Module "Design & architecture"

Submodule "Design patterns and architecture patterns"

Part 1

UA Resource Development Unit

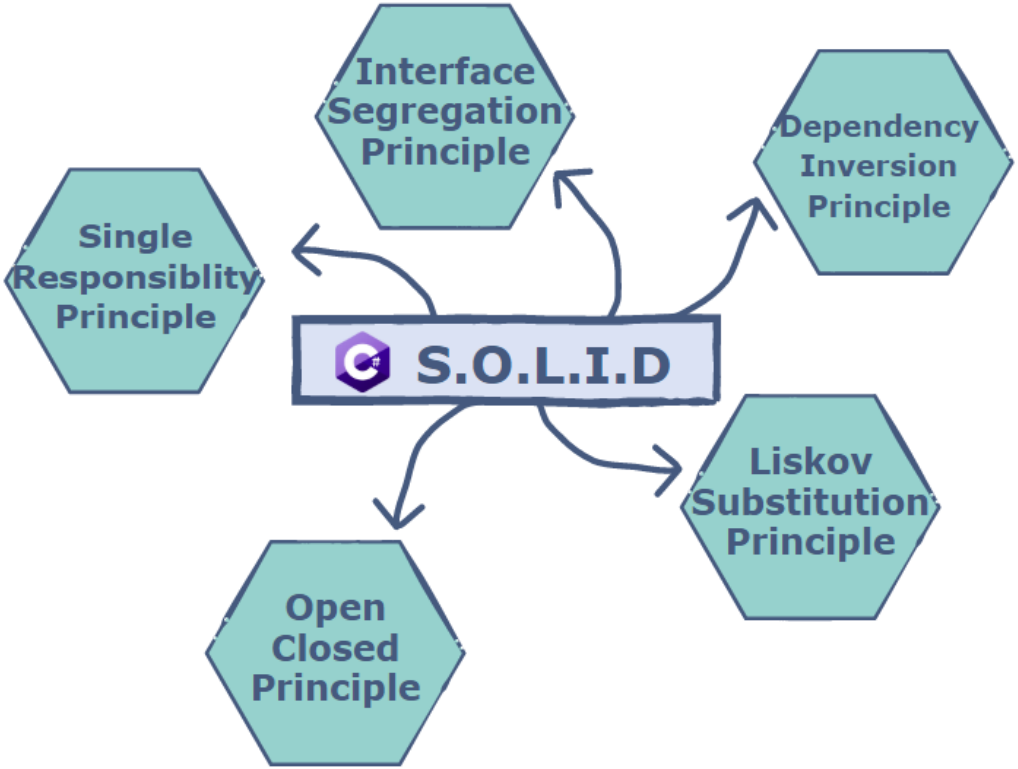
2021

AGENDA

1. S.O.L.I.D. principles
2. IoC
3. Composition over inheritance
4. GoF overview
5. Factory method vs Builder, Strategy vs Bridge, Decorator vs Adapter, Iterator, Observer

S.O.L.I.D.

S.O.L.I.D.



SOLID is an acronym for the five object-oriented design principles. These principles enable developers to develop software that is easily extendable and maintainable. SOLID principles also ensure that software can easily adapt to changing requirements.

S.O.L.I.D.

1. **S** stands for **SRP** (Single responsibility principle)
2. **O** stands for **OCP** (Open closed principle)
3. **L** stands for **LSP** (Liskov substitution principle)
4. **I** stands for **ISP** (Interface segregation principle)
5. **D** stands for **DIP** (Dependency inversion principle)

S.O.L.I.D.: S - SRP (SINGLE RESPONSIBILITY PRINCIPLE)

SRP says "Every software module should have only one reason to change".

The single-responsibility principle (SRP) is a computer-programming principle that states that every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class, module or function. All its services should be narrowly aligned with that responsibility.

S.O.L.I.D.: S - SRP (SINGLE RESPONSIBILITY PRINCIPLE)

S.O.L.I.D.: S - SRP (SINGLE RESPONSIBILITY PRINCIPLE)

S.O.L.I.D.: S - SRP (SINGLE RESPONSIBILITY PRINCIPLE)

```
class Customer
{
  void Add(Database db)
  {
      try
      {
         db.Add();
      }
      catch (Exception ex)
      {
        File.WriteAllText(@"C:\Error.txt", ex.ToString());
      }
  }
}
```

Bad Way

```
class Customer
   {
       private FileLogger logger = new FileLogger();
       void Add(Database db)
       {
           try
           {
               db.Add();
           }
           catch (Exception ex)
           {
               logger.Handle(ex.ToString());
           }
       }
   }
   class FileLogger
   {
       void Handle(string error)
       {
           File.WriteAllText(@"C:\Error.txt", error);
       }
   }
```

Good Way

S.O.L.I.D.: S - SRP (SINGLE RESPONSIBILITY PRINCIPLE)

```
public class Account
    {
        public string Number;
        public decimal CurrentBalance;
        public void Deposit(decimal amount) { ... }
        public void Withdraw(decimal amount) { ... }
        public void Transfer(decimal amount, Account recipient) { ... }
        public TaxTable CalculateTaxes(int year) { ... }
        public void GetByNumber(string number) { ... }
        public void Save() { ... }
    }
```

Bad Way

S.O.L.I.D.: S - SRP (SINGLE RESPONSIBILITY PRINCIPLE)

```
public class Account
    {
        public string Number;
        public decimal CurrentBalance;
        public void Deposit(decimal amount) { ... }
        public void Withdraw(decimal amount) { ... }
        public void Transfer(decimal amount, Account recipient) { ... }
    }

    public class AccountRepository
    {
        public Account GetByNumber(string number) { ... }
        public void Save(Account account) { ... }
    }

    public class TaxCalculator
    {
        public TaxTable CalculateTaxes(int year) { ... }
    }
```

Good Way

S.O.L.I.D.: O - OCP (Open closed principle)

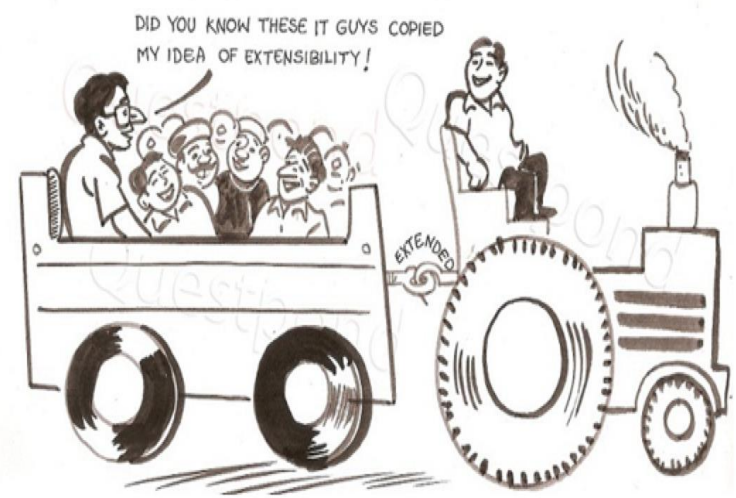
The open–closed principle states "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"; that is, such an entity can allow its behaviour to be extended without modifying its source code.

S.O.L.I.D.: O - OCP (Open closed principle)

```
class Customer
{
        private int _CustType;

        public int CustType
        {
            get { return _CustType; }
            set { _CustType = value; }
        }

        public double getDiscount(double TotalSales)
        {
                if (_CustType == 1)
                {
                    return TotalSales - 100;
                }
                else
                {
                    return TotalSales - 50;
                }
        }
}
```

Bad Way

S.O.L.I.D.: O - OCP (Open closed principle)

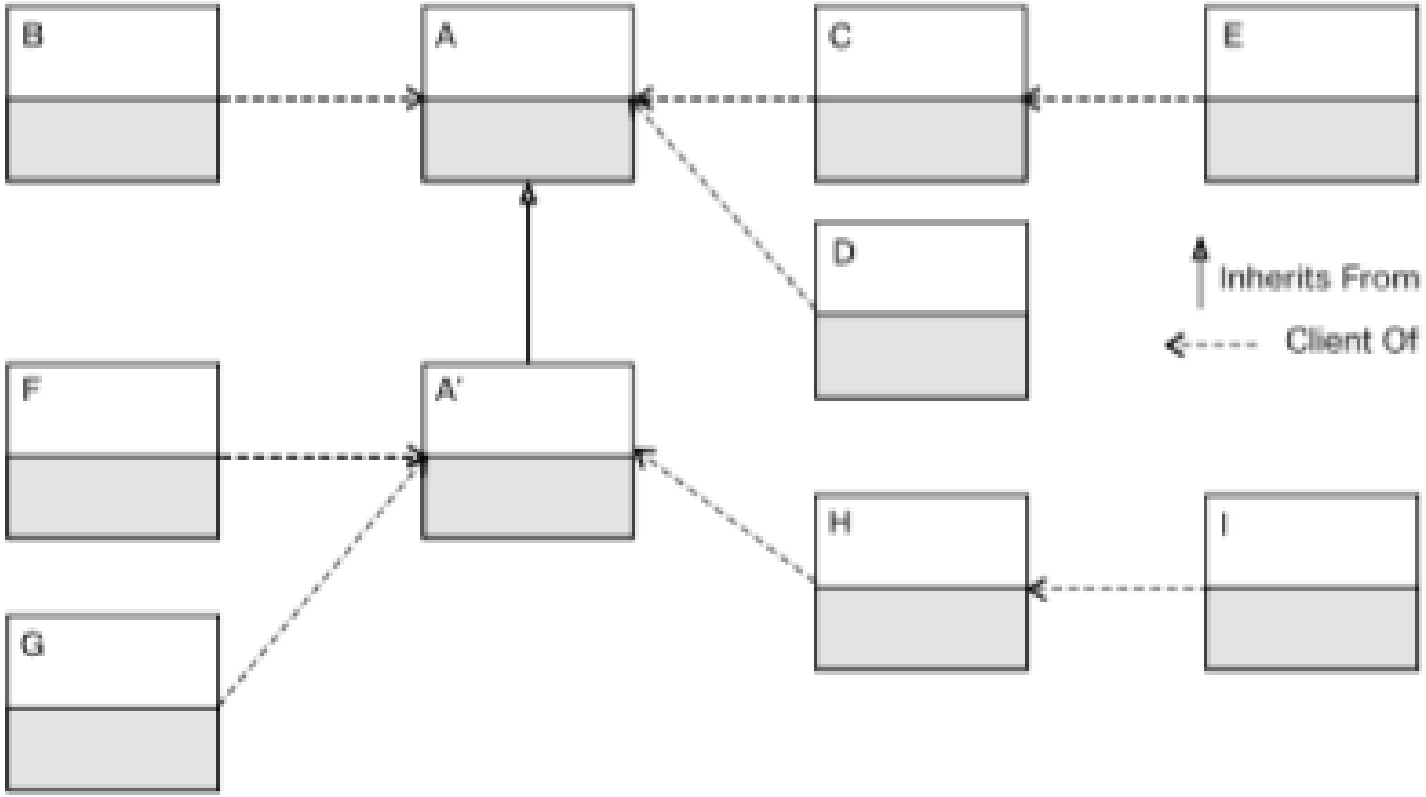


Good Way

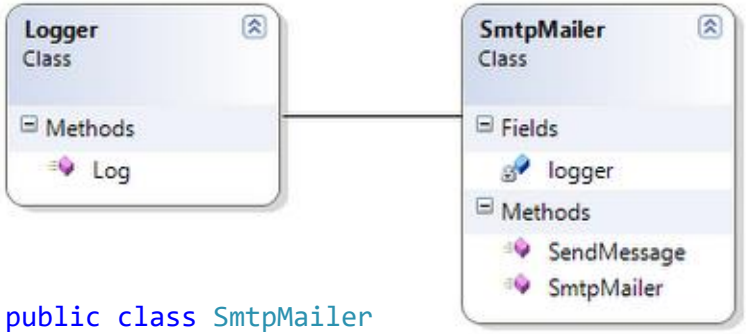
```
class Customer
{
        public virtual double getDiscount(double TotalSales)
        {
            return TotalSales;
        }
}

  class SilverCustomer : Customer
    {
        public override double getDiscount(double TotalSales)
        {
            return base.getDiscount(TotalSales) - 50;
        }
    }
```

```
class goldCustomer : SilverCustomer
    {
        public override double getDiscount(double TotalSales)
        {
            return base.getDiscount(TotalSales) - 100;
        }
    }
```

S.O.L.I.D.: O - OCP (Open closed principle)



S.O.L.I.D.: O - OCP (Open closed principle)

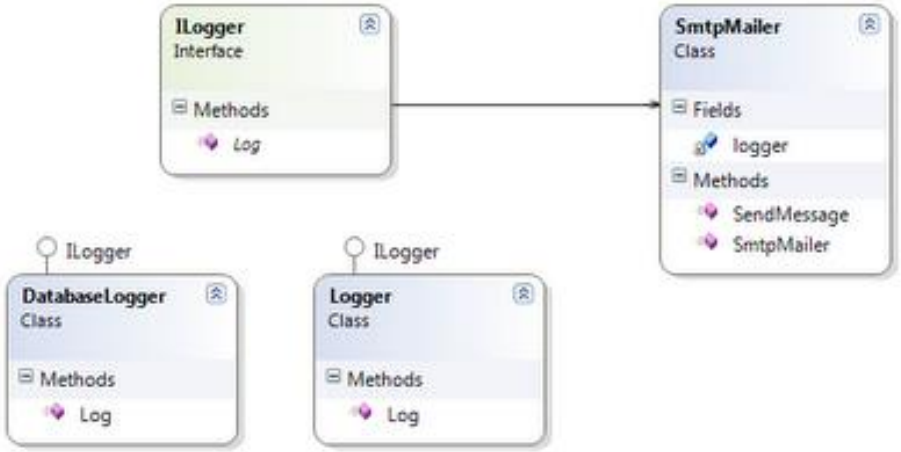



```
public class SmtpMailer
   {
       private readonly Logger logger;

       public SmtpMailer()
       {
           logger = new Logger();
       }

       public void SendMessage(string message)
       {
           // send message
           logger.Log(message);
       }
   }
```

Bad Way

```
public class DatabaseLogger
   {
       public void Log(string logText)
       {
           // save log to data base
       }
   }
   public class SmtpMailer
   {
       private readonly DatabaseLogger logger;

       public SmtpMailer()
       {
           logger = new DatabaseLogger();
       }
       public void SendMessage(string message)
       {
           // send message
           logger.Log(message);
       }
   }
```

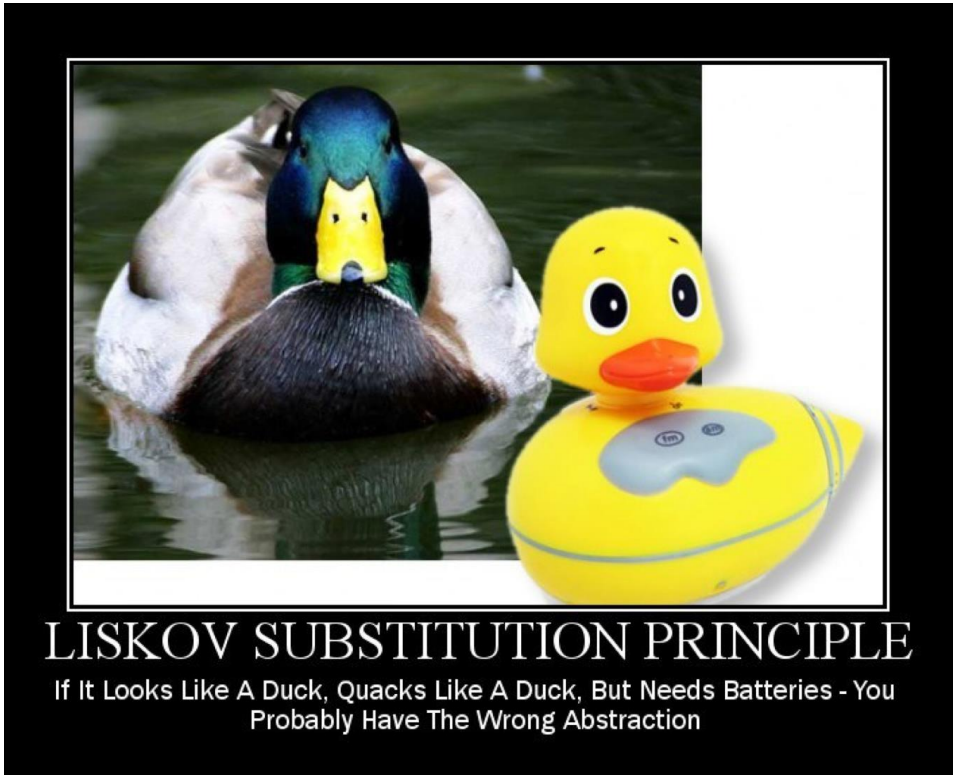
S.O.L.I.D.: O - OCP (Open closed principle)




```
public class SmtpMailer
    {
        private readonly ILogger logger;

        public SmtpMailer(ILogger logger)
        {
            this.logger = logger;
        }

        public void SendMessage(string message)
        {
            // send message
            logger.Log(message);
        }
    }
```

Good Way

S.O.L.I.D.: L- LSP (LISKOV SUBSTITUTION PRINCIPLE)



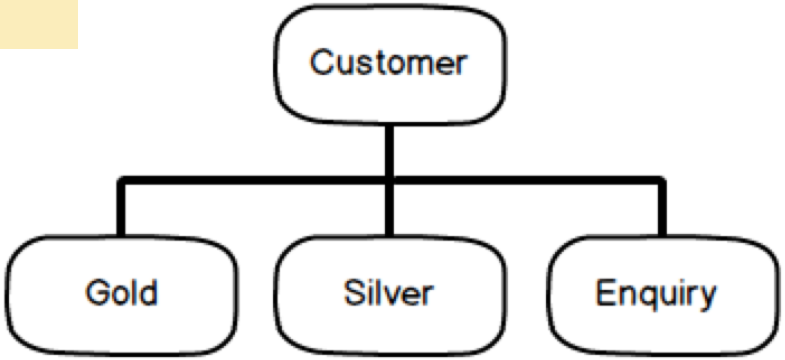
- Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.
- A parent class object should be able to refer child objects seamlessly during runtime polymorphism.

