# Лабораторная работа №3: Абстракция, наследование, полиморфизм.

**Цель работы:**

– Знакомство с основными принципами ООП.

**Теоретическая информация.**

**Абстракция -** это процесс выделения существенных характеристик объекта, которые отличают его от других объектов, и сокрытие менее важных деталей. Абстракция позволяет сосредоточиться на общих чертах и интерфейсах, упрощая сложные системы.

В качестве примера возьмем первое задание первой лабораторной работы. В этом задании требовалось разработать два класса для разных фигур – треугольника и прямоугольника. Два данных класса по сути имеют одинаковые параметры – оба обладают определенным набором точек и используют одинаковые функции.

Вместо реализации отдельных классов лучшим решением будет применить принцип абстракции. Для этого может быть создан класс фигуры Shape и затем на основе этого класса уже можно создавать отдельные фигуры. Затем можно добавлять новые фигуры, не изменяя существующий код, просто используя этот базовый класс Shape.

Создание абстрактного класса начинается с использования ключевого слова abstract:

```
abstract class Shape //абстрактный класс обозначается ключевым словом abstract
    {
        Point2D[] points; //набор точек фигуры

        //конструктор класса
        public Shape(Point2D[] points)
        {
            this.points = points;
        }

        //метод получения точек
        public Point2D[] getPoints() { return points; }

        //абстрактный метод расчета площади фигуры
        //абстрактный метод также обозначается ключом abstract
        public abstract double getArea();
    }
```

В данном примере можно заметить следующие элементы реализации абстрактного класса:

– Атрибуты не имеют приставки абстракции потому что они всегда остаются идентичными и неизменными как для абстрактного класса, так и для основанных на нем. В то же время если использовать вместо стандартных атрибутов свойства – свойства могут иметь абстрактную реализацию.

– Абстрактные классы могут содержать методы с реализацией так и без. В данном классе Shape реализован метод получения точек, соответственно этот же метод будет использоваться в таком же виде в классах, основанных на классе Shape. В то же время метод расчета площади не реализуется, отследить это можно по ключевому слову abstract. Такой метод строго определяет сигнатуру функции: имя метода, возвращаемый тип данных и передаваемые в метод данные.

**Наследование** позволяет создавать новый класс на основе существующего, при этом новый класс унаследует все свойства и методы базового класса, но может добавлять новые или изменять существующие. Это способствует повторному использованию кода и созданию более сложных структур на основе простых.

В качестве примера системы наследования в реальной жизни можно вспомнить иерархическую структуру биологических видов или по-простому царства:


Рисунок 1 – Наследование на примере класса животных

В данном примере существует общий класс-наследник «животное» от которого наследуются все остальные классы.

Принцип наследования заключается в том, что наследники получают все характеристики их родителя. К примеру животное может спать и есть – так и наследуемые от этого класса млекопитающие, рыбы и птицы так же смогут это делать.

Другой возможностью наследования является добавление новых методов и атрибутов, а также переопределение старых. Допустим млекопитающие реализуют возможность издавать звуки – значит и их последующие наследники смогут издавать звуки. Только этот метод кошка переопределит чтобы издавать звуки мяуканья, а собака чтобы издавать звук лая.

