



Team Lead 3 Presentation

Ankeet, Davin, Gavin, Joe, Luke, Mikayla, Owen,
Sam



Presentation Outline

Patterns

- What is a pattern
- Creational, Structural, and Behavioral Patterns

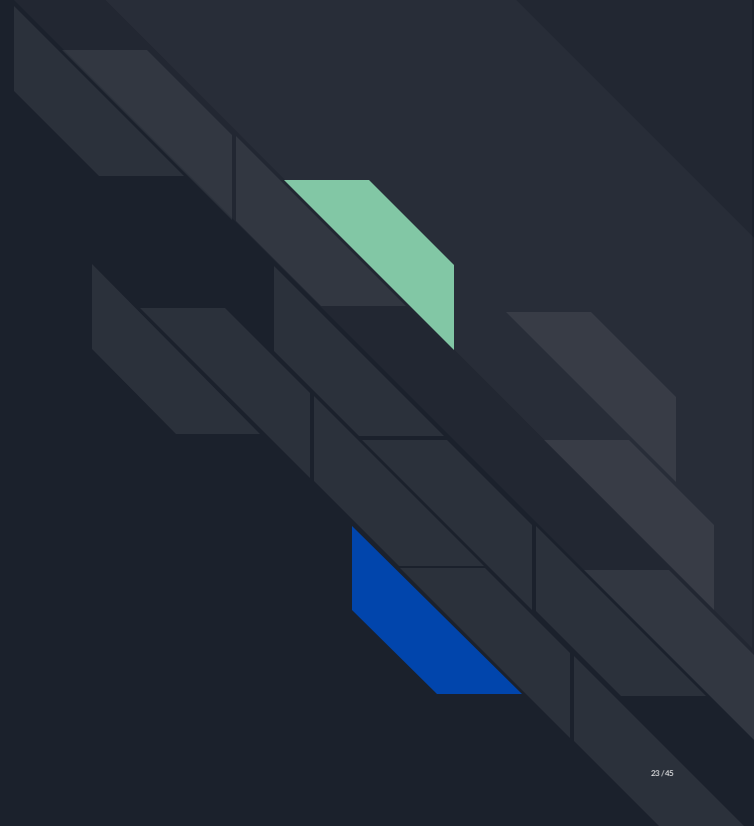
Testing Introduction

- What is testing
- Why is testing important
- Types of testing

Unit and Stress Testing

- Writing testable code (loose coupling)
- Edit mode Vs. Play mode tests
- Setting up testing (assemblies)
- Boundary tests and creating them
- Executing tests
- Stress testing

Patterns

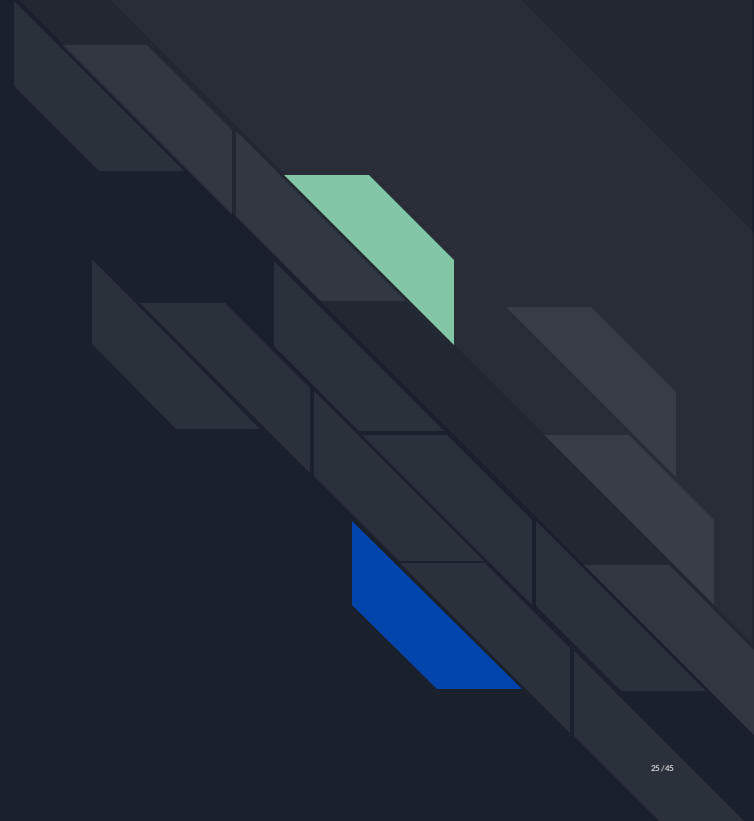




What is a pattern

- Intended to make code easier to understand and maintain, as well as reduce coupling
- **Design patterns** are typical solutions to commonly occurring problems in software design. They are like pre-made blueprints that you can customize to solve a recurring design problem in your code.

