**ASSIGNMENT 1 FRONT SHEET**

|  |  |  |  |
| --- | --- | --- | --- |
| **Qualification** | **BTEC Level 5 HND Diploma in Computing** | | |
| **Unit number and title** | Unit 20: Advanced Programming | | |
| **Submission date** | 22/06/2023 | **Date Received 1st submission** |  |
| **Re-submission Date** |  | **Date Received 2nd submission** |  |
| **Student Name** | Ngo Thanh Tung | **Student ID** | GCH220075 |
| **Class** | GCH1101 | **Assessor name** | Do Hong Quan |
| **Student declaration**  I certify that the assignment submission is entirely my own work and I fully understand the consequences of plagiarism. I understand that making a false declaration is a form of malpractice. | | | |
|  |  | **Student’s signature** | Tung |

**Grading grid**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| P1 | P2 | M1 | M2 | D1 | D2 |
|  |  |  |  |  |  |

|  |  |  |
| --- | --- | --- |
| **❒ Summative Feedback: ❒ Resubmission Feedback:** | | |
| **Grade:** | **Assessor Signature:** | **Date:** |
| **Lecturer Signature:** | | |

Table of Content

1. Introduction

2. OOP general concepts

2.1 What is Object-oriented Programming?

Object-oriented programming (OOP) is a computer programming model that organizes software design around data, or objects, rather than functions and logic. An object can be defined as a data field that has unique attributes and behavior.

OOP focuses on the objects that developers want to manipulate rather than the logic required to manipulate them. This approach to programming is well-suited for programs that are large, complex and actively updated or maintained. This includes programs for manufacturing and design, as well as mobile applications; for example, OOP can be used for manufacturing system simulation software.

The organization of an object-oriented program also makes the method beneficial to collaborative development, where projects are divided into groups. Additional benefits of OOP include code reusability, scalability and efficiency.

(Alexander S. Gillis. 2021)

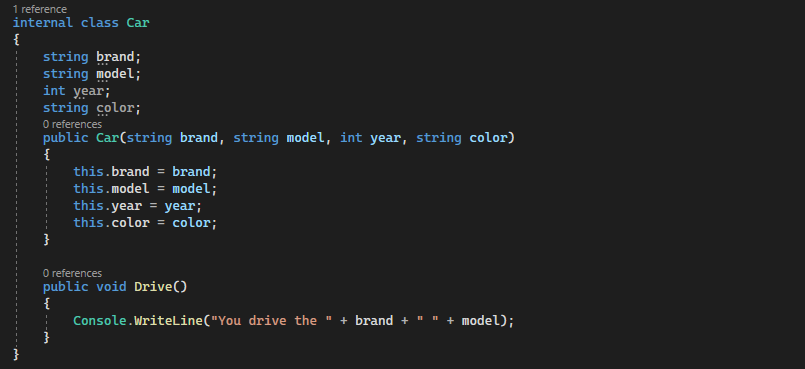
2.2 Structure of Object-oriented Programming

Object-oriented programming (OOP) is a programming paradigm that organizes code into reusable and modular structures called classes and objects. It promotes the concept of objects, which encapsulate data (attributes) and behavior (methods) into a single entity. The structure, or building blocks, of object-oriented programming include the following:

* **Classes** are user-defined data types that act as the blueprint for individual objects, attributes and methods.
* **Objects** are instances of a class created with specifically defined data. Objects can correspond to real-world objects or an abstract entity. When class is defined initially, the description is the only object that is defined.
* **Methods** are functions that are defined inside a class that describe the behaviors of an object. Each method contained in class definitions starts with a reference to an instance object. Additionally, the subroutines contained in an object are called instance methods. Programmers use methods for reusability or keeping functionality encapsulated inside one object at a time.
* **Attributes** are defined in the class template and represent the state of an object. Objects will have data stored in the attributes field. Class attributes belong to the class itself.
* **Constructor** is a method subtype. This method is called automatically when the object is created. Constructors do not return values.

(Alexander S. Gillis. 2021)

Code example of an object



2.3 General concepts of Object-oriented Programming

The main ideas behind C#’s Object-Oriented Programming, OOP concepts include abstraction, encapsulation, inheritance and polymorphism. Basically, C# OOP concepts let us create working methods and variables, then re-use all or part of them without compromising security.

2.3.1 Encapsulation

a. What is encapsulation in OOP?

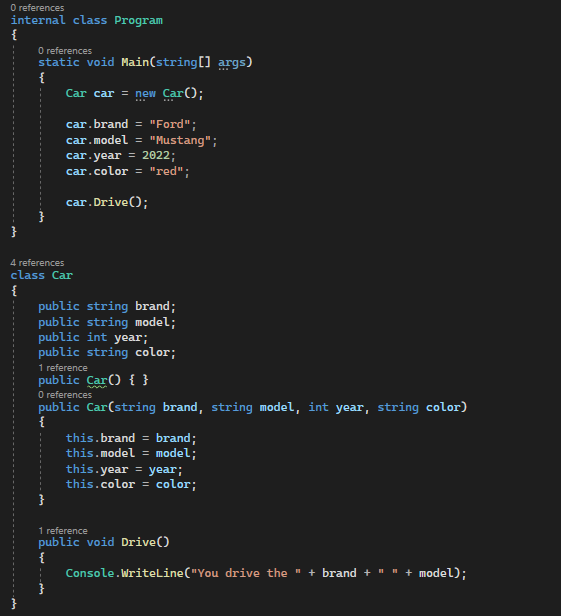
Encapsulation in object-oriented programming (OOP) is a mechanism of hiding the internal details (implementation) of an object from other objects and the outside world. It allows an object to control access to its data and methods, which can improve the security and stability of the system. In OOP, encapsulation is typically achieved through the use of access modifiers which restrict access to certain members of a class.

b. Access specifiers

Encapsulation is implemented by using access specifiers. An access specifier defines the scope and visibility of a class member. C# supports the following access specifiers:

* Public
* Private
* Protected
* Internal
* Public Access Specifier

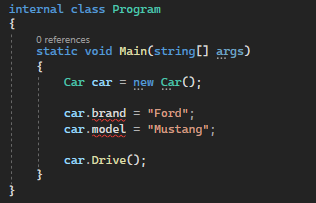
Public access specifier allows a class to expose its member variables and member functions to other functions and objects. Any public member can be accessed from outside the class.



