Code Quality Review Report

Executive Summary

Since Unity's primary backend scripting language is C#, which is similar to Java in its class-focused file structure, it was both feasible and beneficial for us to follow object-oriented programming paradigms. In addition, we drew upon our knowledge of time complexity and software design patterns from previous courses to create additional principles by which to hold ourselves accountable. From a time complexity standpoint, we wanted to avoid computing backend processes on every frame whenever possible, and instead opt for event-driven logic that would only execute when specific events were invoked. This philosophy ensured that our code base would run in constant time (i.e. O(1)) whenever possible, which is the most efficient time complexity for programs to run in. Additionally, since its internal components resemble those of an Android smartphone, the Oculus Quest is an underpowered system to begin with, and we knew that rendering the application's frontend aesthetics would require considerable processing power. As such, we wanted to minimize the amount of resources consumed by actual code execution. Next, we sought to follow clean-code principles, such as SOLID object-oriented design and minimization of hard-coded constants, in order to ensure that our modules were easily extendable, reusable, and understandable. We expected that adhering to these principles would greatly assist us in debugging logic errors, and we were correct. Finally, we emphasized the importance of using structural and behavioural design patterns in our complex software entities. This was integral to our development process, as since many of the application's event-listening objects are decoupled from their event-triggering counterparts, numerous abstract dependencies exist in the project's code base. This document serves as a post-development reflection on the extent to which this envisioned development process was adhered. It contains two key modules; the first is a high-level overview of the software development principles that were just introduced, while the second is a more detailed breakdown of how closely each module in our code base complies with these principles.

General Principles

1. **Avoid per-frame computation whenever possible**

MonoBehaviour, the parent class of all non-static C# scripts in a Unity project, contains an *Update()* function that is called once per frame. It is tempting for developers to insert all game logic and computations into *Update()* in order to ensure that their code will run, but this practice is highly detrimental to the project's runtime. Instead, it is advised that developers implement event-driven logic in their Unity projects, which often results in functions only needing to be invoked once every event, rather than once every frame. While not identical to the difference between constant and linear time complexity, this distinction is analogous to it, and considering the already-reduced processing capabilities of the Oculus Quest compared to other VR systems, its effect on the application's efficiency is not inconsequential. As such, when writing

the code for ArachnoTherapy VR, we sought to use the *Update()* function as little as possible, and instead opted for event-driven logic.

Examples of this principle being followed in our source code include the manner in which *Grabbable.WhenReleased()* and *RoomParameterGrabbable.SetGrabStatus()* are implemented. *Grabbable.WhenReleased()* computes translational and rotational velocity vectors for grabbable objects when they are released from the user's hands, while *RoomParameterGrabbable.SetGrabStatus()* invokes *RoomController.UpdateRoomConditions()* whenever the user grabs an object containing it. It would not be unreasonable to expect Unity developers to run per-frame computations for both of these examples; that is, calculating a new velocity vector on every frame in which the object moves, or updating the room conditions on every frame in which a room parameter is held by the user. However, our implementation is structured in such a way that these functions only need to be called once per event. Both functions correspond to an event listener for the Trigger button on the Oculus Touch controllers, and thus, only ever need to be invoked when the state of this button changes. As a result, the inefficient velocity calculations are only ever performed on the exact frame in which a grabbable object is dropped, and *RoomController.UpdateRoomConditions()* is only invoked on the exact frame in which a room parameter grabbable is grabbed.

2. **Minimize use of hard-coded constant values**

One of the most common smells in modern code, as we learned in ENSE 374, is the presence of hard-coded constant values, which inhibit both the code's flexibility across situations within the same application, and the code's reusability in other applications. Instead, it is better to use variables that can be toggled across multiple instances of the same program. Perhaps the clearest illustration of this principle's importance in ArachnoTherapy VR arises whenever an *AbstractButton* object is toggled, as this invariably invokes a translation along the object's local Z-axis. Since none of the application's *AbstractButton* objects share a common position, orientation, and world-space scale, unique position vectors are required for each object's "pressed position" and "released position." Rather than attempting to hard-code every single pressed and released position for every single *AbstractButton* object, we simply created public variables in the *AbstractButton* script, which were then instantiated at compile-time through the Unity Inspector window.

Some hard-coded constants are present in our code, as we did allow exceptions for when the value itself was justifiable or easily understood. For example, whenever a spider collides with a wall, it chooses a random orientation around its local Y-axis between -180˚ and 180˚, and rotates to it in order to simulate the action of turning around. These two integers are hard-coded, as from our point of view, they are rather obvious limits for a random rotation function. Furthermore, were we to extend this module to other creatures, we would almost certainly implement the exact same behaviour for whenever those creatures collided with walls. As such, reusability would not be affected.

