# Паттерн Абстрактная Фабрика (Abstract Factory)

Фабрика – это второй по популярности паттерн после паттерна Синглтона. Существует две классические разновидности фабрик – Абстрактная Фабрика и Фабричный Метод – предназначенные для инкапсуляции создания объекта или семейства объектов. На практике очень часто отходят от классических реализаций этих паттернов и называют «Фабрикой» любой класс, инкапсулирующий в себе создание объектов.

В данной главе будет рассмотрены паттерн Абстрактная Фабрика и его особенности, а в следующей - Фабричный Метод.

**Назначение**: Абстрактная Фабрика предоставляет интерфейс для создания семейства взаимосвязанных или родственных объектов (dependent or related objects), не специфицируя их конкретных классов.

**Другими словами**: Абстрактная Фабрика представляет собой стратегию создания семейства взаимосвязанных или родственных объектов.

## Мотивация

Выразительность наследования обеспечивается за счет полиморфизма. Использование интерфейсов или базовых классов позволяет "абстрагироваться" от конкретной реализации, что делает решение простым и расширяемым. Однако где-то в приложении должна быть точка, в которой создаются объекты и известен их конкретный тип.

В некоторых случаях решение о конкретных типах можно откладывать до последнего, вплодь до корня приложения (Application Root). В этом случае, конструирование конкретных типов происходит в методе Main (или аналогичном методе, в зависимости от типа приложения), и затем созданные объекты передаются для последующей обработки. Однако в некоторых случаях, создание объекта должно происходить раньше в коде приложения.

Давайте вернемся к задаче сохранения прочитанных лог-файлов в хранилище для последующего полнотекстового поиска. Многие реляционные базы данных, например, SQL Server поддерживают полнотекстовый поиск по текстовым полям. Чтобы не завязываться на конкретную СУБД (систему управления базами данных), реализация класса LogSaver может использовать класс DbProviderFactory библиотеки ADO.NET:

public class LogSaver  
{  
 private readonly DbProviderFactory \_factory;  
  
 public LogSaver(DbProviderFactory factory)  
 {  
 \_factory = factory;  
 }  
  
 public void Save(IEnumerable<LogEntry> logEntries)  
 {  
 using (var connection = \_factory.CreateConnection())  
 {  
 SetConnectionString(connection);  
 using (var command = \_factory.CreateCommand())  
 {  
 SetCommandArguments(logEntries);  
 command.ExecuteNonQuery();  
 }  
 }  
 }  
  
 private void SetConnectionString(DbConnection connection)  
 {}  
  
 private void SetCommandArguments(IEnumerable<LogEntry> logEntry)  
 {}  
}

Листинг 2.1 - Пример использования абстрактной фабрики

Теперь, вызывающий код может передать нужный экезмпляр DbProviderFactory, например, SqlClientFactory, для работы с SQL Server, а затем передать NpgsqlConnectionFactory, чтобы перейти на PostgreSql.

Класс DbProviderFactory в данном случае предназначен для создания семейства взаимосвязанных объектов, таких как подключение к базе данных (DbConnection), команд (DbCommand), адаптеров (DbDataAdapter) и др., и является примером паттерна Абстрактная фабрика.

