PART III

Under the hood

CHAPTER 17

The database layer

In this chapter

Introduction

The Microsoft Dynamics AX 2012 application run time provides robust database features that make creating an enterprise resource planning (ERP) application much easier. Many new and powerful database features have been added to Microsoft Dynamics AX 2012. This chapter focuses on several of these new capabilities. The information provided here introduces the features, provides information about how to use them in an application, and, when appropriate, explains in detail how each feature works.

Many database features, such as optimistic concurrency control (OCC), transaction support, and the query system, have been available in Microsoft Dynamics AX for several releases. For detailed information about these and other database features in Microsoft Dynamics AX, see the "Database" section in the Microsoft Dynamics AX 2012 software development kit (SDK) at *http://msdn.microsoft.com/en-us/library/aa588039.aspx*.

You can also refer to previous editions of "*Inside Microsoft Dynamics AX*," which contain useful information about database functionality that still applies to Microsoft Dynamics AX 2012.

Temporary tables

By default, any table that is defined in the Application Object Tree (AOT) is mapped in a one-to-one relationship to a permanent table in the underlying relational database. Microsoft Dynamics AX also supports the functionality of temporary tables. In previous releases, Microsoft Dynamics AX provided

the capability to create *InMemory* temporary tables that are mapped to an indexed sequential access method (ISAM) file-based table that is available only during the run-time scope of the Application Object Server (AOS) or a client. Microsoft Dynamics AX 2012 provides a new type of temporary table that is stored in the TempDB database in Microsoft SQL Server.

InMemory temporary tables

The ISAM file that represents an *InMemory* temporary table contains the data and all of the indexes that are defined for the table in the AOT. Because working on smaller datasets is generally faster than working on larger datasets, the Microsoft Dynamics AX run time monitors the size of each *InMemory* temporary table. If the size is less than 128 kilobytes (KB), the temporary table remains in memory. If the size exceeds 128 KB, the temporary table is written to a physical ISAM file. Switching from memory to a physical file affects performance significantly. A file with the naming syntax *$tmp<8 digits>.$$$* is created when data is switched from memory to a physical file. You can monitor the threshold limit by noting when this file is created.

Although *InMemory* temporary tables don't map to a relational database, all of the data manipulation language (DML) statements in X++ are valid for tables that operate as *InMemory* temporary tables. However, the Microsoft Dynamics AX run time executes some of the statements in a downgraded fashion because the ISAM file functionality doesn't offer the same functionality as a relational database. For example, set-based operators always execute as record-by-record operations.

Using *InMemory* temporary tables

When you declare a record buffer for an *InMemory* temporary table, the table doesn't contain any records. You must insert records to work with the table. The *InMemory* temporary table and all of the records are lost when no declared record buffers point to the temporary dataset.

Memory and file space aren't allocated to the *InMemory* temporary table until the first record is inserted. The temporary table is located on the tier where the first record was inserted. For example, if the first insert occurs on the server tier, the memory is allocated on this tier, and eventually the temporary file will be created on the server tier.

Important Use temporary tables carefully to ensure that they don't cause excessive round trips between the client and the server, resulting in degraded performance. For more information, see Chapter 13, "Performance."

A declared temporary record buffer contains a pointer to the dataset. If you use two temporary record buffers, they point to different datasets by default, even though the table is of the same type. To illustrate this, the X++ code in the following example uses the TmpLedgerTable temporary table defined in Microsoft Dynamics AX 2012. The table contains four fields: *AccountName*, *AccountNum*, *CompanyId*, and *LedgerDimension*. The *AccountNum* and *CompanyId* fields are both part of a unique index, *AccountNumIdx*, as shown in Figure 17-1.

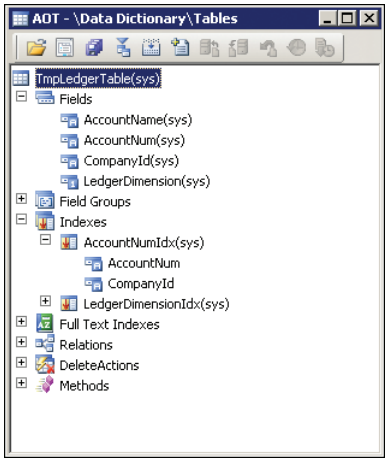


FIGURE 17-1 TmpLedgerTable temporary table.

The following X++ code shows how the same record can be inserted in two record buffers of the same type. Because the record buffers point to two different datasets, a "duplicate value in index" failure doesn't result, as it would if both record buffers pointed to the same temporary dataset, or if the record buffers were mapped to a database table.

```
static void TmpLedgerTable(Args _args)
{
    TmpLedgerTable tmpLedgerTable1;
    TmpLedgerTable tmpLedgerTable2;

    tmpLedgerTable1.CompanyId = 'dat';
    tmpledgerTable1.AccountNum = '1000';
    tmpLedgerTable1.AccountName = 'Name';
    tmpLedgerTable1.insert(); // Insert into tmpLedgerTable1's dataset.

    tmpLedgerTable2.CompanyId = 'dat';
    tmpledgerTable2.AccountNum = '1000';
    tmpLedgerTable2.AccountName = 'Name';
    tmpLedgerTable2.insert(); // Insert into tmpLedgerTable2's dataset.
}
```

To have the record buffers use the same temporary dataset, you must call the *setTmpData* method on the record buffer, as illustrated in the following X++ code. In this example, the *setTmpData* method is called on the second record buffer and is passed in the first record buffer as a parameter.

```
static void TmpLedgerTable(Args _args)
{
    TmpLedgerTable tmpLedgerTable1;
    TmpLedgerTable tmpLedgerTable2;

    tmpLedgerTable2.setTmpData(tmpLedgerTable1);

    tmpLedgerTable1.CompanyId = 'dat';
```

```
    tmpledgerTable1.AccountNum = '1000';
    tmpLedgerTable1.AccountName = 'Name';
    tmpLedgerTable1.insert(); // Insert into shared dataset.

    tmpLedgerTable2.CompanyId = 'dat';
    tmpledgerTable2.AccountNum = '1000';
    tmpLedgerTable2.AccountName = 'Name';
    tmpLedgerTable2.insert(); // Insert will fail with duplicate value.
}
```

The preceding X++ code fails on the second insert operation with a "duplicate value in index" error because both record buffers point to the same dataset. You would notice similar behavior if, instead of calling *setTmpData*, you simply assigned the second record buffer to the first record buffer, as illustrated here:

```
tmpLedgerTable2 = tmpLedgerTable1;
```

However, the variables would point to the same object, which means that they would use the same dataset.

When you want to use the *data* method to copy data from one temporary record buffer to another, where both buffers point to the same dataset, write the code for the copy operation as follows:

```
tmpLedgerTable2.data(tmpLedgerTable1);
```

Warning The connection between the two record buffers and the dataset is lost if the code is written as *tmpLedgerTable2 = tmpLedgerTable1.data*. In this case, the temporary record buffer points to a new record buffer that has a connection to a different dataset.

As mentioned earlier, if no record buffer points to the dataset, the records in the temporary table are lost, the allocated memory is freed, and the physical file is deleted. The following X++ code example illustrates this situation, in which the same record is inserted twice using the same record buffer. But because the record buffer is set to *null* between the two insert operations, the first dataset is lost, so the second insert operation doesn't result in a duplicate value in the index because the new record is inserted into a new dataset.

```
static void TmpLedgerTable(Args _args)
{
    TmpLedgerTable tmpLedgerTable;

    tmpLedgerTable.CompanyId = 'dat';
    tmpledgerTable.AccountNum = '1000';
    tmpLedgerTable.AccountName = 'Name';
    tmpLedgerTable.insert(); // Insert into first dataset.

    tmpLedgerTable = null; // Allocated memory is freed
                           // and file is deleted.
    tmpLedgerTable.CompanyId = 'dat';
    tmpledgerTable.AccountNum = '1000';
```

```
    tmpLedgerTable.AccountName = 'Name';
    tmpLedgerTable.insert(); // Insert into new dataset.
}
```

Notice that none of these *InMemory* temporary table examples uses the *ttsbegin*, *ttscommit*, and *ttsabort* statements. These statements affect only ordinary tables that are stored in a relational database. For example, the following X++ code adds data to an *InMemory* temporary table. Because the table is an *InMemory* temporary table, the value of the *accountNum* field is printed to the Infolog even though the *ttsabort* statement executes.

```
static void TmpLedgerTableAbort(Args _args)
{
    TmpLedgerTable tmpLedgerTable;

    ttsbegin;
    tmpLedgerTable.CompanyId = 'dat';
    tmpledgerTable.AccountNum = '1000';
    tmpLedgerTable.AccountName = 'Name';
    tmpLedgerTable.insert(); // Insert into table.
    ttsabort;

    while select tmpLedgerTable
    {
        info(tmpLedgerTable.AccountNum);
    }
}
```

To cancel the insert operations on the table in the preceding scenario successfully, you must call the *ttsbegin* and *ttsabort* methods on the temporary record buffer instead, as shown in the following example:

```
static void TmpLedgerTableAbort(Args _args)
{
    TmpLedgerTable tmpLedgerTable;

    tmpLedgerTable.ttsbegin();
    tmpLedgerTable.CompanyId = 'dat';
    tmpledgerTable.AccountNum = '1000';
    tmpLedgerTable.AccountName = 'Name';
    tmpLedgerTable.insert(); // Insert into table.
    tmpLedgerTable.ttsabort();

    while select tmpLedgerTable
    {
        info(tmpLedgerTable.AccountNum);
    }
}
```

When you work with multiple temporary record buffers, you must call the *ttsbegin*, *ttscommit*, and *ttsabort* methods on each record buffer because there is no correlation between the individual temporary datasets.

Considerations for working with *InMemory* temporary tables

When working with *InMemory* temporary tables, keep the following points in mind:

- When exceptions are thrown and caught outside the transaction scope, if the Microsoft Dynamics AX run time has already called the *ttsabort* statement, temporary data isn't rolled back. When you work with temporary datasets, make sure that you're aware of how the datasets are used both inside and outside the transaction scope.
- The database-triggering methods on temporary tables behave almost the same way as they do with ordinary tables, but with a few exceptions. When *insert*, *update*, and *delete* are called on the temporary record buffer, they don't call any of the database-logging or event-raising methods on the application class if database logging or alerts have been set up for the table.

Note In general, you can't set up logging or events on *InMemory* temporary tables that you define. However, because ordinary tables can be changed to temporary tables, logging or events might already be set up.

- Delete actions are also not executed on *InMemory* temporary tables. Although you can set up delete actions, the Microsoft Dynamics AX application run time doesn't try to execute them.
- Microsoft Dynamics AX lets you trace Transact-SQL statements, either from within the Microsoft Dynamics AX Windows client, or from the Microsoft Dynamics AX Configuration Utility or the Microsoft Dynamics AX Server Configuration Utility. However, Transact-SQL statements can be traced only if they are sent to the relational database. You can't trace data manipulation in *InMemory* temporary tables with these tools. However, you can use the Microsoft Dynamics AX Trace Parser to accomplish this. For more information, see the section, "Microsoft Dynamics AX Trace Parser" in Chapter 13.
- You can query a record buffer to find out whether it is acting on a temporary dataset by calling the *isTmp* record buffer method, which returns a value of *true* or *false* depending on whether the table is temporary.

TempDB temporary tables

The application code in Microsoft Dynamics AX often uses temporary tables for intermediate storage. This requires performing joins with regular tables and in some cases, executing set-based operations. But *InMemory* temporary tables provide restricted support for joins. These joins are performed by the data layer in the AOS, which does not give ideal performance. Also, as mentioned earlier, set-based operations are always downgraded to row-by-row operations for *InMemory* tables. *TempDB* temporary tables have been added to Microsoft Dynamics AX to provide a high-performance solution for these scenarios. Because these temporary tables are stored in the SQL Server database, database operations such as joins can be used.

TempDB temporary tables use the same X++ programming constructs as *InMemory* temporary tables. The key difference is that they are stored in the SQL Server *TempDB* database. The following code example shows the usage of a *TempDB* temporary table:

```
void select2Instances()
{
    TmpDBTable1 dbTmp1;
    TmpDBTable1 dbTmp2;

    dbTmp1.Field1 = 1;
    dbTmp1.Field2 = 'First';
    dbTmp1.insert();

    dbTmp2.Field1 = 2;
    dbTmp2.Field2 = 'Second';
    dbTmp2.insert();
    info("First Instance.");
    while select * from dbTmp1
    {
        info(strfmt("%1 - %2", dbTmp1.Field1, dbTmp1.Field2));
    }
    info("Second Instance.");
    while select * from dbTmp1
    {
        info(strfmt("%1 - %2", dbTmp2.Field1, dbTmp2.Field2));
    }
}
```

This example uses a table called TmpDBTable1 that contains two fields. The *TableType* property for the TmpDBTable1 table is set to *TempDB*. Similar to an *InMemory* temporary table, the *TempDB* temporary table is created only when the data is inserted into the table buffer. To see the data in the temporary table, insert a breakpoint before the first *select* statement in the X++ code, and then open SQL Server Management Studio (SSMS) to examine the tables that are created in the *TempDB* system database. Each table buffer instance for the temporary table has a corresponding table in the database in the following format: *t<table_id>_GUID*. In the example, the table ID of the TmpDBTable1 table is *101420*. This means that two tables were created in the *TempDB* database, with one of them having the table name t101420_E448847EACA4482997F4CD8BCAAAE0CE. After the method runs and the table buffers are destroyed, these two tables are truncated. The Microsoft Dynamics AX run time uses a pool to keep track of these tables in the *TempDB* database. The run time will reuse one of these table instances when a TmpDBTable1 table buffer is created again in X++.

Creating temporary tables

You can create temporary tables in the following ways:

- At design time by setting metadata properties.
- At configuration time by enabling licensed modules or configurations.
- At application run time by writing explicit X++ code.

The following sections describe each method.

Design time

To define a table as temporary, you must set the appropriate value in the *TableType* property for the table resource. By default, the *TableType* property is set to *Regular*. To create a temporary table, choose one of the other two options: *InMemory* or *TempDB*, as shown in Figure 17-2. The temporary tables are created in memory, and are backed by a file or created in the *TempDB* database when needed.

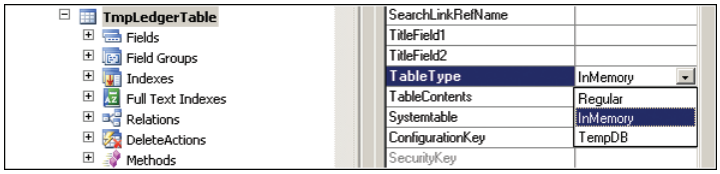


FIGURE 17-2 Marking a table as temporary at design time.

Tip Tables that you define as temporary at design time should have *Tmp* inserted as part of the table name instead of at the beginning or end of the name; for example, InventCostTmpTransBreakdown. This improves readability of the X++ code when temporary tables are explicitly used. In previous versions of Microsoft Dynamics AX, the best practice was to prefix temporary tables with *Tmp*, which is why a number of temporary tables still use this convention.

Configuration time

When you define a table by using the AOT, you can attach a configuration key to the table by setting the *ConfigurationKey* property on the table. The property belongs to the Data category of the table properties.

When the Microsoft Dynamics AX run time synchronizes the tables with the database, it synchronizes tables for all modules and configurations, regardless of whether they're enabled except the SysDeletedObjects configuration keys. Whether a table belongs to a licensed module or an enabled configuration depends on the settings in the *ConfigurationKey* property. If the configuration key is disabled, the table is disabled and behaves like a *TempDB* temporary table. Therefore, no run-time error occurs when the Microsoft Dynamics AX run time interprets X++ code that accesses tables that aren't enabled. For more information about the SysDeletedObjects configuration key, see "Best Practices: Tables" at *http://msdn.microsoft.com/en-us/library/aa876262.aspx*.

Note Whether it is enabled doesn't affect a table that is already defined as a temporary table. The table remains temporary even though its configuration key is disabled, and you can expect the same behavior regardless of the configuration key setting.

Run time

You can use X++ code to turn an ordinary table into an *InMemory* temporary table by calling the *setTmp* method on the record buffer. From that point forward, the record buffer is treated as though the *TableType* property on the table is set to *InMemory*.

Note You can't define a record buffer of a temporary table type and turn it into an ordinary table, partly because there is no underlying table in the relational database.

The following X++ code illustrates the use of the *setTmp* method, in which two record buffers of the same type are defined; one is temporary, and all records from the database are inserted into the temporary version of the table. Therefore, the temporary record buffer points to a dataset that contains a complete copy of all of the records from the database belonging to the current company.

```
static void TmpCustTable(Args _args)
{
    CustTable custTable;
    CustTable custTableTmp;

    custTableTmp.setTmp();
    ttsbegin;
    while select custTable
    {
        custTableTmp.data(custTable);
        custTableTmp.doInsert();
    }
    ttscommit;
}
```

Notice that the preceding X++ code uses the *doInsert* method to insert records into the temporary table. This prevents execution of the overridden *insert* method. The *insert* method inserts records in other tables that aren't switched automatically to temporary mode just because the *custTable* record buffer is temporary.

Caution Use great care when changing an ordinary record buffer to a temporary record buffer because application logic in overridden methods that manipulates data in ordinary tables could execute inadvertently. This can happen if the temporary record buffer is used in a form and the form application run time calls the database-triggering methods.

Surrogate keys

The introduction of surrogate keys is a significant change to table keys in Microsoft Dynamics AX 2012. A surrogate key is a system-generated, single-column primary key that does not carry domain-specific semantics. It is specific to the Microsoft Dynamics AX installation that generates the key. A surrogate key value cannot be generated in one installation and used in another installation.

The natural choice for a surrogate key in Microsoft Dynamics AX is the *RecId* column. However, in Microsoft Dynamics AX 2009 and earlier, the *RecId* index is still paired with the *DataAreaId* column for company-specific tables. In Microsoft Dynamics AX 2012, the *RecId* index does not contain the *DataAreaId* column and becomes a single-column unique key.

Surrogate key support is enabled when you create a new table the AOT. To use the surrogate key pattern for a table, set the *PrimayIndex* property to *SurrogateKey*, as shown in Figure 17-3.

Note The *SurrogateKey* value is not available for tables that were created in an earlier version of Microsoft Dynamics AX. If you want to implement a surrogate key for an existing table, you must re-create the table in the AOT.

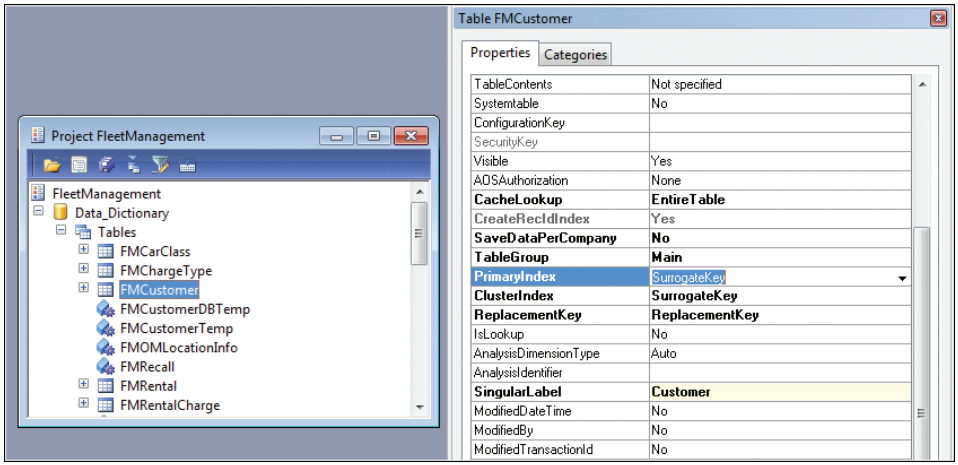


FIGURE 17-3 Surrogate key in the FMCustomer table.

Defining a surrogate key on a table in Microsoft Dynamics AX has several benefits. The first benefit is performance. When a surrogate key and the foreign key that references it are used to join two tables, performance is improved when compared to joins created with other data types. The benefit is more prominent when compared to Microsoft Dynamics AX 2009 and earlier versions because the kernel automatically adds the *DataAreaId* column to any key defined for any company-specific table. You identify these company-specific tables through the *SaveDataPerCompany* property. Without a surrogate key, the join must be based on at least on two columns, with one of them being the *DataAreaId* column.

The second benefit of using a surrogate key is that a surrogate key value never changes, which eliminates the need to change the values of foreign keys. For example, the Currency table (Figure 17-4) uses the *CurencyCodeIdx* index as the primary index, which contains the *CurrencyCode* column. The Ledger table has two foreign keys in the Currency table that are based on the *CurrencyCode* column. If there is ever a need to change the *CurrencyCode* value for a record in the Currency table, the corresponding records in the Ledger table also must be updated. But if a surrogate key were used for the Currency table and the Ledger table holds the surrogate foreign key, you could update the

CurrencyCode value in the Currency table without affecting the key relationship between the row in the Currency table and the rows in the Ledger table.

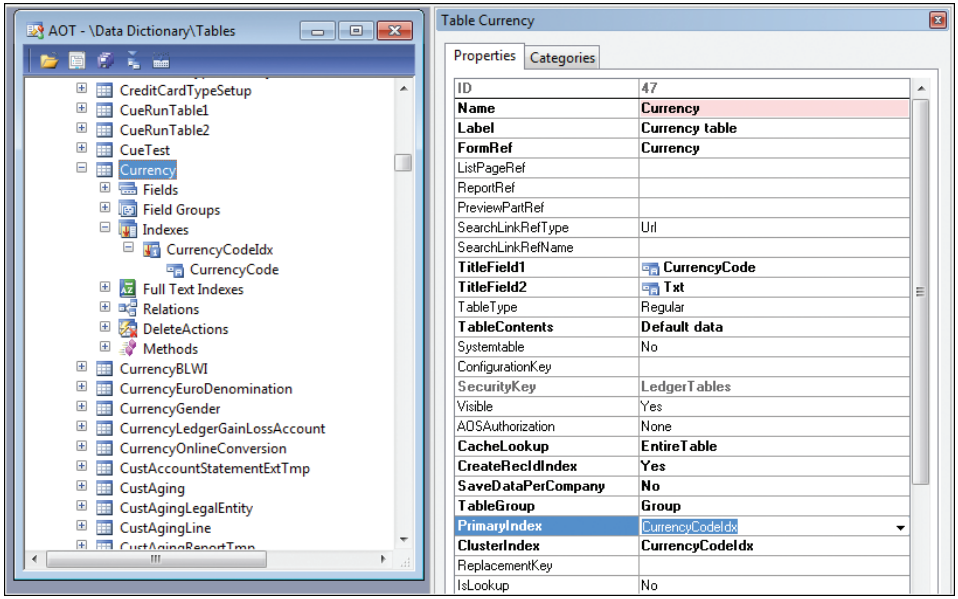


FIGURE 17-4 Currency table without a surrogate key.

The third benefit of using a surrogate key is that it is always a single-column key. Some features of SQL Server, such as full-text search, require a single-column key. Using a surrogate key lets you take advantage of these features.

Surrogate keys do have some drawbacks. The most prominent drawback is that the key value is not human-readable. When you look at the foreign keys on a related table, it is not easy to determine what the related row is. To display meaningful information that identifies the foreign key, some human-readable information from the related entity must be retrieved and displayed instead. This requires a join to the related table, and the join adds performance overhead.

Alternate keys

For a table, a *candidate key* is a minimal set of columns that uniquely identifies each row in the table. An *alternate key* is a candidate key for a table that is not a primary key. In Microsoft Dynamics AX 2012, you can mark a unique index to be an alternate key.

Because Microsoft Dynamics AX already has the concept of a unique index, you might wonder what the value is of having the concept of an alternate key. Developers create unique indexes for various reasons. Typically, one unique index serves as the primary key. Sometimes, additional unique indexes are created for performance reasons, such as to support a specific query pattern. When you look at the unique indexes for a table, it is not always obvious which index is used for the primary key

and which have been added for other reasons. For example, if you were to extract your data model and present it to a business analyst, you would not want the analyst to see the keys that were created solely for performance reasons. You need a way to separate your semantic model from your physical data model for this purpose. Being able to designate the additional unique indexes as alternate keys helps you to achieve this.

Figure 17-5 shows a unique index that was added to the FMVehicleModel table of the Fleet Management sample application. This is not the primary index for the table, so the *AlternateKey* property for the index is set to *Yes*.

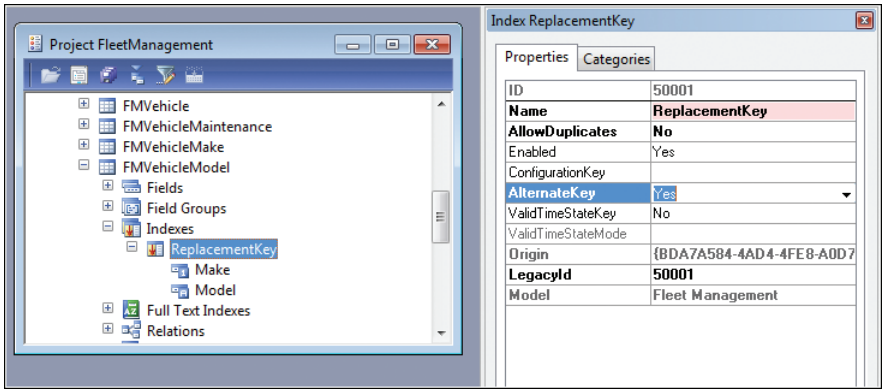


FIGURE 17-5 Alternate key on the FMVehicleModel table.

Table relations

Relationships between tables (called *relations* in Microsoft Dynamics AX) are key to the data model and run-time behavior. They are used in the following run-time scenarios in Microsoft Dynamics AX:

- Join conditions in the query or form data source and form dynalink
- Delete actions on tables
- Lookup conditions for bound (backed by a data source) or unbound controls on forms

Several changes and enhancements have been made for table relations in Microsoft Dynamics AX 2012.

EDT relations and table relations

In Microsoft Dynamics AX, table relations can be explicitly defined on tables, or they can be derived from relation properties that are defined on extended data types (EDTs) associated with table fields. The Microsoft Dynamics AX kernel looks up the relation properties for EDTs, and then for table relations. The order may be switched, depending on the scenario.

There are several issues with defining the relation properties on EDTs and mixing them with the relations defined on tables. First, EDT relations capture relations on only a single field. They cannot be used to capture multiple-field relations, which leads to incomplete relationships that are used in join or delete actions. Second, most of the properties for the relation depend on the context in which the relation is used. This context cannot be captured for EDT relations because they are stored in a central location in the AOT. For example, the role and related role name and cardinality and related cardinalities could be different for relations on different tables. Third, it is difficult to figure out how many relations are actually available for a table because the table relations give you only a partial view. You need to also look at relations that are defined on EDTs for the fields in the table and their base EDTs.

To address these issues, Microsoft Dynamics AX 2012 has migrated most of the EDT relations to table relations. To begin deprecating the *Relations* node under individual EDTs, the addition of new relations is not allowed. If an EDT has no nodes defined under the *Relations* node, the node is not displayed in Microsoft Dynamics AX 2012. An EDT has a new *Table References* node for cases where a control is bound directly to an EDT, and not through a table field. To reduce the work of manually adding a table relation that can be used in place of the EDT relation, you are prompted to create the table relation automatically when an EDT with a valid foreign key reference is used on a table field.

Because the Microsoft Dynamics AX run time performs lookups for EDT relations and table relations in a specific order, you need to ensure that the same relation gets picked up before and after the migration in all scenarios. This is especially important when migrating EDT relations in multiple table relations between two tables. The Microsoft Dynamics AX run time achieves backward compatibility by examining some properties that were set on table fields and table relations. These properties enable the run time to determine whether a table relation was created from an EDT relation that existed before. These properties are explained in Table 17-1.

TABLE 17-1 Properties of table fields.

Property location	Property name	Values	Description
Table relation	EDTRelation	Yes/No	Indicates to the kernel whether the relation was created because an EDT relation was migrated. The kernel uses this information to order and pick the relation for join, lookup, and delete actions when no explicit relation is specified.
Table field	IgnoreEDTRelation	Yes/No	Indicates whether the EDT relation on the field should be migrated to a table relation. The Microsoft Dynamics AX run time does not use this property; however, the EDT Relation Migration tool does.
Table relation link	SourceEDT	EDT name	Used by the Microsoft Dynamics AX run time to determine the cases in which the EDT relation is used.

You do not have to migrate an EDT relation manually to a table relation and then set the properties described in Table 17-1. The EDT Relation Migration tool can help with this process. You can access this tool from Tools > Code Upgrade > EDT Relation Migration Tool, as shown in Figure 17-6. For information about how to use this tool, see the topic "EDT Relation Migration Tool" at *http://msdn.microsoft.com/en-us/library/gg989788.aspx*.

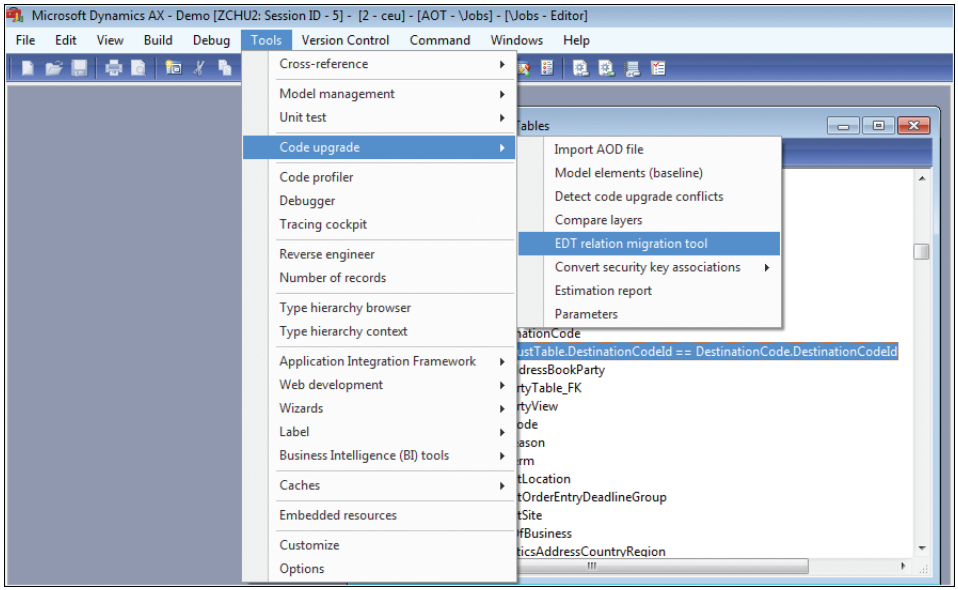


FIGURE 17-6 EDT Relation Migration tool.

After you migrate the EDT relations to table relations, verify that delete actions, query joins, and lookups work the same way they did before the migration. Focus on cases in which the migration of an EDT relation resulted in multiple table relations between two tables. The EDT Relation Migration tool lists the artifacts that are affected.

The following are some examples of the artifacts that might be affected. For example, the SalesTable table and the CustTable table have two relations between them defined on SalesTable. One of them was migrated from an EDT relation because it did not exist as a table relation before. The two relations are based on the fields *CustAccount* and *InvoiceAccount*. For delete actions, the relation based on *CustAccount* should be picked up before and after the migration. The *SalesHeading* query has a join between the SalesTable table and the CustTable table with the *Relations* property set to *Yes* on the data source for the SalesTable table. This query should pick up the relation based on *CustAccount* instead of the relation based on *InvoiceAccount* before and after the migration.

Foreign key relations

A goal for Microsoft Dynamics AX 2012 was to make the data model more consistent. This includes using surrogate key patterns when appropriate. It also includes using only primary keys as your foreign key references whenever possible. To facilitate the latter, Microsoft Dynamics 2012 introduces a special type of foreign key relation that you can use when creating relations between tables, as shown in Figure 17-7. A foreign key relation allows only two kinds of references to the related table. The first is a reference to the primary key. The second is a reference to a single-column alternate key of the related table. This reference to the single-column alternate key is provided to reduce the number of surrogate foreign key joins that were discussed in the "Surrogate keys" section, earlier in this chapter.

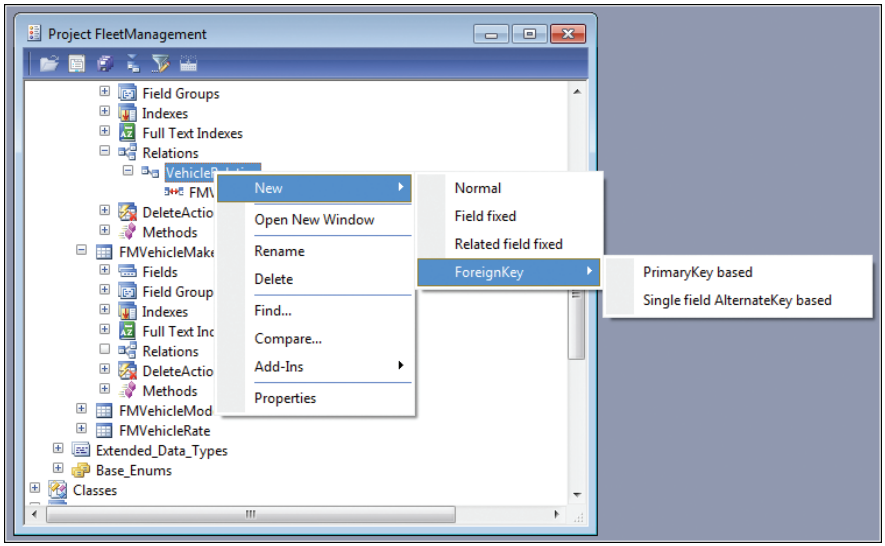


FIGURE 17-7 The new foreign key relation in Microsoft Dynamics AX 2012.

If you use a human-readable alternate key as your foreign key, you can display the foreign key on forms without the need for a join. You might wonder why only single-column alternate keys are allowed for foreign key references. This is to balance performance for a different usage pattern, when you actually want to join between the two tables (not for purposes of the user interface). Joins that are based on smaller columns (both the size of the columns and the number of columns) perform faster. By restricting the pattern to only single-column alternate keys, the performance degradation of the join is limited.

A consistent table relation pattern can result in performance benefits, too. For example, if one table references the CustTable table by using *keyA*, and another table references the CustTable table by using *keyB*, both tables must be joined to the CustTable table to correlate the rows in these two tables. However, if both of them use the same key, they can correlate directly, eliminating the need for joins.

Foreign key relations have some capabilities that other relations do not. For example, you can use them as join conditions in queries. This saves you from having to manually enter the field join conditions, which can be prone to error. Navigation property methods can also be generated for foreign key relations, as discussed in the next section.

The *CreateNavigationPropertyMethods* property

When you expand the *Relations* node for a table defined in the AOT, you can see the table relationships that are defined. The *CreateNavigationPropertyMethods* property, which is available only for foreign key relations, has special significance. Setting this property to *Yes,* as shown in Figure 17-8, causes kernel-generated methods on a table buffer to be created. You can use these methods to set, retrieve, and navigate to the related table buffer through the relation specified. The examples later in this section show the method signatures and usage patterns.

