

## Оглавление

<b>Об авторе</b> .....	15
<b>Кому адресована эта книга</b> .....	16
<b>Как читать эту книгу</b> .....	17
<b>Отзывы</b> .....	18
<b>Благодарности</b> .....	19
<b>От издательства</b> .....	20
<b>Предисловие</b> .....	21
GoF-паттерны на платформе .NET.....	21
Отношение к паттернам проектирования.....	22
Фреймворки паттернов.....	24
Гибкость vs. конкретность.....	25
Для чего нужна еще одна книга о паттернах .....	25

## Часть I. Паттерны поведения

<b>Глава 1. Паттерн «Стратегия» (Strategy)</b> .....	28
Мотивация .....	28
Варианты реализации в .NET .....	30

Обсуждение паттерна «Стратегия» .....	31
Выделять интерфейс или нет .....	32
Интерфейс vs. делегат .....	32
Применимость .....	35
Примеры в .NET Framework .....	36
<b>Глава 2. Паттерн «Шаблонный метод» (Template Method) .....</b>	<b>37</b>
Мотивация .....	37
Варианты реализации в .NET .....	39
Локальный шаблонный метод на основе делегатов .....	39
Шаблонный метод на основе методов расширения .....	42
Обсуждение паттерна «Шаблонный метод» .....	44
Изменение уровня абстракции .....	44
Стратегия vs. шаблонный метод .....	45
Шаблонный метод и обеспечение тестируемости .....	46
Шаблонный метод и контракты .....	48
Применимость .....	50
Примеры в .NET Framework .....	50
<b>Глава 3. Паттерн «Посредник» (Mediator) .....</b>	<b>57</b>
Мотивация .....	57
Обсуждение паттерна «Посредник» .....	60
Явный и неявный посредник .....	61
Явные и неявные связи .....	62
Тестировать или не тестировать? Вот в чем вопрос! .....	64
Архитектурные посредники .....	65
Применимость .....	66
Когда третий лишний .....	66
Примеры в .NET Framework .....	66
<b>Глава 4. Паттерн «Итератор» (Iterator) .....</b>	<b>68</b>
Мотивация .....	68
Обсуждение .....	70

Особенности итераторов в C#/.NET . . . . .	72
«Ленивость» итераторов . . . . .	76
Использование итераторов в цикле foreach . . . . .	76
Итераторы или генераторы . . . . .	78
Валидность итераторов . . . . .	79
Итераторы и структуры . . . . .	80
Push-based-итераторы . . . . .	80
Применимость . . . . .	81
Примеры в .NET Framework . . . . .	82
<b>Глава 5. Паттерн «Наблюдатель» (Observer) . . . . .</b>	<b>83</b>
Общие сведения . . . . .	83
Мотивация . . . . .	84
Варианты реализации . . . . .	86
Методы обратного вызова . . . . .	86
События . . . . .	87
Строго типизированный наблюдатель . . . . .	89
IObserver/IObservable . . . . .	90
Обсуждение паттерна «Наблюдатель» . . . . .	93
Выбор варианта реализации «Наблюдателя» . . . . .	93
Делегаты . . . . .	93
События . . . . .	94
Наблюдатель в виде специализированного интерфейса . . . . .	94
Сколько информации передавать наблюдателю . . . . .	95
Наблюдатели и утечки памяти . . . . .	97
Применимость . . . . .	98
Примеры в .NET Framework . . . . .	99
<b>Глава 6. Паттерн «Посетитель» (Visitor) . . . . .</b>	<b>100</b>
Мотивация . . . . .	100
Обсуждение . . . . .	105

Функциональная vs. Объектная версия . . . . .	105
Двойная диспетчеризация . . . . .	108
Интерфейс vs. абстрактный класс посетителя . . . . .	109
Применимость . . . . .	110
Примеры в .NET Framework. . . . .	111
<b>Глава 7. Другие паттерны поведения . . . . .</b>	<b>112</b>
Паттерн «Команда» . . . . .	112
Паттерн «Состояние» . . . . .	114
Паттерн «Цепочка обязанностей» . . . . .	116

