M05

DESIGN PATTERNS

Dr. Issarapong Khuankrue

KMUTT INC we make computers do more.
KMUTT King Mongkut's University of Technology Thonburi

OUTLINE

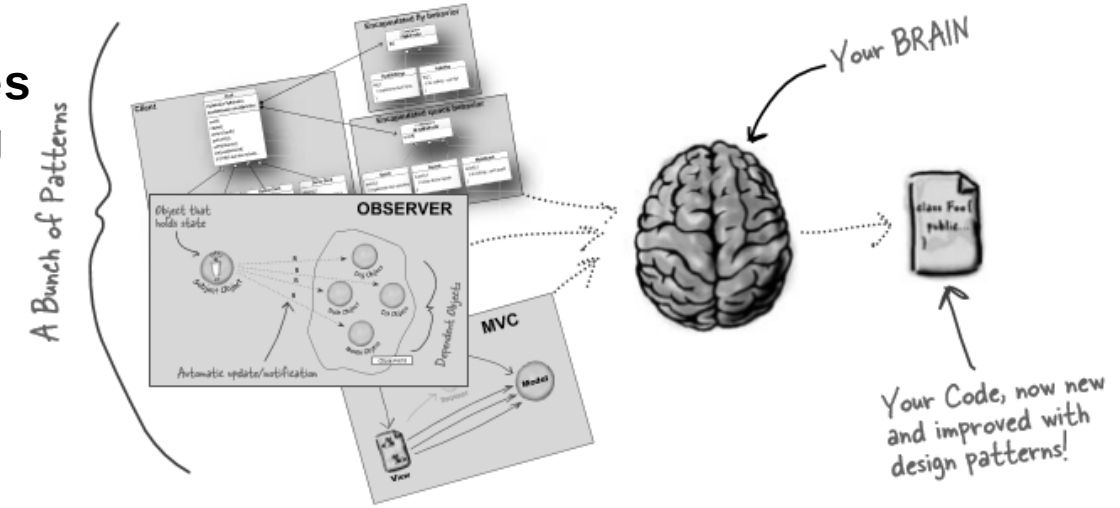
https://refactoring.guru/

- Motivation
- Classification of patterns
  - **Creational patterns**
    - Abstract Factory
    - Factory Method
    - Prototype
    - Singleton
  - **Structural patterns**
    - Adapter
    - Decorator
    - Proxy
  - **Behavioural patterns**
    - Observer

WHAT'S A DESIGN PATTERN?

Design patterns represent the **best practices and typical solutions to commonly occurring problems in software design.**

Design patterns are **solutions to general problems** that software developers faced during software development.



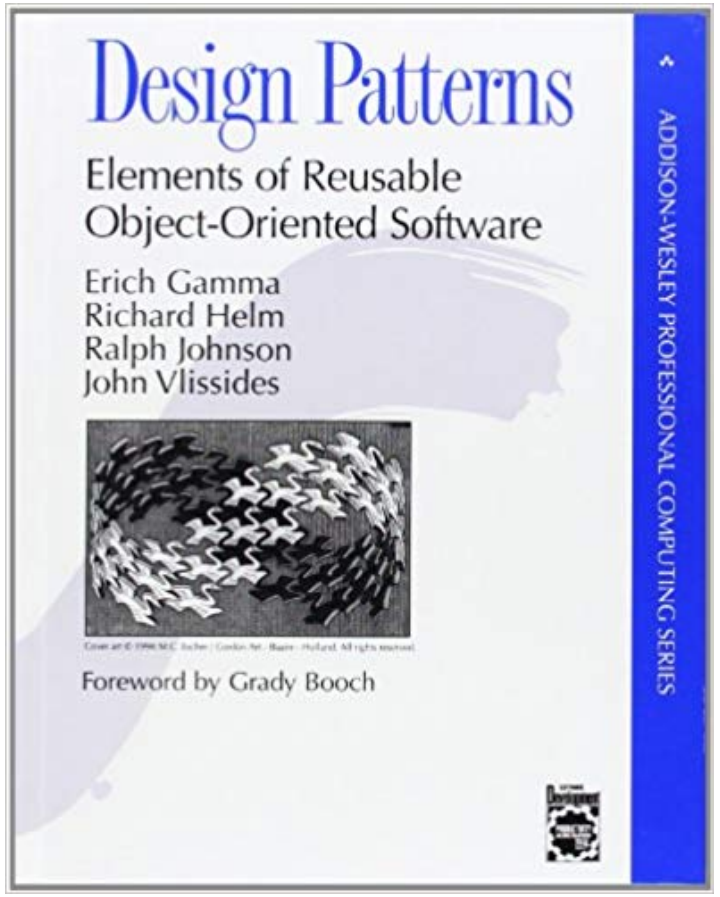
Patterns are often confused with algorithms, because both concepts describe typical solutions to some known problems.

Algorithm always defines a clear set of actions that can achieve some goal.

Pattern is a more high-level description of a solution. The code of the same pattern applied to two different programs may be different.

MOTIVATION

- Applied the concept of design patterns to programming.
- These solutions were obtained by trial and error by numerous software developers over quite a substantial period.
- **So why would you spend time learning them?**
  - A toolkit of **tried and tested solutions** to common problems in software design. Even if you never encounter these problems, knowing patterns is still useful because it teaches you how to solve all sorts of problems using principles of object-oriented design.
  - Define a **common language that you and your teammates can use** to communicate more efficiently.



CLASSIFICATION OF PATTERNS

CREATIONAL PATTERNS

Provide object creation mechanisms that increase flexibility and reuse of existing code.

STRUCTURAL PATTERNS

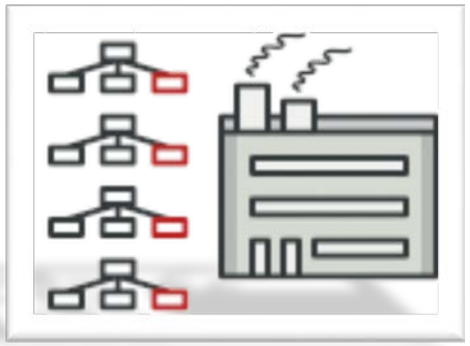
Explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient.

BEHAVIORAL PATTERNS

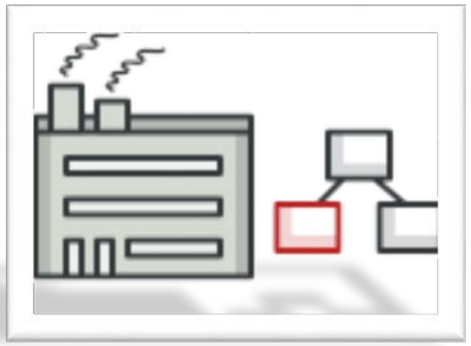
Take care of effective communication and the assignment of responsibilities between objects.

CREATIONAL PATTERNS

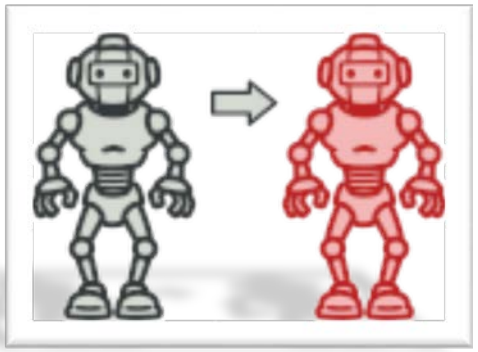
- These patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.



