Swinburne University of Technology

COS20007 Object Oriented Programming

Semester Test Resit

PDF generated at 08:36 on Friday 26th May, 2023

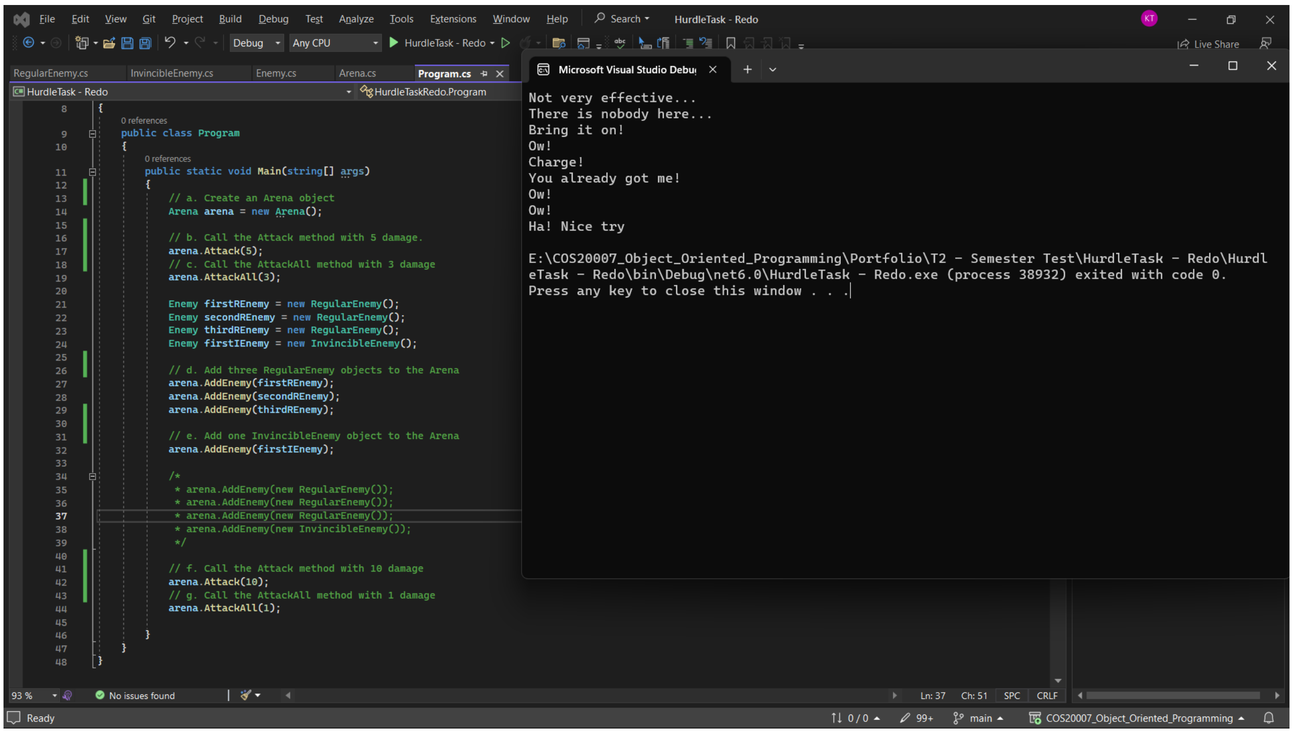
```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HurdleTaskRedo
{
    public class Program
    {
        public static void Main(string[] args)
        {
            // a. Create an Arena object
            Arena arena = new Arena();

            // b. Call the Attack method with 5 damage.
            arena.Attack(5);
            // c. Call the AttackAll method with 3 damage
            arena.AttackAll(3);

            Enemy firstREnemy = new RegularEnemy();
            Enemy secondREnemy = new RegularEnemy();
            Enemy thirdREnemy = new RegularEnemy();
            Enemy firstIEnemy = new InvincibleEnemy();

            // d. Add three RegularEnemy objects to the Arena
            arena.AddEnemy(firstREnemy);
            arena.AddEnemy(secondREnemy);
            arena.AddEnemy(thirdREnemy);

            // e. Add one InvincibleEnemy object to the Arena
            arena.AddEnemy(firstIEnemy);

            /*
             * arena.AddEnemy(new RegularEnemy());
             * arena.AddEnemy(new RegularEnemy());
             * arena.AddEnemy(new RegularEnemy());
             * arena.AddEnemy(new InvincibleEnemy());
             */

            // f. Call the Attack method with 10 damage
            arena.Attack(10);
            // g. Call the AttackAll method with 1 damage
            arena.AttackAll(1);
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HurdleTaskRedo
{
    public class Arena
    {
        private List<Enemy> _enemies;

        public Arena()
        {
            _enemies = new List<Enemy>();
        }

        public void AddEnemy(Enemy enemy)
        {
            _enemies.Add(enemy);
        }

        public void Attack(int damage)
        {
            if (_enemies.Count > 0)
            {
                Console.WriteLine("Bring it on!");
                _enemies[0].GetHit(damage);
            }
            else
            {
                Console.WriteLine("Not very effective...");
            }
        }

        public void AttackAll(int damage)
        {
            if (_enemies.Count > 0)
            {
                Console.WriteLine("Charge!");
                foreach (Enemy enemy in _enemies)
                {
                    enemy.GetHit(damage);
                }
            }
            else
            {
                Console.WriteLine("There is nobody here...");
            }
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HurdleTaskRedo
{
    public abstract class Enemy
    {
        public Enemy()
        {
        }

        public abstract void GetHit(int damage);
