Test-Driven Design

Using Mocks And Tests To Design Role-Based Objects

Isaiah Perumalla

Code download available at:MSDN Code Gallery

This article discusses:

- Testing interaction, not implementation
- Discovering roles and abstracting conversations
- Abstracting conversations
- Refactoring code to clarify intent

This article uses the following technologies:
Test-Driven Development, NMock Framework

Contents

Interaction, Not Implementation
Reading Barcodes
Discovering Roles
Finishing the Sale
Abstracting Conversations
Calculating Receipts
Getting the Product Description
Refactoring
Wrapping Up

Within the realm of Test-Driven Development (TDD), mock objects can help you discover the roles that objects should play within a system, emphasizing how the objects relate to each rather than their internal structure. This technique can be employed to support good object-oriented design. Using mock objects as an aid to design turns out to be much more interesting than the common practice of using them simply to isolate a system from external dependencies.

One of the most important benefits of TDD is that it improves the overall design of your code by forcing you to think about the design of an object's interface based on its intended use rather than its implementation. Mock objects complement the TDD process of object-oriented systems, by allowing you to write the test code for an object as though it already had everything 3 from its environment. You do this by filling in the place of an object's collaborators with mocks. This lets you design the interfaces of an object's collaborators in terms of the roles they play, before any concrete implementations of them even exist. This leads to a process of discovery where the interfaces of an object's collaborators are brought into existence based on immediate requirements, driven by need.

Following this process of test-first development using mock objects, you can not only design an object's interface from its intended use, but you can also discover and design the interfaces an object needs from its collaborators.

This article describes how to use TDD with mock objects to design object-oriented code in terms of roles and responsibilities, not categorization of objects into class hierarchies.

Interaction, Not Implementation

One of the fundamental principles of object-oriented programming is to localize all logic operating on state to the object holding the state, and to hide an object's internal structures and their state transitions. The emphasis should be on how objects communicate with other objects in the environment when triggered by an event. In practice, this can be hard to achieve. The result of designing objects this way is that each object exposes no visible state nor any of its internal structure. Since there is no visible state, you cannot query any internal structures or state to test the object's behavior by making assertions upon its state. The only thing visible is the way the object interacts with its environment. However, you can track these interactions to verify an object behavior.

Using mock objects lets you discover how an object interacts with its collaborators by making assertions that an object sends the right message to its collaborators in a given scenario. This shifts focus and design effort from how objects are classified and structured to how objects communicate with each other. This, in turn, drives the design of your system to a "Tell, don't ask" design style where each object knows very little about the structure or state of its surrounding objects. This makes the system much more flexible by allowing you to change the behavior of the system by composing different sets of objects.

To illustrate this technique, I will walk you through a short example that demonstrates TDD using mock objects.

Reading Barcodes

I am building a point-of-sale system for a large supermarket chain. The product catalog systems are in the head office and accessed by a RESTful service. The main focus for the first feature is for the system to use barcode information, either manually entered or read from a barcode scanner, to identify an item, retrieve its price, and calculate the total receipt for the sale.

MSDN Magazine Blog

MSDN Magazine June Issue Preview
Monday morning the June issue of MSDN Magazine will go live on our Web site. Here's what you can expect to find in the magazine. Windows 8 figures pr... More...
Friday, May 31

MSDN Magazine May Issue Preview
Tomorrow afternoon the May issue of MSDN Magazine will go live on our Web site. Here's what you can look forward to in the upcoming issue. Craig Shoe... More...
Tuesday, Apr 30

More MSDN Magazine Blog entries >

Current Issue

Browse All MSDN Magazines

Subscribe to MSDN Flash newsletter

Receive the MSDN Flash e-mail newsletter every other week, with news and information

The system is triggered when the cashier enters commands using a touch screen or a barcode scanner. Commands from input devices are sent as a string in the following format:

- Command: NewSale;
- Command: EndSale;
- Input: Barcode=100008888559, Quantity =1;

All barcodes follow the UPC scheme; the first six characters of the barcode identify the manufacturer code, and the next five characters identify a product code. The product information system requires the manufacturer code and product code to be able to retrieve product description for an item.

The system first needs to receive and interpret commands from input devices (keyboard and scanners). When the sale is finished, the system should calculate and print a receipt using the product catalog from the head office to retrieve product description and price.