## Диаграмма паттерна Абстрактная Фабрика

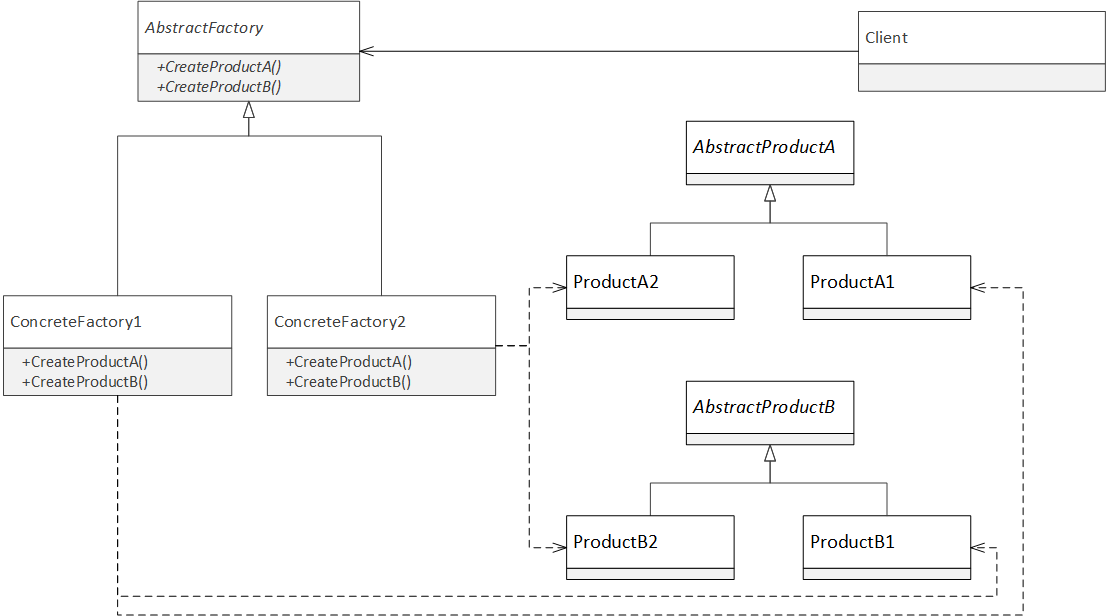


Рисунок 2.1 - Классическая диаграмма паттерна Абстрактная Фабрика

**Участники**

* AbstractFactory (DbProviderFactory) объявляет интерфейс (с возможной базовой реализацией) для создания семейства продуктов.
* AbstractProductA, AbstractProductB (DbCommand, DbConnection) семейство продуктов, которые будут использоваться клиентом для выполнения своих задач.
* ProductA1, ProductB1 (SqlConnection, NpgsqlConnection) - конкретные типы продуктов.
* Client (LogSaver) клиент фабрики, который получает конкретные продукты для реализации своего поведения.

## Обсуждение паттерна Абстрактная Фабрика

Основная особенность абстрактной фабрики заключается в том, что она предназначена для создания семейства объектов, что несколько сильно сужает ее применимость. Но в некоторых предметных областях или инфраструктурном коде периодически возникают задачи, которые решаются набором классов. Сериализаторы/десериализаторы, классы для сжатия/распаковки, шифрования/дешифрования и т.п. Приложение должно использовать согласованные типы объектов и абстрактная фабрика идеально подходит для решения этой задачи. Интерфейс абстрактной фабрики объявляет набор фабричных методов, а конкретная реализация обеспечивает создания этого семейства объектов.

Абстрактная фабрика представляет собой стратегию (\*) создания семейства объектов. Но при этом остается один вопрос: а кто создает конкретный экземпляр абстрактной фабрики? Еще одна фабрика?

(\*) Сноска: паттерна Стратегия был рассмотрен в первой части книги.

### Проблема курицы и яйца

**Использование конкретного типа**. В случае простых приложений конкретный экземпляр абстрактной фабрики можно создать в корне приложения. Например, можно создать конкретное приложение импорта файлов вашего прилоежния для сохранения их в SQL Server. В этом случае можно создать экземпляр SqlClientFactory и передать его классу LogSaver:

public static void Main(string[] args)  
{  
 var saver = new LogSaver(SqlClientFactory.Instance);  
 var loader = new LogFileLoader(args[0]);  
 saver.Save(loader.Load());  
}

Листинг 2.2 - Использование абстрактной фабрики в точке входа приложения

**Фабрика фабрик**. Вместо использования конкретного типа абстрактной фабрики, можно выделить отдельный класс, задачей которого будет получение экземпляра фабрики. В этом случае мы имеем дело с фабрикой фабрик и главная задача не попасть в бесконечную рекурсию. В случае с абстрактной фабрикой DbProviderFactory такой класс уже существует: [DbProviderFactories.GetFactory](http://msdn.microsoft.com/en-us/library/h508h681(v=vs.110).aspx): (\*)

public static void Main(string[] args)  
{  
 var saver = new LogSaver(GetDbProviderFactory());  
 var loader = new LogFileLoader(args[0]);  
 saver.Save(loader.Load());  
}  
  
private static DbProviderFactory GetDbProviderFactory()  
{  
 const string factoryName = "System.Data.SqlClient";  
 return DbProviderFactories.GetFactory(factoryName);  
}

Листинг 2.3 - Использование фабричного метода

Сноска(\*): Фабричный метод более подробно будет рассмотрен в следующей главе.