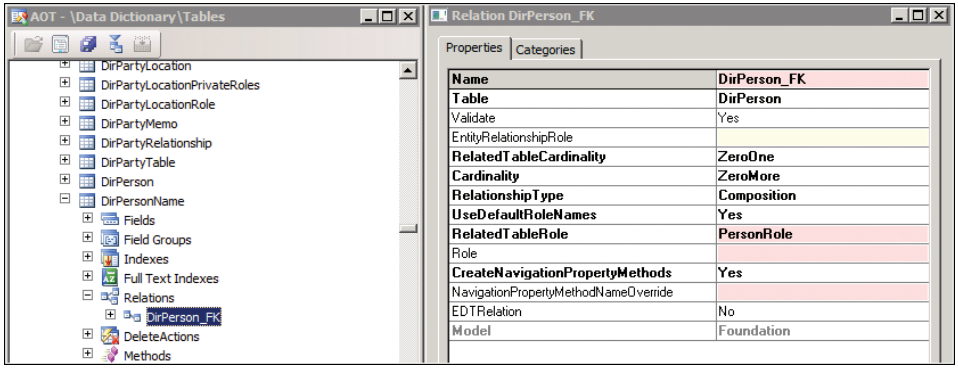


FIGURE 17-8 The *CreateNavigationPropertyMethods* property on a foreign key relation.

The navigation *setter* method links two related table buffers together. It is frequently used with the *UnitOfWork* class to create rows in the database from those table buffers. This effectively allows you to create an in-memory object graph with a related table buffer so that you can push them into the database with the proper relationship established among the rows. For more information, see the "Unit of Work" section later in this chapter.

The navigation *getter* method retrieves the related table buffer if a *setter* method has set it. Otherwise, the method retrieves the related table buffer from the database. This can effectively replace the *find* method pattern that is commonly used on tables. In the latter case, the table buffer that is returned is not linked to the table buffer on which the method was called. This means that the method will try to retrieve data from the database again. Note that when the navigation property getter method queries the database to get the related record, it selects all fields for that record. This can affect performance, particularly in cases where you had selected a smaller field list to achieve a performance benefit.

The following code uses the *DirPartyTable_FK* method to retrieve the related DirPartyTable table record for a customer with an account number of *1101* and prints the customer's name to the Infolog:

```
static void NavigationPropertyMethod(Args _args)
{
    CustTable cust;

    select cust where cust.AccountNum == '1101';

    // The DirPartyTable_FK() methods retrieves the related DirPartyTable record
    // through the DirPartyTable_FK role defined on the CustTable

    info(strFmt('Customer name for %1 is %2',cust.AccountNum, cust.DirPartyTable_FK().Name));
}
```

Figure 17-9 shows the output of this example in the Infolog.

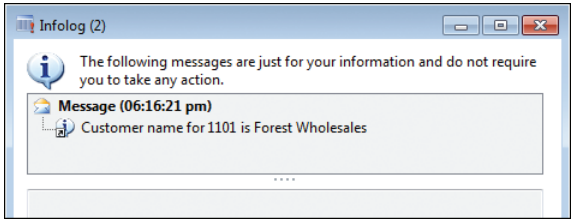


FIGURE 17-9 Output from the *DirPartyTable_FK* method example.

However, if the navigation property *setter* method is used to set the related DirPartyTable record, that record is always returned and the run time does not query the database:

```
static void NavigationPropertyMethodSetter(Args _args)
{
    CustTable cust;
    DirPartyTable dp;

    select cust where cust.AccountNum == '1101';
    dp.Name = 'NotARealCustomer';

    // Set the related DirPartyTable record

    cust.DirPartyTable_FK(dp);

    // The DirPartyTable_FK() methods retrieves the DirPartyTable record set above and
    // does not retrieve from the database.

    info(strFmt('Customer name for %1 is %2',cust.AccountNum, cust.DirPartyTable_FK().Name));
}
```

Figure 17-10 shows the output from this example in the Infolog.

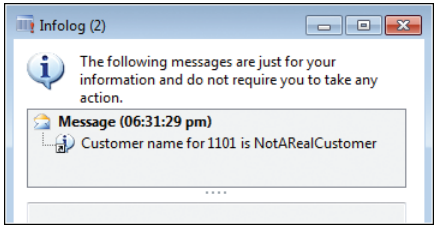


FIGURE 17-10 Output from an example with a navigation property *setter* method.

Each navigation method must have a name. Like any other method on the table, its name cannot conflict with other methods. By default, the *RelatedTableRole* property is used for the method name. An error is thrown during table compilation if a conflict with another method name is detected. If a conflict occurs, use the *NavigationPropertyMethodNameOverride* property to specify the name to use.

Table inheritance

Table inheritance has long been part of extended entity relationship (ER) modeling, but there was no built-in support for this in earlier versions of Microsoft Dynamics AX. Any inheritance or object-oriented characteristics had to be implemented manually by the developer. Microsoft Dynamics AX 2012 supports table inheritance natively from end to end, including modeling, language, run time, and user interface.

Modeling table inheritance

To model table inheritance in Microsoft Dynamics AX 2012, you must first create a root table and then create a derived table. These tasks are described in the following sections. Later sections describe how to work with existing tables, view the type hierarchy, and specify table behavior.

Create the root table

First, you must create the table that is the root of the table hierarchy. Before you create any fields for the table, set the *SupportInheritance* property to *Yes*. For the root table, you must add an *Int64* column that is named InstanceRelationType, which holds the information about the actual type of a specific row. This column should have the *ExtendedDataType* property set to *RelationType*, and the *Visible* property set to *No*. After you create this field, you must set the *InstanceRelationType* property for the base table to the field that you just added. From this point, you can model the root table as you normally would.

Create a derived table

Next, create a derived table, and set the *SupportInheritance* property to *Yes*. Set the *Extends* property to point to the table on which the derived table is based. Set these properties before you create any fields for the table. This will help ensure that all fields in tables in the hierarchy have unique names and IDs, which is necessary for the run time to work correctly. It also makes it possible to choose different storage models, such as storing all types in a single table, without causing name collisions. Storage is discussed later in this section.

Work with existing tables

If tables already have fields before you add the tables to an inheritance hierarchy, you might need to update the field names and IDs in both metadata and code. If the tables already contain data, the existing data will need to be upgraded to work with the new table hierarchy. These are nontrivial tasks. For these reasons, creating a new table inheritance hierarchy from existing tables is not supported.

View the type hierarchy

You can use the Type Hierarchy Browser to view a table inheritance hierarchy. To do so, right-click a table in the AOT, and then click Add-Ins > Type Hierarchy Browser. Figure 17-11 shows the hierarchy for the FMVehicle table.

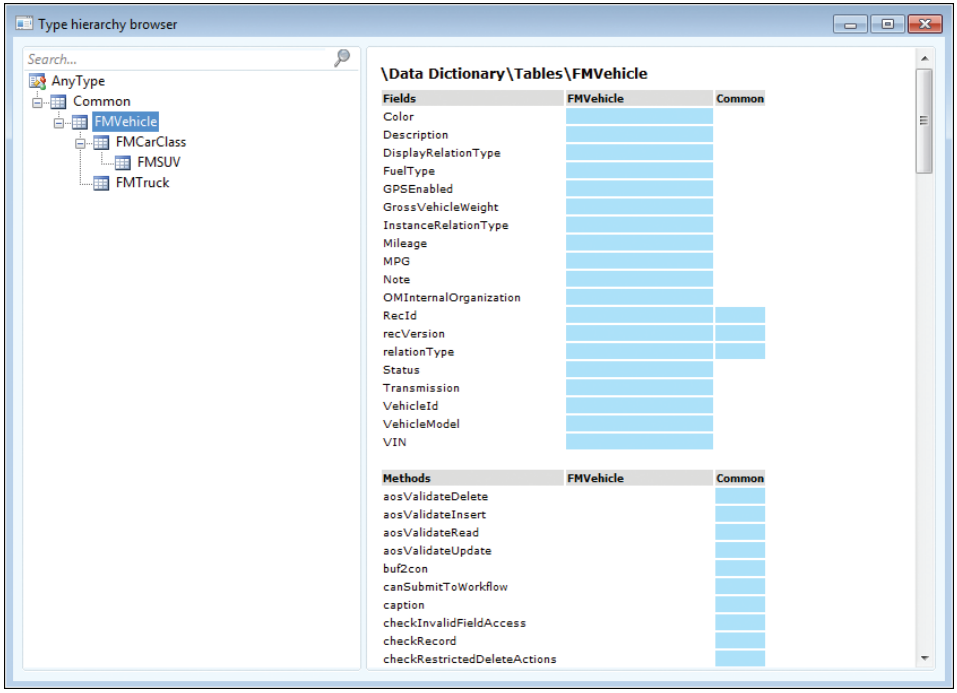


FIGURE 17-11 Hierarchy for the FMVehicle table.

Specify table behavior

Tables in an inheritance hierarchy share some property settings, so that table behavior is consistent throughout the hierarchy. These settings include the cache lookup mode, the OCC setting, and the save-data-per-company setting.

Configuration keys should be consistent with the table inheritance hierarchy. In other words, if a configuration key is disabled for the base table, the configuration key for the derived table should not be enabled. This condition is checked when you compile a table in the hierarchy, and errors are reported if the condition is found. For more information about configuration keys, see Chapter 11, "Security, licensing, and configuration."

You can specify whether a table in an inheritance hierarchy is concrete or abstract. By default, tables are concrete. This means that you can create a row that is of that table type. Specifying that a table is abstract means that you cannot create a row that is of that table type. Any row in the abstract table must be of a type of one of the derived tables (further up in the hierarchy) that is not marked as abstract. This concept aligns with the concept of an abstract class in an object-oriented programming language.

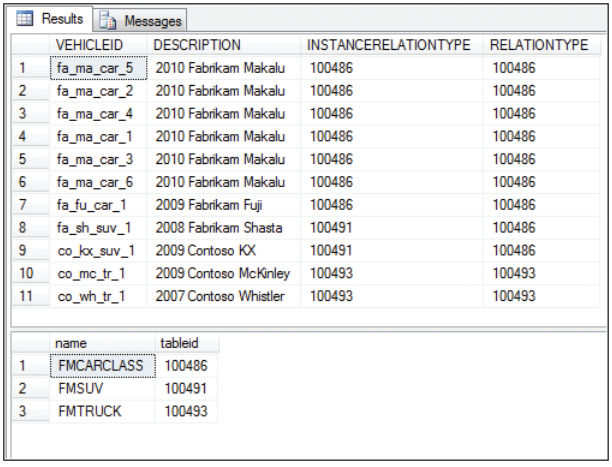
The table inheritance model in Microsoft Dynamics AX 2012 is a discrete model. Any row in the table hierarchy can be of only one concrete type (a table that is not marked as abstract). You cannot change the type of a row from one concrete type to another concrete type after the row is created.

Table inheritance storage model

In some implementations of object-relational (OR) mapping technologies, you can choose how the table inheritance hierarchy is mapped to data storage. The choices typically are one table for every modeled type, one table for every concrete type, or one table for every hierarchy. Microsoft Dynamics AX 2012 creates one table for every modeled type. Like a regular table, a table in a table inheritance hierarchy maps to a physical table in the database. Records in the inheritance hierarchy are linked through the *RecId* field. The data for a specific row of a type instance may be stored in multiple tables in the hierarchy, but they share the same *RecId*.

Every table in an inheritance hierarchy automatically has a system column that is named *RELATIONTYPE*. You will see this column in SQL Server, but not in the AOT. This column acts as a discriminator. The data for a concrete type is stored in multiple tables that make up the inheritance chain for that type. For a row in one of the tables, the discriminator column identifies the next table in the chain.

Figure 17-12 shows some rows in the FMVehicle table and the corresponding table IDs for its derived tables. The value of the *InstanceRelationType* field for cars equals the table ID of the FMCarClass table; for SUVs, the *InstanceRelationType* field equals the table ID of the FMSUV table; and for trucks, the *InstanceRelationType* field equals the table ID of the FMTrucks table. These represent the concrete type of each row in the FMVehicle table. The value of the *RelationType* field for both cars and SUVs equals the table ID of the FMCarClass table because FMCarClass is the next directly derived table for those rows. For trucks, the value of the *RelationType* field equals the table ID of the FMTruck table.


Results Messages

	VEHICLEID	DESCRIPTION	INSTANCERELATIONTYPE	RELATIONTYPE
1	fa_ma_car_5	2010 Fabrikam Makalu	100486	100486
2	fa_ma_car_2	2010 Fabrikam Makalu	100486	100486
3	fa_ma_car_4	2010 Fabrikam Makalu	100486	100486
4	fa_ma_car_1	2010 Fabrikam Makalu	100486	100486
5	fa_ma_car_3	2010 Fabrikam Makalu	100486	100486
6	fa_ma_car_6	2010 Fabrikam Makalu	100486	100486
7	fa_fu_car_1	2009 Fabrikam Fuji	100486	100486
8	fa_sh_suv_1	2008 Fabrikam Shasta	100491	100486
9	co_kx_suv_1	2009 Contoso KX	100491	100486
10	co_mc_tr_1	2009 Contoso McKinley	100493	100493
11	co_wh_tr_1	2007 Contoso Whistler	100493	100493

	name	tableid
1	FMCARCLASS	100486
2	FMSUV	100491
3	FMTRUCK	100493

FIGURE 17-12 Tables in the FMVehicle hierarchy.

Polymorphic behavior

When you issue a query on a table that is part of a table inheritance hierarchy, the Microsoft Dynamics AX run time provides the type fidelity by default. This means that if a *select* * statement is performed for a table, all of the rows of that table type are returned. For example, if you issue

the query *select * from DirPartyTable*, all instances of *DirParty* are returned, including *DirPerson*, *OperatingUnit*, and so on. Morever, because *select* * is used, the query returns complete data. This means that the DirPartyTable table must be joined to all of its derived tables.

When a *select* * is performed for a table that is part of a table inheritance hierarchy, that table is joined with an inner join to all of its base tables (all the way up to the root table), and then joined with an outer join to all of its derived tables (including derived tables at all levels). The reason for this is that any row in that table must have a corresponding row in all of its base tables, but could have matching rows in any of the concrete type paths. This ensures that, no matter what concrete type a row is, complete data for that row is always retrieved. Similar to the polymorphic behavior in object-oriented programming, this mechanism provides polymorphic data retrieval for a table that is part of a table inheritance hierarchy. In cases where you need to have all of the data for a row, this behavior is very convenient. You can use the dynamic method binding feature of table inheritance to write code that is clean and extensible.

For example, in the Fleet Management project, the FMVehicle table has a *doAnnualMaintenance* method that simply throws an exception. This happens because FMVehicle is an abstract table and any concrete table that is derived from it must override this method. The tables FMCarClass, FMTruck, and FMSUV all override this method, as shown in Figure 17-13. Note that each overridden method accesses a field that is not accessible from the base table.

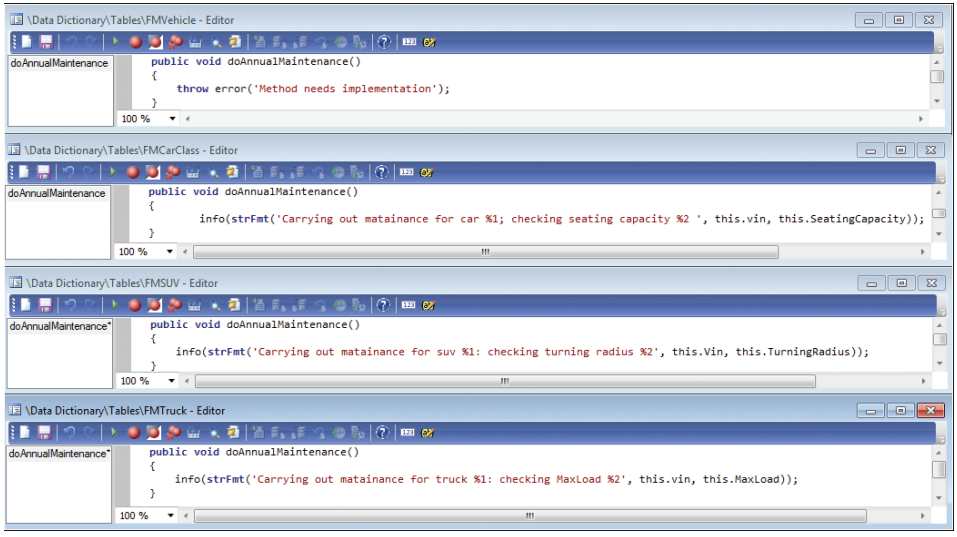


FIGURE 17-13 Overridden *doAnnualMaintenance* method on derived tables.

The following code queries the FMVehicle table and calls the *doAnnualMaintenance* method:

```
static void PolyMorphicQuery(Args _args)
{
    FMVehicle vehicle;

    while select vehicle
    {
```

```
        vehicle.doAnnualMaintenance();
    }
}
```

If you run the code as a job, you would get results that look similar to those shown in Figure 17-14.

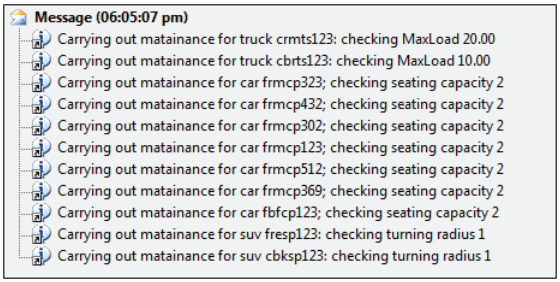


FIGURE 17-14 Result of calls to the overridden *doAnnualMaintenance* method.

As you can see, even though the *select* statement executes on the FMVehicle table, the statement returns fields in the derived tables. The actual *select* statement for this query looks like the following:

```
SELECT <all fields from all tables in the hiearchy> FROM  FMVEHICLE T1 LEFT OUTER JOIN FMTRUCK
T2 ON (T1.RECID=T2.RECID) LEFT OUTER JOIN FMCARCLASS T3 ON (T1.RECID=T3.RECID) LEFT OUTER JOIN
FMSUV T4 ON (T3.RECID=T4.RECID)
```

Tip You can get the Transact-SQL *select* statement directly from X++ code without having to use SQL Profiler. The following is an example:

```
static void PolyMorphicQuery(Args _args)
{
    FMVehicle vehicle;

    select generateonly vehicle;

    info(vehicle.getSQLStatement());
}
```

Performance considerations

When the inheritance hierarchy is very wide, very deep, or both, a polymorphic query can result in numerous table joins, which can degrade query performance. For example, the query *select * from DirPartyTable* produces eight table joins.

Exercise caution when using the table inheritance feature. Determine whether you really need all of the data from every derived type instance. If the answer is no, you should list the fields that you need explicitly in the field list. (Note that you can also list fields from derived tables when you model a query in the AOT or use the query object in X++ code. But you can only list fields from current and base tables when you write the X++ *select* statement.) The Microsoft Dynamics AX run time then adds joins to only the tables that contain the fields in the list. For example, if you change the query on the

DirPartyTable table to *select name from DirpartyTable*, only the DirPartyTable table is included in the query. No joins to other tables in the hierarchy are created because no data is being accessed from them. Careful query construction can improve query performance significantly.

Listing only the fields that are needed is not only beneficial here, but it is a good practice in general. This may allow SQL Server to use a covering index when processing the query and reduce the network load. A common concern about this practice is passing the table buffer to another function in another module, because you need to ensure that the other function does not use fields that were not selected to be returned. When an attempt is made to read fields that are not in the field list, Microsoft Dynamics AX 2009 and earlier versions does not produce an exception. You get whatever value is on the table buffer, which in most cases is an empty value. In Microsoft Dynamics AX 2012, an attempt to access an unavailable field raises an exception, but only if the field is included in a table that is part of a table inheritance hierarchy. A configuration setting is available to raise either a warning or exception for all tables that encounter this issue. To change this setting, do the following:

1. Click System Administration > Setup > System > Server Configuration.
2. On the Performance Optimization FastTab, under Performance Settings, click the drop-down list next to Error On Invalid Field Access to change the setting.

To maintain backward compatibility and to reduce run-time overhead, this setting is turned off by default. It is recommended that you activate this setting for testing purposes only.

Unit of Work

Maintaining referential integrity is important for any ERP application. Microsoft Dynamics AX 2012 lets you model table relations with richer metadata and express referential integrity in your data model more precisely. However, the application is still responsible for making sure that referential integrity is maintained. Microsoft Dynamics AX table relations are not implemented as database foreign key constraints in the SQL Server database. Implementing these constraints would add performance overhead in SQL Server. Also, for performance and other reasons, application code might violate referential integrity rules temporarily and fix the violations later.

Maintaining referential integrity requires performing data operations in the correct order. This is most prominent in cases where records are created and deleted. The parent record must be created first, before the child record can be inserted with the correct foreign key. But child records must be deleted before the parent record. Ensuring this manually in code can be error-prone, especially with the more normalized data model in Microsoft Dynamics AX 2012. Also, data operations are often spread among different code paths. This leads to extending locking and transaction spans in the database.

Microsoft Dynamics AX 2012 provides a programming concept called Unit of Work to help with these issues. Unit of Work is essentially a collection of data operations that is performed on related data. Application code establishes relationships between data in memory, modifies the data, registers the operation request with the Unit of Work framework, and then requests that the Unit of Work

perform all of the registered data operations in the database as a unit. Based on the relationships established among the entities in the in-memory data, the Unit of Work framework determines the correct sequence for the requested operations and propagates the foreign keys, if necessary.

The following code example shows Unit of Work in use:

```
public static void fmRentalAndRentalChargeInsert()
{
    FMTruck truck;
    FMRental rental;
    FMRentalCharge rentalCharge;
    FMCustomer customer;
    UnitofWork uoW;

    // Populate rental and RentalChange in UoW. 3 types of Rental Charge Records
    // for the same Rental.

    // Getting the customer and the truck that the customer is renting
    // These records are referred to from the rental record

    select truck where truck.VehicleId == 'co_wh_tr_1';
    select customer where customer.DriverLicense == 'S468-3184-6541';

    uoW = new UnitofWork();
    rental.RentalId = 'Redmond_546284';

    // This links the rental record with the truck record.
    // During insert, rental record will have the correct foreign key into the truck record.

    rental.fmVehicle(truck);

    // This links the rental record with the customer record.
    // During insert, rental record will have the correct foreign key into the
    // customer record.

    rental.fmCustomer(customer);
    rental.StartDate = DateTimeUtil::newDateTime(1\1\2008,0);
    rental.EndDate = DateTimeUtil::newDateTime(10\1\2008,0);
    rental.StartFuelLevel = 'Full';

    // Register the rental record with unit of work for save.
    // Unit of work makes a copy of the record.

    uoW.insertonSaveChanges(rental);

    uoW.saveChanges();

}
```

It is important to understand that Unit of Work copies the data changes into its own buffer when the registration method executes. After that, the original buffer is disconnected from Unit of Work. Any changes made to the table buffer after that will not be picked up by Unit of Work. If you want to save these changes through Unit of Work, you need to call the corresponding registration method again.

When you register multiple changes on the same record with Unit of Work, the last changes that are registered overwrite all previous changes.

In the previous code example, the code runs on the server because Unit of Work is a server-bound construct and cannot be instantiated or used on the client.

The form run time in Microsoft Dynamics AX uses the Unit of Work framework in its internal implementation to handle data operations on form data sources, where several form data sources are joined together directly. These scenarios did not work in previous releases of Microsoft Dynamics AX. When the form run time uses the Unit of Work framework, it is not accessible through X++.

Date-effective framework

Many business scenarios require tracking how data changes over a period of time. Some of the requirements include viewing the value of a record as it was in the past, viewing the value at the current time, or being able to enter a record that will become effective on a future date. A typical example of this is employee data in an application that is used by the Human Resources team of a company. Some questions that such an application will help answer might be as follows:

- What position did a specific employee hold on a specific date?
- What is the current salary for the employee?
- What is the current contact information for the employee?

In addition to answering these questions, there might also be a requirement to allow users to enter new data and change existing data. For example, a user could change the contact information for Employee C and make the new information effective on September 15, 2012. Such data is often referred to as date-effective data. Microsoft Dynamics AX 2012 supports creating and managing date-effective data in the database. The date-effective framework provides a number of features that include support for modeling entities, programmatic access for querying and updating date-effective data, run time support for maintaining data consistency, and user interface support. Core Microsoft Dynamics AX features, such as the Global Address Book and modules like Human Resources, use date-effective tables extensively in their data models.

Relational modeling of date-effective entities

Microsoft Dynamics AX provides support for modeling date-effective entities at the table level. The *ValidTimeStateFieldType* property of a table indicates whether the table is date-effective. This information is stored in the metadata for the table and is used at run time.

Figure 17-15 shows the DirPersonName table, which is used to track the history of a person's name. The table is date-effective because the *ValidTimeStateFieldType* property is set to *UtcDateTime*. When

you set this property on a table, the date-effective framework automatically adds the columns *ValidFrom* and *ValidTo*, which are required for every date-effective table. The data type of these columns is based on the value chosen for the *ValidTimeStateFieldType* property. Two data types are available:

- ***Date*** Tracking takes place at the day level. Records are effective starting from the *ValidFrom* date through the *ValidTo* date.
- ***UtcDateTime*** Tracking takes place at the second level. In this case, multiple records can be valid within the same day.

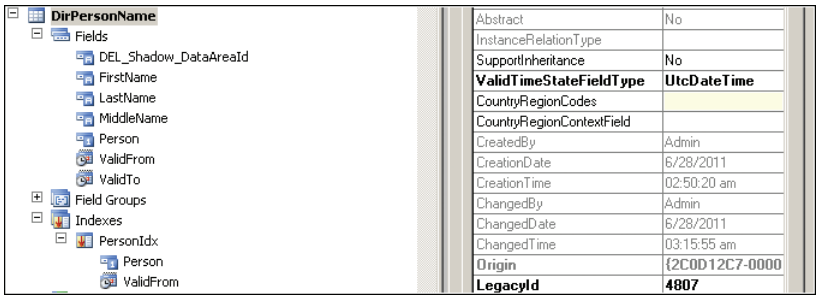


FIGURE 17-15 DirPersonName table with *ValidTimeStateFieldType* property set to *UtcDateTime*.

In addition to the fields each date-effective table is required to have, the table must have at least one alternate key that is implemented as a unique index. This alternate key is referred to as the *validtimestate* key in the date-effective framework, and it is used to enforce the time period semantics that are enabled by a date-effective table. The *validtimestate* key must contain the *ValidFrom* column and at least one other column other than the *ValidTo* column. The *validtimestate* key has an additional property that indicates whether gaps (missing records for a period of time) are allowed in the data. In Figure 17-16, the DirPersonName table is used track changes to the *Person* column. The *validtimestate* key contains the *Person* column and the *ValidFrom* column. When the *ValidTimeStateKey* property is set to *Yes* for an index, the index also needs to be a unique index and is required to be an alternate key.

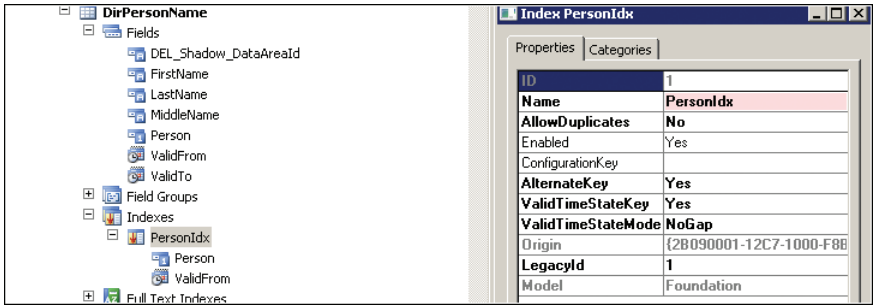


FIGURE 17-16 The *validtimestate* key index for the DirPersonName table.

In Table 17-2, the records reflect the changing names over a period of time for two people. The table must have a unique index with the following columns: *Person* and *ValidFrom*. The *ValidTo*

column can be part of the index, but it is optional. This index has the *ValidTimeStateKey* property set to *Yes*. Because the objective is to track the history of a person, the column that represents the person is also part of the *validtimestate* key. The *validtimestate* key enables the date-effective framework to indicate the field for which the history is being tracked.

TABLE 17-2 Tracking name changes over time in a date-effective table.

Person	FirstName	MiddleName	LastName	ValidFrom	ValidTo
1	Jim	M	Corbin	02/10/1983 00:00:00	04/16/1984 23:59:59
1	Jim	M	Daly	04/17/1984 00:00:00	12/31/2154 23:59:59
2	Anne		Wallace	04/14/2001 00:00:00	07/04/2005 23:59:59
2	Anne		Weiler	07/05/2005 00:00:00	12/31/2154 23:59:59

This scenario has a requirement not to allow gaps in each person's name data. To implement this requirement, the *ValidTimeStateMode* property is set to *NoGap*. For other scenarios, gaps in the data might be acceptable. This is also implemented by using the *ValidTimeStateMode* property. The date-effective framework also enforces that a person cannot have more than one name at the same time. This is called prevention of overlaps in the data. If the *ValidTo* column for a record contains the maximum value allowed (*12/31/215423:59:59*), it indicates that the record does not expire.

Support for data retrieval

Business application logic that is written in X++ may need to retrieve data that is stored in date-effective tables. To support this, the framework has three modes of data retrieval:

- **Current** Returns the record that is currently active by default, when you use a *select* statement or an application programming interface (API) to retrieve data from the table.
- **AsOfDate** Retrieves the record that is valid for the passed-in date or the *UtcDateTime* parameter. This can be in the past, current, or future. The *ValidFrom* column of the retrieved record is less than or equal to the value passed in. The *ValidTo* column is greater than or equal to the value passed in.
- **Range** Returns the records that are valid for the passed in range.

The X++ language supports a syntax that is similar to the Transact-SQL syntax that is used when querying relational databases. The date-effective framework enhances this syntax by adding the *validtimestate* keyword to indicate the type of query. The modes described earlier translate to the following queries.

Note The *ValidFrom* and *ValidTo* columns in these examples use the *Date* data type. If they used the *UtcDateTime* data type, the dates passed in would have to be in *UtcDateTime* format.

This query retrieves the current emergency contact information for *Employee A*. There is no need to specify any values for the *ValidFrom* and *ValidTo* columns in the *where* clause because the Microsoft Dynamics AX run time automatically adds them:

```
select * from EmployeeEmergencyContact where EmployeeEmergencyContact.Employee == 'A';
```

This query retrieves the record that was in effect on April 21, 1986:

```
select validtimestate (21\04\1986) * from EmployeeEmergencyContact where
EmployeeEmergencyContact.Employee == 'A';
```

This query retrieves all of the records for *Employee A* for the time period that is passed into the statement:

```
select validtimestate(01\01\1985, 31\12\2154)
* from EmployeeEmergencyContact where EmployeeEmergencyContact.Employee == 'A';
```

X++ also exposes a Query API to retrieve data from tables. This API has been extended with the following methods to allow different forms of querying:

- *validTimeStateAsOfDate(date)*;
- *validTimeStateAsOfDateTime(utcdatetime)*;
- *validTimeStateDateRange(date)*;
- *validTimeStateDateTimeRange(utcdatetime)*;

The date-effective framework uses these methods and transforms them by adding additional predicates on the *ValidFrom* and *ValidTo* columns to fetch the data that meets the requirement of the query.

Run-time support for data consistency

The data that is stored in a date-effective table must conform to the following consistency requirements:

- The data must not contain overlaps.
- Gaps are either allowed or disallowed, depending on the value of the *ValidTimeStateMode* property of the *validtimestate* key.

Because the data that is stored in the table can be added to or changed, the date-effective framework ensures that these consistency requirements are enforced. The date-effective framework implements these requirements by adjusting other records using the following rules:

- If *ValidFrom* is being updated, retrieve the previous record, and then update the *ValidTo* of the previous record to a value of *ValidFrom -1* to ensure that there is no overlap or gap. If the *ValidFrom* of the edited record is less than the *ValidFrom* of the previous record, display an error. The error is displayed because further action is required to delete the previous record

to avoid overlaps. The date-effective framework does not delete automatically records during adjustments.

- If *ValidTo* is being updated, retrieve the next record, and then update the *ValidFrom* of the next record to a value of *ValidTo + 1* to ensure that there is no overlap or gap. If the *ValidTo* of the edited record is greater than the *ValidTo* of the next record, display an error.
- The date-effective framework does not allow simultaneous editing of *ValidTo* and *ValidFrom* columns.
- The date-effective framework does not allow editing of any other columns that are part of the *validtimestate* key.
- When a new record is inserted, the *ValidTo* of the existing record is updated to a value of *ValidFrom -1* of the newly inserted record. New records cannot be inserted in the past or future if records already exist for that time period.
- When a record is deleted, the *ValidTo* of the previous record is adjusted to a value of *ValidFrom -1* of the next record, but only if the system requires that there should be no gaps. If gaps are allowed, no adjustment is performed.

Modes for updating records

The date-effective framework allows regular updates of records in a date-effective table. It also provides additional modes that are typical for the types of changes that are made to date-effective tables. The date-effective framework provides the following three modes:

- ***Correction*** This mode is analogous to regular updates to data in a table. If the *ValidFrom* or *ValidTo* columns are updated, the system updates additional records if necessary to guarantee that the data does not contain gaps or overlaps.
- ***CreateNewTimePeriod*** The date-effective framework creates a new record with updated values, and updates the *ValidTo* of the edited record to a value of *ValidFrom -1* of the newly inserted record. By default, the *ValidFrom* column of the newly inserted record has the current date for the *Date* data type columns or the current time for *UtcDateTime* data type columns. This mode does not allow editing of records in the past. This mode hides the date-effective characteristics of the data from the user. The user edits the records as usual, but internally a new record is created to continue tracking the history of changes made to the record.
- ***EffectiveBased*** This mode is a hybrid of the other two modes. Records in the past are prevented from being edited. Current active records are edited by using the same semantics as *CreateNewTimePeriod* mode. Records in the future are updated by using the same semantics as *Correction* mode.

The update mode must be specified by calling the *validTimeStateUpdateMode(ValidTimeState Update _validTimeStateUpdateMode)* method on the table buffer that is being updated. This method

takes a value from the *ValidTimeStateUpdate* enumeration as a parameter. This enumeration has the list of the various update modes.

Important The Microsoft Dynamics AX run time displays an error if the update mode is not specified when a date-effective table is updated through X++.

User experience

When the user updates a record in a date-effective table, other records might be updated as a consequence. The date-effective framework first provides a dialog box that informs the user about the additional updates and asks the user to confirm whether the action should proceed. The user's actions are simulated without actually updating the data. After the user chooses to update the data, the user interface is refreshed by retrieving all of the updated records.

Full-text support

Microsoft Dynamics AX 2012 has the capability to execute full-text search queries against the database. Full-text search functionality is provided by SQL Server and allows the ability to perform linguistic searches against text data stored in the database. For more information, see "Full-Text Search (SQL Server)" at *http://msdn.microsoft.com/en-us/library/ms142571.aspx*.

