ARCHITECTURE

ENG12020TEAM24

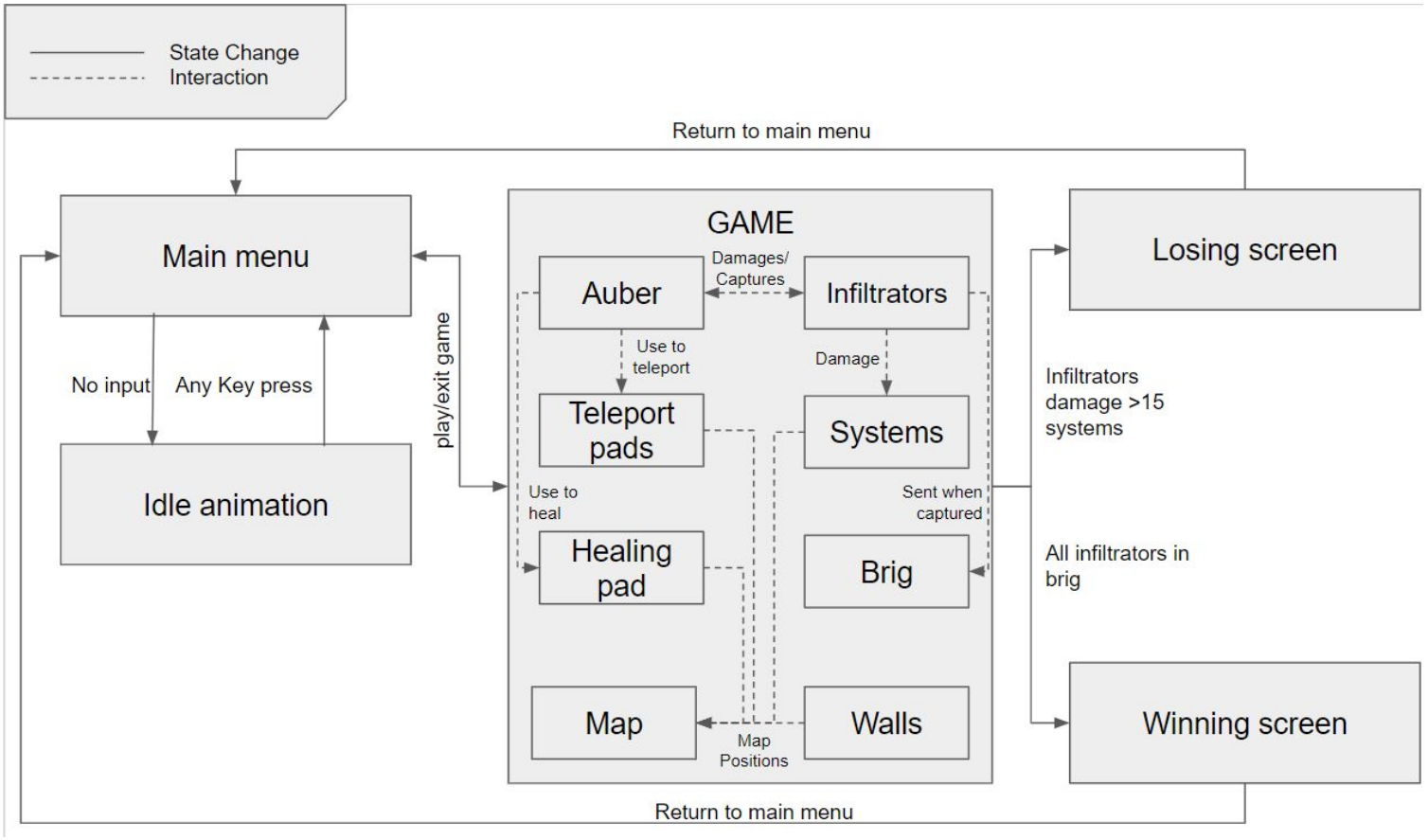
Daniel Allinson
Pratham Bhat
Eghosasere Ewansiha-Vlachavas
Tara Harley
Mahir Rahman
Kyle Wilson

Architecture

The purpose of this document is to connect our requirements of the software to the implementation. This will describe the objects, functions and attributes required to display the flow of the software.

