



Programming Patterns

Introduction to Game Development in Unity
Spring 2026, 98-127, Lecture 13

Instructors: Jingxuan Chen, Dario Quintero, Shangyi Zhu, Jeffrey Wang

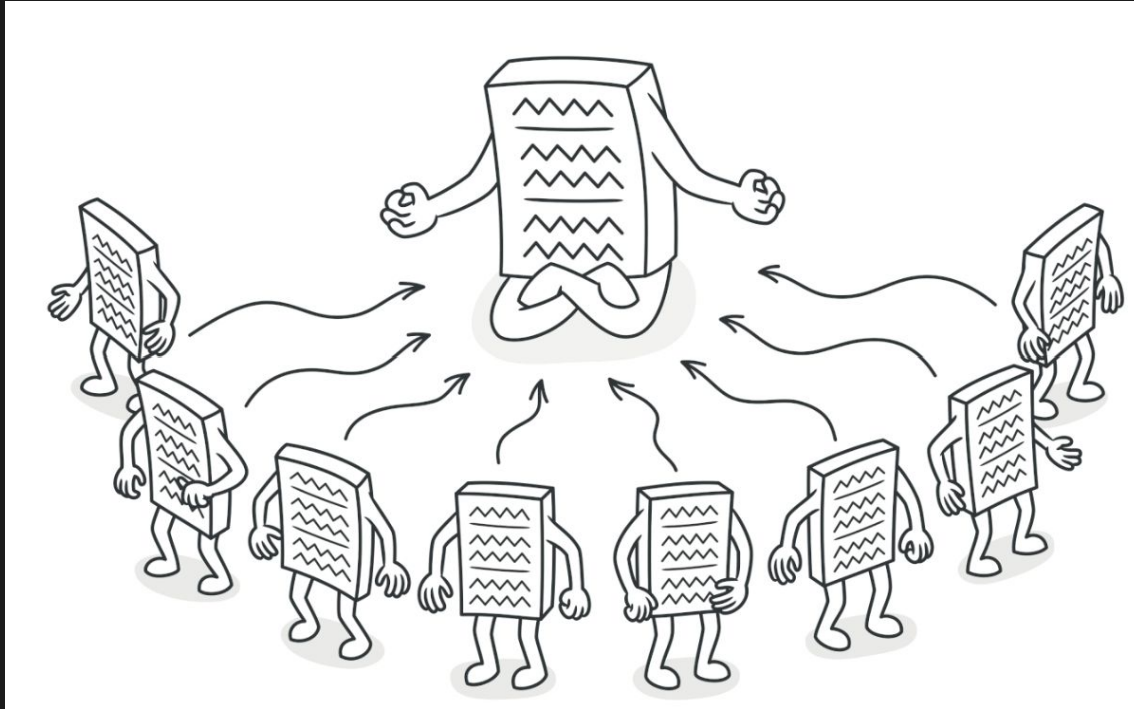


What are Programming Patterns

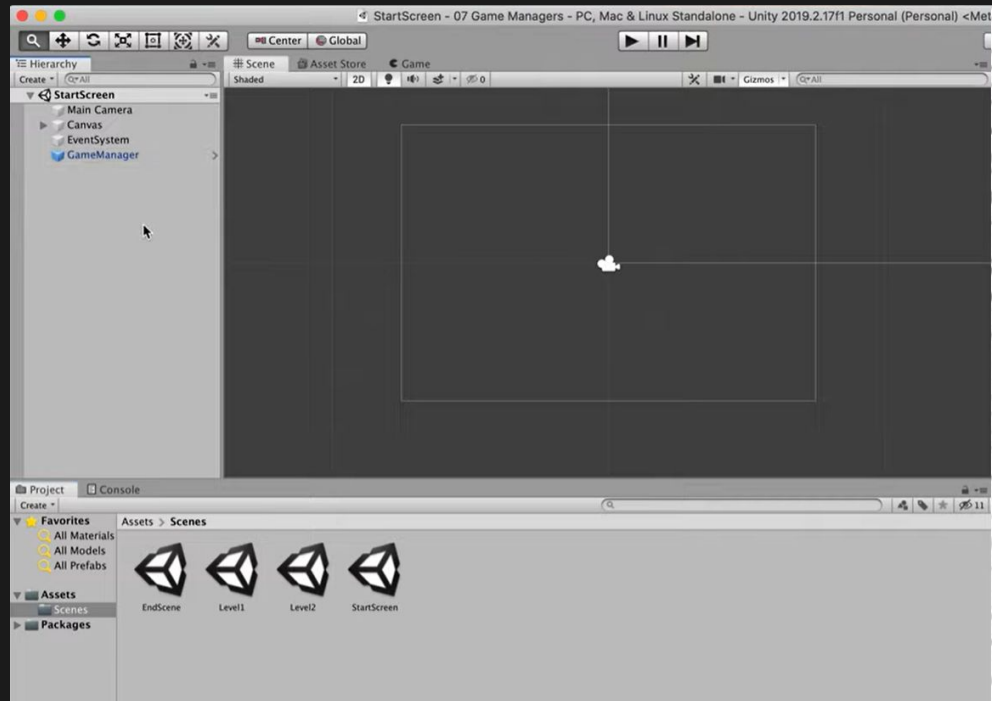
A **game programming pattern** is a reusable solution to a common design problem in game development, describing both how to structure code and the tradeoffs involved.

Singleton

Ensures a class has one instance, and provide a global point of access to it.



Example



```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 public class GameManagement : MonoBehaviour
6 {
7
8     public static GameManagement manager;
9
10    void Awake()
11    {
12        if (manager == null)
13        {
14            manager = this;
15            DontDestroyOnLoad(this);
16        } else if (manager != this)
17        {
18            Destroy(gameObject);
19        }
20    }
21 }
```





Pros

Convenient global access

Cons

...Convenient global access
Hidden call-order dependency





Revisiting Interfaces

What to do when you have multiple classes that each react to the same interaction in slightly different ways?



Before

0 references

```
public class Sword : MonoBehaviour
```

```
{
```

1 reference

```
    [SerializeField] float atk = 10f;
```

