**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра «Информационные Системы и Технологий (ИСиТ)»**

КУРСОВАЯ РАБОТА

**по дисциплине «Программирование»**

Тема: «Основы ООП. Введение в паттерны проектирования»

|  |  |  |
| --- | --- | --- |
| Студент гр. 0324 |  | Зимацкий С. Н. |
| Преподаватель |  | Глущенко А.Г. |

Санкт-Петербург

2022

**ЗАДАНИЕ**

**НА КУРСОВУЮ РАБОТУ**

Студент Зимацкий С. Н.

Группа 0324

Тема работы: Основы ООП. Введение в паттерны проектирования

Исходные данные:

Содержание пояснительной записки:

|  |  |  |
| --- | --- | --- |
| Предполагаемый объем пояснительной записки:  Не менее 69 страниц. | | |
| Дата выдачи задания: 01.09.2021 | | |
| Дата сдачи реферата: 26.03.2022 | | |
| Дата защиты реферата: 26.03.2022 | | |
| Студент |  | Зимацкий С. Н. |
| Преподаватель |  | Глущенко А.Г. |

**Аннотация**

Лабораторные работы, объединённые в курсовую работу.

Выполненные работы покрывают следующие разделы:

* Классы.
* Объектно-ориентированная парадигма.
* Принципы объектно-ориентированного программирования.
* Паттерны проектирования.

В процессе выполнения работ были изучены вышеупомянутые разделы, и на основании изученного реализована программа.

Программа представляет собой игру, где можно управлять Университетом, создавать Юнитов (рабочих), управлять ими, взаимодействовать с окружающей средой, передвигаться по заранее созданным картам.

**содержание**

|  |  |  |
| --- | --- | --- |
|  | Введение | 5 |
| 1. | Первая практическая работа | 6 |
| 2. | Вторая практическая работа | 34 |
| 3. | Третья практическая работа | 47 |
|  | Заключение | 68 |
|  | Приложение А. Код программы | 69 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

**введение**

Лабораторные работы, объединённые в курсовую работу.

Выполненные работы покрывают следующие разделы:

* Классы.
* Объектно-ориентированная парадигма.
* Принципы объектно-ориентированного программирования.
* Паттерны проектирования.

В процессе выполнения работ были изучены вышеупомянутые разделы, и на основании изученного реализована программа.

Программа представляет собой игру, где можно управлять Университетом, создавать Юнитов (рабочих), управлять ими, взаимодействовать с окружающей средой, передвигаться по заранее созданным картам.

1. **Первая практическая работа**

**Цель работы.**

Разработать и реализовать набор классов:

●     Класс игрового поля

●     Набор классов юнитов

Игровое поле является контейнером для объектов представляющим прямоугольную сетку. Основные требования к классу игрового поля:

●     Создание поля произвольного размера

●     Контроль максимального количества объектов на поле

●     Возможность добавления и удаления объектов на поле

●     Возможность копирования поля (включая объекты на нем)

Юнит является объектов, размещаемым на поля боя. Один юнит представляет собой отряд. Основные требования к классам юнитов:

●     Все юниты должны иметь как минимум один общий интерфейс

●     Реализованы 3 типа юнитов (например, пехота, лучники, конница)

●     Реализованы 2 вида юнитов для каждого типа (например, для пехоты могут быть созданы мечники и копейщики)

●     Юниты имеют характеристики, отражающие их основные атрибуты, такие как здоровье, броня, атака.

●     Юнит имеет возможность перемещаться по карте

**Основные теоретические положения.**

## **Описание класса**

Класс является абстрактным типом данных, определяемым пользователем, и представляет собой модель реального объекта в виде данных и функций для работы с ними.

Данные класса называются полями (по аналогии с полями структуры), а функции класса — методами. Поля и методы называются элементами класса.

Класс может определять переменные и константы для хранения состояния объекта и функции для определения поведения объекта.

## **Описание объектов**

Конкретные переменные типа «класс» называются экземплярами класса, или объектами. Время жизни и видимость объектов зависит от вида и места их описания и подчиняется общим правилам С++:

monstr Vasia; // Объект класса monstr с параметрами по умолчанию

monstr Super(200, 300); // Объект с явной инициализацией

monstr stado[100]; // Массив объектов с параметрами по умолчанию

monstr\* beavis = new monstr (10); // Динамический объект

//(второй параметр задается по умолчанию)

monstr& butthead = Vasia; // Ссылка на объект

При создании каждого объекта выделяется память, достаточная для хранения всех eib полей, и автоматически вызывается конструктор, выполняющий их инициализацию. Методы класса не тиражируются. При выходе объекта из области действия он уничтожается, при этом автоматически вызывается деструктор.

Доступ к элементам объекта аналогичен доступу к полям структуры. Для этого используются операция . (точка) при обращении к элементу через имя объекта и операция -> при обращении через указатель, например:

int ammo = Vasid.getAmmo();

stado[5].draw;

cout << beavis -> getHealth();

## **Указатели на объекты классов**

Рассмотрим пример.

#include <iostream>

#include <string>

using std::string;

using std::cout;

class Hero

{

public:

string name;

int armor;

void move() {

cout << name << " is moving" << '\n';

}

};

int main()

{

Hero hero;

person.name = "Tom";

person.armor = 100;

cout << "Name: " << hero.name << "\tAge: " << hero.age << '\n';

person.move();

return 0;

}

Класс Hero имеет две переменных name и armor, которые предназначены для хранения имени персонажа и количество его брони соответственно. Также класс определяет функцию move, которая выводит строку на консоль. Также стоит обратить внимание на модификатор доступа public:, который указывает, что идущие после него переменные и функции будут доступны извне, из внешнего кода.

На объекты классов, как и на объекты других типов, можно определять указатели. Затем через указатель можно обращаться к членам класса - переменным и методам. Однако если при обращении через обычную переменную используется символ точка, то для для обращения к членам класса через указатель применяется стрелка (->):

#include <iostream>

#include <string>

using std::string;

using std::cout;

class Hero

{

public:

string name;

int armor;

void move() {

cout << name << " is moving" << '\n';

}

};

int main()

{

Hero hero;

Hero \*ptr = &hero;

ptr->name = "Tom";

ptr->armor = 100;

ptr->move();

cout << "Name: " << ptr->name << "\tArmor: " << ptr->armor << '\n';

cout << "Name: " << hero.name << "\tArmor: " << hero.armor << '\n';

return 0;

}

Изменения по указателю ptr в данном случае приведут к изменениям объекта person.

Ниже перечислены правила использования указателей на методы классов.

Указателю на метод можно присваивать только адреса методов, имеющих соответствующий заголовок.

Нельзя определить указатель на статический метод класса.

Нельзя преобразовать указатель па метод в указатель на обычную функцию, не являющуюся элементом класса.

Как и указатели на обычные функции, указатели на методы используются в том случае, когда возникает необходимость вызвать метод, имя которого неизвестно. Однако в отличие указателя на переменную или обычную функцию, указатель на метод не ссылается на определенный адрес памяти. Он больше похож на индекс в массиве, поскольку задает смещение. Конкретный адрес в памяти получается путем сочетания указателя на метод с указателем на определенный объект.

## **Конструкторы**

* Конструктор предназначен для инициализации объекта и вызывается автоматически при его создании. Ниже перечислены основные свойства конструкторов:
* Конструктор не возвращает значение, даже типа void. Нельзя получить указатель на конструктор.
* Класс может иметь несколько конструкторов с разными параметрами для разных видов инициализации (при этом используется механизм перегрузки).
* Конструктор, вызываемый без параметров, называется конструктором по умолчанию.
* Параметры конструктора могут иметь любой тип, кроме этого же класса. Можно задавать значения параметров по умолчанию. Их может содержать только один из конструкторов.
* Если программист не указал ни одного конструктора, компилятор создает его автоматически. Такой конструктор вызывает конструкторы по умолчанию для полей класса и конструкторы по умолчанию базовых классов. В случае, когда класс содержит константы или ссылки, при попытке создания объекта класса будет выдана ошибка, поскольку их необходимо инициализировать конкретными значениями, а конструктор по умолчанию этого делать не умеет.
* Конструкторы не наследуются.
* Конструкторы нельзя описывать с модификаторами const, virtual и static.
* Конструкторы глобальных объектов вызываются до вызова функции main. Локальные объекты создаются, как только становится активной область их действия. Конструктор запускается и при создании временного объекта (например, при передаче объекта из функции).

## **Статические элементы класса**

С помощью модификатора static можно описать статические поля и методы класса. Их можно рассматривать как глобальные переменные или функции, доступные только в пределах области класса.

## **Статические поля**

Статические поля применяются для хранения данных, общих для всех объектов класса, например, количества объектов или ссылки на разделяемый всеми объектами ресурс. Эти поля существуют для всех объектов класса в единственном экземпляре, то есть не дублируются.

## **Статические методы**

Статические методы предназначены для обращения к статическим полям класса. Они могут обращаться непосредственно только к статическим полям и вызывать только другие статические методы класса, потому что им не передается скрытый указатель this. Обращение к статическим методам производится так же, как к статическим полям — либо через имя класса, либо, если хотя бы один объект класса уже создан, через имя объекта.

# **ООП**

## **Абстракция**

История телефонов превышает 100 лет и современный гаджеты, в отличие от своего предшественника из 19 века, представляет собой куда более сложное устройство.

Когда мы пользуемся телефоном, то не задумываемся о его устройстве и процессах, происходящих внутри него. Мы просто используем функции, предоставленные разработчиками телефона — кнопки или сенсорный экран для выбора номера и совершения вызовов.

Одним из первых интерфейсов телефона была рукоятка, которую нужно было вращать, чтобы сделать вызов. Разумеется, это было не очень удобно. Тем не менее, свою функцию рукоять исправно выполняла.

Если посмотреть на самый современный и на самый первый телефон, можно сразу выделить самые важные детали, которые важны и для устройства конца 19-го века, и для суперсовременного смартфона. Это совершение вызова (набор номера) и приём вызова. По сути это то, что делает телефон телефоном, а не чем-то другим. Сейчас мы применили принцип в ООП — выделение наиболее важных характеристик и информации об объекте. Этот принцип ООП называется абстракцией.

Абстракцию в ООП можно также определить, как способ представления элементов задачи из реального мира в виде объектов в программе. Абстракция всегда связана с обобщением некоторой информации о свойствах предметов или объектов, поэтому главное — это отделить значимую информацию от незначимой в контексте решаемой задачи. При этом уровней абстракции может быть несколько.

## **Инкапсуляция**

Инкапсуляция упоминается как часть определения ООП потому, что языки ООП поддерживают простой и эффективный способ инкапсуляции данных и функций. Как результат, есть возможность очертить круг связанных данных и функций. За пределами круга эти данные невидимы и доступны только некоторые функции. Воплощение этого понятия можно наблюдать в виде приватных членов данных и общедоступных членов-функций класса.

С помощью абстракции мы выделяем общее для всех объектов. Однако каждая модель телефона — индивидуальна и чем-то отличается от других. Как же нам в программе провести границы и обозначить эту индивидуальность?

Как сделать так, чтоб никто из пользователей случайно или преднамеренно не смог сломать наш телефон, или попытаться переделать одну модель в другую? Для мира реальных объектов ответ очевиден: нужно поместить все детали в корпус телефона. Ведь если этого не сделать и оставить все внутренности телефона и провода, соединяющие их снаружи, обязательно найдется любознательный экспериментатор, который захочет “улучшить” работу нашего телефона. Для исключения подобного вмешательства в конструкцию и работу объекта **в ООП используют принцип инкапсуляции** – еще один базовый принцип ООП, при котором атрибуты и поведение объекта объединяются в одном классе, внутренняя реализация объекта скрывается от пользователя, а для работы с объектом предоставляется открытый интерфейс.

Задача программиста — определить, какие атрибуты и методы будут доступны для открытого доступа, а какие являются внутренней реализацией объекта и должны быть недоступны для изменений.

В мире C++ состояние будущих объектов описывается в классе с помощью полей, а их поведение – с помощью методов. Возможность же изменения состояния и поведения осуществляется с помощью модификаторов доступа к полям и методам – private, protected, public, а также default (доступ по умолчанию).

Модификатор private делает доступными поля и методы класса только внутри данного класса. Это означает, что получить доступ к private полям из вне невозможно, как и нет возможности вызвать private методы.

