

# Tests Mandatory 2

By Oliver Dehnfeld and Jonas Hansen

## Introduction

The objective of this mandatory assignment is to practice doing user driven acceptance testing.

## Task at hand

We're given the task of creating a system for a phone company, which is using a storage depot in Sweden.

The system must be able to do 3 things:

- Create deliveries of hardware
- Order new materials to the storage depot
- Check the stock quantity of the storage depot

The system will eventually be a web application, but the goal of this task is to make it easy to write user driven acceptance tests along the way.

## Step 1: Write a DTD for the 3 requirements

First up, we have written a DTD for creating a delivery of a phone.

It contains very basic information about a delivery, and is not handling the payment at all.

This could be implemented in a later version of the system.

```
<?xml encoding="UTF-8"?>
```

```
<!ELEMENT DeliveryInfo (DeliveryId, Customer, Address, Delivery, OrderInfo)>
```

```
<!ATTLIST DeliveryInfo
```

```
  xmlns CDATA #FIXED ">
```

```
<!ELEMENT DeliveryId (#PCDATA)>
```

```
<!ATTLIST DeliveryId
```

```
  xmlns CDATA #FIXED ">
```

```
<!ELEMENT Delivery (Type, Price, Company)>
```

```
<!ATTLIST Delivery
```

```
  xmlns CDATA #FIXED ">
```

```
<!ELEMENT OrderInfo (Customer, Address, Items)>
```

```
<!ATTLIST OrderInfo
```

```
  xmlns CDATA #FIXED ">
```

```
<!ELEMENT Type (#PCDATA)>
```

<!ATTLIST Type  
xmlns CDATA #FIXED ">

<!ELEMENT Price (#PCDATA)>  
<!ATTLIST Price  
xmlns CDATA #FIXED ">

<!ELEMENT Company (Name)>  
<!ATTLIST Company  
xmlns CDATA #FIXED ">

<!ELEMENT Items (Item)+>  
<!ATTLIST Items  
xmlns CDATA #FIXED ">

<!ELEMENT Item (Model,Quantity)>  
<!ATTLIST Item  
xmlns CDATA #FIXED ">

<!ELEMENT Model (#PCDATA)>  
<!ATTLIST Model  
xmlns CDATA #FIXED ">

<!ELEMENT Quantity (#PCDATA)>  
<!ATTLIST Quantity  
xmlns CDATA #FIXED ">

<!ELEMENT Customer (Name,Email,Phone)>  
<!ATTLIST Customer  
xmlns CDATA #FIXED ">

<!ELEMENT Email (#PCDATA)>  
<!ATTLIST Email  
xmlns CDATA #FIXED ">

<!ELEMENT Phone (#PCDATA)>  
<!ATTLIST Phone  
xmlns CDATA #FIXED ">

<!ELEMENT Address (#PCDATA)>  
<!ATTLIST Address  
xmlns CDATA #FIXED ">

<!ELEMENT Name (#PCDATA)>  
<!ATTLIST Name  
xmlns CDATA #FIXED ">

An easier way of showing what this means, is by writing an XML file, which follows this DTD file:

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE DeliveryInfo SYSTEM "Delivery.dtd">
<!-- Delivery.xml stores information about order deliveries -->
<DeliveryInfo>
  <DeliveryId>50</DeliveryId>
  <Customer>
    <Name>Anders Madsen</Name>
    <Email>andersmadsen@gmail.com</Email>
    <Phone>+4520392938</Phone>
  </Customer>
  <Address>Stårevej 50, 2200 København N</Address>
  <Delivery>
    <Type>Express</Type>
    <Price>50</Price>
    <Company>
      <Name>GLS</Name>
    </Company>
  </Delivery>
  <OrderInfo>
    <Customer>
      <Name>Anders Madsen</Name>
      <Email>andersmadsen@gmail.com</Email>
      <Phone>+4520392938</Phone>
    </Customer>
    <Address>Stårevej 50, 2200 København N</Address>
    <Items>
      <Item>
        <Model>iPhone 8</Model>
        <Quantity>1</Quantity>
      </Item>
      <Item>
        <Model>Samsung Galaxy S10</Model>
        <Quantity>1</Quantity>
      </Item>
    </Items>
  </OrderInfo>
</DeliveryInfo>
```

In this picture, it's way easier to see what the different objects in the XML are taking, and how the objects are connected.

Secondly, we have written a similar DTD and XML for the ordering part of the system, where the storage depot is able to give the basic amount of information for ordering new phones. Again, this is not fully completed, and could be expanded a lot in the further development of the application.

For this, we are only going to show the XML in this report, as it's way more manageable to look at. If you're interested in seeing the DTD, you can look at the source code, which is provided in the zip file.

