Паттерны проектирования в C# и .NET

Паттерн представляет определенный способ построения программного кода для решения часто встречающихся проблем проектирования. В данном случае предполагается, что есть некоторый набор общих формализованных проблем, которые довольно часто встречаются, и паттерны предоставляют ряд принципов для решения этих проблем.

Структурные шаблоны проектирования. Эти шаблоны в основном посвящены компоновке объектов (object composition). То есть тому, как сущности могут друг друга использовать. Ещё одно объяснение: структурные шаблоны помогают ответить на вопрос «Как построить программный компонент?»

Поведенческие шаблоны проектирования определяют алгоритмы и способы реализации взаимодействия различных объектов и классов. Они обеспечивают гибкость взаимодействия между объектами.

Порождающими называют шаблоны, которые используют механизмы создания объектов, чтобы создавать объекты подходящим для данной ситуации способом. Базовый способ создания может привести к проблемам в архитектуре или к её усложнению. Порождающие шаблоны пытаются решать эти проблемы, управляя способом создания объектов.

Не стоит применять паттерны ради самих паттернов. Хорошая программа предполагает использование паттернов. Однако не всегда паттерны упрощают и улучшают программу. Паттерн должен быть оправданным и эффективным способом решения проблемы.

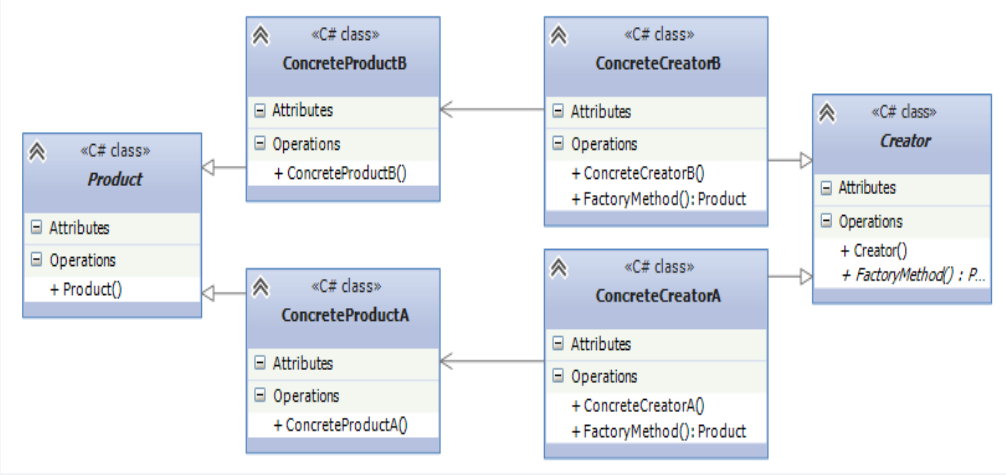
Порождающие паттерны

Factory Method

Фабричный метод (Factory Method) - это паттерн, который определяет интерфейс для создания объектов некоторого класса, но непосредственное решение о том, объект какого класса создавать происходит в подклассах.

Когда надо применять паттерн?

- **Когда заранее неизвестно, объекты каких типов необходимо создавать**
- **Когда система должна быть независимой от процесса создания новых объектов и расширяемой: в нее можно легко вводить новые классы, объекты которых система должна создавать.**
- **Когда создание новых объектов необходимо делегировать из базового класса классам наследникам**



Factory Method

Участники

- **Product** определяет интерфейс класса, объекты которого надо создавать
- **ConcreteProductA** и **ConcreteProductB** представляют реализацию класса Product. Таких классов может быть множество
- **Creator** определяет абстрактный фабричный метод FactoryMethod(), который возвращает объект Product.
- **ConcreteCreatorA** и **ConcreteCreatorB** - наследники класса Creator, определяющие свою реализацию метода FactoryMethod(). Причем метод FactoryMethod() каждого отдельного класса-создателя возвращает определенный конкретный тип продукта. Для каждого класса определяется свой конкретный класс создателя.

Factory Method (пример)

Product and concrete products

```
/// <summary>
/// Product
/// </summary>
abstract class Ingredient { }
```

```
/// <summary>
/// Concrete Product
/// </summary>
class Bread : Ingredient { }

/// <summary>
/// Concrete Product
/// </summary>
class Turkey : Ingredient { }

/// <summary>
/// Concrete Product
/// </summary>
class Lettuce : Ingredient { }

/// <summary>
/// Concrete Product
/// </summary>
class Mayonnaise : Ingredient { }
```

Creator

```
/// <summary>
/// Creator
/// </summary>
abstract class Sandwich
{
    private List<Ingredient> _ingredients = new List<Ingredient>();

    public Sandwich()
    {
        CreateIngredients();
    }

    //Factory method
    public abstract void CreateIngredients();

    public List<Ingredient> Ingredients
    {
        get { return _ingredients; }
    }
}
```

Factory Method (пример)

Concrete creators

```
/// <summary>
/// Concrete Creator
/// </summary>
class TurkeySandwich : Sandwich
{
    public override void CreateIngredients()
    {
        Ingredients.Add(new Bread());
        Ingredients.Add(new Mayonnaise());
        Ingredients.Add(new Lettuce());
        Ingredients.Add(new Turkey());
        Ingredients.Add(new Turkey());
        Ingredients.Add(new Bread());
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        var turkeySandwich = new TurkeySandwich();
        var dagwood = new Dagwood();
        //Do something with these sandwiches (like, say, eat them).
        ...
    }
}
```

