МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ

ФЕДЕРАЦИИ

ФГБОУ «ДонНТУ»

Институт КНТ

Кафедра ПИ

Лабораторная работа №4

по курсу: «Профессиональная практика программной инженерии»

по теме: «Создание самодокументирующегося кода»

Выполнил:

ст. гр. ПИ-21а

Тимощук Д.И.

Проверил:

асс. каф. ПИ

Филипишин Д.А.

Донецк 2024

Цель работы: научиться добавлять в программный код специальным образом оформление докблок-комментарии, для последующей автоматической генерации API reference, а также познакомиться с форматом оформления документации DocBook.

Задание к лабораторной работе:

Внедрить в ранее разработанный программный код докблоки, соответствующие выбранному языку программирования. Разработать DocBook и проверить корректность созданного DocBook файла с помощью средств редактора или одного из онлайн валидаторов. Автоматически сформировать справочное руководство для программистов и выслать вместе с отчётом. В руководстве должны присутствовать все модули, описанные в лабораторной работе №1.

Задание по лабораторной работе

1. Во всех файлах с исходным кодом добавить докблоки ко всем классам, методам, свойствам и функциям.

2. Закоммитить изменения в репозиторий.

3. Установить один из XML-редакторов (по желанию).

4. Оформить краткое описание разработанного продукта (не менее пяти абзацев).

5. Рассмотреть с примерами кода не менее трех сценариев использования вашего приложения.

6. Проверить корректность созданного DocBook файла с помощью средств редактора или одного из онлайн валидаторов, например, mashuosoft.com/docbook/validator

Содержание отчёта

1. Титульный лист.

2. Тема выбранного проекта.

3. Несколько примеров докблоков из программного кода разрабатываемого проекта подходящих к выбранному языку программирования, а также XML.

4. Скриншот полученного справочного руководства.

5. Ссылка на аккаунт.

6. Ссылка на репозиторий.

7. Руководство высылать вместе с отчётом.

Ссылка на репозиторий:

<https://github.com/maliiinka/PPII-Unity-/tree/main>

Ссылка на аккаунт:

https://github.com/maliiinka

Вариант 31

Конструктор видео игр (на примере Unity)

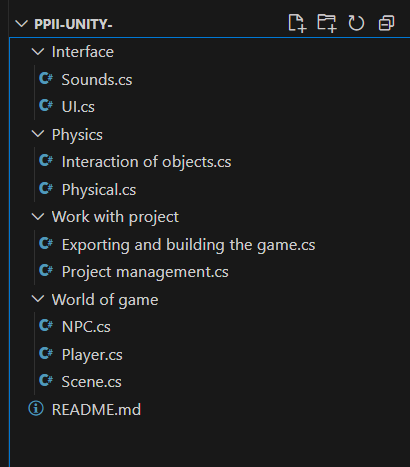


Рисунок 1- Структура проекта

**//Sounds.cs**

using System;

using System.Collections.Generic;

using NAudio.Wave;

namespace GameExample.Audio

{

/// <summary>

/// Класс для управления воспроизведением звуков в игре.

/// </summary>

public static class SoundManager

{

/// <summary>

/// Словарь для хранения загруженных звуков.

/// </summary>

private static readonly Dictionary<string, AudioFileReader> LoadedSounds = new Dictionary<string, AudioFileReader>();

/// <summary>

/// Воспроизводит звук из указанного файла.

/// </summary>

/// <param name="soundFilePath">Путь к файлу звука.</param>

public static void PlaySound(string soundFilePath)

{

if (!LoadedSounds.TryGetValue(soundFilePath, out var audioFileReader))

{

try

{

audioFileReader = new AudioFileReader(soundFilePath);

LoadedSounds[soundFilePath] = audioFileReader;

}

catch (Exception ex)

{

Console.WriteLine($"Ошибка загрузки файла: {ex.Message}");

return;

}

}

using (var outputDevice = new WaveOutEvent())

{

outputDevice.Init(audioFileReader);

outputDevice.Play();

}

}

/// <summary>

/// Останавливает воспроизведение указанного звука.

/// </summary>

/// <param name="soundFilePath">Путь к файлу звука.</param>

public static void StopSound(string soundFilePath)

{

if (LoadedSounds.TryGetValue(soundFilePath, out var audioFileReader))

{

audioFileReader.Stop();

}

}

/// <summary>

/// Освобождает ресурсы, закрывая все загруженные звуковые файлы.

/// </summary>

public static void DisposeAll()

{

foreach (var reader in LoadedSounds.Values)

{

reader.Dispose();

}

LoadedSounds.Clear();

}

}

}

**//UI.cs**

using System.Windows;

using System.Windows.Controls;

namespace Game.UI{

    /// <summary>

    /// Главное окно приложения.

