Structural Design Patterns

📌 **Назначение:** Организуют отношения между классами и объектами для формирования более крупных структур.

🔌 1. Adapter (Адаптер)

🔷 **Назначение:** Позволяет несовместимым интерфейсам работать вместе.

👥 **Участники:**

- **Target** - целевой интерфейс, который ожидает клиент. То, с чем должен уметь работать клиент
- **Adaptee** - существующий класс с полезной логикой, но **несовместимым интерфейсом**
- **Adapter** - обёртка, реализует интерфейс **Target**, а внутри использует **Adaptee.** Преобразует вызовы клиента в формат, понятный **Adaptee**

🛠️ **Примеры:**

✔️ Интеграция нового платежного сервиса со старым кодом

✔️ Подключение современных датчиков к legacy-системе

💡 **Преимущества:**

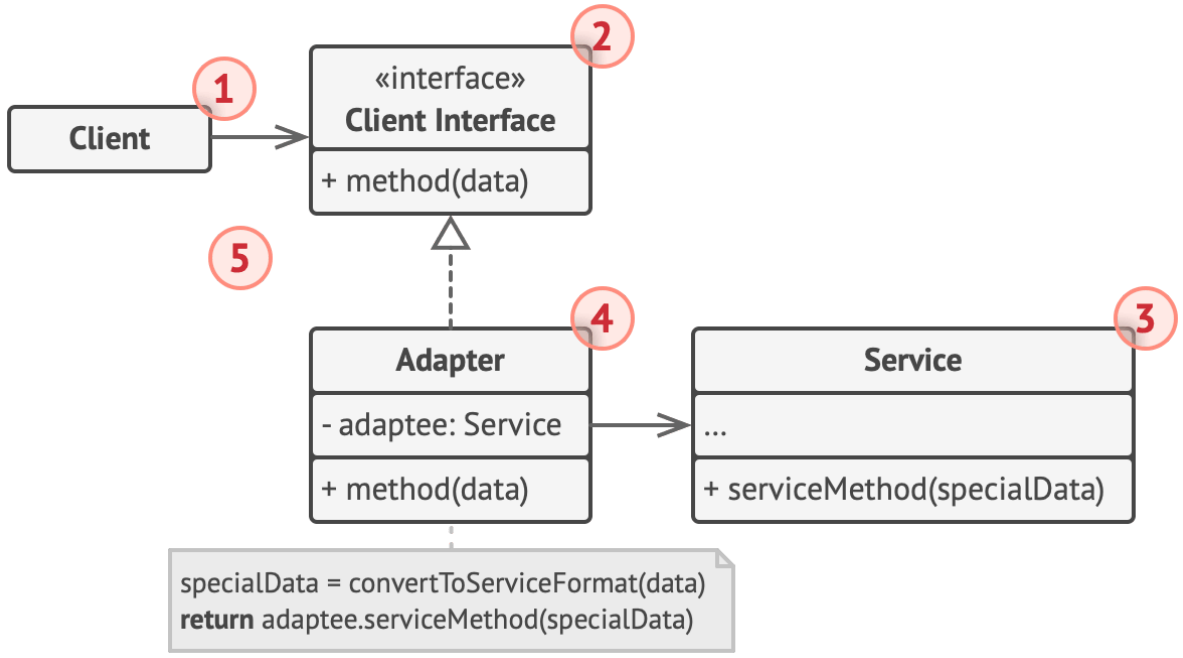
✅ Позволяет работать с несовместимыми интерфейсами

✅ Сохраняет принцип открытости/закрытости

⚠ Недостатки:

❌ Увеличивает сложность кода дополнительными классами

❌ Может снижать производительность из-за двойного преобразования




```
// Несовместимый класс
public class EuropeanPlug {
    public string GetPower() => "220V";
}

// Целевой интерфейс
public interface IAmericanSocket {
    string Get120VPower();
}

// Адаптер
public class PlugAdapter : IAmericanSocket {
    private EuropeanPlug _plug;

    public PlugAdapter(EuropeanPlug plug) => _plug = plug;
```

```
    public string Get120VPower() => $"Converted {_plug.GetPower()} to 120
V";
}

// Использование:
var euroPlug = new EuropeanPlug();
var adapter = new PlugAdapter(euroPlug);
Console.WriteLine(adapter.Get120VPower()); // "Converted 220V to 120V"
```

🎁 2. Decorator (Декоратор)

🔷 **Назначение:** Динамически добавляет объектам новое поведение не меняя исходный код.

