A Simple Unit Testing Framework for CA-Visual Objects 2.5

Mark Langley

CA-World 2000
eBusiness Solutions in Internet Time
JP195SN

Introduction

> ***Caution: The information presented here could change the way you approach program development!***

Software developers know that they should write unit tests for their code. While there are a few developers who include unit testing as a part of the development process, most do not. The answer to "Why not?" is something like, "It takes too much time", or "It's too difficult", or "It's not worth the extra effort". One reason for the typical response is related to the manner in which most software testing is performed.

This session will present an easy-to-use unit testing framework for CA-Visual Objects, and it will demonstrate that unit testing does not have to be a time consuming effort with -minimal payback.

Typical Testing Scenario

The reasons raised earlier in answer to the "Why not?" question are legitimate for most developers. It is time consuming, it is difficult, and the payback on effort is limited because of the way many programmers go about testing their code. Most developers often limit their testing to a "manual method", either by setting breakpoints and stepping through the code in a debugging session or by manipulating the user interface and "hand checking" the results. While these techniques do help identify and resolve some problems, they should not be considered the primary means of conducting unit tests. There are several reasons for this.

One drawback is that testing via a "manual method" is not likely to be repeated consistently from one testing session to the next. By its nature, this type of testing process is too dependent on developer interaction. It is unlikely that the exact same testing steps will be performed each time. Some steps might be omitted, and new steps might be added at the discretion of the person doing the testing. The reality is that this "manual method" of testing will not cover all previous scenarios because it takes too much time. Thus it is harder to ensure that all parts, which were previously tested, still test successfully. You guess and hope. Guessing and hoping do not do much to increase your confidence.

Also, “manual” tests have a temporary existence. This ensures that the tests cannot be shared between members of the development team, nor can they be archived and easily run at a later time. Because it is so time consuming to run “manual” tests, they are not run as often as they should be. When delivery schedule pressure increases, manual testing is often minimized or ignored.

There Is a Better Way

Fortunately, there is a better way to test your code. The unit testing framework (VO Unit) presented here for CA-Visual Objects 2.5 is an adaptation of an open source unit testing framework originally developed for the Smalltalk community by Kent Beck and Eric Gamma. It has also been implemented in development environments like Java, C++, and a few others.

While the VO Unit testing framework does not guarantee that your code will be error free, it will benefit you in the following ways.

The Benefits of a Unit Testing Framework

The word *framework* generally means a basic structure. The first benefit that VO Unit provides is a basic structure for conducting unit tests. Haphazard unit testing should become a thing of the past. Instead, more tests will be performed consistently. More tests that are easily and consistently conducted will lead to software with fewer errors.

Another benefit is that VO Unit will help focus your development efforts. Before a unit test can run successfully, it must be able to exercise the class properties and methods. So the class being tested must provide access to those properties and methods in some way. Tests put focus on the interface to the class rather than on the implementation. The test itself becomes another user of the class. Also, tests put the focus on error conditions that might cause the class to fail. This makes the design of the class more robust. The by-product of testing is that your development efforts are focused on smaller bits of code with nearly immediate feedback about the success of recent code changes. This increases productivity, because most problems are likely to be corrected earlier in the development cycle.

Third, the use of VO Unit will reduce the time spent debugging. VO Unit provides nearly immediate feedback about the result of the test, so it's easier to isolate the problem while testing than to trace a problem using the debugger.

Fourth, it will increase the amount of confidence you have in what you are developing. Unit testing will allow you to gain a better understanding of the manner in which the class works. You can develop various test conditions that exercise the code thoroughly. These tests will attempt to make the software fail in many different ways. In the process you will develop software which is more robust than if your efforts were primarily focused on code development alone.

The fifth benefit occurs when you have to change existing code to meet new feature requests. The framework allows you to confidently change existing classes because you are less likely to break something without knowing it. If tests have run successfully before a change was made, and they continue to be successful after modifying the code, you can be confident that the changes have not introduced errors.

Sixth, it provides a tangible means of measuring your progress. With VO Unit, you can know specifically which classes have been tested and which have not. For those classes that have, it is known exactly which classes failed the tests. Without the testing framework this type of knowledge is unavailable. More importantly, with the VO Unit testing framework you know when you are finished because all of the tests for the important classes and methods run successfully. Without it you guess and hope.

Seventh, your code is self-testing. Your tests can be run automatically and unattended, letting tests check their own results so you don't have to. This makes tests easy to run and results easy to analyze. If they are easy to run and analyze they will be run more frequently. You can run tests while you are in a meeting or when you are away from the office. When you return the results will be waiting for you. This makes testing very convenient.

Finally, use of the framework will break the following cycle: You feel pressure to get the code working as quickly as possible. You rationalize that you don't have time to do systematic testing so you conduct fewer tests. Your focus is primarily on code development. You don't really know if it operates exactly as you intended, so you may have some problems and you just don't know it yet. The fewer tests you conduct, the less productive you become, and the less stable your code becomes. The less productive and accurate you are, the more pressure you feel. Then the cycle repeats itself, until something is done to change it. VO Unit can help you change.

After a brief discussion of unit testing philosophy, you will learn how to:

- Include the VO Unit testing framework in a CA-Visual Objects application
- Subclass the TestCase class
- Set up a test fixture
- Write a test
- Add tests to a suite of tests
- Run the test suites

Unit Testing Defined

So there is no confusion, let's talk about what unit testing is and is not. In this context, unit testing can be expressed in this equation:

one unit test = one method in one class

You can see that the focus of the test is local to a single method in one class. This is unlike functional testing, which examines the software at a much higher level and with a significantly different purpose.

So let's describe what unit testing is not. It is not manually operated. It is not some form of automated screen-driver test that simulates user input. It is not highly user interactive. It is not tightly coupled. In other words, unit tests run with minimal dependencies on objects other than the one being tested.

Unit tests automatically report their results in the form of failures, errors, or the absence of either failures or errors. When tests run, they check for expected results. As will be discussed later, this checking is simply an evaluation of a Boolean condition. Failures are anticipated problems, and occur when the expected result of a test evaluates to false. An error is defined as an unexpected exception. These are standard CA-Visual Objects errors.

When to Write Tests

Consider writing tests when you are adding functionality to untested code. When fixing a bug, write unit tests that expose the bug. Then change the production code to fix the bug. This will ensure that a similar bug doesn't get past the unit tests again. Write tests when refactoring code.

Refactoring is all about making code easier to change. However, change involves risk. Having unit tests gives you the security you need to change the program later. Testing will eliminate the hesitation to modify and improve existing software, because it will ensure that any changes to the existing code base do not introduce errors.

Most of all, remember that the main purpose of writing tests is to help get programs working and keep them working, nothing more. Make a reasonable judgement about which methods should be tested. Don't try to test everything. Access and assign methods are not very likely to fail, so don't waste time developing tests for them. Instead concentrate on where the greatest risk of failure is. Look at the code and see where it becomes complex, and write tests for the complex part. When the entire test results for the key methods are successful, you're done.

