**PYTHON**

1. **Утиная" типизация**

Концепция утиной типизация в основном принята в языках программирования, поддерживающих динамическую типизацию, таких как Python и JavaScript. Общей особенностью этих языков является возможность объявления переменных без указаниях их типа. Позднее в коде при желании можно назначать другие типы данных этим переменным. Ниже приведены некоторые примеры на Python:

>>> a = 2000

>>> type(a)

<class 'int'>

>>> a = 'Dynamic Typing'

>>> type(a)

<class 'str'>

>>> a = [1, 2, 3]

>>> type(a)

<class 'list'>

В этом фрагменте мы сначала присвоили целое число переменной a, присвоив тем самым ей тип int. Позже мы присвоили тип “строка” и “список” той же переменной, и её тип стал соответственно str, а затем list. Интерпретатор не ругался на изменение типов данных одной и той же переменной.

Как видите, код не может быть скомпилирован, потому что не может присвоить строку или массив переменной a, которая изначально объявлена как целое число.

**Теоретический пример**

В разделе выше мы упоминали, что Python является языком с динамической типизацией, как было показано на самом простом примере, включающем встроенные типы данных. Однако динамическую типизацию можно применять и на пользовательских типах. Давайте рассмотрим теоретический пример ниже:

>>> class Duck:

... def swim\_quack(self):

... print("I'm a duck, and I can swim and quack.")

...

>>> class RoboticBird:

... def swim\_quack(self):

... print("I'm a robotic bird, and I can swim and quack.")

...

>>> class Fish:

... def swim(self):

... print("I'm a fish, and I can swim, but not quack.")

...

>>> def duck\_testing(animal):

... animal.swim\_quack()

...

>>> duck\_testing(Duck())

I'm a duck, and I can swim and quack.

>>> duck\_testing(RoboticBird())

I'm a robotic bird, and I can swim and quack.

>>> duck\_testing(Fish())

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 2, in duck\_testing

AttributeError: 'Fish' object has no attribute 'swim\_quack'

Из успешного вызова функции duck\_testing во фрагменте выше видно, что экземпляр класса Duck точно умеет плавать и крякать. Это же верно и для класса RoboticBird, который так же реализует необходимую функцию swim\_quack. Однако класс Fish не реализует функцию swim\_quack, вследствие чего его экземпляр не проходит утиный тест.

Принимая во внимание эти наблюдения, стоит понимать основные обозначения утиной типизации. При использовании пользовательских типов для определённых целей, реализация связанных функций важнее, чем точные типы данных. В нашем примере, хоть роботизированная птица и не является настоящей уткой, **её реализация функции swim\_quack “превращает” её в утку — животное, которое плавает и крякает.**

1. **Абстрактные типы данных и классы. Сравнение**

Абстрактные типы данных (АТД) в Python используются для логического описания того, как выглядят объекты данных (их состояние) и что они могут делать (их методы). АТД реализуются через новые классы.

Класс в Python — это шаблон кода, который используется для описания структуры и создания объектов, то есть экземпляров этого класса. По сути классы представляют собой типы данных, а объекты — отдельные экземпляры этих типов.

Абстрактный класс предназначен только для наследования, его экземпляры обычно не имеют большого смысла. Классы с рабочими экземплярами называются конкретными. В Python примером абстрактного класса является встроенный тип basestring, у которого есть конкретные подклассы str и unicode.

Простой пример абстрактного класса:

class AbstractClassExample(ABC):

@abstractmethod

def do\_something(self):

pass

Таким образом, **основное различие** между абстрактными типами данных и классами в Python в том, **что АТД реализуются через новые классы**, а **класс определяет тип объекта**, то есть его возможные состояния и набор операций.

1. **Внутренние (вложенные) классы**

Внутренние (вложенные) классы в Python — это классы, определённые внутри

Чтобы создать вложенный класс, просто определите класс внутри другого класса. Вот пример:

class OuterClass:

class InnerClass:

pass других классов.

Здесь InnerClass является вложенным классом, определенным внутри OuterClass.

**Преимущества использования вложенных классов:**

* Повышение читаемости и организации кода. 1 Вложенные классы позволяют легко понять, какие классы связаны.
* Создание более сложных структур данных.

Чтобы создать вложенный класс, нужно определить класс внутри другого класса. Чтобы получить доступ к внутреннему классу из внешнего, необходимо использовать имя внешнего класса вместе с именем внутреннего класса.

Внутренние классы бывают двух типов:

* Множественный внутренний класс. Один внешний класс содержит более одного внутреннего класса. Каждый внутренний класс работает независимо, но он может взаимодействовать с членами внешнего класса. 3
* Многоуровневый внутренний класс. Он ссылается на внутренний класс, который сам содержит другой внутренний класс. Создаёт несколько уровней вложенных классов.

1. **Генерация исключений**

Для принудительной генерации исключения в Python используется инструкция raise. 1 Её синтаксис: raise <класс исключения>(параметры). В качестве параметра можно, например, передать строку с сообщением об ошибке.

Пример генерации встроенного исключения:

def divide(a, b):

if b == 0:

# генерируем встроенное исключение

raise ZeroDivisionError("Деление на ноль невозможно!")

return a / b

# исключение перехватывается в этом блоке

try:

result = divide(10, 0)

except ZeroDivisionError as e:

print(f"Ошибка: {e}")

Также в Python можно создавать собственные исключения. 13 Для этого нужно создать класс, являющийся наследником от одного из классов исключений.

1. **Инкапсуляция и области видимости. Защита на уровне объекта и класса.**

Введение в инкапсуляцию

Инкапсуляция является одним из ключевых принципов объектно-ориентированного программирования (ООП). Она позволяет скрывать внутренние детали реализации класса и предоставлять доступ к данным только через специально определенные методы. Это помогает улучшить модульность кода, повышает его читаемость и облегчает поддержку. Инкапсуляция также способствует защите данных от некорректного использования и изменений, что делает код более надежным и безопасным.

В Python инкапсуляция достигается с помощью специальных соглашений по именованию и использования методов и атрибутов. В этой статье мы рассмотрим основные принципы инкапсуляции, примеры ее использования и практические задачи, которые помогут вам лучше понять эту концепцию. Мы также обсудим, как инкапсуляция может быть полезна при разработке сложных приложений и систем, и как она помогает в управлении зависимостями между различными компонентами кода.

Основные принципы инкапсуляции в Python

Приватные и защищенные атрибуты

В Python нет строгого контроля доступа, как в некоторых других языках программирования, таких как Java или C++. Однако, существует соглашение по именованию, которое помогает разработчикам обозначать приватные и защищенные атрибуты. Это соглашение основано на использовании подчеркиваний в начале имени атрибута.

* Приватные атрибуты: начинаются с двойного подчеркивания \_\_. Они не доступны напрямую из вне класса. Это достигается за счет механизма, называемого "name mangling", который изменяет имя атрибута, добавляя к нему имя класса.
* Защищенные атрибуты: начинаются с одного подчеркивания \_. Они доступны, но считается, что их не следует изменять напрямую. Это больше соглашение, чем строгий запрет, и оно помогает разработчикам понимать, что эти атрибуты предназначены для внутреннего использования.

class MyClass:

def \_\_init\_\_(self):

self.\_protected\_attr = "I am protected"

self.\_\_private\_attr = "I am private"

def get\_private\_attr(self):

return self.\_\_private\_attr

Примеры использования инкапсуляции

Пример 1: Банковский счет

Рассмотрим пример класса BankAccount, который инкапсулирует баланс счета и предоставляет методы для его изменения. Это позволяет контролировать доступ к балансу и предотвращать некорректные операции.

class BankAccount:

def \_\_init\_\_(self, initial\_balance):

self.\_\_balance = initial\_balance

def deposit(self, amount):

if amount > 0:

self.\_\_balance += amount

def withdraw(self, amount):

if 0 < amount <= self.\_\_balance:

self.\_\_balance -= amount

@property

def balance(self):

return self.\_\_balance

account = BankAccount(100)

account.deposit(50)

print(account.balance) # 150

account.withdraw(30)

print(account.balance) # 120

Практические примеры и задачи

Задача 1: Создание класса Rectangle

Создайте класс Rectangle, который инкапсулирует ширину и высоту прямоугольника и предоставляет методы для расчета площади и периметра. Это позволит вам контролировать доступ к атрибутам и обеспечивать корректные значения.

class Rectangle:

def \_\_init\_\_(self, width, height):

self.\_\_width = width

self.\_\_height = height

@property

def width(self):

return self.\_\_width

@width.setter

def width(self, value):

if value > 0:

self.\_\_width = value

@property

def height(self):

return self.\_\_height

@height.setter

def height(self, value):

if value > 0:

self.\_\_height = value

def area(self):

return self.\_\_width \* self.\_\_height

def perimeter(self):

return 2 \* (self.\_\_width + self.\_\_height)

rect = Rectangle(3, 4)

print(rect.area()) # 12

print(rect.perimeter()) # 14

Создайте класс Student, который инкапсулирует имя и оценки студента и предоставляет методы для добавления оценок и расчета среднего балла. Это поможет вам управлять данными студента и обеспечивать корректные значения.

class Student:

def \_\_init\_\_(self, name):

self.name = name

self.\_\_grades = []

def add\_grade(self, grade):

if 0 <= grade <= 100:

self.\_\_grades.append(grade)

def average\_grade(self):

return sum(self.\_\_grades) / len(self.\_\_grades) if self.\_\_grades else 0

student = Student("Alice")

student.add\_grade(90)

student.add\_grade(85)

print(student.average\_grade()) # 87.5

1. **История развития объектно-ориентированных языков, их классификация и архитектура. Основные понятия объектно-ориентированного анализа. Объектная декомпозиция.**

**История развития объектно-ориентированных языков включает несколько ключевых этапов:**

В конце 1950-х и начале 1960-х годов появились первые попытки разработки программ на основе идей объектно-ориентированного программирования (ООП).

Первым объектно-ориентированным языком программирования стал Simula 67, разработанный в Норвегии в 1967 году. 14 Этот язык был создан для моделирования сложных систем и включал такие понятия, как классы и объекты.

В 1970-х годах в Xerox PARC был разработан язык Smalltalk, который стал первым полностью объектно-ориентированным языком программирования. Smalltalk ввёл многие концепции, которые стали основой для современных ООП-языков, такие как наследование, полиморфизм и инкапсуляция.

В 1980-е и 1990-е годы объектно-ориентированное программирование стало широко распространённым благодаря языкам программирования, таким как C++, Java и Python. Эти языки установили стандарты в области ООП и используются до сих пор

**Основные понятия объектно-ориентированного анализа**

Основными понятиями объектно-ориентированного подхода являются объект и класс. Объект — предмет или явление, имеющее четко определенное поведение и обладающие состоянием, поведением и индивидуальностью. Структура и поведение схожих объектов определяют общий для них класс. Класс – это множество объектов, связанных общностью структуры и поведения. Следующую группу важных понятий объектного подхода составляют наследование и полиморфизм.

**Объектная декомпозиция**

Объектная декомпозиция в Python означает разделение сложной задачи на более мелкие и управляемые компоненты.

Основные принципы объектной декомпозиции:

1. Определение объектов: Начните с определения объектов, которые имеют значение в вашей системе. Каждый объект будет представлять собой экземпляр класса.

2. Определение классов: Каждому объекту соответствует класс с его атрибутами и методами. Класс описывает поведение и характеристики объекта.

3. Инкапсуляция: Скрывайте внутреннее представление объектов от пользователя, предоставляя только необходимые интерфейсы для взаимодействия.

4. Наследование: Используйте наследование для создания новых классов, которые расширяют или изменяют поведение существующих классов.

5. Полиморфизм: Позволяет использовать один интерфейс для работы с различными классами, что помогает поддерживать код более гибким.

### Пример объектной декомпозиции

Рассмотрим пример, в котором мы создаем систему учета электронной библиотеки. Мы можем определить несколько классов: Book, Author, Library.

class Author:

def \_\_init\_\_(self, name, birth\_year):

self.name = name

self.birth\_year = birth\_year

def get\_info(self):

return f"{self.name}, born in {self.birth\_year}"

class Book:

def \_\_init\_\_(self, title, author, publication\_year):

self.title = title

self.author = author # Ожидается, что автор - это экземпляр класса Author

self.publication\_year = publication\_year

def get\_info(self):

author\_info = self.author.get\_info()

return f"{self.title}, written by {author\_info}, published in {self.publication\_year}"

class Library:

def \_\_init\_\_(self):

self.books = []

def add\_book(self, book):

self.books.append(book)

def get\_books\_info(self):

return [book.get\_info() for book in self.books]

# Пример использования

if \_\_name\_\_ == "\_\_main\_\_":

# Создание авторов

author1 = Author("J.K. Rowling", 1965)

author2 = Author("George Orwell", 1903)

# Создание книг

book1 = Book("Harry Potter and the Philosopher's Stone", author1, 1997)

book2 = Book("1984", author2, 1949)

# Создание библиотеки и добавление книг

library = Library()

library.add\_book(book1)

library.add\_book(book2)

# Получение информации о всех книгах в библиотеке

for book\_info in library.get\_books\_info():

print(book\_info)

### Объяснение примера:

1. Класс Author: Представляет автора книги и хранит его имя и год рождения. Метод get\_info возвращает информацию об авторе.