The first step is to decode the raw messages sent from the input devices into something that represents check-out events. I start with the simplest command. The initiating new sale command simply triggers a new sale event within the system. I need an object to decode the raw messages from the input devices and convert them into something that represents an event in terms of the application domain. The first test is shown below:

```
[TestFixture] public class CommandParserTests { [Test] public void
NotifiesListenerOfNewSaleEvent() { var commandParser = new CommandParser(); var
newSaleCommand= "Command:NewSale"; commandParser.Parse(newSaleCommand); } }
```

Note that CommandParser doesn't even exist at this point. I am using the tests to drive out the interface this object provides.

How would I know whether this object interpreted the command sent from input devices correctly? Following the "tell, don't ask" principle, the CommandParser object should tell the object it has a relationship with that a new sale was initiated. At this point, I don't know who or what it has a relationship with. This is yet to be discovered.

Discovering Roles

So far, all I know about the CommandParser's collaborator is that it needs to know when events related to a sale are detected in the system. I will choose a name that describes just this functionality—SaleEventListener—and note that it represents a role in the system. A role can be viewed as a named slot in a software system that can be filled by any object that can fulfill the responsibilities of the role. To discover roles, you need to examine the visible interactions between an object and its collaborator when performing an activity in the system, focusing only on the aspects of the objects required for the activity to be described.

In C#, a role can be specified by an interface. In this example, the CommandParser requires an object from its environment that can play the role of SaleEventListener. However, interfaces alone are inadequate to describe how a group of objects communicate to fulfill a task. Using mock objects, I can describe this in my test as shown in **Figure 1**.

Figure 1 Defining the SaleEventListener Role

```
[TestFixture] public class CommandParserTests { private Mockery mockery; [SetUp]
public void BeforeTest() { mockery = new Mockery(); } [Test] public void
NotifiesListenerOfNewSaleEvent() { var saleEventListener =
mockery.NewMock<ISaleEventListener>(); var commandParser = new CommandParser(); var
newSaleCommand = "Command:NewSale"; commandParser.Parse(newSaleCommand); } }
```

I explicitly define the role SaleEventListener using an interface, ISaleEventListener. Why am I using a listener interface instead of the built-in support C# provides for events through delegates?

There are a number of reasons why I chose to use a listener interface over the events and delegates. The listener interface explicitly identifies the role the command parser collaborates with. Note that I'm not coupling the CommandParser to any particular class, but by using a listener interface, I am stating explicitly the relationship between the CommandParser and the role it collaborates with. Events and delegates may allow you to hook in arbitrary code to the CommandParser, but it doesn't express the possible relationships between the objects in the domain. Using a listener interface in this case allows me to use the communication patterns between the objects to clearly express the domain model.

In my example, the CommandParser will parse commands and send different types of events in terms of the application domain. These will be always hooked up at the same time. In this case it is much more convenient to pass a reference to an instance of a listener interface that can process a set of events rather than attaching different delegates to each different event. I then need an instance of this interface so the CommandParser can interact with it. I use the NMock framework to dynamically create an instance of this interface.

I now need to specify what I expect CommandParser object to tell the object playing the role of ISaleEventListener when it has interpreted a command from the input device. I specify this by writing an expectation on the mock implementation of ISaleEventListener:

```
[Test] public void NotifiesListenerOfNewSaleEvent() { var saleEventListener =
mockery.NewMock<ISaleEventListener>(); var commandParser = new CommandParser(); var
newSaleCommand = "Command:NewSale";
Expect.Once.On(saleEventListener).Method("NewSaleInitiated");
Expect.Once.On(saleEventListener).Method("NewSaleInitiated");
commandParser.Parse(newSaleCommand); mockery.VerifyAllExpectationsHaveBeenMet(); }
```

The act of writing this expectation drove out the interface the CommandParser requires from its collaborator. Using a mock object, you can discover and design the interfaces an object requires from its collaborators before any implementations of these collaborators even exist. This allows you to stay

focused on the CommandParser without worrying about the implementations of its collaborators.