A Generic Testing Strategy

Consider using the following testing/coding strategy. This may seem strange at first, but it has some benefits.

Code the unit test first. Start by writing a test for the class before the class is even developed. Ask yourself, "What are the things that need to exist before the test can run?" Think about what the class is supposed to do. What are the properties and methods the class needs? What are some conditions that might cause functionality in the class to fail? Look for boundary conditions that can cause the test to fail. In order for the test to exercise the class, what are the test preconditions (test fixtures) that must exist?

Next, run the test before the class method is developed to verify failure. This will prove that the test does run and that it is testing what it should. Sometimes you may be surprised when a test that is expected to fail succeeds instead. Then you must investigate why. In the process of this investigation, you will increase your understanding. The more you understand how your code operates, the better your development will be.

Then code just enough to get the test running successfully. Do not try to do too much at one time. Think and work in small increments of development. Take the simplest increment of functionality and write a unit test for it. Instead of developing a full-blown class, ask, "What is the smallest increment of functionality that could be implemented?"

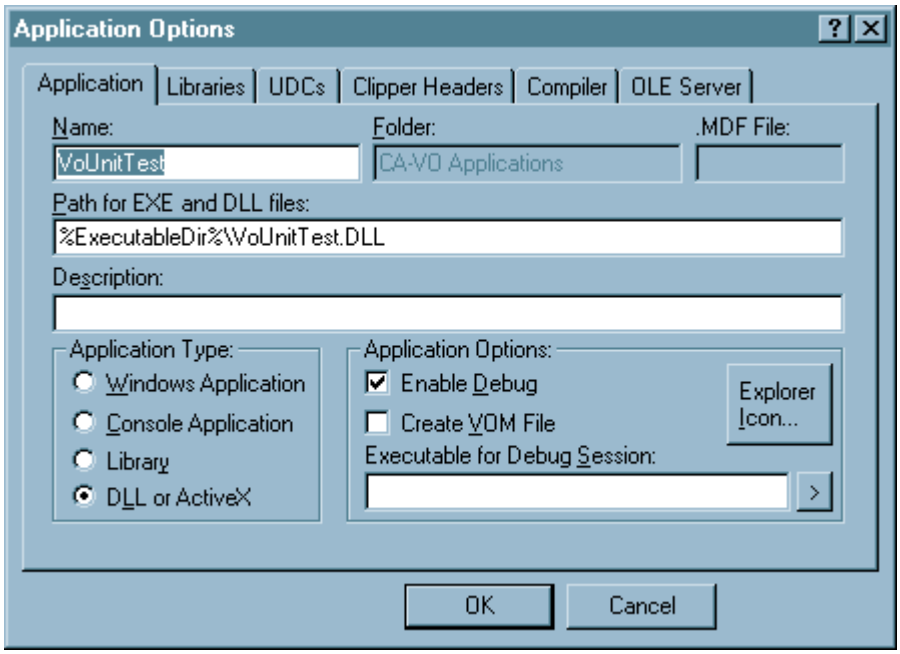
Then run the test again to verify success. Alternate coding and testing until the unit test is successful. After the test runs successfully, then work on adding functionality to the class.

Repeat the process until the class implements all of the functionality you desire. And when all tests run successfully, you're done.

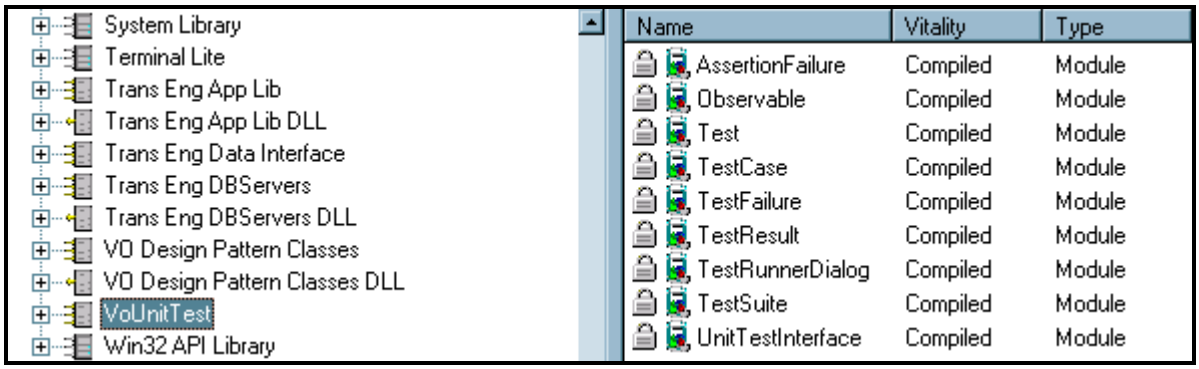
Now let's see how the VO Unit framework is designed and how to use it in a CA-Visual Objects application.

Understanding and Using the Unit Testing Framework

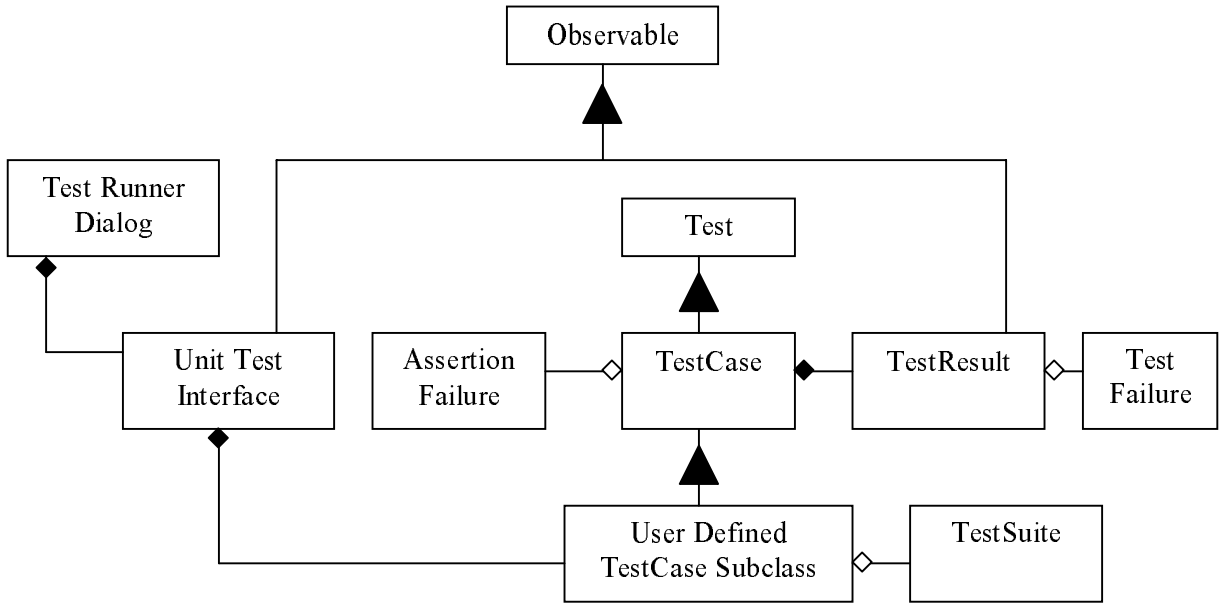
Start by compiling and linking the VoUnitTest AEF to create the VoUnitTest.DLL. This DLL will provide the core support for the unit tests.



