Summarizing the content:

1. Summarise the main points in this module. You may include references to the learning objectives.

1) No clear boundaries between analysis and design.
2) This includes both a low-level component and algorithm design and a high-level, architecture design.
3) Reduction of Dependencies.
4) High-level modules should not depend on low-level modules.
5) Abstraction should not depend on details, details should depend on abstraction.
6) Classes should be open for extension, but closed for modification.
7) Inheritance: common state and behaviour in one place (called the parent or the superclass) and different state and behaviour in the individual classes that inherit from the parent (called child or subclass).
8) Inheritance (Sub-class).
9) Subtypes are substitutable for supertypes.
10) Subtyping without inheritance: Interfaces, Inheritance without subtyping snterfaces: Composition.
11) Interface Segregation Principle.
12) Single Responsibility Principle.

2. How is this useful?

1) Dependency Inversion Principle: Reduction of Dependencies.
2) Open-Closed Principle: Write code in a way that does not needs to be changed anytime the requirement of your program changes.
3) Liskov Substitution Principle: The extension of the class does not introduce new errors into the existing system, reducing the possibility of code errors. Strengthen the robustness of the program, at the same time, it can achieve very good compatibility when changing, improve the maintainability and scalability of the program, and reduce the risk introduced when the requirements are changed.
4) Interface Segregation Principle: This removes implicit coupling between the clients, that is present due to shared interface.
5) Single Responsibility Principle: Implement code with high cohesion and low coupling.

3. How do you plan to use this information?

When writing code in the future, think more and analyze more before starting to write, write the entire code framework first, analyze the design patterns of those modules, and finally write the code.

4. Provide summary of your reading list — external resources, websites, book chapters, code libraries, etc.

1) https://dev.to/tamerlang/understanding-solid-principles-dependency-inversion-1b0f
2) https://en.wikipedia.org/wiki/Open%E2%80%93closed_principle

3) https://www.alpharithms.com/liskov-substitution-principle-lsp-solid-114908/
4) https://en.wikipedia.org/wiki/Interface_segregation_principle
5) https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design

Reflecting on the content:

1. What is the most important thing you learnt in this module?

I think the most important thing in this module is the programming idea, do my best to achieve high cohesion and low coupling of the code, and enhance the readability and portability of the code.

2. How does this relate to what you already know?

Before, I only knew that there were subclasses. After learning this module, I learned that there are subtypes. They can be implemented by inheritance through subclasses, but subtypes can be implemented without inheritance. It can also appear in any parent class place to replace the parent class.

3. Reflect on the code that was given to you in the lab. You can take the screen shot of your python code and add image or just provide the code as text in your report. A good reflection includes:

Dependency Inversion Principle:

```
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");

            ButtonClient lamp = new Lamp();
            Button b1 = new Button(lamp);
            b1.detect();

            ButtonClient motor = new Motor();
            Button b2 = new Button(motor);
            b2.detect();
        }
    }
```

In the main function, two buttonclient objects are instantiated, namely lamp and motor.

```
    abstract class ButtonClient {
        abstract public void TurnOn();
        abstract public void TurnOff();
    }
```

The abstract class buttonclient is defined, which conforms to the principle of DIP. Abstraction should not depend on details, details should depend on abstraction.

```
    class Lamp : ButtonClient {

        override public void TurnOn() {
            Console.WriteLine("Turning on the Lamp");
        }

        override public void TurnOff() {
            Console.WriteLine("Turning off the Lamp");
        }

    }
```

Inherit the methods of the abstract parent class and override the methods of the parent class.

```
    class Button {

        ButtonClient buttonClient;

        public Button(ButtonClient buttonClient) {
            Console.WriteLine("I am in the constructor of Button");
            this.buttonClient = buttonClient;
        }

        public void detect() {
            Boolean buttonOn = false;

            if (buttonOn) {
                buttonClient.TurnOn();
            } else {
                buttonClient.TurnOff();
            }
        }

    }
}
```

Depends on the buttonclient class and implements the detect function.
The disadvantage is that it is difficult to abstract the method.

Open-Closed Principle:

```
public class AreaCalculator
{
    public double Area(Rectangle[] shapes)
    {
        double area = 0;
        foreach (var shape in shapes)
        {
            area += shape.Width*shape.Height;
        }

        return area;
    }
}
```

This method can only calculate the area of the rectangle.

```
public class AreaCalculator_Iter2
{
    public double Area(object[] shapes)
    {
        double area = 0;
        foreach (var shape in shapes)
        {
            if (shape is Rectangle)
            {
                Rectangle rectangle = (Rectangle) shape;
                area += rectangle.Width*rectangle.Height;
            }
            else
            {
                Circle circle = (Circle)shape;
                area += circle.Radius * circle.Radius * Math.PI;
            }
        }
```

