From Zero to 1000 tests in 6 months

Or how not to lose your mind with 2 week iterations

Name is Max Vasilyev

Senior Developer and QA Manager

at Solutions Aberdeen

solutions
IT solutions for business

http://tech.trailmax.info

@trailmax

Business Does Not Care

- Business does not care about tests.
- Business does not care about internal software quality.
- Business does not care about architecture.
- Some businesses don’t care so much, they even don’t care about money.

Don’t Tell The Business

Just do it!

Just write your tests, ask no one.

Honestly, tomorrow in the office just create new project, add NUnit package and write a test.

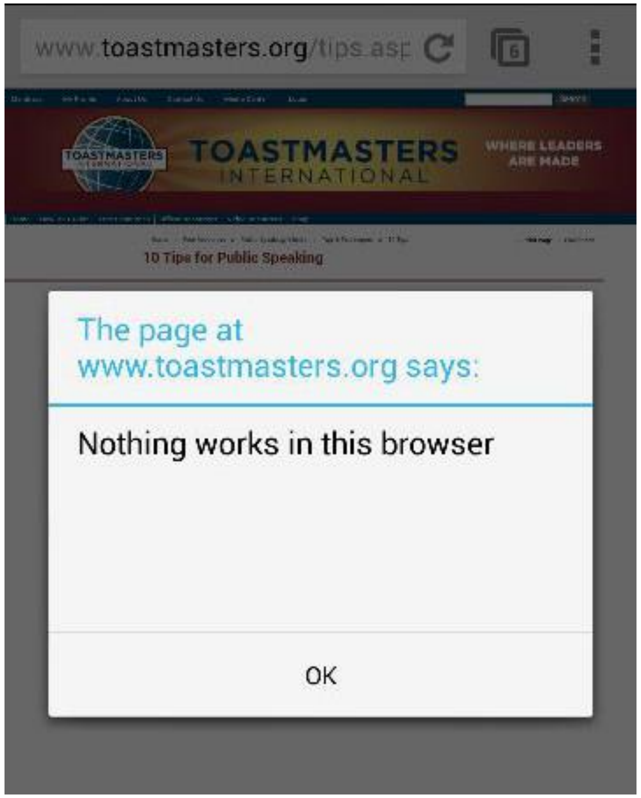
That’ll take you 10 minutes.

Simple?

Writing a test is simple. Writing a good test is hard.

Main questions are:

- What do you test?
- Why do you test?
- How do you test?


www.toastmasters.org/tips.asp
TOASTMASTERS INTERNATIONAL
WHERE LEADERS ARE MADE
10 Tips for Public Speaking
OK

The page at www.toastmasters.org says:

Nothing works in this browser

Our Journey: Stone Age

Started with Selenium browser tests:

- Recording tool is OK to get started
- Boss loved it!
- Things fly about on the screen
  - very dramatic

But:

- High maintenance effort
- Problematic to check business logic

Our Journey: Iron Age

After initial Selenium fever, moved on to integration tests:

- Hook database into tests and part-test database.

But:

- Very difficult to set up (data + infrastructure)
- Problematic to test logic

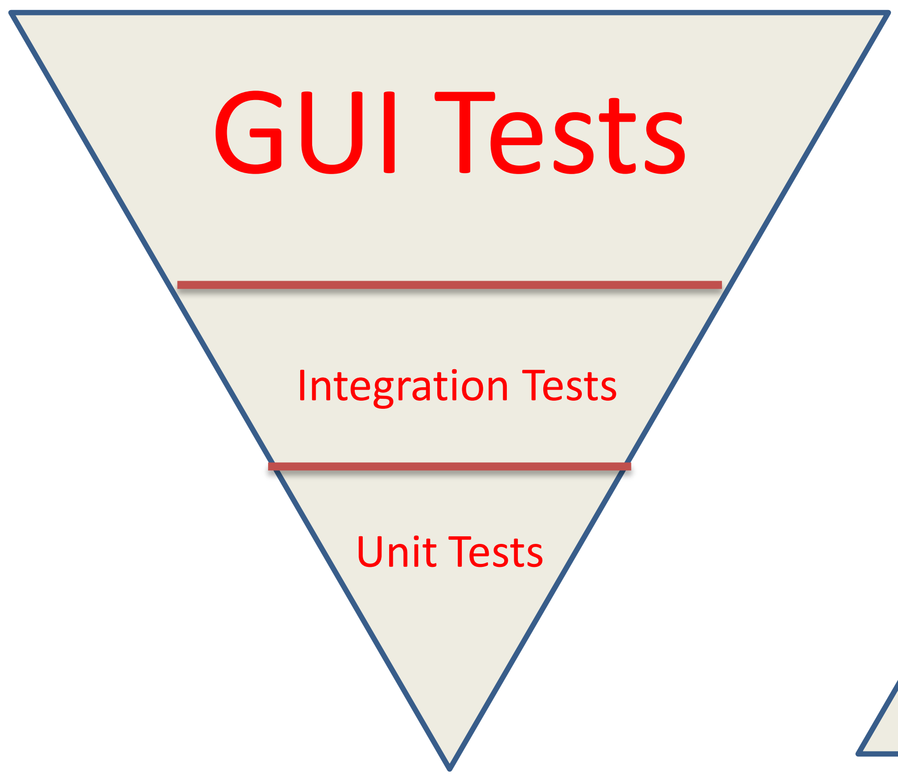
Our Journey: Our Days

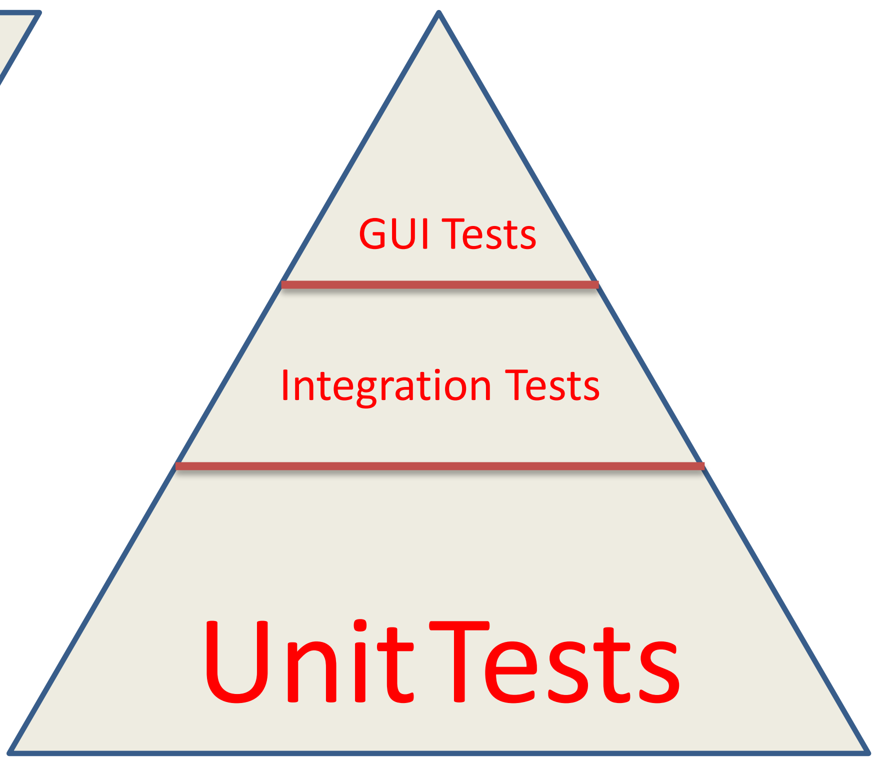
- Now no Selenium tests
- A handful of integration tests
- Most of the tests are unit-ish* tests
- 150K lines of code in the project
- Around 1200 tests with 30% coverage**
- Tests are run in build server TeamCity

* Discuss Unit vs Non-Unit tests later

** Roughly 1 line of test code covers 2 lines of production code

Testing Triangle

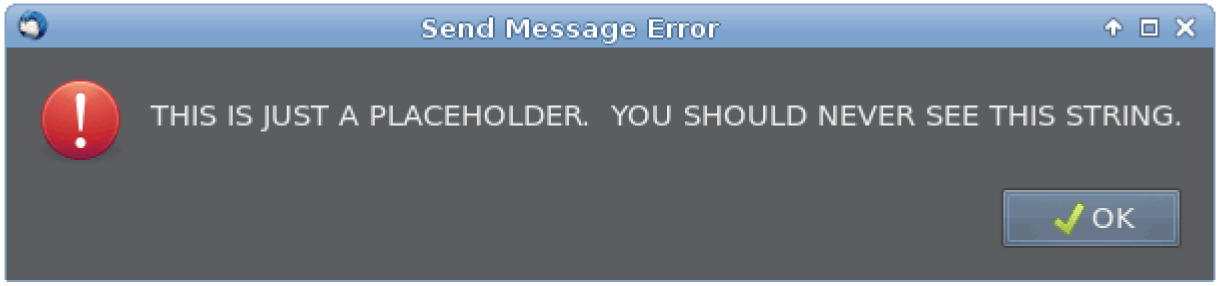




Our Journey: 2 Week Iterations?