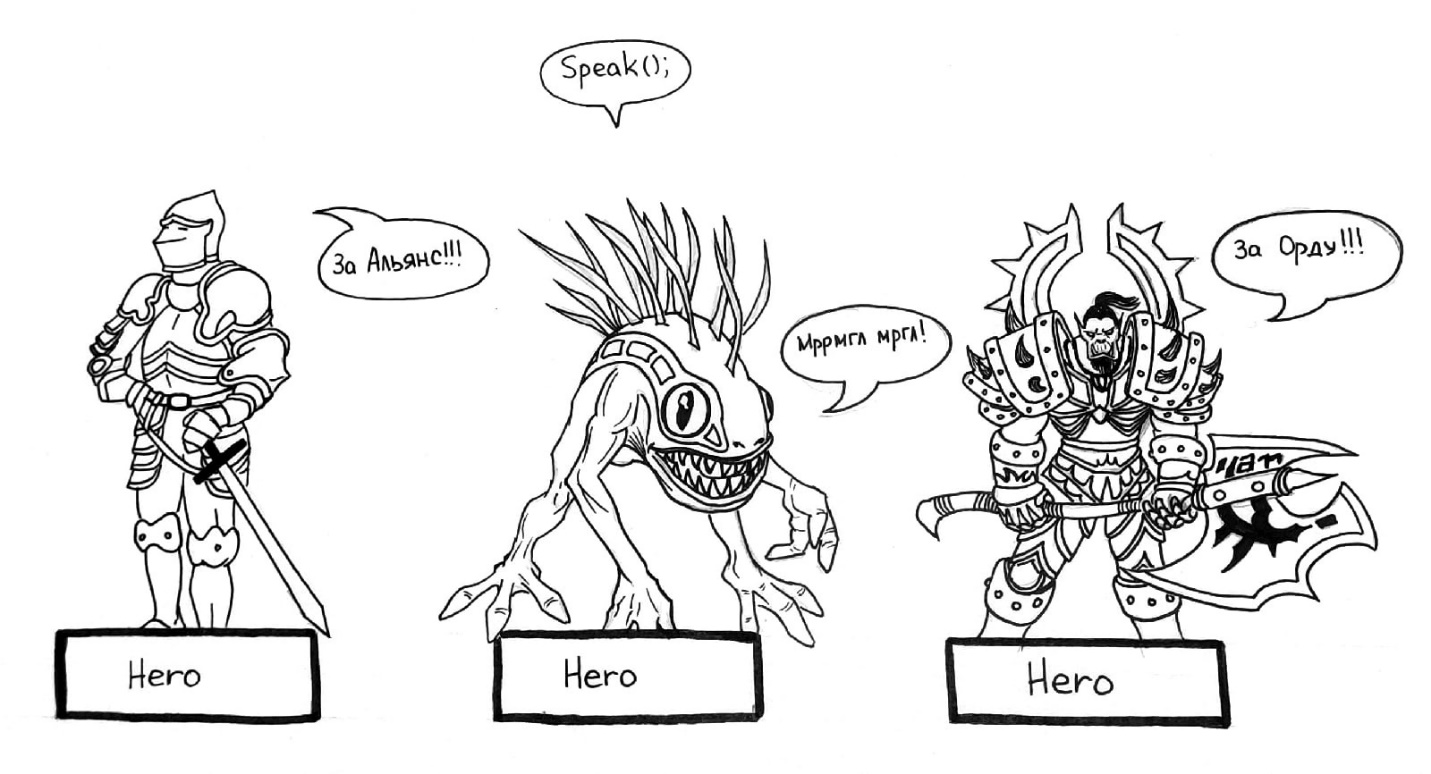
## **Наследование**

Механизм наследования классов позволяет строить иерархии, в которых производные классы получают элементы родительских, или базовых, классов и могут дополнять их или изменять их свойства. При большом количестве никак не связанных классов управлять ими становится невозможным. Наследование позволяет справиться с этой проблемой путем упорядочивания и ранжирования классов, то есть объединения общих для нескольких классов свойств в одном классе и использования его в качестве базового.

Классы, находящиеся ближе к началу иерархии, объединяют в себе наиболее общие черты для всех нижележащих классов. По мере продвижения вниз по иерархии классы приобретают все больше конкретных черт. Множественное наследование позволяет одному классу обладать свойствами двух и более родительских классов.

## **Полиморфизм**

Полиморфизм – это способность программы идентично использовать объекты с одинаковым интерфейсом без информации о конкретном типе этого объекта. Если вы ответите на вопрос, что такое полиморфизм, таким образом, вас, скорее всего, попросят объяснить, что вы имели ввиду. Лишний раз, не напрашиваясь на кучу дополнительных вопросов, разложите интервьюеру все по полочкам. Тесно связано с наследованием.



Поскольку С++ относится к строго типизированным языкам, в программном коде всегда нужно указать тип объекта при объявлении переменных. К этому добавьте, что строгая типизация повышает безопасность кода, и надежность программы и позволяет еще на стадии компиляции предотвратить ошибки несовместимости типов (например, попытку разделить строку на число).

В C++ каждая виртуальная функция в классе представлена указателем в таблице виртуальных методов vtable и все вызовы виртуальных функций выполняются через эту таблицу. Конструкторы производных классов просто инициализируют таблицу vtable объекта указателями на свои версии функций.

Суть полиморфизма заключается в применении указателей на функции. Программисты использовали указатели на функции для достижения полиморфного поведения еще со времен появления архитектуры фон Неймана в конце 1940-х годов. Иными словами, парадигма ООП не принесла ничего нового.

Впрочем, это не совсем верно. Пусть полиморфизм появился раньше языков ООП, но они сделали его намного надежнее и удобнее.

Проблема явного использования указателей на функции для создания полиморфного поведения в том, что указатели на функции по своей природе опасны. Такое их применение оговаривается множеством соглашений. Вы должны помнить об этих соглашениях и инициализировать указатели. Вы должны помнить об этих соглашениях и вызывать функции посредством указателей. Если какой-то программист забудет о соглашениях, возникшую в результате ошибку будет чертовски трудно отыскать и устранить.

Языки ООП избавляют от необходимости помнить об этих соглашениях и, соответственно, устраняют опасности, связанные с этим. Поддержка полиморфизма на уровне языка делает его использование тривиально простым. Это обстоятельство открывает новые возможности, о которых программисты на С могли только мечтать. Отсюда можно заключить, что ООП накладывает ограничение на косвенную передачу управления.

## Ключи доступа

При описании класса в его заголовке перечисляются все классы, являющиеся для него базовыми. Возможность обращения к элементам этих классов регулируется с помощью ключей доступа private, protected и public:

class имя : [private | protected | public] базовыйКласс

{ тело класса };

Если базовых классов несколько, они перечисляются через запятую. Ключ доступа может стоять перед каждым классом, например:

class А { ... };

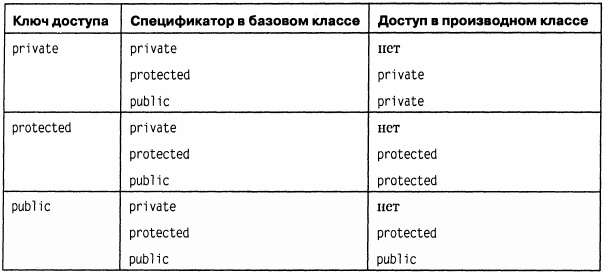
class В { ... };

class С { ... };

class D: А, protected В, public С { ... }:

По умолчанию для классов используется ключ доступа private, а для структур — public.

До сих пор мы рассматривали только применяемые к элементам класса спецификаторы доступа private и public. Для любого элемента класса может также использоваться спецификатор protected, который для одиночных классов, не входящих в иерархию, равносилен private. Разница между ними проявляется при наследовании, что можно видеть из приведенной таблицы:



Как видно из таблицы, private элементы базового класса в производном классе недоступны вне зависимости от ключа. Обращение к ним может осуществляться только через методы базового класса.

Элементы protected при наследовании с ключом private становятся в производном классе private, в остальных случаях права доступа к ним не изменяются.

Доступ к элементам public при наследовании становится соответствующим ключу доступа.

## Абстрактные классы

Класс, содержащий хотя бы один чисто виртуальный метод, называется абстрактным. Абстрактные классы предназначены для представления общих понятий, которые предполагается конкретизировать в производных классах. Абстрактный класс может использоваться только в качестве базового для других классов — объекты абстрактного класса создавать нельзя, поскольку прямой или косвенный вызов чисто виртуального метода приводит к ошибке при выполнении.

При определении абстрактного класса необходимо иметь в виду следующее:

* абстрактный класс нельзя использовать при явном приведении типов, для описания типа параметра и типа возвращаемого функцией значения;
* допускается объявлять указатели и ссылки на абстрактный класс, если при инициализации не требуется создавать временный объект;
* если класс, производный от абстрактного, не определяет все чисто виртуальные функции, он также является абстрактным.

Таким образом, можно создать функцию, параметром которой является указатель на абстрактный класс. На место этого параметра при выполнении программы может передаваться указатель на объект любого производного класса. Это позволяет создавать полиморфные функции^ работающие с объектом любого типа в пределах одной иерархии.

# **Паттерны проектирования**

## Фабричный метод

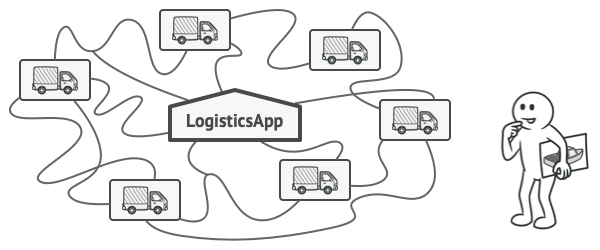
**Фабричный метод** — это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.



### 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 Грузовик.

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

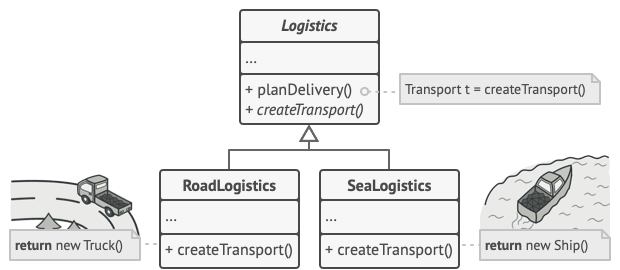


Отличные новости, правда?! Но как насчёт кода? Большая часть существующего кода жёстко привязана к классам Грузовиков. Чтобы добавить в программу классы морских Судов, понадобится перелопатить всю программу. Более того, если вы потом решите добавить в программу ещё один вид транспорта, то всю эту работу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

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

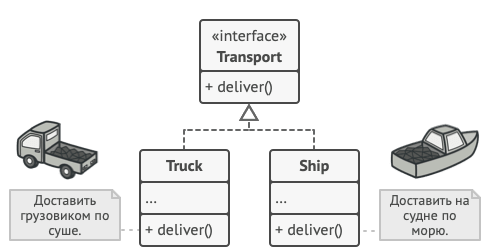
### Решение

Паттерн Фабричный метод предлагает создавать объекты не напрямую, используя оператор new, а через вызов особого фабричного метода. Не пугайтесь, объекты всё равно будут создаваться при помощи new, но делать это будет фабричный метод.

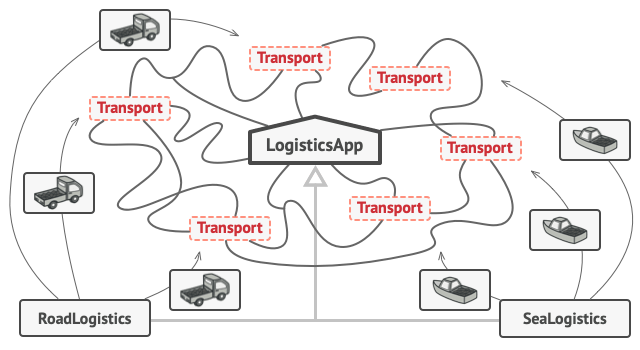


На первый взгляд, это может показаться бессмысленным: мы просто переместили вызов конструктора из одного конца программы в другой. Но теперь вы сможете переопределить фабричный метод в подклассе, чтобы изменить тип создаваемого продукта.

Чтобы эта система заработала, все возвращаемые объекты должны иметь общий интерфейс. Подклассы смогут производить объекты различных классов, следующих одному и тому же интерфейсу.