Abstract Factory

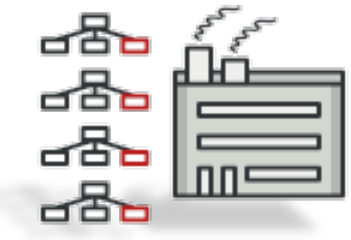


Factory Method



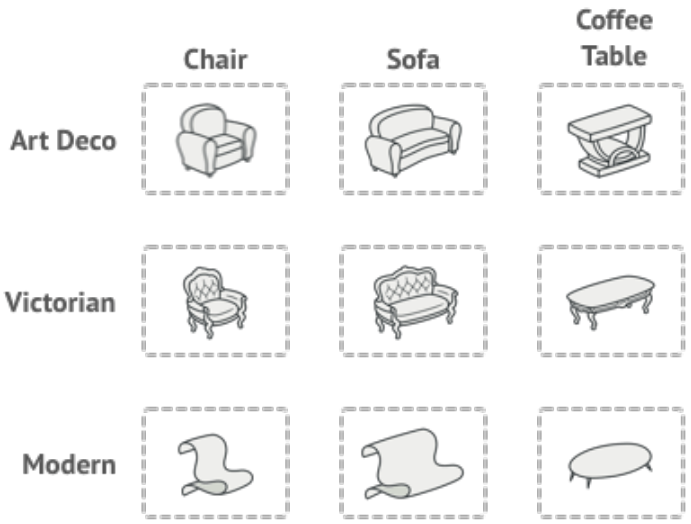
Prototype

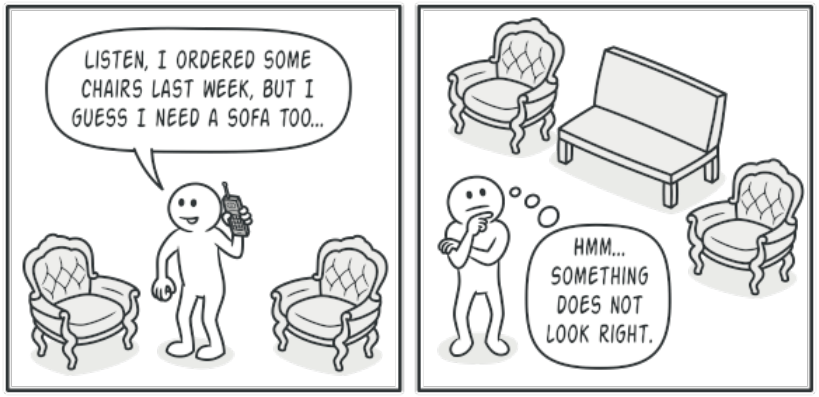
Singleton

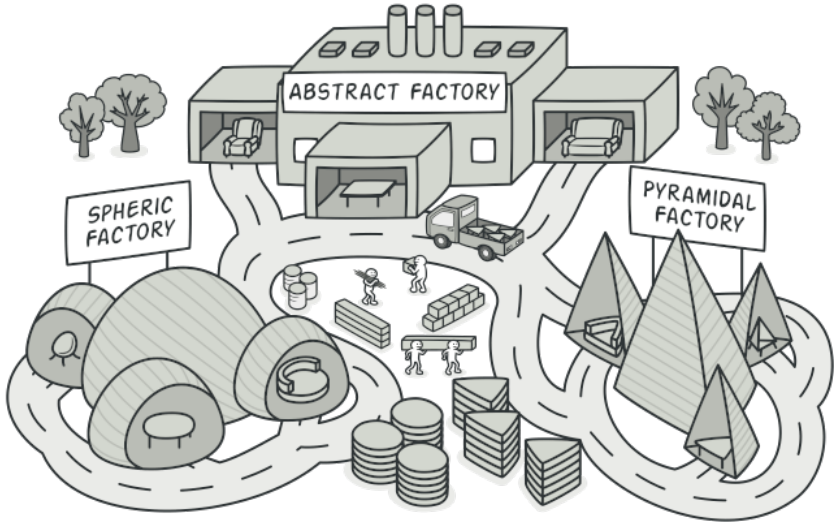


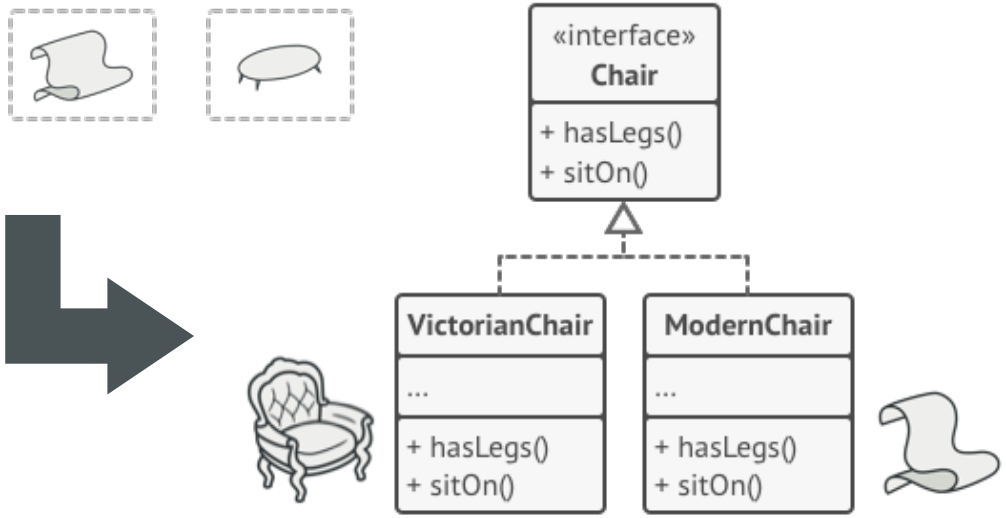
ABSTRACT FACTORY

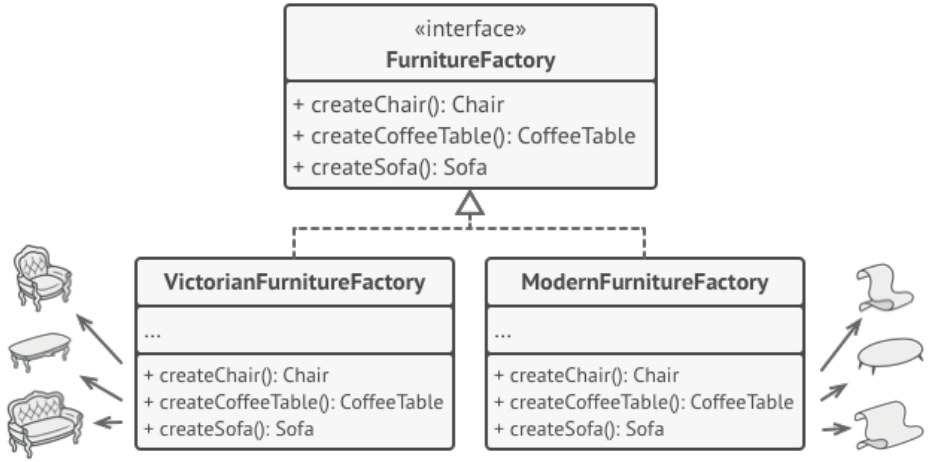
is a creational design pattern that produce families of related objects without specifying their concrete classes.

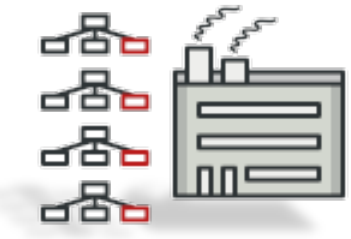






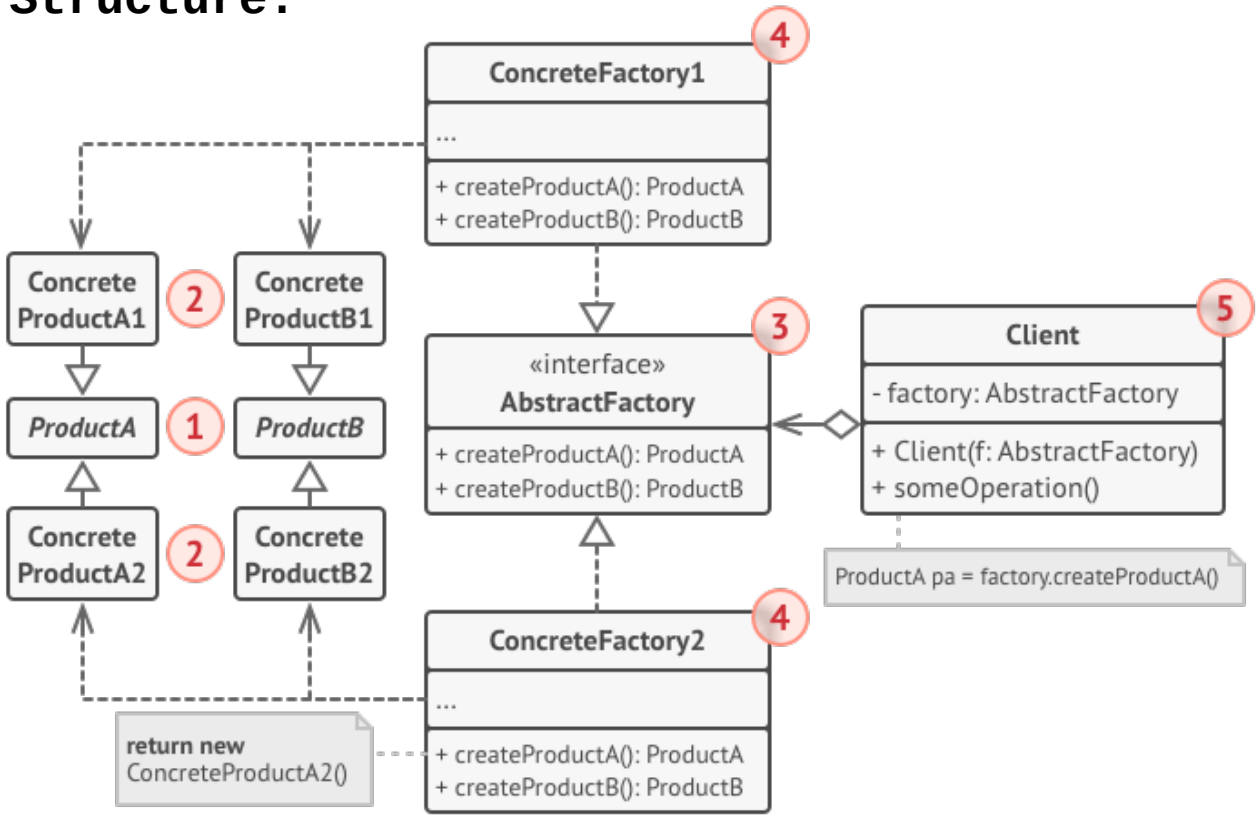






ABSTRACT FACTORY

Structure:



1.Abstract Products declare interfaces for a set of distinct but related products which make up a product family.

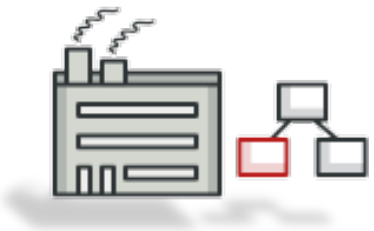
2.Concrete Products are various implementations of abstract products, grouped by variants.

3.Abstract Factory interface declares a set of methods for creating each of the abstract products.

4.Concrete Factories implement creation methods of the abstract factory. Each concrete factory corresponds to a specific variant of products and creates only those product variants.

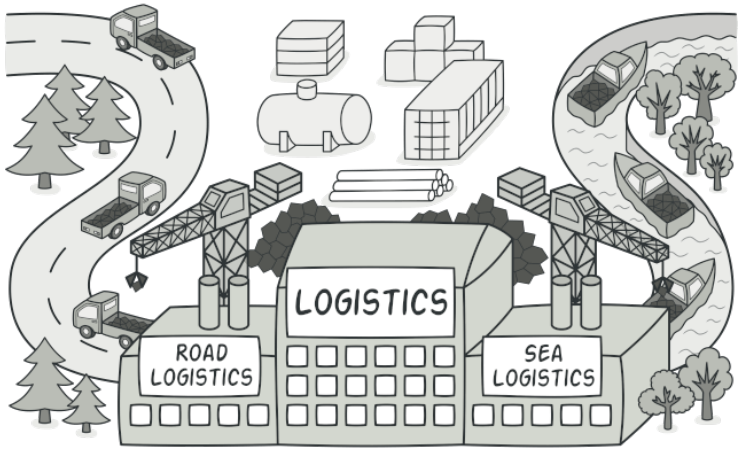
5.Although concrete factories instantiate concrete products, signatures of their creation methods must return corresponding *abstract* products.

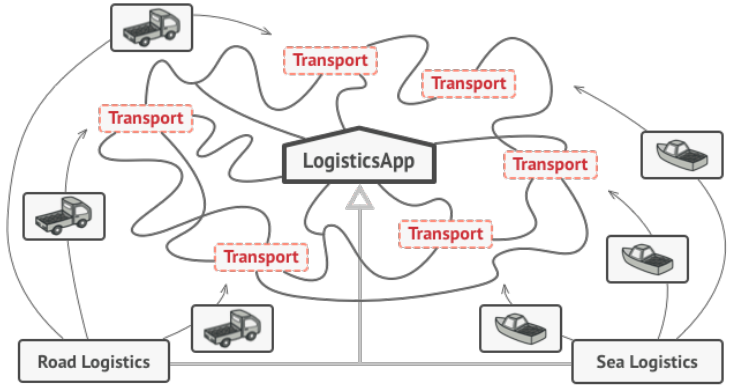
The **Client** can work with any concrete factory/product variant, if it communicates with their objects via abstract interfaces.

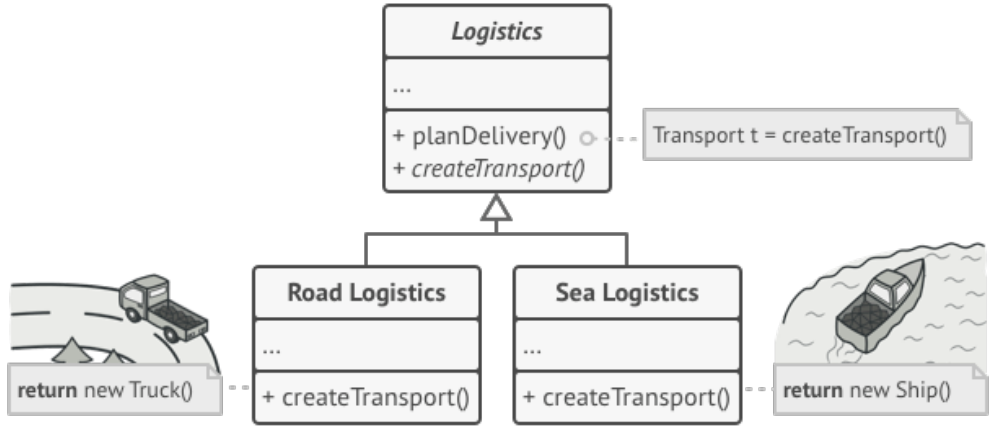


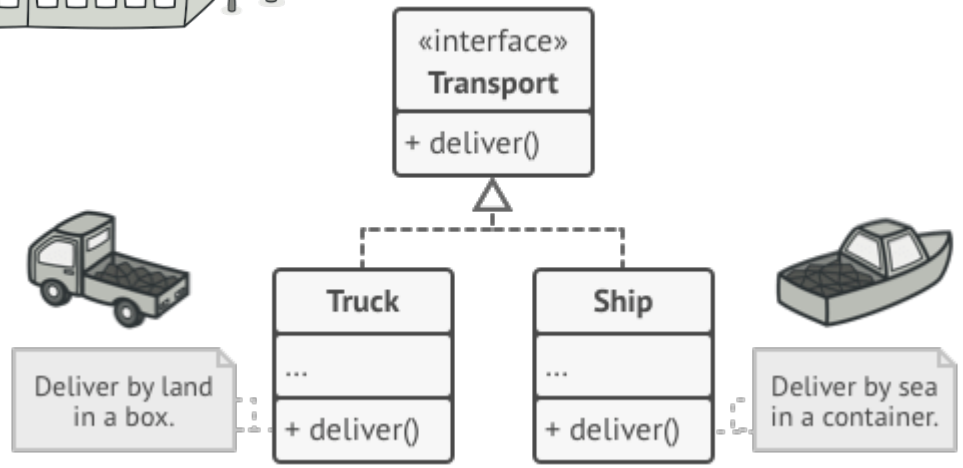
FACTORY METHOD

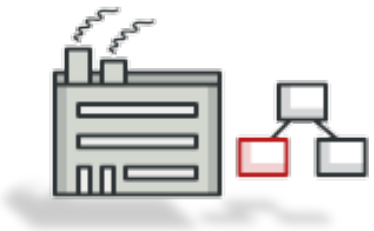
is a creational design pattern that provides an interface for creating objects in a superclass **but** allows subclasses to alter/change the type of objects that will be created.