The team realised tests are not optional after first 2-week iteration:

- There simply was no time to manually test everything at the end of iteration.


Send Message Error

THIS IS JUST A PLACEHOLDER. YOU SHOULD NEVER SEE THIS STRING.

ADD: Annoyance Driven Development

Bits of code annoy you?

FIX IT!

That's me on bug-fixing day

Annoyance is Bad

- Bad smells in code slow you down
- Frustrate you
- Other team members might come across the same issues (so you can be frustrated together!)

Placeholder for funny picture about frustration

Some examples

- Namespacing in your Razor *.cshtml pages? Take them to web.config
- Get latest changes from git/svn/tfs/hg/etc. And you can’t build the project? Start using build server
- Receive a ticket for “already-fixed” bug? Add a regression test.

Fixed it yet?

- Do the fixing in the start of an Iteration
- Does it not affect anything? Do it now! Don’t let it annoy you (or anybody else) anymore
- Fixing it might consume your time, but only once. If not fixed, you’ll get annoyed again.
- And put a regression test so the problem does not come back! (where applicable)

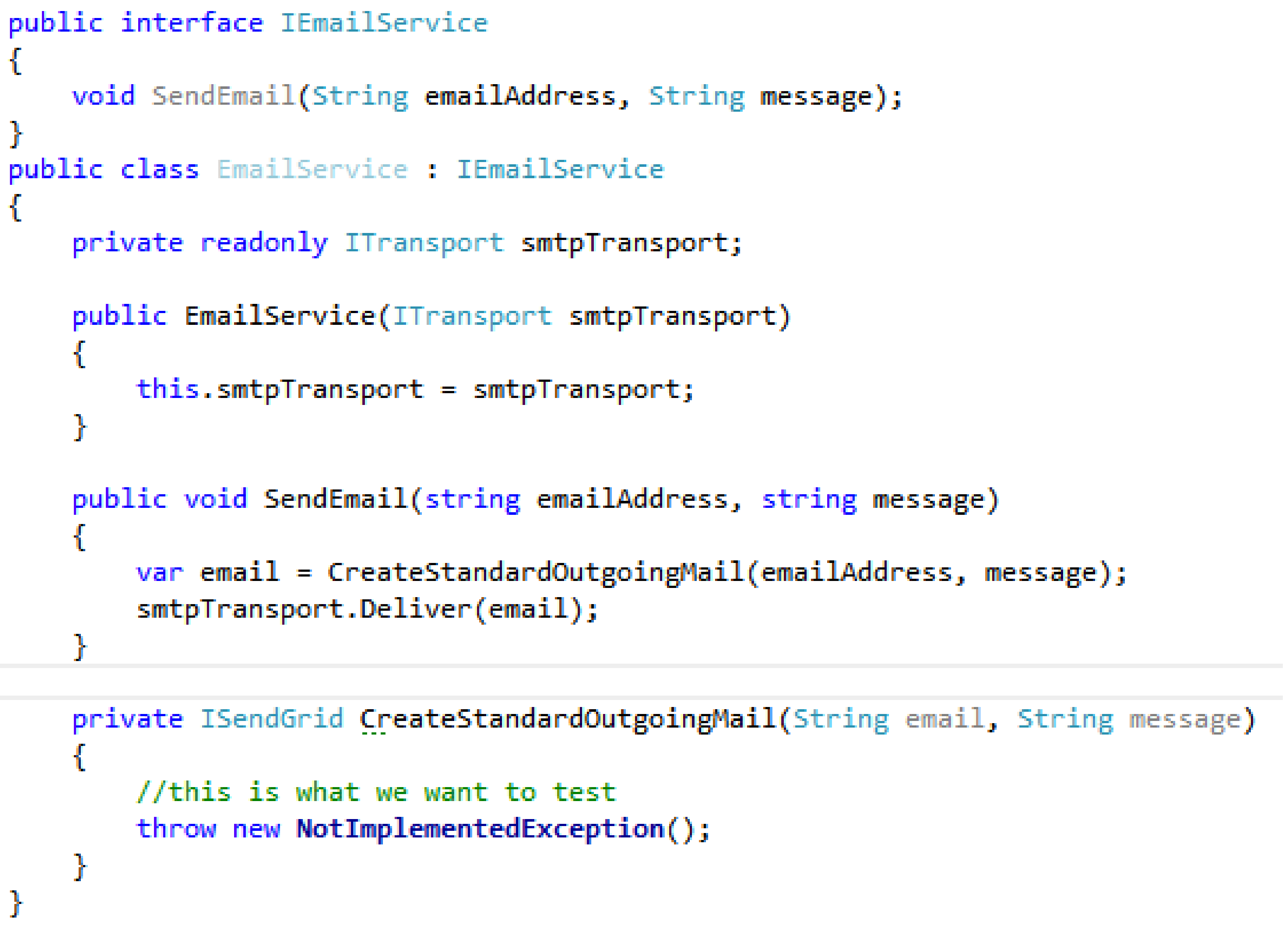
Unit Testing: Isolation Frameworks

- “**Mock objects** are simulated objects that mimic the behaviour of real objects in controlled ways”*
- You create mock object to isolate class under test. You tell mock object to create situation that you want to simulate.

* Wikipedia

Unit Testing: Isolation Frameworks

- Hand-crafted stubs and mocks
- Moq, Nsubstitute, Rhino Mocks, MS Fakes, FakeItEasy, etc.
- Mocks fail your tests. Stubs keep them running
- You verify against mocks

