MODULE 12: TESTING AND DEBUGGING

Module Overview

Software does not always behave as developers expect, and various types of errors and bugs frequently occur during development and production of every application. Microsoft Dynamics NAV 2013 includes a comprehensive set of tools and features that you can use to guarantee the highest quality of the customizations that you ship.

You can use the testing features to develop fully automated unit tests. These tests guarantee that your code always runs as designed, and that any bug or error that is introduced by a later change is detected immediately by the developer, instead of the end-users.

When there are bugs in the code, you can use the **Debugger** to step through code execution line by line and inspect the values of variables, parameters, or text constants.

Objectives

- Demonstrate the unit testing features of Microsoft Dynamics NAV 2013.
- Explain the test codeunits, test functions, and handler functions.
- Describe how to automate user interface testing.
- Explain the functionality and purpose of test runner codeunits.
- Develop a unit testing framework for the Seminar Management solution.
- Describe the Debugger functionality and features.
- Demonstrate the debugging process.

Prerequisite Knowledge

Test-driven Development Fundamentals

Test-driven development (TDD) is an advanced technique that uses automated unit tests to drive the design of software and force decoupling of dependencies. The technique results in a comprehensive suite of unit tests that you run at any time to provide feedback that the software is still working. Developers who use the agile development methodology favor this technique.

The motto of test-driven development is "Red, Green, Refactor."

- Red: Create a test and make it fail.
- Green: Make the test pass by any means necessary.
- Refactor: Change the code to remove duplications in your project and to improve the design while making sure that all tests still pass.

These steps explain the example TDD process:

1. Understand the requirements of the story, work item, or feature that you are working on.
2. **Red**: Create a test and make it fail.
   a. Imagine how the new code should be called and write the test as if the code already existed.
   b. Create the new production code stub. Write just enough code so that it compiles.
   c. Run the test. It should fail. This is a calibration measure to make sure that your test is calling the correct code and that the code is not working by accident. This is a meaningful failure, and you expect it to fail.

3. **Green**: Make the test pass by any means necessary.
   a. Write the production code to make the test pass. Keep it simple.
   b. Some developers advocate hard-coding of the expected return value first to verify that the test correctly detects success. This varies from practitioner to practitioner.
   c. If you have written the code so that the test passes as intended, you are finished. You do not have to write more code. If new functionality is still needed, then another test is needed. Make this one test pass, and then continue.
   d. When the test passes, you might want to run all tests to this point to guarantee that everything else is still working.

4. **Refactor**: Change the code to remove duplication in your project and to improve the design while ensuring that all tests still pass.
    a. Remove duplication that is caused by the addition of the new functionality.
    b. Make design changes to improve the overall solution.
    c. After each refactoring, rerun all the tests to make sure that they all still pass.

5. Repeat the cycle. Each cycle should be very short, and a typical hour should contain many Red/Green/Refactor cycles.

Characteristics of a Good Unit Test

A good unit test has the following characteristics:

- Runs fast. If the tests are slow, they will not be run frequently.
- Is very limited in scope. If the test fails, the source of the problem should be obvious. It is important to only test one thing in a single test.
- Runs and passes in isolation. If the tests require special environmental setup or fail unexpectedly, then they are not good unit tests. Change them for simplicity and reliability. Tests should run and pass on any computer. The "it works on my box" excuse does not work.
- Clearly reveals its intention. Another developer should be able to view the test and understand what is expected of the production code.

Benefits of Test-Driven Development

Even though the Test-Driven Development approach seems to add overhead and increase the total amount of work that is needed to complete a task, it has the following benefits:

- The suite of unit tests provides constant feedback that each component is still working.
- The unit tests act as documentation that cannot go out-of-date, unlike separate documentation that frequently is outdated.
- When the test passes and the production code is refactored to remove duplication, it is clear that the code is finished. The developer can move on to a new test.
- Test-driven development forces critical analysis and design because you cannot create production code without truly understanding the result that you want and how to test it.

- The software is better designed, that is, loosely coupled and easily maintainable. The developer is free to make design decisions and refactor at any time with the confidence that the software is still working. This confidence is gained by running the tests. The need for a design pattern may emerge, and the code can be changed at that time.
- The test suite acts as a regression safety net on bugs: If a bug is found, the developer should create a test to reveal the bug and then modify the production code so that the bug is removed and all the other tests still pass. On each successive test run, all previous bug fixes are verified.
- Debugging time is reduced.

Note: *Test-Driven Development practices may be enabled to varying extents in different development environments. Practices that work for some development tools may not work in others. Microsoft Dynamics NAV 2013 enables you to write automated unit tests that you can run easily and frequently. Therefore it enables and promotes the most important TDD practices.*

Test Features

Microsoft Dynamics NAV 2013 includes the following features to help you test your application:

- Test codeunits
- Test runner codeunits
- Test pages
- UI handlers
- ASSERTERROR statement

Test Codeunits

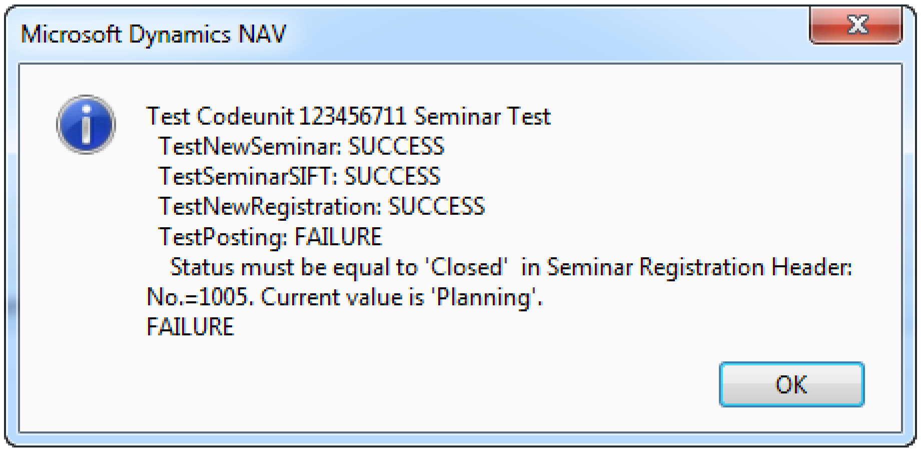
Test codeunits are a special type of codeunit that enable you to write and run code that automatically tests application functionality. Test codeunits can contain the following types of functions:

- Test functions
- Handler functions
- Normal functions

When a test codeunit runs, it does the following:

- Runs the OnRun trigger.
- Runs each test function in the test codeunit.
- Records the result in a log.
- Displays the resulting log on message window.

The result of a test function is either SUCCESS or FAILURE. If any error is raised by either the code that is tested or the test code itself, then the result is FAILURE, and the error text is included in the results log. The codeunit continues to run the remaining test functions in the codeunit whether a function results in SUCCESS or FAILURE.

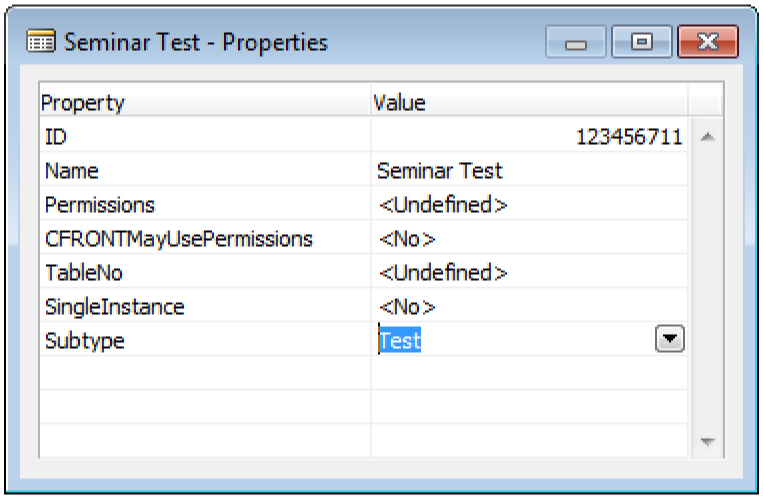


```
Test Codeunit 123456711 Seminar Test
 TestNewSeminar: SUCCESS
 TestSeminarSIFT: SUCCESS
 TestNewRegistration: SUCCESS
 TestPosting: FAILURE
  Status must be equal to 'Closed'  in Seminar Registration Header:
No.=1005. Current value is 'Planning'.
FAILURE
```

FIGURE 12.1: EXAMPLE

You create a test function by setting the Subtype codeunit property to Test:

1. Design a codeunit.
2. Click **View** > **Properties**.
3. Set **Subtype** to Test.



Property	Value
ID	123456711
Name	Seminar Test
Permissions	<Undefined>
CFRONTMayUsePermissions	<No>
TableNo	<Undefined>
SingleInstance	<No>
Subtype	Test

FIGURE 12.2: SUBTYPE PROPERTY FOR TEST CODEUNITS

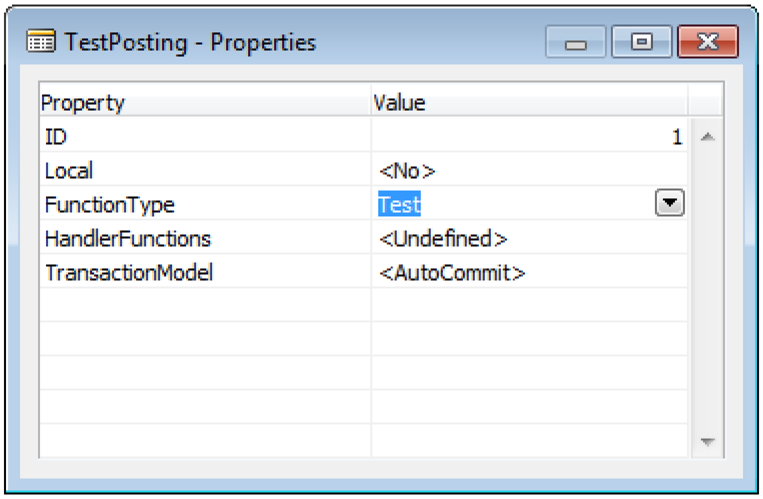
Test Functions

A *test function* is a special type of a function that you use to perform a test on an area of the application. Test functions do not accept any parameters, and do not return a value. You can only define test functions in a test codeunit.

You can check or set the type of a function in a test codeunit by inspecting its FunctionType property by doing the following:

1. In a test codeunit, click **View > C/AL Globals**.
2. Click the **Functions** tab.
3. Select a function.
4. Click **View > Properties**.
5. Check or set the FunctionType property.

The test functions have their FunctionType property set to Test.


TestPosting - Properties

Property	Value
ID	1
Local	<No>
FunctionType	Test
HandlerFunctions	<Undefined>
TransactionModel	<AutoCommit>

FIGURE 12.3: THE FUNCTIONTYPE PROPERTY FOR TEST FUNCTIONS

Note: *You do not have to explicitly set the FunctionType property for test functions. This property is automatically set to Test for all new functions that you create in a test codeunit.*

Transaction Model for Test Functions

Most of the tests in Microsoft Dynamics NAV 2013 are data related, and most of data is very volatile. Writing tests that are completely independent of data can be difficult, and failed tests could easily cause data chaos. Therefore, set the transaction behavior for each test function by setting the TransactionModel property.

The TransactionModel property has following values.

TransactionModel	Remarks
AutoCommit	A commit is issued at the end of the test function. If an error occurs during the test function, then the transaction is rolled back. If an error occurs and you catch it with an ASSERTERROR statement, then the transaction is rolled back. If the code that is being tested calls the **COMMIT** function before an error occurs, then the transaction is rolled back only to the point at which the **COMMIT** was called. AutoCommit is the default value.

TransactionModel	Remarks
AutoRollback	The transaction is rolled back after test execution. Calls to the **COMMIT** function fail with an error during a test that is set to AutoRollback.
None	The None transaction model enables a TestPage to behave exactly like an actual page. The test function does not have an open write transaction. Therefore, it cannot write directly to the database. Each interaction with the database occurs through TestPage triggers. They open their own write transactions. At the end of each transaction, if no errors occurred, then all changes are committed to the database. If an error occurred, then changes are rolled back to the end of the transaction. This differs from the AutoCommit and AutoRollback transaction models. With AutoCommit and AutoRollback, the test function starts a write transaction. Triggers that are invoked by the test code inherit this open transaction instead of running in their own separate transactions. With the AutoCommit and AutoRollback settings, if several page interactions are invoked from test code, then they share the same transaction. With the None setting, each page interaction runs in a separate transaction. You should use the None transaction model for tests that do not write to the database, such as tests that validate calculation formulas or tests that only read from the database.

Note: *The **ERROR** function always causes the transaction to be rolled back, regardless of the TransactionModel setting.*

***Note:** If you select the AutoRollback transaction model, then the code that the function tests must not call the **COMMIT** function. If the code calls it, an error occurs and the test function reports FAILURE.*

Demonstration: Creating and Running a Test Codeunit

This demonstration shows how you can create a test codeunit, add a test function, and run the test codeunit directly to view the resulting log.

Demonstration Steps

1. Create a new codeunit and save it as 90001, Test Sales.
   a. In Object Designer, click Codeunit.
   b. Click **New**.
   c. Click **File > Save**.
   d. In the **Save As** dialog box, in the **ID** field, type "90001".
   e. In the **Name** field, type "Test Sales".
   f. Make sure that the **Compiled** check box is selected, and then click **OK**.

2. Set the properties to make the new codeunit a test codeunit.
   a. Click **View > Properties**, or press SHIFT+F4.
   b. Set Subtype to Test.
   c. Close the **Properties** window.

3. Create a typical function that creates a sales order and returns a reference to its header.
   a. Click **View > C/AL Globals**.
   b. On the **Functions** tab, in the first empty line, type "CreateSalesOrder".
   c. Select the **CreateSalesOrder** function, and then press SHIFT+F4.
   d. Set the FunctionType property to Normal.
   e. Close the **Properties** window.
   f. Click **Locals**.
   g. On the **Parameters** tab, enter the following information.

Var	Name	DataType	Subtype	Length
Yes	SalesHeader	Record	Sales Header	
	CustNo	Code		20
	ItemNo	Code		20
	Qty	Decimal		

h. On the **Variables** tab, enter the following information.

Name	DataType	Subtype
SalesLine	Record	Sales Line

i. Close the **C/AL Locals** and **C/AL Globals** windows.

j. In the function trigger for the **CreateSalesOrder** function, enter the following C/AL code.

```
SalesHeader.INIT;

SalesHeader."Document Type" := SalesHeader."Document Type"::Order;

SalesHeader."No." := '';

SalesHeader.INSERT(TRUE);

SalesHeader.VALIDATE("Sell-to Customer No.",CustNo);

SalesHeader.MODIFY(TRUE);

SalesLine.INIT;

SalesLine."Document Type" := SalesHeader."Document Type";

SalesLine."Document No." := SalesHeader."No.";

SalesLine."Line No." := 10000;

SalesLine.INSERT(TRUE);

SalesLine.VALIDATE(Type,SalesLine.Type::Item);

SalesLine.VALIDATE("No.",ItemNo);

SalesLine.VALIDATE(Quantity,Qty);

SalesLine.MODIFY(TRUE);
```

4. Add a new function to test creating a sales order, and name it **TestReleaseSalesOrder**. Make sure that it automatically rolls back any data changes after it finishes.
   a. Click **View > C/AL Globals**.
   b. On the **Functions** tab, in the first empty row, type "TestReleaseSalesOrder".
   c. Select the **TestReleaseSalesOrder** function, and then press SHIFT+F4.
   d. Set the TransactionModel property to AutoRollback.

e. Close the **Properties** window.

f. Click **Locals**.

g. On the **Variables** tab, create the following local variable.

