**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

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

отчет

**по лабораторной работе №1**

**по дисциплине «Программирование»**

Тема: «Введение в ООП: создание классов, конструкторов классов, методов классов; наследование»

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

Санкт-Петербург

2020

**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.

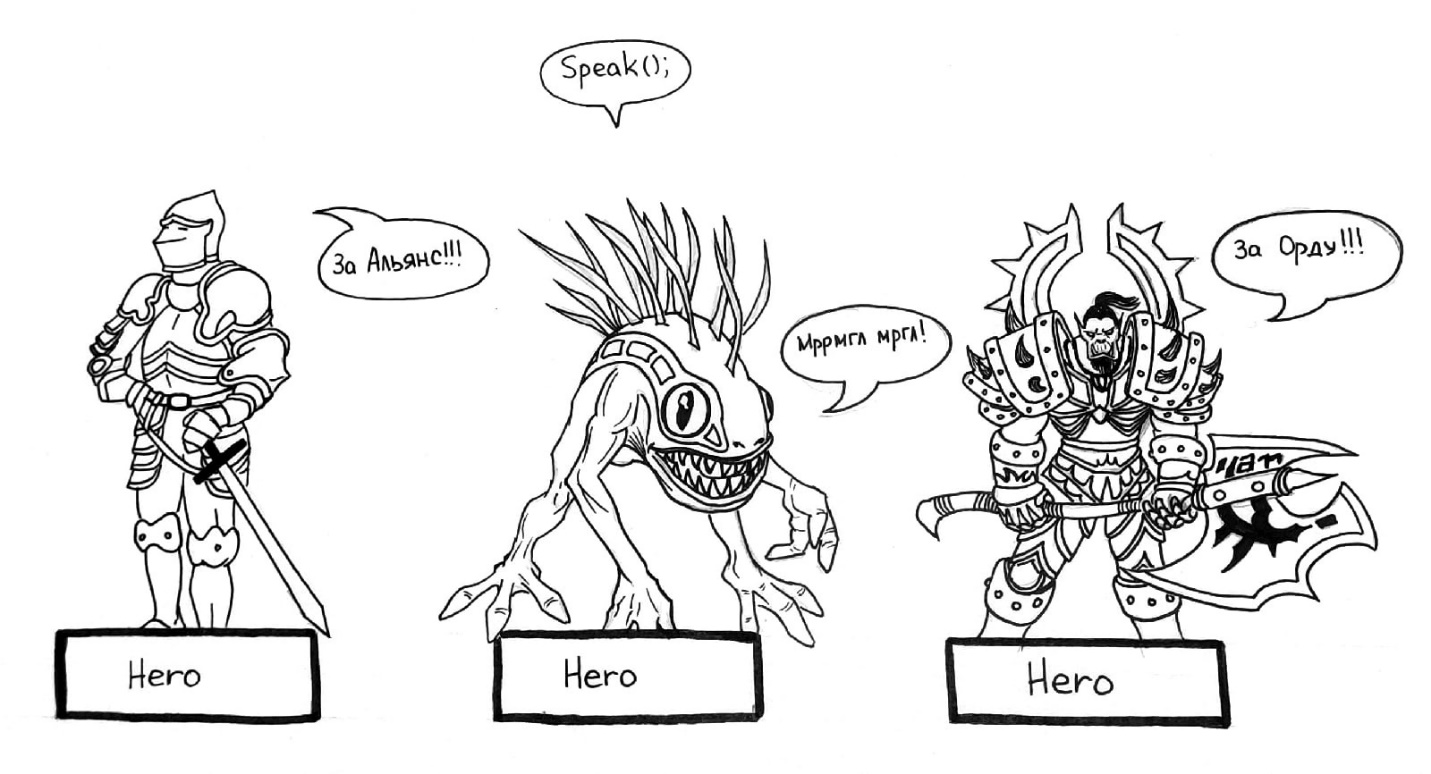
## **Наследование**

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 { ... };

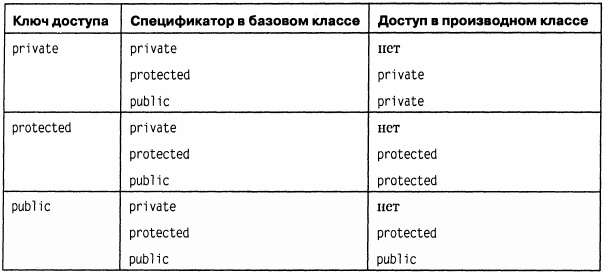
class В { ... };

