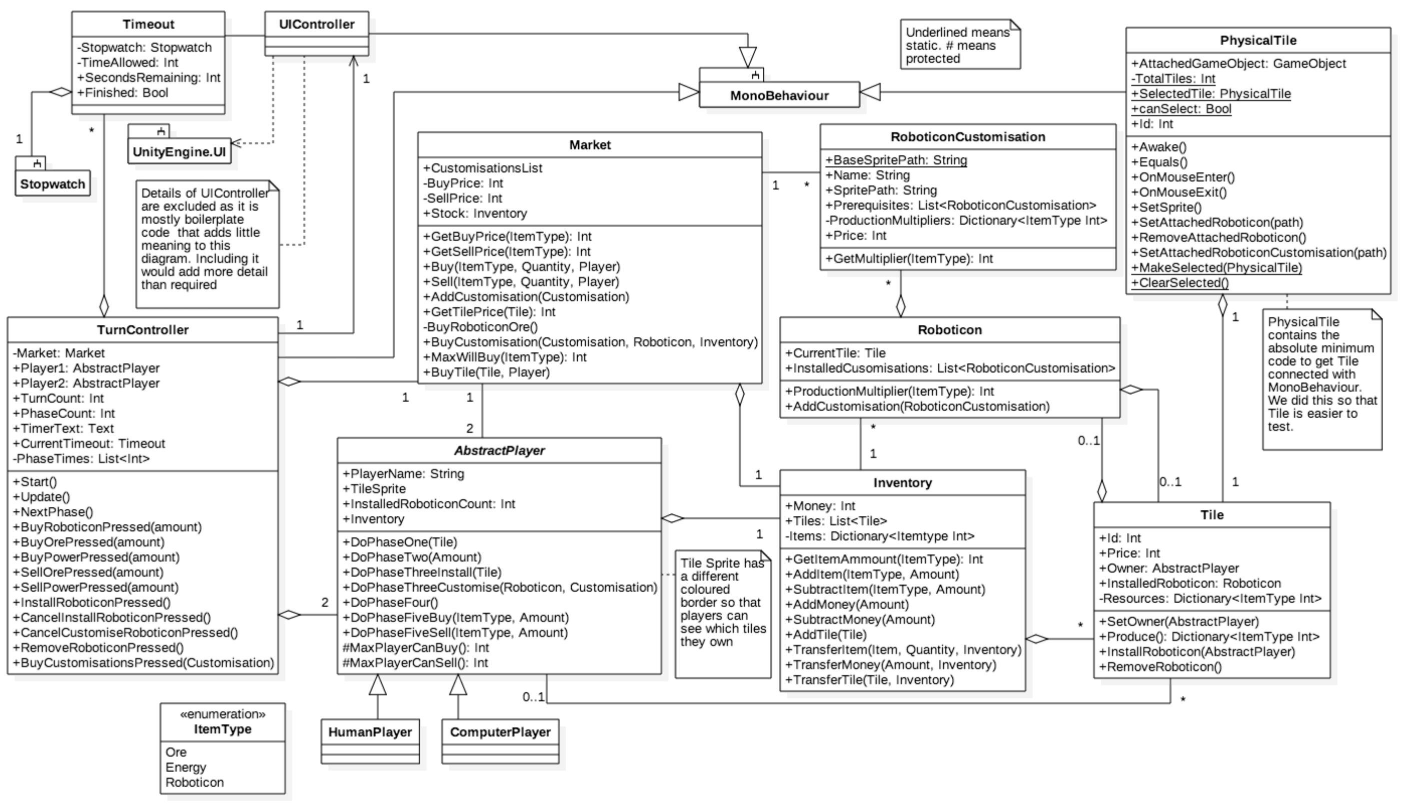