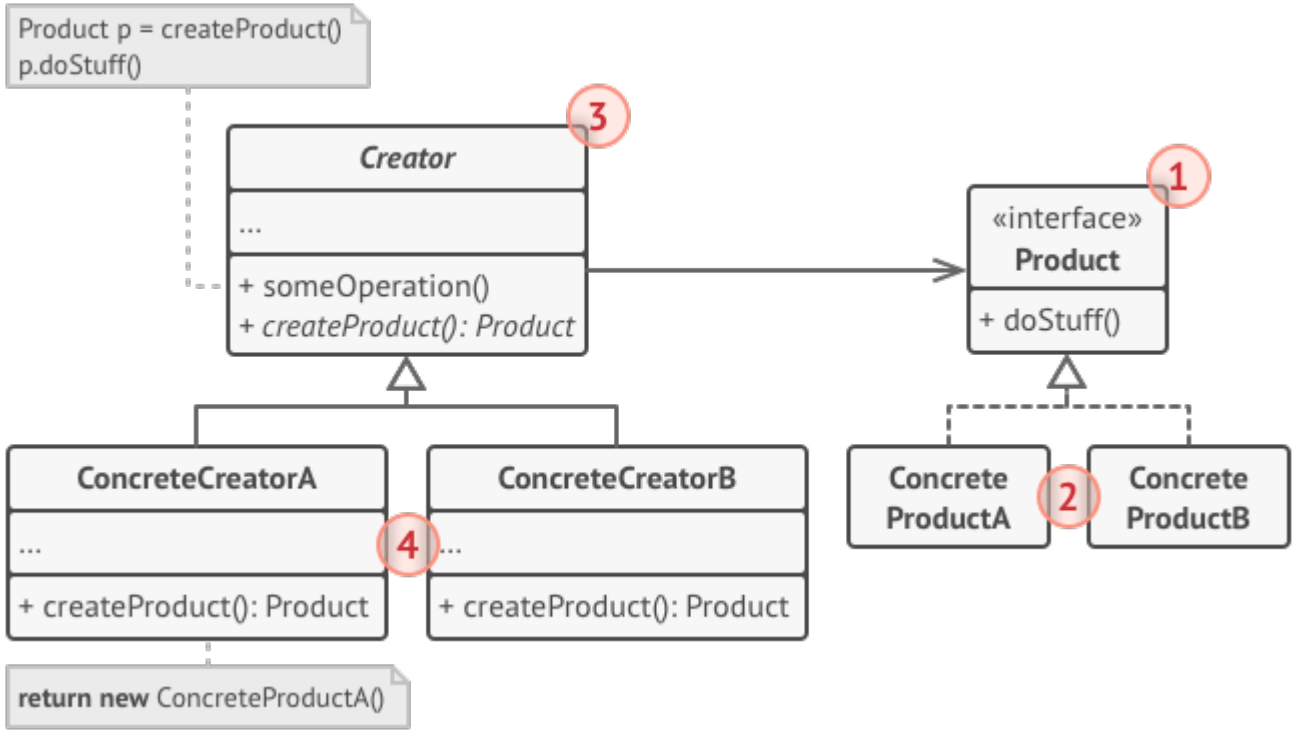






FACTORY METHOD

Structure:

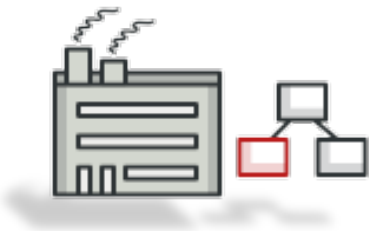


1.Product declares the interface, which is common to all objects that can be produced by the creator and its subclasses.

2.Concrete Products are different implementations of the product interface.

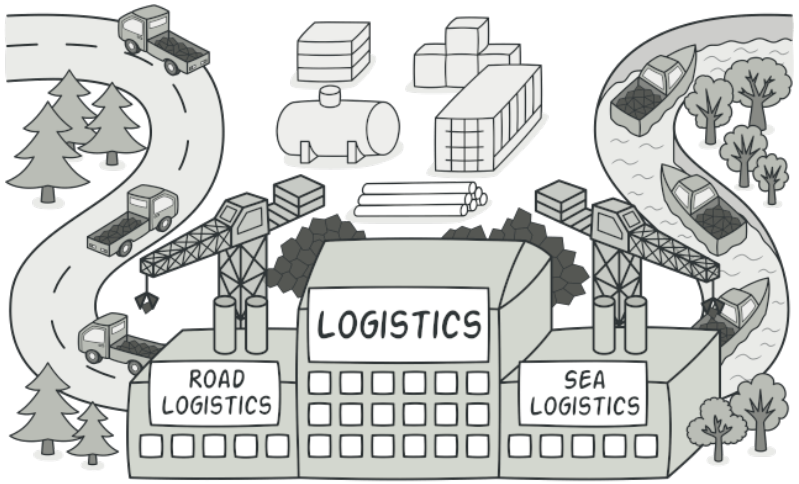
3.The **Creator** class declares the factory method that returns new product objects. It’s important that the return type of this method matches the product interface.

4.Concrete Creators override the base factory method, so it returns a different type of product.



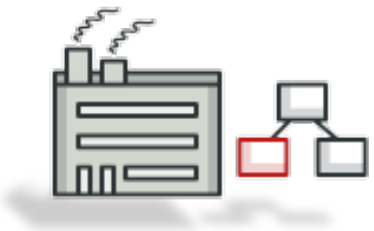
EXAMPLE : FACTORY METHOD #1

- What classes does it consist of?
- What roles do these classes play?
- In what way the elements of the pattern are related?:




```
//Abscract class
6 references
abstract class Vehicle
{
    VehicleType vehicleType;
    int topSpeed;
    int capacity;
    int price;
    2 references
    public VehicleType VehicleType
    {
        get { return vehicleType; }
        set { vehicleType = value; }
    }
    4 references
    public int TopSpeed
    {
        get { return topSpeed; }
        set { topSpeed = value; }
    }
    2 references
    public int Price
    {
        get { return price; }
        set { price = value; }
    }
    4 references
    public int Capacity
    {
        get { return capacity; }
        set { capacity = value; }
    }
}
```

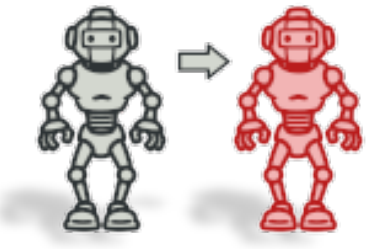
```
public enum VehicleType
{
    BIKE,
    SCOTTER,
    CAR
}
//Concrete Product
2 references
class Bike : Vehicle
{
    1 reference
    public Bike() : base()
    {
        VehicleType = VehicleType.BIKE;
        TopSpeed = 200;
        Capacity = 4;
        Price = 150000;
    }
}
2 references
class Scotter : Vehicle
{
    1 reference
    public Scotter(): base()
    {
        VehicleType = VehicleType.SCOTTER;
        TopSpeed = 100;
        Capacity = 2;
        Price = 120000;
    }
}
```



EXAMPLE : FACTORY METHOD #2

```
//Creator
1 reference
abstract class VehicleCreationFactory
{
    3 references
    public abstract Vehicle GetVehicle(VehicleType vehicleType);
}
//Concrete Creator
2 references
class VehicleFactory : VehicleCreationFactory
{
    3 references
    public override Vehicle GetVehicle(VehicleType vehicleType)
    {
        switch (vehicleType)
        {
            case VehicleType.BIKE:
                return new Bike();
            case VehicleType.SCOTTER:
                return new Scotter();
            default:
                throw new NotImplementedException();

        }

    }
}
```

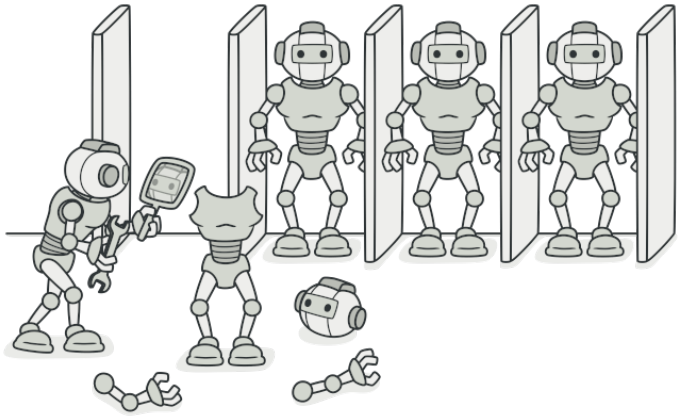
```
//Client
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        VehicleFactory factory = new VehicleFactory();
        Vehicle bike = factory.GetVehicle(VehicleType.BIKE);
        Console.WriteLine("Bike data: ");
        Console.WriteLine("TopSpeed: {0}, Capacity: {1}",
            bike.TopSpeed.ToString(),
            bike.Capacity.ToString());

        Vehicle scotter = factory.GetVehicle(VehicleType.SCOTTER);
        Console.WriteLine("Scotter data: ");
        Console.WriteLine("TopSpeed: {0}, Capacity: {1}",
            scotter.TopSpeed.ToString(),
            scotter.Capacity.ToString());

    }
}
```

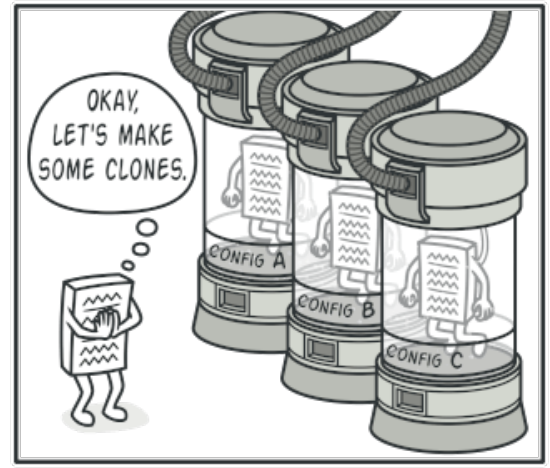


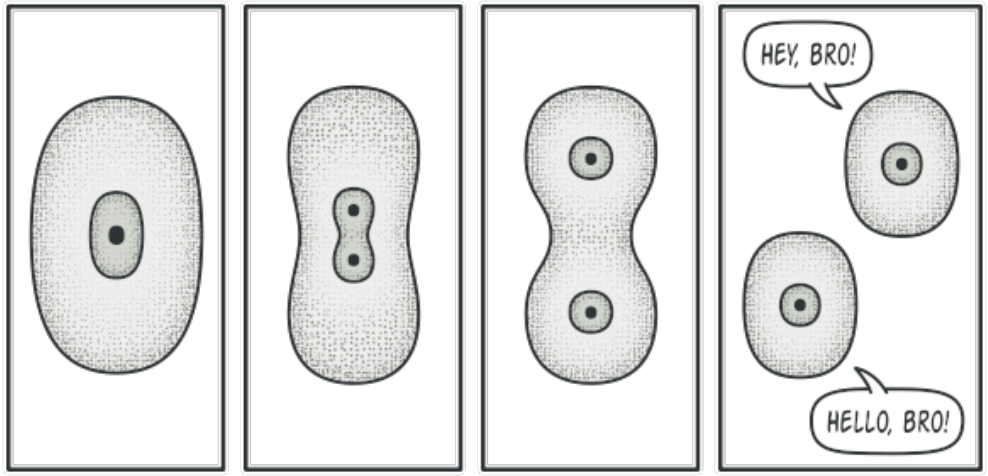
PROTOTYPE

is a creational design pattern that copy existing objects **without** making your code dependent on their classes.

