MEF

Сегодня я хочу рассказать о системе постороения динамического, модульного приложения с использованием Managed Extensibility Framework (MEF). Вообще это уже достаточно старая технология, котороя вылилась из системы Add-On от компании Микрософт. MEF стала дружелюбной оболочкой над монструозной системой предложенной ранее. Результат оказался настолько хорош, что MEF включили в поставку .Net Framework 4 по умолчанию, так что не потребуется искать и загружать библиотеки отдельно, беспокоится, есть ли данная библиотека у пользователя.

Так же уже понятно, что данную технологию можно использовать в Enterprise разработке, так как она в основной поставке фреймворка, а это такой аргумент, против которого менеджеры проектов не попрут. Обычно, менеджеры проектов очень неохотно идут на использование технологий и продуктов которые не упомянуты в пресс-релизах MS, даже если данная технология широко и успешно используется сообществом. Хотя их можно понять, это будет первый пункт, по которому их будут пинать в случае *любых* накладок.

Итак, MEF легко позволяет сделать приложение модульным, и подключение новых библиотек может быть осуществленно на лету, без использования сложных приемов. Весь код остается прозрачным и легким для понимания.

MEF можно использовать начиная с версии .Net 3.5, взяв необходимые библиотеки с CodePlex. Как вы уже догадались, оттуда же можно взять исходный код и посмотреть как все это работает.

Основные возможности MEF:

- Предоставляет стандартизированный способ для организации мастер-приложения и для дополнений к нему. В общем случае дополнения не привязаны к конкретному приложению и при реализации общего интерфейса могут быть использованы разными программами, что в целом не отменяет и жесткой привязки к мастер-приложения. Дополнения могут быть зависимы друг от друга и MEF самостоятельно подключит их в нужном порядке.
- Предоставляет возможности поиска и загрузки дополнений силами самого MEF
- Позволяет помечать дополнения метаданными, для дополнительных возможностей по подключению и фильтрации дополнений.

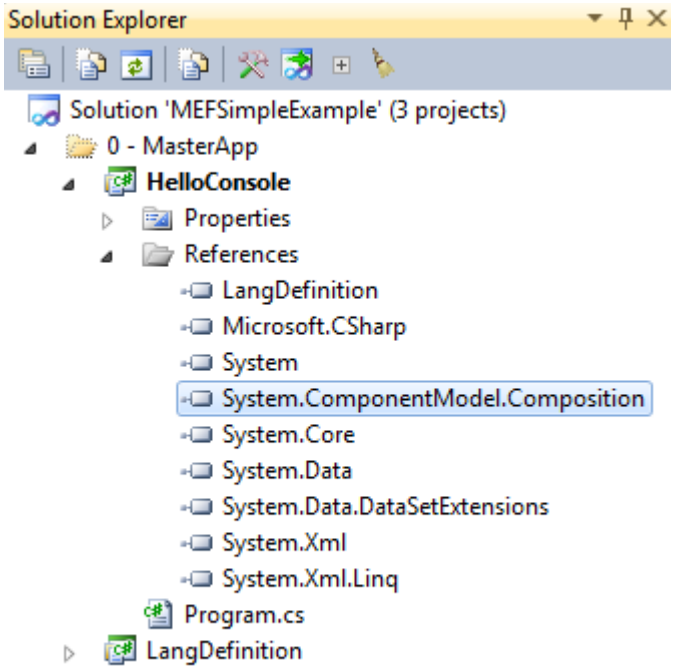
Создание простого приложения

Для начала сделаем что-то в духе HelloWorld приложения, а потом сделаем что-то более полезное, так как я бы сам себя проклял, если бы остановился на примере HelloWorld которое обычно ничего не демонстрирует))

Итак, приложение будет самое простое которое только может быть. Пусть оно будет консольное и будет выводить фразу на разных языках, в зависимости от того, какую библиотеку мы подуснем приложению. Для начала самое оно! Можно будет сосредоточится только на MEF коде.

Создаем консольное приложение. Оно будет мастер-приложением. Для того, чтобы подключать и узнавать другие библиотеки выделим интерфейс коммуникации в дополнительную сборку. Итак, на данном этапе у нас будет консольный проект **HelloConsole** и сборка **LangDefiniton**.

