Визитка

АНДРЕЙ УШАКОВ, *НПО «Сапфир», ведущий разработчик/архитектор, std_string@mail.ru*

Создание контейнера IoC под себя

С темой IoC-контейнеров связано достаточно много «черной магии». На самом деле все очень просто; мы покажем это на примере создания своего IoC-контейнера

При переходе в компанию НПО «Сапфир» [1] мне пришлось столкнуться с большим объемом унаследованного кода (написанного на языке C#), при работе с которым возник ряд проблем. И одной из таких проблем была достаточно нетривиальная и запутанная инициализация приложения с использованием IoC-контейнеров из библиотеки Castle Windsor.

По ходу устранения разнообразных проблем с инициализацией стало понятно, что необходимо полностью отказаться от использования библиотеки Castle Windsor. В результате поисков замены оказалось, что нет библиотек, устраивающих требованиям (по крайней мере мне не удалось найти). И тогда было принято решение написать свой собственный IoC-контейнер. Хочу рассказать, как у меня это получилось.

Следует отдельно сказать: все, о чем пойдет речь в статье, относится к языку C# версии 3.0 и выше (и платформе .NET Framework версии 3.5 и выше).

Что такое IoC?

Прежде чем начинать что-либо делать, стоит разобраться с предметной областью; давайте и мы поступим точно так же.

Первый вопрос, который стоит обсудить: что такое инверсия управления (inversion of control, IoC)? Обычно приложение состоит из небольших кирпичиков кода, взаимодействующих друг с другом. В объектно-ориентированных языках программирования эти кирпичики кода называются классами (в дальнейшем мы будем говорить в терминах классов и объектов). Вполне логично, что при этом одни классы зависят от других классов (как на уровне самих классов, так и на уровне их экземпляров – объектов). Часто эта зависимость выражается следующим образом.

Допустим, что у нас есть два класса A и B. Объекты класса A зависят от объектов класса B следующим образом: при создании объекта класса A создается объект класса B, сохраняется в одном из полей объекта класса A и в дальнейшем используется объектом класса A для выполнения своей работы.

При таком подходе получается, что классы A и B являются сильно связанными классами. Действительно, класс B реализует некоторую функциональность и о конкретной реализации этой функциональности знает класс A; если мы захотим изменить конкретную реализацию, используемую классом A, то нам необходимо будем изменять сам класс A. При написании тестов на класс A мы будет вынуждены тестировать при этом и класс B, что сильно усложняет написание тестов. Поэтому обычно связанность между классами снижают (бывают ситуации, когда этого делать не нужно, например, в случае вспомогательных классов).

Для этого поступают следующим образом: вводят некоторый интерфейс I (или абстрактный базовый класс), одной из реализаций которого будет класс B, после чего заменяют зависимость класса A от класса B на зависимость от интерфейса I. При такой замене класс A уже не может содержать код по созданию класса B, поэтому экземпляр реализации интерфейса I передается при создании объекта класса A в качестве одного из параметров (например, конструктора).

Такой подход к организации зависимостей между классами и называется инверсией управления. Процесс передачи экземпляра реализации интерфейса I при создании объектов класса A называется внедрением зависимостей (dependency injection).

И, наконец, контейнеры IoC – это компоновщики, позволяющие централизовать и автоматизировать создание компонентов (объектов) с учетом внедрения зависимостей.

Существующие IoC-контейнеры: чем не устраивают?

Возникает вопрос: возможно, уже есть готовые библиотеки IoC-контейнеров для использования в .NET Framework и нет смысла создавать что-то свое? Да, такие библиотеки есть и не мало: Castle Windsor, Spring.NET и другие. Но у большинства таких библиотек есть один большой минус (это мнение автора – вы с ним можете не соглашаться): их работа основана на использовании метаданных и рефлексии (reflection).

Давайте посмотрим на небольшой пример, чтобы понять, в чем здесь проблема (в этом примере используется IoC-контейнер из библиотеки Castle Windsor).

