Final Architecture and Traceability report

1.1 Introduction

The architecture of our system is relatively similar to the architecture inherited from Team Watson's Assessment 3 project [1], changed only to reflect the new requirements for multiplayer gaming (user requirement 18) and having a locked room (user requirement 20), from our updated requirement table [2].

1.2 Architecture

In our final implementation the game begins with a character selection screen as part of system requirement 4a (with the scripts *CharacterSelector* and *GameMaster* attached to the scene within Unity). When instantiated, the *GameMaster* object creates all possible *NonPlayerCharacter, Clue* and *VerbalClue* objects, and then when the user selects their detective(s) *GameMaster.CreateNewGame(...)* is called to generate a new *Scenario* object (for system requirements 22 & 23) and either one or two *PlayerCharacter* objects are are created depending on the selected gamemode. The rest of the game revolves around Unity Scenes with several attached scripts handling user input and the *GameMaster* object which holds all relevant objects for gameplay including the *PlayerCharacter* and *Clue* objects.

1.2.1 UML Diagram

We found the previous team's UML class diagram representation of the system to be useful so we updated it for the final architecture specification (changes highlighted in red). It is available on our website http://wedunnit.me/webfiles/ass4/ClassDiagram.jpg

1.2.2 New class & method description/reasoning

The following table contains descriptions of some of the newly added methods, classes and properties.

Class/ Method/ Property	**Purpose**
+GameMaster.playerCharacters :PlayerCharacter[]	An array containing the detective characters used in the game, necessary to have multiple players for system requirement 18a.
-GameMaster.lockedRoomIndex:Int	Contains the build index of the locked room for system requirement 20a.
-GameMaster.playerHasPassedRiddle:Bool[]	An array that states whether each player has passed the riddle yet for system requirement 20a.
+isMultiplayer:Bool	False if the game is a single player game. True otherwise. As described in section 1.3.1 this is used by several methods which depend on the number of players.
-GameMaster.currentPlayerIndex:Int	The index of the active player.
+GameMaster.SwitchPlayers()	Swaps players after a certain number of turns to give another player a turn, following system requirement 18d.
+GameMaster.GetPlayerCharacter()	Returns the *PlayerCharacter* object of the player whose turn it is, used for displaying a character when a scene is loaded.
+GameMaster.UseTurn()	Deducts an action from the current player as part of system requirement 18d.

+GameMaster.SaveCurrentPlayerRoom(level)	Saves the room the current player is in when their turn ends, so they are in the correct room when it is their turn again.
Puzzle (class)	Gets and stores a random puzzle (Puzzle.riddleText), the correct answer (Puzzle.correctAnswer) and all incorrect answers (puzzle.wrongAnswers), required to support user requirement 20 and requirement 20a.
Puzzle.LoadJSON()	Loads a random riddle from a JSON file.
PuzzleScript (class)	Unity Scene controller script for the puzzle room. Initially populates fields with data from a new Puzzle object, then handles the user input in accordance with system requirement 20a.
PuzzleScript.puzzle	Stores a Puzzle object instantiated from PuzzleScript.Start() when the Scene is loaded.
PuzzleScript.AssignAnswers()	Uses PuzzleScript.puzzle to populate the answer buttons.
PuzzleScript.IsCorrect(int index)	Called when any answer button is clicked, and depending on the index of the button pressed the room will either be unlocked or the previous scene will be loaded.
GameOver (class)	Unity Scene controller script for the endgame screen. Congratulates the winner of the game (for requirement 19) and displays a name entry textbox for each player before saving score(s) to the leaderboard. Also prepares the game for another playthrough for requirement system 21.
GameOver.P1Start() [if singleplayer] GameOver.P2Start() [if multiplayer]	Called depending on the number of players by GameOver.Start() when the scene is loaded, these methods turn on the relevant HUD features and destroy *GloablScripts* and *NotebookCanvas* objects so another game can begin from fresh.
GameOver.P1CloseScreen() [solo only] GameOver.P2CloseScreen() [multi. only]	Called when the ‘confirm name’ button is pressed to save the available scores/names to the leaderboard file and return the player(s) to the menu.
LevelManager.DisplayCharacterChange()	Called by GameMaster.SwitchPlayers() if a multiplayer game is being played to alert the players that it is time for them to swap, in accordance with system requirement 18d.