This method can calculate the area of rectangles and circles, but it cannot calculate the area of triangles. If you add triangles, you must rewrite the AreaCalculator class

```
    public class Circle
        {
        public double bottom { get; set; }
        public double Height { get; set; }

    }
```

define triangle class.

```
//An abstract computational superclass
public  abstract class  AreaCalculator
    {

    public abstract double Area();

}
```

Define an abstract computational superclass.

```
//Calculate the area of a rectangle
public class rectangle : AreaCalculator
{

    public double Area(object[] shapes)
    {
        Rectangle rectangle = (Rectangle) shape;
        area += rectangle.Width*rectangle.Height;
        return area

    }
}

//Calculate the area of a circle
public class circle : AreaCalculator
{
    public override double GetResult()
    {
        Circle circle = (Circle)shape;
        area += circle.Radius * circle.Radius * Math.PI;
        return area
    }
}

//Calculate the area of a circle
public class Triangle : AreaCalculator
{
    public override double GetResult()
    {
        Triangle triangle = (Triangle)shape;
        area += triangle.bottom * triangle.height / 2
        return area
    }
}
```

Three ways to calculate the area of a shape.

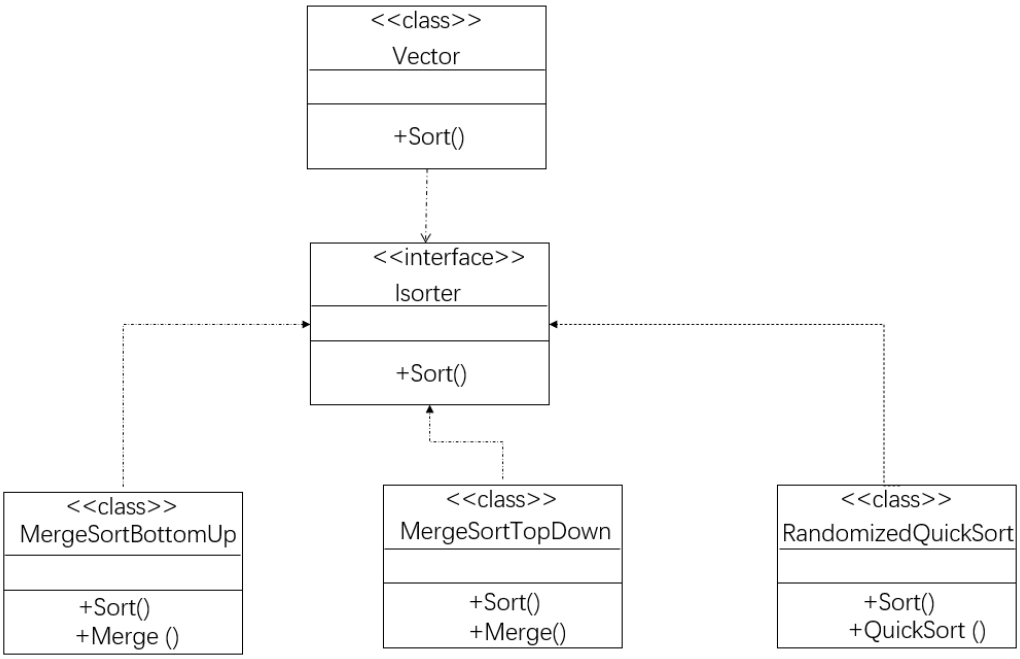
```
public class AreaCalculator_Iter3
{
    public double Area(object[] shapes)
    {
        double area = 0;
        foreach (var shape in shapes)
        {
            if (shape is Rectangle)
            {
                area = new Rectangle()
            }
            if (shape is Circle)
            {
                area = new Circle()
            }
            if (shape is Triangle)
            {
                area = new Triangle()
            }
        }

        return area;
    }
}
```

class to calculate area.

Submit a solution to all the activities that you did as part of this module.
In SIT221, for task 3.2D, you were expected to implement various sorting techniques. You were given some code, specifically: Vector.cs, ISorter.cs. We have given you a dummy solution file MergeSortBottomUp.cs (you can create dummy MergeSortTopDown.cs yourself). Given these three files, analyse the code with-in and identify the DIP and OCP principles, if they exist. If they do not, you can modify the code to demonstrate your understanding of DIP and OCP. Make sure you can draw a valid UML to demonstrate these principles. You can access the files here: Code Handout. Attaching the task 3.2D for completeness sake or in case you have lost access to 221 ontrack: 3.2D from SIT221.

The UML diagram looks like this:



The DIP principle is reflected in the following code:

```
namespace Vector
{
    5 references
    public interface ISorter
    {
        3 references
        void Sort<K>(K[] sequence, IComparer<K> comparer) where K : IComparable<K>;
    }
}
```

The Isort class implements abstract methods, and subclasses are responsible for implementing them.

