2

Test-Driven Development in .NET—By Example

In this chapter, we'll demonstrate how to implement a *Stack* using Test-Driven Development (TDD). We have found that the best way to understand TDD is to see it practiced and follow along step by step. The following are the steps we used to build a *Stack* using this method.

The Task

The task is to implement an unbounded *Stack*, which is a data structure in which access is restricted to the most recently inserted item.

> **Note** An *unbounded Stack* doesn't have to be presized, and you can insert an unlimited number of elements onto it.

The operations include *Push*, *Pop*, *Top,* and *IsEmpty*. The *Push* function inserts an element onto the top of the *Stack*. The *Pop* function removes the topmost element and returns it; the *Top* operation returns the topmost element but does not remove it from the *Stack*. The *IsEmpty* function returns true when there are no elements on the *Stack*. Figure 2-1 shows the *Push* operation in action, Figure 2-2 shows the *Pop* operation, and Figure 2-3 shows the *Top* operation.

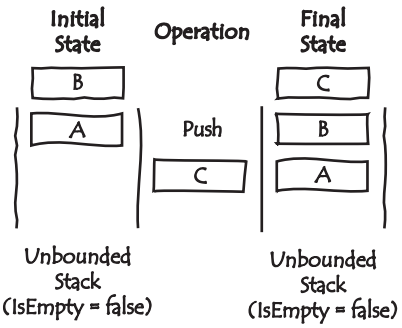

Figure 2-1 *Push* operation

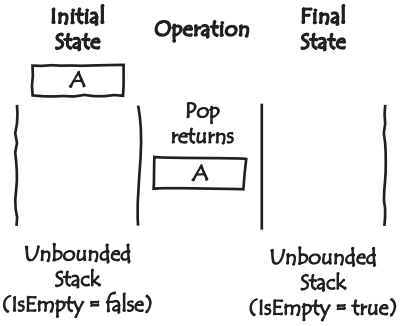

Figure 2-2 *Pop* operation

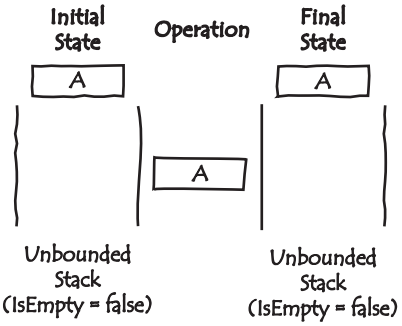

Figure 2-3 *Top* operation

Test List[1]

In Chapter 1, "Test-Driven Development Practices," we stated that the first step is to brainstorm a list of tests for the task. The goal of this activity is to create a

1. Beck, Kent. *Test-Driven Development: By Example.* Addison-Wesley, 2003.

test list that verifies the detailed requirements and describes the completion criteria. One thing to keep in mind is that the list is not static. As you implement each test, you might have to revisit the list to add new tests or delete them as appropriate.

Let's try to write the test list for the unbounded *Stack*.

Unbounded *Stack* Test List

- Create a *Stack* and verify that *IsEmpty* is true.
- *Push* a single object on the *Stack* and verify that *IsEmpty* is false.
- *Push* a single object, *Pop* the object, and verify that *IsEmpty* is true.
- *Push* a single object, remembering what it is; *Pop* the object, and verify that the two objects are equal.
- *Push* three objects, remembering what they are; *Pop* each one, and verify that they are removed in the correct order.
- *Pop* a *Stack* that has no elements.
- *Push* a single object and then call *Top*. Verify that *IsEmpty* is false.
- *Push* a single object, remembering what it is; and then call *Top*. Verify that the object that is returned is the same as the one that was pushed.
- Call *Top* on a *Stack* with no elements.

Choosing the First Test

There are differing opinions about which test to choose first. One says that you should choose the simplest test that gets you started and solves a small piece of the problem. Another says that you should choose a test that describes the essence of what you are trying to accomplish. For example, looking at the test list in the previous section, the simplest test is the first one: Create an empty *Stack* and verify that *IsEmpty* is true. This operation looks as if it would be easy to implement, but it does not provide a great deal of useful feedback when developing a *Stack* because the *IsEmpty* function is a supporting function.

