Advanced development techniques

Dependency Injection
Test Doubles, Moq

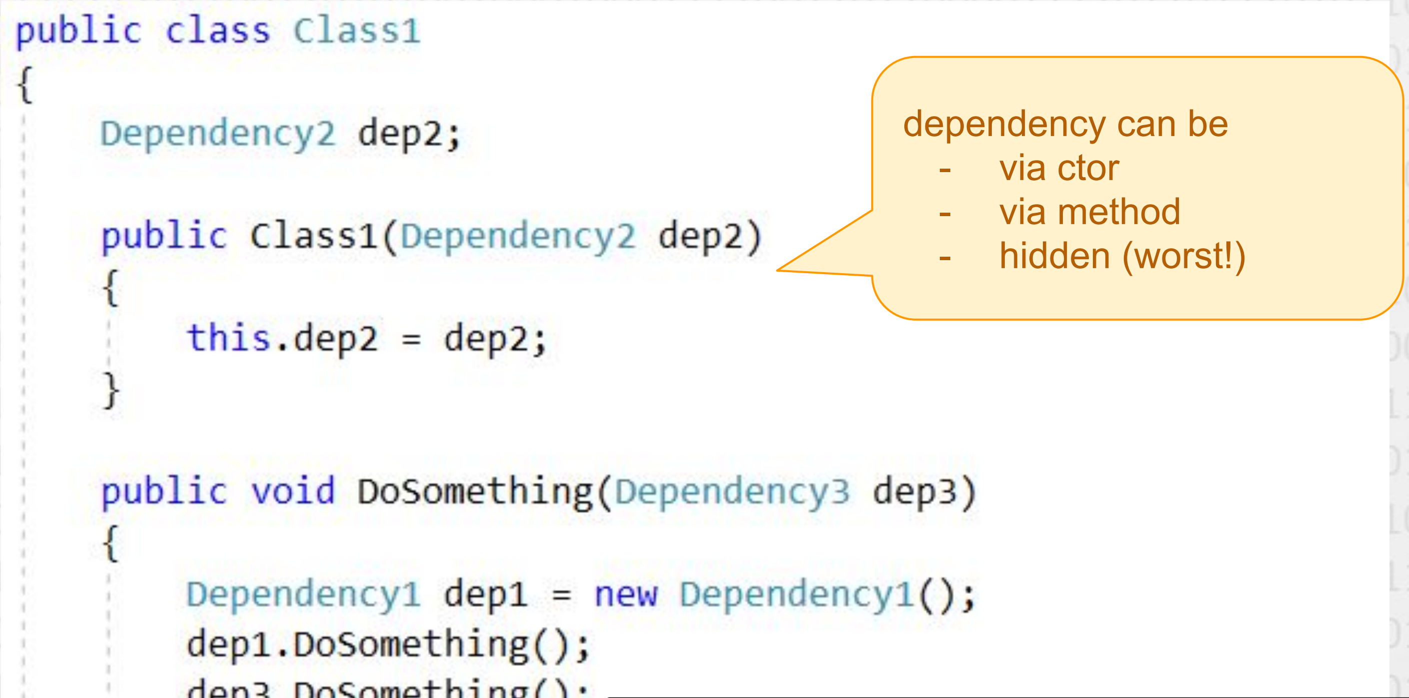
„Testing = I will try it manually”

„The tester will test the code”

It is hard to write a good test!

- **The tests must be fast**
  - Slow tests are impossible to run over and over again □ test will not be executed, bugs will be found later
- **The tests must be independent**
  - Order, timing, etc. must not affect the results
- **Naming convention must be easy-to-read**
  - A good test list is basically a requirement-list that documents the capabilities of the program
- **We must not cover every possible inputs**
  - Examples are good
  - Finding the corner cases are important!
- **Only test a single feature of a single class**
  - Always independent from the live data (database/settings)
  - We can substitute the dependencies too: Dependency Injection + fake dependencies, mocking... (=> Moq)

Warning – not a real unittest!

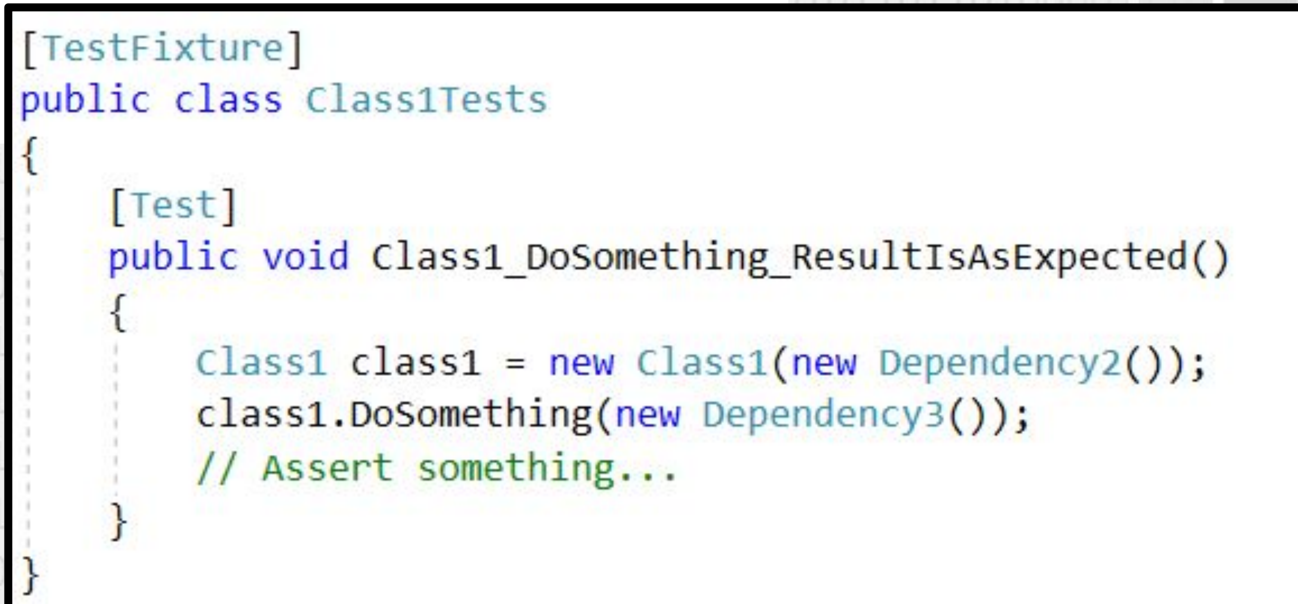


```
public class Class1
{
    Dependency2 dep2;

    public Class1(Dependency2 dep2)
    {
        this.dep2 = dep2;
    }

    public void DoSomething(Dependency3 dep3)
    {
        Dependency1 dep1 = new Dependency1();
        dep1.DoSomething();
        dep3.DoSomething();
    }
}
```

dependency can be
- via ctor
- via method
- hidden (worst!)