The VO Unit testing framework contains nine base classes plus one or more user-defined TestCase subclasses.



A class diagram is illustrated below. It shows how the classes are related to each other.



Now let's take a closer look at the classes which comprise the framework.

Assertion Failure Class

Properties	Methods	Comments
oError	Init	

Observable Class

Properties	Methods	Comments
aObservers	AddObserver	Adds client objects to a list to be notified of changes
bHasChanged	Axit	
	ClearChanged	Resets the state
	CountObservers	
	DeleteObservers	Empties the client object list
	DeleteObserver	Removes a specific client from the list
	HasChanged	
	Init	
	NotifyObservers	Calls the Update method for each client in the list
	SetChanged	

Test Class (Abstract)

Properties	Methods	Comments
	CountTestCases	All implementation is deferred to subclass
	Init	
	Run	

TestCase Class

Properties	Methods	Comments
sName	Assert	Key method, where test is evaluated
oResult	Axit	
	CountTestCases	Number of test cases in the class
	CreateResult	Builds a TestResult object
	DefaultResult	
	DefaultRun	
	DynamicallyAddTestsToSuite	Automatically builds test case list
	Fail	Builds an assertion failure object
	GetResult	
	Init	
	IsTestMethod	Determines if method name begins with "Test"
	Run	
	RunTest	Invokes the method for the test
	SelfFactory	Creates a TestCase object

TestFailure Class

Properties	Methods	Comments
oFailedTest	Init	Collection of failed test information
oException		Collection of error information

TestResult Class

Properties	Methods	Comments
aErrors	AddError	Adds error object to the error collection
aFailures	AddFailure	Adds failure object to the failure collection
iTestsRun	EndTest	Sets state
bStop	ErrorCount	Returns count of errors
	FailureCount	Returns count of failures
	Init	Observes the UnitTestInterface class
	Run	Calls RunTest method of TestCase class
	SetState	Triggers NotifyObservers method
	ShouldStop	
	StartTest	Increments test counter and sets new state
	Stop	
	TestErrors	Returns aErrors
	TestFailures	Returns aFailures
	TestsRun	Current count of tests run, queried by UnitTestInterface
	Update	Queries state of the UnitTestInterface class
	WasSuccessful	Boolean indicator, queried by UnitTestInterface

TestRunnerDialog GUI Class

Properties	Methods	Comments
StandardGUI	BuildListViewColumns	
oUTI	BuildListViewItems	
	CancelButton	
	DeleteListViewItems	
	Init	
	ListBoxSelect	
	PostInit	Observes the UnitTestInterface class
	RefreshListView	
	RefreshUI	Updates the controls in the dialog
	Reset	Clears the control values in the dialog
	RunButton	Sends class name to oUTI:RunTestsForClass
	StopButton	Sends HaltTesting message to oUTI
	Update	Queries Observable object for state information

TestSuite Class

Properties	Methods	Comments
sName	AddTest	Adds test to collection
aTests	Axit	
	CountTestCases	Queried by UnitTestInterface
	Init	
	Run	Runs test for each TestCase in aTests

UnitTestInterface Class

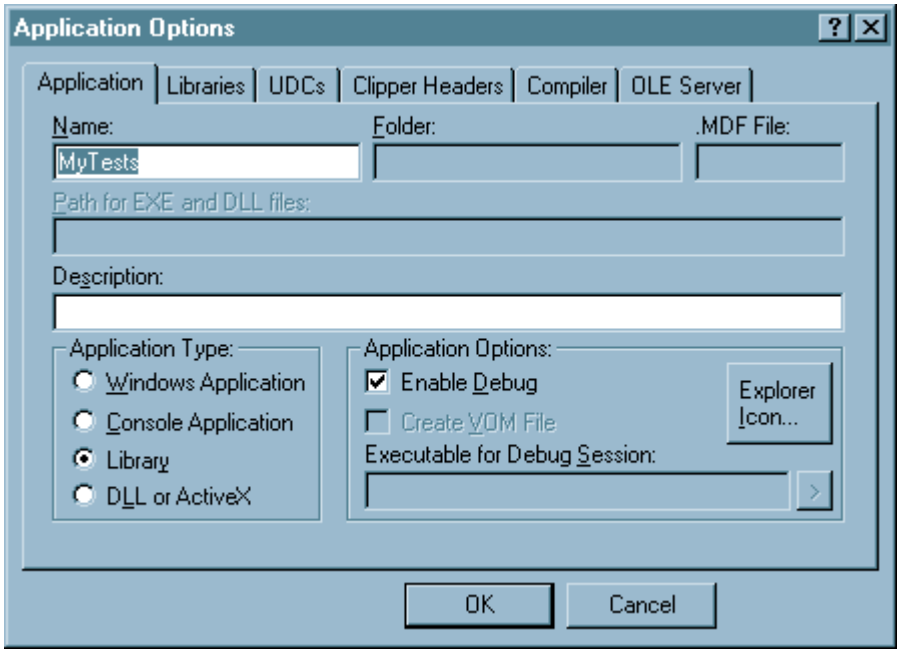
Properties	Methods	Comments
iTestCaseCount	BeginTesting	Initialized some variables, and set state
iCurrentTestCount	GetCurrentTestCount	This method is used to report the current test count to the TestRunnerDialog
iErrorCount	GetErrorCount	This method is used to report the error count to the TestRunnerDialog
iFailureCount	GetErrorsAndFailures	Returns the error and failure objects
sProcessState	GetFailureCount	This method is used to report the failure count to the TestRunnerDialog
bWasSuccessful	GetProcessState	This method is used to report the current processing state to the TestRunnerDialog
bStopTest	GetTestCaseCount	This method is used to report the total test count to the TestRunnerDialog
oFinalResult	GetUnitTestClassList	Builds a list of test case methods for a given test class
	HaltTesting	The TestResult object is also an observer of the UnitTestInterface class, and is queried by the TestResult object
	Init	
	RunTestsForClass	Key method for running the tests
	SetState	Starts the process of notifying all interested observers that a change in state has occurred in the UnitTestInterface object
	ShouldStop	Queried by TestResult
	Update	The UserTestInterface observes the TestResult object. This method is called by the NotifyObservers method of TestResult when the state of the TestResult object changes
	WasSuccessful	Queried by TestRunnerDialog

