# Принцип единственной обязанности

**ЦИТАТА:** *"Нельзя объять необъятное"*. Козьма Прутков

**Принцип единственной обязанности (Single-Responsibility Principle, SRP)**: *У класса должна быть только одна причина для изменения*. Роберт Мартин. "Принципы, паттерны и практики гибкой разработки" [Martin2006]

В разработке ПО есть одна неизменная составляющая - это неизбежность изменений. Как бы мы ни старались, как бы не пытались продумать все до мелочей, рано или поздно требования изменятся. Требования меняются из-за исходного недопонимания задачи, из-за изменений во внешнем мире, из-за более точного понимания своих собственных нужд заказчиком, или из-за десятков других причин. На самом деле, важны не столько причины изменений, сколько наши возможности по адаптации системы к новым требованиям и легкость внесения изменений.

Многие разработчики считают, что обобщенные решения являются лучшим способом борьбы с изменениями требований. Полагается, что если заложиться на все возможные сценарии, и предоставить различные точки расширения, то система сможет решать любые задачи пользователя. Но у этого подхода есть несколько серьезных недостатков. Мы (разработчики) очень плохо разбираемся в вопросах предсказания будущего и не знаем, в какую сторону будут изменяться требования в будущем, и в какой плоскости система должна легко расширяться. А поскольку гибкость всегда приводит к увеличению сложности, то полученное решение не всегда справляется с исходной задачей и плохо поддается модификации.

Другой подход к обеспечению расширяемости заключается в использовании наиболее простых решений: программная сущность (класс, модуль, метод) должна решать, по возможности, лишь одну задачу, но делать это хорошо. Как это решает проблему изменений? Очень просто. Чем меньше у метода, класса или модуля вспомогательных задач, тем ниже вероятность "случайных" изменений.

Фредерик Брукс в своей книге "Мифический человеко-месяц" вводит понятия естественной сложности (essential complexity) и привнесенной или случайной сложности (accidental complexity). Естественная сложность исходит из предметной области, и является неотъемлемой частью любого решения. Привнесенная сложность, внесена нами в процессе реализации, за счет плохого дизайна, неудачно выбранных библиотек или неподходящего языка программирования.

Изменения, вносимые в систему, также можно разделить на "естественные", которые возникают из-за изменения бизнес-логики или требуемого поведения, и "случайные" - которые мы вынуждены вносить во второстепенные модули из-за неудачного дизайна. Изменения в законодательстве приведут к изменениям в классе вычисления заработной платы и являются "естественными". Но мы не хотим его менять при изменении пользовательского интерфейса, смене базы данных, или из-за изменений формата одного из отчетов.

Существует ряд патологических случаев нарушения принципа единственной обязанности. Классы пользовательского интерфейса, которые знают о бизнес-правилах или работают напрямую с базой данных, или классы Windows-сервисов c обилием бизнес-логики. Есть примеры нарушения SRP на уровне приложений: Windows Forms приложение, в котором располагается WCF сервис; Windows-сервис, взаимодействующий с пользователем с помощью диалоговых окон. Эти примеры показывают, что нарушения SRP бывают как на микро уровне - на уровне классов или методов, так и на макро уровне - на уровне модулей, подсистем и целых приложений.

Приведенные выше примеры не столько нарушают SRP, сколько противоречат здравому смыслу. В реальном мире проблемы бывают более тонкими, когда один разработчик говорит, что дизайн хорош, а для другого он серьезно «попахивает» и вам хочется найти весомые аргументы против текущего решения. Чтобы понять, нарушает ли код принцип единственной обязанности важно понимать, какую проблему он должен решать.

## Для чего нужен SRP?

**Принцип единственной обязанности предназначен для борьбы со сложностью**. Когда в приложении всего 200 строк, то дизайн как таковой вообще не нужен. Достаточно аккуратно написать 5-7 методов и решить задачу любым доступным способом. Проблемы возникают, когда система растет и увеличивается в масштабах.

Зависимость между числом строк кода и сложностью решения является не линейной. Добавление каждой новой функции в систему требует все больше и больше усилий. Когда речь касается десятков и сотен тысяч строк кода, то приходится вспоминать такие "страшные" понятия, как "абстракция" и "сокрытие информации", и лучше продумывать обязанности каждого класса. В крупной системе очень важно иметь возможность сосредоточиться на главной задаче метода, класса или модуля, и отбросить из рассмотрения все второстепенные детали.

Основным строительным блоком объектно-ориентированного приложения является класс, поэтому обычно принцип единственной обязанности рассматривается в контексте класса. Но, поскольку основную работу выполняют методы, то очень важно, чтобы они также были нацелены на решение одной задачи.