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HurdleTaskRedo
{
    public class InvincibleEnemy : Enemy
    {
        public override void GetHit(int damage)
        {
            Console.WriteLine("Ha! Nice try");
        }
    }
}
```

```
using Microsoft.VisualBasic;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HurdleTaskRedo
{
    public class RegularEnemy : Enemy
    {
        private int _health;

        public RegularEnemy()
        {
            _health = 10;
        }

        public override void GetHit(int damage)
        {
            // When it is hit, one of two things happen:
            //  If it has more than 0 health
            if (_health > 0)
            {
                Console.WriteLine("Ow!");       // Ow! is printed to the terminal
                _health -= damage;              // Its health is reduced by the
  ↪   specified amount of damage
            }
            else
            {
                Console.WriteLine("You already got me!");          // You already
  ↪   got me! is printed
            }
        }
    }
}
```



```
{
    public class Program
    {
        public static void Main(string[] args)
        {
            // a. Create an Arena object
            Arena arena = new Arena();

            // b. Call the Attack method with 5 damage.
            arena.Attack(5);
            // c. Call the AttackAll method with 3 damage
            arena.AttackAll(3);

            Enemy firstREnemy = new RegularEnemy();
            Enemy secondREnemy = new RegularEnemy();
            Enemy thirdREnemy = new RegularEnemy();
            Enemy firstIEnemy = new InvincibleEnemy();

            // d. Add three RegularEnemy objects to the Arena
            arena.AddEnemy(firstREnemy);
            arena.AddEnemy(secondREnemy);
            arena.AddEnemy(thirdREnemy);

            // e. Add one InvincibleEnemy object to the Arena
            arena.AddEnemy(firstIEnemy);

            /*
             * arena.AddEnemy(new RegularEnemy());
             * arena.AddEnemy(new RegularEnemy());
             * arena.AddEnemy(new RegularEnemy());
             * arena.AddEnemy(new InvincibleEnemy());
             */

            // f. Call the Attack method with 10 damage
            arena.Attack(10);
            // g. Call the AttackAll method with 1 damage
            arena.AttackAll(1);

        }
    }
}
```

```
Not very effective...
There is nobody here...
Bring it on!
Ow!
Charge!
You already got me!
Ow!
Ow!
Ha! Nice try

E:\COS20007_Object_Oriented_Programming\Portfolio\T2 - Semester Test\HurdleTask - Redo\Hurdl
eTask - Redo\bin\Debug\net6.0\HurdleTask - Redo.exe (process 38932) exited with code 0.
Press any key to close this window . . .
```