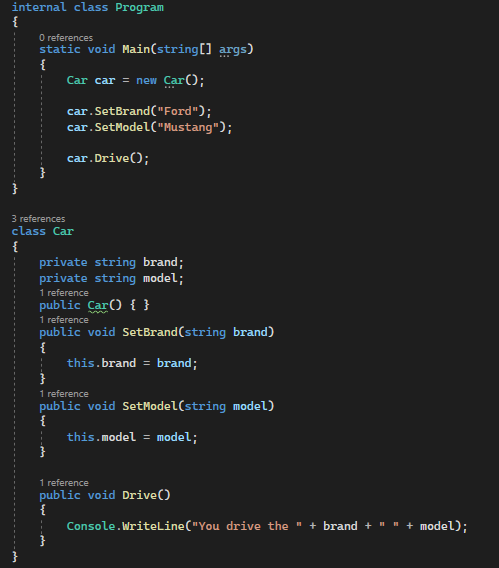
In the example, the member variables brand, model, year and color are declared public, so they can be accessed from the function Main() using an instance of the Car class, named “car”. The member functions Display() is also declared public, so it can also be accessed from Main() using an instance of the Car class, named “car”.

* Private Access Specifier

Private access specifier allows a class to hide its member variables and member functions from other functions and objects. Only functions of the same class can access its private members. Even an instance of a class cannot access its private members.



The member variables brand and model are declared private so they cannot be accessed from the function Main()



The member functions SetBrand(), SetModel and Display() can access these variables. Since those member functions are declared public, they can be accessed from Main() using an instance of the Car class, named “car”.

* Protected Access Specifier

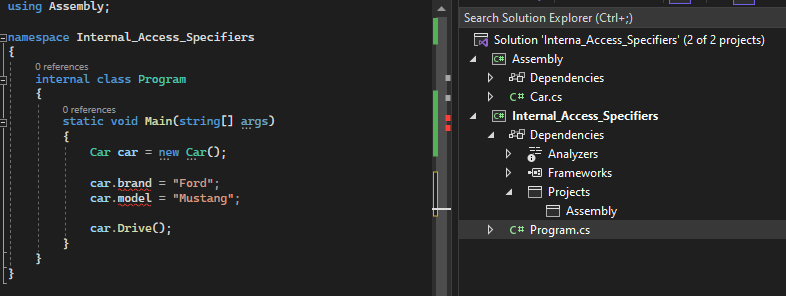
Protected access specifier allows a child class to access the member variables and member functions of its base class. This way it helps in implementing inheritance.

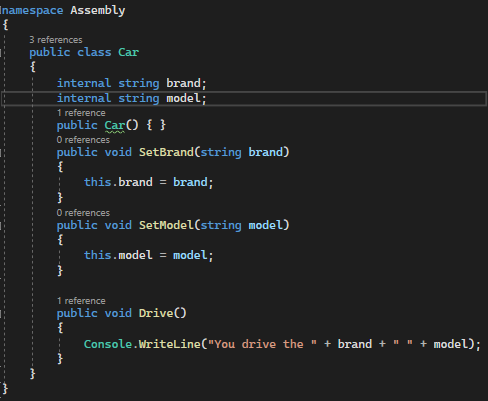


When we declare a type member as protected, we are not able to access it from the Program class. However, if we inherited the Program class from the Car class, the protected member can be accessed from derived classes, we are able to access name from the Program class.

* Internal Access Specifier

When we declare a type or type member as internal, it can be accessed only within the same assembly. An assembly is a collection of types (classes, interfaces, etc.) and resources (data). They are built to work together and form a logical unit of functionality. That's why when we run an assembly all classes and interfaces inside the assembly run together.

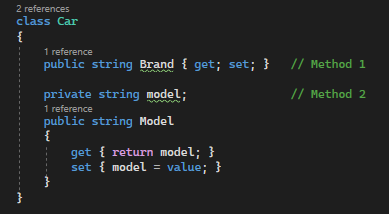




If the Car class attributes are set to public access modifier, the class Program is able to access it but set to public, but when it changes to internal access modifier it will no longer able to access.

c. How can we implement encapsulation in C# using properties?

Here is an example of how to implement encapsulation in C# using properties:



In this example, the private field "model" is encapsulated by the public property "Model." The "get" accessor of the property allows other objects to retrieve the value of the private field, while the "set" accessor allows them to modify it.

2.3.2 Inheritance

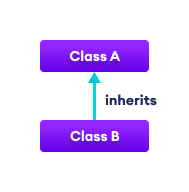
a. What is Inheritance?

Inheritance is one of the key features of object-oriented programming (OOP). It is simply the process by which one class (the child or derived class) acquires the properties, methods, and fields of another class (the base, parent, or super class). Inheritance in object-oriented programming means that you're creating classes that can pass down their properties to other classes without having to explicitly define the properties in new classes. Inheritance does not only ensure the reusability of the codebase, but it also reduces your code’s complexity.

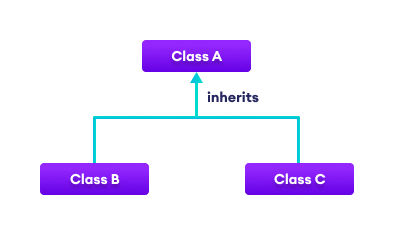
b. Types of Inheritance in C#

In C#, there are various types of inheritance which will be discuss following:

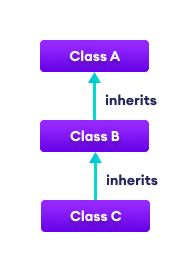
* **Single Inheritance:** Single inheritance usually occurs between two classes – the base class, and the derived class. It occurs when a class is inherited from a single-parent class.



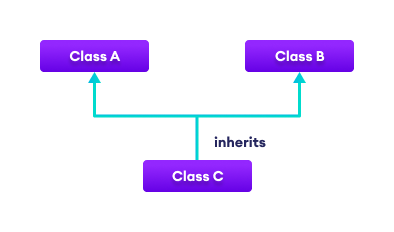
* **Hierarchical Inheritance:** Hierarchical inheritance occurs when more than one derived class is created from a single-parent class.