Name	DataType	Subtype
SalesHeader	Record	Sales Header

h. Close the **C/AL Locals** and **C/AL Globals** windows.

i. In the TestReleaseSalesOrder function trigger, enter the following C/AL code.

```
CreateSalesOrder(SalesHeader,'10000','1000',10);

SalesHeader.TESTFIELD(Status,SalesHeader.Status::Open);

CODEUNIT.RUN(CODEUNIT::"Release Sales Document",SalesHeader);

SalesHeader.TESTFIELD(Status,SalesHeader.Status::Released);
```

5. Save the codeunit, run it, and then view the results.

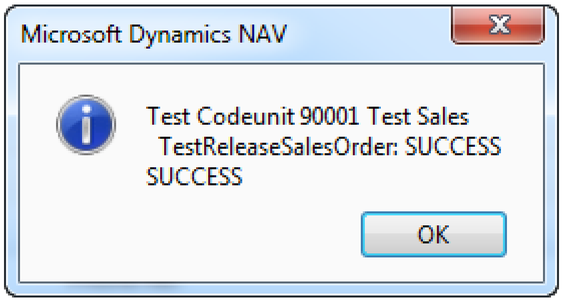
a. Click **File > Save**. Click OK in the **Save** dialog box.

b. Close the codeunit.

c. In Object Designer, select the codeunit 90001, TestSales.

d. Click **Run** to run the test.

e. After the test finishes, view the log message.



```
Test Codeunit 90001 Test Sales
 TestReleaseSalesOrder: SUCCESS
SUCCESS
```

FIGURE 12.4: TESTSALES INITIAL TEST RESULTS

ASSERTERROR Statement

When Microsoft Dynamics NAV 2013 runs test codeunits, a run-time error causes the currently running test function to return FAILURE. However, you sometimes have to test whether a specific error occurs. From the perspective of a test function, the error is expected. For example, if you want to test that you cannot change a value of a field on a released sales header, you actually expect an error. If the error occurs, the test must return SUCCESS; on the other hand, if the error does not occur, the test must return FAILURE, because the application behavior is not as you expected.

You use the ASSERTERROR statement in test functions to test how your application behaves under failing conditions. The ASSERTERROR keyword specifies that an error is expected at run time in the statement that follows the ASSERTERROR keyword.

If a simple or compound statement that follows the ASSERTERROR keyword causes an error, then the execution successfully continues to the next statement in the test function. You can receive the error text of the statement by using the **GETLASTERRORTEXT** Function.

If a statement that follows the ASSERTERROR keyword does not cause an error, then the ASSERTERROR statement causes the error. The test function that is running produces a FAILURE result. The error text states: "An error was expected inside an ASSERTERROR statement."

Note: When a run-time error occurs in Microsoft Dynamics NAV 2013, the transaction is rolled back. This applies even to the expected error when you use ASSERTERROR. However, the execution continues, the transaction is rolled back, and any data that was written to the database before ASSERTERROR does not exist in the database after ASSERTERROR. Therefore, you should write code that tests only whether the appropriate error has occurred after the ASSERTERROR statement.

Note: *If you must have two successive ASSERTERROR statements in a single test function, then you should write another test function, and move the second ASSERTERROR there. Every test function should be as limited in scope as possible, and should always test a single condition or case.*

Demonstration: Using ASSERTERROR in Test Functions

This demonstration shows how you can use the ASSERTERROR statement to test failing conditions.

Demonstration Steps

1. Add a new test function to the Test Sales codeunit to verify that you cannot modify header fields for a released order.
   a. Design codeunit 90001, Test Sales.
   b. Click **View > C/AL Globals**.
   c. On the **Functions** tab, select the row for the **TestReleaseSalesOrder** function.

Note: *Make sure that you select the whole row, not just the name text for the function.*

   d. Press the following keyboard shortcut: CTRL+C, DOWN, F3, CTRL+V.

Note: *This creates a copy of the* ***TestReleaseSalesOrder*** *function.*

   e. Overwrite the name of the copy of the **TestReleaseSalesOrder** function by using the following text: "TestChangeReleasedSalesOrder".
   f. At the end of the TestChangeReleasedSalesOrder function trigger, enter the following code line.

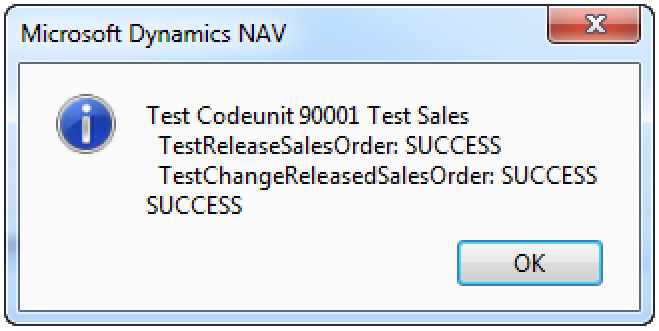
```
ASSERTERROR SalesHeader.VALIDATE("Location Code",'SILVER');
```

This is the complete function trigger code for **TestChangeReleasedSalesOrder**:

TestChangeReleasedSalesOrder Function

```
CreateSalesOrder(SalesHeader,'10000','1000',10);

SalesHeader.TESTFIELD(Status,SalesHeader.Status::Open);

CODEUNIT.RUN(CODEUNIT::"Release Sales Document",SalesHeader);

SalesHeader.TESTFIELD(Status,SalesHeader.Status::Released);

ASSERTERROR SalesHeader.VALIDATE("Location Code",'SILVER');
```

2. Save the codeunit, run it, and then view the results.
   a. Click **File > Save**. Click **OK** in the **Save** dialog box.
   b. Close the codeunit.
   c. In Object Designer, select the codeunit 90001, TestSales.
   d. Click **Run** to run the test.
   e. After the test is complete, view the log message.



```
Test Codeunit 90001 Test Sales
  TestReleaseSalesOrder: SUCCESS
  TestChangeReleasedSalesOrder: SUCCESS
SUCCESS
```

FIGURE 12.5: TESTSALES SECOND RUN RESULTS

Note: *The test fails because instead of calling the Sales-Post codeunit, you should call the Sales-Post (Yes/No). This is the codeunit that runs when users start the posting process.*

Handler Functions

When writing test codeunits, it is a best practice to fully automate tests that do not require any user input. In the event that any user interaction is required, such as making a choice, or confirming a condition, then the code must make those selections on behalf of the user. Moreover, to fully test the functionality of the solution, the test code must test all possible code branches that result from different user choices.

A handler function lets you automate tests by handling instances when user interaction is required by the code that is being tested. In these instances, the test function calls the handler function. This runs instead of the user interface. You specify that a function is a handler function by setting its FunctionType property.

Following are several types of handler functions:

- MessageHandler
- ConfirmHandler
- StrMenuHandler
- PageHandler
- ModalPageHandler
- ReportHandler
- RequestPageHandler

Each handler function replaces a specific type of user interaction that may occur in the code. Because there are several types of interactions, different handler functions take different parameters. The following table shows the function signatures for handler functions.

Handler Type	Function Signature
MessageHandler	<Function name>(<Msg> : Text[1024])
ConfirmHandler	<Function name>(<Question> : Text[1024]; VAR <Reply> : Boolean)
StrMenuHandler	<Function name>(<Options : Test[1024]; VAR <Choice> : Integer; <Instruction> : Text[1024])
PageHandler	<Function name>(VAR <variable name> : Page <page id>) <Function name>(VAR <variable name> : TestPage <testpage id>)
ModalPageHandler	<Function name>(VAR <variable name> : Page <page id>; VAR <Response> : Action) <Function name>(VAR <variable name> : Page <testpage id>)
ReportHandler	<Function name>(VAR <report name> : Report <report id>)
RequestPageHandler	<Function name>(VAR <TestRequestPage> : TestRequestPage)

Note: *When you create handler functions, you must define the appropriate parameters that are defined by the signature for the handler type. If the signature of your handler function does not match the expected signature, you cannot compile the codeunit.*

HandlerFunctions Property

Defining a handler function does not cause the handler to automatically replace the interaction. You must manually attach a handler function to a test function where the user interaction occurs.

To attach a handler function to a test function, you specify the handler function name in the HandlerFunctions property of the test function as follows:

1. On the **Functions** tab of the **C/AL Globals** window, select the test function.
2. Click **View > Properties**.
3. Type the handler function name in the **HandlerFunctions** property.

Note: *If the test function uses more than one handler function, separate the handler function names by a comma.*

Every handler function that you enter in the **HandlerFunctions** property must be called at least one time in the test function. If you run a test function that has a handler function listed, and that handler function is not called, then the test fails.

Demonstration: Using Handler Functions to Automate User Interaction

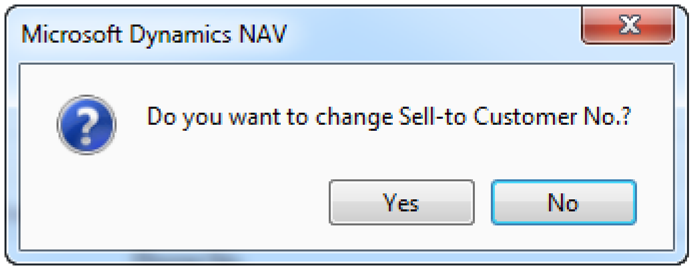
This demonstration shows how you can use the **ConfirmHandler** and **StrMenuHandler** functions to handle different types of user interaction.

Demonstration Steps

1. Add a new test function to the Test Sales codeunit to test changing the **Sell-to Customer No.** field for an existing sales order.
   a. Design codeunit 90001, Test Sales.
   b. Click **View > C/AL Globals**.
   c. Create a new test function, and name it "TestChangeCustomerOnSalesOrder".
   d. For the **TestChangeCustomerOnSalesOrder** function, define a new local variable for the **Sales Header** table, and call it SalesHeader.
   e. In the TestChangeCustomerOnSalesOrder function trigger, enter the following code.

```
CreateSalesOrder(SalesHeader,'10000','1000',10);

SalesHeader.VALIDATE("Sell-to Customer No.",'20000');
SalesHeader.MODIFY(TRUE);
```

2. Save the codeunit, run it, and then view the results.
   a. Click **File > Save**. Click **OK** to the **Save** dialog box.
   b. Close the codeunit.
   c. In Object Designer, select the codeunit 90001, TestSales.
   d. Click **Run** to run the test.
   e. The test codeunit requests confirmation:


Microsoft Dynamics NAV

Do you want to change Sell-to Customer No.?

FIGURE 12.6: TESTSALES ASKING FOR CONFIRMATION

Note: Tests should always be automated and never require user interaction.

3. Create a handler function to handle the confirmation dialog box. Attach it to the **TestChangeCustomerOnSalesOrder** function.
   a. Design the Test Sales codeunit.
   b. Create a new function and name it "HandleChangeCustomerConfirm".
   c. Select the **HandleChangeCustomerConfirm** function, and then press SHIFT+F4.
   d. Set **FunctionType** to ConfirmHandler.
   e. Close the **Properties** window.
   f. Click **Locals**.
   g. On the **Parameters** tab, enter the following information.

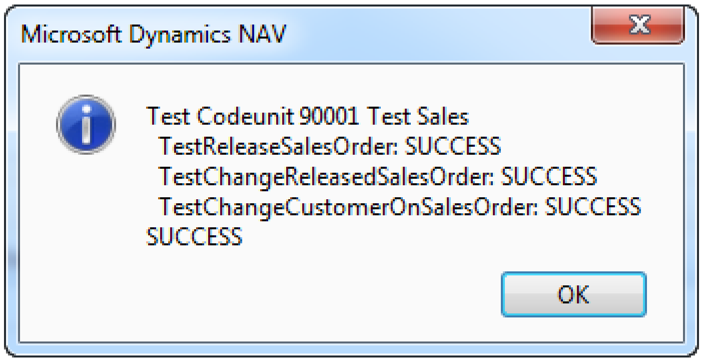
Var	Name	DataType	Length
	Question	Text	1024
Yes	Reply	Boolean	

   h. Close the **C/AL Locals** window.
   i. Select the **TestChangeCustomerOnSalesOrder** function, and then press SHIFT+F4.
   j. In the Value column for the HandlerFunctions property, type "HandleChangeCustomerConfirm".
   k. Close the **Properties** and **C/AL Globals** windows.

l. In the HandleChangeCustomerConfirm function trigger, enter the following code:

```
Reply := TRUE;
```

4. Save the codeunit, run it, and then view the results.
   a. Click **File > Save**. Confirm the **Save** dialog box.
   b. Close the codeunit.
   c. In Object Designer, select the codeunit 90001, TestSales.
   d. Click **Run** to run the test.
   e. After the test is complete, view the log message.



```
Test Codeunit 90001 Test Sales
 TestReleaseSalesOrder: SUCCESS
 TestChangeReleasedSalesOrder: SUCCESS
 TestChangeCustomerOnSalesOrder: SUCCESS
SUCCESS
```

FIGURE 12.7: TEST SALES RESULTS AFTER CONFIRMHANDLER IS IMPLEMENTED

5. Add a handler function to handle the **StrMenu** function for sales order posting.
   a. Design the **Test Sales** codeunit.
   b. Create a new function and name it “HandleSalesPostStrMenu”.
   c. Select the **HandleSalesPostStrMenu** function, and then press SHIFT+F4.
   d. Set **FunctionType** to StrMenuHandler.
   e. Close the **Properties** window.
   f. Click **Locals**.
   g. On the **Parameters** tab, enter the following information.

Var	Name	DataType	Length
No	Options	Text	1024
Yes	Choice	Integer	
No	Instruction	Text	1024

   h. Close the **C/AL Locals** window.
   i. On the **Variables** tab of the **C/AL Globals** window, declare an Option variable and name it PostingType.
   j. Select the PostingType variable and press SHIFT+F4.

k. Set the OptionString property to “Ship,Invoice,All” (make sure not to enter any spaces).

l. Close the **Properties** and **C/AL Globals** windows.

m. In the HandleSalesPostStrMenu function trigger, enter the following code.

```
CASE PostingType OF

  PostingType::Ship: Choice := 1;

  PostingType::Invoice: Choice := 2;

  PostingType::All: Choice := 3;

END;
```

6. Add a function to test posting of an invoice from a sales order. This process must fail. Therefore, make sure that the function tests for the correct failing conditions.

   a. Create a new function and name it “TestPostInvoiceFromSalesOrder”.

   b. For the **TestPostInvoiceFromSalesOrder** function, define a new local variable for the **Sales Header** table, and call it SalesHeader.

   c. Select the **TestPostInvoiceFromSalesOrder** function, and press SHIFT+F4.

   d. In the Value column for the HandlerFunctions property, type “HandleSalesPostStrMenu”.

   e. Close the **Properties** window.

   f. On the **Text Constants** tab, enter the following information.

