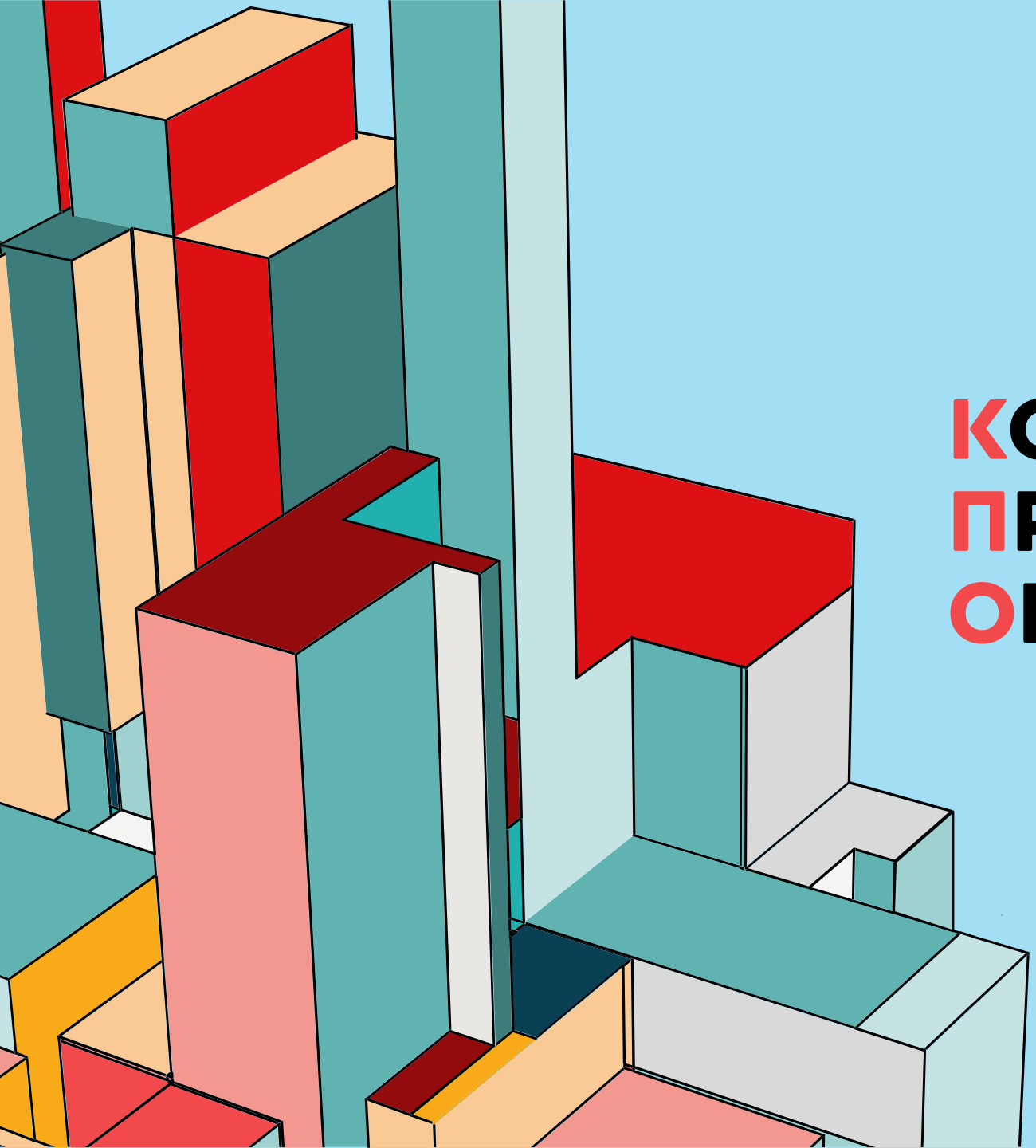
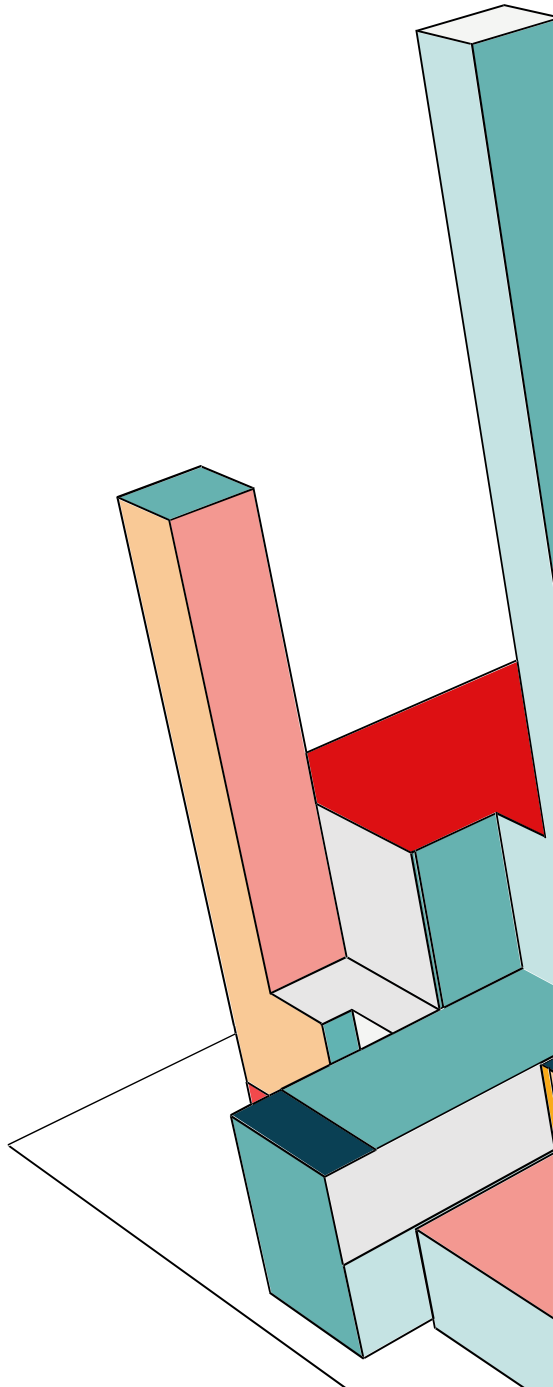


КОНСТРУИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

ПЛАН ЛЕКЦИИ № 6

Поведенческие паттерны проектирования

1. Команда (Command)
2. Посредник (Mediator)
3. Состояние (State)
4. Стратегия (Strategy)
5. Шаблонный метод (Template method)
6. Посетитель (Visitor)



ВИДЫ ПАТТЕРНОВ

- **Поведенческие** паттерны заботятся об эффективной коммуникации между объектами.

- **Порождающие** паттерны беспокоятся о гибком создании объектов без внесения в программу лишних зависимостей.

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

СПИСОК ШАБЛОНОВ

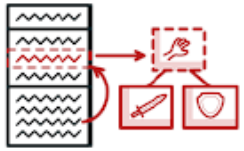
- C Абстрактная фабрика
- S Адаптер
- S Мост
- C Строитель
- B Цепочка обязанностей
- B Команда
- S Компоновщик
- S Декоратор
- S Фасад
- C Фабричный метод
- S Приспособленец
- B Интерпретатор
- B Итератор
- B Посредник
- B Хранитель
- C Прототип
- S Прокси
- B Наблюдатель
- C Одиночка
- B Состояние
- B Стратегия
- B Шаблонный метод
- B Посетитель

- B — поведенческие (behavioral);
- C — порождающие (creational);
- S — структурные (structural).

ПОВЕДЕНЧЕСКИЕ ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

Команда
Command

Превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций.

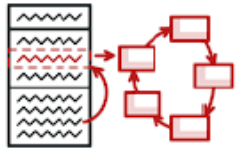


Стратегия
Strategy

Определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы.

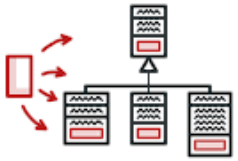
Шаблонный метод
Template Method

Определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Паттерн позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.



Состояние
State

Позволяет объектам менять поведение в зависимости от своего состояния. Извне создаётся впечатление, что изменился класс объекта.



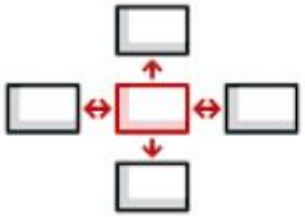
Посетитель
Visitor

Позволяет добавлять в программу новые операции, не изменяя классы объектов, над которыми эти операции могут выполняться.

Наблюдатель
Observer

Создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.

Решают задачи эффективного и безопасного взаимодействия между объектами программы



Посредник

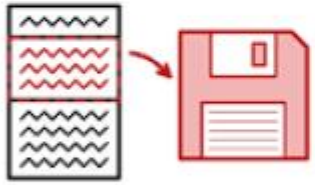
Mediator

Позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник.

Итератор

Iterator

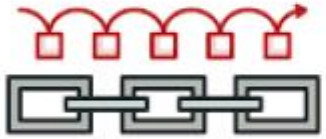
Даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.



Снимок

Memento

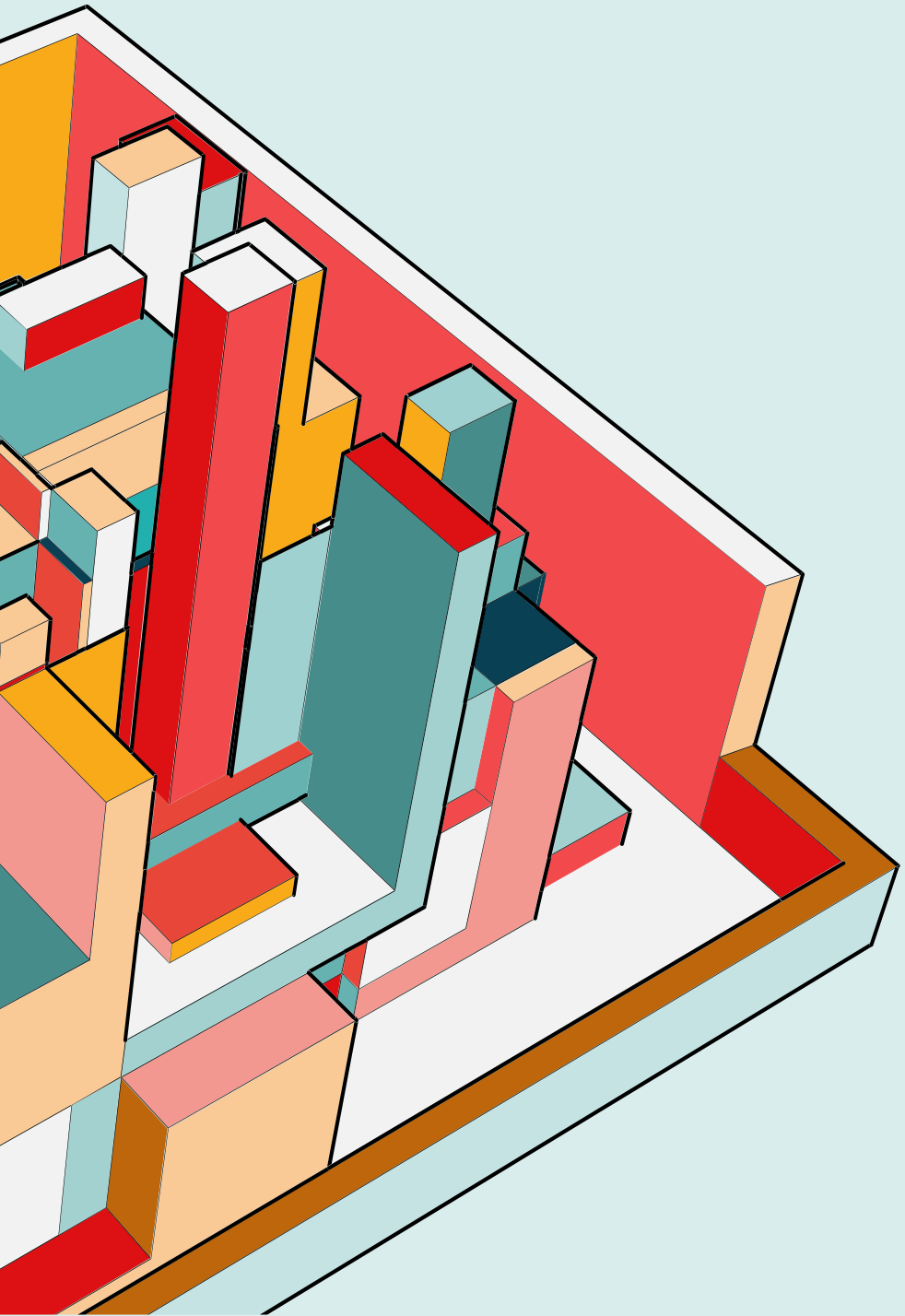
Позволяет сохранять и восстанавливать прошлые состояния объектов, не раскрывая подробностей их реализации.



Цепочка обязанностей

Chain of Responsibility

Позволяет передавать запросы последовательно по цепочке обработчиков. Каждый последующий обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи.

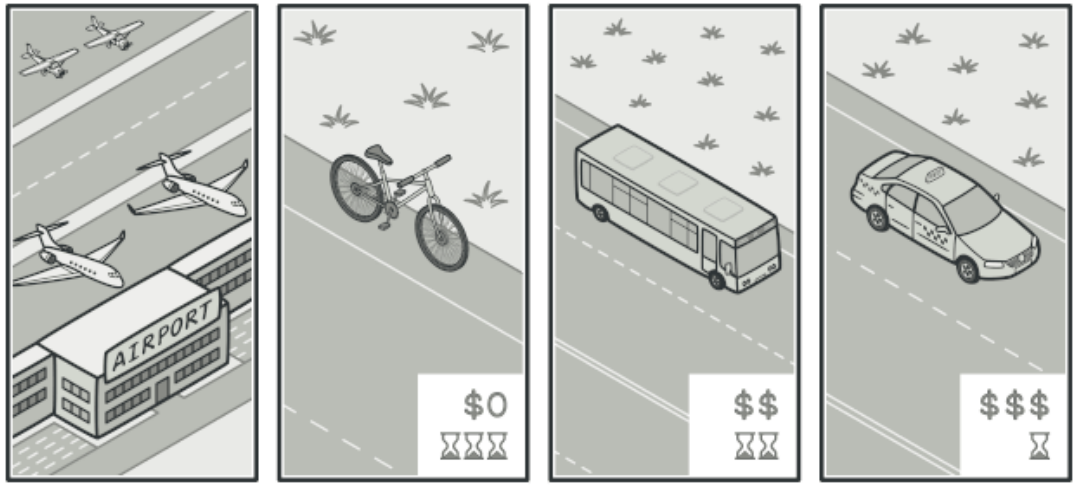


ПОВЕДЕНЧЕСКИЕ ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

Strategy

СТРАТЕГИЯ

Определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы

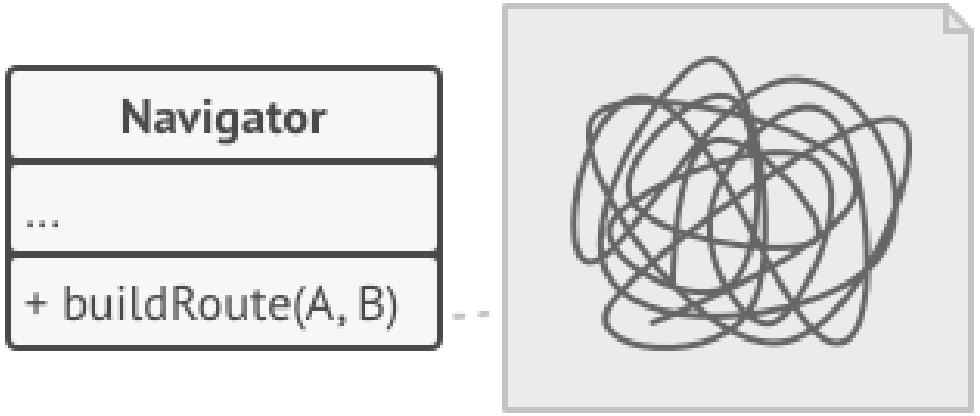


До аэропорта можно доехать на автобусе, такси или велосипеде. Здесь вид транспорта является стратегией.

Вы решили написать приложение-навигатор для путешественников.

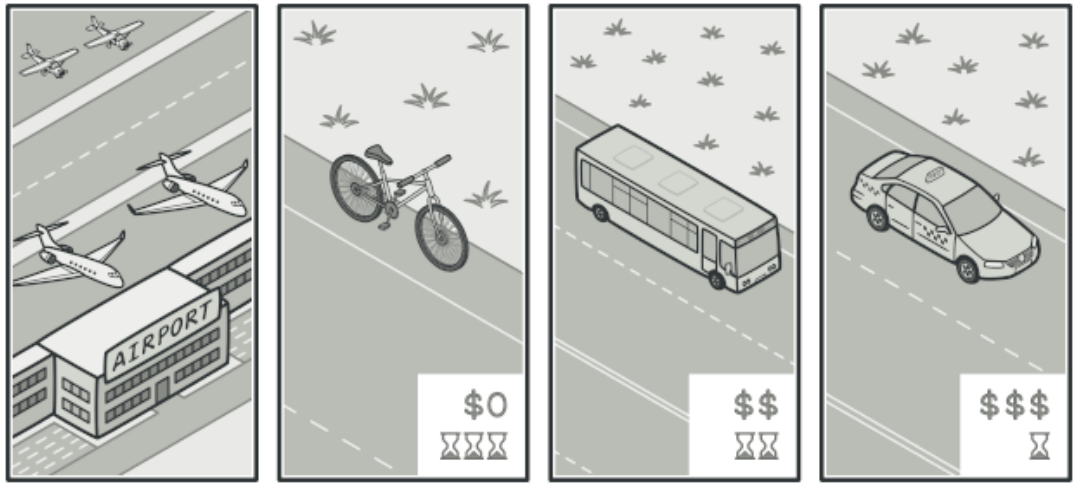
Первая версия вашего навигатора могла прокладывать маршрут лишь по дорогам

В ближайшей перспективе вы хотели бы добавить прокладывание маршрутов по велодорожкам, по городу на общественном транспорте



СТРАТЕГИЯ

Определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы

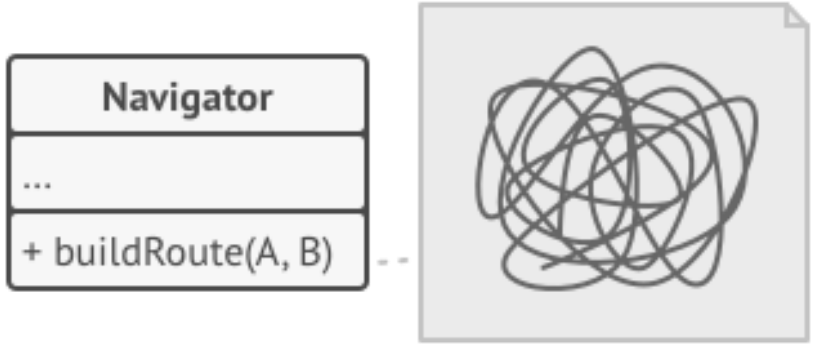


До аэропорта можно доехать на автобусе, такси или велосипеде. Здесь вид транспорта является стратегией.

Вы решили написать приложение-навигатор для путешественников.

