PART I

A tour of the development environment

CHAPTER 1

Architectural overview

In this chapter

Introduction

The Microsoft Dynamics AX solution is an enterprise resource planning (ERP) solution that integrates financial resource management, operations resource management, and human resource management processes that can be owned and controlled by multinational, multi-company, and multi-industry organizations, including the public sector. The Microsoft Dynamics AX solution encompasses both the Microsoft Dynamics AX application and the Microsoft Dynamics AX application platform on which it is built. The Microsoft Dynamics AX application platform is designed to be the platform of choice for developing scalable, customizable, and extensible ERP applications in the shortest time possible, and for the lowest cost. The following key architectural design principles make this possible.

- **Separation of concerns** A Microsoft Dynamics AX end-to-end solution is delivered by many development teams working both inside Microsoft, inside the Microsoft partner channel, and inside end-user IT support organizations. The separation of concerns principle realized in the Microsoft Dynamics AX architecture makes this possible by separating the functional concerns of a solution into five globalized, secure layers. This separation reduces functional overlap between the logical components that each team designs and develops.
- **Separation of processes** A Microsoft Dynamics AX end-to-end solution scales to satisfy the processing demands of a large number of concurrent users. The separation of processes principle that is realized in the Microsoft Dynamics AX architecture makes this possible by separating processing into three-tiers—a data tier, a middle tier, and a presentation tier. The Microsoft Dynamics AX Windows client, the Microsoft Dynamics AX Enterprise Portal web client, and the Microsoft Office clients are components of the presentation tier; the Microsoft Dynamics AX Application Object Server (AOS), the Microsoft Dynamics AX Enterprise Portal extensions to Microsoft SharePoint Server, and Microsoft SQL Server Reporting Services (SSRS)

are components of the middle tier; the SQL Server and Microsoft SQL Server Analysis Services (SSAS) are components of the data tier of the Microsoft Dynamics AX platform architecture.

- **Model-driven applications** A Microsoft Dynamics AX application team can satisfy application domain requirements in the shortest time possible. The model-driven application principle that is realized in the Microsoft Dynamics AX architecture makes this possible by separating platform-independent development from platform-dependent development, and by separating organization-independent development from organization-dependent development. With platform-independent development, you can model the structure and specify the behavior of application client forms and reports, of application object entities, and of application data entities that run on multiple platform technologies such as the Microsoft Dynamics AX Windows client, SharePoint Server, SQL Server, and the Microsoft .NET Framework. With organization-independent development, you can use domain-specific reference models, such as the units of measure reference model; domain-specific resource-models, such as the person, product, and location models; and domain-specific workflow models, such as approval and review models, which are relevant to all organizations.

Microsoft Dynamics AX five-layer solution architecture

The Microsoft Dynamics AX five-layer solution architecture, illustrated in Figure 1-1, logically partitions a Microsoft Dynamics AX solution into an application platform layer, a foundation application domain layer, a horizontal application domain layer, an industry application domain layer, and a vertical application domain layer. The components in all architecture layers are designed to meet Microsoft internationalization, localization, and security standards, and all layers are built on the Microsoft technology platform.

Note The layers in the Microsoft Dynamics AX five-layer architecture are different from the model layers that are part of the Microsoft Dynamics AX customization framework described later in this book. Architectural layers are logical partitions of an end-to-end solution. Customization layers are physical partitions of application domain code. For more information, see Chapter 21, "Application models."

The Microsoft Dynamics AX application platform and application domain components are delivered on the Microsoft technology platform. This platform consists of the Windows client, the Office suite of products, Windows Server, SQL Server, SSAS, SSRS, SharePoint Server, the Microsoft ASP.NET web application framework, the .NET Framework, and the Microsoft Visual Studio integrated development environment (IDE).

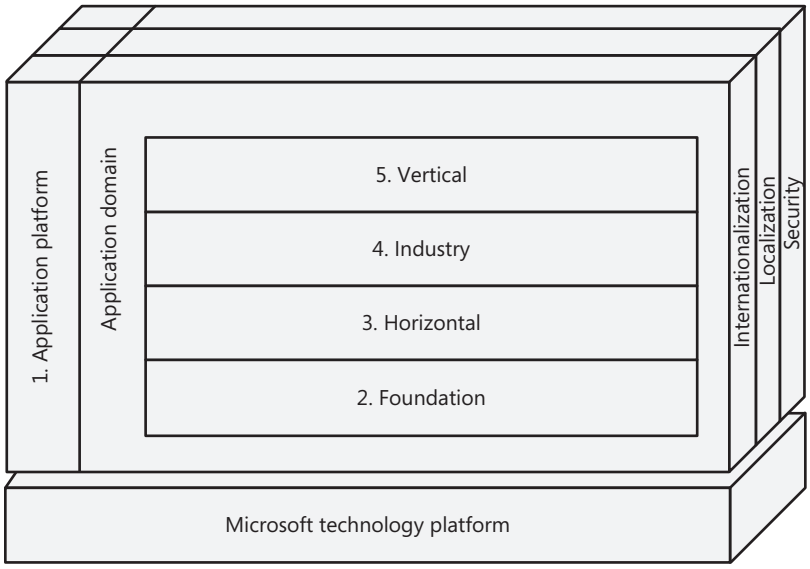

FIGURE 1-1 Microsoft Dynamics AX five-layer architecture.

The following logical partitions are layered on top of the Microsoft technology platform:

- **Layer 1: Application platform layer** The application platform layer provides the system frameworks and tools that support the development of scalable, customizable, and extensible application domain components. This layer consists of the MorphX model-based development environment, the X++ programming language, the Microsoft Dynamics AX Windows client framework, the Enterprise Portal web application framework, the AOS, and the application platform system framework. The architecture of the components in the application platform layer is described in the following section.
- **Layer 2: Foundation application domain layer** The foundation application domain layer consists of domain-specific reference models in addition to domain-specific resource modeling, policy modeling, event documenting, and document processing frameworks that are extended into organization administration and operational domains. Examples of domain-specific reference models include the fiscal calendar, the operations calendar, the language code, and the unit of measure reference models. Examples of domain-specific resource models include the party model, the organization model, the operations resource model, the product model, and the location model. The source document framework and the accounting distribution and journalizing process frameworks are also part of this layer. Chapter 19, "Application frameworks," describes the conceptual design of a number of the frameworks in this layer.

- **Layer 3: Horizontal application domain layer** The horizontal application layer consists of application domain workloads that integrate the financial resource, operations resource, and human resource management processes that can be owned and controlled by organizations. Example workloads include the operations management workload, the supply chain management workload, the supplier relationship management workload, the product information management workload, the financial management workload, the customer relationship management workload, and the human capital management workload. The Microsoft Dynamics AX application can be extended with additional workloads. (The workloads that are part of the Microsoft Dynamics AX solution are beyond the scope of this book.)
- **Layer 4: Industry application domain** The industry application layer consists of application domain workloads that integrate the financial resource, operations resource, and human resource management processes that are specific to organizations that operate in particular industry sectors. Example industries include discrete manufacturing, process manufacturing, distribution, retail, service, and public sector. Workloads in this layer are customized to satisfy industry-specific requirements.
- **Layer 5: Vertical application domain** The vertical application layer consists of application domain workloads that integrate the financial resource, operations resource, and human resource management processes that are specific to organizations that operate in a particular vertical industry and to organizations that are subject to local customs and regulations. Example vertical industries include beer and wine manufacturing, automobile manufacturing, government, and advertising professional services. Workloads in this layer are customized to satisfy vertical industry and localization requirements.

Microsoft Dynamics AX application platform architecture

The architecture of the Microsoft Dynamics AX application platform supports the development of Windows client applications, SharePoint web client applications, Office client integration applications, and third-party integration applications. Figure 1-2 shows the components that support these application configurations. This section provides a brief description of the application development environments, and a description of the components in each of the data, middle, and presentation tiers of the Microsoft Dynamics AX platform architecture.

Application development environments

The Microsoft Dynamics AX application platform includes two model-driven application development environments:

- **Microsoft Dynamics AX MorphX development environment** Use this development environment to develop data models and application code using the Application Object Tree (AOT) application modeling tool and the X++ programming language. This development environment accesses Microsoft Dynamics AX application server services through Microsoft Remote Procedure Call (RPC) technology.

- **Visual Studio** Use this development environment to develop Microsoft .NET plug-ins for and extensions to Microsoft Dynamics AX clients, servers, and services; to develop for Enterprise Portal, and to develop SSRS reports. This development environment accesses the Microsoft Dynamics AX application server services through RPC.

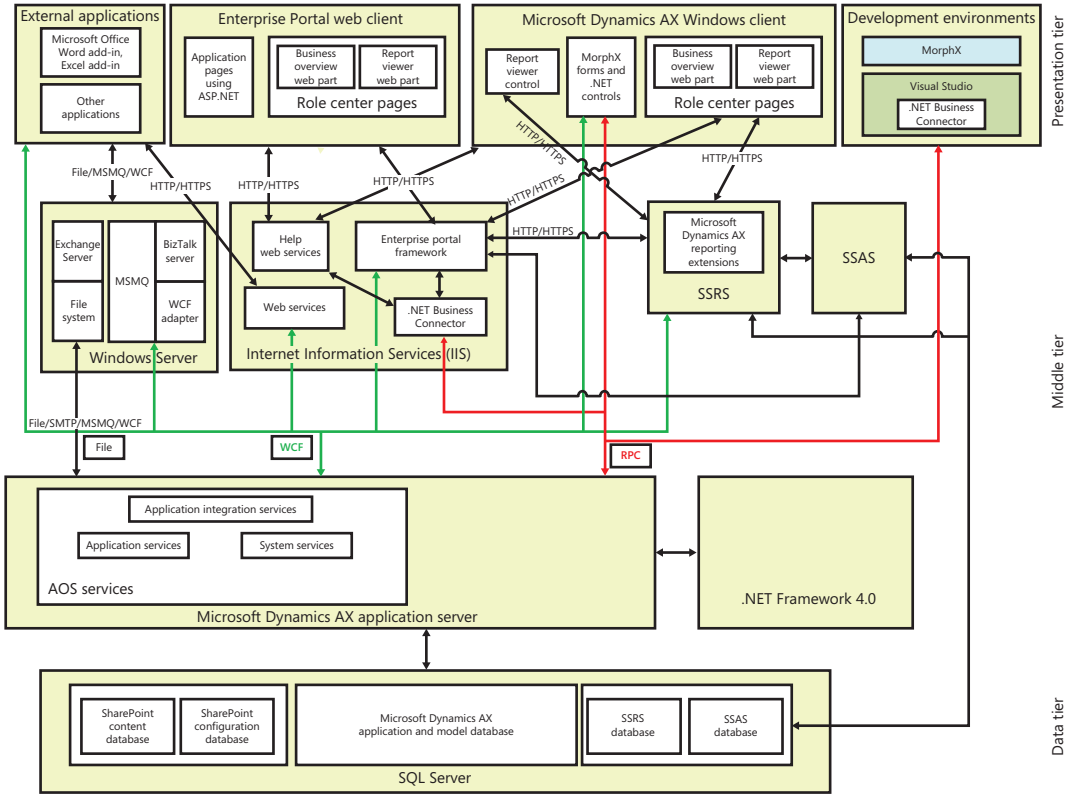

FIGURE 1-2 Architecture of Microsoft Dynamics AX.

Data tier of the Microsoft Dynamics AX platform

The SQL Server database is the only component in the data tier. The database server hosts the SharePoint Server content and configuration databases, the Microsoft Dynamics AX model and application database, the SSRS database, and the SSAS database.

Middle tier of the Microsoft Dynamics AX platform

The middle tier includes the following components:

- **AOS** The AOS executes MorphX application services that are invoked through RPC technology and Windows Communication Foundation (WCF) technology in the .NET Framework. The AOS can be hosted on one computer, but it can also scale out to many computers when additional concurrent user sessions or dedicated batch servers are required.

- **.NET Framework components** These components can be referenced in the AOT so that their application programming interfaces are accessed from X++ programs. The Windows Workflow Foundation (WWF) component is integral to the Microsoft Dynamics AX workflow framework, and WCF is integral to the Microsoft Dynamics AX application integration framework.
- **SQL Server Analysis Services (SSAS)** These services process requests for analytics data hosted by the SQL Server component in the data tier.
- **SSRS and Microsoft Dynamics AX reporting extensions** The reporting extensions provide SSRS with features that are specific to the Microsoft Dynamics AX application platform. These extensions access the AOS through WCF services and access SSAS through HTTP and HTTPS.
- **Microsoft Dynamics AX Enterprise Portal framework** This framework extends the SharePoint application platform with features that are specific to the Microsoft Dynamics AX application platform. The Enterprise Portal framework composes SharePoint content with Microsoft Dynamics AX content accessed from the AOS through the .NET Business Connector and RPC, and content accessed from SSAS and SSRS through HTTP and HTTPS. Enterprise Portal is typically hosted on its own server or in a cluster of servers.
- **Microsoft Dynamics AX Help web service** This web service processes requests for Help content.
- **Web services hosted by Microsoft Internet Information Services (IIS)** The Microsoft Dynamics AX system services can be deployed to and hosted by IIS.
- **Application Integration services** These services provide durable message queuing and transformation services for integration clients.

Presentation tier of the Microsoft Dynamics AX platform

The presentation tier consists of the following components:

- **Windows client** This client executes Microsoft Dynamics AX MorphX and .NET programs developed in MorphX and Visual Studio. The client application communicates with the AOS primarily by using RPC. The client composes navigation, action pane, area page, and form controls for rapid data entry and data retrieval. Form controls have built-in data filtering and search capabilities and their content controls are arranged automatically by the Intellimorph rendering technology. The client additionally hosts role center pages rendered in a web browser control.
- **Enterprise Portal web client** This client executes MorphX application models, X++ programs, and .NET Framework programs developed in the MorphX development environment, Visual Studio, and the SharePoint Server framework. Enterprise Portal is hosted by the Microsoft Dynamics AX runtime, the ASP.NET runtime, and the SharePoint runtime environments. SharePoint and ASP.NET components communicate by means of the Microsoft Dynamics AX .NET Business Connector.

- **Office clients** The Microsoft Word client and Microsoft Excel client are extended by add-ins that work with the Microsoft Dynamics AX platform.
- **Third-party clients** These clients integrate with the Microsoft Dynamics AX platform by means of integration service components such as the file system, Microsoft Message Queuing (MSMQ), Microsoft BizTalk Server, and a WCF adaptor.

Microsoft Dynamics AX application meta-model architecture

Microsoft Dynamics AX application meta-model architecture is based on the principle of model-driven application development. You declaratively program an application by building a model of application components instead of procedurally specifying their structure and behavior with code. The Microsoft Dynamics AX development environment supports both model-driven and code-driven application development.

A model of an application model is called a *meta-model*. Figure 1-3 shows the element types in the Microsoft Dynamics AX application meta-model that you use to develop Microsoft Dynamics AX Windows client applications.

Note To keep the diagram simple, the figure does not list all type dependencies on model element types.

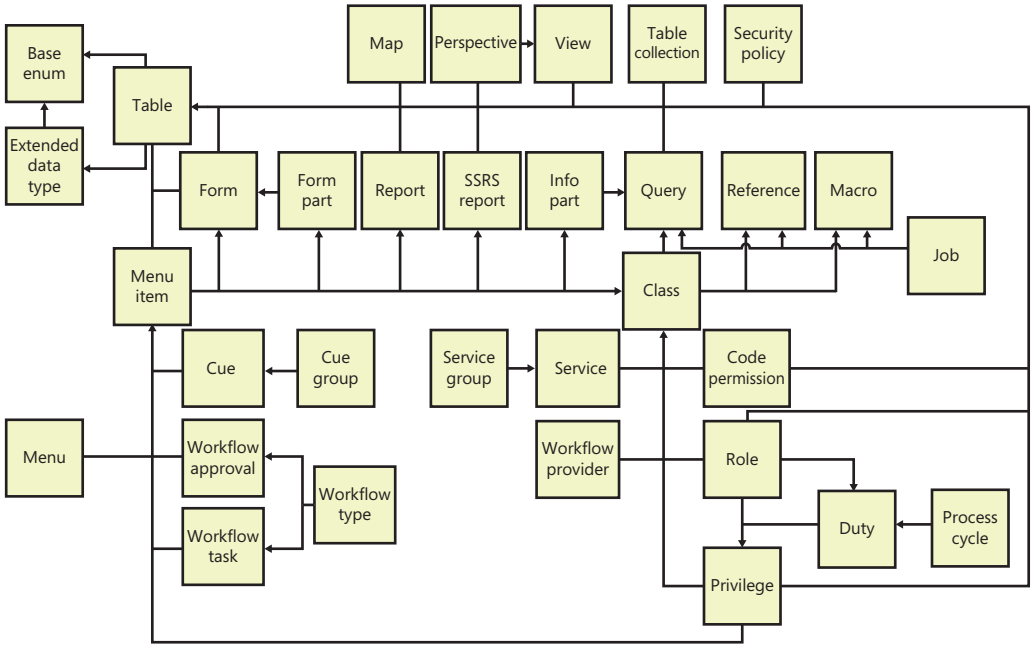

FIGURE 1-3 Element types of the Microsoft Dynamics AX meta-model for developing Microsoft Dynamics AX Windows client applications.

Application data element types

The following model element types are part of the Microsoft Dynamics AX application data meta-model:

- **Base enum** Use a base enumeration (base enum) element type to specify value type application model elements whose fields consist of a fixed set of symbolic constants. For example, you can create a base enum named *WeekDay* to name a set of symbolic constants that includes Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, and Saturday.

- **Extended data type** Use an extended data type element type to specify value type application model elements that extend base enums, in addition to *string, boolean, integer, real, date, time, UtcDateTime, int64, guid,* and *container* value types. The Microsoft Dynamics AX runtime uses the properties of an extended data type to generate a database schema and to render user interface controls. For example, you could specify an *account number* extended data type as an extension to a *string* value type that is limited to 10 characters in length, and that is described using the *Account number* label when bound to a user interface text entry control. Extended data types also support inheritance. For example, an extended data type that defines an account number can be specialized by other extended data types to define customer and vendor account numbers. The specialized extended data type inherits properties, such as string length, label text, and help text. You can override some of the properties on the specialized extended data type.

- **Table** Use a table element type to specify data entity types that the Microsoft Dynamics AX application platform uses to generate a SQL Server database table schema. Tables specify data entity type fields along with their base enum or extended data type, field groups, indexes, relationships, delete actions, and methods. Tables can also inherit the fields of base tables that they are specified to extend. The Microsoft Dynamics AX runtime uses table specifications to render data entry presentation controls and to maintain the referential integrity of the data stored in the application database. The X++ editor also uses table elements to provide IntelliSense information when you write X++ code that manipulates data stored in the application database. Tables can be bound to form, report, query, and view data sources.

- **Map** Use a map element type to specify a data entity type that factors out common table fields and methods for accessing data stored in horizontally partitioned tables. For example, the *CustTable* and *VendTable* tables in the Microsoft Dynamics AX application model are mapped to the *DirPartyMap* map element so that you can use one *DirPartyMap* object to access common address fields and methods.

Note Consider table inheritance as an alternative to using maps because it increases the referential integrity of a database when base tables are referenced in table relationships.

- **View** Use a view element type to specify a database query that the Microsoft Dynamics AX application platform uses to generate a SQL Server database view schema. Views can include

a query model element that filters data accessed from one table or from multiple joined tables. Views also include table field mappings and methods. Views are read-only and primarily provide an efficient method for reading data. Views can be bound to form, report, and query data sources.

- **Perspective** Use a perspective element type to specify a group of tables and views that are used together when designing and generating SSAS unified dimensional models.
- **Table collection** Use a table collection element type to specify a group of tables whose data is shared by two or more Microsoft Dynamics AX companies assigned to the same virtual company. An application administrator maintains virtual companies, their effective company assignments, and their table collection assignments. The Microsoft Dynamics AX runtime uses the virtual company data area identifier instead of the effective company data area identifier to securely access data stored in tables grouped by a table collection.

Caution The tables in a table collection should only reference tables inside the table collection unless you write application extensions to maintain the referential integrity of the database.

- **Query** Use a query element type to specify a database query. You add tables to query element data sources and specify how they should be joined. You also specify how data is returned from the query by using sort order and range specifications.

MorphX user interface control element types

The following model element types are part of the Microsoft Dynamics AX MorphX user interface control meta-model:

- **Menu item** Use a menu item element type to specify presentation control actions that change the state of the Microsoft Dynamics AX system or user interface or that generate reports. If you specify a label for the menu item, the Microsoft Dynamics AX runtime uses it to name the action when it is rendered in the user interface. The Microsoft Dynamics AX form engine also automatically adds a View details menu item to a drop-down menu, a menu that appears when a user right-clicks a cell in a column that is bound to a table field that is specified as a foreign key in a table relationship. The Microsoft Dynamics AX runtime uses the referenced table's menu item binding to open the form that renders the data from the table. The Microsoft Dynamics AX form and report rendering engines ignore menu items that are disabled by configuration keys or role-based access controls.
- **Menu** Use a menu element type to specify a logical grouping of menu items. Menu specifications can also group submenus. The menu element named *MainMenu* specifies the menu grouping for the Microsoft Dynamics AX navigation pane.

- **Form** Use a form element type to specify a presentation control that a user uses to insert, update, and read data stored in the application database. A form binds table, view, and query data sources to presentation controls. A form is opened when a user selects a control bound to a menu item, such as a button.
- **Form part** Use a form part element type to specify a presentation control that renders a form in the FactBox area of the user interface. For more information about the FactBox area, see Chapter 6, "Designing the user experience."
- **Info part** Use an info part element type to specify a presentation control that renders the result set of a query in the FactBox area of the user interface.
- **Report** Use a report element type to specify a presentation control that renders database data and calculated data in a page-layout format. A user can send a report to the screen, a printer, a printer archive, an email account, or the file system. A report specification binds data sources to presentation controls. A report is opened when a user clicks an output menu item control, such as a button.
- **SSRS report** Use an SSRS report element type to reference a Visual Studio Report Project that is added to the Microsoft Dynamics AX model database.
- **Cue** Use a cue element type to bind a menu item to a presentation control that renders a pictorial representation of a numeric metric, such as the number of open sales orders. A cue is rendered in a Microsoft Dynamics AX Role Center webpage.
- **Cue group** Use a cue group element type to specify a group of cues that are displayed together on the Microsoft Dynamics AX Role Center web part.

Workflow element types

Workflow element types define the workflow tasks, such as review and approval, by binding the tasks to menu items. When a form is workflow-enabled, it automatically renders controls that support the user in performing the tasks in the workflow. Workflow elements define workflow documents and event handlers by using class elements. The following model element types are part of the Microsoft Dynamics AX workflow meta-model:

- **Workflow type** Use a workflow type element type to specify a workflow for processing workflow documents. A workflow configuration consists of event handler specifications, custom workflow task specifications, and menu item bindings.
- **Workflow task** Use a workflow task element type to specify a workflow task. A workflow task comprises a list of task outcomes, event handler registrations, and menu item bindings.
- **Workflow approval** Use a workflow approval element type to specify specialized workflow approval tasks. A workflow approval task consists of approve, reject, request change, and deny task outcomes, a list of event handler registrations, and menu item bindings.

- **Workflow provider** Use a workflow provider element type to specify the name of a class that provides data to a workflow. Example data includes a list of workflow participants, a list of task completion dates, and a structure of users that reflect positions in a position-reporting hierarchy.

Code element types

The following model element types are part of the Microsoft Dynamics AX code meta-model:

- **Class** Use a class element type to specify the structure and behavior of custom X++ types that implement data maintenance, data tracking, and data processing logic in a Microsoft Dynamics AX application. You specify class declarations, methods, and event handlers by using the X++ programming language. Class methods can be bound to menu items so that they are executed when users select action, display, or output menu item controls on a user interface. You can also use a class model element type to specify class interfaces that only include method definitions.
- **Macro** Use a macro element type to specify a library of X++ syntax replacement procedures that map X++ input character sequences, such as readable names, to output character sequences, such as numeric constants, during compilation.
- **Reference** Use a reference element type to specify the name of a. NET Framework assembly that contains .NET Framework common language runtime (CLR) types that can be referenced in X++ source code. The MorphX editor reads type data from the referenced assemblies so that IntelliSense is available for CLR namespaces, types, and type members. The MorphX compiler uses the CLR type definitions in the referenced assembly for type and member syntax validation, and the Microsoft Dynamics AX runtime uses the reference elements to locate and load the referenced assembly.
- **Job** Use a job element type to specify an X++ program that runs when you select the Command\Go menu item or press F5. Developers often write jobs when experimenting with X++ language features. You should not use jobs to write application code.

Services element types

The following model element types are part of the Microsoft Dynamics AX services meta-model:

- **Service** Use a service element type to enable an X++ class to be made available on an integration port.
- **Service group** Use a service group element type to specify a web service deployment configuration that exposes web service operations as basic ports with web addresses.

Role-based security element types

The following model element types are part of the Microsoft Dynamics AX role-based access control security meta-model:

- **Security policy** Use a security policy element type to specify a configuration for constraining the view that a user has of data stored in one or more tables. A security policy configuration consists of a primary table specification and a policy query.
- **Code permission** Use a code permission element type to specify one or more access permissions that secure access to logical units of application data and functionality. You can specify data access permissions to secure access to data stored in tables. You can specify code access permissions to secure access to forms, web controls, and server methods.
- **Privilege** Use a privilege element type to specify one or more permissions that a user requires to perform a task, such as a data maintenance task; or a step in a task, such as a data view or data deletion step.
- **Duty** Use a duty element type to specify a set of privileges that are required for a user to carry out his or her internal control approval, review, and inquiry responsibilities and data maintenance responsibilities.
- **Role** Use a role element type to specify the organization role, functional role, or application role that a user is assigned to in an organization. Sales agent is an example of an organization role, manager is an example of a functional role, and system user role is an example of an application role.
- **Process cycle** Use a process cycle element type to specify the operations and administration activities that are repetitively performed by users who are assigned duties in the security model. The expenditure cycle, the revenue cycle, the conversion cycle, and the accounting cycle are examples of process cycles.

Web client element types

The elements of the Microsoft Dynamics AX application meta-model that are used to develop Enterprise Portal web client applications are illustrated in Figure 1-4.

The following model element types are part of the Microsoft Dynamics AX web client meta-model:

- **Web menu item** Use a menu item element type to specify web navigation actions that change the state of the Microsoft Dynamics AX system or user interface. If a label is specified for the menu item, the Microsoft Dynamics AX runtime will use it to name the action when it is rendered in the user interface.
- **Web menu** Use a web menu element type to specify a logical grouping of web menu items. Web menu specifications can group submenus. Web menus are rendered as hyperlinks on webpages.

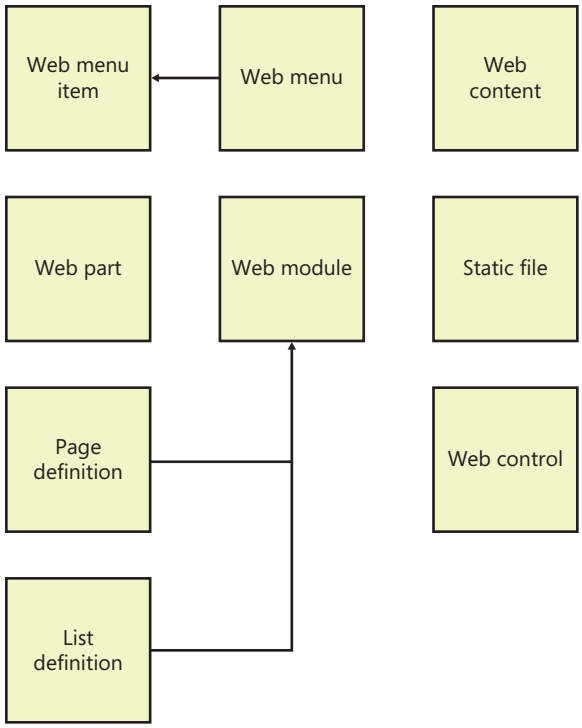

FIGURE 1-4 Element types of the Microsoft Dynamics AX meta-model for developing Enterprise Portal web applications.