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

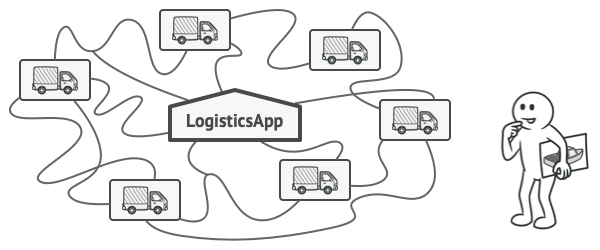
**Фабричный метод** — это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.



### 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.

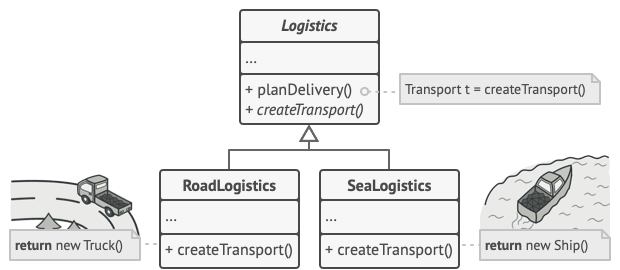


Отличные новости, правда?! Но как насчёт кода? Большая часть существующего кода жёстко привязана к классам Грузовиков. Чтобы добавить в программу классы морских Судов, понадобится перелопатить всю программу. Более того, если вы потом решите добавить в программу ещё один вид транспорта, то всю эту работу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

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

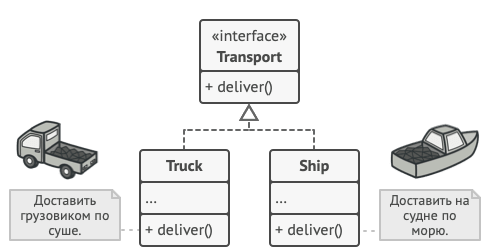
### Решение

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.

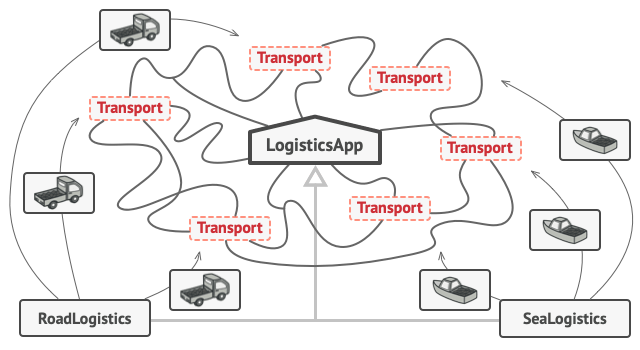


На первый взгляд, это может показаться бессмысленным: мы просто переместили вызов конструктора из одного конца программы в другой. Но теперь вы сможете переопределить фабричный метод в подклассе, чтобы изменить тип создаваемого продукта.

