# Паттерн Шаблонный Метод (Template Method)

**Назначение:** Шаблонный Метод определяет основу алгоритма и позволяет подклассам переопределять некоторые шаги алгоритма, не изменяя его структуру в целом.

**Другими словами**: шаблонный метод – это каркас, в который наследники могут подставить реализации недостающих элементов.

## Мотивация

На заре становления ООП, наследование считалось ключевым механизмом для расширения и повторного использования кода. Однако со временем, многим разработчикам стало очевидно, что наследование не такой уж и простой инструмент, использование которого приводит к сильной связности (tight coupling) между базовым классом и его наследником. Эта связность приводит к сложности понимания иерархии классов, а отсутствие формализации отношений между базовым классом и наследниками не позволяет четко понять, как именно разделены обязанности между классами Base и Derived, что можно делать наследнику, а что – нет.

Наследование подтипов подразумевает возможность подмены объектов базового класса объектами классов наследников. Такое наследование моделирует отношение "ЯВЛЯЕТСЯ" и поведение наследника должно соответствовать принципу наименьшего удивления: все, что корректно работало с базовыми классами должно работать и с наследниками.

Более формальные отношения между "родителями" и "потомками" описываются с помощью предусловий и постусловий. Эта техника лежит в основе принципа подстановки Лисков, который мы рассмотрим в одной из глав этой книги. Другой способ заключается в использовании паттерна "Шаблонный Метод", который позволяет более четко определить "контракт" между базовым классом и потомками.

Давайте вернемся к теме разработке приложения по импорту логов для полнотекстового поиска. Процесс чтения лога состоит из нескольких этапов:

1. Прочитать новые записи с места последнего чтения.
2. Разобрать их и вернуть вызывающему коду.

Можно в каждой реализации продублировать эту логику, или же можно описать основные шаги алгоритма в базовом классе и "отложить" реализацию конкретных шагов на момент реализации наследников.

public abstract class LogReader

{

private int \_currentPosition;

// Метод ReadLogEntry невиртуальный: определяет алгоритм импорта

public IEnumerable<LogEntry> ReadLogEntry()

{

return ReadEntries(ref \_currentPosition).Select(ParseLogEntry);

}

protected abstract IEnumerable<string> ReadEntries(ref int position);

protected abstract LogEntry ParseLogEntry(string stringEntry);

}

Листинг 2.1 – Класс LogReader

Теперь все реализации читателей логов будут вынуждены следовать согласованному протоколу. Классу LogFileReader достаточно будет реализовать методы ReadEntries и ParseLogEntry и не думать о порядке вызова этих методов или о необходимости вызова базовой реализации виртуального метода.

Шаблонный Метод позволяет создать небольшой каркас (framework) для решения определенной задачи, когда базовый класс описывает основные шаги решения, заставляя наследников предоставить недостающие куски головоломки.

## Классическая диаграмма классов паттерна Шаблонный Метод

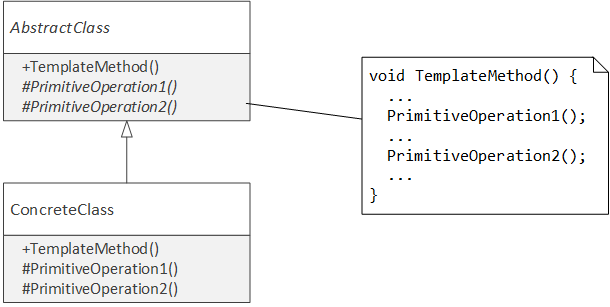


Рисунок 2.1 - Диаграмма классов паттерна Шаблонный Метод

**Участники**

* AbstractClass (LogReader) определяет невиртуальный метод TemplateMethod (ReadLogEntry), который вызывает внутри примитивные операции: PrimitiveOperation1(), PrimitiveOperation2() и т.д. (ReadEntries иParseLogEntry).
* ConcreteClass (LogFileReader) реализует примитивные шаги алгоритма.

