# Team Lead 3 – Quality Assurance Presentation

Anne Johnson, Eric Johnson, Alex Johnson, Erik Peavey, Jack Kroll, Nathan Sawyer, & Jaskaran Singh

# Presentation overview

## Introduction

- What is testing?
- The purpose of testing
- Types of testing
- isomorphism

## Design Patterns

- What are design patterns?
- Creational patterns
- Structural patterns
- Behavioral patterns

## Introduction to Tests

- Unity Framework (UTP)
- ATRIP
- Unit Testing
- Integration Testing
- EditMode vs PlayMode tests

## *Boundary Tests*

- What is a boundary test?
- Running boundary test

## *Stress Tests*

- What is a stress test?
- Running a stress test

# Individual Requirements

- Initial Test Plan (Due: Next Thursday)
  - 1 stress test
  - 2 boundary tests
- Full test plan (Oral Exam)
- 2 Patterns somewhere in individual code (Oral Exam)
  - Be able to justify pattern identified.
  - Discuss where you would not use it.

# Testing

# What is Testing?

- Testing is the process of identifying weaknesses, errors, or defects in an application or system.

# The Purpose of Testing

The purpose of testing is to gain a better understanding, of how a system currently works under specific conditions, so that we can either verify that it is working properly or, identify errors that need to be fixed.

# Types of Testing

- Release testing
- Use Case testing
- Requirement based testing
- Performance testing
- User testing
- Integration testing

# Alpha Testing & Beta Testing

- Alpha tests – initial tests done by the creator.
- Alpha tests find expected bugs

- Beta tests – done by other users
- Beta tests find unexpected and strange bugs

# What Makes a Good Test Plan?

- Reasonable and valid tests
- Finds Bugs within the game
- Finds a bug that ends up saving time!
- Will ensure you pass the Oral Exam!

# ATRIP

When testing programs it should follow these attributes:

**A - Automatic**
- Tests should run automatically without manual input
- Allows for easy integration in continuous integration systems.

**T - Thorough**
- Tests should cover all important functionality and edge cases
- Ensures that each part of the program behaves as expected under various conditions.

**R - Repeatable**
- Test should be reproducible with the same results regardless of environment or timing
- Avoids inconsistent test outcomes

**I - Independent**
- Each test should run on its own without a dependence on data or other tests
- Makes it easier to identify what test failed and why.

**P - Professional**
- Tests should be clear, maintainable, and well-documented
- Use consistent names, syntax, and formatting.

# Unit Testing

Also known as competent testing, unit testing is where individual units or components are tested.

- Ensures that all units of software preform as they are supposed to via the use of units (small bits of code).
- *Unit:* the smallest testable part of code in any software.
  - I.e., Function Procedure – in OOP, a method is the smallest unit
- Follows the ATRIP model.

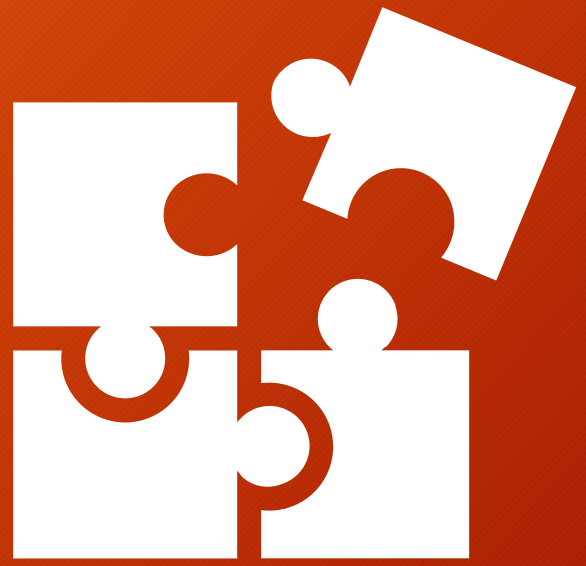
**Why use unit testing?**

It is easy way to improve code speed, promotes consistency, and overall better quality of life.

Without Unit Testing your program becomes a “house of cards”

# Integration Testing

Integration testing is the use of combining the unit test modules into complete interface to verify all modules have working functionality together.



