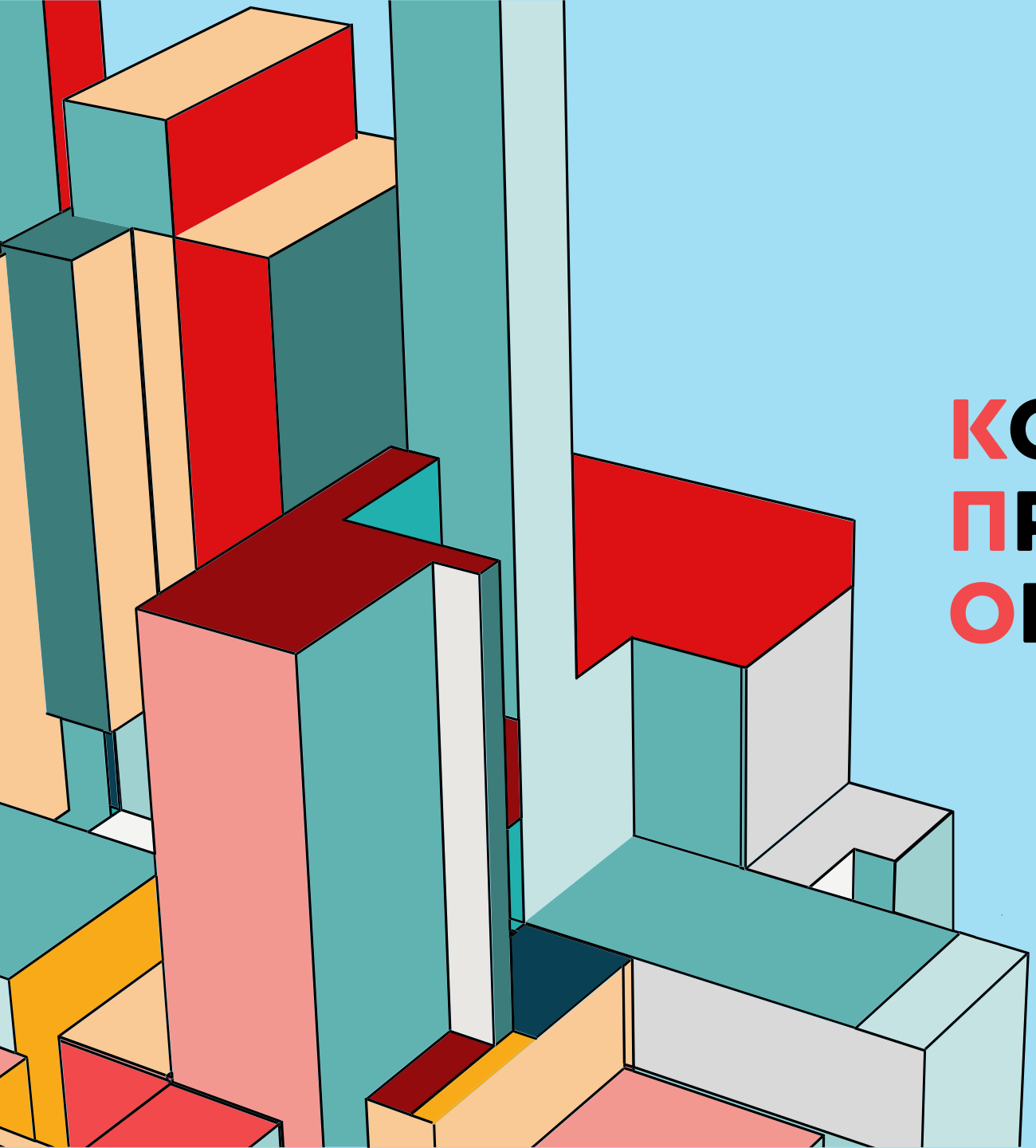


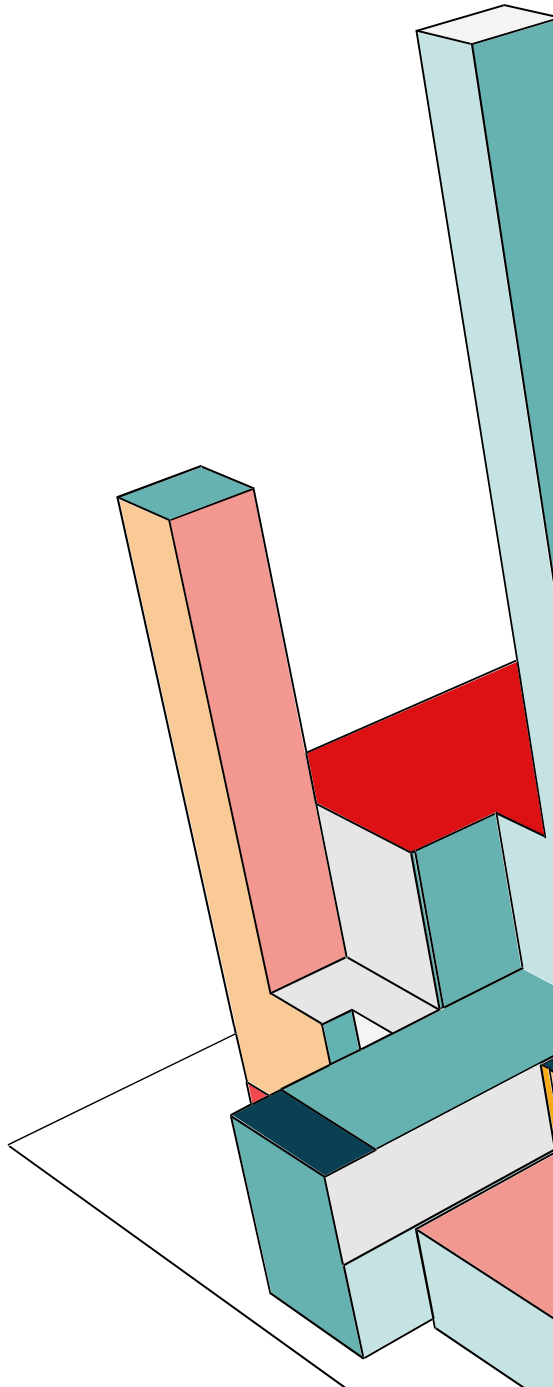
КОНСТРУИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

ПЛАН ЛЕКЦИИ № 6

Структурные паттерны проектирования

1. **Декоратор** (Decorator)
2. **Адаптер** (Adapter)
3. **Фасад** (Facade)
4. **Заместитель** (Прокси)

Компоновщик (Composite)
Мост (Bridge)
Приспособленец (Flyweight)

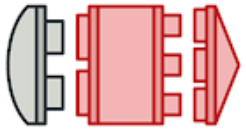


ВИДЫ ПАТТЕРНОВ

- **Структурные** паттерны показывают различные способы построения связей между объектами.

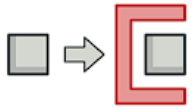
- **Поведенческие** паттерны заботятся об эффективной коммуникации между объектами.
- **Порождающие** паттерны беспокоятся о гибком создании объектов без внесения в программу лишних зависимостей.

СТРУКТУРНЫЕ ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ



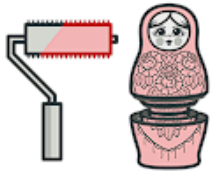
Адаптер
Adapter

Позволяет объектам с несовместимыми интерфейсами работать вместе.



Заместитель
Proxy

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.



Декоратор
Decorator

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

Фасад
Facade

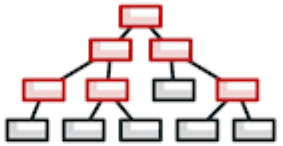
Предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку.

Рассматривает, как классы и объекты образуют более крупные структуры - более сложные по характеру классы и объекты.

Мост

Bridge

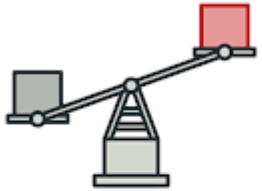
Разделяет один или несколько классов на две отдельные иерархии — абстракцию и реализацию, позволяя изменять их независимо друг от друга.



Компоновщик

Composite

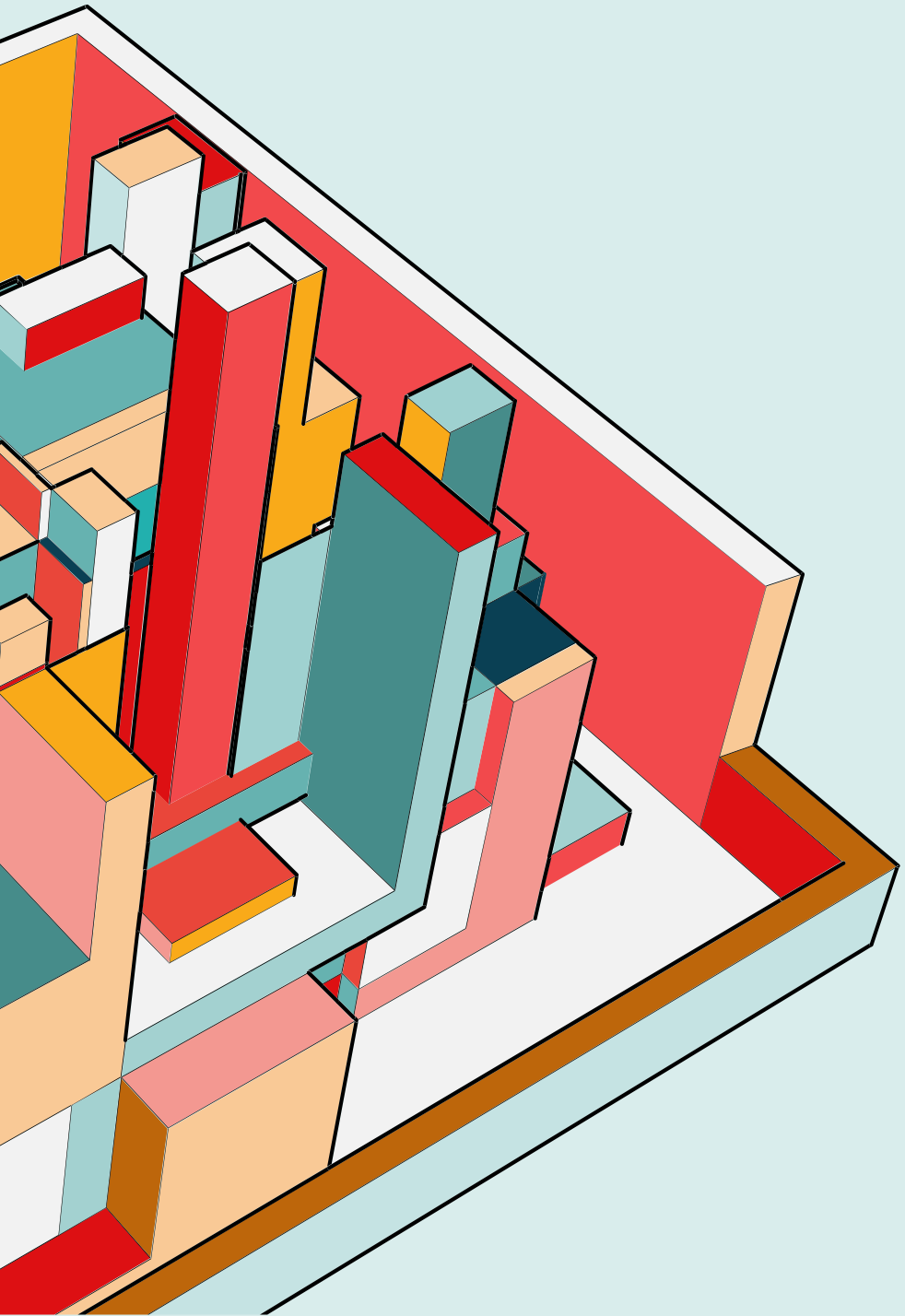
Позволяет сгруппировать множество объектов в древовидную структуру, а затем работать с ней так, как будто это единичный объект.



Легковес

Flyweight

Позволяет вместить бóльшее количество объектов в отведённую оперативную память. Легковес экономит память, разделяя общее состояние объектов между собой, вместо хранения одинаковых данных в каждом объекте.

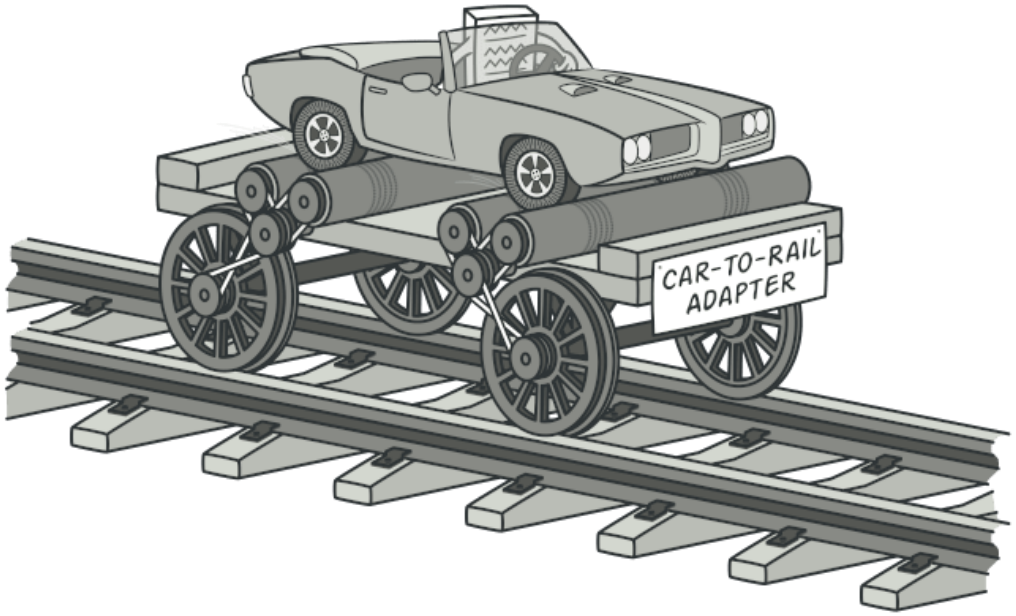


СТРУКТУРНЫЕ ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

Adapter

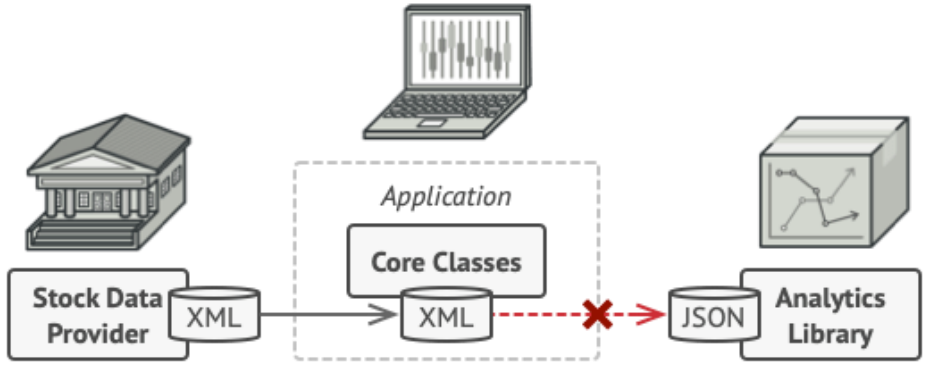
АДАПТЕР

Позволяет объектам с несовместимыми интерфейсами работать вместе.