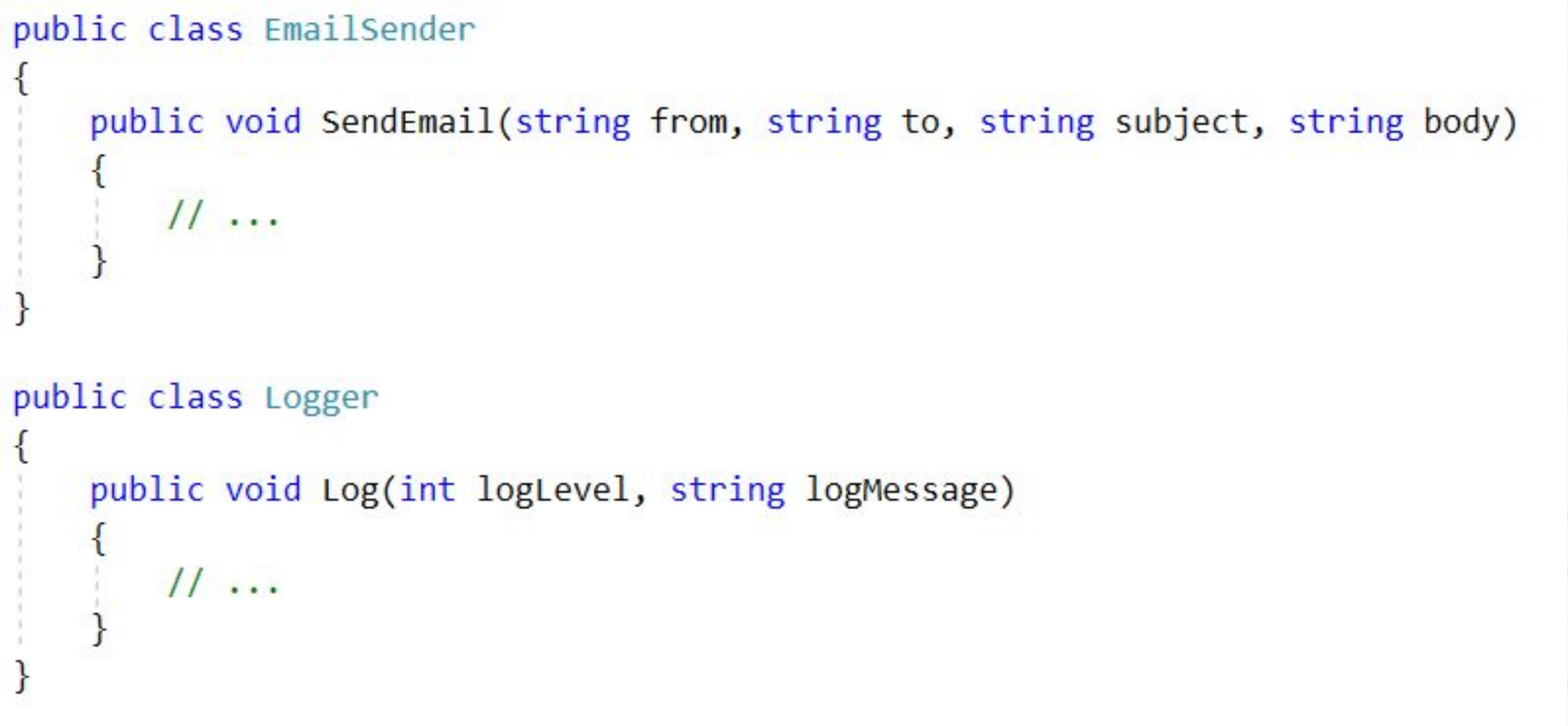
In order to test a feature (method of a class) we have to create the class's or method's dependency as well!!!

If there is 'red test' maybe it is because that the dependency has problems, not our code!



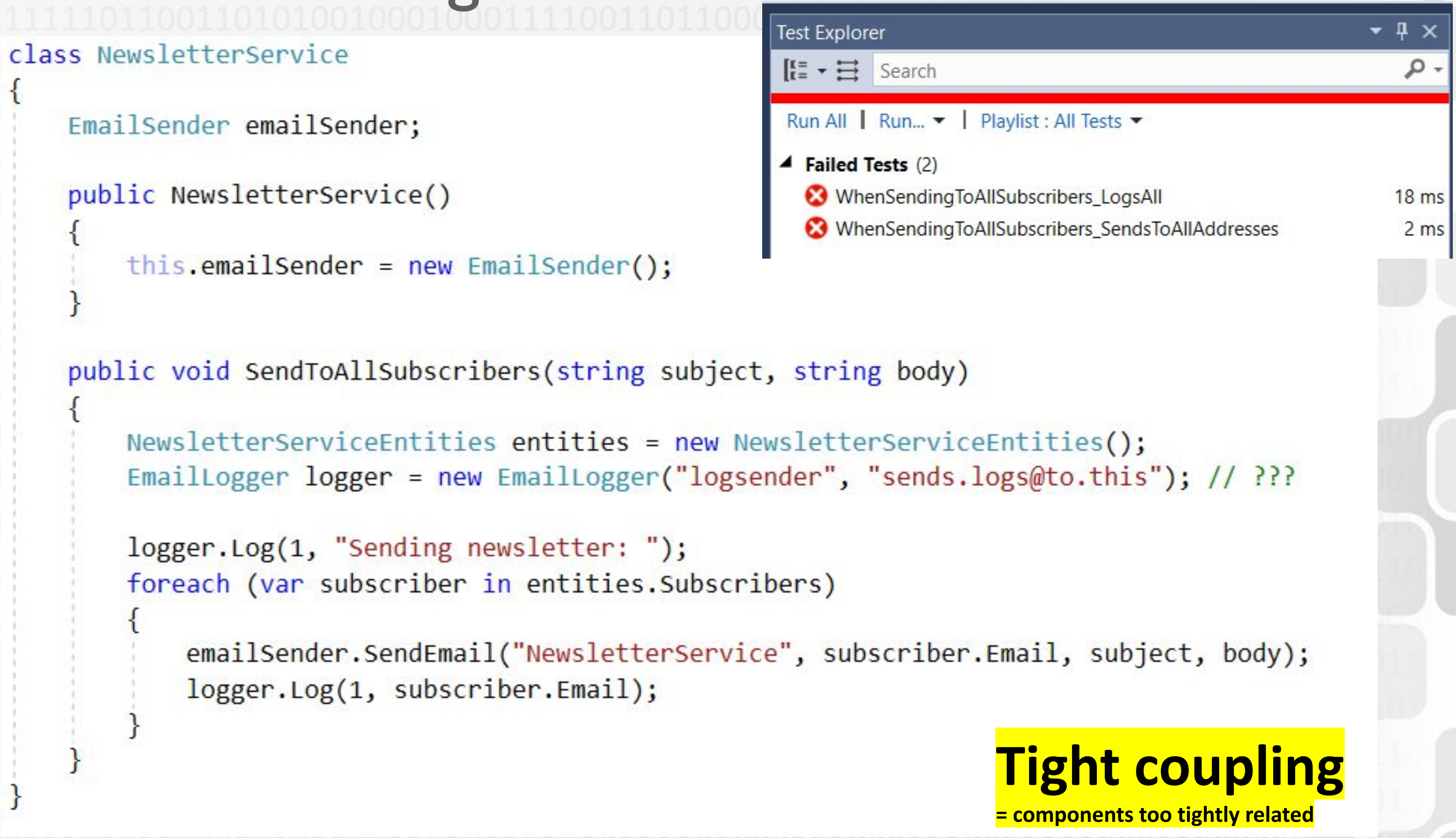
```
[TestFixture]
public class Class1Tests
{
    [Test]
    public void Class1_DoSomething_ResultIsAsExpected()
    {
        Class1 class1 = new Class1(new Dependency2());
        class1.DoSomething(new Dependency3());
        // Assert something...
    }
}
```

Example



```
public class EmailSender
{
    public void SendEmail(string from, string to, string subject, string body)
    {
        // ...
    }
}

public class Logger
{
    public void Log(int logLevel, string logMessage)
    {
        // ...
    }
}
```

Aim: Refactoring



```
class NewsletterService
{
    EmailSender emailSender;

    public NewsletterService()
    {
        this.emailSender = new EmailSender();
    }

    public void SendToAllSubscribers(string subject, string body)
    {
        NewsletterServiceEntities entities = new NewsletterServiceEntities();
        EmailLogger logger = new EmailLogger("logsender", "sends.logs@to.this"); // ???

        logger.Log(1, "Sending newsletter: ");
        foreach (var subscriber in entities.Subscribers)
        {
            emailSender.SendEmail("NewsletterService", subscriber.Email, subject, body);
            logger.Log(1, subscriber.Email);
        }
    }
}
```

Failed Tests (2)