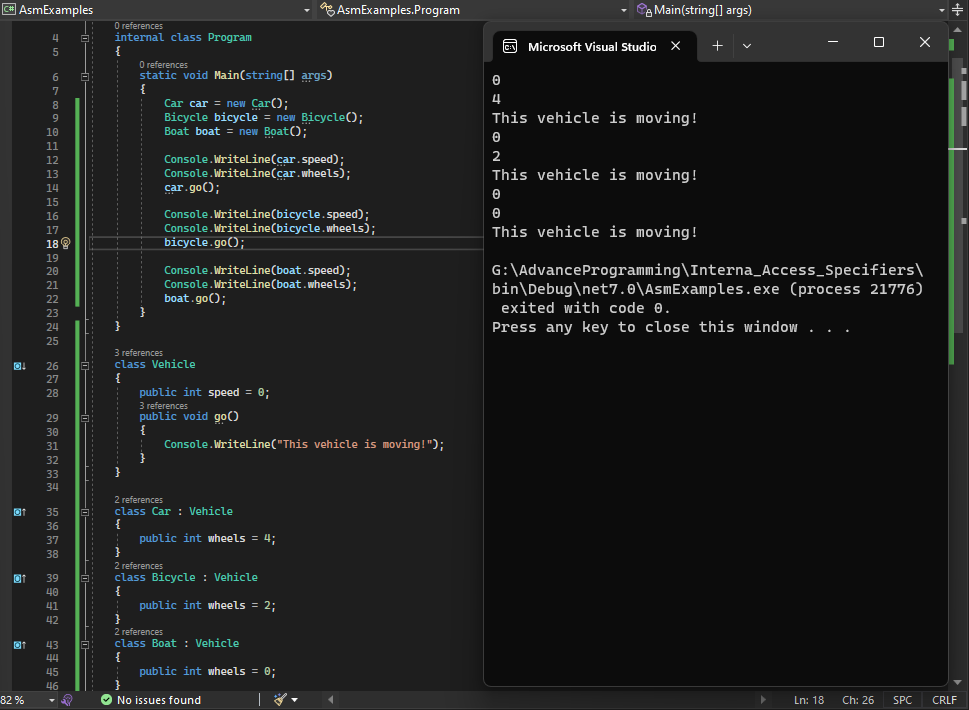
* **Multi-level Inheritance:** Multi-level inheritance occurs when a class is derived from another derived class. It is simply a situation where a derived class is created and used as a base class for another class.



**Multiple Inheritances – Interfaces:** Multiple inheritances allow a derived class to be inherited from multiple parent classes. (Multiple inheritances are not supported in C#. But you can achieve it by using interfaces)



c. Code Example



In the above example, the go() method is present in both the base class and derived class.

2.3.3 Polymorphism

a. What is Polymorphism?

Polymorphism is one of the fundamental OOPs concepts and is a term used to describe situations where something takes various roles or forms. In the programming world, these things can be operators or functions.

The word polymorphism is derived from two Greek words: poly and morphs. The word “Poly” means many and “morphs” means forms. Therefore, polymorphism means “many forms” or we can say that the word polymorphism means the ability to take more than one form. That is one thing that can take many forms.

Polymorphism is a concept by which we can perform a single task in different ways. That is, when a single entity behaves differently in different cases, it is called polymorphism in C#. The term polymorphism is an object-oriented programming term that means a function, or an operator behaves differently in different scenarios.

b. Types of Polymorphism

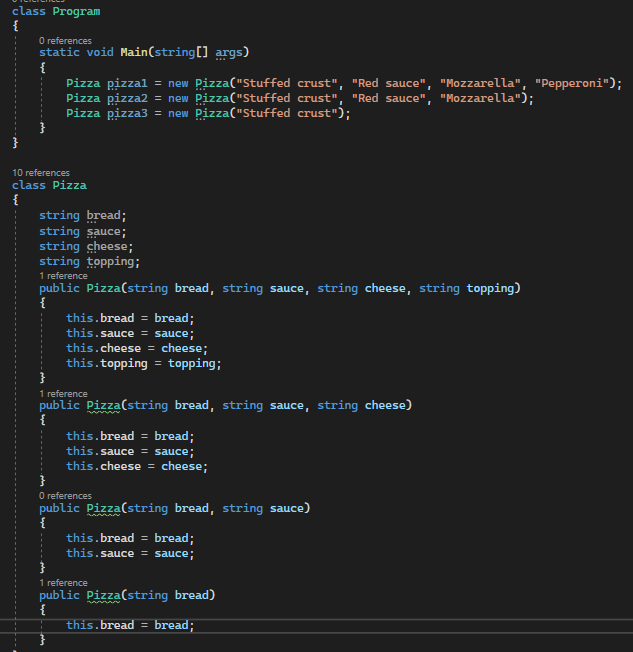
There are two types of polymorphism in C#:

* Static / Compile Time Polymorphism
* Dynamic / Runtime Polymorphism

**Compile Time Polymorphism**

Method overloading is an example of Static polymorphism. Overloading is the concept in which method names are the same with different parameters. The method/function has the same name but different signatures in overloading. It is also known as Early binding. It is also known as Compile Time Polymorphism because the decision of which method is to be called is made at compile time.

Here C# compiler checks the number of parameters passed and the parameter type, decides which method to call, and throws an error if no matching method is found.

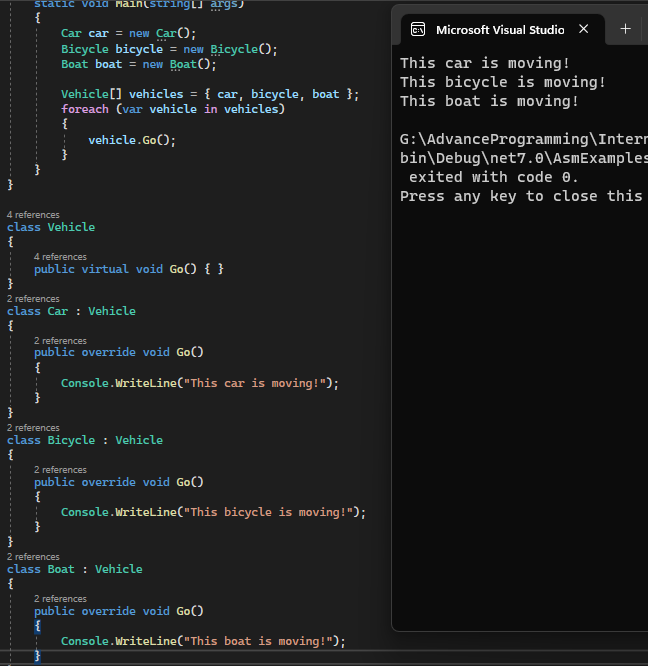


Here we have different types and numbers of parameters in the class Pizza constructor. This is known as method overloading in C#. The same method will perform different operations based on the parameter.

**Run-Time Polymorphism**

Dynamic/runtime polymorphism is also known as late binding. Here, the method name and the method signature (the number of parameters and parameter type must be the same and may have a different implementation). Method overriding is an example of dynamic polymorphism.

Method overriding can be done using inheritance. With method overriding, it is possible for the base class and derived class to have the same method name and the same something. The compiler would not be aware of the method available for overriding the functionality, so the compiler does not throw an error at compile time. The compiler will decide which way to call at runtime, and if no method is found, it throws an error.



