# Паттерн Строитель (Builder)

**Назначение**: *Строитель отделяет конструирование сложного объекта от его представления, так что в результате одного и того же процесса конструирования могут получаться разные представления.*

## Мотивация

Фабричный метод скрывает от своих клиентов процесс создания и конкретный тип возвращаемого объекта. Фабрика позволяет развивать иерархию типов, не затрагивая существующих клиентов, но в некоторых случаях требуется гибкость иного рода.

Иногда процесс создания является довольно сложным, состоит из нескольких этапов. Создаваемому объекту нужно передать множества аргументов, часть из которых нужны одним клиентам, но не нужны другим. В этом случае фабричный метод с 10 аргументами, 9 из которых будет повторяться каждый раз, нельзя назвать удачным решением. В этом случае поможет другой порождающий паттерн - Строитель.

Давайте рассмотрим задачу создания объектов Email для отправки сообщений электронной почты. Часть свойств объекта Email являются обязательными, а часть - нет, количество приложений (attachment) и получателей может быть произвольным и т.п. Электронные сообщения можно создавать руками, задавая нужные свойства, как это делается при работе с классом MailMessage из .NET Framework, а можно выделить процесс создания в отдельный класс - MailMessageBuilder:

public sealed class MailMessageBuilder  
{  
 private readonly MailMessage \_mailMessage = new MailMessage();  
  
 public MailMessageBuilder From(string address)  
 {  
 \_mailMessage.From = new MailAddress(address);  
 return this;  
 }  
  
 public MailMessageBuilder To(string address)  
 {  
 \_mailMessage.To.Add(address);  
 return this;  
 }  
  
 public MailMessageBuilder Cc(string address)  
 {  
 \_mailMessage.CC.Add(address);  
 return this;  
 }  
  
 public MailMessageBuilder Subject(string subject)  
 {  
 \_mailMessage.Subject = subject;  
 return this;  
 }  
  
 public MailMessageBuilder Body(string body, Encoding enconding)  
 {  
 \_mailMessage.Body = body;  
 \_mailMessage.BodyEncoding = enconding;  
 return this;  
 }  
  
 public MailMessage Build()  
 {  
 return \_mailMessage;  
 }  
}

Листинг 4.1 – Строитель сообщений электронной почты

Теперь этот класс можно использовать так:

var mail = new MailMessageBuilder()  
 .From("st@unknown.com")  
 .To("support@microsof.com")  
 .Cc("my\_boss@unknown.com")  
 .Subject("Msdn is down!")  
 .Body("Please fix!", Encoding.UTF8)  
 .Build();  
   
new SmtpClient().Send(mail);

Листинг 4.2 – Пример использования класса MailMessageBuilder

Использование паттерна Cтроитель позволяет более четко разграничить ответственности между создателем и потребителем объектов, а также делает процесс создания более удобным в использовании.

## Классическая диаграмма классов паттерна Строитель

Паттерн Строитель применяется довольно часто в современных приложениях, но не в том виде, в котором он был описан бандой четырех. На практике Строитель все так же отвечает за создание объектов, но гораздо реже обладает всеми изначальными свойствами.

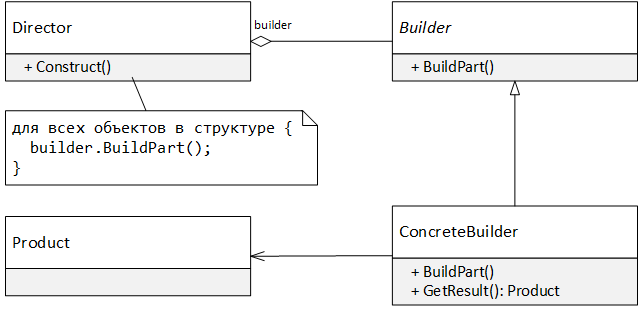


Рисунок 4.1 - Классическая диаграмма паттерна Строитель

**Участники**

* Builder определяет интерфейс конструирования продукта по частям.
* Director управляет процессом создания, не зная, какой продукт будет создан в результате.
* ConcreteBuilder конкретный строитель, который создает только известный ему объект класса Product.

