[PHASE 2 REPORT - GROUP 25]

# ◆COSMIC ESCAPE◆

*Ananga Bajgai, Annie Boltwood, Mohamed Mustafa, Austin Yu*

---

**GAME PLANNING**

1. [Milestone 1 - Midway Point]
2. [Milestone 2 - Implemented Game]

## [1] MILESTONE 1 - MIDWAY POINT

*The primary focus for this first milestone was to plan out our game and create the basic functionalities*. In order to create this project in an organized manner, we delegated tasks, utilized 'to-do' lists, and used a planned communication system to ensure we did not have conflicting pushes on GitHub.

### [1.1] Basic Project Layout

The first part was for us to lay down the groundwork of our project in order to create a structured and organized base. The steps are as follows:

- ➢ Team strategizing regarding our approach on the project.
- ➢ Creating the maven project and pushing it to the remote repository.
- ➢ Creating the classes to match our UML diagram:
  - Game Class
  - Factory Classes
  - Subclasses
- ➢ We noted that these classes were subject to change and would not mirror the UML diagram exactly as we progressed throughout the project.
- ➢ Populating the classes with their core methods and variables.
- ➢ The project was created with a modular design meaning that each class had its own file. After creating our project we one-by-one created the files and their class.
- ➢ Creating a dedicated "resources" folder inside the **"main/src"** directory of our project. This folder is a designated space for storing resources and assets, in this case images for our game sprites.
- ➢ Researching the most optimal GUI to use when building our project. We chose to use **Java Swing** because it is versatile and meets the needs of our project in terms of GUI functionalities.

### [1.2] GUI and Game Clock

The next part focuses on the implementation of the GUI and the game clock, which are essential components for the game. Creating these main elements were necessary before implementing other parts of the game.

**[Graphical User Interface]**

- ➢ We chose to create a Jframe inside of the **Game** class constructor along with a Jpanel where the game graphics would be displayed, meaning that for every instance of the game class a new window would be created. At this point we decided to stray from the UML diagram slightly, realizing we did not need a BoardStateFactory, because the GUI and Clock would be housed in the **Game** class.

**[Game Clock]**

- ➢ We then created the game clock in the **Game** class constructor that keeps track of the ticks per second while the game is active.

## [1.3] Entities and Entity Factory

As we studied the game's structure and mechanics, it was evident that separating the types of objects that can be inserted and removed in the games' map is the optimal way to ensure a dynamic and clear structure for both the player and the developer. The **entities** are elements with which the player interacts, including enemies, barriers, rewards, and the main character. The **EntityFactory** class instantizes these elements using the factory design pattern, which creates entities and organizes them in an ArrayList for orderly tracking of each entity.

**[Main Character and Movement]**

- ➢ In this part we implemented the main character class and the user controlled movements of the character. In the **MainCharacter** class the object is given x/y coordinates, an image to represent it on screen, speed, and methods that move it on screen. We added an action listener for the event that the up/down/left/right key is pressed that calls the main character's movement methods.

**[Enemies]**

- ➢ In Cosmic Escape, enemies are hardwired to thwart the player's mission back to the Space Station. Two types of enemies were created and inherited from the **Enemy** class to exist in the cosmic abyss: meteors and aliens, which are moving enemies and punishments respectively. The **MovingEnemy** class was implemented to have a move function which takes in two private movement speed parameters to adjust the locomotion of the meteor using the MovingEnemy's position. The alien uses the **Punishment** class which was implemented to have an adjustable and gettable scoreDecrease variable for when the main character makes contact with the alien. Since both the **MovingEnemy** class and the **Punishment** class inherit the **Enemy** class, the enemies' position attributes are stored in the **Enemy** class and, which allows it to be easily be accessed from another entity in situations where the main character makes contact with an enemy or an enemy makes contact with a barrier.

## [2] MILESTONE 2 - IMPLEMENTED GAME

*The second milestone focused on turning the game's implementation from a framework of ideas and entities to an actualized, immersive game*. We moved from conceptual frameworks to concrete gameplay with a score system, obtaining rewards, and winning/losing, creating an immersive experience for the player. This phase was marked by the meticulous integration of the game's core mechanics, fine-tuning of the user interface, and careful testing to ensure a polished game.

## [2.1] Entity Mechanisms

**[Main Character]**

- ➢ The main character has been established in Milestone 2 to allow the player to traverse the map with the versatility of a spaceship, combining speed and agility. For collision detection, the **Main Character** class contains isContact functions for each type of entity such as moving enemies and barriers, which dictate whether to stop movement or end the game respectively. Central to the game is the collection of keys; the Main Character uses a keys_tracker dictionary to keep track of each key colour collected, and a count to signal when all the key colours have been collected.

