Максимов Ю.Е. 1335 группа

- ✓ 20. Абстрактные классы. Сравнение абстрактных классов с интерфейсами

Абстрактный класс

Кроме обычных классов в C# есть **абстрактные классы**. Зачем они нужны? Классы обычно представляют некий план определенного рода объектов или сущностей. Например, мы можем определить класс Car для преставления машин или класс Person для представления людей, вложив в эти классы соответствующие свойства, поля, методы, которые будут описывать данные объекты. Однако некоторые сущности, которые мы хотим выразить с помощью языка программирования, могут не иметь конкретного воплощения. Например, в реальности не существует геометрической фигуры как таковой. Есть круг, прямоугольник, квадрат, но просто фигуры нет. Однако же и круг, и прямоугольник имеют что-то общее и являются фигурами. И для описания подобных сущностей, которые не имеют конкретного воплощения, предназначены абстрактные классы.

Абстрактный класс похож на обычный класс. Он также может иметь переменные, методы, конструкторы, свойства. Единственное, что при определении абстрактных классов используется ключевое слово **abstract**. Например, определим абстрактный класс, который представляет некое транспортное средство:

```
abstract class Transport
{
    public void Move()
    {
        Console.WriteLine("Транспортно средство движется");
```

```
    }
}
```

Транспортное средство представляет некоторую абстракцию, которая не имеет конкретного воплощения. То есть есть легковые и грузовые машины, самолеты, морские судна, кто-то на космическом корабле любит покататься, но как такового транспортного средства нет. Тем не менее все транспортные средства имеют нечто общее - они могут перемещаться. И для этого в классе определен метод Move, который эмулирует перемещение.

Но главное отличие абстрактных классов от обычных состоит в том, что мы **НЕ можем** использовать конструктор абстрактного класса для создания экземпляра класса. Например, следующим образом:

```
Transport tesla = new Transport();
```

Тем не менее абстрактные классы полезны для описания некоторого общего функционала, который могут наследовать и использовать производные классы:

```
Transport car = new Car();
Transport ship = new Ship();
Transport aircraft = new Aircraft();

car.Move();
ship.Move();
aircraft.Move();
abstract class Transport
{
    public void Move()
    {
        Console.WriteLine("Транспортное средство движется");
    }
}
// класс корабля
```

```
class Ship : Transport { }
// класс самолета
class Aircraft : Transport { }
// класс машины
class Car : Transport { }
```

В данном случае от класса Transport наследуются три класса, которые представляют различные типы транспортных средств. Тем не менее они имеют общую черту - они могут перемещаться с помощью метода `Move()`.

Интерфейс

Для определения интерфейса используется ключевое слово **interface**. Как правило, названия интерфейсов в C# начинаются с заглавной буквы **I**, например, IComparable, IEnumerable (так называемая венгерская нотация), однако это не обязательное требование, а больше стиль программирования.

Что может определять интерфейс? В целом интерфейсы могут определять следующие сущности:

- Методы
- Свойства
- Индексаторы
- События
- Статические поля и константы (начиная с версии C# 8.0)

Однако интерфейсы не могут определять нестатические переменные. Например, простейший интерфейс, который определяет все эти компоненты:

```
interface IMovable
{
    // константа
    const int minSpeed = 0;     // минимальная скорость
    // статическая переменная
```

```
        static int maxSpeed = 60;    // максимальная скорость
        // метод
        void Move();                 // движение
        // свойство
        string Name { get; set; }    // название

        delegate void MoveHandler(string message);  // определение делегата для события
        // событие
        event MoveHandler MoveEvent;     // событие движения
    }
```

В данном случае определен интерфейс IMovable, который представляет некоторый движущийся объект. Данный интерфейс содержит различные компоненты, которые описывают возможности движущегося объекта. То есть интерфейс описывает некоторый функционал, который должен быть у движущегося объекта.

Методы и свойства интерфейса могут не иметь реализации, в этом они сближаются с абстрактными методами и свойствами абстрактных классов. В данном случае интерфейс определяет метод Move, который будет представлять некоторое передвижение. Он не имеет реализации, не принимает никаких параметров и ничего не возвращает.

То же самое в данном случае касается свойства Name. На первый взгляд оно похоже на автоматическое свойство. Но в реальности это определение свойства в интерфейсе, которое не имеет реализации, а не автосвойство.

Сравнение абстрактного класса с интерфейсов

Один из принципов проектирования гласит, что при создании системы классов надо программировать на уровне интерфейсов, а не их конкретных реализаций. Под интерфейсами в данном случае понимаются не только типы C#, определенные с помощью ключевого слова `interface`, а определение

функционала без его конкретной реализации. То есть под данное определение попадают как собственно интерфейсы, так и абстрактные классы, которые могут иметь абстрактные методы без конкретной реализации.

В этом плане у абстрактных классов и интерфейсов много общего. Нередко при проектировании программ в паттернах мы можем заменять абстрактные классы на интерфейсы и наоборот. Однако все же они имеют некоторые отличия.

Когда следует использовать абстрактные классы:

- Если надо определить общий функционал для родственных объектов
- Если мы проектируем довольно большую функциональную единицу, которая содержит много базового функционала
- Если нужно, чтобы все производные классы на всех уровнях наследования имели некоторую общую реализацию. При использовании абстрактных классов, если мы захотим изменить базовый функционал во всех наследниках, то достаточно поменять его в абстрактном базовом классе.

  Если же нам вдруг надо будет поменять название или параметры метода интерфейса, то придется вносить изменения и также во всех классы, которые данный интерфейс реализуют.

Когда следует использовать интерфейсы:

- Если нам надо определить функционал для группы разрозненных объектов, которые могут быть никак не связаны между собой.
- Если мы проектируем небольшой функциональный тип

Ключевыми здесь являются первые пункты, которые можно свести к следующему принципу: если классы относятся к единой системе классификации, то выбирается абстрактный класс. Иначе выбирается интерфейс. Посмотрим на примере.

Допустим, у нас есть система транспортных средств: легковой автомобиль, автобус, трамвай, поезд и т.д. Поскольку данные объекты являются родственными, мы можем выделить у них общие признаки, то в данном случае можно использовать абстрактные классы:

```
public abstract class Vehicle
{
    public abstract void Move();
}

public class Car : Vehicle
{
    public override void Move()
    {
        Console.WriteLine("Машина едет");
    }
}

public class Bus : Vehicle
{
    public override void Move()
    {
        Console.WriteLine("Автобус едет");
    }
}

public class Tram : Vehicle
{
    public override void Move()
    {
        Console.WriteLine("Трамвай едет");
    }
```

```
}
```

Абстрактный класс Vehicle определяет абстрактный метод перемещения Move(), а классы-наследники его реализуют.

Но, предположим, что наша система транспорта не ограничивается вышеперечисленными транспортными средствами. Например, мы можем добавить самолеты, лодки. Возможно, также мы добавим лошадь - животное, которое может также выполнять роль транспортного средства. Также можно добавить дирижабль. Вобщем получается довольно широкий круг объектов, которые связаны только тем, что являются транспортным средством и должны реализовать некоторый метод Move(), выполняющий перемещение.

Так как объекты малосвязанные между собой, то для определения общего для всех них функционала лучше определить интерфейс. Тем более некоторые из этих объектов могут существовать в рамках параллельных систем классификаций. Например, лошадь может быть классом в структуре системы классов животного мира.

Возможная реализация интерфейса могла бы выглядеть следующим образом:

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

public abstract class Vehicle : IMovable
{
    public abstract void Move();
}

public class Car : Vehicle
{
```

```
    public override void Move() => Console.WriteLine("Машина едет");
}

public class Bus : Vehicle
{
    public override void Move() => Console.WriteLine("Автобус едет");
}

public class Hourse : IMovable
{
    public void Move() => Console.WriteLine("Лошадь скачет");
}