Первая версия вашего навигатора могла прокладывать маршрут лишь по дорогам

В ближайшей перспективе вы хотели бы добавить прокладывание маршрутов по велодорожкам, по городу на общественном транспорте



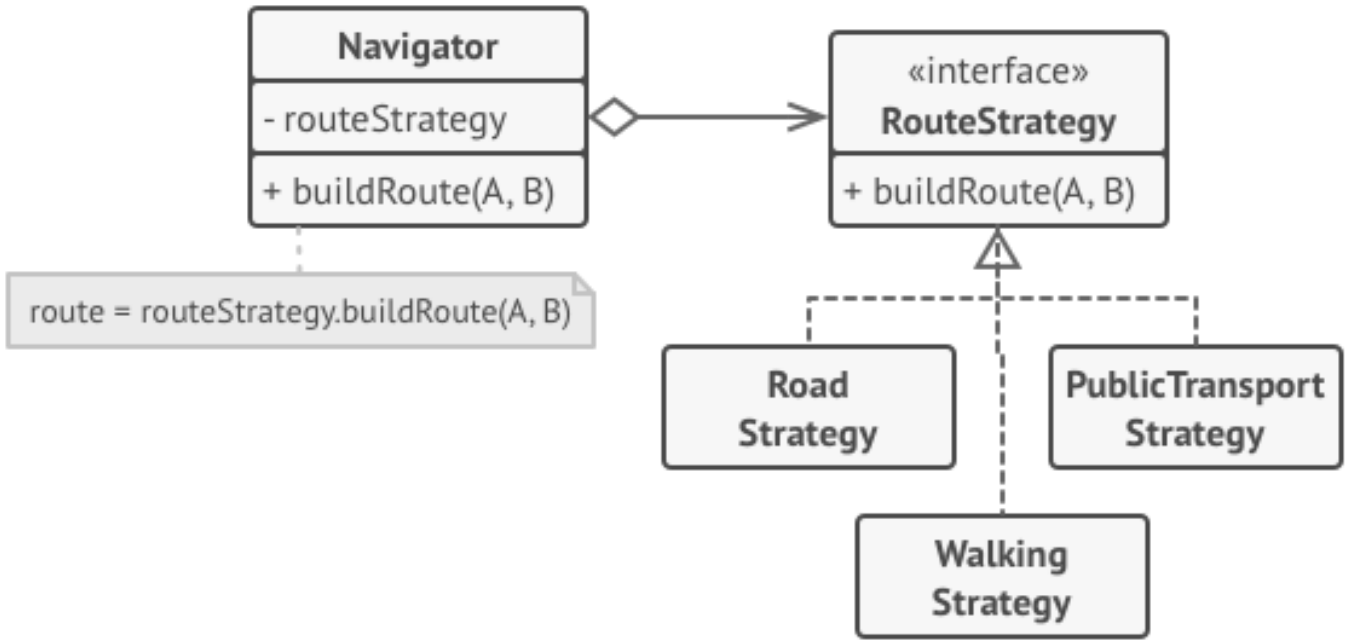
Паттерн Стратегия предлагает определить семейство схожих алгоритмов, которые часто изменяются или расширяются, и вынести их в собственные классы, называемые стратегиями.

РЕШЕНИЕ

Вместо того, чтобы изначальный класс сам выполнял тот или иной алгоритм, он будет играть роль контекста, ссылаясь на одну из стратегий и делегируя ей выполнение работы.

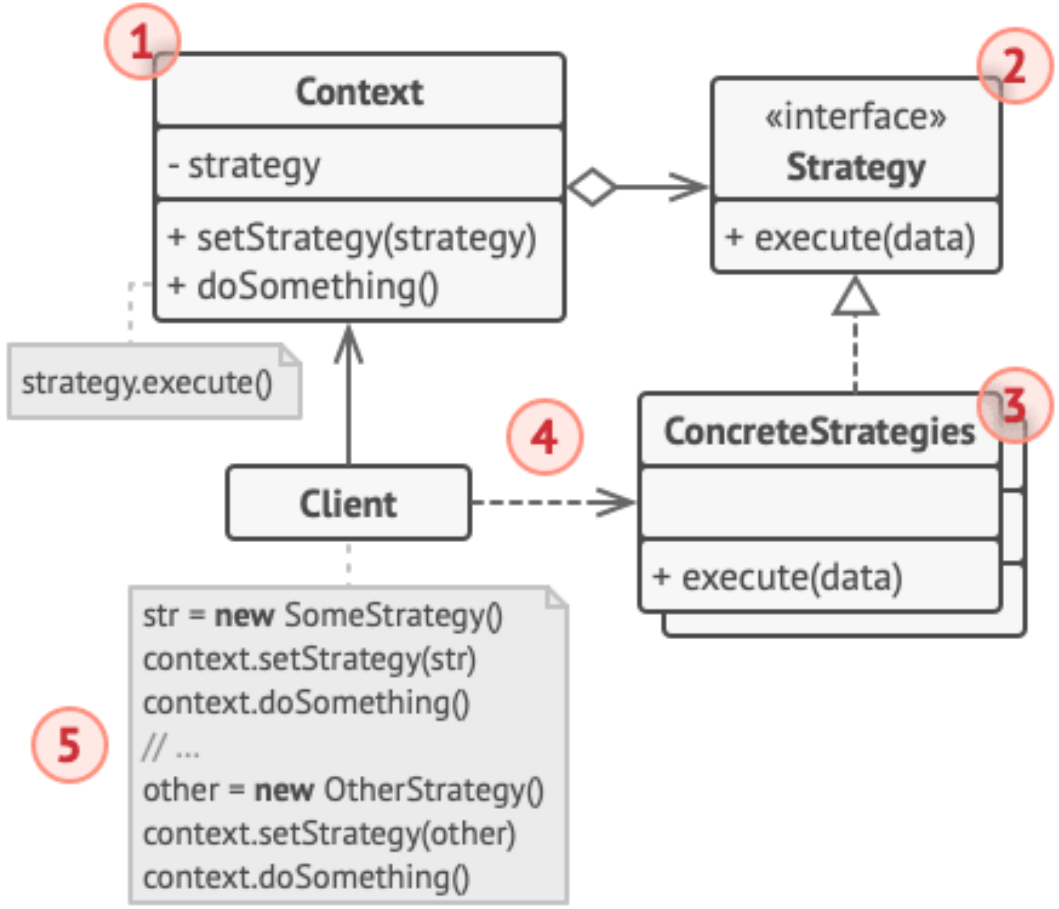
Чтобы сменить алгоритм, вам будет достаточно подставить в контекст другой объект-стратегию.

Стратегии построения пути.



СТРУКТУРА

1. **Контекст** хранит ссылку на объект конкретной стратегии.
2. **Стратегия** определяет интерфейс, общий для всех вариаций алгоритма.
3. **Конкретные стратегии** реализуют различные вариации алгоритма.
4. Во время выполнения программы контекст получает вызовы от клиента и делегирует их объекту конкретной стратегии.
5. Клиент должен создать объект конкретной стратегии и передать его в конструктор контекста. Кроме этого, клиент должен иметь возможность заменить стратегию на лету, используя сеттер.



```
public interface IStrategy
{
    void Algorithm();
}

public class ConcreteStrategy1 : IStrategy
{
    public void Algorithm()
    {}
}

public class ConcreteStrategy2 : IStrategy
{
    public void Algorithm()
    {}
}
```

```
public class Context
{
    public IStrategy ContextStrategy { get; set; }

    public Context(IStrategy _strategy)
    {
        ContextStrategy = _strategy;
    }

    public void ExecuteAlgorithm()
    {
        ContextStrategy.Algorithm();
    }
}
```

РАССМОТРИМ НА ПРИМЕРЕ

Существуют различные легковые машины, которые используют разные источники энергии: электричество, бензин, газ и так далее.

Есть гибридные автомобили. В целом они похожи и отличаются преимущественно видом источника энергии. Не говоря уже о том, что мы можем изменить применяемый источник энергии, модифицировав автомобиль.

```
interface IMovable
{
    void Move();
}

class PetrolMove : IMovable
{
    public void Move() ...
}

class ElectricMove : IMovable
{
    public void Move() ...
}
```

```
class Car
{
    protected int passengers; // кол-во пассажиров
    protected string model; // модель автомобиля

    public Car(int num, string model, IMovable mov)
    {
        this.passengers = num;
        this.model = model;
        Movable = mov;
    }

    public IMovable Movable { private get; set; }
    public void Move()
    {
        Movable.Move();
    }
}
```

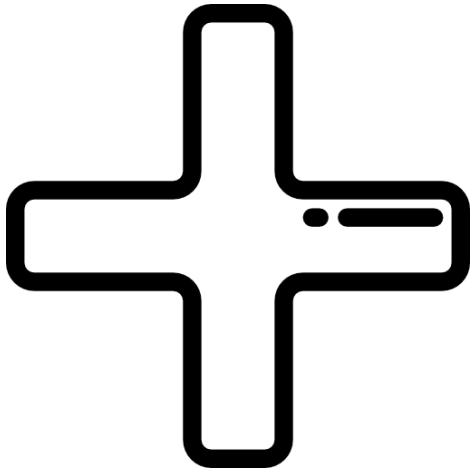
```
interface IMovable
{
    void Move();
}

class PetrolMove : IMovable
{
    public void Move() ...
}

class ElectricMove : IMovable
{
    public void Move() ...
}
```

```
class Car
{
    protected int passengers; // кол-во пассажиров
    protected string model; // модель автомобиля

    public Car(int num, string model, IMovable mov)
    {
        this.passengers = num;
        this.model = model;
        Movable = mov;
    }

    public IMovable Movable { private get; set; }
    public void Move()
    {
        Movable.Move();
    }
}

Car auto = new Car(4, "Volvo", new PetrolMove());
auto.Move();
auto.Movable = new ElectricMove();
auto.Move();
```

ПРИМЕНИМОСТЬ

- Когда вам нужно использовать разные вариации какого-то алгоритма внутри одного объекта.
- Когда у вас есть множество похожих классов, отличающихся только некоторым поведением.
- Когда вы не хотите обнажать детали реализации алгоритмов для других классов.
- Когда различные вариации алгоритмов реализованы в виде развесистого условного оператора. Каждая ветка такого оператора представляет собой вариацию алгоритма.

ПРЕИМУЩЕСТВА И НЕДОСТАТКИ



Горячая замена алгоритмов на лету.

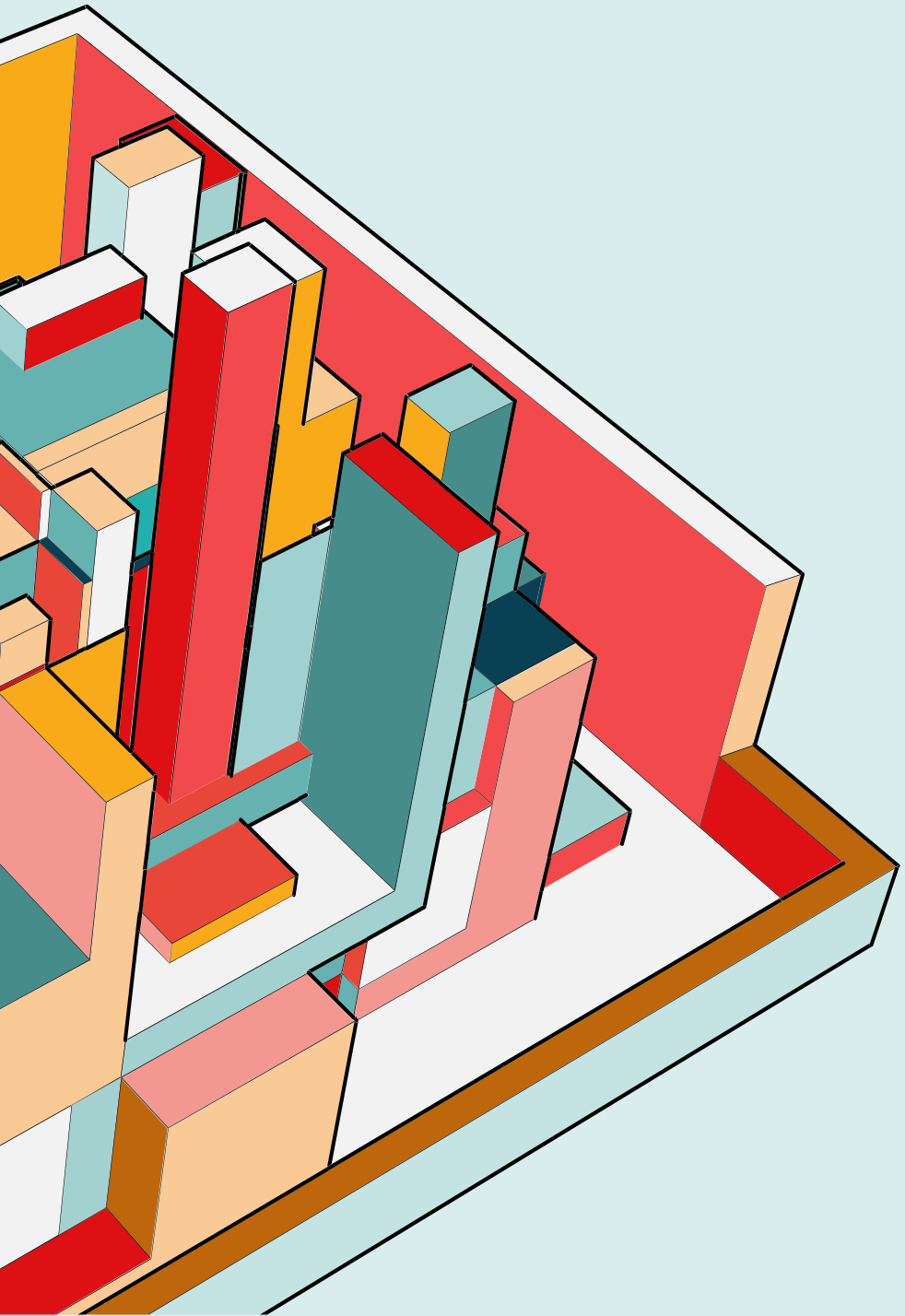
Изолирует код и данные алгоритмов от остальных классов.

Уход от наследования к делегированию.

Реализует принцип открытости/закрытости.

Усложняет программу за счёт дополнительных классов.

Клиент должен знать, в чём состоит разница между стратегиями, чтобы выбрать подходящую.

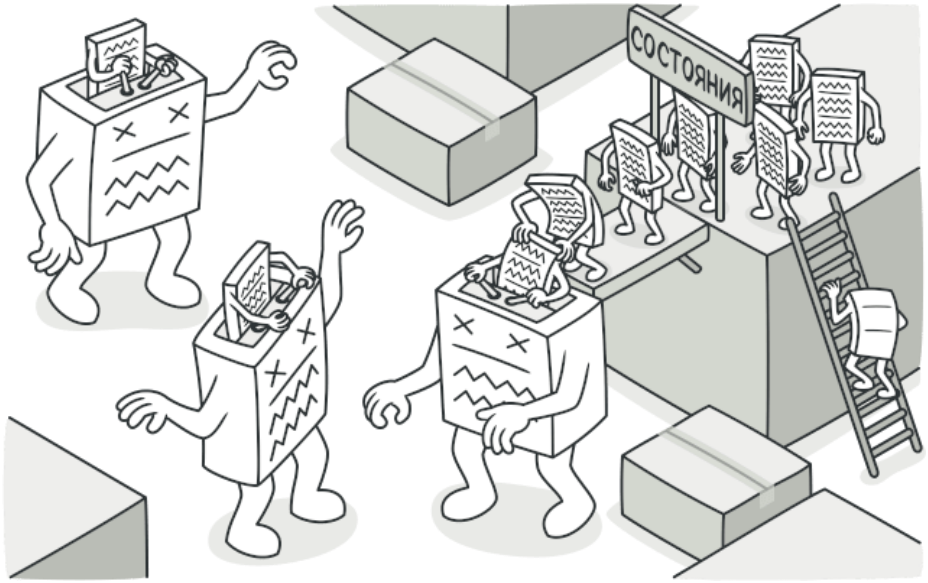


ПОВЕДЕНЧЕСКИЕ ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

State

СОСТОЯНИЕ

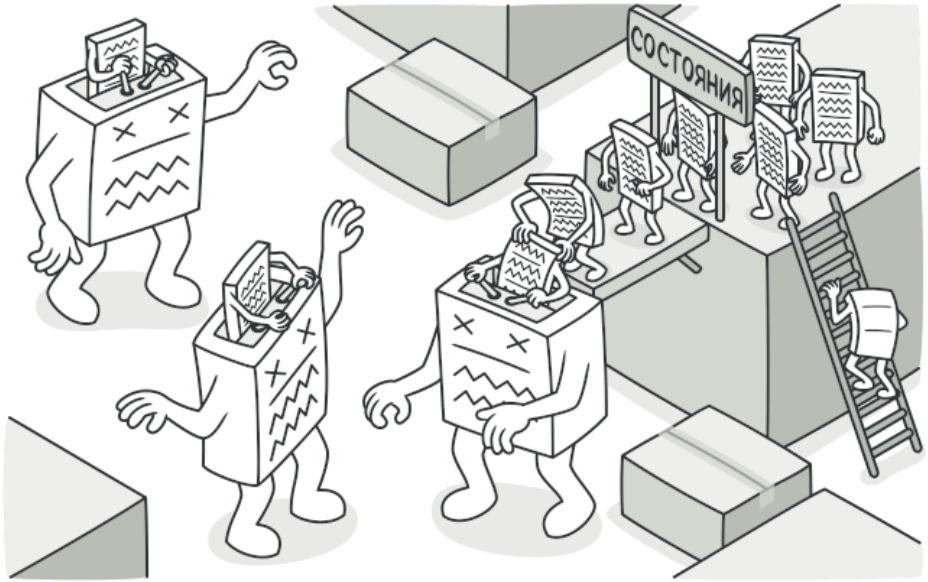
Позволяет объектам менять поведение в зависимости от своего состояния. Извне создаётся впечатление, что изменился класс объекта.



Находясь в разных состояниях, программа может по-разному реагировать на одни и те же события, которые происходят с ней.

