



МОСКОВСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Факультет Информационных технологий
Кафедра Информатики и информационных технологий

направление подготовки

09.03.02 «Информационные системы и технологии»

ЛАБОРАТОРНАЯ РАБОТА № 1

Дисциплина: «Шаблоны проектирования»

Тема: *Реализация паттерна Strategy*

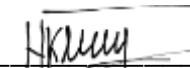
Выполнил: студент группы: 231-339

Карапетян Нвер Каренович

(Фамилия И.О.)

Дата, подпись: 13.03.25

(Дата)


(Подпись)

Проверил:

(Фамилия И.О., степень, звание)

(Оценка)

Дата, подпись

(Дата)

(Подпись)

Москва
2025

Цель:

Реализуйте простую систему принятия решений для NPC (персонажа, управляемого компьютером) с использованием шаблона Strategy.

Описание:

В играх NPC часто требуется динамически реагировать на различные ситуации, меняя свое поведение в зависимости от контекста. Шаблон Strategy позволяет инкапсулировать различные алгоритмы или стратегии поведения, что делает систему AI более гибкой и легко модифицируемой.

План работы

Определение стратегий:

Определить несколько различных стратегий поведения для моего NPC-моба. Например, «патрулирование», «преследование игрока», «бегство от игрока при низком уровне здоровья» и так далее.

Реализация шаблона Strategy:

Создать абстрактный класс или интерфейс для стратегии, который определяет общий метод действия.

Для каждой стратегии создать конкретный класс, реализующий наш интерфейс или абстрактный класс стратегии. Каждый класс должен содержать логику, специфичную для этой стратегии поведения.

Интеграция с NPC:

Интегрировать нашу систему стратегий с NPC, позволяя ему выбирать и менять стратегии в зависимости от игровой ситуации.

Тестирование:

Запустить нашу игру и понаблюдать, как NPC меняет свое поведение в различных ситуациях, основываясь на выбранной стратегии.

Ход работы

Обоснование выбора паттерна проектирования

Шаблон **Strategy** был выбран в качестве основного механизма организации поведения NPC, поскольку он позволяет разделить логику принятия решений и реализации конкретных действий. Это способствует:

- Повышению модульности и читаемости кода.
- Легкости расширения функционала без изменения основных классов.
- Следованию принципу открытости/закрытости (**Open/Closed Principle**) и принципу единственной ответственности (**Single Responsibility Principle**) в рамках **SOLID**.
- Повышению переиспользуемости и тестируемости кода.

Данный подход позволяет NPC динамически изменять своё поведение в зависимости от игрового контекста, что делает их поведение более реалистичным и адаптивным.

Реализация интерфейса

Для реализации данной системы были разработаны три основные стратегии поведения: **патрулирование, преследование цели и бегство при низком уровне здоровья**.

В первую очередь был определен интерфейс стратегий поведения, от которого в дальнейшем будут наследоваться поведенческие классы.

Листинг 1. `IEnemyBehavior.cs`

```
public interface IEnemyBehavior
{
    void Execute(EnemyController enemy);
}
```

В нем определен лишь метод `Execute` с возвращаемым типом `void`, который принимает в качестве параметра экземпляр класса, в котором реализована логика и «мозги» нашего NPC.

Реализация стратегий

Стратегия патрулирования (PatrolBehavior)

Логика данной стратегии кристально проста: NPC в зависимости от своего искомого положения определяет две крайние точки по обе стороны от себя, находящиеся на заданном расстоянии. Он начинает двигаться влево и вправо в рамках области, заключенной между крайними точками. Достигая края маршрута, он разворачивается и продолжает свое движение в обратном направлении.

Листинг 2. PatrolBehavior.cs

```
using UnityEngine;

public class PatrolBehavior : IEnemyBehavior
{
    public void Execute(EnemyController enemy)
    {
        float direction = enemy.moveRight ? 1f : -1f;
        enemy.RB.linearVelocity = new Vector2(direction * enemy.moveSpeed, enemy.RB.linearVelocity.y);

        if ((enemy.transform.position.x >= enemy.rightPoint.position.x && enemy.moveRight) ||
            (enemy.transform.position.x <= enemy.leftPoint.position.x && !enemy.moveRight))
        {
            enemy.Flip();
        }
    }
}
```

