Конспекты по программированию

2 семестр

ИКТ
2021 — 2022

Автор:
Даниил Швалов

Оглавление

1 Процесс создания ПО . . . 3
2 Жизненный цикл программного продукта . . . 3
3 Каскадная модель . . . 4
4 Спиральная модель . . . 4
5 Agile . . . 5
6 Kanban . . . 5
7 Scrum . . . 6
8 Основные элементы объектно-ориентированного подхода . . . 6
9 Объектно-ориентированное программирование . . . 7
10 Классы . . . 7
11 Инкапсуляция . . . 8
12 Отношение между классами . . . 8
13 Определение и перегрузка операторов . . . 10
14 Абстрактные классы . . . 10
15 Множественное наследование . . . 11
16 Работа с SQLite . . . 12
17 ORM . . . 13
18 Peewee . . . 13
19 SQLAlchemy . . . 15

1. Процесс создания ПО

Процесс создания ПО – совокупность мероприятий, целью которых является создание или модернизация ПО.

1. Анализ предметной области (постановка задачи)
2. Разработка проекта системы
   (a) Создание модели, отражающей основные функциональные требования, предъявляемые к программе
   (b) Выбор метода решения (построение мат. модели)
   (c) Разработка алгоритма – последовательности действий по решению задачи
3. Реализация программы на языке программирования (кодирование)
4. Анализ полученных результатов (тестирование)
5. Внедрение и сопровождение

Этап анализа состоит в исследовании системных требований и проблемы. Различают:

- анализ требований — исследование требований к системе;
- объектный анализ — исследование объектов предметной области.

2. Жизненный цикл программного продукта

Жизненный цикл - это непрерывный процесс, который начинается с момента принятия решения о необходимости его создания и заканчивается в момент его полного изъятия из эксплуатации. Он базируется на 3-х группах процессов:

1. **Основные процессы** — реализуются под управлением основных сторон (заказчик, поставщик, разработчик, оператор и персонал сопровождения), вовлеченных в жизненный цикл программных средств.
   Примеры: заказ, поставка, разработка, эксплуатация, сопровождение.
2. **Вспомогательные процессы** — обеспечивают выполнение основных процессов.
   Примеры: документирование, управление конфигурацией, обеспечение качества, верификация, аттестация, совместный анализ, аудит, решение проблем.
3. **Организационные процессы** — применяются для создания, реализации и постоянного совершенствования основной структуры, охватывающей взаимосвязанные процессы жизненного цикла и персонал.
   Примеры: управление, создание инфраструктуры, усовершенствование, обучение.

Стадии жизненного цикла: замысел $\longrightarrow$ разработка $\longrightarrow$ производство $\longrightarrow$ применение $\longrightarrow$ поддержка $\longrightarrow$ списание.

3. Каскадная модель

Этапы каскадной модели:

1. определение требований;
2. проектирование;
3. кодирование, тестирование модулей;
4. интеграция, тестирования;
5. эксплуатация, сопровождение.

Характеристика модели:

- фиксированный набор стадий;
- каждая стадия — законченный результат;
- стадия начинается, когда закончилась предыдущая.

Недостатком модели является ее «негибкость»:

- фаза должна быть закончена, прежде чем приступить к следующей;
- набор фаз фиксирован;
- тяжело реагировать на изменение требований.

Каскадную модель **рекомендуется использовать** там, где требования заранее известны и неизменны.

4. Спиральная модель

Спиральная модель — частный случай итерационного подхода:

- вместо действий с обратной связью — спираль;
- отсутствуют заранее фиксированные фазы;
- каждый виток спирали — 1 итерация;
- каждый виток разбит на 4 сектора:
  - определение целей;
  - оценка и разрешение рисков;
  - разработка и тестирование;
  - планирование.
- на каждом витке могут применяться разные модели процесса разработки ПО.

Главное отличие — акцент на анализ и преодоление рисков.

