Санкт-Петербургский национальный исследовательский университет ИТМО

Факультет программной инженерии и компьютерной техники

Направление подготовки 09.03.04 Программная инженерия

Дисциплина «Архитектура программных систем»

Лабораторная работа №2

Студент

Бобрусь Александр Владимирович

Группа P33091

Преподаватель

Перл Иван Андреевич

Санкт-Петербург, 2023 г

Содержание

Задание **3**

Ход работы **3**

Декоратор (GoF) **3**

Сценарий #1: 3

Сценарий #2: 4

Сценарий #3: 4

Фасад (GoF) **5**

Сценарий #1: 5

Сценарий #2: 5

Информационный эксперт (GRASP) **6**

Сценарий #1: 6

Сценарий #2: 6

Сценарий #3: 6

Устойчивый к изменениям (GRASP) **7**

Сценарий #1: 7

Сценарий #2: 7

Сценарий #3: 7

Вывод **8**

Задание

Из списка шаблонов проектирования GoF и GRASP выбрать 3-4 шаблона и для каждого из них придумать 2-3 сценария, для решения которых могу применены выбранные шаблоны.

Сделать предположение о возможных ограничениях, к которым можем привести использование шаблона в каждом описанном случае. Обязательно выбрать шаблоны из обоих списков.

Ход работы

Декоратор (GoF)

Паттерн, позволяющий динамически расширять функционал объекта. Удобен, когда объектам нужно придавать различные конфигурации.

Сценарий #1:
Декоратор для хот-дога. Разные люди любят разные начинки, нельзя сделать что-то универсальное. Поэтому в данном случае удобно использовать паттерн декоратор: есть общая основа (булка с сосиской), а дальше каждый сам добавляет то, что хочет. Кто-то может посыпать карамелизированным луком, кто-то добавить горчицу или кетчуп и т.д. – можно использовать разные комбинации.

Пример кода:

```
public class Program {
    public static void main(String[] args) {
        HotDog defaultHotDog1 = new HotDog();
        HotDog defaultHotDog2 = new HotDog();

        //В первый хот-дог добавляем кетчуп и лук
        Decorator ketchupHotDog = new KetchupDecorator(defaultHotDog1);
        Decorator onionKetchupHotDog = new OnionDecorator(ketchupHotDog);

        //Во второй хот-дог добавляем горчицу
        Decorator mustardHotDog = new MustardDecorator(defaultHotDog2);
    }
}
```

Сценарий #2:
Декоратор уведомлений в приложении. Базовый функционал – всплывающие уведомления в телефоне. Затем расширяем функционал – добавляем возможность оповещения по почте, либо по телефону.

Пример кода:

```
public class Program {
    public static void main(String[] args) {
        NotificationManager notificationManager = new NotificationManager();

        //Добавляем функционал для оповещения по почте
        Decorator mailNotificationManager = new mailDecorator(notificationManager);

        //Добавляем функционал для оповещения по телефону
        Decorator phoneMailNotificationManager =
                new PhoneNotificationManager(mailNotificationManager);
    }
}
```

Сценарий #3:
Декоратор логирования. Базовый логгер просто выводит действия пользователя. Можно расширить функционал выводом времени. Затем добавить вывод даты.

Пример кода:

```
public class Program {
    public static void main(String[] args) {
        Logger logger = new Logger();

        //Добавляем вывод времени
        Decorator timeLogger = new TimeLogger(logger);

        //Добавляем вывод даты
        Decorator dateTimeLogger = new DateLogger(timeLogger);
    }
}
```

Фасад (GoF)

Паттерн, позволяющий упростить взаимодействие со сложной системой (включающей с себя много компонентов), объединяя все в одно целое (образно говоря, компоненты при этом жестко не связываются между собой).

Сценарий #1:

Цель – поесть. Без паттерна фасад человек бы покупал еду, готовил ее и только потом мог бы поесть. Паттерн фасад можно представить как кафе - посетитель не знает, что происходит на кухне – какие поставщики привозят продукты, как именно готовятся блюда, все процессы скрыты. Он просто делает заказ и получает готовое блюдо.

