**Наблюдатель – это поведенческий паттерн.**

Поведенческий паттерны – определяют алгоритмы и взаимодействия между классами и объектами, то есть их поведение.

В паттернах уровня класса используется наследование, а в уровнях объектов – композиция.

**Суть паттерна**

Определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом и автоматически обновляются.

Такого рода взаимодействие часто называется отношением **издатель – подписчик**. Субъект издает или публикует уведомление и рассылает их, даже не имея информации о том, какие объекты являются подписчиками. На получение уведомлений может подписаться неограниченное количество наблюдателей.

**Пример из жизни**

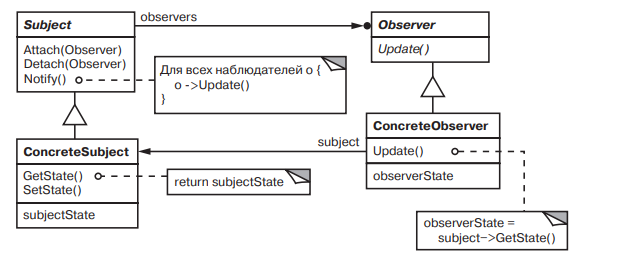
После того как вы оформили подписку на газету или журнал, вам больше не нужно ездить в супермаркет и проверять, не вышел ли очередной номер. Вместо этого издательство будет присылать новые номера по почте прямо к вам домой сразу после их выхода.

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

**Применимость.**

* у абстракции есть два аспекта, один из которых зависит от другого. Инкапсуляция этих аспектов в разные объекты позволяет изменять и повторно использовать их независимо;
* при модификации одного объекта требуется изменить другие, и вы не знаете, сколько именно объектов нужно изменить;
* один объект должен оповещать других, не делая предположений об уведомляемых объектах. Другими словами, объекты не должны быть тесно связаны между собой

**Структура**

****

**Участники**

* **Subject –** субъект:
  + - Располагает информацией о своих наблюдателях. За субъектом может «следить» любое число наблюдателей;
    - Представляет интерфейс для присоединения и отделения наблюдателей;
  + **Observer –** наблюдатель:
    - Определяет интерфейс обновления для объектов, которые должны уведомляться об изменении субъекта;
  + **ConcreteSubject –** конкретный субъект:
    - Сохраняет состояние, представляющее интерес для конкретного наблюдателя *ConcreteObserver*;
    - Посылает информацию своим наблюдателям, когда происходит изменение;
  + **ConcreteObserver –** конкретный наблюдатель:
    - Хранит ссылку на объект класса o *ConcreteSubject;*
    - Сохраняет данные, которые должны быть согласованы с данными субъекта;
    - Реализует интерфейс обновления, определенный в классе *Observer*, чтобы поддерживать согласованность с субъектом.

Т.е. (описание связей и простыми словами написать)

<https://www.youtube.com/watch?v=m71Da-SSe0I>

**Шаги реализации**

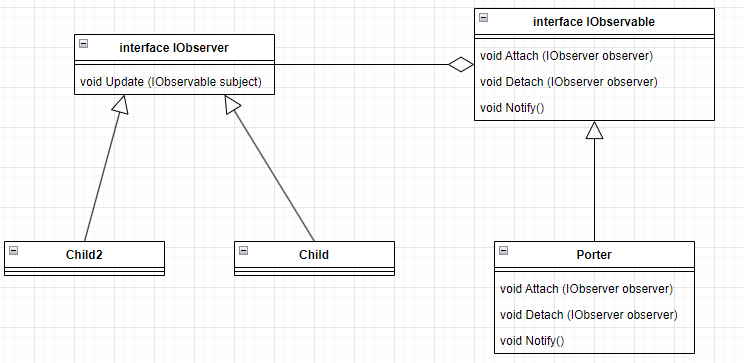
1. Разбейте вашу функциональность на две части: независимое ядро и опциональные зависимые части. Независимое ядро станет издателем. Зависимые части станут подписчиками.
2. Создайте интерфейс подписчиков. Обычно в нём достаточно определить единственный метод оповещения.
3. Создайте интерфейс издателей 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.

