# Принцип разделения интерфейсов

**Цитата**: *Многие вещи нам непонятны не потому, что наши понятия слабы; но потому, что сии вещи не входят в круг наших понятий.*  
Козьма Прутков

**Принцип разделения интерфейсов** (Interface Segregation Principle, ISP): *клиенты не должны вынужденно зависеть от методов, которыми не пользуются.* Роберт Мартин. "Принципы, паттерны и практики гибкой разработки" [Martin2006]

Необходимым условием повторного использования кода является простота его использования. Разработчик никогда не будет использовать чужой код, если ему проще реализовать эту возможность самостоятельно, а не искать и разбираться в чьём-то решении. С одной стороны, программисты достаточно ленивы и не склонны делать больше, чем того требует задача, но, с другой стороны, они достаточно горды и упрямы, чтобы считать любое чужое решение по определению менее удачным, чем свое собственное.

Синдром "я бы сделал это не так" является самым главным камнем преткновения во время рецензирования кода. Этот же синдром останавливает многих разработчиков от использования чужого кода. На своем горьком опыте разработчики уяснили, что разработка повторно используемого кода - дело сложное. Поэтому они нечасто используют чужой код, но активно продвигают повторное использование своего собственного кода.

Одной из причин скептического отношения к повторно используемому коду является сложность проектирования библиотек, простых и удобных в использовании. С одной стороны, библиотека должна решать базовые вещи простым и интуитивно понятным способом. С другой стороны, она должна делать возможными и более сложные сценарии. Добиться простоты использования кода можно за счет проектирования "чистых" интерфейсов: интерфейсов, которые предоставляют цельный набор операций, который будет необходим и достаточен большинству клиентов.

## Для чего нужен принцип разделения интерфейса?

Принцип разделения интерфейса предназначен для получения простого и слабосвязного кода. Он гласит, что клиенты должны зависеть лишь от тех методов, которые они используют, и не должны знать о существовании не интересующих их частей в интерфейсе используемых ими сервисов. Как мы увидим позднее, разработчик сервиса не всегда знает о том, кто и как его будет использовать. Поэтому может потребоваться несколько итераций, чтобы перегруппировать методы таким образом, чтобы их использование было удобным максимальному числу клиентов.

Данный принцип является частным случаем принципа наименьшего знания, который много лет используется в нашей индустрии. Для получения простого в сопровождении кода, каждый класс должен знать минимум информации об окружающем коде, необходимой для решении своей задачи. На практике это проявляется в минимизации числа зависимостей и в стремлении к использованию наиболее простых типов зависимостей.

Сопровождаемость класса определяется его внутренней сложностью и связанностью с другими классами. Чем больше у класса зависимостей, тем сложнее понять его роль, сложнее тестировать и использовать повторно. К тому же, большое число связей увеличивает вероятность поломки класса при изменении зависимостей. Стабильность зависимостей играет важную роль. Чем ниже вероятность изменения интерфейса зависимости или его поведения, тем меньше вероятность поломки вашего кода. Ниже представлены виды зависимостей, стабильность которых уменьшается от "очень стабильной" до "нестабильной":

* Примитивные типы.
* Объекты-значения (неизменяемые пользовательские типы).
* Объекты со стабильным интерфейсом и поведением (пользовательские типы, интерфейс которых стабилен, а поведение не зависит от внешнего окружения).
* Объекты с изменчивым интерфейсом и поведением (типы расположены на стыке модулей, которые постоянно подвергаются изменениям или типы, которые работают с внешним окружением: файлами, базами данных, сокетами и т.п.). (\*)

(\*) СНОСКА или ПРИМЕЧАНИЕ: мы тут затронули очень сложную тему проектирования, связанную с управлением зависимостями. Это весьма сложная тема, которая требует нескольких глав или даже целой книги. В качестве дополнительных источников по этой теме может служить книга Марка Симана "Dependency Injection in .NET", и цикл моих статей по управлению зависимостями - http://bit.ly/DependencyManagement.

Принцип разделения интерфейсов также соответствует одному из принципов контрактного программирования: **требуй меньше, но гарантируй больше**. В терминах контрактов это проявляется в виде использования более слабого предусловия и более строгого постусловия. Чем проще аргументы метода, тем проще удовлетворить предусловиям метода. На практике это правило выражается в виде использования наиболее простых типов (согласно приведенной выше шкалы), а также в использовании наиболее базовых типов в качестве входных аргументов:

public Result Process1(List<RequestArgs> input)  
{  
 foreach(var ra in input)  
 {  
 // ...  
 }  
}  
  
public Result Process2(IEnumerable<RequestArgs> input)  
{  
 foreach(var ra in input)  
 {  
 // ...  
 }  
}

Листинг 4.1 - Использование базовых и производных типов в качестве аргументов метода

Если метод использует лишь члены интерфейса IEnumerable<T>, то нет смысла заявлять, что он требует List<T>. Если метод может работать с любым потоком ввода-вывода, то лучше ему принимать Stream, а не MemoryStream. Если классу требуется конфигурация, то лучше передавать в аргументах конструктора экземпляр класса Configuration (объект-значение), а не провайдер IConfigurationProvider, который будет читать конфигурацию в методе ReadConfiguration.