3. Use structural and behavioural design patterns to manage complexity

Since the objects used to invoke events are usually different from the objects that are affected by events, the code base of ArachnoTherapy VR contains numerous dependencies. In addition, although there are only two primary methods of interacting with objects in ArachnoTherapy VR, many of these interactable objects possess additional or unique properties that require additional backend logic. We were aware that this degree of complexity would be necessary, and as a result, our priority was to maximize our code base's cleanliness and readability. Structural and behavioural design patterns were absolutely integral in mitigating the aforementioned issues and fulfilling this priority. Examples of non-creational design patterns used for the ArachnoTherapy code base - although creational design patterns were also used, and are discussed in a subsequent section - include the Facade pattern, which was used for button event invocation, and the MVC pattern, which was used for the spider creation interface.

Since the button event architecture contains text, audio, video, and video-plus-audio events, references to several different Unity Engine components are required. Rather than providing knowledge of all of these libraries to *AbstractButton* and its derived classes, we simply included a reference to an *AbstractButtonEvent* within the button classes, and encapsulated the event-specific logic within the concrete child classes by overriding *AbstractButtonEvent.ExecuteEvent()*. This Facade-based implementation ensured that the button classes would only ever call an abstract, shared function, and that if additional button event types were needed, they could be added without jeopardizing the functionality of the underlying button logic.

As for the spider creation interface, using the MVC behavioural design pattern was an obvious choice. In addition to a collection of static data, such as lower and upper bounds for the amount of spiders, and a list of spawn locations, the spider creation interface contains fourteen different buttons that each serve a unique purpose. It would have been exceedingly difficult to combine any of these functionalities without creating a GOD class. Thus, we chose to apply a modified version of the MVC design pattern to the spider creation interface. In this system, *SpiderCreatorSingleton* and *SpiderCreatorModel* comprise the model, as the former contains all static data, while the latter references this static data and also contains creation, deletion, and amount adjustment functions, although it itself is not capable of invoking them. Instead, invocation of each operation is handled by an individual controller class. Examples of controller classes include *SpiderDiversityController*, *SpiderAmountController*, *SpiderIntensityController*, *SingleCreationController*, *SingleDeletionController*, *MassCreationController*, and *MassDeletionController*, which all derive from *AbstractButton* and maintain a reference to a *SpiderCreatorModel* instance. Finally, the view is simply the room in which the spiders are created and observed, which directly updates based on the user's operation of the controller functions. The use of the MVC design pattern ensured that no GOD classes would be created from the implementation of the spider creation interface, and also that existing logic for user-button interaction would not need to be modified.

4. **Follow the principles of SOLID object-oriented design**

As specified by Robert C. Martin, the tenets of SOLID object-oriented design are as follows:

- **S**ingle Responsibility Principle
    - A class should only have one reason to exist, one reason to change, and one primary responsibility. Directly related to this principle is the issue of GOD classes, which we previously discussed when explaining the spider creation interface architecture.
- **O**pen-Closed Principle
    - Software entities should be open for extension, but closed for modification. In other words, the addition of new behaviour to a software entity should never compromise the functionality of existing behaviour.
- **L**iskov Substitution Principle
    - Concrete subclasses must be substitutable for their abstract parent classes. In ArachnoTherapy VR, for example, an *AbstractButton* object is able to call *AbstractButtonEvent.ExecuteEvent()* and *VideoButtonEvent.ExecuteEvent()* without inherently knowing the difference between the two.
- **I**nterface Segregation Principle
    - Classes and subclasses should not be forced to depend on unused or unnecessary features. The separation of spider creation interface functions into individual classes is an example of this principle being followed, as each controller class handles one function, which reduces coupling and more easily facilitates extension.
- **D**ependency Inversion Principle
    - Software entities should depend on abstractions, rather than concretions. For example, the object interaction module *Grabber* in ArachnoTherapy VR maintains references to *AbstractGrabbable* and *AbstractButton*, but does not contain any logic for their concrete subclasses. This principle dramatically simplifies the logic in *Grabber*, as it never needs to distinguish the exact type of grabbable with which the Oculus controller is interacting. Consequently, the *AbstractGrabbable* framework is rather easy to extend if new types of grabbables are desired, as additional concrete subclasses can simply depend on the abstractions of the parent class.