Creational Patterns

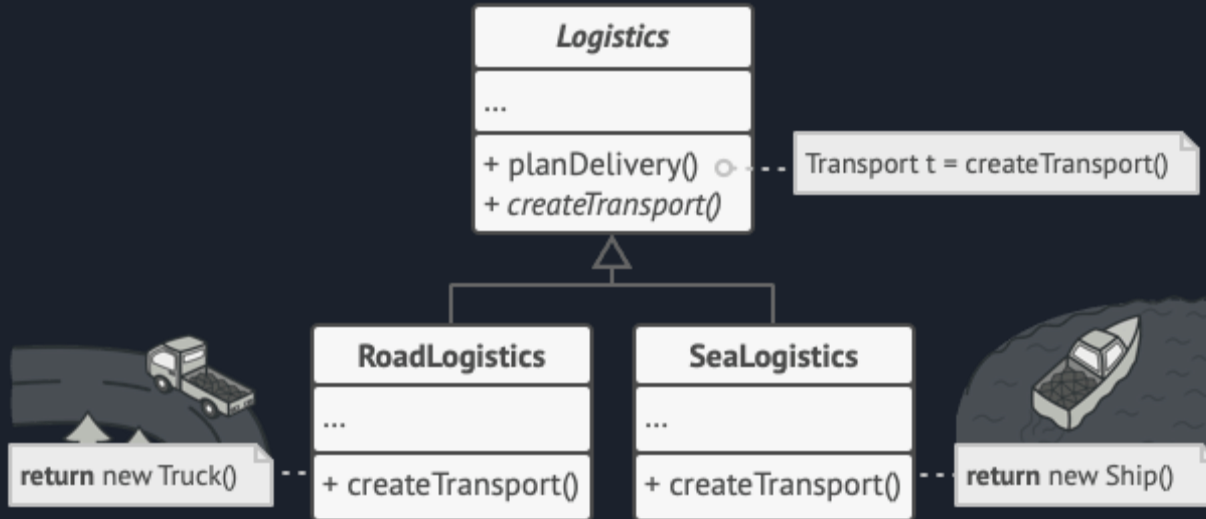




Factory

- Factory is one of the simpler patterns.
- It provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

Factory Example

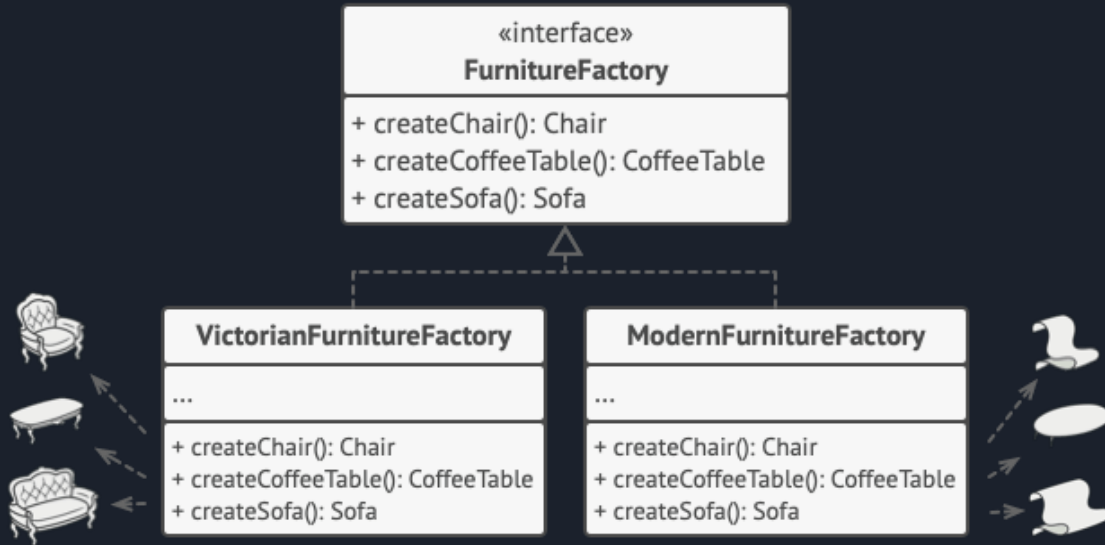




Abstract Factory

- Pattern that lets you produce families of related objects without specifying their concrete classes.
- Useful when we have a set of things we want to create in a factory but some of them need to be in their own distinct groups.

Abstract Factory



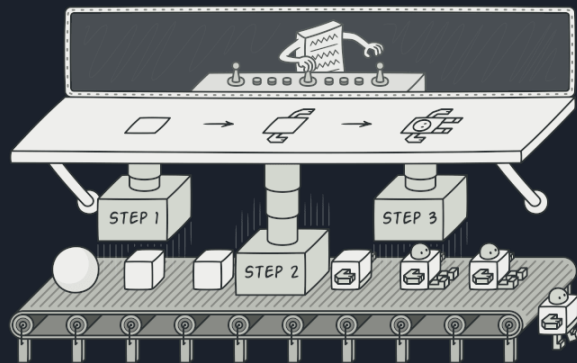
- createChair
- createSofa
- createCoffeeTable

These methods return **abstract** product types represented by: Chair, Sofa, CoffeeTable

