## Министерство образования Республики Беларусь

Учреждение образования «Витебский государственный технологический университет»

Кафедра: информационные системы и автоматизация производства

Расчётно-графическая работа

по курсу

Технологии программирования

Вариант №13

Выполнил:

студент группы 3ИТ-4

Самарин Владислав Васильевич

Проверил: заведующий кафедрой

Казаков Вадим Евгеньевич

**Витебск, 2019**

**Задания к расчётно-графической работе**

**по курсу**

**Технологии программирования**

**Изучить и описать паттерн, согласно индивидуальному варианту.**

**Содержание отчёта:**

1. Описание паттерна: тип, задачи.

2. Диаграмма стандартной схемы организации паттерна.

3. Описание назначения, задач и особенностей каждого из входящих в схему классов.

4. Преимущества применения паттерна.

5. Пример использования паттерна в разработке на языке С++ или Java.

1. **Описание паттерна: тип, задачи.**

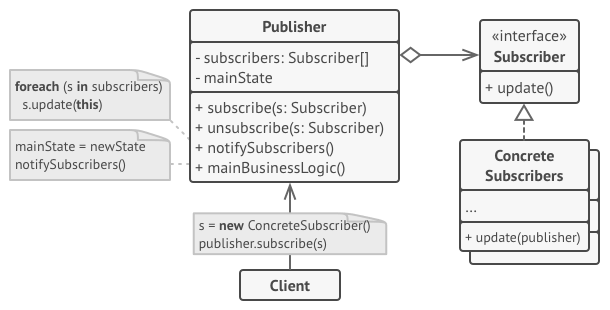
**Наблюдатель** — это поведенческий паттерн проектирования, который создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.

Наблюдатель передаёт запрос одновременно всем заинтересованным получателям, но позволяет им динамически подписываться или отписываться от таких оповещений.

Наблюдатель используется для передачи событий между объектами текстового редактора. Всякий раз когда объект редактора меняет своё состояние, он оповещает своих наблюдателей. Объекты *EmailNotificationListener* и *LogOpenListener* следят за этими уведомлениями и выполняют полезную работу в ответ.

Классы подписчиков не связаны с классом редактора и могут быть повторно использованы в других приложениях если потребуется. Класс *Editor* зависит только от общего интерфейса подписчиков. Это позволяет добавлять новые типы подписчиков не меняя существующего кода редактора.

1. **Диаграмма стандартной схемы организации паттерна.**



1. **Описание назначения, задач и особенностей каждого из входящих в схему классов.**
2. **Издатель** владеет внутренним состоянием, изменение которого интересно отслеживать подписчикам. Издатель содержит механизм подписки: список подписчиков и методы подписки/отписки.
3. Когда внутреннее состояние издателя меняется, он оповещает своих подписчиков. Для этого издатель проходит по списку подписчиков и вызывает их метод оповещения, заданный в общем интерфейсе подписчиков.
4. **Подписчик** определяет интерфейс, которым пользуется издатель для отправки оповещения. В большинстве случаев для этого достаточно единственного метода.
5. **Конкретные подписчики** выполняют что-то в ответ на оповещение, пришедшее от издателя. Эти классы должны следовать общему интерфейсу подписчиков, чтобы издатель не зависел от конкретных классов подписчиков.
6. По приходу оповещения подписчику нужно получить обновлённое состояние издателя. Издатель может передать это состояние через параметры метода оповещения. Более гибкий вариант — передавать через параметры весь объект издателя, чтобы подписчик мог сам получить требуемые данные. Как вариант, подписчик может постоянно хранить ссылку на объект издателя, переданный ему в конструкторе.
7. **Клиент** создаёт объекты издателей и подписчиков, а затем регистрирует подписчиков на обновления в издателях.
8. **Преимущества применения паттерна.**

**Преимущества:**

Издатели не зависят от конкретных классов подписчиков и наоборот.

Возможность подписывать и отписывать получателей на лету.

Реализует принцип открытости/закрытости.

**Недостатки:**

Подписчики оповещаются в случайном порядке.

1. **Пример использования паттерна в разработке на языке Java.**

**class EventManager - Класс базового издателя**

**package** publisher;

**import** java.io.File;

**import** java.util.ArrayList;

**import** java.util.HashMap;

**import** java.util.List;

**import** java.util.Map;

**import** listeners.EventListener;

**public** **class** EventManager {

Map<String, List<EventListener>> listeners = **new** HashMap<>();

**public** EventManager(String... operations) {

**for** (String operation : operations) {

**this**.listeners.put(operation, **new** ArrayList<>());

}

}

**public** **void** subscribe(String eventType, EventListener listener) {

List<EventListener> users = listeners.get(eventType);

users.add(listener);

}

**public** **void** unsubscribe(String eventType, EventListener listener) {

List<EventListener> users = listeners.get(eventType);

users.remove(listener);

}

**public** **void** notify(String eventType, File file) {

List<EventListener> users = listeners.get(eventType);

**for** (EventListener listener : users) {

listener.update(eventType, file);

}

}

}

**class Editor - Конкретный издатель, изменение которого хотят отслеживать наблюдатели**

**package** editor;

**import** java.io.File;

**import** publisher.EventManager;

**public** **class** Editor {

**public** EventManager events;

**private** File file;

**public** Editor() {

**this**.events = **new** EventManager("open", "save");

}

**public** **void** openFile(String filePath) {

**this**.file = **new** File(filePath);

events.notify("open", file);

}

**public** **void** saveFile() **throws** Exception {

**if** (**this**.file != **null**) {

events.notify("save", file);

} **else** {

**throw** **new** Exception("Please open a file first.");

}

}

}

**interface EventListener - Интерфейс подписчиков**

**package** listeners;

**import** java.io.File;

**public** **interface** EventListener {

**void** update(String eventType, File file);

}

**class EmailNotificationListener - Слушатель, рассылающий email-уведомления**

**package** listeners;

**import** java.io.File;

**public** **class** EmailNotificationListener **implements** EventListener {

**private** String email;

**public** EmailNotificationListener(String email) {

**this**.email = email;

}

@Override

**public** **void** update(String eventType, File file) {

System.***out***.println("Email to " + email + ": Someone has performed " + eventType

+ " operation with the following file: " + file.getName());

}

}

**class LogOpenListener - Слушатель, записывающий лог операций**

**package** listeners;

**import** java.io.File;

**public** **class** LogOpenListener **implements** EventListener {

**private** File log;

**public** LogOpenListener(String fileName) {

**this**.log = **new** File(fileName);

}

@Override

**public** **void** update(String eventType, File file) {

System.***out***.println("Save to log " + log + ": Someone has performed " + eventType

+ " operation with the following file: " + file.getName());

}

}

**class Test – Пример клиентского кода**

**package** test;

**import** editor.Editor;

**import** listeners.EmailNotificationListener;

**import** listeners.LogOpenListener;

**public** **class** Test {

**public** **static** **void** main(String[] args) {

Editor editor = **new** Editor();

editor.events.subscribe("open", **new** LogOpenListener("/path/to/log/file.txt"));

editor.events.subscribe("save", **new** EmailNotificationListener("admin@gmail.com"));

**try** {

editor.openFile("test.txt");

editor.saveFile();

} **catch** (Exception e) {

e.printStackTrace();

}

}

}

**OutputTest.txt – Результат выполнения клиентского кода**

Save to log \path\to\log\file.txt: Someone has performed open operation with the following file: test.txt

Email to admin@gmail.com: Someone has performed save operation with the following file: test.txt