К проекту HelloConsole добавляем ссылку на сборку **System.ComponentModel.Composition** и на нашу библиотеку **LangDefinition**. К данному моменту решение должно выглядеть следующим образом:



В проекте LangDefinition создаем новый интерфейс, где определим метод SayHello, который будет возвращать нужный текст.

```
namespace LangDefinition {
    public interface ILanguage {
        string SayHello();
    }
}
```

После того, как определили интерфейс, можно написать код, который будет подгружать и использовать дополнения к нашему приложению. Для этого переходим в проект HelloConsole и там создаем класс **AddingComposer**.

Метод, который отвечает за обнаружение и подгрузку дополнений будет LoadAddings. В нем будем указывать в какой директории искать дополнения и подгружать их.

```
    public void LoadAddings() {
        var catalog = new AggregateCatalog();
        catalog.Catalogs.Add(new DirectoryCatalog("Adding"));

        var container = new CompositionContainer(catalog);
        container.ComposeParts(this);
    }
```

На третьей строке указывается, где именно искать дополнения. В данном случае указан провайдер DirectoryCatalog, который позволяет указать путь до директории с дополнениями (абсолютные или относительные пути от исполняемой библиотеки, в данном случае исполняемого файла), в том числе указать фильтры поиска библиотек.

Так же можно указывать сборки (Assembly), где искать дополнения. Для этого надо будет создать **AssemblyCatalog**. Кроме этого можно указать конкретно типы, которые надо будет импортировать. Для этого используйте **TypeCatalog**.

На пятой строке указывается что и каким образом собирать в основное приложение. Можно помечать дополнения как потокобезопасные.

Чтобы получить доступ к найденому дополнению, необходимо выставить поле с нашим интерфейсом и пометить его специальным аттрибутом **Import**, который укажет системе что это импортируемый из дополнения тип. Итого класс будет выглядеть таким образом:

```
public class AddingComposer {
    [Import]
    public ILanguage Language { get; set; }

    public void LoadAddings() {
        var catalog = new AggregateCatalog();
        catalog.Catalogs.Add(new DirectoryCatalog("Adding"));

        var container = new CompositionContainer(catalog);
        container.ComposeParts(this);
    }
}
```

На данный момент все готово, к тому, чтобы написать собственно код дополнения.

Добавляем в проект новую библиотеку классов. Назовем ее RussianHello. Ссылаемся в ней на проект с интерфейсом, на **System.ComponentModel.Composition** и реализуем интерфейс ILanguage.

```
namespace RussianHello {
    [Export(typeof (ILanguage))]
    public class RuHello : ILanguage {
        public string SayHello() {
            return "Привет";
        }
    }
}
```

Для того, чтобы MEF увидел класс, надо его пометить аттрибутом **Export**. В данном случае можно было не конкретизировать интерфейс экспорта.

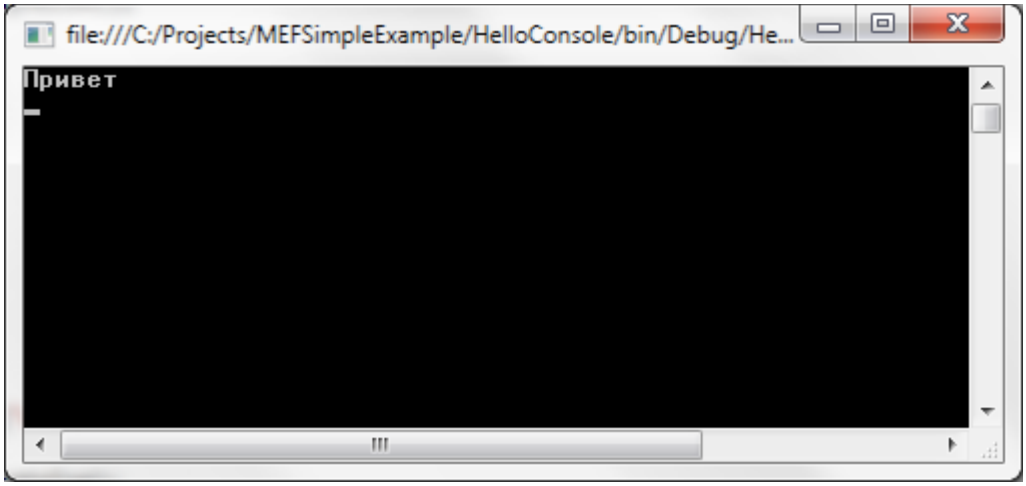
Еще один момент, для перед проверкой приложения. Будет полезно указать папку формирования библиотеки сразу на Addings в директории, где будет лежать исполняемый файл.

