# Паттерн Декоратор (Decorator)

**Назначение:** Динамически добавляет объекту новые обязанности. Является гибкой альтернативой порождению подклассов с целью расширения функциональности.

## Мотивация

Хорошо спроектированный класс отвечает за определенную функциональность, не распыляясь на решение второстепенных задач. Но что делать, если второстепенные задачи, такие как логирование, кеширование, замеры времени исполнения проникают в код класса и увеличивают сложность реализации непомерным образом? Можно выделить эти аспекты поведения во вспомогательные классы, но все равно останется вопрос по их координации. Паттерн Декоратор элегантным образом решает задачу "нанизывания" обязанностей одних на другие.

Давайте рассмотрим задачу импорта логов несколько с иной стороны. Помимо консольной утилиты или локального Windows-сервиса, мы можем разработать облачное приложение, которое будет принимать трассировочные сообщения от приложений пользователя и предоставлять интерфейс для последующего поиска (\*). Большинство облачных приложений не позволяют передавать произвольное количество данных от конкретного пользователя. При достижении определенного лимита, например, 10 сообщений в секунду от конкретного пользователя, включается режим ограничений входящих запросов (throttling).

(\*) Сноска: именно этим занимается Application Insights - набор облачных сервисов от компании Майкрософт, которые принимают, хранят и анализируют телеметрические данные приложений. Application Insights представляет API для сохранения телеметрических данных и веб-интерфейс для анализа сохраненных данных, включая полнотекстовый поиск в логах, исключениях и других данных.

В простом случае логику тротлинга можно смешать с логикой сохранения, но такое решение сложно поддерживать в длительной перспективе. Вместо этого, можно воспользоваться декоратором (листинг 3.1):

public interface ILogSaver  
{  
 Task SaveLogEntry(string applicationId, LogEntry logEntry);  
}  
  
public sealed class ElasticsearchLogSaver : ILogSaver  
{  
 public Task SaveLogEntry(string applicationId, LogEntry logEntry)  
 {  
 // Сохраняем переданную запись в Elasticsearch  
 return Task.FromResult<object>(null);  
 }  
}  
  
public abstract class LogSaverDecorator : ILogSaver  
{  
 protected readonly ILogSaver \_decoratee;  
  
 protected LogSaverDecorator(ILogSaver decoratee)  
 {  
 \_decoratee = decoratee;  
 }  
  
 public abstract Task SaveLogEntry(  
 string applicationId, LogEntry logEntry);  
}

Листинг 3.1 - Исходный класс декоратора сохранения данных

Идея паттерна декоратор в том, что у интерфейса (ILogSaver) появляется два вида реализаций. Основная реализация бизнес-функциональности (ElasticsearchLogSaver) и набор классов-декораторов, которые реализуют тот же интерфейс, но довольно специфическим образом. Декоратор принимает в конструкторе тот же самый интерфейс, а в реализации делегирует работу декорируемому объекту с подмешиванием некоторого поведения до или после вызова метода (листинг 3.2):

public class ThrottlingLogSaverDecorator : LogSaverDecorator  
{  
 public ThrottlingLogSaverDecorator(ILogSaver decoratee)   
 : base(decoratee)  
 {}  
  
 public override async Task SaveLogEntry(  
 string applicationId, LogEntry logEntry)  
 {  
 if (!QuotaReached(applicationId))  
 {  
 IncrementUsedQuota();  
  
 // Сохраняем записи. Обращаемся к декорируемому объекту!  
 await \_decoratee.SaveLogEntry(applicationId, logEntry);  
 return;  
 }  
  
 // Сохранение невозможно! Лимит приложения исчерпан!  
 throw new QuotaReachedException();  
 }  
  
 private bool QuotaReached(string applicationId)  
 {  
 return true;  
 }  
  
 private void IncrementUsedQuota()  
 {}  
}

Листинг 3.2 - Реализация декоратора ThrottlingLogSaverDecorator

Декоратор с ограничением числа вызовов (ThrottlingLogSaverDecorator) вначале проверяет, не исчерпано ли число запросов со стороны текущего приложения, и лишь затем делегирует работу основному объекту (полю decoratee). Если квота не достигла лимита, то вызывается основной метод и данные успешно сохраняются. В противном случае генерируется исключение QuotaReachedException, которое скажет клиенту попробовать выполнить тот же самый запрос через время (\*).

(\*) Сноска: если функциональность по сохранению записей лог-файлов находится в сервисе, доступном по протоколу HTTP, то можно даже воспользоваться специальным возвращаемым статусом 429 - Too Many Requests, чтобы сообщить клиенту о чрезмерной нагрузке с его стороны.