0 references

```
    void OnTriggerEnter(Collider other)
```

```
    {
```

```
        if (other.TryGetComponent<Enemy>(out var enemy))
```

```
            enemy.TakeDamage(atk);
```

```
        else if (other.TryGetComponent<Grass>(out var grass))
```

```
            grass.GetCut();
```

```
        else if (other.TryGetComponent<Vase>(out var vase))
```

```
            vase.Shatter();
```

```
    }
```

```
}
```

After

Have each relevant class implement the interface

0 references

```
public class Sword : MonoBehaviour
```

```
{
```

1 reference

```
[SerializeField] float atk = 10f;
```

0 references

```
void OnTriggerEnter(Collider other)
```

```
{
```

```
    if (other.TryGetComponent<IDamageable>(out var damageable))  
        damageable.OnHit(atk);
```

```
}
```

```
}
```

1 reference

```
public interface IDamageable
```

```
{
```

1 reference

```
    public void OnHit(float dmg);
```

```
}
```



What patterns already in Unity?

"State Pattern"



States

A **State pattern** is a way to organize code so an object changes its behavior by switching between separate state objects that each handle one mode of behavior.

"Observer Pattern"



Events

A **Observer pattern** is a way to let one object broadcast events so that multiple other objects can automatically react without being tightly coupled.



