# Team Lead 3 Presentation

Chadwick, Toby, Bob

## Presentation Outline

## Individual Requirements (by next Tuesday)

Initial Test Plan

Fully Automated:

- Unit tests of at least two different boundary tests for a single script

Can be automated or manual:

- A single stress test that breaks unity and records the breaking point as well as logs it in the console
- The stress test visually shows the stress on Unity
- The failure under stress can be implied (logs success until failure)

## What is Testing?

Testing is intended to show that a program does what it is meant to and to catch defects before release.

- Testing is executing a program with artificial data
- A part of a software validation and verification process

Testing can reveal the presence of errors but not their absence.

- Results can be checked for information on errors, and non-functional aspects of the program

## Why is testing Important?

- Allows us to find situations in which software operates incorrectly
- Enables us to validate that the software meets its requirements

Validation

- Shows that the system is operating as intended from design and implementation
- Operates correctly under a set of test conditions

## Types of Testing

There are several different types of software testing:

- System testing
    - Use-case testing
- Release testing
- User testing
    - Alpha
    - Beta
    - Acceptance
- Requirement based testing
- Performance testing
    - Stress testing

## System Testing

System testing is the testing of a fully integrated and complete piece of software.

Use -case testing is a basis for system testing.

- Use cases are used to identify the interactions of the system
- These interactions between system components are then tested

## Release Testing

Release testing is a form of system testing.

There are some key differences between the two:

- A separate team not involved in development is responsible for release testing
- System testing should be focused on discovering bugs
- Release testing  should check whether a system meets its requirements and is ready for validation testing

## User Testing

User testing is a type of testing in which the customers/users provide input on system testing.

It is essential to conduct even after release and system testing because the user's environment can't be reproduced in a testing environment.

Alpha

- Users work with the dev team to test the software at the developer's site

Beta

- A release of the software is made available to users to allow them to experiment and find problems with the system

Acceptance

- Customers test a system to decide whether or not it is ready to be accepted from the developers

## Requirement Based Testing

- Involves examining each requirement of a system and designing a specific test (or tests) for that requirement

## Performance Testing

- Performance tests typically involve tests that incrementally increases the load on a system until the performance of that system reaches an unacceptable point

Stress Testing

- Stress testing is a form of performance testing where the system is purposefully overloaded to see how it would react in a failure scenario
- Seeks to test and analyze the failure behavior of the system

## Create Loosely Coupled Code

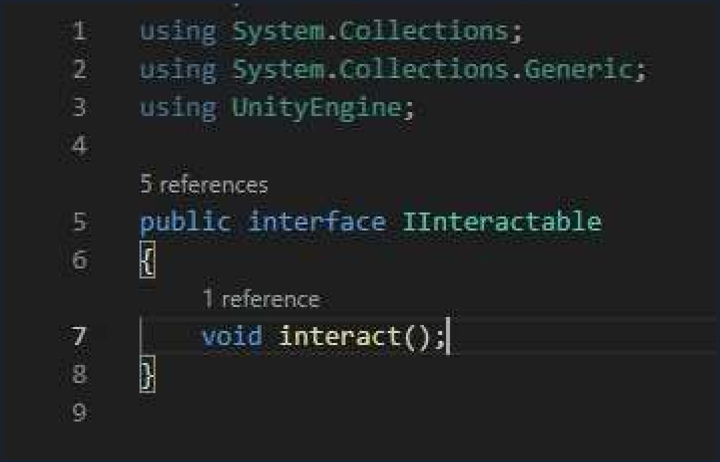
- Code is easier to understand
- Makes program more modular
- Program is easier to change, update, and expand
- More testable code

