Патерн наблюдатель

Когда нужно реагировать на события

Содержание

1. О чем этот патерн, когда его использовать, суть

В предыдущей лекции мы рассмотрели минусы массива и пришли к новой штуке под названием списки. Если помните мы хотели отойти в нашей задаче от цепочки обязанностей и сделать механизм, который бы оповещал кого нужно (дизайнера или программиста или тестировщика) когда в колонку с задачами добавилась новая. Т.е. раньше мы брали задачу, отдавали ее дизайнеру, тот смотрел на статус и или делал ее или отдавал программисту. Теперь же дизайнеру будет приходить оповещение о том, что в колонку TODO добавилась новая задача. Он ее возьмет в работу и отдаст в колонку IN_PROGRESS. Программист же подписан на обновления этой колонки и возьмет эту задачу и будет ее делать. По окончанию перебросит в колонку тестирования. Т.е. если ранее дизайнер знал о такой профессии как программист и перекидывал ему напрямую задачу, то теперь мы делаем более гибко. Дизайнер взял задачу, сделал ее, отдал куда надо. Ему уже не нужно знать кто ее будет делать и как и будет ли вообще. Теперь работа дизйанера будет проще и быстрее – взял задачу из колонки когда она появилась – сделал – отдал в другую колонку.

Для того, чтобы эту задачу решить нам нужно рассмотреть патерн наблюдатель. Мы используем этот патерн тогда, когда есть события (добавилась новая задача) и есть исполнители этой задачи (дизайнер). Самое хорошее еще то, что у нас может быть несколько дизайнеров – и они все сразу будут знать когда появилась задача. И здесь уже нужно сделать разные уровни сложности чтобы не было ситуации когда одну задачу одновременно делают 2 человека. Потому этот патерн более подходит для ситуации, когда 1 событие могут наблюдать несколько человек. Например в андроид есть событие – появился интернет – как только он появился множество приложений (или даже части одного приложения) может его наблюдать и делать то, что нужно. Т.е. в этом случае в андроид есть независимые друг от друга подписчики событий.

И кто-то может спросить – а зачем мы тогда рассматриваем этот патерн? У нас 1 дизайнер, а если и будет более 1 то нужно делать разные уровни сложности чтобы избежать ситуации с лишней работой? Можно придумать следующее – пусть у работника будет состояние занят или нет. Если работник не занят, то он берет задачу и делает ее, а если он занят, то когда появляется новая задача он ее не берет в работу. Попробуем реализовать это в другой лекции.

Ладно, давайте для начала рассмотрим патерн наблюдателя на конкретном нашем случае. Напишем класс колонки задач и поставим ему наблюдателей работников.

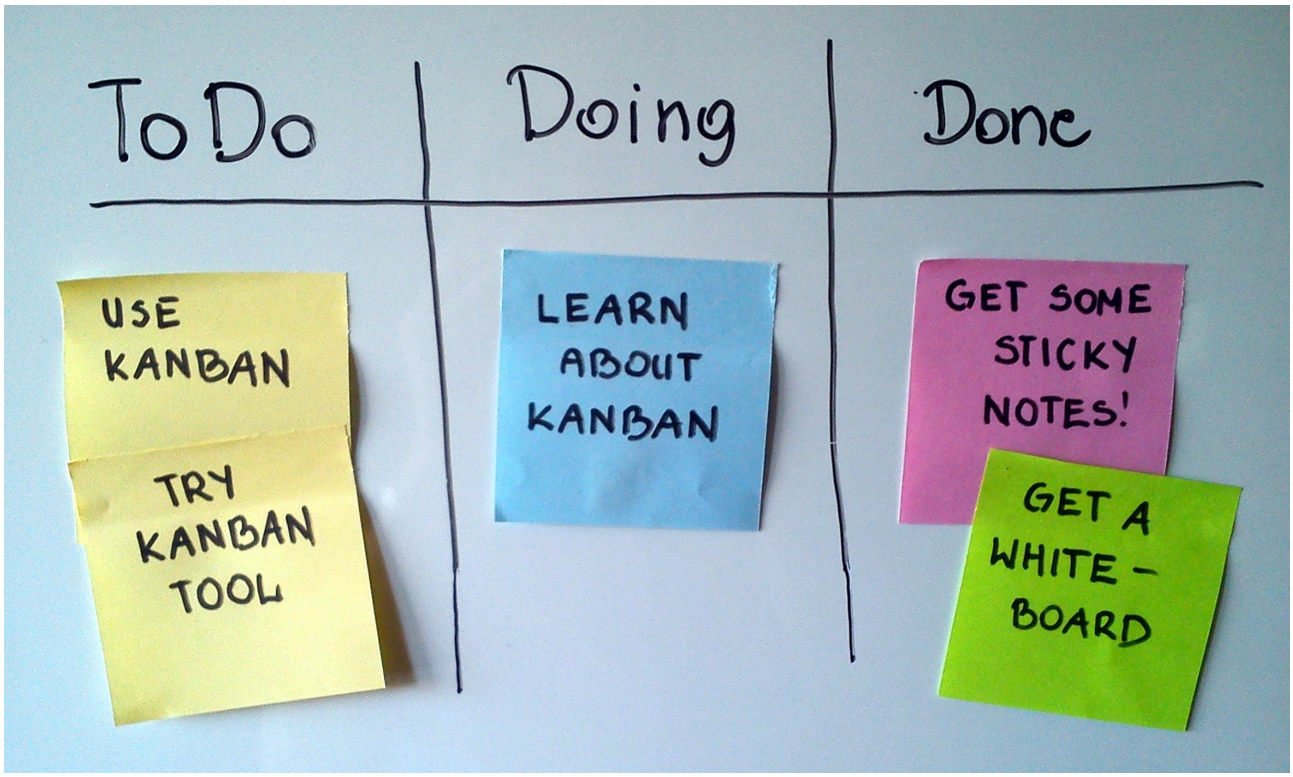
```java
public interface Observer {

    void handleTask(Task task);
}
```

