# Принцип подстановки Лисков

**Цитата**: *Отыщи всему начало и ты многое поймешь.*  
Или: *Где начало того конца, которым оканчивается начало?*  
Козьма Прутков

**Принцип подстановки Лисков** (Liskov Substitution Principle, LSP): *Должна быть возможность вместо базового типа подставить любой его подтип.* Роберт Мартин. "Принципы, паттерны и практики гибкой разработки" [Martin2006]

*...если для каждого объекта o1 типа S существует объект o2 типа T такой, что для всех программ P, определенных в терминах T, поведение P не изменяется при замене o2 на o1, то S является подтипом (subtype) для T.* Барбара Лисков "Абстракция данных и иерархия", 1988

Наследование и полиморфизм является ключевым инструментом "объектно-ориентированного" разработчика для борьбы со сложностью, для получения простого и расширяемого решения. Наследование используется в большинстве паттернов проектирования и лежит в основе Принципа открыт/закрыт и Принципа инверсии зависимостей.

Большинство опытных разработчиков знает, что с наследованием не все так просто. Наследование – это одна из самых сильных связей в ОО мире, которая крепко привязывает наследников к базовому классу (сильнее только лишь отношение дружбы (\*)). Кроме того, не всегда легко ответить на два простых вопроса: 1) когда наследование уместно и 2) как его реализовать корректным образом.

(\*) Сноска: дружеские отношения (friendship relationship) между классами полноценно поддерживаются в языке С++ и дают полный доступ "другу" к внутренней реализации класса. В языке C# отсутствует отношение дружбы между типами. Схожего поведения можно добиться путем использования вложенных классов. Вложенный класс определяется внутри другого класса и получает полный доступ к закрытым полям внешнего класса.

Наследование обычно моделирует отношение "ЯВЛЯЕТСЯ" (IS-A Relationship) между классами. Говорят, что экземпляр наследника также ЯВЛЯЕТСЯ экземпляром базового класса, что выражается в возможности использования экземпляров наследника везде, где ожидается использование базового класса. Данный вид наследования еще называется наследованием подтипов (Subtype Inheritance), но он не является единственно возможным. Бертран Мейер в своей книге "Объектно-ориентированное конструирование программных систем" приводит 12 (!) различных видов наследования, включая наследование реализации (закрытое наследование), IS-A, Can-Do (реализация интерфейсов) и т.п.

Большинство современных объектно-ориентированных языков программирования не поддерживают множественного или закрытого наследования, поэтому наибольший интерес вызывает именно наследование подтипов. Принцип подстановки Лисков призван помочь в корректной реализации этого вида наследования, что также должно помочь отказаться от наследования, если его корректная реализация невозможна.

Чтобы принцип подстановки был более полезным, нужно подобрать для него более звучное определение. Проблема с приведенными выше определениями в том, что определение Роберта Мартина повторяет определение отношения «ЯВЛЯЕТСЯ», а исходное определение от Барбары Лисков выглядит слишком академичным и также слабо применимым на практике. Одно из лучших определений этого принципа находится на сайте Уорда Каннингема (Ward Canningham) - c2.com (\*):

**Принцип подстановки Лисков**: *должна существовать возможность использовать объекты производного класса вместо объектов базового класса. Это значит, что объекты производного класса должны вести себя согласованно, согласно контракту базового класса.*

(\*) Сноска: Уорд Каннингем является создателем языка разметки Wiki, а также одним из соавторов экстремального программирования. Данное определение взято из его заметки Liskov Substitution Principle - http://c2.com/cgi/wiki?LiskovSubstitutionPrinciple.

## Для чего нужен принцип подстановки Лисков?

Основной смысл любой иерархии наследования в том, что она позволяет использовать базовые классы полиморфным образом, не задумываясь о том, экземпляр какого конкретного класса был передан.