In the above example, we have created a superclass: Vehicle and a subclass: Car, Bicycle and Boat. Notice, we have used virtual and override with methods of the base class and derived class respectively.

* virtual - allows the method to be overridden by the derived class
* override - indicates the method is overriding the method from the base class

2.3.4 Abstraction

a. What is abstraction?

Objects only expose internal mechanisms that are essential for the use of other objects, while masking any extraneous implementation code. The functionality of the derived class can be expanded. This notion can make it easier for developers to make further adjustments or additions over time.

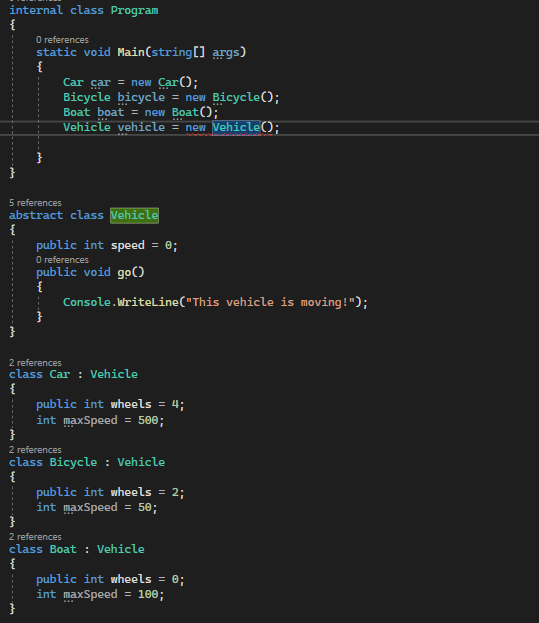
An abstract class is an incomplete class or special class we can't be instantiated. The purpose of an abstract class is to provide a blueprint for derived classes and set some rules what the derived classes must implement when they inherit an abstract class.

We can use an abstract class as a base class and all derived classes must implement abstract definitions. An abstract method must be implemented in all non-abstract classes using the override keyword. After overriding the abstract method is in the non-Abstract class. We can derive this class in another class and again we can override the same abstract method with it.

b. C# Abstract Class Features

* An abstract class can inherit from a class and one or more interfaces.
* An abstract class can implement code with non-Abstract methods.
* An Abstract class can have modifiers for methods, properties etc.
* An Abstract class can have constants and fields.
* An abstract class can implement a property.
* An abstract class can have constructors or destructors.
* An abstract class cannot be inherited from by structures.
* An abstract class cannot support multiple inheritance.

c. Code Example



An abstract method is a method that does not have a body. To create abstract methods, we use the abstract keyword. In the above example, we have created an abstract class named Vehicle. We have an abstract method go() inside the class.

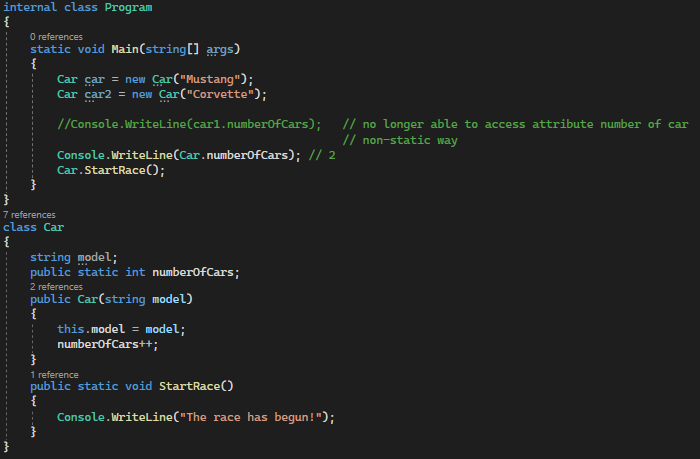
We have subclasses Car, Bicycle and Boat that inherits from the Vehicle class. Those classes provide the implementation of the abstract method go().

2.3.5 Static modifier

a. What is static in C#?

In C#, static means something which cannot be instantiated. You cannot create an object of a static class and cannot access static members using an object. C# classes, variables, methods, properties, operators, events, and constructors can be defined as static using the static modifier keyword.

b. Code example



In the above example, we have created a static variable named “numberOfCars”. Since the variable is static, we have used the class name Car to access the variable.

Just like static variables, we can call the static methods using the class name. In this example, we have accessed the static method directly from Program classes using the class name and when we declare a method static, all objects of the class share the same static method.

In C#, when we declare a class as static, we cannot create objects of the class. Furthermore, the field and method of the static class are also static because we can only have static members inside the static class. An example of a static class is a Math class.

2.3.6 Interface

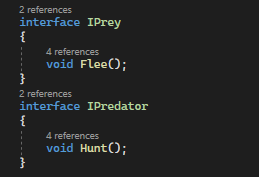
a. What is an Interface?

An interface looks like a class but has no implementation. The only thing it contains is declarations of events, indexers, methods, and/or properties. The reason interfaces only provide declarations is that they are inherited by structs and classes, which must provide an implementation for each interface member declared.

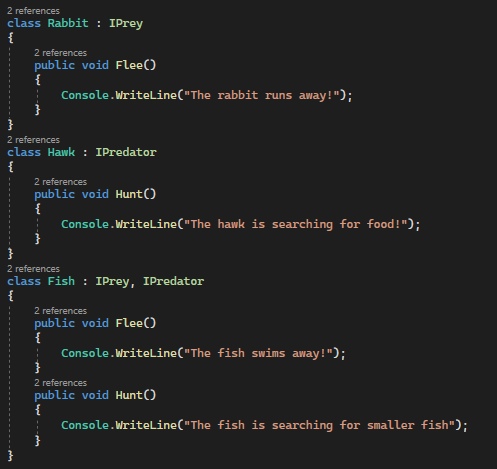
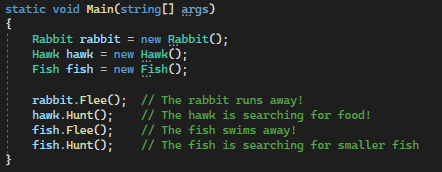
b. Code example

In C#, an interface is similar to abstract class. However, unlike abstract classes, all methods of an interface are fully abstract (method without body).

We use the “interface” keyword to create an interface.



To use an interface, other classes must implement it, we use “:” symbol to implement an interface.