# What Makes a Good Integration Test?

- Ensures all features work together as a group
- Exposes interaction faults
- Unveils unintended consequences

# 3 Important Parts of Integration Testing

- All modules work together
- Uncovers errors that would have plagued the product
- Ensures underlying features are not affected

# The Four Approaches

## Top Down

- High level to low level
- Bad for early release

## Bottom Up

- Low level to high level
- May detect key defects late

## Big Bang

- Test product as a whole
- No prioritization

## Sandwich

- Top down and bottom up at the same time
- Difficult, and not extensive

# Design Patterns

# What are design patterns?

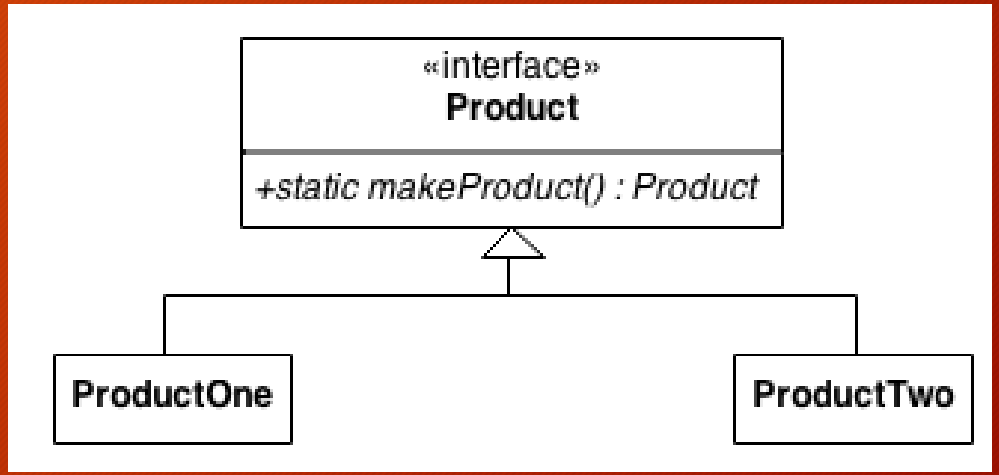
Design patterns are general paradigms for the organization of objects/classes which have been proven to work time and time again within software and can be used as a general starting template for structuring code.

# Creational patterns

Creational patterns are built around object creation mechanisms, an attempt to create objects in a way that is relevant to the current situation

# Factory Method

- Helps create game objects without writing new code for each type
- Reduces setup time and keeps creation organized
- **Example:** The game spawns different enemies (Goblin, Troll, Dragon) using a factory instead of manually creating each one



