Лекции находятся здесь: https://yadi.sk/d/u1qu83FiBUvHRg

1. Программные продукты как сложные системы. Признаки сложных систем. Декомпозиция.

- Сложные системы являются иерархическими и состоят из взаимозависимых систем, которые в свою очередь могут быть иерархическими и также состоять из систем.
- Выбор, какие компоненты в данной системе будут элементарными, произволен и в большей степени зависит от исследователя.
- Внутрикомпонентная связь элементов выше, чем внешнекомпонентная связь.
- Иерархические системы обычно состоят из немногих типов подсистем, по-разному спроектированных и реализованных.
- Любая сложная система является развитием более простой.

Декомпозиция - это разбиение целого на части.

2. Причины сложности программных систем.

- Сложная предметная область
- Трудность управления
- Гибкость программы - свойство, которое позволяет использовать один и тот же код в разных приложениях
- Проблема описания больших дискретных систем

3. Объект. Что не является объектом, подходы к выделению объектов.

Объект - это сущность. способная сохранять свое состояние и обеспечивать набор операций для проверки и изменения этого состояния.
Объект - это конкретное представление абстракции.
Объект - это экземпляр класса. Структура и поведение объекта описано в классе.

4. Способы выделения объектов для объектной декомпозиции

1. Метод подчеркивания существительных
2. Выделение объектов по категориям
   - Выделяем объекты на этапе анализа (методом 1)
   - Объекты на этапе реализации
   - Объекты на этапе проектирования
3. Изучение потенциальных источников объектов
   - анализ аналогичного ПО
   - Опытное обнаружение объектов: предполагается выявление элементов информации, перемещаемой от одного объекта к другому.
4. Использование сторонних библиотек

5. Классы. Абстракции.

Класс определяет абстракцию существующего объекта. Объект существует в течение некоторого времени, а класс не существует (условно).
Класс - это некое множество объектов, имеющих общую структуру и общее поведение.
Класс - это структурный вид данных, который включает в себя описание полей и функций, названные методами.
Класс - это пользовательский тип данных.

6. Понятие модуля. Интерфейс.

Интерфейс - это совокупность абстракций, доступных пользователю извне абстракции.
Модуль - это физический контейнер некоторого набора логических элементов.

7. Парадигмы программирования.

Императивное программирование: Инструкция, Состояние
В этой парадигме вычисления описываются в виде инструкций, шаг за шагом изменяющих состояние программы. В низкоуровневых языках (таких как язык ассемблера) состоянием могут быть память, регистры и флаги, а инструкциями — те команды, что поддерживает целевой процессор. В более высокоуровневых (таких как Си) состояние — это только память, инструкции могут быть сложнее и вызывать выделение и освобождение памяти в процессе своей работы. В совсем высокоуровневых (таких как Python, если на нем программировать императивно) состояние ограничивается лишь переменными, а команды могут представлять собой комплексные операции, которые на ассемблере занимали бы сотни строк.
Структурное программирование: Блок, Цикл, Ветвление
Эта парадигма вводит новые понятия, объединяющие часто используемые шаблоны написания императивного кода. В структурном программировании мы по прежнему оперируем состоянием и инструкциями, однако вводится понятие составной инструкции (блока), инструкций ветвления и цикла. Благодаря этим простым изменениям возможно отказаться от оператора goto в
большинстве случаев, что упрощает код. Иногда goto все-же делает код читабельнее, благодаря чему он до сих пор широко используется, несмотря на все заявления его противников.
Процедурное программирование: Процедура
Процедура — самостоятельный участок кода, который можно выполнить как одну инструкцию. В современном программировании процедура может иметь несколько точек выхода (return в C-подобных языках), несколько точек входа (с помощью yield в Python или статических локальных переменных в C++), иметь аргументы, возвращать значение как результат своего выполнения, быть перегруженной по количеству или типу параметров и много чего еще.
Модульное программирование: Модуль, Импортирование.
Модуль — это отдельная именованная сущность программы, которая объединяет в себе другие программные единицы, близкие по функциональности.

8. Этапы разработки программных средств с использованием объектно-ориентированного подхода. Объектно-ориентированное программирование.

Объектно-ориентированный анализ - методология, при которой требования к системе воспринимаются с точки зрения класса и объекта, выявленных в предметной области.
Объектно-ориентированное проектирование - методология проектирования, соединяющая в себе процесс декомпозиции и приемы представления логической и физической, а также статических и динамических моделей проектирования системы.
Объектно-ориентированное программирование - осн. на представлении программы в виде совокупности объектов, каждый из которых может являться экземпляром определенного класса или типа в иерархии наследования.

9. Структура описания класса в C++. Области видимости в классах.

В классе могут быть объявлены 3 метки спецификации доступа: public, private и protected. Все методы и свойства класса, объявленные после спецификатора доступа public будут доступны другим функциям и объектам в программе. Все методы и свойства класса, объявленные после спецификатор доступа private будут доступны только внутри класса. Все методы и свойства класса, объявленные с модификатором protected будут доступны только классам-наследникам.

10. Принципы объектно-ориентированного представления программных систем (абстрагирование, ограничение доступа)

1. Абстрагирование - процесс выделения абстракций. Абстракция это совокупность существенных характеристик объекта, которые отличают его от других видов, то есть, четко определяют данный объект с точки зрения дальнейшего рассмотрения и анализа. (Абстракция = класс).
2. Ограничение доступа - это сокрытие отдельных элементов абстракции, не затрагивающий ее существенных характеристик. Необходимость ограничения доступа предполагает выделение двух частей:
Интерфейс (совокупность доступных извне абстракции характеристик, состояний и поведения абстракции) и совокупность недоступных извне элементов абстракции, включает внутреннюю организацию абстракции и ее реализацию.

11. Виды методов в C++. Раннее и позднее связывание.

Модификатор - операция для изменения состояния текущего объекта
Селектор - получение/установка состояния
Итератор - операция последовательного доступа к объекту
Конструктор - операция создания
Деструктор - операция разрушения

Связывание — это сопоставление вызова функции с вызовом. В C++ все функции по умолчанию имеют *раннее связывание*, то есть компилятор и компоновщик решают, какая именно функция должна быть вызвана, до запуска программы. Виртуальные функции имеют *позднее связывание*, то есть при вызове функции нужное тело выбирается на этапе выполнения программы.

12. Виртуальные методы. Таблица виртуальных методов.

Виртуальный метод — в объектно-ориентированном программировании метод класса, который может быть переопределён в классах-наследниках так, что конкретная реализация метода для вызова будет определяться во время исполнения.
Таблица виртуальных функций хранит в себе адреса всех виртуальных методов класса (по сути, это массив указателей), а также всех виртуальных методов базовых классов этого класса.

13. Виртуальные методы. Практическое применение.

Виртуальный метод — метод класса, который может быть переопределён в классах-наследниках так, что конкретная реализация метода для вызова будет определяться во время исполнения. Таким образом, программисту необязательно знать точный тип объекта для работы с ним через виртуальные методы: достаточно лишь знать, что объект принадлежит классу или наследнику класса, в котором метод объявлен.
Описывается так:
virtual <имя метода> (<список параметров>){}
Если мы хотим указать, что реализация метода не определяется в данном классе, дописываем в конце = 0; Получаем:
virtual <имя метода> (<список параметров>) = 0;
Виртуальный метод нужно обязательно переопределить в классе-потомке, иначе потомок - тоже абстрактный.
Абстрактный класс - хотя бы 1 метод абстрактный.
Практически виртуальные методы нужны для реализации динамического полиморфизма. Его общий принцип - это единообразная работа с разными объектами, имеющими разное поведение.
Этот принцип (единообразная работа с объектами разных типов) наиболее чисто воплощается в концепции интерфейса. Мы определяем некий класс-предок, он же интерфейс, в котором содержатся общие операции и естественным образом ограничиваемся только перечисленными в этом предке операциями. Каждая операция задаётся в виде виртуальной функции, причём чистой: сама функция только объявлена в интерфейсе, но не реализована. Больше в интерфейсе нет ничего - ни данных, ни вспомогательных операций. От интерфейса наследуют классы, которые обязаны реализовать все операции интерфейса. Такие классы называются реализацией интерфейса. При создании объектов-реализаций указатели на них приводятся к базовому классу, после чего работа с ними происходит единообразно в рамках, заданных интерфейсом. Возможны случаи, когда конкретный тип объектов

вообще скрывается от того, кто эти объекты использует (например, при загрузке из файла). Тогда всё, что известно программисту - только сам интерфейс, и работа может вестись только через перечисленные там операции.

14. Принципы объектно-ориентированного представления программных систем (модульность, иерархическая организация).

Модульность - принцип разработки программной системы, которая предполагает ее реализацию в отдельных модулях. Модуль - это физический контейнер некоторого набора логических элементов. Следование этому принципу значительно упрощает разработку.
Иерархическая организация предполагает использование иерархий при разработке программной системы. Иерархии упрощают систему абстракций. Иерархия - это упорядочивание абстракций, расположение их по уровню.

15. Наследование. Виды наследования в C++.

Наследование. Механизм, позволяющий создавать новые классы на основе других классов(родителей).
Новые классы могут быть производными от существующих классов, с помощью механизма «наследования». Классы, используемые для наследования, называются "базовыми классами" определенного производного класса.

- Одиночное наследование:

При одиночном наследовании, каждый класс имеет только один базовый класс.

- Множественное наследование:

При множественном наследовании у класса может быть более одного предка. В этом случае класс наследует методы всех предков. Достоинства такого подхода в большей гибкости.

Есть три типа наследования public, protected и private.
При public - наследовании все спецификаторы остаются без изменения.
При protected-наследовании все спецификаторы остаются без изменения, кроме спецификатора public, который меняется на спецификатор protected (то есть public-члены базового класса в потомках становятся protected).
При private-наследовании все спецификаторы меняются на private.

тип наследования	поле родителя	поле наследника
public	public	public
public	protected	protected
public	private	private
protected	public	protected
protected	protected	protected
protected	private	private
private	public	private
private	protected	private
private	private	private

16. Виды методов в C++ (конструктор, деструктор, абстрактные методы).

Абстрактный метод - это метод класса, реализация для которого отсутствует.
Конструктор– это специальный метод класса, который предназначен для инициализации элементов класса некоторыми начальными значениями.
Деструктор — специальный метод класса, который служит для уничтожения элементов класса.

- при объявлении конструктора, тип данных возвращаемого значения не указывается, в том числе — void;
- у деструктора также нет типа данных для возвращаемого значения, к тому же деструктору нельзя передавать никаких параметров;
- имя класса и конструктора должно быть идентично;
- имя деструктора идентично имени конструктора, но с приставкой ~ ;
- В классе допустимо создавать несколько конструкторов, если это необходимо. Имена, согласно пункту 2 нашего списка, будут одинаковыми. Компилятор будет их различать по передаваемым параметрам (как при перегрузке функций). Если мы не передаем в конструктор параметры, он считается конструктором по умолчанию;
- в классе может быть объявлен только один деструктор;
    - Конструкторы можно перегрузить как и любой метод в языке

17. Конструкторы и инициализация объектов.

