# Паттерн Стратегия (Strategy)

**Назначение:** определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми. Стратегия позволяет изменять алгоритмы независимо от клиентов, которые ими пользуются.

**Другими словами:** стратегия инкапсулирует некоторое поведение с возможностью его подмены.

## Мотивация

Паттерн стратегия является настолько распространенным и общепринятым, что многие его используют постоянно, даже не задумываясь о том, что это хитроумный паттерн проектирования, расписанный когда-то бандой четырех (\*).

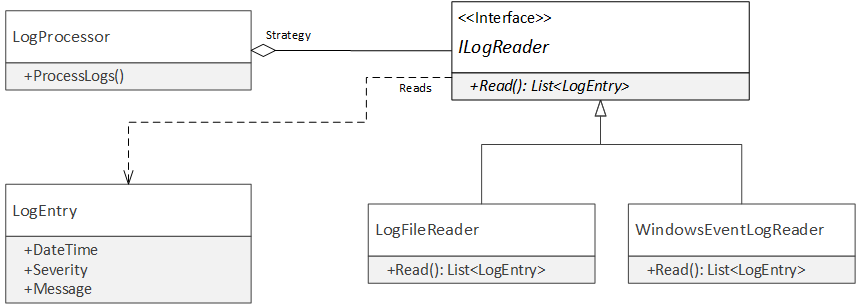
(\*) СНОСКА: Речь о знаменитой книге о паттернах проектирования, с которой началось популяризация паттернов.

Каждый второй раз, когда мы пользуемся наследованием, мы используем стратегию; каждый раз, когда мы абстрагируемся от некоторого процесса, поведения или алгоритма за базовым классом или интерфейсом – мы используем стратегию. Сортировка, анализ данных, валидация, разбор данных, сериализация, кодирование/декодирование, получение конфигурации, все эти концепции могут и должны быть выражены в виде стратегий или политик (policy).

Стратегия является фундаментальным паттерном, поскольку она проявляется в большинстве других классических паттернов проектирования, которые поддерживают специализацию за счет наследования. Абстрактная фабрика – это стратегия создания семейства объектов; фабричный метод – стратегия создания одного объекта; строитель – стратегия построения объекта; итератор – стратегия перебора элементов и т.д. (\*)

(\*) Подробнее все эти паттерны проектирования будут рассмотрены в последующих главах.

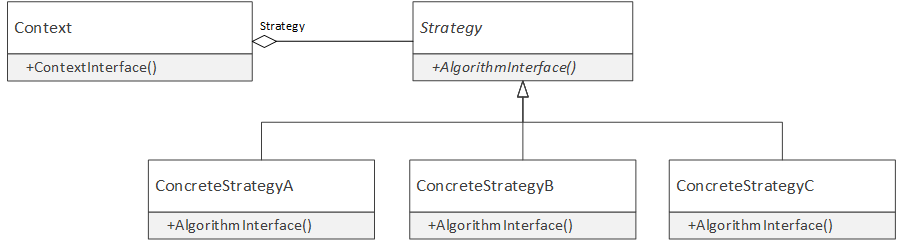
Давайте в качестве примера рассмотрим задачу импорта лог-файлов для последующего полнотекстового поиска (full text search). Главной задачей данного приложения является чтение лог-файлов из различных источников, приведение их к некоторому каноническому виду и сохранение их в некотором хранилище, например в Elasicsearch, или в SQL Server.

  
Рисунок 1.1 – Диаграмма классов импорта логов

LogProcessor отвечает за импорт лог-файлов и должен работать с любой разновидностью логов: файлами (LogFileReader), логами Windows (WindowsEventLogReader), и т.д. Для этого процесс чтения логов выделяется в виде интерфейса или базового класса ILogReader, а класс LogProcessor знает лишь о нем, и не зависет от конкретной реализации.

**Мотивация использования паттерна Стратегия:** выделение поведения или алгоритма с возможностью его замены во время исполнения.

## Классическая диаграмма классов паттерна Стратегия

  
Рисунок 1.2 - Диаграмма классов паттерна Стратегия

**Участники**

* Strategy (ILogReader) определяет интерфейс алгоритма.
* Context (LogProcessor) является клиентом стратегии.
* ConcreteStrategyA, ConcreteStrategyB (LogFileReader, WindowsEventLogReader) являются конкретными реализации стратегии.

Обратите внимание, что классический паттерн Стратегии весьма абстрактен.