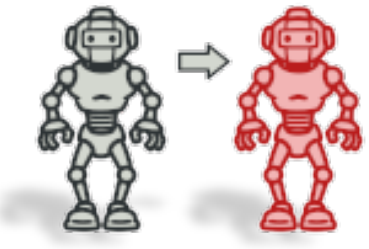
*Copying an object "from the outside" **isn't** always possible.*





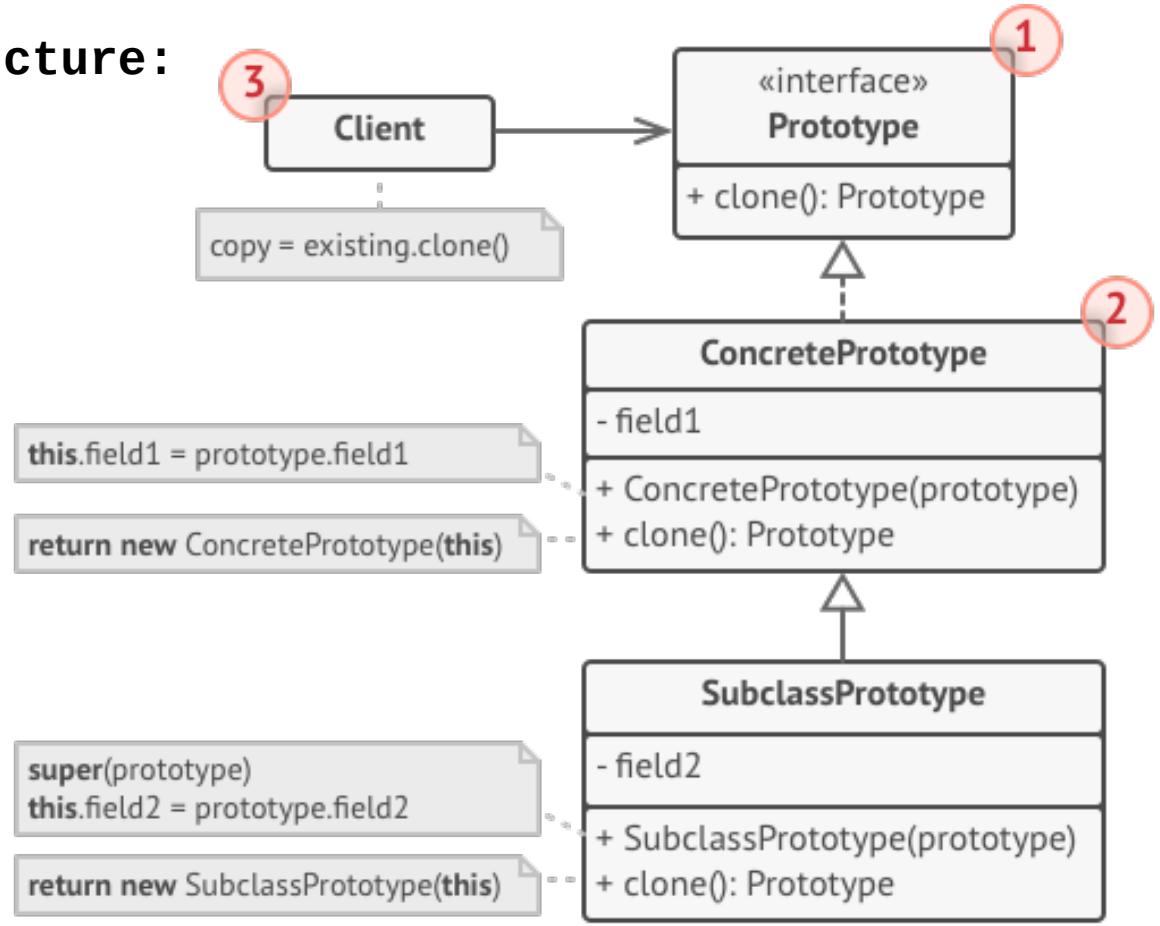


The division of a cell.



PROTOTYPE

Structure:

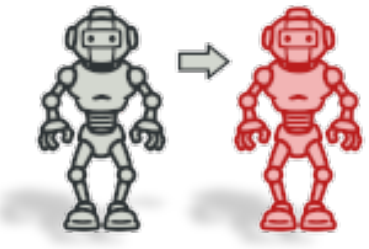


1.Prototype interface declares the cloning methods. In most cases, it’s a single clone method.

2.Concrete Prototype class implements the cloning method. In addition to copying the original object’s data to the clone, this method may also handle some edge cases of the cloning process related to cloning linked objects, untangling recursive dependencies, etc.

3.Client can produce a copy of any object that follows the prototype interface.

The division of a cell.



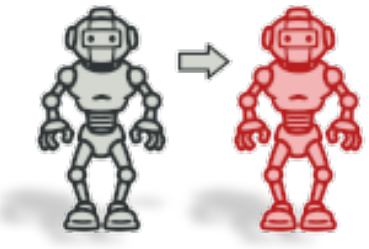
EXAMPLE : PROTOTYPE #1

- What classes does it consist of?
- What roles do these classes play?
- In what way the elements of the pattern are related?:

```
public abstract class Enemy
{
    int m_Health;
    int m_Attck;
    double m_Damage;
    3 references
    public int Health
    {
        get { return m_Health; }
        set { m_Health = value; }
    }
    3 references
    public int Attack
    {
        get { return m_Attck; }
        set { m_Attck = value; }
    }
    3 references
    public double Damage
    {
        get { return m_Damage; }
        set { m_Damage = value; }
    }
    2 references
    public abstract Enemy Clone();
}
```

```
public class Elephant : Enemy
{
    2 references
    public override Enemy Clone()
    {
        return this.MemberwiseClone() as Enemy;
    }
}
```

```
class Program
{
    0 references
    static void Main(string[] args)
    {
        Elephant elephant = new Elephant();
        elephant.Health = 100;
        elephant.Attack = 10;
        elephant.Damage = 5.0;

        Console.WriteLine("Original Enemy stats:");
        Console.WriteLine("Health: {0}, Attack: {1}, Damage: {2}",
        elephant.Health.ToString(),
        elephant.Attack.ToString(),
        elephant.Damage.ToString());

        Elephant playerToSave = elephant.Clone() as Elephant;
        Console.WriteLine("\nCopy of Enemy to save on disk:");
        Console.WriteLine("Health: {0}, Attack: {1}, Damage: {2}",
        playerToSave.Health.ToString(),
        playerToSave.Attack.ToString(),
        playerToSave.Damage.ToString());


    }
}
```

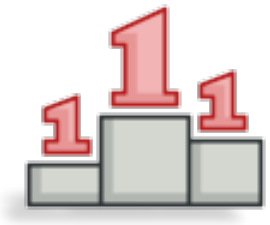


EXAMPLE : PROTOTYPE #2

```
public class EnemeyTrait
{
    int boostTime;
    int boostDamage;
    0 references
    public int BoostTime
    {
        get {return boostTime;}
        set {boostTime = value;}
    }
    0 references
    public int BoostDamage
    {
        get {return boostDamage;}
        set {boostDamage = value;}
    }
}
```

```
public abstract class Enemy
{
    int m_Health;
    int m_Attck;
    double m_Damage;
    //add more
    EnemeyTrait m_EnemyTrait = new EnemeyTrait();

    0 references
    public EnemeyTrait EnemeyTrait
    {
        get {return m_EnemyTrait; }
        set {m_EnemyTrait = value; }
    }
    3 references
    public int Health
    {
        get { return m_Health; }
        set { m_Health = value; }
    }
    3 references
    public int Attack
    {
        get { return m_Attck; }
        set { m_Attck = value; }
    }
    3 references
    public double Damage
    {
        get { return m_Damage; }
        set { m_Damage = value; }
    }

    2 references
    public abstract Enemy Clone();
}
```

```
public class Elephant : Enemy
{
    2 references
    public override Enemy Clone()
    {
        Elephant elephant = this.MemberwiseClone() as Elephant;
        elephant.EnemeyTrait = new EnemeyTrait();
        elephant.EnemeyTrait.BoostDamage = this.EnemeyTrait.BoostDamage;
        elephant.EnemeyTrait.BoostTime = this.EnemeyTrait.BoostTime;
        return elephant as Enemy;
    }
}

class Program
{
    0 references
    static void Main(string[] args)
    {
        Elephant elephant = new Elephant();
        elephant.Health = 100;
        elephant.Attack = 10;
        elephant.Damage = 5.0;
        //add more
        elephant.EnemeyTrait.BoostTime = 7;
        elephant.EnemeyTrait.BoostDamage = 7;


        Elephant playerToSave = elephant.Clone() as Elephant;
        playerToSave.EnemeyTrait.BoostTime = 10;
        playerToSave.EnemeyTrait.BoostDamage = 10;

        Console.WriteLine("Original Enemy stats:");
        Console.WriteLine("BoostTime: {0}, BoostDamage: {1}",
        elephant.EnemeyTrait.BoostTime.ToString(),
        elephant.EnemeyTrait.BoostDamage.ToString());

        Console.WriteLine("Copied Object Value:");
        Console.WriteLine("BoostTime: {0}, BoostDamage: {1}",
        playerToSave.EnemeyTrait.BoostTime.ToString(),
        playerToSave.EnemeyTrait.BoostDamage.ToString());
    }
}
```

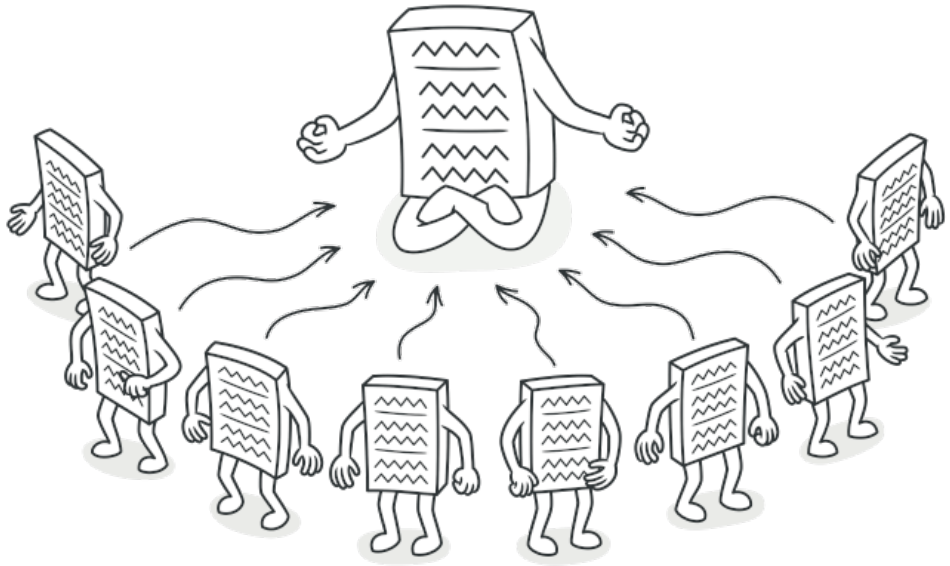


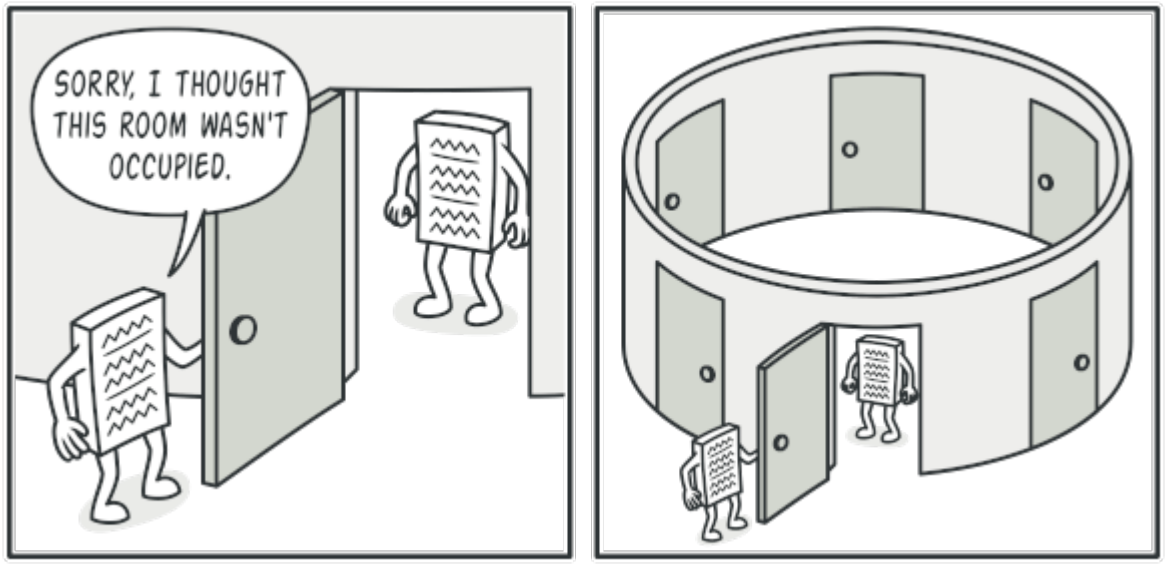
SINGLETON

is a creational design pattern, which ensures that only **one object** of its kind exists and provides **a single point of access** to it for any other code.