2. Класс Book: Представляет книгу, содержит ее название, экземпляр класса Author и год публикации. Метод get\_info возвращает информацию о книге.

3. Класс Library: Управляет коллекцией книг. Метод add\_book добавляет книгу в библиотеку, а get\_books\_info возвращает информацию о всех книгах.

4. Использование классов: В основном блоке создаются авторы и книги, затем они добавляются в библиотеку, и выводится информация обо всех книгах.

1. **Источники данных и компоненты доступа к базам данных**.

Некоторые источники данных и компоненты доступа к базам данных в Python:

SQLite. Встроенная база данных в Python, подходит для небольших проектов и тестирования. Для работы с ней достаточно импортировать модуль sqlite3. 5

ODBC. Интерфейс доступа к базам данных, разработанный в компании Microsoft. Самый используемый метод связи через ODBC — пакет pyodbc.

PostgreSQL. 12 Для работы с этой базой данных в Python используется модуль psycopg2. 2

MySQL. 15 Для подключения к базе данных можно использовать пакет mysqlclient. 1

Oracle. Для работы с этой базой данных применяется пакет cx\_Oracle. 5

Также существуют библиотеки для работы с SQL и объектно-реляционными отображениями, такие как SQLAlchemy и Django ORM, которые автоматизируют задачи взаимодействия Python с базами данных

1. **Классы для работы с датой и временем**

**Некоторые классы для работы с датой и временем в Python:**

* date (дата). Представляет собой календарную дату в формате «год-месяц-день». Используется, когда нужно работать только с датой без времени. **(date(year, month, day)**
* time (время). Класс представляет собой момент времени в течение суток. Содержит атрибуты для часов, минут, секунд и микросекунд. **time([hour] [, min] [, sec] [, microsec])**
* datetime (дата и время). Этот класс объединяет функциональность date и time и представляет собой точку во времени. Он используется для работы с полной датой и временем. **datetime(year, month, day [, hour] [, min] [, sec] [, microsec])**
* timedelta (временной интервал). Класс представляет разницу между двумя моментами времени. Используется, например, для вычисления новой даты на основе текущей и определённого временного интервала

Для получения текущих даты и времени можно вызвать метод now():

1. Классы для работы с файлами и каталогами

Классы для работы с файлами и каталогами в Python предоставляются модулем pathlib. Он включает следующие классы: 2

* Path — базовый класс для работы с путями; 2
* PosixPath — подкласс Path, специфичный для POSIX-систем (Linux, macOS);
* WindowsPath — подкласс Path, специфичный для Windows.

Обычно не нужно заботиться о специфичных классах, так как модуль автоматически определит, какой класс использовать в зависимости от операционной системы.

Некоторые методы классов для работы с путями:

* Path.cwd() — возвращает объект пути, представляющий текущий рабочий каталог;
* Path.home() — возвращает объект пути, представляющий домашний каталог пользователя;
* Path(path).expanduser() — возвращает объект абсолютного пути path, в котором пользовательские конструкции ~ или ~user в начале пути заменяются на домашний каталог пользователя;
* Path(path).resolve(strict=False) — преобразует относительный путь path в абсолютный.

С помощью объектов Path можно выполнять и операции с файлами, например, открывать и читать файл, записывать в него данные, переименовывать файлы.

1. **Конструкторы и деструкторы**

**Конструктор в Python —** это метод, который автоматически вызывается при создании объектов. 13 Роль конструктора играет метод init.

class Employee:

def \_\_init\_\_(self, name, id):

self.id = id

self.name = name

def display(self):

print("ID: %d nName: %s" % (self.id, self.name))

emp1 = Employee("John", 101)

emp2 = Employee("David", 102)

# accessing display() method to print employee 1 information

emp1.display()

# accessing display() method to print employee 2 information

emp2.display()

Источник: https://pythonpip.ru/osnovy/konstruktory-python

**Деструктор в Python** — это метод, который вызывается при уничтожении объекта. Деструктор используется для выполнения действий по очистке перед разрушением объекта, например, закрытия соединений с базой данных или дескриптора файла. 2 В классах Python функцию деструктора выполняет метод del.

Деструктор вызывается, когда объект выходит за пределы области видимости или когда счётчик ссылок на объект достигает нуля. 2 Также все объекты уничтожаются, когда программа завершает свою работу.

Вот простой пример, который иллюстрирует, как работает деструктор:

class MyClass:

def \_\_init\_\_(self, name):

self.name = name

print(f"{self.name} создан")

def \_\_del\_\_(self):

print(f"{self.name} уничтожен")

# Создаем экземпляр класса

obj = MyClass("Объект 1")

# Удаляем ссылку на объект

del obj

# Если не удалить явным образом, объект будет уничтожен при выходе из области видимости

Таким образом, конструктор используется для инициализации объектов, а деструктор — для удаления или уничтожения объектов, в результате чего освобождается ресурс, занятый этими объектами.

**Главные моменты о деструкторах в Python**

1. Сборка мусора: Деструкторы в Python работают в рамках механизма сборки мусора. Python использует подсчет ссылок для управления памятью, и объект считается «мертвым», когда на него больше нет ссылок.

2. Неявный вызов: Деструктор вызывается автоматически без необходимости явного вызова. Однако вы не можете полагаться на то, что \_\_del\_\_ будет вызван в точное время, так как это зависит от процесса сборки мусора.

3. Проблемы с циклическими ссылками: Если у вас есть циклические ссылки (то есть объект ссылается на другой, и наоборот), это может помешать сборке мусора, и деструкторы могут не сработать.

4. Не рекомендуется: Пользоваться деструкторами не всегда считается хорошей практикой в Python, так как лучше явным образом управлять ресурсами. Например, для управления файлами и сетевыми соединениями предпочтительнее использовать контекстные менеджеры (with).

1. **Множественное наследование. Порядок разрешения методов (MRO)**

Множественное наследование в Python — это возможность класса иметь более одного родительского класса. При множественном наследовании дочерний класс наследует все свойства родительских классов.

Пример множественного наследования

Рассмотрим пример, в котором два родительских класса A и B предоставляют метод hello, а дочерний класс C наследует от них:

class A:

def hello(self):

return "Hello from A"

class B:

def hello(self):

return "Hello from B"

class C(A, B):

pass

c\_instance = C()

print(c\_instance.hello()) # Выведет "Hello from A"

Порядок разрешения методов (MRO) определяет, где Python будет искать нужный метод и какой метод будет вызываться при возникновении конфликта. 4 По порядку разрешения методов любой указанный атрибут сначала ищется в объявленном классе. Если его там нет, поиск продолжается в родительских классах на максимальную глубину слева направо без прохода по одному классу дважды.

Вы можете узнать MRO класса, вызвав метод \_\_mro\_\_ или метод mro():

print(C.\_\_mro\_\_) # Выводит кортеж, в котором перечислены классы в порядке разрешения методов

print(C.mro()) # То же самое

Вывод будет примерно таким:

(<class '\_\_main\_\_.C'>, <class '\_\_main\_\_.A'>, <class '\_\_main\_\_.B'>, <class 'object'>)

Сначала будет проверяться класс C, затем A, затем B, и, наконец, базовый класс object.

Например, в случае с классом MultiDerived порядок поиска будет следующим: [MultiDerived, Base1, Base2, object]. 1

MRO класса можно просмотреть в атрибуте mro или с помощью метода mro(). Вызов атрибута возвращает кортеж, а вызов метода — список

1. **Обработка событий клавиатуры и мыши**

Для обработки событий клавиатуры и мыши в Python можно использовать различные библиотеки и подходы.

* Pygame. Популярная библиотека для создания видеоигр и мультимедийных приложений на Python. Пример обработки событий клавиатуры и мыши с её помощью
* Tkinter. 45 Стандартная библиотека Python для создания графических пользовательских интерфейсов (GUI).

1. **Объектная модель программы. Определение объекта и класса, атрибута и свойства. Отношения, основные типы отношений.**

Объектная модель программы в Python основана на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.

**Объект (экземпляр класса)** — это конкретный представитель класса. 2 Для создания объекта достаточно вызвать класс по имени и задать параметры конструктора.

Класс описывает множество объектов, имеющих общую структуру и обладающих одинаковым поведением. Это шаблон кода, по которому создаются объекты.

**Атрибут класса (объекта)** — любой элемент (свойство, метод, подкласс), на который можно сослаться через символ точки. Атрибуты делятся на встроенные и пользовательские:

**Встроенные (служебные) атрибуты** — методы и свойства, унаследованные от общего для всех классов в Python родительского класса object. Многие из этих атрибутов можно переопределить внутри своего класса.

**Пользовательские атрибуты** — поля и методы, которые описываются программистом в теле класса. Добавляются в общий список атрибутов наряду со встроенными.

**Свойства** — это особый вид атрибутов, имитирующий поле, но который при чтении вызывает какой-либо метод. У них есть методы получения, установки и удаления, такие как get, set и delete.

**Основные типы отношений в объектно-ориентированном программировании (ООП) в Python:**

«IS-A» («есть» — между экземпляром и классом);

«AKO» («a kind of» — «разновидность» — между классом и суперклассом)

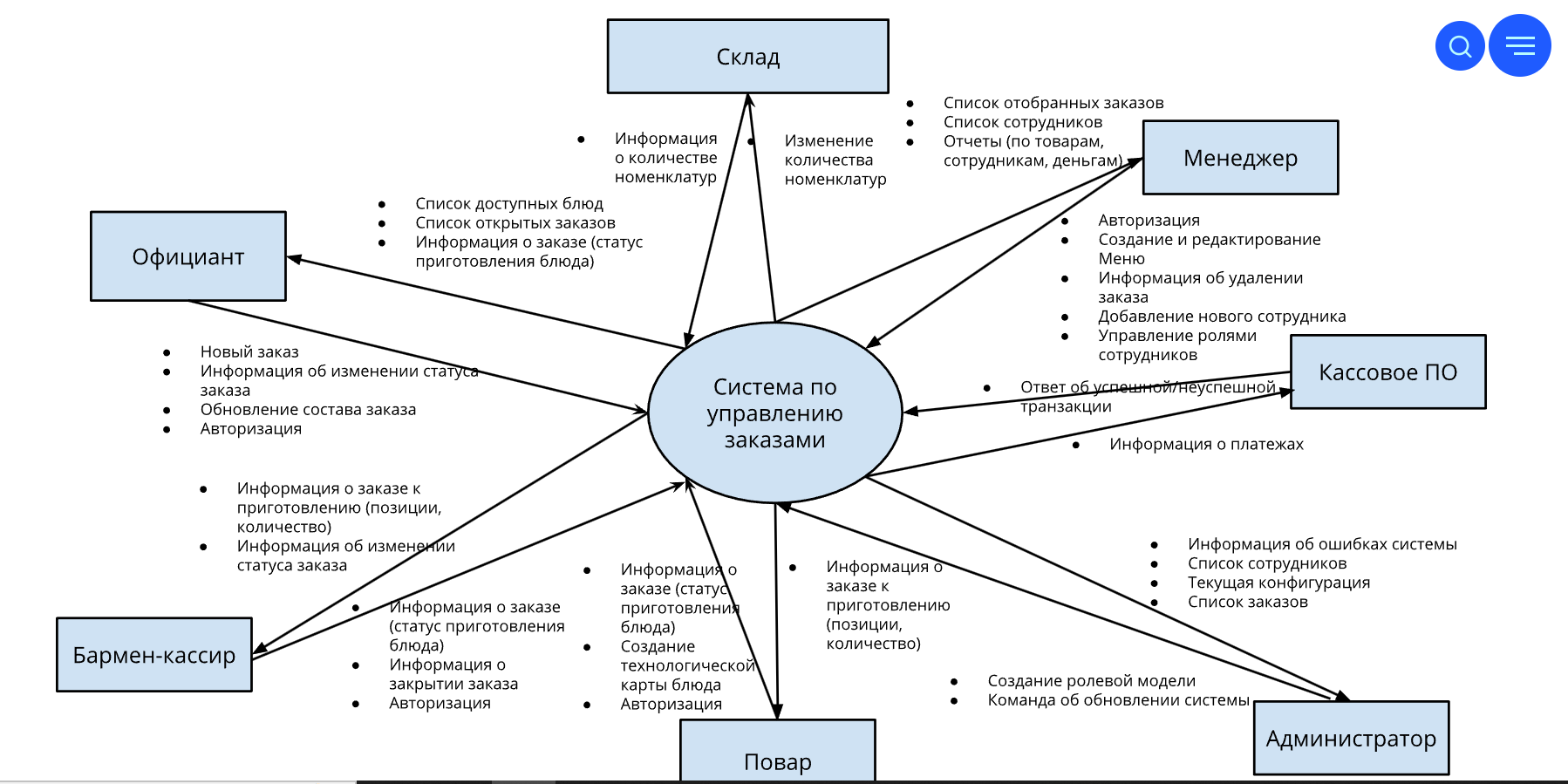
**Контекстная диаграмма классов**

Контекстная диаграмма относится к категории диаграмм, описывающих систему на уровне «чёрного ящика» — а именно, только внешние свойства (в данном случае — потоки данных), но не содержание системы.