Теперь останется правильным образом сконструировать экземпляр ILogSaver и "навесить" на него нужный набор декораторов. При этом клиенты интерфейса будут работать с ним, как и раньше, не замечая наличия дополнительного поведения:

ILogSaver logSaver = new ThrottlingLogSaverDecorator(new ElasticsearchLogSaver());  
  
var controller = new LogSaverController(logSaver);

Листинг 3.3 - Пример инициализации декоратора

## Классическая диаграмма классов паттерна Адаптер

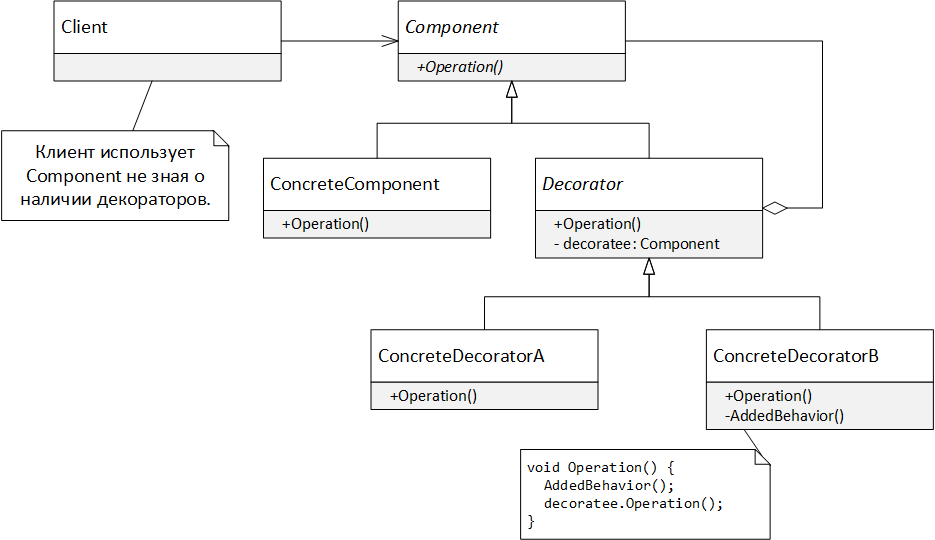


Рисунок 3.1 - Классическая диаграмма классов паттерна Декоратор

**Участники**

* Component (ILogSaver) - базовый класс компонента, чье поведение будет расширяться декораторами.
* ConcreteComponent (ElasticsearchLogSaver) - конкретная реализация компонента.
* Decorator (LogSaverDecorator) - базовый класс декоратора, предназначенный для расширения поведения компонента.
* ConcreteDecoratorA (ThrottlingLogSaverDecorator) - конкретный декоратор, который добавляет декорируемому объекту специфическое поведение.

## Обсуждение паттерна Декоратор

### Композиция vs. Наследование

Задачу с управлением количества сохраненных лог-файлов можно было бы решить и с помощью наследования. Для этого достаточно было создать наследника ElasticsearchLogSaver и добавить в него логику по ограничению числа запросов (рис. 3.2).

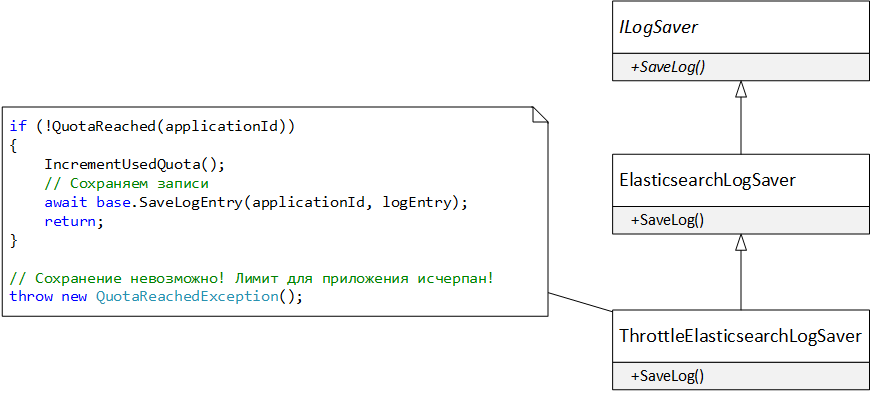


Рисунок 3.2 - Тротлинг на основе наследования

Разница между этими подходами в том, что наследование обеспечивает более жесткую связь по сравнению с агрегацией. Класс ThrottleElasticsearchLogSaver жестко привязан к своему базовому классу и не может быть использован с другой реализацией интерфейса ILogSaver.

Декораторы же могут использоваться с любой реализацией интерфейса ILogSaver, включая другие декораторы. Агрегация позволяет добавлять поведение объектам, во время исполнения, наследование же расширяет поведение лишь во время компиляции. Гибкость агрегации позволяет "нанизывать" несколько декораторов друг на друга, создавая довольно сложное поведение.