Name: Trung Kien Nguyen

StudentID: 104053642

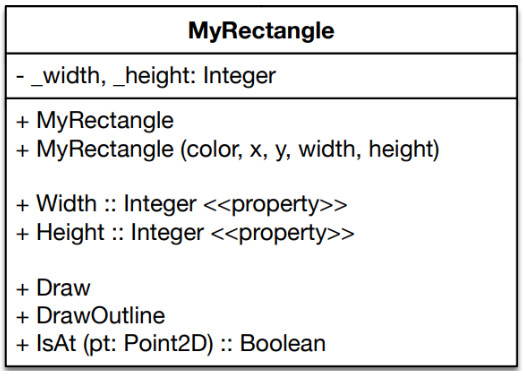
TASK 2

- **Abstraction:**

  - Definition: **Abstraction** principle in OOP is usually used to decide the ideas, or features of the objects within a program, and also the program itself. This helps us to concentrates mainly on "making decisions" of what we should implement in the program, such as:
    - which objects we should include or create in the program,
    - which features of those objects we have to implement in the code,
    - and which are not necessary that should be excluded from the clients/users.

    Alternatively, **abstraction** helps identify and establish the objects' classifications, roles, responsibilities and collborations within the program, thus, it is much easier to manage the complexity of the whole program in the first place, and frequently helpful in dealing with the enormous and complex program, project or system.

  - Example: Taking the task Multiple Shape Kinds (4.1) that I have done in this semester:


MyRectangle
- _width, _height: Integer
+ MyRectangle
+ MyRectangle (color, x, y, width, height)
+ Width :: Integer <<property>>
+ Height :: Integer <<property>>
+ Draw
+ DrawOutline
+ IsAt (pt: Point2D) :: Boolean

I have used the **Abstraction** principle when thinking about a "MyRectangle" class for the rectangle objects, I comes up with some of its aspects:

- Its position in float numbers, representing the position of the rectangle in the screen.
- Its size ("_width" and "_height" int fields), representing the size, including the width and height, of the rectangle.
- Its color ("_colors" Color field), to display the color of the rectangle to the output.
- Its roles of drawing ("Draw()" method), to draw itself to the screenn when needed.
- Its roles of drawing outline ("DrawOutline()" method), to draw is outline when the mouse is in the rectangle.
- Its roles of check if it is a point is inside the rectangle area or not ("IsAt" method).

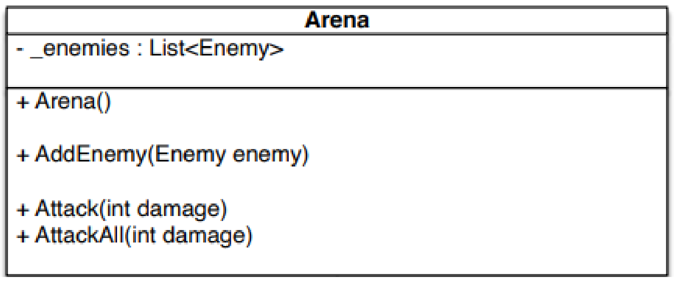
These aspects have made a "model" (or interface) of a typical rectangle in my program that the users need. This is what I used the "**abstraction**" principle for.

- **Encapsulation:**

  - Definition: **Encapsulation** is the principle that encourages the bundling of the data (fields) inside an object, and its functionalities (properties and methods) related to that data. We often provide, control or limit the access to those functionalities for the other external objects or programs to interact and make changes indirectly to that inner data only when needed, which helps us to control effectively the data accessibility, make sure the internal state of each object remains consistency and safe from the outside world, while still ensuring the efficient functions of the program. More simply, because we, as well as the external objects, don't need to know how an object work inside it, the object encapsulates the complexity of its state.