public class Aircraft : IMovable
{
    public void Move() => Console.WriteLine("Самолет летит");
}
```

Теперь метод `Move()` определяется в интерфейсе IMovable, а конкретные классы его реализуют.

Говоря об использовании абстрактных классов и интерфейсов можно привести еще такую аналогию, как состояние и действие. Как правило, абстрактные классы фокусируются на общем состоянии классов-наследников. В то время как интерфейсы строятся вокруг какого-либо общего действия.

Например, солнце, костер, батарея отопления и электрический нагреватель выполняют функцию нагревания или излучения тепла. По большому счету выделение тепла - это единственный общий между ними признак. Можно ли для них создать общий абстрактный класс? Можно, но это не будет оптимальным решением, тем более у нас могут быть какие-то родственные сущности, которые мы, возможно, тоже захотим использовать. Поэтому для каждой вышеперечисленной сущности мы можем определить свою систему

классификации. Например, в одной системе классов, которые наследуются от общего астрактного класса, были бы звезды, в том числе и солнце, планеты, астероиды и так далее - то есть все те объекты, которые могут иметь какое-то общее с солнцем состояние. В рамках другой системы классов мы могли бы определить электрические приборы, в том числе электронагреатель. И так, для каждой разноплановой сущности можно было бы составить свою систему классов, исходяющую от определенного абстрактного класса. А для общего действия определить интерфейс, например, IHeatable, в котором бы был метод Heat, и этот интерфейс реализовать во всех необходимых классах.

Таким образом, если разноплановые классы обладают каким-то общим действием, то это действие лучше выносить в интерфейс. А для одноплановых классов, которые имеют общее состояние, лучше определять абстрактный класс.

✓ Паттерны проектирования приложений.

1. Порождающие паттерны проектирования
    a. Фабричный метод – это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.

- Проблема

Представьте, что вы создаёте программу управления грузовыми перевозками. Сперва вы рассчитываете перевозить товары только на автомобилях. Поэтому весь ваш код работает с объектами класса Truck (грузовик).

В какой-то момент ваша программа становится настолько известной, что морские перевозчики выстраиваются в очередь и просят добавить поддержку морской логистики в программу.

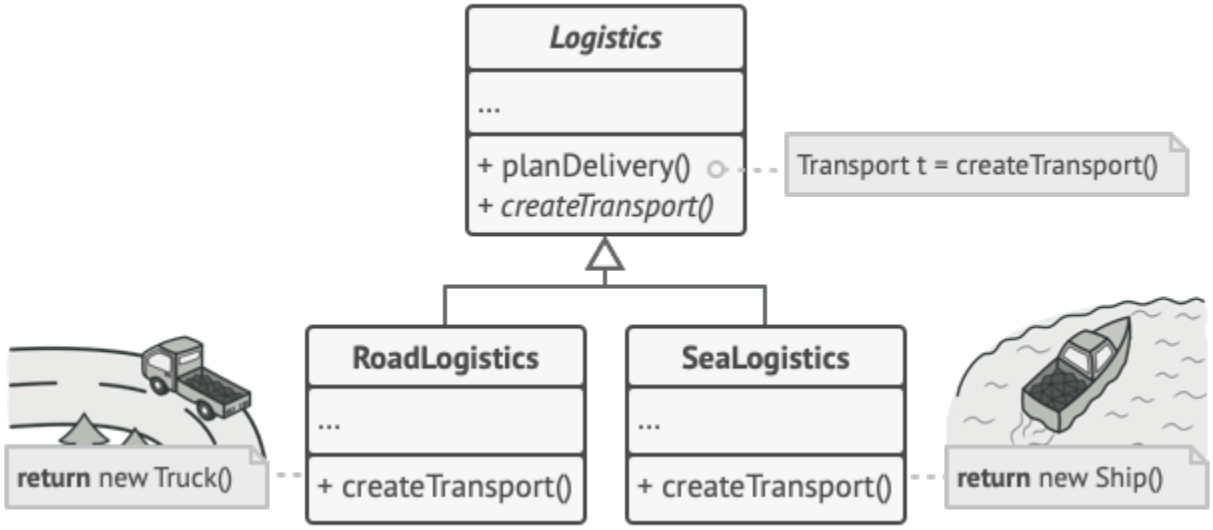
Добавить новый класс не так-то просто, если весь код уже завязан на конкретные классы.

Отличные новости, правда?! Но как насчёт кода? Большая часть существующего кода жёстко привязана к классу Truck. Чтобы добавить в программу классы морских судов, понадобится перелопатить всю программу. Более того, если вы потом решите добавить в программу ещё один вид транспорта, то всю эту работу придётся повторить.

В итоге вы получите ужасающий код, наполненный условными операторами, которые выполняют то или иное действие, в зависимости от класса транспорта.

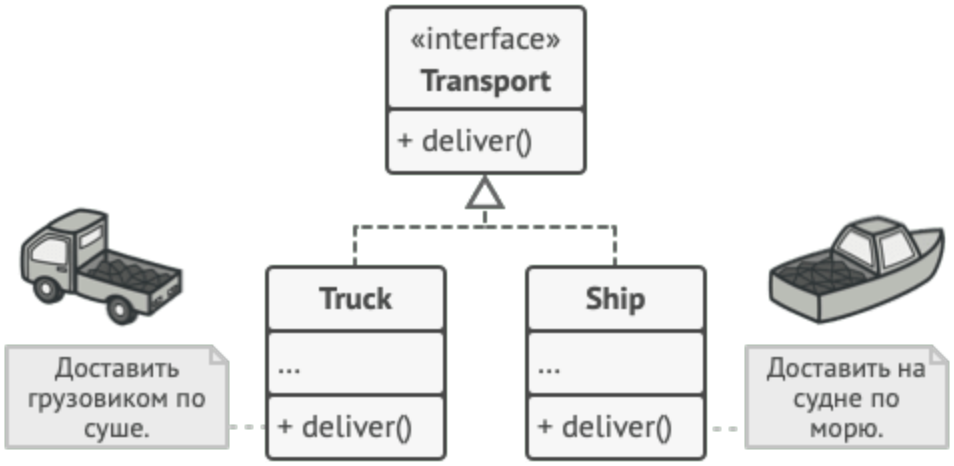
- Решение

Паттерн Фабричный метод предлагает создавать объекты не напрямую, используя оператор new, а через вызов специального фабричного метода. Не пугайтесь, объекты всё равно будут создаваться при помощи new, но делать это будет фабричный метод. Объекты, возвращаемые фабричным методом, часто называют продуктами.

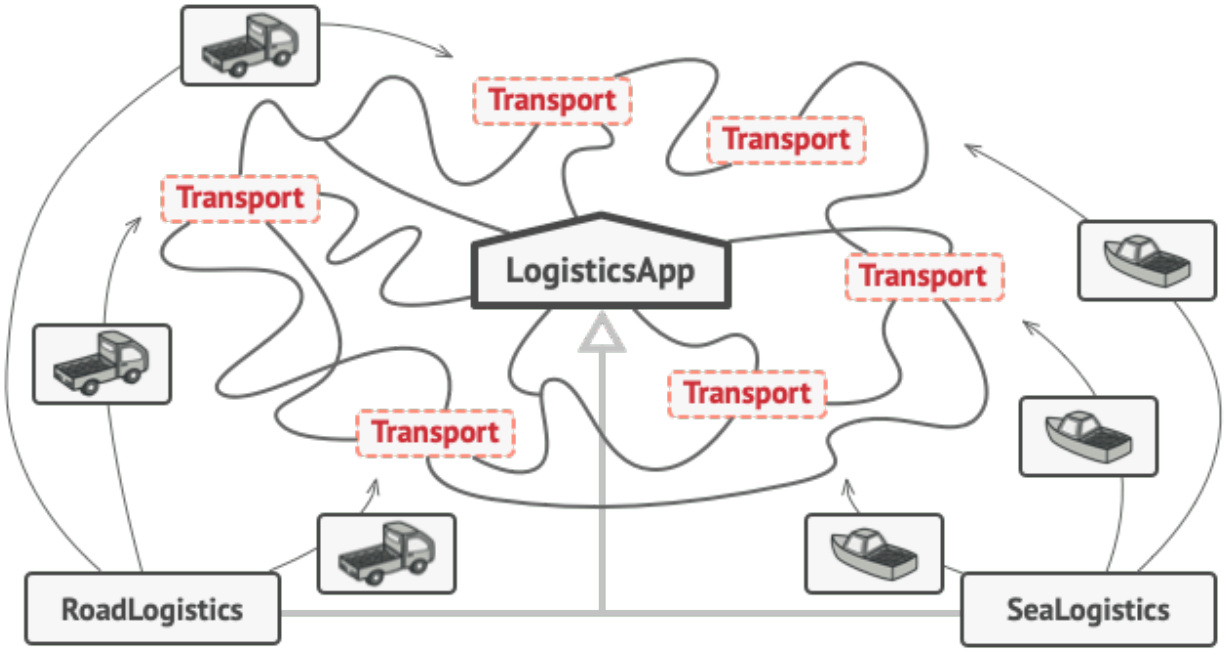


На первый взгляд, это может показаться бессмысленным: мы просто переместили вызов конструктора из одного конца программы в другой. Но теперь вы сможете переопределить в подклассе фабричный метод, чтобы изменить тип создаваемого продукта.

Однако здесь есть небольшое ограничение: подклассы могут возвращать различные типы объектов-продуктов, только если эти объекты имеют общий базовый класс или интерфейс. Кроме того, тип возвращаемого значения для фабричного метода в базовом классе должен быть объявлен как этот интерфейс.



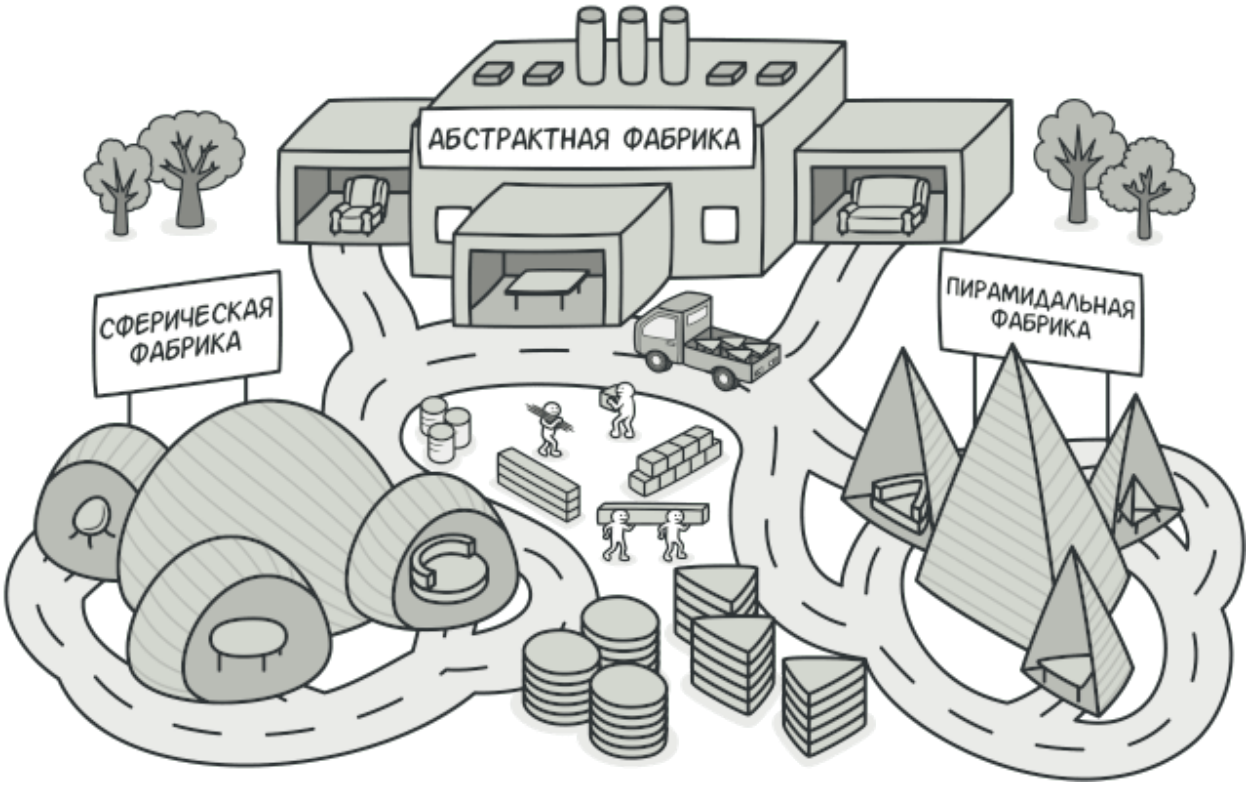