Обратите внимание на два момента: 1) наличие наследования и 2) о классе Product знает только конкретный строитель (ConcreteBuilder). Ни базовый класс строителя, ни его клиент (класс Director) не знают о типе создаваемого продукта. Это развязывает руки конкретным строителям, которые могут формировать совершенно разнородные объекты: от сообщения электронной почты, до строки в формате Json. Но это оставляет открытыми многие вопросы: а кто потребляет созданный продукт? как конкретный строитель узнает об этом потребителе? всегда ли процесс формирования и потребления продуктов должен быть так удален друг от друга: продукт конструируется классом Director, а потребляется не понятно кем? и т.п.

На практике обычно используется более простая разновидность паттерна Строитель, без наследования и с более явной моделью взаимодействия между участниками.

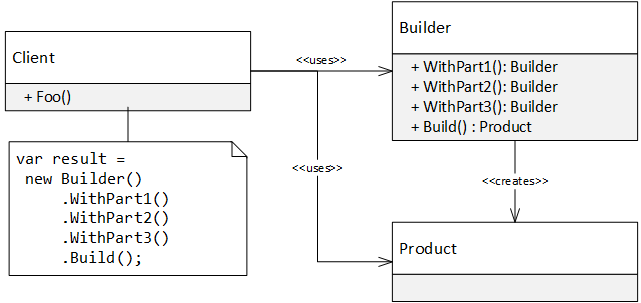


Рисунок 4.2 - Упрощенная диаграмма паттерна Строитель

Участники в этом случае остались теми же самыми, но ответственности немного изменяется. Director (клиент класса MailMessageBuilder) управляет созданием сложного объекта и получает созданный объект путем вызова метода Build. Builder (MailMessageBuilder) отвечает за создание конкретного продукта (MailMessage).

## Особенности реализации в .NET

При работе со строителями в .NET есть два специфических момента: использование "текучего интерфейса" (fluent interface) и возможность использования методов расширения.

### Использование текучего интерфейса

Текучий интерфейс предназначен для повышения читабельности кода. Каждый метод возвращает объект, над которым выполняется текущая операция, что позволяет связывать методы в цепочку.

var result = Enumerable.Range(1, 10).Select(n => n + 1).Count();

Листинг 4.3 – Пример использования текучего интерфейса

Текучий интерфейс изначально появился в мире Java и активно используется для создания мини-DSL (Domain-Specific Language). При реализации строителей данная идиома используется постоянно, поскольку позволяет создавать требуемый объект одним оператором:

public override string ToString()  
{  
 return new StringBuilder()  
 .AppendFormat("Id: {0}", Id).AppendLine()  
 .AppendFormat("Name: {0}", Name).AppendLine()  
 .ToString();  
}

Листинг 4.4 – Пример использования класса StringBuilder

### Методы расширения

Методы расширения (extension methods) позволяют создать видимость, что мы добавляем операции в существующие типы. В некоторых случаях, можно создать набор методов расширения, которые будут выполнять роль строителя для существующих типов.

Вместо использования объекта MailMessageBuilder, можно обойтись методами расширения над классом MailMessage:

public static class MailMessageBuilderEx  
{  
 public static MailMessage From(this MailMessage mailMessage, string address)  
 {  
 mailMessage.From = new MailAddress(address);  
 return mailMessage;  
 }  
  
 public static MailMessage To(this MailMessage mailMessage, string address)  
 {  
 mailMessage.To.Add(address);  
 return mailMessage;  
 }  
  
 public static MailMessage Cc(this MailMessage mailMessage, string address)  
 {  
 mailMessage.CC.Add(address);  
 return mailMessage;  
 }  
  
 public static MailMessage Subject(this MailMessage mailMessage, string subject)  
 {  
 mailMessage.Subject = subject;  
 return mailMessage;  
 }  
  
 public static MailMessage Body(this MailMessage mailMessage,   
 string body, Encoding enconding)  
 {  
 mailMessage.Body = body;  
 mailMessage.BodyEncoding = enconding;  
 return mailMessage;  
 }  
}

