Design Patterns in Game Development

Experience and Knowledge of Participants

How many years of experience do you have in software development / programming? *

- ◯ None
- ◯ Less than 1 year
- ◯ 1 - 3 years
- ◯ 3 - 5 years
- ◯ More than 5 years

How familiar are you with design patterns in software development? *

Not familiar at all — Expert

Have you ever used design patterns in one of your projects? *

- ◯ No
- ◯ Hardly ever
- ◯ Yes, occasionally
- ◯ Yes, frequently

Comparison of Prototypes I

Code implementing weapons using design patterns. The design uses the **Strategy Pattern** to define different shooting behaviors, allowing the BalloonGun to delegate shooting logic to a BalloonShotStrategy. It also uses the **Factory Pattern** (IWeaponFactory and BalloonGunFactory) to create weapon objects, which are spawned by the WeaponSpawner class at specific spawn points.

```cs
using UnityEngine;

public interface IWeapon
{
    IShootStrategy ShootStrategy { get; set; }

    public void Shoot();
    public void OnShot(Collider collider);
}
```

```cs
using UnityEngine;

public interface IWeaponFactory
{
    public GameObject Create(Vector3 position, Quaternion rotation);
}
```

```cs
using UnityEngine;

public class BalloonGun : MonoBehaviour, IWeapon
{
    public IShootStrategy { get; set; }

    void Start()
    {
        ShootStrategy = new BalloonShotStrategy();
    }

    public void Shoot()
    {
        ShootStrategy.Shoot(this);
    }

    public void OnShot(Collider collider)
    {
        // Enemy hit logic
    }
}
```

```cs
using UnityEngine;

public class BalloonGunFactory : IWeaponFactory
{
    private GameObject balloonGunObject;

    public BalloonGunFactory(GameObject balloonGunObject)
    {
        this.balloonGunObject = balloonGunObject;
    }

    public GameObject Create(Vector3 position, Quaternion rotation)
    {
        return Object.Instantiate(balloonGunObject, position, rotation);
    }
}
```

```cs
using System.Collections.Generic;
using UnityEngine;

public class WeaponSpawner : MonoBehaviour
{
    private List<IWeaponFactory> factories = new List<IWeaponFactory>();
    private List<Transform> spawnPoints = new List<Transform>();

    public void SpawnWeapons
    {
        foreach (Transform spawnPoint in spawnPoints)
        {
            Vector3 position = spawnPoint.position;
            Quaternion rotation = spawnPoint.rotation;
            // Pick random factory and run factory.Create(position, rotation)
        }
    }
}
```

How would you rate the readability of the code in the prototype with design patterns? *

Very poor — Very good

How would you assess the maintainability of the code with design patterns? *

Very poor — Very good

How high is the risk of code smells occurring in this code (e.g., Large Class, God Object, Duplicate Code, ...)? *

Very low — Very high

Code implementing weapons **without using design patterns**. The design directly handles weapon instantiation and functionality. The IWeapon interface is minimal, and each weapon class, like BalloonGun, must implement its own shooting logic, including cooldown and enemy hit logic. The WeaponSpawner class manages weapon objects and spawns them directly without using a factory.

```
public interface IWeapon
{
    public void Shoot();
}
```

```
using UnityEngine;

public class BalloonGun : MonoBehaviour, IWeapon
{
    private float cooldown = 0.5f;
    private float lastShootTime = 0.0f;

    public void Shoot()
    {
        // Cooldown logic
        // Shooting logic
        // Enemy hit logic
    }
}
```

```
using System.Collections.Generic;
using UnityEngine;

public class WeaponSpawner : MonoBehaviour
{
    private List<GameObject> weaponObjects = new List<GameObject>();
    private List<Transform> spawnPoints = new List<Transform>();

    public void SpawnWeapons()
    {
        foreach (Transform spawnPoint in spawnPoints)
        {
            Vector3 position = spawnPoint.position;
            Quaternion rotation = spawnPoint.rotation;
            // Pick random weapon and run Object.Instantiate(weapon, position, rotation);
        }
    }
}
```