Для начала пусть у нас будет интерфейс наблюдателя. В нем 1 метод для принятия задачи.

```java
public interface Observable {
    void registerObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers();
}
```

Далее у нас будет интерфейс для того, что мы собственно наблюдаем. Это и будет наша колонка задач. Методы регистрации наблюдателя, удаления наблюдателя и оповещения всех наблюдателей.

Теперь напишем класс Колонки задач. У него будет статус задач чтобы разграничивать разные колонки, также список задач которые в нем есть и список наблюдателей – работников. Заметьте что колонка имплементит интерфейс Observable. Т.е. колонку мы и будем наблюдать. Чтобы вы понимали о какой колонке речь вообще вот там иллюстрация.



Как видите в каждой колонке свой тип задачи. Даже стикеры разного цвета. Так же и у нас задачи с разными статусами. Которые совпадают со статусом колонки.

Теперь, у нас есть статус колонки, список задач и список наблюдателей. Мы можем добавить задачу и когда мы это сделаем то оповестим всех наблюдателей об этом событии. В этом и суть патерна – когда произошло действие все кому интересно узнают об этом сразу же.

```java
public class Column implements Observable {

    private final Task.Status status;
    private final List<Observer> observers;
    private final List<Task> tasks;

    public Column(Task.Status status) {
        this.status = status;
        this.observers = new ArrayList<>();
        this.tasks = new ArrayList<>();
    }

    public void addTask(Task task) {
        tasks.add(task);
        notifyObservers();
    }

    public void removeTask(Task task) {
        tasks.remove(task);
    }

    public boolean contains(Task.Status status) {
        return this.status == status;
    }

    @Override
    public void registerObserver(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.handleTask(tasks.get(0));
        }
    }
}
```

Так же мы можем удалить задачу из колонки когда закончим ее делать и методы интерфейса тоже пишем – добавляем в список наблюдателей нашего наблюдателя, можем так же его оттуда убрать и уведомляем всех наблюдателей просто в цикле пройдя по всем и вызвав метод. И еще один метод чтобы снаружи знать что нужно удалить задачу именно из этой колонки когда работник ее сделает. Т.е. какой у нас будет механизм – в колонку добавилась задача – работник узнал о ней – сделал задачу и по завершению должен удалить ее из этой колонки и добавить в другую колонку новую задачу, т.е. ту же но с новыми данными и с новым статусом. Заметьте, задачи добавятся в конец списка, а берем их с начала.

Далее нам нужно немного изменить класс Работника – ранее он получал задачу любого статуса и проверял сможет ли он сделать ее или нет, теперь же не будет ситуации когда наш работник будет рассматривать задачу другого статус, поэтому давайте имплементить ему интерфейс наблюдателя и поменяем немного код.

```
*/
public abstract class Employee implements Observer {

    private final TaskProgressCallback callback;
    private final String name;
    private final Task.Status taskStatus;

    protected Employee(TaskProgressCallback callback,
                       String name,
                       Task.Status taskStatus) {
        this.callback = callback;
        this.name = name;
        this.taskStatus = taskStatus;
    }

    @Override
    public void handleTask(Task task) {
        System.out.println(getClass().getSimpleName() + " " + name
                + " is doing task " + getDetails(task));
        callback.updateTask(getTaskWhenDone(task));
    }

    public final boolean canBeObserverForColumn(Column column) {
        return column.contains(taskStatus);
    }

    protected abstract Task getTaskWhenDone(Task task);

    protected abstract String getDetails(Task task);
}
```

Заметьте мы убрали старый интерфейс TaskHandler и заменили его новым интерфейсом наблюдателя. И мы добавили метод, по которому сможем определить наблюдателя для колонки. Зачем я написал этот метод? Чтобы руками не вписывать всех работников. Т.е. я должен был вписывать каждой колонке своих наблюдателей, но теперь я смогу сделать это автоматически в цикле. Ниже покажу как. И заметьте, что я обьявил метод final иначе бы его можно было бы переопределить в наследниках. Теперь же никто не может поменять этот метод. Его можно лишь использовать.