1.3 New Requirements

1.3.1 Multiplayer Gaming

To implement multiplayer gaming for changed requirement 18a, we first thought about how the finished product should work. For example, we asked ourselves the following questions: should players compete or cooperate? Should a user’s turn be time or action based? Should one player have to wait for another to complete a full game before they play, or should all players share one game session? Should each player have have their own notebooks and scenarios?

Our solution needed to simply extend single player mode to work with multiple players, rather than duplicating the game and adapting the second version to work with more players. The reason for this is

that the latter method would make maintaining the system inefficient: if one change is needed it would have to be propagated to both singleplayer and multiplayer modes independently. Overall we achieved this well, we created a new Unity scene to handle multiplayer detective selection rather than extending the existing character selection scene. This could be refactored in the future.
Another consideration we made was extending single player mode to cope with *n* players rather than just two as part of system requirement 18a, however after deliberation we admitted that the extra complexity would be unnecessary for the requirements we had to fulfil, and the time would be better spent fulfilling other requirements.

We extended our code so that most methods always treat the game as having multiple players, and the methods that explicitly deal with handling character switching consult the new boolean variable *GameMaster.isMultiplayer.* This can be seen in functions like *ItemScript.OnMouseDown()* which refers to *GameMaster.UseTurn()* and *GameMaster.SwitchPlayers()*, two methods clearly for multiplayer mode, but compatible with solo play as both methods consult the class variable *GameMaster.isMultiplayer* and handle their response appropriately as described below. Another example of this is how scoring is tracked in methods including *GameMaster.Update(), GameMaster.ClueCollected() and GameMaster.GivePoints(Float:points)*. The current score for all players (for user requirement 13) is stored in *GameMaster.scoreArray,* and is accessed by the index *GameMaster.currentPlayerIndex.* These methods now automatically work for both single and multiplayer games as *currentPlayerIndex* is incremented elsewhere in the code (*GameMaster.SwitchPlayers()*).

1.3.2 Locked Room & Puzzle

To implement new changed system requirement 20a, a locked room, we considered several mechanisms of unlocking the room (for example finding a key, passing a test etc.) with the idea that they should be scalable to multiplayer gaming. Unlike finding an object (ie key) to unlock the room, passing a test would work naturally for multiple detectives in the same scenario, so we settling on this idea using a new Unity Scene *Puzzle* to display the puzzle and several new properties (namely *PlayerCharacter.unlockedPuzzle* and *GameMaster.lockedRoomIndex*) and methods (including *PlayerCharacter.unlockPuzzle()* and *PuzzleScript.IsCorrect(int index)* as described above). The class *Puzzle* was added to store the details of a particular puzzle, reading a random puzzle from an external file when instantiated. The new script *PuzzleScript* was added to the *Puzzle* Unity Scene so *PuzzleScript.Start()* is called immediately as the scene is loaded. This method generates a new *Puzzle* object and uses *PuzzleScript.AssignAnswers()* to display the potential answers to the riddle, and *PuzzleScript.IsCorrect(int index)* to verify the player's response to the challenge and handle the consequences. To limit hardcoding we used external JSON files to store the questions and possible responses which could easily be migrated to a web server allowing us to improve the quality and quantity of riddles over time.

1.3.3 Scoring & Leaderboard

Our scoring model differs from Team Watson's original system as they used a 'top three' scoring podium which we adapted to a more traditional leaderboard. We found this change necessary to make the difference between rankings more apparent, to allow more people onto the leaderboard and to reduce the risk of tied multiplayer games. In the new system identical scores are less likely due to an increased reliance on game time; scores are now modified in real time as opposed to being calculated at the end of the game. This is essential when lots of people are playing, such as on open days (as part of non-functional requirement 27). Additionally, point allocation has been modified to fit with a more competitive game, with points being awarded for finding clues and correct accusations, and points being deducted for time taken and incorrect accusations as part of system requirement 13a.

Bibliography

[1] Beldie. C. et al. “Team Watson Architecture Diagram” [Online] Available: http://teamwatson.webs.com/Architecture%20Diagram.png [Accessed: 25- Apr- 2017].

[2] H. Cadogan, S. Davison, T. Fox, W. Hodkinson, C. Hughes and A. Pearcy, "ReqTable.pdf", *Wedunnit!*, 2017. [Online]. Available: http://wedunnit.me/webfiles/ass4/ReqTable.pdf . [Accessed: 10- Mar- 2017].