Microsoft Dynamics AX provides the capability to create a full-text index on a table. New methods that are available in the *Query* class that let you write queries that use this index. Figure 17-17 shows a full-text index that has been created for the VendTable table in the AOT.

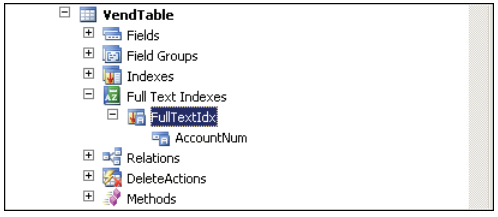


FIGURE 17-17 Full-text index for the VendTable table.

Only one full-text index can be created for a table. Only fields that have the *string* data type can be used as fields for the full-text index. When the table is synchronized, a corresponding full-text index is created in the database. Microsoft Dynamics AX also requires that the table be part of either the Main table group or the Group table group. You cannot create a full-text index for temporary tables.

The following example shows how full-text queries can be executed by using the Query API:

```
Query q;
QueryBuildDataSource qbds;
```

```
QueryBuildRange qbr;
QueryRun qr;
VendTable vendTable;

q = new Query();
qbds = q.addDataSource(tablenum(VendTable));
qbr = qbds.addRange(fieldnum(VendTable, AccountNum));
qbr.rangeType(QueryRangeType::FullText);
qbr.value(queryvalue('SQL'));
qr = new QueryRun(q);

while (qr.next())
{
   vendTable = qr.get(tablenum(VendTable));
   print vendTable.AccountNum;
}
```

The *QueryRangeType::FullText* enumeration used by the *rangeType* method causes the data layer to generate a full-text search query in the database. Microsoft Dynamics AX uses the *FREETEXT* keyword provided by SQL Server when it generates a full-text query that is executed on the database. For the previous code example, the following Transact-SQL query is generated:

```
SELECT T1.ACCOUNTNUM,T1.INVOICEACCOUNT,…
FROM VENDTABLE T1 WHERE (((PARTITION=?) AND (DATAAREAID=?)) AND (FREETEXT(ACCOUNTNUM,?))) ORDER
BY T1.ACCOUNTNUM
```

For more information about the *FREETEXT* keyword, see "Querying SQL Server Using Full-Text Search" on MSDN at *http://msdn.microsoft.com/en-us/library/ms142559.aspx*.

You can also use the extended query range syntax to generate a full-text query. This is shown in the following example. The *freetext* keyword specifies that the data layer should generate a full-text query.

```
qbrCustDesc = qsbdCustGrp.addRange(fieldnum(VendTable, AccountNum));
qbrCustDesc.value('((AccountNum freetext "bike") || (AccountNum freetext "run"))');
```

The QueryFilter API

A favorite Transact-SQL interview question is to ask a candidate to explain the difference between the *on* clause and the *where* clause in a *select* statement that involves joins. You can find the long version of the answer on MSDN, which talks about the different logical phases of query evaluation. The short answer is that there is no difference between the two for an inner join. For an outer join, rows that do not satisfy the *on* clause are included in the result set, but rows that do not satisfy the *where* clause are not.

So you might wonder when to use *on* and when to use *where*. An example will help illustrate. The following polymorphic query finds all DirParty instances with the name *John*. There are two kinds of

predicates here. The first one matches the individual rows with their corresponding base or derived type:

```
<baseTable>.recID == <derivedTable>.recid.
```

The second predicate matches the name:

```
DirPartyTable.name == 'John'
```

To start with the first predicate, when you take a specific row from the root table, it may have a matching row in any one of the derived tables. Because the goal is to retrieve complete data for all types, you do not want to discard a row just because it does not match one of the derived tables. For this reason, you want to use the *on* clause. For the second predicate, you want only the rows that qualify the predicate. Thus, you want to use the *where* clause. The Transact-SQL for the query looks like this:

```
SELECT * FROM  DIRPARTYTABLE T1 LEFT OUTER JOIN DIRPERSON T2 ON (T1.RECID=T2.RECID) LEFT OUTER
JOIN DIRORGANIZATIONBASE T3 ON (T1.RECID=T3.RECID) LEFT OUTER JOIN DIRORGANIZATION T4 ON (T3.
RECID=T4.RECID) LEFT OUTER JOIN OMINTERNALORGANIZATION T5 ON (T3.RECID=T5.RECID) LEFT OUTER JOIN
OMTEAM T6 ON (T5.RECID=T6.RECID) LEFT OUTER JOIN OMOPERATINGUNIT T7 ON (T5.RECID=T7.RECID) LEFT
OUTER JOIN COMPANYINFO T8 ON (T5.RECID=T8.RECID) WHERE (T1.NAME='john')
```

You might notice that the *on* clause is specified immediately after the table join, and the *where* clause is specified at the end of the query after all of the table joins and *on* clauses. This matches the order in which the query is evaluated. The *where* clause predicates are evaluated after all of the joins have been processed. The following X++ *select* statement produces a similar query:

```
static void Job3(Args _args)
{
    SalesTable so;
    SalesLine  sl;

    select so where so.CustAccount == '1101'
        outer join sl where sl.SalesId == so.SalesId;
}
```

The Transact-SQL for the query looks like this:

```
SELECT * FROM  SALESTABLE T1 LEFT OUTER JOIN SALESLINE T2 ON ((T2.DATAAREAID=N'ceu') AND (T1.
SALESID=T2.SALESID)) WHERE ((T1.DATAAREAID=N'ceu') AND (T1.CUSTACCOUNT=N'1101'))
```

There is something of a mix here. The keyword is *where*, but it is specified after each table join. So where does the predicate go in the Transact-SQL? If you use SQL trace, you'll see that for an outer join, the predicates appear in the *on* clause. For an inner join, it shows in the *where* clause. To understand this, keep in mind that the X++ *where* is actually the *on* clause in Transact-SQL. Because there is no difference between *on* and *where* for inner joins, the Microsoft Dynamics AX run time simply moves those to the *where* clause. The X++ Query programming model has the same behavior. Query ranges that you specify by using the *QueryBuildRange* clause go in the *on* clause.

So how do you specify *where* clause predicates? For the X++ *select* statement, you may be able to attach these *where* clause predicates on one of the tables that are inner-joined. Alternatively, you could add these to the *where* clause of the first table if all the other tables are outer-joined. The solution is more difficult with the Query programming model because you cannot specify *QueryBuildRange* on another data source without using the extended query range feature. To solve this problem, Microsoft Dynamics AX 2012 added support for the *QueryFilter* API.

Because the *where* clause is evaluated at the query level after all of the joins have been evaluated, the *QueryFilter* API is available at the query level. You can refer to any query data source that is not part of an exists/notexists subquery. The following example shows the use of *QueryFilter*:

```
public void setFilterControls()
{
    Query query = fmvehicle_qr.query(); // Use QueryRun's Query so that the filter can be
cleared
    QueryFilter filter;
    QueryBuildRange range;
    boolean useQueryFilter = true;  // Change to false to see QueryRange on outer join

    if (useQueryFilter)
    {
        filter = this.findOrCreateQueryFilter(
            query,
            query.dataSourceTable(tableNum(FMVehicleMake)),
            fieldStr(FMVehicleMake, Make));
        makeFilter.text(filter.value());
    }
    else
    {
        // Optional code to illustrate behavior difference
        // between QueryFilter and QueryRange
        range = SysQuery::findOrCreateRange(
            query.dataSourceTable(tableNum(FMVehicleMake)),
            fieldNum(FMVehicleMake, Make));
        makeFilter.text(range.value());
    }
}

public QueryFilter findOrCreateQueryFilter(
    Query _query,
    QueryBuildDataSource _queryDataSource,
    str _fieldName)
{
    QueryFilter filter;
    int i;

    for(i = 1; i <= _query.queryFilterCount(); i++)
    {
        filter = _query.queryFilter(i);
        if (filter.dataSource().name() == _queryDataSource.name() &&
            filter.field() == _fieldName)
```

```
        {
            return filter;
        }
    }

    return _query.addQueryFilter(_queryDataSource, _fieldName);
}
```

QueryFilter has similar grouping rules about how individual predicates are constructed with *AND* or *OR* operators. First, *QueryFilters* are grouped by query data source and the results are combined by using the *AND* operator. Next, within a group for a specific query data source, the same rules for *QueryRange* are applied: predicates on the same fields use the *OR* operator first and then the *AND* operator.

If you run the following X++ code and look at the Transact-SQL trace, you will see CROSS JOIN in the Transact-SQL statement. You may think that it misses the join condition and is doing a Cartesian product of the two tables. The join condition is actually in the *where* clause. A cross join like this is equivalent to an inner join with the join condition. The cross join is needed because the two tables must be listed before the outer join tables, because they appear as outer join conditions. Transact-SQL does not allow you to reference tables before they are used in the query.

```
static void CrossJoin(Args _args)
{
    CustTable cust;
    SalesTable so;
    SalesLine sl;

    select generateonly cust where cust.AccountNum == '*10*'
         join so where cust.AccountNum == so.CustAccount
            outer join sl where so.SalesId == sl.SalesId;

    info(cust.getSQLStatement());
}
```

The Transact-SQL for the query looks like this:

```
SELECT * FROM CUSTTABLE T1 CROSS JOIN SALESTABLE T2 LEFT OUTER JOIN SALESLINE T3 ON (((T3.
PARTITION=?) AND (T3.DATAAREAID=?)) AND (T2.SALESID=T3.SALESID)) WHERE (((T1.PARTITION=?) AND
(T1.DATAAREAID=?)) AND (T1.ACCOUNTNUM=?)) AND (((T2.PARTITION=?) AND (T2.DATAAREAID=?)) AND (T1.
ACCOUNTNUM=T2.CUSTACCOUNT))
```

Data partitions

In previous releases of Microsoft Dynamics AX, the *DataAreaId* column in a table was used to provide the data security boundary. It was also used to define legal entities through the concept of a company. As part of the organizational model and data normalization changes, a large number of entities like Products and Parties that were previously stored per-company, have been updated to be shared (through global tables) in Microsoft Dynamics AX 2012. This was done primarily to enable sharing of data across legal entities and to avoid data inconsistencies.

But in some deployments of Microsoft Dynamics AX, data is not expected to be shared across legal entities. In such deployments, the *DataAreaId* column is primarily used as a security boundary to segregate data into various companies. Such customers want to share the deployment, implementation, and maintenance cost of Microsoft Dynamics AX, but they have no other shared data or shared business processes. There are also holding companies that grow by acquiring independent businesses (subsidiaries), but the data and processes are not shared among these subsidiaries.

Microsoft Dynamics AX 2012 R2 provides a solution to these requirements. This release of Microsoft Dynamics AX uses the concept of data partitioning by adding a *Partition* column to the tables in the database. This allows the data to be truly segregated into separate partitions. When a user logs into Microsoft Dynamics AX, he or she always operates in the context of a specific partition. The system ensures that data from only the specified partition is visible, and that all business processes run in the context of that specific partition.

Partition management

The Partitions table contains the list of partitions that are defined for the system. During setup, Microsoft Dynamics AX creates a default partition called *initial*. A system administrator can create additional partitions by using the Partitions form in the System Administration module.

Development experience

A new property named *SaveDataPerPartition* has been added for all tables in the AOT. By default, the value is set to *Yes*, and the property cannot be edited. This property can be edited only if the *SaveDataPerCompany* property is set to *No* and the table is marked as a SystemTable, or if the table belongs to the Framework table group. These checks are put in place to enable all application tables to be partitioned. Only specific tables that are used by the kernel can have data that is not partitioned.

All of the tables whose *SaveDataPerPartition* property is set to *Yes* have a Partition system column in the metadata. In the database, the table has a *PARTITION* column with a data type *int64*. It is a surrogate foreign key to the RECID column of the PARTITION table. This column always contains a value from one of the rows in the PARTITION table. The column has a default constraint with the RECID value of the initial partition. The kernel adds the *PARTITION* column to all the indexes in a partition-enabled table except for the RECID index.

Run-time experience

The Microsoft Dynamics AX kernel framework handles the population and retrieval of the *Partition* column based on the partition that is specified for the current session. The various database operations provided by Microsoft Dynamics AX have the following functionality:

- ***select* statements** All *select* statements are filtered automatically based on the current partition of the session. The Transact-SQL statement that is generated always has the *Partition* column in the *WHERE* clause.
- ***insert* statements** The inserted buffer always has the *Partition* column set to the partition of the current session. Microsoft Dynamics AX displays an error if the application code sets the column to a value that is different from the current partition of the session.
- ***update* statements** All updates are performed in the current partition. Updating the *Partition* column is not allowed.

Because of this functionality, you usually do not have to write code to handle the *Partition* column. However, an exception is any code that uses direct Transact-SQL. The *Partition* column will not be handled automatically, and the direct SQL code has to ensure that the *WHERE* clause contains the partition of the current session.

To provide strict data isolation, the framework does not provide the ability to change partitions programmatically at run time. To change the partition, a new session has to be created that is set to use the other partition. Certain framework components like Setup and Batch use the *runAs* method, which creates a new session to execute code in a different partition. This is not a common pattern and should not be used in non-framework application logic.

Similarly, the framework does not allow execution of database operations that span multiple partitions. This is a contrast from the cross-company functionality that allows execution of database statements across multiple legal entities.

CHAPTER 18

The batch framework

In this chapter

Introduction

The batch framework in Microsoft Dynamics AX 2012 is an asynchronous, server-based task execution environment. It lets users execute asynchronous tasks in parallel and across multiple instances of the Application Object Server (AOS). In this release, the batch framework has been further enhanced from Microsoft Dynamics AX 2009. Among other enhancements, the batch framework now runs code in .NET common intermediate language (CIL), gives system administrators and developers increased control over batch jobs, and enables better performance and greater reliability when those jobs execute.

Microsoft Dynamics AX 2012 includes several tools that support the batch. The Batch Job form gives system administrators increased flexibility in the design, setup, and execution of complex batch jobs. In addition, the Batch Job form provides the ability to add multiple batch tasks to a single batch job and to define the dependencies among those tasks. An enhanced Batch application programming interface (API) gives X++ developers more control over complex batch jobs, along with the ability to process batch jobs directly from business logic.

A new framework, the SysOperation framework, was also introduced in this release. This framework enables application logic to be written in a way that supports running interactively or via the batch framework. It is a refinement of the RunBase framework and provides additional flexibility for creating new batch jobs. For more information about the SysOperation framework, see Chapter 14, "Extending Microsoft Dynamics AX."

Batch processing in Microsoft Dynamics AX 2012

Batch processing is a noninteractive task-processing technique where users create batch jobs to organize appropriate types of tasks to be processed as a unit. Batch processing has some important advantages: it lets users schedule batch tasks and define the conditions under which they execute, add the tasks to a queue, and set them to run automatically on a batch server. After execution is

complete, the batch server logs any errors and sends alerts. A batch job might involve printing reports, closing inventory, or performing periodic maintenance. By scheduling a batch job to process these types of resource-intensive tasks in off-peak hours, users can avoid slowing down the system during working hours.

Table 18-1 describes how standard batch processing concepts are represented in Microsoft Dynamics AX. These concepts are discussed in greater detail throughout this chapter.

TABLE 18-1 Batch processing concepts in Microsoft Dynamics AX 2012.

Concept	Description
Batch task	The smallest unit of work that can be executed using the batch framework. It is a batch-executable class that contains business logic to perform a certain action. The Microsoft Dynamics AX classes that are used for batch tasks are designated to run on the server. These tasks can run automatically as part of a batch job on the AOS. This version of the product has limited support for client batch jobs; it is recommended that you use server-side batch jobs to take full advantage of the new features in Microsoft Dynamics AX 2012. For more information, see the "Create a batch-executable class" section, later in this chapter.
Batch job	A complete process that achieves a goal, such as printing a report or performing the inventory closing process. A batch job is made up of one or more batch tasks.
Batch group	A logical categorization for batch tasks that lets administrators specify which AOS instance runs a particular task. Tasks that are not explicitly assigned to a batch group are, by default, assigned to an *empty* (default) group.
Batch server	An AOS instance that processes batch jobs. Read more about AOS in Chapter 1, "Architectural overview." For more information about configuring an AOS instance to be a batch server, see the "Configure the batch server" section, later in this chapter.

Common uses of the batch framework

Organizations can use the batch framework to perform asynchronous operations in a variety of scenarios. Typically, organizations create batch jobs to address the following kinds of needs:

- **Enable scheduling flexibility** The batch framework can perform periodic tasks on a regular schedule, such as data cleanup or invoice processing. For example, to run invoice processing at the end of every month, you can set up a recurring batch job that runs at midnight on the last working day of each month. The batch framework automatically picks up the job and processes pending invoices according to the specified schedule.
- **Control the order in which tasks execute** With the batch framework, you can develop a workflow or perform a complex data upgrade in a sequence that you specify. You can also set up dependencies between the tasks and create a dependency tree that ensures that certain tasks run in sequence while others run in parallel.
- **Enable conditional processing** Decision trees can help you implement a reliable way of processing data. Developers or system administrators can set up dependencies between tasks in such a way that different tasks are executed, depending on whether a particular task succeeds or fails. (Figure 18-4, later in the chapter, shows an example of a dependency tree.) System administrators can also set up alerts so that they are notified if a job fails.

- **Improve performance by using parallelization** The batch framework lets you take advantage of multithreading, which ensures that your processor's capabilities are used fully. This is particularly important for long-running processes, such as inventory closing. You can improve performance further by breaking a process into tasks and executing them against different AOS instances, thus increasing the throughput and reducing overall execution time.
- **Implement advanced logging and profiling** The batch framework lets you see what errors or exceptions were thrown the last time the batch ran, and it also shows you how long a process takes to execute. Advanced logging and the new profiling capabilities are also useful for performance benchmarking and security auditing.

Performance

The new capability to run larger and more complex batch jobs has required performance enhancements to the batch framework. In Microsoft Dynamics AX 2012, the batch framework is designed to be a server-side component. This lets you design multithreaded server processes in a controlled manner. By configuring the number of parallel execution threads and servers, defining the set and order of tasks for processing, and setting the execution schedule, you can achieve greater scalability across your hardware.

As mentioned earlier, the batch framework is now designed to run X++ that has been compiled as .NET CIL code for batch jobs. This significantly improves performance compared to Microsoft Dynamics AX 2009, which ran interpreted X++ code. Compared to the interpreted code, garbage collection is much better, and because of session pooling, scheduling new batch jobs is less resource intensive. You can also profile the performance of jobs by using Microsoft Visual Studio Performance Profiler.

Microsoft developers use the batch framework as a foundation for many performance-critical processes, such as maximizing hardware scalability during a data upgrade and maximizing throughput during journal posting. For more information about how Microsoft uses the batch framework for performance-critical processes, see the white paper "Journal Batch Posting," available at *http://www.microsoft.com/en-us/download/details.aspx?id=13379*.

Create and execute a batch job

Microsoft Dynamics AX 2012 includes numerous batch jobs that perform operations such as generating reports, creating sales invoices, and processing journals. However, in several situations, organizations need to create their own batch jobs. The batch framework provides full flexibility in the types of jobs that you can create. This section walks you through the following steps, which are required for creating, executing, and managing a batch job:

1. Create a batch-executable class.
2. Create a batch job and define the execution schedule.

3. Configure a batch server and create a batch group.
4. Manage the batch job.

Create a batch-executable class

The first step in developing a batch job is to define a class that can be executed as a batch task. Many classes included with Microsoft Dynamics AX 2012 are already enabled for batch processing. You can also design a batch-executable class, as shown in the following example:

```
public class ExampleBatchTask extends RunBaseBatch
```

To run as a batch task, a class must implement the Batchable interface. The best way to implement the interface contract is to extend the *RunBaseBatch* abstract class, which provides much of the necessary infrastructure for creating a batch-executable class. An alternative is to use the SysOperation framework, which provides additional advantages compared to extending the *RunBaseBatch* class. For more information about the SysOperation framework, see Chapter 14.

The *RunBaseBatch* class is an extension of the RunBase framework, so your batch class must adhere to the patterns and guidelines of the *RunBase* extended classes (see Chapter 14 for more information).

Table 18-2 describes the methods that must be implemented when you extend the *RunBaseBatch* class. The following sections describe these methods in more detail.

TABLE 18-2 Required methods for extensions to the *RunBaseBatch* class.

Method	Description
run	Contains the core logic for your batch task
pack	Serializes the list of variables used in the class
unpack	Deserializes the list of variables used in the class
canGoBatchJournal	Determines whether the class appears in the Batch Task form

run method

You implement the core logic of your batch class in the *run* method. The *run* method is called by the batch framework for executing the task defined within it. You can run most of the X++ code in this method; however, there are some limitations on the operations that you can implement. For example, you can't call any client logic or dialog boxes. However, you can still use the *Infolog* class. All Infolog and exception messages are captured when the batch class executes, and they are stored in the batch table. You can view these later in the Batch Job form or the Batch Job History form, both of which are located under System Administration > Inquiries > Batch Jobs.

Note If an error message is written to the Infolog, it does not mean that the task has failed; instead, an exception must be thrown to indicate the failure.

pack and *unpack* methods

A class that extends *RunBaseBatch* must also implement the *pack* and *unpack* methods to enable the class to be serialized. When a batch task is created, its member variables are serialized by using the *pack* method and stored in the *batch* table. Later, when the batch server picks up the task for execution, it deserializes class member variables by using the *unpack* method. So it's important to provide a correct list of the variables that are necessary for class execution. If any member variable isn't packable, then the class can't be serialized and deserialized to the same state.

The following example shows the implementation of the *pack* and *unpack* methods:

```
public container pack()
{
    return [#CurrentVersion,#CurrentList];
}

public boolean unpack(container _packedClass)
{
    Version version     = RunBase::getVersion(_packedClass);
    switch (version)
    {
        case #CurrentVersion:
            [version,#CurrentList] = _packedClass;
            break;
        default:
            return false;
    }
    return true;
}
```

The *#CurrentList* and *#CurrentVersion* macros that are referenced in the preceding code must be defined in the class declaration. Using a macro simplifies the management of variables in the class. If you add or remove variables later, you can manage the list by modifying the macro. The *#CurrentList* macro holds a list of the class member variables to pack, as shown here:

```
    #define.CurrentVersion(1)
    #localmacro.CurrentList
            methodVariable1,
            methodVariable2
    #endmacro
```

canGoBatchJournal method

When a system administrator creates a new batch task by using the Batch Task form, the *canGoBatchJournal* method determines whether the batch task class appears in the list of available classes. For an example of how to use *canGoBatchJournal*.

Create a batch job

The second step in developing a batch job is to create the batch job and add batch tasks. You can create a batch job in three ways:

- By using the dialog box of a batch-enabled class
- By using the Batch Job Designer form
- By using the Batch API

The method you use depends on the degree of flexibility that you need and the complexity of the batch job. To create a simple batch job, consisting of a single task with no dependencies, you typically use the dialog box of a batch-executable class; to create a more complex batch job, consisting of several tasks that might have dependencies, use the Batch Job form; to create a highly complex or very large batch job, or one that needs to be integrated with other business logic, use the Batch API. The following sections provide an example of using each method.

Create a batch job from the dialog box of a batch-executable class

The simplest way to run a batch-executable class as a batch job is to invoke the class by using a menu item. A menu item that points to a batch-executable class automatically opens a dialog box that lets the user create a batch job. On the Batch tab of the dialog box, select the Batch Processing check box, as shown for the *Change based alerts* class in Figure 18-1. When you select Batch Processing and click OK, a new batch job with the task that represents the batch-executable class is created. The batch job then runs asynchronously at the date and time you specify. You can also set up recurrences or alerts for the job by clicking the appropriate button on the right side of the dialog box. You can also specify the batch group for the task by using the drop-down list.

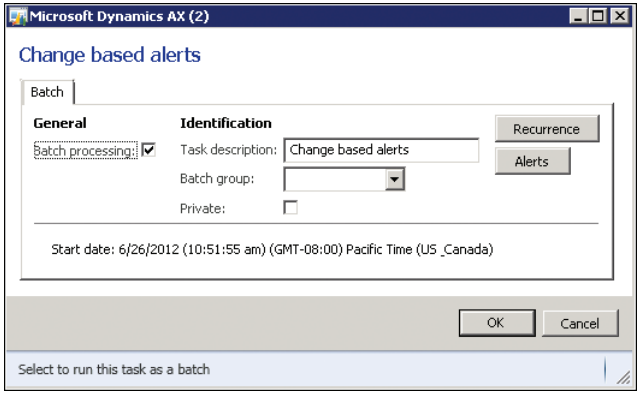


FIGURE 18-1 An example of the Batch tab for a class.

Create a batch job by using the Batch Job form

You can open the Batch Job form from several places. For example, you can open it by clicking Batch Jobs from System Administration > Inquiries > Batch Jobs or by selecting My Batch Jobs (for users) from Home > Inquiries > My Batch Jobs. Both menu items open the same form, but the information that is presented in the form differs, depending on the menu item that you use to open it. Depending on how you open the form and your level of access, you can view either the batch jobs that you have created or all batch jobs that are scheduled in the system.

Press Ctrl+N to create a new batch job, and then enter the details for the job in the grid or on the General tab: a description, and the date and time at which you want the job to start. You can also set up recurrence for the batch job by clicking Recurrence on the menu bar, and then entering a range and pattern for the recurrence.

Note If you don't enter a date and time, the current date and time are entered automatically.

Figure 18-2 shows the Batch Job form.

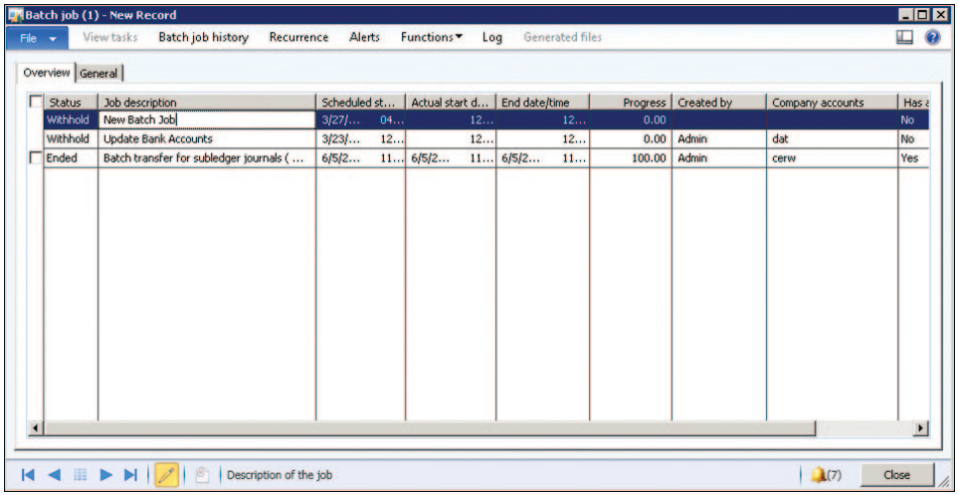


FIGURE 18-2 The Batch Job form.

After you create a batch job, you can add tasks to it and create dependencies between them by using the Batch Tasks form (shown in Figure 18-3). The Batch Tasks form opens when you click View Tasks on the menu bar in the Batch Job form. From the Batch Tasks form, you can also change the status of batch tasks or delete tasks that are no longer needed.

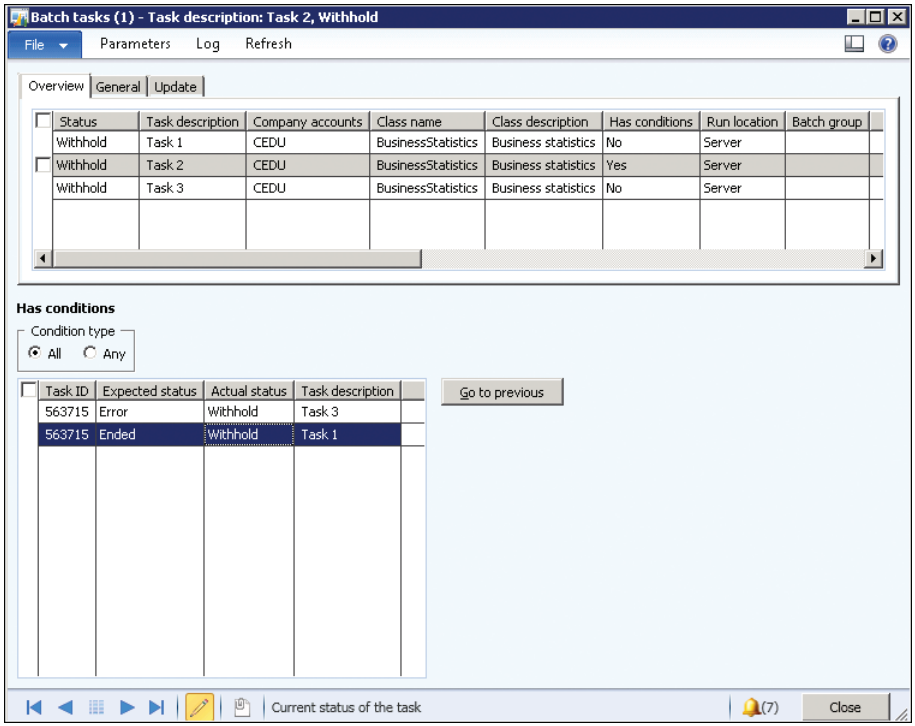


FIGURE 18-3 The Batch Tasks form.

To create a task, do the following:

1. Press Ctrl+N to create the task.
2. In Task Description, enter a description of the task.
3. In Company Accounts, select the company in which the task runs.
4. In Class Name, select the process that you want the task to run. Classes appear in a lookup list containing all available batch-enabled classes. The lookup list appears only if the *CanGoBatchJournal* property is enabled.
5. In Batch Group, select a batch group for the task if necessary.
6. Save the task by pressing Ctrl+S.
7. Specify class parameters if necessary. As mentioned in previous sections, each batch task represents a batch-executable class. Sometimes you need to set up parameters for that class. For example, you might need to specify posting parameters for invoice posting. To do that, click Parameters on the menu bar in the Batch Tasks form. A dialog box specific to the selected class is displayed.

Note If you are creating a custom batch class, you must design the parameters form manually. If you implement a batch based on the SysOperation framework, this process is highly simplified. After you specify the necessary parameters and click OK, the class parameters are packed and saved in the *Batch* table and then are restored when the class executes. For more information about the SysOperation framework, see Chapter 14.

8. Set up dependencies or advanced sequencing between tasks, if necessary.

After you create the batch job and add tasks to it, you can use the Batch Tasks form to define dependencies between the tasks. If no dependencies or conditions are defined within a job, the batch server automatically executes the tasks in parallel. (To configure the maximum number of parallel tasks, use the Maximum Batch Threads parameter in the Server Configuration form.)

If you need to use advanced sequencing to accommodate your business process flow, you can use either the Batch Tasks form or the Batch API. You can use these tools to construct complex dependency trees that let you schedule batch jobs tasks in parallel, add multiple dependencies between batch tasks, choose different execution paths based on the results of the previous batch task, and so on.

For example, suppose that the job, JOB1, has seven tasks: TASK1, TASK2, TASK3, TASK4, TASK5, TASK6, and TASK7, and you want to set up the following sequence and dependencies for it:

- TASK1 runs first.
- TASK2 runs on completion (Ended or Error) of TASK1 (regardless of the success or failure of TASK1).
- TASK3 runs on success (Ended) of TASK2.
- TASK4 runs on success (Ended) of TASK2.
- TASK5 runs on failure (Error) of TASK2.
- TASK6 runs on failure (Error) of TASK3.
- TASK7 runs on success (Ended) of both TASK3 and TASK4.

Figure 18-4 shows the dependency tree for JOB1.

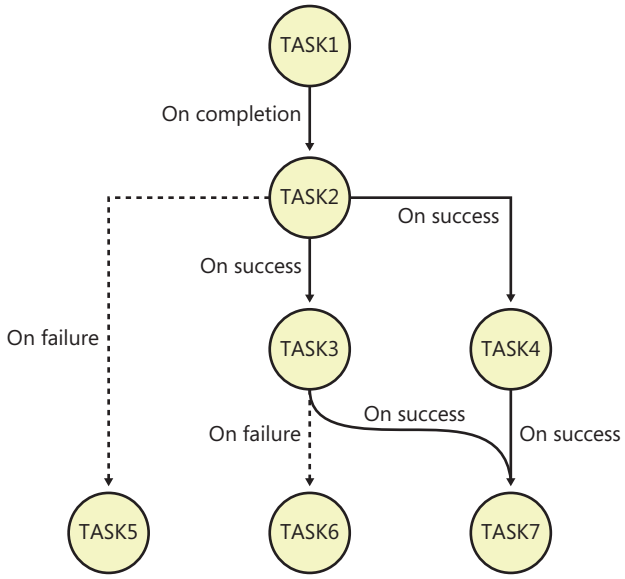


FIGURE 18-4 The batch task dependency tree for JOB1.

To define these task dependencies and to tell the system how to handle them, select a child task—for example, TASK2—from the preceding list, and then do the following:

1. In the Batch Tasks form, click in the Has Conditions grid, and then press Ctrl+N to create a new condition.
2. Select the task ID of the parent task, such as TASK1.
3. Select the status that the parent task must reach before the dependent task can run. For example, TASK2 starts when the status of TASK1 becomes Ended or Error.
4. Press Ctrl+S to save the condition.
5. If you enter more than one condition, and if all conditions must be met before the dependent task can run, select a condition type of All.

   Alternatively, if the dependent task can run after any of the conditions are met, select a condition type of Any.

You can use the Batch Tasks form to define how the system handles task failures. To ignore the failure of a specific task, select Ignore Task Failure for that task on the General tab. If you select this option, the failure of the task doesn't cause the job to fail. You can also use Maximum Retries to specify the number of times a task should be retried before it fails.

Use the Batch API

For advanced scenarios requiring complex or large batch jobs, such as inventory closing or data upgrades, the batch framework provides an X++ API that you can use to create or modify batch jobs, tasks, and their dependencies as needed; and to create runtime batch tasks dynamically. This flexible API helps you automate task creation and integrate batch processing into other business processes. It can also be useful when your batch job or task requires additional logic. To create a batch job by using the Batch API, the following steps are necessary:

1. Use the *BatchHeader* class to create the batch job.
2. Modify the parameters for the batch job.
3. Add tasks.
4. Define the dependencies between tasks.
5. Save the batch job.