Например, классы Truck (грузовик) и Ship (судно) реализуют интерфейс Transport (транспорт) с методом deliver() (доставить). Каждый из этих классов реализует этот метод по-своему: грузовики везут грузы по земле, а суда – по морю. Фабричный метод в классе RoadLogistic (дорожная логистика) возвращает объекты-грузовики, а класс SeaLogistics (морская логистика) – объекты-суда.



Код, использующий фабричный метод (часто называемый клиентским кодом), не видит разницы между фактическими продуктами, возвращаемыми различными подклассами. Клиент рассматривает все продукты как объекты абстрактного класса Transport (транспорт). Клиент знает, что все объекты Transport должны иметь метод deliver(), но то, как именно он работает, для клиента не имеет значения.

b. Паттерн Абстрактная фабрика

Абстрактная фабрика – это порождающий паттерн проектирования, который позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.



- Проблема

Представьте, что вы пишете симулятор мебельного магазина. Ваш код содержит:

- ✓ Семейство зависимых продуктов. Скажем, Кресло + Диван + Столик.

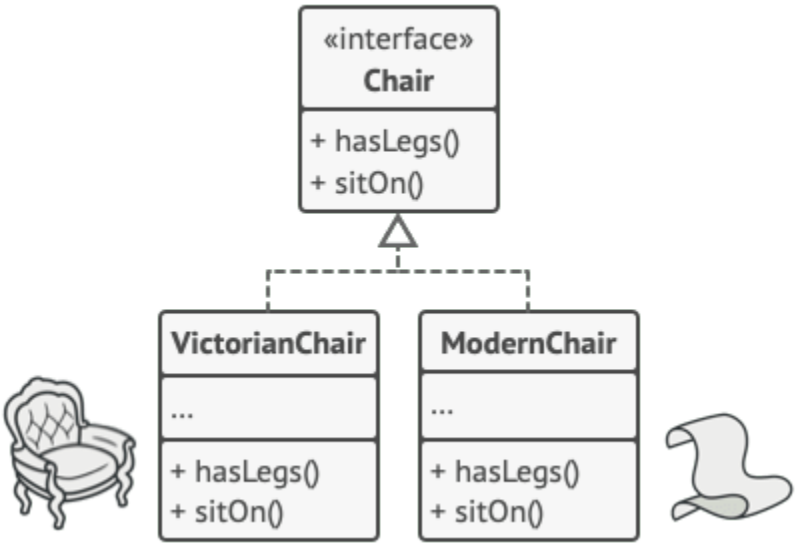
- ✓ Несколько вариаций этого семейства. Например, продукты Кресло, Диван и Столик представлены в трёх разных стилях: Ар-деко, Викторианском и Модерне.

Вам нужен такой способ создавать объекты продуктов, чтобы они сочетались с другими продуктами того же семейства. Это важно, так как клиенты расстраиваются, если получают несочетающуюся мебель.

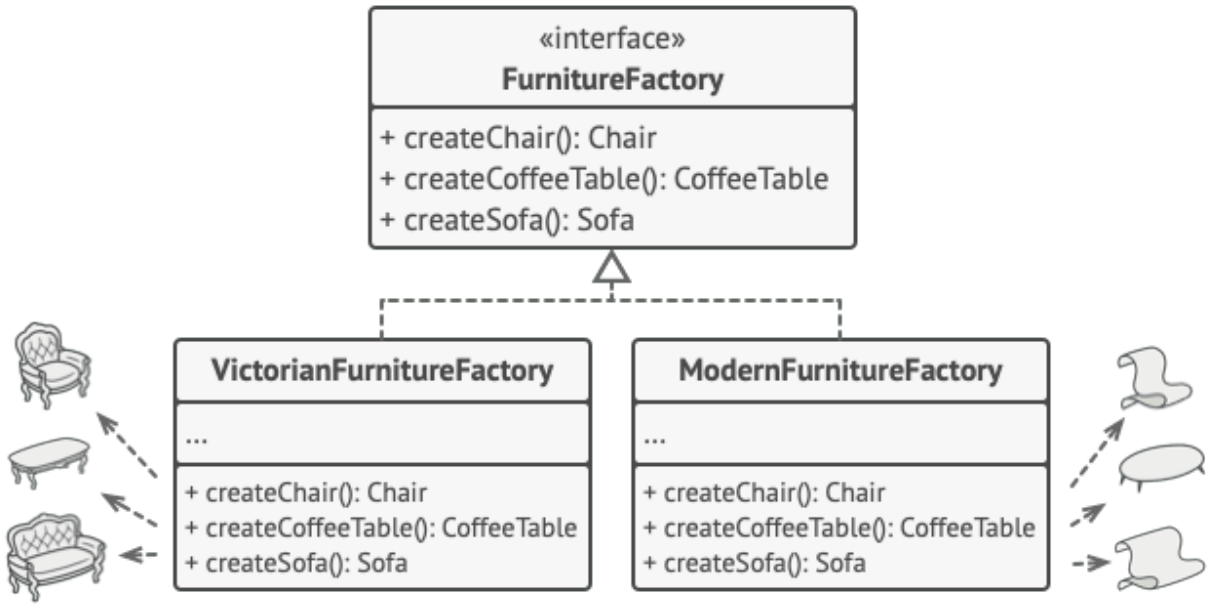
Кроме того, вы не хотите вносить изменения в существующий код при добавлении новых продуктов или семейств в программу. Поставщики часто обновляют свои каталоги, и вы бы не хотели менять уже написанный код каждый раз при получении новых моделей мебели.

- Решение

Для начала паттерн Абстрактная фабрика предлагает выделить общие интерфейсы для отдельных продуктов, составляющих семейства. Так, все вариации кресел получат общий интерфейс Кресло, все диваны реализуют интерфейс Диван и так далее.



Далее вы создаёте абстрактную фабрику – общий интерфейс, который содержит методы создания всех продуктов семейства (например, создатьКресло, создатьДиван и создатьСтолик). Эти операции должны возвращать абстрактные типы продуктов, представленные интерфейсами, которые мы выделили ранее – Кресла, Диваны и Столики.



Как насчёт вариаций продуктов? Для каждой вариации семейства продуктов мы должны создать свою собственную фабрику, реализовав абстрактный

интерфейс. Фабрики создают продукты одной вариации. Например, ФабрикаМодерн будет возвращать только КреслаМодерн, ДиваныМодерн и СтоликиМодерн.

Клиентский код должен работать как с фабриками, так и с продуктами только через их общие интерфейсы. Это позволит подавать в ваши классы любой тип фабрики и производить любые продукты, ничего не ломая.

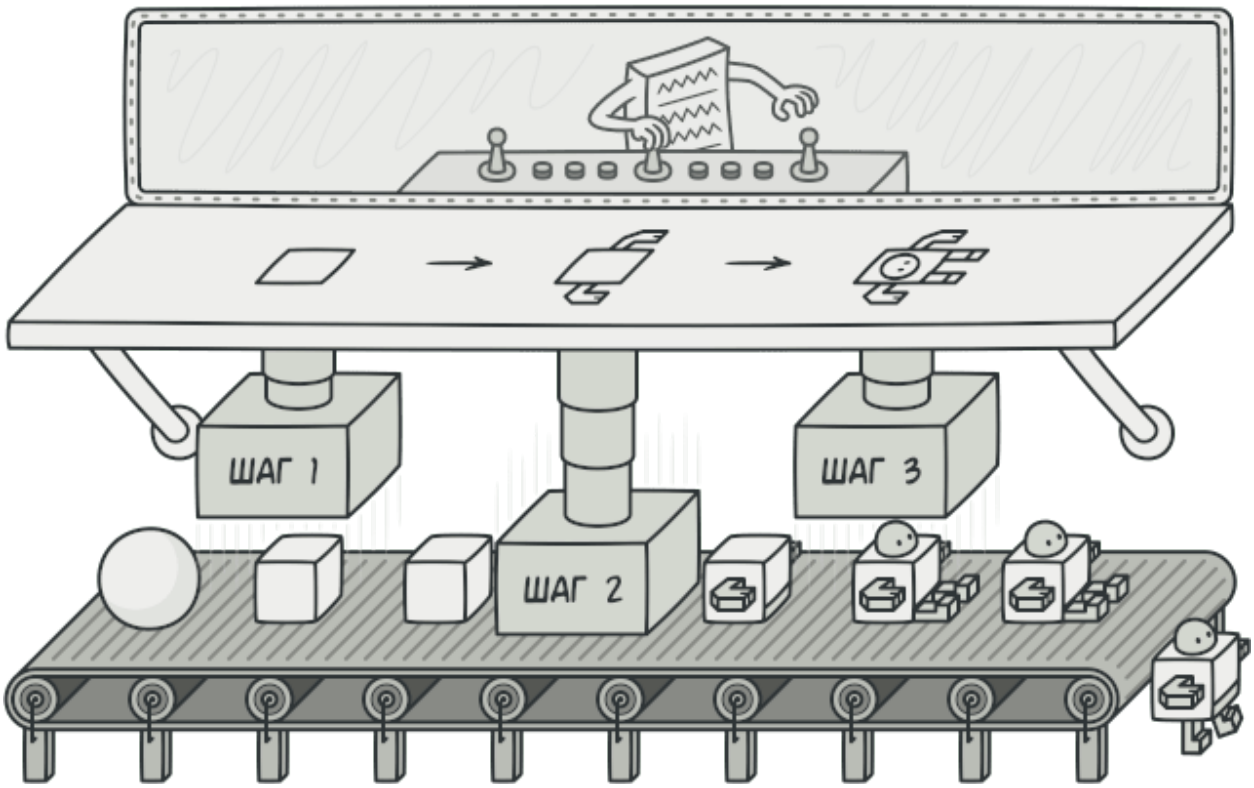
Например, клиентский код просит фабрику сделать стул. Он не знает, какого типа была эта фабрика. Он не знает, получит викторианский или модерновый стул. Для него важно, чтобы на стуле можно было сидеть и чтобы этот стул отлично смотрелся с диваном той же фабрики.

Осталось прояснить последний момент: кто создаёт объекты конкретных фабрик, если клиентский код работает только с интерфейсами фабрик? Обычно программа создаёт конкретный объект фабрики при запуске, причём

тип фабрики выбирается, исходя из параметров окружения или конфигурации.

c. Строитель

Строитель – это порождающий паттерн проектирования, который позволяет создавать сложные объекты пошагово. Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений объектов.



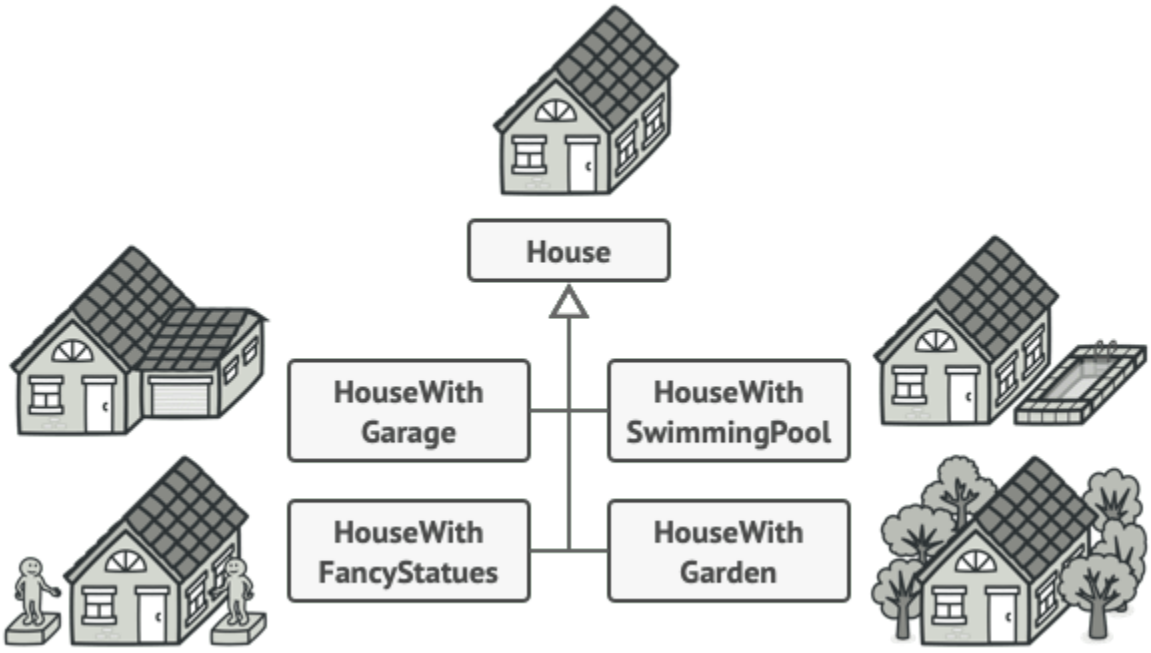
- Проблема

Представьте сложный объект, требующий кропотливой пошаговой инициализации множества полей и вложенных объектов. Код инициализации таких объектов обычно спрятан внутри монструозного конструктора с

десятком параметров. Либо ещё хуже – распылён по всему клиентскому коду.

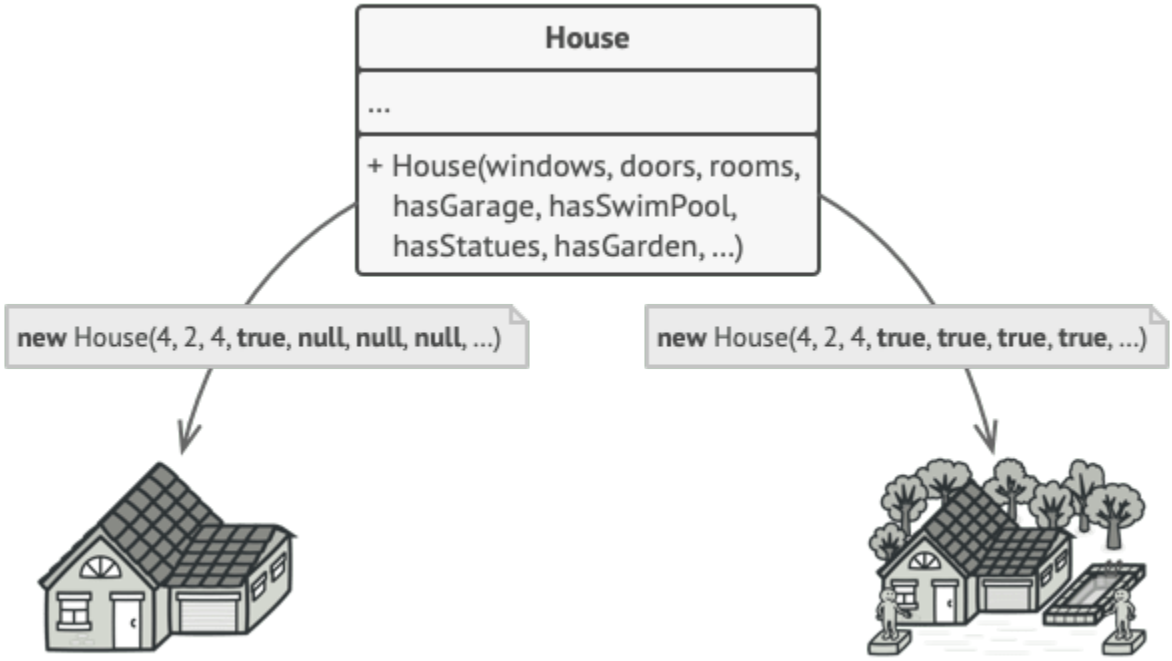
Например, давайте подумаем о том, как создать объект Дом. Чтобы построить стандартный дом, нужно поставить 4 стены, установить двери, вставить пару окон и положить крышу. Но что, если вы хотите дом побольше да посветлее, имеющий сад, бассейн и прочее добро?