Module Review

The truncated path names found in the *Files* cells assume that the reader is already located in the Unity project's Assets directory. A full path name might look something like this: *your_home_directory/ArachnoTherapy VR/Assets/Scripts/Buttons/AbstractButton.cs*

Category	Button Events
Files (in Scripts/Button Events)	AbstractButtonEvent.cs (base class), AudioButtonEvent.cs, VideoButtonEvent.cs, TextButtonEvent.cs, AudioAndVideoButtonEvent.cs
Design Patterns	**Facade** (AbstractButtonEvent.ExecuteEvent() encapsulates all specific event logic)
Dependencies Not Native to UnityEngine Namespace	UnityEngine.Video.VideoPlayer (VideoButtonEvent, AudioAndVideoButtonEvent) UnityEngine.UI.Text (TextButtonEvent)
Time Complexity	All functions are O(1). There is no use of MonoBehaviour.Update() in any of the classes.
SOLID?	S - Each concrete subclass handles exactly one event type. O - Adding additional event types is trivial and would not affect existing event types. L - Other classes only ever reference an AbstractButtonEvent. I - Every subclass overrides every function defined in the superclass for a specific reason. Additional logic for a specific event type is confined to said event type. D - The dependencies in the concrete subclasses are also concrete, but there is no abstraction-based alternative for any of them. The underlying structure in each of them, however, is dictated by the abstract superclass.

Category	Buttons
Files (in Scripts/Buttons)	AbstractButton.cs (base class), ChildButton.cs, ConcreteButton.cs, FinishButton.cs, TransitionButton.cs
Design Patterns	N/A (standard OOP inheritance and polymorphism)
Dependencies Not Native to UnityEngine Namespace	AbstractButtonEvent (all classes) RoomController (ChildButton, TransitionButton)
Time Complexity	All functions are O(1). There is no use of MonoBehaviour.Update()

	in any of the classes.
SOLID?	S - Each concrete subclass handles exactly one set of button properties. O - Adding additional button types is trivial and would not affect existing button types. L - Other classes only ever reference an AbstractButton. I - Every subclass overrides every function defined in the superclass for a specific reason. Additional logic for a specific event type is confined to said event type. D - The AbstractButtonEvent dependency shared by all classes fulfills this principle. The RoomController dependency shared by two concrete classes is concrete, but since RoomController has no instance-specific properties, we do not consider this dependency to be problematic.

Category	Controller Input
Files (in Scripts/Controller Input)	Controller.cs, Grabber.cs, HapticGenerator.cs, HapticReceiver.cs, Movement.cs
Design Patterns	**Facade** (AbstractGrabbable.SetGrabStatus and AbstractButton.OnPress / OnRelease encapsulate all specific interactable object logic)
Dependencies Not Native to UnityEngine Namespace	AbstractButton, ChildButton, AbstractGrabbable, Controller, OVRInput (Grabber) Controller, HapticGenerator, OVRInput (HapticReceiver)
Time Complexity	Update() is used extensively in both Grabber and Movement, as per-frame event listeners are mandatory in the former, and per-frame physics calculations are mandatory in the latter. No iterative processes occur in either. Grabber.OnTriggerStay() is O(N) in the worst case. All other functions are O(1).
SOLID?	S - Each concrete subclass handles exactly one event type. O - Adding additional event types is trivial and would not affect existing event types. L - N/A. Grabber, Movement, and HapticReceiver are not referenced by any other class, and Controller and HapticGenerator are standalone classes. I - Grabber is a GOD class in the sense that it contains logic for both object grabbing and button pressing, but this decision was made with the intent of minimizing the number of scripts that

	depend on OVRInput, as this library was not written by us. D - The ChildButton dependency in Grabber breaks this principle, and there is unfortunately no straightforward way of refactoring this, as ChildButtons are the only type of button that are invoked *without* the need for a hand collision.

Category	Grabbable Objects
Files (in Scripts/Grabbable Objects)	AbstractGrabbable.cs (base class), Grabbable.cs, HandleProxy.cs, Remote.cs, RoomParameterGrabbable.cs (derives from Grabbable)
Design Patterns	N/A (standard OOP inheritance and polymorphism)
Dependencies Not Native to UnityEngine Namespace	AbstractGrabbable (all classes) Grabbable, RoomController (RoomParameterGrabbable) AbstractButton, OVRInput (Remote)
Time Complexity	All functions are O(1). There is no use of MonoBehaviour.Update() in any of the classes.
SOLID?	S - Each concrete subclass handles exactly one object type, with the most common general case being handled by Grabbable.. O - Adding additional object types is trivial and would not affect existing object types. L - Other classes only ever reference AbstractGrabbable. I - Every subclass overrides every function defined in the superclass for a specific reason. Additional logic for a specific event type is confined to said event type. D - The dependencies in RoomParameterGrabbable are concrete, but in a similar vein to ChildButton and TransitionButton, they are not considered to be problematic.

