StructureMap

Сегодня я хочу рассказать о IoC контейнере StructureMap, который мне приглянулся гораздо больше чем Unity. Хотя, честно сказать, мои взаимоотношения с Unity не сложились с самого начала, когда я увидел километровые файлы конфигурации к нему или же двухсот знаковые строки конфигурации в коде. Не будем о грустном.

StructureMap не только мне показался более удобным чем другие реализации DI\IoC, достаточно набрать в гугле StructureMap vs Unity и получить кучу ссылок, где люди обсуждают и показывают наглядно, что в работе самым удобным, гибким и естественным является StructureMap.
http://stackoverflow.com/questions/411660/enterprise-library-unity-vs-other-ioc-containers

http://stackoverflow.com/questions/21288/which-net-dependency-injection-frameworks-are-worth-looking-into

http://weblogs.asp.net/gunnarpeipman/archive/2010/09/21/unity-castle-windsor-structuremap-ninject-who-has-best-performance.aspx

Ко всему прочему StructureMap еще и достаточно быстрый.

Вот еще очень интересные подборки материалов по сравнению фреймворков
http://www.sturmnet.org/blog/2010/03/04/poll-results-ioc-containers-for-net

Я думаю можно опустить рассуждения о том, стоит ли такие контейнеры использовать в своем проекте или нет, и стоит ли писать свою реализацию. Я могу сказать, что в моей практике сначала случилось так, что были самописные реализации, которые заметно уступали по возможностям и удобству, но они решали свои конкретные задачи и этого на тот момент хватало. С развитием проекта, как-то было недосуг переводить все на какую-то другую технологию. Потом был Unity, Unity, но в конце концов я пришел к тому, что надо попробовать StructureMap и не пожалел об этом ни разу.

В чем, на мой взгляд, плюсы StructureMap:

- Настройка с помощью DSL
- Очень гибкая настройка всего
- Простота конечного использования
- Возможности по проверки внутренних связей
- Поддержка тестирования out-of-the-box

Еще много есть интересных и полезных вещей в StructureMap, но их было бы неправильно отнести к плюсам, лучше назвать их приятным дополнением, которые только облегчают жизнь.

Краткий план последующего материала выглядит следующим образом:

- Установка
- Регистрация
    - Основа
    - Профили
    - Плагины

  - Сканирование
  - Внедрение
- Конструкторы
  - Простые типы
  - Конструктор по умолчанию
  - Составные типы
  - Приведение типов
  - Задание аргументов
- Свойства
  - Простое задание свойств
  - Встроенное задание свойств
  - Задание свойств фреймворком
  - Допостроение существующих классов
- Время жизни
- Перехватчики
  - OnCreation
  - EnrichWith
- Дженерик типы
- Аттрибуты
  - DefaultConstructor
  - ValidationMethod
  - Все остальные
- Тесты

Установка StructureMap

Устанавливать StructureMap в свое приложение я советую с помощью NuGet. Одна команда в Package Manager Console (install-Package StructureMap) или же поиск и установка с помощью визарда – это наиболее легкие пути по получению. Но если хочется, то можно скачать проект с официальной страницы http://github.com/structuremap/structuremap/downloads

Регистрация

Самое главное действие наверно для любого IoC контейнера – это регистрация. От того насколько она удобна и понятна зависит, будут ли этим пользоваться люди и насколько вероятно неверное использование инструмента.

Автор советует использовать DSL так широко, как только это возможно и прибегать к файлу конфигурации, только когда надо отдельно задавать строки соединения, URL адреса, пути для файлов и все остальное в таком же духе.

Начнем с самого простого, регистрация класса с конструктором по умолчанию.

Пусть у нас есть такой простой набор классов:

```
public interface IClass {}
public interface IClass1 : IClass {}
public interface IClass2 : IClass {}

public class Class1 : IClass1 {}
```

```
public class Class2 : IClass2 {}
```

Это набор будет пока что основным, для демонстрации возможностей. Но вы не волнуйтесь, дальше классы и связи будут сложнее, потому что на таких классах многого тоже не покажешь.

Основа

Регистрация возможна с помощью контейнера (**Container**), с помощью статического класса **ObjectFactory** или же с помощью класса **Registry**. Постепенно рассмотрим регистрацию с помощью всех этих объектов. Основной класс для регистрации это Registry, остальные классы прокидывают его функционал, для удобства.

Регистрация с помощью статического класса ObjectFactory.

```
public class RegisterByObjectFactory {
    public RegisterByObjectFactory() {
        ObjectFactory.Initialize(x => {
                                    x.For<IClass1>().Use<Class1>();
                                    x.For<IClass2>().Use<Class2>();
                                });
    }
}
```

Все делается с помощью DSL и лямбда выражений. Сам DSL достаточно лаконичный и понятный, полученный код с легкостью складывается в осмысленные выражения. В данном случае можно легко прочитать, что *для интерфейса IClass1 надо использовать Class1*.

Получение объектов можно осуществить следующим способом, тоже интуитивно понятным:

```
    private static string ObjectFactoryExample() {
        new RegisterByObjectFactory();

        var class1Inst = ObjectFactory.GetInstance<IClass1>();
        var class2Inst = ObjectFactory.GetInstance<IClass2>();

        return ObjectFactory.WhatDoIHave();
    }
```

Основной метод для получения объекта GetInstance, в данном случае с указанием интерфейса. Дальше мы еще рассмотрим различные способы получения готовых объектов. Вы можете заметить, что метод возвращает строку, которую получаем от метода с говорящим названием **WhatDoIHave**. С помощью данного метода можно проводить диагностику внутренностей контейнера, смотреть что, как и где зарегистрировано.

