IoC

Dependency Injection

autofac

1) DI and IoC concepts
2) DI configuration
3) CCC/aspects, Interception
4) Autofac
5) AutoMapper (bonus)

DI and IoC (Inversion of Control)

Суть инверсии зависимостей сводится к тому, что класс перекладывает ответственность за создание зависимости или получение ее экземпляра на более высокий уровень, в результате чего, он сам не создает экземпляр конкретного класса, а получает его от более высокоуровневого кода. В результате, мы заменяем композицию агрегацией и перекладываем часть проблем с текущего класса куда-то выше.

Основная суть инверсии зависимостей, как и любого другого принципа проектирования, сводится к борьбе со сложностью и упрощению сопровождения и не является самоцелью.

DI and IoC (Inversion of Control)

Что дает?

- Малое сцепление
- Тестируемость
- Повторное использование

Example

```
class StatsService : IStatsService
{
        public string GetStats()
        {
                var repository = new ItemRepository();
                // …
        }
}

class ForecastService : IForecastService
{
        public string Forecast()
        {
                var statsService = new StatsService();
                var stats = statsService.GetStats();
                // …
        }
}
```

Завязались на реализацию

И здесь тоже

Example –> DI

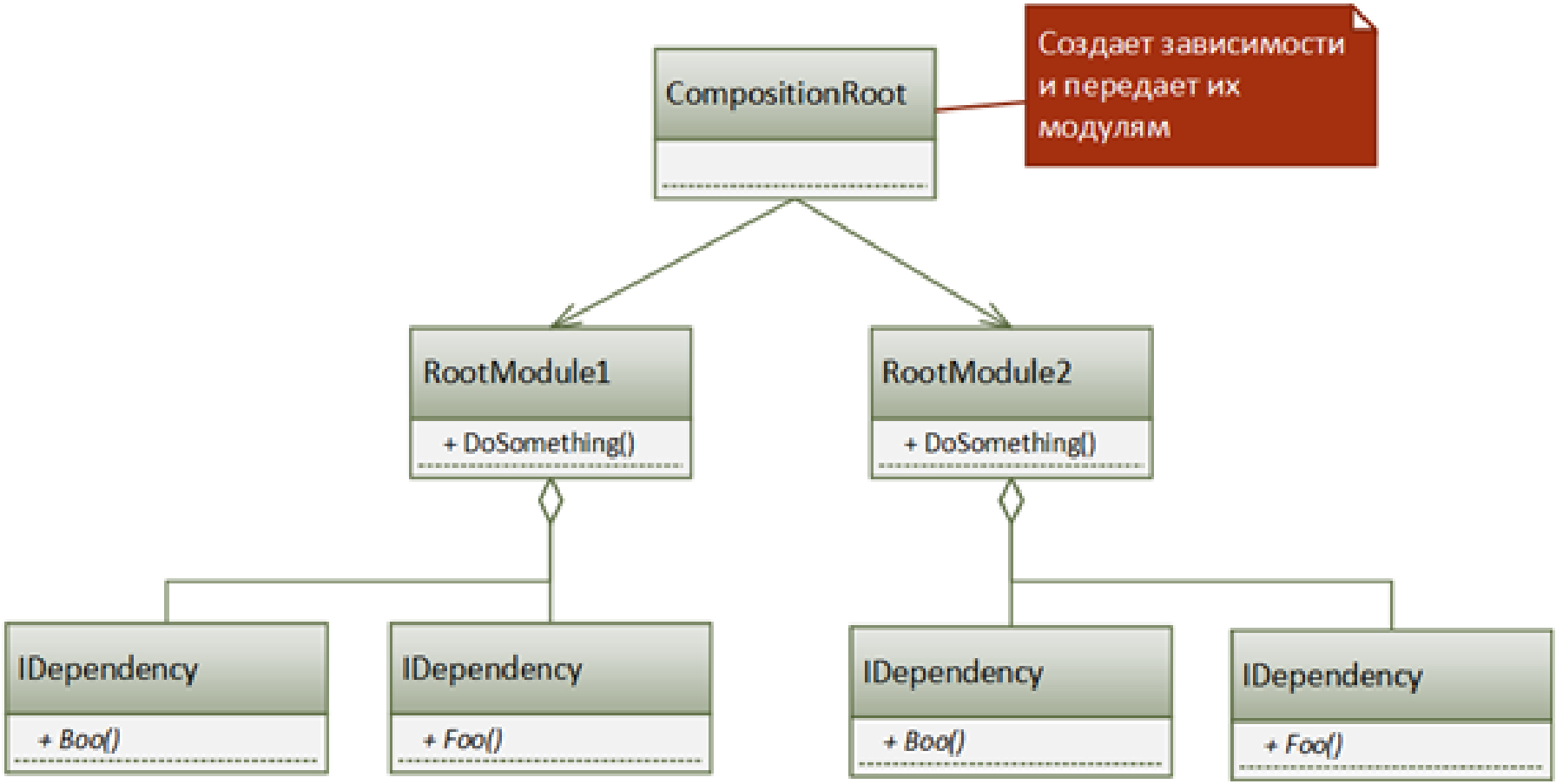
```
class StatsService : IStatsService
{
        public StatsService(IRepository repository)
        {
                _repository = repository;
        }
        public string GetStats()
        {
                var items = _repository.GetAll();
                // …
        }
}

class ForecastService : IForecastService
{
        public ForecastService(IStatsService statsService)
        {
                _statsService = statsService;
        }
        public string Forecast()
        {
                var stats = _statsService.GetStats();
                // …
        }
}
```