A test in the list that is closer to the essence of the problem is the following: *Push* a single object, remembering what it is; *Pop* the object, and verify that it is equal to the object that was pushed. In this test, you are verifying that the *Push* and *Pop* methods work as expected. Which style to use really is a matter of personal preference because both will work.

There are times when the essence approach can take too much time to implement. If that is the case, you should choose a simpler test to get started. The suggestion that we give people who are learning TDD is to choose the simplest test approach and graduate to the essence approach after becoming familiar with the technique. Therefore, the first test that we chose to implement is "Create an empty *Stack* and verify that *IsEmpty* is true."

Red/Green/Refactor

The following is the implementation of each test in the test list.

Test 1: Create a *Stack* and verify that *IsEmpty* is true.

This test requires creating a *Stack* and then calling the *IsEmpty* property. The *IsEmpty* property should return true because we haven't put any elements into the *Stack*. Let's create a file called StackFixture.cs, in which we write a test fixture class, called *StackFixture*, to hold the tests.

```
using System;
using NUnit.Framework;

[TestFixture]
public class StackFixture
{ /* … */ }
```

There are a few things of interest about this class. The line *using NUnit.Framework;* is needed to reference the custom attributes defined in NUnit that are used to mark the test fixture. The *[TestFixture]* attribute can be associated only with a class. This attribute is an indicator to NUnit that the class contains test methods.

The next activity is to write the method that does the test. (Here, the test method name is *Empty*.)

```
[Test]
public void Empty()
{
    Stack stack = new Stack();
    Assert.IsTrue(stack.IsEmpty);
}
```

The test method is marked with the attribute *[Test]*. The first thing is to create a *Stack* object. After creating the object, we use the *Assert.IsTrue(…)* method to verify that the return value of *IsEmpty* is true.

Although the class used in the test, *Stack*, and the property *IsEmpty* do not exist, we are writing test code as if they do. We are thinking about how the class and its methods are used instead of how to implement it, which is an important distinction. This is why many people refer to test-driven development as much a design technique as a testing technique. Many times, class library designers implement a library and then figure out how to use it, which can lead to libraries that require a lot of initialization, complex method interactions, and increased dependencies. Thinking about how to use the library before implementing it places a greater emphasis on usage, which often leads to better design.

Because the *Stack* class does not exist, the test does not compile. That's easy enough to fix. What is the smallest amount of work that needs to be done to get this to compile?

```
using System;

public class Stack
{
    private bool isEmpty = true;

    public bool IsEmpty
    {
        get
        {
            return isEmpty;
        }
    }
}
```

This implementation is certainly small; in fact, it might seem surprising. Remember that the goal is to do the smallest amount of work possible to get the code to compile. Some people might say that this code is too complicated: given the test, they might argue that the following code would be sufficient:

```
    public bool IsEmpty
    {
        get
        {
            return true;
        }
    }
```

There is a balance to achieve between anticipating future tests and implementation and being totally ignorant of the next test. In the beginning, you should focus on the test you are writing and not think about the other tests. As you become familiar with the technique and the task, you can increase the size of the steps. You should always keep in mind that large steps are harder to

debug than smaller steps. Also, if your code is too complicated or provides functionality that is not tested, additional refactorings can result later.

This discussion is also relevant to the earlier discussion about the test list. It is very clear from the test list that you have to store multiple items. Should you go ahead and use an *ArrayList* because you might need it later? No—the current tests do not support the need for an *ArrayList*. Wait and see what the tests look like before making that decision.

Now that the code compiles, it is time to run the test in NUnit. The green bar displays, which indicates success. We can check off the first test and move on.

Which test should we choose next? Perhaps we should stay focused on the *IsEmpty* property because it is probably the smallest increment over what we have now. Let's look at "*Push* a single object on the *Stack* and verify that *IsEmpty* is false."

Test 2: *Push* a single object on the *Stack* and verify that *IsEmpty* is false.

Test 2 says to *Push* an object onto the *Stack* and then verify that *IsEmpty* returns false. Let's try and write a test that does this. We'll call the method *PushOne*:

```
[Test]
public void PushOne()
{
   Stack stack = new Stack();
   stack.Push("first element");
   Assert.IsFalse(stack.IsEmpty,
     "After Push, IsEmpty should be false");
}
```

The test, like the previous one, creates a *Stack* object. Then, using a method named *Push* puts a *String* object onto the *Stack*. Finally, we call the *IsEmpty* property on *Stack* and verify that it returns false.