СОСТОЯНИЕ

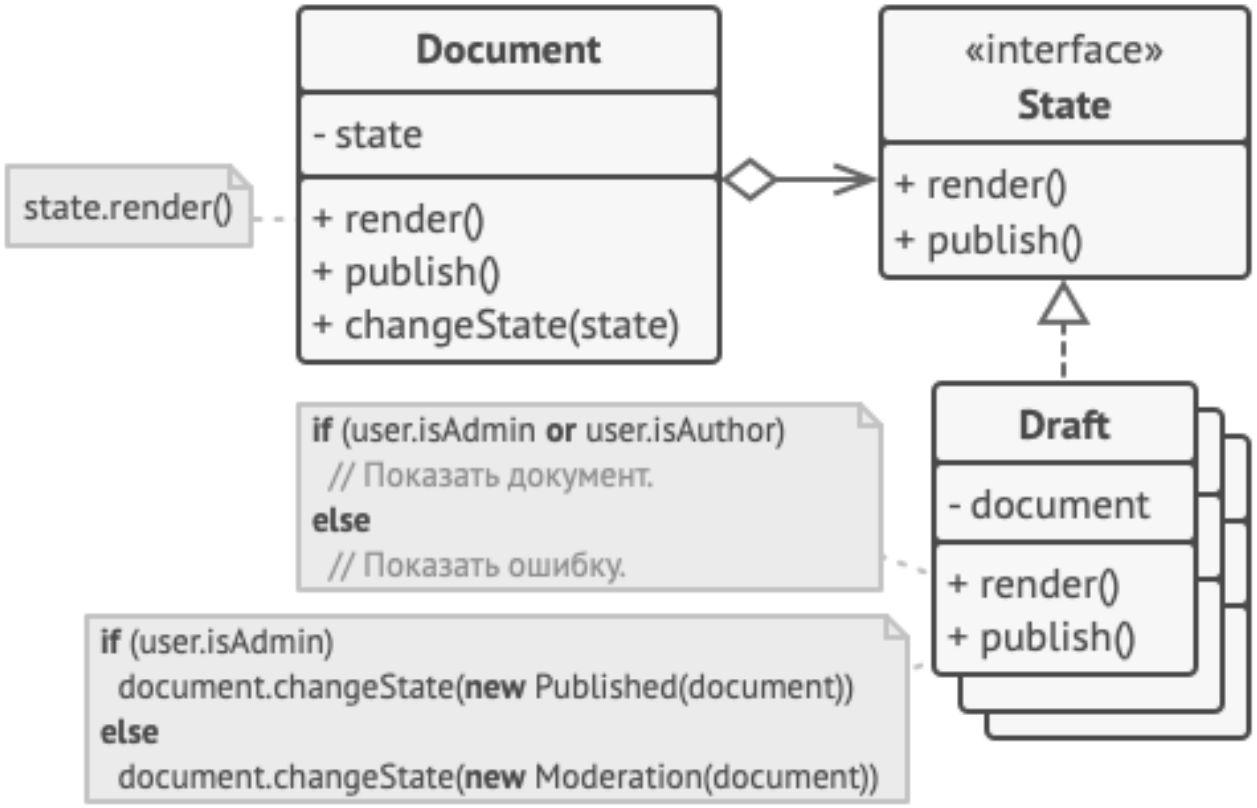
Позволяет объектам менять поведение в зависимости от своего состояния. Извне создаётся впечатление, что изменился класс объекта.



Находясь в разных состояниях, программа может по-разному реагировать на одни и те же события, которые происходят с ней.

Создать классы для каждого состояния, в котором может пребывать объект, а затем вынести туда поведения, соответствующие этим состояниям

РЕШЕНИЕ

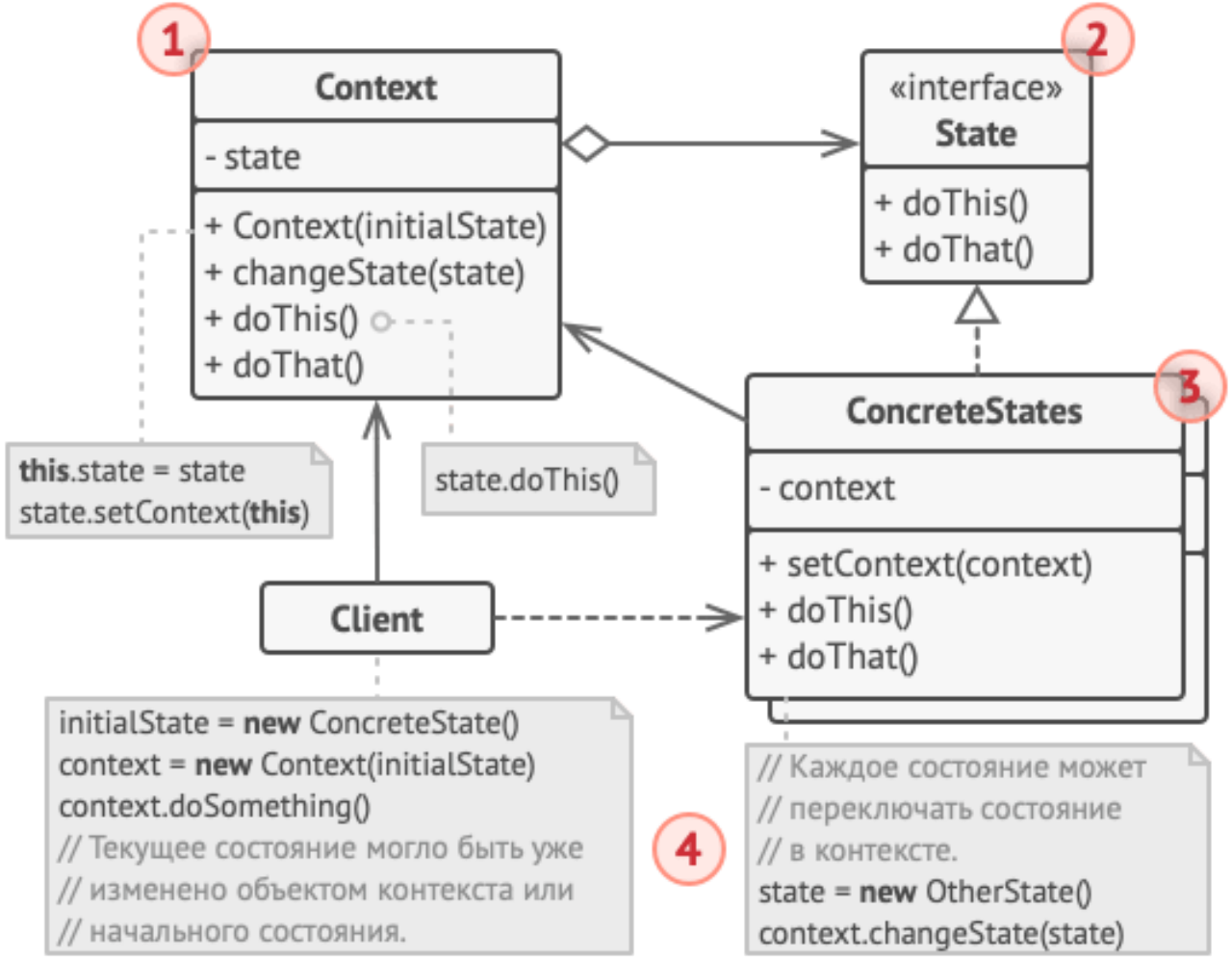


Документ делегирует работу своему активному объекту-состоянию.

Вместо того, чтобы хранить код всех состояний, первоначальный объект, называемый контекстом, будет содержать ссылку на один из объектов-состояний и делегировать ему работу, зависящую от состояния.

СТРУКТУРА

1. **Контекст** хранит ссылку на объект состояния и делегирует ему часть работы, зависящей от состояний.
2. **Состояние** описывает общий интерфейс для всех конкретных состояний.
3. **Конкретные состояния** реализуют поведения, связанные с определённым состоянием контекста.
4. И контекст, и объекты конкретных состояний могут решать, когда и какое следующее состояние будет выбрано.



```
abstract class State
{
    public abstract void Handle(Context context);
}
class StateA : State
{
    public override void Handle(Context context)
    {
        context.State = new StateB();
    }
}
class StateB : State
{
    public override void Handle(Context context) ...
}

class Context
{
    public State State { get; set; }
    public Context(State state) ...
    public void Request()
    {
        this.State.Handle(this);
    }
}
```

```
abstract class State
{
    public abstract void Handle(Context context);
}
class StateA : State
{
    public override void Handle(Context context)
    {
        context.State = new StateB();
    }
}
class StateB : State
{
    public override void Handle(Context context) ...
}

class Context
{
    public State State { get; set; }
    public Context(State state) ...
    public void Request()
    {
        this.State.Handle(this);
    }
}
```

```
Context context = new Context(new StateA());
context.Request(); // Переход в состояние StateB
context.Request();  // Переход в состояние StateA
```

РАССМОТРИМ НА ПРИМЕРЕ

Например, вода может находиться в ряде состояний: твердое, жидкое, парообразное. Допустим, нам надо определить класс Вода, у которого бы имелись методы для нагревания и заморозки воды.

```
class Water
{
    public WaterState State { get; set; }

    public Water(WaterState ws) ...

    public void Heat()
    {
        if (State == WaterState.SOLID)
        {
            Console.WriteLine("Превращаем лед в жидкость");
            State = WaterState.LIQUID;
        }
        else if (State == WaterState.LIQUID) ...
        else if (State == WaterState.GAS) ...
    }
    public void Frost()
    {
        if (State == WaterState.LIQUID) ...
        else if (State == WaterState.GAS) ...
    }
}
```

```
class Water
{
    public WaterState State { get; set; }

    public Water(WaterState ws) ...

    public void Heat()
    {
        if (State == WaterState.SOLID)
        {
            Console.WriteLine("Превращаем лед в жидкость");
            State = WaterState.LIQUID;
        }
        else if (State == WaterState.LIQUID) ...
        else if (State == WaterState.GAS) ...
    }
    public void Frost()
    {
        if (State == WaterState.LIQUID) ...
        else if (State == WaterState.GAS) ...
    }
}
```

```
Water water =
    new Water(WaterState.LIQUID);
water.Heat();
water.Frost();
water.Frost();
```

```
class Water
{
    public IWaterState State { get; set; }

    public Water(IWaterState ws)
    {
        State = ws;
    }

    public void Heat()
    {
        State.Heat(this);
    }
    public void Frost()
    {
        State.Frost(this);
    }
}
```

```
interface IWaterState
{
    void Heat(Water water);
    void Frost(Water water);
}

class SolidWaterState : IWaterState
{
    public void Heat(Water water)
    {
        Console.WriteLine("Превращаем лед в жидкость");
        water.State = new LiquidWaterState();
    }

    public void Frost(Water water)
    {
        Console.WriteLine("Продолжаем заморозку льда");
    }
}
class LiquidWaterState ...
class GasWaterState ...
```

```
class Water
{
    public IWaterState State { get; set; }

    public Water(IWaterState ws)
    {
        State = ws;
    }

    public void Heat()
    {
        State.Heat(this);
    }
    public void Frost()
    {
        State.Frost(this);
    }
}
```

```
interface IWaterState
{
    void Heat(Water water);
    void Frost(Water water);
}

class SolidWaterState : IWaterState
{
    public void Heat(Water water)
    {
        Console.WriteLine("Превращаем лед в жидкость");
        water.State = new LiquidWaterState();
    }

    public void Frost(Water water)
    {
        Console.WriteLine("Продолжаем заморозку льда");
    }
}
class LiquidWaterState ...
class GasWaterState ...
```

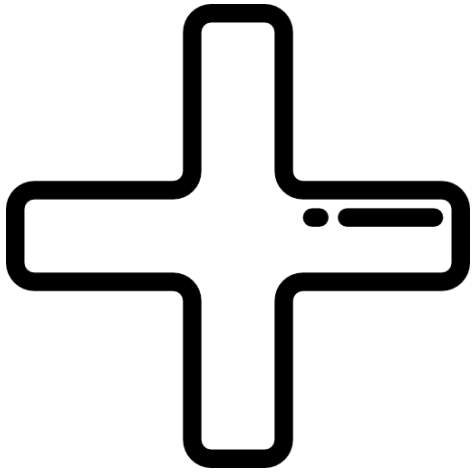
```
class Water
{
    public IWaterState State { get; set; }

    public Water(IWaterState ws)
    {
        State = ws;
    }

    public void Heat()
    {
        State.Heat(this);
    }
    public void Frost()
    {
        State.Frost(this);
    }
}
```

```
Water water =
    new Water(WaterState.LIQUID);
water.Heat();
water.Frost();
water.Frost();
```

ПРИМЕНИМОСТЬ

- Когда у вас есть объект, поведение которого кардинально меняется в зависимости от внутреннего состояния.
- Когда код класса содержит множество больших, похожих друг на друга, условных операторов, которые выбирают поведения в зависимости от текущих значений полей класса.
- Когда вы сознательно используете табличную машину состояний, построенную на условных операторах, но вынуждены мириться с дублированием кода для похожих состояний и переходов.

ПРЕИМУЩЕСТВА И НЕДОСТАТКИ

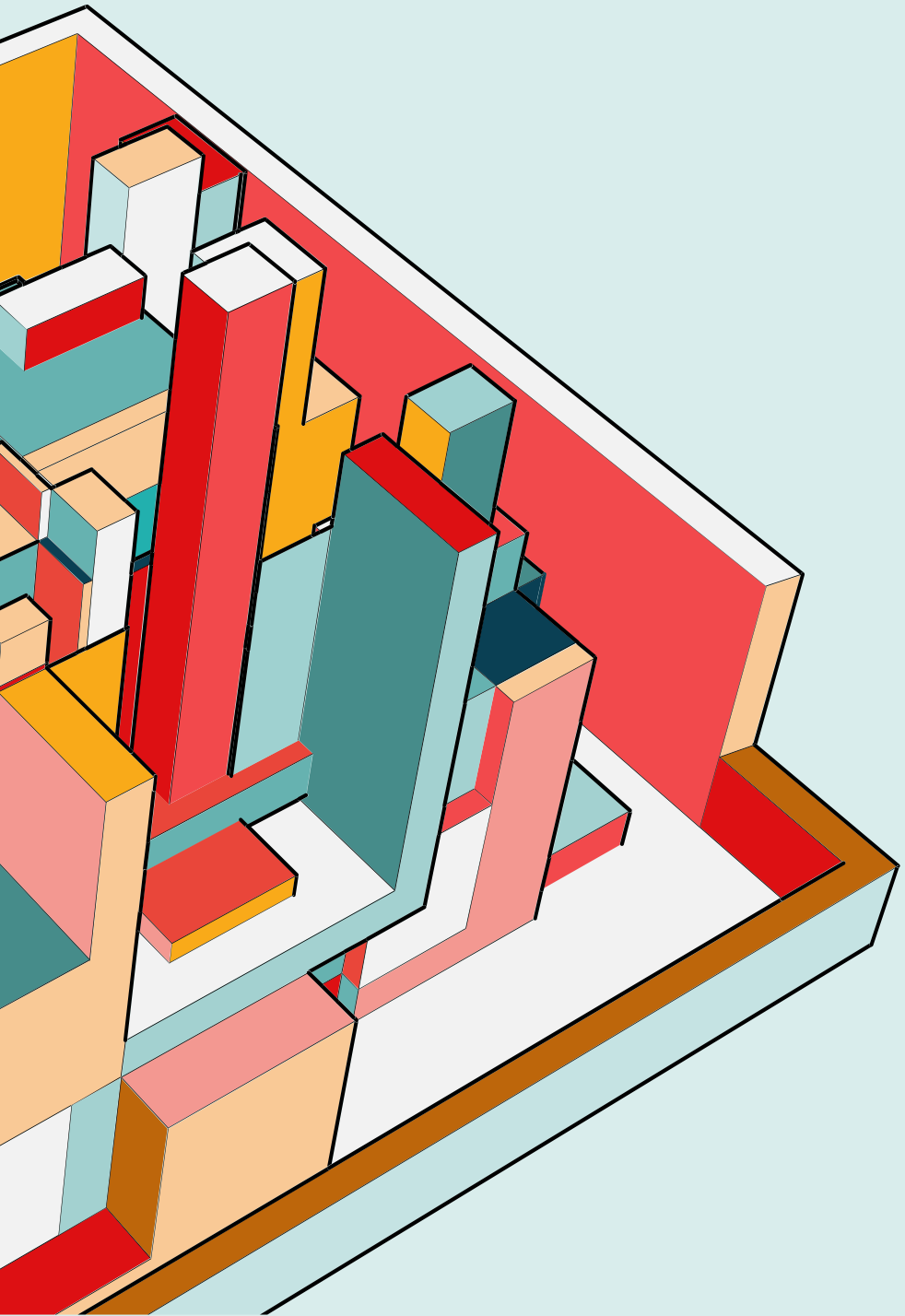


Избавляет от множества больших условных операторов

Концентрирует в одном месте код, связанный с определённым состоянием.

Упрощает код контекста.

Может неоправданно усложнить код, если состояний мало и они редко меняются.

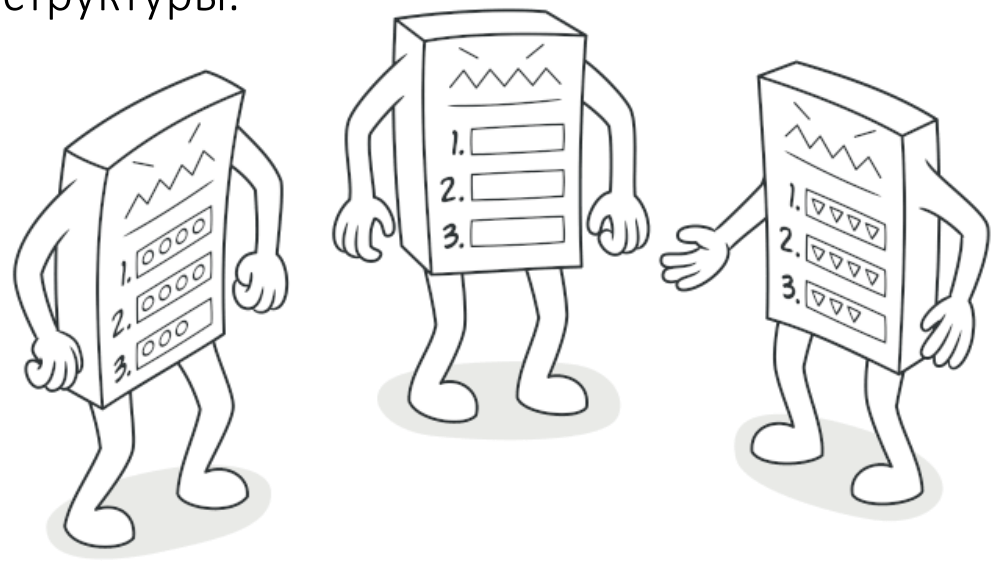


ПОВЕДЕНЧЕСКИЕ ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

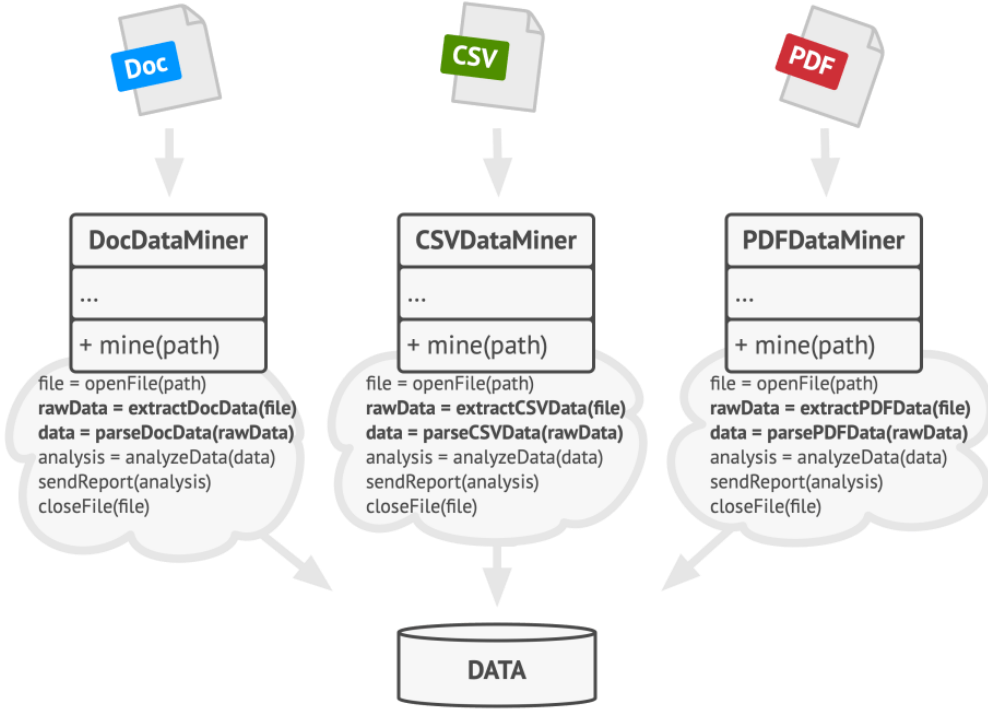
Template Method

ШАБЛОННЫЙ МЕТОД

Определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Паттерн позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.