## One Function, One Function

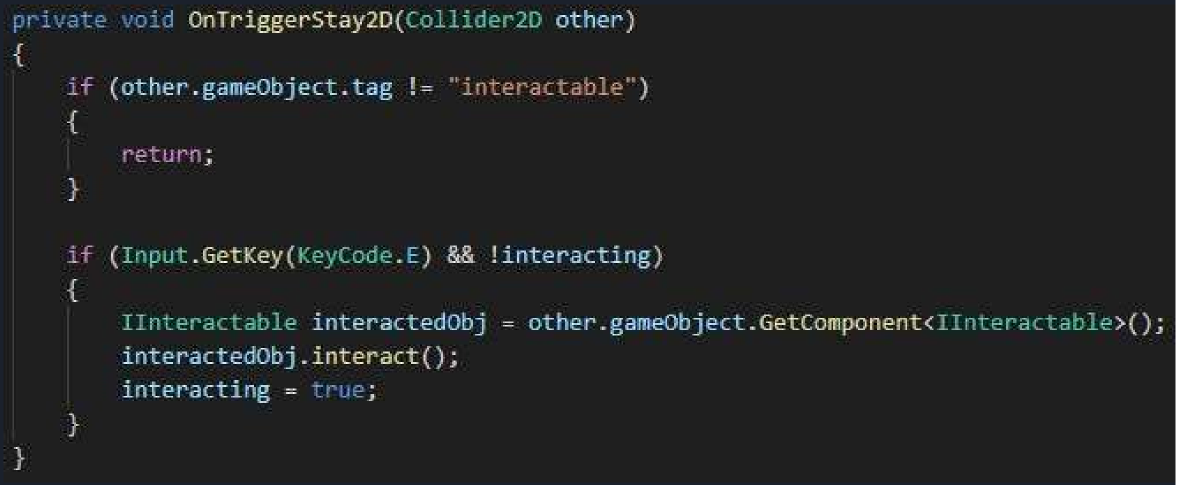
- One function should have only one function
- Functions should be non-deterministic
- Single Responsibility Principle
    - A function should either produce or process information, not both.
- If a function needs extra data, have it passed as a parameter
- Inversion of Control
    - Separate decision making code and action code

# Interfaces

- Very helpful in keeping code loosely coupled



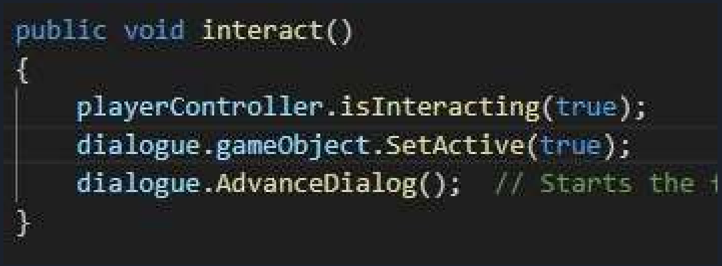
```csharp
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

5 references
public interface IInteractable
{
    1 reference
    void interact();
}

```



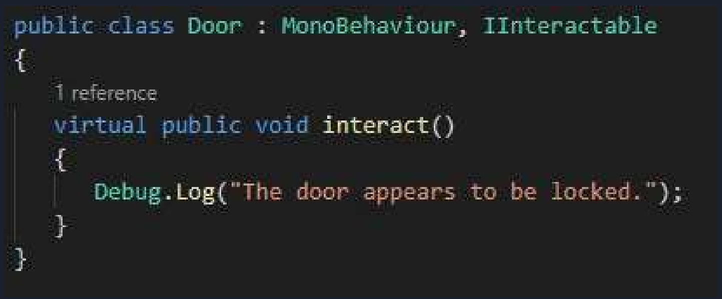
```csharp
private void OnTriggerStay2D(Collider2D other)
{
    if (other.gameObject.tag != "interactable")
    {
        return;
    }

    if (Input.GetKey(KeyCode.E) && !interacting)
    {
        IInteractable interactedObj = other.gameObject.GetComponent<IInteractable>();
        interactedObj.interact();
        interacting = true;
    }
}
```

