Оглавление

Часть III. Структурные паттерны

Часть IV. Принципы проектирования

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.[1]

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.

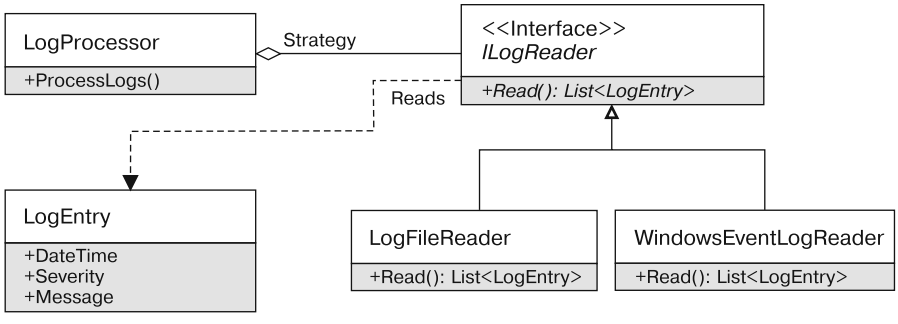

Рис. 1.1. Диаграмма классов импорта логов

`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.

[1] Подробнее все эти паттерны проектирования будут рассмотрены в последующих главах.

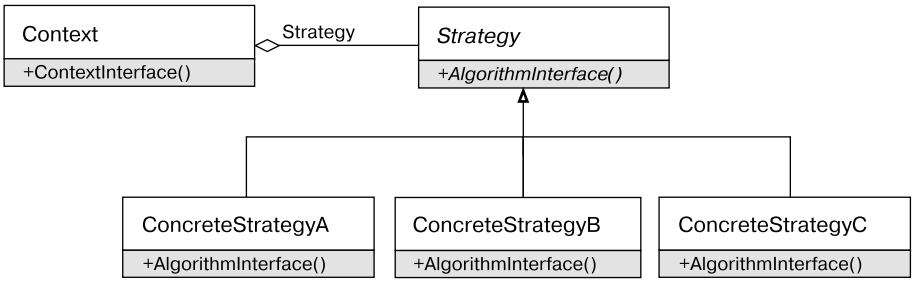

Рис. 1.2. Диаграмма классов паттерна Стратегия

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 экземпляр 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 показывает 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 `LogFile-Importer`, 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()
    {
      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));
}
```

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.OrderyBy: 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[1] существенно усложняют поддержку и сопровождение такого кода.

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

[1] Принцип замещения Лисков будет рассмотрен в части IV книги.