# Паттерн Посетитель (Visitor)

**Назначение**: описывает операцию, выполняемую с каждым объектом из некоторой иерархии классов. Паттерн Посетитель позволяет определить новую операцию, не изменяя классы этих объектов.

## Мотивация

Объектно-ориентированное программирование предполагает единство данных и операций. Обычно классы представляют некоторые операции, скрывая структуры данных, над которыми эти операции производятся. Но не всегда удобно или возможно смешивать их в одном месте.

Структуры данных некоторых предметных областей могут быть довольно сложными, а операции над ними настолько разнообразными, что совмещать эти два мира вместе нет никакого смысла. Например, в мире финансовых инструментов гораздо логичнее отделить данные ценных бумаг, от выполняемых над ними операций. Деревья выражений - это еще один классический пример, где происходит такое разделение. Но даже в простой задаче, такой как экспорт лог-файлов, можно найти места, где такой подход будет более разумным.

Различные виды прочитанных записей формируют простую иерархию наследования:

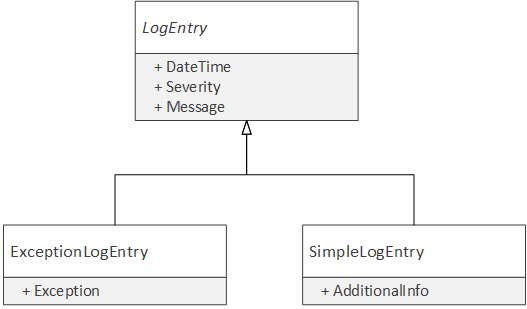


Рисунок 6.1 - Иерархия классов LogEntry

Разные типы записей могут обрабатываться по-разному. Например, сообщения с исключениями могут сохраняться в другую таблицу базы данных, разные типы сообщений могут иметь разную логику группировки для "пакетного" сохранения и т.п. Эту логику нельзя поместить прямо в иерархию сообщений, поскольку логика сохранения может изменяться независимо от самих записей. Также, наличие логики сохранения прямо в классе LogEntry сделает эту иерархию слишком тяжеловесной. Мы не хотим нарушать принципы Открыт-Закрыт, и Принцип единой обязанности, а значит должны вынести подобные аспекты поведения в отдельные классы.

Добавление подобной бизнес-логики вне иерархии чревато дублированием и хрупкостью. Один из вариантов реализации выглядит так:

public class DatabaseLogSaver  
{  
 public void SaveLogEntry(LogEntry logEntry)  
 {  
 var exception = logEntry as ExceptionLogEntry;  
 if (exception != null)  
 {  
 SaveException(exception);  
 }  
 else  
 {  
 var simpleLogEntry = logEntry as SimpleLogEntry;  
 if (simpleLogEntry != null)  
 SaveSimpleLogEntry(simpleLogEntry);  
  
 throw new InvalidOperationException("Unknown log entry type");  
 }  
 }  
  
 private void SaveSimpleLogEntry(SimpleLogEntry logEntry) {...}  
 private void SaveException(ExcetpionLogEntry exceptionLogEntry) {...}  
}

Листинг 6.1 - Простой вариант перебора типов LogEntry

Поскольку иерархия LogEntry является легковесной, то можно ожидать наличие подобного кода в других местах приложения. Решение заключается в выделении логики "посещения" иерархии классов с помощью паттерна Посетитель.

Для этого в базовый класс LogEntry добавляется абстрактный метод Accept, который принимает ILogEntryVisitor, а каждый конкретный класс иерархии просто вызывает метод Visit:

public interface ILogEntryVisitor  
{  
 void Visit(ExceptionLogEntry exceptionLogEntry);  
 void Visit(SimpleLogEntry simpleLogEntry);  
}  
  
public abstract class LogEntry  
{  
 public abstract void Accept(ILogEntryVisitor logEntryVisitor);  
 // Остльные члены остались без изменения  
}  
  
public class ExceptionLogEntry : LogEntry  
{  
 public override void Accept(ILogEntryVisitor logEntryVisitor)  
 {  
 // Благодаря перегрузке методов выбирается метод  
 // Visit(ExceptionLogEntry)  
 logEntryVisitor.Visit(this);  
 }  
}

