Структурные паттерны. Часть 2

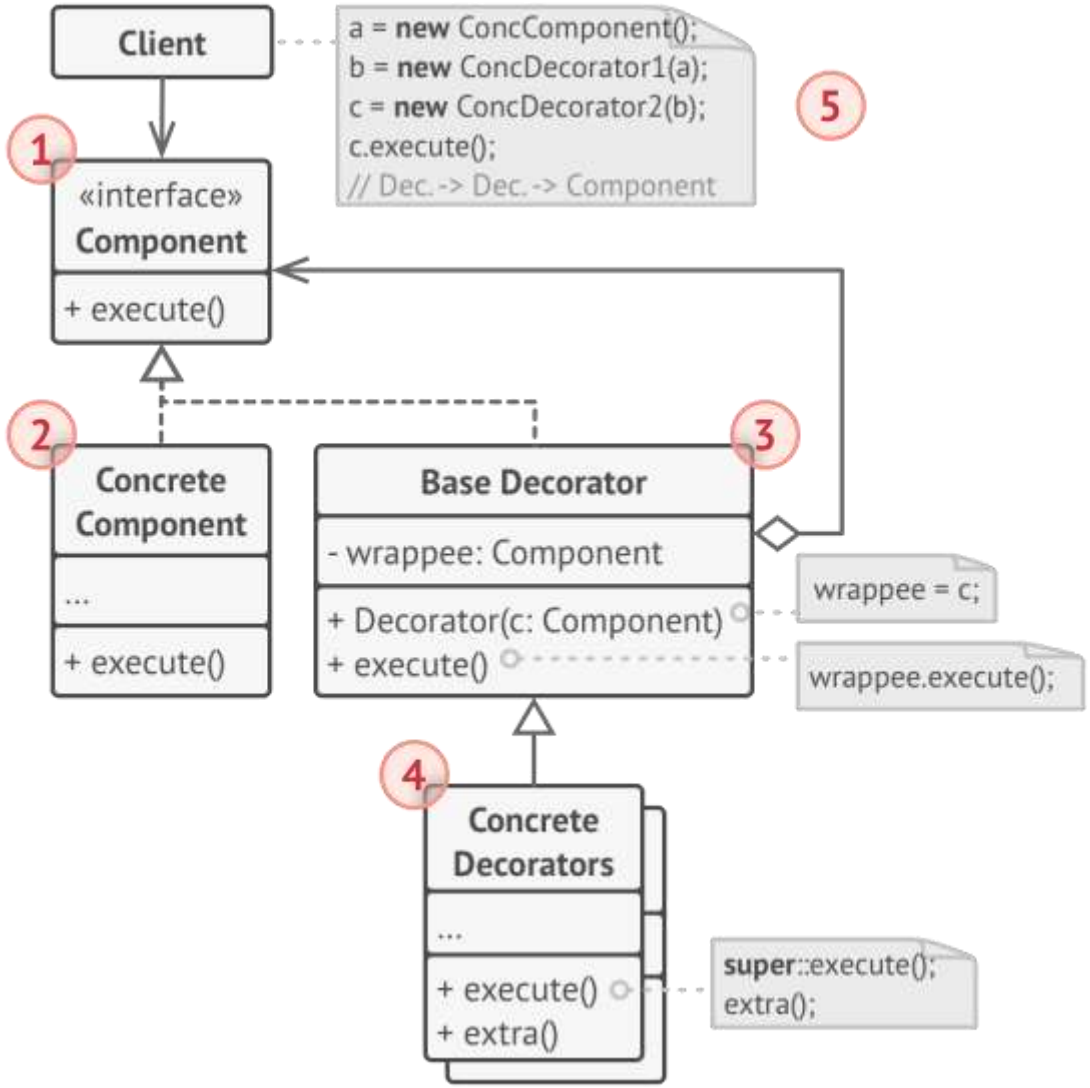
Декоратор

Позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».

Когда применять?

- Когда надо динамически добавлять к объекту новые функциональные возможности. При этом данные возможности могут быть сняты с объекта.
- Когда применение наследования неприемлемо.