    /// </summary>

    public partial class MainWindow : Window    {

        /// <summary>

        /// Конструктор главного окна. Инициализирует компоненты и загружает настройки.

        /// </summary>

        public MainWindow()        {

            InitializeComponent();

            LoadSettings();

        }

        /// <summary>

        /// Загружает начальные настройки окна.

        /// </summary>

        private void LoadSettings()        {

            Width = 800;

            Height = 600;

            Title = "Игра";

        }

        /// <summary>

        /// Обработчик события нажатия на кнопку старта игры.

        /// </summary>

        private void Button\_Start\_Click(object sender, RoutedEventArgs e)        {

            MessageBox.Show("Игра началась!");

        }

        /// <summary>

        /// Обработчик события открытия окна настроек.

        /// </summary>

        private void Button\_Options\_Click(object sender, RoutedEventArgs e)

        {

            OptionsWindow optionsWindow = new OptionsWindow            {

                Owner = this

            };

            optionsWindow.ShowDialog();

        }

        /// <summary>

        /// Обработчик события выхода из приложения.

        /// </summary>

        private void Button\_Quit\_Click(object sender, RoutedEventArgs e)        {

            Close();

        }

    }

    /// <summary>

    /// Окно настроек приложения.

    /// </summary>

    public partial class OptionsWindow : Window

    {

        /// <summary>

        /// Конструктор окна настроек. Инициализирует компоненты и загружает текущие настройки.

        /// </summary>

        public OptionsWindow()        {

            InitializeComponent();

            LoadCurrentOptions();

        }

        /// <summary>

        /// Загружает текущие настройки пользователя.

        /// </summary>

        private void LoadCurrentOptions()        {

            CheckBox\_Fullscreen.IsChecked = Properties.Settings.Default.Fullscreen;

            Slider\_Volume.Value = Properties.Settings.Default.Volume;

        }

        /// <summary>

        /// Сохраняет настройки и закрывает окно.

        /// </summary>

        private void Button\_Save\_Click(object sender, RoutedEventArgs e)        {

            Properties.Settings.Default.Fullscreen = (bool)CheckBox\_Fullscreen.IsChecked;

            Properties.Settings.Default.Volume = (double)Slider\_Volume.Value;

            Properties.Settings.Default.Save();

            MessageBox.Show("Настройки сохранены!");

            DialogResult = true;

        }

        /// <summary>

        /// Отменяет изменения и закрывает окно.

        /// </summary>

        private void Button\_Cancel\_Click(object sender, RoutedEventArgs e)

        {

            DialogResult = false;

        }

    }

}

**//Interaction of oblects.cs**

using Jitter.Dynamics;

using Jitter.Collision;

using Jitter.LinearMath;

namespace GameExample.Physics

{

/// <summary>

/// Физический движок для обработки симуляции физических объектов.

/// </summary>

public class PhysicsEngine

{

/// <summary>

/// Объект физического мира.

/// </summary>

private World \_world;

/// <summary>

/// Размер шага симуляции.

/// </summary>

private const float TimeStep = 1f / 60f;

/// <summary>

/// Конструктор физического движка. Инициализирует физический мир.

/// </summary>

public PhysicsEngine()

{

\_world = new World();

\_world.Gravity = new JVector(0, -9.81f, 0);

}

/// <summary>

/// Добавляет жесткое тело в физический мир.

/// </summary>

/// <param name="shape">Форма тела.</param>

/// <param name="mass">Масса тела.</param>

/// <param name="position">Начальная позиция тела.</param>

/// <returns>Созданное жесткое тело.</returns>

public RigidBody AddRigidBody(JShape shape, float mass, JVector position)

{

var body = new RigidBody(shape, mass);

body.Position = position;

\_world.AddBody(body);

return body;

}

/// <summary>

/// Обновляет состояние физического мира.

/// </summary>

public void UpdatePhysics()

{

\_world.Step(TimeStep);

}

/// <summary>

/// Удаляет жесткое тело из физического мира.

/// </summary>

/// <param name="body">Тело для удаления.</param>

public void RemoveRigidBody(RigidBody body)

{

\_world.RemoveBody(body);

}

/// <summary>

/// Очищает физический мир и освобождает ресурсы.

/// </summary>

public void Dispose()

{

\_world.Clear();

}

}

}

**//Physical.cs**

using Jitter.Dynamics;

using Jitter.Collision;

using Jitter.LinearMath;

namespace GameExample.Physics{

/// <summary>

/// Представляет физический объект в игровом мире.

/// </summary>

public class PhysicalObject {

/// <summary>

/// Жесткое тело Jitter Physics.

/// </summary>

public RigidBody Body { get; private set; }

/// <summary>

/// Форма коллизии (например, сфера, ящик).