Category	Helper Scripts
Files (in Scripts/Helpers)	Handle.cs
Design Patterns	N/A
Dependencies Not Native to UnityEngine Namespace	N/A

Time Complexity	Update() is used because per-frame rigidbody movement is required. Start() and the contents of Update() are O(1).
SOLID?	N/A

Category	**Room Logic**
Files (in Scripts/Room Logic)	ManyToOneAudioSource.cs, PlayAudioOnDelay.cs, PlayAudioOnDisplacement.cs, PlayAudioOnHeadCollision.cs, UpdateRoomConditionsOnPress.cs (Room 0) UpdateRoomConditionsOnCollision (Room 4) RoomActivator.cs, RoomColliderArea.cs, RoomController.cs (Rooms 1-5)
Design Patterns	N/A
Dependencies Not Native to UnityEngine Namespace	RoomColliderArea (RoomActivator) AbstractGrabbable (RoomController)
Time Complexity	The Update() function is used in PlayAudioOnDisplacement, as it requires a per-frame event listener. However, its components are O(1). All other functions are O(1).
SOLID?	S - Each class exists for exactly one specific purpose. O - Adding additional Room 0 logic classes is trivial and would not affect existing ones. RoomController is easily expandable. L - With the exception of RoomController, which is a parentless script, no classes in this module are referenced by classes outside of this module. The principle may not be fulfilled, but it certainly is not broken. I - All of the Room 0 logic components are encapsulated from one another, and since the process by which conditions are checked is different from that by which room parameters are updated, RoomController is also considered to be acceptable. D - Abstractions are used as dependencies whenever possible (i.e. AbstractGrabbable in Room Controller). Any concrete dependencies represent situations in which no alternative exists.

Category	**Spiders**
Files (in Scripts/Spiders)	AbstractSpider.cs (base class), BasicSpider.cs, IntermediateSpider.cs, ComplexSpider.cs, ExtremeSpider.cs

Design Patterns	None, but potential for **decorator** if models and animation controllers were not unique across every spider type
Dependencies Not Native to UnityEngine Namespace	N/A
Time Complexity	Update() is necessary when a spider is moving, as per-frame calculations are applied to its rigid body component. All other functions are O(1).
SOLID?	S - Each concrete subclass handles exactly one spider type. O - Adding additional spider types is trivial and would not affect existing spider types. L - Other classes only ever reference AbstractSpider. I - Every subclass overrides every function defined in the superclass for a specific reason. Additional logic for a specific spider is confined to an overridden function in said spider's class. D - The underlying logic in the concrete classes is dependent only on the structure of the abstract parent class.

Category	**Spider Creation Interface**
Files (in Scripts/Spider Creation Interface)	SpiderCreatorSingleton.cs, SpiderCreatorModel.cs (model) SpiderAmountController.cs, SpiderDiversityController.cs, SpiderIntensityController.cs, SingleCreationController.cs, MassCreationController.cs, SingleDeletionController.cs, MassDeletionController.cs, DefaultState.cs
Design Patterns	**Model-View-Controller** (invocation and execution of functions) **Singleton** (data pertinent to every interface instance) **Factory** (instantiation of spiders)
Dependencies Not Native to UnityEngine Namespace	AbstractButton (parent of all Controller classes) SpiderDiversityController (DefaultState) SpiderIntensityController (DefaultState) SpiderCreatorSingleton (SpiderAmountController, SpiderDiversityController, SpiderIntensityController, SpiderCreatorModel) SpiderCreatorModel (all Controller classes)
Time Complexity	SpiderCreatorModel.CreateAllSpiders() and RemoveAllSpiders() are O(N) in the worst-case scenario. All other functions are O(1).
SOLID?	S - Each Controller class handles exactly one functionality of the spider creation interface. SpiderCreatorModel contains methods for

	all of these functionalities, but cannot actually invoke them itself. Thus, it is not considered to disobey this principle. O - Adding additional functions would require extension of SpiderCreatorModel, but not modification. L - The Controller subclasses cannot be substituted by AbstractButton, as they contain several additional functionalities that neither AbstractButton nor any other children of AbstractButton need. I - From both an observational perspective and an invocation perspective, all functions of the spider creation interface are perfectly encapsulated from one another. D - Any concrete dependencies represent situations in which no alternative exists