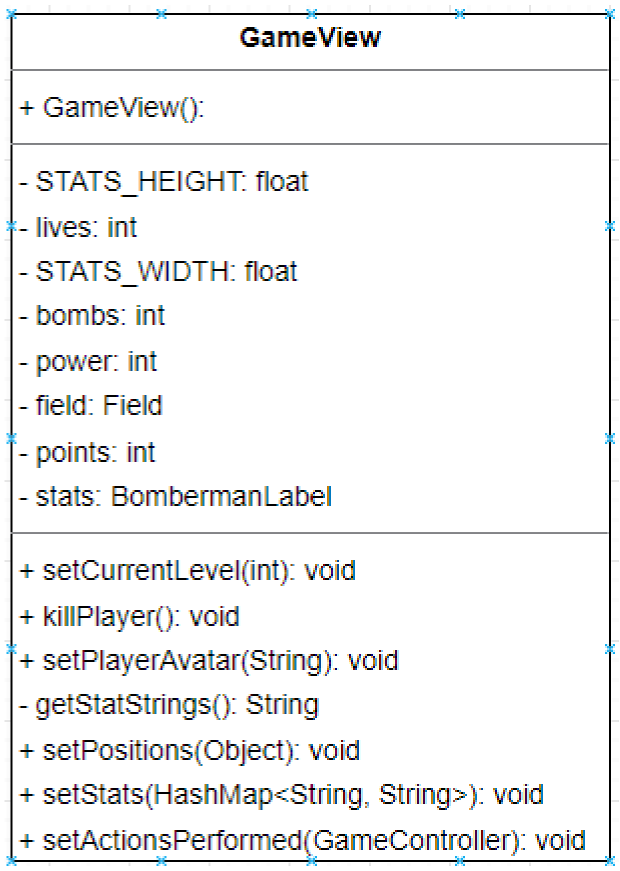
```
/**
 * Generic Bomberman view, it must perform BombermanController typed actions
 * @param <C> Super class of Bomberman controller
 */
7 implementations   new *
public interface BombermanView <C extends BombermanController> {

    /**
     * Attach action to view from Bomberman Controller
     * @param controller instance of controller in charge
     */
    7 implementations   new *
    void setActionsPerformed(C controller);
}
```

This interface exposes a method (which is actually a setter) called "setActionPerformed(C controller)", which is the implementation of the "Command Callback" design pattern. In summary, as we will see later in detail, we will have the controller that implements an inner class (the "concrete" command), which extends action listener or event listener. The latter will be set as a callback in the method of the class that implements the BombermanView interface. Besides, a generic type "C" has been defined as the supertype of BombermanController, to establish a one-to-one dependency between the controller and the view. This means that the MainMenu class will implement BombermanView, specializing C as MainMenuController. We will see this in more detail later. Also, the fact that they implement BombermanView will be useful later on. Below is the UML diagram of the menus, and the type of controller implemented is defined in the method signature:


GamePauseMenu
+ GamePauseMenu():
- exit: BombermanButton
- resume: BombermanButton
- setResumeActionPerformed(ActionListener): void
+ setActionsPerformed(GamePauseController): void
- setExitActionPerformed(ActionListener): void
CreatePlayerMenu
+ CreatePlayerMenu():
- userEntry: BombermanTextField
- white: BombermanButton
- black: BombermanButton
- red: BombermanButton
- blue: BombermanButton
- setBlackActionPerformed(ActionListener): void
- setWhiteActionPerformed(ActionListener): void
- setRedActionPerformed(ActionListener): void
+ getNickname(): String
- setBlueActionPerformed(ActionListener): void
+ setActionsPerformed(CreatePlayerController): void
SelectDifficultMenu
+ SelectDifficultMenu():
- medium: BombermanButton
- hard: BombermanButton
- easy: BombermanButton
- setEasyActionPerformed(ActionListener): void
- setMediumActionPerformed(ActionListener): void
- setHardActionPerformed(ActionListener): void
+ setActionsPerformed(StartGameController): void
<<interface>>
BombermanView
+ setActionsPerformed(C): void
GameFinishMenu
+ GameFinishMenu(String, int, int, String):
- finish: BombermanButton
- setFinishActionPerformed(ActionListener): void
+ setActionsPerformed(GameFinishController): void
MainMenu
+ MainMenu():
- exit: BombermanButton
- selectPlayer: BombermanButton
- createPlayer: BombermanButton
- startGame: BombermanButton
- setSelectPlayerActionPerformed(ActionListener): void
- setCreatePlayerActionPerformed(ActionListener): void
- setExitActionPerformed(ActionListener): void
+ setActionsPerformed(MainMenuController): void
- setStartGameActionPerformed(ActionListener): void
SelectPlayerMenu
+ SelectPlayerMenu():
- USER_BUTTON_HEIGHT: float
- userPanel: JPanel
- buttonDimension: Dimension
- USER_BUTTON_WIDTH: float
+ setActionsPerformed(SelectPlayerController): void
+ setUsers(List<User>): void