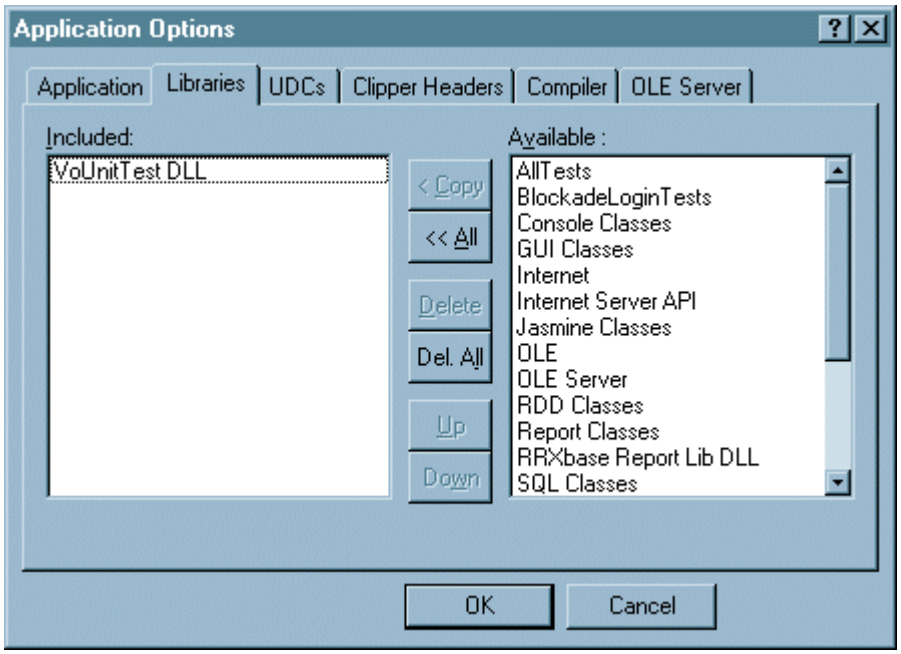
User-Defined TestCase Subclass (One or More Defined As Needed)

Properties	Methods	Comments
Varies as needed by test fixture	Axit	Add any class clean up here
	Init	The objects and variables in the test fixture are initialized here
	Suite	This method creates one self-contained TestCase object for each test method which has been defined in the subclass
	One or more test case methods as required	

The next step to using the VO Unit framework involves creating a library module for the test case classes. Include the VoUnitTest DLL library. This library was created when the VoUnitTest.DLL was compiled and linked. It should be found in the \CAVO25\BIN directory.

In this example it is called MyTests. The VO Unit framework is capable of supporting more than one test case class library. This is a nice feature for team development environments. Each team member can have a test case class library. This permits tests to be shared among team members.





User-Defined TestCase Class Example

There are five major steps to building a user-defined TestCase class.

First, create a subclass of TestCase class. Declare any instance variables needed here.

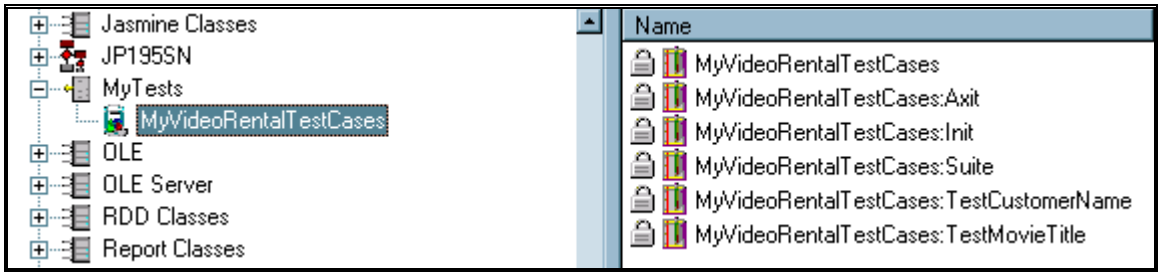
Next add the all important Init method. It is in this method that you will build the test fixture needed by the tests.

Next you should add an Axit method. This is optional, but might be handy to centralize any clean-up activity here.

Then create a Suite method. Any test methods that begin with the letters "Test" will be automatically added to the suite. However if you have test method names that do not begin with the letters "Test", you will have to add those test method names manually.

Finally, add one or more test methods that invoke the TestCase's Assert method. The suggested approach is to name the method beginning with the letters "Test". This will save the extra step of manually adding the test to the suite. In this example you will see two tests called TestCustomerName and TestMovieTitle.

The list of tests will continue to grow as more of your application is covered by various unit tests.



Now let's take a closer look at the MyVideoRentalTestCases class.

Subclassing TestCase Class

Any class that contains test methods must inherit the TestCase class. In this example a class that is intended to test the VideoRental class is simply called MyVideoRentalTestCases.

Declare instance variables needed by the class. These variables will be referenced when the test fixture is built, as well as being referenced by each of the test methods.

Normally, tests are kept in their own library module, and application-specific libraries are not included in the test library. Therefore, the variables are declared as types of OBJECT in order to avoid an UNKOWN CLASS compiler error.

In the example below, some movie, rental, and customer objects are declared. These are the objects that will be exercised by the methods of the MyVideoRentalTestCases class.

```
CLASS MyVideoRentalTestCases INHERIT TestCase

      // movie objects
      PROTECT oLionKing                    AS OBJECT

      // Lion King rental objects
      PROTECT oLionKingTwoDayRental        AS OBJECT

      // customer object
      PROTECT oCustomer                    AS OBJECT
```

Creating a TestFixture

A test fixture essentially contains the objects that act as samples and support for testing. All subclasses of TestCase are expected to create and destroy test fixtures by overriding the Init and Axit methods of the class. Though most of the time you won't need to do anything in the Axit method, the garbage collector will handle any cleanup.

```
METHOD Init(sTestName) CLASS MyVideoRentalTestCases

  LOCAL CHILDRENS := 2        AS INT
  LOCAL REGULAR := 0          AS INT
  LOCAL NEW_RELEASE := 1      AS INT

  IF !Empty(sTestName)
    SELF:sName := sTestName
  ENDIF

  // build the test fixture for the class here

  // create some movie objects
  SELF:oLionKing := CreateInstance(#Movie, "The Lion King", CHILDRENS)

  // create some rental objects
  SELF:oLionKingTwoDayRental := CreateInstance(#Rental, SELF:oLionKing, 2)
  SELF:oLionKingThreeDayRental := CreateInstance(#Rental, SELF:oLionKing,
3)
  SELF:oLionKingFourDayRental := CreateInstance(#Rental, SELF:oLionKing, 4)

// create a customer object
  SELF:oCustomer := CreateInstance(#Customer, "John Q. Public")

  RegisterAxit(SELF)

RETURN SELF
```

Building a Test Method

When a test case is created, the constructor gets a string argument that is the name of the method being tested; this creates one object of the test class that tests that one method. Because each test case is isolated, if one test case fails, it does not affect the test fixtures of subsequent test cases. Also, because the test cases are independent of each other, they can be run in any order.

The test is actually performed in the Assert method of the TestCase class. If the value inside the assert is true, all is well; if not, an error or failure is signaled and subsequently displayed in the ListView control on the TestRunner dialog.

In this example we are testing that the customer "John Q. Public" is contained in the text string returned by the RentalStatement method.

CA-Visual Objects has many built-in functions that will aid in developing tests. Any function that returns a Boolean value can be used. Also functions like AScan() and Eval(), which can contain code blocks, make for very flexible and powerful testing scenarios. In this example the Instr() function is used.

