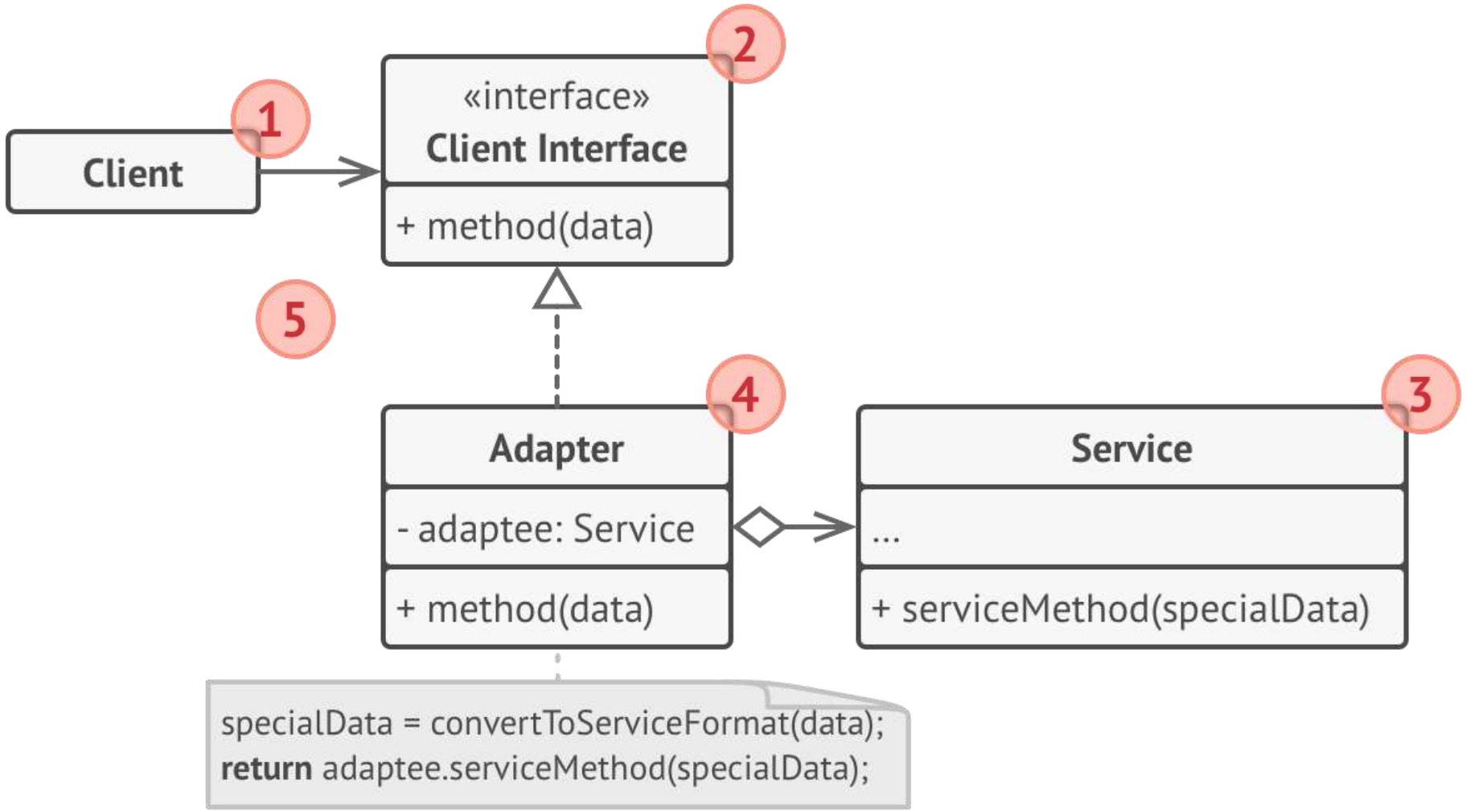
Структурные паттерны. Часть 1

Адаптер (Adapter)

Позволяет объектам с несовместимыми интерфейсами работать вместе.

