# Паттерн Наблюдатель (Observer)

**Назначение**: 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.

**Другими словами**: наблюдатель уведомляет все заинтересованные стороны о произошедшем событии или об изменении своего состояния.

## Общие сведения

Существует два способа общения между двумя программными элементами. Компонент1 может обратиться к Компоненту2 для получения некоторых данных или выполнения некоторой операции. В этом случае Компонент 2выполняет некоторую работу, когда его об этом попросят.

В некоторых случаях Компонент 2 является активным, содержит свой собственный поток исполнения или каким-то другим способом следит за своим собственным состоянием. В этом случае, Компонент 2 может уведомить Компонент 1 о некотором событии.

Первая модель взаимодействия называется pull-моделью, а вторая - push-моделью.

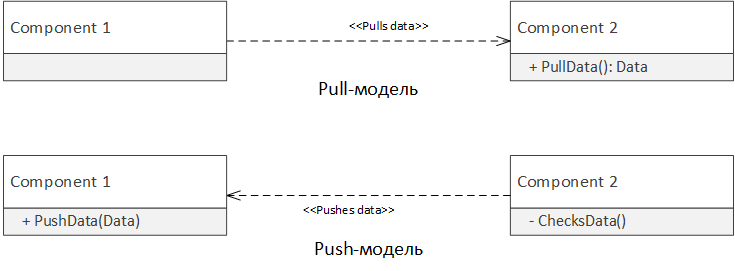


Рисунок 5.1 - Pull и Push модели взаимодействия

Push-модель взаимодействия появилась задолго до распространения ООП и паттернов проектирования. В мире структурного программирования push-модель реализуется с помощью методов обратного вызова (callbacks). В мире функционального программирования, которое тоже появилось задолго до ООП, push-модель представлена в виде реактивной модели программирования. А в мире ООП, push-модель реализуется с помощью паттерна Наблюдатель.

## Мотивация

TODO: формулировка ниже изменится в книге. Текущая для статьи! TODO: сейчас есть небольшая рассогласованность в подаче материала. В разделе Мотивация не используется конкретный пример!

В системе по импорту лог-файлов, приложение может следить за набором файлов и загружать их содержимое по мере появления в логах новых записей. За чтение записей из лог-файла отвечает LogFileReader.

Возможны две реализации данного класса. Класс LogFileReader может быть пассивным и вычитывать новый фрагмент лог-файла при вызове метода Read (pull-модель). Или же класс LogFileReader может быть активным и сам вычитывать новые фрагменты лог-файла по мере его обновления (push-модель).

Возможны два варианта реализации push-модели: класс LogFileReader может знать о классе LogSaver, который будет обрабатывать и сохранять новые фрагменты лог-файла в некотором хранилище. Или же LogFileReader может быть наблюдаемым и уведомлять любых заинтересованных подписчиков о прочитанных новых фрагментах, например, с помощью событий.

В первом случае мы получаем жесткую связь между классами LogFileReader и LogSaver, а во втором случае мы получаем слабосвязанный дизайн, в котором процесс чтения и сохранения логов может развиваться независимо.

**Мотивация использования паттерна Наблюдатель:** уменьшить связанность класса с его зависимостями, путем развязывания инициатора некоторого события от его обработчиков.

## Классическая диаграмма классов паттерна Наблюдатель

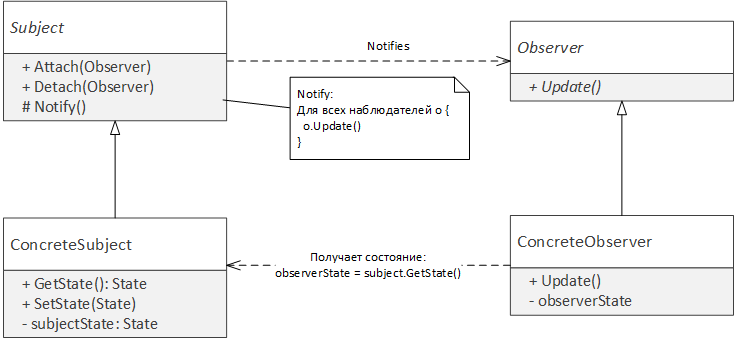


Рисунок 5.2 - Классическая диаграмма паттерна наблюдатель

**Участники**

* Observer определяет интерфейс наблюдателя.
* Subject (наблюдаемый объект) определяет методы подключения и отключения наблюдателей.
* ConcreteObserver реализует интерфейс наблюдателя.
* ConcreteSubject конкретный тип наблюдаемого объекта.