* Паттерн Стратегия не определяет, как стратегия получит данные, необходимые для выполнения своей работы. Они могут передаваться в аргументах метода AlgorithmInterface, или стратегия может получать ссылку на сам контекст и получать требуемые данные самостоятельно.
* Паттерн Стратегия не определяет, каким образом контекст получает экземпляр стратегии. Контекст может получать ее в аргументах конструктора, через метод, свойство или получать ее у третьей стороны.

## Варианты реализации в .NET

В общем случае паттерн Стратегия не определяет, какое количество операций будет у "выделенного поведения или алгоритма". Это может быть одна операция (метод Sort интерфейса ISortable), а может быть и семейство операций (Encode/Decode интерфейса IMessageProcessor).

При этом, если операция лишь одна, то вместо выделения и передачи интерфейса в современных .NET приложениях очень часто используются делегаты. Так, в нашем случае, вместо передачи интерфейса ILogReader класс LogProcessor мог бы принимать делегат вида: Func<List<LogEntry>>, который соответствует сигнатуре единственного метода стратегии:

class LogProcessor

{

private readonly Func<List<LogEntry>> \_logImporter;

public LogProcessor(Func<List<LogEntry>> logImporter)

{

\_logImporter = logImporter;

}

public void ProcessLogs()

{

foreach(var logEntry in \_logImporter.Invoke())

{

SaveLogEntry(logEntry);

}

}

// Остальные методы пропущены...

}

Листинг 1.1 – Класс LogProcessor

Использование "функциональных стратегий" является единственной платформенно-зависимой особенностью паттерна Стратегия на платформе .NET. Да и то, эта особенность обусловлена не столько самой платформой, сколько возрастающей популярностью техник функционального программирования.

**ПРИМЕЧАНИЕ**  
В некоторых командах возникают попытки обобщить паттерны проектирования и использовать их повторно в виде библиотечного кода. В результате появляются интерфейсы IStrategy и IContext, вокруг которых строится решение в коде приложения. Есть лишь несколько паттернов проектирования, повторное использование которых возможно на уровне библиотеки: Синглтон, Наблюдатель, Команда. В общем же случае, попытка обобщить паттерны приводит к переусложненным решением и показывает непонимание фундаментальных принципов паттернов проектирования.

## Обсуждение паттерна Стратегия

По определению, применение стратегии обусловлено двумя причинами: (1) необходимостью инкапсуляции поведения или алгоритма и (2) необходимостью замены поведения или алгоритма во время исполнения. Любой нормально спроектированный класс уже инкапсулирует в себе поведение или алгоритм, но не любой класс с некоторым поведением является или должен быть стратегией. **Стратегия нужна тогда, когда нужно не просто спрятать алгоритм, а когда важно иметь возможность заменить его во время исполнения!**

Другими словами, стратегия обеспечивает точку расширения системы в определенной плоскости: класс-контекст принимает экземпляр стратегии и не знает, какой вариант стратегии он собирается использовать.

### Выделять интерфейс или нет?

**ПРИМЕЧАНИЕ**  
Выделение интерфейсов является достаточно острым вопросом в современной разработке ПО и является актуальным не только в контексте стратегий. Поэтому все нижележащие размышления применимы к любым иерархиям наследования.

Сейчас существует два противоположных лагеря в мире объектно-ориентированного программирования: ярые сторонники выделения интерфейсов и ярые противники этого явления. Когда возникает вопрос о необходимости выделения интерфейса и добавления наследования мне нравится думать об этом, как о необходимости выделения стратегии. Это не всегда точно, но может быть хорошей лакмусовой бумажкой.

Нужно ли выделять интерфейс IValidator для проверки корректности ввода пользователя? Нужен ли нам интерфейсы IFactory или IAbstractFactory, или нам подойдет один конкретный класс? Ответы на эти вопрос зависят от того, нужна ли нам "стратегия" (или политика) валидации или создания объектов. Хотим ли мы заменять эту стратегию во время исполнения или мы можем использовать конкретную реализацию и внести в нее изменение в случае необходимости?

У выделения интерфейса и передачи его в качестве зависимости есть еще несколько особенностей.

Передача интерфейса ILogReader классу LogProcessor повышает гибкость, но и повышает сложность. Теперь, клиентам класса LogProcessor нужно решить, какую реализацию использовать, или перекладывать эту ответственность на свой вызывающий код.

Важно понимать, нужен ли дополнительный уровень абстракции именно сейчас. Может быть на текущем этапе достаточно использовать напрямую класс LogFileImporter, а выделить стратегию импорта тогда, когда в этом действительно появится необходимость.

### Интерфейс vs. Делегат

Поскольку некоторые стратегии содержат лишь один метод, то очень часто вместо классической стратегии на основе наследования, можно использовать стратегию на основе делегатов. Иногда эти подходы совмещаются, что позволяет использовать наиболее удобный вариант.