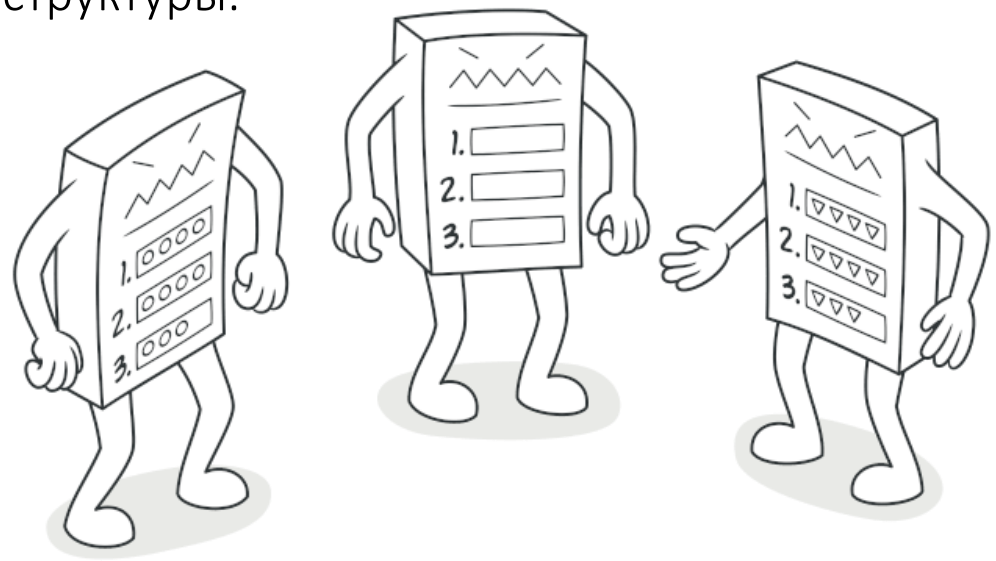


Вы пишете программу для дата-майнинга в офисных документах. Пользователи будут загружать в неё документы в разных форматах (PDF, DOC, CSV), а программа должна извлекать из них полезную информацию.

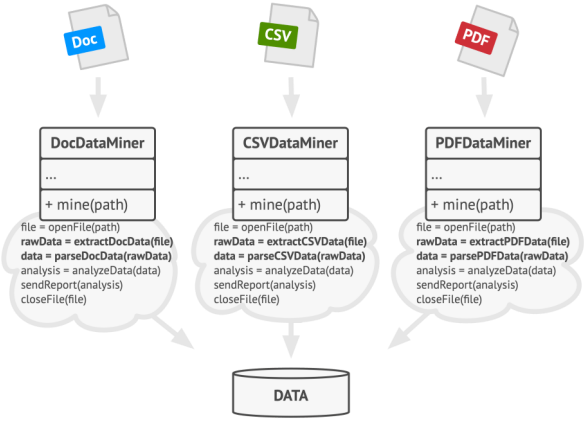


ШАБЛОННЫЙ МЕТОД

Определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Паттерн позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.

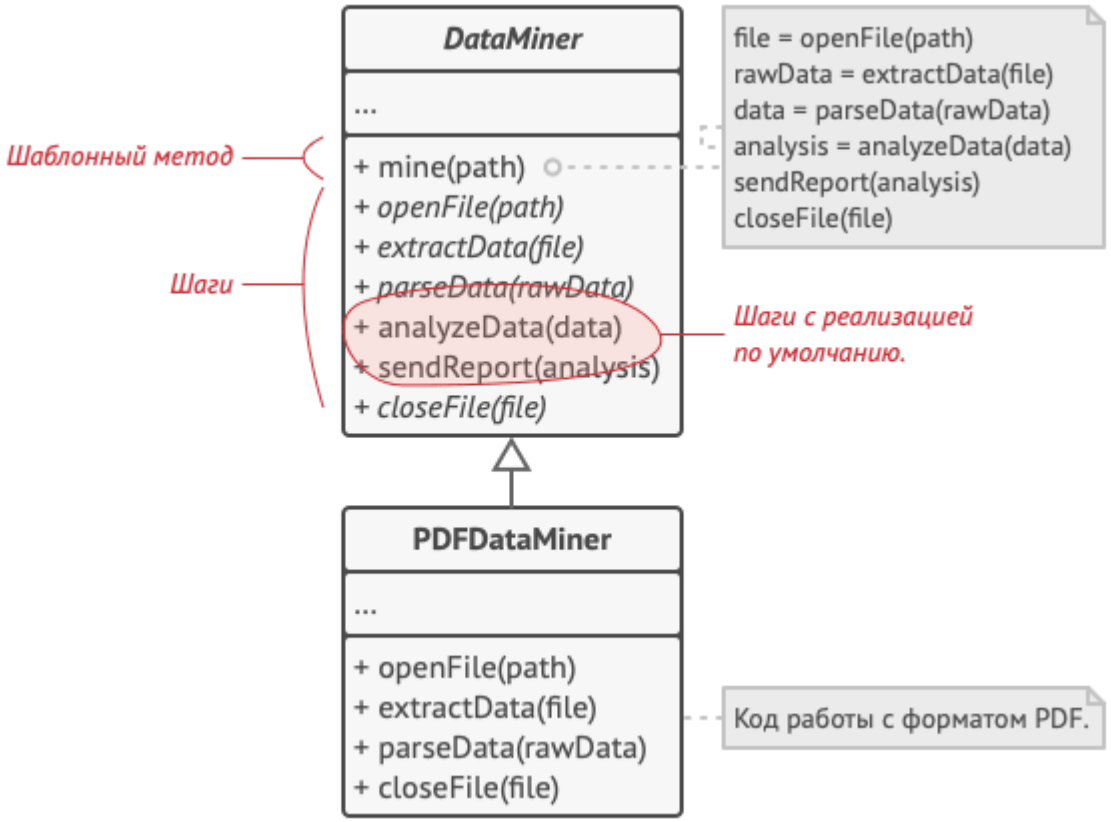


Вы пишете программу для дата-майнинга в офисных документах. Пользователи будут загружать в неё документы в разных форматах (PDF, DOC, CSV), а программа должна извлекать из них полезную информацию.



Паттерн Шаблонный метод предлагает разбить алгоритм на последовательность шагов, описать эти шаги в отдельных методах и вызывать их в одном шаблонном методе друг за другом.

РЕШЕНИЕ



Шаблонный метод разбивает алгоритм на шаги, позволяя подклассам переопределить некоторые из них.

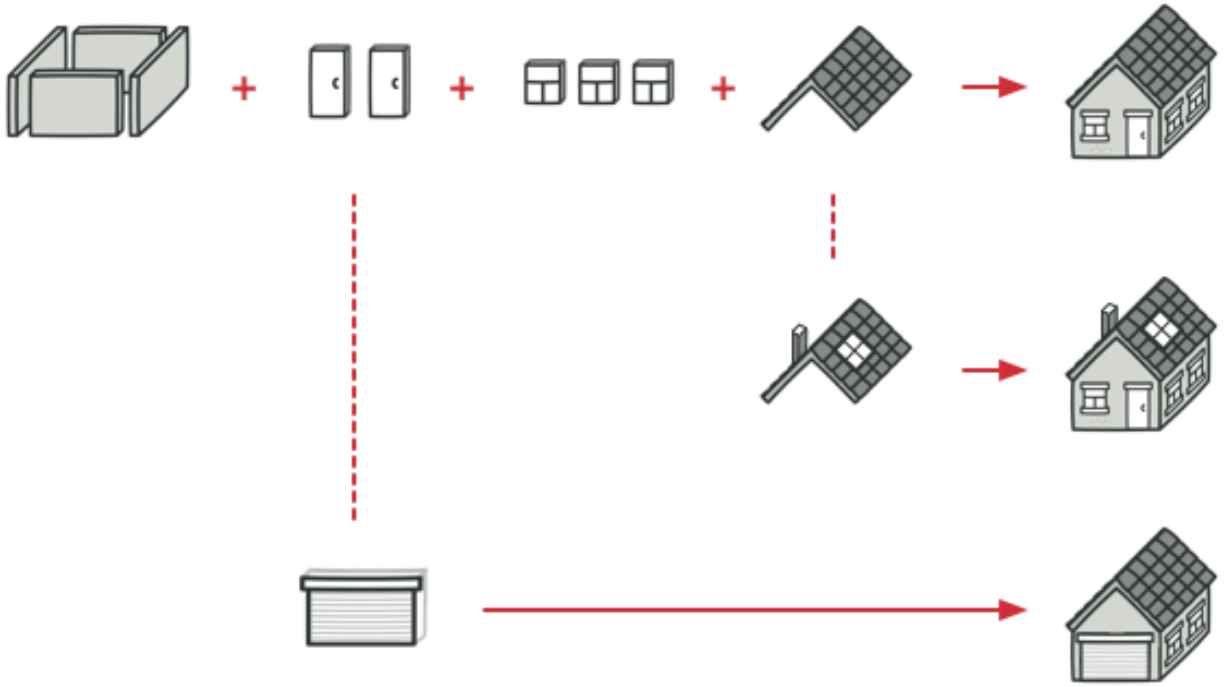
Мы можем **создать общий базовый класс для всех трёх алгоритмов**. Этот класс будет состоять из шаблонного метода, который последовательно вызывает шаги разбора документов.

Для начала шаги шаблонного метода можно сделать абстрактными.

В последующем мы можем определить общее для всех классов поведение и вынести его в суперкласс.

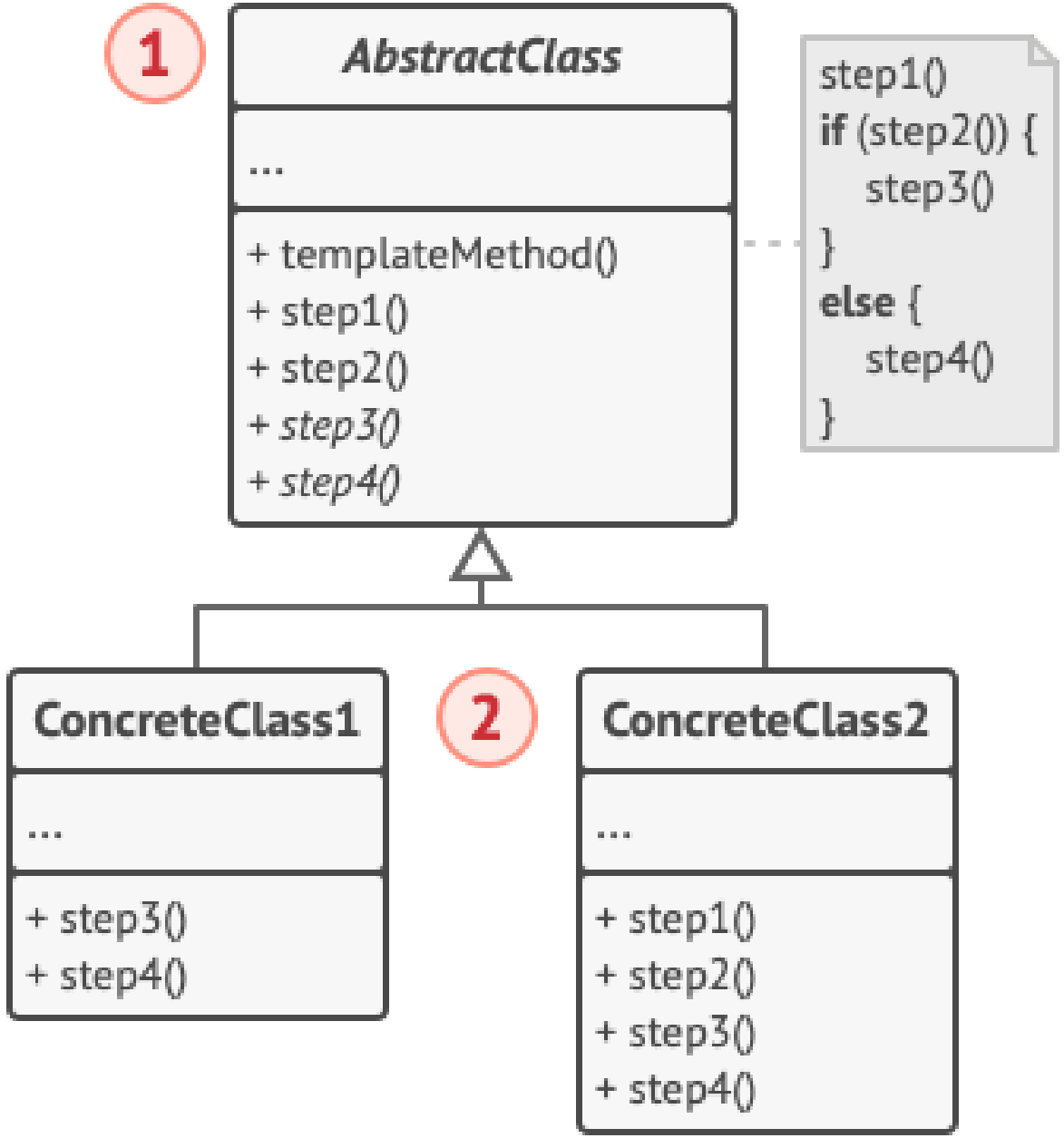
ПРОЕКТ ТИПОВОГО ДОМА МОГУТ НЕМНОГО ИЗМЕНИТЬ ПО ЖЕЛАНИЮ КЛИЕНТА

Несмотря на стандартизацию каждого этапа, строители могут вносить небольшие изменения на любом из этапов, чтобы сделать дом чуточку непохожим на другие



СТРУКТУРА

1. **Абстрактный класс** определяет шаги алгоритма и содержит шаблонный метод, состоящий из вызовов этих шагов. Шаги могут быть как абстрактными, так и содержать реализацию по умолчанию.
2. **Конкретный класс** переопределяет некоторые (или все) шаги алгоритма. Конкретные классы не переопределяют сам шаблонный метод.



```
abstract class AbstractClass
{
    public void TemplateMethod()
    {
        Operation1();
        Operation2();
    }
    public abstract void Operation1();
    public abstract void Operation2();
}

class ConcreteClass : AbstractClass
{
    public override void Operation1()
    {
    }

    public override void Operation2()
    {
    }
}
```

РАССМОТРИМ НА ПРИМЕРЕ

Рассмотрим применение паттерна на примере организации учебы в школе и в вузе. Эти классы очень похожи, и самое главное, реализуют примерно общий алгоритм.

Да, где-то будет отличаться реализация методов, где-то чуть больше методов, но в целом мы имеем общий алгоритм, а функциональность обоих классов по большому счету дублируется.

Поэтому для улучшения структуры классов мы могли бы применить шаблонный метод

```
abstract class Education
{
    public void Learn()
    {
        Enter();
        Study();
        PassExams();
        GetDocument();
    }
    public abstract void Enter();
    public abstract void Study();
    public virtual void PassExams()
    {
        Console.WriteLine("Сдаем выпускные экзамены");
    }
    public abstract void GetDocument();
}
```

```
abstract class Education
{
    public void Learn()
    {
        Enter();
        Study();
        PassExams();
        GetDocument();
    }
    public abstract void Enter();
    public abstract void Study();
    public virtual void PassExams()
    {
        Console.WriteLine("Сдаем выпускные экзамены");
    }
    public abstract void GetDocument();
}
```

```
class School : Education
{
    public override void Enter()
    {
        Console.WriteLine("Идем в первый класс");
    }

    public override void Study() ...
    public override void GetDocument() ...
}
```

```
class University : Education
{
    public override void Enter()
    {
        Console.WriteLine("Поступаем в ВУЗ");
    }

    public override void Study() ...
    public override void PassExams() ...
    public override void GetDocument() ...
}
```

```
abstract class Education
{
    public void Learn()
    {
        Enter();
        Study();
        PassExams();
        GetDocument();
    }
    public abstract void Enter();
    public abstract void Study();
    public virtual void PassExams()
    {
        Console.WriteLine("Сдаем выпускные экзамены");
    }
    public abstract void GetDocument();
}
```

```
class School : Education
{
    public override void Enter()
    {
        Console.WriteLine("Идем в первый класс");
    }

    public override void Study() ...
    public override void GetDocument() ...
}
```

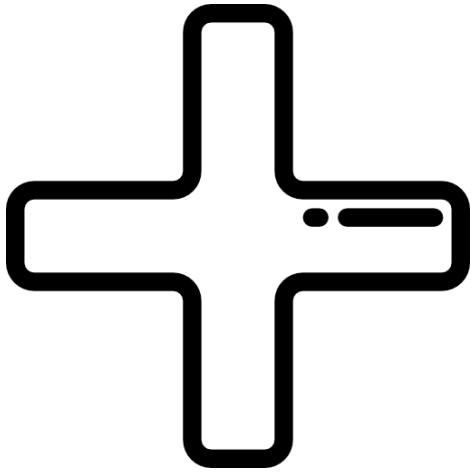
```
class University : Education
{
    public override void Enter()
    {
        Console.WriteLine("Поступаем в ВУЗ");
    }

    public override void Study() ...
    public override void PassExams() ...
    public override void GetDocument() ...
}
```

```
School school = new School();
University university = new University();

school.Learn();
university.Learn();
```

ПРИМЕНИМОСТЬ

- Когда подклассы должны расширять базовый алгоритм, не меняя его структуры.
- Когда у вас есть несколько классов, делающих одно и то же с незначительными отличиями. Если вы редактируете один класс, то приходится вносить такие же правки и в остальные классы.

