CHAPTER 4: POSTING

Objectives

The objectives are:

- Create journal posting routines.
- Create document posting routines.
- Write internal documentation for modifications to existing objects.
- Debug code.
- Program for low-impact on the application.

Introduction

The Seminar module now contains master files and a means of creating registrations. The next step is to use the registration information to create ledger entries for seminars through a posting routine. This functionality will be added to the Seminar module by the end of this chapter.

Before going on, however, it's important to review some of the fundamentals of posting, including journal tables, ledger tables, and posting routines.

Prerequisite Information

Before beginning work on posting, it is important to know about journal tables, ledger tables, and some of the elements involved in posting.

Posting Tables and Routines

Journal Tables

A Journal is a temporary work area for the user. Records can be inserted, modified and deleted as the user wishes. There are three tables that make up the Journal:

- Journal Line (the main table)
- Journal Template (supplemental table)
- Journal Batch (supplemental table)

The primary key of the Journal Line table is a compound key (that is, it has more than one field). It consists of **Journal Template Name**, **Journal Batch Name** and **Line No.** The journal form that the user chooses sets the **Journal Template Name**, and this does not change unless the user goes into a different journal form. The **Journal Batch Name** may be changed at the top of the form (the options that are available depend upon the template chosen), but only one can be viewed at a time. The **Line No**. keeps each record in the same template and batch unique. Line number is incremented automatically by the form (see the **AutoSplitKey** property). The user never sees most of the primary key fields on the form (other than the batch).

The journal form lets users enter journal lines that are added to the detail tables of the system (the ledgers), but nothing happens until users decide to post. Users can leave the lines in the journal table as long as they wish without posting.

Ledger Tables

A ledger is a protected table that holds all the transactions for a particular functional area. These records are permanent and cannot be deleted or modified except through special objects. Also, records cannot be inserted directly into the table. Records cannot get into a ledger except through a posting routine, and posting routines only post journal lines.

The primary key is the **Entry No.** field. There are many secondary keys, and most are compound. They are set up for reports, forms, and FlowFields. The **Ledger** table holds the majority of detail information for the functional area.

A ledger table should never be modified directly, especially not by inserting or deleting records. There is a linkage between most of the ledger tables back to the **General Ledger**. Because of this linkage, any modifications made directly to the table can be disastrous. Usually, the only way to undo such changes is to restore the most recent backup of the database.

Posting Routines

A posting routine is a group of codeunits responsible for ensuring that all transactions put into the corresponding ledgers are correct per line and correct as a whole. The posting routine takes Journal Lines, checks them, converts them to ledger entries, inserts them into the Ledger table, and ensures that all transactions made were consistent.

Although there are many different types of posting routines in Microsoft Dynamics™ NAV, there are some general rules that pertain to all of them. The primary codeunit that does the work of posting for a particular journal is simply called the posting routine. This codeunit is named after the Journal name with the addition of "-Post Line." Its main job is to transfer the information from the Journal record to the Ledger table, though it also does other things, such as calculations and data checking.

Posting Routine Companion Codeunits

The posting routine codeunit has two companion codeunits:

- -Check Line
- -Post Batch

The **Check Line** routine is called by the **Post Line** routine to check each Journal line before sending it to the server for processing. Thus, all of its testing routines either do not touch the server at all, or, at most, only once in each posting process.

The **Post Batch** routine repeatedly calls the **Check Line** routine to test all lines. It then repeatedly calls the **Post Line** routine to post all lines. The **Post Batch** routine is the only one that actually reads or updates the Journal table; the others simply use the Journal record passed into them. In this way, a programmer can call the **Post Line** routine directly (from another posting routine) without having to update the Journal table. The **Post Batch** routine is used only when the user selects Post within the Journal form.

The last digits of the object numbers of these posting routines are standardized. The **Check Line** always ends with 1, the **Post Line** with 2, and the **Post Batch** with 3. As an example, **Gen. Jnl.-Check Line** is codeunit 11, **Gen. Jnl.-Post Line** is codeunit 12 and **Gen. Jnl.-Post Batch** is codeunit 13.

Note that none of these routines have any interface that requires user input. This is so that they can be called from other applications without having to worry about messages popping up (except error messages). The **Post Batch** routine has a dialog that displays the progress of the posting and lets the user cancel. The rest of the user interface that has to do with posting is handled by another set of routines:

- Post routine (which just asks whether to post and then calls **Post Batch**) has an Object ID that ends with 1.
- Post and Print routine (which asks whether to post, calls **Post Batch**, and then calls the Register Report) ends with 2.
- Batch Post (which asks whether to post the selected batches and then repeatedly calls **Post Batch** for each one) is called from the Journal Batches form and ends with 3.
- Batch Post and Print (which confirms that the user wants to post, then calls **Post Batch** for each batch, and then calls the Register Report) ends with 4.

As an example, **Gen. Jnl.-Post** is codeunit 231, **Gen. Jnl.-Post+Print** is codeunit 232, **Gen. Jnl.-B.Post** is codeunit 233, and **Gen. Jnl.-B. Post+Print** is codeunit 234.

Check Line Codeunit

The name of the **Check Line** codeunit explains its function. It is designed to check the Journal Line that is passed to it. It does so without reading from the server, except perhaps the first time it is called.

Before checking any of the fields, this codeunit usually makes sure the journal line is not empty. It does so by calling the **EmptyLine** function in the Journal table. If the line is empty, the codeunit skips it by calling the EXIT function. There is no error, and the posting process continues.

The last thing that the codeunit verifies is the validity of the dimensions that are passed into the function in a temporary Dimensions table. This is done by some simple calls to the **DimensionManagement** codeunit. Dimensions are discussed later in this course. If the codeunit does not stop the process with an error, then the journal line is accepted.

Post Line Codeunit

The **Post Line** codeunit is responsible for actually writing the journal line to the ledger. It only posts one **Journal Line** at a time. It does not look at previous or upcoming records.

The code in the **OnRun** trigger of this codeunit is inserting records. The **OnRun** trigger of the **Post Line** codeunit is normally never called, but in prior versions of the product it was. For backward compatibility, the **OnRun** trigger loads the temporary Dimensions table with the two global dimensions and then calls the **RunWithCheck** function. This is the
function that is normally called by other functions or triggers. It in turn calls the workhorse of the **Post Line** routine, the **Code** function.

Like **Check Line**, this codeunit skips empty lines by exiting. This ensures that empty lines are not inserted into the ledger. The first thing the codeunit does, if the line is not empty, is to call **Check Line** to verify that all the needed journal fields are correct.

Next, the codeunit checks the important table relations. This requires reading the database (using **GET**), which is why it is done here rather than in **Check Line**.

Before writing to the ledger, **Post Line** writes to the register. The first time the program runs through the **Post Line** codeunit, it inserts a new record in the **Register** table. In every subsequent run through **Post Line**, the program modifies the record by incrementing the **To Entry No**.

Then the codeunit takes the next entry number and the values from the journal line and puts them into a ledger record. Finally, it can insert the ledger record.

Near the bottom of the **Code** function is the call to the **DimensionManagement** codeunit that copies the dimensions from the temporary **Journal Line Dimension** table to the real **Ledger Entry Dimension** table.

The entry number for this new ledger record is passed into the function to keep the dimension records associated with this ledger entry. The very last thing that the codeunit does is to increment the variable that holds the next entry number by one. Thus, when the codeunit is called again, the next entry number is ready.

When the **Post Line** codeunit is done, one journal line has been processed, but more than one ledger record may have been inserted into more than one ledger.

Post Batch Codeunit

The **Post Batch** codeunit is responsible for posting the Template and Batch that were passed to it. Only one record variable for the journal is actually passed in, but the codeunit starts by filtering down to the template and batch of the record that was passed in. Then it finds out how many records are in the record set that the record variable represents. If the answer is none, the codeunit exits without an error, and it is up to the calling routine to let the user know that there was nothing to post.

The **Post Batch** codeunit can then begin checking each journal line in the record set by calling the **Check Line** codeunit for each line. Once all the lines are checked, they can be posted by calling the **Post Line** codeunit for each line. By then, the codeunit has looped through all the records twice, once for **Check Line** and a second time for Post Line. To call the appropriate functions in **Check Line** and Post Line, **Post Batch** must fill in a temporary Journal Line Dimension table with all of the dimensions for the journal line. It can then pass this temporary record variable into the **RunCheck** function of **Check Line** (or the **RunWithCheck** function of **Post Line**) along with the journal line record variable.

Where **Check Line** checks consistency of a given line, the **Post Batch** codeunit is responsible for checking the inter-relation of the lines. For example, if the codeunit is for general journal lines, it may also be responsible for making sure that the journal lines balance. This usually takes place after they are checked and before they are posted.

The codeunit may do other special things depending on the **Journal Template**. For recurring journals, the journal lines are updated with new dates based on the date formula. When a recurring journal line is posted, the codeunit must check the **Description** and **Document No.** fields and perhaps replace any replaceable parameters with the correct values (%1 = day, %2 = week, %3 = month, and so on).

If the template is not recurring, the codeunit deletes all the journal lines.

The diagram in Figure 4-1 outlines the steps in the Posting Routine when **Post Batch** is called.