👥 **Участники:**

- **Component** - базовый интерфейс
- **ConcreteComponent** - конкретная реализация компонента. Объект, который **будет расширяться декораторами**
- **Decorator** - абстрактный декоратор, реализует интерфейс **Component** и содержит ссылку на оборачиваемый компонент. Делегирует ему работу
- **ConcreteDecorator** - конкретный декоратор, добавляющий **новое поведение**

🛠️ **Примеры:**

✔️ Добавление логирования/кеширования к существующим методам

✔️ Динамическое изменение поведения UI-элементов

💡 **Преимущества:**

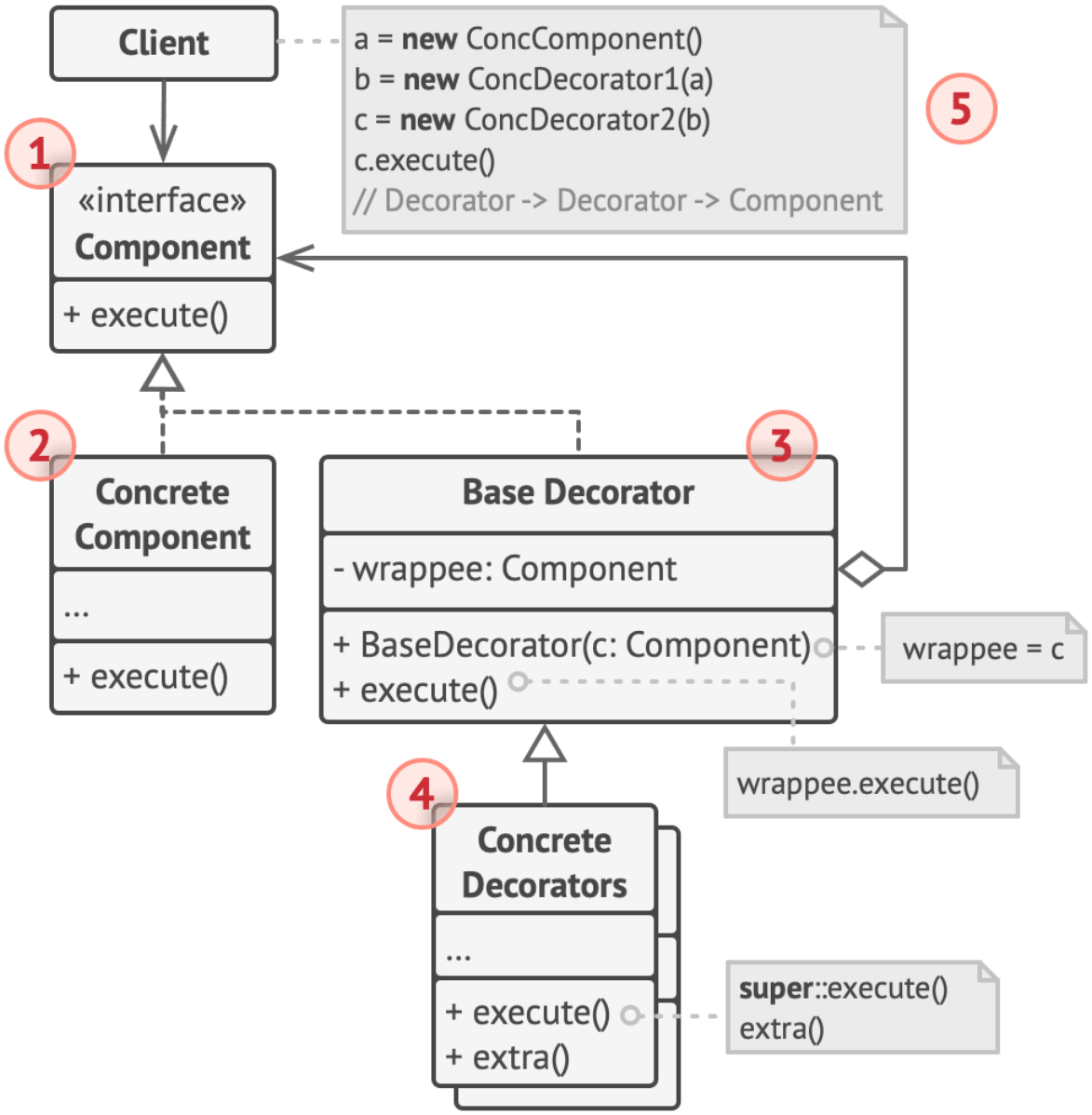
✅ Гибче наследования (можно комбинировать декораторы)

✅ Позволяет добавлять функционал без изменения исходного кода

⚠️ **Недостатки:**

❌ Много мелких классов усложняют архитектуру

❌ Трудно поддерживать порядок декорирования




```
public interface ICoffee {
    string GetDescription();
    double GetCost();
}

public class SimpleCoffee : ICoffee {
    public string GetDescription() ⇒ "Простой кофе";
    public double GetCost() ⇒ 1.0;
}

public abstract class CoffeeDecorator : ICoffee {
```

```
    protected ICoffee _coffee;

    public CoffeeDecorator(ICoffee coffee) => _coffee = coffee;

    public virtual string GetDescription() => _coffee.GetDescription();
    public virtual double GetCost() => _coffee.GetCost();
}

public class MilkDecorator : CoffeeDecorator {
    public MilkDecorator(ICoffee coffee) : base(coffee) {}

    public override string GetDescription() => _coffee.GetDescription() + ", м
олоко";
    public override double GetCost() => _coffee.GetCost() + 0.5;
}

// Использование:
ICoffee coffee = new MilkDecorator(new SimpleCoffee());
Console.WriteLine($"{coffee.GetDescription()} - ${coffee.GetCost()}");
// "Простой кофе, молоко - $1.5"
```

☯️ 3. Facade (Фасад)

🔷 **Назначение:** Предоставляет простой интерфейс к сложной системе.