To get this test to compile, I need to create the CommandParser class and the ISaleEventListener interface:

```
public class CommandParser { public void Parse(string messageFromDevice) { } } public
interface ISaleEventListener { void NewSaleInitiated(); }
```

The test compiles, I run the test, and I get the following failure:

```
TestCase 'Domain.Tests.CommandParserTests.NotifiesListenerOfNewSaleEvent' failed:
NMock2.Internal.ExpectationException : not all expected invocations were performed
Expected: 1 time: saleEventListener.NewSaleInitiated(any arguments) [called 0 times]
at NMock2.Mockery.FailUnmetExpectations() at
NMock2.Mockery.VerifyAllExpectationsHaveBeenMet()
```

The NMock framework reports that the mock implementation of ISaleEventListener was expecting the method NewSaleInitiated to be invoked once, but this never happens. To get the test to pass, I need to pass the mock instance of the saleEventListener to the CommandParser object as a dependency.

```
[Test] public void NotifiesListenerOfNewSaleEvent() { var saleEventListener =
mockery.NewMock<ISaleEventListener>(); var commandParser = new
CommandParser(saleEventListener); var newSaleCommand = "Command:NewSale";
Expect.Once.On(saleEventListener).Method("NewSaleInitiated");
ommandParser.Parse(newSaleCommand); mockery.VerifyAllExpectationsHaveBeenMet(); }
```

The test now explicitly specifies the dependency the CommandParser has on its environment, which specifies what message (method call) the saleEventListener should receive.

Here's the simplest implementation that passes this test:

```
public class CommandParser { private readonly ISaleEventListener saleEventListener;
public CommandParser(ISaleEventListener saleEventListener) { this.saleEventListener =
saleEventListener; } public void Parse(string messageFromDevice) {
saleEventListener.NewSaleInitiated(); } }
```

Finishing the Sale

Now that the test is passing, I can move on to the next test. The next simple success scenario will be to test that the CommandParser can decode the finish sale command and notify the system.

```
[Test] public void NotifiesListenerOfSaleCompletedEvent() { var saleEventListener =
mockery.NewMock<ISaleEventListener>(); var commandParser = new
CommandParser(saleEventListener); var endSaleCommand = "Command:EndSale";
Expect.Once.On(saleEventListener).Method("SaleCompleted");
commandParser.Parse(endSaleCommand); mockery.VerifyAllExpectationsHaveBeenMet(); }
```

The test drives out the need for another method the ISaleEventListener interface must support.

```
public interface ISaleEventListener { void NewSaleInitiated(); void SaleCompleted();
}
```

Running the test fails, as you would expect. NMock displays the following error message:

```
TestCase 'Domain.Tests.CommandParserTests.NotifiesListenerOfSaleCompletedEvent'
```

```
failed: NMock2.Internal.ExpectationException : unexpected invocation of
saleEventListener.NewSaleInitiated() Expected: 1 time:
saleEventListener.SaleCompleted(any arguments) [called 0 times]
```

I need to interpret the raw command and call the appropriate method on instance of the saleEventListener object. The simple implementation in **Figure 2** should get the test to pass.

Figure 2 Implementing saleEventListener

```
public class CommandParser { private const string END_SALE_COMMAND = "EndSale";
private readonly ISaleEventListener saleEventListener; public
CommandParser(ISaleEventListener saleEventListener) { this.saleEventListener =
saleEventListener; } public void Parse(string messageFromDevice) { var commandName =
messageFromDevice.Split(':')[1].Trim(); if (END_SALE_COMMAND.Equals(commandName))
saleEventListener.SaleCompleted(); else saleEventListener.NewSaleInitiated(); } }
```

Before moving on to the next test, I remove duplication in the test code (see **Figure 3**).

Figure 3 Updating the Tests

```
[TestFixture] public class CommandParserTests { private Mockery mockery; private
CommandParser commandParser; private ISaleEventListener saleEventListener; [SetUp]
public void BeforeTest() { mockery = new Mockery(); saleEventListener =
mockery.NewMock<ISaleEventListener>(); commandParser = new
CommandParser(saleEventListener); mockery = new Mockery(); } [TearDown] public void
AfterTest() { mockery.VerifyAllExpectationsHaveBeenMet(); } [Test] public void
NotifiesListenerOfNewSaleEvent() { var newSaleCommand = "Command:NewSale";
Expect.Once.On(saleEventListener).Method("NewSaleInitiated");
commandParser.Parse(newSaleCommand); } [Test] public void
NotifiesListenerOfSaleCompletedEvent() { var endSaleCommand = "Command:EndSale";
Expect.Once.On(saleEventListener).Method("SaleCompleted");
commandParser.Parse(endSaleCommand); } }
```