Листинг 4.5 – Использование методов расширения для реализации строителя

Пример использования будет практически аналогичен примеру с классом MailMessageBuilder:

var mail = new MailMessage()  
 .From("st@unknown.com")  
 .To("support@microsof.com")  
 .Cc("my\_boss@unknown.com")  
 .Subject("Msdn is down!")  
 .Body("Please fix!", Encoding.UTF8);  
   
new SmtpClient().Send(mail);

Листинг 4.6 – Пример использования методов расширения

Разница лишь в том, что методы расширения манипулируют уже созданным продуктом, просто делают это более декларативным образом.

**ПРИМЕЧАНИЕ**  
Подобные методы расширения не всегда могут заменить полноценный строитель. Строитель обладает большей гибкостью: он может накапливать состояние в собственных полях, что даст больше свободы при валидации состояния создаваемого объекта. Строитель может проверять накопленное состояние и генерировать исключение в методе Build, если клиент еще не вызвал один из обязательных методов.

## Обсуждение паттерна Строитель

При описании диаграммы классов паттерна Строитель, уже было сказано, что в классическом виде этот паттерн применяется довольно редко. Возможность создания разнообразных продуктов нужна относительно редко и возрастающая при этом сложность не всегда окупается.

Если посмотреть на примеры строителей в .NET Framework, то все они аналогичны рассмотренному ранее классу MailMessageBuilder - это специализированные классы для создания конкретных продуктов - строк, коллекций, контейнеров и т.д.

В случае паттерна Строитель разумно следовать той же логике, что и при выделении стратегий. Начните с конкретного класса Builder, предназначенного для создания конкретных продуктов. Выделяйте стратегию конструирования, такую как IBuilder, лишь тогда, когда в этом будет четкая необходимость.

### "Строго типизированный" строитель

TODO: Не уверен в необходимости этого раздела. Нужна обратная связь от рецензентов!!

В некоторых случаях процесс создания может содержать обязательные и необязательные этапы. Так, в случае класса MailMessageBuilder можно запретить конструирование объекта MailMessage с пустым списком получателей. Для этого, метод Build может проверять состояние конструированного объекта и генерировать исключение (например, InvalidOperationException), если поле To пусто:

public MailMessage Build()  
{  
 if (\_mailMessage.To.Count == 0)  
 {  
 throw new InvalidOperationException(  
 "Can't create a mail message with empty To. Please call 'To' method first");  
 }  
  
 return \_mailMessage;  
}

Листинг 4.7 – Генерация исключения в методе Build

Вместо генерации исключений во время исполнения, можно изменить дизайн таким образом, чтобы эта ошибка была недопустимой во время компиляции.

1. Использование конструктора

Самый простой способ добиться этого, передать обязательные аргументы в конструкторе Строителя:

public class MailMessageBuilder  
{  
 public MailMessageBuilder(string address)  
 {  
 To(address);  
 }  
  
 // Остальные методы остались без изменения

Листинг 4.8 – Добавление конструктора в класс MailMessageBuilder

Проблема этого подхода в том, что он плохо расширяем, если количество обязательных этапов станет больше 3-4. Если обязательная часть более сложная, то можно воспользоваться вторым подходом.

2. Использование паттерна

Состояние Более сложный в реализации подход основан на использовании варианта паттерна Состояние. Можно создать набор строителей, каждый из которых будет содержать лишь операцию для перевода объекта к следующему этапу. При этом метод Build будет доступен лишь на последнем этапе, когда все нужные этапы были проинициализированы.