Name	ConstValue
Text001	There is nothing to post.
Text002	Expected error:\%1\\Actual error:\%2

   g. Close the **C/AL Globals** window.

   h. In the TestPostInvoiceFromSalesOrder function trigger, enter the following code.

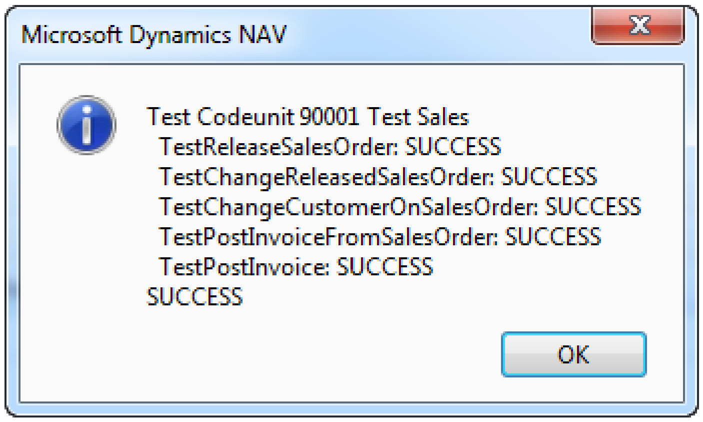
```
CreateSalesOrder(SalesHeader,'10000','1000',10);

PostingType := PostingType::Invoice;

ASSERTERROR CODEUNIT.RUN(CODEUNIT::"Sales-Post (Yes/No)",SalesHeader);

IF GETLASTERRORTEXT <> Text001 THEN

  ERROR(Text002,Text001,GETLASTERRORTEXT);
```

Note: *When this function runs, the Sales-Post (Yes/No) codeunit runs the* ***STRMENU*** *function, which asks users whether they want to ship, invoice, or both. This* ***STRMENU*** *is substituted with the code in the* ***HandleSalesPostStrMenu*** *function, which returns the option for Invoice. Then the posting fails because the invoice cannot be posted before shipment. This error is trapped by the ASSERTERROR. Finally, the code verifies whether the error message has the expected value of "There is nothing to post." and raises an error if it does not.*

7. Add a function which posts a shipment and then an invoice from a sales order.
   a. Create a new function and name it "TestPostSalesOrder".
   b. For the **TestPostSalesOrder** function, define a new local variable for the **Sales Header** table, and call it SalesHeader.
   c. Select the **TestPostSalesOrder** function, and press SHIFT+F4.
   d. In the Value column for the HandlerFunctions property, type "HandleSalesPostStrMenu".
   e. Close the **Properties** and the **C/AL Globals** windows.
   f. In the TestPostSalesOrder function trigger, enter the following code.

```
CreateSalesOrder(SalesHeader,'10000','1000',10);

PostingType := PostingType::Ship;

CODEUNIT.RUN(CODEUNIT::"Sales-Post (Yes/No)",SalesHeader);

PostingType := PostingType::Invoice;

CODEUNIT.RUN(CODEUNIT::"Sales-Post (Yes/No)",SalesHeader);
```

8. Save the codeunit, run it, and then view the results.
   a. Click **File > Save**. Click **OK** the **Save** dialog box.
   b. Close the codeunit.
   c. In Object Designer, select the codeunit 90001, TestSales.
   d. Click **Run** to run the test.
   e. After the test finishes, view the log message.



```
Test Codeunit 90001 Test Sales
  TestReleaseSalesOrder: SUCCESS
  TestChangeReleasedSalesOrder: SUCCESS
  TestChangeCustomerOnSalesOrder: SUCCESS
  TestPostInvoiceFromSalesOrder: SUCCESS
  TestPostInvoice: SUCCESS
SUCCESS
```

FIGURE 12.8: TEST SALES FINAL RESULTS

Testing Pages

Microsoft Dynamics NAV 2013 enables you to use variables of type TestPage to automate user interaction testing. With test pages, you can do the following:

- View or change the value of a field on a test page.
- View the data on page parts.
- View or change the value of a field on a subpage.
- Filter the data on a test page.
- Perform any actions that are available on the page.
- Browse to different records.

Note: Test functions and code on test pages run on the Microsoft Dynamics NAV 2013 Server instance, even though they simulate client interactions.

Test Pages and the TransactionModel Property

To create meaningful tests, you first must understand how transactions run on pages. In a typical user scenario, a user who is logged on to a client enters data into one field of a page. Then the user enters data in another field on the page. The user also checks the value of a third field. Finally, the user saves and closes the page. Every time that a user enters data into a field, C/AL code may be triggered and a new transaction is automatically started. The trigger code runs within this new transaction. Field data is sent to the server where it is processed and frequently updated in the database. When the C/AL code in the trigger is finished, the transaction is automatically committed to the database and the page is refreshed with updated data.

When you create test functions that exercise pages that interact with the database, you have the following options for simulating user scenarios, and then returning the database to its initial, familiar state:

- Set the TransactionModel property on the test function to AutoRollback. This assumes that the code that you test does not include calls to the **COMMIT** function. Any calls to the **COMMIT** function result in an error. Most business logic does not call the **COMMIT** function, but relies on implicit commits at the end of the outermost C/AL trigger. The test continues as follows:
    a. The test function starts a transaction.
    b. The test function initializes data in the database. Database changes are made in the transaction that was started by the test function.
    c. Fields on the test page are set or updated. Database changes are made in the transaction that was started by the test function.
    d. The test function reads the values of fields on the test page or reads from the database to validate the test.
    e. After the test function finishes, the transaction is rolled back and the database is returned to its initial state.

- If the code that you test includes calls to the **COMMIT** function, then set the TransactionModel property on the test function to AutoCommit. The test continues as follows:
    a. The test function starts a transaction.
    b. The test function initializes data in the database. Database changes are made in the transaction that was started by the test function.
    c. Fields on the test page are set or updated. Database changes are made in the transaction that was started by the test function.
    d. When the **COMMIT** function is called, changes are committed to the database.
    e. The test function reads the values of fields on the test page, or reads from the database to validate the test.
    f. After the test function finishes, changes are committed to the database. To return the database to its initial state, you must manually revert the changes by deleting, updating, or inserting records, or you must use the TestIsolation property on the test runner codeunit to roll back changes.

- Set the TransactionModel property on the test function to None to simulate the behavior of an actual user. The test function does not start a transaction and cannot write directly to the database. However, a new transaction is started every time that a field on the page is updated and C/AL code is triggered. At the end of each transaction, changes are automatically committed to the database. Use this option if your test does not write to the database. You do not have to initialize data in the database before the test starts. For example, use this option for tests that validate calculation formulas or tests that read from the database. The test continues as follows:
    a. If a field on the test page is set or updated, then the test page starts a transaction. At the end of the transaction, changes are committed to the database.
    b. The test function runs the test code.
    c. After the test finishes, no transactions are rolled back. To return the database to its initial state, you must manually revert the changes by deleting, updating, or inserting records, or you must use the TestIsolation property on the test runner codeunit to roll back changes.

Accessing Fields on Test Pages

You access the fields on a test page by using the dot notation. For example, if you have a test page variable named CustomerCard that represents the **Customer Card page**, then to access the **Name** field on the test page, you write CustomerCard.Name in your code.

To retrieve the value of a field or to write a value in a field, use the Value property. For example, if you have a test page variable named CustomerCard that represents the **Customer Card** page, then you can read the value from a field or assign a value to a field, by writing code that resembles the following.

```
CustNo := CustomerCard."No.".Value;

CustomerCard.Address.Value := ' 612 South Sunset Drive';
```

Accessing Page Parts and Subpages

You access page parts and subpages on a test page by using the dot notation. For example, to compare the value of the **No.** field on a page to the value of the **No.** field on a FactBox on the page, write the following code.

```
If CustomerCard."No.".Value <> CustomerCard."Sales Hist. Sell-to
FactBox"."No.".Value THEN

  ERROR('Page part data is not updated. Expected Customer No. is %1, actual
Customer No. is %2.',CustomerCard."No.".Value, CustomerCard."Sales Hist. Sell-to
FactBox"."No.".Value);
```

You can use the **Symbol Menu** to view page parts and subpages and to access the functions, properties, fields, and actions on test page parts and subpages.

Filtering Data on Test pages

To filter the data that can be accessed on a test page, you use the FILTER property and filter functions. For example, to filter the customers on the **Customer List** page based on a range of values in the **No.** field, write the following code.

```
CustomerList.FILTER.SETFILTER("No.", '20000..30000');
```

Invoking Actions on Test Pages

Any action that is available on a page is also available on the test page that mimics that page. You access page actions by using the dot notation and the **INVOKE** function. Use the **Symbol Menu** to view the actions that are available on a test page. To view actions that you designed by using **Page Designer**, select the test page variable in the first column of the symbols, and then select **Actions** in the second column.

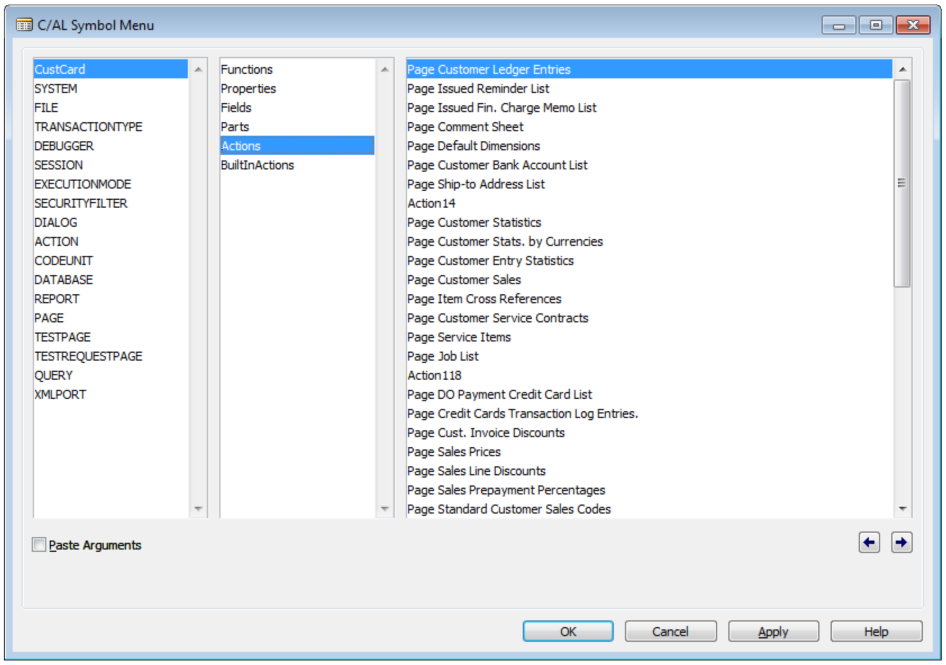
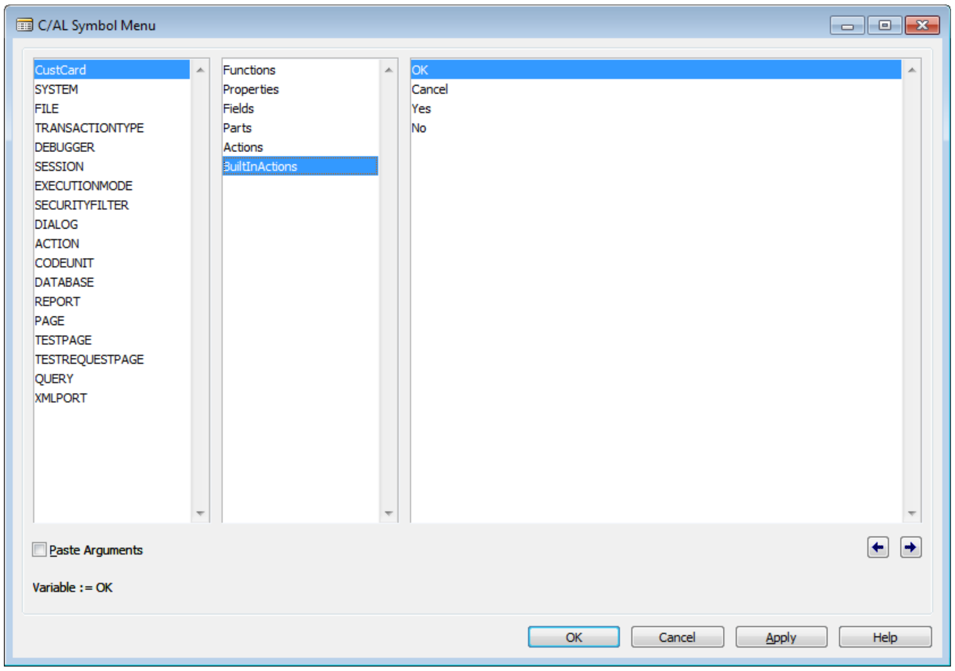

C/AL Symbol Menu
CustCard
SYSTEM
FILE
TRANSACTIONTYPE
DEBUGGER
SESSION
EXECUTIONMODE
SECURITYFILTER
DIALOG
ACTION
CODEUNIT
DATABASE
REPORT
PAGE
TESTPAGE
TESTREQUESTPAGE
QUERY
XMLPORT
Functions
Properties
Fields
Parts
Actions
BuiltInActions
Page Customer Ledger Entries
Page Issued Reminder List
Page Issued Fin. Charge Memo List
Page Comment Sheet
Page Default Dimensions
Page Customer Bank Account List
Page Ship-to Address List
Action14
Page Customer Statistics
Page Customer Stats. by Currencies
Page Customer Entry Statistics
Page Customer Sales
Page Item Cross References
Page Customer Service Contracts
Page Service Items
Page Job List
Action118
Page DO Payment Credit Card List
Page Credit Cards Transaction Log Entries.
Page Cust. Invoice Discounts
Page Sales Prices
Page Sales Line Discounts
Page Sales Prepayment Percentages
Page Standard Customer Sales Codes
Paste Arguments
OK
Cancel
Apply
Help

FIGURE 12.9: ACTIONS FOR A TESTPAGE VARIABLE FOR THE CUSTOMER CARD PAGE

For example, to simulate clicking the Sales Prices action on the **Customer Card** page, write the following code.

```
CustCard.OPENVIEW;

CustCard."Page Sales Prices".INVOKE;
```

To view built-in actions, such as **Yes**, **No**, **OK**, or **Cancel**, select the test page variable in the first column of the symbols, and then select **BuiltInActions** in the second column.



Variable := OK

FIGURE 12.10: BUILT-IN ACTIONS FOR A TESTPAGE VARIABLE FOR THE CUSTOMER CARD PAGE

Note: *The **Symbol Menu** may include built-in actions that are not available on the page. If you call a built-in action that is not available on the page, then the test fails.*

For example, to simulate clicking **OK** after you create a new customer on the **Customer Card** page, you can write the following code.

```
CustomerCard.OPENNEW;

CustomerCard.Name.Value := 'Adventure Works';

CustomerCard.OK.INVOKE;
```

Navigating Among Records

To simulate moving to different items on a list page or moving to different records on a card page, you use one of the following navigation functions:

1. NEXT
2. PREVIOUS
3. FIRST
4. LAST
5. GOTORECORD
6. GOTOKEY

7. FINDFIRSTFIELD
8. FINDNEXTFIELD
9. FINDPREVIOUSFIELD

For example, to simulate showing a specific customer on a TestPage variable for the **Customer Card** page, write the following code.

```
CustCard.OPENVIEW;

CustCard.GOTOKEY('30000');
```

Demonstration: Using a TestPage Variable in a Test Codeunit

The following demonstration shows how to automate testing a page by using a TestPage variable.