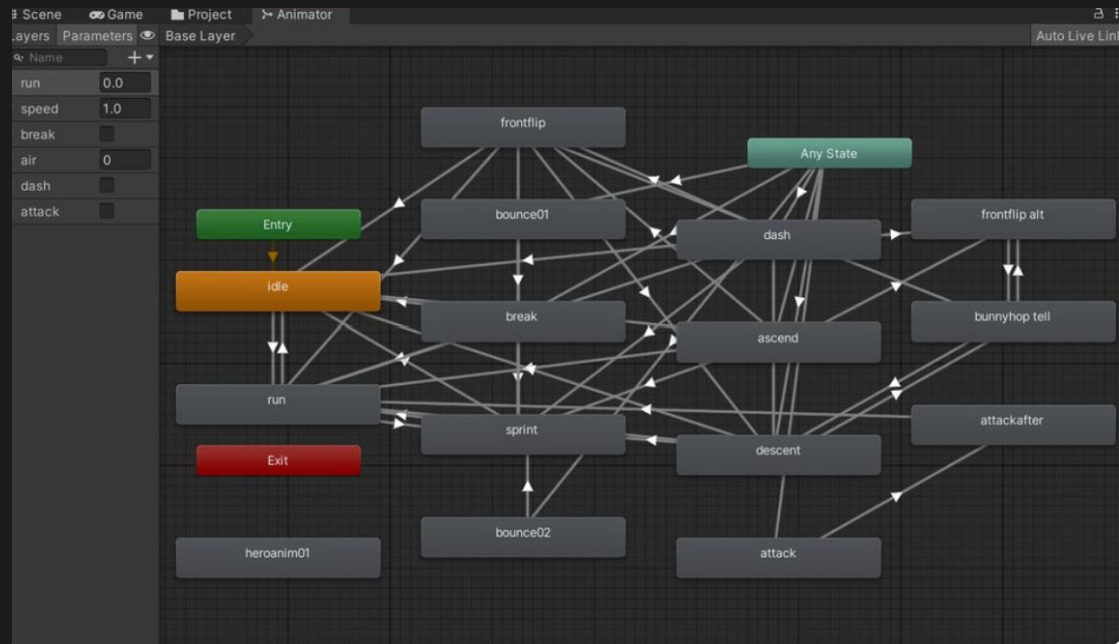
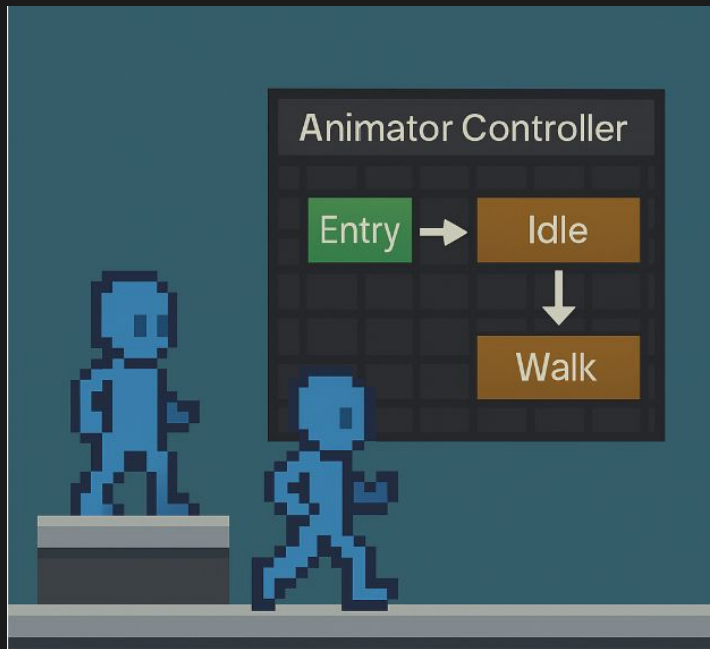
State

Organize code so an object changes its behavior by switching between separate state objects that each handle one mode of behavior.

- Replace boolean logic with states
- Encapsulate behavior per state
- Clear transitions between states



State Pattern



Problem: State Explosion

```
if (isJumping && !isAttacking && !isDead && !isStunned)
{
    // logic
}
```



State Pattern

```
enum PlayerState
{
    Idle,
    Jumping,
    Attacking,
    Dead
}
```

```
PlayerState currentState;
```

```
if (currentState == PlayerState.Jumping)
{
    // jump logic
}
```

Instead of:

```
if (state == Jumping)
```

Can now do:

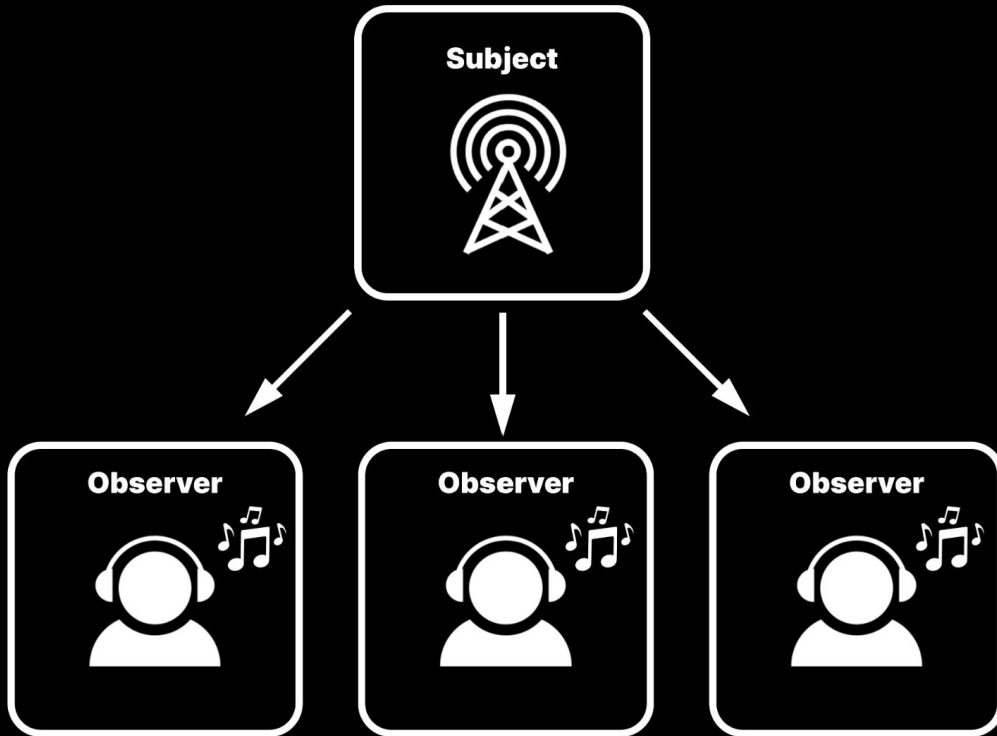
```
currentState.Update();
```

State Implementation

```
// --- State Interface ---  
public interface IPlayerState  
{  
    void Enter(PlayerController player);  
    void Update(PlayerController player);  
    void Exit(PlayerController player);  
}
```

```
public class JumpState : IPlayerState  
{  
    public void Enter(PlayerController  
player)  
    {  
        Debug.Log("Jump!");  
        // apply jump force here  
    }  
    public void Update(PlayerController player)  
    {  
        // simulate gravity / airborne logic  
        if (player.IsGrounded())  
        {  
            player.ChangeState(new  
IdleState());  
        }  
    }  
}
```

Observer



- Broadcast events
- Systems subscribe
- No direct dependencies

Problem: Tight Coupling

```
class JumpState
{
    public void Enter()
    {
        audio.PlayJump(); // X tightly coupled
        ui.UpdateHealth();
        achievement.Track();
    }
}
```



Observer Pattern

```
public static Action OnJump;
```

```
class JumpState
```

```
{
```

```
    public void Enter()
```

```
    {
```

```
        OnJump?.Invoke();
```

```
    }
```

```
}
```



Inside an Action

```
class Action
{
    List<Function> listeners;

    void Invoke()
    {
        foreach (var f in listeners)
        {
            f();
        }
    }
}

// subscribe to OnJump
OnJump += PlayJumpSound;
OnJump += UpdateUI;

// unsubscribe to OnJump
OnJump -= PlayJumpSound;
OnJump -= UpdateUI;
```

Use Cases

- Player movement
- Enemy AI
- Boss phases
- Animation logic

Main Idea

- Systems should be modular
- Low coupling, high cohesion
- Events > direct references



