Final Architecture and Traceability Report

This report outlines the final architecture for the system and explains how the requirement changes for Assessment 4 were accommodated. To do this, the previous concrete class diagram was expanded using UML 2 and the tool draw.io. This tool was used because it is a web-based application, meaning that the diagram could be developed using multiple devices. Also, the same tool was used for the creation of the first concrete architecture, which means that the style remains consistent between versions.
The original concrete architecture only covered the preliminary implementation, so to improve traceability, the Assessment 2 and 3 additions, which were omitted from the original architecture diagram, have been highlighted in green. Also, the Assessment 4 additions have been highlighted in blue to make the new classes and methods stand out. Furthermore, the accessor and mutator methods have not been included in Figure 1 to simplify the diagram and ensure the focus remains on the important features.

Full-Sized Final Architecture: as2378.github.io/unlucky/files/Assessment4/Final_Architecture_Diagram.jpg
Abstract Architecture: as2378.github.io/unlucky/files/Assessment4/Arch1-Abstract_Architecture.pdf
Assessment2 Architecture: as2378.github.io/unlucky/files/Assessment4/Arch2-Original_Concrete_Architecture.pdf
Updated Requirements: as2378.github.io/unlucky/files/Assessment4/Updated_Requirements.pdf

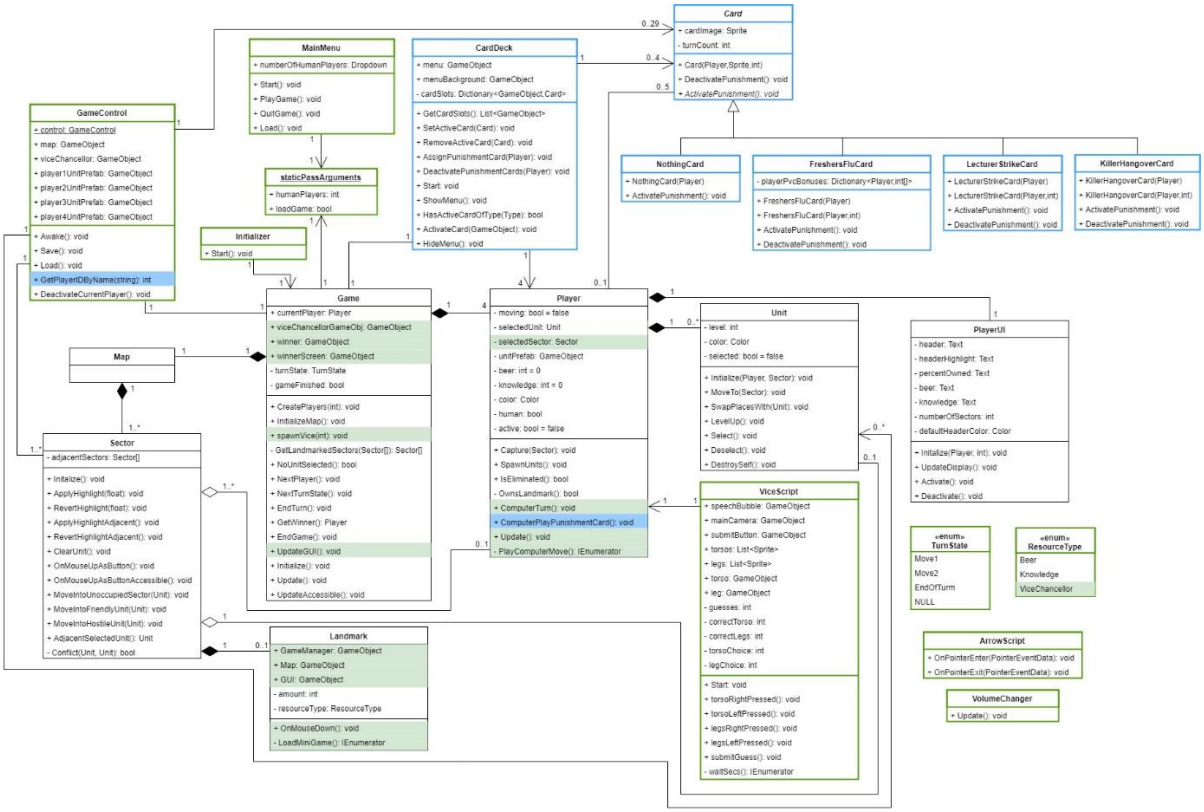

Figure 1. A UML class diagram presenting the final architecture of the system.