Самое простое решение – расширить класс Дом, создав подклассы для всех комбинаций параметров дома. Проблема такого подхода – это громадное количество классов, которые вам придётся создать. Каждый новый параметр, вроде цвета обоев или материала кровли, заставит вас создавать всё больше и больше классов для перечисления всех возможных вариантов.



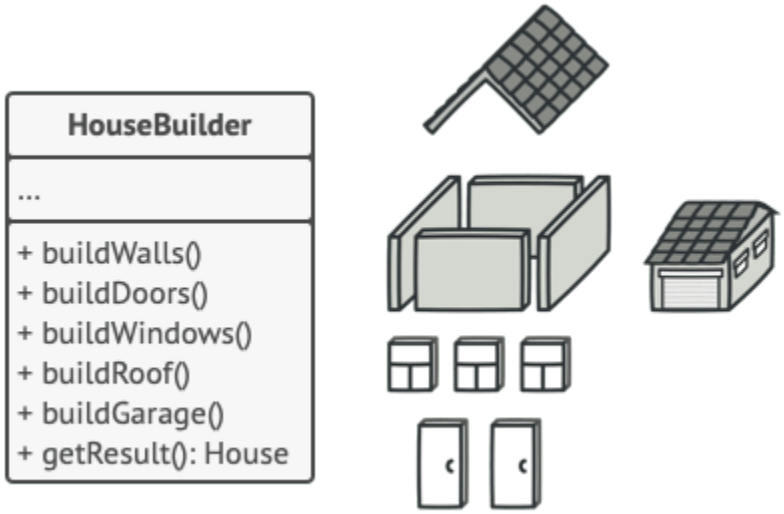
Чтобы не плодить подклассы, вы можете подойти к решению с другой стороны. Вы можете создать гигантский конструктор Дома, принимающий уйму параметров для контроля над создаваемым продуктом. Действительно, это избавит вас от подклассов, но приведёт к другой проблеме.

Большая часть этих параметров будет простаивать, а вызовы конструктора будут выглядеть монструозно из-за длинного списка параметров. К примеру, далеко не каждый дом имеет бассейн, поэтому параметры, связанные с бассейнами, будут простаивать бесполезно в 99% случаев.



- Решение

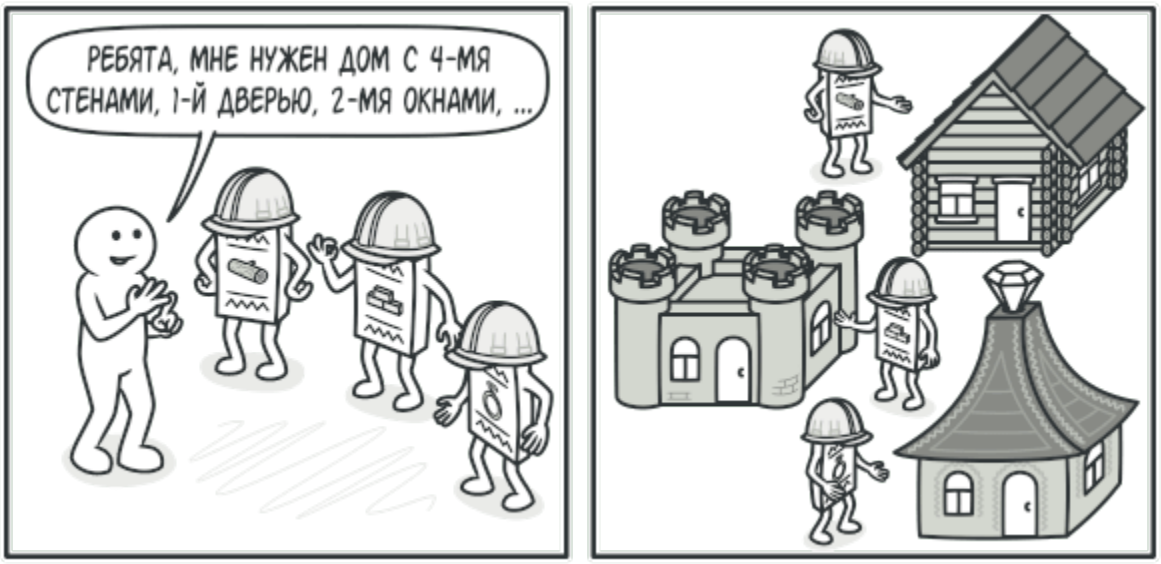
Паттерн Строитель предлагает вынести конструирование объекта за пределы его собственного класса, поручив это дело отдельным объектам, называемым строителями.



Паттерн предлагает разбить процесс конструирования объекта на отдельные шаги (например, построитьСтены, вставитьДвери и другие). Чтобы создать объект, вам нужно поочерёдно вызывать методы строителя. Причём не нужно запускать все шаги, а только те, что нужны для производства объекта определённой конфигурации.

Зачастую один и тот же шаг строительства может отличаться для разных вариаций производимых объектов. Например, деревянный дом потребует строительства стен из дерева, а каменный – из камня.

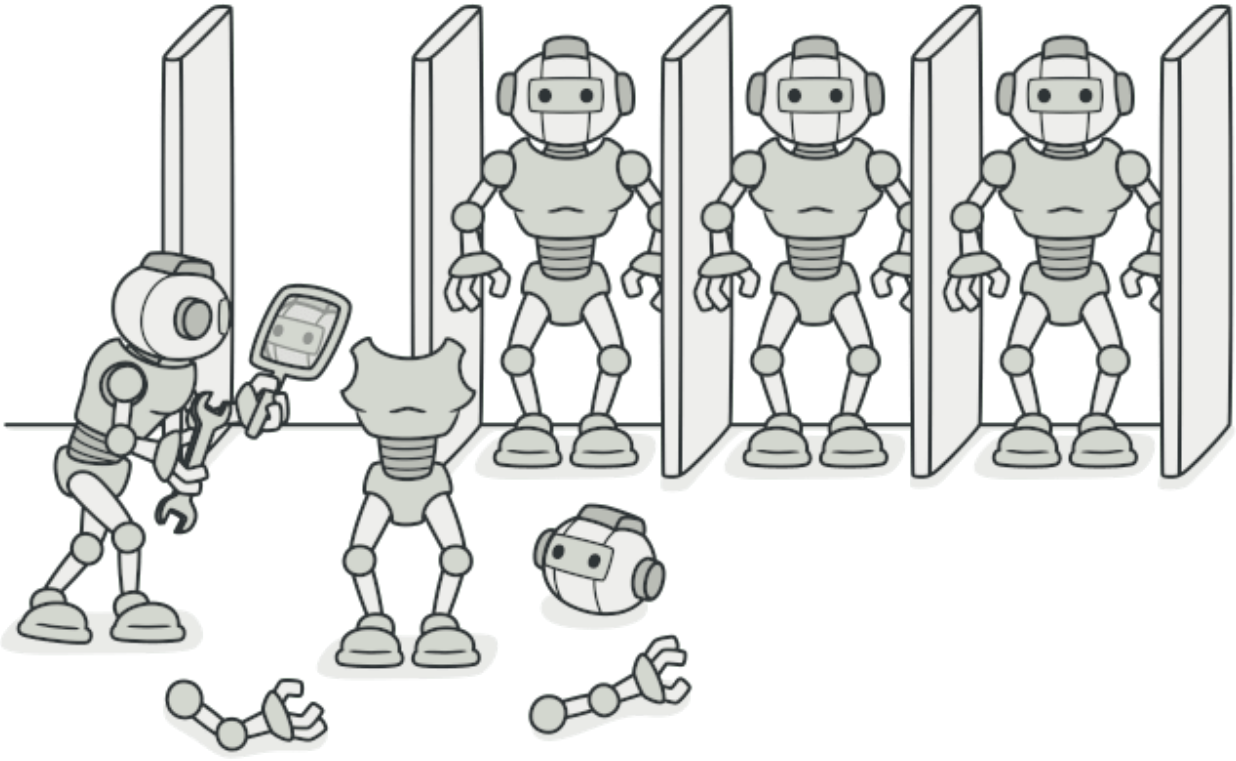
В этом случае вы можете создать несколько классов строителей, выполняющих одни и те же шаги по-разному. Используя этих строителей в одном и том же строительном процессе, вы сможете получать на выходе различные объекты.



Например, один строитель делает стены из дерева и стекла, другой из камня и железа, третий из золота и бриллиантов. Вызвав одни и те же шаги строительства, в первом случае вы получите обычный жилой дом, во втором – маленькую крепость, а в третьем – роскошное жилище. Замечу, что код, который вызывает шаги строительства, должен работать со строителями через общий интерфейс, чтобы их можно было свободно взаимозаменять.

d. Прототип

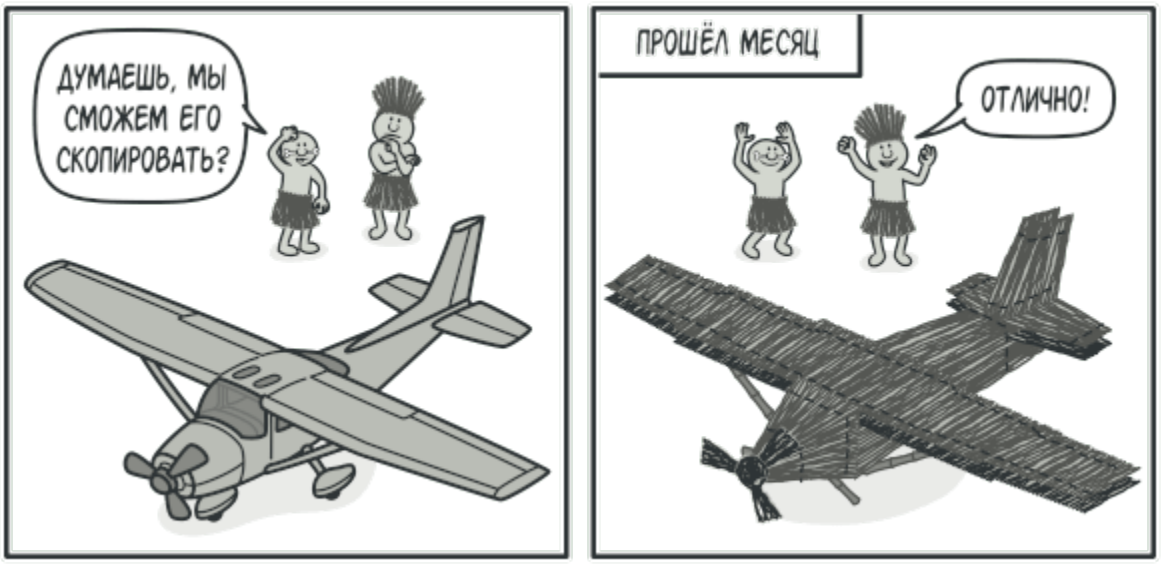
Прототип – это порождающий паттерн проектирования, который позволяет копировать объекты, не вдаваясь в подробности их реализации.



- Проблема

У вас есть объект, который нужно скопировать. Как это сделать? Нужно создать пустой объект такого же класса, а затем поочерёдно скопировать значения всех полей из старого объекта в новый.

Прекрасно! Но есть нюанс. Не каждый объект удастся скопировать таким образом, ведь часть его состояния может быть приватной, а значит – недоступной для остального кода программы.



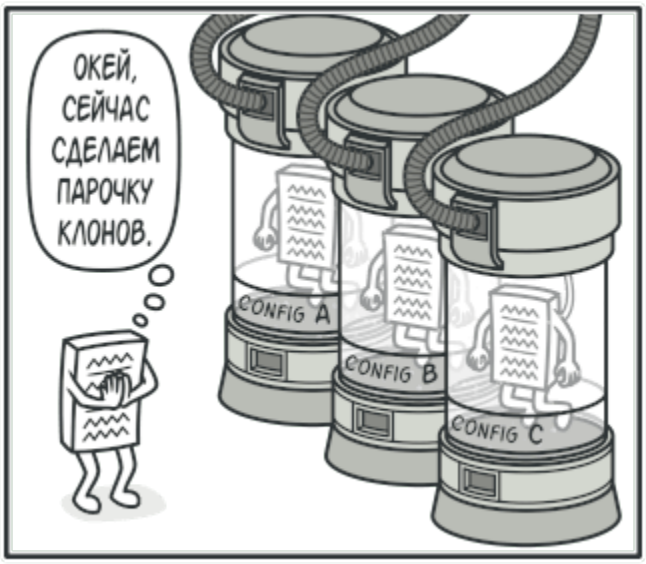
Но есть и другая проблема. Копирующий код станет зависим от классов копируемых объектов. Ведь, чтобы перебрать все поля объекта, нужно привязаться к его классу. Из-за этого вы не сможете копировать объекты, зная только их интерфейсы, а не конкретные классы.

- Решение

Паттерн Прототип поручает создание копий самим копируемым объектам. Он вводит общий интерфейс для всех объектов, поддерживающих клонирование. Это позволяет копировать объекты, не привязываясь к их конкретным классам. Обычно такой интерфейс имеет всего один метод clone.

Реализация этого метода в разных классах очень схожа. Метод создаёт новый объект текущего класса и копирует в него значения всех полей собственного объекта. Так получится скопировать даже приватные поля, так как большинство языков программирования разрешает доступ к приватным полям любого объекта текущего класса.

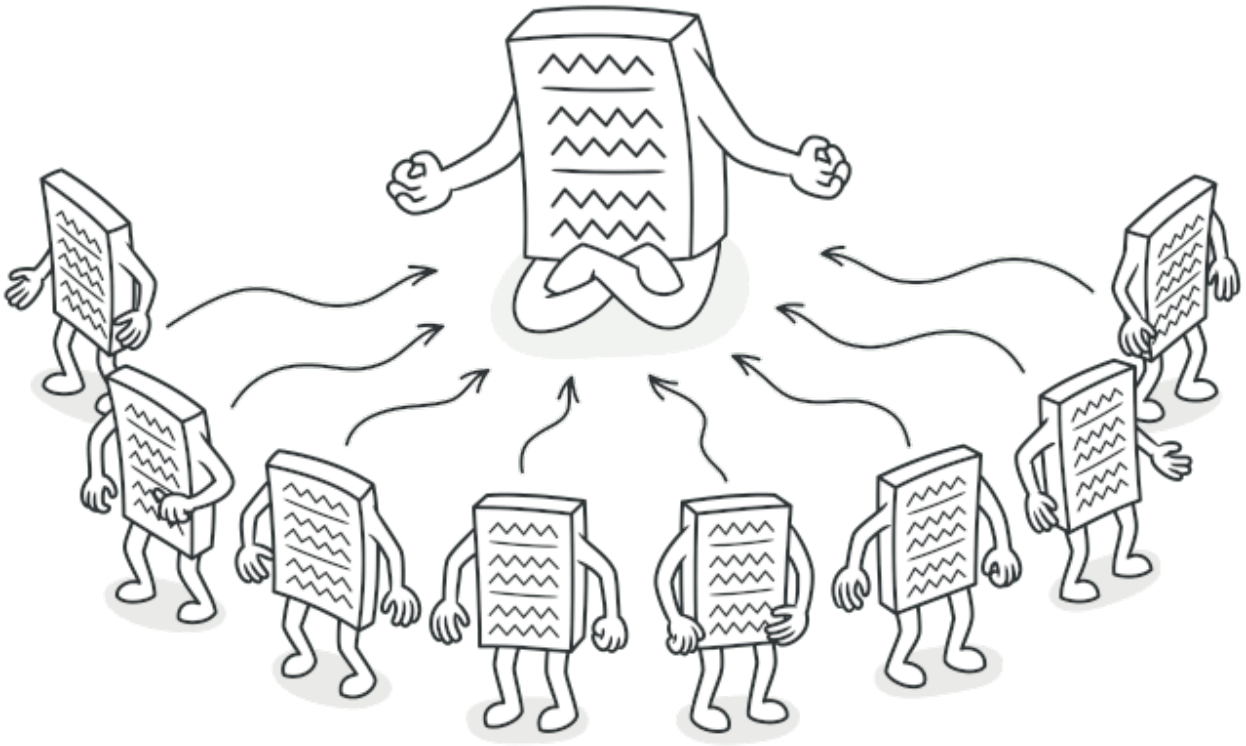
Объект, который копируют, называется прототипом (откуда и название паттерна). Когда объекты программы содержат сотни полей и тысячи возможных конфигураций, прототипы могут служить своеобразной альтернативой созданию подклассов.



В этом случае все возможные прототипы заготавливаются и настраиваются на этапе инициализации программы. Потом, когда программе нужен новый объект, она создаёт копию из приготовленного прототипа.