**[Rewards]**

- ➢ The two types of rewards, RegularRewards (Keys) and Bonus Rewards (Stars) were both implemented modularly, extending the Reward class. This allows the information about rewards such as positioning and score amounts to be taken from one class instead of repeated in the **RegularReward** and **BonusReward** subclasses. The rewards not only change the score according to the type of reward, but also change the player's path as they navigate through the map to find keys. This serves as the marker of progress for the player as they try to find the unique key colours to return to the Space Station. The unique key colours were implemented by tracking the number of each key colour collected in a hashtable, making it easily accessible in situations such as counting the unique keys to open the Space Station.

**[Moving Enemy]**

- ➢ Originally, the moving enemies used a simple algorithm to traverse in the direction of the Main Character, notwithstanding the requirement of moving only in the up, left, right, or down instructions for the project. In Milestone 2, we adjusted and implemented a new algorithm for the traversal of the **Moving Enemy** which finds the direction with the shortest path to the enemy and moves in that X or Y direction, eliminating the disorganized diagonal movements of Milestone 1.

**[Punishment]**

- ➢ In Milestone 2, **Punishments** were balanced to serve an optimal amount of reduction to the player's score upon contact. They were implemented to reduce the score by 5 each tick, with the isPunishmentContact function being in the Main Character class. This allows for a desire for the player to survive at the beginning by avoiding all elements and earning stars which increase the score. Later on when the player has amassed a high score, the Punishment is designed to provide a way out for when a player is trapped by moving enemies, with credit to the optimal amount of score reduction.

**[2.2] User Interface**

- ➢ The Starting Screen is now implemented in Milestone 2. It uses **jFrame** functionality from **Swift** to create the main frame of the game, and a game panel which extends the **JPanel** for rendering the elements of the game. Within the **GamePanel**, the paintComponent method is overridden to draw the background of the game and the entities. The different game states, PLAY, START_SCREEN, GAMEOVER, and WIN, each correspond to different GamePanels that are drawn. The different game states are displayed whenever events happen like the player's score reaches 0 or the player wins the game. The game states are changed from the Game class respectively.
- ➢ In addition, the score and keys collected interface is placed at the top right of the screen, and the position is designed to stay at the top right regardless of the resizing of the game window by using a resize **component listener**. Various images and design elements were also implemented for the different game states, and in-game entities and the map.
- ➢ Swing was chosen as the graphical user interface (G.U.I.) because it provided exceptional platform-independent G.U.I. components, and supports advanced graphics for creating a game. The integration of rendering GamePanels allows for easily setting game states depending on events like Start Screen, Game Over, and Win.

**[2.3] Factory Method Design**

- ➢ We implemented the **Factory Method Design** pattern to create the in-game entities and interfaces. The abstraction of entity instantiation allowed us to encapsulate the object creation logic within respective factories, resulting in a modular and extendable structure that accommodates additional entity types when needed during the creation process. **EntityFactory** and **BoardStateFactory** are factories which we used to instantiate the objects depending on whether they are in-game entities or user interface elements. This allows us to create the objects using a common interface and hide the complexities of the creation process from its subclasses. Moreover, these subclasses benefitted as they could override methods such as the draw method, which draws the entity onto the board, depending on the entity's specific requirements. In connection, the GamePanel does not have to create the objects when the game is in the playing state, leading to improved performance and memory usage. There were many cases where we had to change an entity's creation method easily without disrupting the entity class itself, such as when passing X and Y positions to the EntityFactory to instantiate the Main Character.

**[2.4] Challenges**

- During Milestone 2, various challenges arose during implementation of moving algorithms, entity drawing, and entity positions. Although, currently the moving enemy is capable of following the player, the movement isn't currently perfect. There are some bugs when at specific positions of the player, the enemy might get stuck at a barrier. Hopefully, once we are able to properly implement the Map class, some sort of A* algorithm may be implemented to improve the moving enemies movement. We also had challenges with drawing the entities and placing it on the map. The entities had to be of precise dimensions and placed in precise coordinates. There are still issues, with the rewards randomly spawning on the barriers sometimes. The collisions between entities could also be improved, but again with the implementation of a Map class, this might get solved.
- Another challenge we had was following our initial design plans. For instance classes like BoardStateFactory and MenuInterface, never were required, unlike initially thought in the UML diagram. Conversely, there were various other methods, we added to our classes, that we had not anticipated. Although we did our very best in the design stage, some of the changes from initial design plans were inevitable. Overall though, most of our core design plans do go in-line with our currently implemented plans.