- **Web content** Use a web content element type to reference an ASP.NET user control. ASP.NET user controls are developed in the Visual Studio IDE and are stored in the Microsoft Dynamics AX model database.
- **Web part** Use a web part element type to store a SharePoint web part in the Microsoft Dynamics AX model database. The web part will be saved to a web server when deployed.
- **Page definition** Use a page definition element type to store a SharePoint webpage in the Microsoft Dynamics AX model database. The page definition will be saved to a web server when deployed.
- **Web control** Use a web control element type to store an ASP.NET user control in the Microsoft Dynamics AX model database. The web controls will be saved to a web server when deployed.
- **List definition** Use a list definition element type to store a SharePoint list definition in the Microsoft Dynamics AX model database. The list definition will be created on a SharePoint server when deployed.
- **Static file** Use a static file element type to store a file in the Microsoft Dynamics AX model database. The file will be saved to a SharePoint server when deployed.
- **Web module** Use a web module element type to specify the structure of a SharePoint website. The web modules are created as subsites under the home site in SharePoint.

Documentation and resource element types

Documentation and resource element types are used to reference help documentation and system documentation and to develop localized string resources and information resources.

The following model element types are part of the Microsoft Dynamics AX documentation and resource meta-model:

- **Help document set** Use a help documentation set element type to reference a collection of published documents. Help document sets are opened from the Help menu of the Microsoft Dynamics AX Windows client. For more information about creating and updating help documents, see Chapter 16, "Customizing and adding help."
- **System documentation** Use a system documentation element type to reference system library content and hyperlinks to MSDN content. System content describes the Microsoft Dynamics AX system reserved words, functions, tables, types, enums, and classes.
- **Label file** Use a label file element type to store files of localized text resources in the Microsoft Dynamics AX model store.
- **Resource** Use a resource element type to store file resources such as image files and animation files. These resources are stored in the Microsoft Dynamics AX model database.

License and configuration element types

The element types of the Microsoft Dynamics AX application meta-model that are used to develop license, configuration, and application model security are illustrated in Figure 1-5. These model element types change the operational characteristics of the Microsoft Dynamics AX development and runtime environments.

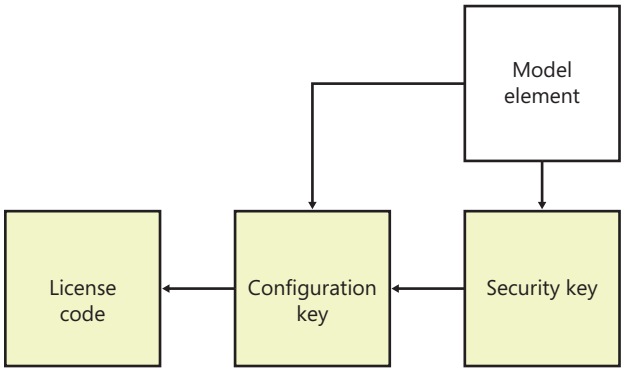

FIGURE 1-5 Element types of the Microsoft Dynamics AX meta-model for developing licensed and configurable application modules.

The following model element types are part of the Microsoft Dynamics AX license, configuration, and application model security meta-model:

- **Configuration key** Use a configuration key element type to assign application model elements to modules that a system administrator then uses to enable and disable application modules and module features. The Microsoft Dynamics AX runtime renders presentation controls that are bound to menu items with active configuration keys. Configuration keys can be specified as subkeys of parent keys.
- **License code** Use a license code element type to lock or unlock the configuration of application modules developed by Microsoft. Modules are locked with license codes that must be unlocked with license keys. License codes can be specified as subcodes of parent codes.

CHAPTER 2

The MorphX development environment and tools

In this chapter

Introduction

Microsoft Dynamics AX includes a set of tools, the MorphX development tools, that you can use to build and modify Microsoft Dynamics AX business applications. Each feature of a business application uses the application model elements described in Chapter 1, "Architectural overview." With the MorphX tools, you can create, view, modify, and delete the application model elements, which contain metadata, structure (ordering and hierarchies of elements), properties (key and value pairs), and X++ code. For example, a table element includes the name of the table, the properties set for the table, the fields, the indexes, the relations, the methods, and so on.

This chapter describes the most commonly used tools and offers some tips and tricks for working with them. You can find additional information and an overview of other MorphX tools in the MorphX Development Tools section of the Microsoft Dynamics AX software development kit (SDK) 2012 on MSDN.

Tip To enable development mode in Microsoft Dynamics AX 2012, press Ctrl+Shift+W to launch the Development Workspace, which holds all of the development tools.

Table 2-1 lists the MorphX tools that are discussed in this chapter.

TABLE 2-1 MorphX tools and other components used for development.

Tool	Use this tool to
Application Object Tree (AOT)	Start development activities. The AOT is the main entry point for most development activities. It allows for browsing the repository of all elements that together make up the business application. You can use the AOT to invoke the other tools and to inspect and create elements.
Projects	Group related elements into projects.
Property sheet	Inspect and modify properties of elements. The property sheet shows key and value pairs.
X++ code editor	Inspect and write X++ source code.
Label editor	Create and inspect localizable strings.
Compiler	Compile X++ code into an executable format.
Best Practices tool	Automatically detect defects in both your code and your elements.
Debugger	Find bugs in your X++ code.
Reverse Engineering tool	Generate Microsoft Visio Unified Modeling Language (UML) and Entity Relationship Diagrams (ERDs) from elements.
Table Browser tool	View the contents of a table directly from a table element.
Type Hierarchy Browser and Type Hierarchy Context	Navigate and understand the type hierarchy of the currently active element.
Find tool	Search for code or metadata patterns in the AOT.
Compare tool	See a line-by-line comparison of two versions of the same element.
Cross-Reference tool	Determine where an element is used.
Version control	Track all changes to elements and see a full revision log.

You can access these development tools from the following places:

- In the Development Workspace, on the Tools menu.
- On the context menu of elements in the AOT.

You can personalize the behavior of many MorphX tools by clicking Options on the Tools menu. Figure 2-1 shows the Options form.

Application Object Tree

The AOT is the main entry point to MorphX and the repository explorer for all metadata. You can open the AOT by clicking the AOT icon on the toolbar or by pressing Ctrl+D. The AOT icon looks like this:

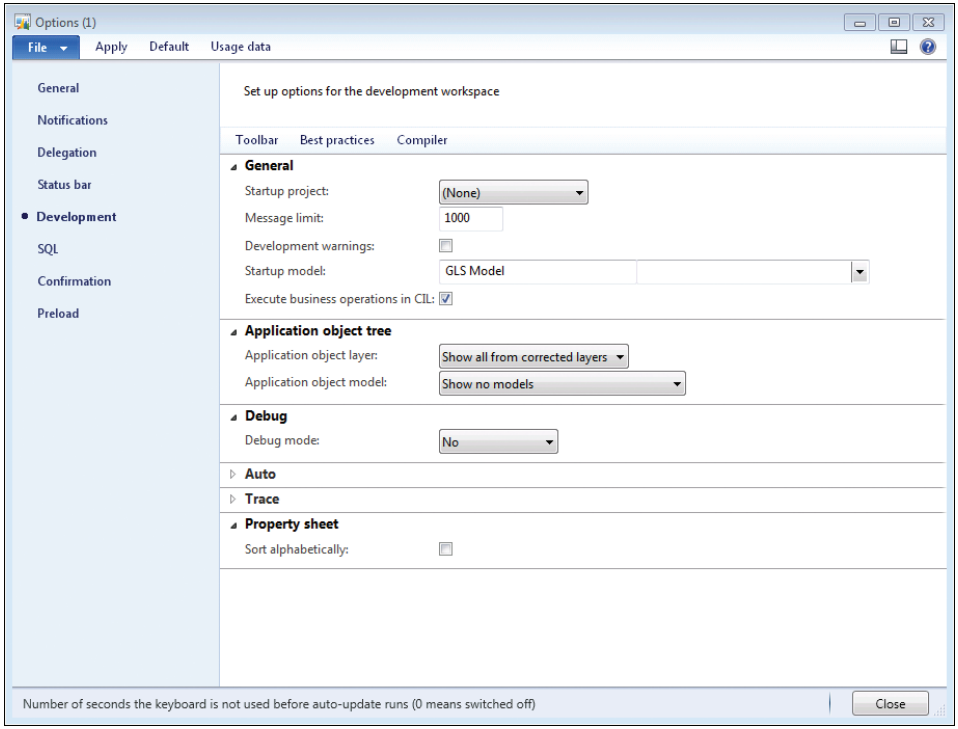

FIGURE 2-1 The Options form, in which development options are specified.

Navigate through the AOT

As the name implies, the AOT is a tree view. The root of the AOT contains the element categories, such as Classes, Tables, and Forms. Some elements are grouped into subcategories to provide a better structure. For example, Tables, Maps, Views, and Extended Data Types are located under Data Dictionary, and all web-related elements are located under Web. Figure 2-2 shows the AOT.

You can navigate through the AOT by using the arrow keys on the keyboard. Pressing the right arrow key expands a node if it has any children.

Elements are arranged alphabetically. Because thousands of elements exist, understanding the naming conventions and adhering to them is important to use the AOT effectively.

All element names in the AOT use the following structure:

<Business area name> + <Functional area> + <Functionality, action performed, or type of content>

With this naming convention, similar elements are placed next to each other. The business area name is also often referred to as the *prefix*. Prefixes are commonly used to indicate the team responsible for an element. For example, in the name *VendPaymReconciliationImport*, the prefix *Vend* is an abbreviation of the business area name (Vendor), *PaymReconciliation* describes the functional area (payment reconciliation), and *Import* lists the action performed (import). The name *CustPaymReconciliationImport* describes a similar functional area and action for the business area Customer.

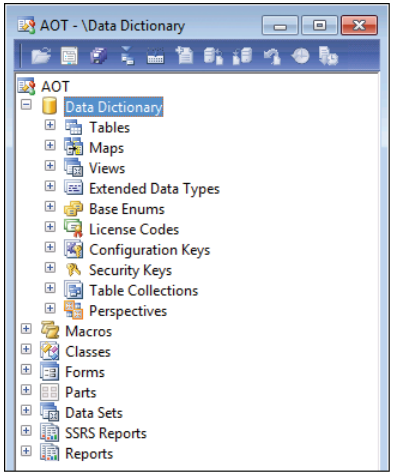

FIGURE 2-2 The AOT.

Tip When building add-on functionality, in addition to following this naming convention, you should add another prefix that uniquely identifies the solution. This additional prefix will help prevent name conflicts if your solution is combined with work from other sources. Consider using a prefix that identifies the company and the solution. For example, if a company called MyCorp is building a payroll system, it could use the prefix *McPR* on all elements added.

Table 2-2 contains a list of the most common prefixes and their descriptions.

TABLE 2-2 Common prefixes.

Prefix	Description
Ax	Microsoft Dynamics AX typed data source
Axd	Microsoft Dynamics AX business document
Asset	Asset management
BOM	Bill of material
COS	Cost accounting
Cust	Customer
Dir	Directory, global address book
EcoRes	Economic resources
HRM/HCM	Human resources
Invent	Inventory management

Prefix	Description
JMG	Shop floor control
KM	Knowledge management
Ledger	General ledger
PBA	Product builder
Prod	Production
Proj	Project
Purch	Purchase
Req	Requirements
Sales	Sales
SMA	Service management
SMM	Sales and marketing management, also called customer relationship management (CRM)
Sys	Application frameworks and development tools
Tax	Tax engine
Vend	Vendor
Web	Web framework
WMS	Warehouse management

Tip When creating new elements, ensure that you follow the recommended naming conventions. Any future development and maintenance will be much easier.

Projects, described in detail later in this chapter, provides an alternative view of the information in the AOT.

Create elements in the AOT

You can create new elements in the AOT by right-clicking the element category node and selecting New <*Element Type*>, as shown in Figure 2-3.

Elements are given automatically generated names when they are created. However, you should replace the default names with new names that conform to the naming convention.

Modify elements in the AOT

Each node in the AOT has a set of properties and either subnodes or X++ code. You can use the property sheet (shown in Figure 2-9) to inspect or modify properties, and you can use the X++ code editor (shown in Figure 2-11) to inspect or modify X++ code.

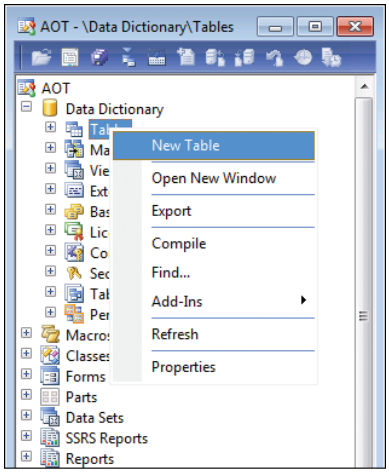

FIGURE 2-3 Creating a new element in the AOT.

The order of the subnodes can play a role in the semantics of the element. For example, the tabs on a form appear in the order in which they are listed in the AOT. You can change the order of nodes by selecting a node and pressing the Alt key while pressing the Up or Down arrow key.

A red vertical line next to a root element name marks it as modified and unsaved, or *dirty*, as shown in Figure 2-4.

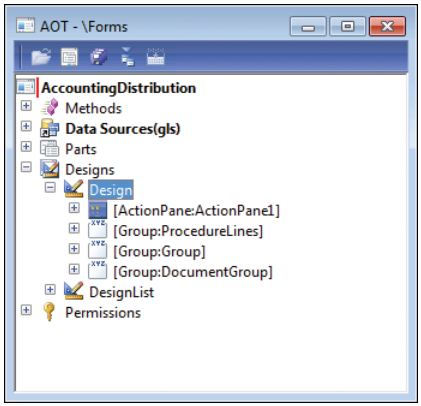

FIGURE 2-4 A dirty element in the AOT, indicated by a vertical line next to the top-level node *AccountingDistribution*.

A dirty element is saved in the following situations:

- The element is executed.
- The developer explicitly invokes the Save or Save All action.

- Autosave takes place. You specify the frequency of autosave in the Options form, which is accessible from the Tools menu.

Refresh elements in the AOT

If several developers modify elements simultaneously in the same installation of Microsoft Dynamics AX, each developer's local elements might not be synchronized with the latest version. To ensure that the local versions of remotely changed elements are updated, an autorefresh thread runs in the background. This autorefresh functionality eventually updates all changes, but you might want to force a refresh explicitly. You do this by right-clicking the element you want to restore and then click Restore. This action refreshes both the on-disk and the in-memory versions of the element.

Typically, the general integrity of what's shown in the AOT is managed automatically, but some operations, such as restoring the application database or reinstalling the application, can lead to inconsistencies that require manual resolution to ensure that the latest elements are used, as follows:

1. Close the Microsoft Dynamics AX client to clear any in-memory elements.
2. Stop the Microsoft Dynamics Server service on the Application Object Server (AOS) to clear any in-memory elements.
3. Delete the application element cache files (*.auc) from the Local Application Data folder (located in "%LocalAppData%") to remove the on-disk elements.

Element actions in the AOT

Each node in the AOT contains a set of available actions. You can access these actions from the context menu, which you can open by right-clicking any node.

Here are two facts to remember about actions:

- The actions that are available depend on the type of node you select.
- You can select multiple nodes and perform actions simultaneously on all the nodes selected.

A frequently used action is Open New Window, which is available for all nodes. It opens a new AOT window with the current node as the root. This action was used to create the screen capture of the *AccountingDistribution* element shown in Figure 2-4. After you open a new AOT window, you can drag elements into the nodes, saving time and effort when you're developing an application.

You can extend the list of available actions on the context menu. You can create custom actions for any element in the AOT by using the features provided by MorphX. In fact, all actions listed on the Add-Ins submenu are implemented in MorphX by using X++ and the MorphX tools.

You can enlist a class as a new add-in by following this procedure:

1. Create a new menu item and give it a meaningful name, a label, and Help text.
2. Set the menu item's *Object Type* property to **Class**.
3. Set the menu item's *Object* property to the name of the class to be invoked by the add-in.
4. Drag the menu item to the *SysContextMenu* menu.
5. If you want the action available only for certain nodes, you need to modify the *verifyItem* method on the *SysContextMenu* class.

Element layers and models in the AOT

When you modify an element from a lower layer, a copy of the element is placed in the current layer and the current model. All elements in the current layer appear in bold type (as shown in Figure 2-5), which makes it easy to recognize changes. For a description of the layer technology, see the "Layers" section in Chapter 21, "Application models."

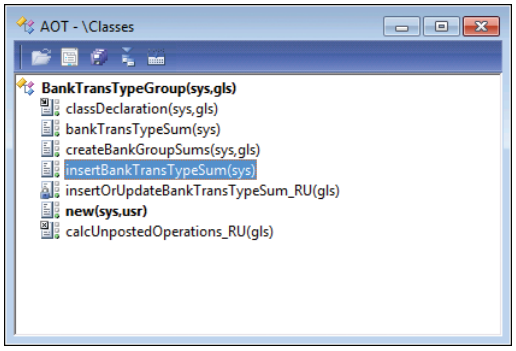

FIGURE 2-5 An element in the AOT that exists in several layers.

You can use the Application object layer and Application object model settings in the Options form to personalize the information shown after the element name in the AOT (see Figure 2-1). Figure 2-5 shows a class with the option set to Show All Layers. As you can see, each method is suffixed with information about the layers in which it is defined, such as *SYS*, *VAR*, and *USR*. If an element exists in several layers, you can right-click it and then click Layers to access its versions from lower layers. It is highly recommended that you use the Show All Layers setting during code upgrade because it provides a visual representation of the layer dimension directly in the AOT.

Projects

For a fully customizable overview of the elements, you can use projects. In a project, you can group and structure elements according to your preference. A project is a powerful alternative to the AOT because you can collect all the elements needed for a feature in one project.

Create a project

You open projects from the AOT by clicking the Project icon on the toolbar. Figure 2-6 shows the Projects window and its *Private* and *Shared* projects nodes.

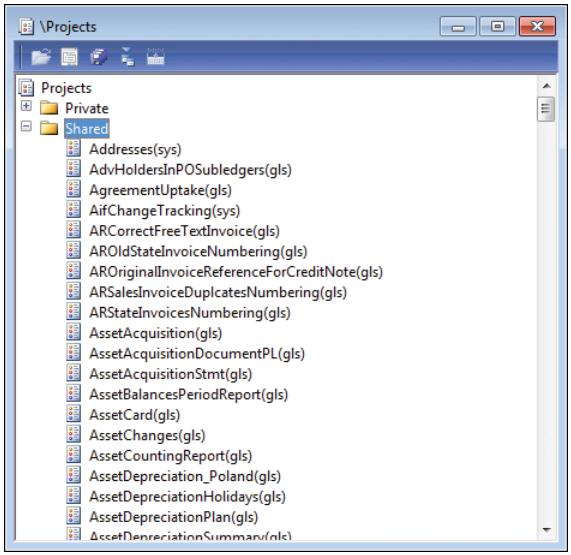

FIGURE 2-6 The Projects window, showing the list of shared projects.

Except for its structure, a project generally behaves like the AOT. Every element in a project is also present in the AOT.

When you create a new project, you must decide whether it should be private or shared among all developers. You can't set access requirements on shared projects. You can make a shared project private (and a private project shared) by dragging it from the shared category into the private category.

Note Central features of Microsoft Dynamics AX 2012 are captured in shared projects to provide an overview of all the elements in a feature. No private projects are included with the application.

You can specify a startup project in the Options form If specified, the chosen project automatically opens when Microsoft Dynamics AX is started.

Automatically generate a project

Projects can be automatically generated in several ways—from using group masks to customizing project types—to make working with them easier. The following sections outline the various ways to generate projects automatically.

Group masks

Groups are folders in a project. When you create a group, you can have its contents be automatically generated by setting the *ProjectGroupType* property (All is an option) and providing a regular expression as the value of the *GroupMask* property. The contents of the group are created automatically and are kept up to date as elements are created, deleted, and renamed. Using group masks ensures that your project is always current, even when elements are created directly in the AOT.

Figure 2-7 shows the *ProjectGroupType* property set to *Classes* and the *GroupMask* property set to *ReleaseUpdate* on a project group. All classes with names containing *ReleaseUpdate* (the prefix for data upgrade scripts) will be included in the project group.

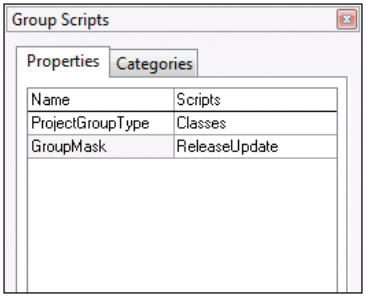

FIGURE 2-7 Property sheet specifying settings for *ProjectGroupType* and *GroupMask*.

Figure 2-8 shows the resulting project when the settings from Figure 2-7 are used.

Filters

You can also generate a project based on a filter. Because all elements in the AOT persist in a database format, you can use a query to filter elements and have the results presented in a project. You create a project filter by clicking the Filter button on the project's toolbar. Depending on the complexity of the query, a project can be generated instantly, or it might take several minutes.

With filters, you can create projects containing elements that meet the following criteria:

- Elements created or modified within the last month
- Elements created or modified by a named user
- Elements from a particular layer

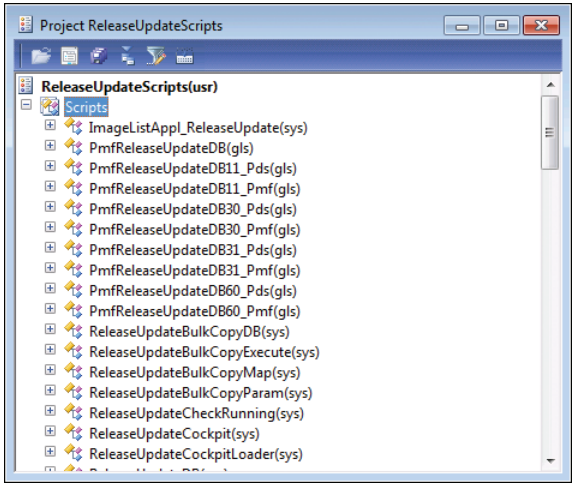

FIGURE 2-8 Project created by using a group mask.

Development tools

Several development tools, such as the Wizard Wizard, produce projects containing elements that the wizard creates. The result of running the Wizard Wizard is a new project that includes a form, a class, and a menu item—all the elements comprising the newly created wizard.

You can also use several other wizards, such as the AIF Document Service Wizard and the Class Wizard, to create projects. To access these wizards, on the Tools menu, click Wizards.

Layer comparison You can compare the elements in one layer with the elements in another layer, which is called the *reference layer*. If an element exists in both layers, and the definitions of the element are different or if the element doesn't exist in the reference layer, the element is added to the resulting project. To compare layers, click Tools > Code Upgrade > Compare Layers.

Upgrade projects When you upgrade from one version of Microsoft Dynamics AX to another or install a new service pack, you need to deal with any new elements that are introduced and existing elements that have been modified. These changes might conflict with customizations you've implemented in a higher layer.

The Create Upgrade Project feature makes a three-way comparison to establish whether an element has any upgrade conflicts. It compares the original version with both the customized version and the updated version. If a conflict is detected, the element is added to the project.

The resulting project provides a list of elements to update based on upgrade conflicts between versions. You can use the Compare tool, described later in this chapter, to see the conflicts in each element. Together, these features provide a cost-effective toolbox to use when upgrading. For more information about code upgrade, see "Microsoft Dynamics AX 2012 White Papers: Code Upgrade" at *http://www.microsoft.com/download/en/details.aspx?id=20864*.

To create an upgrade project, click Tools > Code Upgrade > Detect Code Upgrade Conflicts.

Project types

When you create a new project, you can specify a project type. So far, this chapter has discussed standard projects. The Test project, used to group a set of classes for unit testing, is another specialized project type provided in Microsoft Dynamics AX.

You can create a custom specialized project by creating a new class that extends the *ProjectNode* class. With a specialized project, you can control the structure, icons, and actions available to the project.

Property sheet

Properties are an important part of the metadata system. Each property is a key and value pair. You can use the property sheet to inspect and modify properties of elements.

When the Development Workspace opens, the property sheet is visible by default. If you close it, you can open it again anytime by pressing Alt+Enter or by clicking the Properties button on the toolbar of the Development Workspace. The property sheet automatically updates itself to show properties for any element selected in the AOT. You don't have to open the property sheet manually for each element; you can leave it open and browse the elements. Figure 2-9 shows the property sheet for the *TaxSpec* class. The two columns are the key and value pairs for each property.

Tip Pressing Esc in the property sheet sets the focus back to your origin.

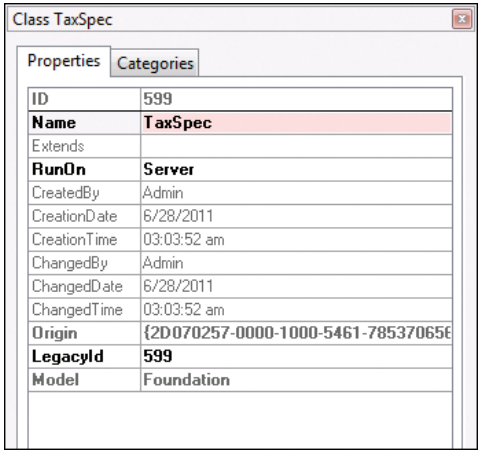

FIGURE 2-9 Property sheet for an element in the AOT.

Figure 2-10 shows the Categories tab for the class shown in Figure 2-9. Here, related properties are categorized. For elements with many properties, this view can make it easier to find the right property.

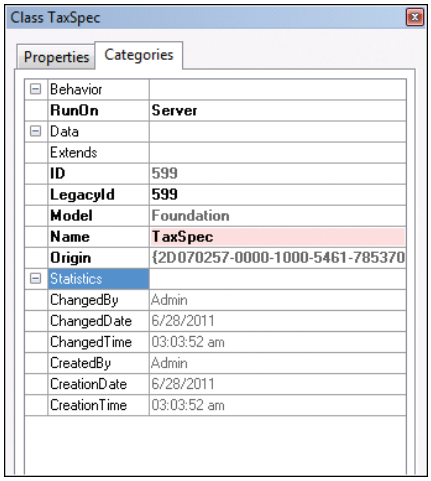

FIGURE 2-10 The Categories tab on the property sheet for an element in the AOT.

Read-only properties appear in gray. Just like files in the file system, elements contain information about who created them and when they were modified. Elements that come from Microsoft all have the same time and user stamps.

The default sort order places related properties near each other. Categories were introduced in an earlier version of Microsoft Dynamics AX to make finding properties easier, but you can also sort properties alphabetically by setting a parameter in the Options form.

You can dock the property sheet on either side of the screen by right-clicking the title bar. Docking ensures that the property sheet is never hidden behind another tool.

