Breaking Dependencies to Allow Unit Testing

Steve Smith
Ardalis.com
@ardalis

Tweet Away!

- Live Tweeting and Photos are **encouraged**
- Questions and Feedback are welcome
- Use #StirTrek (so I'll be sure to see them) or mention @ardalis

Online Training

DevIQ
Upgrade Your Brain.

ASP.NET Core
Quick Start
ASP.Net

See me after for:

- 1-month free Pluralsight pass
- 50% off ASP.NET Core Quick Start

PLURALSIGHT

Pair Programming	Beginner	2h 29m	7 Apr 2016
Domain-Driven Design Fundamentals	Intermediate	4h 16m	24 Jun 2014
Refactoring Fundamentals	Intermediate	8h 1m	13 Dec 2013
Creating N-Tier Applications in C#, Part 2	Intermediate	1h 40m	30 Dec 2012
Creating N-Tier Applications in C#, Part 1	Intermediate	2h 1m	16 Jul 2012
Kanban Fundamentals	Beginner	1h 31m	12 Feb 2012
Web Application Performance and Scalability Testing	Intermediate	3h 19m	26 Jul 2011
Design Patterns Library	Intermediate	15h 38m	9 Sep 2010
SOLID Principles of Object Oriented Design	Intermediate	4h 8m	9 Sep 2010

Questions up Front

What kinds of *dependencies* in your code cause you the most pain today?

Legacy Code

“To me, legacy code is simply code without tests.”

Michael Feathers

Working Effectively with Legacy Code

Unit Testing (Legacy) Code is…

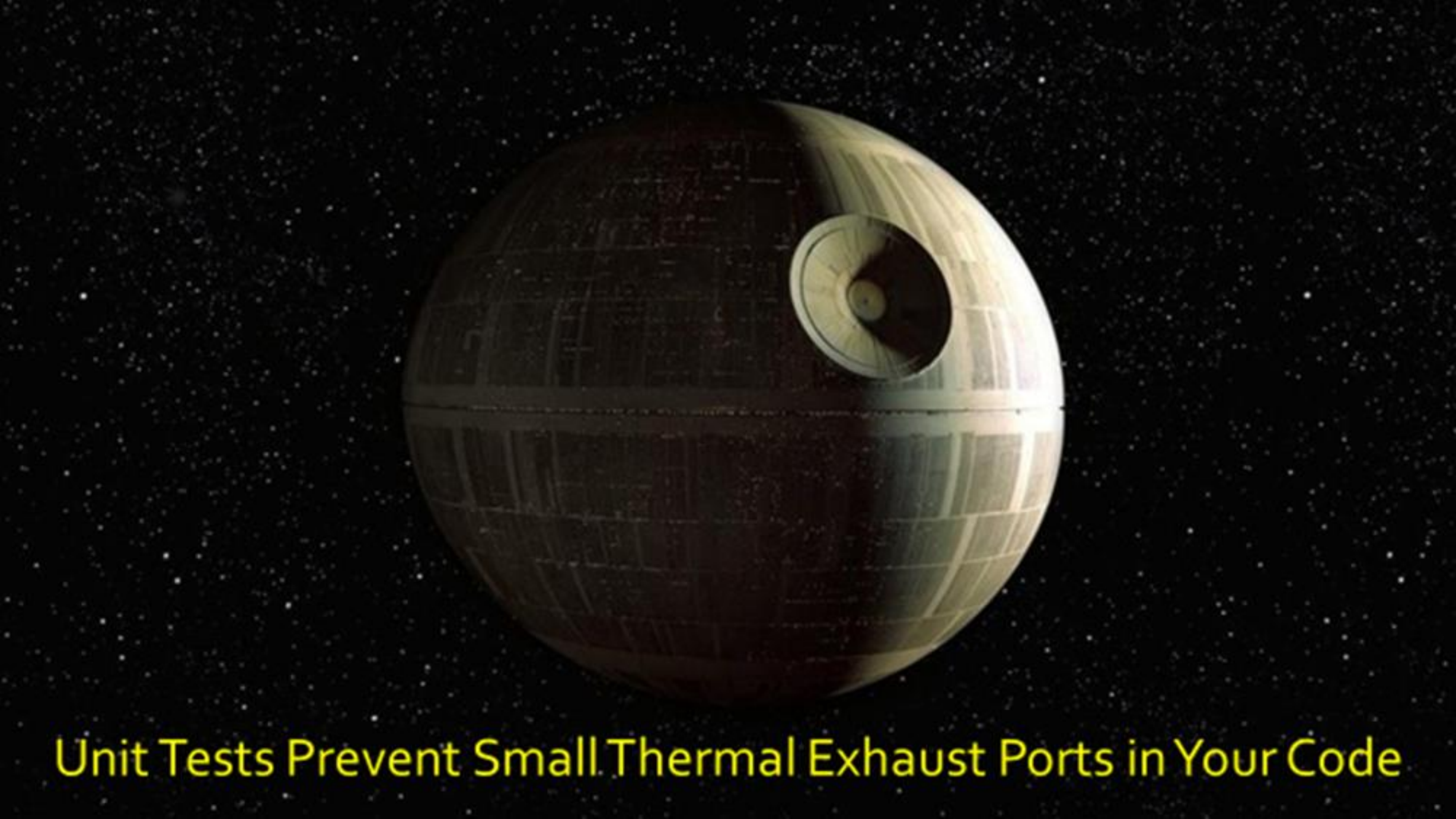
Here's (Mostly) Why...