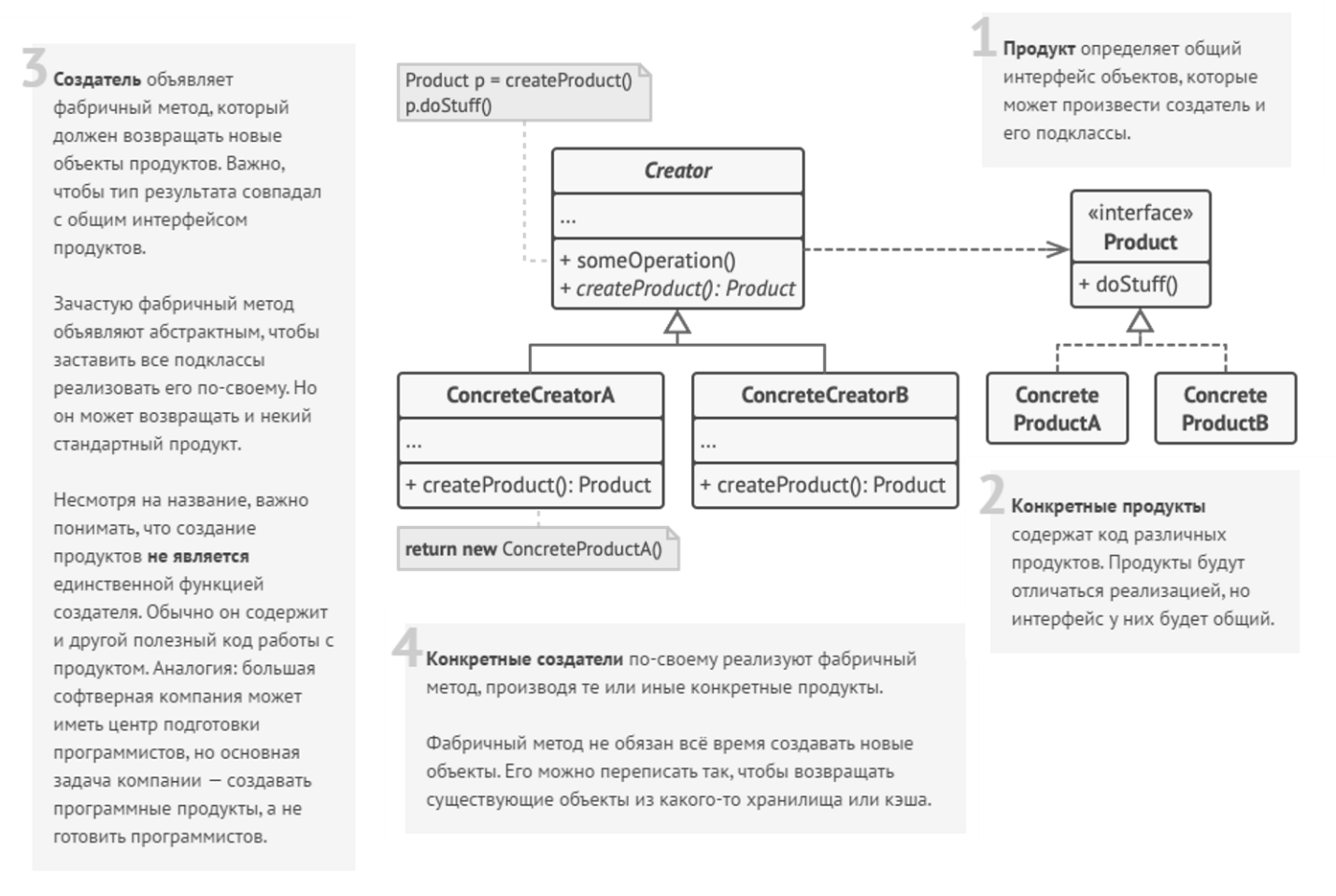


 Например, классы Грузовик и Судно реализуют интерфейс Транспорт с методом доставить. Каждый из этих классов реализует метод по-своему: грузовики везут грузы по земле, а суда — по морю. Фабричный метод в классе ДорожнойЛогистики вернёт объект-грузовик, а класс МорскойЛогистики — объект-судно.



 Для клиента фабричного метода нет разницы между этими объектами, так как он будет трактовать их как некий абстрактный Транспорт. Для него будет важно, чтобы объект имел метод доставить, а как конкретно он работает — не важно.

### Структура



### Применимость

 Когда заранее неизвестны типы и зависимости объектов, с которыми должен работать ваш код.

Фабричный метод отделяет код производства продуктов от остального кода, который эти продукты использует.

Благодаря этому, код производства можно расширять, не трогая основной. Так, чтобы добавить поддержку нового продукта, вам нужно создать новый подкласс и определить в нём фабричный метод, возвращая оттуда экземпляр нового продукта.

Когда вы хотите дать возможность пользователям расширять части вашего фреймворка или библиотеки.

Пользователи могут расширять классы вашего фреймворка через наследование. Но как сделать так, чтобы фреймворк создавал объекты из этих новых классов, а не из стандартных?

Решением будет дать пользователям возможность расширять не только желаемые компоненты, но и классы, которые создают эти компоненты. А для этого создающие классы должны иметь конкретные создающие методы, которые можно определить.

Например, вы используете готовый UI-фреймворк для своего приложения. Но вот беда — требуется иметь круглые кнопки, вместо стандартных прямоугольных. Вы создаёте класс RoundButton. Но как сказать главному классу фреймворка UIFramework, чтобы он теперь создавал круглые кнопки, вместо стандартных?

Для этого вы создаёте подкласс UIWithRoundButtons из базового класса фреймворка, переопределяете в нём метод создания кнопки (а-ля createButton) и вписываете туда создание своего класса кнопок. Затем используете UIWithRoundButtons вместо стандартного UIFramework.

Когда вы хотите экономить системные ресурсы, повторно используя уже созданные объекты, вместо порождения новых.

Такая проблема обычно возникает при работе с тяжёлыми ресурсоёмкими объектами, такими, как подключение к базе данных, файловой системе и т. д.

Представьте, сколько действий вам нужно совершить, чтобы повторно использовать существующие объекты:

1. Сначала вам следует создать общее хранилище, чтобы хранить в нём все создаваемые объекты.
2. При запросе нового объекта нужно будет заглянуть в хранилище и проверить, есть ли там неиспользуемый объект.
3. А затем вернуть его клиентскому коду.
4. Но если свободных объектов нет — создать новый, не забыв добавить его в хранилище.

Весь этот код нужно куда-то поместить, чтобы не засорять клиентский код.

Самым удобным местом был бы конструктор объекта, ведь все эти проверки нужны только при создании объектов. Но, увы, конструктор всегда создаёт **новые** объекты, он не может вернуть существующий экземпляр.

Значит, нужен другой метод, который бы отдавал как существующие, так и новые объекты. Им и станет фабричный метод.

### Шаги реализации

1. Приведите все создаваемые продукты к общему интерфейсу.
2. В классе, который производит продукты, создайте пустой фабричный метод. В качестве возвращаемого типа укажите общий интерфейс продукта.
3. Затем пройдитесь по коду класса и найдите все участки, создающие продукты. Поочерёдно замените эти участки вызовами фабричного метода, перенося в него код создания различных продуктов.

В фабричный метод, возможно, придётся добавить несколько параметров, контролирующих, какой из продуктов нужно создать.

На этом этапе фабричный метод, скорее всего, будет выглядеть удручающе. В нём будет жить большой условный оператор, выбирающий класс создаваемого продукта. Но не волнуйтесь, мы вот-вот исправим это.

1. Для каждого типа продуктов заведите подкласс и переопределите в нём фабричный метод. Переместите туда код создания соответствующего продукта из суперкласса.
2. Если создаваемых продуктов слишком много для существующих подклассов создателя, вы можете подумать о введении параметров в фабричный метод, которые позволят возвращать различные продукты в пределах одного подкласса.

Например, у вас есть класс Почта с подклассами АвиаПочта и НаземнаяПочта, а также классы продуктов Самолёт, Грузовик и Поезд. Авиа соответствует Самолётам, но для НаземнойПочты есть сразу два продукта. Вы могли бы создать новый подкласс почты для поездов, но проблему можно решить и по-другому. Клиентский код может передавать в фабричный метод НаземнойПочты аргумент, контролирующий тип создаваемого продукта.

1. Если после всех перемещений фабричный метод стал пустым, можете сделать его абстрактным. Если в нём что-то осталось — не беда, это будет его реализацией по умолчанию.

### Преимущества и недостатки



### Отношения с другими паттернами

Многие архитектуры начинаются с применения Фабричного метода (более простого и расширяемого через подклассы) и эволюционируют в сторону Абстрактной фабрики, Прототипа или Строителя (более гибких, но и более сложных).

Классы Абстрактной фабрики чаще всего реализуются с помощью Фабричного метода, хотя они могут быть построены и на основе Прототипа.

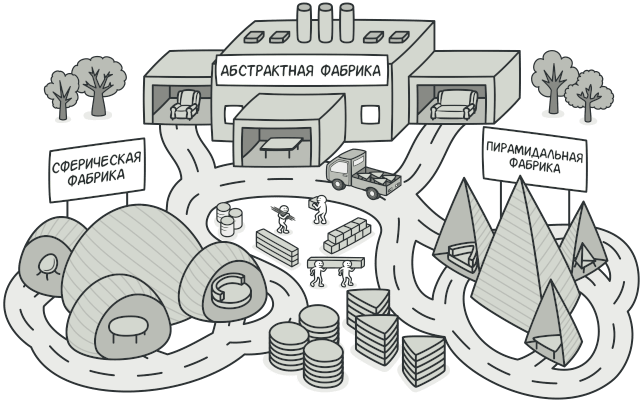
Фабричный метод можно использовать вместе с Итератором, чтобы подклассы коллекций могли создавать подходящие им итераторы.

Прототип не опирается на наследование, но ему нужна сложная операция инициализации. Фабричный метод, наоборот, построен на наследовании, но не требует сложной инициализации.

Фабричный метод можно рассматривать как частный случай Шаблонного метода. Кроме того, Фабричный метод нередко бывает частью большого класса с Шаблонными методами.

## Абстрактная фабрика

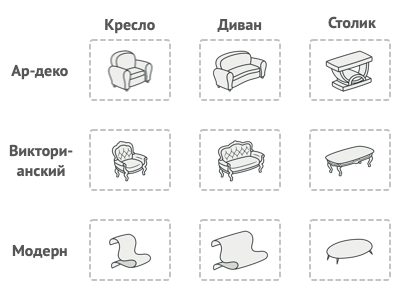
**Абстрактная фабрика** — это порождающий паттерн проектирования, который позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.



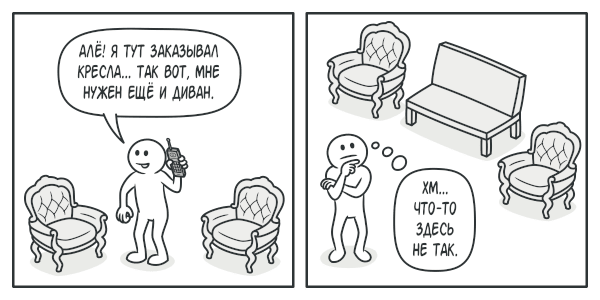
### Проблема

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

1. Семейство зависимых продуктов. Скажем, Кресло + Диван + Столик.
2. Несколько вариаций этого семейства. Например, продукты Кресло, Диван и Столик представлены в трёх разных стилях: Ар-деко, Викторианском и Модерне.



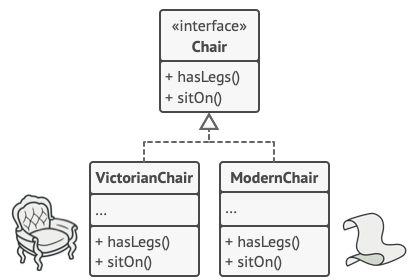
Вам нужен такой способ создавать объекты продуктов, чтобы они сочетались с другими продуктами того же семейства. Это важно, так как клиенты расстраиваются, если получают несочетающуюся мебель.



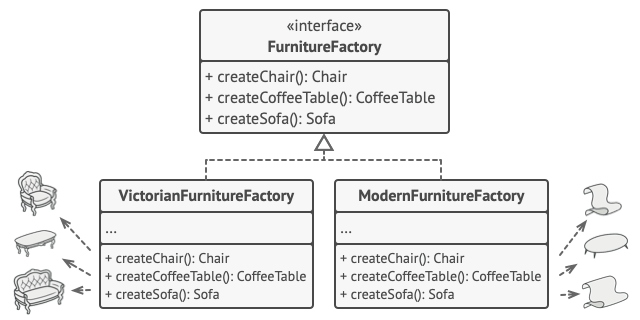
Кроме того, вы не хотите вносить изменения в существующий код при добавлении новых продуктов или семейств в программу. Поставщики часто обновляют свои каталоги, и вы бы не хотели менять уже написанный код каждый раз при получении новых моделей мебели.