Builder

- Simplifies a process where there are multiple steps each having different possible pieces

```
class HotDog {  
    constructor( ...  
    ) {}  
  
    addKetchup() {  
        this.ketchup = true;  
    }  
    addMustard() {  
        this.mustard = true;  
    }  
}
```





Resource Pool

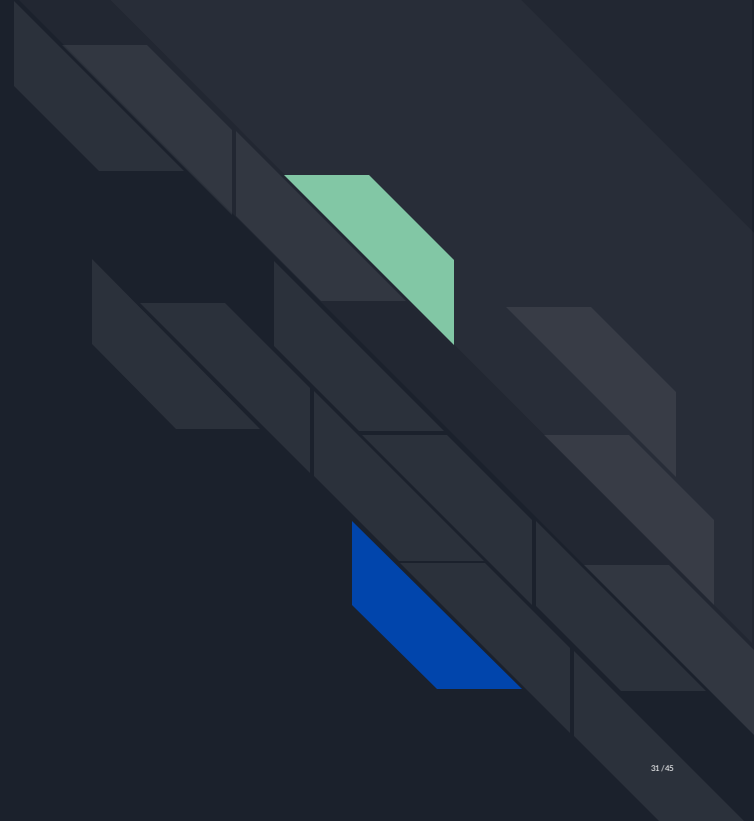
- Instantiate a reasonable amount of resource intense objects towards the start, then loan them to different objects/processes as needed.
- Reduces work on the system so that it doesn't have to re create the resource intense objects repeatedly.
- Example: Creating Bullets

Prototype

- The Prototype design pattern is all about cloning. Imagine you have an object, and instead of creating new instances of this object from scratch every time, you want to create copies of the existing one.
- Unity's prefab feature handles this for us



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.