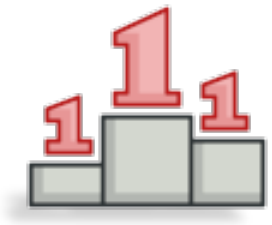
Single Responsibility Principle:

- **Ensure that a class has just a single instance.**
- **Provide a global access point to that instance.**



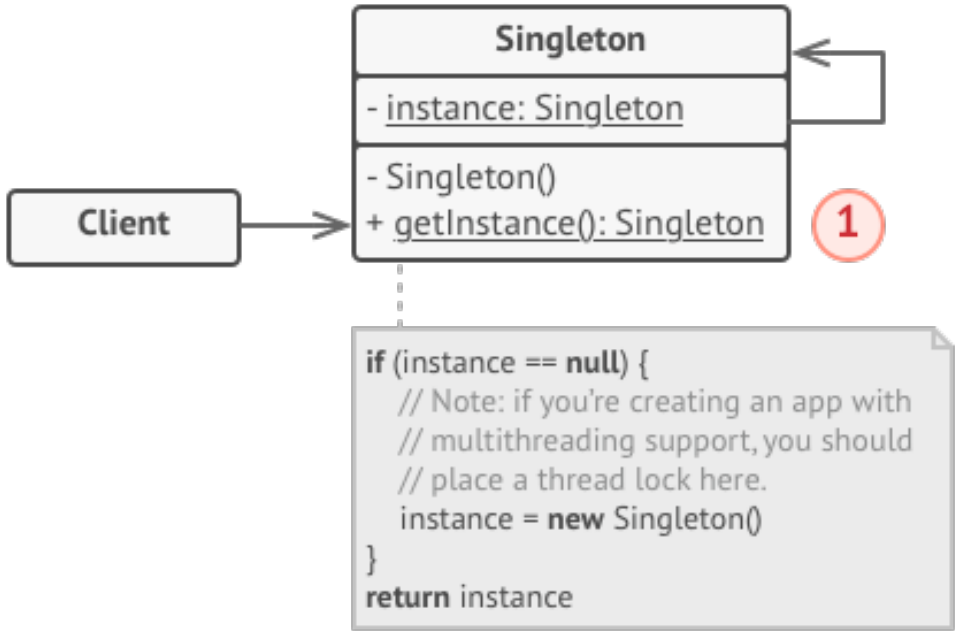


Clients may not even realize that they're working with the same object all the time.



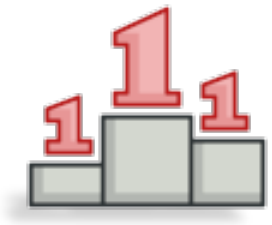
SINGLETON

Structure:



1.Singleton class declares the static method *getInstance* that returns the same instance of its own class.

2.Singleton's constructor should be hidden from the client code. Calling the *getInstance* method should be the only way of getting the Singleton object.



EXAMPLE : SINGLETON

Naïve Singleton

- Need to hide constructor and implement a static creation method.
- The same class behaves incorrectly in a multithreaded environment.
- Multiple threads can call the creation method simultaneously and get several instances of Singleton class.

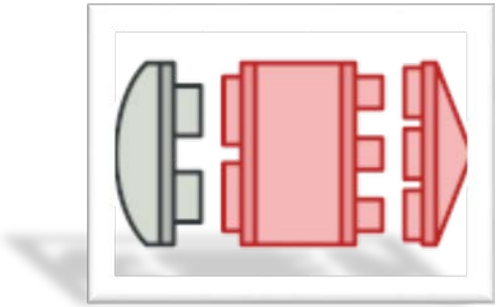
```
class Singleton
{
    1 reference
    private Singleton()
    {

    }

    private static Singleton _instance;

    2 references
    public static Singleton GetInstance()
    {
        if(_instance == null)
        {
            _instance = new Singleton();
        }
        return _instance;
    }

    0 references
    public static void someBusinessLogic()
    {

    }
}
```

```
class Program
{
    0 references
    static void Main(string[] args)
    {
        Singleton s1 = Singleton.GetInstance();
        Singleton s2 = Singleton.GetInstance();
        if (s1 == s2)
        {
            Console.WriteLine("Singleton works, both variables contain the same instance.");
        } else {
            Console.WriteLine("Singleton failed, variables contain different instances.");
        }
    }
}
```

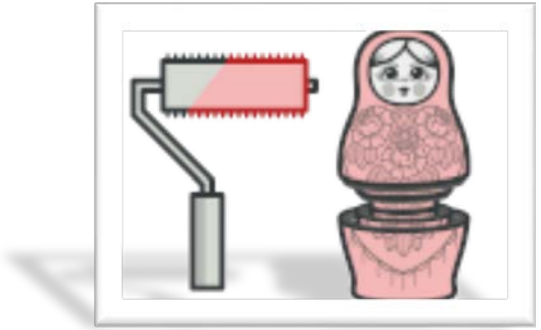
More : https://csharpindepth.com/Articles/Singleton

STRUCTURAL PATTERNS

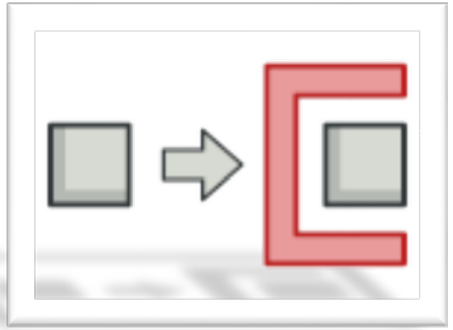
- Structural patterns explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient.