S.O.L.I.D.: L- LSP (LISKOV SUBSTITUTION PRINCIPLE)

"Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program."

S.O.L.I.D.: L- LSP (LISKOV SUBSTITUTION PRINCIPLE)

```
class Enquiry : Customer
    {
        public override double getDiscount(double TotalSales)
        {
            return base.getDiscount(TotalSales) - 5;
        }

        public override void Add()
        {
            throw new Exception("Not allowed");
        }
    }
```

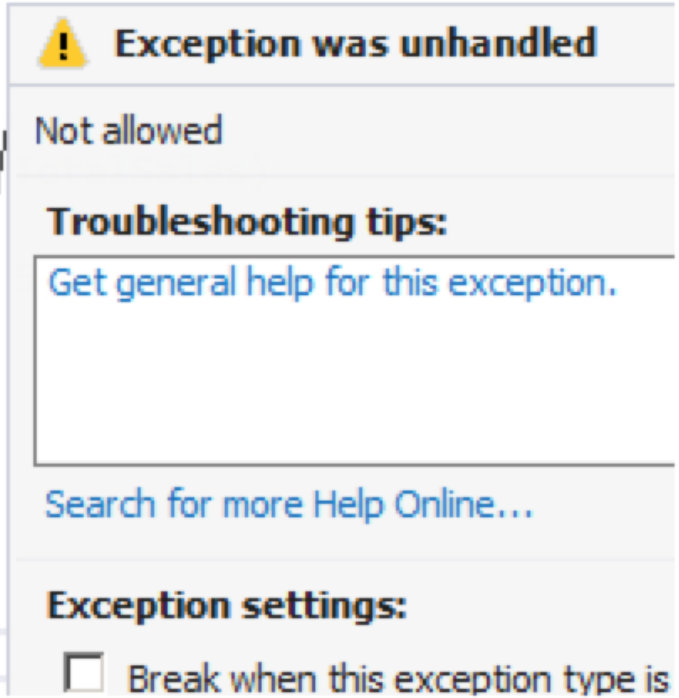


S.O.L.I.D.: L- LSP (LISKOV SUBSTITUTION PRINCIPLE)

```
List<Customer> Customers = new List<Customer>();
Customers.Add(new SilverCustomer());
Customers.Add(new goldCustomer());
Customers.Add(new Enquiry());

 foreach (Customer o in Customers)
 {
                o.Add();
 }
}
```

S.O.L.I.D.: L- LSP (LISKOV SUBSTITUTION PRINCIPLE)

```
class Enquiry : Customer
{
    public override double getDiscount(double
    {
        return base.getDiscount(TotalSales)
    }

    public override void Add()
    {
        throw new Exception("Not allowed");
    }
}
```



S.O.L.I.D.: L- LSP (LISKOV SUBSTITUTION PRINCIPLE)

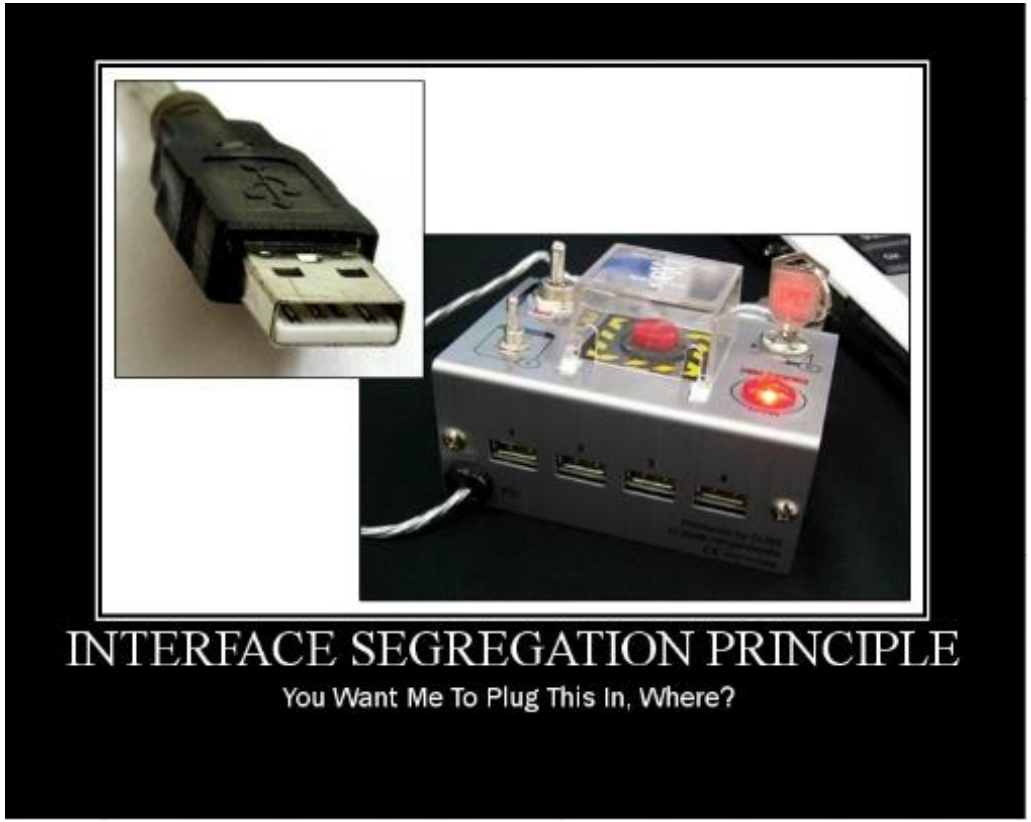
```
interface IDiscount
{
        double getDiscount(double TotalSales);
}


interface IDatabase
{
        void Add();
}
```

```
class Enquiry : IDiscount
    {
        public  double getDiscount(double TotalSales)
        {
            return TotalSales - 5;
        }
    }
```

S.O.L.I.D.: L- LSP (LISKOV SUBSTITUTION PRINCIPLE)

```
class Customer : IDiscount, IDatabase
   {


       private MyException obj = new MyException();
       public virtual void Add()
       {
           try
           {
               // Database code goes here
           }
           catch (Exception ex)
           {
               obj.Handle(ex.Message.ToString());
           }
       }

       public virtual double getDiscount(double TotalSales)
       {
           return TotalSales;
       }
   }
```

S.O.L.I.D.: I - ISP (INTERFACE SEGREGATION PRINCIPLE)

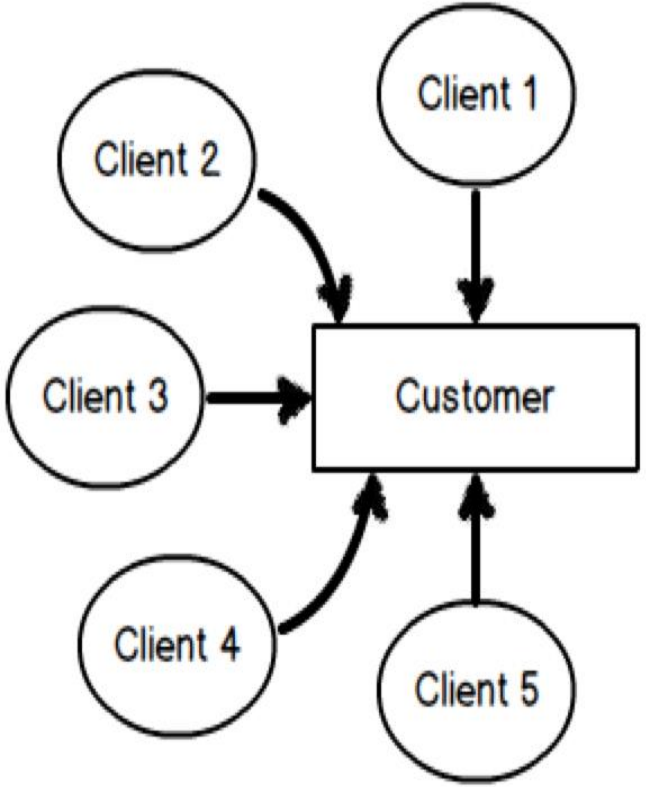


The interface-segregation principle (ISP) states that no client should be forced to depend on methods it does not use.

S.O.L.I.D.: I - ISP (INTERFACE SEGREGATION PRINCIPLE)

Some new clients come up with a demand saying that we also want a method which will help us to “Read” customer data.
So developers who are highly enthusiastic would like to change the “IDatabase” interfaceas
But by doing so we have done something terrible, can you guess ?

```
interface IDatabase
{
        void Add(); // old client are happy with these.
voidRead(); // Added for new clients.
}
```



S.O.L.I.D.: I - ISP (INTERFACE SEGREGATION PRINCIPLE)

```
interface IDatabaseV1 : IDatabase // Gets the Add method
{
Void Read();
}
```

```
class CustomerwithRead : IDatabase, IDatabaseV1
    {

public void Add()
{
    Customer obj = new Customer();
Obj.Add();
}
    Public void Read()
    {
    // Implements  logic for read
}
    }
```

```
IDatabase i = new Customer(); // 1000 happy old clients not touched
i.Add();

IDatabaseV1 iv1 = new CustomerWithread(); // new clients
Iv1.Read();
```

S.O.L.I.D.: D stands for DIP (Dependency inversion principle)

The dependency inversion principle is a specific form of decoupling software modules. When following this principle, the conventional dependency relationships established from high-level, policy-setting modules to low-level, dependency modules are reversed, thus rendering high-level modules independent of the low-level module implementation details. The principle states:

High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g. interfaces).

- Abstractions should not depend on details.
- Details (concrete implementations) should depend on abstractions.

S.O.L.I.D.: D stands for DIP (Dependency inversion principle)

```
class Customer
    {
        private FileLogger obj = new FileLogger();
        publicvirtual void Add()
        {
            try
            {
                // Database code goes here
            }
            catch (Exception ex)
            {
                obj.Handle(ex.ToString());
            }
        }
    }
```

```
interface ILogger
{
        void Handle(string error);
}
```

S.O.L.I.D.: D stands for DIP (Dependency inversion principle)

```
class FileLogger : ILogger
    {
        public void Handle(string error)
        {
            System.IO.File.WriteAllText(@"c:\Error.txt", error);
        }
    }
```

```
class EverViewerLogger : ILogger
    {
        public void Handle(string error)
        {
            // log errors to event viewer
        }
    }
```

```
class EmailLogger : ILogger
  {
      public void Handle(string error)
      {
          // send errors in email
      }
  }
```

S.O.L.I.D.: D stands for DIP (Dependency inversion principle)

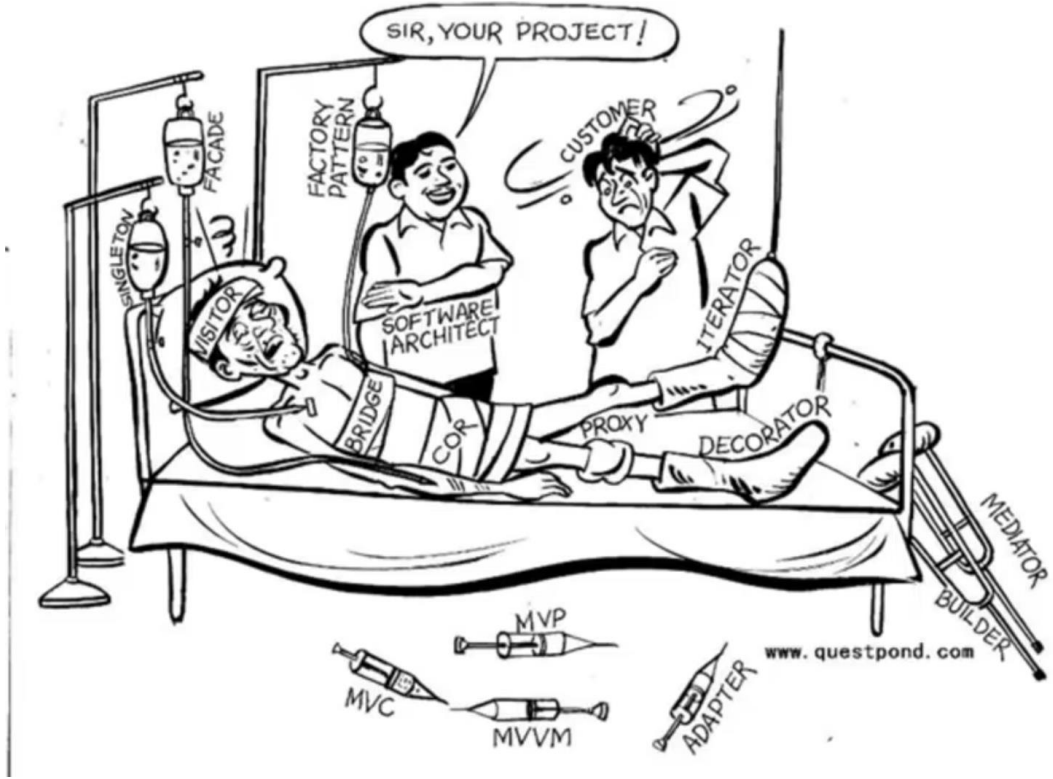
```
class Customer : IDiscount, IDatabase
    {
        private IException obj;

public virtual void Add(int Exhandle)
        {
            try
            {
                // Database code goes here
            }
            catch (Exception ex)
            {
                if (Exhandle == 1)
                {
                    obj = new MyException();
                }
                else
                {
                    obj = new EmailException();
                }
                obj.Handle(ex.Message.ToString());
            }
        }
```

S.O.L.I.D.: D stands for DIP (Dependency inversion principle)

```
class Customer : IDiscount, IDatabase
 {
        private Ilogger obj;
        public Customer(ILogger i)
        {
            obj = i;
        }
}
```

```
IDatabase i = new Customer(new EmailLogger());
```

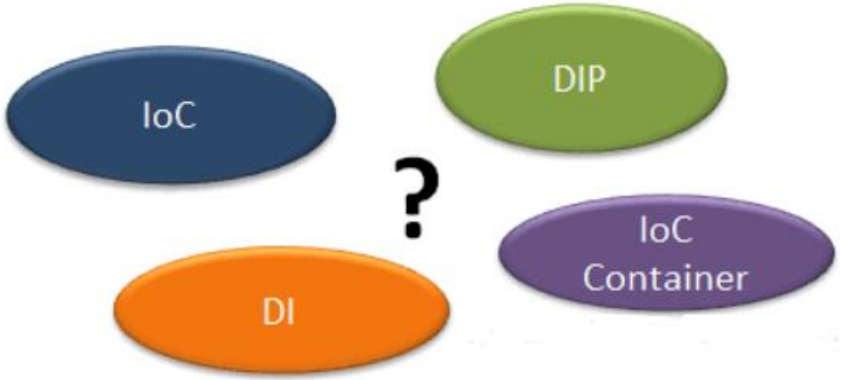
S.O.L.I.D.