Example –> DI

```
void Main()
{
        var forecastService = new ForecastService(
                                        new StatsService(
                                                new ItemRepository()));

        var forecast = forecastService.Forecast();

        Console.WriteLine(forecast);
}
```

Любое приложение содержит точку входа, с которой начинается его исполнение. В идеальном случае, эта точка содержит всего несколько строк кода, которые сводятся к созданию одного (или нескольких) экземпляров самых высокоуровневых классов, представляющих основную логику приложения. Именно здесь нам придется решить, какие абстракции требуются нашим модулям верхнего уровня и именно точка входа приложения является идеальным местом для разрешения всех зависимостей.

Composition Root



Dependencies: Stable vs. Volatile

Стабильные зависимости (stable dependencies) - «абстрагироваться» от которых нет особого смысла, поскольку они доступны «из коробки», являются стандартом «де факто» в вашей команде и их поведение не меняется в зависимости от состояния окружения.

Изменчивые зависимости (volatile dependencies) - реализация (или даже публичный интерфейс) которых может измениться в будущем, либо их поведение может измениться «самостоятельно» без нашего ведома, поэтому их нужно выделить в отдельную абстракцию и ограничить их использование. К такому классу зависимостей относятся третисторонние библиотеки, собственный код, область применения которого хорошо бы ограничить. К этому же типу зависимости относятся зависимости, завязанные на текущее окружение, такие как файлы, сокеты, базы данных и так далее. И хотя интерфейс таких зависимостей едва ли изменится в будущем, их «изменчивость» проявляется в том, что их *поведение* может измениться без изменения пользовательского кода.

Паттерны

- Constructor Injection
- Property Injection
- Method Injection
- Ambient Context

Constructor Injection

```
class EquationSolver
{
        private IRoots _roots;

        public EquationSolver(IRoots roots)
        {
                _roots = roots;
        }
        …
}

…

public void Main()
{
        var solver = new EquationSolver(new PolyRoots());

        var solution = solver.Solve(new [] {1.0, 1, -6});
}
```

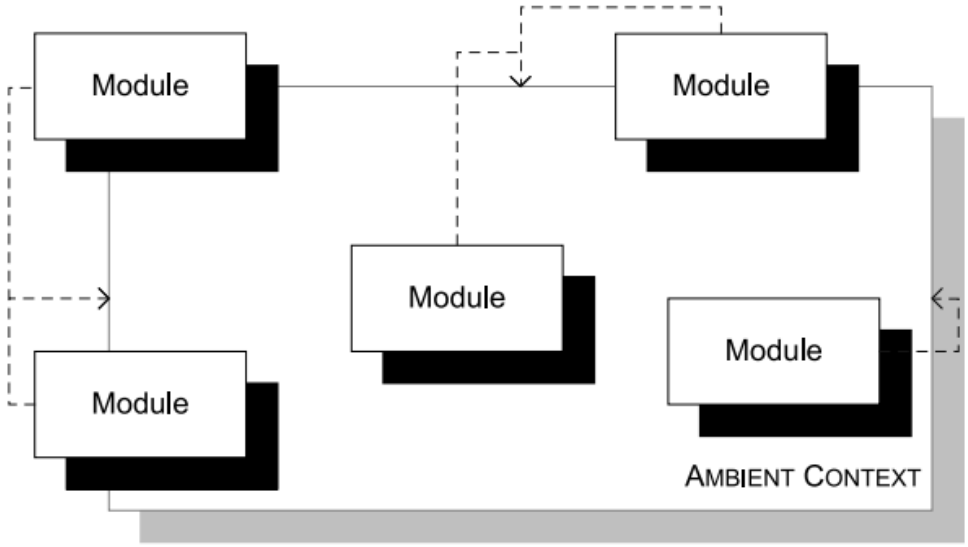
Property Injection

```
class EquationSolver
{
        public IRoots Roots { get; set; }
        …
}

…

public void Main()
{
        var solver = new EquationSolver();
        solver.Roots = new PolyRoots();

        var solution = solver.Solve(new [] {1.0, 1, -6});
}
```