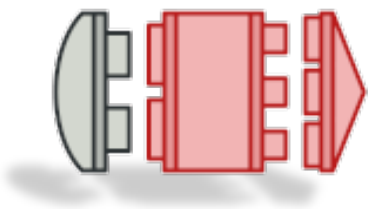
Adapter



Decorator

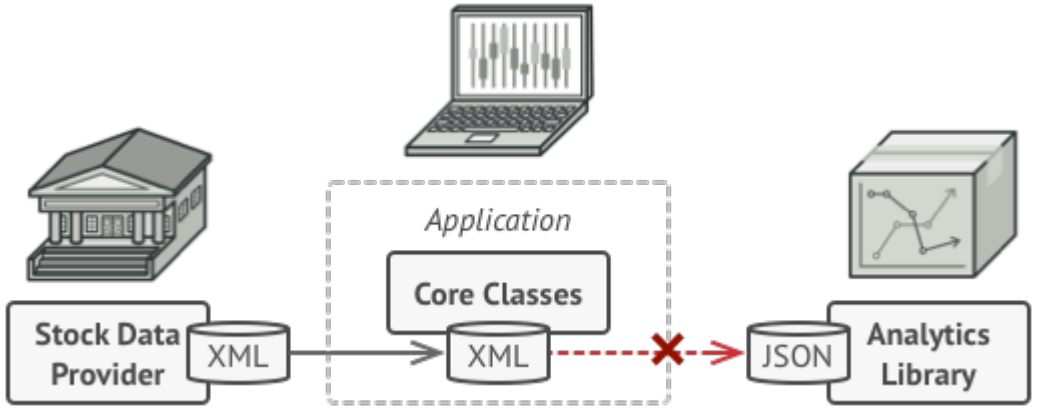


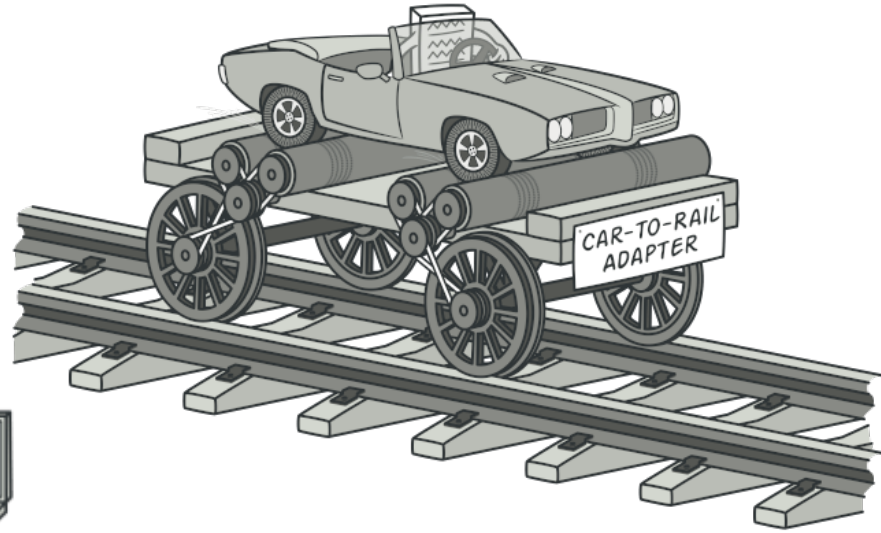
Proxy

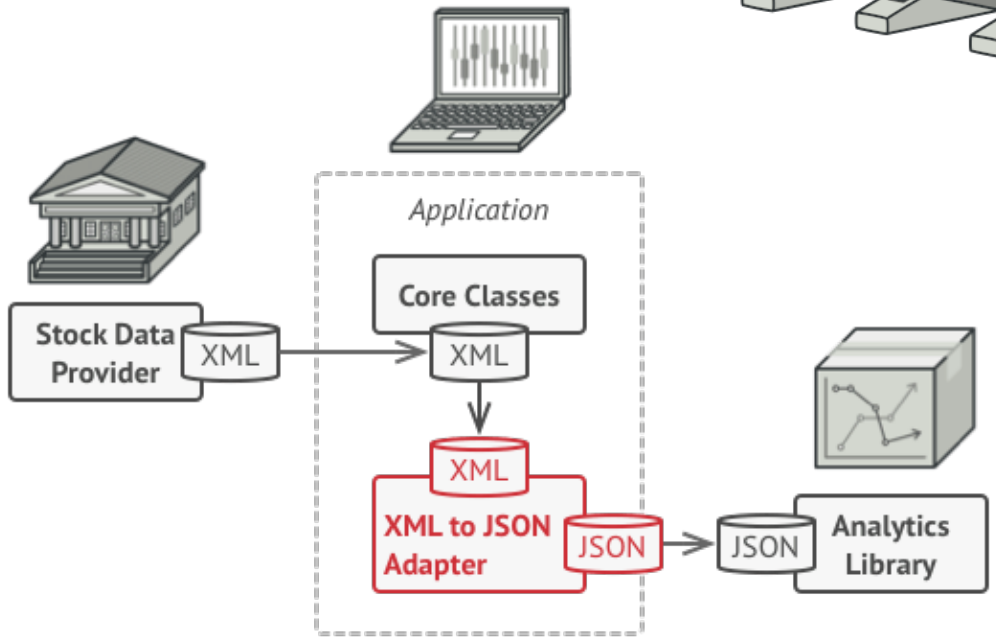


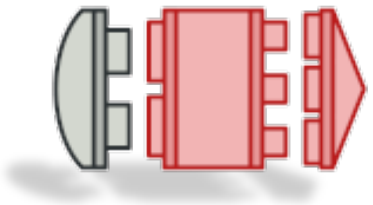
ADAPTER

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



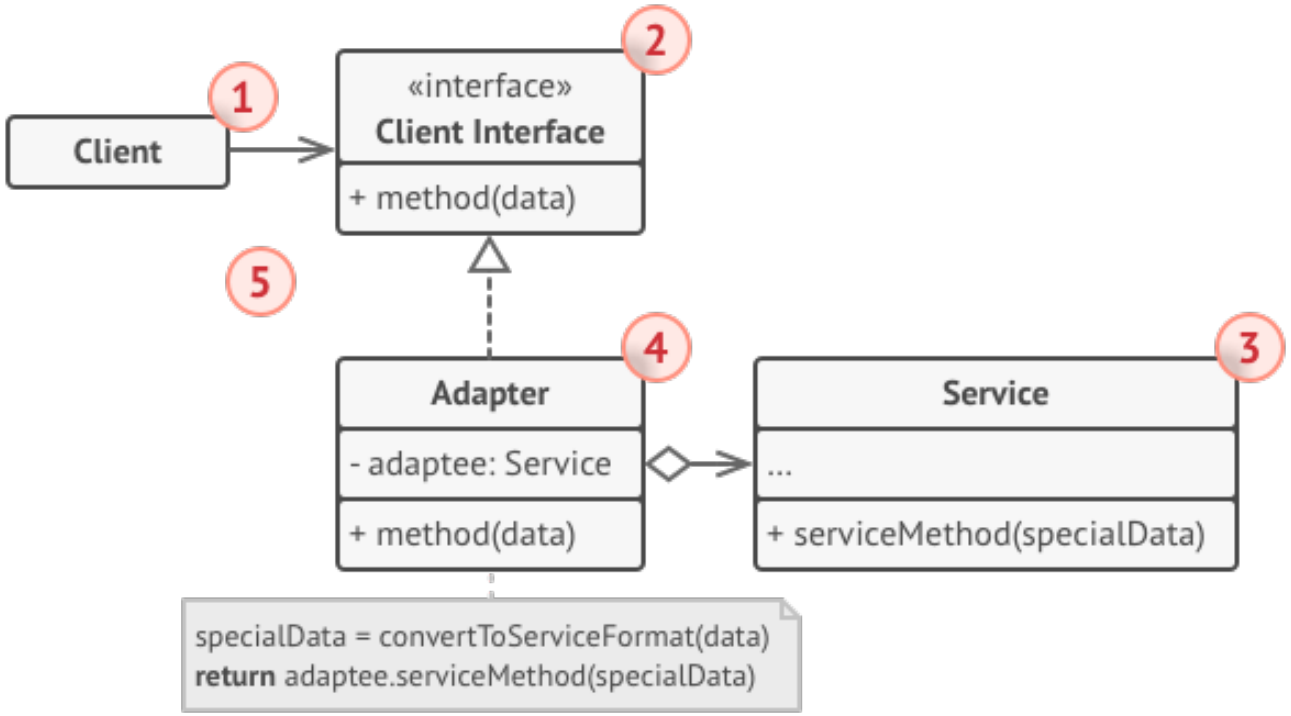






ADAPTER

Structure:

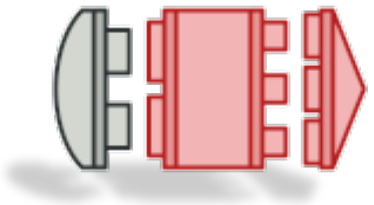


1.Client is a class that contains the existing business logic of the program.

2.Client Interface describes a protocol that other classes must follow to be able to collaborate with the client code.

3.Service is some useful class (usually 3rd-party or legacy). The client can’t use this class directly because it has an incompatible interface.

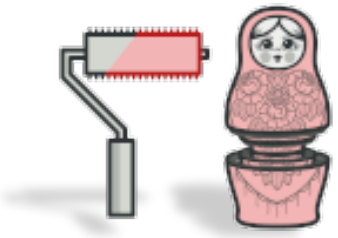
4.Adapter is a class that’s able to work with both the client and the service



EXAMPLE : ADAPTER

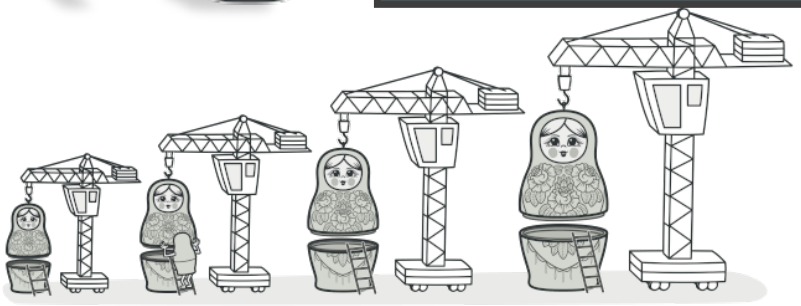
```
public interface Target
{
    2 references
    string GetRequest();
}
4 references
class Adaptee
{
    1 reference
    public string GetSpecificRequest()
    {
        return "Specific Request";
    }
}
2 references
class Adapter : Target
{
    private readonly Adaptee _adaptee;

    1 reference
    public Adapter(Adaptee adaptee)
    {
        this._adaptee = adaptee;
    }
    2 references
    public string GetRequest()
    {
        return $"This is '{this._adaptee.GetSpecificRequest()}'";
    }
}
```

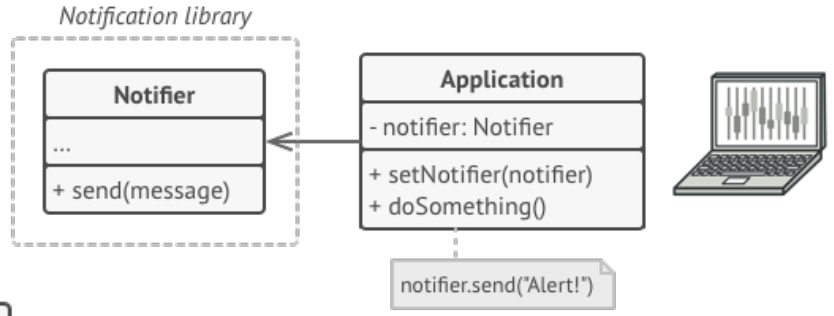
```
class Program
{
    0 references
    static void Main(string[] args)
    {
        Adaptee adaptee = new Adaptee();
        Target target = new Adapter(adaptee);
        Console.WriteLine("Adaptee interface is incompatible with the client.");
        Console.WriteLine("But with adapter client can call it's method.");
        Console.WriteLine(target.GetRequest());
    }
}
```

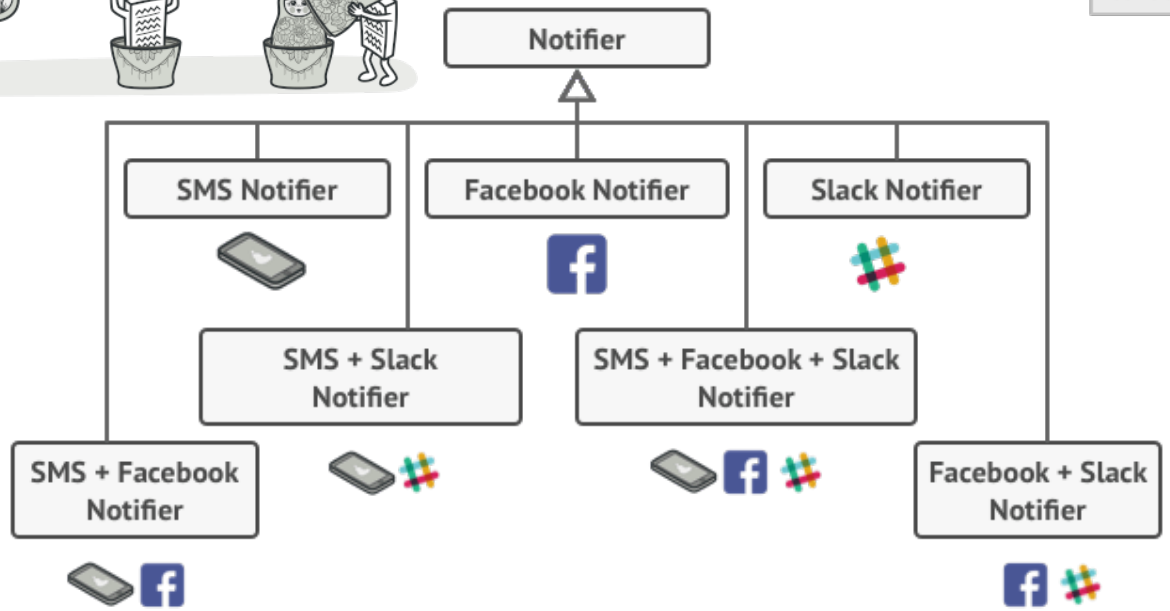


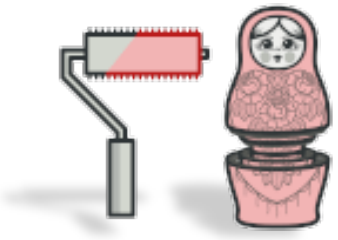
DECORATOR

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

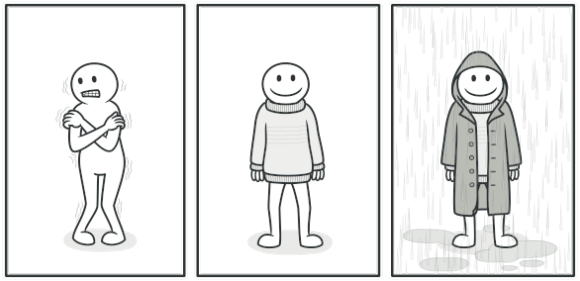






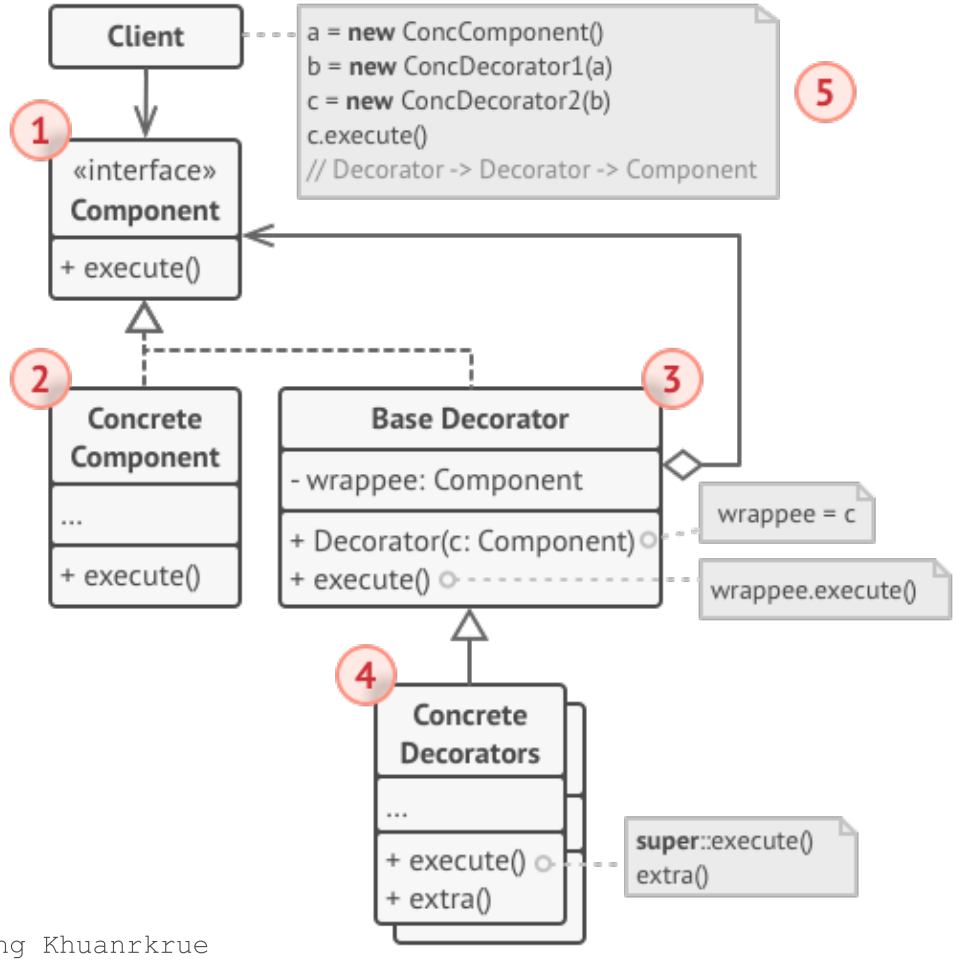


DECORATOR

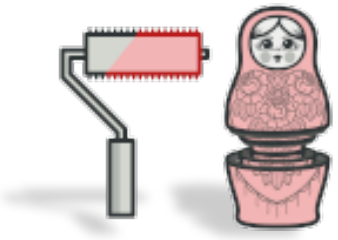


combined effect from wearing multiple pieces of clothing.