Hollywood made a whole movie about it...

But let’s back up…

- Why Unit Tests?
- Why not just use other kinds of tests?
- What are dependencies?

- How do we break these dependencies?



Unit Tests Prevent Small Thermal Exhaust Ports in Your Code

Unit Test Characteristics

- Test a single unit of code
  - A method, or at most, a class
- Do not test Infrastructure concerns
  - Database, filesystem, etc.
- Do not test “through” the UI
  - Just code testing code; no screen readers, etc.

Unit Tests are (should be) FAST

- No dependencies means 1000s of tests per second
- Run them *constantly*

Unit Tests are SMALL

- Testing one thing should be simple
  - If not, can it be made simpler?
- Should be quick to write

Test Naming

- Descriptive And Meaningful Phrases (DAMP)

- Name Test Class: ClassNameMethodName
- Name Test Method: DoesSomethingGivenSomething

Or

- Name Test Class: ClassNameMethodName*Should*
- Name Test Method: DoSomethingGivenSomething

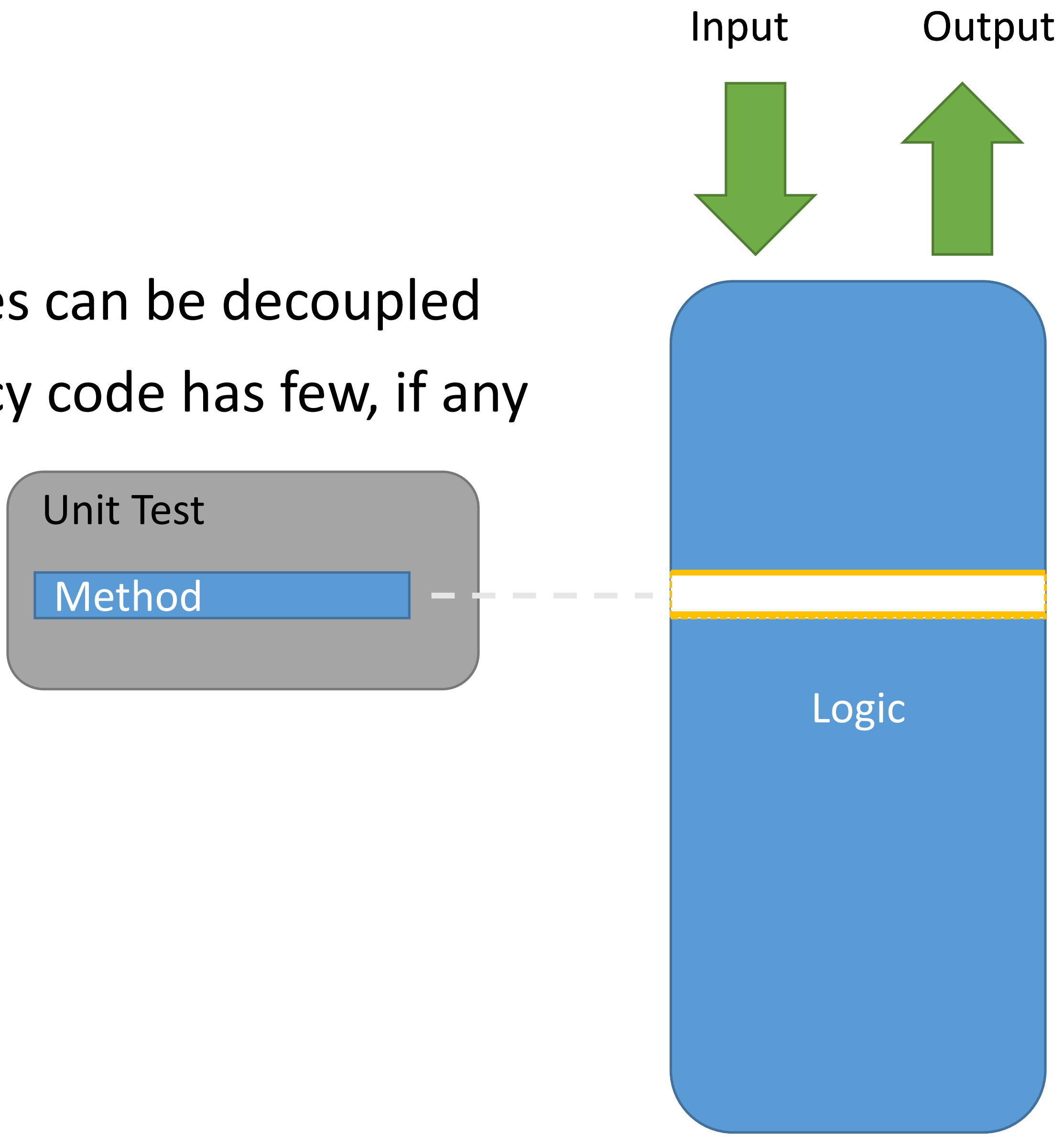
http://ardalis.com/unit-test-naming-convention

