Architecture - Runtime Errors - Team 25

Names:

Charles Stubbs

Alex Shore

Annabelle Partis

Kieran Ashton

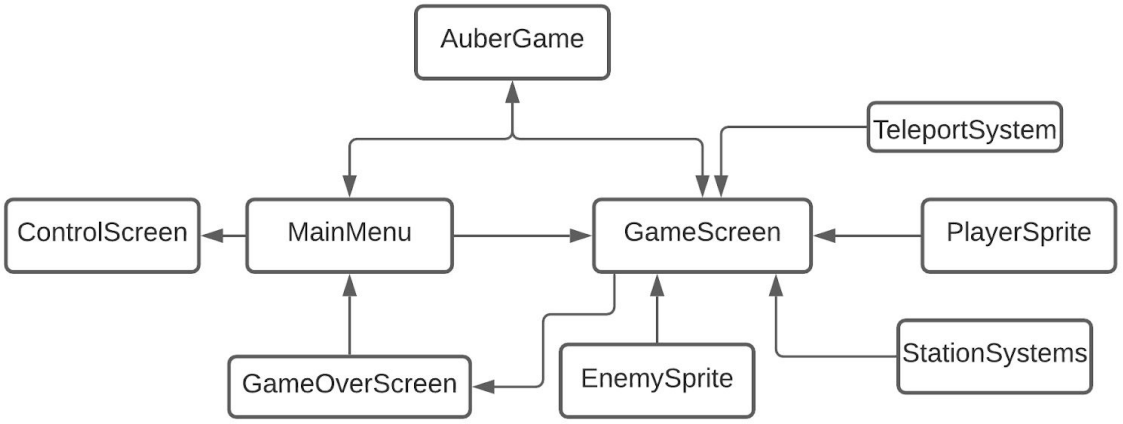
Yu Li

George Tassou

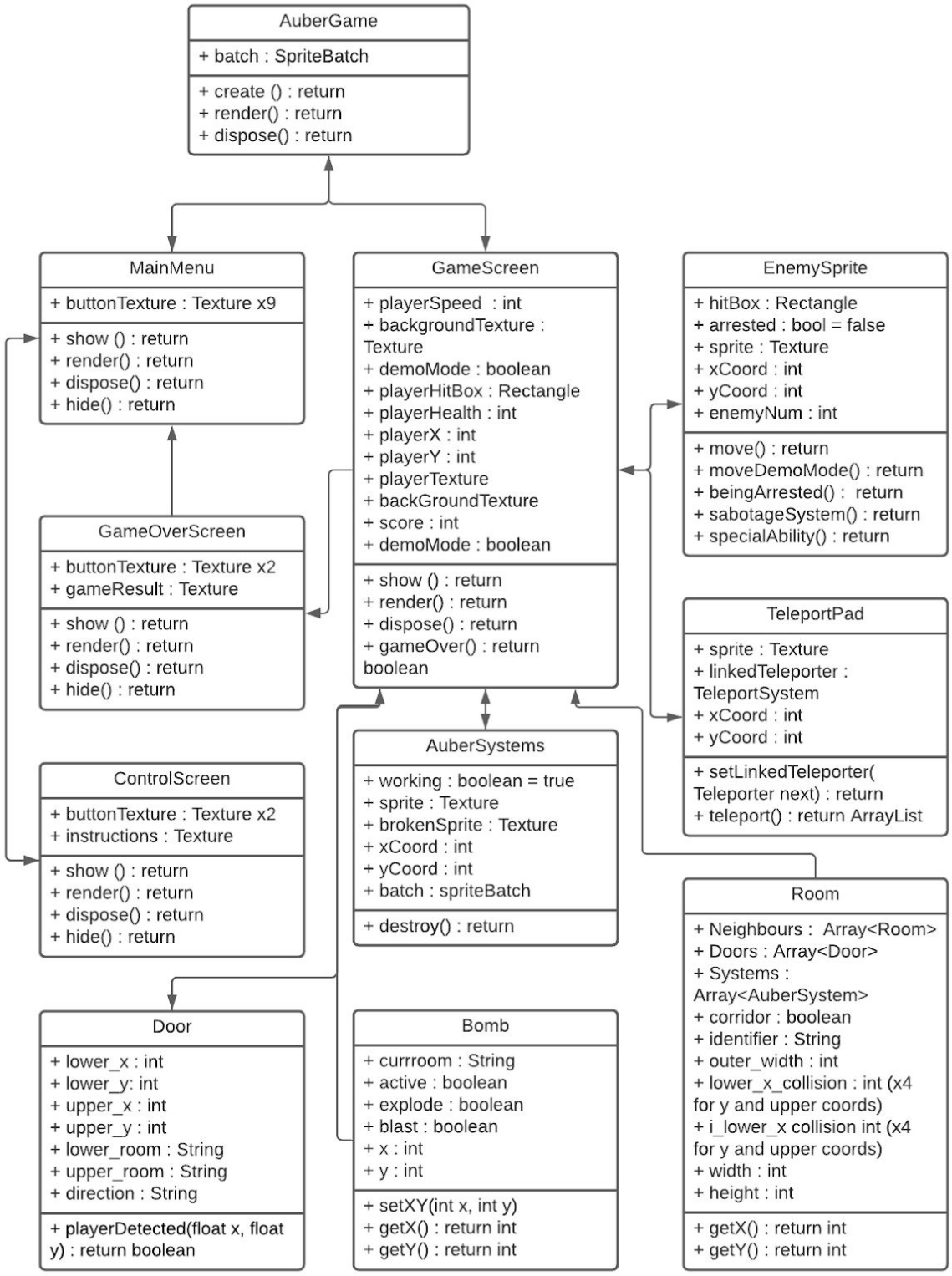
Part A

Both the concrete and abstract representation give insight into the structure of our game. The main functions that run the game will be called from the AuberGame and the GameScreen classes. For our abstract representation, we have broken down the code required in our game into classes and modelled them using a UML structure diagram which includes all the classes and functions required to implement the functionality stated in the requirements. The concrete representation of our classes was also modelled in UML. We followed class diagram conventions to ensure it was easy to understand. The arrows in the representation represent association, not inheritance. In the concrete architecture, we broke our classes down into variables and functions, specifying their data types and parameters respectively, to make them easier to translate into code. In order to follow good data flow principles, the functions relating to each sprite object will be stored inside the relevant class. Our implementation is supported by the library LibGDx, a popular repository for 2D and 3D games in Java. We have to instantiate some objects from this library in our implementation. These are included in our concrete representation, but their attributes and methods will not be listed because we do not have to write them.

Abstract Representation



Concrete Representation



Part B

The concrete architecture was built using a combination of the abstract architecture and the requirements. Most of the classes defined in the abstract architecture were retained, but the PlayerSprite class was not, as we decided that it would be easier to implement player movement directly in the GameScreen class. Thus it contains all the variables responsible for generating and moving the player sprite around the screen, including the x and y coordinates specified in requirement FR_MOVEMENT. The GameScreen class also contains the variable “score”, which was suggested in requirement FR_SCORE, and was also identified in FR_LOSS_STATE_SABOTAGE as a way of telling whether the game has reached the loss state. A boolean variable demoMode was also included in GameScreen to trigger the demo mode specified in requirement FR_DEMO. This class also includes the collision implementation code, which also prevents the player sprite from leaving the bounds of the map, obeying requirement FR_BOUNDARIES.

The concrete architecture for the MainMenu class includes 9 texture variables. Eight of these are for the buttons on the home screen, one of which links to the GameScreen class which runs the game, another to the ControlScreen class which displays the instructions to the user as required by UR_CONTROL_SCREEN, and another labelled Demo mode, which runs GameScreen with the boolean value of demoMode set to true. The final button exits the game.

