# Принцип инверсии зависимостей

**Цитата**: *Когда душа уходит в пятки, встань вверх ногами и встряхнись!*

**Принцип инверсии зависимостей (Dependency Inversion Principle – DIP)**:

* Модули верхнего уровня не должны зависеть от модулей нижнего уровня. И те и другие должны зависеть от абстракций.
* Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций. Роберт Мартин. "Принципы, паттерны и практики гибкой разработки" [Martin2006]

Принцип инверсии зависимостей – один из самых известных сегодня принципов проектирования, который лежит в основе популярных техник внедрения зависимостей (Dependency Injection). Однако, если посмотреть лишь на его название и описание, то будет довольно сложно понять, что же он означает. А если спросить простых обывателей о том, как они понимают этот принцип, то они начнут что-то говорить о пользе интерфейсов и абстракций, и, вообще, будут "путаться в показаниях".

## Интерфейсы

В основе принципа инверсии зависимостей лежит идея использования интерфейсов (\*). Одна группа классов реализует некоторый набор интерфейсов, а другая группа классов принимает эти интерфейсы в качестве аргументов конструктора:

(\*) Сноска: под интерфейсами в данном случае понимаются программные конструкции, объявленные с помощью ключевого слова interface. В большинстве случаев, вместо интерфейсов возможно использование абстрактных классов, и все рассуждения данной главы еще будут актуальными.

interface IFileReader  
{  
 string ReadLine();  
}  
  
class LogEntryParser  
{  
 public LogEntryParser(IFileReader fileReader)  
 {}  
   
 public IEnumerable<LogEntry> ParseLogEntries()   
 {}  
}  
  
class FileReader : IFileReader {...}

Листинг 5.1 - Пример наивного использования DIP

Использование интерфейсов приводит "слабосвязанному" (loosely couplied) дизайну, поскольку класс LogEntryParser знает лишь об интерфейсе IFileReader, и не знает о конкретной реализации этого интерфейса. А следование Принципу замещения Лисков позволит заменить одну реализацию на другую, и получить гибкое решение, соответствующее принципу Открыт/Закрыт.

И хотя данные рассуждения вполне логичны, не стоит забывать, что у такого решения есть и обратная сторона. Наличие интерфейсов образует "дополнительный уровень косвенности" (или "дополнительный уровень абстракции"), что затрудняет понимание системы. Полиморфизм - это GOTO объектного мира, который делает решение более гибким ценой увеличения сложности. Понять логику исполнения лишь путем чтения кода становится довольно сложно, поскольку конкретный тип наследника определяется во время исполнения.

В нашем случае, клиентам класса LogEntryParser теперь приходится решать проблему с поиском подходящей зависимости, даже если им это не интересно. Только представьте себе, что у класса List<T> появилась бы зависимость вида IListGrowingPolicy, которая бы отвечала за способ "роста" списка, а у класса String появилась бы зависимость IInterningPolicy, которая бы отвечала за политику интернирования! Подобные решения обеспечивают гибкость не там, где нужно, и могут подрывать инкапсуляцию.

Не для всех классов нужно выделять интерфейсы, и не все зависимости следует требовать извне в виде интерфейсов. Принцип инверсии зависимостей достаточно четко дает понять, когда нужно зависимость инвертировать, а когда можно создавать зависимости непосредственно в месте их использования.

## Слои

У лука есть слои, у торта есть слои, и у Людоеда есть слои. Шрек

Любое современное приложение разбито на слои, каждый из которых отвечает за определенный аспект поведения. На нижних уровнях находятся повторно используемые компоненты и инфраструктурный код, а слои более высокого уровня отвечают за логику приложения и пользовательский интерфейс (Larman2006):