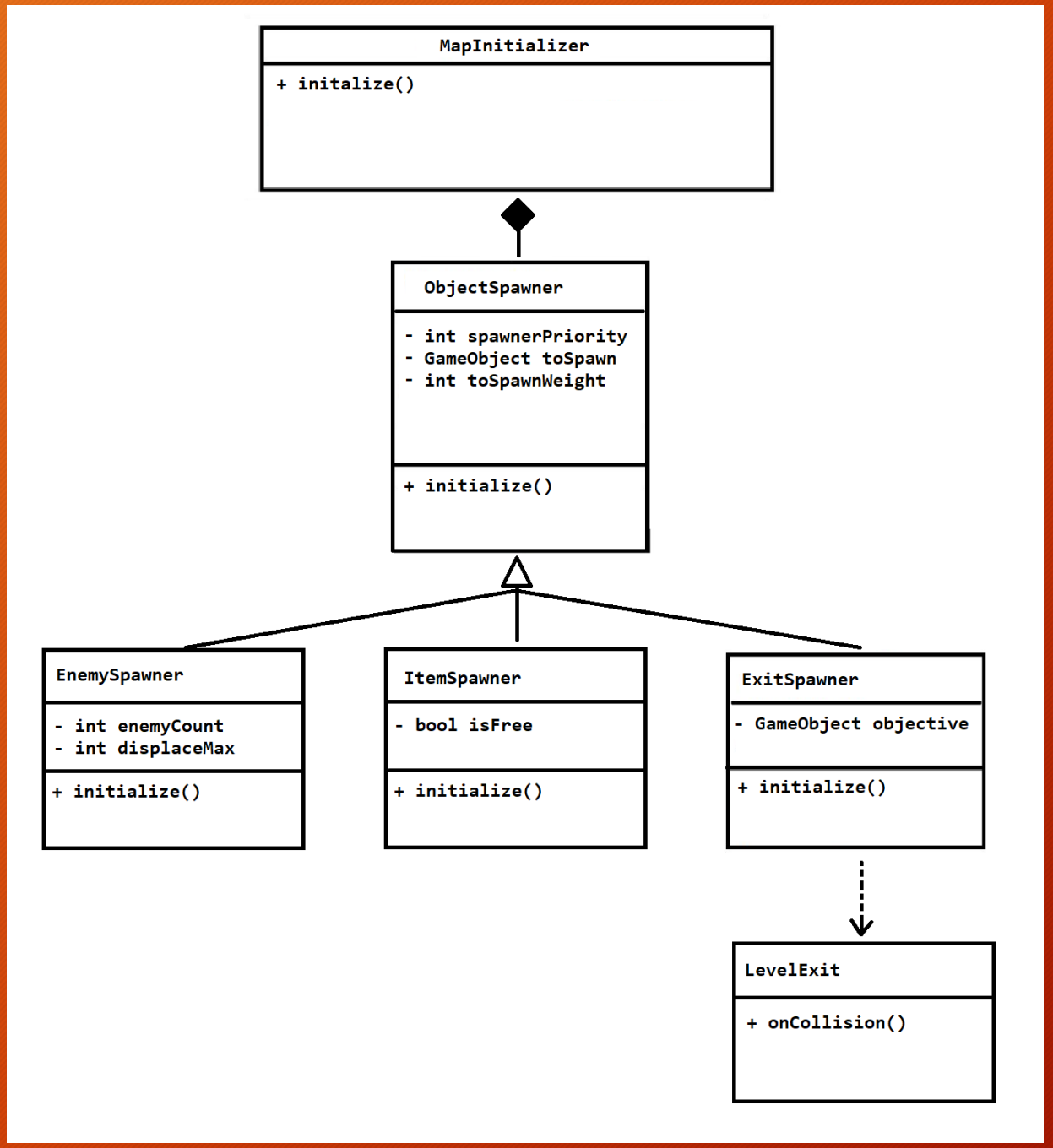

(Objects are not individual classes)

# Abstract Factory

- Used when you need to make **groups** of related objects
- Keeps your scenes consistent by grouping objects that belong together
- **Example:** A level factory creates matching tiles, enemies, and items for each biome (desert, forest, snow)

# Builder

- Builds complex game objects step by step
- Useful when something has lots of parts that change
- **Example:** A map generator that builds a level one section at a time — terrain, enemies, and collectibles


MapInitializer
+ initalize()
ObjectSpawner
- int spawnerPriority
- GameObject toSpawn
- int toSpawnWeight
+ initialize()
EnemySpawner
- int enemyCount
- int displaceMax
+ initialize()
ItemSpawner
- bool isFree
+ initialize()
ExitSpawner
- GameObject objective
+ initialize()
LevelExit
+ onCollision()

# Prototype

- Makes copies of existing objects using a separate class instead of making new ones from scratch (avoiding "new" operator in C++)
- Saves performance when many similar objects are needed
- **Example:** The game duplicates a basic projectile or enemy prefab instead of creating new ones manually

# Singleton

- Ensures only **one instance** of something exists in the game
- Great for managers or systems that should never be duplicated
- **Example:** GameManager, AudioManager, or UIManager — they control game flow or sound globally

- In multi-threaded environments, extra care is needed to prevent multiple instances

