Mocking, Stubbing and DI

The challenge of dependencies

The objects we tested yesterday were all

free of dependencies.

In order to fulfill their purpose, they did not depend on any other objects to help

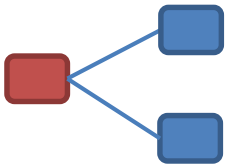
(If we ignore data vessels, lists, dictionaries…).

The challenge of dependencies

EDUMENT

In reality, **many objects cannot avoid having dependencies** on other objects. These dependencies may...

- Retrieve or persist data using a **database**
- Access the **file system**
- Make **web requests**, for example to a web service or web API
- Send **email** or **SMS** messages



The challenge of dependencies

EDUMENT

At the same time, we know our unit tests must be **independent of the environment**.

So, what can we do?

Sample problem: flight status notifications

To help us understand the problems and solutions, we'll turn to another sample domain:

flight status notifications.

Sample problem: flight status notifications

EDUMENT

Wing and a Prayer Airlines wish to keep passengers up to date by sending them SMS notifications when the status of their flight changes. Of note, they wish to send notifications...

- When the flight opens for check-in; this should include the booking reference and information about any delay
- If a flight is delayed, or the amount of delay changes, after check-in
- If a flight is cancelled

Passenger lists for a flight can be obtained using a class providing a Data Access Layer, and another class exists that can send SMS notifications.

The problem with **new**

EDUMENT

Yesterday, we carefully designed our classes so we could control the data they worked with from our tests. We wish to do similar with dependencies.

The problem we face is that dependencies are often instantiated and used like this:

```
var sms = new SMSSender();
sms.Send(number, body);
```

This code **couples tightly** to SMSSender. There's no way anything outside of the class can take control of the dependency.

The problem with **new**

EDUMENT

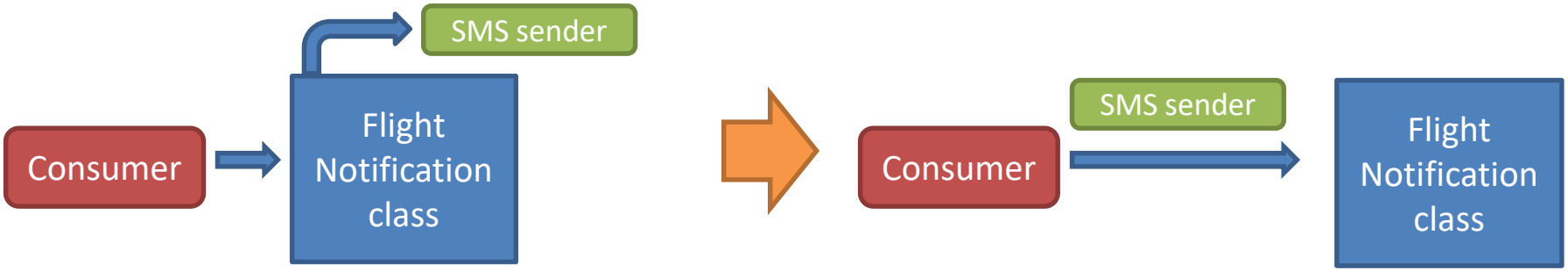
So, how can we avoid using the **new** keyword and coupling so tightly to the **SMSSender** class?

```
var sms = new SMSSender();
sms.Send(number, body);
```

The Dependency Inversion solution

The **Dependency Inversion Principle**, one of the **SOLID** principles of Object Oriented design, points us towards a solution.

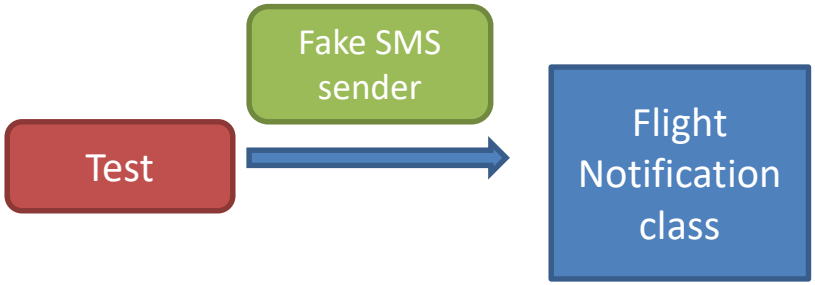
It states that a class **should not create its dependencies**. Instead, any dependencies a class has should be **chosen and provided by its consumers**.