Финальным действием будет вызов класса AddingComposer в теле Main.

```
        private static void Main(string[] args) {
            var addingComposer = new AddingComposer();

            addingComposer.LoadAddings();
            var hello = addingComposer.Language.SayHello();
            Console.WriteLine(hello);
```

```
            Console.ReadLine();
        }
```

После этого запускаем приложение и видим:



Это базовые возможности MEF, на которых многие статьи и заканчиваются, но мы конечно пойдем дальше, последовательно развивая приложение.

Добавляем еще один язык

Создаем еще одну сборку, где реализуем интерфейс ILanguage для английского языка.

```
namespace EnglishHello {
    [Export(typeof (ILanguage))]
    public class EnHello : ILanguage {
        public string SayHello() {
            return "Hello";
        }
    }
}
```

После этого можно будет закидывать нужные библиотеки в папку Adding и наблюдать результат, что сообщение меняется.

Но это произойдет только в том случае, если в папке дополнений только одно дополнение. Если же вы закините оба дополнения, то в процессе выполнения приложение выкинет ошибку о том, что невозможно разрешить какое же дополнение загружать и использовать.

Для разрешения этой ситуации будем использовать метаданные для экспорта приложений.

Метаданные для экспорта

Метаданные для экспорта могут быть нетипизированные и типизированные. Начнем с нетипизированных, как более простых в использовании.

Слабая типизация

Метаданные добавляются с помощью специального аттрибута ExportMetadata. Нам потребуется конструктор, где можно задавать имя переменной и ее значение, которое будет использоваться для однозначной идентификации дополнения.

Для класса RuHello добавим

```
[ExportMetadata("Lang","Ru")]
```

Тогда получится

```
[Export(typeof (ILanguage))]
[ExportMetadata("Lang","Ru")]
public class RuHello : ILanguage { ... }
```

Аналогично дополним класс EnHello, чтобы получилось

```
[Export(typeof (ILanguage))]
[ExportMetadata("Lang","En")]
public class EnHello : ILanguage { ...  }
```

После этого можно будет выбирать сборки по полю **Lang**. Саму выборку будем осуществлять в мастер-приложении.

Для множественного импорта используется аттрибут **ImportMany** с указанием типа для импорта. Этот аттрибут лучше навешивать на ленивую коллекцию из MEF пространства имен, для отложенной инициализации компонентов.

```
[ImportMany(typeof (ILanguage))]
private Lazy<ILanguage, IDictionary<string, object>>[] langVaults { get; set; }
```

Выбор нужного дополнения можно осуществлять следующей выборкой:

```
public ILanguage GetHello(string type) {
            LoadAddings();

            return (from codeVault in langVaults
                    where (string) codeVault.Metadata["Lang"] == type
                    select codeVault).Single().Value;
}
```

После этого основной код приложения будет выглядеть как:

```
public class AddingComposer {
        [ImportMany(typeof (ILanguage))]
        private Lazy<ILanguage, IDictionary<string, object>>[] langVaults { get; set; }

        private void LoadAddings() {
            var catalog = new AggregateCatalog();
            catalog.Catalogs.Add(new DirectoryCatalog("Adding"));

            var container = new CompositionContainer(catalog);
            container.ComposeParts(this);
        }

        public ILanguage GetHello(string type) {
            LoadAddings();
```

```
            return (from codeVault in langVaults
                    where (string) codeVault.Metadata["Lang"] == type
                    select codeVault).Single().Value;
        }
    }


        private static void Main(string[] args) {
            var lang = Console.ReadLine();

            var addingComposer = new AddingComposer();
            var hello = addingComposer
                .GetHello(lang)
                .SayHello();
            Console.WriteLine(hello);


            Console.ReadLine();
        }
```

Можно пробовать вводить значения **Ru** и **En** и видеть как приложение нам выводит данные из нужного дополнения.