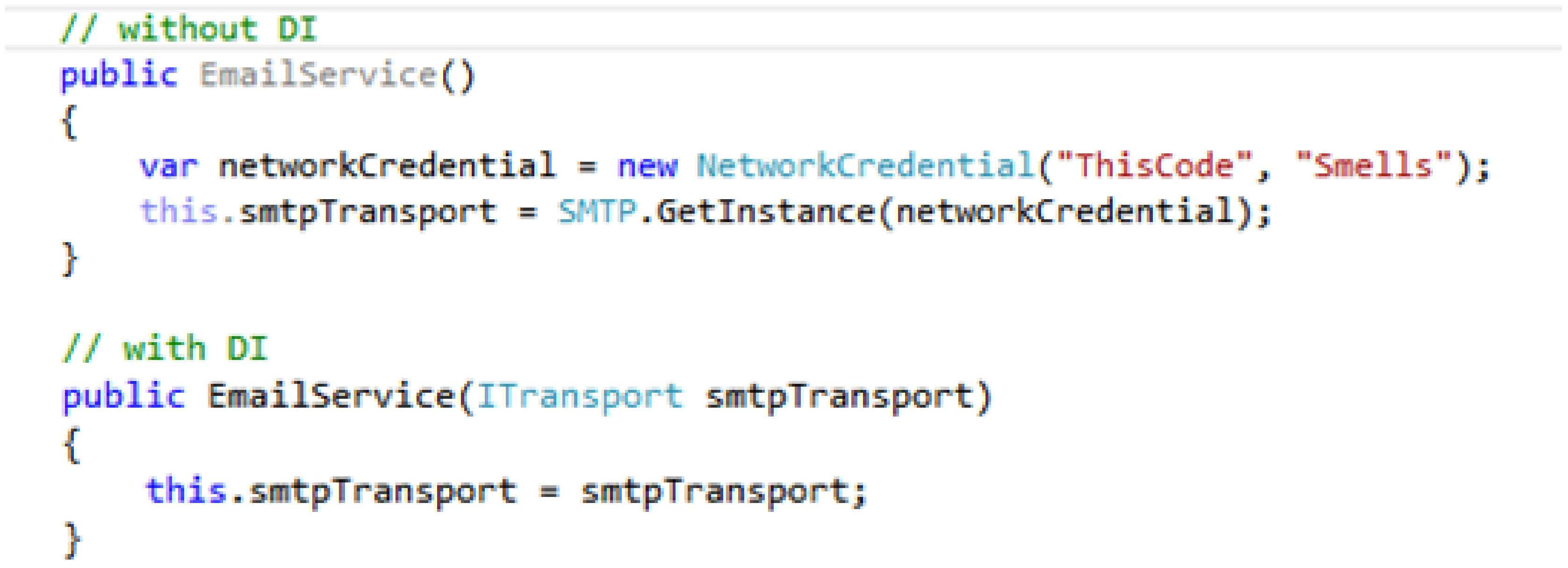


```
public interface IEmailService
{
    void SendEmail(String emailAddress, String message);
}
public class EmailService : IEmailService
{
    private readonly ITransport smtpTransport;

    public EmailService(ITransport smtpTransport)
    {
        this.smtpTransport = smtpTransport;
    }

    public void SendEmail(string emailAddress, string message)
    {
        var email = CreateStandardOutgoingMail(emailAddress, message);
        smtpTransport.Deliver(email);
    }

    private ISendGrid CreateStandardOutgoingMail(String email, String message)
    {
        //this is what we want to test
        throw new NotImplementedException();
    }
}
```

```
[Test]
public void SendEmail_Always_UsesTransport()
{
    //Arrange
    var transport = new Mock<ITransport>();

    var sut = new EmailService(transport.Object);
    // Act
    sut.SendEmail("bill@gates.com", "Start button is not funny!");

    // Assert
    transport.Verify(t => t.Deliver(It.IsAny<ISendGrid>()));
}
```

Dependency Injection

- Classes don’t create their dependencies
- Dependencies are given to classes



```
// without DI
public EmailService()
{
    var networkCredential = new NetworkCredential("ThisCode", "Smells");
    this.smtpTransport = SMTP.GetInstance(networkCredential);
}

// with DI
public EmailService(ITransport smtpTransport)
{
    this.smtpTransport = smtpTransport;
}
```

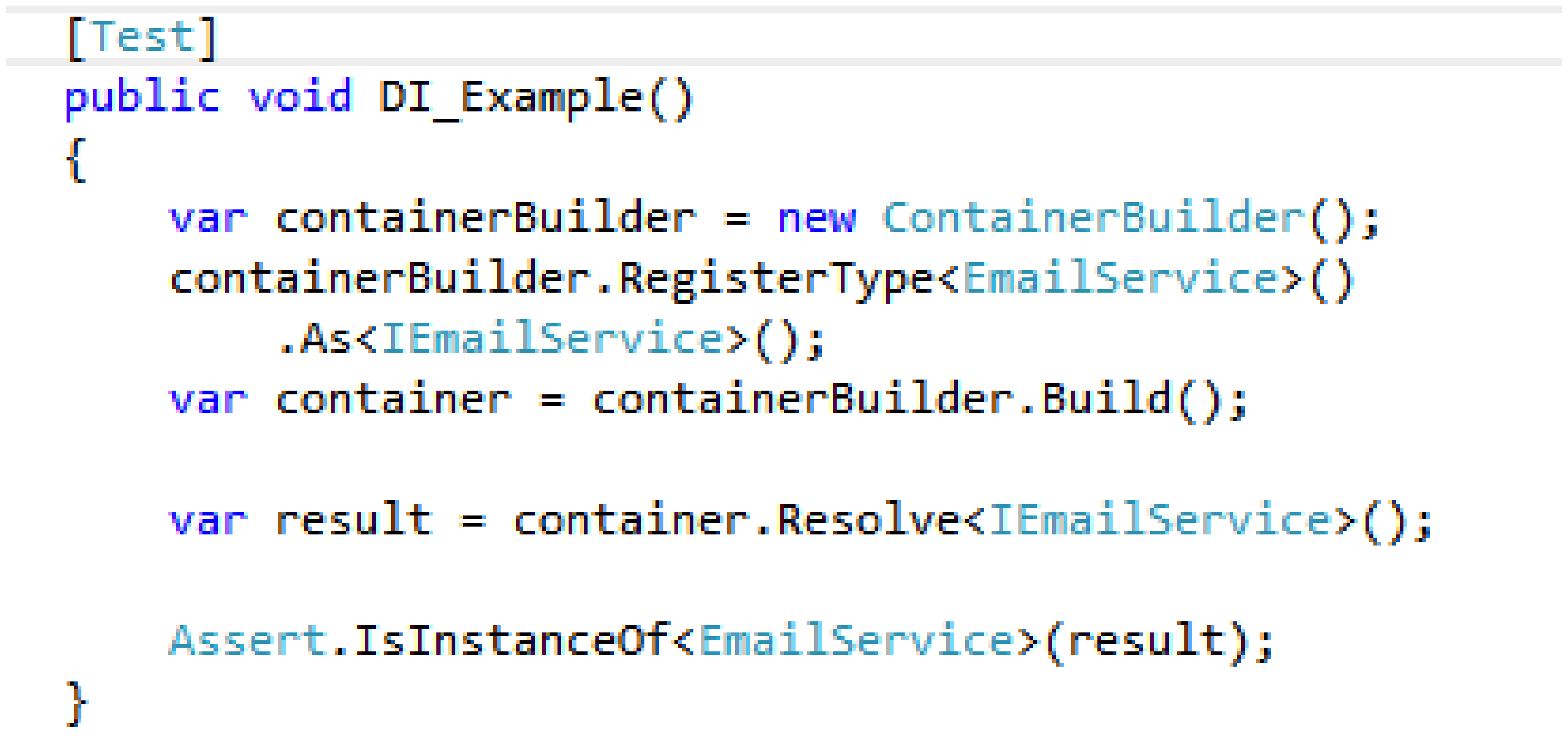
DI Principles

EVERY TIME YOU USE 'NEW'
GOD KILLS A KITTEN
memegenerator.net

Dependency Injection Benefits

- Improves testability
- Improves application architecture
- Can do crazy things with DI container
  - Adding decorators to implementations
  - Multi-tenancy implementation
  - Provide different implementations depending on environment/conditions
  - Lifetime management. “Singleton” is not a pattern!