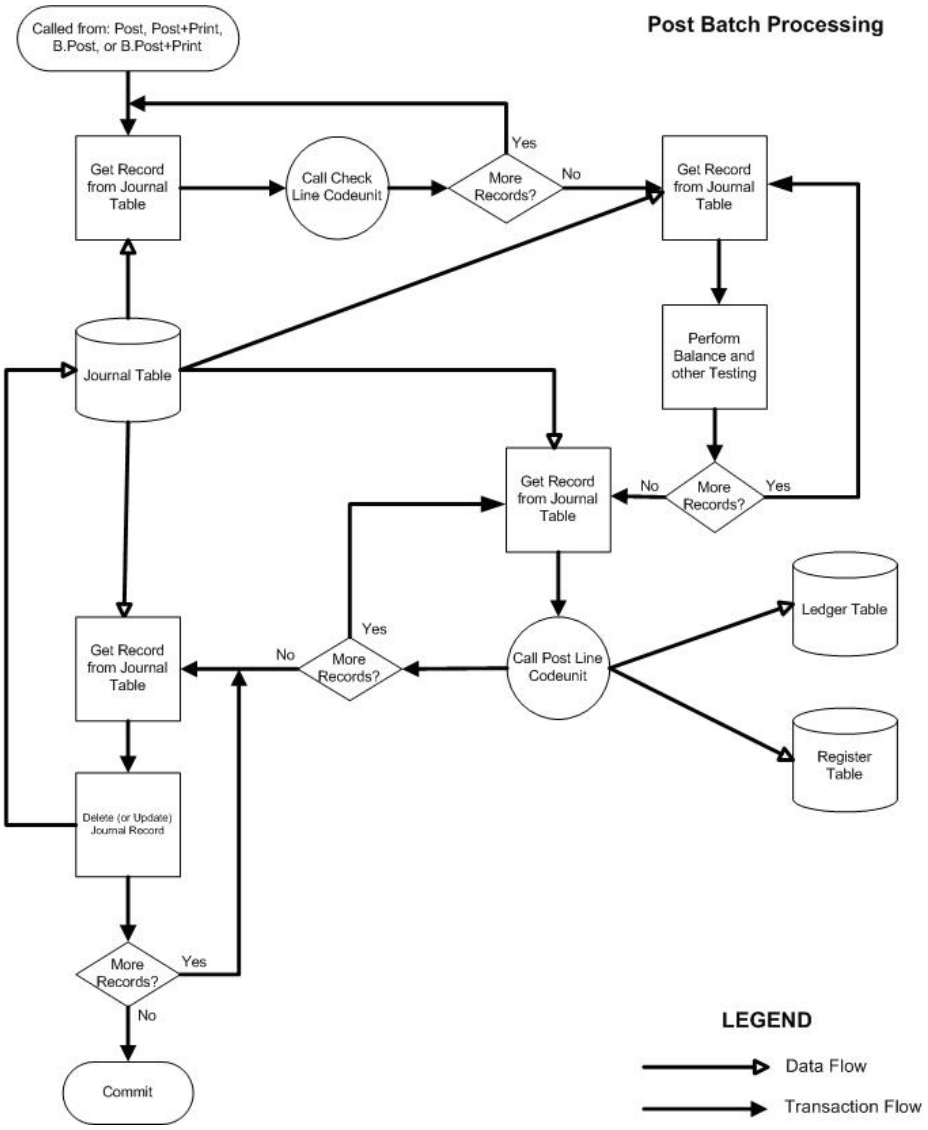


FIGURE 4–1: THE POST BATCH LOGIC DIAGRAM

Example Posting Routine

In this section existing posting routines are discussed to provide a basic idea of how these routines tend to be written.

Check Line

Design the **Res. Jnl.-Check Line** codeunit (211). Notice that the **OnRun** trigger gets the GL Setup record and then sets a temporary record for the dimensions. Dimensions are discussed later in this course.

In the **RunCheck** function trigger, notice that the codeunit checks to see if the line is empty. If the line is empty, the codeunit skips further checking and exits without error.

The codeunit also checks to see if the posting date is within the allowable posting date from the GL Setup table.

Finally, the **Check Line** codeunit calls functions from the DimensionManagement codeunit to verify the dimensions.

If this codeunit goes through without error, then the posting routine continues.

Post Line

Design the **Res. Jnl.-Post Line** codeunit (212). Notice that the **OnRun** trigger again gets the GL Setup record and also sets a temporary record for the dimensions. As mentioned previously in this chapter, the **OnRun** trigger contains code for backward compatibility. Most codeunits call the **RunWithCheck** function, which then calls the Code function, where most of the posting is done in the posting routine.

Notice, like the **Check Line** codeunit, the **Post Line** codeunit skips empty lines by exiting. This ensures that empty lines are not inserted into the ledger. Next, if the line is not empty, it calls **Check Line** to verify that all the needed journal fields are correct.

Next, the codeunit gets the next entry number from the **Resource Ledger Entry** table to be used with the resource register table. As previously mentioned, before writing to the ledger, **Post Line** writes to the register. The codeunit first adds a new record into the **Register** table, and then every subsequent run through **Post Line** increments the **To Entry No**.

Then the codeunit takes the next entry number and the values from the journal line and puts them into a ledger record. Finally, it can insert the ledger record.

After the Ledger is inserted into the table, this codeunit calls the **DimensionManagement** codeunit to transfer the dimension lines from the temporary record to the Ledger entry table.

Post Batch

Design the **Res. Jnl.-Post Batch** codeunit (213). This codeunit is responsible for posting the Resource Template and Resource Batch that were passed to it.

The codeunit starts by filtering to the template and batch of the **Resource Journal Line**. If no records are found in this range, **Post Batch** exits and it is up to the calling routine (codeunit 271, 272, 273, or 274 in this case) to let the user know that there was nothing to post.

The **Post Batch** codeunit then loops and checks each journal line in the record set by calling the **Check Line** codeunit. Once all the lines are checked, they enter another loop which posts the records by calling the **Post Line** codeunit for each line. **Post Batch** must fill in a temporary **Journal Line Dimension** table with all of the dimensions for the journal line. It can then pass this temporary record variable into the **RunCheck** function of **Check Line** (or the **RunWithCheck** function of Post Line) along with the journal line record variable.

Unlike the **General Journal**, there are no interdependencies between **Resource Journal** lines, so no checks such as checking the balance need to be done here.

Lastly, this codeunit calls the **UpdateAnalysisView** codeunit to update all of the Analysis Views.

Document Posting Routines

A document in Microsoft Dynamics NAV is an easy interface for a user to perform many complicated transactions. The seminar registration document is used to tie a seminar registration to a number of customers and participants as well as to comments and seminar charges. The document posting routine for seminar registration translates this document information into journal entries, which can then be posted to ledger entries.

To understand more clearly how a document posting routine works and what its components are, consider the example of a sales order with three sales lines:

- Line 1: selling a G/L Account (perhaps this line adds some kind of surcharge or freight).
- Line 2: selling an item (for example, a computer).
- Line 3: selling a resource (for example, the time that an employee has spent custom-building the computer).

When the user posts the document, the program generates an entry that debits the **Accounts Receivable Account** in the G/L, and each line could generate a separate G/L entry for the amount of that line. At the same time, entries are being made for the **Item** and **Resource** journals, as well as the **General** journal for the **Customer**.

When these journal entries are posted, they are posted as if the user had entered them into the journals. The biggest difference is that the journal records are posted one at a time, which allows the Sales Post routine to bypass **Post Batch** and call **Post Line** directly.

In the example of a sales order posting, the **Gen. Jnl.-Post Line** codeunit (12) would be called at least twice, the **Item Jnl.-Post Line** codeunit (22) would be called once, and the **Res. Jnl.-Post Line** codeunit (212) would be called once.

A sales document is posted primarily by the **Sales-Post** codeunit (80). An entire batch of sales documents can be posted by calling the **Batch Post Sales Invoices** report (297). Note that this report is for invoices only. There is a separate report for each document type.

These reports call the **Sales-Post** codeunit repeatedly for each document. For this to work, **Sales-Post** must not interact with the user. In fact, **Sales-Post** is never called directly by a form. The form calls the **Sales-Post (Yes/No)** codeunit (81), codeunit **Sales-Post + Print** (82), or one of the reports mentioned previously. They in turn interact with the user (getting confirmation or other information) and then call **Sales-Post** as appropriate. These interactions are shown in Figure 4-2.

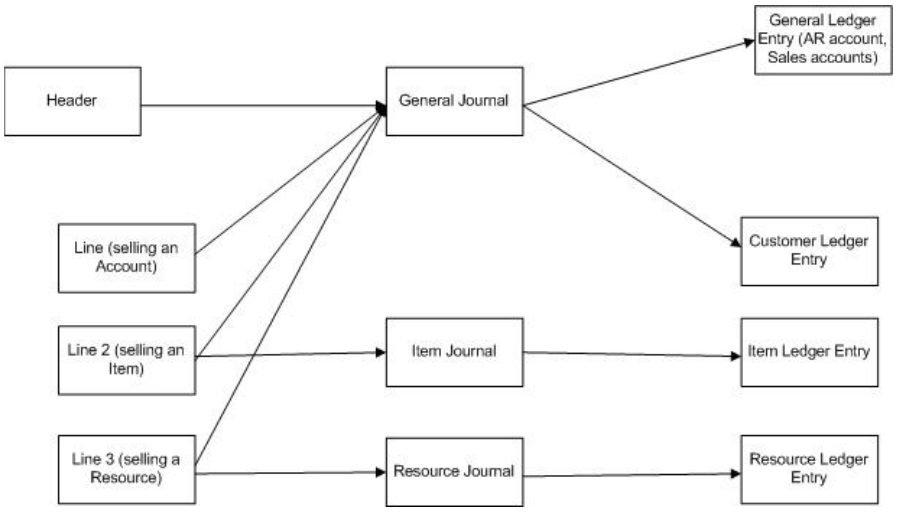


FIGURE 4–2: THE DOCUMENT POSTING DIAGRAM

Document Posting Routine

Codeunit 80 posts documents. Assume that the user is shipping and invoicing a **Sales Order**, and scroll through the codeunit and identify the sections that perform the following tasks:

- The codeunit starts by determining the document type and validates the information that appears on the sales header and lines. The codeunit determines what the posted document numbers are going to be and updates the header. This section ends with a **COMMIT**.
- The codeunit locks the appropriate tables.
- The codeunit inserts the **Shipment Header** and **Invoice** or **Credit Memo Header**.
- The codeunit processes the sales lines, starting by clearing the **Posting Buffer** (a temporary table based on the Invoice **Post.Buffer** table) and ends with the **UNTIL** that goes with **REPEAT**, which loops through all the sales lines. Each line is processed individually.
- Within the **REPEAT** loop, the codeunit checks each line with its matching Shipment line (if the line was previously shipped). If the line type is an **Item** or a **Resource**, it is posted through the correct journal. The line is then added to the posting buffer. It may be inserted, or it may update a row already there. If the line is related to a job, it posts a journal line through the Job Journal. Then, if there is no shipment line, it inserts one. Finally, it copies the **Sales Line** to the **Invoice Line** or **Credit Memo Line** (the posted tables).
- The codeunit can now post all entries in the **Posting Buffer** to the **General Ledger**. These are the **Credits** that are created from the sale of the lines. Then the codeunit can post the **Debit** to the **General Ledger**. The customer entry is made to the **Sales Receivables Account**. The routine then checks whether there is a balancing account for the header. This corresponds to an automatic payment for the invoice.
- Lastly, the codeunit updates and/or deletes the **Sales Header** and **Lines** and commits all changes.

Documentation in Existing Objects

When changes are made to an existing object, a note should be entered in the **Documentation** trigger. Usually, this means one note for each feature. The note heading should contain a reference number, the date the modification was completed, the name of the developer responsible for the modification, the project name, and a short description of the change. Here is an example:

```
Microsoft Business Solutions
------------------------------------------------------
Project: Solution Development II Training
jtd: John T. Doe

No.  Date        Sign    Description
------------------------------------------------------
001  01.21.2004  jtd     Created
002  05.05.2004  jtd     Transfer new field
                         Seminar Registration No.
```

Notice that documentation in an existing object looks similar to the documentation in a new object, except that a new reference number is created for each modification.

Code Comments

Along with the general comments that provided in the **Documentation** trigger, it is important to provide comments in the code at the lines where a change has been made. Only do this when modifying an existing object, not when creating a new object.