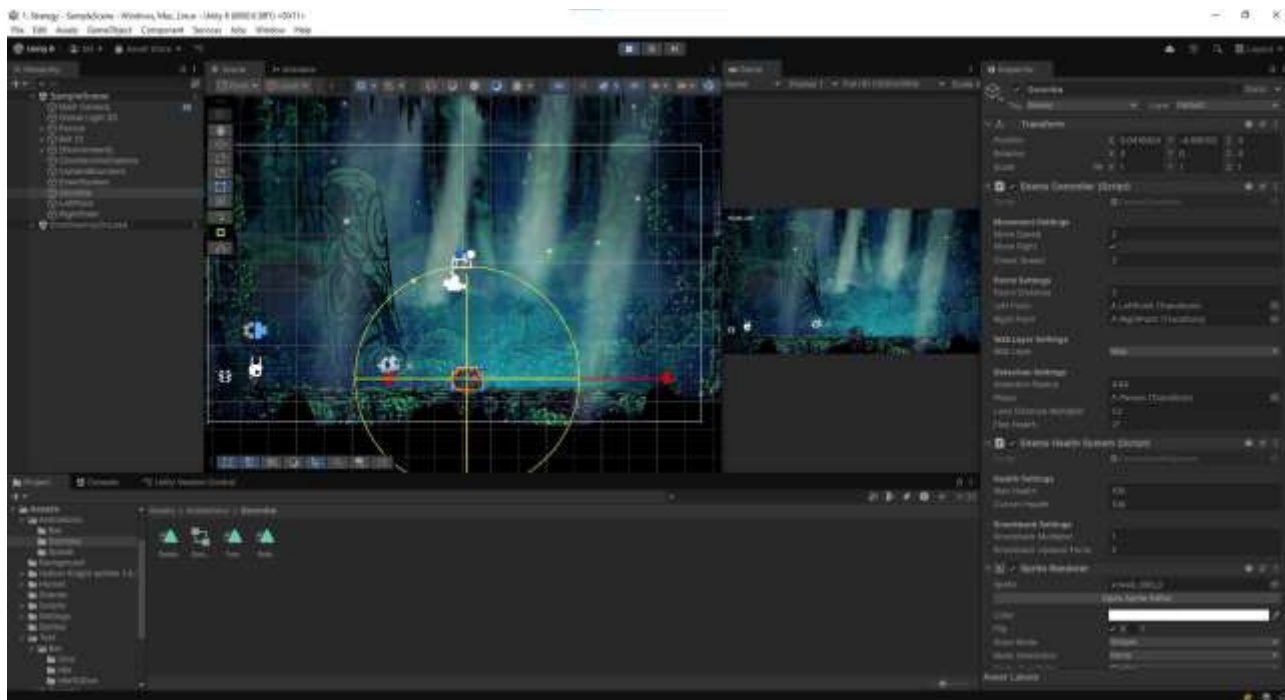


Рисунок 1. NPC патрулирует.

Стратегия преследования (ChaseBehavior)

По мере своего движения вокруг NPC существует определенная круговая граница, так называемая область видимости NPC. Если игрок находится внутри этого круга и при этом NPC развернут лицом к игроку, что мы, в свою очередь, расцениваем таким образом, словно NPC «заметил» игрока и «видит» его, NPC начинает его преследовать с увеличенной скоростью, непрерывно обновляя данные своего «зрения».

Листинг 3. ChaseBehavior.cs

```
using UnityEngine;

public class ChaseBehavior : IEnemyBehavior
{
    public void Execute(EnemyController enemy)
    {
        if (enemy.player is null)
            return;

        Vector2 targetPosition = new Vector2(enemy.player.position.x, enemy.transform.position.y);
        Vector2 directionToPlayer = (targetPosition - (Vector2)enemy.transform.position).normalized;
```

```

        enemy.RB.linearVelocity = new Vector2(directionToPlayer.x * enemy.chaseSpeed, enemy.RB.linearVelocity.y);

        if ((enemy.transform.position.x < enemy.player.position.x && !enemy.moveRight) ||
            (enemy.transform.position.x > enemy.player.position.x && enemy.moveRight))
        {
            enemy.Flip();
        }
    }
}

```



Рисунок 2. NPC преследует игрока.