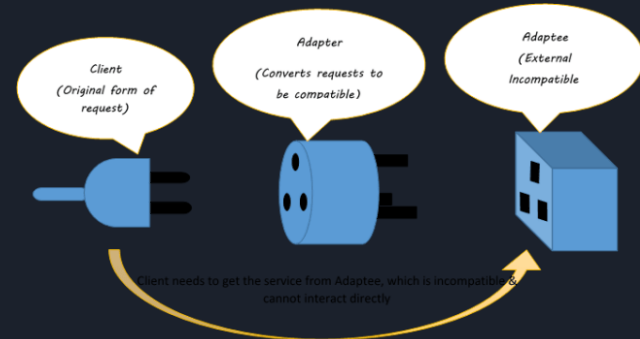


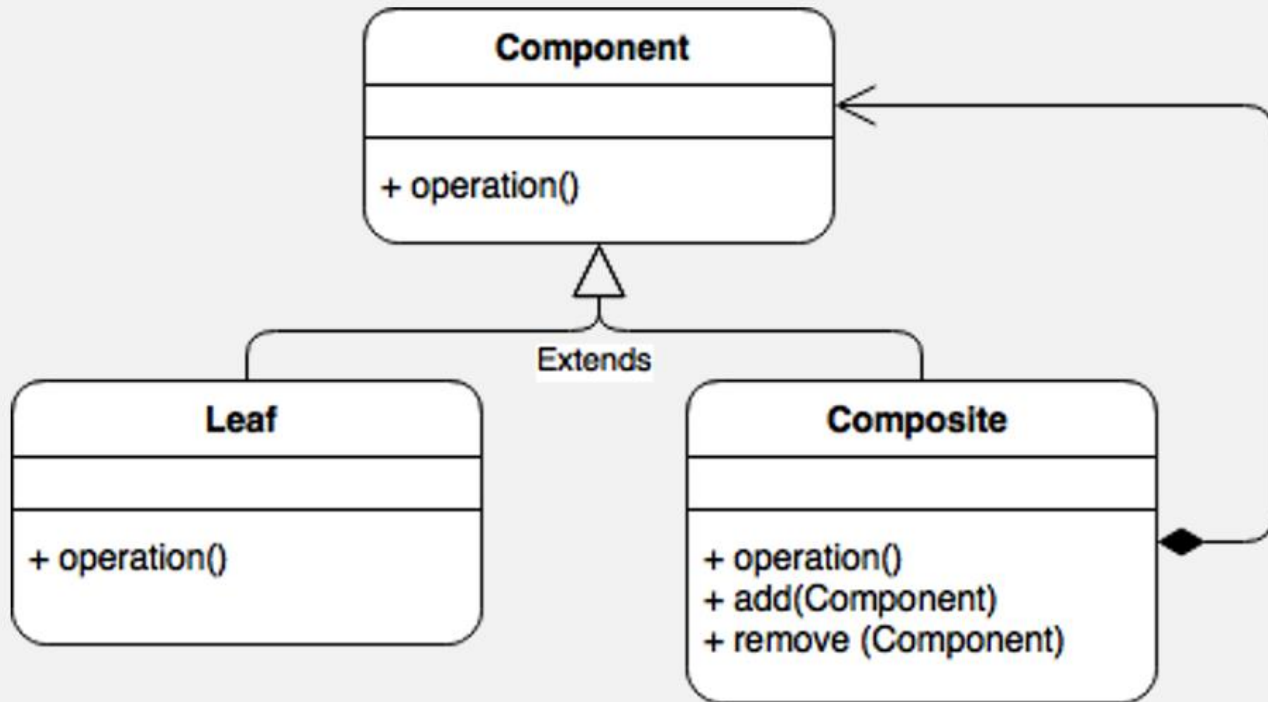
Figure 1-Adapter Pattern Concept



Composite

- Composites apply to tree like structures
- They **describes a group of objects that are treated the same way as a single instance of the same type of object.**


Composite pattern – Class diagram





Composite Example

- **Component:** An abstract class/interface `IEnergy` with a method `Attack()`.
- **Leaf:** A class `SimpleEnemy` implementing `IEnergy`.
- **Composite:** A class `EnemyGroup` that can hold and manage multiple `IEnergy` objects.



```
public interface IEnemy
{
    void Attack();
}
```

```
public class SimpleEnemy : IEnemy
{
    public void Attack()
    {
        // Code for a single enemy attack
    }
}
```

```
public class EnemyGroup : IEnemy
{
    private List<IEnemy> enemies = new List<IEnemy>();

    public void Add(IEnemy enemy)
    {
        enemies.Add(enemy);
    }

    public void Remove(IEnemy enemy)
    {
        enemies.Remove(enemy);
    }

    public void Attack()
    {
        foreach (IEnemy enemy in enemies)
        {
            enemy.Attack();
        }
    }
}
```



Facade

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



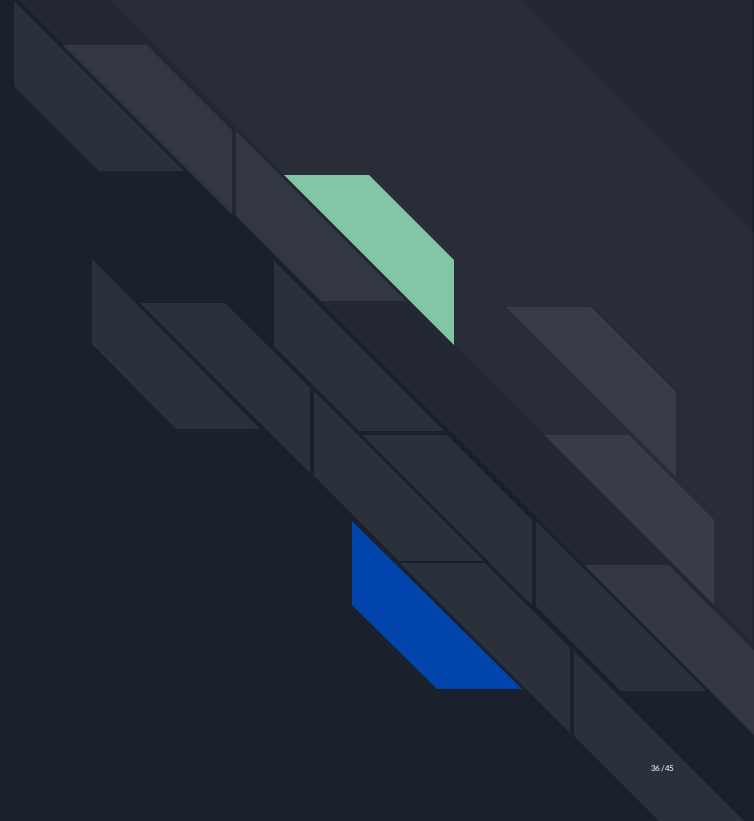
Proxy

- 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.

Applicability: Lazy initialization. If you have a heavyweight service object that wastes system resources by being always up, even though you only need it from time to time.