The key is to mark the changed code with the same reference number as used in the **Documentation** trigger of the object. For example, a modified single line of code should be marked like this:

```
State := "Employee."Default Work State"; // jtd002

If an entire block of code has been added or modified, mark
the change as follows:

// - jtd:002 ---
  JobLedgEntry."Seminar Registration No." := "Seminar
Registration No.";
// + jtd:002 +++
```

For the removal of a block of code, mark it as follows:

```
{ - jtd:002 --- Start Deletion
State := "Employee."Default Work State"; Locality :=
"Employee."Default Work Locality"; "Work Type Code" :=
Employee."Default Work Type Code";
 + jtd:002 +++ End Deletion}
```

Note that this keeps the old code in place, just commented out. Never delete Microsoft Dynamics NAV base code – comment it out instead.

Performance Issues

When writing large posting routines, it is important to program with the idea of maximizing performance. There are a number of steps to take while programming a solution in Microsoft Dynamics NAV that help improve performance.

Table Locking

Normally there is no need to be concerned with transactions and table locking when developing applications in C/SIDE. There are, however, some situations where a table must be locked explicitly. For example, suppose that in the beginning of a function, data in a table is inspected and then used to perform various checks and calculations. Finally, the record is written back based upon the result of this processing. The values retrieved at the beginning should be consistent with the final data in the table. In short, other users cannot update the table while a function is busy doing calculations.

The solution is to lock the table at the beginning of the function by using the **LOCKTABLE** function. This function locks the table until the write transaction is committed or aborted. This means that other users can read from the table, but they cannot write to it. Calling the **COMMIT** function unlocks the table.

The **RECORDLEVELLOCKING** property is used to detect whether record level locking is being used. This property is only used with the SQL Server Option for Microsoft Dynamics NAV, which is currently the only server that supports record level locking.

Keys and Queries

When writing a query that searches through a subset of the records in a table, always carefully define the keys both in the table and in the query so that Microsoft Dynamics NAV can quickly identify this subset. For example, the entries for a specific customer are normally a small subset of a table containing entries for all the customers. If Microsoft Dynamics NAV can locate and read the subset efficiently, the time it takes to complete the query will only depend on the size of the subset.

To maximize performance, define the keys in the table so that they facilitate the necessary queries. These keys must then be specified correctly in the queries.

Reducing Impact on the Server

Keeping the processing of transactions on the client as much as possible minimizes the load on the server and improves performance, particularly when there are a large number of users. There are several ways to achieve this:

- Try to use the **COMMIT** function as little as possible. This function is handled automatically by the database for almost all circumstances.
- Limit the use of the **LOCKTABLE** function to where it is necessary. Remember that an insert, modify, rename, or delete locks the table automatically, so for most situations tables do not need to be locked explicitly.
- Try to structure the code so that it does not have to examine the return values of the **INSERT**, **MODIFY**, or **DELETE** functions. When using the return value, the server has to be notified right away so that a response can be obtained. Therefore, if they aren't necessary, do not look at these return values.
- Use the **CALCSUMS** and **CALCFIELDS** functions whenever possible to avoid going through records to add up values.

Reducing Impact on Network Traffic

Consider setting keys and filters and then using **MODIFYALL** or **DELETEALL** functions, which send only one command to the server, rather than getting and deleting or modifying each of many records, which sends more information back and forth through the network.

Since **CALCSUMS** and **CALCFIELDS** can both take multiple parameters, use these functions to perform calculations on several fields with one function call.

Debugging Tools

There are three categories of possible errors when developing applications in C/AL code:

- Syntax errors – These are errors where the program encounters an unknown identifier, or where the program expects keywords that are not there, such as an **IF** or **THEN** that is missing. Syntax errors are detected by the compiler.
- Runtime errors – These errors are not detected by the compiler and only occur when the code is executed. A good example of this is when division by zero occurs in the code.
- Program logic errors – These errors occur when the code compiles and runs, but does not function as intended.

Microsoft Dynamics NAV has tools that help track the source of errors and report what code has been run.

Debugger

Microsoft Dynamics NAV provides an integrated debugging tool to help check and correct code that does not run smoothly.

The debugger allows breakpoints to be defined. These are marks in the code where the debugger will stop execution so that the current state of the program can be examined. The Breakpoint on Triggers setting (SHIFT+CTRL+F12) is enabled by default when the debugger is activated for the first time. Otherwise the code would be executed normally because there are no breakpoints. The debugger therefore suspends execution of the code when it reaches the first trigger. At this point other breakpoints can be set, and then the Breakpoint on Triggers option can be disabled if desired. If the Breakpoint on Triggers setting is not disabled, the debugger suspends execution of the code each time a trigger is executed.

Breakpoints can also be set or removed from the C/AL Editor by either pressing F9 or clicking TOOLS→DEBUGGER→TOGGLE BREAKPOINTS. Information about breakpoints is stored in the **Breakpoints** virtual table when the C/AL Editor is closed.

Activate the debugger from Microsoft Dynamics NAV by clicking TOOLS→DEBUGGER→ ACTIVE (SHIFT+CTRL+F11).

In the debugger interface there are several windows and menus:

- From the **Edit** menu, the Breakpoints dialog box (SHIFT+F9) displays a list of the breakpoints that are set for the object being debugged. This dialog box can be used to enable, disable, and remove breakpoints in the list.
- The **View** menu contains commands that display the various debugger windows, such as the Variables window and the Call Stack window.
- The **Debug** menu contains commands that start and control the debugging process, such as Go, Step Into, Step Over, and Show Next Statement.
  - The Go command executes code from the current statement until a breakpoint or the end of the code is reached, or until the application pauses for user input.
  - The Step Into command executes statements one at a time, by stepping into any function that is called. This means that the debugger follows execution into the function as opposed to executing the function and moving to the next command.
  - The Step Over command executes statements one at a time, like Step Into, but if using this command, when a function call is reached, the function is executed without the debugger stepping into the function instructions.
  - The Show Next Statement command shows the next statement in the code.

Code Coverage

When the code coverage functionality is activated, the program logs the code and objects that are run. This applies only for the duration that the code coverage is activated. This can be useful when customizing Microsoft Dynamics NAV and want to test the implementation. It provides a quick overview of the objects for which code has been executed and displays the code that has been run.

To activate code coverage:

1. Click TOOLS→DEBUGGER→CODE coverage. The Code Coverage window opens.
2. Click **Start** to begin logging code.
3. When the monitors transactions have completed, return to the Code Coverage window, which now contains a list of any tables, forms, reports, dataports, and codeunits that were executed.
4. Click **Stop**.
5. Select an object and click **Code** to open the Code Overview window.

The Code Overview window displays code for the object selected in the Code Coverage window. Lines of code that were executed during the transaction(s) are shown in black. Lines of code that were not executed are shown in red.

Handling Runtime Errors

It is possible to avoid some runtime errors by writing code that handles possible errors. A typical example of this is the **GET** function. The return value of the **GET** function is a Boolean, so that if the program finds the record in the table, the function returns TRUE; if the program does not find the record, the function returns FALSE.

If the **GET** function is used as it is shown here, a runtime error occurs if the program cannot find the specified record:

```
Customer.GET("Customer Number");
```

Handling a **FALSE** result from the **GET** fuction with a message or error provides a much better opportunity for the user to be able to understand how to correct the problem. For example, the **GET** function can be called this way, with the error handled by the **ELSE** clause:

```
IF Customer.GET("Customer Number") THEN
....
ELSE
....
```

Client Monitor

The Client Monitor, which is accessed by clicking **Tools**, Client Monitor, is actually a virtual table called **Monitor**, which can be used to get an overview of the time consumption of specific operations. This tool can be valuable when optimizing performance.

In the Client Monitor window, use the **Options** tab to specify the kind of information gathered by the Client Monitor. The **Options** tab also contains advanced parameters that are only available with the SQL Server Option. Refer to the Application Designer's Guide for more information on these parameters.

Posting Seminar Registrations

In this section the client's functional requirements are analalized and a solution is designed and implemented.

Solution Analysis

The client's functional requirements presented in Chapter 1 describe the posting of registration information this way:

When each seminar is finished, the customers with participants should be invoiced. This should be done by project and resources.

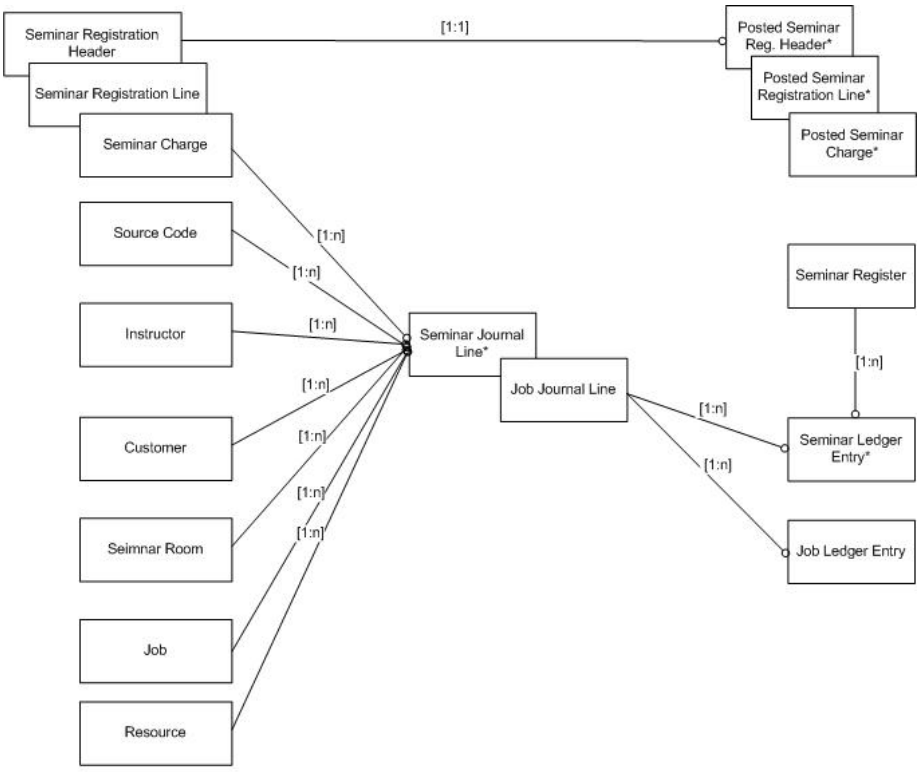
Therefore seminars should be posted as jobs, but also have a separate ledger with seminar-specific information. Posting the registration enables customers to be invoiced participation in seminars.

Upon completion of a seminar, seminar managers can verify the participant list with the trainer, and post the registration document.

Solution Design

Implementing the registration posting functionality means that the completed transaction data is used to create ledger entries from which to view history, create statistics, and create invoices.