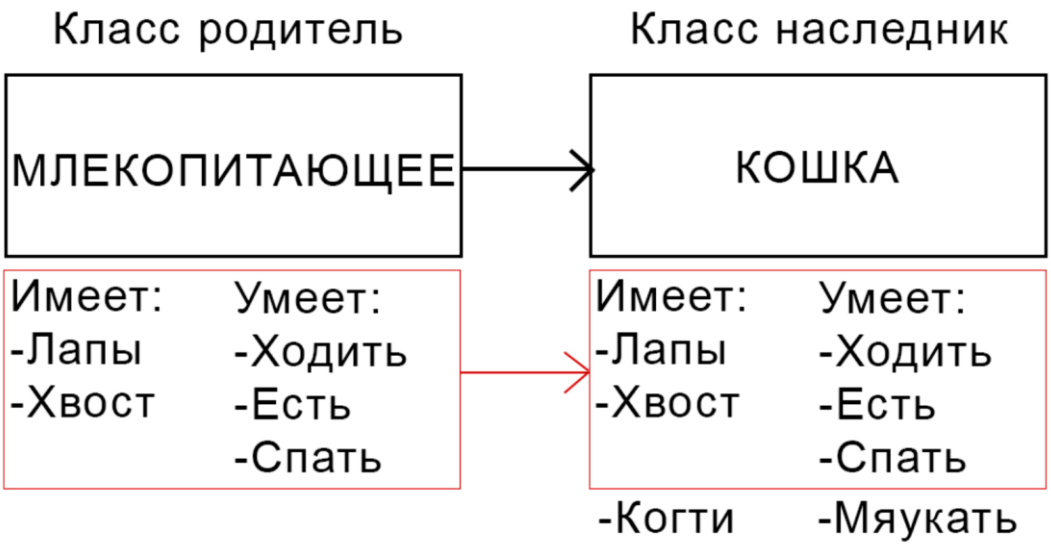


Рисунок 2 – Добавление новых возможностей при наследовании

Переходя к более конкретным примерам реализуем классы Triangle и Rectangle на основе класса Shape:

```
class Triangle : Shape //Через двоеточие определяется от какого класса идет наследование
    {
        //Атрибуты родителя не требуется перечислять в наследнике
        //В данном примере это атрибут Point2D[] points

        //Методы которые не имеют абстракции не требуется перечислять в наследнике
        //В данном примере это метод getPoints()

        //Если наследуемый класс сам не является абстрактным, то все абстрактные методы
        //должны быть переопределены. Для этого используется ключевое слово override
        public override double getArea()
        {
            return 0; //метод был переопределен
        }
    }
```

Класс-наследник может показывать ошибку компиляции если не выполнены следующие условия:

– Если в базовом классе есть конструкторы с параметрами и нет конструктора по умолчанию (без параметров), то класс-наследник должен явно вызвать один из конструкторов базового класса.

– При переопределении метода в классе-наследнике сигнатура метода должна полностью соответствовать базовому методу (тип возвращаемого значения, параметры).

– Если базовый класс содержит абстрактные методы, то класс-наследник обязан реализовать все эти методы. В противном случае, класс-наследник также должен быть объявлен как abstract.

– Если в классе-наследнике определено поле, свойство или метод с тем же именем, что и в базовом классе, это может вызвать ошибки.

В приведенном выше примере все еще существует ошибка – в реализуемом классе нет наследования конструктора из родительского класса. Если в родительском классе существует не пустой конструктор, наследник должен вызывать в своем конструкторе хотя бы один из родительских конструкторов. Вызов родительского конструктора происходит через слова base, куда затем передаются переменные, запрашиваемые конструктором:

```
abstract class Shape
    {
        public Shape(Point2D[] points) //то что принимает конструктор
        {
            this.points = points; //то что сделает конструктор
            Console.WriteLine(“Сейчас работает конструктор Shape”);

        }
    }

class Triangle : Shape
    {
        //передаем в конструктор переменные
        public Triangle(Point2D[] points) : base(points) //эти переменные передаются в
конструктор родительского класса через ключевое слово base
        {
            //конструктор сначала вызывает функцию родительского класса,
            //в примере вызовется this.points = points;
            //затем работает конструктор наследуемого класса:
            Console.WriteLine(“А сейчас работает конструктор Triangle”);
        }
    }
```

Другим возможным инструментом редактирования наследуемых классов является использования виртуальных методов. Виртуальные методы могут быть переопределены в классе наследнике как абстрактные, но в отличии от абстрактных методов виртуальные имеют реализацию. Так же, как и абстрактные методы виртуальные могут переопределить только реализацию – сама сигнатура функции остается идентичной той что находится в родительском классе. Виртуальные методы можно используются в неабстрактных классах. Виртуальный метод оперяется ключевым словом virtual:

```
class Animal
    {
        //Виртуальный метод имеющий реализацию
        public virtual void Speak() { Console.WriteLine("Animal sound"); }
    }

class Dog : Animal
    {
        //Переопределение виртуального метода
        public override void Speak() { Console.WriteLine("Bark"); }
    }
```