3.3 Game View (Window)

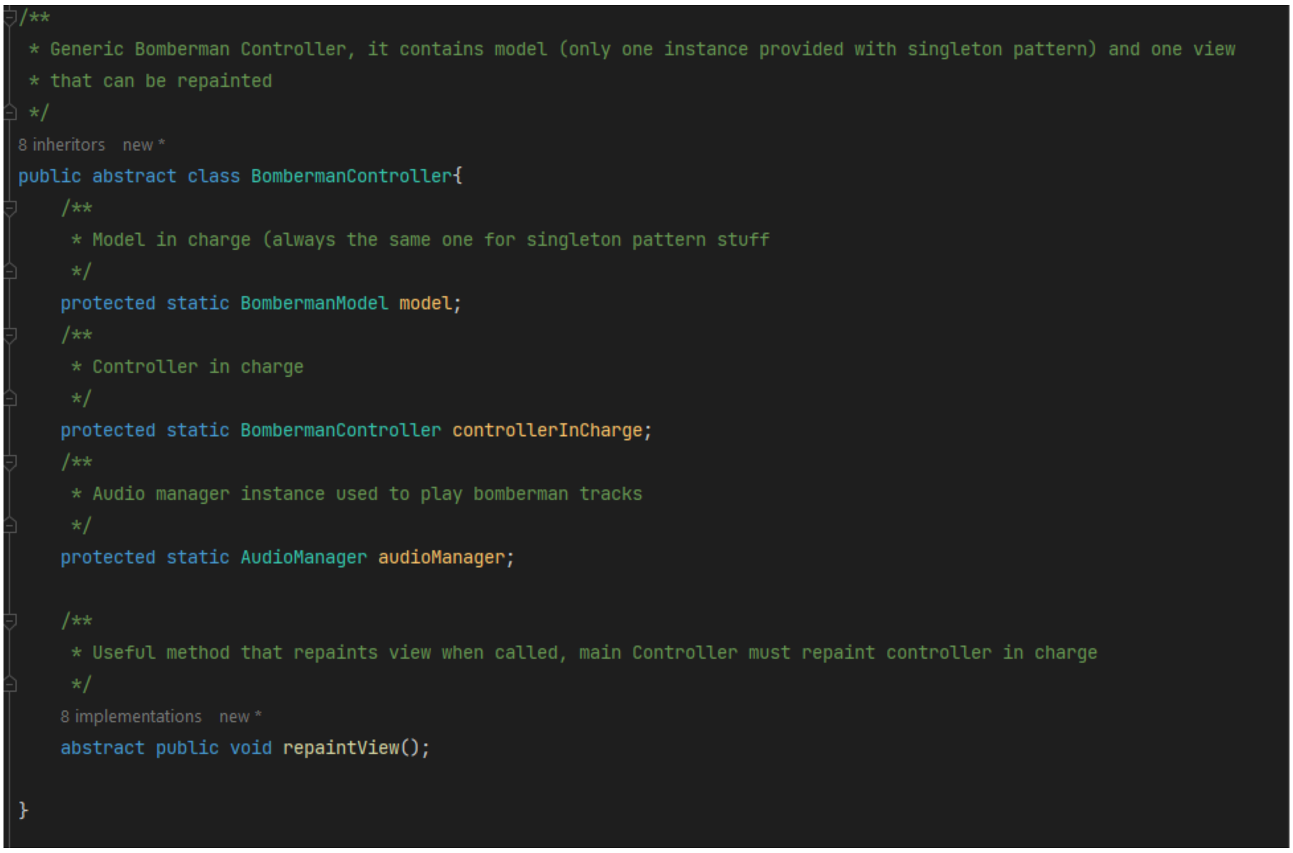
Below is the UML diagram of the game window, which maintains the menu mechanism but also stores some game data that is displayed to the user


GameView
+ GameView():
- STATS_HEIGHT: float
- lives: int
- STATS_WIDTH: float
- bombs: int
- power: int
- field: Field
- points: int
- stats: BombermanLabel
+ setCurrentLevel(int): void
+ killPlayer(): void
+ setPlayerAvatar(String): void
- getStatStrings(): String
+ setPositions(Object): void
+ setStats(HashMap<String, String>): void
+ setActionsPerformed(GameController): void

It also implements the BombermanMenu interface and uses the GameController as a specialization of the controller. Besides, it provides some methods to set the level state and data on the game field. In particular, the setPositions method has remained a generic object due to time constraints. However, it can be a list or a map and represents either the game matrix or the changes (see JavaDoc for more information).

4 CONTROLLER

Finally, let's analyze the controller that will maintain the reference of the model (unique because of the singleton), manage the game audio, instantiate the View, and add observers to the model. To fulfill these tasks, the controller has been subdivided into three packages: one for audio, one for the game, and one for the menu. Additionally, it will have a main controller class that will be called in the Main, and a BombermanController interface, which is shown in the following figure:



```java
/**
 * Generic Bomberman Controller, it contains model (only one instance provided with singleton pattern) and one view
 * that can be repainted
 */
8 inheritors  new *
public abstract class BombermanController{
    /**
     * Model in charge (always the same one for singleton pattern stuff
     */
    protected static BombermanModel model;
    /**
     * Controller in charge
     */
    protected static BombermanController controllerInCharge;
    /**
     * Audio manager instance used to play bomberman tracks
     */
    protected static AudioManager audioManager;

    /**
     * Useful method that repaints view when called, main Controller must repaint controller in charge
     */
    8 implementations  new *
    abstract public void repaintView();

}
```

This interface represents the beating heart of the controller, let's analyze it in detail:

- It maintains a single static reference to the model shared by all classes that implement the interface and will be instantiated in the main controller.