If the string we are looking for is not found, a failure will be indicated and the failure message "Customer name not found." will be displayed in the Invalid Condition column. The line number where the failure occurred will also be displayed.

```
METHOD TestCustomerName() CLASS MyVideoRentalTestCases

  LOCAL sResult   AS STRING

  sResult := SELF:oCustomer:RentalStatement()

  SELF:Assert(  Instr("John Q. Public", sResult), ;   // expected result
                "Customer name not found.", ; // failure message
                ProcLine(0))                          // procedure line

  RETURN NIL
```

Creating the TestSuite

The Suite method is responsible for building an array of tests for a respective class. This method automatically adds any test method that begins with the letters "Test" to the array via the DynamicallyAddTestsToSuite method. In addition, the Suite method is designed to permit adding test and suites manually. The object that is returned by this method contains the all of the TestCase objects that will be run by the UnitTestInterface's RunTestForClass method.

```
METHOD Suite() CLASS MyVideoRentalTestCases

LOCAL oSuite      AS TestSuite

// construct the suite
oSuite := TestSuite{}

// dynamically add any test methods which begin with "Test"
oSuite := SELF:DynamicallyAddTestsToSuite(SELF, oSuite)

// manually add any other tests or suites here
// e.g. oSuite:AddTest(MyVideoRentalTestCases{"XYZTest..."})
// e.g. oSuite:AddTest(CreateInstance(#AnotherClass):Suite())

RETURN oSuite
```

Dynamically Adding TestCases to a TestSuite

This method of the TestSuite class builds a test suite that contains a test case for every method that starts with the letters "Test". This convenient method was designed so that you would not have to manually edit the Suite method each time a TestCase was added to or deleted from the user-defined test case class.

The constructor for the test suite takes a test case class object and a suite object as parameters. Using the test case class object, it builds a list of method names for that class. It then examines each method name to see if it begins with the letters "Test". If so, it creates a test object with that method name via the SelfFactory method and adds the test to the suite.

```
METHOD DynamicallyAddTestsToSuite(oTestCaseClass, oSuite) CLASS TestCase

LOCAL aMethodList AS ARRAY
LOCAL oTest       AS OBJECT
LOCAL i           AS INT

aMethodList := ArrayBuild()

// dynamically add all methods of oTestCaseClass
// which begin with the string "Test"

aMethodList := MethodList(oTestCaseClass)
IF ALen(aMethodList) > 0
  FOR i := 1 UPTO ALen(aMethodList)
    IF SELF:IsTestMethod(aMethodList[i])

oTest := SELF:SelfFactory(ClassName(oTestCaseClass),  ;
  Symbol2String(aMethodList[i]))
      oSuite:AddTest(oTest)

    ENDIF
  NEXT
ENDIF

RETURN oSuite
```

Creating a Master TestSuite

The VO Unit framework is capable of automatically running tests for multiple suites. In this example the suite method of the AllTestCases class combines the test suites from the MyVideoRentalTestCases and YourVideoRentalTestCases.

```
METHOD Suite() CLASS AllTestCases

LOCAL oSuite      AS TestSuite
LOCAL oMyTests    AS OBJECT
LOCAL oYourTests  AS OBJECT

// create objects for the other test cases
oMyTests := CreateInstance(#MyVideoRentalTestCases)
oYourTests := CreateInstance(#YourVideoRentalTestCases)

// construct the suite
oSuite := TestSuite{}

// automatically add any methods of this class which begin with "Test"
oSuite := SELF:DynamicallyAddTestsToSuite(SELF, oSuite)

// add any other tests or suites here
oSuite:AddTest(oMyTests:Suite())
oSuite:AddTest(oYourTests:Suite())

RETURN oSuite
```

The first step is to create instances of each TestCase class that will be combined. In this example the tests from MyVideoRentalTestCases and YourVideoRentalTestCases are being combined.

The AddTest method of the TestSuite class will accept either a single test or an entire suite of tests from another TestCase class.

A call to the Suite method for both the MyVideoRentalTestCases and YourVideoRentalTestCases classes is all that is needed to combine the test cases.

The oSuite object of AllTestCases class now contains a collection of the following tests:

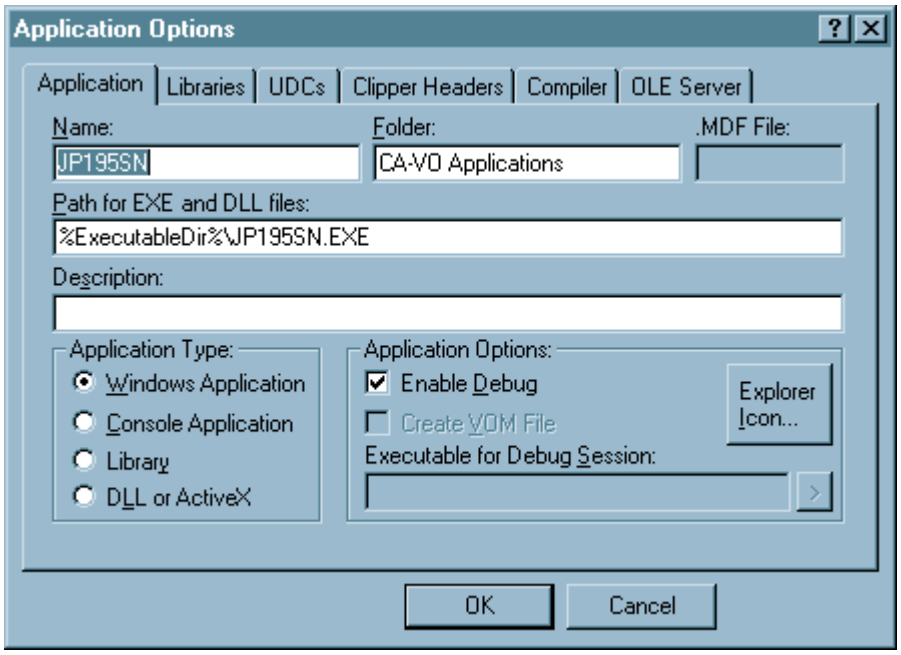
TestCustomerName

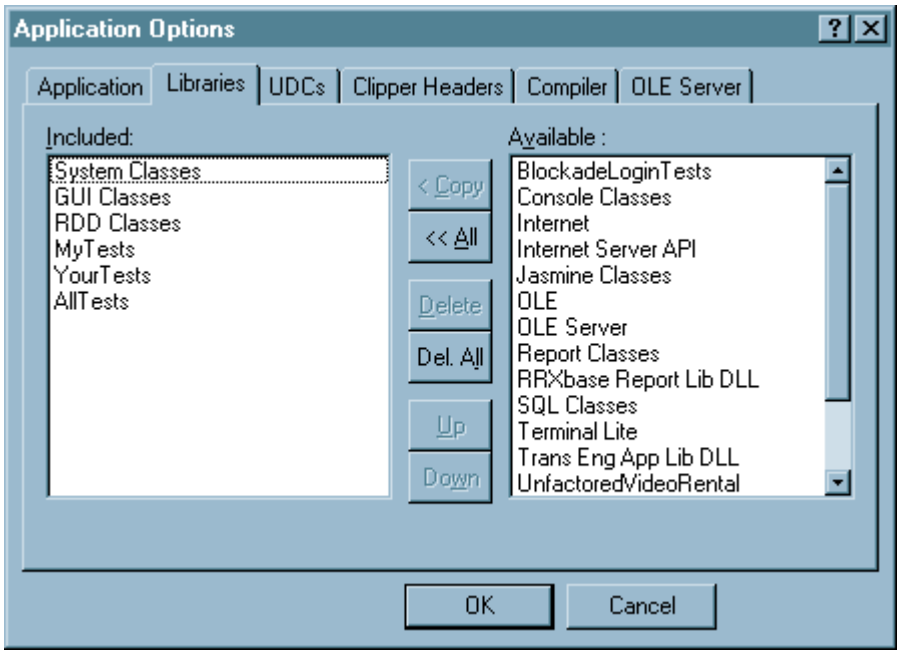
TestMovieTitle

TestFrequentRenterPoints

TestRentalCalculation

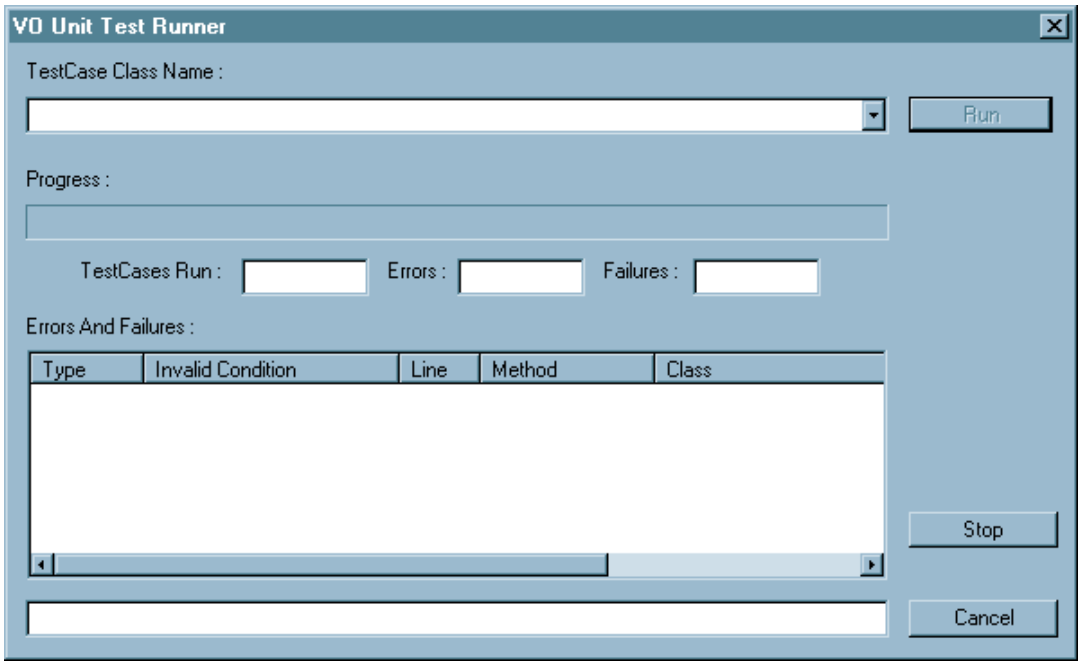
Once the test case library has been created, include it in the main application. Three test libraries are included in the example, MyTests, YourTests, and AllTests.





In the Start method of the application, add a local reference to the TesterRunnerDialog. Note that the CreateInstance function is used here so that you don't have to include the VoUnitTest DLL library in the main application.

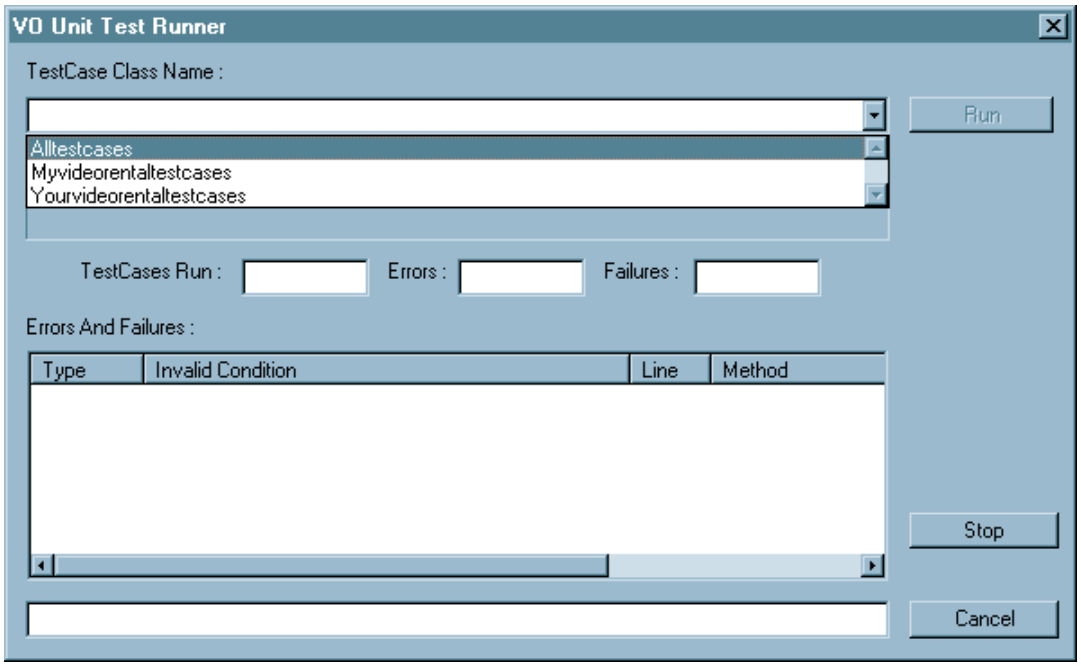
```
METHOD Start() CLASS App

LOCAL oMainWindow       AS StandardSDIWindow
      LOCAL oTester           AS OBJECT

      SELF:Initialize()

      oMainWindow := StandardSDIWindow{SELF}
      oMainWindow:Show(SHOWCENTERED)

      oTester := CreateInstance(#TestRunnerDialog, SELF)
      oTester:Show()
      .
      .
      .
```



Control	Comment
ListBox	Contains a list of all class names that inherit from the class TestCase
ProgressBar	Sets color on result: Green for success, Red for errors or failures
FixedText	Displays count of TestCases, Errors, Failures, and text for status messages
ListView	Error or failure result messages are displayed here
Button	Run, Stop, Cancel

Running the TestSuite(s)

It is very easy to run the tests. Simply select a test case from the drop-down list box and click the Run button.