1. Создайте классы конкретных издателей. Реализуйте их так, чтобы после каждого изменения состояния они отправляли оповещения всем своим подписчикам.
2. 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.

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



* **IObservable –** субъект:
  + - Располагает информацией о своих наблюдателях. За субъектом может «следить» любое число наблюдателей;
    - Представляет интерфейс для присоединения и отделения наблюдателей;
  + **IObserver –** наблюдатель:
    - Определяет интерфейс обновления для объектов, которые должны уведомляться об изменении субъекта;
  + **Porter –** конкретный субъект:
    - Сохраняет состояние, представляющее интерес для конкретного наблюдателя *Child*;
    - Посылает информацию своим наблюдателям, когда происходит изменение;
  + **Child –** конкретный наблюдатель:
    - Хранит ссылку на объект класса o *Porter;*
    - Сохраняет данные, которые должны быть согласованы с данными субъекта;
    - Реализует интерфейс обновления, определенный в классе *IObserver*, чтобы поддерживать согласованность с субъектом.

Observalbe агрегирующий Observer

Child, Child производные классы Observer

/// <summary>

/// Шаблон интерфейса, главный(подписчики)

/// </summary>

public interface IObserver

{

// Получает обновление от вахтера

void Update(IObservable subject);

}

/// <summary>

/// Шаблон интерфейса, реализующий интерфейс (контроллер)

/// </summary>

public interface IObservable

{

// Присоединяет учеников к вахтеру.

void Attach(IObserver observer);

// Отсоединяет учеников от вахтеру.

void Detach(IObserver observer);

// Уведомляет всех наблюдателей о событии.

void Notify();

}

/// <summary>

/// Класс, реализуюший Observable

/// </summary>

public class Porter : IObservable

{

//// Для удобства в этой переменной хранится состояние Издателя,

//// необходимое всем подписчикам.

public int state;

// Список подписчиков. В реальной жизни список подписчиков может

// храниться в более подробном виде (классифицируется по типу события и

// т.д.)

private List<IObserver> \_observers = new List<IObserver>();

// Методы управления подпиской.

public void Attach(IObserver observer)

{

Console.WriteLine("Пришел в школу"); /// Подписался

\_observers.Add(observer);

}

public void Detach(IObserver observer)

{

if (\_observers.Remove(observer) == true)

{

\_observers.Remove(observer); /// Отписался

Console.WriteLine("Вышел из школы");

}

else

{

Console.WriteLine("Наблюдатель отсутствует");

}

}

// Запуск обновления в каждом подписчике.

public void Notify()

{

Console.WriteLine("Звонок"); /// Получил уведомление

foreach (var observer in \_observers)

{

observer.Update(this);

}

}

public void Bell(int p)

{

state = p;

Notify();

}

}

// Конкретные Наблюдатели реагируют на обновления, выпущенные Издателем, к

// которому они прикреплены.

class Child : IObserver

{

public void Update(IObservable subject)

{

if ((subject as Porter).state == 0)

{

Console.WriteLine("Child: Ураа, домой");

}

else

{

Console.WriteLine("Child: Пора на урок");

}

}

}

class Child2 : IObserver

{

public void Update(IObservable subject)

{

if ((subject as Porter).state == 0)

{

Console.WriteLine("Child2: Перемена!!");

}

else

{

Console.WriteLine("Child2: Эх");

}

}

}

class Program

{

static void Main(string[] args)

{

// Клиентский код.

var Porter = new Porter();

var Child = new Child();

var Child2 = new Child2();

Porter.Bell(0);

Porter.Attach(Child);

Porter.Attach(Child);

Porter.Bell(1);

Porter.Bell(0);

Porter.Bell(1);

Porter.Detach(Child);

Porter.Bell(0);

Porter.Detach(Child);

}

}