Например, помимо тротлинга, легко добавить еще несколько реализаций декоторов, например, для кэширования или замера времени исполнения (листинг 3.4):

// Декоратор для замера времени исполнения метода  
public class TraceLogSaverDecorator : LogSaverDecorator  
{  
 public TraceLogSaverDecorator(ILogSaver decoratee) : base(decoratee)  
 {}  
  
 public override Task SaveLogEntry(  
 string applicationId, LogEntry logEntry)  
 {  
 var sw = Stopwatch.StartNew();  
 try  
 {  
 return \_decoratee.SaveLogEntry(applicationId, logEntry);  
 }  
 finally  
 {  
 Trace.TraceInformation("Операция сохранения зввершена за {0}мс",   
 sw.ElapsedMilliseconds);  
 }  
 }  
}  
  
ILogSaver logSaver =   
 new ThrottlingLogSaverDecorator(  
 new TraceLogSaverDecorator(  
 new ElasticsearchLogSaver()));  
  
// Используем logSaver с двумя декораторами

Листинг 3.4 - Использование нескольких декораторов

### Инициализация декораторов

Наличие декораторов делает процесс инициализации компонентов более сложным. В простых случаях код инициализации может находится в корне приложения, например, в методе Main. В более сложных случаях декоратор может инициализироваться в IoC-контейнере или же создаваться фабрикой.

Если есть ряд предустановленных конфигураций, то достаточно выделить фабричный метод, отвечающий за создание объекта:

public static class LogSaverFactory  
{  
 public static ILogSaver CreateLoader()  
 {  
 return   
 new ThrottlingLogSaverDecorator(  
 new TraceLogSaverDecorator(  
 new ElasticsearchLogSaver()));  
 }  
}

Листинг 3.5 - Фабричный метод создания объекта ILogSaver

Статический фабричный метод не позволит изменить конфигурацию декоратора на лету, но обычно такой гибкости вполне достаточно. В случае изменения требований, достаточно будет заменить реализацию фабрики и развернуть приложение заново.

### Недостатки декораторов

Декоратор, как и большинство других паттернов, обладает своими недостатками.

* **Чувствительность к порядку**. Код инициализации декораторов очень важен, поскольку именно в процессе создания определяется вложенность и порядок исполнения разных декораторов.
* **Сложность отладки**. Разработчику, не знакомому с этим паттерном замер времени исполнения или кэширование результатов декораторами может показаться черной магией. Отлаживать проблемы, которые возникают декоратором внутри декоратора может быть непростой задачей.
* **Увеличение сложности**. Декоратор является достаточно тяжеловесным паттерном, к которому стоит прибегать тогда, когда выделяемый аспект поведения достаточно сложен. Если нужно закешировать результаты в одном из десяти методов, то сложность, привнесенная декоратором будет не оправдана.

### Генерация декораторов

Многие IoC-контейнеры, такие как Unity или StructureMap, поддерживают генерацию декораторов на лету с помощью перехватчиков (Interceptors). Идея заключается в генерации IL-кода (Intermediate Langauge) во время исполнения, который будет выполнять пользовательский код до или после вызова декорируемого метода. Я предпочитаю использовать более простые решения и переходить к средствам, вроде генерации кода, лишь в случае необходимости. Если вас интересует эта тема, то достаточно поискать в вашем любимом поисковом сервисе (кто сказал Гугл?) материалы по теме "Unity interceptors" и вы найдете массу примеров.

## Применимость

Декоратор позволяет динамически расширять поведение объектов. Декораторы идеально подходят для добавления аспектов поведения, применимых для всех методов интерфейса. Если кэшировать нужно лишь результаты одного метода класса, то использование декоратора будет слишком тяжеловесным.

Декораторы применяются для добавления всем методом интерфейса определенной функциональности. Декораторы отлично подходят для решения следующих задач:

* Кэширование результатов работы
* Замер времени исполнения методов
* Логирование аргументов
* Управление доступом пользователей
* Модификация аргументов или результата работы методов: упаковка/распаковка, шифрование и т.п.

Динамическая природа позволяет "нанизывать" аспекты один на другой, обходя ограничения наследования, использование которого привело бы к комбинаторному взрыву числа наследников.

**ПРИМЕЧАНИЕ**  
За последние несколько лет я неоднократно применял декораторы на практике. Пример, рассмотренный в разделе "Мотивация" основан на практическом опыте использования декоратора в одном из сервисов Application Insights, и решает задачу чрезмерного числа запросов со стороны одного пользователя.

До этого, я неоднократно использовал декораторы для замера длительности вызова методов, а также для кэширования. Пример использования декораторов для кэширования объектов Task я описывал в статье "Кэширующий декоратор на деревьях выражений" - http://sergeyteplyakov.blogspot.com/2012/09/blog-post\_24.html.

## Примеры в .NET Framework

В .NET Framework существует довольно большое количество декораторов. Большая их часть предназначена для работы с потоками ввода вывода, но есть и исключения:

* System.IO.BufferedStream - добавляет буферизацию потоку ввода вывода.
* System.IO.Compression.GZipStream, System.IO.Compression.DeflateStream - добавляют возможности сжатия потоку ввода вывода
* System.CodeDom.Compiler.IndentedTextWriter управляет форматированием объекта System.IO.TextWriter.
* System.Reflection.TypeDelegator - декоратор для добавление дополнительных аспектов поведения объекту System.Type.
* System.Collections.SortedList.SyncSortedList - декоратор, который является вложенным классом SortedList, и вызывает все методы декорируемого списка внутри конструкции lock.