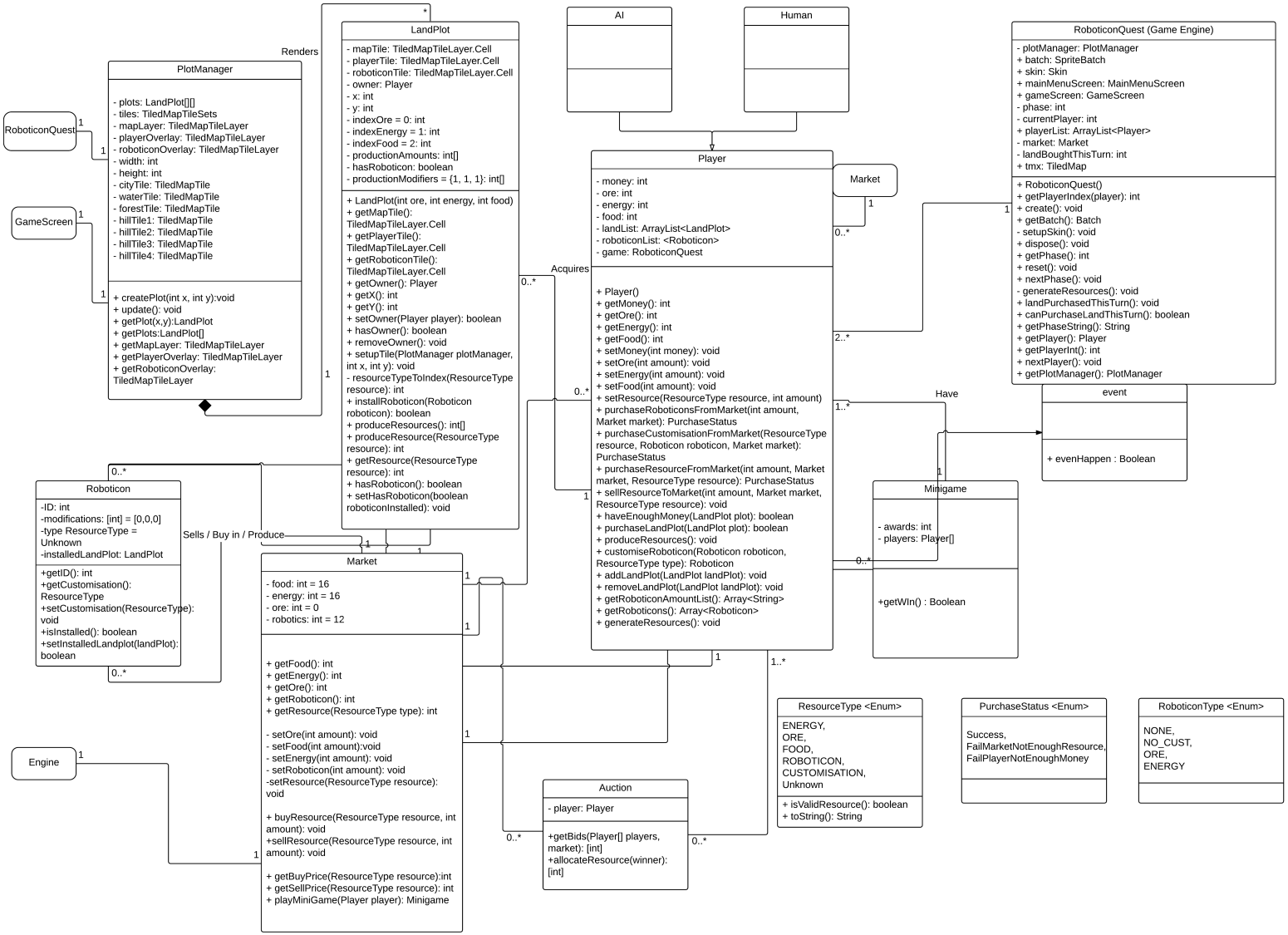
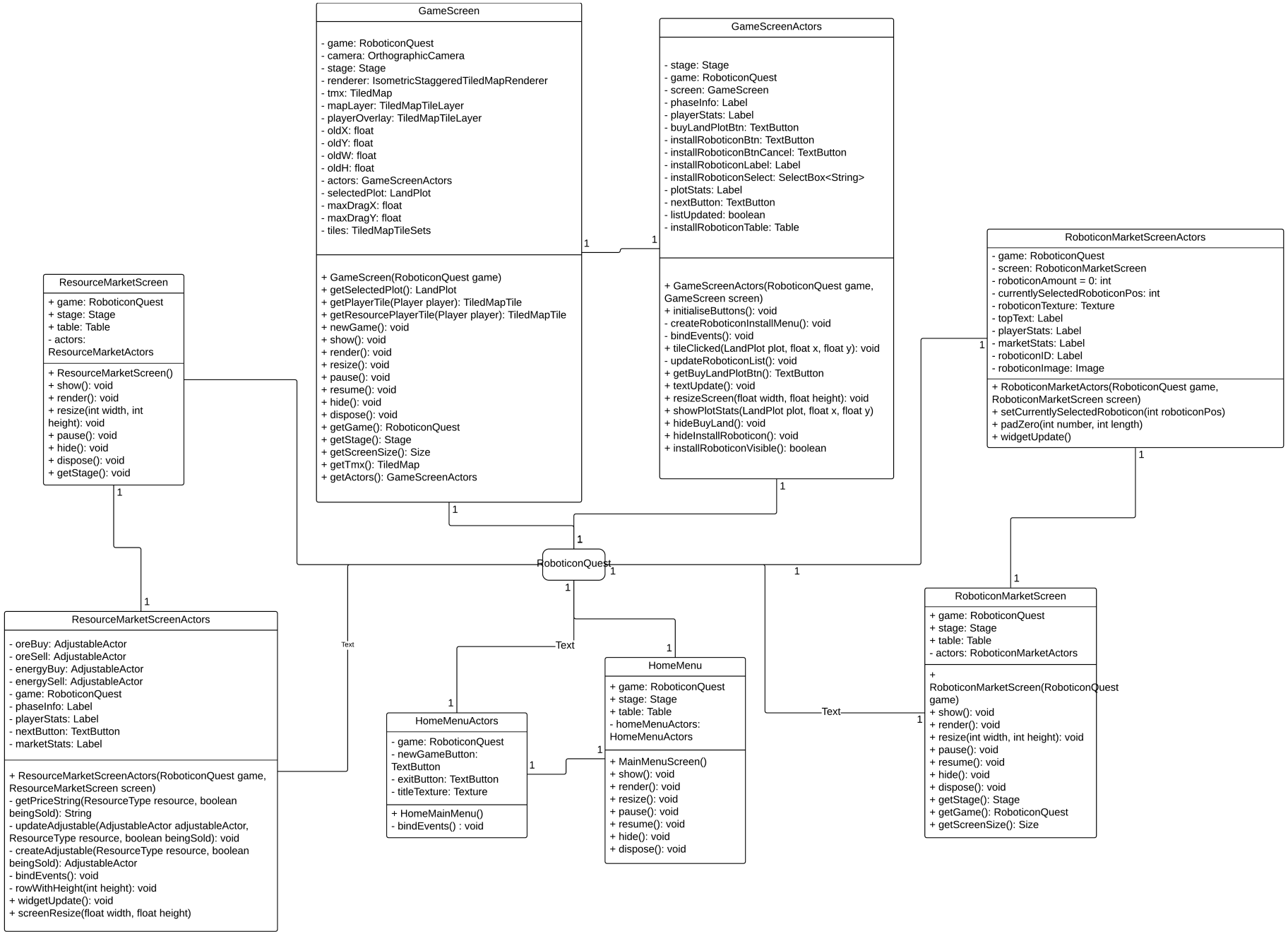
We have created a class diagram,as seen below, in UML 2.X Specification using Lucidchart as it had all of the necessary symbols to represent our concrete architecture. It can also be found at