Of course, this code does not compile because we have not defined the *Push* method. Once again, what is the minimal amount of work needed to get this code to compile?

```
public void Push(object element)
{
}
```

That is as small as it gets. Now, we can run the test. Running the tests yields the following result:

```
Tests run: 2, Failures: 1, Not run: 0, Time: 0.015627 seconds

Failures:
1) StackFixture.PushOne : After Push, IsEmpty should be false
   at StackFixture.PushOne() in c:\stackfixture.cs:line 19
```

The test failed because it was expecting the *IsEmpty* property of the *Stack* to return false and it returned true. Now that we have a failing test, we can implement *Push* correctly. Clearly, we need to change the *isEmpty* member variable to be false when an element is pushed onto the *Stack*. The *Push* method also implies that we have to store the elements in some collection to satisfy the other operations, but there are no tests for this, so we will wait until we have tests to implement this behavior. Let's change the *Push* method to implement this correctly.

```
public void Push(object element)
{
     isEmpty = false;
}
```

Before we decide which test to implement next, we need to make sure that there is no code duplication. The *StackFixture* class has some duplicated test code. The test code is just as important as the production code; it is critical to the overall communication and it also serves as an example of how the client code should work. Both the *PushOne* and *Empty* tests create a *Stack* in the first line of their methods. So, let's move the creation of the *Stack* object from the test methods to a new method called *Init*. After the modifications are completed, the *StackFixture* class looks like this:

```
using System;
using NUnit.Framework;

[TestFixture]
public class StackFixture
{
    private Stack stack;

    [SetUp]
    public void Init()
    {
        stack = new Stack();
    }

    [Test]
    public void Empty()
```

```
    {
        Assert.IsTrue(stack.IsEmpty);
    }

    [Test]
    public void PushOne()
    {
        stack.Push("first element");
        Assert.IsFalse(stack.IsEmpty,
               "After Push, IsEmpty should be false");
    }
}
```

We had to create a private instance variable called *stack* so that all the methods of the class could access the same object. The function *Init* is marked with an attribute called [*SetUp*]. NUnit uses this attribute to ensure that this method is called prior to each test being run, which means that each test method gets a newly created *Stack*, instead of one modified from a previous test.

This is an excellent example of the second rule of TDD: eliminate duplication. Prior to the refactoring, there was a small amount of duplicated code in *StackFixture*. It may seem trivial, but it serves as an example of how following red/green/refactor leads to code that is cleaner and easier to understand. It also has the added benefit of making the tests easier to maintain because changes to the initialization code would be made in one place.

Because the code compiled and passed the tests, it's time to move on to the next test. We want to stay focused on the *IsEmpty* property, so *Push* a single object, *Pop* the object and verify that *IsEmpty* is true seems like a natural.

Test 3: *Push* a single object, *Pop* the object, and verify that *IsEmpty* is true.

This test introduces a new method called *Pop*, which returns the topmost element and removes it from the *Stack*. To test that behavior, we need to insert an element onto the *Stack* and then remove it. After that sequence is completed, calling *IsEmpty* on the *Stack* should be true. Let's see what that test looks like:

```
[Test]
public void Pop()
{
    stack.Push("first element");
    stack.Pop();
    Assert.IsTrue(stack.IsEmpty,
        "After Push - Pop, IsEmpty should be true");
}
```

Of course, the code does not compile because we haven't defined the method *Pop*. So we'll fake it:[2]

```
public void Pop()
{
}
```

The code compiles, but the tests fail with the following message:

```
Tests run: 3, Failures: 1, Not run: 0, Time: 0.0156336 seconds

Failures:
1) StackFixture.Pop : After Push - Pop, IsEmpty should be true
   at StackFixture.Pop() in c:\stackfixture.cs:line 33
```

We now need to implement the *Pop* method so that it passes the test. We will simply change *IsEmpty* to be true when *Pop* is called.

```
public void Pop()
{
    isEmpty = true;
}
```

The code compiles and we run the tests. The tests pass, so we can check off Test 3 on the test list.