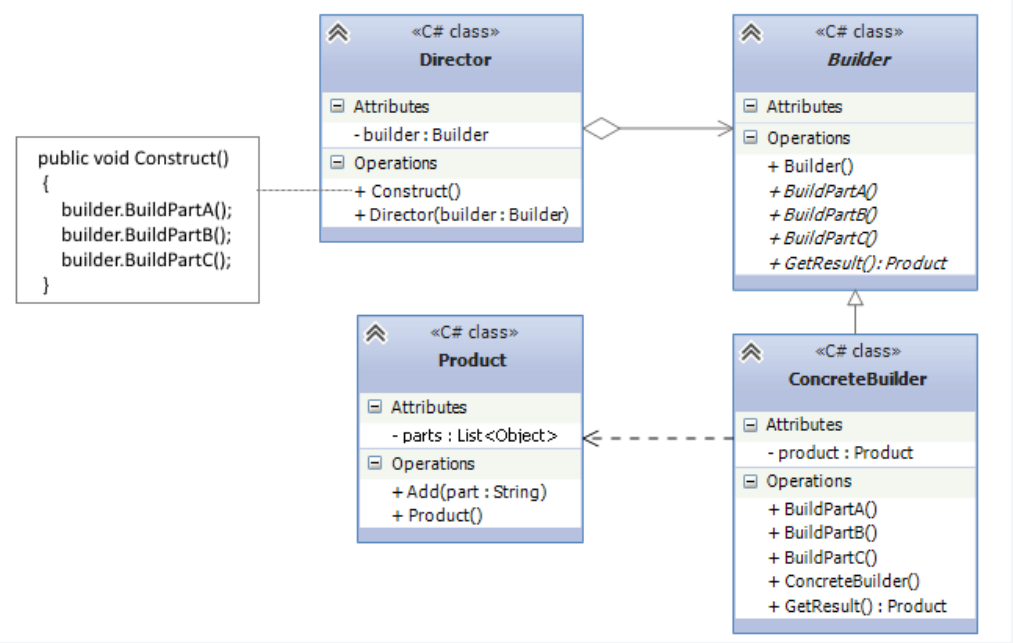
```
/// <summary>
/// Concrete Creator
/// </summary>
class Dagwood : Sandwich //OM NOM NOM
{
    public override void CreateIngredients()
    {
        Ingredients.Add(new Bread());
        Ingredients.Add(new Turkey());
        Ingredients.Add(new Turkey());
        Ingredients.Add(new Lettuce());
        Ingredients.Add(new Lettuce());
        Ingredients.Add(new Mayonnaise());
        Ingredients.Add(new Bread());
        Ingredients.Add(new Turkey());
        Ingredients.Add(new Turkey());
        Ingredients.Add(new Lettuce());
        Ingredients.Add(new Lettuce());
        Ingredients.Add(new Mayonnaise());
        Ingredients.Add(new Bread());
        Ingredients.Add(new Turkey());
        Ingredients.Add(new Turkey());
        Ingredients.Add(new Lettuce());
        Ingredients.Add(new Lettuce());
        Ingredients.Add(new Mayonnaise());
        Ingredients.Add(new Bread());
        Ingredients.Add(new Turkey());
        Ingredients.Add(new Turkey());
        Ingredients.Add(new Lettuce());
        Ingredients.Add(new Lettuce());
        Ingredients.Add(new Mayonnaise());
        Ingredients.Add(new Bread());
        Ingredients.Add(new Turkey());
        Ingredients.Add(new Turkey());
        Ingredients.Add(new Lettuce());
        Ingredients.Add(new Lettuce());
        Ingredients.Add(new Mayonnaise());
        Ingredients.Add(new Bread());
        Ingredients.Add(new Turkey());
        Ingredients.Add(new Turkey());
        Ingredients.Add(new Lettuce());
        Ingredients.Add(new Lettuce());
        Ingredients.Add(new Mayonnaise());
        Ingredients.Add(new Bread());
    }
}
```

Строитель (Builder)

Строитель(Builder) - шаблон проектирования, который инкапсулирует создание объекта и позволяет разделить его на различные этапы.

Когда надо применять паттерн?

- **Когда процесс создания нового объекта не должен зависеть от того, из каких частей этот объект состоит и как эти части связаны между собой**
- **Когда необходимо обеспечить получение различных вариаций объекта в процессе его**



Строитель (Builder)

Участники

- **Product** представляет объект, который должен быть создан
- **Builder** определяет интерфейс для создания различных частей объекта Product
- **ConcreteBuilder** конкретная реализация Builder. Создаёт объект Product и определяет интерфейс для доступа к нему
- **Director** - создаёт объект, используя объекты Builder

Строитель (пример)

Director

```
/// <summary>
/// The Director
/// </summary>
class AssemblyLine
{
    // Builder uses a complex series of steps
    //
    public void Assemble(SandwichBuilder sandwichBuilder)
    {
        sandwichBuilder.AddBread();
        sandwichBuilder.AddMeats();
        sandwichBuilder.AddCheese();
        sandwichBuilder.AddVeggies();
        sandwichBuilder.AddCondiments();
    }
}
```

