Swinburne University of Technology

COS20007 Object Oriented Programming

Semester Test Resit

PDF generated at 13:13 on Thursday 26th October, 2023

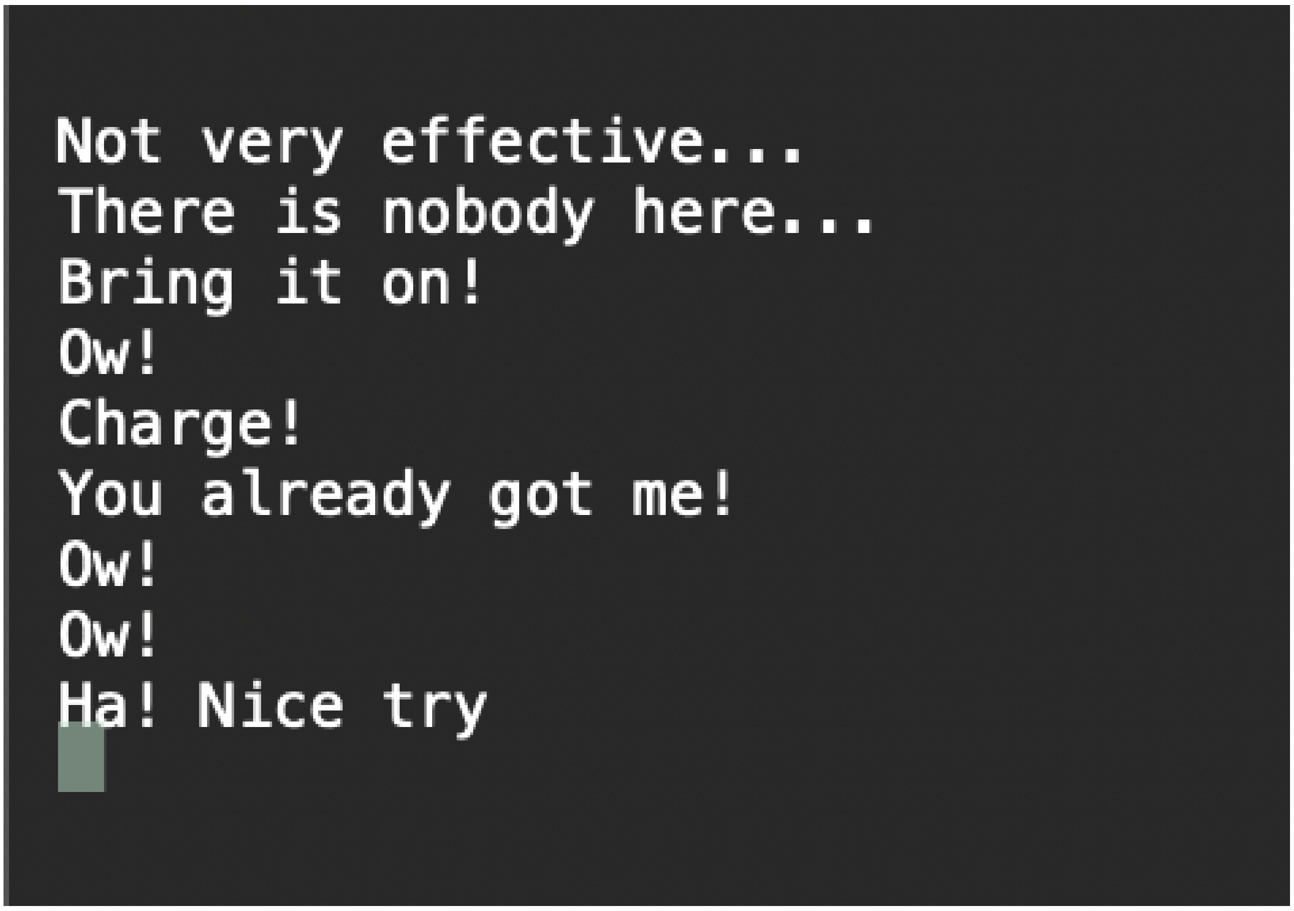
```csharp
using System;
namespace Test2;

public class Program
{
    static void Main()
    {
        //Create an arena object
        Arena _arena = new Arena();
        //Call the Attack method with 5 damage
        _arena.Attack(5);
        //Call the AttackAll method with 3 damage
        _arena.AttackAll(3);
        //Add three RegularEnemy objects to the Arena
        _arena.AddEnemy(new RegularEnemy());
        _arena.AddEnemy(new RegularEnemy());
        _arena.AddEnemy(new RegularEnemy());
        //Add one InvincibleEnemy object to the Arena
        _arena.AddEnemy(new InvincibleEnemy());
        //Call the Attack method with 10 damage
        _arena.Attack(10);
        //Call the AttackAll method with 1 damage
        _arena.AttackAll(1);

        Console.ReadLine();
    }
}