Конструктор — функция, предназначенная для инициализации объектов класса. Конструкторы имеют имена, совпадающие с именами классов, и не имеют возвращаемых значений.
Существует три способа инициализации объектов

```
A(int a1):a(a1){} // список инициализации

A(int a1){ a = a1;} // Присваивается после выделения памяти под объект }
```

```
A(int a1):a{a1}{}; // Uniform инициализация
```

18. Конструкторы и методы создания экземпляра класса.

Создать объект можно тремя способами:
1. Вызов его конструктора data d()
2. Выделением памяти под указатель и вызовом конструктора data *d = new data()
3. Присваивание data d = d1;

19. Дополнительные принципы ООП.

Типизация - ограничение, накладываемое на свойства объектов, препятствующее взаимозаменяемости абстракций различного типа. Использование принципа типизации помогает выполнить раннее обнаружение ошибок, упрощает комментирование, помогает генерировать более эффективный код.
Параллелизм - свойство, которое позволяет нескольким абстракциям одновременно находиться в активном состоянии. Данный принцип реализует ОС.
Устойчивость (сохраняемость) - свойство абстракции существовать во времени, независимо от процесса, породившего ее.

20. Объектная декомпозиция. Объектная модель.

Объектная декомпозиция - представление предметной области в виде объектов, взаимодействующих между собой посредством передачи сообщений.
Объектная модель описывает структуру объектов, состоящих из атрибутов, операций и взаимосвязей между ними.

21. Диаграмма классов.

Диаграмма классов демонстрирующая общую структуру иерархии классов системы, их методов, интерфейсов и взаимосвязей между ними, создается на языке моделирования UML.

На диаграмме классы представлены в рамках, содержащих три компонента:

- В верхней части написано имя класса
- Посередине располагаются поля класса.
- Нижняя часть содержит методы класса.

Для задания видимости членов класса, перед каждым членом класса должено быть написано **+**(публичный член класса), **-**(приватный член класса) или **#**(защищенный член класса)

22. Полиморфизм. Инкапсуляция.

Полиморфизм как механизм - возможность задания различных реализаций некоторого единого по названию метода для классов различных уровней иерархии.
Полиморфными называю объекты, которым в процессе выполнения программы можно присвоить переменные, тип которых отличается от текущего.
Различают **статический** (простой) полиморфизм и **виртуальный** (сложный) полиморфизм.Статический реализует раннее связывание (связь метода и реализации на этапе компиляции),виртуальный - поздний (в процессе выполнения программы).
При виртуальном полиморфизме компилятор неявно выполняет механизм позднего связывания, используя таблицу виртуальных методов.

Раннее связывание:

```
class A{
	int a;
public:
	A(int a1),a(a1){};
	void print(){
		std:cout<< "print A";
	}
	void show(){
		std:cout<< "show A";
		print();
	}
}
class B : public A{
	int b;
public:
	B(int b1,int a1),A(a1),b(b1){};
	void print(){
		std:cout<< "print B";
	}
}
class C: public B{
	int c;
public:
	C(int c1, int b1, int a1): B(b1), A(a1), c(c1){}
	void print(){
		std:cout<< "print C";
	}
A a(1); B b(1, 2); C c(1, 2, 3);
a.print();	// print A
a.show();	// show A print A
b.print();	// print B
```

```
b.show();       // show A print A  !!! т.е. результат метода show из класса A
c.print();      // print C
c.show();       // show A print A !!!
```

Позднее связывание:
для реализации позднего связывания используется ключевое слово virtual, которое выполняет неявную постройку позднего связывания и ТВМ.
Если метод print класса A сделать виртуальным, то получим позднее связывание:

```
virtual void print(){
        std::cout << “print A”;
}
A *a = new A(1); B *b = new B(1, 2); C *c = new C(1, 2, 3);
a->print();     // print A
a->show();      // show A print A
b->print();     // print B
b->show();      /* show A print B !!!! метод show не переопределён, поэтому show A, но print является виртуальным и вызывается в show, причём вызывается соответствующий текущему объекту метод print - это и есть виртуальный полиморфизм*/
c->print();     // print C
c>show();       //show A print C !!!
```

Полиморфизм используется:

1) при передаче объекта в качестве параметра функции или метода, фактическим параметром которых является класс - родитель.

```
int draw(A *a){
        a->show();
}
draw(A); draw(B); draw(C);
```

2) Когда объекту по указателю на класс - родитель нужно присвоить объект указатель на класс - потомок.
3) в лекции это неявно. Могу предположить, что речь идёт об абстрактных классах. Абстрактным (или виртуальным) называется класс, который имеет хотя бы один чисто виртуальный метод: virtual void show() = 0; Нельзя создать объект такого класса, но можно наследовать и переопределить чисто виртуальный метод. Абстрактные классы используются как интерфейсы наследуемых классов. Если не переопределить чисто виртуальный метод в дочернем классе, то он тоже будет абстрактным.

```
class Abstract{
        int k;
public:
        Abstract(int k): k(k) {}
```

```
        virtual void show() = 0;
};
class A: public Abstract{
public:
        void show(){
                std::cout << “show A”;
        }
}
Abstract abs(1);        // не скомпилится
Abstract *abs;
abs  = new Abstract(1)          //тоже не лучшая идея
abs = new A(1);         // а это совсем другой разговор
abs->show();            // show A
```

Инкапсуляция – это свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе и скрыть детали реализации от пользователя.

Инкапсуляция неразрывно связана с понятием интерфейса класса. По сути, всё то, что не входит в интерфейс, инкапсулируется в классе.
Или из лекции:
Инкапсуляция - сочетание объединения всех свойств предмета, характеризующих его состояние и поведение в единую абстракцию с ограничением доступа к её реализации.

23. Общая характеристика объектов.

Объект - сущность, способная сохранять своё состояние и обеспечивать набор операций для проверки и изменения этого состояния.
Объект - конкретное представление абстракции
Объект - экземпляр класса.
Структура и поведение объекта описаны в классе.
Каждый объект обладает:

1) Индивидуальность (информацией о характеристиках)) - совокупность характеристик объекта, отличающих его от других объектов.
2) Состояние - характеризуется перечнем свойств и их конкретными значениями.
3) Поведение - описывает как объект взаимодействует с другими объектами или подвергается их воздействию.

Для реализации поведения существуют 5 видов операций над объектами:

1) модификация - операция для изменения состояния объекта.
2) селектор - операция, дающая доступ для определения состояния объекта без его изменения (операция чтения).

3) Итератор - операция доступа к содержанию объекта по частям (в определенной последовательности).
4) Конструктор - операция создания и (или) инициализация объекта.
5) Деструктор - операция разрушения объекта и (или) освобождение занимаемой им памяти.

Объекты бываю пассивные и активный. Активный может изменять своё состояние без внешнего влияния, пассивные не может.

Объекты бывают:

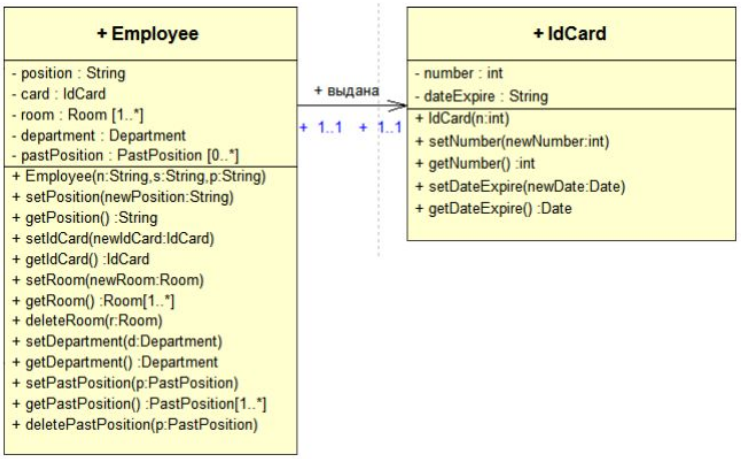
1) Актёр - может изменять другие объекты, но не может быть изменён другими объектами
2) Агент - может менять своё и чужие состояния.
3) Сервер - предоставляет сервисы другим объектам.

Виды отношений:

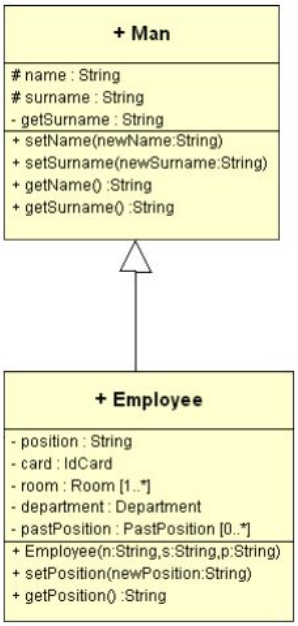
1) Связь - характеризуется передачей сообщений друг другу.
2) Агрегация - один объект включает в себя другой. Агрегация бывает физической и по ссылке.

24. Виды отношений между классами.

1) Ассоциация - один тип объектов ассоциируется с другим.

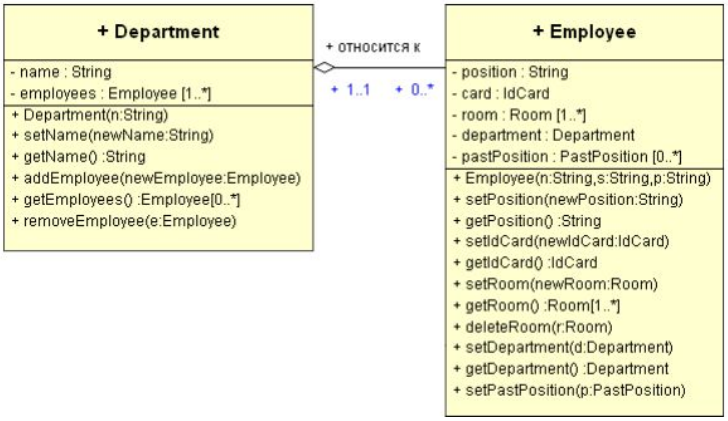


2) Наследование - механизм, позволяющий создавать классы на основе родительских классов.