Structure :



1.Component declares the common interface for both wrappers and wrapped objects.
2.Concrete Component is a class of objects being wrapped. It defines the basic behavior, which can be altered by decorators.
3.Base Decorator class has a field for referencing a wrapped object.
4.Concrete Decorators define extra behaviors that can be added to components dynamically.
5.Client can wrap components in multiple layers of decorators,



EXAMPLE : DECORATOR

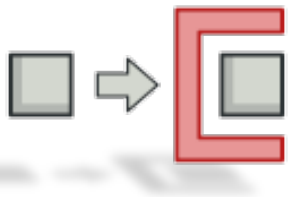
- What classes does it consist of?
- What roles do these classes play?
- In what way the elements of the pattern are related?:

```
// Component interface
8 references
public abstract class Component
{
    8 references
    public abstract string Operation();
}
1 reference
class ConcreteComponent : Component
{
    8 references
    public override string Operation()
    {
        return "ConcreteComponent";
    }
}
```

```
//Decorator class
3 references
abstract class Decorator : Component
{
    protected Component _component;
    2 references
    public Decorator(Component component)
    {
        this._component = component;
    }
    0 references
    public void SetComponent(Component component)
    {
        this._component = component;
    }
    8 references
    public override string Operation()
    {
        if (this._component != null)
        {
            return this._component.Operation();
        }
        else
        {
            return string.Empty;
        }
    }
}
```

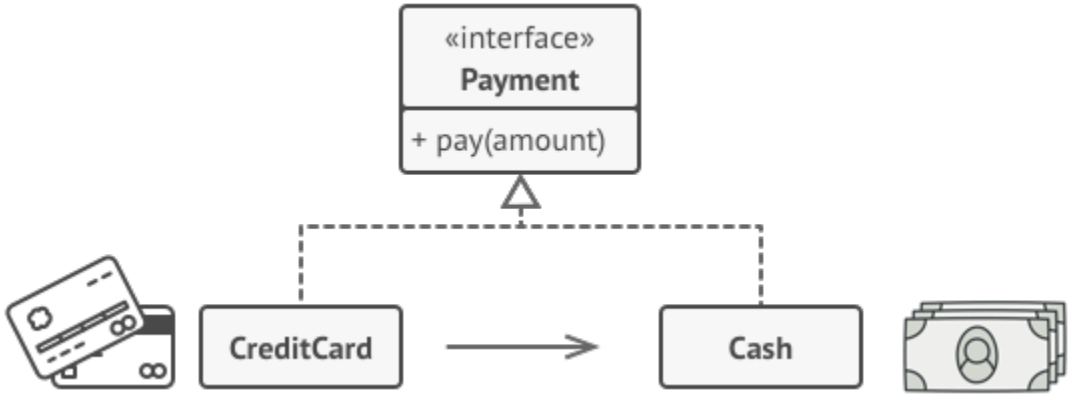
```
//Concrete Decorators
3 references
class ConcreteDecoratorA : Decorator
{
    1 reference
    public ConcreteDecoratorA(Component comp) : base(comp)
    {

    }
    8 references
    public override string Operation()
    {
        return $"ConcreteDecoratorA({base.Operation()})";
    }
}
3 references
class ConcreteDecoratorB : Decorator
{
    1 reference
    public ConcreteDecoratorB(Component comp) : base(comp)
    {

    }
    8 references
    public override string Operation()
    {
        return $"ConcreteDecoratorB({base.Operation()})";
    }
}
//Client
2 references
public class Client
{
    2 references
    public void ClientCode(Component component)
    {
        Console.WriteLine("RESULT: " + component.Operation());
    }
}
```

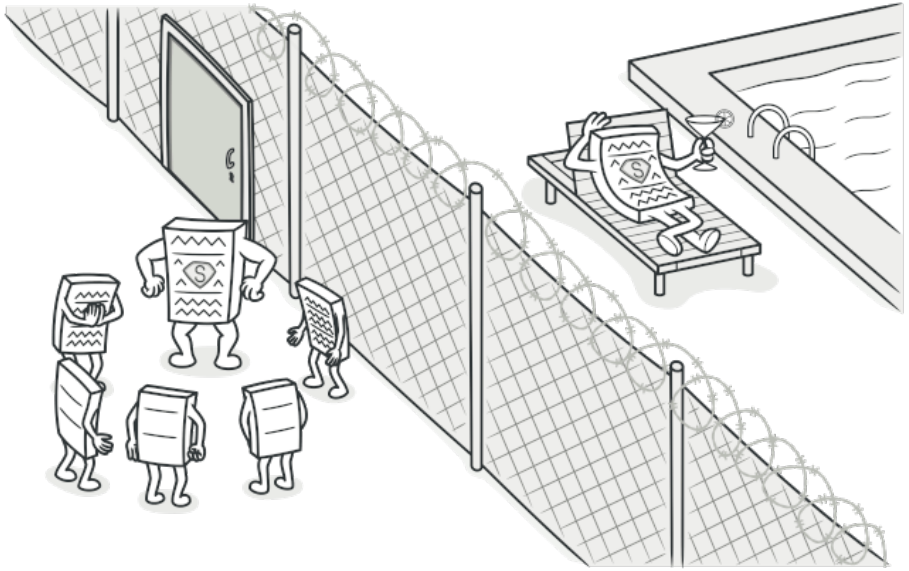
```
class Program
{
    0 references
    static void Main(string[] args)
    {
        Client client = new Client();
        var simple = new ConcreteComponent();
        Console.WriteLine("Client: I get a simple component:");
        client.ClientCode(simple);
        Console.WriteLine();

        ConcreteDecoratorA decorator1 = new ConcreteDecoratorA(simple);
        ConcreteDecoratorB decorator2 = new ConcreteDecoratorB(decorator1);
        Console.WriteLine("Client: Now I've got a decorated component:");
        client.ClientCode(decorator2);
    }
}
```

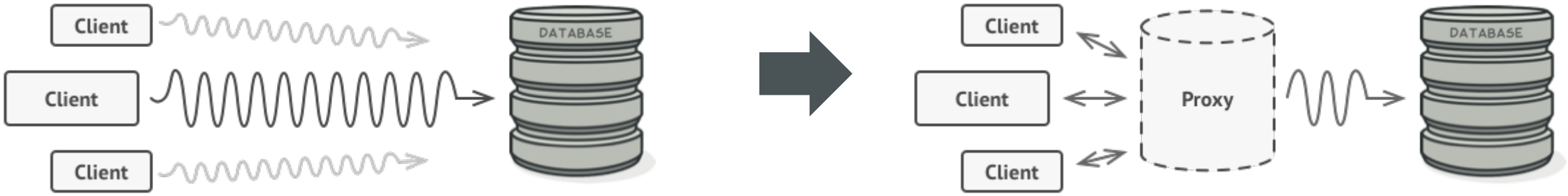


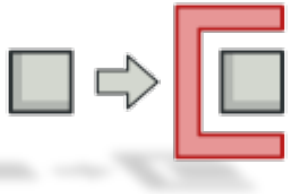
PROXY

is a structural design pattern that **provide a substitute or placeholder** for another object.



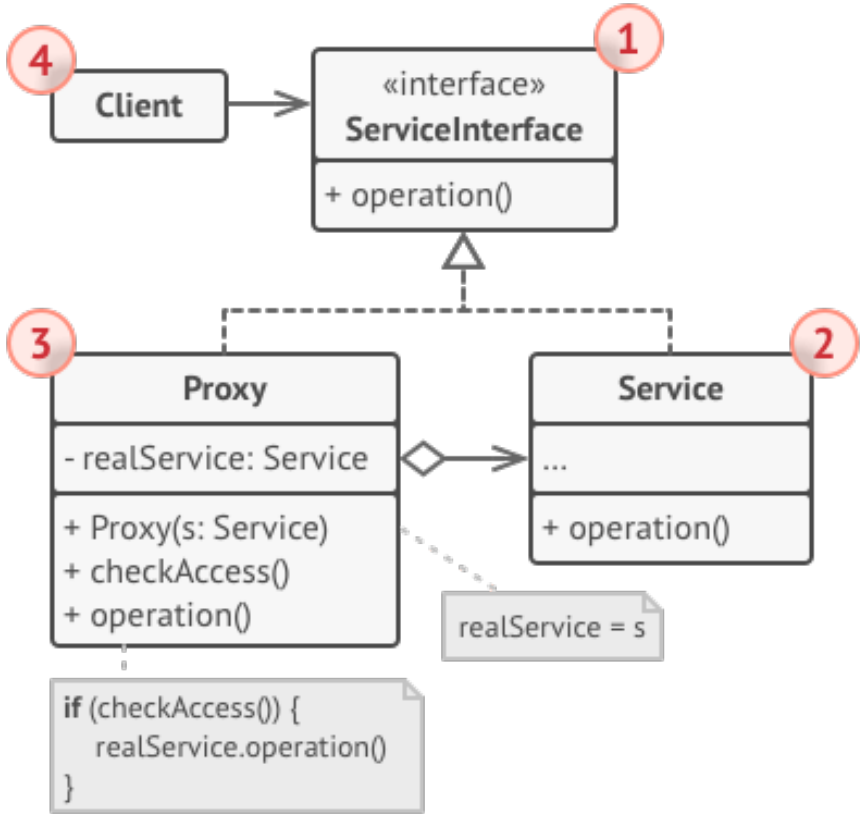






PROXY

Structure :

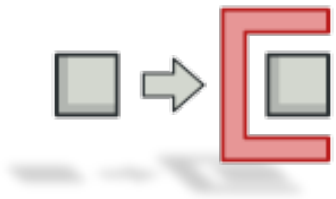


1.Service Interface declares the interface of the Service. The proxy must follow this interface to be able to disguise itself as a service object.

2.Service is a class that provides some useful business logic.

3.Proxy class has a reference field that points to a service object.

4.Client should work with both services and proxies via the same interface. This way you can pass a proxy into any code that expects a service object.