Структура


1
Client
2
«interface»
Client Interface
+ method(data)
5
4
Adapter
- adaptee: Service
+ method(data)
3
Service
...
+ serviceMethod(specialData)
specialData = convertToServiceFormat(data);
return adaptee.serviceMethod(specialData);

Когда применять?

- Когда необходимо использовать имеющийся класс, но его интерфейс не соответствует потребностям.
- Когда надо использовать уже существующий класс совместно с другими классами, интерфейсы которых не совместимы.

Пример

```
// Целевой класс объявляет интерфейс, с которым может работать клиентский
// код.
public interface ITarget
{
    string GetRequest();
}
```

```
// Адаптируемый класс содержит некоторое полезное поведение, но его
// интерфейс несовместим  с существующим клиентским кодом. Адаптируемый
// класс нуждается в некоторой доработке,  прежде чем клиентский код сможет
// его использовать.
class Adaptee
{
    public string GetSpecificRequest()
    {
        return "Specific request.";
    }
}
```

```
// Адаптер делает интерфейс Адаптируемого класса совместимым с целевым
// интерфейсом.
class Adapter : ITarget
{
    private readonly Adaptee _adaptee;

    public Adapter(Adaptee adaptee)
    {
        this._adaptee = adaptee;
    }

    public string GetRequest()
    {
        return $"This is '{this._adaptee.GetSpecificRequest()}'";
    }
}
```

Пример

```
class Program
{
    static void Main(string[] args)
    {
        Adaptee adaptee = new Adaptee();
        ITarget 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());
    }
}
```

```
Adaptee interface is incompatible with the client.
But with adapter client can call it's method.
This is 'Specific request.'
```

Пример

```
namespace Adapter
{
    /// <summary>
    /// Интерфейс кассового аппарата.
    /// </summary>
    public interface ICashMachine
    {
        /// <summary>
        /// Уникальный номер кассового аппарата.
        /// </summary>
        string Number { get; }

        /// <summary>
        /// Коллекция товаров в текущем чеке.
        /// </summary>
        IEnumerable<Product> Products { get; }

        /// <summary>
        /// Собрать чек и вывести его на печать.
        /// </summary>
        /// <remarks>
        /// Печать чека вызывает его сохранение и очистку коллекции товаров текущего чека.
        /// </remarks>
        /// <returns>Текст чека</returns>
        string PrintCheck();

        /// <summary>
        /// Добавить товар в коллекцию товаров текущего чека.
        /// </summary>
        /// <param name="product">Товар, добавляемый в чек.</param>
        void AddProduct(Product product);

        /// <summary>
        /// Сохранить чек.
        /// </summary>
        /// <remarks>
        /// Сохранение чека вызывает очистку коллекции товаров текущего чека.
        /// </remarks>
        /// <param name="checkText"></param>
        void Save(string checkText);
    }
}
```

```
/// <summary>
/// Товар.
/// </summary>
public class Product
{
    /// <summary>
    /// Наименование товара.
    /// </summary>
    public string Name { get; set; }

    /// <summary>
    /// Стоимость.
    /// </summary>
    public decimal Price { get; set; }

    /// <summary>
    /// Создать новый экземпляр товара.
    /// </summary>
    /// <param name="name">Название.</param>
    /// <param name="price">Цена.</param>
    public Product(string name, decimal price)
    {
        Name = name;
        Price = price;
    }

    /// <summary>
    /// Приведение объекта к строке.
    /// </summary>
    /// <returns>Название товара.</returns>
    public override string ToString()
    {
        return Name;
    }
}
```