```

<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE OrderInfo SYSTEM "Order.dtd">
<!-- Order.xml stores information about orders from customers -->
<OrderInfo>
  <OrderId>256</OrderId>
  <Items>
    <Item>
      <Model>iPhone 8</Model>
      <Quantity>1000</Quantity>
    </Item>
    <Item>
      <Model>Samsung Galaxy S10</Model>
      <Quantity>1000</Quantity>
    </Item>
  </Items>
</OrderInfo>

```

Again we see how the objects are connected, and what types the objects are taking.

Finally we have written the DTD and XML for querying the stock of the different objects at the depot storage.

Again, we are only going to show the XML, as it's the easiest to understand.

```

<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE StorageInfo SYSTEM "Storage.dtd">
<!-- Storage.xml stores information about the storage on items on the depot -->
<StorageInfo>
  <Items>
    <Item>
      <Id>256773113</Id>
      <Category>Phone</Category>
      <Model>Samsung Spacestar 500</Model>
      <Price>8700</Price>
      <Quantity>26</Quantity>
    </Item>
    <Item>
      <Id>2786868372</Id>
      <Category>Phone</Category>
      <Model>iPhone MoneyStealer</Model>
      <Price>15000</Price>
      <Quantity>250</Quantity>
    </Item>
  </Items>
</StorageInfo>

```

## Step 2: Parse the XML to objects

We have written objects which correspond to the objects within the XML.

We don't want to show every single object, so we're just gonna show one object, and how we parse the XML data into the object.

```

namespace PhoneCompany.Model
{
    2 references
    public class DeliveryInfo
    {
        1 reference | 1/1 passing
        public string DeliveryId { get; set; }
        0 references
        public Customer Customer { get; set; }
        0 references
        public string Address { get; set; }
        0 references
        public Delivery Delivery { get; set; }
        0 references
        public OrderInfo OrderInfo { get; set; }
    }
}

```

This is the DeliveryInfo object, which corresponds to the Delivery.XML file. To parse this, we are using a custom Serializer, which is written in C#, as that is the language we are using for the UAT.

```

2 references
public class Serializer
{
    1 reference | 1/1 passing
    public T Deserialize<T>(string input) where T : class
    {
        XmlSerializer ser = new XmlSerializer(typeof(T));
        using (StringReader sr = new StringReader(input))
        {
            return (T)ser.Deserialize(sr);
        }
    }
}

```

In this method, we use the XmlSerializer from C#, and converts the given XML string to an object of the given type T. This is using a generic pattern, so you can use the same Deserialize method for multiple objects.

### Step 3: Run tests on the converted object

After converting the XML objects into objects in our system, we can perform some tests, to see if the conversion is made correctly.

In the following, we are using a test to check if the DeliveryId given in the XML is being parsed:

```

[Test]
0 references
public void TestDeliveryId()
{
    XmlDocument deliveryXml = new XmlDocument();

    Serializer ser = new Serializer();

    var workingDirectory = Environment.CurrentDirectory;

    var path = Directory.GetParent(workingDirectory).Parent.Parent.FullName + @"\Delivery.xml";

    deliveryXml.Load(path);

    var xmlInputData = File.ReadAllText(path);

    DeliveryInfo deliveryInfo = ser.Deserialize<DeliveryInfo>(xmlInputData);

    var element = deliveryXml.GetElementsByTagName("DeliveryId");

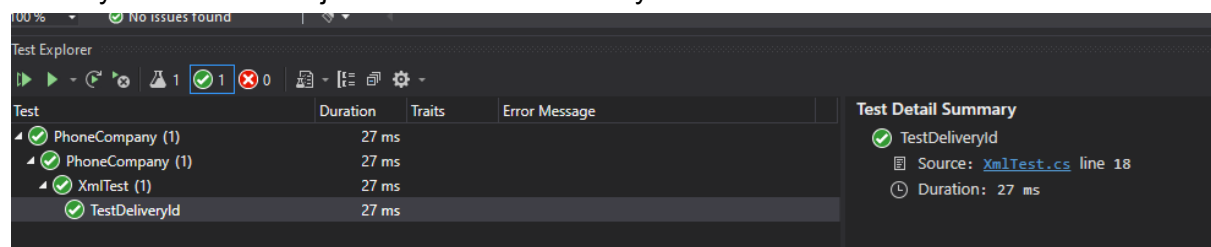
    Assert.AreEqual(element.Item(0).InnerText, deliveryInfo.DeliveryId);
}

```

We are using our serializer to convert the XML DeliveryInfo object into our own DeliveryInfo object in our system.

Then, we load the XML into the application with XmlDocument.Load(), which is built in C#, and then we get the value of the DeliveryId from the XML file.

At last, we perform a test using Assert.AreEqual(), to check if the converted value of the DeliveryId in our own object matches the DeliveryId from the XML.



Then, we run the test, and as it passes, we know our conversion has been successful.

We want to dive deeper into the DeliveryInfo object, and check if the Customer object inside DeliveryInfo is also being set, so we do the following:

```

[Test]
0 references
public void TestDeliveryInfoCustomer()
{
    XmlDocument deliveryXml = new XmlDocument();

    Serializer ser = new Serializer();

    var workingDirectory = Environment.CurrentDirectory;

    var path = Directory.GetParent(workingDirectory).Parent.Parent.FullName + @"\Delivery.xml";

    deliveryXml.Load(path);

    var xmlInputData = File.ReadAllText(path);

    DeliveryInfo deliveryInfo = ser.Deserialize<DeliveryInfo>(xmlInputData);

    var element = deliveryXml.GetElementsByTagName("Customer");

    Assert.AreEqual(element.Item(0).FirstChild.InnerText, deliveryInfo.Customer.Name);
}

```

We still get the XElement by id, but this time we use “Customer” instead, as we want to get the Customer inside the XML.

We want to check if the name is being mapped correctly, so we get the first child of the XElement, which is the name, and checks if it’s the same as the DeliveryInfo.Customer.Name object from our own system.

Test	Duration	Traits	Error Message	Test Detail Summary
PhoneCompany (2)	55 ms			TestDeliveryInfoCustomer
PhoneCompany (2)	55 ms			Source: <a href="#">XmlTest.cs</a> line 40
XmlTest (2)	55 ms			Duration: 28 ms
TestDeliveryId	27 ms			
TestDeliveryInfoCustomer	28 ms			

The test passes, which tells us, that the Customer object inside the DeliveryInfo object is also being converted correctly.

## Task 2: Use a non-XML-domain specific language

To showcase the possibility of using another non-XML tool, we have chosen to work with SpecFlow, also in our .NET project.

SpecFlow will allow us to chain together Gherkin Syntax and tests.

On [cucumber’s website](#) you can find a list of implementations, one of them being SpecFlow.

SpecFlow can be downloaded as a NuGet for a .NET project. Information and guidance can be found [here](#)

Usually Gherkin Syntax is used as a part of behaviour driven development, which is a software development process which can clarify the communication between software developers and non-technical stakeholders. This we will not go into depth with, as it is beyond the scope of the task. It is just good to understand that it is a tool to create a

middleware for developer and user communication, and it can be used to allow the users to write test scenarios.

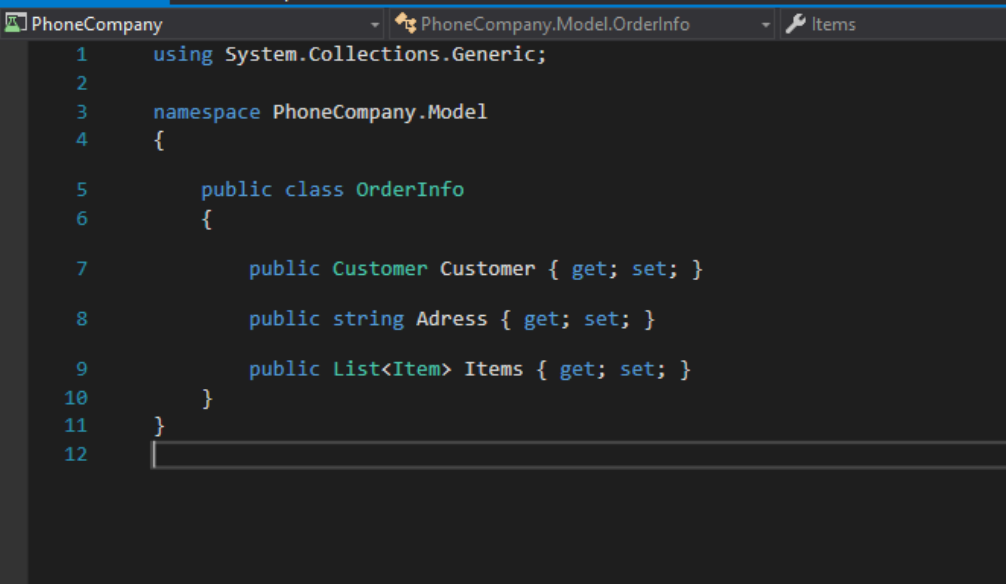
In short, Gherkin Syntax can be broken down to a “Given, When, Then” formula. More keywords exist, but this gets the basic understanding down.

## Example

To showcase an example of how this can be done, we will show how you can configure the possibility to use Gherkin Syntax and connect that to tests.

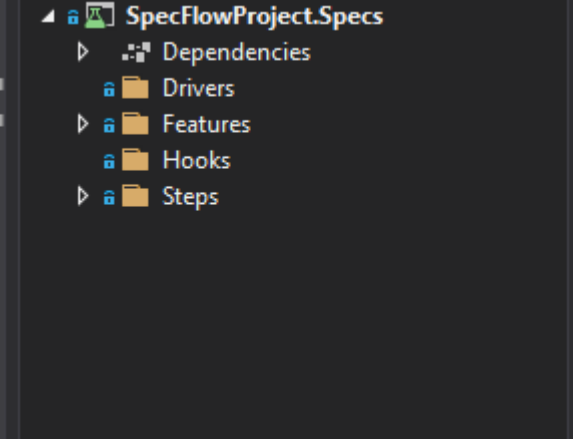
We will assume that the necessary NuGet package has been installed and a new SpecFlow solution has been created. You could configure different test-runners, but we have just chosen to go with SpecFlow's own.

In this example we will use the OrderInfo model.