Create a batch job by using the *BatchHeader* class

Create an instance of the *BatchHeader* class that represents your batch job. The following example creates a *BatchHeader* instance named *sampleBatchHeader*:

```
sampleBatchHeader = BatchHeader::construct();
```

You can also construct a *BatchHeader* object for an existing batch job by providing an optional *batchJobId* parameter for the *construct* method, as shown here:

```
//job1 is an existing job
sampleBatchHeader = BatchHeader::construct(job1.parmCurrentBatch().BatchJobId);
```

Modify batch job parameters

The *BatchHeader* class lets you access and modify most parameters by using *parm* methods. For example, you can set up recurrences and alerts for your batch job, as shown in the following example:

```
// Set the batch recurrence
sysRecurrenceData = SysRecurrence::defaultRecurrence();
sysRecurrenceData = SysRecurrence::setRecurrenceStartDateTime(sysRecurrenceData,
DateTimeUtil::utcNow());
sysRecurrenceData = SysRecurrence::setRecurrenceNoEnd(sysRecurrenceData);
sysRecurrenceData = SysRecurrence::setRecurrenceUnit(sysRecurrenceData, SysRecurrenceUnit::Hour,
1);
sampleBatchHeader.parmRecurrenceData(sysRecurrenceData);

// Set the batch alert configurations
sampleBatchHeader.parmAlerts(NoYes::No, NoYes::Yes, NoYes::No, NoYes::No, NoYes::No);
```

Add a task to the batch job

You add tasks to the batch job by calling the *addTask* method. The first parameter for this method is an instance of a batch-executable class that is scheduled to execute as a batch task:

```
void addTask(Batchable batchTask,
 [BatchConstraintType constraintType])
```

Another way to create a task is to use the *addRuntimeTask* method, which creates a dynamic batch task. This task exists only for the current run; it is copied into the history tables and deleted at the end of the run. It copies settings such as the batch group and child dependencies from the *inheritFromTaskId* task:

```
void addRuntimeTask(Batchable batchTask,
RecId inheritFromTaskId,
[BatchConstraintType constraintType])
```

Define dependencies between tasks

The *BatchHeader* class provides the *addDependency* method, which you can use to define a dependency between the *batchTaskToRun* and *dependsOnBatchTask* tasks.

You can use the optional parameter *batchStatus* to specify the type of the dependency. By default, a dependency of type *BatchDependencyStatus::Finished* is created, which means that a task starts execution only if the task that it depends on finishes successfully. Other options are *BatchDependencyStatus::Error* (the task starts execution if the preceding task finishes with an error) and *BatchDependencyStatus::FinishedOrError* (the task starts execution if the preceding task finishes with any status result). The following example shows the signature of the *addDependency* method:

```
public BatchDependency addDependency(
    Batchable batchTaskToRun,
    Batchable dependsOnBatchTask,
   [BatchDependencyStatus batchStatus])
```

Save the batch job

The final step in creating the batch job using the Batch API is to save the job by calling the *batchHeader.save* method. The *save* method inserts records into the *BatchJob*, *Batch*, and *BatchConstraints* tables, from which the batch server can automatically pick them up for execution.

Example of a batch job

The following example shows how to create a batch job and add two batch tasks by using the Batch API. The example assumes that a batch-enabled class named *ExampleBatchTask* already exists:

```
static void ExampleSchedulingJob (Args _args)
{
    BatchHeader     sampleBatchHeader;
    RunBaseBatch    sampleBatchTask;
```

```
    // create batch header
    sampleBatchHeader = BatchHeader::construct();

    // create and add batch tasks
    sampleBatchTask1 = new ExampleBatchTask();

    sampleBatchHeader.addTask(sampleBatchTask1);

    sampleBatchTask2 = new ExampleBatchTask();

    sampleBatchHeader.addTask(sampleBatchTask2);

    // add dependencies between batch tasks
    sampleBatchHeader.addDependency(sampleBatchTask1, sampleBatchTask2);

    // save batch job in the database
    sampleBatchHeader.save();
}
```

For more examples of programmatic batch job creation, see "Walkthrough: Extending *RunBaseBatch* Class to Create and Run a Batch," at *http://msdn.microsoft.com/en-us/library/cc636647.aspx*.

Manage batch execution

The final step in implementing a batch job is to manage the execution process. Before a batch job can be executed on an AOS instance, you must configure the AOS instance as a batch server and set up the batch groups that tell the system which AOS instance should execute the job.

In addition to these initial configuration tasks, you'll likely need to manage the batch tasks and jobs: checking status, reviewing history, and sometimes canceling a batch job. You'll probably also need to debug a batch task at some point. The following sections describe how to configure an AOS instance as a batch server, set up batch groups, manage batch jobs, and debug a batch task.

Configure the batch server

You can configure an AOS instance to be a batch server, including specifying when the batch server is available for processing and how many tasks it can run, by using the Server Configuration form. The Server Configuration form is located at System Administration > Setup > System > Server Configuration. Note that the first AOS instance is automatically designated as a batch server, but you can configure additional AOS instances manually as batch servers.

Tip Use multiple batch servers to enable parallel processing and increase processing throughput.

1. In the Server Configuration form, select a server in the left pane.
2. Select the Is Batch Server check box to enable batch processing on the server, as shown in Figure 18-5.

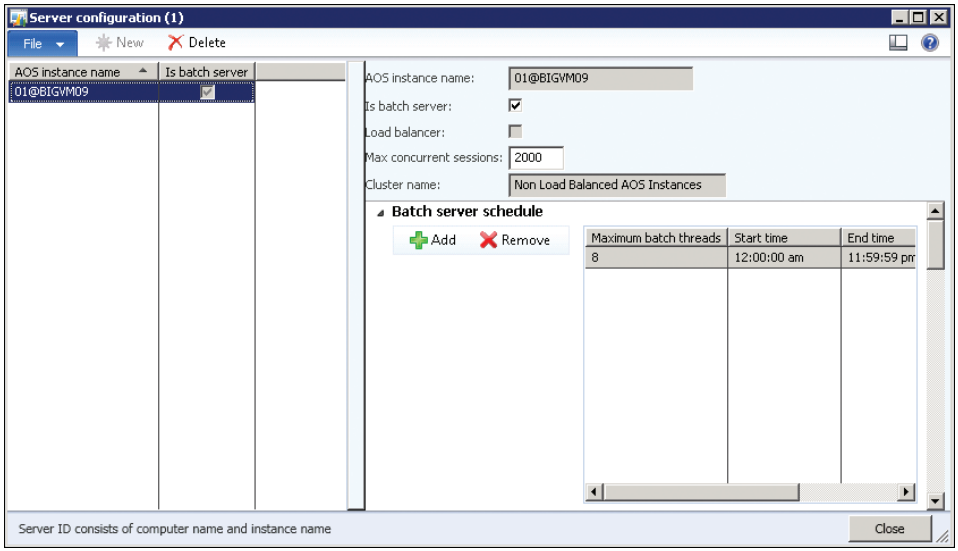


FIGURE 18-5 The Server Configuration form.

3. On the Batch Server Schedule FastTab, click Add to enter a new schedule. Enter the maximum number of batch tasks that can be run on the AOS instance at one time. The server continues to pick up tasks from the queue until it reaches its maximum.
4. Enter a starting time in Start Time and an ending time in End Time to specify the time window in which the server processes batch jobs. Press Ctrl+N to enter an additional time window.

Tip It's a good idea to exclude a server from batch processing when it is busy processing regular transactions. You can set server schedules so that each AOS instance is available for user traffic during the day and batch traffic overnight. Keep in mind that if the server is running a task when its batch processing availability ends, the task continues running to completion. However, the server doesn't pick up any more tasks from the queue.

Create a batch group

A batch group is a logical categorization of batch tasks that lets a user (typically a system administrator) determine which AOS instance runs the batch task. This section describes how to create a batch group so that it can be assigned to a specific server for execution. The first step is to create batch groups by using the Batch Group form at System Administration > Setup > Batch Group.

To create a batch group, press Ctrl+N in the Batch Group form, and then type a name and description for the batch group. The Batch Group form is shown in Figure 18-6.

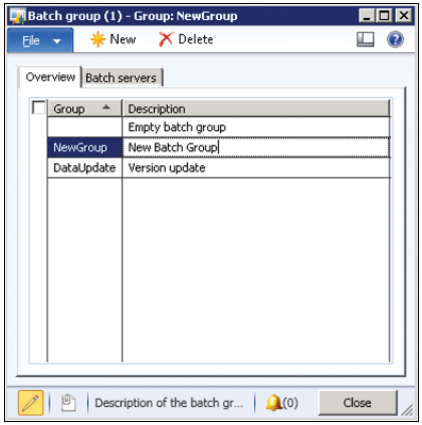


FIGURE 18-6 The Batch Group form.

Note By default, the system contains an empty batch group that can't be removed. This is a default batch group for tasks that are not explicitly assigned to a group.

After you create batch groups, assign each group to a server as follows:

1. In the Server Configuration form (shown in Figure 18-7), click the Batch Server Groups FastTab. The Selected Groups list shows the batch groups specified to run on the selected server.
2. In the Remaining Groups list, select a group, and then click the left arrow button to add this group to run on the selected server.

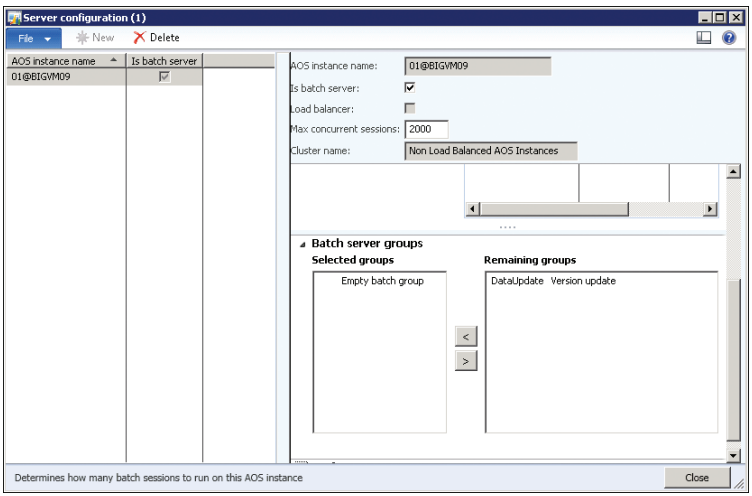


FIGURE 18-7 The Server Configuration form.

Manage batch jobs

After you create and schedule a batch job, you might want to check its status, review its history, or cancel it. The following sections describe some of the most common management tasks associated with batch jobs.

View and change the batch job status

The Batch Job list form provides a snapshot view of the current state of batch jobs. The list displays the progress and the status of running and completed jobs. It also displays any jobs that are scheduled to start soon.

You can change the status of a batch job by selecting the batch job in the list and then following these steps:

1. Click Functions, and then click Change Status.
2. In the Select New Status dialog box, select a new status for the job. For example, if the status is Waiting, you can temporarily remove the batch job from the waiting list by changing the status to Withhold.

Tip If a job exits with a status of Error/Ended and you want to rerun the job, change its status to Waiting. The job will automatically be picked up by the server for execution.

You can cancel a batch job by changing its status to Canceling. Tasks in the Waiting or Ready state are changed to Not Run; currently executing tasks are interrupted, and their status is changed to Canceled.

Control the maximum retries

If an AOS fails because of an infrastructure failure or a power outage while a batch task is executing, the batch framework has the built-in capability to retry tasks after the AOS is restarted. Any tasks that were left in an executing state, and that have not reached the maximum retry limit, are changed to the Ready state and will run shortly after the failure.

Tip If you create custom tasks and want to enable retries, design the task so that it is idempotent—that is, it can be executed multiple times without unexpected consequences.

You can modify the Maximum Retries attribute for each batch task on the General tab. By default, the value is set to 1; when the Actual Retries field on the Update tab exceeds the maximum number of retries, the batch task fails. When this happens, the recurrence that is set for the batch job is not honored, and the status of the batch job is set to either Success or Error.

Review the batch job history

You can view a history of all batch jobs that have finished running in the Batch Job History form at System Administration > Inquiries > Batch Job History. This form displays detailed information about the status of the jobs, including any messages encountered while the batch job was running.

You can also view the logs for each batch job as follows:

- To view log information for an entire batch, select a batch job, and then click Log.
- To view log information for individual tasks, select a batch job, and then click View Tasks. In the Batch History list form, select a task, and then click Log.

Tip In the batch job settings, you can specify when log information is written to the history tables: Always (the default), On Error, or Never. Use On Error or Never to save disk space for batch jobs that run constantly. This option is located on the General tab of the Batch Job form.

Debug a batch task

Because batch tasks run in noninteractive mode and X++ executes in the common language runtime (CLR), to debug a batch task, you have to perform additional steps to configure the AOS and the Visual Studio debugger, in addition to setting up breakpoints.

First, configure the AOS for batch debugging. This is necessary for two reasons. First, the AOS modifies the X++ assembly to disable Just-In-Time (JIT) optimizations in the CLR. This is necessary to enable variables and object contents to be viewed and analyzed in the debugger. Second, the AOS produces source files containing the X++ code under the server Bin\XppIL\Source folder. You can open these files in Visual Studio to set breakpoints and perform common tasks, such as step ping into and step ping over.

Configure AOS for batch debugging

Use the Microsoft Dynamics AX Server Configuration Utility (see Figure 18-8) to configure the AOS for batch debugging. The utility is available on the computer on which you installed the AOS. To do this, perform the following steps:

1. Open the Microsoft Dynamics AX Server Configuration Utility. Click Start > All Programs > Administrative Tools > Microsoft Dynamics AX 2012 Server Configuration.
2. Select the Enable Breakpoints to Debug X++ Code Running on This Server check box.
3. Click OK to close the utility, and then restart the AOS.

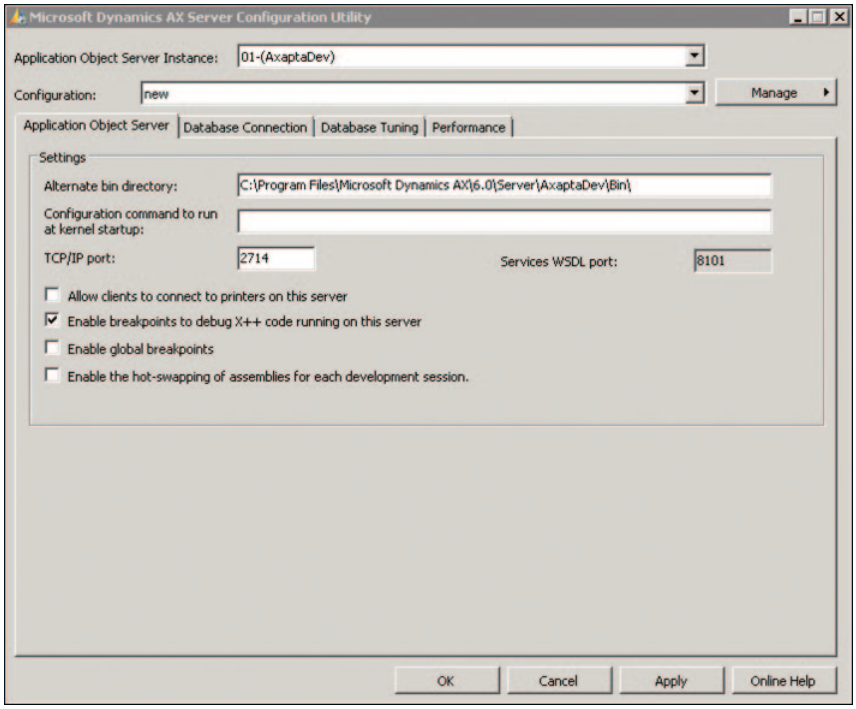


FIGURE 18-8 The Microsoft Dynamics AX 2012 Server Configuration Utility.

Configure Visual Studio for debugging X++ in a batch

To configure Visual Studio for batch debugging, attach to the AOS process Ax32Serv.exe by following these steps:

1. In Visual Studio, on the Debug menu, click Attach To Process.
2. When the Attach To Process dialog box opens (see Figure 18-9), click Select to select Managed (v4.0) code, and then select the following check boxes:
   - Show Processes From All Users
   - Show Processes In All Sessions
3. Click Ax32Serv.exe, and then click Attach.

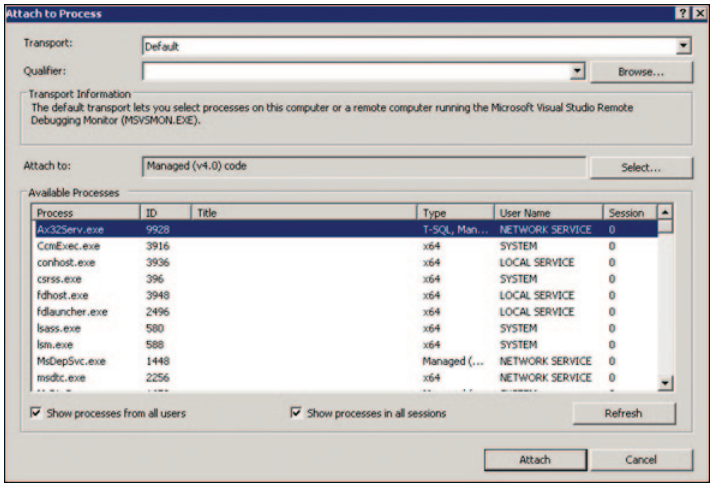


FIGURE 18-9 The Attach To Process dialog box in Visual Studio.

The next step is to disable the Just My Code option, so that the debugger breaks on X++ source code. To do this, perform the following steps:

1. On the Tools menu, click Options, and then navigate to the Debugging\General node (shown in Figure 18-10).

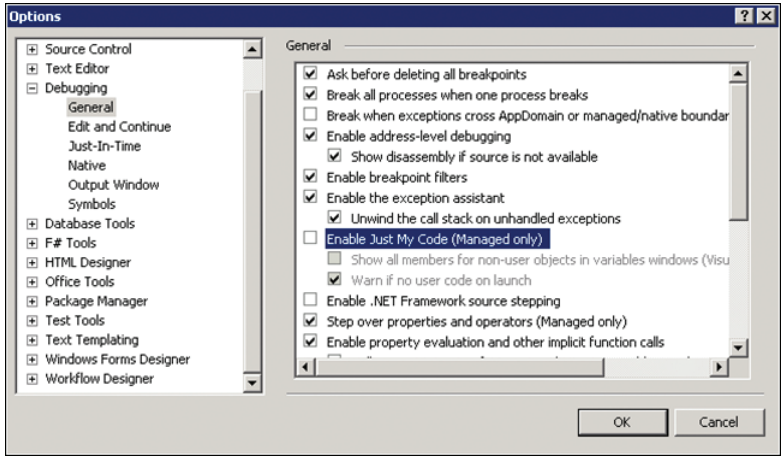


FIGURE 18-10 The Options dialog box in Visual Studio.

Clear the Enable Just My Code (Managed Only) check box, and then click OK.

After you complete these steps, you can open the X++ source code from the Bin\XppIL\Source folder on the server and set breakpoints.

CHAPTER 19

Application domain frameworks

In this chapter

Introduction

Microsoft Dynamics AX 2012 includes a number of new domain-specific application frameworks that improve code reuse between application modules, reduce the need to assign artificial relationship roles in application modules, and reduce the number of tightly coupled interdependencies between application modules. For example:

- The operations resource role that is assigned to people and work centers by the operations resource framework eliminates the need to assign employees and independent contractors artificially to the work center role, so that they can participate in production planning and scheduling activities.
- The performance dimensions extended from the dimension framework, the double-entry subledger journal provided by the accounting framework, and the source document abstraction provided by the source document framework enable operations processes such as purchasing, receiving, and invoicing to be decoupled from accounting processes such as budget control and financial reporting.

This chapter provides a short description of the conceptual foundation of some of the key application domain frameworks, in addition to links to white papers that provide more detailed implementation guidelines and code samples. This chapter does not contain an exhaustive list of application domain frameworks for Microsoft Dynamics AX 2012. Instead, it is intended to provide an overview of important new functionality and suggestions about how you can use it. For information about additional application domain frameworks, see "White Papers for Developers" at *http://technet.microsoft.com/EN-US/library/hh272882*. This page is updated frequently with links to new white papers.

Note By convention, the names of entities in the conceptual domain models in this chapter are denoted in title case and italicized for emphasis.

The organization model framework

Microsoft Dynamics AX 2012 introduces a new organization model framework. This framework is designed to model key scenarios that are required by government organizations and corporations that have global operations, and those that have separate legal and operating organization structures. The organization model framework extends the company feature that was used in Microsoft Dynamics AX 2009 and earlier versions.

With the new types of organizations that are included in Microsoft Dynamics AX 2012, you can model organizations in Microsoft Dynamics AX to mirror the way you operate them without having to customize the application. Most organizations go through an iterative operating cycle of monitor, measure, analyze, and improve. The analysis phase results in new business rules and policies and new strategic and operational initiatives to improve the organization's performance. With the organization framework, you can create hierarchical structures that enable this cycle of performance improvement.

How the organization model framework works

The organization model has two major components: *Organization Types* and *Organization Hierarchies*. The following sections describe these components.

Organization types

The organization model in Microsoft Dynamics AX 2012 introduces two new types of organizations: *Legal Entity* and *Operating Unit*.

- ***Legal Entity*** An organization with a registered or legislated legal structure that has the authority to enter into legal contracts and that is required to prepare statements that report on its performance. A legal entity and a company in Microsoft Dynamics AX 2012 are semantically the same. However, some functional areas in the application are still based on a data model that uses the concept of a company (or DataArea). These areas may have the same limitations as in Microsoft Dynamics AX 2009 and may have an implicit data security boundary.
- ***Operating Unit*** An organization that divides the control of economic resources and operational processes among people who have a duty to maximize the use of resources, to improve processes, and to account for their performance.

Microsoft Dynamics AX 2012 includes several types of operating units:

- **Business unit** A semi-autonomous *Operating Unit* that is created to meet strategic business objectives.
- **Cost center** A type of *Operating Unit* that describes an organization that is used to track costs or expenses. A cost center is a cost accumulator, and it is used to manage costs.
- **Department** A type of *Operating Unit* that may have profit and loss responsibility and might consist of a group of cost centers. Departments also are often created based on functional responsibility or skill, such as sales and marketing.
- **Value Stream** A type of *Operating Unit* that is commonly used in lean manufacturing. In lean manufacturing, a value stream owns one or more production flows that describe the activities and flows needed to supply a product, an item, or a service to the consumers of the product.
- **Retail Channel** A type of *Operating Unit* that is commonly used in retail to represent a retail channel.

A *Team* is also a type of *Internal Organization*, but it is an informal group of people that is typically created for a specific purpose over a short duration. Teams might be created for specific projects or services. The other types of organizational units described here are more permanent, although they could require frequent minor updates or major changes because of restructuring.

These types of operating units support application functionality in Microsoft Dynamics AX 2012. However, every industry and business has unique requirements for its operating units and may call them by different names. Additionally, organizations can create custom types of operating units to meet their needs. For more information, see the section "Create a custom operating unit type," later in this chapter.

When you arrange legal entities and operating units into hierarchies and use them for aggregated reporting, to secure access to data, and to implement business policies, they help enable internal control of your organization.

Organization hierarchies

An *Organization* is a group of people who work together to perform operational and administration processes. *Organization Hierarchies* represent the relationships between the *Organizations* that make up an enterprise or a government entity. In previous releases, companies could not be organized into a hierarchy to represent the structure of an organization. In reality, organizations typically have a hierarchical structure for the reasons mentioned in the previous section.

The organization model framework in Microsoft Dynamics AX 2012 supports the creation of multiple hierarchies that take effect on multiple dates. This is useful in restructuring scenarios, where you want the updated hierarchy to become effective at a certain date in the future. The framework also supports hierarchies that are used for multiple purposes.

A *Purpose* defines how the *Organizational Hierarchy* is used in application scenarios. The *Purpose* that you select determines the types of organizations that can be included in the hierarchy. Table 19-1 shows the types of organizations that you can use for each hierarchy purpose that is included in Microsoft Dynamics AX 2012.

TABLE 19-1 Purposes and organization types.

Purpose	Description	Organization types
Procurement internal control	Use this purpose to define policies that control the purchasing process.	All
Expenditure internal control	Use this purpose to define policies for expense reports.	All
Organization chart	Use this purpose in human resources to define reporting relationships.	All
Signature authority internal control	Use this purpose to define policies for signing limits. These policies control the spending and approval limits that are assigned to employees.	All
Vendor payment internal control	Use this purpose to define policies for the payment of vendor invoices.	Legal entities
Audit internal control	Use this purpose to define policies for identifying documents for audit.	Legal entities
Centralized payments	Use this purpose to make payments by one legal entity on behalf of other legal entities.	Legal entities
Security	Use this purpose to define the data security access for organizations.	All
Retail assortment	Use this purpose to define assortments for retail channels.	All
Retail replenishment	Use this purpose to define replenishment configurations.	All
Retail reporting	Use this purpose to define dimensions for retail reporting cubes.	All

The organization model has a significant impact on the implementation of Microsoft Dynamics AX 2012 and on the business processes being implemented. Executives and senior managers from different functional areas such as finance and accounting, human resources, operations, and sales and marketing should participate in defining the organization structures.

Figure 19-1 shows the conceptual domain model of the organization model framework.

When to use the organization model framework

You use the organization model framework to model how the business operates. You can use the organization model framework in two ways: by using the built-in integration with other application frameworks and existing Microsoft Dynamics AX modules, or by modeling custom scenarios to meet the needs of your organization.

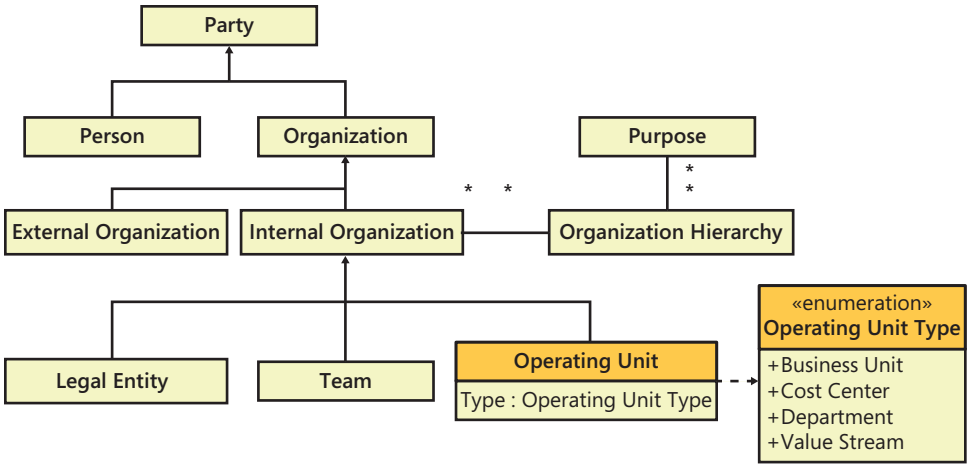


FIGURE 19-1 The organization model framework.

Integration with other frameworks application modules

The organization model framework is inherently integrated with certain frameworks and modules in Microsoft Dynamics AX:

- **Address book** All internal organizations—legal entity, operating unit, and team—are types of the *Party* entity. This means that these organizations can use the capabilities of the address book to store address and contact information. For more information, see the white paper, "Implementing the Global Address Book Framework" at *http://technet.microsoft.com/en-us/library/hh272867.aspx*.
- **Financial dimensions** You can use legal entities and operating units to define financial dimensions and then use those financial dimensions in account structures. By using organizations as financial dimensions, an enterprise or government entity can analyze an organization's financial performance. If two types of organizations are used as separate financial dimensions in the account structure, the relationships between organizations described through hierarchies can also be used as constraints. For more information, see the section "The dimension framework" later in this chapter.
- **Policy framework** You can use the policy framework to define an internal control policy for an organization. The policy framework can be used to define policies for expense reports, purchase requisitions, audit control of documents, and vendor invoice payments. The policy framework provides support for override and default behavior for organizations based on their hierarchies, and enables internal management control of organizations to facilitate cost control, fraud detection, better operating efficiency, and better performance in general. For more information, see the white paper, "Using the Policy Framework," at *http://technet.microsoft.com/en-us/library/hh272869.aspx*.

- **Extensible data security** The new extensible data security framework provides capabilities to secure data based on any condition. Security access to organizations can be defined based on hierarchies. For more information, see Chapter 11, "Security, licensing, and configuration."

The organization model framework is also used in the following application modules:

- **Procurement and sourcing** The lines of a purchase requisition are created for a buying legal entity, and they are received by an operating unit, such as a cost center or a department. The organization model framework enables various scenarios by allowing the viewing or creation of purchase requisitions for anyone buying legal entities and receiving operating units in which you have access to create purchase requisitions.
- **Human resources** In human resources, workers hold employment contracts in a legal entity and have a position in a department. All transaction scenarios in human resources use these concepts to view and modify data.
- **Travel and expense** Expense reports and expense line items are associated with a legal entity to which the expense line item should be charged from a statutory perspective, and they also are associated with an operating unit for internal reporting.

Model your own functional scenarios

You can use the organization model framework to model your own scenarios. For example, a common scenario for data security is to filter application data based on a user's roles and membership in internal organizations. For example, organizations might seek to limit an individual account manager's access to specific sales orders based on geography, allowing her to view only the sales orders that originate in her region.

You can set up a new scenario or customize an existing scenario by taking the following high-level steps:

1. Define or change the data model.
2. Create a new table with the *SaveDataPerCompany* property set to *No*. If you are working with existing tables that are marked per-company, change the value of the *SaveDataPerCompany* property from *Yes* to *No*.
3. Reference organizations as foreign keys (FKs) on the table. It may be necessary to reference an operating unit and a legal entity if the legal entity cannot be established through legal entity or operating unit organization hierarchies.
4. If the table includes redundant data in the *Legal Entity* field, set up hierarchical constraints between legal entities and operating units to maintain data consistency.
5. Build a new form (for example, a list page) for the scenarios, or change the existing user experience to view or maintain data. You can use custom filters to make it possible for users to view and maintain data across organizations.

6. Apply default organizations on the table in financial dimensions by including them in account structures.
7. Create extensible data security policies that are based on the organizations that the user belongs to or has access to.
8. Use the policy framework to set up policies to apply when users access data in the scenario.

Extending the organization model framework

You can extend the organization model framework by creating a custom type of operating unit or a custom purpose, or by extending the hierarchy designer, a tool that is included with Microsoft Dynamics AX 2012.

Create a custom operating unit type

A core extensibility scenario is to extend the organization model to accommodate specific vertical industry requirements. For example, branches, schools, and school districts are essentially organization concepts, and you can model them as new types of operating units.

Suppose that you want to create an operating unit called *Branch*. To do so, you would follow these steps:

1. Create a new base enum value for the new *Operating Unit Type*.
2. Create a view.
3. Create a menu item for the new operating unit.

The following sections describe these steps in more detail.

Create a new base enum value Define a new base enum value for the *OMOperatingUnitType* enum, which corresponds to the new type of operating unit:

1. In the Application Object Tree (AOT), navigate to Data Dictionary\Base Enums\ OMOperatingUnitType.
2. Right-click the *OMOperatingUnitType* enum, click New Element, and then add an element named *Branch*.
3. In the Properties window for the new element, set both the *Name* and *Label* properties to *Branch*. Change the default *EnumValue* property to an integer value that will prevent clashes between your new enum and future enums added by the Microsoft Dynamics AX team.

Create a view Define a view, DimAttributeBranchView, which is similar to views created for other types of operating units. For example, the view DimAttributeOMBusinessUnit was created for business units. The DimAttributeOMBusinessUnit view contains the fields that allow the OMOperatingUnit table to be used as a financial dimension for all business units specified.

1. In the AOT, navigate to the Data Dictionary\Views node, and then locate DimAttributeOMBusinessUnit.
2. Duplicate DimAttributeOMBusinessUnit: Right-click the *DimAttributeOMBusinessUnit* node, and then click Duplicate. The AOT will create a node called *CopyOfDimAttributeOMBusinessUnit*.
3. In the Properties window for the newly created view, set the properties as shown in the following table:

Property	Value
Name	*DimAttributeBranchView*
Label	*Branches*
SingularLabel	*Branch*
DeveloperDocumentation	Type a description for the view. For example, The *DimAttributeBranchView* contains all records from the OMOperatingUnit table that are specified as branches.

4. Under the new DimAttributeBranchView view, locate the OMOperatingUnitTypeOMBusinessUnit range. The range is under the Metadata\Data Sources\BackingEntity(OMOperatingUnit)\Ranges node.

In the Properties window for the OMOperatingUnitTypeOMBusinessUnit range, set the properties as shown in the following table:

Property	Value
Name	*OperatingUnitTypeBranch*
Field	*OMOperatingUnitType*
Value	*Branch*

Create a menu item Finally, create a menu item for the new operating unit type as follows:

1. Under Menu Item\Display, create a menu item named *BranchMenuItem*.
2. In the Properties window for *BranchMenuItem*, set the following properties:

Property	Value
Name	*BranchMenuItem*
Label	*Branches*
Object	*OMOperatingUnit*
EnumTypeParameter	*OMOperatingUnitType*
EnumParameter	*Branch*

The new operating unit type will now appear in the list of operating unit types that are available when a system administrator creates a new operating unit.

Create a custom purpose

You can extend the Microsoft Dynamics AX organization model to create a custom purpose. A purpose defines how the organization hierarchy is used in application scenarios.

Suppose you want to create a new purpose called Sales. To do so, you would follow these steps:

1. Create a new base enum value for the new purpose.
2. Create a method to add the new purpose, and then call that method to add the purpose to the HierarchyPurposeTable table.

The following sections describe these steps in more detail.

Create a new base enum value First, create a new base enum value for the new purpose. To do so, you follow these steps:

1. In the AOT, navigate to Data Dictionary\Base Enums\HierarchyPurpose.
2. Right-click the *HierarchyPurpose* enum, click New Element, and then add an element named *Sales*.
3. In the Properties window for the new element, set the Name and Label properties to *Sales*. Change the default EnumValue to an integer value that will prevent clashes between your new enum and future enums that are added by Microsoft or independent software vendors (ISVs).

Create and call a method to add the new purpose Next, create the method to add the new purpose, and then call the method to add it to the table.

1. In the *Classes* node in the AOT, locate *OMHierarchyPurposeTableClass*.
2. Duplicate the *addSecurityPurpose* method: Right-click the *addSecurityPurpose* node, and then click Duplicate. The AOT will create a method called *CopyOfaddSecurityPurpose*.
3. Replace the code for the *CopyOfaddSecurityPurpose* method with the following code. This code renames the method:

```
private static void addSalesPurpose()
{
    OMHierPurposeOrgTypeMap omHPOTP;
    select RecId from omHPOTP
        where omHPOTP.HierarchyPurpose == HierarchyPurpose::Sales;
    if (omHPOTP.RecId <= 0)
    {
        omHPOTP.clear();
        omHPOTP.HierarchyPurpose = HierarchyPurpose::Sales;
        omHPOTP.OperatingUnitType = OMOperatingUnitType::OMAnyOU;
        omHPOTP.IsLegalEntityAllowed = NoYes::No;
        omHPOTP.write();
        omHPOTP.clear();
        omHPOTP.HierarchyPurpose = HierarchyPurpose::Sales;
        omHPOTP.OperatingUnitType = 0;
        omHPOTP.IsLegalEntityAllowed = NoYes::Yes;
        omHPOTP.write();
    }
}
```

The preceding code is similar to the code in most of the methods of the *OMHierarchyPurposeTableClass* class. The code was changed in only the places where the *Hierarchy Purpose* enum values are referenced. In the code, you can see three occurrences of HierarchyPurpose::Sales.

