
Daniel Alcaide Nombela - 100383530

Alejandro Martínez Riera - 100370039

Group 88 – Computer science degree, UC3M

Table of contents

# 1. Abstract

In this document we will discuss how the final project has been resolved, explaining the most remarkable aspects of the class design and algorithms used. First, we will see how the data structure of the game has been resolved, starting by the main body of the game, followed by the environment (map and levels) and finally how the entities and NPC's are handled. Then, we will look at some of the most important algorithms used, mainly the ones concerning runtime, entity management and animation. Finally, we will discuss the work we have done. Also, a list with all the extra functionality provided is included.

For more information visit the Javadoc at
https://github.com/zakneifian/bomberman_uc3m/tree/master/javadoc.

# 2. Class design

The program is divided into the following packages:

### `uc3m.bomberman.main:`

- **`Main:`** Contains the `main` method and its decomposition. It also provides the function `static int nextId()`, which provides the id a new entity should have. The return of this function is increased by 1 every time it is called, so that no entities have the same id if they are created using this method. Our intention was to keep here all executions and runtime invocations to the API provided.
  (see Algorithms -> Main method workflow).
- **`Game:`** Stores and handles all the information needed to execute the game, being helped by other classes.
  A game has a `Player`, a `Map` array representing the different levels and an `Entity` array, representing the entities currently alive and active.
  Its most important methods are:
  - `public Game(int, String, int)` (constructor): This class's constructor creates the player of the game and all the different maps (levels). It then adds the player to the entity array as the first object. Finally, it spawns all the enemies in the positions the currently selected level (level 1) generated them (`see` Map).
  - `public int nextMap()`: This method first removes all entities from the `Entity` array (except the player), calculates and adds the corresponding time bonus to the player, and, if possible (i.e.: if there is next level) the level is advanced. If there is next level, it returns the bonus; else, it returns the bonus negative. This return makes it able possible to know whether there is next level or not, while providing also the time bonus received.
  - `public boolean addEntity(Entity)`: adds the parameter to the `Entity` array (see Algorithms -> Entity array management)
  - `public boolean removeEntity(int)`: removes the entity with the given id from the array (see Algorithms -> Entity array management)
  - `public void explodeAt(Coordinates)`: Creates explosions appropriately, with the centre of the explosion at the coordinates passed as parameter (these coordinates are meant to be the position of the bomb, so they should be in tenths). It adds explosions of the appropriate type to the current map, given the range of the player, extending through green and brick tiles towards the four cardinal points, and stopping when there is a wall (see `Explosion`, `Map`).

## uc3m.bomberman.map:

- **Coordinates:** Stores two integer values, `x` and `y`, corresponding to the two components of some coordinates. It also is provided with two methods for conversion from the two reference systems of the API provided: square coordinates and tenth coordinates:
  - `public Coordinates tenthsToUnits():` returns the coordinates stored as tenths converted to units.
  - `public Coordinates unitsToTenths():` returns the coordinates stored as units converted to tenths.
- **Map:** Stores the map as a `Tile` matrix and provides the methods regarding the creation of the layout and the enemies, as well as the manipulation of the environment.
  - `public void createMap(Coordinates[], int):` creates the tiles, upgrades and enemies layout. It first sets all cells green, then creates the wall layout and then fills up green tiles with bricks in a random fashion, up to 50 bricks. The `Coordinate` array are the coordinates of the squares that should not be filled up with bricks. Finally, the bricks are filled up with the upgrades depending on the level (the `int` parameter) and the green tiles with enemies. The places where these two must be created when needed is stored in a `String` matrix, where the types of upgrades/enemies are named in the coordinates of the matrix where they should be in the map. It is decomposed into `genEnemies(int)` and `genUpgrades(int)`
- **Tile:** Stores the tile type in a `char` type to save memory, and provides methods for tile management such as:
  - `public void setType(String):` it lower cases the parameter and depending on what string it is, it sets a certain type to the Tile object and whether it is walkable.
  - `public String getType():` It returns the type of the tile ins a String type.
  - `public String getSprite():` It returns the sprite path string given the object type.
- **Explosion:** Extends `Tile`, stores the type of explosion tiles (North, South, etc.) with a character type and handles them.
  - `private void setExplosionType(String):` Sets the string to lower case and depending on the string, it sets a type.
  - `public boolean tick():` returns true if the explosion should disappear, and animates the tile appropriately each time it is called.
- **Upgrade:** Extends `Tile`, stores the type of upgrade tiles and handles them.
  - `private void setUpgradeType(String):` depending on the string, it sets the appropriate type.
  - `public String getUpgradeType():` Returns upgrade type as a String.
  - `public String getSprite():` Returns the sprite path String of the upgrade.