Вы делаете приложение для торговли на бирже. **Ваше приложение скачивает** биржевые котировки из нескольких источников **в XML**, а затем рисует красивые графики.

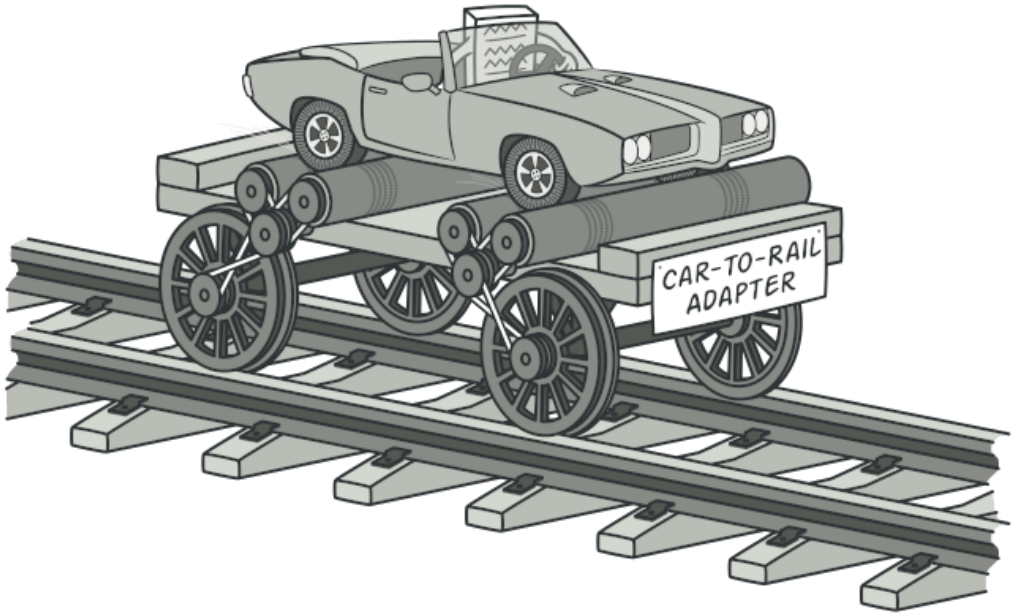
В какой-то момент вы решаете улучшить приложение, **применив стороннюю библиотеку** аналитики.



Подключить стороннюю библиотеку не выйдет из-за несовместимых форматов данных.

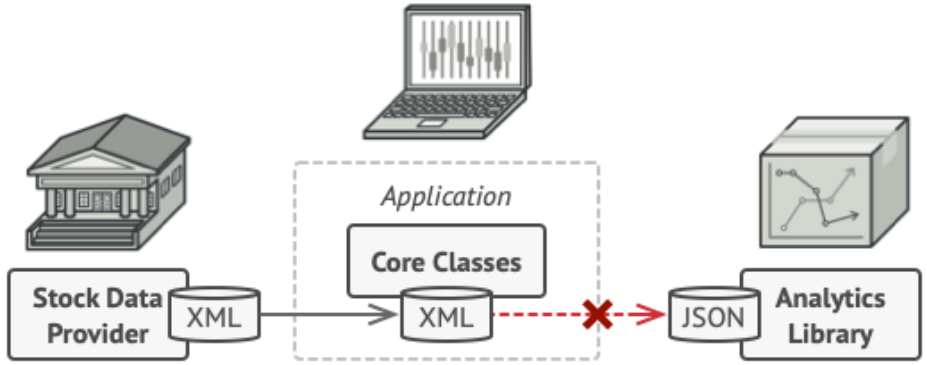
АДАПТЕР

Позволяет объектам с несовместимыми интерфейсами работать вместе.



Вы делаете приложение для торговли на бирже. **Ваше приложение скачивает** биржевые котировки из нескольких источников **в XML**, а затем рисует красивые графики.

В какой-то момент вы решаете улучшить приложение, **применив стороннюю библиотеку** аналитики.



Подключить стороннюю библиотеку не выйдет из-за несовместимых форматов данных.

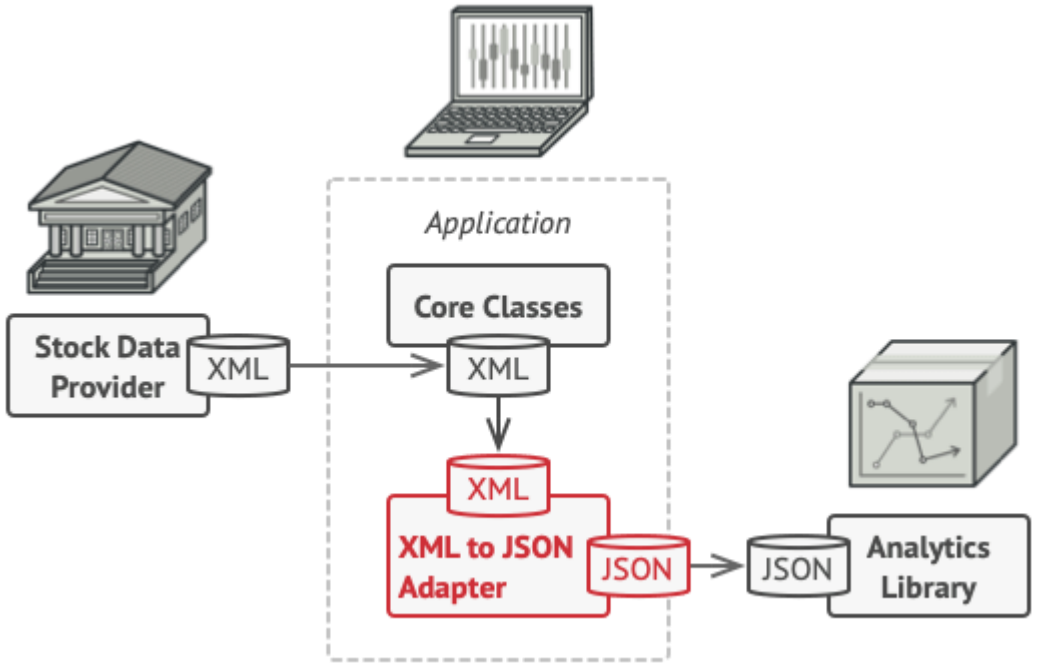
Вы можете создать адаптер – объект-переводчик, который трансформирует интерфейс одного объекта к другому.

РЕШЕНИЕ

Адаптеры могут не только переводить данные из одного формата в другой, но и помогать объектам с разными интерфейсами работать сообща.

Это работает так:

1. Адаптер имеет интерфейс, который совместим с одним из объектов.
2. Поэтому этот объект может свободно вызывать методы адаптера.
3. Адаптер получает эти вызовы и перенаправляет их второму объекту.



Иногда возможно создать даже двухсторонний адаптер, который работал бы в обе стороны

СТРУКТУРА

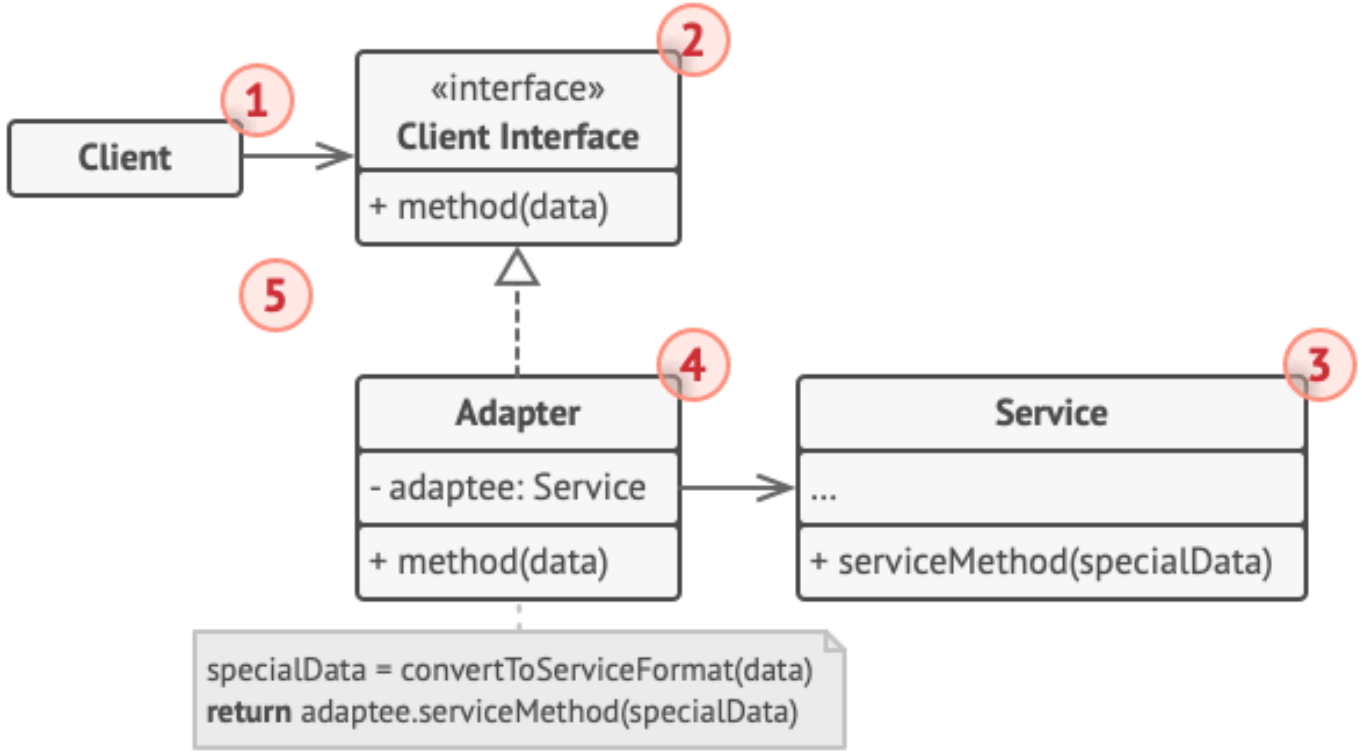
1. **Клиент** — содержит существующую бизнес-логику программы.

2. **Клиентский интерфейс** описывает протокол, через который клиент может работать с другими классами.

3. **Сервис** — сторонний класс, который мы не можем использовать.

4. **Адаптер** — это класс, который может одновременно работать и с клиентом, и с сервисом. Он реализует клиентский интерфейс и содержит ссылку на объект сервиса

5. Работая с адаптером через интерфейс, **клиент не привязывается к конкретному классу адаптера**.



Эта реализация использует агрегацию:
объект адаптера «оборачивает»,
то есть содержит ссылку на служебный объект.

```
class Client
{
    public void Request(Target target)
    {
        target.Request();
    }
}

// класс, к которому надо адаптировать
class Target
{
    public virtual void Request() { }
}
```

```
// Адаптер
class Adapter : Target
{
    private Adaptee adaptee = new Adaptee();

    public override void Request()
    {
        adaptee.SpecificRequest();
    }
}

// Адаптируемый класс
class Adaptee
{
    public void SpecificRequest() { }
}
```

РАССМОТРИМ НА ПРИМЕРЕ

Допустим, у нас есть путешественник, который **путешествует на машине**.

Но в какой-то момент ему приходится передвигаться **по пескам** пустыни,
где **он не может ехать** на машине. Зато он может **использовать** для передвижения **верблюда**.

Однако в классе путешественника **использование класса верблюда не предусмотрено**,
поэтому нам **надо использовать адаптер**

```
// интерфейс транспорта
interface ITransport
{
    void Drive();
}
// класс машины
class Auto : ITransport
{
    public void Drive() ...
}
```

```
// интерфейс животного
interface IAnimal
{
    void Move();
}
// класс верблюда
class Camel : IAnimal
{
    public void Move() ...
}
```

```
class Driver
{
    public void Travel(ITransport transport)
    {
        transport.Drive();
    }
}
```

```
// интерфейс транспорта
interface ITransport
{
    void Drive();
}
// класс машины
class Auto : ITransport
{
    public void Drive() ...
}
```

```
class Driver
{
    public void Travel(ITransport transport)
    {
        transport.Drive();
    }
}
```

```
// интерфейс животного
interface IAnimal
{
    void Move();
}
// класс верблюда
class Camel : IAnimal
{
    public void Move() ...
}
```

```
// Адаптер от Camel к ITransport
class CamelToTransportAdapter : ITransport
{
    Camel camel;
    public CamelToTransportAdapter(Camel c)
    {
        camel = c;
    }

    public void Drive()
    {
        camel.Move();
    }
}
```

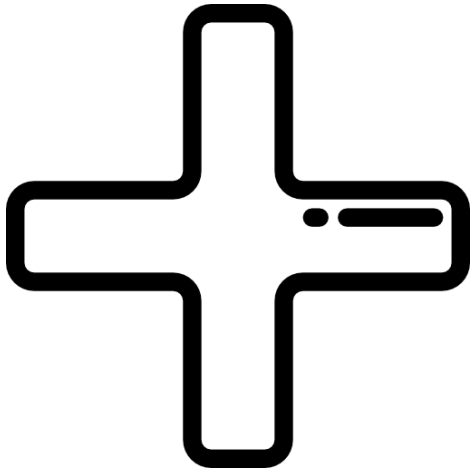
```
// путешественник
Driver driver = new Driver();
// машина
Auto auto = new Auto();

// отправляемся в путешествие
driver.Travel(auto);

// встретились пески, надо использовать верблюда
Camel camel = new Camel();

// используем адаптер
ITransport camelTransport = new CamelToTransportAdapter(camel);

// продолжаем путь по пескам пустыни
driver.Travel(camelTransport);
```

ПРИМЕНИМОСТЬ

- Когда вы хотите использовать сторонний класс, но его интерфейс не соответствует остальному коду приложения.

