Оглавление

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
    1. Выделяем объекты на этапе анализа (методом 1)
    2. Объекты на этапе реализации
    3. Объекты на этапе проектирования
3. Изучение потенциальных источников объектов
    1. анализ аналогичного ПО
    2. 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.

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.

3. Модульность - принцип разработки программной системы, которая предполагает ее реализацию в отдельных модулях. Модуль - это физический контейнер некоторого набора логических элементов.
4. Иерархическая организация предполагает использование иерархий при разработке программной системы. Иерархии упрощают систему абстракций. Иерархия - это упорядочивание абстракций, расположение их по уровню.
5. 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.
6. Параллелизм - свойство, которое позволяет нескольким абстракциям одновременно находиться в активном состоянии. Данный принцип реализует ОС.
7. Устойчивость (сохраняемость) - свойство абстракции существовать во времени, независимо от процесса, породившего ее.

Парадигма программирования — это совокупность идей и понятий, определяющих стиль написания компьютерных программ (подход к программированию). Это способ концептуализации, определяющий организацию вычислений и структурирование работы, выполняемой компьютером [WIKI].

1. Императивное программирование – это парадигма программирования (стиль написания исходного кода компьютерной программы), для которой характерно следующее:

· в исходном коде программы записываются инструкции (команды);

· инструкции должны выполняться последовательно;

· данные, получаемые при выполнении предыдущих инструкций, могут читаться из памяти последующими инструкциями;

· данные, полученные при выполнении инструкции, могут записываться в память.

Относятся к императивным:

1) Процедурное программирование — программирование на императивном языке, при котором последовательно выполняемые операторы можно собрать в подпрограммы, то есть более крупные целостные единицы кода, с помощью механизмов самого языка;

2) Структурное программирование — парадигма программирования, в основе которой лежит представление программы в виде иерархической структуры блоков;

3) Аспектно-ориентированное программирование (АОП) — парадигма программирования, основанная на идее разделения функциональности для улучшения разбиения программы на модули;

4) Объектно-ориентированное программирование (ООП) — методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования:

· Агентно-ориентированный подход;

· Компонентно-ориентированное;

· Прототипное программирование.

5) Обобщённое программирование — парадигма программирования, заключающаяся в таком описании данных и алгоритмов, которое можно применять к различным типам данных, не меняя само это описание.

2. Декларативное программирование — парадигма программирования, в которой задаётся спецификация решения задачи, то есть описывается, что представляет собой проблема и ожидаемый результат. Противоположностью декларативного является императивное программирование, описывающее на том или ином уровне детализации, как решить задачу и представить результат.

Относятся к декларативным:

1) Функциональное программирование — раздел дискретной математики и парадигма программирования, в которой процесс вычисления трактуется как вычисление значений функций в математическом понимании последних (в отличие от функций как подпрограмм в процедурном программировании);

2) Логическое программирование — парадигма программирования, основанная на автоматическом доказательстве теорем, а также раздел дискретной математики, изучающий принципы логического вывода информации на основе заданных фактов и правил вывода. Логическое программирование основано на теории и аппарате математической логики с использованием математических принципов резолюций.

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

Состоит из: 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)

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). 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 полиморфизм позволяет переопределить методы с одинаковым названием для разных классов в иерархии.Виртуальные методы нужна для увеличения гибкости программы.

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). Есть три типа наследования public, protected и 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 объектов 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

Конструкторы и инициализация объектов.

Существует три способа инициализации объектов

```
class A{
        int a;
public:
                A(int a1):a(a1){} // Инициализация параметра a, до создания объекта
}
```

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


    class A{
        const int
a; public:
```

Сигнатура метода - имя и перечень входных параметров.

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 кооперaций, атрибутов, методов, интерфейсов и взаимосвязей между ними, создается на языке моделирования UML.

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(){
                                s
          td: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";
                    }
```

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
          >show();          // show A
   print A b->print();                //
   print B
          >show();          /* show A print B !!!! метод show не переопределён, поэтому show A, но print является
```

Полиморфизм используется: при передаче объекта в качестве параметра функции или метода, фактическим параметром которых является класс - родитель.

```
                in
       t draw(A
       *a){ a-
```

Когда объекту по указателю на класс - родитель нужно присвоить объект указатель на класс - потомок. 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.

```
                cl
       ass
       Abstract{ int
       k;
      public:
                    Abstract(int k): k(k) {}
       virtual void show() = 0;
      };
      class A: public Abstract{
  public:
                    void show(){
                                std::cout << “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:

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

Для реализации поведения существуют 5 видов операций над объектами:

- модификация - операция для изменения состояния объекта.
- селектор - операция, дающая доступ для определения состояния объекта без его изменения (операция чтения).
- итератор - операция доступа к содержанию объекта по частям (в определенной последовательности).
- Конструктор - операция создания и (или) инициализация объекта.
- 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:

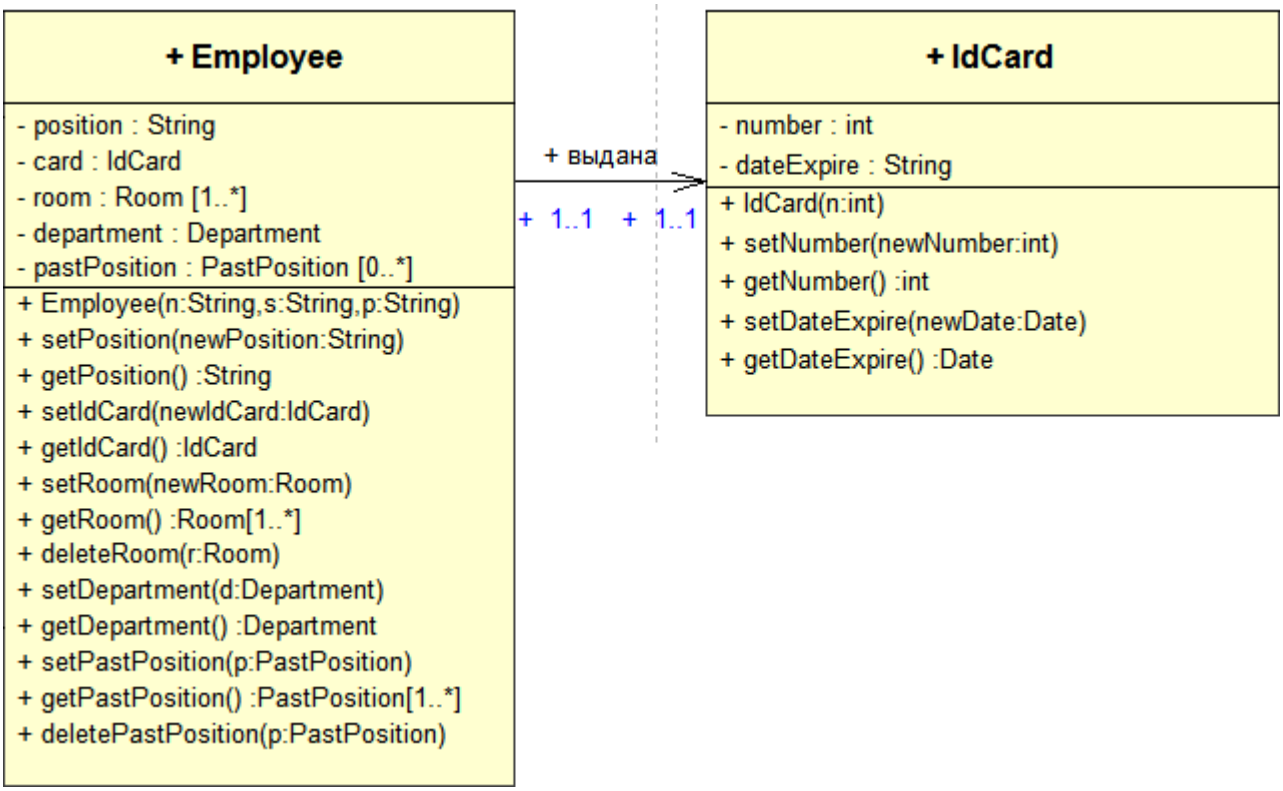
- Актёр - может изменять другие объекты, но не может быть изменён другими объектами
- Агент - может менять своё и чужие состояния.
- Сервер - предоставляет сервисы другим объектам.