/// </summary>

public CollisionShape Shape { get; private set; }

/// <summary>

/// Вектор силы, приложенной к объекту.

/// </summary>

public JVector Force { get; set; }

/// <summary>

/// Вектор скорости объекта.

/// </summary>

public JVector Velocity { get; set; }

/// <summary>

/// Масштаб объекта.

/// </summary>

public float Scale { get; set; }

/// <summary>

/// Создает новый физический объект.

/// </summary>

/// <param name="shape">Форма коллизии объекта.</param>

/// <param name="mass">Масса объекта.</param>

/// <param name="position">Начальная позиция объекта.</param>

/// <param name="force">Начальная сила, приложенная к объекту.</param>

/// <param name="velocity">Начальная скорость объекта.</param>

public PhysicalObject(CollisionShape shape, float mass, JVector position, JVector force, JVector velocity){

Shape = shape;

Body = new RigidBody(shape, mass);

Body.Position = position;

Force = force;

Velocity = velocity;

Scale = 1.0f;

}

/// <summary>

/// Применяет силу к телу.

/// </summary>

public void ApplyForce() {

Body.ApplyImpulse(Force);

}

/// <summary>

/// Перемещает объект с заданной скоростью.

/// </summary>

public void Move() { Body.Velocity = Velocity;

}

/// <summary>

/// Обновляет положение объекта в мире.

/// </summary>

public void UpdatePosition() {

JVector newPosition = Body.Position + Velocity \* TimeSpan.FromSeconds(1.0f / 60.0f).TotalSeconds;

Body.Position = newPosition;

}

/// <summary>

/// Устанавливает начальную позицию объекта.

/// </summary>

/// <param name="initialPosition">Начальная позиция.</param>

public void SetInitialPosition(JVector initialPosition) {

Body.Position = initialPosition;

}

/// <summary>

/// Проверяет столкновение с другим физическим объектом.

/// </summary>

/// <param name="other">Другой физический объект.</param>

/// <returns>True, если объекты сталкиваются, иначе False.</returns>

public bool IsCollidingWith(PhysicalObject other) {

return Body.Shape.CollidesWith(other.Body.Shape);

}

/// <summary>

/// Отображает объект на экране (заглушка).

/// </summary>

public void Render() {

// Вызывается рендеринг объекта на основе его текущей позиции и ориентации.

}

}

}

**//Exporting and building the game.cs**

using System; using System.IO;

using System.Diagnostics;

namespace Game.BuildSystem{

/// <summary>

/// Управляет процессом сборки игрового проекта.

/// </summary>

public class BuildManager {

/// <summary>

/// Директория исходников игры.

/// </summary>

private string SourceDirectory { get; set; }

/// <summary>

/// Директория для выходных файлов.

/// </summary>

private string OutputDirectory { get; set; }

/// <summary>

/// Имя исполняемого файла.

/// </summary>

private string ExecutableName { get; set; }

/// <summary>

/// Создает экземпляр менеджера сборки.

/// </summary>

/// <param name="sourceDir">Путь к директории исходного кода.</param>

/// <param name="outputDir">Путь к директории выходных файлов.</param>

/// <param name="exeName">Имя исполняемого файла.</param>

public BuildManager(string sourceDir, string outputDir, string exeName) {

SourceDirectory = sourceDir;

OutputDirectory = outputDir;

ExecutableName = exeName;}

/// <summary>

/// Выполняет сборку проекта.

/// </summary>

public void BuildProject() {

LogMessage("Начало сборки проекта...");

CompileProject();

CopyRequiredFiles();

ArchiveProject();

LogMessage("Сборка проекта завершена."); }

/// <summary>

/// Компилирует проект с помощью MSBuild.

/// </summary>

private void CompileProject() {

LogMessage("Компиляция проекта...");

ProcessStartInfo processInfo = new ProcessStartInfo{ FileName = "msbuild.exe",

Arguments = Path.Combine(SourceDirectory, "MyGame.sln") + " /p:Configuration=Release" };

Process compilerProcess = new Process { StartInfo = processInfo };

compilerProcess.Start();

compilerProcess.WaitForExit();

LogMessage("Компиляция проекта завершена."); }

/// <summary>

/// Копирует необходимые файлы в выходную директорию.

/// </summary>

private void CopyRequiredFiles() {

LogMessage("Копирование необходимых файлов..."); string binFolder = Path.Combine(SourceDirectory, "bin", "Release"); string[] filesToCopy = Directory.GetFiles(binFolder, "\*.\*", SearchOption.AllDirectories);

foreach (string file in filesToCopy){

string destinationPath = Path.Combine(OutputDirectory, Path.GetFileName(file));

File.Copy(file, destinationPath, true); }

LogMessage("Копирование необходимых файлов завершено."); }

/// <summary>

/// Архивирует собранный проект в ZIP-файл. /// </summary>

private void ArchiveProject() {

LogMessage("Архивация проекта..."); ZipFile.CreateFromDirectory(OutputDirectory, Path.Combine(OutputDirectory, "MyGame.zip"));

LogMessage("Архивация проекта завершена."); }

/// <summary>

/// Выводит сообщение в журнал сборки.