На платформе .NET практически невозможно встретить классическую реализацию паттерна Наблюдатель.

Наличие делегатов и событий, а также интерфейсов IObserver/IObservable и библиотеки реактивных расширений, делает использование наследования излишним.

**ПРИМЕЧАНИЕ**  
Язык C# и платформа .NET очень многое позаимствовала из Java, но наблюдатели были реализованы по-своему. В Java используется каноническая версия паттерна: базовый класс object содержит методnotify/notifyAll и выступает одновременно в роли базового класса наблюдаемых объектов (BaseSubject) и базового класса наблюдателей (BaseObserver).

## Варианты реализации

Существует несколько вариантов реализации паттерна Наблюдатель на платформе .NET:

* С помощью делегатов (методов обратного вызова).
* С помощью событий (events).
* С помощью специализированных интерфейсов-наблюдателей.
* С помощью интерфейсов IObserver/IObservable.

### Методы обратного вызова

Самая простая форма Наблюдателя на платформе .NET реализуется с помощью делегатов. Для этого достаточно, чтобы класс потребовал делегат в аргументах конструктора и уведомлял вызывающий код с его помощью:

public class LogFileReader : IDisposable

{

private readonly string \_logFileName;

private readonly Action<string> \_logEntrySubscriber;

private readonly static Timespan CheckFileInterval = Timespan.FromSeconds(5);

private readonly Timer \_timer;

public LogFileReader(string logFileName, Action<string> logEntrySubscriber)

{

Contract.Requries(File.Exists(logFileName));

\_logEntrySubscriber = logEntrySubscriber;

\_timer = new Timer(() => CheckFile(), CheckFileInterval, CheckFileInterval);

}

public void Dispose()

{

\_timer.Dispose();

}

private void CheckFile()

{

foreach(var logEntry in ReadNewLogEntries())

{

\_logEntrySubscriber(logEntry);

}

}

private IEnumerable<string> ReadNewLogEntries()

{

// ...

// Читаем новые записи из файла,

// которые появились с момента последнего чтения

}

}

Листинг 5.1 – Реализация класса LogFileReader на основе делегатов

Это самая простая и ограниченная реализация паттерна Наблюдатель. Делегаты на платформе .NET могут содержать цепочку вызовов, но подобный код моделирует отношение 1 к 1 между классом LogFileReader и его наблюдателем. Зачастую, этого вполне достаточно, к тому же, это позволяет гарантировать наличие наблюдателя, что может быть полезным, когда наблюдаемому объекту потребуется получить назад некоторые результаты.

### События

Паттерн Наблюдатель является настолько распространенным, что многие языки программирования поддерживают его из коробки. В языках платформы .NET паттерн Наблюдатель (или паттерн Издатель-Подписчик) реализуется с помощью событий.

События представляют собой умную оболочку над делегатами, которая позволяет клиентам лишь подписываться или отписываться от событий, а владельцу события еще и инициировать событие для уведомления всех подписчиков.

public class LogEntryEventArgs : EventArgs

{

public string LogEntry {get; internal set;}

}

public class LogFileReader : IDisposable

{

private readonly string \_logFileName;

public LogFileReader(string logFileName)

{

//...

}

public event EventHandler<LogEntryEventArgs> OnNewLogEntry;

private void CheckFile()

{

foreach(var logEntry in ReadNewLogEntries())

{

RaiseNewLogEntry(logEntry);

}

}

private void RaiseNewLogEntry(string logEntry)

{

var handler = OnNewLogEntry;

if (handler != null)

handler(this, new LogEntryEventArgs(logEntry));

}

}

Листинг 5.2 – Реализация класса LogFileReader на основе событий

Данный вариант очень похож на вариант с делегатами с одной важной разницей. Интерфейс класса LogFileReader позволяет подписаться на событие получения новых записей лог-файлов любому числу подписчиков. При этом нет гарантии, что эти подписчики вообще будут.