5. Agile

Agile — это семейство «гибких» подход к разработке ПО. Agile предполагает, что при реализации проекта

- не нужно опираться только на заранее созданные подробные планы;
- важно ориентироваться на постоянно меняющиеся условия внешней и внутренней среды;
- учитывать обратную связь от заказчиков и пользователей.

Частными случаями agile-подходов являются *scrum* и *kanban*.

6. Kanban

Особенности:

- kanban — это «подход баланса»;
- задача — сбалансировать разных специалистов внутри команды и избежать ситуации, когда дизайнеры работают сутками, а разработчики жалуются на отсутствие новых задач;
- вся команда едина — отсутствуют роли владельца продукта и scrum-мастера;
- бизнес-процесс делится не на универсальные спринты, а на стадии выполнения конкретных задач (планируется, разрабатывается, тестируется, завершено);
- для визуализации используются доски (физ. и эл.) — они позволяют сделать рабочий процесс открытым и понятным для всех специалистов.

Практики:

- визуализация потока задач;
- ограничение невыполненных работ;
- управление рабочим потоком;
- использование явных правил;
- введение петли обратной связи;
- улучшение и эволюция.

Kanban отлично подходит для небольших проектов, бизнес вебсайтов, где не требуется много времени на планирование. Также он хорошо подходит для долгосрочных проектов, где нет четкой спецификации где задания формируются в процессе разработки.

7. Scrum

Спринт — итерация (цикл выпуска продукта):

- имеет фиксированную длительность, обычно от 2 до 8 недель;
- результат — готовый продукт, который потенциально можно передать заказчику;
- в течении спринта делаются все работы по сбору требований, дизайну, кодированию;
- рамки спринта фиксированы;
- каждый спринт начинается с собрания по его планированию и заканчивается собранием, где подводятся итоги спринта.

Scrum подходит для крупного проекта (длительность от 3 месяцев), который имеет полную спецификацию и требования перед началом разработке. В таком случае команда может составить детальный план разработки и весь процесс поделить на спринты.

8. Основные элементы объектно-ориентированного подхода

Объектом можно назвать то, что имеет четкие границы и обладает состоянием и поведением. Причем состояние определяет поведение (незаряженное ружье не может выстрелить).

Этапы ОО-анализа:

1. выделить объекты;
2. определить их существенные свойства;
3. описать поведение (команды, которые они могут выполнять).

В процессе ОО-анализа основное внимание уделяется определению и описанию объектов в терминах предметной области.

Для ОО стиля концептуальная база — *объектная модель*.

Основополагающая идея — объединение данных и действий, производимых над данными, в единое целое, которое называется объектом.

Программа — множество объектов, каждый из которых обладает своими свойствами и поведением, но его внутреннее устройство скрыто от других объектов.

В процессе ОО-проектирования определяются программные объекты и способы их взаимодействия с целью выполнения системных требований.

GRAPS — General Responsibility Assignment Software Patterns (Общие шаблоны распределения обязанностей в программных системах):

- **Information expert** — класс, у которого имеется информация, требуемая для выполнения обязанности.
- **Creator** — класс, создающий экземпляры какого-то определенного класса.

- **High conhension** — распределение обязанностей, поддерживающее высокую степень зацепления.
- **Low coupling** — распределение обязанностей таким образом, чтобы степень связности оставалась низкой.
- **Controller** — делегирование обязанностей по обработке системных сообщений другому классу.

9. Объектно-ориентированное программирование

Объектно-ориентированное программирование – это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

Особенности:

- ООП использует в качестве базовых элементов объекты, а не алгоритмы;
- каждый объект является экземпляром какого-либо класса;
- классы организованы иерархически.

Достоинства:

- программа простая и понятная;
- классы могут разрабатывать разные программисты независимо друг от друга;
- повторное использование кода.

Недостатки:

- неэффективно для небольших задач.

10. Классы

Класс — это структура данных, объединяющая состояние (поля) и действия (методы и др. функции-члены). При определении класса создается новое пространство имен и используется в качестве локальной области видимости при выполнении инструкций в теле определения.

Атрибуты — это переменные, заключенные в класс. Все атрибуты хранятся в поле `__dict__`.