The code in EnemySprite was not relegated to GameScreen as there will need to be multiple instances of enemies. The position of the enemy is given by x and y coordinate variables and the movement is controlled by the move and moveDemo() functions which allow for the ai of the enemies to adapt to demo mode, as recommended in the FR_DEMO requirement. Each enemy also has a boolean variable “arrested”. Setting this to true sends the enemy a certain spot in the map and prevents them from moving, providing the functionality for enemies to be arrested, which is required to fulfil the game’s win condition specified by FR_WIN_STATE. The requirement UR_SPECIAL_ABILITIES specifies that the enemies must have at least 3 distinct special abilities, hence the function specialAbility(). This will contain code for all of the possible special abilities, which will be selected between using a switch case statement based on enemyNum variable. GameScreen also has function gameOver() which returns either true or false depending on whether the game has reached any of the end game states defined in requirements FR_WIN_STATE, FR_LOSS_STATE_SABOTAGE, and FR_LOSS_STATE_DEATH. If so, the game will be ended and a GameOverScreen object will be instantiated.

Requirement FR_TELEPORT_PADS specifies that the user must be able to move around the station via teleport pads. These are implemented with the TeleportPad class that has been included in both the abstract and concrete architecture. UR_INFIRMARY also requires teleport pads to be implemented. Each TeleportPad object will have a variable linkedTeleporter which points to another TeleportPad object which stores the coordinates the player should come out at.

The class System, identified in both the abstract and concrete architecture comes requirement FR_LOSS_STATE_SABOTAGE, which specifies that there must be systems that the enemies can sabotage, which, upon all of them being destroyed, will end the game. Hence System has a boolean variable “working” which will turn to false if the system has

been sabotaged. It also has a function destroy() which will be called upon destruction which will set “working” to false.

GameOverScreen is an essential part of the project, hence is included in both the abstract and concrete implementation. It instantiated when GameScreen identifies that the game has reached any of the end game states defined in requirements FR_WIN_STATE, FR_LOSS_STATE_SABOTAGE, and FR_LOSS_STATE_DEATH, and will load a different texture into the gameResult variable depending on the outcome.

The requirement NFR_FIRST_TIME_USER specifies that a first time user with no experience with the game should be able to pick it up without difficulty. Hence we have included a ControlScreen class that creates a window that can be accessed via a button from the main menu. It loads a texture listing the controls of the game into the variable “instructions” and displays it on the screen.

Requirement UR_SPECIAL_ABILITIES specifies that the enemies are required to have at least three special abilities between them. Most of these are managed by the enemy class, however one of the ones we have chosen involves the enemy placing a bomb, which detonates and harms the player. This functionality is implemented by the Bomb class.

UR_ROOM_VARIETY demands that there be at least four different types of station in the game’s map. The Room class provides the data to construct the collision systems for these different rooms. Requirement UR_INFIRMARY specifies that there must be an infirmary. Thus this is included in the objects created using this class.

The Door, Bomb, and Room functions were added in the concrete architecture to implement technical aspects of the game, such as collision detection.