Предположим, что у нас есть следующие определения интерфейсов:

```
public interface ISomeInnerService
{
        ...
}
public interface ISomeOtherInnerService
{
        ...
}
public interface ISomeService
{
        ...
}
```

Предположим, что у нас есть следующие реализации приведенных выше интерфейсов:

```
public class SomeInnerService : ISomeInnerService
{
    ...
}
public class SomeOtherInnerService : ISomeOtherInnerService
{
    ...
}
public class SomeService : ISomeService
{
    public SomeService(ISomeInnerService inner)
    {
        ...
    }
    ...
}
public class SomeServiceOther : ISomeService
{
    public SomeServiceOther(ISomeOtherInnerService inner)
    {
    }
    ...
}
```

Пусть мы сконфигурировали IoC-контейнер следующим образом:

```
IWindsorContainer container = new WindsorContainer();
container.Register(Component.For<ISomeService>(). ↵
        ImplementedBy<SomeService>());
container.Register(Component.For<ISomeInnerService>(). ↵
        ImplementedBy<SomeInnerService>());
```

Что произойдет при выполнении следующей строки кода:

```
ISomeService service = container.Resolve<ISomeService>();
```

Очевидно, что локальная переменная service будет содержать экземпляр реализации интерфейса ISomeService (это будет экземпляр класса SomeService). Предположим, что мы решили использовать другую реализацию интерфейса ISomeService и изменили конфигурацию IoC-контейнера следующим образом:

```
IWindsorContainer container = new WindsorContainer();
container.Register(Component.For<ISomeService>(). ↵
        ImplementedBy<SomeServiceOther>());
container.Register(Component.For<ISomeInnerService>(). ↵
        ImplementedBy<SomeInnerService>());
```

Что произойдет при таком конфигурировании IoC-контейнера? Все будет в порядке, несмотря на то что в конструктор класса SomeInnerService необходимо передавать экземпляр реализации интерфейса ISomeOtherInnerService. Ошибка возникнет только при попытке получить экземпляр класса SomeServiceOther, т.е. при выполнении следующей строки кода:

```
ISomeService service = container.Resolve<ISomeService>();
```

Эта ошибка будет ошибкой времени выполнения, а не времени компиляции, что наиболее неприятно, т.к. без использования контейнеров IoC мы бы получили ошибку времени компиляции. И это не единственная проблема, связанная с рефлексией, есть и другие.

Помимо проблем, связанных с использованием рефлексии (reflection), контейнеры IoC из библиотеки Castle Windsor излишне усложнены и содержат возможности, без которых вполне можно обойтись (это опять же мнение автора – вы с ним можете не соглашаться). Но мы не будем больше останавливаться на этой библиотеке (из-за ограниченности объема статьи) и пойдем дальше.

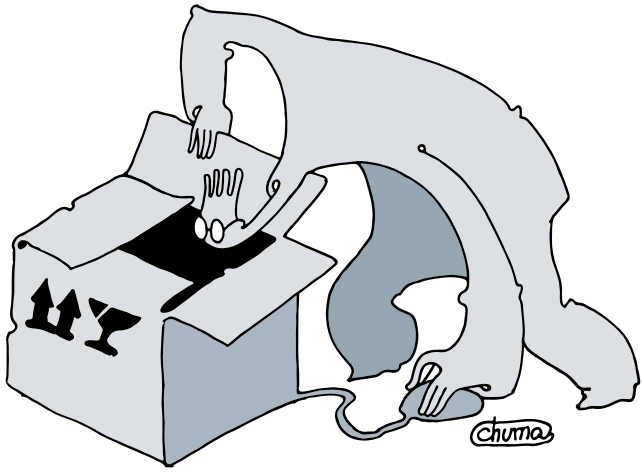
Требования к нашим контейнерам

Пришла пора заняться созданием своего контейнера IoC. В качестве первого шага на пути написания своего IoC-контейнера давайте определим, что же мы от этого контейнера ожидаем (составим своего рода мини-ТЗ):

- Создаваемый нами контейнер должен быть контейнером IoC. Другими словами, создаваемый нами контейнер должен поддерживать создание объектов с учетом зависимостей, а также поддерживать возможность конфигурирования во время выполнения.
- Определение создаваемых объектов (конфигурирование IoC-контейнера) должно проходить проверку во время компиляции (чтобы не возникало проблем, как в примере с Castle Winsdor).
- При определении создаваемых объектов у нас должна быть возможность указывать их время создания и жизни. Под временем создания и жизни мы понимаем следующее: будет ли IoC-контейнер использовать объект, созданный заранее, должен ли объект быть создан при первом обращении или же при каждом обращении (к IoC-контейнеру) необходимо создавать новый объект.
- В качестве ключа для доступа к объектам в IoC-контейнере должна быть возможность использовать объект любого типа. При этом необходимо выделить случаи, когда в качестве ключа используются строка, тип объекта в контейнере или же пара строка – тип. В качестве типа объекта в контейнере (для использования его в качестве ключа) может быть использован любой тип, к которому можно привести объект в контейнере (с помощью операции явного приведения типа).