В своем приложении мы можем «абстрагироваться» от конкретной реализации потоков ввода-вывода (Streams), коллекций и последовательностей (Enumerables), провайдеров, репозиториев и т.п. Но для корректного поведения, приложение должно отталкиваться от некоторых допущений, которые будут справедливы для всех реализаций абстракции: что возвращаемое значение никогда не будет null, что будут генерироваться исключения лишь определенного типа, что любая реализация должна сохранить данные в базу данных (не важно какую), или метод добавления элемента обязательно его добавит, и размер коллекции увеличится на единицу.

Здесь мы сталкиваемся с такой особенностью: с одной стороны, любая реализация должна следовать некоторому абстрактному протоколу или контракту, а с другой стороны, она должна иметь возможность выбрать конкретный способ реализации этого протокола. Именно контракт, не важно, формальный или нет, описывает ожидаемое видимое поведение абстракции, оставляя реализации решать, каким образом это поведение будет реализовано.

Если же реализация (т.е. наследники) не будет знать об этом протоколе или не будет ему следовать, то в приложении мы будем вынуждены обрабатывать конкретную реализацию специальным образом, что сводит на нет идею использования наследования и полиморфизма.

## Почему важно следовать принципу подстановки Лисков?

1. Поскольку в противном случае иерархии наследования приведут к неразберихе. Неразбериха будет заключаться в том, что передача в метод экземпляра класса-наследника приведет к странному поведению существующего кода.
2. Поскольку в противном случае юнит-тесты базового класса никогда не будут проходить для наследников. (\*)

(\*) Сноска: взято из все той же заметки Уорда Каннингема http://c2.com/cgi/wiki?LiskovSubstitutionPrinciple.

## Классический пример нарушения: квадраты и прямоугольники

Наследование моделирует отношение «ЯВЛЯЕТСЯ». Но поскольку это лишь слово, мы не можем использовать возможность его использования в качестве безоговорочного доказательства возможности использования наследования. Можем ли мы сказать, что цветная фигура является фигурой, контрактник является сотрудником, который, в свою очередь является человеком, квадрат является прямоугольником, а круг – овалом?

Приведенные утверждения могут быть корректными или не корректными, в зависимости от того, какое поведение мы будем приписывать фигурам, сотрудникам или квадратам. Именно в зависимости от спецификации или контракта этих сущностей (т.е. ожидаемого поведения) мы можем говорить о возможности использования наследования.

Давайте рассмотрим более подробно известный пример с квадратами и прямоугольниками. С точки зрения математики, квадрат является прямоугольником, но актуально ли это отношение для классов Rectangle и Square (рис. 3.1)?

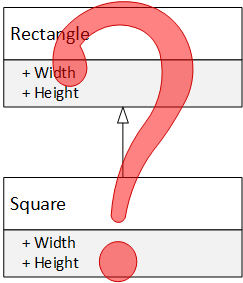


Рисунок 3.1 - Является ли квадрат прямоугольником?

Чтобы понять, будет ли нарушать данная иерархия классов принцип подстановки, нужно постараться сформулировать контракты этих классов:

* **Контракт прямоугольника (инвариант)**: ширина и высота положительны.
* **Контракт квадрата (инвариант)**: ширина и высота положительны; ширина и высота равны.

Пока нельзя сказать, являются ли контракты согласованными. Поскольку у квадрата все стороны равны, то изменение его ширины должно приводить и к изменению его высоты, и наоборот. Это значит, контракт свойств Width и Height квадрата становится несогласованным с контрактом этих свойств прямоугольника! (С точки зрения клиента прямоугольника свойства Width и Height полностью независимы, а значит замена прямоугольника квадратом во время исполнения нарушит это предположение клиента.)

Но это не значит, что данная иерархия наследования является невозможной. Квадрат перестает быть нормальным прямоугольником, ТОЛЬКО если квадрат и прямоугольник являются изменяемыми! Так, если мы сделаем их неизменяемыми (immutable), то проблема с контрактами, принципом подстановки и нарушением поведения клиентского кода при замене прямоугольников квадратами пропадет. **Если клиент не может изменить ширину и высоту, то его поведение будет одинаковым при замене прямоугольников квадратами!**

**ПРИМЕЧАНИЕ** Именно этой же логики следуют правила ковариантности и контравариантности обобщений в языке C#. Так, именно отсутствие изменяемости, позволяет трактовать объект IEnumerable<string> как IEnumerable<object>. Поскольку мы можем лишь достать элемент из последовательности и не можем положить его обратно, то такое преобразование является типобезопасным.

