QUALITY ASSURANCE

By Gary, Karina, Jordan, Nathan, & Tryston

OUTLINE

1. Testing Background
   - What is testing?
   - Why test?
   - Types of Testing
2. Unit Boundary & Stress Testing
   - Unity Test Framework
   - Stress Test Demonstration
3. Design Patterns
   - What are design Patterns?
   - Types of Patterns
     - Pattern Methods
   - Examples
4. Deliverables

TESTING BACKGROUND

An introduction to testing motivations & various test types

WHAT IS TESTING?

Testing is defined by:

- Showing a program functions as intended & identifying defects prior to use
- Executing a program with artificial data
- Uncovering the existence of errors, not absence of
- Belonging to the general verification & validation process

WHY TEST?

- Ensure program handles expected & unexpected inputs
- Define load constraints to stay within
- Verify functioning of code before pushing to GitHub
- Identify scenarios where software behaviour is incorrect, undesirable, or non-conforming to specification
- Demonstrate to customer & developers software satisfies requirements

WHY TEST?

- Expected to contribute 1 stress test & 2 unit boundary tests by next Tuesday (10 marks)
- Expected to provide complete test plan by Oral Exam

VERIFICATION VS. VALIDATION

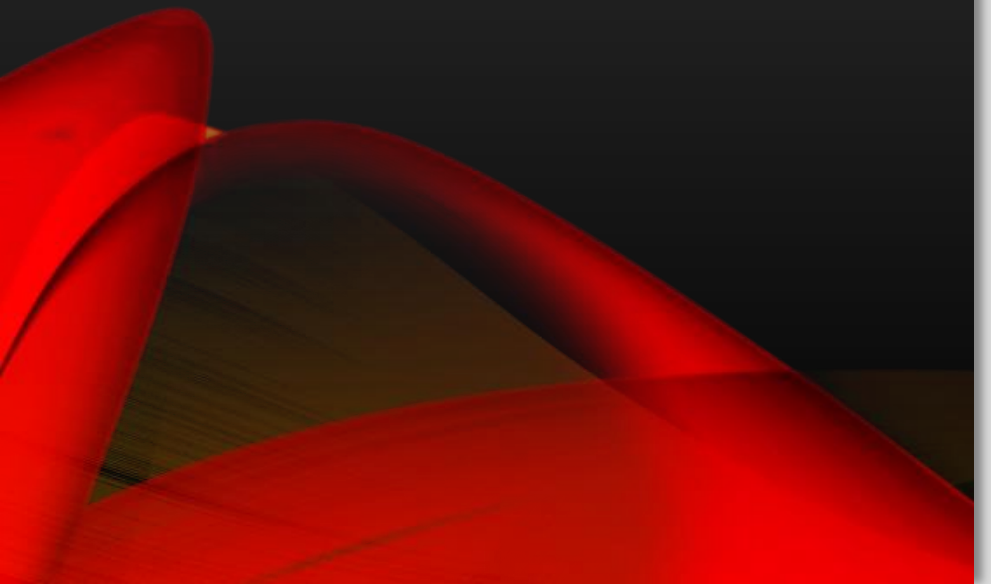
Verification: software conforms to specifications

"Are we building the product right?"

Validation: software does what user needs it to

"Are we building the right product?"

TYPES OF TESTING



Testing is divided into several categories based on goals and characteristics.

- System Testing
  - Use-case Testing
  - Release Testing
- User Testing
  - Alpha Testing
  - Beta Testing
  - Acceptance Testing
- Requirements-based Testing
- Unit Testing
  - Boundary Testing
- Performance Testing
  - Stress Testing

SYSTEM TESTING

System Testing: testing on a complete, fully-integrated software product to identify defects

Types (of Interest):

- Use-case Testing
- Release Testing

USE-CASE TESTING

Use-case Testing: testing developed to identify system interactions by simulating typical use

Characteristics:

- Involves multiple system components
- Forces interactions to occur
- Informs system testing test cases
- Sequence diagrams document components & interactions

RELEASE TESTING

Release Testing: testing on a specific release of a system to validate it satisfies the requirements to be released for external use

Characteristics:

- Performed by team external to the system development
- Usually black-box testing process
- Demonstrate dependability, functionality, & performance under normal conditions

-Switch

USER TESTING

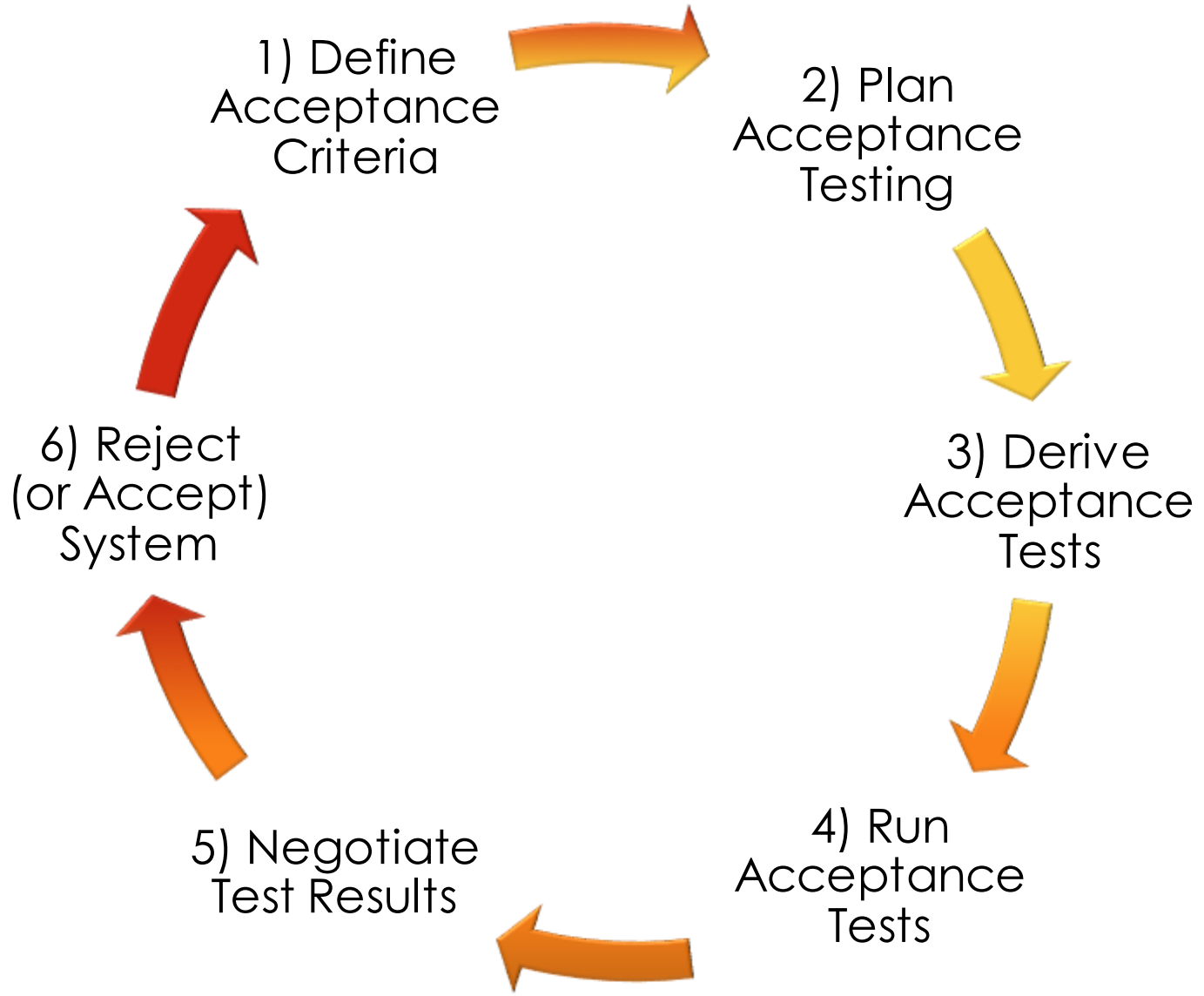
User Testing: stage of testing in which users provide feedback on system testing

Types (of Interest):