Пример

```
/// <summary>
/// Кассовый аппарат собственного производства.
/// </summary>
public class CashMachine : ICashMachine
{
    /// <summary>
    /// Список товаров в чеке.
    /// </summary>
    private List<Product> _products;

    /// <summary>
    /// Уникальный номер кассового аппарата.
    /// </summary>
    private Guid _number;

    /// <inheritdoc />
    public IEnumerable<Product> Products => _products;

    /// <inheritdoc />
    public string Number => _number.ToString();

    /// <summary>
    /// Создать новый экземпляр кассового аппарата.
    /// </summary>
    public CashMachine()
    {
        _number = Guid.NewGuid();
        _products = new List<Product>();
    }

    /// <inheritdoc />
    public void AddProduct(Product product)
    {
        _products.Add(product);
    }

    /// <inheritdoc />
    public string PrintCheck()
    {
        var checkText = GetCheckText();
        Save(checkText);
        return checkText;
    }

    /// <inheritdoc />
    public void Save(string checkText)
    {
        using (var sr = new StreamWriter("checks.txt", true, Encoding.Default))
        {
            sr.WriteLine(checkText);
        }
        _products.Clear();
    }

    /// <summary>
    /// Приведение объекта к строке.
    /// </summary>
    /// <returns>Номер кассового аппарата.</returns>
    public override string ToString()
    {
        return $"Касса №{Number}";
    }

    /// <summary>
    /// Сформировать текст чека для вывода на печать и сохранения в файл.
    /// </summary>
    /// <returns>Форматированный текст чека.</returns>
    private string GetCheckText()
    {
        var date = DateTime.Now.ToString("dd MMMM yyyy HH:mm");
        var checkText = $"Кассовый чек от {date}\r\n";
        checkText += $"Идентификатор чека: {Guid.NewGuid()}\r\n";
        checkText += "Список товаров:\r\n";
        foreach (var product in _products)
        {
            checkText += $"{product.Name}\t\t\t{product.Price}\r\n";
        }
        var sum = _products.Sum(p => p.Price);
        checkText += $"ИТОГО\t\t\t{sum}\r\n";
        return checkText;
    }
}
```

Пример

```
namespace ForeignCashMachine
{
    /// <summary>
    /// Кассовый чек.
    /// </summary>
    public class Check : ICloneable
    {
        /// <summary>
        /// Список товаров в кассовом чеке.
        /// </summary>
        private List<(string Name, double Price)> _products;

        /// <summary>
        /// Номер чека.
        /// </summary>
        public int Number { get; private set; }

        /// <summary>
        /// Дата создания чека.
        /// </summary>
        public DateTime DateTime { get; private set; }

        /// <summary>
        /// Товары в чеке.
        /// </summary>
        public IEnumerable<(string Name, double Price)> Products => _products;

        /// <summary>
        /// Создать экземпляр чека.
        /// </summary>
        public Check()
        {
            var rnd = new Random();

            Number = rnd.Next(10000, 99999);
            DateTime = DateTime.Now;
            _products = new List<(string Name, double Price)>();
        }

        /// <summary>
        /// Добавить товар в чек.
        /// </summary>
        /// <param name="name"></param>
        /// <param name="price"></param>
        public void Add(string name, double price)
        {
            _products.Add((name, price));
        }

        /// <summary>
        /// Создать копию чека.
        /// </summary>
        /// <returns>Копия чека.</returns>
        public object Clone()
        {
            return new Check()
            {
                _products = _products,
                DateTime = DateTime,
                Number = Number
            };
        }

        /// <summary>
        /// Приведение объекта к строке.
        /// </summary>
        /// <returns>Текст чека.</returns>
        public override string ToString()
        {
            var checkText = $"Кассовый чек от {DateTime.ToString("HH:mm dd.MMMM.yyyy")}\r\n";
            checkText += $"Номер чека: {Number}\r\n";
            return checkText;
        }
    }
}
```