## Interfaces

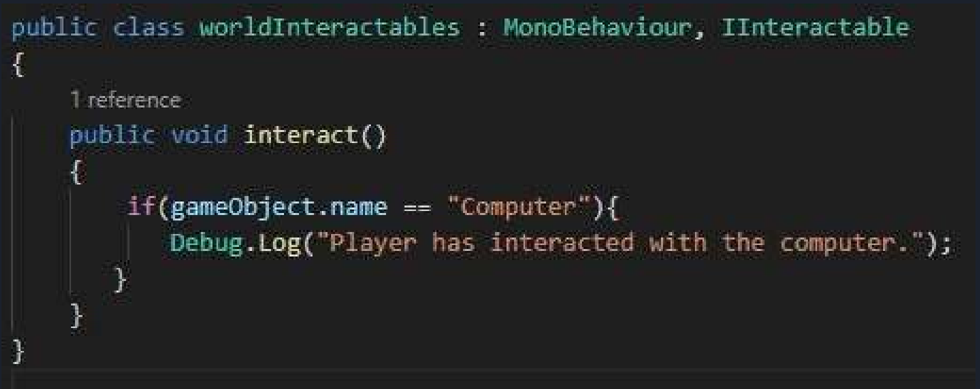
- A single function call can have many implementations



```
public void interact()
{
    playerController.isInteracting(true);
    dialogue.gameObject.SetActive(true);
    dialogue.AdvanceDialog();  // Starts the
}
```



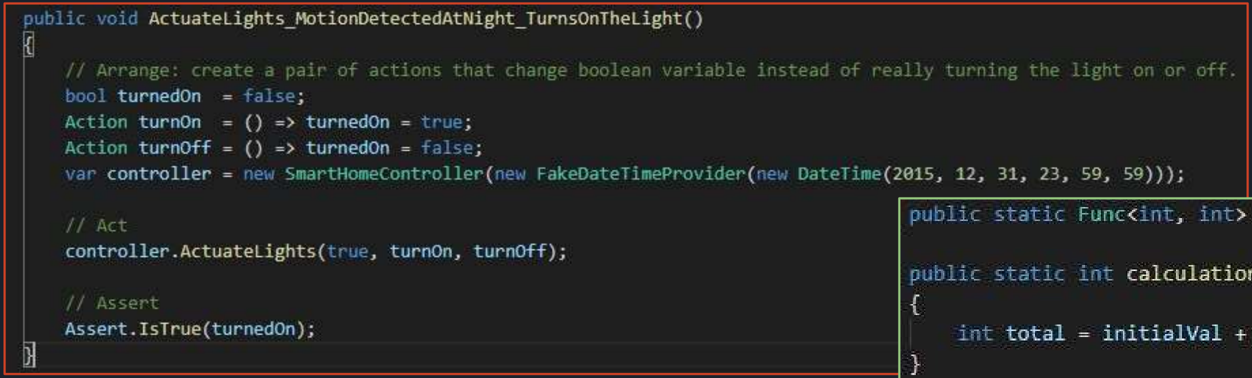
```
public class Door : MonoBehaviour, IInteractable
{
   1 reference
   virtual public void interact()
   {
      Debug.Log("The door appears to be locked.");
   }
}
```



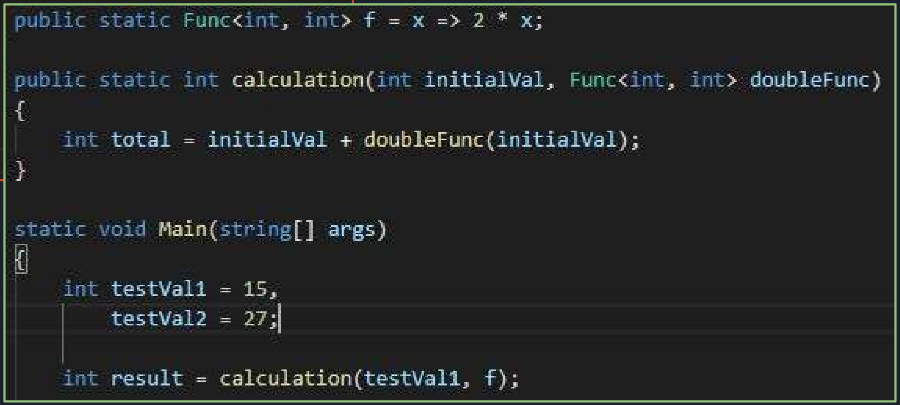
```
public class worldInteractables : MonoBehaviour, IInteractable
{
    1 reference
    public void interact()
    {
        if(gameObject.name == "Computer"){
            Debug.Log("Player has interacted with the computer.");
        }
    }
}
```

