



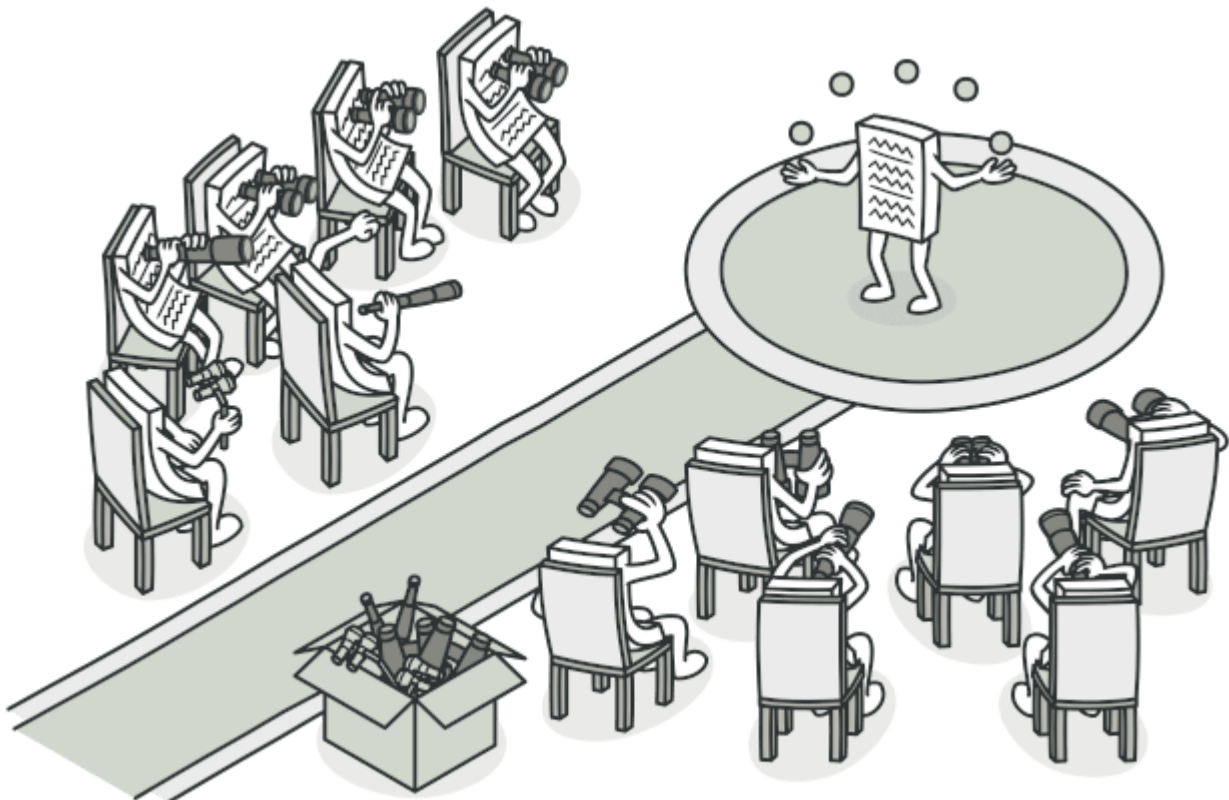
🏠 / Паттерны проектирования / Поведенческие паттерны