Method Injection

```
class EquationSolver
{
        public double[] Solve(double[] coeffs, IRoots roots)
        {
                // use IRoots functionality
        }
        …
}

…

public void Main()
{
        var solver = new EquationSolver();

        var solution = solver.Solve(new [] {1.0, 1, -6},
                                    new PolyRoots());
}
```

Ambient Context




```
class EquationSolver
{
        public double[] Solve(double[] coeffs)
        {
                var tmp = PolyRoots.Calculate(coeffs);
        }
        …
}
```

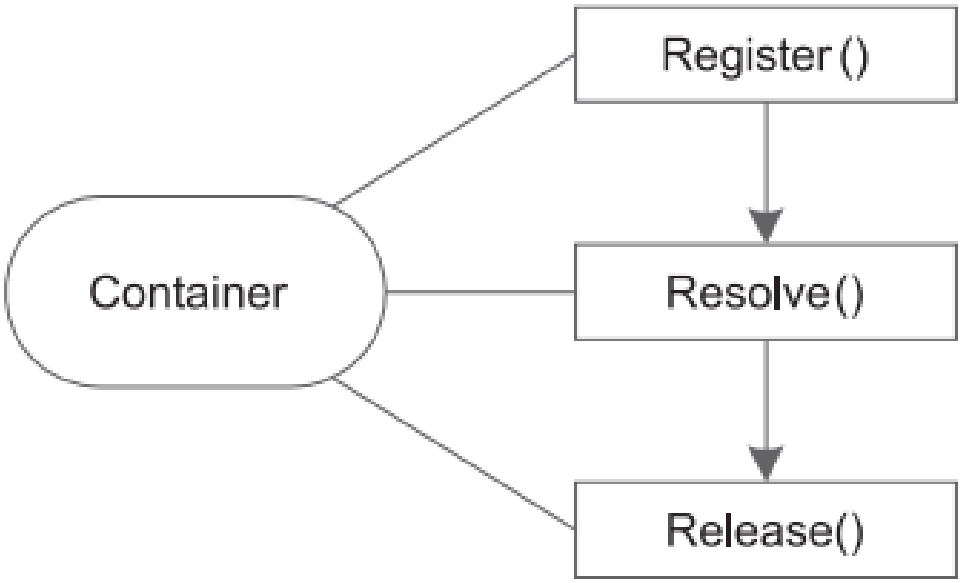
DI Frameworks

Внезапно: если спроектировано все сугубо по лучшим рекомендациям IoC, то можно обойтись и без DI фреймворка.

Суть: в композиционном корне создать объекты всех необходимых реализаций интерфейсов и далее использовать их.

И все же: если есть много вложенных и/или запутанных зависимостей, то фреймворк облегчит кодинг. Плюс дают возможность конфигурирования зависимостей во внешних файлах, что позволит внедрять разные зависимости без перекомпиляции приложения (хотя эта возможность сейчас уже не является ключевой).