Структура


Client
a = new ConcComponent();
b = new ConcDecorator1(a);
c = new ConcDecorator2(b);
c.execute();
// Dec.-> Dec.-> Component
5
1
«interface»
Component
+ execute()
2
Concrete
Component
...
+ execute()
3
Base Decorator
- wrappee: Component
+ Decorator(c: Component)
+ execute()
wrappee = c;
wrappee.execute();
4
Concrete
Decorators
...
+ execute()
+ extra()
super::execute();
extra();

Пример

```
// Базовый интерфейс Компонента определяет поведение, которое изменяется
// декораторами.
public abstract class Component
{
    public abstract string Operation();
}
```

```
// Конкретные Компоненты предоставляют реализации поведения по умолчанию.
// Может быть несколько вариаций этих классов.
class ConcreteComponent : Component
{
    public override string Operation()
    {
        return "ConcreteComponent";
    }
}
```

```
// Базовый класс Декоратора следует тому же интерфейсу, что и другие
// компоненты. Основная цель этого класса - определить интерфейс обёртки для
// всех конкретных декораторов. Реализация кода обёртки по умолчанию может
// включать в себя  поле для хранения завёрнутого компонента и средства его
// инициализации.
abstract class Decorator : Component
{
    protected Component _component;

    public Decorator(Component component)
    {
        this._component = component;
    }

    public void SetComponent(Component component)
    {
        this._component = component;
    }

    // Декоратор делегирует всю работу обёрнутому компоненту.
    public override string Operation()
    {
        if (this._component != null)
        {
            return this._component.Operation();
        }
        else
        {
            return string.Empty;
        }
    }
}
```

Пример

```
// Конкретные Декораторы вызывают обёрнутый объект и изменяют его результат
// некоторым образом.
class ConcreteDecoratorA : Decorator
{
    public ConcreteDecoratorA(Component comp) : base(comp)
    {
    }

    // Декораторы могут вызывать родительскую реализацию операции, вместо
    // того, чтобы вызвать обёрнутый объект напрямую. Такой подход упрощает
    // расширение классов декораторов.
    public override string Operation()
    {
        return $"ConcreteDecoratorA({base.Operation()})";
    }
}
```

```
// Декораторы могут выполнять своё поведение до или после вызова обёрнутого
// объекта.
class ConcreteDecoratorB : Decorator
{
    public ConcreteDecoratorB(Component comp) : base(comp)
    {
    }

    public override string Operation()
    {
        return $"ConcreteDecoratorB({base.Operation()})";
    }
}
```

Пример

```
public class Client
{
    // Клиентский код работает со всеми объектами, используя интерфейс
    // Компонента. Таким образом, он остаётся независимым от конкретных
    // классов компонентов, с которыми работает.
    public void ClientCode(Component component)
    {
        Console.WriteLine("RESULT: " + component.Operation());
    }
}
```

Пример

```
class Program
{
    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();

        // ...так и декорированные.
        //
        // Обратите внимание, что декораторы могут обёртывать не только
        // простые компоненты, но и другие декораторы.
        ConcreteDecoratorA decorator1 = new ConcreteDecoratorA(simple);
        ConcreteDecoratorB decorator2 = new ConcreteDecoratorB(decorator1);
        Console.WriteLine("Client: Now I've got a decorated component:");
        client.ClientCode(decorator2);
    }
}
```

```
Client: I get a simple component:
RESULT: ConcreteComponent

Client: Now I've got a decorated component:
RESULT: ConcreteDecoratorB(ConcreteDecoratorA(ConcreteComponent))
```