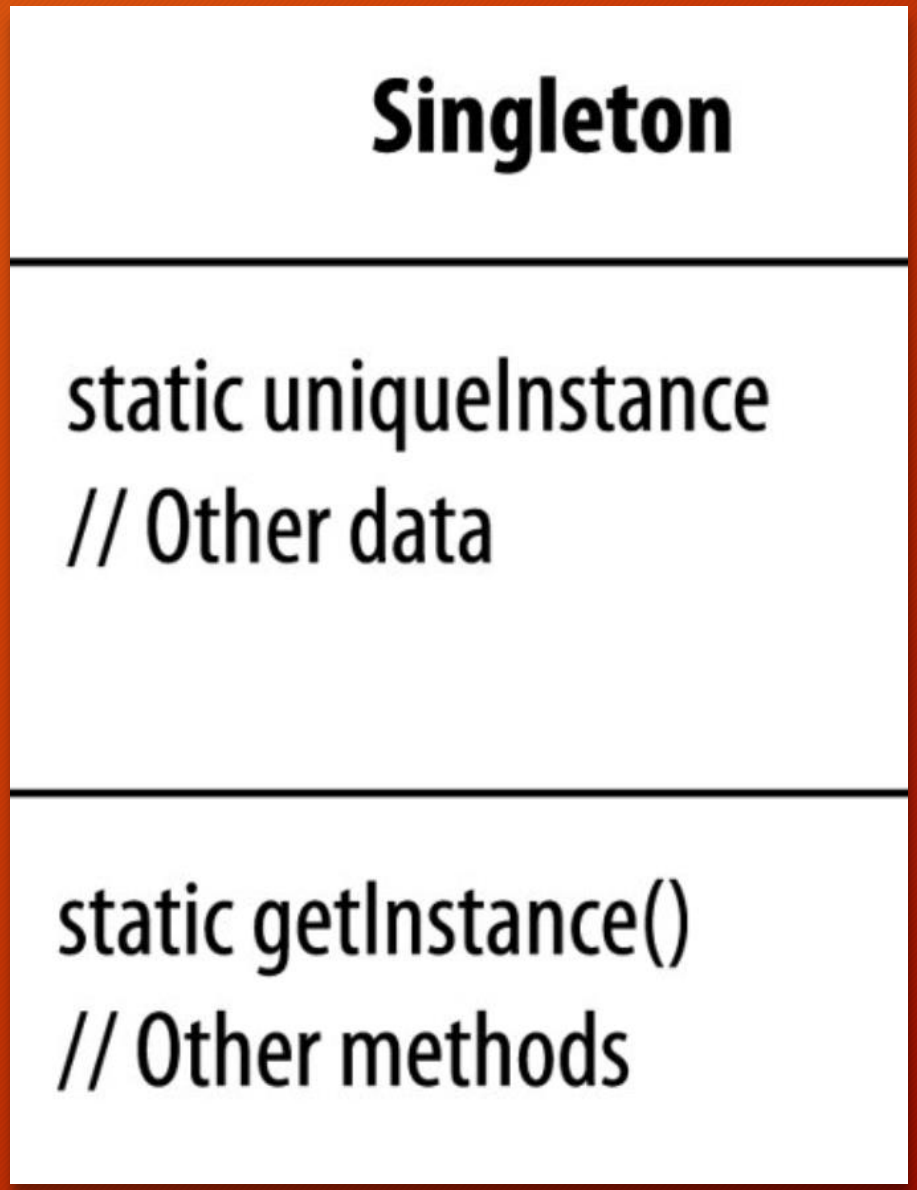
# Classic Singleton Pattern

- `static instance` holds the only object
- Constructor is private → no outside class can create it
- `getInstance()` creates it once → lazy initialization
- Always returns the same object

```
class Singleton {
    private static Singleton instance;
    private Singleton() {} // private constructor
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

# Singleton Class Diagram

- Holds one static object
- `getInstance()` gives global access
- Behaves like a controlled global variable
- Guarantees a single instance



# Multithreading Safety

**Why the bad version fails:**

- Two threads might check at the same time and both see `instance == null`
- Each thread creates its own copy → breaks the “only one instance” rule

**Why the lock version works:**

- The `lock` makes threads take turns entering the code
- Only one thread can create the instance → guarantees a true Singleton

```
// ❌ Not thread-safe
if (instance == null)
    instance = new Singleton();