Poor Test Naming Approaches

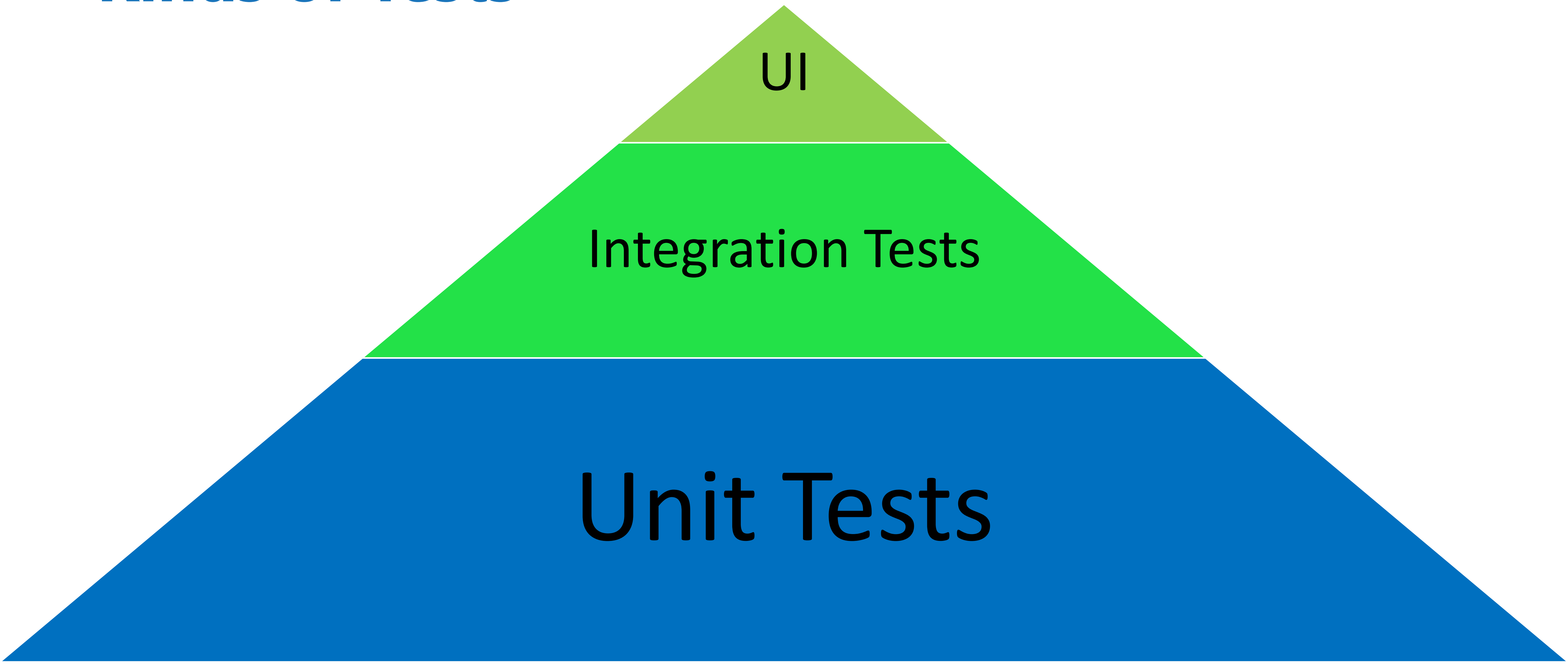
- CustomerTests.cs
- Test1()
- Test2()

Seams

- Represent areas of code where pieces can be decoupled
- Testable code has many seams; legacy code has few, if any