4. In the *OMHierarchyPurposeTableClass* class, update the *populateHierarchyPurposeTable* method to call the new method that you created, by adding the following line of code:

```
OMHierarchyPurposeTableClass::addSalesPurpose();
```

The following code shows the modification to the *populateHierarchyPurposeTable* method:

```
public static void populateHierarchyPurposeTable()
{
    OMHierPurposeOrgTypeMap omHPOTP;
    if (omHPOTP.RecId <= 0)
    {
        ttsbegin;
        OMHierarchyPurposeTableClass::AddOrganizationChartPurpose();
        OMHierarchyPurposeTableClass::AddInvoiceControlPurpose();
        OMHierarchyPurposeTableClass::AddExpenseControlPurpose();
        OMHierarchyPurposeTableClass::AddPurchaseControlPurpose();
        OMHierarchyPurposeTableClass::AddSigningLimitsPurpose();
        OMHierarchyPurposeTableClass::AddAuditInternalControlPurpose();
        OMHierarchyPurposeTableClass::AddCentralizedPaymentPurpose();
        OMHierarchyPurposeTableClass::addSecurityPurpose();
            //Add the following line.
        OMHierarchyPurposeTableClass::addSalesPurpose();
        ttscommit;
    }
}
```

After you complete these steps, the new purpose will appear under Organization Administration > Setup > Organization > Organization Hierarchy Purposes.

Extend the hierarchy designer

System administrators can view or modify organizational hierarchies by using the hierarchy designer. This form is available through Organization Administration > Setup > Organization > Organization Hierarchies.

Developers have a few options for extending the hierarchy designer. The hierarchy designer control can be customized for four parameters of the organization nodes within the hierarchy: border color, node image, top gradient color, and bottom gradient color. For more information, download the white paper "Implementing and Extending the Organization Model in Microsoft Dynamics AX 2012" from *http://download.microsoft.com/download/4/E/3/4E36B655-568E-4D4A-B161-152B28BAAF30/Implementing_and_extending_the_organization_model_in_Microsoft_Dynamics_AX_2012.pdf*.

The product model framework

Microsoft Dynamics AX 2012 offers a flexible product data management framework, supporting both centralized and legal-entity-specific management of information about a product, which is defined as an item or a service that results from an economic activity.

How the product model framework works

In the product model, product information is centralized around the concept of a *Product*, which represents information that is shared across the organizational structure, and the concept of a *Released Product*, which controls information that is specific to a legal entity. Figure 19-2 shows the conceptual domain model of the product model framework.

Product types and subtypes

A *Product* can be an *Item*—which represents a physical entity for which inventory levels can be tracked, like finished goods or raw components—or a *Service*—modeling a nonphysical entity for which inventory levels are not tracked, like providing consulting services. A *Product* can be divided further into additional subtypes: a *Distinct Product* that is uniquely identifiable and can be used in economic activities, or a *Product Master*. A *Product Master* is a standard or functional product representation that serves a basis for configuring distinct *Product Variants*. In this case, a *Product Variant* is a uniquely identifiable *Product* that can be used in economic activities. However, because all *Product Variants* are bound to a *Product Master*, they share a significant part of the information defined for the *Product Master*.

Consider a manufacturing company named Contoso that sells different models of bicycles, which come in different colors and sizes, and accessories such as bicycle lights. In this scenario, every bicycle model can be modeled as a *Product Master*, with each color and size combination defining a *Product Variant*. Lights do not come in various configurations, so they can be modeled as distinct *Products*.

Product dimensions

A *Product Variant* is defined by using *Product Dimensions*, which are active for a given *Product Master*. *Product Dimensions* are characteristics that uniquely identify a product. Microsoft Dynamics AX 2012 includes four *Product Dimensions*: Configuration, Size, Color, and Style. (You can rename them.) A *Product Dimension Group*, which is required for a *Product Master*, encompasses the information about the *Product Dimensions* that are active and can be used for controlling which *Product Dimensions* should be considered in the price calculation in trade agreements. Because *Product Dimensions* define unique *Products*, a *Product Dimension Group* must be assigned to the *Product Master* and is shared across the organizational structure.

For example, Contoso sells two bicycle models. Each model is available in three sizes (small, medium, and large) and three colors (black, red, and white). Contoso can model this assortment by creating a *Product Dimension Group* named Bicycles, with Size and Color as the active *Dimensions*, and assigning that *Product Dimension Group* to the two *Product Masters* that represent the two bicycle models. Because Contoso also sells helmets, which come in one color but in different sizes, Contoso can create a second *Product Dimension Group* called *Helmets* that has only one active *Dimension*: Size.

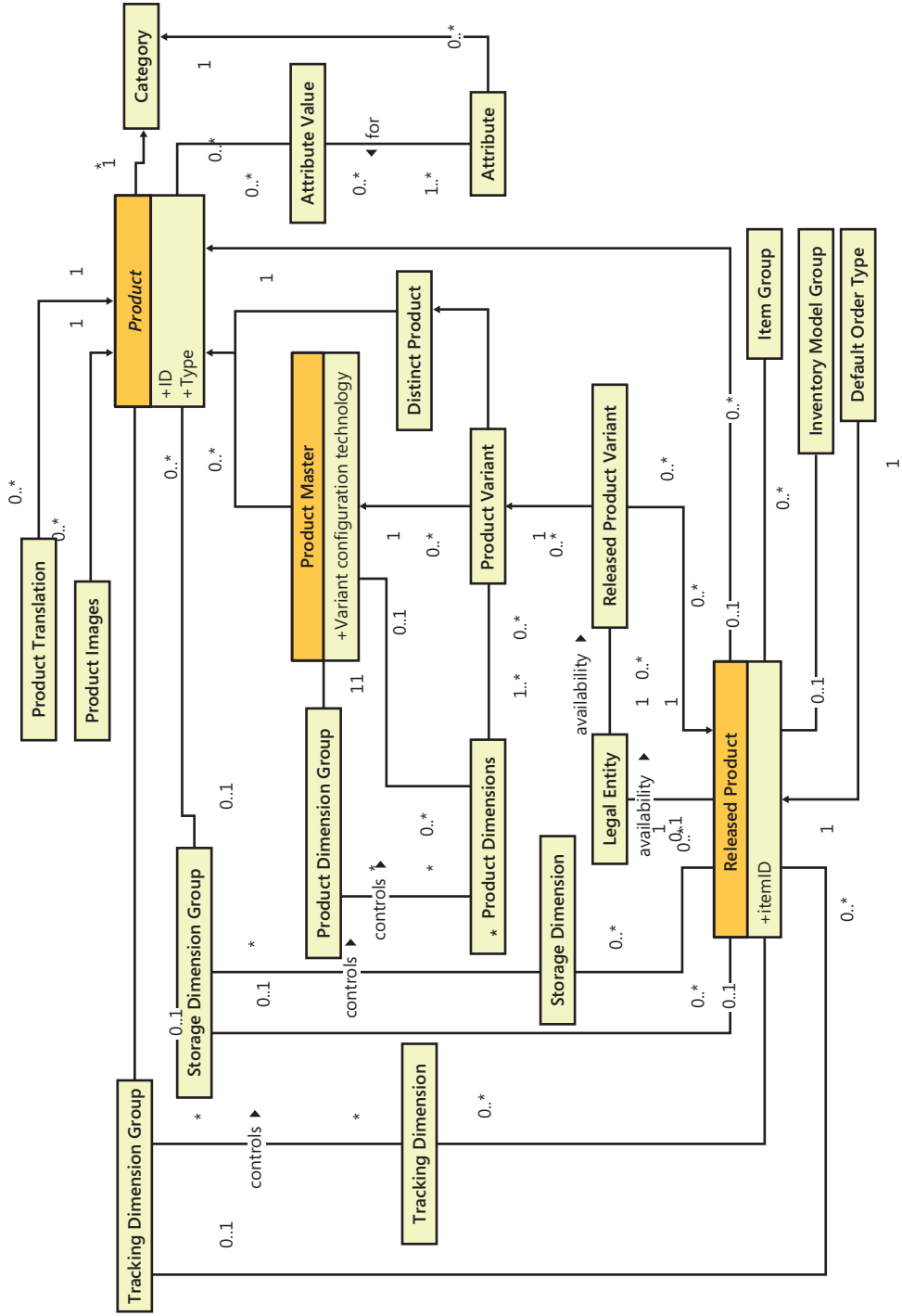


FIGURE 19-2 The product model framework.

Storage and tracking dimensions

Besides *Product Dimensions*, there are two more types of dimensions: *Storage* and *Tracking*. The *Storage Dimension Group* defines the level of detail used to identify the physical location of goods, with the possible dimensions being Site, Warehouse, Location, and Pallet. The *Tracking Dimension Group* defines how specific instances of the same *Released Product* are tracked, with *Batch* and *Serial Number* being available. These dimension groups also define policies regarding how specific dimension values affect business activities. Because these policies may differ in different legal entities, different storage and tracking dimensions can be assigned to the same product in different legal entities.

For example, Contoso might be required by law to track batch numbers of bicycle brakes in certain countries/regions, so that an entire batch can be recalled easily in case a widespread defect is discovered that causes a safety hazard. This requirement does not apply in other countries/regions, so Contoso may choose not to track brake batches in these countries/regions.

Released products

All of the information described so far applies to the concept of a *Product*, representing the information that is shared across the entire organization structure. But this information is not sufficient for a legal entity to be able to use a product. To enable a legal entity to use a product, the product, together with the relevant product variants, has to be released to the legal entity. On a *Released Product*, you can define various policies that control how these products can be used within that legal entity. These policies include storage and tracking dimension groups (unless they were defined on the shared product information), *Inventory Model Group* and *Item Group*, and various inventory and costing policies that apply to the product. Once these policies are defined, the product can be used for transactions within the legal entity.

Product availability within legal entities can be controlled both on the product and the product variant level. For example, a specific bicycle model can be released only to Contoso U.S. because the company wants to sell it only in that market. Another model can be released to Contoso U.S. in only black and red because the company has decided that it's not worth investing in white bikes in the United States.

Additional product information

Beside the required information, additional details can be provided for a product, either to provide a reference or to control the interaction with other system components. For example, you might define *Product Translations* for a product to control the product name or descriptions that are displayed in various contexts. Similarly, you can attach *Images* to provide a graphical representation of the product. An example of the information used to control other components may be the *Default Order Type*, which a legal entity can set to decide the type of order that is generated to cover the demand for a particular product. For example, Contoso Italy, which owns a clothing production facility, could set the *Default Order Type* for cycling clothes to *Production*, while bicycles, which are bought from external vendors, would have the default order type set to *Purchase*.

Product attributes and categories

Microsoft Dynamics AX 2012 introduces the concept of product *Attributes*. User-defined product *Attributes* can be associated with a product *Category* to describe characteristics that are common to all products within that *Category*. Product attributes can be of various data types. Each *Attribute* might have a default value within that *Category*. Once a product is added to a category, the product inherits all product attributes within that category along with their default values. However, the value of a product attribute can be changed for each product.

For example, a company in the food industry has a product attribute called *Fat* that is associated with the procurement category for milk products. The default value is *1%*. The category contains the product *Light-milk*, which inherits the default value of *1%* from the category. The category also contains the product *Fat-milk*, which also inherits the *Fat* value. However, the value for *Fat-milk* has been overwritten with a value of *3%*. A purchasing clerk could search all products that match specific criteria; for example, to find all milk products with a *Fat* value between *1%* and *3%*.

Note Product attributes primarily provide an additional product description that can be used for search functions. You cannot use product attributes to track inventory. The system uses only *Product*, *Tracking,* and *Storage* dimensions to track physical and financial inventory.

Variant configuration technology

The product master definition has a mandatory variant configuration technology, which you use to define the configuration strategy of the *Product Master*. The configuration strategy identifies the method used to create a new *Product Variant* to meet a customer's needs. As a result of product configuration, a new product variant is created in the shared product repository and automatically released to the legal entity when product configuration takes place.

For example, in a configure-to-order environment, a manufacturing company provides a number of predefined product models with different constraints. If the existing models do not meet the expectations of a specific customer, the system creates a new unique product variant to represent the product variation that the customer wants.

Constraint-based configuration technology With the new *Product Configuration* module in Microsoft Dynamics AX 2012, you can describe a shared product model in terms of product structure, product components, attributes, and constraints. After you create a product model, you can associate it with the product master definition, which follows a constraint-based configuration strategy. This advanced configuration technology allows modeling of complex product structures.

For example, a company produces a home theater system that contains 10 components with a number of predefined constraints based on various component characteristics (attributes). During sales order entry, the system exposes a rich product configuration experience that guides the user through the configuration process to create the correct product variant successfully.

Dimension-based configuration technology In Microsoft Dynamics AX 2012, you can set up the configuration group and configuration rules as part of the general Inventory management setup. This information can be used in a bill of material (BOM) definition. You can use the configuration rules to predefine the product configurations to use as part of a specific BOM configuration.

For example, a manufacturing company produces gaming devices in several configurations. To produce these products, the company uses a BOM, which contains a raw component that comes in different configurations. With configuration groups and configuration rules, the system can enforce that only a specific raw component configuration can be included in a specific configuration of a gaming device.

This functionality allows a lightweight approach to configuring products on the order line that have relatively simple BOM structures.

Predefined variant configuration technology Predefining product variants is the simplest configuration strategy in Microsoft Dynamics AX 2012. The different product variants can be created automatically or manually based on the product dimension values, which are associated with the product master.

For example, a product master might have active dimensions of Size and Color. Every time a user adds a new color or size value, the system automatically creates all possible product variants. This functionality is especially valuable in the retail industry, where companies offer a wide range of products based on different styles, colors, and sizes.

When to use the product model framework

You can extend the product model to align the stored product information and product behavior with organizational master data management practices. These practices might include processes such as data governance, centralized master data control, or a product-specific policy. Such a policy can affect the product lifecycle or specific behavior within business processes, such as procurement, production, reserve logistics, and so on.

For example, a multi-country/region retail organization might require a process for defining an organization-wide policy such as product sales price. After the new sales price is set, that price should be used in all retail stores.

Extending the product model framework

In Microsoft Dynamics AX 2012, you can customize the product model by extending tables and classes with the prefix *EcoRes*. For example, you could begin to implement the sales price policy in the previous example by adding a new field to the EcoResProduct table to represent the sales price in the currency assigned as the primary currency in the legal entity's ledger configuration. This field is inherited automatically by all product subtypes such as distinct product, product master, and product variant, which means that you can define a specific sales price for product variations. Once the policy has been implemented, you need to expose the information in the user interface of the Product information management module to allow users to manage sales prices.

The next step is to adjust the product release process to propagate the sales price to released products. You can do this by modifying the *EcoResProductReleaseManager* class, which is responsible for the creation of released products. Specifically, you need to set a proper value in the InventTableModule table, which stores the default sales, purchase, and production prices for the released product within a legal entity.

If the shared product sales price changes, the new sales price value should be propagated to all legal entities in which the product has been released. One of the ways to achieve this is to add such logic to the *update* method of the EcoResProduct table.

The potential conceptual model is illustrated in Figure 19-3.

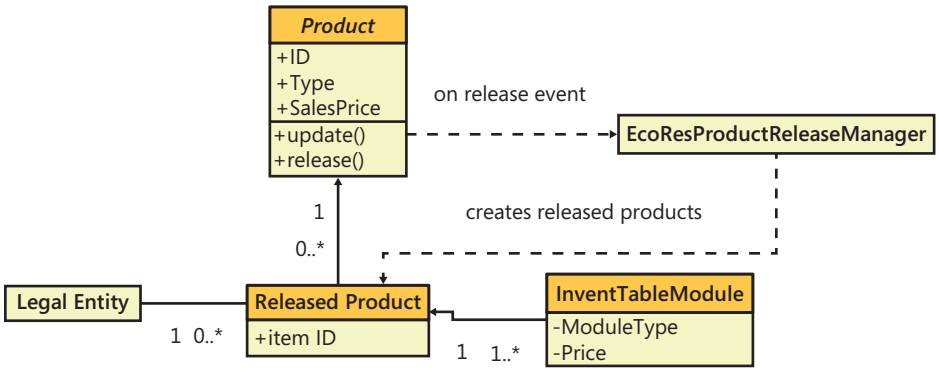


FIGURE 19-3 The customization model.

For more information about the product model framework, download the white paper "Implementing the Item-Product Data Management Framework" from *http://technet.microsoft.com/EN-US/library/hh272877*.

The operations resource framework

Designing a manufacturing process within an enterprise resource planning (ERP) system has traditionally been done by describing what activity should be performed and who should do it. This requires the process engineer to know not only how a product is built, but also which resources are available for building the product. In Microsoft Dynamics AX 2012, a new model has been put in place that allows decoupling of the process from the resources, so that the process can be described without having to reference specific resources.

How the operations resource framework works

The primary entity of the operations resource model is the *Resource*, which is defined as anything that is used for the creation, production, or delivery of an item or service other than the materials that are consumed in the process. There are multiple types of resources: Tool, Machine, Human Resource, Location, and Vendor.

A *Resource* can be a member of a *Resource Group* and the *Resource Group Membership* can change over time. You can think of a *Resource Group* as a vehicle for organizing *Resources*. A *Resource Group* is located at a particular site. A *Resource* can be a member of only a single *Resource Group* at a time. A *Resource* does not have to belong to a *Resource Group*, but the *Resource* is considered for scheduling only during the period or periods that it is connected to a *Resource Group*.

Figure 19-4 shows the conceptual domain model for *Resources* and *Resource Groups*.

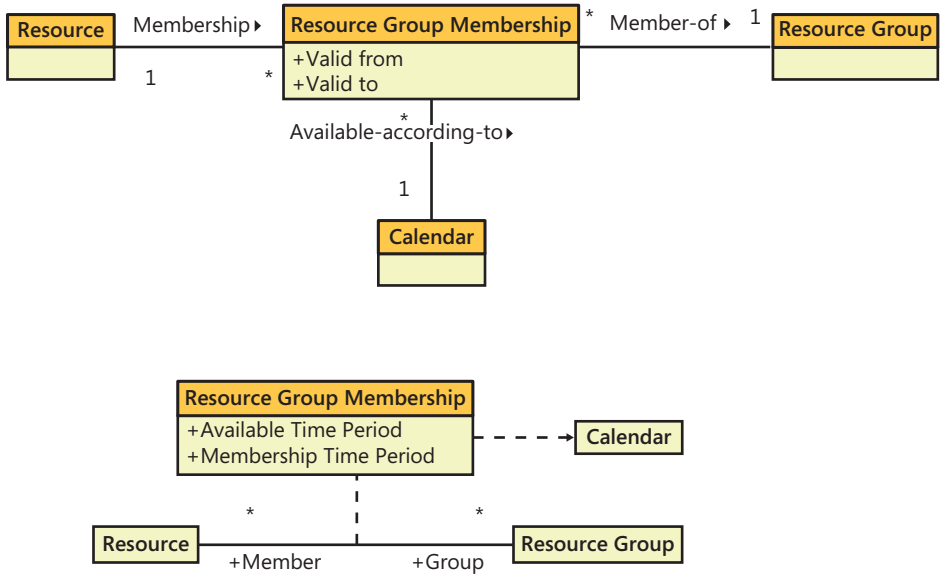


FIGURE 19-4 *Resources* and *Resource Groups*.

Capabilities

A *Capability* is the ability for a resource to perform a given activity, such as welding, pressing, or floor sweeping. A Resource can be assigned one or more *Capabilities* and can have multiple *Capabilities* on the same date. For each assignment, you can set a priority and level at which the *Capability* can be performed; for example, stamping with 4 tons of pressure.

Figure 19-5 shows the conceptual domain model for *Resources* and *Capabilities*.

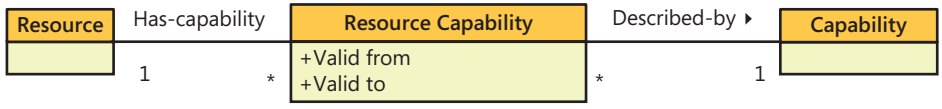


FIGURE 19-5 *Resources* and *Capabilities*.

A *Capability* can be assigned to any *Resource* regardless of its type. If a *Resource* is of the type *Human Resource,* which is associated with a worker, skills, courses, certificates, and title information from the *Human Resources* module, you can use that information in addition to the *Capability* to define the competencies of the resource.

Activities and requirements

An *Activity* is a common abstraction of the unit of work to be performed by one or more *Resources*. The *Activity* entity in itself is not visible to the user but is used internally in Microsoft Dynamics AX for a common representation of the following business entities: *Hour Forecast* (Project), *Production Route*, *Operation Relation*, *Product Model Operation Relation*, and *Product Builder Operation Relation*.

Figure 19-6 shows the conceptual domain model for an *Activity*.

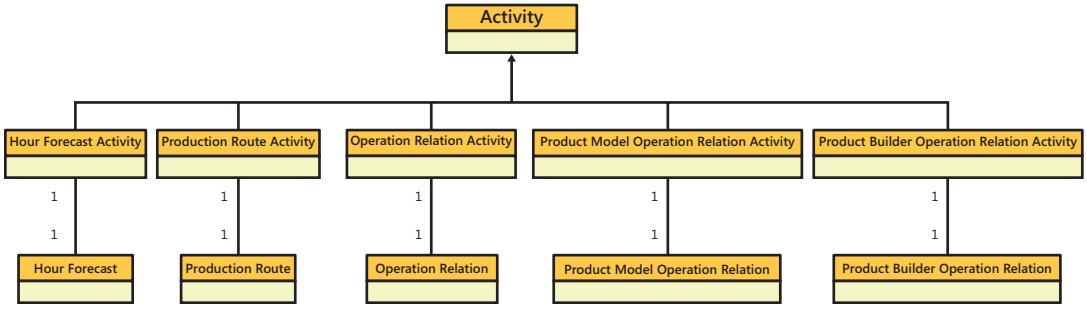


FIGURE 19-6 *Activity* model.

Each *Activity* can have a set of *Activity Requirements* that specifies how many *Resources* are needed for the *Activity* and what abilities the *Resources* must have to participate in the *Activity*. Multiple *Activity Requirements* can be contained in an *Activity Requirement Set*. In order for a *Resource* to be applicable to an *Activity*, the *Resource* must meet all of the requirements in the *Activity Requirement Set*. For each *Activity Requirement*, you can specify whether the *Requirement* should be considered when operations scheduling or job scheduling is performed. Figure 19-7 shows the conceptual domain model for *Activities, Activity Requirements Sets*, and *Activity Requirements*.

Identify applicable resources

Finding the *Resources* that are applicable for an *Activity* requires that at least the following information is known:

- The date as of which to perform the search, because *Resource* and membership information can vary over time an as-of date must be provided.
- The site context, because in most cases the site will be a limiting factor as the resources must be a member of a resource group on the site where the production takes place.
- The scheduling method, because an *Activity Requirement* can be applicable for either operations scheduling, job scheduling, or both.

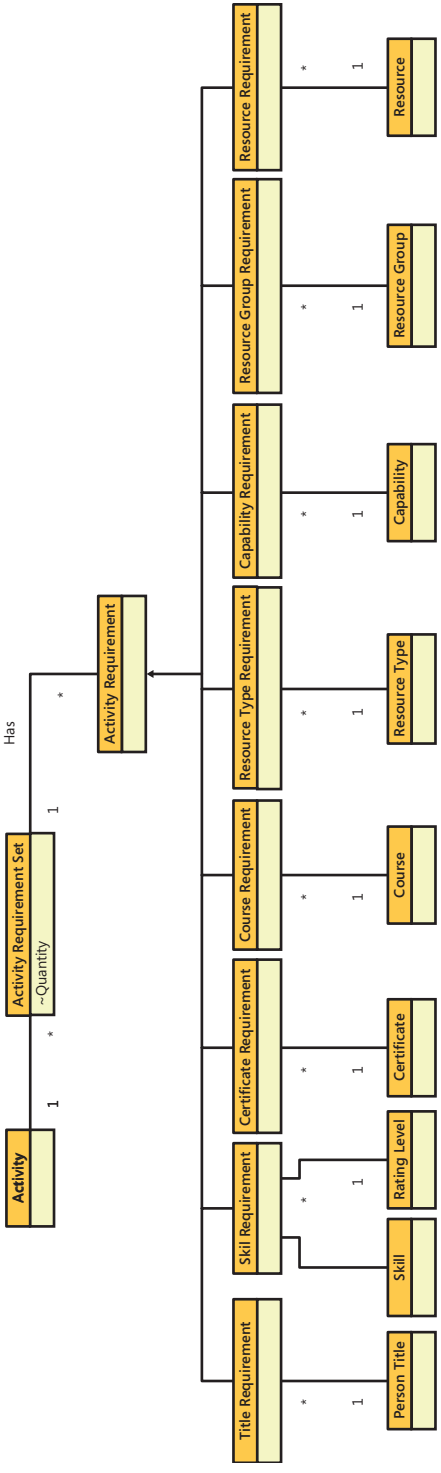


FIGURE 19-7 *Activities, Activity Requirement Sets,* and *Activity Requirements*.

Conceptually, identifying an applicable *Resource* is easy; it is simply a matter of traversing through all *Resources* while checking to determine whether the skills, capabilities, resource type, and so on, of the *Resource* match the ones stated in the requirements and ensuring that the *Resource* is not associated with a *Resource Group* that is marked as a lean work cell.

In code, you can find applicable *Resources* for an *Activity* by using one of the two main application programming interfaces (APIs) offered for the activity requirement set:

- ***applicableResourcesList*** returns the IDs of all applicable resources in a simple list.
- ***applicableResourcesQuery*** creates a query object with the WrkCtrTable table as the primary data source.

When the *Activity* has been planned by the scheduling engine, the chosen *Resource* (or *Resources*) can be found through querying the capacity reservations.

When to use the operations resource framework

You can use the operations resource framework for any activity that requires one or more resources. The framework provides good integration with the scheduling engine, which can perform the resource selection and allocate time according to the requirements and priorities.

Extensions to the operations resource framework

You can extend the operations resource framework in at least two ways: by adding a new class of activity and by adding a new class of activity requirement. The following sections provide details about the integration points and describe some of the considerations that must be taken.

Add a new class of activity

To add a new class of activity, you first need a primary table (called X in the example) that contains the activity definition, including the task to be performed, the duration, links to other activities, and so on. To connect this table to the generic activity, create a new WrkCtrXActivity table with a *0-1* relation to the WrkCtrActivity table and a *1-1* relation to the X table. Having this data structure in place makes it possible to create a form where users can fill in the activity requirements for your X table and navigate further to see the resulting applicable resources. The ProdRoute form is a good example of how such a form can be constructed and the logic that is needed to control the WrkCtrActivityRequirement and related data sources.

If the activity must be scheduled, this can be done by using the same resource scheduling engine that is used for master planning, production orders, and projects. The main class is *WrkCtrScheduler*.

Each type of activity has its own derivative class, such as *WrkCtrScheduler_Proj*, which has information about how that type of activity is handled. At the minimum, the following methods must be implemented:

- ***loadData*** Feeds the engine with information about which activities should be scheduled, the duration for applicable resources, dependencies, and links between activities, and so on.
- ***saveData*** Iterates through the results from the core engine saving the from and to time on the activity and creates capacity reservations in the WrkCtrCapRes table. It is recommended that you add a new value to the *WrkCtrCapRefType* enum and use this value when saving the capacity reservations. Doing so allows for better traceability of who owns the reservation.

Add a new class of activity requirement

If information exists that is related either directly to an operations resource or to the vendor that is associated with the resource, and that information determines the resources' ability to perform an activity, you can incorporate this information into the resource selection process.

If the information related to the resource is stored in a table named *Y*, first create a new value in the *WrkCtrActivityRequirementType* enumeration to represent the *Y* entity. Next add a new table named WrkCtrActivityYRequirement that contains a foreign key to the WrkCtrActivity table and the Y table. Because much of the logic surrounding resource requirements relies on reflection, the new table must implement a certain set of methods. Use the WrkCtrActivityPersonTitleRequirement table as an example of the table methods needed.

After the requirement table is in place, the application must be modified in several places to take the new table into consideration. The best way to ensure that the new requirement is implemented throughout the application is to use cross references for one of the existing tables, such as WrkCtrActivityPersonTitleRequirement, and then add the new table in a similar way.

For performance reasons, the matching of the resource requirements for an activity against the actual abilities of a resource is done by the core scheduling engine by converting capabilities, skills, certificates, and so on to a common property that can be compared against the requirements by simple string matching. This transformation is performed by the *computeResourceCapabilities* and *computeResourceGroupCapabilities* methods of the *WrkCtrSchedulingInteropDataProvider* class, which also must take into account information from the Y table. Consider carefully whether you want the new requirement to be available both for job and operation scheduling. If the requirement must be available for operation scheduling, the used capacity for the group with regards to the Y property must be saved and maintained, along with the capacity reservations, to avoid overbooking. This capability comes at a high performance cost during scheduling.

MorphX model element prefixes for the operations resource framework

All elements that concern the operations resource model are prefixed with WrkCtr*. Most are named very similarly to the conceptual names—except for the Resource entity, which for legacy reasons is stored in the WrkCtrTable table.

For more information about the operations resource model framework, see the following Core Concepts documents on Microsoft Dynamics InformationSource (*http://informationsource.dynamics.com*):

- Allocating resources based on resource requirements
- Operations scheduling based on capabilities

To access these documents, sign in to Microsoft Dynamics InformationSource, click Library, and then type the document title in the Search box.

The dimension framework

The dimension framework provides a method for tracking additional pieces of information like department, cost center, or purpose regarding documents throughout the application. That information can be used in accounting to categorize information.

How the dimension framework works

A *Dimension Attribute* is a type of information that is tracked by the dimension framework. The domain of values for a dimension attribute is defined by the instances of the business entity that exist for the backing business entity type. For instance, the OMOperatingUnit table can provide the list of values for an organization unit dimension. *Dimension Attributes* can be placed in a *Dimension Hierarchy* to indicate ordering. For example, one specialization of a *Dimension Hierarchy* is an *Account Structure*. *Dimension Attributes* can be grouped into a *Dimension Attribute Set*, which is used in some *Setup Data* to specify the *Dimension Attributes* that apply in particular situations; for example, the check box next to each dimension on the LedgerAllocation form.

Figure 19-8 shows the conceptual domain model for the dimension framework.

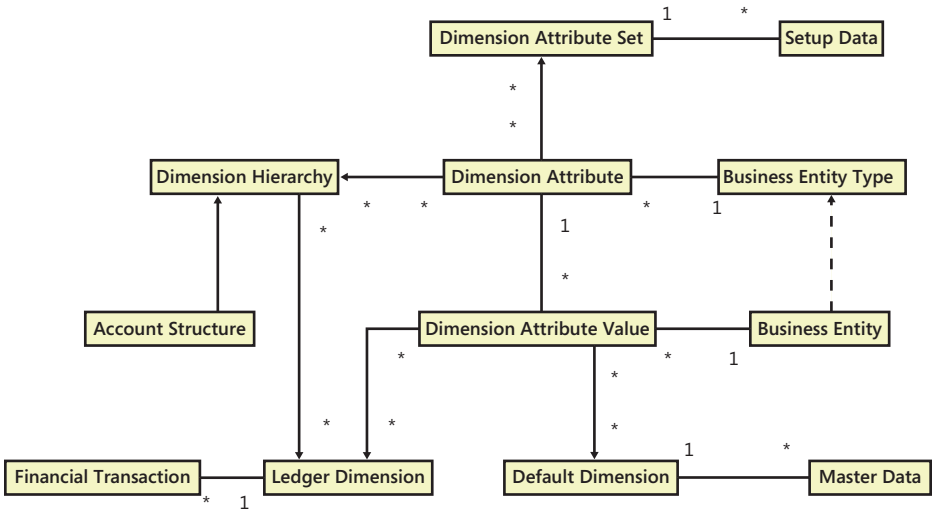


FIGURE 19-8 The dimension framework.

There are four primary storage patterns for exposing and tracking dimension information:

- *Ledger Dimensions* are ordered sets of *Dimension Attribute Values* that are constrained by an account structure and additional accounting rules; for example, *Sales-11005-NorthAmerica-Xbox-70004*. This pattern is normally used on financial data such as journal lines.
- *Default Dimensions* are unordered, unconstrained sets of *Dimension Attribute Values*. For example, a record in the CustTable table may be set to *SalesRegion=NorthAmerica*. This value would then be defaulted into the *Ledger Dimension* when the customer record was used on a sales order. When *Default Dimensions* are shown on a form, all dimensions that are in use by the chart of accounts for the current legal entity are shown.
- *Dimension Attribute Sets* are unordered sets of *Dimension Attributes* that have an enumeration value associated with each *Dimension Attribute*. For example, in the allocation process, the user can mark which *Dimension Attributes* should default from the original transaction and which should take on a specific value. This pattern is used infrequently.
- *Dimension Sets* are ordered sets of dimension values similar to *Ledger Dimensions*, but without the requirement that they contain a main account. They are used primarily for reporting and balance tracking. For example, in order to view the trial balance list page by main account and department, a dimension set containing those two dimensions would be created.

These four primary patterns are further specialized into dozens of specific uses. A few of the more common specializations are as follows:

- The *Default Account* pattern is a specialization of the *Ledger Dimension* storage pattern. Instances of the *Default Account* pattern are stored like a standard ledger account but only

contain a value for the main account *Dimension Attribute*. This pattern is used in areas such as posting profiles to specify which main account is used when a *Ledger Dimension* is created by financial processes.

- The *Dynamic Account* pattern is a specialization of the *Ledger Dimension* storage pattern. This pattern is used on journals where an *Account Type* field is available. When the *Account Type* is set to *Ledger*, then it behaves like a standard *Ledger Dimension* account. When the account type is set to something else, like *Customer*, it acts as a customer lookup. When a non-ledger type is used, a predefined hidden *Dimension Attribute* is used to signify customer, vendor, item, or whatever type is used.

In addition, a variety of budgeting patterns mirror the accounting patterns.

Constrain combinations of values

You can constrain the combinations of values that are valid in *Ledger Dimensions* in two ways.

If constraints are set up in the tree in the Configure Account Structure form (General Ledger > Setup > Financial Dimensions > Configure Account Structures), these constraints are stored in the DimensionConstraintNode and DimensionConstraintNodeCriteria tables. Because the structure of the data in these tables is highly complex, it is much easier to use the *DimensionValidation::validateByTree* method to perform validation rather than reading the constraint node tables directly. The *validateByTree* method validates that a *Ledger Dimension* matches the constraints specified in these tables.

The other method of constraining values is to click the Relationships button on the Action Pane of the Configure Account Structures form, and then use the Select Relationships form to specify the relationships that you want to apply to the account structure. The Select Relationships form shows all of the organization model hierarchies that contain organization model types used as the backing entities for *Dimension Attributes* in the current hierarchy. For example, if an account structure contains departments and cost centers, and an organization model exists that relates departments to cost centers, then that information appears in this form. The information will appear twice with party A and party B reversed. This allows a system administrator to specify whether departments must be parents or children of a given cost center to be valid. These organization model constraints are similarly applied when you use the *DimensionValidation* methods.

