# Паттерн Фабричный Метод (Factory Method)

Когда при обсуждении дизайна упоминается "Фабрика", то в подавляющем большинстве случаев имеется ввиду одна из разновидностей паттерна Фабричный Метод.

**Назначение**: *определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать. Фабричный метод позволяет классу делегировать инстанцирование подклассам.*

## Мотивация

Как мы уже увидели в предыдущей главе, иерархии классов обеспечивают гибкость за счет полиморфного использования, но привносят дополнительную сложность. Абстрактная Фабрика решает задачу полиморфного создания семейства объектов, но очень часто возникает более простая задача - создания одного экземпляра иерархии наследования.

В задаче импорта лог файлов класс LogFileReader может самостоятельно открывать файл, анализировать его содержимое, создавать нужный экземпляр иерархии LogEntry (ExceptionLogEntry для исключений и SimpleLogEntry - для обычных записей) и возвращать его вызывающему коду. Это может быть неплохим решением для первой итерации, но его нельзя назвать удачным с точки зрения долгосрочной перспективы.

Существует как минимум два изменения, которые сделают решение более расширяемым и тестируемым. Можно выделить класс LogReaderBase который будет оперировать потоками вода вывода (экземплярами Stream) вместо файлов, а также создать отдельный класс LogEntryParser ответственный за создание экземпляров LogEntry:

public static class LogEntryParser  
{  
 public static LogEntry Parse(string data)  
 {  
 // Анализирует содержание data и создает нужный  
 // экземлпяр: ExceptionLogEntry или SimpleLogEntry  
 }  
}  
  
public abstract class LogReaderBase  
{  
 public IEnumerable<LogEntry> Read()  
 {  
 using (var stream = OpenLogSource())  
 {  
 using (var reader = new StreamReader(stream))  
 {  
 string line = null;  
 while ((line = reader.ReadLine()) != null)  
 {  
 yield return LogEntryParser.Parse(line);  
 }  
 }  
 }  
 }  
  
 protected abstract Stream OpenLogSource();  
}

Листинг 3.1 - Мотивация использования фабричного метода

Теперь клиенты класса LogReaderBase, такие как LogImporter, могут работать с различными источниками лог файлов, не зависимо от их местоположения. Также данную реализацию легко протестировать в юнит-тестах, путем создания класса MemoryStreamLogReader, который будет возвращать записи, помещенные туда во время инициализации теста.

В этом коде паттерн Фабричный Метод используется дважды. Метод OpenLogSource является классическим Фабричным Методом, когда базовый класс определяет абстрактный метод для создания объекта, а наследник его реализует. Класс LogEntryParser является разновидностью Фабричного Метода, который создает нужный экземпляр иерархии в зависимости от переданных аргументов.

**Цель любой фабрики**: *отгородить клиентов от подробностей создания экземпляров класса или иерархии классов.*

## Диаграмма паттерна Фабричный Метод

На практике встречается три вида паттерна Фабричный Метод.

* Классическая реализация на основе Шаблонного Метода
* Статический Фабричный Метод
* Полиморфный Фабричный Метод

### Классическая реализация

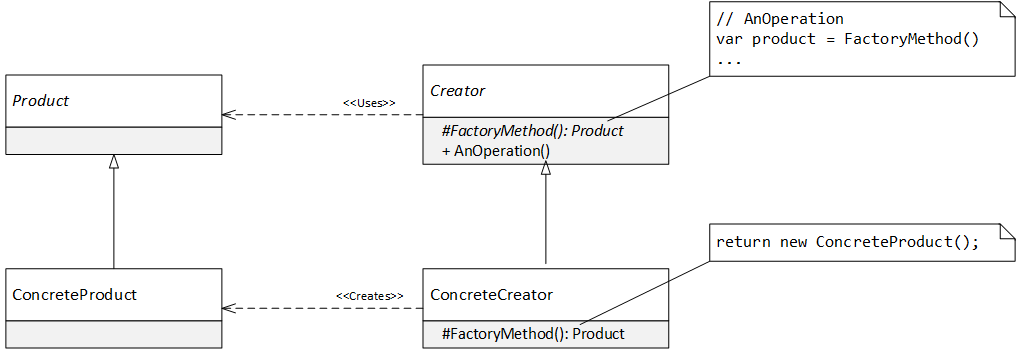


Рисунок 3.1 - Диаграмма классического паттерна Фабричный Метод

Классический Фабричный Метод является частным случаем паттерна Шаблонный Метод, переменный шаг которого отвечает за создание нужного типа объекта.

