Design Patterns for Testability

DEVELOPMENTOR

DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

Part 0: Foundations

DEVELOPMENTOR

DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

The Open/Closed Principle

“Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification”

Translation:

With regard to classes and OO, to change the behavior of a class you should not have to

- Touch the source code (of the class)
- Create a derived class

May sound impossible, but we’ll see how.

Part 1: Inversion of Control

DEVELOPMENTOR

DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

Coding to Concrete Types

- **Tempting to create instances of dependent classes in code**
  - Tightly couples you to implementation
  - Difficult to test
  - Beware of `new` keyword

```
public User Logon( string userId, string password)
{
    LDAPRepository repo = new LDAPRepository();

    LDAPUser ldapUser = repo.Authenticate(userId, password);

    return new User{ Name = ldapUser.Name,
                     Groups = …};
}
```

Coding to Abstraction

- **Coding to abstract types decouples software**
  - Can replace implementation
  - Can test more easily with mock version
- **Extract interface from concrete implementation**
  - Refactoring support can do this
  - Use interface in component
  - May involve extra work
  - May need to build façade to eliminate implementation types

```
public User Logon( string userId, string password)
{
    IUserRepository repo = ????

    User user = repo.Authenticate(userId, password);

    return user;
}
```

Dependency Injection

- **If not able to create dependencies in component where do they come from?**
  - Must be passed to the component, known as **Dependency Injection**
- **Three types of Dependency Injection**
  - Parameter Injection
  - Setter Injection
  - Constructor Injection

Parameter Injection

- **Pass dependency as method parameter**
  - Useful for operation specific strategies

```
public User Logon( string userId,
                   string password,
                   IUserRepository repo )
{
    User user = repo.Authenticate(userId, password);

    return user;
}
```

Setter Injection

- **Have property where dependency can be set**
  - Good for optional dependencies
  - Consider null pattern to remove null checks from code

```
IUserRepository repo;

public IUserRepository Repository
{
   get{ return repo;}
   set{ repo = value ?? new NullRepository();}
}

public User Logon( string userId, string password )
{
    User user = Repository.Authenticate(userId, password);
    return user;
}
```

Constructor Injection

- **Pass dependency to constructor of type**
  - most common form
- **Can also have default constructor with concrete type**
  - maintains interface if changing an existing code base

```
IUserRepository repo;

public OrderingSystem(IUserRepository repo)
{
  this.repo = repo
}

public User Logon( string userId, string password)
{
    User user = repo.Authenticate(userId, password);
    return user;
}
```

Who Creates Dependencies

- **Can define dependencies in config and use creator method**
  - creator creates concrete class and injects dependencies

```
public class OrderSystemCreator
{
  public OrderingSystem CreateOrderingSystem()
  {
    string repoString =
           ConfigurationManager.AppSettings["repo"];
    Type repoType = Type.GetType(repoString);

    IUserRepository repo =
           (IUserRepository)Activator.CreateInstance(repoType);

    return new OrderingSystem(repo);
  }
}
```

Inversion of Control Containers

- **Building custom factories breaks down quickly**
  - Complex dependency trees hard to construct
- **Inversion of Control Containers take control of complex construction**
- **Many IoC Containers available for .NET**
  - Unity
  - Castle Windsor
  - StructureMap
  - Ninject
  - Spring.NET

Unity

- **Unity is a traditional IoC container**
  - from Patterns and Practices
  - maps abstractions to concrete types
  - can be configured in code or configuration file
  - open source project (http://unity.codeplex.com)
- **Dependencies can be supplied via**
  - constructor parameters (most common)
  - public properties[1]
  - method parameters

Registering services in Unity

- **Can map abstract types to concrete types**
  - can name the type so register multiple implementations
  - can control over lifetime of created object
    - by default a new instance is returned each time

```
UnityContainer container = new UnityContainer();

container.RegisterType<IUserLookup, LDAPRepository>();
```

- **Can map abstract types to specific instances**
  - allows creation of objects with runtime parameters
  - inherently a singleton with this mapping

```
UnityContainer container = new UnityContainer();

container.RegisterInstance<IAuthenticationService>(this);
```