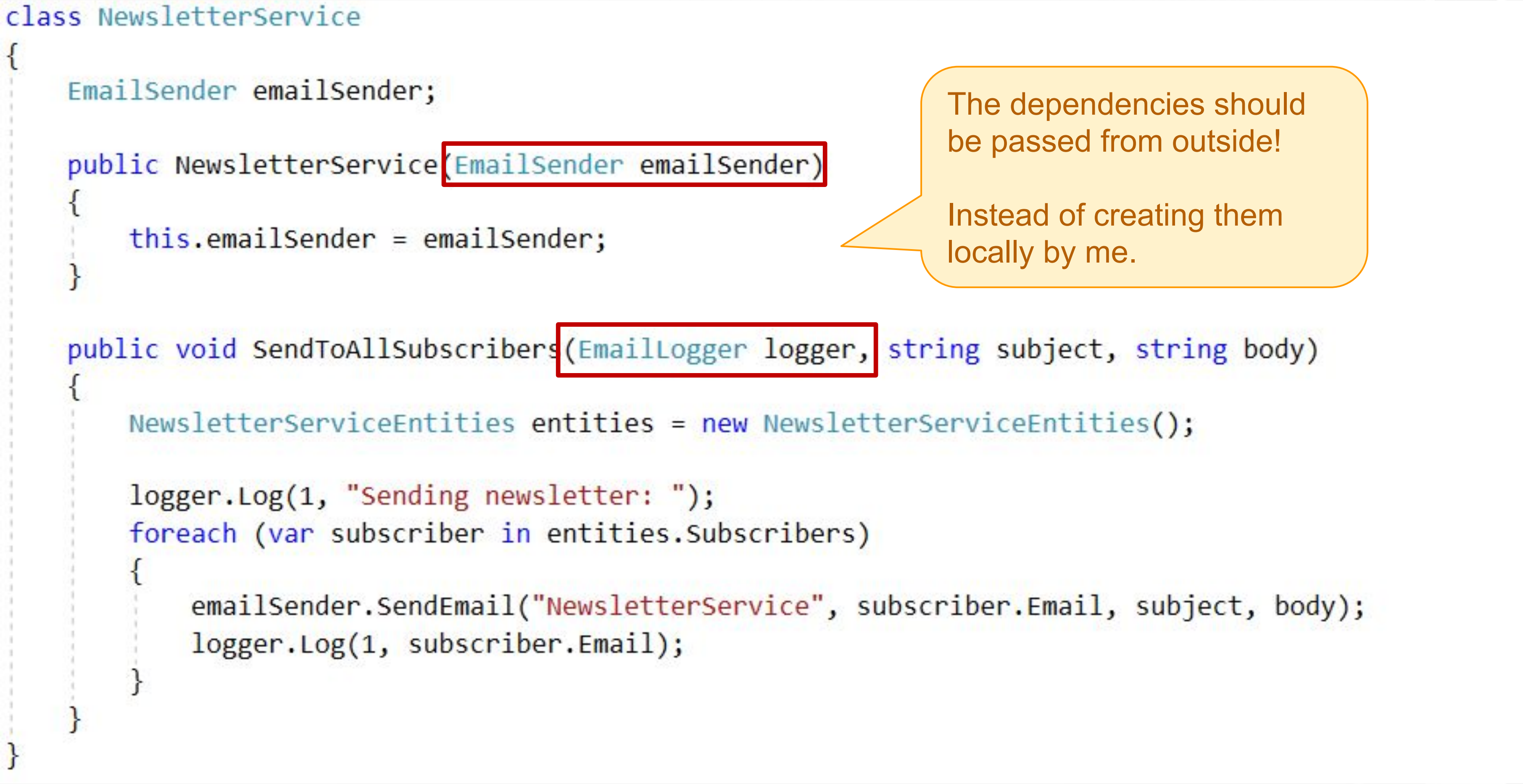
WhenSendingToAllSubscribers_LogsAll	18 ms
WhenSendingToAllSubscribers_SendsToAllAddresses	2 ms

Tight coupling

= components too tightly related

- **Refactoring = changing the structure of the code without modifying the behavior of the class/method/product**
- **Refactoring for testability**

1. Using parameters

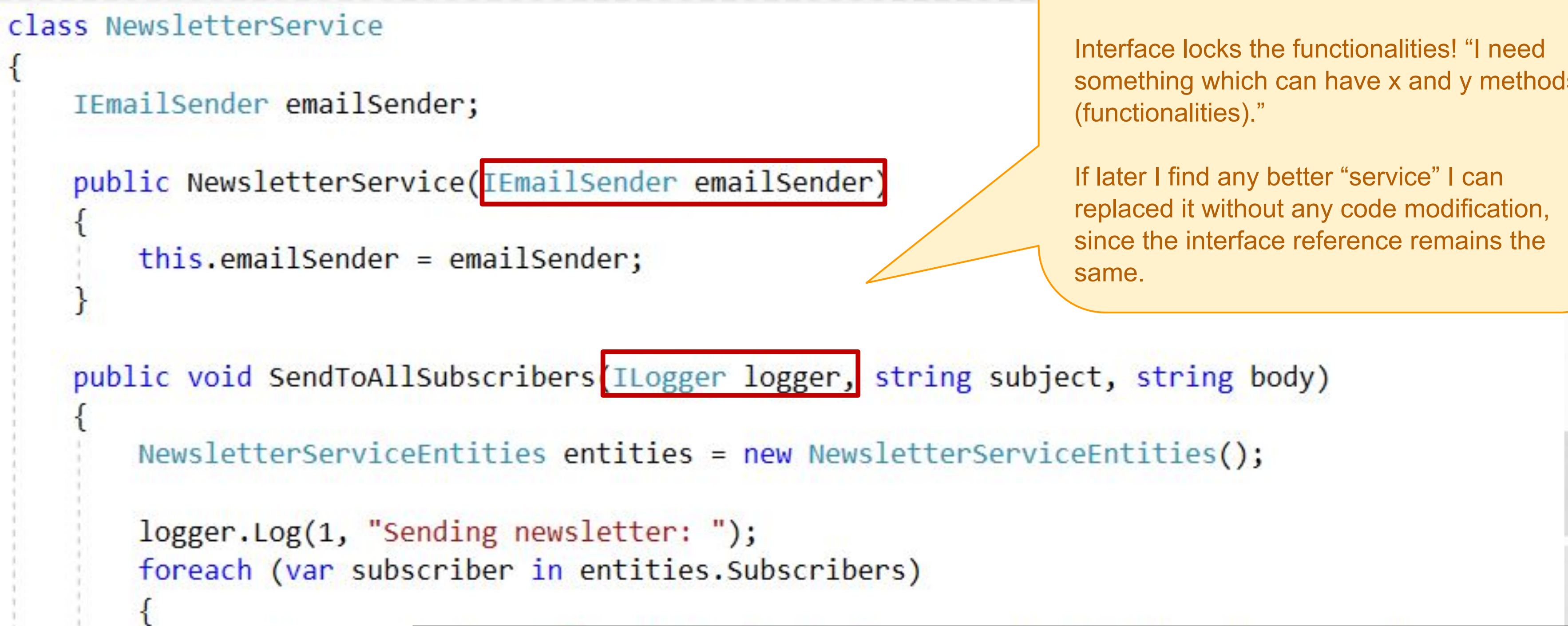


```
class NewsletterService
{
    EmailSender emailSender;

    public NewsletterService(EmailSender emailSender)
    {
        this.emailSender = emailSender;
    }

    public void SendToAllSubscribers(EmailLogger logger, string subject, string body)
    {
        NewsletterServiceEntities entities = new NewsletterServiceEntities();

        logger.Log(1, "Sending newsletter: ");
        foreach (var subscriber in entities.Subscribers)
        {
            emailSender.SendEmail("NewsletterService", subscriber.Email, subject, body);
            logger.Log(1, subscriber.Email);
        }
    }
}
```

The dependencies should be passed from outside!

Instead of creating them locally by me.