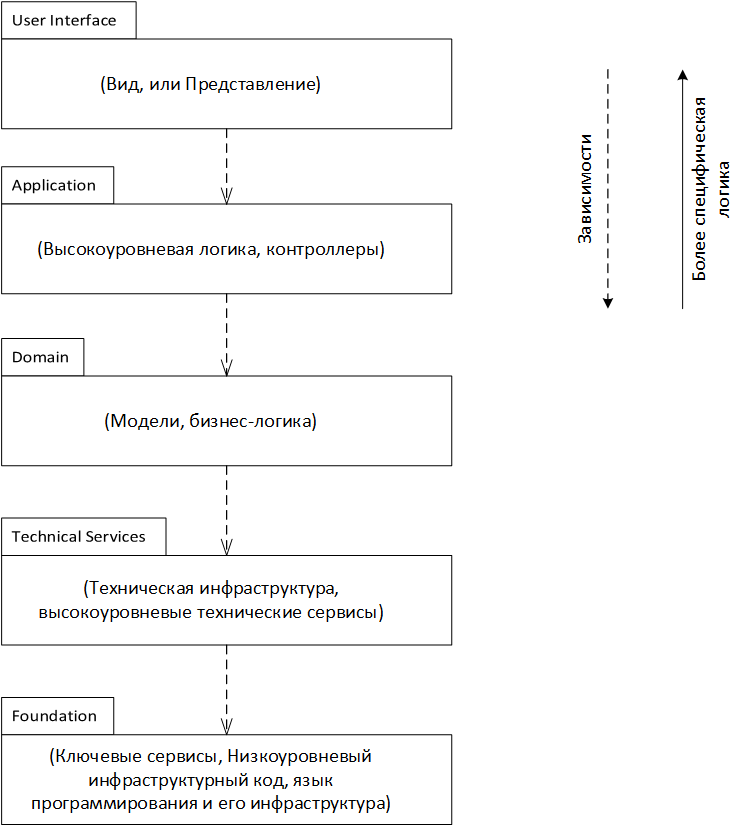


Рисунок 5.1 - Многоуровневая архитектура современного приложения

Каждый слой отвечает за определенную область и использует сервисы нижележащих уровней для решения своих задач. Принцип единственной обязанности говорит, что каждый класс, модуль или слой должен решать лишь одну задачу. Это значит, что инфраструктурный код доступа к данным не должен содержать бизнес-логики, а бизнес-логика не должна знать о пользовательском интерфейсе.

Слои нижнего уровня не знают и не должны знать о слоях верхнего уровня. Это позволяет использовать низкоуровневые слои повторно, упрощает понимание и развитие каждого из них, а также ограничивает распространение изменений.

Чем выше слой, тем специфичнее он для каждого конкретного приложения. На нижних уровнях находятся инфраструктурный код, сервисы и ключевая бизнес-логика (core business logic). На верхних уровнях находятся высокоуровневая бизнес-логика, которая отличает одно приложение от другого, а также пользовательский интерфейс.

Иерархичность и "слоеность" присуща как программной системе в целом, так и отдельным крупным модулям. По мере роста сложности предметной области или за счет появления дополнительных деталей, компоненты начинают дробиться на более мелкие составляющие, в основе которых будут лежать уже примитивные типы платформы и языка программирования.

Сложные системы часто являются иерархическими и состоят из взаимозависимых подсистем, которые в свою очередь также могут быть разделены на подсистемы, и т.д., вплоть до самого низкого уровня. Гради Буч "Объектно-ориентированный анализ и проектирование с примерами приложений"

Классы более высокого уровня обычно контролируют процесс создания и передачу управления классам нижних уровней. В предыдущей главе мы остановились на следующем дизайне класса LogImporter (рис. 5.2). Класс LogImporter создает экземпляр одного из наследников класса LogEntryParser, и "просит" его проанализировать записи, прочитанные из файла. В некоторых случаях, класс верхнего уровня не знает, какой точно класс нижнего уровня использовать. В этом случае выделяется Фабричный Метод, который "прячет" процесс создания экземпляра нужной Стратегии (\*).

(\*) Сноска: Паттерн Стратегия был рассмотрен в Главе 1 первой части, а Фабричный метод - в Главе 3 второй части этой книги.