**ПРИМЕЧАНИЕ** В случае с коллекциями и последовательностями нужно очень обдуманно подходить к выбору типа аргументов. Часто бывает, что в качестве аргумента используется IEnumerable<T>, но при этом рассчитывается на то, что будет передаваться коллекция в памяти. Например, метод может несколько раз вызывать метод расширения Count(), в расчете на то, что он будет выполняться быстро, поскольку передается в метод лишь список или массив. Это является примером "неявной связи" (implicit coupling) и может привести к серьезным проблемам при сопровождении, когда в качестве аргумента метода начнет передаваться последовательность, генерируемая на лету с помощью блока итераторов.

## SRP vs. ISP

Принцип разделения интерфейсов является довольно простым и очень полезным принципом проектирования. Но его иногда путают с принципом единственной обязанности. Причина такого недопонимания лежит в классическом описании ISP [Martin2006]:

Этот принцип относится к недостаткам "жирных" интерфейсов. Говорят, что класс имеет жирный интерфейс, если функции этого интерфейса недостаточно сцепленные (not cohesive). Иными словами, интерфейс класса можно разбить на группу методов. Каждая группа предназначена для обслуживания разных клиентов. Одним клиентам нужна одна группа методов, другим – другая.

Проблема этого определения в том, что в нем делается акцент на "жирности". "Жирный" интерфейс, который содержит несколько групп методов, недостаточно сцепленных между собой нарушают принцип единственной обязанности. Такие методы используются разными клиентами и будут развиваться независимым образом. Однако все не так просто.

Давайте вспомним, как мы определяем, что класс или модуль нарушает принцип единственной обязанности. Мы открываем исходный код этого класса или модуля, и смотрим, не делает ли он слишком многого. Если класс или модуль отвечает за выполнение разнородных задач, значит он однозначно нарушает принцип единственной обязанности. Но можем ли мы глядя на класс или его интерфейс сказать, нарушает он принцип разделения интерфейсов или нет?

Вот, например, у нас есть класс репозитория, который содержит CRUD операции (\*). Нарушает ли он ISP? Мы не знаем! Нарушение этого принципа зависит не столько от самого класса, сколько от сценариев его использования. Если в нашей бизнес-модели четко разделяются операции чтения и обновления данных, то наличие одного класса со всеми операциями по работе с данными однозначно делает интерфейс слишком "толстым". С другой стороны, если наше приложение содержит множество простых форм пользовательского интерфейса, которые соотносятся 1 к 1 с нашими поставщиками данных, то тогда принцип ISP не нарушается.

(\*) СНОСКА: CRUD - Create, Read, Update, Delete, набор ключевых операций при работе с постоянными хранилищами данных, такими как база данных.

Из предыдущего обсуждения следует важное отличие принципов SRP и ISP. **Следование принципу единоственной обязанности приводит к связанным (cohesive) классам, что позволяет с меньшими усилиями их понимать и развивать. Следование принципу разделения интерфейсов уменьшает связанность (coupling) между классами и их клиентами, ведь теперь клиенты используют более простые зависимости, чем раньше.**

## Принцип разделения интерфейсов на практике

На практике периодически возникает необходимость выделения у класса вспомогательных интерфейсов. Класс может наследоваться от базового класса для повторного использования кода, но реализовывать дополнительный интерфейс, который нужен лишь части его клиентов.

Одна из реализаций паттерна Адаптер (\*) заключается в реализации адаптируемым классом дополнительного интерфейса. Итак, у нас есть два класса сохранения данных с несовместимыми интерфейсами - SqlServerLogSaver и ElasticsearchLogSaver. При наличии доступа к их исходному коду, мы можем не создавать новые классы-адаптеры, а просто заставить текущие классы реализовать новый интерфейс - ILogSaver (рис. 4.1):

(\*) Сноска: подробнее паттерн Адаптер был рассмотрен в главе 12.

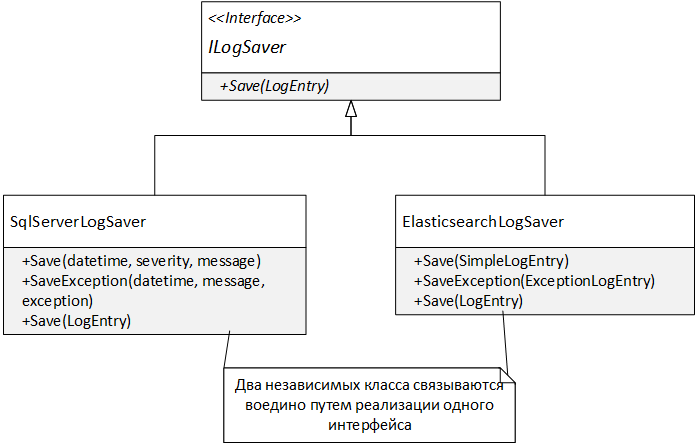


Рисунок 4.1 - Использование паттерна Адаптер для разделения интерфейсов

Теперь у классов SqlServerLogSaver и ElasticsearchLogSaver появляется два вида клиентов: старые и новые. При этом, реализовать интерфейс следует явным образом (explicit interface implementation), что позволит изолировать данных клиентов друг от друга.