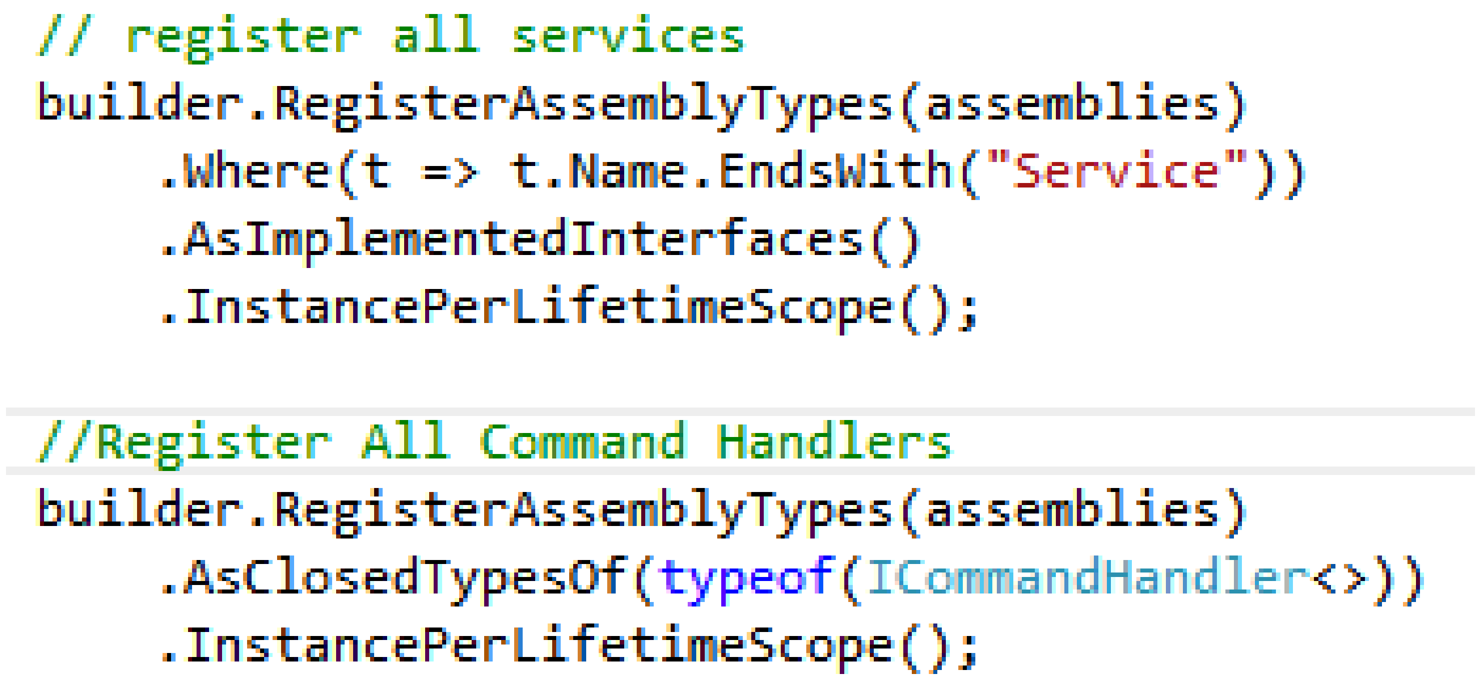
DI Container Example*



```
[Test]
public void DI_Example()
{
    var containerBuilder = new ContainerBuilder();
    containerBuilder.RegisterType<EmailService>()
        .As<IEmailService>();
    var container = containerBuilder.Build();

    var result = container.Resolve<IEmailService>();

    Assert.IsInstanceOf<EmailService>(result);
}
```

* With Autofac container. There are other good containers.

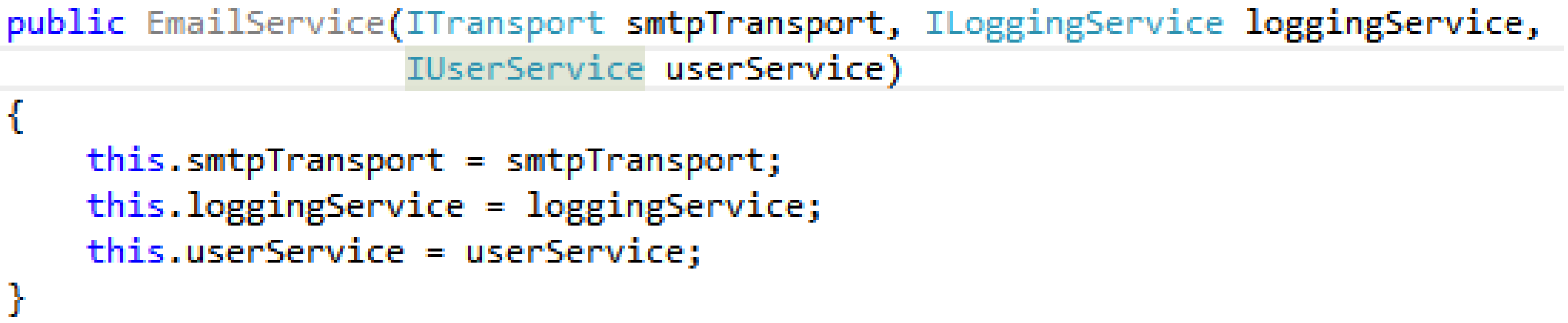
DI Container Registration Example



```
// register all services
builder.RegisterAssemblyTypes(assemblies)
    .Where(t => t.Name.EndsWith("Service"))
    .AsImplementedInterfaces()
    .InstancePerLifetimeScope();

//Register All Command Handlers
builder.RegisterAssemblyTypes(assemblies)
    .AsClosedTypesOf(typeof(ICommandHandler<>))
    .InstancePerLifetimeScope();
```

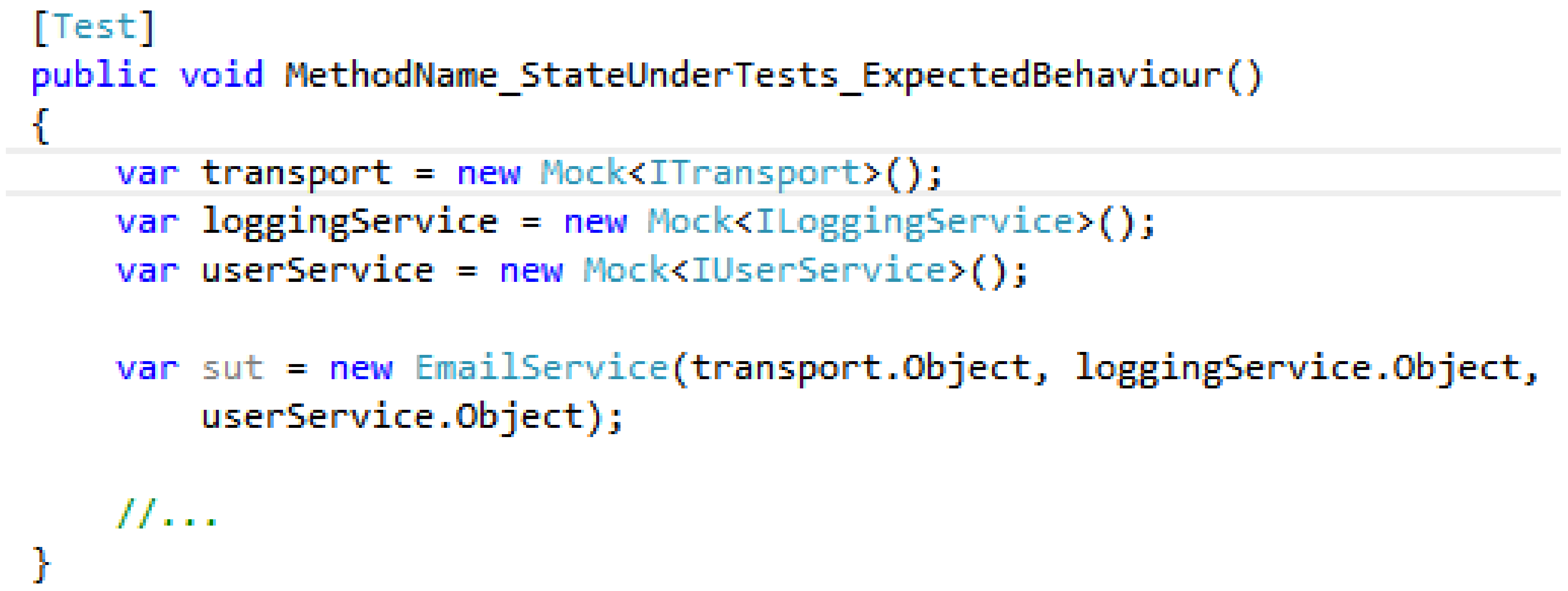
Back to Tests

If your classes grow with dependencies, tests are getting boring and time-consuming



```
public EmailService(ITransport smtpTransport, ILoggingService loggingService,
                    IUserService userService)
{
    this.smtpTransport = smtpTransport;
    this.loggingService = loggingService;
    this.userService = userService;
}
```

With many dependencies your tests become bloated and a maintenance nightmare. And nobody wants to write them anymore!