OCP principles are reflected in the Vector.cs file.

```
public override string ToString()
{
    string result = $"{data[0]}";
    for (int i = 1; i < Count; i++)
    {
        result = string.Format($"{result}, {data[i]}");
    }
    //foreach (var item in data)
    //{
    //    result = string.Format($"{result}, {item}");
    //}
    return string.Format($"[{result}]");

}

0 references
public string Print()
{
    string result = "";
    foreach (var item in data)
    {
        result = string.Format($"{result}, {item}");
    }
    return result;
}
```

Override and adding methods in vector.cs can expand the functions of vector.cs and print out the information in Vector.

The result printed by the code is as follows:

```
Intital data: [333, 236, 312, 780, 100, 722, 511, 966, 213, 724, 122, 120, 263, 175, 752, 958, 596, 299, 995, 772]
Resulting order: [995, 966, 958, 780, 772, 752, 724, 722, 596, 511, 333, 312, 299, 263, 236, 213, 175, 122, 120, 100]
 :: SUCCESS

Test C: Sort integer numbers applying RandomizedQuickSort with EvenNumberFirstComparer:
Intital data: [333, 236, 312, 780, 100, 722, 511, 966, 213, 724, 122, 120, 263, 175, 752, 958, 596, 299, 995, 772]
Resulting order: [772, 236, 312, 780, 100, 722, 596, 966, 958, 724, 122, 120, 752, 175, 263, 213, 511, 299, 995, 333]
 :: SUCCESS

Test D: Sort integer numbers applying MergeSortTopDown with AscendingIntComparer:
Intital data: [333, 236, 312, 780, 100, 722, 511, 966, 213, 724, 122, 120, 263, 175, 752, 958, 596, 299, 995, 772]
Resulting order: [100, 120, 122, 175, 213, 236, 263, 299, 312, 333, 511, 596, 722, 724, 752, 772, 780, 958, 966, 995]
 :: SUCCESS

Test E: Sort integer numbers applying MergeSortTopDown with DescendingIntComparer:
Intital data: [333, 236, 312, 780, 100, 722, 511, 966, 213, 724, 122, 120, 263, 175, 752, 958, 596, 299, 995, 772]
Resulting order: [995, 966, 958, 780, 772, 752, 724, 722, 596, 511, 333, 312, 299, 263, 236, 213, 175, 122, 120, 100]
 :: SUCCESS

Test F: Sort integer numbers applying MergeSortTopDown with EvenNumberFirstComparer:
Intital data: [333, 236, 312, 780, 100, 722, 511, 966, 213, 724, 122, 120, 263, 175, 752, 958, 596, 299, 995, 772]
Resulting order: [772, 596, 958, 752, 120, 122, 724, 966, 722, 100, 780, 312, 236, 995, 299, 175, 263, 213, 511, 333]
 :: SUCCESS

Test G: Sort integer numbers applying MergeSortBottomUp with AscendingIntComparer:
Intital data: [333, 236, 312, 780, 100, 722, 511, 966, 213, 724, 122, 120, 263, 175, 752, 958, 596, 299, 995, 772]
Resulting order: [100, 120, 122, 175, 213, 236, 263, 299, 312, 333, 511, 596, 722, 724, 752, 772, 780, 958, 966, 995]
 :: SUCCESS

Test H: Sort integer numbers applying MergeSortBottomUp with DescendingIntComparer:
Intital data: [333, 236, 312, 780, 100, 722, 511, 966, 213, 724, 122, 120, 263, 175, 752, 958, 596, 299, 995, 772]
Resulting order: [995, 966, 958, 780, 772, 752, 724, 722, 596, 511, 333, 312, 299, 263, 236, 213, 175, 122, 120, 100]
 :: SUCCESS

Test I: Sort integer numbers applying MergeSortBottomUp with EvenNumberFirstComparer:
Intital data: [333, 236, 312, 780, 100, 722, 511, 966, 213, 724, 122, 120, 263, 175, 752, 958, 596, 299, 995, 772]
Resulting order: [772, 596, 958, 752, 120, 122, 724, 966, 722, 100, 780, 312, 236, 995, 299, 175, 263, 213, 511, 333]
 :: SUCCESS


------------------ SUMMARY ------------------
Tests passed: ABCDEFGHI
```

2. Critically comment on the use of inheritance in programming languages. Argue a case for or against its use.

I support the use of inheritance in programming languages. Inheritance can extend existing code modules, improve programming efficiency, and improve code reuse rate. A subclass inherits a parent class, which means that it inherits all its accessible members, and can also extend new members by itself. It should be noted that the inheritance of C# classes can only be single-inherited, and a parent class can have multiple subclasses, but that is, a subclass can only inherit from one parent class.

But inheritance also has disadvantages. For example, creating a base animal class that includes methods such as running, flying, and swimming, but not all animals can fly and swim. For example, cats cannot fly. So inheritance increases the coupling between subclasses and superclasses.