2. Interface-typed parameters

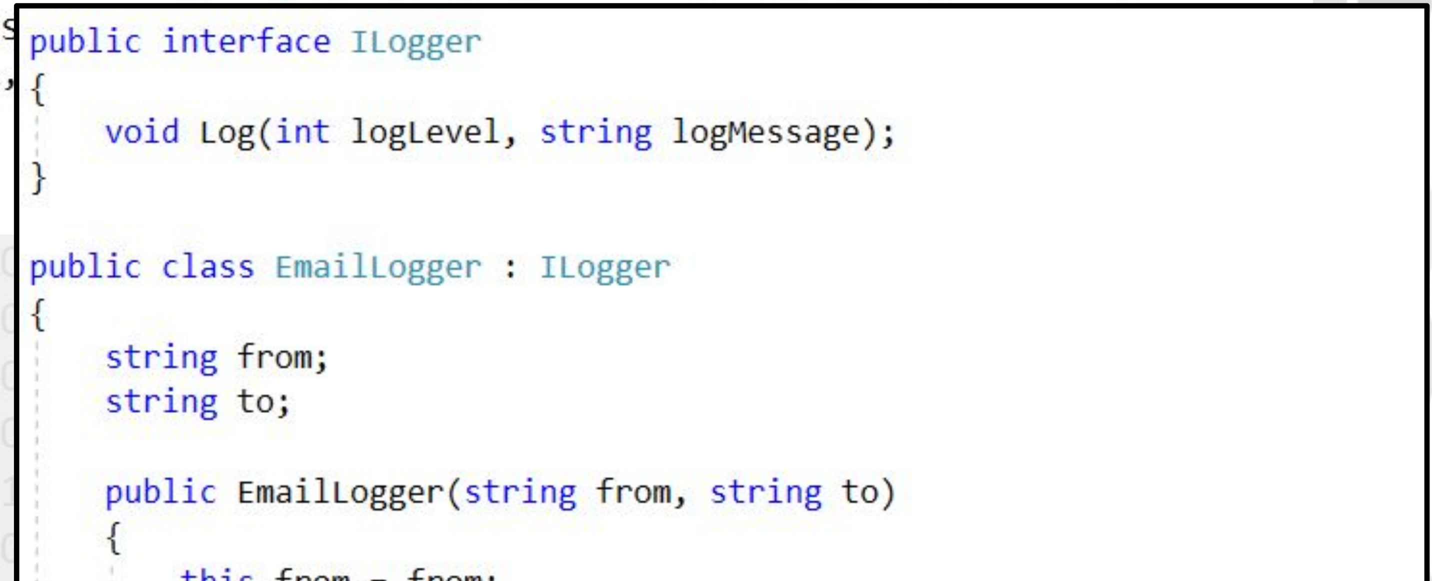


```
class NewsletterService
{
    IEmailSender emailSender;

    public NewsletterService(IEmailSender emailSender)
    {
        this.emailSender = emailSender;
    }

    public void SendToAllSubscribers(ILogger logger, string subject, string body)
    {
        NewsletterServiceEntities entities = new NewsletterServiceEntities();

        logger.Log(1, "Sending newsletter: ");
        foreach (var subscriber in entities.Subscribers)
        {
            emailSender.S
            logger.Log(1,
        }
    }
}
```

Even better to use interface references, not direct classes!

Interface locks the functionalities! “I need something which can have x and y methods (functionalities).”

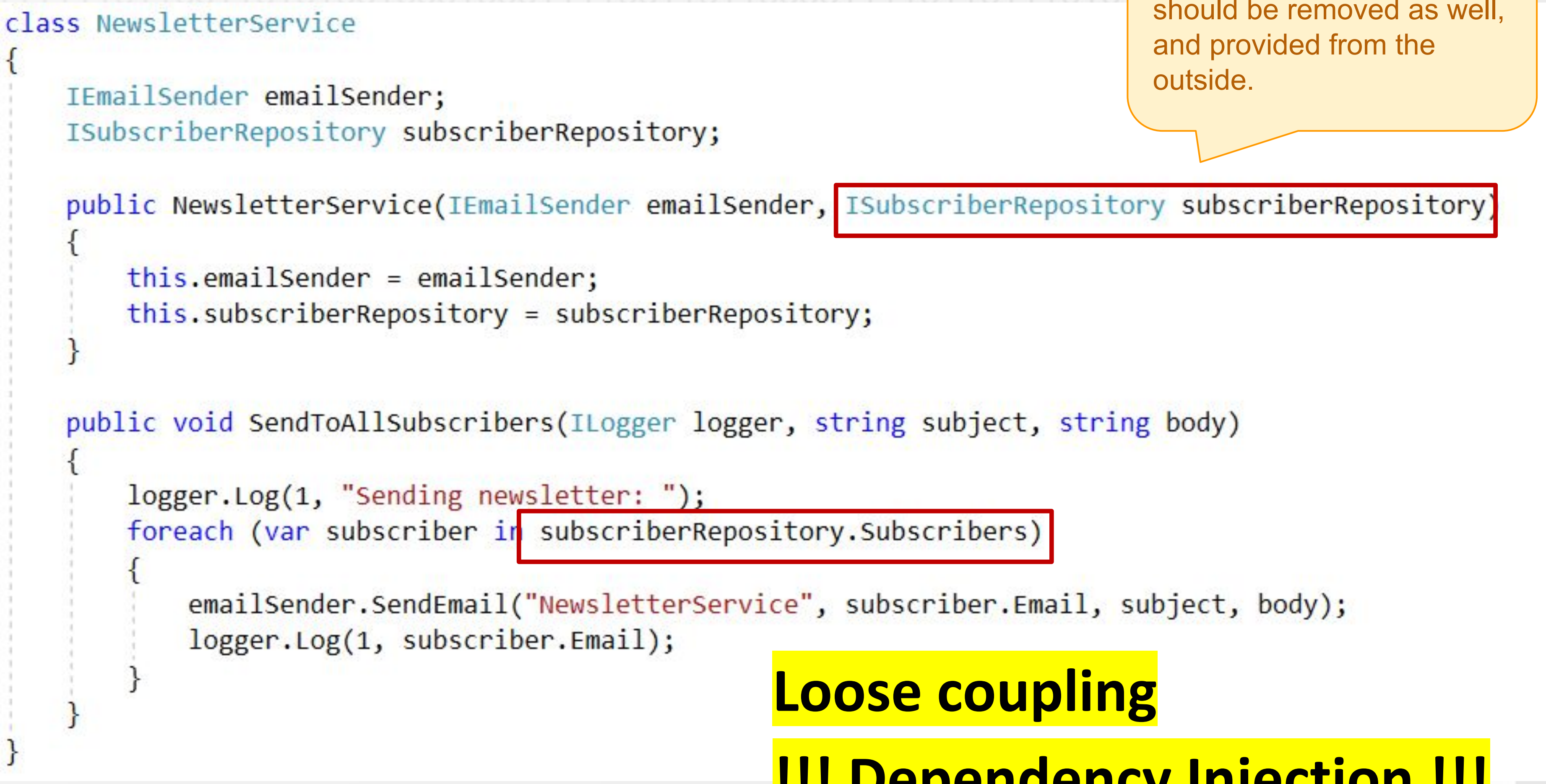
If later I find any better “service” I can replaced it without any code modification, since the interface reference remains the same.



```
public interface ILogger
{
    void Log(int logLevel, string logMessage);
}

public class EmailLogger : ILogger
{
    string from;
    string to;

    public EmailLogger(string from, string to)
    {
        this.from = from;
```

3. Avoid hidden dependencies

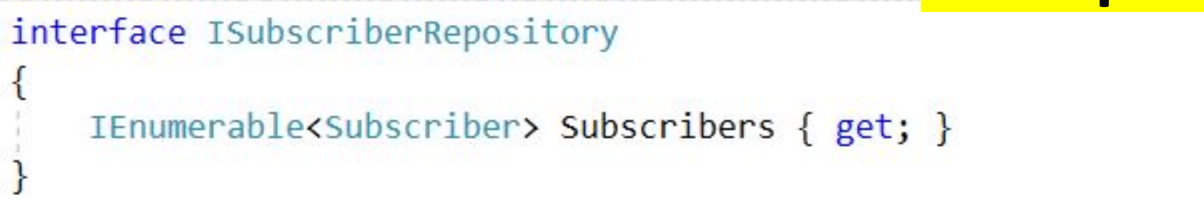
Hidden dependencies should be removed as well, and provided from the outside.



```
class NewsletterService
{
    IEmailSender emailSender;
    ISubscriberRepository subscriberRepository;

    public NewsletterService(IEmailSender emailSender, ISubscriberRepository subscriberRepository)
    {
        this.emailSender = emailSender;
        this.subscriberRepository = subscriberRepository;
    }

    public void SendToAllSubscribers(ILogger logger, string subject, string body)
    {
        logger.Log(1, "Sending newsletter: ");
        foreach (var subscriber in subscriberRepository.Subscribers)
        {
            emailSender.SendEmail("NewsletterService", subscriber.Email, subject, body);
            logger.Log(1, subscriber.Email);
        }
    }
}
```

Loose coupling

!!! Dependency Injection !!!



```
interface ISubscriberRepository
{
    IEnumerable<Subscriber> Subscribers { get; }
}
```

Constructor Injection

```
interface IMyDependency
{
    string DoSomething();
}

class MyClass
{
    private IMyDependency dependency;

    public MyClass(IMyDependency dependency)
    {
        this.dependency = dependency;
    }

    // in methods: dependency.DoSomething()
}

// usage
IMyDependency myDependency; // assignment, instance creation...
MyClass myClass = new MyClass(myDependency);
```

Method Injection, Setter Injection

```
class MyClass
{
    public void DoSomethingUsingDependency(IMyDependency dependency)
    {
        // ... dependency.DoSomething() ...
    }
}
```

```
class MyClass
{
    public IMyDependency Dependency { get; set; }

    // in the methods: dependency.DoSomething()
}
```

- **Constructor Injection – common, if the dependency is required multiple times**
- **Method Injection – common, in case of a one-time requirement**
- **Setter Injection – rare**
  - Must check always: is set?
  - Even harder in multi-threaded environment

What injection type to use?