👥 **Участники:**

- **Facade** - упрощённый интерфейс, который **оборачивает вызовы** к нескольким подсистемам
- **Subsystems** - сложные компоненты системы

🛠️ **Примеры:**

✔️ Упрощенный API для сложной CRM-системы

✔️ Игровой движок для начинающих разработчиков

💡 **Преимущества:**

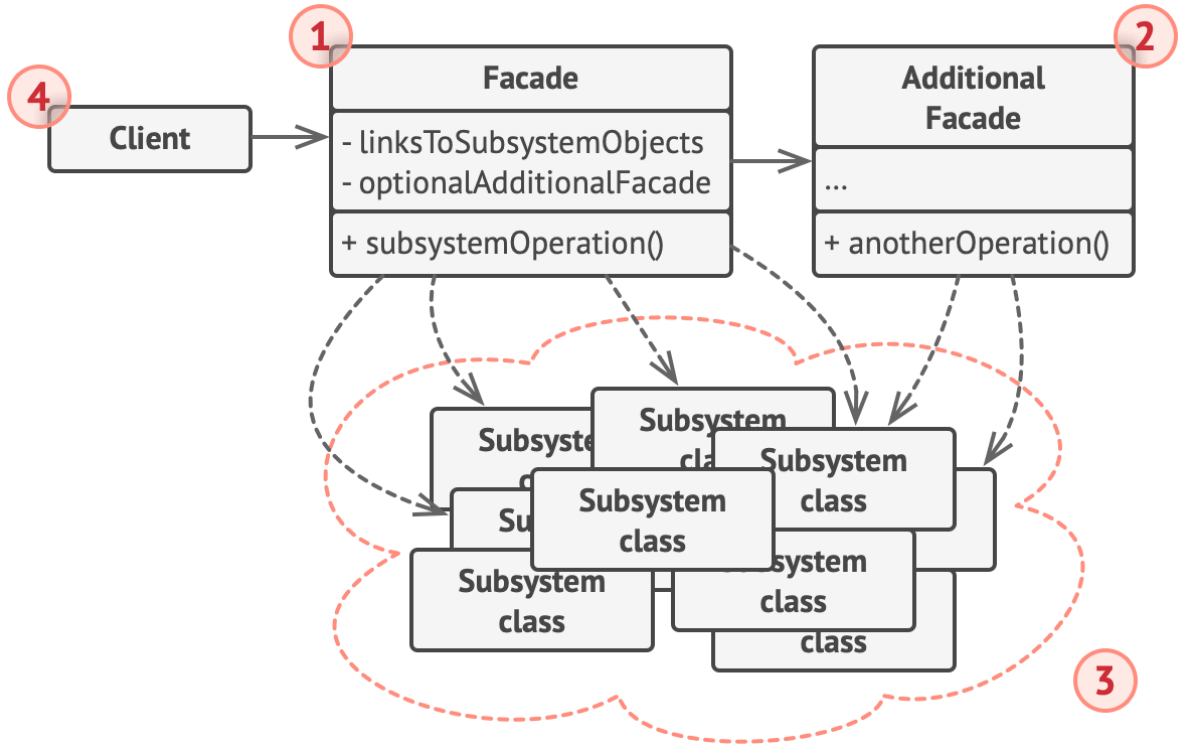
✅ Снижает сложность для клиентского кода

✅ Изолирует изменения в подсистемах

⚠️ **Недостатки:**

❌ Может стать "божественным объектом"

❌ Ограничивает гибкость для продвинутых сценариев




```
// Сложные подсистемы
public class CPU {
    public void Start() => Console.WriteLine("CPU started");
}

public class Memory {
    public void Load() => Console.WriteLine("Memory loaded");
}

// Фасад
public class ComputerFacade {
    private CPU _cpu;
    private Memory _memory;

    public ComputerFacade() {
        _cpu = new CPU();
        _memory = new Memory();
```

```
    }

    public void Start() {
        _cpu.Start();
        _memory.Load();
        Console.WriteLine("Computer ready!");
    }
}

// Использование:
var computer = new ComputerFacade();
computer.Start();
```

🎭 4. Proxy (Заместитель)

🔷 **Назначение:** Контролирует доступ к реальному объекту, добавляя дополнительную логику.

