МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №5

по дисциплине «Объектно-ориентированное программирование»

Тема: Паттерны

Студент гр. 7304 _____ Нгуен К.Х.

Преподаватель _____ Размочаева Н.В.

Санкт-Петербург

2019

Цель работы.

Исследование паттернов проектирования и их реализация на языке программирования C ++.

Задание

Вариант 2. Обработчик последовательности.

Реализовать обработчик последовательности объектов стандартного (int, string, double, char) типа. Обработчик представляет собой ориентированный параллельно-последовательный граф с одним входом и выходом. Каждая вершина графа является функциональным элементом, выполняющим какое-то действие над объектом.

Функциональные элементы могут быть различаться по количеству входов и выходов: 1 вход – 1 выход; 1 вход – 2 выхода; 2 входа – 1 выход: 2 входа – выхода. Вход обработчика может быть элементом только с 1 входов, а выход обработчика может быть элементом только с 1 выходом. Причем, функциональные элементы с 2 входами выполняются только когда объекты поступили на оба входа. Например, могут быть следующие функциональные элементы: умножение числа на другое число n; определение более длинной строки; расчет целой части и остатка от деления числа на число n.

Такт – одна итерация работы всего обработчика. За один такт выполняется каждый функциональный элемент, передает объект дальше и принимает следующий. Таким образом, работа обработчика происходит по принципе конвейера.

При запуске программы, набор обработчиков заранее известен, но должна быть возможность построения обработчика. Также должна быть возможность замены одного функционального элемента на другой в ходе работы обработчики, причем состояние должно сохраняться.

Также необходимо предусмотреть возможность отката на один такт назад.

Экспериментальные результаты.

1. Использование как минимум один паттерна каждого типа для решения задачи (порождающие, структурные, поведенческие). (подробности ниже)
2. Сохранение логов хода работы программы в файл. Реализовал 3 вида сохранения

- Без сохранения логов
- Моментальное сохранение (информацию о событии сразу записывается в файл)
- Сохранение с кэшированием (информация сохраняется в кэш, при заполнении кэша происходит запись в файл и очистка кэша)

3. UML-диаграммы классов.
4. Реализация своего класса исключений их обработка.

Опциональные пункты:

- Сохранение и загрузка состояния программы в файл. Реализовал для 2 расширений файлов (txt, бинарный файл).
- Реализация GUI. При создании GUI необходимо разделять логику интерфейса и бизнес-логику.

Минимальные требования к заданию:

- По 2 типа обработчика для каждой комбинации входа/выхода
- Обработчик должен работать с двумя стандартными типами (int и double)
- Обработчик должен работать с одним пользовательским типом (Coordinate)

Дополнительное задание:

- Замена одного функционального элемента набором функциональных элементов

- Реализация обработчика с несколькими входами и выходами. Добавлялся возможность обработки разных последовательностей одного типа
- Функциональный элементы могут преобразовывать тип объектов.

Паттерны проектирования, используемые в лабораторной работе

Порождающие

Одиночка (Singleton)

Одиночка — это порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

В этом проекте он используется в классе LogWriter, обеспечивая простой способ доступа и записи логов со всей программы. Поскольку программа записи может открыть файл журнала на диске, несколько экземпляров программы записи могут вызвать конфликт доступа к файлу. Таким образом, решение здесь заключается в использовании Singleton.

Чтобы реализовать класс LogWriter как синглтон, нам нужно добавить в него поле типа LogWriter * (указатель на объект этого класса) и функцию getInstance (). В этой функции мы проверим значение указателя, определенного выше, если оно равно nullptr (undefined), мы создадим новый объект и назначим его; если он уже определен, мы возвращаем его значение. Нам также нужно сделать конструктор частным, чтобы предотвратить создание объекта класса вне класса (это означает, что единственный способ создания объекта этого класса - это вызов функции getInstance ()).

Прототип (Prototype)

Прототип — это порождающий паттерн проектирования, который позволяет копировать объекты, не вдаваясь в подробности их реализации.

В этой работе я реализовал общий репозиторий прототипов как синглтон, так что я могу создать обработчик, просто вызвав getInstance (), затем getByName (string s) c s - это имя типа обработчика.

Для этого сначала нужно создать функцию Clone для обработчиков (чисто виртуальная функция в классе Unit, реализованная в каждом из производных обработчиков). После этого я создаю новый класс HandlerRegistry. В этом классе я создал карту типа <string, Unit *> для хранения имени и прототипа обработчиков. В конструкторе я инициировал значение этой карты. Я также создал функцию getByName. Эта функция принимает имя в качестве параметра, пытается найти его на карте, если прототип найден, мы возвращаем его клон.

