***Section 1 - Introduction***

**Design Patterns:**

In programming, we encounter various problems and challenges while writing and developing software. There are standard and reliable solutions to these problems, known as Design Patterns.

These patterns are the result of many years of investigation and research by prominent programmers worldwide. Design Patterns empower you to overcome programming challenges and implement scalable, modern, and clean software.

**Why Use Design Patterns?**

1. Improved software development
2. Modern and maintainable software
3. Clean and structured code
4. Reusable code
5. Enhanced software flexibility
6. Time-saving solutions

**How Are Design Patterns Created?**

Design Patterns are not created by a specific person or company but have evolved through contributions from various programmers.

In 1993, four individuals identified the 24 most effective and widely used design patterns. These pioneers are known as the Gang of Four (GoF).

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

***Section 2 - What is UML and its types?***

UML (Unified Modeling Language) is a simple graphical modeling language used for documenting software systems. It provides various diagrams to help with software documentation.

**UML Diagrams:**

1. Structure Diagrams
2. Behavioral Diagrams
3. Interaction Diagrams

**- Types of Structural Diagrams:**

1. Class Diagram
2. Object Diagram
3. Composite Structure Diagram
4. Component Diagram
5. Deployment Diagram
6. Package Diagram

We use these diagrams to understand design patterns. In this course, we use C# to implement these design patterns, but the diagrams help us apply them in other programming languages as well.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

***Section 3 - Required Software***

To work with UML diagrams and implement design patterns in C#, you may need the following software:

For UML diagrams we use => Drawio.com (we can use download for app or online in site)

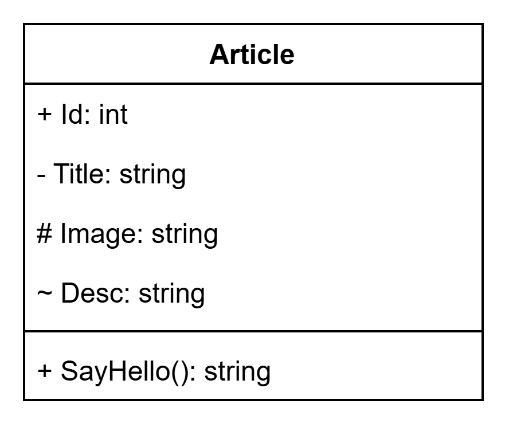
And for coding with C#, we need **Visual Studio**, **VS Code**, or **Rider**.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

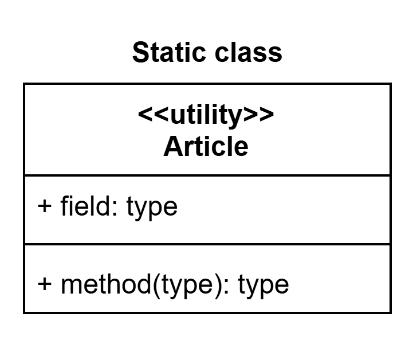
***Section 4 - Class and Interface Diagrams in UML***

In this section, we are going to use **Diagram IO** and describe its features.

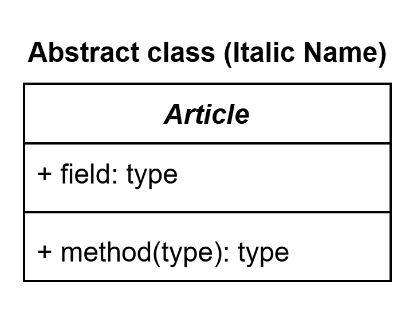
1. **Creating a Class:**
   * Drag a new class from the shapes menu.
   * Give it a name, such as **Article**.
   * Add properties to the class.
2. **Access Modifiers:**  
   In UML, the symbol before a property or method represents its access level:
   * **+ (Public)** → Accessible from anywhere
   * **- (Private)** → Accessible only within the class
   * **# (Protected)** → Accessible within the class and its subclasses
   * **~ (Internal / Package Private)** → Accessible within the same package
3. **Declaring Properties:**
   * Example: + Id: int
   * This means: **public int Id;** in C#.
4. **Declaring Methods:**
   * Example: + methodName(paramType): returnType
   * The type inside () represents the **input parameters**, and the type after : represents the **return type**.
   * Example: + GetArticle(id: int): Article
     + This means:  
       public Article GetArticle(int id);



This is a sample of a class with some properties and one method.  
If we want to create a static class, we should write the name of class like this:



And if we want to create an Abstract Class, we Should Italic the name of Class:



Also, for property or Field if we want that field be abstract, we italic the name of field or property.