SIR, YOUR PROJECT!
FACADE
FACTORY PATTERN
CUSTOMER
SINGLETON
VISITOR
SOFTWARE ARCHITECT
ITERATOR
BRIDGE
COR
PROXY
DECORATOR
MEDIATOR
BUILDER
MVP
MVC
MVVM
ADAPTER
www.questpond.com

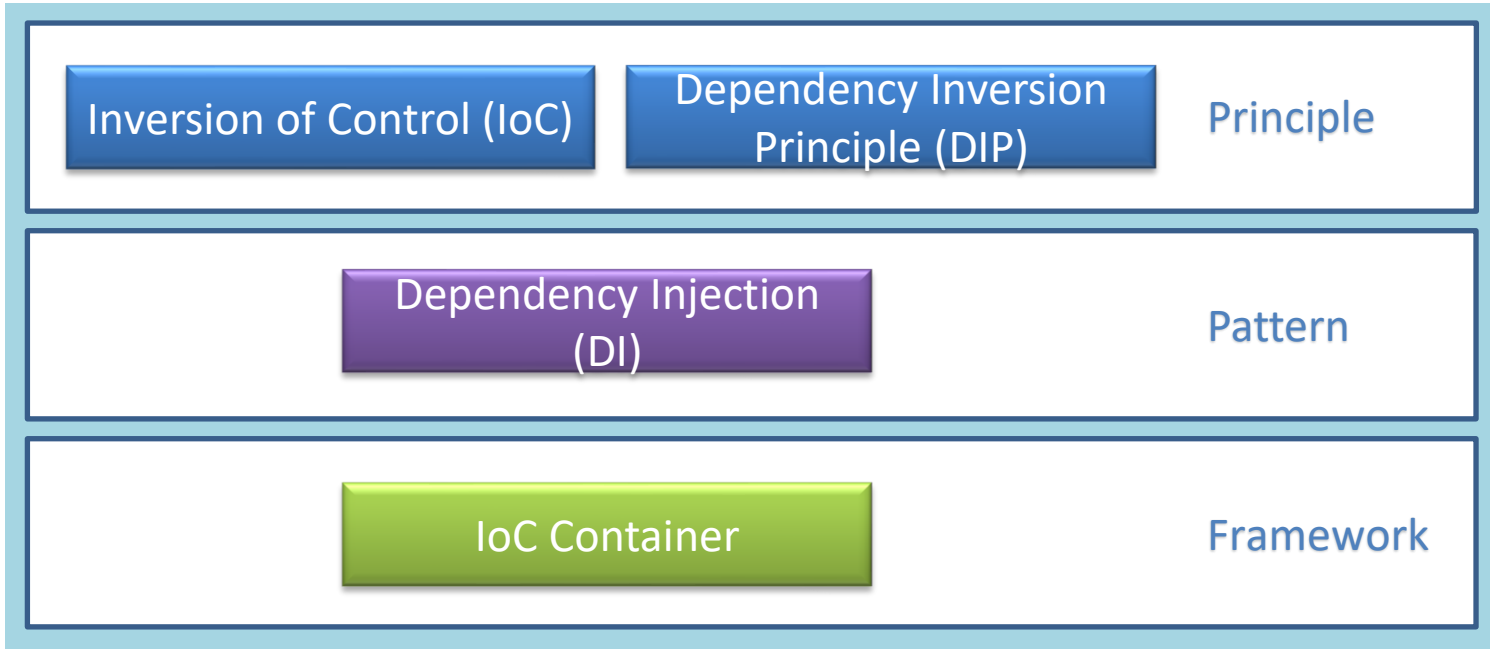
IoC

IoC: Introduction

The terms Inversion of Control (IoC), Dependency Inversion Principle (DIP), Dependency Injection (DI), and IoC containers may be familiar. But are you clear about what each term means?



IoC: Introduction

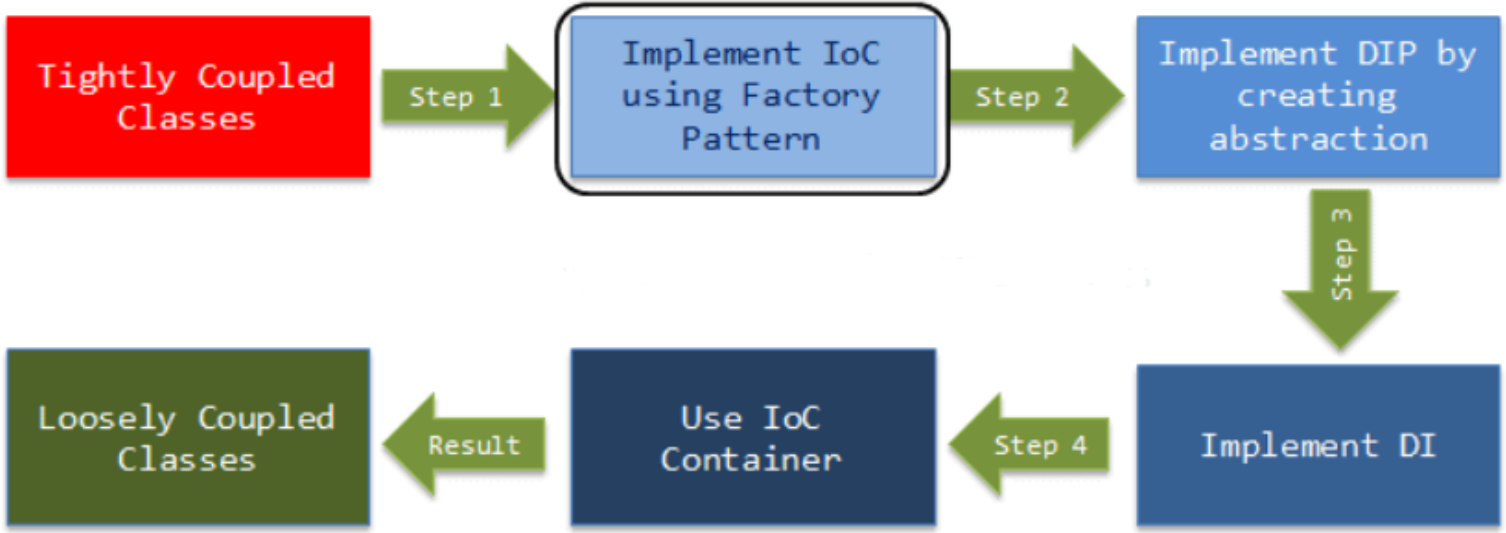


IoC: Inversion of Control

IoC is a design principle which recommends the inversion of different kinds of controls in object-oriented design to achieve loose coupling between application classes. In this case, control refers to any additional responsibilities a class has, other than its main responsibility, such as control over the flow of an application, or control over the dependent object creation and binding (Remember SRP - Single Responsibility Principle).

If you want to do TDD (Test Driven Development), then you must use the IoC principle, without which TDD is not possible

IoC


Tightly Coupled Classes
Step 1
Implement IoC using Factory Pattern
Step 2
Implement DIP by creating abstraction
Step 3
Implement DI
Step 4
Use IoC Container
Result
Loosely Coupled Classes

IoC

IoC is all about inverting the control

The IoC principle helps in designing loosely coupled classes which make them testable, maintainable and extensible.

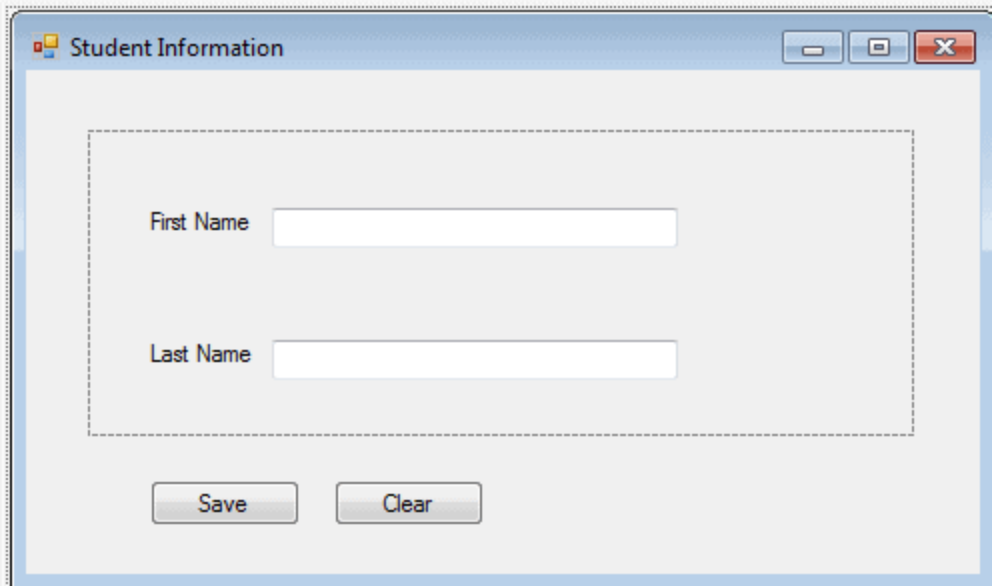
IoC: Different kinds of control.

```
namespace FlowControlDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            bool continueExecution = true;
            do
            {
                Console.Write("Enter First Name:");
                var firstName = Console.ReadLine();
                Console.Write("Enter Last Name:");

                var lastName = Console.ReadLine();
                Console.Write("Do you want to save it? Y/N: ");

                var wantToSave = Console.ReadLine();

                if (wantToSave.ToUpper() == "Y")
                        SaveToDB(firstName, lastName);
                Console.Write("Do you want to exit? Y/N: ");

                var wantToExit = Console.ReadLine();
                    if (wantToExit.ToUpper() == "Y") continueExecution = false;
                } while (continueExecution);
            }
            private static void SaveToDB(string firstName, string lastName)
            {
                //save firstName and lastName to the database here..
            }
        }
    }
```

➢ Control Over the Flow of a Program

In a typical console application in C#, execution starts from the Main() function. The Main() function controls the flow of a program or, in other words, the sequence of user interaction.

IoC: Different kinds of control.



IoC: Different kinds of control.

- **Control Over the Dependent Object Creation**

```
public class A
{
   B b;

   public A()
   {
      b = new B();
   }

   public void Task1()
   {
      // do something here..
      b.SomeMethod();
      // do something here..
   }
}
```

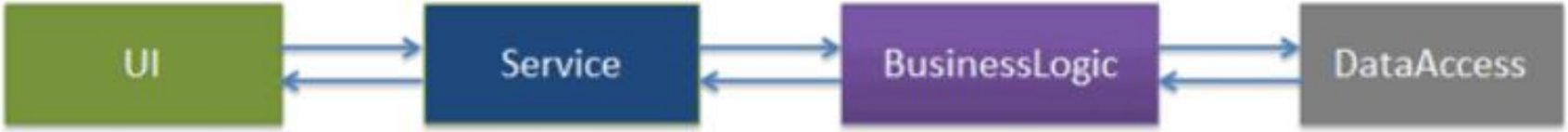
```
public class B
{
   public void SomeMethod()
   {
      //doing something..
   }
}
```

Class A cannot complete its task without class B and so you can say that "Class A is dependent on class B" or "class B is a dependency of class A".

IoC: Different kinds of control.

```
public class A
{
   B b;
   public A()
   {
      b = Factory.GetObjectOfB ();
   }
   public void Task1()
   {
      // do something here..
      b.SomeMethod();
      // do something here..
   }
}

public class Factory
{
   public static B GetObjectOfB()
   {
      return new B();
   }
}
```

IoC



IoC

```
public class CustomerBusinessLogic
{
   DataAccess _dataAccess;

   public CustomerBusinessLogic()
   {
      _dataAccess = new DataAccess();
   }

   public string GetCustomerName(int id)
   {
      return _dataAccess.GetCustomerName(id);
   }
}
```

```
public class DataAccess
{
   public DataAccess()
   {
   }

   public string GetCustomerName(int id)
   {
      return "Dummy Customer Name";
      // get it from DB in real app
   }
}
```

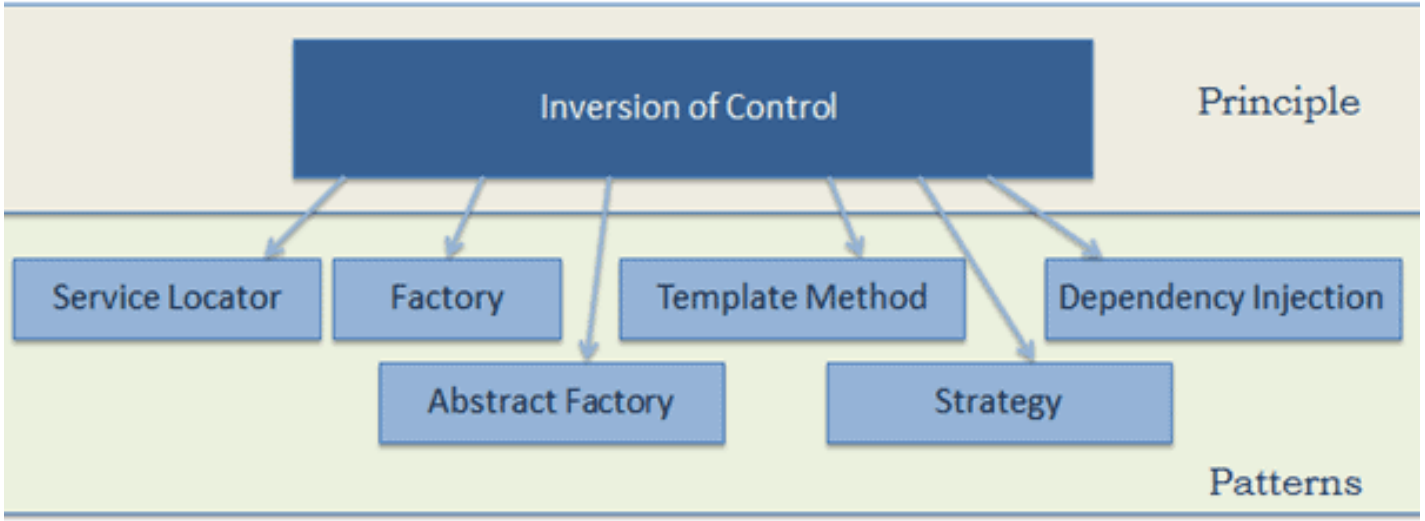
IoC

Problems in the above example classes:

1. CustomerBusinessLogic and DataAccess classes are tightly coupled classes. So, changes in the DataAccess class will lead to changes in the CustomerBusinessLogic class. For example, if we add, remove or rename any method in the DataAccess class then we need to change the CustomerBusinessLogic class accordingly.
2. Suppose the customer data comes from different databases or web services and, in the future, we may need to create different classes, so this will lead to changes in the CustomerBusinessLogic class.
3. The CustomerBusinessLogic class creates an object of the DataAccess class using the new keyword. There may be multiple classes which use the DataAccess class and create its objects. So, if you change the name of the class, then you need to find all the places in your source code where you created objects of DataAccess and make the changes throughout the code. This is repetitive code for creating objects of the same class and maintaining their dependencies.
4. Because the CustomerBusinessLogic class creates an object of the concrete DataAccess class, it cannot be tested independently (TDD). The DataAccess class cannot be replaced with a mock class.

IoC

The following pattern (but not limited) implements the IoC principle.



IoC is a principle, not a pattern!