- Alpha Testing: users work with development team onsite
- Beta Testing: release made available to enable user input
- Acceptance Testing: customers test to determine if ready to be accepted

Irreplaceable as the user environment, which guides tuning for reliability, usability, & performance, can not be replicated in a testing environment

ACCEPTANCE TESTING PROCESS



REQUIREMENTS-BASED TESTING

Requirements-based Testing: testing in which test cases, conditions, & data are derived from customer-determined requirements

UNIT TESTING

Unit Testing: automated testing of individual components of software to validate each is performing as expected

Types (of Interest):

- Boundary testing: validates value lies within, on, or outside of a boundary to ensure proper edge case handling

PERFORMANCE TESTING

Performance Testing: testing emergent properties of a system, e.g., performance & reliability, with incremental load increases

Types (of Interest):

- Stress Testing: system intentionally overloaded to determine breakpoint & failure behaviour

UNIT BOUNDARY & STRESS TESTING

An introduction to the Unity Test Framework

UNITY TEST FRAMEWORK (UTF)

Unity package offering a standard test framework

Notes:

- Offers 2 testing modes
- For automated (or not) tests
- Not necessary for stress tests
- Use guide coming soon to your GitHub

EDIT MODE VS. PLAY MODE TESTS

Edit Mode Tests:

- Only run in Unity Editor
- Access to Editor & game code
- For code for which game need not be executed
- Best suited for testing with automated inputs e.g., unit tests
- Faster runtime

Play Mode Tests:

- Run as coroutines
- Best suited for conditions only met while game running
- Better for movement, animations, etc.

UTF INSTALLATION

Prerequisite: Unity Editor versions 2019.2 or higher

1. Open project in Unity Editor
2. Select **Window** > **Package Manager**
3. Select **Packages: Unity Registry** from dropdown at top of window
4. Locate **Test Framework** by scrolling or entering in search
5. Select **Install** at bottom right of window

UTF BOUNDARY TEST CREATION

Prerequisite: project open in Unity Editor version 2019.2 or higher

1. Select **Window** > **General** > **Test Runner**
2. Select **EditMode**
3. Select **Create EditMode Test Assembly Folder**
4. Select **Create Test Script in current folder**
5. Exit **Test Runner**
6. Rename created folder (e.g., EditMode_Tests) & script

UTF TEST EXECUTION

Prerequisite: preexisting test

1. Open project in Unity Editor
2. Select **Window** > **General** > **Test Runner**
3. Select **PlayMode** or **EditMode**
4. Select **Run All** or **Run Selected**

STRESS TESTING

A Demonstration

WHAT IS STRESS TESTING?

A stress test is a test that puts the application to its limits in order to see what the engine or hardware can support.

- Stress Testing requires a metric to be chosen based on the goal of the test (ie. Breaking Physics, CPU Usage, Memory Usage)
- Stress Testing is meant to push the application to its limits, so spawning hundreds of enemies or interacting with an object rapidly are good ways to stress a system
- Have fun with the stress tests and be creative