Так же последующее переопределение виртуального класса можно запретить, используя ключевое слово sealed:

```
class Dog : Animal
    {
        //Переопределение виртуального метода и запрет на его дальнейшее переопределение
        public sealed override void Speak() { Console.WriteLine("Bark"); }
    }

class Puppy : Dog
    {
        //При переопределении произойдет ошибка компиляции
        public override void Speak() { Console.WriteLine("Bark"); }
    }
```

Последним рассматриваемым принципом ООП будет является принцип **полиморфизма**. **Полиморфизм** позволяет объектам разных типов обрабатываться одинаково, то есть одно и то же действие может выполняться по-разному в зависимости от того, с каким объектом оно применяется.

В C# можно различить два вида полиморфизма – первый из которых **полиморфизм времени выполнения**. Полиморфизм времени выполнения достигается через переопределение методов. Этот тип полиморфизма позволяет вызывать методы производного класса через ссылку на базовый класс. Какая именно версия метода будет вызвана, определяется во время выполнения программы, а не во время компиляции:

```
public class Animal
    {
        public virtual void Speak(){ Console.WriteLine("Animal sound"); }
    }
public class Dog : Animal
    {
        public override void Speak(){ Console.WriteLine("Bark"); }
    }
public class Cat : Animal
    {
        public override void Speak() { Console.WriteLine("Meow"); }
    }
```

Кошка и собака наследуются от класса животного и переопределяют его метод, при вызове этого метода от экземпляра класса результат будет определен тем к реализации какого из классов он относится:

```
public MainWindow()
        {
            InitializeComponent();

            Animal animal = new Animal(); //Экземпляр класса животного
            animal.Speak(); //Выведется Animal sound

            Dog dog = new Dog(); //Экземпляр класса собаки
            dog.Speak(); //Выведется Bark

            Cat cat = new Cat(); //Экземпляр класса кошки
            cat.Speak(); //Выведется Meow
        }
```

**Полиморфизм времени компиляции** достигается через **перегрузку методов**. Этот тип полиморфизма позволяет методам с одним и тем же именем иметь разные параметры, и какая версия метода будет вызвана, определяется во время компиляции программы:

```
public class Calculator
    {
        //сложение целых чисел
        public int Add(int a, int b) { return a + b; }
        //сложение вещественных чисел
        public double Add(double a, double b) {  return a + b; }
        //сложение байт
        public byte Add(byte a, byte b) { return (byte)(a & b); }

    }
```

В данном примере в классе калькулятор существует три метода с одинаковым именем, но разную сигнатуру используемых и возвращаемых данных. Благодаря полиморфизму использование метода будет определятся исходя из передаваемых значений:

```
public MainWindow()
        {
            InitializeComponent();
            Calculator calculator = new Calculator();

            //Используется метод сложения int
            int intNumber = calculator.Add(3, 5);

            //Используется метод сложения double
            double doubleNumber = calculator.Add(3.75, 5.45);

            //Используется метод сложения byte
            byte byteNumber = calculator.Add((byte)13, (byte)20);
        }
```

**Задание №1. Создание базового функционала кликера на время.**

Разработать основной функционал программы, в которой игроку предстоит нажимать на появляющиеся объекты на время.

**Функционал программы:**

– Создание визуальных объектов в случайных точках в определенной зоне окна программы.
– Объекты создаются в определенном промежутке времени после начала и перестают появляться после окончания времени.
– После нажатия на объект игроку начисляются очки и объект исчезает.

**Задачи:**

– Реализовать программу для отрисовки объектов в окошке.
– Логика игры и нажимаемый объект должны быть реализованы в виде классов.