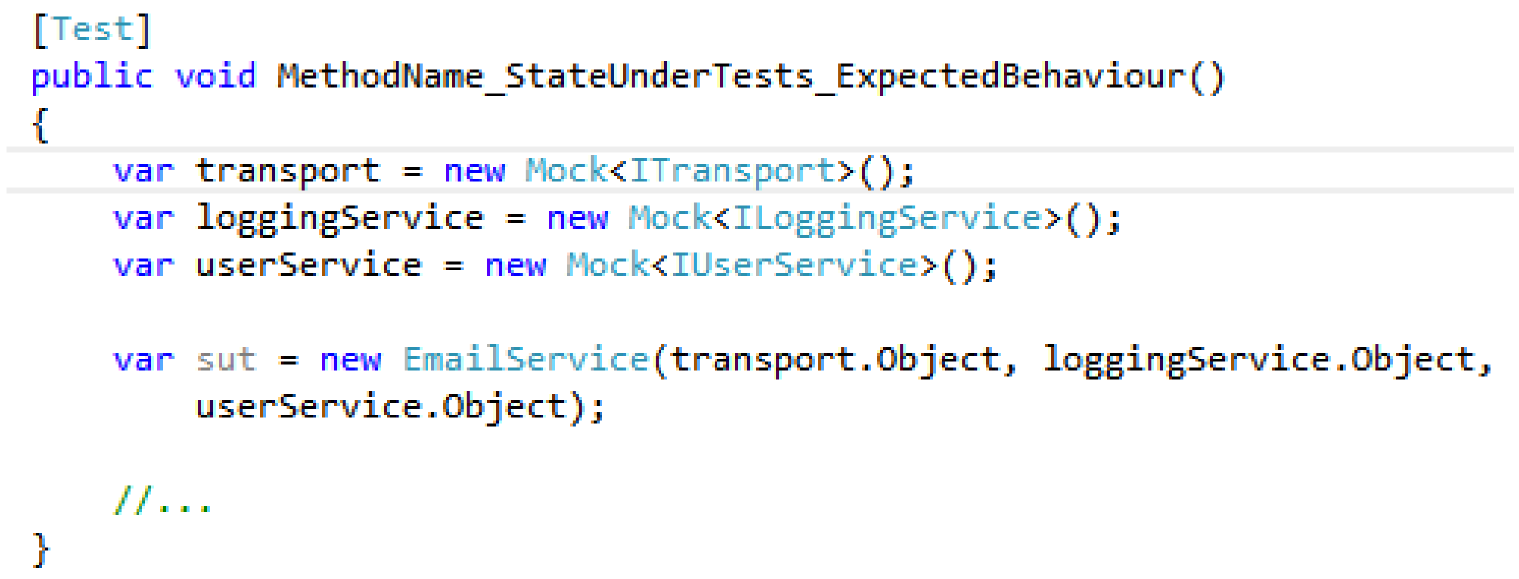


```
[Test]
public void MethodName_StateUnderTests_ExpectedBehaviour()
{
    var transport = new Mock<ITransport>();
    var loggingService = new Mock<ILoggingService>();
    var userService = new Mock<IUserService>();

    var sut = new EmailService(transport.Object, loggingService.Object,
        userService.Object);

    //...
}
```

DI + Mocking = Automocking

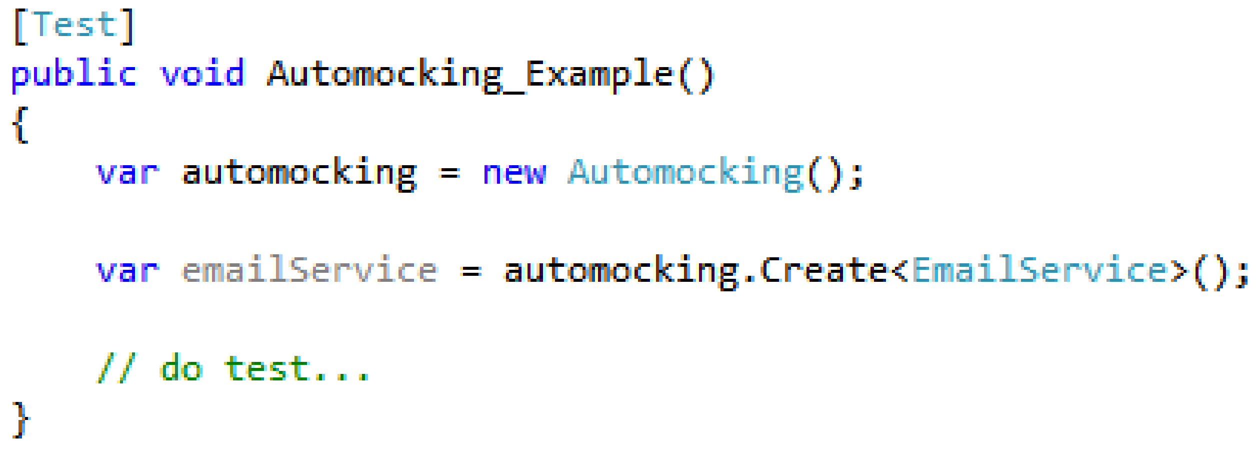
- Automocking container is DI container configured to give you mock objects as dependencies
- Streamlines test-writing
- Tests are no longer broken with introduction of a new dependency
- Can do crazy things with it: some objects are mocked, some are real.

Without Automocking



```
[Test]
public void MethodName_StateUnderTests_ExpectedBehaviour()
{
    var transport = new Mock<ITransport>();
    var loggingService = new Mock<ILoggingService>();
    var userService = new Mock<IUserService>();

    var sut = new EmailService(transport.Object, loggingService.Object,
        userService.Object);

    //...
}
```

With Automocking:



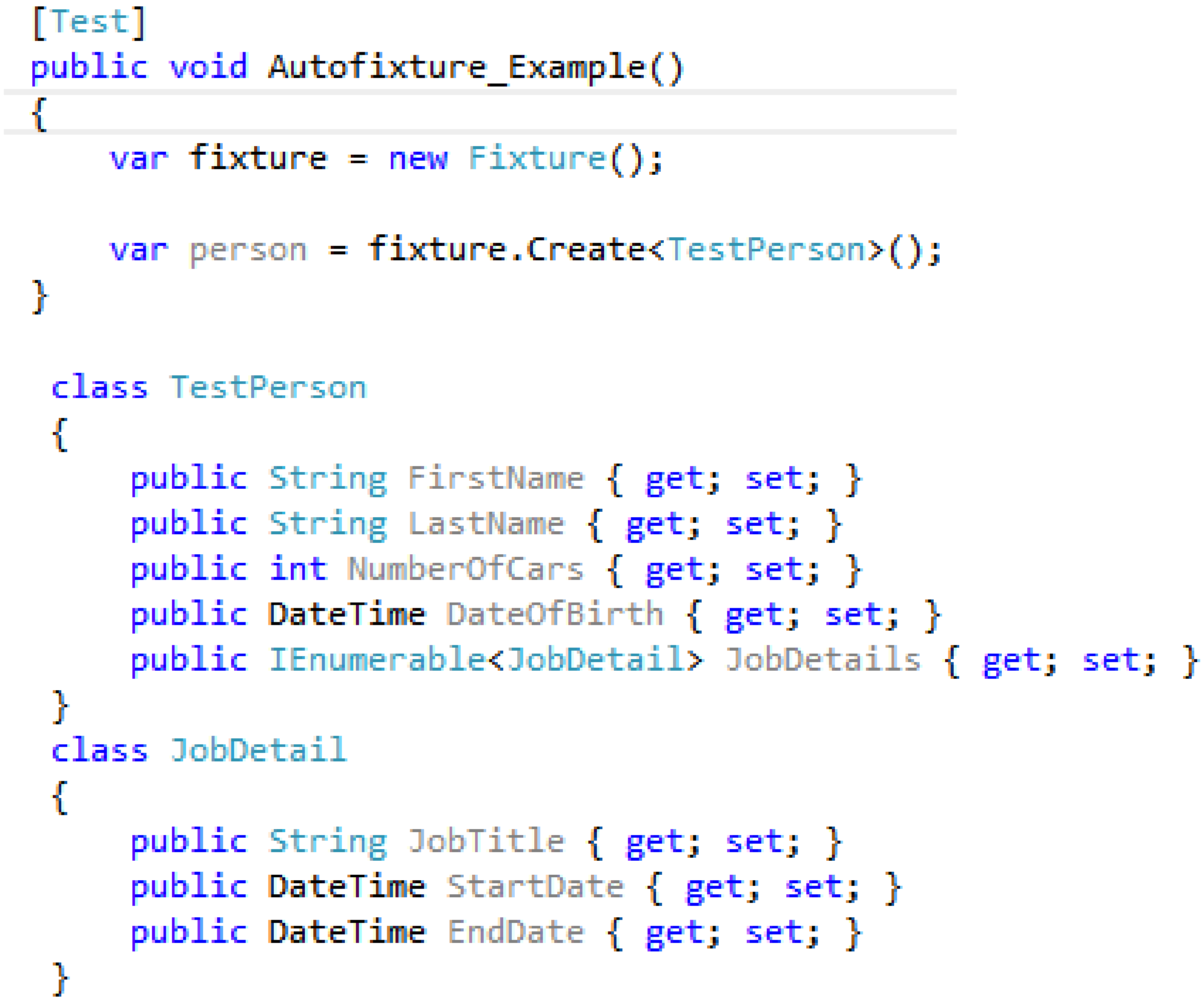
```
[Test]
public void Automocking_Example()
{
    var automocking = new Automocking();

    var emailService = automocking.Create<EmailService>();

    // do test...
}
```

Autofixture*

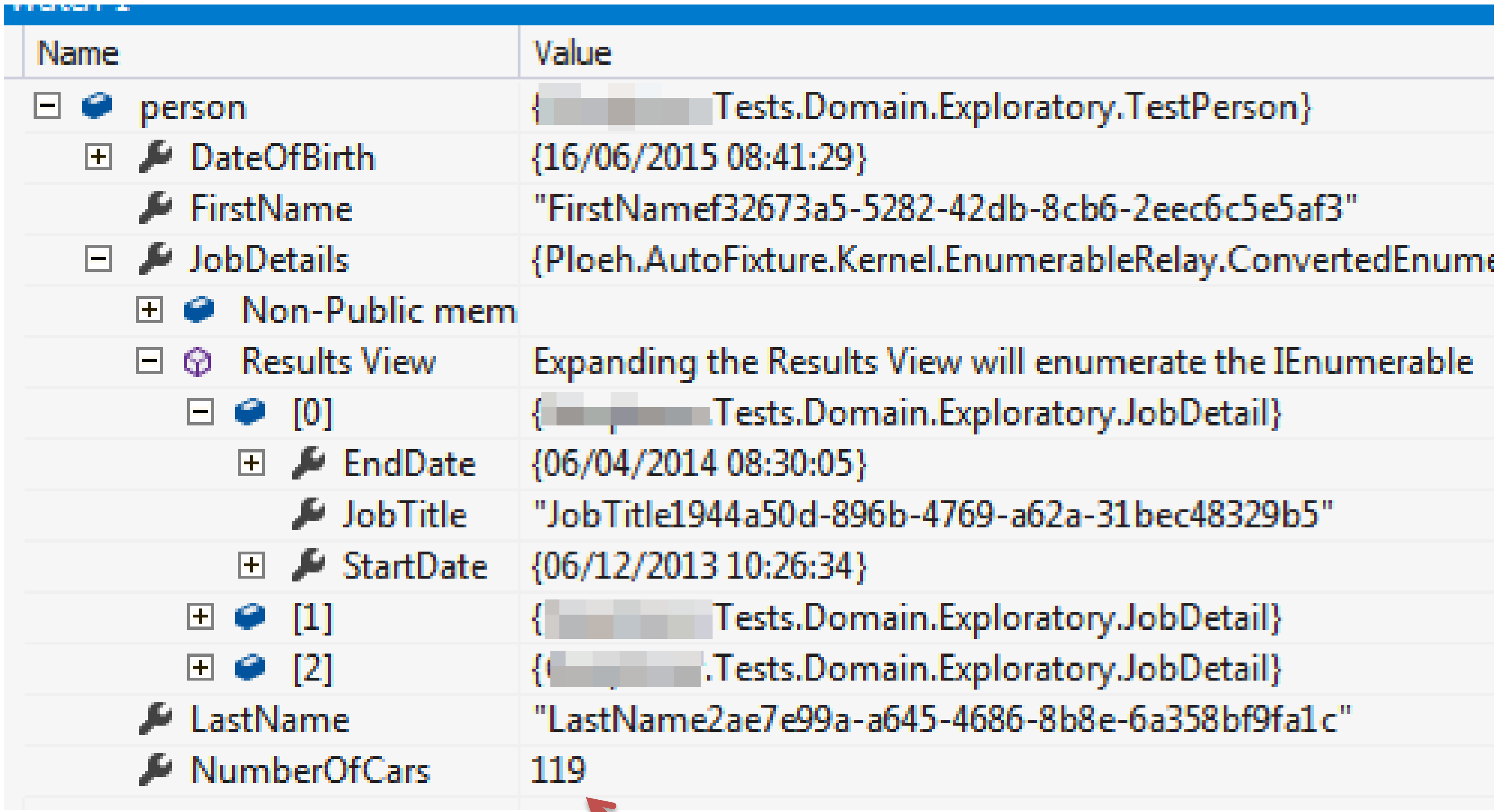
- Test data generator
- Automocking container
- Takes care of NullReferenceExceptions
- Eliminates a lot of work in test setup

* By Mark Seemann

Autofixture Example: Data Generation



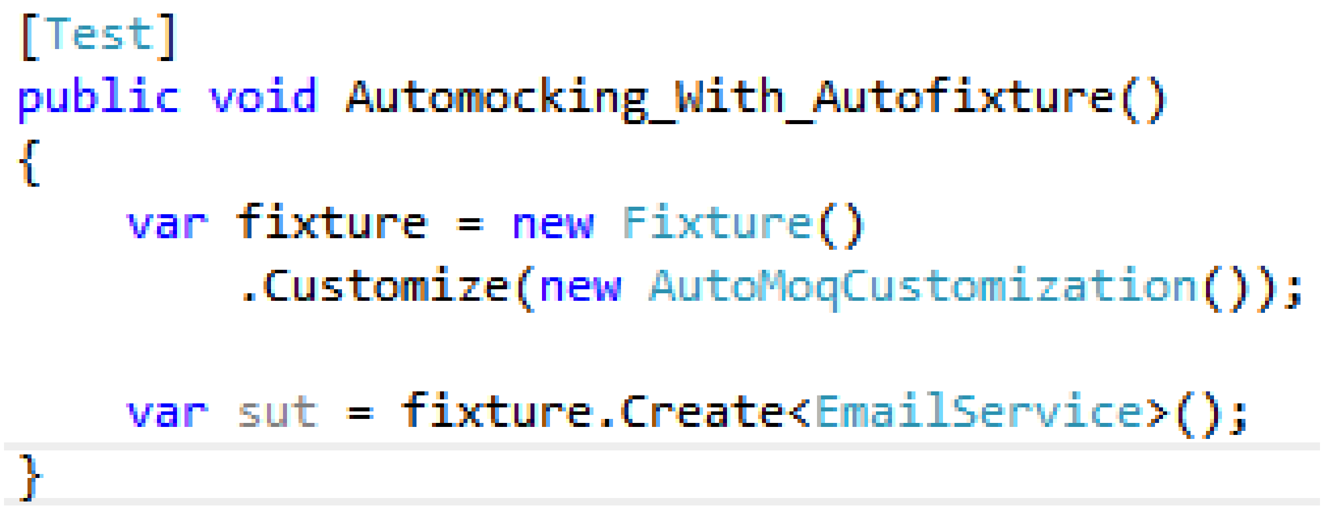
```
[Test]
public void Autofixture_Example()
{
    var fixture = new Fixture();

    var person = fixture.Create<TestPerson>();
}

 class TestPerson
 {
     public String FirstName { get; set; }
     public String LastName { get; set; }
     public int NumberOfCars { get; set; }
     public DateTime DateOfBirth { get; set; }
     public IEnumerable<JobDetail> JobDetails { get; set; }
 }
 class JobDetail
 {
     public String JobTitle { get; set; }
     public DateTime StartDate { get; set; }
     public DateTime EndDate { get; set; }
 }
```