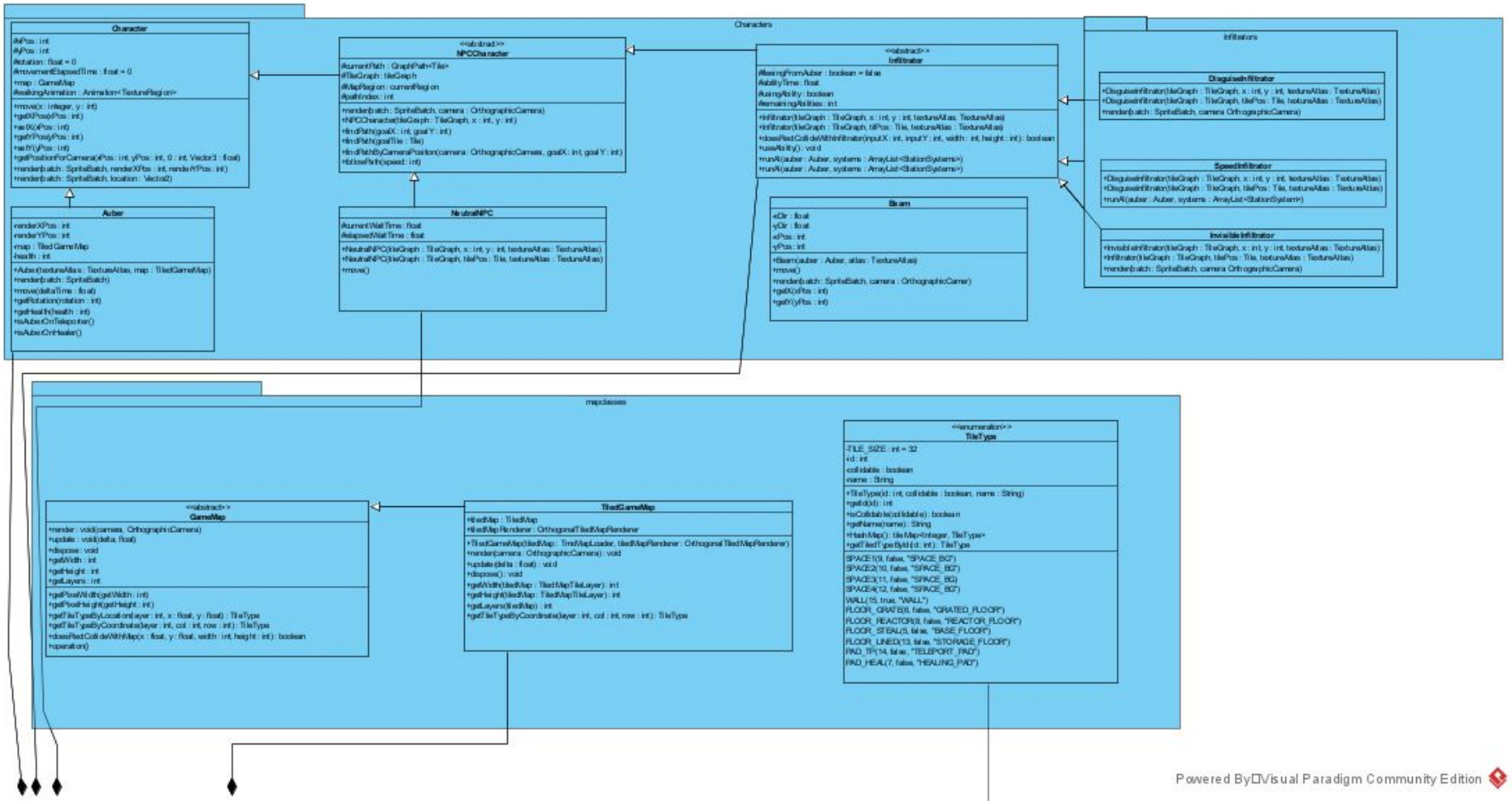
Abstract Architecture

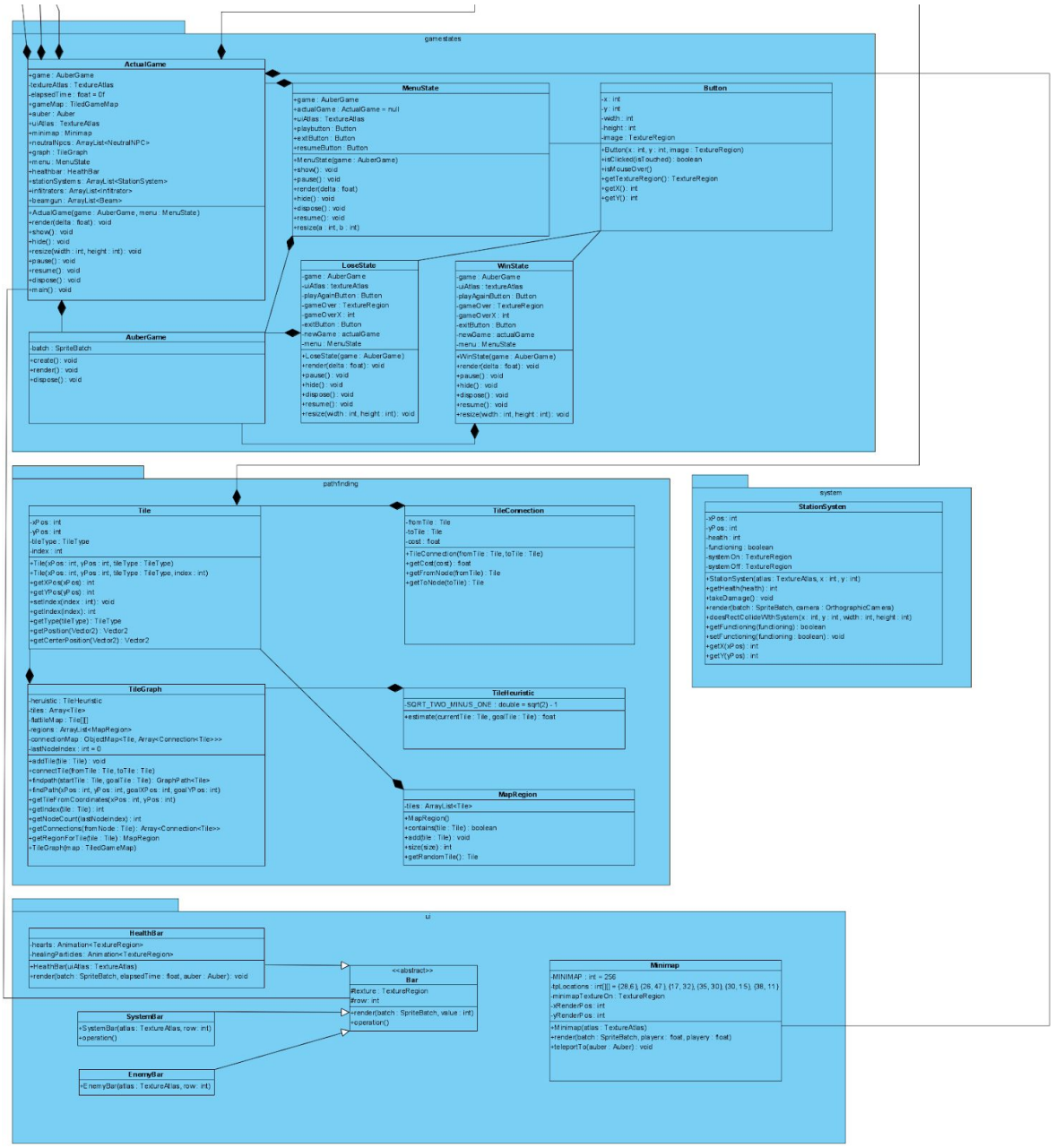
A prescriptive abstraction of the software that reflects decisions made prior to implementation. This was created using Google Slides.