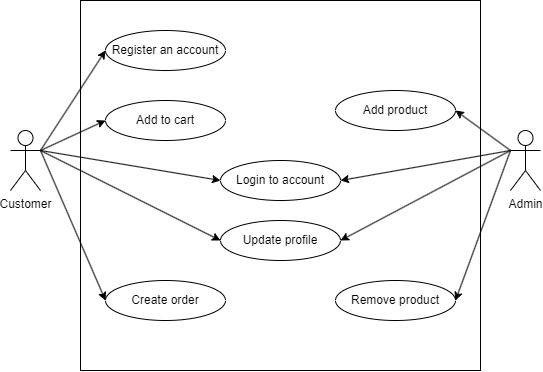


Unlike inheritance, a class can implement multiple interfaces. In the example above the class Fish have two interface IPrey and IPredator.

3. OOP scenario

3.1 Scenario

3.2 Use case Diagram



3.3 Class Diagram

3.3.1 What is Class Diagram?

In software engineering, a class diagram in the Unified Modeling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects. Code reuse is one of the many benefits of OOP. Reusability is feasible because of the various types of relationships that can be implemented among classes.

**a. Purpose of Class Diagrams**

1. Shows static structure of classifiers in a system
2. Diagram provides a basic notation for other structure diagrams prescribed by UML
3. Helpful for developers and other team members too
4. Business Analysts can use class diagrams to model systems from a business perspective

A UML class diagram is made up of:

* A set of classes and
* A set of relationships between classes

**b. Class Notation**

A class notation consists of three parts:

1. Class Name
   * The name of the class appears in the first partition.
2. Class Attributes
   * Attributes are shown in the second partition.
   * The attribute type is shown after the colon.
   * Attributes map onto member variables (data members) in code.
3. Class Operations (Methods)
   * Operations are shown in the third partition. They are services the class provides.
   * The return type of a method is shown after the colon at the end of the method signature.
   * The return type of method parameters is shown after the colon following the parameter name.
   * Operations map onto class methods in code

**c. Class Relationships**

In UML, object interconnections (logical or physical), are modeled as relationships. There are three kinds of relationships in UML:

* Dependencies
* Generalizations
* Associations

**Dependencies relationship**

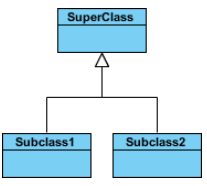
Exists between two classes if the changes to the definition of one may cause changes to the other (but not the other way around).



Class1 depends on Class2

**Inheritance relationship (or Generalization)**

It denotes an inheritance of attributes and behavior from the superclass to the subclass and indicates a specialization in the subclass of the more general superclass.



SubClass1 and SubClass2 are specializations of Super Class.

**Simple Association**

This relationship transpires when objects of one class know the objects of another class. The relationship can be one to one, one to many, many to one, or many to many. Moreover, objects can be created or deleted independently.



Association between Class1 and Class2

**Aggregation**

Is a "part-of" type relationship and is a one-way form of association. This relationship exists when a class owns but shares a reference to objects of another class.



Class2 is part of Class1

**Composition**

A special type of aggregation where parts are destroyed when the whole is destroyed.



Objects of Class2 live and die with Class1

4. Design Pattens

Design patterns are typical solutions to commonly occurring problems in software design. They are like pre-made blueprints that you can customize to solve a recurring design problem in your code.

Most patterns are described very formally so people can reproduce them in many contexts. Here are the sections that are usually present in a pattern description:

* **Intent** of the pattern briefly describes both the problem and the solution.
* **Motivation** further explains the problem and the solution the pattern makes possible.
* **Structure** of classes shows each part of the pattern and how they are related.
* **Code example** in one of the popular programming languages makes it easier to grasp the idea behind the pattern.

Some pattern catalogs list other useful details, such as applicability of the pattern, implementation steps and relations with other patterns.

(Alexander Shvets, 2021)

4.1 Creational pattern

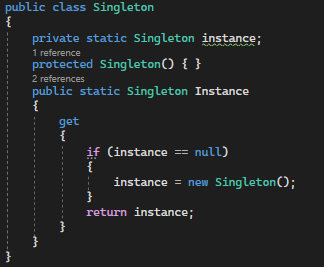
The creational patterns aim to separate a system from how its objects are created, composed, and represented. They increase the system’s flexibility in terms of the what, who, how, and when of object creation. Creational patterns encapsulate the knowledge about which classes a system uses, but they hide the details of how the instances of these classes are created and put together. Programmers have come to realize that composing systems with inheritance makes those systems too rigid. The creational patterns are designed to break this close coupling. (Judith Bishop. 2007)

Some common creational patterns include:

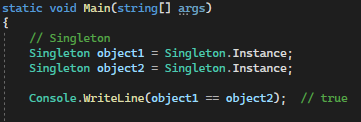
* **Factory Method:** Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.
* **Abstract Factory:** Lets you produce families of related objects without specifying their concrete classes.
* **Builder:** Lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.
* **Prototype:** Lets you copy existing objects without making your code dependent on their classes.
* **Singleton:** Lets you ensure that a class has only one instance, while providing a global access point to this instance
* **Singleton pattern**

The purpose of the Singleton pattern is to ensure that there is only one instance of a class, and that there is a global access point to that object. The pattern ensures that the class is instantiated only once and that all requests are directed to that one and only object. Moreover, the object should not be created until it is actually needed. In the Singleton pattern, it is the class itself that is responsible for ensuring this constraint, not the clients of the class. (Judith Bishop. 2007)

The following is the basic structure of the singleton class in C#:



The above singleton class uses the static property to return an instance of the class. It has a private/protected parameter less constructor which will restrict the creation of an object using the new keyword. You must use the Instance property to get its object. You can make the constructor protected if you want to allow it to be inherited in a subclass.



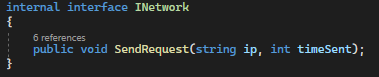
In the above example, object1 and object2 are the same instances. However, the above Singleton class is not thread-safe. It may give a wrong result in a multi-threaded application.

* **Factory Method Pattern**

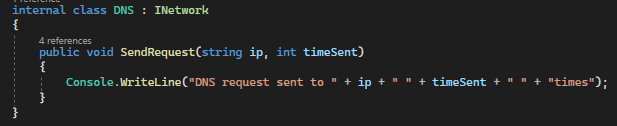
The Factory Method pattern is a way of creating objects, but letting subclasses decide exactly which class to instantiate. Various subclasses might implement the interface; the Factory Method instantiates the appropriate subclass based on information supplied by the client or extracted from the current state. (Judith Bishop. 2007)

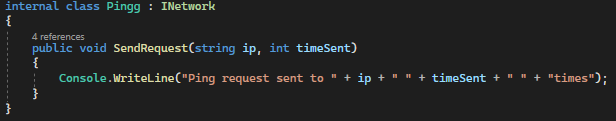
The following is the basic structure of the Factory Method Pattern in C#:

**Supper Class**

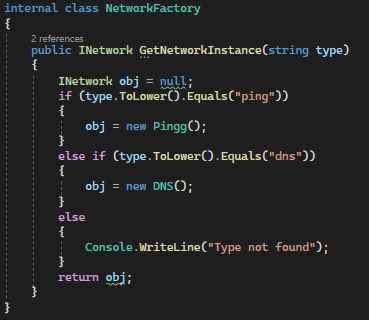


**Sub Classes**

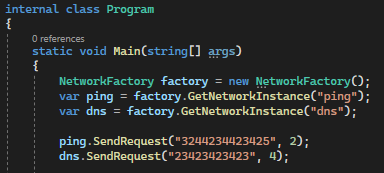




**Factory class**



**Main class**



In the example above, we need to use only one function which is GetNetworkInstance() with the keyword as the network utility name and you can send request to that specific network. When the system needs to add another network utility, it only needs to create a new class with the INetwork implement without affecting other classes.