ПРЕИМУЩЕСТВА И НЕДОСТАТКИ

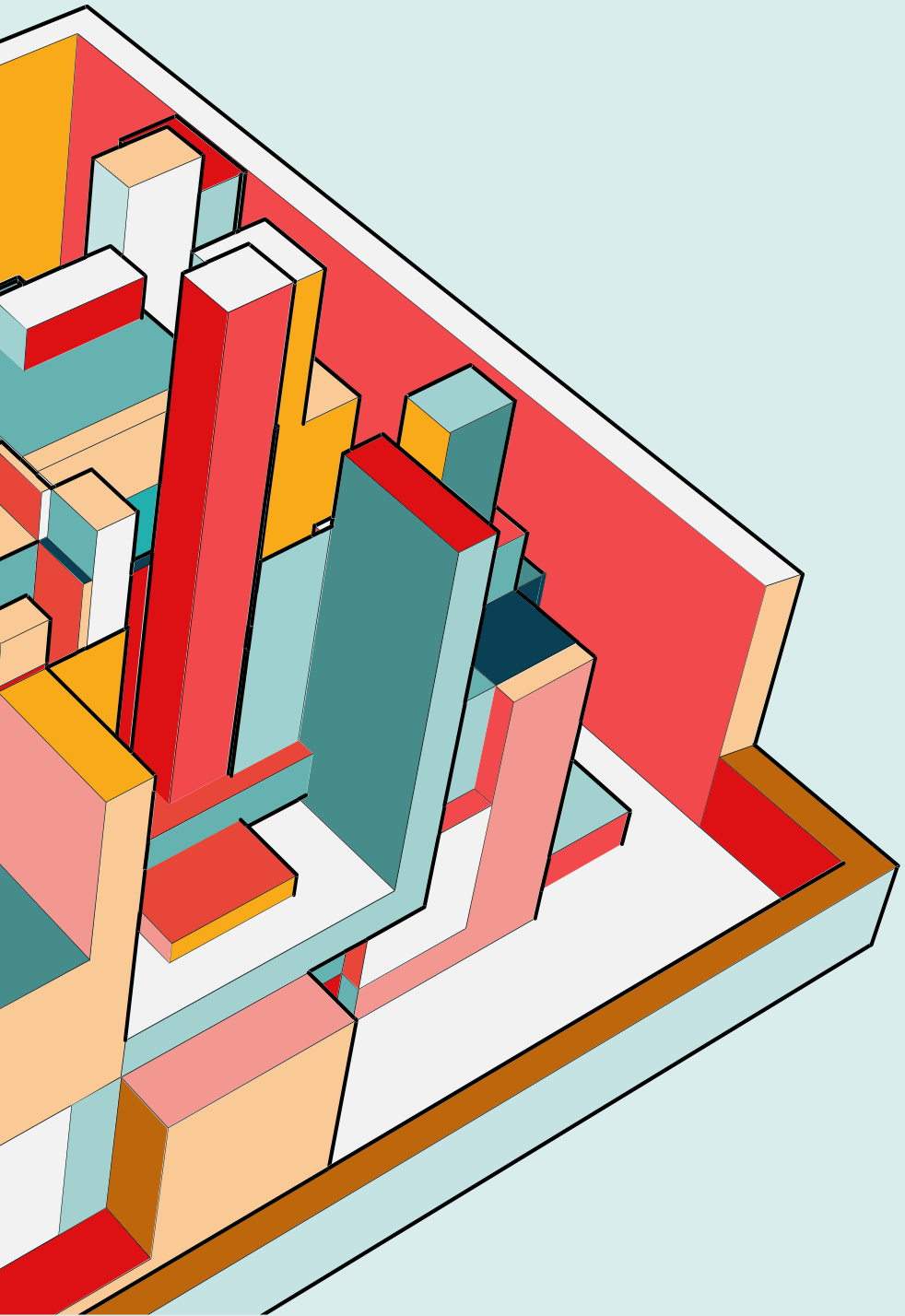


Облегчает повторное использование кода.

Вы жёстко ограничены скелетом существующего алгоритма.

Вы можете нарушить принцип подстановки Лисков, изменяя базовое поведение одного из шагов алгоритма через подкласс.

С ростом количества шагов шаблонный метод становится слишком сложно поддерживать.

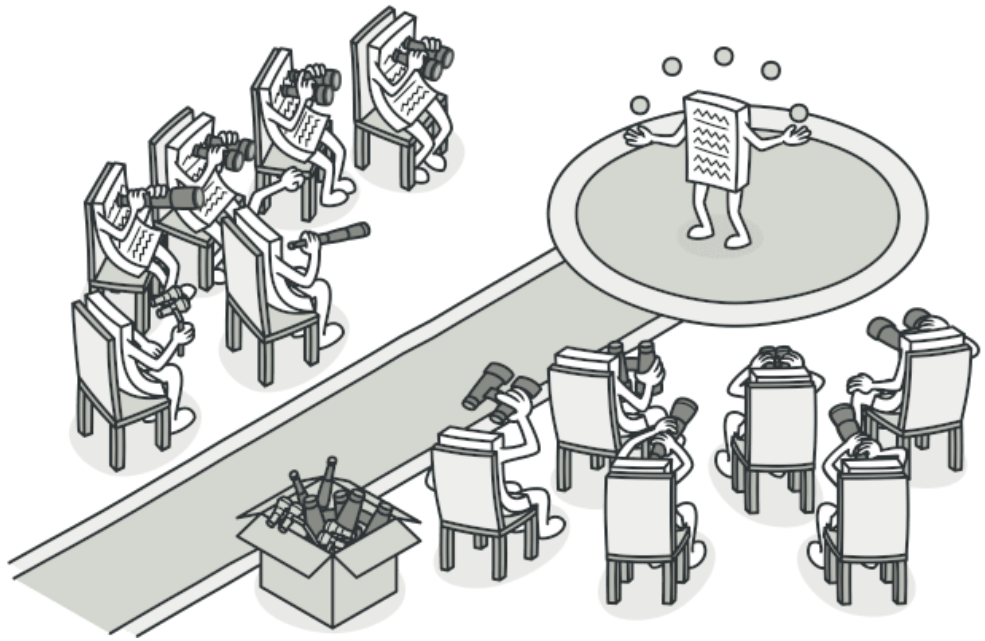


ПОВЕДЕНЧЕСКИЕ ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

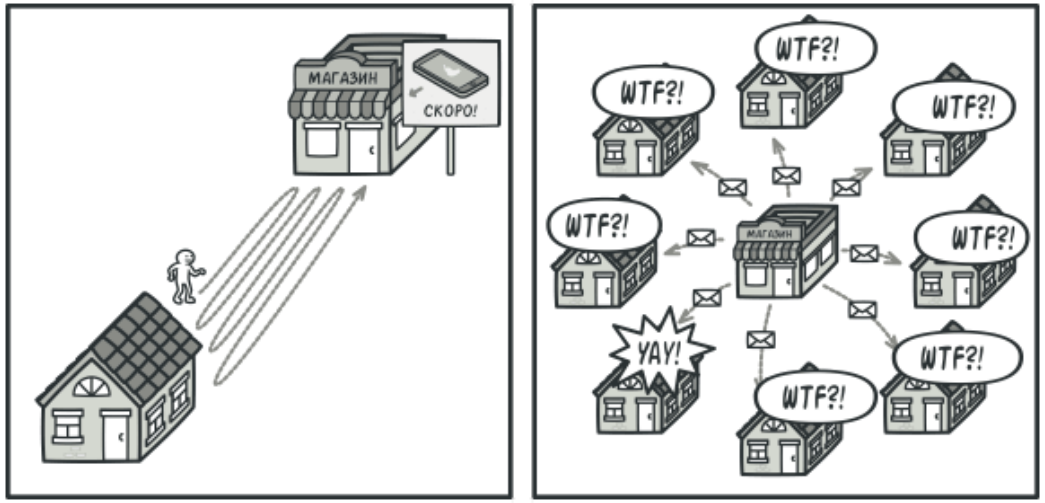
Observer

НАБЛЮДАТЕЛЬ

Создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.



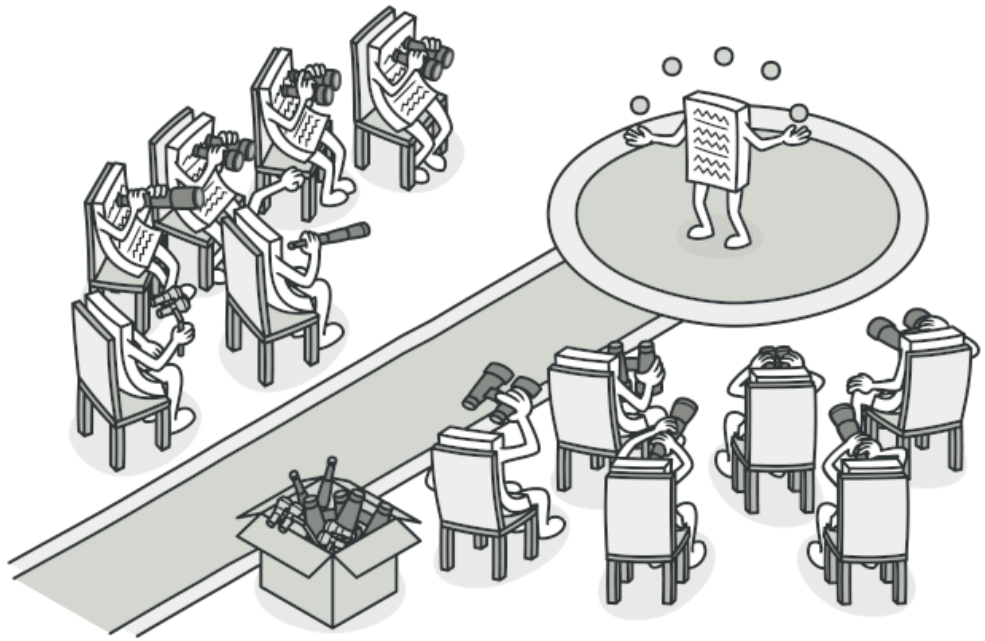
Представьте, что вы имеете два объекта: *Покупатель* и *Магазин*. В магазин вот-вот должны завезти новый товар, который интересен покупателю.



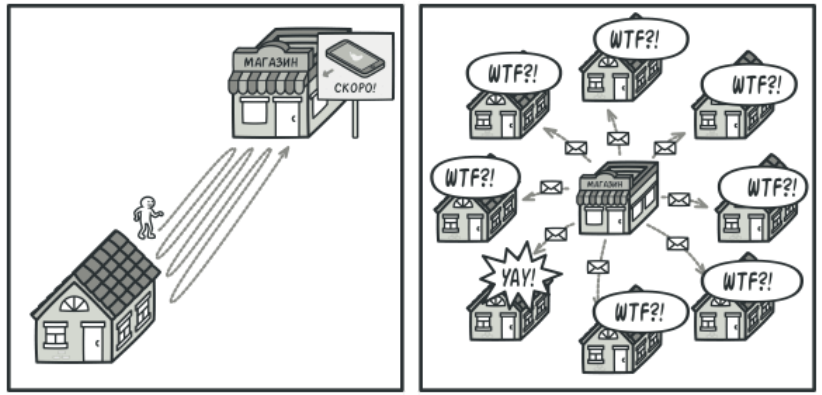
Каждое решение плохо по-своему:
либо покупатель тратит время на периодические проверки, либо магазин тратит ресурсы на бесполезные оповещения.

НАБЛЮДАТЕЛЬ

Создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.



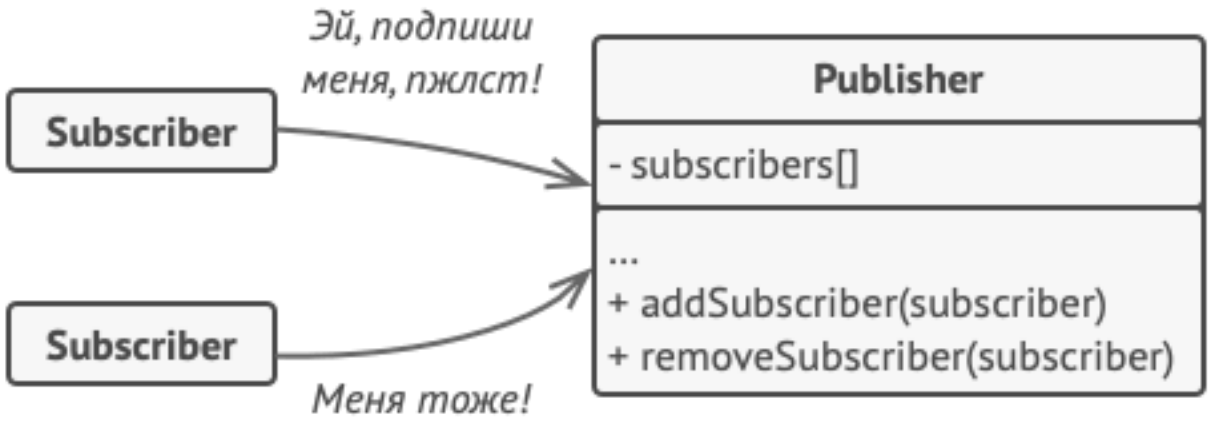
Представьте, что вы имеете два объекта: *Покупатель* и *Магазин*. В магазин вот-вот должны завезти новый товар, который интересен покупателю.



Либо покупатель тратит время на периодические проверки, либо магазин тратит ресурсы на бесполезные оповещения.

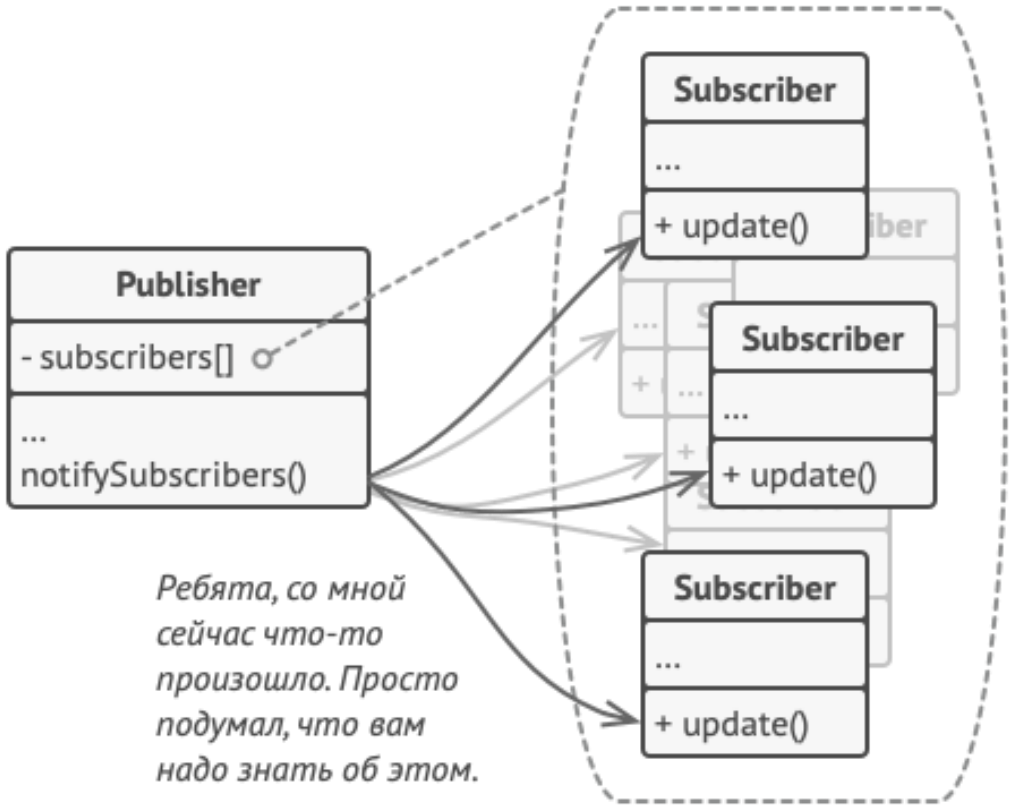
Паттерн Наблюдатель предлагает хранить внутри объекта издателя список ссылок на объекты подписчиков

РЕШЕНИЕ



Подписка на события.

Когда в издателе будет происходить важное событие, он будет проходиться по списку подписчиков и оповещать их об этом, вызывая определённый метод объектов-подписчиков.



Оповещения о событиях.

СТРУКТУРА

1. Издатель владеет внутренним состоянием, изменение которого интересно отслеживать подписчикам.

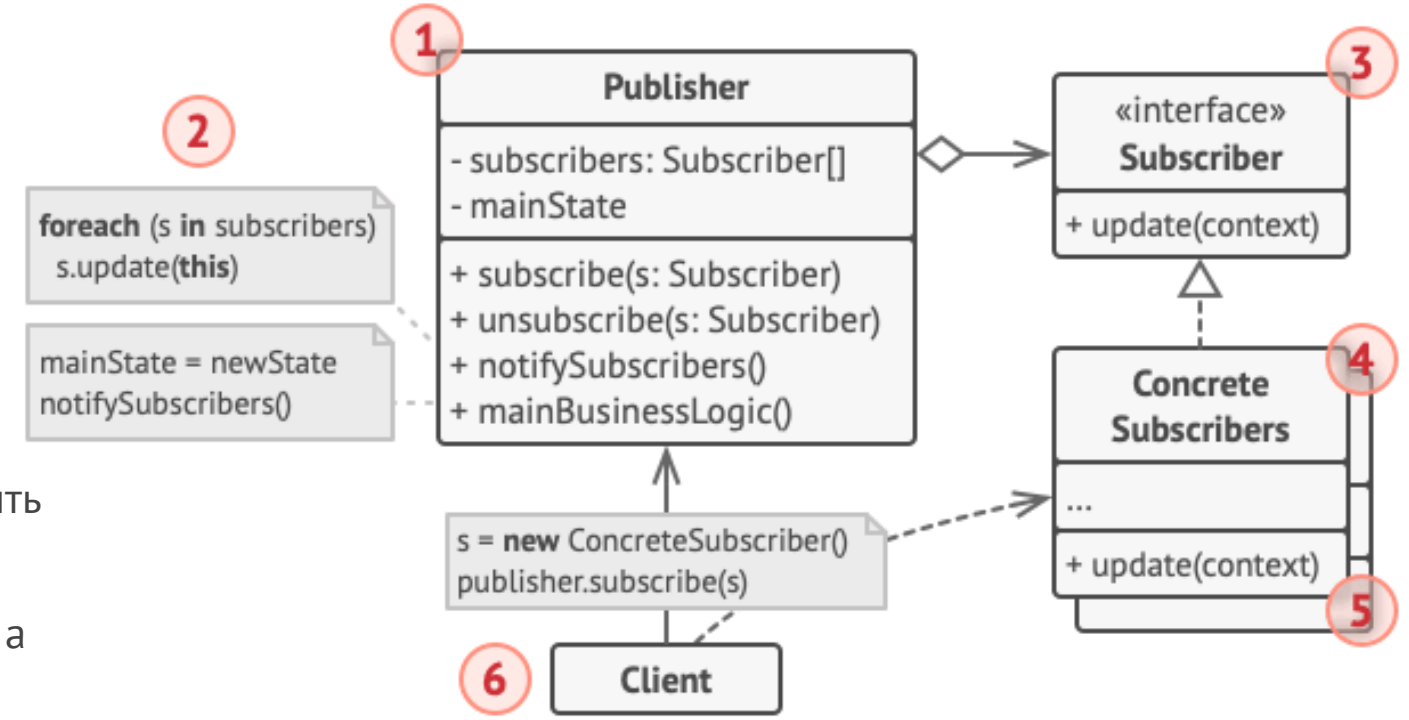
2. Когда внутреннее состояние издателя меняется, он оповещает своих подписчиков.

3. Подписчик определяет интерфейс, которым пользуется издатель для отправки оповещения.

4. Конкретные подписчики выполняют что-то в ответ на оповещение, пришедшее от издателя.

5. По приходу оповещения подписчику нужно получить обновлённое состояние издателя.