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

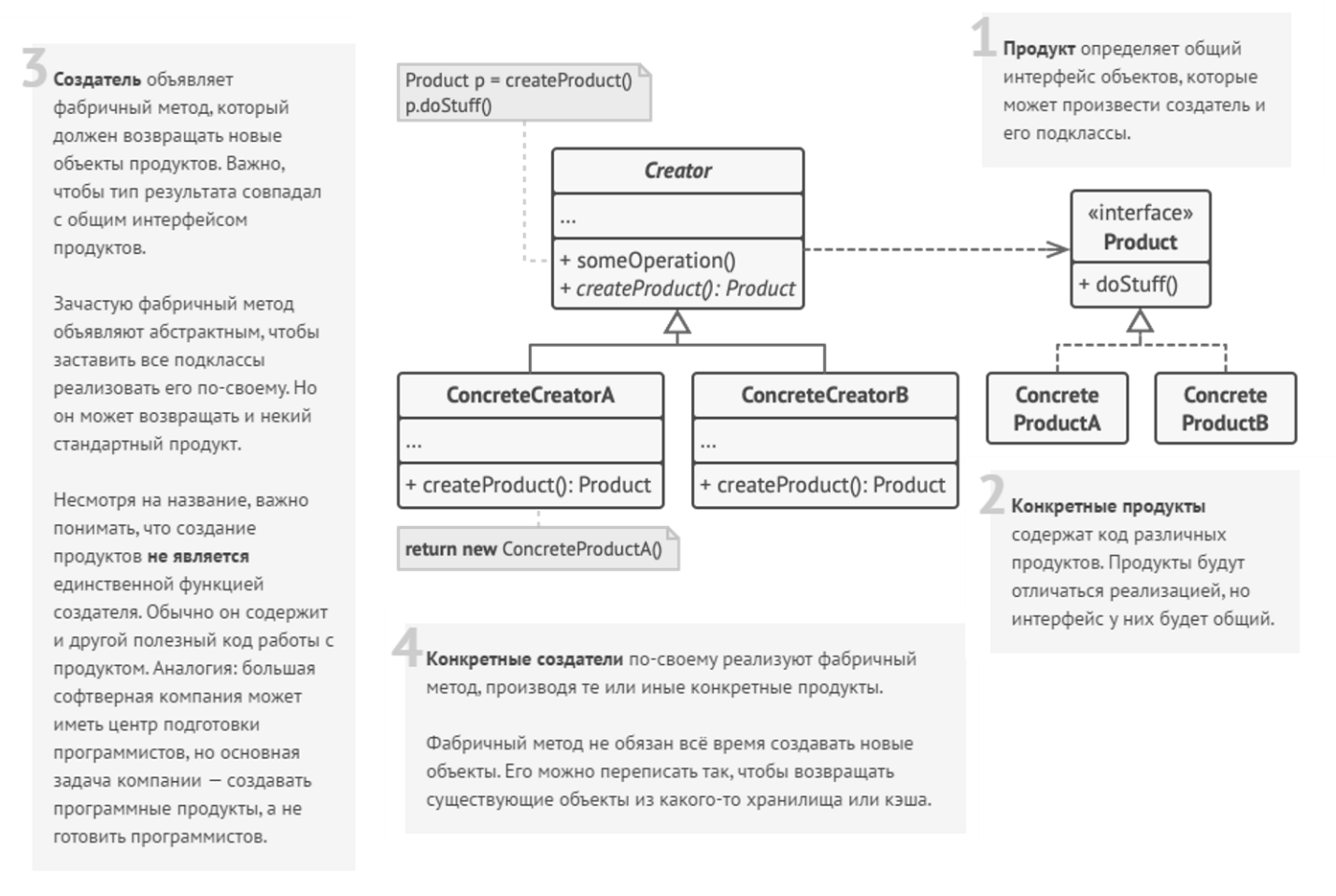


 Например, классы Грузовик и Судно реализуют интерфейс Транспорт с методом доставить. Каждый из этих классов реализует метод по-своему: грузовики везут грузы по земле, а суда — по морю. Фабричный метод в классе ДорожнойЛогистики вернёт объект-грузовик, а класс МорскойЛогистики — объект-судно.



 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.

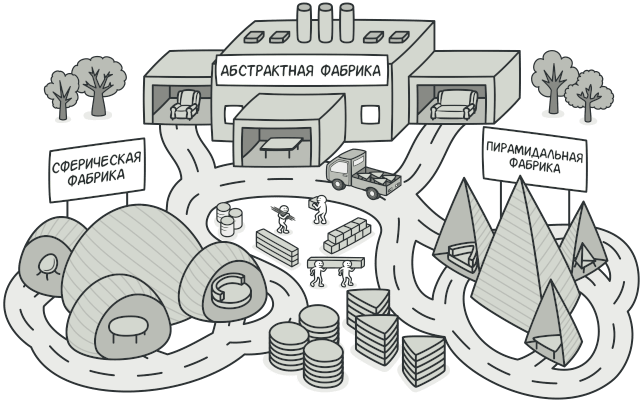
Фабричный метод можно использовать вместе с Итератором, чтобы подклассы коллекций могли создавать подходящие им итераторы.

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

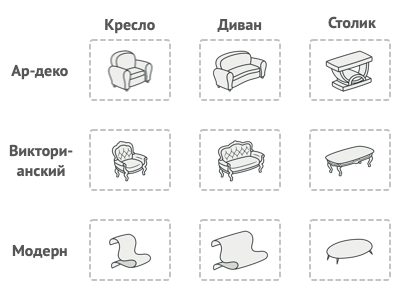
**Абстрактная фабрика** — это порождающий паттерн проектирования, который позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.