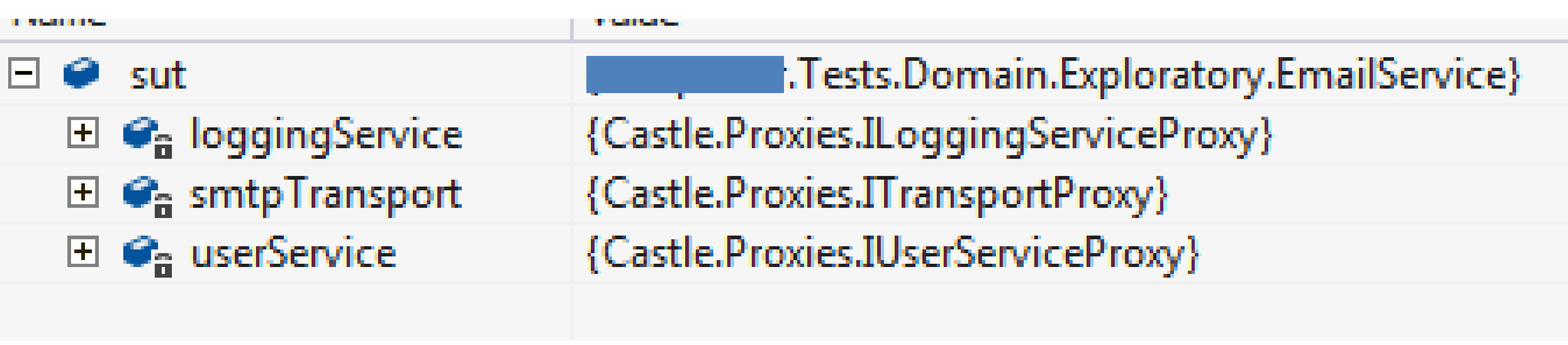
```
Name                          Value
⊟ person                      {[redacted]Tests.Domain.Exploratory.TestPerson}
  ⊞ DateOfBirth               {16/06/2015 08:41:29}
    FirstName                 "FirstNamef32673a5-5282-42db-8cb6-2eec6c5e5af3"
  ⊟ JobDetails                {Ploeh.AutoFixture.Kernel.EnumerableRelay.ConvertedEnume
    ⊞ Non-Public mem
    ⊟ Results View            Expanding the Results View will enumerate the IEnumerable
      ⊟ [0]                   {[redacted]Tests.Domain.Exploratory.JobDetail}
        ⊞ EndDate             {06/04/2014 08:30:05}
          JobTitle            "JobTitle1944a50d-896b-4769-a62a-31bec48329b5"
        ⊞ StartDate           {06/12/2013 10:26:34}
      ⊞ [1]                   {[redacted]Tests.Domain.Exploratory.JobDetail}
      ⊞ [2]                   {[redacted].Tests.Domain.Exploratory.JobDetail}
    LastName                  "LastName2ae7e99a-a645-4686-8b8e-6a358bf9fa1c"
    NumberOfCars              119
```

Some sort of MAGIC!!

Autofixture Example: Automocking



```
[Test]
public void Automocking_With_Autofixture()
{
    var fixture = new Fixture()
        .Customize(new AutoMoqCustomization());

    var sut = fixture.Create<EmailService>();
}
```



⊟ sut	[redacted].Tests.Domain.Exploratory.EmailService}
⊞ loggingService	{Castle.Proxies.ILoggingServiceProxy}
⊞ smtpTransport	{Castle.Proxies.ITransportProxy}
⊞ userService	{Castle.Proxies.IUserServiceProxy}

Continuous Integration

- Developers are lazy
- Nobody run my tests
- Automation for the win!
- Every time you check in, tests are executed for you
- Compilation + Test execution = Build Server

Build Server: Our Process

- Stylecop
- Scan JavaScript files for issues
- Check for elevated usernames/passwords in config files
- Compile + Run Tests
- For nightly build add test coverage analysis
- Whoever breaks the build – gets to fix it and to wear a Santa hat
- Don’t get latest or check-in if the build is broken

Build Server: Advantages

- Machine independence
- Static and dynamic analysis (StyleCop, FxCop, nDepend, etc.)
- Saves a lot of time!
- Improves internal software quality
- Defects are identified and fixed quicker

Disadvantages: some developers play chicken and don’t check-in for days

Build Server: Software

- TFS
  - Need to have TFS licence
  - Angle Brackets Tax
- TeamCity
  - Free for small teams/projects
  - Easy to configure
  - Easy to use
- Many other CI Servers I have not tried

Reflection In Tests*

- Check if DI container can create instances of all controllers in MVC project
- Check if DI controller can create instances of all Command Handlers
- Check if all controllers depend only on interfaces
- Check if all objects have no more than 5 dependencies

*Some examples available in my blog: http://tech.trailmax.info

Reflection In Tests

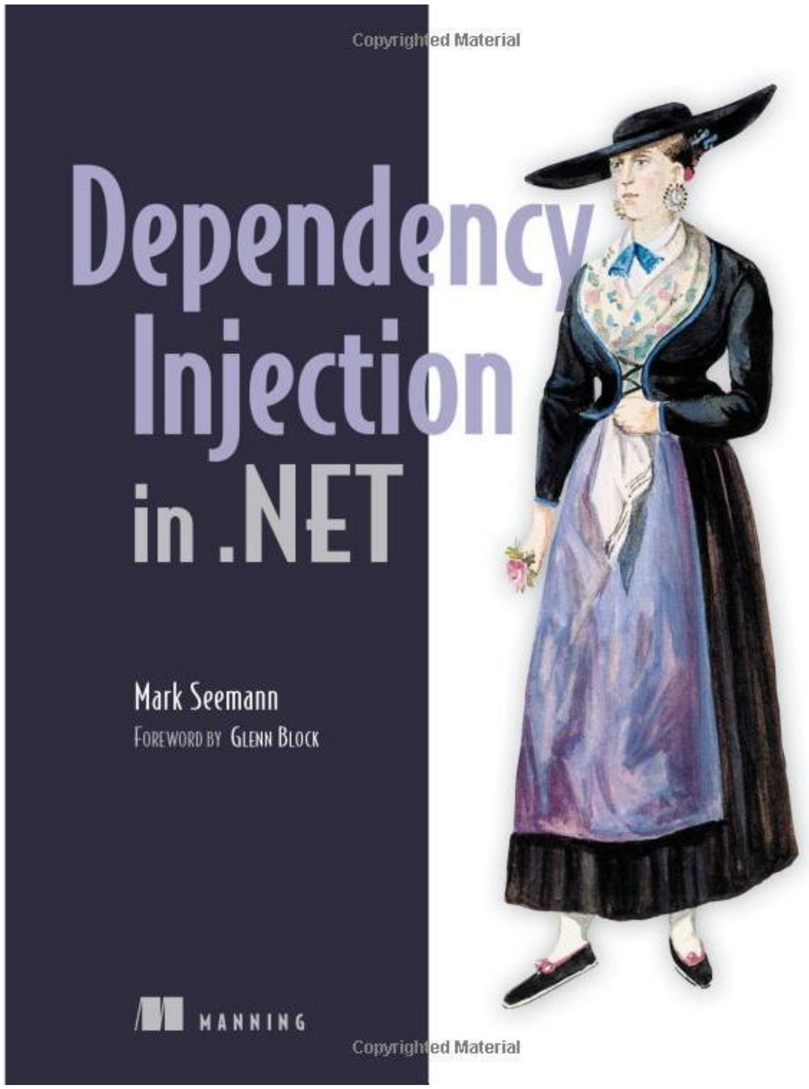
- Not a unit test? I don’t care!
- Not unit tests in strict meaning of “unit”
- High value tests
- Can be interpreted as a lot of unit tests crammed into one execution

Reflection In Tests: Bad

- If you try to access/modify private member
- Breaks if internal implementation changes
- Your code is not testable – redesign!