## Higher Order Functions

- Functions as arguments or return values of other functions



```
public void ActuateLights_MotionDetectedAtNight_TurnsOnTheLight()
{
    // Arrange: create a pair of actions that change boolean variable instead of really turning the light on or off.
    bool turnedOn  = false;
    Action turnOn  = () => turnedOn = true;
    Action turnOff = () => turnedOn = false;
    var controller = new SmartHomeController(new FakeDateTimeProvider(new DateTime(2015, 12, 31, 23, 59, 59)));

    // Act
    controller.ActuateLights(true, turnOn, turnOff);

    // Assert
    Assert.IsTrue(turnedOn);
}
```



```
public static Func<int, int> f = x => 2 * x;

public static int calculation(int initialVal, Func<int, int> doubleFunc)
{
    int total = initialVal + doubleFunc(initialVal);
}

static void Main(string[] args)
{
    int testVal1 = 15,
        testVal2 = 27;

    int result = calculation(testVal1, f);
```

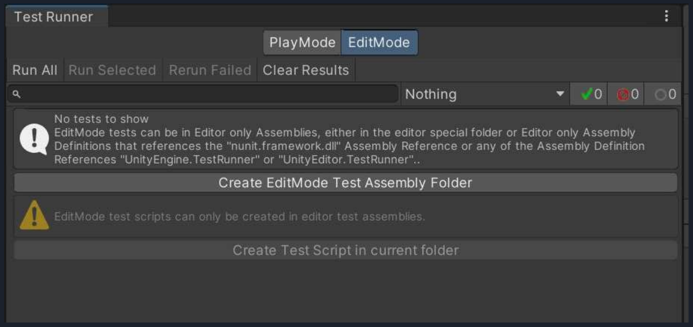
## Edit Mode Vs. Play Mode tests

Edit mode:

- Tests code that doesn’t require a running scene to test
- More useful for calculation
- Much faster to run

Play mode:

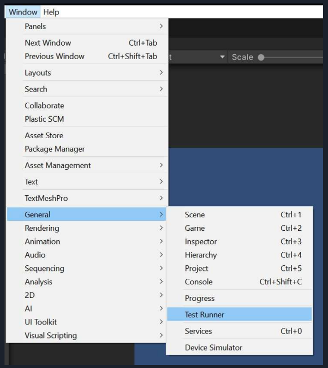
- Tests code that needs to be executed in a running scene
- Tests are ran as coroutines
- More useful for testing things like movement