- Example: NPC class with NPCProxy class that grants access to NPC class

Behavioral Patterns





Command

- Separate the command from the issuer and the receiver
- Allows us to manipulate the command freely as an object
- Example: Waiter taking orders in a restaurant



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: Vending machine behavior depends on it's current state



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 daily routine



Strategy

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

Ex: Transportation methods



Null Object

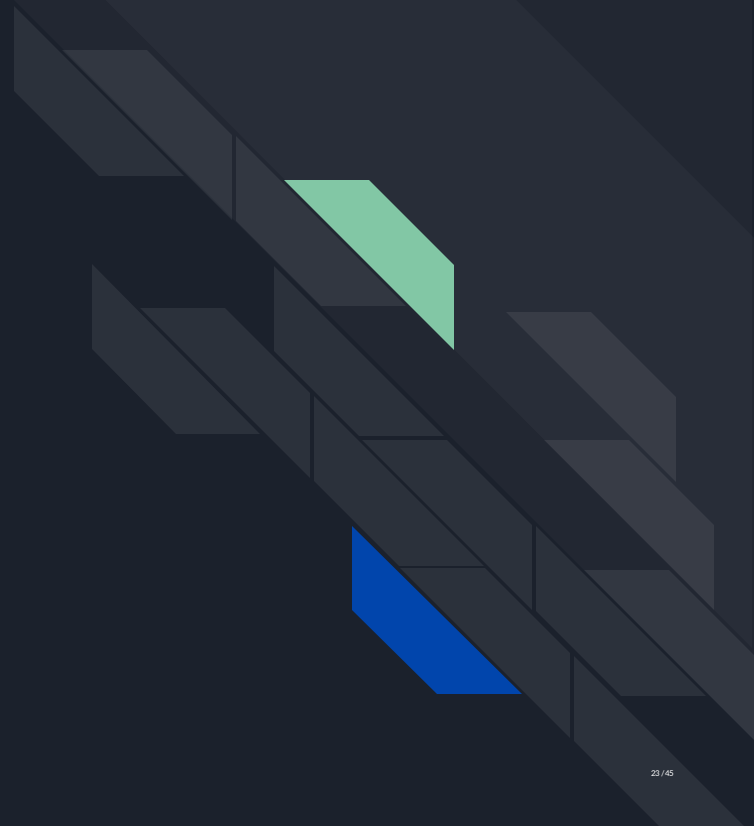
- This is an object which does nothing
- Intended to be used when some object is required but no action is wanted
- Example: Customer support for registered vs unregistered users



Other Patterns

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

Patterns





Individual Requirements

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 specific needs.



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

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 references
6 public interface IInteractable
7 {
8     void interact();
9 }
```

```
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
- A contract

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

```
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
- Returns functions as result