### Строго типизированный наблюдатель

В некоторых случаях, группу событий или делегатов удобно объединить в одном интерфейсе. Иногда это говорит о наличии скрытой абстракции, но иногда бывает просто удобно оперировать интерфейсом, а не набором событий.

public interface ILogFileReaderObserver

{

void NewLogEntry(string logEntry);

void FileWasRolled(string oldLogFile, string newLogFile);

}

Public class LogFileReader : IDisposable

{

private readonly ILogFileReaderObserver \_observer;

private readonly string \_logFileName;

public LogFileReader(string logFileName, ILogFileReaderObserver observer)

{

\_logFileName = logFileName;

\_observer = observer;

}

// Добавлена дополнительная логика, которая определяет

// что логгер перестал писать в текущий лог файл и переключился

// на новый.

private void DetectThatNewFileWasCreated()

{

// Метод вызывается по таймеру

if (NewLogFileWasCreated())

\_observer.FileWasRolled(\_logFileName, GetNewLogFileName());

}

}

Листинг

Данный вариант очень похож на классическую версию паттерна Наблюдатель, с той лишь разницей, что обычно наблюдатель является единственным. Если нужно множество "подписчиков", то проще воспользоваться версией с событиями.

### IObserver/IObservable

Все перечисленные выше варианты реализации паттерна Наблюдатель содержат одно ограничение: они плохо композируются. Над событиями или делегатами невозможно выполнять операции, доступные над последовательностями. В случае pull-based реализации, клиенты класса LogFileReader могут использовать LINQ (Language Integrated Query) для манипулированием прочитанными сообщениями.

Например, с помощью LINQ довольно легко обрабатывать критические сообщения особым образом, сохраняя их пачками по 10 элементов:

// Сохраняем только критические ошибки группами по 10 элементов

var messages =

logFileReader.Read()

.Select(ParseLogMessage)

.Where(m => m.Severity == Critical);

foreach(var criticalMessages in messages.Buffer(10))

{

BulkSaveMessages(criticalMessages);

}

Метод расширения Buffer отсутствует в классе Enumerable, но его довольно легко реализовать самостоятельно.

С 4-й версии в .NET Framework появилась пара интерфейсов IObserver/IObservable с набором методов расширений, известных под названием "Реактивных расширений" (Rx, Reactive Extensions). Интерфейс IObservable моделирует "реактивные последовательности" и позволяют работать с наблюдаемыми последовательностями через привычный LINQ-синтаксис.

Реактивные расширения представляют собой push-последовательности и для их использования класс LogFileReader нужно изменить следующим образом:

public class LogFileReader : IDisposable

{

private readonly string \_fileName = fileName;

private readonly Subject<string> \_logEntriesSubject = new Subject<string>();

public LogFileReader(string fileName)

{

\_fileName = fileName;

}

public void Dispose()

{

// Закрываем файл

CloseFile();

// Уведомляем подписчиков, что событий больше не будет

\_logEntriesSubject.OnComplete();

}

public IObservable<string> NewMessages

{

get { return \_logEntriesSubject; }

}

private void CheckFile()

{

foreach(var logEntry in ReadNewLogEntries())

{

\_logEntriesSubject.OnNext(logEntry);

}

}

}

Что позволяет использовать с наблюдателями привычный LINQ синтаксис:

var messagesObservable =

logFileReader.NewMessages

.Select(ParseLogMessages)

.Where(m => m.Severity == Critical);

messagesObservable

.Buffer(10)

.Subscribe(IList<LogEntry> criticalMessages => BulkSaveMessages(criticalMessages));

**ПРИМЕЧАНИЕ**

Реактивные последовательности являются своеобразной смесью паттернов "Итератор" и "Наблюдатель". IObservable можно рассматривать, как "вывернутые наизнанку" последовательности, когда процессом итерирования управляет не вызывающая сторона, как в случае с IEnumerable, а сама последовательность. Стандартные последовательности являются pull-based, поскольку процессом получения новых элементов управляет вызывающая сторона, а реактивные последовательности - push-based, и наблюдатели получают новые элементы по мере их поступления. Подробнее паттерн "Итератор" рассмотрен в предыдущей главе. (TODO: дать ссылку для публикации в блог!)