Виды отношений:

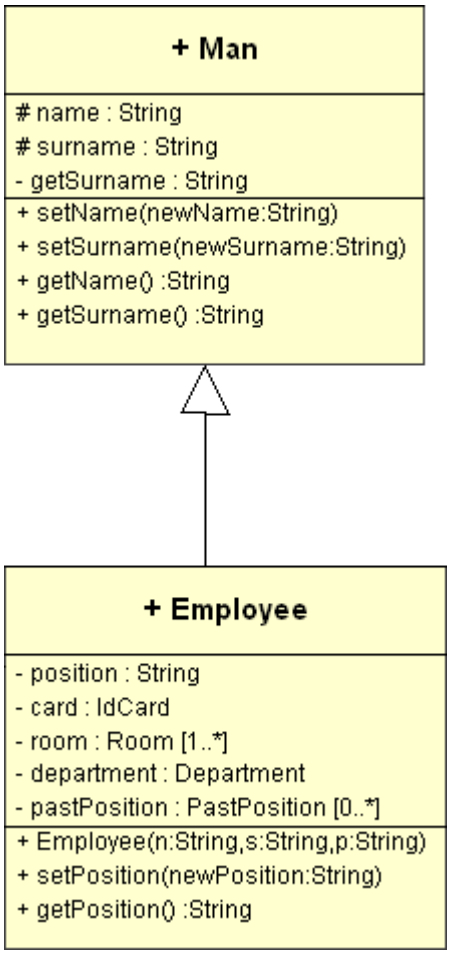
- Связь - характеризуется передачей сообщений друг другу.
- Агрегация - один объект включает в себя другой. Агрегация бывает физической и по ссылке.

24. Виды отношений между классами.

1. Ассоциация - один тип объектов ассоциируется с другим.

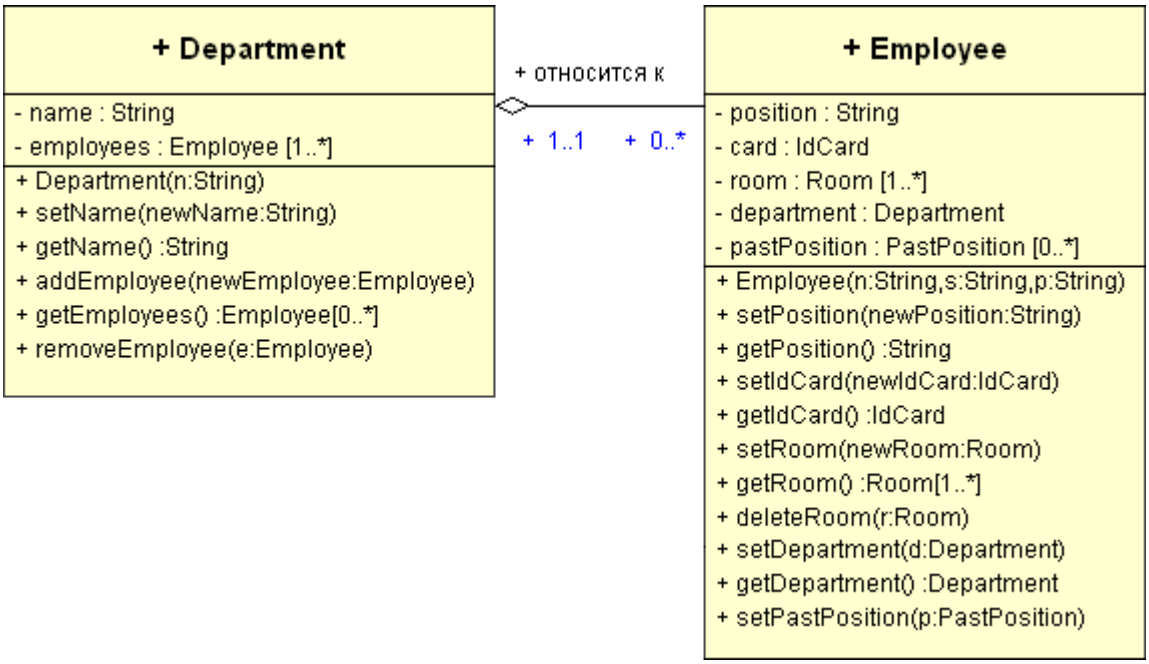


2. Наследование - механизм, позволяющий создавать классы на основе родительских классов.

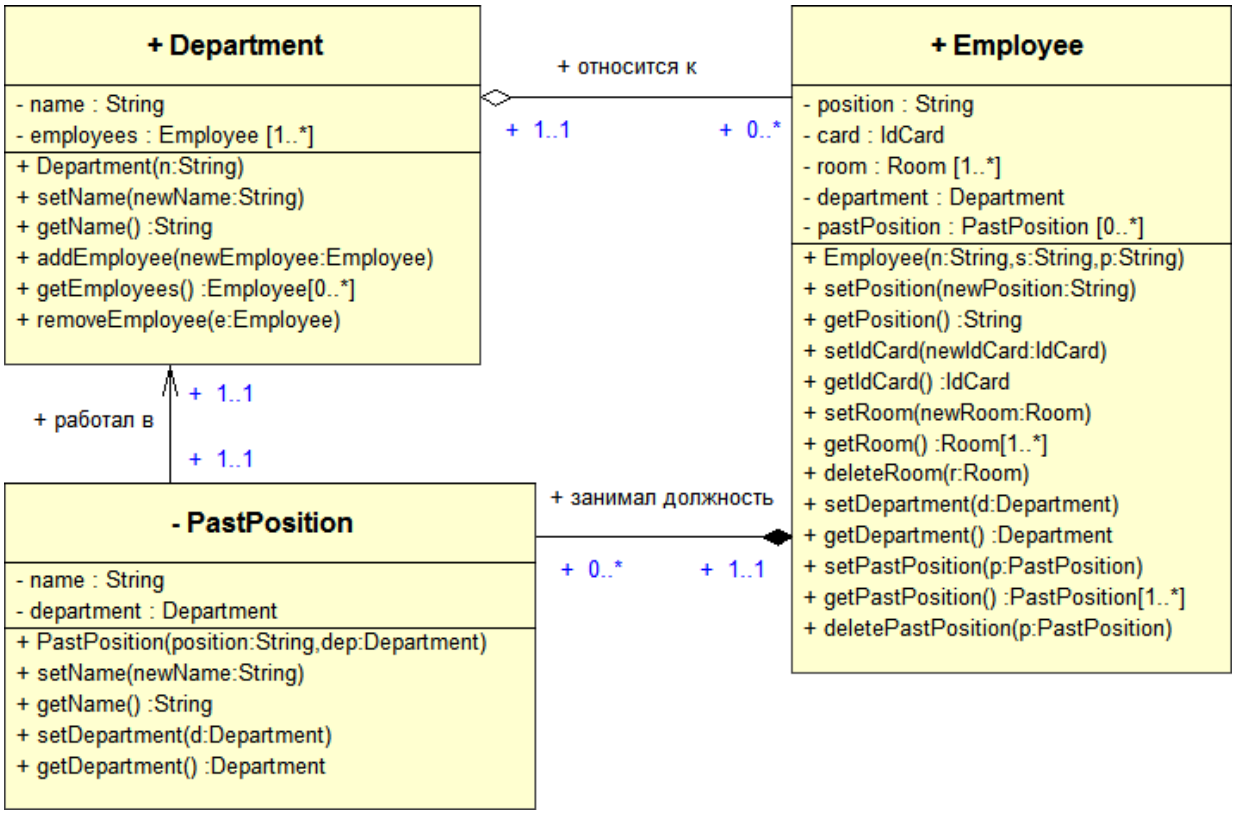


3. Множественное наследование

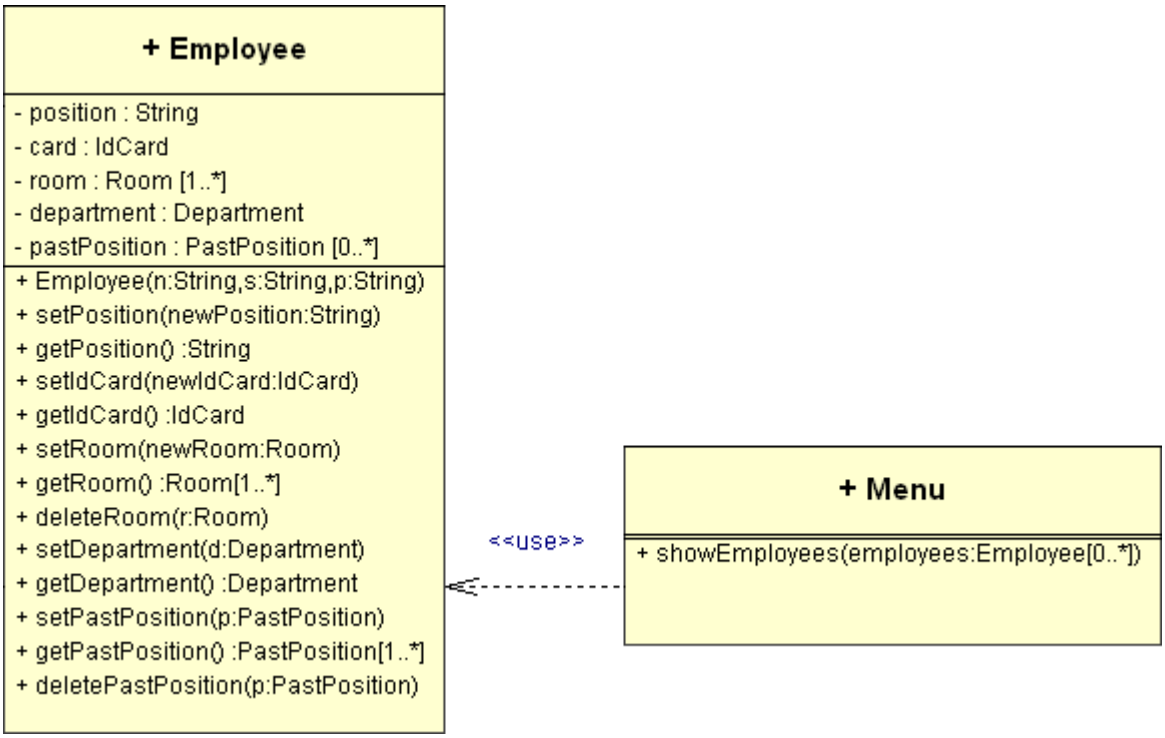
4. Агрегация - включение по ссылке (необязательное, т.е включаемый объект может существовать без исходного)