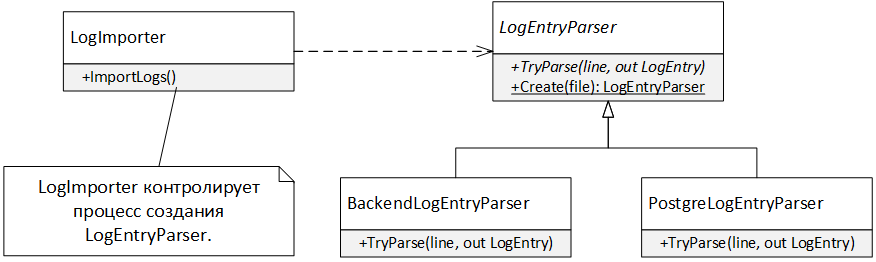


Рисунок 5.2 - Высокоуровневый дизайн класса LogImporter

Однако, иногда возникает необходимость обеспечить передачу управления не только сверху вниз по слоям приложения, но и снизу вверх. Классы бизнес логики могут потребовать что-то от слоя пользовательского интерфейса. Инфраструктурному коду может потребоваться информация, недоступная на его уровне абстракции. Иногда классы вынуждены "зависеть" от чего-то, что находится вне поля их зрения.

## Наблюдатели

TODO: нужна обратная связь по этому разделу! Вполне вероятно, его нужно пристрелить! Размышления полезны, но их слишком много и они могут увести от цели главы!

Классическим решением задачи связи нижних слоев с верхними является паттерн Наблюдатель (\*). Самый простой способ отвязать класс от внешних зависимостей - добавить в него событие (event), с помощью которого он будет уведомлять всех заинтересованных подписчиков об изменении состояния. При этом логически, управление будет передано с нижнего уровня на верхний, однако знать о том, "что" будет происходить при вызове этого делегата, текущий класс не будет.

(\*) Сноска: Более подробно паттерн Наблюдатель был рассмотрен в Главе 5 первой части книги.

Хорошим примером использования наблюдателей для передачи управления снизу вверх по слоям приложения является паттерн MVC (Model-View-Controller, Модель-Представление-Контроллер). Каждый класс в этом паттерне отвечает за отдельный аспект: Модель определяет бизнес правила, Вид отвечает за пользовательский интерфейс, а Контроллер обрабатывает входные события от пользователя и отвечает за логику приложения.

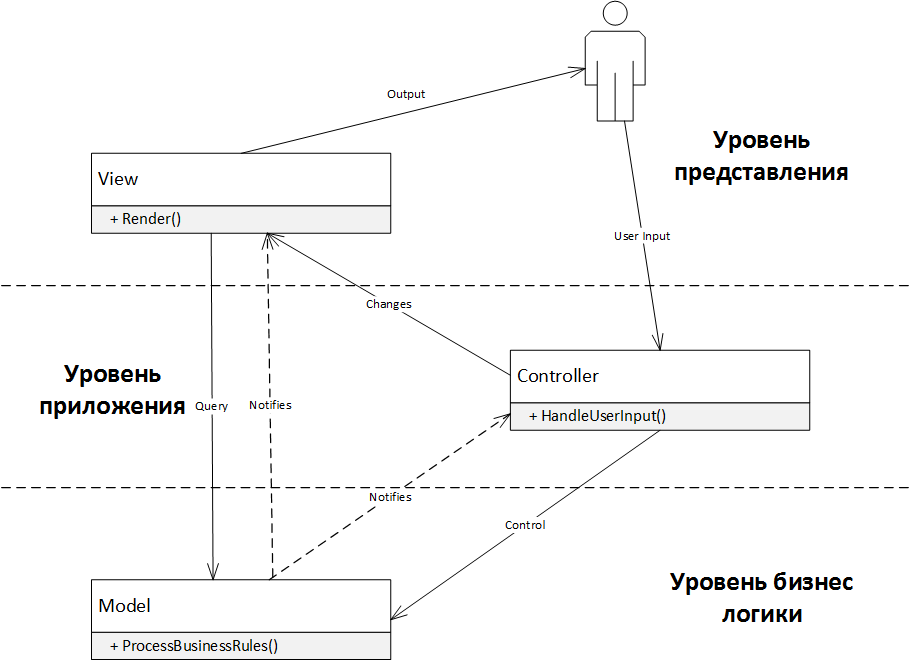


Рисунок 5.3 - Паттерн Модель-Представление-Контроллер (\*)