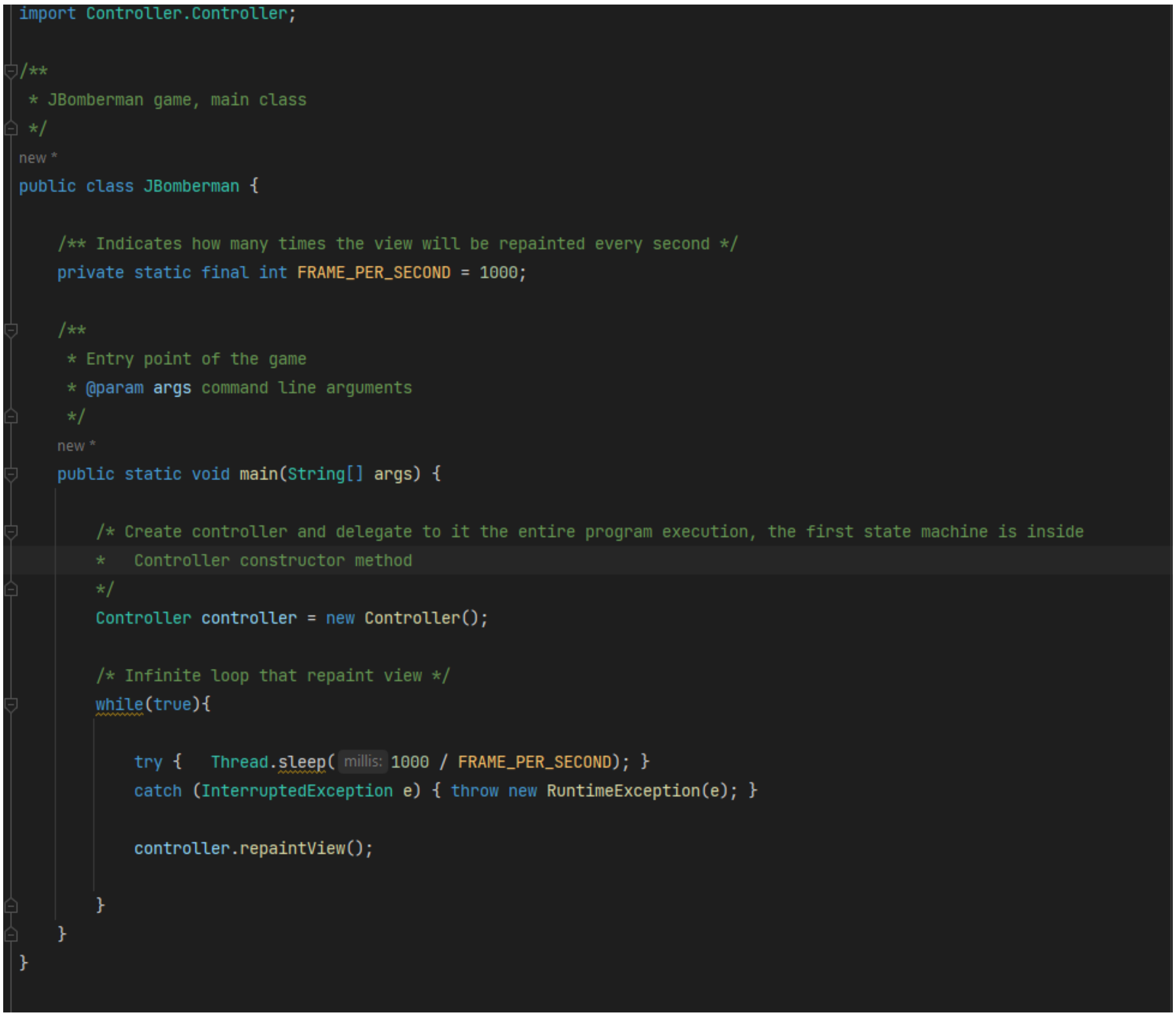
- It maintains a static reference to the current controller.

- It maintains the reference of the class that manages audio.

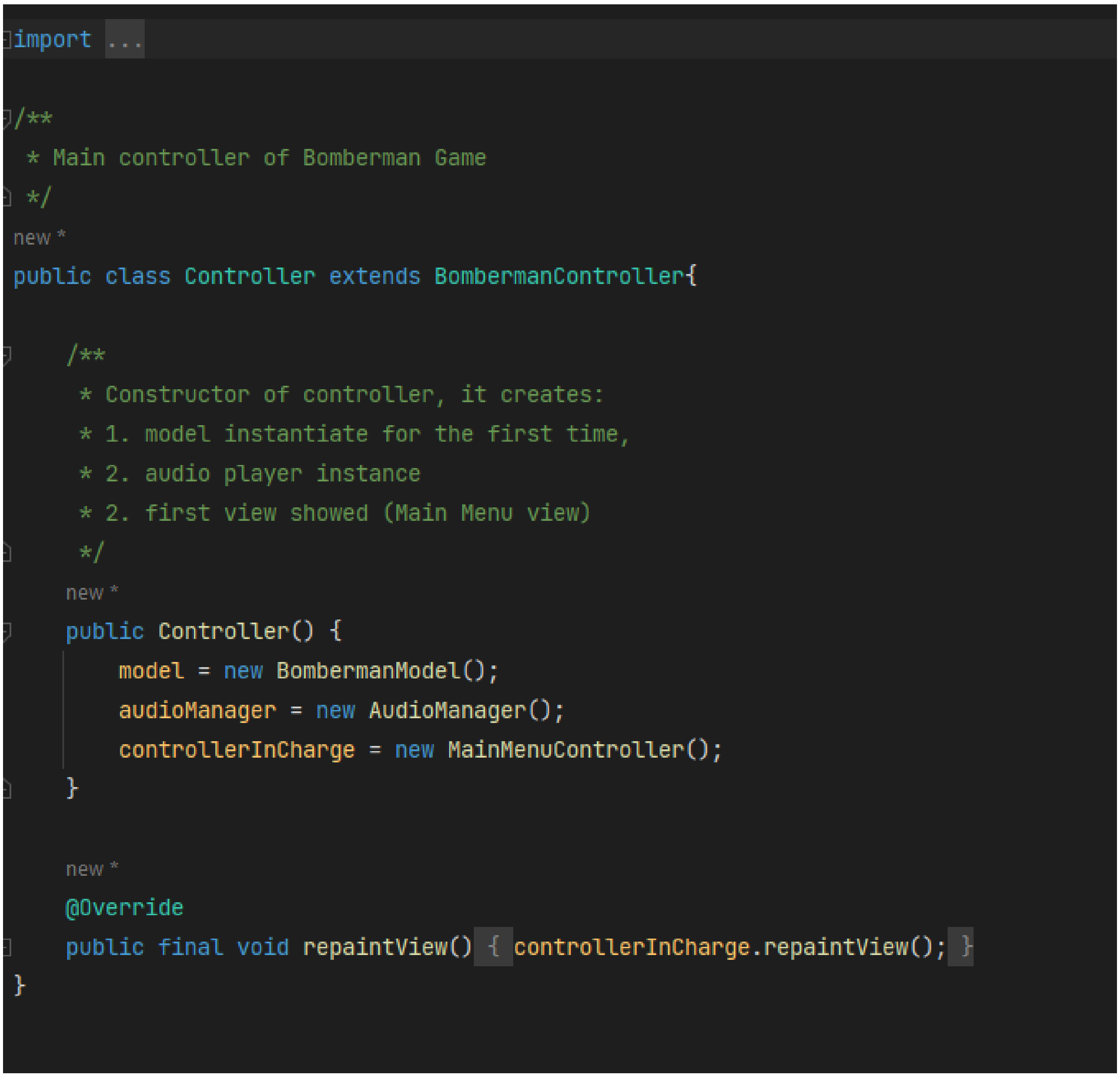
- It exposes a method to redraw the current view on the screen.