**Участники**

* Creator (LogReaderBase) объявляет абстрактный или виртуальный метод создания продукта. Использует фабричный метод в своей реализации.
* ConcreteCreator (LogFileReader) реализует фабричный метод, который возвращает ConcreteProduct (FileStream).
* Product (Stream) определяет интерфейс продуктов, создаваемых фабричным методом.
* ConcreteProduct (FileStream) конкретный вид продуктов.

### Статический Фабричный Метод

Самой простой версией Фабричного Метода является статический метод, который создает экземпляр нужного типа в зависимости от переданных аргументов.

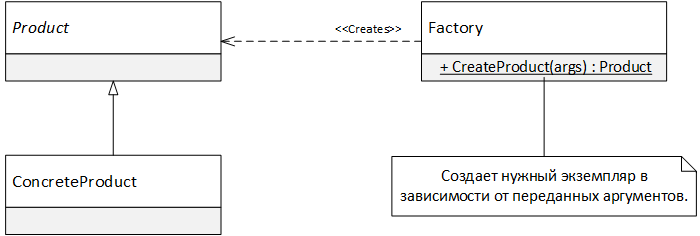


Рисунок 3.2 - Диаграмма статического Фабричного Метода

Основное отличие от классического Фабричного Метода в том, что тип создаваемого фабрикой объекта определяется не типом наследника, а аргументами, переданными методу Create.

### Полиморфный Фабричный Метод

Полиморфный Фабричный Метод определяет интерфейс фабрики, а за создание конкретного экземпляра продукта отвечает конкретная фабрика.

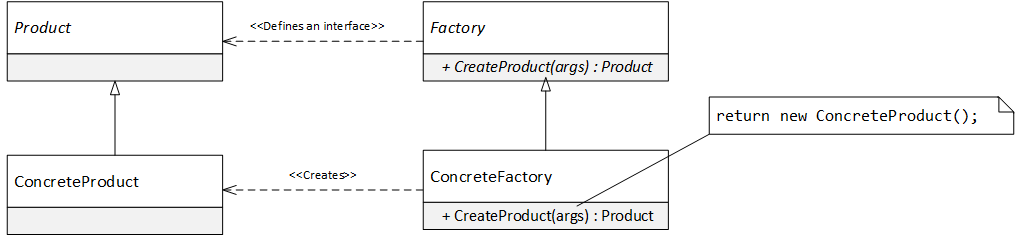


Рисунок 3.3 - Диаграмма полиморфного Фабричного Метода

## Варианты реализации

В .NET существует несколько классических идиом, которые используются для реализации фабрик.

### Использование делегатов в статической фабрике

Типичная реализация статического Фабричного Метода содержит оператор switch, в котором происходит выбор создаваемого типа. Вместо этого, можно воспользоваться словарем, ключом которого будет некоторый идентификатор (строка, перечисление и т.п.), а в качестве значения - делегат, который будет создавать объект нужного класса.

Например, этот подход может быть использован для создания различных классов импорта данных, в зависимости от расширения файла:

static class ImporterFactory  
{  
 private static readonly Dictionary<string, Func<Importer>> \_map =  
 new Dictionary<string, Func<Importer>>();  
   
 static ImporterFactory()  
 {  
 \_map[".json"] = () => new JsonImporter();  
 \_map[".xls"] = () => new XlsImporter();  
 \_map[".xlsx"] = () => new XlsImporter();  
 }  
   
 public static Importer Create(string fileName)  
 {  
 var extension = Path.GetExtension(fileName);  
   
 var creator = GetCreator(extension);  
 if (creator == null)  
 throw new UnsupportedImporterTypeException(extension);  
   
 return creator();  
 }  
   
 private static Func<Importer> GetCreator(string extension)  
 {  
 Func<Importer> creator;  
 \_map.TryGetValue(extension, out creator);  
 return creator;  
 }  
}

Листинг 3.1 - Статический фабричный метод

В статическом конструкторе фабрики все доступные типы регистрируются в словаре \_map, который затем используется для создания нужного класса в методе Create.

### Обобщенные фабрики

Существует проблема, с которой сталкиваются практически все разработчики, не зависимо от используемого языка программирования: как гарантировать вызов виртуального метода при конструировании любого объекта определенной иерархии типов? Вызов виртуального метода в конструкторе базового класса не подходит, поскольку в языке C# это может привести к непредсказуемому поведению, поскольку будет вызван метод наследника, конструктор которого еще не отработал. Можно воспользоваться приемами аспект-ориентированного программирования, а можно воспользоваться фабрикой, которая вызовет виртуальный метод уже после создания экземпляра.