e. Одиночка

Одиночка – это порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.



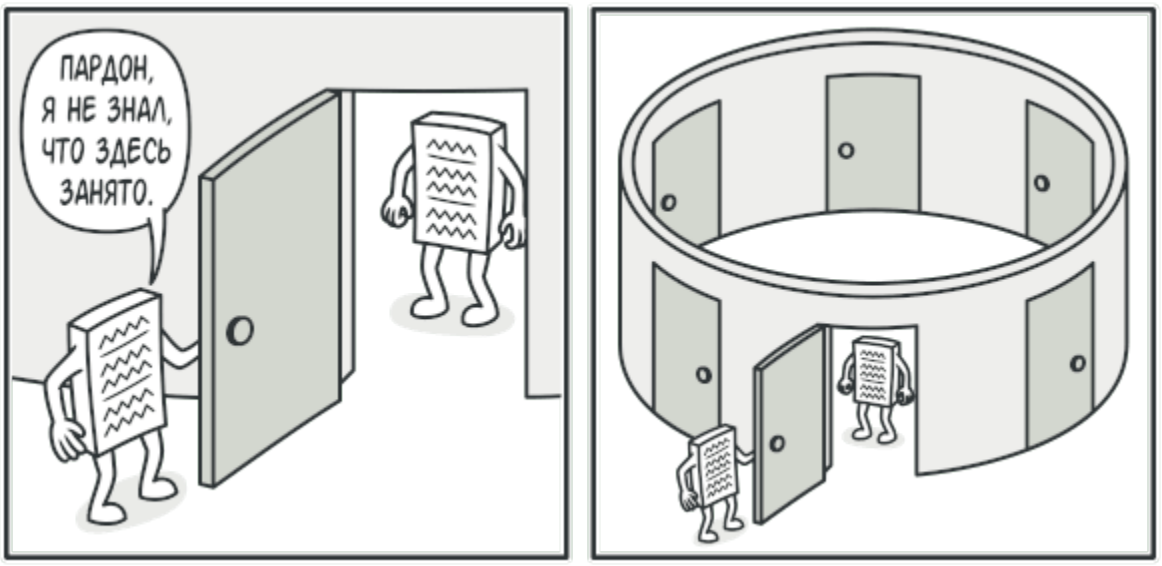
- Проблема

Одиночка решает сразу две проблемы, нарушая принцип единственной ответственности класса.

- ✓ Гарантирует наличие единственного экземпляра класса. Чаще всего это полезно для доступа к какому-то общему ресурсу, например, базе данных.

Представьте, что вы создали объект, а через некоторое время пробуете создать ещё один. В этом случае хотелось бы получить старый объект, вместо создания нового.

Такое поведение невозможно реализовать с помощью обычного конструктора, так как конструктор класса всегда возвращает новый объект.



- ✓ Предоставляет глобальную точку доступа. Это не просто глобальная переменная, через которую можно достучаться к определённому объекту. Глобальные переменные не защищены от записи, поэтому любой код может подменять их значения без вашего ведома.

Но есть и другой нюанс. Неплохо бы хранить в одном месте и код, который решает проблему №1, а также иметь к нему простой и доступный интерфейс.

Интересно, что в наше время паттерн стал настолько известен, что теперь люди называют «одиночками» даже те классы, которые решают лишь одну из проблем, перечисленных выше.

- Решение

Все реализации одиночки сводятся к тому, чтобы скрыть конструктор по умолчанию и создать публичный статический метод, который и будет контролировать жизненный цикл объекта-одиночки.

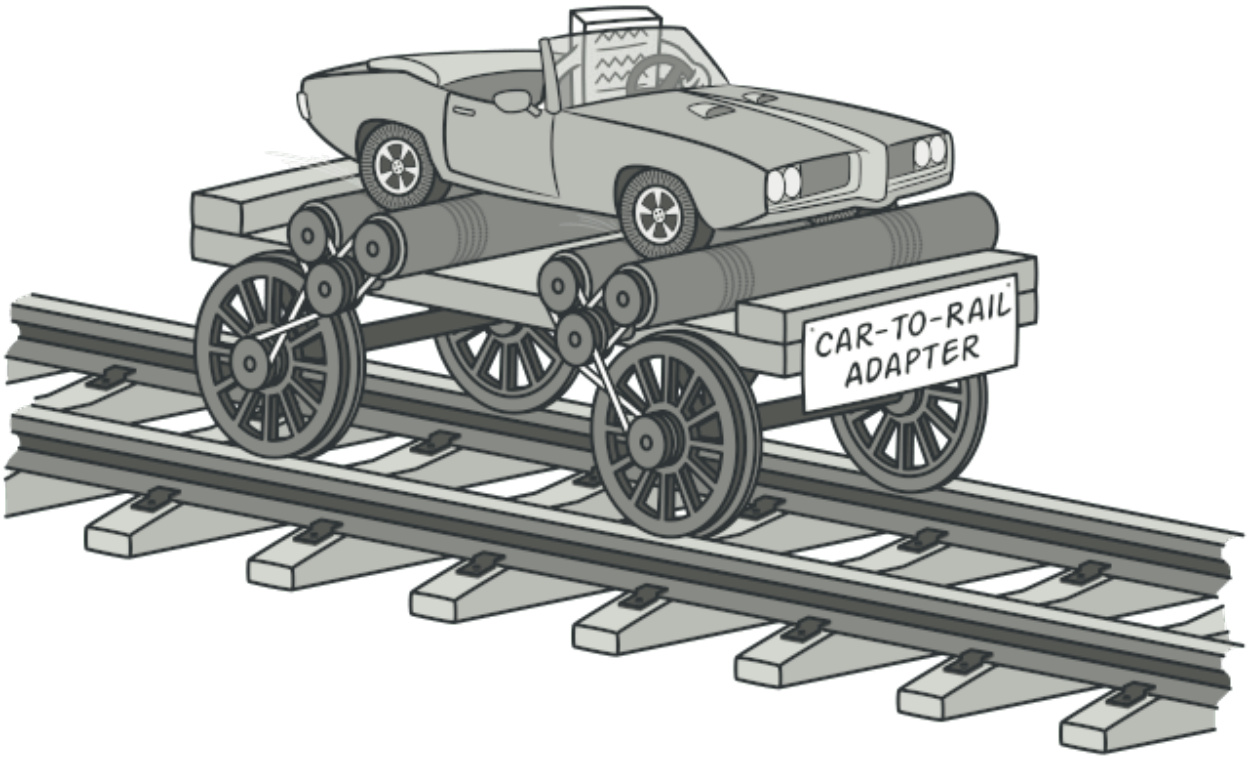
Если у вас есть доступ к классу одиночки, значит, будет доступ и к этому статическому методу. Из какой точки кода вы бы его ни вызвали, он всегда будет отдавать один и тот же объект.

2. Структурные паттерны

Список структурных паттернов проектирования, которые отвечают за построение удобных в поддержке иерархий классов.

a. Адаптер

Адаптер – это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе.



Проблема

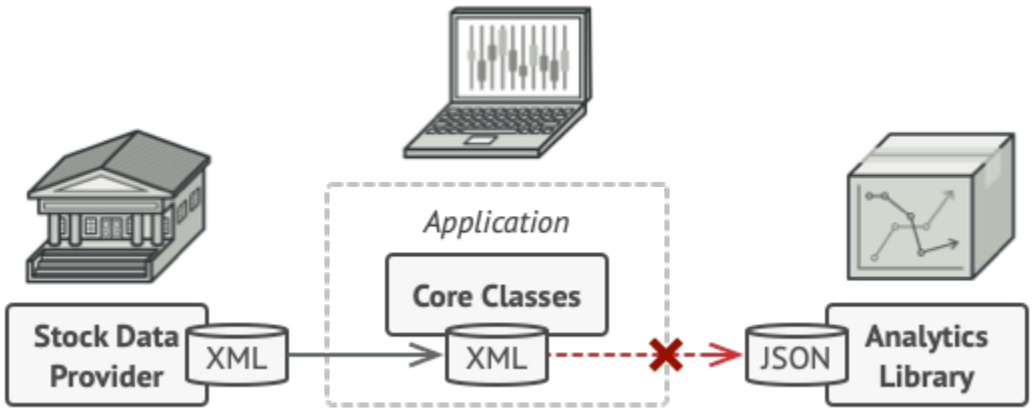
Представьте, что вы делаете приложение для торговли на бирже. Ваше приложение скачивает биржевые котировки из нескольких источников в XML, а затем рисует красивые графики.

В какой-то момент вы решаете улучшить приложение, применив стороннюю библиотеку аналитики. Но вот беда – библиотека поддерживает только формат данных JSON, несовместимый с вашим приложением.

- Проблема

Представьте, что 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. Но вот беда – библиотека поддерживает только формат данных JSON, несовместимый с вашим приложением.



Вы смогли бы переписать библиотеку, чтобы та поддерживала формат XML. Но, во-первых, это может нарушить работу существующего кода, который уже зависит от библиотеки. А во-вторых, у вас может просто не быть доступа к её исходному коду.

- Решение

Вы можете создать адаптер. Это объект-переводчик, который трансформирует интерфейс или данные одного объекта в такой вид, чтобы он стал понятен другому объекту.

При этом адаптер оборачивает один из объектов, так что другой объект даже не знает о наличии первого. Например, вы можете обернуть объект, работающий в метрах, адаптером, который бы конвертировал данные в футы.

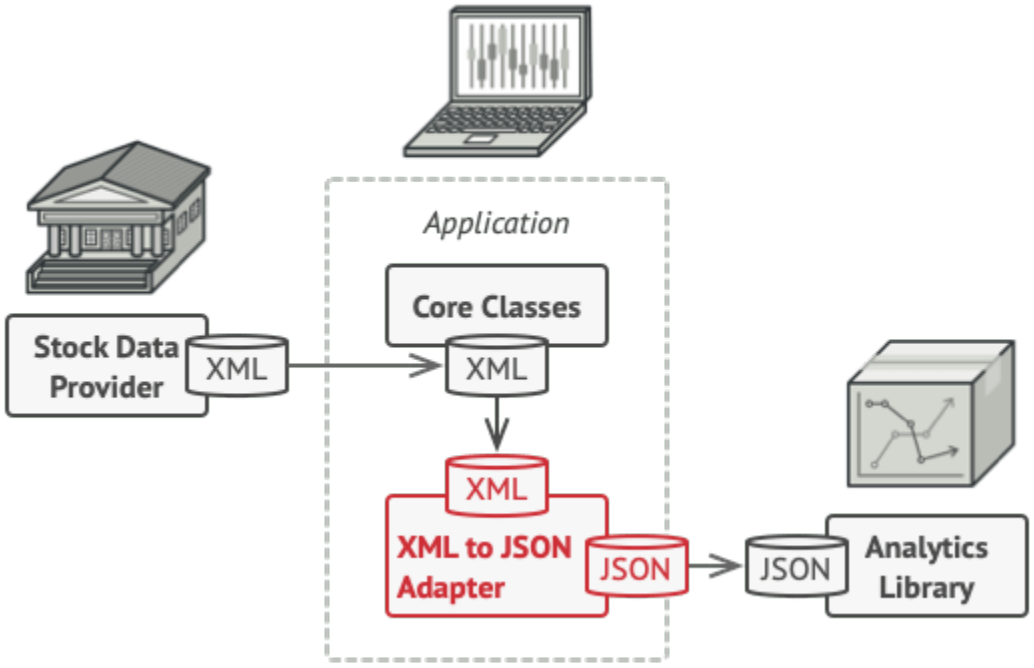
Адаптеры могут не только переводить данные из одного формата в другой, но и помогать объектам с разными интерфейсами работать сообща. Это работает так:

Адаптер имеет интерфейс, который совместим с одним из объектов.

Поэтому этот объект может свободно вызывать методы адаптера.

Адаптер получает эти вызовы и перенаправляет их второму объекту, но уже в том формате и последовательности, которые понятны второму объекту.

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

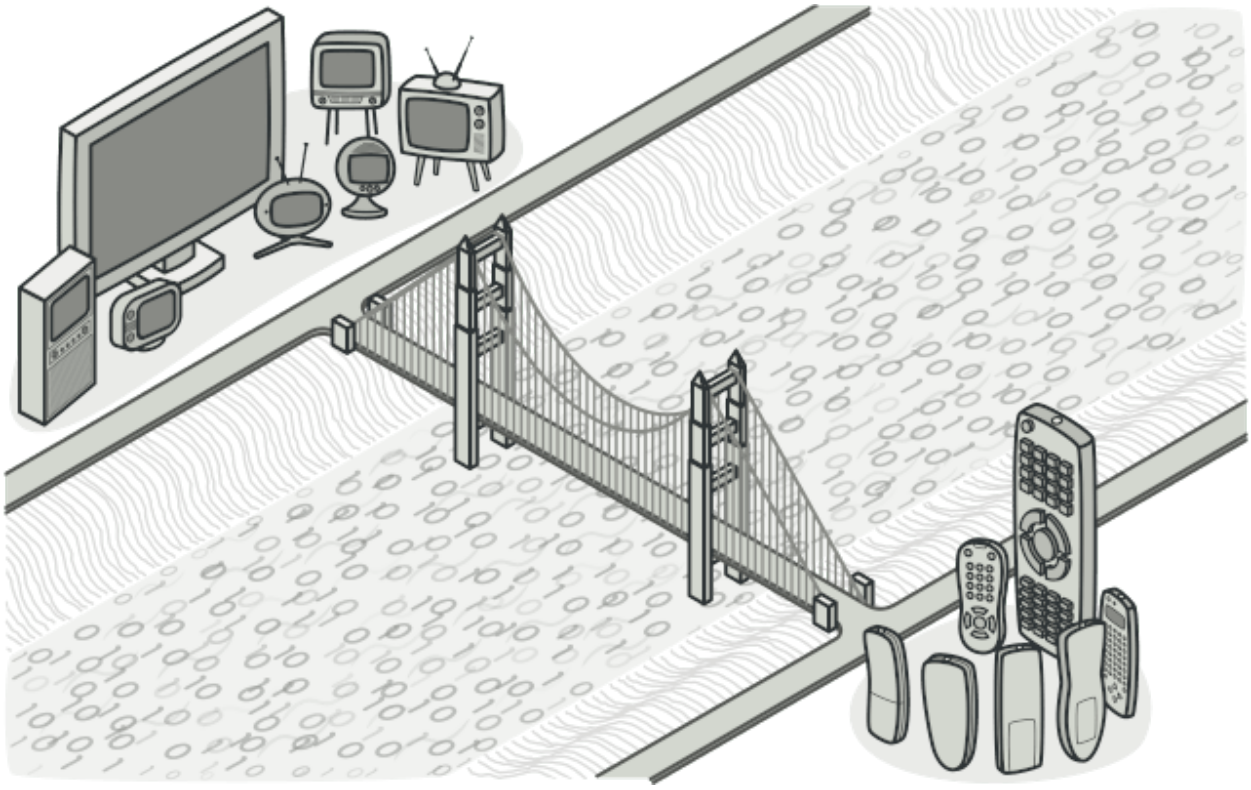


Таким образом, в приложении биржевых котировок вы могли бы создать класс XML_To_JSON_Adapter, который бы оборачивал объект того или иного класса библиотеки аналитики. Ваш код посылал бы адаптеру запросы в

формате XML, а адаптер сначала транслировал входящие данные в формат JSON, а затем передавал бы их методам обёрнутого объекта аналитики.

b. Мост