Concrete Architecture

A descriptive concrete model of the software which reflects implementation decisions that will be closely linked to the infrastructure as well. These models will closely reflect the code and design of the game as implemented using libGDX. In addition, we will also be using methods from within libGDX and its associated libraries, such as gdxAI in order to complete the project. This was created using Visual Paradigm Community Edition. The .vpp file used to create this will be available to download from our website, along with a vector image of the concrete architecture.




gamestates
ActualGame
+game : AuberGame
-textureAtlas : TextureAtlas
-elapsedTime : float = 0f
+gameMap : TiledGameMap
+auber : Auber
-uiAtlas : TextureAtlas
+minimap : Minimap
+neutralNpcs : ArrayList<NeutralNPC>
+graph : TileGraph
+menu : MenuState
+healthbar : HealthBar
+stationSystems : ArrayList<StationSystem>
+infiltrators: ArrayList<infiltrator>
+beamgun : ArrayList<Beam>
+ActualGame(game : AuberGame, menu : MenuState)
+render(delta : float) : void
+show() : void
+hide() : void
+resize(width : int, height : int) : void
+pause() : void
+resume() : void
+dispose() : void
+main() : void
MenuState
+game : AuberGame
+actualGame : ActualGame = null
+uiAtlas : TextureAtlas
+playbutton : Button
+exitButton : Button
+resumeButton : Button
+MenuState(game : AuberGame)
+show() : void
+pause() : void
+render(delta : float)
+hide() : void
+dispose() : void
+resume() : void
+resize(a : int, b : int)
Button
-x : int
-y : int
-width : int
-height : int
-image : TextureRegion
+Button(x : int, y : int, image : TextureRegion)
+isClicked(isTouched) : boolean
+isMouseOver()
+getTextureRegion() : TextureRegion
+getX() : int
+getY() : int
LoseState
-game : AuberGame
-uiAtlas : textureAtlas
-playAgainButton : Button
-gameOver : TextureRegion
-gameOverX : int
-exitButton : Button
-newGame : actualGame
-menu : MenuState
+LoseState(game : AuberGame)
+render(delta : float) : void
+pause() : void
+hide() : void
+dispose() : void
+resume() : void
+resize(width : int, height : int) : void
WinState
-game : AuberGame
-uiAtlas : textureAtlas
-playAgainButton : Button
-gameOver : TextureRegion
-gameOverX : int
-exitButton : Button
-newGame : actualGame
-menu : MenuState
+WinState(game : AuberGame)
+render(delta : float) : void
+pause() : void
+hide() : void
+dispose() : void
+resume() : void
+resize(width : int, height : int) : void
AuberGame
-batch : SpriteBatch
+create() : void
+render() : void
+dispose() : void
pathfinding
Tile
-xPos : int
-yPos : int
-tileType : TileType
-index : int
+Tile(xPos : int, yPos : int, tileType : TileType)
+Tile(xPos : int, yPos : int, tileType : TileType, index : int)
+getXPos(xPos) : int
+getYPos(yPos) : int
+setIndex(index : int) : void
+getIndex(index) : int
+getType(tileType) : TileType
+getPosition(Vector2) : Vector2
+getCenterPosition(Vector2) : Vector2
TileConnection
-fromTile : Tile
-toTile : Tile
-cost : float
+TileConnection(fromTile : Tile, toTile : Tile)
+getCost(cost) : float
+getFromNode(fromTile) : Tile
+getToNode(toTile) : Tile
TileGraph
-heruistic : TileHeuristic
-tiles : Array<Tile>
-flatTileMap : Tile[][]
-regions : ArrayList<MapRegion>
-connectionMap : ObjectMap<Tile, Array<Connection<Tile>>>
-lastNodeIndex : int = 0
+addTile(tile : Tile) : void
+connectTile(fromTile : Tile, toTile : Tile)
+findpath(startTile : Tile, goalTile : Tile) : GraphPath<Tile>
+findPath(xPos : int, yPos : int, goalXPos: int, goalYPos: int)
+getTileFromCoordinates(xPos : int, yPos : int)
+getIndex(tile : Tile) : int
+getNodeCount(lastNodeIndex) : int
+getConnections(fromNode : Tile) : Array<Connection<Tile>>
+getRegionForTile(tile : Tile) : MapRegion
+TileGraph(map : TiledGameMap)
TileHeuristic
-SQRT_TWO_MINUS_ONE : double = sqrt(2) - 1
+estimate(currentTile : Tile, goalTile : Tile) : float
MapRegion
-tiles : ArrayList<Tile>
+MapRegion()
+contains(tile : Tile) : boolean
+add(tile : Tile) : void
+size(size) : int
+getRandomTile() : Tile
system
StationSystem
-xPos : int
-yPos : int
-health : int
-functioning : boolean
-systemOn : TextureRegion
-systemOff : TextureRegion
+StationSystem(atlas : TextureAtlas, x : int, y : int)
+getHealth(health) : int
+takeDamage() : void
+render(batch : SpriteBatch, camera : OrthographicCamera)
+doesRectCollideWithSystem(x : int, y : int, width : int, height : int)
+getFunctioning(functioning) : boolean
+setFunctioning(functioning : boolean) : void
+getX(xPos) : int
+getY(yPos) : int
ui
HealthBar
-hearts : Animation<TextureRegion>
-healingParticles : Animation<TextureRegion>
+HealthBar(uiAtlas : TextureAtlas)
+render(batch : SpriteBatch, elapsedTime : float, auber : Auber) : void
<<abstract>>
Bar
#texture : TextureRegion
#row : int
+render(batch : SpriteBatch, value : int)
+operation()
SystemBar
+SystemBar(atlas : TextureAtlas, row: int)
+operation()
EnemyBar
+EnemyBar(atlas : TextureAtlas, row: int)
Minimap
-MINIMAP : int = 256
-tpLocations : int[][] = {28,6}, {26, 47}, {17, 32}, {35, 30}, {30, 15}, {38, 11}
-minimapTextureOn : TextureRegion
-xRenderPos: int
-yRenderPos: int
+Minimap(atlas : TextureAtlas)
+render(batch : SpriteBatch, playerx : float, playery : float)
+teleportTo(auber : Auber) : void

