**Московский государственный технический**

**университет им. Н.Э. Баумана**

Факультет «Информатика и системы управления»

Кафедра ИУ5 «Системы обработки информации и управления»

Курс «Парадигмы и конструкции языков программирования»

Отчет по лабораторным работам №5-6

«**Реализация псевдоабстрактных и наследующих**

**от них классов на Godot C#.**»

|  |  |  |
| --- | --- | --- |
| Выполнил: |  | Проверил: |
| студент группы ИУ5-31Б |  | преподаватель каф. ИУ5 |
| Ларкин Б. В. |  | Гапанюк Ю. Е. |
| Подпись и дата: |  | Подпись и дата: |
|  |  |  |

Москва, 2023 г.

Цель работы

Реализовать псевдоабстрактные классы на Godot C#, то есть родительские классы для наследования с виртуальными функциями; экземпляры наследующих классов для написания проекта на платформе.

Коды псевдоабстрактных классов Ability и entity – Л. р. №5:

**Ability**

using Godot;

using System;

using System.Data.Common;

using System.Runtime.CompilerServices;

public unsafe partial class Ability : Node2D, ICloneable

{

    public float CD;

    public float use\_time;

    public float cost;

    protected Timer useTimer;

    protected Timer CDTimer;

    protected Node ParentalAbilityNode; //Connects parent ability node to inheriting children classes and nodes.

    protected entity passive\_application; //from

    protected entity active\_application; //to

    protected string input\_key; //input\_key to enable kbm input in order to call the function. Otherwise cast on CD

    public void perform(entity obj){

        if (get\_state() & !is\_oneshot()){

            Use(passive\_application);

        }

        else{

            if (Input.IsActionJustPressed(input\_key)){

                UseAbility(obj);

                passive\_application = obj;

            }

        }

    }

    public void perform(entity from, entity to){

        if (get\_state() & !is\_oneshot()){

            Use(passive\_application, active\_application);

        }

        else{

            if (Input.IsActionJustPressed(input\_key) || (input\_key == null)){

                UseAbility(from, to);

                passive\_application = from;

                active\_application = to;

            }

        }

    }

    public bool get\_state(){

        return (bool)ParentalAbilityNode.GetMeta("is\_using");

    }

    public void set\_state(bool x){

        ParentalAbilityNode.SetMeta("is\_using",x);

        return;

    }

    public void set\_canuse(bool x){

        ParentalAbilityNode.SetMeta("CanUse",x);

        return;

    }

    public bool get\_canuse\_state(){

        return (bool)ParentalAbilityNode.GetMeta("CanUse");

    }

    public void set\_oneshot(bool x){

        ParentalAbilityNode.SetMeta("OneShot",x);

        return;

    }

    public bool is\_oneshot(){

        return (bool)ParentalAbilityNode.GetMeta("OneShot");

    }

    public object Clone()

    {

        return this.MemberwiseClone();

    }

    protected virtual void Use(entity obj){}

    protected virtual void Use(entity from, entity to){}

    public void UseAbility(entity obj)

    {

        if (get\_canuse\_state() == true){

            Use(obj);

            passive\_application = obj;

            set\_state(true);

            set\_canuse(false);

            useTimer.Start();

        }

    }

    public void UseAbility(entity from, entity to)

    {

        GD.Print(get\_canuse\_state());

        if (get\_canuse\_state() == true){

            Use(from, to);

            passive\_application = from;

            active\_application = to;

            set\_state(true);

            set\_canuse(false);

            useTimer.Start();

        }

    }

    void \_on\_ready()

    {

        useTimer = GetNode<Timer>("useTimer");

        CDTimer = GetNode<Timer>("CDTimer");

        useTimer.WaitTime = use\_time;

        CDTimer.WaitTime = CD;

        ParentalAbilityNode = GetNode(this.GetPath());

        set\_state(false);

        set\_oneshot(true);

    }

    protected Ability(Ability Obj){

        if (this != Obj){

            this.CD = Obj.CD;

            this.use\_time = Obj.use\_time;

            this.cost = Obj.cost;

        }

    }

    protected Ability(float cd, float uset, float ct, string input\_key){

        CD = cd;

        use\_time = uset;

        cost = ct;

        this.input\_key = input\_key;

    }

    protected void \_on\_use\_timer\_timeout()