GameScreen
- game: RoboticonQuest
- camera: OrthographicCamera
- stage: Stage
- renderer: IsometricStaggeredTiledMapRenderer
- tmx: TiledMap
- mapLayer: TiledMapTileLayer
- playerOverlay: TiledMapTileLayer
- oldX: float
- oldY: float
- oldW: float
- oldH: float
- actors: GameScreenActors
- selectedPlot: LandPlot
- maxDragX: float
- maxDragY: float
- tiles: TiledMapTileSets
+ GameScreen(RoboticonQuest game)
+ getSelectedPlot(): LandPlot
+ getPlayerTile(Player player): TiledMapTile
+ getResourcePlayerTile(Player player): TiledMapTile
+ newGame(): void
+ show(): void
+ render(): void
+ resize(): void
+ pause(): void
+ resume(): void
+ hide(): void
+ dispose(): void
+ getGame(): RoboticonQuest
+ getStage(): Stage
+ getScreenSize(): Size
+ getTmx(): TiledMap
+ getActors(): GameScreenActors
GameScreenActors
- stage: Stage
- game: RoboticonQuest
- screen: GameScreen
- phaseInfo: Label
- playerStats: Label
- buyLandPlotBtn: TextButton
- installRoboticonBtn: TextButton
- installRoboticonBtnCancel: TextButton
- installRoboticonLabel: Label
- installRoboticonSelect: SelectBox<String>
- plotStats: Label
- nextButton: TextButton
- listUpdated: boolean
- installRoboticonTable: Table
+ GameScreenActors(RoboticonQuest game, GameScreen screen)
+ initialiseButtons(): void
- createRoboticonInstallMenu(): void
- bindEvents(): void
+ tileClicked(LandPlot plot, float x, float y): void
- updateRoboticonList(): void
+ getBuyLandPlotBtn(): TextButton
+ textUpdate(): void
+ resizeScreen(float width, float height): void
+ showPlotStats(LandPlot plot, float x, float y)
+ hideBuyLand(): void
+ hideInstallRoboticon(): void
+ installRoboticonVisible(): boolean
ResourceMarketScreen
+ game: RoboticonQuest
+ stage: Stage
+ table: Table
- actors: ResourceMarketActors
+ ResourceMarketScreen()
+ show(): void
+ render(): void
+ resize(int width, int height): void
+ pause(): void
+ hide(): void
+ dispose(): void
+ getStage(): void
RoboticonMarketScreenActors
- game: RoboticonQuest
- screen: RoboticonMarketScreen
- roboticonAmount = 0: int
- currentlySelectedRoboticonPos: int
- roboticonTexture: Texture
- topText: Label
- playerStats: Label
- marketStats: Label
- roboticonID: Label
- roboticonImage: Image
+ RoboticonMarketActors(RoboticonQuest game, RoboticonMarketScreen screen)
+ setCurrentlySelectedRoboticon(int roboticonPos)
+ padZero(int number, int length)
+ widgetUpdate()
RoboticonQuest
ResourceMarketScreenActors
- oreBuy: AdjustableActor
- oreSell: AdjustableActor
- energyBuy: AdjustableActor
- energySell: AdjustableActor
- game: RoboticonQuest
- phaseInfo: Label
- playerStats: Label
- nextButton: TextButton
- marketStats: Label
+ ResourceMarketScreenActors(RoboticonQuest game, ResourceMarketScreen screen)
- getPriceString(ResourceType resource, boolean beingSold): String
- updateAdjustable(AdjustableActor adjustableActor, ResourceType resource, boolean beingSold): void
- createAdjustable(ResourceType resource, boolean beingSold): AdjustableActor
- bindEvents(): void
- rowWithHeight(int height): void
+ widgetUpdate(): void
+ screenResize(float width, float height)
HomeMenu
+ game: RoboticonQuest
+ stage: Stage
+ table: Table
- homeMenuActors: HomeMenuActors
+ MainMenuScreen()
+ show(): void
+ render(): void
+ resize(): void
+ pause(): void
+ resume(): void
+ hide(): void
+ dispose(): void
HomeMenuActors
- game: RoboticonQuest
- newGameButton: TextButton
- exitButton: TextButton
- titleTexture: Texture
+ HomeMainMenu()
- bindEvents() : void
RoboticonMarketScreen
+ game: RoboticonQuest
+ stage: Stage
+ table: Table
- actors: RoboticonMarketActors
+ RoboticonMarketScreen(RoboticonQuest game)
+ show(): void
+ render(): void
+ resize(int width, int height): void
+ pause(): void
+ resume(): void
+ hide(): void
+ dispose(): void
+ getStage(): Stage
+ getGame(): RoboticonQuest
+ getScreenSize(): Size
1
Text