IoC

DataAccess Factory:

```
public class DataAccessFactory
{
   public static DataAccess GetDataAccessObj()
   {
      return new DataAccess();
   }
}
```

Use Factory Class to Retrieve Object

```
public class CustomerBusinessLogic
{
   public CustomerBusinessLogic()
   {
   }
   public string GetCustomerName(int id)
   {
      DataAccess _dataAccess = DataAccessFactory.GetDataAccessObj();
      return _dataAccess.GetCustomerName(id);
   }
}
```

Dependency Inversion Principle

```
public class CustomerBusinessLogic
{
   public CustomerBusinessLogic()
   {
   }
   public string GetCustomerName(int id)
   {
      DataAccess _dataAccess = DataAccessFactory.GetDataAccessObj();
      return _dataAccess.GetCustomerName(id);
   }
}
```

```
 public class DataAccessFactory
 {
    public static DataAccess GetDataAccessObj()
    {
       return new DataAccess();
    }
 }
```

```
public class DataAccess
{
   public DataAccess()
   {
   }

   public string GetCustomerName(int id)
   {
      return "Dummy Customer Name";
      // get it from DB in real app
   }
}
```

Dependency Inversion Principle

```
public interface ICustomerDataAccess
{
   string GetCustomerName(int id);
}
```

```
public class CustomerDataAccess: ICustomerDataAccess
{
   public CustomerDataAccess()
   {
   }

   public string GetCustomerName(int id)
   {
      return "Dummy Customer Name";
   }
}
```

```
public class DataAccessFactory
{
   public static ICustomerDataAccess GetCustomerDataAccessObj()
   {
      return new CustomerDataAccess();
   }
}
```

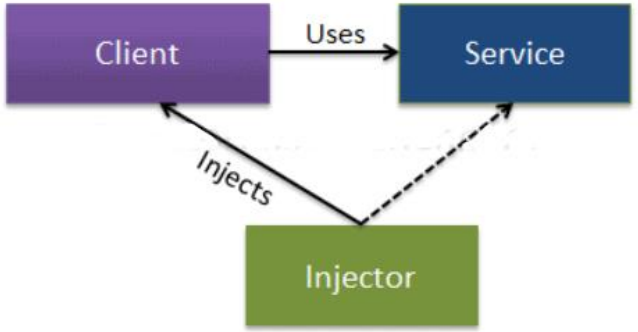
```
public class CustomerBusinessLogic
{
   ICustomerDataAccess _custDataAccess;
   public CustomerBusinessLogic()
   {
      _custDataAccess = DataAccessFactory.GetCustomerDataAccessObj();
   }
   public string GetCustomerName(int id)
   {
      return _custDataAccess.GetCustomerName(id);
   }
}
```

Dependency Injection (DI)

Dependency Injection (DI) is a design pattern used to implement IoC. It allows the creation of dependent objects outside of a class and provides those objects to a class through different ways. Using DI, we move the creation and binding of the dependent objects outside of the class that depends on them.

The Dependency Injection pattern involves 3 types of classes.

1. **Client Class**: The client class (dependent class) is a class which depends on the service class
2. **Service Class**: The service class (dependency) is a class that provides service to the client class.
3. **Injector Class**: The injector class injects the service class object into the client class.



Dependency Injection

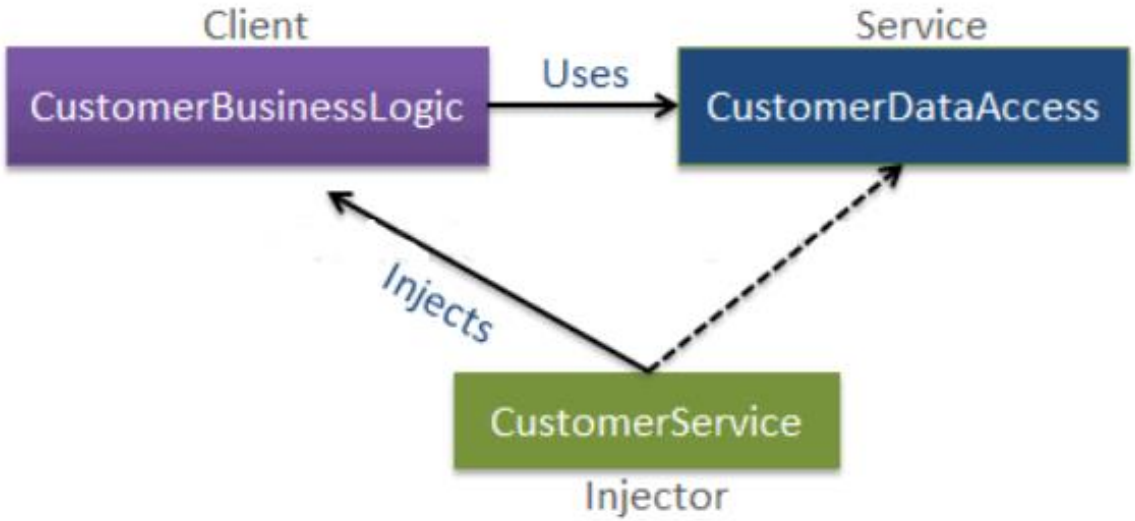
Types of Dependency Injection

Constructor Injection: In the constructor injection, the injector supplies the service (dependency) through the client class constructor.

Property Injection: In the property injection (aka the Setter Injection), the injector supplies the dependency through a public property of the client class.

Method Injection: In this type of injection, the client class implements an interface which declares the method(s) to supply the dependency and the injector uses this interface to supply the dependency to the client class.

Dependency Injection (DI)



Dependency Injection (DI): Constructor Injection

```
public class CustomerBusinessLogic
{
   ICustomerDataAccess _dataAccess;

   public CustomerBusinessLogic(ICustomerDataAccess custDataAccess)
   {
      _dataAccess = custDataAccess;
   }

   public CustomerBusinessLogic()
   {
      _dataAccess = new CustomerDataAccess();
   }

   public string ProcessCustomerData(int id)
   {
      return _dataAccess.GetCustomerName(id);
   }
}
```

```
public interface ICustomerDataAccess
{
   string GetCustomerData(int id);
}

public class CustomerDataAccess: ICustomerDataAccess
{
   public CustomerDataAccess()
   {
   }
   public string GetCustomerName(int id)
   {
      //get the customer name from
      //the db in real application
      return "Dummy Customer Name";
   }
}
```

Dependency Injection (DI): Constructor Injection

```
public class CustomerService
{
   CustomerBusinessLogic _customerBL;

   public CustomerService()
   {
      _customerBL = new CustomerBusinessLogic(new CustomerDataAccess());
   }

   public string GetCustomerName(int id)
   {
      return _customerBL.GetCustomerName(id);
   }
}
```

Dependency Injection (DI): Property Injection

```
public class CustomerBusinessLogic
{
   public CustomerBusinessLogic()
   {
   }
   public string GetCustomerName(int id)
   {
      return DataAccess.GetCustomerName(id);
   }
   public ICustomerDataAccess DataAccess { get; set; }
}
```

```
public class CustomerService
{
   CustomerBusinessLogic _customerBL;
   public CustomerService()
   {
      _customerBL = new CustomerBusinessLogic();
      _customerBL.DataAccess = new CustomerDataAccess();
   }
   public string GetCustomerName(int id)
   {
      return _customerBL.GetCustomerName(id);
   }
}
```

Dependency Injection (DI): Method Injection

```
interface IDataAccessDependency
{
    void SetDependency(ICustomerDataAccess customerDataAccess);
}

public class CustomerBusinessLogic : IDataAccessDependency
{
    ICustomerDataAccess _dataAccess;
    public CustomerBusinessLogic()
    {
    }
    public string GetCustomerName(int id)
    {
        return _dataAccess.GetCustomerName(id);
    }
    public void SetDependency(ICustomerDataAccess customerDataAccess)
    {
        _dataAccess = customerDataAccess;
    }
}
```

```
public class CustomerService
{
    CustomerBusinessLogic _customerBL;
    public CustomerService()
    {
        _customerBL = new CustomerBusinessLogic();
        ((IDataAccessDependency)_customerBL).
                SetDependency(new CustomerDataAccess());
    }
    public string GetCustomerName(int id)
    {
        return _customerBL.GetCustomerName(id);
    }
}
```

IoC Container

IoC Container (a.k.a. DI Container) is a framework for implementing automatic dependency injection. It manages object creation and it's life-time, and also injects dependencies to the class.

The IoC container creates an object of the specified class and also injects all the dependency objects through a constructor, a property or a method at run time and disposes it at the appropriate time. This is done so that we don't have to create and manage objects manually.

IoC Container

All the containers must provide easy support for the following DI lifecycle.

- **Register**: The container must know which dependency to instantiate when it encounters a particular type. This process is called registration. Basically, it must include some way to register type-mapping.

- **Resolve**: When using the IoC container, we don't need to create objects manually. The container does it for us. This is called resolution. The container must include some methods to resolve the specified type; the container creates an object of the specified type, injects the required dependencies if any and returns the object.

- **Dispose**: The container must manage the lifetime of the dependent objects. Most IoC containers include different lifetimemanagers to manage an object's lifecycle and dispose it.

IoC Container

There are many open source or commercial containers available for .NET. Some are listed below.

- Unity https://github.com/unitycontainer/unity
- StructureMap https://structuremap.github.io/
- Castle Windsor http://www.castleproject.org/
- Ninject http://www.ninject.org/
- Autofac https://autofac.org/
- DryIoc https://bitbucket.org/dadhi/dryioc
- Simple Injector https://simpleinjector.org/index.html
- Light https://github.com/seesharper/LightInject

Composition over inheritance

Composition over inheritance

Composition over inheritance (or composite reuse principle, CRP) in object-oriented programming (OOP) is the principle that classes should achieve polymorphic behavior and code reuse by their composition (by containing instances of other classes that implement the desired functionality) rather than inheritance from a base or parent class. This is an often-stated principle of OOP, such as in the influential book Design Patterns.

Composition over inheritance

```
class Program
{
    static void Main()
    {
        var player = new GameObject(new Visible(), new Movable(), new Solid());
        player.Update(); player.Collide(); player.Draw();

        var cloud = new GameObject(new Visible(), new Movable(), new NotSolid());
        cloud.Update(); cloud.Collide(); cloud.Draw();

        var building = new GameObject(new Visible(), new NotMovable(), new Solid());
        building.Update(); building.Collide(); building.Draw();

        var trap = new GameObject(new Invisible(), new NotMovable(), new Solid());
        trap.Update(); trap.Collide(); trap.Draw();
    }
}
```

```
interface IVisible
{
     void Draw();
}
class Invisible : IVisible
{
    public void Draw()
    {
        Console.Write("I won't appear.");
    }
}
class Visible : IVisible
{
    public void Draw()
    {
        Console.Write("I'm showing myself.");
    }
}
```

Composition over inheritance

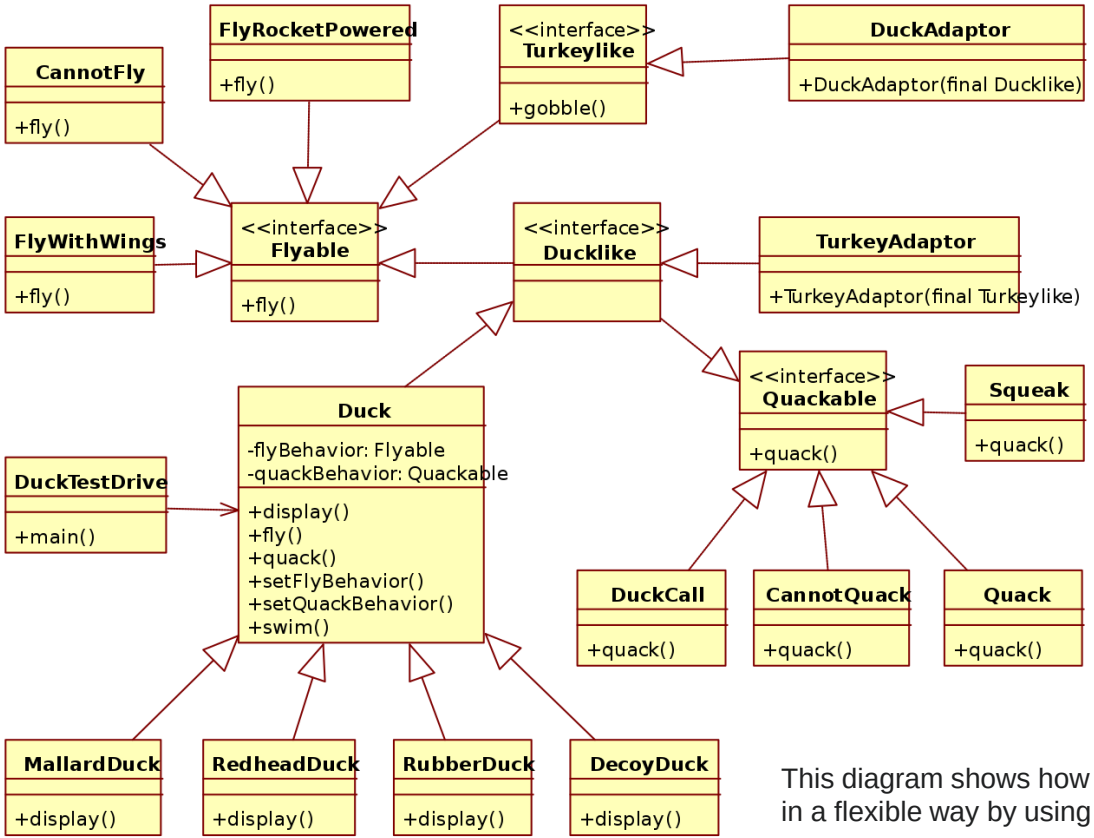
```
interface ICollidable
{
    void Collide();
}
class Solid : ICollidable
{
    public void Collide()
    {
        Console.Write("Bang!");
    }
}
class NotSolid : ICollidable
{
    public void Collide()
    {
        Console.Write("Splash!");
    }
}
```

```
interface IUpdatable
{
    void Update();
}
class Movable : IUpdatable
{
    public void Update()
    {
        Console.Write("Moving forward.");
    }
}
class NotMovable : IUpdatable
{
    public void Update()
    {
        Console.Write("I'm staying put.");
    }
}
```

Composition over inheritance

```
class GameObject : IVisible, IUpdatable, ICollidable
{
    private readonly IVisible _visible;
    private readonly IUpdatable _updatable;
    private readonly ICollidable _collidable;
    public GameObject(IVisible visible, IUpdatable updatable, ICollidable collidable)
    {
        _visible = visible;
        _updatable = updatable;
        _collidable = collidable;
    }
    public void Update()
    {
        _updatable.Update();
    }
    public void Draw()
    {
        _visible.Draw();
    }
    public void Collide()
    {
        _collidable.Collide();
    }
}
```

Composition over inheritance



This diagram shows how the fly and sound behavior of an animal can be designed in a flexible way by using the composition over inheritance design principle.

Composition over inheritance: Benefits

To favor composition over inheritance is a design principle that gives the design higher flexibility. It is more natural to build business-domain classes out of various components than trying to find commonality between them and creating a family tree.

Composition also provides a more stable business domain in the long term as it is less prone to the quirks of the family members. In other words, it is better to compose what an object can do (HAS-A) than extend what it is (IS-A).

Initial design is simplified by identifying system object behaviors in separate interfaces instead of creating a hierarchical relationship to distribute behaviors among business-domain classes via inheritance. This approach more easily accommodates future requirements changes that would otherwise require a complete restructuring of business-domain classes in the inheritance model. Additionally, it avoids problems often associated with relatively minor changes to an inheritance-based model that includes several generations of classes.

Composition over inheritance: Drawbacks

One common drawback of using composition instead of inheritance is that methods being provided by individual components may have to be implemented in the derived type, even if they are only forwarding methods (this is true in most programming languages, but not all; see Avoiding drawbacks.) In contrast, inheritance does not require all of the base class's methods to be re-implemented within the derived class. Rather, the derived class only needs to implement (override) the methods having different behavior than the base class methods. This can require significantly less programming effort if the base class contains many methods providing default behavior and only a few of them need to be overridden within the derived class.

Composition over inheritance

```
// Base class
public abstract class Employee
{
   // Properties
   protected string Name { get; set; }
   protected int ID { get; set; }
   protected decimal PayRate { get; set; }
   protected int HoursWorked { get; }
   // Get pay for the current pay period
   public abstract decimal Pay();
}

// Derived subclass
public class HourlyEmployee : Employee
{
   // Get pay for the current pay period
   public override decimal Pay()
   {
      // Time worked is in hours
      return HoursWorked * PayRate;
   }
}
```

```
// Derived subclass
public class SalariedEmployee : Employee
{
   // Get pay for the current pay period
   public override decimal Pay()
   {
      // Pay rate is annual salary instead of hourly rate
      return HoursWorked * PayRate / 2087;
   }
}
```

GoF overview

What is Design Pattern

A **design pattern** is a general repeatable solution to a commonly occurring problem in software design.