Интерфейсы IObserver/IObservable не являются наблюдателями общего назначения и предполагают определенный протокол между наблюдаемым объектом и его подписчиками. Интерфейс IObservableпредполагает, что однородные события будут периодически повторяться. Наблюдаемый объект может уведомить о новом событии (OnNext), о том, что в процессе события произошла ошибка (OnError), или о том, что цепочка событий завершена (OnComplete). Нет особого смысла использовать IObservable для уведомления о переконфигурации приложения, поскольку композиция подобных событий вряд ли возможна.

**ПРИМЕЧАНИЕ**  
Подробное описание реактивных расширений выходит за рамки данной книги. Для более подробного знакомства подойдет [Introduction to Rx](http://www.introtorx.com/content/v1.0.10621.0/01_WhyRx.html) Ли Кэмпбелла.

## Обсуждение паттерна "Наблюдатель"

### Выбор варианта реализации Наблюдателя

Проще всего дело обстоит с IObserver/IObservable. Это вариант подходит, когда события возникают периодически, и их можно рассматривать в виде push-based последовательности, над которой удобно производить трансформации с помощью LINQ-запросов. Поток сетевых сообщений от клиента или сервера, координаты устройства, все это подходящие задачи для использования реактивных последовательностей.

К тому же, наличие классов Subject<T> и специальных методов-адаптеров (Observable.FromEvent), позволяет легко получить реактивную последовательность из других реализаций наблюдателей. Выбор между остальными тремя реализациями более интересный.

#### Делегаты

Реализация паттерна Наблюдатель на основе методов обратного вызова пришла из мира структурного программирования и не похожа на классическую реализацию из "банды четырех". Тем не менее, эта реализация весьма распространена и успешно решает задачи данного паттерна: получить слабосвязанный дизайн путем разделения инициатора события от его обработчиков.

Данная реализация моделирует отношение 1 к 1 между наблюдаемым объектом и наблюдателем. Обязательное наличие наблюдателя позволяет установить двустороннюю связь: наблюдаемый объект может не только уведомлять об изменении своего состояния, но и требовать некоторый результат. Результатом может быть объект класса Task, поскольку обработка события может быть длительной, или некоторое значение, требуемое наблюдаемому объекту. Это может быть внешняя валидация, взаимодействие с пользователем или что угодно еще.

**ПРИМЕЧАНИЕ**  
Делегаты могут использоваться для реализации Наблюдателя или Стратегии.

*Не уверен, что это нужно!!*

**Когда использовать?**

* Наблюдатель должен быть обязательно.
* Наблюдаемый объект не просто уведомляет наблюдателя, но и ожидает некоторый результат.

**Когда не использовать?**

* Когда число делегатов начинает расти и передавать их все через аргументы конструктора становится неудобным. В этом случае лучше использовать именованную зависимость или выделить интерфейс наблюдателя.
* Для повторно используемых компонентов. В этом случае лучше использовать события.
* Когда через делегат передается поток событий, которым будет удобнее манипулировать с помощью IObservable.

#### События

События являются самым распространенным вариантом реализации паттерна Наблюдатель на платформе .NET. С их помощью происходит уведомление о событиях пользовательского интерфейса, приеме данных по сети и т.п. Главная особенность событий в том, что класс не может гарантировать наличие наблюдателей-подписчиков, а значит не может потребовать от них никаких результатов. В некоторых случаях используются изменяемые аргументы, например, для отмены некоторого действия (в событиях Form.OnClosing или TaskScheduler.UnobservedException), но это уже примеры паттерна "Цепочка ответственностей", а не наблюдателя.

**Когда использовать?**

* Для повторно используемых компонентов.
* Для уведомления множества наблюдателей, без ожидания от них каких-либо ответных действий.
* Для реализации pull-модели получения данных наблюдателем.

**Когда не использовать?**

* Когда наблюдаемому объекту нужно получить от наблюдателей некоторый результат.