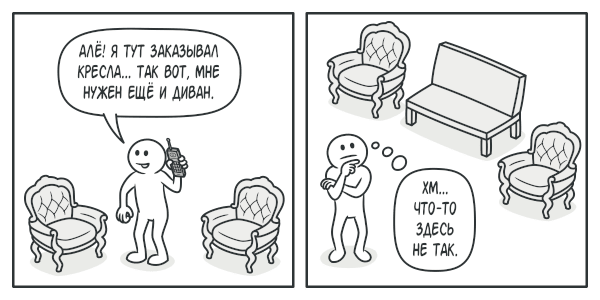
### Проблема

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.



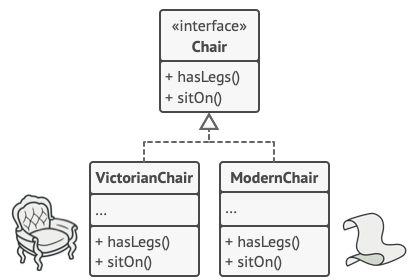
Вам нужен такой способ создавать объекты продуктов, чтобы они сочетались с другими продуктами того же семейства. Это важно, так как клиенты расстраиваются, если получают несочетающуюся мебель.



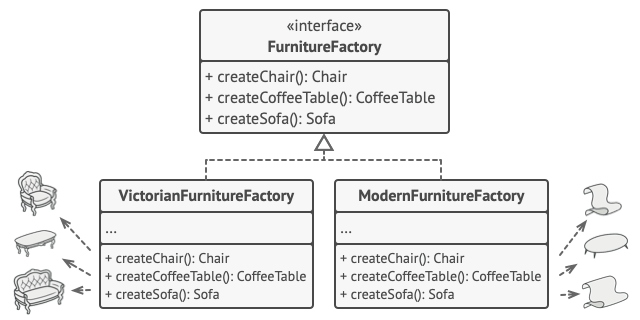
Кроме того, вы не хотите вносить изменения в существующий код при добавлении новых продуктов или семейств в программу. Поставщики часто обновляют свои каталоги, и вы бы не хотели менять уже написанный код каждый раз при получении новых моделей мебели.

### 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.

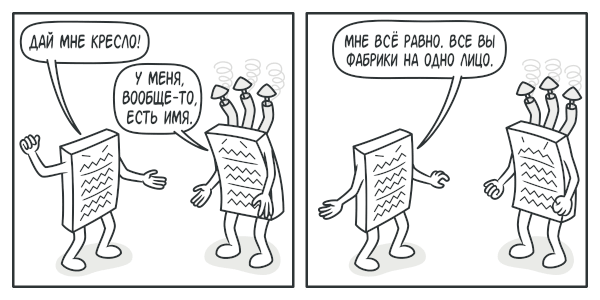


Далее вы создаёте абстрактную фабрику — общий интерфейс, который содержит методы создания всех продуктов семейства (например, создатьКресло, создатьДиван и создатьСтолик). Эти операции должны возвращать **абстрактные** типы продуктов, представленные интерфейсами, которые мы выделили ранее — Кресла, Диваны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

