Интеллектуальные информационные системы

Лекция 5. Некоторые современные аспекты разработки в информационных системах

Кафедра информатики
Институт кибербезопасности и цифровых технологий
РТУ МИРЭА

SOLID

Принцип **SOLID** — это набор из пяти основных принципов объектно-ориентированного проектирования, которые помогают создавать модульный, расширяемый и поддерживаемый код. Эти принципы были предложены Робертом Мартином ("дядя Боб") и широко применяются в разработке для повышения качества программного обеспечения.

1. S — Single Responsibility Principle (Принцип единственной ответственности):

- **Суть:** У каждого класса должна быть только одна причина для изменения.
- **Пояснение:** Класс должен решать только одну задачу или иметь единственную область ответственности.
- **Проблема без SRP:** Если класс выполняет несколько задач, то изменение одной функциональности может затронуть другие, нарушая стабильность.

2. O — Open/Closed Principle (Принцип открытости/закрытости):

- **Суть:** Классы должны быть открыты для расширения, но закрыты для модификации.
- **Пояснение:** Новая функциональность должна добавляться без изменения существующего кода, чтобы избежать регрессий.
- **Проблема без OCP:** Изменение существующего кода для добавления новых функций может вносить баги.

3. L — Liskov Substitution Principle (Принцип подстановки Барбары Лисков):

- **Суть:** Объекты должны быть заменяемы их подтипами без нарушения работы программы.
- **Пояснение:** Наследуемые классы должны полностью соответствовать контракту базового класса, чтобы можно было использовать их вместо родительских без изменений.
- **Проблема без LSP:** Если подкласс не ведёт себя как базовый класс, то полиморфизм нарушается.

SOLID

4. I — Interface Segregation Principle (Принцип разделения интерфейсов):

- **Суть:** Интерфейсы должны быть узкоспециализированными и соответствовать конкретным задачам.
- **Пояснение:** Не заставляйте классы реализовывать методы, которые им не нужны.
- **Проблема без ISP:** Если интерфейс слишком широк, то классы вынуждены реализовывать ненужные методы, что нарушает их ответственность.

5. D — Dependency Inversion Principle (Принцип инверсии зависимостей):

- **Суть:** Модули верхнего уровня не должны зависеть от модулей нижнего уровня. Оба должны зависеть от абстракций.
- **Пояснение:** Реализация должна зависеть от абстракций, а не наоборот. Это делает код гибче и позволяет подменять зависимости.
- **Проблема без DIP:** Верхние уровни кода зависят от конкретных реализаций, что затрудняет модификацию и тестирование.

Реализация DIP посредством внедрение зависимостей

Внедрение зависимостей (Dependency Injection, DI) — это программный подход, при котором объект получает (или ему "внедряются") свои зависимости извне, а не создаёт их самостоятельно. Это позволяет разделить ответственность между компонентами системы, сделать код более модульным, гибким и лёгким для тестирования.

DI — это реализация **DIP**, позволяющая управлять зависимостями более гибко.

Основные концепции внедрения зависимостей

1. Зависимость:

- Зависимость — это любой объект или ресурс, который необходим другому объекту для выполнения своей работы. Например, сессия базы данных, настройки приложения, сервис для отправки уведомлений.

2. Внедрение:

- Внедрение означает, что зависимость передается объекту извне, обычно через параметры конструктора, методы или свойства объекта.

3. Контейнер зависимости:

- Это механизм, который управляет созданием, передачей и завершением работы зависимостей. В FastAPI, например, **Depends** выступает в роли простого контейнера.

Реализация DIP посредством внедрение зависимостей

- **Пример внедрения зависимостей**
- **Без внедрения зависимостей:**

```python
class Service:
    def __init__(self):
        self.database = DatabaseConnection()  # Зависимость создаётся внутри класса

    def do_something(self):
        self.database.query("SELECT * FROM table")
```

В данном случае класс **Service** сам создаёт объект **DatabaseConnection**. Это плохо, так как:

- **Service** жёстко связан с конкретной реализацией базы данных.
- Тестировать класс сложно, так как нет возможности подменить **DatabaseConnection**.

