ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«Национальный исследовательский университет ИТМО»
Факультет Программной инженерии и компьютерной техники

Лабораторная работа №2
«Использование шаблонов проектирования»
по дисциплине
«Архитектура программных систем»

Выполнил студент группы P3315:
Барсуков Максим Андреевич

Преподаватель:
Перл Иван Андреевич

г. Санкт-Петербург
2024

СОДЕРЖАНИЕ

1. 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.

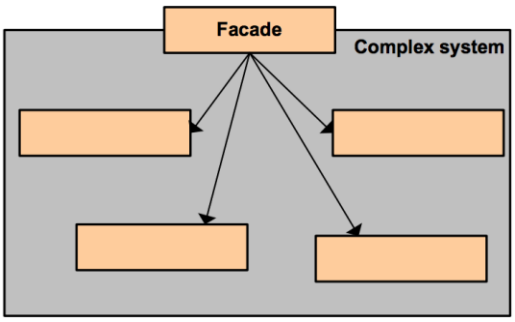
Сделать предположение о возможных ограничениях, к которым можем привести использование шаблона в каждом описанном случае. Обязательно выбрать шаблоны из обоих списков.

2. Facade (GoF)

2.1. Описание

Facade (Фасад) – **структурный** паттерн. Предоставляет унифицированный интерфейс к множеству интерфейсов в некоторой подсистеме. Определяет интерфейс более высокого уровня, облегчающий работу с подсистемой.

Разбиение на подсистемы облегчает проектирование сложной системы в целом. Общая цель всякого проектирования – свести к минимуму зависимость подсистем друг от друга и обмен информацией между ними. Один из способов решения этой задачи – введение объекта фасад, предоставляющий единый упрощенный интерфейс к более сложным системным средствам.



2.2. Сценарии использования

2.2.1. Упрощение взаимодействия с подсистемами в электронной коммерции

В интернет-магазине используется несколько подсистем для обработки заказов, управления пользователями, расчетов доставки и платежей. Для клиентов магазина необходимо предоставить упрощенный интерфейс для взаимодействия с этими подсистемами.

Ограничения:

- Скрытие важных функций: Некоторые из сложных функций, таких как индивидуальная настройка доставки или гибкие системы скидок, могут быть недоступны для клиентов через фасад.
- Малое расширение: Добавление новых возможностей в одну из подсистем может потребовать значительных изменений в фасаде, если он не предусмотрен для таких расширений.

2.2.2. Интерфейс для доступа к различным сервисам в облаке

Фасад может использоваться для упрощения доступа к различным облачным сервисам, таким как хранение данных, вычислительные ресурсы и средства мониторинга. Разработчики взаимодействуют с этим интерфейсом вместо того, чтобы напрямую работать с API облачных сервисов.

Ограничения:

- Узкое место: Если фасад управляет всеми операциями с облаком, он может стать узким местом в системе, особенно при больших нагрузках.
- Зависимость от фасада: Обновление облачных сервисов может потребовать изменений в фасаде, что потребует переработки всех компонентов, которые с ним взаимодействуют.

2.2.3. Упрощение взаимодействия с системой управления контентом (CMS)

Фасад может скрывать сложные API для управления контентом, например, создание, редактирование и публикация материалов на сайте. Вместо работы с многочисленными классами API, редакторы контента могут использовать простой интерфейс фасада.

Ограничения:

- Утрата гибкости: При добавлении новых функций в CMS, фасад может не поддерживать их, если изменения не были заранее учтены.
- Зависимость от подсистемы: Если CMS сильно изменяется, фасад может стать сложно поддерживаемым.

2.2.4. Интеграция с внешними API в корпоративной системе

Фасад может быть использован для упрощения взаимодействия с внешними API, такими как платежные шлюзы, службы поддержки и системы отчетности. Это позволяет скрыть детали реализации и предоставить унифицированный интерфейс для работы с различными сторонними сервисами.

Ограничения:

- Обновление API: При изменении внешних API необходимо обновлять фасад, что может быть трудоемким и требовать тестирования.
- Проблемы с масштабируемостью: При значительном увеличении количества сторонних API фасад может стать слишком сложным для управления.