A design pattern isn't a finished design. It is a description or template for how to solve a problem that can be used in many different situations.

Why Use Patterns?

- ✓ Clean code
- ✓ Easy to maintain
- ✓ Improves code readability and testability
- ✓ Reduce time for development
- ✓ Already tested solution with known pros and cons

Gang of Four (GoF)

In 1994, four authors *Erich Gamma*, *Richard Helm*, *Ralph Johnson* and *John Vlissides* published a book titled **Design Patterns - Elements of Reusable Object-Oriented Software** which initiated the concept of Design Pattern in Software development.

These authors are collectively known as **Gang of Four (GoF)**.

According to these authors design patterns are primarily based on the following principles of object orientated design:

- Program to an interface not an implementation
- Favor object composition over inheritance

General Responsibility Assignment Software Patterns (GRASP)

General Responsibility Assignment Software Patterns (or **Principles**), abbreviated **GRASP**, consist of guidelines for assigning responsibility to classes and objects in object-oriented design.

It is not related to the SOLID design principle.

The different patterns and principles used in GRASP are controller, creator, indirection, information expert, low coupling, high cohesion, polymorphism, protected variations, and pure fabrication.

All these patterns answer some software problems, and these problems are common to almost every software development project. These techniques have not been invented to create new ways of working, but to better document and standardize old, tried-and-tested programming principles in object-oriented design.

GoF: Usage of Design Pattern

Design Patterns have two main usages in software development.

- **Common platform for developers**

Design patterns provide a standard terminology and are specific to particular scenario. For example, a singleton design pattern signifies use of single object so all developers familiar with single design pattern will make use of single object and they can tell each other that program is following a singleton pattern.

- **Best Practices**

Design patterns have been evolved over a long period of time and they provide best solutions to certain problems faced during software development. Learning these patterns helps unexperienced developers to learn software design in an easy and faster way.

GoF: Types of Design Patterns

As per the design pattern reference book Design Patterns - Elements of Reusable Object-Oriented Software, there are **23 design patterns** which can be classified in three categories:

- **Creational Patterns**

These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. This gives program more flexibility in deciding which objects need to be created for a given use case.

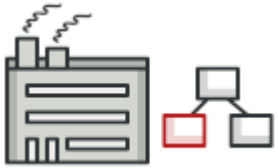
- **Structural Patterns**

These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.

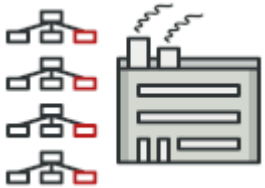
- **Behavioral Patterns**

These design patterns are specifically concerned with communication between objects.

GoF: Creational Design Patterns

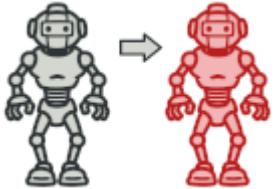


Factory Method
Also known as:
Virtual Constructor



Abstract Factory

Builder



Prototype
Also known as:
Clone

Singleton

Dependency Injection

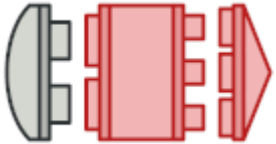
Resource Acquisition Is Initialization

Lazy initialization

Object pool

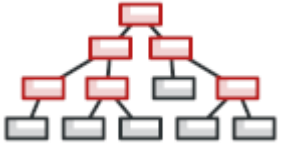
Multiton pattern

GOF: Structural Design Patterns

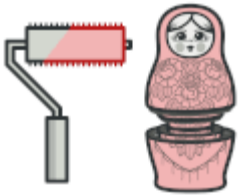


Adapter
Also known as: **Wrapper**

Bridge

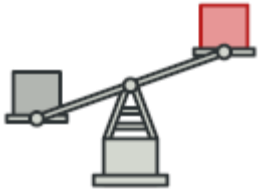


Composite
Also known as: **Object Tree**

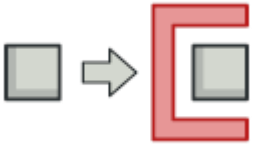


Decorator
Also known as: **Wrapper**

Facade



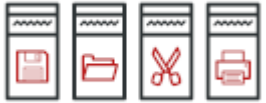
Flyweight
Also known as: **Cache**



Proxy

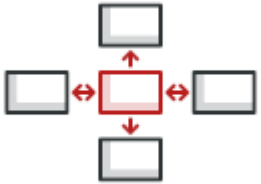
GOF: Behavioral Design Patterns

Chain of Responsibility
Also known as: **CoR**, **Chain of Command**



Command
Also known as: **Action**, **Transaction**

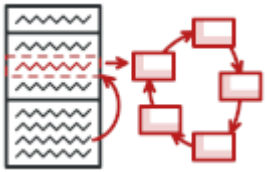
Iterator



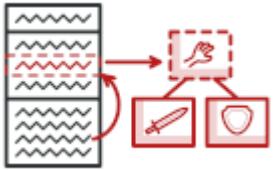
Mediator
Also known as: **Intermediary**, **Controller**

Memento
Also known as: **Snapshot**

Observer
Also known as: **Event-Subscriber**, **Listener**



State



Strategy

Template Method

Visitor

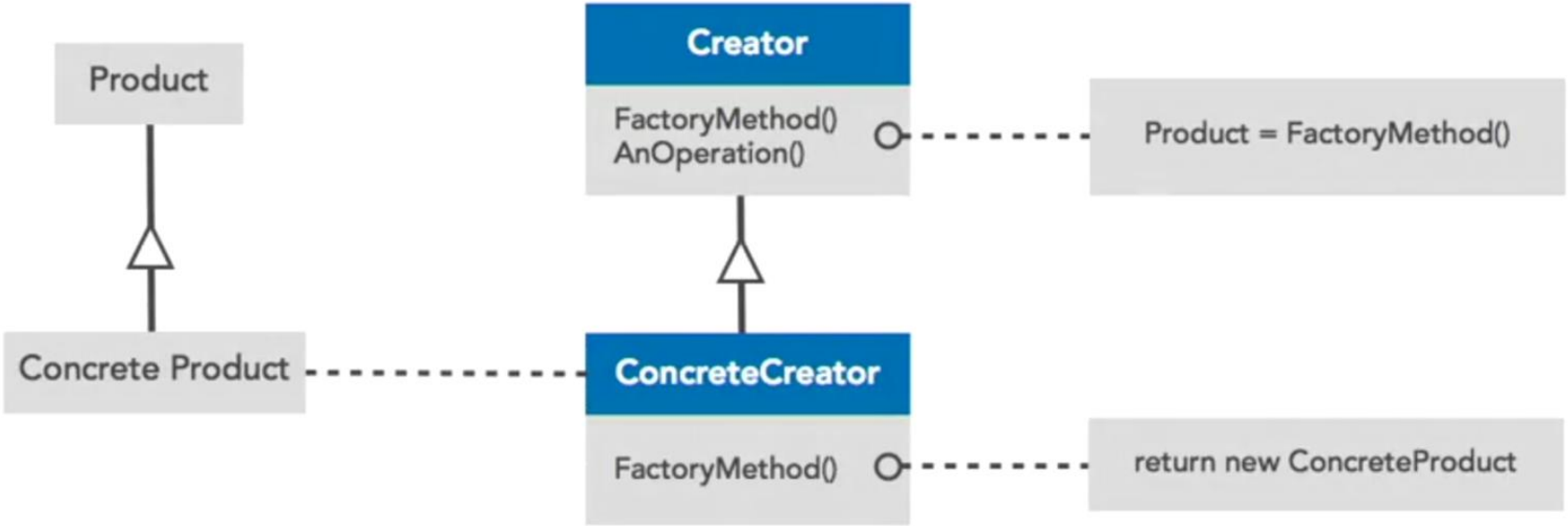
Design Patterns

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy Flyweight	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

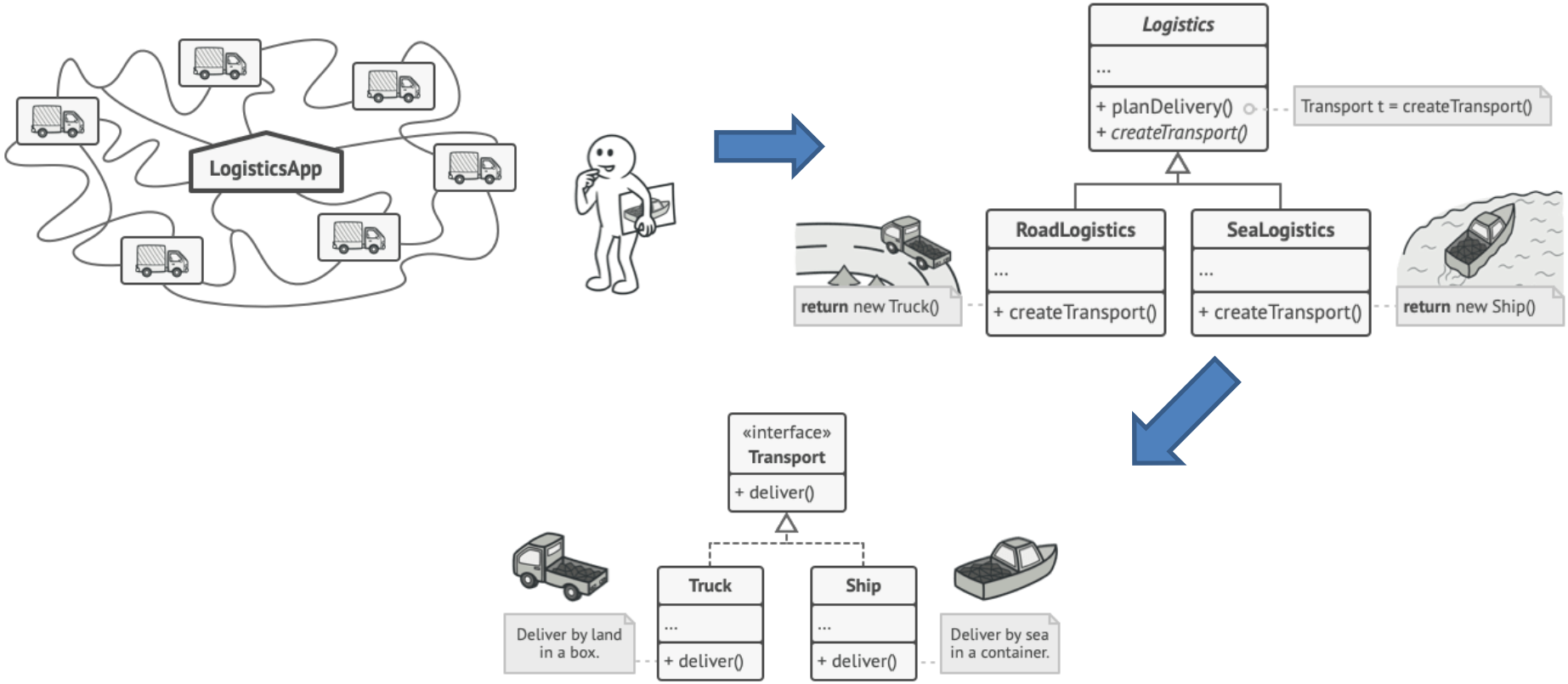
Factory method vs Builder,
Strategy vs Bridge,
Decorator vs Adapter,
Iterator,
Observer

Factory method (Virtual Constructor, Creational)

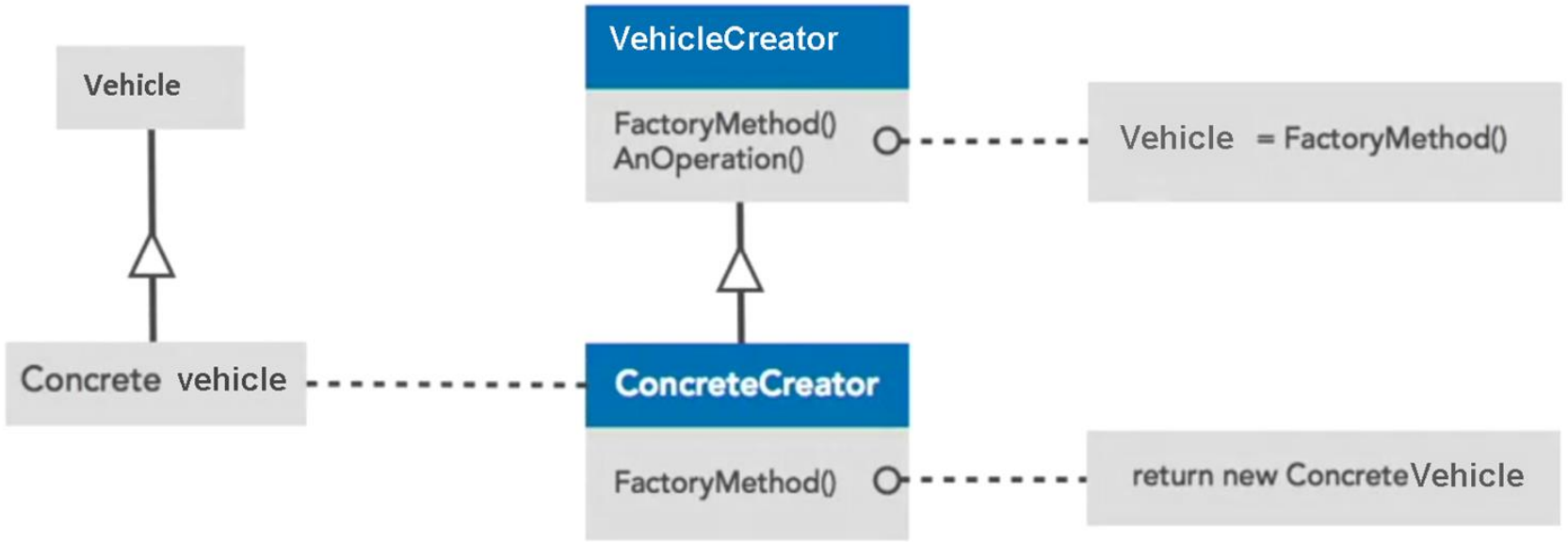
Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.