Структурные

Компоновщик (Composite)

Компоновщик — это структурный паттерн проектирования, который позволяет сгруппировать множество объектов в древовидную структуру, а затем работать с ней так, как будто это единичный объект.

Используя этот шаблон, я могу создавать более сложные обработчики, используя ряд основных обработчиков

Чтобы реализовать этот шаблон, класс Model наследуется от класса BaseHandler, поэтому модель также может функционировать как элементарный обработчик. В классе Model я создал массив для хранения указателей на обработчики под его управлением, указателя на первый и указателя на последний обработчик (в этой модели). Я переопределяю функцию handle () в Model, чтобы когда Модель получала параметры для обработки, она передавала эти параметры первому обработчику под своим управлением. Когда все обработчики под его управлением завершат обработку, он возьмет результат и передаст его следующему в цепочке.

Заместитель (Proxy)

Заместитель — это структурный паттерн проектирования, который позволяет подставлять вместо реальных объектов специальные объекты-

заменители. Эти объекты перехватывают вызовы к оригинальному объекту, позволяя сделать что-то до или после передачи вызова оригиналу.

В этом проекте он используется для обеспечения возможности кэширования для писателя (TxtWriter). BufferedTxtWriter является заменой TxtWriter, он содержит в себе фактический TxtWriter, но вызывает функцию записи на диск только после заполнения буфера. Это помогает уменьшить количество вызовов функции записи, что занимает много времени.

Чтобы реализовать этот шаблон, сначала я создал интерфейс Writer с чисто виртуальной функцией записи. Затем создается класс TxtWriter и класс BufferedTxtWriter. Эти 2 класса реализованы интерфейсом Writer. Помимо реализации функции Write, в классе BufferedTxtWriter есть указатель на Writer (Writer *) и строковое действие в качестве буфера. Реализация функции "write (string message)" в TxtWriter записывает в файл на диске. При реализации в BufferedTxtWriter записывают в строку буфера. Когда этот буфер заполнен, он вызывает функцию «запись» в связанном модуле записи (в случае, если этот связанный модуль записи является TxtWriter, буферизованный текст будет записан в файл).

Поведенческие

Снимок (Memento)

Снимок — это поведенческий паттерн проектирования, который позволяет сохранять и восстанавливать прошлые состояния объектов, не раскрывая подробностей их реализации.

HandlerOutput может создавать свой снимок, который хранит его состояние, передает его в модель и восстанавливает из него состояние при необходимости. Используя этот шаблон, Модель может сохранять состояния обработчиков, выполнять такт, резервные копии после этого могут быть использованы для возврата такта в любое время.

Чтобы реализовать этот шаблон, создан класс Memento, который имеет массив char (байт) для хранения данных, интерфейс IBackupable (содержит

функции backup () и restore ()). Класс HandlerOutput реализовал этот интерфейс. Функция Backup принимает состояние HandlerOutput для создания объекта класса Memento, функция Restore делает обратное, принимает Memento для восстановления состояния HandlerOutput.

В классе Model создана функция резервного копирования, когда эта функция вызывается, Model вызывает функцию резервного копирования для каждого из обработчиков, находящихся под ее управлением, а затем сохраняет созданный сувенир. Модель не должна знать внутреннюю структуру созданного Memento, она просто хранит Memento и возвращает его обработчику, который создал Memento, когда в модели вызывается функция Undo.

Цепочка обязанностей (Chain of Responsibility)

Цепочка обязанностей — это поведенческий паттерн проектирования, который позволяет передавать запросы последовательно по цепочке обработчиков. Каждый последующий обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи.

Задача в этой лабораторной работе - обновить традиционную цепочку ответственности с 1 входом и 1 выходом. В этом случае каждый обработчик обрабатывает данные, а затем передает их следующему обработчику (одному или нескольким).

Для реализации этого паттерна создан интерфейс IHandler (содержит функцию «handle»), класс BaseHandler, который является реализацией интерфейса IHandler. В BaseHandler хранится указатель на следующий обработчик в цепочке (указатель типа BaseHandler). Реализован дескриптор функции в BaseHandler так, чтобы он принимает параметр, обрабатывает его, затем вызывает функцию «handle» для следующего обработчика в цепочке, если он существует.

Вывод.

В результате этой лабораторной работы я узнал о шаблонах проектирования, их вариантах использования, их плюсах и минусах. Я реализовал 4 шаблона проектирования на языке программирования C ++.