Пример

```
namespace ForeignCashMachine
{
    /// <summary>
    /// Иностранный кассовый аппарат.
    /// </summary>
    public class ForeignCashMachine
    {
        /// <summary>
        /// Список чеков, хранящихся в кассовом аппарате.
        /// </summary>
        private List<Check> _checks = new List<Check>();

        /// <summary>
        /// Название кассового аппарата.
        /// </summary>
        public string Name { get; private set; }

        /// <summary>
        /// Массив чеков, хранящихся в кассовом аппарате.
        /// </summary>
        public Check[] Checks => _checks.ToArray();

        /// <summary>
        /// Текущий заполняемый чек.
        /// </summary>
        public Check CurrentCheck => _checks.LastOrDefault();

        /// <summary>
        /// Создать экземпляр кассового аппарата.
        /// </summary>
        public ForeignCashMachine()
        {
            var rnd = new Random();
            Name = $"KKA{rnd.Next(10000, 99999)}";
            _checks.Add(new Check());
        }

        /// <summary>
        /// Добавить товар в текущий чек.
        /// </summary>
        /// <param name="name">Наименование товара.</param>
        /// <param name="price">Стоимость товара.</param>
        public void Add(string name, double price)
        {
            CurrentCheck.Add(name, price);
        }

        /// <summary>
        /// Сохранить чек.
        /// </summary>
        /// <remarks>
        /// Возвращается копия последнего заполненного кассового чека,
        /// а в кассовом аппарате заводится новый пустой чек.
        /// </remarks>
        /// <returns>Чек.</returns>
        public Check Save()
        {
            var check = (Check)CurrentCheck.Clone();
            _checks.Add(new Check());
            return check;
        }

        /// <summary>
        /// Приведение объекта к строке.
        /// </summary>
        /// <returns>Название кассового аппарата.</returns>
        public override string ToString()
        {
            return Name;
        }
    }
}
```