Constructor is more preferable, because

- all the dependencies can be visible at one place (ctor)
  - if the dependencies are scattered among all the methods (using method injection) it is hard to see and find them
- it easily can be seen that class X requires 15 dependencies in the ctor → meaning that the Single Responsibility (**S**OLID) principle is violated

Dependency Injection

”
I give you dependency injection for five-year-olds.

When you go and get things out of the refrigerator for yourself, you can cause problems. You might leave the door open, you might get something Mommy or Daddy doesn't want you to have. You might even be looking for something we don't even have or which has expired.

What you should be doing is stating a need, "I need something to drink with lunch," and then we will make sure you have something when you sit down to eat.
”

https://stackoverflow.com/questions/1638919/how-to-explain-dependency-injection-to-a-5-year-old/1638961#1638961

(Injection vs Inversion, in this semester we only need the Injection)

Dependency Inversion (is the fundamental principle):

- someone else creates the entity for you
- not you
- you only use it

Dependency Injection:

- the someone who creates the entity is the caller
- the caller calls the method, which needs to receive the dependency as a parameter

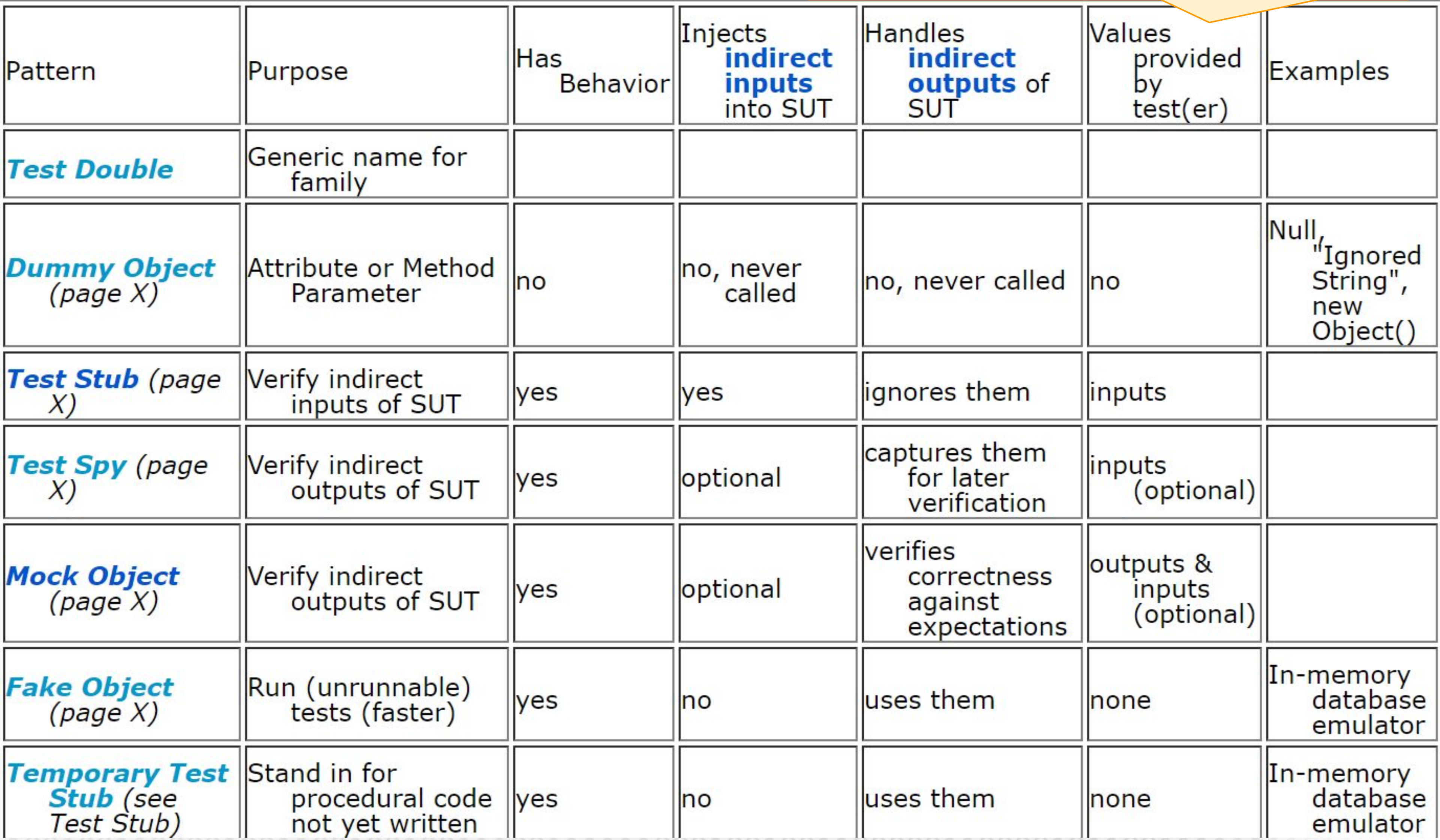
Dependency Injection

- **We provide the dependencies of the classes externally – via interfaces, which allows us**
  - To change the dependency later at any time to another class – while keeping the interface
  - To perform “truly” independent individual unit-tests
- **When testing, we replace the dependency with a known and simple object solely created for the tests**
  - The class will be independent from the dependency
  - Speed: we use a simple class representing only the tested methods instead of a complex logic
  - Independent: 1 test □ 1 exchange-object, usable for all operations
  - Various techniques: Dummy, Stub, Spy, Fake, Mock: **Same interface, different philosophy**

Sidenote: creating a codebase which fully meets the requirements of the DI is not so easy. In our case, the Main() function must know about all the dependencies?! → IoC containers can be used to overcome this.

Test doubles

Namings are not straightforward! Depending on which language / book / whatever we are reading, one phrase (eg. fake) means a bit different thing.