EXAMPLE : PROXY

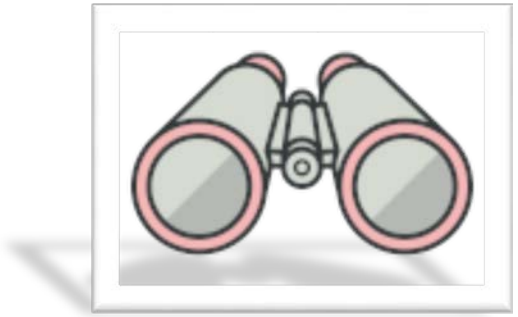
```
// The Subject interface declares common operations
3 references
public interface Subject
{
    4 references
    void Request();
}
5 references
class RealSubject : Subject
{
    4 references
    public void Request()
    {
        Console.WriteLine("RealSubject: Handling Request.");
    }
}
```

```
// The Proxy has an interface
3 references
class Proxy : Subject
{
    private RealSubject _realSubject;

    1 reference
    public Proxy(RealSubject realSubject)
    {
        this._realSubject = realSubject;
    }
    4 references
    public void Request()
    {
        if (this.CheckAccess())
        {
            this._realSubject = new RealSubject();
            this._realSubject.Request();

            this.LogAccess();
        }
    }
    1 reference
    public bool CheckAccess()
    {
        Console.WriteLine("Proxy: Checking access prior to firing a real request.");
        return true;
    }
    1 reference
    public void LogAccess()
    {
        Console.WriteLine("Proxy: Logging the time of request.");
    }
}
```

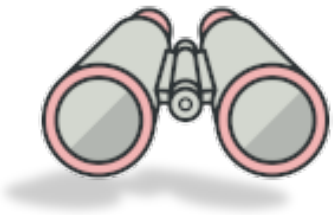
```
public class Client
{
    2 references
    public void ClientCode(Subject subject)
    {
        subject.Request();
    }
}
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        Client client = new Client();

        Console.WriteLine("Client: Executing the client code with a real subject:");
        RealSubject realSubject = new RealSubject();
        client.ClientCode(realSubject);

        Console.WriteLine();

        Console.WriteLine("Client: Executing the same client code with a proxy:");
        Proxy proxy = new Proxy(realSubject);
        client.ClientCode(proxy);
    }
}
```

BEHAVIORAL PATTERNS

- Behavioral design patterns are concerned with algorithms and the assignment of responsibilities between objects.

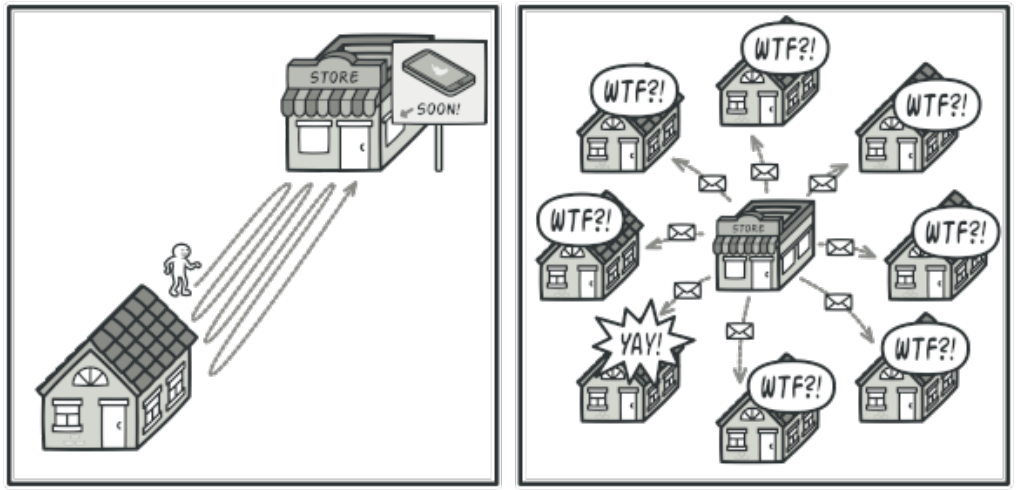


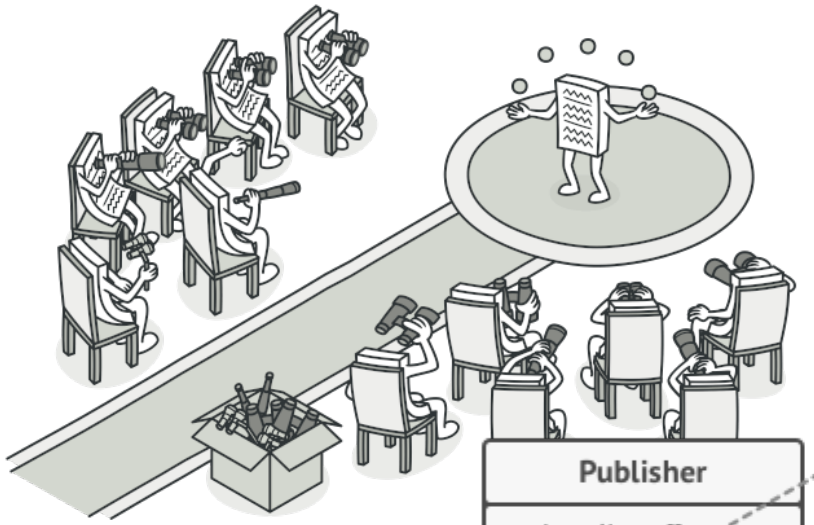
Observer

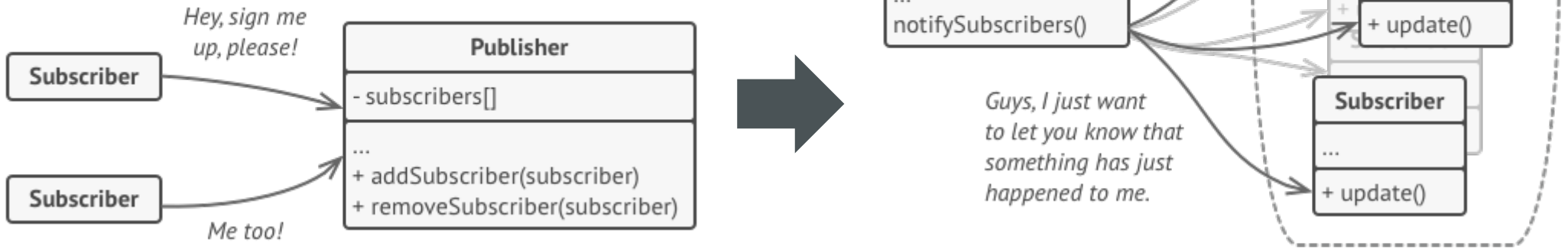


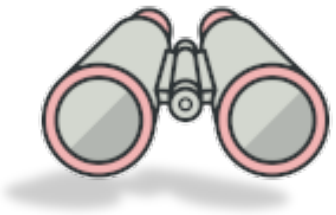
OBSERVER

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.



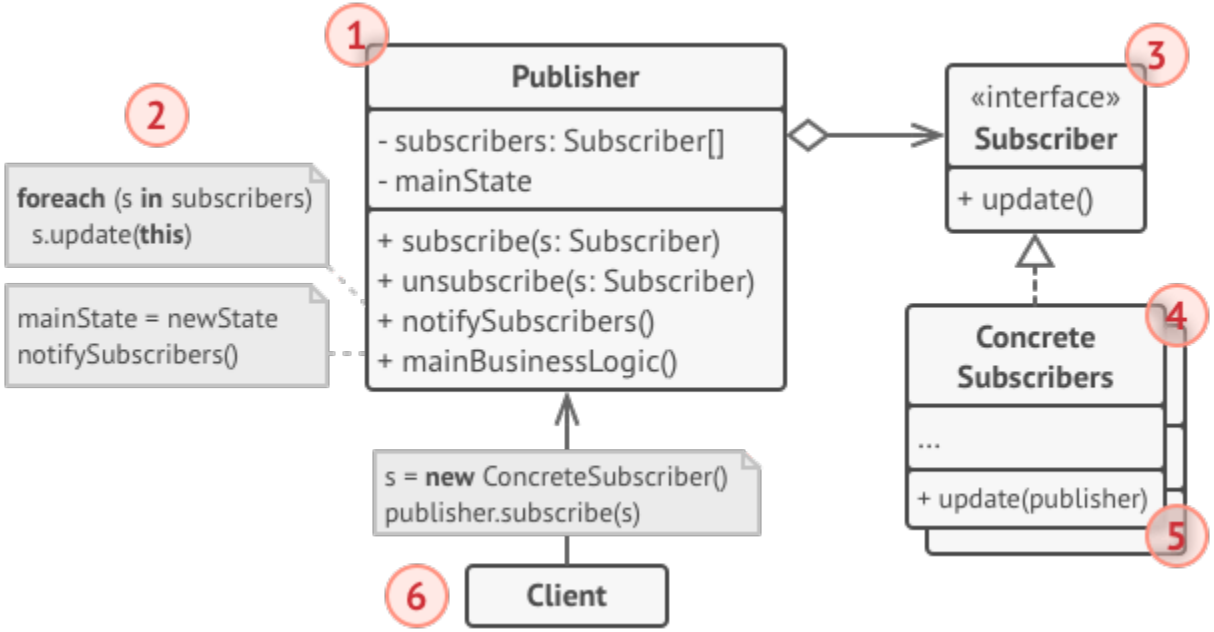




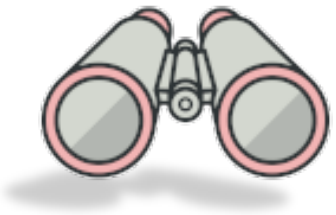


OBSERVER

Structure:



1.Publisher issues events of interest to other objects.
2.the publisher goes over the subscription list and calls the notification method.
3.Subscriber interface declares the notification interface. In most cases, it consists of a single update method.
4.Concrete Subscribers perform some actions in response to notifications issued by the publisher.
5.subscribers need some contextual information to handle the update correctly.
6.Client creates publisher and subscriber objects separately and then registers subscribers for publisher updates.

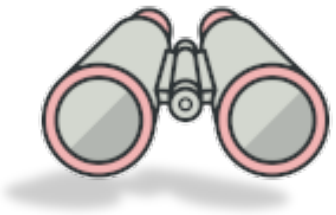


EXAMPLE : OBSERVER #1

```
using System;
using System.Collections.Generic;
using System.Threading;
```

```
public interface Observer
{
    3 references
    void Update(ISubject subject);
}
4 references
public interface ISubject
{
    3 references
    void Attach(Observer observer);
    2 references
    void Detach(Observer observer);
    2 references
    void Notify();
}
```

```
// The Subject
4 references
public class Subject : ISubject
{
    5 references
    public int State { get; set; } = -0;
    // List of subscribers.
    private List<Observer> _observers = new List<Observer>();

    // The subscription management methods.
    3 references
    public void Attach(Observer observer)
    {
        Console.WriteLine("Subject: Attached an observer.");
        this._observers.Add(observer);
    }
    2 references
    public void Detach(Observer observer)
    {
        this._observers.Remove(observer);
        Console.WriteLine("Subject: Detached an observer.");
    }
    // Trigger an update in each subscriber.
    2 references
    public void Notify()
    {
        Console.WriteLine("Subject: Notifying observers...");

        foreach (var observer in _observers)
        {
            observer.Update(this);
        }
    }
    3 references
    public void SomeBusinessLogic()
    {
        Console.WriteLine("\nSubject: I'm doing something important.");
        this.State = new Random().Next(0, 10);

        Thread.Sleep(15);

        Console.WriteLine("Subject: My state has just changed to: " + this.State);
        this.Notify();
    }
}
```



EXAMPLE : OBSERVER #2

```
// Concrete Observers react to the updates
1 reference
class ConcreteObserverA : Observer
{
    3 references
    public void Update(ISubject subject)
    {
        if ((subject as Subject).State < 3)
        {
            Console.WriteLine("ConcreteObserverA: Reacted to the event.");
        }
    }
}

1 reference
class ConcreteObserverB : Observer
{
    3 references
    public void Update(ISubject subject)
    {
        if ((subject as Subject).State == 0 || (subject as Subject).State >= 2)
        {
            Console.WriteLine("ConcreteObserverB: Reacted to the event.");
        }
    }
}
```

```
class Program
{
    0 references
    static void Main(string[] args)
    {
        // The client
        var subject = new Subject();
        var observerA = new ConcreteObserverA();
        subject.Attach(observerA);

        var observerB = new ConcreteObserverB();
        subject.Attach(observerB);
        subject.SomeBusinessLogic();
        subject.SomeBusinessLogic();

        subject.Detach(observerB);
        subject.SomeBusinessLogic();
    }
}
```

SUMMARY

- Design patterns differ by their complexity, level of detail and scale of applicability to the entire system being designed.
- The most basic and low-level patterns are often called idioms. They usually apply only to a single programming language.
- Three main groups of patterns:
  - **Creational patterns** provide object creation mechanisms that increase flexibility and reuse of existing code.
  - **Structural patterns** explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient.
  - **Behavioral patterns** take care of effective communication and the assignment of responsibilities between objects.