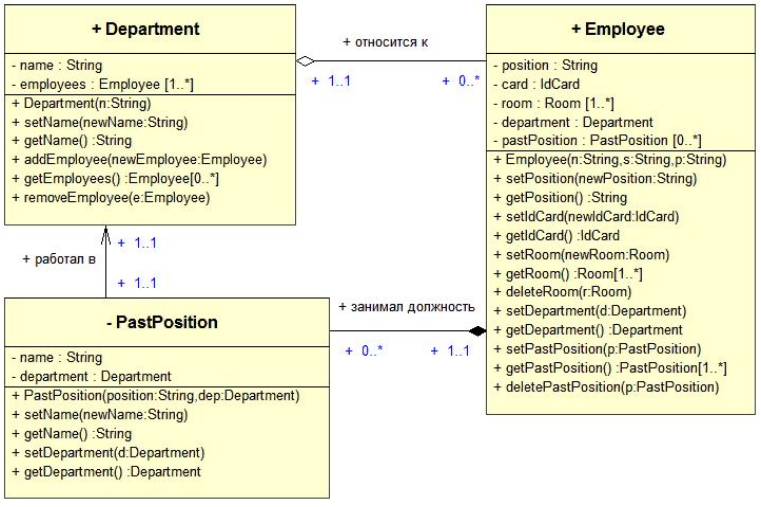


наследование	секция в base	доступ в child
private	private	нет
	protected	private
	public	private
protected	private	нет
	protected	protected
	public	protected
public	private	нет
	protected	protected
	public	public

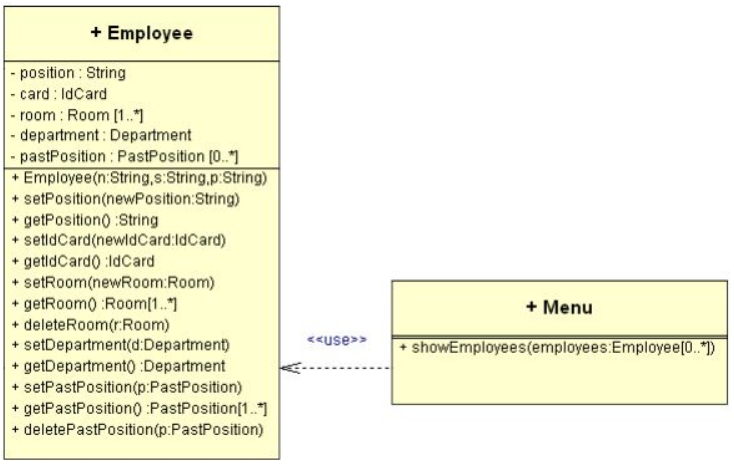
3) Множественное наследование
4) Агрегация - включение по ссылке (необязательное, т.е включаемый объект может существовать без исходного)



5) Композиция - обязательное включение (физическое). Исходный объект не может существовать без включаемого.



6) Зависимость - отношение между классами, при котором один класс использует сервисы другого.




```
class Unit{
        int x,y;
public:
        Unit()ix(1),y(1){};
        void print();
        static void printS(){}
}
class Field{
public:
        void createUnit(Unit &a);
        Unit createUnit(){
                return new Unit();
        }
        void printUnit(){
                Unit.printS();
        }
}
```

25. Классовые поля и методы.

В качестве компонентов в описании класса фигурируют поля, используемые для хранения параметров объектов (переменные), и функции (методы), описывающие правила взаимодействия с ними. Структура класса делится на 3 секции: private (по умолчанию), protected и public, в которых мог быть описаны поля и методы.

Private - внутренние компоненты класса, они доступны только методам этого же класса и дружественным функция и классам.

Protected - защищённые компоненты класса, они доступны методам этого же класса его потомков, дружественным функциям и классам.

Public - общие компоненты, они доступны в любом месте программы. Это интерфейс класса.

Поля класса всегда описываются внутри класса, методы могут быть описаны и внутри и снаружи (прототипы внутри).
При вызове метода, ему неявно передаётся указатель this на объект, для которого вызывается метод.
Для полей класса существует модификатор static, при его применении поле будет общим для всех объектов данного класса, т.е. существует только одна копия этого поля, инициализация статического поля осуществляется вне определения класса.

```
class A{
        int a;
        static int count;
public:
        A(int a): a(a){}
        void func() {a++; count++;}
}
int A::count = 0;
…
```

методы можно задать как константные, явно указываем, что они не могут изменить поля класса: void show() const;
Идентификатором метода является его сигнатура - имя и параметры.
Все методы класса образуют его протокол.

26. Общая характеристика классов.

Класс определяет абстракцию существующего объекта. Абстракция - совокупность существенных характеристик, объекта, которые отличают его от других объектов.
Класс - некоторое множество объектов,имеющих общую структуру и общее поведение.
Класс - структурный пользовательский тип данных, который включает в себя описание полей и функций, называющихся методами.
Описания класса:
class <имя класса> {
private:<описание полей и методов>
protected: <описание полей и методов>
public:<описание полей и методов>
}

27. Обработка исключительных ситуаций в C++.

Для обработки исключений в Си++ используют 3 ключевых слова:

1) try (пытаться) - начало блока исключений;
2) catch (поймать) - начало блока, "ловящего" исключение;
3) throw (бросить) - ключевое слово, "создающее" ("возбуждающее") исключение.

```
try{
        <опасный участок кода>
```

throw <любой тип данных (переменная, константа, литерал, объект)>
}
catch(<принимаемый тип данных от throw>){ <код обработки ошибки> }
catch должен следовать непосредственно после try, throw может кидать исключения любых типов, catch может ловить любые исключения, если написать следующим образом catch(...){}, в одном блоке try может быть несколько throw, выбрасывающие разные исключения, может быть несколько catch подряд, исключение может выбрасываться внутри функции, тогда функция должна помещаться в блок try, после захвата исключения с помощью catch можно выбросить его на уровень выше, чтобы его обработал следующий catch с этим же принимаемым типом, для этого нужно написать throw внутри блока catch. Операторы, следующие за throw никогда не выполняются, после throw выполняется просматривается стек функций с вызовами деструкторов локальных объектов. Блок try-catch может быть вложенным.
Может пригодиться https://metanit.com/cpp/tutorial/6.3.php

28. Обработка общих исключительных ситуаций в C++.

лол? чем отличается от 27?

29. Виды отношений между объектами.

Виды отношений:

1) Связь - характеризуется передачей сообщений друг другу.
2) Агрегация - один объект включает в себя другой. Агрегация бывает физической и по ссылке.

30. Библиотека стандартных классов(STL).

#include <T>, где T — название коллекции.
Содержит набор шаблонов контейнерных классов, алгоритмов и итераторов.
Последовательные контейнеры:

- *Класс vector (вектор) - это динамический массив, способный увеличиваться по мере необходимости для содержания всех своих элементов. Доступ []. А также вставка и удаление.*
- *Класс deque (дек) - это двусторонняя очередь, реализованная в виде динамического массива, который может расти с обоих концов.*
- *Класс List (список) - это двусвязный список, каждый элемент которого содержит 2 указателя: один указывает на следующий элемент списка, а ещё один – на предыдущий элемент списка. list предоставляет доступ только к началу и к концу списка — произвольный доступ запрещён.*

Ассоциативные контейнеры - это контейнерные классы, которые автоматически сортируют все свои элементы:

- *set — это контейнер, в котором хранятся только уникальные элементы, повторения – запрещены. Элементы сортируются в соответствии с их значениями.*
- *multiset — это set, но в котором допускаются повторяющиеся элементы.*
- *map (или ещё «ассоциативный массив») — это set, в котором каждый элемент является парой «ключ-значение». Ключ используется для сортировки и индексации данных и должен быть уникальным. А значение — это фактические данные.*
- *multimap (или ещё «словарь») — это map, который допускает дублирование ключей. Все ключи отсортированы в порядке возрастания, и вы можете посмотреть значение по ключу.*

Адаптеры — это специальные предопределенные контейнерные классы, которые адаптированы для выполнения конкретных заданий:

- *stack (стек) — это контейнерный класс, элементы которого работают по принципу LIFO («Last In, First Out» – «Последним Пришёл, Первым Ушёл»)*
- *queue (очередь) — это контейнерный класс, элементы которого работают по принципу FIFO(«First In, First Out» – «Первым Пришёл, Первым Ушёл»)*
- *priority_queue (очередь с приоритетом) — это тип очереди, в которой все элементы отсортированы. При вставке элемента, он автоматически сортируется.*

Итераторы.

Итератор — это объект, который способен перебирать элементы контейнерного класса без необходимости пользователю знать, как реализован определенный контейнерный класс.

Типы итераторов:

container::iterator — итератор для чтения/записи;

container::const_iterator — итератор только для чтения.

Функционал:

*Оператор * возвращает элемент, на который в данный момент указывает итератор.*

Оператор ++ (--) перемещает итератор к следующему элементу контейнера.

Операторы == и != используются для определения того, указывают ли два итератора на один и тот же элемент или нет. Для сравнения значений используется разыменование.

Оператор = присваивает итератору новую позицию (обычно начало или конец элементов контейнера). Для изменения значения используется разыменование.

Каждый контейнерный класс имеет 4 основных метода для работы с оператором =:

begin() возвращает итератор, представляющий начало элементов контейнера.

end() возвращает итератор, представляющий элемент, который находится после последнего элемента в контейнере.
cbegin() возвращает константный (только для чтения) итератор, представляющий начало элементов контейнера.
cend() возвращает константный (только для чтения) итератор, представляющий элемент, который находится после последнего элемента в контейнере.
Агоритмы STL: Работа с элементами контейнера
1. min_element – наименьший лемент
2. max_element – наибольший элемент
3. find – поиск элемента
4. list::insert – добавление элемента
5. sort – сортировка контейнера
6. reverse – переворот контейнера

31. Полиморфизм. Виды полиморфизма в C++.

см. вопрос №21

32. Шаблоны классов, template.

см. вопрос №30

33. Контейнеры STL. Основные методы.

см. вопрос №30

34. Стандартная библиотека STL (map, list)

см. вопрос №30

35. Стандартная библиотека STL (vector, stack)

см. вопрос №30

36.Стандартная библиотека STL (set, queue)

см. вопрос №30

37. Многопоточное программирование в C++. Класс Thread.

Многопоточность – свойство платформы или приложения, состоящее в том, что процесс, порожденный в операционной системе, может состоять из нескольких потоков, выполняющихся параллельно, то есть без предписанного порядка по времени.

Класс Thread позволяет создать и запустить дополнительный поток, который будет работать параллельно с основным потоком.
std::thread thr(threadFunction)
thr – это объект, представляющий поток, в котором выполняется функция threadFunction. Передача переменных в функцию производится с помощью дополнительных аргументов, добавляемых в конструктор после указателя на функцию. Для передачи ссылки, переменную необходимо обернуть функцией res(). Вызов `join` блокирует вызывающий поток до тех пор, пока функция потока не выполнит свою работу. `detach` позволяет отсоединить поток от объекта, иными словами, сделать его фоновым. К отсоединенным потокам больше нельзя применять `join`.

38. Объектная декомпозиция.

Объектная декомпозиция – процесс представления предметной области задачи в виде совокупности объектов, обменивающихся сообщениями.

39. Потоки данных в C++. Класс ios и его наследники.

Класс istream используется для работы с входными потоками. Оператор извлечения >> используется для извлечения значений из потока. Это имеет смысл: когда пользователь нажимает на клавишу клавиатуры, код этой клавиши помещается во входной поток. Затем программа извлекает это значение из потока и использует его.
Класс ostream используется для работы с выходными потоками. Оператор вставки << используется для помещения значений в поток. Это также имеет смысл: вы вставляете свои значения в поток, а затем потребитель данных (например, монитор) использует их.
Класс iostream может обрабатывать как ввод, так и вывод данных, что позволяет ему осуществлять двунаправленный ввод/вывод.
Наконец, остались 3 класса, оканчивающихся на «_withassign». Эти потоковые классы являются дочерними классам istream, ostream и iostream (соответственно). В большинстве случаев вы не будете работать с ними напрямую.
cin – класс istream_withassign, связанный со стандартным вводом (обычно это клавиатура);
cout – класс ostream_withassign, связанный со стандартным выводом (обычно это монитор);
cerr – класс ostream_withassign, связанный со стандартной ошибкой (обычно это монитор), обеспечивающий небуферизованный вывод;
clog – класс ostream_withassign, связанный со стандартной ошибкой (обычно это монитор), обеспечивающий буферизованный вывод.