Классическим примером такой ситуации является "стратегия сортировки", представленная интерфейсами IComparable<T> и делегатом Comparison<T>:

class Employee

{

public int Id { get; set; }

public string Name { get; set; }

public override string ToString()

{

returnstring.Format("Id = {0}, Name = {1}", Id, Name);

}

}

class EmployeeByIdComparer : IComparer<Employee>

{

public int Compare(Employee x, Employee y)

{

return x.Id.CompareTo(y.Id);

}

}

public static void SortLists()

{

var list = new List<Employee>();

// Используем "функтор"

list.Sort(new EmployeeByIdComparer());

// Используем делегат

list.Sort((x, y) => x.Name.CompareTo(y.Name));

}

Листинг 1.2 – Примеры стратегий сортировки

Реализация интерфейса требует больше кода и приводит к переключению контекста при чтении, по сравнению с использованием лямбда-выражений. В случае с методом List.Sort у нас есть оба варианта, но в некоторых случаях классы могут принимать лишь стратегию на основе интерфейса, и не принимать стратегию на основе делегатов, как в случае с классами SortedList или SortedSet:

var comparer = new EmployeeByIdComparer();

// Конструктор принимает IComparable

var set = new SortedSet<Employee>(comparer);

// Нет конструктора, принимающего делегат Comparison<T>

Лисринг 1.3 – Конструирование объекта SortedSet

В этом случае можно сделать небольшой "адаптерный фабричный" класс, который будет принимать делегат Comparison<T> и возвращать интерфейс IComparable<T>:

class ComparerFactory

{

Public static IComparer<T> Create<T>(Comparison<T> comparer)

{

return new DelegateComparer<T>(comparer);

}

private class DelegateComparer<T> : IComparer<T>

{

private readonly Comparison<T> \_comparer;

public DelegateComparer(Comparison<T> comparer)

{

\_comparer = comparer;

}

public int Compare(T x, T y)

{

return \_comparer(x, y);

}

}

}

Листинг 1.4 – Фабричный класс для создания экземпляров IComparer

Теперь можно использовать этот фабричный следующим образом:

var comparer = ComparerFactory.Create<Employee>(

(x, y) => x.Id.CompareTo(x.Id));

var set = new SortedSet<Employee>(comparer);

Листинг 1.5 – Пример использования класса ComparerFactory

**ПРИМЕЧАНИЕ**  
Аналогично, можно пойти еще дальше, и вместо метода "императивного" подхода на основе делегатаComparison<T>, можно получить более "декларативное" решение аналогичное тому, что используется в методе [Enumerable.OrderyBy](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.orderby(v=vs.110).aspx): на основе "селектора" свойств для сравнения.

## Применимость

Применимость стратегии полностью определяется ее назначением: паттерн Стратегия нужно использовать для моделирования семейства алгоритмов и операций, когда есть необходимость замены одного поведения другим во время исполнения.

Не следует использовать стратегию "на всякий случай". Наследование добавляет гибкости, но и приводит к увеличению сложности. Любой класс уже ограничивает своих клиентов от деталей реализации, и позволяет изменять эти детали не затрагивая клиентов. Наличие полиморфизма усложняет чтение кода, а "дырявые абстракции" и нарушения принципа замещения Лисков (\*) существенно усложняет поддержку и сопровождение такого кода.

(\*) Сноска: принцип замещения Лисков будет рассмотрен в четвертой части книги.

Гибкость не бывает бесплатной, поэтому выделять стратегии стоит тогда, когда действительно нужна замена поведения во время исполнения.

## Примеры в .NET Framework

Стратегия является невероятно распространенным паттерном в .NET Framework.

* LINQ (Language Integrated Query) – это набор методов расширения, принимающих "стратегии" фильтрации, получения проекции и т.д. Коллекции принимают стратегии сравнения элементов, а значит любой класс, который принимает IComparer<T> или IEqualityComparer<T> используют паттерн Стратегия.
* WCF просто переполнен стратегиями: IErrorHandler – стратегия обработки коммуникационных ошибок; IChannelInitializer – стратегия инициализации канала; IDispatchMessageFormatter – стратегия форматирования сообщений; MessageFilter – стратегия фильтрации сообщений и т.д. Также обилие стратегий есть в Windows Forms, WPF, ASP.NET и других фреймворках.

Любая библиотека просто переполнена стратегиями, поскольку они представляют собой универсальный механизм расширения требуемого функционала пользователем и адаптации поведения под неизвестные заранее требования.