PostSharp. OnMethodBoundary Aspect

Прочитать статью на сайте

Продолжаем подробный разговор о аспекте OnMethodBoundary. Данный аспект применяется, когда у вас есть доступ к исходным кодам метода и вы хотите отслеживать события начала вызова метода, конец вызова, исключение и успешность завершения. Все это делается переопределением методов *OnEntry*, *OnExit*, *OnException* и *OnSuccess* соответственно.

Применение аспекта OnMethodBoundary рассмотрим на примере задач о разрешении или запрещении тех или иных действий, логирование интересующих нас действий, генерация служебного кода. Так же с помощью данного аспекта можно декларативно указывать запускать ли методы в отдельном потоке или нет, но это рассмотрим подробно и в деталях в другой статье. Сегодня на повестке дня логирование – это будет показано в одном приложении, исходный код которого можно свободно скачать и поиграться с ним.

Logging

Полезная функция, которая в состоянии подсказать где происходит ошибка и с каким набором данных, подсказать какие функции наиболее востребованы пользователями. Да мало ли еще какое применение можно найти для собранной детальной информации! =)

Будем основываться на приложении из прошлой статьи. У нас готов интерфейс для отображения происходящего в системе. Так что можно начинать.

Сам класс, который занимается логированием прост до безобразия. Собственно он и должен-то уметь только хранить записи и отдавать их по требованию. У меня этот класс получился таким:

```
public class Log {
        private static string offset = "";
        private static readonly List<string> loggedEvents;

        public static Action Update { get; set; }

        public static ReadOnlyCollection<string> LoggedEvents {
            get { return loggedEvents.AsReadOnly(); }
        }

        static Log() {
            loggedEvents = new List<string>();
            Update = () => { };
        }

        public static void Indent() {
            offset = offset + "    ";
        }

        public static void Unindent() {
            offset =  offset.Length < 4 ? "" : offset.Substring(4);
        }

        public static void RecEvent(string desc) {
            loggedEvents.Add(offset + desc);
            Update();
        }
```

```
        public static void Clear() {
            loggedEvents.Clear();
            Update();
        }
    }
```

> В .Net есть специальный класс Trace, с более богатыми возможностями, который и следует использовать в реальных приложениях. Данный пример упрощен специально для отображения принципа работы и простоты.

Класс логирования обозначен как статический, чтобы доступ к нему был из любого места программы. Ведь не передавать же его как параметр в аспект, придется его таскать через все приложение, чего делать, конечно же, не надо.

Без применения аспектов пришлось бы в каждый метод, прописывать кучу кода, который только бы создавал визуальный шум. Можно было бы определенными способами минимизировать код, который будет писаться в каждой функции для логирования, но все равно это оставалось бы ручной работой, на которую легко забить, потерять. На мой взгляд гораздо проще написать один атрибут перед классом, или вообще применить его ко всей сборке написав в **AssemblyInfo.cs**:

```
[assembly: Logging(AttributeTargetTypes = "WBR.VioletTape.*")]
```

С помощью этой команды мы указали, что аспект **Logging** применять ко всем классам в сборке, в пространстве имен **WBR.VioletTape**.

Как вы уже догадались, аспект для логирования будет называться Logging. Для успешного отслеживания всех событий будем переопределять события входа в метод, успешного его завершения и если случилась ошибка. Таким образом, каркас будет таким:

```
    [Serializable]
    public class Logging : OnMethodBoundaryAspect {
        public override void OnEntry(MethodExecutionEventArgs eventArgs) {
            base.OnEntry(eventArgs);
        }

        public override void OnSuccess(MethodExecutionEventArgs eventArgs) {
            base.OnSuccess(eventArgs);
        }

       public override void OnException(MethodExecutionEventArgs eventArgs) {
        }
    }
```

Из eventArgs можно получить всю интересующую информацию о методе который будет выполнятся: аргументы метода, его тело, доступ к классу, определить атрибуты метода, возвращаемое значение и много чего еще. Я думаю для начала будет достаточно логировать время начала/конца операции и имя метода. По желанию можно записывать и все аргументы, если нужно скурпулезное логирование.

```
public override void OnEntry(MethodExecutionEventArgs eventArgs) {
            var line = string.Format("{0}.{1}: Enter",
                                     DateTime.Now.ToLocalTime(),
                                     eventArgs.Method.Name);

            Log.RecEvent(line);
            Log.Indent();
            base.OnEntry(eventArgs);
```

```
        }
```

По аналогии пишем код для остальных методов.