Next, I want to ensure the CommandParser can process an input command with barcode information. The application receives a raw message in the following format:

```
Input:Barcode=100008888559, Quantity =1
```

I want to tell the object that plays the role of a SaleEventListener that an item with a barcode and quantity was entered:

```
[Test] public void NotifiesListenerOfItemAndQuantityEntered() { var message = "Input:
Barcode=100008888559, Quantity =1";
Expect.Once.On(saleEventListener).Method("ItemEntered") .With("100008888559", 1);
commandParser.Parse(message); }
```

The test drives out the need for yet another method the ISaleEventListener interface must provide:

```
public interface ISaleEventListener { void NewSaleInitiated(); void SaleCompleted();
void ItemEntered(string barcode, int quantity); }
```

Running the test produces this failure:

```
TestCase 'Domain.Tests.CommandParserTests.NotifiesListenerOfItemAndQuantityEntered'
failed: NMock2.Internal.ExpectationException : unexpected invocation of
saleEventListener.NewSaleInitiated() Expected: 1 time:
saleEventListener.ItemEntered(equal to "100008888559", equal to <1>) [called 0 times]
```

The failure message tells me that the wrong method was called on the saleEventListener. This is expected as I haven't implemented any logic in CommandParser to handle input messages containing barcode and quantity. **Figure 4** shows the updated CommandParser.

Figure 4 Handling Input Messages

```
public class CommandParser { private const string END_SALE_COMMAND = "EndSale";
private readonly ISaleEventListener saleEventListener; private const string INPUT =
"Input"; private const string START_SALE_COMMAND = "NewSale"; public
CommandParser(ISaleEventListener saleEventListener) { this.saleEventListener =
saleEventListener; } public void Parse(string messageFromDevice) { var command =
messageFromDevice.Split(':'); var commandType = command[0].Trim(); var commandBody =
command[1].Trim(); if(INPUT.Equals(commandType)) { ProcessInputCommand(commandBody);
} else { ProcessCommand(commandBody); } } private void ProcessCommand(string
commandBody) { if (END_SALE_COMMAND.Equals(commandBody))
saleEventListener.SaleCompleted(); else if (START_SALE_COMMAND.Equals(commandBody))
saleEventListener.NewSaleInitiated(); } private void ProcessInputCommand(string
commandBody) { var arguments = new Dictionary<string, string>(); var commandArgs =
commandBody.Split(','); foreach(var argument in commandArgs) { var argNameValues =
argument.Split('='); arguments.Add(argNameValues[0].Trim(),
argNameValues[1].Trim()); } saleEventListener.ItemEntered(arguments["Barcode"],
int.Parse(arguments["Quantity"])); } }
```

Abstracting Conversations

Before moving on to the next test, I need to refactor and tidy up the interactions between CommandParser and saleEventListener. I want to specify interactions between an object and its collaborators in terms of the application domain. The ItemEntered message takes into two arguments a string representing a barcode and an integer representing quantity. What do these two arguments really represent in the domain of the application?

Rule of thumb: If you are passing around primitive data types among object collaborators, it may be an indication you are not communicating at the right level of abstraction. You need to see whether the primitive data types represent concepts in your domain that you may have missed.

In this case, the barcode is decomposed into manufactureCode and itemCode, which represents an item identifier. I can introduce the concept of an item identifier in the code. This should be an immutable type that can be constructed from a barcode, and I can give the ItemIdentifier the responsibility of decomposing the barcode into a manufacturer code and an item code. Similarly, quantity should be a value object as it represents a measurement—for example, the quantity of an item could be measured by weight.

For now, I don't have a need yet to decompose the barcode or handle different types of measurements for quantity. I will simply introduce these value objects to ensure that communication between objects remains in domain terms. I refactor the code to include the concept of item identifier and quantity in the test.

```
[Test] public void NotifiesListenerOfItemAndQuantityEntered() { var message = "Input:
Barcode=100008888559, Quantity=1"; var expectedItemid = new ItemId("100008888559");
var expectedQuantity = new Quantity(1);
Expect.Once.On(saleEventListener).Method("ItemEntered").With( expectedItemid,
expectedQuantity); commandParser.Parse(message); }
```

Neither ItemId nor Quantity exist yet. To get the test to pass, I need to create these new classes and modify the code to reflect these new concepts. I implement these types as value objects. The identities of these objects are based on the values they hold (see **Figure 5**).