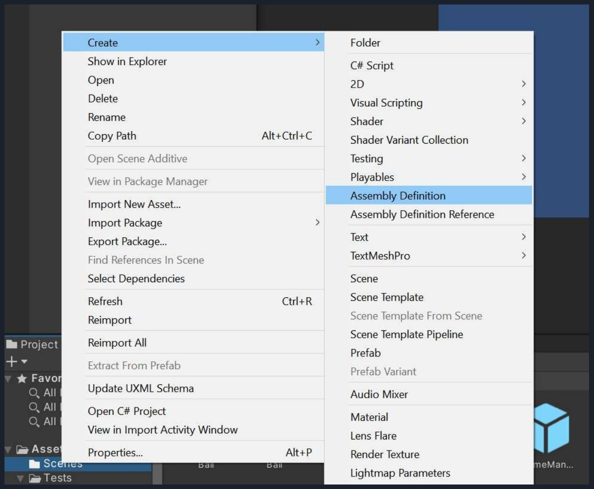


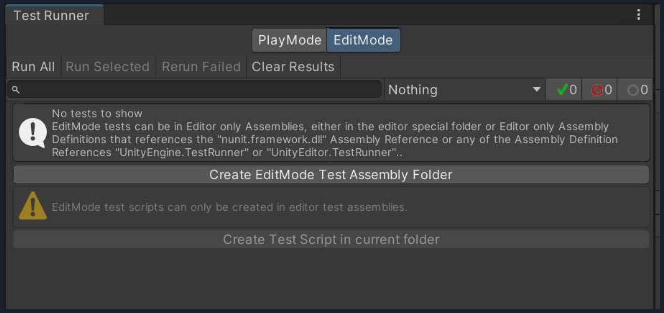
No tests to show
EditMode tests can be in Editor only Assemblies, either in the editor special folder or Editor only Assembly Definitions that references the "nunit.framework.dll" Assembly Reference or any of the Assembly Definition References "UnityEngine.TestRunner" or "UnityEditor.TestRunner"..

EditMode test scripts can only be created in editor test assemblies.

# Setting up Testing (Refer to the how to document)





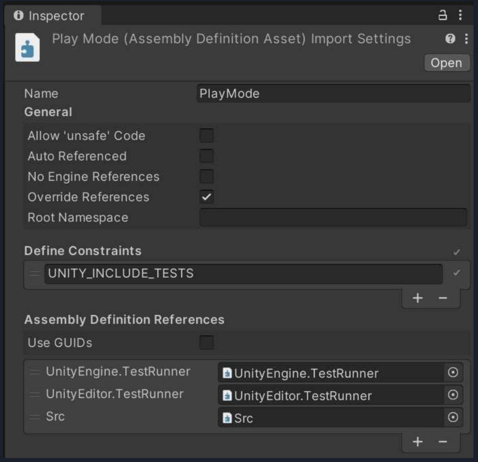


No tests to show
EditMode tests can be in Editor only Assemblies, either in the editor special folder or Editor only Assembly Definitions that references the "nunit.framework.dll" Assembly Reference or any of the Assembly Definition References "UnityEngine.TestRunner" or "UnityEditor.TestRunner"..

Create EditMode Test Assembly Folder

EditMode test scripts can only be created in editor test assemblies.

Create Test Script in current folder



Play Mode (Assembly Definition Asset) Import Settings

Name PlayMode

**General**

Allow 'unsafe' Code
Auto Referenced
No Engine References
Override References ✓
Root Namespace

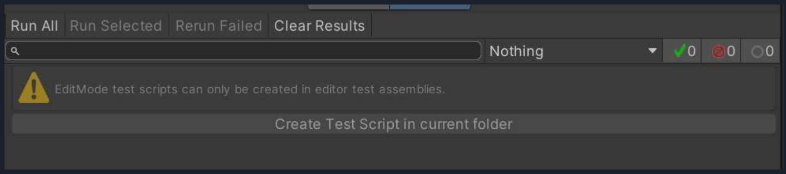
**Define Constraints**

UNITY_INCLUDE_TESTS

**Assembly Definition References**

Use GUIDs

UnityEngine.TestRunner — UnityEngine.TestRunner
UnityEditor.TestRunner — UnityEditor.TestRunner
Src — Src



EditMode test scripts can only be created in editor test assemblies.

Create Test Script in current folder

## Boundary Tests

What are boundary tests?

- A boundary test is any sort of test that checks whether a value is within some specified range
- These tests can check for if a value is within, on, or outside of a boundary