**ПРИМЕЧАНИЕ**  
Использование фабрик для выполнения обязательных действий после создания объекта является достаточно распространенным подходом. Подробнее об этом можно почитать в статье "Better Construction with Factories" by Tal Cohen, Journal of Object Technology, 2002 (http://www.jot.fm/issues/issue\_2007\_07/article3.pdf).

Для решения этой задачи на языке C# нам понадобятся обобщения (generics), отражение (reflection) и немного магии для корректного пробрасывания исключений:

public abstract class Product  
{  
 protected internal abstract void PostConstruction();  
}  
  
public class ConcreteProduct : Product  
{  
 // Внутренний конструктор не позволит клиентам иерархии  
 // создавать объекты напрямую.  
 internal ConcreteProduct() {}  
  
 protected internal override void PostConstruction()  
 {  
 Console.WriteLine("ConcreteProduct: post construction");  
 }  
}  
   
// Единственно законный способ создания объектов семейства Product  
public static class ProductFactory  
{  
 public static T Create<T>() where T : Product, new()  
 {  
 try  
 {  
 var t = new T();  
 // Вызываем постобработку  
 t.PostConstruction();  
   
 return t;  
 }  
 catch (TargetInvocationException e)  
 {  
 // "разворачиваем" исключение и бросаем исходное  
 var edi = ExceptionDispatchInfo.Capture(e.InnerException);  
 edi.Throw();  
 // эта точка недостижима, но компилятор об этом не знает!  
 return default(T);  
 }  
 }  
}

Листинг 3.3 - Обобщенный фабричный метод

Пример использования:

var p1 = ProductFactory.Create<ConcreteProduct>();  
var p2 = ProductFactory.Create<AnotherProduct>();

Листинг 3.4 - Пример использования обобщенного фабричного метода

Обратите внимание на реализацию метода Create и на перехват TargetInvocationException. Поскольку конструкция вида new T() использует отражение для создания экземпляра типа T, то в случае возникновения исключения в конструкторе типа T, исходное исключение будет завернуто в TargetInvocationException. Чтобы упростить работу с нашей фабрикой можно "развернуть" это исключение в методе Create и пробросить исходное исключение с сохранением стека вызовов с помощью ExceptionDispatchInfo (\*).

**ПРИМЕЧАНИЕ**  
Оборачивание исключения в TargetInvocationExpression при вызове new T(), является одним из примеров "дырявых абстракций" (leaky abstractions). Более подробное изложение причин такого поведения и особенностей поведения класса ExceptionDispatchInfo выходит за рамки данной книги. Подробнее о этом можно прочитать в моей статье ["Повторная генерация исключений"](http://sergeyteplyakov.blogspot.com/2011/11/blog-post_03.html).

(\*) Сноска: тип ExceptionDispatchInfo появился лишь в .NET 4.5.

## Обсуждение паттерна Фабричный Метод

У каждой реализации Фабричного Метода есть свои особенности.

* **Классический Фабричный Метод** является частным случаем Шаблонного метода. Это значит, что фабричный метод привязан к текущей иерархии типов и не может быть использован повторно в другом контексте.
* **Полиморфный Фабричный Метод** является стратегией создания экземпляров некоторого семейства типов, что позволяет использовать одну фабрику в разных контекстах. Тип создаваемого объекта определяется типом фабрики, и обычно не зависит от аргументов фабричного метода.
* **Статический Фабричный Метод** является самой простой формой Фабричного Метода. Статический метод создания позволяет обойти ограничения конструкторов. Например, тип создаваемого объекта может зависеть от аргументов метода, экземпляр может возвращаться из кэша, а не создаваться заново, или же фабричный метод может быть асинхронным.

### Сокрытие наследования

Фабричный метод скрывает от своих клиентов детали конструирования объектов. Это бывает полезным, когда процесс создания экземпляра сложен, состоит из нескольких этапов, или когда мы хотим скрыть от клиентов настоящий тип создаваемых объектов.

