Mocking and IoC

DEVELOPMENTOR

DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

Agenda

- **Understand the need for mocking**
- **Understand the need for IoC containers**
- **Using Rhino**
- **Using Unity**

Code to abstractions

- **Using concrete types is often a bad idea**
  - Better to code to interfaces
  - Can replace real objects with test objects
  - Can use dependency injection

Creating instances

- **Using 'new' means we are creating objects**
  - Tied to that implementation
  - Use of 'new' considered evil!

```
IEnumerable<Users> GetUsers() {
    UsersDAO dao = new UsersDAO();
    dao.GetUsers();
}
```

This is hard to test

- **Using a concrete DAO**
  - Tied to a specific data access mechanism
- **Impossible to mock**

Code to abstractions

- **Extract the methods into an interface**
  - Then need a way to construct the concrete instance
  - Also need a way to get a reference to the concrete instance
  - Can use "dependency injection"

Use 'dependency injection'

- **Three ways to do**
  - Parameter injection
  - Setter injection
  - Constructor injection

A problem with injection

- **Concrete class still coded somewhere**
  - Changing implementation means changing code
- **Can solve by putting concrete type in configuration file**
  - Use a factory
  - Use IoC

```
public class ToDoListDaoFactory {
    public IEnumerable<ToDoList> GetToDoLists(){
                                    repository = LoadRepository();
        return new ToDoListDao().GetToDoLists(repository);
    }
}

private static IToDoListRepository LoadRepository() {
string typeAsString = ConfigurationManager.AppSettings["factory"];
    type = Type.GetType(typeAsString);
    repository = (IToDoListRepository)Activator.CreateInstance(type);
    return repository;
}
```

Inversion Of Control (IoC) to the Rescue

- **IoC containers**
  - Manage creation and disposal of application objects
  - Manage dependencies
- **IoC is a principle**
  - Framework knows about and makes invocations on application objects
  - This is the opposite (inverse) of standard APIs
- **Dependency injection**
  - Is a consequence of IoC
- **Microsoft container**
  - Unity
  - Part of Enterprise Patterns Lib

Using a Container

- **Container is typically a single instance**
  - Essentially a dictionary of interface => concrete implementation
- **Types are registered with the container**
  - In configuration file
  - or code
- **Container resolves types when needed**

Registering objects in code

- **Call container's Register method**

```
IUnityContainer unityContainer = new UnityContainer();

unityContainer.RegisterType<IToDoListDaoFactory, TestToDoRepository>();
```

Registering objects through configuration

- **Unity configuration section**
  - Register type mappings and aliases

```
<configuration>
  <configSections>
    <section name="unity"
             type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection,
                   Microsoft.Practices.Unity.Configuration" />
  </configSections>
  <unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
    <alias alias="IToDoListRepository"
                          type="ToDoRepository.IToDoListRepository, ToDoRepository" />
    <alias alias="ToDoListRepository"
                          type="ToDoRepository.ToDoListRepository, ToDoRepository" />

    <container>
      <register type="IToDoListRepository" mapTo="ToDoListRepository" >
        <constructor>
          <param name="connectionString" value="..." />
        </constructor>
      </register>
    </container>
  </unity>
</configuration>
```

Accessing objects in code

- **Use an instance of the container**
  - Call it's Resolve method
  - Must load configuration first

```
UnityConfigurationSection section =
                ConfigurationManager.GetSection("unity") as UnityConfigurationSection;
if (section != null)
{
    section.Configure(unityContainer);
}
```

```
IUnityContainer unityContainer = new UnityContainer();

_repository = IoC.Container.Instance.Resolve<IToDoListRepository>();
```

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

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 IToDoListRepository
  - Maybe for different scenarios
  - Rather than create multiple instances of IToDoListRepository
    - can use mocking library

```
public class SimpleBankFactory {
    public IEnumerable<ToDoList> GetToDoLists(IToDoListRepository repository){
        return repository.GetToDoLists();
    }

    public ToDoList GetToDoList(int id, IToDoListRepository repository){
        return repository.GetToDoList(id);
    }

    public bool SaveToDoList(ToDoList ToDoList, IToDoListRepository repository){
        return repository.SaveToDoList(toDoList);
    }
}
```

Rhino Basics

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

```
[TestMethod]
public void ToDoListFactory_GetToDoList_Succeeds()
{
    IToDoListRepository repo = MockRepository.GenerateStub<IToDoListRepository>();

    repo.Stub(r => r.GetToDoList(1)).Return(new ToDoList());

    ToDoList toDoList = bankFactory.GetToDoList(1, repo);
    Assert.IsNotNull(toDoList);
}
```

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

```
[TestMethod]
public void ToDoListFactory_GetToDoList_Succeeds() {
    mocks = new MockRepository();
    repo = mocks.DynamicMock<IToDoListRepository>();
    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

```
[TestMethod]
public void ToDoListFactory_GetToDoList_Succeeds()
{
    MockRepository mocks = new MockRepository();
    SimpleBankFactory bankFactory = new SimpleBankFactory();
    IToDoListRepository repo;
    ToDoList mockToDoList = mocks.DynamicMock<ToDoList>(200);

    Expect.Call(repo.GetToDoList(1)).Return(mockToDoList);
    mocks.ReplayAll();

    ToDoList toDoList = bankFactory.GetToDoList(1, repo);

    mocks.VerifyAll();
}
```

Rhino 'using' syntax

```
[TestMethod]
public void ToDoListFactory_GetToDoList_Succeeds()
{
    MockRepository mocks = new MockRepository();
    SimpleBankFactory bankFactory = new SimpleBankFactory();
    IToDoListRepository repo;
    using (mocks.Record())
    {
        repo = mocks.DynamicMock<IToDoListRepository>();
        ToDoList mockToDoList = mocks.DynamicMock<ToDoList>(200);
        Expect.Call(repo.GetToDoList(1)).Return(mockToDoList);
    }
    using (mocks.Playback())
    {
        ToDoList toDoList = bankFactory.GetToDoList(1, repo);
    }
}
```

When to use expectations

- **Checking certain code paths have been executed**
  - Different paths depend of data in class
- **Check that specific method is called**
  - May have 'indirect' output only
  - e.g. logging or auditing

Summary

- Inversion of control allows the framework to control type creation
- Allows for greater flexibility
- Types can be injected into your code
- Containers make this easier through configuration
- 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