6. Клиент создаёт объекты издателей и подписчиков, а затем регистрирует подписчиков на обновления в издателях.



```
interface IObservable
{
    void AddObserver(IObserver o);
    void RemoveObserver(IObserver o);
    void NotifyObservers();
}

class ConcreteObservable : IObservable
{
    private List<IObserver> observers = new List<IObserver>();

    public void AddObserver(IObserver o) {
        observers.Add(o);
    }

    public void RemoveObserver(IObserver o) {
        observers.Remove(o);
    }

    public void NotifyObservers() {
        foreach (IObserver observer in observers)
            observer.Update();
    }
}
```

```
interface IObservable
{
    void AddObserver(IObserver o);
    void RemoveObserver(IObserver o);
    void NotifyObservers();
}

class ConcreteObservable : IObservable
{
    private List<IObserver> observers = new List<IObserver>();

    public void AddObserver(IObserver o) {
        observers.Add(o);
    }

    public void RemoveObserver(IObserver o) {
        observers.Remove(o);
    }

    public void NotifyObservers() {
        foreach (IObserver observer in observers)
            observer.Update();
    }
}
```

```
interface IObserver
{
    void Update();
}

class ConcreteObserver : IObserver
{
    public void Update() {    }
}
```

РАССМОТРИМ НА ПРИМЕРЕ

Допустим, у нас есть биржа, где проходят торги, и есть брокеры и банки, которые следят за поступающей информацией и в зависимости от поступившей информации производят определенные действия

```
interface IObservable
{
    void RegisterObserver(IObserver o);
    void RemoveObserver(IObserver o);
    void NotifyObservers();
}

class Stock : IObservable
{
    StockInfo sInfo = new StockInfo(); // информация о торгах
    List<IObserver> observers = new List<IObserver>();

    public void RegisterObserver(IObserver o) ...
    public void RemoveObserver(IObserver o) ...

    public void NotifyObservers()
    {
        foreach (IObserver o in observers) {
            o.Update(sInfo);
        }
    }

    public void Market()
    {
        Random rnd = new Random();
        sInfo.USD = rnd.Next(20, 40);
        sInfo.Euro = rnd.Next(30, 50);
        NotifyObservers();
    }
}
```

```
interface IObservable
{
    void RegisterObserver(IObserver o);
    void RemoveObserver(IObserver o);
    void NotifyObservers();
}

class Stock : IObservable
{
    StockInfo sInfo = new StockInfo(); // информация о торгах
    List<IObserver> observers = new List<IObserver>();

    public void RegisterObserver(IObserver o) ...
    public void RemoveObserver(IObserver o) ...

    public void NotifyObservers()
    {
        foreach (IObserver o in observers) {
            o.Update(sInfo);
        }
    }

    public void Market()
    {
        Random rnd = new Random();
        sInfo.USD = rnd.Next(20, 40);
        sInfo.Euro = rnd.Next(30, 50);
        NotifyObservers();
    }
}
```

```
interface IObserver
{
    void Update(Object ob);
}
class Broker : IObserver
{
    IObservable stock;
    public Broker(IObservable obs)
    {
        stock = obs;
        stock.RegisterObserver(this);
    }
    public void Update(object ob)
    {
        StockInfo sInfo = (StockInfo)ob;

        if (sInfo.USD > 30)
            Console.WriteLine("Брокер {0} продает доллары");
        else
            Console.WriteLine("Брокер {0} покупает доллары");
    }
    public void StopTrade()
    {
        stock.RemoveObserver(this);
        stock = null;
    }
}
```

```
interface IObservable
{
    void RegisterObserver(IObserver o);
    void RemoveObserver(IObserver o);
    void NotifyObservers();
}

class Stock : IObservable
{
    StockInfo sInfo = new Sto
    List<IObserver> observers

    public void RegisterObser
    public void RemoveObserve

    public void NotifyObserve
    {
        foreach (IObserver o
            o.Update(sInfo);
        }
    }

    public void Market()
    {
        Random rnd = new Rand
        sInfo.USD = rnd.Next(
        sInfo.Euro = rnd.Next
        NotifyObservers();
    }
}
```

```
interface IObserver
{
    void Update(Object ob);
}
class Broker : IObserver
{
```

```
class Bank : IObserver
{
    IObservable stock;
    public Bank(IObservable obs) ...
    public void Update(object ob)
    {
        StockInfo sInfo = (StockInfo)ob;

        if (sInfo.Euro > 40)
            Console.WriteLine("Банк {0} продает евро");
        else
            Console.WriteLine("Банк {0} покупает евро");
    }
}
```

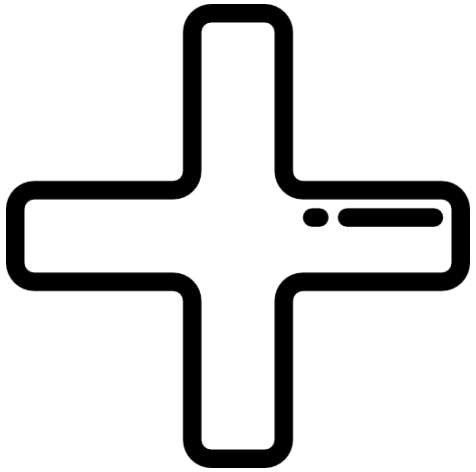
```
);

0} продает доллары");

0} покупает доллары");
```

```
Stock stock = new Stock();
Bank bank = new Bank(stock);
Broker broker = new Broker(stock);
// имитация торгов
stock.Market();
// брокер прекращает наблюдать за торгами
broker.StopTrade();
// имитация торгов
stock.Market();
```

ПРИМЕНИМОСТЬ

- Когда после изменения состояния одного объекта требуется что-то сделать в других, но вы не знаете наперёд, какие именно объекты должны отреагировать.
- Когда одни объекты должны наблюдать за другими, но только в определённых случаях.

ПРЕИМУЩЕСТВА И НЕДОСТАТКИ

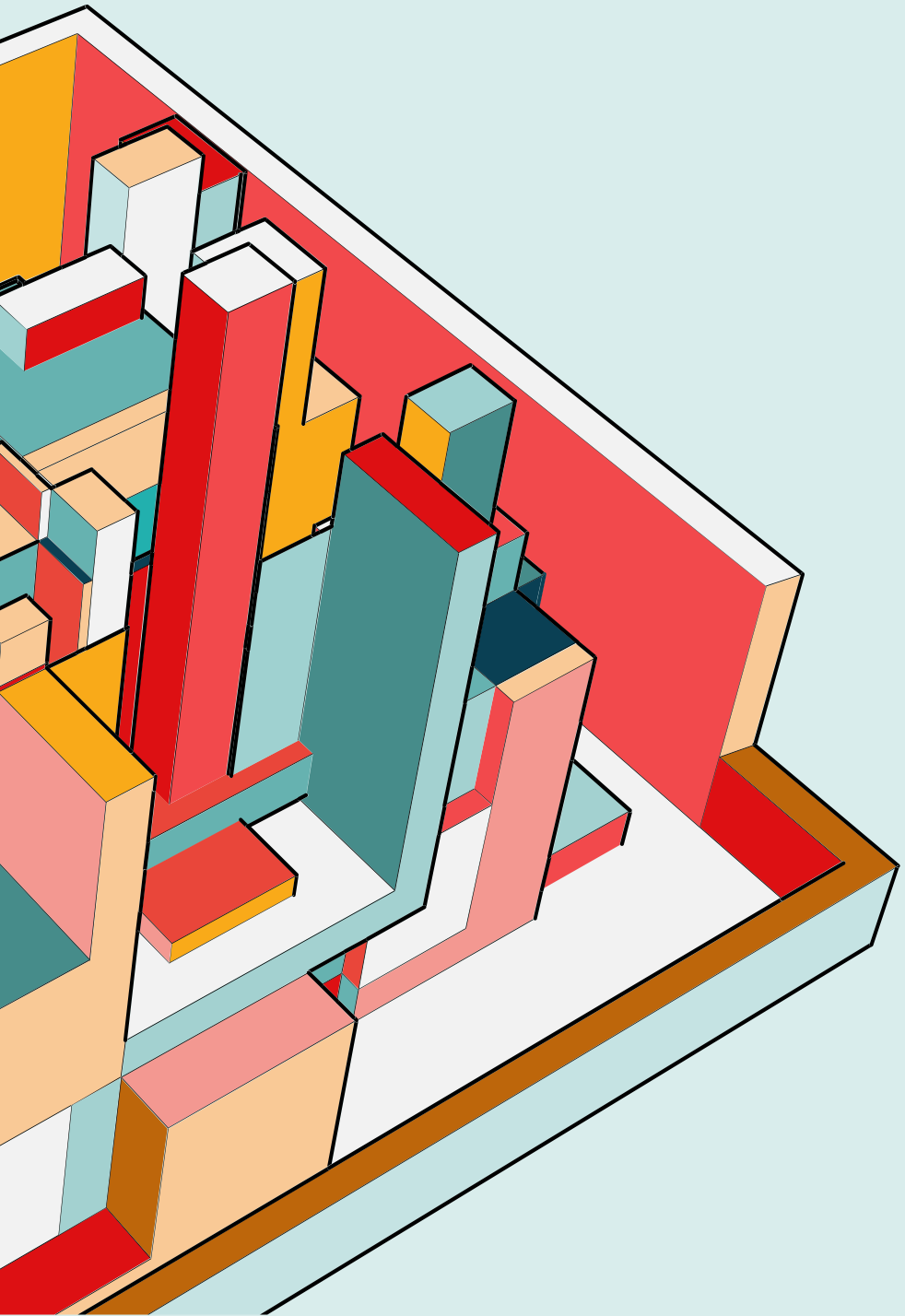


Издатели не зависят от конкретных классов подписчиков и наоборот.

Вы можете подписывать и отписывать получателей на лету.

Реализует *принцип открытости/закрытости*.

Подписчики оповещаются в случайном порядке.



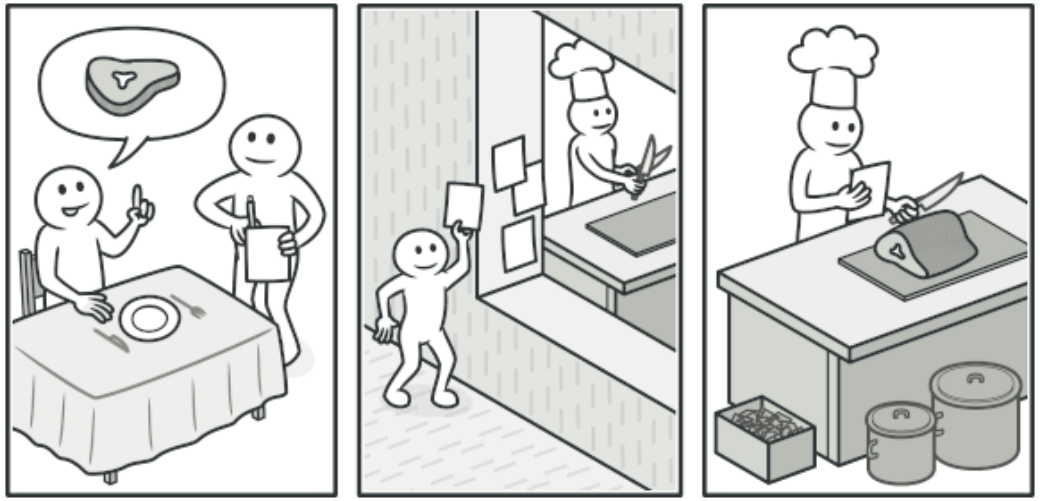
ПОВЕДЕНЧЕСКИЕ ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

Command

Также известен как: Действие, Транзакция, Action

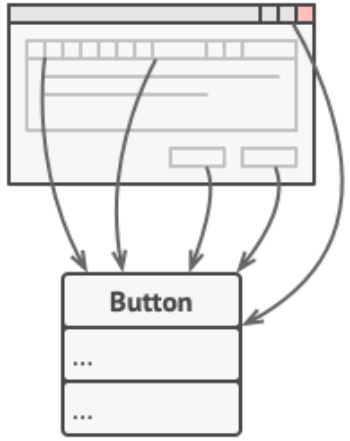
КОМАНДА

Превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций.

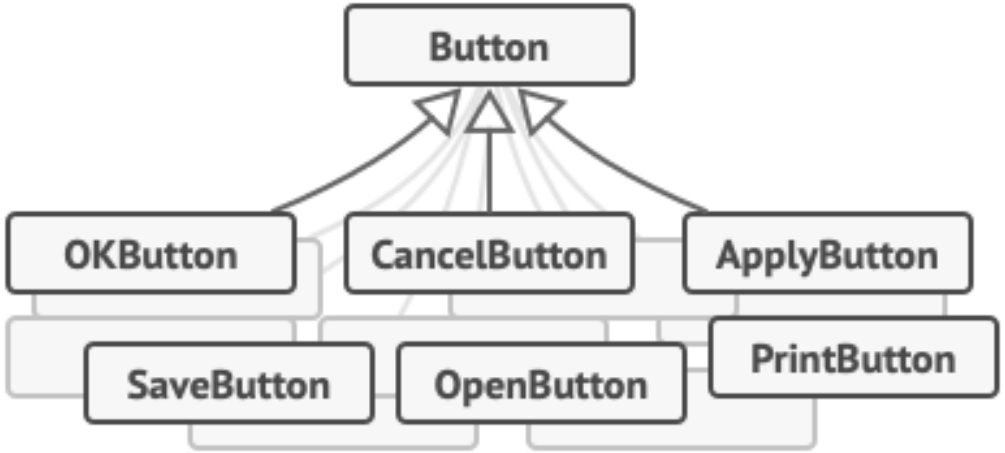


В этом примере посетитель является отправителем команды, официант с блокнотом — командой, а повар — получателем.

Вы создали класс красивых Кнопок и хотите использовать его для всех кнопок приложения

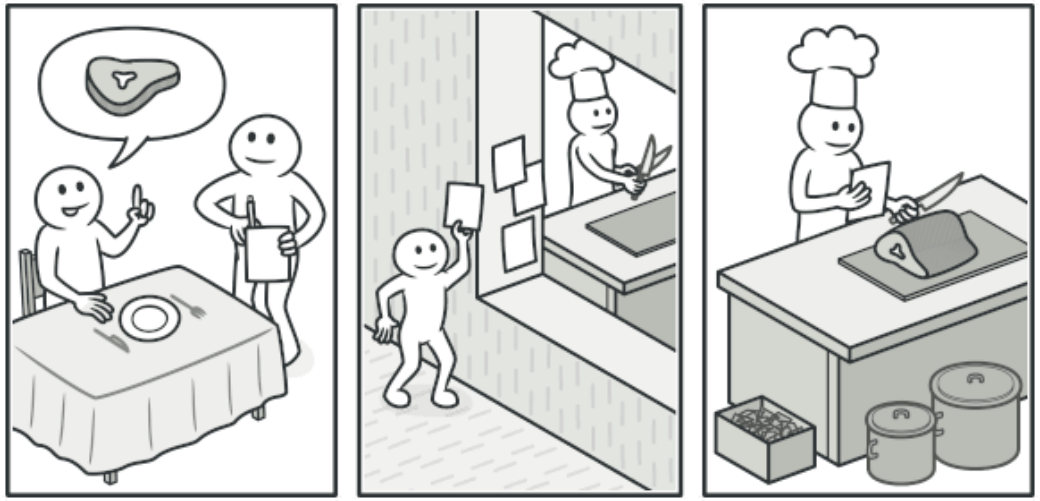


Все эти кнопки, хоть и выглядят схоже, но делают разные вещи.
Тогда вы создаете множество подклассов кнопок.

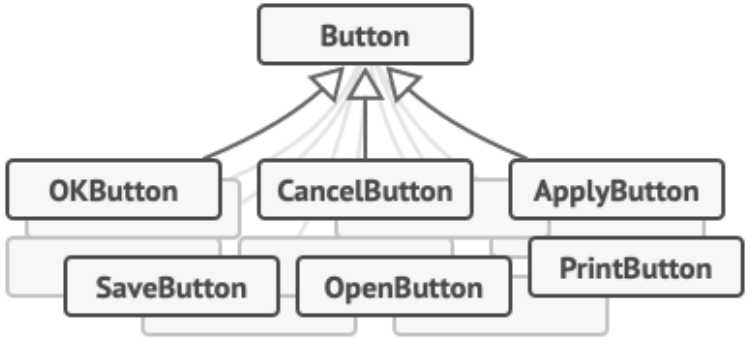


КОМАНДА

Превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций.



В этом примере посетитель является отправителем команды, официант с блокнотом — командой, а повар — получателем.



И ваш код кнопок начинает зависеть от классов бизнес-логики!

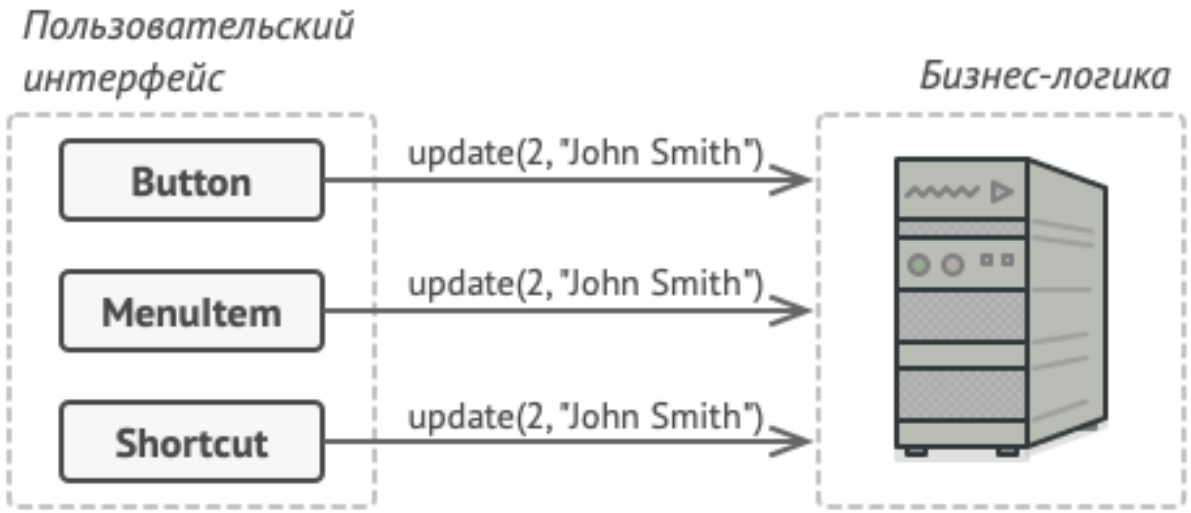
Но самое обидное ещё впереди...

Некоторые операции, например, «сохранить», можно вызывать из нескольких мест: нажав кнопку или просто нажав клавиши Ctrl+S

И теперь код из кнопки придётся продублировать.

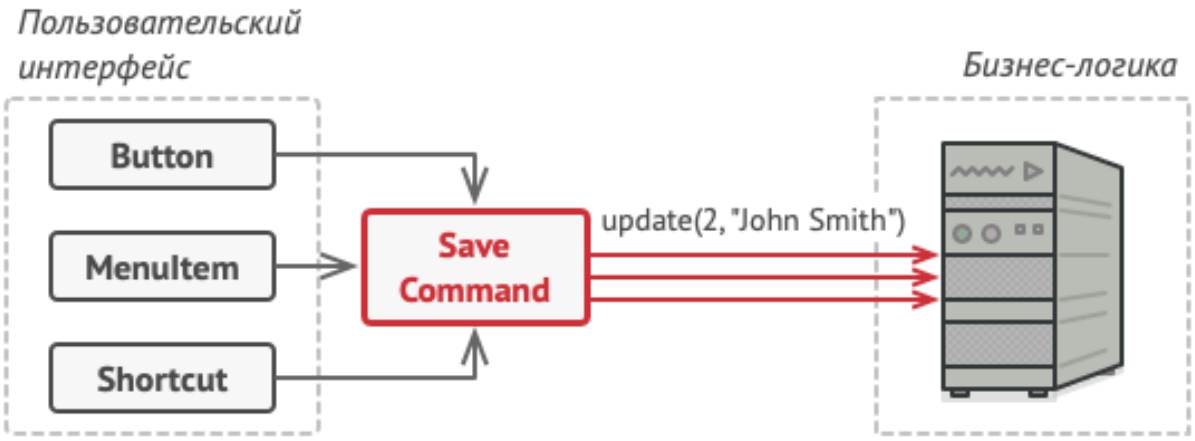
РЕШЕНИЕ

Прямой доступ из UI в бизнес-логику.



Паттерн Команда предлагает больше не отправлять такие вызовы напрямую.

Каждый вызов следует завернуть в собственный класс с единственным методом, который и будет осуществлять вызов.