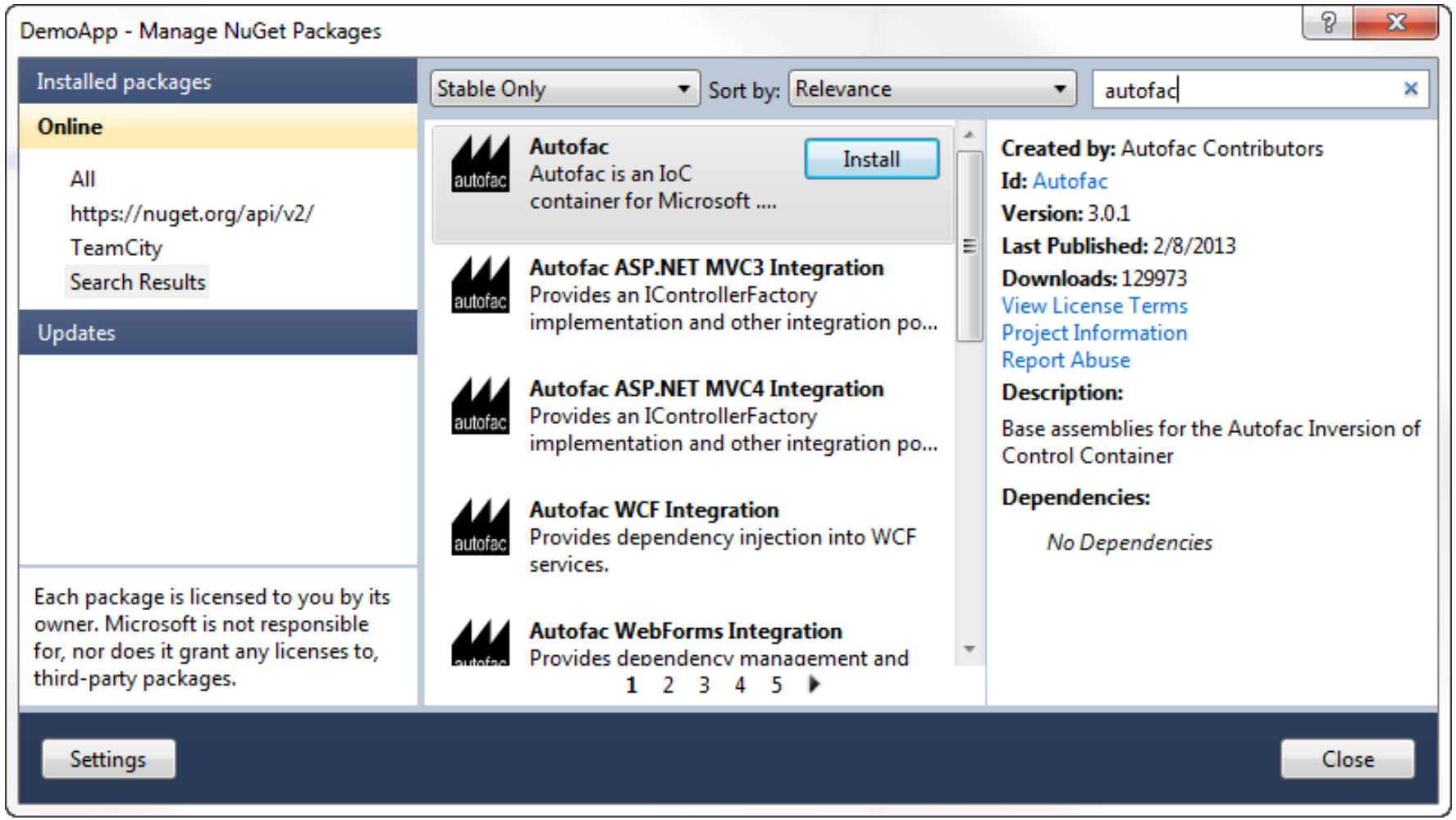
Register / Resolve / Release



Register / Resolve / Release

Phase	What happens in this phase?
Register	Register components with the container. You configure the container by informing it about which classes it can use, how it should map ABSTRACTIONS to concrete types, and, optionally, how certain classes should be wired together.
Resolve	Resolve root components. A single object graph is resolved from a request for a single type.
Release	Release components from the container. All object graphs resolved in the previous phase should be released when they're no longer needed. This signals to the container that it can clean up the object graph, which is particularly important if some of the components are disposable.

Autofac



https://autofac.readthedocs.io/en/latest/getting-started/index.html

```
// This interface helps decouple the concept of
// "writing output" from the Console class. We
// don't really "care" how the Write operation
// happens, just that we can write.
public interface IOutput
{
  void Write(string content);
}
```

```
// This interface decouples the notion of writing
// a date from the actual mechanism that performs
// the writing. Like with IOutput, the process
// is abstracted behind an interface.
public interface IDateWriter
{
  void WriteDate();
}
```

```
// This implementation of the IOutput interface
// is actually how we write to the Console. Technically
// we could also implement IOutput to write to Debug
// or Trace... or anywhere else.
public class ConsoleOutput : IOutput
{
  public void Write(string content)
  {
    Console.WriteLine(content);
  }
}
```