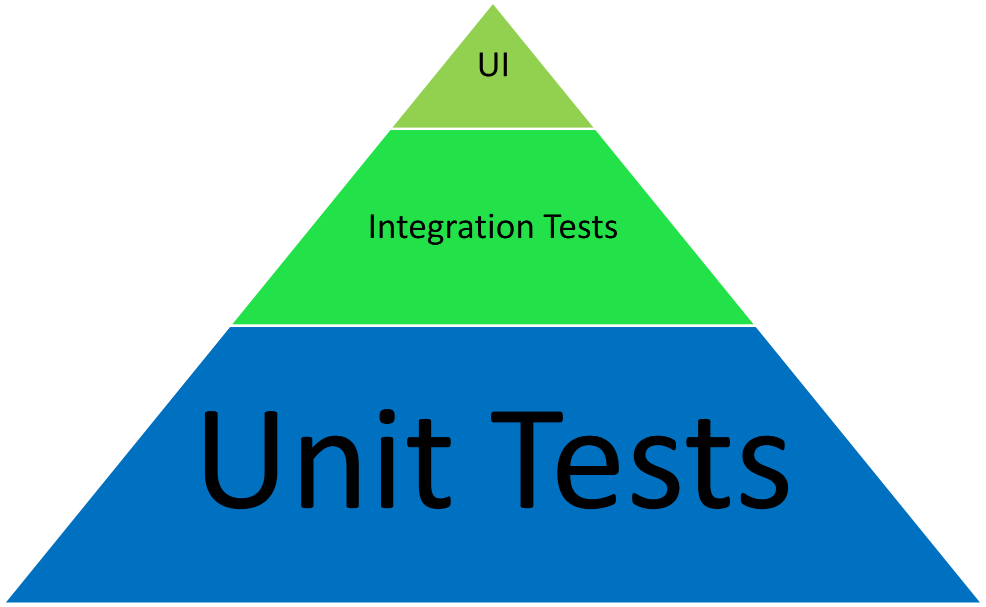
Kinds of Tests



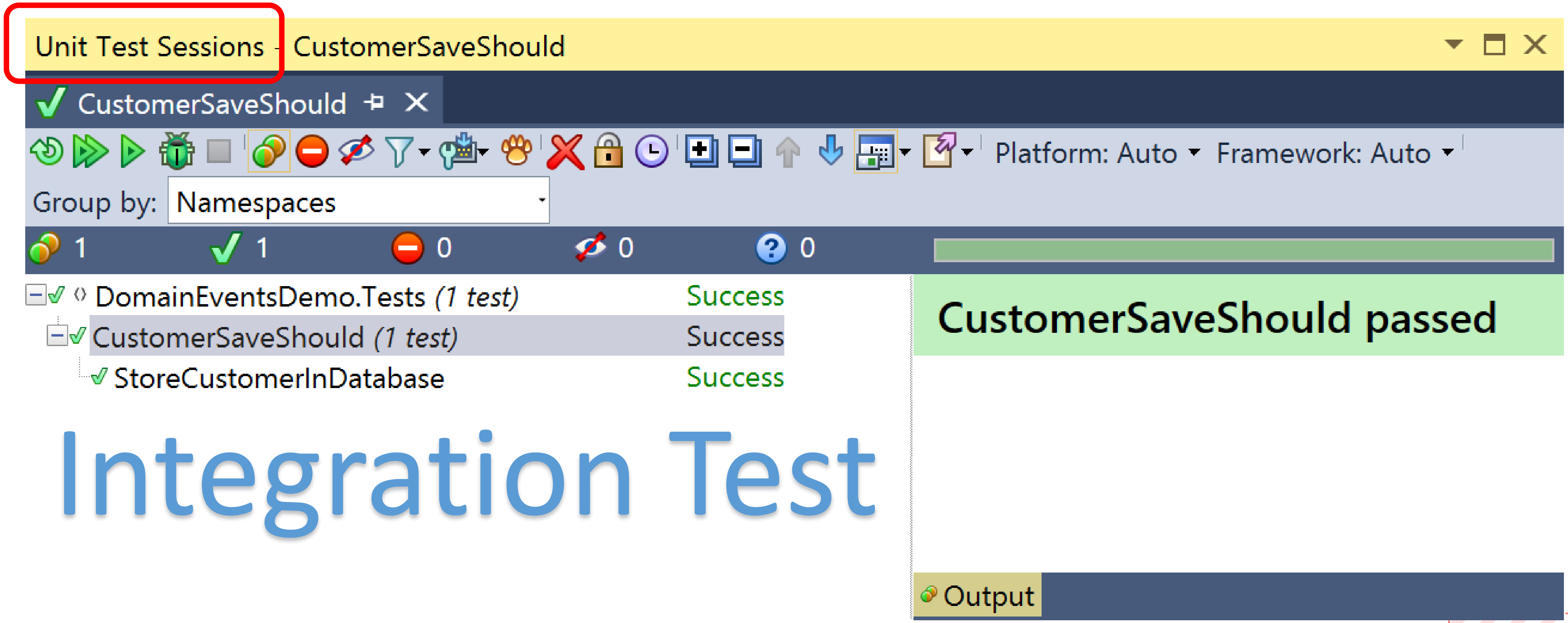
http://martinfowler.com/bliki/TestPyramid.html

Ask yourself:

- Can I test this scenario with a Unit Test?
  - Can I test it with an Integration Test?
    - Can I test it with an automated UI Test?



Don't believe your test runner…



Unit Test Sessions - CustomerSaveShould

✓ CustomerSaveShould

Group by: Namespaces

1 ✓ 1 0 0 0

```
DomainEventsDemo.Tests (1 test)      Success
  CustomerSaveShould (1 test)        Success
    StoreCustomerInDatabase          Success
```

CustomerSaveShould passed

Integration Test

Unit Test?

- Requires a database or file?
- Sends emails?
- Must be executed through the UI?

