2020

Architecture

TEAM ZANETTA

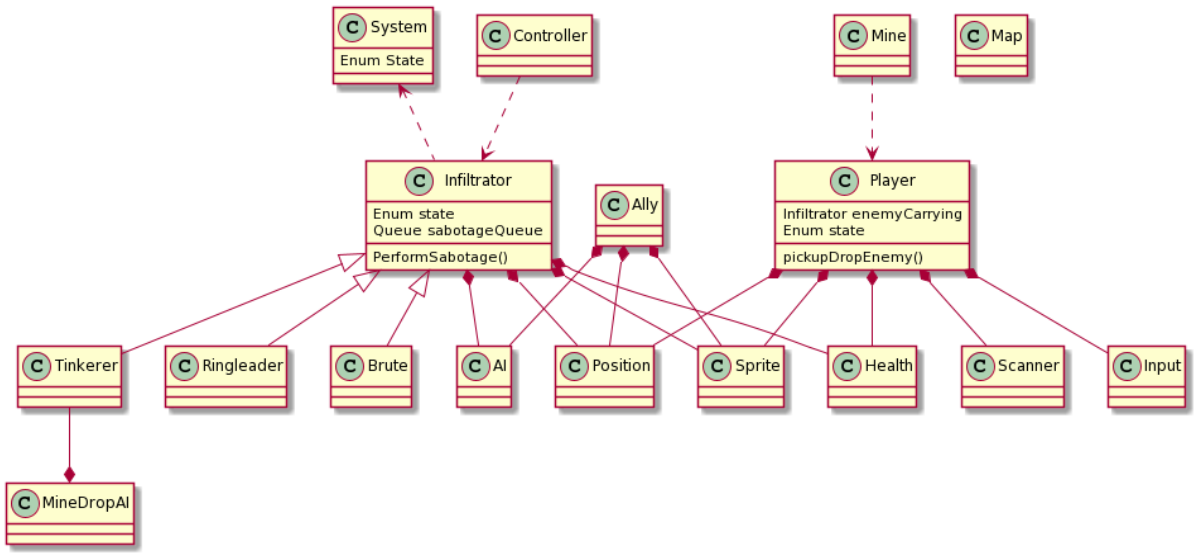
HARRY KELLY, TIMOTHY KERR-CHIN, JOE SANDERS, SIYAAM MAHMOOD, EMILY TIERNEY & ALEXANDRE PINNEAU

Part A

Abstract

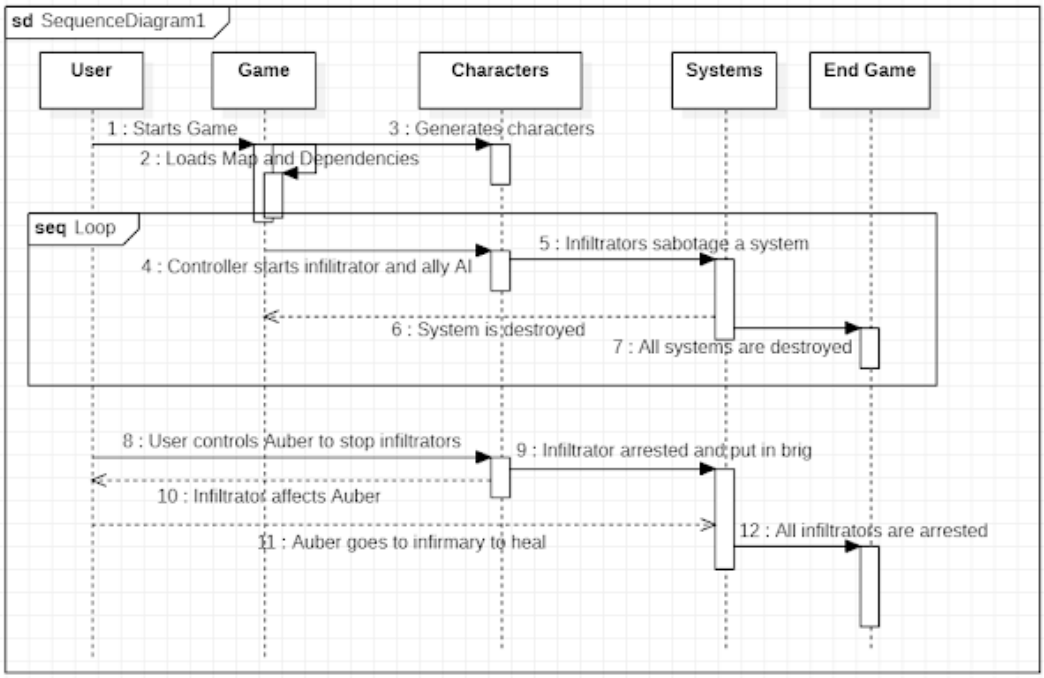
We have decided to use PlantUML to map out a static perspective of the module views. We chose to use an OOP entity-class-diagram to represent the different objects and systems and how they interact.

Static Perspective (classes, functions, interfaces)



The above OOP entity-class-diagram gives an illustration of how we imagine classes will look like. We would choose an inherited based system due common properties such as AI pathfinding for both the infiltrators and the allies.