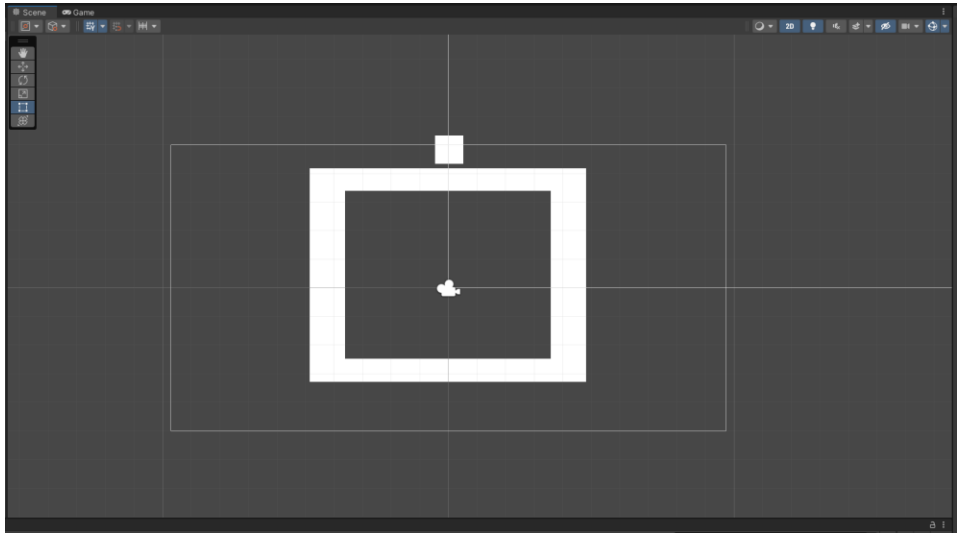
STRESS TESTING BASIC EXAMPLE

- In this basic example, a test will be done to see how many enemies can be spawned in a confined space before physics break
- The test will confine enemies in a small space to force collisions
- The test will spawn more enemies until an enemy breaks out of the cube

- **Note: FPS based stress tests only count for 80%**

STRESS TESTING BASIC EXAMPLE SETUP

- First a separate scene must made to keep production and testing separate
- Stress Testing does not require the use of the Unit Testing Framework
- This basic scene has a box to confine the enemies and a player on top that they will detect for extra load



STRESS TESTING BASIC EXAMPLE SCRIPT

- In order to run the test a simple game manager script will need to facilitate it

```
    void Update()
    {
        float iFramerate = 1 / Time.smoothDeltaTime;

        if (Time.frameCount % 10 == 0 && iBelow60 < 5)
        {
            if (iFramerate > 60)
            {
                Instantiate(goBasicEnemy, new Vector3(0, 0, 0), Quaternion.identity);
                iEnemyCount++;
                tmpEnemyCount.SetText("Enemies: " + iEnemyCount);
            }
            else
            {
                iBelow60++;
            }

        }

        if (Time.frameCount % 20 == 0)
        {
            tmpFPS.SetText("FPS: " + iFramerate);
        }
    }
}
```

```
public class GameManagerStressTest1 : MonoBehaviour
{
    public GameObject goBasicEnemy;
    public int iEnemyCount;
    public TextMeshProUGUI tmpEnemyCount;
    public TextMeshProUGUI tmpFPS;
    private int iBelow60;

    Unity Message | 0 references
    void Start()
    {
        iEnemyCount = 0;
        iBelow60 = 0;
    }
```

STRESS TESTING BASIC EXAMPLE RUNNING

- Running the test spawns 180 enemies before the threshold of 60 fps is reached
- This shows that the game can handle 180 enemies with collision trying to pathfind the player before the performance is severely impacted

DESIGN PATTERNS

Types of patterns & their methods

WHAT ARE DESIGN PATTERNS?

General repeatable solution for common software design problems

Uses:

- Speeds up development process
- Prevents subtle issues that cause major problems
- Improves code readability for those who use patterns
- Provides general solutions in a format that does not require specifics
- Allows developers to communicate using known names for software interactions

TYPES OF PATTERNS

- Creational Patterns
  - Class creation patterns use inheritance effectively during instantiation
  - Object creation patterns use delegation effectively to do the job
- Structural Patterns
  - Uses class inheritance to compose interfaces
  - Define ways to compose objects to obtain new functionality
- Behavioral Patterns
  - All about class and object communication

CREATIONAL PATTERNS

- Abstract Factory
  - Creates an instance of several families of classes
- Builder
  - Separates object construction from its representation
- Factory Method
  - Creates an instance of several derived classes
- Object Pool
  - Avoids expensive acquisition and release of resources by recycling objects that are no longer in use
- Prototype
  - A fully initialized instance to be copied or cloned
- Singleton
  - A class of which only a single instance can exist

https://sourcemaking.com/design_patterns

STRUCTURAL PATTERNS

- Adapter
  - Match interfaces of different classes
- Bridge
  - Separates an object's interface from its implementation
- Composite
  - A tree structure of simple and composite objects
- Decorator
  - Add responsibilities to objects dynamically
- Façade
  - A single class that represents an entire subsystem
- Flyweight
  - A fine-grained instance used for efficient sharing
- Private class data
  - Restricts accessor/mutator access
- Proxy
  - An object representing another object