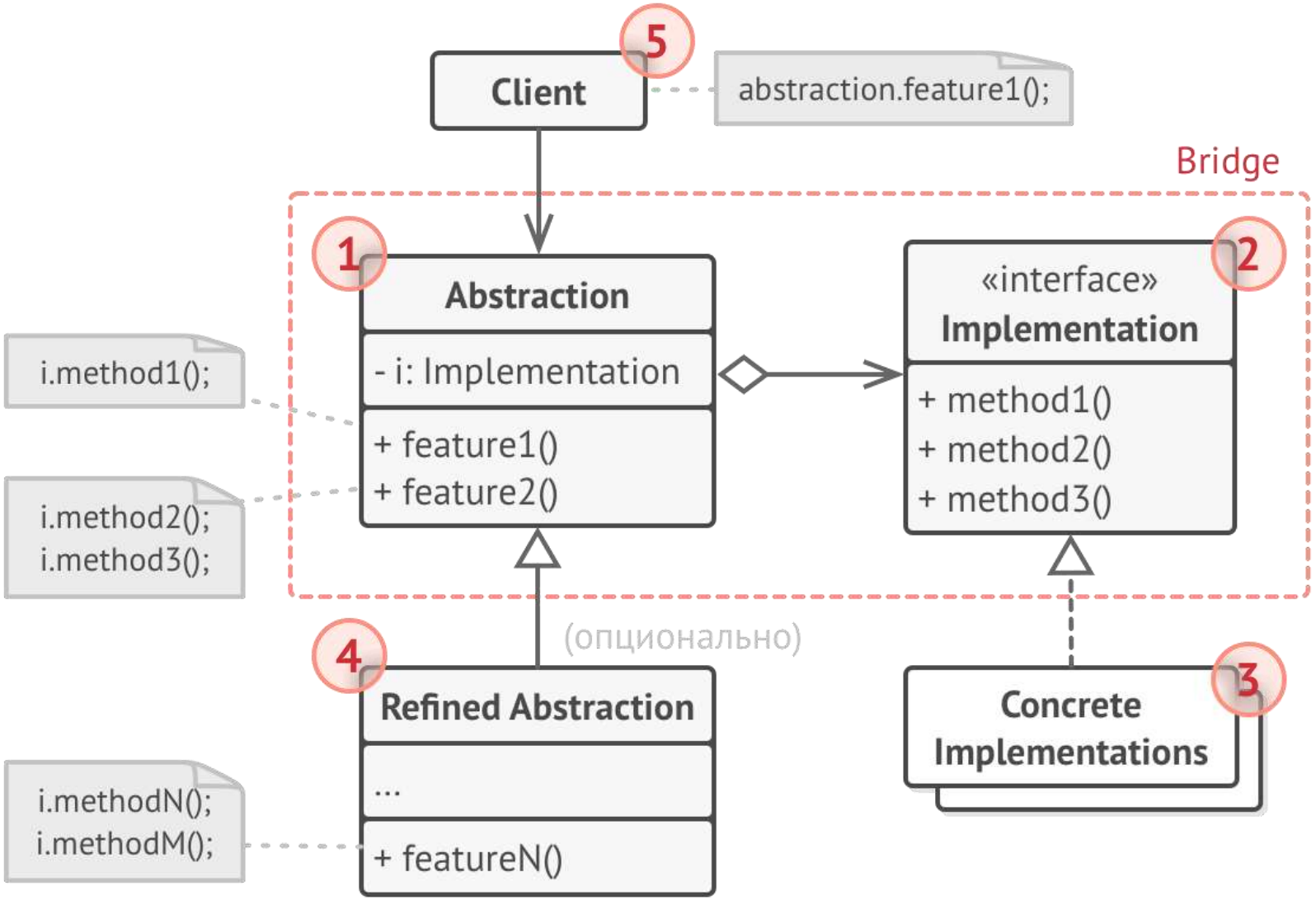
Пример

```
namespace Adapter
{
    /// <summary>
    /// Адаптер для работы с иностранными кассовыми аппаратами как с обычными.
    /// </summary>
    public class ForeignCashMachineAdapter : ICashMachine
    {
        /// <summary>
        /// Иностранный кассовый аппарат.
        /// </summary>
        private ForeignCashMachine.ForeignCashMachine _foreignCashMachine;

        /// <inheritdoc />
        public string Number => _foreignCashMachine.Name;

        /// <inheritdoc />
        public IEnumerable<Product> Products
        {
            get
            {
                var productsTuple = _foreignCashMachine.CurrentCheck.Products;
                var products = productsTuple.Select(s => new Product(s.Name, Convert.ToDecimal(s.Price)));
                return products;
            }
        }

        /// <summary>
        /// Создать экземпляр адаптера иностранного кассового аппарата под обычный.
        /// </summary>
        /// <param name="foreignCashMachine">Иностранный кассовый аппарат.</param>
        public ForeignCashMachineAdapter(ForeignCashMachine.ForeignCashMachine foreignCashMachine)
        {
            _foreignCashMachine = foreignCashMachine;
        }

        /// <inheritdoc />
        public void AddProduct(Product product)
        {
            _foreignCashMachine.Add(product.Name, Convert.ToDouble(product.Price));
        }

        /// <inheritdoc />
        public string PrintCheck()
        {
            var check = _foreignCashMachine.Save();
            var checkText = GetCheckText(check);
            Save(checkText);
            return checkText;
        }

        /// <inheritdoc />
        public void Save(string checkText)
        {
            using (var sr = new StreamWriter("checks2.txt", true, Encoding.Default))
            {
                sr.WriteLine(checkText);
            }
        }

        /// <summary>
        /// Сформировать текст чека для вывода на печать и сохранения в файл.
        /// </summary>
        /// <param name="check">Чек иностранного кассового аппарата.</param>
        /// <returns>Форматированный текст чека.</returns>
        private string GetCheckText(ForeignCashMachine.Check check)
        {
            var date = check.DateTime.ToString("dd MMMM yyyy HH:mm");
            var checkText = $"Кассовый чек от {date}\r\n";
            checkText += $"Идентификатор чека: {check.Number}\r\n";
            checkText += "Список товаров:\r\n";
            foreach (var (Name, Price) in check.Products)
            {
                checkText += $"{Name}\t\t\t{Price}\r\n";
            }
            var sum = check.Products.Sum(p => p.Price);
            checkText += $"ИТОГО\t\t\t{sum}\r\n";
            return checkText;
        }
    }
}
```

Мост (Bridge)

Разделяет один или несколько классов на две отдельные иерархии — абстракцию и реализацию, позволяя изменять их независимо друг от друга.