```

```
using System;
namespace Test2
{
    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) // The arena contains enemys
            {
                Console.WriteLine("Bring it on!");
                _enemies[0].GetHit(damage);
            }
            else // There are no enemys
            {
                Console.WriteLine("Not very effective...");
            }
        }
        public void AttackAll(int damage)
        {
            if (_enemies.Count != 0)
            {
                Console.WriteLine("Charge!");
                int i = 0;
                foreach(Enemy e in _enemies)
                {
                    _enemies[i].GetHit(damage);
                    i++;
                }
            }
            else
            {
                Console.WriteLine("There is nobody here...");
            }

        }
    }
}

```

```
using System;
namespace Test2
{
    public abstract class Enemy
    {
        public Enemy()
        {
        }

        public abstract void GetHit(int damage);
    }
}

```

```csharp
using System;
namespace Test2
{
    public class InvincibleEnemy : Enemy
    {
        public InvincibleEnemy()
        {
        }

        public override void GetHit(int damage)
        {
            Console.WriteLine("Ha! Nice try");
        }
    }
}

```

```
using System;
namespace Test2
{
    public class RegularEnemy : Enemy
    {
        private int _health;

        public RegularEnemy()
        {
            _health = 10;
        }

        public override void GetHit(int damage)
        {
            if(_health > 0)
            {
                _health = _health - damage;
                Console.WriteLine("Ow!");
            }
            else
            {
                Console.WriteLine("You already got me!");
            }
        }
    }
}

```



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

Test 2 Part 2

Abstraction

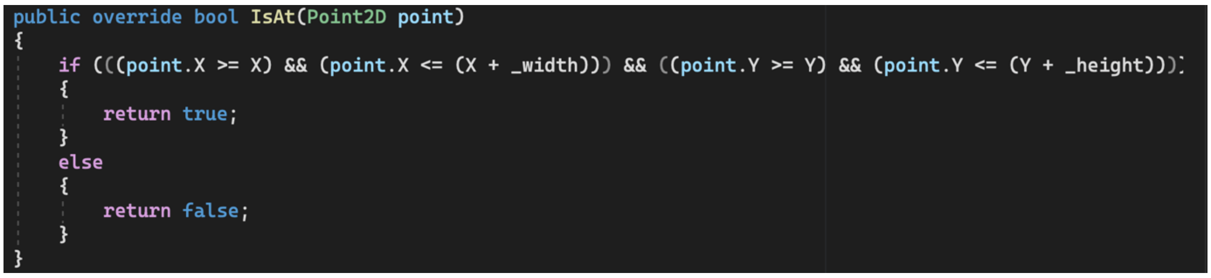
Abstraction helps identify classifications, roles, responsibilities, and collaborations. It is about simplifying, and reducing or removing complexity entirely. Abstraction is about deciding how to simplify an idea. It involves identifying what you need something to do, but not how it does that thing.

An example of Abstraction is for our Maze Task, where we use Iterations to reduce the overall complexity of the project. To aid with removing the complexity, UML diagrams allows for easier planning, figuring out what we need each class to do.

