# Design for Unit Testing

DEVELOPMENTOR

DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

## Agenda

- Tight coupling
- Loose coupling with interfaces
- Dependency Injection
- Slicing
- Inversion of Control Containers

## The Problem with Tight Coupling

- Tight coupling makes testing very difficult
  - How do I write a test that doesn’t need the database?
  - new indicates a potential issue

```
public Account(int accountNumber)
{
    var repository = new AccountRepository();

    this.actualBalance =
repository.GetBalance(accountNumber);
}
```

# Statics also Lead to Tight Coupling

- Static members couple types together

```
private int accountNumber;
public Account(decimal initialBalance)
{
    this.accountNumber =
Account.GetNextAccountNumber();
    this.initialBalance = initialBalance;
}
```

## Coding to Abstraction

- Using abstract types allows you to provide different implementations
  - Can vary implementation as required
  - Can provide an implementation that satisfies the test conditions

```
public Account(int accountNumber)
{
    IAccountRepository repository = ???

    this.actualBalance =
repository.GetBalance(accountNumber);
}
```

## A Problem

- Where does the implementation of the abstraction come from?
  - Using new will tightly couple
- Need to "inject" the implementation
  - Allows implementation to vary at runtime
- Technique is called Dependency Injection

## Dependency Injection

- Three different models for Dependency Injection
  - Parameter Injection
  - Property Injection
  - Constructor Injection

## Parameter Injection

- Pass implementation as method parameter
  - Good for dependencies that vary over the lifetime of an object

```
public Credit(decimal amount, IAccountRepository
repository )
{
    repository.CreditBalance(accountNumber, amount);
}
```

## Property Injection

- Assign property with dependency
  - Good for optional dependencies

```
public IAccountRepository Repository {get; set;}

public Account(int accountNumber)
{
    if(Repository != null)
    {
        this.actualBalance = 
Repository.GetBalance(accountNumber);
    }
}
```

## Constructor Injection

- Good for dependencies that are fixed for the lifetime of an object
  - Most common type of dependency injection

```
private IAccountRepository repository;

public Account(int accountNumber,
IAccountRepository repository)
{
    this.repository = repository;
    this.actualBalance =
this.repository.GetBalance(accountNumber);
}
```

## Dealing with Statics

- Interfaces and abstract base classes don’t solve the static problem
  - Cannot override a static implementation
- Two mechanisms for dealing with statics
  - Wrap static in a service
  - Inject a delegate

# Wrapping Statics in an Interface

- Create an interface that models the static behavior
  - Interface can be injected
- Create "real" version of interface that hands off to static

```
public interface IAccountService
{
    int GetNextAccountNumber();
}

public class RealAcccountService :
IAccountService
{
    public int GetNextAccountNumber()
    {
        return Account.GetNextAccountNumber();
    }
}
```

## Injecting a Delegate to Wrap a Static

- A lightweight way to abstract a static is to wrap it in a delegate
  - Means increasing the public “interface” of a class
  - Sometimes called “slicing”

```
public Account(decimal initialBalance)
    :this(initialBalance,
Account.GetNextAccountNumber)
{
}

public Account(decimal initialBalance, Func<int>
accountNumberProvider)
{
    accountNumber = accountNumberProvider();
    this.initialBalance = initialBalance;
}
```

## Another Problem – How to Construct You Application

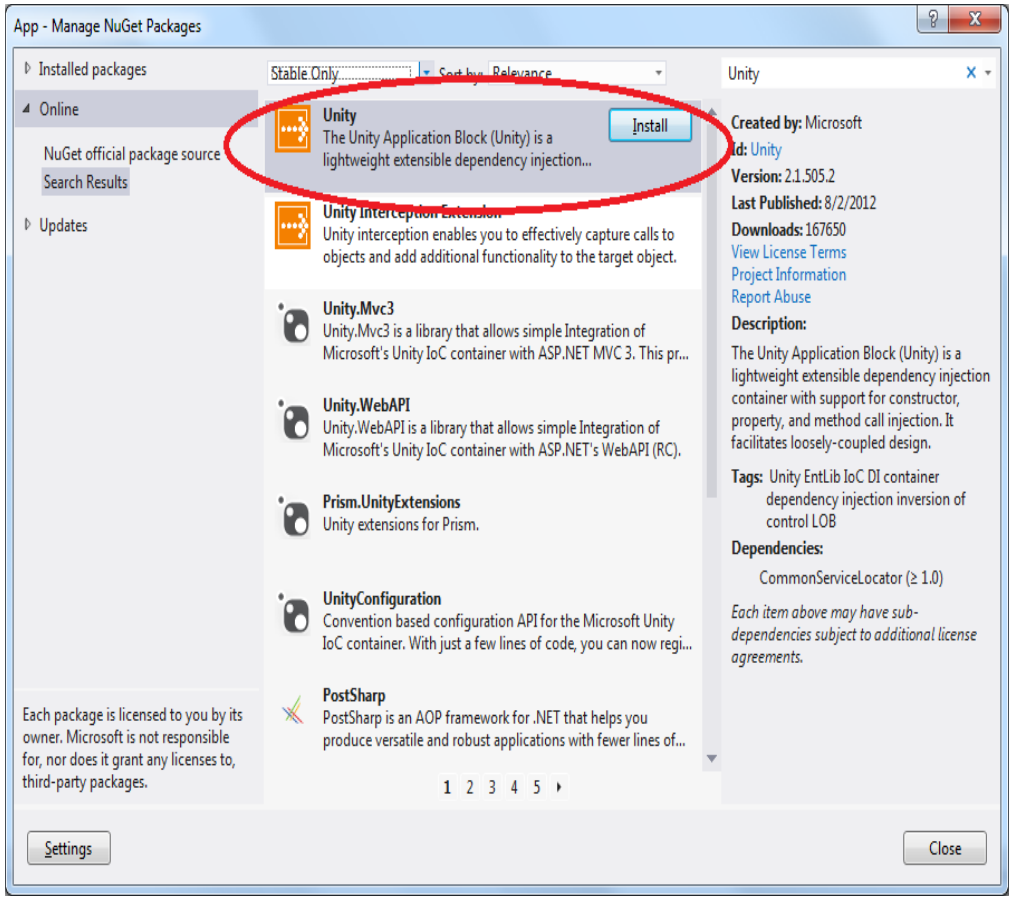
- How do you inject dependencies?
  - What if the dependencies have dependencies?
- Need a reflection framework that knows how to resolve dependencies
  - Can walk a dependency tree
- Inversion of Control means external code creates dependencies
  - In effect Inversion of Control (IoC) = Dependency Injection
- An IoC Container is a tool to wire up dependencies
  - Used to construct application, not generally in testing