Наблюдатель

Также известен как: Издатель-Подписчик, Слушатель, Observer

💬 Суть паттерна

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

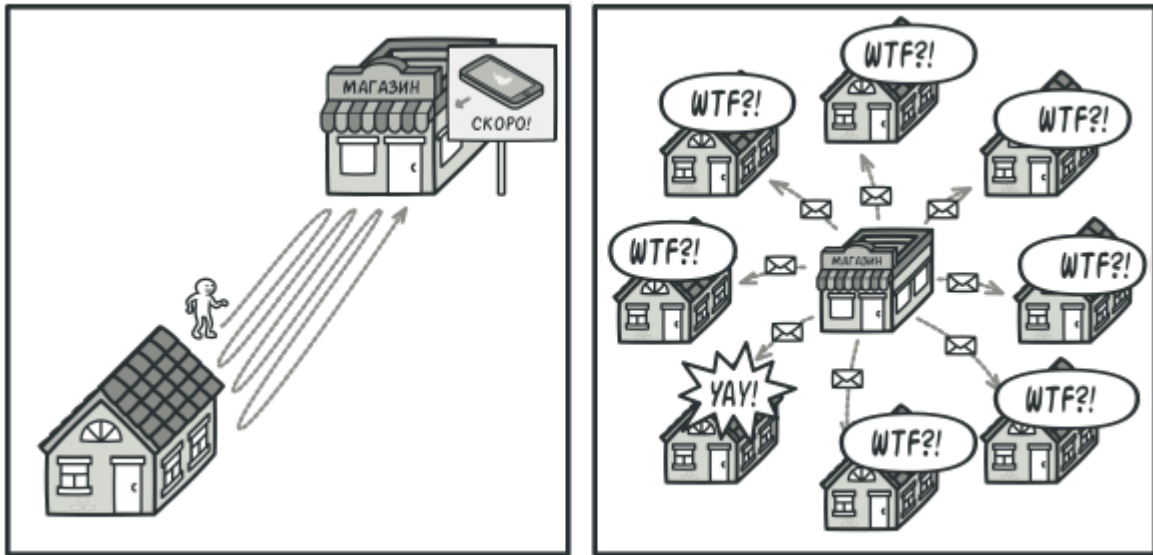


😞 Проблема

Представьте, что вы имеете два объекта: `Покупатель` и `Магазин`. В магазин вот-вот должны завезти новый товар, который интересен покупателю.


НОВОГОДНЯЯ РАСПРОДАЖА!

 ЭТОМ ОН БУДЕТ ЗАНЯТ, БЕЗ ТОЛКУ ТРАТЯ СВОЕ ДРАГОЦЕННОЕ ВРЕМЯ.



Постоянное посещение магазина или спам?

С другой стороны, магазин может разослать спам каждому своему покупателю. Многих это расстроит, так как товар специфический, и не всем он нужен.

Получается конфликт: либо покупатель тратит время на периодические проверки, либо магазин тратит ресурсы на бесполезные оповещения.

😊 Решение

Давайте называть **Издателями** те объекты, которые содержат важное или интересное для других состояние. Остальные объекты, которые хотят отслеживать изменения этого состояния, назовём **Подписчиками**.

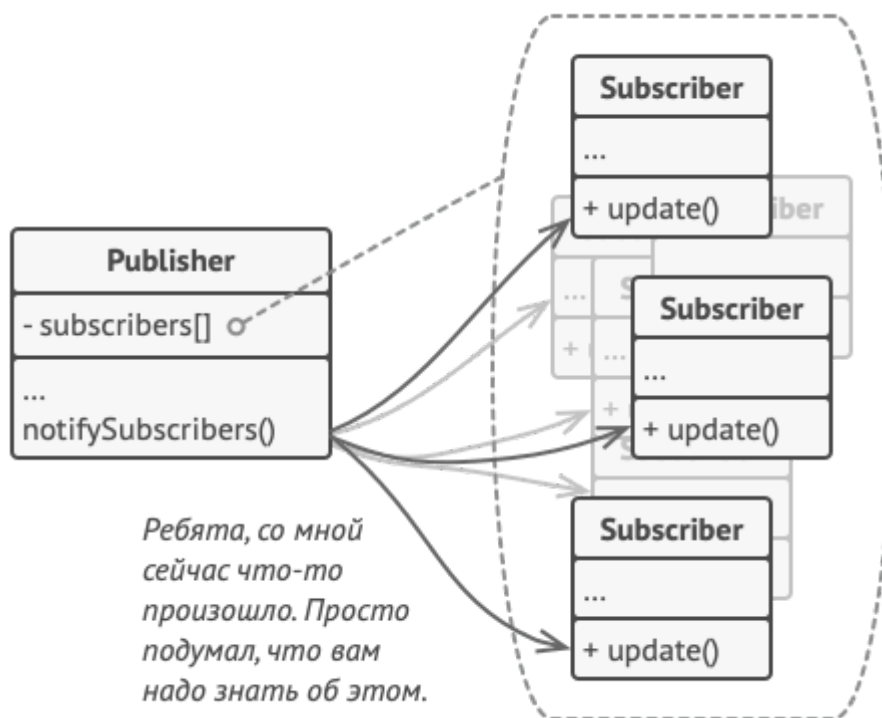
Паттерн Наблюдатель предлагает хранить внутри объекта издателя список ссылок на объекты подписчиков, причём издатель не должен вести список подписки самостоятельно. Он предоставит методы, с помощью которых подписчики могли бы добавлять или убирать себя из списка.