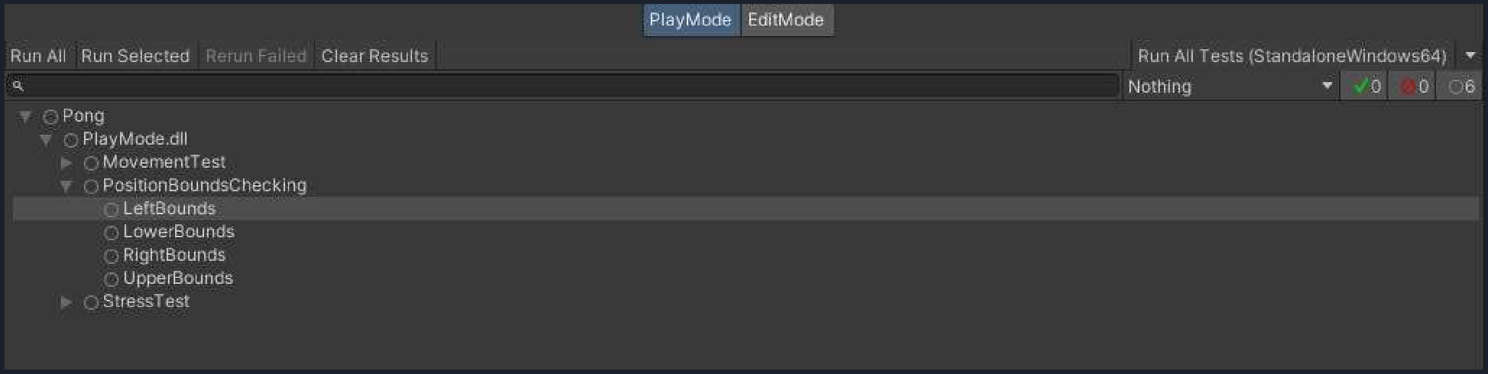
## Creating Unit Boundary Tests

- 3 main areas of a unit test
  - Arrange
  - Act
  - Assert

- A boundary test is a specific type of unit test
- Tests to verify that edge cases are properly handled
- Verify that unexpected behaviour doesn’t occur if given unexpected values

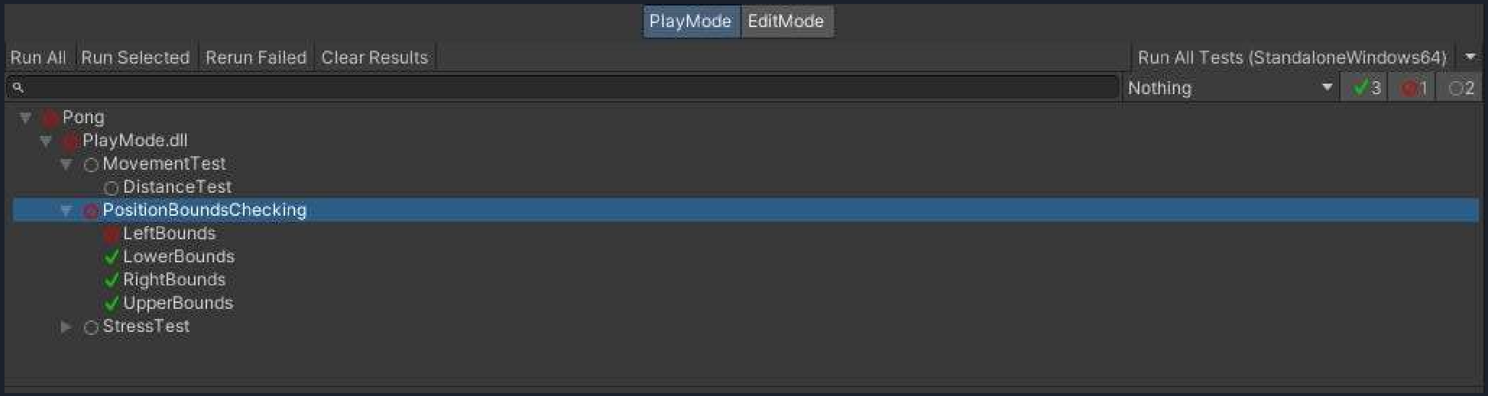
- Tests just inside the boundary
- Tests on the boundary
- Tests just outside the boundary

## Executing tests



PlayMode | EditMode
Run All | Run Selected | Rerun Failed | Clear Results | Run All Tests (StandaloneWindows64)
Nothing | ✓0 | ●0 | ○6

```
▼ ○ Pong
  ▼ ○ PlayMode.dll
    ▶ ○ MovementTest
    ▼ ○ PositionBoundsChecking
        ○ LeftBounds
        ○ LowerBounds
        ○ RightBounds
        ○ UpperBounds
    ▶ ○ StressTest
```