Структура


5
Client
abstraction.feature1();
Bridge
1
Abstraction
- i: Implementation
+ feature1()
+ feature2()
2
«interface»
Implementation
+ method1()
+ method2()
+ method3()
i.method1();
i.method2();
i.method3();
(опционально)
4
Refined Abstraction
...
+ featureN()
3
Concrete
Implementations
i.methodN();
i.methodM();

Когда применять?

- Когда надо избежать постоянной привязки абстракции к реализации.
- Когда наряду с реализацией надо изменять и абстракцию независимо друг от друга. То есть изменения в абстракции не должно привести к изменениям в реализации.

Пример

```
// Реализация устанавливает интерфейс для всех классов реализации. Он не
// должен соответствовать интерфейсу Абстракции. На практике оба интерфейса
// могут быть совершенно разными. Как правило, интерфейс Реализации
// предоставляет только примитивные операции, в то время как Абстракция
// определяет операции более высокого уровня, основанные на этих примитивах.
public interface IImplementation
{
    string OperationImplementation();
}
```

```
// Абстракция устанавливает интерфейс для «управляющей» части двух иерархий
// классов. Она содержит ссылку на объект из иерархии Реализации и
// делегирует ему всю настоящую работу.
class Abstraction
{
    protected IImplementation _implementation;

    public Abstraction(IImplementation implementation)
    {
        this._implementation = implementation;
    }

    public virtual string Operation()
    {
        return "Abstract: Base operation with:\n" +
            _implementation.OperationImplementation();
    }
}
```

Пример

```
// Можно расширить Абстракцию без изменения классов Реализации.
class ExtendedAbstraction : Abstraction
{
    public ExtendedAbstraction(IImplementation implementation) : base(implementation)
    {
    }

    public override string Operation()
    {
        return "ExtendedAbstraction: Extended operation with:\n" +
            base._implementation.OperationImplementation();
    }
}
```

Пример

```
// Каждая Конкретная Реализация соответствует определённой платформе и
// реализует интерфейс Реализации с использованием API этой платформы.
class ConcreteImplementationA : IImplementation
{
    public string OperationImplementation()
    {
        return "ConcreteImplementationA: The result in platform A.\n";
    }
}
```

```
class ConcreteImplementationB : IImplementation
{
    public string OperationImplementation()
    {
        return "ConcreteImplementationA: The result in platform B.\n";
    }
}
```

Пример

```
class Client
{
    // За исключением этапа инициализации, когда объект Абстракции
    // связывается с определённым объектом Реализации, клиентский код должен
    // зависеть только от класса Абстракции. Таким образом, клиентский код
    // может поддерживать любую комбинацию абстракции и реализации.
    public void ClientCode(Abstraction abstraction)
    {
        Console.Write(abstraction.Operation());
    }
}
```