Но подождите, мы теперь должны по завершении вынимать из колонки старую задачу и новую ставить в новую колонку. Значит наш метод колбека нужно поменять немного.

```
*/
public interface TaskProgressCallback {

    void updateTasks(Task oldTask, Task newTask);
}
```

```
@Override
public void handleTask(Task task) {
    System.out.println(getClass().getSimpleName() + " " + name
            + " is doing task " + getDetails(task));
    callback.updateTasks(task, getTaskWhenDone(task));
}
```

Мы отдадим наружу 2 задачи – старую (чтобы убрать из колонки) и новую (чтобы добавить в новую колонку). Вот теперь все правильно. Хотя в действительности это та же задача, она просто меняется по ходу перехода из одной колонки в другую. Но мы считаем их разными задачами потому что придерживаемся принципа немутабельности класса (private final поля).

И осталось самое сложное – переписать класс фабрики задач.

Только перед этим давайте немного упростим себе жизнь. У нас одна лишняя колонка, ведь программист берет задачу в статусе “готово к работе” потом меняет статус “в работе” и потом уже отдает в колонку “можно тестировать”. Получается на колонку “в работе” никто не подпишется. Потому просто уберем эту колонку из статусов.

```
enum Status {
    ASSEMBLING_REQUIREMENTS,
    READY_TO_DO,
    READY_FOR_TESTING,
    DONE
}
```

Рассмотрим код фабрики задач.

```
public class TaskFactory {

    private static final int SIZE = 10;
    private final Task[] tasks;
    private final List<Column> columns;

    public TaskFactory() {
        tasks = new Task[SIZE];
        for (int i = 0; i < SIZE; i++) {
            tasks[i] = new Task(i, Task.Status.ASSEMBLING_REQUIREMENTS,
                    description: "description " + i, designLink: "", testcase: "", buildLink: "");
        }

        columns = new ArrayList<>();
        columns.add(new Column(Task.Status.ASSEMBLING_REQUIREMENTS));
        columns.add(new Column(Task.Status.READY_TO_DO));
        columns.add(new Column(Task.Status.READY_FOR_TESTING));
        columns.add(new Column(Task.Status.DONE));
    }
```

Рассмотрим частями. У нас так же будет массив из задач для старта(можно в принципе удалить) и добавим список из колонок. В конструкторе инициализируем все задачи и все колонки. Создаем колонки со статусами. Они не знают о своих исполнителях пока что.

Далее нам нужен метод добавления работников к колонкам. Для этого пишем следующий метод

```
public void addEmployees(List<Employee> employees) {
    for (Employee employee : employees) {
        for (Column column : columns) {
            if (employee.canBeObserverForColumn(column)) {
                column.registerObserver(employee);
            }
        }
    }
}
```

Наша фабрика получает работников в списке. Проходит по каждому и для каждого смотрит на колонки и добавляет работника к колонке. Точнее регистрирует в качестве наблюдателя. Заметьте, что фабрика задач не хранит работников, а хранит колонки, на которые подпишутся наши работники. Надеюсь вам все понятно, здесь цикл в цикле. Я специально не поставил брейк предполагая что в дальнейшем может быть работник который подпишется на несколько колонок, но для этого надо будет немного поменять метод canBeObserver.

А еще нам нужен метод для обновления задач. Точнее он был ранее, но нам нужно переписать.

```
public void updateTasks(Task oldTask, Task newTask) {
    for (Column column : columns) {
        if (column.contains(oldTask.getStatus())) {
            column.removeTask(oldTask);
        }
        if (column.contains(newTask.getStatus())) {
            column.addTask(newTask);
        }
    }
}
```

Если помните наши работники отдавали в колбеке при завершении 2 задачи и нам нужно в фабрике поменять содержимое колонок. Смотрим по всем колонкам. Если старая задача содержится в колонке, то она оттуда убирается, если же новая задача должна быть в колонке (согласен, имя метода contains немного не то, должно быть что-то типа belong) то она туда добавляется. И как вы помните, как только задача добавляется к колонке, все кто подписан на изменения будут уведомлены. Не беспокойтесь если вам кажется что все это сложно. Мы подебажим и увидим как развиваются дела. И на последок нам нужен метод, который первично добавит из массива задач в первую колонку.

```
public void start() {
    for (Task task : tasks) {
        columns.get(0).addTask(task);
    }
}
```

Просто берем задачи из массива и кладем в колонку номер 1 – дизайнеры узнают об этом и пойдут делать их.