// ✅ Thread-safe (lock ensures only one thread creates it)
lock(padlock) {
    if (instance == null)
        instance = new Singleton();
}
```

# Object Pool

- Reuses existing objects instead of creating and destroying them constantly
- Improves performance and avoids lag spikes
- **Example:** Bullets or arrows get recycled in a pool instead of being destroyed and recreated every time you shoot

# Structural Patterns

- These patterns focus on **how classes and objects are connected**
- They make it easier to build flexible, organized systems
- Commonly used for **UI, game systems, and level organization**

# Adapter

- Lets two different systems work together
- Great when you have code that doesn’t match how your new system works
- **Example:** Use an Adapter to convert **screen coordinates** to **world coordinates**, so UI clicks line up correctly in the game

# Bridge

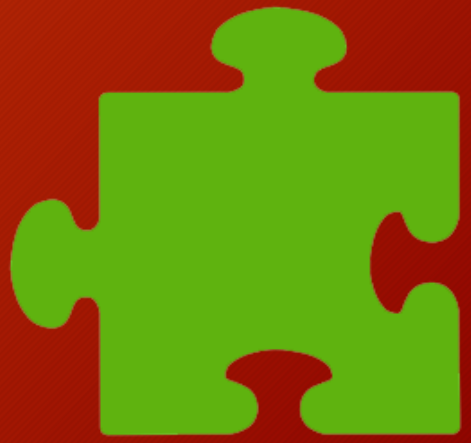
- Separates what an object does from how it does it
- Makes it easier to change features without breaking other parts
- **Example:** A weapon system where the **Weapon class** is separate from its **AttackType** — you can swap between melee, ranged, or magic attacks easily

# Composite

- Lets you treat **single objects and groups of objects the same way**
- Useful when you have objects that contain other objects
- **Example:** A **game menu** made of buttons and panels — you can move or hide one button (a leaf) or the whole menu (a composite) with the same command

# Decorator

- Adds new features to objects **without changing their original code**
- Makes it easy to add or remove abilities dynamically
- **Example:** Adding **power-ups** to the player like speed boost or shield — each decorator adds new behavior on top of the player



# Facade

- Gives a **simple interface** to a complex system
- Helps organize messy subsystems into one clean access point
- **Example:** A **ShopManager** that handles inventory, UI, and currency systems all together behind one easy-to-use interface

# Flyweight

- Shares objects to avoid creating too many duplicates
- Saves memory when you have lots of repeated items
- **Example:** Many **trees or rocks** in a level share the same model, and only their position or size changes

# Private Class Data

- Protects important game data so it can’t be changed accidentally
- Keeps variables hidden and controlled through methods
- **Example:** A **PlayerStats** class hides health, XP, and gold values — other scripts can only access them safely through getter/setter functions

# Proxy

- A **stand-in** that controls access to something else
- Used when the real object is heavy to load or needs protection
- **Example:** A **loading screen** acts as a proxy while the real level is still being loaded in the background

# Behavioral Patterns

Behavioral patterns focus on how objects communicate.
They define common ways for objects to work together.
Their goal is to make communication flexible and reusable.

# Chain of Responsibility

- Passes a request along a **chain of handlers** until one processes it
- Useful for organizing checks or layered systems
- **Example:** A **damage system** where effects pass through armor, shield, and health each layer decides if it handles the hit

# Command

- Turns player actions into **command objects** that can be saved, undone, or replayed
- Makes it easy to manage controls or history
- **Example: Player input** is stored as commands like “Jump,” “Attack,” or “Dash” used for undo systems or replay features

# Interpreter

- Defines grammar or rules for interpreting data or commands
- Helpful when adding custom scripting or dialogue systems
- **Example:** A **dialogue system** that reads simple script lines like “NPC says: Hello” and displays them in-game

# Iterator

- Lets you **loop through items** in a collection without knowing how it’s built
- Makes it easier to organize lists or sequences
- **Example:** A **quest log** uses an iterator to go through all active quests and display them on the UI

# Mediator

- A central object that **manages communication** between others
- Prevents scripts from directly depending on each other
- **Example:** A **Combat Manager** that handles player and enemy interactions so they don’t communicate directly

# Memento

- Saves and restores an object’s state
- Commonly used for undo systems or saving progress
- **Example:** A **Save System** that stores player position, health, and inventory so you can load it later

# Null Object

- Provides a “**do-nothing**” **object** instead of using null
- Avoids crashes or checks for missing references
- **Example:** If there’s no equipped weapon, use a **NullWeapon** that just does nothing when “Attack” is called

# Observer

- One-to-many relationship: when one object changes, others are notified
- Ideal for event systems and UI updates
- **Example:** When the **player's health** changes, the health bar and sound effects update automatically

# State

- Changes an object’s behavior based on its **current state**
- Simplifies complex if/else conditions
- **Example:** A **player** switches between states like Idle, Running, Jumping, and Attacking each with different behavior

# Strategy

- Lets you **swap algorithms or behaviors easily**
- Keeps code clean when you want different options for similar tasks
- **Example:** Enemy AI can switch between **aggressive**, **defensive**, or **patrol** behavior strategies

# Template Method

- Defines the **overall steps of an algorithm** but lets subclasses fill in details
- Great for shared processes with small variations
- **Example:** A **base enemy class** defines attack steps, but each enemy type overrides its own attack animation or damage value

# Visitor

- Adds new operations without changing existing classes
- Useful when you want to process objects in many ways
- **Example:** A **loot system** where a visitor checks each item type (weapon, armor, potion) and gives different rewards