Долгое время автор фреймворка не мог принять термин контейнер, по отношению к своему детищу, поэтому следующий способ был скрыт достаточно долго и только в более поздних реализациях открыл естественный ход регистрации, как оно было реализовано внутри, за статическим классом. Итак,

```
public class RegisterByContainer {
    public IContainer Container;

    public RegisterByContainer() {
        Container = new Container(x => {
```

```
                x.For<IClass1>().Use<Class1>();
                x.For<IClass2>().Use<Class2>();
            });
        }
    }
```

На первый взгляд все то же самое, лямбда та же самая, но теперь мы создаем класс, который будем потом отдавать наружу, и по которому будем обращаться к контейнеру. Т.е. еще раз, ObjectFactory это просто статический класс обертка над классом Container.

Получение объектов пойдет по тому же сценарию:

```
        private static string ContainerExample() {
            var container = new RegisterByContainer().Container;
            var class1Inst = container.GetInstance<IClass1>();
            var class2Inst = container.GetInstance<IClass2>();

            return container.WhatDoIHave();
        }
```

Следующий на очереди объект это **Registry**. Собственно его вы косвенно и вызывали все предыдущие разы. Для разнообразия зарегистрируем конкретные классы.

```
    public class RegisterByRegister {
        public IContainer Container;

        public RegisterByRegister() {
            var registry = new Registry();
            registry.ForConcreteType<Class1>();
            registry.ForConcreteType<Class2>();

            Container = new Container(x => x.AddRegistry(registry));
        }
    }
```

В данном случае используется метод **ForConcreteType**, что является синонимом для **.For<T>().Use<T>()**. Так же можно видеть, что класс Registry можно использовать как подконтейнер, собирать его и потом передавать на сборку в один контейнер. В данном случае проиллюстрировано добавление Registry в момент создания, но ничто не помешает написать:

```
            Container = new Container();
            Container.Configure(x => x.AddRegistry(registry));
```

Чтение конкретных классов ничем не отличается от обычного:

```
        private static string ConcreteClassExample() {
            var container = new RegisterByRegister().Container;
            var class1Inst = container.GetInstance<Class1>();
            var class2Inst = container.GetInstance<Class2>();

            return container.WhatDoIHave();
        }
```

Профили

StructureMap позволяет группировать соответствия классов используя именованные профили. Т.е. можно быстро переключаться между маппингов классов.

```
public class RegisteringProfiles {
    public IContainer Container;

    public RegisteringProfiles() {
        var registry = new Registry();
        registry.Profile("p1", x => x.For<IClass>().Use<Class1>());
        registry.Profile("p2", x => x.For<IClass>().Use<Class2>());

        Container = new Container(x => x.AddRegistry(registry));
    }
}
```

Здесь стоит обратить внимание, что классы Class1 и Class2 регистрируются по общему интерфейсу, но в разных профилях. Для того чтобы получить необходимый класс, надо переключиться между профилями в контейнере с помощью метода **SetDefaultProfile** который принимает имя профиля.

```
    private static string ProfilesExample() {
        var container = new RegisteringProfiles().Container;

        container.SetDefaultsToProfile("p1");
        var class1Inst = container.GetInstance<IClass>();

        container.SetDefaultsToProfile("p2");
        var class2Inst = container.GetInstance<IClass>();

        return container.WhatDoIHave();
    }
```

Имя профиля может быть только строковой переменной, но это уже не большая проблема. Я к тому, что не стоит в реальности писать имя профиля как в примере открытой строкой. Вредно для кармы.

После установки активного профиля, можно работать с контейнером как обычно. В итоге при выполнении одной и той же строки *container.GetInstance<IClass>();* мы получаем разные классы.

Плагины

Существует еще один способ для решения проблемы получения конкретного класса по общему интерфейсу, это именованный плагин.

Немного о терминологии. В IntelliSense и немного здесь можно встретить термин плагин, PluginType и PluggedType, в общем случае это означает *тип, который вы хотите получить*. Т.е. во всех предыдущих примерах IClass можно назвать PluginType, а Class1 или Class2 – PluggedType.

```
public class RegisterAsPlugin {
    public IContainer Container;

    public RegisterAsPlugin() {
        Container = new Container(x => {
                                      x.For<IClass>().Use<Class1>().Named("Class1");
                                      x.For<IClass>().Use<Class2>().Named("Class2");
                                  });
    }
```

```
}
```

На примере видно, что мы регистрируем классы по общему интерфейсу, но при этом задаем им конкретные имена. С помощью метода **Named** можно легко теперь запрашивать конкретный тип.

```
private static string PluginExample() {
    var container = new RegisterAsPlugin().Container;
    var class1Inst = container.GetInstance<IClass>("Class1");
    var class2Inst = container.GetInstance<IClass>("Class2");
    var instanceDef = container.GetInstance<IClass>();

    return container.WhatDoIHave();
}
```

В примере показывается, как можно обратиться к контейнеру и получить конкретный тип на общий интерфейс. Однако тут мы заодно затронем вопрос, а что же будет при попытке вызова метода GetInstance с общим интерфейсом без указания имени плагина?