Architecture Justification

Engine

The engine class controls the flow of the game, it contains the market and the player, it controls the change of phases (see Requirement 5.1.2) and the change of players (see Requirement 4.1.3). It allows all of the classes to communicate. We chose to have a class like this to stop one class with another responsibility, e.g the plot manager or the market, from having too much responsibility within the game. It allows us to keep classes seperate whilst having a central location to build the game around. In the abstract, we planned to make it so the engine would only be in communication with two classes however we found that it was more useful for the engine to be able to communicate with more classes as it gave everything a central link. It inherits from the Game class implemented by libGDX.

Plot Manager

The plot manager is what initialises every land plot in the game, it does this by reading in the tiled map and assigning each plot resources based on the type of tile. This makes it very easy to customise the Tiled Map whilst changing little to no code. The plot manager also makes it easy to handle things like clicking on individual plots and getting that plots statistics (see Requirement 1.1.2), as you only need the plots x and y coordinates to locate it in the manager. Since the abstract, it has been given the ability to generate land plots rather than just store them as this allows us to easily initialise all land plots in one place. The plot manager can communicate with the engine which then communicates with other classes and the land plots it generated .

Land Plot

The land plot class contains the information about a single plot: who owns it, which roboticon is installed, and it's production rates. The land plot is generated and stored in the plot manager and when it is bought by the player it is also stored in the player, this makes it easy to generate resources across all plots a player owns and only allow them to interact with their own plots without having to check every plot on the map. The land plot can communicate with the player which owns it and the plot manager.

Player

The Player Class will model the user and the simulated player. It will store the resources, land plots and roboticons owned by the player and facilitate the decisions they make. It will communicate with the market to allow the player to buy and sell resources. The player will store a list of land plots which it owns and will trigger production across those land plots in phase 4 (see Requirement 7.1.2). The player Class is the parent to a human player class and an AI class, these will both inherit functionality from the player but will overwrite certain methods (See requirement requirement 4.1.1). Each instance of player will store the food, energy, ore, and money that it owns so it is easy to manipulate for the individual player. The Player will Communicate with the plot manager, multiple instances of land plot, the market and the engine to perform all of its required tasks.

Market

The market handles the production of Roboticons, the purchase/sale of resources from the player. It must be able to communicate with the players, the engine and the roboticons in it's

inventory. It will also communicate with the auction so it can place bids when players are selling resources to other players. The market has it's own internal inventory of resources, money, and roboticons, it generates prices for resources based on the amount of resources it has (see Requirement 8.1.2).

Auction
The auction handles the players selling resources to other players (see Requirement 8.1.1) , it will take resources to be sold from a player and then take bids from other players and the market and give the highest bidder the resources. It therefore must be able to communicate with the players and the market. It will also communicate with the

Roboticon
This class is initialised in the market and can be bought by players (see Requirement 6.1.1), it can also be customised to change the production rates (see Requirement 6.1.4). It can also be installed on a land plot and affect the production rate of resources on that plot (see Requirement 6.1.5). There will be multiple instances with their own modifications and unique identifiers. The roboticon will be able to communicate with players, land plots, and the market. Roboticons have a type which is an enumerable which defines the modification made to the roboticon. The roboticon also stores the production modifiers that it will implement on the land plot it is installed on.

Mini game
The mini game will be accessible from the market and affects the player's money. There will be a chance to bet your money on a game and either win back more or lose it (see Requirement 9.1.1). The mini game will be able to communicate with the player and the market.

Screens
There are multiple screens implemented for different parts of the game (see GUI report), they extend the Screen class from libGDX. The screens allow us to take user input and output a GUI for them to use, making the game much more usable. The Screen contains a stage in which the Actors (stored in a separate class) are positioned and rendered. The screens communicate with the engine to trigger other classes to act. These along with the Screen Actors have been newly introduced since the abstract. This is because they are need for the input and output from the player which was not needed when designing the abstract player

Screen Actors
Screen Actors are the buttons, labels and other widgets that are placed on the screens, having separate classes for them prevents the screen classes from becoming too monolithic and allows us to manage and update them together.

Enumerables
We have enumerables for Resource Type and Purchase Status, these allow us to easily manipulate functions such as buying a particular resource or installing a particular modification, these were not present in the abstract architecture.