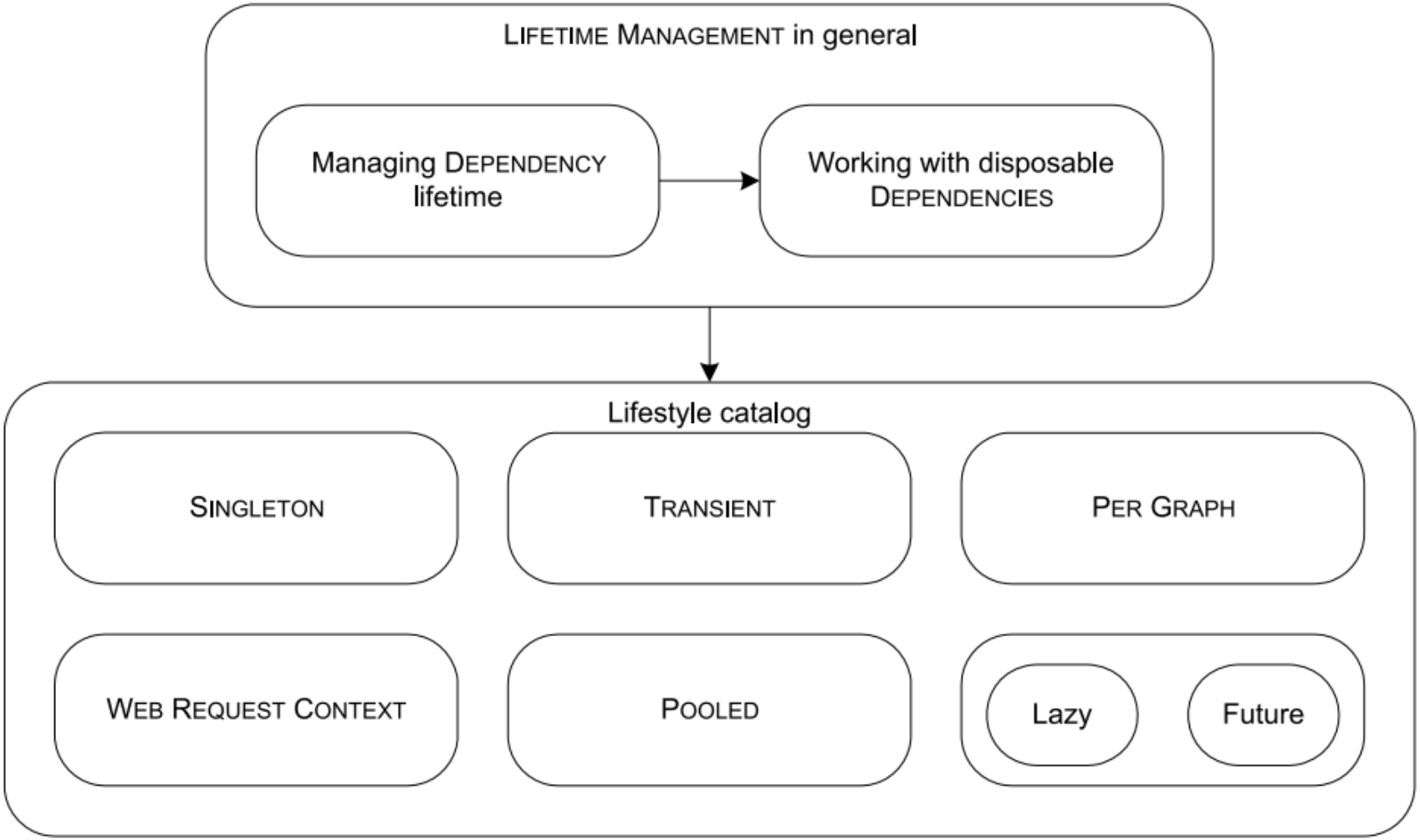
```
// This TodayWriter is where it all comes together.
// Notice it takes a constructor parameter of type
// IOutput - that lets the writer write to anywhere
// based on the implementation. Further, it implements
// WriteDate such that today's date is written out;
// you could have one that writes in a different format
// or a different date.
public class TodayWriter : IDateWriter
{
  private IOutput _output;
  public TodayWriter(IOutput output)
  {
    this._output = output;
  }

  public void WriteDate()
  {
    this._output.Write(DateTime.Today.ToShortDateString());
  }
}
```

https://autofac.readthedocs.io/en/latest/getting-started/index.html

```
public class Program
{
  private static IContainer Container { get; set; }

  static void Main(string[] args)
  {
    var builder = new ContainerBuilder();
    builder.RegisterType<ConsoleOutput>().As<IOutput>();
    builder.RegisterType<TodayWriter>().As<IDateWriter>();
    Container = builder.Build();

    // The WriteDate method is where we'll make use
    // of our dependency injection. We'll define that
    // in a bit.
    WriteDate();
  }

  public static void WriteDate()
  {
    // Create the scope, resolve your IDateWriter,
    // use it, then dispose of the scope.
    using (var scope = Container.BeginLifetimeScope())
    {
      var writer = scope.Resolve<IDateWriter>();
      writer.WriteDate();
    }
  }
}
```