X++ code editor

You write all X++ code with the X++ code editor. You open the editor by selecting a node in the AOT and pressing Enter. The editor contains two panes. The left pane shows the methods available, and the right pane shows the X++ code for the selected method, as shown in Figure 2-11.

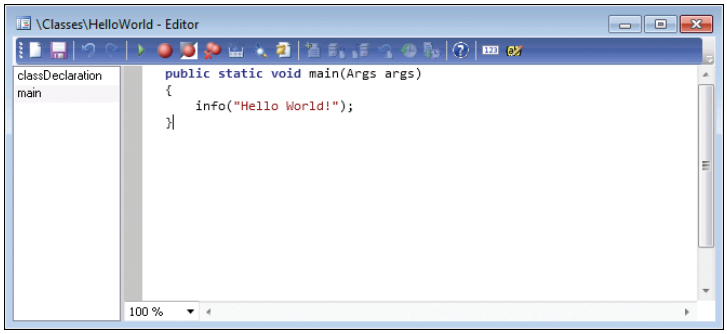

FIGURE 2-11 The X++ code editor.

The X++ code editor is a basic text editor that supports color coding and IntelliSense.

Shortcut keys

Navigation and editing in the X++ code editor use standard shortcuts, as described in Table 2-3. For Microsoft Dynamics AX 2012, some shortcuts differ from those in earlier versions to align with commonly used integrated development environments (IDEs) such as Microsoft Visual Studio.

TABLE 2-3 X++ code editor shortcut keys.

Action	Shortcut	Description
Show Help window	F1	Opens context-sensitive Help for the type or method currently selected in the editor.
Go to next error message	F4	Opens the editor and positions the cursor at the next compilation error, based on the contents of the compiler output window.
Execute current element	F5	Starts the current form, job, or class.
Compile	F7	Compiles the current method.
Toggle a breakpoint	F9	Sets or removes a breakpoint.
Run an editor script	Alt+R	Lists all available editor scripts and lets you select one to execute (such as Send To Mail Recipient).
Open the Label editor	Ctrl+Alt+Spacebar	Opens the Label editor and searches for the selected text.
Go to implementation (drill down in code)	F12	Goes to the implementation of the selected method. This shortcut is highly useful for fast navigation.
Go to the next method	Ctrl+Tab	Sets the focus on the next method in the editor.
Go to the previous method	Ctrl+Shift+Tab	Sets the focus on the previous method in the editor.
Enable block selection	Alt+<mouse select> or Alt+Shift+arrow keys	Selects a block of code. Select the code you want by pressing the Alt key while selecting text with the mouse. Alternatively, hold down Alt and Shift while moving the cursor with the arrow keys.
Cancel selection	Esc	Cancels the current selection.
Delete current selection/line	Ctrl+X	Deletes the current selection or, if nothing is selected, the current line.
Incremental search	Ctrl+I	Starts an incremental search, which marks the first occurrence of the search text as you type it. Pressing Ctrl+I again moves to the next occurrence, and Ctrl+Shift+I moves to the previous occurrence.
Insert XML document header	///	Inserts an XML comment header when you type ///. When done in front of a class or method header, this shortcut prepopulates the XML document with template information relevant to the class or method.
Execute editor script	<*name of script*>+Tab	Runs an editor script when you type the name of an editor script on an empty line in the editor and press Enter. Script names are case sensitive.
Comment selection	Ctrl+E, C	Inserts comment marking for the current selection.
Uncomment selection	Ctrl+E, U	Removes comment marking for the current selection.

Editor scripts

The X++ code editor contains a set of editor scripts that you can invoke by clicking the Script icon on the X++ code editor toolbar or by typing ***<name of script>*+TAB** directly in the editor. Built-in editor scripts provide functionality such as the following:

- Send to mail recipient.
- Send to file.
- Generate code for standard code patterns such as *main*, *construct*, and *parm* methods.
- Open the AOT for the element that owns the method.

Note By generating code, in a matter of minutes you can create a new class with the right constructor method and the right encapsulation of member variables by using *parm* methods. *Parm* methods (*parm* is short for "parameter") are used as simple property getters and setters on classes. Code is generated in accordance with X++ best practices.

Tip To add a main method to a class, add a new method, press Ctrl+A to select all code in the editor tab for the new method, type **main**, and then press the Tab key. This will replace the text in the editor with the standard template for a static main method.

The list of editor scripts is extendable. You can create your own scripts by adding new methods to the *EditorScripts* class.

Label editor

The term *label* in Microsoft Dynamics AX refers to a localizable text resource. Text resources are used throughout the product as messages to the user, form control labels, column headers, Help text in the status bar, captions on forms, and text on web forms, to name just a few places. Labels are localizable, meaning that they can be translated into most languages. Because the space requirement for displaying text resources typically depends on the language, you might fear that the actual user interface must be manually localized as well. However, with IntelliMorph technology, the user interface is dynamically rendered and honors any space requirements imposed by localization.

The technology behind the label system is simple. All text resources are kept in a Unicode-based label file that must have a three-letter identifier. In Microsoft Dynamics AX 2012, the label files are managed in the AOT and distributed using model files. Figure 2-12 shows how the *Label Files* node in the AOT looks with multiple label files and the language en-us.

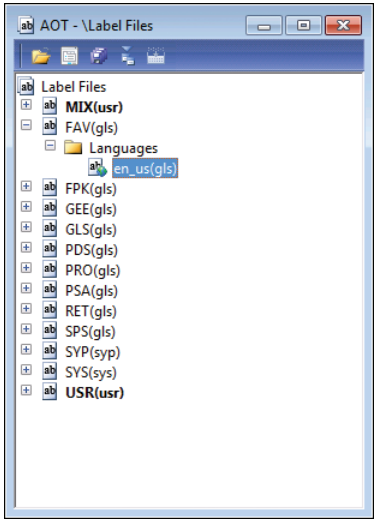

FIGURE 2-12 The *Label Files* node in the AOT.

The underlying source representation is a simple text file following this naming convention:

Ax<Label file identifier><Locale>*.ALD*

The following are two examples, the first showing U.S. English and the second a Danish label file:

Axsysen-us.ALD

Axtstda.ALD

Each text resource in the label file has a 32-bit integer label ID, label text, and an optional label description. The structure of the label file is simple:

@<Label file identifier><Label ID> <Label text>

[Label description]

Figure 2-13 shows an example of a label file.

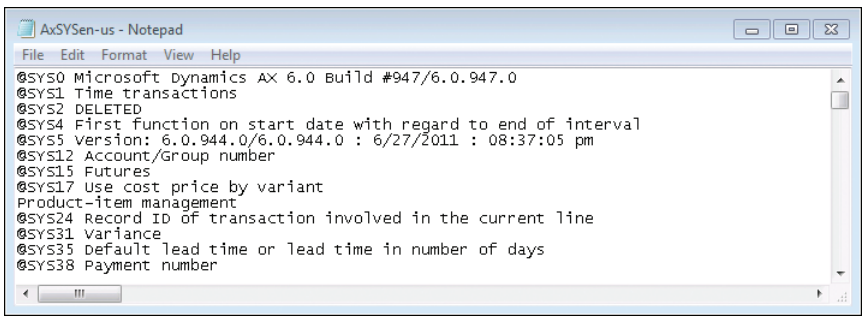

FIGURE 2-13 Label file opened in Windows Notepad showing a few labels from the en-us label file.

This simple structure allows for localization outside of Microsoft Dynamics AX with third-party tools. The AOT provides a set of operations for the label files, including an Export To Label file that can be used to extract a file for external translation.

You can create new label files by using the Label File Wizard, which you access directly from the *Label Files* node in the AOT, or from the Tools menu by pointing to Wizards > Label File Wizard. The wizard guides you through the steps of adding a new label file or a new language to an existing label file. After you run the wizard, the label file is ready to use. If you have an existing .ald file, you can also create the appropriate entry in the AOT by using Create From File on the context menu of the *Label Files* node in the AOT.

Note You can use any combination of three letters when naming a label file, and you can use any label file from any layer. A common misunderstanding is that the label file identifier must match the layer in which it is used. Microsoft Dynamics AX includes a *SYS* layer and a label file named *SYS*; service packs contain a *SYP* layer and a label file named *SYP*. This naming standard was chosen because it is simple, easy to remember, and easy to understand. However, Microsoft Dynamics AX doesn't impose any limitations on the label file name.

Consider the following tips for working with label files:

- When naming a label file, choose a three-letter ID that has a high chance of being unique, such as your company's initials. Don't choose the name of the layer such as *VAR* or *USR*. Eventually, you'll probably merge two separately developed features into the same installation, a task that will be more difficult if the label file names collide.
- When referencing existing labels, feel free to reference labels in the label files provided by Microsoft, but avoid making changes to labels in these label files because they are updated with each new version of Microsoft Dynamics AX.

Create a label

You use the Label editor to create new labels. You can start the Label editor by using any of the following procedures:

- On the Tools menu, point to Label > Label Editor.
- On the X++ code editor toolbar, click the Lookup Label > Text button.
- On text properties in the property sheet, click the Lookup button.

You can use the Label editor (shown in Figure 2-14) to find existing labels. Reusing a label is sometimes preferable to creating a new one. You can create a new label by pressing Ctrl+N or by clicking New.

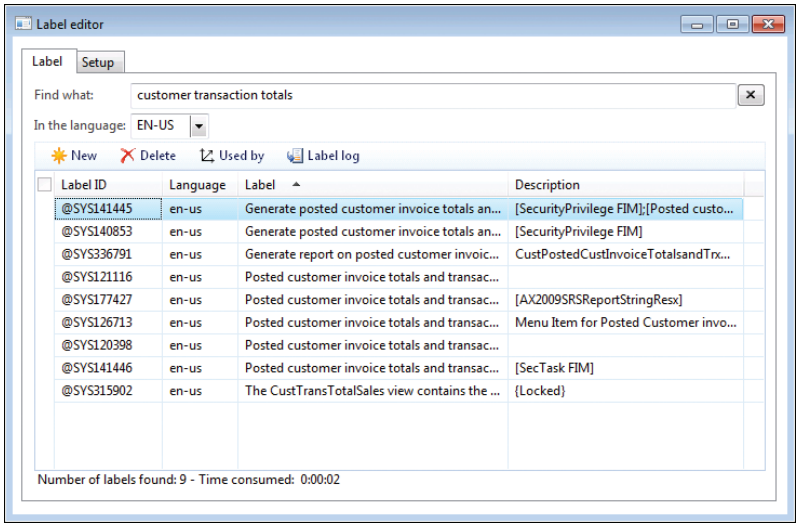

FIGURE 2-14 The Label editor.

In addition to finding and creating new labels, you can also use the Label editor to find out where a label is used. The Label editor also logs any changes to each label.

Consider the following tips when creating and reusing labels:

- When reusing a label, make sure that the label means what you intend it to in all languages. Some words are homonyms (words that have many meanings), and they naturally translate into many different words in other languages. For example, the English word *can* is both a verb and a noun. Use the description column to note the intended meaning of the label.
- When creating a new label, ensure that you use complete sentences or other stand-alone words or phrases. Don't construct complete sentences by concatenating labels with one or two words because the order of words in a sentence differs from one language to another.

Reference labels from X++

In the MorphX design environment, labels are referenced in the format @<LabelFileIdentifier><LabelID>. If you don't want a label reference to be converted automatically to the label text, you can use the *literalStr* function. When a placeholder is needed to display the value of a variable, you can use the *strFmt* function and a string containing %*n*, where *n* is greater than or equal to 1. Placeholders can also be used within labels. The following code shows a few examples:

```
// prints: Time transactions
print "@SYS1";

// prints: @SYS1
print literalStr("@SYS1");
```

```
// prints: Microsoft Dynamics is a Microsoft brand
print strFmt("%1 is a %2 brand", "Microsoft Dynamics", "Microsoft");
pause;
```

The following are some best practices to consider when referencing labels from X++:

- Always create user interface text by using a label. When referencing labels from X++ code, use double quotation marks.
- Never create system text such as file names by using a label. When referencing system text from X++ code, use single quotation marks. You can place system text in macros to make it reusable.

Using single and double quotation marks to differentiate between system text and user interface text allows the Best Practices tool to find and report any hard-coded user interface text. The Best Practices tool is described in depth later in this chapter.

Code compiler

Whenever you make a change to X++ code, you must recompile, just as you would in any other programming language. You start the recompile by pressing F7 in the X++ code editor. Your code also recompiles whenever you close the editor or save changes to an element.

The compiler also produces a list of the following information:

- **Compiler errors** These prevent code from compiling and should be fixed as soon as possible.
- **Compiler warnings** These typically indicate that something is wrong in the implementation. See Table 2-4, later in this section, for a list of compiler warnings. Compiler warnings can and should be addressed. Check-in attempts with compiler warnings are rejected unless specifically allowed in the version control system settings.
- **Tasks (also known as *to-dos*)** The compiler picks up single-line comments that start with *TODO*. These comments can be useful during development for adding reminders, but you should use them only in cases in which implementation can't be completed. For example, you might use a to-do comment when you're waiting for a check-in from another developer. Be careful when using to-do comments to postpone work, and never release code unless they are addressed. For a developer, there is nothing worse than debugging an issue and finding a to-do comment indicating that the issue was already known but overlooked.
- **Best practice deviations** The Best Practices tool carries out more complex validations. For more information, see the "Best Practices tool" later in this chapter.

Note Unlike other languages, X++ requires that you compile only code you've modified, because the intermediate language the compiler produces is persisted along with the X++ code and metadata. Of course, your changes can require other methods consuming your code to be changed and recompiled if, for example, you rename a method or modify its parameters. If the consumers are not recompiled, a run-time error is thrown when they are invoked. This means that you can execute your business application even when compile errors exist, so long as you don't use the code that can't compile. Always ensure that you compile the entire AOT when you consider your changes complete and fix any compilation errors found. If you're changing the class declaration somewhere in a class hierarchy, all classes deriving from the changed class should be recompiled too. This can be achieved using the Compile Forward option under Add-Ins in the context menu for the changed class node.

The Compiler output window provides access to every issue found during compilation, as shown in Figure 2-15. The window presents one list of all relevant errors, warnings, best practices, and tasks. Each type of message can be disabled or enabled by using the respective buttons. Each line in the list contains information about each issue that the compiler detects, a description of the issue, and its location.

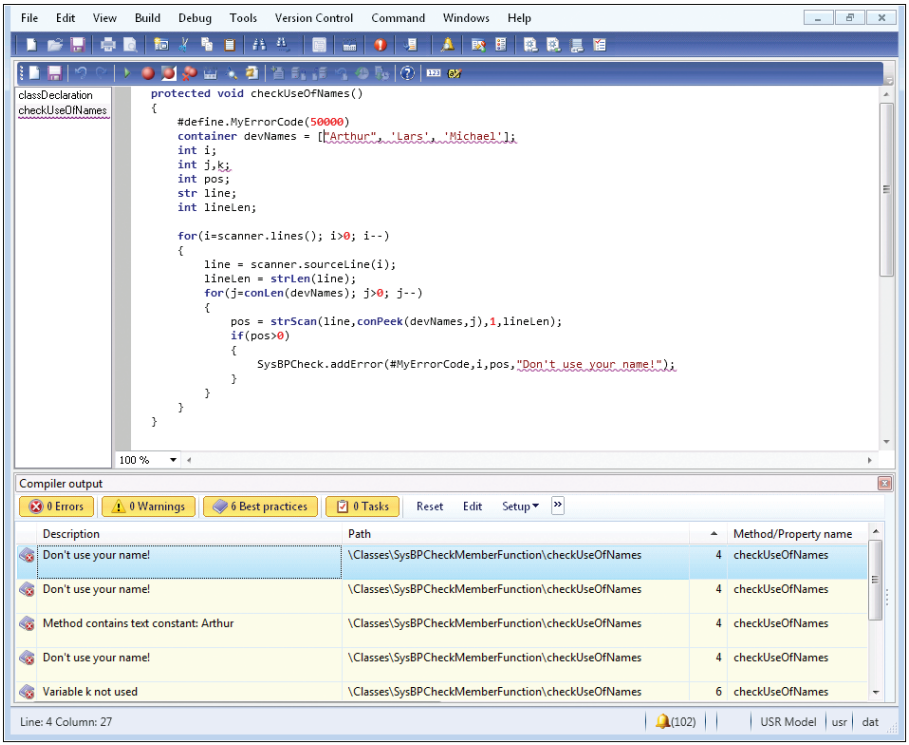

FIGURE 2-15 The powerful combination of the X++ code editor and the Compiler output window.

You can export the contents of the Compiler output window. This capability is useful if you want to share the list of issues with team members. The exported file is an HTML file that can be viewed in Windows Internet Explorer or reimported into the Compiler output window in another Microsoft Dynamics AX session.

In the Compiler output window, click Setup > Compiler to define the types of issues that the compiler should report. Compiler warnings are grouped into four levels, as shown by the examples in Table 2-4. Each level represents a certain level of severity, with 1 being the most critical and 4 being recommended to comply with best practices.

TABLE 2-4 Example compiler warnings.

Warning message	Level
Break statement found outside legal context	1
The new method of a derived class does not call super()	1
The new method of a derived class may not call super()	1
Function never returns a value	1
Not all paths return a value	1
Assignment/comparison loses precision	1
Unreachable code	2
Empty compound statement	3
Class names should start with an upper case letter	4
Member names should start with a lower case letter	4

Best Practices tool

Following Microsoft Dynamics AX best practices when you develop applications has several important benefits:

- You avoid less-than-obvious pitfalls. Following best practices helps you avoid many obstacles, even those that appear only in border scenarios that would otherwise be difficult and time consuming to detect and test. Using best practices allows you to take advantage of the combined experience of Microsoft Dynamics AX expert developers.
- Your learning curve is flattened. When you perform similar tasks in a standard way, you are more comfortable in an unknown area of the application. Consequently, adding new resources to a project is more cost effective, and downstream consumers of the code can make changes more readily.
- You are making a long-term investment. Code that conforms to standards is less likely to require rework during an upgrade process, whether you're upgrading to Microsoft Dynamics AX 2012, installing service packs, or upgrading to future releases.
- You are more likely to ship on time. Most of the problems you face when implementing a solution in Microsoft Dynamics AX have been solved at least once before. Choosing a proven solution results in faster implementation and less regression. You can find solutions to known problems in both the Developer Help section of the SDK and the code base.

The Microsoft Dynamics AX 2012 SDK contains an important discussion about conforming to best practices in Microsoft Dynamics AX. Constructing code that follows proven standards and patterns can't guarantee a project's success, but it minimizes the risk of failure through late, expensive discovery and decreases the long-term maintenance cost. The Microsoft Dynamics AX 2012 SDK is available at *http://msdn.microsoft.com/en-us/library/aa496079.aspx*.

The Best Practices tool is a powerful supplement to the best practices discussion in the SDK. This tool is the MorphX version of a static code analysis tool, similar to FxCop for the Microsoft .NET Framework. The Best Practices tool is embedded in the compiler, and the results are reported in the Compiler output window the same way as other messages from the compilation process.

The purpose of static code analysis is to detect defects and risky coding patterns in the code automatically. The longer a defect exists, the more costly it becomes to fix—a bug found in the design phase is much cheaper to correct than a bug in shipped code running at several customer sites. The Best Practices tool allows any developer to run an analysis of his or her code and application model to ensure that it conforms to a set of predefined rules. Developers can run analysis during development, and they should always do so before implementations are tested. Because an application in Microsoft Dynamics AX is much more than just code, the Best Practices tool also performs static analysis on the metadata—the properties, structures, and relationships that are maintained in the AOT.

The Best Practices tool displays deviations from the best practice rules, as shown in Figure 2-15. Double-clicking a line on the Best Practices tab opens the X++ code editor on the violating line of code or, if the Best Practices violation is related to metadata, it will open the element in an AOT window.

Rules

The Best Practices tool includes about 400 rules, a small subset of the best practices mentioned in the SDK. You can define the best practice rules that you want to run in the Best practice parameters dialog box: on the Tools menu, click > Options > Development, and then click Best Practices.

Note You must set the compiler error level to 4 if you want best practice rule violations to be reported. To turn off the Best Practices tool, on the Tools menu, click Options > Development, and then click Compiler and set the diagnostic level to less than 4.

The best practice rules are divided into categories. By default, all categories are turned on, as shown in Figure 2-16.

The best practice rules are divided into three levels of severity:

- **Errors** The majority of the rules focus on errors. Any check-in attempt with a best practice error is rejected. You must take all errors seriously and fix them as soon as possible.
- **Warnings** Follow a 95/5 rule for warnings. This means that you should treat 95 percent of all warnings as errors; the remaining 5 percent constitute exceptions to the rule. You should provide valid explanations in the design document for all warnings you choose to ignore.

- **Information** In some situations, your implementation might have a side effect that isn't obvious to you or the user (for example, if you assign a value to a variable but you never use the variable again). These are typically reported as information messages.

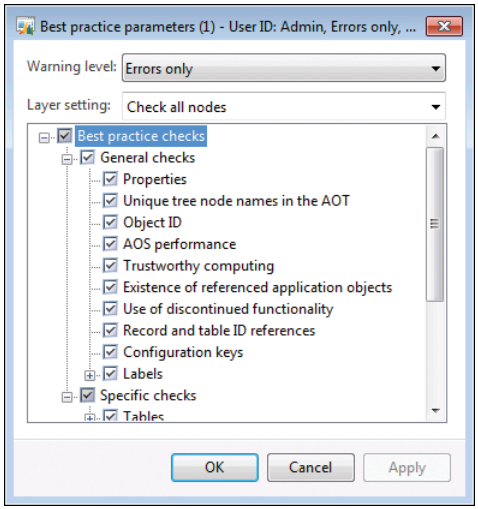

FIGURE 2-16 The Best Practice Parameters dialog box.

Suppress errors and warnings

The Best Practices tool allows you to suppress errors and warnings. A suppressed best practice deviation is reported as information. This gives you a way to identify the deviation as reviewed and accepted. To identify a suppressed error or warning, place a line containing the following text just before the deviation:

```
//BP Deviation Documented
```

Only a small subset of the best practice rules can be suppressed. Use the following guidelines for selecting which rules to suppress:

- **Dangerous API exceptions** When exceptions exist that are impossible to detect automatically, examine each error to ensure the correct implementation. Dangerous application programming interfaces (APIs) are often responsible for such exceptions. A dangerous API is an API that can compromise a system's security when used incorrectly. If a dangerous API is used, a suppressible error is reported. You can use some so-called dangerous APIs when you take certain precautions, such as using code access security (CAS). You can suppress the error after you apply the appropriate mitigations.
- **False positives** About 5 percent of all warnings are false positives and can be suppressed. Note that only warnings caused by actual code can be suppressed this way, not warnings caused by metadata.

After you set up the best practices, the compiler automatically runs the best practices check whenever an element is compiled. The results are displayed on the Best Practices list in the Compiler output dialog box.

Some of the metadata best practice violations can also be suppressed, but the process of suppressing them is different. Instead of adding a comment to the source code, the violation is added to a global list of ignored violations. This list is maintained in the macro named *SysBPCheckIgnore*. This allows for central review of the number of suppressions, which should be kept to a minimum.

Add custom rules

You can use the Best Practices tool to create your own set of rules. The classes used to check for rules are named *SysBPCheck*<Element type>. You call the *init*, *check*, and *dispose* methods once for each node in the AOT for the element being compiled.

One of the most interesting classes is *SysBPCheckMemberFunction*, which is called for each piece of X++ code whether it is a class method, form method, macro, or other method. For example, if developers don't want to include their names in the source code, you can implement a best practice check by creating the following method on the *SysBPCheckMemberFunction* class:

```
protected void checkUseOfNames()
{
    #Define.MyErrorCode(50000)
    container devNames = ['Arthur', 'Lars', 'Michael'];
    int i;
    int j,k;
    int pos;
    str line;
    int lineLen;

    for (i=scanner.lines(); i>0; i--)
    {
        line = scanner.sourceLine(i);
        lineLen = strLen(line);
        for (j=conLen(devNames); j>0; j--)
        {
            pos = strScan(line, conPeek(devNames, j), 1, lineLen);
            if (pos)
            {
                sysBPCheck.addError(#MyErrorCode, i, pos,
                    "Don't use your name!");
            }
        }
    }
}
```

To enlist the rule, make sure to call the preceding method from the *check* method. Compiling this sample code results in the best practice errors shown in Table 2-5.

TABLE 2-5 Best practice errors in *checkUseOfNames*.

Message	Line	Column
Don't use your name!	4	28
Don't use your name!	4	38
Don't use your name!	4	46
Variable k not used	6	11
Method contains text constant: 'Don't use your name!'	20	59

In an actual implementation, names of developers would probably be read from a file. Ensure that you cache the names to prevent the compiler from going to the disk to read the names for each method being compiled.

Note The best practice check also identified that the code contained a variable named *k* that was declared, but never referenced. This is one of the valuable checks ensuring that the code can easily be kept up to date, which helps avoid mistakes. In this case, *k* was not intended for a specific purpose and can be removed.

Debugger

Like most development environments, MorphX features a debugger. The debugger is a stand-alone application, not part of the Microsoft Dynamics AX shell like the rest of the tools mentioned in this chapter. As a stand-alone application, the debugger allows debugging of X++ in any of the following Microsoft Dynamics AX components:

- Microsoft Dynamics AX client
- AOS
- Business Connector (BC.NET)

For other debugging scenarios, such as web services, Microsoft SQL Server Reporting Services (SSRS) reports, and Enterprise Portal, see Chapter 3, "Microsoft Visual Studio tools for Microsoft Dynamics AX."

Enable debugging

For the debugger to start, a breakpoint must be hit when X++ code is executed. You set breakpoints by using the X++ code editor in the Microsoft Dynamics AX Development Workspace. The debugger starts automatically when any component hits a breakpoint.

You must enable debugging for each component as follows:

- In the Development Workspace, on the Tools menu, click Options > Development > Debug, and then select When Breakpoint in the Debug mode list.
- From the AOS, open the Microsoft Dynamics AX Server Configuration Utility under Start > Administrative Tools > Microsoft Dynamics AX 2012 Server Configuration. Create a new configuration, if necessary, and then select the check box Enable Breakpoints to debug X++ code running on this server.
- For Enterprise Portal code that uses the BCPROXY context to run interpreted X++ code, in the Microsoft Dynamics AX Server Configuration Utility, create a new configuration, if necessary, and select the check box Enable Global Breakpoints.

Ensure that you are a member of the local Windows Security Group named Microsoft Dynamics AX Debugging Users. This is normally ensured using setup, but if you did not set up Microsoft Dynamics AX by using your current account, you need to do this manually through Edit Local Users And Groups in the Windows Control Panel. This is necessary to prohibit unauthorized debugging, which could expose sensitive data, provide a security risk, or impose unplanned service disruptions.

Caution It is recommended that you do not enable any of the debugging capabilities in a live environment. If you do, execution will stop when it hits a breakpoint, and the client will stop responding to users. Running the application with debug support enabled also noticeably affects performance.

To set or remove breakpoints, press F9. You can set a breakpoint on any line you want. If you set a breakpoint on a line without an X++ statement, however, the breakpoint will be triggered on the next X++ statement in the method. A breakpoint on the last brace will never be hit.

To enable or disable a breakpoint, press Ctrl+F9. For a list of all breakpoints, press Shift+F9.

Breakpoints are persisted in the *SysBreakpoints* and *SysBreakpointLists* database tables. Each developer has his or her own set of breakpoints. This means that your breakpoints are not cleared when you close Microsoft Dynamics AX and that other Microsoft Dynamics AX components can access them and break where you want them to.