Figure 5 ItemID and Quantity

```
public interface ISaleEventListener { void SaleCompleted(); void NewSaleInitiated();
void ItemEntered(ItemId itemId, Quantity quantity); } public class ItemId { private
readonly string barcode; public ItemId(string barcode) { this.barcode = barcode; }
public override bool Equals(object obj) { var other = obj as ItemId; if(other ==
null) return false; return this.barcode == other.barcode; } public override int
GetHashCode() { return barcode.GetHashCode(); } public override string ToString() {
return barcode; } } public class Quantity { private readonly int value; public
Quantity(int qty) { this.value = qty; } public override string ToString() { return
value.ToString(); } public override bool Equals(object obj) { var otherQty = obj as
Quantity; if(otherQty == null) return false; return value == otherQty.value; }
public override int GetHashCode() { return value.GetHashCode(); } }
```

With interaction-based tests using mock objects, you can streamline the interactions between an object and its collaborators by using tests to explicitly describe communication protocols between objects at a high-level of abstraction in terms of the domain. Since mocks allow you do this without having any concrete implementations of the collaborators to exist, you can try out alternate collaboration patterns until you have designed the collaborations between objects in terms of the application domain. Also by examining and closely following the interactions between object and its collaborator, it also helps dig out any domain concepts you may have overlooked.

Calculating Receipts

Now that I have an object to decode commands from input devices as point-of-sale events, I need an object that can respond and process these events. The requirement is to print out receipts, so I need an object that can calculate receipts. To fill these responsibilities, I look for an object that can play the role of a SaleEventListener. The concept of a Register comes to mind and seems to fit the role of a SaleEventListener, so I create a new class Register. Since this class responds to sale events, I make it implement ISaleEventListener:

```
public class Register : ISaleEventListener { public void SaleCompleted() { } public
void NewSaleInitiated() { } public void ItemEntered(ItemId itemId, Quantity quantity)
{ } }
```

One of the main responsibilities of the Register is to calculate receipts and send them to a printer. I will rig some events on the object by invoking its methods. I start with the simple scenario. Calculating the receipt for a sale with no items should have a total of 0. I need to ask the question: Who would know if the register has calculated the total for the items correctly? My first guess is a receipt printer. I express this in code by writing a test:

```
[Test] public void ReceiptTotalForASaleWithNoItemsShouldBeZero() { var receiptPrinter
= mockery.NewMock<IReceiptPrinter>(); var register = new Register();
register.NewSaleInitiated();
Expect.Once.On(receiptPrinter).Method("PrintTotalDue").With(0.00);
register.SaleCompleted(); }
```

The test implies that the Register object tells the receipt printer to print the total due.

Before moving forward let's take a step back and examine the communication protocol between the Register object and the receipt printer. Is the PrintTotalDue method meaningful to the Register object? Looking at this interaction it's clearly not the Register's responsibility to be concerned with printing receipts. The Register object is concerned with calculating the receipt and sending it to an object that receives receipts. I will choose a name for the method that describes just that behavior: ReceiveTotalDue. This is much more meaningful to the Register object. In doing this I discovered that the collaborating role the Register requires is a ReceiptReceiver rather than a ReceiptPrinter. Finding the appropriate name for a role is an important part of the design activity, as it helps you design objects that have a cohesive set of responsibilities. I rewrite the test to reflect the new name for the role.

```
[Test] public void ReceiptTotalForASaleWithNoItemsShouldBeZero() { var
receiptReceiver = mockery.NewMock<IReceiptReceiver>(); var register = new Register();
register.NewSaleInitiated();
Expect.Once.On(receiptReceiver).Method("ReceiveTotalDue").With(0.00);
register.SaleCompleted(); }
```

To get this to compile, I create an IReceiptReceiver interface to represent the role ReceiptReceiver.

```
public interface IReceiptReceiver { void ReceiveTotalDue(decimal amount); }
```