The basic design for posting seminar registrations is similar to that of standard journal posting. Since a document is being posted, additional posted document tables and forms that contain historical information can be created as well. The master table relationships are shown in Figure 4.3.



* new table

FIGURE 4-3: TABLE INTERACTION

GUI Design

The forms for the seminar registration posting and the navigation between them reflect the relationships shown in Figure 4–3. By designing the simplest forms first, integration with the more complex forms is made easier.

The **Source Code Setup** form should be modified by adding one new field to the form, **Seminar**, as shown in Figure 4–4.

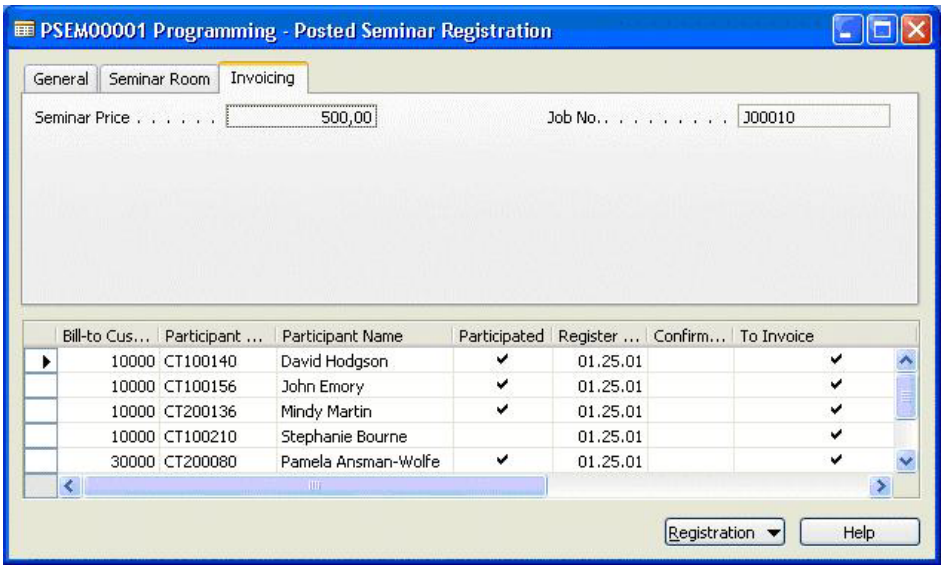


FIGURE 4–4: THE SOURCE CODE SETUP FORM (279)

The **Seminar Ledger Entries** form, shown in Figure 4-5, displays the Ledger entries.

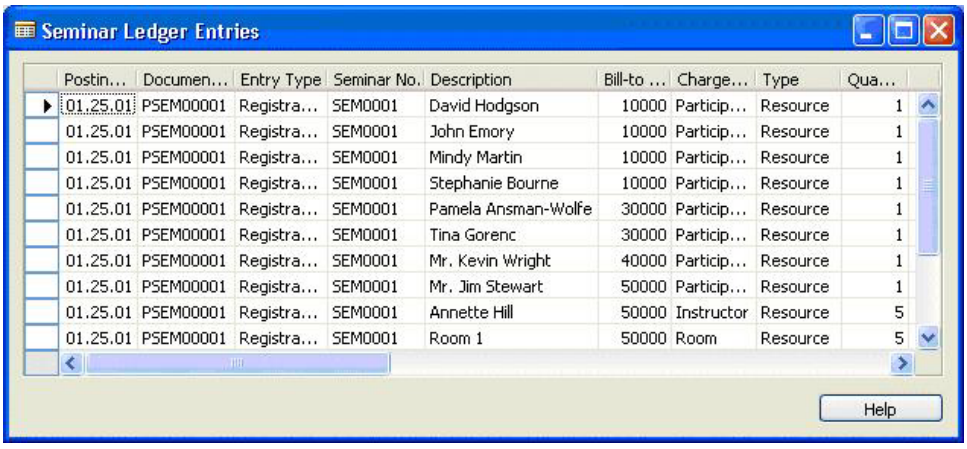


FIGURE 4–5: THE SEMINAR LEDGER ENTRIES FORM (123456721)

The **Seminar Registers** form (Figure 4-6) displays the registers created when seminar registrations are posted.

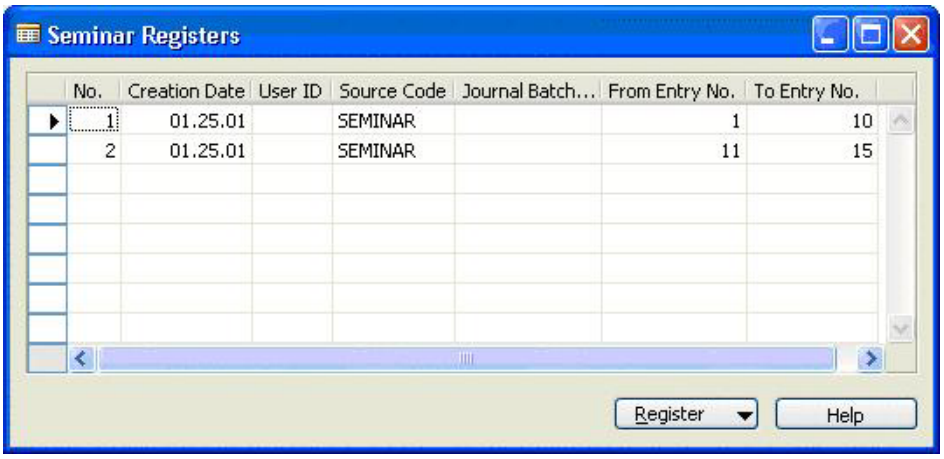


FIGURE 4–6: THE SEMINAR REGISTERS FORM (123456722)

The **Posted Seminar Charges** form (123456739) shown in Figure 4-7 shows the posted seminar charges from a posted seminar registration.

FIGURE 4–7: POSTED SEMINAR CHARGES (FORM 123456739):

The **Posted Seminar Registration** form shown in Figure 4–8 shows the posted seminar registration header and line information.

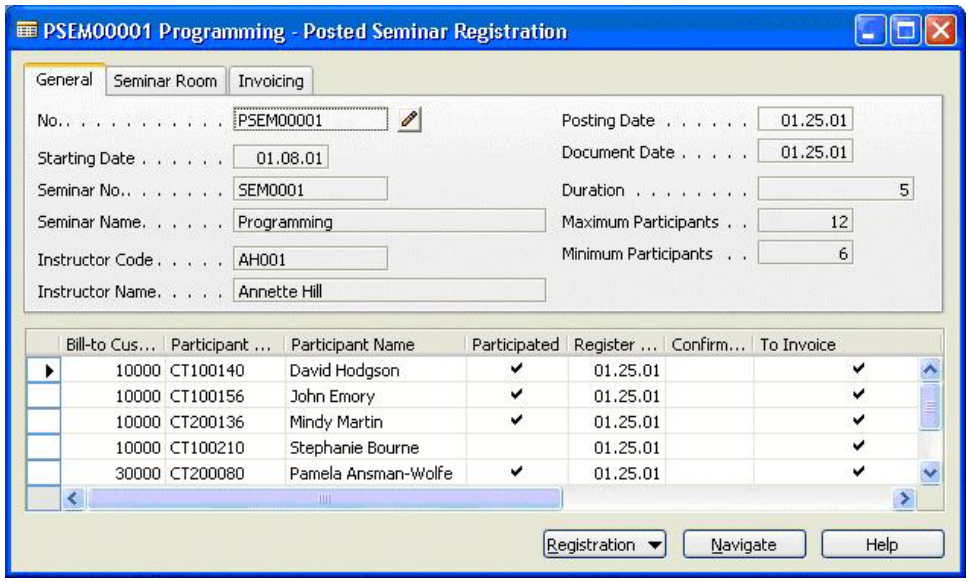


FIGURE 4–8: THE POSTED SEMINAR REGISTRATION FORM (123456734) GENERAL TAB

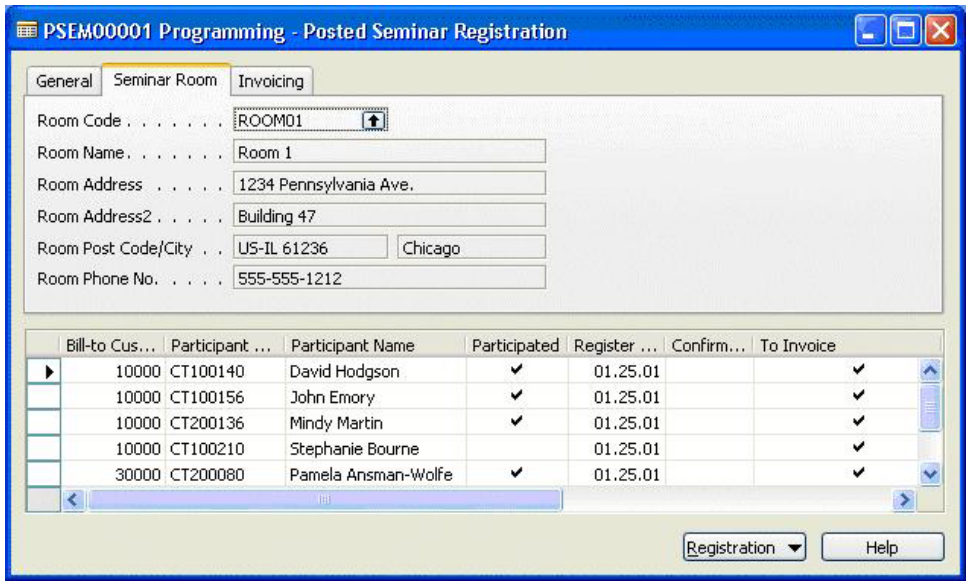


FIGURE 4–9: THE POSTED SEMINAR REGISTRATION FORM (123456734) SEMINAR ROOM TAB

The Posted Seminar Registration form (Figure 4-10) shows the posted seminar registration line information.