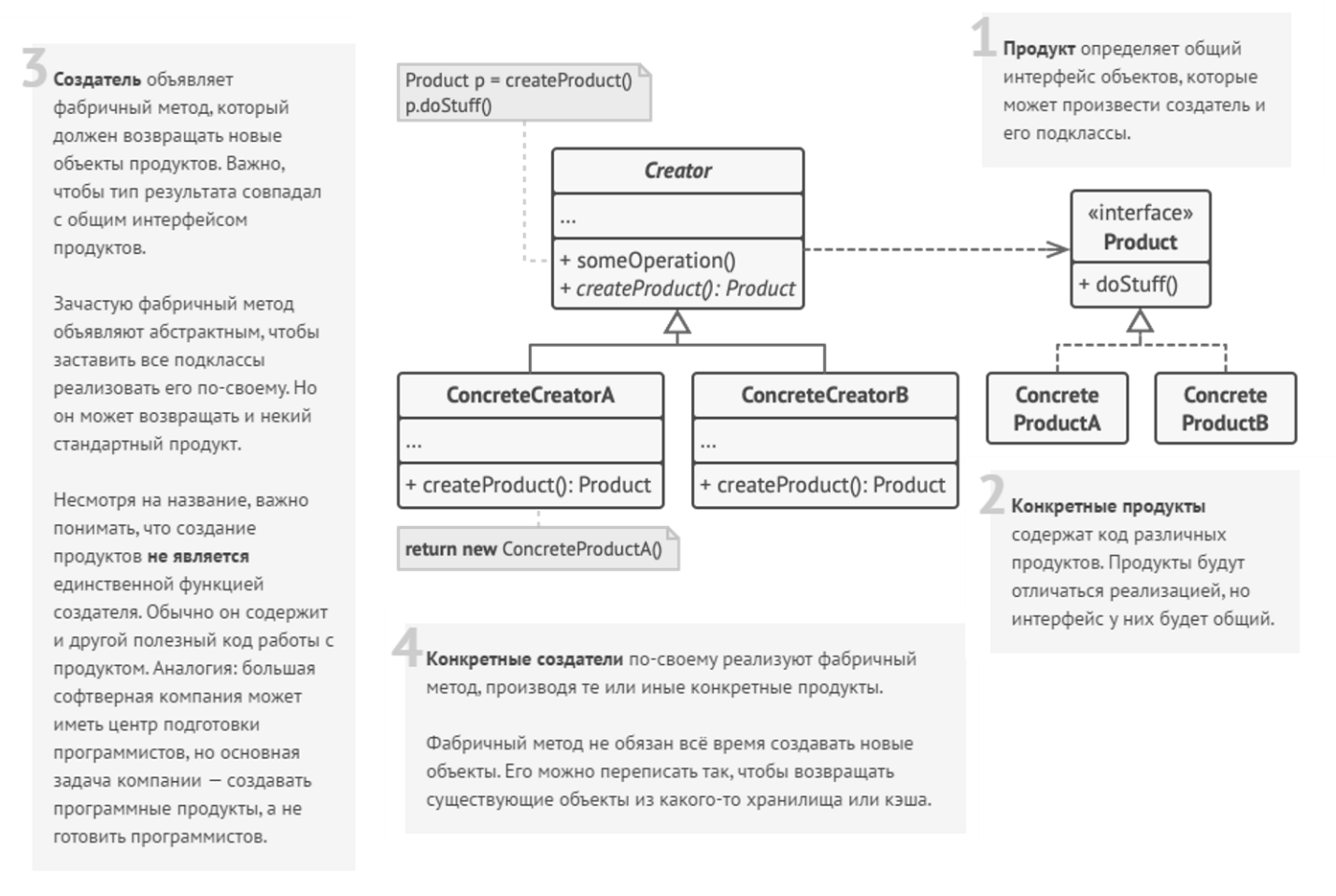
Клиентский код должен работать как с фабриками, так и с продуктами только через их общие интерфейсы. Это позволит подавать в ваши классы любой тип фабрики и производить любые продукты, ничего не ломая.



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.

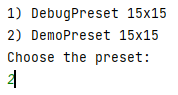
Классы Абстрактной фабрики чаще всего реализуются с помощью Фабричного метода, хотя они могут быть построены и на основе Прототипа.

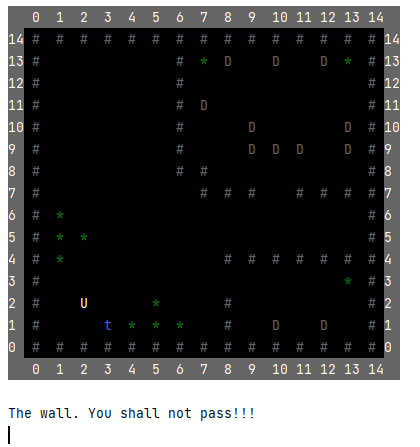
Абстрактная фабрика может быть использована вместо Фасада для того, чтобы скрыть платформо-зависимые классы.

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.

**Экспериментальные результаты.**





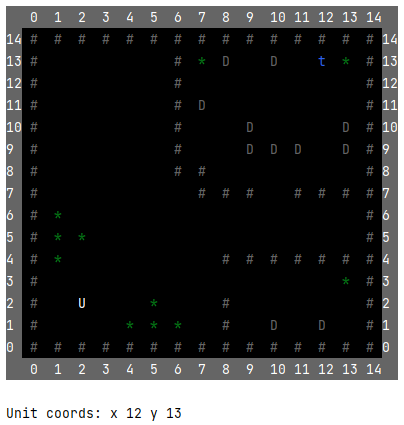


Рисунок 5 – Результат работы программы (полный код программы представлен в приложении А)

Приложение а  
полный код программы

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