Проверка во время компиляции

Наиболее сложный вопрос при реализации нашего IoC-контейнера – это реализация пункта о проверке определения создаваемых объектов во время компиляции. Первое, что приходит в голову, – это использовать шаблон проектирования «абстрактная фабрика» (с точки зрения ООП



Контейнеры IoC — компоновщики, позволяющие централизовать и автоматизировать создание компонентов (объектов)

это наиболее правильный путь). Применение шаблона проектирования «абстрактная фабрика» приводит к тому, что появляются специальные объекты фабрики, которые отвечают за создание требуемого объекта. В нашем случае это будет выглядеть примерно следующим образом (по аналогии с примером для контейнера IoC из библиотеки Castle Winsdor):

```
public class SomeService
{
    public SomeService(ISomeInnerService inner)
    {
    }
}
public interface IFactory
{
    Object Create(IServiceContainer container);
}
public class SomeServiceFactory : IFactory
{
    public Object Create(IServiceContainer cont)
    {
        return new SomeService(cont.Resolve ↵
                <ISomeInnerService>());
    }
}
```

Здесь у нас объявлен класс SomeService, объекты которого мы намереваемся хранить в нашем IoC-контейнере. Конструктор объектов класса SomeService зависит от параметра с типом ISomeInnerService. Это означает, что при создании экземпляра SomeService класса нам необходимо в его конструктор передать объект, реализующий тип ISomeInnerService (разрешить зависимости). Интерфейс IFactory определяет поведение шаблона проектирования «абстрактная фабрика»; его реализацией является класс SomeServiceFactory, служащий для разрешения зависимостей и создания экземпляра класса SomeService. Хорошо видно, что зависимости при создании экземпляра класса SomeService ищутся в нашем контейнере IoC.

Какие минусы есть у такого подхода? Очевидно, что для большинства классов, экземпляры которых мы захотим положить в IoC-контейнер, нам придется создавать соответствующие фабрики. В большинстве случаев эти фабрики будут тривиальны: мы находим в контейнере IoC необходимые объекты, от которых зависит создаваемый объект, после чего создаем его.

Очевидно, что если бы язык C# реализовывал только объектно-ориентированный подход к созданию приложений, то других вариантов конфигурирования нашего контейнера IoC у нас не было бы. К нашему счастью, в языке C# начиная с версии 3.0 (т.е. уже достаточно давно) есть поддержка и некоторых элементов функционального подхода к созданию приложений, в частности, есть поддержка лямбда-выражений. Лямбда-выражение – это возможность объявления анонимной функции в месте ее использования. С точки зрения нашей задачи создания своего IoC-контейнера мы можем использовать лямбда-выражения для конфигурирования контейнера с проверкой во время компиляции (без создания фабрики практически для каждого класса, экземпляры которого мы хотим хранить в нашем IoC-контейнере). Выглядеть это будет следующим образом (на примере конфигурирования для экземпляров класса SomeService):

```
container.Add<SomeService>(cont => ↵
        new SomeService(cont.Resolve<ISomeInnerService>());
```

Здесь мы конфигурируем наш IoC-контейнер и сохраняем лямбда-функцию, которая будет вызвана при создании экземпляра класса SomeService. При создании экземпляра класса SomeService будут разрешены все необходимые зависимости; причем конфигурирование нашего IoC-контейнера проходит проверку времени компиляции. Другими словами, подход с использованием лямбда-функций полностью эквивалентен подходу с использованием шаблона проектирования «абстрактная фабрика».

Управление временем жизни объектов

Прежде чем перейти к непосредственной реализации нашего IoC-контейнера, давайте поговорим об управлении временем создания и жизни наших объектов. У нас со-

гласно требованиям должны быть реализованы следующие возможности: использование уже созданного объекта в контейнере IoC, создание объекта по требованию (один раз) и создание каждый раз нового объекта при обращении к контейнеру IoC. Есть несколько подходов для реализации этого: например, объявить несколько разных методов AddXXX в интерфейсе нашего контейнера IoC, отражающих ту или иную стратегию создания и жизни объектов. Наиболее гибким будет подход, основанный на шаблоне проектирования «стратегия»: в этом случае при конфигурации нашего IoC-контейнера мы задаем промежуточный объект, управляющий стратегией создания и жизни искомого объекта. Так как согласно требованиям у нас должна быть реализована поддержка трех возможных вариантов времени создания и жизни объектов, то и конкретных реализаций шаблона проектирования «стратегия» тоже будет три.