Runtime Perspective

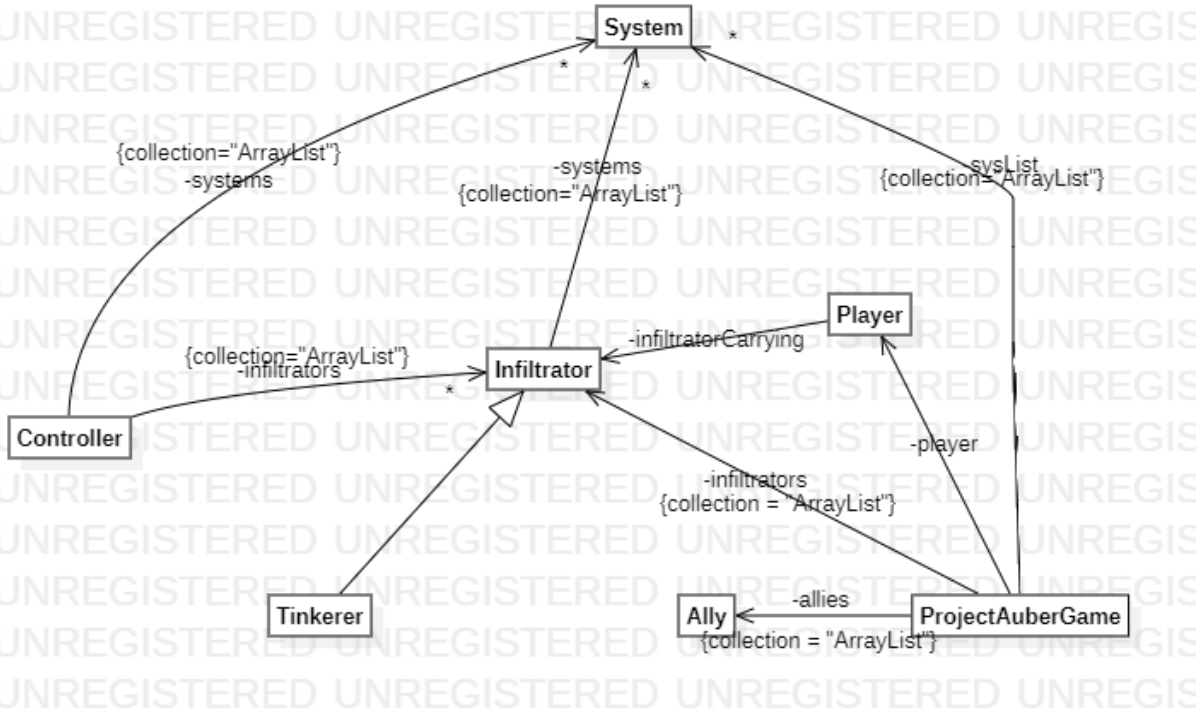


Fundamentally this game can be run in both demo mode and with a user. With this in mind we decided that the best approach would be to have 2 primary game loops, with the second being independent from the first. The first loop describes how the game would implement the scenario of an infiltrator win, which is made possible through the destruction of all the systems in place. The second half of the sequence diagram gives a description of how the player interacts with the infiltrators. Both halves to the sequence diagram have a path where the game can end, the first being a user loss, the second being a user victory.

Statement about editors

We decided to create the Run Perspective Sequence Diagram and the Concrete Software Architecture Class Diagram using **StarUML** which utilises the UML programming language. We chose this application due to its simplicity and that java code could be reversed engineered into a diagram.

Concrete Architecture class diagram



The game's implementation as of now has some key similarities and differences to how we originally planned to implement it. The inheritance-based structure has been kept, but a lot of the elements we originally planned to implement as interfaces were found to already be a part of LibGDX's classes (such as Actor). The structure currently consists of three core classes, all of which are subclasses of LibGDX's Actor class: Player, Infiltrator, and System. All three classes implement the interface Sprite, which has some basic methods for handling textures. Project Auber Game holds arrays of allies, systems, and infiltrators, as well as a variable for player. Player has a variable of the infiltrator being carried. Infiltrators have lists of systems they can sabotage, and the controller has a list of infiltrators and systems. There are a series of methods for interactions between the classes, allowing players to attack and pick up infiltrators, infiltrators to sabotage systems, and the controller to tell infiltrators when

to sabotage systems. The main behaviour of allies and tinkerers take place within their classes.

Part b

We stuck to an inheritance based style, creating different classes that stemmed from main class types like Sprite, Texture and Actor. We decided that we would use an inherited based structure for code reusability and reliability. This became apparent to us when we decided to have different infiltrators who all shared some key properties. In this way it was also more reliable than implementing 4 different types of infiltrators each with their attributes and methods.

Although a downside to using an inheritance based style is that if one were to make a change in a base class then that could cause a ripple effect into it's child classes; our main base classes were not large, only linked the subclasses or gave them characteristics that all similar class types were to have. Therefore a change was unlikely to be made.

In the UML diagrams there is a Infiltrator class along with 3 types of infiltrators that extend from it and a controller class which also extends from it. This fulfils the requirement "ur_operative" that calls for 8 infiltrators with 3 different types that roam around sabotaging systems. Within the Player class, there are two methods, keyUp and keyDown, that relate to the requirements "ur_input" and "ur_operability" making the game easily operable with use of only the movement keys and the spacebar. Through achieving this requirement this class also relates to "FR_MOVE" allowing users to move up, down, left and right using the movement keys.

The ProjectAuberGame class holds the render method that carries out the requirement "FR_TRAP_PLACE" which has the tinkerer actor place mines for Auber. Requirement "FR_TASER" was implemented in the Player class in the method attackInfiltrator. "FR_SCAN" was also in the Player class implemented through the methods scan and unScanAll. Extended from the parent class Infiltrator, the requirement "FR_IMPOSTER_TYPES" was met in the different classes Tinkerer, Brute, and Ringleader Also, within the Infiltrator class, the methods performSabotage and queueSabotage relate to the requirement "FR_SABOTAGE". The final requirement that was implemented is "FR_SHOCK". This was implemented in the methods infiltratorCarrying, attackInfiltrator, pickupDropInfiltrator.