BSc (Hons) in Applied Computing (Fintech)

INF1009: Object Oriented Programming

Academic Year 23/24, Trimester 2

Report Proposal 2: Improvement to Game

Prepared for: Professors of INF1009

Prepared by:

Ainsley Chang Qi Jie (2302954)/ 2302954@sit.singaporetech.edu.sg

Jovan Lim Yu Hang (2303397)/ 2303397@sit.singaporetech.edu.sg

Lin Zhenming (2302993)/ 2302993@sit.singaporetech.edu.sg

Varsha Kulkarni (2303153)/ 2303153@sit.singaporetech.edu.sg

Yeow Dao Xing(2303175)/ 2303175@sit.singaporetech.edu.sg

Due by:

1 Introduction

In the dynamic world of development, the structure of a game is pivotal to its success. This report provides a comprehensive analysis of a game's architecture, focusing on its core components, implementation of design patterns, engine enhancements, and unique gameplay features. Additionally, it explores how the game addresses a prevalent issue: the lack of interest among teens in scientific subjects, particularly space-themed topics.

The game's architecture is delineated into two main segments: the Game Engine Layer and the Game Layer. This segregation ensures organizational clarity and assists developers in managing different facets of the game. The Game Engine Layer handles fundamental functions such as graphics and controls, while the Game Layer delves into the specific narrative and features of the game. Integral to the game's design are various design patterns, including Singleton and Factory, which optimize functionality and scalability. These patterns streamline the game's code, making it more manageable and facilitating smoother development processes.

Continuous enhancements to the game engine, such as improved map design tools and smoother character movements, elevate the overall gaming experience. Moreover, the game serves as a solution to a pressing problem: the lack of interest among secondary school students in space-themed science. By leveraging a game-based approach and adaptive learning techniques, the game aims to foster interest and learning in space-related topics among teens.

Through educational elements like interactive signs and quiz-based challenges, the game not only entertains but also educates, offering a unique learning experience. Despite challenges in debugging due to the complexity of interactions between different game elements, thorough testing and strategic code notes help mitigate these issues.

In essence, this report provides a comprehensive examination of the game's architecture, highlighting its strengths, weaknesses, and innovative approach to addressing the lack of interest in space-themed science among teens. Understanding its design sheds light on the effort invested in creating an engaging and educational gaming experience.

2 Overall system design

2.1 Components and layers used

The game's architecture is strategically divided into two main components: the Game Engine Layer and the Game Layer. This separation aligns with modern game development principles, where the Game Engine Layer provides a reusable foundation across different projects, and the Game Layer adds the specific logic and assets unique to the game. This structure not only enhances modularity and reusability but also clarifies the development process, making it easier to differentiate between game-specific functionalities and the underlying engine capabilities.

2.1.1 Game Engine Layer

The Game Engine Layer serves as the game's common core, built on the robust and versatile LibGDX framework. This layer is engineered to be game-agnostic, offering reusable components such as rendering systems, audio management, input handling, and physics simulation. For example, the handling of inputs through classes like InputManager abstracts away the complexity of different hardware inputs, providing a uniform interface that can be leveraged across various game scenarios. Similarly, the physics components, facilitated by the integration with Box2D through LibGDX, establish a generic physics engine that can simulate real-world physics without being tied to the specific rules or entities of the game.

In addition, the game utilizes the LibGDX's scene2d package for stage management and UI components, allowing for a decoupled and reusable approach to rendering and interface management. This setup ensures that elements such as buttons, labels, and menus can be created and manipulated independently of the game's logic, promoting reusability. By structuring the Game Engine Layer around these core functionalities, the project ensures that the bulk of the codebase remains applicable and easily adaptable to new games, demonstrating a clear and effective separation from the game-specific layer.

2.1.2 Game Layer

The Game Layer of the game is tailored specifically to the game's context and narrative, providing the unique assets, levels, characters, and mechanics that bring the game world to life. Unlike the Game Engine Layer, this layer is rich with game-specific entities such as **Player**, **Enemy**, **Coin**, and various scene classes like **PlayScene** and **MenuScene**, which embed the game's rules, objectives, and story. Each of these entities extends from generic classes provided by the engine layer but incorporates unique attributes and behaviors essential to the game's identity.

For instance, the Player class extends a generic Entity but introduces specific mechanics like movement controls and interactions with other game objects, which are pertinent only to this game's design. Similarly, the game's levels and environments are constructed using unique tilesets and layouts, utilizing the engine's rendering capabilities to create a distinctive world. The SceneManager class orchestrates the transition between different parts of the game, such as navigating from the main menu to gameplay, encapsulating the flow and progression unique to the game's structure.

