Dependency Injection в генераторе случайного заказа

Введение

В этом проекте реализован генератор случайных заказов из меню различных служб доставки с демонстрацией паттерна **Dependency Injection (DI)**. Основная идея: отделить логику получения данных от логики их обработки, что делает код гибким и легко расширяемым.

Архитектура проекта

Основные компоненты

Проект разделен на несколько слоев согласно принципам чистой архитектуры:

1. **Domain Layer** (`domain.py`)

```python
class Dish:
    def __init__(self, name: str, price: str):
        self.__name = name
        self.__price = price

    def get_price_value(self) -> float:
        """Парсинг цены для математических операций"""
```

Класс `Dish` — шаблон блюда:

```python
class HtmlProvider(ABC):
    @abstractmethod
    def get_html(self, source: str) -> str:
        pass
```

Интерфейс `HtmlProvider` — контракт для поставщиков HTML:

Базовый класс `Delivery` — шаблон для всех реализаций доставок:

- Принимает **`HtmlProvider`** через конструктор (DI)
- Хранит меню в виде словаря **`{id: Dish}`**
- Определяет абстрактный метод **`_parse_html()`** для конкретных реализаций

2. **Infrastructure Layer** (`infrastructure.py`)
   Две реализации `HtmlProvider`:

```python
class LocalFileProvider(HtmlProvider):
    def get_html(self, source: str) -> str:
        with open(source, 'r', encoding='utf-8') as f:
            return f.read()
```

`LocalFileProvider` — чтение из локальных файлов:

```python
class HttpProvider(HtmlProvider):
    def get_html(self, source: str) -> str:
        response = requests.get(source, headers=self.headers)
        response.raise_for_status()
        response.encoding = response.apparent_encoding
        return response.text
```

`HttpProvider` — загрузка через HTTP:

3. **Delivery Implementation** (`delivery_impl.py`)

   Класс `SushiFast` — конкретная реализация парсера:

   - Наследуется от `Delivery`
   - Переопределяет `load_menu()` для обработки пагинации
   - Использует `self.provider.get_html()` для всех запросов
   - Парсит HTML с помощью BeautifulSoup

   Особенность: автоматически обрабатывает пагинацию меню, загружая все страницы последовательно.

Как работает Dependency Injection

Принцип работы

```
# Создаём провайдера
provider = HttpProvider()

# Внедряем его в доставку
delivery = SushiFast(provider)

# Загружаем меню
delivery.load_menu("https://sushifast.ru/menu")
```

Ключевой момент: `SushiFast` не знает, откуда приходит HTML — из файла или из интернета. Он просто вызывает `self.provider.get_html()` и получает результат.

Преимущества подхода

- **Гибкость:** легко переключаться между источниками данных
- **Тестируемость:** можно создать MockProvider для тестов
- **Разделение ответственности:** каждый класс выполняет одну задачу
- **Расширяемость:** легко добавить новые доставки

Обработка пагинации

```
def load_menu(self, source: str) -> None:
    # 1. Загружаем первую страницу через провайдер
    html_content = self.provider.get_html(source)
    soup = BeautifulSoup(html_content, "lxml")

    # 2. Парсим первую страницу
    self._parse_page(soup)

    # 3. Ищем пагинацию
    pagination = soup.find("nav", class_="woocommerce-pagination")
    if pagination:
        # 4. Определяем количество страниц
        max_page = self._find_max_page(pagination)

        # 5. Загружаем остальные страницы через провайдер
        for page_num in range(2, max_page + 1):
            page_url = f"{base_url}/page/{page_num}/"
            page_html = self.provider.get_html(page_url)  # DI!
            self._parse_page(BeautifulSoup(page_html, "lxml"))
```

Важно: Все HTTP-запросы идут `через self.provider`, сохраняя чистоту архитектуры.