**4.2 Structural pattern**

Structural patterns are a category of design patterns that focus on the composition of classes and objects to form larger structures or solve structural problems. These patterns help define relationships between objects, provide mechanisms for object composition, and simplify the design and organization of code.

Here are some commonly used structural design patterns:

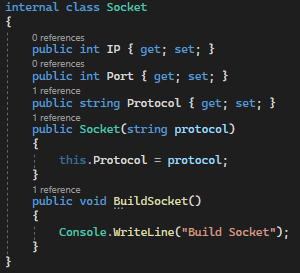
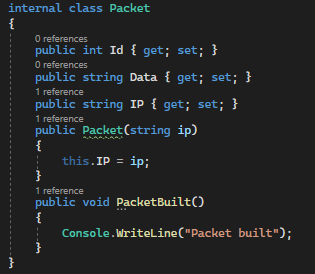
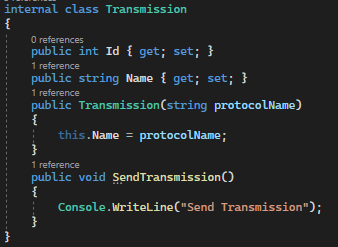
* **Adapter:** Allows objects with incompatible interfaces to work together by providing a common interface that both can use.
* **Bridge:** Decouples an abstraction from its implementation, allowing them to vary independently.
* **Composite:** Represents a hierarchical structure of objects as a tree, where individual objects and groups of objects are treated uniformly.
* **Decorator:** Dynamically adds new behavior to an object by wrapping it in an object of a decorator class.
* **Facade:** Provides a simplified interface to a complex subsystem, making it easier to use and reducing dependencies.
* **Flyweight:** Efficiently shares objects with similar characteristics to minimize memory usage.
* **Proxy:** Provides a surrogate or placeholder for another object to control its access or add additional functionality.
* **Facade pattern**

The role of the Façade pattern is to provide different high-level views of subsystems whose details are hidden from users. In general, the operations that might be desirable from a user’s perspective could be made up of different selections of parts of the subsystems. (Judith Bishop. 2007)

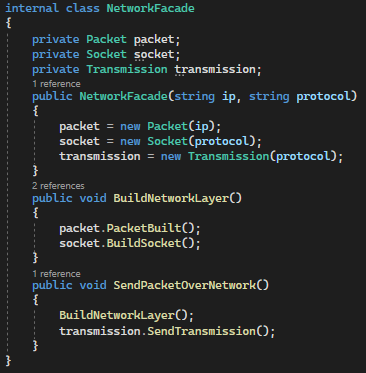
The structure of the Facade pattern typically includes the following components:

* **Facade:** This is the main component of the pattern and acts as a unified interface to the subsystem. It encapsulates the complexity of the subsystem and delegates the client requests to the appropriate objects within the subsystem. The Facade knows which subsystem classes to call and how to coordinate their actions.
* **Subsystem:** This refers to the set of classes that make up the complex subsystem. Each class within the subsystem has its own responsibilities and functionality.
* **Client:** The client code interacts with the Facade to perform operations on the subsystem. The client doesn't need to have knowledge of the subsystem's internal implementation details; it simply uses the simplified interface provided by the Facade.

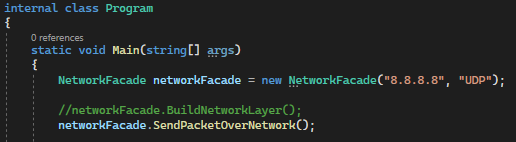
The following is the basic structure of the facade pattern in C#:

**Subsystem**

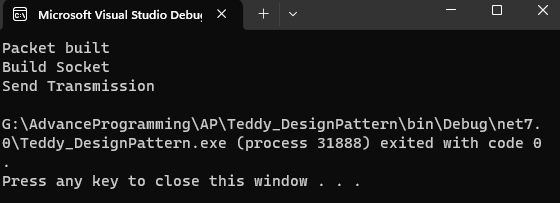
**Facade**



**Client**



**Output**

****

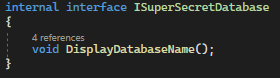
In the example above, even if the system has multiple functions, the client only needs to use one function SendPacketOverNetwork() to build a network layer and send packets over the network. Without Facade, the client will struggle to find the right function and run it in the correct order to achieve the objective. Furthermore, when a new function or service is introduced, it is much more difficult to implement and may have an impact on the client side.

* **Proxy Pattern**

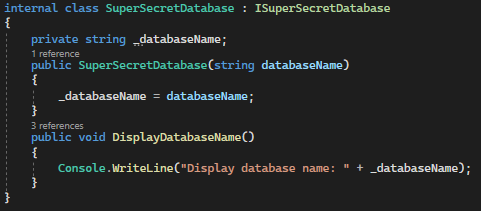
The Proxy pattern is a structural design pattern that provides a surrogate or placeholder for another object to control its access or add additional functionality. The Proxy pattern allows the client to interact with the proxy object instead of the real object, providing an extra level of indirection and encapsulation. It supports objects that control the creation of and access to other objects. The proxy is often a small object that stands in for a more complex object that is activated once certain circumstances are clear.