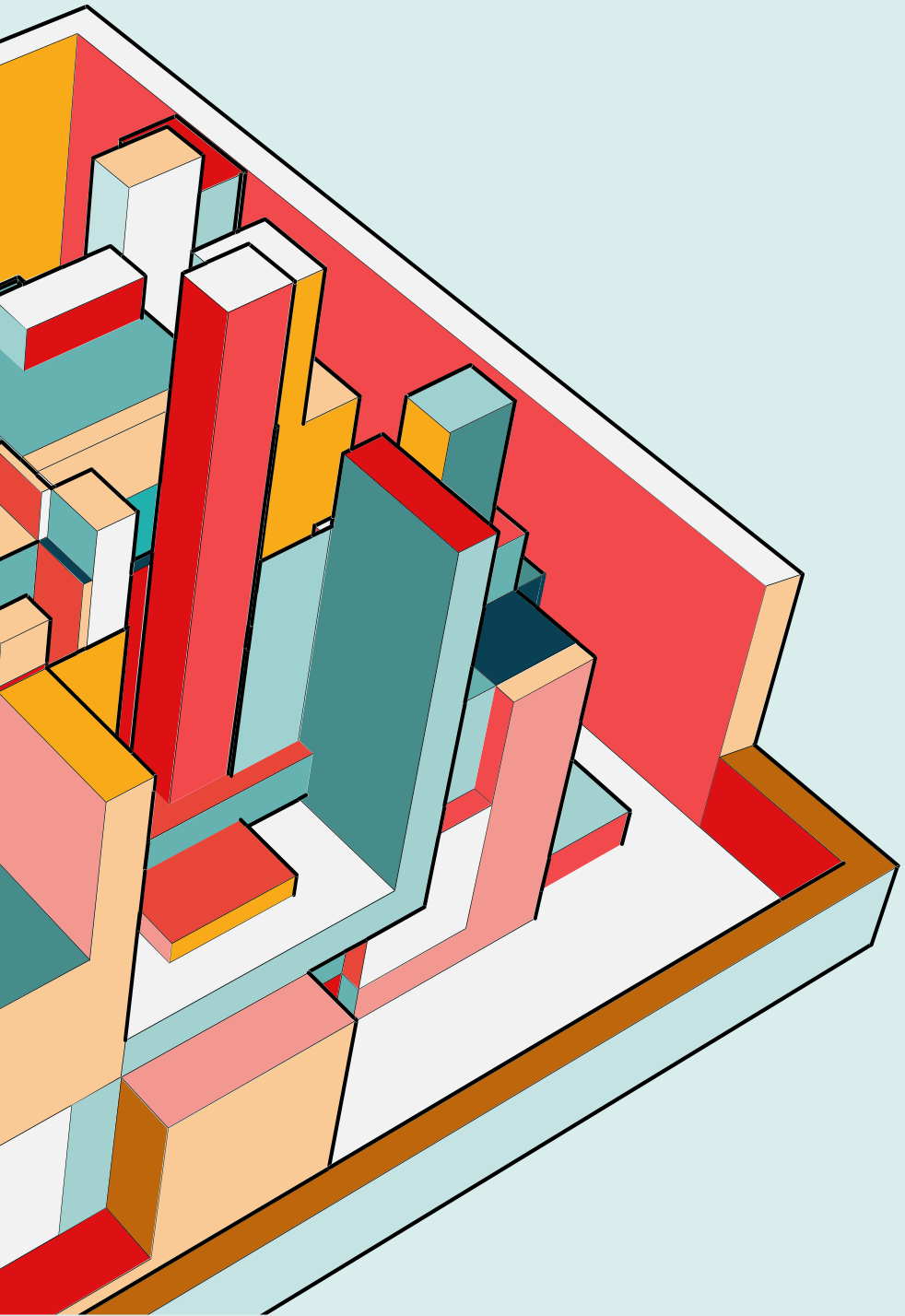
- Когда вам нужно использовать несколько существующих подклассов, но в них не хватает какой-то общей функциональности, причём расширить суперкласс вы не можете.

ПРЕИМУЩЕСТВА И НЕДОСТАТКИ



Отделяет и скрывает от клиента подробности преобразования различных интерфейсов.

Усложняет программу за счёт дополнительных классов.

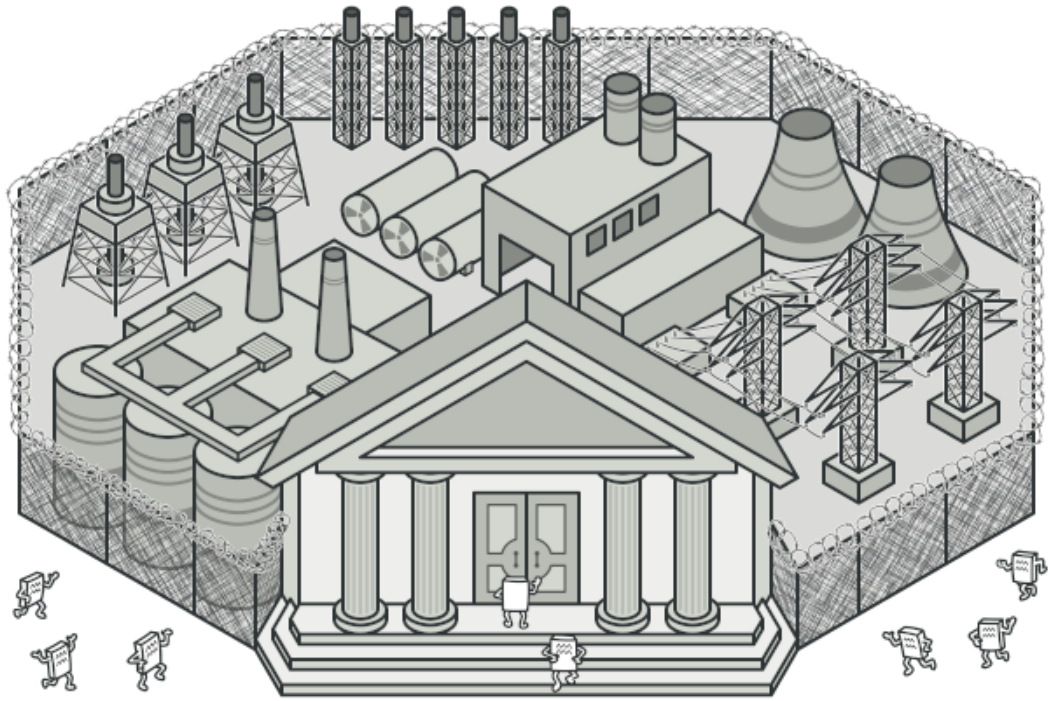


СТРУКТУРНЫЕ ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

Facade

ФАСАД

Предоставляет простой интерфейс к сложной системе классов.



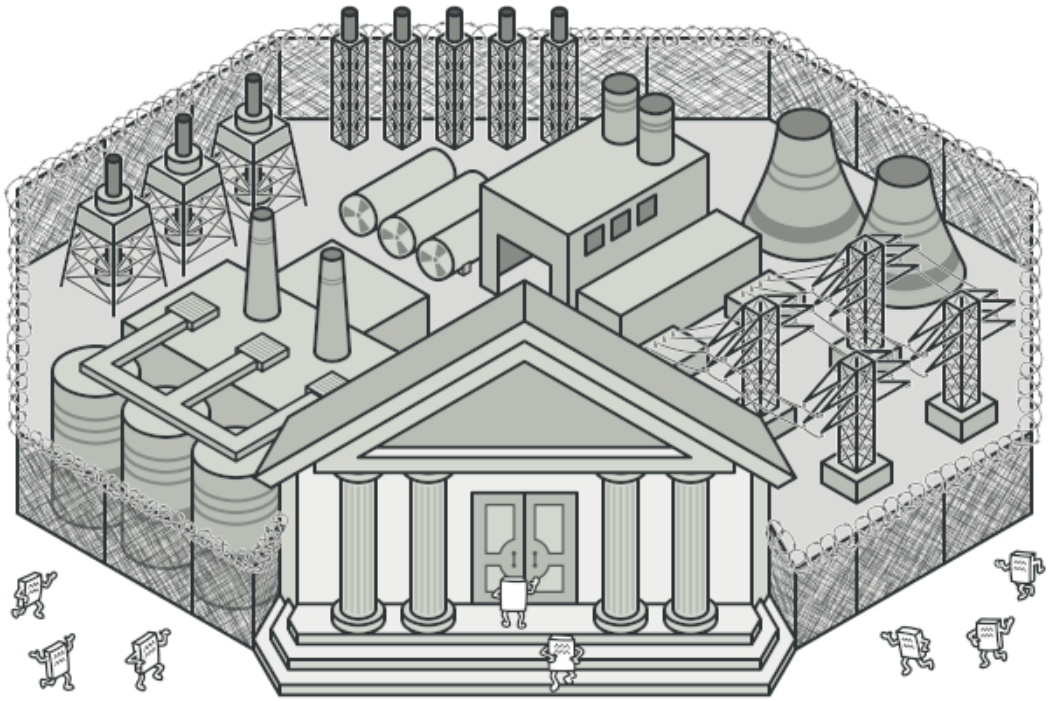
Вашему коду приходится работать с большим количеством объектов?

Вы должны самостоятельно инициализировать эти объекты, следить за правильным порядком зависимостей и так далее?

Такой код довольно сложно понимать и поддерживать.

ФАСАД

Предоставляет простой интерфейс к сложной системе классов.



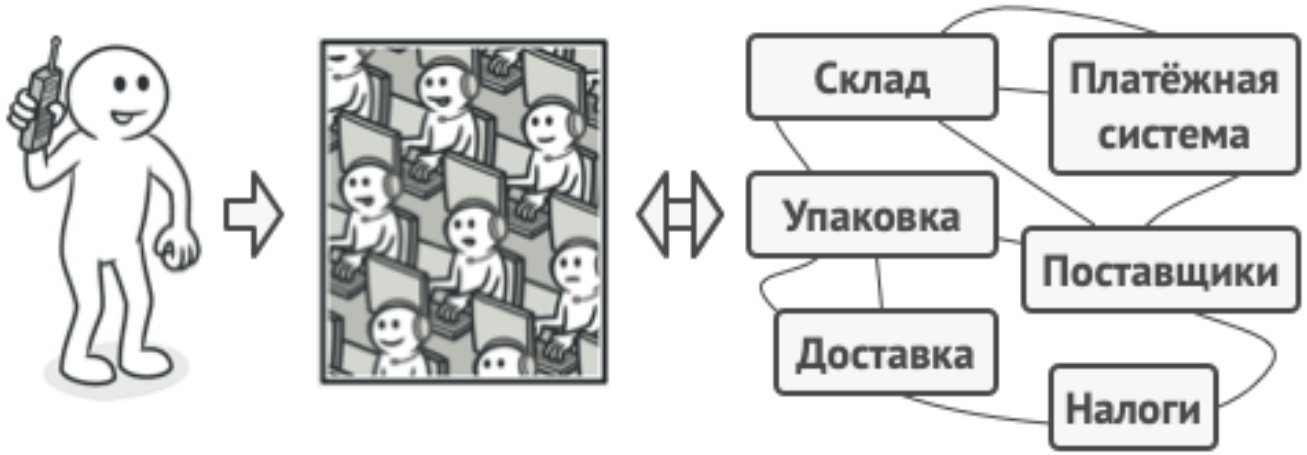
Вашему коду приходится работать с большим количеством объектов?

Вы должны самостоятельно инициализировать эти объекты, следить за правильным порядком зависимостей и так далее?

Такой код довольно сложно понимать и поддерживать.

Фасад может иметь урезанный интерфейс, но он предоставляет именно те фичи, которые нужны клиенту, и скрывает все остальные.

ПРИМЕР ТЕЛЕФОННОГО ЗАКАЗА



Фасад полезен, если вы используете какую-то сложную библиотеку со множеством подвижных частей, но вам нужна только часть её возможностей.

СТРУКТУРА

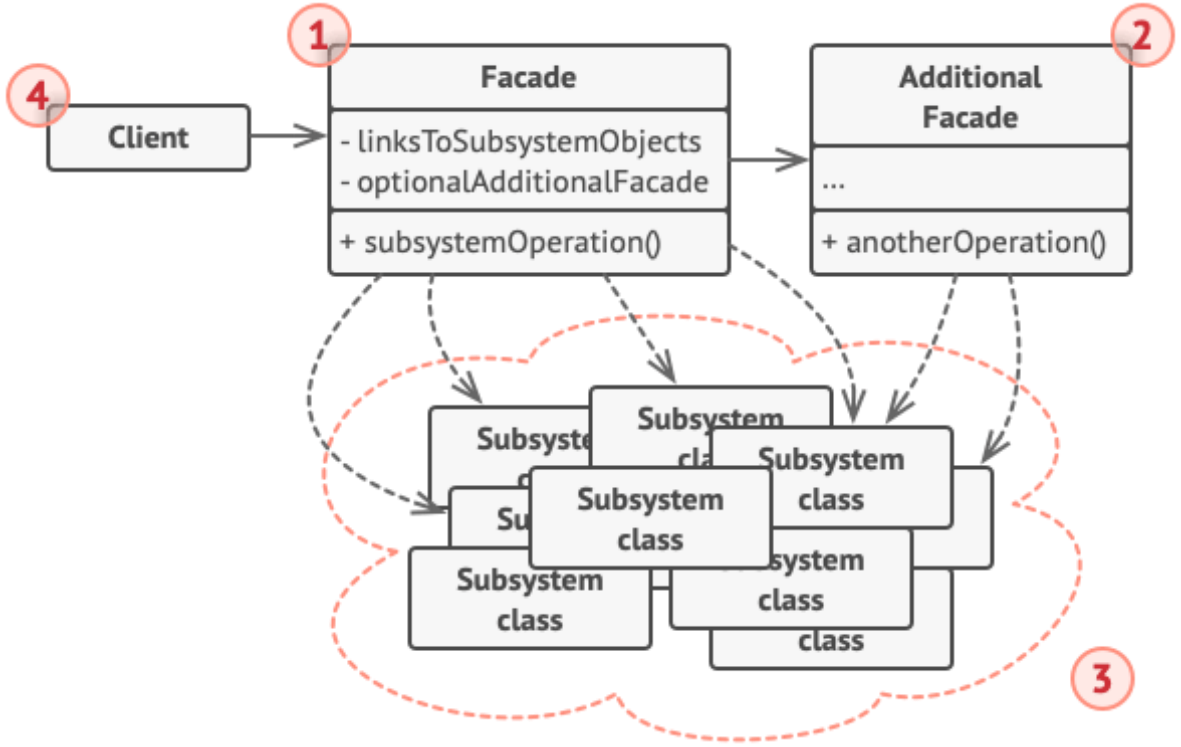
1. **Фасад** предоставляет быстрый доступ к определённой функциональности подсистемы.