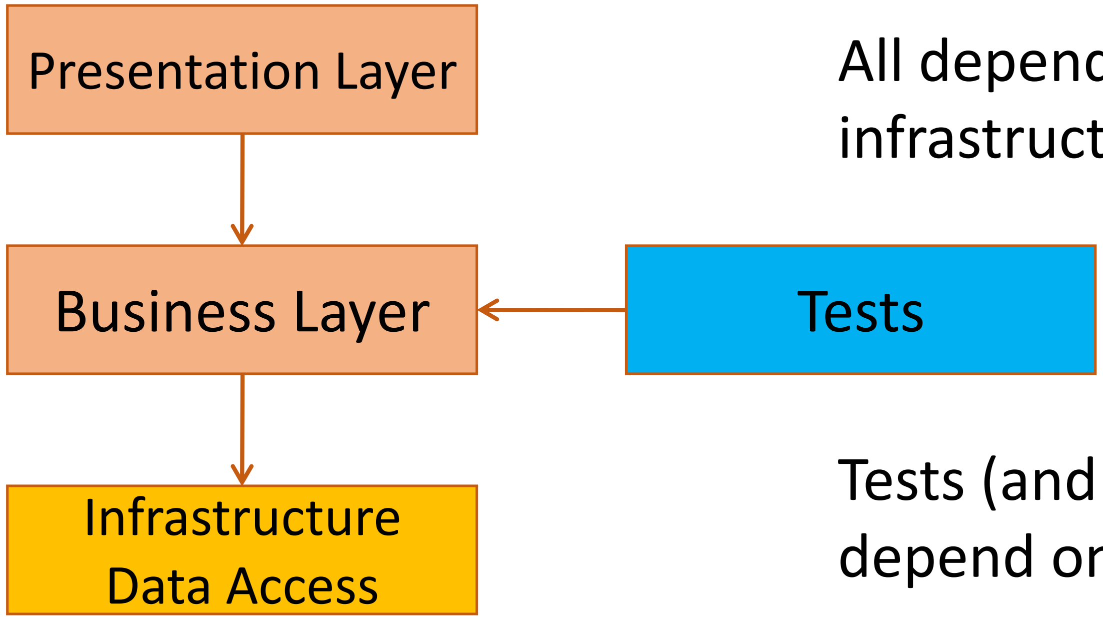
Not a unit test

Unit tests are great, but not sufficient!

Unit tests are great, but not sufficient!

2 UNIT TESTS, 0 INTEGRATION TESTS
via reddit.com/r/programmerhumor

Dependencies and Coupling



All dependencies point toward infrastructure

Tests (and everything else) now depend on Infrastructure

Dependency Inversion Principle

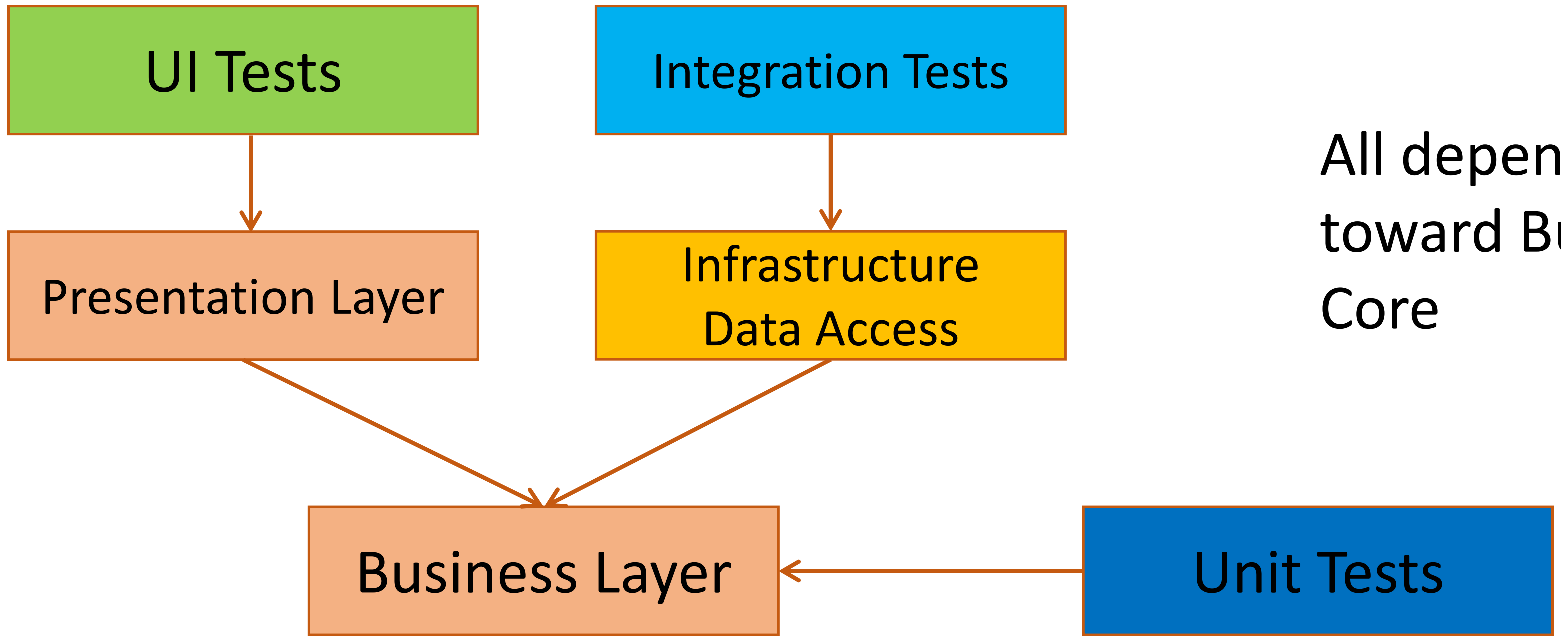
High-level modules should not depend on low-level modules. Both should depend on abstractions.

Abstractions should not depend on details. Details should depend on abstractions.

Agile Principles, Patterns, and Practices in C#

DEPENDENCY INVERSION
Would you solder a lamp directly to the electrical wiring in a wall?