### 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.

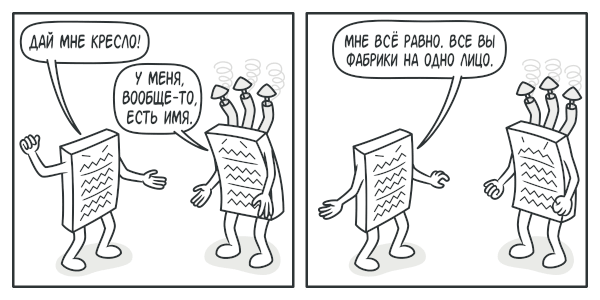


Далее вы создаёте абстрактную фабрику — общий интерфейс, который содержит методы создания всех продуктов семейства (например, создатьКресло, создатьДиван и создатьСтолик). Эти операции должны возвращать **абстрактные** типы продуктов, представленные интерфейсами, которые мы выделили ранее — Кресла, Диваны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

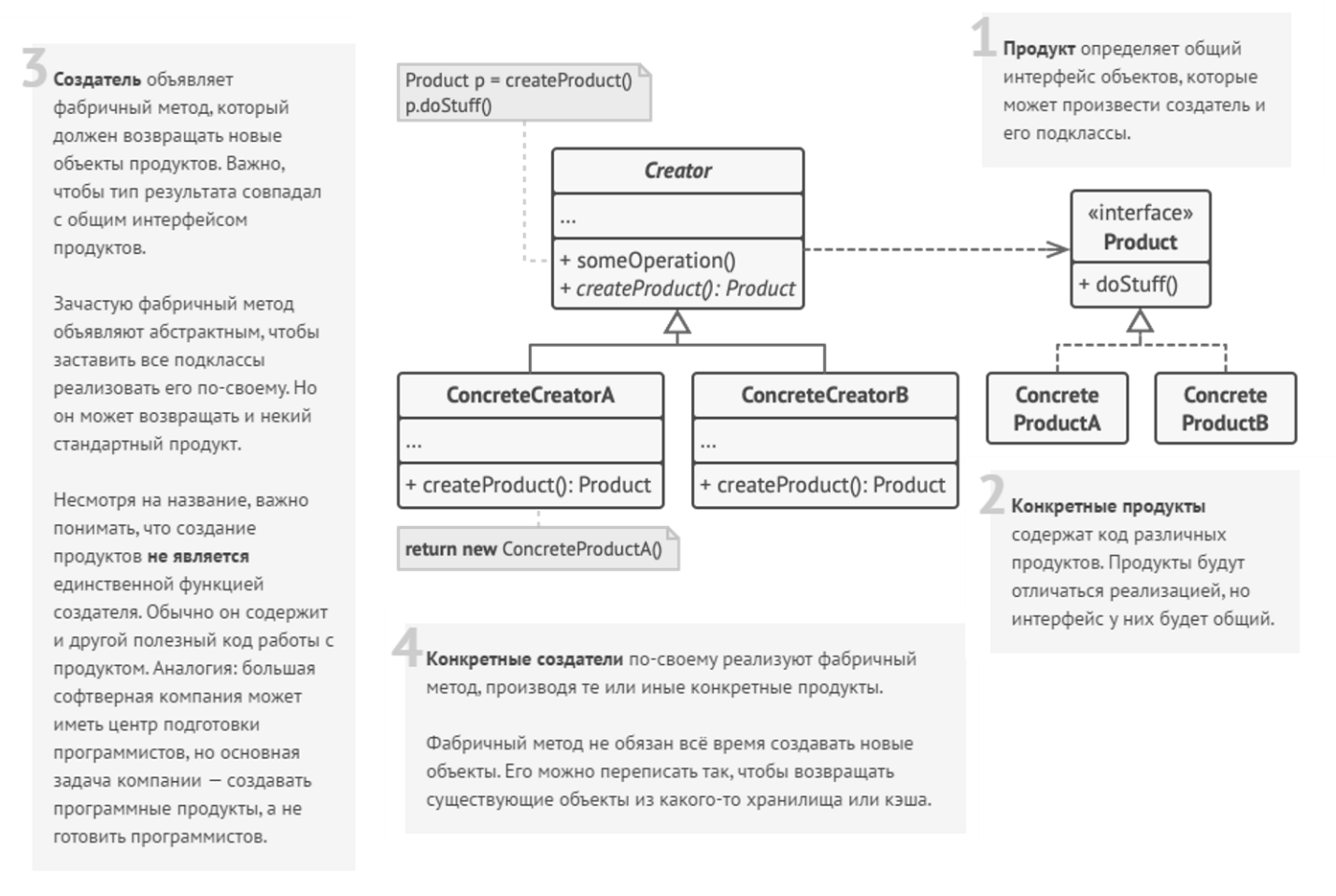
Клиентский код должен работать как с фабриками, так и с продуктами только через их общие интерфейсы. Это позволит подавать в ваши классы любой тип фабрики и производить любые продукты, ничего не ломая.



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.

### Структура



## Применимость

Когда бизнес-логика программы должна работать с разными видами связанных друг с другом продуктов, не завися от конкретных классов продуктов.

Абстрактная фабрика скрывает от клиентского кода подробности того, как и какие конкретно объекты будут созданы. Но при этом клиентский код может работать со всеми типами создаваемых продуктов, поскольку их общий интерфейс был заранее определён.

Когда в программе уже используется Фабричный метод, но очередные изменения предполагают введение новых типов продуктов.

 В хорошей программе каждый класс отвечает только за одну вещь. Если класс имеет слишком много фабричных методов, они способны затуманить его основную функцию. Поэтому имеет смысл вынести всю логику создания продуктов в отдельную иерархию классов, применив абстрактную фабрику.

## Шаги реализации

1. Создайте таблицу соотношений типов продуктов к вариациям семейств продуктов.
2. Сведите все вариации продуктов к общим интерфейсам.
3. Определите интерфейс абстрактной фабрики. Он должен иметь фабричные методы для создания каждого из типов продуктов.
4. Создайте классы конкретных фабрик, реализовав интерфейс абстрактной фабрики. Этих классов должно быть столько же, сколько и вариаций семейств продуктов.
5. Измените код инициализации программы так, чтобы она создавала определённую фабрику и передавала её в клиентский код.
6. Замените в клиентском коде участки создания продуктов через конструктор вызовами соответствующих методов фабрики.

### Преимущества и недостатки

### Отношения с другими паттернами

Многие архитектуры начинаются с применения Фабричного метода (более простого и расширяемого через подклассы) и эволюционируют в сторону Абстрактной фабрики, Прототипа или Строителя (более гибких, но и более сложных).

Строитель концентрируется на построении сложных объектов шаг за шагом. Абстрактная фабрика специализируется на создании семейств связанных продуктов. Строитель возвращает продукт только после выполнения всех шагов, а Абстрактная фабрика возвращает продукт сразу же.

Классы Абстрактной фабрики чаще всего реализуются с помощью Фабричного метода, хотя они могут быть построены и на основе Прототипа.

Абстрактная фабрика может быть использована вместо Фасада для того, чтобы скрыть платформо-зависимые классы.

Абстрактная фабрика может работать совместно с Мостом. Это особенно полезно, если у вас есть абстракции, которые могут работать только с некоторыми из реализаций. В этом случае фабрика будет определять типы создаваемых абстракций и реализаций.

Абстрактная фабрика, Строитель и Прототип могут быть реализованы при помощи Одиночки.

1. **Вторая практическая работа**

**Цель работы.**

Разработать и реализовать набор классов:

● Класс базы

● Набор классов ландшафта карты

● Набор классов нейтральных объектов поля

Класс базы должен отвечать за создание юнитов, а также учитывать юнитов, относящихся к текущей базе. Основные требования к классу база:

● База должна размещаться на поле

● Методы для создания юнитов

● Учет юнитов, и реакция на их уничтожение и создание

● База должна обладать характеристиками такими, как здоровье, максимальное количество юнитов, которые могут быть одновременно созданы на базе, и.т.д.

Набор классов ландшафта определяют вид поля. Основные требования к классам ландшафта:

Должно быть создано минимум 3 типа ландшафта

● Все классы ландшафта должны иметь как минимум один интерфейс

● Ландшафт должен влиять на юнитов (например, возможно пройти по клетке с определенным ландшафтом или запрет для атаки определенного типа юнитов)

● На каждой клетке поля должен быть определенный тип ландшафта

Набор классов нейтральных объектов представляют объекты, располагаемые на поле и с которыми могут взаимодействие юнитов. Основные требования к классам нейтральных объектов поля:

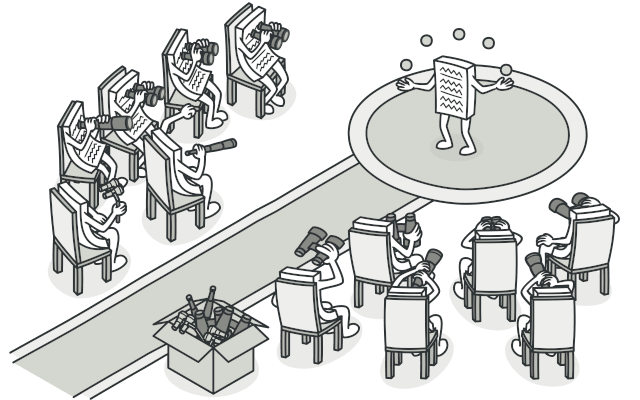
● Создано не менее 4 типов нейтральных объектов

● Взаимодействие юнитов с нейтральными объектами, должно быть реализовано в виде перегрузки операций

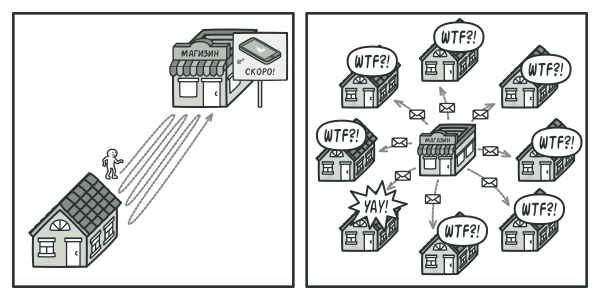
● Классы нейтральных объектов должны иметь как минимум один общий интерфейс

**Основные теоретические положения.**

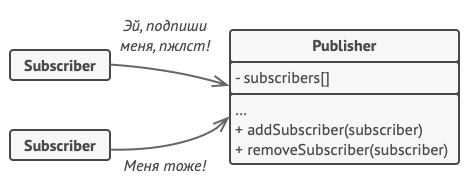
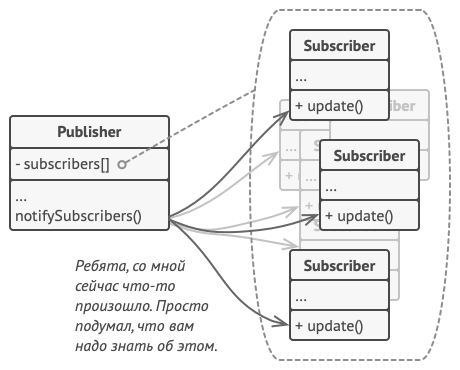
Наблюдатель

* **Наблюдатель** — это поведенческий паттерн проектирования, который создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.
* 

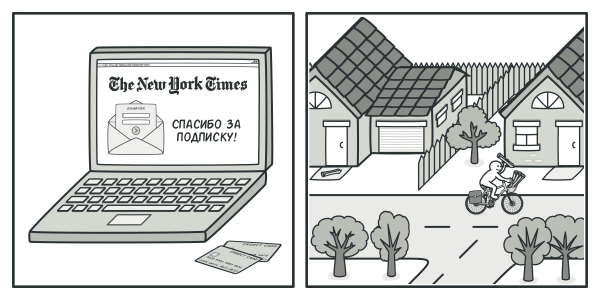
## Проблема

* Представьте, что вы имеете два объекта: Покупатель и Магазин. В магазин вот-вот должны завезти новый товар, который интересен покупателю.
* Покупатель может каждый день ходить в магазин, чтобы проверить наличие товара. Но при этом он будет злиться, без толку тратя своё драгоценное время.
* 
* другой стороны, магазин может разослать спам каждому своему покупателю. Многих это расстроит, так как товар специфический, и не всем он нужен.
* Получается конфликт: либо покупатель тратит время на периодические проверки, либо магазин тратит ресурсы на бесполезные оповещения.