2. **Дополнительный фасад** можно ввести, чтобы не «захламлять» единственный фасад.

3. **Сложная подсистема –** сложная, нечего добавить.

Классы подсистемы не знают о существовании фасада и работают друг с другом напрямую.

4. **Клиент** использует фасад вместо прямой работы с объектами сложной подсистемы.



```
public class SubsystemA
{
    public void A1()
    { }
}
public class SubsystemB ...
public class SubsystemC ...

public class Facade
{
    SubsystemA subsystemA;
    SubsystemB subsystemB;
    SubsystemC subsystemC;

    public Facade(SubsystemA sa, SubsystemB sb, SubsystemC sc) ...

    public void Operation1()
    {
        subsystemA.A1();
        subsystemB.B1();
        subsystemC.C1();
    }
    public void Operation2()
    {
        subsystemB.B1();
        subsystemC.C1();
    }
}
```

```
public class SubsystemA
{
    public void A1()
    { }
}
public class SubsystemB ...
public class SubsystemC ...

public class Facade
{
    SubsystemA subsystemA;
    SubsystemB subsystemB;
    SubsystemC subsystemC;

    public Facade(SubsystemA sa, SubsystemB sb, SubsystemC sc) ...

    public void Operation1()
    {
        subsystemA.A1();
        subsystemB.B1();
        subsystemC.C1();
    }
    public void Operation2()
    {
        subsystemB.B1();
        subsystemC.C1();
    }
}
```

```
Facade facade = new Facade(
    new SubsystemA(),
    new SubsystemB(),
    new SubsystemC());

facade.Operation1();
facade.Operation2();
```

РАССМОТРИМ НА ПРИМЕРЕ

Интегрированная среда разработки представляет собой фасад, который скрывает всю сложность процесса компиляции и запуска приложения.

```
class TextEditor
{
    public void CreateCode() => Console.WriteLine("Написание кода");
    public void Save() => Console.WriteLine("Сохранение кода");
}

class Compiler
{
    public void Compile() => Console.WriteLine("Компиляция приложения");
}

class CLR
{
    public void Execute() => Console.WriteLine("Выполнение приложения");
    public void Finish() => Console.WriteLine("Завершение работы приложения");
}
```

```
class TextEditor
{
    public void CreateCode() => Console.WriteLine("Написание кода");
    public void Save() => Console.WriteLine("Сохранение кода");
}

class Compiler
{
    public void Compile() => Console.WriteLine("Компиляция приложения");
}

class CLR
{
    public void Execute() => Console.WriteLine("Выполнение приложения");
    public void Finish() => Console.WriteLine("Завершение работы приложения");
}
```

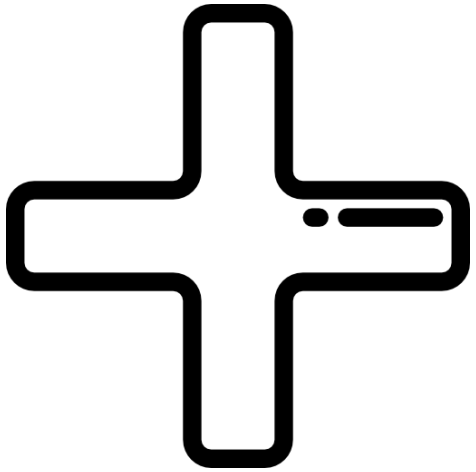
```
class VisualStudioFacade
{
    readonly TextEditor editor;
    readonly Compiler compiler;
    readonly CLR clr;

    public VisualStudioFacade(TextEditor editor, Compiler compiler, CLR clr) =>
        (this.editor, this.compiler, this.clr) = (editor, compiler, clr);

    public void Start()
    {
        editor.CreateCode();
        editor.Save();
        compiler.Compile();
        clr.Execute();
    }

    public void Stop() => clr.Finish();
}
```

```
var ide = new VisualStudioFacade(new TextEditor(), new Compiler(), new CLR());
new Programmer().CreateApplication(ide);

class Programmer
{
    public void CreateApplication(VisualStudioFacade ide)
    {
        ide.Start();
        ide.Stop();
    }
}
```

ПРИМЕНИМОСТЬ

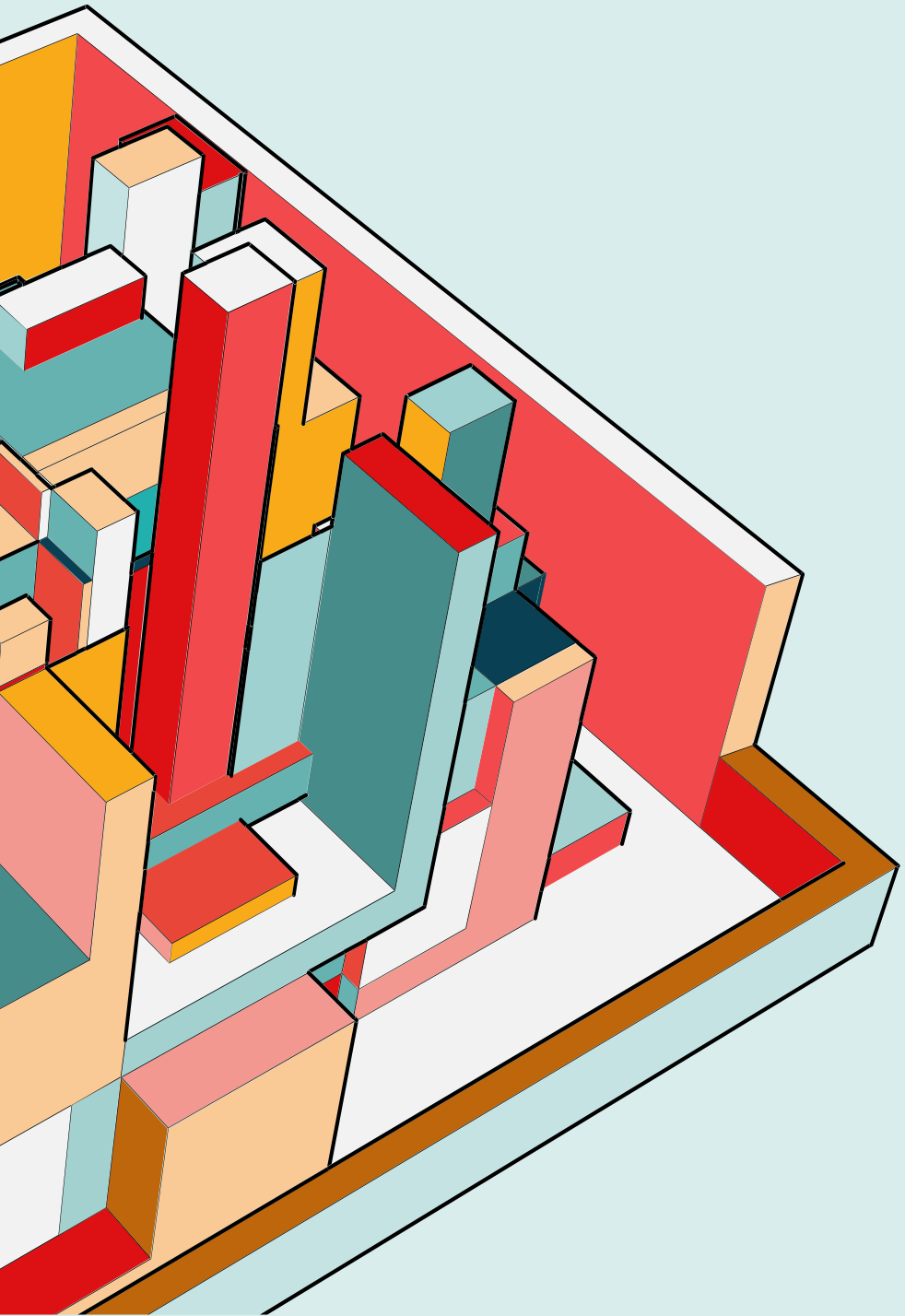
- Когда вам нужно представить простой или урезанный интерфейс к сложной подсистеме.
- Когда вы хотите разложить подсистему на отдельные слои.

ПРЕИМУЩЕСТВА И НЕДОСТАТКИ



Изолирует клиентов от компонентов сложной подсистемы.

Фасад рискует стать божественным объектом, привязанным ко всем классам программы.

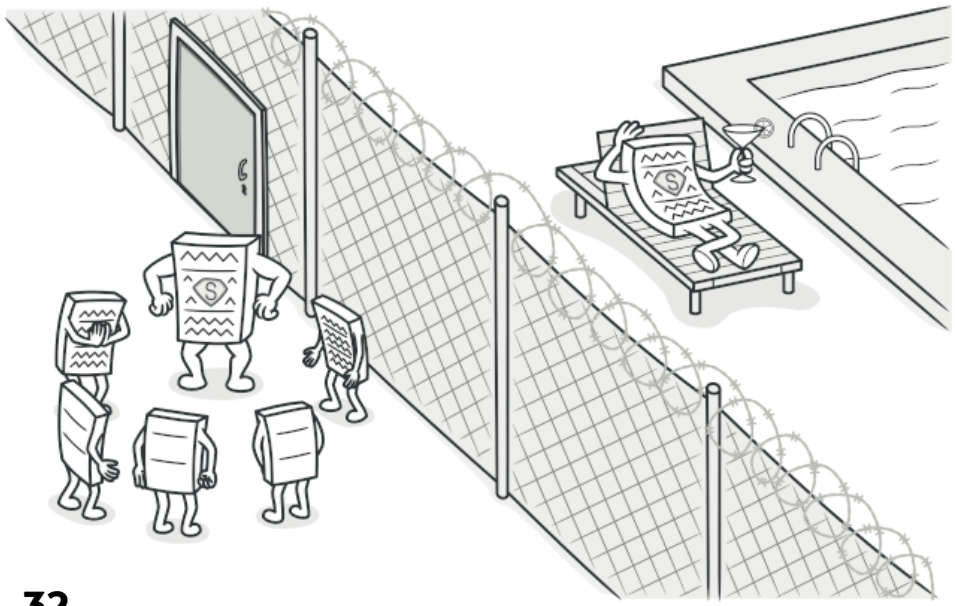


СТРУКТУРНЫЕ ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

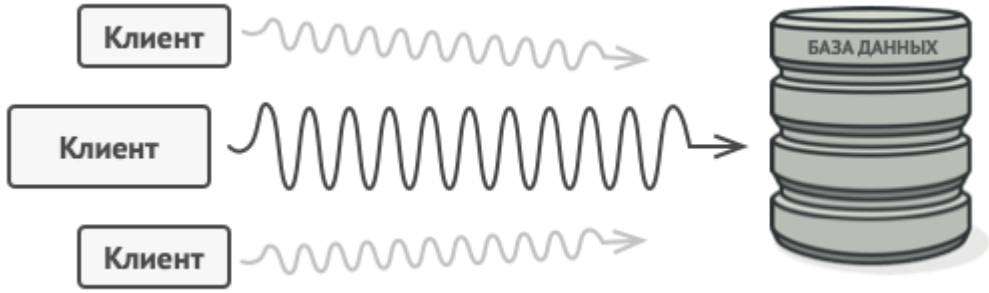
Proxy

ЗАМЕСТИТЕЛЬ

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



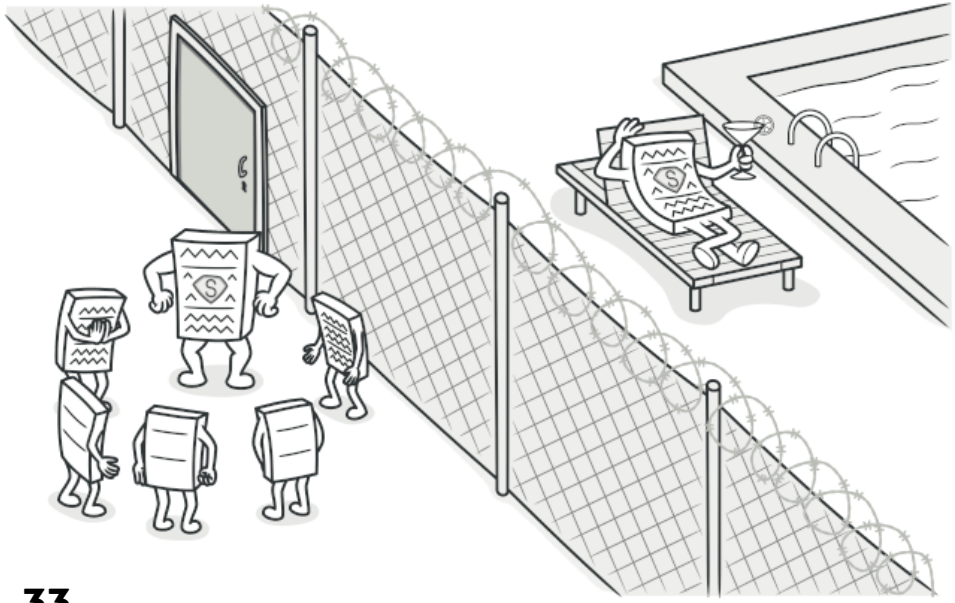
У вас есть внешний ресурсоёмкий объект, который нужен не все время, а изредка.



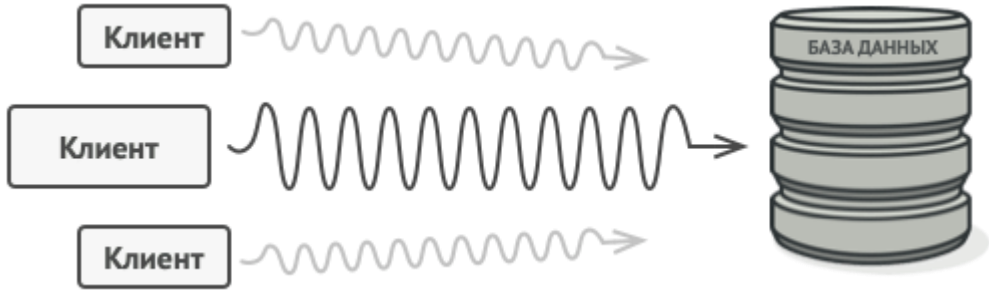
Отложенная инициализация не всегда возможна

ЗАМЕСТИТЕЛЬ

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



У вас есть внешний ресурсоёмкий объект, который нужен не все время, а изредка.