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();
                }
```

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++.

Исключительные ситуации – механизм языка c++, предназначенный для описания реакции программы на ошибки времени выполнения и другие возможные исключения, которые могут возникнуть при выполнение программы и проводят к невозможности дальнейшей обработки программой базового алгоритма.
Создатель библиотеки способен обнаружить динамические ошибки, но не представляет какой в общем случае должна быть реакция на них. Пользователь библиотеки способен написать реакцию на такие ошибки, но не в силах их обнаружить. Если бы он мог, то сам разобрался бы с ошибками в своей программе, и их не пришлось бы выявлять в библиотечных функциях. Для решения этой проблемы в язык введено понятие особой ситуации.
Существует два вида исключений:

· Аппаратные (структурные, SE-Structured Exception), которые генерируются процессором. К ним относятся, например,

- деление на 0;
- выход за границы массива;
- обращение к невыделенной памяти;
- переполнение разрядной сетки.

· Программные, генерируемые операционной системой и прикладными программами – возникают тогда, когда программа их явно инициирует. Когда встречается аномальная ситуация, та часть программы, которая ее обнаружила, может сгенерировать, или возбудить, исключение.

В языке C++ исключение – это специальный объект класса или значение базового типа, который описывает (определяет) конкретную исключительную ситуацию и соответствующим образом обрабатывается.

Средства работы с исключительными ситуациями в C++:

- *try-блок* — так называемый блок повторных попыток. В нем надо располагать код, который может привести к ошибке и аварийному закрытию программы;
- *throw* генерирует исключение. То что остановит работу try-блока и приведет к выполнению кода catch-блока. Тип исключения должен соответствовать, типу принимаемого аргумента catch-блока;
- *catch-блок* — улавливающий блок, поймает то, что определил throw и выполнит свой код. Этот блок должен располагаться непосредственно под try-блоком. Никакой код не должен их разделять.
- если в try-блоке исключение не генерировалось, catch-блок не сработает. Программа его обойдет.

28. Обработка общих исключительных ситуаций в C++.

лол? чем отличается от 27?

29. Виды отношений между объектами.

Виды отношений: Связь - характеризуется передачей сообщений друг другу. Агрегация - один объект включает в себя другой. Агрегация бывает физической и по ссылке.

30. Библиотека стандартных классов.

Осторожно, копипаст с википедии!!!! Стандартная Библиотека означает коллекцию классов и функций, написанных на базовом языке. Стандартная Библиотека поддерживает несколько основных контейнеров, функций для работы с этими контейнерами, объектов-функции, основных типов строк и потоков (включая интерактивный и файловый ввод-вывод), поддержку некоторых языковых особенностей, и часто используемые функции для выполнения таких задач, как, например, нахождение квадратного корня числа. Стандартная Библиотека языка C++ также включает в себя спецификации стандарта ISO C90 стандартной библиотеки языка Си. Функциональные особенности Стандартной Библиотеки объявляются внутри пространства имен std. Стандартная библиотека шаблонов (STL) — подмножество стандартной библиотеки C++ и содержит контейнеры, алгоритмы, итераторы, объекты-функции и т. д. http://inf-w.ru/?page_id=4872 https://ru.wikipedia.org/wiki/Стандартная*библиотека*языка_C%2B%2B

31. Полиморфизм. Виды полиморфизма в C++.

см. вопрос №21

32. Шаблоны классов, template.

Инстанцирование — отношение между классами, при котором один класс «параметризуется» другими классами, объектами, операциями. Класс, который принимает вышеописанный параметр называется «обобщенным классом» или «параметризованным» классом.

Инстанцирование - подстановка параметров шаблона обобщенного или параметризованного класса. Сам по себе шаблон не может иметь экземпляров. В результате создается конкретный класс, который может иметь экземпляры.

Шаблон – “Класс для классов” (цитата с лекции и на защите прокатило)

Шаблон типа для класса задает способ построения отдельных классов, подобно тому, как описание класса задает способ построения его отдельных объектов.

- Префикс template<class T> указывает, что описывается шаблон типа с параметром T, обозначающим тип, и что это обозначение будет использоваться в последующем описании.

- После того, как идентификатор T указан в префиксе, его можно использовать как любое другое имя типа.

- Область видимости T продолжается до конца описания, начавшегося префиксом template<class T>.

- T объявляется типом, и оно не обязано быть именем класса.

Имя шаблонного класса, за которым следует тип, заключенный в угловые скобки <>, является именем класса (определяемым шаблоном типа), и его можно использовать как все имена класса.

Например:

stack<char> sc(100); // стек символов

stack<m1> stack_m1(100); // стек структур

stack<A> stack_a(10); // стек A

stack<int> stack_int(10); //стек int

```
template<class T>
void swap_values(T &first, T &second) {
    T temp = first;
    first = second;
    second = temp;
}

template<class T>
void print_values(T first, T second) {
    std::cout << first << " " << second << std::endl;
}

int main() {
    int a = 3;
    int b = 4;
    print_values<int>(a, b);
    swap_values<int>(a, b);
    print_values<int>(a, b);
    return 0;
}
```

Компилятор понимает, какие функции нужно создать.
В большинстве случаев и без указания типа

```
int main() {
    int a = 3;
    int b = 4;
    print_values(a, b);
    swap_values(a, b);
    print_values(a, b);
    return 0;
}
```

И так сойдет

Чот может быть аргументом для шаблона

Объект или переменная базового типа:

```
template< class T, int SIZE >
    class Array {
    int Elements_2x = SIZE * 2;
    };
```

Несколько аргументов шаблона:

```
template< class T, class M >
    class Array {
    T t;
    M m;
    };
```

Инициализация по умолчанию:

```
const int _size = 10;
template< class T, int SIZE=_size >
class Array {
    int START = SIZE;
};
```

Необычные решения:

```
template< class Type1, class Type2 =
Type1 >
        class Pair {
            Type1 first;
            Type2 second;
        };