Notice that the implemented *Pop* method returns *void*. The requirements stated previously said that *Pop* should also return the topmost element. Because we do not have a test that tests that functionality, we will leave it that way until we have a failing test.

So far, we have been concerned with verifying that the *IsEmpty* property on the *Stack* is correct in regard to the *Push* and *Pop* operations. However, this is leading to code that does not further our understanding of the problem. In fact, we have written three tests that manage a Boolean variable. It is now time to change direction and look at the actual objects that are pushed and popped onto the *Stack*.

Test 4: *Push* a single object, remembering what it is; *Pop* the object, and verify that the two objects are equal.

In this test, we need to create an object (in this case, an *int*), push the object onto the *Stack*, pop the *Stack*, and verify that the object that is returned is equal to the object pushed on the *Stack*. The following is the test method *PushPop-ContentCheck*:

2. Beck, Kent. *Test-Driven Development: By Example.* Addison-Wesley, 2003.

```
[Test]
public void PushPopContentCheck()
{
    int expected = 1234;
    stack.Push(expected);
    int actual = (int)stack.Pop();
    Assert.AreEqual(expected, actual);
}
```

Of course, this code does not compile. The *Pop* method returns *void*, not *object*. So let's change the *Pop* method to return an *object*. The simplest code is to have it return *null*:

```
public object Pop()
{
    isEmpty = true;
    return null;
}
```

Let's compile and run the tests. The code compiles, but the tests fail with the following message:

```
Tests run: 4, Failures: 1, Not run: 0, Time: 0.0156311 seconds

Failures:

1) StackFixture.PushPopContentCheck : System.NullReferenceException : Object
reference not set to an instance of an object.
   at StackFixture.PushPopContentCheck() in c:\stackfixture.cs:line 42
```

The test failed because we did not return the value that was on the top of the *Stack*. In order for this test to pass, we have to change the *Push* method to retain the object and alter the *Pop* method to return the object. The following code is our next attempt:

```
using System;

public class Stack
{
    private bool isEmpty = true;
    private object element;

    public bool IsEmpty
    {
        get
        {
            return isEmpty;
        }
    }
```

```
    public void Push(object element)
    {
        this.element = element;
        isEmpty = false;
    }

    public object Pop()
    {
        isEmpty = true;
        object top = element;
        element = null;

        return top;
    }
}
```

Let's compile and run the tests. All the tests pass, so we can now mark this test off the list. Before we move on, let's do a little refactoring because there is a change that could be made related to the *isEmpty* member variable. The change that we will make is to remove the *isEmpty* member variable and replace it with a conditional expression using the *element* member variable. Here is the modified code for the *Stack*:

```
using System;

public class Stack
{
    private object element;

    public bool IsEmpty
    {
        get
        {
            return (element == null);
        }
    }

    public void Push(object element)
    {
        this.element = element;
    }

    public object Pop()
    {
        object top = element;
        element = null;

        return top;
    }
}
```

This is much better because we use the *element* variable itself to represent whether the *Stack* is empty or not. Prior to this, we had to update two variables; now we use only one variable, which makes this solution better and we can move on. One of the key behaviors of the *Stack* is that when you push an element onto the *Stack,* it becomes the topmost element. Therefore, if you push more than one item, the *Stack* should push the top item down and replace the top item with the newly pushed item. This behavior should be consistent, no matter how many items have been pushed.

We want the next test to verify that the *Stack* works as expected.

Test 5: *Push* three objects, remembering what they are; *Pop* each one, and verify that they are correct.

The previous test, *PushPopContentCheck*, pushed and popped only one item. In this test, we have to push three items to ensure that the *Stack* behaves in the correct fashion:

```
[Test]
public void PushPopMultipleElements()
{
    string pushed1 = "1";
    stack.Push(pushed1);
    string pushed2 = "2";
    stack.Push(pushed2);
    string pushed3 = "3";
    stack.Push(pushed3);

    string popped = (string)stack.Pop();
    Assert.AreEqual(pushed3, popped);
    popped = (string)stack.Pop();
    Assert.AreEqual(pushed2, popped);
    popped = (string)stack.Pop();
    Assert.AreEqual(pushed1, popped);
}
```

In the *PushPopMultipleElements* method, we push 3 items onto the *Stack* and then pop them off and verify that they are in the correct order. In this example, we push "*1*", "*2*", and "*3*" in that order and verify that when we call *Pop* repeatedly, the strings come off "*3*", "*2*", and "*1*".