В простом виде, данный подход будет выглядеть так:

public class MailMessageBuilder  
{  
 private readonly MailMessage \_mailMessage = new MailMessage();  
  
 internal MailMessageBuilder(MailMessage mailMessage)  
 {  
 \_mailMessage = mailMessage;  
 }  
  
 public FinalMailMessageBuilder To(string address)  
 {  
 \_mailMessage.To(address);  
 // Для большей эффективности может быть добавлено кэширование  
 return new FinalMailMessageBuilder(\_mailMessage);  
 }  
  
 // Остальные методы остались без изменения  
}  
  
public class FinalMailMessageBuilder  
{  
 private readonly MailMessage \_mailMessage;  
  
 internal FinalMailMessageBuilder(MailMessage mailMessage)  
 {  
 \_mailMessage = mailMessage;  
 }  
  
 public MailMessage Build()  
 {  
 return \_mailMessage;  
 }  
}

Листинг 4.9 – Пример строго типизированного строителя

Для обязательного этапа выделен специализированный тип строителя - FinalStringBuilder, который можно получить вызовом метода To основного строителя. Этот подход гарантирует, что при вызове метода Build получатель был установлен и создаваемый объект будет в корректном состоянии.

**ПРИМЕЧАНИЕ**  
Данный подход устраняет ошибки времени исполнения, но за это приходится платить дополнительными усилиями во время разработки. В простых случаях в методе Build достаточно генерировать исключение InvalidOperationException с понятным сообщением об ошибке, а строго типизированный вариант оставить на тот случай, когда важность корректного использования очень высока. Еще одна проблема этого подхода заключается в выделении нового экземпляра строителя для каждого нового этапа. Избавиться от нее можно путем замены классов на структуры.

### Создание неизменяемых объектов

Паттерн Строитель четко разделяет процесс создания продукта от его потребления. Это разделение особенно полезно, если продукт физически или логически является неизменяемым (immutable).

Конструирование неизменяемых объектов является дорогостоящим, поскольку каждая команда (операция, изменяющая состояние) создает новый экземпляр. В .NET строки являются неизменяемыми, поэтому в состав .NET Framework был добавлен класс StringBuilder, предназначенный для более эффективного создания строк.

Паттерн Строитель очень часто идет в комплекте с другими неизменяемыми структурами данных, например, неизменяемыми коллекциями (\*). Добавление нового элемента в неизменяемую коллекцию приводит к гарантированному выделению памяти и созданию новой коллекции. В следующем примере будет создано 10 экземпляров коллекции ImmutableList<int>:

(\*) Сноска: неизменяемые коллекции (immutable collections) входят в состав .NET Framework, но распространяются отдельным пакетом. Для их использования с помощью пакетного менеджера nuget нужно добавить пакет System.Collections.Immutable for .NET 4.0.

var list = ImmutableList<int>.Empty;  
foreach (var element in Enumerable.Range(1, 10))  
{  
 // Объект list пересоздается 10 раз!  
 list = list.Add(element);  
}

Листинг 4.10 – Пример работы с неизменяемой коллекцией

Чтобы избежать дополнительных аллокаций памяти, каждая неизменяемая коллекция содержит внутренний Строителя, с помощью которого создание коллекции будет более эффективным:

ImmutableList<int>.Builder builder =   
 ImmutableList<int>.Empty.ToBuilder();  
  
foreach (var element in Enumerable.Range(1, 10))  
{  
 builder.Add(element);  
}  
  
ImmutableList<int> list = builder.ToImmutable();

Листинг 4.11 – Использования строителя для создания неизменяемой коллекции

### Частичная изменяемость

Неизменяемые коллекции показывают хороший пример частичной неизменяемости. С точки зрения внешних клиентов объекты класса ImmutableList<T> являются неизменяемым. Но есть одно исключение: вложенный класс ImmutableList<T>.Builder, который может получить непосредственный доступ к закрытым полям класса ImmutableList<T>.