  - Example:

    - Taking the example in this T2 test ("Arena" class):



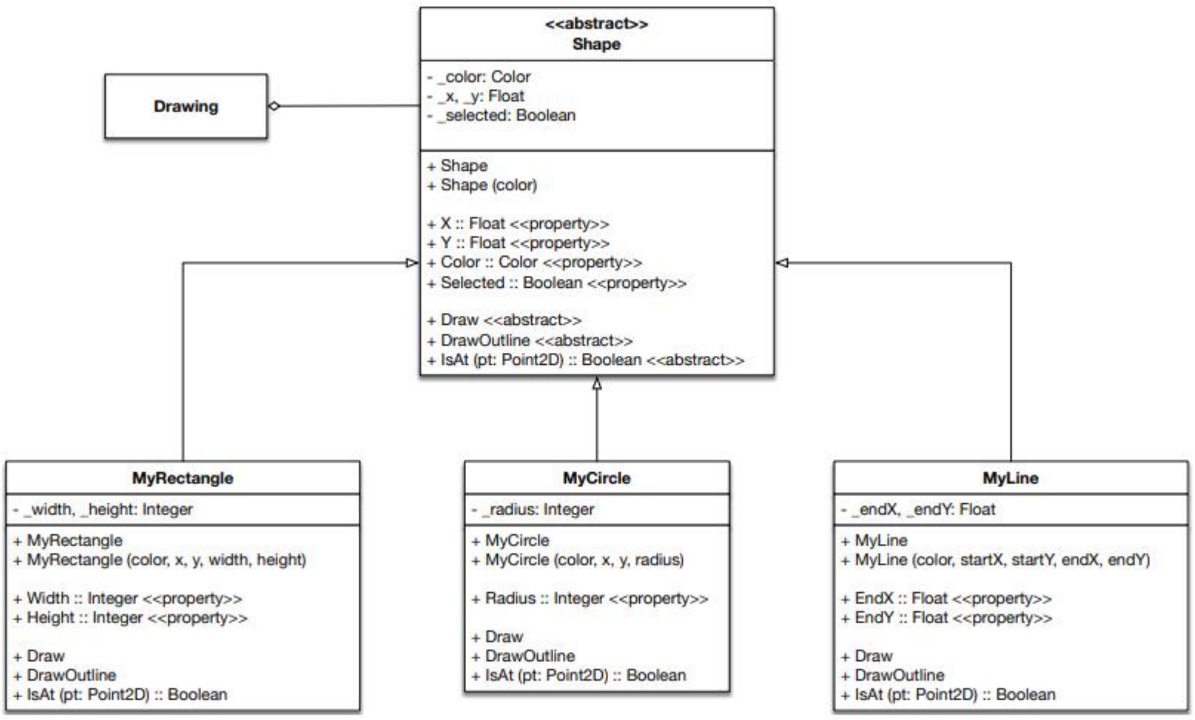
```
Arena
- _enemies : List<Enemy>

+ Arena()

+ AddEnemy(Enemy enemy)

+ Attack(int damage)
+ AttackAll(int damage)
```

- Generally, I – the user, as well as the external part of the program, only needs to know that the "Arena" object can add enemies to its list, and attack them with certain damage value. We don't need to know the specific mechanism inside the "Arena" class to perform those tasks, so it is hidden from the rest of the program.
- The "Arena" class has the data of "_enemies" fields, which cannot be accessed and modified directly. However, it has the method "AddEnemy()" for external world (Main method of the Program) to add a new "Enemy" into that list.
- We can also change the state of the first Enemy, or the whole elements in the list with the "Attack()" and "AttackAll()" methods respectively.

- **Inheritence:**

  - Definition: This principle is about a "is-a" relationship between classes, meaning a child class (derived class/ subclass) is a specialized version of its parent class (base class). That child class inherits the parent one's members, including attributes and behaviours, which usually are properties and methods. Inheritence helps avoid code duplication by reusing it, and thus simplify the program. Also, it is closely related to the **Polymorphism** principle.

