Автообнаружение и регистрация классов в StructureMap

Некоторое время назад я рассказывал про чудесный IoC\DI контейнер StructureMap. На тот момент я описал основные приемы работы, скорее как справочник, показавший возможности, без углубления в конкретные темы. На этот раз я хочу рассказать о возможности автоматического обнаружения и регистрации классов средствами StructureMap.

Уже по самому названию понятно, что это хорошо и несет благодать, так как обещает сократить трудовые расходы и написание тривиального кода, что всегда скучно и всегда забываешь его дописать где-то. Авторегистрация решит все эти проблемы, хотя, конечно, сначала придется попыхтеть.

Дальнейшие примеры покажут, как работать с обнаружением только по интерфейсу\базовому классу, а так же как работать при именовании классов по соглашениям (Convention over Contract).

Вводная

Сначала достаточно простой пример. Пусть у нас есть множество классов для трансляции данных из базы, все они наследованы от базового класса. Т.е. что-то в духе:

```
public class ClientAdapter : ReadOnlyAdapter<Client, vClient> {
    ...
}

public class InvoiceAdapter : WriteAdapter<Invoice, vInvoice> {
    ...
}
```

И таких классов достаточно много, всех их регистрировать вручную не очень хочется, т.к. это выглядит ужасно в коде и писать долго. Регистрация таких адаптеров может занять экран, а то и два. Гораздо лучше поднапрячь извилины и записать это в несколько строк.

Итак, StructureMap предоставляет метод Scan, который пробегает по интересующим нас сборкам или папкам и регистрирует подходящие объекты. Для того чтобы метод Scan нашел и зарегистрировал типы необходимо соблюдение нескольких условий:

- Тип должен быть явным, дженерик типы не регистрируются автоматически;
- Тип должен иметь публичный конструктор;
- Конструктор не может иметь аргументов примитивных типов;
- Множественное регистрирование не допускается.

Указание сборки для сканирования можно задать несколькими способами:

- Явно прописать имя сборки или же передать ее саму;
- Обратиться к вызывающей сборке;
- Найти сборку содержащую определенный тип;
- Найти сборки по определенному пути.

Сопоставление один к одному

В нашем случае мы переопределим способ сканирования и метод регистрации, так как получение адаптера должно происходить по доменному типу, который в дженерик параметре базового класса. Для это создадим отдельный класс, который будет реализовывать интерфейс **IRegistrationConvention**, что требуется для указания класса как отдельного регистратора\сканера.

Пусть этот класс называется **AutoRegistryAdapterScanner**, тогда для конфигурирования StructureMap надо написать:

```
var adapterScanner = new AutoRegistryAdapterScanner();
ObjectFactory.Configure(x => x.Scan(s => {
                                        s.Assembly("Infrastructure");
                                        s.With(adapterScanner);
                                    }));
```

Т.е. создали класс сканера, далее указываем SM что надо сканировать сборку инфраструктуры, с использованием нашего сканера. В данном случае имя сборки можно указать текстом, мне показалось это проще всего.

Конечно же, основная магия будет происходить в классе **AutoRegistryAdapterScanner.** Окидывая сейчас взором реализацию класса, обнаруживаю, что никаких сложностей в нем нет, только аккуратное вычленение нужных классов из тех, что реализованы в модуле инфраструктуры. Но тем не менее.

Класс AutoRegistryAdapterScanner должен реализовывать интерфейс IRegistrationConvention, в котором определен только один метод Process, который принимает тип и объект Registry, с помощью которого происходит регистрация классов. При сканировании StructureMap находит все объекты в указанной сборке и передает их в метод Process. Мы же должны определить, будем регистрировать переданный класс или нет. Звучит достаточно просто.