## Часть II. Порождающие паттерны

<b>Глава 8. Паттерн «Синглтон» (Singleton) . . . . .</b>	<b>122</b>
Мотивация . . . . .	122
Варианты реализации в .NET . . . . .	123
Реализация на основе Lazy of T . . . . .	123
Блокировка с двойной проверкой . . . . .	124
Реализация на основе инициализатора статического поля . . . . .	126
Обсуждение паттерна «Синглтон» . . . . .	129
Singleton vs. Ambient Context . . . . .	129
Singleton vs. Static Class . . . . .	132
Особенности и недостатки. . . . .	132
Применимость: паттерн или антипаттерн . . . . .	134
Примеры в .NET Framework. . . . .	135
Дополнительные ссылки . . . . .	136
<b>Глава 9. Паттерн «Абстрактная фабрика» (Abstract Factory) . . . . .</b>	<b>137</b>
Мотивация . . . . .	138
Обсуждение паттерна «Абстрактная фабрика» . . . . .	141
Проблема курицы и яйца. . . . .	141
Обобщенная абстрактная фабрика . . . . .	142

Применимость паттерна «Абстрактная фабрика» . . . . .	143
Примеры в .NET Framework. . . . .	144
<b>Глава 10. Паттерн «Фабричный метод» (Factory Method) . . . . .</b>	<b>145</b>
Мотивация . . . . .	145
Диаграмма паттерна «Фабричный метод» . . . . .	147
Классическая реализация . . . . .	147
Статический фабричный метод. . . . .	148
Полиморфный фабричный метод. . . . .	148
Варианты реализации . . . . .	149
Использование делегатов в статической фабрике. . . . .	149
Обобщенные фабрики. . . . .	150
Обсуждение паттерна «Фабричный метод» . . . . .	153
Соккрытие наследования . . . . .	153
Устранение наследования . . . . .	154
Использование Func в качестве фабрики . . . . .	156
Конструктор vs. фабричный метод. . . . .	156
Применимость паттерна «Фабричный метод» . . . . .	157
Применимость классического фабричного метода . . . . .	157
Применимость полиморфного фабричного метода. . . . .	157
Применимость статического фабричного метода. . . . .	158
Примеры в .NET Framework. . . . .	158
<b>Глава 11. Паттерн «Строитель» (Builder) . . . . .</b>	<b>160</b>
Мотивация . . . . .	160
Особенности реализации в .NET . . . . .	164
Использование текучего интерфейса. . . . .	164
Методы расширения . . . . .	165
Обсуждение паттерна «Строитель» . . . . .	167
Строго типизированный строитель. . . . .	167
Создание неизменяемых объектов . . . . .	170
Частичная изменяемость. . . . .	171

Применимость .....	173
Примеры в .NET Framework.....	174

## Часть III. Структурные паттерны

<b>Глава 12. Паттерн «Адаптер» (Adapter) .....</b>	<b>188</b>
Мотивация .....	188
Обсуждение паттерна «Адаптер» .....	191
Адаптер классов и объектов .....	191
Адаптивный рефакторинг .....	192
Языковые адаптеры .....	194
Применимость .....	196
Примеры в .NET Framework.....	196
<b>Глава 13. Паттерн «Фасад» (Facade) .....</b>	<b>197</b>
Мотивация .....	197
Обсуждение паттерна «Фасад» .....	199
Инкапсуляция стороннего кода .....	199
Повышение уровня абстракции .....	199
Применимость .....	200
Примеры в .NET Framework.....	200
<b>Глава 14. Паттерн «Декоратор» (Decorator) .....</b>	<b>201</b>
Мотивация .....	201
Обсуждение паттерна «Декоратор» .....	205
Композиция vs. наследование .....	205
Инициализация декораторов .....	207
Недостатки декораторов .....	208
Генерация декораторов .....	208
Применимость .....	209
Примеры в .NET Framework.....	209

<b>Глава 15. Паттерн «Компоновщик» (Composite)</b> .....	214
Мотивация .....	214
Обсуждение паттерна «Компоновщик» .....	218
Применимость .....	219
Примеры в .NET Framework .....	220
<b>Глава 16. Паттерн «Заместитель» (Proxy)</b> .....	221
Мотивация .....	221
Обсуждение паттерна «Заместитель» .....	223
Прозрачный удаленный заместитель .....	224
Заместитель vs. декоратор .....	224
Виртуальный заместитель и Lazy<T> .....	225
Применимость .....	226
Примеры в .NET Framework .....	226