Game:

The **Game** class is primarily used for initialising games, progressing turns and detecting wins, but also plays an important role in linking all the major classes together. The game initialization is used to ensure that requirement F1, which states any player not controlled by a human must be controlled by a computer, is satisfied. Also, **Game** includes a 1-to-4 association between itself and **Player**, which has been included because requirement N3 declares that the game must include exactly four players.

Excluding new additions, this class follows a similar structure to the abstract architecture, with only slight changes to the type of associations **Game** has with **Map** and **Player**. The class also closely adheres to the previous concrete architecture; however, several new relations, methods and attributes have been added because the original concrete architecture did not cover the assessment 3 features.

Player:

Overall, the **Player** class is used to represent the players, both human and computer, that are playing the game. The class keeps track of important information about each player, such as the **Units**, **Sectors** and bonuses they own. In terms of computer-controlled players, **Player** includes the methods *ComputerTurn*, *PlayComputerMove* and *ComputerPlayPunishmentCard*, which simulate the movement of a player's units and activation of punishment cards, which relate to requirements F1, F5 and F13.

This class is also visible in the abstract architecture, however, it no longer consists of two subclasses, Non-Human Player and Human Player, because the difference between these is minimal and so have been combined into the single **Player** class.

PlayerUI:

Each **Player** in the game is composed of an instance of a **PlayerUI** class**.** This class is used purely for aesthetic purposes, displaying a **Player**'s percentage of **Sector**s owned and beer/knowledge bonuses. **PlayerUI** was created during the preliminary implementation phase and helps improve the playability of the game by providing users with useful information.

Unit:

Unit is used to control the gang members within the **Game** and contributes towards requirements N8, N10, F5, F6 and F10. The class' methods and associations shown in the abstract and concrete architectures have not changed, however, an association has been added from **GameControl** because the loading has to create new instances of **Unit.**

Map:

The purpose of the **Map** is to provide a link between all the **Sector**s in the game to the **Game** class and so does not contain any methods. The class helps towards satisfying requirement N5, which states "The game map must be divided into sectors." Also, **Map** has not changed since the preliminary concrete architecture and is also visible in the abstract version.

Landmark:

The **Landmark** class is used to represent the special landmark sectors by storing information about the type of landmark and how much resources it provides when captured. Within this system, the PVC is represented as a landmark because the PVC does not move from the **Sector** it is spawned in. Overall, the functionality that **Landmark** offers is important for effectively distinguishing landmarks from standard sectors, which is key for satisfying requirement N6.

In the abstract architecture, this landmark/resource subsystem is represented as three separate classes: Landmark, Resource, Beer and Knowledge. However, in the current system, this has been replaced by the **Landmark** class and an association within the class to an enumerator, **ResourceType.** Also, using instances of **Landmark** as a method of representing the PVC maintains the associations between **Map, Sector** and **PVC** shown in the abstract architecture and satisfies requirement F2.

Sector:

Sector is used to fulfil requirements N5, N8, N9, F3 and F5, by representing sectors of the map and providing methods that move **Unit**s, resolve conflicts (*Conflict)*, and switch ownership of sectors. Also, the class has not been changed since the first concrete architecture diagram and the only major difference from the abstract architecture is the removal of the association with a **PVC** class, as explained above.

ViceScript:

The **ViceScript** class is used to control the PVC minigame (requirement F3). It contains attributes that record a player's guess and the correct answer, as well as methods that change these guesses and check to see if the player has guessed correctly. This has been given a directional association from one **ViceScript** to one **Player** because when the minigame finishes, the **Players** must receive a bonus, as stated in requirement F4.

GameControl:

This class is used to control the saving and loading feature, fulfilling requirement F7. To do this, **GameControl** must access information about **Player**s, **Sectors**, **Units** and **Cards,** and so an association with **Game** has been included because through **Game** the class can easily access the others.

MainMenu:

The purpose of the **MainMenu** class is to provide methods for the game's main menu. This is an important feature because it allows users to load saved games (F7) and allow them to choose the number of human **Player**s they want (N3, F1).

Assessment 4 Changes:

Overall, to accommodate punishment cards six additional classes have been included in the architecture: **CardDeck, Card, NothingCard**, **FreshersFluCard**, **LecturerStrikeCard**, and **KillerHangoverCard**. This was done to separate the punishment card system from the old system, minimising the amount of interference and integration problems it may cause.

CardDeck:

The **CardDeck** class has two main roles and the first of these is to control the functionality of the punishment card menu, which is achieved though the methods *ShowMenu, ActivateCard* and *HideMenu.* The second role is to provide an interface for the card system by providing methods that; assign **Card**s to players, *AssignPunishmentCard* (F12); activate cards, *ActivateCard,* manage active cards and deactivate cards.