Пример

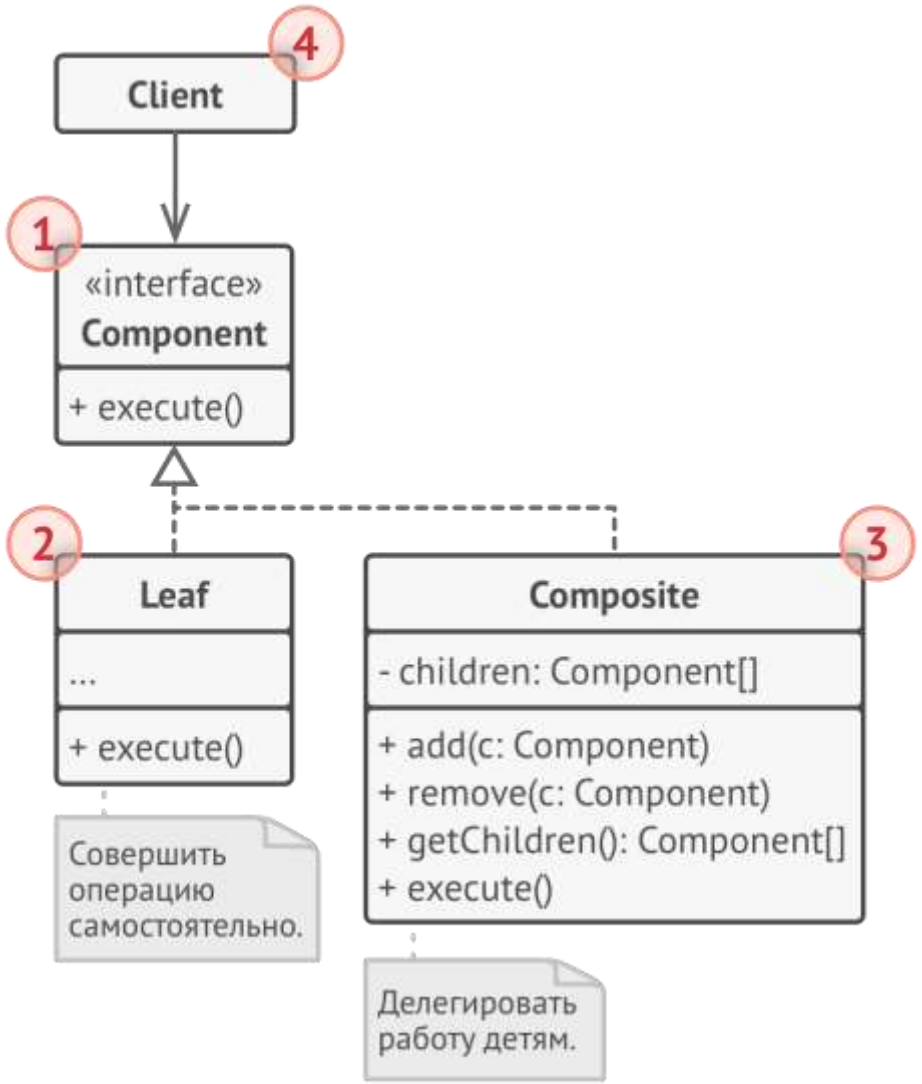
```
class Program
{
    static void Main(string[] args)
    {
        Client client = new Client();

        Abstraction abstraction;
        // Клиентский код должен работать с любой предварительно
        // сконфигурированной комбинацией абстракции и реализации.
        abstraction = new Abstraction(new ConcreteImplementationA());
        client.ClientCode(abstraction);

        Console.WriteLine();

        abstraction = new ExtendedAbstraction(new ConcreteImplementationB());
        client.ClientCode(abstraction);
    }
}
```

```
Abstract: Base operation with:
ConcreteImplementationA: The result in platform A.

ExtendedAbstraction: Extended operation with:
ConcreteImplementationA: The result in platform B.
```

Компоновщик (Composite)

Позволяет сгруппировать объекты в древовидную структуру, а затем работать с ними так, если бы это был единичный объект.

Структура


4
Client
1
«interface»
Component
+ execute()
2
Leaf
...
+ execute()
Совершить операцию самостоятельно.
3
Composite
- children: Component[]
+ add(c: Component)
+ remove(c: Component)
+ getChildren(): Component[]
+ execute()
Делегировать работу детям.

Когда применять?

- 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.