(\*) СНОСКА: Распределение модели, контроллера и представления по слоям приложения может быть неоднозначным. Что считать верхним слоем, а что нижним? Согласно общепринятой классификации, описанной Крэгом Ларманом в его книге "Применение UML и шаблонов проектирования", самым верхним слоем приложения является пользовательский интерфейс. За ней следом идет логика приложения, которая связывает слой пользовательского с доменными объектами. Именно этим объясняется порядок слоев на данном рисунке.

Классы моделей могут требовать что-то от вышестоящих слоев. Но поскольку слои нижнего уровня не должны знать напрямую о классах верхнего уровня, то данное "общение" осуществляется с помощью наблюдателей. Так, Модель не знает ничего о Контроллере и Представлении и общается с ними опосредованно с помощью событий. Паттерн MVC несколько нарушает данное правило и Контроллер обычно знает о пользовательском интерфейсе больше, чем нам бы того хотелось. Контроллер также обрабатывает входные данные от пользовательского интерфейса, что делает этот паттерн широко используемым в веб-приложениях, но практически не применимым в других видах приложений, например, WPF или Windows Forms.

Чтобы справиться с этими особенностями, возник другой паттерн, под названием - MVP (Model-View-Presenter, Модель-Представление-Презентер).

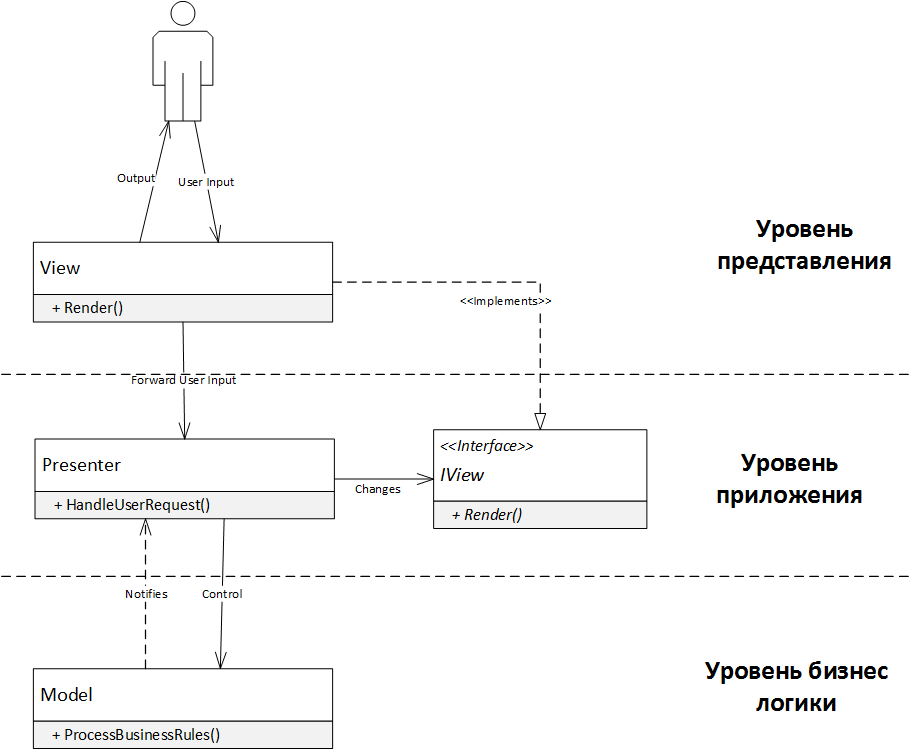


Рисунок 5.4 - Паттерн Модель-Представление-Презентер

Главное отличие между паттернами MVP и MVC в том, что Презентер не обрабатывает входные данные пользователя, а также то, как он взаимодействует с Представлением. Презентер знает лишь об абстрактном представлении (IView), интерфейс которого объявляется на его уровне (например, в той же сборке), а реализуется - уровнем представления.