Product

```
/// <summary>
/// The Product class
/// </summary>
class Sandwich
{
    private string _sandwichType;
    private Dictionary<string, string> _ingredients = new Dictionary<string,

    // Constructor
    public Sandwich(string sandwichType)
    {
        this._sandwichType = sandwichType;
    }

    // Indexer
    public string this[string key]
    {
        get { return _ingredients[key]; }
        set { _ingredients[key] = value; }
    }

    public void Show()
    {
        Console.WriteLine("\n---------------------------");
        Console.WriteLine("Sandwich: {0}", _sandwichType);
        Console.WriteLine(" Bread: {0}", _ingredients["bread"]);
        Console.WriteLine(" Meat: {0}", _ingredients["meat"]);
        Console.WriteLine(" Cheese: {0}", _ingredients["cheese"]);
        Console.WriteLine(" Veggies: {0}", _ingredients["veggies"]);
        Console.WriteLine(" Condiments: {0}", _ingredients["condiments"]);
    }
}
```

Строитель (пример)

ConcreteBuilders

```
/// <summary>
/// A ConcreteBuilder class
/// </summary>
class HamAndCheese : SandwichBuilder
{
    public HamAndCheese()
    {
        sandwich = new Sandwich("Ham and Cheese");
    }

    public override void AddBread()
    {
        sandwich["bread"] = "White";
    }

    public override void AddMeats()
    {
        sandwich["meat"] = "Ham";
    }

    public override void AddCheese()
    {
        sandwich["cheese"] = "American";
    }

    public override void AddVeggies()
    {
        sandwich["veggies"] = "None";
    }

    public override void AddCondiments()
    {
        sandwich["condiments"] = "Mayo";
    }
}
```

```
/// <summary>
/// A ConcreteBuilder class
/// </summary>
class TurkeyClub : SandwichBuilder
{
    public TurkeyClub()
    {
        sandwich = new Sandwich("Turkey Club");
    }

    public override void AddBread()
    {
        sandwich["bread"] = "12-Grain";
    }

    public override void AddMeats()
    {
        sandwich["meat"] = "Turkey";
    }

    public override void AddCheese()
    {
        sandwich["cheese"] = "Swiss";
    }

    public override void AddVeggies()
    {
        sandwich["veggies"] = "Lettuce, Tomato";
    }

    public override void AddCondiments()
    {
        sandwich["condiments"] = "Mayo";
    }
}
```

```
/// <summary>
/// A ConcreteBuilder class
/// </summary>
class BLT : SandwichBuilder
{
    public BLT()
    {
        sandwich = new Sandwich("BLT");
    }

    public override void AddBread()
    {
        sandwich["bread"] = "Wheat";
    }

    public override void AddMeats()
    {
        sandwich["meat"] = "Bacon";
    }

    public override void AddCheese()
    {
        sandwich["cheese"] = "None";
    }

    public override void AddVeggies()
    {
        sandwich["veggies"] = "Lettuce, Tomato";
    }

    public override void AddCondiments()
    {
        sandwich["condiments"] = "Mayo, Mustard";
    }
}
```

Строитель (пример)

Builder

```
/// <summary>
/// The Builder abstract class
/// </summary>
abstract class SandwichBuilder
{
    protected Sandwich sandwich;

    // Gets sandwich instance
    public Sandwich Sandwich
    {
        get { return sandwich; }
    }

    // Abstract build methods
    public abstract void AddBread();
    public abstract void AddMeats();
    public abstract void AddCheese();
    public abstract void AddVeggies();
    public abstract void AddCondiments();
}
```

Main

```
static void Main(string[] args)
{
    SandwichBuilder builder;

    // Create shop with sandwich assembly line
    AssemblyLine shop = new AssemblyLine();

    // Construct and display sandwiches
    builder = new HamAndCheese();
    shop.Assemble(builder);
    builder.Sandwich.Show();

    builder = new BLT();
    shop.Assemble(builder);
    builder.Sandwich.Show();

    builder = new TurkeyClub();
    shop.Assemble(builder);
    builder.Sandwich.Show();

    // Wait for user
    Console.ReadKey();
}
```