/// </summary>

/// <param name="message">Сообщение для логирования.</param>

private void LogMessage(string message) { Console.WriteLine(message);

} }

/// <summary>

/// Точка входа в программу сборки игры.

/// </summary>

class Program {

static void Main(string[] args)

{ string sourceDir = @"C:\\Projects\\MyGame\\Source";

string outputDir = @"C:\\Projects\\MyGame\\Output";

string executableName = MyGame.exe";

BuildManager buildManager = new BuildManager(sourceDir, outputDir, executableName);

buildManager.BuildProject();

} } }

**//Project management.cs**

using System;

using System.Collections.Generic;

namespace ProjectManagement{

/// <summary>

/// Представляет проект, содержащий задачи и ресурсы.

/// </summary>

public class Project{

public Guid Id { get; set; }

public string Name { get; set; }

public string Description { get; set; }

public DateTime StartDate { get; set; }

public DateTime EndDate { get; set; }

public List<Task> Tasks { get; set; }

public List<Resource> Resources { get; set; } /// <summary>

/// Создаёт новый проект.

/// </summary>

public Project(string name, string description, DateTime startDate, DateTime endDate){

Id = Guid.NewGuid();

Name = name;

Description = description;

StartDate = startDate;

EndDate = endDate;

Tasks = new List<Task>();

Resources = new List<Resource>();}

/// <summary>

/// Добавляет задачу в проект.

/// </summary>

public void AddTask(Task task) {

Tasks.Add(task);

}

/// <summary>

/// Добавляет ресурс в проект.

/// </summary>

public void AddResource(Resource resource){

Resources.Add(resource);}

/// <summary>

/// Назначает ресурс на задачу.

/// </summary>

public void AssignResourceToTask(Resource resource, Task task){

if (Resources.Contains(resource) && Tasks.Contains(task)){ task.AssignedResources.Add(resource);

} }

public override string ToString() {

return $"{Id}: {Name} ({StartDate} - {EndDate})"; } }

/// <summary>

/// Представляет задачу в проекте.

/// </summary>

public class Task {

public Guid Id { get; set; }

public string Name { get; set; }

public string Description { get; set; }

public DateTime StartDate { get; set; }

public DateTime EndDate { get; set; }

public List<Resource> AssignedResources { get; set; }

/// <summary>

/// Создаёт новую задачу.

/// </summary>

public Task(string name, string description, DateTime startDate, DateTime endDate) {

Id = Guid.NewGuid();

Name = name;

Description = description;

StartDate = startDate;

EndDate = endDate;

AssignedResources = new List<Resource>(); }

public override string ToString() {

return $"{Id}: {Name} ({StartDate} - {EndDate})"; } }

/// <summary>

/// Представляет ресурс, который может быть назначен на задачи.

/// </summary>

public class Resource{

public Guid Id { get; set; }

public string Name { get; set; }

public string Role { get; set; }

/// <summary>

/// Создаёт новый ресурс.

/// </summary>

public Resource(string name, string role){

Id = Guid.NewGuid();

Name = name;

Role = role; }

public override string ToString() {

return $"{Id}: {Name} ({Role})";

} }

/// <summary>

/// Точка входа в программу.

/// </summary>

public class Program {

public static void Main(string[] args) {

// Создаём проект

Project project = new Project(

"Пример проекта",

"Описание проекта",

DateTime.Today,

DateTime.Today.AddDays(30)); // Создаём задачи

Task task1 = new Task(

"Задача 1",

"Описание задачи 1",

DateTime.Today,

DateTime.Today.AddDays(10));

Task task2 = new Task(

"Задача 2",

"Описание задачи 2",

DateTime.Today.AddDays(11),

DateTime.Today.AddDays(20));

// Создаём ресурсы

Resource resource1 = new Resource("Иван Иванов", "Разработчик");

Resource resource2 = new Resource("Анна Петрова", "Тестировщик");

// Добавляем задачи и ресурсы в проект

project.AddTask(task1);

project.AddTask(task2);

project.AddResource(resource1);

project.AddResource(resource2);

// Назначаем ресурсы на задачи

project.AssignResourceToTask(resource1, task1);

project.AssignResourceToTask(resource2, task2);

// Выводим информацию о проекте

Console.WriteLine(project.ToString());

foreach (var task in project.Tasks)

{

Console.WriteLine($"\t{task.ToString()}");

foreach (var resource in task.AssignedResources)

{

Console.WriteLine($"\t\t{resource.ToString()}");

} } }} }

**//NPC.cs**

using System;

namespace GameExample.Entities{

/// <summary>

/// Класс, представляющий NPC (неигрового персонажа) в игре.