Поведение по умолчанию следует поговорке «кто последний, тот и папа», т.е. в данном случае в переменную **instanceDef** попадет экземпляр класса **Class2**. Однако мы можем определять вполне явно класс «по-умолчанию». Для этого надо воспользоваться несколько иной формой регистрации плагинов.

```
public class RegisterAsPluginWithDefault {
    public IContainer Container;

    public RegisterAsPluginWithDefault() {
        Container = new Container(x => x.For<IClass>()
                        .AddInstances(i => {
                              i.Type(typeof (Class1)).Named("Class1");
                              i.Type(typeof (Class2)).Named("Class2");
                                                    })
                        .Use(new Class1())
            );
    }
}
```

И опять можно сказать, что пример сам себя описывает. Если прочитать его, то буквально дословно будет: *для интерфейса IClass добавить реализации типов Class1 по имени Class1, Class2 по имени Class2, использовать же Class1* (в данном случае очень даже конкретный класс, но можно было и как в предыдущих примерах написать .Use<Class1>()).

В данном случае именно метод **Use** говорит, какой тип будет использоваться для интерфейса по умолчанию. Если теперь выполнить следующий код

```
var instanceDef = container.GetInstance<IClass>();
```

то получим экземпляр класса Class1.

Use уже само по себе выставляет тип, используемый по умолчанию.

Сканирование

Логичным продолжением будет являться поиск и автоматическая регистрация типов в контейнере. Представим, что у нас не два класса наследуются от общего интерфейса, а 50! Будет

очень грустно и скучно заполнять руками все эти регистрации и зависимости. На такой случай есть у StructureMap метод **Scan**, который пробегает по интересующим вас сборкам или папкам и регистрирует подходящие объекты. Таким образом можно реализовать структуру плагинов для приложения и даже в чем-то составить конкуренцию MEF либо заменить его.

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.

Метод сканирование и поведение может быть переопределено, но пока что это не будет рассматриваться.

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.

После того, как вы указали на подопытные сборки, можно настроить процесс импорта более детально с помощью методов по включению/исключению типов по различным параметрам. За более детальной информацией лучше обратиться к документации. Она устарела, но общее представление о возможностях дает.

Итак, рассмотрим пример попроще:

```
public class RegisterByScan {
    public IContainer Container;

    public RegisterByScan() {
        Container = new Container(x => x.Scan(s => {
                                                  s.AddAllTypesOf(typeof (IClass));
                                                  s.AssembliesFromPath(".");
                                                  s.WithDefaultConventions();
                                                  s.LookForRegistries();
                                              }));
    }
}
```

В этом классе мы говорим, что хотим импортировать все типы, которые реализуют интерфейс IClass, из папки приложения, руководствоваться следует соглашениями по умолчанию. И последней строчкой идет команда на запуск поиска. Ранее все работало без явного указания. Но сейчас надо четко прописать метод **LookForRegistries**.

После того, как метод отработает можно посмотреть, что нашлось и зарегистрировалось в контейнере.

```
        private static string RegisterByScanExample() {
            var container = new RegisterByScan().Container;
```

```
            var instances = container.GetAllInstances<IClass>();

            return container.WhatDoIHave();
        }
```

Обратите внимание, что сейчас вызывается метод Get**All**Instances. Если вызвать метод для получения какого-то конкретного класса из зарегистрированных, то будет ошибка, так как StructureMap не знает какой именно класс возвращать «по-умолчанию».

Честно сказать, при такой реализации пользоваться результатами команды Scan невозможно. Для того чтобы все стало хорошо, и можно было бы к найденным классам обращаться по имени, надо код сканирования надо переписать немного.

```
    public class RegisterByScanWithNaming {
        public IContainer Container;

        public RegisterByScanWithNaming() {
            Container = new Container(x => x.Scan(s => {
                            s.AddAllTypesOf(typeof (IClass)).NameBy(t => t.Name);
                            s.AssembliesFromPath(".");
                            s.WithDefaultConventions();
                            s.LookForRegistries();
                                                    }));
        }
    }
```

К методу AddAllTypesOf добавили уточняющее правило, что все классы надо регистрировать по их имени. После такой модификации можно работать с конкретными типами:

```
var instance = container.GetInstance<IClass>("Class1");
```

Внедрение

В процессе работы с контейнером можно переопределить тип возвращаемый по умолчанию. Это применяется в основном в тестах. Демонстрация работы:

```
        private static string InjectExample() {
            var container = new RegisterAsPluginWithDefault().Container;
            var instance1 = container.GetInstance<IClass>("Class1");
            var instance2 = container.GetInstance<IClass>("Class2");
            var class1Inst = container.GetInstance<IClass>();

            container.Inject(typeof (IClass), new Class2());

            var class2Inst = container.GetInstance<IClass>();

            return container.WhatDoIHave();
        }
```

Ранее мы уже объявляли класс RegisterAsPluginWithDefault, который возвращает класс Class1 по умолчанию. С помощью метода **Inject** можно переопределить возвращаемый тип, необходимо лишь указать тип плагина и новый класс.

Данные примеры были на общие принципы регистрации, когда сами классы простые. В следующей теме рассмотрим, как быть с классами, у которых конструкторы с параметрами.

Конструкторы

Очень важный вопрос в реальном программировании применительно к разрешению зависимостей в IoC контейнерах, как быть с классами, у которых несколько конструкторов. Как их инициализировать, как задавать параметры, как дорабатывать и прочее и прочее. Надеюсь что на большинство вопросов ниже будет дан ответ. StructureMap действительно мощная и гибкая штука.

Перед тем как начать описывать возможности фреймворка, надо поговорить о тестовых классах. В этот раз они будут сложнее. Наследование, конструкторы с простыми типами, с составными.

