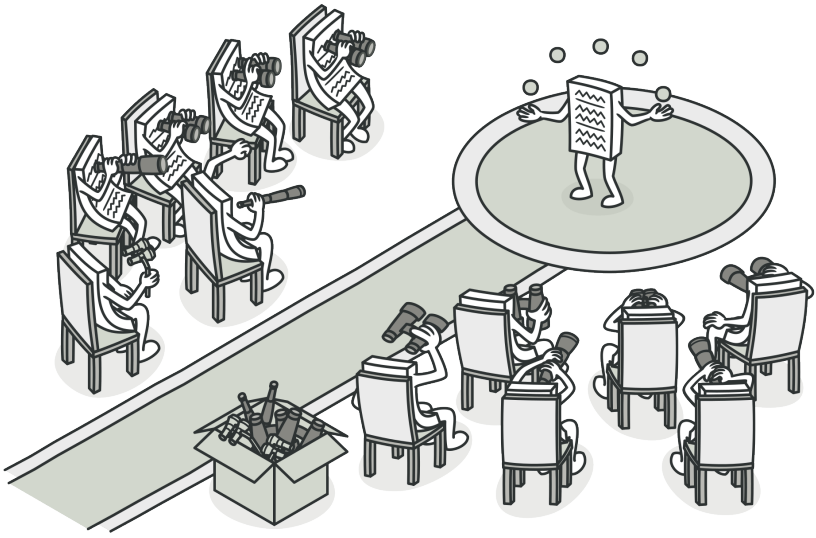


НАБЛЮДАТЕЛЬ

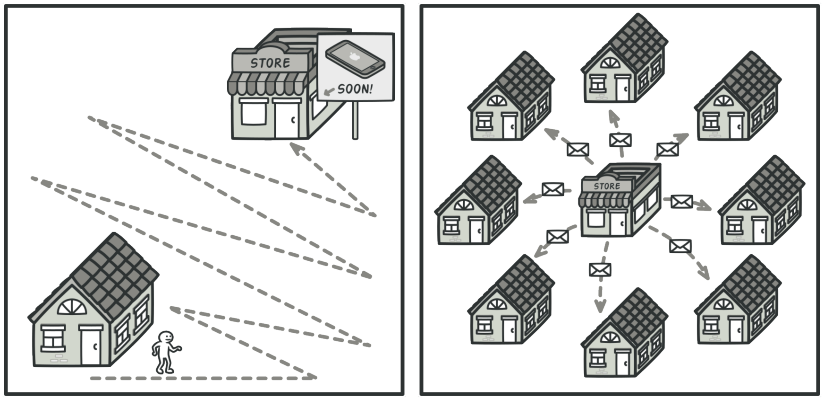
Также известен как: Издатель-Подписчик, Слушатель, Observer

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

☹ Проблема

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

Покупатель может каждый день ходить в магазин, чтобы проверить наличие товара. Но при этом он будет тратить драгоценное время и злиться.



Постоянное посещение магазинга или спам?

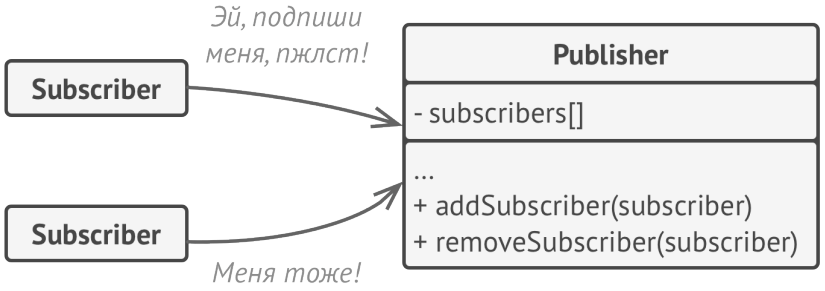
С другой стороны, магазин может разослать спам каждому своему покупателю. Многих это расстроит, так как товар специфический и не всем он нужен.

Получается конфликт: либо один объект работает неэффективно, тратя ресурсы на периодические проверки, либо второй объект оповещает слишком широкий круг пользователей, тоже тратя ресурсы впустую.

☺ Решение

Давайте будем называть объекты, которые содержат интересное состояние `Издателями`. А другие объекты, которым интересно это состояние давайте звать `Подписчиками`.

Паттерн Наблюдатель предлагает издателям хранить список подписчиков, которым интересно наблюдать за их состоянием. Причём, издатели не должны вести этот список сами, а предоставить возможность самим подписчикам вписывать себя в этот список или вычёркиваться оттуда.