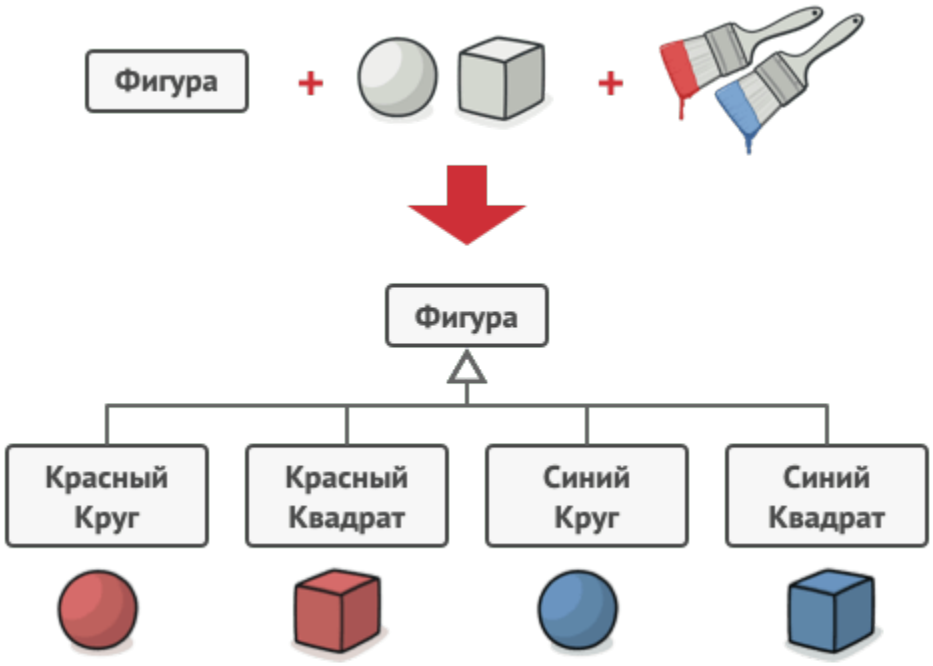
Мост – это структурный паттерн проектирования, который разделяет один или несколько классов на две отдельные иерархии – абстракцию и реализацию, позволяя изменять их независимо друг от друга.



Проблема

Абстракция? Реализация?! Звучит пугающе! Чтобы понять, о чём идёт речь, давайте разберём очень простой пример.

У вас есть класс геометрических Фигур, который имеет подклассы Круг и Квадрат. Вы хотите расширить иерархию фигур по цвету, то есть иметь Красные и Синие фигуры. Но чтобы всё это объединить, вам придётся создать 4 комбинации подклассов, вроде СинииеКруги и КрасныеКвадраты.



При добавлении новых видов фигур и цветов количество комбинаций будет расти в геометрической прогрессии. Например, чтобы ввести в программу фигуры треугольников, придётся создать сразу два новых подкласса треугольников под каждый цвет. После этого новый цвет потребует создания уже трёх классов для всех видов фигур. Чем дальше, тем хуже.

Решение

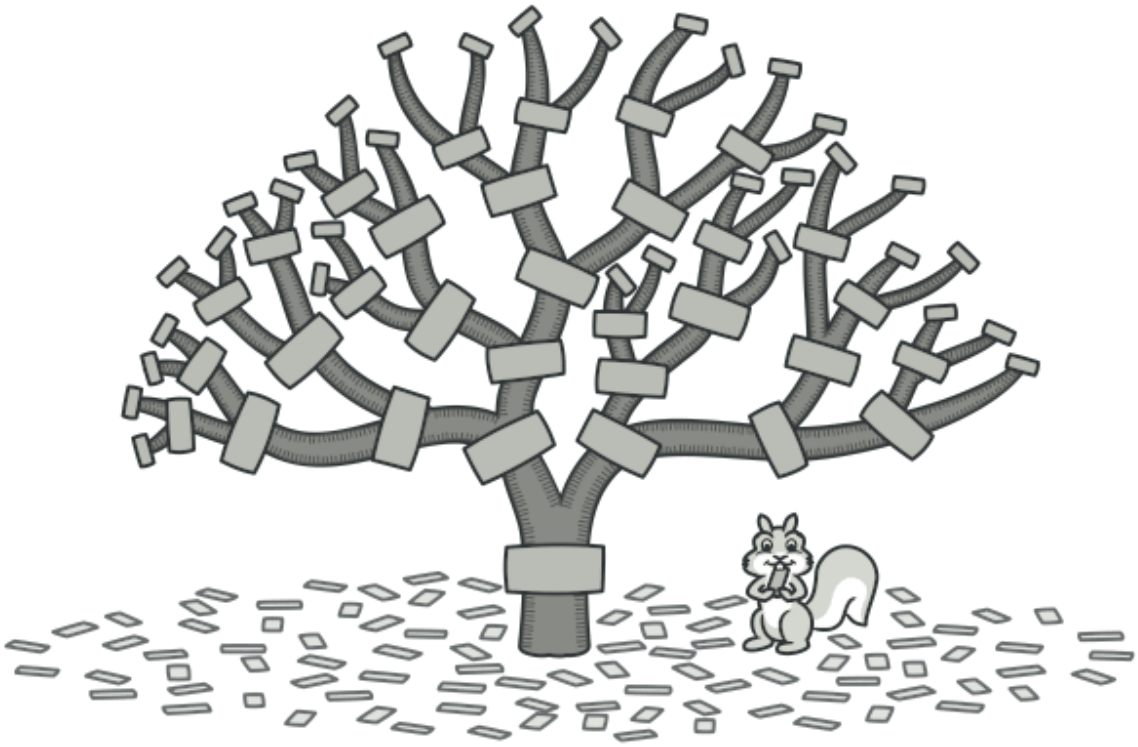
Корень проблемы заключается в том, что мы пытаемся расширить классы фигур сразу в двух независимых плоскостях – по виду и по цвету. Именно это приводит к разрастанию дерева классов.

Паттерн Мост предлагает заменить наследование агрегацией или композицией. Для этого нужно выделить одну из таких «плоскостей» в отдельную иерархию и ссылаться на объект этой иерархии, вместо хранения его состояния и поведения внутри одного класса.

Таким образом, мы можем сделать Цвет отдельным классом с подклассами Красный и Синий. Класс Фигур получит ссылку на объект Цвета и сможет делегировать ему работу, если потребуется. Такая связь и станет мостом между Фигурами и Цветом. При добавлении новых классов цветов не потребуется трогать классы фигур и наоборот.

c. Компоновщик

Компоновщик – это структурный паттерн проектирования, который позволяет сгруппировать множество объектов в древовидную структуру, а затем работать с ней так, как будто это единый объект.

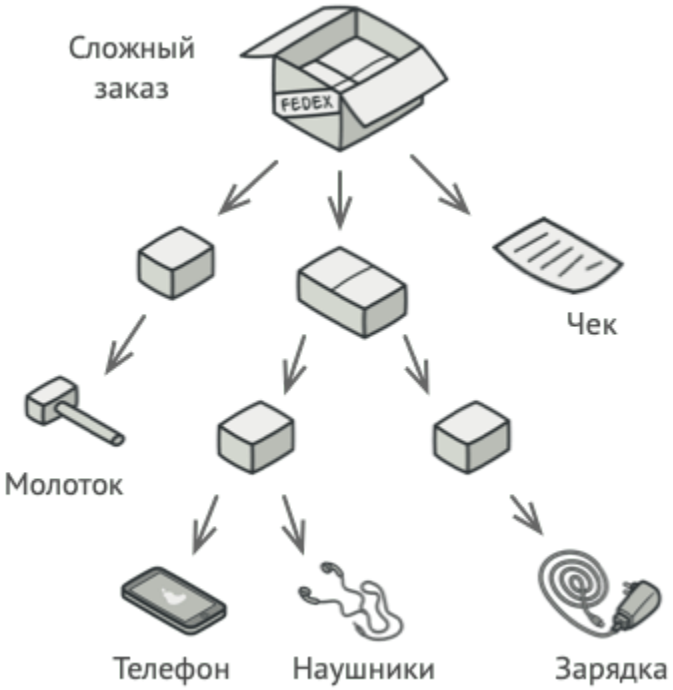


- Проблема

Паттерн Компоновщик имеет смысл только тогда, когда основная модель вашей программы может быть структурирована в виде дерева.

Например, есть два объекта: Продукт и Коробка. Коробка может содержать несколько Продуктов и других Коробок поменьше. Те, в свою очередь, тоже содержат либо Продукты, либо Коробки, и так далее.

Теперь предположим, ваши Продукты и Коробки могут быть частью заказов. Каждый заказ может содержать как простые Продукты без упаковки, так и составные Коробки. Ваша задача состоит в том, чтобы узнать цену всего заказа.



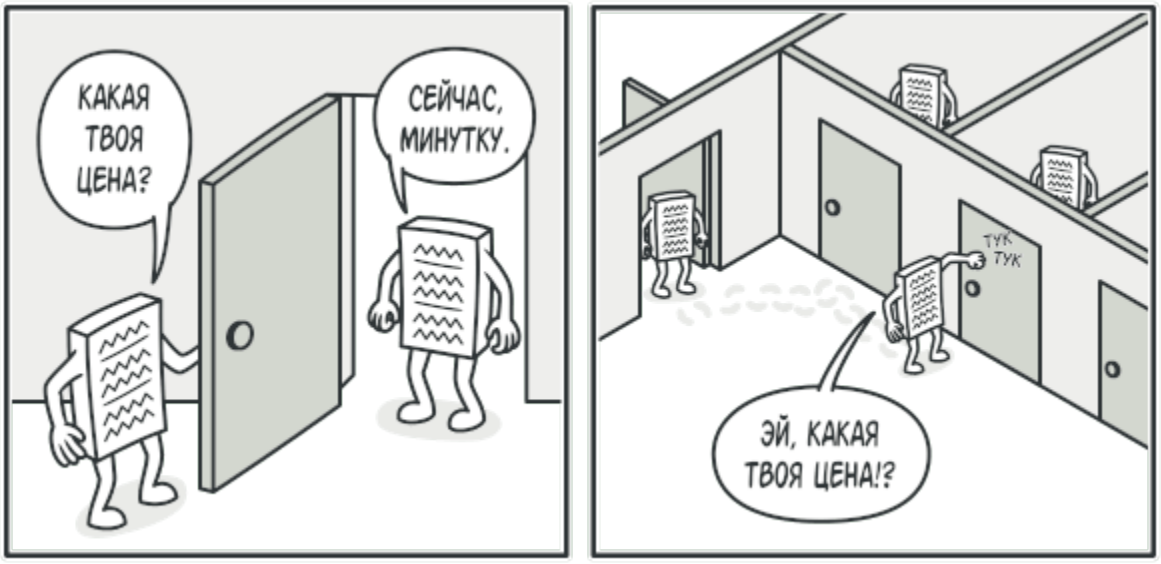
Если решать задачу в лоб, то вам потребуется открыть все коробки заказа, перебрать все продукты и посчитать их суммарную стоимость. Но это слишком хлопотно, так как типы коробок и их содержимое могут быть вам неизвестны. Кроме того, наперёд неизвестно и количество уровней вложенности коробок, поэтому перебрать коробки простым циклом не выйдет.

- Решение

Компоновщик предлагает рассматривать Продукт и Коробку через единый интерфейс с общим методом получения стоимости.

Продукт просто вернёт свою цену. Коробка спросит цену каждого предмета внутри себя и вернёт сумму результатов. Если одним из внутренних

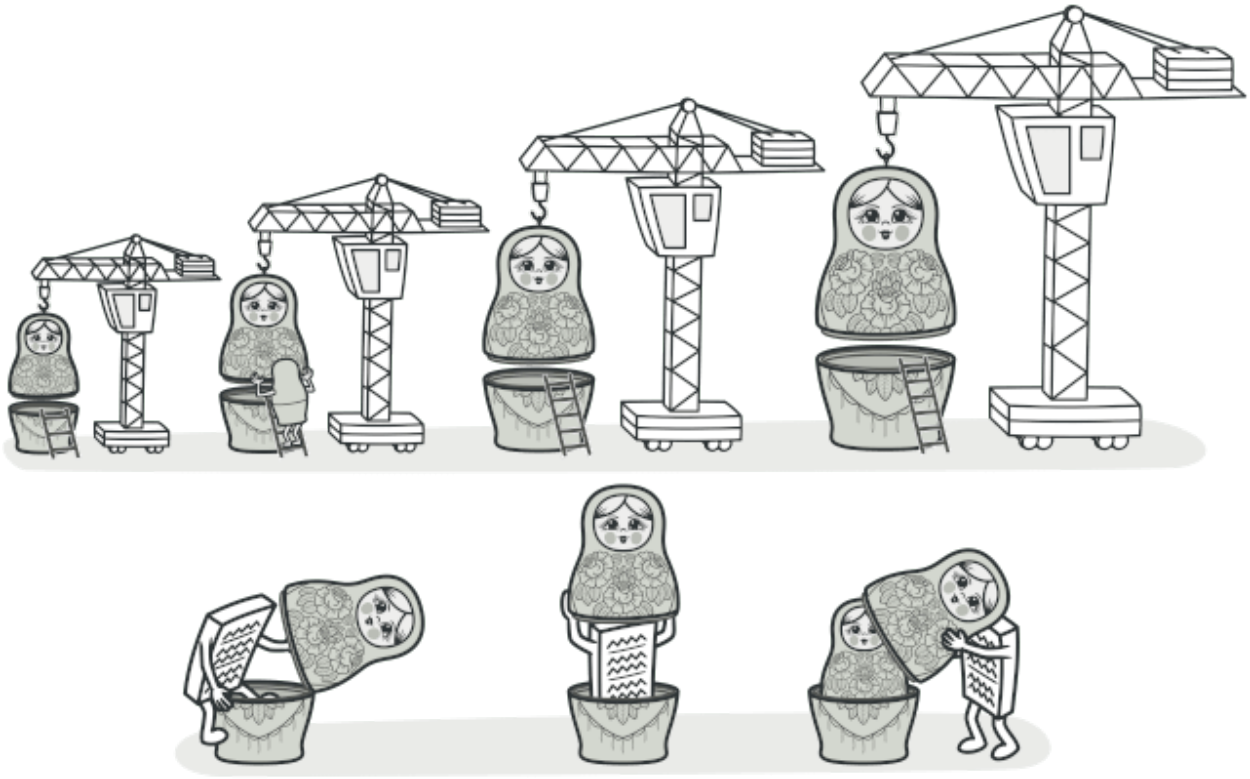
предметов окажется коробка поменьше, она тоже будет перебирать своё содержимое, и так далее, пока не будут посчитаны все составные части.



Для вас, клиента, главное, что теперь не нужно ничего знать о структуре заказов. Вы вызываете метод получения цены, он возвращает цифру, а вы не тонете в горах картона и скотча.

d. Декоратор

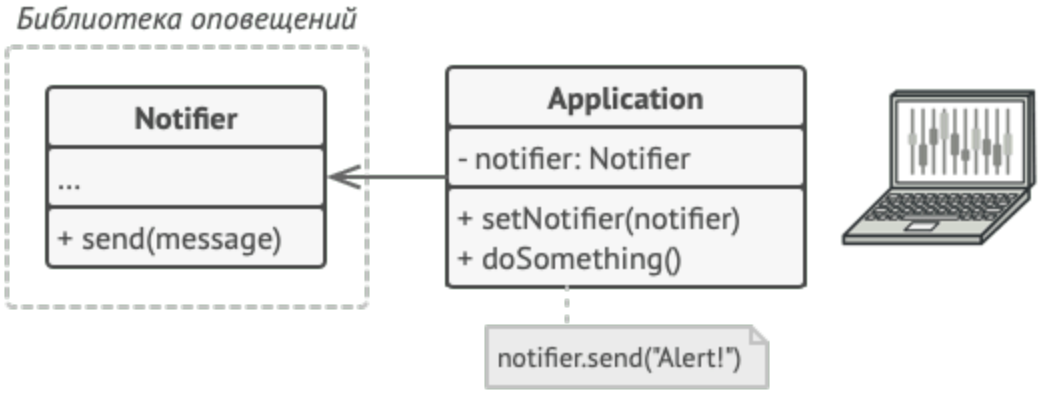
Декоратор – это структурный паттерн проектирования, который позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».



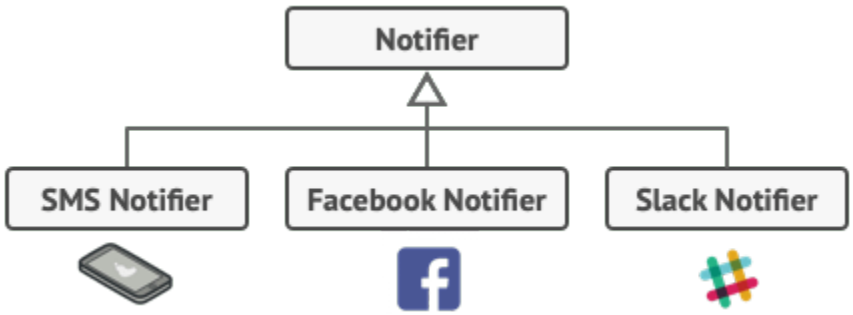
- Проблема

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

Основой библиотеки является класс Notifier с методом send, который принимает на вход строку-сообщение и высылает её всем администраторам по электронной почте. Сторонняя программа должна создать и настроить этот объект, указав кому отправлять оповещения, а затем использовать его каждый раз, когда что-то случается.