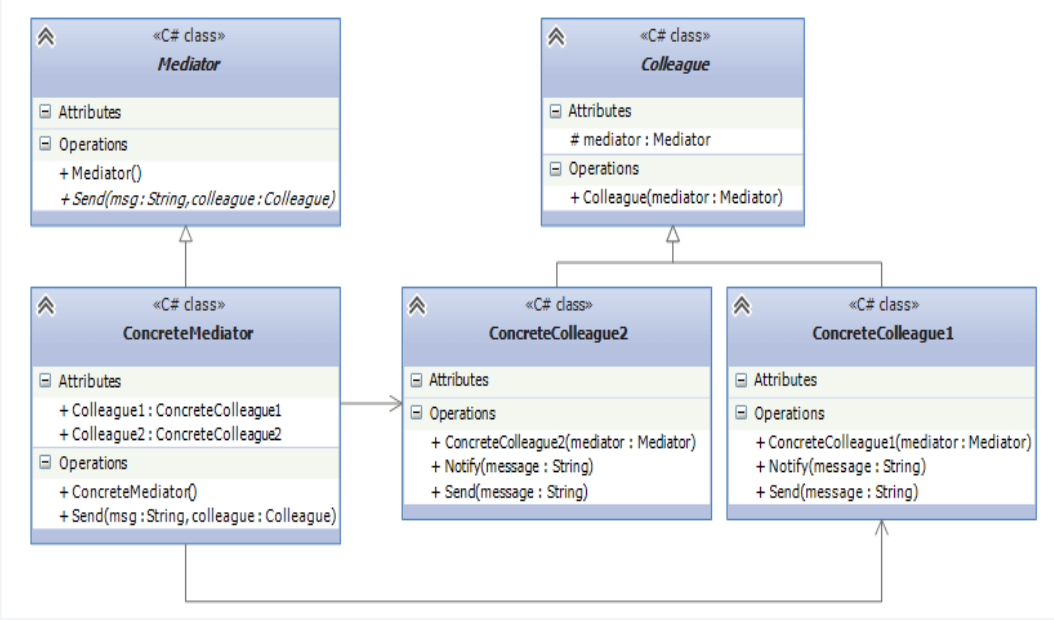
Паттерны поведения

Посредник(Mediator)

Паттерн Посредник (Mediator) представляет такой шаблон проектирования, который обеспечивает взаимодействие множества объектов без необходимости ссылаться друг на друга. Тем самым достигается слабосвязанность взаимодействующих объектов.

Когда надо применять паттерн?

- **Когда имеется множество взаимосвязанных объектов, связи между которыми сложны и запутаны.**
- **Когда необходимо повторно использовать объект, однако повторное использование затруднено в силу сильных**
- **связей с другими объектами.**



Посредник(Mediator)

Участники

- **Mediator** представляет интерфейс для взаимодействия с объектами Calleague
- **Calleague** представляет интерфейсс для взаимодействия с объектом Mediator
- **ConcreteColleague1** и **ConcreteColleague2** конкретные классы коллег, которые обмениваются друг с другом через объект Mediator
- **ConcreteMediator** – конкретный посредник, реализующий интерфейс типа Mediator

Посредник (пример)

Mediator interface

```
/// <summary>
/// The Mediator interface, which defines a send message method which t
/// </summary>
interface Mediator
{
    void SendMessage(string message, ConcessionStand concessionStand);
}
```

Colleagues abstract class

```
/// <summary>
/// The Colleague abstract class, representing
/// </summary>
abstract class ConcessionStand
{
    protected Mediator mediator;

    public ConcessionStand(Mediator mediator)
    {
        this.mediator = mediator;
    }
}
```

Посредник (пример)

Concrete colleague classes

```
/// <summary>
/// A Concrete Colleague class
/// </summary>
class NorthConcessionStand : ConcessionStand
{
    // Constructor
    public NorthConcessionStand(Mediator mediator) : base(mediator)
    {
    }

    public void Send(string message)
    {
        Console.WriteLine("North Concession Stand sends message: " + message);
        mediator.SendMessage(message, this);
    }

    public void Notify(string message)
    {
        Console.WriteLine("North Concession Stand gets message: "  + message);
    }
}
```

```
/// <summary>
/// A Concrete Colleague class
/// </summary>
class SouthConcessionStand : ConcessionStand
{
    public SouthConcessionStand(Mediator mediator) : base(mediator)
    {
    }

    public void Send(string message)
    {
        Console.WriteLine("South Concession Stand sends message: " + message);
        mediator.SendMessage(message, this);
    }

    public void Notify(string message)
    {
        Console.WriteLine("South Concession Stand gets message: " + message);
    }
}
```

Посредник (пример)

Concrete mediator class

```
/// <summary>
/// The Concrete Mediator class, which implement the send message metho
/// </summary>
class ConcessionsMediator : Mediator
{
    private NorthConcessionStand _northConcessions;
    private SouthConcessionStand _southConcessions;

    public NorthConcessionStand NorthConcessions
    {
        set { _northConcessions = value; }
    }

    public SouthConcessionStand SouthConcessions
    {
        set { _southConcessions = value; }
    }

    public void SendMessage(string message, ConcessionStand colleague)
    {
        if (colleague == _northConcessions)
        {
            _southConcessions.Notify(message);
        }
        else
        {
            _northConcessions.Notify(message);
        }
    }
}
```