Pair< int > IntPair1;
Pair< int, int > IntPair2;
```

В качестве значения по умолчанию для второго аргумента является переданный первый аргумент.

33. Контейнеры STL. Основные методы.

Библиотека стандартных шаблонов (STL) (англ. Standard Template Library) — набор согласованных обобщённых алгоритмов, контейнеров, средств доступа к их содержимому и различных вспомогательных функций в C++.

Контéйнер в программировании — тип, позволяющий инкапсулировать в себе объекты других типов. Контейнеры реализуют конкретную структуру данных. https://ravesli.com/urok-197-kontejnery-stl/

34. Стандартная библиотека STL (map, list)

map — стандартный шаблонный класс библиотеки языка программирования C++, предназначенный для реализации абстракции отображения в виде упорядоченного ассоциативного контейнера. Его объявление расположено в пространстве имён std заголовочного файла <map> библиотеки STL. В контексте доступа к своим элементам класс map рассматривается как ассоциативный массив, в котором роль индексов играют значения ключей, что позволяет провести аналогию со словарём или телефонной книгой

Элементами класса map являются пары из ключей и соответствующих им значений. Хранение элементов класса map реализовано в упорядоченном виде на основании критерия сортировки, который применяется по значениям ключей. По умолчанию критерий сортировки задаётся оператором operator< . В отличие от контейнера set класс map предоставляет своему пользователю operator []Для контроля за управлением памятью возможно подключать пользовательские версии распределителей памяти. Для практической реализации класса map обычно используются деревья двоичного поиска

Название	Функции
`size()`	Возвращает количество элементов в контейнере
`empty()`	Возвращает `true` если контейнер пуст
`find(k)`	Возвращает итератор, указывающий на значение, соответствующее значению ключа `k` . Если такого значения в контейнере нет, то возвращается итератор `end`
`operator[k]`	Возвращает ссылку на значение, соответствующее ключу `k` . Если такого ключа не существует, то он создаётся.
`insert(pair(k,v))`	Вставляет в контейнер пару `(k,v)` , возвращая адрес его позиции
`erase(k)`	Удаляет из контейнера элемент с ключом `k`
`erase(p)`	Удаляет из контейнера элемент, на который указывает итератор `p`
`begin()`	Возвращает итератор на начало контейнера
`end()`	Возвращает итератор на конец контейнера

35. Стандартная библиотека STL (vector, stack)

Vector – Стандартный шаблон обобщённого программирования языка C++ std::vector<T> — реализация динамического массива.

Методы

	Метод	Описание	Сложность
Конструкторы	`vector::vector`	Конструктор по умолчанию. Не принимает аргументов, создаёт новый экземпляр вектора	`O(1)` (выполняется за константное время)
	`vector::vector(const vector& c)`	Конструктор копии по умолчанию. Создаёт копию вектора `c`	`O(n)` (выполняется за линейное время, пропорциональное размеру вектора `c`)
	`vector::vector(size_type n, const T& val = T())`	Создаёт вектор с `n` объектами. Если `val` объявлена, то каждый из этих объектов будет инициализирован её значением; в противном случае объекты получат значение конструктора по умолчанию типа `T`.	`O(n)`
	`vector::vector(input_iterator start, input_iterator end)`	Создаёт вектор из элементов, лежащих между `start` и `end`	`O(n)`
Деструктор	`vector::~vector`	Уничтожает вектор и его элементы	
Операторы	`vector::operator=`	Копирует значение одного вектора в другой.	`O(n)`
	`vector::operator==`	Сравнение двух векторов	`O(n)`
Доступ к элементам	`vector::at`	Доступ к элементу с проверкой выхода за границу	`O(1)`
	`vector::operator[]`	Доступ к определённому элементу	`O(1)`
	`vector::front`	Доступ к первому элементу	`O(1)`
	`vector::back`	Доступ к последнему элементу	`O(1)`
Итераторы	`vector::begin`	Возвращает итератор на первый элемент вектора	`O(1)`
	`vector::end`	Возвращает итератор на место после последнего элемента вектора	`O(1)`
	`vector::rbegin`	Возвращает `reverse_iterator` на конец текущего вектора.	`O(1)`
	`vector::rend`	Возвращает reverse_iterator на начало вектора.	`O(1)`
Работа с размером вектора	`vector::empty`	Возвращает `true`, если вектор пуст	`O(1)`
	`vector::size`	Возвращает количество элементов в векторе	`O(1)`
	`vector::max_size`	Возвращает максимально возможное количество элементов в векторе	`O(1)`
	`vector::reserve`	Устанавливает минимально возможное количество элементов в векторе	`O(n)`
	`vector::capacity`	Возвращает количество элементов, которое может содержать вектор до того, как ему потребуется выделить больше места.	`O(1)`
	`vector::shrink_to_fit`	Уменьшает количество используемой памяти за счёт освобождения неиспользованной (C++11)	`O(1)`
Модификаторы	`vector::clear`	Удаляет все элементы вектора	`O(n)`
	`vector::insert`	Вставка элементов в вектор	Вставка в конец, при условии, что память не будет перераспределяться — `O(1)`, в произвольное место — `O(n)`
	`vector::erase`	Удаляет указанные элементы вектора (один или несколько)	`O(n)`
	`vector::push_back`	Вставка элемента в конец вектора	`O(1)`
	`vector::pop_back`	Удалить последний элемент вектора	`O(1)`
	`vector::resize`	Изменяет размер вектора на заданную величину	`O(n)`
	`vector::swap`	Обменять содержимое двух векторов	`O(1)`
Другие методы	`vector::assign`	Ассоциирует с вектором поданные значения	`O(n)`, если установлен нужный размер вектора, `O(n*log(n))` при перераспределении памяти
	`vector::get_allocator`	Возвращает аллокатор, используемый для выделения памяти	`O(1)`

36.Стандартная библиотека STL (set, queue)

Set - является ассоциативным контейнером, который содержит отсортированный набор уникальных объектов типа Key. Сортировка выполняется с помощью функции сравнения ключей Compare. Операции поиска, удаления и вставки имеют логарифмическую сложность. Наборы обычно реализуются как красно-черные деревья
Подробное описание всех методов - https://en.cppreference.com/w/cpp/container/set

Класс std :: queue - это контейнерный адаптер, который предоставляет программисту функциональность очереди, в частности, структуру данных FIFO (первым-пришёл-первым-вышел). Подробное описание всех методов - https://en.cppreference.com/w/cpp/container/queue

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

Дополнительные принципы ООП.

1. Параллелизм.

2. Устойчивость.

3. Типизация.

Параллелизм - это свойство нескольких абстракций одновременно находится в активном состоянии, т. е. выполнять некоторые операции «одновременно».

«Виды» параллелизма:

1. Разделение вычислений на потоки.

2. Разделение вычислений на процессы.

3. Разделение вычислений на потоки для графических процессоров.

Пото́к выполне́ния — наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы.

Запуск созданных объектов-потоков можно осуществлять в двух режимах: object_thread.join() - запуск в «синхронном» режиме.

object_thread.detach() - запуск в «асинхронном» режиме.

Синхронно — выполнение операций последовательно.

Асинхронно — «одновременно»

Синхронизация — это организация процесса последовательного выполнения операций.

Существуют различные технологии, которые как правило, связанны с «глобальными/общими» переменными. Рассмотрим механизм синхронизации на основе использования mutex.

std::mutex - объект ядра ОС, по сути позволяет блокировать свое состояние одному потоку и возвращать свой статус другим потокам. Статья про потоки - https://habr.com/ru/post/182610/

38. Объектная декомпозиция.

Объектно-ориентированная технология основывается на так называемой объектной модели. Объектноориентированная технология включает в себя объектно-ориентированное программирование, проектирование и анализ.

При использовании технологии ООП решение задачи представляется в виде результата взаимодействия отдельных функциональных элементов некоторой системы, имитирующей процессы, происходящие в предметной области поставленной задачи.

В такой системе каждый функциональный элемент, получив некоторое входное воздействие (которое называют

сообщением) в процессе решения задачи, выполняет заранее определённые действия (например, может изменить собственное состояние, выполнить некоторые вычисления, нарисовать окно или график и в свою очередь воздействовать на другие элементы). Процессом решения задачи управляет последовательность сообщений. Передавая эти сообщения от элемента к элементу, система выполняет необходимые действия.

Функциональные элементы системы, параметры и поведение которой определяются условием задачи, обладающие самостоятельным поведением (то есть «умеющие» выполнять некоторые действия, зависящие от полученных сообщений и состояния элемента) получили название объектов.

Процесс представления предметной области задачи в виде совокупности объектов, обменивающихся сообщениями, называется объектной декомпозицией.

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

Поток (stream) — это абстракция, которая представляет устройство с операциями ввода и вывода. Таким образом, поток можно понимать как источник и/или приёмник бесконечного количества символов.

Поток представляет собой стек команд со счетчиком, обладающий несколькими важными свойствами, такими как состояние и приоритет. Состояний потока всего три: состояние активности, то есть поток выполняется на данный момент, состояние неактивности, когда поток ожидает выделения процессора для перехода в состояние активности, и третье – состояние блокировки, когда потоку не выделяется время (соответственно он не занимает место в очереди, освобождая ресурсы) вне зависимости от его приоритета.

Класс ios является базовым классом для входных (входной поток используется для ранения данных, полученных от источника данных) и выходных (выходной поток используется для хранения данных, предоставляемых конкретному потребителю данных) потоковых классов в c++.

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.

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

Для использования консольного ввода-вывода в программу необходимо включить заголовочный файл <iostream>, для файлового <fstream>.

Библиотека iostream определяет три стандартные входной - cin и выходной

cout потоки.

Для выполнения операций ввода-вывода переопределены две операции поразрядного сдвига:

· «>>» Извлечь значение из потока;

· «<<» Поместить значение в поток;

При выводе информации значение преобразуется в последовательность символов и выводится в выходной поток:

cout << ‘значение’;

Возможно многократное назначение потоков: cout << 'значение1' << 'значение2' << ... << 'значение n';

При вводе информации из входного потока читается последовательность символов до пробела, затем эта последовательность преобразуется к типу идентификатора, и получаемое значение помещается в идентификатор:

int n;

cin >> n;

Возможно многократное назначение потоков: cin >> переменная1 >> переменная2 >>...>> переменная n;

41. Виды отношений между объектами. Типы объектов.

Отношения между парой объектов основываются на взаимной информации о разрешённых операциях и ожидаемом поведении. Особо интересны два вида отношений между объектами: связи и агрегация. Связь — это физическое или понятийное соединение между объектами. Объект сотрудничает с другими объектами через соединяющие их связи. Связь обозначает соединение, с помощью которого:

- объект-клиент вызывает операции объекта-поставщика;

- один объект перемещает данные к другому объекту.

Можно сказать, что связи являются рельсами между станциями-объектами, по которым ездят «трамвайчики сообщений».

Как участник связи объект может играть одну из трех ролей:

- актёр — объект, который может воздействовать на другие объекты, но никогда не подвержен воздействию других объектов; - сервер — объект, который никогда не воздействует на другие объекты, он только используется другими объектами;

- агент — объект, который может как воздействовать на другие объекты, так и использоваться ими. Агент создаётся для выполнения работы от имени актера или другого агента.

Рассмотрим два объекта, A и B, между которыми имеется связь. Для того чтобы объект A мог послать сообщение в объект B, надо, чтобы B был виден для A. Различают четыре формы видимости между объектами.

1. Объект-поставщик (сервер) глобален для клиента.

2. Объект-поставщик (сервер) является параметром операции клиента.

3. Объект-поставщик (сервер) является частью объекта-клиента. 4. Объект-поставщик (сервер) является локально объявленным объектом в операции клиента.

На этапе анализа вопросы видимости обычно опускают. На этапах проектирования и реализации вопросы видимости по связям обязательно должны рассматриваться. Связи обозначают равноправные (клиент—серверные) отношения между объектами. Агрегация обозначает отношения объектов в иерархии «целое—часть». Агрегация обеспечивает возможность перемещения от целого (агрегата) к его частям (свойствам). Агрегация может обозначать, а может и не обозначать физическое включение части в целое.

При выборе вида отношения между объектами должны учитываться следующие факторы:

связи обеспечивают низкое сцепление между объектами; агрегация инкапсулирует части как секреты целого.

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.

функция не может быть членом двух классов. Надо иметь в языке возможность предоставлять функции, не являющейся членом, право доступа к частным членам класса. Функция - не член класса, - имеющая доступ к его закрытой части, называется другом этого класса. Функция может стать другом класса, если в его описании она описана как friend (друг).

Функция-друг не имеет никаких особенностей, за исключением права доступа к закрытой части класса. В частности, в такой функции нельзя использовать указатель this, если только она действительно не является членом класса. Описание friend является настоящим описанием. Оно вводит имя функции в область видимости класса, в котором она была описана, и при этом происходят обычные проверки на наличие других описаний такого же имени в этой области видимости. Описание friend может находится как в общей, так и в частной частях класса, это не имеет значения.

Отметим, что подобно функции-члену дружественная функция явно описывается в описании класса, с которым дружит. Поэтому она является неотъемлемой частью интерфейса класса наравне с функциейчленом. Функция-член одного класса может быть другом другого класса.

Вполне возможно, что все функции одного класса являются друзьями другого класса. Для этого есть краткая форма записи:

```
class x {

friend class y;

// ...

};
```

В результате такого описания все функции-члены y становятся друзьями класса x.

Вместо того, чтобы делать дружественным целый класс, мы можем сделать дружественными только определённые методы класса. Их объявление

аналогично объявлениям обычных дружественных функций, за исключением имени метода с префиксом className:: в начале (например, Display::displayItem)

45. Дружественные функции.

Функция - не член класса, - имеющая доступ к его закрытой части, называется другом этого класса. Функция

может стать другом класса, если в его описании она описана как friend (друг).

Дружественная функция — это функция, которая не является членом класса, но имеет доступ к членам класса, объявленным в полях private или protected.

46. Перегрузки операторов

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

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

· Арифметические операторы

o Присваивание «=»;

o Сложение «+»;

o Вычитание «-»;

o Унарный плюс «+»;

o Унарный минус «-»;

o Умножение «*»;

o Деление «/»;

o Инкремент префиксный/постфиксный «++»;

o Декремент префиксный/постфиксный «--»;

Оператор присваивания можно реализовать только, как функцию-член, которая должна иметь ровно один параметр. Тип этого параметра произвольный, соответственно, перегрузок может быть несколько, для разных типов параметра. Перегрузка оператора присваивания является составной частью поддержки семантики копирования/перемещения и к ней приходится прибегать достаточно часто.

Бинарные операторы(+, -, *, /) перегружают как свободную функцию(функции определенные в пространстве имён) с двумя аргументами.

В бинарных операторах тип операндов может не совпадать.

Унарный операторы (+, -) не должны изменять операнд и должны возвращать результат по значению.

· Операторы сравнения

o Равенство «==»;

o Неравенство «!=»;

o Больше «>»;

o Меньше «<»;

o Больше или равно «>=»;

o Меньше или равно «<=»;

Операторы сравнения перегружают как свободные функции с двумя аргументами. Перегруженный оператор не должен изменять операнды и должен возвращать bool. При перегрузке

Следует учитывать условия рефлексивности, симметричности и транзитивности

Логические операторы

o Логическое отрицание «!»;

o Логическое умножение «&&»;

o Логическое сложение «||»;

Оператор «!» перегружают для того чтобы проверять, не является ли объект не инициализированным. Он должен возвращать true, если объект не инициализирован («пустой», «нулевой»).

· Побитовые операторы

o Побитовая инверсия «~»;

o Побитовое И «&»;

o Побитовое ИЛИ «|»;

o Побитовое исключающее ИЛИ «^»;

o Побитовые сдвиг влево «<<»;

o Побитовый сдвиг вправо «>>»;

Операторы сдвига нужно перегружать в виде свободных функций с двумя аргументами

· Операторы составного присваивания

o Сложение, совм. с присваиванием «+=»;

o Вычитание совм. с присваиванием «-=»;

o Умножение, совм. с присваиванием «*=»;

o Деление, совм. с присваиванием «/=»;

o Вычисление остатка от деления, совм. с присваиванием «%=»;

o Побитовое И, совм. с присваиванием «&=»;

o Побитовое ИЛИ, совм. с присваиванием «|=»;

o Побитовое исключающее ИЛИ, совм. с присваиванием «^=»;

o Побитовый сдвиг влево, совм. с присваиванием «<<=»;

o Побитовый сдвиг вправо, совм. с присваиванием «>>=»;

Операторам составного присваивания нежелательно скрытое приведение типов, поэтому эти операторы следует перегружать как методы класса.

· Операторы работы с указателями и членами класса

o Обращение к элементу массива «[ ]»;

Бинарный оператор, может быть реализован только как функция – член класса, которая имеет ровно один параметр. Тип этого параметра произвольный, соответственно, перегрузок может быть несколько, для разных типов параметра. Возвращаемое значение является

ссылкой на элемент контейнера

о Оператор «->»

Этот оператор является унарным и может быть реализован только как функция-член (обычно константная). Он должен возвращать либо указатель на класс (структуру, объединение), либо тип, для которого перегружен оператор ->. Перегрузка этого оператора используется для интеллектуальных указателей и итераторов.

о Унарный оператор «*»

Этот унарный оператор часто перегружают в паре с оператором ->. Как правило, он возвращает ссылку на элемент, указатель на который возвращает оператор ->. Этот оператор обычно реализуется как константная функция-член.

В стандартной библиотеке оператор * перегружен для интеллектуальных указателей и итераторов.

· Другие операторы

о Функтор «( )»;

Этот оператор можно реализовать только как функцию-член. Он может иметь любое число параметров любого типа, тип возвращаемого значения также произвольный. Классы, с перегруженным оператором (), называются функциональными, их экземпляры называются функциональными объектами или функторами. Функциональные классы и объекты играют очень важную роль в программировании на C++ и в частности активно используются в стандартной библиотеке. Именно с помощью таких классов и объектов в C++ реализуется парадигма функционального программирования.

48. Исключительные ситуации.

Исключительные ситуации – механизм языка c++, предназначенный для описания реакции программы на ошибки времени выполнения и другие возможные исключения, которые могут возникнуть при выполнение программы и проводят к невозможности дальнейшей обработки программой базового алгоритма.

Создатель библиотеки способен обнаружить динамические ошибки, но не представляет какой в общем случае должна быть реакция на них. Пользователь библиотеки способен написать реакцию на такие ошибки, но не в силах их обнаружить. Если бы он мог, то сам разобрался бы с ошибками в своей программе, и их не пришлось бы выявлять в библиотечных функциях. Для решения этой проблемы в язык введено понятие особой ситуации.

Существует два вида исключений:

· Аппаратные (структурные, SE-Structured Exception), которые генерируются процессором. К ним относятся, например,

· деление на 0;

· выход за границы массива;

· обращение к невыделенной памяти;

· переполнение разрядной сетки.

· Программные, генерируемые операционной системой и прикладными программами – возникают тогда, когда программа их явно инициирует. Когда встречается аномальная ситуация, та часть программы, которая ее обнаружила, может сгенерировать, или возбудить, исключение.

В языке C++ исключение – это специальный объект класса или значение базового типа, который описывает (определяет) конкретную исключительную ситуацию и соответствующим образом обрабатывается.

Средства работы с исключительными ситуациями в C++:

· try-блок — так называемый блок повторных попыток. В нем надо располагать код, который может привести к ошибке и аварийному закрытию программы;

· throw генерирует исключение. То что остановит работу try-блока и приведет к выполнению кода catch-блока. Тип исключения должен соответствовать, типу принимаемого аргумента catch-блока;

· catch-блок — улавливающий блок, поймает то, что определил throw и выполнит свой код. Этот блок должен располагаться непосредственно под try-блоком. Никакой код не должен их разделять.

· если в try-блоке исключение не генерировалось, catch-блок не сработает. Программа его обойдет.

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 - Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

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 Строитель Сингтон Прототип

53. Шаблоны проектирования. Структурные шаблоны.

Структурные паттерны показывают различные способы построения связей между объектами. Эти паттерны отвечают за построение удобных в поддержке иерархий классов.

[WIKI] Структурные шаблоны — шаблоны проектирования, в которых рассматривается вопрос о том, как из классов и объектов образуются более крупные структуры. Использование: структурные шаблоны уровня класса используют наследование для составления композиций из интерфейсов и реализаций. Простой пример — использование множественного наследования для объединения нескольких классов в один. В результате получается класс, обладающий свойствами всех своих родителей. Особенно полезен этот шаблон, когда нужно организовать совместную работу нескольких независимо разработанных библиотек.

Паттерны:

1. Адаптер – позволяет объектам с несовместимыми интерфейсами работать вместе;

2. Мост – разделяет один или несколько классов на две отдельные иерархии — абстракцию и реализацию, позволяя изменять их независимо друг от друга;

3. Компоновщик – позволяет сгруппировать множество объектов в древовидную структуру, а затем работать с ней так, как будто это единичный объект;

4. Декоратор – позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки»;

5. Фасад – предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку;

6. 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;

7. 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. Оригинал: https://refactoring.guru/ru/design-patterns/structural-patterns.

54. Шаблоны проектирования. Шаблоны поведения.

Поведенческие паттерны заботятся об эффективной коммуникации между объектами. Эти паттерны решают задачи эффективного и безопасного взаимодействия между объектами программы.

[WIKI] Поведенческие шаблоны — шаблоны проектирования, определяющие алгоритмы и способы реализации взаимодействия различных объектов и классов. Использование: в поведенческих шаблонах уровня класса используется наследование, чтобы определить поведение для различных классов. В поведенческих шаблонах уровня объекта используется композиция. Некоторые из них описывают, как с помощью кооперации несколько равноправных объектов работают над заданием, которое они не могут выполнить по отдельности. Здесь важно то, как объекты получают информацию о существовании друг друга. Объекты-коллеги могут хранить ссылки друг на друга, но это усиливает степень связанности системы. При

высокой связанности каждому объекту пришлось бы иметь информацию обо всех остальных. Некоторые из шаблонов решают эту проблему.

Паттерны:

1. Цепочка обязанностей – позволяет передавать запросы последовательно по цепочке обработчиков. Каждый последующий обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи;

2. Команда – превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций;

3. Итератор – даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления;

4. Посредник – позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник;

5. Снимок – позволяет сохранять и восстанавливать прошлые состояния объектов, не раскрывая подробностей их реализации;

6. Наблюдатель – создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах;

7. Состояние – позволяет объектам менять поведение в зависимости от своего состояния. Извне создаётся впечатление, что изменился класс объекта;

8. Стратегия – определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы;

9. Шаблонный метод – определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Паттерн позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры;

10. Посетитель – позволяет добавлять в программу новые операции, не изменяя классы объектов, над которыми эти операции могут выполняться. Оригинал: https://refactoring.guru/ru/design-patterns/behavioral-patterns.

55. QT. Слоты и сигналы.

Qt — кроссплатформенный фреймворк для разработки программного обеспечения на языке программирования C++. Есть также «привязки» ко многим другим языкам программирования: Python — PyQt, PySide; Ruby — QtRuby; Java — Qt Jambi; PHP — PHP-Qt и другие.

Qt позволяет запускать написанное с его помощью программное обеспечение в большинстве современных операционных систем путём простой компиляции программы для каждой системы без изменения исходного кода. Включает в себя все основные классы, которые могут потребоваться при разработке прикладного программного обеспечения, начиная от элементов графического интерфейса и заканчивая классами для работы с сетью, базами данных и XML. Является полностью объектно-ориентированным, расширяемым и поддерживающим технику компонентного программирования.

Отличительная особенность — использование метаобъектного компилятора — предварительной системы обработки исходного кода. Расширение возможностей обеспечивается системой плагинов, которые возможно размещать непосредственно в панели визуального редактора. Также существует возможность расширения привычной функциональности виджетов, связанной с размещением их на экране, отображением, перерисовкой при изменении размеров окна.

Комплектуется визуальной средой разработки графического интерфейса Qt Designer, позволяющей создавать диалоги и формы в режиме WYSIWYG.

Про слоты и сигналы: http://doc.crossplatform.ru/qt/4.7.x/signalsandslots.html; https://habr.com/ru/post/50812/.

Обе статьи – переводы, поэтому могут быть ошибки со стороны переводчика.

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 Возвращаемый тип input() - строка. Для преобразования строки в число используется приведение:

Для нескольких элементов можно разбить ввод на подстроки и обработать каждую

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) int – целое число float – число с плавающей точкой complex – комплексное число Списки (Sequence Type) list – список [ ] tuple – кортеж () range – диапазон Строки (Text Sequence Type) str “ ”, ‘ ‘, ‘’’ ‘’’, “”” “”” (Строки могут обозначаться апострофами, кавычками, тройными апострофами, тройными кавычками) Бинарные списки (Binary Sequence Types) - вряд ли кто спросит bytes – байты bytearray – массивы байт memoryview – специальные объекты для доступа к внутренним данным объекта через protocol buffer Множества (Set Types) set – множество Инициализация пустого множества только с

помощью set(), непустое множество можно записать как {1, 2, 3} frozenset – неизменяемое множество Словари (Mapping Types) 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!')
```

