# Часть 2. Принципы проектирования

Любая индустрия по мере взросления старается делиться своим опытом с подрастающим поколением. Паттерны проектирования, рассмотренные в предыдущей части, являются отличным примером повторного использования знаний и опыта более опытных проектировщиков. Паттерны весьма полезны (иначе зачем бы я решил написать еще одну книгу по этой теме?), но они показывают типовые решения типовых задач. Некоторые паттерны весьма общие, другие паттерны специфичны для конкретной предметной области, но многим все равно будет не хватать более фундаментальных путеводных нитей, за которыми будет двигаться развитие дизайна.

В этой части мы рассмотрим популярные принципы проектирования, получившие звучную аббревиатуру - SOLID. В середине 90-х годов Боб Мартин начал публикацию статей в журнале C++ Report на тему проектирования. В качестве основы были взяты несколько известных ранее принципов проектирования, добавлены свои собственные мысли и на свет появились фундаментальные принципы объектно-ориентированного проектирования.

Изначально эти принципы были описаны несколько в ином порядке, и в архиве "дядюшки Боба" значатся под абревиатурой SOLDI. Через несколько лет они перекочевали в книгу "Agile Software Development, Principles, Patterns, and Practices", а со временем в аналогичную книгу с примерами на языке C# - "Принципы, практики и методики гибкой разработки на языке C#", но в них порядок уже был иным, и так появились "цельные" (SOLID) принципы, звучность названия которых в немалой степени обеспечила им успех.

Оригинальные статьи были опубликованы почти 20 лет назад, в качестве примеров использовался язык С++, а объектно-ориентированное мышление лишь только становилось "мейнстримом". Столь почтенный возраст дает о себе знать, и оригинальные статьи Роберта Мартина содержат ряд советов, полезных лишь в языке С++ двадцатилетней давности. Основной упор делается на критике структурного программирования и восхвалении объектно-ориентированного подхода. Любопытно, что многие "современные" описания SOLID принципов все еще показывают пользу полиморфизма на примере иерархии фигур и говорят о проблемах транзитивных зависимостей, которые далеко не столь актуальны в C# или Java.

Я же хочу рассмотреть SOLID принципы с более абстрактной точки зрения, подумать о том, какую проблему они призваны решать (и решают ли), и как мы должны смотреть на них сегодня, когда ООП уже давно стало широко распространённой парадигмой программирования.

**Принцип единственной обязанности** (SRP - The **S**ingle Responsibility Principle: *у класса/модуля должна быть лишь одна причина для изменения*. Данный принцип говорит о борьбе с изменениями, но на самом деле суть его сводится к борьбе со сложностью (tackling the complexity). Любой сложный класс должен быть разбит на несколько простых составляющих, отвечающих за определенный аспект поведения. Это упрощает как понимание, так и развитие класса в будущем. Простой класс с небольшим числом зависимостей легко изменить, не зависимо от того, сколько причин для изменения существует. Разработчик очень редко знает, как требования будут изменяться в будущем, что делает простое решение лучшим способом обеспечения гибкости.

**Принцип открыт/закрыт** (OCP - The **O** pen-Closed Principle): *программные сущности (классы, модули, функции и т.п.) должны быть открытыми для расширения, но закрытыми для модификации*. Любой готовый модуль должен быть стабильным (закрытым) с точки зрения своего интерфейса, но открытым с точки зрения реализации. Закрытость модулей означает стабильность интерфейса и возможность использования модулей его клиентами. Открытость модулей означает возможность его изменения, путем изменения реализации или же путем переопределения поведения за счет создания наследников. Сокрытие информации и полиморфизм позволяют ограничить количество изменений, которые понадобится внести в систему при очередном изменении требований, что сделает этот процесс более простым и управляемым.

**Принцип замещения Лисков** (LSP - The **L** iskov Substitution Principle): *должна быть возможность вместо базового типа подставить любой его подтип*. Поскольку наследование является одним из ключевых механизмов объектно-ориентированного проектирования, очень важно использовать его корректным образом. Данный принцип дает четкие рекомендации о том, в каких пределах может изменяться поведение методов, переопределенных в производных классах, чтобы между классами сохранялось отношение "ЯВЛЯЕТСЯ".

**Принцип разделения интерфейсов** (ISP - The **I** nterface Segregation Principle): *клиенты не должны вынужденно зависеть от методов, которыми не пользуются*. Интерфейс класса определяется некоторым контрактом, которому он должен следовать ради своих клиентов. Иногда возникают ситуации, когда у разных клиентов появляются различные сценарии использования класса, в результате чего его интерфейс становится несогласованным и неудобным в использовании. Данный принцип говорит о том, что клиенты хотят иметь цельный и согласованный интерфейс сервисов, не зависимо от того, пользуются ли этими сервисами кроме них кто-то еще или нет.

**Принцип инверсии зависимостей** (DIP - The **D** ependency Inversion Principle): *Модули верхнего уровня не должны зависеть от модулей нижнего уровня. И те и другие должны зависеть от абстракций.* Слишком большое число зависимостей класса говорит о проблемах в его дизайне. Возможно класс делает слишком многое, или же класс неудачно спроектирован, что приводит к необходимости вызова по одному методу у большого числа зависимостей. Любая объектная система представляет собой граф взаимодействующих объектов. При этом, некоторые зависимости являются деталями реализации и контролируются классами самостоятельно, а некоторые - должны передаваться извне при конструировании объектов. Данный принцип говорит о необходимости выделения и передачи ключевых зависимостей через аргументы конструктора, что позволяет перенести проблемы создания и выбора конкретных зависимостей на вызывающий код.