Самое приятное, что до запуска приложения можно вообще очистить папку Adding, и копировать туда дополнения по мере работы. Они будут подхватываться без перезапуска приложения.

Строгая типизация

Теперь попробуем все то же самое с типизированными метаданными.

В общем алгоритм выглядит следующим образом: объявляем класс, который будет типизированным аттрибутом, наследуем его от ExportAttribute. Создаем интерфейс, который будет повторять набор свойств, но только на чтение. Используем вновь созданный аттрибут на нужном классе.

Допустим что у нас метаданные разрослись до такого состояния:

```
namespace EnglishHello {
    [Export(typeof (ILanguage)),
        ExportMetadata("Lang","En"),
        ExportMetadata("Version","2"),
        ExportMetadata("Comment","English language"),
    ]
    public class EnHello : ILanguage {
        public string SayHello() {
            return "Hello";
        }
    }
}
```

Логически языки вполне ограничены и могут задаваться перечислимым типом, версия может быть только целочисленным значением, комментарий это строка. Чтобы получить все плюсы типизации, нужно создать свой атрибут с необходимыми типами. С этого и начнем.

```
    [MetadataAttribute]
    [AttributeUsage(AttributeTargets.Class, AllowMultiple = false)]
    public class LangMetaData : ExportAttribute {
        public Lang Language { get; set; }
```

```
        public int Version { get; set; }
        public string Comment { get; set; }
    }
```

Класс наследуется от ExportAttribute, при этом к нему применяются атрибуты MetadataAttribute и указывается с какими типами он будет использоваться и как. В данном случае аттрибут должен применятся только к классам, при этом множественное применение через наследование запрещено.

Теперь надо создать интерфейс который будет повторять набор свойств, причем они должны быть объявлены только на чтение.

```
public interface ILangMetaData {
        Lang Language { get;  }
        int Version { get;  }
        string Comment { get; }
    }
```

Интерфейс можно объявить рядом с аттрибутом.

Далее можно применять аттрибут. Это выглядит более симпатично, нежели нетипизированные метаданые.

```
namespace EnglishHello {
    [Export(typeof(ILanguage)),
    LangMetaData(
         Language = Lang.En,
         Version = 2,
         Comment = "English Language"
         )
    ]
    public class EnHello : ILanguage {
        public string SayHello() {
            return "Hello";
        }
    }
}
```

Премущества такого подхода очевидны:

- Нет возможности сделать опечатку, компилятор заметит;
- Точно известно какие параметры могут быть, intelliSense подскажет;
- Легко изменять и искать по проекту использование свойств;
- Вменяемый фильтр при поиске дополнений к приложению.

Осталось только модифицировать мастер-приложение, чтобы оно понимало строготипизированные метаданные. Список объявим как словарь из нужного нам интерфейса и интерфейса метаданных.

```
  [ImportMany]
  private Lazy<ILanguage, ILangMetaData>[] langVaults { get; set; }
```

Обращаю внимание, что у атрибута ImportMany не указан тип, который надо загружать. Если оставить тип, то дополнения подгрузятся два раза и MEF не сможет корректно разрешить зависимости. Надо указать тип либо в ImportMany, либо в Export, но не в двух местах одновременно.

Метод GetHello теперь изменится, и на мой взгляд станет более информативным и аккуратным

```
public ILanguage GetHello(Lang lang) {
    LoadAddings();

    return (from codeVault in langVaults
            where codeVault.Metadata.Language == lang
            select codeVault).Single().Value;
}
```

Хм... не очень пока получается сложные примеры показать, да? Ну так это наверно потому что не особенно сложные вещи и рассматривали. Остальное в следующей части, а то и так тут слишком много букв. Ждите ;)

Hard'n'heavy!

MEF II

Продолжаем разговор о MEF и сегодня на повестке дня небольшие дополнения о экспорте/импорте, время жизни дополнений, способы создания дополнений, наследование дополнений.

Экспорт/импорт по контракту

До этого экспорт/импорт компонентов происходил по типу дополнения. Таким образом создавался конракт по умолчанию и можно было либо использовать единственное поле, куда экспортировалось дополнение, либо получать их пачкой и выбирать оттуда нужное дополнение по метаданным.