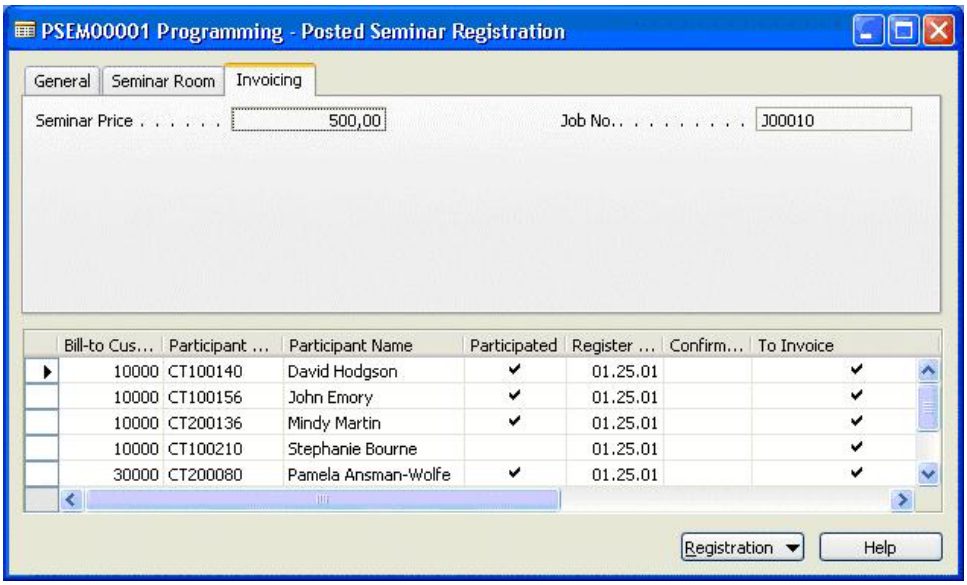


FIGURE 4–10: THE POSTED SEMINAR REGISTRATION FORM (123456734)
INVOICING TAB

The **Posted Seminar Reg. List form** displays a list of posted seminar registrations, as shown in Figure 4–11.

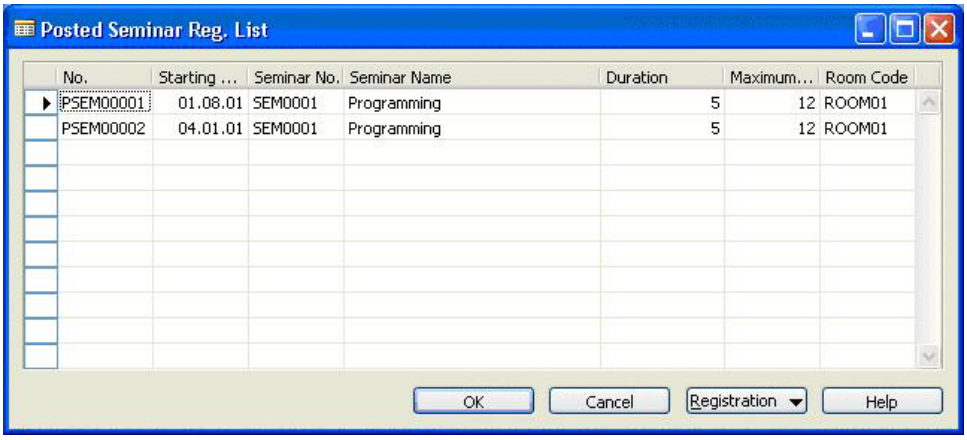


FIGURE 4–11: POSTED SEMINAR REG. LIST (FORM 123456736)

The **Seminar Registration** form (123456710) should be modified by adding a **Posting** button to it as shown in Figure 4–12.

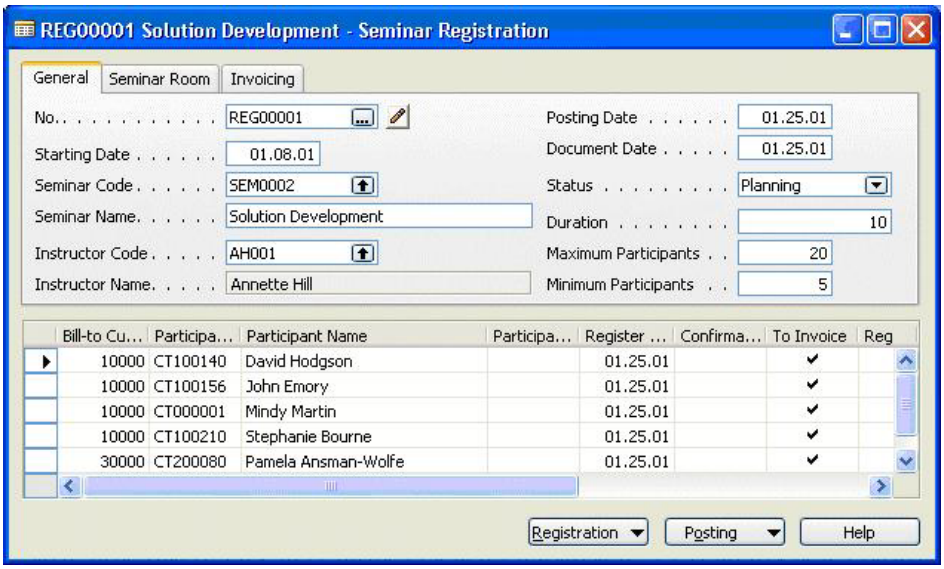


FIGURE 4–12: THE SEMINAR REGISTRATION FORM 123456710

The **Seminar Registration List** form (123456713) should be modified by adding a **Posting** button to it as shown in figure 4-13.

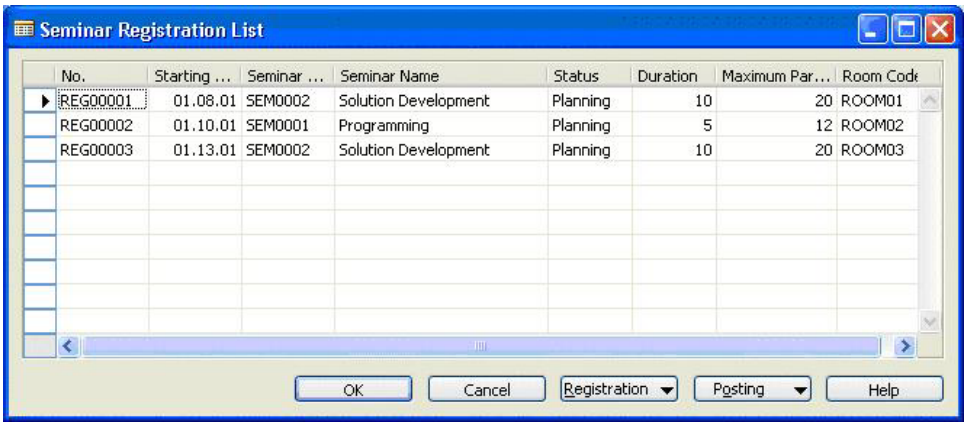


FIGURE 4–13: SEMINAR REGISTRATION LIST FORM (123456713)

Functional Design

As in all journal postings, the journal posting codeunits check **Seminar Journal** lines and post them. However, unlike some posting codeunits (such as **General Journal**), a codeunit is not needed to post a batch of journal lines, because posting batches is not required for this solution.

Check Line: This codeunit helps ensure the data validity of a seminar journal line before it is sent to the posting routine. The codeunit checks that the journal line is not empty and that there are values for the **Posting Date**, **Job No.**, **Instructor Code**, and **Seminar No.** Depending on whether the line is posting an **Instructor**, a **Room**, or a **Participant**, the codeunit checks that the key fields are not blank. The codeunit also verifies that the dates are valid.

Post Line: This codeunit performs the posting of the **Seminar Journal Line**. The codeunit creates a **Seminar Ledger Entry** per **Seminar Journal Line** and creates a **Seminar Register** to track which entries were created during the posting.

Modify the **Job Jnl.-Post** codeunit to ensure that the **Seminar Registration No.** is recorded in the **Job Ledger Entry**.

Two codeunits are necessary to enable the posting of the **Seminar Registration** document: **Seminar Post** and **Seminar Post (Y/N)**.

Seminar Post: This codeunit does the posting of an entire seminar registration, including the job posting and seminar posting. The codeunit transfers the comment records to new comment records corresponding to the posted document, and it copies charges to new tables containing posted charges. The codeunit creates a new **Posted Seminar Registration Header** record as well as **Posted Seminar Registration** Lines. The codeunit then runs the job journal posting, and it posts seminar ledger entries for each participant, for the instructor, and for the room. Finally, the codeunit deletes the records from the transaction tables, including the header, lines, comment lines, and charges.

Seminar Post (Y/N): This codeunit interacts with users, asking whether they really want to post the registration. If users answer Yes, the codeunit runs the **Seminar Post** codeunit.

Table Design

The following tables are required to implement the posting routines:

- Table 123456731 Seminar Journal Line
- Table 123456732 Seminar Ledger Entry
- Table 123456733 Seminar Register
- Table 123456721 Posted Seminar Charge
- Table 123456719 Posted Seminar Reg. Line
- Table 123456718 Posted Seminar Reg. Header

Add fields to the following Microsoft Dynamics NAV tables:

- Table 210 Job Journal Line
- Table 169 Job Ledger Entry
- Table 242 Source Code Setup

Lab 4.1 - Creating the Tables and Forms for Seminar Registration Posting

Follow these steps to create the tables and forms required for seminar registration posting:

1. Source codes are used in posting tables to identify where entries originated. Add the additional field to table 242 Source Code Setup as follows:

No.	Field Name	Type	Length	Comment
123456700	Seminar	Code	10	Relation to the Source Code table.

2. Modify the **Source Code Setup** form (279) by adding the new **Seminar** field as shown in the GUI Design section.

3. Create the **Seminar Journal Line** table (123456731) with the following fields:

No.	Field Name	Type	Length	Comment
1	Journal Template Name	Code	10	
2	Line No.	Integer		
3	Seminar No.	Code	20	Relation to Seminar table.
4	Posting Date	Date		
5	Document Date	Date		
6	Entry Type	Option		Options: Registration, Cancellation.
7	Document No.	Code	20	
8	Description	Text	50	
9	Bill-to Customer No.	Code	20	Relation to Customer table.
10	Charge Type	Option		Options: Instructor, Room, Participant, Charge.

No.	Field Name	Type	Length	Comment
11	Type	Option		Options: Resource,G/L Account
12	Quantity	Decimal		DecimalPlaces = 0:5
13	Unit Price	Decimal		AutoFormatType = 2
14	Total Price	Decimal		AutoFormatType = 1
15	Participant Contact No.	Code	20	Relation to Contact table.
16	Participant Name	Text	50	
17	Chargeable	Boolean		Initial value is Yes.
18	Room Code	Code	10	Relation to Seminar Room table.
19	Instructor Code	Code	10	Relation to Instructor table.
20	Starting Date	Date		
21	Seminar Registration No.	Code	20	
22	Job No.	Code	20	Relation to Job table.
23	Job Ledger Entry No.	Integer		Relation to Job Ledger Entry table.
24	Source Type	Option		Options: , Seminar.
25	Source No.	Code	20	If Source Type=Seminar, relation to Seminar table.
26	Journal Batch Name	Code	10	
27	Source Code	Code	10	Relation to Source Code table.
28	Reason Code	Code	10	Relation to Reason Code table.
29	Posting No. Series	Code	10	Relation to No. Series table.

4. The primary key for this table is Journal Template Name, Journal Batch Name, Line No.

5. Create a new function in the **Seminar Journal Line** table (123456731) with a return type of **Boolean**, called **EmptyLine**. Enter one line of code in the function trigger so that the function returns **TRUE** if the **Seminar No**. is blank.