Here's an example of the Proxy pattern in C#:

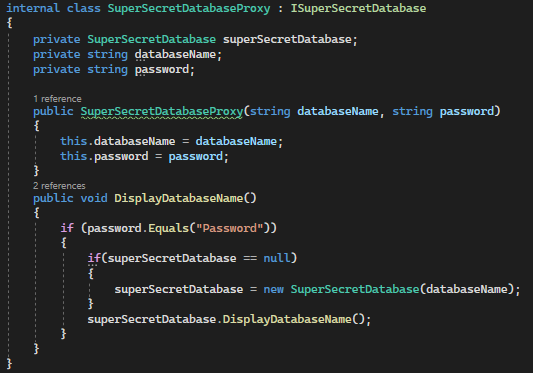
**Subject interface**



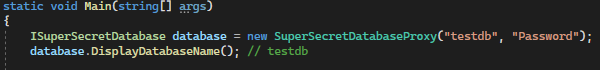
**Real Subject class**



**Proxy class**



**Client code**



In this example, we have an ISuperSecretDatabase interface that defines the common operations for displaying an image. The SuperSecretDatabase class is the real object that performs the actual display operations. The SuperSecretDatabaseProxy class acts as a proxy for the SuperSecretDatabase and controls its access.

When the client code interacts with the SuperSecretDatabaseProxy object and calls the DisplayDatabaseName() method, the proxy checks if the SuperSecretDatabase object has been created. If it hasn't been created yet, the proxy creates an instance of the SuperSecretDatabase and check the condition password. Then it delegates the display method call to it. Subsequent calls to display reuse the already created SuperSecretDatabase object.

**4.3 Behavioral pattern**

Behavioral patterns are concerned with algorithms and communication between them. The operations that make up a single algorithm might be split up between different classes, making a complex arrangement that is difficult to manage and maintain. The behavioral patterns capture ways of expressing the division of operations between classes and optimize how the communication should be handled. (Judith Bishop. 2007)

Here are some commonly used behavioral design patterns:

* **Observer:** Defines a one-to-many dependency between objects, so that when one object changes state, all its dependents are notified and updated automatically.
* **Strategy:** Encapsulates a family of algorithms and makes them interchangeable. It allows the algorithm to vary independently from the clients that use it.
* **Template Method:** Defines the skeleton of an algorithm in a base class, allowing subclasses to provide specific implementations for certain steps of the algorithm.
* **Command:** Encapsulates a request as an object, thereby allowing clients to parameterize clients with different requests, queue or log requests, and support undoable operations.
* **Iterator:** Provides a way to access elements of an aggregate object sequentially without exposing its underlying representation.
* State: Allows an object to change its behavior when its internal state changes, effectively encapsulating different behaviors within separate state objects.
* **Chain of Responsibility:** Establishes a chain of objects, where each object has a chance to handle a request and pass it along the chain if necessary.
* **Mediator:** Defines an object that encapsulates the communication and coordination between a set of objects, reducing direct dependencies between them.
* **Visitor:** Separates an algorithm from the objects on which it operates, allowing new operations to be added to the objects without modifying their structure

.

* **Chain of Responsibility pattern**

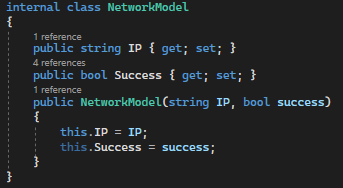
The Chain of Responsibility pattern works with a list of Handler objects that have limitations on the nature of the requests they can deal with. If an object cannot handle a request, it passes it on to the next object in the chain. At the end of the chain, there can be either default or exceptional behavior. (Judith Bishop. 2007)

The structure of the Chain of Responsibility pattern typically includes the following components:

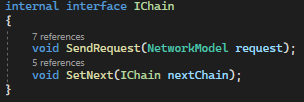
* **Handler:** This is an abstract class or interface that defines the common interface for handling requests. It contains a reference to the next handler in the chain, allowing the request to be passed along.
* **Concrete Handlers:** These are the subclasses or implementations of the Handler interface. Each Concrete Handler decides whether to handle the request or pass it on to the next handler in the chain.
* **Client:** The client code initiates the request and passes it to the first handler in the chain. It is unaware of the specific handlers in the chain and treats them uniformly through the Handler interface.

Here's a simplified example to illustrate the Chain of Responsibility pattern:

**NetworkModel class (request object)**

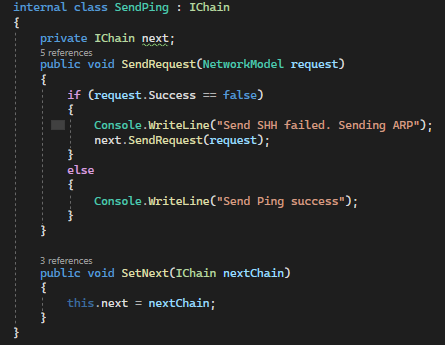


**Handler (abstract class or interface)**

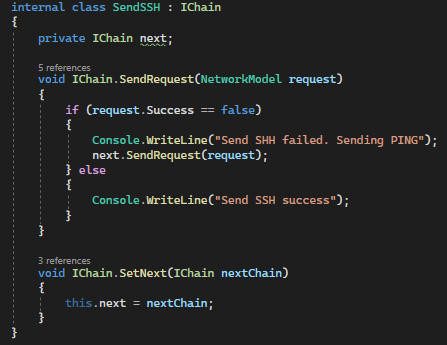


**Concrete Handlers**

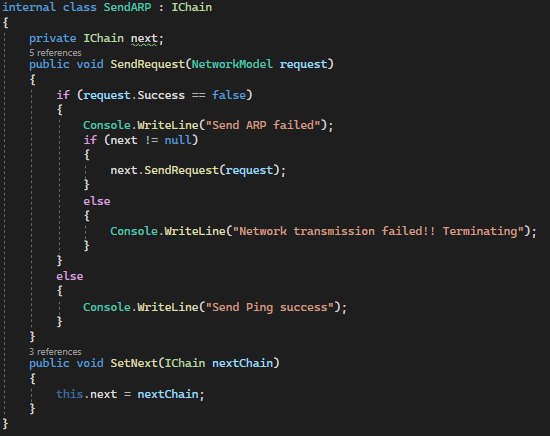
* Send Ping



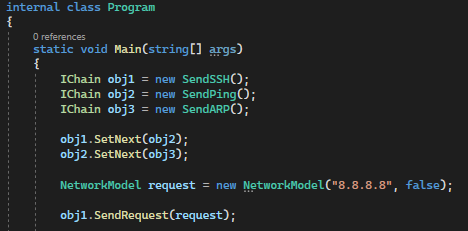
* Send SSH



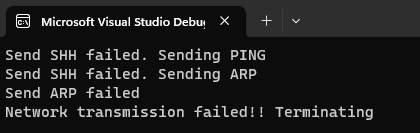
* Send ARP



**Client**



**Output**



In this example, we have three Concrete Handlers (**Ping**, **SSH**, and **ARP**) representing different levels of chain for sending request to the network. Each handler decides whether to approve the request based on a specific criterion (success == true) or passes it to the next handler in the chain.