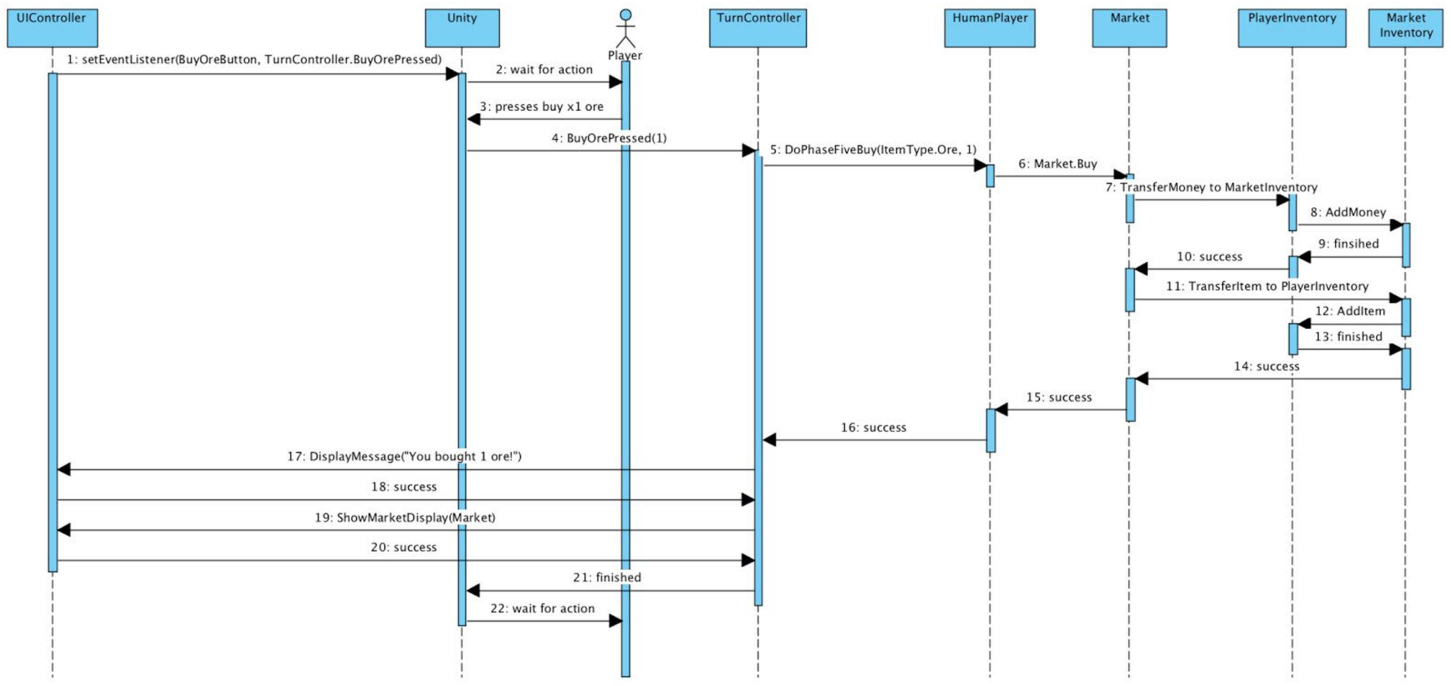
Concrete Architecture (2 Pages)