When I run the test I get a failure, as expected. The mock framework tells me the method call ReceiveTotalDue was never made. To make the test pass I need to pass a mock implementation of IReceiptReceiver to the Register object, so I change the test to reflect this dependency.

```
[Test] public void ReceiptTotalForASaleWithNoItemsShouldBeZero() { var
receiptReceiver = mockery.NewMock<IReceiptReceiver>(); var register = new
Register(receiptReceiver); register.NewSaleInitiated();
Expect.Once.On(receiptReceiver).Method("ReceiveTotalDue").With(0.00m);
register.SaleCompleted(); }
```

The simple implementation here should get the test to pass:

```
public class Register : ISaleEventListener { private readonly IReceiptReceiver
receiptReceiver; public Register(IReceiptReceiver receiver) { this.receiptReceiver =
receiver; } public void SaleCompleted() { receiptReceiver.ReceiveTotalDue(0.00m); }
public void NewSaleInitiated() { } public void ItemEntered(ItemId itemId, Quantity
quantity) { } }
```

The primitive type decimal used to represent the total amount due is just a scalar, which has no meaning in the domain. What this really represents is monetary values, so I will create an immutable value object to represent money. At this point, there is no need for multi-currency or rounding of monetary values. I simply create a Money class that wraps the decimal value. When the need arises, I can add currency and rounding rules in this class. For now, I will stay focused on the current task and modify the code to reflect this. The implementation is shown in **Figure 6**.

Figure 6 Using Money

```
public interface IReceiptReceiver { void ReceiveTotalDue(Money amount); } public
class Register : ISaleEventListener { private readonly IReceiptReceiver
receiptReceiver; public Register(IReceiptReceiver receiver) { this.receiptReceiver =
receiver; } public void SaleCompleted() { receiptReceiver.ReceiveTotalDue(new
Money(0.00m)); } public void NewSaleInitiated() { } public void ItemEntered(ItemId
itemId, Quantity quantity) { } }
```

```
[Test] public void ReceiptTotalForASaleWithNoItemsShouldBeZero() { var
receiptReceiver = mockery.NewMock<IReceiptReceiver>(); var register = new
Register(receiptReceiver); register.NewSaleInitiated(); var totalDue = new Money(0m);
Expect.Once.On(receiptReceiver).Method("ReceiveTotalDue") .With(totalDue);
register.SaleCompleted(); }
```

The next test will flesh out some additional behavior on the Register object. The Register should not calculate receipts if new sales are not initiated, so I write a test to specify this behavior:

```
[SetUp] public void BeforeTest() { mockery = new Mockery(); receiptReceiver =
mockery.NewMock<IReceiptReceiver>(); register = new Register(this.receiptReceiver); }
[Test] public void ShouldNotCalculateRecieptWhenThereIsNoSale() {
Expect.Never.On(receiptReceiver); register.SaleCompleted(); }
```

This test explicitly specifies that the receiptReceiver should never receive any method calls on it. The test fails as expected with the following error:

```
TestCase 'Domain.Tests.RegisterTests. ShouldNotCalculateRecieptWhenThereIsNoSale'
failed: NMock2.Internal.ExpectationException : unexpected invocation of
receiptReceiver.ReceiveTotalDue(<0.00>)
```

To get this test to pass, the Register object has to keep track of some state—to track whether there is a sale in progress. I can make the test pass with the implementation shown in **Figure 7**.

Figure 7 Keeping Track of State with Register

```
public class Register : ISaleEventListener { private readonly IReceiptReceiver
receiptReceiver; private bool hasASaleInprogress; public Register(IReceiptReceiver
receiver) { this.receiptReceiver = receiver; } public void SaleCompleted() {
if(hasASaleInprogress) receiptReceiver.ReceiveTotalDue(new Money(0.00m)); } public
void NewSaleInitiated() { hasASaleInprogress = true; } public void
ItemEntered(ItemId itemId, Quantity quantity) { } }
```

Getting the Product Description

The product information systems are located in the head office, which is exposed as a RESTful service. The Register object will need to retrieve product information from this system to work out the receipt for a sale. I don't want to be constrained by the implementation details of this external system, so I will define my own interface in domain terms for the services the point of sale system needs.

I will write a test to calculate the receipt for a sale with a couple of items. To work out the total for a sale, the register needs to collaborate with another object. To retrieve [a] product description for an item, I introduce the role of a product catalog. This could be a RESTful service, a database, or some other system. The implementation detail is not important nor of any concern to the Register object. I want to design an interface that is meaningful to the Register object. The test is shown in **Figure 8**.