С помощью контрактов, можно назначать дополнения на конкретные, заранее определенные поля, без привлечения фильтров и воборок в ручную.

Итак, пусть у нас есть все те же два дополнения. Теперь только мы объявим контракты к ним. Это делается очень просто:

```
[Export("RuContract", typeof(ILanguage))]
    public class RuHello : ILanguage { ... }

[Export("EnContract", typeof(ILanguage))]
    public class EnHello : ILanguage { ... }
```

После это в мастер-приложении надо будет только указать поля, помеченные как импорт определенных контрактов. Конечно про композицию не сто ит забывать тоже.

```
    public class AddingComposer {
        [Import("RuContract")]
        public ILanguage Ru { get; set; }

        [Import("EnContract")]
        public ILanguage En { get; set; }

        public void Compose() {
            var catalog = new AggregateCatalog();
            catalog.Catalogs.Add(new DirectoryCatalog("Adding"));

            var container = new CompositionContainer(catalog);
            container.ComposeParts(this);
        }
    }
```

Как видите, не надо теперь делать выборок по массиву загруженных дополнений. Они сами встали на свои места, как мы того ждем. В документации кажется приводятся примеры, когда можно не объявлять тип контракта, но тогда MEF не узнает, что чему сопоставлять. Выходом может стать тип dynamic.

Область видимости дополнений/время жизни

После того, как вы наиграетесь с основными настройками и возможностями MEF, возникает вопрос, а в какой момент создаются дополнения, сколько они живут в рамках нашего процесса.

По умолчанию, если не указано другого, все дополнения инициализируются при первом обращении пустым конструктором и не пересоздаются на протяжении всего времени работы приложения. При запросе на новую импортируюемую часть, будет найдена и сопоставлена уже существующий элемент.

С помощью атрибутов мы можем самостоятельно задать время жизни дополнения.

Существует три возможных состояния управления:

- Any - по умолчанию. Сразу может быть как Shared, так и NonShared
- Shared – используется один экземпляр экспорта, для всех импортируемых элементов в рамках одного контейнера.
- NonShared – на каждый запрос импорта создается новый экземпляр экспортируемого типа.

-	**Part.Any**	**Part.Shared**	**Part.NonShared**
Import.Any	Shared	Shared	Non Shared
Import.Shared	Shared	Shared	*No Match*
Import.NonShared	Non Shared	*No Match*	Non Shared

Лучше всего продемонстрировать это на примере.

Итак, у нас есть все те же классы для экспорта. Модифицируем их для демонстрации работы обсуждаемого атрибута. Один из классов пометим как общий ресурс, другой как индивидуальный.

```
[Export("EnContract", typeof(ILanguage))]
[PartCreationPolicy(CreationPolicy.NonShared)]
public class EnHello : ILanguage { ... }

[Export("RuContract", typeof(ILanguage))]
[PartCreationPolicy(CreationPolicy.Shared)]
public class RuHello : ILanguage { ... }
```

Для визуализации пусть в теле класса инициализируется и обновляется сообщение, которое мы увидим в итоге.

Необходимо внести изменения и в импорт дополнений в мастер-приложении. Сделаем дубликаты полей, чтобы стало примерно так:

```
public class AddingComposer {
    [Import("RuContract", RequiredCreationPolicy = CreationPolicy.Shared)]
    public ILanguage Ru1 { get; set; }

    [Import("RuContract", RequiredCreationPolicy = CreationPolicy.Shared)]
    public ILanguage Ru2 { get; set; }

    [Import("EnContract", RequiredCreationPolicy = CreationPolicy.NonShared)]
```

```
        public ILanguage En1 { get; set; }

        [Import("EnContract", RequiredCreationPolicy = CreationPolicy.NonShared)]
        public ILanguage En2 { get; set; }
```

...