Рисунок 3 – Пример интерфейса программы

В качестве основных атрибутов нажимаемого объекта можно использовать следующую структуру, в данном варианте объект отображается в качестве круга:

```
public CObject(Point position, double size, double lifetime)    //конструктор
        {
            this.position = position;
            this.size = new Size(size, size);
            this.lifetime = lifetime;

            //создание фигуры
            sprite = new Ellipse();

            //цвет фигуры
            sprite.Fill = Brushes.BlueViolet;
            sprite.StrokeThickness = 2;
            sprite.Stroke = Brushes.Black;

            //центрирование фигуры
            sprite.HorizontalAlignment = HorizontalAlignment.Center;
            sprite.VerticalAlignment = VerticalAlignment.Center;

            //размеры фигуры
            sprite.Width = this.size.Width;
            sprite.Height = this.size.Height;
            sprite.RenderTransform = new TranslateTransform(position.X, position.Y);

            pointsValue = ((1/this.size.Width) / lifetime)*1000; //очковая стоимость обьекта
        }
```

В данном классе следует реализовать следующие функции:

– Получение формы объекта. Возвращает фигуру.
– Получение очковой стоимости. Возвращает число.
– Проверка нахождения курсора мыши в пределах объекта. Возвращает булево значение.
– Изменения времени жизни. Возвращает булево значение для подтверждения того что время жизни закончилось.

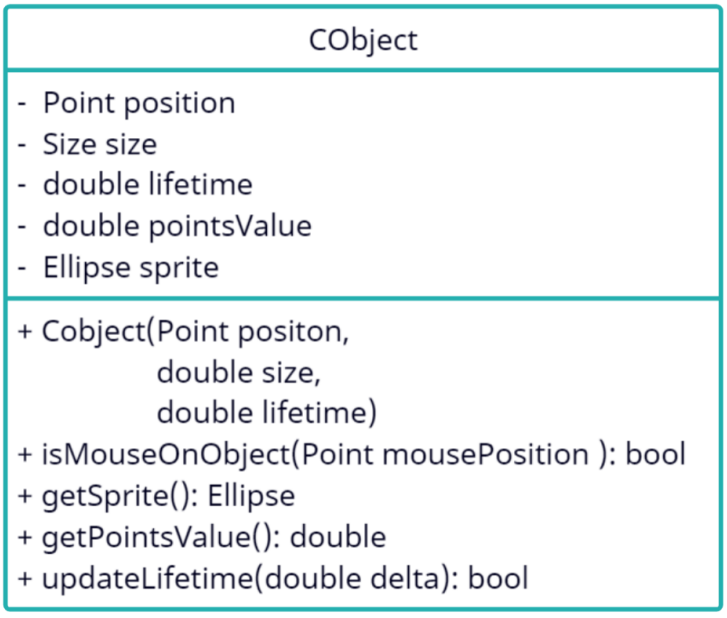


Рисунок 4 – UML-диаграмма класса CObject

Для контроллера логики можно использовать следующий код:

```
public class CController    //класс управляющий собираемыми объектами
    {
        List<CObject> objects;  //список собираемых объектов

        double spawnRate;       //время, между созданием собираемых объектов
        double time;            //время с момента создания последнего собираемого объекта

        Random rng;

        //минимальное и максимальное время жизни собираемых объектов
        double minLifetime;
        double maxLifetime;

        //минимальный и максимальный размер собираемых объектов
        double minSpriteSize;
        double maxSpriteSize;

        Size sceneSize; //размер сцены

        double points;  //набранные очки

        public CController(double spawnRate, ulong startTime, Size sceneSize)
        {
            rng = new Random();
            objects = new List<CObject>();

            this.spawnRate = spawnRate;
            time = startTime;
            this.sceneSize = sceneSize;
            points = 0;

            minLifetime = 1;
            maxLifetime = 5;

            minSpriteSize = 10;
            maxSpriteSize = 20;
        }

}
```

Помимо функций получения приватных атрибутов следует реализовать следующие функции:

- Создание нового объекта.
- Удаление объекта.
- Обновление состояния времени жизни объектов.
- Отслеживание нажатия курсора.

В следующем примере UML-диаграммы класса не описаны методы для получения приватных атрибутов:

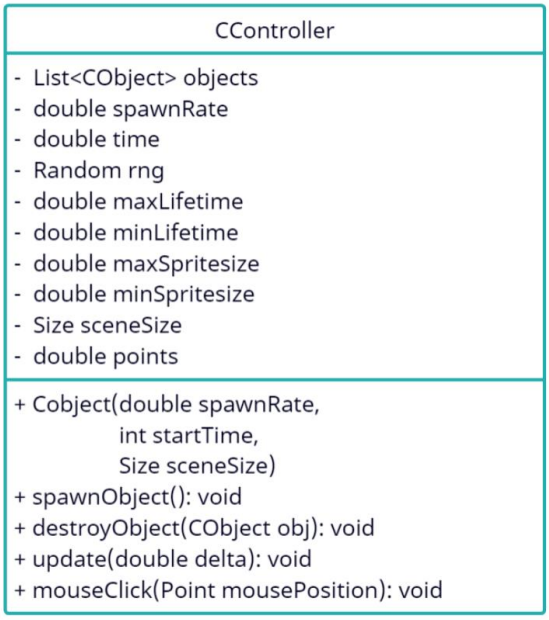


Рисунок 5 – UML-диаграмма класса CController

Обновление состояния объектов можно привязать к обновлению таймера в основном теле программы:

```
CController controller;

public MainWindow()
        {
            InitializeComponent();

            //Создание таймера обновляющегося каждые 100 миллисекунд
            timer = new DispatcherTimer();
            timer.Interval = new TimeSpan(0, 0, 0, 0, 100);
            timer.Tick += Timer_Tick;
        }

private void Timer_Tick(object sender, EventArgs e)
        {
            controller.update(0.1); //обновляем состояние контроллера
        }
```

**Задание №2. Дополнительный функционал, наследование.**

Модифицировать созданную в первом задании программу добавив различный функционал для объектов методом наследования.

**Функционал программы:**

– Нажатие имеет время перезарядки.
– Помимо обычных сфер создаются сферы имеющие различные бонусы: уменьшение перезарядки, увеличение времени жизни сферы, уменьшение времени появления новой сферы, добавление очков.

**Задачи:**

– Реализовать новые классы выполняющие требуемые функции используя принцип наследования.
– Добавить класс игрока, содержащий информацию о перезарядке клика.

