Part I

Reflective Comments

0.1 Technical Issues For Coding Part

This part will mainly cover why Unity Cloud was chosen as our version control system, how the design principles were used in the project, the reflection on implementing design principles, and how the codes were written .

0.1.1 Version Control System

GitHub was utilized as the version control system at first. Later on, unity collaboration Cloud is being used, for several reasons:

1. Difference shown in the project

   The project consists of not only codes but lots of sprite and prefab assets. Difference of sprite and prefab between last version and latest version could not be shown on GitHub, because images(sprites) could not be shown by GitHub. However, unity Cloud is embedded in Unity, the difference could be shown directly by opening that sprite in Unity Project.

2. Simple push and pull mechanism

   Working concurrently with GitHub means that lots of branches should be created. Unity Cloud is simpler, the number of people working on the project, what they are working on, if you should update or upload, they all shown in the Unity Cloud, which lowers down the possibility of version confliction.

3. Easy integration

   Integrating the project using GitHub is complicated, for instance, if subsystem A and B are required to be integrated together, B should be downloaded first, and then it should be imported into the project. With Unity Cloud, above two steps are developed as one step. One single button could integrate the work.

However, it still has a problem. To upload the work, Unity Cloud requires proxy server or eduroam, but sometimes proxy server and eduroam might not work, so the version could not be updated.

0.1.2 Design Principles Implementation

1. Single Responsibility Principle

   Some parts of our codes were modified based on single responsibility principle. For example, in class TextTrigger, the method TriggerText is used for scrolls and the non-player characters. Later on, this method is shown to be unsufficient for both scrolls and npcs, so we added if and else sentences. That is where the codes violated the single responsibility principle. The more responsibility the method has, the lower readability it has. Gradually the method became so complicated that sometimes there is a doubt the developers had, that if the method is used in a right way. After that, that part of codes was rewritten to meet with the principle.
   There were some that were not changed, even though the principle was violated. For example, in class PlayerCollision, a method called InteractionBetweenPlayerAndObjects, has lots of switch cases. The reason is polymorphism for that part is very difficult to implement. The difference between the door object and the scroll object is huge, although they both could be triggered by the play object. However, that part keeps the code clean, because that part could be read easily by other developers without any comments.

2. Dependence Inversion Principle

   For our project, there is only a few interfaces or abstracts, which was considered enough. This principle was not violated because Interactable (which is an interface) is used to detect collision. In our UML class diagram design Interactable and KeyGiver are interfaces, and InteractableObject is an abstract class. Neither of them were implemented by coding, because tag and layer are allowed to use in Unity, which are two polymorphism features. They are almost the same as abstractions, but they could be set directly in Unity UI. For example, in class PlayerCollision there is a method called EnterCollision2D, which is to do actions when collision occurs. There are several kinds of collisions, the way to differentiate them is using the layer. If the layer is Interactable (which is one of the initially designed interface) then something would be done. The example for abstract class is ObjectController, which is the superclass of ScrollController and KeyController. Those abstractions above were sufficient for the project. This principle, sometimes understood as interface oriented programming, is generally using poplymorphism as mush as possible, since the occurences of code

repetition will be reduced significantly.
Applying this principle has given us great convenience to develop the codes. Codes for game object (object in the scene, not class) Scroll, Key, NPC and Door were not written but their layer is set as Interactable. The interaction button is tested using the layer Interactable, even though the codes for interactable objects were not finished. Therefore, test driven development is applied in our development, by meeting with the dependence inversion principle.

3. Liskov Substitution Principle

   This principle is not violated. For the example shown above, the methods in ObjectController were not overridden by ScrollController and KeyController. After applying the principle, the repetition occurrence of the codes is lower down, since the function for that child class would not be written. However, the single responsibility principle was violated, because the methods should also be called by child class or classes, which might includes more than two responsibilities. There are some conflictions among the principles, meeting with all the principles is not always feasible.

4. Law Of Demeter

   The understanding of the law is that the coupling between classes should be lowered down. For example, if class B is a member variable of class A, and class C is a member variable of Class B, then C should better be used in A by calling B. In our programs, the methods of manager classes(or controller, handler classes) are called by trigger classes. The methods of trigger classes are called by other game objects. Therefore, there is no direct relationship between manager classes and game objects. However, extra game objects for loading manager classes as components are required, because manager classes are derived from game object in trigger classes. Therefore, more memory for eatra game objects should be allocated, which is resource-consuming.

5. Open Close Principle

   The open close principle was not well kept in the project. The principle is that the programs should open for extension, and close for modification. From our understanding, it should be based on well-design abstractions and polymorphism. Therefore, when the requirements are

changed, implementing the abstraction in a different way (the new class should be pointed by the abstract or interface reference) rather than modifying original implementation would be efficient. In our project, it was found very hard to change the codes at the end, because the project was not started by writing abstraction. Long-term abstractions were helpful for maintaining the software. For example, interface IMoving-Method should be designed. If the movement of player is changed from 4 directions to 8 directions, the original 4 direction movement would not be modified. The developers could just implement a new 8 direction movement, and reference it.

Meeting with the design principles was the expectation of the developers, yet they were not used to using the principles, especially when the time left for coding is not sufficient, due to our immature time plan. Moveover, in the coding part the importance of UML was highlighted again. In the class diagram, only a few abstractions were designed. The extensibility of the project would be increased by adding more interfaces, which is the benefit of the open close principle.

In our opinion, the parts developed using design principle was quite maintainable, comparing with other parts. They have taken the developers less time to debug than others. If the team could do the project again, the design principles should be understood further, in order to benefit the project.

0.1.3 Coding Regulations Implementation

1. Naming

   The meaning of the variables, functions and classes should be given by their names, rather than comments, according the book clean code. Although it is achieved in our project, the length of the names was too long sometimes. For example, the method in PlayerCollision, is initially called InteractionBetweenPlayerAndInteractables. The purpose of this was shown clearly, but using abbreviation and comments might be better. Personally, the latter one is preferred, so it is changed into IterctPlyrAndItrtbs.

2. Encapsulation

Data encapsulation was utilized in the project. However, the code reusability has been improved using pubic variables instead of encapsulation, because the member variables of the same prefab could be set in Unity user interface with the same piece of code.

0.2 Project management Issues

0.2.1 Work Allocation Improvement

When the project starts, the division of collaboration is simply divided into coding, testing, user interface design and prototyping. After a long discussion within the team members and the supervisor, the project is finally divided into three divisions: game system design and testing, user interface design, game play scrpit design.
The division of the project has three main modification.

Firstly, testing and system design are combined together, since the people who design the system know which parts should be tested (The testing above just consists of unit testing, integration testing and system testing.The acceptance testing will be delivered by customers.)

Secondly, the work of prototyping is deleted, because all requirements could be implemented by the previous prototype. The prototype would be modified if the requirement is changed or a more acceptable prototype is given, based on our weekly meeting discussion.

Thirdly, game play context design is added into the project division. When the project was in the design phase, the importance of context throughout the game is neglected. The users might not be interested in playing the game if they are not attracted by the game scenario. Moreover, without context the educational property of this project will be less likely to implement. Dialogue with non-character player, context shown in the scrolls are included in the game play context design.

The disadvantage of this kind of collaboration is that not everyone could write the unity script, and not everyone could design the user interface (although the user interface design could be discussed by the team, the details are decided by the user interface leader). Therefore, not everyone could learn all skills involved in the project. However, since the team did not do much in the winter vacation, working concurrently would be an efficient way to finish

our project with expected quality.