Locating registered services in Unity

- **Call `Resolve<T>` to get dependency instance**
  - Unity creates instance and supplies any needed dependencies

```
IOrderLookup orderSystem = container.Resolve<IOrderLookup>();

IOrder order = orderSystem.FindOrderById(...);
```

- **`ResolveAll<T>` returns `IEnumerable` of dependencies**
  - needed if more than one type can satisfy request

```
List<IAuditor> auditers = new List<IAuditor>
        (container.ResolveAll<IAuditor>());

foreach (var auditor in auditors) { ... }
```

Named Mappings

- **Sometimes need to register more than one concrete type for an abstraction**
  - can give a mapping a name
  - pass name to **`Resolve`**

```
UnityContainer container = new UnityContainer();

container.RegisterType<IUserLookup, LDAPRepository>("win");
container.RegisterType<IUserLookup, FormsRepository>("db");

...

var catalog = container.Resolve<IUserLookup>("db");
```

Constructor Injection

- **Dependency passed to constructor of type**
  - most common form in Prism
- **Can also have default constructor with concrete type**
  - maintains interface if changing an existing code base

```
IUserRepository repo;

public OrderingSystem(IUserRepository repo)
{
  this.repo = repo
}
```

Property Injection

- **Dependency assigned to property**
  - unity general used to create type (i.e. **`Resolve<T>`**)
  - happens *after* construction
  - useful if type cannot have parameters in constructor

```
class OrderingSystem
{
    [Dependency]
    IUserRepository UserRepository { get; set; }

    public OrderingSystem()
    {
    }
}
```

attribute used to identify dependency requirement

Registering dependencies in Unity in configuration

- **Unity supports configuration-driven registration**
  - more flexible than code-based registration, but can be fragile
  - requires **`Microsoft.Practices.Unity.Configuration`**

```
<configuration>
   <configSections>
      <section name="unity"
          type="Microsoft.Practices.Unity.Configuration
                         .UnityConfigurationSection,
                Microsoft.Practices.Unity.Configuration" />
   </configSections>
   <unity>
      <container>
         <register type="Interfaces.IOrderService"
                   mapTo="Services.OrderService" />
      </container>
   </unity>
</configuration>
```

Using MEF to manage dependencies

- **Managed Extensibility Framework (MEF) is a .NET component**
  - introduced as part of .NET 4 and Silverlight 4
  - originally released as http://mef.codeplex.com
  - contained in `System.ComponentModel.Composition`
- **MEF is designed around dynamic discovery of components**
  - utilizes an attributed design
  - holds located components in a catalog
  - catalogs can be constructed from various sources
- **Completely decouples contract from implementation**
  - composes values from loaded catalog at runtime
  - no registration with any container necessary
  - can be difficult to diagnose where types are coming from

Identifying services in MEF

- **MEF uses attributes to declare composable services**
  - `ExportAttribute` defines exported service
  - `ImportAttribute` defines necessary service import
  - `ImportManyAttribute` provides `IEnumerable` of services
- **Allows property, field, method or constructor injection**
  - Silverlight requires public properties

```
[Export(typeof(IAuthenticationService))]
[PartCreationPolicy(CreationPolicy.Shared)]
public class AuthenticationService : IAuthenticationService
{
    [Import]
    IUserRepository _userRepo;

    [ImportMany]
    List<IAuditor> _auditors;
}
```

Performing composition in MEF

- **MEF composition is driven through catalogs**
  - define sources for dependencies
  - can be directory, assembly list, .xap list, or even custom
- **CompositionContainer then consults loaded catalogs**

```
var catalogSources = new AggregateCatalog(
      new AssemblyCatalog(Assembly.GetExecutingAssembly()),
      new DirectoryCatalog(@".\Extensions"));

CompositionContainer container = new
                    CompositionContainer(catalog);
```

Locating registered objects with MEF

- **Call GetExport<T> to locate specific dependency**
  - returns **`Lazy<T>`** for lazy-creation

```
CompositionContainer container = ...;

IOrderService orderService =
        container.GetExport<IOrderService>().Value;
```

- **Requires that the interface or implementation is defined as a valid export through an attribute**
  - **`InheritedExport`** implies any concrete implementation is automatically exported to MEF which is convenient

```
[Export(typeof(IOrderService))]
class OrderService : IOrderService
{
}
```

```
[InheritedExport]
public interface IOrderService
{
}
```