Конструктор — метод, который автоматически вызывается при создании экземпляра класса. В `python` роль конструктора играет метод `__init__(self)`. Конструктор гарантирует, что все необходимые поля класса будут проинциализированы правильно.

Деструктор — метод, который автоматически вызывается при удалении объекта. В `python` роль конструктора играет метод `__del__(self)`. Объект удаляется, если

- исчезают все связанные с ним переменные;
- им присваивается другое значение, в результате чего связь со старым объектом теряется.
- он удаляется с помощью `del`.

11. Инкапсуляция

Инкапсуляция — процесс отделения друг от друга элементов объекта, определяющих его устройство и поведение. Инкапсуляция выполняется посредством сокрытия внутренней информации, т. е. маскировкой всех внутренних деталей, не влияющих на внешнее поведение.

Инкапсуляция необходима для того, чтобы изолировать контрактные обязательства абстракции от их реализации.

Для реализации инкапсуляции можно использовать геттеры и сеттеры:

```python
class Person:
    def __init__(self, age):
        self.__age = age

    def get_age(self):
        return self.__age

    def set_age(self, age):
        assert age ⩾ 0
        self.__age = age


p = Person(12)
print(p.get_age()) # 12
p.set_age(15)
print(p.get_age()) # 15
```

или аннотации:

```python
class Person:
    def __init__(self, age):
        self.__age = age

    @property
    def age(self):
        return self.__age

    @age.setter
    def age(self, age):
        assert age ⩾ 0
        self.__age = age

p = Person(12)
print(p.age) # 12
p.age = 15
print(p.age) # 15
```

12. Отношение между классами

Виды отношений между классами:

- зависимость;

- обобщение (наследование);
- реализация;
- ассоциация (агрегация, композиция).

Иерархия — это упорядочение абстракций, расположение их по уровням. Основными видами иерархических структур являются

- структура классов (иерархия «is-a»), пример — наследование;
- структура объектов (иерархия «part of»);

Наследование — это такое отношение между классами (родитель/потомок), что один класс заимствует структурную и функциональную часть одного или нескольких других классов (одиночное/множественное наследование).

Роль наследования:

- общая часть структуры и поведения сосредоточена в наиболее общем суперклассе;
- суперклассы отражают наиболее общие, а подклассы — более специализированные абстракции, в которых члены суперкласса могут быть дополнены, модифицированы или скрыты.
- принцип наследования позволяет:
  - упростить выражение абстракций;
  - делает проект менее громоздким;
  - более выразительным.

Зависимость — однонаправленное отношение использования между двумя классами: один класс зависимый, а второй — независимый. Зависимость — **не структурная** связь. Объект зависимого класса для своего корректного функционирования пользуется услугами объекта-сервера независимого класса.

Зависимость отражает связь между объектами по применению, когда изменение поведения сервера может повлиять на поведение клиента.

Модель включения/делегации реализует отношение имеет («has-a», агрегация). Это отношение позволяет одному классу определять переменную-член другого класса и опосредованно представлять его функциональность пользователю объекта. Эта форма использования не применяется для отношения «parent-child».

Агрегация — направленное отношение между классами, предназначенное для представления ситуации, когда один из классов — сущность, состоящая из других сущностей (пример: ПК состоит из процессора, видеокарты, мат. платы). Агрегация является частным случаем ассоциации.

Композиция — более сильная форма отношения «часть-целое», при которой с удалением объекта класса-контейнера удаляются и все объекты, являющиеся его составными частями (пример: окно программы, состоящее из заголовка, полосы прокрутки, рабочей области). Композиция является частным случаем ассоциации.

13. Определение и перегрузка операторов

Методы, имена которых обрамлены двумя нижними подчеркиваниями, python трактует как **специальные** (например __init__ или __str__). Рекомендуется определять спец. методы раньше, чем обычные методы.

