PostSharp. OnMethodBoundary Aspect

Прочитать статью на сайте

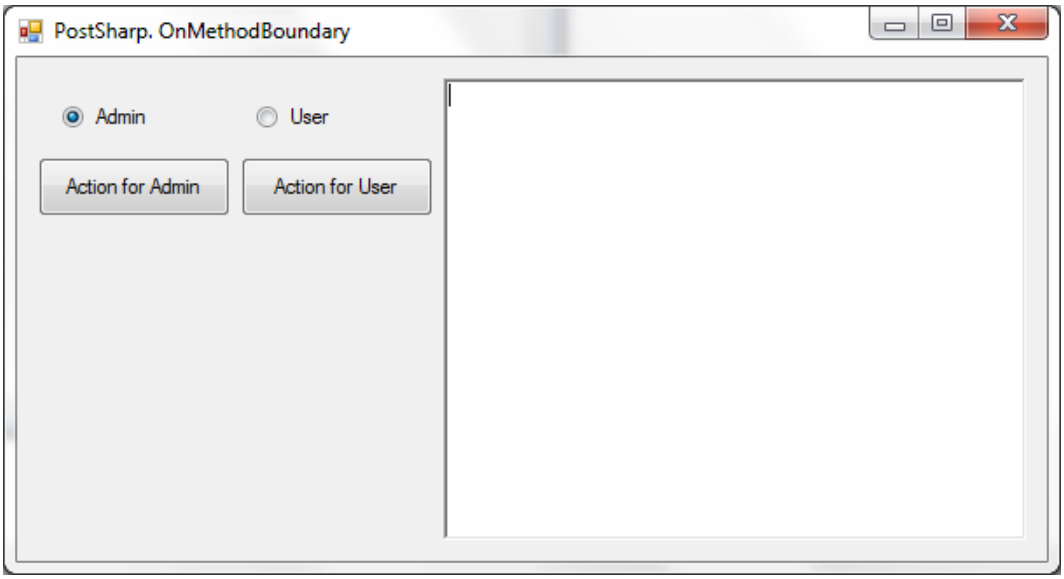
Сегодня я хочу остановиться поподробнее на аспекте OnMethodBoundary. Данный аспект применяется, когда у вас есть доступ к исходным кодам метода и вы хотите отслеживать события начала вызова метода, конец вызова, исключение и успешность завершения. Все это делается переопределением методов *OnEntry*, *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 безопасность, логирование и генерация кода – все это будет показано в одном приложении, исходный код которого можно свободно скачать и поиграться с ним.

Code Generation

На мой взгляд, больше всего «бесполезного» кода появляется на пользовательских формах и представлениях (UserControl) для WinForms. Необходимо подписывать почти каждый элемент на событие щелчка, изменения состояния элемента. Это куча кода, которую можно не писать совсем в большинстве случаев, если принять некоторые соглашения и строить весь интерфейс на биндинге. В конечном итоге такая практика поможет лучше понять идеологию WPF, если вы еще не перешли на нее. Да и поддерживать и тестировать такое подход, на мой взгляд, легче.

Итак, допустим у нас есть такая вот форма:



Все стандартные элементы, и если бы мы подписывались на действия стандартно, то в коде было бы что-то в духе:

```
private void salaryFor_Click(object sender, System.EventArgs e) {
    //some actions
}

private void salaryMy_Click(object sender, System.EventArgs e) {
    //some actions
```

```
        }
```

Это код для обработки нажатия кнопок, еще как-то надо обрабатывать информацию с радио-кнопок – геморрой еще тот. Да и к тому же, сложный код с логикой в кнопке вы писать не будете, а вызовите функцию из слоя презентации, которая и обработает все что нужно. Т.е. написание такого кода еще более неблагодарное занятие.

Ко всему прочему все эти вызовы только создают визуальный шум.

Создание инфо моделей

Предлагаю делать это следующим образом:

- Создать информационную модель для каждой формы или представления;
- Использовать привязывание (binding) для отображения данных;
- Использовать привязывание (binding) для возбуждения событий на действия пользователя.

При создании информационных моделей (далее просто «модели») важно четко определить их функциональность. В большинстве случаев должно получаться один экран – одна модель, в этом случае у вас будут экраны с простой функциональностью, понятной для пользователей. Если же вы понимаете что на одном экране присутствуют несколько моделей данных, то ничего страшного не случилось, главное в порыве страсти вы их не объединили в одну.