The Dependency Inversion solution

When running in production, the consumer of the class would supply a real data access layer and an object that really sends SMS messages - the same ones we would likely have created inside the class using the **new** keyword.

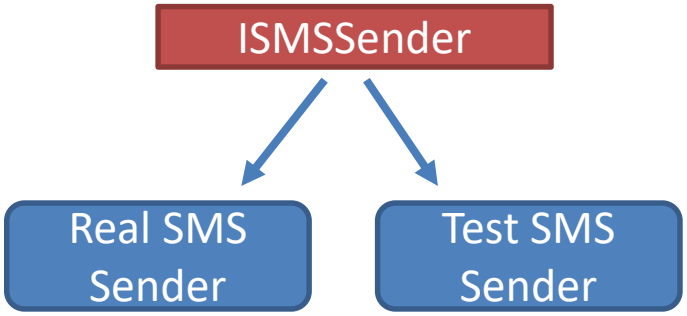
Our tests, however, are able to provide **objects controlled by the tests**, which will enable reliable testing.



Interfaces

If we're going to have multiple implementations of a dependency - a real one and one for testing - we need them to have something in common, so we can write code that works against either of them.

We could use a common abstract base class, but **inheritance creates tight coupling**. Therefore, we'll use interfaces instead.




```
public interface ISMSSender
{
    void Send(string number, string message);
}
```

Applying Dependency Inversion

The most common way to apply the **DI principle** is **constructor injection**. The constructor of a class is used to pass in its dependencies.

```
public class Notifier
{
    private ISMSSender smsSender;

    public Notifier(ISMSSender smsSender)
    {
        this.smsSender = smsSender;
    }
}
```

The dependencies are stored in private fields, so they can be used when implementing methods in the class.

When to define the interface?

EDUMENT

If the class to be developed has a dependency, and no prior work exists to implement the dependency, at what point should the interface be defined?

- We need it to exist so we can start writing tests
- However, we want to avoid too much up-front design

A way forward is to create empty interfaces, and fill them in as we start to understand the system through writing tests.

```
public interface IPassengerInfoDAL
{
}
```

The role of stubs

There are many frameworks we can use to create mock or stub objects. We'll consider mock objects later; for now we'll just consider **stubs**, and show how they can be created without a framework of any kind.

A hand-crafted stub should implement the interface it is stubbing. Rather than implementing the real functionality, it needs to:

- Provide a way for the test to control method and property return values.
- Record what methods were called and what arguments they were called with, so the test can assert that the expected calls were made.

A stub SMS sender

Our tests will want to check that the expected SMS messages were sent. Our stub object simply collects each message that is sent and stores it.

```
class SMSSenderStub : ISMSSender
{
    public List<Tuple<string, string>> sent =
        new List<Tuple<string, string>>();

    public void Send(string number, string message)
    {
        sent.Add(new Tuple<string, string>(number, message));
    }
}
```

Note that each test will receive a fresh stub object.

Planning for our first test

Our first test will check that when a flight opens with no delay, then:

- All passengers on that flight are notified
- The message contains the flight code
- The message does not mention a delay

To write these tests, we can see we'll need a way to control the passenger list. Therefore, it seems like a good time to flesh out the interface.

```
public interface IPassengerInfoDAL
{
    List<Passenger> PassengersOnFlight(string flightCode);
}
```

A stub for IPassengerInfoDAL

EDUMENT

The stub provides a way for the test to control the resulting passenger list. It also records the flight that results were requested for.

```
class PassengerInfoDALStub : IPassengerInfoDAL
{
    public string flightCodeRequested;
    public List<Passenger> results;

    public List<Passenger> PassengersOnFlight(string flightCode)
    {
        flightCodeRequested = flightCode;
        return results;
    }
}
```

Setup

The stub objects, as well as the System Under Test, are initialized in a setup method. Notice how the **SUT convention** now becomes especially useful in picking out the object being tested.

```
public class test
{
    private SMSSenderStub smsStub;
    private PassengerInfoDALStub passStub;
    private Notifier sut;

    [SetUp]
    public void Setup()
    {
        smsStub = new SMSSenderStub();
        passStub = new PassengerInfoDALStub();
        sut = new Notifier(smsStub, passStub);
    }
}
```