Стратегия бегства от игрока при низком уровне здоровья NPC

Во много похож на прошлую стратегию и ее реализацию, но здесь NPC движется не в направлении игрока, а от него. NPC также ориентируется на свое «зрение», так что если игрок подойдет со спины, то NPC начнет убегать лишь тогда, когда при патрулировании повернется в сторону игрока и «заметит» его.

Листинг 4. FleeBehavior.cs

```

using UnityEngine;

public class FleeBehavior : IEnemyBehavior
{
    public void Execute(EnemyController enemy)
    {

```

```

{
    if (enemy.player is null)
        return;

    Vector2 targetPosition = new Vector2(enemy.player.position.x, enemy.trans-
form.position.y);
    Vector2 directionAwayFromPlayer = ((Vector2)enemy.transform.position - tar-
getPosition).normalized;

    enemy.RB.linearVelocity = new Vector2(directionAwayFromPlayer.x * en-
emy.chaseSpeed, enemy.RB.linearVelocity.y);

    if ((enemy.transform.position.x < enemy.player.position.x && en-
emy.moveRight) ||
        (enemy.transform.position.x > enemy.player.position.x && !en-
emy.moveRight))
    {
        enemy.Flip();
    }
}
}

```

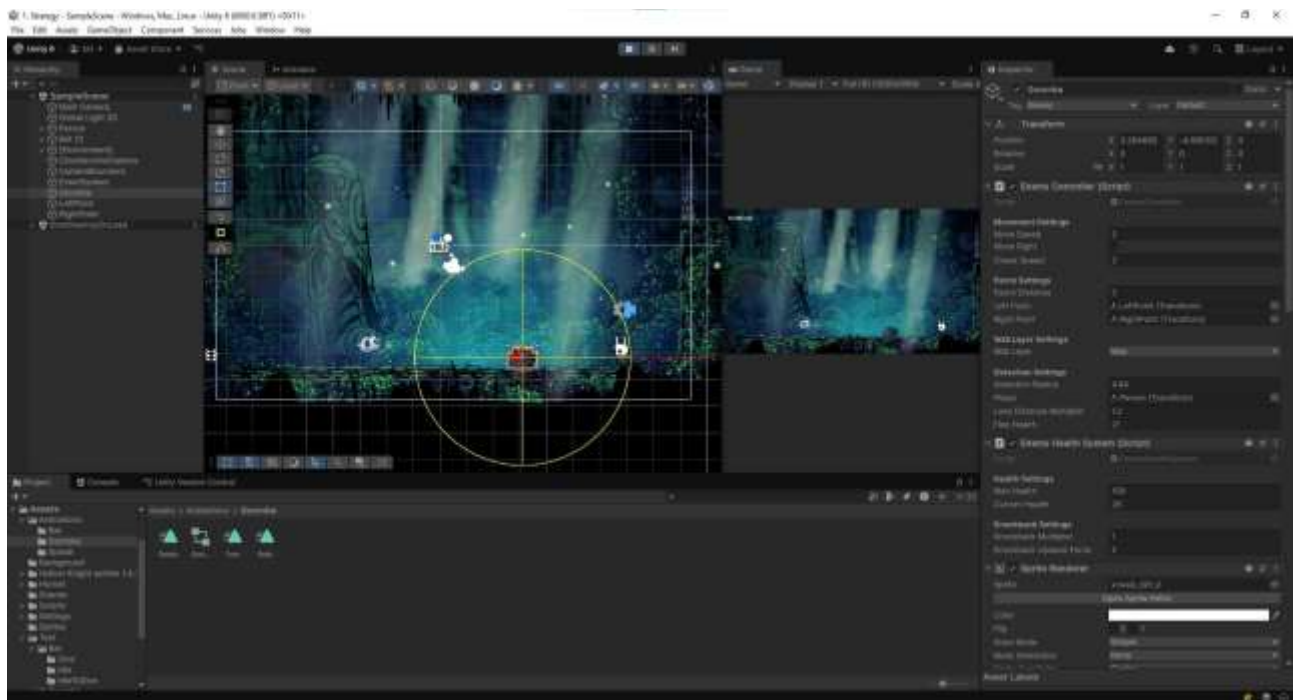


Рисунок 3. NPC убегает с низким уровнем здоровья.