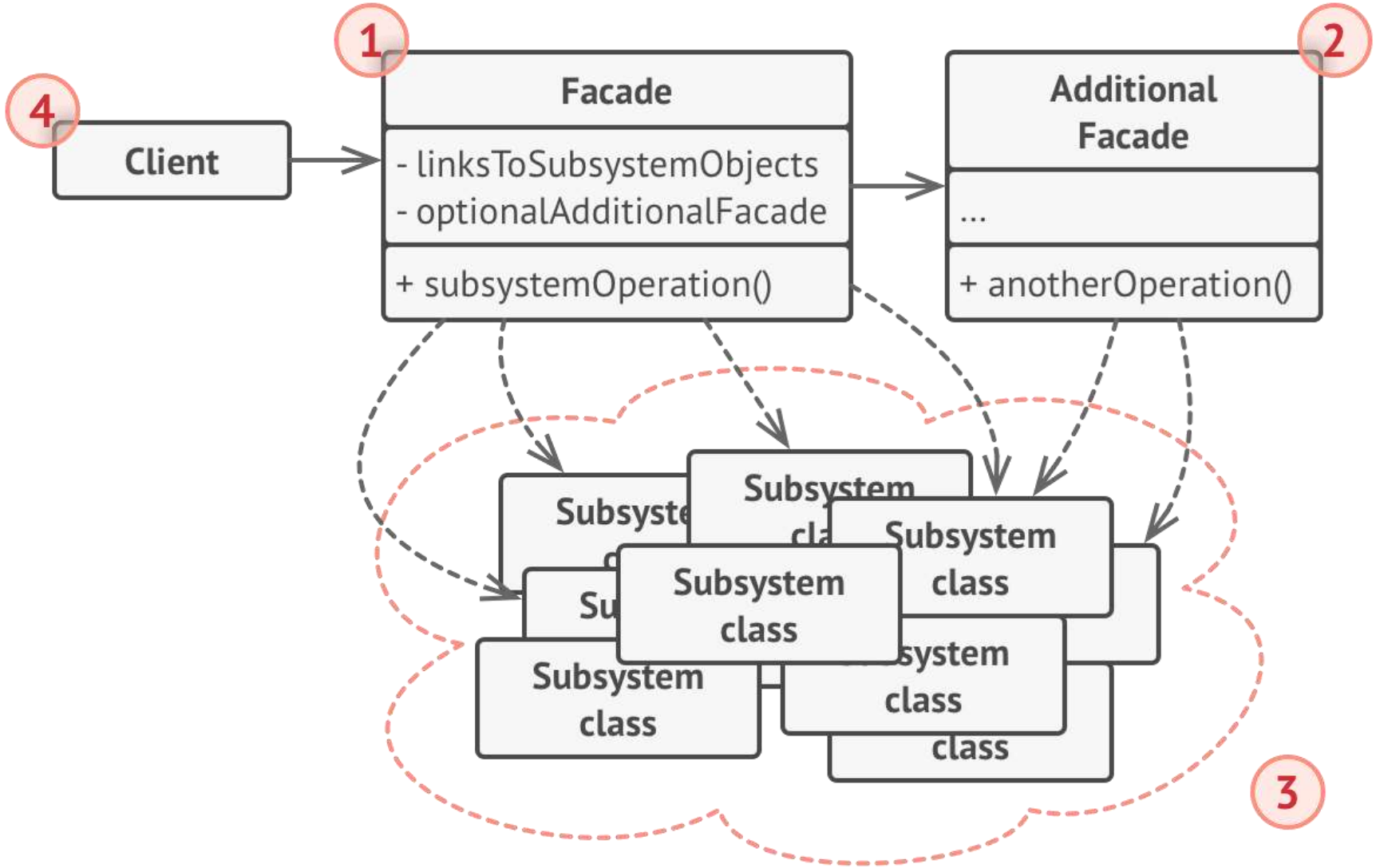
Фасад

Предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку.

Когда применять?

- Когда имеется сложная система, и необходимо упростить с ней работу. Фасад позволит определить одну точку взаимодействия между клиентом и системой.
- Когда надо уменьшить количество зависимостей между клиентом и сложной системой. Фасадные объекты позволяют отделить, изолировать компоненты системы от клиента и развивать и работать с ними независимо.

Структура


1
2
3
4
Facade
- linksToSubsystemObjects
- optionalAdditionalFacade
+ subsystemOperation()
Additional Facade
...
+ anotherOperation()
Client
Subsystem class
Subsystem class
Subsystem class
Subsystem class
Subsystem class

Пример

```
// Подсистема может принимать запросы либо от фасада, либо от клиента
// напрямую. В любом случае, для Подсистемы Фасад – это еще один клиент, и
// он не является частью Подсистемы.
public class Subsystem1
{
    public string operation1()
    {
        return "Subsystem1: Ready!\n";
    }

    public string operationN()
    {
        return "Subsystem1: Go!\n";
    }
}
```

```
// Некоторые фасады могут работать с разными подсистемами одновременно.
public class Subsystem2
{
    public string operation1()
    {
        return "Subsystem2: Get ready!\n";
    }

    public string operationZ()
    {
        return "Subsystem2: Fire!\n";
    }
}
```