Demonstration Steps

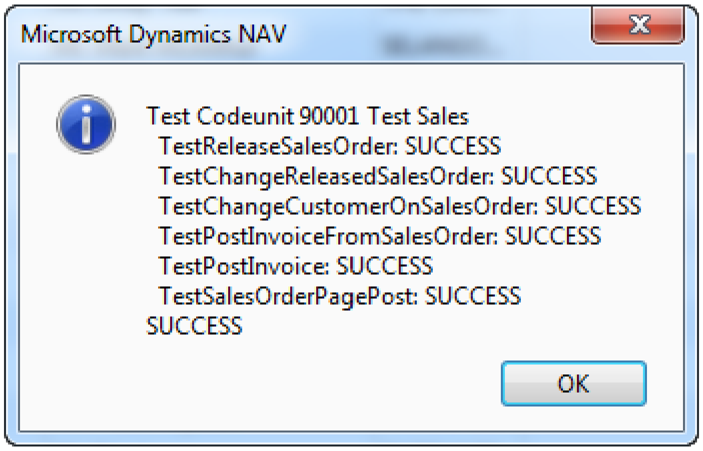
1. Add a function to create and post a sales order through a TestPage variable.
   a. Design the Test Sales codeunit.
   b. Create a new function, name it “TestSalesOrderPagePost”, and set its Handlers property to “HandleSalesPostStrMenu”.
   c. Define the following local variables for the **TestSalesOrderPagePost** function.

Name	DataType	Subtype
SalesHeader	Record	Sales Header
SalesOrder	TestPage	Sales Order

   d. In the TestSalesOrderPagePost function trigger, write the following code.

```
CreateSalesOrder(SalesHeader,'10000','1000',10);

SalesOrder.OPENVIEW;

SalesOrder.GOTOKEY(SalesHeader."Document Type",SalesHeader."No.");

PostingType := PostingType::Ship;

SalesOrder.Post.INVOKE;

PostingType := PostingType::Invoice;

SalesOrder.Post.INVOKE;
```

2. Save the codeunit, run it, and then view the results.
   a. Click **File > Save**. Click **OK** in the **Save** dialog box.
   b. Close the codeunit.
   c. In Object Designer, select the codeunit 90001, TestSales.
   d. Click **Run** to run the test.
   e. After the test finishes, view the log message.



```
Test Codeunit 90001 Test Sales
  TestReleaseSalesOrder: SUCCESS
  TestChangeReleasedSalesOrder: SUCCESS
  TestChangeCustomerOnSalesOrder: SUCCESS
  TestPostInvoiceFromSalesOrder: SUCCESS
  TestPostInvoice: SUCCESS
  TestSalesOrderPagePost: SUCCESS
SUCCESS
```

FIGURE 12.11: TEST SALES RESULTS AFTER A TESTPAGE TEST

Test Runner Codeunits

You use test runner codeunits to manage the execution of test codeunits and to integrate with test management or test reporting frameworks. A test runner codeunit manages the execution of test codeunits that are run from its OnRun trigger.

1. With test runner codeunits you can do the following:
2. Run multiple test codeunits.
3. Intercept each test codeunit or test function before they run.
4. Control which codeunits or test functions to run or to skip.
5. Retrieve the test results of a test function and the whole test codeunit after they run.

To create a test runner codeunit, you set the Subtype property to TestRunner.

Test runner codeunits support two built-in triggers:

6. OnBeforeTestRun
7. OnAfterTestRun

These two triggers are not automatically created when you create a test runner codeunit. You must manually create them if you want to use them. If you do not need either or both of these triggers, you do not have to create them.

When you create these triggers, you must match the following signatures.

Trigger	Signature
OnBeforeTestRun	OnBeforeTestRun(CodeUnitId: Integer;CodeUnitName: Text[30];FunctionName: Text[128]): Boolean
OnAfterTestRun	OnAfterTestRun(CodeUnitId: Integer;CodeUnitName: Text[30];FunctionName: Text[128];Success: Boolean)

OnBeforeTestRun Trigger

The OnBeforeTestRun trigger runs before each test codeunit and test function and enables you to skip running a test codeunit or a test function. The OnBeforeTestRun trigger parameters contain the information about the test codeunit and test function. You control whether to run the current test codeunit or test function through the return value. If you return TRUE, the test codeunit or test function runs. Otherwise, it is skipped.

If the FunctionName parameter is blank, you can control whether to run or skip the whole test codeunit. In this case, returning TRUE runs all test functions, and returning FALSE skips all test functions in the test codeunit.

OnBeforeTestRun always runs in its own database transaction.

You can use the OnBeforeTestRun triggers to perform preprocessing, such as general initialization and logging, or to automate tests by integrating the test runner codeunit with a test management framework.

OnAfterTestRun Trigger

When it is implemented, the OnAfterTestRun trigger is called after each test function runs and after the whole test codeunit runs. The OnAfterTestRun trigger also tells you whether the test codeunit or test function failed or succeeded. If you implement the OnAfterTestRun trigger, then the result logs of the test codeunits that run from the test runner codeunit are not shown.

You can use the OnAfterTestRun trigger to perform post-processing, such as logging, or to automate tests by integrating the test runner codeunit with a test management framework. The OnAfterTestRun trigger is run in its own database transaction.

Demonstration: Creating and Running a Test Runner Codeunit

This demonstration shows how to create a test runner codeunit to control which test codeunits or test functions run during unit testing.

Demonstration Steps

1. Create a new test runner codeunit, and save it as 90002, Test Runner.
    a. In Object Designer, click **Codeunit**, and then click **New**.
    b. Press SHIFT+F4.
    c. Set the Subtype property to TestRunner.
    d. Close the **Properties** window.
    e. Click **File > Save**.
    f. In the **Save As** dialog box, in the **ID** field, type "90002".
    g. In the Name field, type "Test Runner".
    h. Make sure that the Compiled check box is selected, and then click OK.

2. Add the OnBeforeTestRun and OnAfterTestRun triggers to the Test Runner codeunit.
    a. Click **View > C/AL Globals**.
    b. On the **Functions** tab, write "OnBeforeTestRun" and "OnAfterTestRun" on two separate lines.
    c. Select the **OnBeforeTestRun** function, and then click **Locals**.
    d. On the **Parameters** tab, enter the following information.

Name	DataType	Length
CodeUnitId	Integer	
CodeUnitName	Text	30
FunctionName	Text	128

    e. On the **Return Value** tab, set Return Type to Boolean.
    f. Close the **C/AL Locals** window.
    g. Select the **OnAfterTestRun** function, and then click **Locals**.
    h. On the **Parameters** tab, enter the following information.

Name	DataType	Length
CodeUnitId	Integer	
CodeUnitName	Text	30
FunctionName	Text	128
Success	Boolean	

   i. Close the **C/AL Locals** window.

3. Define global variables to keep track of succeeded, failed, and skipped tests, and a text constant to show the results.
   a. On the **Variables** tab of the **C/AL Globals** window, enter three Integer variables, and name them Succeeded, Failed, and Skipped.
   b. On the **Text Constants** tab, create the Text001 constant with the following value: “Testing summary:\\Succeeded: %1\Failed: %2\Skipped: %3”.
   c. Close the **C/AL Globals** window.

4. Write code to run the Test Sales codeunit, and to show the results.
   a. In the OnRun trigger, write the following code.

```
CODEUNIT.RUN(CODEUNIT::"Test Sales");

MESSAGE(Text001,Succeeded,Failed,Skipped);
```

5. Write code to skip any test functions where name includes the word “Invoice”, and to run everything else.
   a. In the **OnBeforeTestRun** function trigger, write the following code.

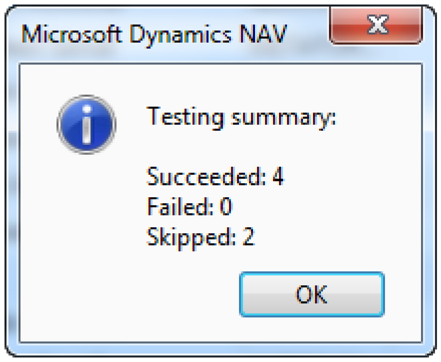
```
IF (FunctionName <> '') AND (STRPOS(FunctionName,'Invoice') <> 0)

THEN BEGIN

  Skipped := Skipped + 1;

  EXIT(FALSE);

END;

EXIT(TRUE);
```

6. Write code to count the number of succeeded and failed test function runs.
   a. In the OnAfterTestRun function trigger, write the following code.

```
IF FunctionName = '' THEN

  EXIT;

IF Success THEN

  Succeeded := Succeeded + 1
```

```
ELSE

  Failed := Failed + 1;
```

7. Save, close, and run the codeunit, and then view the results.
   a. Click **File** > **Save**.
   b. Confirm the **Save** dialog box.
   c. Close the codeunit.
   d. In Object Designer, select the Test Runner codeunit.
   e. Click **Run**.
   f. The message box shows the summary of the results.



```
Microsoft Dynamics NAV

Testing summary:

Succeeded: 4
Failed: 0
Skipped: 2
```

FIGURE 12.12: TEST RUNNER RESULTS

Note: *The default log message box that you view when you directly run a test codeunit is suppressed because you implemented the OnAfterTestRun trigger.*

Testing Seminar Management

An important part of every development process is development of unit test scripts. A comprehensive set of features in Microsoft Dynamics NAV 2013 enables comprehensive testing automation of almost every aspect of the application. This includes all C/AL code and most of the user interface.

Whether you follow a Test-Driven Development approach and follow the red-green-refactor paradigm, or you first develop your application and then write test scripts, your solution should always include a unit test framework that consists of test and test runner codeunits.

Solution Design

Most customers do not explicitly specify the kinds of unit tests that you have to run to guarantee highest code quality and lowest bug rates. However, all customers require a high-quality solution. This maintains the integrity of the standard application and does not disrupt daily use due to frequent bugs or errors.

CRONUS International Ltd. did not specify any testing requirements, but you must still make sure that the solution that you deliver is as bug-free as possible. You also must avoid regression issues in the future, and provide automated means to pinpoint any bugs that result from rework or an upgrade to a future version of Microsoft Dynamics NAV 2013.

Your design decision is to implement a test framework that provides the following functionality:

1. Developers can easily configure which test codeunits to add to or remove from the framework.
2. Developers can easily select which test codeunits and test functions to run or skip.
3. During each test run, only the selected test codeunits and test functions must run.
4. For each test run, the framework must keep a history that contains the following information:
    a. Date and time of the test run
    b. ID of the user who ran the test

5. For each test function, the framework must keep a history with the following information:
    a. Date and time of the test run
    b. Indicator of test success or failure
    c. Error message in the event of failure

6. For each test codeunit and test function, the framework must provide statistics with the following information broken down by last run, previous run, and all previous runs:
    a. Count of successful tests
    b. Count of failed tests
    c. Total count of tests
    d. Success ratio

All developers who are working on the implementation project for CRONUS International Ltd. must provide unit tests for all functionality that they developed. In the future, developers must follow the Test-Driven Development practices to first develop the unit tests and add them to the framework, and then to develop the functionality. For easier maintenance and management of test runs, there must be one test codeunit per functional area.

The test framework will use the test runner codeunit functionality of Microsoft Dynamics NAV 2013. The test runner codeunit will read the list of test codeunits from the setup table and run each test codeunit that is selected for testing.

Through the OnBeforeTestRun trigger, the test runner codeunit will decide whether to run a specific test function based on the information in the setup table.

The OnAfterTestRun trigger will log the success or failure information about test runs into the history tables.

Solution Development

To meet the design goals, you first have to develop the test framework, and then develop the test codeunits for application functionality.

Test Framework

The test framework consists of the following objects.

Object Type	Object ID	Object Name	Remarks
Table	123456751	Seminar Unit Test Setup	Contains the list of test codeunits, test functions, and a check box to specify whether a specific test codeunit or test function is included in test runs. When a user inserts a new row by specifying the **Codeunit ID**, the list of functions from that test codeunit must automatically be inserted into the **Seminar Unit Test Setup** table. Users can delete rows for test codeunits only, but not test functions. These are the rows where **Function Name** is empty. When a user deletes a row, all test function rows for that test codeunit must be deleted automatically.
Table	123456752	Seminar Unit Test Register	Contains the history of test runs. A *test run* is the event when a user starts all selected tests that are configured in the **Seminar Unit Test Setup** table. Each line in the **Seminar Unit Test Register** table corresponds to one test run.
Table	123456753	Seminar Unit Test Entry	Contains the success or failure history of test functions. Each line in the **Seminar Unit Test Entry** corresponds to one test run of a test function.

Object Type	Object ID	Object Name	Remarks
Page	123456751	Seminar Unit Test Setup	Editable list page over the **Seminar Unit Test Setup** table that lets users configure which test codeunits and test functions are included in test runs. It also lets users run tests and access the test history and statistics of a specific test codeunit or test function. The only editable fields are **Codeunit ID** and **Run**.
Page	123456752	Seminar Unit Test Register	Noneditable list page over the **Seminar Unit Test Register** table. It lets users access the details of a specific test run.
Page	123456753	Seminar Unit Test Entries	Noneditable list page over the **Seminar Unit Test Entry** table.
Page	123456754	Seminar Unit Test Statistics	Noneditable card page that follows all design rules of statistics pages.

Object Type	Object ID	Object Name	Remarks
Codeunit	123456751	Seminar Unit Test Runner	Test runner codeunit that reads information from the **Seminar Unit Test Setup** table, runs the selected test codeunits and test functions, and logs the test history into the **Seminar Unit Test Register** and **Seminar Unit Test Entry** tables. The codeunit must include a setup mode in which it runs a single test codeunit. Instead of running the tests, the codeunit skips all test functions and inserts a row into the **Seminar Unit Test Setup** table for each test function that is present in the test codeunit. This test mode is run from the OnInsert trigger of the **Seminar Unit Test Setup** table.

Specific Test Codeunits

For each functional area of the Seminar Management application area, you must provide one test codeunit. This means that there should be the following test codeunits.

Test Codeunit	Unit Test Examples
123456752 Seminar Master Data Tests	• Test the **Seminar Setup Card** page to make sure that users can look up number series and cannot enter invalid number series. • Test the **Seminar Card** page to make sure that number series functionality is integrated and works as expected by Microsoft Dynamics NAV 2013 standards. • Test the **Seminar Card** page to make sure that the **VAT Prod. Posting Group** field is updated when **Gen. Prod. Posting Group** field is modified. • Test the **Seminar List** and **Seminar Card** pages to make sure that the Ledger Entries action shows the correct ledger entries for the selected seminar.
123456753 Seminar Registration Tests	• Test that creating a new seminar registration from the **Seminar List** creates a new seminar registration for the selected seminar, and that seminar related fields are copied from the Seminar record into the **Seminar Registration** page. • Test that when a room is selected, the room-related fields are copied from the Resource record into the **Seminar Registration** page. • Test to specify that when a room allows fewer participants than is specified in the Seminar Registration Header record, the user has to confirm that room. • Test that only canceled seminar registrations can be deleted.

