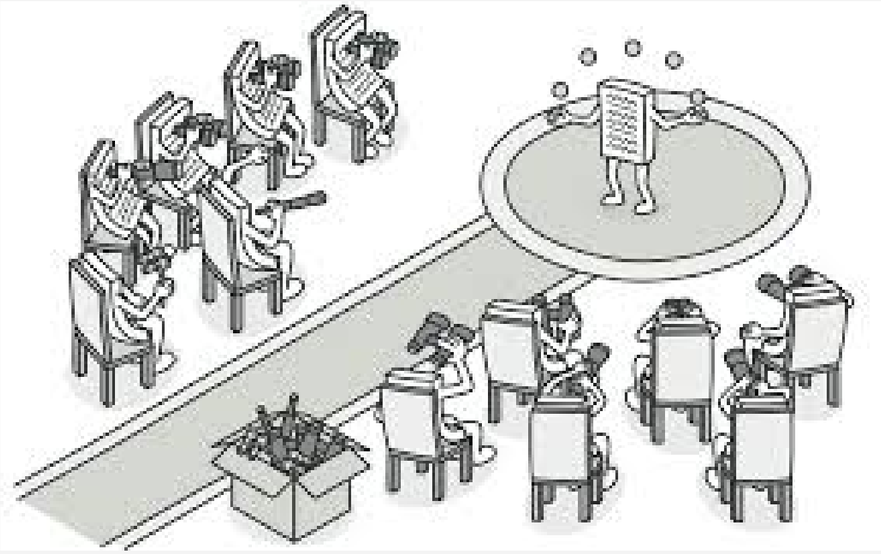
OBSERVER

Design Pattern

13/04/2024

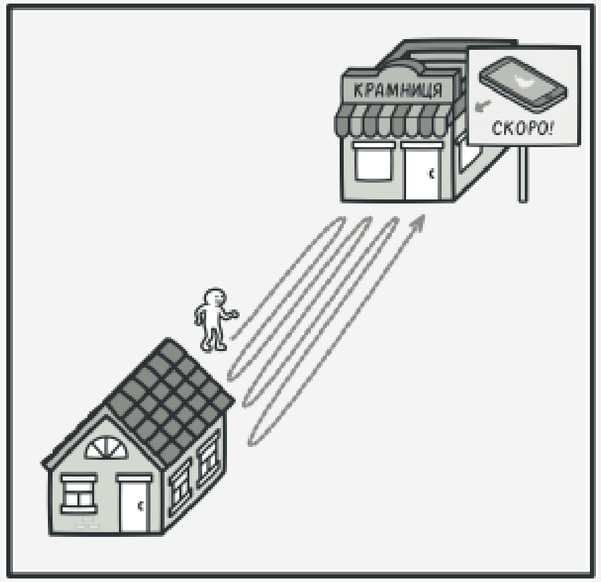


PRESENTED BY

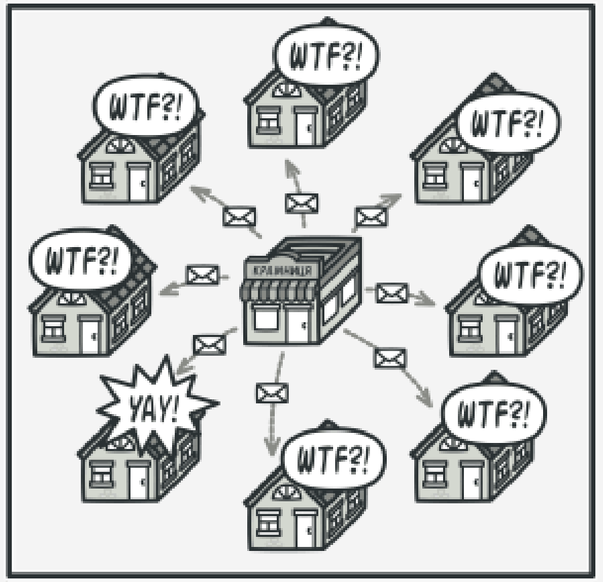
Кучеренко Данiіл та
Яковенко Єлизавета

ПРОБЛЕМА

Уявімо, що покупець цікавиться якимось товаром і очікує на його постачу



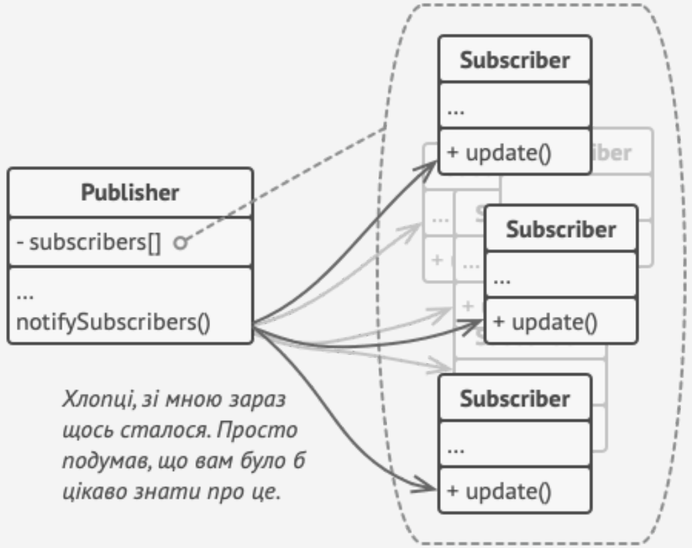