```
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);
}
```

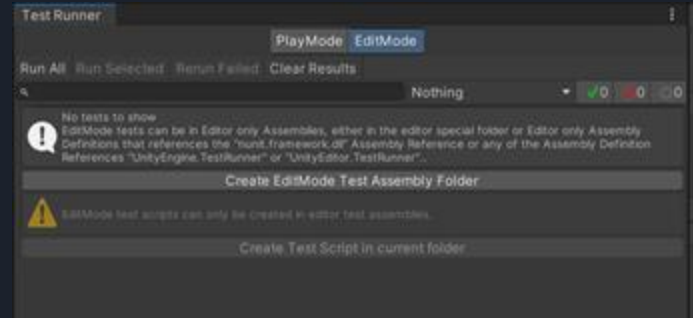
Edit Mode Vs. Play Mode tests

Edit mode:

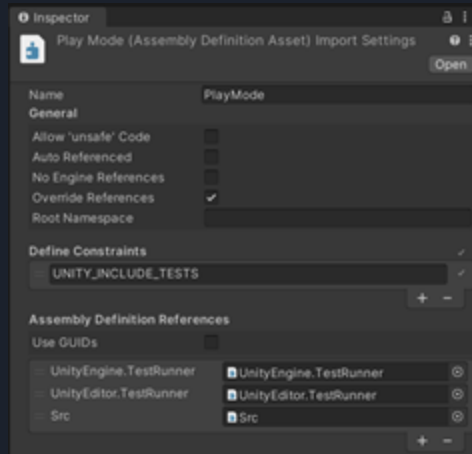
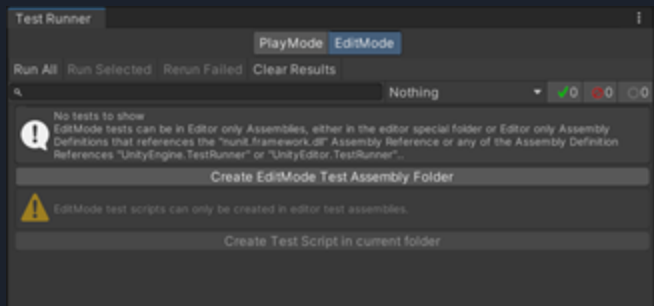
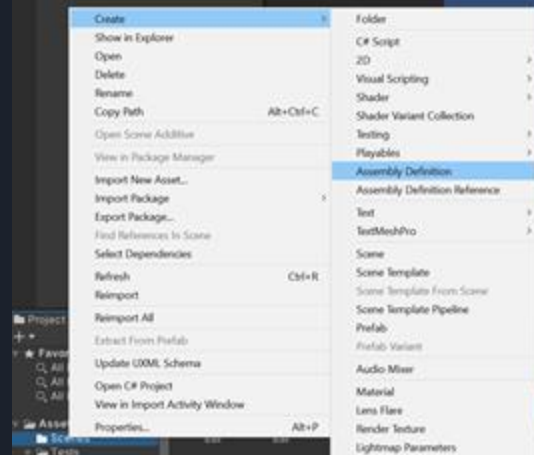
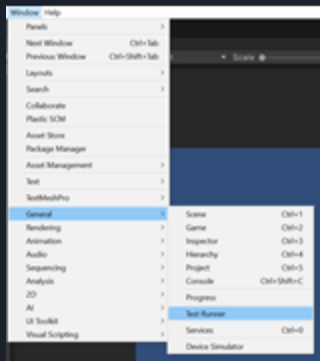
- Tests code that doesn't require a running scene to test
- More useful for calculation
- Much faster to run

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



Setting up Testing (Refer to the how to document)





Boundary Tests

What are boundary tests?

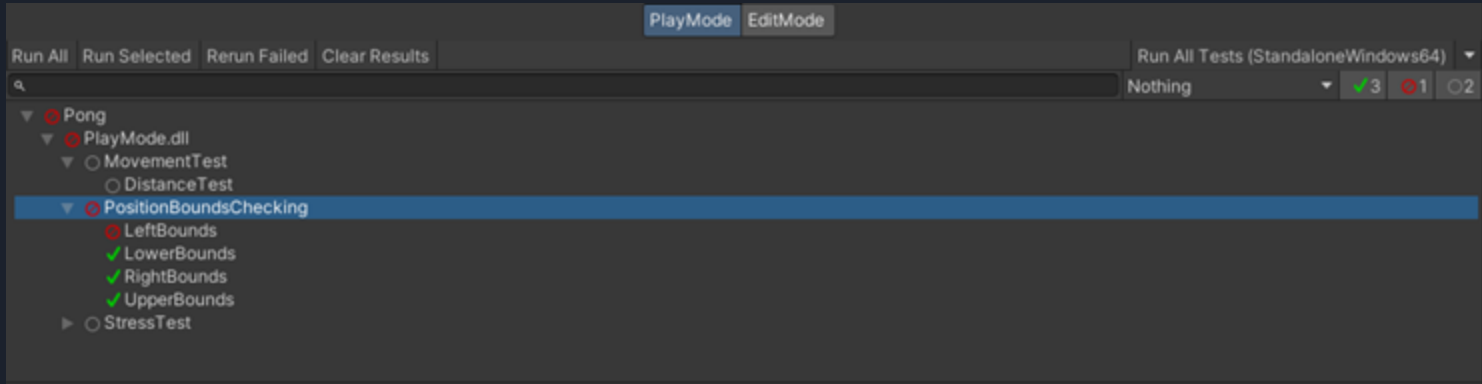
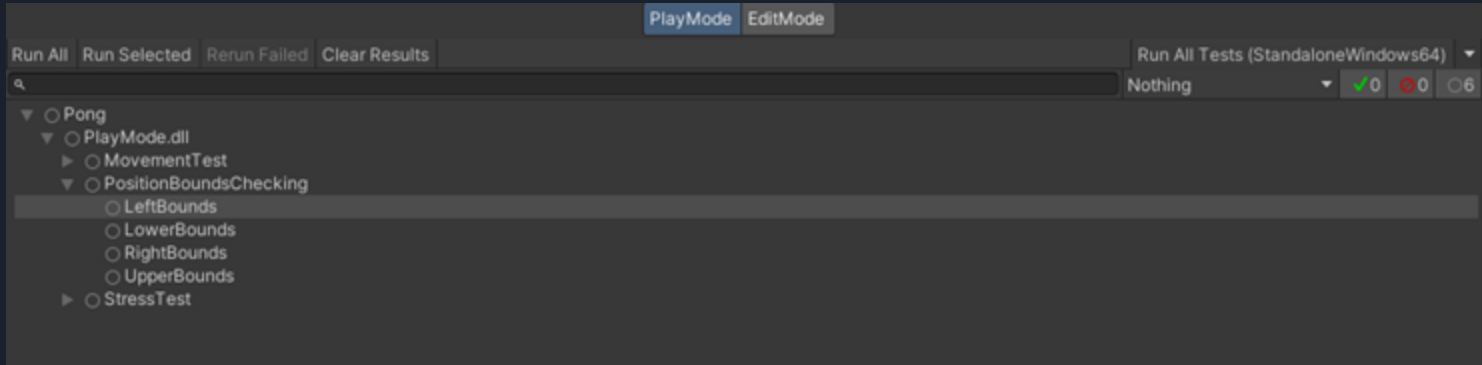
- A boundary test is any sort of test that checks whether a value is within some specified range
- These tests can check for if a value is within, on, or outside of a boundary
- Ex: Test whether a `gainHealth` function brings the health of the player to over 100



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



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





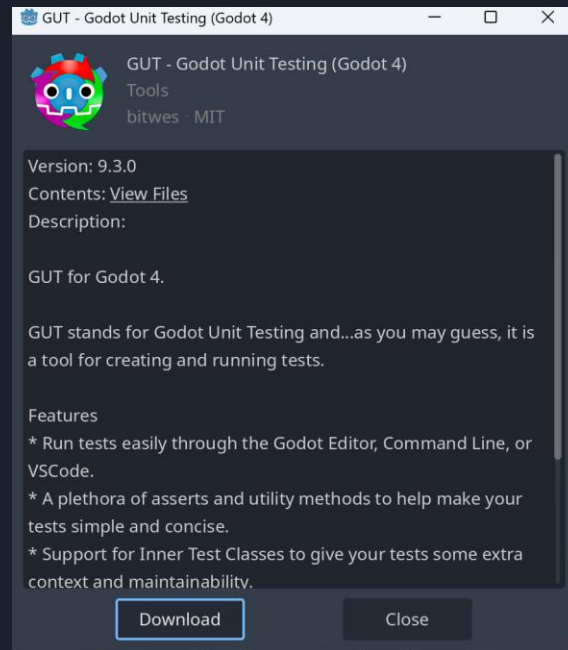