Test Codeunit	Unit Test Examples
123456754 Seminar Posting Tests	• Test that posting succeeds only for closed seminar registrations. • Test that posting succeeds only for seminars that have registered participants. • Test that posting generates a register record with correct information for the user, date, and source code.
123456756 Seminar Reporting Tests	• Test that clicking **Print** on the **Seminar Registration** page runs the Seminar Reg.-Participant List report.
123456757 Seminar Statistics Tests	• Test that clicking **Statistics** on the **Seminar Card** and **Seminar List** pages shows the **Seminar Statistics** page that has the correct seminar selected.
123456758 Seminar Dimensions Tests	• Test that global dimensions that are set to the Seminar record are available in the **Default Dimensions** page that is invoked from the Dimensions action on the **Seminar Card** page. • Test that dimensions that are set to Seminar Registration through the Dimensions action update the **Shortcut Dimension 1 Code** and **Shortcut Dimension 2 Code** fields.

Note: *The list of the unit tests for each test codeunit is not comprehensive, and you can add more tests. In a real-world project, there would be many more tests for each functional area.*

Lab 12.1: Create Seminar Management Unit Tests

Scenario

Because of how much work is related to the development of the Seminar Management unit test framework and unit tests, Simon, the development manager on the implementation project for CRONUS International Ltd., has split the work tasks. Simon will develop the test framework, and you will develop the individual unit tests.

Exercise 1: Import the Testing Framework

Exercise Scenario

Simon developed the test framework and gave you the objects. You now import the Seminar Management test framework from the.fob file that was provided.

Task 1: Import the Objects

High Level Steps

1. Import the Lab 12.1 - Starter.fob object file into the Microsoft Dynamics NAV 2013 Development Environment.

Detailed Steps

1. Import the Lab 12.1 - Starter.fob object file into the Microsoft Dynamics NAV 2013 Development Environment.
   a. In Object Designer, click **File > Import**.
   b. Browse to the Lab 12.A - Starter.fob file, and then click **Open**.
   c. In the dialog box stating that there were no conflicts, click **Yes** to complete the import.

Exercise 2: Create the Unit Tests

Exercise Scenario

You develop a test codeunit to contain all unit tests for Seminar Management master data.

Task 1: Master Data Unit Tests

High Level Steps

1. Create the Seminar Master Data Tests codeunit.
2. Create a test function to test that the **AssistEdit(...)** button on the **No.** field on the **Seminar Card** page runs the standard **No. Series** functionality.

3. Create a modal page handler function for the **No. Series List** page that simulates clicking **OK**, and then attach this handler to the **TestSeminarCardNoSeries** function.
4. Include the Seminar Master Data Tests codeunit in the test framework configuration.

Detailed Steps

1. Create the Seminar Master Data Tests codeunit.
   a. In Object Designer, click **Codeunit**, and then click **New**.
   b. Set the properties on the codeunit to make it a test codeunit.
   c. Save the codeunit as 123456752, Seminar Master Data Tests.

2. Create a test function to test that the **AssistEdit(...)** button on the **No.** field on the **Seminar Card** page runs the standard **No. Series** functionality.
   a. Create a test function and name it "TestSeminarCardNoSeries".
   b. Declare the following local variables for the **TestSeminarCardNoSeries** function.

Name	DataType	Subtype
SeminarCard	TestPage	Seminar Card

   c. In the TestSeminarCardNoSeries function trigger, write the following code.

```
SeminarCard.OPENNEW;

SeminarCard."No.".ASSISTEDIT;
```

3. Create a modal page handler function for the **No. Series List** page that simulates clicking **OK**, and then attach this handler to the **TestSeminarCardNoSeries** function.
   a. Create a new function, and name it "NoSeriesListHandler".
   b. Set the properties on the function to make it a modal page handler.
   c. Define the following parameter for the **NoSeriesListHandler** function.

Var	Name	DataType	Subtype
Yes	NoSeriesList	TestPage	No. Series List

   d. In the NoSeriesListHandler function trigger, write the following code.

```
NoSeriesList.OK.INVOKE;
```

   e. Set the HandlerFunctions property for the **TestSeminarCardNoSeries** function to “NoSeriesListHandler”.
   f. Save, and then close the codeunit.

4. Include the Seminar Master Data Tests codeunit in the test framework configuration.
   a. Run page 123456751, **Seminar Unit Test Setup**.
   b. Click **New**.
   c. In the **Codeunit ID** field, type “123456752”. The **TestSeminarCardNoSeries** function is added to the list automatically.
   d. Click **Close**.

Task 2: Seminar Registration Unit Tests

High Level Steps

1. Create the Seminar Registration Tests codeunit.
2. Create a test function to test the confirmation of the change of maximum number of participants, when the selected room accommodates less than was defined for the seminar.
3. Create a confirm handler that confirms the question, and then attach it to the **TestSeminarRegistrationRoomMaxParticipants** function.
4. Include the Seminar Registration Tests codeunit in the test framework configuration.

Detailed Steps

1. Create the Seminar Registration Tests codeunit.
   a. In Object Designer, click **Codeunit**, and then click **New**.
   b. Set the properties on the codeunit to make it a test codeunit.
   c. Save the codeunit as 123456753, Seminar Registration Tests.

2. Create a test function to test the confirmation of the change of maximum number of participants, when the selected room accommodates less than was defined for the seminar.
   a. Create a new test function and name it “TestSeminarRegistrationRoomMaxParticipants**”**.
   b. Define the following local variables for the **TestSeminarRegistrationRoomMaxParticipants** function.

Name	DataType	Subtype
Seminar	Record	Seminar
Resource	Record	Resource
SeminarRegistration	TestPage	Seminar Registration

c. In the function trigger for the TestSeminarRegistrationRoomMaxParticipants function, write the following code.

```
Seminar.FINDFIRST;

SeminarRegistration.OPENNEW;

SeminarRegistration."Seminar No.".SETVALUE(Seminar."No.");

Resource.SETRANGE(Type,Resource.Type::Machine);

Resource.SETFILTER("Maximum Participants",'<>0');

Resource.FINDFIRST;

SeminarRegistration."Maximum Participants".SETVALUE(Resource."Maximum
Participants" + 1);

SeminarRegistration."Room Resource No.".SETVALUE(Resource."No.");

SeminarRegistration."Maximum Participants".ASSERTEQUALS(Resource."Maximum
Participants");

SeminarRegistration.OK.INVOKE;
```

3. Create a confirm handler that confirms the question, and then attach it to the **TestSeminarRegistrationRoomMaxParticipants** function.
   a. Create a new function, and name it “ConfirmMaxParticipants**”**.
   b. Set the properties for the **ConfirmMaxParticipants** function to make it a confirm handler.
   c. Define the following parameters for the **ConfirmMaxParticipants** function.

Var	Name	DataType	Length
No	Question	Text	1024
Yes	Reply	Boolean	

   d. In the ConfirmMaxParticipants function trigger, write the following code:

```
Reply := TRUE;
```

   e. Set the HandlerFunctions property of the **TestSeminarRegistrationRoomMaxParticipants** function to “ConfirmMaxParticipants”.
   f. Save and close the codeunit.

4. Include the Seminar Registration Tests codeunit in the test framework configuration.
    a. Run page 123456751, **Seminar Unit Test Setup**.
    b. Click **New**.
    c. In the **Codeunit ID** field, type "123456753".
    d. Click **Close**.

Task 3: Dimension Integration Unit Tests

High Level Steps

1. Create the Seminar Dimensions Tests codeunit.
2. Create a function to test that global dimensions set through the Dimensions action of the **Seminar Registration** page are correctly copied to the **Shortcut Dimension 1 Code** and **Shortcut Dimension 2 Code** fields.
3. Create a new modal page handler function for the **Edit Dimension Set Entries** page to simulate entering values for two global dimensions for a seminar registration.
4. Include the Seminar Dimensions Tests codeunit in the test framework configuration.

Detailed Steps

1. Create the Seminar Dimensions Tests codeunit.
    a. In Object Designer, click **Codeunit**, and then click **New**.
    b. Set the properties on the codeunit to make it a test codeunit.
    c. Define the following global variables for the function.

Name	DataType	Subtype
GLSetup	Record	General Ledger Setup
DimVal1	Record	Dimension Value
DimVal2	Record	Dimension Value

    d. Save the codeunit as 123456758, Seminar Dimensions Tests.

2. Create a function to test that global dimensions set through the Dimensions action of the **Seminar Registration** page are correctly copied to the **Shortcut Dimension 1 Code** and **Shortcut Dimension 2 Code** fields.
    a. Create a new function and name it "TestShortcutDimensionsRegistration**"**.

b. Define the following local variables for the **TestShortcutDimensionsRegistration** function.

Name	DataType	Subtype
SeminarRegistration	TestPage	Seminar Registration
Seminar	Record	Seminar
SemReg	Record	Seminar Registration Header

c. In the TestShortcutDimensionsRegistration function trigger, write the following code.

```
Seminar.FINDFIRST;

SeminarRegistration.OPENNEW;

SeminarRegistration."Seminar No.".SETVALUE(Seminar."No.");

GLSetup.GET;

GLSetup.TESTFIELD("Global Dimension 1 Code");

GLSetup.TESTFIELD("Global Dimension 2 Code");

SeminarRegistration.Action43.INVOKE;

SemReg.GET(SeminarRegistration."No.".VALUE);

SemReg.TESTFIELD("Shortcut Dimension 1 Code",DimVal1.Code);

SemReg.TESTFIELD("Shortcut Dimension 2 Code",DimVal2.Code);
```

Note: *Action43 in this code example refers to the Dimension action on the* ***Seminar Registration*** *page. Its ID may differ, depending on the exact steps you made during developing the page. If this code example does not compile because Action43 is unknown, then design the* ***Seminar Registration*** *page, find the* ***Dimensions*** *action, and then note its ID. Then use that ID in this code to refer to that action.*

3. Create a new modal page handler function for the **Edit Dimension Set Entries** page to simulate entering values for two global dimensions for a seminar registration.
   a. Create a new function and name it "EditDimSetHandler**"**.
   b. Set properties on the **EditDimSetHandler** function to make it a modal page handler.
   c. Define the following parameters for the **EditDimSetHandler** function.

Var	Name	DataType	Subtype
Yes	EditDimSet	TestPage	Edit Dimension Set Entries

d. In the EditDimSetHandler function trigger, write the following code.

```
DimVal1.SETRANGE("Dimension Code",GLSetup."Global Dimension 1 Code");

DimVal1.FINDFIRST;

EditDimSet.NEW;

EditDimSet."Dimension Code".SETVALUE(GLSetup."Global Dimension 1 Code");

EditDimSet.DimensionValueCode.SETVALUE(DimVal1.Code);

DimVal2.SETRANGE("Dimension Code",GLSetup."Global Dimension 2 Code");

DimVal2.FINDFIRST;

EditDimSet.NEW;

EditDimSet."Dimension Code".SETVALUE(GLSetup."Global Dimension 2 Code");

EditDimSet.DimensionValueCode.SETVALUE(DimVal2.Code);

EditDimSet.OK.INVOKE;
```

e. Set the HandlerFunctions property for the **TestShortcutDimensionsRegistration** function to “EditDimSetHandler”.
f. Save, and then close the codeunit.

4. Include the Seminar Dimensions Tests codeunit in the test framework configuration.
   a. Run page 123456751, **Seminar Unit Test Setup**.
   b. Click **New**.
   c. In the **Codeunit ID** field, type “123456758”.
   d. Click **Close**.

Exercise 3: Run Unit Tests

Exercise Scenario

You now run the unit tests through the Seminar Management unit test framework, and then check the unit test history to verify that the tests have run successfully.

Task 1: Run the Tests

High Level Steps

1. Use the test framework to run unit tests.

Detailed Steps

1. Use the test framework to run unit tests.
    a. Run page 123456751, **Seminar Unit Test Setup**.
    b. Click **Run Tests**. The tests run automatically and the progress is shown in a dialog box.
    c. After the progress dialog box closes, click **Test Entries** to verify that tests have generated results.
    d. Close the **Test Entries** page.
    e. Click **Run Tests** two more times to generate more test results.

Task 2: Check Test Statistics

High Level Steps

1. Show statistics for a test codeunit.

Detailed Steps

1. Show statistics for a test codeunit.
    a. In the **Seminar Unit Test Setup** page, select the row for the Seminar Master Data Tests codeunit.
    b. Click **Statistics**.
    c. Verify that the statistics show the number of succeeded and failed tests over all previous tests.
    d. Close **Statistics**.

Task 3: Check the Test Register

High Level Steps

1. Show the unit test register.
2. Show test entries for a test run.

Detailed Steps

1. Show the unit test register.
    a. Run the page 123456752, **Seminar Unit Test Register**.
    b. Verify that there are three register lines that contain date, time, and user ID for each test run.
2. Show test entries for a test run.
    a. Select the first row in the **Seminar Unit Test Register** page.
    b. Click **Test Entries** to show the **Seminar Unit Test Entries** page.
    c. Verify that the entries shown match the entries specified in the **From Entry No.** and **To Entry No.** fields of the **Seminar Unit Test Register** page.
    d. Close the **Seminar Unit Test Entries** and **Seminar Unit Test Register** pages.

Debugging

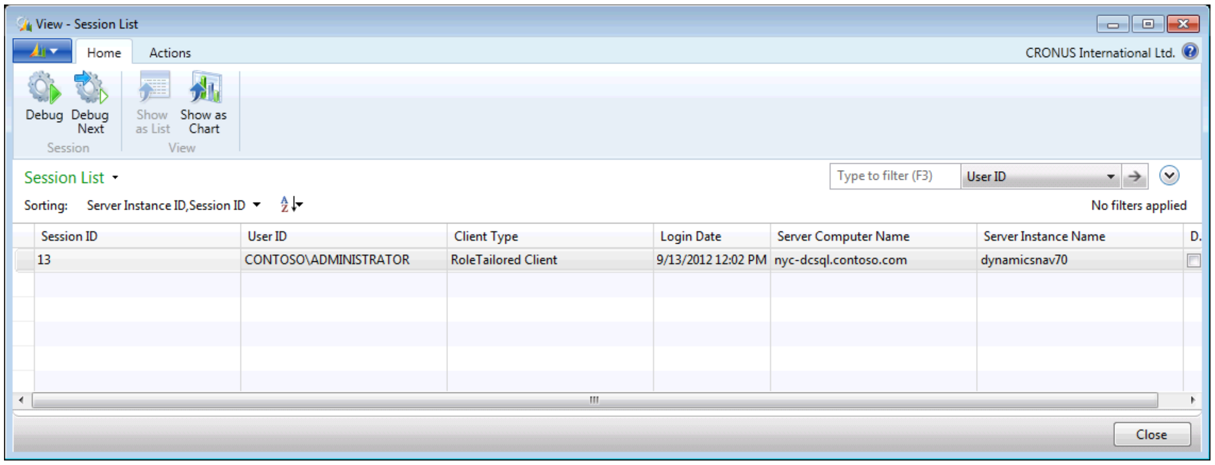
The process of finding and correcting errors is called *debugging*. Microsoft Dynamics NAV 2013 provides an integrated debugger to help you inspect your code to verify that your application runs as expected. The debugger user interface (UI) runs in the Microsoft Dynamics NAV 2013 client for Windows. The debugger services run in the Microsoft Dynamics NAV Server.