Листинг 6.2 - Пример интерфейса посетителя

Теперь, если кому-то понадобится добавить операцию над иерархией записей, то достаточно будет реализовать интерфейс ILogEntryVisitor:

public class DatabaseLogSaver : ILogEntryVisitor  
{  
 public void SaveLogEntry(LogEntry logEntry)  
 {  
 logEntry.Accept(this);  
 }  
  
 void ILogEntryVisitor.Visit(ExceptionLogEntry exceptionLogEntry)  
 {  
 SaveException(exceptionLogEntry);  
 }  
  
 void ILogEntryVisitor.Visit(SimpleLogEntry simpleLogEntry)  
 {  
 SaveSimpleLogEntry(simpleLogEntry);  
 }  
  
 private void SaveException(ExceptionLogEntry logEntry) {...}  
 private void SaveSimpleLogEntry(SimpleLogEntry logEntry) {...}  
}

Листинг 6.3 - Пример использования посетителя

**Паттерн Посетитель предназначен для добавления новых операций над иерархией типов, без ее изменения.**

## Классическая диаграмма классов паттерна Посетитель

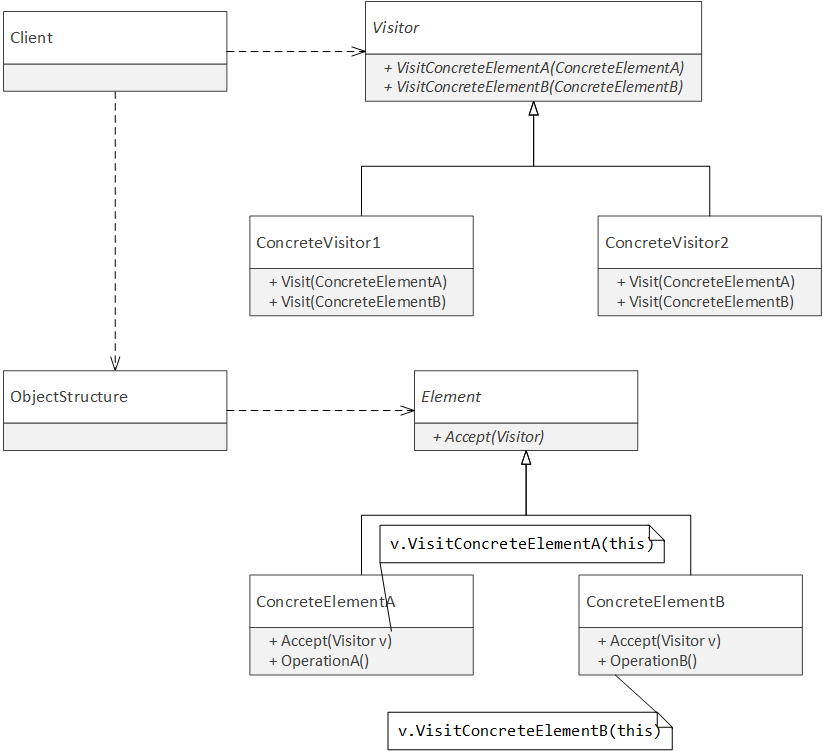


Рисунок 6.2 - Классическая диаграмма классов паттерна Посетитель

* **Участники**  
  Visitor (ILogEntryVisitor) определяет интерфейс посетителя.
* Element (LogEntry) базовый класс иерархии, для которой нужно добавить новую операцию.
* Client (DatabaseLogSaver) использует посетитель для обработки иерархии элементов.

## Обсуждение

Базовый класс иерархии задает семейство операций, поведение которых определяется наследниками. В иерархии фигур, класс Shape может определять набор допустимых операций, а конкретные классы будут реализовывать все операции для конкретного вида фигур. Класс Rectangle будет отвечать за всё поведение прямоугольника: рисование, вычисление площади и т.п. Полиморфное использование базовых классов позволяет сделать систему расширяемой, поскольку добавить новый тип в иерархию наследования довольно просто.