6. Create the **Seminar Ledger Entry** table (123456732) with the following fields:

No.	Field Name	Type	Length	Comment
1	Entry No.	Integer		
2	Seminar No.	Code	20	Relation to Seminar table.
3	Posting Date	Date		
4	Document Date	Date		
5	Entry Type	Option		Options: Registration, Cancellation.
6	Document No.	Code	20	
7	Description	Text	50	
8	Bill-to Customer No.	Code	20	Relation to Customer table.
9	Charge Type	Option		Options: Instructor, Room, Participant, Charge.
10	Type	Option		Options: Resource,G/L Account.
11	Quantity	Decimal		DecimalPlaces = 0:5
12	Unit Price	Decimal		AutoFormatType = 2
13	Total Price	Decimal		AutoFormatType = 1
14	Participant Contact No.	Code	20	Relation to Contact table.
15	Participant Name	Text	50	
16	Chargeable	Boolean		Initial value is Yes.
17	Room Code	Code	10	Relation to Seminar Room table.
18	Instructor Code	Code	10	Relation to Instructor table.
19	Starting Date	Date		
20	Seminar Registration No.	Code	20	
21	Job No.	Code	20	Relation to Job table.

No.	Field Name	Type	Length	Comment
22	Job Ledger Entry No.	Integer		Relation to Job Ledger Entry table.
23	Remaining Amount	Decimal		FlowField; CalcFormula looks up the Remaining Amount in the corresponding Job Ledger Entry. AutoFormatType = 1. Must not be editable.
24	Source Type	Option		Options: , Seminar.
25	Source No.	Code	20	If Source Type=Seminar, relation to Seminar table.
26	Journal Batch Name	Code	10	
27	Source Code	Code	10	Relation to Source Code table.
28	Reason Code	Code	10	Relation to Reason Code table.
29	No. Series	Code	10	Relation to No. Series table.
30	User ID	Code	20	Relation to User table. Table relation should not be tested.

7. Enter code in the appropriate trigger so that when the user performs a lookup on the **User ID** field, the program runs the **LookupUserID** function from the **LoginManagement** codeunit.
8. Set the properties to specify form 123456721 as the lookup and drill down form for table 123456732.
9. The primary key for table 123456732 is **Entry No**., with one secondary key of **Seminar No**., **Posting Date**, and a second secondary key of **Bill-to Customer No**., **Seminar Registration No**., **Charge Type**, **Participant Contact No**.
10. Create the **Seminar Register** table (123456733) with the following fields:

No.	Field Name	Type	Length	Comment
1	No.	Integer		
2	From Entry No.	Integer		Relation to Seminar Ledger Entry table.
3	To Entry No.	Integer		Relation to Seminar Ledger Entry table.
4	Creation Date	Date		

No.	Field Name	Type	Length	Comment
5	Source Code	Code	10	Relation to Source Code table.
6	User ID	Code	20	Relation to User table. Table relation should not be tested.
7	Journal Batch Name	Code	10	

11. The primary key for table 123456733 is **No.**, with one secondary key of **Creation Date** and a second secondary key of **Source Code**, **Journal Batch Name**, **Creation Date**.

12. Enter code in the appropriate trigger so that when the user performs a lookup on the **User ID** field, the program runs the **LookupUserID** function from the **LoginManagement** codeunit.

13. Set the properties to specify form 123456722 as the lookup and drill down form for table 123456733.

14. Create the Seminar Ledger Entries form (123456721) with the fields Posting Date, Document No., Document Date (not visible), Entry Type, Seminar No., Description, Bill-to Customer No., Charge Type, Type, Quantity, Unit Price, Total Price, Remaining Amount, Chargeable, Participant Contact No., Participant Name, Instructor Code, Starting Date, Seminar Registration No., Job No., Job Ledger Entry No., and Entry No. as shown in the GUI Design section. Set the property to make this form not editable.

With the new tables, the codeunits to post seminar-specific information from the seminar journal to the seminar ledger can be created. These codeunits are similar to other posting codeunits, except that a **Post Batch** codeunit for the document posting is not needed.

Lab 4.2 - Creating the Codeunits and Form for Seminar Journal Posting

First create a codeunit to show the **Seminar Ledger Entry** form for a set of entries:

1. Create the **Seminar Reg.-Show Ledger** codeunit (123456734). Set the property to specify **Seminar Register** as the source table for this codeunit.
2. Enter code in the appropriate trigger so that when the program runs the codeunit, it runs the **Seminar Ledger Entries** form, showing only those entries between the **From Entry No**. and **To Entry No.** on the **Seminar Register**.

HINT: *Look at Codeunit 275 Res.Reg.-Show Ledger.*

3. The next step is to create the **Check Line** codeunit to help ensure data validity before posting. Create the **Seminar Jnl.-Check Line** codeunit (123456731). Set the property to specify **Seminar Journal Line** as the source table for this codeunit.
4. In codeunit 123456731, create a function called **RunCheck** that takes a parameter that is passed by reference. This parameter is a record variable of the **Seminar Journal Line** table, called **SemJnlLine**.
5. Enter code in the appropriate trigger so that when the program runs codeunit 123456731, it runs the **RunCheck** function for the current record.
6. Enter code in the **RunCheck** function trigger so that the function performs the following tasks:

HINT: *Look at the RunCheck function in Codeunit 211 Res.Jnl.-Check Line.*

   - Tests whether the **Seminar Journal Line** is empty using the EmptyLine function. If the line is empty, the function exits.
   - Tests that the **Posting Date**, **Job No.**, and **Seminar No.** fields are not empty.
   - Depending on the value of the **Charge Type**, tests that the **Instructor Code**, **Room Code**, and **Participant Contact No**. are not empty.

- If the line is **Chargeable**, tests that the **Bill-to Customer No**. is not blank.
- Shows an error if the **Posting Date** is a closing date.
- Tests that the **Posting Date** is between the **Allow Posting From** and **Allow Posting To** dates on the user's User Setup record; if those fields are empty on the User Setup record, tests that the **Posting Date** is between the **Allow Posting From** and **Allow Posting To** dates on the G/L Setup record. The function shows an error if the **Posting Date** is not between the dates on either of these records.
- Shows an error if the **Document Date** is a closing date.

7. Now create the Post Line codeunit to post Seminar Journal lines. Create the Seminar Jnl.-Post Line codeunit (123456732).

   - Set the property to specify **Seminar Journal Line** as the source table for this codeunit.
   - Define the following global variables for the codeunit:

Name	Data Type	Subtype	Length
SemJnlLine	Record	Seminar Journal Line	
SemLedgEntry	Record	Seminar Ledger Entry	
SemReg	Record	Seminar Register	
SemJnlCheckLine	Codeunit	Seminar Jnl.-Check Line	
NextEntryNo	Integer		

8. Create a function called **GetSemReg** that takes as a parameter a record variable for the **Seminar Register** table, called **NewSemReg**. This parameter is passed by reference.
9. Create a function called **RunWithCheck** that takes as a parameter a record variable for the **Seminar Journal Line**, called **SemJnlLine2**, which is passed by reference.
10. Create a function called **Code**.
11. Enter code in the appropriate trigger so that when the program runs codeunit 123456732 **Seminar Jnl.-Post Line**, it runs the **RunWithCheck** function for the current record.
12. Enter code in the **GetSemReg** function trigger so that the function sets the **NewSemReg** parameter to the **SemReg** record.

13. Enter code in the **RunWithCheck** function trigger so that the function copies the **SemJnlLine** from the **SemJnlLine2** record, runs the **Code** trigger, and copies the **SemJnlLine2** record from the **SemJnlLine**.

14. Enter code in the **Code** function trigger so that the function performs the following tasks:

HINT: *Look at the Code function in Codeunit 212 Res. Jnl.-Post Line.*

- Tests whether the **SemJnlLine** is empty using the **EmptyLine** function. If it is empty, the function exits.
- Runs the **RunCheck** function of the **SemJnlCheckLine** codeunit.
- If the **NextEntryNo** is 0, the function locks the **SemLedgEntry** table and sets the **NextEntryNo** to one more than the **Entry No.** of the last record in the **SemLedgEntry** table.
- If the **Document Date** is empty, the function sets the **Document Date** to the **Posting Date**.
- Creates or updates the **SemReg** record, depending on whether or not the register record has been created for this posting. The function does this by checking whether the **No.** of the **SemReg** record is 0. If so, the function locks the **SemReg** table. If the function either cannot find the last record of the **SemReg** table, or if the **To Entry No**. is not 0, the function creates a new **SemReg** record, fills the fields as appropriate, and inserts the new record. Regardless of whether the function creates a new record or not, the function sets the **To Entry No**. for the record to the **NextEntryNo** and modifies the record.
- Creates a new **SemLedgEntry** record, fills the fields as appropriate from the **SemJnlLine** record, sets the **Entry No.** to **NextEntryNo**, inserts the new record, and increments **NextEntryNo** by 1.

15. Now create the Seminar Registers form from which the seminar registers can be viewed. Create a Seminar Registers form (123456722) with fields No., Creation Date, User ID, Source Code, Journal Batch Name, From Entry No., and To Entry No. as shown in the GUI Design section.

    – Set the property to specify that this form is not editable.
    – Add a menu button and menu item to the form as follows:

Menu Button	Option	Comment
Register	Seminar Ledger	Runs the codeunit 123456734 Seminar Reg.-Show Ledger.

One of the requirements specified by the client's functional requirements was that the seminar registration should post as a job with additional seminar-specific information. The journals and codeunits to cover the seminar-specific information have been created, so now it is necessary to modify the tables, forms, and codeunits for posting job journals to allow the posting of seminar registrations as jobs.

Lab 4.3 - Modifying the Tables, Forms and Codeunits for Job Posting

Follow these steps to make the neceessary changes to the Seminar solution objects for performing Job Posting:

1. Add a new field to the **Job Journal Line** table (210) as follows:

No.	Field Name	Type	Length	Comment
123456700	Seminar Registration No.	Code	20	

2. Add a new field to the **Job Ledger Entry** table (169) as follows:

No.	Field Name	Type	Length	Comment
123456700	Seminar Registration No.	Code	20	

3. In codeunit 202 **Job Jnl.-Post Line**, enter code into the **Code** function trigger so that when the function is filling the **JobLedgEntry** fields, it fills the ledger's **Seminar Registration No**. field from the **Job Journal Line**.