https://sourcemaking.com/design_patterns

BEHAVIORAL PATTERNS

- Chain of Responsibility
  - A way of passing a request between a chain of objects
- Command
  - Encapsulate a command request as an object
- Interpreter
  - A way to include language elements in a program
- Iterator
  - Sequentially access the elements of a collection
- Mediator
  - Defines simplified communication between classes
- Memento
  - Capture and restore an object's internal state

https://sourcemaking.com/design_patterns

BEHAVIORAL PATTERNS

- Null Object
  - Designed to act as a default value of an object
- Observer
  - A way of notifying change to a number of classes
- State
  - Alter an object's behavior when its state changes
- Strategy
  - Encapsulates an algorithm inside a class
- Template Method
  - Defer the exact steps of an algorithm to a subclass
- Visitor
  - Defines a new operation to a class without change

https://sourcemaking.com/design_patterns

DESIGN PATTERN EXAMPLES

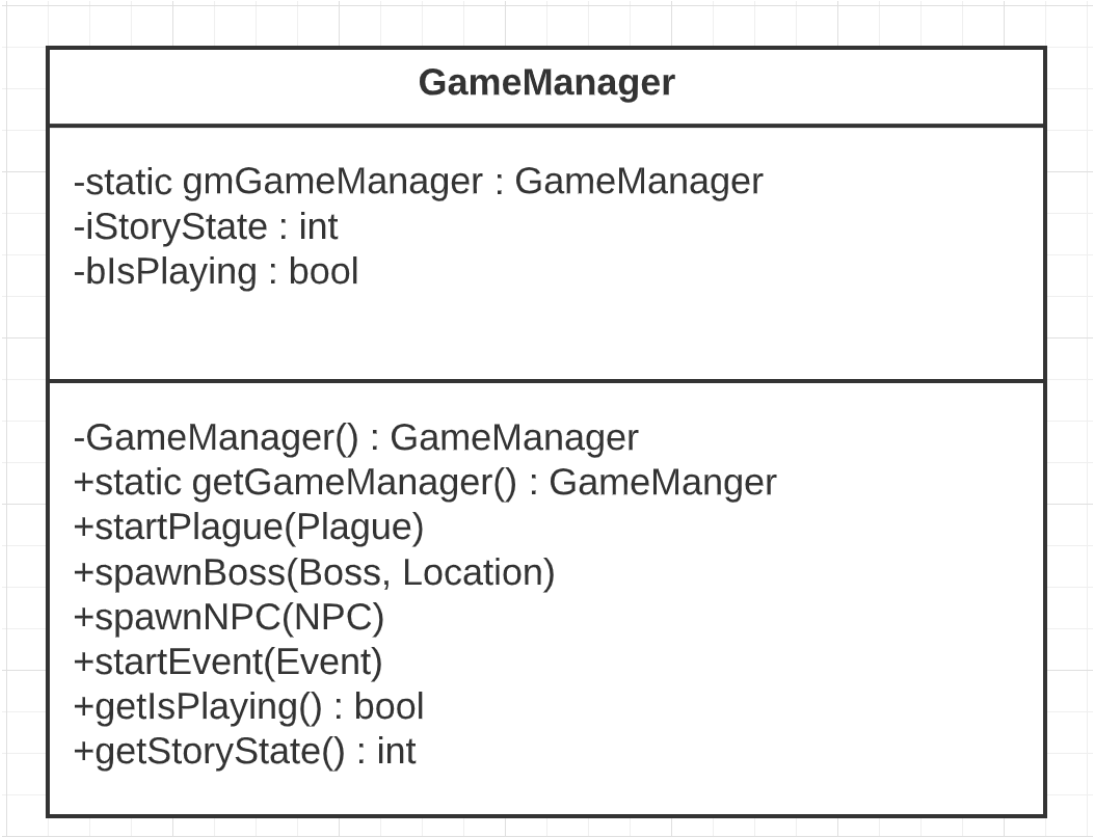
A few illustrative pattern examples

EX1: (JORDAN) SINGLETON

- The Singleton pattern is a class that ensures it only ever has one instance at a time.
- A singleton should have a private constructor and a public static accessor function.
- Upon calling the accessor, it returns the singleton. If it has not been instantiated yet, it first does so before returning it.
- Good uses for a singleton include game managers & loggers.