Основная сложность принципа SRP в том, что понятие "обязанности" является относительным. Если мы говорим, что у класса или метода должна быть лишь одна обязанность, то может ли метод валидировать свои аргументы? Или логировать определенные этапы своей работы? А может ли класс читать и сохранять данные?

Наличие или отсутствие нарушения SRP очень сильно зависит от того, насколько сложным является каждый из описанных выше шагов. Метод будет нарушать SRP если валидация аргументов занимает 40 строк кода и находится в разных его частях. Метод также будет нарушать SRP, если за обилием трассировочных сообщений не видно его основной логики. Но класс может и не нарушать SRP, если он читает и сохраняет данные, но на каждую операцию требуется 2 строки кода.

## Принцип единственной обязанности на практике

Развитие программного проекта приводит к увеличению хаоса. Спешка, непродуманные решения, не полное понимание намерений оригинального решения, все это приводит к постепенному ухудшению качества дизайна. При внесении изменений очень важно оценивать качество решения на предмет его чистоты, соответствия принципам проектирования и здравому смыслу. То, что на первых этапах должно было находится в одном классе, должно быть перемещено в совершенно другое место, а логика, которая помещалась в одном методе, может перерасти в целую иерархию классов.

Приведенная в предыдущих главах задача импорта лог-файлов для полнотекстового поиска является вполне реальной. Наша команда разработала утилиту для импорта лог-файлов в Elasticsearch, с целью изучения узких мест производительности и быстрого поиска определенных сообщений. Самая первая версия этой утилиты предназначалась для экспорта логов определенного приложения и была реализована за несколько часов. Это было простое консольное приложение в пару сотню строк, и главный его класс выглядел примерно так:

public class LogImporter  
{  
 public void ImportLogs()  
 {  
 string[] logFileNames = GetListOfFilesFromAppConfig();  
 foreach (var file in logFileNames)  
 {  
 var logEntries = ReadLogEntries(file);  
 SaveLogEntries(logEntries);  
 }  
 }  
  
 private string[] GetListOfFilesFromAppConfig()  
 {  
 // Читаем список файлов из конфига приложения  
 }  
  
 private IEnumerable<string> ReadLogEntries(string fileName)  
 {  
 // Читаем файл построчно  
 }  
  
 private void SaveLogEntries(IEnumerable<string> entries)  
 {  
 foreach (var entry in entries)  
 {  
 DateTime dateTime = ParseEntryTime(entry);  
 // Сохраняем для полнотестового поиска  
 LogSaver.SaveEntry(dateTime, entry);  
 }  
 }  
  
 private DateTime ParseEntryTime(string entry)  
 {  
 // Получаем время из лог-файла, зная его формат  
 }  
}

Листинг 1.1 - Первая реализация утилиты импорта лог-файлов

Класс LogImporter полностью отвечает за процесс импорта лог-файлов. Он читал конфигурационный файл приложения для получения списка импортируемых файлов, анализировал прочитанные строки и сохранял записи в очень простом формате с помощью фасадного класса LogSaver.

Насколько это решение удачное? Все зависит от поставленных целей. Если требуется одноразовая утилита, для анализа логов "продакшн" сервера в воскресенье вечером, то это решение вполне оправдано. Но оно совершенно не подходит, если поставлена цель разработать полноценную утилиту, способную экспортировать логи разного формата, или, хотя бы, учитывать важность (severity) записей и обрабатывать сохраненные исключения особым образом.

Давайте посмотрим, как следование принципу единственной обязанности может улучшить дизайн этого кода. Вот как я подхожу к внесению изменений.

**1. Убираем ненужные обязанности**

Первое, что нужно сделать, это найти обязанности, которые можно переложить на плечи вызывающего кода. В данном случае - это чтение конфигурации. Работа с конфигурационным файлом приложения делает этот код менее автономным, затрудняет юнит-тестирование и повторное использование.

public class LogImporter  
{  
 private readonly ICollection<string> \_logFileNames;  
  
 public LogImporter(ICollection<string> logFileNames)  
 {  
 \_logFileNames = logFileNames;  
 }  
  
 public void ImportLogs()  
 {  
 foreach (var file in \_logFileNames)  
 {  
 var logEntries = ReadLogEntries(file);  
 SaveLogEntries(logEntries);  
 }  
 }  
   
 // Остальные методы остались без изменения  
}

Листинг 1.2 - Улучшенная версия класса LogImporter

Этот шаг кажется незначительным, но он очень важен. Формирование списка анализируемых файлов является ответственностью вызывающего кода, и не должно контролироваться классом LogImporter. Новое решение является более простым и гибким одновременно. Так, например, теперь нам не придется изменить код этого класса, если список анализируемых файлов будет передаваться из командной строки, а не читаться из конфигурации.

