Architecture Report

Link to original UML: UML1
Link to updated UML: UML2

UML 2.X was used as it is the industry standard in terms of creating diagrams. This allows for ease of understanding for anybody reading the diagram given it is very commonly used, and there are standard guidelines as to how draw a model. UML was also used because it is an easy visual representation of what is important, classes and their relationships in object oriented programming. Additionally, StarUML was used to model the UML 2.X diagrams which was chosen due to its ease of use.

As the diagram is detailed and too large to appropriately display in this document, please see the link above for concrete architecture. The structure of the diagram can be split into two parts, the logic/data structure (the bottom section) and the graphical structure (the top section). The bottom section mostly consists of our original architecture submitted for Assessment 1, with language relating to lists changed to more closely match UML.

The top section is composed of a set of visual screens which extend the LibGDX class ScreenAdapter (which we could have shown in the diagram through the use of a generalization arrow to an empty ScreenAdapter, but ultimately chose not to as we felt it could cause confusion in an already rather line heavy section of the architecture) as well as AllHandsOnDeckGame which extends the LibGDX class Game. There are some LibGDX specific language used within the individual classes such as Texture, Label, Stage and TextButton. Because of the fact that we use LibGDX to develop our game, we felt it was appropriate to include language directly associated with the engine in our UML diagram.

There are many methods and attributes within classes that we chose not to include in our diagram in order to cut down on complexity and make it simpler and easier to read, but we felt it would be best to include a brief mention of what is missing in general here in case anyone is confused about the implementation. Getters and setters in general are not included, with the exception of a few of note we felt would be specifically relevant. This is because it is a fairly standard practice to include in classes and including them specifically in the diagram would be obsolete unless they are unique or non-standard in some way. Within the visual section of the diagram there are a multitude of possible methods which were ultimately included in the implementation but we felt were unnecessary to include in the diagram, such as render(), dipose() and any method which creates buttons or labels. Whilst button/label creators and dipose() are both fairly simple methods initializing labels and buttons and clearing screen of its stage respectively, render() is of note as it can have a different function dependent on screen. In general render() simply clears the screen before using the stage to draw the various graphical elements, and it does so for every ScreenAdapter in the architecture. However, specifically in GameVisual, we have also used it to accomplish functionality which occurs on screen change. This is because render is always called when the game switches to a specific screen, and allows for things like win evaluation to be done immediately after encounters or battles. Whilst we felt as if this could warrant including it in the diagram, ultimately we didn't due to the general simplicity of how

render() functions normally, and our use of it in the implementation could have been done in other ways.

Also generally not included are various styles related to labels and buttons, which dictate how a button or label looks. Whilst used in the implementation, we felt as if it was structurally simply a part of button and wasn't significant enough to include in the diagram except in situations where there are multiple to accomplish some function (specifically in GameVisual).

Architecture Breakdown: ArchJustification

Any relation to requirements can be seen as a reference in the format {A1.1,B2.2,..} where A1.1 and B2.2 are direct references to our requirement documentation.

A full list of the changes to the architecture in a truncated format can be seen in the document Architecture Breakdown.

The changes to our architecture can be split into two parts, the pruning of our original abstract architecture of unnecessary features for this stage of development for Assessment 2, and the addition of the visual section of our architecture necessary for the actual implementation of the game.

Whilst our original architecture submitted for Assessment 1 was meant to be abstract, it was in actuality fairly concrete to begin with. Whilst we tried to avoid going implementation specific (and as such avoided the visual aspects of the game completely), our architecture for the actual functionality of the game and how it was going to be played was in depth and relatively specific. This means not many changes to this section of the architecture was needed in order to realistically implement the game from the architecture. What was changed was the removal of classes relating to features that we were not going to implement for development in Assessment 2, such as the main menu for save/load game functionality or minigame. We removed these from our architecture so that during development we wouldn't get confused or accidentally implement non-essential features into the game, but our original architecture still exists and so any future re-addition of these aspects of the architecture should be fairly simple.

In regards to the relation to our requirements, this section of the architecture covers most of the logical and data functionality of the code. GameLogic covers a lot of the functionality of the game, as it checks whether the game is over {B1.5,B1.8}, and if so whether the game is lost or whether it is won or lost {A2.4,A2.2}; manages the resources of the player such as their gold and supplies {A1.3,B1.3,B2.1}; manages the overall game objectives, which can tweaked to change how long a game may last {A2.2,A3.1,A4.2,B2.2}, managing and storing the node map which constitutes the sailing mode {A2.8} and is essentially the root class from which all other classes in the section of the architecture are spawned (hence the composition in the diagram).

From GameLogic you go to node, which spawns encounters and can hold colleges/departments which are spawned on node map generation{B1.4}. The colleges/departments are what allow for the theming around the UoY {2.9}, but functionally serve an important purpose in the overall objective of the game and the ability to spend the resources you earn to upgrade your ship or your deck (without, of course, spending what you don't have {A2.1,A2.5,B1.6,B3.1}. Attacking colleges/departments is a key part of the overall objective of the game {A2.2,A3.1} and spawns a new instance of a battle specific to the college/department {B1.2}

Non college/department nodes spawn encounters {A1.2} , a key aspect of the sailing mode of our game. In encounters you are presented with a choice, and choices can affect your resources or on their own trigger a battle with an enemy ship {B1.2}

BattleMode manages the second mode of the game {A2.8}, and is how you actually engage with enemy ships (including ones spawned by colleges and departments {A2.5}. This class holds all information relevant to the system of battle we developed, such as the hands and decks of the opponents, the stats of the fighting ships and the possible reward for the player on defeating the opponent {A1.1}

The second portion of the architecture relates to how we display the game information to the player as a GUI. This allows the game to be more intuitive and easy to use than if we had gone for a much more simple to code command line interface {A4.3,B4.1,B4.2}. Our graphics were created with the idea in mind that this game could be used in presentations for the university {B4.3}.

The AllHandsOnDeck game class is used to manage the switching between relevant screens, such as from an encounter to a battle {B1.2}. This LibGDX class includes functionality to switchScreen, which we then call within screens themselves. Because we need to call this function within the screen, AllHandsOnDeck contains a copy of a pointer to itself, which is then used throughout all the screens to allow for switching between them.

The most important screen, not unlike GameLogic, is GameVisual. GameVisual contains a GameLogic which is necessary to run the game, but is also lost when GameVisual is destroyed due to our current lack of save/load functionality (hence the composition between the two). GameVisual is considered the root visual screen, and all visual screens return to GameVisual when they are done with. This is shown through the use of parent throughout. This is especially relevant as in the instance of BattleModeVisual, GameLogic doesn't actually create an instance of it itself (this is done in encounter, college and departments as shown by the diagram) but it is done with the parent of BattleModeVisual set to GameLogic so it switches back to the root visual when it is done. GameVisual itself is used to display the resources available to the player, the nodeMap (including the currentNode and neighborNodes to make it intuitive for users to know where they can sail to through the use of different TextButtonStyles {A4.3,B4.2}), as well as a button to access ShipVisual to display ship stats and objectives {A2.4}. It also includes some of the turn change functionality that would have previously been included in GameLogic but has moved to the screen as it requires the screen to change.

Other visual screens simply display relevant information for their respective function such as the GameEndVisual showing whether you won or lost and your score, but some also include some logic functionality which would have been kept in the other part of the architecture but couldn't due to it requiring a screen change, such as EncounterVisuals interpret effect. These methods are included in the diagram to reflect that.