Давайте вернемся к рассмотренному ранее примеру с классами LogReaderBase и LogFileReader. Сам факт иерархии наследования достаточно просто сделать деталью реализации. Для этого достаточно переименовать класс LogReaderBase в LogReader, и добавить в него два статических фабричных метода FromFile и FromStream:

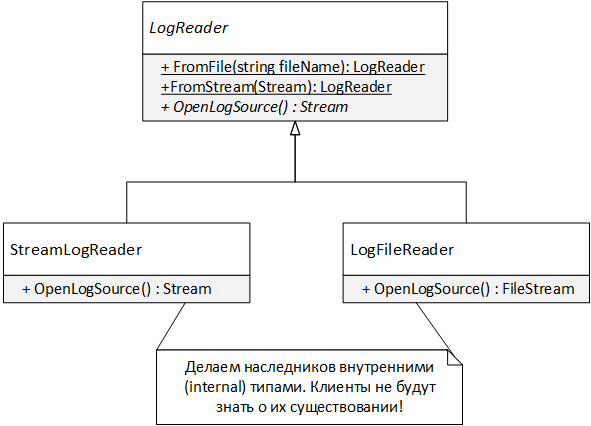


Рисунок 3.4 - Сокрытие иерархии наследования за фабричными методами

**ПРИМЕЧАНИЕ**  
В данном случае иерархия классов LogReader содержит два вида фабричных методов: классический фабричный метод OpenLogSource для создания нужно экземпляра Stream, и два статических фабричных метода FromStream и FromFile.

Данный подход обеспечивает высокую адаптивность решения, поскольку позволяет модифицировать иерархию наследования не затрагивая существующих клиентов.

### Устранение наследования

Рассмотренный ранее подход скрывает наследников от клиентов фабрики, что позволяет избавиться от иерархии наследования, если в ней отпадает необходимость. В случае простых иерархий наследования, как в нашем случае, виртуальные методы можно легко заменить делегатами. Вместо трех классов LogReader, LogFileReader и StreamLogReader достаточно оставить лишь первый, а виртуальный метод OpenLogStream "реализовать" с помощью делегата Func<Stream>:

public class LogReader  
{  
 private readonly Func<Stream> \_streamFactory;  
  
 private LogReader(Func<Stream> streamFactory)  
 {  
 \_streamFactory = streamFactory;  
 }  
  
 public static LogReader FromFile(string fileName)  
 {  
 Func<Stream> factory = () => new FileStream(fileName, FileMode.Open);  
 return new LogReader(factory);  
 }  
  
 public static LogReader FromStream(Stream stream)  
 {  
 Func<Stream> factory = () => stream;  
 return new LogReader(factory);  
 }  
  
 public IEnumerable<LogEntry> Read()  
 {  
 using (var stream = OpenLogSource())  
 {  
 using (var reader = new StreamReader(stream))  
 {  
 string line = null;  
 while ((line = reader.ReadLine()) != null)  
 {  
 yield return LogEntryParser.Parse(line);  
 }  
 }  
 }  
 }  
  
 private Stream OpenLogSource()  
 {  
 return \_streamFactory();  
 }  
}

Листинг 3.5 - Реализация класса LogReader на основе делегатов

### Использование Func в качестве фабрики

В предыдущем разделе, мы использовали Func<Stream> в качестве детали реализации. В некоторых случаях, Func<T> может использоваться в качестве полноценной фабрики и передаваться классу извне его клиентами.

Данный вариант фабрики является допустимым во внутреннем (internal) коде и только для функций с небольшим числом аргументов. Понять, что делает Func<Stream> довольно просто, но разобраться в назначении Func<int, string, int, ValidationResult> factory без контекста будет практически невозможно. В случае повторно используемого кода понятность кода является очень важной, поэтому именованная фабрика является более предпочтительным вариантом.

### Конструктор vs. Фабричный метод

В объектно-ориентированных языках программирования конструктор отвечает за корректную инициализацию создаваемого объекта. В большинстве случаев они прекрасно справляется со своей задачей, но иногда лучше воспользоваться статическим фабричным методом.

#### Именованные конструкторы

В языке C# имя конструктора совпадает с именем класса, что делает невозможным использование двух конструкторов с одним набором и типом параметров. Хорошим примером такого ограничения является структура Timespan, которая представляет собой интервал времени. Очень удобно создавать интервал времени по количеству секунд, минут, часов и дней, но сделать несколько конструкторов, каждый из которых принимает один параметр типа double невозможно. Для этого структура Timespan содержит набор фабричных методов:

public struct Timespan  
{  
 public Timespan(double ticks) { ... }  
 public static Timespan FromMilliseoncds(double value) {...}  
 public static Timespan FromSeconds(double value) {...}  
 public static Timespan FromMinutes(double value) {...}  
 // Остальные фабричные методы  
}

Листинг 3.5 - Пример фабричных методов в качестве именованных конструкторов

#### Тяжеловесный процесс создания

Конструктор отвечает за корректную инициализацию объекта, после которой объект должен быть готовым для использования своими клиентами. Обычно логика инициализации относительно простая и должна выполняться конструктором, но слишком тяжеловесную логику лучше вынести из конструктора в статический фабричный метод (\*).