https://autofac.readthedocs.io/en/latest/getting-started/index.html

Object Lifetime



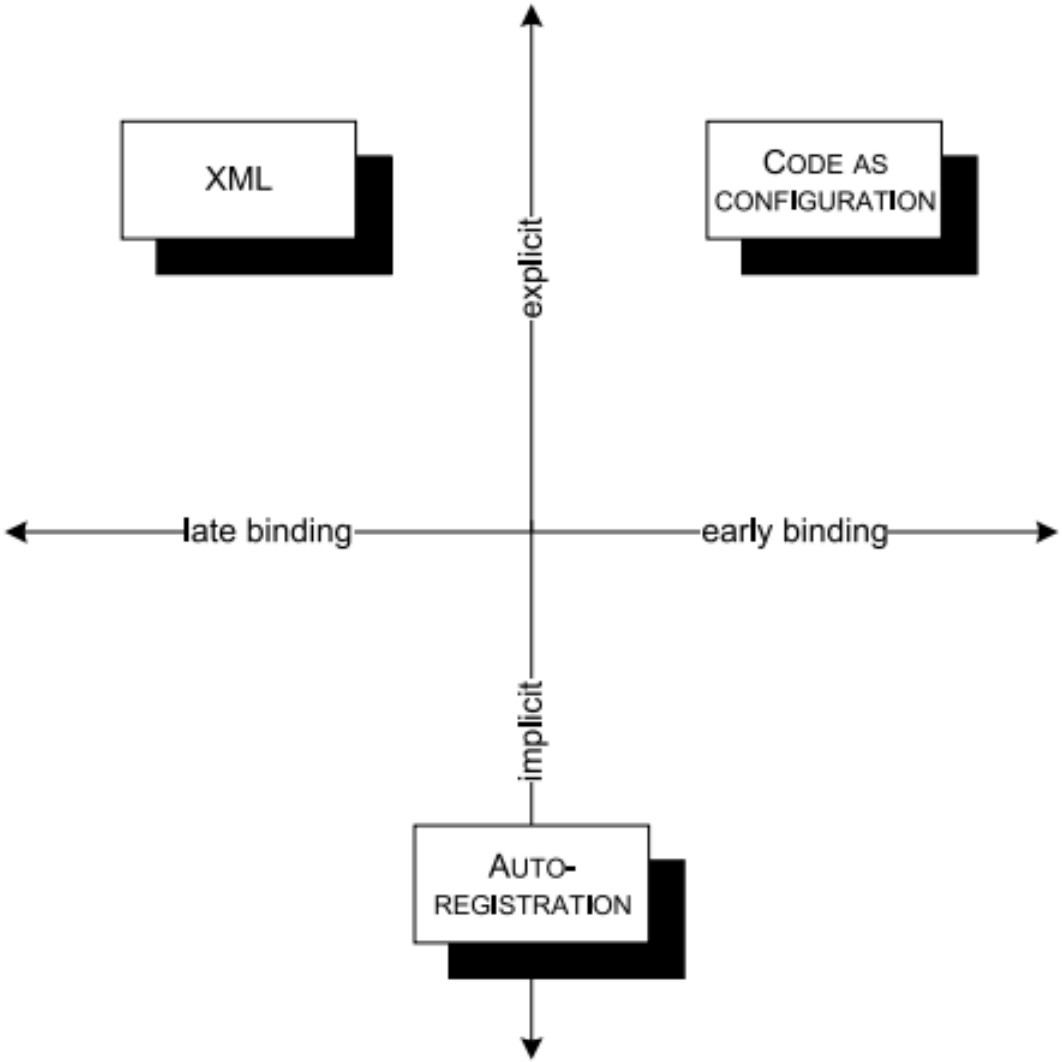
Object Lifetime

```
var builder = new ContainerBuilder();
builder.RegisterType<Worker>().SingleInstance();
```

- Instance Per Dependency
- Single Instance
- Instance Per Lifetime Scope
- Instance Per Matching Lifetime Scope
- Instance Per Request
- Instance Per Owned
- Thread Scope

```
// It's generally not good to resolve things from the
// container directly, but for singleton demo purposes
// we do...
var root = container.Resolve<Worker>();

// We can resolve the worker from any level of nested
// lifetime scope, any number of times.
using(var scope1 = container.BeginLifetimeScope())
{
  for(var i = 0; i < 100; i++)
  {
    var w1 = scope1.Resolve<Worker>();
    using(var scope2 = scope1.BeginLifetimeScope())
    {
      var w2 = scope2.Resolve<Worker>();

      // root, w1, and w2 are always literally the
      // same object instance. It doesn't matter
      // which lifetime scope it's resolved from
      // or how many times.
    }
  }
}
```