/// </summary>

public class NPC : Entity {

/// <summary>

/// Уникальный идентификатор NPC.

/// </summary>

public Guid Id { get; private set; }

/// <summary>

/// Имя NPC.

/// </summary>

public string Name { get; private set; }

/// <summary>

/// Текущее состояние NPC (Idle, Moving, Attacking).

/// </summary>

public NPCState State { get; private set; }

/// <summary>

/// Цель NPC (может быть координатой или другим объектом).

/// </summary>

public object Target { get; private set; }

/// <summary>

/// Текущая позиция NPC в пространстве.

/// </summary>

public Vector2D Position { get; private set; }

/// <summary>

/// Скорость перемещения NPC.

/// </summary>

public double Speed { get; private set; }

/// <summary>

/// Урон, наносимый NPC при атаке.

/// </summary>

public double Damage { get; private set; }

/// <summary>

/// Создаёт новый объект NPC.

/// </summary>

/// <param name="name">Имя NPC.</param>

/// <param name="position">Начальная позиция NPC.</param>

/// <param name="speed">Скорость перемещения NPC.</param>

/// <param name="damage">Наносимый урон.</param>

public NPC(string name, Vector2D position, double speed, double damage) {

Id = Guid.NewGuid();

Name = name;

Position = position;

Speed = speed;

Damage = damage;

State = NPCState.Idle; }

/// <summary>

/// Обновляет состояние NPC в зависимости от текущего состояния.

/// </summary>

/// <param name="deltaTime">Прошедшее время в секундах.</param>

public void Update(double deltaTime){

switch (State) {

case NPCState.Moving:

Move(deltaTime);

break;

case NPCState.Attacking:

Attack(Target as Entity);

break;

default:

break; } }

/// <summary>

/// Выполняет перемещение NPC к цели.

/// </summary>

/// <param name="deltaTime">Прошедшее время в секундах.</param>

private void Move(double deltaTime) {

if (Target is Vector2D targetPosition) { Vector2D direction = targetPosition - Position;

if (direction.MagnitudeSquared > 1e-6) // Проверка, чтобы избежать деления на 0 { direction.Normalize();

Position += direction \* Speed \* deltaTime;

if ((targetPosition - Position).MagnitudeSquared < 1e-6) {

State = NPCState.Idle;

} } }

/// <summary>

/// Выполняет атаку NPC.

/// </summary>

/// <param name="targetEntity">Цель атаки (другой объект).</param>

private void Attack(Entity targetEntity) {

if (targetEntity != null){

targetEntity.Health -= Damage;

} }

/// <summary>

/// Структура для представления вектора в 2D-пространстве.

/// </summary>

public struct Vector2D {

/// <summary>

/// Координата X.

/// </summary>

public double X { get; private set; }

/// <summary>

/// Координата Y.

/// </summary>

public double Y { get; private set; }

/// <summary>

/// Создаёт новый 2D-вектор.

/// </summary>

/// <param name="x">Координата X.</param>

/// <param name="y">Координата Y.</param>

public Vector2D(double x, double y){

X = x;

Y = y; }

/// <summary>

/// Оператор вычитания двух векторов.

/// </summary>

public static Vector2D operator -(Vector2D a, Vector2D b) {

return new Vector2D(a.X - b.X, a.Y - b.Y); }

/// <summary>

/// Оператор сложения двух векторов.

/// </summary>

public static Vector2D operator +(Vector2D a, Vector2D b) {

return new Vector2D(a.X + b.X, a.Y + b.Y); }

/// <summary>

/// Оператор умножения вектора на скаляр.

/// </summary>

public static Vector2D operator \*(Vector2D a, double scalar) {

return new Vector2D(a.X \* scalar, a.Y \* scalar); }

/// <summary>

/// Нормализует вектор (приводит его длину к 1).

/// </summary>

public void Normalize() {

double magnitude = Magnitude;

if (magnitude == 0) return; // Защита от деления на 0

X /= magnitude;

Y /= magnitude; }

/// <summary>

/// Длина вектора.

/// </summary>

public double Magnitude => Math.Sqrt(X \* X + Y \* Y);

/// <summary>

/// Квадрат длины вектора (для оптимизированных вычислений).

/// </summary>

public double MagnitudeSquared => X \* X + Y \* Y; }

/// <summary>

/// Перечисление состояний NPC.

/// </summary>

public enum NPCState {

/// <summary>

/// NPC бездействует.