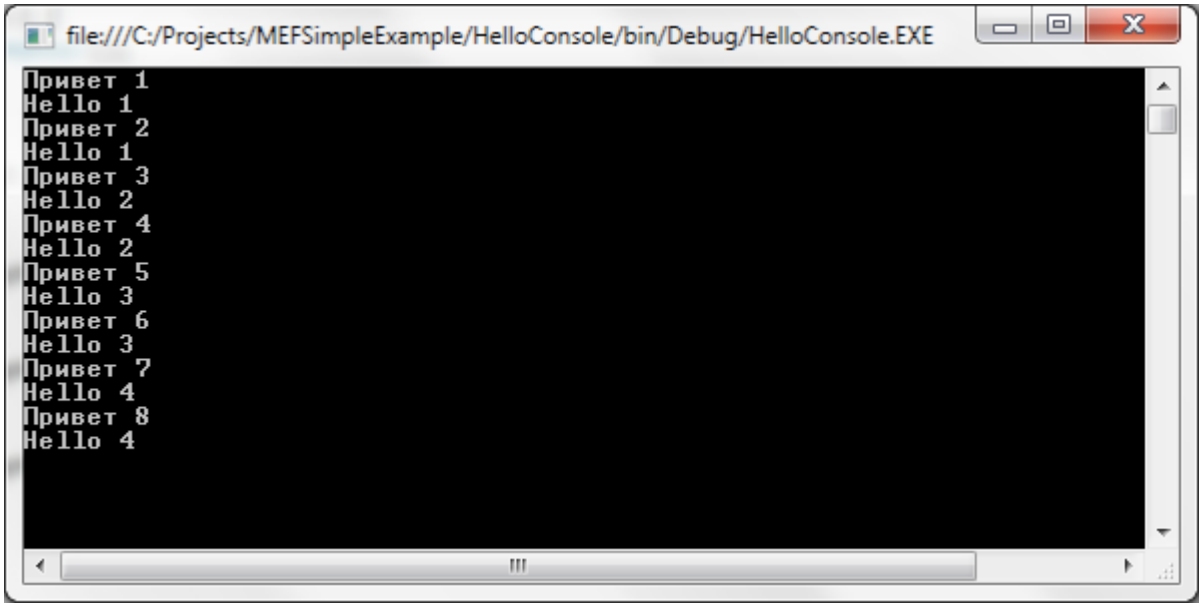
}

Получается, что разные свойства импортируют один и тот же контракт и с помощью политики создания мы указали, каким образом создавать экспортируемый элемент. По идее Ru1 и Ru2 должны обращаться к одному и тому же классу совместно. En1 и En2 получили по индивидуальному экземпляру дополнения.

Проверим на практике:

```
var addingComposer = new AddingComposer();
addingComposer.Compose();

for (var i = 0; i < 4; i++) {
    var hello = addingComposer.Ru1.SayHello();
    Console.WriteLine(hello);

    hello = addingComposer.En1.SayHello();
    Console.WriteLine(hello);

    hello = addingComposer.Ru2.SayHello();
    Console.WriteLine(hello);

    hello = addingComposer.En2.SayHello();
    Console.WriteLine(hello);
}
```

Получим вполне ожидаемый результат:



Наследование дополнений

Как вы поняли из подзаголовка MEF поддерживает наследование. Наследование возможно как для импорта, так и для экспорта.

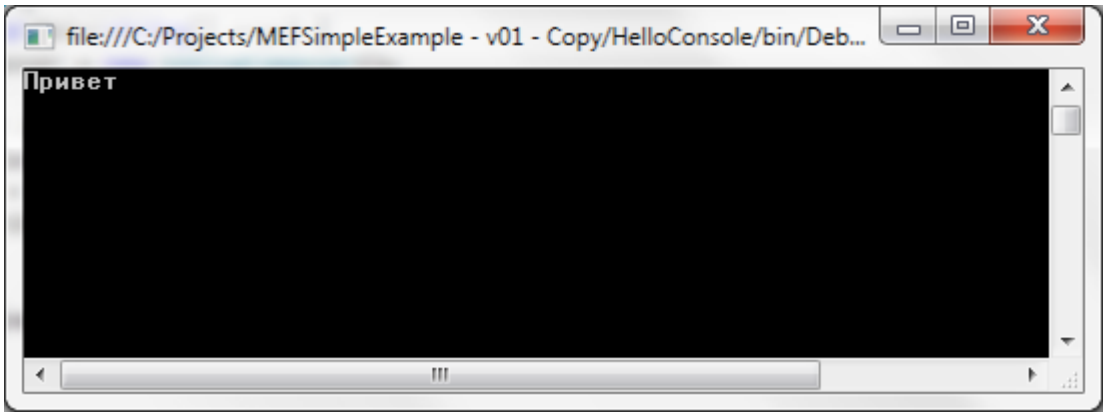
Наследование по аттрибуту **Import** всегда осуществляется автоматически, для наследования по аттрибуту **Export** надо это указать явно. Экспорт наследования на уровне полей класса не поддерживается.