Данный пример показывает несколько важных моментов. Во-первых, именно наличие контракта позволяет четко понять, нарушает ли производный класс принцип подстановки Лисков или нет. А во-вторых, пример с неизменяемостью показывает пользу неизменяемости в объектно-ориентированном мире: контракт неизменяемых типов проще, поскольку контролируется лишь конструктором и инвариантом класса.

В-третьих, этот пример показывает, почему некоторые специалисты рекомендуют, чтобы классы были либо абстрактными, либо запечатанными (sealed) (\*), и не было возможности создавать экземпляры классов из середины иерархии наследования. Так, классическим решением проблемы квадратов/прямоугольников является выделение промежуточного абстрактного класса «четырехугольника», от которого уже наследуются квадрат и прямоугольник. И хотя, в некоторых случаях, такой совет оправдан, слишком строго следовать ему не стоит. Вместо этого лучше точнее описывать контракт базовых классов с помощью контрактов.

(\*) СНОСКА: "Запечатанные" классы не могут иметь наследников.

## Принцип подстановки Лисков и контракты

Не будет преувеличением сказать, что лишь с помощью принципов Проектирования по контракту вы можете точно понять, что представляет собой наследование. Бертран Мейер «Объектно-ориентированное конструирование программных систем», раздел 16.1.

Существует несколько способов описать ожидаемое поведение типа, т.е. его спецификацию. Для этого мы можем использовать комментарии (их никто не читает!), юнит-тесты (лучше, но кто нам даст тесты для классов из BCL?) или контракты (заставить бы всех ими пользоваться!).

Пример с квадратами и прямоугольниками показал, что мы не можем доказать, нарушает ли конкретный класс принцип подстановки Лисков или нет, пока не определимся с тем, что ожидают клиенты от поведения базового класса.

Если посмотреть на исходное описание принципа подстановки в трудах Барбары Лисков, то можно с удивлением обнаружить, что оно полностью основано на таких понятиях, как предусловия, постусловия и инварианты. Другими словами, описание этого принципа полностью основано на принципах проектирования по контракту:

1. Производные классы не должны усиливать предусловия (не должны требовать большего от своих клиентов).
2. Производные классы не должны ослаблять постусловия (должны гарантировать, как минимум тоже, что и базовый класс).
3. Производные классы не должны нарушать инварианты базового класса (инварианты базового класса и наследников суммируются)
4. Производные классы не должны генерировать исключения, не описанные базовым классом.

Попытка формализации обязанностей класса в терминах предусловий, постусловий и инвариантов позволит более четко сказать, является ли поведение наследника согласованным или нет.

## О сложностях наследования в реальном мире

Отношение ЯВЛЯЕТСЯ подразумевает, что поведение клиентского кода должно быть предсказуемым при использовании любого конкретного класса иерархии типов. Существует мнение, что генерация методами наследника исключений InvalidOperationException или NotSupportedException будет нарушать работу клиентского кода, а значит такие наследники будут нарушать принцип подстановки Лисков. В некоторых случаях такое мнение справедливо, но в общем случае судить так не следует.

Давайте рассмотрим, является ли нарушением LSP, что ImmutableList<T> и ReadOnlyCollection<T> реализует IList<T>, ведь попытка добавления элемента в такую коллекцию приводит к генерации NotSupportedException. Наличие или отсутствие нарушения принципа подстановки определяется не наличием исключений, а контрактом реализуемого интерфейса. В «контракте» интерфейса IList<T> (точнее, интерфейса ICollection<T>) четко сказано, что метод Add может добавить элемент (\*), только в случае выполнения «предусловия» – коллекция должна быть изменяемой - свойство IsReadOnly возвращает False! Аналогично дела обстоят с потоками ввода/вывода, методы Read/Write которых могут генерировать исключения, если свойства CanRead/CanWrite возвращают False.