## IoC Containers

- There are many IoC containers for .NET
  - Unity
  - StructureMap
  - Ninject
  - Spring.NET
  - Autofac
  - Castle Windsor
  - … many others
- All have similar basic functionality
  - Register mappings of concrete types to abstractions
  - Request implementation of abstraction
  - Container walks dependency tree creating instances from abstractions to wire up objects

# Unity – an Example IoC Container

- Unity is an IoC Container from Microsoft Patterns and Practices
- Install as a Nuget Package


App - Manage NuGet Packages

Installed packages
Online
NuGet official package source
Search Results
Updates

Stable Only | Sort by: Relevance

Unity
The Unity Application Block (Unity) is a lightweight extensible dependency injection...
Install

Unity Interception Extension
Unity interception enables you to effectively capture calls to objects and add additional functionality to the target object.

Unity.Mvc3
Unity.Mvc3 is a library that allows simple Integration of Microsoft's Unity IoC container with ASP.NET MVC 3. This pr...

Unity.WebAPI
Unity.WebAPI is a library that allows simple Integration of Microsoft's Unity IoC container with ASP.NET's WebAPI (RC).

Prism.UnityExtensions
Unity extensions for Prism.

UnityConfiguration
Convention based configuration API for the Microsoft Unity IoC container. With just a few lines of code, you can now regi...

PostSharp
PostSharp is an AOP framework for .NET that helps you produce versatile and robust applications with fewer lines of...

1 2 3 4 5

Unity

**Created by:** Microsoft
**Id:** Unity
**Version:** 2.1.505.2
**Last Published:** 8/2/2012
**Downloads:** 167650
View License Terms
Project Information
Report Abuse
**Description:**
The Unity Application Block (Unity) is a lightweight extensible dependency injection container with support for constructor, property, and method call injection. It facilitates loosely-coupled design.
**Tags:** Unity EntLib IoC DI container dependency injection inversion of control LOB
**Dependencies:**
CommonServiceLocator (≥ 1.0)

*Each item above may have sub-dependencies subject to additional license agreements.*

Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.

Settings | Close

## Unity Basics – Mapping Abstraction to Concrete - 1

- You can map an abstraction to a concrete type with RegisterType
  - By default new instance created on demand

```
IUnityContainer container = new UnityContainer();

container.RegisterType<IAccountRepository,
EFAccountRepository>();
```

## Unity Basics – Mapping Abstraction to Concrete - 2

- You can map an abstraction to an instance with RegisterInstance
  - Reuses same instance by default

```
IUnityContainer container = new UnityContainer();

var repository = new
EFAccountRepository(connectionString, pageSize);

container.RegisterInstance<IAccountRepository>(reposi
tory);
```

## Unity Basics – Creating Multiple Mappings

- You can map same abstraction to multiple concrete types
  - Must name the mapping

```
container.RegisterType<ILogger,
EventLogLogger>("system");

container.RegisterType<ILogger,
FileLogger>("tracing");
```

## Unity Basics – Retrieving Objects

- You can request an instance of a mapped type using Resolve
  - Simply state the abstraction you need
- Can also request instance via mapping name

```
IAccountRepository repo =
container.Resolve<IAccountRepository>();

ILogger traceLogger =
container.Resolve<ILogger>("tracing");
```

## Using Config for Mappings

- Can declare mapping in config file
  - Unity has own configuration section
- Config allows mapping to vary without compilation
  - Extra flexibility
- Do you really want uncontrolled remapping?
  - Normally better to register a map to a factory
  - Allows flexibility with greater control

```
container.RegisterType<ILoggerFactory,
FlexibleLoggerFactory>();
```

```
ILoggerFactory factory =
container.Resolve<ILoggerFactory>();

ILogger logger = factory.Create(LogLevel.Fatal);
```

## Can Override Default Lifetime Management

- Pass LifetimeManager when registering
  - Singleton (container)
  - Transient
  - PerResolve
  - PerThread
  - …

```
container.RegisterType<IAccountRepository, EFAccountRepository>(
                                                                 new
ContainerControlledLifetimeManager());
```

## Summary

- Coding to abstraction decouples your code
- Dependencies injected into classes
- IoC containers can simplify application wireup