Moreover, the Game Layer is where the game's narrative and aesthetics come into play. Custom assets, storylines, and UI designs are integrated within this layer, distinguishing the game from other potential projects using the same engine. Through this specialized layer, the game achieves its own identity and provides players with a unique experience, showcasing how the game layer serves as the definitive context in which the common core engine is utilized and showcased.

2.2 Design pattern implementation

Design patterns serve as standardized methodologies used in software engineering to solve common design challenges. They are essential for creating robust, scalable, and maintainable code. By applying design patterns, developers can avoid reinventing the wheel, reduce code complexity, and improve communication among team members. Design patterns facilitate the development of software frameworks and architectures by offering tested, proven development paradigms. Effective use of design patterns also leads to faster development cycles and enhances code readability, making future code modifications easier and more efficient.

2.2.1 Singleton (Creational)

The simple definition of Singleton would be the creation of an instance and allow the instance to be accessed globally and be used. This is prevalent in our project in the context of the SceneManager Class. In the SceneManager class, we create the instance of the class of the Stack where it can be accessed by other classes. In the class of SceneManager, it would be able to control the stack to show the different playscenes for the appropriate output of the scenes. The SceneManager would be then used by all the other classes to be able to display the Scenes needed for the users.

One way the SceneManager reflects the principles of OOP would be in the fundamentals aspect of Reusability. As the SceneManager would be the central instance to be called globally, it would be the main instance of having their data read by other classes. The SceneManager would be reused every time the instance of the Scenes are required and would then promote the idea of Reusability since it can be reused over multiple times.

2.2.2 Factory (Creational)

The Factory design pattern is evident in the game's EntityManager class, which abstracts the instantiation process of various game entities such as Enemy, Terrain, and Coin. This pattern allows for the creation of objects without specifying the exact class of the object that will be created, promoting flexibility and scalability in the game architecture. For example, within EntityManager, methods like createEnemy() and createTerrain() serve as factory methods, dynamically creating different types of game entities based on the game's current context or level requirements. By utilizing the Factory pattern, the game can easily accommodate new entity types or variations without altering existing code, enhancing maintainability and future expansion. This approach separates object creation from its implementation, allowing for a more modular and decoupled design that aligns with the principles of clean architecture and effective software design.

Due to the way Factory is designed it would promote the Reusability Aspect for OOP Principles. This is as the codes can be reused to be called if more of the objects created are needed. Reusability is shown as anytime more code wants to be used, the factory can be called to reuse it and easily create more objects as needed.

Another aspect of the Design Pattern that reflects OOP theories would be the scalability aspect. As the objects are created in the Factory, the factory would be the basis of all objects and would behave as the core design functionality. Scalability can be done if the objects created need to be enhanced or scaled to be more comprehensive and more detail oriented, the factory can be adjusted to scale up the complexity and to scale up based on the basis needed.

2.3 Improvements to the Engine

One improvement we have made to the game engine was the use of TMX Map loader. By making use of the desktop all “TILED” it can easily help us design the game terrain in a more efficient manner. With the TMX Map loader it significantly increases the simplicity in the way a map can be generated and played within our game. It is as simple as dragging and dropping terrain blocks into our map to create a playable environment. In our playScene we initialize the TMX Map Loader to load the maps in the game with the use of a text string, similarly to how we replace images for different sprites. This improves our game engine as the game can be changed easily from adding a new chest, removing enemies, adding coins, moving the signboards or the terrain. Previously this was done manually but we found it to be impractical for a big map as it was very tedious. We previously had to manually track the position of each terrain and enemy for their x and y position, for example creating an enemy at 1160, 224. With this approach we feel it is much more flexible for users who want to potentially improve or edit our game, and can be done so with ease building as this makes our engine reusable or a variety of games that need to be built.

Another improvement we have made to the game is for it to render different images that are tagged to the player’s input or to a loop. Previously it was just static images, but to provide a better experience for users when playing our game, we decided to implement animated characters and sprites into our game. With our implementation of the animation, it can be easily changed via the different entity files. For example, if there was a need to adjust the type of coins from a gold to silver, this can be easily done by replacing the image in the asset folder with the respective number of images, in our game it is currently set to 5 different images for each iteration of the animation. A time loop is then used to change the image every second or however long you wish for it to be, to produce the animated sprite.

As for the player sprite, it is a little different as we have tagged each image to a different key that the user presses. This was easily implemented as we could make use of what we had done previously on our input manager where it registered the user’s input. Therefore, once a key is pressed it will change images according to the user’s input and will then loop through the various images we have for running either left, right or jumping.