/// </summary>

Idle,

/// <summary>

/// NPC движется к цели.

/// </summary>

Moving,

/// <summary>

/// NPC атакует цель.

/// </summary>

Attacking

}

}

}

//Player.cs

using System;

using System.Collections.Generic;

namespace GameExample{

/// <summary>

/// Класс, представляющий игрока в игре.

/// </summary>

public class Player {

// Поля класса

private string \_name;

private int \_healthPoints;

private int \_experiencePoints;

private int \_level;

private DateTime \_lastLogin;

private Inventory \_inventory;

/// <summary>

/// Создаёт нового игрока с указанным именем.

/// </summary>

/// <param name="name">Имя игрока.</param>

public Player(string name) {

\_name = name;

\_healthPoints = 100;

\_experiencePoints = 0;

\_level = 1;

\_lastLogin = DateTime.Now;

\_inventory = new Inventory(); }

/// <summary>

/// Имя игрока.

/// </summary>

public string Name {

get { return \_name; }

set { \_name = value; } }

/// <summary>

/// Очки здоровья игрока (не могут быть меньше 0).

/// </summary>

public int HealthPoints {

get { return \_healthPoints; }

set { \_healthPoints = Math.Max(0, value); } }

/// <summary>

/// Очки опыта игрока (не могут быть меньше 0).

/// </summary>

public int ExperiencePoints {

get { return \_experiencePoints; }

set { \_experiencePoints = Math.Max(0, value); } }

/// <summary>

/// Уровень игрока (не может быть меньше 1).

/// </summary>

public int Level

{

get { return \_level; }

set { \_level = Math.Max(1, value); } }

/// <summary>

/// Дата и время последнего входа игрока в игру.

/// </summary>

public DateTime LastLogin {

get { return \_lastLogin; }

set { \_lastLogin = value; } }

/// <summary>

/// Добавляет игроку опыт и проверяет повышение уровня.

/// </summary>

/// <param name="amount">Количество полученного опыта.</param>

public void GainExperience(int amount){

\_experiencePoints += amount;

CheckLevelUp(); }

/// <summary>

/// Проверяет, достиг ли игрок нового уровня, и при необходимости повышает его.

/// </summary>

private void CheckLevelUp() {

if (\_experiencePoints >= 100 \* \_level) {

\_level++;

Console.WriteLine($"Поздравляем! Вы достигли уровня {\_level}.");

} }

/// <summary>

/// Наносит урон игроку.

/// </summary>

/// <param name="damage">Количество полученного урона.</param>

public void TakeDamage(int damage) {

\_healthPoints -= damage;

if (\_healthPoints <= 0) {

Die();

} }

/// <summary>

/// Выполняет действия при смерти игрока.

/// </summary>

private void Die()

{

Console.WriteLine($"{\_name} погиб.");

ResetHealth();

}

/// <summary>

/// Восстанавливает здоровье игрока после смерти.

/// </summary>

private void ResetHealth()

{

\_healthPoints = 100;

}

/// <summary>

/// Возвращает строковое представление игрока.

/// </summary>

/// <returns>Строка с информацией об игроке.</returns>

public override string ToString()

{

return $"Имя: {\_name}, Здоровье: {\_healthPoints}, Уровень: {\_level}";

}

/// <summary>

/// Класс инвентаря игрока, содержащий предметы.

/// </summary>

public class Inventory

{

private List<Item> \_items;

/// <summary>

/// Создаёт новый инвентарь.

/// </summary>

public Inventory()

{

\_items = new List<Item>();

}

/// <summary>

/// Добавляет предмет в инвентарь.

/// </summary>

/// <param name="item">Предмет, который нужно добавить.</param>

public void AddItem(Item item)

{

\_items.Add(item);

}

/// <summary>

/// Получает предмет из инвентаря по его имени.

/// </summary>

/// <param name="name">Имя предмета.</param>

/// <returns>Предмет, если найден, иначе null.</returns>

public Item GetItemByName(string name)

{

return \_items.Find(i => i.Name == name);

}

/// <summary>

/// Удаляет предмет из инвентаря.

/// </summary>

/// <param name="item">Предмет, который нужно удалить.</param>

public void RemoveItem(Item item)

{

\_items.Remove(item);

}

}

/// <summary>

/// Класс, представляющий предмет в инвентаре.

/// </summary>

public class Item

{

/// <summary>

/// Имя предмета.

/// </summary>

public string Name { get; set; }

/// <summary>

/// Ценность предмета.

/// </summary>

public int Value { get; set; }

/// <summary>

/// Создаёт новый предмет.

/// </summary>

/// <param name="name">Имя предмета.</param>

/// <param name="value">Ценность предмета.</param>

public Item(string name, int value)

{

Name = name;

Value = value;

}

}

}

}

//Scene.cs

using System;

using System.Collections.Generic;

namespace GameExample

{

/// <summary>

/// Класс, представляющий игровую сцену.