**ПРИМЕЧАНИЕ** «Контракт» и «предусловие» специально взяты в кавычки. На самом деле, это логический контракт и логическое предусловие, описанные в документации интерфейса ICollection<T>, а не в классе его контракта: классе ICollectionContract<T>.

**ВРЕЗКА: Контракты списков и коллекций** TODO: не уверен в необходимости врезки, нужно бы обсудить этот момент! ОБратите внимание на фразу "может добваить элемент". Между постусловиями метода Add интерфейсов IList<T> и ICollection<T> существует небольшое, но очень важное отличие. В случае выполнения предусловия метода Add, реализация "списка" обязана добавить элемент в коллекцию. Постусловие интерфейса ICollection<T> слабее и такой гарантии нет. Существует некоторые виды коллекций, вызов метода Add которых не увеличивает количество элементов на 1. Любые "наборы" (sets) подразумевают наличие в коллекции уникальных элементов, а значит вызов метода Add на экземпляре HashSet<int> не обязательно приведет к увеличению числа элементов.

Если поведение метода зависит от состояния объекта, то мы можем говорить о неудачном дизайне, но не можем утверждать, что реализация нарушает Принцип подстановки Лисков! Даже если метод наследника всегда генерирует InvalidOperationException, то это лишь говорит о том, что не все методы интерфейса или базового класса применимы к наследнику, но это также не означает некорректное использование наследования.

Принцип подстановки Лисков говорит о том, что если при замене объектов класса T, объектами класса S, поведение клиентского кода не меняется, то S является подтипом T (T является базовым классом для S). А что, если поведение при такой замене все-таки изменится? Означает ли это, что нарушается некоторый принцип проектирования, или это значит, что S не является подтипом T?

Согласно принципу наименьшего удивления, мы привыкли считать, что должна быть возможность использовать экземпляр наследника вместо экземпляра базового класса. Но ведь это не значит, что мы никогда не используем классы наследники напрямую! Мы добавляем свойства и методы в производные классы, хотя и понимаем, что они не будут доступны через «призму базового класса». Нас при этом совсем не волнует, что клиент должен знать о классе наследнике, чтобы ими воспользоваться.

Наследование подтипов (subtype inheritance) является одним из многих видов наследования, но далеко не единственным. Наследник имеет право добавлять новые методы («расширяющее наследование»), «удалять» или не реализовывать некоторые методы базового класса («сужающее наследование»), переименовывать методы базового класса для более точной передачи семантики производного класса. Наследование вообще может быть закрытым (private inheritance), и тогда вообще преобразование экземпляров класса наследника к базовому классу будет запрещено компилятором!

Данные техники наследования в полном виде поддерживается лишь несколькими языками программирования. Язык Eiffel, разработанный Бертраном Мейером поддерживает большинство таких техник, но и в практике современного .NET разработчика они также встречаются.

Явная реализация интерфейсов в C# (explicit interface implementation) является примером «сужающего наследования»: класс реализует интерфейс, но при этом методы интерфейса видны лишь в случае явного приведения экземпляра класса к интерфейсу.

var list = new List<int> { 42 }.AsReadOnly();  
  
//list.Add(42); // ошибка компиляции!  
((IList<int>)list).Add(42); // ошибка времени исполнения  
IList<int> list2 = list; // к сожалению, здесь явное приведение не требуется!  
list2.Add(42); // ошибка времени исполнения

Листинг 3.1 - Работа с коллекциями только для чтения

К сожалению, в языке C# остается возможность неявного приведения экземпляра класса к явно реализованному интерфейсу. В противном случае сходство с сужающим наследованием было бы максимальным: нам обязательно пришлось бы приводить экземпляр класса к нужному интерфейсу, чтобы воспользоваться его членами.

Явная реализация интерфейса используется также для «переименования членов интерфейса» - возможности, активно используемой в языке Eiffel при решении конфликтов множественного наследования, а также для использования в классах наследниках более подходящих имен. Классическим примером использования этой возможности, является класс Socket, который реализует интерфейс IDisposable явно и при этом вводит «синоним» спрятанного метода Dispose с именем Close.