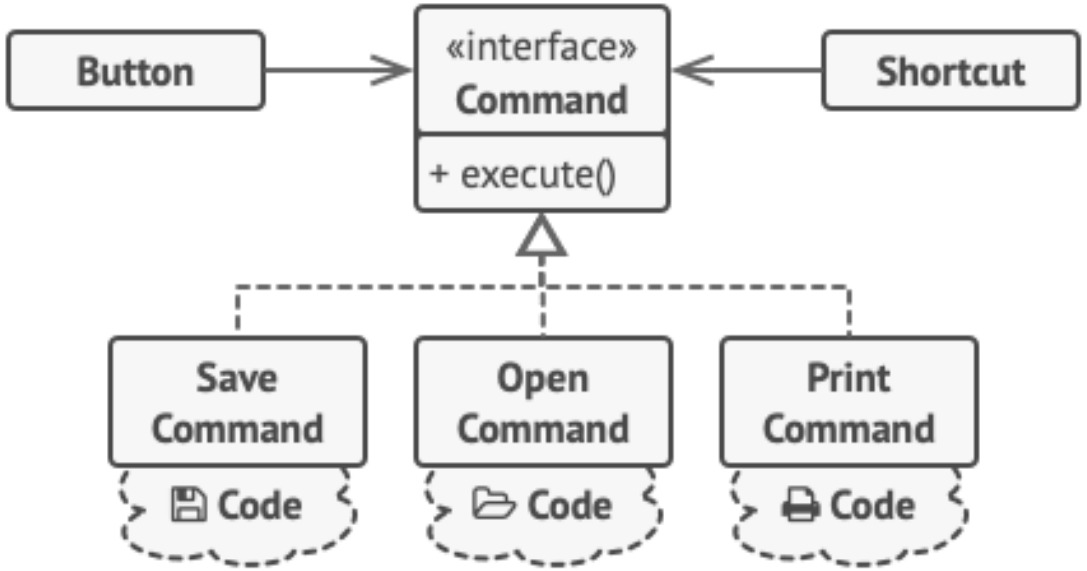
Доступ из UI в бизнес-логику через команду.

КЛАССЫ UI ДЕЛЕГИРУЮТ РАБОТУ КОМАНДАМ

Классы команд можно объединить под общим интерфейсом с единственным методом запуска.

Используя общий интерфейс команд, объекты кнопок будут ссылаться на объекты команд различных типов.

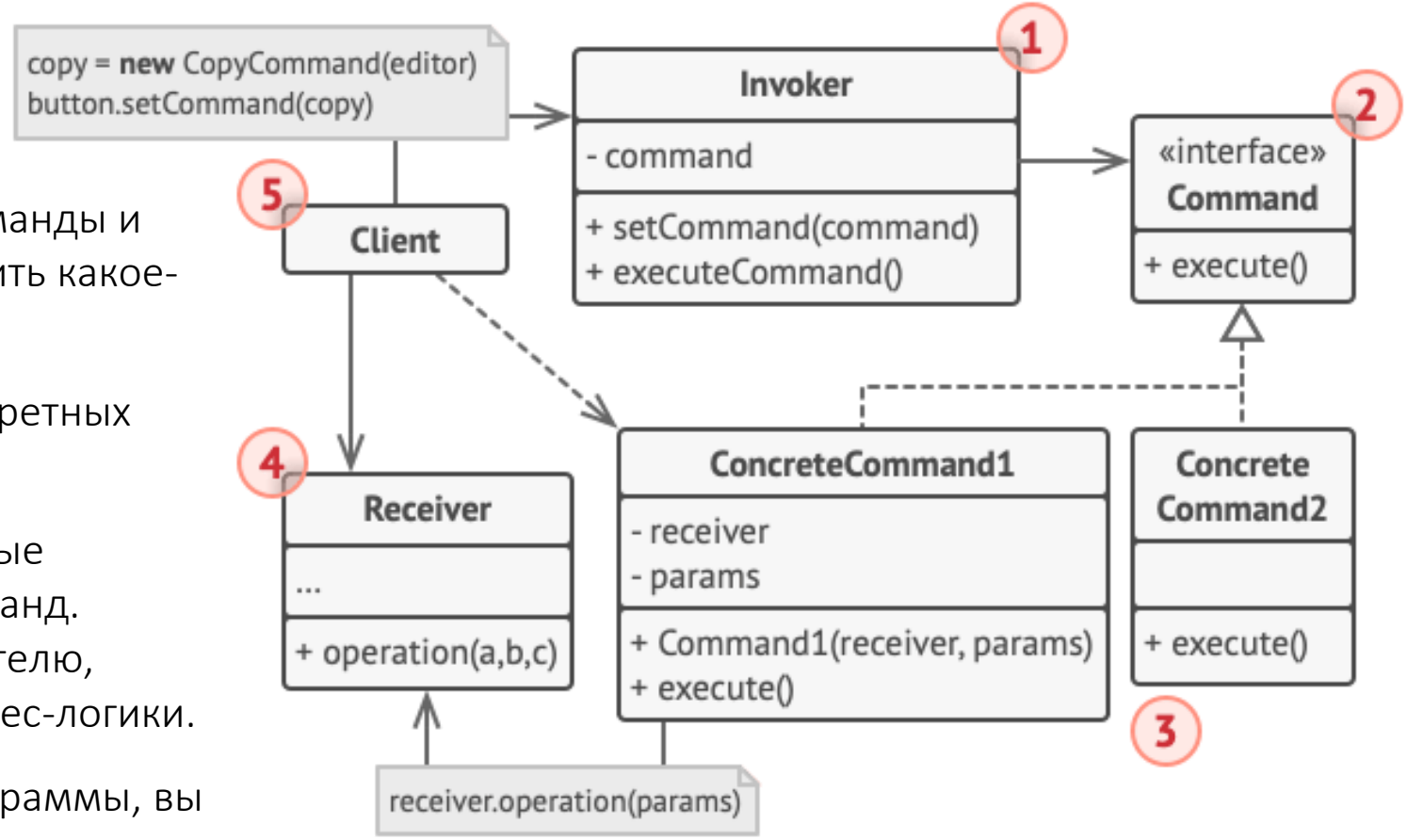
При нажатии кнопки будут делегировать работу связанным командам, а команды — перенаправлять вызовы тем или иным объектам бизнес-логики.



Таким образом, команды станут гибкой прослойкой между пользовательским интерфейсом и бизнес-логикой.

СТРУКТУРА

1. **Отправитель** хранит ссылку на объект команды и обращается к нему, когда нужно выполнить какое-то действие..
2. **Команда** описывает общий для всех конкретных команд интерфейс.
3. **Конкретные команды** реализуют различные запросы, следуя общему интерфейсу команд. Обычно команда передаёт вызов получателю, которым является один из объектов бизнес-логики.
4. **Получатель** содержит бизнес-логику программы, вы можете избавиться от получателей, «слив» их код в классы команд.
5. **Клиент** создаёт объекты конкретных команд, передавая в них ссылки на объекты получателей. После этого клиент связывает объекты отправителей с созданными командами.



```
abstract class Command
{
    public abstract void Execute();
    public abstract void Undo();
}
// конкретная команда
class ConcreteCommand : Command
{
    Receiver receiver;
    public ConcreteCommand(Receiver r)
    {
        receiver = r;
    }
    public override void Execute()
    {
        receiver.Operation();
    }

    public override void Undo()
    { }
}
```

```
abstract class Command
{
    public abstract void Execute();
    public abstract void Undo();
}
// конкретная команда
class ConcreteCommand : Command
{
    Receiver receiver;
    public ConcreteCommand(Receiver r)
    {
        receiver = r;
    }
    public override void Execute()
    {
        receiver.Operation();
    }

    public override void Undo()
    { }
}
```

```
// получатель команды
class Receiver
{
    public void Operation()
    { }
}
// инициатор команды
class Invoker
{
    Command command;
    public void SetCommand(Command c)
    {
        command = c;
    }
    public void Run()
    {
        command.Execute();
    }
    public void Cancel()
    {
        command.Undo();
    }
}
```

```
abstract class Command
{
    public abstract void Execute();
    public abstract void Undo();
}
// конкретная команда
class ConcreteCommand : Command
{
    Receiver receiver;
    public ConcreteCommand(Receiver r)
    {
        receiver = r;
    }
    public override void Execute()
    {
        receiver.Operation();
    }

    public override void Undo()
    { }
}
```

```
// получатель команды
class Receiver
{
    public void Operation()
    { }
}
// инициатор команды
class Invoker
{
    Command command;
    public void SetCommand(Command c)
    {
        command = c;
    }
    public void Run()
    {
        command.Execute();
    }
    public void Cancel()
    {
        command.Undo();
    }
}
```

```
Invoker invoker = new Invoker();
Receiver receiver = new Receiver();
ConcreteCommand command = new ConcreteCommand(receiver);
invoker.SetCommand(command);
invoker.Run();
```

РАССМОТРИМ НА ПРИМЕРЕ

Нередко в роли инициатора команд выступают панели управления или кнопки интерфейса. Рассмотрим самую простую ситуацию - надо программно организовать включение и выключение прибора, например, телевизора.

```
// Receiver - Получатель
class TV
{
    public void On()
    {
        Console.WriteLine("Телевизор включен!");
    }

    public void Off()
    {
        Console.WriteLine("Телевизор выключен...");
    }
}

// Invoker - инициатор
class Pult
{
    ICommand command;

    public Pult() { }

    public void SetCommand(ICommand com)
    {
        command = com;
    }

    public void PressButton()
    {
        command.Execute();
    }
    public void PressUndo()
    {
        command.Undo();
    }
}
```

```
interface ICommand
{
    void Execute();
    void Undo();
}
class TVOnCommand : ICommand
{
    TV tv;
    public TVOnCommand(TV tvSet)
    {
        tv = tvSet;
    }
    public void Execute()
    {
        tv.On();
    }
    public void Undo()
    {
        tv.Off();
    }
}
```

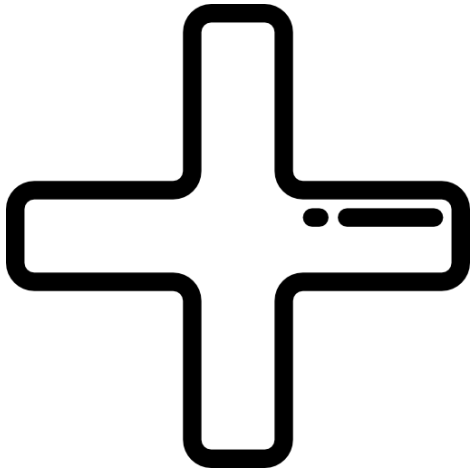
```
// Receiver - Получатель
class TV
{
    public void On()
    {
        Console.WriteLine("Телевизор включен!");
    }

    public void Off()
    {
        Console.WriteLine("Телевизор выключен...");
    }
}

// Invoker - инициатор
class Pult
{
    ICommand command;

    public Pult() { }

    public void SetCommand(ICommand com)
    {
        command = com;
    }

    public void PressButton()
    {
        command.Execute();
    }
    public void PressUndo()
    {
        command.Undo();
    }
}
```

```
interface ICommand
{
    void Execute();
    void Undo();
}
class TVOnCommand : ICommand
{
    TV tv;
    public TVOnCommand(TV tvSet)
    {
        tv = tvSet;
    }
    public void Execute()
    {
        tv.On();
    }
    public void Undo()
    {
        tv.Off();
    }
}

Pult pult = new Pult();
TV tv = new TV();
pult.SetCommand(new TVOnCommand(tv))
pult.PressButton();
pult.PressUndo();
```

ПРИМЕНИМОСТЬ

- Когда вы хотите параметризовать объекты выполняемым действием.
- Когда вы хотите ставить операции в очередь, выполнять их по расписанию или передавать по сети.
- Когда вам нужна операция отмены.

ПРЕИМУЩЕСТВА И НЕДОСТАТКИ



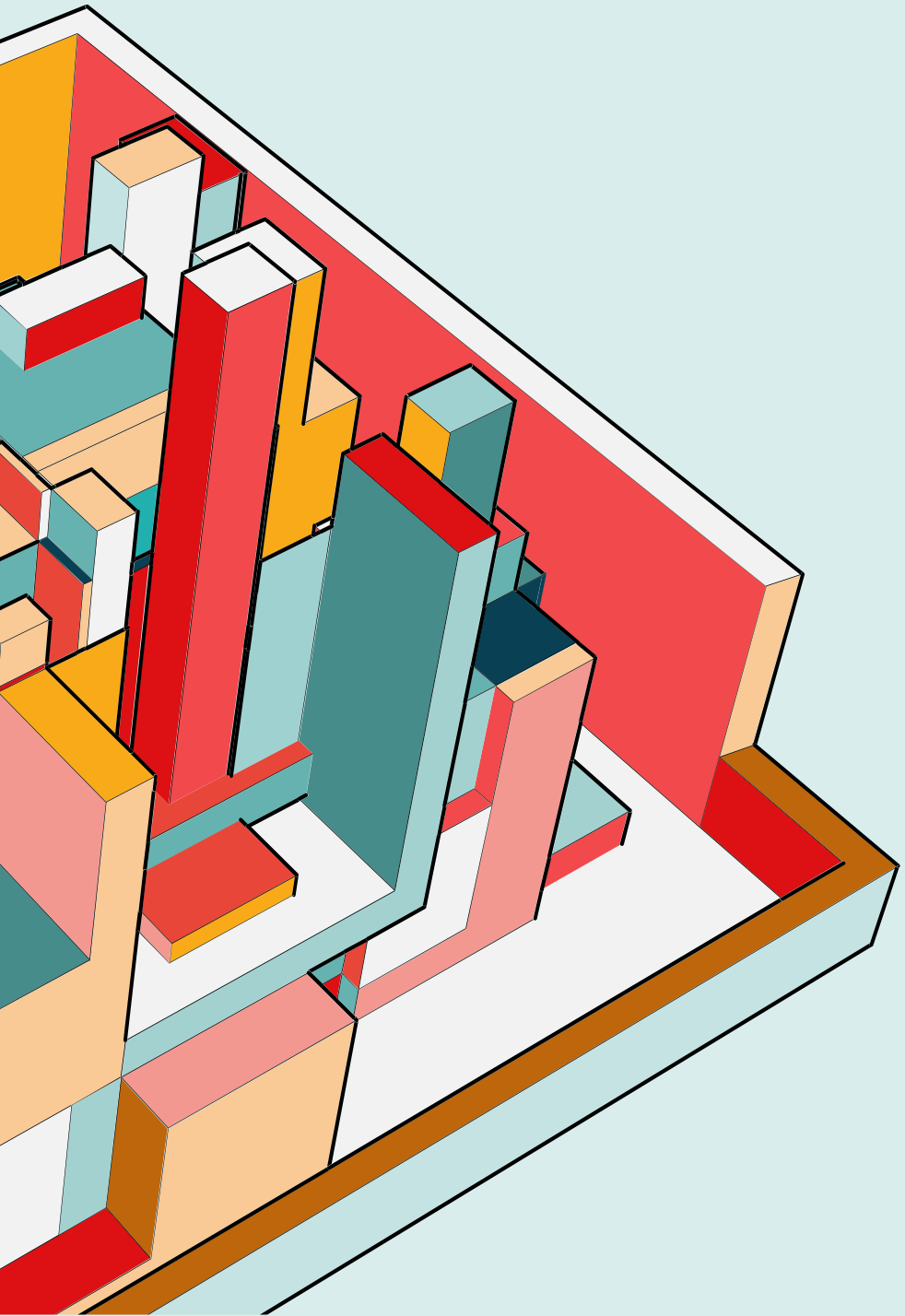
Убирает прямую зависимость между объектами, вызывающими операции, и объектами, которые их непосредственно выполняют.

Позволяет реализовать отложенный запуск операций.

Позволяет собирать сложные команды из простых.

Реализует принцип открытости/закрытости.

Усложняет код программы из-за введения множества дополнительных классов.

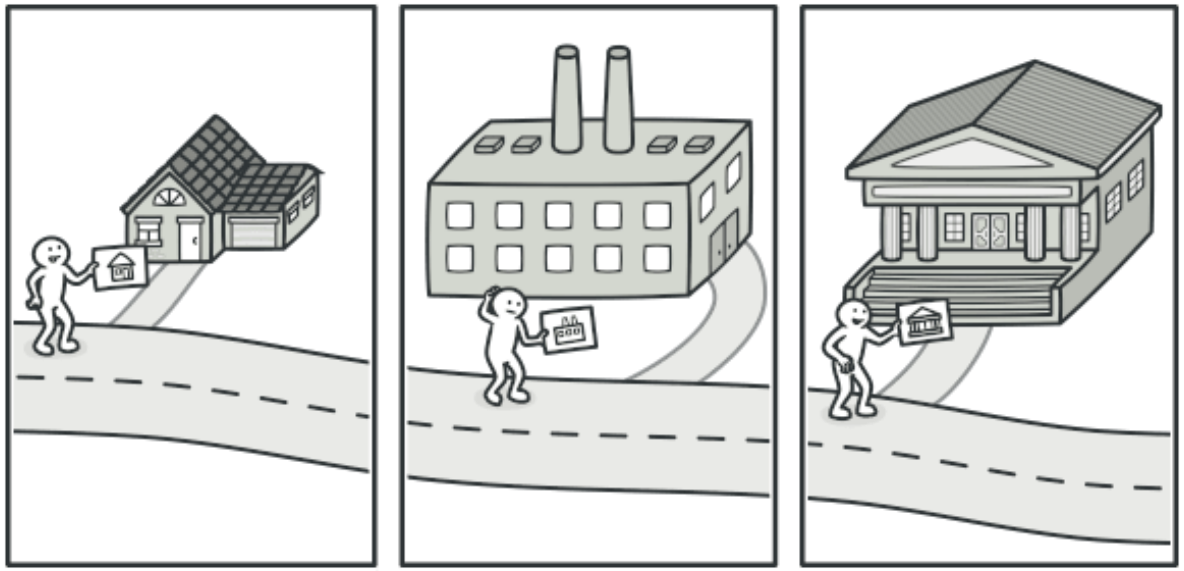


ПОВЕДЕНЧЕСКИЕ ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

Visitor

ПОСЕТИТЕЛЬ

Позволяет добавлять в программу новые операции, не изменяя классы объектов, над которыми эти операции могут выполняться.



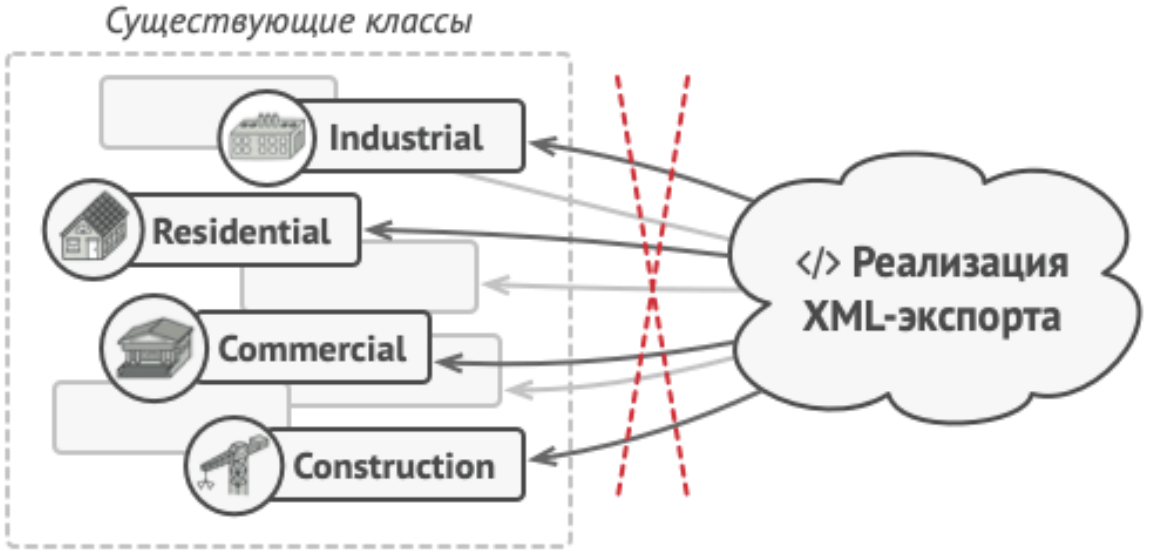
У страхового агента приготовлены полисы для разных видов организаций.