Заместитель «притворяется» базой данных, ускоряя работу за счёт ленивой инициализации и кеширования повторяющихся запросов

СТРУКТУРА

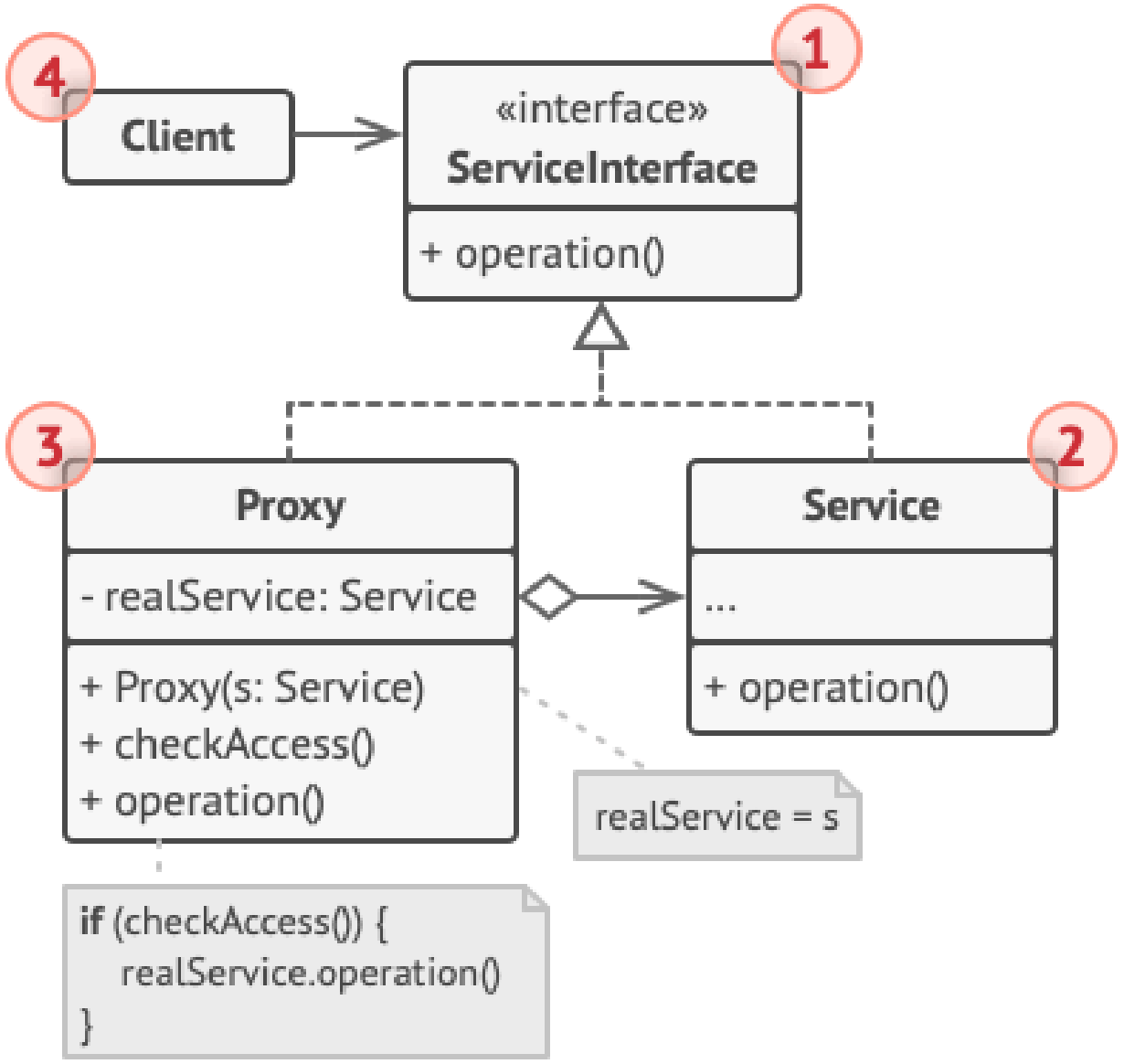
1. **Интерфейс сервиса** определяет общий интерфейс для сервиса и заместителя.

2. **Сервис** содержит полезную бизнес-логику.

3. **Заместитель** хранит ссылку на объект сервиса.

После того как заместитель заканчивает свою работу, он передаёт вызовы вложенному сервису. Заместитель может сам отвечать за создание и удаление объекта сервиса.

4. **Клиент** работает с объектами через интерфейс сервиса.



```
abstract class Subject
{
    public abstract void Request();
}

class RealSubject : Subject
{
    public override void Request()
    { }
}

class Proxy : Subject
{
    RealSubject realSubject;
    public override void Request()
    {
        if (realSubject == null)
            realSubject = new RealSubject();
        realSubject.Request();
    }
}
```

```
abstract class Subject
{
    public abstract void Request();
}

class RealSubject : Subject
{
    public override void Request()
    { }
}

class Proxy : Subject
{
    RealSubject realSubject;
    public override void Request()
    {
        if (realSubject == null)
            realSubject = new RealSubject();
        realSubject.Request();
    }
}
```

```
Subject subject = new Proxy();
subject.Request();
```

РАССМОТРИМ НА ПРИМЕРЕ

Допустим, мы взаимодействуем с базой данных через Entity Framework.

У нас есть модель и контекст данных. Класс Page представляет отдельную страницу книги, у которой есть номер и текст. Взаимодействие с базой данных может уменьшить производительность приложения.

Для оптимизации приложения мы можем использовать паттерн Прокси.

```
class Page { public int Number; public string Text; }

interface IBook
{
    Page GetPage(int number);
}

class BookStore : IBook
{
    public Page GetPage(int number)
        => new Page { Number = number, Text = $"Страница {number}" };
}

class BookStoreProxy : IBook
{
    List<Page> cache = new();
    BookStore store;
    public Page GetPage(int number)
    {
        var page = cache.FirstOrDefault(p => p.Number == number);
        if (page == null)
        {
            store ??= new BookStore();
            page = store.GetPage(number);
            cache.Add(page);
        }
        return page;
    }
}
```

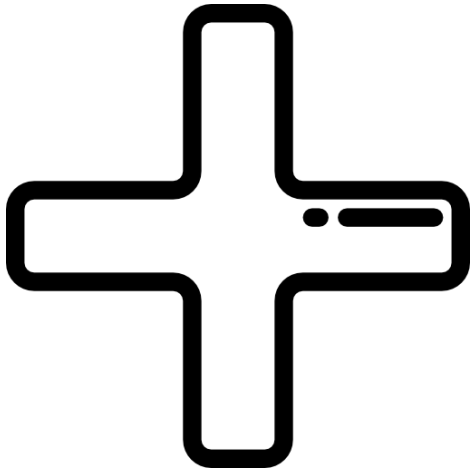
```
class Page { public int Number; public string Text; }

interface IBook
{
    Page GetPage(int number);
}

class BookStore : IBook
{
    public Page GetPage(int number)
        => new Page { Number = number, Text = $"Страница {number}" };
}

class BookStoreProxy : IBook
{
    List<Page> cache = new();
    BookStore store;
    public Page GetPage(int number)
    {
        var page = cache.FirstOrDefault(p => p.Number == number);
        if (page == null)
        {
            store ??= new BookStore();
            page = store.GetPage(number);
            cache.Add(page);
        }
        return page;
    }
}
```

```
IBook book = new BookStoreProxy();

Console.WriteLine(book.GetPage(1).Text);
Console.WriteLine(book.GetPage(2).Text);
Console.WriteLine(book.GetPage(1).Text);
```

ПРИМЕНИМОСТЬ

- Ленивая инициализация (виртуальный прокси). Когда у вас есть тяжёлый объект, грузящий данные из файловой системы или базы данных.
- Защита доступа (защищающий прокси). Когда в программе есть разные типы пользователей, и вам хочется защищать объект от неавторизованного доступа. Например, если ваши объекты — это важная часть операционной системы, а пользователи — сторонние программы (хорошие или вредоносные).
- Локальный запуск сервиса (удалённый прокси). Когда настоящий сервисный объект находится на удалённом сервере.
- Логирование запросов (логирующий прокси). Когда требуется хранить историю обращений к сервисному объекту.
- Кеширование объектов («умная» ссылка). Когда нужно кешировать результаты запросов клиентов и управлять их жизненным циклом.

ПРЕИМУЩЕСТВА И НЕДОСТАТКИ



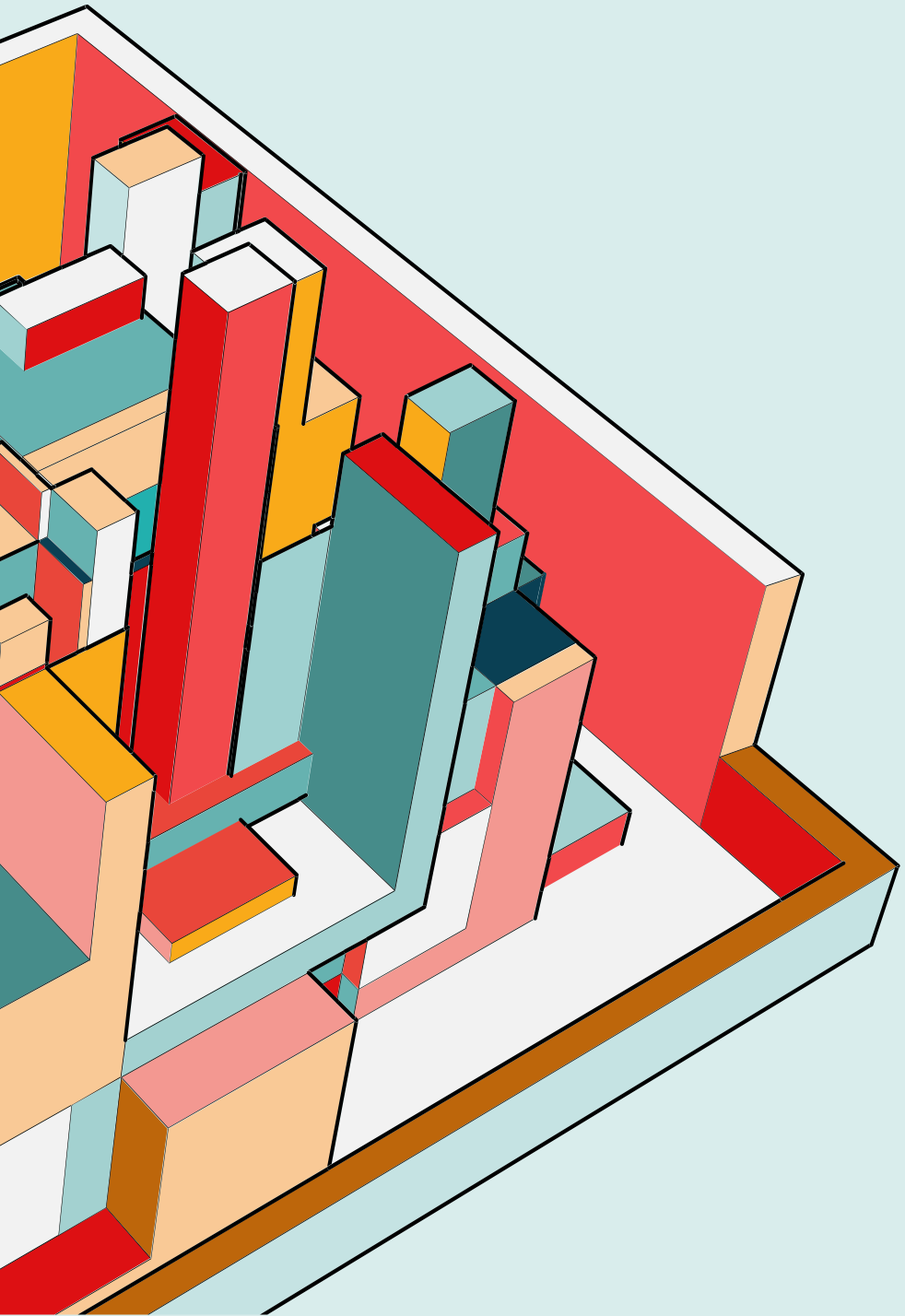
Позволяет контролировать сервисный объект незаметно для клиента.

Может работать, даже если сервисный объект ещё не создан.

Может контролировать жизненный цикл служебного объекта.

Усложняет код программы из-за введения дополнительных классов.

Увеличивает время отклика от сервиса.