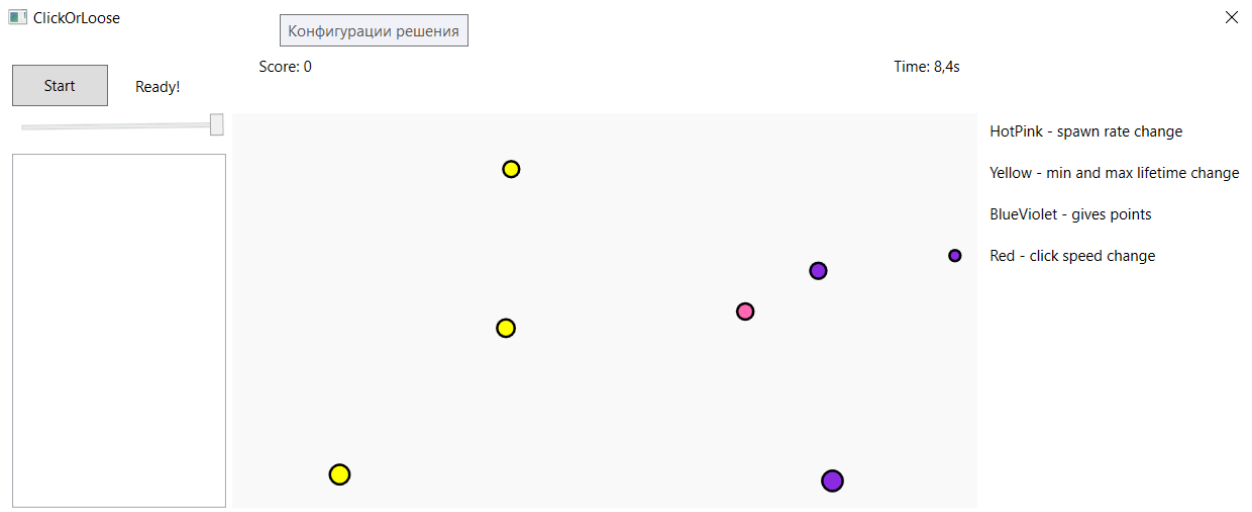


Рисунок 6 – Пример интерфейса программы

Для решения этой задачи лучше всего добавить новый класс CCollectable который будет модификацией класса CObject. Стоит сделать его абстрактным и добавить в этот класс функцию отработки нажатия:

```
public abstract class CCollectable
    {
        Point position;     //позиция собираемого объекта в сцене
        protected Size size;          //размер собираемого объекта
        double lifetime;    //время жизни собираемого объекта
        protected Ellipse sprite;     //визуальное отображение собираемого объекта
        public CCollectable(Point position, double size, double lifetime)    //конструктор
        {
            this.position = position;
            this.size = new Size(size, size);
            this.lifetime = lifetime;

            sprite = new Ellipse();

            sprite.Fill = Brushes.BlueViolet;
            sprite.StrokeThickness = 2;
            sprite.Stroke = Brushes.Black;

            sprite.HorizontalAlignment = HorizontalAlignment.Center;
            sprite.VerticalAlignment = VerticalAlignment.Center;

            sprite.Width = this.size.Width;
            sprite.Height = this.size.Height;
            sprite.RenderTransform = new TranslateTransform(position.X, position.Y);
        }

//абстрактная функция отработки нажатия на объект

        public abstract bool onClick(CPlayer player, CController controller, Point
mousePosition);

    }
```

Данную абстрактную функцию следует переопределить в классах наследниках для выполнения того функционала который будет подразумевать класс – добавление времени, уменьшение перезарядки и т.д.

Объект игрока и контроллера передаются в функцию для того чтобы можно было модифицировать их параметры.