**ПРИМЕЧАНИЕ**  
Некоторые читател могут сказать, что выделение Стратегии конфигурирования тоже решило бы поставленную задачу. Выделение интерфейса ILogFileListProvider позволит иметь несколько разных реализаций, и обеспечит требуемую гибкость. Но такой подход значительно сложнее, обладает дополнительным уровнем косвенности и требует от клиентов класса LogImporter значительно большего числа усилий. Если классу для своей работы требуется коллекция строк, то отразите это требование в своем решении наиболее простым и ясным способом!

**2. Выделяем класс разбора записей лог-файла**

У класса может быть несколько ответственностей, пока все они являются относительно простыми. Какой аспект класса LogImporter является самым сложным? Разбор прочитанных записей! У этого аспекта множество граничных условий: разные форматы даты/времени, разные уровни важности сообщений, многострочные записи с исключениями и т.п. Даже для анализа лог-файлов одного приложения, сложность этого класса будет достаточно высокой.

Все это говорит о необходимости выделения класса LogEntryParser:

class LogEntryParser  
{  
 public bool TryParse(string line, out LogEntry logEntry)  
 {  
 // Используем регулярное выражения для анализа содержимого строки.  
 // Метод возвращает true, если запись полностью прочитана.  
 // Возвращает false, если мы столкнулись с многострочной записью,  
 // и для получения записи нужно проанализировать еще одну или более строку  
 }  
}  
  
public class LogImporter  
{  
 private readonly LogEntryParser \_parser = new LogEntryParser();  
  
 public void ImportLogs()  
 {  
 foreach (var file in \_logFileNames)  
 {  
 IEnumerable<LogEntry> logEntries = ReadLogEntries(file);  
 SaveLogEntries(logEntries);  
 }  
 }  
   
 private IEnumerable<LogEntry> ReadLogEntries(string fileName)  
 {  
 // Читаем файл построчно  
 using (var file = File.OpenText(fileName))  
 {  
 string line = null;  
 while ((line = file.ReadLine()) != null)  
 {  
 LogEntry logEntry;  
 if (\_parser.TryParse(line, out logEntry))  
 {  
 yield return logEntry;  
 }  
 }  
 }  
 }  
 // Остальные методы остались без изменения  
}

Листинг 1.3 - Выделенный класс LogEntryParser

Наличие класса LogEntryParser имеет ряд важных преимуществ: \* Сложная логика анализа записей изолирована и может развиваться независимо. \* Логику анализа записей легко тестировать в изоляции путем передачи строк нужного формата. \* Класс LogImporter превратился в довольно простого Посредника, который отвечает за пересылку данных из одного источника в другой.

**3. Тестирование кода**

Многие разработчики считают, что каждый аспект реализации должен быть покрыт юнит-тестами. Такое стремление не во всех случаях оправдывает затраченные усилия, а иногда, приводит к ложному чувству безопасности. Более разумным является смешанный подход к тестированию, в котором наиболее сложные аспекты покрывается юнит-тестами, а вспомогательная логика покрывается интеграционными тестами.

В процессе работы над этой задачей, я бы начал с тестов класса LogEntryParser:

[TestFixture]  
public class LogEntryParserTests  
{  
 [TestCase("2014-01-12 [DEBUG] message", Result = Severity.Debug)]  
 [TestCase("[Info] Message", Result = Severity.Info)]  
 public Severity ParseSeverity(string line)  
 {  
 // Arrange  
 var parser = new LogEntryParser();  
  
 // Act & Assert  
 return parser.Parse(line).Severity;  
 }  
}  
  
static class LogEntryParserEx  
{  
 public static LogEntry Parse(this LogEntryParser parser, string line)  
 {  
 LogEntry logEntry;  
 parser.TryParse(line, out logEntry);  
 return logEntry;  
 }  
}

Листинг 1.4 - Юнит-тесты класса LogEntryParser

Наличие у класса LogEntryParser четкого интерфейса с минимальным числом зависимостей делает его тестирование очень простым. Использование параметризованных тестов NUnit (\*) обеспечивают высокое покрытие кода тестами с минимальным числом усилий, а добавление нового тест-кейса требует добавление лишь одной строки кода.

(\*) Подробнее о пользе параметризованных тестов можно почитать в моей статье "Параметризованные юнит тесты": http://sergeyteplyakov.blogspot.com/2012/08/blog-post\_28.html.

**ПРИМЕЧАНИЕ** Обратите внимание на метод расширения Parse класса LogEntryParser. Качество тестов не менее важно, чем качество основного кода. Интерфейс класса LogEntryParser специализирован для решения своих основных задач, но он менее удобный для других сценариев. Использование методов расширения для тестов показывает применение еще одного принципа проектирования под названием Принцип разделения интерфейсов.