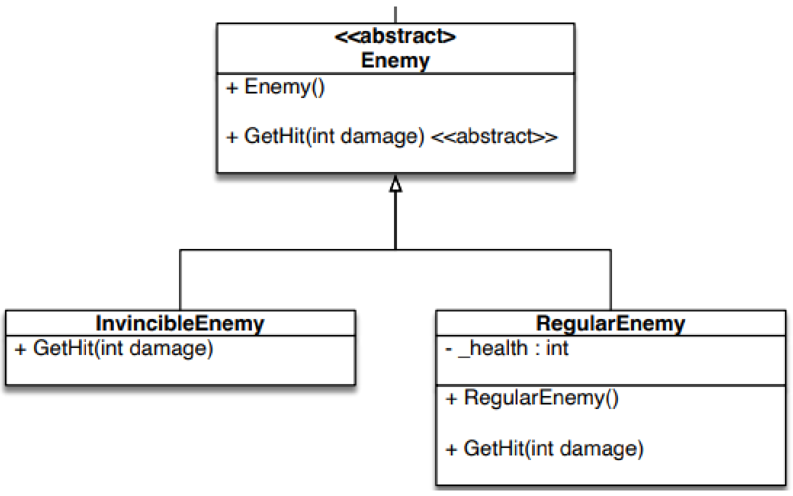
- Example: Still taking the 4.1 task:



<<abstract>>
Shape
- _color: Color
- _x, _y: Float
- _selected: Boolean
+ Shape
+ Shape (color)
+ X :: Float <<property>>
+ Y :: Float <<property>>
+ Color :: Color <<property>>
+ Selected :: Boolean <<property>>
+ Draw <<abstract>>
+ DrawOutline <<abstract>>
+ IsAt (pt: Point2D) :: Boolean <<abstract>>
Drawing
MyRectangle
- _width, _height: Integer
+ MyRectangle
+ MyRectangle (color, x, y, width, height)
+ Width :: Integer <<property>>
+ Height :: Integer <<property>>
+ Draw
+ DrawOutline
+ IsAt (pt: Point2D) :: Boolean
MyCircle
- _radius: Integer
+ MyCircle
+ MyCircle (color, x, y, radius)
+ Radius :: Integer <<property>>
+ Draw
+ DrawOutline
+ IsAt (pt: Point2D) :: Boolean
MyLine
- _endX, _endY: Float
+ MyLine
+ MyLine (color, startX, startY, endX, endY)
+ EndX :: Float <<property>>
+ EndY :: Float <<property>>
+ Draw
+ DrawOutline
+ IsAt (pt: Point2D) :: Boolean

"MyRectangle" class is inherited from the "Shape" class, so it also inherited the public members, including "X", "Y" properties, and "IsAt", "Draw()" and "DrawOutline()" methods, then, as a result, any "MyRectangle" objects can have and use them.

- **Polymorphism:**
  - Definition: With a literal meaning, "poly" means "many", while "morph" means "forms". So, according to this principle, an object in an OO program can take different forms or behaviours, or objects created from different classes can be considered as they are from the same class. In other words, besides its own type, an object can also have its parent's type. **Polymorphism** is often gained when class(es) inherited from a parent class, or implemented the same interface, that is why this is closely related to **Inheritence** principle.
  - Example: Still taking the example of this T2 test:



Each "Enemy" element in the "_enemies" list (of the "Arena" class), which expects a "Enemy" object, can also be "InvicibleEnemy" or "RegularEnemy" object there, as "InvicibleEnemy" and "RegularEnemy" classes are inherited from "Enemy" abstract class. This is a clear example of the **Polymorphism** principle.

In addition, in the "Enemy" class, there is the abstract "GetHit()" method, which is then overridden in "InvincibleEnemy" and "RegularEnemy" with separated implementation inside to fit with each type of enemy when the "Arena" do "Attack()" or "AttackAll()".

REFERENCES

[1] My task of 6.2P – Key Object Oriented Concepts