Итак, пусть у нас будут следующие классы:

```
public interface IClassA : IClass {
    int A { get; set; }
}

public interface IClassB : IClass {}

public class ClassA : IClassA {
    public int A { get; set; }
    public int B { get; set; }
    public Class1 Class1 { get; set; }

    [DefaultConstructor]
    public ClassA() {}

    public ClassA(int a) {
        A = a;
    }
}

public class ClassB : IClassB {
    public IClassA ClassA;

    public ClassB(IClassA classA) {
        ClassA = classA;
    }
}

public class ClassM : IClassA {
    public int A { get; set; }

    public ClassM(int a) {
        A = a;
    }

    public ClassM(int a, int b) {
        A = a + b;
    }
}
```

Они с некоторой избыточностью на данный момент, которая пригодиться чуть дальше в повествовании.

Итак, начнем с самых простых вариантов.

Простые типы

Начнем с класса ClassA, у которого один из конструкторов принимает целочисленный параметр.

```
public class WithSimpeArguments {
    public IContainer Container;

    public WithSimpeArguments() {
        Container = new Container(x => x.For<IClassA>().Use<ClassA>()
                                        .Ctor<int>().Is(5)
            );
    }
}
```

Как видно из примера, к уже привычному объявлению добавляется указание на инициализацию конструктора значениями. В данном случае объявление конструктора идет по упрощенной схеме, так как только один параметр типа int. В процессе инициализации мы сразу указали значение. При этом StructureMap подскажет, что значение должно быть целочисленное.

Для проверки работы, можно вызвать уже привычный код

```
        private static string WithSimpleArgumentsExample() {
            var container = new WithSimpeArguments().Container;
            var classA = (ClassA) container.GetInstance<IClassA>();

            return classA.A.ToString();
        }
```

В результате на консоль выведется цифра 5.

Усложняем пример для случая, когда у нас в конструкторе два параметра одного типа. В качестве подопытного класс ClassM.

```
public class WithMultipleSimpeArguments {
    public IContainer Container;

    public WithMultipleSimpeArguments() {
        Container = new Container(x => x.For<IClassA>().Use<ClassM>()
                                        .Ctor<int>("a").Is(6)
                                        .Ctor<int>("b").Is(5));
    }
}
```

Теперь конструкция **.Ctor<int>("a").Is(6)** дополнена именем аргумента, для того, чтобы фреймворк смог точно сопоставить значения аргументам. Фреймворк всегда находит самый «жадный» конструктор и хочет, чтоб все аргументы были проинициализированы. Нельзя опустить задание значения для второго аргумента в текущей реализации класса. Но можно указать StructureMap, какой конструктор использовать по умолчанию, для этого надо использовать атрибут **DefaultConstructor**.

Конструктор по умолчанию

Атрибут **DefaultConstructor** позволяет явно указать, какой конструктор использовать для создания экземпляра класса. Чтобы можно было в предыдущем примере опустить объявление переменной b и ничего не упало в процессе работы.

```
public class ClassM : IClassA {
    public int A { get; set; }
```

```
        [DefaultConstructor]
        public ClassM(int a) {
            A = a;
        }

        public ClassM(int a, int b) {
            A = a + b;
        }
    }
```

Теперь можно использовать конструктор с одним параметром

Составные типы

Работать с составными типами совсем легко, потому что StructureMap сам обнаруживает и разрешает все зависимости по составным типам. Т.е. если посмотреть в начало, на классы, то видно, что класс ClassB инициализируется классом ClassA. Сейчас увидим что ничего особенного для разрешения такого рода зависимостей не надо.

```
    public class WithObjectArguments {
        public IContainer Container;

        public WithObjectArguments() {
            Container = new Container(x => {
                                x.For<IClassA>().Use<ClassA>().Ctor<int>().Is(5);
                                x.For<IClassB>().Use<ClassB>();
                                        });
        }
    }
```

Как видно из примера, никаких дополнительных операторов не применено. Зато можно запросить класс ClassB, который будет инициализирован ClassA.

```
        private static string WithObjectArgumentsExample() {
            var container = new WithObjectArguments().Container;
            var classA = (ClassB) container.GetInstance<IClassB>();

            return classA.ClassA.A.ToString();
        }
```

На экран будет выведена цифра 5.

Приведение типов

Если посмотреть на объявление класса ClassB, то можно заметить что переменная в конструкторе является интерфейсом, а не конкретным типом. Перепишем класс так, чтобы вместо интерфейса, конструктор принимал класса ClassA

```
public class ClassB : IClassB {
        public ClassB(ClassA classA) {
            ClassA = classA;
        }
    }
```

Теперь связывания типов не произойдет, потому что контейнер регистрирует и возвращает один тип данных. В нашем случае это IClassA.

```
        private static string WithObjectArgumentsForwardingAndWiringExample() {
            var container = new WithObjectArgumentsForwarding().Container;
            var classB = (ClassB) container.GetInstance<IClassB>();

            return classB.ClassA.A.ToString();
        }
```

Данный код вернет 0, так как будет вызван конструктор по умолчанию для класса ClassA, связывания не произошло.

Как вы догадались это не проблема. Можно указать StructureMap что к чему может быть приведено. Делается это с помощью команды **Forward** которой надо указать исходный тип и желаемый.

```
    public class WithObjectArgumentsForwarding {
        public IContainer Container;

        public WithObjectArgumentsForwarding() {
            Container = new Container(x => {
                            x.For<IClassA>().Use<ClassA>().Ctor<int>().Is(5);
                            x.Forward<IClassA, ClassA>();
                            x.For<IClassB>().Use<ClassB>();
                                      });
        }
    }
```

Теперь можно вызывать метод `WithObjectArgumentsForwardingAndWiringExample` и в ответе получить 5.