40. Организация процесса ввода/вывода в C++.

41. Виды отношений между объектами. Типы объектов.

Виды отношений:

- Связь – передача сообщений от 1 объекта другому
- Агрегация – один объект включает в себя другие (физическая или вкл. по ссылке)

Типы объектов:

- Актер – может выполнять изменение состояние других объектов
- Агент – изменяет свое и чужие состояния
- Сервер – предоставляет сервисы другим объектам.

42. Виды иерархий.

Иерархическая организация: использование иерархий при разработке программной системы упрощает системы абстракций.
Иерархия - упорядочение абстракций, расположение их по уровням.
Классификация:

- Часть/целое - предполагается, что некоторая абстракция включает другую. Используется в ранних этапах проектирования. (На логическом уровне при разбиении предметной области на объектном уровне).
- Общее/частное - некоторая абстракция является частным случаем другой. Используется в основном механизме ООП - наследовании.

43. Виды операция над объектами.

Объект - сущность, способная сохранять свое состояние и обеспечивать набор операций для проверки и изменения этого состояния.
Также, объект - конкретное представление абстракции или экземпляр класса.
Каждый объект обладает:

- Индивидуальность (информация о характеристиках)
- Состояние
- Поведение

Для реализации поведения существуют:

- Модификация
- Селектор
- Итератор
- Конструктор
- Деструктор

Данные операции объявляются как методы. Все методы класса образуют его протокол (допустимое поведение объекта)

44. Дружественные элементы. Ключевое слово friend.

Дружественные элементы:

- Дружественные функции - это функция, которая имеет доступ к закрытым членам класса, как если бы она сама была членом этого класса. Во всех других отношениях дружественная функция является обычной функцией. Ею может быть как обычная функция, так и метод другого класса. Для объявления дружественной функции используется ключевое слово friend перед прототипом функции, которую вы хотите сделать дружественной классу. Неважно, объявляете ли вы её в public или в private зоне класса.
- Дружественные классы - один класс можно сделать дружественным другому классу. Это откроет всем членам первого класса доступ к закрытым членам второго класса
- Дружественные методы - вместо того, чтобы делать дружественным целый класс, мы можем сделать дружественными только определённые методы класса.

45. Дружественные функции.

Дружественная функция — это функция, которая имеет доступ к закрытым членам класса, как если бы она сама была членом этого класса. Во всех других отношениях дружественная функция является обычной функцией. Ею может быть как обычная функция, так и метод другого класса. Для объявления дружественной функции используется ключевое слово friend перед прототипом функции, которую вы хотите сделать дружественной классу. Неважно, объявляете ли вы её в public или в private зоне класса.

46. Перегрузки операторов

https://habr.com/ru/post/132014/

47. Виды операторов в C++ и особенности их перегрузки.

Виды операторов:

- Арифметические (=, +a, -a, a+b, a-b, *, /, %, ++a, a++, - -a,a - -)
- Сравнения (==, !=, >, <, >=, <=)
- Логические (!, &&, ||)
- Побитовые (~, &, |, ^, <<, >>)
- Составное присваивание(+=, -=, *=, /=,%=, &=, |=, ^=, <<=, >>=,)
- Операторы работы с указателями и членами класса ([], *a, &a, a->b, a.b)
- Другие (a(a1, a2), (a, b), a ? b : c, sizeof(), alignof(), (type)a)

Нельзя перегрузить:

- Тернарный оператор (a?b:c)
- Оператор sizeof()
- Оператор разрешения области видимости (::)
- Оператор выбора члена (.)
- Определение или разыменовывание указателя (*)

Изначальное количество операндов, поддерживаемых оператором, изменить невозможно

Все операторы сохраняют свой приоритет и ассоциативность по умолчанию

48. Исключительные ситуации.

Исключение – события при выполнении программы, которое приводит к её ненормальному или неправильному поведению.
Существуют исключения:

- Аппаратные – генерируются процессором (деление на 0, выход за границы массива, обращение к невыделенной памяти и др.)
- Программные – генерируются операционной системой и прикладными программами (возникают, когда программа их явно инициирует)

Обработка исключений:
Для обработки исключений используется 3 выражения: try, throw и catch
Блок try {…} включает операторы, которые могут создавать исключения.
Выражение throw используется, когда исключение в блоке try произошло, оно передаёт информацию об ошибке в блок catch
Блок catch находится сразу после блока try, его аргументом может быть что угодно или ничего (…), в этом блоке обрабатывается исключение. catch'ей может быть несколько.

49. Принципы SOLID (SO).

S - single responsibility - Принцип единственной ответственности, иначе говоря, "Существует только одна причина, чтобы существовал данный класс". Каждый класс выполняет только одну задачу.
O - open/close - Программные элементы должны быть открытыми для расширения и закрыты для модификации.

50. Принципы SOLID (LI).

L - принцип Барбары Лисков - программные компоненты должны иметь возможность быть замененными на экземпляры их подтипов без изменения основной части.
I - interface segregation - "Лучше много маленьких интерфейсов, чем один большой". Принцип разделения интерфейсов говорит о том, что слишком «толстые» интерфейсы необходимо разделять на более маленькие и специфические, чтобы программные сущности маленьких интерфейсов знали только о методах, которые необходимы им в работе. В итоге, при изменении метода интерфейса не должны меняться программные сущности, которые этот метод не используют.

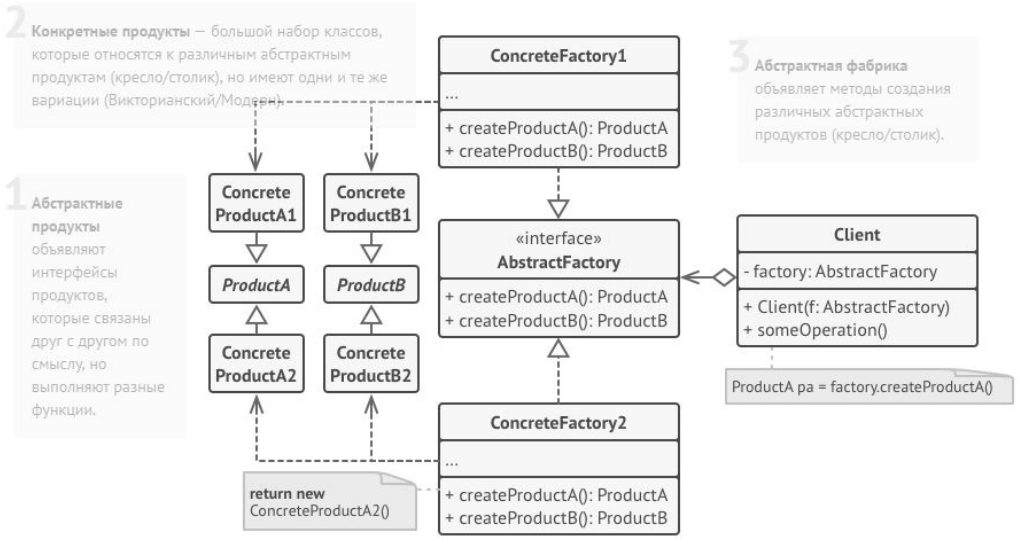
51. Принципы SOLID (D).

D - dependency inversion - реализация должна зависеть от интерфейса а не наоборот
ИЛИ
D - dependency inversion; инверсия зависимостей. Все должно зависеть от абстракций, но не от их экземпляров.

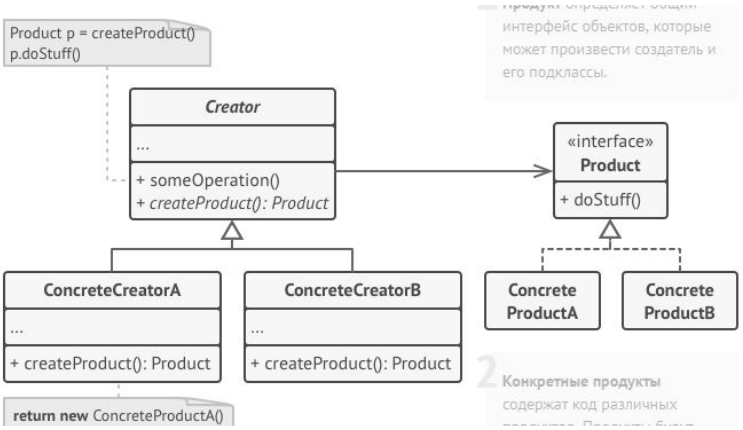
52. Шаблоны проектирования. Порождающие шаблоны.

Отвечают за удобное и безопасное создание новых объектов или даже целых семейств объектов
Бывают:

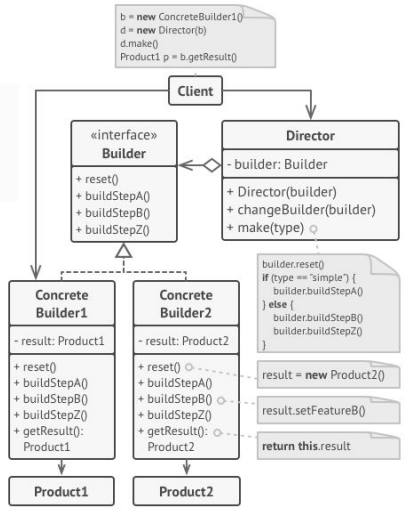
- Абстрактная фабрика
  Назначение: интерфейс для создания семейств взаимосвязанных объектов или взаимозависимых объектов, не специфицируя их конкретных классов
  Изолирует классы (“Продукты”)Б упрощает замену семейств продуктов, Гарантирует сочетаемость продуктов, но сложно добавить поддержку нового продукта.
  Можно использовать, когда система должна оставаться независимой как от процесса создания новых объектов, так и от типов порождающих объектов (new нежелателен внутри клиента); Когда необходимо создать группы или семейства взаимосвязанных объектов, исключая возможность одновременного использования объектов разных семейств в одном контексте.



- Фабричный метод
  Назначение: система остается расширяемой при добавлении новых типов объектов.
  Позволяет системе оставаться независимой от процесса порождения объектов. Заранее известен момент создания объекта, но неизвестен тип.



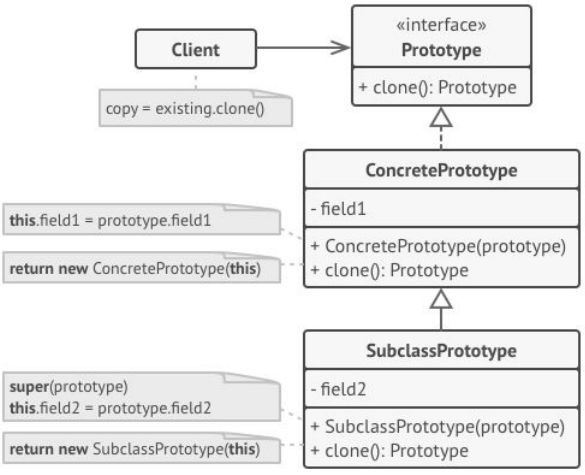
- Строитель
  Назначение: применим, когда каждый экземпляр класса отличается от другого (например, двухэтажный и трехэтажный дом: оба дома, но один этаж придется достроить).