The name of the test case class selected from the ListBox control is passed to the RunTestsForClass method on the UnitTestInterface object. This important method is responsible for running the tests, so let's look at this method in more detail.

After doing some initialization, it creates an instance of the TestCase. It then gets a TestResult object that will be used to report the status of the test. The tests are actually invoked by the Suite method of the TestCase class.

```
METHOD RunTestsForClass(sTestClassName) CLASS UnitTestInterface

      .
      .
      .

      oTestCase := CreateInstance(AsSymbol(sTestClassName))
      oResult := oTestCase:DefaultResult(SELF)

      // run the tests
      oSuite := oTestCase:Suite()

      oFinalResult := oSuite:Run(oResult)
```

When a Run message is sent to a TestSuite object, a TestResult object is returned. In this example the AllTestCases class Suite method was executed. As seen earlier, this suite included all of the tests from MyVideoRentalTestCases and YourVideoRentalTestCases.

The dialog indicates that there were four TestCases run. This is because the suite for AllTestCases included the following tests:

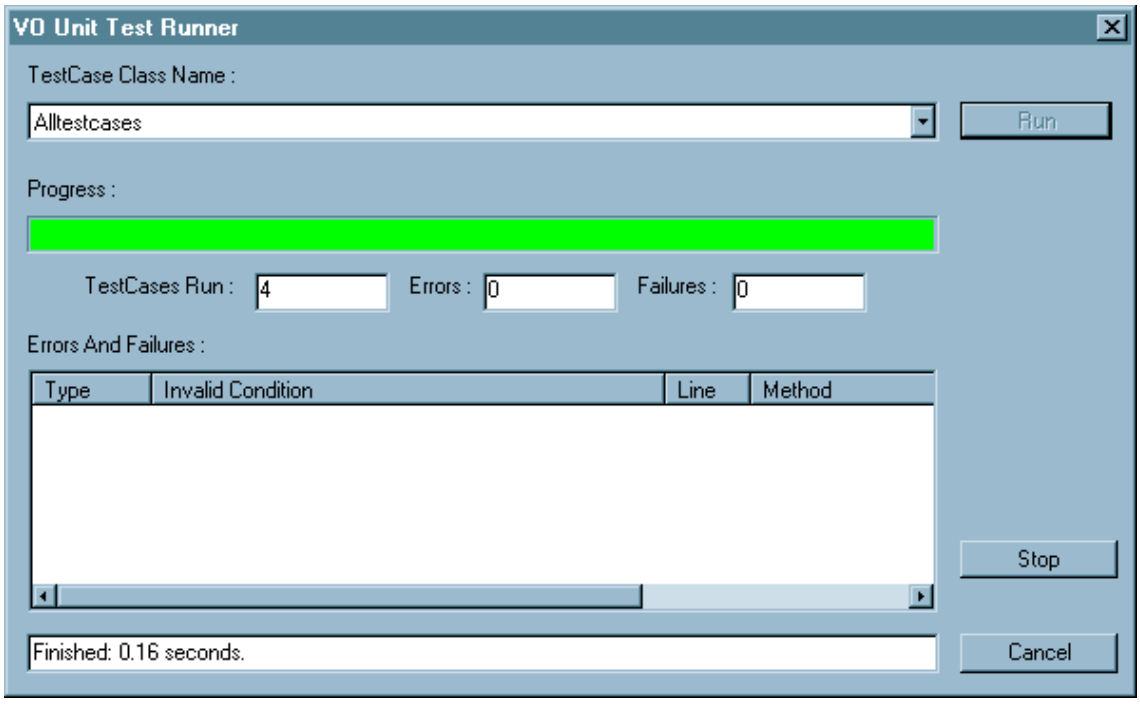
TestCustomerName

TestMovieTitle

TestFrequentRenterPoints

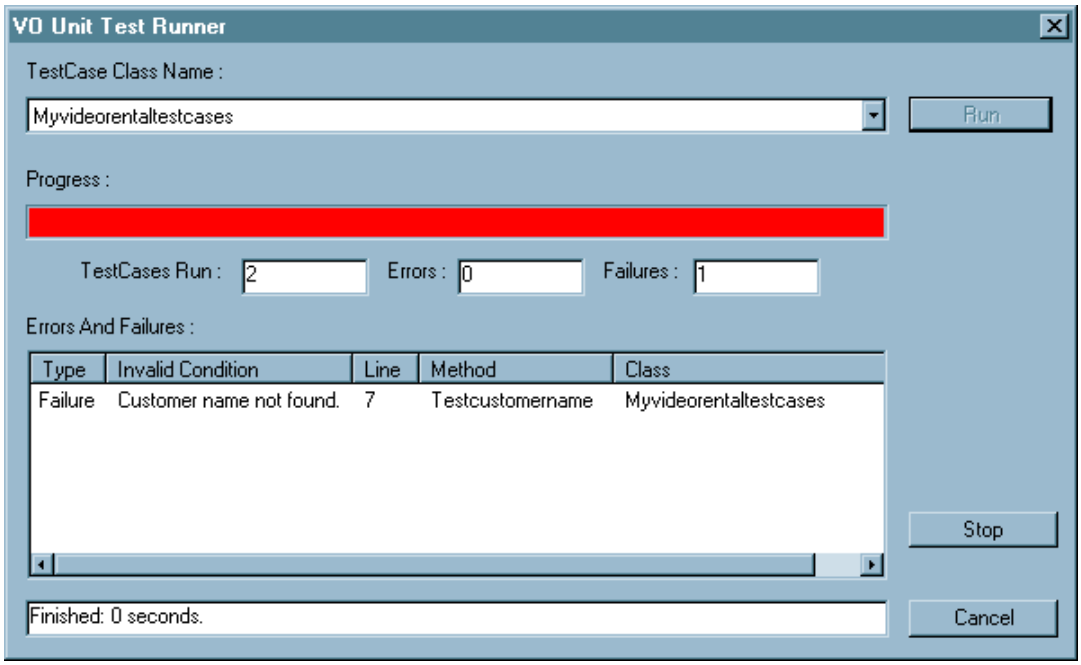
TestRentalCalculation

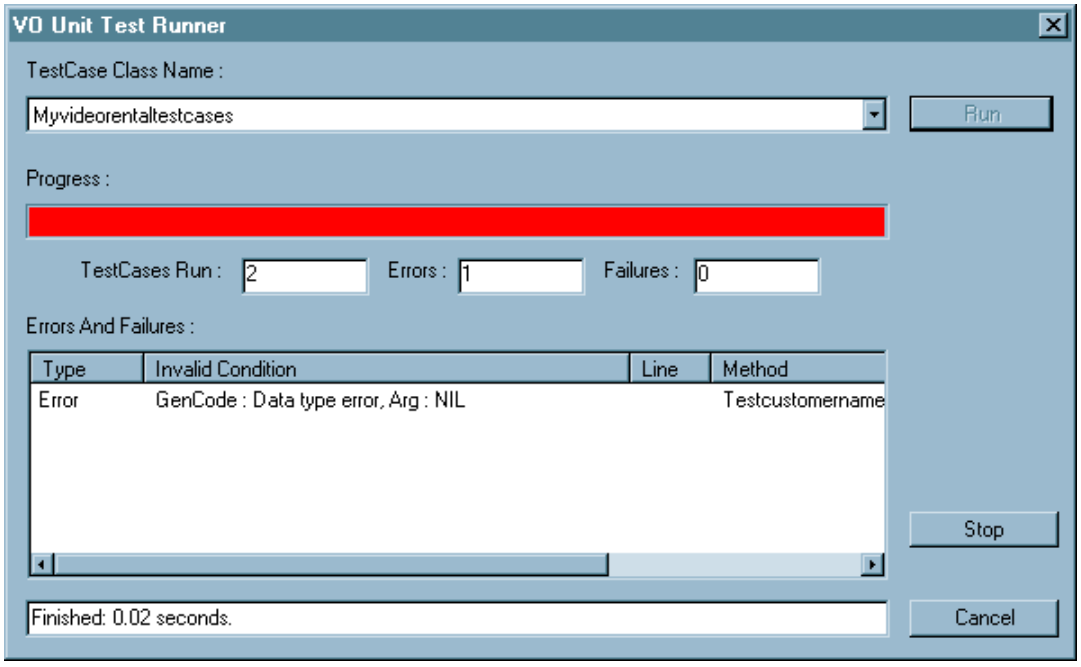
When all of the tests run successfully, the VO Unit Test Runner will look similar to this.



The color of the progress bar will be green and the Errors and Failures count will be zero. Also the ListView control will not contain any information.

If this TestResult object contains any failures or errors generated by the tests, the VO Unit Test Runner dialog will look similar to this. The information contained in the ListView control will help in diagnosing and correcting any problems. However the color of the progress bar will be red, and the count of Errors and/or Failures will be greater than zero.





Unit Testing and Refactoring

This example will explore how to use VO Unit when refactoring the RentalStatement method of the Customer class.

The purpose of this method is to calculate and print a statement of customer charges at a video store. This method is more complex than it should be, because it is doing several things other than printing the rental statement. Let's simplify the method and move the code not directly involved in printing the statement elsewhere. However, we want to insure that our changes do not introduce errors into the application.

Decomposing and Redistributing the RentalStatement Method

These are the tasks we'll be doing in this refactoring. Make note of the rhythm: a little testing; a little coding; a little testing; a little coding.

- Before making any changes - run tests.

  Four user-defined test cases have been developed, and all tests run successfully with the application as it is currently written.

Module	Test Class	Tests
MyTests	MyVideoRentalTestCases	TestCustomerName
MyTests	MyVideoRentalTestCases	TestMovieTitle
YourTests	YourVideoRentalTestCases	TestFrequentRenterPoints
YourTests	YourVideoRentalTestCases	TestRentalCalculation

- Find a logical clump of code and extract it into its own method - run tests.

  The case statement seems like a good choice. The purpose of this section of code is to determine the amount for each line of the statement. Therefore, let's call the new method AmountFor. We can see from the code that in order to generate the amount, it uses the Rental object, so it will be passed in as a parameter.

```
Method AmountFor(oRental) CLASS Customer
```

- Move the amount calculation to the rental class - run tests.

```
Method AmountFor() CLASS Rental