## uc3m.bomberman.entity:

- **Direction:** Enum representing the four directions an Entity can go towards to (***UP***, ***DOWN***, ***LEFT***, ***RIGHT***).
- **Entity:** Stores and handles all the information and methods needed to instance an Entity. An entity has an id, its sprite, its position, health, maximum health and whether or not it's alive.
  - `public boolean collides(Entity):` returns true if Entity Coordinates and the Object Coordinates are the same, otherwise returns false.
  - `public boolean collides(Map):` returns true if at the Coordinates of the Object, the Map has a Tile that is not walkable, otherwise false.

    - `public void takeDamage(int dmg)`: If the int is positive, lowers the health of the Object, and if the health is below 0, it sets the alive property to false.
- **`MovableEntity`:** Extends `Entity`, handles all the information and methods needed for entities that might move.
    - `public abstract void onCollision(Entity);`
    - `public void animateMovement(int, String[])`: Depending on the String[] that has two values and the int, which changes which sprite to show from a certain type of movement, the method sets a new sprite to render.
    - `public void moveTowards(Direction, Map)`: Depending on one of the four values of Direction, it moves the entity in the map towards said direction.
- **`Enemy`:** Extends `MovableEntity`, handles all the information and methods needed for enemy entities.
    - `public void onCollision(Entity)`: if Entity is instance of Player and Object collisions Entity, Entity lowers its health
    - `public abstract void moveEnemy(Game)`: Handles the movement, implemented on each enemy.
    - `public void animateMovement(int, String[])`: Depending on the String[] that has two values and the int, which changes which sprite to show from a certain type of movement, the method sets a new sprite to render.
- **`Balloon`:** Extends `Enemy`, handles all the information and methods needed for the Balloon enemy.
    - `public void moveEnemy(Game)`: Randomly moves the Balloon at determined spans of time.
- **`Slime`:** Extends `Enemy`, handles all the information and methods needed for the Slime enemy.
    - `public void moveEnemy(Game)`: Tries to move towards the player comparing their positions.
- **`AntiBomberman`:** Extends `Enemy`, handles all the information and methods needed for the AntiBomberman enemy.
    - `public void animateMovement(int, String[])`: Depending on the String[] that has two values and the int, which changes which sprite to show from a certain type of movement, the method sets a new sprite to render. In this specific case, it imitates the player.
    - `public void moveEnemy(Game game)`: It imitates the movement actions from the player.
- **`Bomb`:** Extends `Entity`, handles all the information and methods needed for the Bomb management such as the animation and whether or not it should've exploded when finishing the tick.
    - `public boolean tick()`: returns true when a time variable reaches 0, meanwhile it animates the sprites of the bomb, otherwise returns false.
- **`Player`:** Extends `MovableEntity`, handles all the information and methods needed for the player, for example its stats, interaction with upgrades, enemies, etc.
    - `public void upgrade(String)`: The string is the type of the upgrade that the player should be touching when on collision with it. It enables the effects of said upgrade.
    - `public void addScore(int)`: Raises the player's score by int.
    - `public boolean putBomb()`: if still has bombs, lower by one and return true, false otherwise.
    - `public void bombExploded()`: Method that should be called when the bomb has exploded. Raises the bomb variable by one, unless it has reached it maximum capacity.

# 3. Algorithms

In order to make the game update itself in a periodic way, we have divided those updates into two sets: *tick* updates, which are the calculations that need to be made in the environment of the game; and *render* updates, which print the result of the previous calculations on the screen, via the API provided. This will make it possible to assign different refresh rates to both of them separately, and so the graphics refreshing rate can be higher than the actual speed of the game.