Генерация заказов

Математический алгоритм (`random_math.py`)

```python
def convert_signed(x, k):
    """
        Основная функция генерации заказа.
        x - seed (ключ), k - количество блюд в меню.
    """
    n = x + 1000000001
    if n < 1 or n > 2000000001:
        raise ValueError("x must be in [-1000000000, 1000000000]")
    return convert(n, k)
```

Проект использует детерминированную генерацию заказов на основе seed-ключей

Алгоритм использует биномиальные коэффициенты для генерации уникальных комбинаций блюд, что позволяет:

- Воспроизводить одинаковые заказы по одному ключу
- Генерировать разнообразные комбинации
- Контролировать случайность через seed

Фильтрация по бюджету

```python
cash = 1588  # Бюджет в рублях

for id in order:
    total += menu[id].get_price_value()

if total > cash:
    print(f"Заказ отфильтрован: сумма {total:.2f}₽ превышает {cash}₽")
    continue
```

Демонстрация работы

```
provider = LocalFileProvider()
delivery = SushiFast(provider)
delivery.load_menu("Меню — SushiFast.html")
```

Локальный режим

```
provider = HttpProvider()
delivery = SushiFast(provider)
delivery.load_menu("https://sushifast.ru/menu")
```

Продакшн режим

Класс `SushiFast` **остается неизменным!** Меняется только одна строка при создании провайдера.

```
C:\Users\Matthew\PycharmProjects\Parse\.venv\Scripts\python.exe C:\Users\Matthew\PycharmProjects\Parse\main.py
[SushiFast] Запрос данных из: Меню — SushiFast.html
[SushiFast] Меню успешно загружено (176 позиций).

--- ДОСТУПНОЕ МЕНЮ ---
1 — Филадельфия (599₽)
2 — Филадельфия мини (449₽)
3 — Калифорния (449₽)
4 — Филадельфия в угре (609₽)
5 — Филадельфия Шик (575₽)
6 — Аляска (525₽)
7 — Кани Эби (499₽)
8 — Орандж гребешок (519₽)
9 — Филадельфия Прайм (619₽)
10 — Цезарь темпура (459₽)
11 — Эби Кранч (469₽)
12 — Юми (505₽)
13 — Сливочный лосось (449₽)
14 — Калифорния с лососем (525₽)
```