Figure 8 Testing Register

```
[TestFixture] public class RegisterTests { private Mockery mockery; private
IReceiptReceiver receiptReceiver; private Register register; private readonly ItemId
itemId_1 = new ItemId("000000001"); private readonly ItemId itemId_2 = new
ItemId("000000002"); private readonly ProductDescription descriptionForItemWithId1 =
new ProductDescription("description 1", new Money(3.00m)); private readonly
ProductDescription descriptionForItemWithId2 = new ProductDescription("description
2", new Money(7.00m)); private readonly Quantity single_item = new Quantity(1);
private IProductCatalog productCatalog; [SetUp] public void BeforeTest() { mockery =
new Mockery(); receiptReceiver = mockery.NewMock<IReceiptReceiver>(); productCatalog
= mockery.NewMock<IProductCatalog>(); register = new Register(receiptReceiver,
productCatalog); Stub.On(productCatalog).Method("ProductDescriptionFor")
.With(itemId_1) .Will(Return.Value(descriptionForItemWithId1));

Stub.On(productCatalog).Method("ProductDescriptionFor") .With(itemId_2)
.Will(Return.Value(descriptionForItemWithId2)); } [TearDown] public void AfterTest()
{ mockery.VerifyAllExpectationsHaveBeenMet(); }     [Test] public void
{ mockery.VerifyAllExpectationsHaveBeenMet(); } ... [Test] public void
ShouldCalculateRecieptForSaleWithMultipleItemsOfSingleQuantity() {
register.NewSaleInitiated(); register.ItemEntered(itemId_1, single_item);
register.ItemEntered(itemId_2, single_item);
Expect.Once.On(receiptReceiver).Method("ReceiveTotalDue") .With(new Money(10.00m));
register.SaleCompleted(); } }
```

This test designed the interface for the productCatalog that is required by the Register object. I also discovered the need for a new type, productDescription, that represents the description of a product. I will model this as a value object (immutable type). I stub the productCatalog to give a productDescription when queried with an ItemIdentifier. I stub the invocation of a ProductDescriptionFor method on productCatalog because this is a query method that returns the productDescription. The Register acts on the result of the query returned. What is important here is that the Register produces the correct side effects when the ProductCatalog returns a specified ProductDescription. The rest of the test verifies that the correct total is calculated correctly and is sent to the receipt receiver.

I run the test to get the expected failure:

```
unexpected invocation of receiptReceiver.ReceiveTotalDue(<0.00>) Expected: Stub:
productCatalog.ProductDescriptionFor(equal to <000000001>), will return
<Domain.Tests.ProductDescription> [called 0 times] Stub:
productCatalog.ProductDescriptionFor(equal to <000000002>), will return
<Domain.Tests.ProductDescription> [called 0 times] 1 time:
receiptReceiver.ReceiveTotalDue(equal to <10.00>) [called 0 times]
```

The mock framework tells me that receiptReceiver should have received a total of 10, but a 0 was received. This is expected since we haven't implemented anything that can calculate the total. **Figure 9** shows a first attempt at the implementation to get the test to pass.

Figure 9 Calculating the Total

```
public class Register : ISaleEventListener { private readonly IReceiptReceiver
receiptReceiver; private readonly IProductCatalog productCatalog; private bool
hasASaleInprogress; private List<ProductDescription> purchasedProducts = new
List<ProductDescription>(); public Register(IReceiptReceiver receiver,
IProductCatalog productCatalog) { this.receiptReceiver = receiver;
this.productCatalog = productCatalog; } public void SaleCompleted() {
if(hasASaleInprogress) { Money total = new Money(0m); purchasedProducts.ForEach(item
=> total += item.UnitPrice); receiptReceiver.ReceiveTotalDue(total); } } public void
NewSaleInitiated() { hasASaleInprogress = true; } public void ItemEntered(ItemId
itemId, Quantity quantity) { var productDescription =
productCatalog.ProductDescriptionFor(itemId);
purchasedProducts.Add(productDescription); } }
```

This code fails to compile since the Money class doesn't define an operation to add money. I now have a need for Money objects to perform addition, so I write a quick test to for the Money class to handle this immediate requirement.

```
[TestFixture] public class MoneyTest { [Test] public void
ShouldBeAbleToCreateTheSumOfTwoAmounts() { var twoDollars = new Money(2.00m); var

threeDollars = new Money(3m); var fiveDollars = new Money(5m); Assert.That(twoDollars
+ threeDollars, Is.EqualTo(fiveDollars)); } }
```

The implementation to get this test passing is shown in **Figure 10**.