- Сингтон
  Назначение: создает 1 экземпляр класса в рамках 1 процесса, обеспечение доступа к его экземпляру в любом месте программы.
  Гарантирует наличие единственного экземпляра класса, предоставляет к нему глобальную точку доступа, реализует отложенную инициализацию объекта-одиночки, НО нарушает принцип единственной ответственности класса,
  маскирует плохой дизайн, проблемы мультипоточности, требует постоянного создания Mock-объектов при юнит-тестировании.

- Прототип

  Назначение:
  необходим для создания точной копии экземпляра класса методом clone().



53. Шаблоны проектирования. Структурные шаблоны.

Отвечают за построение удобных в поддержке иерархий классов. Среди них:
Адаптер - позволяет объектам с несовместимыми интерфейсами работать вместе.
Мост - разделяет один или несколько классов на две отдельные иерархии — абстракцию и реализацию, позволяя изменять их независимо друг от друга.
Компоновщик - позволяет сгруппировать множество объектов в древовидную структуру, а затем работать с ней так, как будто это единичный объект.
Декоратор позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».
Фасад предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку.
Легковес позволяет вместить бóльшее количество объектов в отведённую оперативную память. Легковес экономит память, разделяя общее состояние объектов между собой, вместо хранения одинаковых данных в каждом объекте.
Заместитель позволяет подставлять вместо реальных объектов специальные объекты-заменители. Эти объекты перехватывают вызовы к оригинальному объекту, позволяя сделать что-то *до* или *после* передачи вызова оригиналу.

54. Шаблоны проектирования. Шаблоны поведения.

Поведенческие **шаблоны** — **шаблоны проектирования**, определяющие алгоритмы и способы реализации взаимодействия различных объектов.
1. Паттерн Chain of Responsibility позволяет обработать запрос нескольким объектам-получателям. Получатели связываются в цепочку, и запрос передается по цепочке, пока не будет обработан каким-то объектом. Паттерн Chain of Responsibility позволяет также избежать жесткой зависимости между отправителем запроса и его получателями.
2. Паттерн Command преобразовывает запрос на выполнение действия в отдельный объект-команду. Это придает системе гибкость: позволяет осуществлять динамическую замену команд, использовать сложные составные команды, осуществлять отмену операций.
3. Паттерн Iterator предоставляет механизм обхода элементов составных объектов (коллекций) не раскрывая их внутреннего представления.
4. Паттерн Interpreter предназначен для решения повторяющихся задач, которые можно описать некоторым языком. Для этого паттерн Interpreter описывает решаемую задачу в виде предложений этого языка, а затем интерпретирует их.
5. Паттерн Mediator инкапсулирует взаимодействие совокупности объектов в отдельный объект-посредник. Уменьшает степень связанности взаимодействующих объектов - им не нужно хранить ссылки друг на друга.
6. Паттерн Memento получает и сохраняет за пределами объекта его внутреннее состояние так, чтобы позже можно было восстановить объект в таком же состоянии.

7. Паттерн Observer определяет зависимость "один-ко-многим" между объектами так, что при изменении состояния одного объекта все зависящие от него объекты уведомляются и обновляются автоматически.
8. Паттерн State позволяет объекту изменять свое поведение в зависимости от внутреннего состояния. Создается впечатление, что объект изменил свой класс. Паттерн State является объектно-ориентированной реализацией конечного автомата.
9. Если поведение системы настраивается согласно одному из некоторого множества алгоритму, то применение паттерна Strategy переносит семейство алгоритмов в отдельную иерархию классов, что позволяет заменять один алгоритм другим в ходе выполнения программы. Кроме того, такую систему проще расширять и поддерживать.
10. Паттерн Template Method определяет основу алгоритма и позволяет подклассам изменить некоторые шаги этого алгоритма без изменения его общей структуры. 11. Паттерн Visitor определяет операцию, выполняемую на каждом элементе из некоторой структуры без изменения классов этих объектов.

55. QT. Слоты и сигналы.

Сигналы и слоты — подход, используемый в Qt, который позволяет реализовать шаблон «наблюдатель», минимизируя написание повторяющегося кода. Концепция заключается в том, что компонент может посылать сигналы, содержащие информацию о событии. В свою очередь другие компоненты могут принимать эти сигналы посредством специальных функций — слотов. Система сигналов и слотов хорошо подходит для описания Графического интерфейса пользователя.

56. Python. Особенности интерпретируемых языков программирования.

Если упрощенно, то интерпретируемые языки выполняются по схеме “считывается строка -> переводится в машинный код -> исполняется -> считывается строка …”
Компилируемые - сначала полный перевод в машинный код, потом исполнение
Интерпретируемые языки позволяют изменять код прямо во время выполнения, не ожидать компиляции
Однако компилируемый язык позволяет проверить валидность программы в ходе компиляции, в то время как интерпретируемый выдаст ошибку только при исполнении.

57. Python. Работа с вводом/выводом.

Основные функции ввода и вывода в питоне - input(), print().
input в качестве параметра принимает приглашение
>>> input(“Введите число: “)
Возвращаемый тип input() - строка. Для преобразования строки в число используется приведение:
a = int(input())

b = float(input())
Для нескольких элементов можно разбить ввод на подстроки и обработать каждую
a, b = map(int, input().split()) - здесь ввод (input()) разбивается по пробелам на список подстрок (split()), а затем функция map применяет к каждой строке из списка строк преобразование int()

58. Python. Базовые типы данных.

Надо заметить, что абсолютно все в питоне является объектом. int и float тоже.

None (неопределенное значение переменной)

Логические переменные (*Boolean Type*)

Числа (*Numeric Type*)

a. *int* – целое число
b. *float* – число с плавающей точкой
c. *complex* – комплексное число

Списки (*Sequence Type*)

d. *list* – список [ ]
e. *tuple* – кортеж ()
f. *range* – диапазон

Строки (*Text Sequence Type*)

g. *str* *“ ”, ‘ ‘, ‘’’ ‘’’, “”” “”” (Строки могут обозначаться апострофами, кавычками, тройными апострофами, тройными кавычками)*

Бинарные списки (*Binary Sequence Types*) - вряд ли кто спросит

h. *bytes* – байты
i. *bytearray* – массивы байт
j. *memoryview* – специальные объекты для доступа к внутренним данным объекта через protocol buffer

Множества (*Set Types*)

k. *set* – множество Инициализация пустого множества только с помощью set(), непустое множество можно записать как {1, 2, 3}
l. *frozenset* – неизменяемое множество

Словари (*Mapping Types*)

m. *dict* – словарь {}

59. Python. Кортежи, строки.

Кортеж - неизменяемая структура данных, аналогичная списку. Может хранить последовательность элементов одного или разных типов. Кортеж можно создать, вызвав конструктор tuple() или записав в круглых скобках объекты, которые надо включить в кортеж. Если в кортеже один элемент, необходимо добавить после него запятую

```
a = (1, 2)
b = (1,)
```

Стоит заметить, что сам кортеж неизменяем, но элементы внутри него, например, списки - изменяемы

Строка - объект класса str; строка представляет собой последовательность символов и поддерживает большое количество методов из списков. Например, доступ по индексу и срезы. Надо сказать, что каждый символ в строке тоже является строкой, т.е., одна буква в питоне - это строка. Типов вроде char в питоне нет.
Строки поддерживают конкатенацию, умножение.

```
a = “Лупа”
b = “Пупа”
print(a + b) # ЛупаПупа
print(3*a) # ЛупаЛупаЛупа
```

Строка может содержать экранированные спецсимволы “\n\r\t и т.д.”
Если перед строкой добавить r, экранирование будет подавлено (raw-строка)

```
r”C:\temp”
```

Если перед строкой добавить b, строка станет строкой байтов (как строка char’ов в си)

```
b”byte”
```

60. Python. Справочники (dict?), списки.

dict - тип данных, представляющий собой набор пар ключ-значение. Вообще является хеш-таблицей.

```
d = dict()
d = {}
d = {1 : ‘a’, 2 : ‘b’}
```

В качестве ключа можно использовать только объекты, которые поддерживают хеширование (изменяемые объекты хеширование не поддерживают)
Получить объект по ключу:

```
a = d[1]
```

Обновить значение:

```
d[1] = a
```

Добавить значение:

```
d[3] = ‘Hello’
```

Чтобы добавить новое значение, надо просто обратиться по новому ключу
Определить наличие элемента
print(2 in d) # True
Длина словаря
len(d)

Список (list) - изменяемый тип данных, представляющий собой последовательность любых элементов. В отличие от массивов, может содержать любые данные и имеет динамический размер.
l = list()
l = [1, 2]
l = []
l = [0]
Поддерживает доступ по индексам
l[0]
В том числе отрицательные индексы
l[-2] - предпоследний элемент
Поддерживает срезы
l[start:finish:step], l[::2] - каждый второй, l[:] - весь список
Можно создать список генератором списков
l = [i ** 2 for i in range(10)]
Некоторые методы:
append(x) - добавление в конец списка
extend(l) - расширение списка списком l (конкатенация)
insert(i, x) - вставка x на i-е место
remove(x) - удаление x
pop() - удаление последнего
sort() - сортировка
clear() - очистка

61. Python. Классы.

Объявление класса в питоне выглядит так:
class ИмяКласса (РодительскийКласс):
    тело

```
class A:
  a = 0
        def go(self):
            print('Go, A!')

    class B(A):
        pass
```

Важным обстоятельством является то, что у классов в питоне нет областей

видимости. Все члены класса являются открытыми (public). По соглашению, имена, начинающиеся с __, считаются закрытыми и их не следует изменять, но это не является синтаксическим правилом и остается на совести программиста.

Методы являются в целом обычными функциями, их основное отличие от других функций - обязательно наличие хотя бы одного параметра. При вызове метода первым параметром в метод передается объект-владелец. Первый параметр обычно называется self

```
class A:
   s = “Hello“
   def say(self, name):
      print(self.s + name)

a = A()
```

Можно вызвать метод say так:
a.say(“Лупа”)
или так:
say(a, “Пупа”)
Эти способы равнозначны

Также можно использовать статические методы и методы класса

```
class A:
        @staticmethod
        def sm(x, y):    # Статические методы не получают ссылку на сам объект
                pass     # Это означаетчто они не имеют доступ к полям и методам
        @classmethod
        def cm(cls, x, y):  # Классовый метод получает первым параметром класс
                pass
```

62. Python. Перегрузка операторов.

Для начала надо сказать, что сигнатура программного объекта в питоне включает в себя только имя, параметры не учитываются.
Классы содержат в себе “магические” методы,которые вызываются при выполнении действий над классами. Например выражение a < b приводит к выполнению кода a.__lt__(b), а выражение a + b - к a.__add__(b).
Перегрузка операторов сводится к переопределению магических методов.
Примеры таких магических методов:
__init__(self[, ...]) - как уже было сказано выше, конструктор.
__del__(self) - вызывается при удалении объекта сборщиком мусора.
__str__(self) - Возвращает строковое представление объекта.
__lt__(self, other) - x < y вызывает x.__lt__(y).
__le__(self, other) - x ≤ y вызывает x.__le__(y).
__eq__(self, other) - x == y вызывает x.__eq__(y).