How would you rate the readability of the code in the prototype without design patterns? *

Very poor — Very good

How would you assess the maintainability of the code without design patterns? *

Very poor — Very good

How high is the risk of code smells occurring in this code (e.g., Large Class, God Object, Duplicate Code, ...)? *

Very low — Very high

Which code would you prefer to work with? *

- ◯ The code using design patterns (1)
- ◯ The code without design patterns (2)

Why would you prefer to work with this code? *

Comparison of Prototypes II

This code uses the **State Pattern** to manage player states (IdleState, WalkingState, SprintingState, JumpingState). The PlayerController delegates state-specific behavior to classes implementing the IPlayerState interface. States encapsulate their own logic, such as movement speed and transitions, making the design modular and easy to extend.

```cs
using UnityEngine;

public class PlayerController : MonoBehaviour
{
    public bool CanMove { get; private set; }
    private IPlayerState playerState;

    void Start()
    {
        SetPlayerState(new IdleState(this));
    }

    void Update()
    {
        if (CanMove)
        {
            playerState.UpdateState();
            CheckForStateChange();
        }
    }

    public void SetPlayerState(IPlayerState state)
    {
        playerState = state;
        playerState.EnterState();
    }

    public void CheckForStateChange()
    {
        // Logic to change state based on user input (e.g., keys).
    }
}
```

```cs
public interface IPlayerState
{
    public void EnterState();
    public void UpdateState();
}

public class IdleState : IPlayerState
{
    private PlayerController player;

    public IdleState(PlayerController player)
    {
        this.player = player;
    }

    public void EnterState() { player.SetMovementSpeed(0.0f); }

    public void UpdateState() { player.CheckForStateChange(); }
}

public class WalkingState : IPlayerState
{
    // Properties, constructor and methods ...
    public void EnterState() { player.SetMovementSpeed(2.0f); }
}

public class JumpingState : IPlayerState
{
    // Properties, constructor and methods ...
    public void EnterState() { /* Jumping logic */ }
}
```

How would you rate the readability of the code in the prototype with design patterns? *

Very poor — Very good

How would you assess the maintainability of the code with design patterns? *

Very poor — Very good

How high is the risk of code smells occurring in this code (e.g., Large Class, God Object, Duplicate Code, ...)? *

Very low — Very high

This code implements a PlayerController **without using the State Pattern**. The movement, rotation, and jumping logic are handled directly within the Update method through dedicated private methods (ProcessMovement, ProcessRotation, ProcessJump).

```
using UnityEngine;

public class PlayerController : MonoBehaviour
{
    [Header("Options")]
    [SerializeField] private bool allowSprinting = true;
    [SerializeField] private bool allowJumping = true;

    public bool CanMove { get; private set; }
    private bool IsSprinting => allowSprinting && Input.GetKey(sprintKey);
    private bool ShouldJump => Input.GetKeyDown(jumpKey) && playerController.isGrounded;

    void Update()
    {
        if (CanMove)
        {
            ProcessMovement();
            ProcessRotation();

            if (allowJumping)
                ProcessJump();

            MovePlayer();
        }
    }

    private void ProcessMovement() { /* Handles movement and walking/sprinting */ }

    private void ProcessRotation() { /* Handles player rotation based on mouse input */ }

    private void ProcessJump() { /* Applies jump force */ }

    private void MovePlayer() { /* Applies final movement and gravity */ }
}

```

How would you rate the readability of the code in the prototype without design patterns? *

Very poor — Very good

How would you assess the maintainability of the code without design patterns? *

Very poor — Very good

How high is the risk of code smells occurring in this code (e.g., Large Class, God Object, Duplicate Code, ...)? *

Very low — Very high

Which code would you prefer to work with? *

- ◯ The code using design patterns (1)
- ◯ The code without design patterns (2)