```

- Change the old method to delegate to the new method - run tests.

```
Method AmountFor(oRental) CLASS Customer
      RETURN oRental:AmountFor()
```

- Change the reference in the old method to the new method - run tests.

```
// old reference
r4ThisAmount := SELF:AmountFor(oRental)

// changed to
r4ThisAmount := oRental:AmountFor()
```

- Remove the old method - run tests.

Before Refactoring, the For Loop Looks Like This:

```
FOR n := 1 UPTO ALen(SELF:_Rentals)
    r4ThisAmount := 0
    oRental := SELF:_Rentals[n]

    // determine amounts for each line
    DO CASE
    CASE oRental:GetMovie():GetPriceCode() = REGULAR
      r4ThisAmount += 2
      IF oRental:GetDaysRented() > 2
        r4ThisAmount += (oRental:GetDaysRented() - 2) * 1.5
      ENDIF

    CASE oRental:GetMovie():GetPriceCode() = NEW_RELEASE
      r4ThisAmount += oRental:GetDaysRented() * 3

    CASE oRental:GetMovie():GetPriceCode() = CHILDRENS
      r4ThisAmount += 1.5
      IF oRental:GetDaysRented() > 3
        r4ThisAmount += (oRental:GetDaysRented() - 3) * 1.5
      ENDIF

    ENDCASE

    // add frequent renter points
    iFrequentRenterPoints += 1

    // add bonus for a two day new release rental
    IF (oRental:GetMovie():GetPriceCode() = NEW_RELEASE) .and. ;
            oRental:GetDaysRented() > 1

      iFrequentRenterPoints += 1

    ENDIF

    // show figures for this rental
    sResult += voTab + oRental:GetMovie():GetTitle() + voTab + ;
                  AllTrim(Str(r4ThisAmount)) + voCRLF

    r4TotalAmount += r4ThisAmount

  NEXT
```

After Refactoring, the For Loop Looks Like This:

```
FOR n := 1 UPTO ALen(SELF:_Rentals)
  r4ThisAmount := 0
  oRental := SELF:_Rentals[n]

  // determine amounts for each line
  r4ThisAmount := oRental:AmountFor()

  // add frequent renter points
  iFrequentRenterPoints += 1

  // add bonus for a two day new release rental
  IF (oRental:GetMovie():GetPriceCode() = NEW_RELEASE) .and. ;
            oRental:GetDaysRented() > 1

    iFrequentRenterPoints += 1

  ENDIF

  // show figures for this rental
  sResult += voTab + oRental:GetMovie():GetTitle() + voTab + ;
                        AllTrim(Str(r4ThisAmount)) + voCRLF

  r4TotalAmount += r4ThisAmount

NEXT
```

In this short example, we made the code better by moving code to a more appropriate class and used VO Unit to insure than no errors were introduced because of the code modification. The same strategy could be used to move the FrequentRenterPoints calculation to a more appropriate class, and VO Unit can give you the confidence you need to modify existing code.

Summary

You have seen the potential benefits of using the VO Unit testing framework and how to:

- Include the VO Unit testing framework in an application
- Subclass the TestCase class
- Set up a test fixture
- Write a test
- Add tests to a suite of tests
- Run the test suites

The VO Unit framework will provide you with a basic structure that permits you to write and save and run your tests. Over time you will build a library of tests that thoroughly exercise the non-GUI classes contained in the code. This will lead to classes that contain fewer errors and an increased assurance that the classes are performing as you intended. In addition, changes can be made with confidence because the tests will indicate if errors have been introduced.

Give VO Unit a try. When you are tempted to fire-up the debugger, or use the famous "? or ??" commands, write a test instead. Initially, it may seem that you are not making much progress, but stay with it. What you will discover is that you are able to write better quality software in less time with less stress and with increased satisfaction. Those are great rewards, and worth the effort.