Activating the Debugger

When you *activate* the debugger, you start it. When you start the debugger, it can be in one of the following states:

- Attached to a session.
- Waiting to attach to a session.

To start the debugger in the Microsoft Dynamics NAV 2013 Development Environment, in the **Tools** menu, click **Debugger > Debug Session**. This opens the **Session List** window that shows all debuggable sessions that currently run on the same Microsoft Dynamics NAV 2013 server instances on the local machine instance as the debugger . This is the same instance that is targeted when you use the Run action on Application objects in the Object Designer window.



Session ID	User ID	Client Type	Login Date	Server Computer Name	Server Instance Name	D.
13	CONTOSO\ADMINISTRATOR	RoleTailored Client	9/13/2012 12:02 PM	nyc-dcsql.contoso.com	dynamicsnav70	☐

FIGURE 12.13: SESSION LIST

In the **Session List** page, you attach the debugger to a session by doing one of the following:

- Select a specific session, andthen click **Debug**.
- Click **Debug Next**, and then start a new session.

Note: *Selecting* ***Debug Next Session*** *is useful if you want to debug web services. A web service call exists as a session only during the web service call. This typically is not long enough for you to select the specific session in the* ***Session List*** *page.*

Debugger Page

After you start a debugger, the **Debugger** page opens. You use the **Debugger** page to manage the debug process as follows:

- Step through the code.
- Manage the code execution.
- Manage the breakpoints.
- View the variables in scope of the current line.
- View the last error message.
- Manage watches.
- View the call stack.

The "Debugger" figure shows the **Debugger** page.

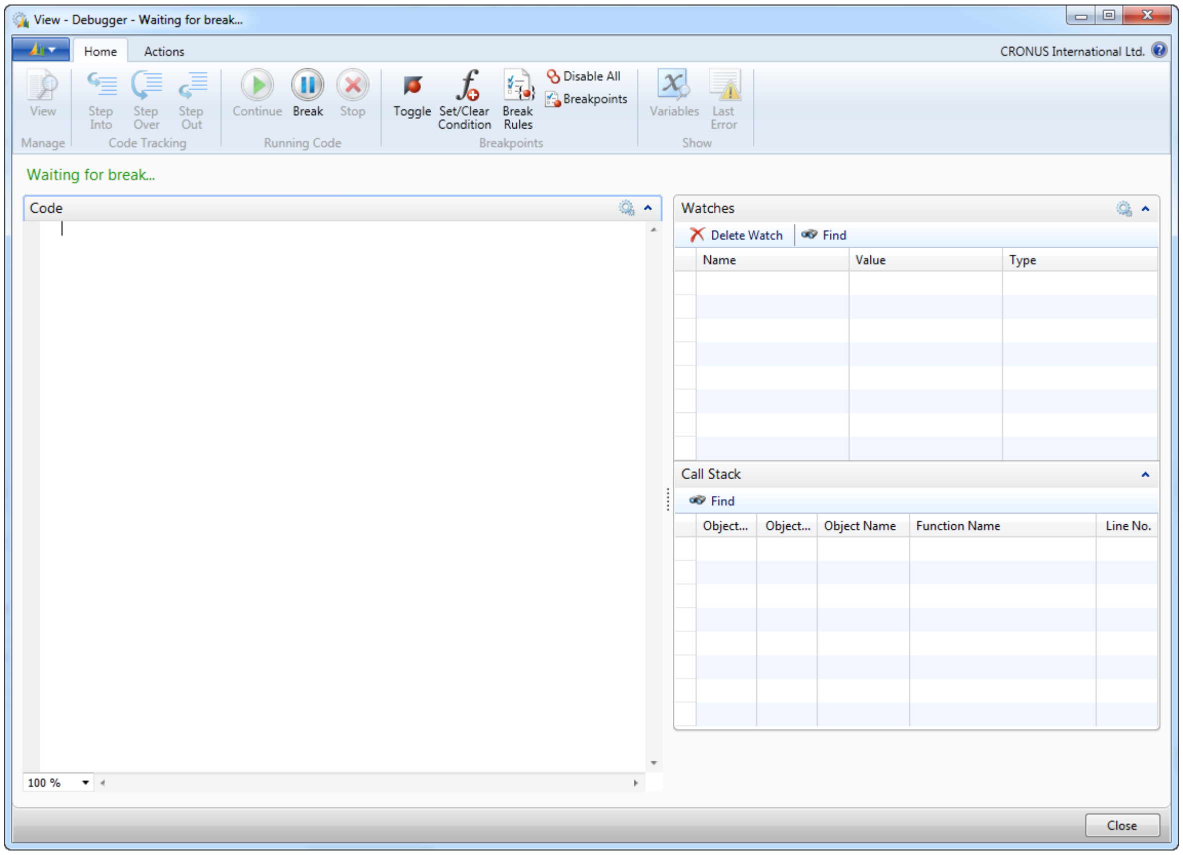


FIGURE 12.14: DEBUGGER

Breakpoints

You can break code execution of the session that you are debugging by doing the following:

- Setting a breakpoint on a line of code.
- Specifying a break on the next statement.
- Specifying a break on errors.
- Specifying a break on record changes.

You can set breakpoints before you start a debugging session or when you are debugging. Breakpoints and break rules are applied immediately in the session to which the debugger is attached.

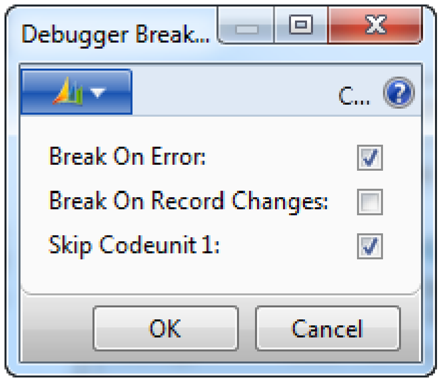
Break Rules

The debugger usually stops on breakpoints. However, you can enable other break rules that enable the debugger to do the following:

- Stop on error.
- Break on record changes.
- Skip any breaks in Codeunit 1.

To define these additional rules in the **Debugger** page, click **Break Rules**. It opens the **Debugger Break Rules** dialog box.

The "Debugger Break Rules" figure shows the Debugger Break Rules dialog box.



Break On Error: ✓
Break On Record Changes: ☐
Skip Codeunit 1: ✓

FIGURE 12.15: DEBUGGER BREAK RULES

Breakpoints in Code

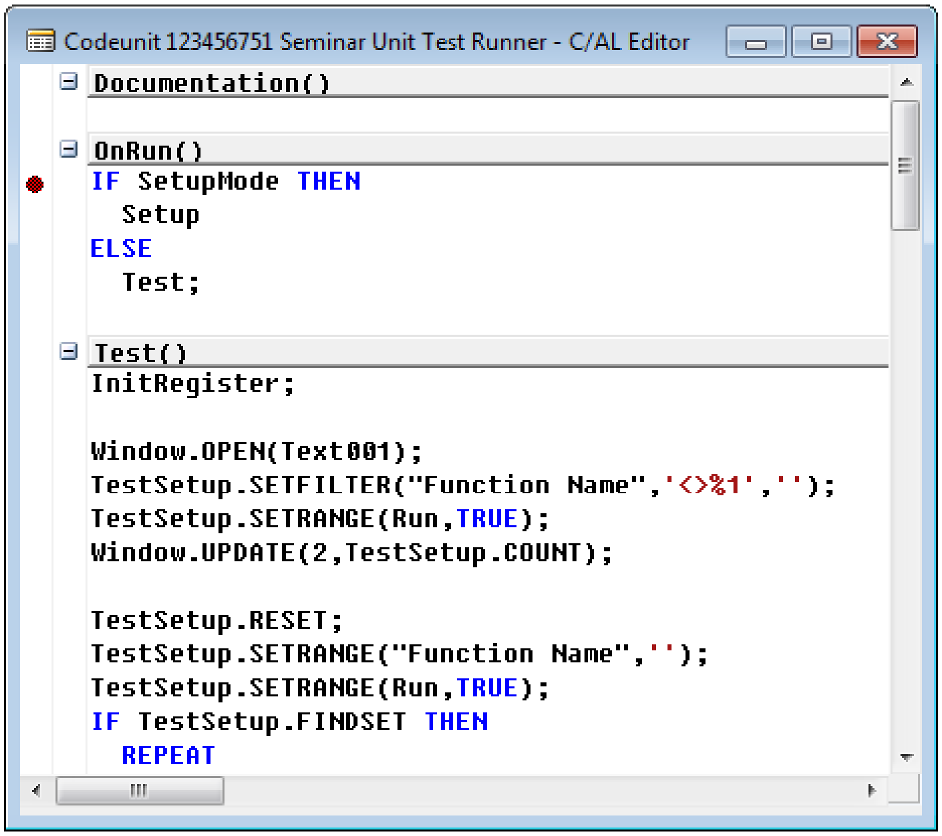
If you set a breakpoint on a line of C/AL code, then execution breaks before the first statement on the line executes. If you set a breakpoint on a line of code that does not have a C/AL statement, then the breakpoint is automatically set on the next statement.

You can set a breakpoint on a code line in the **C/AL Editor** window or the **Debugger** page by positioning the cursor in the line where you want to set the breakpoint, and then pressing F9.

Note: *As an alternative, in the* ***Tools*** *menu, click* ***Debugger*** *>* ***Toggle Breakpoint*** *in the* ***C/AL Editor*** *window, or click* ***Toggle*** *in the* ***Debugger*** *page.*

You can enable or disable a breakpoint. The debugger only stops on enabled breakpoints.

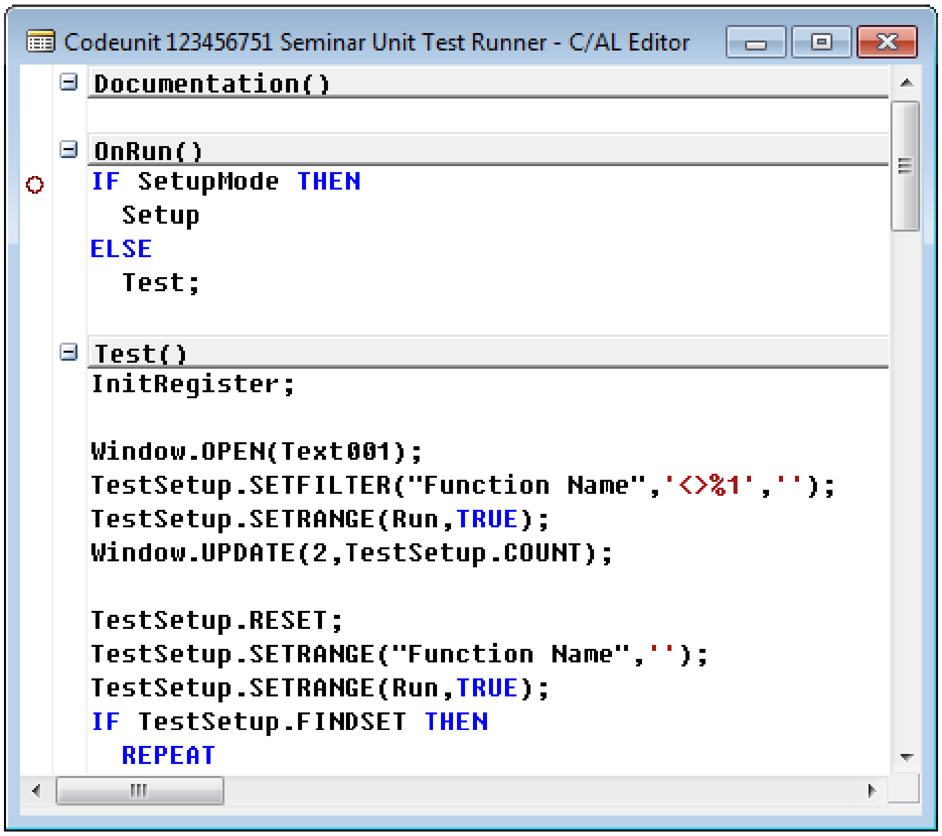
The "Enabled Breakpoint" figure shows how the enabled breakpoint looks in the **C/AL Editor** window.



```
Documentation()

OnRun()
IF SetupMode THEN
  Setup
ELSE
  Test;

Test()
InitRegister;

Window.OPEN(Text001);
TestSetup.SETFILTER("Function Name",'<>%1','');
TestSetup.SETRANGE(Run,TRUE);
Window.UPDATE(2,TestSetup.COUNT);

TestSetup.RESET;
TestSetup.SETRANGE("Function Name",'');
TestSetup.SETRANGE(Run,TRUE);
IF TestSetup.FINDSET THEN
  REPEAT
```

FIGURE 12.16: ENABLED BREAKPOINT

The "Disabled Breakpoint" figure shows the disabled breakpoint in the **C/AL Editor** window.



```
Documentation()

OnRun()
IF SetupMode THEN
  Setup
ELSE
  Test;

Test()
InitRegister;

Window.OPEN(Text001);
TestSetup.SETFILTER("Function Name",'<>%1','');
TestSetup.SETRANGE(Run,TRUE);
Window.UPDATE(2,TestSetup.COUNT);

TestSetup.RESET;
TestSetup.SETRANGE("Function Name",'');
TestSetup.SETRANGE(Run,TRUE);
IF TestSetup.FINDSET THEN
  REPEAT
```

FIGURE 12.17: DISABLED BREAKPOINT

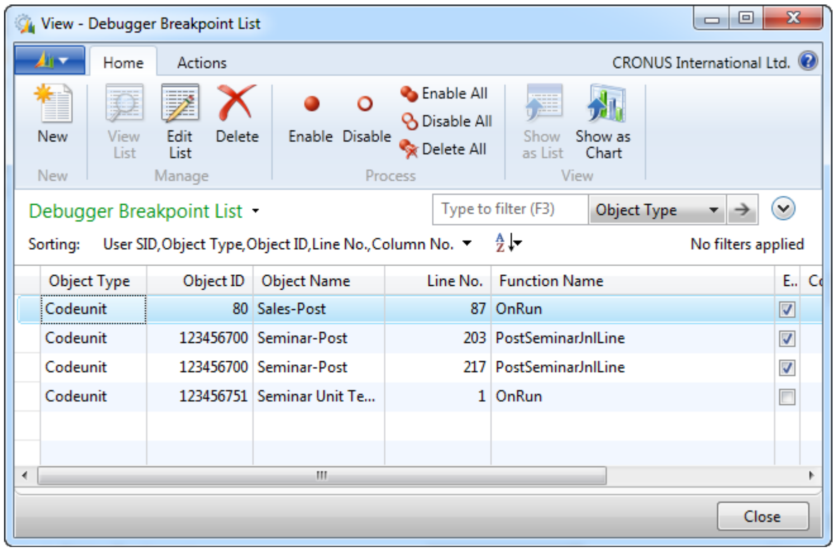
Note: *You disable breakpoints when you do not want the debugger to stop at specific points in the code, but may later want to debug those points. Then instead of searching through the code and creating a new breakpoint, you can enable a disabled breakpoint from the list of all breakpoints.*

Breakpoints Overview

You manage all breakpoints from the **Debugger Breakpoint List** page. To access it, in the **Debugger** page, click **Breakpoints**. The **Debugger Breakpoint List** page enables you to do the following:

- Delete a specific breakpoint.
- Delete all breakpoints.
- Enable or disable a specific breakpoint.
- Enable or disable all breakpoints.
- Create a new breakpoint by specifying the object type, ID, and line number manually.