Интеграционные тесты показывают корректность реализации в реальном окружении. В данном случае, класс LogImporter работает с двумя внешними источниками: файлами и хранилищем лог-файлов. Поскольку его логика весьма проста, то вместо выделения дополнительных уровней косвенности для юнит-тестирования, будет достаточно нескольких интеграционных тестов:

[TestFixture]  
public class LogImporterIntegrationTests  
{  
 [Test]  
 public void ImportFileWithTenEntries()  
 {  
 string fileName = "10Entry.log";  
  
 // Cleanup  
 TryRemoveExistingEntries(fileName);  
  
 // Arrange  
 var importer = new LogImporter(new[] {fileName});  
   
 // Act  
 importer.ImportLogs();  
  
 // Assert  
 Assert.That(GetEntriesCountForLogFile(fileName), Is.EqualTo(10));  
 }

Листинг 1.5 - Интеграционные тесты класса LogImporter

Интеграционные тесты более хрупкие по своей природе, и требуют особого подхода при реализации. К трем основным этапам: инициализация, действие, утверждение (Arrange, Act, Assert, AAA), может быть добавлен еще один - очистка состояния. Это предотвратит прохождение теста из-за наличия в хранилище старых данных, хотя и потребует дополнительного времени при каждом запуске.

### Подводим итоги

Использование принципа единственной обязанности привело нас к следующему дизайну класса LogImporter (рис. 1.1):

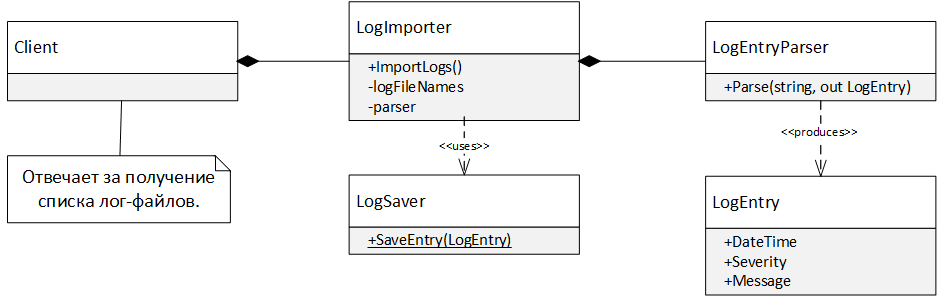


Рисунок 1.1 - Переработанный дизайн класса LogImporter

Нельзя сказать, что каждый класс этого решения имеет лишь одну обязанность, но ведь следование принципам проектирования не должно быть самоцелью. Сложность дизайна определяется сложностью решаемой задачи. На данный момент, класс LogImporter наиболее простым образом решает поставленную перед ним задачу: импортирует лог-файлы определенного приложения.

Полученные дизайн является достаточно простым, без ненужных уровней абстракции или интерфейсов. В последующих главах мы рассмотрим использование других принципов проектирования, которые помогут справиться с последующими изменениями требований.

## Типичные примеры нарушения SRP

* *Смешивание логики с инфраструктурой*. Бизнес-логика смешана с представлением, слоем персистентности, находится внутри WCF или windows-сервисов. Должна быть возможность сосредоточиться на бизнес-правилах, не обращая внимания на второстепенные инфраструктурные детали.
* *Слабая связность (low cohesion)*. Класс/модуль/метод не является цельным и решает несколько несвязанных задач. Проявляется несколько групп методов, каждая из которых обращается к подмножеству полей, которые не используются другими методами.
* *Выполнение нескольких несвязанных задач*. Класс/модуль может быть цельным, но решать несколько несвязанных задач (вычисление заработной платы и построение отчета). Класс/модуль/метод должен быть "сфокусированным" на решении минимального числа задач.
* *Решение задач разных уровней абстракции*. Класс/метод не должен отвечать за задачи, разного уровня. Например, класс удаленного заместителя не должен самостоятельно проверять аргументы, заниматься сериализацией и шифрованием. Каждый из этих аспектов должен решаться отдельным классом.

## Выводы

Следование принципам проектирования является не статической, а динамической характеристикой дизайна. Наибольшая опасность заключается в "загнивании" дизайна, когда внесение нескольких изменений приводит к разрастанию обязанностей и увеличению сложности. То, что вчера казалось лишь одной обязанностью, сегодня может потребовать целой иерархии классов. Каждый раз, при изменении логики, нужно анализировать дизайн на соответствие здравому смыслу и принципам проектирования.

Важность принципа единственной обязанности резко возрастает при увеличении сложности. Если решение перестает помещаться в голове, то пришло время разбить его на более простые составляющие, каждая из которых будет решать лишь одну задачу.