Create values

You can create *Ledger Dimensions* programmatically in two ways. To explicitly create them, use the *DimensionStorage* class. You can use this class to add multiple hierarchies and values. When you call the *save* method, it attempts to find an existing combination. If no combination is found, a new one

is created. *Ledger Dimensions* are immutable and only one exists for any given combination. So if the same account is used twice, this method guarantees that only one instance is created in the database.

When working with existing default accounts and ledger dimensions, you can use the *DimensionDefaultingService* class to combine the values into new combinations. For example, the *DimensionDefaultingService*::*serviceCreateLedgerDimension* method takes a default account and one or more Default Dimensions and combines them to form a full *Ledger Dimension*.

Extend the dimension framework

The most common customization of the dimension framework is to add a new backing entity type. Microsoft Dynamics AX 2012 includes approximately 30 backing entities. To add a backing entity type, the only requirement is that the entity must have a natural key that consists of a unique, single-part string that is 30 characters or less.

To add a new backing entity type, create a view to wrap the entity that meets the following criteria:

- The view name must be DimAttribute*[entityname]*. For example, DimAttributeCustTable.
- The view must contain a root data source named BackingEntity, which is backed by the table containing the surrogate key and the natural key.
- The view can optionally contain additional related data sources to handle inheritance or relational associations to provide additional fields, such as a name from the table DirPartyTable.
- The view must contain the following fields named exactly as follows:
  - **Key** Must be the surrogate key field of the backing entity; for example, an *Int64 RecId* field.
  - **Value** Must be the natural key field of the backing entity; for example, a *str30 AccountNum* field.
  - **Name** Must point to the additional description for the entity; for example, a *str60* description field.

If the view meets these criteria, the entity will automatically become available as a backing entity type:

> Because the list of backing entity types are cached both on the client and server, a new type does not appear in the list of existing entities until a call to clear the caches is performed, or until both the client and server are restarted. To clear the caches and have the new entity type appear immediately, use the options on the Tools > Caches menu in the Development Workspace.

Query data

Dimension Attributes are data and can be added or removed by the user. This means that specific dimensions should not be referenced directly in code because there is no guarantee that a given dimension exists. Instead, treat dimension references as configurable data. The one exception to this rule is the *Main Account Dimension Attribute*. All installations are guaranteed to have exactly one *Dimension Attribute* that is backed by *Main Account*. To retrieve this *Dimension Attribute*, use the *DimensionAttribute::getMainAccountDimensionAttribute* method.

Querying dimension information depends on the pattern being used. In the case of a *Ledger Dimension*, either the full combination can be used or the constituent parts. To get the full concatenated combination, create a join to the DimensionAttributeValueCombination table, as shown in the following example:

```
GeneralJournalAccountEntry          gjae;
DimensionAttributeValueCombination  davc;

select gjae join DisplayValue from davc where
    davc.RecId == gjae.LedgerDimension;
```

To get a constituent part of the Ledger Dimension, you can use the DimensionAttributeLevelValueView abstraction to abstract some of the complexity of the dimension model:

```
GeneralJournalAccountEntry          gjae;
DimensionAttributeLevelValueView    dalvv;
DimensionAttribute                  department;

department = DimensionAttribute::findByName('Department');

select gjae join DisplayValue from dalvv where
    dalvv.ValueCombinationRecId == gjae.LedgerDimension &&
    dalvv.DimensionAttribute == department.RecId;
```

The main account *Dimension Attribute* is a special case. This *Dimension Attribute* has been denormalized to the DimensionAttributeValueCombination table to optimize the performance of queries for this value, because it is the most often used:

```
GeneralJournalAccountEntry          gjae;
DimensionAttributeValueCombination  davc;
MainAccount                         mainAccount;

select gjae
    join MainAccount from davc where
        davc.RecId == gjae.LedgerDimension
    join Name from mainAccount where
        mainAccount.RecId == davc.MainAccount;
```

You query *Default Dimensions* in a similar way to *Ledger Dimensions*; however, *Default Dimensions* do not have a concatenated representation because they are unordered sets.

The DimensionAttributeValueSetItemView abstraction joins the DimensionAttributeValueSetItem and DimensionAttributeValue tables to simplify queries:

```
CustTable                           custTable;
DimensionAttributeValueSetItemView  davsiv;
DimensionAttribute                  department;

department = DimensionAttribute::findByName('Department');

select custTable
    join DisplayValue from davsiv where
        davsiv.DimensionAttributeValueSet == custTable.DefaultDimension &&
        davsiv.DimensionAttribute == department.RecId;
```

Physical table references

Table 19-2 maps the concept names in the conceptual domain model to the names of physical table elements that realize these concepts in the application where the names are not the same.

TABLE 19-2 Mapping between concepts and physical tables.

Concept name	Physical tables
Ledger Dimension	DimensionAttributeValueCombination, DimensionAttributeValueGroupCombination, DimensionAttributeValueGroup, DimensionAttributeLevelValue
Default Dimension	DimensionAttributeValueSet, DimensionAttributeValueSetItem
Dimension Attribute Set	DimensionAttributeSet

For more information about the dimension framework, download the following white papers:

- "Securing Data by Dimension Value by Using Extensible Data Security (XDS)" at *http://www.microsoft.com/download/en/details.aspx?id=26921*.
- "Implementing the Account and Financial Dimensions Framework" at *http://technet.microsoft.com/en-us/library/hh272858.aspx*.

The accounting framework

The accounting framework uses policies and rules to derive accounting requirements for amounts and business events that are documented on source document lines. These policies and rules are abstracted as five categories:

- ***Accounting Policy*** Used to determine if accounting applies for a business event – monetary amount combination.

- ***Main Account Derivation Rule*** Used to determine main account values.
- ***Main Account Dimension List Provider*** Used to provide a list of main accounts and side (debit or credit) combinations.
- ***Dimension Derivation Rule*** Used to determine dimension values.
- ***Accounting Journalization Rule*** Used to determine which main account dimension list provider should be used and to determine journalization parameters, such as the posting type, that should be used.

The accounting framework is also responsible for transferring *Subledger Journal* entries to the *General Journal*. Rules for *Subledger Journal* transfers are specified by legal entity and source document type, and they determine when the *Subledger Journal* is transferred to the *General Journal* and whether summarization occurs on transfer.

How the accounting framework works

The *Accounting Distribution* process creates at least one *Accounting Event*. An *Accounting Event* groups a set of distributions based on their accounting date. When a *Source Document* header is submitted to a *Processor* for processing and the *Processor* transitions the document from an *In Process* state to a *Completed* state, the *Journalization Processor* (journalizer) is called. The journalizer processes all *Accounting Events* associated with the document that are in a *Started Process State*, and transitions them to a *Journalized Process State*. An *Accounting Policy* determines whether accounting is required for amounts and business events that are documented on a *Source Document Line*. If the *Accounting Policy* specifies that accounting is required, the journalizer uses *Journalization Rules*, *Main Account Derivation Rules*, *Dimension Derivation Rules*, and the *Main Account Dimension List Provider* to determine the main account–dimension combinations to use when creating balanced subledger journal entries.

Figure 19-9 shows the conceptual domain model for the accounting framework.

Subledger Journal Transfer Rules, shown in Figure 19-10, specify when the transfer should occur (synchronous, asynchronous, or scheduled batch) and whether amounts for the same main account – dimension combination should be summarized when transferred to the general journal.

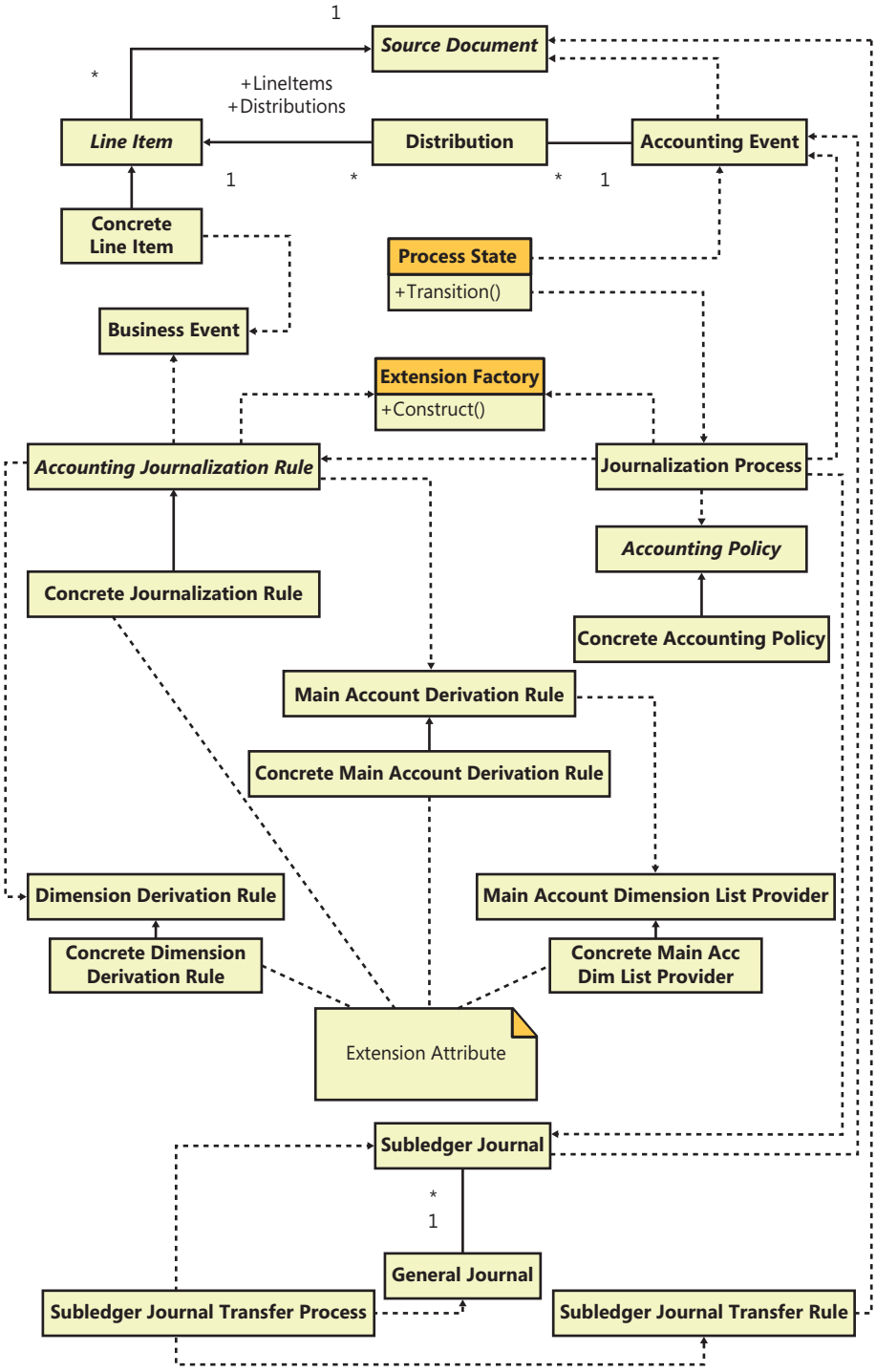


FIGURE 19-9 The accounting framework.

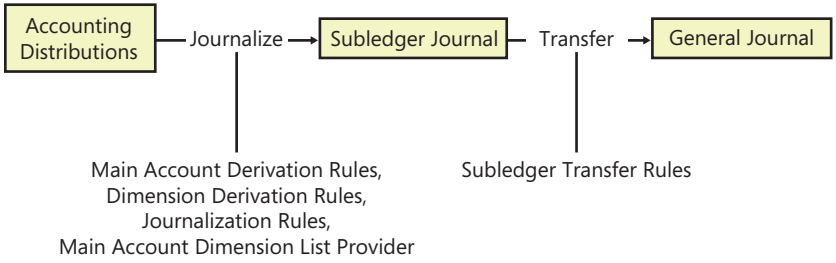


FIGURE 19-10 Rule application in the accounting process.

When to use the accounting framework

Extend the accounting framework to create concrete implementations of accounting policy, journalization, main account derivation, main account dimension list providers, and dimension derivation rules to support new source document implementations. In Microsoft Dynamics AX 2012, the accounting framework has been extended to create concrete accounting policies, journalization, main account derivation, and dimension derivation rules used to generate subledger journal entries on the purchase requisition, purchase order, product receipt, vendor invoice, travel requisition, expense report, and free-text invoice source documents.

Extensions to the accounting framework

The Microsoft Dynamics AX 2012 purchase requisition (PurReqSourceDocument prefix) is an example of an extension of the source document framework components. The AccPolicyCommitFunds-ExpensedProd accounting policy and AccJourRuleCommitFundsForExpProdExtPrice dimension derivation rule are extensions to the accounting framework that specify the accounting requirements for the purchase requisition document. These are examples of extensions to the accounting framework.

Accounting framework process states

The *Process States* for the accounting process are illustrated in Figure 19-11.

Each processing state performs an action and updates the status of the accounting event that is being processed. Table 19-3 describes the process states.

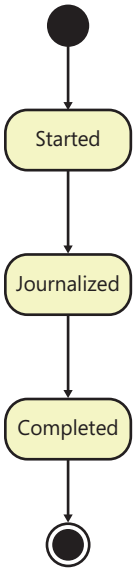


FIGURE 19-11 State model for the accounting process.

TABLE 19-3 Process states for the accounting framework.

State	Process	Description
Started	Accounting distribution process	The accounting event is created and is awaiting journalization.
Journalized	Subledger journalizing process	Subledger journal entries have been created, recording the accounting impact of the distributions that are associated with the accounting event. The subledger journal entries have not been transferred to the general journal.
Completed	Subledger journal transfer process	The subledger journal entries that are associated with the accounting event have been transferred to the general journal.

MorphX model element prefixes for the accounting framework

Table 19-4 maps the concept names in the conceptual domain model to the prefixes added to the names of MorphX model elements that realize these concepts in the application.

TABLE 19-4 Mapping between accounting framework concepts and prefixes of MorphX model elements.

Concept	MorphX model element prefix
Accounting Policy	*AccPolicy*
Subledger Journal Transfer Process	*SubledgerJournalTransfer*
Accounting Event	*AccountingEvent*

Concept	MorphX model element prefix
Accounting Journalization Rule	*AccJourRule*
Dimension Derivation Rule	*DimensionDerivationRule*
Main Account Derivation Rule	*MainAccountDerivationRule*
Main Account Dimension List Provider	*MainAccountDimensionListProvider*
Journalization Process	*SubledgerJournalization* *SubledgerJournalizer*
Subledger Journal	*SubledgerJournal*
General Journal	*GeneralJournal*
Subledger Journal Transfer Rule	*SubledgerJournalTransferRule*

The source document framework

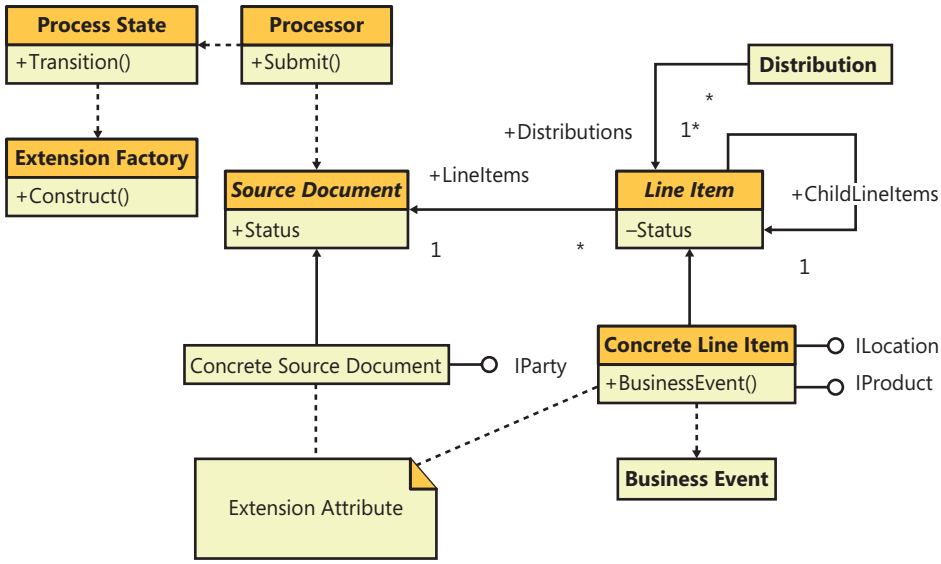
A *Source Document* is an original record that documents the occurrence of one or more *Business Events* in an accounting system. *Concrete Source Documents,* such as purchase orders, product receipts, and vendor invoices, are entered into an accounting system that records, classifies, tracks, and reports on the quantity and value of economic resources that are exchanged or committed for exchange when performing activities identified by *Business Events* such as purchase, product receipt, and payment request.

How the source document framework works

The source document framework generates a projection of a concrete source document for a process that transitions the source document status to reflect the state of the process. Figure 19-12 shows the domain model for the source document framework.

Microsoft Dynamics AX submits a *Source Document* header or line record to a *Processor* for processing when a user confirms that the documentation requirements of business events and internal process controls have been met. A *Processor* is a state machine that transitions the processing of the source document and its lines from one *Process State* to another. The *Processor* creates a *Process State* object that corresponds to the status of the source document or the status of the source document line item and then commands the *Process State* to transition the process to the next state.

A *Process State* first constructs a *Concrete Source Document* or a *Concrete Line Item* from the provided *Source Document* header or line record by using an extension factory facility. The extension factory facility uses the source document type and the table number of the *Concrete Source Document* or the *Concrete Line Item* provided by the header or line record to find a matching concrete source document class. A matching source document class is one that is annotated with a class attribute recognized as an *Extension Attribute* by the *Extension Factory* and that also specifies a matching source document type and table number as arguments.



Concept	Model Element Prefix
Source Document	SourceDocument
Line Item	SourceDocumentLineItem
Processor	SourceDocumentProcessor
Extension Factory	SysExtension
Process State	SourceDocumentState
	SourceDocumentLineState
Distribution	AccountingDistribution
Business Event	BusinessEvent

FIGURE 19-12 The source document domain model.

A *Process State* accesses a data projection of the *Concrete Source Document* and *Concrete Source Document Line*, performs an action, transitions the process to a new state, and updates the status of the process's *Concrete Source Document* or *Concrete Source Document Line* accordingly. The data projections of the *Concrete Source Document* and *Concrete Source Document Line* are defined by one or more interfaces. For example, implementing the *IParty* interface provides a party account number, and implementing the *IProduct* interface provides an item number and a production category to an accessing *Process State*.

When to use the source document framework

Extend the source document framework to implement concrete source documents that document business events whose financial consequences are recorded in the subledger journal. In Microsoft Dynamics AX 2012, the source document framework has been extended to implement the source documents purchase requisition, purchase order, product receipt, vendor invoice, travel requisition, expense report, and free-text invoice.

Extensions to the source document framework

The Microsoft Dynamics AX 2012 free-text invoice (CustInvoiceSourceDocument prefix) is the simplest extension of the source document framework components. Readers new to the source document framework should review this extension of the source document framework first. The concrete source document and concrete line item implement only those source document projection interfaces that are required by the accounting distribution processor and the subledger journalizing processor.

The process states for the subledger journalizing process and the accounting distribution process are illustrated in Figure 19-13. Each processing state performs an action and updates the status of the source document or source document line item that participated in the process. Table 19-5 describes the states of the subledger journalizing process and the accounting distribution process.

TABLE 19-5 States of the subledger journalizing and accounting distribution processes.

State	Process	Description
In Process	Subledger journalizing process	The state reached when a source document is first created and when it is changed. The subledger journalizing process transitions to the completed state when the original source document or a changed source document is confirmed.
Completed	Subledger journalizing process	The state reached when a source document is confirmed and the documented consequences of business events are journalized. The subledger journalizing process transitions to the In Process state when a source document is changed or when a source document is finalized and can no longer be changed.
Finalized	Subledger journalizing process	The state reached when a source document can no longer be changed. The subledger journalizing process balances any open account entries when finalizing the source document.
Draft	Accounting distribution process	The state reached when a source document line is first created or when all accounting distributions that reference a source document line are deleted.
Fully Distributed	Accounting distribution process	The state reached when an accounting distribution is first added to distribute an amount that is documented on a source document line; for example, a discount. This state is also reached when an accounting distribution is generated or derived from amounts documented on a source document line; for example, an extended price. This state is also reached when the sum of the distribution amounts equals the distributed amount, or when a source document is changed.
Partly Distributed	Accounting distribution process	The state reached when the sum of the distribution amounts does not equal the distributed amount.
Cancelled	Subledger journalizing process Accounting distribution process	The state reached when a source document or source document line item is cancelled. All accounting distributions are reversed and the consequences of the cancelled business event are journalized before this state is reached.

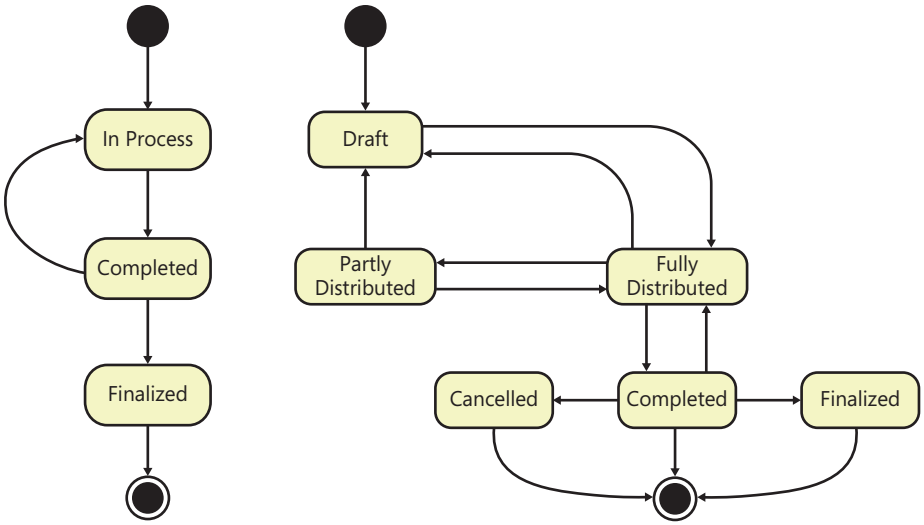


FIGURE 19-13 State model for the documentation process, the accounting distribution process, and the subledger journalizing process.

MorphX model element prefixes for the source document framework

Table 19-6 maps the concept names in the conceptual domain model for the source document framework to the prefixes added to the names of MorphX model elements that realize these concepts in the application.

TABLE 19-6 Mapping between source document framework concepts and prefixes for MorphX model elements.

Concept	MorphX model element prefix
Source Document	*SourceDocument*
Line Item	*SourceDocumentLineItem*
Processor	*SourceDocumentProcessor*
Process State	*SourceDocumentState* *SourceDocumentLineState*
Business Event	*BusinessEvent*
Distribution	*AccountingDistribution*
Extension Factory	*SysExtension*

CHAPTER 20

Reflection

In this chapter

Introduction

Reflection is a programmatic discoverability mechanism of a type system. By using the reflection application programming interfaces (APIs) of the Microsoft Dynamics AX application model, you can read and traverse element definitions as though they were in a table, an object model, or a tree structure.

You can perform interesting analyses with the information that you get through reflection. The Reverse Engineering tool provides an excellent example of the power of reflection. Based on element definitions in MorphX, the tool generates Unified Modeling Language (UML) models and entity relationship diagrams (ERDs) that you can browse in Microsoft Visio.

You can also use reflection to invoke methods on objects. This capability is of little value to business application developers. But for framework developers, the power to invoke methods on objects can be valuable. Suppose you want to programmatically write any record to an XML file that includes all of the fields and display methods. With reflection, you can determine the fields and their values and also invoke the display methods to capture their return values.

X++ features a set of system functions that you can use for reflection, in addition to three reflection APIs. The reflection system functions are as follows:

- **Intrinsic functions** A set of functions that you can use to refer to an element's name or ID safely
- ***typeOf* system function** A function that returns the primitive type for a variable
- ***classIdGet* system function** A function that returns the ID of the class for an instance of an object

The reflection APIs are as follows:

- **Table data** A set of tables that contains all element definitions. The tables provide direct access to the contents of the model store files. You can query for the existence of elements

and certain properties, such as model, *created by*, and *created datetime*. However, you can't retrieve information about the contents or structure of each element.

- **Dictionary** A set of classes that provide a type-safe mechanism for reading metadata from an object model. Dictionary classes provide basic and more abstract information about elements in a type-safe manner. With few exceptions, this API is read-only.
- **Treenodes** A class hierarchy that provides the Application Object Tree (AOT) with an API that can be used to create, read, update, and delete any piece of metadata or source code. This API can provide all information about anything in the AOT. You navigate the treenodes in the AOT through the API and query for metadata in a non-type-safe manner.

This chapter delves into the details of these system functions and APIs.

Reflection system functions

The X++ language features a set of system functions that can be used to reflect on elements. They are described in the following sections.

Intrinsic functions

Use intrinsic functions whenever you need to reference an element from within X++ code. Intrinsic functions provide a way to make a type-safe reference. The compiler recognizes the reference and verifies that the element being referenced exists. If the element doesn't exist, the code doesn't compile. Because elements have their own lifecycles, a reference doesn't remain valid forever; an element can be renamed or deleted. Using intrinsic functions ensures that you are notified of any broken references at compile time. A compiler error early in the development cycle is always better than a run-time error later.

All references you make using intrinsic functions are captured by the Cross-reference tool. You can determine where any element is referenced, regardless of whether the reference is in metadata or code. The Cross-reference tool is described in Chapter 2, "The MorphX development environment and tools."

Consider these two implementations:

```
print "MyClass";          //Prints MyClass
print classStr(MyClass);  //Prints MyClass
```

Both lines of code have the same result: the string *MyClass* is printed. As a reference, the first implementation is weak. It will eventually break if the class is renamed or deleted, meaning that you'll need to spend time debugging. The second implementation is strong and unlikely to break. If you were to rename or delete *MyClass*, you could use the Cross-reference tool to analyze the impact of your changes and correct any broken references.

By using the intrinsic functions *<Concept>Str*, you can reference all elements in the AOT by their names. You can also use the intrinsic function *<Concept>Num* to reference elements that have an ID. Intrinsic functions are not limited to root elements; they also exist for class methods, table fields, indexes, and methods. More than 50 intrinsic functions are available. Here are a few examples:

```
print fieldNum(MyTable, MyField);   //Prints 60001
print fieldStr(MyTable, MyField);   //Prints MyField
print methodStr(MyClass, MyMethod); //Prints MyMethod
print formStr(MyForm);              //Prints MyForm
```

The ID of an element is assigned when the element is created in the model store. In the preceding example, the ID *60001* is assigned to the first element field created in a table. (Element IDs are explained in Chapter 21, "Application models.")

Two other intrinsic functions are worth noting: *identifierStr* and *literalStr*. The *identifierStr* function allows you to refer to elements if a more feature-rich intrinsic function isn't available. The *identifierStr* function provides no compile-time checking and no cross-reference information. However, using the *identifierStr* function is still better than using a literal because the intention of referring to an element is captured. If a literal is used, the intention is lost—the reference might be to user interface text, a file name, or something completely different. The Best Practices tool detects the use of *identifierStr* and issues a best practice warning.

The Microsoft Dynamics AX runtime automatically converts any reference to a label ID to its corresponding label text. In most cases, this behavior is what you want; however, you can prevent the conversion by using *literalStr*. The *literalStr* function allows you to refer to a label ID without converting the label ID to the label text, as shown in this example:

```
print "@SYS1";             //Prints Time transactions
print literalStr("@SYS1"); //Prints @SYS1
```

In the first line of the example, the label ID (*@SYS1*) is automatically converted to the label text (*Time transactions*). In the second line, the reference to the label ID isn't converted.

typeOf system function

The *typeOf* system function takes a variable instance as a parameter and returns the base type of the parameter. Here is an example:

```
int i = 123;
str s = "Hello world";
MyClass c;
guid g = newGuid();

print typeOf(i);  //Prints Integer
```

```
print typeOf(s);  //Prints String
print typeOf(c);  //Prints Class
print typeOf(g);  //Prints Guid
pause;
```

The return value is an instance of the *Types* system enumeration. It contains an enumeration for each base type in X++.

classIdGet system function

The *classIdGet* system function takes an object as a parameter and returns the class ID for the class element of which the object is an instance. If the parameter passed is null, the function returns the class ID for the declared type, as shown in this example:

```
MyBaseClass c;
print classIdGet(c);  //Prints the ID of MyBaseClass

c = new MyDerivedClass();
print classIdGet(c);  //Prints the ID of MyDerivedClass
pause;
```

This function is particularly useful for determining the type of an object instance. Suppose you need to determine whether a class instance is of a particular class. The following example shows how you can use *classIdGet* to determine the class ID of the *_anyClass* variable instance. If the *_anyClass* variable really is an instance of *MyClass*, it's safe to assign it to the *myClass* variable.

```
void myMethod(object _anyClass)
{
    MyClass myClass;
    if (classIdGet(_anyClass) == classNum(MyClass))
    {
        myClass = _anyClass;
        ...
    }
}
```

Notice the use of the *classNum* intrinsic function, which evaluates the parameter at compile time, and the use of *classIdGet*, which evaluates the parameter at run time.

Because inheritance isn't taken into account, this sort of implementation is likely to break the object model. In most cases, any instance of a derived *MyClass* class should be treated as an actual

MyClass instance. The simplest way to handle inheritance is to use the *is* and *as* operators. For more information, see Chapter 4, "The X++ programming language."

Note This book promotes customization through inheritance by using the Liskov substitution principle.

Reflection APIs

The X++ system library includes three APIs that can be used to reflect on elements. They are described in the following sections.

Table data API

Suppose that you want to find all classes whose names begin with *Invent*. The following example shows one way to conduct your search:

```
static void findInventoryClasses(Args _args)
{
    SysModelElement modelElement;

    while select name from modelElement
        where modelElement.ElementType == UtilElementType::Class
           && modelElement.Name like 'Invent*'
    {
        info(modelElement.Name);
    }
}
```

The SysModelElement table provides access to all elements. The *ElementType* field holds the concept to search for. The data model for the model store contains nine tables, which are shown and explained in Figure 20-1.

Note The UtilElements table is still available for backward compatibility. It is implemented as an aggregated view on top of the SysModel tables. For performance reasons, you should limit usage of this compatibility feature and eventually rewrite your code to use the new API.

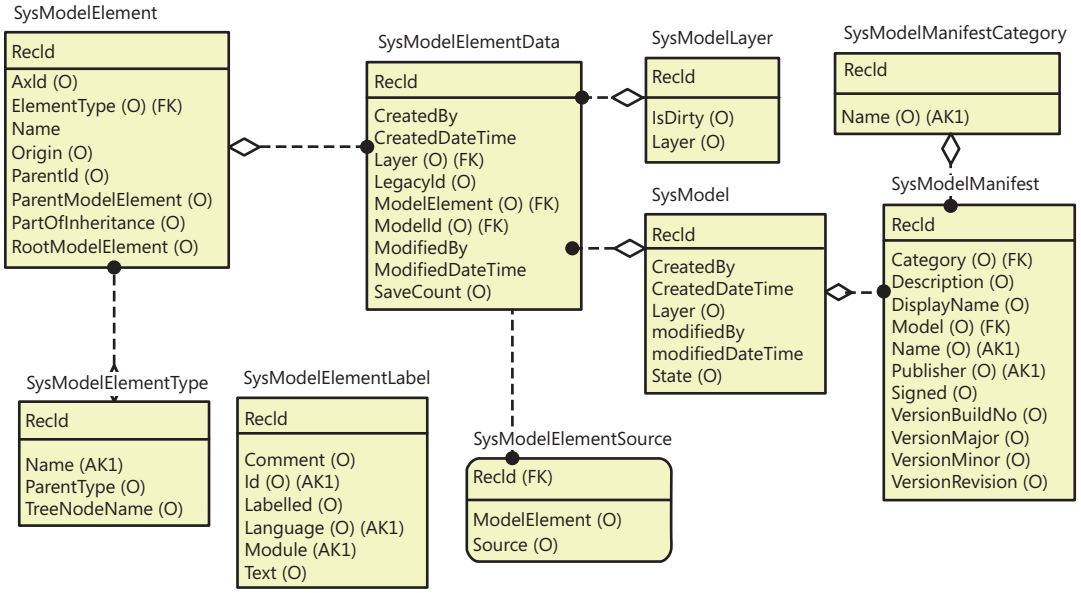


FIGURE 20-1 The data model for the model store.

Because of the nature of the table data API, the SysModel tables can also be used as data sources in a form or a report. A form showing the table data is available from Tools > Model Management > Model elements. In the form, you can use standard query capabilities to filter and search the data.

The SysModelElement table contains all of the elements in the model store; it is related to the SysModelElementData table, which contains the various definitions of each element. For each SysModelElement record, there is at least 1 SysModelElementData record—and perhaps as many as 16 if the element is customized across all 16 layers. In other words, the element defines the customization granularity. You cannot customize less than an element. For example, even if you change just one property on an element, a new record is inserted into the SysModelElementData table that includes all properties of the element.

Note System elements, as listed under *System Documentation* node in the AOT, are not present in these tables.

Elements are structured in hierarchies. The root of a hierarchy is the root element; for example, a form. The form contains data source, control, and method elements. The hierarchy can encompass multiple levels; for example, a form control can have methods. The root element and parent element are exposed in the *RootModelElement* and *ParentModelElement* fields of the SysModelElement table. The job on the next page finds all elements under the *CustTable* form element and lists the name and type of each element, the name of the parent element, and the AOT path of the associated *TreeNode* class.

```
static void findElementsOnCustTable(Args _args)
{
    SysModelElement modelElement;
    SysModelElement rootModelElement;
    SysModelElement parentModelElement;
    SysModelElementType modelElementType;

    while select name from modelElement
        join Name from modelElementType
            where modelElementType.RecId == modelElement.ElementType
        join name from parentModelElement
            where parentModelElement.RecId == modelElement.ParentModelElement
        exists join rootModelElement
            where rootModelElement.RecId == modelElement.RootModelElement
               && rootModelElement.Name == formStr(CustTable)
               && rootModelElement.ElementType == UtilElementType::Form
    {
        info(strFmt("%1, %2, %3, %4",
            parentModelElement.Name, modelElementType.Name, modelElement.Name,
            SysTreeNode::modelElement2Path(modelElement)));
    }
}
```

Notice the use of the *ElementType* field in the two preceding examples. If the element type is a *UtilElement*, you will find a matching entry in the *UtilElementType* enum; alternatively, you can always join to the SysModelElementType table, which contains information about all element types. All root elements and a few former subelements are *Utilelements*. You can access them through the legacy UtilElements table. Data models with higher fidelity were introduced in this release to support more granular customizations, which among other things facilitate easier upgrade and simpler side-by-side installation of models. For more information, see Chapter 21.