public class SqlServerLogSaver : ILogSaver  
{  
 public void Save(DateTime dt, string severity, string message) {...}  
 public void SaveException(DateTime dt, Exception e) {...}  
  
 // Явная реализация интерфейса!  
 ILogSaver.Save(LogEntry logEntry)  
 {  
 var exception = logEntry as ExceptionLogEntry;  
 if (exceptiony != null)  
 {  
 SaveException(exception.EntryDateTime, exception.Exception);  
 }  
 else  
 {  
 Save(logEntry.EntryDateTime, logEntry.Severity.ToString(),   
 logEntry.Message);  
 }  
 }  
}

Листинг 4.2 - Явная реализация интерфейса ILogSaver

Явная реализация интерфейсов позволяет вызывать методы интерфейса лишь при явном приведении объекта к типу интерфейса. Это значит, что обычные клиенты класса SqlServerLogSaver не будут "видеть" метод Save(LogEntry), а смогут им пользоваться лишь в случае явного приведения типов:

var saver = new SqlServerLogSaver();  
LogEntry logEntry = GetLogEntry();  
saver.Save(logEntry); // Ошибка компиляции  
  
ILogSaver logSaver = saver;  
logSaver.Save(logEntry); // ОК

Листинг 4.3 - Явная реализация интерфейсов для четкого разделения интерфейсов

Еще один пример следования Принципу разделения интерфейсов заключается в использовании методов расширения.

Вместо создания "толстого" интерфейса, которым будет удобно пользоваться всем клиентам, можно выделить базовый интерфейс, а вспомогательные методы реализовать в виде методов расширения. В этом случае базовый интерфейс будет максимально простым, и разные клиенты будут самостоятельно решать, какие методы расширения использовать.

Подобный пример был рассмотрен в главе, посвященной Принципу единственной обязанности. Интерфейс класса LogEntryParser содержит универсальный метод TryParse, пользоваться которым не всегда удобно. Поэтому для тестов был добавлен метод расширения Parse, который всегда возвращает результат и не предназначен для разбора многострочных записей:

// В основном коде приложения  
var parser = new LogEntryParser();  
LogEntry logEntry;  
if (parser.TryParse(stringEntry, out logEntry))  
{  
 SaveEntry(logEntry);  
}  
  
// В тестах  
// Импортируем пространство имен с методом расширения  
var parser = new LogEntryParser();  
var logEntry = parser.Parse(sampleStringEntry);  
Assert.That(logEntry.Severity, Is.EqualTo(Severity.Debug));

Листинг 4.4 - Использование Принципа разделения интерфейсов с методами расширения

**Явная реализация интерфейсов и методы расширения позволяют четко следовать Принципу разделения интерфейсов. Каждый клиент знает лишь о подмножестве операций и не видит лишних деталей, которые ему не интересны.**

## Типичные примеры нарушения ISP

* Метод принимает аргументы производного класса, хотя достаточно использовать базовый класс.
* У класса два или более ярко выраженных вида клиентов.
* Класс зависит от более сложной зависимости, чем нужно: принимает интерфейс провайдера, вместо результатов его работы и т.п.
* Класс зависит от сложного интерфейса, что делает его зависимым от всех типов, используемых в этом интерфейсе.

## Выводы

Принцип разделения интерфейсов является частным случаем управления зависимостями и борьбы со сложностью. Чем проще зависимости класса, тем легче понять, что класс делает, поскольку в голове приходится держать лишь минимальное число ненужных деталей. Чем стабильнее зависимости класса, тем меньше вероятность, что его поведение будет нарушено при внесении изменений в других частях системы.

Принцип разделения интерфейсов лежит на стыке классов или модулей. Глядя на исходный код некоторого класса мы можем сказать, соответствует ли реализация принципу единственной обязанности, принципу замещения Лисков или принципу открыт/закрыт. Но лишь по исходному коду класса или его интерфейса мы не можем судить о том, нарушает ли он принцип разделения интерфейсов или нет. Для этого нам нужно посмотреть контекст его использования: есть ли разные группы клиентов, которые используют его по разному, или нет?

Если класс используется разными клиентами, то это может говорить о слишком большом числе обязанностей, и его нужно упростить. В некоторых случаях, у класса может быть одна обязанность, которая рассматривается клиентами с разных точек зрения. Тогда этот факт нужно сделать явным, путем реализации двух или более интерфейсов.