In this assessment, our approach to change and change management have not been changed. We felt that it worked really well in the last assessment and we will continue to follow the same process. First of all, we have acquired all documentation, code and art assets from the other team. We then read through all the documentation and code to get a better understanding of their game and its structure. Additionally, we have created a new GitHub repository to manage version control and any changes to implementation. This ensures that none of our code or images mix; since we should not be using any of our initial game.

When identifying the initial set of "change requests" we compared the requirements document[1] with the current state of the game's architecture and the features already implemented. This helped us find a set of initial "change requests" which we would begin to act on. To identify possible corrective and perfective changes[1] we play tested the game and looked through the documentation for the code alongside the code itself. Additional changes were suggested when we felt the requirement no longer represented the game we were trying to make.

We will break up the changes being made to the project into perfective, corrective, additive and subtractive changes and assign priorities to each request based on how important the issue being raised is, how easy the change would be to implement, technical and non-technical costs, risks and the project deadline[2]. The priority would be assessed by all members of the team since we are a relatively small group and everybody would be affected by them. Any changes that would mean a change to requirements, architecture, testing or risks would need to be accepted by the group as these changes were more significant[3].

Most of the documentation inherited from the previous group will remain similar in terms of content. We will attempt to maintain or improve traceability by keeping the style consistent throughout our documentation and similar to the previous groups documentation so we can clearly see how the project has changed and evolved. Changes to documentation will be directly influenced by the proposed changes to the software.

Change Process

1. A change is requested by a team member.
2. The change request is assessed using the following attributes by all members of the team: requirements, difficulty of implementation, technical and non-technical cost, risks and estimated time required to make the change by.
3. Following assessment, the request is accepted and assigned a priority or rejected. If the change means the documentation is no longer relevant then the respective document will also be changed to reflect this. The priority system works similarly to the Agile Requirements Change [4]; the highest priority will be implemented first, and whenever a new change is accepted its position in the queue is given by its priority. If two tasks have the same priority the request which was accepted first is allocated and implemented first.

In a few cases a higher priority can be ignored if it is dependant on a change which is currently being made. The list is also regularly reviewed to make sure the order of the list reflects the true priority of the task in relation to those around it as priorities can change.

[1] Can be found on our team website at http://www.teampochard.co.uk/Req4.pdf

Architecture and Requirements changes

The change in requirements for assessment 4:

- Demented waterfowl mode: it must be possible for a small number of waterfowl, including the player and computer-controlled waterfowl, to behave randomly and unpredictably. For a player, the duck must randomly contradict player input. For computer controlled characters, behaviour must be random or unpredictable. Demented characters should be able to return to normal versions of themselves
- Two cheat methods: add two forms of cheating to your game. Invent any type of cheat you like, but you should be sure that the cheats do not make gameplay too easy or unfair.

The corresponding requirements which were added to our requirements document were G14 and G15 respectively.

For G14 we narrowed the requirement down further as to not infringe on requirement S2, specifically the game's fun to play aspect. Random gameplay the player has no control over tends to irritate the player as their skill is no longer the key to success in a game. To combat this we specified that the player must fight off the dementia virus which is the cause of the demented duck mode, this way the player has more game mechanics to master and better players are rewarded. We also specified that the random behaviour for players is restricted to movement only whereas we don't have this restriction for computer controlled characters.

The architectural[2] change made to complete the requirement was the addition of the *DementedPlayer* class - the rest of the changes were purely changing the update functions of existing classes slightly. We used composition to reference the Player instance inside the *DementedPlayer* class opposed to putting all the new methods and logic in the existing Player class and also opposed to inheriting from the Player class. Firstly adding all the new game logic into an already fairly large class would make the class less maintainable and also the class would be doing too many different things. The reason we couldn't use inheritance to make a subclass of the Player is the Player Class would never actually be used. Any time the player became demented we would need to switch out the Player instance with a *DementedPlayer* instance, this design didn't make sense and so just extending the Player class and never using the superclass would have similar problems to the first idea. Even with the decision we made there were drawbacks namely the tight coupling of the *DementedPlayer* class to the Player class and if we were to be carrying on further with the game it would have been a good idea to redesign the architecture of the Player class and other associated classes. Change AR3 in the architecture report.

For G15 we specified that the cheat should be entered in some sort of pause menu. The reason for this being that having the player trying to enter a code while still playing the game would be frustrating as the player could die easily. Also the cheats must not affect the difficulty of the game with mainly visual effects. The architectural decision to make an abstract Cheat class which will then be extended for each Cheat was so every subclass would implement the same

[2] References here relate to our architecture report, which can be found on our website at http://www.teampochard.co.uk/Arch4.pdf.

methods allowing one *CheatInputHandler* class which would work with every Cheat subclass. Change AR1 and AR2 in the architecture report.