Задание аргументов

Чаще всего не удается заранее узнать и задать аргументы для класса, они становятся известны только на момент создания требуемого класса. И так как это рабочие будни написания программ, то в StructureMap не могла не появиться поддержка задания аргументов на момент создания класса.

Процесс вызова класса с новыми значениями аргументов похож на общение на языке мастера Йоды, но такова уж специфика DSL. Технические ограничения я имею в виду.

Итак, пусть мы хотим вызвать класс ClassB с новым классом ClassA. Для этого понадобится оператор With.

```
        private static string WithObjectArgumentsOverridingExample() {
            var container = new WithObjectArgumentsForwarding().Container;
            var classB = (ClassB) container
                                      .With(new ClassA(8))
                                      .GetInstance<IClassB>();

            return classB.ClassA.A.ToString();
        }
```

Так как фреймворк, в данном случае, может однозначно определить какому аргументу сопоставить экземпляр класса, то не надо указывать имя аргумента.

Если же надо указать большее количество аргументов для класса, то необходимо использовать большее количество метода With с указанием имени параметра и его значением.

```
        private static string WithObjectArgumentsOverridingExample() {
            var container = new WithObjectArgumentsForwarding().Container;
            var classB = (ClassB) container
                                    .With(new ClassA(8))
                                    .With("s").EqualTo(5)
                                    .GetInstance<IClassB>();

            return classB.ClassA.A.ToString();
        }
```

Теперь работа с конструкторами, я думаю, освещена в достаточной степени, чтобы можно было решить почти все архитектурные подходы, которые есть в вашем приложении.

Свойства

Помимо обязательных параметров конструктора, очень приятно во время создания экземпляра класса задавать значения свойств класса. Т.е. вместо того, чтобы писать

```
var classA = new ClassA();
classA.A = 8;
classA.B = 20;
```

можно гораздо короче и красивее задать свойства как:

```
var classA = new ClassA {
                            A = 8,
                            B = 20
                        };
```

Для StuctureMap это вполне по силам, причем все таким же элегантным и понятным способом.

Простое задание свойств

В самом простом случае, как на иллюстрации выше, можно задавать параметры с помощью метода **SetProperty**.

```
public class WithArgumentsSetProperty {
    public IContainer Container;

    public WithArgumentsSetProperty() {
        Container = new Container(x => x.For<IClassA>().Use<ClassA>()
                                        .Ctor<int>().Is(5)
                                        .SetProperty(p => {
                                                        p.A = 8;
                                                        p.B = 20;
                                                    }));
    }
}
```

Как видно из примера, свойства строго типизированны и можно пользоваться подсказками intelliSense, т.е. задание свойств протекает легко и непринужденно. Понятное дело, что можно задавать не все свойства, а только те, что хочется проинициализировать на этапе построения экземпляра класса.

Встроенное задание свойств

Для использования встроенного (inline) задания параметров используется метод **Setter**. С помощью этого метода можно задавать значения для одного параметра за раз. Так как аргументом метода является функция.

Самое простое, это явное задание параметра для инициализации.

```
public class WithArgumentsSetterExplicit {
    public IContainer Container;

    public WithArgumentsSetterExplicit() {
        Container = new Container(x => {
                        x.For<IClass1>().Use<Class1>();
                        x.For<IClassA>().Use<ClassA>()
                            .Ctor<int>().Is(5)
                            .Setter(c => c.Class1).Is(new Class1());
                            });
    }
}
```

В примере показано, что свойство Class1 будет всегда проинициализированно новым классом Class1. Такая запись должна применяться, если у класса более одного свойства одинакового типа. Если же у вас только одно свойство заданного типа, то фреймворк сможет самостоятельно определить к какому свойству присвоить значение переданного параметра.

Итак, неявная инициализации свойства.

```
public class WithArgumentsSetterImplicit {
    public IContainer Container;

    public WithArgumentsSetterImplicit() {
        Container = new Container(x => {
                        x.For<IClass1>().Use<Class1>();
                        x.For<IClassA>().Use<ClassA>()
                              .Ctor<int>().Is(5)
                              .Setter<Class1>().Is(new Class1());
                                  });
    }
}
```

В этом примере мы не указали имя свойства, но все прошло успешно, так как свойство типа Class1 только одно.

Задание свойств фреймворком

Раз уж StructureMap может автоматически подставлять экземпляры классов в аргументы конструкторов, то он сможет автоматически заполнять свойства классов?

Конечно сможет!

Но конечно он это сделает не по своему желанию и не для всех полей, а только для тех, которые будут указаны как автозаполняемые.

Можно модифицировать предыдущий пример, так, чтобы зависимости свойств разрешались средствами StructureMap и контролировались им же.

```
public class WithArgumentsSetterImplicitDefault {
    public IContainer Container;

    public WithArgumentsSetterImplicitDefault() {
        Container = new Container(x => {
                                      x.For<IClass1>().Use<Class1>();

                                      x.For<IClassA>().Use<ClassA>()
                                          .Ctor<int>().Is(5)
                                          .Setter<Class1>().IsTheDefault();
                                  });
    }
}
```

В примере появился новый метод **IsTheDefault**, который говорит фреймворку разрешить зависимость своими средствами. Т.е. в данном случае свойство типа Class1 у класса ClassA будет создано и присвоено исходя из того, как зарегистрирован Class1.