/// </summary>

public class Scene

{

/// <summary>

/// Название сцены.

/// </summary>

public string Name { get; set; }

/// <summary>

/// Список объектов, находящихся на сцене.

/// </summary>

public List<GameObject> Objects { get; set; }

/// <summary>

/// Список активных игроков на сцене.

/// </summary>

public List<Player> Players { get; set; }

/// <summary>

/// События, происходящие на сцене.

/// </summary>

public Dictionary<string, Event> Events { get; set; }

/// <summary>

/// Создает новую игровую сцену.

/// </summary>

/// <param name="name">Название сцены.</param>

public Scene(string name)

{

Name = name;

Objects = new List<GameObject>();

Players = new List<Player>();

Events = new Dictionary<string, Event>();

}

/// <summary>

/// Добавляет объект на сцену.

/// </summary>

/// <param name="obj">Игровой объект.</param>

public void AddObject(GameObject obj)

{

Objects.Add(obj);

}

/// <summary>

/// Удаляет объект со сцены.

/// </summary>

/// <param name="obj">Игровой объект.</param>

public void RemoveObject(GameObject obj)

{

Objects.Remove(obj);

}

/// <summary>

/// Добавляет игрока на сцену.

/// </summary>

/// <param name="player">Игрок.</param>

public void AddPlayer(Player player)

{

Players.Add(player);

}

/// <summary>

/// Удаляет игрока со сцены.

/// </summary>

/// <param name="player">Игрок.</param>

public void RemovePlayer(Player player)

{

Players.Remove(player);

}

/// <summary>

/// Запускает событие на сцене.

/// </summary>

/// <param name="eventName">Название события.</param>

public void TriggerEvent(string eventName)

{

if (Events.ContainsKey(eventName))

{

Events[eventName].Execute();

}

}

/// <summary>

/// Класс, представляющий игровой объект на сцене.

/// </summary>

public class GameObject

{

/// <summary>

/// Название объекта.

/// </summary>

public string Name { get; set; }

/// <summary>

/// Позиция объекта на сцене.

/// </summary>

public Vector Position { get; set; }

/// <summary>

/// Создает новый игровой объект.

/// </summary>

/// <param name="name">Название объекта.</param>

/// <param name="position">Позиция объекта.</param>

public GameObject(string name, Vector position)

{

Name = name;

Position = position;

} }

/// <summary>

/// Структура, представляющая 2D-вектор.

/// </summary>

public struct Vector {

/// <summary>

/// Координата X.

/// </summary>

public float X { get; set; }

/// <summary>

/// Координата Y.

/// </summary>

public float Y { get; set; }

/// <summary>

/// Создает новый 2D-вектор.

/// </summary>

/// <param name="x">Координата X.</param>

/// <param name="y">Координата Y.</param>

public Vector(float x, float y)

{

X = x;

Y = y;

}

}

/// <summary>

/// Класс, представляющий событие на сцене.

/// </summary>

public class Event {

/// <summary>

/// Описание события.

/// </summary>

public string Description { get; set; }

/// <summary>

/// Действие, выполняемое при запуске события.

/// </summary>

public Action ExecuteAction { get; set; }

/// <summary>

/// Создает новое событие.

/// </summary>

/// <param name="description">Описание события.</param>

/// <param name="action">Действие события.</param>

public Event(string description, Action action){

Description = description;

ExecuteAction = action;

}

/// <summary>

/// Выполняет действие события.

/// </summary>

public void Execute()

{

ExecuteAction?.Invoke();

}

}

}

}

**DokBook Документация**

<?xml version="1.0" encoding="UTF-8"?>

<book xmlns="http://docbook.org/ns/docbook" version="5.0">

<title>Документация проекта Unity</title>

<chapter>

<title>Краткое описание проекта Unity</title>

<para>

Проект Unity представляет собой фреймворк для разработки игровых приложений.

Он включает в себя инструменты для управления сценами, объектами, персонажами и взаимодействиями между ними. В основе системы лежит объектно-ориентированный подход, который обеспечивает гибкость и расширяемость.

</para>

<para>

Основные возможности:

<itemizedlist>

<listitem>Управление игровыми проектами (создание, экспорт, управление зависимостями).</listitem>

<listitem>Работа со сценами и объектами.</listitem>

<listitem>Создание и настройка персонажей.</listitem>

<listitem>Физика, искусственный интеллект и взаимодействие объектов.</listitem>

<listitem>Пользовательский интерфейс и обработка пользовательского ввода.</listitem>

<listitem>Экспорт, сборка и тестирование игр.</listitem>

</itemizedlist>

</para>

</chapter>

<chapter>

<title>Сценарии использования</title>

<section>

<title>Сценарий 1: Создание и управление сценой</title>

<para>

В этом сценарии создается новая игровая сцена, добавляются объекты и игроки.