**Наблюдатель в виде специализированного интерфейса**

Интерфейс наблюдателя, обычно, является первым этапом для выделения именованной зависимости. На некотором этапе разработки, становится понятно, что у класса должна быть зависимость, но с именем пока определиться сложно. Поэтому, как в случае с классом LogFileReader, выделяется интерфейс наблюдателя (ILogFileReaderObserver), который затем может трансформироваться обратно к событиям, или же переродиться в полноценную зависимость (например, в ILogFileProcessor).

**Когда использовать?**

* В качестве врЕменной именованной зависимости для группировки набора событий.

**Когда не использовать?**

* В открытом API: в повторно используемом коде или на стыке модулей.

### Сколько информации передавать наблюдателю

Существует две модели передачи нужной информации наблюдателям: push и pull (\*). В случае push-модели, наблюдаемый объект передает всю нужную информацию о произошедшем событии в аргументах: в свойствах объекта EventArgs, в аргументах делегата или метода. В случае pull-модели, наблюдаемый объект лишь уведомляет о произошедшем событии, а всю нужную информацию для его обработки наблюдатель самостоятельно получает у наблюдаемого объекта.

(\*) СНОСКА: Push/pull-модель взаимодействия в этом обсуждении встречается дважды. Паттерн Наблюдатель сам по себе представляет собой push-модель взаимодействия между объектами, поскольку наблюдаемый объект самостоятельно "проталкивает" информацию о произошедшем событии. Но затем возникает вопрос о количестве передаваемой информации. И здесь push/pull-модели вступают в игру по новой, но уже в ином контексте: в контексте количества передаваемой информации.

Хорошим примером компромиссного решения между push и pull вариантами реализации наблюдателей являются события в .NET.

Стандартное событие в .NET объявляется с помощью обобщенного делегета EventHandler<T>, который содержит два аргумента: object sender и TEventArgs args:

public class LogEntryEventArgs : EventArgs

{

public string LogEntry {get; internal set;}

}

public class LogFileReader : IDisposable

{

private readonly string \_logFileName;

public string LogFileName { get { return \_logFileName; } }

public event EventHandler<LogEntryEventArgs> OnNewLogEntry;

// Остальные методы пропущены ...

}

// Код наблюдателя

public class LogForwarder

{

public LogForwarder(LogFileReader logFileReader)

{

logFileReader.OnNewLogEntry += HandleNewLogEntry;

}

private void HandleNewLogEntry(object sender, LogEntryEventArgs ea)

{

var logEntry = ea.LogEntry;

var logFile = ((LogFileReader)sender).LogFileName;

// Обрабатываем logEntry с учетом имени файла

}

}

**ПРИМЕЧАНИЕ**  
Технически, никто не запрещает использовать в качестве событий любые другие делегаты, а не толькоEventHandler. Но стандарты кодирования платформы .NET предполагают использовать именно EventHandler<T>.

В данном случае наблюдаемый объект "выталкивает" (push) наблюдателям лишь минимально необходимую информацию, позволяя им самостоятельно "вытянуть" (pull) дополнительные данные, в случае необходимости.

#### Рекомендации по выбору модели взаимодействия

Компромисс между push и pull-моделями взаимодействия представляет собой классическую проблему выбора между простым и гибким решением. Push-модель проще, поскольку вся информация, требуемая для обработки события передается наблюдателю в виде аргументов. Pull-модель гибче, поскольку позволяет передавать наблюдателям лишь минимальный объем информации, и дает возможность наблюдателям самим решать, что им нужно для обработки события.

На практике pull-модель взаимодействия бывает полезной в случае пользовательского интерфейса, когда UI-компонент не знает, что пользователю может понадобиться для обработки определенного события. В случае бизнес-логики, push-модель является более предпочтительной, поскольку простота решения в этом случае полезнее гибкости.

