# Паттерны поведения. Заключение

Раннее были рассмотрены 6 ключевых паттернов поведения. Каждый из них применяется ежедневно в подавляющем числе проектов и оказывает важное влияние на дизайн приложения.

Без **стратегии** не было бы возможности заменить алгоритм во время исполнения. Без **шаблонного метода** было бы сложно использовать повторно базовые классы. **Посредник** прячет способ взаимодействия более простых объектов и делает их менее зависимыми друг от друга. Без **итераторов** было бы невозможно писать обобщенные алгоритмы, работающие с множеством коллекций. **Наблюдатель** позволяет полностью упразднить явную связь между классом и подписчиками. **Посетитель** делает возможным "функциональный дизайн" и позволяет добавлять новое поведение в иерархии классов без их модификации.

Более детальное рассмотрение лишь ряда паттернов не значит, что другие паттерны поведения не важны, просто они либо реже используются, либо же не оказывают столь сильного влияния на дизайн типичного приложения.

## Паттерн Команда

**Назначение**: Инкапсулирует запрос как объект, позволяя тем самым задавать параметры клиентов для обработки соответствующих запросов, ставить запросы в очередь или протоколировать их, а также поддерживать отмену операций.

Паттерн команда позволяет спрятать действие в объекте и отвязать источник этого действия от места его исполнения. Классический пример - проектирование пользовательского интерфейса. Пункт меню не должен знать, что происходит при его активации пользователем, он должен знать лишь о некотором действии, которое нужно выполнить при нажатии кнопки.

Классическая диаграмма паттерна Команда представляет собой интерфейс ICommand с методом Execute, который может принимать необязательный контекст исполнения (рис. 7.1):

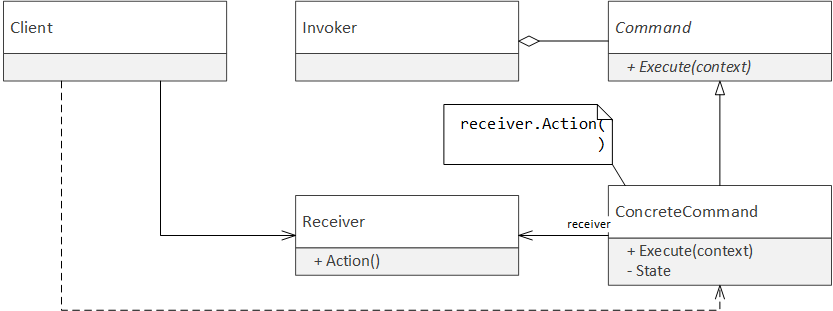


Рисунок 7.1 - Диаграмма классов паттерна Команда

Invoker (пункт меню) получает команду во время создания, которая затем делегирует свою работу объекту класса Receiver (объекту ViewModel).

Существует две основных реализации команды.

* **Самостоятельная команда** может выполнять простую операцию самостоятельно. Например, команда SimplifyBinaryConditionCommand может самостоятельно упростить выражение из двух аргументов.
* **Делегирующая команда** не выполняет операцию самостоятельно, но знает, кто это может сделать. Классическая диаграмма паттерна Команда описывает именно этот вид реализации.

Помимо объектно-ориентированной версии, в языке C# очень распространено использование "функциональной" версии паттерна Команда на основе делегатов. С помощью анонимных методов легко получить делегат, который будет захватывать внешний контекст и выполнять требуемое действие. Подобный вид команд применяется очень часто и мало кто задумывается, что при этом используется один из классических паттернов проектирования:

public class LogExporterViewModel  
{  
 public Action GetExportLogsCommand()  
 {  
 return () => \_logSaver.Save(LogEntries);  
 }  
  
 public IEnumerable<LogEntry> LogEntries {get; private set;}  
}

Листинг 7.1 - Пример паттерна Команда

Популярность команд на основе делегатов привела к появлению особой реализации интерфейса ICommand - RelayCommand, которая принимает в качестве аргумента конструктора Func<T> или Action (\*).

(\*) Сноска: Подробнее о командах в WPF и паттерне MVVM можно почитать в классической статье Джоша Смита (Josh Smith) "Приложения WPF с шаблоном проектирования модель-представление-модель представления", MSDN Magazine, Февраль 2009.

### Примеры в .NET Framework

* [ICommand](http://msdn.microsoft.com/en-us/library/system.windows.input.icommand%28v=vs.110%29.aspx) в WPF, на основе которых строится привязка операций к событиям пользовательского интерфейса.
* [IDbCommand](http://msdn.microsoft.com/en-us/library/system.data.idbcommand%28v=vs.110%29.aspx) в ADO.NET инкапсулирует операцию, исполняемую на стороне СУБД.
* Объект класса Task<T> принимает делегат Func<T>, который можно рассматривать в виде команды, которая будет исполнена в будущем для получения результата задачи.

## Паттерн Состояние

**Назначение**: Позволяет объекту варьировать свое поведение в зависимости от внутреннего состояния. Извне создается впечатление, что изменился класс объекта.

Некоторые объекты предметной области могут выполнять разный набор операций в зависимости от состояния. Объект класса Socket может выполнить операцию Connect только если соединение еще не установлено, а объект потока (класса Thread) может быть запущен только если он не был запущен ранее. Подобные классы моделируют конечный автомат с набором допустимых переходов.