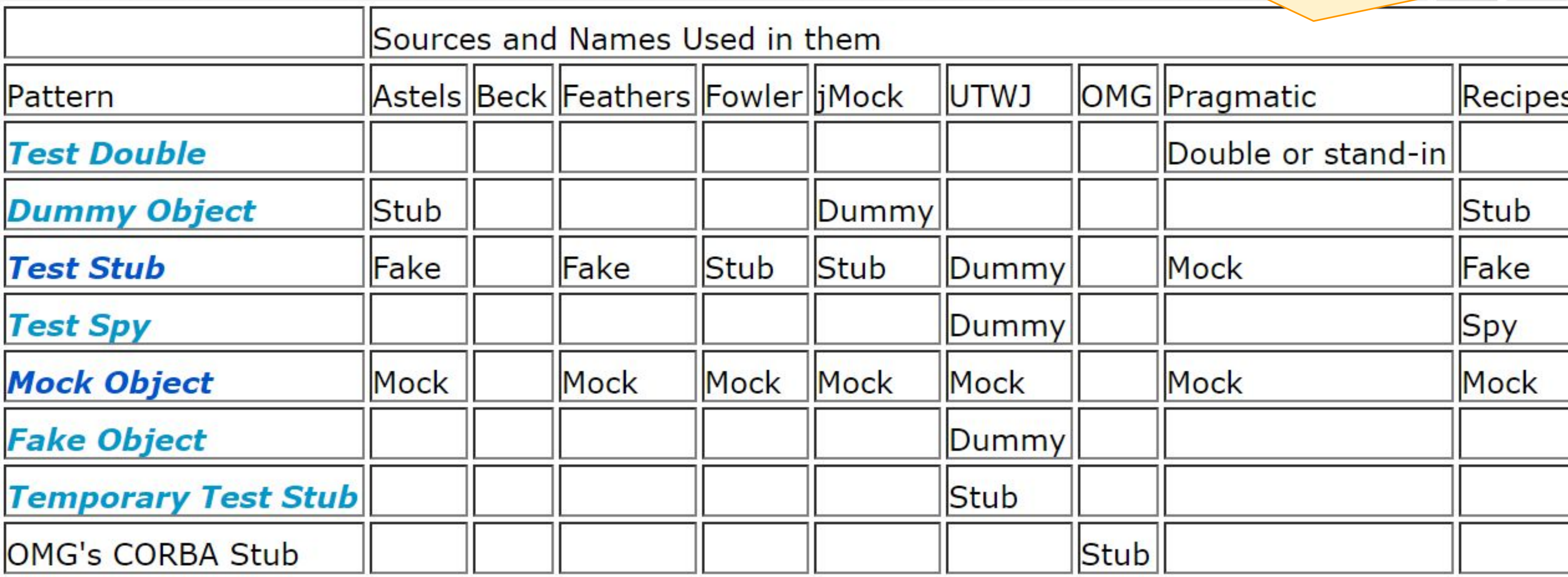


Pattern	Purpose	Has Behavior	Injects **indirect inputs** into SUT	Handles **indirect outputs** of SUT	Values provided by test(er)	Examples
Test Double	Generic name for family					
Dummy Object *(page X)*	Attribute or Method Parameter	no	no, never called	no, never called	no	Null, "Ignored String", new Object()
Test Stub *(page X)*	Verify indirect inputs of SUT	yes	yes	ignores them	inputs	
Test Spy *(page X)*	Verify indirect outputs of SUT	yes	optional	captures them for later verification	inputs (optional)	
Mock Object *(page X)*	Verify indirect outputs of SUT	yes	optional	verifies correctness against expectations	outputs & inputs (optional)	
Fake Object *(page X)*	Run (unrunnable) tests (faster)	yes	no	uses them	none	In-memory database emulator
Temporary Test Stub *(see Test Stub)*	Stand in for procedural code not yet written	yes	no	uses them	none	In-memory database emulator

Forrás: http://xunitpatterns.com/Mocks,%20Fakes,%20Stubs%20and%20Dummies.html

Test doubles

This table creates a “lookup” for which phrase was used in which context, based on the previous table.

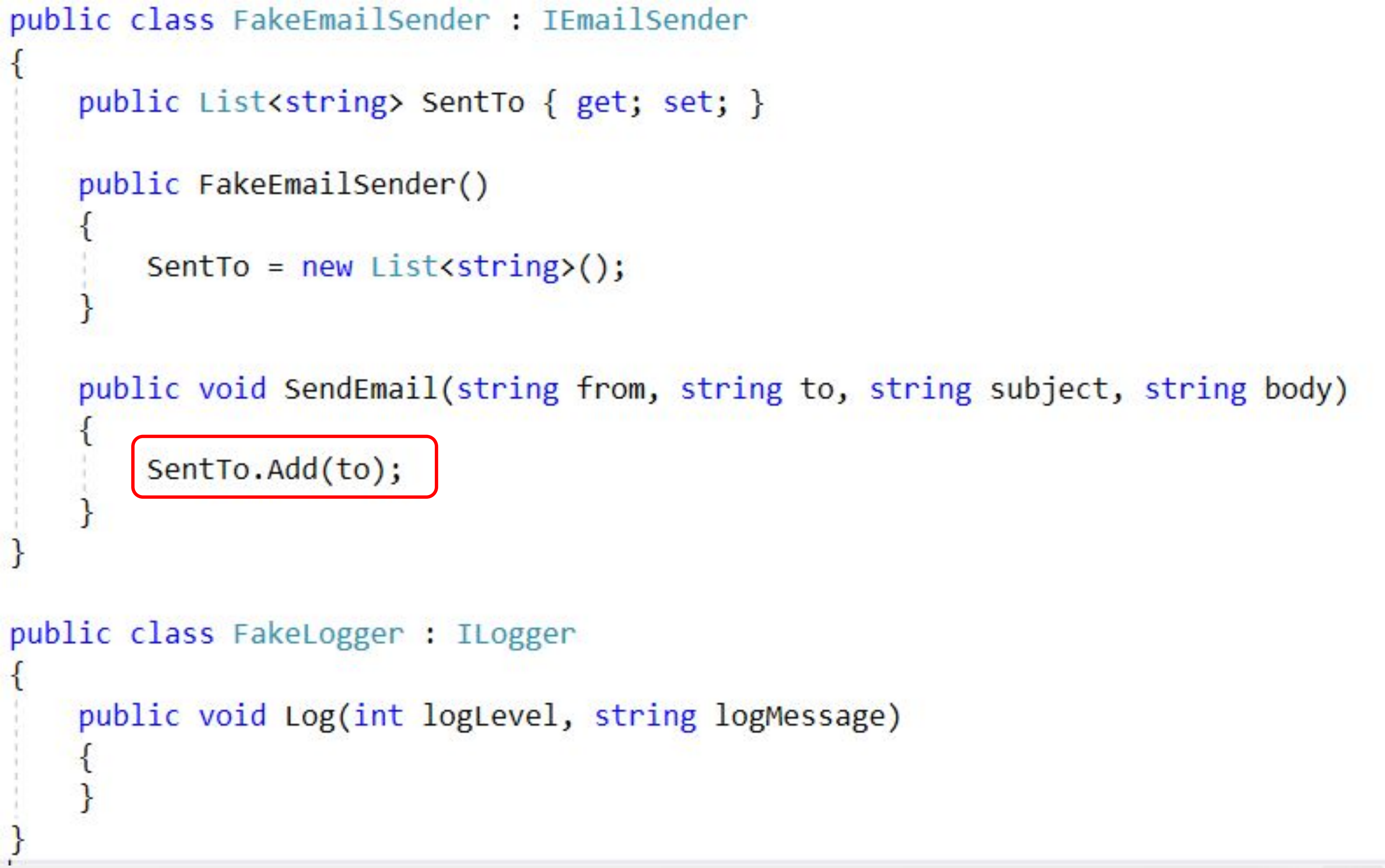


	Sources and Names Used in them								
Pattern	Astels	Beck	Feathers	Fowler	jMock	UTWJ	OMG	Pragmatic	Recipes
Test Double								Double or stand-in	
Dummy Object	Stub				Dummy				Stub
Test Stub	Fake		Fake	Stub	Stub	Dummy		Mock	Fake
Test Spy						Dummy			Spy
Mock Object	Mock		Mock	Mock	Mock	Mock		Mock	Mock
Fake Object						Dummy			
Temporary Test Stub						Stub			
OMG's CORBA Stub							Stub		

- **Astels = David Astels: Test-Driven Development - A practical guide**
- **Beck = Kent Beck: Test-Driven Development By Example**
- **Feathers = Michael Feathers: Working Effectively with Legacy Code**
- **Fowler = Martin Fowler: Mocks are Stubs**
- **jMock = Steve Freeman, Tim Mackinnon, Nat Pryce, Joe Walnes: Mock Roles, Not Objects**
- **UTWJ = Morgan Kaufmann: Unit Testing With Java**
- **OMG = Object Management Group's CORBA specs**
- **Pragmatic = Andy Hunt, Dave Thomas: Pragmatic Unit Testing with Nunit**
- **Recipes = J.B.Rainsberger: JUnit Recipes**

