OVERVIEW

Goal

The goal of this technical test is to increase the developer's knowledge base and improve their skill by requiring them to use specific programming principles and tools to solve problems and complete the assigned task. The instructor may work with them to provide a deeper explanation of the material and better evaluate the developer's problem solving process and offer assistance when teachable moments arise.

Review of the Basics

Good Code Comments Itself (GCCI) - More often than not, you aren't solo programming a feature/app and you don't own the code you are writing. In the professional industry it is best practice to write team compliant code, code that cannot be extended, read or debugged is in professional practice, useless. All code written must be properly formatted, explicitly named and written in readable syntax. Never assume or engage in hot-patches or temp code, as it almost always leads to code rot. All code should be written as though the current iteration is the final iteration.

Keep It Simple Stupid (KISS) - Keep your program simple, make sure the solution you're applying matches the complexity of the issue you're trying to solve. Over engineering simple solutions will contribute to the creation of blackbox apps and waste development time and prevent other developers from contributing and therefore blocking team compliance.

You Aren't Gonna Need It (YAGNI) - Avoid unnecessary extension of code and introducing new complexities into the project unless absolutely necessary. Try to solve the problem using the existing code-base and tools (assuming they exist).

Don't Repeat Yourself (DRY) - Every piece of functionality must have a single, unambiguous, authoritative representation within a system. Avoid the practice of re-writing solutions on a case-by-case basis. If the problem is evident enough to be spread throughout the system, it can be solved systematically.

SOLID Programming

Single Responsibility - Every piece of functionality should be written to a new and explicit class. No single class should be responsible for more than its prescribed purpose.

Open-to-Extension-Closed-to-Modification - After a class is written to its prescribed purpose, it should remain completed, and closed to future modification. If additional related functionality is required, it should be done so through extension of the base class.

Liskov Substitution Principle - Writing logic to ensure that in a programmatic system, when referencing an extended class, the parent class can be replaced with any given child class without error.

Interface Segregation - A combination of the principles of “Single Responsibility” and “Write to an Interface”. Every solution that requires open and repeatable use should be written to an interface. Every interface should have an explicit responsibility and should NOT be extended with additional or related responsibilities.

Dependency Inversion - Also known as “Write to an Interface”. High level applications should be constructed through abstract classes (interfaces) and depend less on inheritance that can lead to more complex dependencies and blackboxing.

TECHNICAL TEST

Target Principles and Tools

1.) Inheritance - Practice the expansion of codebase via Inheirentance
2.) Interfaces - Program to interfaces to avoid redundant code structure
3.) Events & Callbacks - Show understanding of and use Unity Events
4.) Coroutines - Show understanding of and use Coroutines
5.) Programming to a System - Practice programming to a pre-designed system to reduce code rot and avoid blackboxing
6.) Program for your Designer - Be conscious of the inspector and design your code for extensibility and ease of use for your in-engine designers
7.) Enums - Show understanding of and use Enums
8.) Data Containers - Pass explicit data container classes into handler classes that act on the data
9.) Controllers & Managers - Use Manager classes to dictate functionality in the context of a designed system
10.) Singletons - Use a Singleton to define persistent data and functionality

Goals

This technical test will require you to grey-box parts of functionality in ASGS’s *Don’t Explode VR*; The test is to complete each goal while using/incorporating the suggested tools & principles listed below. These are guidelines and there are logic gaps to fill; you may complete the goal however you wish, just keep in mind this is a benchmark of your problem solving and understanding of the target tools and principles listed above.

Note: This is a technical test, art and aesthetic are not a consideration, basic grey-boxing of UI and assets are acceptable.

Task

1. Create a Main Menu scene (Approx. 30m - 1hr 30m)
    a. **Goals**
        i. Create two new scenes named “0_MainMenu” and “1_GameLevel”
        ii. Create a MainMenuManager.cs
        iii. MainMenuManager.cs in-inspector should allow a designer to create and populate any number of Level Options
        iv. MainMenuManager.cs should populate these options on the screen and add them to a Layout Group on Start(); each spawned container should be interactable and contain it’s respective data
        v. Selecting a Level Option launches the “1_GameLevel” scene
    b. **Suggested Principles and Tools**
        i. Singleton
        ii. Data Containers
        iii. Enums
        iv. Program for your Designer
2. In the 1_GameLevel scene, create a system that randomly triggers three “Alarms” (Approx. 1hr - 2h)
    a. **Goals**
        i. “Alarms” should trigger from a manager class
        ii. The manager class should invoke a “TriggerAlarm” event at the end of every iteration of a recursive coroutine
        iii. The manager classes’ recursive coroutine should recalculate the time delay and chance to trigger before every invocation
        iv. Each “Alarm” should be a different extended class from the parent class Alarm.cs
        v. Each "Alarm” should change its mesh’s color via a callback to the manager class’s “TriggerAlarm” event; the color should be defined by the extended “Alarm” class’s data
    b. **Suggested Principles and Tools**
        i. Programming to a system
        ii. Inheritance
        iii. Events & Callbacks
        iv. Coroutines
3. Create an InteractionController.cs so individual “Alarms” are disarmed by clicking them (Approx. 15m - 30m)
    a. **Goals**
        i. “Alarms” should implement the interface “IClickable”, which prescribes functionality for what happens on click
        ii. InteractionController.cs should check and register if it has clicked a valid “IClickable” object
    b. **Suggested Principles and Tools**
        i. Programming to a System
        ii. Interfaces

Submit Project

Following instructions are for applicants;

When you've completed the test, please zip your unity project (named Technical Test - don't Explode, Last Name, First Name) and upload to the google drive linked below;
https://drive.google.com/drive/folders/1JdpD4dL-oiDVUhxFsYNbc0d0RzpsWOtx?usp=sharing

Following instructions are for employees;

When you've completed the test, please zip your unity project (named Technical Test - Don't Explode, Name) and upload it to your Personnel folder on the A Square drive.