Architecture Justification

ECS vs Inheritance

We have taken an inheritance approach for the concrete architecture for a couple of reasons:

- Lack of experience and knowledge in designing software using an Entity-Component System. LibGDX supports ECS via the ashley library however we will not be using that, instead continuing with an inheritance class model using abstract classes.
- The complexity of the task: Auber is a relatively simple game in which the benefits of ECS do not affect the task. With very few entities an inheritance based approach is sufficient.

Requirements to Abstract

Justifying our abstract model and linking our components to our requirements:

- MAIN MENU: UR_GAME, FR_IDLE, the menu is the first screen/state of the game. In this state the user can move to the game state or if no input is entered then the game will go to the idle state.
- IDLE ANIMATION: FR_IDLE, this state is the idle state where the game plays itself or shows a video of the game being played. This will revert back to the menu state if anything is inputted.
- LOSING SCREEN: FR_LOSS, this is the game over state where once the player loses is sent to which will redirect to the menu to repeat the cycle of the game for the option to replay.
- WINNING SCREEN: FR_WIN, this is the victory game over state where once the player has won is sent to which then redirects back to the menu for the option to replay.
- GAME: UR_GAME, this is the game state, a container that holds all the components and entities of the game and the interactions between them.
- AUBER: FR_AUBER, this is the player character with links to what the player will interact with during the game.
- INFILTRATORS: FR_INFILTRATORS, this is the enemy entity, this shows the links of what the enemy can interact with.
- TELEPORT PADS: FR_TELEPORTER, these are objects that allow the player to teleport between on the map. This links to map as it is a map object.
- HEALING PADS: FR_RESPAWN, FR_INFIRMARY, this is a map object that allows the player to heal and respawn back into the game.
- SYSTEMS: FR_SHIP_SYSTEM, FR_LOSS, this is a map object that is interactable by the infiltrators. These are part of the requirements to lose the game when 15 are destroyed.
- BRIG: FR_BEAM, FR_WIN, this is an abstract location where infiltrators are sent to when they are beamed. We have decided not to implement a visible brig but note it as a virtual location.
- WALLS: FR_MAP, another map object, this object is just a collidable object to contain all the entities within the map.
- MAP: FR_MAP, this is an abstract component that represents the game map and all the objects part of it.