Main

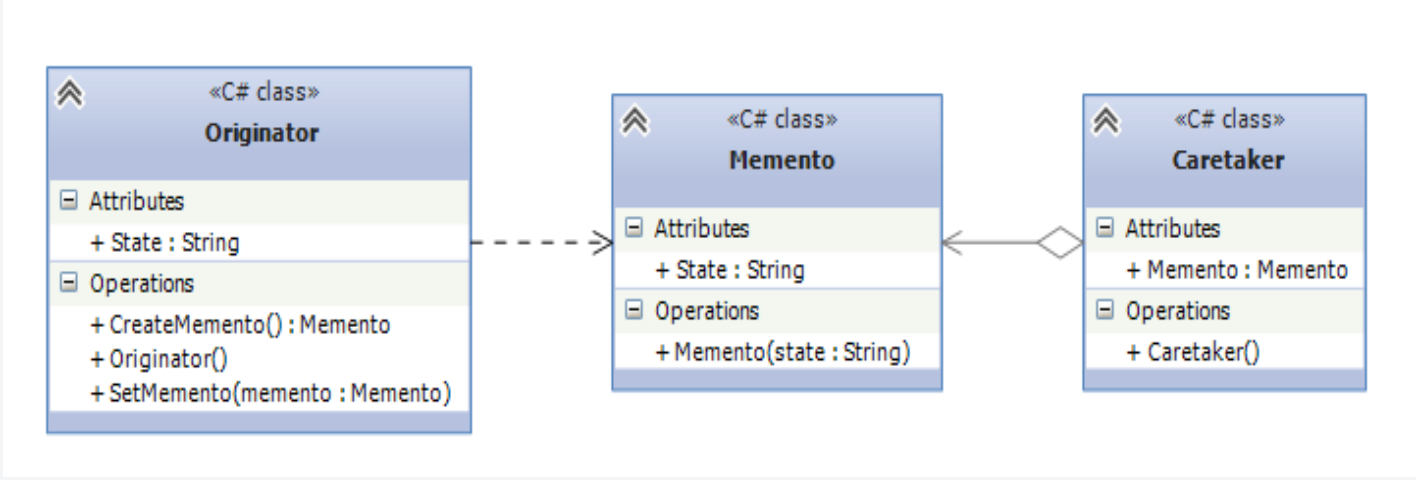
```
static void Main(string[] args)
{
    ConcessionsMediator mediator = new ConcessionsMediator();

    NorthConcessionStand leftKitchen = new NorthConcessionStand(mediator);
    SouthConcessionStand rightKitchen = new SouthConcessionStand(mediator);

    mediator.NorthConcessions = leftKitchen;
    mediator.SouthConcessions = rightKitchen;

    leftKitchen.Send("Can you send some popcorn?");
    rightKitchen.Send("Sure thing, Kenny's on his way.");

    rightKitchen.Send("Do you have any extra hot dogs?  We've had a rush on
    leftKitchen.Send("Just a couple, we'll send Kenny back with them.");

    Console.ReadKey();
}
```

Хранитель (Memento)

Паттерн Хранитель (Memento) позволяет выносить внутреннее состояние объекта за его пределы для последующего возможного восстановления объекта без нарушения принципа инкапсуляции.

Когда надо применять паттерн?

- **Когда нужно сохранить состояние объекта для возможного последующего восстановления.**
- **Когда сохранение состояния должно проходить без нарушения принципа инкапсуляции**



Хранитель (Memento)

Участники

- **Originator** хранитель, который сохраняет состояние объекта Originator и предоставляет полный доступ только этому объекту Originator
- **Memento** создает объект хранителя для сохранения своего состояния
- **Carataker** выполняет только функцию хранения объекта Memento, в то же время у него нет полного доступа к хранителю и никаких других операций над хранителем, кроме собственно сохранения, он не производит

Хранитель (пример)

Originator

```
/// <summary>
/// The Originator class, which is the class for which we want t
/// </summary>
class FoodSupplier
{
    private string _name;
    private string _phone;
    private string _address;

    public string Name
    {
        get { return _name; }
        set
        {
            _name = value;
            Console.WriteLine("Proprietor:  " + _name);
        }
    }

    public string Phone
    {
        get { return _phone; }
        set
        {
            _phone = value;
            Console.WriteLine("Phone Number: " + _phone);
        }
    }
```

```
    public string Address
    {
        get { return _address; }
        set
        {
            _address = value;
            Console.WriteLine("Address: " + _address);
        }
    }

    public FoodSupplierMemento SaveMemento()
    {
        Console.WriteLine("\nSaving current state\n");
        return new FoodSupplierMemento(_name, _phone, _address);
    }

    public void RestoreMemento(FoodSupplierMemento memento)
    {
        Console.WriteLine("\nRestoring previous state\n");
        Name = memento.Name;
        Phone = memento.PhoneNumber;
        Address = memento.Address;
    }
}
```

Хранитель (пример)

Memento

```
/// <summary>
/// The Memento class
/// </summary>
class FoodSupplierMemento
{
    public string Name { get; set; }
    public string PhoneNumber { get; set; }
    public string Address { get; set; }

    public FoodSupplierMemento(string name, string phone, string address)
    {
        Name = name;
        PhoneNumber = phone;
        Address = address;
    }
}
```

Caretaker

```
/// <summary>
/// The Caretaker class.  This class never examines t
/// responsible for keeping that memento.
/// </summary>
class SupplierMemory
{
    private FoodSupplierMemento _memento;

    public FoodSupplierMemento Memento
    {
        set { _memento = value; }
        get { return _memento; }
    }
}
```

Main

```
static void Main(string[] args)
{
    //Here's a new supplier for our restaurant
    FoodSupplier s = new FoodSupplier();
    s.Name = "Harold Karstark";
    s.Phone = "(482) 555-1172";

    // Let's store that entry in our database.
    SupplierMemory m = new SupplierMemory();
    m.Memento = s.SaveMemento();

    // Continue changing originator
    s.Address = "548 S Main St. Nowhere, KS";

    // Crap, gotta undo that entry, I entered the wrong address
    s.RestoreMemento(m.Memento);

    Console.ReadKey();
}
```