Configuration: Code vs. XML



XML/json configuration

```
{
  "defaultAssembly": "Autofac.Example.Calculator",
  "components": [{
    "type": "Autofac.Example.Calculator.Addition.Add, Autofac.Example.Calculator.Addition",
    "services": [{
      "type": "Autofac.Example.Calculator.Api.IOperation"
    }],
    "injectProperties": true
  }, {
    "type": "Autofac.Example.Calculator.Division.Divide, Autofac.Example.Calculator.Division",
    "services": [{
      "type": "Autofac.Example.Calculator.Api.IOperation"
    }],
    "parameters": {
      "places": 4
    }
  }]
}
```

XML/json configuration

```
// Add the configuration to the ConfigurationBuilder.
var config = new ConfigurationBuilder();
// config.AddJsonFile comes from Microsoft.Extensions.Configuration.Json
// config.AddXmlFile comes from Microsoft.Extensions.Configuration.Xml
config.AddJsonFile("autofac.json");

// Register the ConfigurationModule with Autofac.
var module = new ConfigurationModule(config.Build());
var builder = new ContainerBuilder();
builder.RegisterModule(module);
```

Configuration

Style	Description	Advantages	Disadvantages
XML	Configuration settings (often in .config files) specify the mappings.	Supports replacement without recompilation High degree of control	No compile-time checks Verbose
CODE AS CONFIGURATION	Code explicitly determines mappings.	Compile-time checks High degree of control	No support for replacement without recompilation
AUTO-REGISTRATION	Rules are used to locate suitable components and build the mappings.	Supports replacement without recompilation Less effort required Helps enforce conventions to make a code base more consistent	Partial compile-time checks Less control

Auto-registration

Method	Description	Example

AsImplementedInterfaces()	Register the type as providing all of its public interfaces as services (excluding IDisposable).	`Builder`
		`.RegisterAssemblyTypes(asm)`
		`.Where(t => t.Name.EndsWith("Repository"))`
		`.AsImplementedInterfaces();`

AsClosedTypesOf(open)	Register types that are assignable to a closed instance of the open generic type.	`Builder`
		`.RegisterAssemblyTypes(asm)`
		`.AsClosedTypesOf(typeof(IRepository<>));`

AsSelf()	The default: register types as themselves - useful when also overriding the default with another service specification.	`Builder`
		`.RegisterAssemblyTypes(asm)`
		`.AsImplementedInterfaces()`
		`.AsSelf();`

Cross Cutting Concerns / Interception

Создание интерсептора:

```
public class CallLogger : IInterceptor
{
  TextWriter _output;

  public CallLogger(TextWriter output)
  {
    _output = output;
  }

  public void Intercept(IInvocation invocation)
  {
    _output.Write("Calling method {0} with parameters {1}... ",
      invocation.Method.Name,
      string.Join(", ", invocation.Arguments.Select(a => (a ?? "").ToString()).ToArray()));

    invocation.Proceed();

    _output.WriteLine("Done: result was {0}.", invocation.ReturnValue);
  }
}
```

Cross Cutting Concerns / Interception

Регистрация интерсептора:

```
// Named registration
builder.Register(c => new CallLogger(Console.Out))
       .Named<IInterceptor>("log-calls");

// Typed registration
builder.Register(c => new CallLogger(Console.Out));
```

Регистрация типа, в котором применяется сквозная функциональность:

```
var builder = new ContainerBuilder();
builder.RegisterType<SomeType>()
       .EnableClassInterceptors()
       .InterceptedBy(typeof(CallLogger));
builder.Register(c => new CallLogger(Console.Out));
```

Анти-паттерны

- Control Freak
- Bastard Injection
- Constrained Construction
- Service Locator

Control Freak

Суть анти-паттерна - зависимости все равно создаются явно

```
private readonly ProductRepository repository;

public ProductService()
{
    string connectionString =
        ConfigurationManager.ConnectionStrings
        ["CommerceObjectContext"].ConnectionString;

    this.repository =
        new SqlProductRepository(connectionString);
}
```

© Mark Seemann, Dependency Injection in .NET

Bastard Injection

Суть анти-паттерна – класс начинает тянуть ненужную зависимость.

```
public ProductService()
    : this(ProductService.CreateDefaultRepository())
{
}

public ProductService(ProductRepository repository)
{
    if (repository == null)
    {
        throw new ArgumentNullException("repository");
    }

    this.repository = repository;
}

private static ProductRepository CreateDefaultRepository()
{
    string connectionString =
        ConfigurationManager.ConnectionStrings
        ["CommerceObjectContext"].ConnectionString;

    return new SqlProductRepository(connectionString);
}
```