Debugger user interface

The main window in the debugger initially shows the point in the code where a breakpoint was hit. You can control execution one step at a time while inspecting variables and other aspects of the code. Figure 2-17 shows the debugger opened to a breakpoint with all the windows enabled.

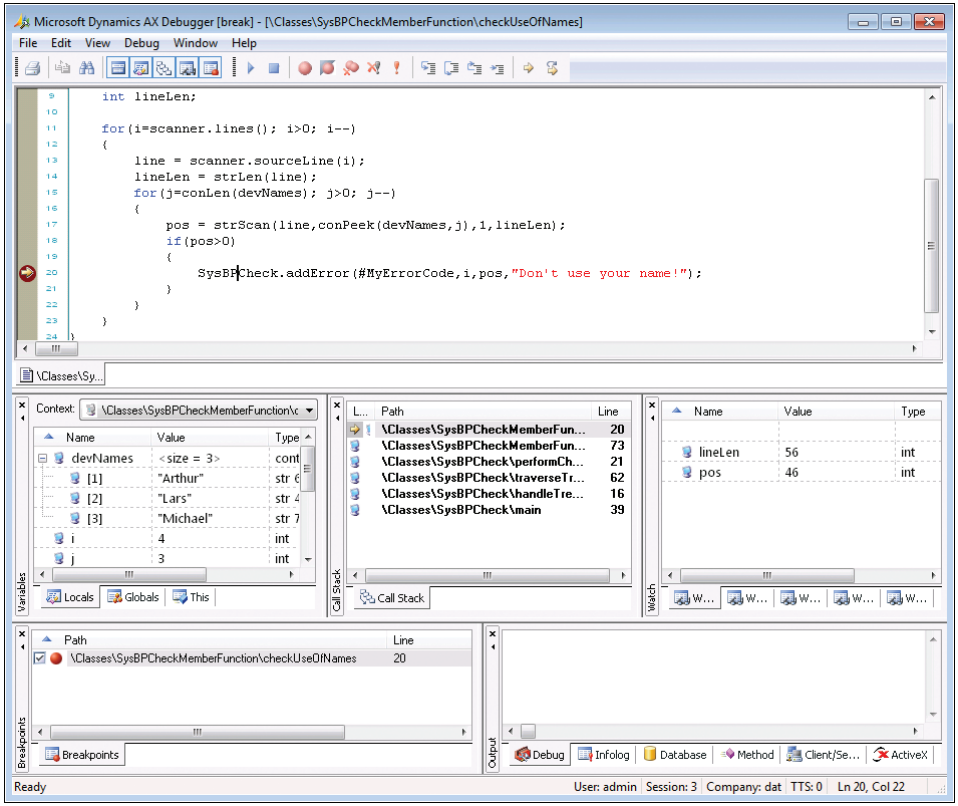

FIGURE 2-17 Debugger with all windows enabled.

Table 2-6 describes the debugger's various windows and some of its other features.

TABLE 2-6 Debugger user interface (UI) elements.

Debugger element	Description
Code window	Shows the current X++ code. Each variable has a ScreenTip that reveals its value. You can drag the next-statement pointer in the left margin. This pointer is particularly useful if the execution path isn't what you expected or if you want to repeat a step.
Variables window	Shows local, global, and member variables, along with their names, values, and types. Local variables are variables in scope at the current execution point. Global variables are the global classes that are always instantiated: Appl, Infolog, ClassFactory, and VersionControl. Member variables make sense only on classes, and they show the class member variables. If a variable is changed as you step through execution, it is marked in red. Each variable is associated with a client or server icon. You can modify the value of a variable by double-clicking the value.

Call Stack window	Shows the code path followed to arrive at a particular execution point. Clicking a line in the Call Stack window opens the code in the Code window and updates the local Variables window. A client or server icon indicates the tier on which the code is executed.
Watch window	Shows the name, value, and type of the variables. Five different Watch windows are available. You can use these to group the variables you're watching in the way that you prefer. You can use this window to inspect variables without the scope limitations of the Variables window. You can drag a variable here from the Code window or the Variables window.
Breakpoints window	Lists all your breakpoints. You can delete, enable, and disable the breakpoints through this window.
Output window	Shows the traces that are enabled and the output sent to the Infolog application framework, which is introduced in Chapter 5, "Designing the user experience." The Output window includes the following pages: ■ **Debug** You can instrument your X++ code to trace to this page by using the *printDebug* static method on the *Debug* class. ■ **Infolog** This page contains messages in the queue for the Infolog. ■ **Database, Client/Server, and ActiveX Trace** Any traces enabled on the Development tab in the Options form appear on these pages.
Status bar window	Provides the following important context information: ■ **Current user** The ID of the user who is logged on to the system. This information is especially useful when you are debugging incoming web requests. ■ **Current session** The ID of the session on the AOS. ■ **Current company accounts** The ID of the current company accounts. ■ **Transaction level** The current transaction level. When it reaches zero, the transaction is committed.

Tip As a developer, you can provide more information in the value field for your classes than what is provided by default. The defaults for classes are *New* and *Null*. You can change the defaults by overriding the *toString* method. If your class doesn't explicitly extend the *object* (the base class of all classes), you must add a new method named *toString*, returning *str* and taking no parameters, to implement this functionality.

Debugger shortcut keys

Table 2-7 lists the most important shortcut keys available in the debugger.

TABLE 2-7 Debugger shortcut keys.

Action	Shortcut	Description
Run	F5	Continue execution
Stop debugging	Shift+F5	Break execution
Step over	F10	Step over next statement
Run to cursor	Ctrl+F10	Continue execution but break at the cursor's position
Step into	F11	Step into next statement
Step out	Shift+F11	Step out of method
Toggle breakpoint	Shift+F9	Insert or remove breakpoint
Variables window	Ctrl+Alt+V	Open or close the Variables window
Call Stack window	Ctrl+Alt+C	Open or close the Call Stack window
Watch window	Ctrl+Alt+W	Open or close the Watch window
Breakpoints window	Ctrl+Alt+B	Open or close the Breakpoints window
Output window	Ctrl+Alt+O	Open or close the Output window

Reverse Engineering tool

You can generate Visio models from existing metadata. Considering the amount of metadata available in Microsoft Dynamics AX 2012 (more than 50,000 elements and more than 18 million lines of text when exported), it's practically impossible to get a clear view of how the elements relate to each other just by using the AOT. The Reverse Engineering tool is a great aid when you need to visualize metadata.

Note You must have Visio 2007 or later installed to use the Reverse Engineering tool.

The Reverse Engineering tool can generate a Unified Modeling Language (UML) data model, a UML object model, or an entity relationship data model, including all elements from a private or shared project. To open the tool, in the Projects window, right-click a project or a perspective, point to Add-Ins > Reverse Engineer. You can also open the tool by selecting Reverse Engineer from the Tools menu. In the dialog box shown in Figure 2-18, you must specify a file name and model type.

When you click OK, the tool uses the metadata for all elements in the project to generate a Visio document that opens automatically. You can drag elements from the Visio Model Explorer onto the drawing surface, which is initially blank. Any relationship between two elements is automatically shown.

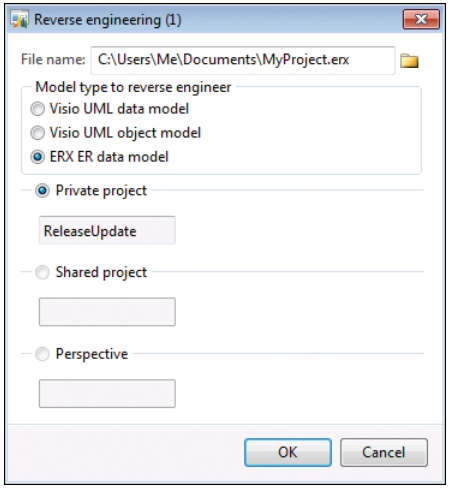

FIGURE 2-18 The Reverse Engineering dialog box.

UML data model

When generating a UML data model, the Reverse Engineering tool looks for tables in the project. The UML model contains a class for each table and view in the project and its attributes and associations. Figure 2-19 shows a class diagram with the *CustTable* (customers), *InventTable* (inventory items), *SalesTable* (sales order header), and *SalesLine* (sales order line) tables. To simplify the diagram, some attributes have been removed.

The UML model also contains referenced tables and all extended data types, base enumerations, and X++ data types. You can include these items in your diagrams without having to run the Reverse Engineering tool again.

Fields in Microsoft Dynamics AX are generated as UML attributes. All attributes are marked as public to reflect the nature of fields in Microsoft Dynamics AX. Each attribute also shows the type. The primary key field is underlined. If a field is a part of one or more indexes, the field name is prefixed with the names of the indexes; if the index is unique, the index name is noted in brackets.

Relationships in Microsoft Dynamics AX are generated as UML associations. The *Aggregation* property of the association is set based on two conditions in metadata:

- If the relationship is validating (the *Validate* property is set to *Yes*), the *Aggregation* property is set to *Shared*. This is also known as a UML aggregation, represented by a white diamond.
- If a cascading delete action exists between the two tables, a composite association is added to the model. A cascading delete action ties the lifespan of two or more tables and is represented by a black diamond.

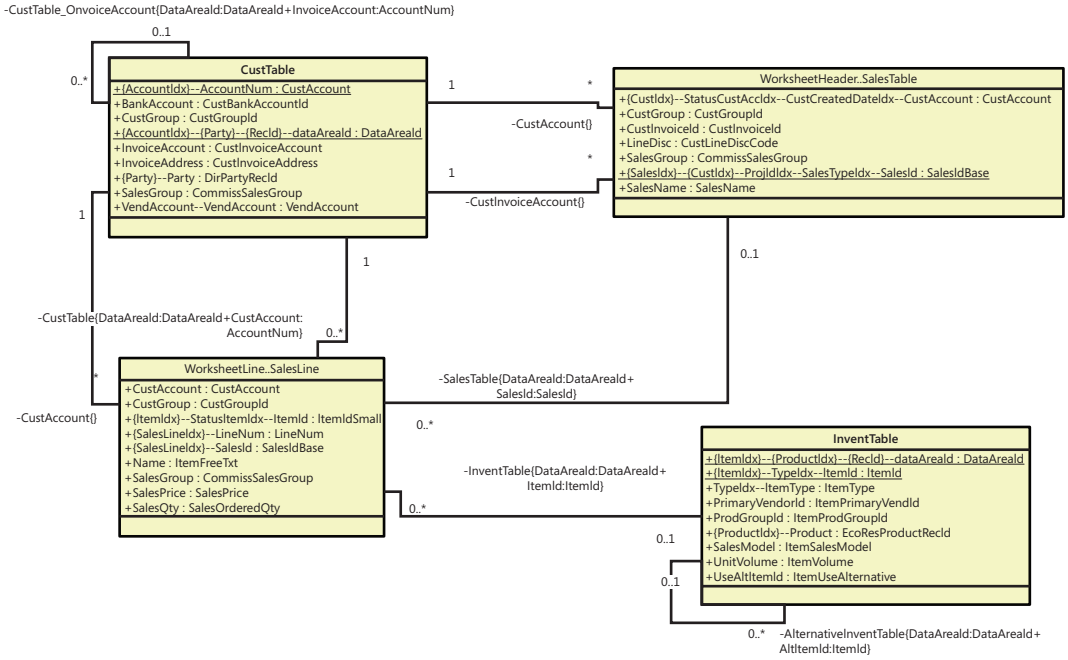

FIGURE 2-19 UML data model diagram.

The name of an association endpoint is the name of the Microsoft Dynamics AX relationship. The names and types of all fields in the relationship appear in brackets.

UML object model

When generating an object model, the Reverse Engineering tool looks for Microsoft Dynamics AX classes, tables, and interfaces in the project. The UML model contains a class for each Microsoft Dynamics AX table and class in the project and an interface for each Microsoft Dynamics AX interface in the project. The UML model also contains attributes and operations, including return types, parameters, and the types of the parameters. Figure 2-20 shows an object model of the most important *RunBase* and *Batch* classes and interfaces in Microsoft Dynamics AX. To simplify the view, some attributes and operations have been removed and operation parameters are suppressed.

The UML model also contains referenced classes, tables, and all extended data types, base enumerations, and X++ data types. You can include these elements in your diagrams without having to run the Reverse Engineering tool again.

Fields and member variables in Microsoft Dynamics AX are generated as UML attributes. All fields are generated as public attributes, whereas member variables are generated as protected attributes. Each attribute also shows the type. Methods are generated as UML operations, including return type, parameters, and the types of the parameters.

The Reverse Engineering tool also picks up any generalizations (classes extending other classes), realizations (classes implementing interfaces), and associations (classes using each other). The associations are limited to references in member variables.

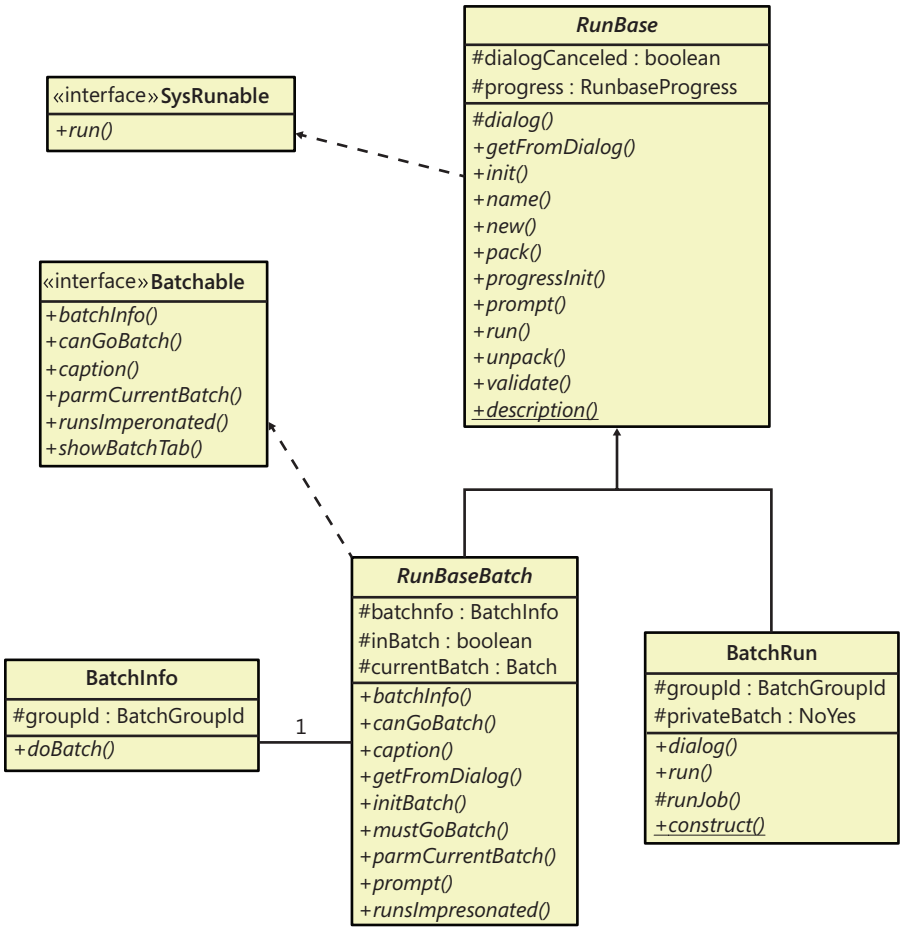

FIGURE 2-20 UML object model diagram.

Note To get the names of operation parameters, you must reverse-engineer in debug mode. The names are read from metadata only and placed into the stack when in debug mode. To enable debug mode, on the Development tab of the Options form, select When Breakpoint in the Debug Mode list.

Entity relationship data model

When generating an entity relationship data model, the Reverse Engineering tool looks for tables and views in the project. The entity relationship model contains an entity type for each AOT table in the project and attributes for the fields in each table. Figure 2-21 shows an Entity Relationship Diagram (ERD) for the tables *HcmBenefit (Benefit)*, *HcmBenefitOption (Benefit option), HcmBenefitType (Benefit type,)* and *HcmBenefitPlan (Benefit plan)*.

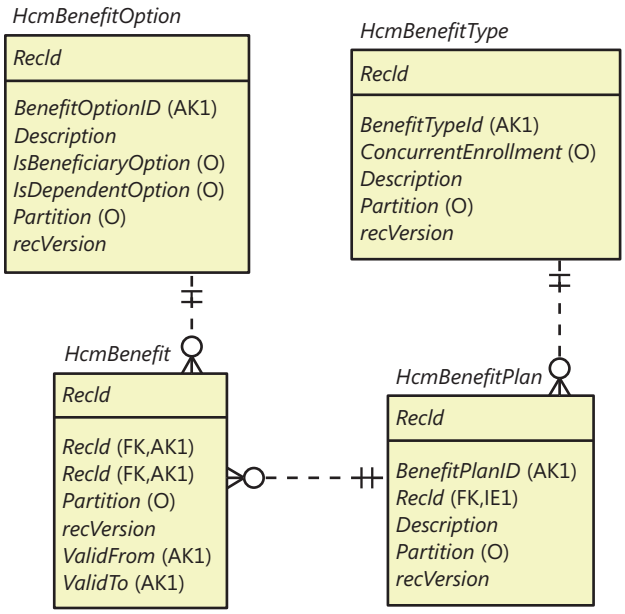

FIGURE 2-21 ERD using IDEF1X notation.

Fields in Microsoft Dynamics AX are generated as entity relationship columns. Columns can be foreign key (FK), alternate key (AK), inversion entry (IE), and optional (O). A foreign key column is used to identify a record in another table, an alternate key uniquely identifies a record in the current table, an inversion entry identifies zero or more records in the current table (these are typical of the fields in nonunique indexes), and optional columns don't require a value.

Relationships in Microsoft Dynamics AX are generated as entity relationships. The *EntityRelationshipRole* property of the relationship in Microsoft Dynamics AX is used as the foreign key role name of the relation in the entity relationship data model.

Note The Reverse Engineering tool produces an ERX file. To work with the generated file in Visio, do the following: In Visio, create a new Database Model Diagram, and then, on the select Database menu, point to Import > Import Erwin ERX file. Afterward, you can drag relevant tables from the Tables And Views pane (available from the Database menu) to the diagram canvas.

Table Browser tool

The Table Browser tool is a small, helpful tool that can be used in numerous scenarios. You can browse and maintain the records in a table without having to build a user interface. This tool is useful when you're debugging, validating data models, and modifying or cleaning up data, to name just a few uses.

To access the Table Browser tool, right-click any of the following types of items in the AOT, and then point to Add-Ins > Table Browser:

- Tables
- Tables listed as data sources in forms, queries, and data sets
- System tables listed in the AOT under System Documentation\Tables

Note The Table Browser tool is implemented in X++. You can find it in the AOT under the name *SysTableBrowser*. It is a good example of how to bind the data source to a table at run time.

Figure 2-22 shows the Table Browser tool when started from the *CustTrans* table. In addition to the querying, sorting, and filtering capabilities provided by the grid control, you can type an SQL *SELECT* statement directly into the form using X++ *SELECT* statement syntax and see a visual display of the result set. This tool is a great way to test complex *SELECT* statements. It fully supports grouping, sorting, aggregation, and field lists.

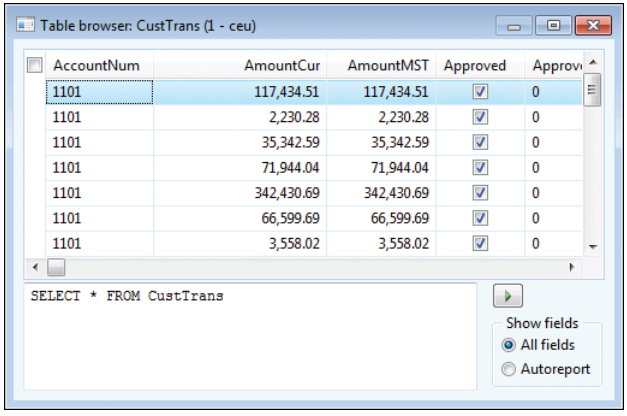

FIGURE 2-22 The Table Browser tool showing the contents of the *CustTrans* table demo data.

You can also choose to see only the fields from the auto-report field group. These fields are printed in a report when the user clicks Print in a form with this table as a data source. Typically, these fields hold the most interesting information. This option can make it easier to find the values you're looking for in tables with many fields.

Note The Table Browser tool is just a standard form that uses IntelliMorph. It can't display fields for which the *visible* property is set to *No* or fields that the current user doesn't have access to.

Find tool

Search is everything, and the size of Microsoft Dynamics AX applications calls for a powerful and effective search tool.

Tip You can use the Find tool to search for an example of how to use an API. Real examples can complement the examples found in the documentation.

You can start the Find tool, shown in Figure 2-23, from any node in the AOT by pressing Ctrl+F or by clicking Find on the context menu. The Find tool supports multiple selections in the AOT.

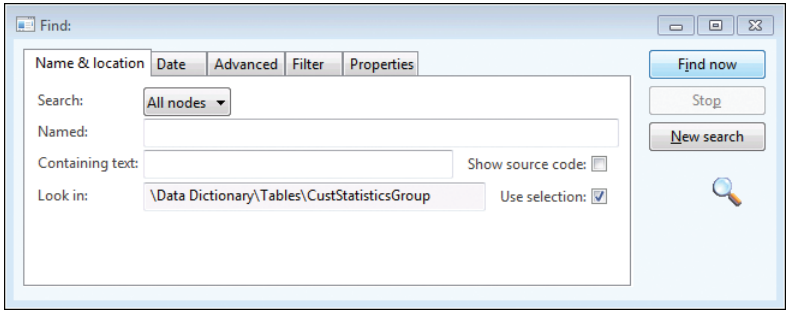

FIGURE 2-23 The Find tool.

On the Name & Location tab, you define what you're searching for and where to look:

- In Search, the menu options are Methods and All Nodes. If you choose All Nodes, the Properties tab appears.
- The Named box limits the search to nodes with the name you specify.
- The Containing box specifies the text to look for in the method, expressed as a regular expression.
- If you select the Show Source Code check box, results include a snippet of source code containing the match, making it easier to browse the results.

By default, the Find tool searches the node (and its subnodes) selected in the AOT. If you change focus in the AOT while the Find tool is open, the Look In value is updated. This is quite useful if you want to search several nodes using the same criterion. You can disable this behavior by clearing the Use Selection check box.

On the Date tab, you specify additional ranges for your search, such as Modified Date and Modified By.

On the Advanced tab, you can specify more advanced settings for your search, such as the layer to search, the size range of elements, the type of element, and the tier on which the element is set to run.

On the Filter tab, shown in Figure 2-24, you can write a more complex query by using X++ and type libraries. The code in the Source text box is the body of a method with the following profile:

```
boolean FilterMethod(str _treeNodeName,
                     str _treeNodeSource,
                     XRefPath _path,
                     ClassRunMode _runMode)
```

The example in Figure 2-24 uses the class *SysScannerClass* to find any occurrence of the *ttsAbort* X++ keyword. The scanner is primarily used to pass tokens into the parser during compilation. Here, however, it detects the use of a particular keyword. This tool is more accurate (though slower) than using a regular expression because X++ comments don't produce tokens.

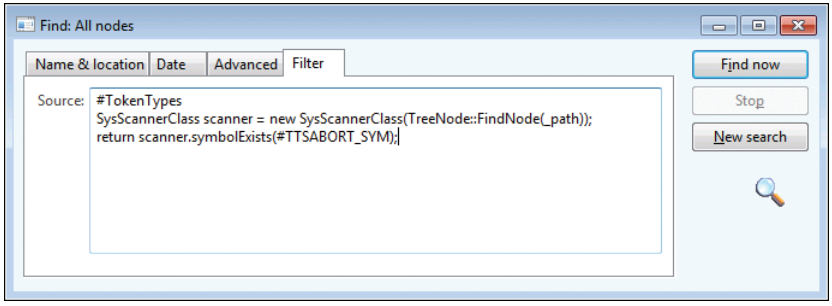

FIGURE 2-24 Filtering in the Find tool.

The Properties tab appears when All Nodes is selected in the Search list. You can specify a search range for any property. Leaving the range blank for a property is a powerful setting when you want to inspect properties: it matches all nodes, and the property value is added as a column in the results, as shown in Figure 2-25. The search begins when you click Find Now. The results appear at the bottom of the dialog box as they are found.

Double-clicking any line in the result set opens the X++ code editor and sets the focus on the code example that matches. When you right-click the lines in the result set, a context menu containing the Add-Ins menu opens.

Compare tool

Several versions of the same element typically exist. These versions might emanate from various layers or revisions in version control, or they could be modified versions that exist in memory. Microsoft Dynamics AX has a built-in Compare tool that highlights any differences between two versions of an element.

The comparison shows changes to elements, which can be modified in three ways:

- A metadata property can be changed.
- X++ code can be changed.
- The order of subnodes can be changed, such as the order of tabs on a form.

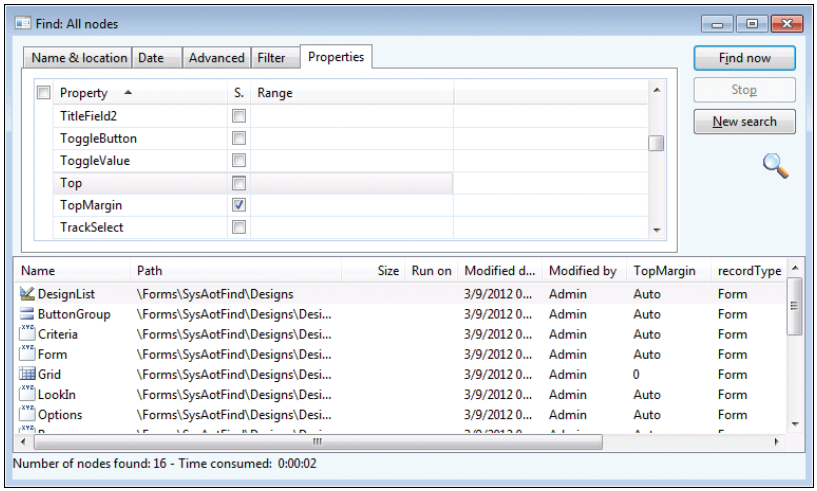

FIGURE 2-25 Search results in the Find tool.

Start the Compare tool

To open the Compare tool, right-click an element, and then click Compare. A dialog box opens where you can select the versions of the element you want to compare, as shown in Figure 2-26.

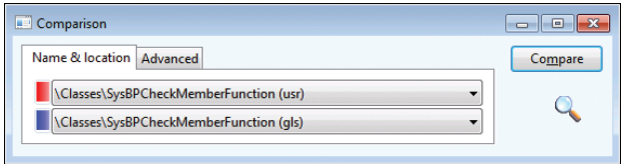

FIGURE 2-26 The Comparison dialog box.

The versions to choose from come from many sources. The following is a list of all possible types of versions:

- **Standard layered version types** These include *SYS*, *SYP*, *GLS*, *GLP*, *FPK*, *FPP*, *SLN*, *SLP*, *ISV*, *ISP*, *VAR*, *VAP*, *CUS*, *CUP*, *USR*, and *USP*.
- **Old layered version types (old *SYS*, old *SYP*, and so on)** If a baseline model store is present, elements from the files are available here. This allows you to compare an older version of an element with its latest version. For more information about layers and the baseline model store, see Chapter 21.