Основной алгоритм такой:

- Если пришел дженерик тип, то переходим к следующему типу.
- Определяем все интерфейсы пришедшего типа.
- Если интерфейсы есть, то находим дженерик интерфейс имя которого удовлетворяет общему имени интерфейса. В нашем случае IReadAdapter<>.
- Если такой находится, то регистрируем пришедший тип с найденным интерфейсом.

```
internal class AutoRegistryAdapterScanner : IRegistrationConvention {
    public void Process(Type type, Registry registry) {
        if (type.IsGenericType) return;

        var interfaces = type.GetInterfaces();
        if (!interfaces.Any()) return;

        var directInterface = interfaces
            .FirstOrDefault(i => i.IsGenericType
                 && i.GetGenericTypeDefinition().Name == typeof (IReadAdapter<>).Name);

        if (directInterface.IsNotNull())
            registry.AddType(directInterface, type);
    }
}
```

То же самое можно проделать для обнаружения адаптеров, которые отвечают и за запись.

После того, как мы зарегистрировали адаптеры, возникает вопрос, а как же их использовать, как получить в процессе работы? Это оказалось тоже не сложно. В классе InfrastructureFacade, который отвечает за сохранение и получение элементов из базы, есть основные методы:

```
public interface IInfrastructureFacade {
    List<T> Get<T>(Specification<T> specification);
    void Save<T>(IEnumerable<T> items);
    void Delete<T>(IEnumerable<T> items);
}
```

При использовании метода Get в фасад приходит спецификация по типу которой можно определить тип данных, с которым будем работать, а соответственно и тип подходящего адаптера. Сам метод Get не очень интересен:

```
public List<T> Get<T>(Specification<T> specification) {
       var adapter = GetAdapter<T>();
       var list = adapter.Get(specification);
       return list.FindAll(specification);
}
```

Гораздо интереснее узнать, как же получается адаптер:

```
private IReadAdapter<T> GetAdapter<T>() {
   var instance = ObjectFactory.Container
              .ForGenericType(typeof (IReadAdapter<>))
              .WithParameters(typeof (T))
              .GetInstanceAs<IReadAdapter<T>>();

   if (instance.IsNull()) {
       var type = typeof (T);
       throw new ArgumentOutOfRangeException("", type,
                          string.Format("Type {0} not registered/founded", type.Name));
       }

       return instance;
}
```

Благодаря fluent интерфейсу StructureMap можно прочитать чистый английский текст, о том как получается адаптер. Несколько непривычные конструкции SM для повседневного использования, но тем не менее. Самопальные сервис-локаторы такой функциональностью не будут обладать долго.

На мой взгляд, кода получилось немного и, что самое главное, его можно использовать с минимальной модификацией для других программ или частей системы, так как существует множество приложений, когда надо регистрировать пачки сервисов или классов по их функционалу и принадлежности к обрабатываемым классам.

Усложнение задачи

В текущем проекте, как и во многих предыдущих, для работы с интерфейсом пользователя я использую только одно физическое окно, меняя в нем пользовательские экраны в зависимости от задачи. При таком подходе идет жесткое сопоставление модели и интерфейса (компонента), которое тоже не хочется описывать руками, так как в большинстве случаев модель и интерфейс надо явно сопоставить, чтобы в дальнейшем загружать только указав интерфейс.

На данный момент не буду вдаваться в детали реализации ViewManager'а, который управляет интерфейсом пользователя, скажу только интерфейс который он реализует.

```
public interface IViewManager {
    void Show<T>();
    IAutoRegistryView CurrentView { get; }
}
```

Очень лаконично, но при этом класс его реализующий должен по интерфейсу получить как компонент UI , так и модель. Для реализации данного функционала на помощь придет возможность именованной регистрации конкретных классов (плагинов), т.е. при регистрации будем просто указывать что это: модель или UI-компонент.