В нашем варианте, несмотря на кажущуюся простоту экрана, будут присутствовать 2 модели:

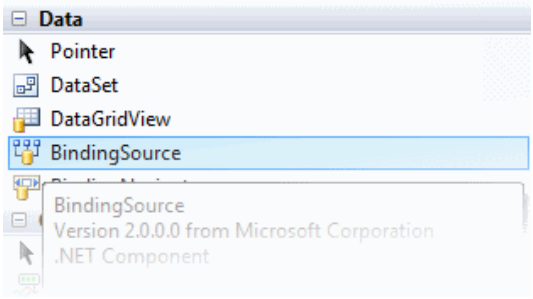
- Обработка реакции на нажатие кнопок пользователем (*SecurityModel*);
- Вывод логов в редактор (*LogModel*).

Все действия, которые будут вызываться с представления или формы должны быть объявлены как экземпляры класса **Action.** *Данные*, которые мы хотим отобразить должны быть объявлены публичными **свойствами** и обязательно с публичными *get* и *set*. Это нужно для того, чтобы корректно сгенерировался источник данных. Для примера SecurityModel может выглядеть так:

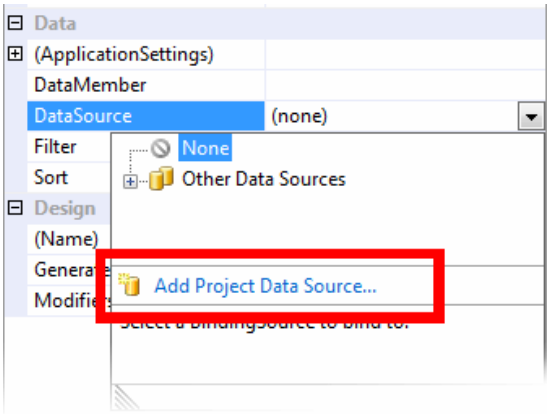
```
public class SecurityModel {
        public bool IsAdmin {
            get { return AppContext.Role == SecurityRole.Administrator; }
            set { AppContext.Role = SecurityRole.Administrator; }
        }

        public bool IsUser {
            get { return AppContext.Role == SecurityRole.User; }
            set { AppContext.Role = SecurityRole.User; }
        }

        public Action SecureActionForAdmin { get; set; }
        public Action SecureActionForUser { get; set; }
    }
```

> Перед тем как сформировать источник данных (DataSource) на основе класса, необходимо пересобрать проект. Без этого вы не увидите вашего класса в списке возможных кандидатов.

После того как модель готова и все собрано, добавляем на форму новый компонент BindingSource.



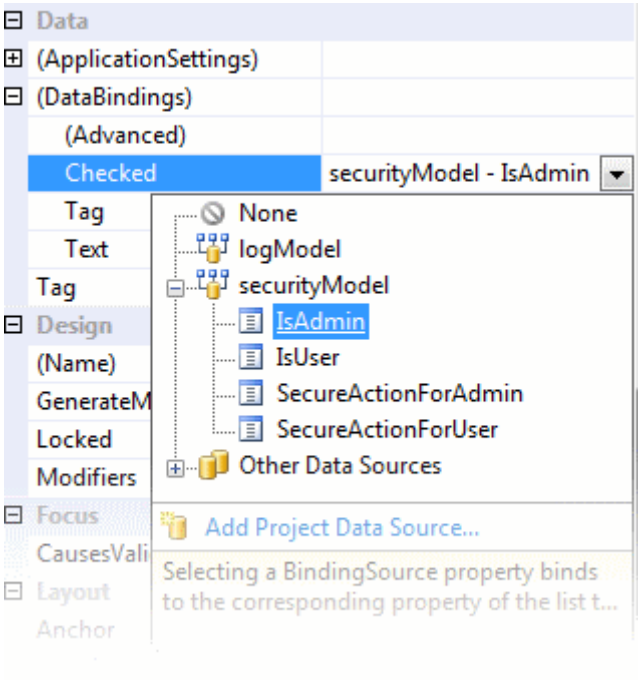
После этого создаем новый DataSource, который будем использовать для привязки данных.



Жмем на стрелочку чтобы появилось выпадающее меню как на картинке выше, и жмем на «Add Project Data Source…». После этого появляется новое окно в котором необходимо выбрать «Object» как тип источника данных и подтвердить свой выбор, на следующем экране найти класс, который будет источником данных. После того как все сделано, можно привязывать свойства и события модели к элементам интерфейса. Да, чуть не забыл: не забывайте переименовывать компоненты чтобы у них были более мнемоничные имена. ;)

Привязка моделей к интерфейсу

Выбираем радио-кнопку с подписью **Admin**, жмем F4 (Свойства), находим секцию **Data**, в ней свойства **(Data Bindings)** и назначаем привязку к свойству из модели и компонента.



В указанной секции обычно помимо свойств *Tag* и *Text* указаны свойства наиболее важные для данного компонента. Для радио-кнопки это **Checked**, что мы собственно и сделали. Такое же действо проделываем и с радио-кнопкой **User**.