2.4 Unique Features (Educational feats and Thrill aspects)

Signboard

The game features educational signboards strategically placed throughout the environment. These signboards provide relevant information and hints to players as they navigate the world, seamlessly imparting knowledge without interrupting gameplay. Each signboard is unique, offering context-specific insights that encourage exploration and interaction.

When a player approaches a signboard, relevant information is displayed on-screen, offering valuable insights or hints. Once the player moves away, the information gently fades, allowing them to focus on the path ahead. This mechanism ensures that knowledge is imparted seamlessly throughout the game, enhancing the player's experience.

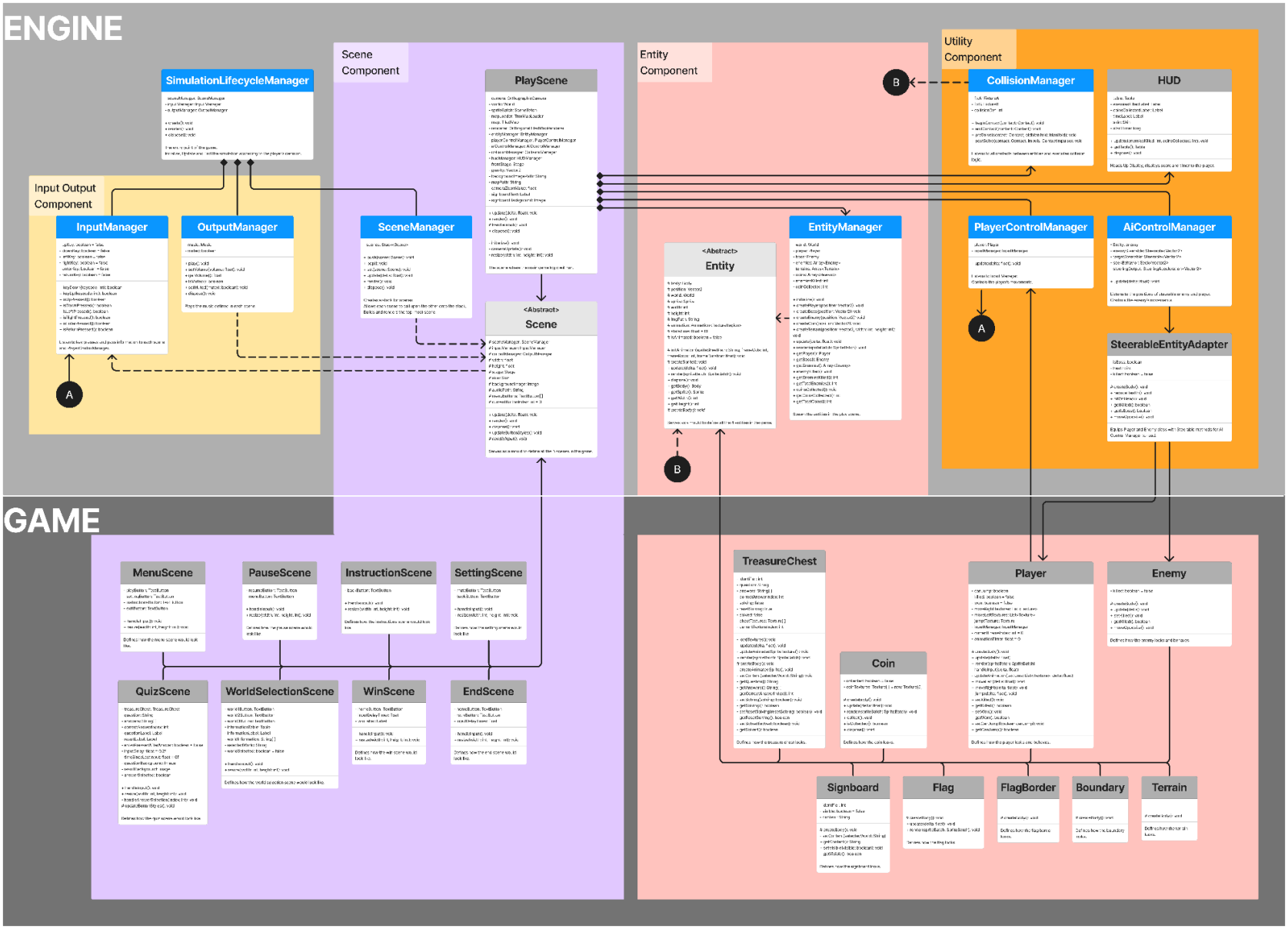
Treasure Chest

Treasure chests guarded by NPC enemies are scattered across the map, each containing a quiz that challenges players' knowledge. These quizzes require the application of information learned from signboards, turning every piece of knowledge into a potential key to unlocking these chests. Solving these puzzles not only tests players' retention but also contributes to dismantling barriers that guard the final flag.