Пример

```
// Класс Фасада предоставляет простой интерфейс для сложной логики одной или
// нескольких подсистем. Фасад делегирует запросы клиентов соответствующим
// объектам внутри подсистемы. Фасад также отвечает за управление их
// жизненным циклом. Все это защищает клиента от нежелательной сложности
// подсистемы.
public class Facade
{
    protected Subsystem1 _subsystem1;

    protected Subsystem2 _subsystem2;

    public Facade(Subsystem1 subsystem1, Subsystem2 subsystem2)
    {
        this._subsystem1 = subsystem1;
        this._subsystem2 = subsystem2;
    }

    // Методы Фасада удобны для быстрого доступа к сложной функциональности
    // подсистем. Однако клиенты получают только часть возможностей
    // подсистемы.
    public string Operation()
    {
        string result = "Facade initializes subsystems:\n";
        result += this._subsystem1.operation1();
        result += this._subsystem2.operation1();
        result += "Facade orders subsystems to perform the action:\n";
        result += this._subsystem1.operationN();
        result += this._subsystem2.operationZ();
        return result;
    }
}
```

Пример

```
class Client
{
    // Клиентский код работает со сложными подсистемами через простой
    // интерфейс, предоставляемый Фасадом. Когда фасад управляет жизненным
    // циклом подсистемы, клиент может даже не знать о существовании
    // подсистемы. Такой подход позволяет держать сложность под контролем.
    public static void ClientCode(Facade facade)
    {
        Console.Write(facade.Operation());
    }
}
```

Пример

```
class Program
{
    static void Main(string[] args)
    {
        // В клиентском коде могут быть уже созданы некоторые объекты
        // подсистемы. В этом случае может оказаться целесообразным
        // инициализировать Фасад с этими объектами вместо того, чтобы
        // позволить Фасаду создавать новые экземпляры.
        Subsystem1 subsystem1 = new Subsystem1();
        Subsystem2 subsystem2 = new Subsystem2();
        Facade facade = new Facade(subsystem1, subsystem2);
        Client.ClientCode(facade);
    }
}
```

```
Facade initializes subsystems:
Subsystem1: Ready!
Subsystem2: Get ready!
Facade orders subsystems to perform the action:
Subsystem1: Go!
Subsystem2: Fire!
```