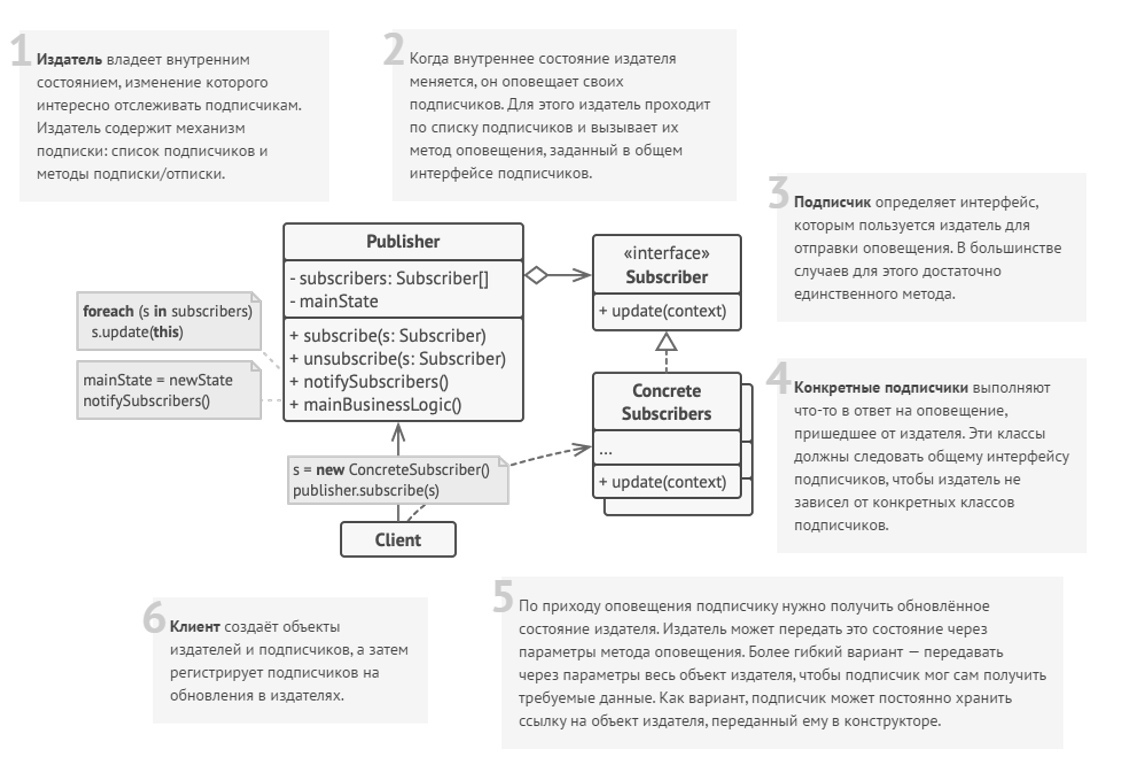
### Решение

* Давайте называть Издателями те объекты, которые содержат важное или интересное для других состояние. Остальные объекты, которые хотят отслеживать изменения этого состояния, назовём Подписчиками.
* Паттерн Наблюдатель предлагает хранить внутри объекта издателя список ссылок на объекты подписчиков, причём издатель не должен вести список подписки самостоятельно. Он предоставит методы, с помощью которых подписчики могли бы добавлять или убирать себя из списка.
* 
* Теперь самое интересное. Когда в издателе будет происходить важное событие, он будет проходиться по списку подписчиков и оповещать их об этом, вызывая определённый метод объектов-подписчиков.
* Издателю безразлично, какой класс будет иметь тот или иной подписчик, так как все они должны следовать общему интерфейсу и иметь единый метод оповещения.
* 
* Увидев, как складно всё работает, вы можете выделить общий интерфейс, описывающий методы подписки и отписки, и для всех издателей. После этого подписчики смогут работать с разными типами издателей, а также получать оповещения от них через один и тот же метод.

### Аналогия из жизни

* 
* После того как вы оформили подписку на газету или журнал, вам больше не нужно ездить в супермаркет и проверять, не вышел ли очередной номер. Вместо этого издательство будет присылать новые номера по почте прямо к вам домой сразу после их выхода.
* Издательство ведёт список подписчиков и знает, кому какой журнал высылать. Вы можете в любой момент отказаться от подписки, и журнал перестанет вам приходить.

### Структура

* 

### Применимость

* Когда после изменения состояния одного объекта требуется что-то сделать в других, но вы не знаете наперёд, какие именно объекты должны отреагировать.
* Описанная проблема может возникнуть при разработке библиотек пользовательского интерфейса, когда вам надо дать возможность сторонним классам реагировать на клики по кнопкам.
* Паттерн Наблюдатель позволяет любому объекту с интерфейсом подписчика зарегистрироваться на получение оповещений о событиях, происходящих в объектах-издателях.
* Когда одни объекты должны наблюдать за другими, но только в определённых случаях.
* Издатели ведут динамические списки. Все наблюдатели могут подписываться или отписываться от получения оповещений прямо во время выполнения программы.

### Шаги реализации

1. Разбейте вашу функциональность на две части: независимое ядро и опциональные зависимые части. Независимое ядро станет издателем. Зависимые части станут подписчиками.
2. Создайте интерфейс подписчиков. Обычно в нём достаточно определить единственный метод оповещения.
3. Создайте интерфейс издателей и опишите в нём операции управления подпиской. Помните, что издатель должен работать только с общим интерфейсом подписчиков.
4. Вам нужно решить, куда поместить код ведения подписки, ведь он обычно бывает одинаков для всех типов издателей. Самый очевидный способ — вынести этот код в промежуточный абстрактный класс, от которого будут наследоваться все издатели.

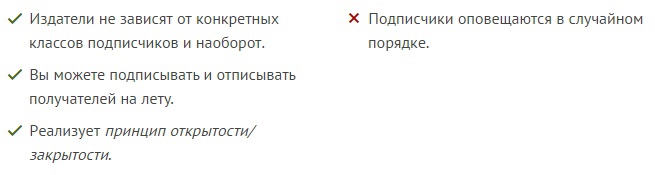
* Но если вы интегрируете паттерн в существующие классы, то создать новый базовый класс может быть затруднительно. В этом случае вы можете поместить логику подписки во вспомогательный объект и делегировать ему работу из издателей.

1. Создайте классы конкретных издателей. Реализуйте их так, чтобы после каждого изменения состояния они отправляли оповещения всем своим подписчикам.
2. Реализуйте метод оповещения в конкретных подписчиках. Не забудьте предусмотреть параметры, через которые издатель мог бы отправлять какие-то данные, связанные с происшедшим событием.

* Возможен и другой вариант, когда подписчик, получив оповещение, сам возьмёт из объекта издателя нужные данные. Но в этом случае вы будете вынуждены привязать класс подписчика к конкретному классу издателя.

1. Клиент должен создавать необходимое количество объектов подписчиков и подписывать их у издателей.

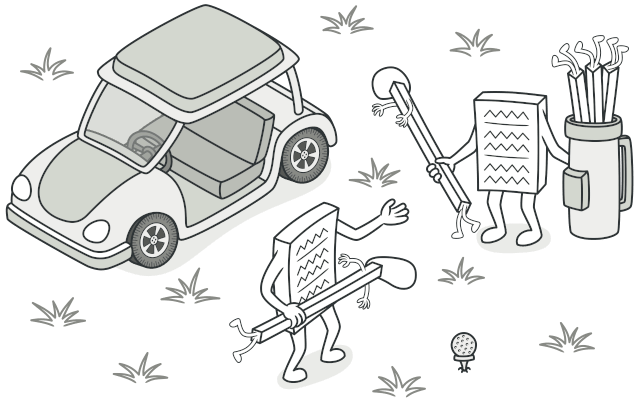
### Преимущества и недостатки

* 

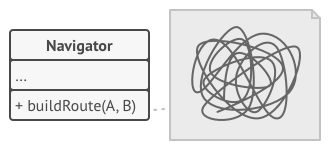
### Отношения с другими паттернами

* Цепочка обязанностей, Команда, Посредник и Наблюдатель показывают различные способы работы отправителей запросов с их получателями:
  + Цепочка обязанностей передаёт запрос последовательно через цепочку потенциальных получателей, ожидая, что какой-то из них обработает запрос.
  + Команда устанавливает косвенную одностороннюю связь от отправителей к получателям.
  + Посредник убирает прямую связь между отправителями и получателями, заставляя их общаться опосредованно, через себя.
  + Наблюдатель передаёт запрос одновременно всем заинтересованным получателям, но позволяет им динамически подписываться или отписываться от таких оповещений.
* Разница между Посредником и Наблюдателем не всегда очевидна. Чаще всего они выступают как конкуренты, но иногда могут работать вместе.
* Цель Посредника — убрать обоюдные зависимости между компонентами системы. Вместо этого они становятся зависимыми от самого посредника. С другой стороны, цель Наблюдателя — обеспечить динамическую одностороннюю связь, в которой одни объекты косвенно зависят от других.
* Довольно популярна реализация Посредника при помощи Наблюдателя. При этом объект посредника будет выступать издателем, а все остальные компоненты станут подписчиками и смогут динамически следить за событиями, происходящими в посреднике. В этом случае трудно понять, чем же отличаются оба паттерна.
* Но Посредник имеет и другие реализации, когда отдельные компоненты жёстко привязаны к объекту посредника. Такой код вряд ли будет напоминать Наблюдателя, но всё же останется Посредником.
* Напротив, в случае реализации посредника с помощью Наблюдателя представим такую программу, в которой каждый компонент системы становится издателем. Компоненты могут подписываться друг на друга, в то же время не привязываясь к конкретным классам. Программа будет состоять из целой сети Наблюдателей, не имея центрального объекта-Посредника.

Стратегия

* **Стратегия** — это поведенческий паттерн проектирования, который определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы.
* 

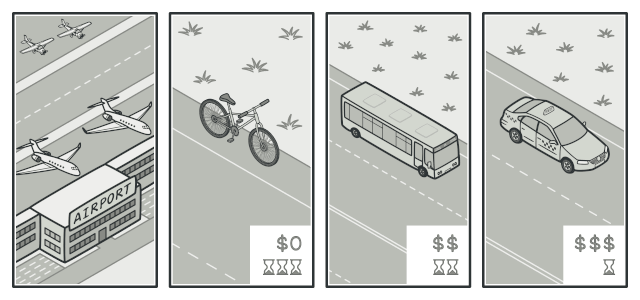
### Проблема

* Вы решили написать приложение-навигатор для путешественников. Оно должно показывать красивую и удобную карту, позволяющую с лёгкостью ориентироваться в незнакомом городе.
* Одной из самых востребованных функций являлся поиск и прокладывание маршрутов. Пребывая в неизвестном ему городе, пользователь должен иметь возможность указать начальную точку и пункт назначения, а навигатор — проложит оптимальный путь.
* Первая версия вашего навигатора могла прокладывать маршрут лишь по дорогам, поэтому отлично подходила для путешествий на автомобиле. Но, очевидно, не все ездят в отпуск на машине. Поэтому следующим шагом вы добавили в навигатор прокладывание пеших маршрутов.
* Через некоторое время выяснилось, что некоторые люди предпочитают ездить по городу на общественном транспорте. Поэтому вы добавили и такую опцию прокладывания пути.
* Но и это ещё не всё. В ближайшей перспективе вы хотели бы добавить прокладывание маршрутов по велодорожкам. А в отдалённом будущем — интересные маршруты посещения достопримечательностей.
* 
* Если с популярностью навигатора не было никаких проблем, то техническая часть вызывала вопросы и периодическую головную боль. С каждым новым алгоритмом код основного класса навигатора увеличивался вдвое. В таком большом классе стало довольно трудно ориентироваться.
* Любое изменение алгоритмов поиска, будь то исправление багов или добавление нового алгоритма, затрагивало основной класс. Это повышало риск сделать ошибку, случайно задев остальной работающий код.
* Кроме того, осложнялась командная работа с другими программистами, которых вы наняли после успешного релиза навигатора. Ваши изменения нередко затрагивали один и тот же код, создавая конфликты, которые требовали дополнительного времени на их разрешение.