Подписка на события.

Теперь самое интересное. Когда в издателе будет происходить важное событие, он будет проходиться по списку подписчиков и оповещать их об этом, вызывая определённый метод объектов-подписчиков.

Издателю безразлично, какой класс будет иметь тот или иной подписчик, так как все они должны следовать общему интерфейсу и иметь единый метод оповещения.

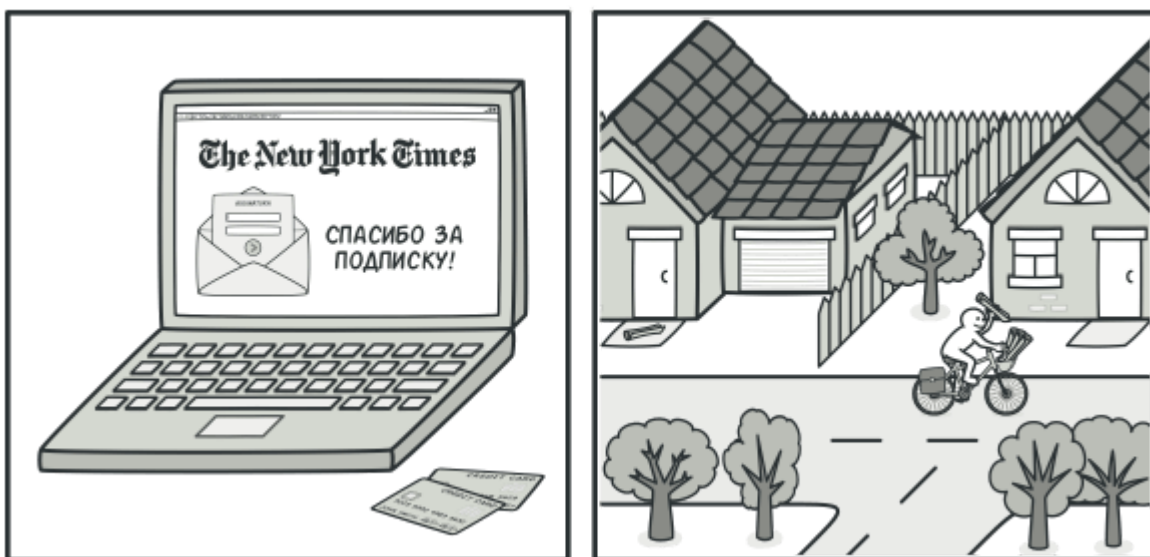


Оповещения о событиях.

Увидев, как сложно всё работает, вы можете выделить общий интерфейс, описывающий методы подписки и отписки, и для всех издателей. После этого подписчики смогут работать с разными типами издателей, а также получать оповещения от них через один и тот же метод.



НОВОГОДНЯЯ РАСПРОДАЖА!