1 **Default Constructor**

2 **Injection Constructor**

Service Locator

Суть паттерна Сервис Локатор сводится к тому, что вместо создания конкретных объектов («сервисов») напрямую с помощью ключевого слова new, мы будем использовать специальный «фабричный» объект, который будет отвечать за создание, а точнее «нахождение» всех сервисов

```
// Статический "локатор"
public static class ServiceLocator
{
    public static object GetService(Type type) {}

    public static T GetService<T>() {}
}

// Сервис локатор в виде интерфейса
public interface IServiceLocator
{
    T GetService<T>();
}
```

http://sergeyteplyakov.blogspot.com/2013/03/di-service-locator.html

Service Locator

```
class EditEmployeViewModel
{
    private readonly IRepository _repository;
    private readonly ILogger _logger;
    private readonly IMailSender _mailSender;
    private readonly IServiceLocator _locator;

    public EditEmployeViewModel(IServiceLocator locator)
    {
        _locator = locator;
        _repository = locator.GetService<IRepository>();
        _mailSender = locator.GetService<IMailSender>();
        _logger = locator.GetService<ILogger>();
    }
}
```

Основной недостаток –
сходу непонятен контракт класса: что он делает и какие зависимости ему нужны;
приходится просматривать все точки, где происходит Resolve() зависимостей.

http://sergeyteplyakov.blogspot.com/2013/03/di-service-locator.html

Service Locator

Самое страшное в Сервис Локаторе то, что он дает видимость хорошего дизайна. У нас никто не знает о конкретных классах, все завязаны на интерфейсы, все «нормально» тестируется и «расширяется». Но когда вы попробуете использовать ваш код в другом контексте или когда кто-то попробует использовать его повторно, вы с ужасом поймете, что у вас есть дикая «логическая» связанность, о которой вы и не подозревали.

http://sergeyteplyakov.blogspot.com/2013/03/di-service-locator.html

Ну давай, расскажи мне опять про S.O.L.I.D.

- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation
- Dependency Inversion

AutoMapper

```
class Order
{
    public int Id {get;set;}
    public string Name {get;set;}
    public int Total {get;set;}
    public string Comment {get;set;}
}
```

```
class OrderDto
{
    public string Name {get;set;}
    public int Total {get;set;}
    public string Comment {get;set;}
}
```

```
Mapper.Initialize(cfg => { cfg.CreateMap<Order, OrderDto>(); });

var order = new Order { Id = 1, Name = " x" , Total = 1};
var orderDto = Mapper.Map<Order, OrderDto>(order);
```

Example

```
private IContainer InitializeDiContainer()
{
    var connectionString = ConfigurationManager.ConnectionStrings["MusCatDbContext"].ConnectionString;

    var builder = new ContainerBuilder();

    builder.Register(c => new UnitOfWork(connectionString)).As<IUnitOfWork>().SingleInstance();
    builder.RegisterType<PerformerService>().As<IPerformerService>();
    builder.RegisterType<AlbumService>().As<IAlbumService>();
    builder.RegisterType<CountryService>().As<ICountryService>();
    builder.Register(c => new StatsService(connectionString)).As<IStatsService>();
    builder.Register(c => new RandomSongSelector(connectionString)).As<ISongSelector>();
    builder.RegisterType<AudioPlayer>().As<IAudioPlayer>();
    builder.RegisterType<RadioService>().As<IRadioService>();
    builder.RegisterType<Mp3SonglistHelper>().As<ISonglistHelper>();
    builder.RegisterType<LastfmDataLoader>().As<IWebDataLoader>();
    builder.RegisterType<RateCalculator>().As<IRateCalculator>();
    builder.RegisterType<MainViewModel>();
    builder.RegisterType<AlbumPlaybackViewModel>();
    builder.RegisterType<CountriesViewModel>();

    return builder.Build();
}
```

Example

```
/// <summary>
/// Initialize AutoMapper mappings
/// </summary>
private void InitializeMappings()
{
    Mapper.Initialize(cfg =>
    {
        cfg.AddCollectionMappers();

        cfg.CreateMap<Performer, PerformerViewModel>()
           .EqualityComparison((o, ovm) => o.Id == ovm.Id)
           .ForMember(m => m.Albums, opt => opt.Ignore())
           .ReverseMap();

        cfg.CreateMap<Album, AlbumViewModel>()
           .EqualityComparison((o, ovm) => o.Id == ovm.Id)
           .ReverseMap();

        cfg.CreateMap<Country, CountryViewModel>()
           .EqualityComparison((o, ovm) => o.Id == ovm.Id);
    });
}
```