__ne__(self, other) - x != y вызывает x.__ne__(y)
__gt__(self, other) - x > y вызывает x.__gt__(y).
__ge__(self, other) - x ≥ y вызывает x.__ge__(y).
__add__(self, other) - сложение. x + y вызывает x.__add__(y).
__sub__(self, other) - вычитание (x - y).
__mul__(self, other) - умножение (x * y).
__truediv__(self, other) - деление (x / y).
__floordiv__(self, other) - целочисленное деление (x // y).
__mod__(self, other) - остаток от деления (x % y).
__divmod__(self, other) - частное и остаток (divmod(x, y)).
__pow__(self, other[, modulo]) - возведение в степень (x ** y, pow(x, y[, modulo])).
__lshift__(self, other) - битовый сдвиг влево (x << y).
__rshift__(self, other) - битовый сдвиг вправо (x >> y).
__and__(self, other) - битовое И (x & y).
__xor__(self, other) - битовое ИСКЛЮЧАЮЩЕЕ ИЛИ (x ^ y).
__or__(self, other) - битовое ИЛИ (x | y).

Пример:

```
class Vector2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __str__(self):
        return '({}, {})'.format(self.x, self.y)
    def __add__(self, other):
        return Vector2D(self.x + other.x, self.y + other.y)
    def __abs__(self):
        return math.hypot(self.x, self.y)
    def __bool__(self):
        return self.x != 0 or self.y != 0
    def __neg__(self):
        return Vector2D(-self.x, -self.y)
```

63. Python. Передача в функцию переменного числа аргументов.

Функции в Python имеют аргументы двух видов: именованные и позиционные.
Позиционные - привычные аргументы типа:

```
def f(x, y):
    pass
f(1, 2) # x = 1, y = 2
```

Именованные аргументы позволяют передавать аргументы не по порядку, а по именам. Чаще всего именованные аргументы могут выступать и как позиционные

```
def f(x = 0, y = 1):
    pass
f(1, 2) # x = 1, y = 2
f(y = 2, x = 1) # x = 1, y = 2
```

f(y = 2) #x = 0, y = 2
Чтобы передать в функцию переменное количество позиционных аргументов, необходимо указать аргумент со звездочкой. Тогда по этому параметру в функции будет доступен кортеж из аргументов

```
def f(x, *args):
    print(x, args)
f(1)   # 1, ()
f(1, 2) # 1, (2)
f(1, 2, 3) # 1, (2, 3)
```

Чтобы передать переменное количество позиционных аргументов, указывается аргумент с двумя звездочками. Этот параметр будет содержать словарь пар имя_аргумента: значение

```
def f(**kwargs):
    print(kwargs)
f(a = 5, b = 7) # {a : 5, b: 7}
```

64. Python. PEP8.

PEP8 является руководством по написанию кода на Python
Основная идея руководства заключается в том, что код пишется один раз, а читается - много раз. Это руководство не является частью синтаксиса питон, а просто является соглашением по стилю.

Здесь приведен PEP8, не думаю, что имеет смысл все переписывать
https://pythonworld.ru/osnovy/pep-8-rukovodstvo-po-napisaniyu-koda-na-python.html

65. Обработка исключительных ситуаций в Python.

Для обработки исключительных ситуаций Python использует операции try except (else finally)
Синтаксис:

```
try:
    Код, который может выкинуть исключение
except имя_исключения:
    Обработка исключения
else:
    Выполняется, если исключения не произошло
finally:
    Выполняется всегда
```

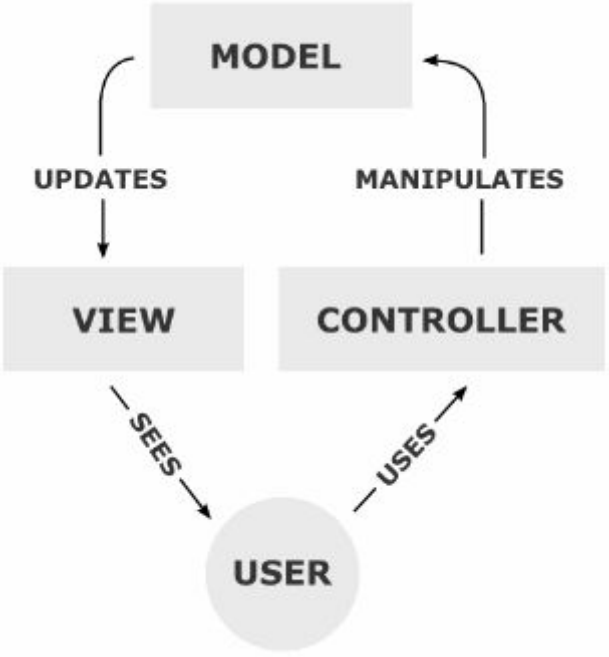
else и finally опциональны
Можно указать except без имени исключения, тогда будут перехватываться вообще все исключения, а также прерывание с клавиатуры и системный выход. Однако для этих целей лучше указать except Exception, перехватывая базовые исключения или его

наследников.

66. MVC

MVC (Model-View-Controller) - шаблон проектирования, предлагающий разделение компонентов приложения на данные, пользовательский интерфейс и управляющую логику.
Модель (*Model*) предоставляет данные и реагирует на команды контроллера, изменяя своё состояние.
Представление (*View*) отвечает за отображение данных модели пользователю, реагируя на изменения модели.
Контроллер (*Controller*) интерпретирует действия пользователя, оповещая модель о необходимости изменений.

Схема работы составляющих приложения:



Преимущества такого подхода:

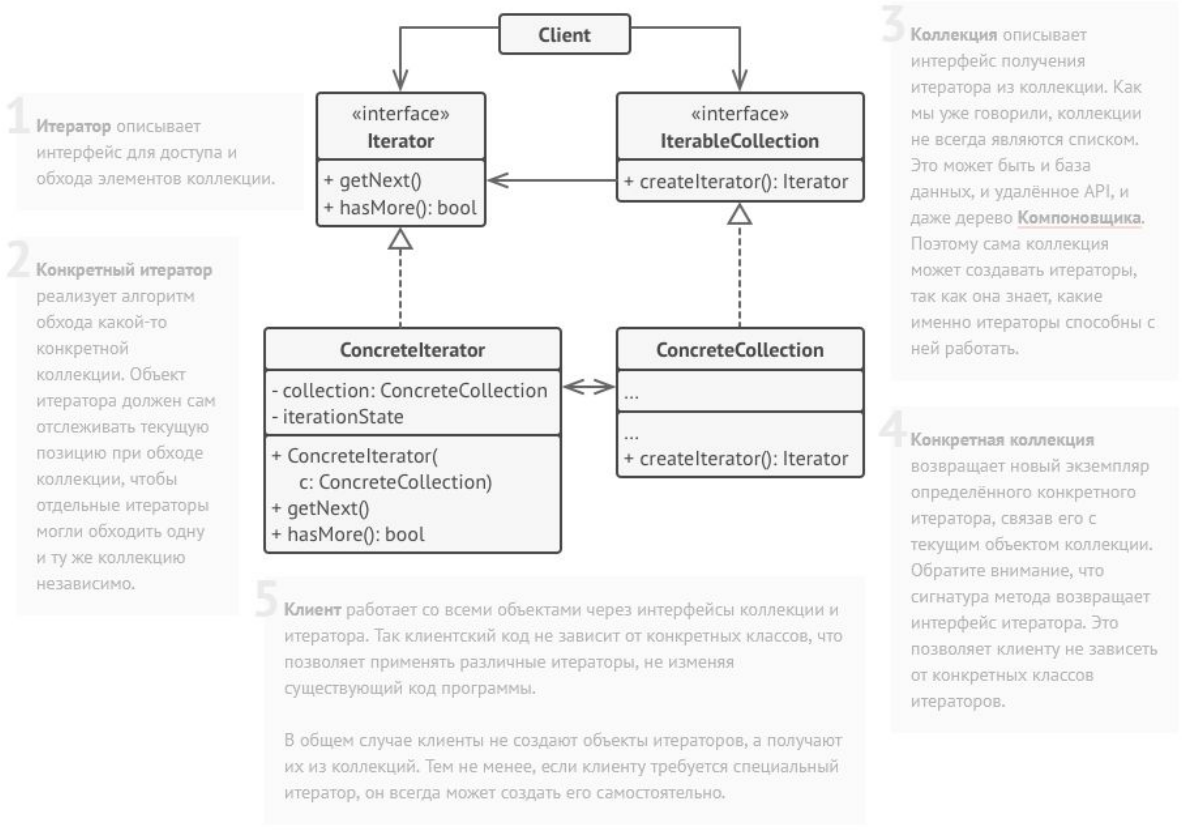
- К одной *модели* можно присоединить несколько ***видов***, при этом не затрагивая реализацию *модели*. Например, некоторые данные могут быть одновременно представлены в виде электронной таблицы, гистограммы и круговой диаграммы;
- Не затрагивая реализацию ***видов***, можно изменить реакции на действия пользователя (нажатие мышью на кнопке, ввод данных) — для этого достаточно использовать другой *контроллер*;
- Ряд разработчиков специализируется только в одной из областей: либо разрабатывают графический интерфейс, либо разрабатывают бизнес-логику. Поэтому возможно добиться того, что программисты, занимающиеся разработкой

бизнес-логики (*модели*), вообще не будут осведомлены о том, какое *представление* будет использоваться.

67. Шаблон проектирования. Итератор.

https://refactoring.guru/ru/design-patterns/iterator

Итератор — это поведенческий паттерн проектирования, который даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.



Зачем?

? - Когда у вас есть сложная структура данных, и вы хотите скрыть от клиента детали её реализации (из-за сложности или вопросов безопасности).

! - Итератор предоставляет клиенту всего несколько простых методов перебора элементов коллекции. Это не только упрощает доступ к коллекции, но и защищает её данные от неосторожных или злоумышленных действий.

? - Когда вам нужно иметь несколько вариантов обхода одной и той же структуры данных.

! - Нетривиальные алгоритмы обхода структуры данных могут иметь довольно объёмный код. Этот код будет захламлять всё вокруг — будь то сам класс коллекции или часть бизнес-логики программы. Применив итератор, вы можете выделить код обхода структуры данных в собственный класс, упростив поддержку остального кода.