Контекстная диаграмма содержит 3 основных компонента:

* Проектируемый объект (например, система)
* Взаимодействующие с проектируемым объектом элементы окружения (группы пользователей, смежные системы)
* Потоки данных (исходящие и входящие)

Пример контекстной диаграммы для программной системы управления Заказами в ресторане:



Потоки данных могут передаваться между окружением и (программной) системой любым образом — с помощью графического пользовательского интерфейса (GUI), командной строки (CLI), программных вызовов (API), почтовых сообщений и т.д.

Если система имеет физические интерфейсы, то это могут быть разнообразные джойстики, рукоятки управления, специализированные клавиатуры, датчики распознавания движения, изображения, жестов и т.д.

В стандартной форме не принято указывать виды интерфейсов взаимодействия и тем более протоколы, чтобы не усложнять диаграмму и не пытаться принимать вторичные решения, пока не приняты первичные.

Как создавать контекстную диаграмму

Контекстная диаграмма может разрабатываться в ходе рабочего семинара, в ходе серии интервью или на основе результатов серии интервью.

Контекстную диаграмму можно рисовать на маркерной доске, в среде проектирования или в онлайн-инструменте (Google Draw, Draw.io, Miro и т.д.). Мы рекомендуем маркерную доску или онлайн-инструмент с совместным редактированием.

Порядок разработки контекстной диаграммы на рабочем семинаре:

1. Из числа заинтересованных лиц собирается рабочая группа (обычно от 3 до 5 человек)
2. Рабочая группа фиксирует в центре диаграммы название конкретной системы
3. Рабочая группа выдвигает и отображает группы пользователей, которые должны взаимодействовать с системой, обсуждает их перечень, дополняет его
4. Рабочая группа выдвигает и отображает смежные системы, которые должны взаимодействовать с системой, обсуждает их перечень, дополняет его
5. Рабочая группа последовательно проходит по каждому элементу окружения и описывает потоки данных, связывающие его с системой
6. Рабочая группа проводит тестирование контекстной диаграммы, дополняя диаграмму по ходу тестирования

Для экономии времени участников тестирование можно производить 1-2 участниками, однако это ухудшает осведомлённость группы о найденных проблемах, поэтому мы не рекомендуем так делать.

**Содержание**

**14 Оконные классы**

[**15 Основные подходы к разработке ПО +**](#_15_вопрос)

[**16 Перегрузка операторов. Оператор сложения +**](#_16_и_17)

[**17 Перегрузка операторов. Оператор сравнения +**](#_16_и_17)

[**18 Полиморфизм: назначение и семантика +**](#_Вопрос_18)

[**19 Принципы ооп +**](#_Вопрос_19._Принципы)

[**20 Диаграмма прецедентов +**](#_Вопрос_20._Диаграмма)

[**21 Рефлексия и интроспекция +**](#_Вопрос_21._Рефлексия)

[**22 Сериализация и десериализация. Pickle +**](#_Вопрос_22_и)

[**23 Файлы json +**](#_Вопрос_22_и)

[**24 Статический полиморфизм**](#_Вопрос_24._Статический)

[**25 Типовые компоненты для обслуживания бд +**](#_Вопрос_25)

[**26 Диаграмма классов +**](#_Вопрос_26)

15 вопрос

Основные подходы к разработке ПО.

**Понятие ООП.**

На начальном этапе важно определить **архитектуру** системы, которая задает основу для дальнейшего проектирования. В этом процессе особое внимание уделяется выбору подходящих алгоритмов, которые обеспечивают эффективное выполнение задач и оптимизацию работы программы. После определения архитектуры следует этап **дизайна**, который включает в себя разработку интерфейса и визуального представления программы. Этот этап направлен на создание удобного и интуитивно понятного пользовательского опыта. Важным аспектом является **кодирование**, где разработчики пишут исходный код с учетом принципов **оптимизации** и **автоматизации** процессов. **Интеграция** различных компонентов системы и тщательное **тестирование** на всех этапах разработки помогают выявить и устранить ошибки, что существенно улучшает стабильность и надежность конечного продукта.

**Этапы разработки программного обеспечения**

* **Проектирование:** Определение архитектуры и дизайна системы, чтобы заложить основу для будущего кодирования.
* **Кодирование:** Написание исходного кода с использованием выбранных языков программирования.
* **Тестирование:** Проверка и отладка кода для выявления и исправления ошибок.
* **Оптимизация:** Улучшение производительности и эффективности кода.
* **Автоматизация:** Использование инструментов и скриптов для автоматизации процессов, что повышает скорость и точность разработки.
* **Интеграция:** Объединение различных систем и компонентов для обеспечения их совместной работы.

**Основные методы тестирования**

* **Юнит-тестирование:** Проверка отдельных компонентов или модулей кода на соответствие спецификациям.
* **Интеграционное тестирование:** Проверка взаимодействия между различными модулями и системами.
* **Системное тестирование:** Полное тестирование системы для проверки соответствия всем требованиям.
* **Тестирование производительности:** Оценка быстродействия и оптимизации программного обеспечения.

**Понятие ООП**

Объектно-ориентированное программирование (ООП) — это парадигма программирования, основанная на использовании объектов, которые объединяют данные и методы для их обработки. Основной идеей ООП является моделирование реальных сущностей в виде объектов, чтобы упростить разработку, поддержку и масштабирование программ. Парадигмой называют набор правил и критериев, которые соблюдают разработчики при написании кода. Если представить, что код — это рецепт блюда, то парадигма — то, как рецепт оформлен в кулинарной книге. Парадигма помогает стандартизировать написание кода.

Самый простой способ объяснить и понять ООП — воспользоваться метафорой. Метафорой объекта в ООП является объект реального мира, например, человек. Объекты надо отличать между собой и у них есть что-то, что их определяет. Например, для человека это может быть имя, когда мы говорим про нашего знакомого Васю, и все понимают о ком речь. Люди неким образом похожи друг на друга. Подмножество людей, обладающих одинаковым набором свойств (имя, фамилия, возраст и т.д.) и общим поведением, будет называться класс. Возьмем для примера сотрудников нашей компании. Для каждого из нас определен департамент (я, например, в департаменте разработки ПО числюсь, ДРПО), должность, уровень зарплаты и т.д. Эти свойства обычно определяют в момент, когда в компанию приходит новый сотрудник. У человека можно запросить информацию по его навыкам или попросить помочь коллеге — это общее поведение для всех сотрудников.

Зарплату сотрудника знает он сам, его руководитель и бухгалтер, остальные — нет. Такое сокрытие данных называется инкапсуляция. Какие свойства и поведение будет доступно другим объектам обычно определяется на уровне класса. Руководитель отдела также является сотрудником, но он обладает рядом дополнительных свойств, например, у него есть подчиненные. Таким образом класс «руководитель», расширяет класс «сотрудник» или, другими словами, происходит наследование. При этом между классами устанавливается отношение «является» — то есть любой руководитель является сотрудником, но не наоборот — не каждый сотрудник является руководителем. Если у класса больше одного наследника, то образуется иерархия. Классы, которые являются родственниками в иерархии не связаны отношением «является», например, бухгалтер является сотрудником, но бухгалтер не является руководителем.

При помощи этих правил иерархию можно проверить на корректность. Если взять ведомость со списком всех сотрудников, в нее очевидным образом попадут и руководители, и бухгалтеры, но в общем списке они не будут отличаться от других сотрудников. Если мы захотим уточнить список подчиненных у каждого руководителя, то нам понадобится подготовить отдельную ведомость со свойствами, специфичными для класса «руководитель». Такое свойство объектов называется полиморфизмом, где состав свойств и поведение будет определяться классом, через который мы смотрим на объект: мы можем обращаться к объекту, как и к любому из предков его класса, но это не верно для потомков или других родственников.

Так мы рассмотрели, как связаны объекты и классы, и такие понятия, как: инкапсуляция, наследование и полиморфизм. Все это — базовые понятия ООП.

16 и 17 вопрос

Перегрузка операций. Операция сложения. Операция сравнения

Перегрузка операторов — один из способов реализации полиморфизма, когда мы можем задать свою реализацию какого-либо метода в своём классе.

**Перегрузка оператора** — это возможность переопределять различные операторы в классах, то есть менять операции, которые они выполняют, в зависимости от контекста.

Python позволяет определять для своих классов встроенные операторы, такие как операции сложения, вычитания и т.д. Для этого в модуле **operator** определен ряд функций:

|  |  |  |
| --- | --- | --- |
| **Операция** | **Синтаксис** | **Функция** |
| Сложение | a + b | \_\_add\_\_(a, b) |
| Проверка наличия | obj in seq | \_\_contains\_\_(seq, obj) |
| Деление | a / b | \_\_truediv\_\_(a, b) |
| Целочисленное Деление | a // b | \_\_floordiv\_\_(a, b) |
| Степень | a \*\* b | \_\_pow\_\_(a, b) |
| Присвоение по индексу | obj[k] = v | \_\_setitem\_\_(obj, k, v) |
| Удаление по индексу | del obj[k] | \_\_delitem\_\_(obj, k) |
| Обращение по индексу | obj[k] | \_\_getitem\_\_(obj, k) |
| Остаток от деления | a % b | \_\_mod\_\_(a, b) |
| Умножение | a \* b | \_\_mul\_\_(a, b) |
| Вычитание | a - b | \_\_sub\_\_(a, b) |
| Проверка на Truе/False | obj | \_\_bool\_\_(obj) |
| Меньше чем | a < b | \_\_lt\_\_(a, b) |
| Меньше чем или равно | a <= b | \_\_le\_\_(a, b) |
| Равенство | a==b | \_\_eq\_\_(a, b) |
| Неравенство | a != b | \_\_ne\_\_(a, b) |
| Больше чем или равно | a >= b | \_\_ge\_\_(a, b) |
| Больше чем | a > b | \_\_gt\_\_(a, b) |
| Сложение с присваиванием | a += b | \_\_iadd\_\_(a, b) |
| Объединение с присваиванием | a += b | \_\_iconcat\_\_(a, b) |

Чтобы определить оператор для некоторого класса, данный класс должен реализовать соответствующую функцию. Так, для определения оператора сложения применяется функция **\_\_add\_\_()**, поэтому внутри класса нам надо определить данную функцию. Например:

class Counter:

    def \_\_init\_\_(self, value):

        self.value = value

    # переопределение оператора сложения

    def \_\_add\_\_(self, other):

        return Counter(self.value + other.value)

counter1 = Counter(5)

counter2 = Counter(15)

counter3 = counter1 + counter2

print(counter3.value)       # 20

Причем в данном случае реализован не единственно возможный вариант оператора сложения. Так, в примере выше второй параметр функции представлял другой объект Counter. Но в реальности это может быть любой тип. Например, что, если мы хотим складывать Counter не с другим объектом Counter, а с числом. Тогда мы могли определить следующий оператор:

class Counter:

    def \_\_init\_\_(self, value):

        self.value = value

    def \_\_add\_\_(self, other):

        return Counter(self.value + other)

counter1 = Counter(5)

counter3 = counter1 + 6

print(counter3.value)       # 11

Еще один пример:

class Point:

def \_\_init\_\_(self, x=0, y=0):

self.x = x

self.y = y

def \_\_str\_\_(self):

return "({0},{1})".format(self.x, self.y)

def \_\_add\_\_(self, other):

x = self.x + other.x

y = self.y + other.y

return Point(x, y)

# задаем координаты точек p1, p2

p1 = Point(1, 2)

p2 = Point(2, 3)

print(p1+p2)

**Перегрузка операторов сравнения**

Python не ограничивается перегрузкой арифметических операторов. Перегружать можно и операторы сравнения.