Теперь надо немного описать то, как описывается модель и UI-компонент. Для примера возьму модель логина пользователя, чтобы не придумывать имен моделям.

```
[ImplementLogicFor(typeof (ILoginView))]
public class LoginViewModel : BaseModel, ILoginView {  ...  }

public class ImplementLogicForAttribute : Attribute {
    public Type Type { get; private set; }

    public ImplementLogicForAttribute(Type type) {
        Type = type;
    }
}
```

Интерфейс пользователя объявлен как:

```
public partial class LoginView : UserControl, ILoginView {  ...  }

public interface ILoginView : IAutoRegistryView { }
```

Наследование модели и интерфейса пользователя от одного и того же интерфейса нужно для того, чтобы их можно было определить как один тип плагина в терминах SM. Определение атрибута для модели значительно упрощает задачу по ее поиску и определению в нужное семейство плагинов в StructureMap.

При определении прямой реализации интерфейса компонентом пользовательского интерфейса, необходимо уже прибегать к разбору и сопоставлению имен. Подробнее расскажу чуть позже.

Итак, сначала будем находить модели, по атрибуту. Для этого точно так же создадим класс AutoRegistryModelTypeScanner, который будет наследован от IRegistrationConvention. Так как класс Type богат на вспомогательные методы для нашего дела, то надо лишь запросить атрибуты требуемого типа, и если они есть, то зарегистрировать пришедший тип для того плагина, который указан в атрибуте. При рассмотрении реализации оказывается, что это мизерный объем кода!

```
internal class AutoRegistryModelTypeScanner : IRegistrationConvention {
    public void Process(Type type, Registry registry) {
        var customAttributes = type.GetCustomAttributes(typeof
(ImplementLogicForAttribute), false);

        if (!customAttributes.Any()) return;

        var implementLogicFor = (ImplementLogicForAttribute) customAttributes[0];
        registry.AddType(implementLogicFor.Type, type, ScannerMode.Model);
    }
}
```

При этом регистрируем тип по перегруженному методу, который принимает имя для зарегистрированного типа – **ScannerMode.Model**. Сам по себе класс со статическими полями строкового типа.

Чуть сложнее будет работа с компонентами пользовательского интерфейса. Назначать им атрибуты отчего-то не захотелось, хотя тоже можно было. Решено было определять по именам, так как имя компонента входит в имя интерфейса.

Сначала надо определить пришедший тип на принадлежность к IAutoRegistryView, далее вычленить из всех интерфейсов по имени самый близкий.

```
internal class AutoRegistryViewTypeScanner : IRegistrationConvention {
    public void Process(Type type, Registry registry) {
        var relatedInterfaces = type.FindInterfaces(AutoRegistryViewFilter, typeof
(IAutoRegistryView));
        if (!relatedInterfaces.Any()) return;

        var interface = type.FindInterfaces(DirectInterfaceFilter, type).First();
        registry.AddType(interface, type, ScannerMode.View);
    }

    internal static bool AutoRegistryViewFilter(Type typeObj, Object criteriaObj) {
        return typeObj.ToString() == criteriaObj.ToString();
    }

    internal static bool DirectInterfaceFilter(Type typeObj, Object criteriaObj) {
        return typeObj.Name.Contains(((Type) criteriaObj).Name);
    }
}
```

Не вижу тут каких-либо подводных камней либо областей для более подробного рассказа.

В итоге у нас получается, что по одному интерфейсу зарегистрировано два класса. Обращение к ним будет таким:

```
ObjectFactory.GetNamedInstance(type, ScannerMode.View);
ObjectFactory.GetNamedInstance(type, ScannerMode.Model);
```

Заключение

Для маленькой программы использование таких подходов наверно не очень целесообразно, однако когда у вас с десяток экранов, сервисов, и прочих сущностей которые отбираются по типу рабочего класса – это может серьезно сэкономить время, силы, нервы.

До того, как я стал использовать StructureMap с авторегистрацией классов, я частенько забывал о регистрации классов в IoC\DI контейнере, а если вспоминал, то забывал в каком месте надо регистрировать все это хозяйство – очень утомляло каждый раз. Зато сейчас я очень доволен такому подходу, так как нервы в порядке, а какой-либо ощутимой разницы по времени при запуске с авторегистрацией не заметил. Так что желаю вам здоровой лени, и лучше работать головой, чем руками.

Hard’n’heavy!

Violet Tape