Перегрузка операторов позволяет объектам участвовать в операциях, свойственным встроенными типам (сложение, вычитание и т. п.). Перегрузка реализуется с помощью определения специальных методов. Например, для реализации операции сложения объект должен реализовать метод __add__:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

p1 = Point(1, 5)
p2 = Point(3, 6)
p3 = p1 + p2
print(p3.x, p3.y) # 4 11
```

14. Абстрактные классы

Абстрактный класс — класс, содержащий хотя бы один абстрактный метод (метод, который был объявлен, но не был определен, т.е. у метода отсутствует тело).

Особенности:

- не могут быть инстанцированы (нельзя создать экземпляр класса);
- для создания экземпляра требуется унаследовать свой класс, реализовать все абстрактные методы и после этого создать экземпляр.

В python это можно реализовать следующим образом:

```
class Shape:
    def square(self):
        raise NotImplementedError()


class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def square(self):
        return self.width * self.height


s = Shape()
r = Rectangle(5, 10)
```

```
# print(s.square()) # error
print(r.square()) # 50
```

У такой реализации есть недостаток — мы все еще можем создать экземпляр абстрактного класса. Это можно исправить с помощью библиотеки `abc`. Кроме того, абстрактный метод можно определить и вызывать его в дочерних классах:

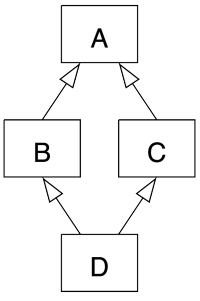
```
import abc


class Shape(abc.ABC):
    @abc.abstractmethod
    def square(self):
        return 0  # можно заменить на pass


class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def square(self):
        return self.width * self.height


class EmptyShape(Shape):
    def square(self):
        return super().square()


# s = Shape() # error
r = Rectangle(5, 10)
es = EmptyShape()
print(r.square())  # 50
print(es.square()) # 0
```

15. Множественное наследование

Множественное наследование — это возможность класса потомка наследовать функционал от нескольких родителей. Использование множественного наследования позволяет создавать **классы-примеси** (mixin).

Для определения, является класс потомком другого класса, используется функция `issubclass(Child, Parent)`. Для того, чтобы проверить, является ли объект экземпляром класса, используется функция `isinstance(object, Class)`.

Проблема алмаза (diamond problem) — неопределенность, возникающая, когда два класса B и C наследуют от A, а класс D наследует от обоих классов B и C. Если метод класса D вызывает метод, определенный в классе A (и этот метод не был переопределен), а классы B и C по-своему переопределили этот метод, то тогда возникает вопрос — от какого класса его наследовать: B или C?



В `python`, в таком случае, используется алгоритм поиска в ширину. То есть, если D сначала наследуется от B, а потом от C, тогда список поиска метода в классах будет B, C, A.

16. Работа с SQLite

SQLite — это библиотека, реализующая легковесную дисковую БД, не требующую отдельного серверного процесса и позволяющую получить доступ к БД с использованием языка запросов SQL.

Python имеет встроенную поддержку SQLite: достаточно импортировать модуль `sqlite3`.

Основные операции при работе с БД:

1. загрузка библиотеки;
2. создание и соединение с БД;
3. создание таблицы БД;
4. добавление данных;
5. запросы на получение данных;
6. обновление данных;
7. удаление данных.

Пример работы с SQLite:

```
import sqlite3


conn = sqlite3.connect("animals.db")  # БД сохраняется на диск
# или
conn = sqlite3.connect(":memory:")  # БД создается в ОЗУ

cursor = conn.cursor()  # курсор требуется для выполнения SQL запросов

cursor.execute("<SQL-query>")  # выполнение <SQL-query>

cursor.executescript(
    """
<SQL-query-1>;
<SQL-query-2>;
...
<SQL-query-n>;
"""
)  # выполнение нескольких запросов

data = [(0, 5), (1, 3), (2, 3)]
cursor.executemany(
    "INSERT INTO product VALUES(?, ?)", data
)  # вставка нескольких строк в БД
```