С внедрением зависимостей:

```python
class Service:
    def __init__(self, database):
        self.database = database  # Зависимость передаётся извне

    def do_something(self):
        self.database.query("SELECT * FROM table")
```

Теперь:

- **Service** не отвечает за создание объекта базы данных.
- При необходимости можно передать другую реализацию базы данных (например, тестовую).

Реализация DIP посредством внедрение зависимостей

- **Внедрение зависимостей в FastAPI**

1. Вы определяете функцию или объект, который выступает в роли зависимости.
2. Указываете зависимость в эндпоинте с помощью **Depends**.
3. FastAPI автоматически вызывает функцию зависимости, обрабатывает её результат и передает его в ваш эндпоинт.

Пример:

```python
from fastapi import FastAPI, Depends

app = FastAPI()

def get_config():
    return {"database_url": "sqlite:///:memory:"}

@app.get("/")
def read_root(config: dict = Depends(get_config)):
    return {"message": "Database URL is", "url": config["database_url"]}
```

Функция get_config — это зависимость.
Depends(get_config) говорит **FastAPI** передать результат функции **get_config** в параметр **config** эндпоинта.

Реализация DIP посредством внедрение зависимостей

Преимущества внедрения зависимостей

1. Модульность:
- Компоненты приложения меньше зависят друг от друга, что упрощает замену и расширение.

2. Тестируемость:
- Легко подменять зависимости в тестах (например, использовать тестовую базу вместо реальной).

3. Повторное использование:
- Зависимости можно использовать в разных частях приложения без необходимости их дублирования.

4. Управление жизненным циклом:
- FastAPI автоматически создаёт и завершает зависимости, например, закрывает соединения с базой данных.

Пример использования в тестах

Чтобы подменить сессию базы данных в тестах, можно использовать **pytest**:

```python
from fastapi.testclient import TestClient
from app.main import app
from app.db.session import SessionLocal, get_db

def override_get_db():
    db = SessionLocal()
    try:
        yield db  # Используем ту же логику, но для тестовой БД
    finally:
        db.close()

app.dependency_overrides[get_db] = override_get_db  # Подменяем зависимость

client = TestClient(app)

def test_create_book():
    response = client.post("/books/", json={"title": "Dune", "author": "Frank Herbert"})
    assert response.status_code == 200
```

Метод Depends

- Часто используемый способ внедрения сессии БД в метод:

```python
@router.post("/")
def create_book(book: BookCreate, db: Session = Depends(get_db)):
    # Здесь db — это активная сессия SQLAlchemy, предоставленная get_db
    return BookService.create_book(db, book)
```

Функция передающаяся в Depends структурно выглядит так:

```python
def get_db():
    db = SessionLocal()  # Создаём сессию базы данных
    try:
        yield db  # Передаём объект сессии вызывающему коду
    finally:
        db.close()  # Гарантированно закрываем сессию
```

До yield:

- Выполняется код, который создаёт ресурс (например, сессию базы данных).

yield db:

- Передает объект (db) вызывающему коду, например, эндпоинту, который использует сессию для выполнения операций.

После завершения использования:

- Как только эндпоинт завершает обработку запроса, FastAPI автоматически завершает работу генератора, выполняя код после **yield** (в данном случае **db.close()**).

Метод Depends

- Код **yield db** используется для создания **генератора**, который предоставляет объект **db** (в данном случае сессия базы данных) вызывающему коду, а затем выполняет завершающие действия, когда генератор закрывается.

Использование **yield** позволяет:

1. Управлять жизненным циклом зависимостей:

- Например, вы создаёте соединение с базой данных, передаёте его эндпоинту, а затем закрываете соединение.

2. Гарантировать очистку:

- Код в finally выполняется всегда, даже если запрос завершился с ошибкой.

3. Избежать утечек ресурсов:

- Закрытие сессий, освобождение памяти или других ресурсов происходит автоматически.

Упрощенная аналогия работы yield

```
def generator():
    print("Начало работы")
    yield "Ресурс"
    print("Очистка")

gen = generator()

# Получаем значение из генератора
print(next(gen))  # Вывод: "Начало работы", затем "Ресурс"

# Завершаем генератор
print(next(gen))  # Вывод: "Очистка"
```