Table 20-1 lists the reflection tables and views. See Figure 20-1 to learn how these tables relate to each other.

TABLE 20-1 Reflection tables and views.

Table or View Name	Description
SysModel	Table containing the models in the model store.
SysModelElement	Table containing the elements in the model store. There is exactly one record for each element in the AOT, regardless of customizations.
SysModelElementData	Table containing the element definitions in the model store. There is one record for each element in each layer.
SysModelElementLabel	Table containing all label text and comments.
SysModelElementSource	Table containing all X++ source code.
SysModelElementType	Table containing definitions of element types. The information in this table is static and is populated at installation time.

Table or View Name	Description
SysModelLayer	Table containing the layers. The information in this table is static and is populated at installation time. There is a record for each of the 16 layers.
SysModelManifest	Table containing the manifest for the models, such as name, publisher, and version number. There is one record for each model.
SysModelManifestCategory	Table containing the categories that a model can belong to. Each model belongs to a category: Standard, Hotfix, Virtual, or Temporary.
UtilElements, UtilIdElements	Aggregated views on top of the SysModel tables. These views are provided for backward compatibility.
UtilModels	View on top of SysModel, SysModelManifest, and SysModelLayer, which make querying models easier.

Note Alternative versions of the tables in Table 20-1 exist. If you postfix the table name with the word *old*, you can access the baseline model store instead of the primary model store. For example, the SysModelElementOld table contains the model elements in the baseline model store. The baseline model store is primarily used in upgrade scenarios.

You can use the *Microsoft.Dynamics.AX.Framework.Tools.ModelManagement* namespace provided by the *AxUtilLib.dll* assembly to create, import, export, and delete models. This assembly can be used from X++—the *SysModelStore* class wraps some of the functionality for easier consumption in X++.

Note When you use the table data API in an environment with version control enabled, the values of some of the fields are reset during the build process. For file-based version control systems, the build process imports .xpo files into empty layers in Microsoft Dynamics AX. The values of the *CreatedBy*, *CreatedDateTime*, *ModifiedBy*, and *ModifiedDateTime* fields are set during this import process and therefore don't survive from build to build.

Dictionary API

The Dictionary API is a type-safe reflection API that can reflect on many elements. The following code example is a revision of the preceding example that finds inventory classes by using the dictionary API. This API gives you access to more detailed type information. This example lists only abstract classes that start with the string *Invent*:

```
static void findAbstractInventoryClasses(Args _args)
{
    Dictionary dictionary = new Dictionary();
    int i;
    DictClass dictClass;
```

```
    for(i=1; i<=dictionary.classCnt(); i++)
    {
        dictClass = new DictClass(dictionary.classCnt2Id(i));

        if (dictClass.isAbstract() &&
            strStartsWith(dictClass.name(), 'Invent'))
        {
            info(dictClass.name());
        }
    }
}
```

The *Dictionary* class provides information about which elements exist and even includes system elements. For example, with this information, you can instantiate a *DictClass* object that provides information about the class, such as whether the class is abstract, final, or an interface; which class it extends; whether it implements any interfaces; what attributes it is decorated with; and what methods it includes. Notice that the *DictClass* class can also reflect on interfaces. Also notice how the class counter is converted into a class ID. This conversion is required because the IDs aren't listed consecutively.

When you run this job, you'll notice that it's much slower than the implementation that uses the table data API—at least the first time you run it. The job performs better after the information is cached.

Figure 20-2 shows the object model for the dictionary API. As you can see, some elements can't be reflected upon by using this API.

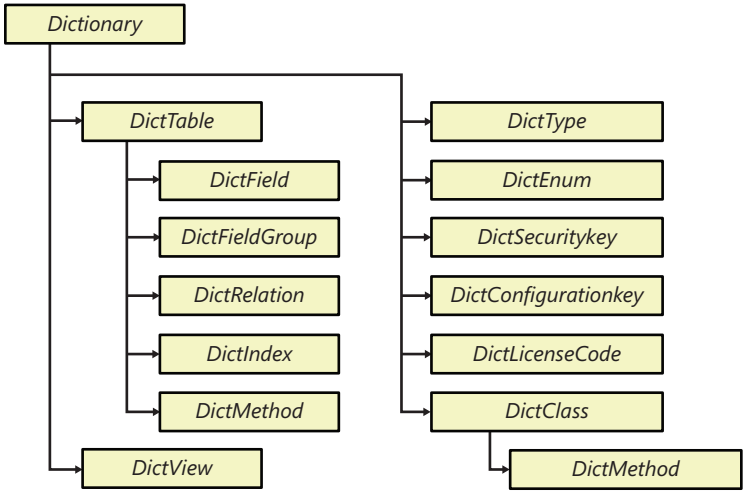


FIGURE 20-2 The object model for the dictionary reflection API.

The following example lists the static methods on the *CustTable* table and reports their parameters:

```
static void findStaticMethodsOnCustTable(Args _args)
{
    DictTable dictTable = new DictTable(tableNum(CustTable));
    DictMethod dictMethod;
    int i;
    int j;
    str parameters;

    for (i=1; i<=dictTable.staticMethodCnt(); i++)
    {
        dictMethod = new DictMethod(
            UtilElementType::TableStaticMethod,
            dictTable.id(),
            dictTable.staticMethod(i));

        parameters = '';
        for (j=1; j<=dictMethod.parameterCnt(); j++)
        {
            parameters += strFmt("%1 %2",
                extendedTypeId2name(dictMethod.parameterId(j)),
                dictMethod.parameterName(j));

            if (j<dictMethod.parameterCnt())
            {
                parameters += ', ';
            }
        }
        info(strFmt("%1(%2)", dictMethod.name(), parameters));
    }
}
```

As mentioned earlier, reflection can also be used to invoke methods on objects. This example invokes the static *find* method on the table *CustTable*:

```
static void invokeFindOnCustTable(Args _args)
{
    DictTable dictTable = new DictTable(tableNum(CustTable));
    CustTable customer;

    customer = dictTable.callStatic(
        tableStaticMethodStr(CustTable, Ffind), '1201');

    print customer.currencyName();    //Prints US Dollar
    pause;
}
```

Notice the use of the intrinsic function *tableStaticMethodStr* to reference the *find* method.

You can also use this API to instantiate class and table objects. Suppose you want to select all records in a table with a given table name. The following example shows you how:

```
static void findRecords(TableId _tableId)
{
    DictTable dictTable = new DictTable(_tableId);
    Common common = dictTable.makeRecord();
    FieldId primaryKeyField = dictTable.primaryKeyField();

    while select common
    {
        info(strFmt("%1", common.(primaryKeyField)));
    }
}
```

First, notice the call to the *makeRecord* method, which instantiates a table cursor object that points to the correct table. You can use the *select* statement to select records from the table. If you want to, you can also insert records by using the table cursor. Notice the syntax used to get a field value out of the cursor object; this syntax allows any field to be accessed by its field ID. This example prints the content of the primary key field. Alternatively, you can use the *getFieldValue* method to get a value based on the name of the field. You can use the *makeObject* method on the *DictClass* class to create an object instance of a class.

All of the classes in the dictionary API discussed so far are defined as system APIs. On top of each of these is an application-defined class that provides even more reflection capabilities. These classes are named *SysDict*<Concept>, and each class extends its counterpart in the system API. For example, *SysDictClass* extends *DictClass*.

Consider the following example. Table fields have a property that specifies whether the field is mandatory. The *DictField* class returns the value of a mandatory property as a bit that is set in the return value of its *flag* method. Testing to determine whether a bit is set is somewhat cumbersome, and if the implementation of the flag changes, the consuming application breaks. The *SysDictField* class encapsulates the bit-testing logic in a *mandatory* method. The following example shows how to use the method:

```
static void mandatoryFieldsOnCustTable(Args _args)
{
    SysDictTable sysDictTable = SysDictTable::newName(tableStr(CustTable));
    SysDictField sysDictField;
    Enumerator enum = sysDictTable.fields().getEnumerator();

    while (enum.moveNext())
    {
        sysDictField = enum.current();
```

```
            if (sysDictField.mandatory())
            {
                info(sysDictField.name());
            }
        }
    }
```

You might also want to browse the *SysDict* classes for static methods. Many of these methods provide additional reflection information and better interfaces. For example, the *SysDictionary* class provides a *classes* method that returns a collection of *SysDictClass* instances. You could use this method to simplify the earlier *findAbstractInventoryClasses* example.

Treenodes API

The two reflection APIs discussed so far have limitations. The table data API can reflect only on the existence of elements and on a small subset of element metadata. The dictionary API can reflect in a type-safe manner, but only on the element types that are exposed through this API.

The treenodes API can reflect on everything, but as always, power comes at a cost. The treenodes API is harder to use than the other reflection APIs, it can cause memory and performance problems, and it isn't type-safe.

In the following code, the example from the "Table data API" section has been revised to use the treenodes API to find inventory classes:

```
static void findInventoryClasses(Args _args)
{
    TreeNode classesNode = TreeNode::findNode(@'\Classes');
    TreeNodeIterator iterator = classesNode.AOTiterator();
    TreeNode classNode = iterator.next();
    ClassName className;

    while (classNode)
    {
        className = classNode.treeNodeName();
        if (strStartsWith(className, 'Invent'))
        {
            info(className);
        }

        classNode = iterator.next();
    }
}
```

First, notice how you find a node in the AOT based on the path as a literal. The *AOT* macro contains definitions for the primary AOT paths. For readability, the examples in this chapter don't use the macro. Also notice the use of a *TreeNodeIterator* class to iterate through the classes.

The following small job prints the source code for the *find* method on the CustTable table by calling the *AOTgetSource* method on the *treenode* object for the *find* method:

```
static void printSourceCode(Args _args)
{
    TreeNode treeNode =
        TreeNode::findNode(@'\Data Dictionary\Tables\CustTable\Methods\find');

    info(treeNode.AOTgetSource());
}
```

The treenodes API provides access to the source code of nodes in the AOT. You can use the class *ScannerClass* to turn the string that contains the source code into a sequence of tokens that can be compiled.

In the following code, the preceding example has been revised to find mandatory fields on the table *CustTable*:

```
static void mandatoryFieldsOnCustTable(Args _args)
{
    TreeNode fieldsNode = TreeNode::findNode(
        @'\Data Dictionary\Tables\CustTable\Fields');

    TreeNode field = fieldsNode.AOTfirstChild();

    while (field)
    {
        if (field.AOTgetProperty('Mandatory') == 'Yes')
        {
            info(field.treeNodeName());
        }

        field = field.AOTnextSibling();
    }
}
```

Notice the alternate way of traversing subnodes. Both this and the iterator approach work equally well. The only way to determine whether a field is mandatory with this API is to know that your node models a field. Field nodes have a property named *Mandatory*, which is set to *Yes* (not to *True*) for mandatory fields.

Use the *Properties* macro when referring to property names. This macro contains text definitions for all property names. By using this macro, you avoid using literal names, like the reference to the *Mandatory* property in the preceding example.

Unlike the dictionary API, which can't reflect all elements, the treenodes API reflects everything. The *SysDictMenu* class exploits this capability, providing a type-safe way to reflect on menus and

menu items by wrapping information provided by the treenodes API in a type-safe API. The following job prints the structure of the MainMenu menu, which typically is shown in the navigation pane:

```
static void printMainMenu(Args _args)
{
    void reportLevel(SysDictMenu _sysDictMenu)
    {
        SysMenuEnumerator enumerator;

        if (_sysDictMenu.isMenuReference() ||
            _sysDictMenu.isMenu())
        {
            setPrefix(_sysDictMenu.label());
            enumerator = _sysDictMenu.getEnumerator();
            while (enumerator.moveNext())
            {
                reportLevel(enumerator.current());
            }
        }
        else
        {
            info(_sysDictMenu.label());
        }
    }

    reportLevel(SysDictMenu::newMainMenu());
}
```

Notice how the *setPrefix* function is used to capture the hierarchy and how the *reportLevel* function is called recursively.

You can also use the treenode API to reflect on forms and reports, and their structure, properties, and methods. The Compare tool in MorphX uses this API to compare any node with any other node. The *SysTreeNode* class contains a *TreeNode* class and implements a cascade of interfaces, which makes *TreeNode* classes consumable for the Compare tool and the Version Control tool. The *SysTreeNode* class also contains a powerful set of static methods.

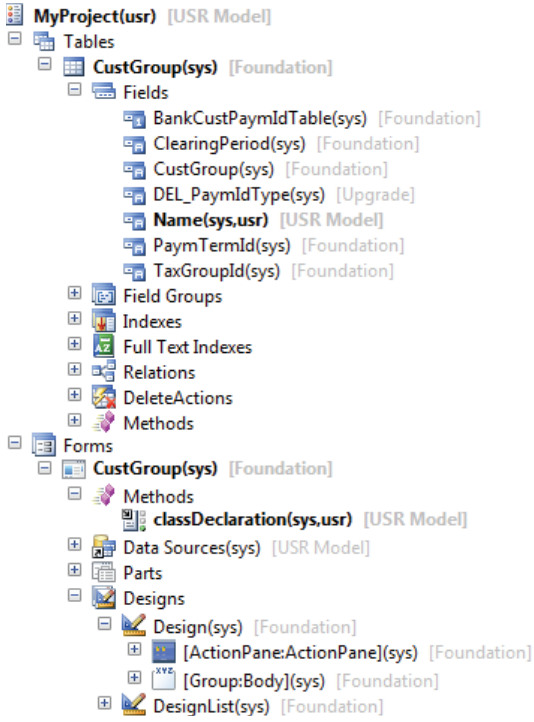
The *TreeNode* class is actually the base class of a larger hierarchy. You can cast instances to specialized *TreeNode* classes that provide more specific functionality. The hierarchy isn't fully consistent for all nodes. You can browse the hierarchy in the AOT by clicking System Documentation, clicking Classes, right-clicking TreeNode, pointing to Add-Ins, and then clicking Type hierarchy browser.

Although this section has only covered the reflection functionality of the treenodes API, you can use the API just as you do the AOT designer. You can create new elements and modify properties and source code. The Wizard Wizard uses the treenodes API to generate the project, form, and class implementing the wizard functionality. You can also compile and get layered nodes and nodes from the baseline model store. The capabilities that go beyond reflection are very powerful, but proceed with great care. Obtaining information in a non-type-safe manner requires caution, but writing in a non-type-safe manner can lead to catastrophic situations.

TreeNodeType

Different types of treenodes have different capabilities. The *TreeNodeType* class can be used to reflect on the treenode. The *TreeNodeType* class provides reliable alternatives to making assumptions about a treenode's capabilities based on its properties. In previous versions of Microsoft Dynamics AX, fragile assumptions could be found throughout the code base; for example, it was assumed that a treenode supported version control if the treenode had a *utilElementType* and no parent ID.

The *TreeNodeType* class provides a method that returns the type identification, plus seven methods that return Boolean values providing information about the treenode's capabilities. The usage of these methods are described later in this section. Figure 20-3 shows the information that the *TreeNodeType* class provides for each treenode in a project containing a table and a form. The left side of the illustration shows a screenshot of the project itself. The right side contains a table that, for each treenode, lists the treenode type ID and capabilities.



ID	isConsumingMemory	isGetNodeInLayerSupported	isLayerAware	isModelElement	isRootElement	isUtilElement	isVCSControllableElement
30							
27							
204	X	X	X	X	X	X	X
110							
416		X	X	X		X	
405		X	X	X		X	
405		X	X	X		X	
405		X	X	X		X	
405		X	X	X		X	
405		X	X	X		X	
405		X	X	X		X	
121							
117							
316							
120							
167							
191							
27							
504	X	X	X	X	X	X	X
191							
201			X	X			
140			X	X			
144							
1435							
141							
148			X	X			
152			X	X			
152			X	X			
148			X	X			

FIGURE 20-3 Information provided by the *TreeNodeType* class for the treenodes in a table and a form.

- ***ID*** The ID of the treenode type is defined in the system and is available in the *TreeNodeSysNodeType* macro. Nodes with the same ID have the same behavior.
- ***isConsumingMemory*** Tree nodes in MorphX contain data that the Microsoft Dynamics AX runtime doesn't manage, and the memory for a node isn't automatically deallocated. For each node where *isConsumingMemory* is true, you should call the *treenodeRelease* method to free the memory when you no longer reference any subnodes. Alternatively, you can use the *TreeNodeTraverser* class because the class will handle this task for you. For an example of this, see the *traverseTreeNodes* method of the *SysBpCheck* class.
- ***isGetNodeInLayerSupported*** With treenodes that support the *getNodeInLayer* method, you can navigate to versions of the node in other layers. In other words, you can access the nodes in the lower layers by using this method.
- ***isLayerAware*** Treenodes that are layer-aware display a layer indicator in the AOT; for example, *SYS* or *USR*. You can retrieve the layer of a node by using the *AOTLayer* method, and you can retrieve all layers that are available by using the *AOTLayers* method. Note that the *AOTLayers* method does not roll up layers for subnodes: this method returns what is shown in the AOT. The roll-up layer information is available through the *ApplObjectLayerMask* method, which is used in the AOT to determine whether a node is shown in bold. If a node is bold in the AOT, either the node itself or one of its subnodes is present in the current layer.
- ***isModelElement*** Treenodes that are model elements are represented by a record in the SysModelElement table.
- ***isRootElement*** A root element is placed in the root of the treenode hierarchy, and the *RootModelElement* field for all submodel elements references the root element's *recid*.
- ***isUtilElement*** If the treenode is a *UtilElement*, a corresponding record can be found in the *UtilElements* view. Further, the primary key information can be retrieved through the treenode's *utilElement* method.
- ***isVCSControllableElement*** You can use the *isVCSControllableElement* method to determine the granularity of the file-based artifacts that are stored in a version control system. In most cases the granularity under version control is per root element; in other words, you are working on entire forms, classes, and tables under version control. However, for Microsoft Visual Studio elements, the granularity is different, and you are able to work on individual Visual Studio files; for example, .cs files.

```
if (treenode.treeNodeType().isVCSControllableElement())
{
    versionControl.checkOut(treenode);
}
```

The following example shows how to access the type information for a treenode:

```
static void GetTreeNodeTypeInfo(Args _args)
{
    TreeNode treeNode = TreeNode::findNode(
        @'\Data Dictionary\Tables\CustTable\Methods\find');
    TreeNodeType treeNodeType = treeNode.treeNodeType();

    info(strFmt("Id: %1", treeNodeType.id()));
    info(strFmt("IsConsumingMemory: %1", treeNodeType.isConsumingMemory()));
    info(strFmt("IsGetNodeInLayerSupported: %1",
        treeNodeType.isGetNodeInLayerSupported()));
    info(strFmt("IsLayerAware: %1", treeNodeType.isLayerAware()));
    info(strFmt("IsModelElement: %1", treeNodeType.isModelElement()));
    info(strFmt("IsRootElement: %1", treeNodeType.isRootElement()));
    info(strFmt("IsUtilElement: %1", treeNodeType.isUtilElement()));
    info(strFmt("IsVCSControllableElement: %1",
        treeNodeType.isVCSControllableElement()));
}
```

Note You can use the *TreeNodeType* class to reflect on the meta model. This class functions on a higher level of abstraction—instead of reflecting on the elements in the AOT, it reflects on element types. The *SysModelMetaData* class provides another way of reflecting on the meta model.

CHAPTER 21

Application models

In this chapter

Introduction

Microsoft Dynamics AX 2012 introduces a new era for managing metadata artifacts.

In previous versions of the product, metadata artifacts were stored in Application Object Data (AOD) files. These files served two purposes. First, they acted as the deployment vehicle for metadata; for example, you could copy an AOD file from the source system to the target system. Second, they provided runtime storage for model elements. The AOD file provided the physical storage for a native indexed sequential access method (ISAM) database that contained the metadata, and the runtime read model elements from this storage.

This method of managing metadata artifacts was not optimal. From a deployment perspective, AOD files didn't allow side-by-side installation of metadata in the same layer, didn't contain any structured information about their contents, and couldn't be digitally signed. From a runtime perspective, the AOD format supported only one table and provided no capability for adding or changing columns. To support the evolution of runtime scenarios, the product had to move toward a relationally correct schema.

In Microsoft Dynamics AX 2012, metadata is stored in the Microsoft SQL Server database along with business data. The tables containing the metadata are called the *model store*. This change removes all obstacles to providing a relationally correct schema. Elements, such as classes, tables, forms, methods, and controls, in the model store are grouped into models. Each model can be exported to a file-based format with the extension *.axmodel*. These files are managed assemblies and therefore, support digital signing, which makes them tamper-proof. Model files are the primary deployment vehicle for model elements in Microsoft Dynamics AX 2012.

In previous versions of Microsoft Dynamics AX, independent software vendors (ISVs) sometimes delivered textual source code files (in XPO format) to customers. Releasing source code files is an undesirable practice, but it was used to overcome restrictions on element IDs for combining multiple solutions in the same layer. With Microsoft Dynamics AX 2012, you no longer need to release source code files to customers. Model files do not contain element IDs, you can install multiple models in the same layer, and each model file contains a manifest that describes the model.

Note XPO files are still fully supported in Microsoft Dynamics AX 2012. Developers primarily use them to exchange source code and for storage in a version control system.

In previous versions of Microsoft Dynamics AX, a complete set of AOD files could be deployed in one operation, typically when a solution was moved from a staging system to a production system. The primary concern in this scenario is to reduce downtime. Copying all AOD files in one operation reduced downtime because there was no need to regenerate the Application Object Index (AOI) file or to recompile the application code. To satisfy the same need in Microsoft Dynamics AX 2012, you can export an entire model store in a binary file with the extension *.axmodelstore*. This file can be imported into the target system, and the system's downtime is restricted to the time it takes to restart the Application Object Server (AOS).

Layers

The Microsoft Dynamics AX 2012 runtime executes a program defined in the MorphX development environment. The program itself is defined as elements.

Unlike most systems, Microsoft Dynamics AX can contain multiple definitions of each element; for example, multiple implementations of the same method. These element definitions are stacked in layers. The Microsoft Dynamic AX runtime uses the element definitions from the highest layer in which they are found. For example, a method defined in the SYS layer (the lowest layer) is not used if another definition of the same method exists in any other layer.

This layered development approach provides several benefits, the most important being the ability to customize the program shipped by Microsoft, Microsoft partners, and ISVs without editing the original source code.

Note The layer metaphor is also used for graphical drawing tools. With layering, you can draw on top of an existing image without touching the original image underneath. Layers in Microsoft Dynamics AX work the same way, but with code and properties instead of pixels, shapes, and shades.

When you start Microsoft Dynamics AX 2012, you specify which layer you want to start in. By default, you start in the *USR* layer. Any element you create or edit is stored in that layer. If you edit an

element that exists in a lower layer, a copy of the element with your edits is moved to your layer. This process is known as *over-layering*.

Other benefits of layers include the ability to revert to the original definition of an element by deleting the over-layering version. You can also compare versions of an element; for example, to see which lines of code you have inserted. This is particularly useful during upgrade.

Caution Develop your solution one layer at a time, from the bottom up. Working in multiple layers at the same time on the same AOS is highly discouraged—even for different users. For more information see the Microsoft Dynamics AX 2012 white paper "Developing Solutions in a Shared AOS Development Environment" (*http://www.microsoft.com/download/en/details.aspx?id=26919*).

The process of editing an element from a higher layer than the current layer is known as under-layering. By design, these edits are routed to the higher layer.

Microsoft Dynamics AX 2012 has 16 metadata layers, each with its own purpose. Table 21-1 describes these layers in alphabetical order.

TABLE 21-1 Metadata layers.

Name	Description
USP (topmost)	Patch layer for the *USR* layer.
USR	User layer. This layer is intended for minor final customizations by power users of the system. These might include simple changes to the layout of a form and new security roles and tasks defined for the company.
CUP	Patch layer for the *CUS* layer.
CUS	Customer layer. This layer enables customer-specific customizations and extensions to the solution. This layer is typically developed by a Microsoft partner or an in-house development team.
VAP	Patch layer for the *VAR* layer.
VAR	Value-added retailer layer. Microsoft partners use this later to deliver customizations and extensions that typically are installed by multiple customers.
ISP	Patch layer for the *ISV* layer.
ISV	Independent Software Vendor layer. Registered Microsoft Dynamics AX ISVs can deliver solutions in this layer.
SLP	Patch layer for the *SLN* layer.
SLN	Solution layer. This layer contains vertical solutions provided by Microsoft partners.
FPP	Patch layer for the *FPK* layer.

Name	Description
FPK	Feature pack layer. This layer contains industry solutions provided by Microsoft. Public sector, Process industries, Retail, and Service industries are available in this layer.
GLP	Patch layer for the *GLS* layer.
GLS	Global solution layer. This layer contains regional extensions to the horizontal solution provided in the *SYS* layer.
SYP	Patch layer for the *SYS* layer.
SYS (Lowest)	System layer. Microsoft platform and foundation layer. This contains the horizontal application developed by Microsoft.

Within a layer, metadata elements are grouped into models. Models are covered in the following section.

Models

A model is a logical container of metadata elements, such as forms, tables, reports, and methods. For more information about element types, see Chapter 1, "Architectural overview."

The model store can contain as many models as you want. Figure 21-1 shows the relationship between layers, models, and elements.

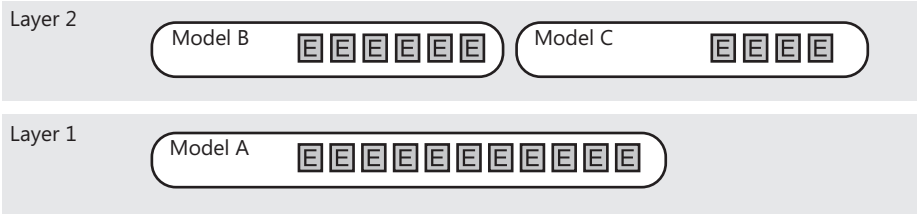


FIGURE 21-1 Layers, models, and elements.

Note The term *model* was selected for several reasons. First, any solution built in Microsoft Dynamics AX is a model of a real-life business. Second, a model is irreducible—even a part of a model is a model, so the term covers stand-alone solutions, extensions, customizations, patches, and so on. And finally; the term is simple and catchy—it will quickly become a part of your Microsoft Dynamics AX vocabulary.

You can have as many models in each layer as you want. This means you can segment your layer into as many models as you like. Here are some development scenarios where this can be useful:

- **When you deliver more than one solution** You can have a model for each solution you are working on. This enables you to work on them simultaneously.

- **When your solution is getting too large** You can segment your solution into several models and have each team or team member work on a different one. A model can be either self-contained or have dependencies on other models. This enables you to clearly define responsibilities between the model clearly, define the application programming interfaces (APIs) between the models clearly, build the models individually, and so on.
- **When you write unit tests** You can have a model for your production code and a model for your unit tests. These enable you to import all your unit tests, run them, and remove them from the system easily.

You can get a model in two ways: you can either create one on your own, or you can receive one from someone else. Because you can have as many models as you want in each layer, you can deploy models from several sources in the same layer.

Suppose you are a customer and want to install two ISV solutions that are available in the *ISV* layer. In previous versions, you would have had a tough choice to make. Either you picked your favorite solution and learned to live without the other one, or you invested in having the two solutions merged into one layer. This merge was technically challenging and costly if updates to either solution were released. In Microsoft Dynamics AX 2012, you just download the two models and then use AXUtil, a command-line utility that is available when you install Microsoft Dynamics AX, to import them. When a new version of either model is released, you simply use AXUtil to update the model.

The layer model element containment hierarchy has one restraint: an element can be defined only once per layer. In other words, you cannot install two models containing a definition of the same element in the same layer on the same system. Here are some examples:

- Two models that contain a class named *MyClass* cannot be installed side by side in the same layer.
- Two elements of the same type under the same parent element (or without a parent) cannot coexist in the same layer if they have same name or the same ID. For example, a table cannot have two fields with the same name or two fields with the same ID, and you cannot have two display menu items with the same name in the same layer.

This limitation enables the Microsoft Dynamics AX runtime to select the right version of an element to execute based on the layer in which it's contained.

You can encounter this limitation in two ways:

- You create an element and accidentally give it a name that is being used for another element in another model. A good way to avoid this is to prefix your new elements with a short string that uniquely identifies you, your company, or your solution. This practice is widely used.
- You customize an existing element that also has been customized by someone else in another model. There are various ways to limit the number of customized elements, such as by using events, but in some situations this is unavoidable.

Note Because element IDs are assigned at deployment time, the system automatically avoids duplicate IDs. Element IDs are covered next.

Element IDs

All element types have names, and a few element types also have an integer-based ID. The ID is a 32-bit integer and is assigned at installation time. This means that the same element might have a different ID on two different systems.

Note In previous versions of Microsoft Dynamics AX, IDs were 16-bit integers that were assigned at creation time from a pool of IDs for each layer. This could result in running out of IDs and ID collisions when installing solutions developed independently.

Two new properties have been introduced to support scenarios in which elements are upgraded or renamed:

- The *LegacyID* property has been added to the few element types that have an element ID. This property enables elements to keep their IDs from the AOD files when imported as a model file to a model store.
- The *Origin* property is a globally unique identifier (GUID) that uniquely identifies the element and eliminates the risk of a collision. The *Origin* property has been added to all root element types and element types with IDs. This property enables AXUtil (and other components) to recognize renamed elements during import.

AXUtil assigns element IDs when a model is imported, based on the following rules:

1. If an element already exists with the same *Origin*, replace the element and reuse its ID, else proceed to step 2.
2. If an element already exists with the same *Type*, *Name*, and *ParentID*, replace the element and reuse its ID, else proceed to step 3.
3. If the imported element has a *LegacyID*, and the *LegacyID* is available on the target system, add the element setting the ID to equal the *LegacyID*, else proceed to step 4.
4. Assign a new installation-specific ID that does not collide with any *LegacyID*s (greater than 60,000 for fields, and greater than 1,000,000 for all other element types).

This algorithm ensures that IDs are maintained in simple and advanced import scenarios. Consider a scenario where you have delivered several variations of the same solution to multiple customers as AOD or XPO files in Microsoft Dynamics AX 2009. This means that you probably maintain a copy of the source code for each of your customers in order to service them. As the number of customers

grows, so does the incentive to consolidate the variations into one common solution. Step 2 in the algorithm ensures that IDs are maintained on the customer's installation when the customer upgrades from a specialized solution to a common solution.

During regular development, the system maintains IDs, and you do not need to be concerned with them. However, you still need to pay attention to IDs in two situations: when upgrading a model and when moving from test to production. These two situations are covered in the following sections.

Note The data export/import functionality available under System administration automatically adjusts element ID references in the imported business data to match the element IDs on the target system. This adjustment skips all unstructured data. If you need to reference an element in a persisted container, for example, it is a best practice to reference the element by name.

Create a model

Before you implement your solution, you need to create a new model. You can create a model in several ways. You can do so in MorphX through Tools > Model Management > Create model, you can use Windows PowerShell from the Microsoft Dynamics AX 2012 Management Shell, or you can use AXUtil. The examples in this chapter use the latter.

```
AXUtil create /model:"My Model" /Layer:USR
```

Notice that you have to specify which layer the model belongs to. A model cannot span layers.

Note Each layer has a system-defined model. If you don't create your own model, the system-defined model is used. The system-defined model has certain deployment restrictions because its manifest is read-only. It is highly recommended that you create your own models.

After creating the model, you need to select it. In the status bar in MorphX, you can see the current model. You can change the model by clicking it. All elements that you create in the AOT are contained in the current model.

You can easily see which model an element belongs to by inspecting the element's properties. You can also enable a model indicator in the AOT on each element in Tools > Options > Development. You can move an element between models in the same layer by right-clicking the element, and then clicking Move to model.

Now that you have your model, you are ready to implement your solution.

Note You can delete any model by using the AXUtil delete command. This applies to models you have created and those you have installed. By using the */layer:<layer>* option, you can even delete all models in a layer.

Prepare a model for publication

When your implementation is complete, it is time to prepare the model for publication. But before you do, you may want to create a MorphX project that contains the elements in your model. You can create the project by using Tools > Model Management > Create project. This allows you to ensure that the model contains what you expect before you publish it. You can also use AXUtil to list the elements in a model.

```
AXUtil view /model:"My Model" /verbose
```

Preparing your model for publication consists of the following steps:

1. Set the model manifest.
2. Export the model to disk.
3. Add a digital signature.

Set the model manifest

A model is the container for your solution. You can describe your model in the model manifest. Table 21-2 contains a description of the properties of the model manifest. When you export your model, the exported file contains the manifest. The manifest helps consumers of your model understand its contents before installing it.

The simplest way to edit a manifest is to use XML notation:

```
AXUtil manifest /model:"My Model" /xml >MyManifest.xml
notepad MyManifest.xml
AXUtil edit /model: "My Model" @MyManifest.xml
```

TABLE 21-2 Model manifest properties.

Type	Description
Name	The full name of the model. The name often contains multiple words and typically the same name as used in marketing materials and other public documents.
Publisher	Publisher of the model; for example, "Microsoft Corp." The *Name* and *Publisher* properties must constitute a unique key. In other words, you cannot install two models with the same *Name* and *Publisher* in the same model store.
Description	A longer text string describing the model.
Display Name	A friendly name that is shown in the development environment, including in the status bar in MorphX and in the AOT. The *Display Name* is typically significantly shorter than *Name*, and it is often just a mnemonic value.
Version	A four-part version number; for example, 1.0.0.0.
Category	The category of the model. Models are grouped into four categories: ■ *Standard* A regular model. ■ *Hotfix* A model that contains a fix for an issue in another model. Hotfixes are typically delivered in a patch layer. ■ *Virtual* A model that is automatically created as a result of conflicting elements during model import, when using the */conflict:push* option. ■ *Temporary* A model that is used during the import process. At the end of the import, it will be deleted again. In most situations, you should set this property to *Standard*.
Details	The extension point of the manifest. If you need to capture more details about your model, you can place it here. The *Details* property must contain well-formed XML text. The model store and the Microsoft Dynamics AX runtime do not use this property. *Standard* models from Microsoft leave this property empty; *Hotfix* models include information about knowledge base (KB) articles in this property.

Note It is not possible to express dependencies between models in the model manifest. However, if you use the slipstream installation mechanism of the Microsoft Dynamics AX Setup program, you can control the installation sequence.

Export the model

When the manifest of the model is populated it is time to export the model to disk so that you can share it outside your organization:

```
AXUtil export /model:"My Model" /file:MyModel.axmodel
```

The extension *.axmodel* is used for model files. The model file contains all of the elements in the model, plus the model manifest. Model files are element ID–agnostic. When the elements in the model file are imported into a target system, they are assigned new installation-specific IDs. XPO files handle element IDs similarly.

Tip Under the covers, a model file is a managed assembly. This means you can use assembly reflection tools, like *ildasm*, to inspect the contents.

You can verify the model file contents using AXUtil:

```
AXUtil view /file:MyModel.axmodel /verbose
```

Note AXUtil is a powerful tool, and, for a command-line tool, also quite user friendly. Notice that some commands, like view and manifest, can be used either on a model in the model store and on a model file on disk. The most frequently used parameter is the */model* parameter. In the examples in this chapter, the name of the models are provided when using this parameter, but you can also specify the model ID (which typically is much shorter, and thus more convenient to write), or the model's manifest XML file. This latter option is particular useful when writing command scripts, such as build scripts for version control. All commands also supports a */verbose* parameter, which displays additional details about the command execution. For a complete list of commands and options, try *AXUtil /?*.