Приложение

Листинг 5. EnemyController.cs

```

using System.Collections;
using UnityEngine;

```

```

public class EnemyController : MonoBehaviour
{
    [Header("Movement Settings")]
    public float moveSpeed = 2f;
    public bool moveRight = true;
    public float chaseSpeed = 4f;

    [Header("Patrol Settings")]
    public float patrolDistance = 5f;
    public Transform leftPoint;
    public Transform rightPoint;

    [Header("Wall Layer Settings")]
    public LayerMask wallLayer;

    [Header("Detection Settings")]
    public float detectionRadius = 5f;
    public Transform player;
    public float loseDistanceMultiplier = 1.5f;

    public int fleeHealth = 19;

    private Rigidbody2D rb;
    public Rigidbody2D RB { get { return rb; } }

    private SpriteRenderer spriteRenderer;
    private Animator animator;
    private bool isTurning = false;

    private IEnemyBehavior currentBehavior;
    private bool isChasing = false;

    private Vector2 initialPosition;

    private EnemyHealthSystem healthSystem;

    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
        spriteRenderer = GetComponent<SpriteRenderer>();
        animator = GetComponent<Animator>();
        healthSystem = GetComponent<EnemyHealthSystem>();

        initialPosition = transform.position;
        leftPoint.position = new Vector2(initialPosition.x - patrolDistance, initialPosition.y);
        rightPoint.position = new Vector2(initialPosition.x + patrolDistance, initialPosition.y);

        UpdateSpriteDirection();
    }
}

```



```

        currentBehavior = new PatrolBehavior();
    }

    private void Update()
    {
        if (healthSystem.currentHealth <= 0)
        {
            currentBehavior = null;
            return;
        }

        if (!isTurning)
        {
            CheckForPlayer();
            currentBehavior?.Execute(this);
        }
    }

    private void CheckForPlayer()
    {
        float distanceToPlayer = Vector2.Distance(transform.position, player.position);

        bool seesPlayer = false;
        if (distanceToPlayer <= detectionRadius)
        {
            bool isFacingPlayer = ((player.position.x - transform.position.x) > 0
&& moveRight) ||
                                ((player.position.x - transform.position.x) < 0
&& !moveRight);
            if (isFacingPlayer)
            {
                seesPlayer = true;
                isChasing = true;
            }
        }

        if (seesPlayer)
        {
            if (healthSystem.currentHealth < fleeHealth)
                currentBehavior = new FleeBehavior();
            else
                currentBehavior = new ChaseBehavior();

            return;
        }

        if (isChasing && distanceToPlayer > detectionRadius * loseDistanceMultiplier)
        {

```

```

        isChasing = false;
        ResetPatrolPoints();
        currentBehavior = new PatrolBehavior();
    }
}

private void ResetPatrolPoints()
{
    Vector2 currentPos = transform.position;
    leftPoint.position = new Vector2(currentPos.x - patrolDistance, current-
Pos.y);
    rightPoint.position = new Vector2(currentPos.x + patrolDistance, current-
Pos.y);
}

private void UpdateSpriteDirection()
{
    spriteRenderer.flipX = moveRight;
}

public void Flip()
{
    if (!isTurning)
        StartCoroutine(TurnCoroutine());
}

private IEnumerator TurnCoroutine()
{
    {
        isTurning = true;
        rb.linearVelocity = Vector2.zero;

        animator.SetBool("IsTurning", true);

        yield return new WaitForSeconds(GetAnimationLength("Turn"));

        moveRight = !moveRight;
        UpdateSpriteDirection();

        animator.SetBool("IsTurning", false);
        isTurning = false;
    }
}

private float GetAnimationLength(string animationName)
{
    {
        foreach (AnimationClip clip in animator.runtimeAnimatorController.anima-
tionClips)
        {
            if (clip.name == animationName)
                return clip.length;
        }
    }
}

```

```

        return 0.2f;
    }

    void OnCollisionEnter2D(Collision2D collision)
    {
        if (((1 << collision.gameObject.layer) & wallLayer) != 0)
            Flip();
    }

    private void OnDrawGizmosSelected()
    {
        if (leftPoint is not null && rightPoint is not null)
        {
            Gizmos.color = Color.red;
            Gizmos.DrawLine(leftPoint.position, rightPoint.position);

            Gizmos.DrawSphere(leftPoint.position, 0.2f);
            Gizmos.DrawSphere(rightPoint.position, 0.2f);
        }

        Gizmos.color = Color.yellow;
        Gizmos.DrawWireSphere(transform.position, detectionRadius);
    }
}

```