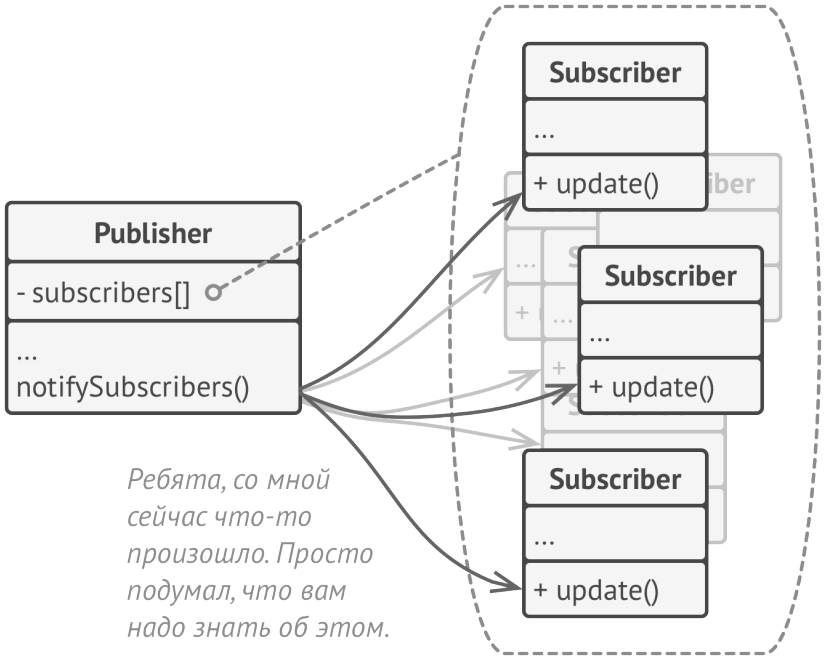


Подписка на события.

Теперь самое интересное. Каждый раз, когда в издателе будет происходить что-нибудь, он должен проходиться по списку своих подписчиков и оповещать их, вызывая определённый метод их объектов.

Для издателя не важно, какого конкретно подписчика он оповещает, так как все подписчики договорились иметь

один и тот же метод оповещения, то есть иметь общий интерфейс.



Оповещения о событиях.

Увидев как складно всё работает, вы можете выделить общий интерфейс и для всех издателей. Он получит методы подписки и отписки. После этого подписчики смогут однотипно работать с издателями, а также получать оповещения от них через один и тот же метод.

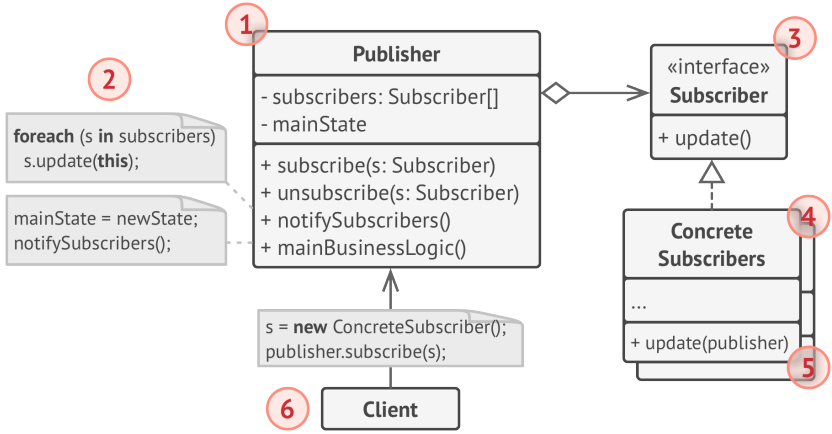
Аналогия из жизни

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 сразу после выхода прямо к вам домой.

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

Структура



1. **Издатель** владеет внутренним состоянием, изменение которого интересно для подписчиков. Он содержит механизм подписки — список подписчиков, а также методы подписки/отписки.

2. Когда внутреннее состояние издателя меняется, он оповещает своих подписчиков. Для этого издатель проходит по списку подписчиков и вызывает их метод оповещения, заданный в интерфейсе подписчика.

3. **Подписчик** определяет интерфейс, которым пользуется издатель для отправки оповещения. В большинстве случаев, для этого достаточно единственного метода.

4. **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.