To perform these updates in a periodic way, we can follow these steps:

```
1.  (System time) -> tickTime
2.  (System time) -> renderTime
3.  If the game should be running, go to 4; else end.
4.  (System time) – tickTime -> deltaTimeTick
5.  (System time) – renderTime -> deltaTimeRender
6.  If deltaTimeTick > 1/(ticks per second) go to 7; else, go to 10.
7.  Perform tick updates:
    public static void tickHandler(Game, GameBoardGUI), etc.
8.  0 -> deltaTimeTick
9.  (System time) -> tickTime
10. If deltaTimeRender > 1/(frames per second) go to 11; else, go to 14.
11. Perform render updates:
    public static void render(Game, GameBoardGUI), etc.
12. 0 -> deltaTimeRender
13. (System time) -> renderTime
14. Go to 3
```

This loop will be running as long as the game should be, and will be stopped when the player is dead, or the game is over for any other reason. However, as the program itself should not end when the game ends (because the user might want to play another game), the loop described above is inside another loop, that checks what the actions of the user are while the game is not running. If a new game is created, then it will let the loop inside run. If the exit order is sent, the loop will stop and the program will terminate.

## Ticks

The ticks are performed invoking the following methods in the ordering they are explained:

**`tickHandler`**

When performing a *tick* update, we ought to pass that tick to every "tickable" object (i.e.: every object that should act without user interaction and only because of the pass of time). To make this, we first need to scan through the `Entity` array of the `Game`. If we find a `Bomb`, a tick is sent to it; the tick will animate the bomb and tell it has exploded or not depending on the time passed since its placement/creation. If it has, an explosion is created on the current map with the bomb's position as the centre (see `Game.explodeAt(Coordinates)`). Elsewise, if an `Enemy` is found, we make it move invoking the proper method. All "tickable" entities act based on an internal timer, so they do not depend on the ticks per second rate of the game.

This is the way that we update the entities, but some tiles, as the explosions, need also to be updated. This is done in the method `Map.tickTiles()`. If when scanning through the `Tile` matrix of the map we find an `Explosion`, we call the method to update it; this method is similar to the method used to update the bomb. So, if the explosion has expired, we substitute it by the proper tile, considering the `String` matrix that stores the places of the upgrades: if there is an upgrade in the place of the explosion, we substitute the explosion by an upgrade of that type; else, by a green tile.

**eventHandler**

When the provided API tells some action has been performed, the event handler will choose what to do according with the action. If the player is alive and the action is something involving game controls, it will perform the actions accordingly (for player movement, see below). It will also parse commands through the proper method. If a new game has been ordered creating, this method will return the name of the new game; it will return `"exit"` if the player hit the exit button; else, it will return an empty String. This way, we can check inside the runtime loop whether the game should be recreated or the program terminated.

**collisionHandler**

The *tick* update also handles collisions between entities and between entities and the active tiles of the map (explosions). When two entities are in the same position, we say that a *collision* has occurred. When this happens, and if the entity is movable, the method `MovableEntity.onCollision(Entity)` is invoked. To check these collisions, for each element in the game's entity array we compare if it has collided with any other entity; if it has, then we invoke the `onCollision` method with that collided entity as parameter.

## Renders

**renderMap**

Every *render* update, the sprite according to every tile is set in the screen through the method Map.getSpriteAt(Coordinates). The methods `gb_setSquare*` are used in this process.

**renderStats**

The stats of the player (the side menu) are also updated every frame.

**renderEntities**

When a new map is created or selected for showing, the method `loadEntities` must be invoked, so that all the entities of the game are added to the game's array AND their sprites are correctly added to the API's board. Once those entities and their sprites have been added, every entity in the array has a sprite with its same id in the board. This way, when passing through all the entities in the game's array, the program will set the image to the sprite the entity gives through the method `Entity.getSprite()` and the position to the position stored in the `Entity` class; and this method will do this with the entities' id's as the sprites' id's. The API methods `gb_setSprite*` are used in this process.

## Game

**Entity[] entities**

All the entities currently being displayed and handled are stored in this array. The player is its first element, and will only change when a new game is created, as it is created when the game is. Several rules apply to this array: there must not be null pointers and there cannot be two elements with the same id. This will avoid both duplicates and null pointer exceptions. In order to handle this array while keeping this rules consistent, several methods are provided:

- `public boolean addEntity(Entity)`: In this method, a new entity will be added to the array if it is alive and there is not an entity with the same id in the array already. If both conditions are satisfied, it will create a new auxiliary array with one more element, add this entity to the end of the aux array, copy all the elements of the old array in it, assign the pointer of the new aux array to the old array, and return true; else, it will return false.
- `public boolean removeEntity(int)`: This method will scan through the entity array until it finds an entity with an id equal to the value passed as parameter. When it does, it will first kill the entity via `Entity.kill()`, and then it will copy all the elements up to

the entity to be removed from the game's entity array to an auxiliary array with one less entry than the original, and then all the entities from the game's array starting from the element next to the entity to be removed. Then the original array will be assigned the pointer of the auxiliary, and return true. If the entity with the proper id has not been found, false will be returned. Thus, the entity will be removed.

