# Паттерн Прокси (Proxy)

**Назначение:** является суррогатом другого объекта и контролирует доступ к нему.

## Мотивация

Изменение требований и эволюция системы, может потребовать внесения серьезных архитектурных изменений. Если на ранних этапах некоторая операция выполнялась на стороне клиента, или же приложение состояло из одного процесса, то со временем, исполнение операции может быть перенесено на сервер, а приложение разбито на несколько процессов.

В результате, возникает задача взаимодействия с удаленным процессом, реализация которой должна быть максимально похожей на локальное взаимодействие. Именно для таких целей предназначен паттерн Прокси.

В случае с приложением по импорту лог-файлов, его разработчики могут прийти к заключению, что прямая работа с удаленных хранилищем является неудачным решением, и выделить отдельный сервис для сохранения данных. При использовании современных библиотек, таких как WCF, ASP.NET Web API или .NET Remoting сделать такой переход будет не сложно.

При использовании WCF, старый интерфейс ILogSaver и все объекты, используемые в качестве аргументов и возвращаемых значений, нужно декорировать специальными атрибутами (ServiceContract + OperationContract и DataContract + DataMember). Затем, нужно сконфигурировать приложение, добавив в его конфигурационный файл информацию о привязках (binding), контрактах (contracts) и точках доступа (endpoint). После чего останется создать простой прокси-класс и пользоваться им так, как будто реализация ILogSaver находится в этом же процессе (\*):

(\*) Сноска: как будет показано ниже, полностью игнорировать распределенную природу не следует, поскольку это может привести к негативным последствиям. Распределенное взаимодействие налагает слишком большое число дополнительных ограничений, игнорировать которые опасно.

class LogSaverClient : ClientBase<ILogSaver>, ILogSaver  
{  
 public void SaveLogEntry(LogEntry logEntry)  
 {  
 // Делегируем всю работу внутреннему свойству Channel,  
 // с типом ILogSaver  
 Channel.SaveLogEntry(logEntry);  
 }  
}  
  
//...  
public void SaveLogEntryAtTheBackend(LogEntry logEntry)  
{  
 var proxy = new LogSaverClient();  
 proxy.SaveLogEntry(logEntry);  
 proxy.Close();  
}

Листинг 5.1 - Пример использования прокси-класса для работы с WCF

Основная "магия" находится в классе ClientBase<T> и нижележащей инфраструктуре WCF, которая отвечает за сериализацию аргументов, передачу их "по проводам" и десериализацию ответа. Класс LogSaverClient является прокси-классом, который позволяет работать с удаленным WCF сервисом так, как будто экземпляр ILogSaver находится в текущем процессе.

## Классическая диаграмма классов паттерна Прокси

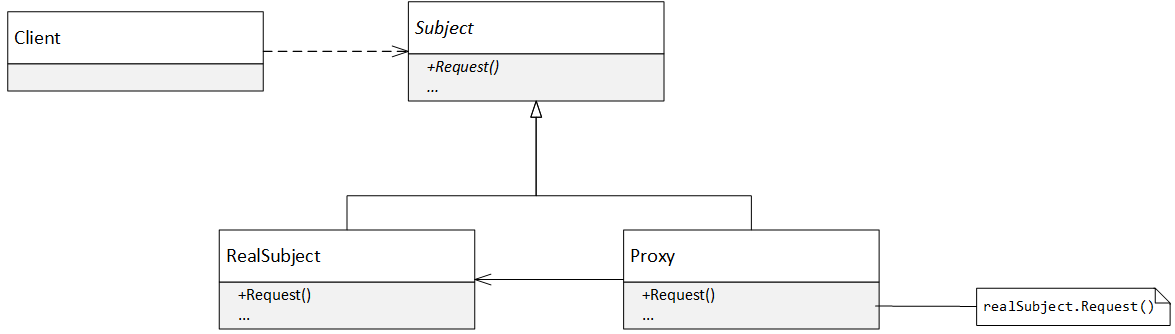


Рисунок 5.1 - Классическая диаграмма классов паттерна Прокси

**Участники**

* Client работает с абстрактным компонентом, не зная, является ли он настоящим или нет.
* Subject (ILogSaver) определяет интерфейс компонента.
* Proxy (LogSaverClient) прокси-объект, который реализует интерфейс компонента, но делегирует всю работу настоящему объекту.
* RealSubject - реальный компонент, доступ к которому осуществляется через прокси.

