# Паттерн Адаптер (Adapter)

**Назначение:** Преобразует интерфейс одного класса в интерфейс другого, который ожидают клиенты. Адаптер обеспечивает совместную работу классов с несовместимыми интерфейсами, которая без него была бы невозможна.

## Мотивация

Далеко не все системы обладают прекрасным дизайном. Даже если модуль или приложение были хорошо продуманы изначально, внесение изменений разными разработчиками в течении длительного времени, может привести к неприятным последствиям. Одно из таких последствий - это рассогласованность реализации однотипных задач.

Давайте снова вернемся к приложению по импорту лог-файлов. На ранних этапах разработки может быть не очевидным, появится ли необходимость в обобщенном решении, базовых классах и гибкости. Если изначально было принято решение хранить данные в Sql Server, то вместо использования иерархии наследования, разработчики создали простой класс SqlServerLogSaver. Со временем требования или понимание задачи могли измениться, в результате чего команда решила добавить еще одно хранилище данных, например, Elasticsearch. Новый класс, ElasticsearchLogSaver проектировался другим разработчиком, в результате он стал обладать интерфейсом, отличным от интерфейса своего предшественника (рис. 1.1):

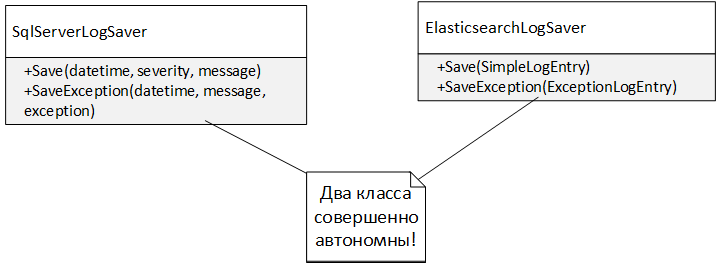


Рисунок 1.1 - Диаграмма классов SqlServerLogSaver и ElasticsearchLogSaver

Классы сохранения данных являются независимыми (что хорошо), но это не позволяет использовать их полиморфным образом (что плохо). Решение заключается создании еще одного абстрактного слоя путем выделения интерфейса ILogSaver. Но вместо того, чтобы изменять существующие классы и реализовывать в них новый интерфейс, можно создать промежуточный слой адаптеров, которые будут связывать старый и новый миры воедино (рис. 1.2):

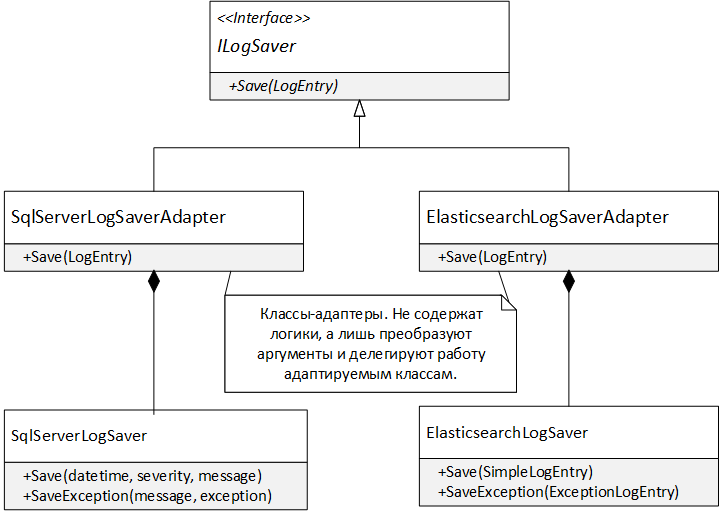
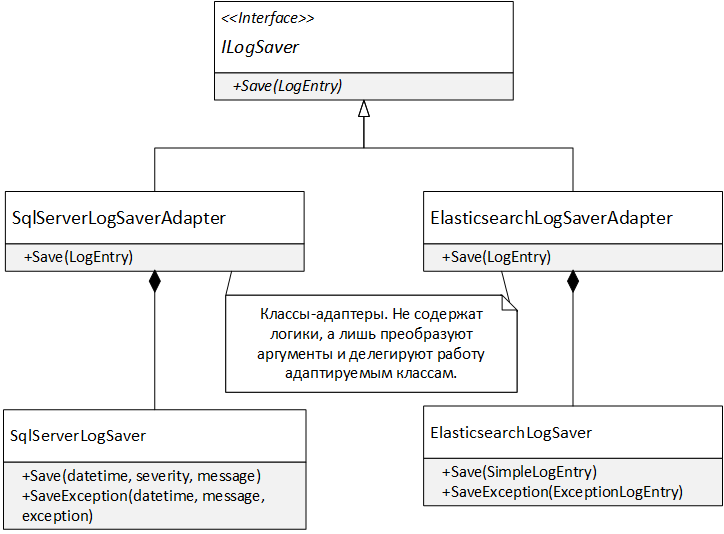