Структурные паттерны

Приспособленец(Flyweight)

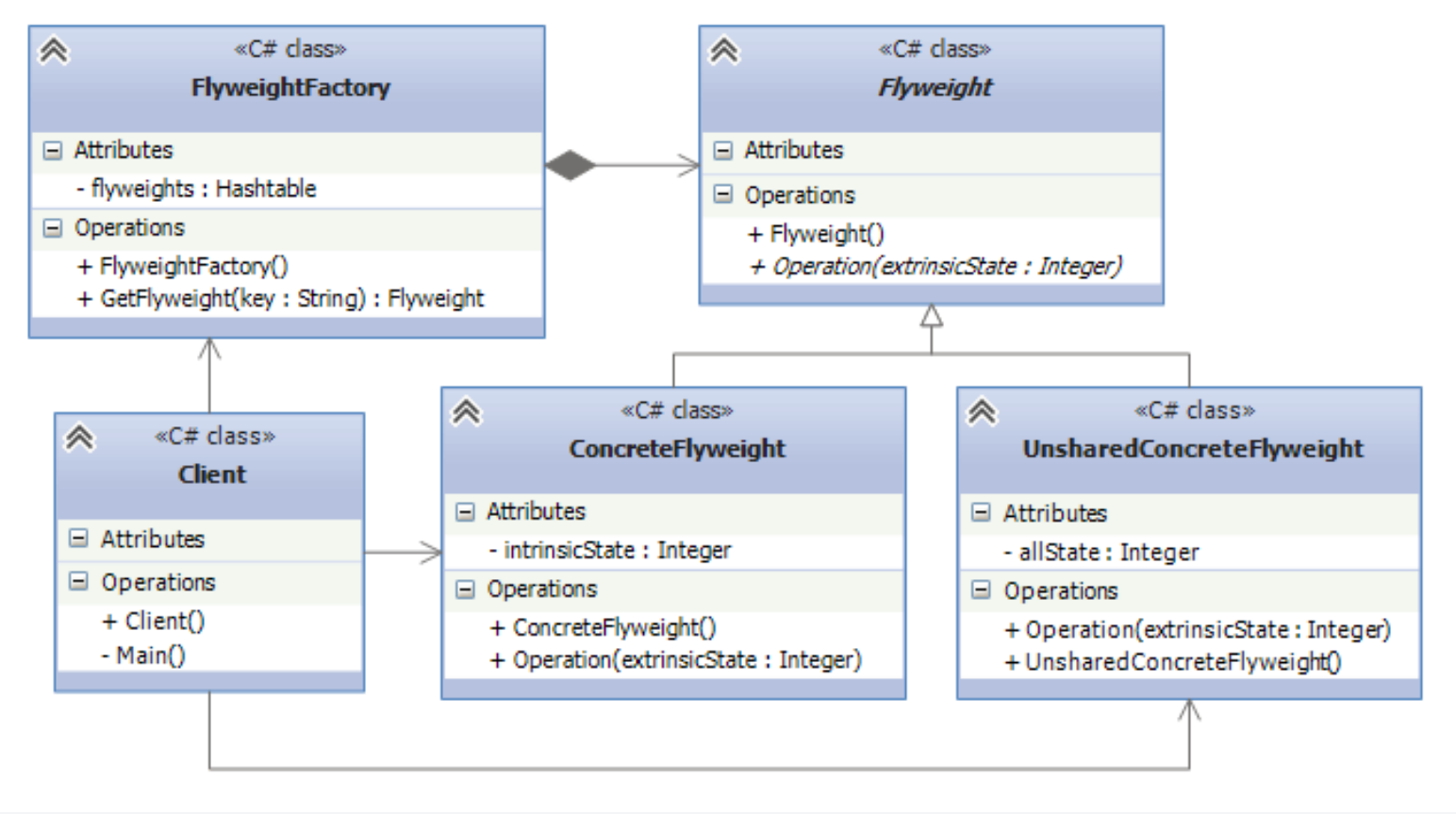
Паттерн Приспособленец (Flyweight) - структурный шаблон проектирования, который позволяет использовать разделяемые объекты сразу в нескольких контекстах. Данный паттерн используется преимущественно для оптимизации работы с памятью. Ключевым моментом здесь является разделение состояния на внутренне и внешнее. Внутреннее состояние не зависит от контекста. Внутреннее состояние выносится в разделяемые объекты.

Внешнее состояние зависит от контекста, является изменчивым. При создании приспособленца внешнее состояние выносится. В приспособленце остается только внутреннее состояние. То есть в примере с символами приспособленец будет хранить код символа.

Когда надо применять паттерн?

