Middle Unity Developer Test

Focus: 3D • Visual Novels • Gameplay
Level: Middle Unity Developer
Time: 90–120 minutes

You may use **Unity**, **Rider / Visual Studio**, and include code or screenshots as attachments.

There is **no single correct solution**.
We evaluate **thinking, structure, and production readiness**.

1. Coding Principles (Short Answer)

Describe **two coding principles or practices** you consider most important when working on real Unity projects that mix:

- 3D gameplay
- UI systems
- Iteration by designers

Explain **why** they matter and **where you apply them**.

2. Save / Load Utility (Production Basics)

Many of our projects require persistent data:

- player progress
- settings
- VN state
- gameplay flags

Task

Implement a **generic save/load utility** that:

- Saves any **serializable class** to file
- Loads it back safely
- Handles missing or invalid data gracefully

Notes

- You may use JSON serialization
- Focus on **clean API and robustness**
- Assume this utility will be reused across multiple projects

3. Popup / UI System (UI + Architecture)

Our games use popups for:

- confirmations
- story choices
- warnings
- tutorials

Task

Design a **simple popup system** that supports:

- Loading a popup
- Setting:
    - Title text
    - Body text
    - Between **1–5 buttons**
- Assigning callbacks to buttons

3.1 Unity Components Question

Which **Unity components** would you use to build the popup prefab, and **why**?

4. UI Performance & Refactoring (Core Unity Skill)

In one of our scenes, the UI shows live information about active gameplay entities

A junior developer wrote the following code that:

- Produces **incorrect results**
- Causes **performance issues**
- Updates far too often

```
public class CharactersView : MonoBehaviour

{
   [SerializedField] private List<Transform> _characters;

   void FixedUpdate()
   {
      float totalValue = 0f;

      foreach (Transform characterTransform in _characters)
      {
         Character character =
            characterTransform.gameObject.GetComponents<Character>();
         totalValue += character != null ? character.Value : 0f;
```

```
        }

        string text = string.Format(
            "Characters: {0} Avg value: {1}",
            _characters.Length,
            _characters.Length / totalValue
        );

        gameObject.GetComponent<Text>().text = text;
        Debug.Log(text);
    }
}
```

Your Goals

1. Fix **bugs and logical errors**
2. Improve **code quality and structure**
3. Optimize **performance** (practical + theoretical)
4. **Limit UI updates** to once every X frames or a fixed interval
5. Briefly explain **why** you made your changes

You may rewrite the code entirely.

5. Gameplay / State Logic (3D + Systems Thinking)

This task focuses on **gameplay logic**, not UI.

Context

We have multiple gameplay entities in a scene (e.g. enemies, interactables, story actors). Some of them become *inactive* due to gameplay events (destroyed, disabled, completed, etc.).

Task

Design and implement a method or small system that:

- Tracks gameplay entities
- Returns **only active entities**
- Cleanly handles:
    - entities being removed
    - entities being disabled
- Is safe and readable for production use

You may choose:

- OOP approach
- Event-driven approach

- Simple manager or service

Explain your reasoning briefly.

What We Evaluate

We are looking for:

- Solid Unity fundamentals
- UI + gameplay balance
- Performance awareness (GC, `GetComponent`, logging, update loops)
- Clear APIs and naming
- Practical architecture (not over-engineered)
- Ability to reason about real production code

Optional Bonus (Not Required)

- How would you scale these systems for larger projects?
- How would designers interact with this code?
- How would you profile or debug performance issues?

Final Notes

- You may refactor freely
- TODOs with explanations are fine
- Clarity > completeness