Let's compile and run the tests. The code compiles, but NUnit fails with the following message:

```
Tests run: 5, Failures: 1, Not run: 0, Time: 0.031238 seconds

Failures:
```

```
1) StackFixture.PushPopMultipleElements :
        expected:<"2">
         but was:<(null)>
   at StackFixture.PushPopMultipleElements() in c:\stackfixture.cs:line 59
```

Clearly, something is wrong. In fact, we can no longer use the simplistic implementation of the *Stack* with a single element. We need to use a collection to hold the elements inside the *Stack*. After making a series of changes, here is the refactored *Stack* code:

```
using System;
using System.Collections;

public class Stack
{
    private ArrayList elements = new ArrayList();

    public bool IsEmpty
    {
        get
        {
            return (elements.Count == 0);
        }
    }

    public void Push(object element)
    {
        elements.Insert(0, element);
    }

    public object Pop()
    {
        object top = elements[0];
        elements.RemoveAt(0);
        return top;
    }
}
```

Let's compile and run the tests. The code compiles and all the tests pass, so we can move on. You should notice that we made big changes to the entire implementation, and no tests were broken afterward. This is a good example of building confidence in the code that we have tests for. We are assured that, based on our tests, the code is no worse than it was previously (which is the best the tests can demonstrate).

This example shows very well the benefits of delaying implementation decisions while writing tests. Some would argue that we should have started out using an *ArrayList* to hold the elements of the *Stack* because it was a fore-

gone conclusion that we would need a collection to hold the elements. We did not do this because we are trying to let the tests drive the need for functionality instead of us thinking we know what is needed and then writing tests that verify that thinking. It is a difference that this test demonstrates very clearly.

So far, we have been careful to call *Pop* on a *Stack* only when it contains elements. In the next test, we need to look at what happens when we call *Pop* on a *Stack* that has no elements.

Test 6: *Pop* a *Stack* that has no elements.

What should happen if we call *Pop* on the *Stack* and there are no elements? There are a number of options:

- We could return *null* for the value (although we could never store *null* on the *Stack*).
- We could use an in/out parameter to indicate success or failure of the *Pop* operation. This procedure is clumsy and requires the user to check the value to determine whether the method was successful.
- We could throw an exception because it's an error we don't expect to occur.

Reviewing the options, it seems to make the most sense to have the *Pop* method throw an exception if there are no elements on the *Stack*. Let's write a test that expects the *Pop* operation to throw an exception:

```
[Test]
[ExpectedException(typeof(InvalidOperationException))]
public void PopEmptyStack()
{
   stack.Pop();
}
```

This test uses another attribute in NUnit that allows the programmer to declare that the execution of the test is expected to throw an exception. We could define a new exception, but we choose to use *InvalidOperationException* in this example. It is in the *System* namespace and is defined as "The exception that is thrown when a method call is invalid for the object's current state."[3]

We compiled and ran the tests. The test failed with the following message:

3. .NET SDK Documentation, System.InvalidOperationException

```
Tests run: 6, Failures: 1, Not run: 0, Time: 0.0156248 seconds

Failures:
1) StackFixture.PopEmptyStack : Expected: InvalidOperationException but was Arg
umentOutOfRangeException
   at System.Collections.ArrayList.get_Item(Int32 index)
   at Stack.Pop() in c:\projects\book\stack\stack.cs:line 23
   at StackFixture.PopEmptyStack() in c:\stackfixture.cs:line 68
```

Not surprisingly, it does not work; we never changed the *Pop* method to return the exception. However, we did get an exception, just not the right one. The *ArgumentOutOfRangeException* occurs when you access an array outside of the allowable range of values. In this case, there were no elements in the array. Clearly, we need to modify the *Pop* method to check to see whether there are any elements in the *Stack* and if not, throw the *InvalidOperationException*. Let's modify the *Pop* method to throw the correct exception:

```
public object Pop()
{
    if(IsEmpty) throw new
            InvalidOperationException("cannot pop an empty stack");

    object top = elements[0];
    elements.RemoveAt(0);
    return top;
}
```

That works. We can now check this test off the list and figure out which test to do next.