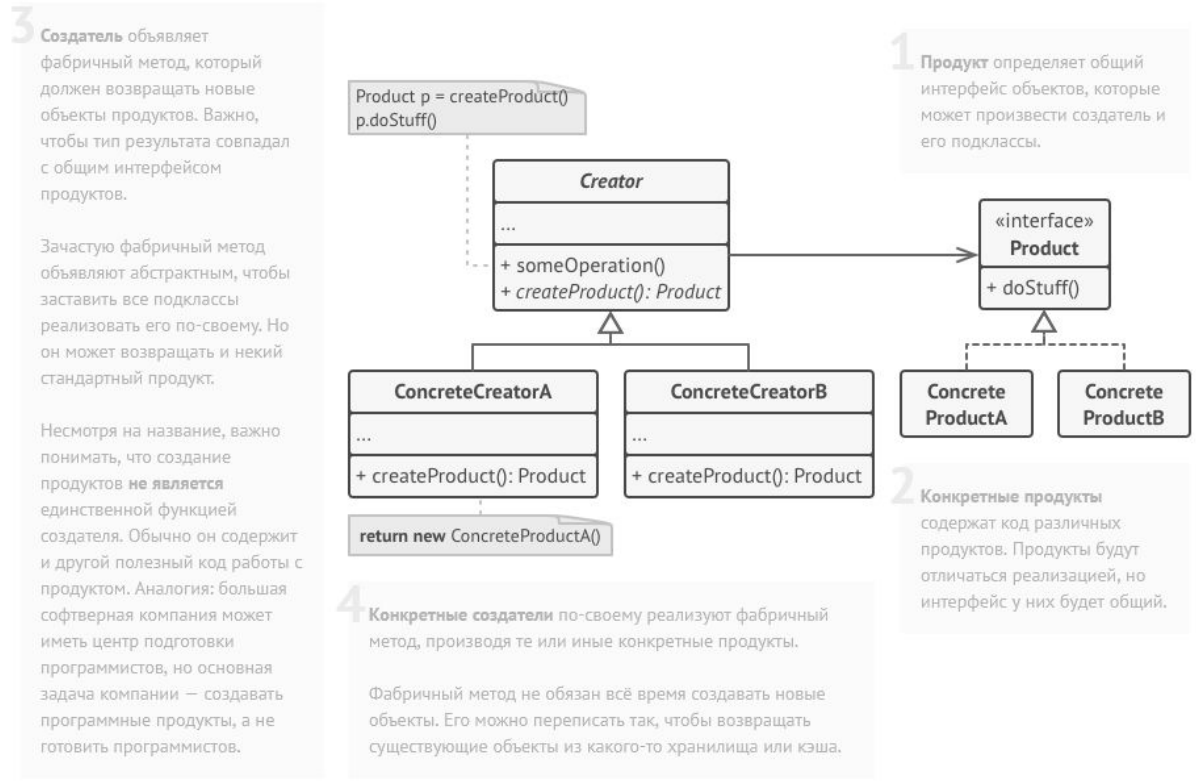
? - Когда вам хочется иметь единый интерфейс обхода различных структур данных.

! - Итератор позволяет вынести реализации различных вариантов обхода в подклассы. Это позволит легко взаимозаменять объекты итераторов, в зависимости от того, с какой структурой данных приходится работать.

68. Шаблон проектирования. Фабричный метод.

https://refactoring.guru/ru/design-patterns/factory-method

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



Зачем?

? - 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

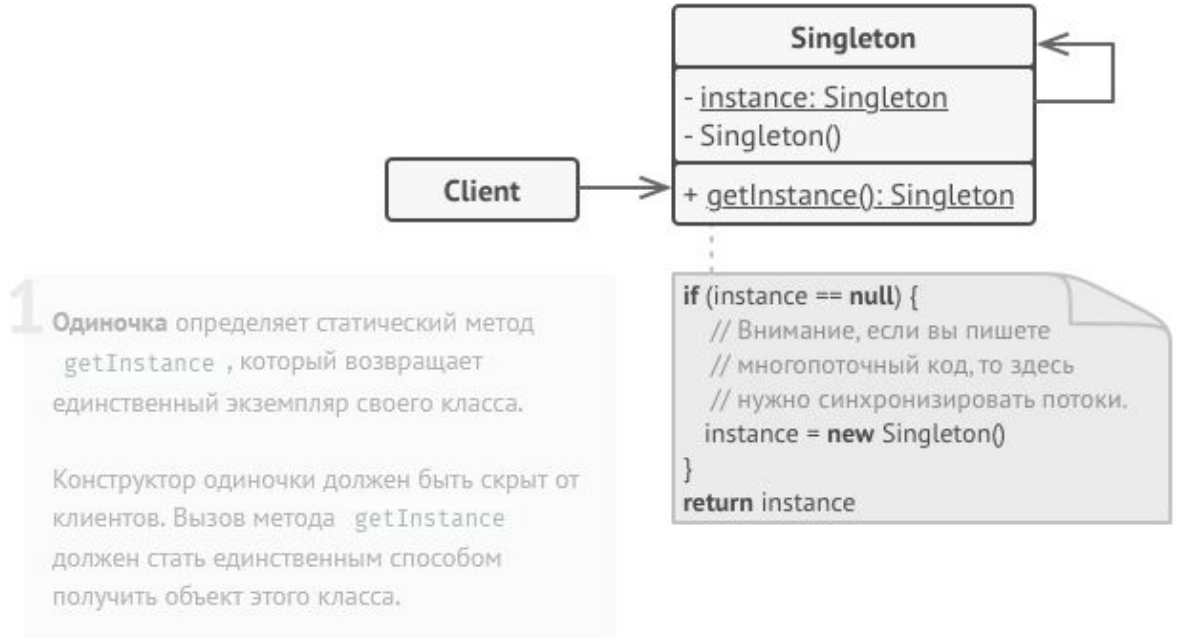
фреймворка UIFramework, чтобы он теперь создавал круглые кнопки, вместо стандартных? Для этого вы создаёте подкласс UIWithRoundButtons из базового класса фреймворка, переопределяете в нём метод создания кнопки (а-ля createButton) и вписываете туда создание своего класса кнопок. Затем используете UIWithRoundButtons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

69. Шаблон проектирования. Одиночка (Singleton).

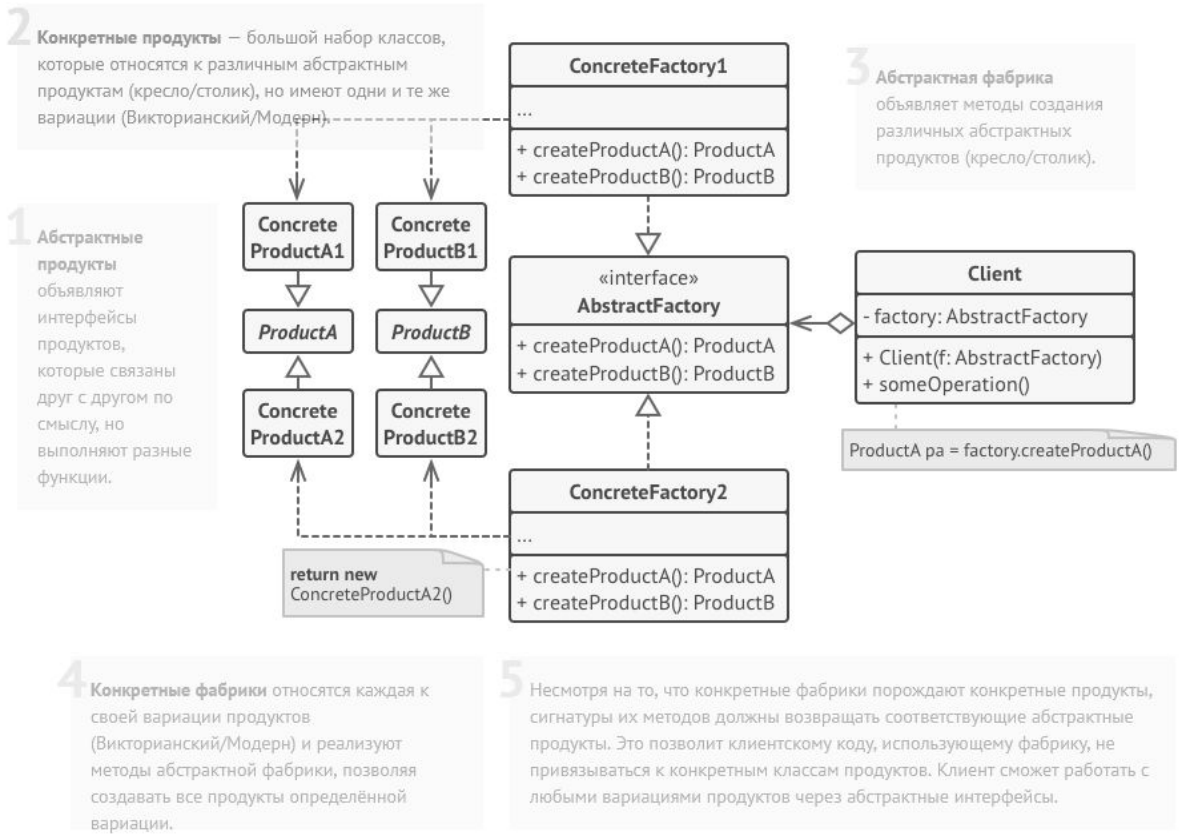
https://refactoring.guru/ru/design-patterns/singleton
Одиночка — это порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.



Зачем?
? - Когда в программе должен быть единственный экземпляр какого-то класса, доступный всем клиентам (например, общий доступ к базе данных из разных частей программы).
! - Одиночка скрывает от клиентов все способы создания нового объекта, кроме специального метода. Этот метод либо создаёт объект, либо отдаёт существующий объект, если он уже был создан.
? - Когда вам хочется иметь больше контроля над глобальными переменными.
! - В отличие от глобальных переменных, Одиночка гарантирует, что никакой другой код не заменит созданный экземпляр класса, поэтому вы всегда уверены в наличии лишь одного объекта-одиночки.
Тем не менее, в любой момент вы можете расширить это ограничение и позволить любое количество объектов-одиночек, поменяв код в одном месте (метод getInstance).

70. Шаблон проектирования. Абстрактная фабрика.

https://refactoring.guru/ru/design-patterns/abstract-factory
Абстрактная фабрика — это порождающий паттерн проектирования, который позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.



Зачем?

? - 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.

71 Шаблон проектирования. Прототипирование.

