# Часть 1. Паттерны поведения

При разработке любого приложения, программист сталкивается с определенным набором типовых задач. Многие из них, он решает практически не задумываясь, и во многих случаях даже не замечает, что раз за разом использует известные паттерны проектирования. Но понимание ряда паттернов проектирования, особенно паттернов поведения, могут существенно повысить качество дизайна приложения.

За последние 20 лет, объектно-ориентированное программирование (ООП) стало стандартной парадигмой программирования. Сегодня не нужно никому доказывать его важность по сравнению с подходами структурного программирования. Но многим разработчикам все еще нужно научиться использовать его правильно.

Одним из ключевых столпов ООП является наследование. Удачное использование иерархии типов может сделать вашу систему простой и расширяемой, а неудачное использование - превратит ее в слабо понятный клубок из интерфейсов и абстрактных классов. Сегодня повально выделяют интерфейсы для любого класса, не задумываясь, нужно это или нет.

Каждый раз, когда я задумываюсь о выделении интерфейса, т.е. о выделении дополнительного уровня абстракции, я думаю об этом процессе, как о выделении стратегии. Паттерн *Стратегия* определяет семейство алгоритмов, клиенты которых ничего не знают о конкретной реализации. Более того, клиенты получают стратегию извне, что делает возможным ее замену во время исполнения. Если мне нужна такая гибкость, то я выделяю интерфейс и создаю "стратегию", в противном случае, я использую конкретные классы.

Если же решение об использовании наследования принято, то остается проблема формализации взаимоотношений между базовым классом и наследниками. В простом случае, данная формализация не требуется. Достаточно посмотреть код базового класса и сценарии его использования, чтобы понять, что можно, а что нельзя делать при переопределении виртуального метода. Однако в более сложных случаях такой наивный подход перестает работать. Паттерн *Шаблонный метод* позволяет формализовать отношения между базовым классом и его наследниками. Данный паттерн определяет каркас алгоритма в базовом классе, давая возможность наследникам определить лишь недостающие куски головоломки. Шаблонный метод вместе с принципами проектирования по контракту позволяют сделать иерархии наследования более понятными и предсказуемыми с точки зрения разработчика.

На заре становления ООП его молодые адепты уверяли нас в превосходстве объектной парадигмы над старыми подходами структурного программирования. Главным преимуществом преподносилось легкая расширяемость объектных систем за счет добавления новых производных классов. Действительно, очень легко добавить новый тип в существующую иерархию наследования. Но как насчет добавления новой операции? Добавление нового абстрактного метода в базовый класс "сломает" всех наследников. Добавление нового виртуального метода решит эту проблему, но далеко не всегда возможно добавить виртуальный метод с разумным поведением по умолчанию. Паттерн *Посетитель* решает проблему добавления новых операций в существующие иерархии типов.

Не все паттерны поведения предназначены для решения проблем наследования. Вторая половина борется с другой фундаментальной проблемой проектирования - с высокой связанностью (high coupling).

Одной из причин высокой связанности является излишняя "осведомленность" классов друг о друге. Класс чтения данных знает о классе сохранения данных; классы бизнес-логики знают о пользовательском интерфейсе; класс формирования отчета знает о классе конфигурации и т.п. В любом реальном приложении классы не могут быть полностью автономными, но мы можем сделать так, чтобы они не "общались с незнакомцами". Нужно связать классы чтения и сохранения данных? Добавьте третьего участника - *посредника*, который будет работать с каждым из них и обеспечивать связь между ними!

Паттерн *Посредник* позволяет связать несколько автономных классов воедино для получения более высокоуровневого поведения. Вместо того, чтобы классы знали друг о друге, они будут знать лишь о посреднике, что уменьшит связанность, а также упростит их понимание и повторное использование.

Большинство современных приложений работают с разнообразными структурами данных. Мы далеко не всегда замечаем между ними разницу, поскольку манипулируем ими с помощью цикла foreach или LINQ-запросов (*). Но это возможно лишь благодаря абстрактному слою* итераторов*. Сегодня паттерн* Итератор\* поддерживается большинством языков программирования, но понимание его особенностей в контексте языка C# будет очень полезным для его корректного использования.

(\*) Сноска: речь идет о встроенном в C# (и VB.NET) языка запросов под названием LINQ - Language Integrated Query.

События в языке C# являются примером очередного паттерна поведения - паттерна *Наблюдатель*. Наблюдатели уменьшают связанность между классом и его клиентами, поскольку позволяют уведомлять всех заинтересованных подписчиков, не зная об их количестве и конкретных типах. Возросший интерес к функциональному программированию, привел к появлению реактивных расширений (reactive extensions), которые являются смесью паттернов наблюдатель и итератор.

Паттерны поведения не являются выдумкой теоретиков. Они органично решают типовые проблемы, с которыми постоянно сталкиваются разработчики. В первой части книги мы их подробно рассмотрим на примере приложения, импортирующего лог-файлы для полнотекстового поиска. Мы подробно рассмотрим влияние этих паттернов на дизайн приложения. Мы вспомним классическое представление этих паттернов, а также выясним, в каком виде они используются в современных .NET приложениях.