Теперь немного поподробнее рассмотрим предыдущий абзац.

Итак, наледование по атрибуту экспорт. Пусть у нас есть такие классы:

```
namespace RussianHello {
    [Export(typeof (ILanguage))]
    public class RuHello : ILanguage {
        public string SayHello() {
            return "Привет";
        }
    }

    public class RuHello2 : RuHello {

    }
}
```

Класс RuHello2 не будет найден контейнером и не будет импортирован в программу. Для проверки в основной программе используем коллекцию с аттрибутом **ImportMany** и вызовем метод SayHello для каждого найденного класса. Получаем:

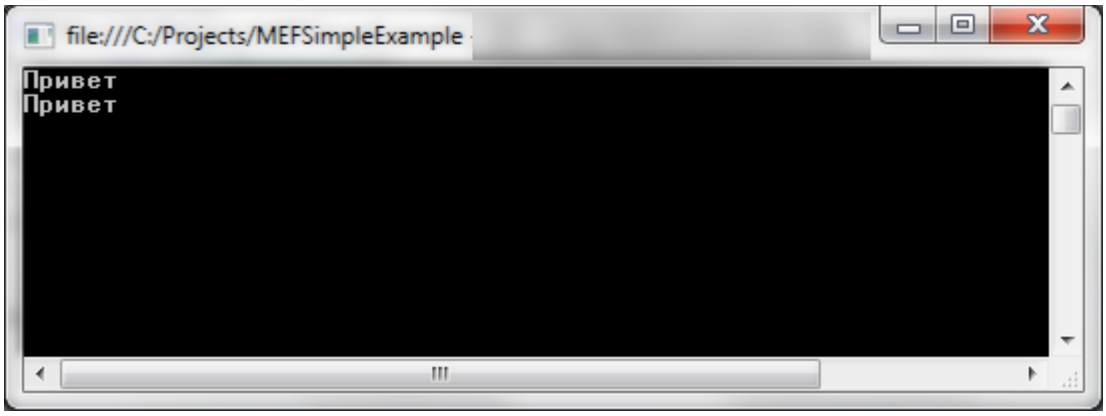


Т.е. экспорта класса RuHello2 не произошло.

Теперь заменим аттрибут **Export** на **ExportInheritance** .

```
namespace RussianHello {
    [InheritedExport(typeof(ILanguage))]
    public class RuHello : ILanguage {
        public string SayHello() {
            return "Привет";
        }
    }

    public class RuHello2 : RuHello {

    }
}
```

Теперь можно увидеть, что нашлись оба класса.



Наследование полей классов не осуществляется. Т.е.

```
namespace RussianHello {
    [InheritedExport(typeof(ILanguage))]
    public class RuHello : ILanguage {

        [Export(typeof (ILanguage))]
        public RuHello3 Hello = new RuHello3();

        public string SayHello() {
            return "Привет";
        }
    }

    public class RuHello2 : RuHello {}

    public class RuHello3 : ILanguage {
        public string SayHello() {
            return "Привет";
        }
    }
}
```

Покажет только 3 (три) раза слово «Привет».

Метадата при экспорте тоже наследуется. Но если в наследуемом классе переопределить хотя бы одно поле метадаты, то только оно и останется.

Классы и поля с аттрибутом **Import** всегда наследуются.

Рекомпозиция

С помощью рекомпозиции можно обновлять коллекции и отдельные элементы помеченные для импорта во время выполнения приложения. Можно назвать это даже онлайн импортом. Например можно отслеживать папку с плагинами и при появлении новых файлов переинициализировать каталог MEF с дополнениями. Очень удобно может быть.