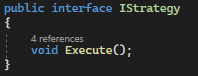
The client code creates the chain by setting the next chain for each handler. When a sending request is initiated, it is passed to the first handler in the chain (Ping). If the network transmission success, the request is approved. Otherwise, the request is passed to the next handler (SSH). This process continues until a handler approves the request or the end of the chain is reached.

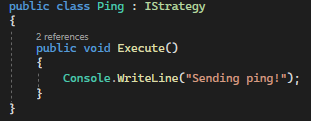
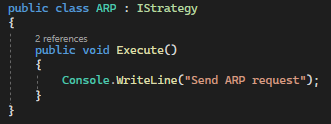
* **Strategy Pattern**

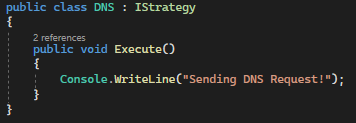
The Strategy pattern involves removing an algorithm from its host class and putting it in a separate class. There may be different algorithms (strategies) that are applicable for a given problem. If the algorithms are all kept in the host, messy code with lots of conditional statements will result. The Strategy pattern enables a client to choose which algorithm to use from a family of algorithms and gives it a simple way to access it. The algorithms can also be expressed independently of the data they are using. (Judith Bishop. 2007)

Example of the Strategy pattern in C#:

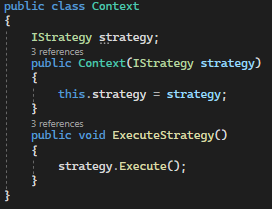
**Strategy interface**



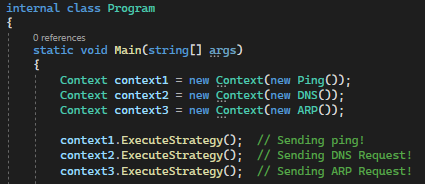
**Concrete strategies**



**Context class**



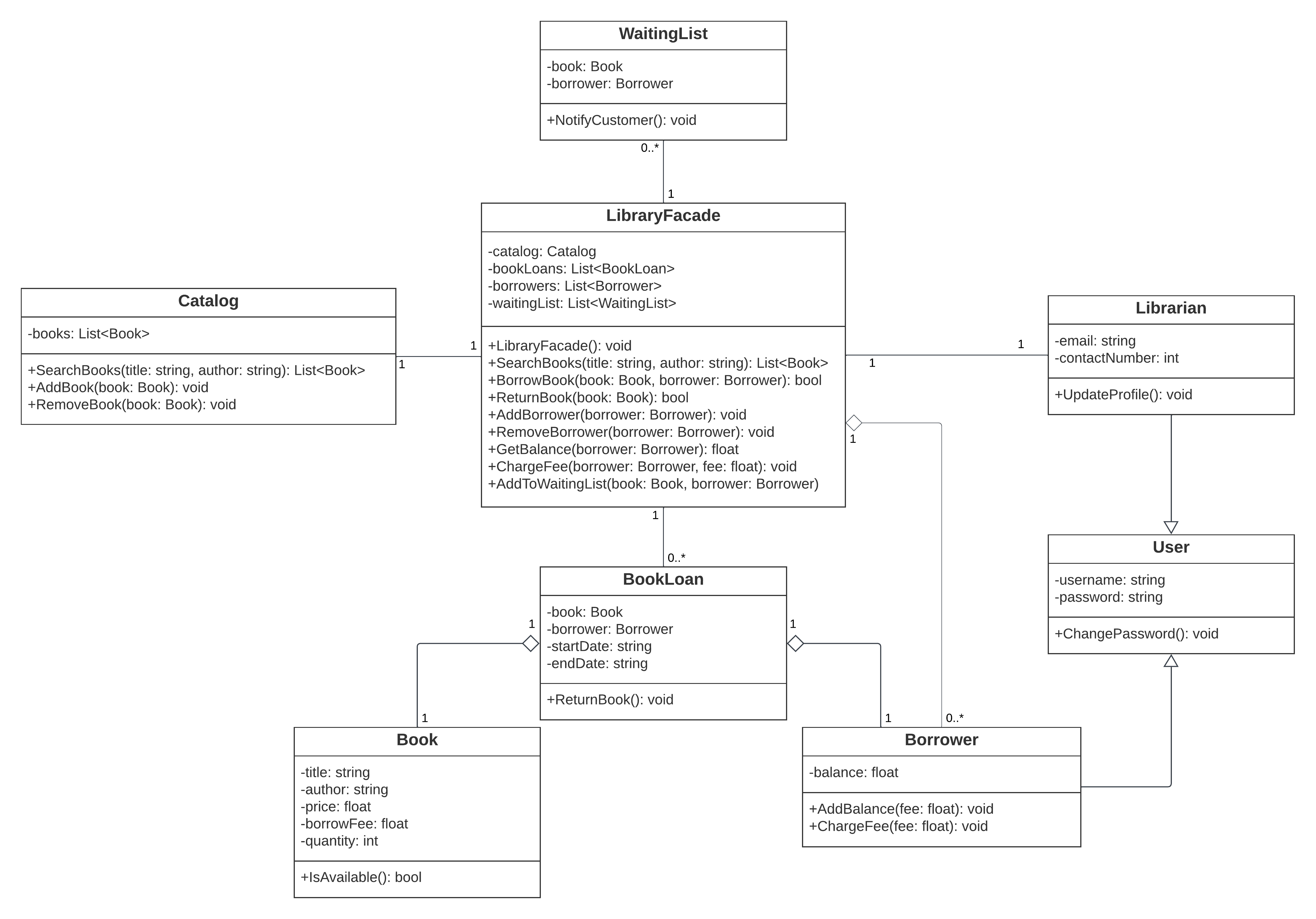
**Client**



In this example, we have an IStrategy interface that defines the common interface for all shipping strategies. Each concrete strategy (Ping, DNS, and ARP) implements this interface and provides its own implementation for sending request.

The Context class represents the context or the client code that uses the strategies. It has a reference to the current strategy and provides a method to execute the sending request based on the selected strategy.

In the client code, when we create a Context and set the initial strategy to Ping, we will send a ping request to the system. Similar concept can also apply to others strategy. Additionally, if we create a SetContext() method and dynamically change the strategy by calling it on the context, and the system can change sending request using the new strategy.

****

REFERENCE

Object-oriented programming (OOP). 2021. Alexander S. Gillis. Available at:

<https://www.techtarget.com/searchapparchitecture/definition/object-oriented-programming-OOP> [Accessed 20 January 2023].

What's a design pattern? 2021. Alexander Shvets Available at:

<https://refactoring.guru/design-patterns/what-is-pattern> [Accessed 20 January 2023].

C# 3.0 Design Patterns. 2007. Judith Bishop