Придя в **дом**, он предлагает оформить **медицинскую страховку**.
Придя в **банк**, он предлагает **страховку от грабежа**.
Придя на **фабрику**, он предлагает **страховку от пожара**.

Ваша команда разрабатывает приложение, работающее с геоданными в виде графа.

Ваша задача — сделать экспорт этого графа в XML.

Добавить метод экспорта в каждый тип узла?

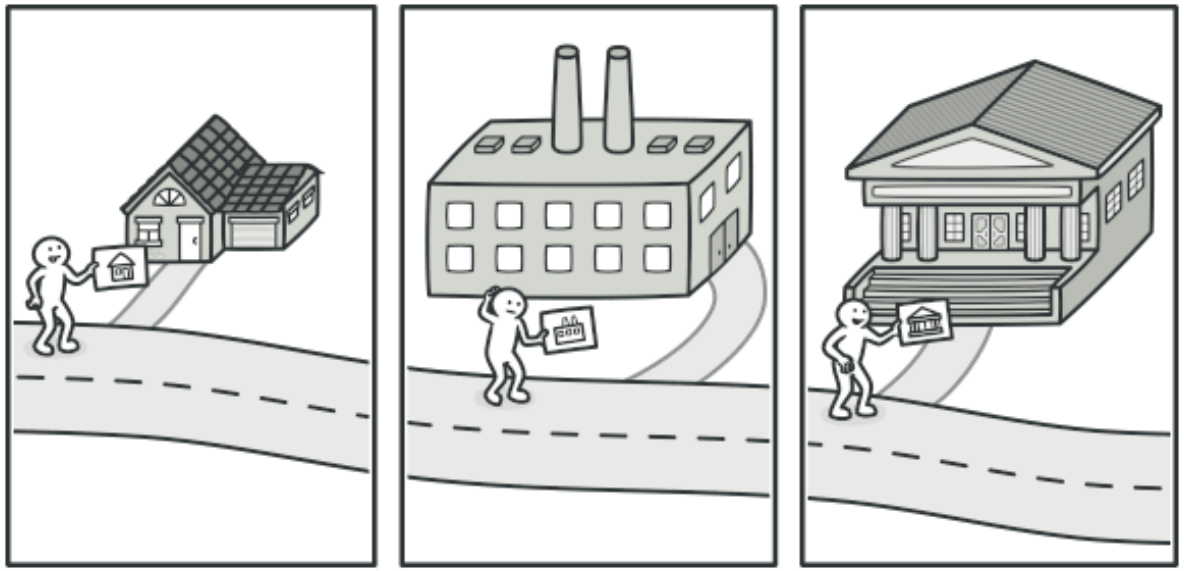


Код XML-**экспорта придётся добавить во все классы узлов, а это слишком накладно.**

Если на следующей неделе вам бы понадобился экспорт в какой-то другой формат?

ПОСЕТИТЕЛЬ

Позволяет добавлять в программу новые операции, не изменяя классы объектов, над которыми эти операции могут выполняться.



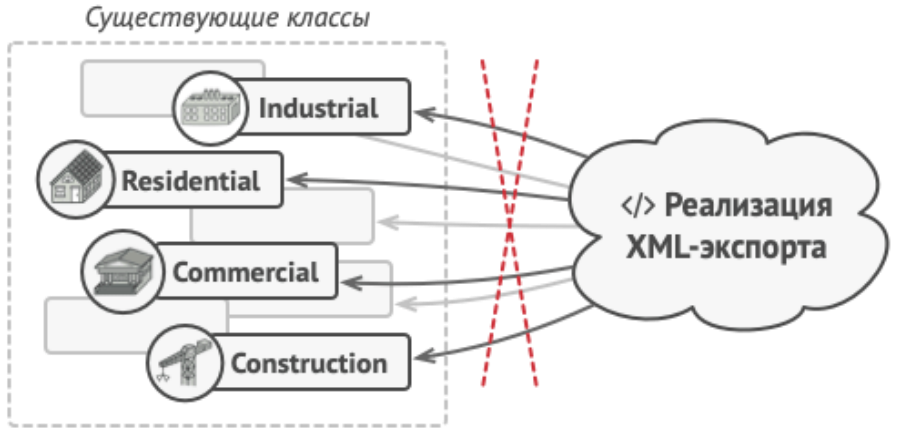
У страхового агента приготовлены полисы для разных видов организаций.

Придя в **дом**, он предлагает оформить **медицинскую страховку**.
Придя в **банк**, он предлагает **страховку от грабежа**.
Придя на **фабрику**, он предлагает **страховку от пожара**.

Ваша команда разрабатывает приложение, работающее с геоданными в виде графа.

Ваша задача — сделать экспорт этого графа в XML.

Добавить метод экспорта в каждый тип узла?



Код XML-экспорта придётся добавить во все классы узлов, а это слишком накладно.

Разместить новое поведение в отдельном классе, вместо того чтобы множить его сразу в нескольких классах.

РЕШЕНИЕ ☹ ☺

```
class ExportVisitor implements Visitor is
    method doForCity(City c) { ... }
    method doForIndustry(Industry f) { ... }
    method doForSightSeeing(SightSeeing ss) { ... }
    // ...
```

```
foreach (Node node in graph)
    if (node instanceof City)
        exportVisitor.doForCity((City) node)
    if (node instanceof Industry)
        exportVisitor.doForIndustry((Industry) node)
    // ...
```

Вместо того, чтобы самим искать нужный метод, мы можем поручить это объектам, которые передаём в параметрах посетителю.

```
// Client code
foreach (Node node in graph)
    node.accept(exportVisitor)

// City
class City is
    method accept(Visitor v) is
        v.doForCity(this)
    // ...

// Industry
class Industry is
    method accept(Visitor v) is
        v.doForIndustry(this)
    // ...
```

Как видите, изменить классы узлов всё-таки придётся – это минус, но есть нюанс... Если придётся добавить в программу новое поведение, вы просто создадите новый класс посетителей.

СТРУКТУРА

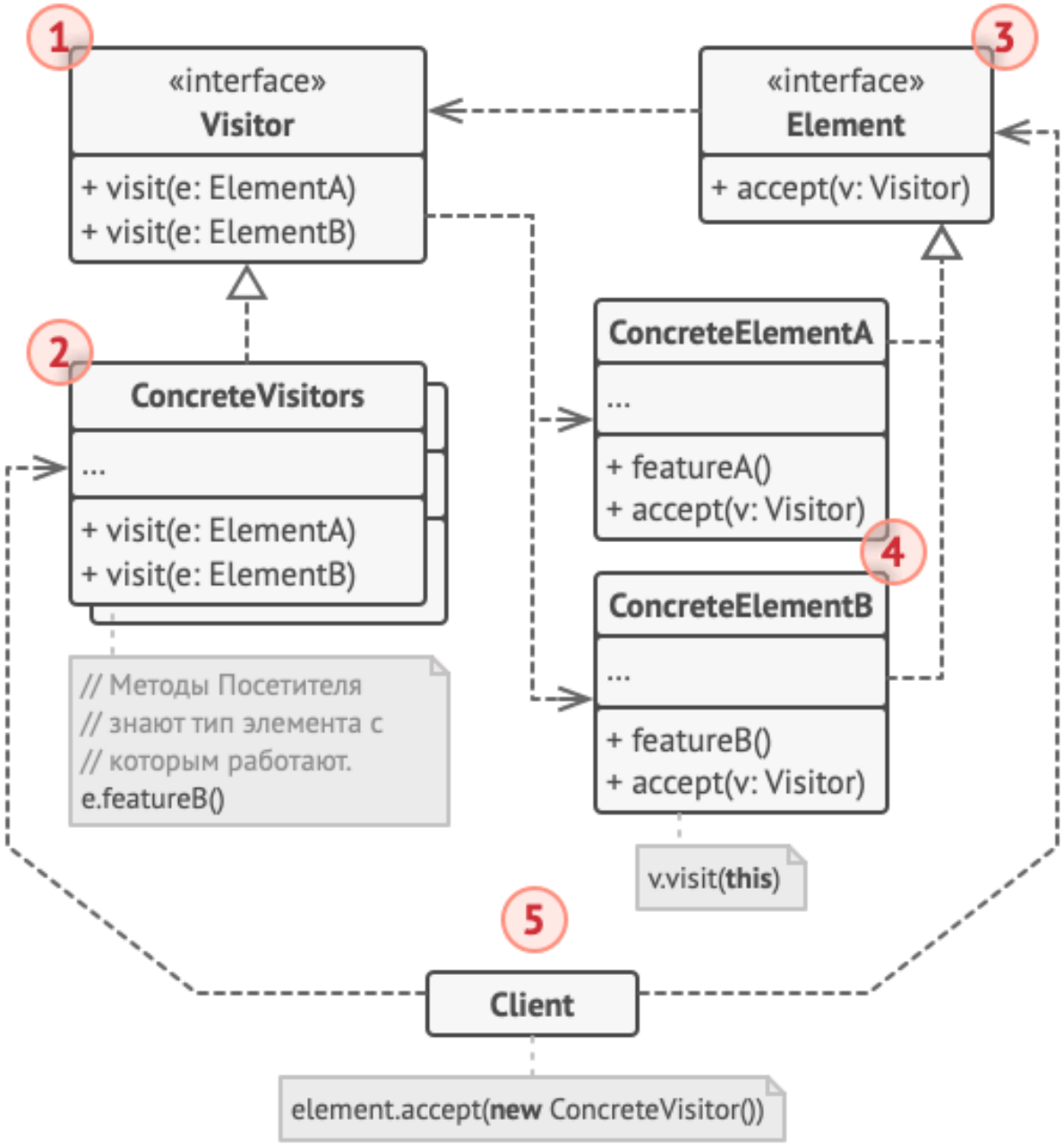
1.Посетитель описывает общий интерфейс для всех типов посетителей. Он объявляет набор методов, отличающихся типом входящего параметра.

2.Конкретные посетители реализуют какое-то особенное поведение для всех типов элементов, которые можно подать через методы интерфейса посетителя.

3.Элемент описывает метод *принятия* посетителя. Этот метод должен иметь единственный параметр, объявленный с типом интерфейса посетителя.

4.Конкретные элементы реализуют методы *принятия* посетителя. Цель этого метода — вызвать тот метод посещения, который соответствует типу этого элемента. Так посетитель узнает, с каким именно элементом он работает.

5.Клиентом зачастую выступает коллекция или сложный составной объект



```
abstract class Visitor
{
    public abstract void VisitElementA(ElementA elemA);
    public abstract void VisitElementB(ElementB elemB);
}

class ConcreteVisitor1 : Visitor
{
    public override void VisitElementA(ElementA elementA)
    {
        elementA.OperationA();
    }
    public override void VisitElementB(ElementB elementB)
    {
        elementB.OperationB();
    }
}
class ConcreteVisitor2 ...
```

```
class ObjectStructure
{
    List<Element> elements = new List<Element>()
    public void Add(Element element)
    {
        elements.Add(element);
    }
    public void Remove(Element element)
    {
        elements.Remove(element);
    }
    public void Accept(Visitor visitor)
    {
        foreach (Element element in elements)
            element.Accept(visitor);
    }
}

abstract class Element
{
    public abstract void Accept(Visitor visitor)
    public string SomeState { get; set; }
}

class ElementA : Element
{
    public override void Accept(Visitor visitor)
    {
        visitor.VisitElementA(this);
    }
    public void OperationA()
    { }
}

class ElementB ...
```

```
abstract class Visitor
{
    public abstract void VisitElementA(ElementA elemA);
    public abstract void VisitElementB(ElementB elemB);
}

class ConcreteVisitor1 : Visitor
{
    public override void VisitElementA(ElementA elementA)
    {
        elementA.OperationA();
    }
    public override void VisitElementB(ElementB elementB)
    {
        elementB.OperationB();
    }
}
class ConcreteVisitor2 ...
```

```
var structure = new ObjectStructure();
structure.Add(new ElementA());
structure.Add(new ElementB());
structure.Accept(new ConcreteVisitor1());
structure.Accept(new ConcreteVisitor2());
```

```
class ObjectStructure
{
    List<Element> elements = new List<Element>()
    public void Add(Element element)
    {
        elements.Add(element);
    }
    public void Remove(Element element)
    {
        elements.Remove(element);
    }
    public void Accept(Visitor visitor)
    {
        foreach (Element element in elements)
            element.Accept(visitor);
    }
}

abstract class Element
{
    public abstract void Accept(Visitor visitor)
    public string SomeState { get; set; }
}

class ElementA : Element
{
    public override void Accept(Visitor visitor)
    {
        visitor.VisitElementA(this);
    }
    public void OperationA()
    { }
}

class ElementB ...
```

ПРИМЕНИМОСТЬ

- Когда вам нужно выполнить какую-то операцию над всеми элементами сложной структуры объектов, например, деревом.
- Когда над объектами сложной структуры объектов надо выполнять некоторые не связанные между собой операции, но вы не хотите «засорять» классы такими операциями.
- Когда новое поведение имеет смысл только для некоторых классов из существующей иерархии.

РАССМОТРИМ НА ПРИМЕРЕ

Как известно, нередко для разных категорий вкладчиков банки имеют свои правила: оформления вкладов, выдача кредитов, начисления процентов и т.д. Соответственно классы, описывающие данные объекты, тоже будут разными.

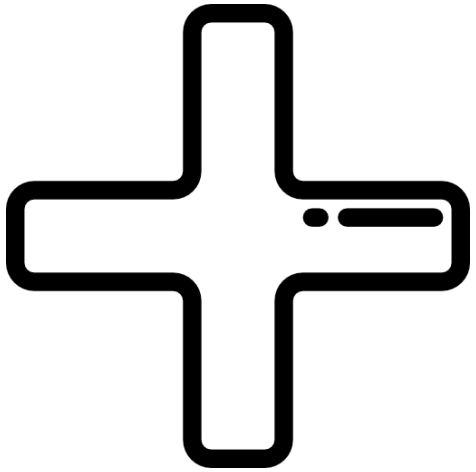
Правила обслуживания четко описают весь набор категорий клиентов. Например, есть физические лица, есть юридические, отдельные правила для индивидуальных или частных предпринимателей и т.д. Поэтому структура классов, представляющая клиентов будет относительно фиксированной, то есть не склонной к изменениям.

И допустим, в какой-то момент мы решили добавить в классы клиентов функционал сериализации в html. **Это легко…**

Но допустим, мы решим добавить потом еще сериализацию в формат xml. **Терпимо…**

Но еще через некоторое время мы захотим добавить сериализацию в формат json. **Достало…**

ПРЕИМУЩЕСТВА И НЕДОСТАТКИ



Упрощает добавление операций, работающих со сложными структурами объектов.

Объединяет родственные операции в одном классе.

Посетитель может накапливать состояние при обходе структуры элементов.

Паттерн не оправдан, если иерархия элементов часто меняется.

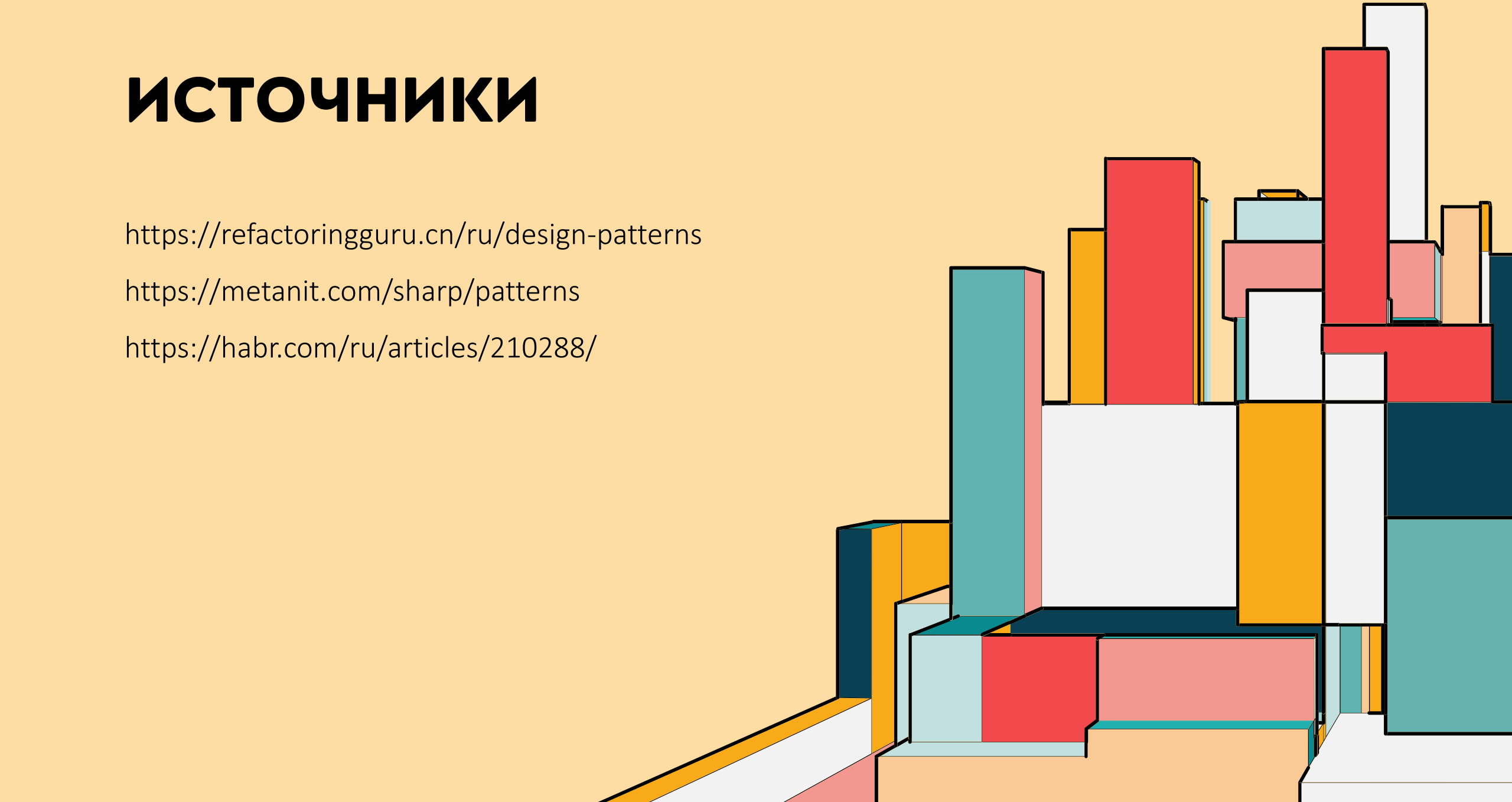
Может привести к нарушению инкапсуляции элементов.

ИСТОЧНИКИ

https://refactoringguru.cn/ru/design-patterns

https://metanit.com/sharp/patterns

https://habr.com/ru/articles/210288/



СПАСИБО!

Виденин Сергей

@videninserg