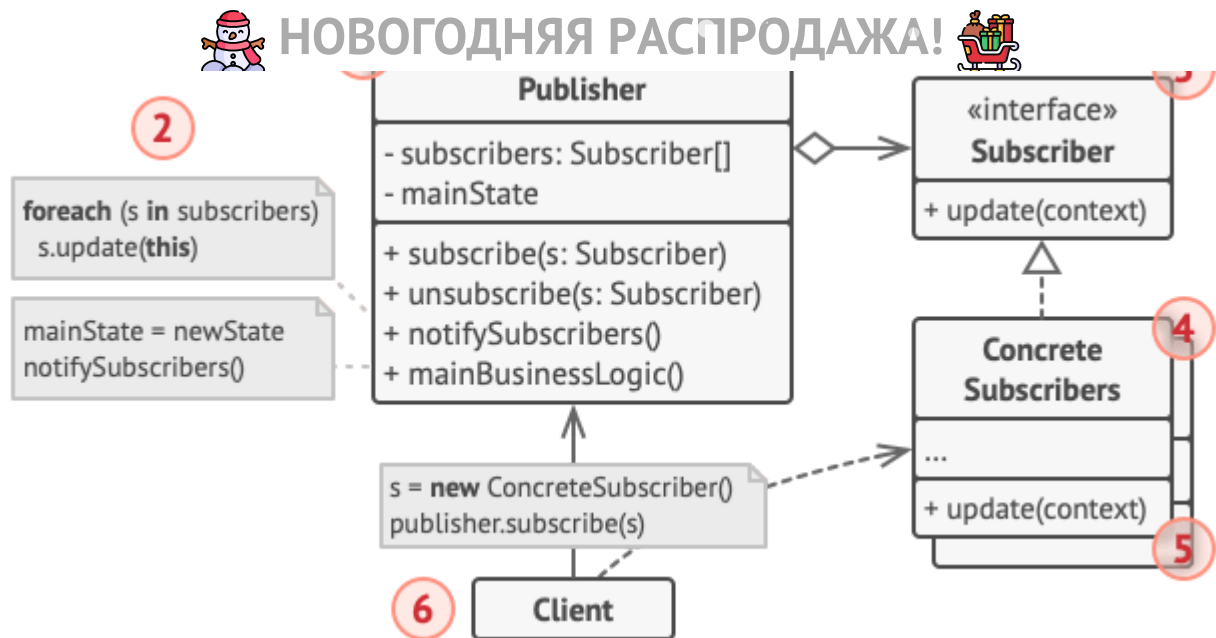


Подписка на газеты и их доставка.

После того как вы оформили подписку на газету или журнал, вам больше не нужно ездить в супермаркет и проверять, не вышел ли очередной номер. Вместо этого издательство будет присылать новые номера по почте прямо к вам домой сразу после их выхода.

Издательство ведёт список подписчиков и знает, кому какой журнал высылать. Вы можете в любой момент отказаться от подписки, и журнал перестанет вам приходить.

Структура



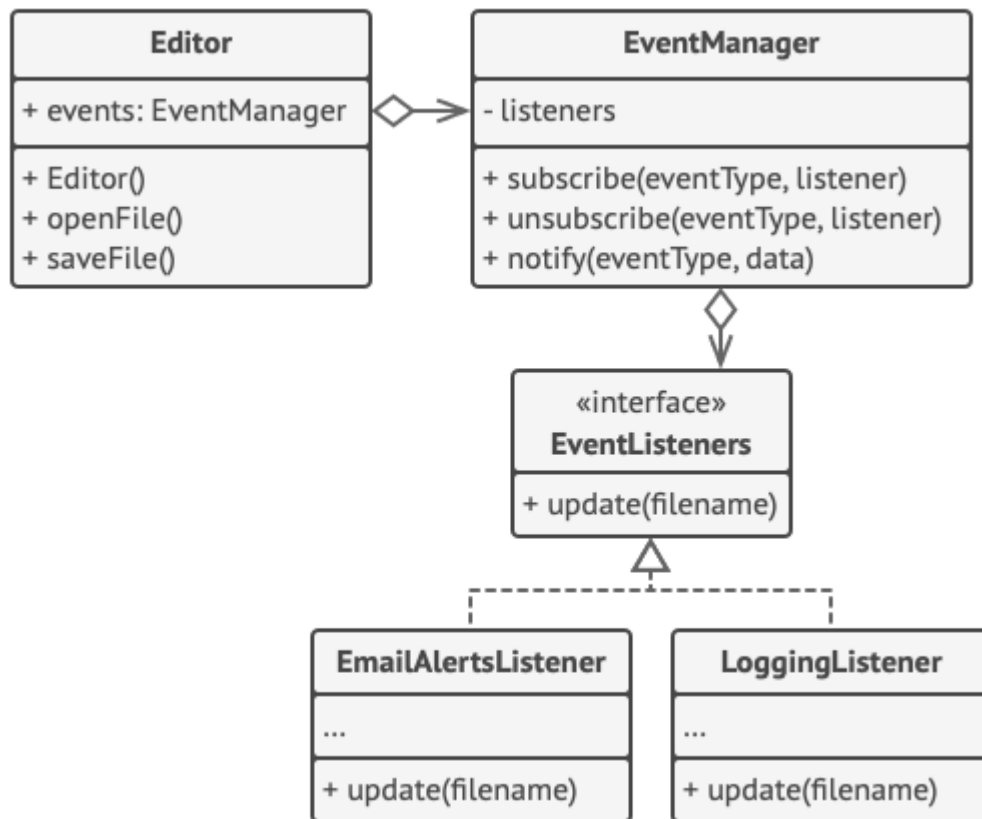
- Издатель** владеет внутренним состоянием, изменение которого интересно отслеживать подписчикам. Издатель содержит механизм подписки: список подписчиков и методы подписки/отписки.
- Когда внутреннее состояние издателя меняется, он оповещает своих подписчиков. Для этого издатель проходит по списку подписчиков и вызывает их метод оповещения, заданный в общем интерфейсе подписчиков.
- Подписчик** определяет интерфейс, которым пользуется издатель для отправки оповещения. В большинстве случаев для этого достаточно единственного метода.
- 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.
- 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.
- Клиент** создаёт объекты издателей и подписчиков, а затем регистрирует подписчиков на обновления в издателях.