5. 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 сам мог получить требуемые данные. Как вариант, подписчик может постоянно хранить ссылку на объект издателя, переданный ему в конструкторе.

6. **Клиент** создаёт объекты издателей и подписчиков, а затем регистрирует подписчиков на обновления в издателях.

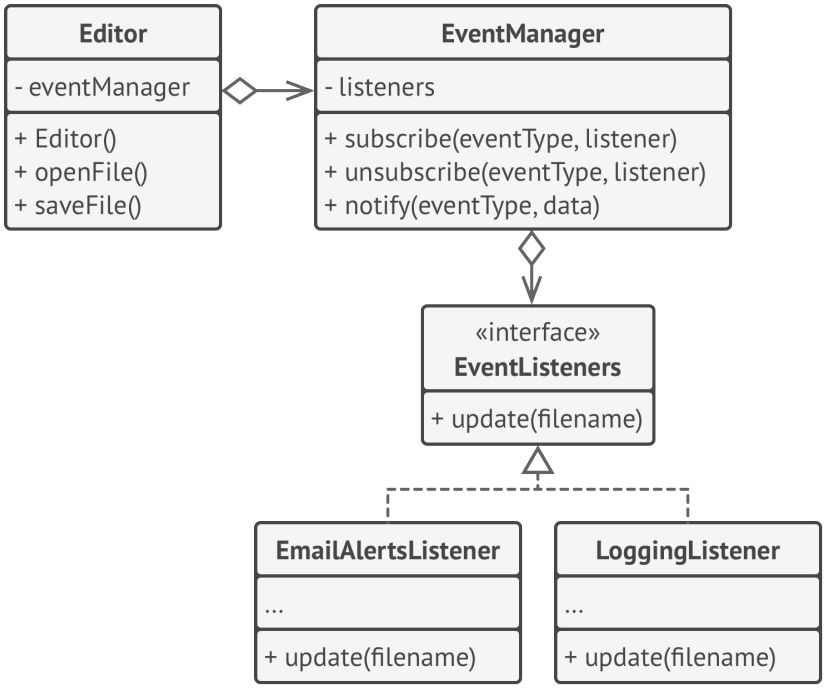
Псевдокод

В этом примере **Наблюдатель** позволяет объекту текстового редактора оповещать другие объекты об изменениях своего состояния.

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 сам, а делегирует это вложенному объекту. Это даёт возможность использовать механизм подписки и в других

объектах программы, а не только в классе редактора. Таким образом, паттерн Наблюдатель позволяет динамически настраивать обработчики тех или иных событий, происходящих с объектом.



Пример оповещения объектов о событиях в других объектах.

Для добавления в программу новых подписчиков, не нужно менять классы издателей, покуда 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
    listeners.remove(eventType, listener)

  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 вы у класса уже есть
// родитель). Поэтому здесь мы подключаем механизм подписки
// при помощи композиции.
class Editor is
  private 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, имеющих
// функциональный 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 is
  private field log: File
  private field message

  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 is
  private field email: string

  constructor EmailAlertsListener(email, message) is
    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 конфигурации.
class Application is
  method config() is
    editor = new TextEditor()

    logger = new LoggingListener(
      "/path/to/log.txt",
      "Someone has opened file: %s");
    editor.events.subscribe("open", logger)

    emailAlers = new EmailAlertsListener(
      "admin@example.com",
      "Someone has changed the file: %s")
    editor.events.subscribe("save", emailAlers)
```

Применимость

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

Эта задача может возникнуть при разработке GUI-фреймворка, когда надо дать возможность сторонним классам реагировать на клики по кнопкам.

Паттерн Наблюдатель даёт возможность любому объекту с интерфейсом подписчика, подписываться на изменения 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 на обновления прямо во время выполнения программы.

Шаги реализации

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 при каждом изменении состояния, они слали оповещения всем своим подписчикам.

6. Реализуйте метод оповещения в конкретных подписчиках. Издатель может отправлять какие-то данные вместе с оповещением (например, в параметрах). Возможен и другой вариант, когда подписчик, получив оповещение, сам берёт из объекта издателя нужные данные. Но при этом подписчик привяжет себя к конкретному классу издателя.

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

⚖ Преимущества и недостатки

- ✔ Издатель не зависит от конкретных классов подписчиков.
- ✔ Вы можете подписывать и отписывать получателей на лету.

✔ Реализует *принцип открытости/закрытости*.

✖ Наблюдатели оповещаются в случайном порядке.

⇄ 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 подписывать 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*.