Implementation Report

Team 12

April 2020

1 Introduction

Software was changed in accordance with requirements change provided in week 1 of Assessment 4 cycle, the updated requirements we reference can be found here. An updated architecture can be found here.New code and changes have been marked in the source code which can be found here (LINK).

2 Required Changes

2.1 Saving

We modified the code to add 11 new classes: Save.java, SaveManager.java, SaveAlien.java, SaveFiretruck.java, SaveFortress, SaveStation.java, SavePowerUps.java, SaveScreen.java, LoadScreen.java, SaveInputHandler.java and LoadInputHandler.java. The save classes handles storing the volatile data and the save manager parses this information using JSON, this is then accessed visually using a button in game or on the start screen to load or save the game. Each save class represents an entity with dynamic data which needs to be stored, these are then stored in save.java. This fulfills the updated requirement of UR_SAVES with the exception of not being able to save during the mini-game. We chose to do this to not complicate the architecture as we would have had to implement it in a inconsistent way and also because of the mini-game's fast nature which would unfairly deposit the player into it when they loaded in. To cause minimal impact to the current architecture we created alternate constructors using the saved data to preserve the architecture whilst following the requirement. Following their architecture we also created new similarly styled screens and buttons allowing saving as well as binding save to the S key. We did this with UR_SCALABILITY and UR_PC in mind as having multiple options helps PC users navigate controls and having buttons helps with portability to touch screens or other similar platforms.

2.2 Power-Ups

We added the PowerUps.java class as well as modifying existing classes such as: FireTruck.java, Alien.java, GUI.java, GameScreen.java. The PowerUps class handles all power up interactions, it is accessed through the game screen class and has five different power up types. Any power up explicitly affecting an entity gets resolved within that entity. The GUI class updates the GUI based on parameters from the PowerUps class. This all fulfills the updated requirement of UR_POWER_UPS. We chose to structure the classes like this so we could add to existing classes and therefore keep the architecture simple. Having one instance of PowerUps in the game screen class as seen in the updated architecture allows for easy access for the entities. We incorporated many GUI changes in GUI.java such as: splash text when power ups are re-spawning, sprite changes and more to provide visual feed back for the power ups. This was important to allow the user to understand when things are happening for UR_ENJOYABILITY and helped the games aesthetic for UR_ATTRACTIVE. We chose for the power ups to spawn randomly and for the player not to know which they will get until they pick it up. We decided this because we thought it would make a more dynamic experience and therefore further achieve UR_ENJOYABILITY.

2.3 Difficulty

We modified the code to add: LevelScreen.java, LevelScreenInputHandler.java and changed GameState.java. When the player clicks the start button they are brought to a difficulty screen which sets the difficulty on entering the game, this is passed to the game screen through the constructor which then sets the difficulty in the game state class. We employed the same structure as the other screens and ensuring that we only called the screen once before the game started, so it had minimal effect on the architecture. We handled all difficulty settings locally within the constructors of classes usually accessing difficulty from the GameState.java indirectly through the game screen, much like our other classes. Although this added a lot of associations and dependencies, the game screen class already acted as the main class so many of these associates were already made. This also maintained the current classes which we felt had a smaller impact on the architecture. This fulfills the updated requirement UR_MANY_DIFFICULTIES as it adds three different difficulties hard, medium and easy. To aid in visualisation we added an explicit title stating the difficulty in the top right corner, since we have minimal reference to the difficulty in actual game play to simplify the architecture. We thought it would aid UR_ENJOYABILITY to state the difficulty in the main screen.

3 Changes

3.1 Alien Behaviour

We modified Alien.java as well as adding our own path finding by creating PathFinder.java which used a breath first search. We did this because the current path finding was inconsistent which hindered UR_PLAYABLE and most of the old patrol GUI was redundant, as it had no effect on anything. We didn't want to overhaul their design as that may have had an adverse affect to the architecture so we restructured the alien class to give each action a use which greatly increased UR_ENJOYABILITY. A natural progression from UR_PATROLS since it explicitly states to avoid ET patrols was to give a reason to avoid the ET patrols, a more stealth based game. So we gave them new behaviours based on new states: patrol state or pursuing state, if a fire engine gets detected they would chase them or if not they would patrol. In keeping consistent with the old architecture we reused their old methods except for any new required change features. We also changed the redundant GUI to give user feed back i.e. if a patrol spots you it changes emoticon GUI based on the player action.

3.2 Hiding Mechanics

In the changing of the alien behaviour we created new balance issues that would effect UR_ENJOYABILITY. In our powerUps.java class we have an invisibility power up, so we decided to extend these to hiding spots on the map in which the ETs would lose track or not be able to detect the player. This had minimal impact to the architecture as hiding was already implemented due to PowerUps.java. So this mitigated the balance effect of changing the aliens and contributed to UR_ENJOYABILITY.

3.3 Mini-Game

In the inherited architecture the mini-game was broken as the game would end sporadically and the outcome would have no effect on the main game, so in order to fulfill UR_MINI_GAME and UR_PLAYABLE we fixed it. We did this by restructuring preexisting methods within MinigameScreen.java as to not change the intended architecture whilst following the requirements.