👥 **Участники:**

- **ISubject** - общий интерфейс для Proxy и RealSubject, чтобы клиент мог использовать их взаимозаменяемо
- **RealSubject** - реальный объект
- **Proxy** - Хранит ссылку на **RealSubject**, перехватывает вызовы клиента, выполняет дополнительную логику и (при необходимости) делегирует выполнение **RealSubject**

🛠️ **Примеры:**

✔️ Ленивая загрузка изображений в веб-приложении

✔️ Защита доступа к базе данных

💡 **Преимущества:**

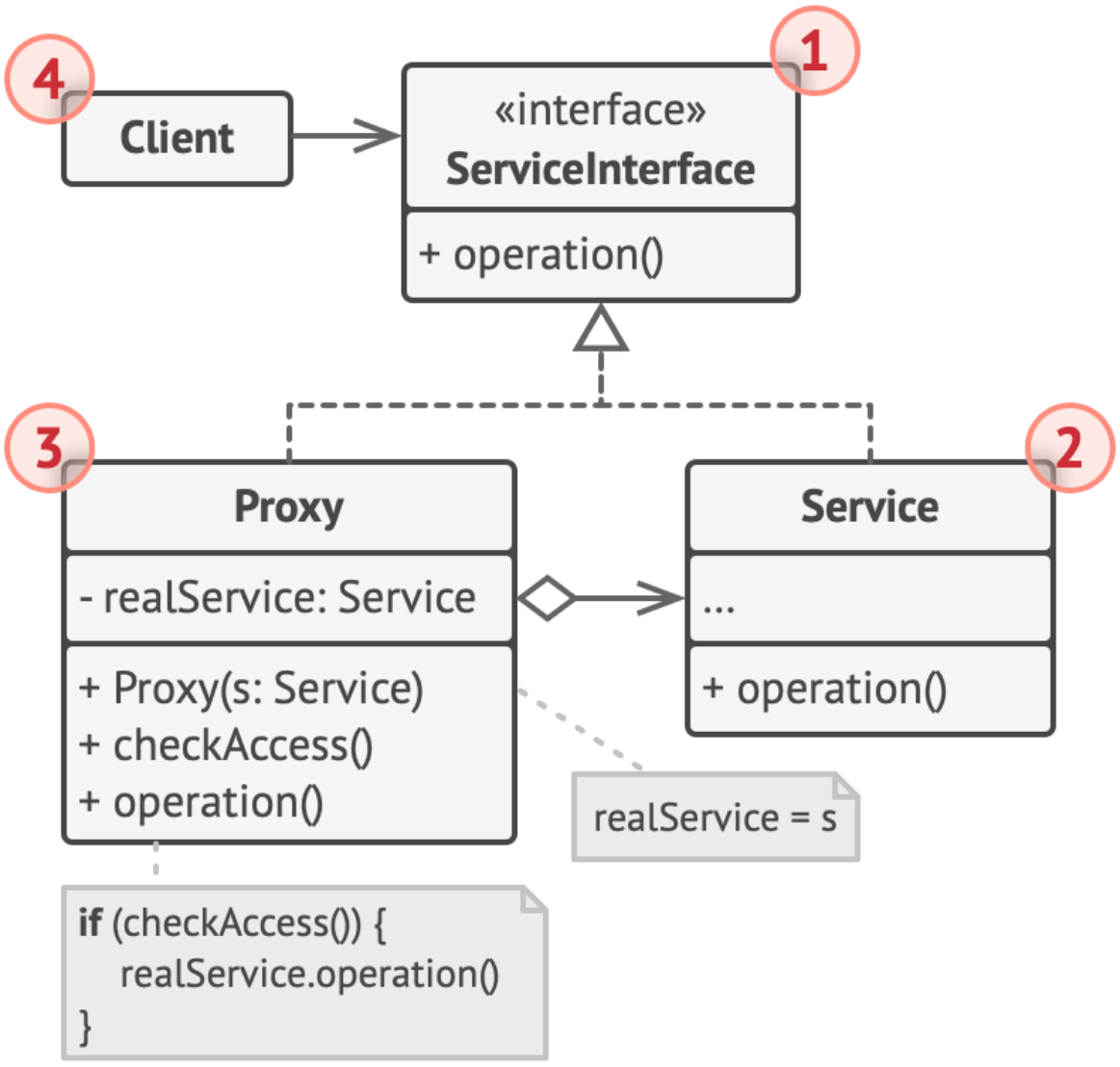
✅ Контроль над созданием и доступом к ресурсам

✅ Позволяет добавлять дополнительную логику (кеширование)