## Часть IV. Принципы проектирования

<b>Глава 17. Принцип единственной обязанности</b> .....	231
Для чего нужен SRP .....	233
Принцип единственной обязанности на практике .....	233
Типичные примеры нарушения SRP .....	241
Выводы .....	242
<b>Глава 18. Принцип «открыт/закрыт»</b> .....	243
Путаница с определениями .....	244
Какую проблему призван решить принцип «открыт/закрыт» .....	247
Принцип «открыт/закрыт» на практике .....	248
Закрытость интерфейсов .....	248
Открытость поведения .....	251
Принцип единственного выбора .....	253

Расширяемость: объектно-ориентированный и функциональный подходы . . . . .	254
Типичные примеры нарушения принципа «открыт/закрыт» . . . . .	258
Выводы . . . . .	258
<b>Глава 19. Принцип подстановки Лисков . . . . .</b>	<b>260</b>
Для чего нужен принцип подстановки Лисков . . . . .	262
Классический пример нарушения: квадраты и прямоугольники . . . . .	263
Принцип подстановки Лисков и контракты . . . . .	265
О сложностях наследования в реальном мире . . . . .	265
Когда наследования бывает слишком мало . . . . .	268
Принцип подстановки Лисков на практике . . . . .	270
Типичные примеры нарушения LSP . . . . .	273
Выводы . . . . .	273
Дополнительные ссылки . . . . .	274
<b>Глава 20. Принцип разделения интерфейсов . . . . .</b>	<b>275</b>
Для чего нужен принцип разделения интерфейса . . . . .	276
SRP vs. ISP . . . . .	278
Принцип разделения интерфейсов на практике . . . . .	279
Типичные примеры нарушения ISP . . . . .	282
Выводы . . . . .	282
<b>Глава 21. Принцип инверсии зависимостей . . . . .</b>	<b>284</b>
Интерфейсы . . . . .	285
Слои . . . . .	286
Наблюдатели . . . . .	288
Для чего нужен принцип инверсии зависимостей . . . . .	291
Остерегайтесь неправильного понимания DIP . . . . .	293
Тестируемость решения vs. подрыв инкапсуляции . . . . .	294
Принцип инверсии зависимостей на практике . . . . .	295
Примеры нарушения принципа инверсии зависимостей . . . . .	300
Выводы . . . . .	300
Дополнительные ссылки . . . . .	304



<b>Глава 22. Размышления о принципах проектирования</b> .....	305
Использование принципов проектирования.....	307
Правильное использование принципов проектирования.....	308
Антипринципы проектирования.....	310
<b>Заключение</b> .....	311
<b>Источники информации</b> .....	313
Книги о дизайне и ООП.....	313
Статьи.....	316

# Глава 1

## 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 определенное 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».

Каждый второй раз, когда мы пользуемся наследованием, мы используем стратегию; каждый раз, когда 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 поддер-

живают 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.<sup>1</sup>

Давайте в качестве примера рассмотрим задачу импорта лог-файлов для последующего полнотекстового поиска. Главной задачей данного приложения является чтение лог-файлов из различных источников (рис. 1.1), приведение их к некоторому каноническому виду и сохранение в каком-то хранилище, например Elasticsearch или SQL Server.

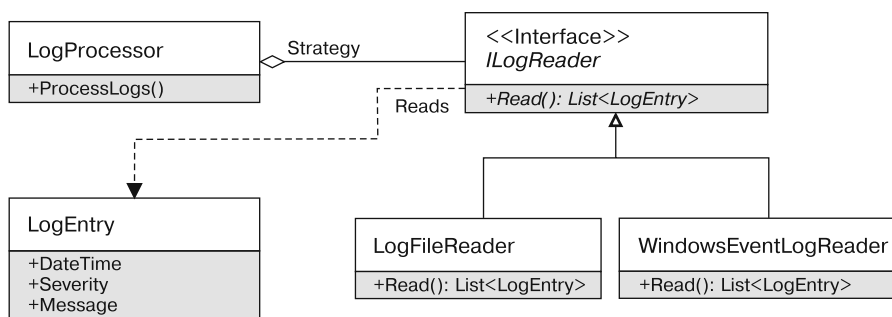


Рис. 1.1. Диаграмма классов импорта логов

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 зависит от конкретной реализации.

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

Классическая диаграмма классов паттерна «Стратегия» приведена на рис. 1.2.