PlayMode | EditMode
Run All | Run Selected | Rerun Failed | Clear Results | Run All Tests (StandaloneWindows64)
Nothing | ✓3 | ●1 | ○2

```
▼ ● Pong
  ▼ ● PlayMode.dll
    ▼ ○ MovementTest
        ○ DistanceTest
    ▼ ● PositionBoundsChecking
        ● LeftBounds
        ✓ LowerBounds
        ✓ RightBounds
        ✓ UpperBounds
    ▶ ○ StressTest
```

## Stress Tests

What is a stress test?

- A stress test is a test that applies stress to the system in an incremental manner until something breaks
- The breaking point is recorded and is the measured limit of the software
- Examples of failure  under stress tests:
    - Failure to detect physical collision
    - Significant drop in frame rate
- Doesn’t have to be automated
- Will be different depending on the system

## Patterns

## What is a pattern

- Used to carry out the same functionality as other code
- Intended to make code easier to understand and maintain, as well as reduce coupling
- Since they see such widespread use, chances are any issues will be discovered before you start using them

## Creational Patterns

## Factory

- The factory is one of the simpler patterns. The goal here is to reduce coupling when instantiating new objects.
- Example: Eggs

## Abstract Factory

- Factory that creates factories
- Useful when we have whatever set of things we want to create in a factory but want some of them need to be in their own distinct groups.
- Example: Tile Hero Enemy Groups

## Builder

- Simplifies a process where there are multiple steps each having different possible pieces
- Example: Tile Hero Random Tile

## Resource Pool

- Instantiate a reasonable amount of resource intense objects towards the start, then loan them to different objects/processes as needed.
- Reduces work on the system so that it doesn’t have to re create the resource intense objects repeatedly.
- If the loaned amount gets low, it should seek for the opportune time to instantiate new objects, though implementation of this is not required.

## Prototype

- Create one version of the object only to be copied, not used
- Unity’s prefab feature handles this for us
- To use this as your pattern you would need to create your own code and just use prefabs

## Structural Patterns

## Adapter

- Take a class that's incompatible with another class or function and encase it in an adapter to make it work as if it were.
- Example: Coordinates

## Composite

- Composites apply to tree like structures made up of primitive elements and composite elements
- In the addition tree, addition operators are Composites, they're made up of two components and are handled differently than the single integers, which are primitives
- Example: Menus

## Facade

- The façade takes a complicated system and provides a simple interface for the user
- Example: Shop

## Proxy

- Hide the real object behind a proxy object that either communicates with the real object or will allow the real object out once a condition is met.
- Example: Bats
- Example: Statues

## Behavioral Patterns

## Command

- Separate the command from the issuer and the receiver
- Allows us to manipulate the command freely as an object
- Example: High Budget Dungeon Jump

## Iterator

- Allows a client to access what they need from a data structure without understanding the data structure.
- Example: Person with a messy house

## Mediator

- Decouples interactions between two entities and allows easier many to many communication
- Example: Combat Manager

## Memento

- A memento is an object which stores the state of an object when it is created and will return the object to that state when returned.
- The memento cannot do anything else, so it must have another object with it in order for it to return to the originator.
- Example: Save System
- Example: Restore State Item

## State

- Used to facilitate a state machine style object which switches between distinct classes
- Must have a separate object which notifies the current state to change, and the current state must encapsulate different classes depending on its state
- Example: Lightswitch

## Template Method

- Create a general framework for several different classes with minor variances
- Differences in behavior between different subclasses can be accounted for by overriding certain functions
- Example: Firefighter and Post Worker

## Strategy

General Approach applicable to an interface which can be adjusted to fit a specific implementation

Ex: Transportation

Ex: Screen saver

## Null Object

- This is an object which does nothing
- Intended to be used when some object is required but no action is wanted
- Example: Strategy/Template

## Other Patterns

- Private Data Class
- Flyweight
- Bridge
- Chain of Responsibility
- Interpreter
- Visitor