А теперь давайте перейдем непосредственно к реализации нашего IoC-контейнера. И начнем мы как раз со стратегий управления временем создания и жизни объектов. Все классы, реализующие конкретную стратегию, должны иметь следующий интерфейс (другими словами, должны его реализовывать):

```
public interface IContainerEntry
{
        Object GetValue(IServiceContainer container);
}
```

Здесь IServiceContainer – это интерфейс нашего IoC-контейнера, который мы рассмотрим чуть позже. Этот интерфейс необходим, т.к. в некоторых случаях нам придется создавать объекты и соответственно разрешать зависимости. Первый возможный вариант стратегии управления временем создания и жизни объекта – это использовать уже готовый объект. Реализация этой стратегии тривиальна:

```
public class SimpleContainerEntry : IContainerEntry
{
        public SimpleContainerEntry(Object value)
        {
                _value = value;
        }
        public Object GetValue(IServiceContainer container)
        {
                return _value;
        }
        private readonly Object _value;
}
```

Чуть более сложный случай – это создание каждый раз нового объекта при обращении к контейнеру. Данное поведение реализуется следующим образом:

```
public class GeneratorContainerEntry : IContainerEntry
{
    public GeneratorContainerEntry(Func<IServiceContainer, ↵
        Object> generator)
    {
        _generator = generator;
    }
    public Object GetValue(IServiceContainer container)
    {
        return _generator(container);
    }
    private readonly Func<IServiceContainer, Object> ↵
        _generator;
}
```

В этой реализации мы используем поле типа Func<IServiceContainer, Object>, которое и содержит лямбда-выражение для конструирования нужного нам объекта с разрешением всех зависимостей. И, наконец, наиболее сложный случай – это создание объекта один раз при первом обращении. Выглядит это поведение следующим образом:

```
public class LazyContainerEntry : IContainerEntry
{
    public LazyContainerEntry(Func<IServiceContainer, ↵
        Object> initializer)
    {
        _initializer = initializer;
        _initialized = false;
        _value = null;
    }
    public Object GetValue(IServiceContainer container)
    {
        if (!_initialized)
        {
            _value = _initializer(container);
            _initialized = true;
        }
        return _value;
    }
    private readonly Func<IServiceContainer, ↵
        Object> _initializer;
    private Boolean _initialized;
    private Object _value;
}
```

Здесь мы храним лямбда-выражение для конструирования нужного нам объекта с разрешением всех зависимостей, сам объект, а также флаг, хранящий статус этого объекта – создан ли этот объект или еще нет. В общем-то, принцип работы данной стратегии времени создания и жизни также тривиален (хотя и сложнее, чем две предыдущие реализации стратегий управления временем создания и жизни).

А теперь реализация

После реализации стратегий управления временем создания и жизни объектов мы готовы перейти к реализации самого нашего контейнера IoC. Начнем мы с определения интерфейса нашего контейнера. Данный интерфейс имеет следующий вид (этот интерфейс приводится в сокращенном виде из-за ограниченного объема статьи):

```
public interface IServiceContainer
{
    // resolve
    Object Resolve(Object key);
    Object Resolve(String name);
    T Resolve<T>();
    T Resolve<T>(String name);
    // resolve by condition
    IList<Object> Resolve(Func<Object, Boolean> ↵
        keyPredicate);
    IList<T> Resolve<T>(Func<Object, Boolean> ↵
        keyPredicate);
    // has
    Boolean HasComponent(Object key);
    ...
    // add
    void AddComponent(Object key, IContainerEntry entry);
    void AddComponent(String name, IContainerEntry entry);
    void AddComponent<T>(IContainerEntry entry);
    void AddComponent<T>(String name, IContainerEntry entry);
    // remove
    void RemoveComponent(Object key);
```

```
    ...
    // add/remove subcontainers
    void AddSubContainer(IServiceContainer container);
    void RemoveSubContainer(IServiceContainer container);
    // clear
    void Clear();
}
```