Factory method (Virtual Constructor, Creational)



Factory method (Virtual Constructor, Creational)



Factory method (Virtual Constructor, Creational)

Fabric Method Example

```
public class SimpleCalculator : Calculator, IOperand<double>
{
    internal SimpleCalculator()
    {
        MemoryNumber = "0";
    }
    public string MemoryNumber { get; set; }
    public double Sum(double first, double second)
    {
        return first + second;
    }
    public double Multiply(double first, double second)
    {
        return first * second;
    }
    public override string Calculate(string first, string second, string operation)
    {
        double firstArg = Convert.ToDouble(first);
        double secondArg = Convert.ToDouble(second);
        switch (operation)
        {
            case "+":
                return Sum(firstArg, secondArg).ToString();
            case "*":
                return Multiply(firstArg, secondArg).ToString();
            default:
                throw new Exception("No such operation!");
        }
    }
}
```

Factory method (Virtual Constructor, Creational)

Fabric Method Example

```
public class ComplexCalculator : Calculator, IOperand<Complex>
{
   internal ComplexCalculator()
   {
      MemoryNumber = "0";
   }
   public string MemoryNumber { get; set; }
   public Complex Sum(Complex first, Complex second)
   {
      return first + second;
   }
   public Complex Multiply(Complex first, Complex second)
   {
      return first * second;
   }
   public override string Calculate(string first, string second, string operation)
   {
   }
}
```

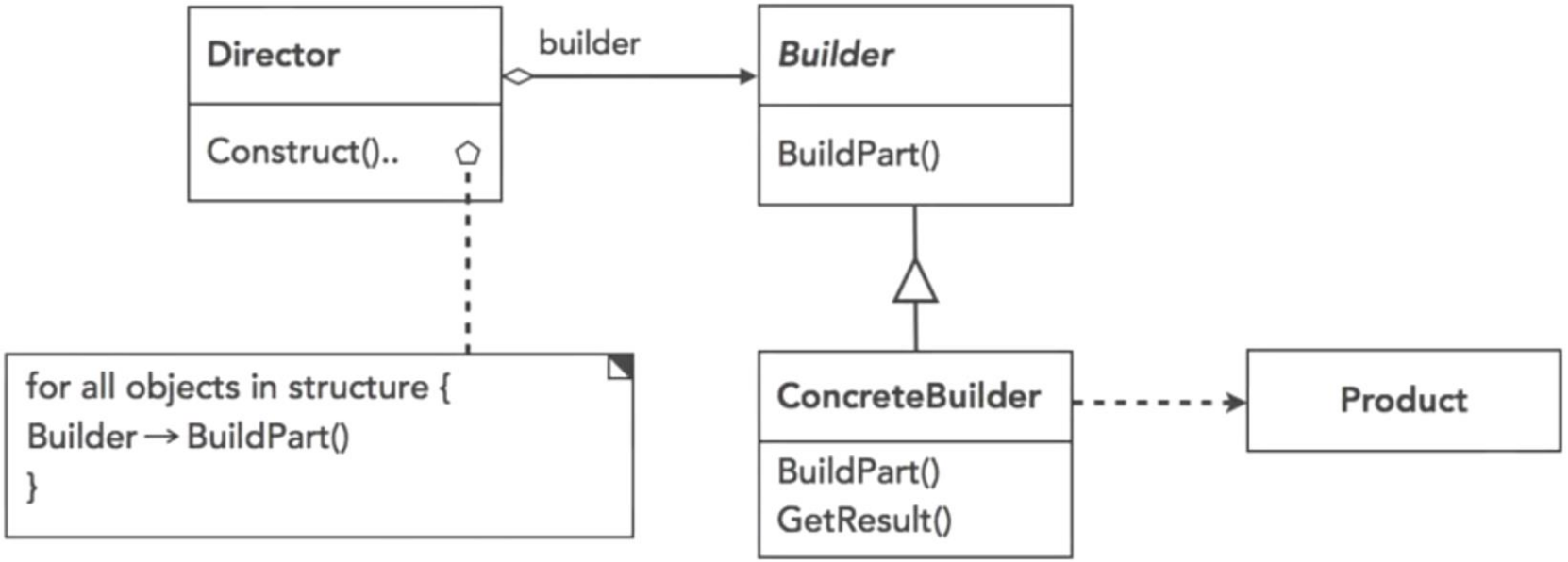
Factory method (Virtual Constructor, Creational)

Calculator Factory

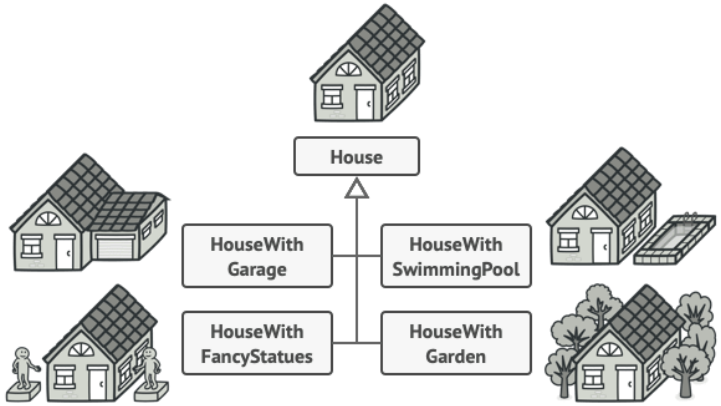
```
public class CalculatorFactory
{
   public static Calculator Create(CalculatorTypeEnum calculatorType)
   {
      switch (calculatorType)
      {
           case CalculatorTypeEnum.Digit:
             return new SimpleCalculator();
           case CalculatorTypeEnum.Complex:
             return new ComplexCalculator();
           default:
             throw new NotImplementedException();
      }
   }
}
```

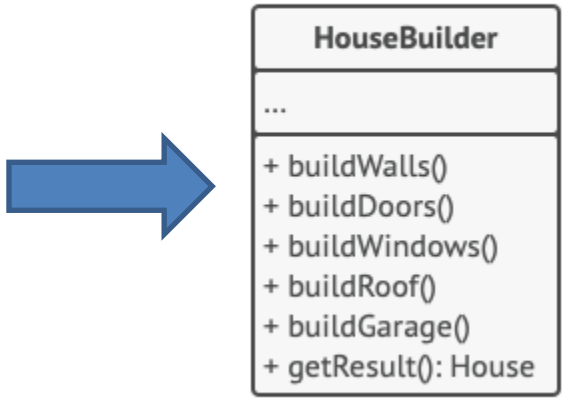
Builder

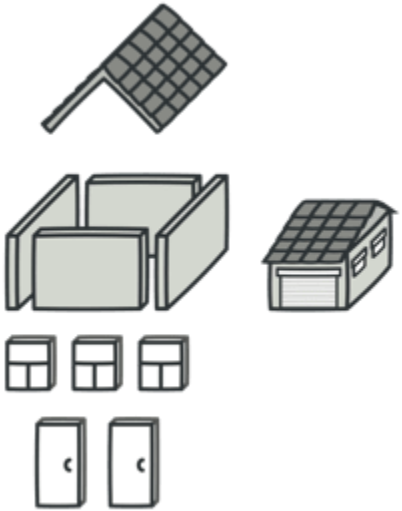
Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.



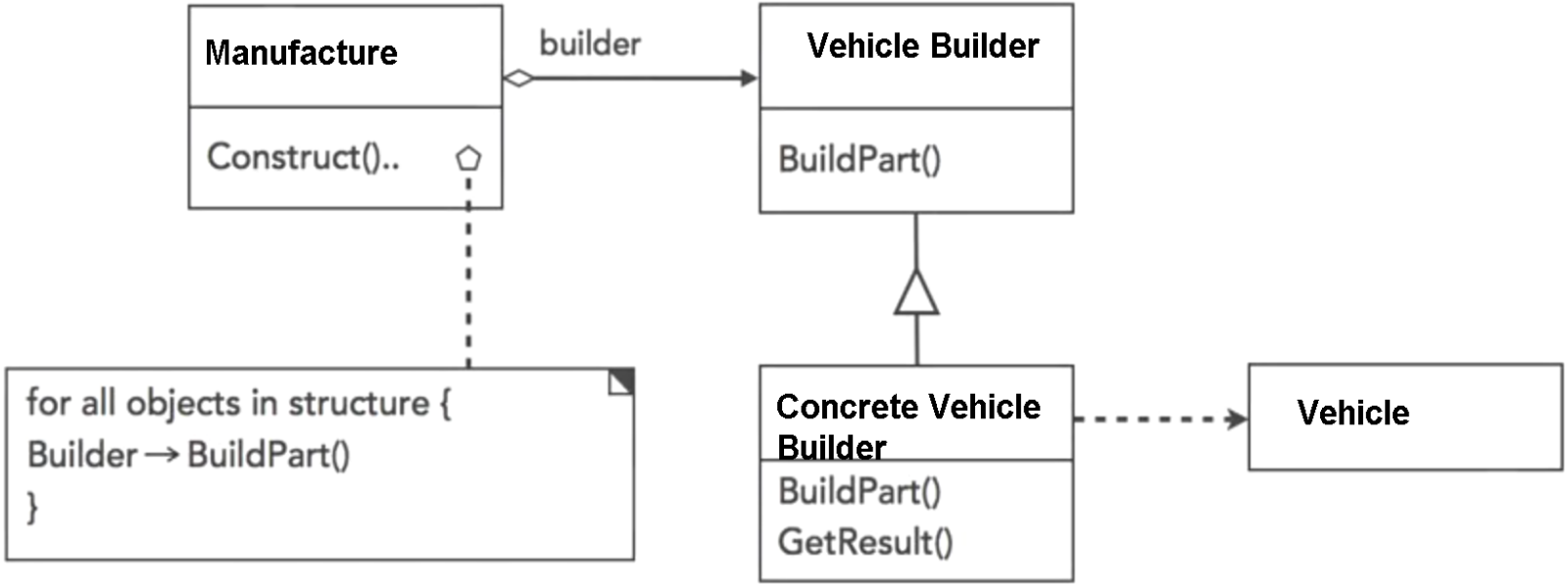
Builder


House
HouseWith Garage
HouseWith SwimmingPool
HouseWith FancyStatues
HouseWith Garden


HouseBuilder
...
+ buildWalls()
+ buildDoors()
+ buildWindows()
+ buildRoof()
+ buildGarage()
+ getResult(): House



Builder

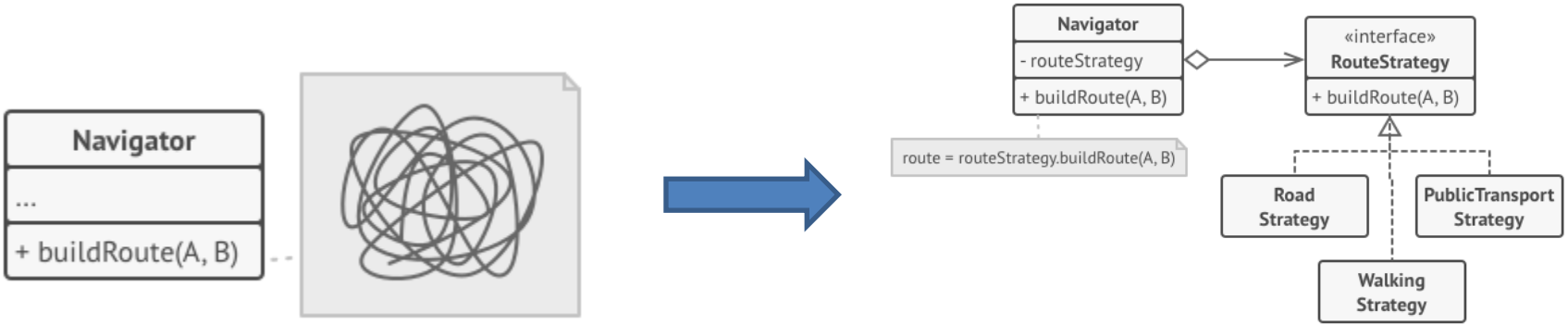

Manufacture
builder
Vehicle Builder
Construct()..
BuildPart()
for all objects in structure {
Builder → BuildPart()
}
Concrete Vehicle Builder
BuildPart()
GetResult()
Vehicle

Factory method vs Builder

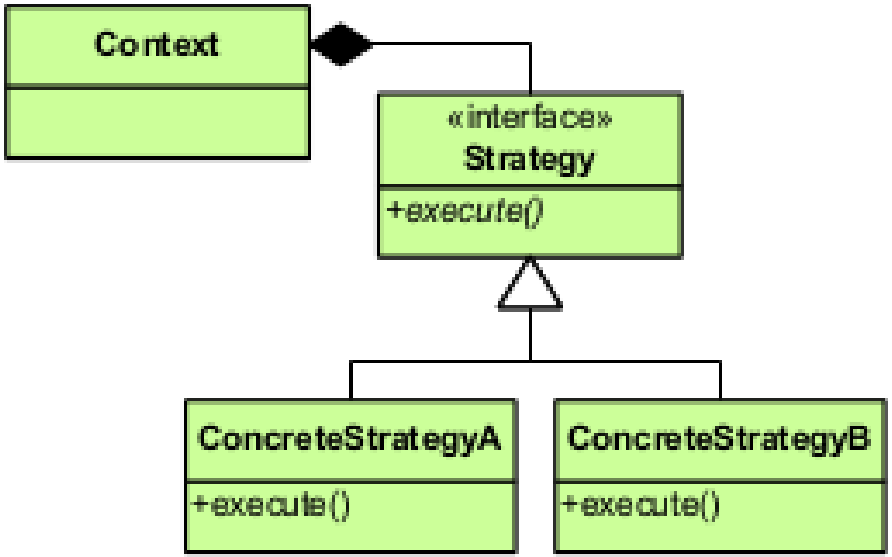
Many designs start by using Factory Method (less complicated and more customizable via subclasses) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, but more complicated).

Strategy

The strategy pattern (also known as the policy pattern) is a behavioral software design pattern that enables selecting an algorithm at runtime. Instead of implementing a single algorithm directly, code receives run-time instructions as to which in a family of algorithms to use.



Strategy


Context
«interface»
Strategy
+execute()
ConcreteStrategyA
+execute()
ConcreteStrategyB
+execute()

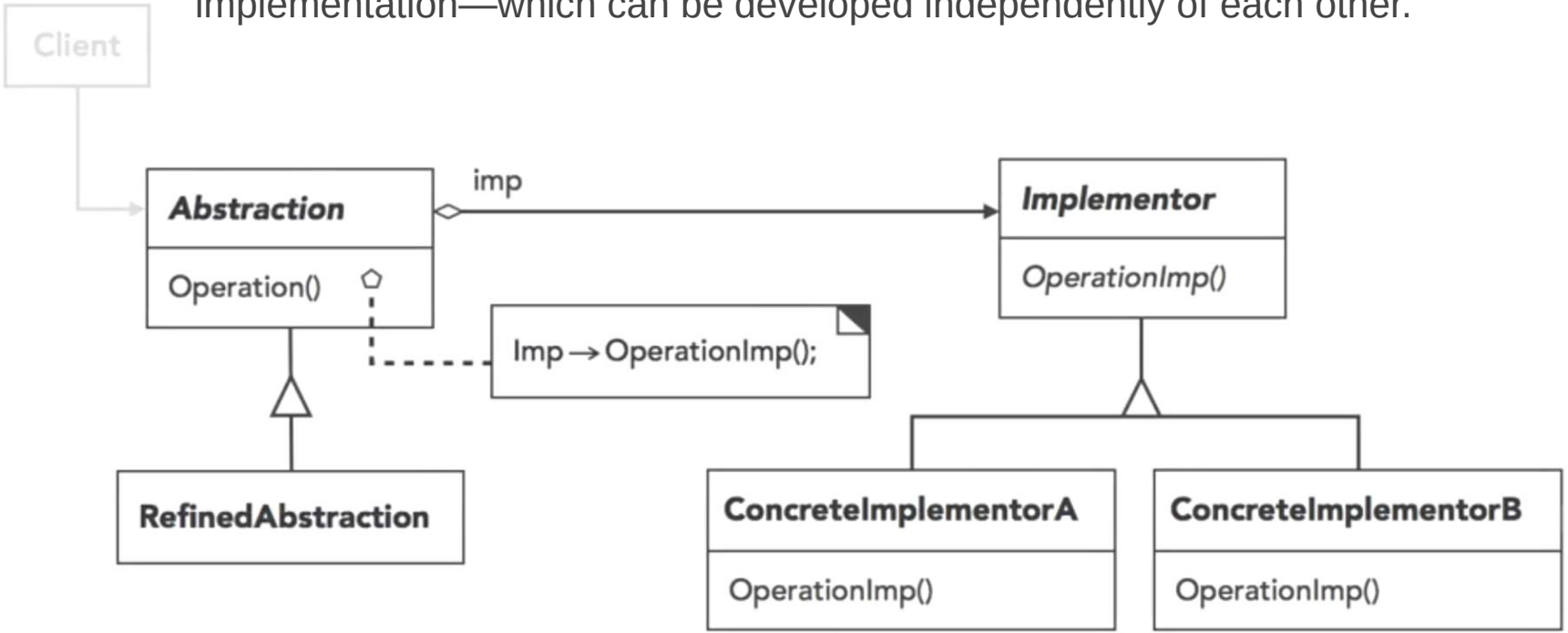
Strategy

```
public interface IStrategy
{
    void Algorithm();
}

public class ConcreteStrategy1 : IStrategy
{
    public void Algorithm()
    { }
}

public class ConcreteStrategy2 : IStrategy
{
    public void Algorithm()
    { }
}
```