В мире функционального программирования данные и функции отделены друг от друга. При этом одна функция может решать одну задачу над множеством типов. В случае иерархии фигур у нас может одна функция Draw для рисования всех типов фигур, одна функция GetArea для вычисления площади всех типов фигур и т.д.

В объектно-ориентированном решении легко добавлять новый тип иерархии, но сложно добавлять новую операцию. Паттерн Посетитель позволяет решить эту проблему. Посетитель позволяет клиентскому коду "исследовать" иерархию типов и выполнять различные операции, в зависимости от конкретного типа объекта.

При этом паттерн Посетитель усложняет добавление новых типов в иерархию наследования. Добавление нового типа требует изменение интерфейса IVisitor и ломает все его реализации. Это значит, что паттерн Посетитель идеально подходит для расширения функциональности стабильных иерархий наследования с переменным числом операций.

**ПРИМЕЧАНИЕ**  
Подробнее расширяемость и роль паттерна Посетитель рассмотрена в главе "Принцип Открыт-Закрыт".

Классическая реализация паттерна посетитель является достаточно сложной. "Посещаемая" иерархия должна решать дополнительную задачу: реализовывать метод Accept. Сами посетители должны использовать наследование и реализовывать интерфейс IVisitor, что делает поток исполнения не вполне очевидным.

### ФП vs. ООП версии

Когда количество конкретных типов иерархии наследования невелико, интерфейс Посетителя можно заменить списком делегатов. Для этого, метод Accept можно переимовать в Match, который будет принимать несколько делегатов для обработки конкретных типов иерархии.

Метод Match (\*) для классов иерархии LogEntry будет выглядеть так:

(\*) СНОСКА: Разумно добавить еще одну перегруженную версию метода T Match<T>(), которая будет принимать Func<EceptionLogEntry, T> и Func<SimpleLogEntry, T>.

public abstract class LogEntry  
{  
 public void Match(  
 Action<ExceptionLogEntry> exceptionEntryMatch,  
 Action<SimpleLogEntry> simpleEntryMatch)  
 {  
 var exceptionLogEntry = this as ExceptionLogEntry;  
 if (exceptionLogEntry != null)  
 {  
 exceptionEntryMatch(exceptionLogEntry);  
 return;  
 }  
  
 var simpleLogEntry = this as SimpleLogEntry;  
 if (simpleLogEntry != null)  
 {  
 simpleEntryMatch(simpleLogEntry);  
 return;  
 }  
  
 throw new InvalidOperationException("Unknown LogEntry type");  
 }   
}

Листинг 6.4 - "Функциональная" версия паттерна Посетитель

Теперь, вместо создания специализированного класса посетителя для каждого случая, можно просто использовать метод Match прямо в коде анализатора:

public class DatabaseLogSaver  
{  
 public void SaveLogEntry(LogEntry logEntry)  
 {  
 logEntry.Match(  
 ex => SaveException(ex),  
 simple => SaveSimpleLogEntry(simple));  
 }  
  
 private void SaveSimpleLogEntry(SimpleLogEntry logEntry) {...}  
 private void SaveException(ExceptionLogEntry exceptionLogEntry) {...}  
}

Листинг 6.5 - Пример использования "функционального" посетителя

**ПРИМЕЧАНИЕ**  
Может показаться, что реализация метода Match на основе приведения типов грубо нарушает принцип Открыт-Закрыт. Если перебор типов находится лишь в одном месте в коде, то такой код следует принципу Единственного Выбора и не нарушает принцип Открыт-Закрыт. Подробнее этот вопрос будет рассмотрен в главе 2 части 4 "Принцип Открыт-Закрыт".