2.3. Общие ограничения Facade

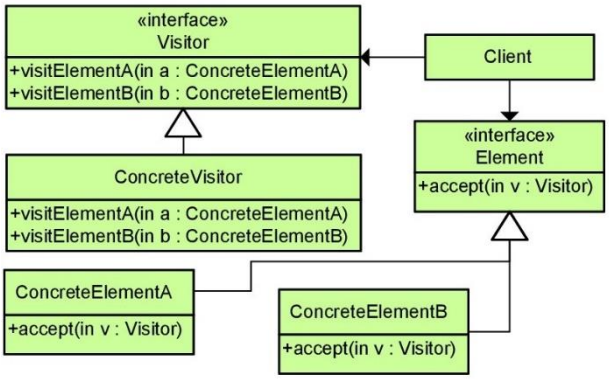
1. **Утрата гибкости**: Фасад упрощает доступ к сложным подсистемам, создавая единый интерфейс для работы с ними. Однако эта упрощенная абстракция может скрывать продвинутые возможности отдельных подсистем. Например, если одна из подсистем меняется и добавляет новые функции, клиентский код, использующий фасад, может быть не в курсе этих изменений, так как он оперирует только через базовый интерфейс фасада.
2. **Узкое место**: Если фасад используется для реализации множества операций, которые раньше выполнялись разными модулями, это может привести к тому, что фасад станет узким местом. В этом случае, если фасад не оптимизирован, система может замедлиться, особенно при высоких нагрузках. Пример: В сложной системе мониторинга, если фасад собрал все метрики из разных подсистем и сам их обрабатывает, это может снизить производительность, особенно при большом количестве запросов.
3. **Зависимость от фасада**: Если все части системы используют фасад для взаимодействия с подсистемами, изменения в фасаде потребуют пересмотра всех клиентских классов, что увеличивает сложность поддержания системы.
4. **Поддержка изменений**: Подсистемы, с которыми работает фасад, могут часто изменяться. Когда фасад стоит между клиентом и подсистемами, поддержка актуальности фасада становится сложной задачей, особенно если подсистемы развиваются или изменяются. Пример: В старой системе документооборота фасад может скрывать взаимодействие с устаревшей версией системы, что затрудняет миграцию на новую систему с улучшенными возможностями.

3. Visitor (GoF)

3.1. Описание

Visitor (Посетитель) – **поведенческий** паттерн. Представляет операцию, которую надо выполнить над элементами объекта. Позволяет определить новую операцию, не меняя классы элементов, к которым он применяется. Посетитель отделяет алгоритмы от структуры объектов, что облегчает расширение функциональности системы.

Применяя паттерн посетитель, вы определяете две иерархии классов: одну для элементов, над которыми выполняется операция (иерархия Node), а другую – для посетителей, описывающих те операции, которые выполняются над элементами (иерархия NodeVisitor). Новая операция создается путем добавления подкласса в иерархию классов посетителей. До тех пор, пока грамматика языка остается постоянной (то есть не добавляются новые подклассы Node), новую функциональность можно получить путем определения новых подклассов NodeVisitor.



3.2. Сценарии использования

3.2.1. Интерпретация синтаксического дерева в языке программирования

При разработке интерпретатора для языка программирования или DSL код сначала преобразуется в абстрактное синтаксическое дерево (AST), где каждый узел соответствует конструкции языка (например, числу, оператору, функции или условию). Посетитель используется для обработки различных узлов дерева: вычисления выражений, выполнения команд, обработки циклов и вызовов функций. Это позволяет отделить логику интерпретации от структуры узлов.

Ограничения:

- Зависимость от структуры дерева: Если добавляются новые типы узлов (например, новые операторы или конструкции языка), необходимо модифицировать существующих посетителей, что увеличивает стоимость их поддержки.
- Сложность расширения: Добавление новых операций к узлам, которые зависят от большого количества типов, усложняет проектирование новых посетителей.
- Трудности тестирования: Каждый посетитель должен быть протестирован на всех типах узлов, что требует дополнительных усилий для создания тестового покрытия.

3.2.2. Обработка различных типов документов в редакторе

В графическом редакторе пользователи могут работать с различными типами объектов (текст, изображения, графики). Посетитель используется для добавления новых операций обработки, таких как сохранение, экспорт, печать, для каждого типа объекта без изменения их классов.

Ограничения:

- Зависимость от структуры данных: Если добавляются новые типы объектов (например, новые формы документов), необходимо модифицировать все существующие посетители.
- Сложность в тестировании: Повышенная сложность тестирования операций, так как каждый посетитель должен быть протестирован для всех типов объектов.

3.2.3. Анализ отчетов по различным меткам в системе

В аналитической платформе используется посетитель для проведения различных видов анализа данных на основе меток (например, время, категория, регион). Это позволяет добавлять новые способы анализа (среднее, минимальное, максимальное значение) без изменения существующих классов отчетов.

Ограничения:

- Перегрузка системы: Если в системе много различных типов отчетов и аналитических операций, это приведет к увеличению числа посетителей, что усложнит систему.
- Неэффективность для малых систем: Для небольших систем использование посетителя может привести к избыточной сложности.