Шаблонный метод - это один из классических паттернов, который и по сей день используется в каноническом виде во многих приложениях.

## Варианты реализации в .NET

### "Локальный Шаблонный Метод" на основе делегатов

Классический вариант паттерна Шаблонный Метод подразумевает, что каркас алгоритма описывается в базовом классе, а "переменные шаги алгоритма" задаются наследниками путем переопределения абстрактных или виртуальных методов. Но в некоторых случаях схожие операции с единым "каркасом" исполнения и переменными составляющими бывают в рамках одного класса. Использование наследования является слишком тяжеловесным решением, поэтому в таких случаях применяется подход, когда переменный шаг алгоритма задается делегатом.

Данный подход устраняет дублирование кода и применяется довольно часто в современных .NET приложениях. Шаблонный Метод на основе делегатов постоянно используется при работе с WCF сервисами, поскольку "протокол работы" с прокси объектами довольно сложен, и отличается лишь конкретным методом сервиса.

// Интерфейс сервиса сохранения записей

interface ILogSaver

{

void UploadLogEntries(IEnumerable<LogEntry> logEntries);

void UploadExceptions(IEnumerable<ExceptionLogEntry> exceptions);

}

// Прокси класс инкапсулирует особенности работы

// с WCF инфраструктурой

class LogSaverProxy : ILogSaver

{

class LogSaverClient : ClientBase<ILogSaver>

{

public ILogSaver LogSaver

{

get { return Channel; }

}

}

public void UploadLogEntries(IEnumerable<LogEntry> logEntries)

{

UseProxyClient(c => c.UploadLogEntries(logEntries));

}

public void UploadExceptions(IEnumerable<ExceptionLogEntry> exceptions)

{

UseProxyClient(c => c.UploadExceptions(exceptions));

}

private void UseProxyClient(Action<ILogSaver> accessor)

{

var client = new LogSaverClient();

try

{

accessor(client.LogSaver);

client.Close();

}

catch (CommunicationException e)

{

client.Abort();

throw new OperationFailedException(e);

}

}

}

Листинг 2.2 – Использование Шаблонного Метода с WCF

Теперь для добавления нового метода сервиса достаточно добавить лишь одну строку кода, при этом протокол работы с сервисом будет соблюден.

**ПРИМЕЧАНИЕ**  
Подробнее о том, почему с WCF прокси нужно работать именно таким образом можно прочитать в разделе "Closing the proxy and using statement" книги "Programming WCF Services" by Juval Lowy.

Подход на основе делегатов может применяться не только для определения "локальных" действий внутри класса, но и передаваться извне другому объекту в аргументах конструктора. В этом случае грань между Шаблонным Методом и Стратегией стирается практически полностью, разница остается лишь на логическом уровне: Стратегия, даже представленная в виде делегата, обычно подразумевает законченное действие, в то время, как "переменный шаг" Шаблонного Метода обычно является более контекстно-зависимой операцией(\*).

(\*) СНОСКА: более подробно речь о связи Шаблонного Метода и Стратегии пойдет в соответствующем разделе этой главы.

**Шаблонный метод на основе методов расширения**

В языке C# существует возможность добавления операции существующим типам с помощью [методов расширения](http://msdn.microsoft.com/ru-ru/library/bb383977(v=vs.90).aspx) (Extension Methods). Обычно методы расширения используются для расширения кода, который находится вне нашего контроля: библиотечные классы, перечисления, классы сторонних производителей. Но эта же возможность может использоваться и для своего собственного кода, что позволит выделить функциональность во вспомогательные классы разгружая при этом основные.