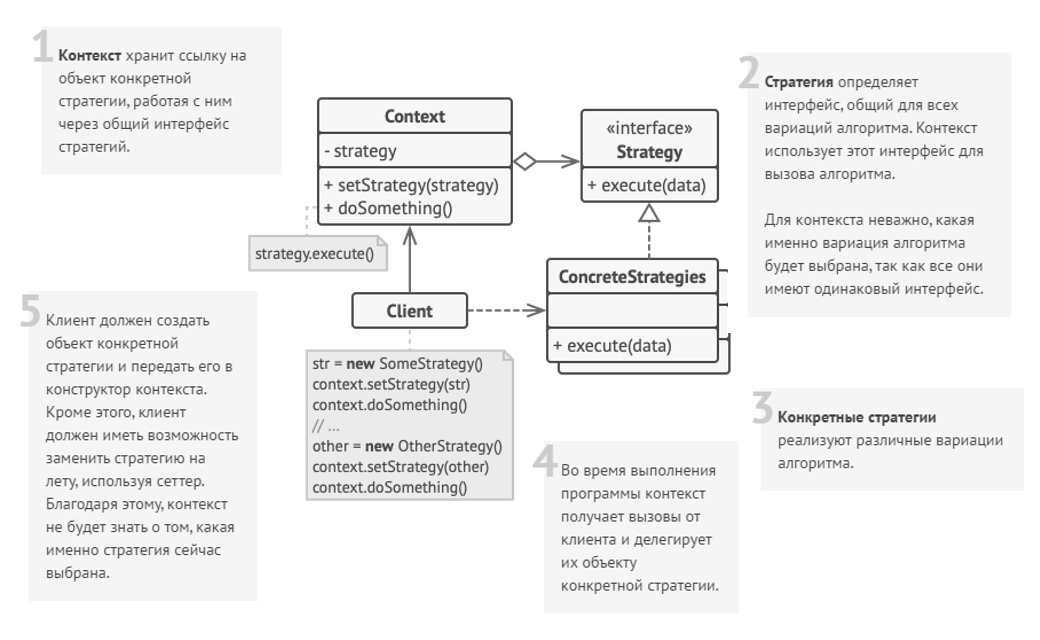
### Решение

* Паттерн Стратегия предлагает определить семейство схожих алгоритмов, которые часто изменяются или расширяются, и вынести их в собственные классы, называемые стратегиями.
* 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.
* Важно, чтобы все стратегии имели общий интерфейс. Используя этот интерфейс, контекст будет независимым от конкретных классов стратегий. С другой стороны, вы сможете изменять и добавлять новые виды алгоритмов, не трогая код контекста.
* 
* В нашем примере каждый алгоритм поиска пути переедет в свой собственный класс. В этих классах будет определён лишь один метод, принимающий в параметрах координаты начала и конца пути, а возвращающий массив точек маршрута.
* Хотя каждый класс будет прокладывать маршрут по-своему, для навигатора это не будет иметь никакого значения, так как его работа заключается только в отрисовке маршрута. Навигатору достаточно подать в стратегию данные о начале и конце маршрута, чтобы получить массив точек маршрута в оговорённом формате.
* Класс навигатора будет иметь метод для установки стратегии, позволяя изменять стратегию поиска пути на лету. Такой метод пригодится клиентскому коду навигатора, например, переключателям типов маршрутов в пользовательском интерфейсе.

### Аналогия из жизни

* 
* Вам нужно добраться до аэропорта. Можно доехать на автобусе, такси или велосипеде. Здесь вид транспорта является стратегией. Вы выбираете конкретную стратегию в зависимости от контекста — наличия денег или времени до отлёта.

### Структура

* 

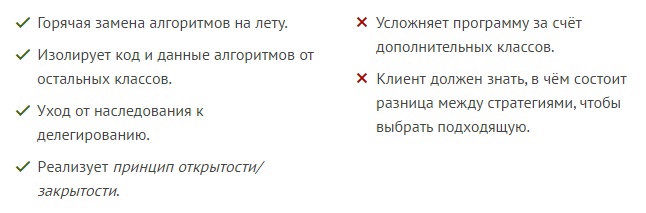
### Применимость

* Когда вам нужно использовать разные вариации какого-то алгоритма внутри одного объекта.
* Стратегия позволяет варьировать поведение объекта во время выполнения программы, подставляя в него различные объекты-поведения (например, отличающиеся балансом скорости и потребления ресурсов).
* Когда у вас есть множество похожих классов, отличающихся только некоторым поведением.
* Стратегия позволяет вынести отличающееся поведение в отдельную иерархию классов, а затем свести первоначальные классы к одному, сделав поведение этого класса настраиваемым.
* Когда вы не хотите обнажать детали реализации алгоритмов для других классов.
* Стратегия позволяет изолировать код, данные и зависимости алгоритмов от других объектов, скрыв эти детали внутри классов-стратегий.
* Когда различные вариации алгоритмов реализованы в виде развесистого условного оператора. Каждая ветка такого оператора представляет собой вариацию алгоритма.
* Стратегия помещает каждую лапу такого оператора в отдельный класс-стратегию. Затем контекст получает определённый объект-стратегию от клиента и делегирует ему работу. Если вдруг понадобится сменить алгоритм, в контекст можно подать другую стратегию.

### Шаги реализации

1. Определите алгоритм, который подвержен частым изменениям. Также подойдёт алгоритм, имеющий несколько вариаций, которые выбираются во время выполнения программы.
2. Создайте интерфейс стратегий, описывающий этот алгоритм. Он должен быть общим для всех вариантов алгоритма.
3. Поместите вариации алгоритма в собственные классы, которые реализуют этот интерфейс.
4. В классе контекста создайте поле для хранения ссылки на текущий объект-стратегию, а также метод для её изменения. Убедитесь в том, что контекст работает с этим объектом только через общий интерфейс стратегий.
5. Клиенты контекста должны подавать в него соответствующий объект-стратегию, когда хотят, чтобы контекст вёл себя определённым образом.

### Преимущества и недостатки

* 

## Отношения с другими паттернами

* Мост, Стратегия и Состояние (а также слегка и Адаптер) имеют схожие структуры классов — все они построены на принципе «композиции», то есть делегирования работы другим объектам. Тем не менее, они отличаются тем, что решают разные проблемы. Помните, что паттерны — это не только рецепт построения кода определённым образом, но и описание проблем, которые привели к данному решению.
* Команда и Стратегия похожи по духу, но отличаются масштабом и применением:
  + Команду используют, чтобы превратить любые разнородные действия в объекты. Параметры операции превращаются в поля объекта. Этот объект теперь можно логировать, хранить в истории для отмены, передавать во внешние сервисы и так далее.
  + С другой стороны, Стратегия описывает разные способы произвести одно и то же действие, позволяя взаимозаменять эти способы в каком-то объекте контекста.
* Стратегия меняет поведение объекта «изнутри», а Декоратор изменяет его «снаружи».
* Шаблонный метод использует наследование, чтобы расширять части алгоритма. Стратегия использует делегирование, чтобы изменять выполняемые алгоритмы на лету. Шаблонный метод работает на уровне классов. Стратегия позволяет менять логику отдельных объектов.
* Состояние можно рассматривать как надстройку над Стратегией. Оба паттерна используют композицию, чтобы менять поведение основного объекта, делегируя работу вложенным объектам-помощникам. Однако в Стратегии эти объекты не знают друг о друге и никак не связаны. В Состоянии сами конкретные состояния могут переключать контекст.

1. **Третья практическая работа**

**Цель работы.**

Разработать и реализовать набора классов для взаимодействия пользователя с юнитами и базой. Основные требования:

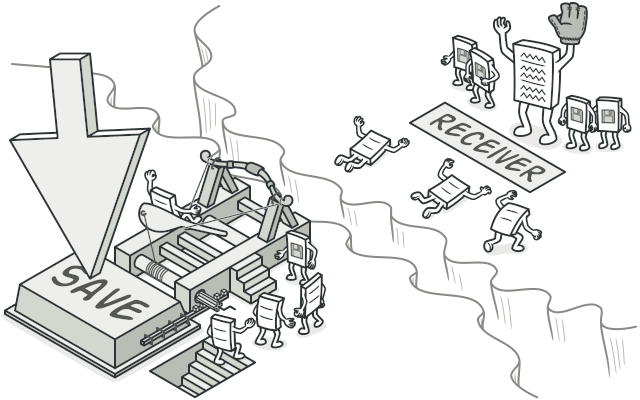
●     Должен быть реализован функционал управления юнитами

●     Должен быть реализован функционал управления базой

**Основные теоретические положения.**

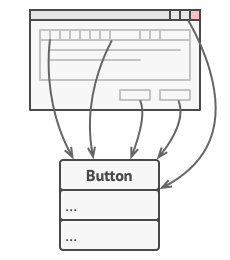
Команда

**Команда** — это поведенческий паттерн проектирования, который превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций.

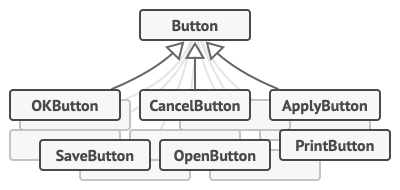


**Проблема**

Представьте, что вы работаете над программой текстового редактора. Дело как раз подошло к разработке панели управления. Вы создали класс красивых Кнопок и хотите использовать его для всех кнопок приложения, начиная от панели управления, заканчивая простыми кнопками в диалогах.



Все эти кнопки, хоть и выглядят схоже, но делают разные вещи. Поэтому возникает вопрос: куда поместить код обработчиков кликов по этим кнопкам? Самым простым решением было бы создать подклассы для каждой кнопки и переопределить в них метод действия под разные задачи.



Но скоро стало понятно, что такой подход никуда не годится. Во-первых, получается очень много подклассов. Во-вторых, код кнопок, относящийся к графическому интерфейсу, начинает зависеть от классов бизнес-логики, которая довольно часто меняется.

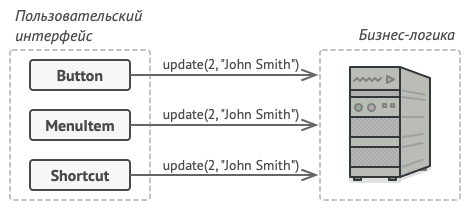


Но самое обидное ещё впереди. Ведь некоторые операции, например, «сохранить», можно вызывать из нескольких мест: нажав кнопку на панели управления, вызвав контекстное меню или просто нажав клавиши Ctrl+S. Когда в программе были только кнопки, код сохранения имелся только в подклассе SaveButton. Но теперь его придётся продублировать ещё в два класса.

**Решение**

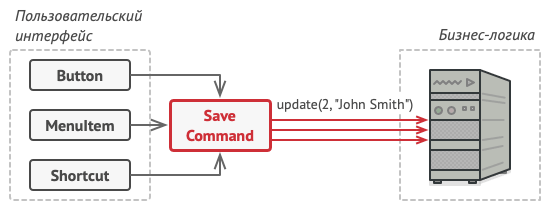
Хорошие программы обычно структурированы в виде слоёв. Самый распространённый пример — слои пользовательского интерфейса и бизнес-логики. Первый всего лишь рисует красивую картинку для пользователя. Но когда нужно сделать что-то важное, интерфейс «просит» слой бизнес-логики заняться этим.

В реальности это выглядит так: один из объектов интерфейса напрямую вызывает метод одного из объектов бизнес-логики, передавая в него какие-то параметры.



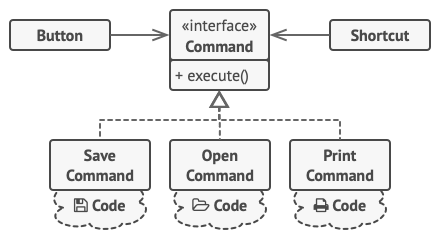
Паттерн Команда предлагает больше не отправлять такие вызовы напрямую. Вместо этого каждый вызов, отличающийся от других, следует завернуть в собственный класс с единственным методом, который и будет осуществлять вызов. Такие объекты называют *командами*.

К объекту интерфейса можно будет привязать объект команды, который знает, кому и в каком виде следует отправлять запросы. Когда объект интерфейса будет готов передать запрос, он вызовет метод команды, а та — позаботится обо всём остальном.



Классы команд можно объединить под общим интерфейсом c единственным методом запуска. После этого одни и те же отправители смогут работать с различными командами, не привязываясь к их классам. Даже больше: команды можно будет взаимозаменять на лету, изменяя итоговое поведение отправителей.

Параметры, с которыми должен быть вызван метод объекта получателя, можно загодя сохранить в полях объекта-команды. Благодаря этому, объекты, отправляющие запросы, могут не беспокоиться о том, чтобы собрать необходимые для получателя данные. Более того, они теперь вообще не знают, кто будет получателем запроса. Вся эта информация скрыта внутри команды.



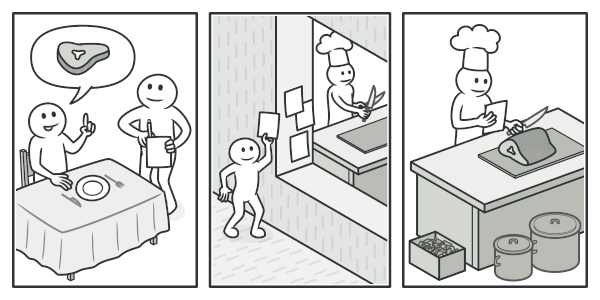
После применения Команды в нашем примере с текстовым редактором вам больше не потребуется создавать уйму подклассов кнопок под разные действия. Будет достаточно единственного класса с полем для хранения объекта команды.

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.

Так же можно поступить и с контекстным меню, и с горячими клавишами. Они будут привязаны к тем же объектам команд, что и кнопки, избавляя классы от дублирования.

Таким образом, команды станут гибкой прослойкой между пользовательским интерфейсом и бизнес-логикой. И это лишь малая доля пользы, которую может принести паттерн Команда!