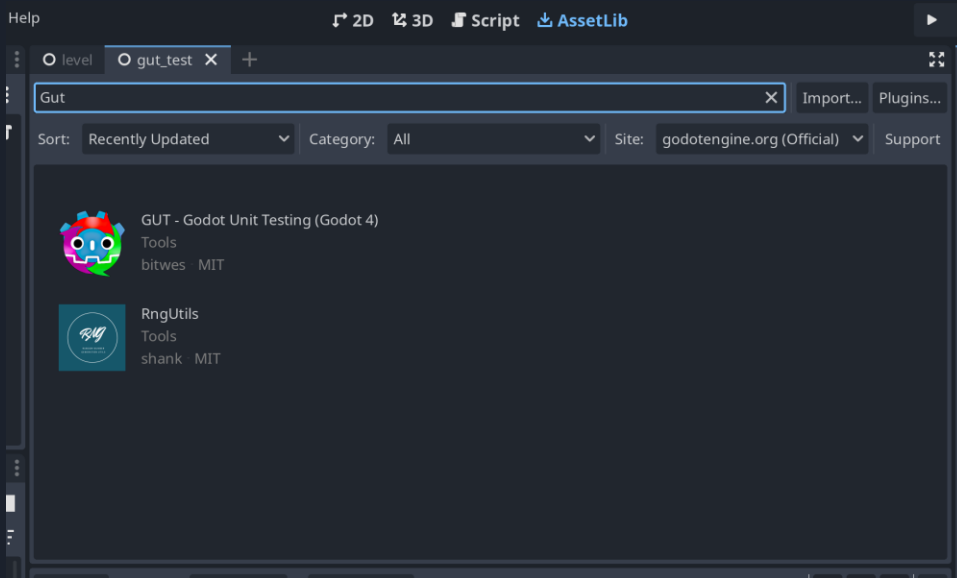
Godot Testing

- Godot does not come with a built-in testing framework by default.
- We need to download and integrate GUT (Godot Unit Testing), which is a popular third-party plugin for unit testing in Godot projects.
- GUT provides comprehensive tools for creating and running tests in GDScript.
- It supports **TDD (Test Driven Development)**, allowing you to write tests for GDScript code before implementing functionality.
- Supports various test types: **unit tests**, **integration tests**, and **UI tests**, helping ensure game functionality and stability.

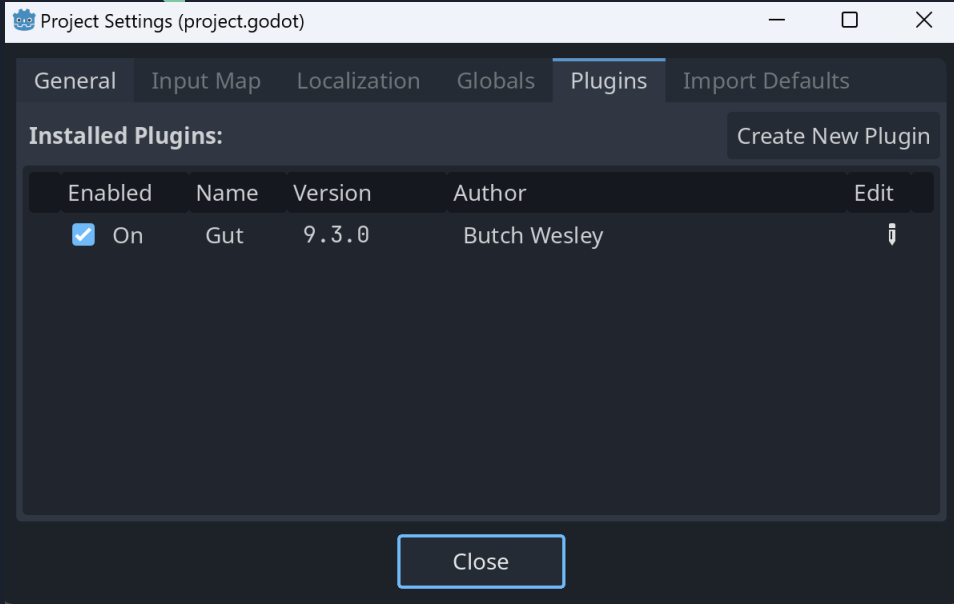
Let's see how we can install and run tests

Install Gut

Open the Godot project and click on the "AssetLib" tab. Search for "GUT" to easily find the Godot Unit Testing framework.



Activation of the GUT



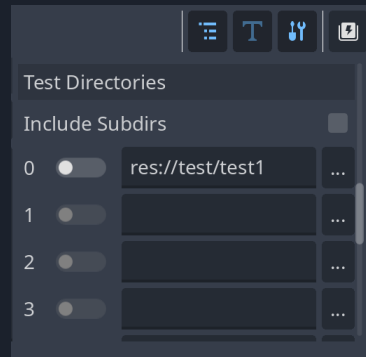
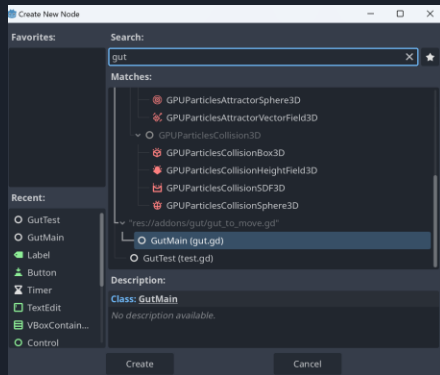
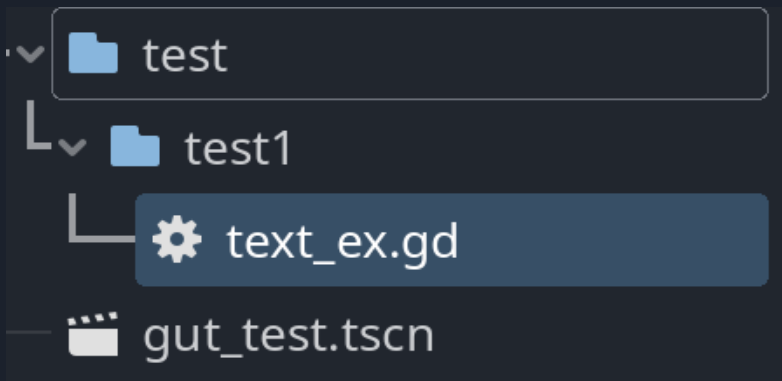
Go to "Project Settings" and navigate to the "Plugins" tab.