Для удобства диагностики было бы разумно иметь возможность получения строкового представления прочитанных записей. Прочитанные записи сами по себе формируют небольшую иерархию наследования:

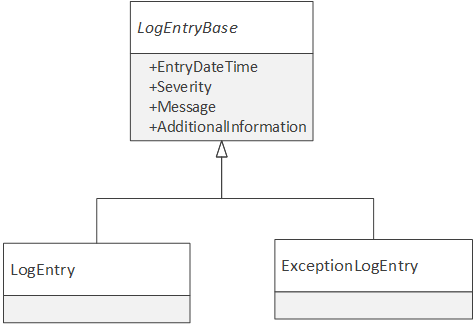


Рисунок 2.1 - Иерархия классов LogEntry

Мы могли бы переопределить метод ToString(), или же вынести эту ответственность в методы расширения:

public abstract class LogEntryBase

{

public DateTime EntryDateTime { get; internal set; }

public Severity Severity { get; internal set; }

public string Message { get; internal set; }

// ExceptionLogEntry будет возвращать информацию об исключении

public string AdditionalInformation { get; internal set; }

}

public static class LogEntryBaseEx

{

public static string GetText(this LogEntryBase logEntry)

{

var sb = new StringBuilder();

sb.AppendFormat("[{0}] ", logEntry.EntryDateTime)

.AppendFormat("[{0}] ", logEntry.Severity)

.AppendLine(logEntry.Message)

.AppendLine(logEntry.AdditionalInformation);

return sb.ToString();

}

}

Листинг 2.3 – Шаблонный метод на основе методов расширения

У этого подхода есть свои преимущества и недостатки.

**Недостатки**

* "Переменные шаги алгоритма" должны определяться открытыми методами.
* Могут потребоваться приведения к конкретным типам наследников, если не вся информация доступна через призму базового класса (нарушение принципа Открыт/Закрыт).

**Достоинства**

* Упрощается исходный класс (следование принципу Открыт/Закрыт).
* Класс и его методы расширения могут находиться в разных пространствах имен. Это позволит клиентам самостоятельно решать, нужно ли им импортировать метод расширения или нет (следование принципу Разделения интерфейса).
* Существует возможность разных реализаций метода, в зависимости от контекста и потребностей. Метод GetText может иметь одну реализацию в серверной части, и другую - в клиентской.

## Обсуждение паттерна Шаблонный Метод

### Изменение уровня абстракции

Шаблонный метод может применяться классом-наследником повторно при реализации "переменного шага алгоритма", объявленного в базовом классе.

В случае импорта лог-файлов, можно выделить отдельный абстрактный базовый класс LogFileReaderBase, который будет содержать логику чтения файла. Наследнику останется лишь переопределить операцию разбора прочитанной строки.

public abstract class LogFileReaderBase : LogImporter, IDisposable

{

private readonly Lazy<Stream> \_stream;

protected LogFileReaderBase(string fileName)

{

\_stream = new Lazy<Stream>(() => new FileStream(fileName, FileMode.Open));

}

public void Dispose()

{

if (\_stream.IsValueCreated)

{

\_stream.Value.Close();

}

}

protected override sealed IEnumerable<string> ReadEntries(ref int position)

{

Contract.Assert(\_stream.Value.CanSeek);

if (\_stream.Value.Position != position)

\_stream.Value.Seek(position, SeekOrigin.Begin);

return ReadLineByLine(\_stream.Value, ref position);

}

protected override abstract LogEntry ParseLogEntry(string stringEntry);

private IEnumerable<string> ReadLineByLine(Stream stream, ref int position)

{

// Построчное чтение из потока ввода/вывода

}

}

Листинг 2.4 – Базовый класс LogFileReaderBase

Бывает, что Шаблонный Метод "скользит" по иерархии наследования. Исходный "переменный шаг алгоритма" может оказаться довольно сложной операцией, что потребует ее дальнейшей декомпозиции. В результате может появиться еще один абстрактный класс, который еще сильнее разобьет исходные шаги алгоритма на еще более простые и конкретные этапы.

### Стратегия vs. Шаблонный метод

В некоторых случаях, переменный шаг алгоритма является достаточно самостоятельной операцией, которую лучше спрятать в отдельной абстракции.