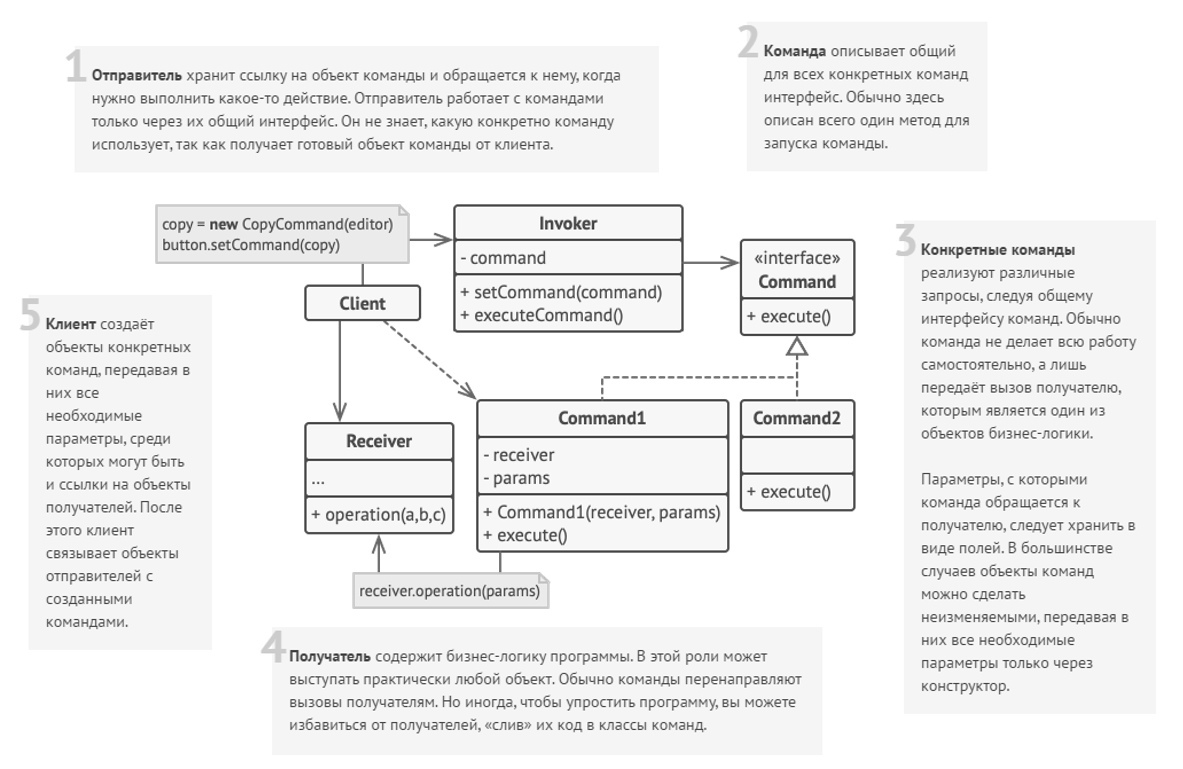
**Аналогия из жизни**



Вы заходите в ресторан и садитесь у окна. К вам подходит вежливый официант и принимает заказ, записывая все пожелания в блокнот. Откланявшись, он уходит на кухню, где вырывает лист из блокнота и клеит на стену. Далее лист оказывается в руках повара, который читает содержание заказа и готовит заказанные блюда.

В этом примере вы являетесь *отправителем*, официант с блокнотом — *командой*, а повар — *получателем*. Как и в паттерне, вы не соприкасаетесь напрямую с поваром. Вместо этого вы отправляете заказ с официантом, который самостоятельно «настраивает» повара на работу. С другой стороны, повар не знает, кто конкретно послал ему заказ. Но это ему безразлично, так как вся необходимая информация есть в листе заказа.

**Структура**



**Применимость**

* Когда вы хотите параметризовать объекты выполняемым действием.

Команда превращает операции в объекты. А объекты можно передавать, хранить и взаимозаменять внутри других объектов.

Скажем, вы разрабатываете библиотеку графического меню и хотите, чтобы пользователи могли использовать меню в разных приложениях, не меняя каждый раз код ваших классов. Применив паттерн, пользователям не придётся изменять классы меню, вместо этого они будут конфигурировать объекты меню различными командами.

* Когда вы хотите ставить операции в очередь, выполнять их по расписанию или передавать по сети.

Как и любые другие объекты, команды можно сериализовать, то есть превратить в строку, чтобы потом сохранить в файл или базу данных. Затем в любой удобный момент её можно достать обратно, снова превратить в объект команды и выполнить. Таким же образом команды можно передавать по сети, логировать или выполнять на удалённом сервере.

* Когда вам нужна операция отмены.

Главная вещь, которая вам нужна, чтобы иметь возможность отмены операций, — это хранение истории. Среди многих способов, которыми можно это сделать, паттерн Команда является, пожалуй, самым популярным.

История команд выглядит как стек, в который попадают все выполненные объекты команд. Каждая команда перед выполнением операции сохраняет текущее состояние объекта, с которым она будет работать. После выполнения операции копия команды попадает в стек истории, все ещё неся в себе сохранённое состояние объекта. Если потребуется отмена, программа возьмёт последнюю команду из истории и возобновит сохранённое в ней состояние.

Этот способ имеет две особенности. Во-первых, точное состояние объектов не так-то просто сохранить, ведь часть его может быть приватным. Но с этим может помочь справиться паттерн Снимок.

Во-вторых, копии состояния могут занимать довольно много оперативной памяти. Поэтому иногда можно прибегнуть к альтернативной реализации, когда вместо восстановления старого состояния команда выполняет обратное действие. Недостаток этого способа в сложности (а иногда и невозможности) реализации обратного действия.

**Шаги реализации**

1. Создайте общий интерфейс команд и определите в нём метод запуска.
2. Один за другим создайте классы конкретных команд. В каждом классе должно быть поле для хранения ссылки на один или несколько объектов-получателей, которым команда будет перенаправлять основную работу.

Кроме этого, команда должна иметь поля для хранения параметров, которые нужны при вызове методов получателя. Значения всех этих полей команда должна получать через конструктор.

И, наконец, реализуйте основной метод команды, вызывая в нём те или иные методы получателя.

1. Добавьте в классы отправителей поля для хранения команд. Обычно объекты-отправители принимают готовые объекты команд извне — через конструктор либо через сеттер поля команды.
2. Измените основной код отправителей так, чтобы они делегировали выполнение действия команде.
3. Порядок инициализации объектов должен выглядеть так:
   * Создаём объекты получателей.
   * Создаём объекты команд, связав их с получателями.
   * Создаём объекты отправителей, связав их с командами.

**Преимущества и недостатки**



**Отношения с другими паттернами**

Цепочка обязанностей, Команда, Посредник и Наблюдатель показывают различные способы работы отправителей запросов с их получателями:

*Цепочка обязанностей* передаёт запрос последовательно через цепочку потенциальных получателей, ожидая, что какой-то из них обработает запрос.

*Команда* устанавливает косвенную одностороннюю связь от отправителей к получателям.

*Посредник* убирает прямую связь между отправителями и получателями, заставляя их общаться опосредованно, через себя.

*Наблюдатель* передаёт запрос одновременно всем заинтересованным получателям, но позволяет им динамически подписываться или отписываться от таких оповещений.

Обработчики в Цепочке обязанностей могут быть выполнены в виде Команд. В этом случае множество разных операций может быть выполнено над одним и тем же контекстом, коим является запрос.

Но есть и другой подход, в котором сам запрос является *Командой*, посланной по цепочке объектов. В этом случае одна и та же операция может быть выполнена над множеством разных контекстов, представленных в виде цепочки.

Команду и Снимок можно использовать сообща для реализации отмены операций. В этом случае объекты команд будут отвечать за выполнение действия над объектом, а снимки будут хранить резервную копию состояния этого объекта, сделанную перед самым запуском команды.

Команда и Стратегия похожи по духу, но отличаются масштабом и применением:

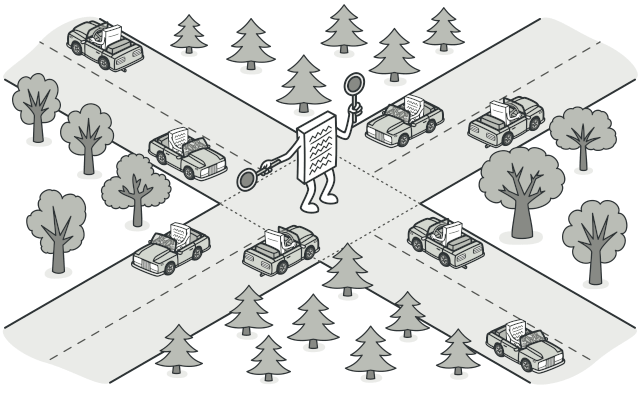
*Команду* используют, чтобы превратить любые разнородные действия в объекты. Параметры операции превращаются в поля объекта. Этот объект теперь можно логировать, хранить в истории для отмены, передавать во внешние сервисы и так далее.

С другой стороны, *Стратегия* описывает разные способы произвести одно и то же действие, позволяя взаимозаменять эти способы в каком-то объекте контекста.

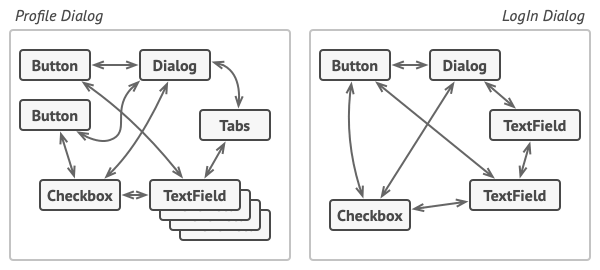
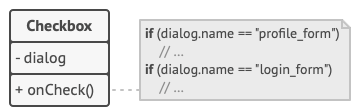
Если Команду нужно копировать перед вставкой в историю выполненных команд, вам может помочь Прототип.

Посетитель можно рассматривать как расширенный аналог Команды, который способен работать сразу с несколькими видами получателей.

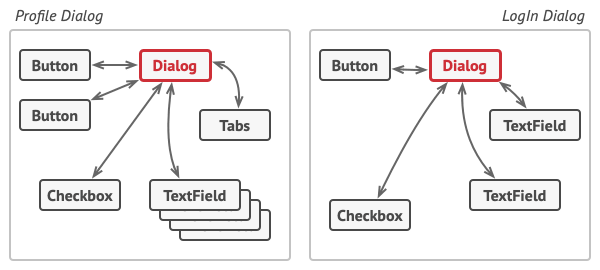
Посредник

* **Посредник** — это поведенческий паттерн проектирования, который позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник.
* 

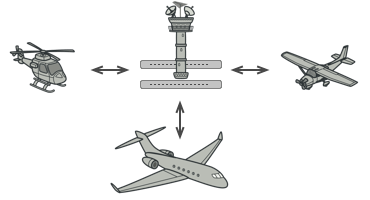
### Проблема

* Предположим, что у вас есть диалог создания профиля пользователя. Он состоит из всевозможных элементов управления — текстовых полей, чекбоксов, кнопок.
* 
* Отдельные элементы диалога должны взаимодействовать друг с другом. Так, например, чекбокс «у меня есть собака» открывает скрытое поле для ввода имени домашнего любимца, а клик по кнопке отправки запускает проверку значений всех полей формы.
* 
* Прописав эту логику прямо в коде элементов управления, вы поставите крест на их повторном использовании в других местах приложения. Они станут слишком тесно связанными с элементами диалога редактирования профиля, которые не нужны в других контекстах. Поэтому вы сможете использовать либо все элементы сразу, либо ни одного.