In the controller, the latest design pattern called "Chain of Responsibility" is used. Each controller is delegated to instantiate the specific view of its controller, to implement a mechanism that redraws it as provided by the interface, and to configure itself as the controller in charge of the interface. In this way, besides maintaining information hiding because no class knows what another one does, neither the main controller, we keep separate the classes that manage the view, and this makes the code more readable and maintainable.

Finally, in its implementation of the repaintView method, the main controller will redraw that of the "controllerInCharge", which, I emphasize, changes from time to time based on user input. In this method, from the main game loop, we will have a single infinite loop that will do nothing but redraw the view of the main controller (and therefore that of the controllerInCharge). With this mechanism, we are able, with a good approximation, to maintain a certain number of frames per second constant, which, during the development of many more complex video games than this project, is a project constraint. Below are screenshots of the game's main and main controller:



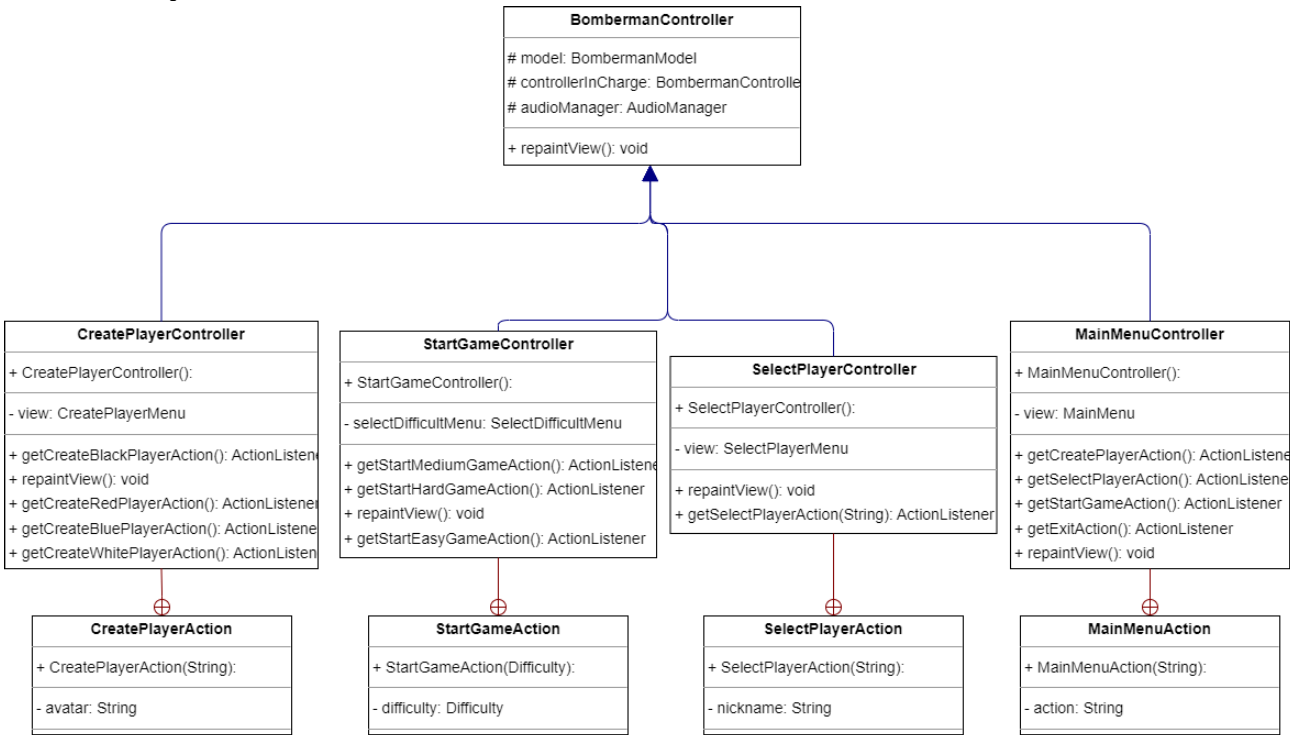
```java
import Controller.Controller;

/**
 * JBomberman game, main class
 */
new *
public class JBomberman {

    /** Indicates how many times the view will be repainted every second */
    private static final int FRAME_PER_SECOND = 1000;

    /**
     * Entry point of the game
     * @param args command line arguments
     */
    new *
    public static void main(String[] args) {

        /* Create controller and delegate to it the entire program execution, the first state machine is inside
        *   Controller constructor method
        */
        Controller controller = new Controller();

        /* Infinite loop that repaint view */
        while(true){

            try {   Thread.sleep( millis: 1000 / FRAME_PER_SECOND); }
            catch (InterruptedException e) { throw new RuntimeException(e); }

            controller.repaintView();

        }
    }
}
```