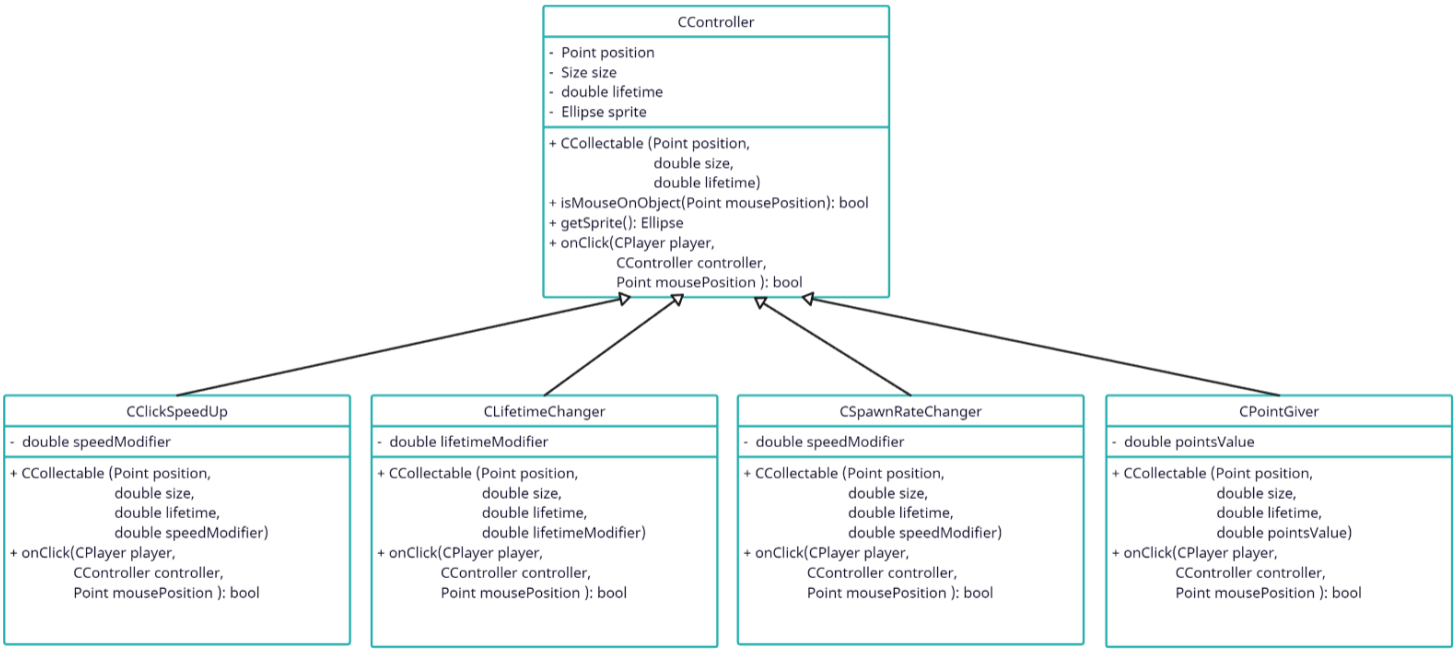


Рисунок 7 – UML-диаграмма зависимости классов

Так может выглядеть объект-наследник, реализующий функционал добавления очков:

```
public class CPointGiver : CCollectable
    {
        //данный атрибут можно убрать из CObject тк только данный
        //класс будет отвечать за добавление очков
        double pointsValue;

        //конструктор класса реализует конструктор родителя
        public CPointGiver(Point position, double size, double lifetime) : base(position,
size, lifetime)
        {
            sprite.Fill = Brushes.BlueViolet;
            pointsValue = ((1 / this.size.Width) / lifetime) * 1000;
        }

        //переопределение функции нажатия
        public override bool onClick(CPlayer player, CController controller, Point
mousePosition)
        {
            if (isMouseOnObject(mousePosition) == false)
                return false;

            controller.pointsIncrease(pointsValue);

            return true;
        }
```

Упрощает текущую задачу принцип полиморфизма – в классе CController потребуются минимальные изменения кода:

```
public class CController    //класс управляющий собираемыми объектами
    {
        List<CCollectable> objects; //список собираемых объектов

    }
```

Не требуется создавать отдельные списки под объекты-наследники, они все могут хранится в списке, обозначенном для хранения объекта-родителя, так как хоть и являясь другими классами они могут использоваться как их класс родитель.

Таким же образом можно проверять метод onClick(), несмотря на то что он применяется к объекту из списка List<CCollectable>, полиморфизм времени выполнения явно определяет класс у которого вызывается данный метод:

```
if (objects[i].onClick(player, this, mousePosition) == true)
    {
        //работа при выполнении условия

    }
```

Стоит разделить функционал, входящий в игрока и счетчик до возможности следующего клика на два класса так как функционал отсчета времени логически непосредственно не входит в возможности игрока и может быть использован для других функций:

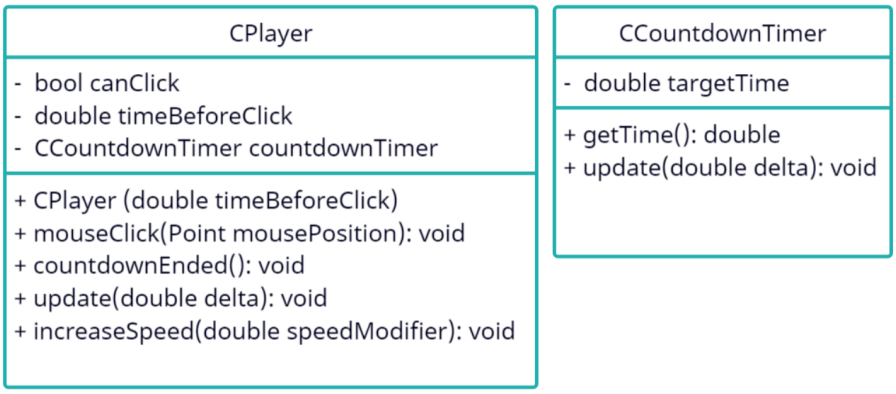


Рисунок 8 – UML-диаграммы класса игрока и отсчета времени

**Задание №3. Перенос механик в игру-кликер.**

Теперь, когда функционал отработан и протестирован его можно включить в начальный проект разрабатывавшийся в ходе лабораторных работ №1 и №2.

**Функционал программы:**

– Между кликами по противнику существует перезарядка.
– Игрок может уменьшить перезарядку улучшая данный параметр за золото.
– В зоне кликов по противнику случайно появляются сферы с эффектами, эффектами работают некоторое время, активными могут быть несколько эффектами.

**Задачи:**

– Модифицировать созданный в ходе предыдущих работ проект игры-кликкера используя код разработанный в данной лабораторной работе.
– Добавить перезарядку между кликами по противнику.
– В качестве возможных эффектов могут быть: увеличение урона по противникам, уменьшение перезарядки между атаками, моментальное добавление золота.