Depend on *Abstractions*



All dependencies point toward Business Logic / Core

Inject Dependencies

- Classes should follow Explicit Dependencies Principle
  - http://deviq.com/explicit-dependencies-principle
- Prefer Constructor Injection
  - Classes cannot be created in an invalid state

https://flic.kr/p/5QsGnB

Common Dependencies to Decouple

- Database
- File System
- Email
- Web APIs / Web Services
- System Clock
- Configuration
- Thread.Sleep
- Random

Tight Couplers: Statics and new

- Avoid static cling
  - Calling static methods with side effects
- Remember: new is glue
  - Avoid gluing your code to a specific implementation
  - Simple types and value objects usually OK

http://ardalis.com/new-is-glue

Coupling Code Smells

- Learn more in my Refactoring Fundamentals course on Pluralsight
  - http://www.pluralsight.com/courses/refactoring-fundamentals

- Coupling Smells introduce tight coupling between parts of a system

Feature Envy

- Characterized by many getter calls
  - Violates the “Tell, Don’t Ask” principle
- Instead, try to package data and behavior together
- Keep together things that change together
  - Common Closure Principle – Classes that change together are packaged together

```
public class Rental
{
  private Movie _movie;

  public decimal GetPrice()
  {
    if (_movie.IsNewRelease)
    {
        if (_movie.IsChildrens)
        {
            return 4;
        }
        return 5;
    }

    if (_movie.IsChildrens)
    {
        return 2;
    }
    return 3;
  }
}
```

```
public class Movie
{
    public bool IsNewRelease { get; set; }
    public bool IsChildrens { get; set; }
    public string Title { get; set; }

    public decimal GetPrice()
    {
        if (IsNewRelease)
        {
            if (IsChildrens)
            {
                return 4;
            }
            return 5;
        }
        if (IsChildrens)
        {
            return 2;
        }
        return 3;
    }
}
```

Law of Demeter

- Or the “Strongly Worded Suggestion of Demeter”
- A Method m on an object O should only call methods on
  - O itself
  - m’s parameters
  - Objects created within m
  - O’s direct fields and properties
  - Global variables and static methods

Law of Demeter

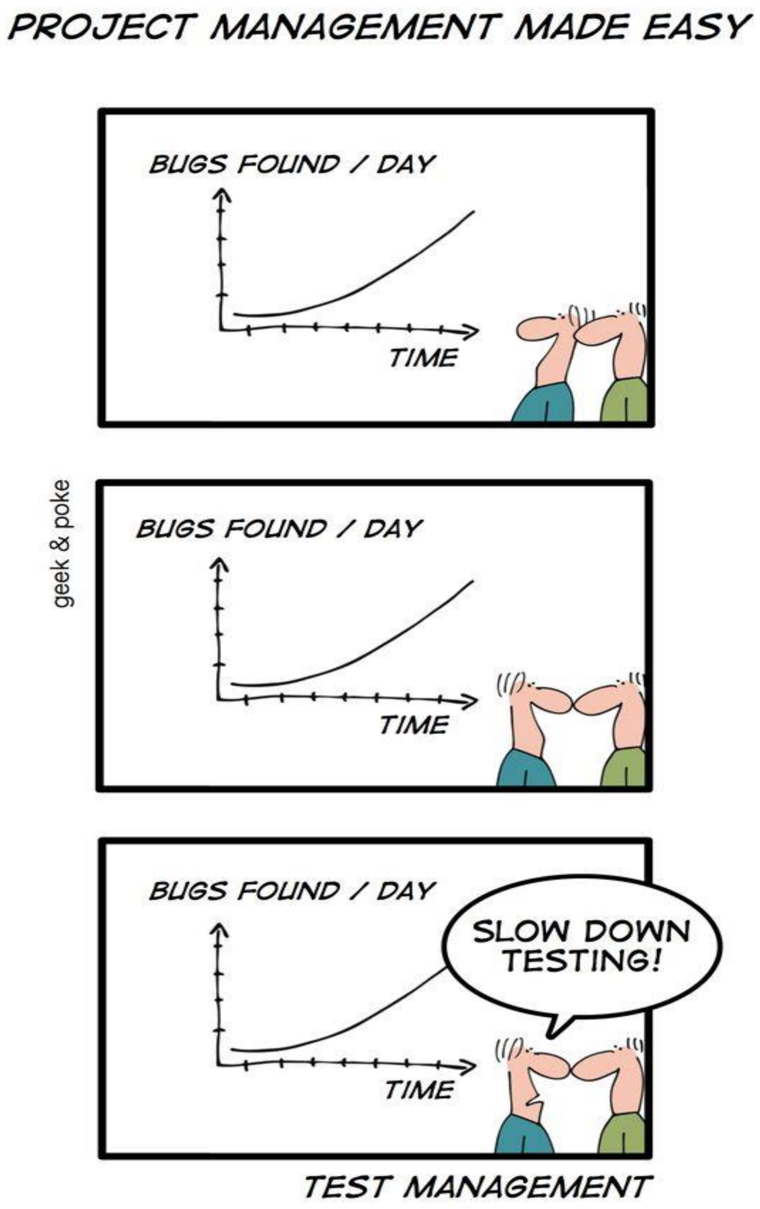
```
public void GetPaidByCustomer(Customer customer)
{
   decimal payment = 12.00;
   var wallet = customer.Wallet;
   if(wallet.Total > payment)
   {
     wallet.RemoveMoney(payment);
   }
   else
   {
     // come back later to get paid
   }
}
```

Law of Demeter

```
public class Customer
{
  private Wallet _wallet;
  public decimal RequestPayment(decimal amount)
  {
    if(_wallet != null && _wallet.Total > amount)
    {
      _wallet.RemoveMoney(amount);
      return amount;
    }
    return 0;
  }
}
```

Law of Demeter

```
public void GetPaidByCustomer(Customer customer)
{
    decimal payment = 12.00;
    decimal amountPaid = customer.RequestPayment(payment);
    if(amountPaid == payment)
    {
      // say thank you and provide a receipt
    }
    else
    {
      // come back later to get paid
    }
}
```



http://geek-and-poke.com

Constructors

Explicit Dependencies Principle

- Methods and classes should require their collaborators as parameters

- Objects should never exist in an invalid state

http://deviq.com/explicit-dependencies-principle/

Constructor Smells

- new keyword (or static calls) in constructor or field declaration
- Anything more than field assignment!
- Database access in constructor
- Complex object graph construction
- Conditionals or Loops

Good Constructors

- Do not create collaborators, but instead accept them as parameters
- Use a Factory for complex object graph creation
- Avoid instantiating fields at declaration

“

IoC Containers are just factories on steroids.