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)
```

Можно вызвать метод say так:

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
```

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):
```

```
                                    s
        elf.x = x
        self.y = y
                            d
    ef_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,
```

63. Python. Передача в функцию переменного числа аргументов.

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

```
                d
ef f(x, y):
```

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
```

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)
```

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)
```

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.

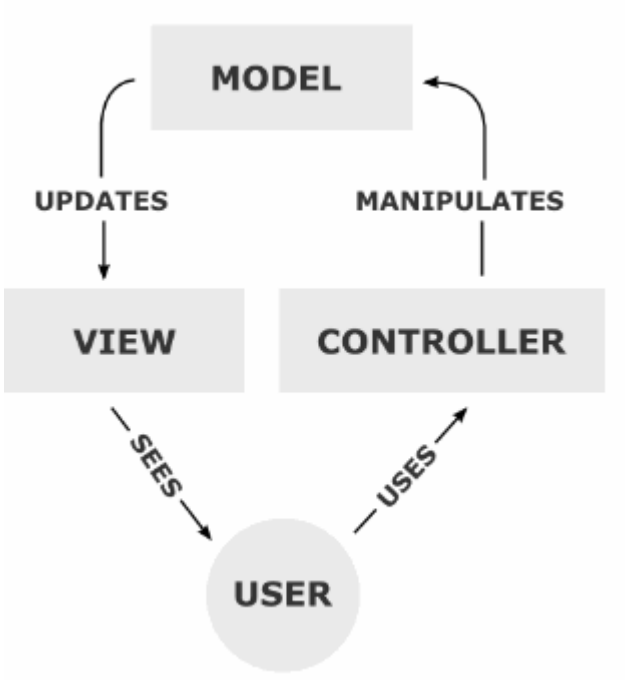
Для обработки исключительных ситуаций Python использует операции try except (else finally) Синтаксис: try: Код, который может выкинуть исключение except имя_исключения: Обработка исключения else: Выполняется, если исключения не произошло finally: Выполняется всегда

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:



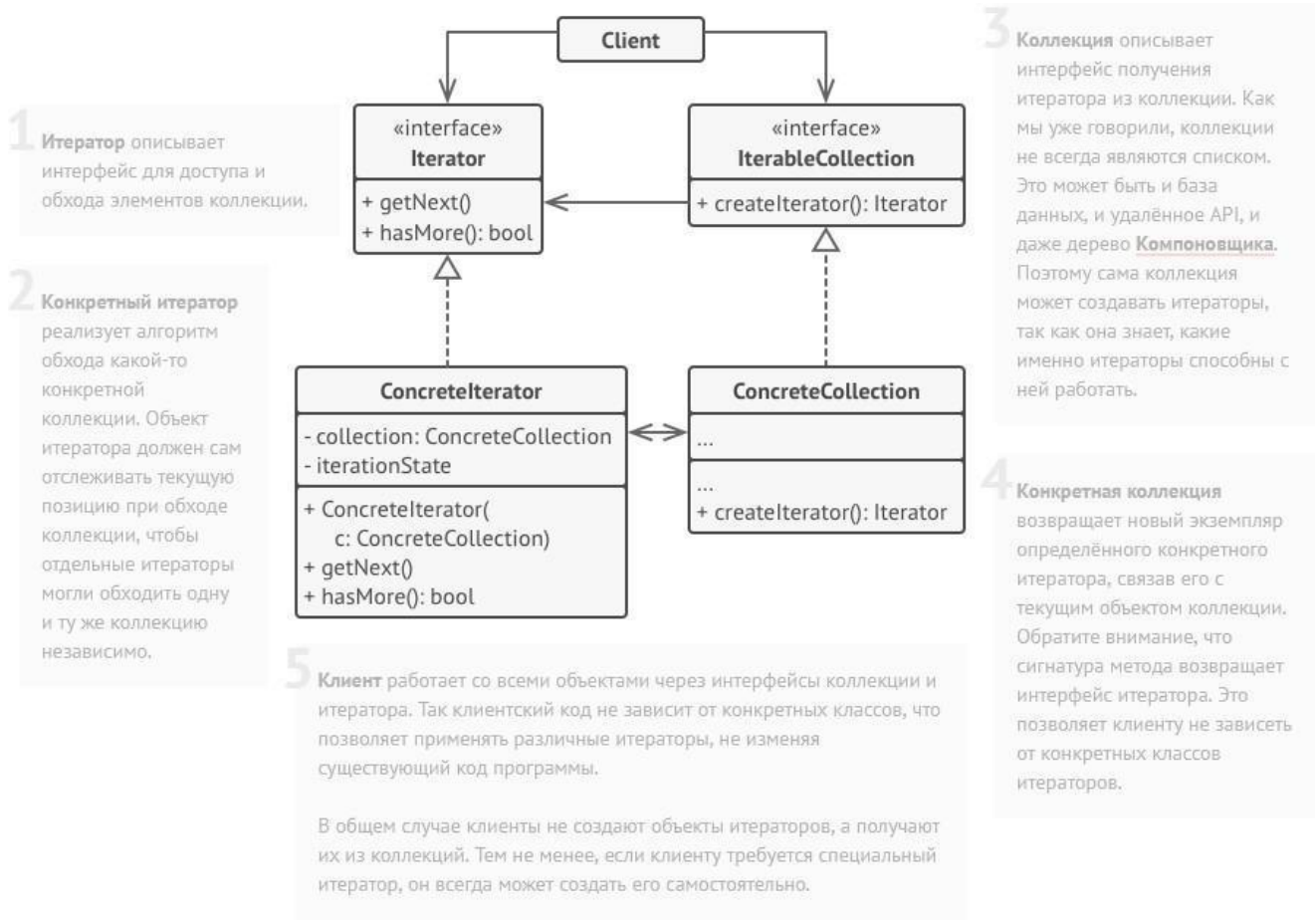
Преимущества такого подхода:

- 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.

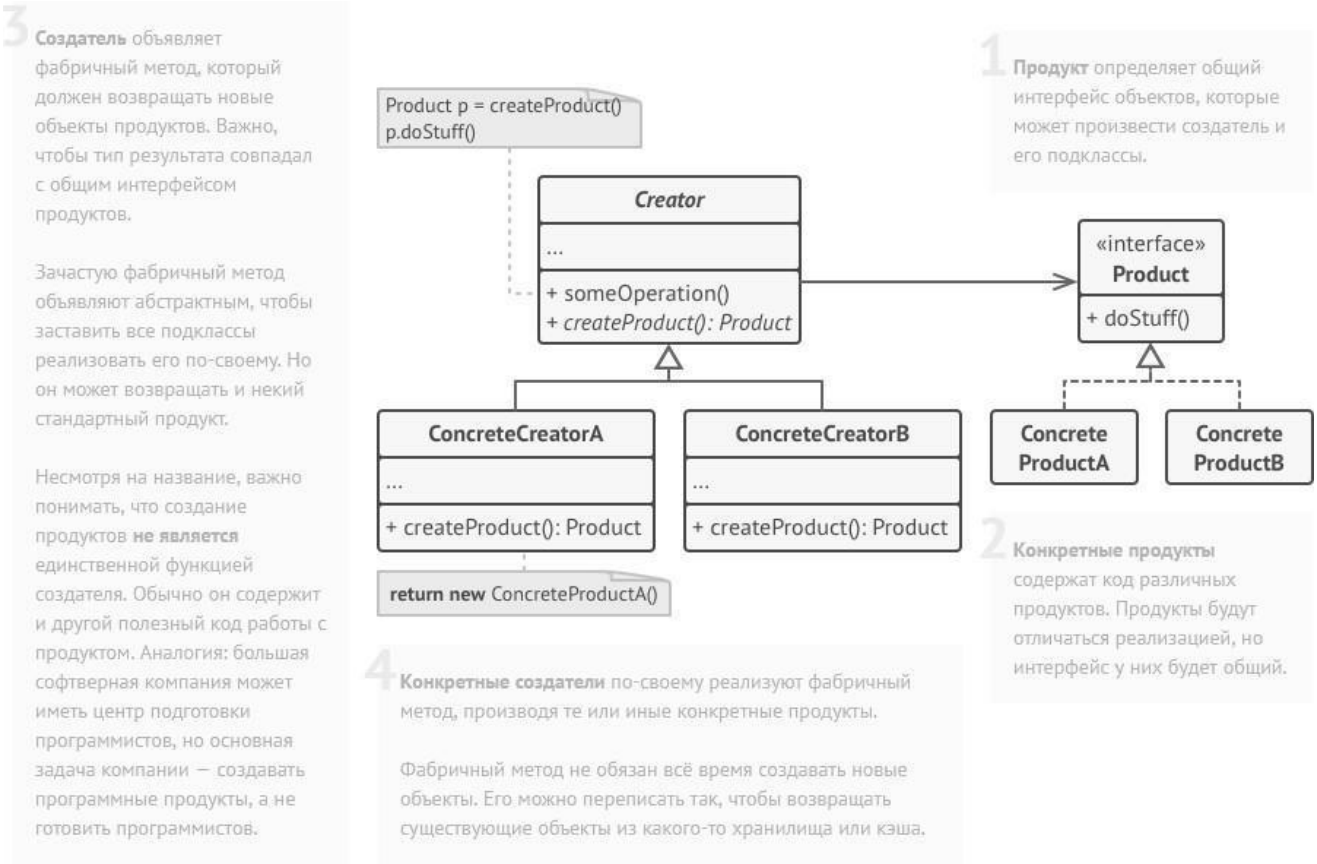
https://refactoring.guru/ru/design-patterns/iterator Итератор — это поведенческий паттерн проектирования, который даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.



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.

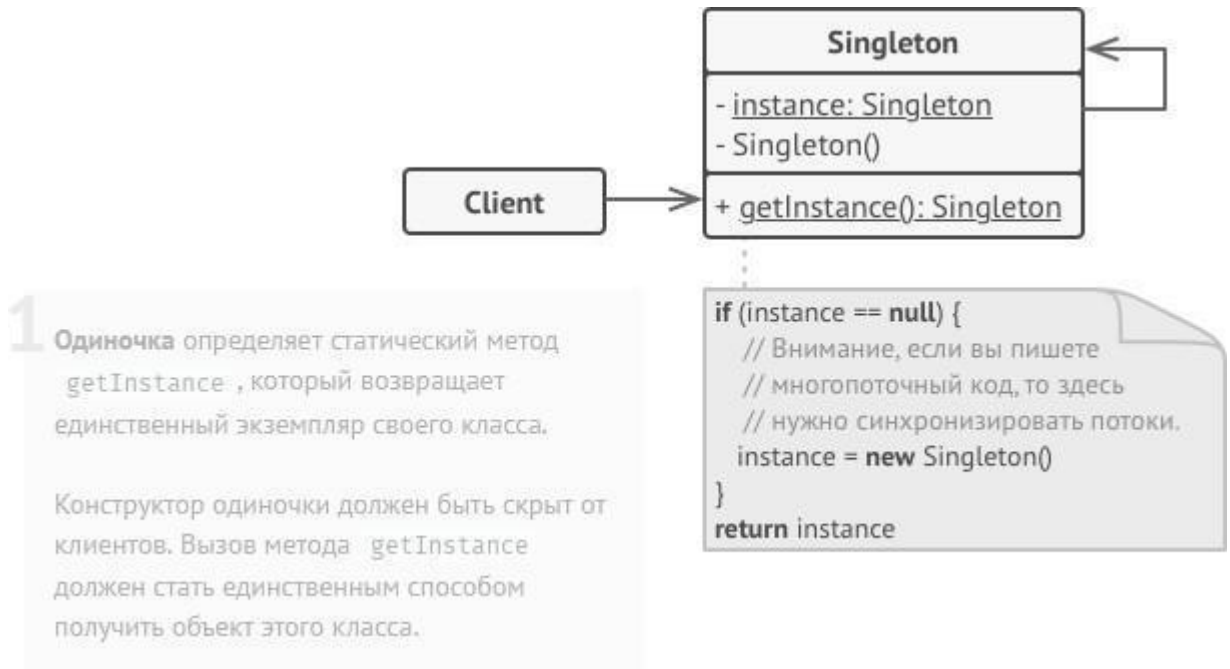


Зачем? ? - Когда заранее неизвестны типы и зависимости объектов, с которыми должен работать ваш код. ! - Фабричный метод отделяет код производства продуктов от остального кода, который эти продукты использует. Благодаря этому, код производства можно расширять, не трогая основной. Так, чтобы добавить 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: Сначала вам следует создать общее хранилище, чтобы хранить в нём все создаваемые объекты. При запросе нового объекта нужно будет заглянуть в хранилище и проверить, есть ли там неиспользуемый объект. А затем вернуть его клиентскому коду. 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).