Допустим, нам надо реализовать оператор «меньше чем» < в классе Point. Мы будем сравнивать, какая из точек дальше от начала координат. Реализовать это можно вот так.

# Перегрузка оператора «меньше чем»

class Point:

def \_\_init\_\_(self, x=0, y=0):

self.x = x

self.y = y

def \_\_str\_\_(self):

return "({0},{1})".format(self.x, self.y)

def \_\_lt\_\_(self, other):

self\_mag = (self.x \*\* 2) + (self.y \*\* 2)

other\_mag = (other.x \*\* 2) + (other.y \*\* 2)

return self\_mag < other\_mag

# задаем значения точек p1, p2, p3

p1 = Point(1,1)

p2 = Point(-2,-3)

p3 = Point(1,-1)

# используем «меньше чем»

print(p1<p2)

print(p2<p3)

print(p1<p3)

**Вывод:**

True  
False  
False

**Истинность объекта**

Определение функции **\_\_bool\_\_** позволяет установить истинность объекта или фактически преобразовать объект к значениям True/False. Например:

class Counter:

    def \_\_init\_\_(self, value):

        self.value = value

    def \_\_bool\_\_(self):

        return self.value > 0

def test(counter):

    if counter: print("Counter = True")

    else: print("Counter = False")

counter1 = Counter(3)

test(counter1)              # Counter = True

counter2 = Counter(-3)

test(counter2)              # Counter = False

**Операторы, которые возвращают значение bool**

Ряд операций призваны возвращать логическое значение **True** или **False**. Например, операции сравнения:

class Counter:

    def \_\_init\_\_(self, value):

        self.value = value

    def \_\_gt\_\_(self, other):

        return self.value > other.value

    def \_\_lt\_\_(self, other):

        return self.value < other.value

counter1 = Counter(1)

counter2 = Counter(2)

if counter1 > counter2:

    print("counter1 больше чем counter2")

elif counter1 < counter2:

    print("counter1 меньше чем counter2")

else:

    print("counter1 и counter2 равны")

**Реализация операторов парами**

Некоторые операторы - операторы сравения удобнее реализовать парами. Если мы реализуем оператор **==**, то можно сразу реализовать и оператор **!=**. Причем чтобы не прописывать одну и ту же логику по два раза, можно реализовать один оператор через другой:

class Counter:

    def \_\_init\_\_(self, value):

        self.value = value

    def \_\_eq\_\_(self, other): return self.value == other.value

    def \_\_ne\_\_(self, other): return not (self == other)

    def \_\_gt\_\_(self, other): return self.value > other.value

    def \_\_le\_\_(self, other): return not (self > other)

    def \_\_lt\_\_(self, other): return self.value < other.value

    def \_\_ge\_\_(self, other): return not (self < other)

c1 = Counter(1)

c2 = Counter(2)

print(c1 == c2)     # False

print(c1 != c2)     # True

print(c1 < c2)      # True

print(c1 >= c2)     # False

Вопрос 18

**Полиморфизм: назначение и семантика**

Полиморфизм — это свойство функции обрабатывать различные типы объектов и классов.

Основная задача полиморфизма — оптимизация [кода](https://kokoc.com/blog/knigi-dlya-programmistov/) и удаление дублирующих друг друга команд. Если проще, функции из разных классов будут выполнять одну и ту же команду. Для этого не нужно прописывать отдельные команды для каждого участка кода.

Простыми словами, полиморфизм — это то, что позволяет использовать один и тот же метод для разных объектов.

В Python существуют различные типы полиморфизма, и мы рассмотрим наиболее распространенные из них:

* **Функциональный полиморфизм** – использование функций разными способами
* **Классовый полиморфизм** – использование методов в разных классах
* **Полиморфизм и наследование** – как наследование помогает в реализации полиморфизма

**Преимущества полиморфизма:**

* **Упрощение кода.** С помощью подобных методов можно снизить дублирование кода. В зависимости от его разновидности абстрактный базовый класс или общий интерфейс указывает на общее поведение функции вне зависимости от типа данных.
* **Расширяемость и гибкость.** Появляется возможность, не изменяя код, добавлять новые операции и типы объектов. При использовании общих методов или интерфейса новые классы легко встраиваются в код.
* **Читаемость кода.** Благодаря такому способу код становится более понятным для других разработчиков. Общие методы работы с разными классами делают его простыми для понимания.
* **Повторное использование кода**. Один и тот метод применяется для разных типов данных. Благодаря полиформизму удаляет необходимость создавать одинаковые функции для разных классов, достаточно сделать их универсальными.
* **Инкапсуляция и абстракция.** С помощью полиморфизма можно заниматься в первую очередь поведением объектов, не обращая внимания на отдельные методы их реализации. Также упрощается инкапсуляцию, разделяя интерфейс и детали реализации кода.
* **Упрощение тестирования.** Так как для разных классов используются одни методы, можно значительно ускорить тестирование. Для теста достаточно написать проверку одного метода, который будет работать для всех классов и участков кода.

**1. Функциональный полиморфизм**

В Python функции по своей природе полиморфны. Это означает, что мы можем использовать одну и ту же функцию разными способами.

**Пример: Встроенные функции.** Встроенные функции Python, такие как len(), демонстрируют функциональный полиморфизм. Функцию len() можно использовать с разными типами данных, такими как строки, списки и словари. Несмотря на разницу в типах, len() работает с ними всеми.

# Использование len() для строки

text = "Привет, мир!"

print(len(text)) # Результат: 13

# Использование len() для списка

numbers = [1, 2, 3, 4, 5]

print(len(numbers)) # Результат: 5

# Использование len() для словаря

data = {'name': 'Алиса', 'age': 30}

print(len(data)) # Результат: 2

Здесь функция len() адаптируется к типу объекта, с которым работает, возвращая длину строк, списков и словарей соответственно.

**2. Классовый полиморфизм -** здесь полиморфизм позволяет использовать одни и те же имена методов в разных классах. Каждый класс может иметь свою версию метода, адаптированную под свои нужды.

**Пример: Классы фигур.** У нас есть различные фигуры, каждая из которых имеет метод area(). Каждый класс фигуры реализует свою версию метода area(), который по-разному вычисляет площадь в зависимости от фигуры.

class Rectangle:

def \_\_init\_\_(self, width, height):

self.width = width

self.height = height

def area(self):

return self.width \* self.height

class Circle:

def \_\_init\_\_(self, radius):

self.radius = radius

def area(self):

return 3.14159 \* (self.radius \*\* 2)

# Создаем экземпляры

rectangle = Rectangle(4, 5)

circle = Circle(3)

# Вызов метода area для обеих фигур

print("Площадь прямоугольника:", rectangle.area()) # Результат: 20

print("Площадь круга:", circle.area()) # Результат: 28.2735

Здесь оба класса Rectangle и Circle имеют метод area(). При вызове area() для каждого экземпляра Python знает, какую версию метода использовать. Это позволяет использовать один и тот же интерфейс (имя метода) по-разному в зависимости от объекта.

**3. Полиморфизм и наследование**

В Python **наследование** – это отличный способ добиться полиморфизма. Создавая базовый класс с методом, который переопределяется в производных классах, мы можем добиться полиморфизма через наследование.

**Пример: Классы животных.** Создадим базовый класс Animal с методом make\_sound(), затем создадим подклассы Dog и Cat, которые переопределяют make\_sound().

class Animal:

def make\_sound(self):

raise NotImplementedError("Подклассы должны реализовать этот метод")

class Dog(Animal):

def make\_sound(self):

return "Гав!"

class Cat(Animal):

def make\_sound(self):

return "Мяу!"

# Создаем экземпляры

dog = Dog()

cat = Cat()

# Вызов метода make\_sound для обоих животных

print(dog.make\_sound()) # Результат: Гав!

print(cat.make\_sound()) # Результат: Мяу!

Здесь метод make\_sound() определен в базовом классе Animal, но каждый подкласс реализует свою версию. При вызове make\_sound() для экземпляра Dog или Cat Python вызывает соответствующий метод в зависимости от типа объекта. Это и есть полиморфизм в действии!

**Пример 1: Полиморфизм в GUI приложениях.** В графических пользовательских интерфейсах (GUI) полиморфизм позволяет создавать различные виды кнопок, полей ввода и других элементов интерфейса, которые могут обрабатываться одинаково. Это упрощает разработку и поддержку интерфейсов.

class Button:

def click(self):

pass

class SubmitButton(Button):

def click(self):

return "Submit form"

class CancelButton(Button):

def click(self):

return "Cancel form"

buttons = [SubmitButton(), CancelButton()]

for button in buttons:

print(button.click())

В этом примере классы SubmitButton и CancelButton наследуют от класса Button и переопределяют метод click. Это позволяет обрабатывать различные типы кнопок одинаково, что упрощает разработку интерфейсов.

**Пример 2: Полиморфизм в обработке файлов**

Полиморфизм позволяет создавать универсальные функции для работы с различными типами файлов, такими как текстовые файлы, бинарные файлы и т.д. Это делает код более модульным и переиспользуемым.

class FileProcessor:

def process(self, file):

pass

class TextFileProcessor(FileProcessor):

def process(self, file):

return f"Processing text file: {file}"

class BinaryFileProcessor(FileProcessor):

def process(self, file):

return f"Processing binary file: {file}"

processors = [TextFileProcessor(), BinaryFileProcessor()]

for processor in processors:

print(processor.process("example.txt"))

В этом примере классы TextFileProcessor и BinaryFileProcessor наследуют от класса FileProcessor и переопределяют метод process. Это позволяет обрабатывать различные типы файлов одинаково, что упрощает разработку и поддержку кода.

**Задача: Реализовать полиморфизм в системе оплаты**

Представьте, что вы разрабатываете систему оплаты, которая должна поддерживать различные способы оплаты, такие как кредитные карты, PayPal и банковские переводы. Используйте полиморфизм для реализации этой системы.

class PaymentMethod:

def pay(self, amount):

pass

class CreditCard(PaymentMethod):

def pay(self, amount):

return f"Paying {amount} using Credit Card"

class PayPal(PaymentMethod):

def pay(self, amount):

return f"Paying {amount} using PayPal"

class BankTransfer(PaymentMethod):

def pay(self, amount):

return f"Paying {amount} using Bank Transfer"

payments = [CreditCard(), PayPal(), BankTransfer()]

for payment in payments:

print(payment.pay(100))

В этом примере классы CreditCard, PayPal и BankTransfer наследуют от класса PaymentMethod и переопределяют метод pay. Это позволяет обрабатывать различные способы оплаты одинаково, что упрощает разработку и поддержку системы оплаты.

Вопрос 19. Принципы ООП

Всё объектно-ориентированное программирование строится на 4 понятиях: инкапсуляции, наследовании, полиморфизме, абстракции.

* ***Наследование*** — это концепция ООП, которая позволяет создавать новые классы на основе существующих. Класс, который наследует свойства и методы от другого класса, называется подклассом (или классом-потомком), а класс, от которого он наследует, называется суперклассом (или классом-родителем). Подкласс может добавлять новые свойства и методы или изменять существующие. Наследование позволяет повторно использовать код и уменьшать количество дублированного кода.
* ***Инкапсуляция*** — это концепция ООП, которая позволяет скрыть детали реализации и защитить данные от изменений извне.
* ***Полиморфизм*** — это концепция ООП, которая основана на наследовании и позволяет использовать объекты разных классов, но имеющих схожее поведение. Например, каждое животное может издавать какие-то звуки. Однако такое животное, как собака обычно издает звук «Гав!», а кошка — звук «Мяу». Иными словами, полиморфизм — это переопределение методов, унаследованных от суперкласса в подклассах.

Объявим класс «Кошка» и будем объяснять ООП на нём:

class Cat():

def \_\_init\_\_(self, breed, color, age):

self.breed = breed

self.color = color

self.age = age

def meow(self):

print('Мяу!')

def purr(self):

print('Мрррр')

**Метод \_\_init**\_\_ — инициализатор класса, он вызывается сразу после создания объекта, чтобы присваивать значения динамическим атрибутам.

**self** — ссылка на текущий объект, она даёт доступ к атрибутам и методам, с которыми вы работаете.

Мы создали класс Cat, в котором объявили три атрибута: порода — breed, цвет — color и возраст — age. А ещё добавили два метода, чтобы наша кошка умела мяукать — meow() и мурчать — purr().