As we were implementing this test, a few additional tests came to mind, so we need to add them to our *test list*. We want to add tests to verify that the *Stack* works when the arguments are equal to *null*. The new tests are as follows:

- *Push null* onto the *Stack* and verify that *IsEmpty* returns false.
- *Push null* onto the *Stack*, *Pop* the *Stack*, and verify that the value returned is *null*.
- *Push null* onto the *Stack*, call *Top*, and verify that the value returned is *null*.

Reviewing the test list indicates that we have not done anything yet with the *Top* method, so let's focus on that next.

Test 7: *Push* a single object and then call *Top*. Verify that *IsEmpty* returns false.

The *Top* method does not change the state of the *Stack*; it simply returns the topmost element. This test verifies that *IsEmpty* is not affected by the call to *Top*. Let's write that test:

```
[Test]
public void PushTop()
{
    stack.Push("42");
    stack.Top();
    Assert.IsFalse(stack.IsEmpty);
}
```

Of course, this code does not compile. We have not written the *Top* method, so we'll fake it:

```
public object Top()
{
   return null;
}
```

That works. So we'll check off another item on the list and decide which test to implement next.

Not so fast! There are a couple of tests that we need to add to the test list that we thought of while doing this test.

- *Push* multiple items onto the *Stack* and verify that calling *Top* returns the correct object.
- *Push* an item on the *Stack*, call *Top* repeatedly, and verify that the object returned each time is equal to the object that was pushed onto the *Stack*.

The test list now contains 14 items. These last two tests are important because they verify that *Top* works as expected. We didn't think of them at the beginning, but they came to mind when we started working on *Top*.

The next test we will implement verifies that *Top* returns the correct object.

Test 8: *Push* a single object, remembering what it is; and then call *Top*. Verify that the object that is returned is equal to the one that was pushed.

In the previous test, we checked to see whether the *IsEmpty* property was correct after we called *Top*. In this test, we verify that the object pushed onto the *Stack* is equal to the one we get back when we call *Top*:

```
[Test]
public void PushTopContentCheckOneElement()
{
   string pushed = "42";
   stack.Push(pushed);
   string topped = (string)stack.Top();
   Assert.Equals(pushed, topped);
}
```

Let's compile and run the tests. NUnit protests with the following message:

```
Tests run: 8, Failures: 1, Not run: 0, Time: 0.0312442 seconds

Failures:
1) StackFixture.PushTopContentCheckOneElement :
        expected:<"42">
         but was:<(null)>
   at StackFixture.PushTopContentCheckOneElement()
     in c:\stackfixture.cs:line 84
```

In the previous test, we faked the implementation of *Top* by just returning *null*. Looks like we have to implement (make) it correctly for this test to pass:

```
public object Top()
{
   return elements[0];
}
```

That works. Although *Top* seems similar to *Pop*, let's wait and see whether it gets more obvious as we add additional tests. Let's write another test.

Test 9: *Push* multiple objects, remembering what they are; call *Top*, and verify that the last item pushed is equal to the one returned by *Top*.

This test states that we need to push more than one object onto the *Stack* and then call *Top*. The return value of *Top* should be equal to the last value that was pushed onto the *Stack*. Let's give it a try:

```
[Test]
public void PushTopContentCheckMultiples()
{
    string pushed3 = "3";
    stack.Push(pushed3);
    string pushed4 = "4";
    stack.Push(pushed4);
    string pushed5 = "5";
    stack.Push(pushed5);

    string topped = (string)stack.Top();
    Assert.AreEqual(pushed5, topped);
}
```

That works. This one just happens to work, so good for us. The next test verifies that calling *Top* repeatedly always returns the same object.

Test 10: *Push* one object and call *Top* repeatedly, comparing what is returned to what was pushed.

As stated previously, the *Top* method is not supposed to change the state of the object, so we should be able to push an object onto the *Stack* and then call *Top* as many times as we want—and it should always return the same object. Let's code it and see whether it works:

```
[Test]
public void PushTopNoStackStateChange()
{
    string pushed = "44";
    stack.Push(pushed);

    for(int index = 0; index < 10; index++)
    {
        string topped = (string)stack.Top();
        Assert.AreEqual(pushed, topped);
    }
}
```

That works, too.