Requirement I6 was also created to specify that a cheat could only be entered on a pause screen, adding the *PauseMenu* class. The *PauseMenu* class stops the game from updating and also has multiple *CheatInputHandler* classes which check for each cheat whether a cheat code has been entered and activates the corresponding cheat. This relates to change AR4 in the architecture report

When inheriting the project we looked at the state of the game and compared it to the requirements document, however not much progress had been made and so we had to remove some requirements in order to finish the game and keep the game to a high quality.

One of the first requirements to change were G9 and consequently G8 requiring collectable resources to be part of the game as drops which could then be spent in some way. No part of this was implemented and no assets for this existed. To complete this would require a lot of assets and a large addition to the architecture. A Resource class of some sort would be needed which would most likely extend the Item class. The Player or Screen class would then need to keep track of the resources similar to the score, alternatively a Gamemode / Gamestate class could be created to keep track of the score and resources. A ShopScreen class, ShopItem class and possibly a Shop class would also be needed to added to display what the player can buy and how much for. On top of this sprites for each resource and each shop item would need to be created with some sort of mechanic for each. This requirement was removed.

Requirement G10 was removed because no progress had been made by the previous team apart from a buggy melee system and a few sprites for a melee attack which were not currently in the game. G10 required much more than a gun and a melee weapon. From an architectural point of view two new classes would need to be added for there to be a variety of weapons and any progression. When considering the architectural impacts we felt the following would be needed; a WeaponItem class which would extend the Item class so it could be picked up and contain some information about the weapon to be picked up, then perhaps a Weapon class which would be constructed when the WeaponItem is picked up and this would be contained within the Player class. This change is fairly large on it's own however the amount of sprites needed to add multiple new weapons to the game made it unrealistic and so we let powerups solve the problem of variety in gameplay and also the demented duck mode.

To add to the variety in gameplay we added the requirement of having stackable powerups which will act simultaneously. This was a change to requirement G11 and had no change to the architecture.

Requirement S5 has been removed since we will not be swapping the game with other teams after this assessment however it should be noted the principles of keeping clean and maintainable code will be respected as this will allow us to keep development at a productive rate all the way until development will finish. This change has no direct impact on software architecture.

To meet requirement S2, we needed to make the GUI more consistent and aesthetically pleasing. The changes to architecture which we made to meet this requirement was the addition

of a MenuClouds class which handled generation and rendering of floating cloud sprites for various Screen subclasses. The MenuClouds class is actually instantiated and kept within the DuckGame class, this was to make sure the background stayed consistent throughout the menus. Change AR5 in the architecture report.

Again to meet requirement S2, two new classes were added; *FloatyNumber* and *FloatyNumberManager*. These classes generated and rendered numbers which appeared to "ping off" characters when they were damaged. This replaced a class with a similar purpose to the AnimatedText class which was inherited from the previous group and we found not to be very extensible and hard to maintain since it was tightly coupled to many other classes. Change AR6 in the architecture report.

The last change to architecture to meet requirement S2 was the addition of a new Objective subclass called *KillObjective*. This change was very quick to implement and there were no side effects to the architecture beyond the addition of this class. Change AR7 in the architecture report.

Changes to our GUI and Code[3]

- G5 of the Requirements required us to have a 'health' system. We decided to represent the player's current health with hearts located on the bottom left corner of the screen, where each heart represents 2 health points. The justification for G5 was to provide a challenge to the player as they progress and encounter danger, so we felt that this representation would be better than using a health bar as it would be easier to estimate how many more hits the player could take before they died. Once we took over the project, we added the health bar to the enemies as well. We think that this provides the player with more feedback on how well they are performing and how many hits are needed to take down the enemies.
- In the original game G11, which states that the player should be able to obtain distinct 'powers'. The images of powerups are used to meet this. When a powerup has been picked up, an image of the powerup appears above the stamina bar. Similarly to the power bar, these images have a continuous countdown to show the remaining time for a powerup effect. The countdown for the images are displayed in a clock-like manner with the image changing to a faded version as the remaining time decreases. To keep the interface uncluttered the images of the powerups only appear once they have been picked up in the game and disappear when the allotted time for the powerup runs out.
- I4 requires a minimap on screen. The previous group has implemented a minimap on the top right of the screen. We feel this design is intuitive but can be improved so we have made the minimap transparent, therefore player has more visibility in the game. Therefore it was decided that the top right of the screen was a better fit, since the player can still look at the minimap easily for navigation. The minimap tells the player if there are obstacles in the vicinity through bright high contrast squares on the map. These are easily visible against the background. To prevent the minimap from blending into the environment a black border is used to separate the contents of the map from the similarly coloured surroundings.
- G14 states that players and non-players will act unpredictably and randomly. We implemented this by using a demented check. After a random time, a demented check, in the form of a

[3] The document detailing things changed in implementation can be found on our website at http://www.teampochard.co.uk/Impl4.pdf

circle with a random key inside, will appear on top of the player. The player will need to press the key before the circle gets too small (i.e. time runs out) or they get demented and their controls get randomly manipulated. If the player succeeds, then they will get some power ups. We feel that this implementation adds more fun to the game, the game will be more challenging and more rewarding at the same time.