Socket s = CreateSocket();  
s.Dispose(); // Не компилируется до.NET 4.0  
((IDisposable)s).Dispose(); // Всегда OK  
s.Close(); // Всегда OK

Листинг 3.2 - Пример работы с сокетами

Об этой технике явно говорится в книге “Framework Design Guidelines” (\*) в разделе “Dispose Pattern Guidelines”: автор класса имеет полное право при реализации интерфейса воспользоваться более подходящим именем, и «спрятать» от клиентов имя из интерфейса.

(\*) СНОСКА: Brad Abrams, Krzysztof Cwalina Framework Design Guidelines, 2nd Edition, Addison-Wesley Professional, 2008.

**ПРИМЕЧАНИЕ** В плане наследования С++ является существенно более функциональным языком, нежели C#. Помимо множественного наследования, С++ поддерживает большую часть техник, описанных Мейером. Так, в С++ есть наследование реализации (закрытое наследование), при этом автоматическая конвертация экземпляров наследников в экземпляры базовых классов производиться не будет. С++ поддерживает изменение видимости функции при переопределении: видимость можно как расширить (от защищенной в базовом классе к открытой в наследнике), так и сузить (от открытой в базовом классе к закрытой/защищенной в наследнике). В С++ легче моделировать «примеси» и др.

## Когда наследования бывает слишком мало!

Обычно, проблемы с наследованием возникают из-за слишком «широких» или «глубоких» иерархий наследования. Но иногда проблемы в дизайне возникают из-за того, что класс унаследован не от того базового класса или не реализует нужный интерфейс.

Классическим нарушением принципа Открыт/Закрыт является «перебор типов»: если переданный объект – это круг, рисуем круг, если квадрат – рисуем квадрат, если слон – рисуем слона; при этом такой перебор "размазан" по коду приложения. Обычно такие подходы характерны структурному подходу, но часто бывают и в объектно-ориентированных решениях.

В идеальном объектно-ориентированном мире, все, что делает иерархия классов находиться прямо в ней, однако в реальном мире часть логики периодически оказывается за ее пределами. Хорошим примером является LINQ – Language Integrated Query (Интегрированный язык запросов). LINQ содержит унифицированную логику по работе с любыми последовательностями - сущностями, которые реализуют интерфейс IEnumerable<T>. Это даже не обязательно должны быть коллекции в памяти, это могут быть бесконечные генераторы, реализованные с помощью блока итераторов и ключевого слова yield return, или это могут быть XML-файлы, или записи из базы данных.

Но даже когда дело касается коллекций, класс Enumerable содержит специализированные реализации некоторых операций, таких как Count(), ElementAt(), Last() и других, для конкретных типов коллекций, например Collection<T>, List<T> и т.п.

Вот, как примерно выглядит реализация методаEnumerable.Count():

public static int Count<TSource>(this IEnumerable<TSource> source)  
{  
 var collectionoft = source as ICollection<TSource>;  
 if (collectionoft != null) return collectionoft.Count;  
  
 var collection = source as ICollection;  
 if (collection != null) return collection.Count;  
  
 int count = 0;  
 using (IEnumerator<TSource> e = source.GetEnumerator())  
 {  
 while (e.MoveNext()) count++;  
 }  
  
 return count;  
}

Листинг 3.3 - Реализация метода расширения Count

Данная специализация позволяет получить значительный прирост производительности при вызове метода Count() на коллекциях (списках, массивах и любых других классах, прямо или косвенно реализующих интерфейс ICollection). Но поскольку проверяется лишь интерфейс ICollection, то эта реализация будет неэффективной для других коллекций, которые знают о своем размере, но не реализуют интерфейс ICollection.