EX 1: (JORDAN) SINGLETON CLASS DIAGRAM

Class diagram of a singleton game manager implementation.



EX 1: (JORDAN) SINGLETON IMPLEMENTATION

This is a basic singleton class, but it is not made to be threadsafe.

```
class Singleton {

private:
    static Singleton uniqueInstance;
    Singleton() { }


public:
    static Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }

}
```

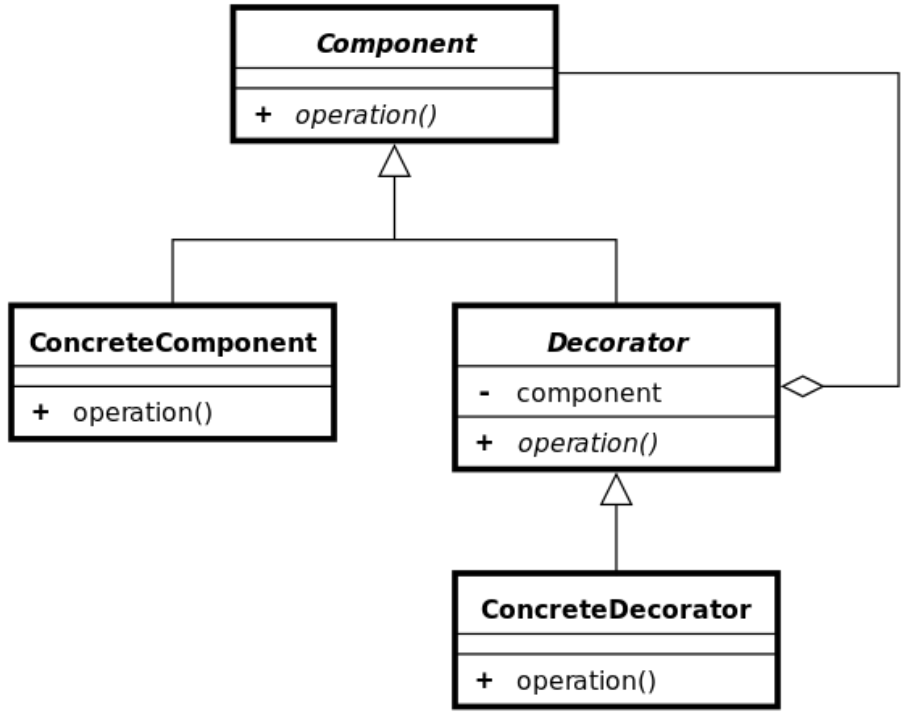
EX 1: (JORDAN) SINGLETON IMPLEMENTATION CONTD.

This singleton class has been modified to make it thread safe using a lock

```
public sealed class Singleton
{
    private static Singleton instance = null;

    private static readonly object padlock = new object();
    Singleton() { }

    public static Singleton Instance {
    get {
        if (instance != null) {
            return instance;
        }
        lock (padlock) {
            if (instance == null) {
                instance = new Singleton();
            }
            return instance;
        }
    }
}
```

EX 2: (NATHAN) DECORATORS

- A decorator, also called a wrapper, is a structural design pattern that adds optional functionality to a class.
- Has an interface, a core component class, a wrapper class, and classes for each decoration.
- The wrapper class implements the component and uses an implementation of the component in its constructor.
- EX:



Decorator Class Diagram -Wikipedia

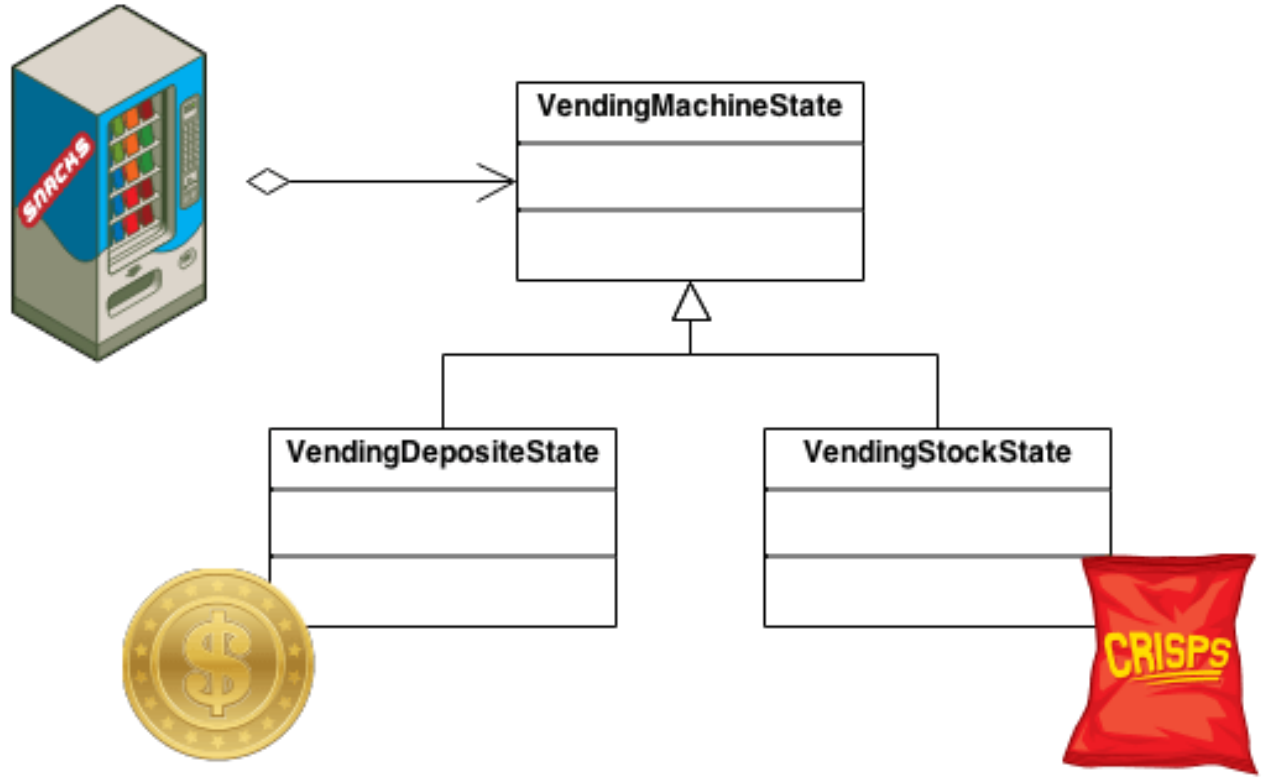
```
DecTree theTree = new d_lights(new d_ornaments(new Tree()));
```

- Here is an example of a decorator code.
- Objects that are dying can be decorated with dropping items, if they have any items.

```
public interface IKillable
{
    void Kill();
}

public class CoreKillable : MonoBehaviour, IKillable
{
    public void Kill()
    {
        Destroy(gameObject);
    }
}

public class HeldItemsDecorator : IKillable
{
    private readonly IKillable obj;
    private Item[] held_items;

    public HeldItemsDecorator(IKillable dying_thing, Item[] items)
    {
        obj = dying_thing;
        held_items = items;
    }

    public void Kill()
    {

        foreach(Item i in held_items)
        {
            //i.generate(gameObject.Transform.Position);
        }
        obj.Kill();
    }
}
```