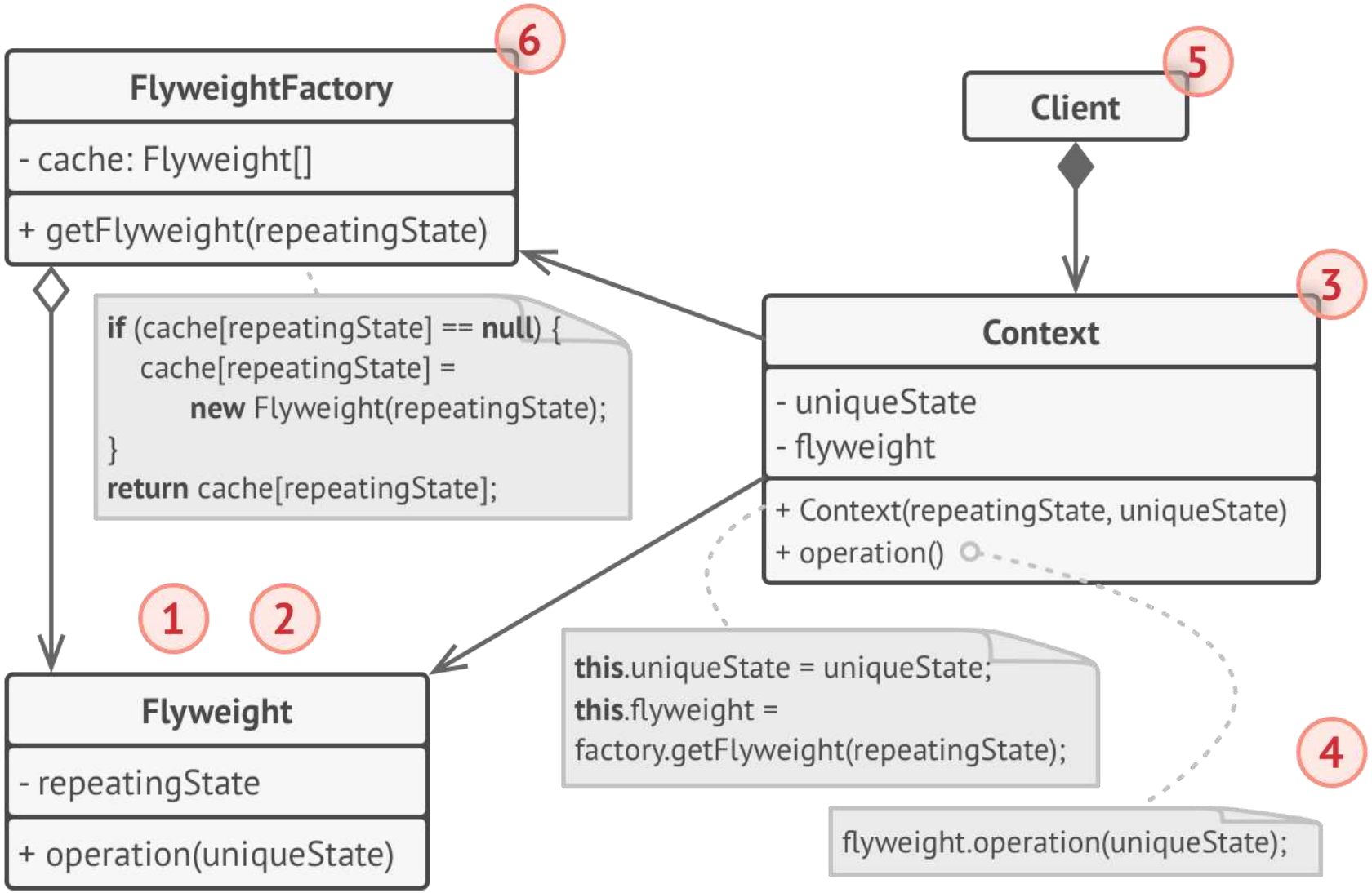
Легковес

Позволяет вместить бóльшее количество объектов в отведенной оперативной памяти за счет экономного разделения общего состояния объектов между собой, вместо хранения одинаковых данных в каждом объекте.

Когда применять?

- 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.

Структура


6
FlyweightFactory
- cache: Flyweight[]
+ getFlyweight(repeatingState)
5
Client
3
Context
- uniqueState
- flyweight
+ Context(repeatingState, uniqueState)
+ operation()
if (cache[repeatingState] == null) {
cache[repeatingState] =
new Flyweight(repeatingState);
}
return cache[repeatingState];
1
2
Flyweight
- repeatingState
+ operation(uniqueState)
this.uniqueState = uniqueState;
this.flyweight =
factory.getFlyweight(repeatingState);
4
flyweight.operation(uniqueState);

Пример

```
1  using System.Drawing;
2
3  namespace ITMO_OOP
4  {
       9 references
5      public class BulletAppearance
6      {
           0 references
7          public BulletAppearance(Color color, double width, double height)
8          {
9              Color = color;
10             Width = width;
11             Height = height;
12         }
           1 reference
13         public Color Color { get; }
           1 reference
14         public double Width { get; }
           1 reference
15         public double Height { get; }
16     }
17 }
```

Пример

```
1  using System.Drawing;
2
3  namespace ITMO_OOP
4  {
       2 references
5      public interface IBulletAppearanceFactory
6      {
           2 references
7          BulletAppearance GetBulletAppearance(Color color, double width, double height);
           1 reference
8          void AddBulletAppearance(BulletAppearance bulletAppearance);
9      }
10 }
```

Пример

```
1  using System.Collections.Generic;
2  using System.Drawing;
3
4  namespace ITMO_OOP
5  {
       1 reference
6      public class BulletAppearanceFactory : IBulletAppearanceFactory
7      {
8          private readonly Dictionary<(Color Color, double Width, double Height), BulletAppearance> _existingAppearances =
9              new Dictionary<(Color Color, double Width, double Height), BulletAppearance>();
10
               2 references
11         public BulletAppearance GetBulletAppearance(Color color, double width, double height)
12         {
13             var id = (color, width, height);
14             if (!_existingAppearances.TryGetValue(id, out var result))
15             {
16                 result = _existingAppearances[id] = new BulletAppearance(color, width, height);
17             }
18
19             return result;
20         }
21
               1 reference
22         public void AddBulletAppearance(BulletAppearance bulletAppearance)
23         {
24             var id = (bulletAppearance.Color, bulletAppearance.Width, bulletAppearance.Height);
25
26             if (_existingAppearances.ContainsKey(id)) return;
27
28             _existingAppearances[id] = bulletAppearance;
29         }
30     }
31 }
```