Component Pattern

```
public class PlayerController : MonoBehaviour
{
    // movement
    public float speed;

    // combat
    public int health;
    public int damage;

    // inventory
    public List<string> items = new List<string>();

    // animation
    public Animator animator;

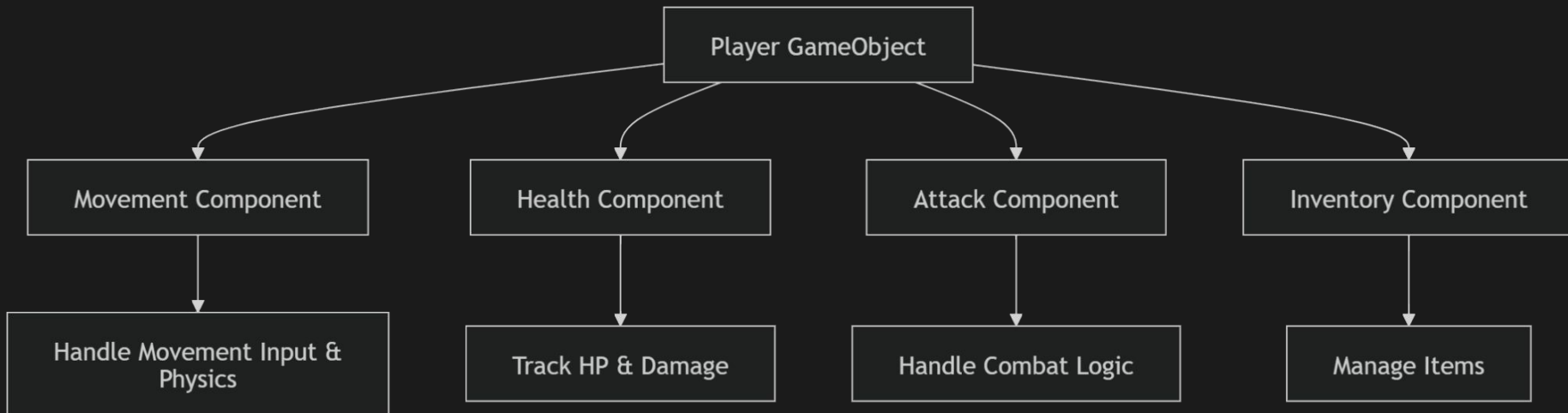
    // audio
    public AudioSource audioSource;

    void Update()
    {
        HandleMovement();
        HandleAttack();
        HandleInventory();
        UpdateAnimation();
        PlayFootsteps();
    }
}
```

- <- This can become hard to maintain
- The Component pattern splits one big object into focused pieces