Помимо двух вышеперечисленных паттернов, существует еще один паттерн из семейства MVx - MVVM (Model-View-View Model, Модель-Представление-Модель представления). MVVM похож на MVP, однако вместо интерфейсов IView Модель представления "общается" с верхним уровнем с помощью событий, спрятанных в инфраструктурном коде за технологией привязки данных (Data Binding).

## Для чего нужен принцип инверсии зависимостей

**Принцип инверсии зависимостей предназначен для устранения прямых связей между классами или модулями с зависимостями более высокого уровня.**

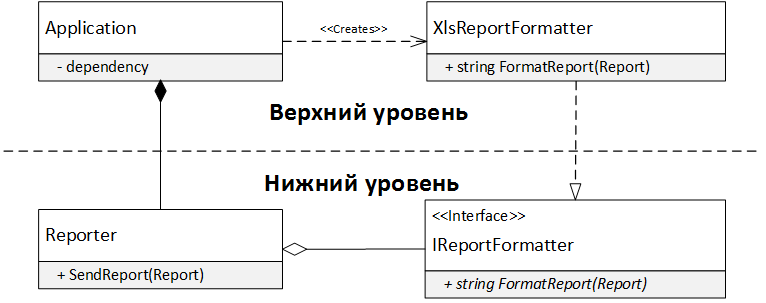


Рисунок 5.5 - Принцип инверсии зависимостей

Название принципа отражает нетипичность направления зависимостей: классы нижнего уровня определяют некоторый "контракт", которому должны следовать классы верхнего уровня. Классы верхнего уровня вынуждены выступать в роли "адаптеров" и подстраиваться под протокол, определенный на уровне ниже. При этом вместо "слабо типизированных" наблюдателей, текущий класс выделяет именованную зависимость в виде интерфейса, и требует ее в своем конструкторе от вышестоящего кода.

// Reporter.dll - Нижний уровень  
public interface IReportFormatter  
{  
 string FormatReport(Report report);  
}  
  
public class Reporter  
{  
 private readonly IReportFormatter \_formatter;  
  
 public Reporter(IReportFormatter formatter)  
 {  
 \_formatter = formatter;  
 }  
  
 public void SendReport(Report report)  
 {  
 var formattedReport = \_formatter.FormatReport(report);  
 SendFormattedReport(formattedReport);  
 }  
  
 private void SendFormattedReport(string formattedReport)  
 {}  
}  
  
// Application.dll - Верхний уровень  
class XlsFormatter : IReportFormatter  
{  
 public string FormatReport(Report report)  
 {  
 // Форматируем отчет для отображения его в Excel  
 }  
}  
  
class Application  
{  
 public void Run()  
 {  
 var reporter = new Reporter(new XlsFormatter());  
 reporter.SendReport(GenerateReport());  
 }  
}

Листинг 5.2 - Использование DIP для взаимодействия разных уровней приложения

Класс Reporter определяет свои зависимости в виде интерфейса IReportFormatter, а класс Application адаптируется под его требования. Поэтому интерфейс IReportFormatter определяется на том же уровне, что и Reporter, а реализация этого интерфейса находится на более высоком уровне - уровне приложения. В результате прямые зависимости направлены сверху вниз - от приложения к модулю отчетов, а косвенные зависимости направлены в обратном направлении, что позволяет классу Reporter взаимодействовать во время исполнения с XlsFormatter полиморфным образом.

## Остерегайтесь неправильного понимания DIP

Чрезмерное или необдуманное использование любого принципа проектирования может привести к переусложнению дизайну и принцип инверсии зависимостей здесь не исключение. Распространение библиотек управления зависимостями - IoC-контейнеров, (\*) может привести к выделению чрезмерного количества интерфейсов, что делает решение настолько "слабосвязным", что разобраться в нем становится практически невозможно.