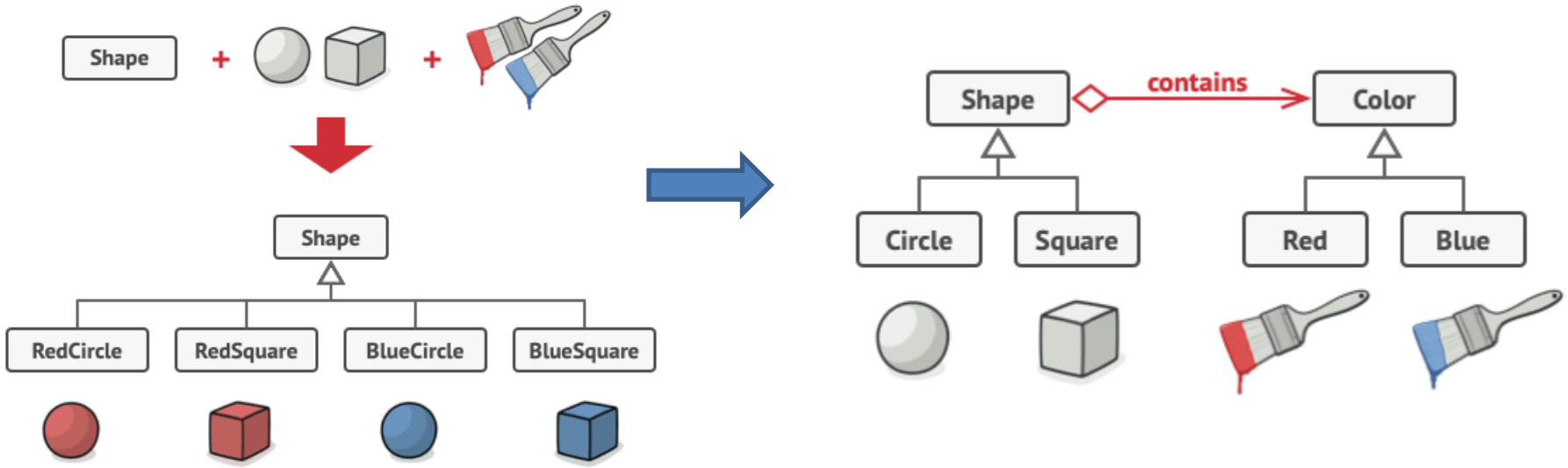
```
public class Context
{
    public IStrategy ContextStrategy { get; set; }

    public Context(IStrategy _strategy)
    {
        ContextStrategy = _strategy;
    }

    public void ExecuteAlgorithm()
    {
        ContextStrategy.Algorithm();
    }
}
```

Bridge

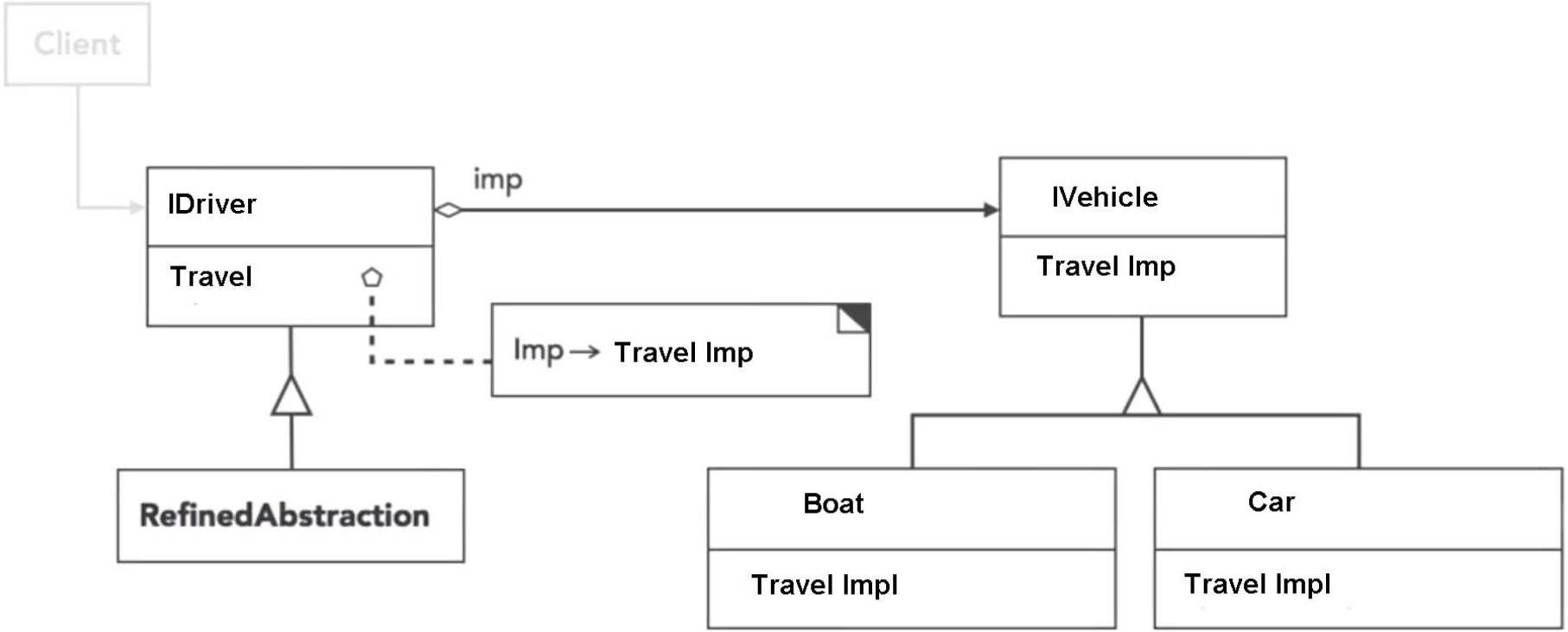
Bridge is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.



Bridge


Shape
+
+
Shape
RedCircle
RedSquare
BlueCircle
BlueSquare
Shape
contains
Color
Circle
Square
Red
Blue

Bridge

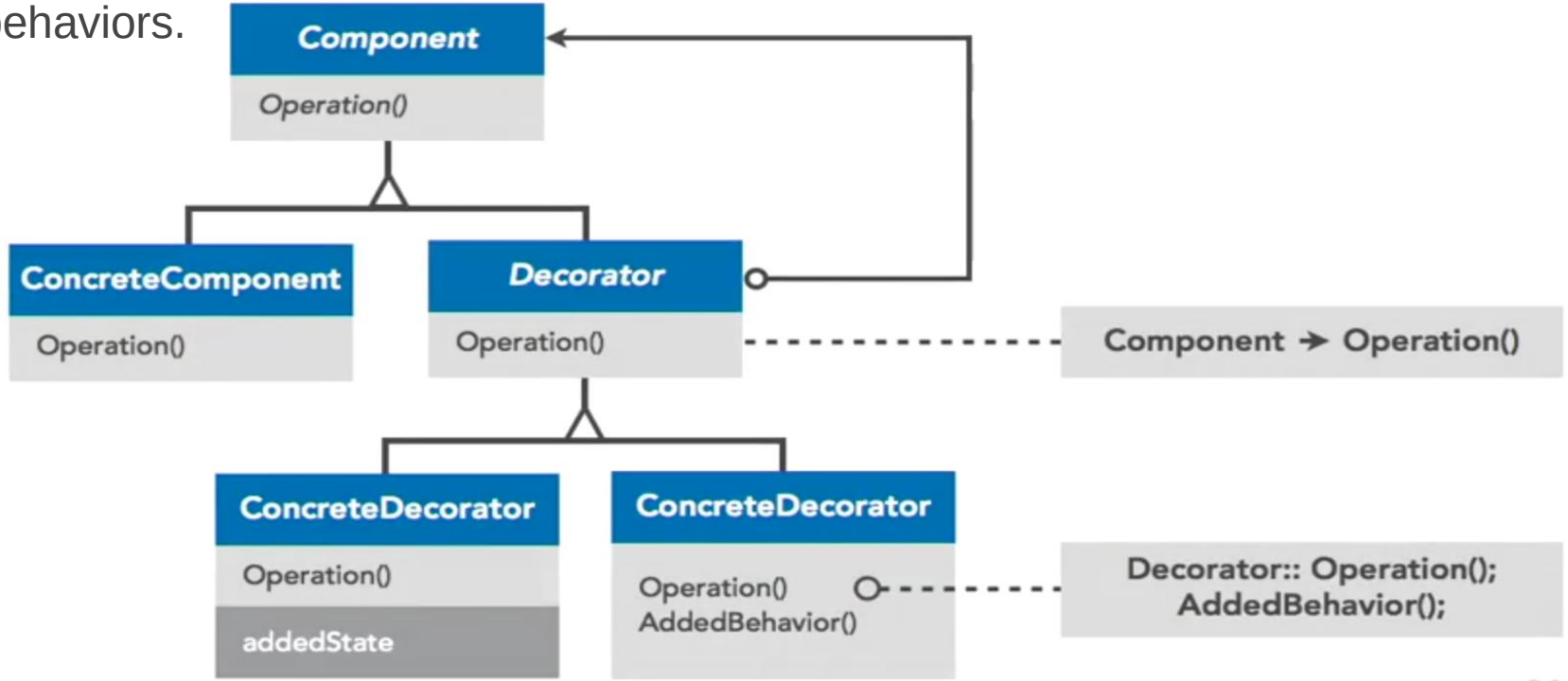

Client
IDriver
Travel
imp
IVehicle
Travel Imp
Imp → Travel Imp
RefinedAbstraction
Boat
Travel Impl
Car
Travel Impl

Strategy vs Bridge

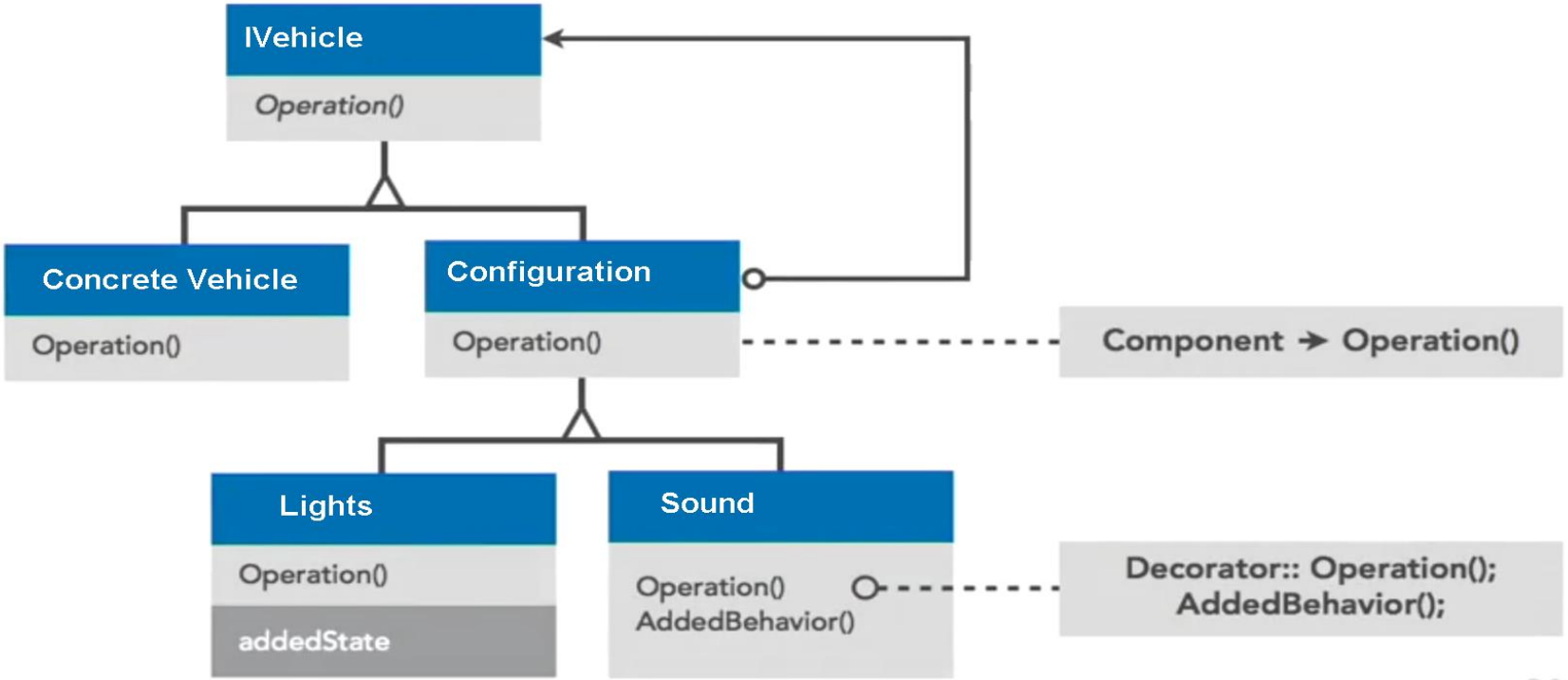
Bridge, **State**, **Strategy** (and to some degree **Adapter**) have very similar structures. Indeed, all of these patterns are based on composition, which is delegating work to other objects. However, they all solve different problems. A pattern isn't just a recipe for structuring your code in a specific way. It can also communicate to other developers the problem the pattern solves.

Decorator

Decorator is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

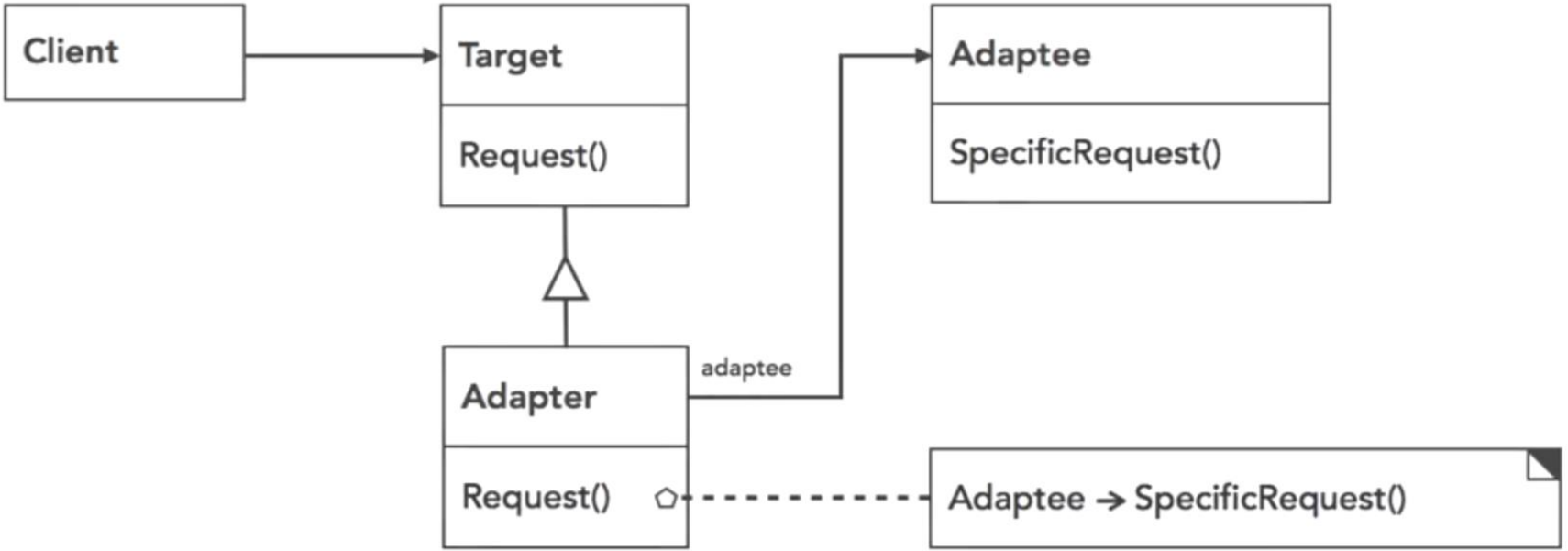


Decorator

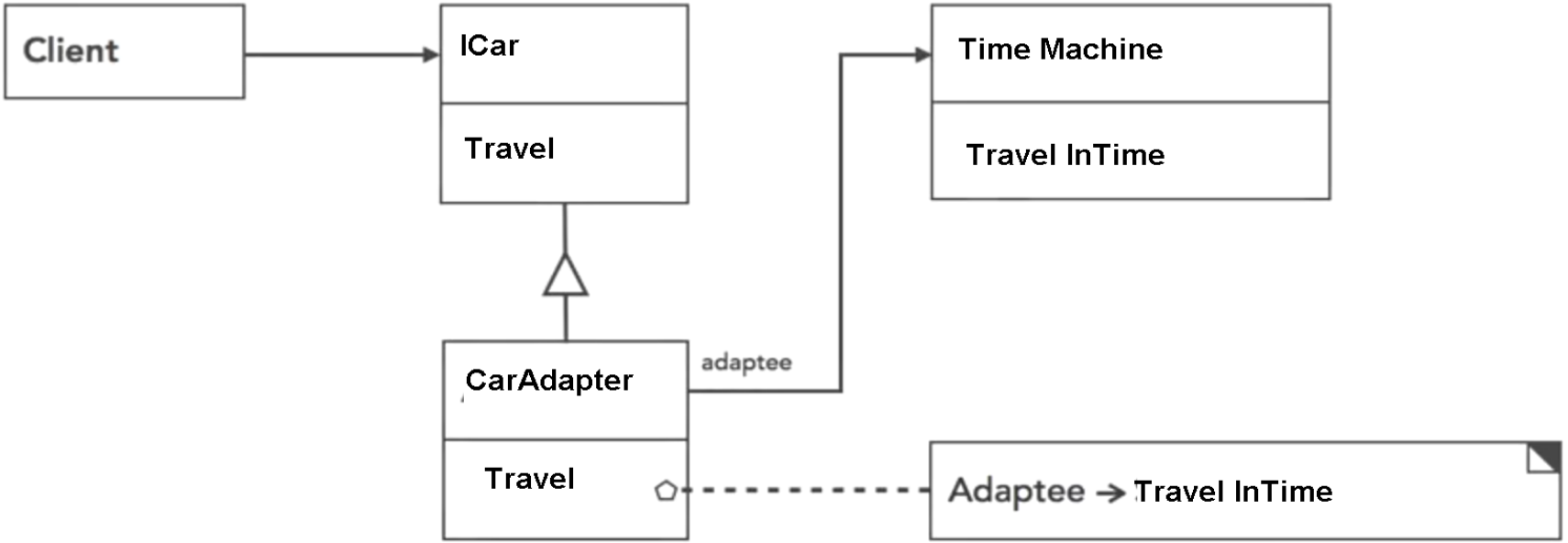

IVehicle
Operation()
Concrete Vehicle
Operation()
Configuration
Operation()
Component → Operation()
Lights
Operation()
addedState
Sound
Operation()
AddedBehavior()
Decorator:: Operation();
AddedBehavior();

Adapter

Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate.