Пример

```
1  using System.Drawing;
2  using System.Numerics;
3
4  namespace ITMO_OOP
5  {
       4 references
6      public class Bullet
7      {
8          private BulletAppearance _appearance;
9
           1 reference
10         public Bullet(IBulletAppearanceFactory factory, Color color, double width, double height, Vector3 speed)
11         {
12             _appearance = factory.GetBulletAppearance(color, width, height);
13             Speed = speed;
14         }
15
           0 references
16         public Color Color => _appearance.Color;
           0 references
17         public double Width => _appearance.Width;
           0 references
18         public double Height => _appearance.Height;
           1 reference
19         public Vector3 Speed { get; }
20     }
21 }
```

Пример

```
1  using System.Collections.Generic;
2  using System.Drawing;
3  using System.Numerics;
4
5  namespace ITMO_OOP
6  {
       0 references
7      class Program
8      {
           0 references
9          static void Main(string[] args)
10         {
11             const int BULLETS_COUNT = 1000;
12             var factory = new BulletAppearanceFactory();
13
14             List<Bullet> bullets = new List<Bullet>(BULLETS_COUNT);
15
16             for (var i = 0; i < BULLETS_COUNT; ++i)
17             {
18                 bullets.Add(new Bullet(factory, Color.Red, width: 10, height: 10, new Vector3(x: i % 2, y: i % 3, z: i % 4)));
19             }
20
21             // Do something with these bullets... draw them, for example
22         }
23     }
24 }
25
```