Рисунок 1.2 - Использование адаптеров для сохранения лог-файлов

Реализация каждого адаптера будет достаточно тривиальной. Нужно преобразовать входные параметры, вызвать метод адаптируемого класса и, возможно, преобразовать полученный результат (листинг 1.1).

public class SqlServerLogSaverAdapter : ILogSaver  
{  
 private readonly SqlServerLogSaver \_sqlServerLogSaver =   
 new SqlServerLogSaver();  
  
 public void SaveLogEntry(LogEntry logEntry)  
 {  
 var simpleEntry = logEntry as SimpleLogEntry;  
 if (simpleEntry != null)  
 {  
 \_sqlServerLogSaver.Save(  
 simpleEntry.EntryDateTime, simpleEntry.Severity.ToString(),  
 simpleEntry.Message);  
 return;  
 }  
  
 var exceptionEntry = (ExceptionLogEntry) logEntry;  
 \_sqlServerLogSaver.SaveException(  
 exceptionEntry.EntryDateTime, exceptionEntry.Message, exceptionEntry.Exception);  
 }  
}

Листинг 1.1 - Реализация SqlServerLogSaverAdapter

**Адаптер - это клей, который связывает воедино два мира путем "подгонки" текущих классов к требуемому интерфейсу**.

## Классическая диаграмма классов паттерна Адаптер

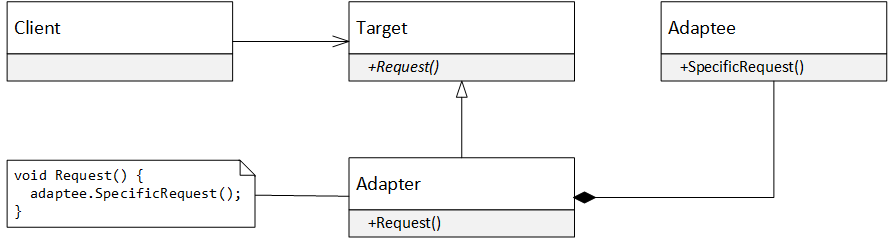


Рисунок 1.3 - Классическая диаграмма классов паттерна Адаптер

**Участники**

* Target (ILogSaver) - целевой интерфейс, к которому нужно преобразовать интерфейс существующих классов.
* Adaptee (SqlServerLogSaver) - существующий класс, чей интерфейс нужно преобразовать.
* Adapter (SqlServerLogSaverAdapter) - класс-адаптер, который преобразует интерфейс адаптируемого класса к целевому.
* Client (клиенты ILogSaver) - клиенты нового интерфейса, которые работают с адаптированными классами полиморфным образом.

## Обсуждение паттерна Адатпер

Адаптер является одним из тех паттернов проектирования, которые мы используем практически не задумываясь. Диаграмма классов этого паттерна настолько общая, что практически любую композицию объектов можно считать примером использования адаптеров.

**ПРИМЕЧАНИЕ: Адаптеры классов и объектов**  
"Бандой четырех" были описаны два вида адаптеров - адаптеры классов и адаптеры объектов. Ранее была приведена диаграмма классов адаптера объектов. Адаптер класса использует множественное наследование для "склеивания" адаптируемого класса к новому интерфейсу, что ограничивает его применимость языками с множественным наследованием реализации.

### "Адаптивный рефакторинг"

Зачастую адаптеры применяются для адаптации классов к существующим интерфейсам. Но, в некоторых случаях, адаптеры могут упростить эволюцию дизайна и упростить рефакторинг.

Давайте снова вернемся к задаче импорта логов, но подойдем к проблеме сохранения логов с другой стороны. Предположим, что у нас есть иерархия классов LogSaver, но мы хотим модифицировать ее таким образом, чтобы операция сохранения стала асинхронной.

Можно, конечно, взять и переписать все классы этой иерархии, но можно пойти более итеративным путем.

1. Проектируем новый интерфейс IAsyncLogSaver с набором нужных асинхронных методов сохранения.
2. Создаем первую реализацию AsyncLogSaverAdapter, которая реализует IAsyncLogSaver, но делегирует все операции старой реализации.
3. Переводим клиентов иерархии LogSaver на "асинхронные рельсы" и проверяем, что клиенты работают нормально.
4. Создаем полноценную асинхронную реализацию сохранения логов.
5. Удаляем AsyncLogSaverAdapter за ненадобностью.