- `public boolean clearEntities()`: Creates a new array with the game's player as the only element and substitutes the original array with the new one.
- `public void spawnEnemies()`: This method will add entities to the game's array via the previous method following the enemies map stored in the String matrix of the current level's map.

**`Map[] map`**

The different levels are created in the constructor of Game, and stored in a Map array with length the number of levels of the game. The method `pubic int nextMap()` is used to advance to the next level. This method first calculates the time bonus of advancing to the next level; then, it increments the selected level by 1. If it is greater than the number of levels generated, it will make the bonus calculated negative, thus marking that there is no next level (if there is no bonus it will return -1); else, it will return the positive bonus normally.

## Map

**`createMap`**

The main algorithm used to generate all things that must appear in random places is:

1. If the object has a probability of appearing, set the number of objects to be generated according to that probability; else, set it another way (# of objects)
2. Set a probability of an object appearing in a said place fixed to (# of objects)/(places possible)
3. 0 -> i
4. 0 -> j
5. Go to row i in the Tile matrix
6. Go to column j in the Tile
7. If the number is less than the probability of step 2, the number of generated objects is less than it should be, and the object can be generated in the coordinates (i, j), generate the object in the coordinates (i, j) and increment by 1 the object counter
8. j++
9. If the row has not ended, go to 6; else continue
10. i++
11. If there is more rows, go to 5; else continue
12. If the number of objects generated is less than it should be, go to step 5; else terminate.

This way, the objects, should them be tiles, upgrades or enemies, are generated randomly, and up to a fixed number and with the proper probability of appearing, in case they have.

## Entity

**`moveTowards`**

In the eventHandler, when the action is "up", "down", "left" or "right" and the player is alive, the game wil call the moveTowards() method in said direction, then it will check if there is a collision with the map in the new position; if there is, it will call again the moveTowards() method but this time in the opposite direction. Note that these are just calculations that are faster than the actual render. And lastly, animateMovement() will be called.

### moveEnemy

It's implemented in its own way in each enemy as each has different ways to move; on the big picture, we could split it in two parts. The first part is "what direction does the enemy need to move" and the second "calculate the movement in said direction". The second part, is basically the same as in moveTowards(). Now, the interesting thing is how the first part is implemented in each enemy:

- Balloon: There's a predefined int called TICK_TIME storing the value 250 (ms), a long variable called time storing `System.currentTimeMillis()`. In the class MovableEntity, there's defined a `String[] entityDir` with values {"down", "down"}, and finally there's an `int` called `dir` uninitialized. moveEnemy will first try to check if (time > TICK_TIME), if it is, moveEnemy will check whether dir is 0, 1, 2 or 3 (each representing "up", "down", "left" and "right" respectively). As it is uninitialized, it takes the value 0 and let's use it as an example of what will happen in each case: First it will try to move the enemy position towards the direction that the dir is representing, then it will update the entityDir's first value to take the last value, and the last value to take the actual direction. This way entityDir stores the previous movement and the present one. Then it will check if the enemy is colliding with the map, and if it is, it will change its position to the previous one (moving it towards the opposite direction). And also, if it collides, 'dir' will take a random value between 0 and 3. Now, having finished the conditional of which value is the 'dir', the game updates the sprite to be used with animateMovement and sets 'time' to `System.currentTimeMillis()`.
- Slime: It will check if the slime's x-axis position is bigger than the player's and if so, move towards that direction, and if it collides, move it towards one time towards the opposite. Then it will check if the slime's y-axis position is bigger than the player's and if so, move towards that direction, and if it collides, move it towards one time towards the opposite. Also it will check if the slime's x-axis position is smaller than the player's and if so, move towards that direction, and if it collides, move it towards one time towards the opposite. And lastly it will check if the slime's y-axis position is smaller than the player's and if so, move towards that direction, and if it collides, move it towards one time towards the opposite. In all this, updating the entityDir array. Finally it will animateMovement()
- AntiBomberman: Firstly, it updates its speed to the player's speed. Gets the player's last action and if it's a movement string, assign it to a lastAction string variable and finally call `moveEnemyAction` which if the AntiBomberman is alive and the player is alive and the lastAction is a direction, it will set entityDir with the updated values, animateMovement() and calculate the new position for the bomberman, with the moveTowards() checking if each if it's colliding to move it to the other way. In macro we could say that the AntiBomberman is imitating the player's movement. The action of the user is stored into game so that this enemy can read it after it has been consumed.