(\*) Сноска: Некоторые специалисты считают, что наличие в конструкторе инструкции if является признаком плохого кода. Достаточно поискать в вашем любимом поисковом сервисе фразу "bloated constructor", чтобы понять, что я не шучу!

Я использую статические фабричные методы, когда сложность или время исполнения конструктора переходит определенную черту. Если для конструирования объекта требуется обращение к внешним ресурсам, то я предпочитаю сразу же выделять эту логику в фабричный метод. Это позволит сделать фабричный метод асинхронным, а также упростит эволюцию решения и разбиение данного класса на более мелкие составляющие в случае необходимости.

TODO: тут был раздел Фабрика vs. Абстрактная Фабрика vs. Фабричный метод, но я решил его убрать. Текста и так получилось много, и я не уверен, что этот раздел является ценным.

## Применимость паттерна Фабричный Метод

Разные виды Фабричного Метода применяются для решения разных задач.

### Применимость классического Фабричного Метода

Классический Фабричный Метод очень редко появляется в результате тщательного проектирования. Будучи частным случаем Метода Шаблона он естественным образом возникает в иерархии наследования, когда базовый класс определяет некоторый алгоритм, одним из этапом которого является конструирование объекта. При этом решение о типе объекта не может быть принято на его уровне и переносится на уровень наследников.

### Применимость полиморфного Фабричного Метода

* **Стратегия создания объектов**. Полиморфный Фабричный Метод применяется для выделения стратегии создания объектов, чтобы отвязать клиентов от создаваемых ими зависимостей.
* **Конфигурирование процесса создания**. В некоторых случаях экземплярная фабрика может быть использована для создания объектов с нужными характеристиками. Например, класс TaskFactory из Task Parallel Library используется для создания задач с заданным планировщиком, маркером отмены и т.д. При этом гибкость обеспечивается не за счет наследования, а за счет конфигурирования объекта фабрики.

### Применимость статического Фабричного Метода

* **Сокрытие иерархии наследования**. Набор статических фабричных методов может скрывать от своих клиентов глубину и даже наличие иерархии наследования, как это делает WebRequest.Create(string) и класс LogReader, рассмотренный ранее.
* **Именованный конструктор**. Фабрика используется для устранения неоднозначности, когда объект может быть создан по аргументам одного типа, но с разным значением. Например, Timespan.FromSeconds, Timespan.FromMilliseconds etc.
* **Для вызова виртуального метода после создания объектов**. Как мы видели ранее, статический фабричный метод может быть использован для эмуляции «вызова виртуального метода в конструкторе базового класса», когда требуется выполнить обязательные действия после создания всех объектов определенной иерархии типов.
* **Фасад для сложного процесса создания**. В некоторых случаях процесс создания может быть достаточно сложным, чтобы изолировать его в отдельном статическом методе. Например, фабричный метод может использовать паттерн Строитель, создавать объект довольно сложным образом или обращаться к внешним ресурсам.
* **Асинхронный конструктор**. Если длительность конструирования относительно велика, то статический фабричный метод можно сделать асинхронным, что невозможно в случае использования конструктора.
* **Кэширование**. Статический фабричный метод позволит возвращать экземпляры из кэша, а не создавать их каждый раз заново.

## Примеры фабрик в .NET Framework

В .NET Framework применяется огромное число фабрик.

* **Классический фабричный метод**: Stream.CreateWaitHandle, SecurityAttribute.CreatePermission, ChannelFactory.CreateChannel, XmlNode.CreateNavigator.
* **Полиморфная фабрика**: \* IControllerFactory в ASP.NET MVC, IHttpHandlerFactory в ASP.NET, ServiceHostFactory и IChannelFactory<T> в WCF, IQueryProvider в LINQ.
* **Неполиморфная фабрика**: TaskFactory в TPL.
* **Обобщенная статическая фабрика**: Activator.CreateInstance, Array.CreateInstance, StringComparer.Create.
* **Сокрытие наследников**: RandomNumberGenerator.Create, WebRequest.Create, BufferManager.CreateBufferManager, Message.CreateMessage, MessageFault.CreateFault.
* **Фасадные фабричные методы**. File.Create, File.CreateText.
* **Именованные конструкторы**. Timespan.FromSecond, Timespan.FromMilliseconds; GCHandle.FromIntPtr, Color.FromArgb