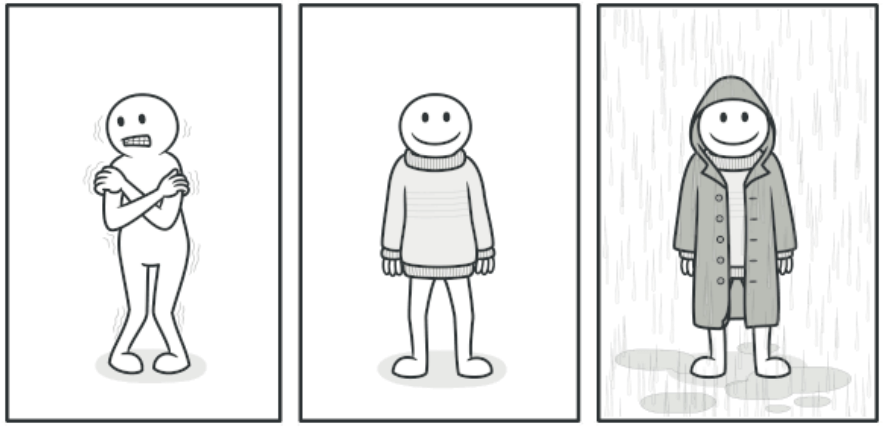


СТРУКТУРНЫЕ ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

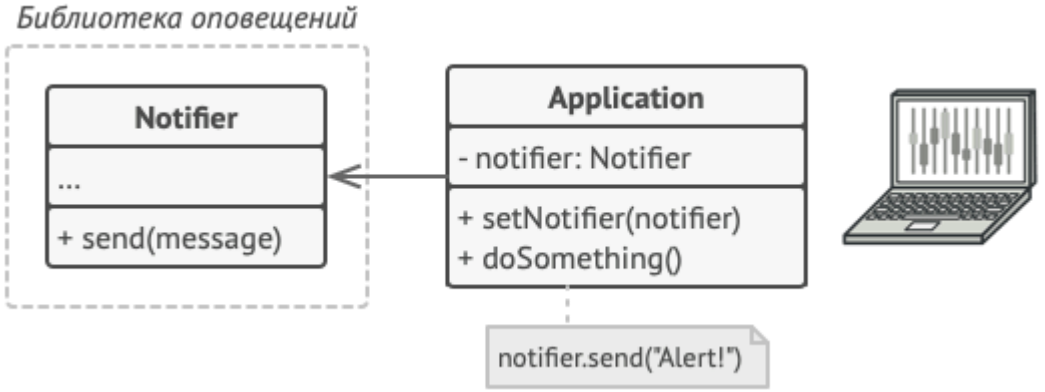
Decorator

ДЕКОРАТОР

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



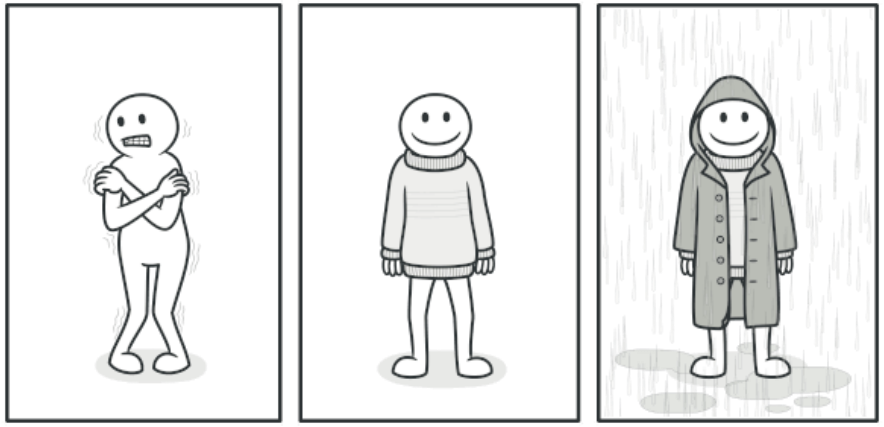
Вы работаете над библиотекой оповещений, которую можно подключать к разнообразным программам, чтобы получать уведомления о важных событиях.



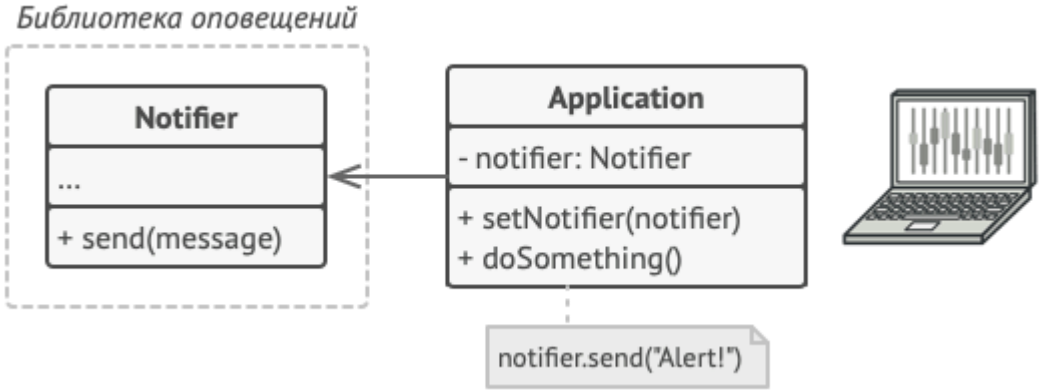
Но одних email-оповещений пользователям мало!
А может наследование?

ДЕКОРАТОР

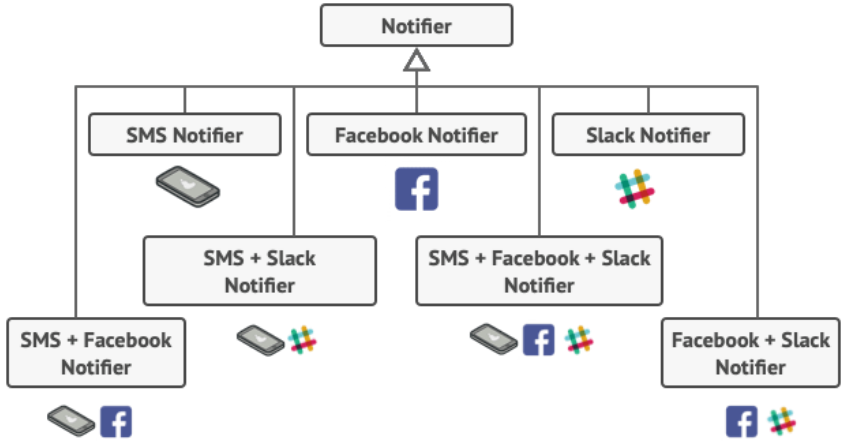
Позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».



Вы работаете над библиотекой оповещений, которую можно подключать к разнообразным программам, чтобы получать уведомления о важных событиях.



Тогда ждите комбинаторный взрыв подклассов



РЕШЕНИЕ

Наследование — первое, что приходит в голову.

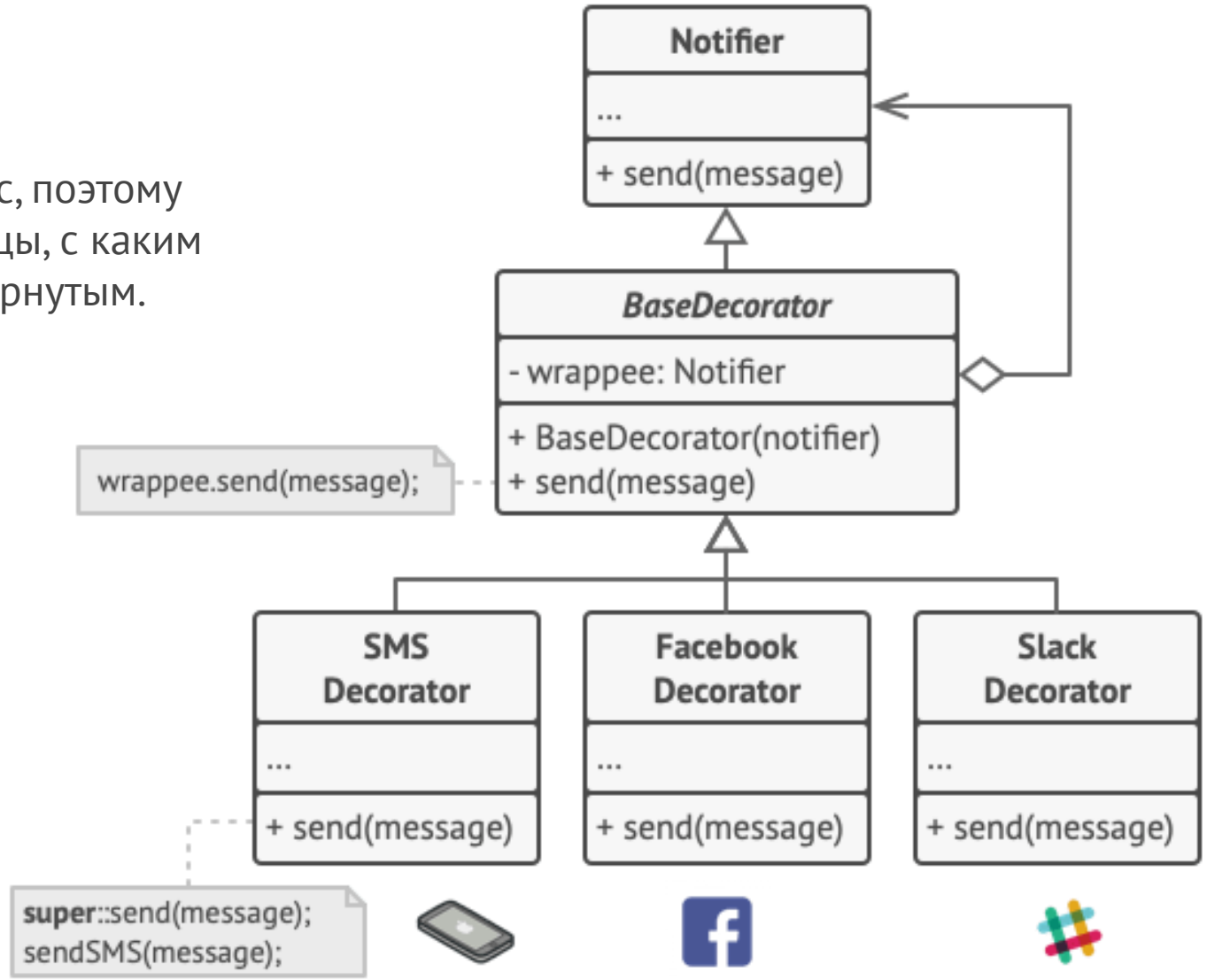
Механизм наследования:

- Статичен. Вы не можете изменить поведение существующего объекта.
- Не разрешает наследовать поведение нескольких классов одновременно.

Обойти эти проблемы можно *агрегацией* либо *композицией*

РЕШЕНИЕ

Оба объекта имеют общий интерфейс, поэтому для пользователя нет никакой разницы, с каким объектом работать — чистым или обёрнутым.



СТРУКТУРА

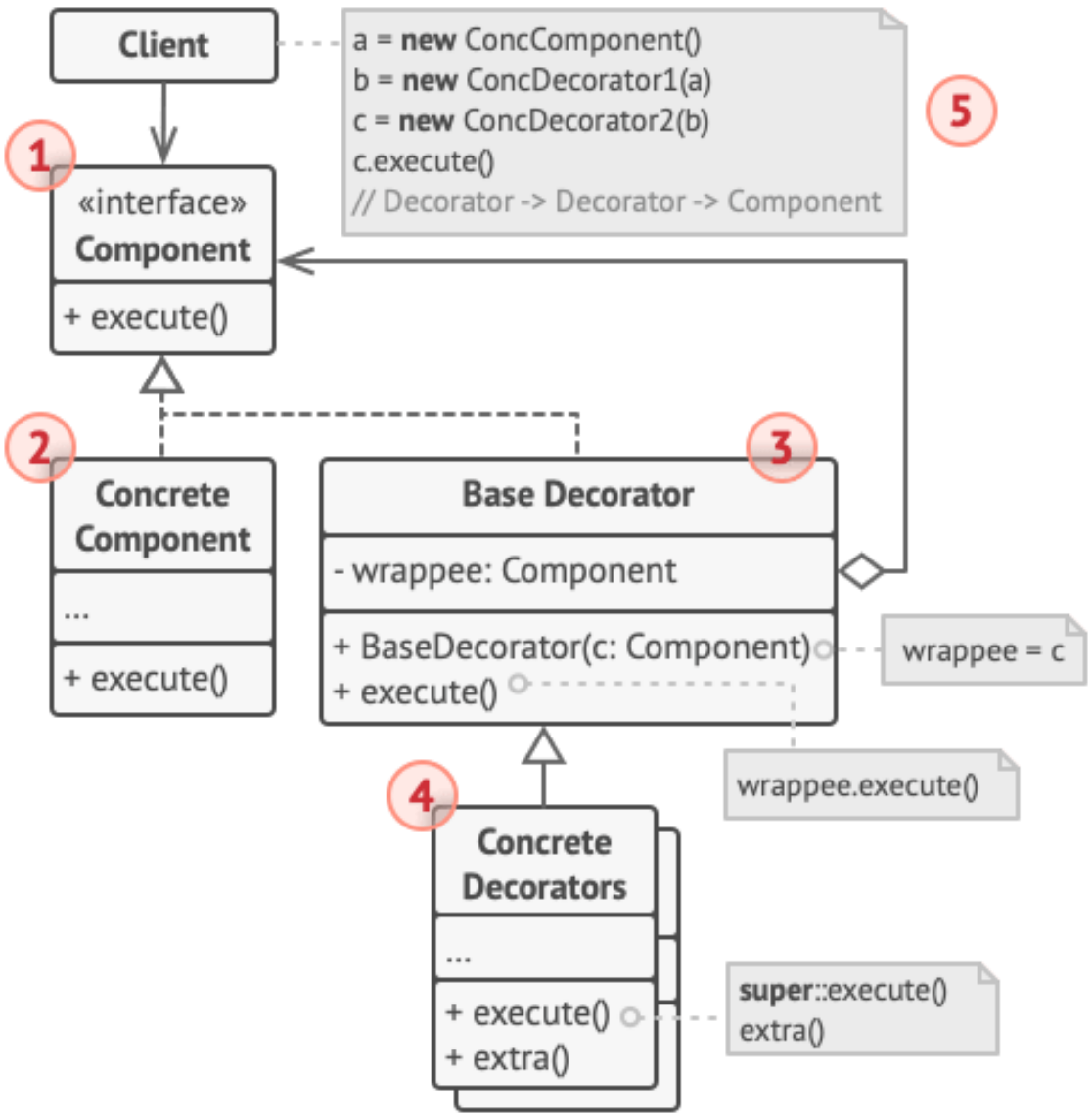
1. **Компонент** задаёт общий интерфейс обёрток и оборачиваемых объектов.

2. **Конкретный компонент** определяет класс оборачиваемых объектов. Он содержит какое-то базовое поведение, которое потом изменяют декораторы.

3. **Базовый декоратор** хранит ссылку на вложенный объект-компонент..

4. **Конкретные декораторы** — это различные вариации декораторов, которые содержат добавочное поведение. Оно выполняется до или после вызова аналогичного поведения обёрнутого объекта.

5. **Клиент** может оборачивать простые компоненты и декораторы в другие декораторы, работая со всеми объектами через общий интерфейс компонентов.



```
abstract class Component
{
    public abstract void Operation();
}

class ConcreteComponent : Component
{
    public override void Operation() { }
}

abstract class Decorator : Component
{
    protected Component Component;
    protected Decorator(Component component) => Component = component;
    public override void Operation() => Component?.Operation();
}

class ConcreteDecoratorA : Decorator
{
    public ConcreteDecoratorA(Component component) : base(component) { }
    public override void Operation() => base.Operation();
}

class ConcreteDecoratorB : Decorator
{
    public ConcreteDecoratorB(Component component) : base(component) { }
    public override void Operation() => base.Operation();
}
```

РАССМОТРИМ НА ПРИМЕРЕ

Допустим, у нас есть пиццерия, которая готовит различные типы пицц с различными добавками. Есть итальянская, болгарская пиццы. К ним могут добавляться помидоры, сыр и т.д.