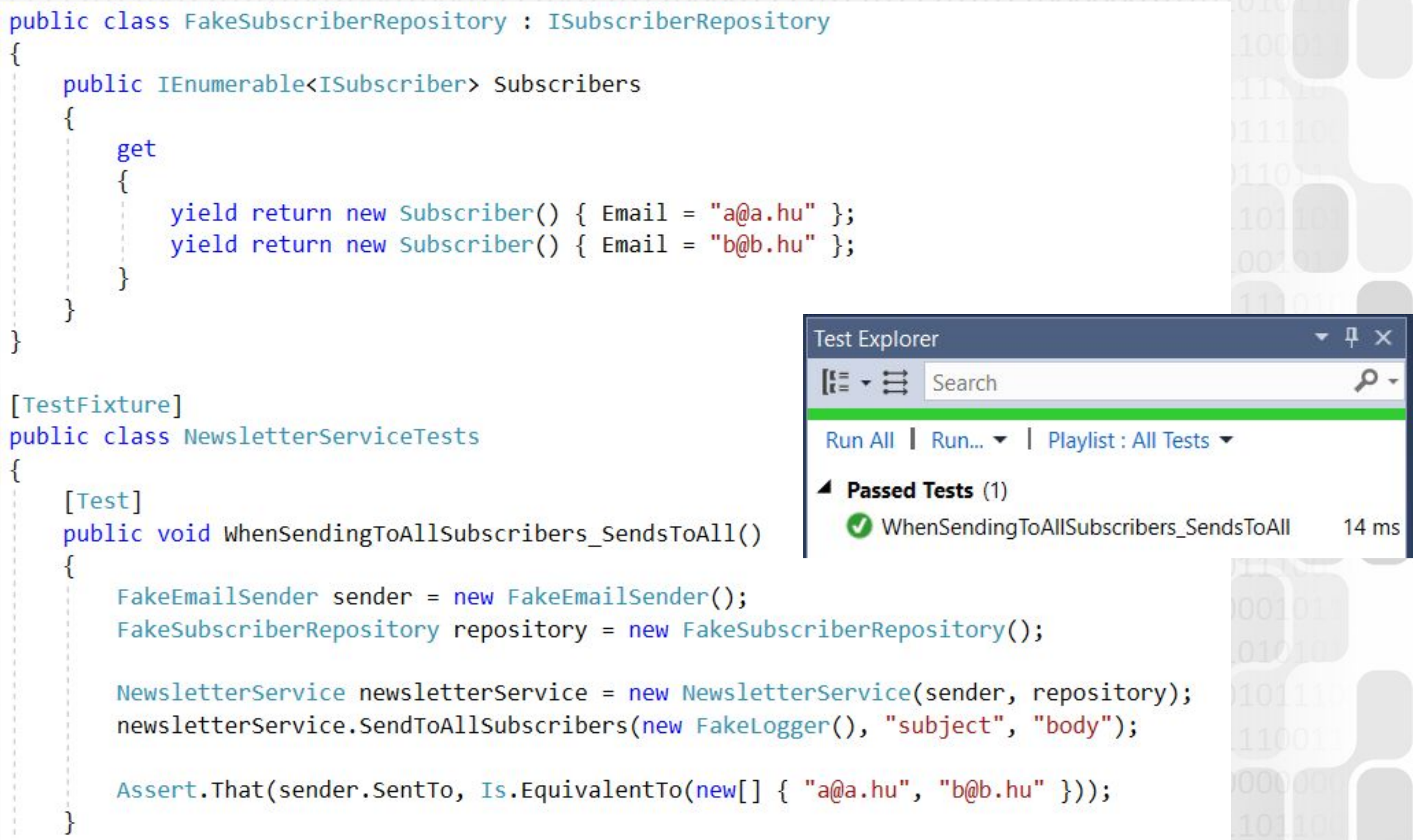
Fake

- **Implements the same interface as the substituted object, but it implements an intentionally simplified logic with different “target”**



```
public class FakeEmailSender : IEmailSender
{
    public List<string> SentTo { get; set; }

    public FakeEmailSender()
    {
        SentTo = new List<string>();
    }

    public void SendEmail(string from, string to, string subject, string body)
    {
        SentTo.Add(to);
    }
}

public class FakeLogger : ILogger
{
    public void Log(int logLevel, string logMessage)
    {
    }
}
```

Fake



```
public class FakeSubscriberRepository : ISubscriberRepository
{
    public IEnumerable<ISubscriber> Subscribers
    {
        get
        {
            yield return new Subscriber() { Email = "a@a.hu" };
            yield return new Subscriber() { Email = "b@b.hu" };
        }
    }
}

[TestFixture]
public class NewsletterServiceTests
{
    [Test]
    public void WhenSendingToAllSubscribers_SendsToAll()
    {
        FakeEmailSender sender = new FakeEmailSender();
        FakeSubscriberRepository repository = new FakeSubscriberRepository();

        NewsletterService newsletterService = new NewsletterService(sender, repository);
        newsletterService.SendToAllSubscribers(new FakeLogger(), "subject", "body");

        Assert.That(sender.SentTo, Is.EquivalentTo(new[] { "a@a.hu", "b@b.hu" }));
    }
```

Passed Tests (1)

✓ WhenSendingToAllSubscribers_SendsToAll 14 ms

- **Fake Repository □ too much code (all CRUD methods for all objects…), hard to make a universal fake**

Mock

- „Partially” implement an interface with a substitute object – no logic, fixed data
- Usually we use a Mocking framework

IEmailSender means that it has a SendEmil() method.

```
Mock<IEmailSender> mockMailSender = new Mock<IEmailSender>();

// Setup the expected behavior of the mock...

NewsletterService service = new NewsletterService(mockMailSender.Object);
```

The mocked (fake) object is given!
This mocked object will behave as a real object, except it is fake.

  - Due to the professional Mocking frameworks, the boundaries between mock, fake, stub etc. is very blurry
  - Mocks can be used as fakes (without expectation checks)
- **One of the most widespread .NET Mocking Framework: Moq**
  - ONLY interface can be mocked (even if the newer Moq can mock classes)
- **Gives us ways to observe the logic that is executed**
  - Expectations: „the DoSomething() method was called once”

Mock

- **Create an interface implementation without a real class**
  - Methods are callable by default, no exception is thrown, default is returned (MockBehavior.Loose vs Strict)
  - The properties are readable/writeable, but the values are not remembered
  - Use the .Setup/.SetupAllProperties method to change the behavior
- **We can create different mocks for different test cases**
  - We only have to setup the required operations with the appropriate data
  - vs as we seen it is hard to create universal fake classes

```
Mock<ISubscriberRepository> repositoryMock = new Mock<ISubscriberRepository>();
Mock<IEmailSender> senderMock = new Mock<IEmailSender>();

// Setup differently the behavior of the mock
repositoryMock.Setup(m => m.Subscribers).Returns(expectedSubscribers);
repositoryMock.Setup(m => m.Subscribers).Returns(Enumerable.Empty<Subscriber>());
repositoryMock.Setup(m => m.Subscribers).Throws<NullReferenceException>();

senderMock.Setup(
     m => m.SendEmail("a@a.hu", "to", "subject", "body")).Returns(true);
senderMock.Setup(
     m => m.SendEmail(null, It.IsAny<string>(), It.IsAny<string>(), It.IsAny<string>()))
          .Throws<NullReferenceException>();
```

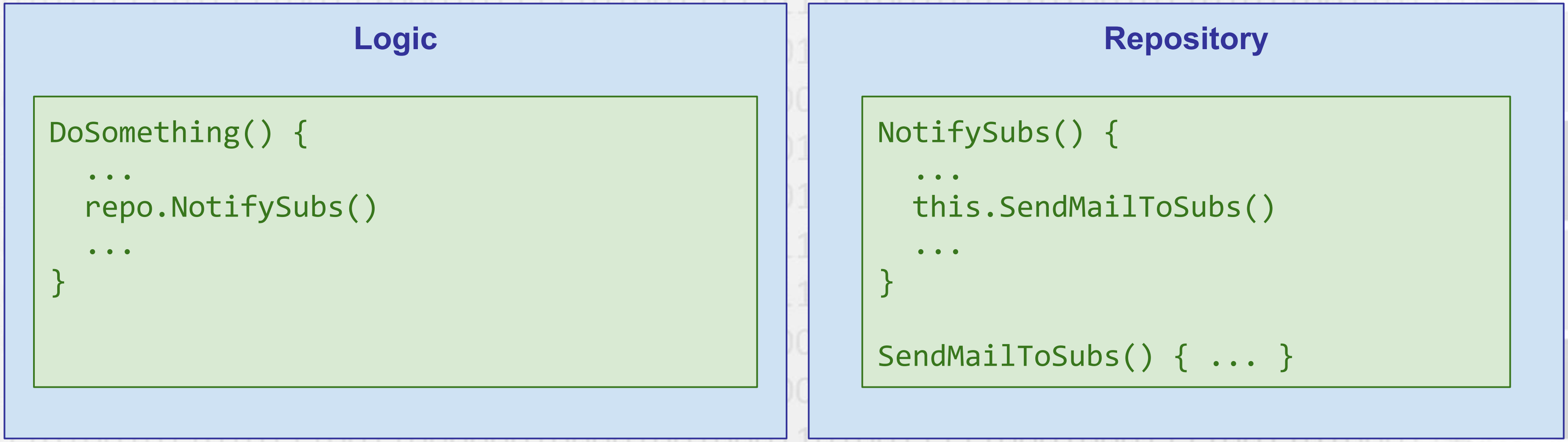
Testing the logic with a mock (after Setup)

```
// ARRANGE
Mock<IEmployeeBusinessLogic> mbl = new Mock<IEmployeeBusinessLogic>();
// ... Call .Setup() for the required methods/properties

// ACT ...

// ASSERT
// Called once with a@a.hu
senderMock.Verify(
    m => m.SendEmail("a@a.hu", It.IsAny<string>(), It.IsAny<string>(), It.IsAny<string>()),
    Times.Once);

// Never called with null
senderMock.Verify(
    m => m.SendEmail(null, It.IsAny<string>(), It.IsAny<string>(), It.IsAny<string>()),
    Times.Never);

// Zero logs with bigger than one first parameter
loggerMock.Verify(
    m => m.Log(It.Is<int>(i => i > 1), It.IsAny<string>()), Times.Never);
```

I'm testing the logic here!