- G15 states that the player can input cheat code through the use of key combinations. This is reflected in our implementation by first pressing the pause button and then either pressing "UP-DOWN-DOWN-UP" or by typing "BOOM" using the keyboard. The first cheat is a screen overlay in which a sprite of a duck will appear and bounce around the screen. Whilst this happens, the player will receive bonus point. The other cheat is explosive ammo which serves as visual effect and adds explosive sound and effect to the game. The idea was inspired by classic games which used the Konami code[5] such as Contra, Metal Gear Solid,... We feel that this design is easy to implement and will not make the player too overpowered in the game.

•I6 requires that the player should be able to pause the game at any point and return to main menu. We implemented this by using the escape key. Once pressed, the game will freeze and a menu will pop up on the screen with 2 options: Main Menu and Exit game. During this time, the player will be able to enter the cheat code. This implementation enables intuitive interaction between main menu and game screen, and convenient for cheating mechanism.

- We have also improved the main menu which now contains a new game button, level select menu and a settings menu and an exit game button. The menu buttons and the logo follow the same colour so that it looks simplistic and less confusing. Upon being hovered over, the menu buttons will change colour so that the user knows if it's being pressed.

- The level select menu contains a button for each level which is faded unless the previous level has been completed. The user is able to select any level which is no

longer faded within the level select menu. We decided to fade the level buttons to make it clear to the user that those options were not yet available. There is a convention of inaccessible items being represented by a faded version, to show the user that the option is there, but not yet available. We decided to incorporate this as it would be intuitive meaning there would be less confusion.

- The settings menu was added so that the user can alter the music and sound effects. This is because some people prefer different levels of noise to get them involved in a gaming experience. The user is able to alter both the music and sound effects by clicking on arrows which contrast against the background. The contrasting arrows are red so that they stand out and are easy to find. The menu page itself, only contains the arrows and the noise levels to keep it simple.

Code Changes

- **Addition of a transparent minimap** (requirement I4) **-** When performing the pixel lookup for the minimap add alpha. It allows the player to see behind the minimap to maintain awareness but also enable them to see where they're going.
- **Damage frames** - in the player class add an invincibility period, only draw character every other frame, have a flag so there is no damage taken.
- **Change the way numbers were stored -** The previous team used *cast* to turn all *doubles* to *ints* so the precision of floats was being lost, therefore use floats as no need to cast them.
- **Sine wave powerups** (requirement G11)- have a sine function that modifies the position of the power up (position +-1) to give them a bounce effect.
- **Move the player with the mouse** - depending on the section of the screen, use a different sprite. (e.g mouse in upper section look up)
- **Fix the water and flying speeds** (requirement C5) by an updated variable.
- **New projectiles** - we edited projectile texture and added some new parameters and some rotational code so the projectile could move with the mouse.
- **Remove the melee** - by deleting some lines of code.
- **Improve flying** - we took the code from previous game but decided to improve how the character flies so the character does not keep getting stuck in objects. It also allows a greater player input and better feedback.
- **Fix the FloatingNumbers text size** (requirements G4 and G5)- this is to increase user feedback so that the player can see the numbers fade out which improves visualisation, makes the game more aesthetically pleasing and increases the distinction between different numbers.
- **Explosive projectiles cheat** (requirement G15)- we added a new instance of cheat input handler and an input string then added a cheat class which checked if whether it was valid, whether the cheat was already on and if so, the explosion effect was instigated.
- **Implement deranged enemies** (requirement G14) **-** enemies move towards the player and once they are at the player (shake +- random int) the game plays an effect and the enemy spawns projectiles in a 360 degree field.
- **Allow shooting in water -** by removing the old code that disabled shooting in water, which allowed increased playability.

Bibliography

[1]Sommerville, I. (2010) Software engineering. 9th edn. Boston: Addison-Wesley Educational Publishers.
[2]"Introduction to Software Engineering - Deployment/Maintenance", en.wikibooks.org. [Online]. Available: https://en.wikibooks.org/wiki/Introduction_to_Software_Engineering/Deployment/Maintenance. [Accessed: 21- Apr- 2016].
[3]R. Khurana, Software engineering. New Delhi: Vikas Publishing House, 2010, p. 293.
[4]"Agile Requirements Change Management", Agilemodeling.com. [Online]. Available: http://agilemodeling.com/essays/changeManagement.htm. [Accessed: 19- Apr- 2016].
[5]"Cracking the Code: The Konami Code from 1UP.com", 1Up.com. [Online]. Available: http://www.1up.com/features/cracking-code-konami-code. [Accessed: 24- Apr- 2016].