⚠️ **Недостатки:**

❌ Может вводить неочевидные задержки

❌ Усложняет отслеживание вызовов методов




```
public interface IImage {
    void Display();
}

public class RealImage : IImage {
    private string _filename;

    public RealImage(string filename) {
        _filename = filename;
        LoadFromDisk();
    }

    private void LoadFromDisk() => Console.WriteLine($"Загрузка {_filenam
```

```
e}");

    public void Display() => Console.WriteLine($"Показ {_filename}");
}

public class ProxyImage : IImage {
    private RealImage _realImage;
    private string _filename;

    public ProxyImage(string filename) => _filename = filename;

    public void Display() {
        if (_realImage == null)
            _realImage = new RealImage(_filename);
        _realImage.Display();
    }
}

// Использование:
IImage image = new ProxyImage("photo.jpg");
image.Display(); // Загрузит и покажет
image.Display(); // Только покажет (уже загружено)
```

🌉 5. Bridge (Мост)

🔷 **Назначение:**

Отделяет **абстракцию** от её **реализации**, позволяя им развиваться **независимо друг от друга**.

👥 **Участники:**

- **Abstraction** - интерфейс или абстрактный класс, определяющий **высокоуровневую логику.** Хранит ссылку на объект **Implementor**
- **RefinedAbstraction** - **расширение Abstraction**, добавляет или уточняет поведение.
- **Implementor** - интерфейс или абстрактный класс для **реализации**, может быть отличным от интерфейса **Abstraction**. Определяет базовые операции

- **ConcreteImplementor** - конкретная реализация интерфейса **Implementor**

🛠️ **Примеры:**

✔️ Кроссплатформенный UI (одна логика для Windows/macOS)

✔️ Драйверы устройств с разными версиями API

💡 **Преимущества:**

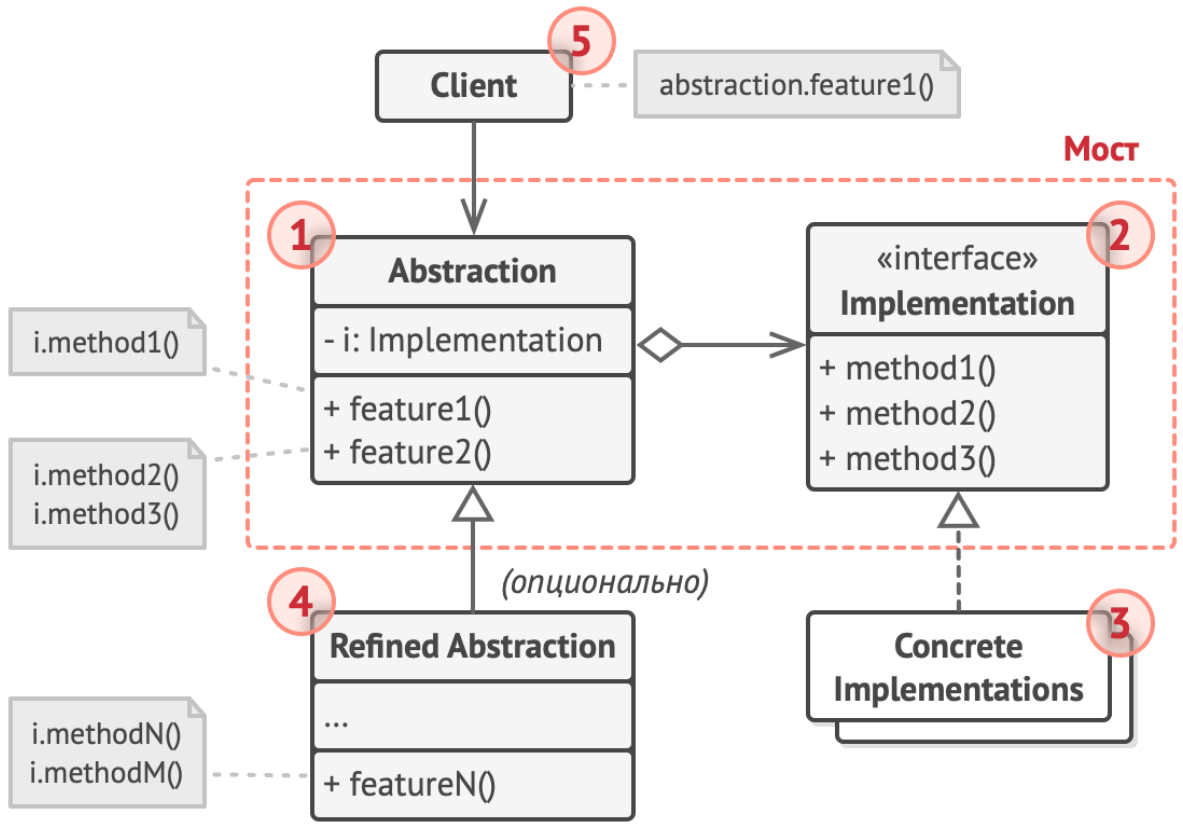
✅ Позволяет независимо изменять абстракцию и реализацию