Диаграмма классов этого решения будет выглядеть так (рис. 1.4):

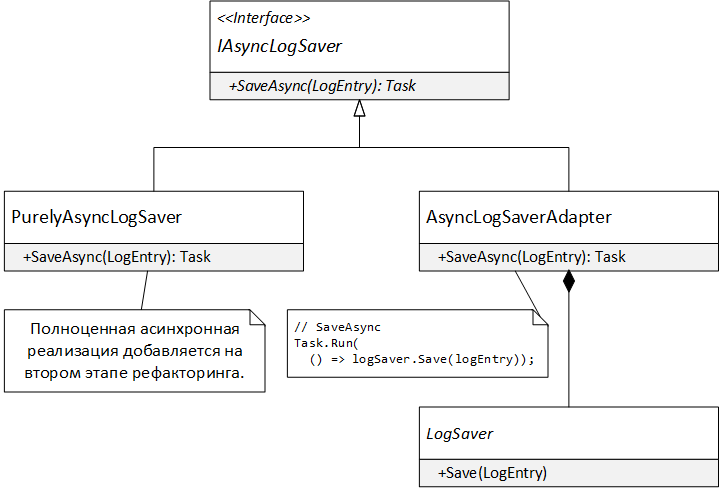


Рисунок 1.4 - Диаграмма классов адаптивного рефакторинга

Преимущество такого подхода в том, что изменения происходят постепенно, а не одним большим скачком. Реализация AsyncLogSaverAdapter будет примитивной, но это позволит уменьшить количество ошибок в единицу времени (листинг 1.2):

public interface IAsyncLogSaver  
{  
 Task SaveAsync(LogEntry logEntry);  
}  
  
public class AsyncLogSaverAdapter : IAsyncLogSaver  
{  
 private readonly LogSaver \_logSaver;  
  
 public AsyncLogSaverAdapter(LogSaver logSaver)  
 {  
 \_logSaver = logSaver;  
 }  
  
 public Task SaveAsync(LogEntry logEntry)  
 {  
 return Task.Run(() => \_logSaver.Save(logEntry));  
 }  
}

Листинг 1.2 - Реализация класса AsyncLogSaverAdapter

### Языковые адаптеры

**TODO: нужна обратная связь от рецензентов. Насколько понятен раздел?!?!?**

Компилятор языка C# налагает определенные ограничения на использование пользовательских типов в некоторых языковых конструкциях. Например, для использования типа в LINQ выражениях вида from s in obj требуется экземплярные методы Select, Where, OrderBy и другие, но также подойдут и методы расширения. Для использования в цикле foreach типу необходимо предоставить метод GetEnumerator, который возвращает тип с методом bool MoveNext и свойством Current. Для использования async/await - требуется метод GetAwaiter и GetResult (\*). Для использования инициализатора коллекций требуется реализация интерфейса IEnumerable<T> (или IEnumerable) и метода Add.

(\*) Сноска: все это примеры "утиной типизации" (duck typing) на уровне языка программирования.

Вместо того, чтобы модифицировать код приложения для удовлетворения этим требованиям, можно создать набор классов-адаптеров или методов расширения. Начиная с 6-й версии языка C# для использования инициализатора коллекций достаточно иметь метод расширения Add (листинг 1.3):

public class CustomCollection : IEnumerable<int>  
{  
 private readonly List<int> \_list = new List<int>();  
  
 public void Insert(int value)  
 {  
 \_list.Add(value);  
 }  
  
 public IEnumerator<int> GetEnumerator()  
 {  
 return \_list.GetEnumerator();  
 }  
  
 IEnumerator IEnumerable.GetEnumerator()  
 {  
 return GetEnumerator();  
 }  
}  
  
public static class CustomCollectionExtensions  
{  
 public static void Add(this CustomCollection customCollection, int value)  
 {  
 customCollection.Insert(value);  
 }  
}

Листинг 1.3 - Класс CustomCollection и его методы расширения

В этом случае класс CustomCollectionExtensions выступает своеобразным адаптером, который позволяет использовать класс CustomCollection с инициализатором коллекций (листинг 1.4):

// При наличии импорта нужного пространства имен,  
// код успешно компилируется и запускается!  
var cc = new CustomCollection {42};

Листинг 1.4 - Пример использования инициализатора коллекций с CustomCollection

## Применимость

Адаптер позволяет использовать существующие типы в новом контексте.

1. Повторное использование чужого кода. В некоторых случаях у нас уже есть код, который решает нужную задачу, но его интерфейс не подходит для текущего приложения. Вместо изменения кода библиотеки можно создать слой адаптеров.
2. "Адаптивный рефакторинг". Адаптеры позволяют плавно изменять существующую функциональность, путем выделения нового "правильного" интерфейса, но с использованием старой проверенной функциональности.

## Примеры в .NET Framework

* TextReader/TextWriter адаптеры над классами Stream для чтения/записи текстовых данных в потоки ввода/вывода.
* BinaryReader/BinaryWriter аналогичные адаптеры для работы с бинарными данными потоков ввода/вывода.
* ReadonlyCollection<T> адаптирует произвольный список (IList<T>) к коллекции только для чтения (IReadonlyCollection<T>).
* Любая реализация LINQ-провайдера (интерфейса IQueryProvider) является адаптером.