В .NET Framework существует класс "коллекции" со свойством Count, который не реализует ICollection – класс Lookup<T>:

var lookup = Enumerable.Range(1, 10000).ToLookup(x => x);  
Console.WriteLine(lookup.Count); // Сложность О(1) (\*)  
Console.WriteLine(lookup.Count()); // Сложность О(n)

Листинг 3.4 - Использование метода Count для типа Lookup

В данном случае использование lookup.Count является значительно более эффективным, поскольку не зависит от числа элементов. Вычислительная сложность lookup.Count() является линейной, а значит время вызова будет пропорционально числу элементов данной коллекции.

(\*) Сноска: О-нотация предназначена для обозначения асимптотической сложности алгоритмов. Другими словами, она говорит о зависимости времени исполнения или количества используемой алгоритмом памяти, от количества входных элементов. О(1) означает, что время алгоритма не зависит от количества входных элементов. O(n) говорит о линейной зависимости: при увеличении числа элементов в 2 раза, время алгоритма также изменится вдвое. Подробнее об асимптотической сложности алгоритмов можно прочитать на Вики (http://ru.wikipedia.org/wiki/Вычислительная\_сложность) или в вашей любимой книге по алгоритмам и структурам данных.

Данный пример показывает, что проблемы с наследованием бывают разных типов: когда поведение наследника противоречит поведению базового класса, или когда класс не реализует нужный интерфейс или базовый класс.

## Принцип подстановки Лисков на практике

В предыдущей главе мы добавили иерархию парсеров лог-файлов:

public abstract class LogEntryParser  
{  
 public abstract bool TryParse(string line, out LogEntry logEntry);  
}

Листинг 3.5 - Интерфейс класса LogEntryParser

Глядя на объявление данного класса, многим опытным разработчикам будет понятен его неформальный контракт. Метод должен возвращать true в случае успешного разбора строки, и false в противном случае. При успешном разборе, выходной аргумент logEntry должен содержать разобранное значение.

Но данный контракт не является формальным. Он не описывает возможные типы генерируемых исключений, а также поведение при передаче null в качестве параметра line. Неформальность контрактов часто приходит к тому, что замена одной реализации на другую ломает поведение существующего кода.

Лучший способ избежать неоднозначности, постараться максимально формализовать контракт абстрактного метода. Сделать это можно с помощью предусловий или постусловий, или, по крайней мере, с помощью комментариев:

public class LogEntryParserException : Exception  
{}  
  
[ContractClass(typeof (LogEntryParserContract))]  
public abstract class LogEntryParser  
{  
 /// <summary>  
 /// Анализирует переданную строку и возвращает результат через <paramref name="logEntry"/>  
 /// </summary>  
 /// <exception cref="LogEntryParserException">  
 /// Генерируется, если переданная строка не соответствует ожидаемому формату.  
 /// </exception>  
 public abstract bool TryParse(string line, out LogEntry logEntry);  
}  
  
[ExcludeFromCodeCoverage,ContractClassFor(typeof (LogEntryParser))]  
abstract class LogEntryParserContract : LogEntryParser  
{  
 public override bool TryParse(string line, out LogEntry logEntry)  
 {  
 Contract.Requires(line != null);  
 Contract.Ensures(!Contract.Result<bool>() ||   
 Contract.ValueAtReturn(out logEntry) != null,  
 "Если результат True, то logEntry не может быть null");  
 throw new NotImplementedException();  
 }  
}

Листинг 3.6 - Контракт интерфейса LogEntryParser

При наличии контракта, наследники будут знать, что ожидают клиенты базового класса, что им можно делать при переопределении методов, а что нет. Следующий перечень действий наследников класса LogEntryParser нарушает Принцип подстановки Лисков:

* Генерация исключений, отличных от LogEntryParserException или его наследников. Другие исключения возможны, но они будут означать ошибки (баги) в реализации классов-наследников.

public class IncorrectLogEntryParser : LogEntryParser  
{  
 public override bool TryParse(string line, out LogEntry logEntry)  
 {  
 if (LineIsIncorrect(line))  
 {  
 // Неизвестный тип исключения! Нарушение LSP!  
 throw new FormatException("Can't parse the line!");  
 }  
 // ...  
 }  
}