Requesting composition with MEF

- **Call `ComposeParts` to fill in dependencies on existing objects**
  - useful if object was created manually
  - scans supplied object(s) and populates dependencies

```
CompositionContainer container = ...;

class AuthenticationWindow
{
   [Import] IAuthenticationService _authService;

   public AuthenticationWindow()
   {
      container.ComposeParts(this);
      InitializeComponent();
   }
}
```

Summary

- Dependency Injection makes code loosely coupled and unit testable
- IoC brings sanity in DI systems
- Unity is Microsoft’s IoC Container
- MEF is alternative for plug-in capability, built into .NET 4

Part 2:
Test Doubles and Mocking

DEVELOPMENTOR

DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

Agenda

- What are doubles?
- Continuum of doubles
- Mocking frameworks
- Different approaches to mocking

What are Doubles?

- **Replacements for part of your code**
  - Depended on Components (DOC) of SUT
  - Double provides same API as DOC

Why Doubles?

- **Return values of components may not be repeatable**
  - Date/time values
- **Calls may be 'risky' or may be charged for**
  - Calling live web services during test
- **Parts of the application are 'slow'**
  - Database access
  - File access
- **Unit tests should not rely on external resources**
  - Databases
  - Web Services

Continuum of Doubles

- **Fake Object**
  - Provides same implementation as DOC but is much simpler
- **Test Spy**
  - Like stub but also captures outputs of SUT
- **Test Stub**
  - Used to specify control options for SUT
  - e.g. different return values force SUT down different paths
- **Mock Object**
  - Provides behaviour verification
  - e.g. correct methods called in correct order

Fakes

- **Provides lightweight implementation to SUT**
  - Return correct values
  - Allow SUT to call methods
  - May hold parameter as values to return later

```
public interface IStatementStrategy {
    void PrintTitle(string title);
    void PrintHeader(string header);
    void PrintLine(string line);
    void PrintFooter(string footer);
}

class FakeStrategy : IStatementStrategy {
    public void PrintTitle(string title) {}
    public void PrintHeader(string header) {}
    public void PrintLine(string line) {}
    public void PrintFooter(string footer) {}
}
```

Stubs

- **Provide 'indirect' inputs to SUT**
  - e.g. create Stub loggers to return true or false

```
public interface ILogger ILogger
{
    bool IsDebugMode { get; set; }
}

class StubLogger : ILogger
{
    public bool IsDebugMode { get { return true; } set {} }
}

public void Log(ILogger logger)
{
    if(logger.IsDebugMode)
    {
        Console.WriteLine();
    }
}
```

Mocks

- **Provide behavior verification**
  - Check that the SUT calls correct methods ...
  - ... with the correct parameters ...
  - ... in the correct order
- **Painful to build on your own**
  - too much work – repeated over and over!
  - tooling becomes very helpful here

Tools

- **Typically created with a tool**
  - RhinoMocks
  - Moq
  - TypeMock
  - NMock
- **Usually used to create mocks and stubs**

Rhino Mocks

- **Created by Ayende Rahien (Oren Eini)**
  - Open source
  - Actively developed
  - Widely used
- **Not standalone**
  - Creates mock objects
  - Still need testing framework to run tests

Using Rhino

- **Imagine testing this**
  - Have to fake up the **`IAccountRepository`**
  - Maybe for different scenarios
  - Rather than create multiple instances of **`IAccountRepository`**
    - can use mocking library

```
public class SimpleBankFactory {
    public IEnumerable<Account> GetAccounts(IAccountRepository repository){
        return repository.GetAccounts();
    }

    public Account GetAccount(int id, IAccountRepository repository){
        return repository.GetAccount(id);
    }

    public bool SaveAccount(Account account, IAccountRepository repository){
        return repository.SaveAccount(account);
    }
}
```

Rhino Basics

- **Stubs return values and throw exceptions**
  - Generated by Rhino's **`MockRepository`** class
    - Then tell the stub what to do

```
[Test]
public void AccountFactory_GetAccount_Succeeds()
{
    IAccountRepository repo =
              MockRepository.GenerateStub<IAccountRepository>();

    repo.Stub(r => r.GetAccount(1)).Return(new Account());

    Account account = bankFactory.GetAccount(1, repo);
    Assert.That(account, Is.Not.Null);
}
```