Теперь можно пойти еще дальше и читать имя фабрики из конфигурационного файла приложения, или обращаться к другому источнику для получения информации о требуемом типе фабрики.

### Обобщенная абстрактная фабрика

С помощью абстрактной фабрики легко добавить новый подвид семейства объектов. Например, в случае с ADO.NET добавить еще одну реализацию DbProviderFactory относительно несложно. Но добавить новый фабричный метод для создания нового продукта достаточно сложно, поскольку добавление нового абстрактного метода сломает все существующие реализации фабрики.

Это свойство абстрактной фабрики было известно ее автором, поэтому одной из разновидностей реализации паттерна Абстрактная Фабрика является обобщенная фабрика, которая позволяет создавать произвольные типы объектов:

class GenericAbstractFactory  
{  
 public object Make(string id) { ... }   
 public IProduct MakeProduct(string id) { ... }  
 public T MakeGeneric<T>(string id) where T : IProduct { ... }  
}

Листинг 2.4 - Пример обобщенной абстрактной фабрики

В данном случае абстрактная фабрика становится конфигурируемой и с ее помощью можно создавать объект любого или заданного типа (IProduct в данном случае). Данная реализация абстрактной фабрики нашла свое воплощение в виде DI-контейнеров (Dependency Injection Containers или IoC - Inversion of Control Containers), таких как Unity, StructureMap и других (\*). Контейнер позволяет создать экземпляр нужного типа и найти экземпляры всех его зависимостей.

Использование обобщенных фабрик и контейнеров обеспечивает высокую гибкость приложения и не требует наследование, поскольку набор создаваемых типов задается в процессе конфигурирования фабрики. Но это решение может привести к переусложненному и хрупкому дизайну, поскольку в приложении появляются неявные связи между точкой конфигурирования фабрики и точкой ее использования.

(\*) Сноска: подробное описание контейнеров выходит за рамки этой книги. Хорошее описание DI-контейнеров можно найти в книге Марка Сиимана "Dependency Injection in .NET". Более подробную информацию о проблемах непосредственного использования контейнеров можно найти в моей статье "DI Паттерны. Service Locator" - http://sergeyteplyakov.blogspot.com/2013/03/di-service-locator.html.

## Применимость паттерна Абстрактная Фабрика

Абстрактная фабрика представляет собой слой для полиморфного создания семейства объектов. Ее использование подразумевает обязательное наличие двух составляющих: 1) семейства объектов, и 2) возможность замены создаваемого семейства объектов во время исполнения.

Наличие наследования обеспечивает гибкость, но и приводит к проблеме курицы и яйца, рассмотренной ранее. Иногда необходимость полноценной абстрактной фабрики очевидна с самого начала, но обычно лучше начать с самого простого решения и добавить гибкость лишь в случае необходимости.

Я предпочитаю идти таким путем: 1. Использую конкретные классы напрямую, пока не появляется необходимость в полиморфном поведении. 2. Прячу процесс создания объекта или семейства объектов за конкретными фабричными методами. Выделяю конкретный класс фабрики, с набором неполиморфным фабричных методов. 3. Перехожу к абстрактной фабрике, лишь когда появляется необходимость подмены процесса создания объектов во время исполнения.

## Примеры в .NET Framework

1. Класс DbProviderFactory из ADO.NET с фабричными методами CreateCommand(): DbCommand, CreateConnection() : DbConnection и др.
2. Класс CodeDomProvider с фабричными методами CreateGenerator(): ICodeGenerator, CreateCompiler(): ICodeCompiler, CreateParser(): ICodeParser.
3. Класс SymetricAlgorithm с фабричными методами CreateEncryptor(): ICryptoTransform и CreateDecryptor(): ICryptoTransform.