Псевдокод

ОБЪЕКТЫ С ИЗМЕНЕНИЯМИ СВОЕГО СОСТОЯНИЯ.



НОВОГОДНЯЯ РАСПРОДАЖА!



Пример оповещения объектов о событиях в других объектах.

Список подписчиков составляется динамически, объекты могут как подписываться на определённые события, так и отписываться от них прямо во время выполнения программы.

В этой реализации редактор не ведёт список подписчиков самостоятельно, а делегирует это вложенному объекту. Это даёт возможность использовать механизм подписки не только в классе редактора, но и в других классах программы.

Для добавления в программу новых подписчиков не нужно менять классы издателей, пока они работают с подписчиками через общий интерфейс.

```
// Базовый класс-издатель. Содержит код управления подписчиками
// и их оповещения.
```

```
class EventManager is
    private field listeners: hash map of event types and listeners

    method subscribe(eventType, listener) is
        listeners.add(eventType, listener)

    method unsubscribe(eventType, listener) is
```



НОВОГОДНЯЯ РАСПРОДАЖА!



```

method notify(eventType, data) is
    foreach (listener in listeners.of(eventType)) do
        listener.update(data)

// Конкретный класс-издатель, содержащий интересную для других
// компонентов бизнес-логику. Мы могли бы сделать его прямым
// потомком EventManager, но в реальной жизни это не всегда
// возможно (например, если у класса уже есть родитель). Поэтому
// здесь мы подключаем механизм подписки при помощи композиции.
class Editor is
    public field events: EventManager
    private field file: File

    constructor Editor() is
        events = new EventManager()

    // Методы бизнес-логики, которые оповещают подписчиков об
    // изменениях.
    method openFile(path) is
        this.file = new File(path)
        events.notify("open", file.name)

    method saveFile() is
        file.write()
        events.notify("save", file.name)
    // ...

// Общий интерфейс подписчиков. Во многих языках, поддерживающих
// функциональные типы, можно обойтись без этого интерфейса и
// конкретных классов, заменив объекты подписчиков функциями.
interface EventListener is
    method update(filename)

// Набор конкретных подписчиков. Они реализуют добавочную
// функциональность, реагируя на извещения от издателя.
class LoggingListener implements EventListener is
    private field log: File
    private field message: string

    constructor LoggingListener(log_filename, message) is
        this.log = new File(log_filename)
        this.message = message

    method update(filename) is
        log.write(replace('%s', filename, message))

class EmailAlertsListener implements EventListener is
    private field email: string
    private field message: string

```



НОВОГОДНЯЯ РАСПРОДАЖА!



```
this.email = email
```

```
this.message = message
```

```
method update(filename) is
```

```
    system.email(email, replace('%s', filename, message))
```

```
// Приложение может сконфигурировать издателей и подписчиков как  
// угодно, в зависимости от целей и окружения.
```

```
class Application is
```

```
    method config() is
```

```
        editor = new Editor()
```

```
        logger = new LoggingListener(
```

```
            "/path/to/log.txt",
```

```
            "Someone has opened file: %s");
```

```
        editor.events.subscribe("open", logger)
```

```
        emailAlerts = new EmailAlertsListener(
```

```
            "admin@example.com",
```

```
            "Someone has changed the file: %s")
```

```
        editor.events.subscribe("save", emailAlerts)
```

💡 Применимость

🔗 Когда после изменения состояния одного объекта требуется что-то сделать в других, но вы не знаете наперёд, какие именно объекты должны отреагировать.