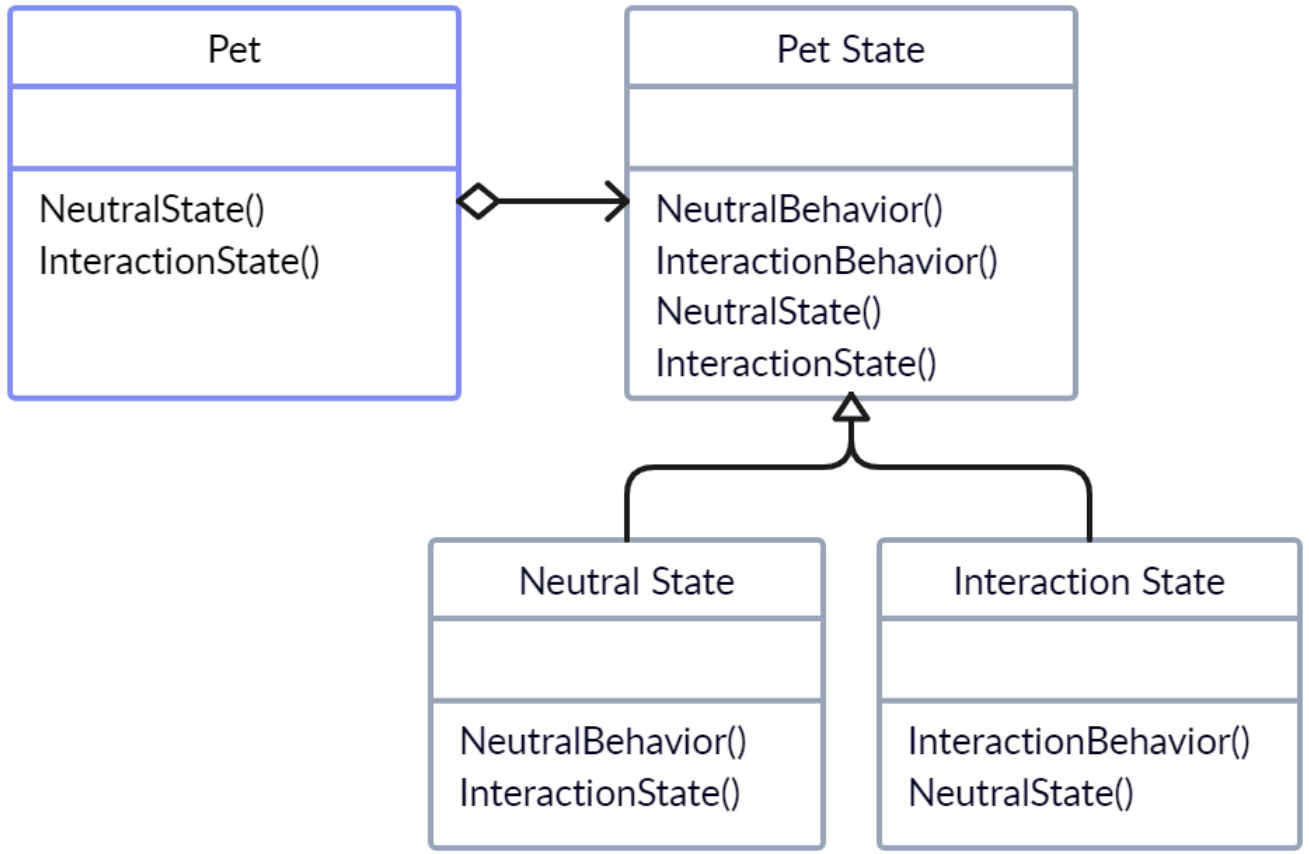
EX 3: (TRYSTON) STATE PATTERNS

- Context Class
  - Client interaction interface
  - Contains references to Concrete States
- State Class
  - Interface for declaring what the Concrete Stats should do.
- Concrete State Class
  - Implementation of methods defined in the state.



EX 3: (TRYSTON) STATE PATTERN CLASS DIAGRAM

- Pet is the Context Class
- Pet State is the State Class
- Neutral State is a Concrete Class
- Interaction State is a Concrete Class



DELIVERABLES

Individual tasks for next Tuesday & Oral Exam

NEXT TUESDAY (07/03/23)

For each member:

- Code 1 stress test
- Code 2 unit boundary tests
- Push tests to test folder in GitHub
- Update Gantt chart

Each individual is expected to demonstrate:

- 2 instances of patterns
  - Justify choices
  - Draw class diagram for 1
- Complete test plan
  - Describe
  - Explain intention
  - Justify choices

ORAL EXAM (16 - 21/04/23)

THANK YOU FOR YOUR ATTENTION.

Refer to your team lead 3 for questions throughout the week.