Так же есть пакетная инициализация параметров, когда можно сказать что все свойства определенного типа должны инициализироваться значениями по умолчанию. Для этого используется команда **SetAllProperties**.

```
public class WithArgumentsSetterBatchImplicitDefault {
    public IContainer Container;

    public WithArgumentsSetterBatchImplicitDefault() {
        Container = new Container(x => {
                                      x.For<IClass1>().Use<Class1>();
                                      x.For<IClassA>().Use<ClassA>()
                                          .Ctor<int>().Is(5);
                                      x.SetAllProperties(c => c.OfType<Class1>());
                                  });
    }
}
```

С помощью этого указания StructureMap автоматически проинициализирует все свойства вызываемых классов, у которых типом свойства является Class1.

Допостроение существующих классов

Порой бывает так, что можно получить только уже готовый класс, без возможности как-то повлиять на его создание. При этом хочется автоматизировать заполнение кучи полей класса в автоматическом режиме. И такое возможно с помощью StructureMap.

Пусть к нам приходит ClassA который создается не в нашей системе. Надо проинициализировать его свойство типа Class1. Сначала настроим StructureMap.

```
public class WithArgumentsBuildUp {
    public IContainer Container;

    public WithArgumentsBuildUp() {
```

```
            Container = new Container(x => {
                                        x.For<IClass1>().Use<Class1>();
                                        x.Forward<IClass1, Class1>();
                                        x.SetAllProperties(c => c.OfType<Class1>());
                                    });
        }
    }
```

Теперь можно вызвать метод **BuildUp** который достроит объект, по имеющимся конфигурациям.

```
var container = new WithArgumentsBuildUp().Container;
var classA = new ClassA(14);

container.BuildUp(classA);
```

На второй строчке свойство Class1 = null, после вызова BuildUp, объект полностью готов.

Время жизни

Немаловажным фактором является возможность управлением времени жизни объекта. Для каких-то классов надо получать один и тот же экземпляр, для других – каждый раз новые. Этим тоже можно управлять в процессе создания правил в контейнере.

Фреймворк оперирует семью политиками управления времени жизни объекта:

- Per reques (default) – создается каждый раз новый объект.
- HttpContextLifecycle
- SingletonLifecycle
- ThreadlocalStorageLifecycle
- UniquePerRequestLifecycle – гарантирует что вся цепочка объектов инициализирующих запрошеный объект будет уникальной.
- HttpSessionLifecycle
- HybridLifecycle - это HttpSessionLifecycle или ThreadlocalStorageLifecycle
- HybridSessionLifecycle – или HttpContextLifecycle, или HttpSessionLifecycle

Рассмотрим некоторые из них на примере простого класса

```
public class ClassX : IClassX {
    public int Counter { get; private set; }

    public void Increase() {
        Counter++;
    }
}

public interface IClassX {}
```

Первым на очереди пусть будет **Singleton**.

```
public class LifecycleSingleton {
    public IContainer Container;

    public LifecycleSingleton() {
```

```
            Container = new Container(x => x.For<IClassX>().LifecycleIs(new
SingletonLifecycle()).Use<ClassX>());
            Container = new Container(x => x.For<IClassX>().Singleton().Use<ClassX>());
        }
    }
```

Для основных политик жизни определены сокращенные методы. Т.е. можно использовать как **Singleton()**, так и **LifecycleIs(new SingletonLifecycle())**.

В качестве проверки можно использовать наглядный пример:

```
        private static string LifecycleSingleton() {
            var singleton = new LifecycleSingleton().Container;
            var classX = (ClassX) singleton.GetInstance<IClassX>();

            classX.Increase();
            Console.WriteLine(classX.Counter);

            classX = (ClassX) singleton.GetInstance<IClassX>();

            classX.Increase();
            Console.WriteLine(classX.Counter);


            return "done";
        }
```

В итоге на консоль выведутся данные: «1, 2, Done». При простом объявлении мы бы получили: «1, 1, Done».

Для хранения экземпляра класса в рамках одного потока используется **ThreadLocalStorageLifecycle**, или сокращенная форма записи **HybridHttpOrThreadLocalScoped**

```
    public class LifecycleThreadLocal {
        public IContainer Container;

        public LifecycleThreadLocal() {
            Container = new Container(x => x.For<IClassX>()
                            .LifecycleIs(new ThreadLocalStorageLifecycle())
                            .Use<ClassX>());
            Container = new Container(x => x.For<IClassX>()
                            .HybridHttpOrThreadLocalScoped()
                            .Use<ClassX>());
        }
    }
```

Для **HttpContextLifecycle** определена сокращенная запись **HttpContextScoped**

```
     public class LifecycleHttpContext {
        public IContainer Container;

        public LifecycleHttpContext() {
            Container = new Container(x => x.For<IClassX>()
                            .LifecycleIs(new HttpContextLifecycle())
                            .Use<ClassX>());
            Container = new Container(x => x.For<IClassX>()
                            .HttpContextScoped()
                            .Use<ClassX>());
        }
    }
```

Перехватчики

С версии 2.5+ появилась возможность постобработки только что созданного объекта, либо полной его замены. В данном случае не ставится целью создать еще одни AOP фреймворк, так как их уже достаточно в мире, просто это может облегчить жизнь.