Например, в предыдущем примере, всю ответственность за разбор строки можно выделить в отдельный аспект - в стратегию разбора записи. При этом, данная стратегия может передаваться как на самом базовом уровне, так и являться деталью реализации конкретного импортера. Вполне возможно, что стратегия разбора будет применяться для всех файловых импортеров, но не применяться в других случаях.

public interface ILogParser

{

LogEntry ParseLogEntry(string stringEntry);

}

public abstract class LogFileReaderBase : LogImporter

{

protected(string fileName, ILogParser logParser)

{

// ...

}

protected override LogEntry ParseLogEntry(string stringEntry)

{

return \_logParser.ParseLogEntry(stringEntry);

}

// ...

}

Листинг 2.5 – Класс LogFileReaderBase, принимающий стратегию ILogParser

**ПРИМЕЧАНИЕ**  
В предыдущей главе уже говорилось о том, что Стратегия обеспечивает гибкость, но и ведет к увеличении сложности. Более разумно, вначале "спрятать" логику разбора строки в классе LogParser, который может принимать формат разбираемой строки. Только если такой гибкости окажется недостаточно, то стоит выделять интерфейс ILogParser.

**Шаблонный метод и обеспечение тестируемости**

Типичным подходом для обеспечения тестируемости является использование интерфейсов. Некоторое поведение, завязанное на внешнее окружение, выделяется в отдельный интерфейс и затем интерфейс передается текущему классу. Теперь с помощью объектов-подделок (или моков, от mock) можно эмулировать внешнее окружение и покрыть класс тестами в изоляции.

Вместо выделения интерфейса можно воспользоваться разновидностью паттерна Шаблонный метод под названием "Выделение и переопределение зависимости" (Extract and Override) (\*). Суть техники заключается в выделении "изменчивого поведения" в виртуальный метод, поведение которого затем можно переопределить в тесте.

(\*) СНОСКА: впервые данный прием был описан Майклом Физерсом в его книге "Working Effectively With Legacy Code", а затем в книге Роя Ошерова "The Art of Unit Testing".

Так, класс LogFileReader можно сделать тестируемым путем вынесения открытия файла в виртуальный метод. Затем, в тестах можно будет переопределить этот виртуальный метод и вернуть экземпляр MemoryStream или своего собственного класса, унаследованного от Stream:

public abstract class LogFileReaderBase : LogImporter, IDisposable

{

protected LogFileReaderBase(string fileName)

{

\_stream = new Lazy<Stream>(() => OpenFileStream(fileName)));

}

protected virtual Stream OpenFileStream(string fileName)

{

return new FileStream(fileName, FileMode.Open);

}

}

// В тестах

class FakeLogFileReader : LogFileReaderBase

{

private readonly MemoryStream \_mockStream;

public FakeLogFileReader(MemoryStream mockStream)

: base(string.Empty)

{

\_mockStream = mockStream;

}

protected override Stream OpenFileStream(string fileName)

{

return \_mockStream;

}

}

[Test]

public void TestFakedMemoryStreamProvidedOneElement()

{

// Arrange

LogFileReaderBase cut = new FakeLogFileReader(GetMemoryStreamWithOneElement());

// Act

var logEntries = cut.ReadLogEntry();

// Assert

Assert.That(logEntries.Count(), Is.EqualTo(1));

}

Листинг 2.6 – Использование Шаблонного Метода в тестах

**ПРЕДУПРЕЖДЕНИЕ**  
Желание протестировать некоторый код в изоляции может привести к ощущению ложной безопасности: код протестирован, а значит в нем нет ошибок. В юнит-тестах практически невозможно покрыть все тонкости, которые могут возникнуть при работе с реальными файлами. При работе с внешним окружением, помимо юнит-тестов обязательно должен быть набор интеграционных тестов, который проверит граничные условия в боевых условиях.

### Шаблонный метод и контракты

Шаблонный метод определяет "контракт" между базовым классом и наследниками, поэтому очень важно, чтобы этот контракт был максимально понятным. Подходящие сигнатуры методов и комментарии могут дать понять разработчику класса-наследника, что ему можно делать, а что - нет. Более подходящим способом формализации отношений между классами является использование принципов проектирования по контракту.