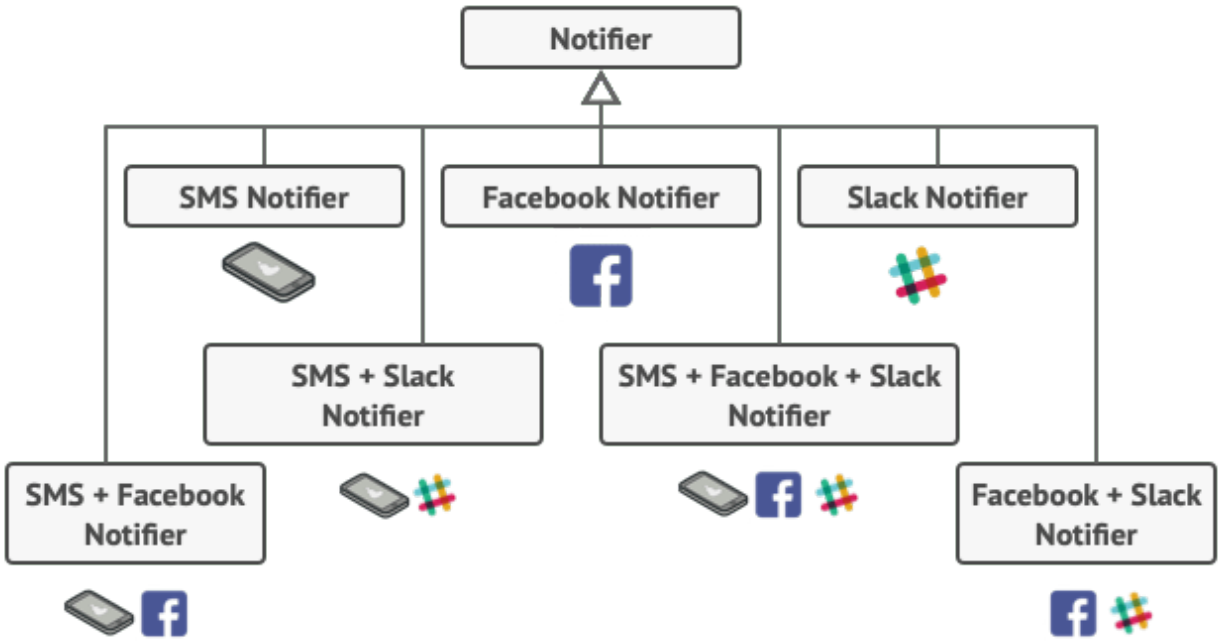
В какой-то момент стало понятно, что одних email-оповещений пользователям мало. Некоторые из них хотели бы получать извещения о критических проблемах через SMS. Другие хотели бы получать их в виде сообщений Facebook. Корпоративные пользователи хотели бы видеть сообщения в Slack.



Сначала вы добавили каждый из этих типов оповещений в программу, унаследовав их от базового класса Notifier. Теперь пользователь выбирал один из типов оповещений, который и использовался в дальнейшем.

Но затем кто-то резонно спросил, почему нельзя выбрать несколько типов оповещений сразу? Ведь если вдруг в вашем доме начался пожар, вы бы хотели получить оповещения по всем каналам, не так ли?

Вы попытались реализовать все возможные комбинации подклассов оповещений. Но после того как вы добавили первый десяток классов, стало ясно, что такой подход невероятно раздувает код программы.



Итак, нужен какой-то другой способ комбинирования поведения объектов, который не приводит к взрыву количества подклассов.

- Решение

Наследование – это первое, что приходит в голову многим программистам, когда нужно расширить какое-то существующее поведение. Но механизм наследования имеет несколько досадных проблем.

Он статичен. Вы не можете изменить поведение существующего объекта. Для этого вам надо создать новый объект, выбрав другой подкласс.

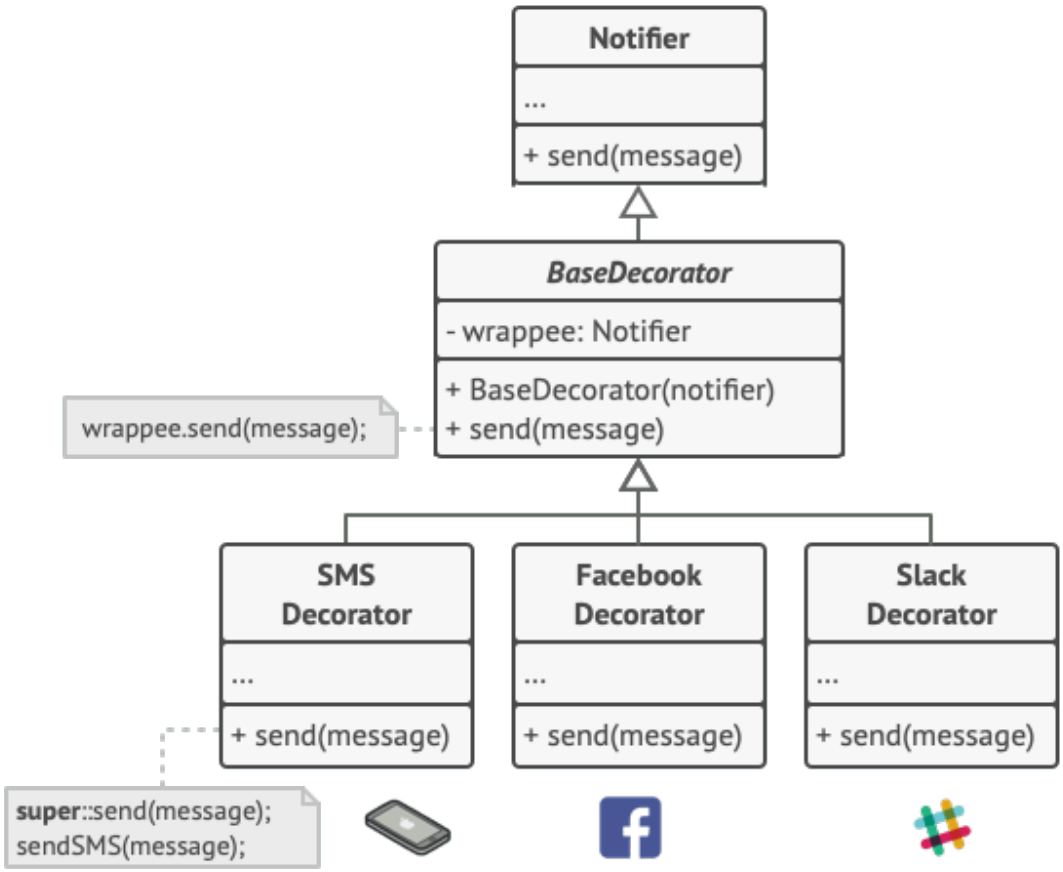
Он не разрешает наследовать поведение нескольких классов одновременно. Из-за этого вам приходится создавать множество подклассов-комбинаций для получения совмещённого поведения.

Одним из способов обойти эти проблемы является замена наследования агрегацией либо композицией . Это когда один объект содержит ссылку на другой и делегирует ему работу, вместо того чтобы самому наследовать его поведение. Как раз на этом принципе построен паттерн Декоратор.

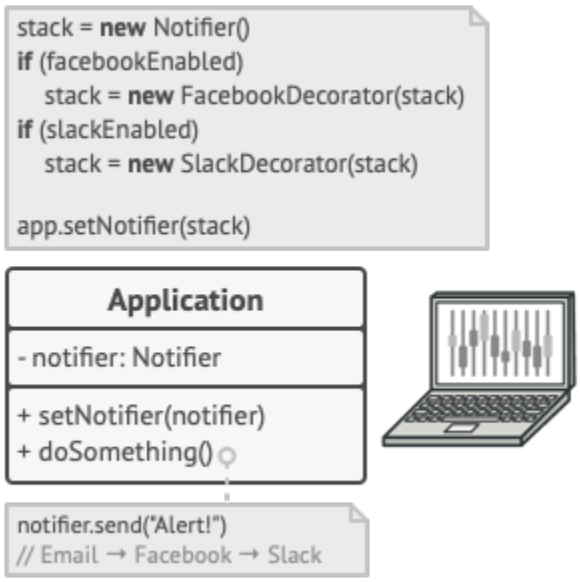
Декоратор имеет альтернативное название – обёртка. Оно более точно описывает суть паттерна: вы помещаете целевой объект в другой объект-обёртку, который запускает базовое поведение объекта, а затем добавляет к результату что-то своё.

Оба объекта имеют общий интерфейс, поэтому для пользователя нет никакой разницы, с каким объектом работать – чистым или обёрнутым. Вы можете использовать несколько разных обёрток одновременно – результат будет иметь объединённое поведение всех обёрток сразу.

В примере с оповещениями мы оставим в базовом классе простую отправку по электронной почте, а расширенные способы отправки сделаем декораторами.



Сторонняя программа, выступающая клиентом, во время первичной настройки будет заворачивать объект оповещений в те обёртки, которые соответствуют желаемому способу оповещения.

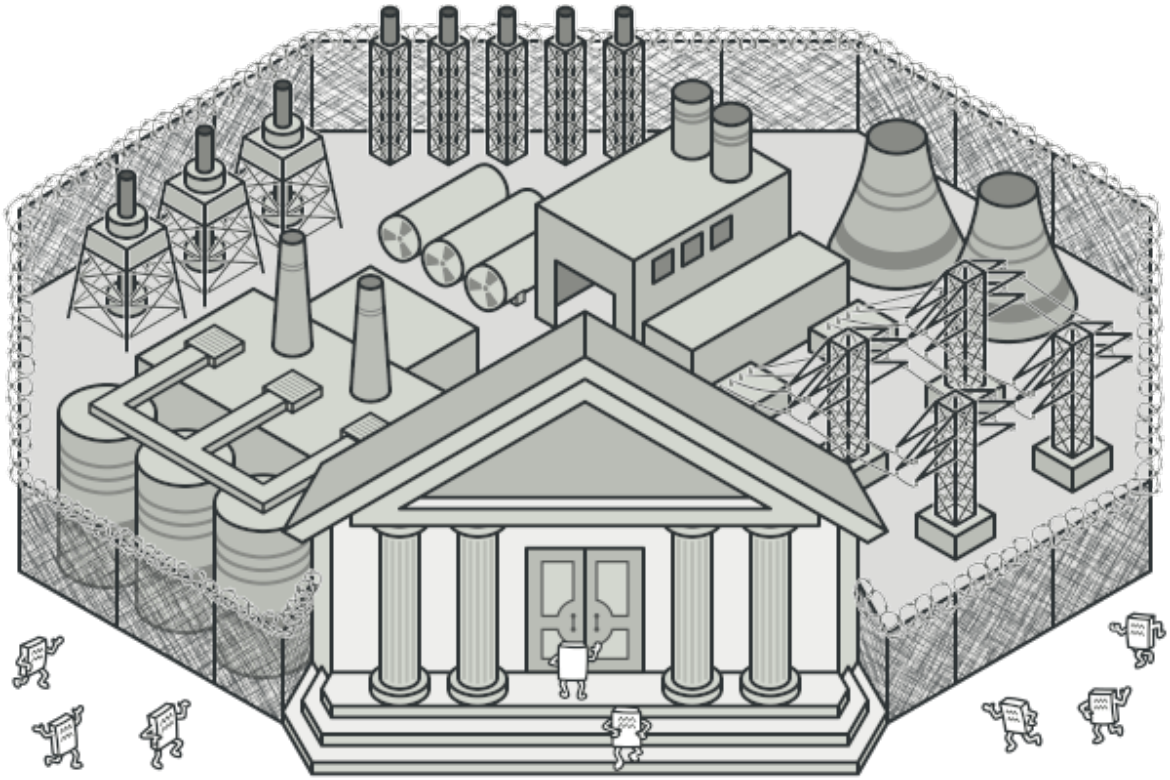


Последняя обёртка в списке и будет тем объектом, с которым клиент будет работать в остальное время. Для остального клиентского кода, по сути, ничего не изменится, ведь все обёртки имеют точно такой же интерфейс, что и базовый класс оповещений.

Таким же образом можно изменять не только способ доставки оповещений, но и форматирование, список адресатов и так далее. К тому же клиент может «дообернуть» объект любыми другими обёртками, когда ему захочется.

e. Фасад

Фасад – это структурный паттерн проектирования, который предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку.



- Проблема

Вашему коду приходится работать с большим количеством объектов некой сложной библиотеки или фреймворка. Вы должны самостоятельно инициализировать эти объекты, следить за правильным порядком зависимостей и так далее.

В результате бизнес-логика ваших классов тесно переплетается с деталями реализации сторонних классов. Такой код довольно сложно понимать и поддерживать.

- Решение

Фасад – это простой интерфейс для работы со сложной подсистемой, содержащей множество классов. Фасад может иметь урезанный интерфейс,

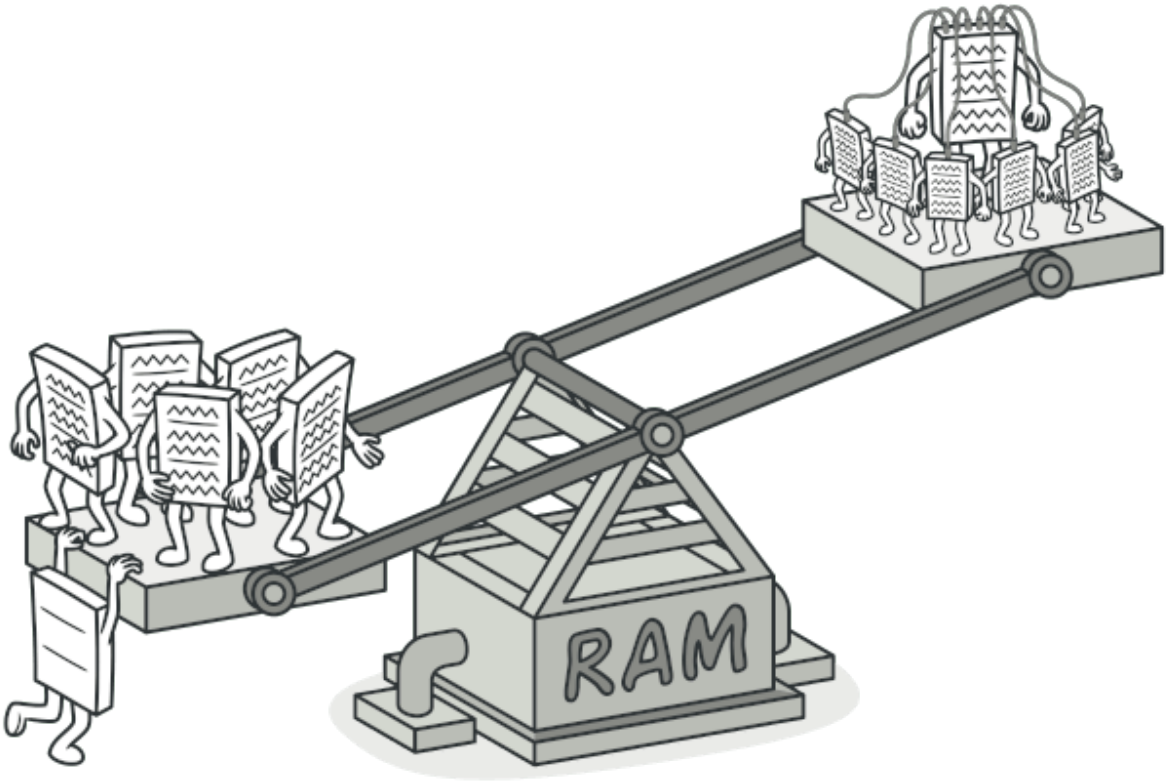
не имеющий 100% функциональности, которой можно достичь, используя сложную подсистему напрямую. Но он предоставляет именно те фичи, которые нужны клиенту, и скрывает все остальные.