### Наблюдатели и утечки памяти

Долгоживущие наблюдаемые объекты являются наиболее распространенной причиной утечки памяти в .NET-приложениях. Поскольку наблюдаемый объект содержит неявную ссылку на все наблюдатели, то пока он жив, будут жить и они:

public class Singleton

{

private static readonly Singleton \_instance = new Singleton();

public static Singleton Instance { get { return \_instance; } }

public event EventHandler Event;

}

class MemoryLeak

{

public MemoryLeak()

{

Singleton.Instance.Event += (s, e) => Console.WriteLine("Hello, Memory Leak!");

}

}

В этом случае, создание объекта MemoryLeak гарантированно приведет к утечке памяти, поскольку вновь созданный экземпляр навсегда останется в списке подписчиков события Singleton.MyEvent.

Существует несколько решений данной проблемы.

* Во-первых, стоит по возможности избегать долгоживущих объектов вообще, и долгоживущих объектов с событиями, в особенности.
* Во-вторых, наблюдатели могут реализовать интерфейс IDisposable и отписываться от событий в методеDispose.
* В-третьих, можно воспользоваться слабыми событиями (Weak Event Pattern). Специальной реализацией событий, в которой используются слабые ссылки (Weak References) для управления наблюдателями. В этом случае "слабое событие" не будет являться корневой ссылкой и не будет препятствовать сборке мусора, когда на наблюдатель не останется других ссылок.

### Применимость

Наблюдатель является универсальным механизмом уменьшения связанности в приложении.

При проектировании некоторого класса у разработчика всегда есть несколько вариантов реализации. Класс А может знать о существовании класса Б и уведомлять его о произошедшем событии. Или же класс А может быть наблюдаемым и уведомлять о некотором событии всех заинтересованных подписчиков.

Использование наблюдателей уменьшает связанность между классами/модулями и упрощает повторное использование. Не менее важно и то, что наблюдатель четко показывает "выходной интерфейс" класса (что нужно классу для его успешной работы) и позволяет проще думать о нем в изоляции.

Благодаря слабой связанности наблюдаемый объект и наблюдатели могут располагаться на разных уровнях абстракции или слоях приложения. Например, слой пользовательского интерфейса знает о модели, но модель не должна ничего знать о пользовательском интерфейсе. Косвенная связь моделей с верхним уровнем реализуется с помощью наблюдателей: модель уведомляет всех подписчиков об изменении состояния, а пользовательский интерфейс обновляет свое состояние или другим способом реагирует на события.

Наблюдатели очень часто используются в качестве составных частей более сложных паттернов. Семейство паттернов MVx (Model View Controller/Presenter/ViewModel) развязывает модель от представления именно с помощью наблюдателей. Паттерн Посредник очень часто реализуется с помощью Наблюдателя и связывает две независимые части системы воедино.

**ПРИМЕЧАНИЕ**  
Подробнее о паттернах MVx и слоях приложения см. в главе "Принцип инверсии зависимостей" части 4.

## Примеры в .NET

**Наблюдатели в форме событий.** В .NET Framework насчитываются тысячи классов, содержащих события.

**Наблюдатели в форме делегатов.** Наблюдатели в форме делегатов часто используются в качестве методов обратного вызова для выполнения дополнительной инициализации ([AppDomainSetup.AppDomainInitializer](http://msdn.microsoft.com/en-us/library/system.appdomainsetup.appdomaininitializer(v=vs.110).aspx), [HttpConfiguration.Initializer](http://msdn.microsoft.com/en-us/library/system.web.http.httpconfiguration.initializer(v=vs.118).aspx)), или в качестве точек расширения (фильтры и селекторы в WPF/Windows Forms/WCF).

**Наблюдатели в форме интерфейсов** Хорошим примером именованного наблюдателя является API по работе с [Event Hub](http://msdn.microsoft.com/en-us/library/azure/dn789972.aspx) - масштабируемой системой обмена сообщений. Интерфейс IEventProcessor содержит методы CloseAsync, OpenAsync и ProcessEventsAsync.

## Дополнительные ссылки

* [Паттерн посредник](http://sergeyteplyakov.blogspot.com/2014/03/mediator-design-pattern.html)
* [Паттерн итератор](http://sergeyteplyakov.blogspot.com/2014/05/iterator-pattern.html)
* [DI vs. DIP vs. IOC](http://sergeyteplyakov.blogspot.com/2014/11/di-vs-dip-vs-ioc.html)