Данный вид посетителя напоминает стандартную технику функционального программирования (ФП) под названием **сопоставление с образцом** (\*) (pattern matching). Выбор между функциональной и объектно-ориентированной версией паттерна Посетитель такой же, как и выбор между функциональной и объектно-ориентированной Стратегией. В некоторых случаях удобнее создавать именованный класс, реализующий сложную стратегию сохранения или шифрования данных, но когда речь заходит о сравнении или сортировке объектов, то стратегия на основе лямбда-выражений будет более предпочтительной. В данном случае компромисс аналогичен: для большой иерархии типов придется использовать классический посетитель и использовать именованные классы. Для небольших иерархий наследования и посетителей, которые лишь перенаправляют работу другим методам, вариант с лямбда-выражениями будет проще и понятнее, поскольку читателю не нужно переключать «контекст» на реализацию посетителя.

(\*) СНОСКА: Сопоставление с образцом является очень мощной конструкцией функциональных языков программирования и позволяет "сопоставлять" не только типы, но и диапазоны значений, а также "деконструировать" кортежи и записи (Records). В общем случае, сопоставление с образцом можно рассматривать как оператор switch «на стероидах». На данный момент язык C# не поддерживает сопоставление с образцом из коробки. Поэтому разработчику приходится дублировать код или же использовать довольно изощренные решения, наподобие того, что описал Барт де Смет (Bart de Smet) в статье ["Pattern Matching in C# - Part 0"](http://community.bartdesmet.net/blogs/bart/archive/2008/04/06/pattern-matching-in-c-part-0.aspx). Разработчики языка C# рассматривают возможность добавления полноценного сопоставления с образцом в одной из следующих версий языка. Функциональные языки программирования поддерживают сопоставление с образцом из коробки, а также поддерживают возможность создания "вариантов" (алгебраических типов) более удобным способом.

В случае использования самописного сопоставления с образцом, мы можем добавить несколько перегруженных методов Match, которые будут принимать не все возможные типы иерархии, а лишь некоторые наиболее часто используемые. При этом метод Match может находиться в базовом классе иерархии наследования, а может быть реализован в виде метода расширения в классе LogEntryEx.

TODO: предыдущие пару разделов я перенес из главы о принципе Открыт-Закрыт, но решил убрать пример с F#-ом совсем, поскольку он не добавляет никакой пользы!

### Двойная диспетчеризация

Выбор виртуального метода осуществляется на основе типа аргумента во время исполнения. Этот выбор называют одиночной диспетчеризацией. Метод Accept является методом с двойной диспетчеризацией: выбор выполняемого метода определяется типом посещаемого объекта и посетителя.

Паттерн посетитель реализует "двойную диспетчеризацию", поскольку выбор метода определяется на основе типов двух объектов: типе посещаемого объекта и типе посетителя.

LogEntry logEntry = new ExceptionLogEntry();  
ILogEntryVisitor visitor = new Visitor1();  
  
// Вызывается Visitor1.Visit(ExcpetionLogEntry)  
logEntry.Accept(visitor);  
  
logEntry = new SimpleLogEntry();  
visitor = new Visitor2();  
   
// Вызывается Visitor2.Visit(SimpleLogEntry)  
logEntry.Accept(visitor);

Листинг 6.6 - Пример двойной диспетчеризации

### Интерфейс vs. Абстрактный класс посетителя

Обычно Посетитель определяется интерфейсом IVisitor. Такой подход налагает меньше ограничений на клиентов, но делает их более хрупкими. Каждый раз, при добавлении типа в иерархию, интерфейс посетителя обновляется и в нем появляется новый метод Visit(YetAnotherType).

Использование абстрактного базового класса VisitorBase позволяет клиентам "посещать" лишь нужные типы иерархии переопределяя лишь нужные методы Visit. Это также делает клиентов менее хрупкими, поскольку добавление нового виртуального метода Visit не нарушает работу существующих клиентов.

Однако базовый класс налагает более жесткие ограничения на клиентов, поскольку языки платформы .NET не поддерживают множественного наследования классов. Поэтому, обычно эти подходы совмещаются и одновременно используется интерфейс IVisitor с базовым классом VisitorBase.