Создадим пару объектов нашего класса:

cat1 = Cat('Абиссинская', 'Рыжая', 4)

cat2 = Cat('Британская', 'Серая', 2)

**Инкапсуляция**

Доступ к данным объекта должен контролироваться, чтобы пользователь не мог изменить их в произвольном порядке и что-то поломать. Поэтому для работы с данными программисты пишут методы, которые можно будет использовать вне класса и которые ничего не сломают внутри.

Вернёмся к проекту Кот. Мы можем разрешить изменять атрибут «возраст», но только в большую сторону, а атрибуты «порода» и «цвет» лучше открыть только для чтения — ведь порода кошки не меняется, а цвет если и меняется, то не по её инициативе.

class Cat():

def \_\_init\_\_(self, breed, color, age):

self.\_breed = breed

self.\_color = color

self.\_age = age

@property

def breed(self):

return self.\_breed

@property

def color(self):

return self.\_color

@property

def age(self):

return self.\_age

@age.setter

def age(self, new\_age):

if new\_age > self.\_age:

self.\_age = new\_age

return self.\_age

Сначала мы сделали все атрибуты закрытыми с помощью символа **\_.** Он говорит интерпретатору, что эта переменная будет доступна только внутри методов класса.

Нам всё ещё нужно получать доступ к атрибутам, поэтому мы предоставляем его через @property и объявляем для каждого атрибута свой метод — breed, color, age. В каждом из этих методов мы возвращаем значение нашего закрытого атрибута. Это делает его доступным только для чтения.

И последнее — мы должны позволить пользователям увеличивать возраст кота. Для этого воспользуемся @age.setter и ещё раз объявим метод age, а внутри него напишем простое условие и вернём значение атрибута.

Создадим экземпляр класса и выведем значения атрибутов:

cat = Cat('Абиссинская', 'Рыжая', 4)

print(cat.breed) # Абиссинская

print(cat.color) # Рыжая

print(cat.age) # 4

И попробуем изменить атрибут age:

cat.age = 5

print(cat.age) # 5

А теперь сделаем это с другим атрибутом:

cat.breed = 'Сиамская'

print(cat.breed) # AttributeError: can't set attribute on line 34 in main.py

**Наследование**

Классы могут передавать свои атрибуты и методы классам-потомкам. Например, мы хотим создать новый класс «Домашняя кошка». Он практически идентичен классу «Кошка», но у него появляются новые атрибуты «хозяин» и «кличка», а также метод «клянчить вкусняшку».

Достаточно объявить «Домашнюю кошку» наследником «Кошки» и прописать новые атрибуты и методы — вся остальная функциональность перейдёт от родителя к потомку.

**Объявим новый класс:**

class HomeCat(Cat):

def \_\_init\_\_(self, breed, color, age, owner, name):

super().\_\_init\_\_(breed, color, age)

self.\_owner = owner

self.\_name = name

@property

def owner(self):

return self.\_owner

@property

def name(self):

return self.\_name

def getTreat(self):

print('Мяу-мяу')

В первой строке мы как раз наследуем все методы и атрибуты класса Cat. А чтобы всё создалось корректно, мы должны вызвать **метод super()** в методе \_\_init\_\_() и через него заполнить атрибуты класса-родителя. Поэтому мы и передаём в этот метод «породу», «окрас» и «возраст».

Кроме атрибутов для класса-родителя у класса-потомка есть и собственные атрибуты: «хозяин» — owner и «кличка» — name. Их мы будем использовать только в этом классе, поэтому они будут недоступны для класса-родителя.

Мы сразу сделали атрибуты класса-потомка закрытыми и объявили для них собственные методы. А также добавили метод **​​**getTreat(), которого нет в классе-родителе.

**Создадим объект класса:**

my\_cat = HomeCat('Сиамская', 'Белая', 3, 'Иван', 'Роза')

print(my\_cat.owner)

print(my\_cat.breed)

my\_cat.getTreat() # Мяу-мяу

my\_cat.purr() # Мрррр

**Полиморфизм**

Этот принцип позволяет применять одни и те же команды к объектам разных классов, даже если они выполняются по-разному. Например, помимо класса «Кошка», у нас есть никак не связанный с ним класс «Попугай» — и у обоих есть метод «спать». Несмотря на то что кошки и попугаи спят по-разному (кошка сворачивается клубком, а попугай сидит на жёрдочке), для этих действий можно использовать одну команду.

**У нас есть два класса — «Кошка» и «Попугай»:**

class Cat:

def sleep(self):

print('Свернулся в клубок и сладко спит.')

class Parrot:

def sleep(self):

print('Сел на жёрдочку и уснул.')

А теперь пусть у нас есть метод, который ожидает, что ему на вход придёт объект, у которого будет метод sleep:

def homeSleep(animal):

animal.sleep()

Посмотрим, как это будет работать:

cat = Cat()

parrot = Parrot()

homeSleep(cat) # Свернулся в клубок и сладко спит.

homeSleep(parrot) # Сел на жёрдочку и уснул.

Хотя классы разные, их одноимённые методы работают похожим образом.

**Абстракция**

При создании класса мы упрощаем его до тех атрибутов и методов, которые нужны именно в этом коде, не пытаясь описать его целиком и отбрасывая всё второстепенное. Например, у всех хищников есть метод «охотиться», поэтому все животные, которые являются хищниками, автоматически будут уметь охотиться.

**Рассмотрим класс Predator:**

class Predator:

def hunt(self):

print('Охотится...')

Этот класс будет общим для всех животных, которые являются хищниками, — например, кошек:

class Cat(Predator):

def \_\_init\_\_(self, name, color):

super().\_\_init\_\_()

self.\_name = name

self.\_color = color

@property

def name(self):

return self.\_name

@property

def color(self):

return self.\_color

У кошки есть свои атрибуты: «имя» — name и «окрас» — color. Но при этом она потомок хищников, а значит, умеет охотиться:

cat = Cat('Даниэла', 'Чёрный')

cat.hunt() # Охотится…

Вопрос 20. Диаграмма прецедентов

**Что такое язык UML**

**UML**, или **Unified Modeling Language**, — это унифицированный язык моделирования. Его используют, чтобы создавать диаграммы и схемы для визуализации процессов и явлений.

Слово «унифицированный» означает, что схемы на UML будут понятны всем, кто знаком с ним. То есть у языка есть определённые правила, по которым применяются все стрелочки, кружки и квадраты.

**Для чего используют язык UML**

UML нужен, чтобы превращать абстракции в визуальные модели и схемы. Поэтому применять его можно во многих областях:

* в программировании — чтобы наглядно видеть связи между классами и другими частями приложения или чтобы построить карту поведения пользователя на сайте;
* в дизайне — чтобы создавать интерфейсы и понимать, как пользователи будут взаимодействовать с ними;
* в бизнесе — чтобы визуально представлять, как работают бизнес-процессы или ведётся документооборот в организации.

Важная особенность UML — этот язык поддерживает объектно-ориентированный подход, где все сущности представлены как объекты с определенными свойствами и методами. В диаграммах UML легко изобразить объекты, связи между ними, наследование и возможности передачи данных от одного объекта к другому. UML-диаграммы хорошо показывают, как устроена та или иная структура, и помогают ничего не пропустить. С их помощью можно понять, как будет вести себя сущность в тех или иных ситуациях, рассмотреть ее с разных сторон.

**Диаграмма прецедентов**

UML-диаграмма use-кейсов применяется для краткого изложения сведений о пользователях проектируемой системы и их взаимодействии с ней.

**Из чего составляется диаграмма use-кейсов?**

Из следующих компонентов:

* **Актер**(Actor)
* Собственно**use-кейсы**
* **Ассоциации**между ними
* **Границы** **системы**(Boundaries), и
* **Отношения**(Relationships или связи)

**Актор в UML**

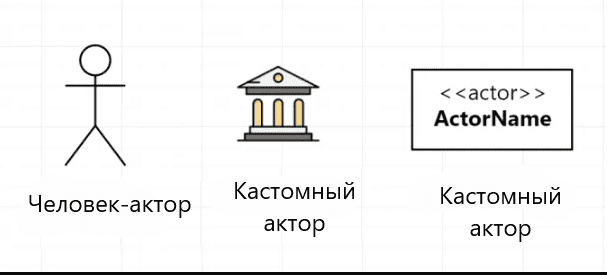
Актор — это пользователь, взаимодействующий с системой (действующее лицо). Это может быть человек, организация или даже сервер/система.

Чтобы идентифицировать актора, необходимо ответить на простой вопрос: «Кто будет **взаимодействовать**с приложением?»

Допустим, нужно создать сайт магазина. Кто будет актором?

* Пользователь, и
* Администратор

В диаграммах use-кейсов актор обычно изображается в виде человечка, но может быть изображен и как на рисунке:



**О use-кейсах, ассоциациях и границах системы**

* **Use кейсы**: широкие овалы, изображающие различные варианты использования, по которым может «проходить» пользователь.
* **Ассоциация**: линия, соединяющая («ассоциирующая») акторов и их use-кейсы. В сложных диаграммах важно четко знать, какие акторы связаны с какими use-кейсами.
* **Границы системы**: граница, задающая пределы системы для use кейсов. Все use кейсы, выходящие за пределы этой границы, будут считаться выходящими за пределы данной системы.

Например: клиент банка может видеть только свои транзакции, а не все проведенные транзакции в системе, поскольку такой use кейс выходит за границы системы.

**Отношения в UML-диаграммах**

Существует 3 типа отношений:

* **Включение**(Include)
* **Расширение**(Extend)
* **Обобщение**(Generalization)

**Отношение «ассоциации» (association)**

Используется для связи между прецедентами и акторами. Оно показывает, что актор взаимодействует с прецедентом. Отношение ассоциации связывает прецеденты с акторами и показывает, какой актор использует данный прецедент.

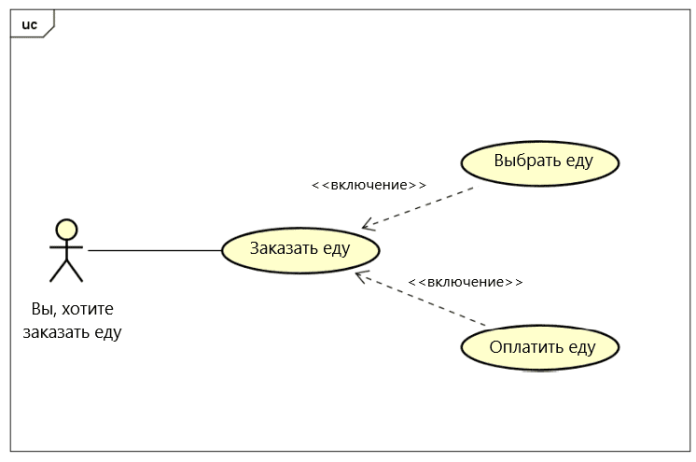
Изображение выглядит как текст, Шрифт, диаграмма, белый

Автоматически созданное описание

**Отношение включения**

Отношение включения — обязательная, неотъемлемая связь (отношение) между use кейсами.

Например, вы голодны и хотите зайти куда-то перекусить. Вы заходите во Вкусно и Точка и говорите: «Я хочу заказать еду». Но что нужно сделать перед этим? Конечно, сначала нужно выбрать еду. Пример диаграммы:

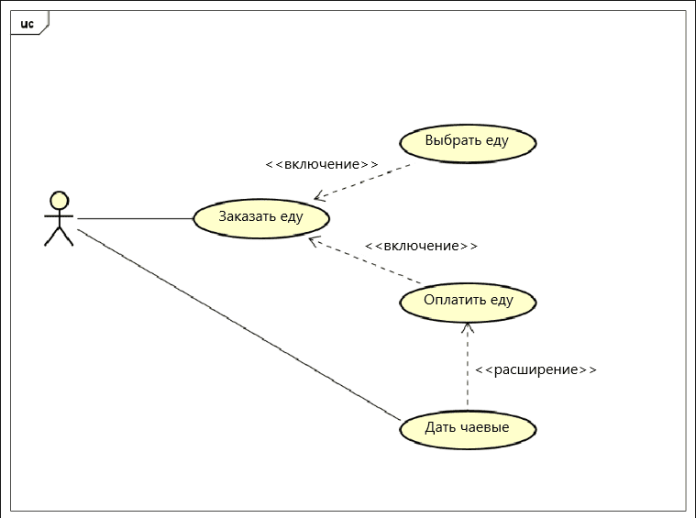


Это и есть **отношение включения**. Только когда вы сделаете выбор, можно переходить к оплате. Поэтому в диаграмму выше уже был добавлен еще один use кейс с включением — «Оплатить еду».

**Отношение расширения**

Как уже сказано, include — обязательная связь между use-кейсами. А расширение (extend) является необязательным. Это означает, что если use кейс не является обязательным, то актор может выбирать; если у актора есть выбор, то это уже отношение extend.