После этого, при переключении радио-кнопок, автоматически будет вызываться методы get и set для указанного свойства из модели. А глядя на модель, можно сказать, что изменения сразу будут вступать в силу: AppContext.Role будет менять свое значение.

Для кнопок все будет точно так же, события назначаем на свойство **Tag**.

Пока никакой магии нет. Но уже скоро будет. Для того, чтобы по нажатию кнопки сработало событие которое нам интересно, необходимо в обработчике нажатия написать следующее:

```
private void OnClick(object sender, EventArgs e) {
            var tag = ((Control) sender).Tag;
            if(tag == null) return;

            ((Action) tag)();
        }
```

Уже по этому коду видно, что все одинаково для любой кнопки. Так почему бы не сделать так, чтобы это генерировалось автоматически?

В этом деле нам поможет аспект OnMethodBoundary.

Создание аспекта

Изнутри аспекта можно получить доступ к экземпляру класса, к которому он применен. Благодаря этому, мы можем перебрать все дочерние компоненты формы или представления, найти интересующие нас классы, организовать подписку на нужные нам события и там же их обрабатывать. Итак, действия по пунктам:

1. Дожаться завершения работы конструктора
2. Найти все кнопки и радио-кнопки
3. Подписаться на интересующие события

Начинаем писать аспект.

Создаем класс *EventActivation*, наследованный от *OnMethodBoundaryAspect*. **Класс** необходимо пометить как **сериализуеммый!** После создания класса, надо перегрузить метод **OnSuccess**.

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

    }
```

Первым делом нам надо дождаться завершения работы конструктора. Это можно сделать, проверив свойства метода через параметр **MethodExecutionEventArgs**.

```
if (eventArgs.Method.IsConstructor) {  }
```

Следующим шагом надо будет найти все важные нам элементы интерфейса. Это делается с помощью рекурсивной функции, так как интерфейс может быть очень сложным и кнопки могут находиться и на панелях дополнительных и на закладках. Вобщем без рекурсии никуда.

```
public void Subscribe(Control root) {
            if (root.Controls.Count == 0) return;
            foreach (var component in root.Controls) {
                if (component as Button != null) {
                    ((Button) component).Click += OnClick;
                    continue;
                }
                if (component as RadioButton != null) {
                    ((RadioButton) component).CheckedChanged += OnClick;
                    continue;
                }
                Subscribe((Control) component);
            }
        }
```

Для начала нам вполне подойдет такая функция. Тут фигурирует еще один метод, который не рассмотрен – **OnClick**. А этот метод фактически переноситься с представления в тело класса аспекта, только добавляется модификатор *static*.

```
private static void OnClick(object sender, EventArgs e) {
            var tag = ((Control) sender).Tag;
            if(tag == null) return;

            ((Action) tag)();
        }
```

Все, наш аспект готов. В конечном варианте метод **OnSuccess** выглядит следующим образом:

```
public override void OnSuccess(MethodExecutionEventArgs eventArgs) {
            if (eventArgs.Method.IsConstructor) {
                var form = eventArgs.Instance as Form;
                Subscribe(form);
            }
```

```
            base.OnSuccess(eventArgs);
        }
```

Теперь можно его применить на форму или представление, и убрать все ненужные подписки. Открываем код формы или представления, и приписываем новый атрибут к классу. Все события можно поудалять. Таким образом класс будет выглядеть очень лаконично и красиво!

```
[EventActivation]
    public partial class MainForm : Form {
        public MainForm() {
            InitializeComponent();
        }
}
```

Все сборки, в которых вы используете аспекты, должны иметь ссылки (reference) на **PostSharp.Laos.dll** и **PostSharp.Public.dll**.

Улучшения аспекта

Данный аспект имеет потенциал для улучшения.

Первым делом, можно задать ограничения, что данный аспект может быть применим только на конструктор, что избавит нас от условного оператора в методе *OnSuccess*. Делается это применением атрибутов сборки к самому аспекту. В нашем случаем это будет выглядеть так:

```
[AttributeUsage(AttributeTargets.Constructor)]

    public class EventActivation : OnMethodBoundaryAspect { <your code here>
}
```

Атрибут **AttributeUsage** указывает какие ограничения накладываются на применение аспекта. Пока нас интересует только первая строчка (первый параметр), где говориться что целью данного аспекта могут выступать только конструкторы, и применение его к методу вызовет ошибку компиляции.

После этого переместим применение аспекта на объявление конструктора.

```
    public partial class MainForm : Form {
      [EventActivation]
        public MainForm() {
            InitializeComponent();
        }
}
```

Так же еще можно создавать параметры к аспекту, где указывать к каким типам элементов интерфейса применять подписку, а какие пропускать.

Задачи безопасности и логирования рассмотрим в следующих статьях. Их план готов, материал есть, так что они не заставят себя долго ждать.

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!