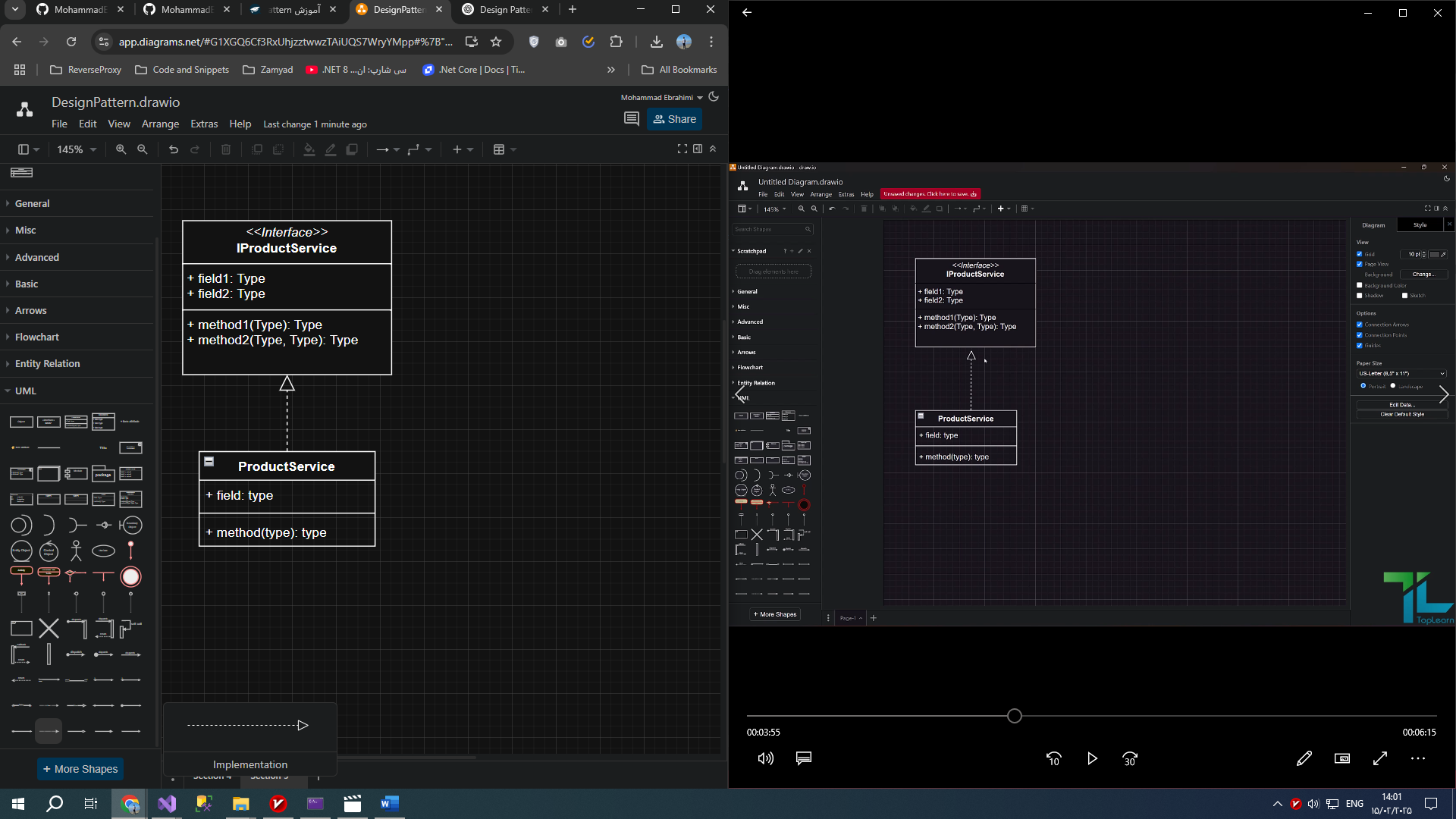
For interface we can use interface or class from UML tool shapes. But with one condition we must write up the name of interface, «interface» to specify that class is interface. If you just add 'I' in first of class name doesn't make it interface and must put «interface» at the top of the name. As class or interface shape :



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

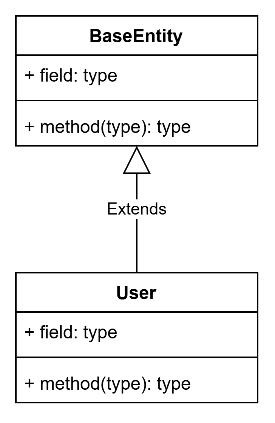
***Section 5 Relationships in UML***

In this Section we create an interface name `IProductService` and class name `ProductService`. Now we use an arrow from shapes in UML name Implementation to say ProductService Implement IProductService.