Lab 4.4 - Creating the Tables and Forms for Posted Information

Now create the tables and forms that will be used to store posted seminar registration information:

1. Create the **Posted Seminar Charge** table (123456721) with the following fields:

No.	Field Name	Type	Length	Comment
1	Seminar Registration No.	Code	10	Relation to table 123456718. Must not be blank.
2	Line No.	Integer		
3	Job No.	Code	20	Relation to Job table.
4	Type	Option		Options: Resource, G/L Account.
5	No.	Code	20	If Type=Resource, relation to the Resource table. If Type=G/L Account, relation to the G/L Account table.
6	Description	Text	50	
7	Quantity	Decimal		Decimal Places=0:5
8	Unit Price	Decimal		AutoFormatType = 2 Minimum value is 0.
9	Total Price	Decimal		AutoFormatType=1 Must not be editable.
10	To Invoice	Boolean		Initial value is Yes.
11	Bill-to Customer No.	Code	20	Relation to Customer table.
12	Unit of Measure Code	Code	10	If Type=Resource, relation to the **Code** field of the Resource Unit of Measure table, where the Resource No. = No.; otherwise, relation to the Unit of Measure table.

No.	Field Name	Type	Length	Comment
13	Gen. Prod. Posting Group	Code	10	Relation to Gen. Product Posting Group table.
14	VAT Prod. Posting Group	Code	10	Relation to VAT Product Posting Group table.
15	Qty. per Unit of Measure	Decimal		
16	Registered	Boolean		Must not be editable.

   - The primary key for the **Posted Seminar Charge** table (123456721) is **Seminar Registration No**., **Line No.**, with a secondary key of Job No.

2. Create form 123456739 **Posted Seminar Charges** with the fields **Type**, **No.**, **Description**, **Bill-to Customer No**., **To Invoice**, **Unit of Measure Code**, **Quantity**, **Unit Price**, and **Total Price** as shown in the GUI Design section. Set the property to specify that the form is not editable.

3. Create the **Posted Seminar Reg. Line** table (123456719**)** with the following fields:

No.	Field Name	Type	Length	Comment
1	Document No.	Code	20	Relation to table 123456718 Posted Seminar Reg. Header.
2	Line No.	Integer		
3	Bill-to Customer No.	Code	20	Relation to Customer table.
4	Participant Contact No.	Code	20	Relation to Contact table.
5	Participant Name	Text	50	Flowfield based on the Participant Contact No. Must not be editable.
6	Register Date	Date		Must not be editable.
7	To Invoice	Boolean		Initial value is Yes.
8	Participated	Boolean		
9	Confirmation Date	Date		Must not be editable.

No.	Field Name	Type	Length	Comment
10	Seminar Price	Decimal		AutoFormatType = 2
11	Line Discount %	Decimal		Decimal places 0:5. The minimum value is 0, and the maximum is 100.
12	Line Discount Amount	Decimal		AutoFormatType = 1
13	Amount	Decimal		AutoFormatType = 1
14	Registered	Boolean		Must not be editable.

4. The primary key for this table is **Document No.**, **Line No**.

5. Create the **Posted Seminar Reg. Header** table (123456718) with the following fields:

No.	Field Name	Type	Length	Comment
1	No.	Code	20	
2	Starting Date	Date		
3	Seminar No.	Code	10	Relation to the Seminar table.
4	Seminar Name	Text	50	
5	Instructor Code	Code	10	Relation to Instructor table.
6	Instructor Name	Text	50	FlowField; The CalcFormula should look up the Name field on the Instructor table. Must not be editable.
7	Duration	Decimal		DecimalPlaces = 0:1
8	Maximum Participants	Integer		
9	Minimum Participants	Integer		

No.	Field Name	Type	Length	Comment
10	Room Code	Code	20	Relation to Seminar Room table.
11	Room Name	Text	30	
12	Room Address	Text	30	
13	Room Address2	Text	30	
14	Room Post Code	Code	20	Relation to Post Code table. There should be no validation or testing of the table relation.
15	Room City	Text	30	
16	Room Phone No.	Text	30	
17	Seminar Price	Decimal		AutoFormatType=1
18	Gen. Prod. Posting Group	Code	10	Relation to Gen. Product Posting Group table.
19	VAT Prod. Posting Group	Code	10	Relation to VAT Product Posting Group table.
20	Comment	Boolean		FlowField; The CalcFormula should check whether lines exist on the Seminar Comment Line table for the current Posted Seminar Registration. Must not be editable.
21	Posting Date	Date		
22	Document Date	Date		
23	Job No.	Code	20	Relation to Job table.
24	Reason Code	Code	10	Relation to Reason Code table.
25	No. Series	Code	10	Relation to No. Series table.
26	Registration No. Series	Code	10	Relation to No. Series table.
27	Registration No.	Code	20	
29	User ID	Code	20	Relation to User table. Relation should not be tested.
30	Source Code	Code	10	Relation to Source Code table.

- The primary key for the **Posted Seminar Reg. Header** table (123456718) is **No.** with a secondary key of **Room Code**. The sum index field for the secondary key is **Duration**.

Set the property to specify form 123456736 as the lookup form for the table.

NOTE: *The field numbers in the Posted Seminar Reg. Header are set to match those of the Seminar Registration Header table, even though they are not using all of the same fields. This is so the TRANSFERFIELDS function, which relies on* ***Field No****., can be used when copying.*

---

6. Create the Posted Seminar Reg. Subform form (123456735) with the fields Bill-to Customer No., Participant Contact No., Participant Name, Participated, Register Date, Confirmation Date, To Invoice, Registered, Seminar Price, Line Discount %, Line Discount Amount, and Amount as shown in the GUI Design section.

   - Set the width, height, and positioning properties for the subform so that there is no empty space around the table box.
   - Set the properties for the **Line Discount %** and **Line Discount Amount** so that they are blank if the value is 0.
   - Set the property to specify that the form is not editable.
   - Set the property to specify that the program automatically creates a new key when a new line is inserted.

7. Create the **Posted Seminar Registration** form (123456734) as shown in the GUI Design section.

   - Set the property to specify that the form is not editable.
   - Add a subform control to the form and set the control properties so that the subform control is the same size as the subform form. Set the subform properties to specify an ID for the form and the proper link to the table.

8. Add a menu button and menu items to form 123456734 as follows:

Menu Button	Option	Comment
Registration	List (f5)	Opens the lookup form.
	Comments	Opens the **Seminar Comment Sheet** form (123456706) showing the corresponding records. The link should run when the form is updated.
	<Separator>	
	Charges	Opens the **Posted Seminar Charges** form (123456739) showing the corresponding records. The link should run when the form is updated.

   – Add a command button next to the **No.** field to provide access to the Comment Sheet for the corresponding record. Set the RunFormLink property for this button so that only the comments corresponding to the selected Posted Seminar Registration are displayed.

HINT: *Copy the command button and picture box from the existing form 21 Customer Card and paste them into the new form.*

   – Enter code in the appropriate trigger so that after the form gets the record, the program releases the filter on the **No.** field of the **Posted Seminar Header** table.

9. Create form 123456736 Posted Seminar Reg. List with the fields No., Starting Date, Seminar No., Seminar Name, Duration, Maximum Participants, and Room Code as shown in the GUI Design section.

   – Set the property to specify that the form is not editable.
   – Add the menu button and menu item to the form as follows:

Menu Button	Option	Comment
Registration	Card (shift + f5)	Opens the form 123456734 Posted Seminar Registration for the selected record.

Lab 4.5 - Creating the Codeunits for Document Posting

As shown in the client's functional requirements, two codeunits are needed to handle the document posting. The first is the codeunit that actually does the work of generating journal lines and running the posting routine, and the second is the codeunit that interacts with the user.

The first codeunit will be the Seminar-Post codeunit:

1. Create the **Seminar-Post** codeunit (123456700).
2. Define the following global variables for the codeunit:

Name	DataType	Subtype	Length
SemRegHeader	Record	Seminar Registration Header	
SemRegLine	Record	Seminar Registration Line	
PstdSemRegHeader	Record	Posted Seminar Reg. Header	
PstdSemRegLine	Record	Posted Seminar Reg. Line	
SemCommentLine	Record	Seminar Comment Line	
SemCommentLine 2	Record	Seminar Comment Line	
SemCharge	Record	Seminar Charge	
PstdSemCharge	Record	Posted Seminar Charge	
SemRoom	Record	Seminar Room	
Instr	Record	Instructor	
Job	Record	Job	
Res	Record	Resource	
Cust	Record	Customer	
JobLedgEntry	Record	Job Ledger Entry	
SemLedgEntry	Record	Seminar Ledger Entry	
JobJnlLine	Record	Job Journal Line	
SemJnlLine	Record	Seminar Journal Line	

Name	DataType	Subtype	Length
SourceCodeSetup	Record	Source Code Setup	
JobJnlPostLine	Codeunit	Job Jnl.-Post Line	
SemJnlPostLine	Codeunit	Seminar Jnl.-Post Line	
NoSeriesMgt	Codeunit	NoSeriesManagement	
ModifyHeader	Boolean		
Window	Dialog		
SrcCode	Code		10
LineCount	Integer		
JobLedgEntryNo	Integer		
SemLedgEntryNo	Integer		

3. Set the property to specify **Seminar Registration Header** as the source table for this codeunit.
4. Create a function called **CopyCommentLines** and set the property for this function to specify it as a local function. This function has the following parameters:
   - An integer variable called **FromDocumentType**
   - An integer variable called **ToDocumentType**
   - A code variable with length 20 called **FromNumber**
   - A code variable with length 20 called **ToNumber**
5. Create a function called **CopyCharges** and set the property to specify it as a local function. This function has two parameters:
   - A code variable with length 20 called **FromNumber**
   - A code variable with length 20 called **ToNumber**
6. Create a function called **PostJobJnlLine** with a return type of **Integer**. Set the property to specify it as a local function. This function has one parameter of an option variable called **ChargeType** with the options **Participant**, **Charge**.
7. Create a function called **PostSeminarJnlLine** with a return type of integer. Set the property to specify it as a local function. This function has one parameter of an option variable called **ChargeType** with the options: **Instructor**, **Room**, **Participant**, and **Charge**.
8. Create a function called **PostCharge** and set the property to specify it as a local function.

9. Enter code in the **CopyCommentLines** function trigger so that this function finds records in the **Seminar Comment Line** table that correspond to the **FromDocumentType** and **FromNumber** values that were passed as parameters. For each record the function finds, the function creates a new **Seminar Comment Line** record that is a copy of the old record, except that the function sets the **Document Type** and **No.** to the **ToDocumentType** and **ToNumber**.

10. Enter code in the **CopyCharges** function trigger so that the function finds all **Seminar Charge** records that correspond to the **FromNumber**. For each record found, the function transfers the values to a new **Posted Seminar Charge** record, using the **ToNumber** as the **Seminar Registration No**.