Листинг 3.7 - Генерация некорректного типа исключения методом TryParse

* Метод TryParse возвращает false при передаче null в качестве значения аргумента line . В этом случае должно генерироваться исключение ArgumentNullException или ContractException.

public override bool TryParse(string line, out LogEntry logEntry)  
{  
 logEntry = null;  
 // Отсутствие генерации исключения, когда line == null!  
 // Нарушение LSP!  
 if (string.IsNullOrEmpty(line))  
 return false;  
 // ...  
}

Листинг 3.8 - Некорректная проверка аргументов методом TryParse

* Метод TryParse возвращает true, но не возвращает значение через возвращаемый аргумент logEntry:

public override bool TryParse(string line, out LogEntry logEntry)  
{  
 logEntry = null;  
 // logEntry равен null! Нарушение LSP!  
 return true;  
 // ...  
}

Листинг 3.9 - Некорректное возвращаемое значение методом TryParse

Далеко не всегда нужно описывать контракт базовых классов столь формально. Но, чем лучше вы, и другие разработчики будете его понимать, тем меньше шансов, что поведение текущего кода будет нарушено при замене одной реализации на другую.

## Типичные примеры нарушения LSP

* Производные классы используются полиморфным образом, но их поведение не согласуется с поведением базового класса: генеруются исключения, не описанные контрактом базового класса или не выполняются действия, предполгаемые контрактом базового класса.
* Контракт базового класса настолько нечеткий, что реализовать согласованное поведение наследником просто невозможно.

## Выводы

Принцип подстановки Лисков не является панацеей в вопросах наследования, он лишь помогает формализовать, в каких пределах может варьироваться поведение наследника с точки зрения контракта базового класса. В своих трудах Барбара Лисков строила свой анализ на основе контрактов класса: предусловий, постусловий и инвариантов. Именно с помощью контрактов мы можем хотя бы с некоторой долей уверенности утверждать, что поведение наследника и базового класса являются согласованными.

Когда речь касается наследования, то разработчик должен четко понимать, для чего он его использует и каким образом клиенты будут пользоваться наследниками: лишь через призму базового класса, напрямую, или же и так, и так. Когда тип предназначен лишь для полиморфного использования, то такое наследование является наследованием подтипов и должно следовать принципу подстановки Лисков. Если же создание наследника нужно для повторного использования кода базового класса, или производный класс будет всегда использоваться напрямую, то вполне возможно, что его интерфейс и контракт будет изменен: добавлены новые методы и/или не реализованы некоторые методы базового класса.

Нельзя говорить, что второй пример использования наследования вреден или не должен использоваться на практике; как минимум, он должен быть обдуман и четко описан в документации класса-наследника, чтобы пользователь вашего класса знал о вашем решении. Создание иерархий наследования удобных для развития и использования - это сложный итеративный процесс. Четкое следование принципам проектирования вообще, и принципу замещения в частности, не гарантирует хорошего дизайна или удобной в использовании иерархии наследования.

## Дополнительные ссылки

* Barbara Liskov and Jannette Wing “A Behavioural Notion of Subtyping” - http://web.cse.ohio-state.edu/~neelam/courses/788/lwb.pdf Это та самая статья, в которой Барбара Лисков описывает свой принцип, опираясь на такие понятия, как предусловия, постусловия и инварианты.
* Обсуждение принципа подстановки Лисков на c2.com - http://c2.com/cgi/wiki?LiskovSubstitutionPrinciple. На этой странице принцип подстановки обсуждают легенды современной разработки ПО, такие как Майкл Физерс (Michael Feathers), Алистер Кокберн (Alistair Cockburn), Роберт Мартин (Robert C. Martin) и другие.
* Programming Stuff. Цикл статей о проектировании по контракту (первая статья серии: http://sergeyteplyakov.blogspot.com/2010/05/design-by-contract.html). Контрактное программирование является очень полезной методикой проектирования. В контексте принципа подстановки просто необходимо понимать, хотя бы на базовом уровне, что такое контракт класса, не зависимо от того, доступны ли инструменты контрактного программирования для вашего языка или нет.