✅ Избегает "взрывного роста" подклассов

⚠️ **Недостатки:**

❌ Усложняет архитектуру примитивных систем

❌ Требует чёткого разделения логики на абстракцию/реализацию




```
// Реализация
public interface IRenderer {
    string RenderButton(string text);
    string RenderCheckbox(bool isChecked);
```

```
}

public class HtmlRenderer : IRenderer {
    public string RenderButton(string text) => $"<button>{text}</button>";
    public string RenderCheckbox(bool isChecked) => $"<input type='check
box' {(isChecked ? "checked" : "")}>";
}

// Абстракция
public abstract class UIControl {
    protected IRenderer _renderer;

    protected UIControl(IRenderer renderer) => _renderer = renderer;

    public abstract string Render();
}

public class Button : UIControl {
    private string _text;

    public Button(IRenderer renderer, string text) : base(renderer) => _text = t
ext;

    public override string Render() => _renderer.RenderButton(_text);
}

// Использование:
var htmlButton = new Button(new HtmlRenderer(), "Login");
Console.WriteLine(htmlButton.Render()); // "<button>Login</button>"
```

🌿 6. Composite (Компоновщик)

🔷 **Назначение:** Позволяет работать с древовидными структурами

👥 **Участники:**

- **Component** - базовый интерфейс для **всех объектов** в структуре. Определяет операции, которые применимы как к простым элементам **(Leaf),** так и к контейнерам **(Composite)**

- **Leaf** - **простой элемент**, не содержащий других компонентов. Реализует поведение, определённое в Component
- **Composite** - контейнер, может содержать другие **Component** (как **Leaf,** так и другие **Composite**). Реализует логику добавления, удаления и обхода вложенных компонентов.

🛠️ **Примеры:**

✔️ Графические редакторы (группировка фигур)

✔️ Системы сборки проектов (задачи/подзадачи)

💡 **Преимущества:**

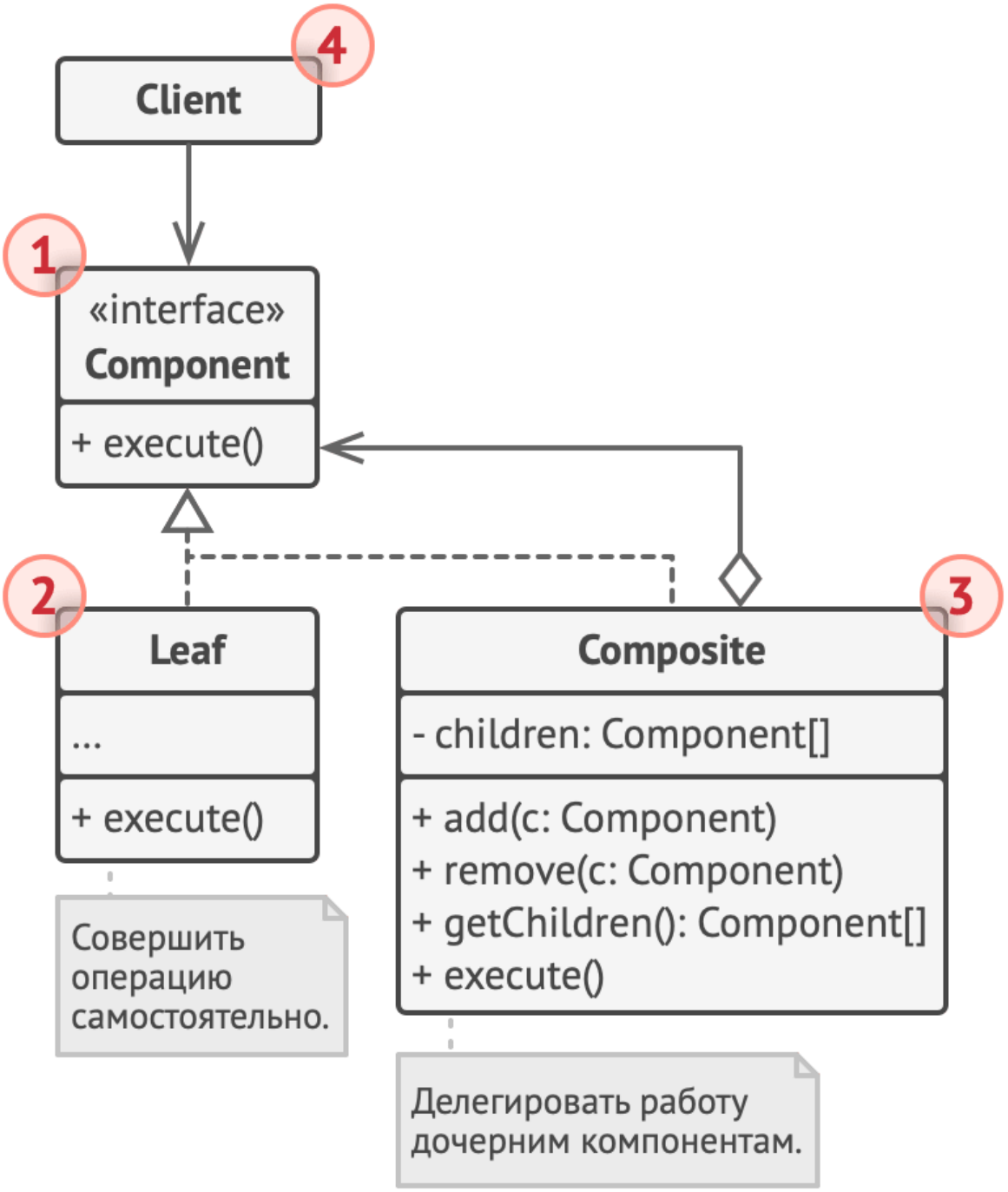
✅ Единообразная работа с простыми и составными объектами