(\*) СНОСКА: IoC-контейнеры (Inversion of Control Containers) представляют собой библиотеки для автоматического управления зависимостями. Их главным действующим лицом является контейнер, который может создавать экземпляры любого типа в приложении. Для этого контейнер конфигурируется в точке входа (Entry Point) приложения путем задания ассоциации между интерфейсами и их реализацией. Теперь, если пользователь "попросит" у контейнера экземпляр некоторого типа, то контейнер самостоятельно "найдет" все его зависимости и передаст их требуемому классу через аргументы конструктора, или установит соответствующие свойства. Подробнее о них можно почитать в книге Марка Сиимана "Dependency Injection in .NET".

Причина неправильного использования принципа инверсии зависимостей кроется в недопонимании его целей, а также в описании этого принципа его автором - Робертом Мартином. Вот что он пишет в своей книге [Martin2006]:

DIP выражается простым эвристическим правилом: «Зависеть надо от абстракций». Оно гласит, что не должно быть зависимостей от конкретных классов; все связи в программе должны вести на абстрактный класс или интерфейс.

* Не должно быть переменных, в которых хранятся ссылки на конкретные классы.
* Не должно быть классов, производных от конкретных классов.
* Не должно быть методов, переопределяющих метод, реализованный в одном из базовых классов.

И далее:

Конечно, эта эвристика хотя бы раз да нарушается в любой программе. … В большинстве систем класс, описывающий строку, конкретный. Такой класс изменяется редко, поэтому в прямой зависимости от него нет никакой беды. Однако конкретные классы, являющиеся частью прикладной программы, которые пишем мы сами, в большинстве случаев изменчивы. Именно от таких конкретных классов мы и не хотим зависеть напрямую. Их изменчивость можно изолировать, скрыв их за абстрактным интерфейсом.

Буквальное следование принципу DIP по такому описанию чревато серьезными последствиями для дизайна. В последнее время легко столкнуться с проблемой чрезмерно "абстрактных" решений, когда интерфейсов слишком много, и решение настолько "гибкое" и "слабосвязное", что просто невозможно понять, кто за что отвечает и откуда начать изучение системы.

### Тестируемость решения vs. Подрыв инкапсуляции

Сегодня выделение интерфейсов часто объясняют необходимостью юнит-тестирования. Дескать, если класс создает свои зависимости самостоятельно, то как же мы сможем протестировать его в изоляции? Если же класс принимает все свои зависимости извне, да еще и в виде интерфейсов, то мы сможем "замокать" (\*) все, что угодно и добиться 100% покрытия тестами!

(\*) Существует несколько видов объектов-подделок, используемых в юнит-тестах, которые объединяются общим названием "test doubles". Моки (mocks) и стабы (stubs) являются двумя наиболее распространенными видами подделок, которые предназначены для проверки поведения и эмуляции состояния. Далеко не все изоляционные фреймворки разделяют эти два понятия, поэтому многие из них (такие, как Moq), любые подделки называют моками (mocks), что сделало этот термин применимым для описания любых подделок. Подробнее о разнице между стабами и моками можно прочитать в моей статье "Programming Stuff: Стабы и Моки" (bit.ly/StubsVsMocks)

Проблема такого подхода в том, что он легко может подорвать инкапсуляцию текущего класса! Если зависимость находится на более низком уровне абстракции, то вызывающему коду она будет не интересна. Пользовательскому интерфейсу не важно, как именно реализована модель, и какая именно инфраструктура используется двумя слоями ниже. А слою бизнес-логики вряд ли интересны подробности реализации инфраструктурного слоя для удаленного взаимодействия с клиентами. С "колокольни" высокого уровня бывает просто невозможно решить, что же нужно передать в качестве IPartitioningStrategy нашей модели, и почему вообще меня это должно интересовать?

Исходный смысл принципа инверсии зависимостей в том, чтобы классы нижнего уровня взаимодействовали с верхним уровнем косвенно, ничего не зная о нем конкретного. Но он совершенно не означает, что теперь классы верхнего уровня должны знать о всех внутренних проблемах нижележащих слоев.

Существует несколько причин, почему класс может требовать зависимость в виде аргумента конструктора или метода: \* Реализация зависимости находится на более высоком уровне (следование принципу DIP). \* Существует множество реализаций зависимости и класс на этом уровне не может решать, какой из них выбрать (использование паттерна Стратегия и следование принципу Открыт-Закрыт). \* Поведение зависимости может быть завязано на внешнее окружение - файлы, базы данных, сокеты.