”

Don’t be afraid to use them where they can help

```
public class MoviesController : Controller
{
    private MovieDBContext _db = new MovieDBContext();
    private UserManager _userManager;

    public MoviesController()
    {
        _userManager = new SqlUserManager();
    }
}


[Test]
public void TestSomeMethod()
{
    var controller = new MoviesController();
    // Boom! Cannot create without a database
}
```

```
public class MoviesController : Controller
{
    private MovieDBContext _db;
    private UserManager _userManager;

    public MoviesController(MovieDbContext dbContext,
    userManager)
    {
    _db = dbContext;
        _userManager = userManager;
    }
}

[Test]
public void TestSomeMethod()
{
    var controller = new MoviesController(fakeContext,
 fakeUserManager);
    // continue test here
}
```

```
public class HomeController: Controller
{
    private User _user;
    private string _displayMode;

    public HomeController()
    {
        _user = HttpContext.Current.User;
    _displayMode = Config.AppSettings[“dispMode”];
    }
}


[Test]
public void TestSomeMethod()
{
    var controller = new HomeController();
    // Boom! Cannot create an active HttpContext
    // Also, how to vary display mode when there is one
    // configuration file for the whole test project?
}
```

```
public class HomeController: Controller
{
    private User _user;
    private string _displayMode;

    public HomeController(User user, IConfig config)
    {
        _user = user;
        _displayMode = config.DisplayMode;
    }
}


[Test]
public void TestSomeMethod()
{
    var config = new Config() {DisplayMode="Landscape"};
    var user = new User();
    var controller = new HomeController(user,config);
}
```

Avoid Initialize Methods

- Moving code out of the constructor and into Init()
  - If called from constructor, no different
  - If called later, leaves object in invalid state until called
- Object has too many responsibilities
- If Initialize depends on infrastructure, object will still be hard to test

```
public class SomeService
{
    private IUserRepository _userRepository;
    private DnsRecord _dnsRecord;
    public SomeService(IUserRepository userRepository)
    {
        _userRepository = userRepository;
    }

    public void Initialize()
    {
       string ip = Server.GetAvailableIpAddress();
       _dnsRecord = DNS.Associate("SomeService", ip);
    }
}

[Test]
public void TestSomeMethod()
{
    // I can construct SomeService, but how do I test it
    // when every method depends on Initialize() ?
}
```

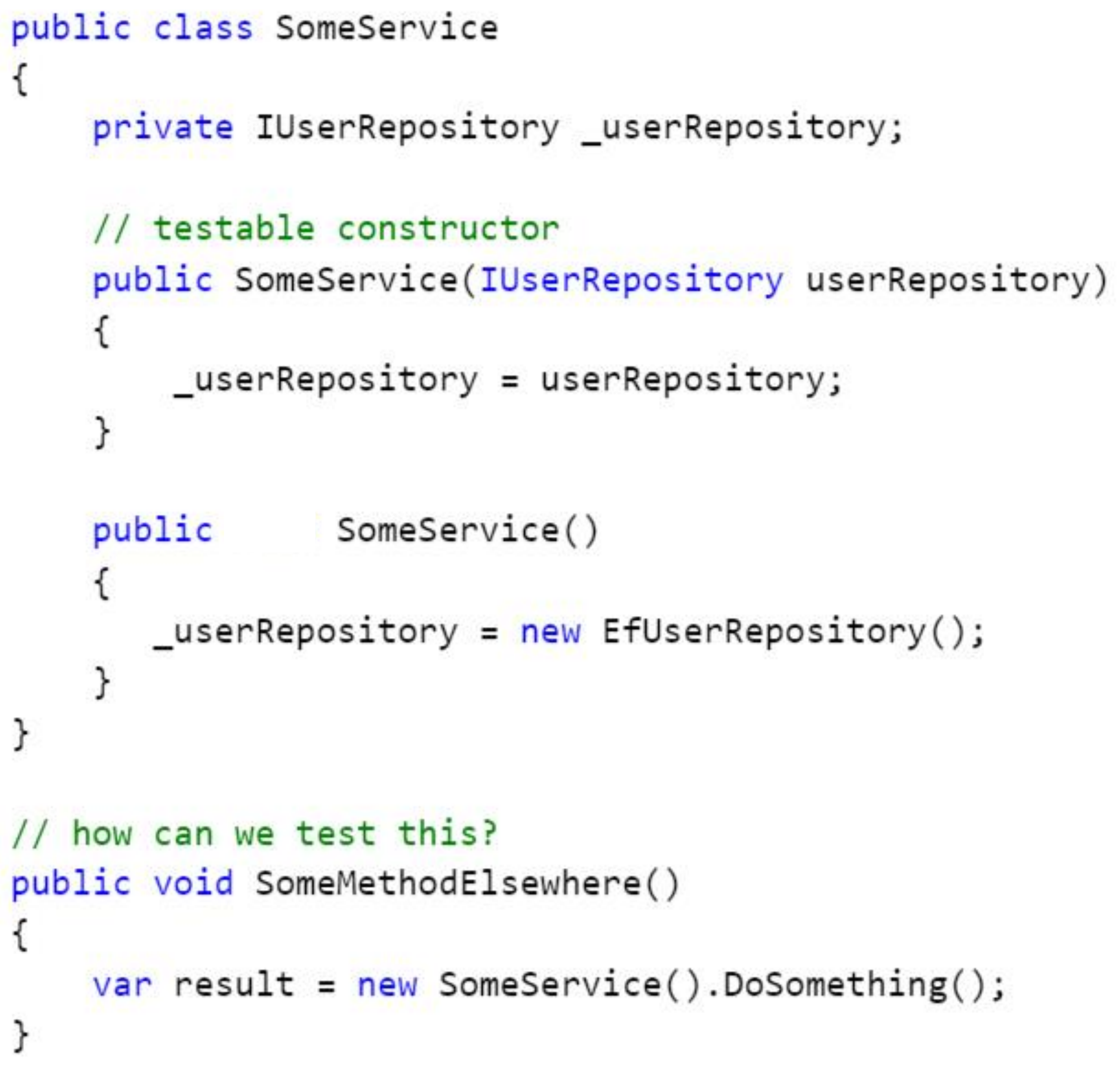
```
public class SomeService
{
    private IUserRepository _userRepository;
    private DnsRecord _dnsRecord;
    public SomeService(IUserRepository userRepository,
    DnsRecord dnsRecord)
    {
        _userRepository = userRepository;
    _dnsRecord = dnsRecord;
    }

// initialize code moved to factory

[Test]
public void TestSomeMethod()
{
    var service = new SomeService(testRepo,
                testDnsRecord);
}
```