Abstract to Concrete

Explaining and justifying the concrete architecture built upon our abstract architecture.
The concrete architecture has been revised from the original, this was due to learning more about libGDX and not following the original concrete architecture. The current one now more closely resembles what we have been implementing in our project.
The concrete builds upon the abstract with the inclusion of attributes and methods for those components and entities.

CHARACTER:

- AUBER is the concrete version of the abstract AUBER with the necessary methods in the implementation for Auber to interact with the game world. It inherits from the Character superclass as Character contains the methods used to render Auber, however Auber's movement is handled within the Auber class itself.
- NPC CHARACTER and NEUTRAL NPC are the concrete components for FR_NPC. Neutral NPC contains the pathfinding code for the neutral NPC character to use, where characters are the object entity. We chose to use a common superclass between the neutral NPCs and the Infiltrators as they shared similar code, both using an AI.
- INFILTRATOR is the concrete version of the abstract INFILTRATOR, this is the parent of the various infiltrator subclasses with unique abilities. This contains the methods and attributes needed for the implementation and general infiltrator methods. We chose to use a superclass between the different Infiltrator types as they shared similar core code and AI, only differing in abilities.
- BEAM is a concrete object for FR_BEAM Auber uses to capture infiltrators.

MAP CLASSES:

- GAMEMAP, TILEDGAMEMAP and TILETYPE are concrete versions of abstract WALL and MAP, they contain all the attributes and methods for the game map and collision.

PATHFINDING

- TILE, TILEGRAPH, TILEHEURISTIC, TILECONNECTION, MAPREGION: these are all concrete classes for the abstract MAP that is used for path finding.
    - Note that the teleporter and healing pads are not their own class. They are just part of the map, interactions between Auber and them are directly inside the Auber class.

SYSTEM

- STATIONSYSTEM is a concrete version of the abstract SYSTEMS. We have decided systems have their own health bar which has been included in the attributes and methods for our implementation when being attacked by an infiltrator.

UI:

- BAR, HEALTHBAR, MINIMAP, SYSTEMBAR, ENEMYBAR are just User Interface objects that give information to the user about the game. This would fall under UR_GAME, as we believe a user interface is a core component of a video game. BAR is an abstract class used for storing the common methods and attributes between them, which is why it is a superclass to all of them.

STATES:

- AUBERGAME: the default state launched when the desktop launcher is executed, this creates the batch for the game used for rendering. This in a sense is the concrete version of the entire abstract model.
- MENUSTATE: this is the concrete version of the abstract MENU, this uses a button interface to exit and move to the game state.
- ACTUALGAME: this is the concrete version of the abstract GAME, this includes all the code for the interactions and flow of the game. The reason for a separate ActualGame and AuberGame is due to constraints discovered during the implementation.
- LOSESTATE, WINSTATE: these are concrete versions of the abstract LOSING SCREEN and WINNING SCREEN respectively, they are both gameover states with similar attributes and methods. These do not share a common superclass as they both implement the libGDX Screen interface.
- BUTTON is a component used for the states as an object for the user to click on. This has been turned into a class due to the number of buttons we have. This would fall under UR_CONTROLS.