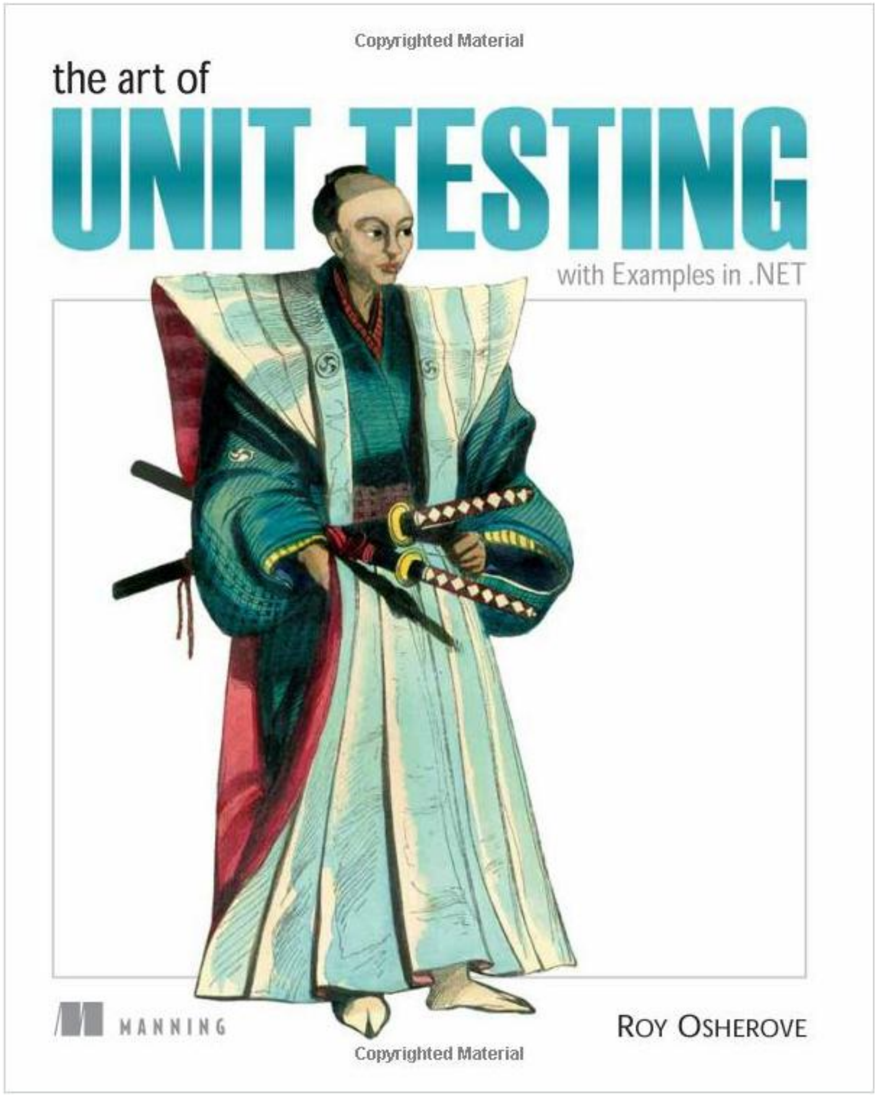
One exception: if you are working with restrictive framework and need to simulate production conditions (i.e. HTTP request)

Books Review: DI



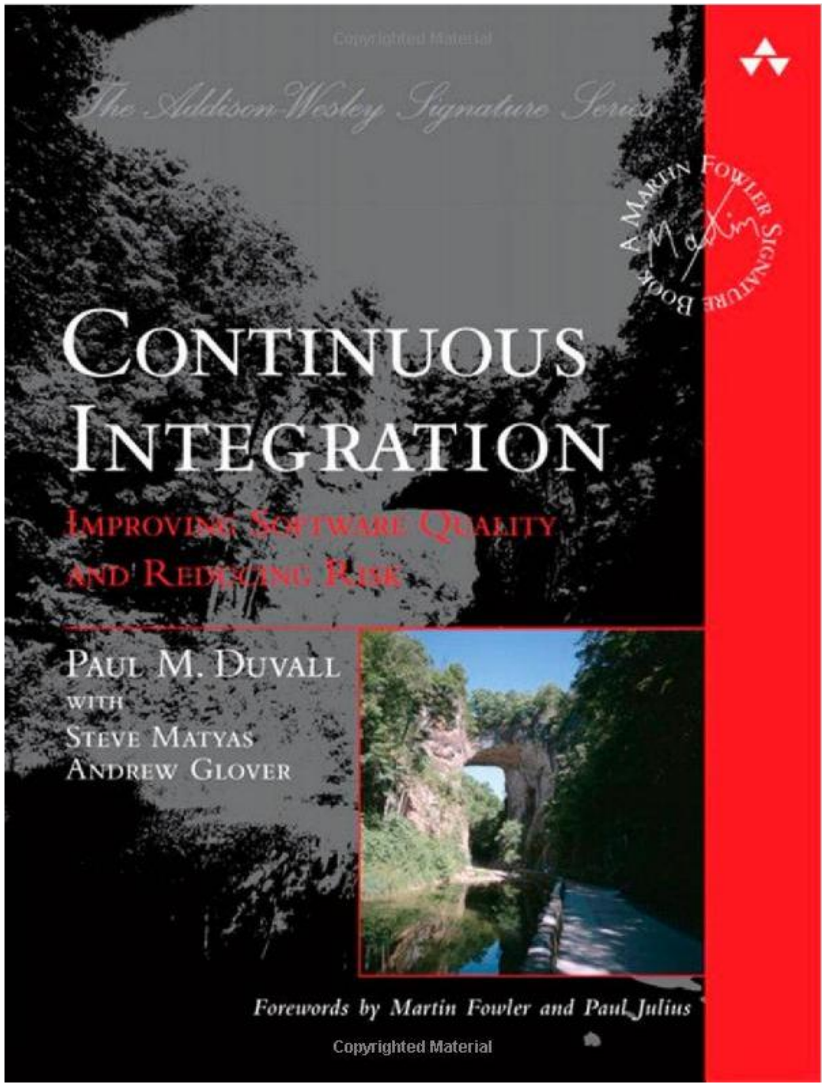
- Must read if you already use DI
- Great starting point
- Sometimes can be confusing for non-DI person
- Heavy

Books Review: Unit Tests



- Simply great book
- Loads of examples
- Many concepts explained
- Testing patterns
- Must read if just starting with tests

Books Review: CI


Copyrighted Material
The Addison-Wesley Signature Series
A Martin Fowler Signature Book
Continuous Integration
Improving Software Quality and Reducing Risk
Paul M. Duvall
with
Steve Matyas
Andrew Glover
Forewords by Martin Fowler and Paul Julius
Copyrighted Material

- Great for managers
- A lot of principles applied to business
- Concepts are explained many times over
- Read first half if introducing CI in your team

You are so awesome! Can I work with you?

We are looking for a good developer (or two) to join our team

- C#, MVC4(5), SQL Server, EF5(6), Azure
- VS2012(3), Resharper

Talk to me after for more details

Questions?

There is no such thing as a silly question!