Target Language/Platform: C# / Unity

UML Class Diagram (link to hi-res version)


Timeout
-Stopwatch: Stopwatch
-TimeAllowed: Int
+SecondsRemaining: Int
+Finished: Bool
UIController
UnityEngine.UI
Stopwatch
Details of UIController are excluded as it is mostly boilerplate code that adds little meaning to this diagram. Including it would add more detail than required
MonoBehaviour
Underlined means static. # means protected
PhysicalTile
+AttachedGameObject: GameObject
-TotalTiles: Int
+SelectedTile: PhysicalTile
+canSelect: Bool
+Id: Int
+Awake()
+Equals()
+OnMouseEnter()
+OnMouseExit()
+SetSprite()
+SetAttachedRoboticon(path)
+RemoveAttachedRoboticon()
+SetAttachedRoboticonCustomisation(path)
+MakeSelected(PhysicalTile)
+ClearSelected()
Market
+CustomisationsList
-BuyPrice: Int
-SellPrice: Int
+Stock: Inventory
+GetBuyPrice(ItemType): Int
+GetSellPrice(ItemType): Int
+Buy(ItemType, Quantity, Player)
+Sell(ItemType, Quantity, Player)
+AddCustomisation(Customisation)
+GetTilePrice(Tile): Int
-BuyRoboticonOre()
+BuyCustomisation(Customisation, Roboticon, Inventory)
+MaxWillBuy(ItemType): Int
+BuyTile(Tile, Player)
RoboticonCustomisation
+BaseSpritePath: String
+Name: String
+SpritePath: String
+Prerequisites: List<RoboticonCustomisation>
-ProductionMultipliers: Dictionary<ItemType Int>
+Price: Int
+GetMultiplier(ItemType): Int
Roboticon
+CurrentTile: Tile
+InstalledCusomisations: List<RoboticonCustomisation>
+ProductionMultiplier(ItemType): Int
+AddCustomisation(RoboticonCustomisation)
PhysicalTile contains the absolute minimum code to get Tile connected with MonoBehaviour. We did this so that Tile is easier to test.
TurnController
-Market: Market
+Player1: AbstractPlayer
+Player2: AbstractPlayer
+TurnCount: Int
+PhaseCount: Int
+TimerText: Text
+CurrentTimeout: Timeout
-PhaseTimes: List<Int>
+Start()
+Update()
+NextPhase()
+BuyRoboticonPressed(amount)
+BuyOrePressed(amount)
+BuyPowerPressed(amount)
+SellOrePressed(amount)
+SellPowerPressed(amount)
+InstallRoboticonPressed()
+CancelInstallRoboticonPressed()
+CancelCustomiseRoboticonPressed()
+RemoveRoboticonPressed()
+BuyCustomisationsPressed(Customisation)
AbstractPlayer
+PlayerName: String
+TileSprite
+InstalledRoboticonCount: Int
+Inventory
+DoPhaseOne(Tile)
+DoPhaseTwo(Amount)
+DoPhaseThreeInstall(Tile)
+DoPhaseThreeCustomise(Roboticon, Customisation)
+DoPhaseFour()
+DoPhaseFiveBuy(ItemType, Amount)
+DoPhaseFiveSell(ItemType, Amount)
#MaxPlayerCanBuy(): Int
#MaxPlayerCanSell(): Int
Tile Sprite has a different coloured border so that players can see which tiles they own
Inventory
+Money: Int
+Tiles: List<Tile>
-Items: Dictionary<Itemtype Int>
+GetItemAmmount(ItemType): Int
+AddItem(ItemType, Amount)
+SubtractItem(ItemType, Amount)
+AddMoney(Amount)
+SubtractMoney(Amount)
+AddTile(Tile)
+TransferItem(Item, Quantity, Inventory)
+TransferMoney(Amount, Inventory)
+TransferTile(Tile, Inventory)
Tile
+Id: Int
+Price: Int
+Owner: AbstractPlayer
+InstalledRoboticon: Roboticon
-Resources: Dictionary<ItemType Int>
+SetOwner(AbstractPlayer)
+Produce(): Dictionary<ItemType Int>
+InstallRoboticon(AbstractPlayer)
+RemoveRoboticon()
«enumeration» ItemType
Ore
Energy
Roboticon
HumanPlayer
ComputerPlayer
1
*
1
1
*
1
1
1
2
2
1
*
1
0..1
0..1
1
1
1
*
0..1
*

UML Sequence Diagram (link to hi-res version)



Explanation of UML Class Diagram

The UML class diagram shows our concrete architecture. Subsystems are used to show dependencies on external libraries. Class members are prepended with + if they are publicly visible, - if they are private, # if they are protected (private, but inheriting classes can access). Note that if something is publicly visible it may still be read only. Static members are underlined. Directional associations are used to describe relationships where the ‘receiver’ doesn't need to know of the existence of what is associated with it. UIController is left blank as it would add little to no meaning to the diagram and add unneeded complexity to the diagram making it harder to understand rendering it less useful as a software engineering tool.