И наконец-то давайте уже напишем метод мейн. Но мы забыли обновить имплементацию колбека

```
public class CallbackImpl implements TaskProgressCallback {

    private final TaskFactory taskFactory;

    public CallbackImpl(TaskFactory taskFactory) {
        this.taskFactory = taskFactory;
    }

    @Override
    public void updateTasks(Task oldTask, Task newTask) {
        taskFactory.updateTasks(oldTask, newTask);
    }
}
```

Вот теперь можно приступать к мейн методу.

```
public static void main(String[] args) {
    TaskFactory factory = new TaskFactory();
    List<Employee> employees = new ArrayList<>();
    TaskProgressCallback callback = new CallbackImpl(factory);
    employees.add(new Designer(callback, name: "Alicya"));
    employees.add(new Programmer(callback, name: "John"));
    employees.add(new Tester(callback, name: "Steve"));
    factory.addEmployees(employees);
    factory.start();
}
```

Посмотрите какая красота. Все ясно и понятно. Создаем фабрику задач. Создаем список работников. Колбек завершенности задачи. В список добавляем работников. Отдаем этот список фабрике и стартуем. Для упрощения я изменю количество задач на 2 (к слову о том, зачем нужно выносить такие константы).

```
Designer Alicya is doing task  with taskId0 and description description 0
Programmer John is doing task  with task id0 and designLinkhttps://project/design_link_for_task_with_id 0
and testcase when description 0 get result 0.30223497480563843
Tester Steve is doing task with id 0and testcase when description 0 get result 0.30223497480563843
Designer Alicya is doing task  with taskId1 and description description 1
Programmer John is doing task  with task id1 and designLinkhttps://project/design_link_for_task_with_id 1
and testcase when description 1 get result 0.035379276446041774
Tester Steve is doing task with id 1and testcase when description 1 get result 0.035379276446041774
```

Так, стоп. Похоже на тот же вывод что и был ранее. Мы говорили что теперь дизайнер должен делать все задачи подряд, а не ждать программиста. Как же нам проверить что так и происходит? Для этого давайте залогируем вызовы добавления и удаления задач из колонки и поставим программисту другой колбек завершенности и посмотрим на логи.

А чтобы не высвечивать логи от работников, просто закомментируем их посредством Ctrl+’/’.

Ставим курсор на линию кода и жмем комбинацию – код стал комментарием , нажали еще раз и код вернулся в работу. Комментарии в джава это чисто текст, который игнорируется компилятором

```
    @Override
    public void handleTask(Task task) {
//        System.out.println(getClass().getSimpleName() + " " + name
//                + " is doing task " + getDetails(task));
        callback.updateTasks(task, getTaskWhenDone(task));
    }
```

```
public static void main(String[] args) {
    TaskFactory factory = new TaskFactory();
    List<Employee> employees = new ArrayList<>();
    TaskProgressCallback callback = new CallbackImpl(factory);
    employees.add(new Designer(callback, name: "Alicya"));
    employees.add(new Programmer(new TaskProgressCallback() {
        @Override
        public void updateTasks(Task oldTask, Task newTask) {
            System.out.println("Programmer updateTask");
        }
    }, name: "John"));
    employees.add(new Tester(callback, name: "Steve"));
    factory.addEmployees(employees);
    factory.start();
}
```

```
public void addTask(Task task) {
    System.out.println("Column " + status + " addTask " + task.getId());
    tasks.add(task);
    notifyObservers();
}

public void removeTask(Task task) {
    System.out.println("Column " + status + " removeTask " + task.getId());
    tasks.remove(task);
}
```

```
/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/java ...
Column ASSEMBLING_REQUIREMENTS addTask 0
Column ASSEMBLING_REQUIREMENTS removeTask 0
Column READY_TO_DO addTask 0
Programmer updateTask
Column ASSEMBLING_REQUIREMENTS addTask 1
Column ASSEMBLING_REQUIREMENTS removeTask 1
Column READY_TO_DO addTask 1
Programmer updateTask
```

Как видите цепочка разорвалась, наш программист не меняет содержимое колонки и тестировщику не попадают задачи. Но это на самом деле ложное доказательство. По-настоящему дело в том, что весь наш код синхронный т.е. последовательный. Мы работаем в одном потоке. Об этом поговорим в следующей лекции.