Пример

```
// Базовый класс Компонент объявляет общие операции как для простых, так и
// для сложных объектов структуры.
abstract class Component
{
    public Component() { }

    // Базовый Компонент может сам реализовать некоторое поведение по
    // умолчанию или поручить это конкретным классам, объявив метод,
    // содержащий поведение абстрактным.
    public abstract string Operation();

    // В некоторых случаях целесообразно определить операции управления
    // потомками прямо в базовом классе Компонент. Таким образом, вам не
    // нужно будет предоставлять  конкретные классы компонентов клиентскому
    // коду, даже во время сборки дерева объектов. Недостаток такого подхода
    // в том, что эти методы будут пустыми для компонентов уровня листа.
    public virtual void Add(Component component)
    {
        throw new NotImplementedException();
    }

    public virtual void Remove(Component component)
    {
        throw new NotImplementedException();
    }

    // Вы можете предоставить метод, который позволит клиентскому коду
    // понять, может ли компонент иметь вложенные объекты.
    public virtual bool IsComposite()
    {
        return true;
    }
}
```

Пример

```
// Класс Лист представляет собой конечные объекты структуры. Лист не может
// иметь вложенных компонентов.
//
// Обычно объекты Листьев выполняют фактическую работу, тогда как объекты
// Контейнера лишь делегируют работу своим подкомпонентам.
class Leaf : Component
{
    public override string Operation()
    {
        return "Leaf";
    }

    public override bool IsComposite()
    {
        return false;
    }
}
```

```
// Класс Контейнер содержит сложные компоненты, которые могут иметь
// вложенные компоненты. Обычно объекты Контейнеры делегируют фактическую
// работу своим детям, а затем «суммируют» результат.
class Composite : Component
{
    protected List<Component> _children = new List<Component>();

    public override void Add(Component component)
    {
        this._children.Add(component);
    }

    public override void Remove(Component component)
    {
        this._children.Remove(component);
    }

    // Контейнер выполняет свою основную логику особым образом. Он проходит
    // рекурсивно через всех своих детей, собирая и суммируя их результаты.
    // Поскольку потомки контейнера передают эти вызовы своим потомкам и так
    // далее,  в результате обходится всё дерево объектов.
    public override string Operation()
    {
        int i = 0;
        string result = "Branch(";

        foreach (Component component in this._children)
        {
            result += component.Operation();
            if (i != this._children.Count - 1)
            {
                result += "+";
            }
            i++;
        }

        return result + ")";
    }
}
```

Пример

```
class Client
{
    // Клиентский код работает со всеми компонентами через базовый
    // интерфейс.
    public void ClientCode(Component leaf)
    {
        Console.WriteLine($"RESULT: {leaf.Operation()}\n");
    }

    // Благодаря тому, что операции управления потомками объявлены в базовом
    // классе Компонента, клиентский код может работать как с простыми, так
    // и со сложными компонентами, вне зависимости от их конкретных классов.
    public void ClientCode2(Component component1, Component component2)
    {
        if (component1.IsComposite())
        {
            component1.Add(component2);
        }

        Console.WriteLine($"RESULT: {component1.Operation()}");
    }
}
```

Пример

```
    class Program
    {
        static void Main(string[] args)
        {
            Client client = new Client();

            // Таким образом, клиентский код может поддерживать простые
            // компоненты-листья...
            Leaf leaf = new Leaf();
            Console.WriteLine("Client: I get a simple component:");
            client.ClientCode(leaf);

            // ...а также сложные контейнеры.
            Composite tree = new Composite();
            Composite branch1 = new Composite();
            branch1.Add(new Leaf());
            branch1.Add(new Leaf());
            Composite branch2 = new Composite();
            branch2.Add(new Leaf());
            tree.Add(branch1);
            tree.Add(branch2);
            Console.WriteLine("Client: Now I've got a composite tree:");
            client.ClientCode(tree);

            Console.Write("Client: I don't need to check the components classes even when managing the tree:\n");
            client.ClientCode2(tree, leaf);
        }
    }
}
```

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

Client: Now I've got a composite tree:
RESULT: Branch(Branch(Leaf+Leaf)+Branch(Leaf))

Client: I don't need to check the components classes even when managing the tree:
RESULT: Branch(Branch(Leaf+Leaf)+Branch(Leaf)+Leaf)
```

Спасибо за внимание!