- **Когда приложение использует большое количество однообразных объектов, из-за чего происходит выделение большого количества памяти**
- **Когда часть состояния объекта, которое является изменяемым, можно вынести во вне. Вынесение внешнего состояния позволяет заменить множество объектов небольшой группой общих разделяемых объектов.**

Приспособленец(Flyweight)



Приспособленец(Flyweight)

Участники

- **Flyweight** определяет интерфейс, через который приспособленцы-разделяемые объекты могут получать внешнее состояние или воздействовать на него
- **ConcreteFlyweight** конкретный класс разделяемого приспособленца. Реализует интерфейс, объявленный в типе Flyweight, и при необходимости добавляет внутреннее состояние
- **UnsharedConcreteFlyweight** еще одна конкретная реализация интерфейса, определенного в типе Flyweight, только теперь объекты этого класса являются неразделяемыми
- **FlyweightFactory** фабрика приспособленцев - создает объекты разделяемых приспособленцев. Так как приспособленцы разделяются, то клиент не должен создавать их напрямую. Все созданные объекты хранятся в пуле. Часто используются объекты Hashtable, но это не обязательно. Можно применять и другие классы коллекций
- **Client** использует объекты приспособленцев

Приспособленец (пример)

Flyweight class

```
/// The 'Flyweight' abstract class
/// a slider is a small burger, typically only 3 or 4 inches in diameter
abstract class Slider
{
    protected string Name;
    protected string Cheese;
    protected string Toppings;
    protected decimal Price;

    public abstract void Display(int orderTotal);
}
```

Приспособленец (пример)

ConcreteFlyweight

```
class BaconMaster : Slider
{
    public BaconMaster()
    {
        Name = "Bacon Master";
        Cheese = "American";
        Toppings = "lots of bacon";
        Price = 2.39m;
    }

    public override void Display(int orderTotal)
    {
        Console.WriteLine("Slider #" + orderTotal + ": " +
        Name + " - topped with " +
        Cheese + " cheese and " + Toppings +
        "! $" + Price.ToString());
    }
}
```

```
class VeggieSlider : Slider
{
    public VeggieSlider()
    {
        Name = "Veggie Slider";
        Cheese = "Swiss";
        Toppings = "lettuce, onion, tomato, and pickles";
        Price = 1.99m;
    }

    public override void Display(int orderTotal)
    {
        Console.WriteLine("Slider #" + orderTotal + ": " +
        Name + " - topped with " +
        Cheese + " cheese and " + Toppings +
        "! $" + Price.ToString());
    }

}
```

Приспособленец (пример)

FlyweightFactory

```
class SliderFactory
{
    private Dictionary<char, Slider> _sliders =
        new Dictionary<char, Slider>();

    public Slider GetSlider(char key)
    {
        Slider slider = null;
        if (_sliders.ContainsKey(key))
        {
            slider = _sliders[key];
        }
        else
        {
            switch (key)
            {
                case 'B': slider = new BaconMaster(); break;
                case 'V': slider = new VeggieSlider(); break;
                case 'Q': slider = new BBQKing(); break;
            }
            _sliders.Add(key, slider);
        }
        return slider;
    }
}
```

Приспособленец (пример)

Main

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Please enter your slider order (use characters B, V):");
        var order = Console.ReadLine();
        char[] chars = order.ToCharArray();

        SliderFactory factory = new SliderFactory();

        int orderTotal = 0;
        foreach (char c in chars)
        {
            orderTotal++;
            Slider character = factory.GetSlider(c);
            character.Display(orderTotal);
        }