Щодня ходити до магазину перевіряти наявність
(марне гаяння часу)



Розсилання електроних листів усім користувач (розтрачення зайвих ресурсів на непотрібні сповіщення

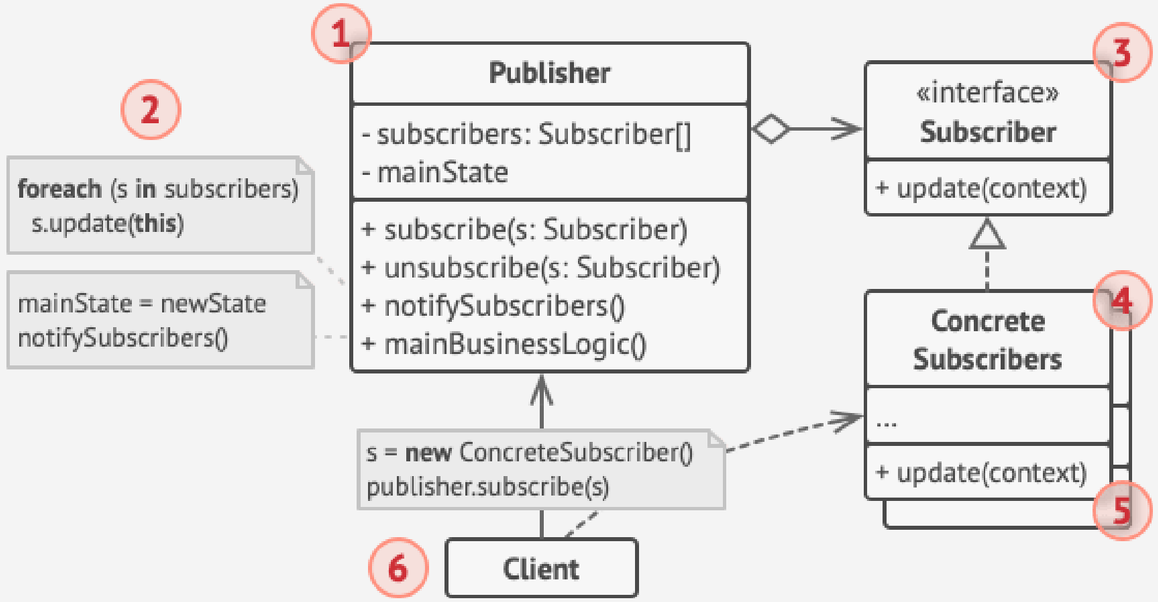
Патерн Observer

Publisher'ами будемо називати ті об'єкти, які містять важливий або цікавий для інших стан.