В первых двух случаях класс самостоятельно не может решить, какой конкретный тип зависимости использовать, поэтому просит "верхний" уровень помочь ему в этом. Последний случай более специфичен. Иногда мы можем осознанно пожертвовать инкапсуляцией в угоду тестируемости, особенно при наличии сложной логики в текущем классе и невозможности разделить его на более простые составляющие для тестирования их в изоляции.

## Принцип инверсии зависимостей на практике

Давайте вернемся к примеру с классом LogEntryParser и интерфейсом IFileReader (листинг 5.3):

class LogEntryParser  
{  
 private readonly IFileReader \_fileReader;  
  
 public LogEntryParser(IFileReader fileReader)  
 {  
 \_fileReader = fileReader;  
 }  
  
 public IEnumerable<LogEntry> ParseLogEntries()  
 {  
 // Читает файл с помощью \_fileReader  
 // и разбирает прочитанные строки  
 yield break;  
 }  
}

Листинг 5.3 - Наивная реализация класса LogEntryParser

Код выглядет весьма неплохо, но отвечает ли он принципам проектирования?

* Класс нарушает SRP. Данный парсер умеет разбирать множество строк, что является дополнительной ответственностью. Поскольку одной записи LogEntry может соответствовать несколько строк лог файла (например, для исключений), такой подход может показаться более разумным. Я же предпочитаю, чтобы интерфейс класса был максимально простым.
* Класс нарушает ISP (\*). Самой простой зависимостью класса LogEntryParser является строка (System.String), и именно ее должен принимать метод Parse в качестве аргумента метода.

(\*) Сноска: здесь происходит очень тонкое нарушение принципа ISP. С точки зрения этого принципа, класс должен обладать самыми простыми из возможных зависимостей. Самой простой и необходимой зависимостью класса LogEntryParser является строка, которую необходимо проанализировать, а не зависимости, такие как IFileReader.

* Класс нарушает DIP. Класс LogEntryParser использует IFileReader, интерфейс, который нельзя назвать "высокоуровневой зависимостью". Первое, что должно приходить на ум при попытке "абстрагироваться" от ввода/вывода - это тип System.Stream. Если он, по какой-то причине не подходит, то вместо интерфейса IFileReader, нужно выделять интерфейс IRawLogEntryStream. IRawLogEntryStream позволит работать с любым источником "сырых" записей, будь-то файл, сетевой поток или что-то еще.

Давайте посмотрим, как я бы изменил этот дизайн.

Поведение класса LogEntryParser не точно отражало его название. Если нам нужен класс или интерфейс, который отвечает за чтение и разбор записей, то ему нужно подходяще имя. Например, LogEntryReader (листинг 5.4):

class LogEntryReader : IDisposable  
{  
 private readonly Stream \_stream;  
 private readonly LogEntryParser \_parser;  
  
 public LogEntryReader(Stream stream, LogEntryParser parser)  
 {  
 \_stream = stream;  
 \_parser = parser;  
 }  
  
 public void Dispose()  
 {  
 \_stream.Close();  
 }  
  
 public IEnumerable<LogEntry> Read()  
 {  
 using (var sr = new StreamReader(\_stream))  
 {  
 string line = null;  
 while ((line = sr.ReadLine()) != null)  
 {  
 LogEntry logEntry;  
 if (\_parser.TryParse(line, out logEntry))  
 {  
 yield return logEntry;  
 }  
 }  
 }  
 }

Листинг 5.4 - Реализация класса LogEntryReader

Теперь, ответственности классов разделены более четко. Класс LogEntryReader отвечает за чтение прочитанных записей, при этом за разбор записей отвечает класс LogEntryParser и его наследники.