Sign the model

The model file is now ready to be shared. However, you should consider one more thing before making it publicly available. The model file contains binary code, and as such, this code can potentially harm a system, especially if the code is tampered with after it leaves your hands. To ensure that the customers who receive your model file can trust the file—or at least be able to tell that the model comes from a trustworthy source—you can add a digital signature to the model file.

When a signed model is imported, you are guaranteed the model file hasn't been tampered with since it was exported. If it has been tampered with, the import process fails. Microsoft Dynamics AX 2012 supports two ways of signing a model: strong name signing and Authenticode signing.

Strong name signing

To strong name sign a model, you need to use the .NET Framework Strong Name Tool, SN.exe, to generate a key/pair file. When you export your model to an *.axmodel* file, you specify the key to sign the model with.

```
SN -k mykey.snk
AXUtil export /model:"My Model" /file:MyModel.axmodel /key:mykey.snk
```

Authenticode signing

If you are a publisher of models, such as an ISV that provides models for download, consider Authenticode signing your model. If you do, your customers are guaranteed that the file hasn't been tampered with and that you created the model.

When an Authenticode signed model is imported, the model's publisher is authenticated. This means the model file can be traced to you.

To Authenticode sign a model file, first export it by using AXUtil. Then you use the SignTool to perform the actual signing:

```
signtool sign /f mycertprivate.pfx /p password MyModel.axmodel
```

Import model files

If you have received or downloaded a model file, you can import it by using AXUtil. The model file is always imported into the layer it was exported from. It is a best practice to stop the AOS before importing model files.

```
AXUtil import /file:SomeModel.axmodel
```

Note You don't have to specify file extensions when using AXUtil. The tool automatically adds the right extension if it is omitted. In this book, extensions are included for clarity.

Figure 21-2 shows a model that has been successfully imported into a layer in which a model already exists.

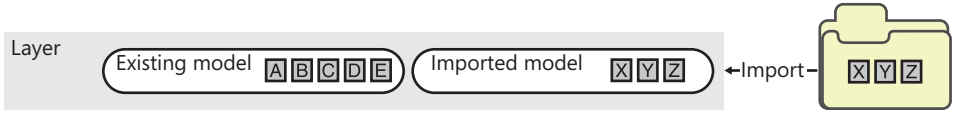


FIGURE 21-2 Side-by-side installation of two models.

The import operation will be cancelled if one or more elements from the model file are already defined in the layer into which the model is being imported. If you rerun the import operation with the */verbose* option, you will get a list of conflicting elements.

```
AXUtil import /file:SomeModel.axmodel /verbose
```

You have two options for proceeding with the import: *overwrite* and *push*.

Import model files with the *overwrite* option

You can decide to overwrite existing conflicting elements with the new definitions of the model element from the model file. You do so by specifying the */conflict:overwrite* option on the import command:

```
AXUtil import /file:SomeModel.axmodel /conflict:overwrite
```

Figure 21-3 shows the result of a successful import using the */conflict:overwrite* option. The imported model and the existing model that contained conflicting elements are linked after this operation. The models are linked because the existing model now is partial. The linkage prevents the imported model from being uninstalled unless the existing model is also uninstalled. This option is primarily used when delivering cumulative patches or service packs.

FIGURE 21-3 Side-by-side installation of two models using the */conflict:overwrite* option.

Import model files with the *push* option

The most typical solution to solve conflicts is the */conflict:push* option. This option creates a new virtual model in a higher layer containing the conflicting elements:

```
AXUtil import /file:SomeModel.axmodel /conflict:push
```

Figure 21-4 shows the result of a successful import operation using the */conflict:push* option. The elements in the virtual model are identical to those imported. In other words, existing models are not affected. After importing the model, log in to the layer containing the virtual model to resolve the conflict. You can use the compare functionality in the AOT to compare the conflicting versions of each element and resolve the conflict.

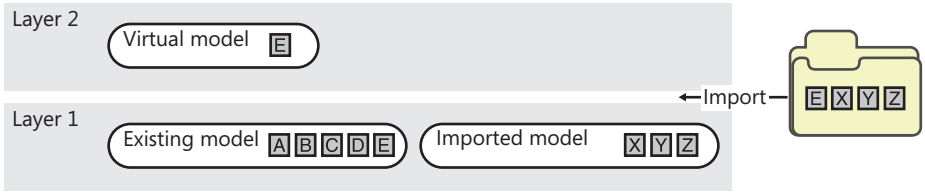


FIGURE 21-4 The result of an import operation using the */conflict:push* option.

If you resolve all conflicts in the same layer, there is no risk of running out of layers when using the */conflict:push* option. However, you may need to move the resolved elements into the same layer manually. For example, if you import a third model that conflicts with elements in the virtual model in Figure 21-4, the resulting virtual model will be created in Layer 3. After you resolve the conflicts in Layer 3, move the elements in Layer 3 to Layer 2. The easiest way to accomplish this is by exporting the elements from Layer 3 to an XPO file, deleting them, and importing them into Layer 2.

By default, the virtual model is created in the layer just above the layer the model is imported into. If you don't have developer access to that layer, you can force AXUtil to create the virtual model in a different layer (for example the *USR* layer) by using the */targetlayer* option, as shown in the following example:

```
AXUtil import /file:SomeModel.axmodel /conflict:push /targetlayer:USR
```

Upgrade a model

When you receive a newer version of a model and you want to replace the older version in the model store, it is important that you import the new model on top of the existing model. AXUtil automatically detects that the model already exists in the model store and performs the actions that are required to ensure the consistency of the model store:

```
AXUtil import /file:NewerModel.axmodel
```

By default, AXUtil import enters upgrade mode when a model with the same name and publisher already exists. Sometimes a model might be renamed or replaced by multiple new models (as the result of segmentation work, for example), or multiple models might be merged into one consolidated model. AXUtil supports upgrading existing models with new models. You can force AXUtil to use this mode by listing the files and models to upgrade, separated by a comma:

```
AXUtil import /file:f1,f2,f3 /replace:m1,m2,m3,m4,m5
```

Caution You might be tempted to uninstall an existing model before importing a newer version of the model, but if you do so, AXUtil does not enter upgrade mode and it assigns new element IDs to all elements being imported. This results in data corruption because business data contains the original element IDs. All references to elements in the uninstalled model will break. For more information, see the "Element IDs" section earlier in this chapter.

Move a model from test to production

It is a good practice to have a test or staging environment where changes to the system are prepared and tested before being deployed to a live production environment.

The model store provides features that you can use to export all model store metadata to a binary file and import it into a target system. Doing this creates a binary, identical copy of metadata between the two systems, including element IDs. Model store files have the extension *.axmodelstore*. Besides the metadata, the model store files also contain the compiled pcode and common intermediate language (CIL) code. This means that you do not have to compile the target system.

Note The size of model store files depend on the contents of the model store. A model store file for the standard installation of Microsoft Dynamics AX 2012 is about 2 GB. Model store files compress well, typically over 80 percent, and thus can be used as a simple backup.

Figure 21-5 shows the cleanest way of creating and preparing a test environment and deploying it to production. Variations and post-setup tasks to this process exist. For a thorough description, see the Microsoft Dynamics AX 2012 white paper "Deploying Customizations Across Microsoft Dynamics AX 2012 Environments" (*http://www.microsoft.com/download/en/details.aspx?id=26571*).

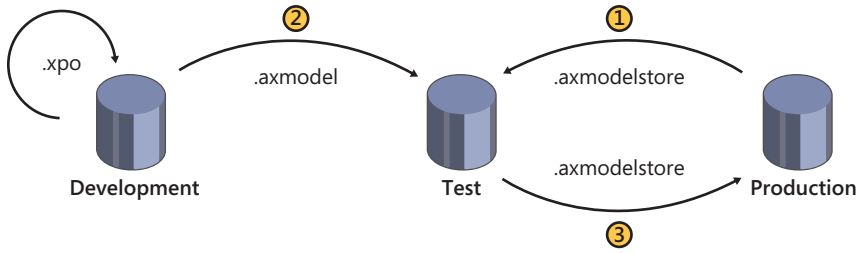


FIGURE 21-5 Creating and preparing a test environment and deploying a model store to production.

Note XPO files are not used in the process of deploying a system from test to production. They are mentioned in Figure 21-5 to show the scenarios that the three file formats should be used in. XPO files should be used for sharing source code between developers.

Create a test environment

The goal of creating a test environment is to ensure that the metadata in the model store is identical to the metadata in the model store in the production environment. The simplest way to achieve the goal is to create a new installation of Microsoft Dynamics AX, and then move the metadata from production to test. To move the metadata, you first need to export the model store from the production environment:

```
AXUtil exportstore /file:ProductionStore.axmodelstore
```

On the test system, you stop the AOS and then import the model store file:

```
Net stop AOS60$01
AXUtil importstore /file:ProductionStore.axmodelstore
Net start AOS60$01
```

Prepare the test environment

The goal of preparing the test environment is to update the system with new metadata, typically by installing new models or upgrading existing models. You import or upgrade models as explained earlier in this chapter.

After you import the models, start the Microsoft Dynamics AX client and complete the installation checklist. The most important steps are the compilation to pcode and CIL, because the products of these steps are part of the model store.

Extensive validation of the system is also recommended. Ensure that you validate both that the new functionality behaves as expected and that existing functionality hasn't regressed.

Deploy the model to production

The goal of deploying to production is to ensure that the metadata on the production system is updated with the metadata from the test environment. To move the metadata, you first need to export the model store from the test environment:

```
AXUtil exportstore /file:TestStore.axmodelstore
```

Import the model store file on the production system. To minimize downtime, AXUtil supports a two-phase import process. The first phase imports the metadata to a new schema in the database. This takes a few minutes and can occur while the production system is still live. The second phase replaces the model store metadata with the imported metadata from the schema. The takes a few seconds and must occur while the AOS is stopped.

Create a new schema:

```
AXUtil schema /schemaname:TransferSchema
```

Import the model store file into the new schema:

```
AXUtil importstore /file:TestStore.axmodelstore /schema:TransferSchema
```

When all users are logged off, stop the AOS:

```
Net stop AOS60$01
```

Apply the changes to the model store to move the new schema to the active schema:

```
AXUtil importstore /apply:TransferSchema /backupschema:dbo_backup
```

Restart the AOS:

```
Net start AOS60$01
```

Note Notice the use of the */backupschema* option in the example. With this option, you can quickly revert to the original metadata if unexpected issues arise. When you no longer need the backup schema, you can delete it by using the AXUtil *schema /drop:*<schemaname> command.

At this stage, the metadata in the production environment is identical to the metadata in the test environment. A few more tasks must be performed before the system is ready for users. These include synchronizing the database, creating role centers, deploying web content, setting up workflows, deploying cubes, importing integration ports, and deploying reports. For more information about these tasks, see the white paper "Deploying Customizations Across Microsoft Dynamics AX 2012 Environments" (*http://www.microsoft.com/download/en/details.aspx?id=26571)*.

Element ID considerations

Business data references metadata element IDs. The process outlined in the previous sections ensures that the element IDs in the production system remain unchanged, and thus ensures the integrity of the business data.

This is achieved by only exchanging metadata between test and production through model store files, which maintains the element IDs. For example, if the element IDs in the test environment and production environment are unsynchronized because XPO files or model files have been imported into both systems, you must rebuild the test environment.

The *importstore* command has a built-in safety mechanism. The command ensures that element IDs in the target system are identical to the element IDs in the file. If any conflicts are detected, the import operation stops. You can use the */verbose* option to get a list of the conflicts, and the */idconflict:overwrite* option to continue with the import operation anyway. Use the latter option only on a system where you don't care about the data — never in a production environment.

For more information, see the "Element IDs" section earlier in this chapter.

Model store API

The AXUtil utility used in all examples in this chapter provides a command-line interface to the model store commands offered by the model store API. A PowerShell interface is also available from the Microsoft Dynamics AX 2012 Management Shell.

Both these interface implementations use the managed assembly file *AXUtilLib.dll*. You can also use this assembly if you want to automate any model store operations. The assembly is referenced in X++, so you can easily access the model store API from X++. Some of the most common commands are available from the *SysModelStore* class.

The model store API also contains a method to generate license keys for a license code in the AOT based on the license holder name and serial number. The following example shows how to invoke this method from a managed website in an automated license purchasing scenario. For more information about how to protect your solution with a license code, see Chapter 14, "Customizing and extending Microsoft Dynamics AX."

```
using Microsoft.Dynamics.MorphX;
using Microsoft.Dynamics.AX.Framework.Tools.ModelManagement;

protected void Submit_Click(object sender, EventArgs e)
{
    string certPath = @"c:\Licenses\MyCertPrivate.pfx";
    string licensePath = @"c:\Licenses\" + Customer.Text + "-license.txt";
    string licenseCodeNameInAot = "MyLicenseCode";
    string certificatePassword = "password";  //TODO: Move to secure storage
    AXUtilContext context = new AXUtilContext();
    AXUtilConfiguration config = new AXUtilConfiguration();

    LicenseInfo licenseInfo = new LicenseInfo(licensePath, certPath,
                    licenseCodeNameInAot, Customer.Text, Serial.Text,
                    null, certificatePassword);
```

```
    config.LicenseInfo = licenseInfo;
    AXUtil AXUtil = new AXUtil(context, config);
    if (AXUtil.GenerateLicense())
    {
        Response.AddHeader("Content-Disposition",
                           "attachment;filename=license.txt");
        Response.TransmitFile(licensePath);
        Response.Flush();
        Response.End();
    }
}
```

APPENDIX

Resources for code upgrade

The resources listed in this section provide guidance and best practices for upgrading source code to Microsoft Dynamics AX 2012. Some of these resources contain information for system administrators, in addition to developers. Many of these resources are updated as new information becomes available or in response to customer requests, so check back often for the latest information.

Resource	Description and location
Upgrade to Microsoft Dynamics AX 2012	**Description:** Information for system administrators and developers that contains end-to-end instructions for upgrading to Microsoft Dynamics AX 2012 from Microsoft Dynamics AX 4.0 and Microsoft Dynamics AX 2009. **Location:** *http://technet.microsoft.com/en-us/library/dd362002*
Code Upgrade Tool User Guide	**Description:** Information about how to install and use the Code Upgrade Tool, which helps developers upgrade X++ code to Microsoft Dynamics AX 2012. **Location:** *http://technet.microsoft.com/en-us/library/hh535215.aspx*
Security Upgrade Advisor Tool User Guide	**Description:** Information about how to install and use the Microsoft Dynamics AX Security Advisor Tool, which simplifies the process of upgrading security settings from earlier versions of Microsoft Dynamics AX. **Location:** *http://technet.microsoft.com/EN-US/library/hh394895*
Upgrade Support for Managed Code	**Description:** Information about code upgrade tools and resources. **Location:** *http://msdn.microsoft.com/en-us/library/gg889224.aspx*
Upgrading Reports	**Description:** Information and guidance for upgrading reports to Microsoft Dynamics AX 2012 **Location:** *http://msdn.microsoft.com/en-us/library/gg724124*
Code Upgrade White Papers	**Description:** Several downloadable white papers covering numerous topics related to code upgrades. **Location:** *http://www.microsoft.com/en-us/download/details.aspx?id=20864*
Resource Page for Upgrading to Microsoft Dynamics AX 2012	**Description:** Downloads, hotfixes, troubleshooting tips, and additional resources related to upgrading. **Location:** *http://community.dynamics.com/product/ax/axtechnical/b/axresources/archive/2012/02/17/resource-page-for-upgrading-to-microsoft-dynamics-ax-2012.aspx*

Index

Symbols and Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

About the authors

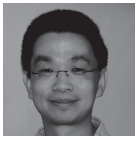
Principal authors

Anees Ansari is a program manager in the Microsoft Dynamics AX product group. His areas of focus include Enterprise Portal and web-based frameworks and clients for Microsoft Dynamics AX. He is passionate about web-based technologies and has been working in that area for more than 11 years, including 7 years at Microsoft.

Anees has a broad range of experience in various roles, both within and outside Microsoft. His last role was technical product manager in the Microsoft Web Platform and Standards group, where he was responsible for product management and marketing strategy for Microsoft ASP.NET and Microsoft Visual Web Developer products. Prior to that, he was a software developer on the Outlook Web App team working on Microsoft Exchange Online and Microsoft Exchange Server products. Before joining Microsoft, Anees worked with start-ups that helped small to mid-sized companies increase their online business and customer base by designing, developing, and managing their web portals.

Anees has a master's degree in computer science from the University of South Florida and a certificate in business fundamentals from the Kelley School of Business at Indiana University.

David Chell is a senior technical writer on the Service Industries and Retail Content Publishing team for Microsoft Dynamics AX.



Zhonghua Chu is a principal development lead on the Microsoft Dynamics AX server team. He has worked on data access and other server-related areas since Microsoft Dynamics AX 4.0. Zhonghua joined the Microsoft SQL Server Data Warehouse team after graduating from the University of Wisconsin–Madison, and has been working on application system design and implementation for over 13 years.

Dave Froslie is a principal test architect on the Microsoft Dynamics AX development team. He joined Microsoft in 2002, and has held a variety of development and test leadership roles in business solutions and development tools. Before joining Microsoft, Dave was a development manager for teams building engineering test systems at MTS Systems in Eden Prairie, Minnesota.

In his current role, Dave is responsible for providing guidance on test approaches for the product, with a focus on automated tools, libraries, and infrastructure. Dave also has a strong interest in engineering processes, particularly agile development. Blog posts that Dave has written on these and other topics can be found at *http://blogs.msdn.com/b/dave_froslie/*. Dave works in the Microsoft development office in Fargo, North Dakota, and lives across the river in Minnesota with his wife, Dawn, and daughter, Ali.

Chris Garty is a senior program manager on the Microsoft Dynamics AX Client Presentation team in Fargo, North Dakota. Chris joined the Microsoft Dynamics AX team during the Microsoft Dynamics AX 2009 development cycle. During the Microsoft Dynamics AX 2012 development cycle, Chris helped guide the changes to List Pages and Details forms, and worked on a range of user experience components. Chris's role on the Microsoft Dynamics AX team has recently expanded to cover integration with Microsoft Office and document management.

Chris has 13 years of experience in software development and consulting, the last 8 of which have been spent at Microsoft.

Chris was born and raised in New Zealand, and he is lucky enough to visit New Zealand and Australia almost yearly to see his family. He moved to Fargo to work for the best company in the world and lives there, eight winters and a couple of floods later, with his wife, Jolene. He spends his time away from Microsoft playing soccer, doing triathlons, running, and relaxing with friends and family as much as possible. Chris has a blog at *http://blogs.msdn.com/chrisgarty*.

Chary Gottumukkala joined Microsoft in 2002 as a software architect on the Microsoft Dynamics team, and he works on ERP/CRM frameworks and applications. Chary is passionate about developing software with the often intangible quality attributes, or "-ilities," such as scalability, maintainability, and extensibility. Prior to joining Microsoft, Chary worked on ERP/CRM frameworks and applications at Oracle and PeopleSoft, and on Professional Services Automation (PSA) applications at Niku. Chary has a BSc in electronics from Jawaharlal Nehru Technological University and an MSc in computer science from the Indian Institute of Technology.

Arthur Greef is a principal software architect who has a passion for developing innovative software that simplifies the lives of working people. Arthur has a BSc and an MSc in mechanical engineering from the University of Natal in South Africa, and a Ph.D. in industrial engineering from the University of Stellenbosch in South Africa. He also spent 2 years in an industrial engineering postdoctoral

program at the University of North Carolina in the United States. Arthur has been at Microsoft for 10 years, 3 of which were spent in Denmark working on Microsoft Dynamics AX when it was still called Axapta. Before joining Microsoft, Arthur worked for IBM in New York, developing product configuration technology for PCs. Arthur also spent two years working as chief architect for the RosettaNet Consortium, where he developed XML business collaboration protocols for the information technology industry.

Jakob Steen Hansen is a development manager, currently responsible for the development and architecture of developer tools, business intelligence, application life cycle, upgrade, and customization of Microsoft Dynamics AX. He joined Damgaard Data in 1993 while completing his MSc in computer science and electronic engineering, and contributed to the incubation of the product that later became Microsoft Dynamics AX. Throughout the releases, he has been involved in various aspects of the product, as well as in exploring how technology can bring previously unseen productivity or capabilities to partners and customers who develop solutions by using Microsoft Dynamics AX. For a few years, he worked on an incubation project in the Microsoft Developer Division, which eventually brought him back to the Microsoft Dynamics team.

Jakob worked in Denmark until 2008, when he moved to Seattle with his wife, Lone, and two teenage daughters, Louise and Ida Marie, to explore new facets of working at Microsoft, expanding his personal experience, and realizing new breakthroughs for Microsoft Dynamics and ERP development. He enjoys family life and the outdoors, and because he's an avid engineer, there's always a technical project cooking somewhere.

Kevin Honeyman played a key role in designing the changes to the Microsoft Dynamics AX 2012 user experience. Kevin has worked for Microsoft for 11 years, focusing on simplifying the user experience for various Microsoft Dynamics ERP products. Prior to working at Microsoft, Kevin worked at Great Plains Software, where he designed the developer user experience and user interface controls for the Great Plains Dynamics product. He started his career as a developer at Boeing Computer Services in Seattle, implementing a user interface control library in X Windows and Motif.

As a senior user experience lead at Microsoft, Kevin is passionate about understanding the user's needs and designing experiences that delight the user. He lives in Fargo, North Dakota, with his fiancée, Tiffanie, his son, Jordan, and his future stepchildren, Drue and Isabelle.

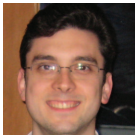
Michael Merz is a program manager for Microsoft Dynamics AX, where he is responsible for the delivery of the Microsoft Dynamics AX services framework and Microsoft Dynamics AX integration capabilities. He has over 15 years of experience in the software industry. Prior to working at Microsoft, Michael held various engineering and management positions in companies including Amazon.com, BEA Systems, and early-stage start-up companies, where he worked on embedded systems, online advertising, social networks, and enterprise software. He has an MSc in computer science from Ulm University, Germany, and lives in Bothell, WA, with his wife, Florina, and his children, Brooke and Joshua.

Amar Nalla is currently a development lead in the Microsoft Dynamics AX product group. He has more than 11 years of experience in the software industry. He started working on the Microsoft Dynamics team during the Axapta 4.0 release. He is part of the foundation team responsible for the Microsoft Dynamics AX server components, and during the past three releases of Microsoft Dynamics AX, he has worked on various components of the server. He maintains a blog at *http://blogs.msdn.com/b/amarnalla/.*

In his spare time, Amar likes to explore the beautiful Puget Sound area.

Parth Pandya is a senior program manager in the Microsoft Dynamics AX product group. For Microsoft Dynamics AX 2012, Parth's area of focus was the new security framework that was built for the release, including the flexible authentication capability and support for Active Directory groups as Microsoft Dynamics AX users. He also contributed to the named user licensing model that was instituted for Microsoft Dynamics AX 2012. Parth has been with Microsoft for over nine years, over five of which were spent working on various releases of the Windows Internet Explorer browser. He particularly enjoyed working as a penetration tester for the number one target of hackers around the world.

Parth swapped the organized chaos of Mumbai, India, for the disorienting tranquility of the Pacific Northwest, where he lives with his wife, Varsha, and three-year-old son, Aarush.



Gustavo Plancarte is a senior software design engineer who joined Microsoft in 2004 after graduating from ITESM in Monterrey, Mexico. He has worked on Microsoft Dynamics AX since version 4.0. On the platform team, he is responsible for driving the common intermediate language (CIL) migration of the X++ programming language, the Software-plus-Services architecture of

the application server, and the batch framework. Gustavo has filed several software-related patents, in areas including garbage collection, incremental generation of assemblies, and batch scheduling and processing. He lives with his wife, Gina, and their sons, Gustavo Jr. and Luis, in Woodinville, WA, where he enjoys spending time working on his yard.

Michael Fruergaard Pontoppidan joined Damgaard Data (which merged with Navision and was eventually acquired by Microsoft) in 1996, after graduating from the Technical University of Denmark. He started as a software design engineer on the MorphX team, delivering the developer experience for the first release of Microsoft Dynamics AX. Today, he is a software architect on the Microsoft Dynamics AX team in Copenhagen. For Microsoft Dynamics AX 2012, Michael primarily focused on the metadata model store, solving problems related to element IDs and the MorphX Development Workspace. In previous releases, he has worked on version control, unit testing, best practices, and the Microsoft Trustworthy Computing initiative, while advocating for code quality improvements through Microsoft Engineering Excellence, tools, processes, and training. Michael frequently appears as a speaker at technical conferences. He lives in Denmark with his wife, Katrine, and their two children, Laura and Malte. His blog is at *http://blogs.msdn.com/mfp*.

Bigyan Rajbhandari is a program manager on the Microsoft Dynamics AX team, working on the security, licensing, and batch framework areas. He has more than seven years of experience in software engineering, consulting, and management, the last four of which have been spent at Microsoft. Prior to his current role, he helped develop a large customer preference management system for Microsoft. He graduated from Drake University in Iowa with a BS in computer science and went on to work for various companies in the Midwest, building custom business applications and customer relationship management (CRM) solutions. Outside of work, he enjoys traveling, hiking, and soccer. He currently lives in Redmond, Washington, with his wife, Jashmin.

Karl Tolgu is a senior program manager for Microsoft Dynamics AX. He is responsible for the development of the business process, workflow, and alert notification framework. Previously, Karl worked on the project accounting modules in Microsoft Dynamics SL and Microsoft Dynamics GP. Since graduating, he has worked in the software industry in both the United Kingdom and the United States and held various software development management positions at Oracle Corporation and Niku Corporation. Karl resides in Seattle with his wife, Karin, and three sons, Karl Christian, Sten Alexander, and Thomas Sebastian.

TJ Vassar has worked in development on various projects at Microsoft for over 13 years, including Microsoft Money, MSN Money, Microsoft Office Accounting, and Microsoft Dynamics AX 2009. Currently, he is a senior program manager on the Microsoft Dynamics AX Business Intelligence team, managing the Reporting framework. Born and raised in Seattle, TJ is married to the woman of his dreams and is a proud father of three. He regularly posts to his MSDN blog (*http://blogs.msnd.com/dynamicsaxbi*) on topics that range from basic development principles to alternate methods of visualizing business insight by using the Reporting framework.

Peter Villadsen is a senior program manager on the Microsoft Dynamics AX X++ language team, developing and maintaining the X++ language stack. After earning his MS in electrical engineering, he started his career by building Ada compilers but quickly became interested in ERP systems, helping to build one for the Apple Macintosh before joining Damgaard Data. There, he helped design and build the first version of what later became the Microsoft Dynamics AX system.

Peter currently lives in Seattle with his wife, Hanne. When not behind the monitor, he enjoys a good game of badminton.

Milinda Vitharana is a senior program manager on the Microsoft Dynamics AX Business Intelligence (BI) team in Redmond, WA. His area of focus is the online analytical framework (OLAP) framework in Microsoft Dynamics AX. Before joining the team in 2008, Milinda spent over 12 years designing, developing, and implementing business intelligence solutions in various industries, including life insurance, financial services, real estate, education, justice, and transportation.

Milinda is passionate about applying BI to solve business problems. He started his career working for an independent software vendor (ISV) developing software solutions in the financial services industry. He then joined a large life insurance company, where he implemented BI solutions. Before joining Microsoft, he worked for a large systems integrator as a BI specialist and consultant. Having seen many applications of BI as an ISV, customer, and partner, he is happy to finally be at the SYS layer.

Milinda is a software engineer and has an MBA in finance. He lives in the greater Seattle area with his wife and two kids.

Christian Wolf is solutions architect and program manager, and is a member of the team that is responsible for the performance and scalability of Microsoft Dynamics AX. Before joining the Microsoft Dynamics AX core development team, he worked as a support, premier field, and escalation engineer, collecting field experience about performance and scalability. He lives in Bellevue, WA, and in his spare time, he enjoys cycling, running, and hiking. Christian's team members maintain a blog about performance and scalability issues at *http://blogs.msdn.com/b/axperf/*.

Contributing authors

Jeff Anderson is a senior software design engineer who joined Microsoft in 2002 after graduating from North Dakota State University in Fargo, North Dakota. He has worked on a variety of Microsoft ERP products, including Microsoft Dynamics GP, Microsoft Dynamics NAV, and Microsoft Dynamics AX. He has been working on Microsoft Dynamics AX since the 2009 release, in the area of global financial management and application performance. He lives in West Fargo, North Dakota, with his wife, Jennifer, and their sons, Nate and Joe.

Wade Baird is a senior software design engineer on the Microsoft Dynamics AX Client Presentation team in Fargo, North Dakota. Wade joined Microsoft in 2001, while completing his final year at North Dakota State University. Since then, he has worked on a variety of Object Relational Mapping (ORM) products, and he began working on Microsoft Dynamics AX for the 2009 release. Since then, he has focused on all of the aspects of the client forms subsystem.

Arijit Basu is a senior solutions architect on the Solutions Architecture team for Microsoft Business Solutions.

Michael Gall is a senior software development engineer on the Microsoft Dynamics AX Costing team at Microsoft Development Center Copenhagen. He joined Microsoft in 2007. Before joining Microsoft, Michael worked with a Microsoft Dynamics AX partner as a solution architect and software development manager, implementing Microsoft Dynamics AX projects in various industries. During the development of Microsoft Dynamics AX 2012, he worked on the lean costing solution and the source document and accounting frameworks. Michael has a Ph.D. in computer science, four master's degrees from the Technical University of Vienna, and an MBA from Copenhagen Business School. Professionally, he is passionate about software architecture and ERP system architecture. He lives in Copenhagen with his children, Laura and Nico. In his spare time, he likes traveling and outdoor activities with his kids.

John Healy is a principal software architect in the Microsoft Dynamics AX product group that focuses on global financial management. He is responsible for the overall vision and adoption of the architecture for global financials and works with a range of Microsoft Dynamics AX architects and technical leaders to ensure consistent direction and adoption across the Microsoft Dynamics AX applications. He has over 32 years of experience in accounting, supply chain, and manufacturing application development. He has worked in a variety of technical and leadership roles. He joined Microsoft in 2001 through the Great Plains Software acquisition. He is a graduate of the University of Minnesota, Twin Cities. He lives in Lake Elmo, Minnesota, with his wife, Jackie.

John is an editor and regular contributor to the Microsoft Dynamics AX Global Financial Management team blog at *http://blogs.msdn.com/b/ax_gfm_framework_team_blog/.*

Vanya Kashperuk is a senior software development engineer in Test on the Supply Chain Management team at Microsoft Development Center Copenhagen. He has been working on Microsoft Dynamics AX since 2004 and has been at Microsoft in Denmark for the last four years. Vanya writes a blog about Microsoft Dynamics AX, which you can read at *http://www.kashperuk.blogspot.com*.

Vanya has a master's degree in computer science from the National Technical University of Ukraine, which is also where he met his wife, Valeriia.

Outside of work, Vanya enjoys all kinds of team sports, sightseeing in countries around the world, photography (as part of his sightseeing trips), and computer games, especially the new Kinect Sports!

Vanya and his wife live in a peaceful area of Charlottenlund, just north of Copenhagen.

Ievgenii Korovin is a software design engineer on the Supply Chain Management team at Microsoft Development Center Copenhagen. He has been working on Microsoft Dynamics AX since 2006, and he is mainly responsible for the architecture and functionality within the Inventory management, Warehouse management, and Product information management areas. Ievgenii has a master's degree in computer science from the National Technical University of Ukraine. He lives in Copenhagen with his wife, Tamara, and their young daughter, Alisa. In their free time, he and his family enjoy outdoor activities, especially skiing, biking, and hiking.

Ievgenii writes a blog about Microsoft Dynamics AX, which you can read at *http://blogs.msdn.com/dynamicsaxscm*.

Maciej Plaza is a software development engineer in Test on the Microsoft Dynamics AX Inventory team. He received an MSc from Poznan University of Technology, and during his time at the university, he actively sought out interesting algorithmic problems, engaging in research projects in cooperation with partners from both industry (Volkswagen) and academia (University

of Nottingham). After graduating in 2007, he started working as Software Development Engineer for Microsoft SQL Server, tackling the challenges of storing unstructured data, working on FILESTREAM, Remote BLOB Storage, and FileTable features. After almost three years, he decided to move back to Europe and joined the Microsoft Dynamics AX team. For the Microsoft Dynamics AX 2012 release, his primary focus was ensuring the quality of the Product information management functionality. Driven by his passion for quality, he also got involved in improving the test tools and the applied processes, to ensure that even higher quality can be achieved in the future.

Maciej lives in Copenhagen with his wife, Anna. In his spare time, besides spending time with his family, he enjoys pursuing his interests in photography and music.

Anders Tind Sørensen joined Microsoft in 2006 as a software development engineer for the Manufacturing team. He has focused primarily on discrete production, master planning, and the resource scheduling engine, but has also been deeply involved in the integration of the Process manufacturing industry module. He has 10 years of ERP development and implementation experience, and is proud to be a geek. Anders lives in Denmark with his girlfriend, Nena.

Manoj Swaminathan is a principal development manager based in Redmond, WA. He joined Microsoft 4 years ago, after working at Oracle, and has spent more than 12 years working on ERP application development for financials. He was responsible for leading global financial management development for the financial foundation, such as the source document and accounting frameworks, multiple ledgers, and globalization/localization of Microsoft Dynamics AX 2012. Currently, he is leading efforts to drive application life cycle management for Microsoft Dynamics AX, RapidStart Services, and setup/deployment initiatives for online and on-premise solutions.

Robin Van Steenburgh joined the Microsoft Dynamics AX team in 2005; she currently enjoys creating developer samples and documentation with the software development kit (SDK) team based in Redmond. After graduating from the University of Toronto, Robin worked as a software developer for several oil companies and a software startup called Sierra Geophysics. Robin joined Microsoft in 1997 and has worked on teams delivering MSN, Site Server, Commerce Server, and Microsoft Learning products. Her favorite role prior to joining Microsoft Dynamics AX was as an acquisitions editor for Microsoft Press. She is a Microsoft Certified Technology Specialist for Microsoft Dynamics AX 2009 and Microsoft Dynamics AX 2012. Robin was responsible for the developer documentation for services and the Application Integration Framework (AIF) on MSDN for Microsoft Dynamics AX 4.0 and Microsoft Dynamics AX 2012. She also maintains the MSDN Developer Center for Microsoft Dynamics AX, and occasionally blogs at *http://blogs.msdn.com/b/aif/*. In her free time, she takes ballet classes and supports the Seattle Sounders.

What do you think of this book?

We want to hear from you!

To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Tell us how well this book meets your needs—what works effectively, and what we can do better. Your feedback will help us continually improve our books and learning resources for you.

Thank you in advance for your input!

Microsoft® Press