Stubbing exceptions

- **Stubs can throw exceptions**
  - Must specify **`IgnoreArguments`**
    - Then tell the stub what to do

```
[Test]
[ExpectedException(typeof(InvalidAccountIdException))]
public void AccountFactory_GetAccount_Succeeds()
{
    IAccountRepository repo = MockRepository.GenerateStub<IAccountRepository>();

    repo.Stub(r => r.GetAccount(0))
        .IgnoreArguments()
        .Throw(new InvalidAccountIdException());

    account = bankFactory.GetAccount(0, repo);
}
```

Rhino Expectations

- **Ask Rhino to create a Mock**
  - Create a **`MockRepository`** instance
  - Ask it to create the mocks
  - Replay the calls
    - This puts the stub into the 'replay' state
  - Verify the calls have happened

```
[Test]
public void AccountFactory_GetAccount_Succeeds() {
    MockRepository mocks = new MockRepository();

    IAccountRepository repo = mocks.DynamicMock<IAccountRepository>();
    mocks.ReplayAll();

    // Use the mock here

    mocks.VerifyAll();
}
```

Different mocks with Rhino

- **Rhino provides mocks with different 'replay semantics'**
- **Strict**
  - Only recorded methods will be replayed
  - Any other methods called on mock are invalid
  - Not calling recorded methods is invalid
- **Dynamic**
  - All method calls accepted
  - Non recorded calls return null or zero
- **Partial**
  - Available for classes only
  - Any non-abstract call uses actual class

Set expectations on the mocks

- **Use Expect to set expectation**
  - What methods will be called
  - What parameters will be passed

```
[Test]
public void AccountFactory_GetAccount_Succeeds()
{
    MockRepository mocks = new MockRepository();
    SimpleBankFactory bankFactory = new SimpleBankFactory();
    IAccountRepository repo = mocks.DynamicMock<IAccountRepository>();
    Account mockAccount = mocks.DynamicMock<Account>(200);

    Expect.Call(repo.GetAccount(1)).Return(mockAccount);
    mocks.ReplayAll();

    Account account = bankFactory.GetAccount(1, repo);

    mocks.VerifyAll();
}
```

Exceptions

- **Can set an expectation to throw an exception**

```
[Test]
[ExpectedException(typeof(InvalidAccountIdException))]
public void AccountFactory_GetAccount_Succeeds()
{
    MockRepository mocks = new MockRepository();
    SimpleBankFactory bankFactory = new SimpleBankFactory();
    IAccountRepository repo = mocks.DynamicMock<IAccountRepository>();

    Expect.Call(repo.GetAccount(1)).Throw(new ArgumentNullException());
    mocks.ReplayAll();

    Account account = bankFactory.GetAccount(1, repo);
    mocks.VerifyAll();
}
```

Void Returns

- **Can't 'Expect' void returns directly**
  - Instead pass an Action delegate

```
[Test]
public void AccountFactory_GetAccount_Succeeds()
{
    MockRepository mocks = new MockRepository();
    IAccountRepository repo = mocks.DynamicMock<IAccountRepository>();
    Account mockAccount = mocks.DynamicMock<Account>(200);
    Expect.Call(repo.GetAccount(1)).Throw(new ArgumentNullException());

    Expect.Call(() => mockAccount.Deposit(200));

    mocks.ReplayAll();

    SimpleBankFactory bankFactory = new SimpleBankFactory();
    Account account = bankFactory.GetAccount(1, repo);
    account.Deposit(200);

    mocks.VerifyAll();
}
```

Properties and Delegates

- **Properties**

```
Expect.Call(customer.Name).Return("Kevin");
Expect.Call(customer.Name = "Kevin");
```

- **Delegates**

```
Predicate<int> predicate = mocks.CreateMock<Predicate<int>>();
Expect.Call(predicate(42)).Return(true);
```

Repetition

- **Calls to methods may be repeated**

```
Repeat.Once()
Repeat.Any()
Repeat.Never()
Repeat.AtLeastOnce()
Repeat.Twice()
Repeat.Times(3)
Repeat.Times(4, int.MaxValue)
```

Conclusion

- Doubles allow for awkward parts of the SUT to be tested
- There is a continuum of doubles
- Can create our own
- Can use a mocking framework