Вернемся к примеру. Когда оплачиваете еду, у вас появляются варианты действий: можете дать официанту чаевые, а можете и не дать. В нашей диаграмме прецедентов это отобразится следующим образом:



Просто запомним: Отношение “Include” — обязательное, а “Extend” — опциональное.

**Отношение обобщения**

Обобщение (генерализация, generalization) — это, по сути, демонстрация отношений типа «порождающая сущность / порожденная сущность» (parent/child), между use кейсами или акторами.

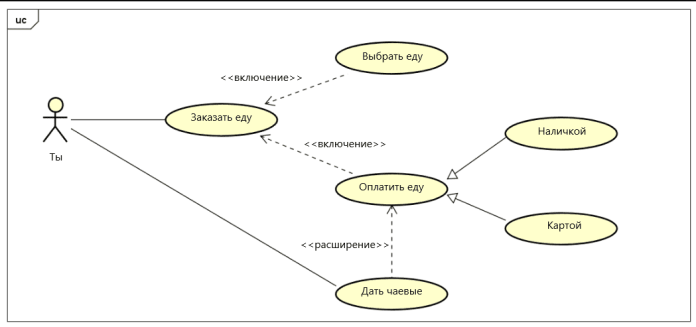
Например, обобщение use кейсов:

* Логин (parent) -> Вход с помощью телефона/электропочты (child)
* Оплата (parent) -> Оплатить с помощью Сбербанка / Наличкой (child)

Отношения обобщения акторов:

* Менеджер -> Персонал
* Оптовики -> Ритейлеры

Отношение обобщения в ИТ-системе заведения быстрого питания:



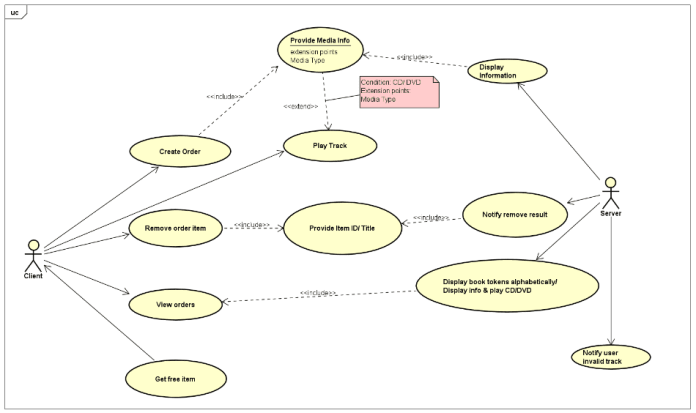
Изображение выглядит как диаграмма, линия, белый, оригами

Автоматически созданное описание

Обобщение помогает более четко изобразить диаграмму use кейсов.

При изучении объектно-ориентированного программирования (ООП), легко понять, что обобщение наследуется (наследование в объектно-ориентированной парадигме), то есть дочерняя структура «наследует» свойства и отношения родительской структуры других use кейсов.

**Пример диаграммы прецедентов**



**Как построить диаграмму вариантов использования**

* **Определить цели и задачи.** Определение целей и задач помогает установить понимание того, что необходимо отражать на диаграмма вариантов использования. Это помогает определить, какая функциональность должна быть включена в систему и выявить потенциальные проблемы и ограничения, связанные с дизайном и разработкой.

**Пример.** На примере модуля авторизации пользователя цели и задачи могут быть следующими:

**Цель** — описать, как пользователь может выполнить вход в систему.

**Задачи**:

1. Идентифицировать и описать все возможные варианты использования для авторизации пользователя.
2. Выявить действия, которые пользователь может выполнять на каждом этапе авторизации.
3. Идентифицировать все взаимодействия между пользователями и системой, связанные с авторизацией.
4. Предоставить четкое понимание, как система реагирует на каждое действие пользователя, связанное с авторизацией.
5. Помочь уточнить требования к системе по авторизации пользователя и выявить возможные проблемы и улучшения.

* **Определить акторов.** Определение акторов на диаграмме прецедентов позволяет определить, кто именно будет использовать систему и каким образом. Акторы представляют собой роли, которые пользователи могут играть в системе.

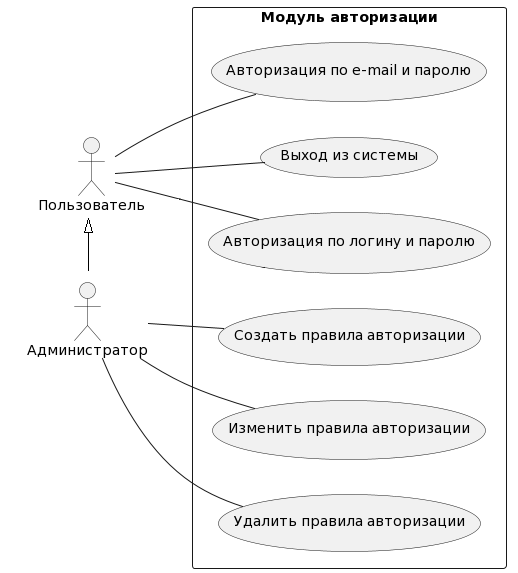
**Пример.** На диаграмме прецедентов для модуля авторизации могут быть следующие акторы:

1. Клиент – пользователь, который использует систему для авторизации и входа в свой аккаунт.
2. Администратор – пользователь, который имеет права на управление настройками авторизации.

* **Определить основные варианты использования системы**

|  |  |
| --- | --- |
| **Вариант использования** | **Акторы** |
| Авторизация по логину и паролю | Клиент, Администратор |
| Авторизация по e-mail и паролю | Клиент |
| Выход из системы | Клиент, Администратор |
| Создать новое правило авторизации | Администратор |
| Изменить правила авторизации | Администратор |
| Удалить правила авторизации | Администратор |

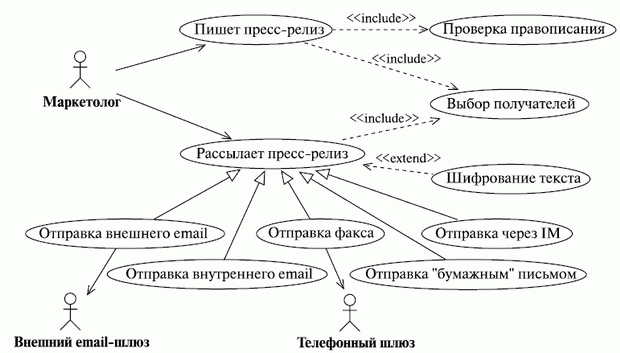
* **Установить отношения (связи) между акторами и вариантами использования**



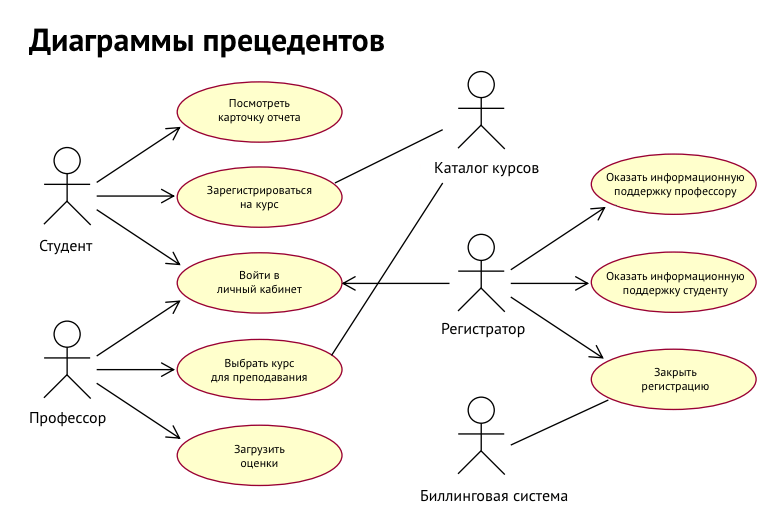
* **Выявить расширенные варианты использования и установить связи с основными сценариями.** Расширенные варианты использования описывают возможные варианты поведения системы, которые могут возникнуть при выполнении основных сценариев. Установление связей между расширенными вариантами использования и основными сценариями позволяет показать, какие дополнительные шаги могут быть выполнены в рамках основного сценария при возникновении определенных условий.

Изображение выглядит как текст, диаграмма, белый, снимок экрана

Автоматически созданное описание







Вопрос 21. Рефлексия и интроспекция.

В программировании **рефлексия** - это как зеркало, которое позволяет программе исследовать, интроспектировать и изменять свою собственную структуру и поведение во время выполнения. Это как если бы ваш код мог посмотреть на себя и сказать: "Эй, из чего я состояю?"

Интроспекция позволяет вам изучать атрибуты объекта во время выполнения программы, а рефлексия — манипулировать ими. Рефлексия — это способность компьютерной программы изучать и модифицировать свою структуру и поведение (значения, мета-данные, свойства и функции) во время выполнения. Простым языком: она позволяет вам вызывать методы объектов, создавать новые объекты, модифицировать их, даже не зная имён интерфейсов, полей, методов во время компиляции.

* **Функция type().** Это как спрашивать объект: "Что ты такое?"

Использование type()

x = 5

y = "Привет"

z = [1, 2, 3]

print(type(x)) # Вывод: <class 'int'>

print(type(y)) # Вывод: <class 'str'>

print(type(z)) # Вывод: <class 'list'>

В этом примере мы спрашиваем разные переменные, что они такого. Функция type(), что x - это целое число, y - строка, а z - список. Это как вести разговор с вашими переменными!

* **Функция isinstance().** Эта функция - как спрашивать: "Ты часть этой семьи?"

Использование isinstance()

x = 5

y = "Привет"

print(isinstance(x, int)) # Вывод: True

print(isinstance(y, str)) # Вывод: True

print(isinstance(x, str)) # Вывод: False

Здесь мы проверяем, является ли x экземпляром int (каков он и является), является ли y экземпляром str (каков он и является), и является ли x экземпляром str (каков он не является). Это как спрашивать кого-то, является ли он членом определенного клуба!

* **Функция issubclass().** Это как спрашивать: "Ты связан с этой семьей?"

Использование issubclass()

**class** **Animal**:

**pass**

**class** **Dog**(**Animal**):

**pass**

print(issubclass(Dog, Animal)) # Вывод: True

print(issubclass(Animal, Dog)) # Вывод: False

В этом примере мы проверяем, является ли Dog подклассом Animal (каков он и является), и является ли Animal подклассом Dog (каков он не является). Это как спрашивать, является ли пудель собакой.

* **Функция callable().** Эта функция проверяет, можно ли вызвать объект (как функцию).

Использование callable()

**def** **привет**():

print("Привет!")

x = 5

print(callable(привет)) # Вывод: True

print(callable(x)) # Вывод: False

Здесь мы проверяем, можно ли вызвать привет (который является функцией), и можно ли вызвать x (который просто число). Это как спрашивать: "Можно ли я использовать тебя как функцию?"

* **Функция getattr().** getattr() - это как вежливо спрашивать у объекта один из его атрибутов.

Использование getattr()

**class** **Person**:

name = "Алиса"

age = 30

p = Person()

print(getattr(p, "name")) # Вывод: Алиса

print(getattr(p, "job", "Нет работы")) # Вывод: Нет работы

В этом примере мы спрашиваем наш объект Person об его атрибуте name, который у него есть. Мы также спрашиваем атрибут job, которого у него нет, поэтому мы предоставляем значение по умолчанию "Нет работы".

* **Функция setattr().** setattr() - это как давать объекту новый атрибут или изменять существующий.

Использование setattr()

**class** **Car**:

color = "красный"

my\_car = Car()

print(my\_car.color) # Вывод: красный

setattr(my\_car, "color", "синий")

print(my\_car.color) # Вывод: синий

setattr(my\_car, "brand", "Toyota")

print(my\_car.brand) # Вывод: Toyota

Здесь мы меняем цвет нашего автомобиля с красного на синий, а затем даем ему новый атрибут brand.

* **Функция hasattr().** hasattr() - это как спрашивать у объекта: "У тебя есть этот атрибут?"

Использование hasattr()

**class** **Book**:

title = "Python для начинающих"

pages = 200

my\_book = Book()

print(hasattr(my\_book, "title")) # Вывод: True

print(hasattr(my\_book, "author")) # Вывод: False

В этом примере мы проверяем, есть ли у нашего объекта Book атрибут title (каков он и есть) и атрибут author (каков он не есть).

* **Функция dir()**. Эта функция - это как спрашивать у объекта: "Что ты можешь делать?"