“Test” Constructors

- “It’s OK, I’ll provide an “extra” constructor my tests can use!”
- Great! As long as we don’t have to test any other classes that use the other constructor.



```
public class SomeService
{
    private IUserRepository _userRepository;

    // testable constructor
    public SomeService(IUserRepository userRepository)
    {
        _userRepository = userRepository;
    }

    public      SomeService()
    {
        _userRepository = new EfUserRepository();
    }
}

// how can we test this?
public void SomeMethodElsewhere()
{
    var result = new SomeService().DoSomething();
}
```

Avoid Digging into Collaborators

- Pass in the specific object(s) you need
- Avoid using “Context” or “Manager” objects to access additional dependencies
  - Violates Law of Demeter: Context.SomeItem.Foo()
- Suspicious Names: environment, principal, container
- Symptoms: Tests have mocks that return mocks

```
public class TaxCalculator
{
    private TaxTable _taxTable;
// other fields
    public decimal ComputeTax(User user, Invoice invoice)
    {
        var address = user.Address;
        var amount = invoice.Subtotal;
        var rate = _taxTable.GetTaxRate(address);
        return amount * rate;
    }
}

// tests must now create users and invoices
// instead of just passing in address and subtotal amount
```

```
public class TaxCalculator
{
    private TaxTable _taxTable;
// other fields
    public decimal ComputeTax(Address address,
                              decimal subtotal)
    {
        var rate = _taxTable.GetTaxRate(address);
            return subtotal * rate;
    }
}

// API is now more honest about what it actually requires
// Tests are much simpler to write
```

Avoid Global State Access

- Singletons
- Static fields or methods
- Static initializers
- Registries
- Service Locators

Singletons

- Avoid classes that implement their own lifetime tracking
  - GoF Singleton Pattern

- It’s **OK** to have a container manage object lifetime and enforce having only a single instance of a class within your application

```
public class TrainScheduler
{
    public Track FindAvailableTrack()
    {
    // loop through available tracks
        if(TrackStatusChecker.IsAvailable(track))
    // do something

    return track;
    }
}

// tests of FindAvailableTrack now depend on
// TrackStatusChecker, which is a slow web service
```

```
public class TrainScheduler
{
    private TrainStatusCheckerWrapper _wrapper;
    TrainScheduler(TrainStatusCheckerWrapper wrapper)
    {
        _wrapper = wrapper;
    }

    public Track FindAvailableTrack()
    {
        // loop through available tracks
        if(_wrapper.IsAvailable(track))
        // do something

        return track;
    }
}

// tests of FindAvailableTrack now can easily inject a
// wrapper to test the behavior of FindAvailableTrack()
// wrapper could just as easily be an interface
```

Questions?

Or tweet me @ardalis and I'll answer later.

Summary

- Inject Dependencies
  - Remember "New is glue".
- Keep your APIs honest
  - Remember the Explicit Dependencies Principle. Tell your friends.
- Maintain seams and keep coupling loose

Thanks!

- Follow me at @ardalis
- Get a weekly dev tip: ardalis.com/tips
- Check out DevIQ.com for more on these topics
- References
  - http://misko.hevery.com/code-reviewers-guide/
  - Working Effectively with Legacy Code

    by Michael Feathers