**Design Patterns – Unity Book**

When approaching any project, the KISS (Keep It Simple, Stupid) principle is always applicable, meaning all unnecessary complexity should be avoided.

The SOLID Principles are five core elements of software:

**Single responsibility**

Each component module/class/function is responsible for one thing. Keep classes as short as possible and break up all functionality where necessary. Unity does some of this work for you with things like components attached to game objects. Scripts should be designed in the same way, many small components combined to create complex behaviour. Below, right preferred ahead of left:

A screenshot of a computer code

Description automatically generated with medium confidenceA screenshot of a computer program

Description automatically generated with medium confidence

The player script can still manage other scripts, but the functionality is broken down into separate classes.

Classes should be short to enhance readability. A good guideline is to limit their line to 200-300 lines of code. A team should decide on a limit and stick to it. Small classes also simplify the concept of inheritance, they are more extensive since they are easier to modify without breaking. Also, smaller classes are more suitable for reuse.

**Open-close**

Classes should be open for extension but closed for modification. Meaning new behaviour can be added but original code should not be changed. If you have many different classes which share similar functionality, you can create an abstract class or interface to define the functionality and then subclasses can provide their own implementation of these methods. Classes that use all of these subclasses can interact with their interfaces. One should take advantage of the principles of abstraction which is deeply rooted in OOP programming.

**Liskov substitution**

Subclasses must be suitable for the superclasses they inherit from. If you are removing features from base classes to implement new subclasses you are violating Liskov substitution. Keep abstraction simple by minimising the complexity of base classes, they only need to express common functionality of their subclasses. Sub classes functions and fields should have the same signature as their base classes. Sometimes it might make sense for similar classes to inherit from separate parent classes. You can always pass functionality through interfaces as opposed to inheritance for specific behaviours and then classes can mix and match from these interfaces.

A screenshot of a computer program

Description automatically generated with medium confidence

Functionality is passed through interfaces as opposed to inheritance. Road vehicles and rail vehicles are two separate concepts.

Circle-eclipse problem states that not every “is a” relationship translates into inheritance. As above both trains and cars are vehicles but they do not work in the same way and cannot therefore inherit from a vehicle class.

**Interface segregation**

Users should not be forced to depend on interfaces that they do not use. This means that large interfaces should be avoided and methods should be short to maximise flexibility.

A picture containing text, screenshot, font, diagram

Description automatically generated

Interfaces can be divided into separate smaller ones. Classes can then extend and implement these interfaces where needed. One can then impose many of these interfaces to implement the required feature of that particular class.

**Dependency inversion**

High level modules should not import anything from love level ones, instead they should both depend upon abstraction. Dependency or coupling is when one class has a direct relationship with another. If one class uses another significantly, changing that class it depends on might break the class using it or vice versa. Large amount of coupling is a sign of bad code and is not good practice in software development. Thus, we should aim for loose coupling between classes. A class should have its internal parts working together without relying too heavily on others. An object is cohesive when it works on internal logic. One should try to achieve loose coupling and high cohesion. Without this a program will be hard to modify.

A diagram of a door

Description automatically generated with low confidenceA picture containing text, screenshot, diagram, font

Description automatically generated

When implementing a switch, if it is tightly coupled to the door, it cannot be used for anything else. The switch should therefore be solely responsible for invoking something to happen. The lower level object can inherit from this functionality. Adding the ISwitchable interface, the Switch class can depend on this and not the door directly. Thus, the interface allows for abstraction in between the classes and reduces the coupling which promotes reusability.

A picture containing screenshot, text, font, diagram

Description automatically generated

Now the ISwitchable class can be used to interact with a variety of objects and not just doors.

One could also implement this with an abstract class. Abstract classes can have fields and methods predefined and define access modifiers, which provides many advantages. Thus, logic can be implemented within the abstract class to define common functionality of subclasses. Problems do arise from inheritance when you wish to extend functionality of two different base classes in a derived class. This is where interfaces can be used as you can choose multiple relationships. However, with an interface they only declare functionality. The logic within methods needs to be defined in the implementing classes.

A picture containing text, screenshot, font, diagram

Description automatically generated A picture containing text, screenshot, diagram, design

Description automatically generated

**Design Patterns**

Built-in design patterns in Unity

**Game loop –** Unity automatically takes into account varying speeds of hardware by providing methods to fix the frame rate and also implementing fixed time steps.

**Update –** The Update() method allows for you to update objects at each frame automatically in MonoBehaviour.

**Prototype –** The prefab system in unity allows developers to create an object one and then clone multiple times. This means that defining a separate class to spawn every time a new instance is needed can be avoided.

**Component –** Unity has an inbuilt component system. This allows developers to build large complex entities made up of small components as opposed to constructing large classes.

Factory Pattern

At runtime, it is not often known in advance what has to be created during a run of the game, therefore, you can create specialist objects that create objects. The factory pattern involves spawning objects with the aim of decluttering code. If objects to be spawned share an interface or base class, they can contain their own logic for construction to declutter the factory class.

Object pool

This is an optimization technique to assist the CPU in initialising and destroying objects. The object pool works by initialising a set of game objects and leaving them in a deactivated pool. When an object is required, instead of initialising a new object, it is requested from the pool and activated. When it’s done, it is not destroyed, it's returned to the pool. Garbage collection spikes cause stuttering and are caused by creating and destroying many objects. If you pre-create all objects during a loading screen the user would not notice the stutter.

State

Can be used to program animation for characters. Each animation represents a different state that an object can be in, and only one state is currently active at any given time. This is essentially a finite state machine. The state pattern assists objects changing behaviour when its internal state changes. State specific behaviour is also defined independently so, adding additional state does not affect the existing states.

Observer

At runtime, many events can happen in a game, health could be decreased/increased, enemies or game objects could be killed/destroyed. The observer allows objects to notify others when an event occurs without closely coupling your classes, allowing communication without one-to-one dependencies. When an object changes state, all dependent objects can be notified automatically. Objects can broadcast events and other objects observe these events. An event is a notification that something has occurred and involves a publisher that creates the event and observers which have an event handler. When a publisher signals an event one or many observers can invoke a method which runs internal logic on the response.

**References:**

Krogh-Jacobsen, T. (2022). *Level up your code with game programming patterns*. Available at: https://resources.unity.com/games/level-up-your-code-with-game-programming-patterns [Accessed 20 July 2023].