3.3. Общие ограничения Visitor

1. **Трудность добавления новых структур данных**: Шаблон посетителя упрощает добавление нового поведения к объектам, но для этого необходимо изменять класс посетителя. Однако если структура данных часто меняется, то добавление нового типа объекта потребует модификации всех посетителей, что делает шаблон сложным для использования в динамично изменяющихся системах. Пример: В графическом редакторе, если структура объектов изменяется (например, добавляются новые типы фигур), каждый посетитель должен быть обновлен, что ведет к высокому уровню зависимости между посетителями и структурами данных.
2. **Сложность понимания**: Шаблон посетителя добавляет дополнительный слой абстракции, что делает систему более сложной для понимания и сопровождения. Многие программисты могут не сразу понять, как использовать или изменять посетителей и объекты, особенно в сложных иерархиях. Пример: В приложении для работы с отчетами, где каждое изменение в данных приводит к вызову различных посетителей для генерации отчетов, сложно поддерживать понимание всей системы без глубокого знания структуры.
3. **Связанность с конкретной структурой**: Посетитель сильно зависит от структуры данных, и любое изменение в этой структуре потребует модификации всех посетителей. Это делает его малопригодным для систем, где структура данных часто меняется. Пример: В системе обработки документов, где данные представлены в виде дерева, любое изменение в структуре дерева потребует изменения логики посетителей, что приведет к затратам на поддержку.

4. Controller (GRASP)

4.1. Описание

Обязанности по обработке входящих системных сообщений необходимо делегировать специальному объекту Controller'y. Controller — это объект, который отвечает за обработку системных событий, и при этом не относится к интерфейсу пользователя. Controller определяет методы для выполнения системных операций. Контролер отвечает за обработку запросов и решает, кому их делегировать на выполнение. Контроллер отделяет логику обработки от представления, обеспечивая таким образом лучшую модульность и расширяемость системы.

4.2. Сценарии использования

4.2.1. Обработка запросов от пользователей в веб-приложении

В веб-приложении для обработки запросов от пользователей создается контроллер, который маршрутизирует запросы, управляет состоянием сессий, выполняет бизнес-логику и взаимодействует с базой данных.

Ограничения:

- Сложность тестирования: Контроллер будет сильно зависеть от других компонентов, таких как база данных, что усложнит тестирование.
- Перегрузка контроллера: Контроллер может стать перегруженным, если будет выполнять слишком много задач, таких как обработка бизнес-логики и аутентификация.

4.2.2. Управление потоками заказов в электронной коммерции

Контроллер обрабатывает различные этапы заказа, начиная от выбора товара и заканчивая оплатой. Он выполняет валидацию данных, взаимодействует с системой складов и платежной системой.

Ограничения:

- Избыточная сложность: Контроллер может стать слишком сложным, если в нем будут реализованы все этапы заказа, включая бизнес-логику и обработку ошибок.
- Нарушение принципа единой ответственности: Контроллер может оказаться перегруженным, если ему придется управлять всеми аспектами процесса заказа, включая логику платежей, уведомлений и доставки.

4.2.3. Организация работы с данными в мобильном приложении

Контроллер управляет взаимодействием с локальной базой данных, обновлением UI, и обработкой событий от пользователя (например, создание новой записи, редактирование данных).

Ограничения:

- Проблемы с тестируемостью: Тестирование контроллеров в мобильных приложениях может быть сложным из-за плотной связи с UI и данными.
- Отсутствие гибкости: Если необходимо изменить логику взаимодействия с базой данных, это потребует изменений в контроллере.

4.3. Общие ограничения Controller

- **Концентрация обязанностей**: Контроллеры часто становятся точками перегрузки, где концентрируются слишком много обязанностей, что приводит к их усложнению и затрудняет поддержку. Когда контроллер выполняет множество операций (например, управление бизнес-логикой, обработка событий и взаимодействие с данными), его сложно тестировать и изменять. Пример: В интернет-магазине контроллер может одновременно обрабатывать запросы пользователей, контролировать аутентификацию, управлять корзиной и выполнять проверки заказов, что приводит к увеличению сложности кода.
- **Трудности тестирования**: Высокая связность контроллера с другими компонентами системы делает его сложным для тестирования. Часто для тестирования контроллера требуется настройка множества других объектов или компонентов, что увеличивает стоимость тестирования. Пример: Тестирование контроллера в веб-приложении, который управляет входом пользователя, может потребовать создания моков для базы данных и других сервисов.
- **Изменение влияет на всю систему**: Контроллеры часто напрямую взаимодействуют с множеством других компонентов системы. Если контроллер меняется, это может повлиять на большое количество классов и функциональности, что делает систему менее устойчивой к изменениям. Пример: В банковской системе, если контроллер изменяется, чтобы поддерживать новый тип транзакции, это может затронуть все операции и процессы, связанные с переводами.
- **Риск создания "божественного" объекта**: Контроллер может стать "божественным объектом" (*Класс, который хотел быть всем сразу. У него была и бизнес-логика, и данные, и контроллеры, и, кажется, даже немного души*), который берет на себя больше обязанностей, чем должен. Это нарушает принцип единой ответственности SRP и делает код сложным для поддержки. Пример: В системе управления проектами контроллер может начать обрабатывать не только запросы пользователей, но и логику аутентификации, валидацию данных и генерацию отчетов.
- **Избыточная сложность**: Если логика слишком сильно сосредоточена в контроллере, это может привести к чрезмерной сложности кода. Контроллер может стать перегруженным, если в нем не будет должным образом распределена ответственность. Пример: В крупном корпоративном приложении, контроллер может начать выполнять задачи, которые должны быть делегированы другим компонентам, например, вычисления и обработка данных.