✅ Упрощает добавление новых типов компонентов

⚠️ **Недостатки:**

❌ Может привести к излишне общей структуре интерфейсов

❌ Сложно ограничить типы компонентов в композиции




```
public abstract class FileSystemComponent {
    public string Name { get; }
    protected FileSystemComponent(string name) => Name = name;
    public abstract void Display(int depth = 0);
}

public class File : FileSystemComponent {
```

```
    public File(string name) : base(name) {}

    public override void Display(int depth = 0) {
        Console.WriteLine($"{new string('-', depth)} {Name}");
    }
}

public class Directory : FileSystemComponent {
    private List<FileSystemComponent> _children = new List<FileSystemCo
mponent>();

    public Directory(string name) : base(name) {}

    public void Add(FileSystemComponent component) => _children.Add(co
mponent);

    public override void Display(int depth = 0) {
        Console.WriteLine($"{new string('-', depth)} [DIR] {Name}");
        foreach (var child in _children)
            child.Display(depth + 2);
    }
}

// Использование:
var root = new Directory("Root");
var docs = new Directory("Documents");
docs.Add(new File("resume.pdf"));
root.Add(docs);
root.Display();
/*
- [DIR] Root
  - [DIR] Documents
    - resume.pdf
*/
```

📌 **Когда использовать:**

- Иерархии объектов (меню, файловые системы)

- Когда клиент должен единообразно работать с простыми и составными объектами

🪶 7. Flyweight (Легковес)

🔷 **Назначение:** Экономит память, разделяя общие состояния между объектами

👥 **Участники:**

- **Flyweight** - интерфейс определяющий метод для работы с объектом и его **внешним состоянием,** которое **передаётся извне**
- **ConcreteFlyweight** - **разделяемый объект** с **внутренним (общим)** состоянием
- **FlyweightFactory** - **фабрика-кэш**, управляет созданием и повторным использованием Flyweight-объектов. При запросе возвращает существующий легковес или создаёт новый, если такого нет
- **Client** - передаёт внешнее состояние объекту **Flyweight** и **не хранит дублирующие данные.**

🛠️ **Примеры:**

✔️ Рендеринг леса в игре (повторное использование моделей деревьев)

✔️ Текстовый процессор (оптимизация хранения символов)

💡 **Преимущества:**

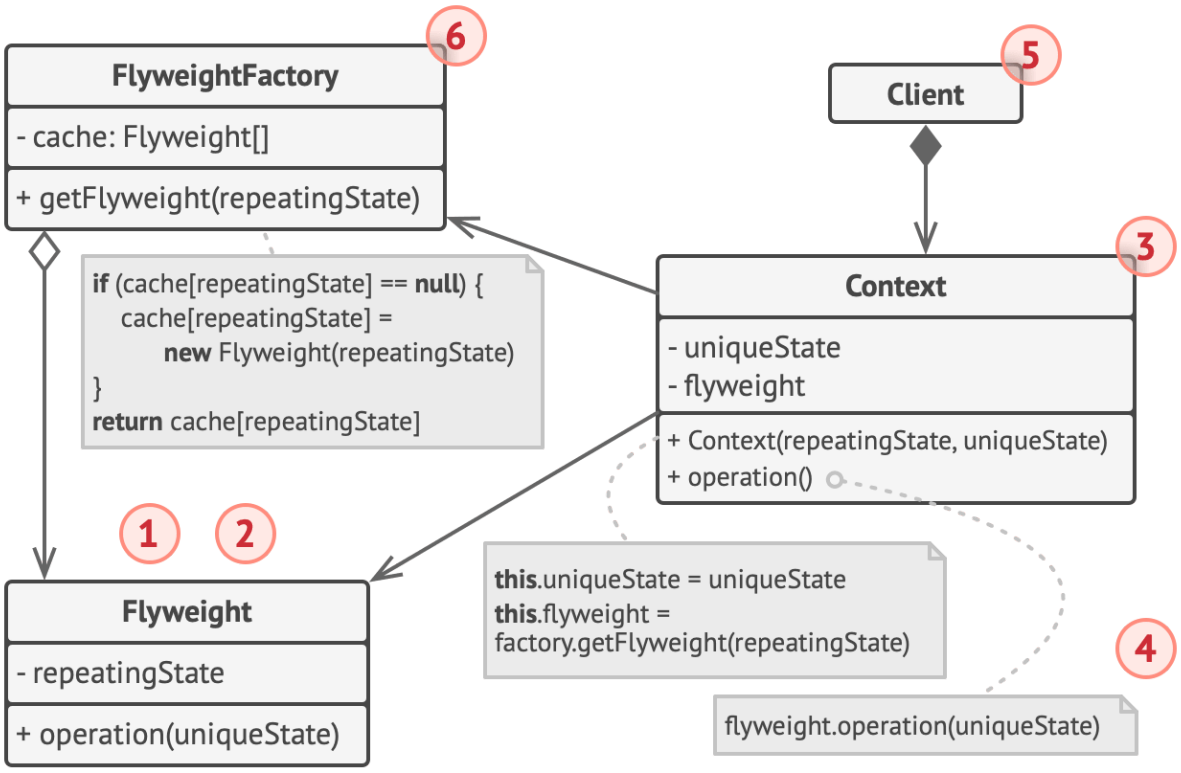
✅ Экономит память при массовом создании объектов