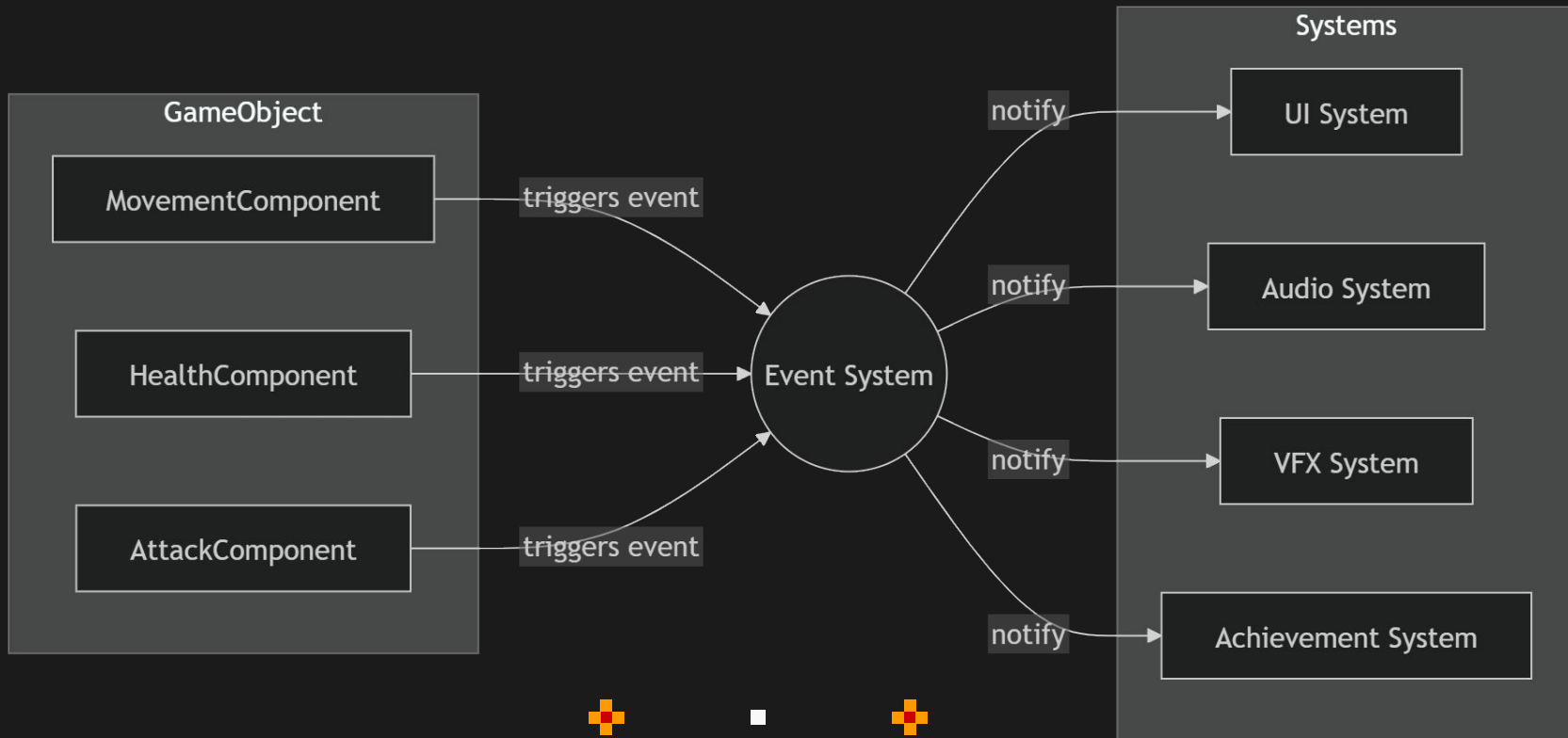
Component Pattern



Component Pattern

```
--
30 public class PlayerMovement : MonoBehaviour
31 {
32     public float speed = 5f;
33
34     public void Move(Vector2 input)
35     {
36         transform.Translate(input * speed * Time.deltaTime);
37     }
38 }
39
40 public class PlayerHealth : MonoBehaviour
41 {
42     public int currentHealth = 100;
43
44     public void TakeDamage(int amount)
45     {
46         currentHealth -= amount;
47     }
48 }
49
50 public class PlayerAttack : MonoBehaviour
51 {
52     public int damage = 10;
53
54     public void Attack()
55     {
56         Debug.Log("Player attacks for " + damage);
57     }
58 }
```