Проблема же с множественным наследованием обычно решается путем отделения посетителей от основной логики:

public abstract class LogEntryVisitorBase : ILogEntryVisitor  
{  
 public virtual void Visit(ExceptionLogEntry exceptionLogEntry)  
 {}  
  
 public virtual void Visit(SimpleLogEntry simpleLogEntry)  
 {}  
}  
  
public class DatabaseExceptionLogEntrySaver : LogSaverBase  
{  
 public void SaveLogEntry(LogEntry logEntry)  
 {  
 logEntry.Accept(new ExceptionLogEntryVisitor(this));  
 }  
  
 private void SaveException(ExceptionLogEntry exceptionLogEntry) {...}  
  
 private class ExceptionLogEntryVisitor : LogEntryVisitorBase  
 {  
 private readonly DatabaseExceptionLogEntrySaver \_parent;  
  
 public ExceptionLogEntryVisitor(  
 DatabaseExceptionLogEntrySaver parent)  
 {  
 \_parent = parent;  
 }  
  
 // "Посещаем" лишь ExceptionLogEntry  
 public override void Visit(ExceptionLogEntry exceptionLogEntry)  
 {  
 \_parent.SaveException(exceptionLogEntry);  
 }  
 }  
}

Листинг 6.7 - Использование вложенного класса посетителя

В данном случае DatabaseExceptionLogEntrySaver не может сам наследовать от LogEntryVisitorBase, поскольку находится частью другой иерархии типов. Обойти это ограничение можно с помощью внутреннего типа, который будет наследником от LogEntryVisitorBase и переопределит лишь один виртуальный метод для обработки записей с исключениями.

Базовые классы посетителей также полезны в случае древовидных структур, таких как деревья выражений. В этом случае базовый класс может содержать логику навигации по составной структуре данных.

### Посетитель деревьев выражений

TODO: Была мысль дать еще и пример с деревьями выражений в .NET. Но я не уверен, что это поможет в понимании темы читателями. Поэтому пока отложил!

## Применимость

Использовать посетитель нужно лишь тогда, когда появляется необходимость разделить иерархию типов и набор выполняемых операций. Паттерн посетитель позволит легко "разбирать" составную иерархическую структуру и обрабатывать разные типы узлов особым образом.

* Использовать паттерн посетитель нужно тогда, когда набор типов иерархии стабилен, а набор операций - нет.
* Классический вариант паттерна лучше всего подходит для больших составных иерархий, и когда заранее неизвестно какие типы будут посещаться чаще других.
* Функциональный вариант посетителя можно всегда построить на основе классической реализации, когда станет известно, что многим клиентам нужно "посещать" лишь небольшое число типов иерархии. Вариант на основе делегатов в самостоятельном виде подходит лишь для небольшой иерархии типов. По сути, он является простой реализацией сопоставления с образцом и подходит для работы с простыми иерархиями типов, которые моделируют размеченные объединения. Также этот вариант может быть добавлен с помощью методов расширения для существующих иерархий.

## Примеры в .NET Framework

1. [ExpressionTreeVisitor](http://msdn.microsoft.com/en-us/library/system.linq.expressions.expressionvisitor(v=vs.110).aspx) для работы с деревьями выражений ([Expression Trees](http://msdn.microsoft.com/en-us/library/bb397951.aspx)) в .NET Framework. Данный посетитель используется для навигации и преобразования деревьев выражений при реализации специализированных LINQ-провайдеров, а также для решения других задач.
2. [Roslyn](roslyn.codeplex.com) содержит множество посетителей. CSharpSyntaxVisitor для работы с синтаксическим деревом, SymbolVisitor<TResult> для работы с символами, и другие.
3. [DbExpressionVisitor](http://msdn.microsoft.com/en-us/library/system.data.entity.core.common.commandtrees.dbexpressionvisitor(v=vs.113).aspx) используется в Entity Framework для SQL выражений.