```
price = 5
cursor.execute(
    "SELECT * FROM product WHERE price ⩽ ?", price
)  # price подставится на место знака вопроса


cursor.execute(
    "SELECT * FROM product WHERE price < :price", {"price": price}
)  # price подставится на место :price

query = cursor.execute("SELECT * FROM product")
all_products = query.fetchall()  # получить все товары
one_product = query.fetchone()  # получить только один
five_products = query.fetchmany(5)  # получить 5 товаров

conn.commit()  # зафиксировать изменения
```

17. ORM

ORM (Object Relational Mapping) — метод программирования для преобразования данных между несовместимыми системами типов в ОО языках программирования. ORM построен поверх языка SQL.

Основной целью API ORM является облегчение связывания пользовательских классов Python с таблицами БД и наоборот. Так, например, изменения в состояниях объектов и строк синхронно сопоставляются друг с другом, поскольку запросы к БД выражаются в терминах пользовательских классов. Таким образом каждый класс отображается на таблицу в БД.

Основные подходы к реализации ORM:

- **Active record** — отображение объекта данных на строку БД.
- **Data mapper** — полное отделение представления данных в программе от представления в БД.

ORM библиотеки:

- **peewee** — легкая, быстрая и гибка ORM, которая поддерживает SQLite, MySQL и PostgreSQL. В библиотеке применяется подход **Active record**.
- **SQLAlchemy** — поддерживает SQLite, MySQL, PostgreSQL. В библиотеке применяется подход **Data mapper**.

18. Peewee

peewee предназначен для работы с реализацией DBAPI, созданной для конкретной базы данных.

Пример работы с SQLite:

```
import peewee as pw
```

```python
conn = pw.SqliteDatabase("product.db")  # создание соединения с БД
cursor = conn.cursor()  # курсор для выполнения запросов


# базовая модель (это, так сказать, база)
class BaseModel(pw.Model):
    class Meta:
        database = conn


# описание модели (таблицы) товара, поля — столбцы таблицы
class Product(BaseModel):
    # можно изменять название полей в БД
    id = pw.AutoField(column_name="product_id")
    name = pw.TextField(column_name="product_name")

    class Meta:
        table_name = "product"


# создаем саму таблицу
Product.create_table()

# различные способы создания записи в БД
Product(name="Book").save()
Product.create(name="Table")
Product.insert({"name": "Apple"}).execute()
Product.insert_many([{"name": "Pen"}, {"name": "Pineapple"}]).execute()

# выбор всех товаров
products = Product.select().execute()
for p in products:
    print(p.name)  # Book, Table, Apple, Pen, Pineapple
print("---")

# выбрать первые 3
products = Product.select().limit(3).execute()
for p in products:
    print(p.name)  # Book, Table, Apple
print("---")

# выбрать товар с ID = 1
p = Product.select().where(Product.id == 1).first()
print(p.name)  # Book
print("---")

# SQL запрос
cursor.execute("SELECT * FROM product WHERE product_id ⩾ 3")
products = cursor.fetchall()
for p in products:
    print(p[1])  # Apple, Pen, Pineapple
print("---")


# изменение товара
Product.update(name="Audio").where(Product.id == 1).execute()
p = Product.select().where(Product.id == 1).first()
print(p.name)  # Audio
```

```
# удаление товара
Product.delete().where(Product.id == 1).execute()
p = Product.select().where(Product.id == 1).first()
print(p)  # None

conn.close()  # закрыть соединение с БД
```

---

19. SQLAlchemy

SQLAlchemy предназначен для работы с реализацией DBAPI, созданной для конкретной БД.

Особенности:

- использует систему диалектов для связи с различными типами реализаций DBAPI и БД;
- все диалекты требуют, чтобы был установлен соответствующий драйвер DBAPI;
- некоторые диалекты включены по умолчанию: FireBird, MS SQL Server, MySQL, Oracle, PostgreSQL, SQLite, Sybase.

Пример работы с SQLAlchemy(Core):

```
import sqlalchemy as sa
from sqlalchemy.ext.declarative import declarative_base