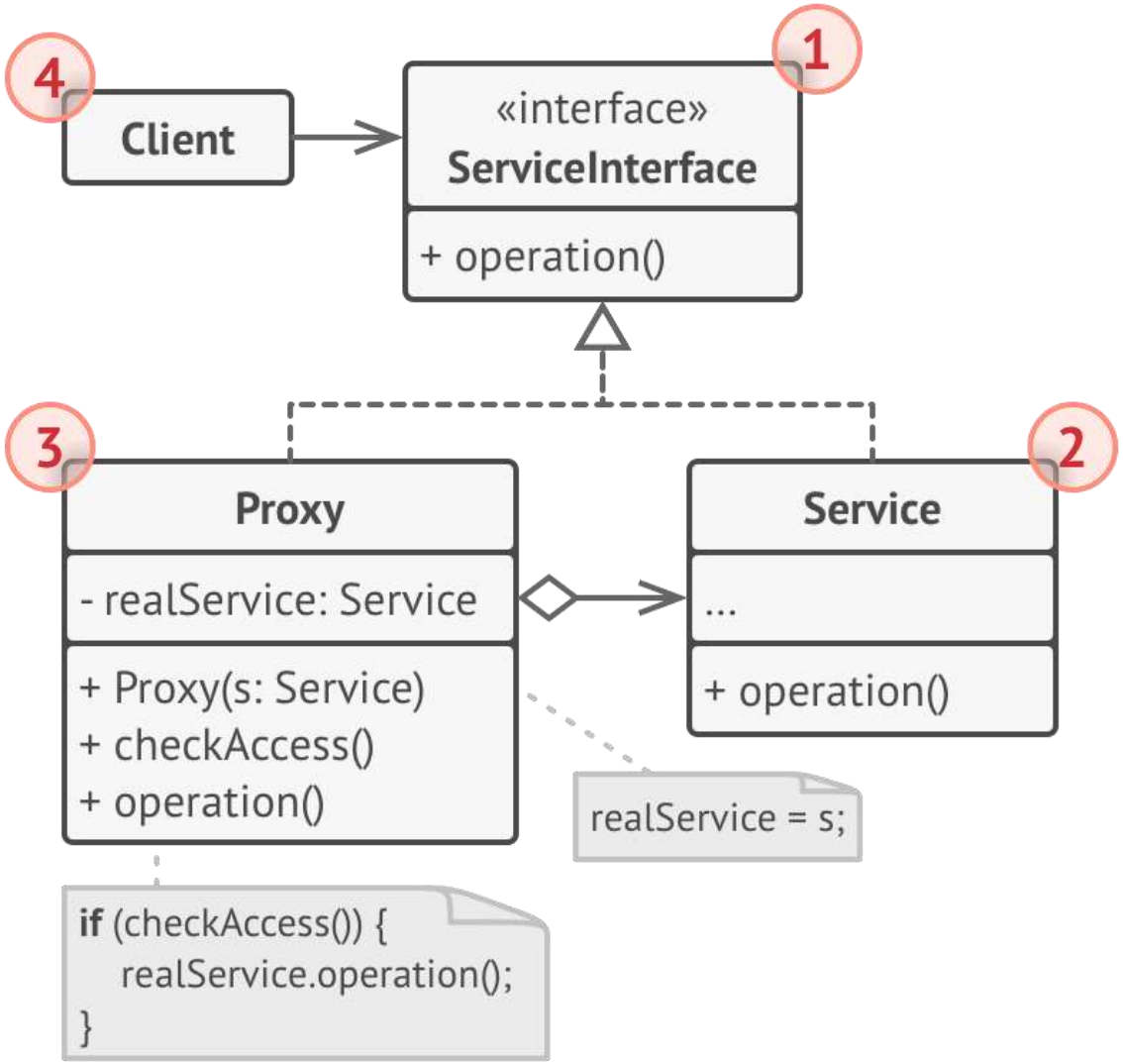
Заместитель

Позволяет подставлять вместо реальных объектов специальные объекты-заменители. Эти объекты перехватывают вызовы к оригинальному объекту, позволяя сделать что-то до или после передачи вызова оригиналу.

Когда применять?

- Когда нужно управлять доступом к ресурсу, создание которого требует больших затрат. Реальный объект создается только тогда, когда он действительно может понадобится, а до этого все запросы к нему обрабатывает прокси-объект.
- Когда необходимо разграничить доступ к вызываемому объекту в зависимости от прав вызывающего объекта.
- Когда нужно вести подсчет ссылок на объект или обеспечить потокобезопасную работу с реальным объектом.

Структура


4
Client
1
«interface»
ServiceInterface
+ operation()
3
Proxy
- realService: Service
+ Proxy(s: Service)
+ checkAccess()
+ operation()
2
Service
...
+ operation()
realService = s;
if (checkAccess()) {
realService.operation();
}

Пример

```
// Интерфейс Субъекта объявляет общие операции как для Реального Субъекта,
// так и для Заместителя. Пока клиент работает с Реальным Субъектом,
// используя этот интерфейс, вы сможете передать ему заместителя вместо
// реального субъекта.
public interface ISubject
{
    void Request();
}
```

```
// Реальный Субъект содержит некоторую базовую бизнес-логику. Как правило,
// Реальные Субъекты способны выполнять некоторую полезную работу, которая к
// тому же может быть очень медленной или точной — например, коррекция
// входных данных. Заместитель может решить эти задачи без каких-либо
// изменений в коде Реального Субъекта.
class RealSubject : ISubject
{
    public void Request()
    {
        Console.WriteLine("RealSubject: Handling Request.");
    }
}
```

Пример

```
// Интерфейс Заместителя идентичен интерфейсу Реального Субъекта.
class Proxy : ISubject
{
    private RealSubject _realSubject;

    public Proxy(RealSubject realSubject)
    {
        this._realSubject = realSubject;
    }

    // Наиболее распространёнными областями применения паттерна Заместитель
    // являются ленивая загрузка, кэширование, контроль доступа, ведение
    // журнала и т.д. Заместитель может выполнить одну из этих задач, а
    // затем, в зависимости от результата, передать выполнение одноимённому
    // методу в связанном объекте класса Реального Субъект.
    public void Request()
    {
        if (this.CheckAccess())
        {
            this._realSubject.Request();

            this.LogAccess();
        }
    }

    public bool CheckAccess()
    {
        // Некоторые реальные проверки должны проходить здесь.
        Console.WriteLine("Proxy: Checking access prior to firing a real request.");

        return true;
    }

    public void LogAccess()
    {
        Console.WriteLine("Proxy: Logging the time of request.");
    }
}
```

Пример

```
public class Client
{
    // Клиентский код должен работать со всеми объектами (как с реальными,
    // так и заместителями) через интерфейс Субъекта, чтобы поддерживать как
    // реальные субъекты, так и заместителей. В реальной жизни, однако,
    // клиенты в основном работают с реальными субъектами напрямую. В этом
    // случае, для более простой реализации паттерна, можно расширить
    // заместителя из класса реального субъекта.
    public void ClientCode(ISubject subject)
    {
        // ...

        subject.Request();

        // ...
    }
}
```

Пример

```
class Program
{
    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);
    }
}
```

```
Client: Executing the client code with a real subject:
RealSubject: Handling Request.

Client: Executing the same client code with a proxy:
Proxy: Checking access prior to firing a real request.
RealSubject: Handling Request.
Proxy: Logging the time of request.
```

Спасибо за внимание!