✅ Уменьшает нагрузку на сборщик мусора

⚠️ **Недостатки:**

❌ Требует тщательного разделения состояния на внутреннее/внешнее

❌ Может усложнить код из-за фабрики и управления состоянием




```
public class CharacterStyle {
    public string Font { get; }
    public int Size { get; }
    public ConsoleColor Color { get; }

    public CharacterStyle(string font, int size, ConsoleColor color) {
        Font = font;
        Size = size;
        Color = color;
    }
}

public class CharacterStyleFactory {
    private Dictionary<string, CharacterStyle> _styles = new Dictionary<strin
g, CharacterStyle>();

    public CharacterStyle GetStyle(string font, int size, ConsoleColor color) {
        string key = $"{font}-{size}-{color}";
        if (!_styles.ContainsKey(key))
            _styles[key] = new CharacterStyle(font, size, color);
```

```
        return _styles[key];
    }
}

public class Character {
    private char _symbol;
    private CharacterStyle _style;

    public Character(char symbol, CharacterStyle style) {
        _symbol = symbol;
        _style = style;
    }

    public void Print() {
        Console.ForegroundColor = _style.Color;
        Console.Write(_symbol);
        Console.ResetColor();
    }
}

// Использование:
var factory = new CharacterStyleFactory();
var style1 = factory.GetStyle("Arial", 12, ConsoleColor.Red);
var style2 = factory.GetStyle("Arial", 12, ConsoleColor.Red); // Вернёт суще
ствующий стиль

var chars = new List<Character> {
    new Character('H', style1),
    new Character('i', style2)
};

foreach (var c in chars)
    c.Print(); // Оба символа используют один экземпляр стиля
```

📊 Сравнение в одной таблице

Паттерн	Решаемая проблема	Ключевое преимущество	Когда использовать
Adapter	Несовместимые интерфейсы	Интеграция старых/новых систем	При работе с legacy-кодом или внешними API
Bridge	Жёсткая связь абстракции и реализации	Независимое развитие компонентов	Для кроссплатформенных UI или драйверов
Composite	Работа с древовидными структурами	Единообразие обработки простых и составных объектов	Файловые системы, UI-деревья
Decorator	Необходимость динамического расширения функционала	Гибкость без наследования	Добавление прав доступа, логирования
Facade	Сложность взаимодействия с подсистемой	Упрощение клиентского кода	Работа со сложными библиотеками/ фреймворками
Flyweight	Большое количество похожих объектов	Экономия памяти	Игровые движки, текстовые процессоры
Proxy	Необходимость контроля доступа к объекту	Ленивая загрузка, кэширование	Загрузка тяжелых ресурсов, защита API

🛠️ Как выбрать паттерн?

1. **Adapter** → Когда нужно "заставить работать" несовместимые интерфейсы
2. **Bridge** → **При частых изменениях реализации**
3. **Composite** → **Для древовидных структур**
4. **Decorator** → **Для динамического расширения**
5. **Facade** → **Для упрощения сложных API**
6. **Flyweight** → При работе с множеством похожих объектов
7. **Proxy** → Для контроля доступа**/загрузки**

💡 Как запомнить?

- **Adapter** - переходник для розеток
- **Decorator** - слоёный торт (добавляем слои)
- **Facade** - пульт от сложной системы
- **Proxy** - секретарь (фильтрует доступ)
- **Bridge** - мост между абстракцией и реализацией (как пульт и TV)
- **Composite** - дерево папок/файлов
- **Flyweight** - общие буквы в книге (разделяем шрифты)