Explanation of UML Sequence Diagram

The above UML Sequence diagram shows a possible set of interactions for a player buying 1 Ore from the market. First the UIController sets up a button listener with the Unity engine. We wait for the player to take an action. At this point it's possible the user chooses another action, but for the sake of this diagram they choose to buy 1 ore. There are several calls that eventually reach Market.Buy. Here you can see the advantages of the Inventory class and how they let Market control transactions easily. Market first attempts to withdraw the cost of the item from the player's inventory, only if this is successful does it give the player’s inventory the item. Then it attempts to transfer the item from its stock to the player's Inventory. If either of these stages fail the Market can restore the state before the transaction. Eventually we return to the turn controller which uses the UIController to update the display for the user, we finish by waiting for another action from the user.

The interactions could be different in many stages if the user has enough money or if the market has enough stock and for a variety of other different states.

Unity, TurnController and UIController are all executing at the same time as the other code as they are either Monobehaviours or the game engine itself.

Tools Used

We continued to use StarUML[1] and VisualParadigm[2] for the class and sequence diagrams because they were useful tools when creating the conceptual architecture. Furthermore having used them before we would not have to spend time learning new tools, we also knew the capabilities of the software and that these meet our requirements for their use again.

StarUML[1] is available on all platforms so everyone in the team could use it easily, and has error checking for things such as duplicate associations. StarUML[1] was however frustrating to use when creating sequence diagrams so we used VisualParadigm[2] for this again as it was acceptable last time.

Justification of Architecture (2 Pages)

Link to previous UML Class Diagram for comparison

Our conceptual architecture was too concrete, as we implemented the game we largely followed the original architecture but obviously some changes had to be made when we came across problems in the original architecture.

Although there was no requirement (see updated requirements #8) to implement food for this deliverable we knew that it would likely be implemented in the future so our architecture is designed to make this very easy. To add this feature developers will have to do little more than add it to the ItemType enumerable.

We have introduced PhysicalTile as a bridge between Tile and MonoBehaviour. This allows us to write cleaner, simpler tests for Tile. Conceptually you can still think of them as a single piece of behaviour, as you only interact with TIle.

Many classes have not yet been implemented as they were not yet required, these have been left out of the diagram.

The inventory class was introduced as we wanted a separation of concerns between the actions players do and the handling of the items in their possession. This also makes it easier to enforce that transactions happen successfully and makes testing them easier. As you can see in the UML Sequence Diagram the market class can handle the entire transaction; adding/removing an item to/from the inventory and adding/subtracting money from the inventory. This is better than trusting a player class to remove money from their Inventory after buying something.

We introduced the timeout class as a more elegant way of handling the required (see updated requirements #42) phase durations. Previously we were planning on having the TurnController open or close the Market. Now we use a timeout to separate this logic from Market and other classes so it can be handled entirely by the TurnController and UIController.

We added the RoboticonCustomisation class to make Roboticon more cohesive and to encapsulate the logic behind customisations. In our conceptual architecture we hadn't considered how much information there is to do with RoboticonCustomisations. Separating it into its own class also makes customisations more extensible in the future, we already have support for RoboticonCustomisations to have prerequisite RoboticonCustomisations, positive and negative effects on resource production and for multiple RoboticonCustomisations on a Roboticon.

In our conceptual architecture AbstractPlayer had a similar design with separate methods for each phase. We have expanded on that by separating phases where possible, for example DoPhaseThree has been split into DoPhaseThreeCustomise and DoPhaseThreeInstall to separate installing and customising Roboticons. This also makes it easier to handle user input.

All associations to do with players now are with AbstractPlayer and not HumanPlayer or ComputerPlayer themselves. This is because we realised that we would have a lot of duplicated code if we had classes implement an interface for HumanPlayer and for ComputerPlayer. Because of this it will also be extremely simple to create ComputerPlayer in the future. Once the class has been created the only changes required will be to allow the user to choose whether they want to play against another human or against the computer.

[1] StarUML. '"StarUML 2. A sophisticated software modeler." staruml.io. [Online]. Available: http://staruml.io/ [Accessed Jan. 23 2017]

[2] VisualParadigm. "Visual Paradigm Software Design Tool." visual-paradigm.com [Online]. Available: https://www.visual-paradigm.com/ [Accessed Jan. 23 2017]