С помощью предусловий и постусловий можно более четко показать желаемое поведение и ограничения методов наследников. Сделать это можно с помощью библиотеки Code Contracts:

[ContractClass(typeof (LogImporterContract))]

public abstract class LogImporter

{

protected abstract IEnumerable<string> ReadEntries(ref int position);

protected abstract LogEntry ParseLogEntry(string stringEntry);

}

[ExcludeFromCodeCoverage, ContractClassFor(typeof (LogImporter))]

public abstract class LogImporterContract : LogImporter

{

protected override IEnumerable<string> ReadEntries(ref int position)

{

Contract.Ensures(Contract.Result<IEnumerable<string>>() != null);

Contract.Ensures(

Contract.ValueAtReturn(out position) >= Contract.OldValue(position));

throw new System.NotImplementedException();

}

protected override LogEntry ParseLogEntry(string stringEntry)

{

Contract.Requires(stringEntry != null);

Contract.Ensures(Contract.Result<LogEntry>() != null);

throw new System.NotImplementedException();

}

}

Листинг 2.7 – Пример использования контрактов с Шаблонным Методом

Предусловия и постусловия в библиотеке Code Contracts задаются с помощью методов класса Contract, таких как Requires, Ensures, Assume и других. Поскольку утверждения задаются с помощью методов, то контракты абстрактных методов и интерфейсов задаются в специальном классе, помеченном атрибутом ContractClassFor.

С помощью предусловий и постусловий разработчик базового класса может четко указать, что методы ReadEntries и ParseLogEntry не могут возвращать null. А также, что значение аргумента position не должно уменьшится после вызова метода ReadEntries.

**ПРИМЕЧАНИЕ**  
Полноценное описание принципов контрактного программирования выходит за рамки данной книги. Лучшим источником по этой теме является книга Бертрана Мейера "Объектно-ориентированное конструирование программных систем", в которой контрактное программирование используется в качестве основного инструмента объектно-ориентированного проектирования. С контрактным программированием на платформе .NET можно познакомиться в моих статьях по этой теме по адресу http://bit.ly/DesignByContractArticles, а также в официальной документации библиотеки Code Contracts.

### Non-Virtual Interface Pattern

### Применимость

Практически всегда, когда у вас в голове появляется мысль о повторном использовании кода с помощью наследования, стоит подумать о том, как можно выразить отношения между базовым классом и его наследником максимально четко. При этом нужно помнить о своих клиентах: как о внешних классах, так и о наследниках: насколько просто создать наследника вашего класса, какие методы нужно переопределить, что в них можно делать, а что – нельзя? Когда и от каких классов иерархии наследования нужно наследоваться? Насколько легко добавить еще одного наследника, не вдаваясь в детали реализации базового класса?

Формализация отношений между базовым классом и наследником с помощью контрактов и Шаблонного Метода сделает жизнь разработчиков наследников проще и понятнее. Шаблонный Метод задает каркас, который четко говорит пользователю, что он может сделать и в каком контексте.

## Примеры в .NET Framework

Примеров использования паттерна Шаблонный Метод в .NET Framework очень много. По большому счету, любой абстрактный класс, который содержит защищенный абстрактный метод является примером паттерна Шаблонный Метод.

WCF просто пропитан этим паттерном. Одним из примеров является класс CommunicationObject, методы Open,Close, Abort и т.д. которого "запечатаны" (sealed), но при этом они вызывают виртуальные или абстрактные методы OnClosed, OnAbort и т.д.

Другими примерами этого паттерна в составе WCF могут служить: ChannelBase,ChannelFactoryBase,MessageHeader, ServiceHostBase, BodyWriter, которые определяют каркас алгоритма и позволяют наследникам задавать лишь некоторые шаги.

Другие примеры:

* SafeHandle (и его наследники) с абстрактным методом ReleaseHandle.
* Класс TaskScheduler, с его внутренним QueueTask и открытым TryExecuteTaskInline.
* Класс HashAlgorithm с его HashCore; класс DbCommandBuilder и многие другие.