The class' *menu* and *menuBackground* attributes are used for showing and hiding the card deck GUI, and the *cardSlots* dictionary links a **Player**'s **Card**s with the card buttons in the menu. This was done because it is a very flexible approach to the problem as the buttons only have to call *ActiveCard* with themselves as parameters and the card linked to the button in the dictionary is activated (F13).

The class has a 1-to-1 association with **Game** because **CardDeck** needs to access the currentPlayer and **Game** needs to tell the **CardDeck** when to assign and deactivate punishment cards. Also, **CardDeck** has a

1-to-4 directional association from itself to **Player** because the **CardDeck** is responsible for giving and removing punishment cards from **Player**. Additionally, the class has a 1-to-0..4 association from **CardDeck** to **Card** because the class records the active cards and must deactivate them when required.

Card:

This class has been added to the architecture to represent the punishment cards and their functionality. The class itself is abstract and has four child classes: **NothingCard**, **FreshersFluCard**, **LecturerStrikeCard**, and **KillerHangoverCard**, which are used to define the behaviour of the different types of punishment cards and must implement the abstract method *ActivatePunishment*. Additionally, **Card** contains two attributes, *cardImage*, which stores a visual representation of the card, and *turnCount*, which is the number of turns the card will be active for and is used to deactivate the cards within *DeactivatePunishmentCards*.

The card architecture was designed this way because it improves flexibility and maintainability by allowing the cards to be independent of each other. Also, **Card** is used as a way of standardising the different types of punishment card, meaning that they can be perceived as a single type. Additionally, **Card** and its subclasses meet requirement F11 and contribute towards requirements F12 and F13.

NothingCard:

This class is a type of punishment card and when activated nothing happens. This was added to balance the punishment card system because the player's may have to play this card to gain better ones. Also, **NothingCard** only contains two methods, it's constructor and the *ActivatePunishment* routine, and works towards satisfying requirement F11.

FreshersFluCard:

The **FreshersFluCard** class represents the fresher's flu card and temporarily removes the bonuses from enemy players as a punishment, satisfying requirements N11 and F11. The class contains a dictionary, *playerPvcBonuses,* as an attribute and is used to remember the bonuses gained from the PVC minigame. This is because when the card is active, the ownership of landmarks can change and so *DeactivatePunishment* recalculates the landmark bonuses and then adds the recorded PVC bonus.

FreshersFluCard has two constructors, one with an integer argument and one without. This integer represents the turnCount value and is used by GameControl when loading a game (F8).

LecturerStrikeCard:

This class is a type of punishment card and when activated, the card prevents enemy players from moving more than once per turn. The addition of the class is used to meet requirement F11, F12 and F13, and the punishment's effect is used to meet requirement N11.

KillerHangoverCard:

The **KillerHangoverCard** is also another type of punishment card and works in a similar way to the others, however, it's effect causes the opposing players to miss their turn.

The second major assessment 4 change is the inclusion of two types of unit: undergraduates and postgraduates (F10). For this addition, there were no architectural changes that needed to be done because the team chose a design that only changes the content of existing methods. To represent the types of unit, it was decided that the simplest approach would be to use the Unit's *level* attribute, where levels 1-4 represent undergraduates and level 5 (the highest level) represents postgraduates. Also, the postgrad's special ability is to move to any player-owned sector, satisfying requirement F10, and the current movement system can be easily updated to accommodate for this.

Changes made to the inherited design:

Overall, there has been a lot of changes made to the architecture diagram, with the additions of both assessment 3 and 4 classes, however, very few changes were made to the actual design of the system. This is because the Assessment 4 additions are designed so that there is minimal interference between the new features and the old ones. Also, limiting the number of changes ensures that traceability from the final architecture to the preliminary concrete architecture is maximised. Additionally, the development team are very familiar with the inherited design from working on the same base project during Assessment 3.

The majority of changes made to the existing architecture are located within the **Player** class. For example, the computer player's functionality had to be extended to be able to activate punishment cards, as noted by requirement F13, resulting in the addition of the *ComputerPlayPunishmentCard* method within **Player**. Additionally, **Player** now contains an association with **Card** because the **Player**s will store the punishment cards they own. The class also has a directional association from **CardDeck** because the **CardDeck** needs access to the **Player**'s cards to add new **Card**s, remove existing **Card**s and display the cards in the menu.

Furthermore, the **GameControl** architecture has also been updated because the saving and loading needed to be extended to accommodate for punishment cards (F8) and postgraduate units (F9). Therefore, a relation from **GameControl** to **Card** was created and the multiplicity is 0..29 because a **Player** can own up to 5 cards and at most 4 cards can be active at once.