The “Debugger Breakpoint List” image shows the **Debugger Breakpoint List** page.



Object Type	Object ID	Object Name	Line No.	Function Name	E..
Codeunit	80	Sales-Post	87	OnRun	☑
Codeunit	123456700	Seminar-Post	203	PostSeminarJnlLine	☑
Codeunit	123456700	Seminar-Post	217	PostSeminarJnlLine	☑
Codeunit	123456751	Seminar Unit Te...	1	OnRun	☐

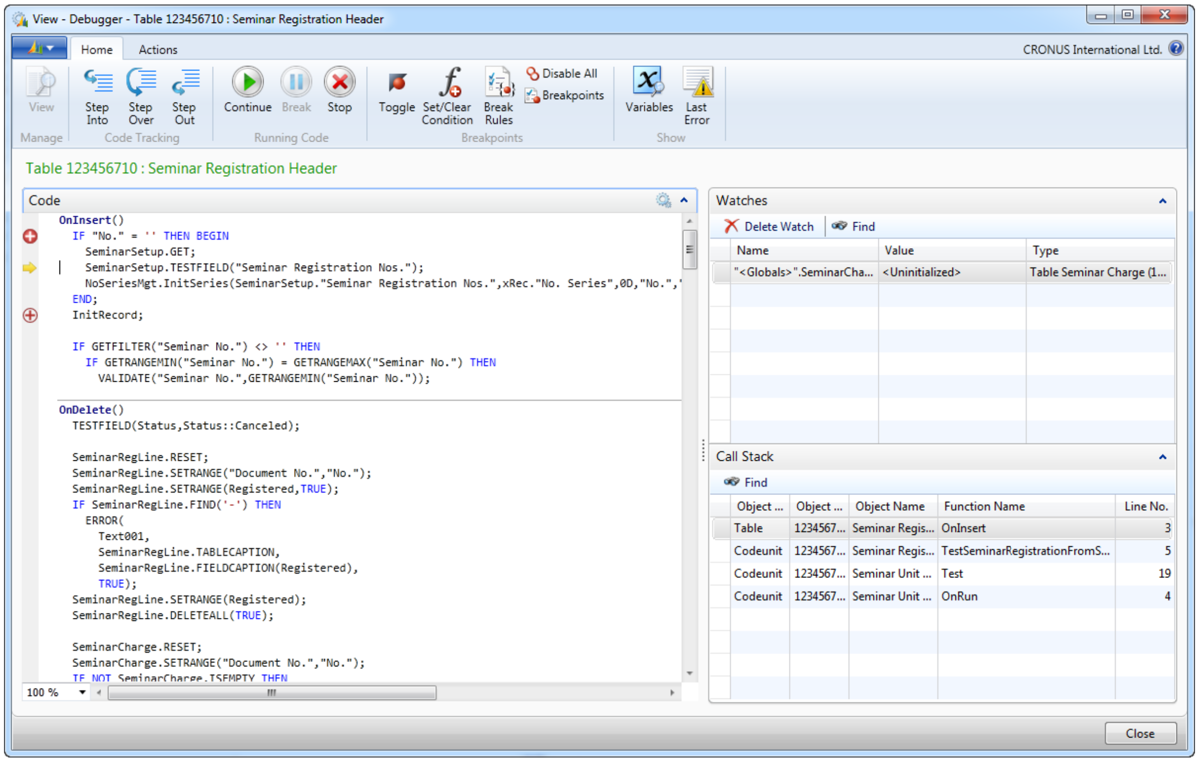
FIGURE 12.18: DEBUGGER BREAKPOINT LIST

Conditional Breakpoints

Most breakpoints break the code execution unconditionally. Sometimes you want to break the execution only if certain conditions are met, but execute the code without breaking. To set a condition on a breakpoint, when the execution stops on the breakpoint, in the **Debugger** page click **Set/Clear Condition**. This shows the **Debugger Breakpoint Condition** dialog box where you can enter a Boolean expression. This may include any of the variables in scope of the breakpoint line. When the code execution reaches the breakpoint, the debugger first evaluates the condition expression. Then, if it evaluates to TRUE, the debugger breaks. If it evaluates to FALSE, it continues execution without breaking.

Note: *You can disable a conditional breakpoint. When you do this, the condition remains defined on the breakpoint. It will apply after you enable the breakpoint again.*

An enabled conditional breakpoint is shown with a white plus (**+)** sign in the red breakpoint circle. A disabled conditional breakpoint is shown with a red plus (+) sign in the white breakpoint circle.



```
OnInsert()
  IF "No." = '' THEN BEGIN
    SeminarSetup.GET;
    SeminarSetup.TESTFIELD("Seminar Registration Nos.");
    NoSeriesMgt.InitSeries(SeminarSetup."Seminar Registration Nos.",xRec."No. Series",0D,"No.",'
  END;
  InitRecord;

  IF GETFILTER("Seminar No.") <> '' THEN
    IF GETRANGEMIN("Seminar No.") = GETRANGEMAX("Seminar No.") THEN
      VALIDATE("Seminar No.",GETRANGEMIN("Seminar No."));

OnDelete()
  TESTFIELD(Status,Status::Canceled);

  SeminarRegLine.RESET;
  SeminarRegLine.SETRANGE("Document No.","No.");
  SeminarRegLine.SETRANGE(Registered,TRUE);
  IF SeminarRegLine.FIND('-') THEN
    ERROR(
      Text001,
      SeminarRegLine.TABLECAPTION,
      SeminarRegLine.FIELDCAPTION(Registered),
      TRUE);
  SeminarRegLine.SETRANGE(Registered);
  SeminarRegLine.DELETEALL(TRUE);

  SeminarCharge.RESET;
  SeminarCharge.SETRANGE("Document No.","No.");
  IF NOT SeminarCharge.ISEMPTY THEN
```

Name	Value	Type
"<Globals>".SeminarCha...	<Uninitialized>	Table Seminar Charge (1...

Object ...	Object ...	Object Name	Function Name	Line No.
Table	1234567...	Seminar Regis...	OnInsert	3
Codeunit	1234567...	Seminar Regis...	TestSeminarRegistrationFromS...	5
Codeunit	1234567...	Seminar Unit ...	Test	19
Codeunit	1234567...	Seminar Unit ...	OnRun	4

FIGURE 12.19: CONDITIONAL BREAKPOINTS

Note: *Conditional breakpoints are only clearly distinct in the* ***Debugger*** *page. In the* ***C/AL Editor*** *window where they are displayed as regular breakpoints, you cannot see the difference between unconditional and conditional breakpoints.*

Code Tracking

After a breakpoint is reached, you use the debugger to execute C/AL code one line at a time. This procedure is called *stepping*. The Code Tracking group on the **Home** tab provides the following three actions for stepping:

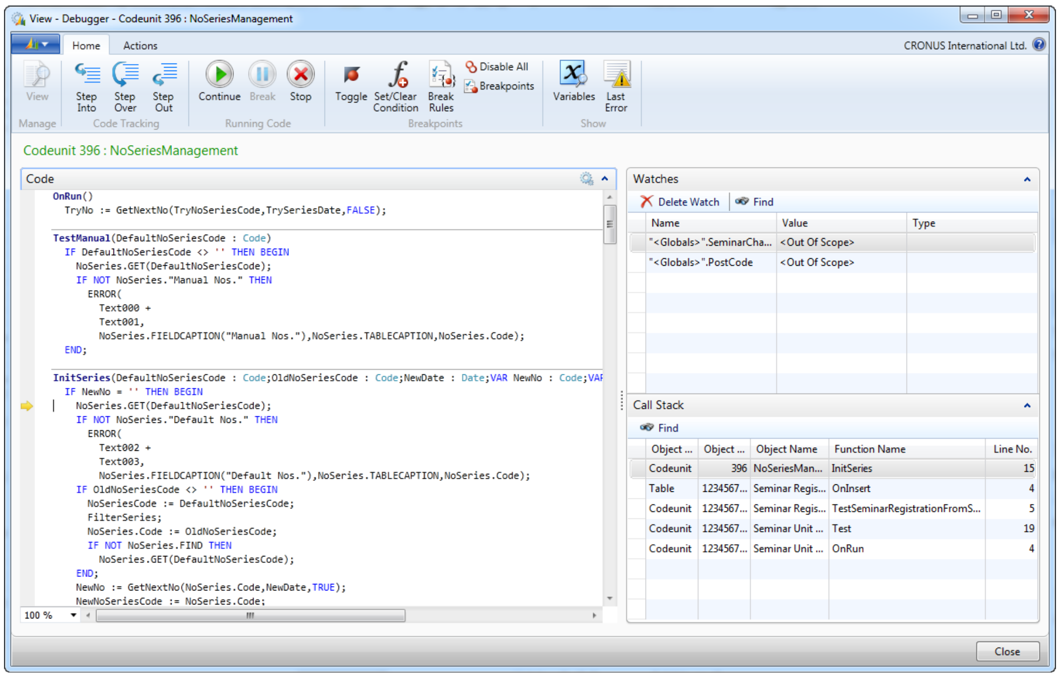
- Step Into
- Step Over
- Step Out

Step Into and Step Over differ in how they handle function calls. Either command instructs the debugger to execute the next line of code. If the line contains a function call, Step Into executes only the call itself and then stops at the first line of code inside the function. Step Over executes the function and then stops at the first line outside the function. Use Step Into if you want to look inside the function call. Use Step Over if you want to avoid stepping into functions.

Use Step Out when you are inside a function call and want to return to the calling function. Step Out resumes execution of your code until the function returns, and then breaks at the return point in the calling function.

The Running Code group on the **Home** tab provides the Continue action. The Continue action executes code until the next breakpoint or until execution ends.

The **Debugger** page indicates the current line of code by a yellow arrow to the left of the line that is being debugged. The arrow is positioned on the line that executes next.



```
OnRun()
  TryNo := GetNextNo(TryNoSeriesCode,TrySeriesDate,FALSE);

TestManual(DefaultNoSeriesCode : Code)
  IF DefaultNoSeriesCode <> '' THEN BEGIN
    NoSeries.GET(DefaultNoSeriesCode);
    IF NOT NoSeries."Manual Nos." THEN
      ERROR(
        Text000 +
        Text001,
        NoSeries.FIELDCAPTION("Manual Nos."),NoSeries.TABLECAPTION,NoSeries.Code);
  END;

InitSeries(DefaultNoSeriesCode : Code;OldNoSeriesCode : Code;NewDate : Date;VAR NewNo : Code;VAF
  IF NewNo = '' THEN BEGIN
    NoSeries.GET(DefaultNoSeriesCode);
    IF NOT NoSeries."Default Nos." THEN
      ERROR(
        Text002 +
        Text003,
        NoSeries.FIELDCAPTION("Default Nos."),NoSeries.TABLECAPTION,NoSeries.Code);
    IF OldNoSeriesCode <> '' THEN BEGIN
      NoSeriesCode := DefaultNoSeriesCode;
      FilterSeries;
      NoSeries.Code := OldNoSeriesCode;
      IF NOT NoSeries.FIND THEN
        NoSeries.GET(DefaultNoSeriesCode);
    END;
    NewNo := GetNextNo(NoSeries.Code,NewDate,TRUE);
    NewNoSeriesCode := NoSeries.Code;
```

Name	Value	Type
"<Globals>".SeminarCha...	<Out Of Scope>	
"<Globals>".PostCode	<Out Of Scope>	

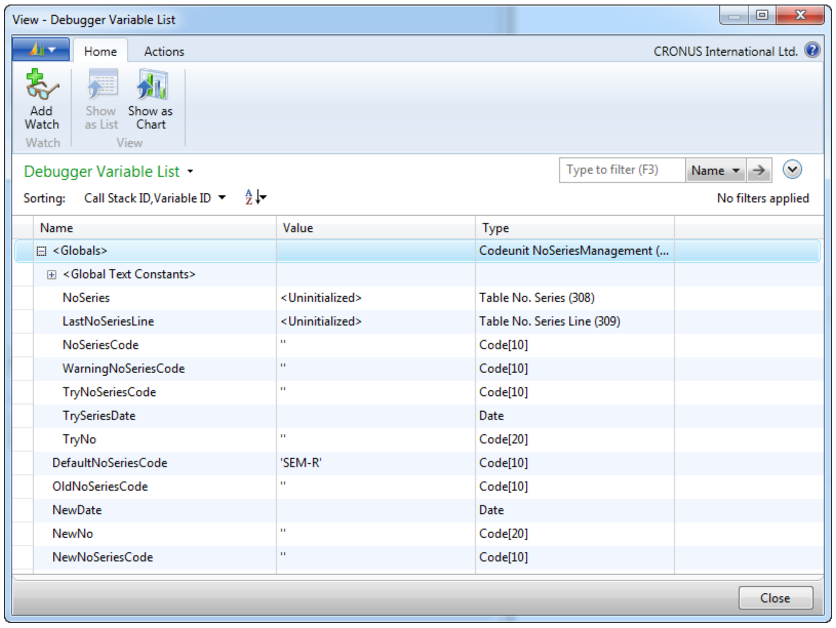
Object ...	Object ...	Object Name	Function Name	Line No.
Codeunit	396	NoSeriesMan...	InitSeries	15
Table	1234567...	Seminar Regis...	OnInsert	4
Codeunit	1234567...	Seminar Regis...	TestSeminarRegistrationFromS...	5
Codeunit	1234567...	Seminar Unit ...	Test	19
Codeunit	1234567...	Seminar Unit ...	OnRun	4

FIGURE 12.20: CURRENT LINE IN THE DEBUGGER PAGE

Variables and Watches

Bugs are frequently caused because variables contain values that differ from what you expected when you designed the application. During debugging, you frequently have to inspect the contents of variables, parameters, or text constants at each point.

The **Debugger Variable List** page provides an overview of all variables in scope of the current line of code. For each variable, you view the name, value, and data type.