Рис. 1 Шапка вывода при локальной работе

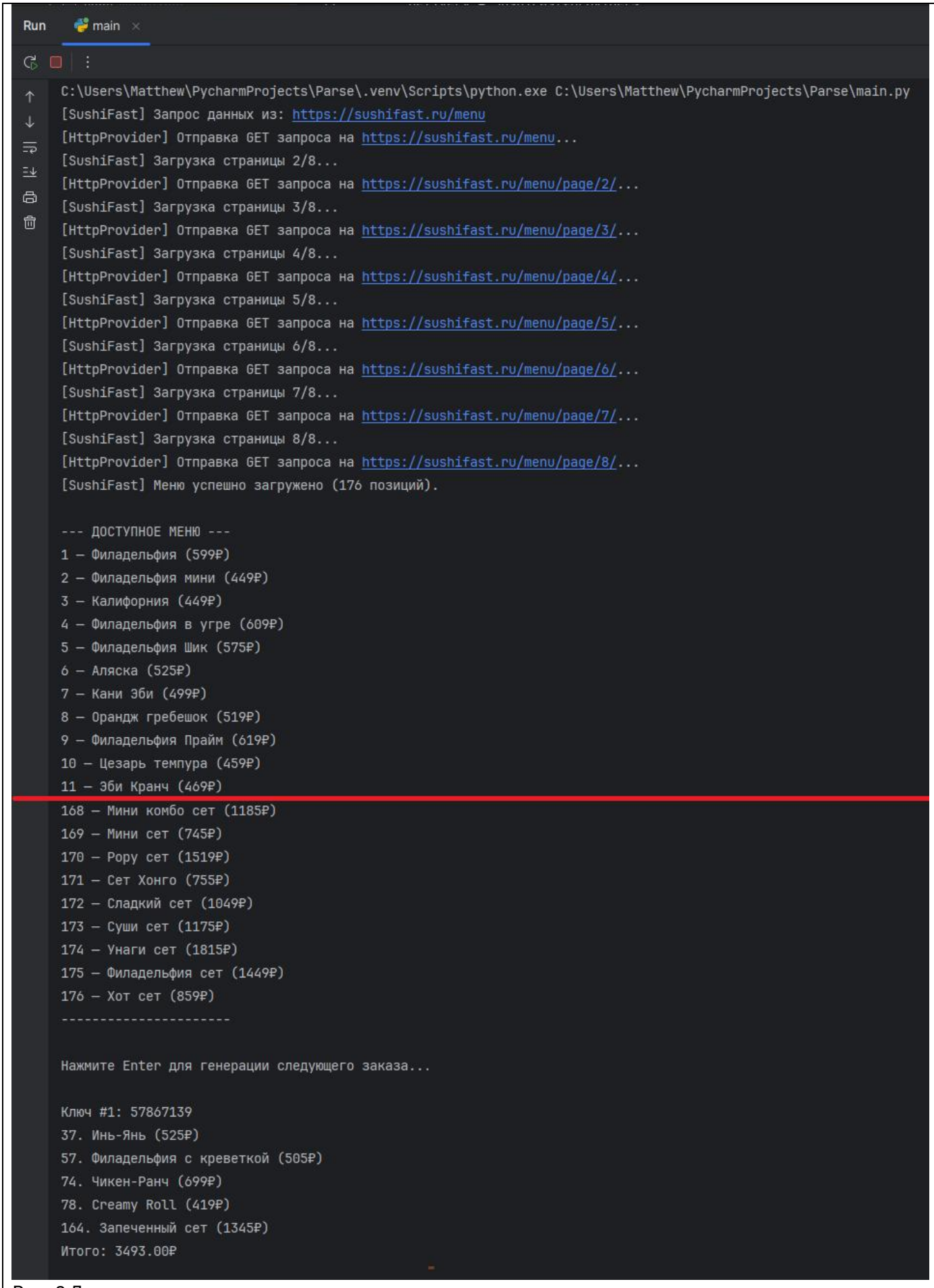

Рис. 2 Демонстрация вывода

Возможности расширения

1. Новые провайдеры

```python
class MockProvider(HtmlProvider):
    def __init__(self, fake_html: str):
        self.html = fake_html

    def get_html(self, source: str) -> str:
        return self.html
```

MockProvider для юнит-тестов

2. Новые службы доставки

```python
class BurgerKing(Delivery):
    def __init__(self, provider: HtmlProvider):
        super().__init__(provider)

    def _parse_html(self, html_content: str) -> None:
        # Логика парсинга конкретного сайта
        soup = BeautifulSoup(html_content, "lxml")
        # ... парсинг специфичных классов ВК
```

3. Telegram-бот

```python
@bot.message_handler(commands=['order'])
def generate_order(message):
    seed = random.randint(-1000000000, 1000000000)
    order = convert_signed(seed, len(menu))
    # Отправить заказ пользователю
```

4. Параллельная загрузка пагинации для оптимизации производительности

```python
import asyncio

async def load_all_pages(urls):
    tasks = [provider.get_html(url) for url in urls]
    return await asyncio.gather(*tasks)
```

Заключение

Проект демонстрирует практическое применение Dependency Injection для создания гибкой и расширяемой архитектуры. Основные достижения:

- ✓ Четкое разделение ответственности между компонентами
- ✓ Легкое переключение между источниками данных
- ✓ Простота добавления новых служб доставки
- ✓ Готовность к расширению функциональности
- ✓ Тестируемость кода через mock-объекты

Паттерн DI позволил создать систему, которая легко адаптируется к новым требованиям без изменения существующего кода — что и является целью качественной архитектуры программного обеспечения.

Стек технологий: Python, BeautifulSoup4, requests, ABC (Abstract Base Classes)

Паттерны: Dependency Injection, Template Method, Strategy