</para>

<programlisting language="csharp">

using GameExample;

Scene mainScene = new Scene("Главная сцена");

// Добавление объекта на сцену

Scene.GameObject tree = new Scene.GameObject("Дерево", new Scene.Vector(10, 20));

mainScene.AddObject(tree);

// Добавление игрока

Player player = new Player("Герой");

mainScene.AddPlayer(player);

</programlisting>

<para>

В результате выполнения кода создается сцена, на которой размещены объект "Дерево" и игрок "Герой".

</para>

</section>

<section>

<title>Сценарий 2: Управление персонажем</title>

<para>

В этом сценарии создается персонаж, настраиваются его характеристики и он получает опыт.

</para>

<programlisting language="csharp">

using GameExample;

// Создание персонажа

Player player = new Player("Воин");

// Нанесение урона

player.TakeDamage(30);

// Получение опыта

player.GainExperience(120);

</programlisting>

<para>

В результате выполнения кода персонаж "Воин" получает урон в 30 единиц и зарабатывает 120 опыта.

Если опыта достаточно, персонаж поднимет уровень.

</para>

</section>

<section>

<title>Сценарий 3: Взаимодействие с объектами</title>

<para>

В этом сценарии персонаж получает предмет из инвентаря сцены.

</para>

<programlisting language="csharp">

using GameExample;

// Создание сцены и персонажа

Scene gameScene = new Scene("Подземелье");

Player hero = new Player("Рыцарь");

gameScene.AddPlayer(hero);

// Создание предмета и добавление его в инвентарь

Player.Item sword = new Player.Item("Меч", 50);

hero.Inventory.AddItem(sword);

// Получение предмета из инвентаря

Player.Item retrievedItem = hero.Inventory.GetItemByName("Меч");

</programlisting>

<para>

В результате выполнения кода персонаж "Рыцарь" получает "Меч" из инвентаря.

</para>

</section>

</chapter>

</book>

**DOCX-версия документации, сгенерированная на сайте** [**https://pandoc.org/try/**](https://pandoc.org/try/)

Документация проекта Unity

# Краткое описание проекта Unity

Проект Unity представляет собой фреймворк для разработки игровых приложений. Он включает в себя инструменты для управления сценами, объектами, персонажами и взаимодействиями между ними. В основе системы лежит объектно-ориентированный подход, который обеспечивает гибкость и расширяемость.

Основные возможности:

* Управление игровыми проектами (создание, экспорт, управление зависимостями).
* Работа со сценами и объектами.
* Создание и настройка персонажей.
* Физика, искусственный интеллект и взаимодействие объектов.
* Пользовательский интерфейс и обработка пользовательского ввода.
* Экспорт, сборка и тестирование игр.

# Сценарии использования

## Сценарий 1: Создание и управление сценой

В этом сценарии создается новая игровая сцена, добавляются объекты и игроки.

using GameExample;  
  
Scene mainScene = new Scene("Главная сцена");  
  
// Добавление объекта на сцену  
Scene.GameObject tree = new Scene.GameObject("Дерево", new Scene.Vector(10, 20));  
mainScene.AddObject(tree);  
  
// Добавление игрока  
Player player = new Player("Герой");  
mainScene.AddPlayer(player);

В результате выполнения кода создается сцена, на которой размещены объект "Дерево" и игрок "Герой".

## Сценарий 2: Управление персонажем

В этом сценарии создается персонаж, настраиваются его характеристики и он получает опыт.

using GameExample;  
  
// Создание персонажа  
Player player = new Player("Воин");  
  
// Нанесение урона  
player.TakeDamage(30);  
  
// Получение опыта  
player.GainExperience(120);

В результате выполнения кода персонаж "Воин" получает урон в 30 единиц и зарабатывает 120 опыта. Если опыта достаточно, персонаж поднимет уровень.

## Сценарий 3: Взаимодействие с объектами

В этом сценарии персонаж получает предмет из инвентаря сцены.

using GameExample;  
  
// Создание сцены и персонажа  
Scene gameScene = new Scene("Подземелье");  
Player hero = new Player("Рыцарь");  
gameScene.AddPlayer(hero);  
  
// Создание предмета и добавление его в инвентарь  
Player.Item sword = new Player.Item("Меч", 50);  
hero.Inventory.AddItem(sword);  
  
// Получение предмета из инвентаря  
Player.Item retrievedItem = hero.Inventory.GetItemByName("Меч");

В результате выполнения кода персонаж "Рыцарь" получает "Меч" из инвентаря.