Adapter


Client
ICar
Travel
Time Machine
Travel InTime
CarAdapter
adaptee
Travel
Adaptee → Travel InTime

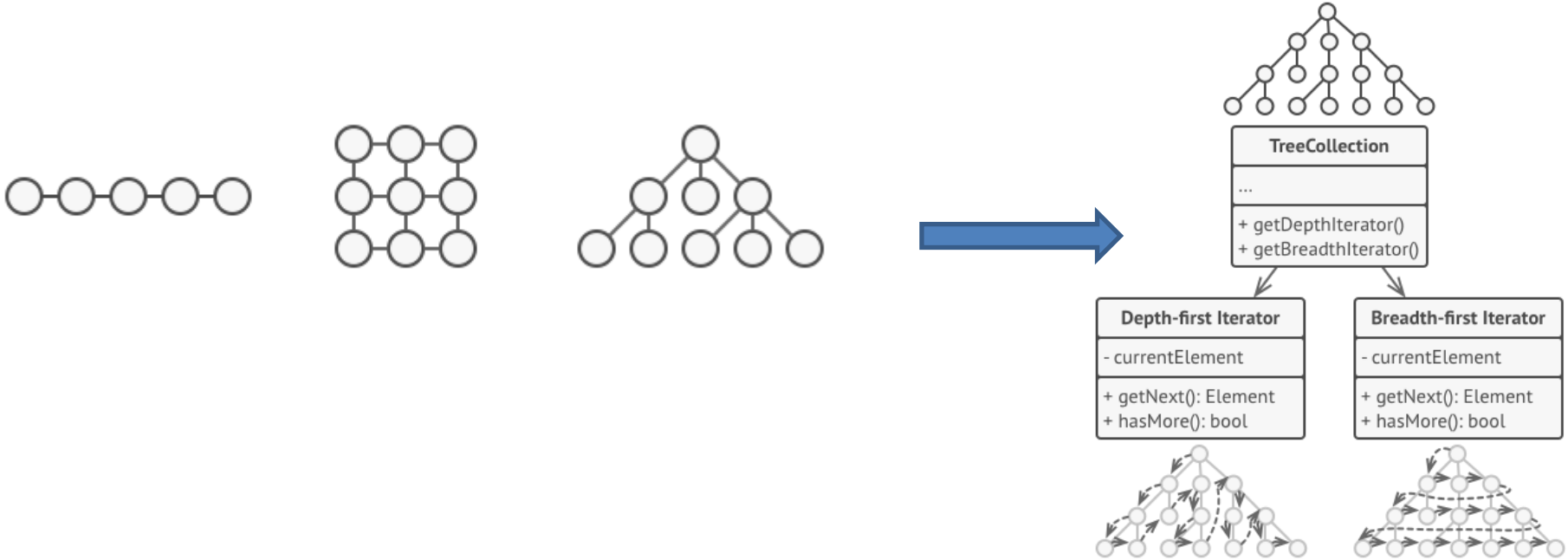
Decorator vs Adapter

Adapter changes the interface of an existing object, while **Decorator** enhances an object without changing its interface. In addition, Decorator supports recursive composition, which isn't possible when you use Adapter.

Adapter provides a different interface to the wrapped object, **Proxy** provides it with the same interface, and **Decorator** provides it with an enhanced interface.

Iterator

Iterator is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).

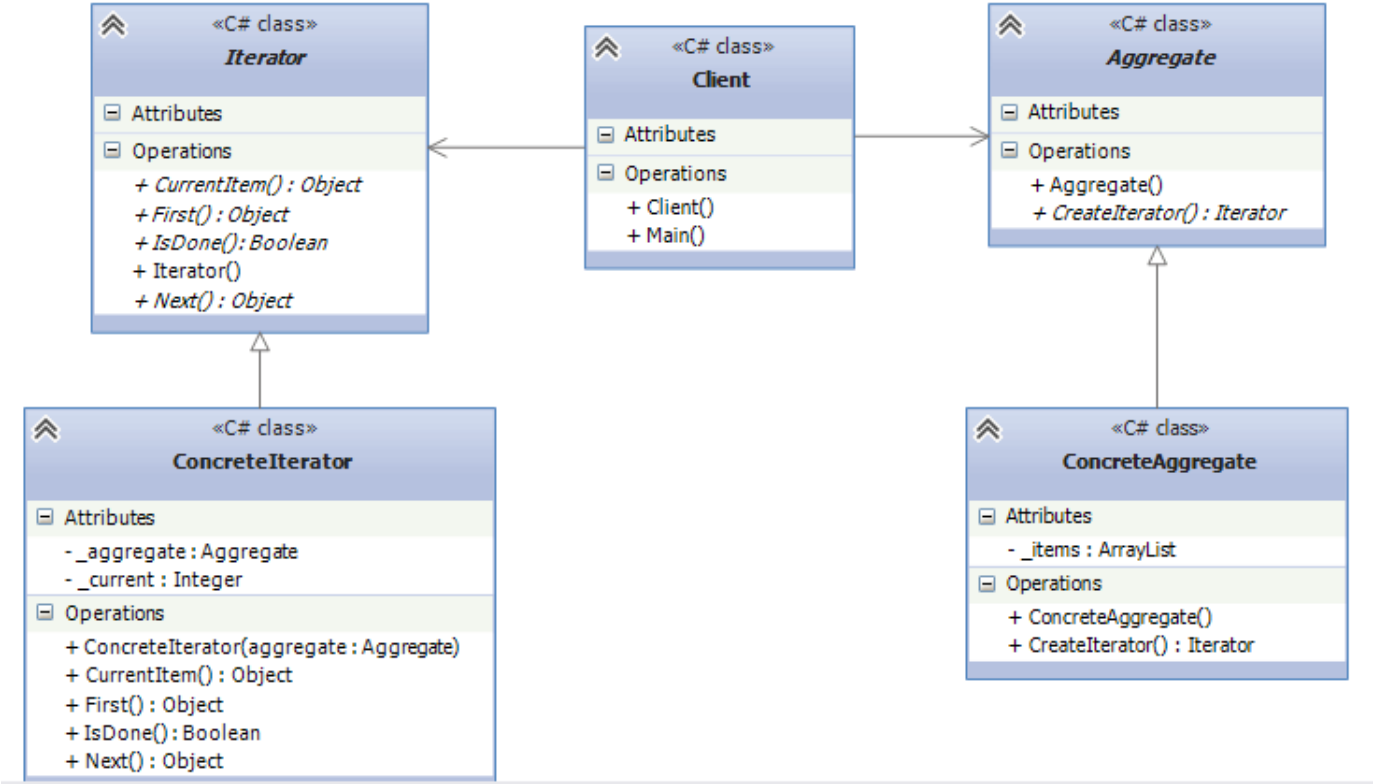


Iterator

```
public interface IEnumerator
{
    bool MoveNext();
    object Current { get;
    void Reset();
}
```

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

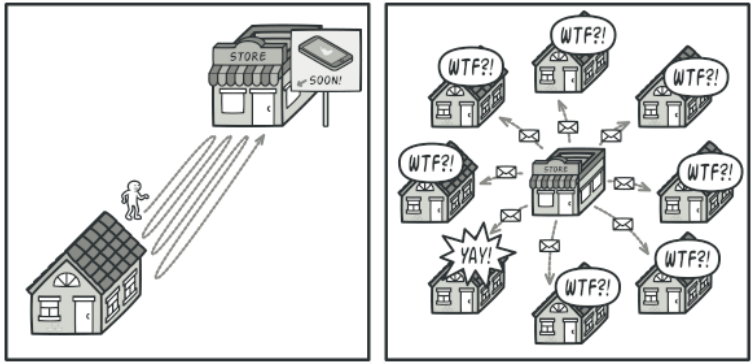
Iterator

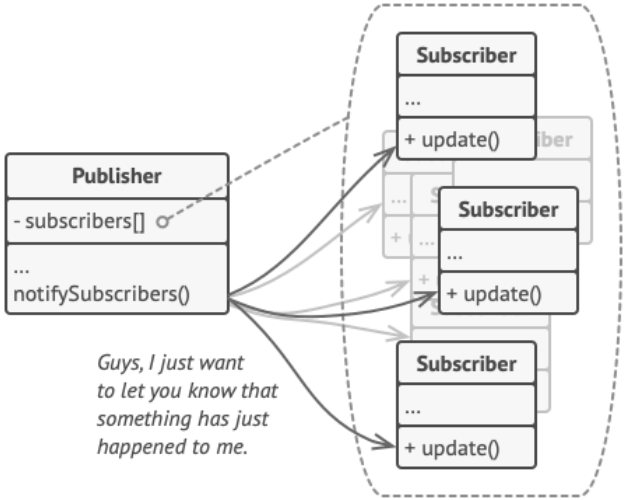

«C# class»
Iterator
Attributes
Operations
+ CurrentItem() : Object
+ First() : Object
+ IsDone(): Boolean
+ Iterator()
+ Next() : Object
«C# class»
Client
Attributes
Operations
+ Client()
+ Main()
«C# class»
Aggregate
Attributes
Operations
+ Aggregate()
+ CreateIterator() : Iterator
«C# class»
ConcreteIterator
Attributes
- _aggregate : Aggregate
- _current : Integer
Operations
+ ConcreteIterator(aggregate : Aggregate)
+ CurrentItem() : Object
+ First() : Object
+ IsDone() : Boolean
+ Next() : Object
«C# class»
ConcreteAggregate
Attributes
- _items : ArrayList
Operations
+ ConcreteAggregate()
+ CreateIterator() : Iterator

Observer

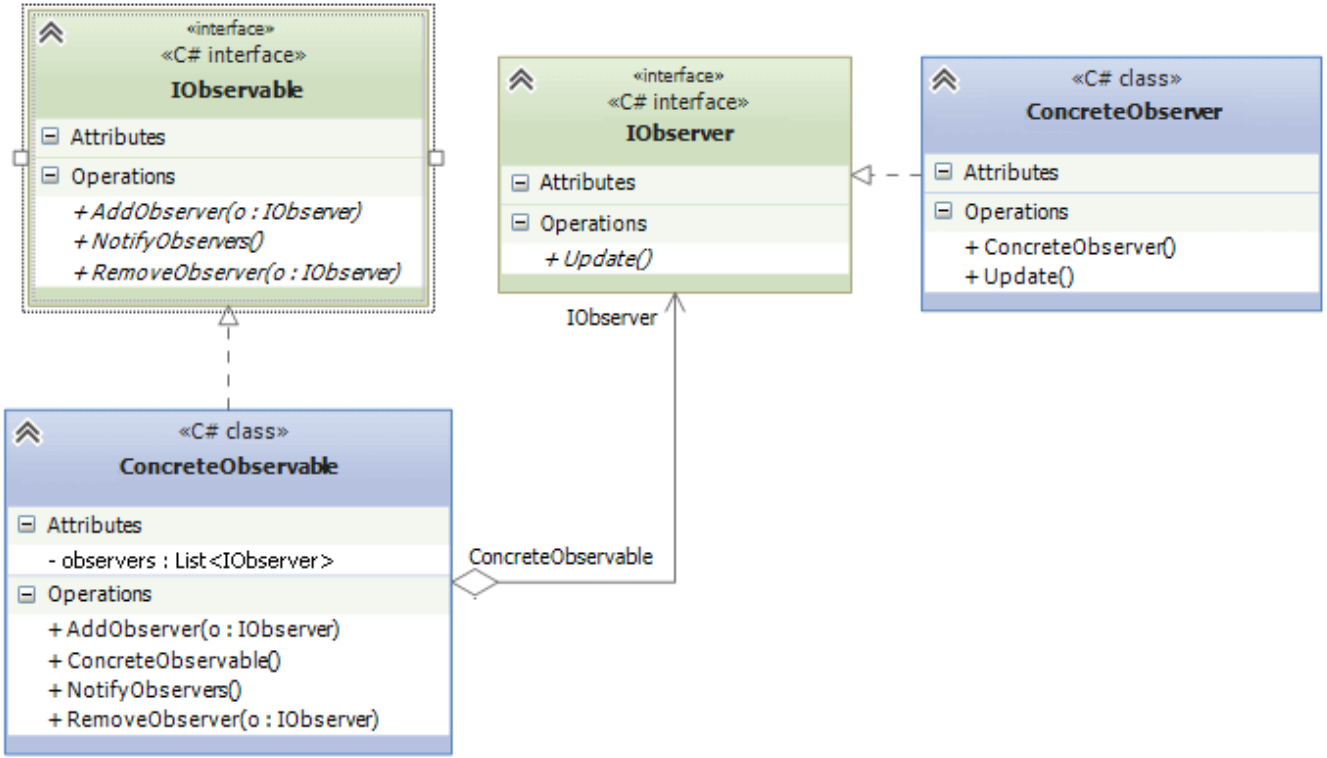
Also known as: Event-Subscriber, Listener

Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.





Observer


«interface»
«C# interface»
IObservable
Attributes
Operations
+ AddObserver(o : IObserver)
+ NotifyObservers()
+ RemoveObserver(o : IObserver)
«interface»
«C# interface»
IObserver
Attributes
Operations
+ Update()
«C# class»
ConcreteObserver
Attributes
Operations
+ ConcreteObserver()
+ Update()
IObserver
«C# class»
ConcreteObservable
Attributes
- observers : List<IObserver>
Operations
+ AddObserver(o : IObserver)
+ ConcreteObservable()
+ NotifyObservers()
+ RemoveObserver(o : IObserver)
ConcreteObservable

To Use or not to Use

- Patterns do not magically improve the quality of your code
- Use Patterns everywhere?
- KISS first, patterns later, maybe much later
- Don't ever try to force your code into a specific pattern
- Pattern or Anti-pattern

.NET Online UA Training Course Feedback

I hope that you will find this material useful.

If you find errors or inaccuracies in this material or know how to improve it, please report on to the electronic address:

Oleksii_Leunenko@epam.com

With the note [.NET Online UA Training Course Feedback]

Thank you.

Q&A

UA .NET Online LAB