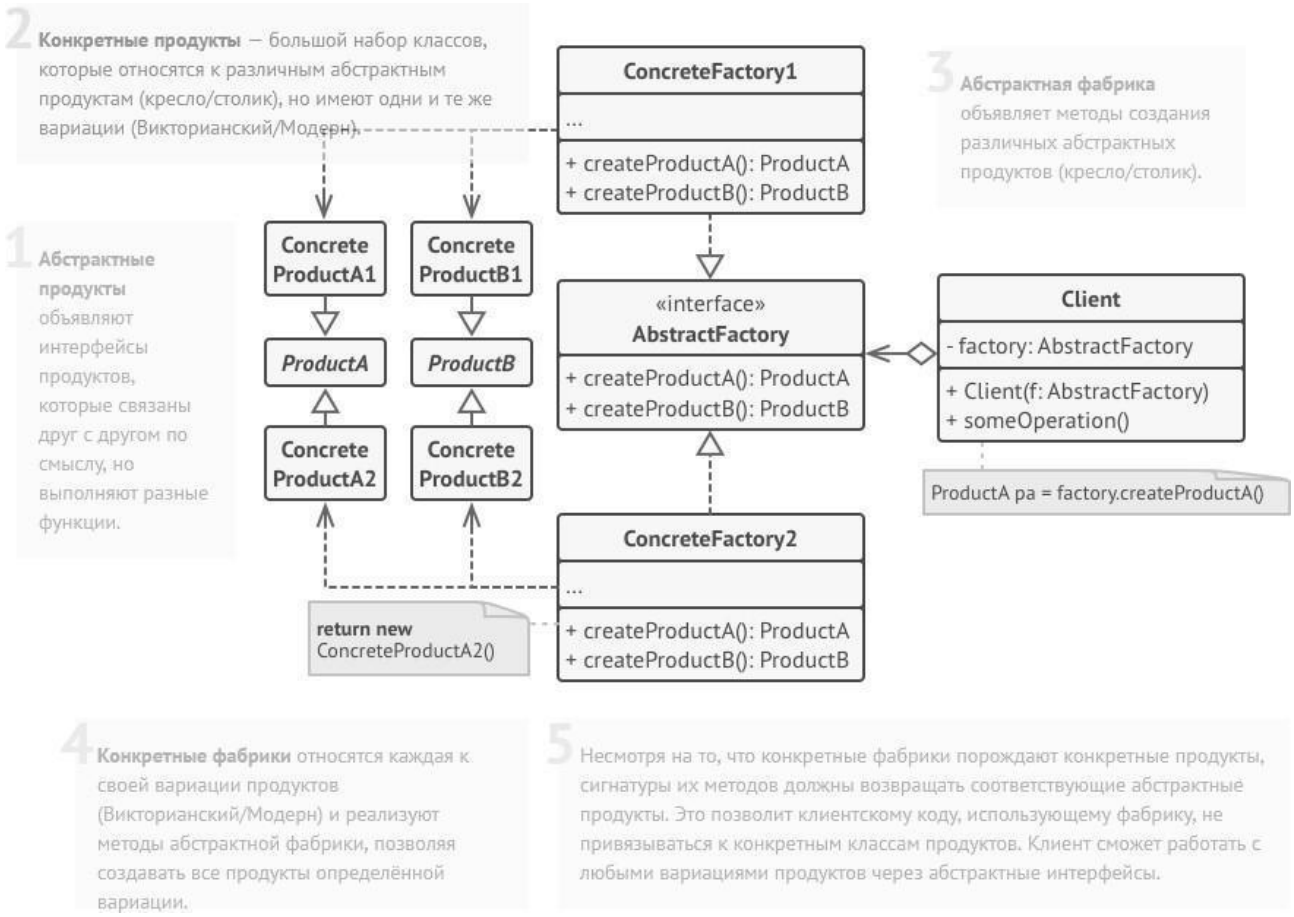
https://refactoring.guru/ru/design-patterns/singleton Одиночка — это порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.



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.

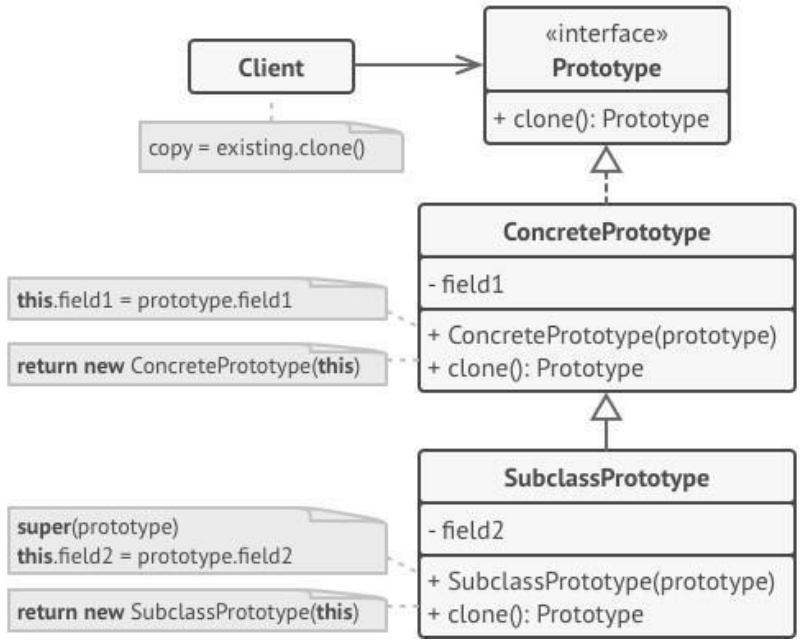
https://refactoring.guru/ru/design-patterns/prototype

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

Базовая реализация

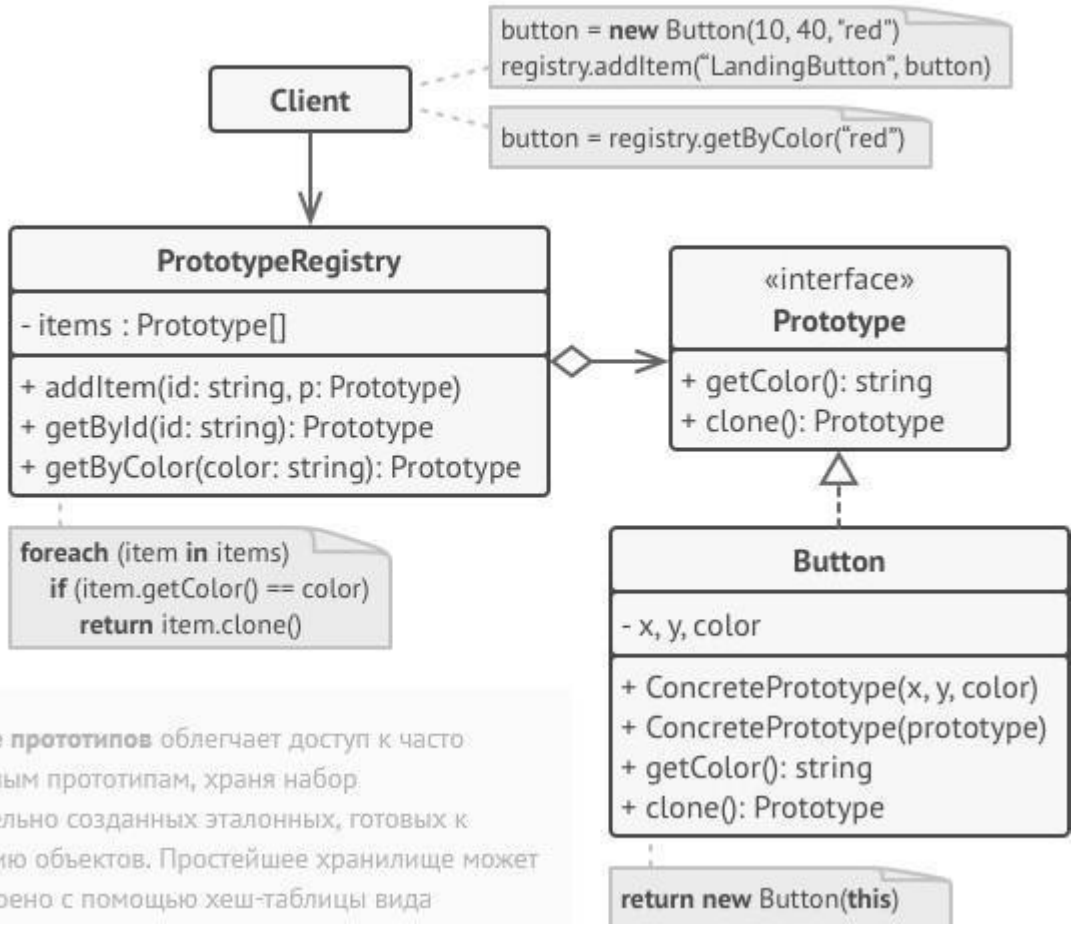
3 **Клиент** создаёт копию объекта, обращаясь к нему через общий интерфейс прототипов.

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



1 **Хранилище прототипов** облегчает доступ к часто используемым прототипам, храня набор предварительно созданных эталонных, готовых к копированию объектов. Простейшее хранилище может быть построено с помощью хеш-таблицы вида

имя-прототипа → прототип . Но для удобства поиска прототипы можно маркировать и другими критериями, а не только условным именем.

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.

73. Способы передачи аргумента в функцию.

74. Статические методы и свойства класса. Константные методы.

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

Препроцессор-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