### animateMovement

Each MovableEntity has an int `spriteRgQty` that stores the amount of sprites the entity has for each direction of movement, which must be equal between them (i.e. the player has 5 sprites to move down, up, left or right each, the enemies have 3 sprites.). We will also use the int spritePhase, which must be between 0 and spriteRgQty. Lastly we must understand how the sprite path names are configured: it's a string that follows this scheme: NAME + DIRECTION + SPRITE NUMBER + ".png". Where NAME is the name of the entity (i.e. "Bomberman", "Enemy1", "Enemy2", "AntiBomberman"), DIRECTION is the string of a number (i.e "0", "1", "2"or "3" meaning up, down, left or right respectively) and SPRITE NUMBER meaning the span of how much sprites does that movement has. (i.e. "1", "2", "3", "4" and "5" for a total of 5 sprites which together animates the movement towards DIRECTION). Now, having understood how the name works, we can proceed with the algorithm. Assuming the reader has read the previous algorithms, you must have some idea of the entityDir array. Well, here

we have a conditional of whether the previous and present directions are equal or not. If they are not equal, the spritePhase will be set to 1 and setSprite() will be called, setting the sprite to NAME + present direction + “1.png”. If the previous and present directions are equal in entityDir (meaning that the actual sprite has already the number of the movement written), we will check if the spritePhase is still smaller than the spriteRgQty. If it is not, we will set it to 1 and reset the sprite to NAME + direction + “1.png”. Now, if the previous movement is equal to the present one and the spritePhase is smaller than the spriteRgQty, we will sum one to spritePhase and set the sprite to: NAME + direction + spritePhase + “.png”. And this is an **abstract** understanding of how animateMovement works.

# 4. Functionality

The map is generated in the same way as the demo, and the basic rules (wall collisions, brick destruction, player’s abilities, etc.) are all satisfied.

When an explosion reaches a bomb, the bomb explodes. However, if the bomb is “killed” by any other means (as through `Entity.removeEntity(int)`) before it explodes, it will not explode.

The player interacts with the environment in a proper way: enemies hurt them, they place the number of bombs they can, they act according to their stats, etc.

All bonuses appear the same way and with the same frequency as in the demo and have the same effects. The time bonus for completing a level is 1 point per second before 3 minutes since entering the level.

All enemies behave properly. A new enemy has been created: Antibomberman.

> The antibomberman has a 10% chance of appearing at any level. It imitates the movement of the player, but oppositely. It has 500 health and 50 damage, what makes it kill the player almost instantly.

The log will print messages in the following situations:

- When the player is on an explosion
- When an enemy is killed (alongside its type, id and score)
- When the player successfully crosses a door (alongside the bonus score)
- When the player cannot cross a door
- When a command is entered

All the commands of the demo provided are implemented in this one, plus some extra commands:

- `detonate` Explodes all placed bombs
- `killall` Kills all alive enemies
- `clear` Clears the command log
- `win` Shows the win screen and ends the game
- `exit` Closes the program

# 5. Conclusion

In conclusion, we have managed to solve all the problems proposed, following a mixture between object oriented (data structure) and structured (runtime flow) programming.

We faced several problems, and have been able to solve them in different ways. One of the most difficult ones was the animation of the sprites: it was hard to choose between doing it inside the proper class or in the runtime flow. Also, since we made a deep abstraction work from the beginning, we found so much abstraction was not needed for the dimensions of the project, and during development we had to reduce overcomplicated ideas.

We also think that some features are bad implemented:

- Some of the previously mentioned overcomplications that we did not remove are unnecessary (for instance, the different between Entity and MovableEntity was not really a need given the requirements for the project).
- We developed the project mainly by turns using a github repository. This was most convenient given the long distance between us, but as the communication was not as perfect as face to face, some problems were solved twice, one time for each one of us, what lead to some redundant methods and solutions.
- The ID management of the entities works, but is not efficient when implementing bigger projects. The fact that the id increments every time a new entity is created makes it impossible to play the game for a long time, as it is session-dependant. Also, the sprites of removed entities are never deleted from the API's list; instead, they are just hidden. This can lead to performance problems or even crashes.