```
1 using System.Collections.Generic;
2
3 namespace PhoneCompany.Model
4 {
5     public class OrderInfo
6     {
7         public Customer Customer { get; set; }
8         public string Address { get; set; }
9         public List<Item> Items { get; set; }
10    }
11 }
12
```

Then, in our SpecFlow solution, we will have the following:



```
SpecFlowProject.Specs
├── Dependencies
├── Drivers
├── Features
├── Hooks
└── Steps
```

Inside our Features folder, we can create feature files, which will contain Gherkin Syntax.



We create a Order.feature.cs file:

```
1 Feature: Order
2
3 Scenario: Order two iPhone 8
4     Given the model ordered is an iPhone 8
5     And the quantity is 2
6     When the order is made
7     Then an order containing model iPhone 8 and quantity 2
```

Inside this file, a user would be able to create test scenarios. The syntax is clear, and fairly self explanatory.

You create scenarios and then you by the keywords specify the actions you wish to occur. You may notice that “iPhone 8” and “2” are greyed out. The reason for that, is that the connected “steps”/test suite is set up, so that it looks at it, and uses it as parameters in the test.

This would allow the user to copy/paste this scenario and input other values, if they wish to. Of course they must follow the same syntactic pattern, otherwise we would need to write more tests.

Now, if we go into our “Steps” folder, we have created a file called “OrderStepDefinition.cs”:

```

namespace SpecFlowProject.Specs.Steps
{
    [Binding]
    1 reference
    public sealed class OrderStepDefinition
    {
        private readonly ScenarioContext _scenarioContext;
        private readonly Item _item = new Item();
        private int _quantity;
        private string _model;

        0 references
        public OrderStepDefinition(ScenarioContext scenarioContext)
        {
            _scenarioContext = scenarioContext;
        }

        [Given("the model ordered is an (.*)")]
        0 references
        public void GivenModelIs(string model)
        {
            _model = model;
        }

        [Given("the quantity is (.*)")]
        0 references
        public void GivenTheQuantityIs(int quantity)
        {
            _quantity = quantity;
        }

        [When("the order is made")]
        0 references
        public void WhenTheOrderIsMade()
        {
            _item.Model = _model;
            _item.Quantity = _quantity;
        }

        [Then("an order containing model iPhone 8 and quantity 2")]
        0 references
        public void ThenTheResultShouldBe()
        {
            Console.WriteLine($"Order is made with Model: " +
                              $"{_item.Model} and Quantity: {_item.Quantity}");
        }
    }
}

```

This is of course an example of a not-implemented/finished product, and should only be examined for conceptual use.

Basically, we connect the Gherkin Syntax to this test setup by using [Binding] and the keywords as seen in the gherkin, [Given] [When] [Then] ..

Below establishes that this block is connected to the specified Given (.) allows it to read the last bit of the Given sentence in the feature file. Meaning you could write it with a different model. Same goes for the quantity.

```
[Given("the model ordered is an (.*)")]
0 references
public void GivenModelIs(string model)
{
    _model = model;
}
```

Our When and Then statements are again, fairly simple and not a real test - keep in mind, this is more for the demonstration purpose:

```
[When("the order is made")]
0 references
public void WhenTheOrderIsMade()
{
    _item.Model = _model;
    _item.Quantity = _quantity;
}

[Then("an order containing model iPhone 8 and quantity 2")]
0 references
public void ThenTheResultShouldBe()
{
    Console.WriteLine($"Order is made with Model: " +
        $"{_item.Model} and Quantity: {_item.Quantity}");
}
```

Basically, in the When, we create the specification, and then in "then", which should be where you would verify your actual result to an expected result (assert equal for instance), we just print the result out.