Можно данный аспект применять к любым объектам и получать всю-всю-всю информацию. Но через некоторое время вы можете заметить, что информации слишком много. Да-да, к примеру, будут логироваться все изменения свойств класса, так как для них формируются методы **get_<PropertyName>** и **set_<PropertyName>**. Это можно обойти в теле метода, применив проверку на специальность имени

```
if (!eventArgs.Method.IsSpecialName) { … }
```

Наследование атрибутов

По умолчанию аспекты не наследуются. Т.е. если у нас есть такой код:

```
[Logging]
    public class CalculationService {
        public decimal HistoryCompute() {
            new CalculationExtendedHistoryService().ComputeAll();
            return 30;
        }
    }

    public class CalculationExtendedHistoryService : CalculationService {
        public decimal ComputeAll() {
            return 50;
        }
    }
```

То логирование в дочернем классе не произойдет. Данную проблему можно решить механически и в лоб – навесить логирование на дочерний класс. Сработает! Однако есть и другие способы локально или глобально указать правила использования.

Локально:

```
[Logging(AttributeInheritance = MulticastInheritance.Multicast)]
    public class CalculationService {
        public decimal HistoryCompute() {
            new CalculationExtendedHistoryService().ComputeAll();
            return 30;
        }
    }

    public class CalculationExtendedHistoryService : CalculationService {
        public decimal ComputeAll() {
            return 50;
        }
    }
```

Или *глобально:*

```
[MulticastAttributeUsage(MulticastTargets.Method |
MulticastTargets.InstanceConstructor,
            Inheritance = MulticastInheritance.Multicast)     ]
    public class Logging : OnMethodBoundaryAspect { … }
```

Думаю, что вы достаточно быстро столкнетесь с проблемой, когда аспекты налагаются друг на друга. Когда вы объявили множественное наследование аспекта глобально и еще раз применили аспект на дочерний класс в ручную, то получится, что один и тот же метод залогируется много раз, так как аспекты наложатся друг на друга. Чтобы это не произошло, надо в глобальном объявлении добавить строчку с явным запретом применения множества аспектов на один и тот же экземпляр класса.

```
[MulticastAttributeUsage(MulticastTargets.Method |
MulticastTargets.InstanceConstructor,
        AllowMultiple = false,
            Inheritance = MulticastInheritance.Multicast)
    ]
    public class Logging : OnMethodBoundaryAspect { … }
```

Теперь не будет дублирования строчек в логе!

Можно настраивать различную степень детализации для логов, путем передачи параметров в аспект. Делается это как и с любым классом, заводим свойство в аспекте.

```
public class Logging : OnMethodBoundaryAspect {

        public bool FullLog{ get; set; }

        public override void OnEntry(MethodExecutionEventArgs eventArgs) {… }

// and so on

}
```

Можно анализировать что выставил пользователь и в зависимости от этого записывать в лог больше данных. В самом коде установка значения выглядит так:

```
[Logging(FullLog = true)]
    public class CalculationService { … }
```

Всегда помните, что для каждого экземпляра класса создается свой экземпляр аспекта. Чтобы сделать поля общими, применяйте общие для этого практики – статичные поля =)

Исходные коды как всегда доступны в полном объеме. Не забудьте только скачать и установить PostSharp. «Happy Postsharping!» – как говорит основатель проекта Gael Fraiteur, и…

Hard’n’heavy!