Использование dir()

class Dog:

def \_\_init\_\_(self, name):

self.name = name

def bark(self):

print("Гав!")

my\_dog = Dog("Бuddy")

print(dir(my\_dog))

Это выведет список всех атрибутов и методов нашего объекта Dog, включая name и bark, а также некоторые встроенные атрибуты, которые все объекты имеют.

* **Функция \_\_class\_\_ -** атрибут **\_\_class\_\_**возвращает ссылку на класс, экземпляром которого объект является.

class MyClass:

pass

MyClass.\_\_class\_\_

* **Функция \_\_base\_\_**

Атрибут **\_\_base\_\_**вернет первый родительский класс для класса объекта, если он есть. Есть так же атрибут **\_\_bases\_\_**, в отличие от **\_\_base\_\_**он вернет всех непосредственных родителей класса объекта в виде кортежа:

class f(object):

pass

class d(f):

pass

class C(d ,f):

pass

print C.\_\_base\_\_

print C.\_\_bases\_\_

Получим:

<class '\_\_main\_\_.d'>

(<class '\_\_main\_\_.d'>, <class '\_\_main\_\_.f'>)

* **Функция \_\_dict\_\_** - это атрибут объекта python, который вернет словарь всех пользовательских атрибутов объекта и их значений.

На что следует обратить внимание, работая с \_\_dict\_\_?

Каждый класс в питоне является всего лишь объектом, который отличается от обычных объектов тем, что он может сам «создавать жизнь». Достигается эта сверхспособность наличием атрибута **\_\_metaclass\_\_** (который так же, как и \_\_dict\_\_, является системным; по умолчанию атрибут \_\_metaclass\_\_ класса имеет значение type, но его можно изменить), но она не делает класс чем-то особенным

Перейдем к примерам и объявим класс:

class MyObject:

MyField = 1

MyField2 = 2

def \_\_init\_\_(self):

self.a = "2"

print MyObject.\_\_dict\_\_ (python 2.7):

{'\_\_module\_\_': '\_\_main\_\_', 'MyField2': 2, 'MyField': 1, '\_\_init\_\_': <function \_\_init\_\_ at 0x01F5E630>, '\_\_doc\_\_': None}

Помимо наших атрибутов: полей *MyField*, *MyField2*и метода *\_\_init\_\_*в выводе мы увидели имя модуля (атрибут *\_\_module\_\_*) и атрибут *\_\_doc\_\_,* который должен содержать описание класса.

Обратите внимание, что в словаре нет атрибута **a**, поскольку атрибут **a** не является атрибутом класса-объекта **MyObject**, а является атрибутом объекта self, им создаваемым.

**Интроспекция** — это возможность запросить тип и структуру объекта во время выполнения программы.

Встроенные функции

| Функция | Возвращаемое значение |
| --- | --- |
| type(obj) | тип объекта |
| dir(obj) | пространство имен объекта |
| dir() | текущее пространство имен |
| id(obj) | адрес объекта в памяти |
| help(obj) | подсказка к объекту |
| hasattr(obj, 'attr') | наличие атрибута у объекта |
| getattr(obj, 'attr'[, default]) | возвращает атрибут объекта |
| locals() | словарь локальных переменных |
| globals() | словарь глобальных переменных |

Переменные

| Переменная | Значение |
| --- | --- |
| \_\_annotations\_\_ | словарь аннотаций типов |
| \_\_name\_\_ | имя модуля, если он импортирован, или '\_\_main\_\_', если запущен непосредственно. |
| \_\_file\_\_ | путь к файлу, содержащему объект в системе |

Модуль sys

| Функция | Возвращаемое значение |
| --- | --- |
| sys.argv | список аргументов, с которыми запущен интерпретатор |
| sys.executable | адрес интерпретатора в системе |
| sys.flags | флаги командной строки |
| sys.getrecursionlimit() | лимит рекурсии |
| sys.getsizeof(obj[, default]) | размер объекта |
| sys.hash\_info | параметры хеширования |
| sys.modules | словарь загруженных модулей |
| sys.path | список путей для поиска модулей |
| sys.platform | операционная система |
| sys.version | версия Python |
| sys.version\_info | версия Python в форме кортежа |

Вопрос 22 и 23. Сериализация и десериализация. Файл Pickle. JSON.

**Сериализация** — это процесс преобразования объекта в последовательность байтов или формат, который можно сохранить в файл, передать по сети или сохранить в базе данных.

**Десериализация** — это обратный процесс, при котором из этой последовательности байтов восстанавливается исходный объект.

Чтобы сохранить объект в файл (или передать его по сети), нужно преобразовать его в некую строку (или набор байт), которую будет легко записать в файл (и считать из файла) или передать по сети.

Вот 4 основных направления, где используется сериализация:

* **Сохранение состояния объектов**: Чтобы сохранить состояние объекта между запусками программы.
* **Передача данных**: Чтобы передавать объекты по сети между различными компонентами системы или между разными системами.
* **Кэширование**: Чтобы сохранить объекты в кэше для быстрого доступа.
* **Базы данных**: Чтобы хранить сложные структуры данных в базах данных.

Библиотек, которые занимаются сериализацией, очень много, и каждая создана под свои нужды..

Рассмотрим наиболее популярные из них:

* **Модуль pickle**
* **Модуль json**
* **Модуль yaml**
* **Модуль marshal**
* **Модуль shelve**

**Модуль pickle**

**pickle** — это встроенный модуль для сериализации и десериализации объектов Python. Он позволяет сохранять и восстанавливать практически любые объекты Python, включая пользовательские классы. Интерфейс pickle обеспечивает четыре метода: dump, dumps, load, и loads. Метод dump() сериализует в открытый файл (файл-подобный объект). Метод dumps() сериализует в строку. Метод load() десериализует из открытого файлового объекта. Метод loads() десериализует из строки. Сначала pickle сериализует объект, а затем преобразует объект в символьный поток, чтобы он содержал всю информацию, необходимую для восстановления объекта в другом скрипте

Пример использования pickle:

**import** pickle

data = {'name': 'Alice', 'age': 30, 'is\_student': False}

**with** open('data.pkl', 'wb') **as** file:

pickle.dump(data, file)

**with** open('data.pkl', 'rb') **as** file:

loaded\_data = pickle.load(file)

**print**(loaded\_data)

**pickle.dump()**

Как хранить данные с помощью Python pickle. Для этого мы должны сначала импортировать модуль pickle. Затем используйте функцию pickle.dump() для сохранения данных объекта в файле. Функция pickle.dump() принимает 3 аргумента. Первый аргумент – это объект, который вы хотите сохранить. Второй аргумент – это объект файла, который вы получаете, открывая желаемый файл в двоичном режиме записи (wb). И третий аргумент – это аргумент «ключ-значение». Этот аргумент определяет протокол. Существует два типа протокола – pickle.HIGHEST\_PROTOCOL и pickle.DEFAULT\_PROTOCOL.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Операционная система

Автоматически созданное описание

**pickle.load()**

Чтобы получить данные, шаги довольно просты. Для этого вам нужно использовать функцию pickle.load(). Основным аргументом функции загрузки pickle является объект файла, который вы получаете, открывая файл в режиме двоичного чтения (rb).

Изображение выглядит как текст, снимок экрана, программное обеспечение, дисплей

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

**Файл Json.**

В Python модуль json предоставляет инструменты для работы с форматом данных JSON (JavaScript Object Notation), который широко используется для обмена данными между различными языками программирования. Модуль json предоставляет функции для сериализации и десериализации (преобразования данных JSON в объекты Python).

JSON построен на двух структурах:

* Он хранит данные в парах имя / значение. Он рассматривается как объект, запись, словарь, хеш-таблица, список с ключами.
* Упорядоченный список значений рассматривается как массив, вектор, список или последовательность.

**Функция dump()**

Python предоставляет функцию dump() для передачи (кодирования) данных в формате JSON, осуществляет запись данных JSON в файл. Она принимает два позиционных аргумента: первый – это объект данных, который нужно сериализовать, а второй – файловый объект, в который должны быть записаны байты.

**Функция load()**

Функция load() используется для десериализации данных JSON в объект Python из файла. Рассмотрим следующий пример:

**Сравнение json.load() и json.loads()**

Функция json.load() используется для загрузки файла JSON, тогда как функция json.loads() используется для загрузки строки.

**Сравнение json.dump() и json.dumps()**

Функция json.dump() используется, когда мы хотим сериализовать объекты Python в файл JSON, а функция json.dumps() используется для преобразования данных JSON в виде строки для анализа и печати.

**Объекты Python преобразуются в следующие объекты JSON. Список приведен ниже:**

**Изображение выглядит как текст, Шрифт, снимок экрана, число

Автоматически созданное описание**

**Пример использования:**

Сериализация

import json

data = {

"name": "John Doe",

"age": 30,

"is\_employee": True,

"skills": ["Python", "JSON", "Data Analysis"]

}

# Сериализация в строку

json\_string = json.dumps(data, indent=4)

print(json\_string)

# Сериализация в файл

with open("data.json", "w") as write\_file:

json.dump(data, write\_file, indent=4)

Десериализация

import json

# Десериализация из строки

json\_string = '{"name": "Jane Doe", "age": 29, "is\_employee": false}'

data = json.loads(json\_string)

print(data)

# Десериализация из файла

with open("data.json", "r") as read\_file:

data = json.load(read\_file)

print(data)

Вопрос 24. Статический полиморфизм

По особенностям реализации полиморфизм можно разделить на ограниченный и неограниченный, динамический и статический.

Статический полиморфизм — это механизм, который позволяет одной и той же функции или методу иметь разные реализации в зависимости от типа передаваемых ей параметров.

Основная особенность статического полиморфизма заключается в том, что выбор нужного метода или функции происходит во время компиляции программы. Таким образом, компилятор определяет, какая версия метода или функции будет вызываться.

Статический полиморфизм достигается с помощью перегрузки функций (overloading) и операторов, а также с помощью наследования и полиморфизма на уровне классов и объектов.

**Пример статического полиморфизма:**

int add(int a, int b) {

return a + b;

}

float add(float a, float b) {

return a + b;

}

int main() {

int result1 = add(1, 2);

float result2 = add(1.5f, 2.5f);

}

**Принципы статического полиморфизма:**

* Оверлоадинг функций: возможность определить несколько функций с одним и тем же именем, но с разными типами аргументов.
* Шаблоны: возможность определить общий тип или шаблон для класса или функции, который может быть использован для работы с разными типами данных.

**Оверлоадинг функций**: В данном примере приведены две функции add с одним и тем же именем, но разными типами данных (int и float). В результате компилятор сможет автоматически выбрать подходящую функцию в зависимости от типов аргументов.

int add(int a, int b)

{

return a + b;

}

float add(float a, float b)

{

return a + b;

}

**Шаблоны:**

template <class T>

T multiply(T a, T b)

{

return a \* b;

}

Данный пример показывает использование шаблонной функции, которая может умножать значения любого типа данных. Компилятор автоматически создаст специализацию функции для каждого типа данных, который будет использоваться в вызовах функции multiply.

Статический полиморфизм играет важную роль в обеспечении гибкости и удобства при написании программ. Он позволяет программисту использовать одно и то же имя для различных функций или методов, что упрощает понимание кода и повышает его читабельность.

**Примеры статического полиморфизма:**

* Перегрузка функций: возможность определить несколько функций с одинаковым именем, но с разными наборами параметров. Компилятор будет автоматически выбирать нужную версию функции в зависимости от переданных аргументов.
* Перегрузка операторов: возможность определить различные варианты работы операторов, таких как сложение, вычитание, сравнение и т. д., для различных типов данных.
* Наследование и полиморфизм: возможность создания классов-потомков, которые могут использовать методы и свойства родительского класса, но при этом иметь собственную реализацию этих методов и свойств. Компилятор будет выбирать версию метода или свойства в зависимости от типа объекта.

Вопрос 25

**Типовые компоненты для обслуживания бд**

В системе баз данных выделяют четыре основных компонента:

* данные;
* аппаратное обеспечение;
* программное обеспечение;
* пользователи.

**Данные.**Различают 2 типа СУБД: однопользовательские и многопользовательские.

Основная задача многопользовательской системы обеспечить работу пользователю как в однопользовательской системе. Мы будем рассматривать данные только в многопользовательских системах. Данные в системе БД являются интегрированными и общими.

*Интегрированные данные* подразумевают возможность представлять БД как объединение нескольких файлов данных, полностью или частично не перекрывающихся.

*Общие данные* подразумевают возможность использования отдельных областей данных в БД несколькими отдельными пользователями отдельно.

Для упрощения мы будем предполагать, что все данные хранятся в одной БД (но возможно в нескольких файлах).

БД состоят из некоторого набора постоянных данных, которые используются прикладными программами.