## Обсуждение паттерна Прокси

Прокси является одним из немногих паттернов проектирования, который с течением времени претерпел довольно серьезные изменения. В классическом труде "банды четырех" описаны три основных сценария использования паттерна Прокси:

* **Удаленный прокси** отвечает за кодирование запроса и его аргументов для работы с компонентом в другом адресном пространстве.
* **Виртуальный прокси** может кэшировать дополнительную информацию о реальном компоненте, чтобы отложить его создание.
* **Защищающий прокси** проверяет, имеет ли вызывающий объект необходимые для выполнения запроса права.

Второй и третий вариант паттерна Прокси применяется на практике, но своей известностью этот паттерн обязан первому варианту. Прокси-классы являются стандартным паттерном в подавляющем большинстве современных технологий построения распределенных приложений. Однако внешний вид прокси-классов и подходы к их реализации за последние 15 лет претерпели очень серьезные изменения.

## Прозрачный удаленный Прокси

Основная идея удаленного прокси заключается в том, что клиент может работать с прокси-классом так, как будто он работает с объектом в своем собственном адресном пространстве. Этот подход настолько понравился разработчикам распределенных технологий конца 90-х начала 2000-х, что многие технологии того времени постарались стереть грань между внутрипроцессным и межпроцессным взаимодействием. Такой подход получил название *прозрачность местоположения* (location transparency) и лег в основу таких технологий, как DCOM и .NET Remoting.

Однако, со временем, разработчики начали осознавать, что постоянно сталкиваются с проблемами "дырявых абстракций" и не могут игнорировать особенности сетевого взаимодействия. Особенности обработки ошибок, необходимость, чтобы все параметры методов были сериализируемыми, задержки и длительное время исполнения методов, все это требует другого подхода с точки зрения разработки (\*). Поэтому в последних технологиях, таких как WCF и ASP.NET Web API, прокси-классы перестали быть "прозрачными". Разработчик может работать с интерфейсом ISomeInterface, каждый вызов которого исполняется на удаленной стороне, но ему самому придется стереть грань между распределенным и локальным мирами.

(\*) Сноска: можно смело говорить, что удаленный прокси нарушает принцип замещения Лисков. Поведение клиента обязательно изменится при замене в приложении локальной реализаций интерфейса ILogSaver на удаленный прокси.

Такие классы, как ChannelBase<T> в WCF, очень сильно упрощают взаимодействие с удаленным процессом, но они скорее выполняют роль фасадов, а не прокси-классов!

### Прокси vs. Декоратор

Структура паттернов Прокси и Декоратор, очень похожи. Каждый из них содержит ссылку на базовый компонент и делегирует ему выполнение всей работы. Однако у этих паттернов разное назначение.

Декоратор добавляет поведение всем методам интерфейса, позволяя "нанизывать" расширения одно на другое. Прокси-класс может выполнять определенные действия, например, создавать настоящий компонент по мере требования, но он не должен "подмешивать" ничего в результаты исполнения операции.

### Виртуальный прокси и Lazy<T>

Класс Lazy<T> можно считать универсальным строительным блоком, с помощью которого легко создавать виртуальные прокси-классы.

public interface IHeavyweight  
{  
 void Foo();  
}  
  
// Стоимость создания класса очень высока  
public class Heavyweight : IHeavyweight  
{  
 public void Foo()  
 {}  
}  
  
// Виртуальный прокси, который будет создавать  
// тяжеловесный объект лишь в случае необходимости  
public class HeavyweightProxy : IHeavyweight  
{  
 private readonly Lazy<Heavyweight> \_lazy = new Lazy<Heavyweight>();  
  
 public void Foo()  
 {  
 \_lazy.Value.Foo();  
 }  
}

Листинг 5.2 - Реализация виртуального прокси с помощью Lazy<T>

## Применимость

Прокси-классы активно применяются там, где нужно спрятать исходный объект и подмешать к нему некоторое поведение: спрятать его распределенную природу, дать возможность отложенного создания или контролировать количество вызовов метода.

## Примеры в .NET Framework

* **Удаленные прокси-классы**. В коммуникационных технологиях в .NET Framework применяются удаленные прокси-классы: ChannelBase<T> в WCF, RealProxy в .NET Remoting.
* **Виртуальные прокси-классы в ORM-фреймворках**. В современных ORM (Object-Relational Mapping, объектно-реляционное отображение), таких как NHibernate или Entity Framework, применяются специализированные виртуальные прокси-классы. ORM генерирует оболочку над сущностями (data entities), чтобы отслеживать изменения и генерировать корректный SQL-код для обновления записей в базе данных.