        Console.ReadKey();
    }
}
```

Компоновщик (Composite)

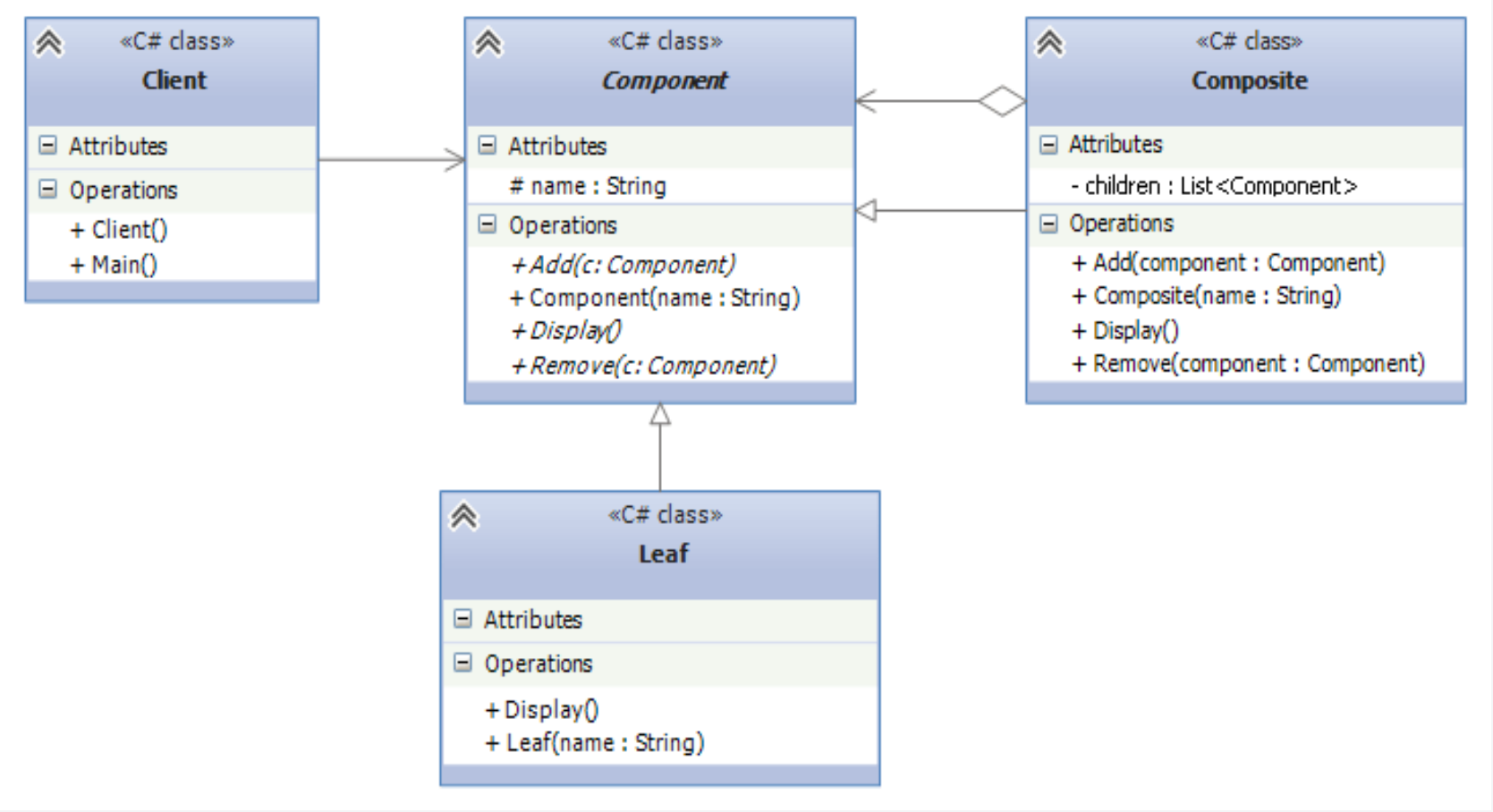
Паттерн Компоновщик (Composite) объединяет группы объектов в древовидную структуру по принципу "часть-целое и позволяет клиенту одинаково работать как с отдельными объектами, так и с группой объектов.

Образно реализацию паттерна можно представить в виде меню, которое имеет различные пункты. Эти пункты могут содержать подменю, в которых, в свою очередь, также имеются пункты. То есть пункт меню служит с одной стороны частью меню, а с другой стороны еще одним меню. В итоге мы однообразно можем работать как с пунктом меню, так и со всем меню в целом.

Когда надо применять паттерн?

- **Когда объекты должны быть реализованы в виде иерархической древовидной структуры**
- **Когда клиенты единообразно должны управлять как целыми объектами, так и их составными частями. То есть целое и его части должны реализовать один и тот же интерфейс**

Компоновщик (Composite)



Компоновщик (Composite)

Участники

- **Component** определяет представляет компонент, который может содержать другие компоненты и реализует механизм для их добавления и удаления
- **Composite** представляет компонент, который может содержать другие компоненты и реализует механизм для их добавления и удаления
- **Leaf** представляет отдельный компонент, который не может содержать другие компоненты
- **Client** использует компоненты

Компоновщик (пример)

Component

```
/// Component abstract class
public abstract class SoftDrink
{
    public int Calories { get; set; }

    public List<SoftDrink> Flavors { get; set; }

    public SoftDrink(int calories)
    {
        Calories = calories;
        Flavors = new List<SoftDrink>();
    }

    /// "Flatten" method, returns all available flavors
    public void DisplayCalories()
    {
        Console.WriteLine(this.GetType().Name + ": " +
            this.Calories.ToString() + " calories.");
        foreach (var drink in this.Flavors)
        {
            drink.DisplayCalories();
        }
    }
}
```

Leaf classes

```
/// Leaf class
public class VanillaCola : SoftDrink
{
    public VanillaCola(int calories) : base(calories) { }
}

/// Leaf class
public class CherryCola : SoftDrink
{
    public CherryCola(int calories) : base(calories) { }
}

    /// Leaf class
public class StrawberryRootBeer : SoftDrink
{
    public StrawberryRootBeer(int calories) : base(calories) { }
}

    /// Leaf class
public class VanillaRootBeer : SoftDrink
{
    public VanillaRootBeer(int calories) : base(calories) { }
}

 /// Leaf class
public class LemonLime : SoftDrink
{
    public LemonLime(int calories) : base(calories) { }
}
```

Компоновщик (пример)

Composite

```
    /// Composite class
public class Cola : SoftDrink
{
    public Cola(int calories) : base(calories) { }
}

    /// Composite class
public class RootBeer : SoftDrink
{
    public RootBeer(int calories) : base(calories) { }
}
```

Root composite

```
    /// Composite class, root node
public class SodaWater : SoftDrink
{
    public SodaWater(int calories) : base(calories) { }
}
```

Main

```
class Program
{
    static void Main(string[] args)
    {
        var colas = new Cola(210);
        colas.Flavors.Add(new VanillaCola(215));
        colas.Flavors.Add(new CherryCola(210));

        var lemonLime = new LemonLime(185);

        var rootBeers = new RootBeer(195);
        rootBeers.Flavors.Add(new VanillaRootBeer(200));
        rootBeers.Flavors.Add(new StrawberryRootBeer(200));

        SodaWater sodaWater = new SodaWater(180);
        sodaWater.Flavors.Add(colas);
        sodaWater.Flavors.Add(lemonLime);
        sodaWater.Flavors.Add(rootBeers);

        sodaWater.DisplayCalories();

        Console.ReadKey();
    }
}
```