⚡ Описанная проблема может возникнуть при разработке библиотек пользовательского интерфейса, когда вам надо дать возможность сторонним классам реагировать на клики по кнопкам.

Паттерн Наблюдатель позволяет любому объекту с интерфейсом подписчика зарегистрироваться на получение оповещений о событиях, происходящих в объектах-издателях.

🔗 Когда одни объекты должны наблюдать за другими, но только в определённых случаях.

⚡ Издатели ведут динамические списки. Все наблюдатели могут подписываться или отписываться от получения оповещений прямо во время выполнения программы.



НОВОГОДНЯЯ РАСПРОДАЖА!



1. Разбейте вашу функциональность на две части: независимое ядро и опциональные зависимые части. Независимое ядро станет издателем. Зависимые части станут подписчиками.
2. Создайте интерфейс подписчиков. Обычно в нём достаточно определить единственный метод оповещения.
3. Создайте интерфейс издателей и опишите в нём операции управления подпиской. Помните, что издатель должен работать только с общим интерфейсом подписчиков.
4. Вам нужно решить, куда поместить код ведения подписки, ведь он обычно бывает одинаков для всех типов издателей. Самый очевидный способ — вынести этот код в промежуточный абстрактный класс, от которого будут наследоваться все издатели.

Но если вы интегрируете паттерн в существующие классы, то создать новый базовый класс может быть затруднительно. В этом случае вы можете поместить логику подписки во вспомогательный объект и делегировать ему работу из издателей.

5. Создайте классы конкретных издателей. Реализуйте их так, чтобы после каждого изменения состояния они отправляли оповещения всем своим подписчикам.
6. Реализуйте метод оповещения в конкретных подписчиках. Не забудьте предусмотреть параметры, через которые издатель мог бы отправлять какие-то данные, связанные с происшедшим событием.

Возможен и другой вариант, когда подписчик, получив оповещение, сам возьмёт из объекта издателя нужные данные. Но в этом случае вы будете вынуждены привязать класс подписчика к конкретному классу издателя.

7. Клиент должен создавать необходимое количество объектов подписчиков и подписывать их у издателей.

⚖️ Преимущества и недостатки

- | | |
|---|---|
| ✓ Издатели не зависят от конкретных классов подписчиков и наоборот. | ✗ Подписчики оповещаются в случайном порядке. |
| ✓ Вы можете подписывать и отписывать получателей на лету. | |



НОВОГОДНЯЯ РАСПРОДАЖА!



⇔ Отношения с другими паттернами

- Цепочка обязанностей, Команда, Посредник и Наблюдатель показывают различные способы работы отправителей запросов с их получателями:
 - *Цепочка обязанностей* передаёт запрос последовательно через цепочку потенциальных получателей, ожидая, что какой-то из них обработает запрос.
 - *Команда* устанавливает косвенную одностороннюю связь от отправителей к получателям.
 - *Посредник* убирает прямую связь между отправителями и получателями, заставляя их общаться опосредованно, через себя.
 - *Наблюдатель* передаёт запрос одновременно всем заинтересованным получателям, но позволяет им динамически подписываться или отписываться от таких оповещений.
- Разница между Посредником и Наблюдателем не всегда очевидна. Чаще всего они выступают как конкуренты, но иногда могут работать вместе.

Цель *Посредника* — убрать обоюдные зависимости между компонентами системы. Вместо этого они становятся зависимыми от самого посредника. С другой стороны, цель *Наблюдателя* — обеспечить динамическую одностороннюю связь, в которой одни объекты косвенно зависят от других.

Довольно популярна реализация *Посредника* при помощи *Наблюдателя*. При этом объект посредника будет выступать издателем, а все остальные компоненты станут подписчиками и смогут динамически следить за событиями, происходящими в посреднике. В этом случае трудно понять, чем же отличаются оба паттерна.

Но *Посредник* имеет и другие реализации, когда отдельные компоненты жёстко привязаны к объекту посредника. Такой код вряд ли будет напоминать *Наблюдателя*, но всё же останется *Посредником*.

Напротив, в случае реализации посредника с помощью *Наблюдателя* представим такую программу, в которой каждый компонент системы становится издателем. Компоненты могут подписываться друг на друга, в то же время не привязываясь к конкретным классам. Программа будет состоять из целой сети *Наблюдателей*, не имея центрального объекта-*Посредника*.