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

К примеру, программа, заливающая видео котиков в социальные сети, может использовать профессиональную библиотеку сжатия видео. Но всё, что нужно клиентскому коду этой программы – простой метод encode(filename, format). Создав класс с таким методом, вы реализуете свой первый фасад.

f. Легковес

Легковес – это структурный паттерн проектирования, который 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.

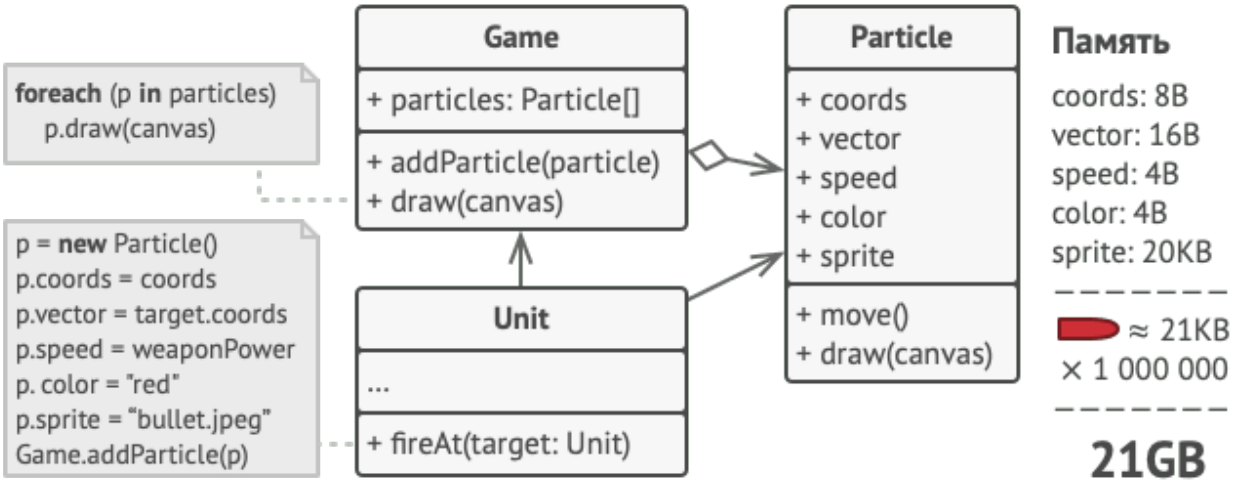


- Проблема

На досуге вы решили написать небольшую игру, в которой игроки перемещаются по карте и стреляют друг в друга. Фишкой игры должна была стать реалистичная система частиц. Пули, снаряды, осколки от взрывов – всё это должно красиво летать и радовать взгляд.

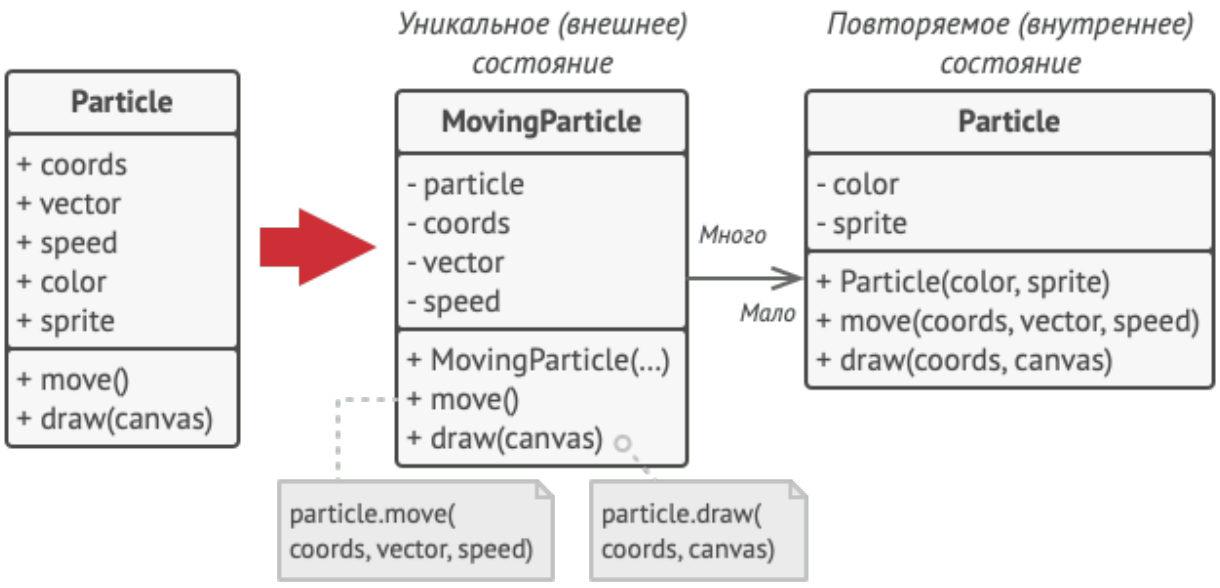
Игра отлично работала на вашем мощном компьютере. Однако ваш друг сообщил, что игра начинает тормозить и вылетает через несколько минут после запуска. Покопавшись в логах, вы обнаружили, что игра вылетает из-за недостатка оперативной памяти. У вашего друга компьютер значительно менее «прокачанный», поэтому проблема у него и проявляется так быстро.

И действительно, каждая частица представлена собственным объектом, имеющим множество данных. В определённый момент, когда побоище на экране достигает кульминации, новые объекты частиц уже не вмещаются в оперативную память компьютера, и программа вылетает.



- Решение

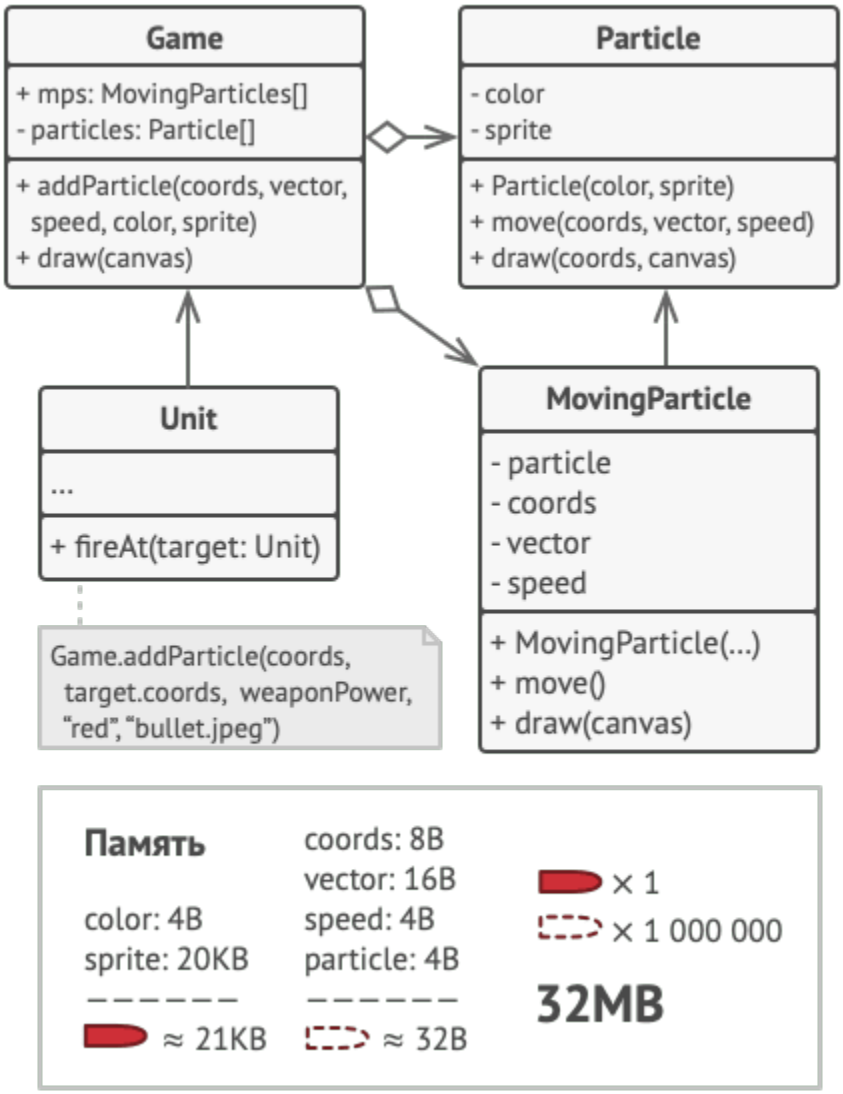
Если внимательно посмотреть на класс частиц, то можно заметить, что цвет и спрайт занимают больше всего памяти. Более того, они хранятся в каждом объекте, хотя фактически их значения одинаковы для большинства частиц.



Остальное состояние объектов – координаты, вектор движения и скорость – отличаются для всех частиц. Таким образом, эти поля можно рассматривать как контекст, в котором частица используется. А цвет и спрайт – это данные, не изменяющиеся во времени.

Неизменяемые данные объекта принято называть «внутренним состоянием». Все остальные данные – это «внешнее состояние».

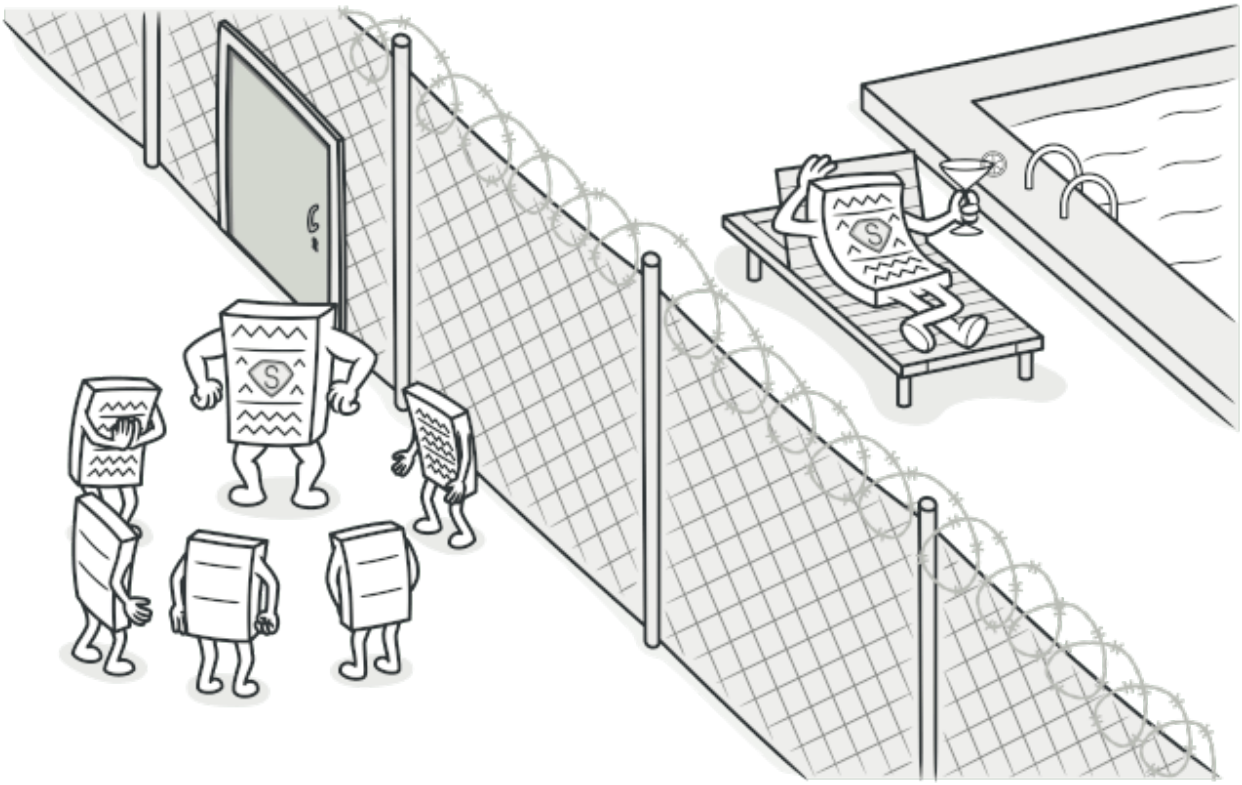
Паттерн Легковес предлагает не хранить в классе внешнее состояние, а передавать его в те или иные методы через параметры. Таким образом, одни и те же объекты можно будет повторно использовать в различных контекстах. Но главное – понадобится гораздо меньше объектов, ведь теперь они будут отличаться только внутренним состоянием, а оно имеет не так много вариаций.



В нашем примере с частицами достаточно будет оставить всего три объекта с отличающимися спрайтами и цветом – для пуль, снарядов и осколков. Несложно догадаться, что такие облегчённые объекты называют легковéсами .

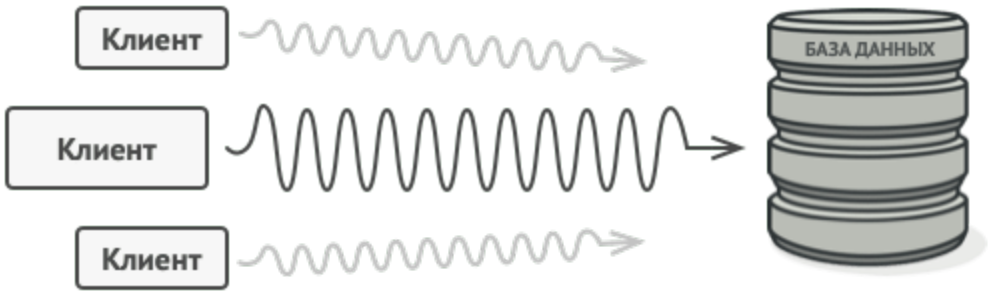
g. Заместитель

Заместитель – это структурный паттерн проектирования, который 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.



- Проблема

Для чего вообще контролировать доступ к объектам? Рассмотрим такой пример: у вас есть внешний ресурсоёмкий объект, который нужен не всё время, а изредка.



Мы могли бы создавать этот объект не в самом начале программы, а только тогда, когда он кому-то реально понадобится. Каждый клиент объекта получил бы некий код отложенной инициализации. Но, вероятно, это привело бы к множественному дублированию кода.

В идеале, этот код хотелось бы поместить прямо в служебный класс, но это не всегда возможно. Например, код класса может находиться в закрытой сторонней библиотеке.

- Решение

Паттерн Заместитель предлагает создать новый класс-дублёр, имеющий тот же интерфейс, что и оригинальный служебный объект. При получении запроса от клиента объект-заместитель сам бы создавал экземпляр служебного объекта и переадресовывал бы ему всю реальную работу.

Но в чём же здесь польза? Вы могли бы поместить в класс заместителя какую-то промежуточную логику, которая выполнялась бы до (или после) вызовов этих же методов в настоящем объекте. А благодаря одинаковому интерфейсу, объект-заместитель можно передать в любой код, ожидающий сервисный объект.