Паттерн состояние предполагает выделение базового класса или интерфейса для всех допустимых операций и наследника для каждого возможного состояния (рис. 7.2):

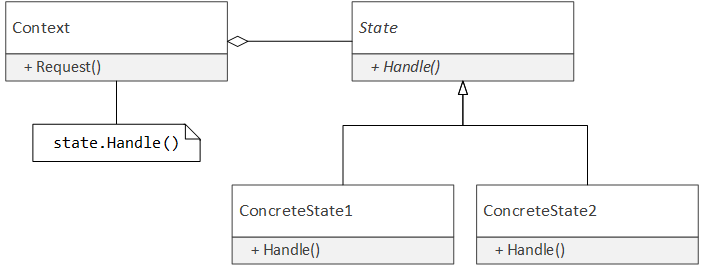


Рисунок 7.2 - Диаграмма классов паттерна Состояние

Контекст (Socket) делегирует операции по переходу между состояниями объектам State, что позволяет перейти из состояния ConcreteState1 (NotConnected) в новое состояние ConcreteState2 (Connected) при вызове метода Handle (метода Connect).

Паттерн Состояние в полноценном виде относительно редко применяется на практике. Его применимость определяется сложностью конечного автомата. Например, для сложной логики анализа состояния тревожной кнопки охранной системы такой подход оправдан, поскольку переход из одного состояния в другое определяется десятком условий. Для более простых случаев, таких как управление соединением к удаленному серверу, обычно достаточно спрятать конечный автомат в отдельном классе, но не выделять отдельный класс на каждое состояние. В еще более простых случаях, таких как управление состоянием класса Thread, разумнее всего поместить логику конечного автомата прямо в класс Thread и не выносить ее отдельно.

### Примеры использования

В .NET Framework конечные автоматы применяются в огромном числе классов, но классической реализации паттерна Стратегия среди открытых (public) типов .NET Framework я не нашел.

* CommunicationObject реализует конечный автомат перехода между состояниями WCF клиента: Created, Opening, Opened, Closing, Closed и Faulted.
* Task реализует конечный автомат перехода между состояниями "задачи": Created, WaitingForActivation, WaitingToRun, Running, RunToCompletion, Canceled, Faulted.

## Паттерн Цепочка обязанностей

**Назначение**: Позволяет избежать привязки отправителя запроса к его получателю, давая шанс обработать запрос нескольким объектам. Связывает объекты-получатели в цепочку и передает запрос вдоль этой цепочки, пока его не обработают.

Цепочка обязанностей является довольно распространенным паттерном в .NET Framework, хотя не все знают, что пользуются им настолько часто. Цепочка обязанностей - это любое событие, аргументы которого позволяют уведомить инициатора, что событие обработано с помощью метода Handle() или путем установки свойства Handled в True.

В случае закрытия формы в Windows Forms, генерируется событие Closing, подписчики которого могут отложить закрытие окна в случае необходимости:

public DialogForm()  
{  
 this.Closing += HandleClosing;  
}  
  
private void HandleClosing(object sender, CancelEventArgs ea)  
{  
 if (UserDontWantToCloseTheWindow())  
 {  
 ea.Cancel = true;  
 }  
}

Листинг 7.2 - Пример паттерна Цепочка обязанностей

Поскольку делегаты в .NET могут содержать более одного обработчика, то на их основе цепочка обязанностей строится простым и элегантным образом. Для этого событие возбуждается не с помощью метода Invoke, а вручную для каждого подписчика:

// Form.cs  
private void ClosingForm()  
{  
 if (Closing != null)  
 {  
 var eventArgs = new CancelEventArgs(cancel: false);  
 var invocationList = Closing.GetInvocationList();  
 foreach (EventHandler<CancelEventArgs> handler in invocationList)  
 {  
 handler(this, eventArgs);  
 if (eventArgs.Cancel)  
 {  
 // Отменяем закрытие формы  
 return;  
 }  
 }  
 }  
   
 CloseForm();  
}

Листинг 7.3 - Псевдокод реализации цепочки обязанностей

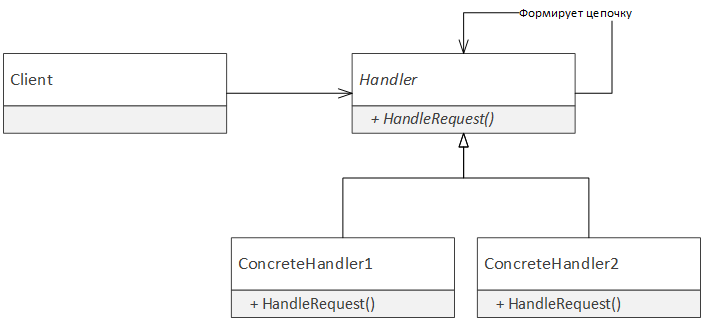


Рисунок 7.3 - Диаграмма классов паттерна Цепочка обязанностей

### Примеры в .NET Framework

* Событие Closing в Windows Forms с возможностью отменить закрытие формы путем установки свойства Cancel аргумента CancelEventArgs в True.
* Событие TaskScheduler.UnobservedException, обработчик которого может уведомить планировщик, что необработанное исключение задачи не является критичным путем вызова метода UnobservedTaskExceptionEventArgs.SetObserved. Для аналогичных целей используются события Contract.ContractFailed и Application.DispatcherUnhandledException.