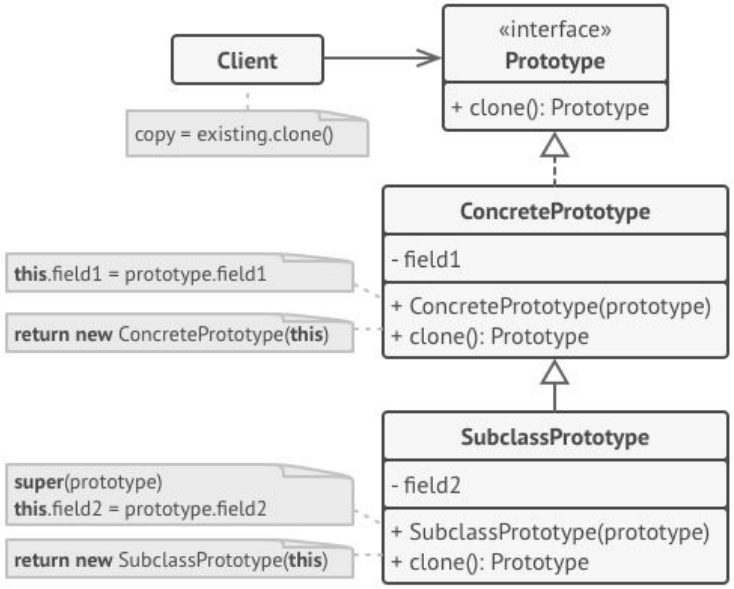
https://refactoring.guru/ru/design-patterns/prototype

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

Базовая реализация

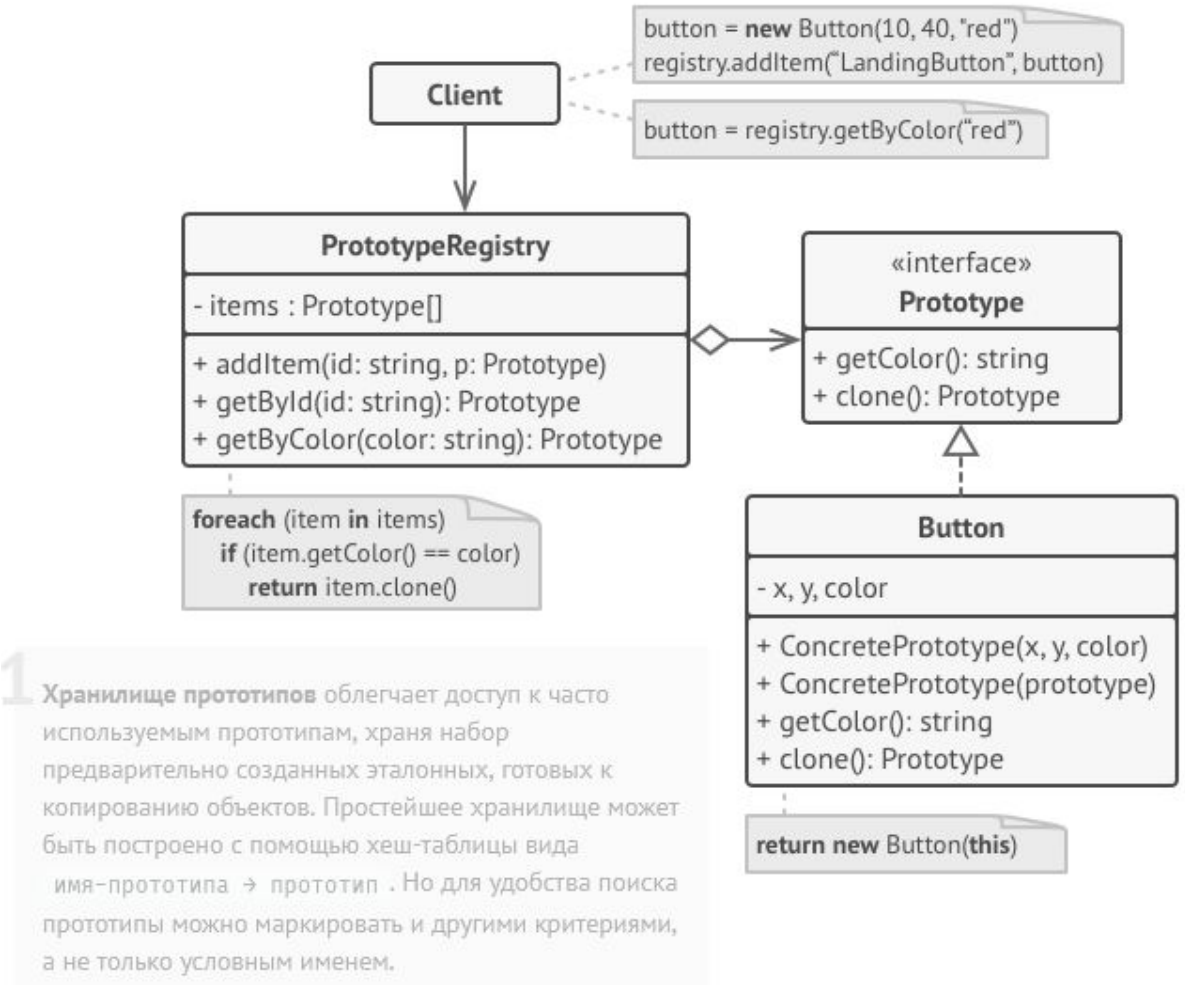
3 **Клиент** создаёт копию объекта, обращаясь к нему через общий интерфейс прототипов.

1 **Интерфейс прототипов** описывает операции клонирования. В большинстве случаев — это единственный метод `clone`.



2 **Конкретный прототип** реализует операцию клонирования самого себя. Помимо банального копирования значений всех полей, здесь могут быть спрятаны различные сложности, о которых не нужно знать клиенту. Например, клонирование связанных объектов, распутывание рекурсивных зависимостей и прочее.

Реализация с общим хранилищем прототипов



Зачем?

? - Когда ваш код не должен зависеть от классов копируемых объектов.

! - Такое часто бывает, если ваш код работает с объектами, поданными извне через какой-то общий интерфейс. Вы не можете привязаться к их классам, даже если бы хотели, поскольку их конкретные классы неизвестны.
Паттерн прототип предоставляет клиенту общий интерфейс для работы со всеми прототипами. Клиенту не нужно зависеть от всех классов копируемых объектов, а только от интерфейса клонирования.

? - Когда вы имеете уйму подклассов, которые отличаются начальными значениями полей. Кто-то мог создать все эти классы, чтобы иметь возможность легко порождать объекты с определённой конфигурацией.

! - Паттерн прототип предлагает использовать набор прототипов, вместо создания подклассов для описания популярных конфигураций объектов.
Таким образом, вместо порождения объектов из подклассов, вы будете копировать существующие объекты-прототипы, в которых уже настроено внутреннее состояние.

Это позволит избежать взрывного роста количества классов в программе и уменьшить её сложность.

72 Умные указатели.

Умный указатель — идиома косвенного обращения к памяти. Как правило, реализуется в виде специализированного класса (обычно — параметризованного), имитирующего интерфейс обычного указателя и добавляющего необходимую новую функциональность.
Основной целью задействования умных указателей является инкапсуляция работы с динамической памятью таким образом, чтобы свойства и поведение умных указателей имитировали свойства и поведение обычных указателей. При этом на них возлагается обязанность своевременного и аккуратного высвобождения выделенных ресурсов, что упрощает разработку кода и процесс отладки, исключая утечки памяти и возникновение висячих ссылок.

```
template <typename T>//Реализация умного указателя
class SmartPointer {
	T *ptr;
	int copy;
public:
	SmartPointer(T *p) {
	ptr=p;
	copy=0;
}

SmartPointer(const T &obj) {
	ptr=obj.ptr;
	copy=1;
}

SmartPointer operator=(const T &obj) {
	ptr=obj.ptr;
	copy=1;
}

~SmartPointer() {
	if (copy==0 && ptr!=NULL) {
		delete ptr;
	}
}

T* operator ->() { return ptr; }
T& operator*() { return *ptr; }
};
```

73. Способы передачи аргумента в функцию.

- передача по значению – значение копируется в параметр функции;

void boo(int y) {}

Аргументы могут быть переменными, выражениями, структурами, классами, перечислениями. Аргументы не изменяются функцией.

- передача по ссылке – передача параметра как ссылки;

void boo(int& y) {}

Аргументы могут быть переменными, выражениями, структурами, классами, перечислениями. Т. к. не происходит копирования аргументов, то передача по ссылке эффективнее и быстрее передачи по значению. Аргументы изменяются функцией.

- передача по адресу – передача адреса переменной-аргумента. Сам параметр – указатель.

void boo(int* y) {}

Аргументы могут быть переменными, выражениями, структурами, классами, перечислениями. Т. к. не происходит копирования аргументов, то передача по ссылке эффективнее и быстрее передачи по значению. Аргументы изменяются функцией (есть побочные эффекты)

74. Статические методы и свойства класса. Константные методы.

Статические методы не привязаны к какому-либо объекту класса (нет *this), поэтому можно их вызывать напрямую через имя класса. Используются для работы со статическими переменными-членами.

```
private:
	static int s_value;
public:
	static int getValue() { return s_value; } // статический метод
};
int Anything::s_value = 3; // определение статической переменной-члена класса
int main() {
std::cout « Anything::getValue() « '\n';
}
```

Свойство класса – это способ доступа к внутреннему состоянию объекта, имитирующий поле (поля). Обращение к свойству объекта выглядит так же, как и обращение к полю, но, в действительности, реализовано через вызов функции. При попытке задать значение свойства вызывается метод называемый сеттером (setter). А при попытке получить значение свойства — геттер (getter).

75. Преобразование типов в C++. dynamic_cast, static_cast, reinterpret_cast, const_cast.

Существует 2 способа преобразования типов:
1. Неявное преобразование типов, когда компилятор автоматически конвертирует один фундаментальный тип данных в другой.
2. Явное преобразование типов, когда разработчик использует один из операторов casts для выполнения конвертации объекта из одного типа данных в другой.
В C++ есть 5 типов операторов casts (операторов явного преобразования типов):

- C-style cast – преобразование с помощью оператора (). Не проверяется компилятором во время компиляции, например: при конвертации типов const или изменении типов данных без учёта их диапазонов (приведёт к переполнению). Пример:
  double res = (double)13 / 7;
- static_cast – преобразование с помощью оператора static_cast. Проверяется компилятором во время компиляции
  double res = static_cast<double>(13) / 7;
- const_cast – константное приведение используются, чтобы константную переменную преобразовать в неконстантную. При этом, константным становится возвращаемое значение операции const_cast, а не сама переменная.;
  void function(char *); // прототип функции с неконстантным параметром
  const char *s = "Sevastopol"; // константная строка
  function(const_cast<char *>(s));
- dynamic_cast – операция dynamic_cast доступна только в C++ и имеет смысл только, применительно к членам класса иерархии «полиморфных типов». Динамическое приведение типов данных может быть использовано для безопасного приведения указателя (или ссылки) на суперкласс, в указатель (или ссылку) на подкласс в иерархии классов. Если динамическое приведение типов — недопустимо, так как реальный тип объекта, указывает не на тот тип подкласса, приведение типов не выполнится;
  type *subClass = dynamic_cast<type *>(objPtr);
  type subClass = dynamic_cast<type &>(objReference);
- reinterpret_cast – операция reinterpret_cast доступна только в C++ и является наименее безопасной формой приведения типов данных в C++, она позволяет интерпретировать значение в другой тип данных. reinterpret_cast не должна быть использована для приведения иерархии классов или преобразования константных переменных.
  reinterpret_cast<dataType>(value);
  reinterpret_cast<int *>(777);

Источник - http://cppstudio.com/post/5343/

76. Директивы препроцессора #define, #ifdef, #ifndef, #if, #elif, #else, #endif, #undef

Процессор-это специальная программа, являющаяся частью компилятора языка Си. Она предназначена для предварительной обработки текста программы. Препроцессор позволяет включать в текст программы файлы и вводить макроопределения.
Работа препроцессора осуществляется с помощью специальных директив (указаний). Они отмечаются знаком решетка #. По окончании строк, обозначающих директивы в языке Си, точку с запятой можно не ставить.

#define — задаёт макроопределение (макрос) или символическую константу
#undef — отменяет предыдущее определение
#if — осуществляет условную компиляцию при истинности константного выражения
#ifdef — осуществляет условную компиляцию при определённости символической константы
#ifndef — осуществляет условную компиляцию при неопределённости символической константы
#else — ветка условной компиляции при ложности выражения
#elif — ветка условной компиляции, образуемая слиянием else и if
#endif — конец ветки условной компиляции