Name	Value	Type
<Globals>		Codeunit NoSeriesManagement (...
<Global Text Constants>		
NoSeries	<Uninitialized>	Table No. Series (308)
LastNoSeriesLine	<Uninitialized>	Table No. Series Line (309)
NoSeriesCode	''	Code[10]
WarningNoSeriesCode	''	Code[10]
TryNoSeriesCode	''	Code[10]
TrySeriesDate		Date
TryNo	''	Code[20]
DefaultNoSeriesCode	'SEM-R'	Code[10]
OldNoSeriesCode	''	Code[10]
NewDate		Date
NewNo	''	Code[20]
NewNoSeriesCode	''	Code[10]

FIGURE 12.21: DEBUGGER VARIABLES LIST

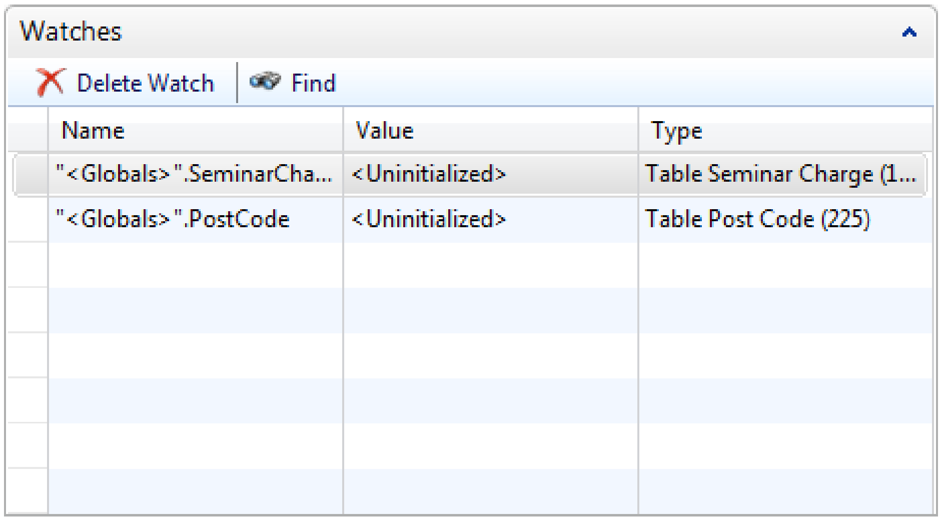
You can use the **Watches** FactBox to view the values of variables. Variables that you add to the **Watches** FactBox are displayed until you delete them, even if they go out of scope in the currently executing code. This differs from the **Debugger Variable List** page, which displays only the variables that are currently in scope. If a variable is out of scope, then <Out of Scope> is displayed in the Value column of the **Watches** FactBox.

To add a variable to the Watches list, follow this procedure:

1. In the **Debugger** window, in the code viewer, rest the pointer over the variable that you want to add to **Watches**. A DataTip appears.
2. In the DataTip, click the **Watch** icon to the left of the variable name.

Alternatively:

1. In the **Debugger** window, click **Variables**.
2. In the **Debugger Variable List** window, select the variable that you want to add to **Watches**, and then click **Add Watch**.



Name	Value	Type
"<Globals>".SeminarCha...	<Uninitialized>	Table Seminar Charge (1...
"<Globals>".PostCode	<Uninitialized>	Table Post Code (225)

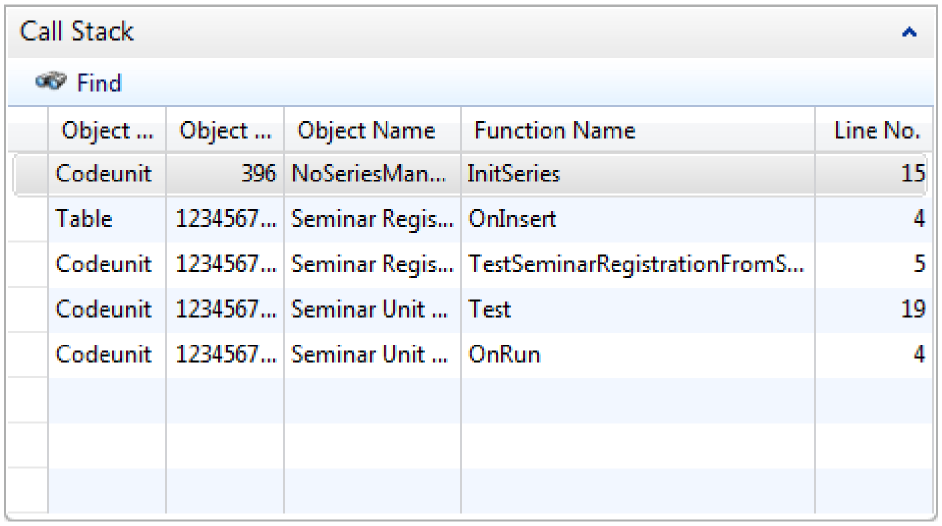
FIGURE 12.22: WATCHES FACTBOX

Note: Variables that you add to the **Watches** FactBox are persisted between debugging sessions.

Call Stack

In the **Call Stack** FactBox, you can view the triggers or function calls that led to the current line of code. Each line shows a single function or a trigger, and shows information about the object type, object ID, object name, function name, and line number of the line where another function was called or a breakpoint was hit. The trigger or function that started the current transaction is at the bottom of the call stack. The current function or trigger is at the top of the call stack.

You can click any row lower in the **Call Stack** FactBox to view the code for the object that is indicated in the Call Stack line with a green arrow. The arrow indicates the line that called a function or ran a trigger higher in the call stack.



Call Stack

Find

Object ...	Object ...	Object Name	Function Name	Line No.
Codeunit	396	NoSeriesMan...	InitSeries	15
Table	1234567...	Seminar Regis...	OnInsert	4
Codeunit	1234567...	Seminar Regis...	TestSeminarRegistrationFromS...	5
Codeunit	1234567...	Seminar Unit ...	Test	19
Codeunit	1234567...	Seminar Unit ...	OnRun	4

FIGURE 12.23: CALL STACK

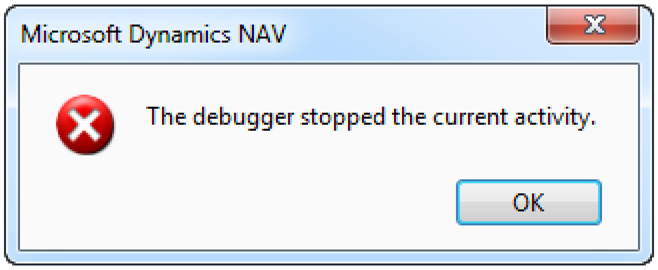
Note: *When you click any line lower in the call stack, you can access the variables in scope at that point of execution. The* ***Watches*** *FactBox also updates to show the values of watched variables at that point.*

Running Code

You can control how debugger runs the code by clicking actions in the Running Code group of the **Home** tab in the **Debugger** page.

To continue the execution of the code without stepping through lines, click **Continue**, or press F5. This runs the code until the next breakpoint is reached, or until all C/AL code in the current call is executed.

To stop current debugging activity, click **Stop**. This ends the current transaction, stops executing any remaining code on its execution path, and shows the following error message.



Microsoft Dynamics NAV

The debugger stopped the current activity.

FIGURE 12.24: RUN-TIME ERROR AFTER STOPPING THE DEBUGGER

***Note**: When you debug test functions that run from a test runner codeunit, each test function call is a separate transaction. When you click **Stop** in a test function, the current transaction ends and returns FAILURE as the result of the current test function. Then the test runner continues to run the remaining test codeunits. Test functions are included in the test run.*

To break the debugger at the next C/AL statement, click **Break**. This action is enabled only when the debugger is not active, and when its caption includes the "Waiting for break" text. If you click **Break**, and then start an activity, such as clicking an action, starting data entry, or running an object, the debugger will break at the first C/AL statement that executes in the current session, regardless of whether a breakpoint is set on it.

Demonstration: Using the Debugger

The following demonstration shows how to use the Debugger feature of Microsoft Dynamics NAV 2013.

Demonstration Steps

1. Set a breakpoint to break when a new record is inserted into the **Seminar** table.
   a. In Object Designer, design table 123456700, **Seminar**.
   b. Click **View > C/AL Code**.
   c. Select the first line of code in the OnInsert trigger, and press F9. An enabled breakpoint indicator is displayed to the left of the line.
   d. Close the **Table Designer**.
   e. Start the Microsoft Dynamics NAV 2013 client for Windows. Make sure that there is only one instance of it running.

2. Start the **Debugger**.
   a. Click **Tools > Debugger > Debug Session**.
   b. In the **Session List** page, select the session where **Client Type** is RoleTailored client, and then click **Debug**. The **Debugger** page opens.

3. Trigger the breakpoint.
   a. In the Microsoft Dynamics NAV 2013 client for Windows, in the **Search** field, type "Seminars", then run the **Seminars** page.
   b. In the **Seminars** page, click **New**.

   c. In the **Seminar Card** page, click the **Name** field. This causes the page to insert a new record into the **Seminar** table, and runs the OnInsert trigger. The **Debugger** page starts, and the code execution stops at the first line of the OnInsert trigger.

4. Add a field to the **Watches** list.
   a. Position the pointer over the "No." text in the first line of code in the OnInsert trigger until the tooltip appears.
   b. Click the **Add Watch** icon next to Rec.Fields."No." A watch with name <Globals>.Rec.Fields."No." appears in the **Watches** FactBox.
   c. Check the watch to verify that the value for the **No.** field is blank.

5. Step through the code.
   a. Click **Step Over** (or press F10) until you reach the NoSeriesMgt.InitSeries function call.

Note: *Depending on how you tested labs in earlier modules, at this point you may get the following error: "The Seminar Setup does not exist." If you get this error, run page* ***123456702, Seminar Setup*** *and then repeat this step.*

   b. Click **Step Into** (or press F11) to start debugging the **InitSeries** function of the NoSeriesManagement codeunit. View the **Call Stack** FactBox, where a new line is added to the top. This indicates the position of the C/AL code in the current call.

6. Add another breakpoint and continue to run the code.
   a. Select the following line of code.

```
NewNo := GetNextNo(NoSeries.Code,NewDate,TRUE);
```

   b. Press F9 to add a breakpoint.
   c. Click **Continue** (or press F5). This continues the execution until the next breakpoint is reached.

7. Step into and out of a function and view **Watches**.
   a. Press F11 to step into the **GetNextNo** function. A new line is added to the top of the **Call Stack** FactBox. It indicates the first line in the **GetNextNo** function of the NoSeriesManagement codeunit.
   b. Press F10 two times.
   c. Position the pointer over the SeriesDate variable in the first line in the **GetNextNo** function until the tooltip appears.

   d. In the tooltip, click the **AddWatch** icon to add a watch for the SeriesDate variable.
   e. Check the **Watches** FactBox. Verify that the <Globals>.Rec.Fields."No." line indicates an <Out Of Scope> value. Verify that the SeriesDate line contains a value of 01/23/14.
   f. Click **Step Out**. Execution returns to the last line that is reached in the **InitSeries** function.

8. Use the **Call Stack** and **Watches** FactBoxes.
   a. Add a watch for the NewNo parameter.
   b. Verify that the value of the NewNo parameter is blank.
   c. Press F10.
   d. Verify that the NewNo parameter in the **Watches** FactBox contains the value that is assigned from a number series.
   e. Verify that the SeriesDate variable is out of scope.
   f. In the **Call Stack** FactBox, click the line for the OnInsert trigger in the **Seminar** table. The **Code** FastTab updates and shows the C/AL code in the **Seminar** table with a green arrow that indicates the line where the **InitSeries** function of the NoSeriesManagement codeunit was called.
   g. Verify that the <Globals>.Rec.Fields."No." line shows the value for the **No.** field that was assigned from the number series.

Note: The **No.** field is in the scope of the OnInsert trigger of the **Seminar** table. This is lower in the Call Stack FactBox. It contains the same value as the NewNo parameter in scope of the **InitSeries** function, because the **No.** field was passed by reference to the NewNo parameter of the **InitSeries** function.

   h. Verify that the NewNo line shows an <Out Of Scope> value.

Note: The NewNo parameter is in scope for the **InitSeries** function and is not known in the scope of the OnInsert trigger that is currently shown in the **Debugger** page.

9. End debugging.
   a. Click F5 to continue running the code. The **Debugger** switches to the "Waiting for break" mode.
   b. Focus the **Seminar Card** page. The **No.** field on the **Seminar Card** page shows the value that was assigned from the number series.
   c. Close the **Debugger** page.
   d. Close the **Session List** page.

Module Review

Module Review and Takeaways

Unit testing functionality of Microsoft Dynamics NAV 2013 includes many features to fully automate testing of code and avoid regression issues. Test codeunits contain test functions. Test functions contain code that simulates transactions or user activities, and validates the application functionality against the design goals. When they are executed, test codeunits report SUCCESS or FAILURE. This indicates whether the tested functionality behaves as expected. Handler functions replace user interactions, such as confirmation dialog boxes, or modal pages. Test pages can simulate the whole user interface and most types of interaction with the user, such as calling an action, drilling down on a field in a page, or inserting a new row in a subpage on a document. Test runner codeunits automate the running of test codeunits and enable you to control which tests are executed, and which tests collected test results for logging or integrating with test management solutions.

Debugger provides the functionality to analyze the code execution, follow the code line by line as it runs, and inspect the variables to determine the causes of bugs, errors, or other types of issues.

Test Your Knowledge

Test your knowledge with the following questions.

1. Which type is not a valid codeunit subtype in Microsoft Dynamics NAV 2013?

   ( ) Normal

   ( ) Test

   ( ) TestRunner

   ( ) UnitTest

2. What kind of functions can test codeunits contain?

____

____

____

____

3. You can define ConfirmHandler functions only in test runner codeunits.

   ( ) True

   ( ) False

4. The OnBeforeTestRun and OnAfterTestRun triggers are defined automatically for every test runner codeunit when you set the Subtype property to TestRunner.

   ( ) True

   ( ) False

5. Which C/AL statement can you use in test code to make sure that the following statement fails?

6. When you set the TransactionModel of a test function to AutoRollback, what happens if the test code run from that function encounters a COMMIT function call?

7. Which C/AL data type do you use to simulate user interaction with a page in test code?

8. You can debug web services sessions in Microsoft Dynamics NAV 2013.

   ( ) True

   ( ) False

9. What is a conditional breakpoint and how do you define it?

Test Your Knowledge Solutions

Module Review and Takeaways

1. Which type is not a valid codeunit subtype in Microsoft Dynamics NAV 2013?

   ( ) Normal

   ( ) Test

   ( ) TestRunner

   (√) UnitTest

2. What kind of functions can test codeunits contain?

   MODEL ANSWER:

   Test codeunits can contain normal, test, and handler functions.

3. You can define ConfirmHandler functions only in test runner codeunits.

   ( ) True

   (√) False

4. The OnBeforeTestRun and OnAfterTestRun triggers are defined automatically for every test runner codeunit when you set the Subtype property to TestRunner.

   ( ) True

   (√) False

5. Which C/AL statement can you use in test code to make sure that the following statement fails?

   MODEL ANSWER:

   ASSERTERROR

6. When you set the TransactionModel of a test function to AutoRollback, what happens if the test code run from that function encounters a COMMIT function call?

   MODEL ANSWER:

   A run time error occurs.

7. Which C/AL data type do you use to simulate user interaction with a page in test code?

   MODEL ANSWER:

   TestPage

8. You can debug web services sessions in Microsoft Dynamics NAV 2013.

   (√) True

   ( ) False

9. What is a conditional breakpoint and how do you define it?

   MODEL ANSWER:

   A conditional breakpoint breaks the execution only if the Boolean expression defined on it evaluates to TRUE. You define a condition for a breakpoint by clicking the **Set/Clear Condition** action in the Debugger.