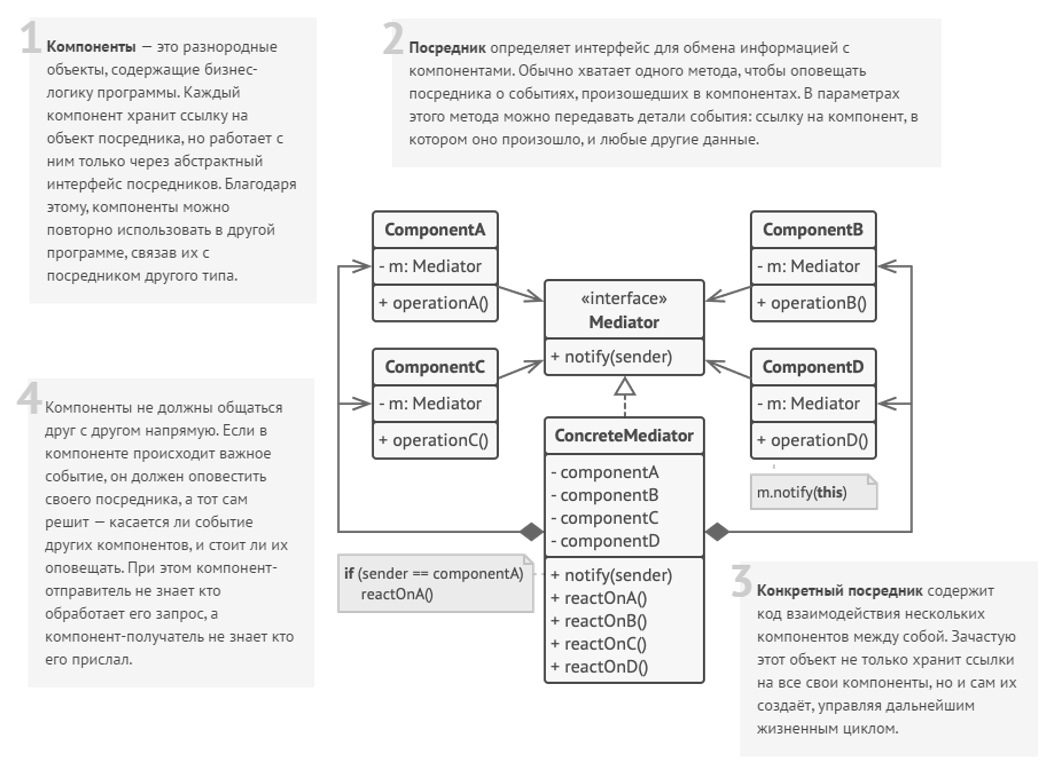
### Решение

* Паттерн Посредник заставляет объекты общаться не напрямую друг с другом, а через отдельный объект-посредник, который знает, кому нужно перенаправить тот или иной запрос. Благодаря этому, компоненты системы будут зависеть только от посредника, а не от десятков других компонентов.
* В нашем примере посредником мог бы стать диалог. Скорее всего, класс диалога и так знает, из каких элементов состоит, поэтому никаких новых связей добавлять в него не придётся.
* 
* Основные изменения произойдут внутри отдельных элементов диалога. Если раньше при получении клика от пользователя объект кнопки сам проверял значения полей диалога, то теперь его единственной обязанностью будет сообщить диалогу о том, что произошёл клик. Получив извещение, диалог выполнит все необходимые проверки полей. Таким образом, вместо нескольких зависимостей от остальных элементов кнопка получит только одну — от самого диалога.
* Чтобы сделать код ещё более гибким, можно выделить общий интерфейс для всех посредников, то есть диалогов программы. Наша кнопка станет зависимой не от конкретного диалога создания пользователя, а от абстрактного, что позволит использовать её и в других диалогах.
* Таким образом, посредник скрывает в себе все сложные связи и зависимости между классами отдельных компонентов программы. А чем меньше связей имеют классы, тем проще их изменять, расширять и повторно использовать.

### Аналогия из жизни

* 
* Пилоты садящихся или улетающих самолётов не общаются напрямую с другими пилотами. Вместо этого они связываются с диспетчером, который координирует действия нескольких самолётов одновременно. Без диспетчера пилотам приходилось бы все время быть начеку и следить за всеми окружающими самолётами самостоятельно, а это приводило бы к частым катастрофам в небе.
* Важно понимать, что диспетчер не нужен во время всего полёта. Он задействован только в зоне аэропорта, когда нужно координировать взаимодействие многих самолётов.

### Структура

* 

### Применимость

* Когда вам сложно менять некоторые классы из-за того, что они имеют множество хаотичных связей с другими классами.
* Посредник позволяет поместить все эти связи в один класс, после чего вам будет легче их отрефакторить, сделать более понятными и гибкими.
* Когда вы не можете повторно использовать класс, поскольку он зависит от уймы других классов.
* После применения паттерна компоненты теряют прежние связи с другими компонентами, а всё их общение происходит косвенно, через объект-посредник.
* Когда вам приходится создавать множество подклассов компонентов, чтобы использовать одни и те же компоненты в разных контекстах.
* Если раньше изменение отношений в одном компоненте могли повлечь за собой лавину изменений во всех остальных компонентах, то теперь вам достаточно создать подкласс посредника и поменять в нём связи между компонентами.

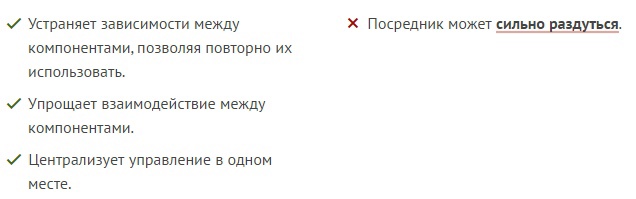
### Шаги реализации

1. Найдите группу тесно переплетённых классов, отвязав которые друг от друга, можно получить некоторую пользу. Например, чтобы повторно использовать их код в другой программе.
2. Создайте общий интерфейс посредников и опишите в нём методы для взаимодействия с компонентами. В простейшем случае достаточно одного метода для получения оповещений от компонентов.

* Этот интерфейс необходим, если вы хотите повторно использовать классы компонентов для других задач. В этом случае всё, что нужно сделать — это создать новый класс конкретного посредника.

1. Реализуйте этот интерфейс в классе конкретного посредника. Поместите в него поля, которые будут содержать ссылки на все объекты компонентов.
2. Вы можете пойти дальше и переместить код создания компонентов в класс посредника, после чего он может напоминать фабрику или фасад.
3. Компоненты тоже должны иметь ссылку на объект посредника. Связь между ними удобнее всего установить, подавая посредника в параметры конструктора компонентов.
4. Измените код компонентов так, чтобы они вызывали метод оповещения посредника, вместо методов других компонентов. С противоположной стороны, посредник должен вызывать методы нужного компонента, когда получает оповещение от компонента.

### Преимущества и недостатки



## Отношения с другими паттернами

Цепочка обязанностей, Команда, Посредник и Наблюдатель показывают различные способы работы отправителей запросов с их получателями:

Цепочка обязанностей передаёт запрос последовательно через цепочку потенциальных получателей, ожидая, что какой-то из них обработает запрос.

Команда устанавливает косвенную одностороннюю связь от отправителей к получателям.

Посредник убирает прямую связь между отправителями и получателями, заставляя их общаться опосредованно, через себя.

Наблюдатель передаёт запрос одновременно всем заинтересованным получателям, но позволяет им динамически подписываться или отписываться от таких оповещений.

Посредник и Фасад похожи тем, что пытаются организовать работу множества существующих классов.

Фасад создаёт упрощённый интерфейс к подсистеме, не внося в неё никакой добавочной функциональности. Сама подсистема не знает о существовании Фасада. Классы подсистемы общаются друг с другом напрямую.

Посредник централизует общение между компонентами системы. Компоненты системы знают только о существовании Посредника, у них нет прямого доступа к другим компонентам.

Разница между Посредником и Наблюдателем не всегда очевидна. Чаще всего они выступают как конкуренты, но иногда могут работать вместе.

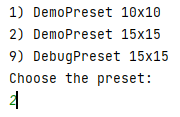
Цель Посредника — убрать обоюдные зависимости между компонентами системы. Вместо этого они становятся зависимыми от самого посредника. С другой стороны, цель Наблюдателя — обеспечить динамическую одностороннюю связь, в которой одни объекты косвенно зависят от других.

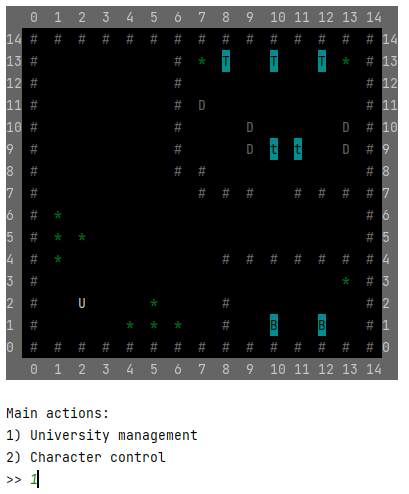
Довольно популярна реализация Посредника при помощи Наблюдателя. При этом объект посредника будет выступать издателем, а все остальные компоненты станут подписчиками и смогут динамически следить за событиями, происходящими в посреднике. В этом случае трудно понять, чем же отличаются оба паттерна.

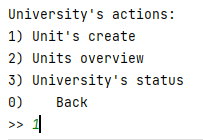
Но Посредник имеет и другие реализации, когда отдельные компоненты жёстко привязаны к объекту посредника. Такой код вряд ли будет напоминать Наблюдателя, но всё же останется Посредником.

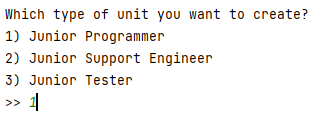
Напротив, в случае реализации посредника с помощью Наблюдателя представим такую программу, в которой каждый компонент системы становится издателем. Компоненты могут подписываться друг на друга, в то же время не привязываясь к конкретным классам. Программа будет состоять из целой сети Наблюдателей, не имея центрального объекта-Посредника.

**Экспериментальные результаты.**

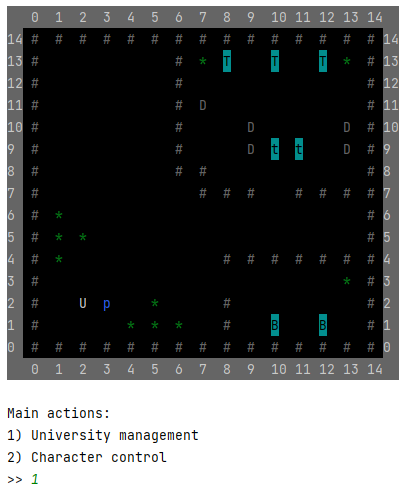


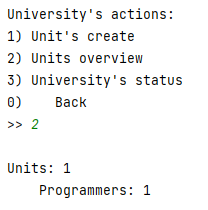


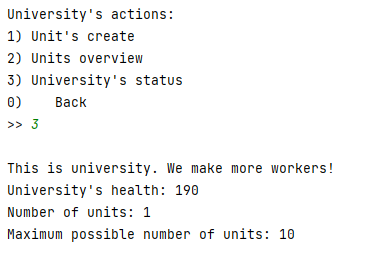


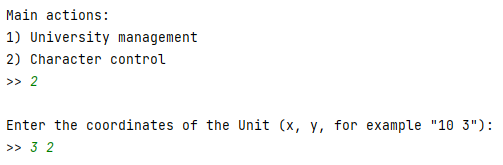


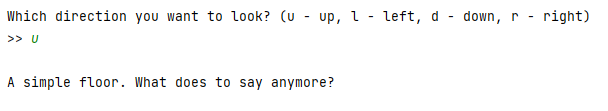


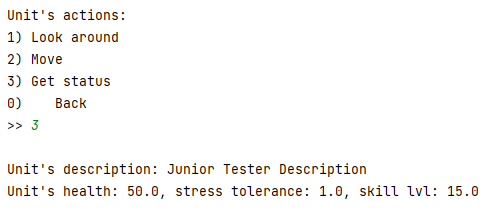


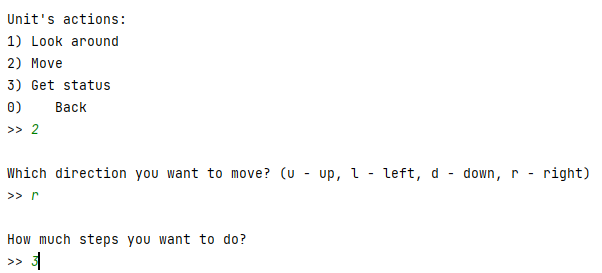


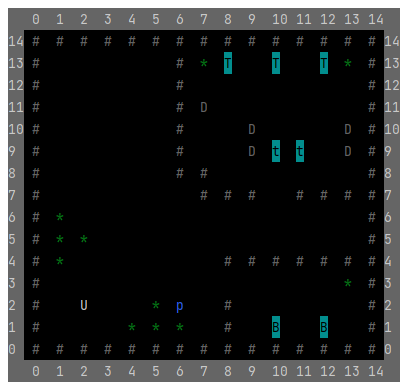










  
Рисунок 5 – Результат работы программы (полный код программы представлен в приложении А)

**заключение**

В рамках выполненной работы были изучены:

* Классы.
* Объектно-ориентированная парадигма.
* Принципы объектно-ориентированного программирования.
* Паттерны проектирования.

Также на их основе была реализована программа, в которой были задействованы все полученные знания.

Программа работает без сбоев, в её коде можно видеть примеры всех изученных тем.

Приложение а  
полный код программы

<https://github.com/Celeron404/leti2021-2/tree/master/src/main/java/com/leti2021_2>