Для постобработки существует два+ метода:

- OnCreation – принимает в качестве параметра Action, позволяет провести свою (custom) инициализацию объекта. Есть доступ к контексту SturctureMap.
- EnrichWith – принимает в качестве параметра Function. Может возвращать любой тип совместимый с запрашиваемым.
- Свои перехватчики

Честно сказать, в простых примерах не очень хорошо получается передать разницу и преимущества использования OnCreation перед EnrichWith и наоборот.

OnCreation

Для демонстрации можно воспользоваться следующим простым классом:

```
public class ClassS : IClassS {
    public int Counter { get; private set; }

    public void Init() {
        Counter = -100;
    }

    public void Init(IClass1 class1) {
        Counter = -50;
    }

    public void Increase() {
        Counter++;
    }
}
```

Теперь, можно продемонстрировать использование метода **OnCreation**.

```
public class InterceptionsSimple {
    public IContainer Container;

    public InterceptionsSimple() {
        Container = new Container(x => x.For<IClassS>()
                                        .Use<ClassS>()
                                        .OnCreation(c => c.Init()));
    }
}
```

При получении класса из IoC фреймворка мы можем увидеть, что значение Counter равняется -100. Как видите, в применении все очень просто и опять же интуитивно понятно, что будет делать код. Неподготовленный человек сможет прочитать определение класса в StructureMap и сразу понять, что и как здесь происходит.

Так же возможен вариант, когда в метод инициализации надо передать объект сконструированный средствами самого StructureMap. Для этих целей используем тот же перегруженный метод.

```
public class InterceptionWithContext {
    public IContainer Container;

    public InterceptionWithContext() {
        Container = new Container(x => {
                            x.For<IClass1>().Use<Class1>();
                            x.For<IClassS>().Use<ClassS>()
                             .OnCreation((context, cls) => {
                                var class1 = context.GetInstance<IClass1>();
                                cls.Init(class1);
                                                        });
                                });
    }
}
```

В данном случае в лямбда-выражении есть доступ к самому контейнеру и можно получать объекты, конструирование которых возложено на StructureMap.

EnrichWith

Данный метод позволяет обернуть конструируемый класс в другой, если возможно приведение типов. Например, пусть будет класс

```
public class ClassSs : ClassS {
    private readonly ClassS s;

    public int Abs {
        get { return Math.Abs(s.Counter); }
    }

    public ClassSs(ClassS s) {
        this.s = s;
    }
}
```

Тогда можно будет его получать из StructureMap с помощью следующего кода:

```
public class InterceptionWithContext2 {
    public IContainer Container;

    public InterceptionWithContext2() {
        Container = new Container(x => {
                            x.For<IClass1>().Use<Class1>();
                            x.For<IClassS>().Use<ClassS>()
                               .EnrichWith(cls => {
                                   cls.Init();
                                   return new ClassSs(cls);
                                                    });
                                });
    }
}
```

В результате, при попытке получить экземпляр класса реализующего интерфейс IClassS мы будем получать класс ClassSs.

Шаблонные (generic) типы

Без шаблонных типов наверно уже сложно представить себе более-менее большую программу, поэтому, хотя данная тема и получится маленькой в разрезе рассмотрения StructureMap, но она важна.

Итак, допустим у нас есть шаблонный адаптер.

```
public interface IAdapter {
    string SomeMethod();
}

public interface IAdapter<T> : IAdapter {}

public class Adapter<T> : IAdapter<T> {
    public string SomeMethod() {
        return typeof (T).ToString();
    }
}
```

Так же существует несколько его наследников с конкретными типами

```
public class StringAdapter : Adapter<string> {}
public class IntAdapter : Adapter<int> {}
```

Никаких ограничений на типы T в адаптере не предусмотрены.

Есть несколько способов для работы с шаблонными типами. Самый простой способ это зарегистрировать и вызывать конкретные типы StringAdapter, IntAdapter.

```
public class GenericTypes {
    public IContainer Container;

    public GenericTypes() {
        Container = new Container(x => {
                        x.For(typeof (IAdapter<>)).Use(typeof (StringAdapter));
                        x.For<IAdapter<int>>().Use<IntAdapter>();
                                });
    }
}
```

В примере показано два идентичных способа регистрации шаблонного класса.

Вызывать их можно несколькими способами.

Способ раз: Можно попросить контейнер выдать нам адаптер по конкретному типу.

```
    private static string GenericTypesExample() {
        var container = new GenericTypes().Container;
        var stringAdapter = container.GetInstance<StringAdapter>();
        var intAdapter = container.GetInstance<IntAdapter>();

        return stringAdapter.SomeMethod() + " " + intAdapter.SomeMethod();
    }
```

Второй способ более общий и, на мой взгляд, более применим на практике. С помощью второго метода можно передавать тип ключа (T), по которому StrutureMap сможет вернуть необходимый класс.

Составление контейнера остается тем же, изменяется способ получения.

```
        private static string GenericTypesExample() {
            var container = new GenericTypes().Container;
            var instance = container
                .ForGenericType(typeof (IAdapter<>))
                .WithParameters(typeof(string))
                .GetInstanceAs<IAdapter>();


             return  instance.SomeMethod();
        }
```

Т.е. сообщаем контейнеру что будем требовать шаблонный тип, для такого-то типа, и вернуть экземпляр как тип IAdapter.

Для реального испльзование оно бы выглядело как:

```
private static string GenericTypesExample<T>() {
            var container = new GenericTypes().Container;
            var instance = container
                .ForGenericType(typeof (IAdapter<>))
                .WithParameters(typeof(T))
                .GetInstanceAs<IAdapter>();

             return  instance.SomeMethod();
        }
```

Конкретный адаптер будет определяться по типу передаваемого параметра T.