И в зависимости от типа пицц и комбинаций добавок пицца может иметь разную стоимость.

```
abstract class Pizza
{
    public string Name { get; protected set; }
    public Pizza(string name) => Name = name;
    public abstract int GetCost();
}

class ItalianPizza : Pizza
{
    public ItalianPizza() : base("Итальянская пицца") { }
    public override int GetCost() => 10;
}
class BulgerianPizza : Pizza
{
    public BulgerianPizza() : base("Болгарская пицца") { }
    public override int GetCost() => 8;
}

abstract class PizzaDecorator : Pizza
{
    protected Pizza pizza;
    public PizzaDecorator(Pizza pizza, string addon)
        : base($"{pizza.Name}, {addon}")
    {
        this.pizza = pizza;
    }
}
```

```
abstract class Pizza
{
    public string Name { get; protected set; }
    public Pizza(string name) => Name = name;
    public abstract int GetCost();
}

class ItalianPizza : Pizza
{
    public ItalianPizza() : base("Итальянская пицца") { }
    public override int GetCost() => 10;
}
class BulgerianPizza : Pizza
{
    public BulgerianPizza() : base("Болгарская пицца") { }
    public override int GetCost() => 8;
}

abstract class PizzaDecorator : Pizza
{
    protected Pizza pizza;
    public PizzaDecorator(Pizza pizza, string addon)
        : base($"{pizza.Name}, {addon}")
    {
        this.pizza = pizza;
    }
}
```

```
class TomatoPizza : PizzaDecorator
{
    public TomatoPizza(Pizza pizza) : base(pizza, "с томатами") { }
    public override int GetCost() => pizza.GetCost() + 3;
}

class CheesePizza : PizzaDecorator
{
    public CheesePizza(Pizza pizza) : base(pizza, "с сыром") { }
    public override int GetCost() => pizza.GetCost() + 5;
}
```

```
abstract class Pizza
{
    public string Name { get; protected set; }
    public Pizza(string name) => Name = name;
    public abstract int GetCost();
}

class ItalianPizza : Pizza
{
    public ItalianPizza() : base("Итальянская пицца") { }
    public override int GetCost() => 10;
}
class BulgerianPizza : Pizza
{
    public BulgerianPizza() : base("Болгарская пицца") { }
    public override int GetCost() => 8;
}

abstract class PizzaDecorator : Pizza
{
    protected Pizza pizza;
    public PizzaDecorator(Pizza pizza, string addon)
        : base($"{pizza.Name}, {addon}")
    {
        this.pizza = pizza;
    }
}
```

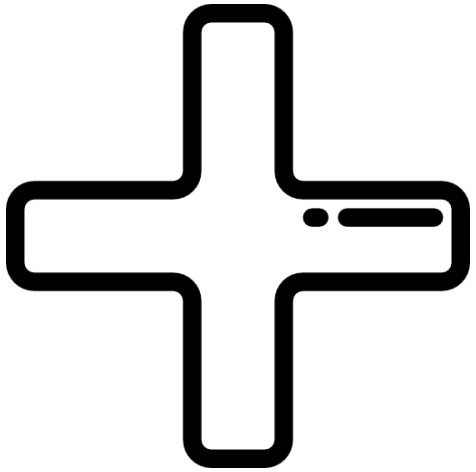
```
class TomatoPizza : PizzaDecorator
{
    public TomatoPizza(Pizza pizza) : base(pizza, "с томатами") { }
    public override int GetCost() => pizza.GetCost() + 3;
}

class CheesePizza : PizzaDecorator
{
    public CheesePizza(Pizza pizza) : base(pizza, "с сыром") { }
    public override int GetCost() => pizza.GetCost() + 5;
}
```

```
Pizza pizza1 = new TomatoPizza(new ItalianPizza());
Console.WriteLine($"Название: {pizza1.Name}, Цена: {pizza1.GetCost()}");

Pizza pizza2 = new CheesePizza(new ItalianPizza());
Console.WriteLine($"Название: {pizza2.Name}, Цена: {pizza2.GetCost()}");

Pizza pizza3 = new CheesePizza(new TomatoPizza(new BulgerianPizza()));
Console.WriteLine($"Название: {pizza3.Name}, Цена: {pizza3.GetCost()}");
```

ПРИМЕНИМОСТЬ

- Когда вам нужно добавлять обязанности объектам на лету, незаметно для кода, который их использует.
- Когда нельзя расширить обязанности объекта с помощью наследования.

ПРЕИМУЩЕСТВА И НЕДОСТАТКИ



Большая гибкость, чем у наследования.

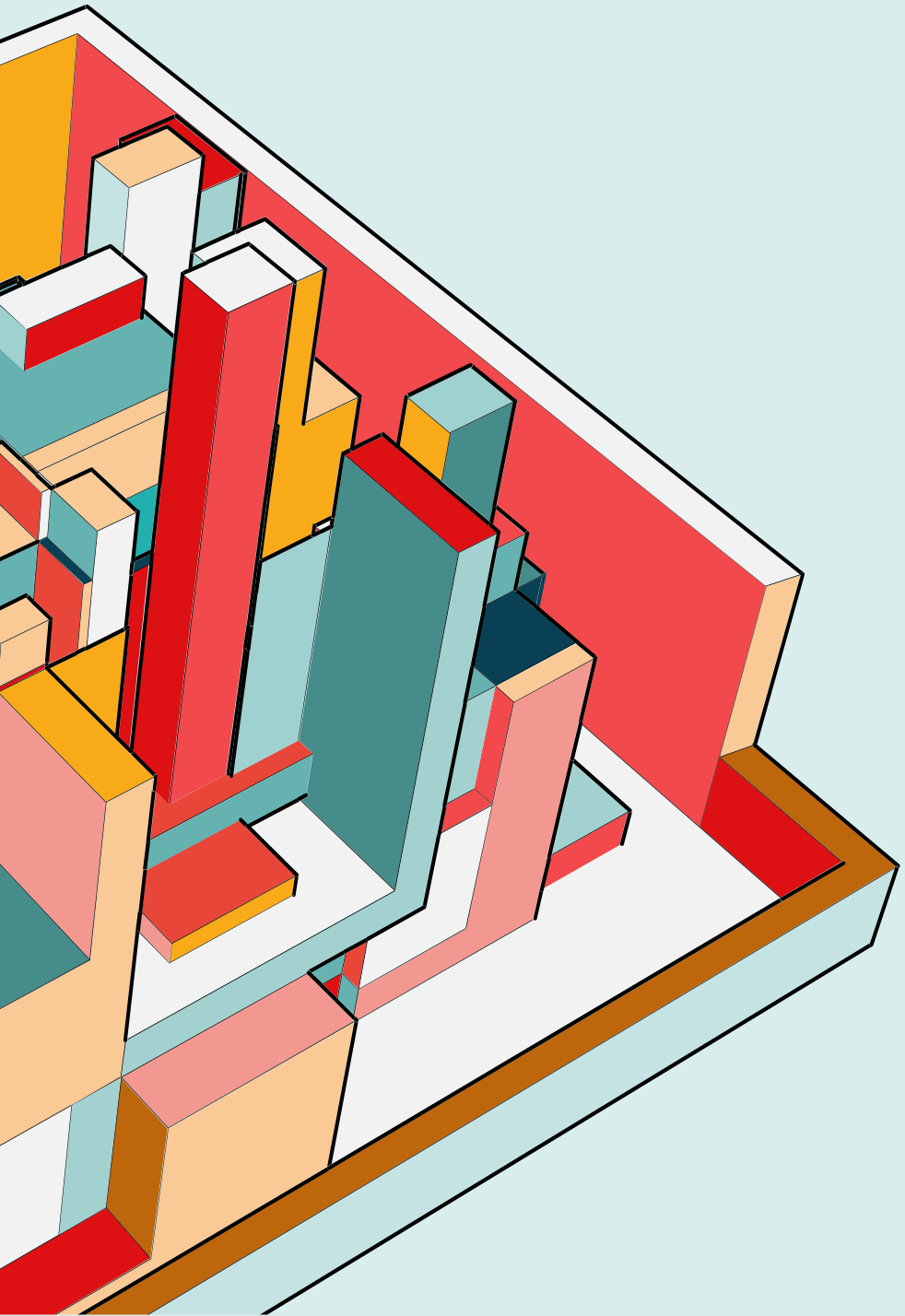
Позволяет добавлять обязанности на лету.

Можно добавлять несколько новых обязанностей сразу.

Позволяет иметь несколько мелких объектов вместо одного объекта на все случаи жизни.

Трудно конфигурировать многократно обёрнутые объекты.

Обилие крошечных классов

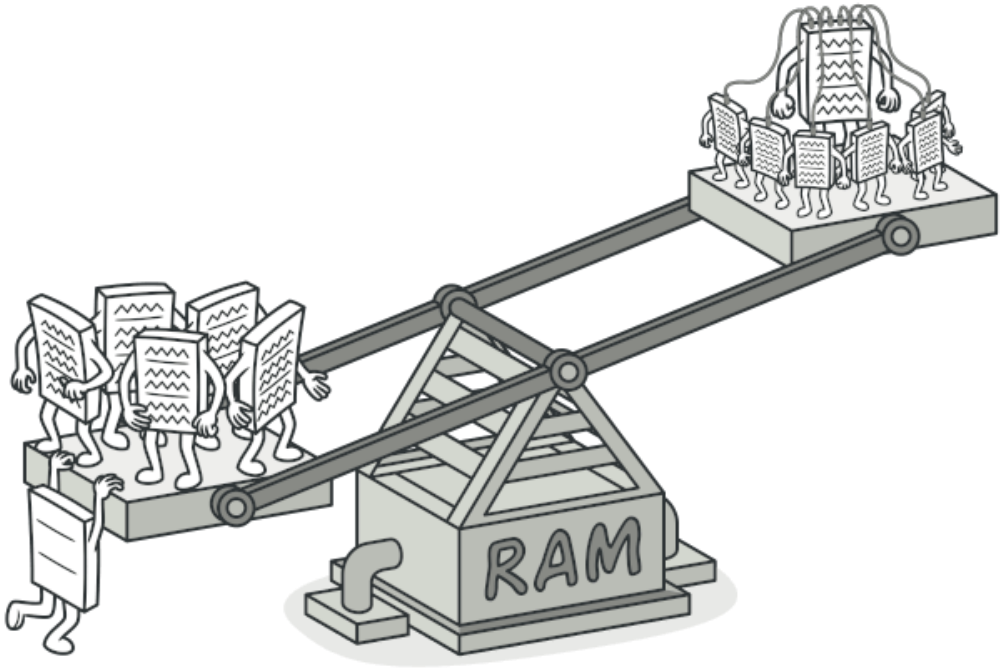


СТРУКТУРНЫЕ ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

Flyweight

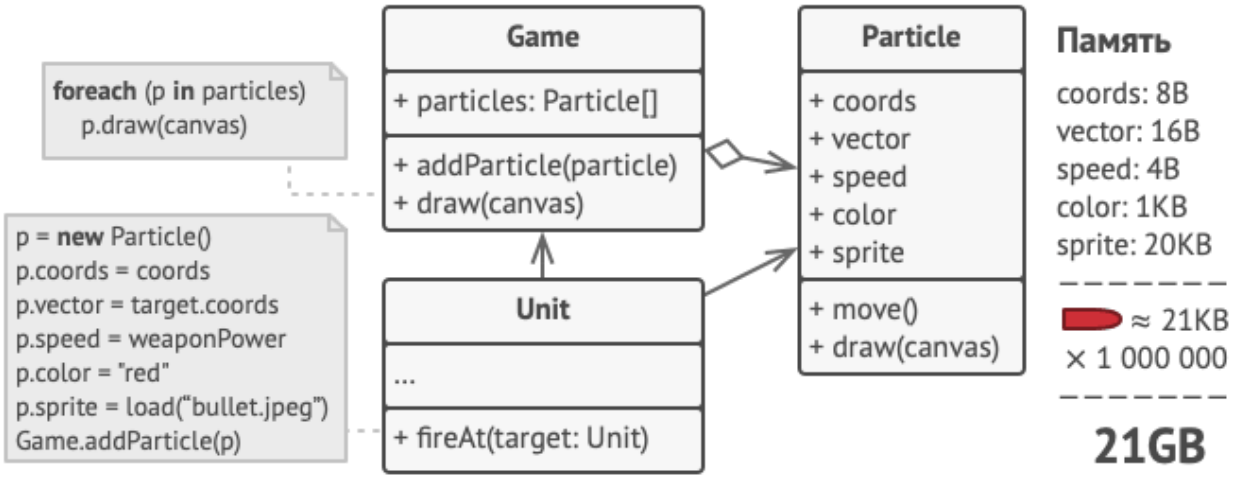
ЛЕГКОВЕС

Позволяет вместить большее количество объектов в отведённую оперативную память



На досуге вы решили написать небольшую игру…

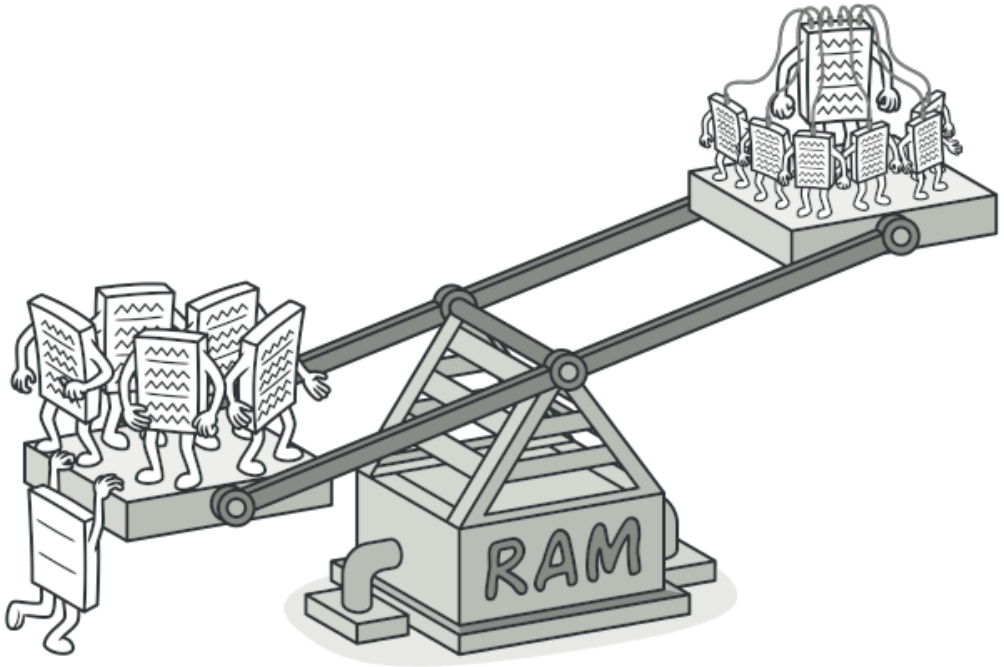
Пули, снаряды, осколки от взрывов — красота!