5. Pure Fabrication (GRASP)

5.1. Описание

Необходимо обеспечивать low coupling и high cohesion. Для этой цели может понадобиться синтезировать искуственную сущность. Паттерн Pure Fabrication говорит о том, что не стоит стесняться это сделать. В качестве примера можно рассматривать фасад к базе данных. Это чисто искуственный объект, не имеющий аналогов в предметной области. В общем случае любой фасад относится к Pure Fabrication (если это конечно не архитектурный фасад в соответствующим приложении). Эти объекты не несут самостоятельной бизнес-логики, но обеспечивают необходимую структуру или функциональность для решения проблем, не нарушая принципы SOLID.

5.2. Сценарии использования

5.2.1. Моделирование работы с кредитами в банке

Создается объект, который абстрагирует работу с кредитными историями и начислением процентов, представляя собой отдельный модуль для расчета кредитов, упрощая взаимодействие с другими компонентами системы, такими как база данных или интерфейс пользователя.

Ограничения:

- Избыточная абстракция: Усложнение архитектуры, если логика работы с кредитами достаточно проста и могла бы быть реализована в рамках существующих объектов, например, клиента или транзакции.

5.2.2. Логика маршрутов доставки в логистической системе

Создается объект для определения оптимальных маршрутов доставки товаров, который объединяет данные о складах, клиентах и транспортных средствах. Этот объект позволяет рассчитывать маршруты на основе различных параметров, таких как расстояние, стоимость и время.

Ограничения:

- Избыточность: Если маршруты могли бы быть рассчитаны с помощью уже существующего модуля логистики, объект выдумки станет ненужным.
- Сложность интеграции: Логика маршрутов может потребовать интеграции с несколькими системами (например, геолокации и аналитики), что увеличит сложность объекта.
- Сложность расширения: Если добавляются новые параметры для расчета маршрутов, объект может потребовать значительной доработки.

5.3. Общие ограничения Pure Fabrication

1. **Избыточность**: Создание дополнительных объектов, которые представляют несуществующие концепты (чистая выдумка), может привести к усложнению системы без реальной необходимости. Такие объекты могут дублировать логику, уже реализованную в других классах. Пример: В системе управления персоналом создание объекта для вычисления бонусов, если такая логика уже реализована в классе сотрудника, может быть избыточным решением.
2. **Нарушение интуитивности**: Использование чистой выдумки может создать абстракции, которые будут неочевидны для других разработчиков. Это нарушает принцип понятности кода и может создать путаницу в понимании системы. Пример: В системе обучения сотрудников создание отдельного объекта для логики рекомендаций по обучению может сбивать с толку, если такие рекомендации могли быть частью другого класса.
3. **Риск дублирования**: Неудачное проектирование "чистой выдумки" может привести к повторению уже существующей логики. Это нарушает принципы DRY (Don’t Repeat Yourself) и усложняет сопровождение. Пример: В CRM-системе создание отдельного объекта для расчета вероятности сделки может дублировать логику, которая уже реализована в модуле аналитики.
4. **Перегрузка архитектуры**: Избыточное использование чистой выдумки может привести к раздроблению системы на слишком большое количество объектов, что затруднит понимание ее структуры. Пример: В системе управления проектами может быть создан отдельный объект "Временная метка задач", хотя такая функциональность могла бы быть частью класса задачи.

6. ВЫВОД

В ходе выполнения лабораторной работы были рассмотрены и проанализированы четыре паттерна проектирования: Facade, Visitor, Controller, Pure Fabrication. Для каждого из них были изучены особенности, предложены примеры сценариев использования, а также выявлены возможные ограничения. Каждый из рассмотренных паттернов проектирования имеет свои сильные стороны и ограничения. Их использование требует тщательного анализа архитектуры системы, чтобы выбрать подходящий инструмент для конкретной задачи. Основная задача при выборе паттерна — обеспечить баланс между простотой, гибкостью и поддерживаемостью системы. Применение паттернов должно основываться на конкретных потребностях проекта и учитывать возможные последствия для дальнейшей разработки и сопровождения системы. Важно помнить, что паттерны инструменты для решения определенного круга архитектурных задач, и их чрезмерное или неуместное использование может привести к over-engineering’у системы.