Аттрибуты

StructureMap может быть сконфигурирован до определенной степени с помощью атрибутов. Уже освещался атрибут **DefaultCounstructor**, который указывает на конструктор, который должен использоваться по умолчанию. Есть атрибуты для указания классов и интерфейсов для регистрации, какие свойства класса автоматически заполнять, а так же задавать методы для валидации.

Сам автор рекомендует не увлекаться атрибутами, так как они узкоспециализированные, позволяют осуществить только базовую конфигурацию и раскиданы по всему проекту, что затрудняет поддержку. Лучше все объявлять в одном месте.

Самый полезный атрибут уже указан, следующий по полезности **ValidationMethod**, остальные использовать не рекомендуется, но если очень хочется, то вот краткое описание.

ValidationMethod

Используется для самопроверки классов. Т.е. вы написали какой-то класс и можно написать метод, который будет определять правильность создания класса. StructureMap может его

подцеплять и выполнять для самопроверки. Кажется это вообще уникальная способность рассматриваемого фреймворка.

Пусть у нас есть класс, для которого надо задавать определенное свойство. Это может быть строка соединения с базой данных, настройки взаимодействия, сложные связи. В нашем примере ограничимся просто тем, что поле не должно быть пустым.

```
public class SelfValidation {
    public string Name { get; set; }


    [ValidationMethod]
    public void IsClassBuildCorrectly() {
        if(string.IsNullOrWhiteSpace(Name))
            throw new ArgumentException("Name can't be null or empty");
    }
}
```

Видно, что метод помечен как ValidationMethod. Вообще таких методов может быть больше одного, StructureMap сканирует их всех и пытается выполнить поочередно при вызове метода **AssertConfigurationIsValid**. Будьте осторожны с этим методом, так как при большой конфигурации фреймворк попробует построить все зависимости, заполнить все поля, запустить все методы помеченные рассматриваемым атрибутом.

Регистрация класса пусть будет проведена следующим образом:

```
public class ValidationShowcase {
    public IContainer Container;

    public ValidationShowcase() {
        Container = new Container(x => x.ForConcreteType<SelfValidation>());
    }
}
```

Теперь можно попробовать его получить из контейнера.

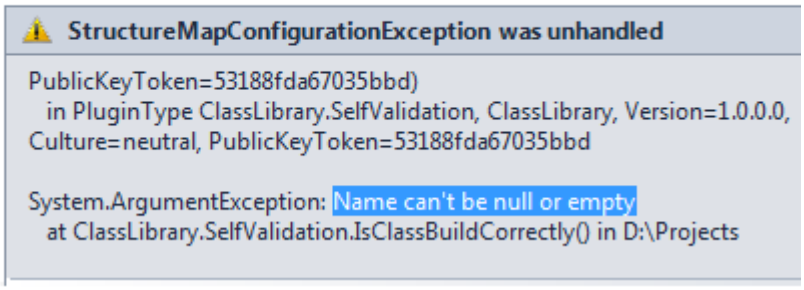
```
    private static string ValidationShowcaseExample() {
        var container = new ValidationShowcase().Container;
        container.AssertConfigurationIsValid();

        return "";
    }
```

При вызове метода выше, вы получите StructureMapConfigurationException где будет показаны внутренние исключения вызванные проверками внутреннего устройства StructureMap.

```
private static string ValidationShowcaseExample() {
    var container = new ValidationShowcase().Container;
    container.AssertConfigurationIsValid();

    return "";
}
```



На скриншоте видно, что в сообщении об исключительной ситуации как раз фигурирует наше сообщение.

PluginFamily

Указывает StructureMap, что помеченный тип будет использоваться как тип плагинов. Эквивалентно тому, как если бы мы написали *.For<plugin type>*.

Можно так же указать тип, который будет возвращаться по умолчанию, и можно конкретно указать, что это будет синглтон.

```
[PluginFamily("Default", IsSingleton = true)]
```

Pluggable

Помеченный тип будет включен в коллекцию плагинов, указывает на конкретную реализацию. Эквивалентно использованию *.Use<pluggable type>*. Необходимо всегда использовать имя для типа.

```
[Pluggable("Default")]
```

SetterProperty

Указывает на то, что помеченное атрибутом свойство нужно инициализировать средствами фреймворка. Они будут являться обязательными и если StructureMap не сможет их инициализировать, то будет ошибка исполнения.

Тесты

Помимо того, что StructureMap может сам себя проверить на корректность определения всех классов, параметров и других составляющих, он еще позволяет точечно работать с проверкой зарегистрированных классов. В фреймворк встроены средства для тестирования, чтобы не пришлось изобретать велосипеды, а красиво и лаконично писать тесты.

В StructureMap встроены RhinoMock и Moq фреймворки. Для того, чтобы ими воспользоваться необходимо будет доставить пакет NuGet **structuremap.automocking**, после чего можно будет использовать мокирование объектов.

Рассмотрение работы Moq и RhinoMock не входит в рамки статьи.

Заключение

Надеюсь, что у вас появилось желание поподробнее и на практике изучить StructureMap. Еще очень советую посмотреть на исходники проекта, можно подчерпнуть интересные и полезные идеи.

За бортом статьи остались темы по созданию контейнера на основе файла конфигурации. Работа с вложенными контейнерами и зачем они нужны. Я надеюсь, что в недалеком будущем этот пробел будет восполнен.

Hard’n’heavy!

Violet Tape