In the 'ACT' I call some method of the logic; then, in the 'ASSERT' I'm checking if -- based on that logic call -- some other method was called accordingly or not?!

Testing **the logic** with a mock (after Setup)



Logic

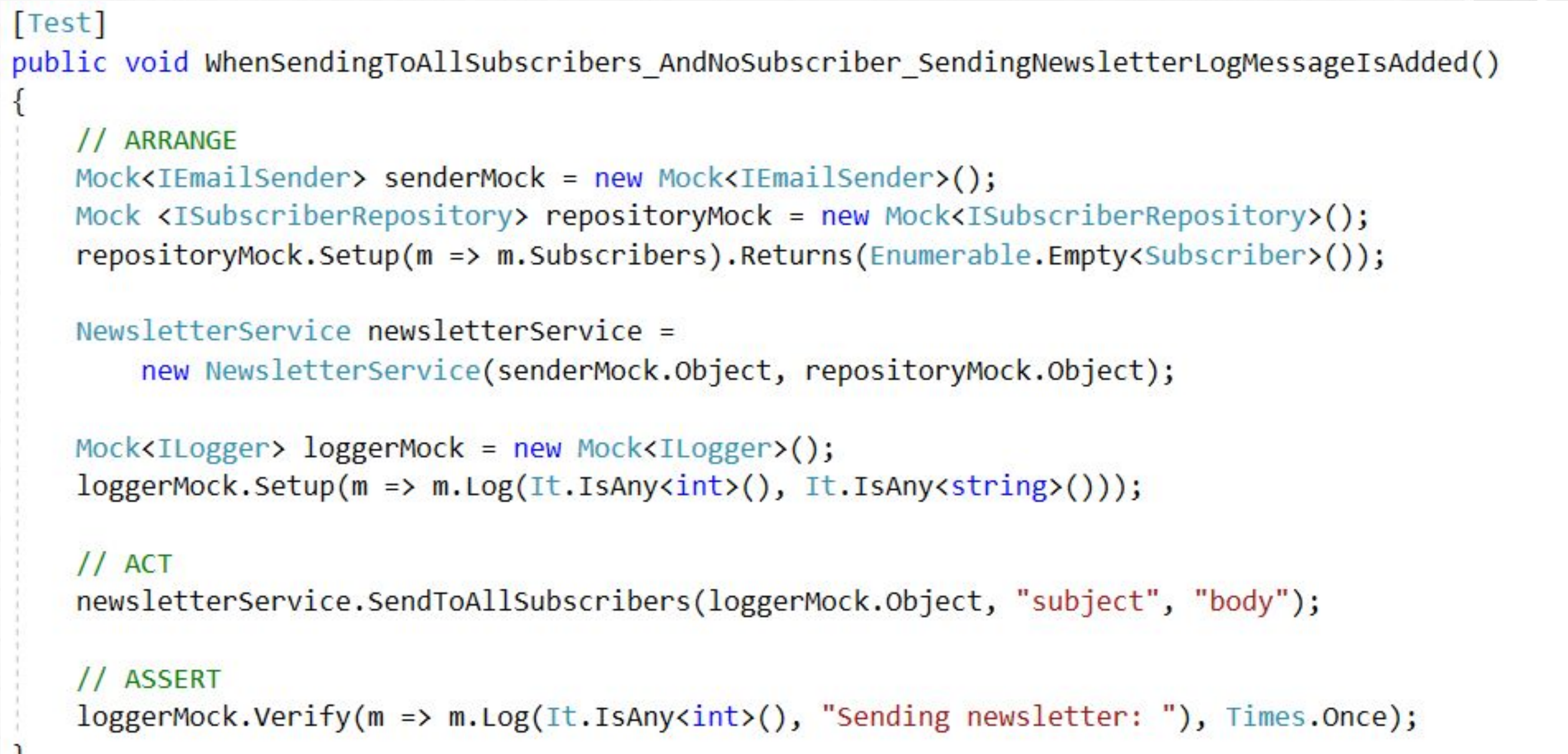
```
DoSomething() {
  ...
  repo.NotifySubs()
  ...
}
```

Repository

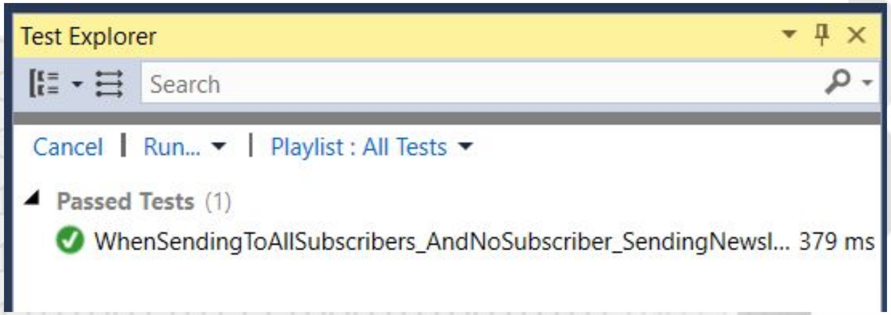
```
NotifySubs() {
  ...
  this.SendMailToSubs()
  ...
}

SendMailToSubs() { ... }
```

Verify if based on the logic's *`DoSomething`* method call, the repository's *`SendMailToSubs`* (or any internal) method was called or not?!

Testing with a mock



```
[Test]
public void WhenSendingToAllSubscribers_AndNoSubscriber_SendingNewsletterLogMessageIsAdded()
{
    // ARRANGE
    Mock<IEmailSender> senderMock = new Mock<IEmailSender>();
    Mock <ISubscriberRepository> repositoryMock = new Mock<ISubscriberRepository>();
    repositoryMock.Setup(m => m.Subscribers).Returns(Enumerable.Empty<Subscriber>());

    NewsletterService newsletterService =
        new NewsletterService(senderMock.Object, repositoryMock.Object);

    Mock<ILogger> loggerMock = new Mock<ILogger>();
    loggerMock.Setup(m => m.Log(It.IsAny<int>(), It.IsAny<string>()));

    // ACT
    newsletterService.SendToAllSubscribers(loggerMock.Object, "subject", "body");

    // ASSERT
    loggerMock.Verify(m => m.Log(It.IsAny<int>(), "Sending newsletter: "), Times.Once);
}
```



Passed Tests (1)

✓ WhenSendingToAllSubscribers_AndNoSubscriber_SendingNewsl... 379 ms

Testability Code Smells

“A code smell is a surface indication that usually corresponds to a deeper problem in the system.” ~ Martin Fowler

- **References/parameters with class-typed variables**
  - Refactor to interface-typed variables
- **new()**
  - Refactor to use DI (usually a Factory pattern)
  - Don’t have to get rid of every “new” keywords, but worth to look after
- **DI: too many parameters**
  - The class is responsible for too many things, refactor/split
  - Or: merge the parameters into one class
- **Usage of static methods and variables (Singleton pattern)**
  - Refactor to use instances
- **Too much/too long static helper methods**
  - This is not OO, refactor to nonstatic class
- **User interaction mixed with logic**
  - Separate them, refactor to separated classes (Humble Object pattern)
- **Database, file IO, resources all mixed with logic…**
  - Separate them, refactor to classes (Repository, Humble Object pattern)
- **Too many mocks created in one test**
  - Too many responsibilities, refactor/split
- **Direct access to mock.Object □ FORBIDDEN**

Exercise

- **Create a business logic for the Cars and Brands tables**
  - Access the data using a Repository
  - Create the interfaces that describe the expected repository operations
  - The logic should be capable of filtering for a brand and calculate the averages for brands

- **To test the logic, do NOT access the real database!**
  - Using Moq, during the Arrange phase create a mocked repository
  - The data access method of the repository should return data suitable for the current test case
  - During the test, we should check if the logic returns with good values, and also that it correctly uses the data access methods of the repository