Why would you prefer to work with this code? *

Comparison of Prototypes III

This code demonstrates the **Observer Pattern**, where Observable objects (e.g., Timer and ScoreManager) maintain a list of Observers (e.g., TimerText, TimerBar, and ScoreText). Observers are notified via the Notify() method whenever the state of the Observable changes.
The Timer class updates all attached Observers whenever time is removed, ensuring the UI elements like TimerText and TimerBar stay synchronized. This design decouples the Observables from the Observers, improving modularity and scalability by allowing new Observers to be added without modifying the core logic.

```
public interface Observable
{
    public void Attach(Observer observer);
    public void Deteach(Observer observer);
    public void Notify();
}
```

```
public interface Observer
{
    public void UpdateObserver();
}
```

```
using TMPro;
using UnityEngine;

public class GameManager : MonoBehaviour
{
    private ScoreManager scoreManager;
    private Timer timer;

    private ScoreText scoreText;
    private TimerText timerText;
    private TimerBar timerBar;

    private void Start()
    {
        scoreManager = new ScoreManager();
        scoreManager.attach(scoreText);

        timer = new Timer(60.0f);
        timer.attach(timerText);
        timer.attach(timerBar);
    }
}
```

```
using System.Collections.Generic;

public class Timer : Observable
{
    private List<Observer> observers = new List<Observer>();

    // Properties and methods

    public void RemoveTime(float time)
    {
        timeRemaining -= time;
        Notify();

        // Timer logic
    }
}
```

```
using TMPro;
using UnityEngine;

public class TimerText : MonoBehaviour, Observer
{
    private TextMeshProUGUI timerText;

    public void UpdateObserver()
    {
        float time = GameManager.Instance.GetTimer().GetTime();
        timerText.text = "Time Remaining: " + time + "s";
    }
}
```

How would you rate the readability of the code in the prototype with design patterns? *

Very poor — Very good

How would you assess the maintainability of the code with design patterns? *

Very poor — Very good

How high is the risk of code smells occurring in this code (e.g., Large Class, God Object, Duplicate Code, ...)? *

Very low — Very high

This code implements a GameManager **without using the Observer Pattern.** The GameManager directly updates UI elements such as scoreText, timerText, and timerBar within its methods (AddPoints, UpdateTimerUI). Any changes to the UI logic require modifications to the GameManager.

```
using TMPro;
using UnityEngine;
using UnityEngine.UI;

public class GameManager : MonoBehaviour
{
    private GameState state;
    private int score = 0;

    private float totalTime = 60.0f;
    private float timeRemaining;
    private float timeDelta;

    private TextMeshProUGUI scoreText;
    private TextMeshProUGUI timerText;
    private Image timerBar;

    private void Update()
    {
        if (state == GameState.RUNNING)
        {
            timeDelta += Time.deltaTime;
            // Timer logic
        }
    }

    public void AddPoints(int points) {
        score += points;
        scoreText.text = "Score: " + score;
    }

    private void UpdateTimerUI()
    {
        timerText.text = "Time Remaining: " + timeRemaining + "s";
        timerBar.fillAmount = timeRemaining / totalTime;
    }
}
```

How would you rate the readability of the code in the prototype without design patterns? *

Very poor — Very good

How would you assess the maintainability of the code without design patterns? *

Very poor — Very good

How high is the risk of code smells occurring in this code (e.g., Large Class, God Object, Duplicate Code, ...)? *

Very low — Very high

Which code would you prefer to work with? *

- ◯ The code using design patterns (1)
- ◯ The code without design patterns (2)

Why would you prefer to work with this code? *

Comparison of Prototypes IV

The following code examples demonstrate how to access the GameManager class with and without the Singleton Pattern.