Figure 10 Updates to Add Money

```
public class Money { private readonly decimal amount; public Money(decimal value) {
this.amount = value; } public static Money operator +(Money money1, Money money2) {
return new Money(money1.amount + money2.amount); } public override string ToString()
{ return amount.ToString(); } public override bool Equals(object obj) { var
otherAmount = obj as Money; if(otherAmount == null) return false; return amount ==
otherAmount.amount; } public override int GetHashCode() { return
amount.GetHashCode(); } }
```

Refactoring

Before going any further, I need to clarify the intent of my code. My tests don't track any of the Register object's internal details. All of this is hidden inside the Register object, which allows me to refactor and clarify the intent of the code.

The Register object is managing state related to the current sale, but I don't have an object that represents the concept of a sale. I decided to extract a Sale class to manage all state and behavior related to a sale. The refactored code is shown in **Figure 11**.

Figure 11 Adding a Sale Class

```
public class Register : ISaleEventListener { private readonly IReceiptReceiver
receiptReceiver; private readonly IProductCatalog productCatalog; private Sale sale;
public Register(IReceiptReceiver receiver, IProductCatalog productCatalog) {
this.receiptReceiver = receiver; this.productCatalog = productCatalog; } public void
SaleCompleted() { if(sale != null) { sale.SendReceiptTo(receiptReceiver); } } public
void NewSaleInitiated() { sale = new Sale(); } public void ItemEntered(ItemId
itemId, Quantity quantity) { var productDescription =
productCatalog.ProductDescriptionFor(itemId);
sale.PurchaseItemWith(productDescription); } } public class Sale { private readonly
List<ProductDescription> itemPurchased = new List<ProductDescription>(); public void
SendReceiptTo(IReceiptReceiver receiptReceiver) { var total = new Money(0m);
itemPurchased.ForEach(item => total += item.UnitPrice);
receiptReceiver.ReceiveTotalDue(total); } public void
PurchaseItemWith(ProductDescription description) { itemPurchased.Add(description); }
}
```

Value Objects

In .NET terminology, value types refer to the primitive types supported by the CLR such as int, bool, structs, enum, etc. This should not be confused with value objects; these are objects that describe things. The important thing to note is that value objects can be implemented by classes (reference types); these objects are immutable and have no conceptual identity. Since Value objects have no individual identity, two values objects are considered equal if they have exactly the same state.

Wrapping Up

This example showed you how mock objects and TDD can guide the design of object-oriented programs. There are a number of important side benefits of this iterative discovery process. Using mock objects not only helped me discover the collaborators and object needs, but I was also able to describe and make explicit in my tests the roles an object's environment must support and communication patterns between the object under test and its collaborators.

Note that the tests show only what the object does in terms of domain concepts. The tests don't specify how the Register interacts with a Sale. These are internal details regarding how the Register object is structured to accomplish its work, and these implementation details remain hidden. I chose mocks only to explicitly specify interactions that should be visible externally between an object and its collaborators in the system.

The tests using mock objects lay out a set of constraints specifying what messages objects can send and when those messages should be sent. The interactions between objects are clearly described in terms of domain concepts. The domain consists of narrow role interfaces that make the system pluggable, and the behavior of the system can be altered easily by changing the composition of its objects.

None of the objects expose their internal state or structure, and only immutable states (value objects like money and itemId) are passed around. This makes programs easier to maintain and modify. One thing to note is that you normally would have started the development with a failing acceptance, which would have driven the requirements and the need for the CommandParser in my example.

Isaiah Perumalla is a senior developer at ThoughtWorks, where he is involved in delivering enterprise-scale software, using object-oriented design and Test-Driven Development.

Build Stunning
ASP.NET
Apps that Run on Desktops and Mobile Devices
Free Download
INFRAGISTICS
DESIGN / DEVELOP / EXPERIENCE