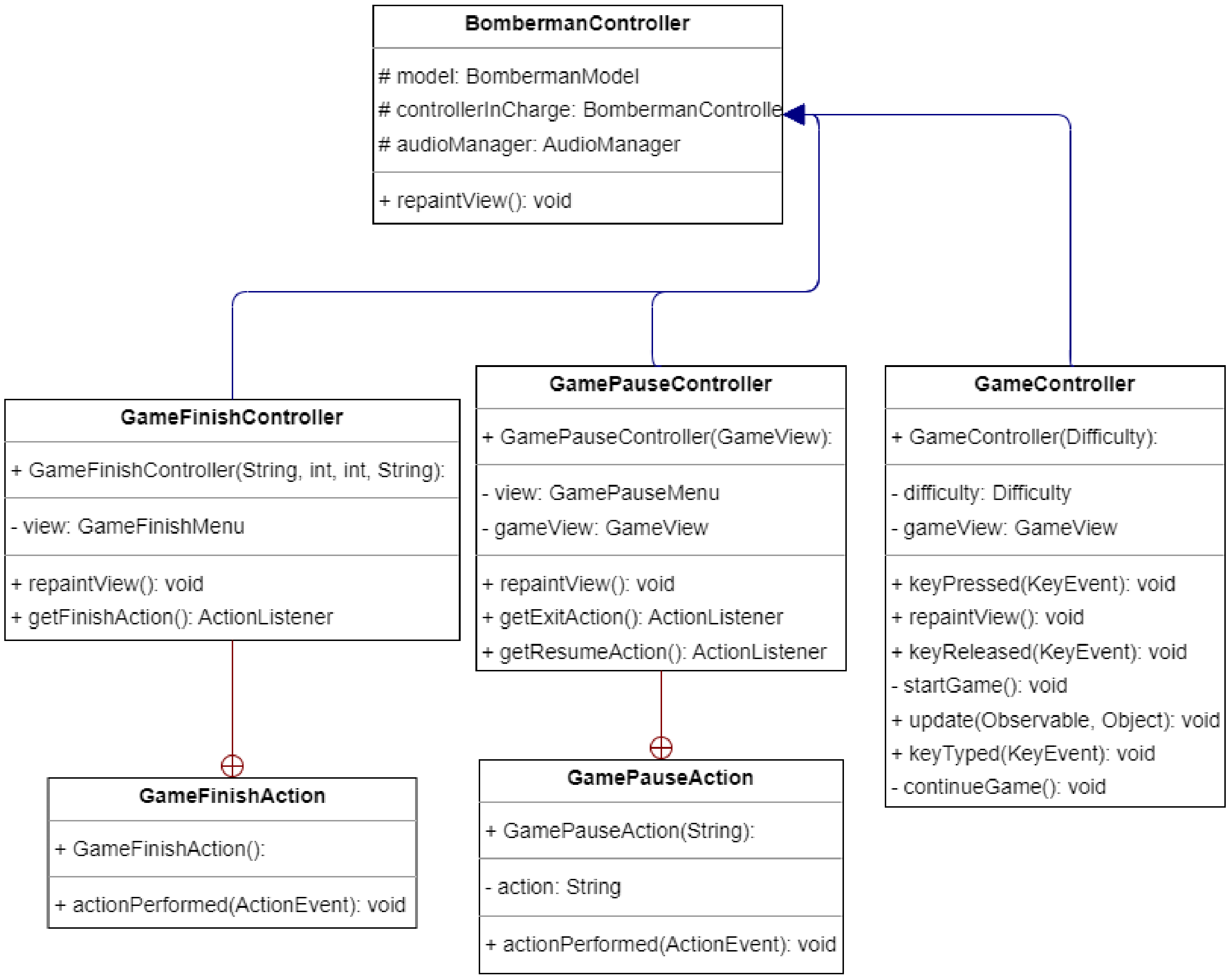
```
import ...

/**
 * Main controller of Bomberman Game
 */
new *
public class Controller extends BombermanController{

    /**
     * Constructor of controller, it creates:
     * 1. model instantiate for the first time,
     * 2. audio player instance
     * 2. first view showed (Main Menu view)
     */
    new *
    public Controller() {
        model = new BombermanModel();
        audioManager = new AudioManager();
        controllerInCharge = new MainMenuController();
    }

    new *
    @Override
    public final void repaintView() { controllerInCharge.repaintView(); }
}
```

Now I present the UML diagrams of the menu and game packages of the controller. I have omitted the dependencies as they will be seen in the last paragraph in the game sequence diagram.

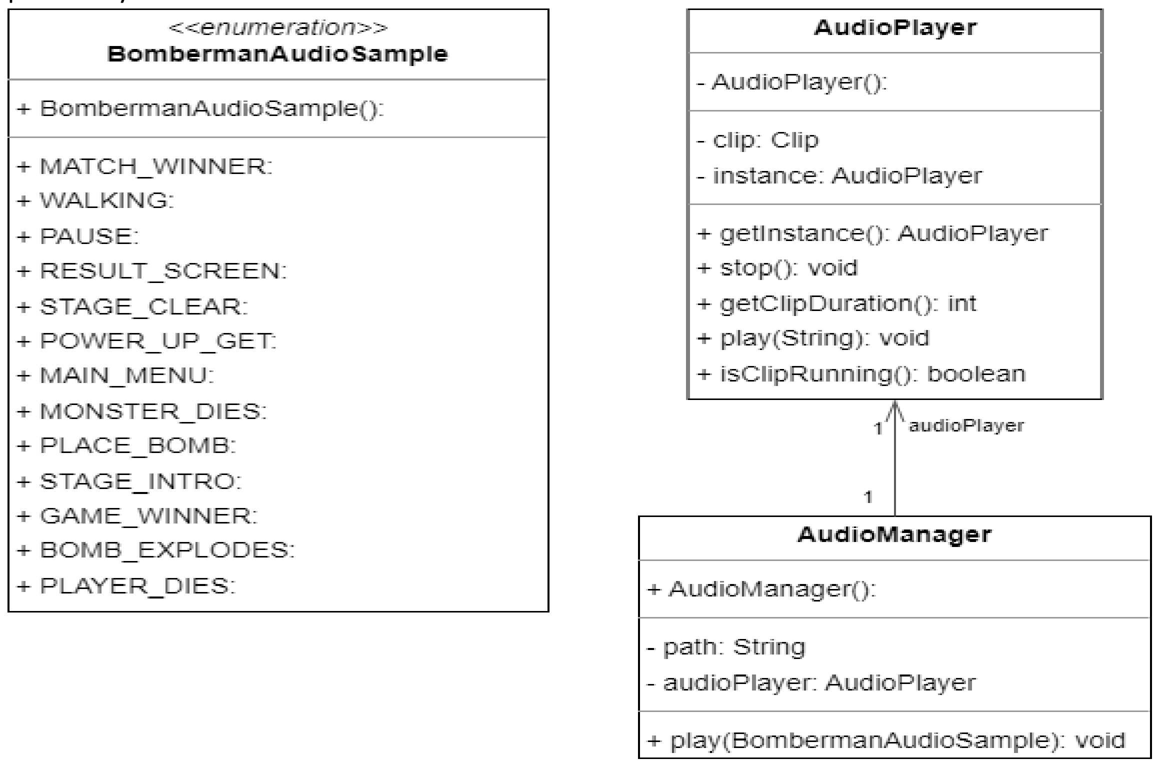
This is the diagram related to the menu controllers:


BombermanController
# model: BombermanModel
# controllerInCharge: BombermanControlle
# audioManager: AudioManager
+ repaintView(): void
CreatePlayerController
+ CreatePlayerController():
- view: CreatePlayerMenu
+ getCreateBlackPlayerAction(): ActionListen
+ repaintView(): void
+ getCreateRedPlayerAction(): ActionListener
+ getCreateBluePlayerAction(): ActionListene
+ getCreateWhitePlayerAction(): ActionListen
StartGameController
+ StartGameController():
- selectDifficultMenu: SelectDifficultMenu
+ getStartMediumGameAction(): ActionListene
+ getStartHardGameAction(): ActionListener
+ repaintView(): void
+ getStartEasyGameAction(): ActionListener
SelectPlayerController
+ SelectPlayerController():
- view: SelectPlayerMenu
+ repaintView(): void
+ getSelectPlayerAction(String): ActionListener
MainMenuController
+ MainMenuController():
- view: MainMenu
+ getCreatePlayerAction(): ActionListene
+ getSelectPlayerAction(): ActionListene
+ getStartGameAction(): ActionListener
+ getExitAction(): ActionListener
+ repaintView(): void
CreatePlayerAction
+ CreatePlayerAction(String):
- avatar: String
StartGameAction
+ StartGameAction(Difficulty):
- difficulty: Difficulty
SelectPlayerAction
+ SelectPlayerAction(String):
- nickname: String
MainMenuAction
+ MainMenuAction(String):
- action: String

This is the diagram related to the game controllers:

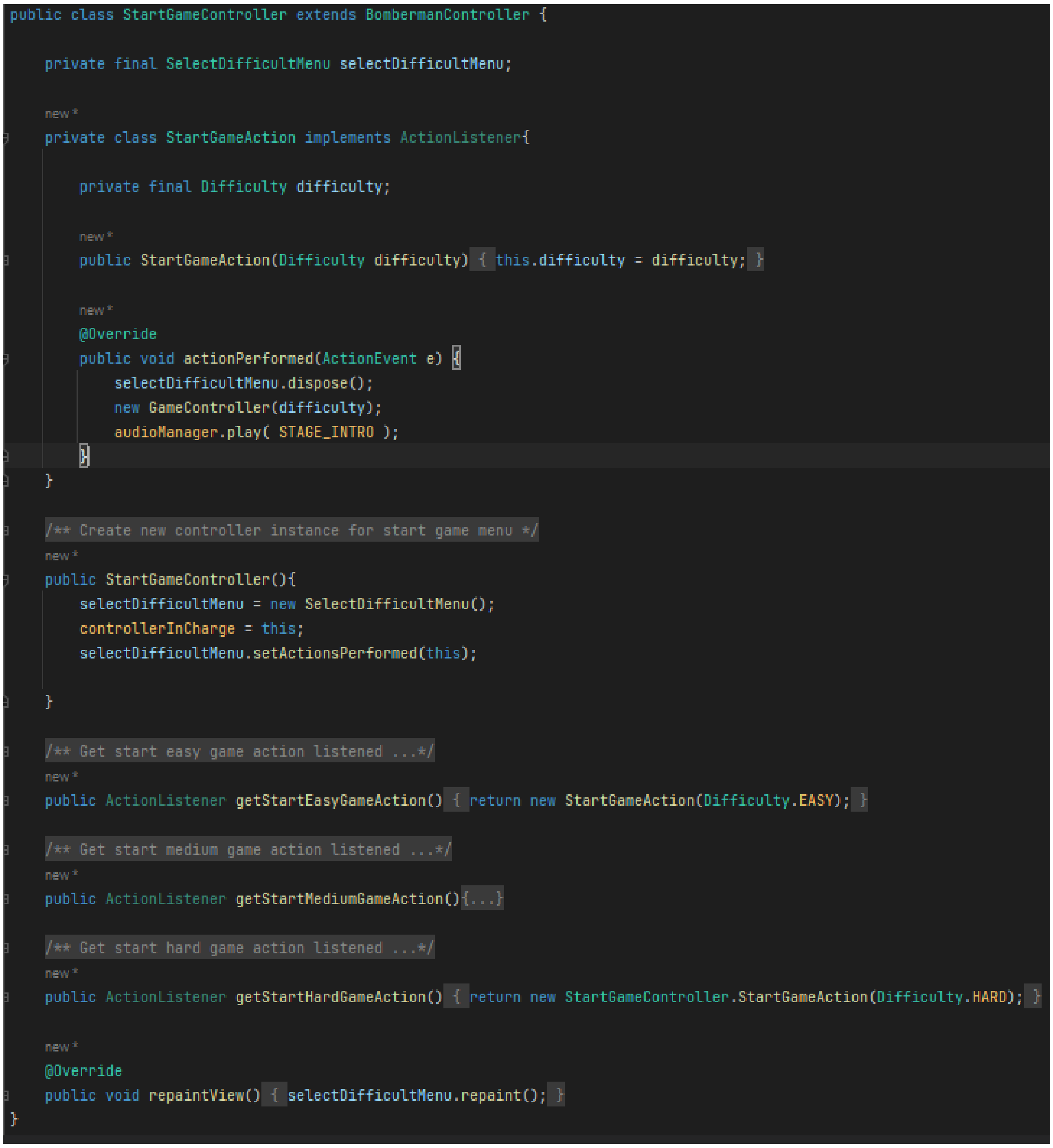

BombermanController
# model: BombermanModel
# controllerInCharge: BombermanControlle
# audioManager: AudioManager
+ repaintView(): void
GameFinishController
+ GameFinishController(String, int, int, String):
- view: GameFinishMenu
+ repaintView(): void
+ getFinishAction(): ActionListener
GamePauseController
+ GamePauseController(GameView):
- view: GamePauseMenu
- gameView: GameView
+ repaintView(): void
+ getExitAction(): ActionListener
+ getResumeAction(): ActionListener
GameController
+ GameController(Difficulty):
- difficulty: Difficulty
- gameView: GameView
+ keyPressed(KeyEvent): void
+ repaintView(): void
+ keyReleased(KeyEvent): void
- startGame(): void
+ update(Observable, Object): void
+ keyTyped(KeyEvent): void
- continueGame(): void
GameFinishAction
+ GameFinishAction():
+ actionPerformed(ActionEvent): void
GamePauseAction
+ GamePauseAction(String):
- action: String
+ actionPerformed(ActionEvent): void

Finally, the UML diagram of the audio controller, which implements the Singleton pattern that has been previously discussed:



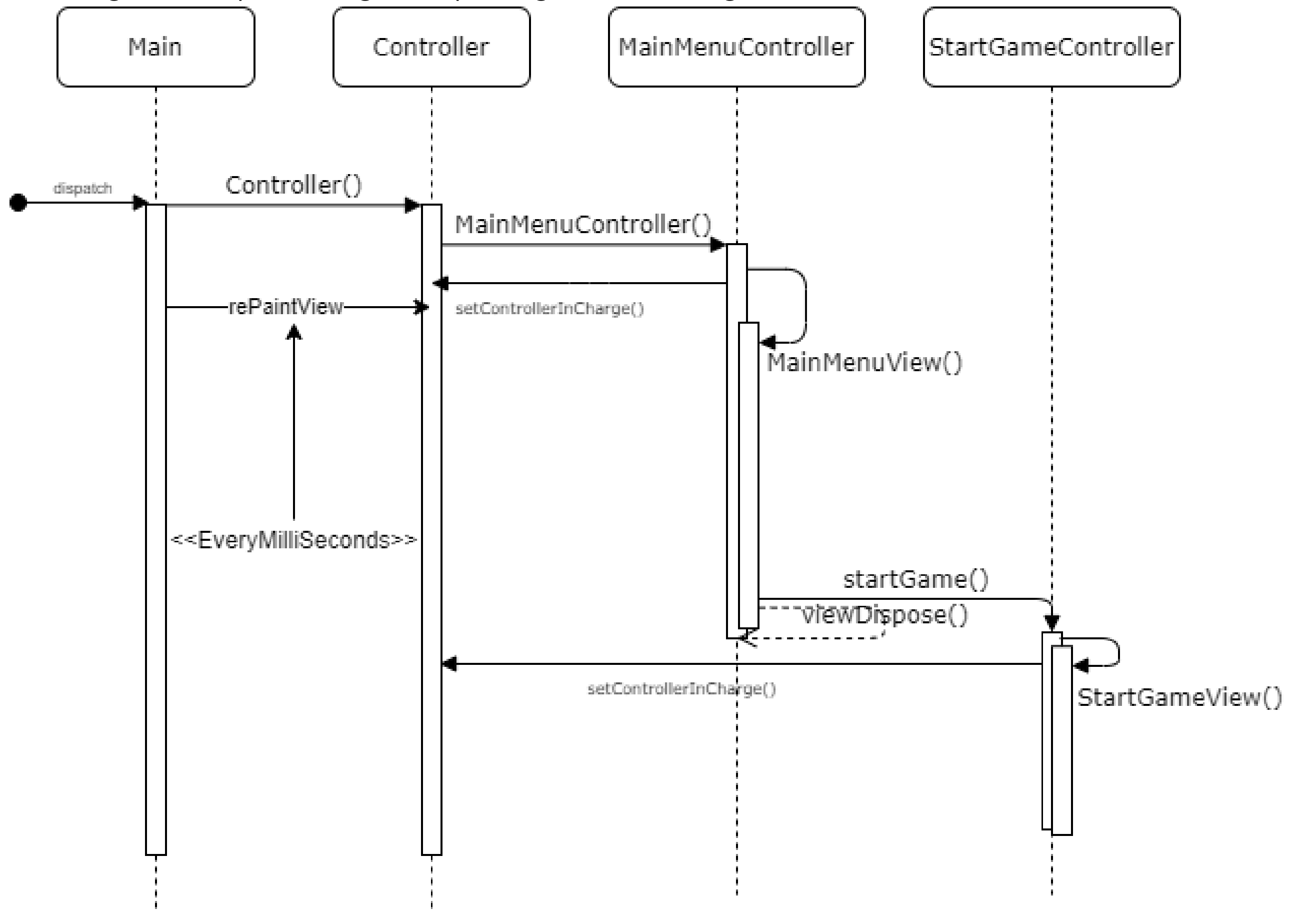
5 BOMBERMAN GAME SEQUENCE

In this final section, the gameplay flow is described, specifically how menus are managed and how one transitions from one to the other. The internal methods that allow for moving between windows are also described as previously explained in the controller. Each controller has the responsibility to establish itself as the "controller in charge", to instantiate its own view and destroy it when required, as well as to proclaim a new "controller in charge". Below is an example code of a controller:

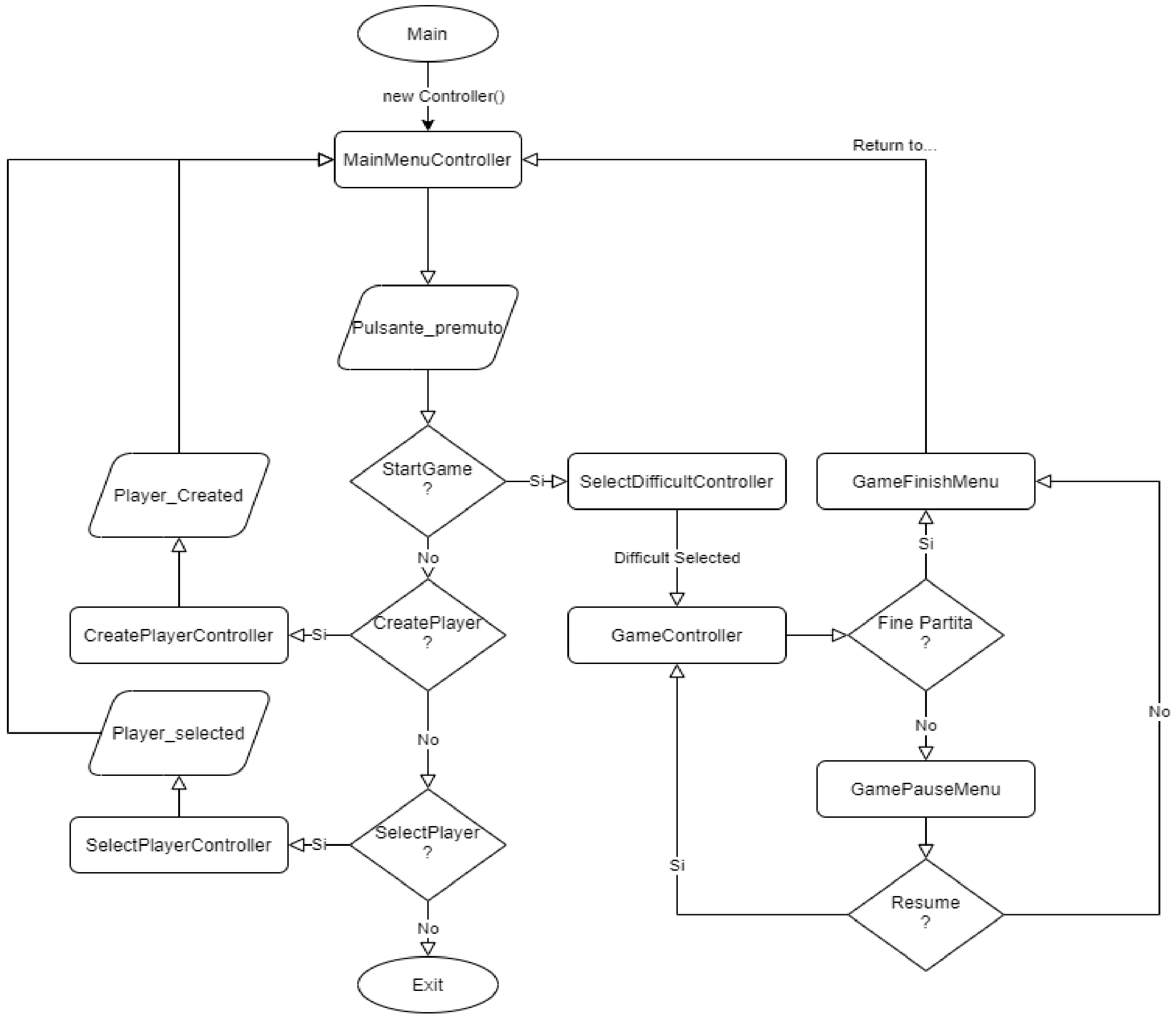


```java
public class StartGameController extends BombermanController {

    private final SelectDifficultMenu selectDifficultMenu;

    new *
    private class StartGameAction implements ActionListener{

        private final Difficulty difficulty;

        new *
        public StartGameAction(Difficulty difficulty) { this.difficulty = difficulty; }

        new *
        @Override
        public void actionPerformed(ActionEvent e) {
            selectDifficultMenu.dispose();
            new GameController(difficulty);
            audioManager.play( STAGE_INTRO );
        }
    }

    /** Create new controller instance for start game menu */
    new *
    public StartGameController(){
        selectDifficultMenu = new SelectDifficultMenu();
        controllerInCharge = this;
        selectDifficultMenu.setActionsPerformed(this);

    }

    /** Get start easy game action listened ...*/
    new *
    public ActionListener getStartEasyGameAction() { return new StartGameAction(Difficulty.EASY); }

    /** Get start medium game action listened ...*/
    new *
    public ActionListener getStartMediumGameAction(){...}

    /** Get start hard game action listened ...*/
    new *
    public ActionListener getStartHardGameAction() { return new StartGameController.StartGameAction(Difficulty.HARD); }

    new *
    @Override
    public void repaintView() { selectDifficultMenu.repaint(); }
}
```

Here is a generic sequence diagram explaining the functioning and behavior of the controller.



Finally, here is the flowchart explaining how the various menus are created and managed:



6 NOTES

Firstly, it should be noted that due to design choices, the MVC pattern has a slight modification compared to the image that represents the generic diagram present on one of the first pages. In this case, the model does not directly notify the view, but instead passes through the controller. This is through the use of the Observer/Observable pattern (and the Java library provided) during the execution of the game. The model will only notify the state of the game, any minor data (number of lives, number of bombs, etc.), and the game graphics (complete or only changes compared to the previous state). The controller will simply pass everything to the view, with the exception of the game status, which, in the case of a won game, will pass to the GameFinish screen (hence the reason for the intervention of the controller between the view and model).

For better readability of the graphics in this document, some .png files have been left in the diagrams folder within the project files. Lastly, unfortunately, due to time constraints, there are a few things that could be optimized and improved. Some of the choices may be questionable and could be changed, but like any respectable software, this presents an opportunity for a version 2.0, perhaps with the addition of multiplayer capabilities.