The GUT appears alongside the Output, Debugger, and Animation tabs at the bottom of the Godot engine interface.



Running the Test

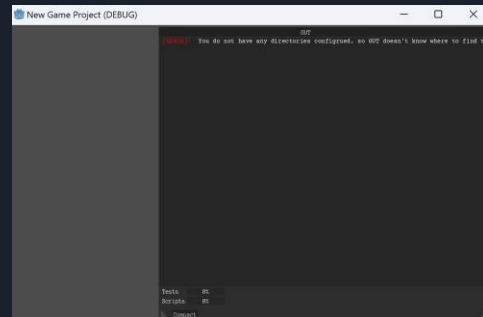
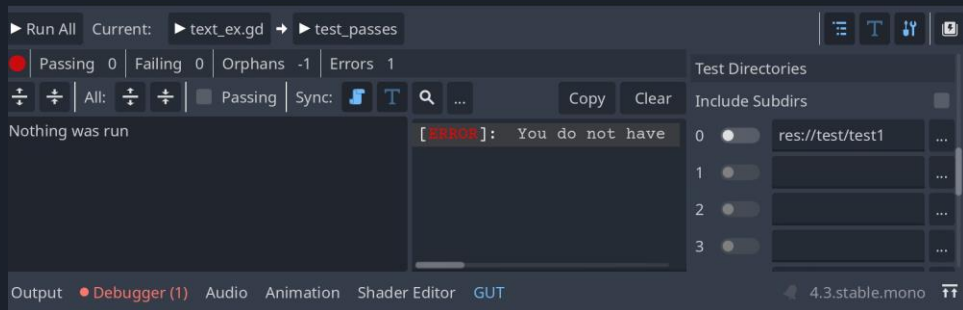
- Create a tests/ folder: Right-click in the FileSystem dock, select "New Folder", and name it tests.
- Add a Node: Create a new scene, add a Node, and rename it to TestRunner.
- We need to guide our runner through the test Directories. We must ensure they know which directories contain the tests



Running the Test

- Create test scripts: Inside the tests folder, choose "New Script", save it with a relevant name like test_player.gd, and ensure it inherits from GutTest.
- Attach a script: Attach a new script to TestRunner to execute tests
- This setup initializes GUT, directs it to the tests folder, and runs all the tests within that directory, displaying results in the Godot output console.

```
1  extends GutTest
2
3  func test_player_initial_health():
4      var player = Player.new()
5      assert_eq(player.health, 100, "Player should start with 100")
6
```



Unity/Godot Testing Demo