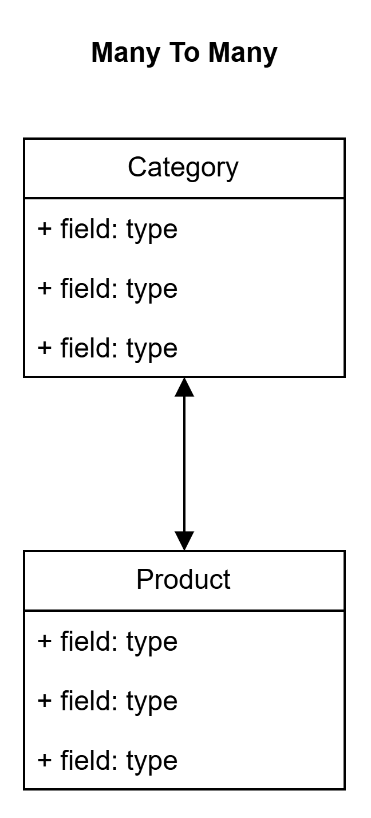


The triangle connects to interface and the end of line connect to class.

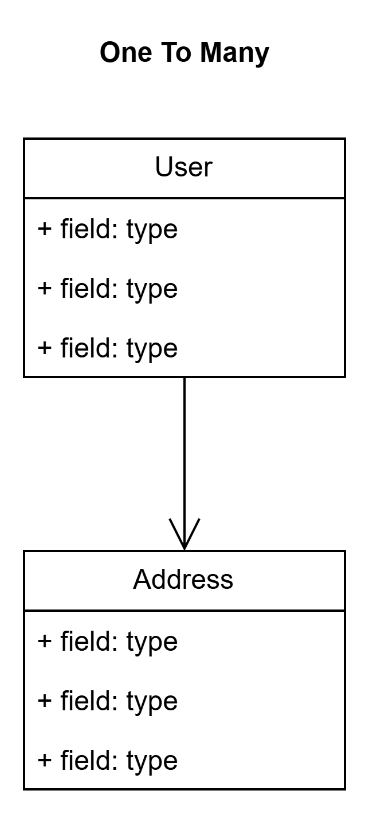
To represent inheritance in UML, we select the **Generalization** arrow from the **Shape** section. The arrowhead, which is an empty triangle, is connected to the parent class, and the other end of the arrow is connected to the child class.



For Many-to-Many Relations:



For One-to-Many Relations:



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

***Section 6 Decorator Pattern - Part 1***

In this section we introduction the Decorator pattern.

The **Decorator Design Pattern** is a structural pattern used to dynamically add new behavior to an object without modifying its existing code. This is particularly useful when working with legacy systems where directly altering the base class could cause unforeseen issues due to dependencies.

**Why Use the Decorator Pattern?**

In some cases, modifying an existing class can lead to unexpected side effects because that class might be used in multiple places. Instead of changing the core class, the Decorator Pattern allows us to wrap the original object inside a new class that adds the desired behavior while keeping the existing functionality intact.

**Example Scenario**

Imagine you have a Product class with a method called purchase(). When a user calls this method, the product gets purchased. Now, the business requires that after purchasing, a confirmation SMS should be sent to the customer.

**Two Possible Approaches:**

1. **Modify the purchase() Method Directly**
   * This might not be ideal because it affects all existing usages of the method.
   * If other parts of the system rely on purchase() working a certain way, they might break.
2. **Use the Decorator Pattern**
   * Instead of modifying the purchase() method, we create a **PurchaseWithSMSDecorator** class.
   * This decorator will call the original purchase() method and then send an SMS.
   * This approach ensures that we only add SMS functionality when needed, without affecting the base class.

**Key Concept of the Decorator Pattern**

* It **wraps** an existing class to enhance its behavior.
* It keeps the original object unchanged while adding new functionality.
* It promotes code reusability and flexibility in software design.

This pattern is also known as the **Wrapper Pattern** in the developer community because it "wraps" an object with additional functionality.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

***Section 7 Decorator Pattern - Part 2***