This code uses the **Singleton Pattern** for the GameManager, ensuring a single instance to manage game state, score, and timer. Various Observers like ScoreText and TimerText access the Singleton to dynamically update UI elements. The Singleton simplifies global data access and centralizes game logic.

```cs
using TMPro;
using UnityEngine;

public class GameManager : MonoBehaviour
{
    public static GameManager Instance { get; private set; }
    private GameState state;
    private ScoreManager scoreManager;
    private Timer timer;

    private void Awake()
    {
        if (Instance != null && Instance != this)
        {
            Destroy(gameObject);
            return;
        }

        Instance = this;
        SetState(GameState.MAIN_MENU);
        scoreManager = new ScoreManager();
        timer = new Timer(60.0f);
        DontDestroyOnLoad(gameObject);
    }

    public GameState GetState() { return state; }

    public ScoreManager GetScoreManager() { return scoreManager; }

    public Timer GetTimer() { return timer; }
}

```

```cs
using UnityEngine;

public class PlayerController : MonoBehaviour
{
    // Properties and Methods

    void Update()
    {
        if (CanMove && GameManager.Instance.GetState() == GameState.RUNNING)
        {
            // Execute movement methods
        }
    }
}


```

```cs
using TMPro;
using UnityEngine;

public class ScoreText : MonoBehaviour, Observer
{
    // Properties and methods

    public void UpdateObserver()
    {
        int score = GameManager.Instance.GetScoreManager().GetScore();
        scoreText.text = "Score: " + score;
    }
}


```

```cs
using UnityEngine;
using UnityEngine.UI;

public class TimerBar : MonoBehaviour, Observer
{
    // Properties and methods

    public void UpdateObserver()
    {
        float time = GameManager.Instance.GetTimer().GetTime();
        float totalTime = GameManager.Instance.GetTimer().GetTotalTime();
        timerBar.fillAmount = time / totalTime;
    }
}

```

```cs
using TMPro;
using UnityEngine;

public class TimerText : MonoBehaviour, Observer
{
    // Properties and methods

    public void UpdateObserver()
    {
        float time = GameManager.Instance.GetTimer().GetTime();
        timerText.text = "Time Remaining: " + time + "s";
    }
}

```

How would you rate the readability of the code in the prototype with design patterns? *

Very poor — Very good

How would you assess the maintainability of the code with design patterns? *

Very poor — Very good

How high is the risk of code smells occurring in this code (e.g., Large Class, God Object, Duplicate Code, ...)? *

Very low — Very high

This code uses a GameManager **without the Singleton Pattern**. The PlayerController requires the GameManager to be manually assigned in the Inspector, and the BalloonGun, as a prefab, retrieves the GameManager using GameObject.Find() (which is inefficient and considered bad practice).

```
using TMPro;
using UnityEngine;
using UnityEngine.UI;

public class GameManager : MonoBehaviour
{
    private GameState state;
    private int score = 0;
    private float totalTime = 60.0f;
    private float timeDelta = 0.0f;
    private float timeRemaining;

    private void Start()
    {
        timeRemaining = totalTime;
        SetState(GameState.MAIN_MENU);
    }

    private void Update()
    {
        // Logic to update state and timer
    }

    public GameState GetState() { return state; }

    public void AddPoints(int points) { /* Adding points logic */ }

    public void UpdateTimer() { /* Update timer UI */ }
}
```

```
using UnityEngine;

public class PlayerController : MonoBehaviour
{
    private GameManager gameManager;
    // Properties and Methods

    void Update()
    {
        if (CanMove && GameManager.Instance.GetState() == GameState.RUNNING)
        {
            // Execute movement methods
        }
    }
}
```

```
using UnityEngine;

public class BalloonGun : MonoBehaviour, IWeapon
{
    // Properties and methods

    public void Shoot()
    {
        // Cooldown and shooting logic
        GameManager manager = GameObject.Find("GameManager").GetComponent<GameManager>();
        Balloon balloonScript = balloonInstance.GetComponent<Balloon>();
        if (balloonScript != null)
            balloonScript.Initialize(manager);
    }
}
```

How would you rate the readability of the code in the prototype without design patterns? *

Very poor — Very good

How would you assess the maintainability of the code without design patterns? *

Very poor — Very good

How high is the risk of code smells occurring in this code (e.g., Large Class, God Object, Duplicate Code, ...)? *

Very low — Very high

Which code would you prefer to work with? *

- ◯ The code using design patterns (1)
- ◯ The code without design patterns (2)

Why would you prefer to work with this code? *

Do you have any additional feedback, suggestions, or thoughts regarding the design patterns or code examples presented in this survey?