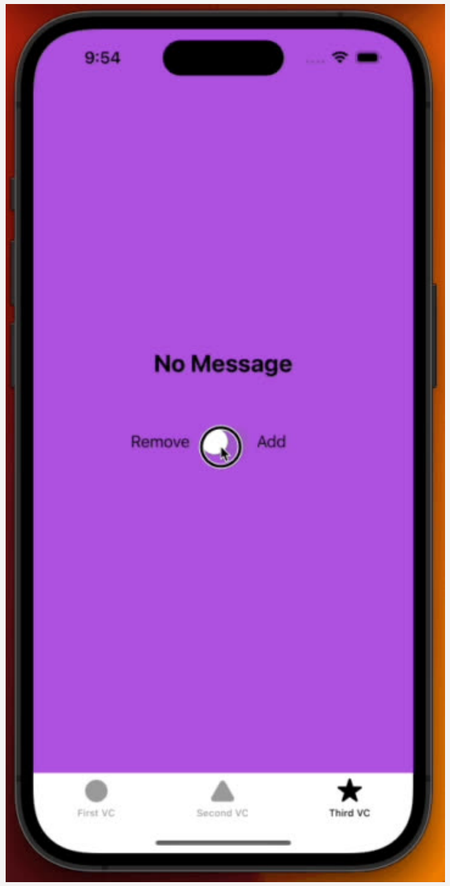
Коли у publisher'а відбуватиметься важлива подія, він буде проходитися за списком subscriber'ів та сповіщувати їх про подію, викликаючи певний метод об'єктів цього класу.

Структура реалізації



- publisher
- сповіщення про зміну стану publisher’a
- subscriber
- concrete subscribers
- client

Застосування на прикладі UI користувача



- Потреба змінити стану в невизначених об’єктах при зміні стану в одному
- Потреба спостерігання одних об’єктів за іншими в певних станах

Переваги

- Видавці не залежать від конкретних класів підписників і навпаки.
- Ви можете підписувати і відписувати одержувачів «на льоту».
- Реалізує принцип відкритості/ закритості.

Недоліки

- Підписники сповіщуються у випадковій послідовності.

Релізація на Java

```java
public class EventManager {
    4 usages
    Map<String, List<EventListener>> listeners = new HashMap<>();

    1 usage
    public EventManager(String... operations) {
        for (String operation : operations) {
            this.listeners.put(operation, new ArrayList<>());
        }
    }

    2 usages
    public void subscribe(String eventType, EventListener listener) {
        List<EventListener> users = listeners.get(eventType);
        users.add(listener);
    }

    no usages
    public void unsubscribe(String eventType, EventListener listener) {
        List<EventListener> users = listeners.get(eventType);
        users.remove(listener);
    }

    2 usages
    public void notify(String eventType, File file) {
        List<EventListener> users = listeners.get(eventType);
        for (EventListener listener : users) {
            listener.update(eventType, file);
        }
    }
}
```

```java
public class Editor {
    5 usages
    public EventManager events;
    4 usages
    private File file;

    1 usage
    public Editor() {
        this.events = new EventManager( ...operations: "open", "save");
    }

    1 usage
    public void openFile(String filePath) {
        this.file = new File(filePath);
        events.notify( eventType: "open", file);
    }

    1 usage
    public void saveFile() throws Exception {
        if (this.file != null) {
            events.notify( eventType: "save", file);
        } else {
            throw new Exception("Please open a file first.");
        }
    }
}
```

```java
public interface EventListener {
    1 usage  2 implementations
    void update(String eventType, File file);
}
```

```java
public class EmailNotificationListener implements EventListener {
    2 usages
    private String email;

    1 usage
    public EmailNotificationListener(String email) {
        this.email = email;
    }

    1 usage
    @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());
    }
}
```

```java
public class LogOpenListener implements EventListener {
    2 usages
    private File log;

    1 usage
    public LogOpenListener(String fileName) {
        this.log = new File(fileName);
    }

    1 usage
    @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());
    }
}
```

Висновки

Основні переваги включають:

- Зниження залежності між класами publisher’ів і subscriber’ів, що дозволяє легко розширювати і модифікувати функціонал обох.
- Можливість динамічно підписуватися і відписуватися від сповіщень, що надає гнучкість у керуванні подіями.
- Підтримка принципу відкритості/закритості, де система відкрита для розширення, але закрита для модифікації.

При реалізації патерна Observer, розробники повинні враховувати такі аспекти:

- Сповіщення відбуваються у випадковій послідовності, що може бути недоліком у деяких застосуваннях.
- Важливо добре продумати систему подій, щоб уникнути надмірної взаємодії або залежностей.

Дякую за увагу!