    {

        set\_state(false);

        CDTimer.Start();

    }

    protected void \_on\_cd\_timer\_timeout()

    {

        set\_canuse(true);

    }

    public Ability()

    {

        CD=1.0f;

        use\_time=0.5f;

        cost = 0;

    }

}

**entity**

using Godot;

using System;

public abstract partial class entity : CharacterBody2D, ICloneable

{

    public float max\_speed;

    public float acceleration;

    public float friction;

    public float HP;

    public override void \_PhysicsProcess(double delta){}

    public Vector2 direction;

    public Vector2 velocity;

    protected entity(entity Obj){

        if (this != Obj){

            max\_speed = Obj.max\_speed;

            acceleration = Obj.acceleration;

            friction = Obj.friction;

            this.HP = Obj.HP;

        }

    }

    protected entity(){

        max\_speed = 300;

        acceleration = 150;

        friction = 100.0f;

        HP = 100.0f;

    }

    protected entity(float max\_spd, float hp, float a, float fr){

        max\_speed = max\_spd;

        acceleration = a;

        friction = fr;

        HP = hp;

    }

    public abstract object Clone();

    public void perform(Ability a){

        a.perform(this);

    }

    public void perform(Ability a, entity to){

        a.perform(this, to);

    }

}

Коды экземпляров наследующих классов Dash и Player – Л. р. №5:

**Dash**

using Godot;

using System;

public unsafe partial class Dash : Ability

{

    public float dash\_speed;

    public bool ghost\_on;

    protected override void Use(entity Obj)

    {

        Obj.velocity = Obj.direction \* dash\_speed;

    }

    public Dash(Dash Obj) : base(Obj)

    {

        dash\_speed = Obj.dash\_speed;

        ghost\_on = Obj.ghost\_on;

    }

    public Dash () : base()

    {

        dash\_speed = 600f;

        ghost\_on = false;

    }

    public void set(float cd, float uset, float ct, float dash\_spd, bool ghost, string input\_key, bool one\_s)

    {

        CD = cd;

        use\_time = uset;

        cost = ct;

        useTimer = GetNode("Ability").GetNode<Timer>("useTimer");

        CDTimer = GetNode("Ability").GetNode<Timer>("CDTimer");

        ParentalAbilityNode = GetNode("Ability");

        CDTimer.WaitTime = cd;

        useTimer.WaitTime = uset;

        dash\_speed = dash\_spd;

        ghost\_on = ghost;

        this.input\_key = input\_key;

        this.set\_oneshot(one\_s);

    }

}

**Player**

using Godot;

using System;

public partial class Player : entity

{

    private Dash dash;

    public override object Clone()

    {

        return new Player(this);

    }

    void \_on\_dash\_ready()

    {

        dash = GetNode<Dash>("Dash");

        dash.set(0.5f, 0.2f, 0, 400.0f, true, "ui\_dash", true);

    }

    protected Player(Player Obj)

    {

        dash = new Dash(Obj.dash);

        HP = Obj.HP;

    }

    protected Player()

    {

        HP = 100.0f;

        max\_speed = 200;

        acceleration=600;

        friction=500;

    }

    public override void \_PhysicsProcess(double delta)

    {

        velocity = Velocity;

        // Input direction and handling the movement/deceleration.

        direction = Input.GetVector("ui\_left", "ui\_right", "ui\_up", "ui\_down");

        if (velocity.Length() > max\_speed){

                velocity -= velocity.Normalized() \* (float)(friction \* delta)\*1.1f;

        }

        if (direction == Vector2.Zero)

        {

            if (velocity.Length() > friction \* delta){

                velocity -= velocity.Normalized() \* (float)(friction \* delta);

            }

            else{

                velocity = Vector2.Zero;

            }

        }

        else

        {

            //cut out unnecessary velocity

            if (direction.X == 0){velocity.X -= (float)(velocity.X \* friction \* delta)\*0.01f;}

            if (direction.Y == 0){velocity.Y -= (float)(velocity.Y \* friction \* delta)\*0.01f;}

            perform(dash);

            if (velocity.Length()<=max\_speed)

            {

                velocity += direction \* acceleration \* (float)delta;

                velocity = velocity.LimitLength(max\_speed);

            }

        }

        Velocity = velocity;

        MoveAndSlide();

    }

}