Let's move on. The next test determines what happens when we call *Top* on a *Stack* that has no elements.

Test 11: Call *Top* on a *Stack* that has no elements.

Consistency is a key component of designing a class library. Because we chose to throw an *InvalidOperationException* when we called *Pop*, we should be consistent and throw the same exception when we call *Top*. Let's write the test:

```
[Test]
[ExpectedException(typeof(InvalidOperationException))]
public void TopEmptyStack()
{
   stack.Top();
}
```

Of course, this does not work. NUnit provides the details:

```
Tests run: 11, Failures: 1, Not run: 0, Time: 0.031263 seconds

Failures:
1) StackFixture.TopEmptyStack : Expected: InvalidOperationException but was
```

```
ArgumentOutOfRangeException
   at System.Collections.ArrayList.get_Item(Int32 index)
   at Stack.Top() in c:\projects\book\stack\stack.cs:line 33
   at StackFixture.TopEmptyStack() in c:\stackfixture.cs:line 119
```

You would think we would learn something—this is the same failure we got when we first implemented *Pop*. We need something similar in *Top*:

```
public object Top()
{
    if(IsEmpty) throw new
        InvalidOperationException("cannot top an empty stack");

    return elements[0];
}
```

This works. However, the similarity between *Top* and *Pop* is very apparent and needs to be refactored. They both check to see whether there are elements in the list and throw an exception if there aren't any. The best solution seems to have *Pop* call *Top*. Let's give that a try:

```
public object Pop()
{
    object top = Top();
    elements.RemoveAt(0);
    return top;
}

public object Top()
{
    if(IsEmpty)
        throw new InvalidOperationException("Stack is Empty");

    return elements[0];
}
```

This works: the duplication has been removed. We did have to make a change to the message that was in the exception—it is more generic now, which is a small price to pay for consistency in the code. We can now check this test off the list. (There are only three more to go, so the end is in sight.)

Test 12: *Push null* onto the *Stack* and verify that *IsEmpty* is false.

This is the first test that was added when we wrote the test that called *Pop* on an empty *Stack*. As you might recall, one of the options was to return a *null* to indicate that there were no elements in the *Stack*. If we had chosen that route,

the programmer couldn't have stored *null* on the *Stack*, so the interface would be less explicit.

This test pushes *null* onto the *Stack* and verifies that *IsEmpty* is false:

```
[Test]
public void PushNull()
{
    stack.Push(null);
    Assert.IsFalse(stack.IsEmpty);
}
```

This works just fine. Let's do the next test.

Test 13: *Push null* onto the *Stack*, *Pop* the *Stack*, and verify that the value returned is *null*.

The previous test verified that *IsEmpty* was correct after pushing *null* onto the *Stack*. In this test, we push a *null* object onto the *Stack* and then call *Pop* to retrieve the element and remove it from the *Stack*. The value returned from *Pop* should be equal to *null*.

```
[Test]
public void PushNullCheckPop()
{
    stack.Push(null);
    Assert.IsNull(stack.Pop());
    Assert.IsTrue(stack.IsEmpty);
}
```

Let's compile and run the test. It works! This test has two asserts: one to check whether the return value is *null*, and one to check that *IsEmpty* is true. They could be done separately, but we chose to combine them because they are associated with the same test setup.

Now there is only one test left. Does the *Top* method work when we push *null* onto the *Stack*?

Test 14: Push *null* onto the *Stack*, call *Top*, and verify that the value returned is *null*.

In this test, we push a *null* object onto the *Stack* and then call *Top* to retrieve the element from the *Stack*. The value returned from *Top* should be equal to *null*.

```
[Test]
public void PushNullCheckTop()
{
```

```
        stack.Push(null);
        Assert.IsNull(stack.Top());
        Assert.IsFalse(stack.IsEmpty);
    }
```

Compiling and running the tests indicate success. This is the last test on the list, and we can't think of any others. Although there are probably things we missed, we can't think of any for now—so the task is complete. We can check the code into the repository and work on the next task. As bugs are found, we will add other tests to cover the cases; this ensures that the test coverage will improve over time.

Summary

In this chapter, we built a *Stack* as an example of Test-Driven Development, and our version resulted in 14 tests. Reviewing the code shows that there is much more test code than actual code.