Обычно данные,  хранящиеся в БД, называются *постоянными* (хотя они недолго могут оставаться такими). «Постоянные» - по отношению к другим данным: промежуточным, входным, выходным.

*Входные данные* – это информация, передаваемая системе (обычно с терминала или рабочей станции). Такая информация может стать причиной изменения постоянных данных.

*Выходные данные* – это сообщения и результаты, выдаваемые системой (обычно на печать или отображается на экране, возможно, записывается на диски). Ясно, что различие между видами данных нельзя назвать четкими, они определяются на интуитивном уровне. БД состоят из некоторого набора постоянных данных, которые используются прикладными программами.

На больших предприятиях в настоящее время все чаще используются два вида БД:

* операционная БД - для поддержания повседневной работы предприятия;
* база данных, содержащая отчетную информацию - данные для поддержания принятия решений по управлению предприятием. Эти  данные периодически обновляются (раз в день, раз в неделю и т.д.), получая информацию из оперативной БД.

**Аппаратное обеспечение:**

* накопители;
* сетевое оборудование;
* оперативная память
* процессор.

**Программное обеспечение:**

* СУБД;
* утилиты;
* средства разработки приложений (программы конечного пользователя);
* средства проектирования;
* генераторы счетов и др.

**Пользователи:**

* + *Прикладные программисты* – пользователи, которые отвечают за написания прикладных программ (приложений), использующих БД.
  + *Конечные пользователи* – пользователи, которые работают с базой  данных через рабочую станцию (терминал). Конечный пользователь получает доступ к БД через приложения или используя интегрированный интерфейс СУБД. Конечный пользователь часто использует интерфейс, основанный на меню и различных формах, что облегчает работу.
  + *Администраторы базы данных* организуют и отвечают за работу с БД.

Вопрос 26

**Диаграмма классов**

**UML** – унифицированный язык моделирования (Unified Modeling Language) – это система обозначений, которую можно применять для объектно-ориентированного анализа и проектирования.  
Словарь UML включает три вида строительных блоков:

* Диаграммы.
* Сущности.
* Связи.

**Сущности** – это абстракции, которые являются основными элементами модели, **связи** соединяют их между собой, а **диаграммы** группируют представляющие интерес наборы сущностей.

**Диаграмма** – это графическое представление набора элементов, чаще всего изображенного в виде связного графа вершин (сущностей) и путей (связей).

Диаграммы классов показывают набор классов, интерфейсов, а также их связи. Диаграммы этого вида чаще всего используются для моделирования объектно-ориентированных систем. Они предназначены для статического представления системы.

**Сущности**

Диаграммы классов оперируют тремя видами сущностей UML:

* Структурные.
* Поведенческие.
* Аннотирующие.

**Структурные сущности** – это «имена существительные» в модели UML. В основном, статические части модели, представляющие либо концептуальные, либо физические элементы. Основным видом структурной сущности в диаграммах классов является **класс**.  
**Поведенческие сущности** – динамические части моделей UML. Это «глаголы» моделей, представляющие поведение модели во времени и пространстве. Основной из них является **взаимодействие** – поведение, которое заключается в обмене сообщениями между наборами объектов или ролей в определенном контексте для достижения некоторой цели. Сообщение изображается в виде линии со стрелкой, почти всегда сопровождаемой именем операции.  
**Аннотирующие сущности** – это поясняющие части UML-моделей, иными словами, комментарии, которые можно применить для описания, выделения и пояснения любого элемента модели. Главная из аннотирующих сущностей – **примечание**. Это символ, служащий для описания ограничений и комментариев, относящихся к элементу либо набору элементов. 

**Структурные сущности - классы**

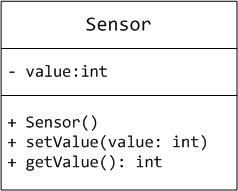
**Класс** – это описание набора объектов с одинаковыми атрибутами, операциями, связями и семантикой.  
  
Графически класс изображается в виде прямоугольника, разделенного на 3 блока горизонтальными линиями:

* имя класса
* атрибуты (свойства) класса
* операции (методы) класса.

Для атрибутов и операций может быть указан один из трех типов видимости:

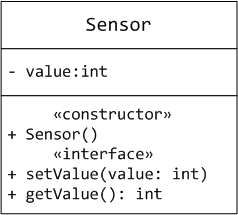
* **-** - private (частный)
* **#** - protected (защищенный)
* **+** - public (общий)

Каждый класс должен обладать именем, отличающим его от других классов.



**Атрибут** (свойство) – это именованное свойство класса, описывающее диапазон значений, которые может принимать экземпляр атрибута. Класс может иметь любое число атрибутов или не иметь ни одного. В последнем случае блок атрибутов оставляют пустым.  
Атрибут представляет некоторое свойство моделируемой сущности, которым обладают все объекты данного класса. Можно уточнить спецификацию атрибута, указав его тип, кратность (если атрибут представляет собой массив некоторых значений) и начальное значение по умолчанию.

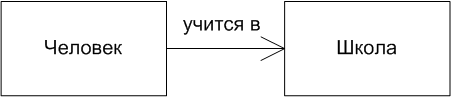
**Операция** (метод) – это реализация метода класса. Класс может иметь любое число операций либо не иметь ни одной. Часто вызов операции объекта изменяет его атрибуты.  
Графически операции представлены в нижнем блоке описания класса.  
Допускается указание только имен операций. Имя операции, как и имя класса, должно представлять собой текст. На практике для именования операции используются короткие глагольные конструкции, описывающие некое поведение класса, которому принадлежит операция. Обычно каждое слово в имени операции пишется с заглавной буквы, за исключением первого, например move (переместить) или isEmpty (проверка на пустоту).  
Можно специфицировать операцию, устанавливая ее сигнатуру, включающую имя, тип и значение по умолчанию всех параметров, а применительно к функциям – тип возвращаемого значения.  
  
Абстрактные методы класса обозначаются курсивным шрифтом.  
Статические методы класса обозначаются подчеркиванием.



**Отношения между классами**

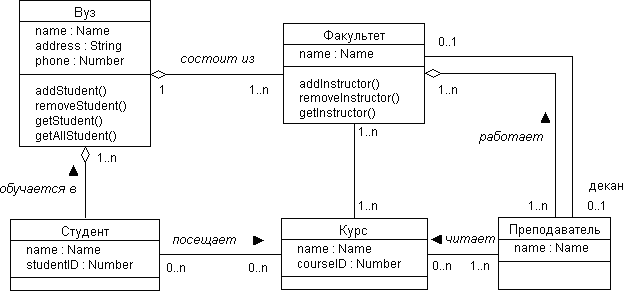
Существует четыре типа связей в UML:

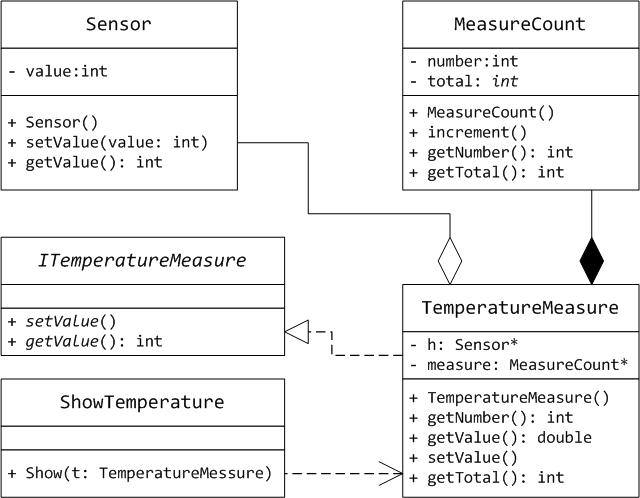
* Зависимость
* Ассоциация
* Обобщение
* Реализация

Первая из них – **зависимость** – семантически представляет собой связь между двумя элементами модели, в которой изменение одного элемента (независимого) может привести к изменению семантики другого элемента (зависимого). Графически представлена пунктирной линией, иногда со стрелкой, направленной к той сущности, от которой зависит еще одна; может быть снабжена меткой.  
   
Зависимость  
  
**Ассоциация** – это структурная связь между элементами модели, которая описывает набор связей, существующих между объектами.  
Ассоциация показывает, что объекты одной сущности (класса) связаны с объектами другой сущности таким образом, что можно перемещаться от объектов одного класса к другому.  
Например, класс **Человек** и класс **Школа** имеют ассоциацию, так как человек может учиться в школе. Ассоциации можно присвоить имя «учится в». В представлении однонаправленной ассоциации добавляется стрелка, указывающая на направление ассоциации.  


Двойные ассоциации представляются линией без стрелок на концах, соединяющей два классовых блока.  
Ассоциация может быть именованной, и тогда на концах представляющей её линии будут подписаны роли, принадлежности, индикаторы, мультипликаторы, видимости или другие свойства.  
  
**Множественность ассоциации** представляет собой диапазон целых чисел, указывающий возможное количество связанных объектов. Он записывается в виде выражения с минимальным и максимальным значением; для их разделения используются две точки. Устанавливая множественность дальнего конца ассоциации, вы указываете, сколько объектов может существовать на дальнем конце ассоциации для каждого объекта класса, находящегося на ближнем ее конце. Количество объектов должно находиться в пределах заданного диапазона. Множественность может быть определена как единица 1, ноль или один 0..1, любое значение 0..\* или \*, один или несколько 1..\*. Можно также задавать диапазон целых значений, например 2..5, или устанавливать точное число, например 3.  


**Агрегация** – особая разновидность ассоциации, представляющая структурную связь целого с его частями. Как тип ассоциации, агрегация может быть именованной. Одно отношение агрегации не может включать более двух классов (контейнер и содержимое).  
Агрегация встречается, когда один класс является коллекцией или контейнером других. Причём, по умолчанию агрегацией называют агрегацию по ссылке, то есть когда время существования содержащихся классов не зависит от времени существования содержащего их класса. Если контейнер будет уничтожен, то его содержимое — нет.  
Графически агрегация представляется пустым ромбом на блоке класса «целое», и линией, идущей от этого ромба к классу «часть».  
Агрегация  
**Композиция** — более строгий вариант агрегации. Известна также как агрегация по значению.  
Композиция – это форма агрегации с четко выраженными отношениями владения и совпадением времени жизни частей и целого. Композиция имеет жёсткую зависимость времени существования экземпляров класса контейнера и экземпляров содержащихся классов. Если контейнер будет уничтожен, то всё его содержимое будет также уничтожено.  
Графически представляется как и агрегация, но с закрашенным ромбиком.  
Композиция  
  
Третья связь – **обобщение** – выражает специализацию или **наследование**, в котором специализированный элемент (потомок) строится по спецификациям обобщенного элемента (родителя). Потомок разделяет структуру и поведение родителя. Графически обобщение представлено в виде сплошной линии с пустой стрелкой, указывающей на родителя.  
Обобщение  
Четвертая – **реализация** – это семантическая связь между классами, когда один из них (**поставщик**) определяет соглашение, которого второй (**клиент**) обязан придерживаться. Это связи между интерфейсами и классами, которые реализуют эти интерфейсы. Это, своего рода, отношение «целое-часть». Поставщик, как правило, представлен абстрактным классом. В графическом исполнении связь реализации – это гибрид связей обобщения и зависимости: треугольник указывает на поставщика, а второй конец пунктирной линии – на клиента.  
Реализация





На диаграмме классов основным классом является класс TemperatureMeasure, который и является измерителем температуры. В качестве измеренного значения формируется среднее арифметическое всех измерений - сумма всех измерений, деленная на их количество.  
Для получения измерений и их суммирования используется класс Sensor (в качестве датчика температуры). В консольной задаче сами измерения передаются в этот класс для суммирования. Класс состоит в отношении агрегации с основным классом TemperatureMeasure: мы сначала создаем объект класса Sensor, а потом передаем его в качестве параметра конструктора классу TemperatureMeasure, чтобы использовать его в качестве части класса.  
Количество измерений формируется классом MeasureCount, который содержит статическое свойство total для подсчета общего измерений, а также свойство count для подсчета количества измерителей конкретного объекта TemperatureMeasure. Класс MeasureCount находится в отношении композиции с классом TemperatureMeasure: объект MeasureCount создается непосредственно при создании объекта TemperatureMeasure (в его конструкторе).  
Класс ITemperatureMeasure представляет собой интерфейс класса TemperatureMeasure и является своего рода **поставщиком** в отношении реализации.  
Наконец, класс ShowTemperature находится в отношении зависимости с классом TemperatureMeasure, поскольку реализация единственного метода Show класса ShowTemperature зависит от структуры класса TemperatureMeasure.