- **Version control revisions (Version 1, Version 2, and so on)** You can retrieve any revision of an element from the version control system individually and use it for comparison. The version control system is explained later in this chapter.
- **Best practice washed version (Washed)** A few simple best practice issues can be resolved automatically by a best practice "wash." Selecting the washed version shows you how your implementation differs from best practices. To get the full benefit of this, select the Case Sensitive check box on the Advanced tab.
- **Export/import file (XPO)** Before you import elements, you can compare them with existing elements (which will be overwritten during import). You can use the Compare tool during the import process (Command > Import) by selecting the Show Details check box in the Import dialog box and right-clicking any elements that appear in bold. Objects in bold already exist in the application.
- **Upgraded version (Upgraded)** MorphX can automatically create a proposal for how a class should be upgraded. The requirement for upgrading a class arises during a version upgrade. The Create Upgrade Project step in the Upgrade Checklist automatically detects customized classes that conflict with new versions of the classes. A class is conflicting if you've changed the original version of the class, and the publisher of the class has also changed the original version. MorphX constructs the proposal by merging your changes with the publisher's changes to the class. MorphX requires access to all three versions of the class—the original version in the baseline model store, a version with your changes in the current layer in the baseline model store, and a version with the publisher's changes in the same layer as the original. The installation program ensures that the right versions are available in the right places during an upgrade. Conflict resolution is shown in Figure 2-27.

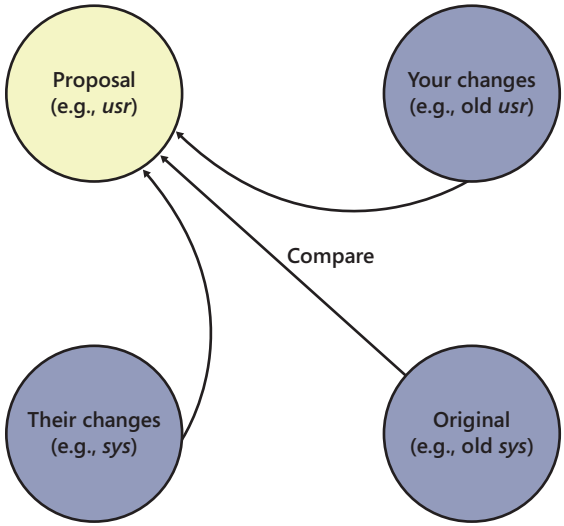

FIGURE 2-27 How the upgraded version proposal is created.

Note You can also compare two different elements. To do this, select two elements in the AOT, right-click, point to Add-Ins, and then click Compare.

Figure 2-28 shows the Advanced tab, on which you can specify comparison options.

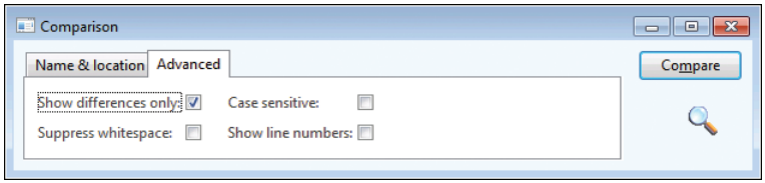

FIGURE 2-28 Comparison options on the Advanced tab.

The following list describes the comparison options shown in Figure 2-28:

- **Show Differences Only** All equal nodes are suppressed from the view, making it easier to find the changed nodes. This option is selected by default.
- **Suppress Whitespace** White space, such as spaces and tabs, is suppressed into a single space during the comparison. The Compare tool can ignore the amount of white space, just as the compiler does. This option is selected by default.
- **Case Sensitive** Because X++ is not case sensitive, the Compare tool is also not case sensitive by default. In certain scenarios, case sensitivity is required and must be enabled, such as when you're using the best practice wash feature mentioned earlier in this section. This option is cleared by default.
- **Show Line Numbers** The Compare tool can add line numbers to all X++ code that is displayed. This option is cleared by default but can be useful during an upgrade of large chunks of code.

Use the Compare tool

After you choose elements and set parameters, start the comparison by clicking Compare. Results are displayed in a three-pane dialog box, as shown in Figure 2-29. The top pane contains the elements and options that you selected, the left pane displays a tree structure resembling the AOT, and the right pane shows details that correspond to the item selected in the tree.

Color-coded icons in the tree structure indicate how each node has changed. A red or blue check mark indicates that the node exists only in a particular version. Red corresponds to the *SYS* layer, and blue corresponds to the old *SYS* layer. A gray check mark indicates that the nodes are identical but one or more subnodes are different. A not-equal-to symbol (≠) on a red and blue background indicates that the nodes are different in the two versions.

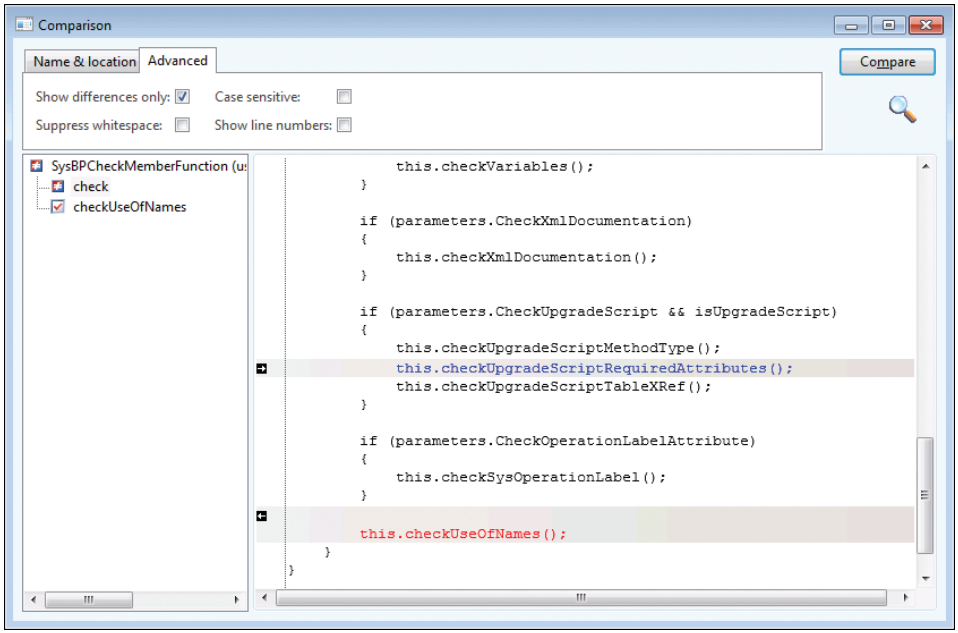

FIGURE 2-29 Comparison results.

Note Each node in the tree view has a context menu that provides access to the Add-Ins submenu and the Open New Window option. The Open New Window option provides an AOT view of any element, including elements in old layers.

Details about the differences are shown in the right pane. Color coding is also used in this pane to highlight differences the same way that it is in the tree structure. If an element is editable, small action icons appear. These icons allow you to make changes to code, metadata, and nodes, which can save you time when performing an upgrade. A right or left arrow removes or adds the difference, and a bent arrow moves the difference to another position. These arrows always come in pairs, so you can see where the difference is moved to and from. If a version control system is in use, an element is editable if it is from the current layer and is checked out.

Compare APIs

Although Microsoft Dynamics AX provides the comparison functionality for development purposes only, the comparison functionality can be reused for other tasks. You can use the available APIs to compare and present differences in the tree structure or text representation of any type of entity.

The *Tutorial_CompareContextProvider* class shows how simple it is to compare business data by using these APIs and present it by using the Compare tool. The tutorial consists of two parts:

- ***Tutorial_Comparable*** This class implements the *SysComparable* interface. Basically, it creates a text representation of a customer.

- ***Tutorial_CompareContextProvider*** This class implements the *SysCompareContextProvider* interface. It provides the context for comparison. For example, it creates a *Tutorial_Comparable* object for each customer, sets the default comparison options, and handles context menus.

Figure 2-30 shows a comparison of two customers, the result of running the tutorial.

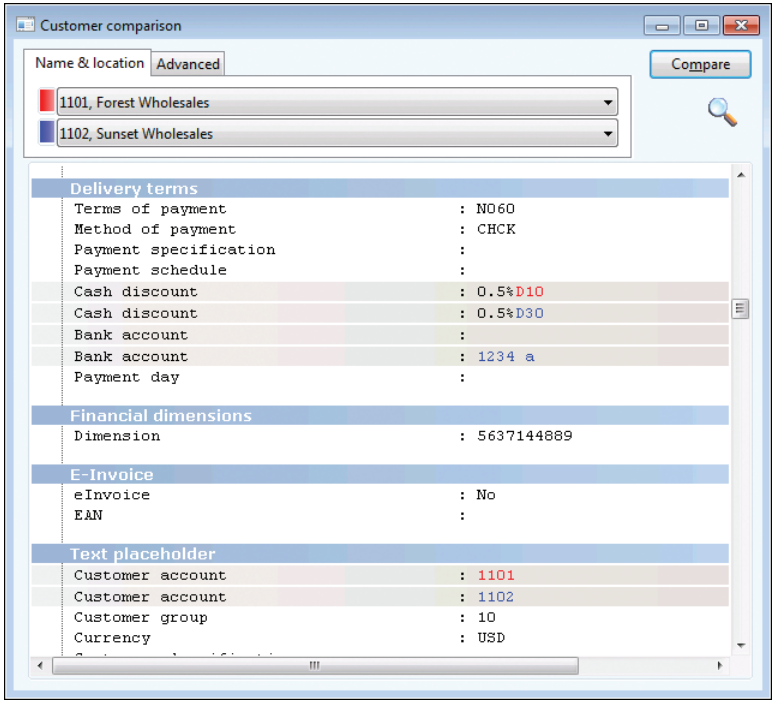

FIGURE 2-30 The result of comparing two customers using the Compare API.

You can also use the line-by-line comparison functionality directly in X++. The static *run* method on the *SysCompareText* class, shown in the following code, takes two strings as parameters and returns a container that highlights differences in the two strings. You can also use a set of optional parameters to control the comparison.

```
public static container run(str _t1,
      str _t2,
      boolean _caseSensitive = false,
      boolean _suppressWhiteSpace = true,
      boolean _lineNumbers  = false,
      boolean _singleLine  = false,
       boolean _alternateLines = false)
```

Cross-Reference tool

The concept of cross-references in Microsoft Dynamics AX is simple. If an element uses another element, the reference is recorded. With cross-references, you can determine which elements a particular element uses and which elements other elements are using. Microsoft Dynamics AX provides the Cross-Reference tool for accessing and managing cross-reference information.

Here are a couple of typical scenarios for using the Cross-Reference tool:

- You want to find usage examples. If the product documentation doesn't help, you can use the Cross-Reference tool to find real implementation examples.
- You need to perform an impact analysis. If you're changing an element, you need to know which other elements are affected by your change.

You must update the Cross-Reference tool regularly to ensure accuracy. The update typically takes several hours. The footprint in a database is about 1.5 GB for a standard application.

To update the Cross-Reference tool, on the Tools menu, point to > Cross-Reference > Periodic > Update. Updating the Cross-Reference tool also compiles the entire AOT because the compiler emits cross-reference information.

Tip Keeping the Cross-Reference tool up to date is important if you want its information to be reliable. If you work in a shared development environment, you share cross-Reference information with your team members. Updating the Cross-Reference tool nightly is a good approach for a shared environment. If you work in a local development environment, you can keep the Cross-Reference tool up to date by enabling cross-referencing when compiling. This option slows down compilation, however. Another option is to update cross-references manually for the elements in a project. To do so, right-click the project and point to Add-Ins > Cross-Reference > Update.

In addition to the main cross-reference information, two smaller cross-reference subsystems exist:

- **Data model** Stores information about relationships between tables. It is primarily used by the query form and the Reverse Engineering tool.
- **Type hierarchy** Stores information about class and data type inheritance.

For more information about these subsystems and the tools that rely on them, see the Microsoft Dynamics AX 2012 SDK (*http://msdn.microsoft.com/en-us/library/aa496079.aspx*).

The cross-reference information the Cross-Reference tool collects is quite comprehensive. You can find the complete list of cross-referenced elements by opening the AOT, expanding the *System Documentation* node, and clicking Enums and then xRefKind.

When the Cross-Reference tool is updating, it scans all metadata and X++ code for references to elements of the kinds listed here.

Tip It's a good idea to use intrinsic functions when referring to elements in X++ code. An intrinsic function can evaluate to either an element name or an ID. The intrinsic functions are named <Element type>*Str* or <Element type>*Num*, respectively. Using intrinsic functions provides two benefits: you have compile-time verification that the element you reference actually exists, and the reference is picked up by the Cross-Reference tool. Also, there is no run-time overhead. An example follows:

```
// Prints ID of MyClass, such as 50001
print classNum(myClass);

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

// No compile check or cross-reference
print "MyClass";
```

For more information about intrinsic functions, see Chapter 20, "Reflection."

To access usage information, right-click any element in the AOT and point to Add-Ins > Cross-Reference > Used By. If the option isn't available, either the element isn't used or the cross-reference hasn't been updated.

Figure 2-31 shows where the *prompt* method is used on the *RunBaseBatch* class.

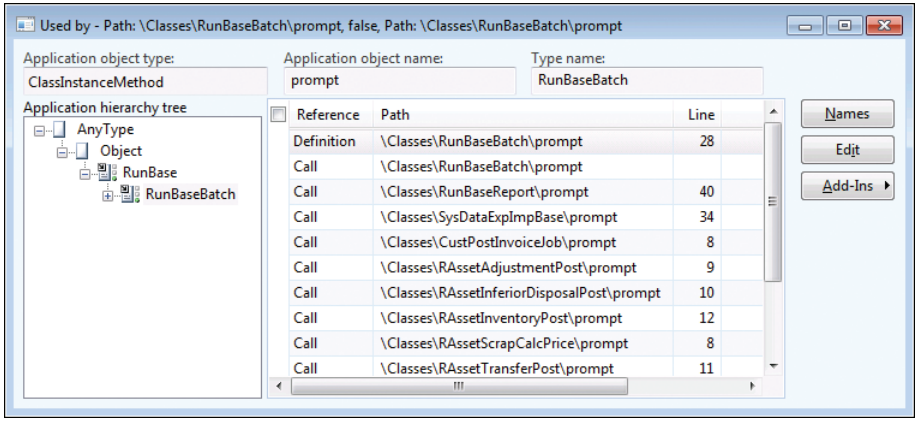

FIGURE 2-31 The Cross-Reference tool, showing where *RunBaseBatch.prompt* is used

When you view cross-references for a class method, the Application hierarchy tree is visible, so that you can see whether the same method is used on a parent or subclass. For types that don't support inheritance, the Application hierarchy tree is hidden.

Version control

The Version Control tool feature in MorphX makes it possible to use a version control system, such as Microsoft Visual SourceSafe (VSS) or Visual Studio Team Foundation Server (TFS), to keep track of changes to elements in the AOT. The tool is accessible from several places: from the Version Control menu in the Development Workspace, from toolbars in the AOT and X++ code editor, and from the context menu on elements in the AOT.

Using a version control system offers several benefits:

- **Revision history of all elements** All changes are captured, along with a description of the change, making it possible to consult the change history and retrieve old versions of an element.
- **Code quality enforcement** The implementation of version control in Microsoft Dynamics AX enables a fully configurable quality standard for all check-ins. With the quality standard, all changes are verified according to coding practices. If a change doesn't meet the criteria, it is rejected.
- **Isolated development** Each developer can have a local installation and make all modifications locally. When modifications are ready, they can be checked in and made available to consumers of the build. A developer can rewrite fundamental areas of the system without causing instability issues for others. Developers are also unaffected by any downtime of a centralized development server.

Even though using a version control system is optional, it is strongly recommended that you consider one for any development project. Microsoft Dynamics AX 2012 supports three version control systems: VSS 6.0 and TFS, which are designed for large development projects, and MorphX VCS. MorphX VCS is designed for smaller development projects that previously couldn't justify the additional overhead that using a version control system server adds to the process. Table 2-8 shows a side-by-side comparison of the version control system options.

TABLE 2-8 Overview of version control systems.

	No version control system	MorphX VCS	VSS	TFS
Application Object Servers required	1	1	1 for each developer	1 for each developer
Database servers required	1	1	1 for each developer	1 for each developer
Build process required	No	No	Yes	Yes
Master file	Model store	Model store	XPOs	XPOs
Isolated development	No	No	Yes	Yes
Multiple checkout	N/A	No	Configurable	Configurable
Change description	No	Yes	Yes	Yes

	No version control system	MorphX VCS	VSS	TFS
Change history	No	Yes	Yes	Yes
Change list support (atomic check-in of a set of files)	N/A	No	No	Yes
Code quality enforcement	No	Configurable	Configurable	Configurable

The elements persisted on the version control server are file representations of the elements in the AOT. The file format used is the standard Microsoft Dynamics AX export format (.xpo). Each .xpo file contains only one root element.

There are no additional infrastructure requirements when you use MorphX VCS, which makes it a perfect fit for partners running many parallel projects. In such setups, each developer often works simultaneously on several projects, toggling between projects and returning to past projects. In these situations, the benefits of having a change history are enormous. With just a few clicks, you can enable MorphX VCS to persist the a changes in the business database. Although MorphX VCS provides many of the same capabilities as a version control server, it has some limitations. For example, MorphX VCS does not provide any tools for maintenance, such as making backups, archiving, or labeling.

In contrast, VSS and TFS are designed for large projects in which many developers work together on the same project for an extended period of time (for example, an independent software vendor building a vertical solution).

Figure 2-32 shows a typical deployment using VSS or TFS, in which each developer locally hosts the AOS and the database. Each developer also needs a copy of all .xpo files. When a developer communicates with the version control server, the .xpo files are transmitted.

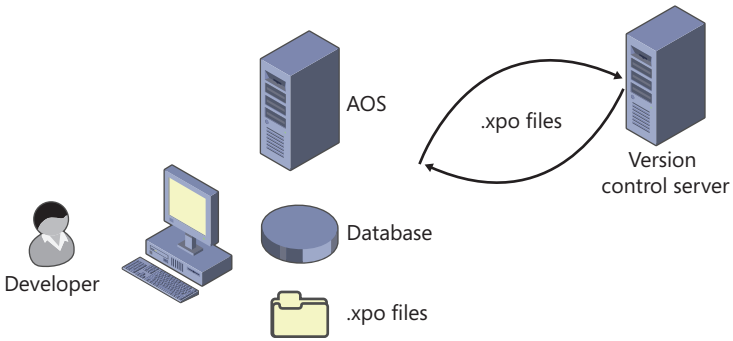

FIGURE 2-32 Typical deployment using version control.

Note In earlier versions of Microsoft Dynamics AX, a Team Server was required to assign unique IDs as elements were created. Microsoft Dynamics AX 2012 uses a new ID allocation scheme, which eliminates the need for the Team Server. For more information element IDs, see Chapter 21.

Element life cycle

Figure 2-33 shows the element life cycle in a version control system. When an element is in a state marked with a lighter shade, it can be edited; otherwise, it is read-only.

You can create an element in two ways:

- Create a new element.
- Customize an existing element, resulting in an *overlayered* version of the element. Because elements are stored for each layer in the version control system, customizing an element effectively creates a new element.

After you create an element, you must add it to the version control system. First, give it a proper name in accordance with naming conventions, and then click Add To Version Control on the context menu. After you create the element, you must check it in.

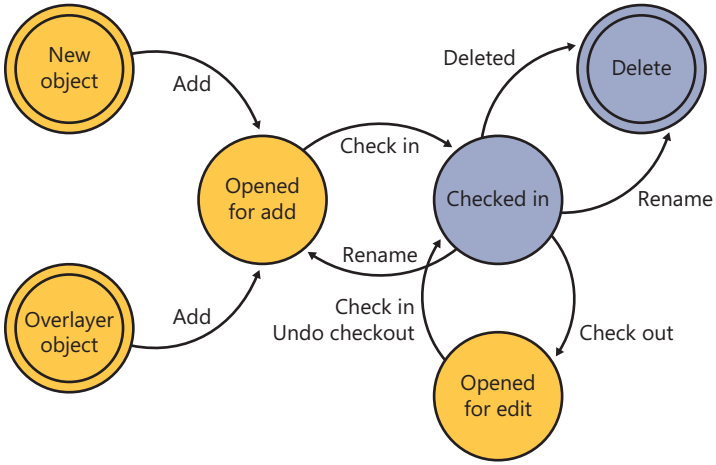

FIGURE 2-33 Element life cycle.

An element that is checked in can be renamed. Renaming an element deletes the element with the old name and adds an element with the new name.

Quality checks

Before the version control system accepts a check-in, it might subject the elements to quality checks. You define what is accepted in a check-in when you set up the version control system. The following checks are supported:

- Compiler errors
- Compiler warnings
- Compiler tasks
- Best practice errors

When a check is enabled, it is carried out when you do a check-in. If the check fails, the check-in stops. You must address the issue and restart the check-in.

Source code casing

You can set the Source Code Title Case Update tool, available on the Add-Ins submenu, to execute automatically before elements are checked in to ensure uniform casing in variable and parameter declarations and references. You can specify this parameter when setting up the version control system by selecting the Run Title Case Update check box.

Common version control tasks

Table 2-9 describes some of the tasks that are typically performed with a version control system. Later sections describe additional tasks that you can perform when using version control with Microsoft Dynamics AX.

TABLE 2-9 Version control tasks.

Action	Description
Check out an element	To modify an element, you must check it out. Checking out an element locks it so that others can't modify it while you're working. To see which elements you have currently checked out, on the Microsoft Dynamics AX menu, click Control > Pending Objects. The elements you've checked out (or that you've created and not yet checked in), appear in blue, rather than black, in the AOT.
Undo a checkout	If you decide that you don't want to modify an element that you checked out, you can undo the checkout. This releases your lock on the element and imports the most recent checked-in revision of the element to undo your changes.
Check in an element	When you have finalized your modifications, you must check in the elements for them to be part of the next build. When you click Check-In on the context menu, the dialog box shown in Figure 2-34 appears, displaying all the elements that you currently have checked out. The Check In dialog box shows all open elements by default; you can remove any elements not required in the check-in from the list by pressing Alt+F9.

	The following procedure is recommended for checking in your work: ■ Perform synchronization to update all elements in your environment to the latest version. ■ Verify that everything is still working as intended. Compilation is not enough. ■ Check in the elements.
Create an element	When using version control, you create new elements just as you normally would in the MorphX environment without a version control system. These elements are not part of your check-in until you click Add To Version Control on the context menu. You can also create all element types except those listed in System Settings (on the Development Workspace Version Control menu, point to Control > Setup > System Settings). By default, jobs and private projects are not accepted. New elements should follow Microsoft Dynamics AX naming conventions. The best practice naming conventions are enforced by default, so you can't check in elements with names such as *aaaElement*, *DEL_Element*, *element1*, or *element2*. (The only *DEL_ elements* allowed are those required for version upgrade purposes.) You can change naming requirements in System Settings.
Rename an element	An element must be checked in to be renamed. Because all references between .xpo files are strictly name-based, all references to renamed elements must be updated. For example, if you rename a table field, you must also update any form or report that uses that field. Most references in metadata in the AOT are ID-based and thus, they are not affected when an element is renamed; in most cases, it is enough to check out the form or report and include it in the check-in to update the .xpo file. You can use the cross-reference tool to identify references. References in X++ code are name-based. You can use the compiler to find affected references. An element's revision history is kept intact when elements are renamed. No tracking information in the version control system is lost because of an element is renamed.
Delete an element	You delete an element as you normally would in Microsoft Dynamics AX. The delete operation must be checked in before the deletion is visible to other users of the version control system. You can see pending deletions in the Pending Objects dialog box.
Get the latest version of an element	If someone else has checked in a new version of an element, you can use the Get Latest option on the context menu to get the version of the element that was checked in most recently. This option isn't available if you have the element checked out yourself. Get Latest is not available with MorphX VCS.

Figure 2-34 shows the Check In dialog box.

Work with labels

Working with labels is similar to working with elements. To change, delete, or add a label, you must check out the label file containing the label. You can check out the label file from the Label editor dialog box.

The main difference between checking out elements and checking out label files is that simultaneous checkouts are allowed for label files. This means that others can change labels while you have a label file checked out.

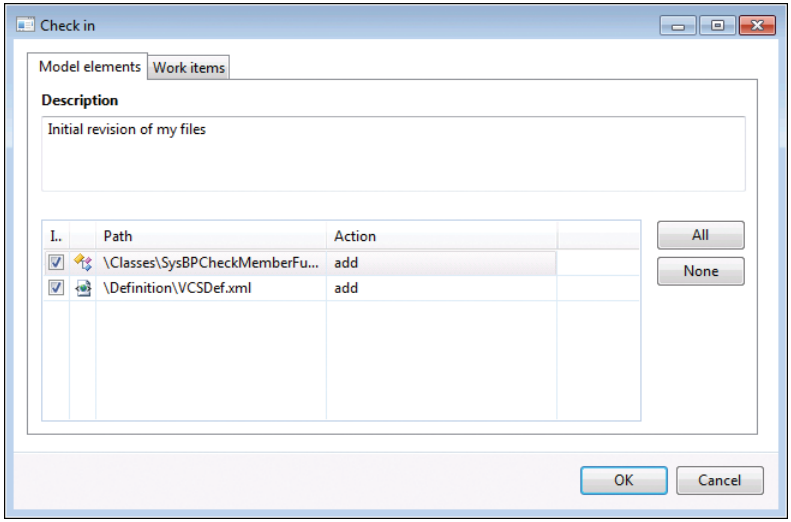

FIGURE 2-34 The Check In dialog box.

If you create a new label when using version control, a temporary label ID is assigned (for example, @$AA0007 as opposed to @USR1921). When you check in a label file, your changes are automatically merged into the latest version of the file and the temporary label IDs are updated. All references in the code are automatically updated to the newly assigned label IDs. Temporary IDs eliminate the need for a central Team Server, which was required for Microsoft Dynamics AX 2009, because IDs no longer have to be assigned when the labels are created. If you modify or delete a label that another person has also modified or deleted, your conflicting changes are abandoned. Such lost changes are shown in the Infolog after the check-in completes.

Synchronize elements

Synchronization makes it possible for you to get the latest version of all elements. This step is required before you can check in any elements. You can initiate synchronization from the Development Workspace. On the Version Control menu, point to Periodic > Synchronize.

Synchronization is divided into three operations that happen automatically in the following sequence:

1. Copy the latest files from the version control server to the local disk.
2. Import the files into the AOT.
3. Compile the imported files.

Use synchronization to make sure your system is up to date. Synchronization won't affect any new elements that you have created or any elements that you have checked out.

Figure 2-35 shows the Synchronization dialog box.

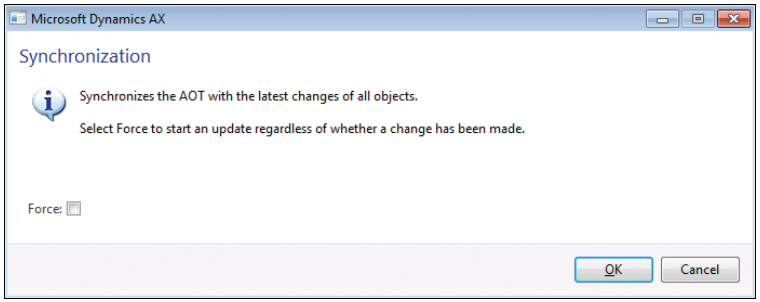

FIGURE 2-35 The Synchronization dialog box.

Selecting the Force check box gets the latest version of all files, even if they haven't changed, and then imports every file.

When using VSS, you can also synchronize to a label defined in VSS. This way, you can easily synchronize to a specific build or version number.