Component + Observer



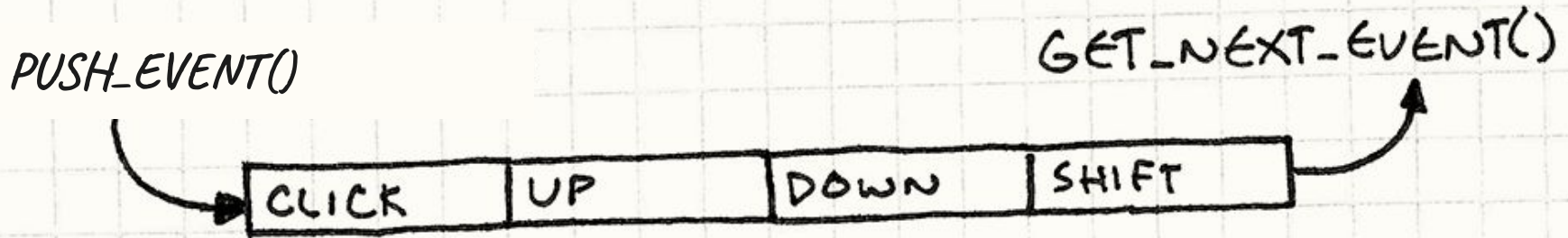
Benefits

- No hardcoding
- Reusable systems
- Scales to large projects



Event Queue

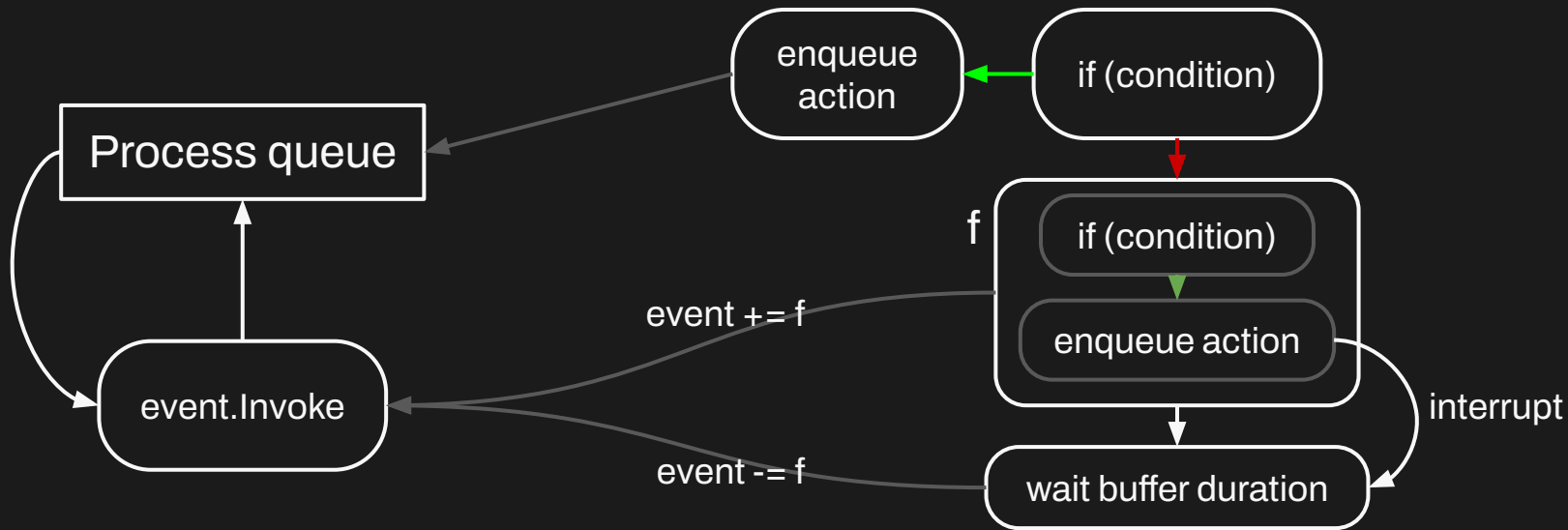
Decouple when a message or event is sent from when it is processed



Event Queue + Observer example

queue update loop

Adding an action to queue:



```
void Buffer(float t):
    if (buffered) return;
    buffered = true;
    queue.event += TryAction;
    StartTimer(t).Forget();
}
```

```
void TryAction():
    if (condition):
        enqueue action
        cts.Cancel()
        cts = new()
    else: Buffer(d)
```

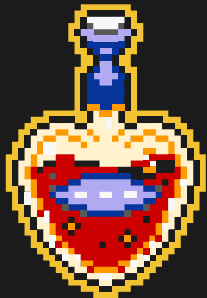
```
async UniTaskVoid StartTimer(float t):
    // wait for t seconds, ending immediately if cts.Cancel() is called
    await UniTask.Delay(
        TimeSpan.FromSeconds(t), cancellationToken: cts.Token
    ).SuppressCancellationThrow()
    buffered = false
    queue.event += TryAction
```



Book for reference

Book by Robert Nystrom:

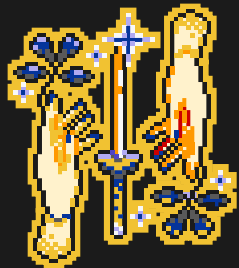
<https://gameprogrammingpatterns.com/contents.html>



End of Lecture

Attendance taking and break time

:v



CREDITS: This presentation template was created by [Slidesgo](#), and includes icons by [Flaticon](#), and infographics & images by [Freepik](#)