Пример кода:

Кафе:

```
public class CafeFacade {
    public Food newOrder(String dish) {
        Product[] products = findProducts(dish);
        products = wash(products);
        products = prepareProducts(products);

        return cook(products);
    }

    Product[] findProducts(String dish) {
        //найти необходимые продукты
    }

    Product[] wash(Products[] products) {
        //помыть продукты
    }

    Product[] prepareProducts(Product[] products) {
        //почистить и порезать
    }

    Food cook(Product[] products) {
        //приготовить еду
    }
}
```

Использование фасада:

```
public class Program {
    public static void main(String[] args) {
        Client client = new Client();
        CafeFacade cafe = new CafeFacade();

        client.eat(cafe.newOrder("Пицца с грибами"));
    }
}
```

Сценарий #2:

Если обычный пользователь хочет работать на компьютере – ему необязательно знать как устроен кэш в процессоре, как происходит разметка виртуальной памяти, нет надобности писать вручную процессорные команды. Ему достаточно пользоваться фасадом – в данном случае экраном, клавиатурой и мышкой. Все, что происходит внутри компьютера – совершенно не нужно пользователю для его использования.

Информационный эксперт (GRASP)

Ответственность дается объекту, который обладает необходимой информацией.

Сценарий #1:
Разработчик использует API банка для подключения оплаты на сайте. В данном случае информационный эксперт – банк. Разработчику не нужно самому заниматься обработкой платежей, переводом со счета на счет, проверкой транзакций и т.д. Он просто использует api банка – готовый инструментарий, реализующий “под капотом” огромный и сложный функционал, который разработчику сайта писать с нуля не имеет никакого смысла.

Сценарий #2:
Сеть супермаркетов обращается к аутсорс-компании для разработки программного обеспечения по учету продукции. В данном случае информационный эксперт – аутсорс-компания, к ней обращаются с задачей разработки ПО и получают готовый результат. Супермаркет не вдается в технические подробности – не знает какой стек технологий используется для разработки, какие методологии применяются и т.д., всем этим занимается аутсорс-компания, которая на вход принимает лишь требования, а на выходе выдает готовое программное обеспечение.

Сценарий #3:
Мы не сильны в математике, но нам очень нужно реализовать модель глубокого обучения для распознавания жестов в нашем приложении. В таком случае мы можем использовать готовые библиотеки, в которых уже реализована математическая часть и вся нужная логика. Нам остается только передать данные для обучения модели. В данном случае эксперт – библиотека для глубокого обучения.

Устойчивый к изменениям (GRASP)

Цель паттерна – сделать систему масштабируемой и модифицируемой. Чтобы этого достичь, точки неустойчивости (части, который с большой вероятностью придется изменять/модифицировать) определяются заранее и представляются в качестве интерфейса

Сценарий #1:
Например, мы разрабатываем программное обеспечение для риэлторов. В таком случае точкой неустойчивости может быть часть, отвечающая за парсинг недвижимости. Мы можем заточить программу под парсинг сайта циан и собирать с него нужную информацию об объектах недвижимости. Однако, если заказчик потребует организовать парсинг не только с циана, но еще и с авито, а также яндекс.недвижимости, это будет проблемой. Поэтому необходимо заранее определить общий интерфейс Parser и имплементировать для каждого из нужных сайтов.

Сценарий #2:
Мы разрабатываем инвестиционную платформу, в которой можно будет торговать ценными бумагами. В данном случае точка неустойчивости – ценные бумаги. Если мы изначально построим архитектуру нашей платформы таким образом, чтобы можно было торговать только акциями, то в дальнейшем потребуется больше усилий, чтобы добавить также поддержку торговли облигациями. А добавить торговлю паевыми фондами будет еще сложнее. Поэтому необходимо сразу определить ценные бумаги как интерфейс и в дальнейшем имплементировать конкретные реализации.

Сценарий #3:
Необходимо разработать сервис для антиплагиата текста. Точкой неустойчивости может оказаться класс TextExtractor, задача которого - считывать текст из файла. Если мы заточим нашу систему под работу с файлами расширения .txt, то в дальнейшем будет сложнее добавить поддержку файлов других расширений (например, pdf, docx и др.). В таком случае нам лучше изначально определить TextExtractor как интерфейс и имплементировать его реализации для конкретных классов (.txt, .pdf, .docx и др.).

Вывод

В ходе данной работы я закрепил свои знания паттернов GoF, а также узнал о паттернах GRASP и их применении. Можно сделать вывод, что использование паттернов – хорошая практика, т.к. это лучшие варианты решения конкретных классов задач. Однако, не стоит переусердствовать и применять их там, где в этом нет нужды, так как это лишь усложнит разрабатываемую систему.