В некоторых случаях можно пойти еще дальше и убрать из продукта все команды, оставить лишь набор свойств только для чтения. И предоставить единственный интерфейс создания объектов через строителя. Если бы я создавал класс MailMessage самостоятельно, то мог бы прийти к такому решению:

public class MailMessage  
{  
 private string \_to;  
 private string \_from;  
 private string \_subject;  
 private string \_body;  
  
 private MailMessage()  
 {}  
  
 public static MailMessageBuilder With()  
 {  
 return new MailMessageBuilder(new MailMessage());  
 }  
  
 public string To { get { return \_to; } }  
 public string From { get { return \_from; } }  
 public string Subject { get { return \_subject; } }  
 public string Body { get { return \_body; } }  
  
 public class MailMessageBuilder {...}  
}

Листинг 4.12 – Неизменяемый класс MailMessage

Класс MailMessage является полностью неизменяемым, с единственным фабричным методом With, который возвращает строителя. MailMessageBuilder является вложенным классом в MailMessage, а значит имеет доступ к его закрытым членам:

// Объявлен внутри класса MailMessage  
public class MailMessageBuilder  
{  
 private readonly MailMessage \_mailMessage;  
  
 internal MailMessageBuilder(MailMessage mailMessage)  
 {  
 \_mailMessage = mailMessage;  
 }  
  
 public MailMessageBuilder To(string to)  
 {  
 \_mailMessage.\_to = to;  
 return this;  
 }  
  
 public MailMessageBuilder From(string from)  
 {  
 \_mailMessage.\_from = from;  
 return this;  
 }  
  
 public MailMessageBuilder Subject(string subject)  
 {  
 \_mailMessage.\_subject = subject;  
 return this;  
 }  
  
 public MailMessageBuilder Body(string body)  
 {  
 \_mailMessage.\_body = body;  
 return this;  
 }  
  
 public MailMessage Build()  
 {  
 return \_mailMessage;  
 }  
}

Листинг 4.13 – Реализация вложенного строителя

Клиенты класса MailMessage вынуждены использовать Строитель для задания объекта MailMessage, который после создания является неизменяемым:

var mailMessage =   
 MailMessage.With()  
 .From("st@unknown.com")  
 .To("support@microsof.com")  
 .Subject("Msdn is down!")  
 .Body("Please fix!")  
 .Build();  
  
Console.WriteLine(mailMessage.To);

Листинг 4.14 – Пример использования неизменяемого типа со строителем

## Применимость

Паттерн Строитель идеально подходит для ситуаций, когда процесс создания является сложным, состоит из нескольких этапов, при этом одним клиентам нужно устанавливать одни параметры создаваемого объекта, а другим клиентам - другие. Строитель может устанавливать разумные значения по умолчанию, предоставляя клиентам сосредоточиться лишь на важных для них параметрах.

Строитель обычно обладает текучим интерфейсом, что делает его использование более читабельным и декларативным.

Я обратил внимание, что в последнее время достаточно часто пользуюсь этим паттерном. Создание тестовых данных, создание и конфигурирование классов экспорта данных (источник, тип получателя, какие данные нужно экспортировать), формирование слабо типизированных объектов, таких как JSON или Xml с помощью строго типизированного Fluent API и т.п.

Строитель идеально сочетается с неизменяемыми классами. Неизменяемость упрощает понимание кода и прекрасно подходят для использования в многопоточной среде. Наличие строителей позволяет обойти ограничения неизменяемости и решить проблемы эффективности, которые обязательно возникнут при работе с такими типами.

## Примеры в .NET Framework

Паттерн Строитель достаточно часто используется в .NET Framework:

* StringBuilder, UriBuilder, DbCommandBuilder и DbConnectionStringBuilder из ADO.NET.
* Строители неизменяемых коллекций: ImmutableList<T>.Builder, ImmutableDictionary<TKey, TValue>.Builder etc.
* Типы из Reflection.Emit: ModuleBuilder, TypeBuilder, EnumBuilder, MethodBuilder etc.
* В WCF используется довольно много внутренних (internal) строителей: ChannelBuilder, DispatcherBuilder, EndpointAddressBuilder etc.
* В autofac, довольно популярном IoC-контейнере, паттерн Строитель используется для разделения этапов конфигурирования и использования контейнеров.