If you study the individual steps, you probably notice that we spent most of our time writing tests instead of writing the *Stack* code. This is because when we write tests, we focus on what the class does and how it is used instead of how it is implemented. This emphasis is very different from other ways of writing software, in which the code is written and then we figure out how to use it.

Comparing the size of the two classes is interesting. The *Stack* class is 35 lines of code; the *StackFixture* class contains 144 lines of code. The test code is more than four times the size of the *Stack* code. The following is the completed code in its entirety:

StackFixture.cs

```
using System;
using NUnit.Framework;

[TestFixture]
public class StackFixture
{
    private Stack stack;

    [SetUp]
    public void Init()
    {
```

```
        stack = new Stack();
    }

    [Test]
    public void Empty()
    {
        Assert.IsTrue(stack.IsEmpty);
    }

    [Test]
    public void PushOne()
    {
        stack.Push("first element");
        Assert.IsFalse(stack.IsEmpty,
            "After Push, IsEmpty should be false");
    }

    [Test]
    public void Pop()
    {
        stack.Push("first element");
        stack.Pop();
        Assert.IsTrue(stack.IsEmpty,
            "After Push - Pop, IsEmpty should be true");
    }

    [Test]
    public void PushPopContentCheck()
    {
        int expected = 1234;
        stack.Push(expected);
        int actual = (int)stack.Pop();
        Assert.AreEqual(expected, actual);
    }

    [Test]
    public void PushPopMultipleElements()
    {
        string pushed1 = "1";
        stack.Push(pushed1);
        string pushed2 = "2";
        stack.Push(pushed2);
        string pushed3 = "3";
        stack.Push(pushed3);

        string popped = (string)stack.Pop();
        Assert.AreEqual(pushed3, popped);
        popped = (string)stack.Pop();
```

```
    Assert.AreEqual(pushed2, popped);
    popped = (string)stack.Pop();
    Assert.AreEqual(pushed1, popped);
}

[Test]
[ExpectedException(typeof(InvalidOperationException))]
public void PopEmptyStack()
{
    stack.Pop();
}

[Test]
public void PushTop()
{
    stack.Push("42");
    stack.Top();
    Assert.IsFalse(stack.IsEmpty);
}

[Test]
public void PushTopContentCheckOneElement()
{
    string pushed = "42";
    stack.Push(pushed);
    string topped = (string)stack.Top();
    Assert.AreEqual(pushed, topped);
}

[Test]
public void PushTopContentCheckMultiples()
{
    string pushed3 = "3";
    stack.Push(pushed3);
    string pushed4 = "4";
    stack.Push(pushed4);
    string pushed5 = "5";
    stack.Push(pushed5);

    string topped = (string)stack.Top();
    Assert.AreEqual(pushed5, topped);
}

[Test]
public void PushTopNoStackStateChange()
{
    string pushed = "44";
    stack.Push(pushed);
```

```
            for(int index = 0; index < 10; index++)
            {
                string topped = (string)stack.Top();
                Assert.AreEqual(pushed, topped);
            }
        }

        [Test]
        [ExpectedException(typeof(InvalidOperationException))]
        public void TopEmptyStack()
        {
            stack.Top();
        }

        [Test]
        public void PushNull()
        {
            stack.Push(null);
            Assert.IsFalse(stack.IsEmpty);
        }

        [Test]
        public void PushNullCheckPop()
        {
            stack.Push(null);
            Assert.IsNull(stack.Pop());
            Assert.IsTrue(stack.IsEmpty);
        }

        [Test]
        public void PushNullCheckTop()
        {
            stack.Push(null);
            Assert.IsNull(stack.Top());
            Assert.IsFalse(stack.IsEmpty);
        }
    }
```

Stack.cs

```
using System;
using System.Collections;

public class Stack
{
    private ArrayList elements = new ArrayList();

    public bool IsEmpty
```

```
    {
        get
        {
            return (elements.Count == 0);
        }
    }

    public void Push(object element)
    {
        elements.Insert(0, element);
    }

    public object Pop()
    {
        object top = Top();
        elements.RemoveAt(0);
        return top;
    }

    public object Top()
    {
        if(IsEmpty)
            throw new InvalidOperationException("Stack is Empty");

        return elements[0];
    }
}
```