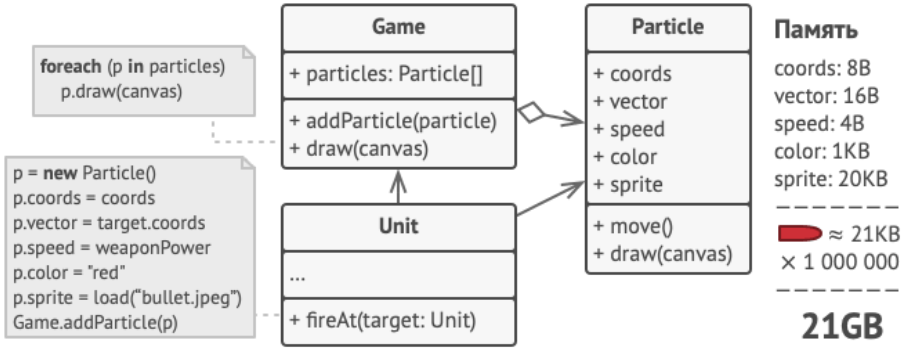


Цвет и спрайт занимают больше всего памяти

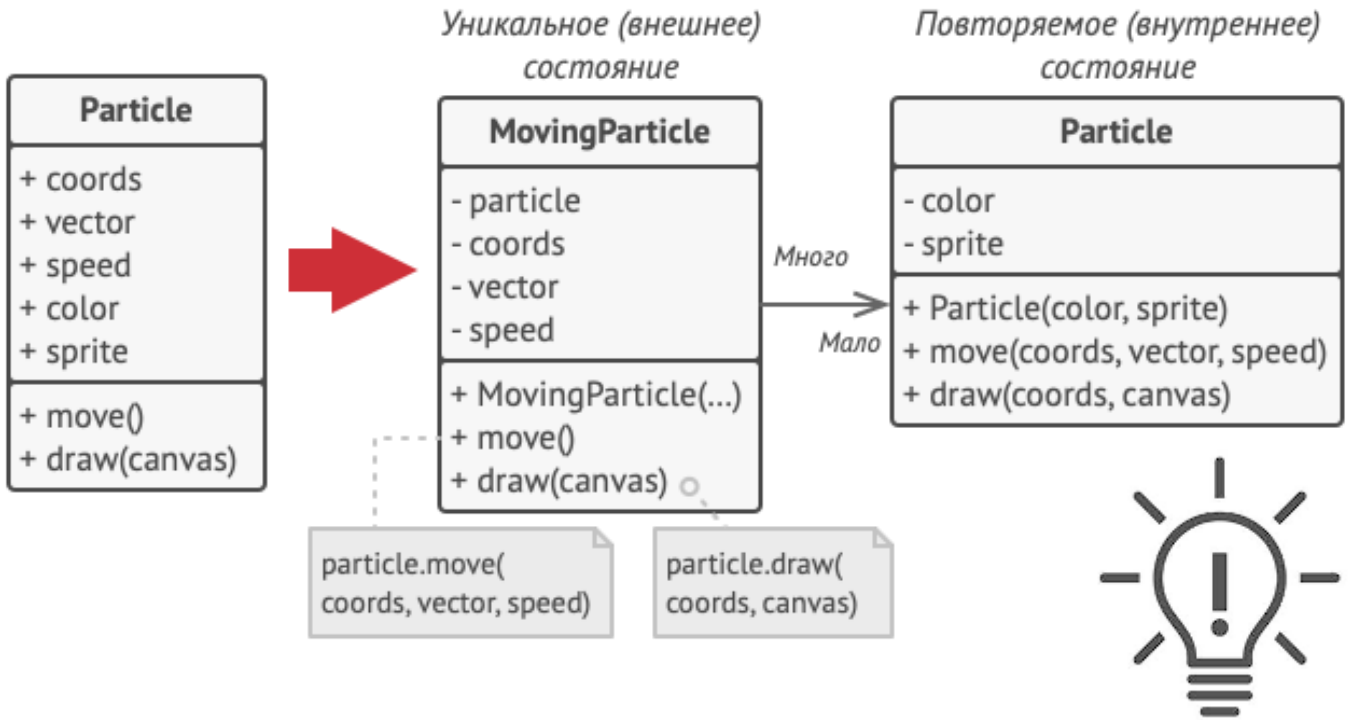
ЛЕГКОВЕС

Позволяет вместить большее количество объектов в отведённую оперативную память





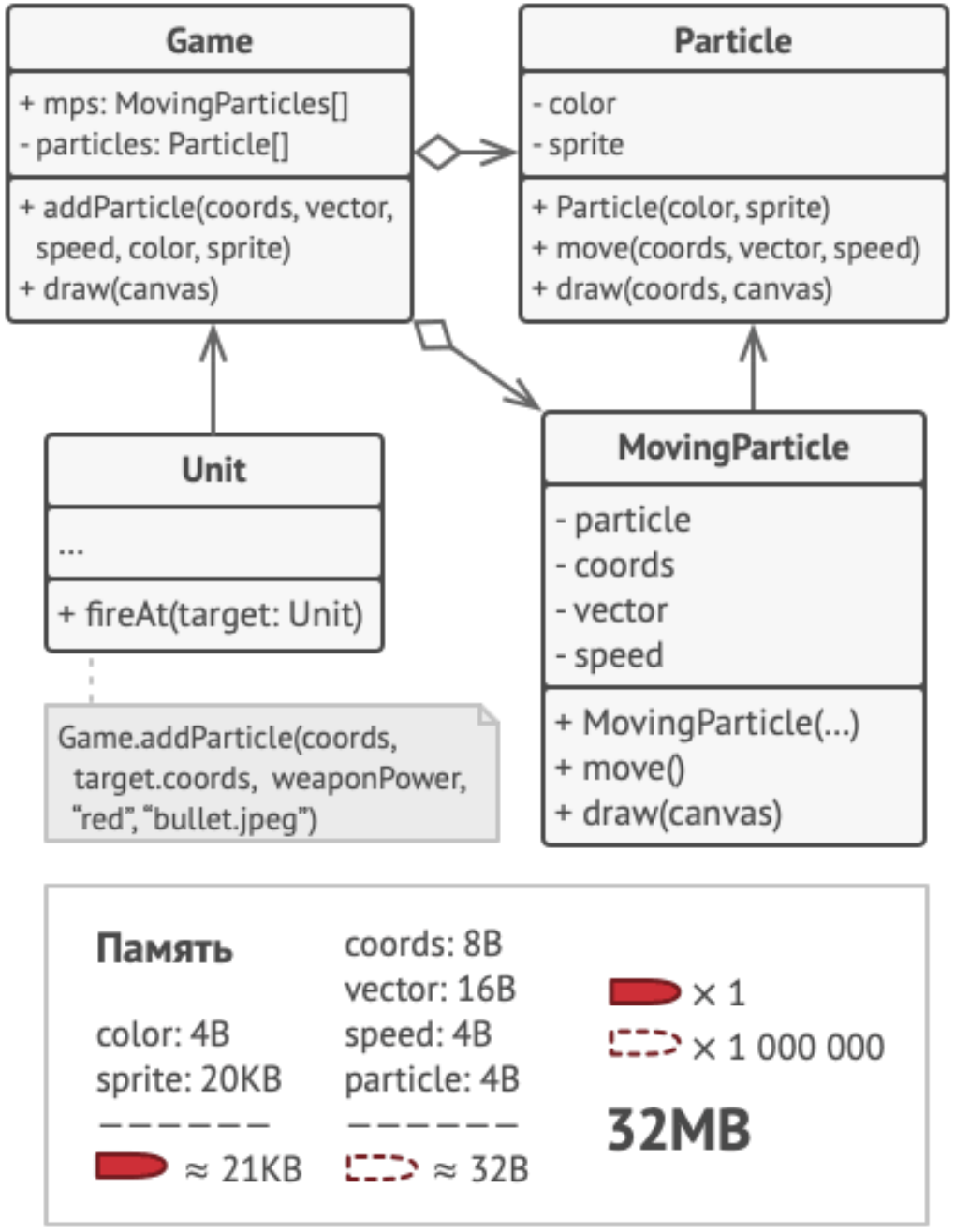
Цвет и спрайт занимают больше всего памяти



РЕШЕНИЕ

Паттерн Легковес предлагает не хранить в классе внешнее состояние, а передавать его в те или иные методы через параметры.

Таким образом, одни и те же объекты можно будет повторно использовать в различных контекстах.



В нашем примере с частицами достаточно будет оставить всего три объекта с отличающимися спрайтами и цветом — для пуль, снарядов и осколков

СТРУКТУРА

1. Вы всегда должны помнить о том, что **Легковес применяется** в программе, имеющей **громадное количество одинаковых объектов**

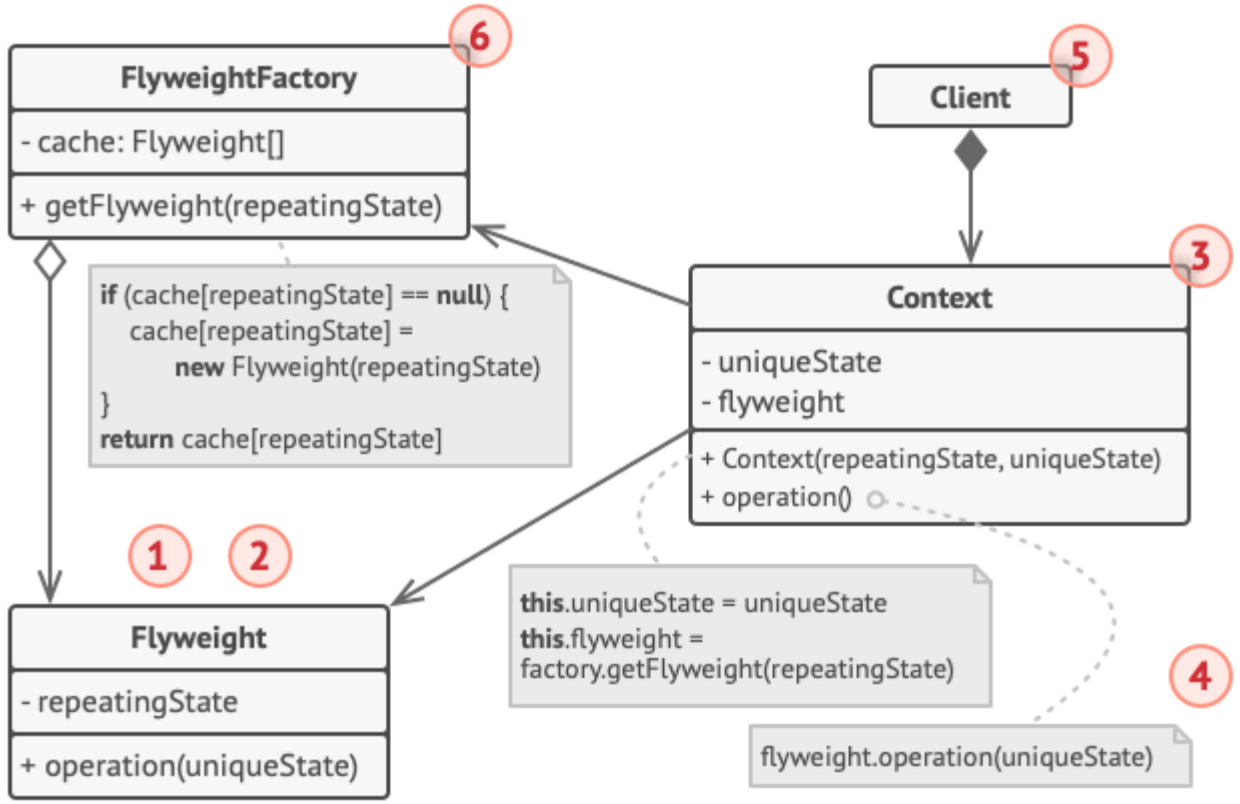
2. **Легковес** содержит состояние, которое повторялось во множестве первоначальных объектов.

3. **Контекст** содержит «внешнюю» часть состояния, уникальную для каждого объекта.

4. Поведение оригинального объекта чаще всего оставляют в Легковесе, передавая значения контекста через параметры методов.

5. **Клиент** вычисляет или хранит контекст, то есть внешнее состояние легковесов.

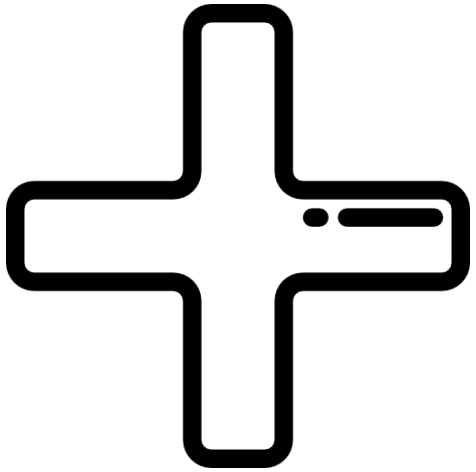
6. **Фабрика легковесов** управляет созданием и повторным использованием легковесов.



ПРИМЕНИМОСТЬ

- **Когда не хватает оперативной памяти для поддержки всех нужных объектов.**
- Эффективность паттерна **Легковес** во многом зависит от того, как и где он используется. Применяйте этот паттерн, когда выполнены все перечисленные условия:
  - в приложении используется большое число объектов;
  - из-за этого высоки расходы оперативной памяти;
  - большую часть состояния объектов можно вынести за пределы их классов;
  - большие группы объектов можно заменить относительно небольшим количеством разделяемых объектов, поскольку внешнее состояние вынесено.

ПРЕИМУЩЕСТВА И НЕДОСТАТКИ



Экономит оперативную память

Расходует процессорное время на поиск/вычисление контекста.

Усложняет код программы из-за введения множества дополнительных классов.

СПАСИБО!

Виденин Сергей

@videninserg