Synchronization is not available with MorphX VCS.

View the synchronization log

The way that you keep track of versions on the client depends on your version control system. VSS requires that Microsoft Dynamics AX keep track of itself. When you synchronize the latest version, it is copied to the local repository folder from the version control system. Each file must be imported into Microsoft Dynamics AX to be reflected in the AOT. To minimize the risk of partial synchronization, a log entry is created for each file. When all files are copied locally, the log is processed, and the files are automatically imported into Microsoft Dynamics AX.

When synchronization fails, the import operation is usually the cause of the problem. Synchronization failure leaves your system in a partially synchronized state. To complete the synchronization, restart Microsoft Dynamics AX and restart the import. You use the synchronization log to restart the import, and you access it from the Development Workspace menu at Version Control > Inquiries > Synchronization log.

The Synchronization Log dialog box, shown in Figure 2-36, displays each batch of files, and you can restart the import operation by clicking Process. If the Completed check box is not selected, the import has failed and should be restarted.

The Synchronization log is not available with MorphX VCS.

Show the history of an element

One of the biggest advantages of version control is the ability to track changes to elements. Selecting History on an element's context menu displays a list of all changes to an element, as shown in Figure 2-37.

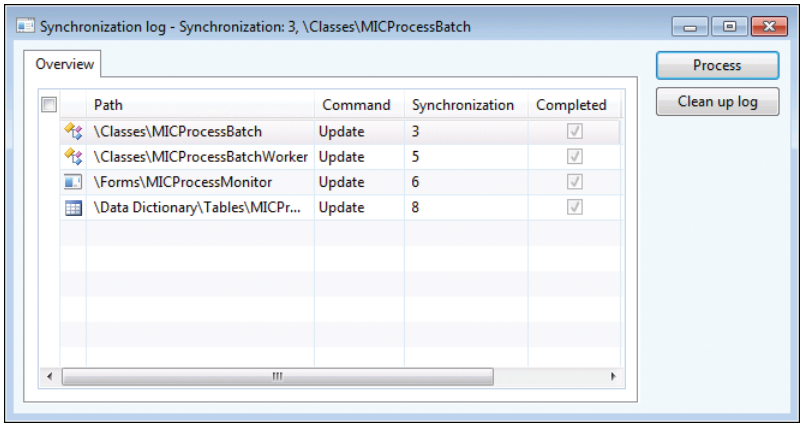

FIGURE 2-36 The Synchronization Log dialog box.

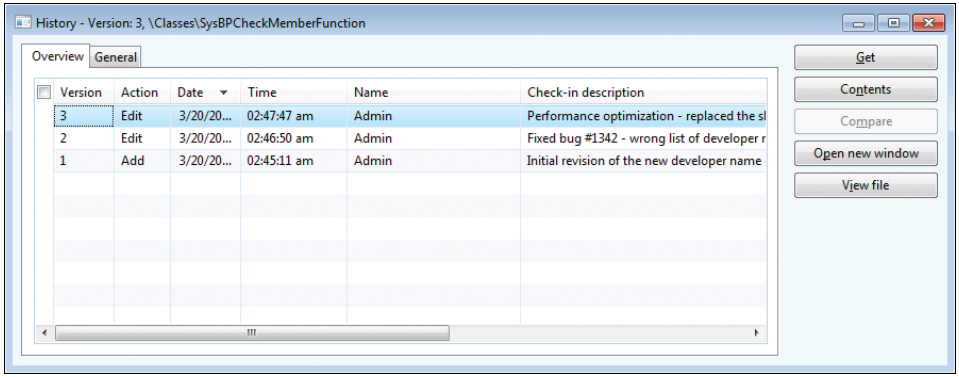

FIGURE 2-37 Revision history of an element.

This dialog box shows the version number, the action performed, the time the action was performed, and who performed the action. You can also see the change number and the change description.

A set of buttons in the History dialog box allows further investigation of each version. Clicking Contents opens a form that shows other elements included in the same change. Clicking Compare opens the Compare dialog box, where you can do a line-by-line comparison of two versions of the element. The Open New Window button opens an AOT window that shows the selected version of the element, which is useful for investigating properties because you can use the standard MorphX toolbox. Clicking View File opens the .xpo file for the selected version in Notepad.

Compare revisions

Comparison is the key to harvesting the benefits of a version control system. You can start a comparison from several places, including from the context menu of an element by pointing to Compare. Figure 2-38 shows the Comparison dialog box, where two revisions of the form *CustTable* are selected.

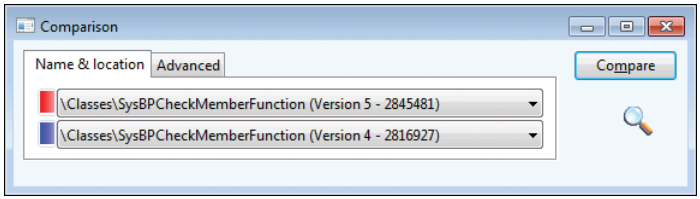

FIGURE 2-38 Comparing element revisions from version control.

The Compare dialog box contains a list of all checked-in versions, in addition to the element versions available in other layers installed.

View pending elements

When you're working on a project, it's easy to lose track of which elements you've opened for editing. The Pending Objects dialog box, shown in Figure 2-39, lists the elements that are currently checked out in the version control system. Notice the column containing the action performed on the element. Deleted elements are available only in this dialog box; they are no longer shown in the AOT.

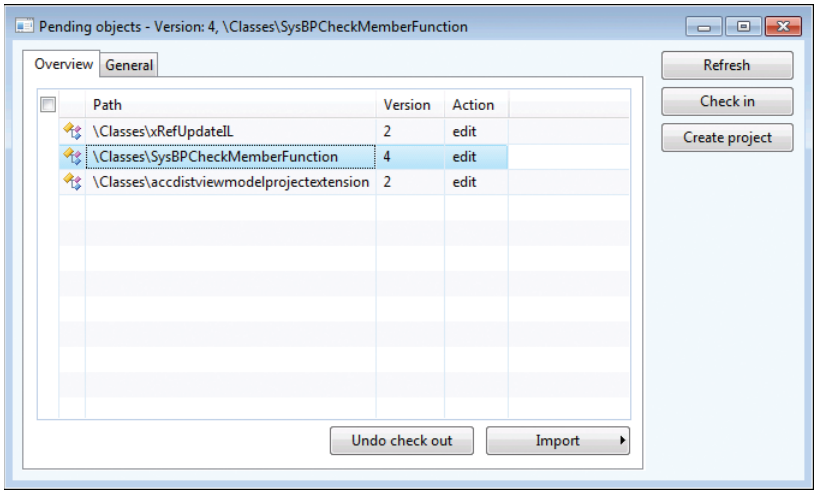

FIGURE 2-39 Pending elements.

You can access the Pending Objects dialog box from the Development Workspace menu: Version Control > Pending Objects.

Create a build

Because the version control system contains .xpo files and not a model file, a build process is required to generate the model file from the .xpo files. The following procedure provides a high-level overview of the build process:

1. Use the CombineXPOs command-line utility to combine all .xpo files into one. This step makes the .xpo file consumable by Microsoft Dynamics AX. Microsoft Dynamics AX requires all referenced elements to be present in the .xpo file or to already exist in the AOT to maintain the references during import.
2. Import the new .xpo file by using the command-line parameter -AOTIMPORTFILE=<*FileNam*e. xpo> -MODEL=<*Model Name*> to Ax32.exe. This step imports the .xpo file and compiles everything. After this step is complete, the new model is ready in the model store.
3. Export the model to a file by using the axutil command-line utility: axutil export /model:<*model name*> /file:<*model file name*>.
4. Follow these steps for each layer and each model that you build.

The build process doesn't apply to MorphX VCS.

Integrate Microsoft Dynamics AX with other version control systems

The implementation of the version control system in Microsoft Dynamics AX is fully pluggable. This means that any version control system can be integrated with Microsoft Dynamics AX.

Integrating with another version control system requires a new class implementing the *SysVersionControlFileBasedBackEnd* interface. It is the implementation's responsibility to provide the communication with the version control system server being used.

CHAPTER 3

Microsoft Dynamics AX and .NET

In this chapter

Introduction

Complex systems, such as Microsoft Dynamics AX 2012, are often deployed in heterogeneous environments that contain several disparate systems. Often, these systems contain legacy data that might be required for running Microsoft Dynamics AX, or they might offer functionality that is vital for running the organization.

Microsoft Dynamics AX 2012 offers several ways of integrating with other systems. For example, your organization might need to harvest information from old Microsoft Excel files. To do this, you could write a simple add-on in Microsoft Visual Studio and easily integrate it with Microsoft Dynamics AX. Or your organization might have a legacy system that is physically located in a distant location that requires invoice information to be sent to it in a fail-safe manner. In this case, you could set up a message queue to perform the transfers. You could use the Microsoft .NET Framework to interact with the message queue from within Microsoft Dynamics AX.

This chapter focuses on some of the ways that you can integrate Microsoft Dynamics AX with other systems by taking advantage of managed code through X++ code. One way is to consume managed code directly from X++ code; another way is to author or extend existing business logic in managed code by using the Visual Studio environment. To facilitate this interoperability, Microsoft Dynamics AX provides the managed code with managed classes (called *proxies*) that represent X++ artifacts. This allows you to write managed code that uses the functionality these proxies provide in a type-safe and convenient manner.

In both cases, the .NET Framework provides access to the functionality, and this functionality is used in Microsoft Dynamics AX.

Note You can also make Microsoft Dynamics AX functionality available to other systems by using services. For more information, see Chapter 12, "Microsoft Dynamics AX services and integration."

Use third-party assemblies

Sometimes, you can implement the functionality that you are looking to provide by using a managed component (a .NET assembly) that you purchase from a third-party vendor. Using these dynamic-link libraries (DLLs) can be—and often is—more cost effective than writing the code yourself. These components are wrapped in managed assemblies in the form of .dll files, along with their Program Database (PDB) files, which contain symbol information that is used in debugging, and their XML files, which contain documentation that is used for IntelliSense in Visual Studio. Typically, these assemblies come with an installation program that often installs the assemblies in the global assembly cache (GAC) on the computer that consumes the functionality. This computer can be either on the client tier, on the server tier, or both. Only assemblies with strong names can be installed in the GAC.

Use strong-named assemblies

It is always a good idea to use a DLL that has a strong name, which means that the DLL is signed by the author, regardless of whether the assembly is stored in the GAC. This is true for assemblies that are installed on both the client tier and the server tier. A strong name defines the assembly's identity by its simple text name, version number, and culture information (if provided)—plus a public key and a digital signature. Assemblies with the same strong name are expected to be identical.

Strong names satisfy the following requirements:

- **Guarantee name uniqueness by relying on unique key pairs.** No one can generate the same assembly name that you can, because an assembly generated with one private key has a different name than an assembly generated with another private key.
- **Protect the version lineage of an assembly.** A strong name can ensure that no one can produce a subsequent version of your assembly. Users can be sure that the version of the assembly that they are loading comes from the same publisher that created the version the application was built with.
- **Provide a strong integrity check.** Passing the .NET Framework security checks guarantees that the contents of the assembly have not been changed since it was built. Note, however, that by themselves, strong names do not imply a level of trust such as that provided by a digital signature and supporting certificate.

When you reference a strong-named assembly, you can expect certain benefits, such as versioning and naming protection. If the strong-named assembly references an assembly with a simple name, which does not have these benefits, you lose the benefits that you derive by using a strong-named assembly and open the door to possible DLL conflicts. Therefore, strong-named assemblies can reference only other strong-named assemblies.

If the assembly that you are consuming does not have a strong name, and is therefore not installed in the GAC, you must manually copy the assembly (and the assemblies it depends on, if applicable) to a directory where the .NET Framework can find it when it needs to load the assembly for execution.

It is a good practice to place the assembly in the same directory as the executable that will ultimately load it (in other words, the folder on the client or the server in which the application is located). You might also want to store the assembly in the Client\Bin directory (even if it is used on the server exclusively), so that the client can pick it up and use it for IntelliSense.

Reference a managed DLL from Microsoft Dynamics AX

Microsoft Dynamics AX 2012 does not have a built-in mechanism for bulk deployment or installation of a particular DLL on client or server computers, because each third-party DLL has its own installation process. You must do this manually by using the installation script that the vendor provides or by placing the assemblies in the appropriate folders.

After you install the assembly on the client or server computer, you must add a reference to the assembly in Microsoft Dynamics AX, so that you can program against it in X++. You do this by adding the assembly to the *References* node in the Application Object Tree (AOT).

To do this, right-click the *References* node, and then click Add Reference. A dialog box like the one shown in Figure 3-1 appears.

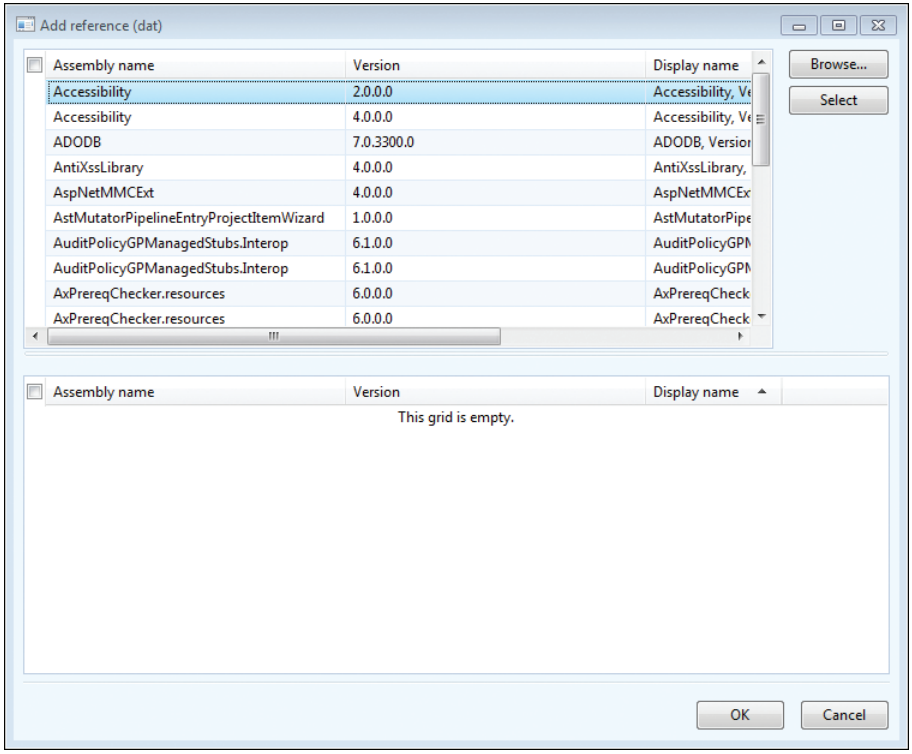

FIGURE 3-1 Adding a reference to a third-party assembly.

The top pane of the dialog box shows the assemblies that are installed in the GAC. If your assembly is installed in the GAC, click Select to add the reference to the *References* node. If the assembly is located in either the Client\Bin or the Server\Bin binary directory, click Browse. A file browser dialog box will appear where you can select your assembly. After you choose your assembly, it will appear in the bottom pane and will be added when you click OK.

Code against the assembly in X++

After you add the assembly, you are ready to use it from X++. If you install the code in the Client\Bin directory, IntelliSense features are available to help you edit the code. You can now use the managed code features of X++ to instantiate public managed classes, call methods on them, and so on. For more information, see Chapter 4, "The X++ programming language."

Note that there are some limitations to what you can achieve in X++ when calling managed code. One such limitation is that you cannot easily code against generic types (or execute generic methods). Another stems from the way the X++ interpreter works. Any managed object is represented as an instance of type *ClrObject*, and this has some surprising manifestations. For instance, consider the following code:

```
static void TestClr(Args _args)
{
    if (System.Int32::Parse("0"))
    {
        print "Do not expect to get here";
    }
    pause;
}
```

Obviously, you wouldn't expect the code in the *if* statement to execute because the result of the managed call is *0*, which is interpreted as *false*. However, the code actually prints the string literal because the return value of the call is a *ClrObject* instance that is not null (in other words, *true*). You can solve these problems by storing results in variables before use: the assignment operator will correctly unpack the value, as shown in the following example:

```
static void TestClr(Args _args)
{
    int i = System.Int32::Parse("0");
    if (i)
    {
        print "Do not expect to get here";
    }
    pause;
}
```

Write managed code

Sometimes your requirements cannot be satisfied by using an existing component and you have to roll up your sleeves and develop some code—in either C# or VB.NET. Microsoft Dynamics AX has great provisions for this: the integration features between Microsoft Dynamics AX and Visual Studio give you the luxury of dealing with X++ artifacts (classes, tables, and enumerations) as managed classes that behave the way that a developer of managed code would expect. The Microsoft Dynamics AX Business Connector manages the interaction between the two environments. Broadly speaking, you can create a project in Visual Studio as you normally would, and then add that project to the Visual Studio *Projects* node in the AOT. This section walks you through the process.

This example shows how to create managed code in C# (VB.NET could also be used) that reads the contents of an Excel spreadsheet and inserts the contents into a table in Microsoft Dynamics AX. This example is chosen to illustrate the concepts described in this chapter rather than for the functionality it provides.

The process is simple: you author the code in Visual Studio, and then add the solution to Application Explorer, which is just the name for the AOT in Visual Studio. Then, functionality from Microsoft Dynamics AX is made available for consumption by the C# code, which illustrates the proxy feature.

Assume that the Excel file contains the names of customers and the date that they registered as customers with your organization, as shown in Figure 3-2.

	A	B	C	D	E	F
1	Name	Date				
2	Abercrombie, Kim	04/28/74				
3	Abolrous, Hazem	06/22/49				
4	Abrus, Luka	03/19/47				
5	Abu-Dayah, Ahmad	08/11/65				
6	Acevedo, Humberto	04/17/85				
7	Achong, Gustavo	09/09/80				
8	Ackerman, Pilar	09/08/81				
9	Adalsteinsson, Gudmundur	04/17/86				
10	Adams, Terry	06/22/49				
11	Affronti, Michael					

Sheet1 Sheet2 Sheet3

FIGURE 3-2 Excel spreadsheet that contains a customer list.

Also assume that you've defined a table (called, say, CustomersFromExcel) in the AOT that will end up containing the information, subject to further processing. You could go about reading the information from the Excel files from X++ in several ways: one is by using the Excel automation model; another is by manipulating the Office Open XML document by using the XML classes. However, because it is so easy to read the contents of Excel files by using ADO.NET, that is what you decide to do. You start Visual Studio, create a C# class library called *ReadFromExcel*, and then write the following code:

```
using System;
using System.Collections.Generic;
using System.Text;
```

```
namespace Contoso
{
    using System.Data;
    using System.Data.OleDb;
    public class ExcelReader
    {
        static public void ReadDataFromExcel(string filename)
        {
            string connectionString;
            OleDbDataAdapter adapter;
            connectionString = @"Provider=Microsoft.ACE.OLEDB.12.0;"
            + "Data Source=" + filename + ";"
            + "Extended Properties='Excel 12.0 Xml;"

            + "HDR=YES'"; // Since sheet has row with column titles

            adapter = new OleDbDataAdapter(
                "SELECT * FROM [sheet1$]",
                connectionString);
            DataSet ds = new DataSet();
            // Get the data from the spreadsheet:
            adapter.Fill(ds, "Customers");
            DataTable table = ds.Tables["Customers"];
            foreach (DataRow row in table.Rows)
            {
                string name = row["Name"] as string;
                DateTime d = (DateTime)row["Date"];
            }
        }
    }
}
```

The *ReadDataFromExcel* method reads the data from the Excel file given as a parameter, but it does not currently do anything with that data. You still need to establish a connection to the Microsoft Dynamics AX system to store the values in the table. There are several ways of doing this, but in this case, you will simply use the Microsoft Dynamics AX table from the C# code by using the proxy feature.

The first step is to make the Visual Studio project (that contains the code) into a Microsoft Dynamics AX "citizen." You do this by selecting the Add ReadFromExcel To AOT menu item on the Visual Studio project. When this is done, the project is stored in the AOT and can use all of the functionality that is available for nodes in the AOT. The project can be stored in separate layers, be imported and exported, and so on. The project is stored in its entirety, and you can open Visual Studio to edit the project by clicking Edit on the context menu, as shown in Figure 3-3.

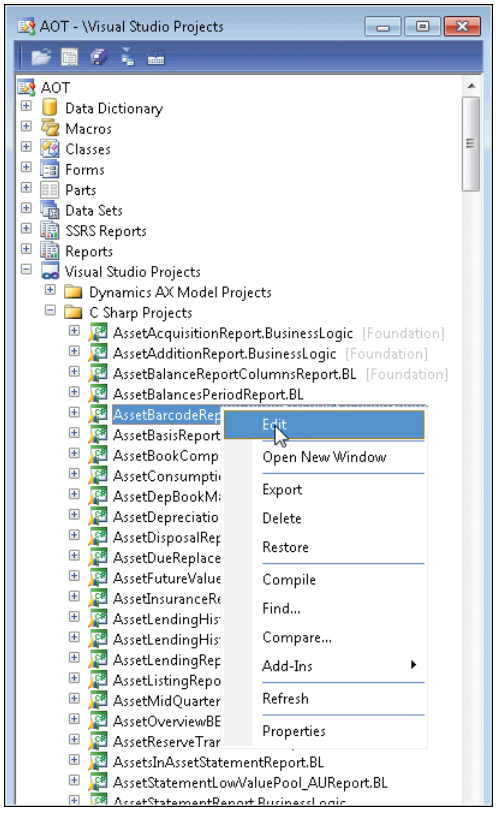

FIGURE 3-3 Context menu for Visual Studio projects that are stored in the AOT.

Tip You can tell that a project has been added to the AOT because the Visual Studio project icon is updated with a small Microsoft Dynamics AX icon in the lower-left corner.

With that step out of the way, you can use the version of the AOT that is available in Application Explorer in Visual Studio to fetch the table to use in the C# code (see Figure 3-4). If the Application Explorer window is not already open, you can open it by clicking Application Explorer on the View menu.

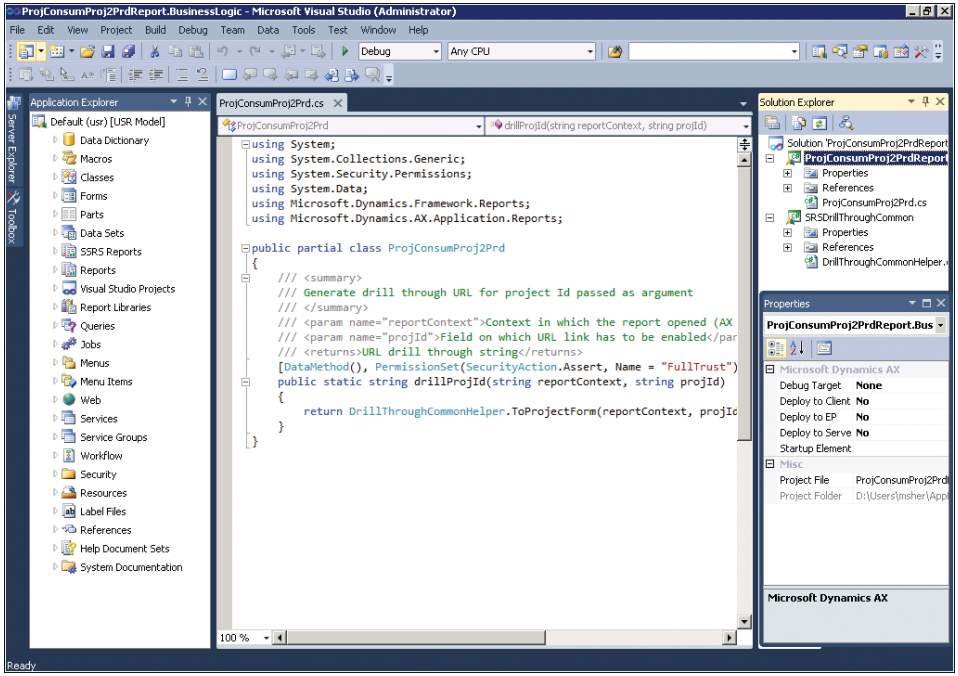

FIGURE 3-4 Application Explorer with a Microsoft Dynamics AX project open.

You can then create a C# representation of the table by dragging the table node from Application Explorer into the project.

After you drag the table node into the Visual Studio project, you will find an entry in the project that represents the table. The items that you drag into the project in this way are now available to code against in C#, just as though they had been written in C#. This happens because the drag operation creates a proxy for the table under the covers; this proxy takes care of the plumbing required to communicate with the Microsoft Dynamics AX system, while presenting a high-fidelity managed interface to the developer.

You can now proceed by putting the missing pieces into the C# code to write the data into the table. Modify the code as shown in the following example:

```
DataTable table = ds.Tables["Customers"];
var customers = new ReadFromExcel.CustomersFromExcel();
foreach (DataRow row in table.Rows)
{
    string name = row["Name"] as string;
    DateTime d = (DateTime)row["Date"];
    customers.Name = name;
    customers.Date = d;
    customers.Write();
}
```

Note The table from Microsoft Dynamics AX is represented just like any other type in C#. It supports IntelliSense, and the documentation comments that were added to methods in X++ are available to guide you as you edit.

The data will be inserted into the CustomersFromExcel table as it is read from the ADO.NET table that represents the contents of the spreadsheet. However, before either the client or the server can use this code, you must deploy it. You can do this by setting the properties in the Properties window for the Microsoft Dynamics AX project in Visual Studio. In this case, the code will run on the client, so you set the *Deploy to Client* property to *Yes*. There is a catch, though: you cannot deploy the assembly to the client when the client is running, so you must close any Microsoft Dynamics AX clients prior to deployment.

To deploy the code, right-click the Visual Studio project, and then click Deploy. If all goes well, a *Deploy Succeeded* message will appear in the status line.

Note You do not have to add a reference to the assembly because a reference is added implicitly to projects that you add to the AOT. You only need to add references to assemblies that are not the product of a project that has been added to the AOT.

As soon as you deploy the assembly, you can code against it in X++. The following example illustrates a simple snippet in an X++ job:

```
static void ReadCustomers(Args _args)
{
    ttsBegin;
    Contoso.ExcelReader::ReadDataFromExcel(@"c:\Test\customers.xlsx");
    ttsCommit;
}
```

When this job runs, it calls into the managed code and insert the records into the Microsoft Dynamics AX database.

Debug managed code

To ease the process of deploying after building, Visual Studio properties let you define what happens when you run the Microsoft Dynamics AX project. You manage this by using the *Debug Target* and *Startup Element* properties. You can enter the name of an element to execute—typically, a class with a suitable main method or a job. When you start the project in Visual Studio, it will create a new instance of the client and execute the class or job. The X++ code then calls back into the C# code where breakpoints are set. For more information, see "Debugging Managed Code in Microsoft Dynamics AX" at *http://msdn.microsoft.com/en-us/library/gg889265.aspx*.

An alternative to using this feature is to attach the Visual Studio debugger to the running Microsoft Dynamics AX client (by using the Attach To Process menu item on the Debug menu in Visual Studio). You can then set breakpoints and use all of the functionality of the debugger that you normally would. If you are running the Application Object Server (AOS) on your own computer, you can attach to that as well, but you must have administrator privileges to do so.

Important Do not debug in a production environment.