#### Участники:

- Strategy (ILogReader) — определяет интерфейс алгоритма;
- Context (LogProcessor) — является клиентом стратегии;
- ConcreteStrategyA, ConcreteStrategyB (LogFileReader, WindowsEventLogReader) — являются конкретными реализациями стратегии.

<sup>1</sup> Подробнее все эти паттерны проектирования будут рассмотрены в последующих главах.

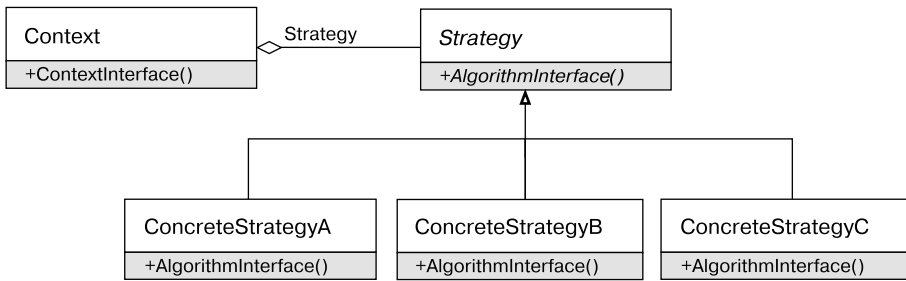


Рис. 1.2. Диаграмма классов паттерна Стратегия

Обратите внимание на то, что классический паттерн «Стратегия» 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 экземпляры стратегии. Контекст может получать ее в аргументах конструктора, через метод, свойство или 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`) или 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 (листинг 1.1).

### Листинг 1.1. Класс `LogProcessor`

```

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...
}
```

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 решениям и вызывает непонимание фундаментальных принципов паттернов проектирования.

## Обсуждение паттерна «Стратегия»

По определению применение стратегии обусловлено двумя причинами:

- необходимостью инкапсуляции поведения или алгоритма;
- 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 не просто требуется спрятать алгоритм, а 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 довольно острым вопросом в современной разработке ПО и актуально не только в контексте стратегий. Поэтому все последующие размышления применимы к любым иерархиям наследования.

Сейчас существует два противоположных лагеря в мире объектно-ориентированного программирования: ярые сторонники и ярые противники выделения интерфейсов. 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? Нужны ли нам интерфейсы `IFactory` или `IAbstractFactory`, или подойдет один конкретный класс? Ответы на эти вопросы зависят от того, нужна ли нам стратегия (или политика) валидации или создания объектов. Хотим ли мы заменять эту стратегию во время исполнения или 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` увеличивает гибкость, но в то же время повышает сложность. Теперь клиентам класса `LogProcessor` нужно решить, какую реализацию использовать, или переложить эту ответственность на 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, 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>` (листинг 1.2).

### Листинг 1.2. Примеры стратегий сортировки

```
class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
    public override string ToString()
    {
        return string.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));
}
```

По сравнению с использованием лямбда-выражений реализация интерфейса требует больше кода и приводит к переключению контекста при чтении. При использовании метода `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` (листинг 1.3).

**Листинг 1.3. Конструирование объекта `SortedSet`**

```
var comparer = new EmployeeByIdComparer();

// Конструктор принимает IComparable
var set = new SortedSet<Employee>(comparer);

// Нет конструктора, принимающего делегат Comparison<T>
```

В этом случае можно создать небольшой адаптерный фабричный класс, который будет принимать делегат `Comparison<T>` и возвращать интерфейс `IComparable<T>` (листинг 1.4).

**Листинг 1.4. Фабричный класс для создания экземпляров `IComparer`**

```
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);
}
}
```

Теперь можно использовать этот фабричный класс следующим образом (листинг 1.5).

#### Листинг 1.5. Пример использования класса `ComparerFactory`

```
var comparer = ComparerFactory.Create<Employee>(
    (x, y) => x.Id.CompareTo(x.Id));
var set = new SortedSet<Employee>(comparer);
```



#### ПРИМЕЧАНИЕ

Можно пойти еще дальше и вместо метода императивного подхода на основе делегата `Comparison<T>` получить более декларативное решение, аналогичное тому, что используется в методе `Enumerable.OrderBy`: 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, однако увеличивает сложность. Любой класс уже отделяет 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<sup>1</sup> существенно усложняют поддержку и сопровождение такого кода.

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

---

<sup>1</sup> Принцип замещения Лисков будет рассмотрен в части IV книги.