Этот интерфейс (а точнее, его реализация) позволяет нам как конфигурировать контейнер, так и обращаться к объектам, которыми этот контейнер управляет. При конфигурировании мы можем добавить объект (а точнее, стратегию, управляющую временем создания и жизни объекта) в контейнер, удалить объект из контейнера, очистить контейнер, а также добавить или удалить вложенный контейнер (вложенные контейнеры позволяют несколько структурировать хранимые объекты). При обращении к объектам мы можем запрашивать интересующий нас объект по ключу, запрашивать несколько объектов по предикату над ключами и проверять, содержит ли контейнер определение интересующего нас объекта. Мы предполагаем, что при запросе несуществующего объекта реализация контейнера будет генерировать некоторое исключение. Видно, что в качестве ключа для доступа к объекту можно использовать объекты любых типов; при этом выделены некоторые методы для работы со строковыми ключами, с ключами, представляющими некий тип (к которому может быть приведен объект, сохраненный в контейнере), а также с ключами в виде пары строка – тип.

После определения интерфейса нашего IoC-контейнера мы готовы перейти и к его реализации в классе ServiceContainer (следует понимать, что мы не приведем полностью реализацию IoC-контейнера в статье). Давайте посмотрим на наиболее интересные детали реализации. Хранение данных в нашем IoC-контейнере будет организовано следующим образом (с помощью объявления следующих полей):

```
private readonly IDictionary<Object, ↵
        IContainerEntry> _containerEntries;
private readonly IList<IServiceContainer> _subContainers;
```

Видно, что у нас есть словарь пар ключ – объект (а точнее, стратегия, управляющая временем создания и жизни объекта) и список дочерних контейнеров. При поиске объектов мы сначала будем их искать в словаре пар ключ – объект, а потом в дочерних контейнерах. Порядок поиска в дочерних контейнерах соответствует порядку прямого обхода двоичного дерева в глубину. Прежде чем перейти к реализации методов нашего IoC-контейнера следует сделать следующее замечание. В коде все параметры у всех открытых методов мы проверяем, чтобы они не были null, все строковые параметры мы дополнительно проверяем, чтобы они не были пустыми строками; но все эти проверки здесь мы приводить не будем.

Начнем мы реализацию с общего метода для добавления нового объекта в контейнер:

```
public void AddComponent(Object key, IContainerEntry entry)
{
    if (HasComponent(key))
        throw new ServiceAlreadyRegisteredException();
    _containerEntries.Add(key, entry);
}
```

Следующий шаг – создание общего метода для удаления объекта из контейнера:

```
public void RemoveComponent(Object key)
{
    if (_containerEntries.Remove(key))
        return;
    foreach (IServiceContainer container in _subContainers)
        if (container.HasComponent(key))
        {
            container.RemoveComponent(key);
            return;
        }
}
```

Предпоследний шаг – создание методов для поиска объекта по ключу и проверки существования:

```
public Object Resolve(Object key)
{
    IContainerEntry entry;
    if (_containerEntries.TryGetValue(key, out entry))
        return entry.GetValue(this);
    foreach (IServiceContainer container in _subContainers)
        if (container.HasComponent(key)) ↵
                return container.Resolve(key);
    throw new ServiceNotFoundException();
}

public Boolean HasComponent(Object key)
{
    if (_containerEntries.ContainsKey(key))
        return true;
    foreach (IServiceContainer container in _subContainers)
        if (container.HasComponent(key)) return true;
    return false;
}
```

И, наконец, последний наш шаг – использование созданного нами IoC-контейнера в деле. Конфигурация этого контейнера выглядит следующим образом (по аналогии с примером для контейнера IoC из библиотеки Castle Winsdor):

```
IServiceContainer container = new ServiceContainer();
container.Add<ISomeService>(new LazyContainerEntry( ↵
        cont => SomeService( ↵
        cont.Resolve<ISomeInnerService >())));
container.Add<ISomeInnerService>( ↵
        new LazyContainerEntry(_ =>new SomeInnerService()));
```

А его использование следующим образом:

```
ISomeService service = container.Resolve<ISomeService>();
```

***

Как видите создание своего IoC-контейнера не такая уж и сложная задача. В данный момент приведенный здесь IoC-контейнер внедрен в несколько приложений и успешно работает.

Исходный код по лицензии GNU GPL этого IoC-контейнера можно скачать по следующей ссылке [2] и использовать в своих проектах. EOF

[1] Сайт компании НПО «Сапфир» – http://www.nposapfir.ru.

[2] Проект с исходным кодом на SourceForge – http://sourceforge.net/projects/simpleiocnet.

Ключевые слова: IoC, Open Source, C#, .NET Framework.