# создание объекта Engine БД
engine = sa.create_engine("sqlite:///product.db")

# создаем соединение с БД
conn = engine.connect()

# определения таблиц и связанных объектов (индекс, представление, триггеры)
# хранятся в метаданных. MetaData — это коллекция объектов Table и связанных
# с ними конструкции схем, которая также содержит привязку к Engine или
# Connection
meta = sa.MetaData()
meta.bind = engine

# Table представляют собой соответствующую таблицу в БД
product = sa.Table(
    "product",
    meta,
    sa.Column(
        "product_id",
        sa.Integer,
        nullable=False,
        unique=True,
        primary_key=True,
        autoincrement=True,
    ),
    sa.Column("product_name", sa.String),
)
```

```
# создание таблицы в БД
meta.create_all(engine)

# добавляем одно значение
conn.execute(sa.insert(product).values(product_name="Book"))

# добавляем несколько значений
conn.execute(
    sa.insert(product),
    [
        {"product_name": "Table"},
        {"product_name": "Apple"},
        {"product_name": "Pineapple"},
        {"product_name": "Pen"},
    ],
)

# выбрать все строки
p_list = conn.execute(sa.select(product))
for p in p_list:
    print(p[1])  # Book, Table, Apple, Pineapple, Pen
print("---")


# выборка с условием
p_list = conn.execute(sa.select(product).where(product.c.product_id ⩾ 3))
for p in p_list:
    print(p[1])  # Apple, Pineapple, Pen
print("---")

# выбрать первый элемент
p = conn.execute(sa.select(product).where(product.c.product_id == 3)).first()
print(p[1])  # Apple
print("---")

# изменение строки по ID
conn.execute(
    sa.update(product).where(product.c.product_id == 3).values(product_name="Notebook")
)
p = conn.execute(sa.select(product).where(product.c.product_id == 3)).first()
print(p[1])  # Notebook
print("---")

# удаление строки по ID
conn.execute(sa.delete(product).where(product.c.product_id == 3))
p = conn.execute(sa.select(product).where(product.c.product_id == 3)).first()
print(p)  # None
print("---")
```

Пример работы с SQLAlchemy(ORM):

```
import sqlalchemy as sa
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative_base

# создание объекта Engine БД
engine = sa.create_engine("sqlite:///product.db")
```

```python
# создаем сессию
Session = sessionmaker(bind=engine)

session = Session()

# определения таблиц и связанных объектов (индекс, представление, триггеры)
# хранятся в метаданных. MetaData — это коллекция объектов Table и связанных
# с ними конструкции схем, которая также содержит привязку к Engine или
# Connection
# meta = sa.MetaData()
Base = declarative_base()


# объявляем таблицу
class Product(Base):
    __tablename__ = "product"

    id = sa.Column(
        "product_id",
        sa.Integer,
        nullable=False,
        unique=True,
        primary_key=True,
        autoincrement=True,
    )
    name = sa.Column("product_name", sa.String)


# создаем все таблицы в БД
Base.metadata.create_all(engine)

# добавляем новый объект
session.add(Product(name="Apple"))
# добавляем сразу несколько
session.add_all(
    [
        Product(name="Book"),
        Product(name="Pen"),
        Product(name="Pineapple"),
        Product(name="Table"),
    ]
)
# фиксируем изменения
session.commit()

# получить все строки из БД
for p in session.query(Product).all():
    print(p.name)  # Apple, Book, Pen, Pineapple, Table
print("---")

# получить первый
p = session.query(Product).where(Product.id ⩾ 3).first()
print(p.name)  # Pen
print("---")

# получить по ID
p = session.query(Product).get(3)
print(p.name)  # Pen
print("---")
```

```
# изменение строки в БД
session.query(Product).where(Product.id == 3).update(
    {Product.name: "Notebook"},
    synchronize_session=False,
)
session.commit()

p = session.query(Product).where(Product.id == 3).first()
print(p.name)  # Notebook
print("---")

# удаление строки в БД
session.query(Product).where(Product.id == 3).delete()
session.commit()

p = session.query(Product).where(Product.id == 3).first()
print(p)  # None
print("---")
```