11. Enter code in the **PostJobJnlLine** function trigger so that the function performs the following tasks:

    - Gets the **Instructor**, **Resource**, and Customer records that correspond to the **Seminar Registration Header** record.
    - Creates a new **Job Journal Line** record and fills the fields appropriately. The **Gen. Bus. Posting Group** comes from the Customer record. The **Entry Type** is Usage. The **Document No.** and the **Seminar Registration No**. are both the **No.** from the Posted Seminar Reg. Header. The **Source Code** is the value in the SrcCode variable. The **Source Currency Total Cost** is the Seminar Price from the Seminar Registration Line.
    - If the **ChargeType** is Participant, certain fields are filled as follows: the **Description** in the **Job Journal Line** is the participant's name, the **No.** is the instructor's **Resource No**., **Chargeable** is the **To Invoice** value from the registration line, the quantity fields are 1, the cost fields are 0, and the price fields are taken from the **Amount** field on the registration line.
    - If the **ChargeType** is Charge, certain fields are filled as follows: the **Description** in the Job Journal Line is the **Description** from the **Seminar Charge** record. If the **Type** from the **Seminar Charge** is **Resource**, the journal line's **Type** is **Resource** and the **Unit of Measure Code** and **Qty. per Unit of Measure** are the corresponding values from the **Seminar Charge** record. If the **Type of the Seminar Charge** is **G/L Account**, the journal line's **Type** is **G/L Account**, **Chargeable** is the **To Invoice** value on the Seminar Charge record, the **Quantity (Base)** is 1, the **Unit Cost** is 0, the **Total Cost** is 0, and the **No.**, **Quantity**, **Unit Price**, and **Total Price** are the corresponding values from the Seminar Charge record.
    - Runs the **Job Jnl.-Post Line** codeunit with the newly created **Job Journal Line**.
    - Exits and returns the **Entry No.** of the last **Job Ledger Entry** record in the table.

12. Enter code in the **PostSeminarJnlLine** function trigger so that the function performs the following tasks:
    - Creates a new **Seminar Journal Line** and fills the fields as appropriate from the **Seminar Registration Header**, **Posted Seminar Reg. Header**, and parameter values.
    - If the **ChargeType** is Instructor, certain fields are filled as follows: the **Description** is the instructor's Name, the **Type** is Resource, **Chargeable** is FALSE, and the **Quantity** is the Duration from the registration header.
    - If the **ChargeType** is Room, certain fields are filled as follows: the **Description** is the room's Name, the **Type** is Resource, **Chargeable** is FALSE, and the **Quantity** is the Duration from the registration header.
    - If the **ChargeType** is Participant, certain fields are filled as follows: the **Bill-to Customer No**., **Participant Contact No**., and **Participant Name** values come from the corresponding fields on the registration line, the **Description** is the **Participant Name** on the registration line, the **Type** is Resource, **Chargeable** comes from the **To Invoice** field on the registration line, and **Quantity** is 1. The **Unit Price** and **Total Price** come from the **Amount** field on the registration line.
    - If the **ChargeType** is **Charge**, certain fields are filled as follows: the **Description**, **Bill-to Customer No**., **Type**, **Quantity**, **Unit Price**, and **Total Price** values all come from the corresponding fields on the **Seminar Charge** record. **Chargeable** comes from To Invoice on the **Seminar Charge** record.
    - Runs the **Seminar Jnl.-Post Line** codeunit with the newly created **Seminar Journal Line**.
    - Exits with the **Entry No.** of the last **Seminar Ledger Entry** record in the table.

13. Enter code in the **PostCharge** function trigger so that the function performs the following tasks:
    - For each record in the **Seminar Charge** table that corresponds with the **Seminar Registration Header**, the function sets the **JobLedgEntryNo** to the result of the **PostJobJnlLine** function (run with a parameter of 1) and runs the **PostSeminarJnlLine** function (with a parameter of 3).
    - Sets the **JobLedgEntryNo** to 0.

14. Enter code in the appropriate trigger so that when the program runs the **Seminar-Post** codeunit, the codeunit performs the following tasks:

    - Clears all variables and sets the **SemRegHeader** variable to the current record.
    - Tests that the **Posting Date**, **Document Date**, **Starting Date**, **Seminar Code**, **Duration**, **Instructor Code**, **Room Code**, and **Job No.** fields on the registration line are not empty. Tests that the **Status** is **Closed**.
    - Gets the **Seminar Room** and **Instructor** records that correspond to the registration header and tests for both of them that the **Resource No.** field is not empty.
    - Gets the **Seminar Registration Line** records that correspond to the registration header and shows an error message if no records are found.
    - Opens a dialog window to keep the user informed of the progress.
    - If the **Posting No.** is blank on the registration header, tests that the **Posting No. Series** is not blank and runs the GetNextNo function of the **NoSeriesManagement** codeunit.
    - Modifies the header if necessary and performs a commit.
    - If record level locking applies, locks the **Seminar Registration Line** and **Seminar Ledger Entry** tables and finds the last **Seminar Ledger Entry** record.
    - Sets the **SrcCode** variable to the **Seminar** value in the **Source Code Setup** table.
    - Creates a **new Posted Seminar Reg. Header** record and transfers fields from the **Seminar Registration Header** record to the new record.

***HINT**: Use the TRANSFERFIELDS function to do this.*

    - In the Posted Seminar Reg. Header record, sets the **No.** to the **Posting No.** from the registration header, sets the **Registration No. Series** to the **No. Series** of the registration header, and sets the **Registration No.** to the **No.** of the registration header.
    - Updates the dialog window.
    - Sets the **Source Code** and **User ID** fields on the **Posted Seminar Reg. Header** record before inserting the record.
    - Runs the **CopyCommentLines** and **CopyCharges** functions.
    - Resets the filter on the **Seminar Registration Line** table to get the records corresponding to **the Seminar Registration Header** record.

- For each registration line found, the function updates the dialog window with an updated line count. The function then tests that the **Bill-to Customer No.** and **Participant Contact No**. are not blank. If **To Invoice** is FALSE, the function sets the **Seminar Price**, **Line Discount %**, **Line Discount Amount**, and **Amount** to 0. The function then sets the **JobLedgEntryNo** variable to the result of the **PostJobJnlLine** function and the SemLedgEntryNo variable to the result of the PostSeminarJnlLine function. The function then creates a new Posted Seminar Reg. Line record and transfers the fields from the Seminar Registration Line record to the new posted record. The function sets the **Document No**. field of the new record to the **No.** of the Posted Seminar Reg. Header. Finally, the function inserts the new record and sets the JobLedgEntryNo and SemLedgEntryNo variables to 0.
- Runs the PostCharge, PostSeminarJnlLine (to post the instructor), and PostSeminarJnlLine (to post the room) functions.
- If record level locking does not apply, locks the **Seminar Registration Line** table.
- Deletes the **Seminar Registration Header** and all corresponding **Seminar Registration Line** records, **Seminar Comment Line** records, and **Seminar Charge** records.
- Sets the current record to the **Seminar Registration Header** record.

15. Create codeunit 123456701 **Seminar-Post (Yes/No)**.
16. Define the following global variables for the codeunit:

Name	DataType	Subtype	Length
SemRegHeader	Record	Seminar Registration Header	
SeminarPost	Codeunit	Seminar-Post	

17. Set the property to specify **Seminar Registration Header** as the source table for this codeunit.
18. Create a function called **Code** and set the property to specify that it is a local function.
19. Enter code into the **Code** function trigger so that the function confirms that the user wants to post the registration.
    - If the user answers **No**, the function exits.
    - If the user answers **Yes**, the function runs the **Seminar-Post** codeunit with the **Seminar Registration Header** record.
    - The function performs a commit at the end.

20. Enter code into the appropriate trigger so that when the program runs this codeunit, the codeunit copies the current record into the **Seminar Registration Header** record, runs the **Code** function and copies **the Seminar Registration Header** into the current record.

21. Finally, modify two forms to enable the registration posting. Add a **Posting** menu button and menu item to form 123456710 Seminar Registration as follows:

Menu Button	Option	Comment
Posting	Post (F11)	Runs codeunit 123456701 Seminar-Post (Yes/No). There should be ellipses on this button.

22. Add a **Posting** menu button and menu item to form 123456713 Seminar Registration List as follows:

Menu Button	Option	Comment
Posting	Post (F11)	Runs the **Seminar-Post (Yes/No)** codeunit (123456701). There should be ellipses on this button.

Testing

It is assumed that some setup has been performed during previous test scripts and that there is some sample data of customers, contacts (participants), seminars, rooms, and instructors when testing posting. Follow these steps to test the solution

1. Some set-up is required to test the posting routine:
   - In the Navigation Pane under FINANCIAL MANAGEMENT→SETUP→TRAIL CODES, open the Source Code Setup window. The new **Seminar** field on the **Jobs** tab will appear.
   - Click the lookup to open the Source Codes window.
   - Enter a new source code of SEMJNL with a description of Seminar Journal and click **OK**.
   - Close the Source Code Setup window.

2. Under RESOURCE PLANNING→JOBS click **Jobs** to open the **Job Card**. Enter a new job to use with testing. Most of the fields do not matter, but on the **General** tab, the **Status** must be set to Order to be able to post. On the **Posting** tab, specify a **Job Posting Group**. Close the **Job Card**.

3. Select the **Seminar Registration** form (123456710) in the Object Designer and click **Run**. Enter a new **Seminar Registration**, filling out all fields and entering at least one line.

4. Select POSTING→POST to initiate the posting routine. Some errors may occur. Some will be due to the validation code, while others will likely be bugs in the code. Use the Debugger to find the sources of unexpected errors that occur when running the code.

5. When all errors are resolved, verify that the posting worked as expected. Select the **Posted Seminar Registration** form (123456734) in the Object Designer and click **Run**. Find the seminar registration you posted there, and verify that:

   – All header fields were copied over
   – All lines and fields within them were copied over
   – Any related charges appear in the Posted Seminar Charges window

6. Create another **Seminar Registration. Run** form (123456713) **Seminar Registration List** and try posting from there.

7. Open the **Job Card** for the new job and select **Ledger Entries** from the **Job** menu button. The records there with a Source Code of SEMJNL created from Seminar posting should appear.

8. Run the **Seminar Registers** form (123456722) from the Object Designer. The entries created from your posting tests should appear.

Conclusion

In this chapter, the following subjects were covered:

- Posting in Microsoft Dynamics NAV from journals and from transaction documents.
- The different tables and codeunits that make up a standard posting routine.
- Using the Microsoft Dynamics NAV debugger and code coverage functionality.
- What key aspects of programming to keep in mind to maximize performance.

The next step is to integrate the different aspects of the solution to integrate the solution into the standard Microsoft Dynamics NAV interface.

Test Your Knowledge

Review Questions

1. What three tables make up a journal?

2. In which type of table is permanent transaction data stored? Can these tables be modified directly?

3. Which function do you use if you want to ensure that the data in a table is not changed by another user until you finish writing to the table? Is it always necessary to use this function?

4. What are the three standard posting routine codeunits? What does each of these codeunits do and how do they interrelate?

5. When is a document posting routine used? How does a document posting routine interact with other posting routines?

6. What is the standard Microsoft Dynamics NAV shortcut for Posting?

Quick Interaction: Lessons Learned

Take a moment to note three key points you have learned from this chapter:

1.

2.

3.