Testing flight opening

EDUMENT

```
[Test]
public void PassengersNotifiedWhenFlightOpensForCheckin()
{
    passStub.results = new List<Passenger>
        {
            new Passenger { Name = "Dave", Mobile = "123456" },
            new Passenger { Name = "Lena", Mobile = "987654" }
        };

    var depTime = new DateTime(2013, 02, 25, 12, 00, 00);
    sut.FlightOpened("WP123", depTime, depTime);

    Assert.AreEqual(passStub.flightCodeRequested, "WP123");
    Assert.AreEqual(2, smsStub.sent.Count);
    Assert.That(smsStub.sent.Any(m => m.Item1 == "123456"), "Dave");
    Assert.That(smsStub.sent.Any(m => m.Item1 == "987654"), "Lena");
}
```

Making the test pass

EDUMENT

The following code will pass the test:

```
public class Notifier
{
    // ....

    public void FlightOpened(string flight,
                             DateTime scheduled,
                             DateTime actual)
    {
        foreach (var pass in dal.PassengersOnFlight(flight))
            smsSender.Send(pass.Mobile,
                "Your flight is open for check-in");
    }
}
```

Remember to write just enough code to make the test pass, but not more! Of note, we do not yet have a test to cover putting the flight code into the message, or doing anything about delays.

Message content

EDUMENT

The messages should contain the flight number, but have no mention of any kind of delay.

```
[Test]
public void MessageContainsFlightNumberAndNoDelay()
{
    passStub.results = new List<Passenger>
        {
            new Passenger { Name = "Dave", Mobile = "123456" },
            new Passenger { Name = "Lena", Mobile = "987654" }
        };

    var depTime = new DateTime(2013, 02, 25, 12, 00, 00);
    sut.FlightOpened("WP123", depTime, depTime);

    Assert.That(smsStub.sent.All(m => m.Item2.Contains("WP123")));
    Assert.That(smsStub.sent.All(m => !m.Item2.Contains("delay")));
}
```

Making this one pass is easy!

Flight opening for check-in with a delay

EDUMENT

When a flight is delayed at the point check-in opens, this should be indicated in the SMS. The number of minutes of delay should be included.

```
[Test]
public void DelayIsReportedInMinutes()
{
    passStub.results = new List<Passenger>
        {
            new Passenger { Name = "Dave", Mobile = "123456" },
            new Passenger { Name = "Lena", Mobile = "987654" }
        };

    var depTime = new DateTime(2013, 02, 25, 12, 00, 00);
    sut.FlightOpened("WP123", depTime, depTime.AddHours(1));

    Assert.That(smsStub.sent.All(m => m.Item2.Contains("delay")));
    Assert.That(smsStub.sent.All(m => m.Item2.Contains("60 min")));
}
```

Including the delay

EDUMENT

The following code is sufficient to make the test pass.

```
public void FlightOpened(string flight, DateTime scheduled,
                         DateTime actual)
{
    var message = "Your flight " + flight + " is open for check-in.";
    if (scheduled != actual)
    {
        var mins = (int)actual.Subtract(scheduled).TotalMinutes;
        message += " It has a delay of " + mins + " minutes.";
    }

    foreach (var pass in dal.PassengersOnFlight(flight))
        smsSender.Send(pass.Mobile, message);
}
```

Rinse and repeat

EDUMENT

There was a little work at the start to get the interfaces and stubs in place. However, beyond that, **it's the usual TDD workflow**: write a test, make it pass, refactor if needed.

Factoring things this way gives our tests **control over the code's input** and a way to **assert against the code's results** - even when it depends on other objects.

Furthermore, our code now **better follows the SOLID principles** and is more **loosely coupled** - benefits that extend beyond having automated tests.

Stubs vs. mocks

The terminology around **test doubles** - objects that stand in for the real ones in a testing scenario - is messy.

The approach we have taken uses **stub objects**. They don't attempt to check the behavior is correct. Instead, they hand back results of our choosing and log what happens. Our tests are still **Arrange, Act, Assert**.

Stubs vs. mocks

By contrast, **mock objects are pre-programmed with expectations**. Typically, this is done in an initial **record mode**. The mock is done put into **playback mode**. It verifies that the expected calls are made, and potentially that no unexpected calls are made.

There are arguments for and against mocks. Against them is the way they upset the AAA practice, and that unexpected calls make tests fragile as the system evolves.

Exercise #4

EDUMENT

In the previous exercise, we implemented the **`TourSchedule`** class, but we still haven't added any logic to book a customer on a certain tour.

This will be a class of its own, and we will want to test it in isolation. At the same time, it will need to have access to at least the signatures of the methods in **`TourSchedule`**.

In this exercise, we will extract an interface from **`TourSchedule`** and inject it into our new class. This will also present us with the opportunity to create a hand-written stub object to use in our tests.