Interactive quizzes in treasure chests serve a dual purpose: testing players' understanding and providing a sense of progression as each chest is conquered. Access to the final flag, the game's completion condition, is granted only once all treasure chests have been unlocked, turning exploration, learning, and problem-solving into tangible achievements.

.

3 UML of Game Engine and Game



4 Object Oriented Principles

4.1 Abstraction

Abstraction simply defined would be the simplification of complex action and turning it into simple functions that are able for users to not understand the difficult details of how the complex function would behave and how the outcome is internalized into the game. This core principle has been used in multiple aspects of the game. This is as for the game, there are areas where there are difficult complexities of behaviors in the system that are defined as functions that can be used by our programme. This core Object Oriented Principles is showcased by hiding the complex function into classes that would be responsible for setting the behavior and the main programme just has to call the function as required.

An example that is relevant would be the different menu scenes handling. As there are 10 classes of menus in the entire programme, it would be difficult for the programme to understand how each of the functions work. In our system, we would create the scene Manager class that would handle just pushing and peeking the required scenes in a queue. The Scene Manager class would be able to call the different classes in the system as required and the Scene Manager primary responsibility would be to peek and reflect the proper scene for the user then, This would reflect Abstraction as the Scene manager class would not be responsible for understanding the complexities of the different scene class. Instead, it has to just push the relevant scenes into its queue as required when certain actions have been fulfilled, like when the player selects the level, it would then push the play scene and not understand the rationality of how the play scene would behave. This would then turn the complex function of the different scenes into just one class that would handle the different output then.

4.2 Encapsulation

Encapsulation would be the bundling of different data attributes and giving the priority to be utilized by different classes in the package and restricting access when required. It would set the different classes to perform as required and not allow certain classes in the same packages to access data that are restricted to the user to then not allow the different classes to alter the data as the data might have a higher sensitivity level of information that would require a higher level of data restriction. The 3 main encapsulation methods that are apparent in the program would be Private, Public and Protected.

The Private class primary function would be to restrict access to the level of information to only the same class. For private classes, it would primarily be handled by the class itself. However, this data is to be passed to other classes as the information has to be interacted with by other classes and thus they tend to

come with a public method. For Public, the definition would be that it is able to interact with other classes as information has to pass around for the data to be conveyed and data of interactions have to be handled simultaneously. Thus resulting in the requirement of public methods. This can be illustrated in the class of the HUD as a simple illustration of how the programme is able to run the different encapsulated variance in its objects. In the HUD class, the element of time is added as the lose condition of the game would occur when the time was to reach 0 and the conditions are not fulfilled. Thus, this would require the class of timing to be able to perform its own independent role. However, it is to note that timing would require to be a private class as the timing would run independently and should not require other classes to be able to interact with it and alter the timing as it should be an independent variable by itself. As such, this would require timing to be set to private to only allow the class of HUD to manage the counting down of the timing. However, it is to note that the timing is a pivotal information that has to be shared for users to be able to see the timing and when the countdown timer reaches 0, the end scene have to be pushed into the queue to show the end results. Thus, this would come with a public method called getElapsedTime which is able to read the elapsed time and once it reaches 0, it would then perform interaction based on conditions set for it. The method has to be set to public as it would then be stored in the same class as the Elapsed time to then read the elapsed time and allow for the information to be shared to other classes as they would then not be required to be shared to different classes via the public method.

For protected attributes, this would run when parent classes are involved. In the programme this would run with the Entity class that have the protected attributes linked and liaise with the width and height. Protected class would be similar to public in a method that both are able to pass the data around but only the class in the same packages are able to affect and alter the values. This would be relevant to the entity class as all entity classes have different width and height and the information from entity would then require to be passed around before it is being accessed and shared and altered by different classes.

4.3 Polymorphism

Polymorphism is allowing the different objects of different classes to be adaptable and to respond differently when different classes call for the function even as they have the same message method. This allows flexibility of the codes and allows the classes to be reused but in different manners making the base more prominent and is able to be a one fit all scenario. Polymorphism would be attained mainly through Method Overriding.

For the programme examined, method overriding is used in the class of playscene. This is the scene

Polymorphism in Javex manifests primarily through method overriding, allowing classes to implement parent class methods based on their specific requirements. This principle enables Javex to handle various game elements with more flexibility.