Для того, чтобы применить данную возможность необходимо указать всего лишь один параметр в атрибуте импорта.

AllowRecomposition = true

```
[Import(AllowRecomposition = true)]
public ILanguage Language { get; set; }
```

В картинках сложно показать пример, весь смысл в динамичности процесса.

Конструкторы

В MEF по умолчанию считается, что у экспортируемого класса должен быть конструктор без параметров. Но в жизни, часто надо конструировать объекты на основе других или же надо передавать какие-то управляющие конструкции. Так же у класса может быть несколько конструкторов и надо указать, какой же использовать для создания объекта при импорте.

Для того, чтобы однозначно указать какой конструктор использовать, воспользуемся аттрибутом **ImportingConstructor**. Если аттрибут использовать более одного раза в классе, MEF выдаст ошибку во время исполнения программы.

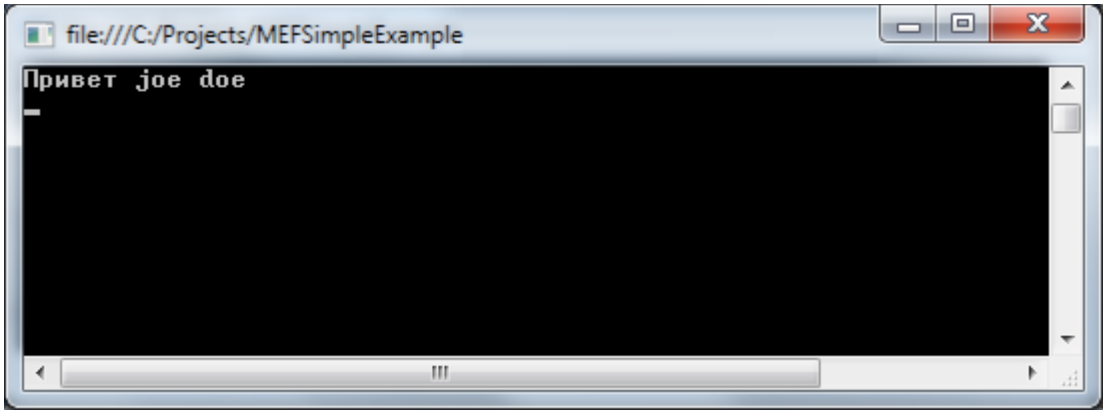
Итак, пусть у нас есть все тот же наш класс RuHello, только теперь он будет принимать в конструктор класс описывающий пользователя.

```
[Export(typeof(ILanguage))]
public class RuHello : ILanguage {
    private readonly Name name;

    [ImportingConstructor]
    public RuHello([Import] Name name) {
        this.name = name;
    }

    public string SayHello() {
        return "Привет " + name.Val;
    }
}

[Export]
public class Name {
    public string Val { get; set; }

    public Name() {
        Val = "joe doe";
    }
}
```

Класс **Name** помечается на экспорт, а в конструкторе класса RuHello он помечается для импорта. На самом деле аттрибут можно опустить в данном случае и все будет работать. Но для более сложных случаев, можно будет явно указать что импортировать.

Самое главное, конструктор помечен аттрибутом **ImportingConstructor**. MEF способен разрешать довольно длинные зависимости классов и корректно их обрабатывать.

Вывод на консоль будет таким:



И еще немного аттрибутов

Скрытие обнаружения

Можно скрыть класс от обнаружения с помощью аттрибута **PartNotDiscoverable**. Это можно использовать при наследовании экспорта, когда базовый класс является абстрактным или же не желателен для использования как есть, и помечен для распространения экспорта для наследников.

Опциональность импорта

В поведении по умолчанию, если поле для импорта не может быть заполнено, то приложение падает с сообщением о том, что не удалось найти подходящий тип и композиция не может быть завершена. Это не очень хорошо, так как можно было бы просто оставить поле значением по умолчанию, а в программе обрабатывать уже эту ситуацию.

И так можно сделать с помощью параметра AllowDefault. Он говорит MEF, что если ничего не найдено для импорта, то можно вставить значение по умолчанию, для классов это будет null.

```
[Import(typeof(XxHello), AllowDefault = true)]
```

На этом все.

Hard'n'Heavy!

Violet Tape