Encapsulation

Encapsulation is where you hide all the un-needed information and only present the needed attributes and fields, like putting It into a capsule. This ensures that certain internal fields aren't accidentally changed or accessible to another mechanism or method, reducing the risk of creating bugs or errors.

Encapsulation can be seen everywhere in code, as methods, classes, and properties technically all use encapsulation. One example of this in the Shape Drawer task is the IsAt method:



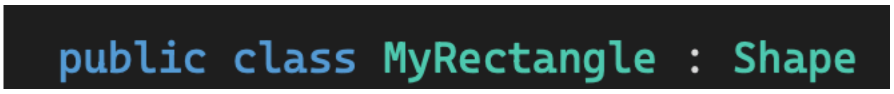
```
public override bool IsAt(Point2D point)
{
    if (((point.X >= X) && (point.X <= (X + _width))) && ((point.Y >= Y) && (point.Y <= (Y + _height))))
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Here we can see that it takes a point and returns true or false. The method encapsulates the other information such as the if statement, that uses MyRectangle's properties to check if they match with the point provided. Any code that calls this method can only access the parameter to be passed in (point in this case) and can only see the result true or false.

Inheritance

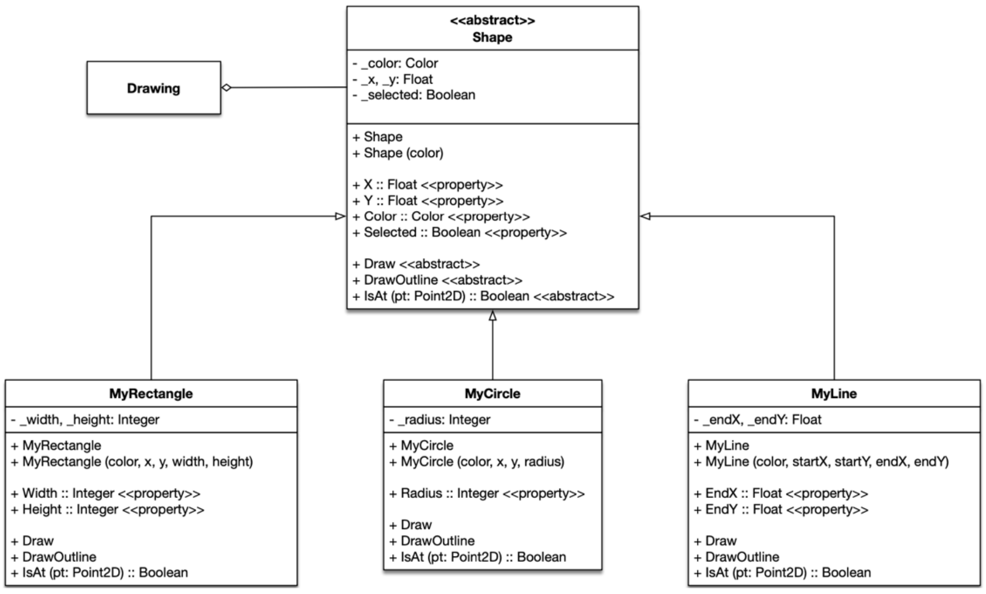
Inheritance is when a class declares that it has a parent class, and therefore is a kind of that class. This means that it will inherit the properties and behaviours of that parent class. The parent class does not know about its children or if it has any, but the children classes know if they have a parent class. Inheritance can work with abstraction to allow for polymorphism to occur. This can be achieved with the 'override' method, as long as the Parent class has defined that method as 'abstract'.

An example of inheritance in the Shape Drawing task:



```
public class MyRectangle : Shape
```

Here we can see MyRectangle is inheriting from Shape.
This means that it contains the same properties and methods of its parent class Shape. In this case, the child classes of Shape have different methods and internal parameters, which is allowed by Shape being an abstract class.



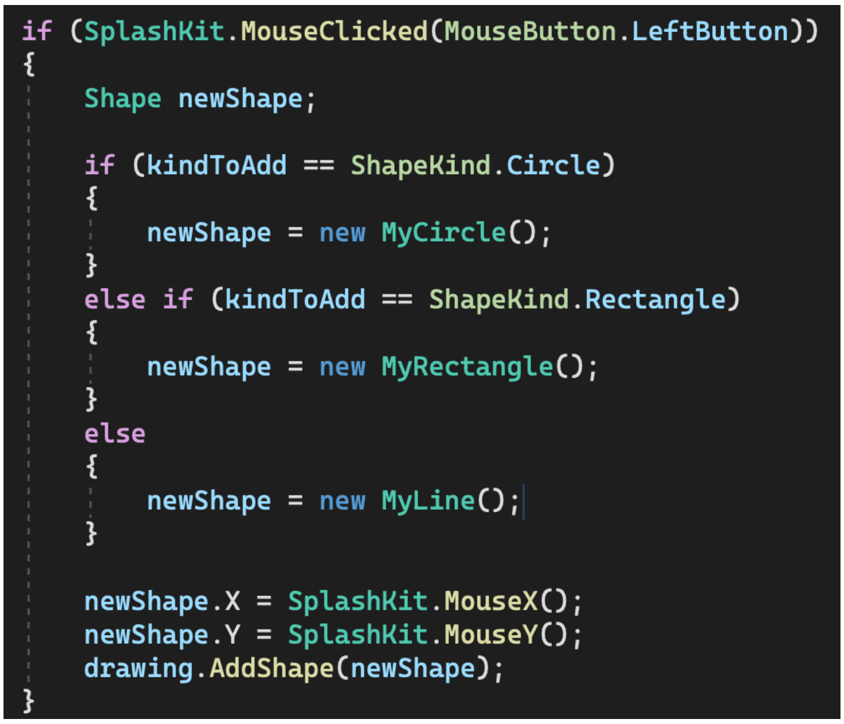
<<abstract>>
Shape
- _color: Color
- _x, _y: Float
- _selected: Boolean
+ Shape
+ Shape (color)
+ X :: Float <<property>>
+ Y :: Float <<property>>
+ Color :: Color <<property>>
+ Selected :: Boolean <<property>>
+ Draw <<abstract>>
+ DrawOutline <<abstract>>
+ IsAt (pt: Point2D) :: Boolean <<abstract>>
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

In this UML diagram, we can see that MyRectangle, MyCircle and MyLine are all children of Shape. This is represented by a solid line with an empty arrowhead pointing to the parent class.

Polymorphism

‘Polymorphism means "**many forms**", and it occurs when we have many classes that are related to each other by inheritance.’ -
https://www.w3schools.com/java/java_polymorphism.asp
Polymorphism is when you use an instance of a child class in the place of a parent class. This means that where a parent class is expected, you can use a different class in the place of that class as long as it has inherited from the parent.

An example of polymorphism is in the Shape Drawing task, where we can add different types of shapes, where a shape is expected, but we can use a specific type of shape in this place. This also means that we can only access the methods that Shape allows us to access, and any additional methods inside our specific shape type cannot be accessed, however, override methods will still be used from the specific type of shape.



```
if (SplashKit.MouseClicked(MouseButton.LeftButton))
{
    Shape newShape;

    if (kindToAdd == ShapeKind.Circle)
    {
        newShape = new MyCircle();
    }
    else if (kindToAdd == ShapeKind.Rectangle)
    {
        newShape = new MyRectangle();
    }
    else
    {
        newShape = new MyLine();
    }

    newShape.X = SplashKit.MouseX();
    newShape.Y = SplashKit.MouseY();
    drawing.AddShape(newShape);
}
```

Inside of this if statement, we can see that newShape is a type of Shape, yet we can initialise this variable with a circle, rectangle or line object which are all children of the shape class. This is a prime example of polymorphism.