Method overriding is extensively used in the scene management system. Each subclass of Scene, such as MenuScene or PlayScene, provides its own implementation of the handleInput() method. While the Scene class provides a general framework for what every scene should do, the actual content and behavior of these methods are tailored to the specific needs of each scene, enabling diverse functionalities while maintaining a consistent interface.

This use of polymorphism allows Javex to dynamically change its behavior during runtime, depending on the current active scene. It eliminates the need for cumbersome conditional statements to determine the scene type, leading to cleaner and more understandable code.

4.4 Inheritance

Inheritance is a cornerstone of Javex's architecture, promoting code reusability and logical hierarchy. The engine defines base classes with common functionality from which other classes can inherit, reducing redundancy and enhancing maintainability.

For example, the Scene class serves as a base class for various specific scenes like MenuScene, PlayScene, and EndScene, each inheriting common methods and attributes. This mechanism not only streamlines code updates and maintenance by allowing changes in the Scene class to propagate to all derived classes but also facilitates the consistent implementation of essential scene methods.

Furthermore, inheritance facilitates the polymorphic use of objects. A function expecting a Scene object can operate with instances of its subclasses, allowing for flexible code that can handle different types of scenes uniformly.

5 Limitation

For the game, there are a few limitations due to the nature of the way the game was developed. The first area of limitation would be the nature of debugging the program. This is because the game is structured using different classes, each serving different purposes and responsible for their own areas. While this allows for reusability and scalability, it can also lead to debugging inflexibility. Each function performs its own task, and as the classes communicate with one another in specific ways to achieve the overall required action, identifying the source of a bug can be challenging. The game's design includes multiple inheritance hierarchies, so if a new element is implemented and the overall function deviates, pinpointing the class causing the issue becomes difficult. With a higher number of classes, untangling each layer to find the root cause of a bug becomes more tedious. However, this challenge can be mitigated by thorough testing and commenting on each interaction. By identifying how each element causes interactions to differ, users can more easily identify and resolve unexpected interactions or issues that arise, allowing for a more tailored approach to debugging. Additionally, the game's heavy reliance on the LibGDX framework introduces dependencies and limitations. Keeping abreast of updates and ensuring compatibility with newer versions of LibGDX could be a challenge, potentially impacting the game's long-term sustainability and adaptability to evolving development practices. Strategies to address these limitations include thorough testing, efficient debugging practices, and staying informed about LibGDX updates to mitigate potential compatibility issues.

6 Contribution

6.1 Ainsley Chang Qi Jie

Ainsley contributed to the project previously to the creation of the input and output manager, as well as making sure it was able to link and work well with the rest of the game code. Ainsley also contributed to rendering the different sprites needed like the player, enemy, terrains, flag, treasure chest and coins. Ainsley also helped to ensure there was animation done for the sprites that were either tagged to player input or making use of time loops.

6.2 Jovan Lim Yu Hang

Jovan contributed by assisting the team by working on all the scenes and scenes management and unique entities/features onto the game. One of the more complex examples is the world selection scene which will initialize different Box2D variables unique to the world selected. He also implemented unique educational features such as signboards, which display unique information when a player is on them, and treasure chests, which trigger quiz scenes based on information learned from the signboards. Jovan was also the main brainstormer for solidifying the game's educational ideas and thrill. All these required complex coding on the Scenes, Entities and Collision classes and managers.

6.3 Lin Zhenming

Zhenming contributed by assisting everyone on the team to set up their development environment, set up GitHub and also organize ideas and feedback into clear actionable items for the team to achieve weekly progress. Zhenming also spent considerable time to proofread and improve other team members' code before merging them into the main branch, to ensure tasks can be distributed and consolidated seamlessly. In addition to his effort to promote collaboration, he worked on certain tasks individually as well, such as (1) Entity rendering with TILED app to create 3 different levels of gameplay, (2) Collision(s) handling to create a satisfactory gameplay experience for players, (3) Generating the final gameplay idea and content (questions, answers and facts). In short, Zhenming shouldered the critical components of the game while ensuring smooth code integration with creations of other members.

6.4 Varsha Kulkarni

Varsha contributed by drawing the base sketches of the "Earth" map on TILED. Other than that, she helped initialize the report and presentation deck, while focusing on the "Introduction" section.

6.5 Yeow Dao Xing

Dao Xing contributed to the HUD management for the implementation of the countdown sequence timer and the initiation of different scenes. Dao Xing also did the initial enemy control movement from the Game Engine and is able to integrate it with OOP Concepts to be utilized in future game development in the context of Learning Space. The initial design was built to be able to call differently with OOP Principles to be reused and for a larger scale. The Initial Factory was initialized by Dao Xing to be able to run and be scaled and reused for Learning Space.