Мы можем пойти еще дальше, и создать фасадный фабричный метод, который будет создавать экземпляр LogEntryReader для чтения конкретного файла:

public static LogEntryReader FromFile(string fileName)  
{  
 var fs = new FileStream(fileName, FileMode.Open);  
 var parser = LogEntryParser.Create(fileName);  
  
 return new LogEntryReader(fs, parser);  
}

Листинг 5.5 - Фабричный метод LogEntryReader.FromFile

Данный фабричный метод упрощает импорт конкретных лог-файлов, поскольку прячет процесс открытия файла и создание конкретного парсера (\*). Наличие конструктора делает решение гибким и тестируемым, а фабричный метод упрощает использование класса большинством клиентов.

(\*) Сноска: фабричный метод LogEntryParser.Create был приведен в главе "Принцип открыт/закрыт". Парсеру нужно имя файла, чтобы определить, какой конкретно экземпляр парсера нужно создавать. В зависимости от имени файла будет использован парсер логов разных прилоежний.

Нужно ли выделять интерфейс ILogEntryReader? Ответить на этот вопрос сложно. Сам класс LogEntryReader не является автономным и будет использоваться для реализации класса LogImporter (листинг 5.6).

public class LogImporter : IDisposable  
{  
 private readonly LogEntryReader \_logEntryReader;  
  
 public LogImporter(string logFile)  
 {  
 \_logEntryReader = LogEntryReader.FromFile(logFile);  
 }  
  
 public void Dispose()  
 {  
 \_logEntryReader.Dispose();  
 }  
  
 public void ImportLogs()  
 {  
 foreach (var logEntry in \_logEntryReader.Read())  
 {  
 LogSaver.SaveEntry(logEntry);  
 }  
 }  
}

Листинг 5.6 - Реализация класса LogImporter

Класс LogImporter является довольно простым посредником, который использует классы LogEntryReader и LogSaver, и практически не содержит бизнес логики. На данном этапе может быть достаточно добавить несколько интеграционных тестов, которые проверят корректность поведения класса в реальном окружении. Если же логика класса LogImporter начнет усложняться, то выделение интерфейсов ILogEntryReader и ILogEntrySaver станет оправданным.

public class LogImporter  
{  
 private readonly ILogEntryReader \_reader;  
 private readonly ILogEntrySaver \_saver;  
  
 public LogImporter(ILogEntryReader reader, ILogEntrySaver saver)  
 {  
 \_reader = reader;  
 \_saver = saver;  
 }  
  
 public void ImportLogs()  
 {  
 foreach (var logEntry in \_reader.Read())  
 {  
 \_saver.SaveEntry(logEntry);  
 }  
 }  
 // Много другой логики!  
}

Листинг 5.7 - Выделение интерфейсов ILogEntryReader и ILogEntrySaver

**ПРИМЕЧАНИЕ**  
Обратите внимание, что я сделал акцент на выделении интерфейсов ILogEntryReader и ILogEntrySaver лишь в случае необходимости. Если класс LogImporter будет содержать лишь код, приведенный в листинге 5.7, то никакой необходимости в интерфейсах нет! Разделения обязанностей между классами LogImporter, LogSaver и LogEntryReader, уже достаточно для успешной эволюции приложения.

## Примеры нарушения принципа инверсии зависимостей

* Низкоуровневые классы напрямую общаются с высокоуровневыми классами: модели знают о пользовательском интерфейсе или инфраструктурный код знает о бизнес-логике.
* Классы принимают слишком низкоуровневые интерфейсы: классы принимают интерфейсы, такие как IFileStream, что может привести к подрыву инкапсуляции и излишнему увеличению сложности.

## Выводы

"Простое эвристическое правило - зависеть нужно от абстракций", на деле оказалось более сложным, чем могло показаться изначально.

Принцип инверсии зависимостей не сводится лишь к выделению интерфейсов и передаче их через конструктор. DIP объясняет для чего нужно это делать. Классы имеют право контролировать свои детали реализации, но некоторые аспекты находятся за пределами их компетенции. Чтобы не завязываться на классы верхнего уровня, класс может объявить некоторый интерфейс, и потребовать его экезмпляр через аргументы конструктора. Таким образом мы можем "инвертировать" зависимости и позволить классам нижних уровней взаимодействовать с другими частями системы, ничего конкретного о них не зная.