Proxies

As you can see, wiring up managed code to work with Microsoft Dynamics AX is quite simple because of the proxies that are generated behind the scenes to represent the Microsoft Dynamics AX tables, enumerations, and classes. In developer situations, it is standard to develop the artifacts in Microsoft Dynamics AX iteratively and then code against them in C#. This process is seamless because the proxies are regenerated by Visual Studio at build time, so that they are always synchronized with the corresponding artifacts in the AOT; in other words, the proxies never become out of date. In this way, proxies for Microsoft Dynamics AX artifacts differ from Visual Studio proxies for web services. These proxies are expected to have a stable application programming interface (API) so that the server hosting the web service is not contacted every time the project is built. Proxies are generated not only for the items that the user has chosen to drop onto the *Project* node as described previously. For instance, when a proxy is generated for a class, proxies will also be generated for all of its base classes, along with all artifacts that are part of the parameters for any methods, and so on.

To see what the proxies look like, place the cursor on a given proxy name in the code editor, such as *CustomersFromExcel* in the example, right-click, and then click Go To Definition (or use the convenient keyboard shortcut F12). All of the proxies are stored in the Obj/Debug folder for the project. If you look carefully, you will notice that the proxies use the Microsoft Dynamics AX Business Connector to do the work of interfacing with the Microsoft Dynamics AX system. The Business Connector has been completely rewritten from the previous version to support this scenario; in older versions of the product, the Business Connector invariably created a new session through which the interaction occurred. This is not the case for the new version of the Business Connector (at least when it is used as demonstrated here). That is why the transaction that was started in the job shown earlier is active when the records are inserted into the table. In fact, all aspects of the user's session are available to the managed code. This is the crucial difference between authoring business logic in managed code and consuming the business logic from managed code. When you author business logic, the managed code becomes an extension to the X++ code, which means that you can crisscross between Microsoft Dynamics AX and managed code in a consistent environment. When consuming business logic, you are better off using the services framework that Microsoft Dynamics AX provides, and then consuming the service from your application. This has big benefits in terms of scalability and deployment flexibility.

Figure 3-5 shows how the Business Connector relates to Microsoft Dynamics AX and .NET application code.

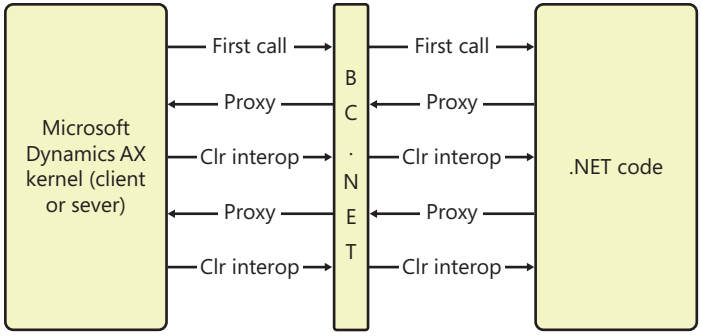

FIGURE 3-5 Interoperability between Microsoft Dynamics AX and .NET code through the Business Connector.

To demonstrate the new role of the Business Connector, the following example opens a form in the client that called the code:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace OpenFormInClient
{
    public class OpenFormClass
    {
        public void DoOpenForm(string formName)
        {
            Args a = new Args();
            a.name = formName;
            var fr = new FormRun(a);
            fr.run();
            fr.detach();
        }
    }
}
```

In the following example, a job is used to call managed code to open the CustTable form:

```
static void OpenFormFromDotNet(Args _args)
{
    OpenFormInClient.OpenFormClass opener;
    opener = new OpenFormInClient.OpenFormClass();
    opener.DoOpenForm("CustTable");
}
```

Note The *FormRun* class in this example is a kernel class. Because only an application class is represented in Application Explorer, you cannot add this proxy by dragging and dropping as described earlier. Instead, drop any class from Application Explorer onto the Visual Studio project, and then set the file name property of the class to *Class.*<kernelclassname>*.axproxy*. In this example, the name would be *Class.FormRun.axproxy*.

This would not have been possible with older versions of the Business Connector because they were basically faceless clients that could not display any user interface. Now, the Business Connector is actually part of the client (or server), and therefore, it can do anything they can. In Microsoft Dynamics AX 2012 R2, you can still use the Business Connector as a stand-alone client, but that is not recommended because that functionality is now better implemented by using services (see Chapter 12). The Business Connector that is included with Microsoft Dynamics AX is built with .NET Framework 3.5. That means that it is easier to build the business logic with this version of .NET; if you cannot do that for some reason, you must add markup to the App.config file to compensate. If you are using a program that is running .NET Framework 4.0 and you need to use the Business Connector through the proxies as described, you would typically add the following markup to the App.config file for your application:

```
<configuration>
  <startup useLegacyV2RuntimeActivationPolicy="true">
    "<supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.0"/>
  </startup>
</configuration>
```

Hot swap assemblies on the server

The previous section described how to express business logic in managed code. To simplify the scenario, code running on the client was used as an example. This section describes managed code running on the server.

You designate managed code to run on the server by setting the *Deploy to Server* property for the project to Yes, as shown in Figure 3-6.

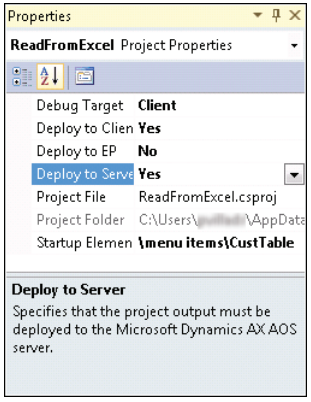

FIGURE 3-6 Property sheet showing the *Deploy to Server* property set to Yes.

When you set this property as shown in Figure 3-6, the assembly is deployed to the server directory. If the server has been running for a while, it will typically have loaded the assemblies into the current application domain. If Visual Studio were to deploy a new version of an existing assembly, the deployment would fail because the assembly would already be loaded into the current application domain.

To avoid this situation, the server has the option to start a new application domain in which it executes code from the new assembly. When a new client connects to the server, it will execute the updated code in a new application domain, while already connected clients continue to use the old version.

To use the hot-swapping feature, you must enable the option in the Microsoft Dynamics AX Server Configuration Utility by selecting the Allow Hot Swapping of Assemblies When The Server Is Running check box, as shown in Figure 3-7. To open the Microsoft Dynamics AX Server Configuration Utility, on the Start menu, point to Administrative Tools, and then click Microsoft Dynamics AX 2012 Server Configuration.

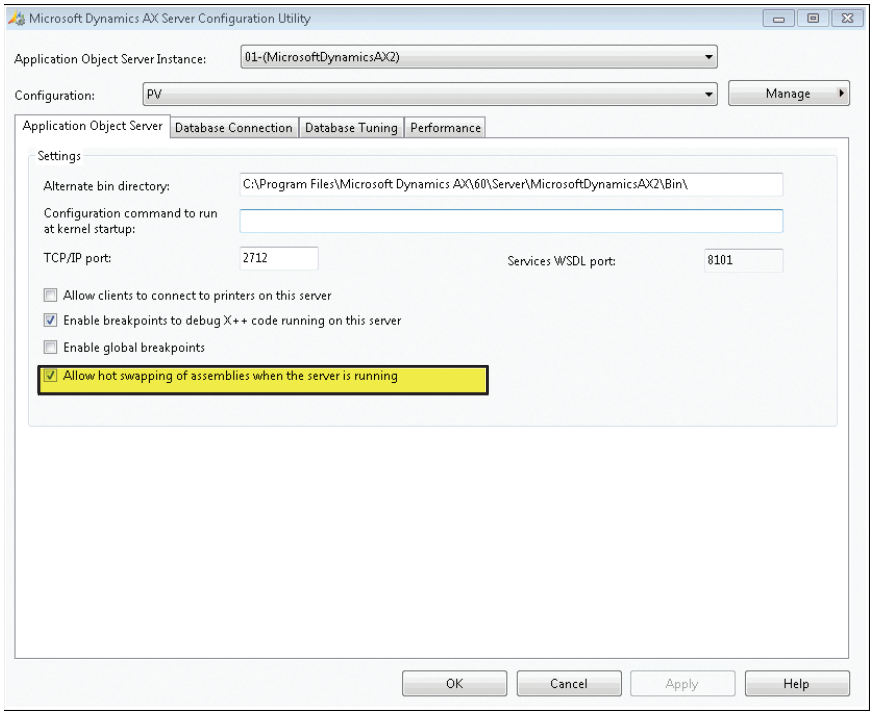

FIGURE 3-7 Allow hot swapping by using the Microsoft Dynamics AX Server Configuration Utility.

Note The example in the previous section illustrated how to run and debug managed code on the client, which is safe because the code runs only on a development computer. You can debug code that is running on the server (by starting Visual Studio as a privileged user and attaching to the server process as described in the "Debug managed code" section). However, you should never do this on a production server because any breakpoints that are encountered will stop all of the managed code from running, essentially blocking any users who are logged on to the server and processes that are running. Another reason you should not use hot swapping in a production scenario is that calling into another application domain extracts a performance overhead. The feature is intended only for development scenarios, where the performance of the application is irrelevant.

CHAPTER 4

The X++ programming language

In this chapter

Introduction

X++ is an object-oriented, application-aware, and data-aware programming language. The language is object oriented because it supports object abstractions, abstraction hierarchies, polymorphism, and encapsulation. It is application aware because it includes keywords such as *client*, *server*, *changecompany*, and *display* that are useful for writing client/server enterprise resource planning (ERP) applications. And it is data aware because it includes keywords such as *firstFast*, *forceSelectOrder*, and *forUpdate*, as well as a database query syntax, that are useful for programming database applications.

You use the Microsoft Dynamics AX designers and tools to edit the structure of application types. You specify the behavior of application types by writing X++ source code in the X++ editor. The X++ compiler compiles this source code into bytecode intermediate format. Model data, X++ source code, intermediate bytecode, and .NET common intermediate language (CIL) code are stored in the model store.

The Microsoft Dynamics AX runtime dynamically composes object types by loading overridden bytecode from the highest-level definition in the model layering stack. Objects are instantiated from these dynamic types. Similarly, the compiler produces .NET CIL from the X++ source code from the highest layer. For more information about the Microsoft Dynamics AX layering technology, see Chapter 21, "Application models."

This chapter describes the Microsoft Dynamics AX runtime type system and the features of the X++ language that are essential to writing ERP applications. It will also help you avoid common programming pitfalls that stem from implementing X++. For an in-depth discussion of the type system and the X++ language, refer to the Microsoft Dynamics AX 2012 software development kit (SDK), available on MSDN.

Jobs

Jobs are globally defined functions that execute in the Windows client run-time environment. Developers frequently use jobs to test a piece of business logic because they are easily executed from within the MorphX development environment, by either pressing F5 or selecting Go on the command menu. However, you shouldn't use jobs as part of your application's core design. The examples provided in this chapter can be run as jobs.

Jobs are model elements that you create by using the Application Object Tree (AOT). The following X++ code provides an example of a job model element that prints the string "Hello World" to an automatically generated window. The *pause* statement stops program execution and waits for user input from a dialog box.

```
static void myJob(Args _args)
{
    print "Hello World";
    pause;
}
```

The type system

The Microsoft Dynamics AX runtime manages the storage of value type data on the call stack and reference type objects on the memory heap. The *call stack* is the memory structure that holds data about the active methods called during program execution. The *memory heap* is the memory area that allocates storage for objects that are destroyed automatically by the Microsoft Dynamics AX runtime.

Value types

Value types include the built-in primitive types, extended data types, enumeration types, and built-in collection types:

- The primitive types are *boolean*, *int*, *int64*, *real*, *date*, *utcDateTime*, *timeofday*, *str*, and *guid*.
- The extended data types are specialized primitive types and specialized base enumerations.
- The enumeration types are base enumerations and extended data types.
- The collection types are the built-in array and container types.

By default, variables declared as value types are assigned their zero value by the Microsoft Dynamics AX runtime. These variables can't be set to null. Variable values are copied when variables are used to invoke methods and when they are used in assignment statements. Therefore, two value type variables can't reference the same value.

Reference types

Reference types include the record types, class types, and interface types:

- The record types are *table*, *map*, and *view*. User-defined record types are dynamically composed from application model layers. Microsoft Dynamics AX runtime record types are exposed in the system application programming interface (API).

Note Although the methods are not visible in the AOT, all record types implement the methods that are members of the system *xRecord* type, a Microsoft Dynamics AX runtime class type.

- User-defined class types are dynamically composed from application model layers and Microsoft Dynamics AX runtime class types exposed in the system API.
- Interface types are type specifications and can't be instantiated in the Microsoft Dynamics AX runtime. Class types can, however, implement interfaces.

Variables declared as reference types contain references to objects that the Microsoft Dynamics AX runtime instantiates from dynamically composed types defined in the application model layering system and from types exposed in the system API. The Microsoft Dynamics AX runtime also performs memory deallocation (garbage collection) for these objects when they are no longer referenced. Reference variables declared as record types reference objects that the Microsoft Dynamics AX runtime instantiates automatically. Class type objects are programmatically instantiated using the *new* operator. Copies of object references are passed as reference parameters in method calls and are assigned to reference variables, so two variables can reference the same object.

More Info Not all nodes in the AOT name a type declaration. Some class declarations are merely *syntactic sugar*—convenient, human-readable expressions. For example, the class header definition for all rich client forms declares a *FormRun* class type. *FormRun* is also, however, a class type in the system API. Allowing this declaration is syntactic sugar because it is technically impossible for two types to have the same name in the Microsoft Dynamics AX class type hierarchy.

Type hierarchies

The X++ language supports the definition of type hierarchies that specify generalized and specialized relationships between class types and table types. For example, a check payment method is a type of payment method. A type hierarchy allows code reuse. Reusable code is defined on base types defined higher in a type hierarchy because they are inherited, or reused, by derived types defined lower in a type hierarchy.

Tip You can use the Type Hierarchy Context and Type Hierarchy Browser tools in MorphX to visualize, browse, and search the hierarchy of any type.

The following sections introduce the base types provided by the Microsoft Dynamics AX runtime and describe how they are extended in type hierarchies.

Caution The Microsoft Dynamics AX type system is known as a weak type system because X++ accepts certain type assignments that are clearly erroneous and lead to runtime errors. Be aware of the caveats outlined in the following sections, and try to avoid weak type constructs when writing X++ code.

The *anytype* type

The Microsoft Dynamics AX type system doesn't have a single base type from which all types ultimately derive. However, the *anytype* type imitates a base type for all types. Variables of the *anytype* type function like value types when they are assigned a value type variable and like reference types when they are assigned a reference type variable. You can use the *SysAnyType* class to explicitly box all types, including value types, and make them function like reference types.

The *anytype* type, shown in the following code sample, is syntactic sugar that allows methods to accept any type as a parameter or allows a method to return different types:

```
static str queryRange(anytype _from, anytype _to)
{
    return SysQuery::range(_from,_to);
}
```

You can declare variables by using *anytype*. However, the underlying data type of an *anytype* variable is set to match the first assignment, and you can't change its type afterward, as shown here:

```
anytype a = 1;
print strfmt("%1 = %2", typeof(a), a); //Integer = 1
a = "text";
print strfmt("%1 = %2", typeof(a), a); //Integer = 0
```

The *common* type

The *common* type is the base type of all record types. Like the *anytype* type, record types are context-dependent types whose variables can be used as though they reference single records or as a record cursor that can iterate over a set of database records.

By using the *common* type, you can cast one record type to another (possibly incompatible) record type, as shown in this example:

```
//customer = vendor; //Compile error
common = customer;
vendor = common;     //Accepted
```

Tables in Microsoft Dynamics AX also support inheritance and polymorphism. This capability offers a type-safe method of sharing commonalities such as methods and fields between tables. It is possible to override table methods but not table fields. A base table can be marked as abstract or final through the table's properties.

Table maps defined in the AOT are a type-safe method of capturing commonalities between record types across type hierarchies, and you should use them to prevent incompatible record assignments. A table map defines fields and methods that safely operate on one or more record types.

The compiler doesn't validate method calls on the *common* type. For example, the compiler accepts the following method invocation, even though the method doesn't exist:

```
common.nonExistingMethod();
```

For this reason, you should use reflection to confirm that the method on the *common* type exists before you invoke it, as shown in this example. For more information, see Chapter 20, "Reflection."

```
if (tableHasMethod(new DictTable(common.tableId), identifierStr(existingMethod)))
{
    common.existingMethod();
}
```

The *object* type

The built-in *object* type is a weak reference type whose variables reference objects that are instances of class or interface types in the Microsoft Dynamics AX class hierarchy.

The type system allows you to implicitly cast base type objects to derived type objects and to cast derived type objects to base type objects, as shown here:

```
baseClass = derivedClass;
derivedClass = baseClass;
```

The *object* type allows you to use the assignment operator and cast one class type to another, incompatible class type, as shown in the following code. The probable result of this action, however, is a run-time exception when your code encounters an object of an unexpected type.

```
Object myObject;
//myBinaryIO = myTextIO; //Compile error
myObject = myTextIO;
mybinaryIO = myObject;   //Accepted
```

Use the *is* and *as* operators instead of the assignment operator to prevent these incompatible type casts. The *is* operator determines if an instance is of a particular type, and the *as* operator casts an instance as a particular type, or null if they are not compatible. The *is* and *as* operators work on class and table types.

```
myTextIO = myObject as TextIO;
if (myBinaryIO is TextIO)
{
}
```

You can use the *object* type for late binding to methods, similar to the dynamic keyword in C#. Keep in mind that a run-time error will occur if the method invoked doesn't exist.

```
myObject.lateBoundMethod();
```

Extended data types

You use the AOT to create extended data types that model concrete data values and data hierarchies. For example, the *Name* extended data type is a *string*, and the *CustName* and *VendName* extended data types extend the *Name* data type.

The X++ language supports extended data types but doesn't offer type checking according to the hierarchy of extended data types. X++ treats any extended data type as its primitive type; therefore, code such as the following is allowed:

```
CustName customerName;
FileName fileName = customerName;
```

When used properly, extended data types improve the readability of X++ code. It's easier to understand the intended use of a *CustName* data type than a *string* data type, even if both are used to declare *string* variables.

Extended data types are more than just type definitions that make X++ code more readable. On each extended data type, you can also specify how the system displays values of this type to users. Further, you can specify a reference to a table. The reference enables the form's rendering engine to automatically build lookup forms for form controls by using the extended data type, even when the form controls are not bound to a data source. On string-based extended data types, you can specify the maximum string size of the type. The database layer uses the string size to define the underlying columns for fields that use the extended data type. Defining the string size in only one place makes it easy to change.

Syntax

The X++ language belongs to the "curly brace" family of programming languages (those that use curly braces to delimit syntax blocks), such as C, C++, C#, and Java. If you're familiar with any of these languages, you won't have a problem reading and understanding X++ syntax.

Unlike many programming languages, X++ is not case sensitive. However, using camel casing *(camelCasing)* for variable names and Pascal casing *(PascalCasing)* for type names is considered a best practice. (More best practices for writing X++ code are available in the Microsoft Dynamics AX 2012 SDK.) You can use the Source Code Titlecase Update tool (accessed from the Add-Ins submenu in the AOT) to automatically apply casing in X++ code to match the best practice recommendation.

Common language runtime (CLR) types, which are case sensitive, are one important exception to the casing guidelines. For information about how to use CLR types, see the "CLR interoperability" section later in this chapter.

Variable declarations

You must place variable declarations at the beginning of methods. Table 4-1 provides examples of value type and reference type variable declarations, in addition to example variable initializations. Parameter declaration examples are provided in the "Classes and interfaces" section later in this chapter.

TABLE 4-1 X++ variable declaration examples.

Type	Examples
anytype	`anytype type = null;` `anytype type = 1;`
Base enumeration types	`NoYes theAnswer = NoYes::Yes;`
boolean	`boolean b = true;`

container	`container c1 = ["a string", 123];`
	`container c2 = [["a string", 123], c1];`
	`container c3 = connull();`

Type	Examples
date	`date d = 31\12\2008;`
Extended data types	`Name name = "name";`
guid	`guid g = newguid();`
int	`int i = -5;` `int h = 0xAB;`

int64	`int64 i = -5;`
	`int64 h = 0xAB;`
	`int64 u = 0xA0000000u;`

Object types	`Object obj = null;`
	`MyClass myClass = new MyClass();`
	`System.Text.StringBuilder sb = new System.Text.StringBuilder();`

real	`real r1 = 3.14;` `real r2 = 1.0e3;`
Record types	`Common myRecord = null;` `CustTable custTable = null;`

str	`str s1 = "a string";`
	`str s2 = 'a string';`
	`str 40 s40 = "string 40";`

TimeOfDay	`TimeOfDay time = 43200;`
utcDateTime	`utcDateTime dt = 2008-12-31T23:59:59;`

Note String literals can be expressed using either single or double quotes. It is considered best practice to use single quotes for system strings, like file names, and double quotes for user interface strings. The examples in this chapter adhere to this guideline.

Declaring variables with the same name as their type is a best practice. At first glance, this approach might seem confusing. Consider this class and its getter/setter method to its field:

```
Class Person
{
    Name name;

    public Name Name(Name _name = name)
    {
        name = _name;
        return name;
    }
}
```

Because X++ is not case sensitive, the word *name* is used in eight places in the preceding code. Three refer to the extended data type, four refer to the field, and one refers to the method (*_name* is used twice). To improve readability, you could rename the variable to something more specific,

such as *personName*. However, using a more specific variable name implies that a more specific type should be used (and created if it doesn't already exist). Changing both the type name and the variable name to *PersonName* wouldn't improve readability. The benefit of this practice is that if you know the name of a variable, you also know its type.

Note Previous versions of Microsoft Dynamics AX required a dangling semicolon to signify the end of a variable declaration. This is no longer required because the compiler solves the ambiguity by reading one token ahead, except where the first statement is a static CLR call. The compiler still accepts the now-superfluous semicolons, but you can remove them if you want to.

Expressions

X++ expressions are sequences of operators, operands, values, and variables that yield a result. Table 4-2 summarizes the types of expressions allowed in X++ and includes examples of their use.

TABLE 4-2 X++ expression examples.

Category	Examples

Access operators	`this //Instance member access`
	`element //Form member access`
	`<datasource>_ds //Form data source access`
	`<datasource>_q //Form query access`
	`x.y //Instance member access`
	`E::e //Enum access`
	`a[x] //Array access`
	`[v1, v2] = c //Container access`
	`Table.Field //Table field access`
	`Table.(FieldId) //Table field access`
	`(select statement).Field //Select result access`
	`System.Type //CLR namespace type access`
	`System.DayOfWeek::Monday //CLR enum access`

Arithmetic operators	`x = y + z // Addition`
	`x = y - z // Subtraction`
	`x = y * z // Multiplication`
	`x = y / z // Division`
	`x = y div z // Integer division`
	`x = y mod z // Integer division remainder`

Bitwise operators	`x = y & z // Bitwise AND`
	`x = y | z // Bitwise OR`
	`x = y ^ z // Bitwise exclusive OR (XOR)`
	`x = ~z // Bitwise complement`

Conditional operators	`x ? y : z`

Logical operators	`if (!obj) // Logical NOT`
	`if (a && b) // Logical AND`
	`if (a || b) // Logical OR`

Method invocations	`super() //Base member invocation`
	`MyClass::m() //Static member invocation`
	`myObject.m() //Instance member invocation`
	`this.m() //This instance member invocation`
	`myTable.MyMap::m(); //Map instance member invocation`
	`f() //Built-in function call`

Category	Examples

Object creation operators	`new MyClass()  //X++ object creation`
	`new System.DateTime()  //CLR object wrapper and`
	`//CLR object creation`
	`new System.Int32[100]() //CLR array creation`

Parentheses	`(x)`

Relational operators	`x < y  // Less than`
	`x > y  // Greater than`
	`x <= y  // Less than or equal`
	`x >= y  // Greater than or equal`
	`x == y  // Equal`
	`x != y  // Not equal`
	`select t where t.f like "a*" // Select using wildcards`

Shift operators	`x = y << z  // Shift left` `x = y >> z  // Shift right`
String concatenation	`"Hello" + "World"`
Values and variables	`"string"` `myVariable`

Statements

X++ statements specify object state and object behavior. Table 4-3 provides examples of X++ language statements that are commonly found in many programming languages. In-depth descriptions of each statement are beyond the scope of this book.

TABLE 4-3 X++ statement examples.

Statement	Example

.NET CLR interoperability statement	`System.Text.StringBuilder sb;`
	`sb = new System.Text.StringBuilder();`
	`sb.Append("Hello World");`
	`print sb.ToString();`
	`pause;`

Assignment statement	`int i = 42;`
	`i = 1;`
	`i++;`
	`++i;`
	`i--;`
	`--i;`
	`i += 1;`
	`i -= 1;`
	`this.myDelegate += eventhandler(obj.handler);`
	`this.myDelegate -= eventhandler(obj.handler);`

break statement	`int i;`
	`for ( i = 0; i < 100; i++ )`
	`{`
	`  if ( i > 50 )`
	`  {`
	`  break;`
	`  }`
	`}`

breakpoint statement	`breakpoint; //Causes the debugger to be invoked`

Statement	Example
Casting statement	`MyObject myObject = object as MyObject;` `boolean isCompatible = object is MyObject;`

changeCompany statement	`MyTable myTable;`
	`while select myTable`
	`{`
	`  print myTable.myField;`
	`}`
	`changeCompany("ZZZ")`
	`{`
	`  while select myTable`
	`  {`
	`  print myTable.myField;`
	`  }`
	`}`
	`pause;`

Compound statement	`int i;`
	`{`
	`  i = 3;`
	`  i++;`
	`}`

continue statement	`int i;`
	`int j = 0;`
	`for( i = 0; i < 100; i++ )`
	`{`
	`  if ( i < 50 )`
	`  {`
	`  continue;`
	`  }`
	`  j++;`
	`}`

do while statement	`int i = 4;`
	`do`
	`{`
	`  i++;`
	`}`
	`while ( i <= 100 );`

flush statement	`MyTable myTable;` `flush myTable;`

for statement	`int i;`
	`for ( i = 0; i < 42; i++ )`
	`{`
	`  print i;`
	`}`
	`pause;`

if statement	`boolean b = true;`
	`int i = 42;`
	`if ( b == true )`
	`{`
	`  i++;`
	`}`
	`else`
	`{`
	`  i--;`
	`}`

Statement	Example

Local function	`static void myJob(Args _args)`
	`{`
	`  str myLocalFunction()`
	`  {`
	`  return "Hello World";`
	`  }`
	`  print myLocalFunction();`
	`  pause;`
	`}`

pause statement	`print "Hello World";` `pause;`

print statement	`int i = 42;`
	`print i;`
	`print "Hello World";`
	`print "Hello World" at 10,5;`
	`print 5.2;`
	`pause;`

retry statement	`try`
	`{`
	`  throw error("Force exception");`
	`}`
	`catch( exception::Error )`
	`{`
	`  retry;`
	`}`

return statement	`int foo()`
	`{`
	`  return 42;`
	`}`

switch statement	`str s = "test";`
	`switch ( s )`
	`{`
	`  case "test" :`
	`  print s;`
	`  break;`
	`  default :`
	`  print "fail";`
	`}`
	`pause;`

System function	`guid g = newGuid();` `print abs(-1);`
throw statement	`throw error("Error text");`

try statement	`try`
	`{`
	`  throw error("Force exception");`
	`}`
	`catch( exception::Error )`
	`{`
	`  print "Error";`
	`  pause;`
	`}`
	`catch`
	`{`
	`  print "Another exception";`
	`  pause;`
	`}`

unchecked statement	`unchecked(Uncheck::TableSecurityPermission)`
	`{`
	`  this.method();`
	`}`

Statement	Example

while statement	`int i = 4;`
	`while ( i <= 100 )`
	`{`
	`  i++;`
	`}`

window statement	`window 100, 10 at 100, 10;`
	`print "Hello World";`
	`pause;`

Data-aware statements

The X++ language has built-in support for querying and manipulating database data. The syntax for database statements is similar to Structured Query Language (SQL), and this section assumes that you're familiar with SQL. The following code shows how a *select* statement is used to return only the first selected record from the MyTable database table and how the data in the record's *myField* field is printed:

```
static void myJob(Args _args)
{
     MyTable myTable;
     select firstOnly * from myTable where myTable.myField1 == "value";
     print myTable.myField2;
     pause;
}
```

The "* from" part of the *select* statement in the example is optional. You can replace the asterisk (*) character with a comma-separated field list, such as *myField2, myField3*. You must define all fields, however, on the selection table model element, and only one selection table is allowed mmediately after the *from* keyword. The *where* expression in the *select* statement can include any number of logical and relational operators. The *firstOnly* keyword is optional and can be replaced by one or more of the optional keywords. Table 4-4 describes all possible keywords. For more information about database-related keywords, see Chapter 17, "The database layer."

TABLE 4-4 Keyword options for *select* statements.

Keyword	Description
crossCompany	Forces the Microsoft Dynamics AX runtime to generate a query without automatically adding the *where* clause in the *dataAreaId* field. This keyword can be used to select records from all or from a set of specified company accounts. For example, the query `while select crosscompany:companies myTable { }` selects all records in the *myTable* table from the company accounts specified in the *companies* container.
firstFast	Fetches the first selected record faster than the remaining selected records.

Keyword	Description
firstOnly *firstOnly1*	Returns only the first selected record.
firstOnly10	Returns only the first 10 selected records.
firstOnly100	Returns only the first 100 selected records.
firstOnly1000	Returns only the first 1,000 selected records.
forceLiterals	Forces the Microsoft Dynamics AX runtime to generate a query with the specified field constraints. For example, the query generated for the preceding code example looks like this: *select * from myTable where myField1='value'*. Database query plans aren't reused when this option is specified. This keyword can't be used with the *forcePlaceholders* keyword.
forceNestedLoop	Forces the SQL Server query processor to use a nested-loop algorithm for table join operations. Other join algorithms, such as hash-join and merge-join, are therefore not considered by the query processor.
forcePlaceholders	Forces the Microsoft Dynamics AX runtime to generate a query with placeholder field constraints. For example, the query generated for the preceding code example looks like this: *select * from myTable where myField1=?*. Database query plans are reused when this option is specified. This is the default option for *select* statements that don't join table records. This keyword can't be used with the *forceLiterals* keyword.
forceSelectOrder	Forces the Microsoft SQL Server query processor to access tables in the order in which they are specified in the query.
forUpdate	Selects records for updating.
generateOnly	Instructs the SQL Server query processor to only generate the SQL statements—and not execute them. The generated SQL statement can be retrieved using the *getSQLStatement* method on the primary table.
noFetch	Specifies that the Microsoft Dynamics AX runtime should not execute the statement immediately because the records are required only by some other operation.
optimisticLock	Overrides the table's *OccEnabled* property and forces the optimistic locking scheme. This keyword can't be used with the *pessimisticLock* and *repeatableRead* keywords.
pessimisticLock	Overrides the table's *OccEnabled* property and forces the pessimistic locking scheme. This keyword can't be used with the *optimisticLock* and *repeatableRead* keywords.
repeatableRead	Locks all records read within a transaction. This keyword can be used to ensure consistent data is fetched by identical queries for the duration of the transaction, at the cost of blocking other updates of those records. Phantom reads can still occur if another process inserts records that match the range of the query. This keyword can't be used with the *optimisticLock* and *pessimisticLock* keywords.
reverse	Returns records in the reverse of the *select* order.
validTimeState	Instructs the SQL Server query processor to use the provided date or date range instead of the current date. For example, the query `while select validTimeState(fromDate, toDate) myTable { }` selects all records in the *myTable* table that are valid in the period from *fromDate to toDate*.

The following code example demonstrates how to use a table index clause to suggest the index that a database server should use when querying tables. The Microsoft Dynamics AX runtime appends an *order by* clause and the *index* fields to the first *select* statement's database query. Records are thus ordered by the index. The Microsoft Dynamics AX runtime can insert a query hint into the second *select* statement's database query, if the hint is feasible to use.

```
static void myJob(Args _args)
{
    MyTable1 myTable1;
    MyTable2 myTable2;

    while select myTable1
        index myIndex1
    {
        print myTable1.myField2;
    }

    while select myTable2
        index hint myIndex2
    {
        print myTable2.myField2;
    }
    pause;
}
```

The following code example demonstrates how the results from a *select* query can be ordered and grouped. The first *select* statement specifies that the resulting records must be sorted in ascending order based on *myField1* values and then in descending order based on *myField2* values. The second *select* statement specifies that the resulting records must be grouped by *myField1* values and then sorted in descending order.

```
static void myJob(Args _args)
{
    MyTable myTable;

    while select myTable
        order by Field1 asc, Field2 desc
    {
        print myTable.myField;
    }
    while select myTable
        group by Field1 desc
    {
        print myTable.Field1;
    }
    pause;
}
```

The following code demonstrates use of the *avg* and *count* aggregate functions in *select* statements. The first *select* statement averages the values in the *myField* column and assigns the result to the *myField* field. The second *select* statement counts the number of records the selection returns and assigns the result to the *myField* field.

```
static void myJob(Args _args)
{
     MyTable myTable;

     select avg(myField) from myTable;
     print myTable.myField;

     select count(myField) from myTable;
     print myTable.myField;
     pause;
}
```

Caution The compiler doesn't verify that aggregate function parameter types are numeric, so the result that the function returns could be assigned to a field of type *string*. The result will be truncated if, for example, the *average* function calculates a value of 1.5 and the type of *myField* is an integer.

Table 4-5 Describes the aggregate functions supported in X++ *select* statements.

TABLE 4-5 Aggregate functions in X++ *select* statements.

Function	Description
avg	Returns the average of the non-null field values in the records the selection returns.
count	Returns the number of non-null field values in the records the selection returns.
maxOf	Returns the maximum of the non-null field values in the records the selection returns.
minOf	Returns the minimum of the non-null field values in the records the selection returns.
sum	Returns the sum of the non-null field values in the records the selection returns.

The following code example demonstrates how tables are joined with *join* conditions. The first *select* statement joins two tables by using an equality *join* condition between fields in the tables. The second *select* statement joins three tables to illustrate how you can nest *join* conditions and use an *exists* operator as an existence test with a *join* condition. The second *select* statement also demonstrates how you can use a *group by* sort in *join* conditions. In fact, the *join* condition can comprise multiple nested *join* conditions because the syntax of the *join* condition is the same as the body of a *select* statement.

```
static void myJob(Args _args)
{
     MyTable1 myTable1;
     MyTable2 myTable2;


     MyTable3 myTable3;

     select myField from myTable1
        join myTable2
           where myTable1.myField1=myTable2.myField1;
     print myTable1.myField;

     select myField from myTable1
        join myTable2
          group by myTable2.myField1
          where myTable1.myField1=myTable2.myField1
             exists join myTable3
                where myTable1.myField1=myTable3.mField2;
     print myTable1.myField;
     pause;
}
```

Table 4-6 describes the *exists* operator and the other *join* operators that can be used in place of the *exists* operator in the preceding example.

TABLE 4-6 Join operators.

Operator	Description
exists	Returns *true* if any records are in the result set after executing the *join* clause. Returns *false* otherwise.
notExists	Returns *false* if any records are in the result set after executing the *join* clause. Returns *true* otherwise.
outer	Returns the left outer *join* of the first and second tables.

The following example demonstrates use of the *while select* statement that increments the *myTable* variable's record cursor on each loop:

```
static void myJob(Args _args)
{
     MyTable myTable;

     while select myTable
     {
         Print myTable.myField;
     }
}
```

You must use the *ttsBegin*, *ttsCommit*, and *ttsAbort* transaction statements to modify records in tables and to insert records into tables. The *ttsBegin* statement marks the beginning of a database transaction block; *ttsBegin*-*ttsCommit* transaction blocks can be nested. The *ttsBegin* statements increment the transaction level; the *ttsCommit* statements decrement the transaction level. The outermost block decrements the transaction level to zero and commits all database inserts and updates performed since the first *ttsBegin* statement to the database. The *ttsAbort* statement rolls back all database inserts, updates, and deletions performed since the *ttsBegin* statement. Table 4-7 provides examples of these transaction statements for single records and operations and for set-based (multiple-record) operations.

The last example in Table 4-7 demonstrates the method *RowCount*. Its purpose is to get the count of records that are affected by set-based operations; namely, *insert_recordset*, *update_recordset*, and *delete_from*.

By using *RowCount,* it is possible to save one round-trip to the database in certain application scenarios; for example, when implementing insert-or-update logic.

TABLE 4-7 Transaction statement examples.

Statement Type	Example

delete_from	`MyTable myTable;`
	`Int64 numberOfRecordsAffected;`
	
	`ttsBegin;`
	`delete_from myTable`
	`  where myTable.id == "001";`
	`numberOfRecordsAffected = myTable.RowCount();`
	`ttsCommit;`

insert method	`MyTable myTable;`
	
	`ttsBegin;`
	`myTable.id = "new id";`
	`myTable.myField = "new value";`
	`myTable.insert();`
	`ttsCommit;`

insert_recordset	`MyTable1 myTable1;`
	`MyTable2 myTable2;`
	`int64 numberOfRecordsAffected;`
	
	`ttsBegin;`
	`insert_recordset myTable2 ( myField1, myField2 )`
	`  select myField1, myField2 from myTable1;`
	`numberOfRecordsAffected = myTable.RowCount();`
	`ttsCommit;`

select forUpdate	`MyTable myTable;`
	
	`ttsBegin;`
	`select forUpdate myTable;`
	`myTable.myField = "new value";`
	`myTable.update();`
	`ttsCommit;`

ttsBegin
ttsCommit
ttsAbort
	`boolean b = true;`
	
	`ttsBegin;`
	`if ( b == true )`
	`  ttsCommit;`
	`else`
	`  ttsAbort;`

Statement Type	Example

update_recordset	`MyTable myTable;`
	`int64 numberOfRecordsAffected;`
	
	`ttsBegin;`
	`update_recordset myTable setting`
	`  myField1 = "value1",`
	`  myField2 = "value2"`
	`  where myTable.id == "001";`
	`numberOfRecordsAffected = myTable.RowCount();`
	`ttsCommit;`

Exception handling

It is a best practice to use the X++ exception handling framework instead of programmatically halting a transaction by using the *ttsAbort* statement. An exception (other than the update conflict and duplicate key exceptions) thrown inside a transaction block halts execution of the block, and all of the inserts and updates performed since the first *ttsBegin* statement are rolled back. Throwing an exception has the additional advantage of providing a way to recover object state and maintain the consistency of database transactions. Inside the *catch* block, you can use the *retry* statement to run the *try* block again. The following example demonstrates throwing an exception inside a database transaction block:

```
static void myJob(Args _args)
{
    MyTable myTable;
    boolean state = false;

    try
    {
        ttsBegin;

        update_recordset myTable setting
            myField = "value"
            where myTable.id == "001";
        if(state==false)
        {
            throw error("Error text");
        }
        ttsCommit;
    }
    catch(Exception::Error)
    {
        state = true;
        retry;
    }
}
```

The *throw* statement throws an exception that causes the database transaction to halt and roll back. Code execution can't continue inside the scope of the transaction, so the runtime ignores

try and *catch* statements when inside a transaction. This means that an exception thrown inside a transaction can be caught only outside the transaction, as shown here:

```
static void myJob(Args _args)
{
    try
    {
        ttsBegin;
        try
        {
            ...
            throw error("Error text");
        }
        catch  //Will never catch anything
        {
        }
        ttsCommit;
    }
    catch(Exception::Error)
    {
        print "Got it";
        pause;
    }
    catch
    {
        print "Unhandled Exception";
        pause;
    }
}
```

Although a *throw* statement takes the exception enumeration as a parameter, using the *error* method to throw errors is considered best practice. The *try* statement's catch list can contain more than one *catch* block. The first *catch* block in the example catches error exceptions. The *retry* statement jumps to the first statement in the outer *try* block. The second *catch* block catches all exceptions not caught by *catch* blocks earlier in the *try* statement's catch list. Table 4-8 describes the Microsoft Dynamics AX system *Exception* data type enumerations that can be used in *try-catch* statements.

TABLE 4-8 *Exception* data type enumerations.

Element	Description
Break	Thrown when a user presses the Break key or Ctrl+C.
CLRError	Thrown when an unrecoverable error occurs in a CLR process.
CodeAccessSecurity	Thrown when an unrecoverable error occurs in the *demand* method of a *CodeAccessPermission* object.
DDEerror	Thrown when an error occurs in the use of a Dynamic Data Exchange (DDE) system class.
Deadlock	Thrown when a database transaction has deadlocked.

Element	Description
DuplicateKeyException	Thrown when a duplicate key error occurs during an insert operation. The *catch* block should change the value of the primary keys and use a *retry* statement to attempt to commit the halted transaction.
DuplicateKeyExceptionNotRecovered	Thrown when an unrecoverable duplicate key error occurs during an insert operation. The *catch* block shouldn't use a *retry* statement to attempt to commit the halted transaction.
*Error**	Thrown when an unrecoverable application error occurs. A *catch* block should assume that all database transactions in a transaction block have been halted and rolled back.
Internal	Thrown when an unrecoverable internal error occurs.
Numeric	Thrown when a mathematical error occurs like division by zero, logarithm of a negative number, or conversion between incompatible types.
PassClrObjectAcrossTiers	Thrown when an attempt is made to pass a CLR object from the client to the server tier or vice versa. The Microsoft Dynamics AX runtime doesn't support automatic marshaling of CLR objects across tiers.
Sequence	Thrown by the Microsoft Dynamics AX kernel if a database error or database operation error occurs.
Timeout	Thrown when a database operation times out.
UpdateConflict	Thrown when an update conflict error occurs in a transaction block using optimistic concurrency control. The *catch* block should use a *retry* statement to attempt to commit the halted transaction.
UpdateConflictNotRecovered	Thrown when an unrecoverable error occurs in a transaction block using optimistic concurrency control. The *catch* block shouldn't use a *retry* statement to attempt to commit the halted transaction.

* The *error* method is a static method of the global X++ class for which the X++ compiler allows an abbreviated syntax. The expression *Global::error("Error text")* is equivalent to the error expression in the code examples earlier in this section. Don't confuse these global X++ methods with Microsoft Dynamics AX system API methods, such as *newGuid*.

UpdateConflict and *DuplicateKeyException* are the only data exceptions that a Microsoft Dynamics AX application can handle inside a transaction. Specifically, with *DuplicateKeyException*, the database transaction isn't rolled back, and the application is given a chance to recover. *DuplicateKeyException* facilitates application scenarios (such as Master Planning) that perform batch processing and handles duplicate key exceptions without aborting the transaction in the midst of the resource-intensive processing operation.

The following example illustrates the usage of *DuplicateKeyException*:

```
static void DuplicateKeyExceptionExample(Args _args)
{
    MyTable myTable;

    ttsBegin;
    myTable.Name = "Microsoft Dynamics AX";
    myTable.insert();
    ttsCommit;

    ttsBegin;
    try
    {
```

```
            myTable.Name = "Microsoft Dynamics AX";
            myTable.insert();
        }
        catch(Exception::DuplicateKeyException)
        {
            info(strfmt("Transaction level: %1", appl.ttsLevel()));
            info(strfmt("%1 already exists.", myTable.Name));
            info(strfmt("Continuing insertion of other records"));
        }
        ttsCommit;
    }
```

In the preceding example, the *catch* block handles the duplicate key exception. Notice that the transaction level is still 1, indicating that the transaction hasn't aborted and the application can continue processing other records.

Note The special syntax where a table instance was included in the *catch* block is no longer available.

Interoperability

The X++ language includes statements that allow interoperability (interop) with .NET CLR assemblies and COM components. The Microsoft Dynamics AX runtime achieves this interoperability by providing Dynamics AX object wrappers around external objects and by dispatching method calls from the Microsoft Dynamics AX object to the wrapped object.

CLR interoperability

You can write X++ statements for CLR interoperability by using one of two methods: strong typing or weak typing. Strong typing is recommended because it is type safe and less error prone than weak typing, and it results in code that is easier to read. The MorphX X++ editor also provides IntelliSense as you type.

The examples in this section use the .NET *System.Xml* assembly, which is added as an AOT references node. (See Chapter 1, "Architectural overview," for a description of programming model elements.) The programs are somewhat verbose because the compiler doesn't support method invocations on CLR return types and because CLR types must be identified by their fully qualified name. For example, the expression *System.Xml.XmlDocument* is the fully qualified type name for the .NET Framework XML document type.

Caution X++ is case sensitive when referring to CLR types.

The following example demonstrates strongly typed CLR interoperability with implicit type conversions from Microsoft Dynamics AX strings to CLR strings in the string assignment statements and shows how CLR exceptions are caught in X++:

```
static void myJob(Args _args)
{
   System.Xml.XmlDocument doc = new System.Xml.XmlDocument();
   System.Xml.XmlElement rootElement;
   System.Xml.XmlElement headElement;
   System.Xml.XmlElement docElement;
   System.String xml;
   System.String docStr = 'Document';
   System.String headStr = 'Head';
   System.Exception ex;
   str errorMessage;

   try
   {
       rootElement = doc.CreateElement(docStr);
       doc.AppendChild(rootElement);
       headElement = doc.CreateElement(headStr);
       docElement = doc.get_DocumentElement();
       docElement.AppendChild(headElement);
       xml = doc.get_OuterXml();
       print ClrInterop::getAnyTypeForObject(xml);
       pause;
   }
   catch(Exception::CLRError)
   {
       ex = ClrInterop::getLastException();
       if( ex )
       {
           errorMessage = ex.get_Message();
           info( errorMessage );
       }
   }
}
```

The following example illustrates how static CLR methods are invoked by using the X++ static method *accessor ::*.

```
static void myJob(Args _args)
{
    System.Guid g = System.Guid::NewGuid();
}
```

The following example illustrates the support for CLR arrays:

```
static void myJob(Args _args)
{
    System.Int32 [] myArray = new System.Int32[100]();

    myArray.SetValue(1000, 0);
    print myArray.GetValue(0);
}
```

X++ supports passing parameters by reference to CLR methods. Changes that the called method makes to the parameter also change the caller variable's value. When non-object type variables are passed by reference, they are wrapped temporarily in an object. This operation is often called *boxing* and is illustrated in the following example:

```
static void myJob(Args _args)
{
    int myVar = 5;

    MyNamespace.MyMath::Increment(byref myVar);

    print myVar;   // prints 6
}
```

The called method could be implemented in C# like this:

```
// Notice: This example is C# code
static public void Increment(ref int value)
{
    value++;
}
```

Note Passing parameters by reference is supported only for CLR methods, not for X++ methods.

The second method of writing X++ statements for CLR uses weak typing. The following example shows CLR types that perform the same steps as in the first CLR interoperability example. In this case, however, all references are validated at run time, and all type conversions are explicit.

```
static void myJob(Args _args)
{
   ClrObject doc = new ClrObject('System.Xml.XmlDocument');
   ClrObject docStr;
```

```
    ClrObject rootElement;
    ClrObject headElement;
    ClrObject docElement;
    ClrObject xml;

    docStr = ClrInterop::getObjectForAnyType('Document');
    rootElement = doc.CreateElement(docStr);
    doc.AppendChild(rootElement);
    headElement = doc.CreateElement('Head');
    docElement = doc.get_DocumentElement();
    docElement.AppendChild(headElement);
    xml = doc.get_OuterXml();
    print ClrInterop::getAnyTypeForObject(xml);
    pause;
}
```

The first statement in the preceding example demonstrates the use of a static method to convert X++ primitive types to CLR objects. The *print* statement shows the reverse, converting CLR value types to X++ primitive types. Table 4-9 lists the value type conversions that Microsoft Dynamics AX supports.

TABLE 4-9 Type conversions supported in Microsoft Dynamics AX.

CLR Type	Microsoft Dynamics AX Type
Byte, SByte, Int16, UInt16, Int32	*int*
Byte, SByte, Int16, UInt16, Int32, Uint32, Int64	*int64*
DateTime	*utcDateTime*
Double, Single	*real*
Guid	*guid*
String	*str*
Microsoft Dynamics AX Type	**CLR Type**
int	*Int32, Int64*
int64	*Int64*
utcDateTime	*DateTime*
real	*Single, Double*
guid	*Guid*
str	*String*

The preceding code example also demonstrates the X++ method syntax used to access CLR object properties, such as *get_DocumentElement*. The CLR supports several operators that are not supported in X++. Table 4-10 lists the supported CLR operators and the alternative method syntax.

TABLE 4-10 CLR operators and methods.

CLR Operators	CLR Methods
Property operators	*get_<property>, set_<property>*
Index operators	*get_Item, set_Item*
Math operators	*op_<operation>(arguments)*

The following features of CLR can't be used with X++:

- Public fields (can be accessed by using CLR reflection classes)
- Events and delegates
- Generics
- Inner types
- Namespace declarations

COM interoperability

The following code example demonstrates COM interoperability with the XML document type in the Microsoft XML Core Services (MSXML) 6.0 COM component. The example assumes that you've installed MSXML. The MSXML document is first instantiated and wrapped in a Microsoft Dynamics AX COM object wrapper. A COM variant wrapper is created for a COM string. The direction of the variant is put into the COM component. The root element and head element variables are declared as COM objects. The example shows how to fill a string variant with an X++ string and then use the variant as an argument to a COM method, *loadXml*. The statement that creates the head element demonstrates how the Microsoft Dynamics AX runtime automatically converts Microsoft Dynamics AX primitive objects into COM variants.

```
static void Job2(Args _args)
{
    COM doc = new COM('Msxml2.DomDocument.6.0');
    COMVariant rootXml =
               new COMVariant(COMVariantInOut::In,COMVariantType::VT_BSTR);
    COM rootElement;
    COM headElement;

    rootXml.bStr('<Root></Root>');
    doc.loadXml(rootXml);
    rootElement = doc.documentElement();
    headElement = doc.createElement('Head');
    rootElement.appendChild(headElement);
    print doc.xml();
    pause;
}
```

Macros

With the macro capabilities in X++, you can define and use constants and perform conditional compilation. Macros are unstructured because they are not defined in the X++ syntax. Macros are handled before the source code is compiled. You can add macros anywhere you write source code: in methods and in class declarations.

Table 4-11 shows the supported macro directives.

TABLE 4-11 Macro directives.

Directive	Description
#define *#globaldefine*	Defines a macro with a value. `#define.MyMacro(SomeValue)` Defines the macro *MyMacro* with the value *SomeValue*.

#macro
...
#endmacro
 	
#localmacro
...
#endmacro
	Defines a macro with a value spanning multiple lines.
	`#macro.MyMacro`
	`  print "foo";`
	`  print "bar";`
	`#endmacro`
	Defines the macro *MyMacro* with a multiple-line value.

#macrolib	Includes a macro library. As a shorthand form of this directive, you can omit *macrolib*.
	`#macrolib.MyMacroLibrary`
	`#MyMacroLibrary`
	Both include the macro library *MyMacroLibrary*, which is defined under the Macros node in the AOT.

#MyMacro	Replaces a macro with its value.
	`#define.MyMacro("Hello World")`
	`print #MyMacro;`
	Defines the macro *MyMacro* and prints its value. In this example, "Hello World" would be printed.

#definc *#defdec*	Increments and decrements the value of a macro; typically used when the value is an integer. `#defdec.MyIntMacro` Decrements the value of the macro *MyIntMacro*.
#undef	Removes the definition of a macro. `#undef.MyMacro` Removes the definition of the macro *MyMacro*.

#if
...
#endif
	Conditional compile. If the macro referenced by the *#if* directive is defined or has a specific value, the following text is included in the compilation:
	`#if.MyMacro`
	`print "MyMacro is defined";`
	`#endif`

Directive	Description

	If *MyMacro* is defined, the *print* statement is included as part of the source code:
	`#if.MyMacro(SomeValue)`
	`print "MyMacro is defined and has value: SomeValue";`
	`#endif`
	
	If *MyMacro* has *SomeValue*, the *print* statement is included as part of the source code.

#ifnot
...
#endif
	Conditional compile. If the macro referenced by the *#ifnot* directive isn't defined or doesn't have a specific value, the following text is included in the compilation:
	`#ifnot.MyMacro`
	`print "MyMacro is not defined";`
	`#endif`
	
	If *MyMacro* is not defined, the *print* statement is included as part of the source code:
	`#ifnot.MyMacro(SomeValue)`
	`print "MyMacro does not have value: SomeValue; or it is not defined";`
	`#endif`
	
	If *MyMacro* is not defined, or if it does not have *SomeValue*, the *print* statement is included as part of the source code.

The following example shows a macro definition and reference:

```
void myMethod()
{
    #define.HelloWorld("Hello World")

    print #HelloWorld;
    pause;
}
```

As noted in Table 4-11, a macro library is created under the Macros node in the AOT. The library is included in a class declaration header or class method, as shown in the following example:

```
class myClass
{
    #MyMacroLibrary1
}
public void myMethod()
{
    #MyMacroLibrary2


    #MacroFromMyMacroLibrary1
    #MacroFromMyMacroLibrary2
}
```

A macro can also use parameters. The compiler inserts the parameters at the positions of the placeholders. The following example shows a local macro using parameters:

```
void myMethod()
{
    #localmacro.add
        %1 + %2
    #endmacro

    print #add(1, 2);
    print #add("Hello", "World");
    pause;
}
```

When a macro library is included or a macro is defined in the class declaration of a class, the macro can be used in the class and in all classes derived from the class. A subclass can redefine the macro.

Comments

X++ allows single-line and multiple-line comments. Single-line comments start with *//* and end at the end of the line. Multiple-line comments start with */** and end with ***/*. You can't nest multiple-line comments.

You can add reminders to yourself in comments that the compiler picks up and presents to you as tasks in its output window. To set up these tasks, start a comment with the word **TODO**. Be aware that tasks not occurring at the start of the comment, (for example, tasks that are deep inside multiple-line comments,) are ignored by the compiler.

The following code example contains comments reminding the developer to add a new procedure while removing an existing procedure by changing it into a comment:

```
public void myMethod()
{
    //Declare variables
    int value;

//TODO Validate if calculation is really required
/*
    //Perform calculation
    value = this.calc();
*/
    ...
}
```

XML documentation

You can document XML methods and classes directly in X++ by typing three slash characters (///) followed by structured documentation in XML format. The XML documentation must be above the actual code.

The XML documentation must align with the code. The Best Practices tool contains a set of rules that can validate the XML documentation. Table 4-12 lists the supported tags.

TABLE 4-12 XML tags supported for XML documentation.

Tag	Description
<*summary*>	Describes a method or a class
<*param*>	Describes the parameters of a method
<*returns*>	Describes the return value of a method
<*remarks*>	Adds information that supplements the information provided in the <*summary*> tag
<*exception*>	Documents exceptions that are thrown by a method
<*permission*>	Describes the permission needed to access methods using *CodeAccessSecurity.demand*
<*seealso*>	Lists references to related and relevant *documentation*

The XML documentation is automatically displayed in the IntelliSense in the X++ editor.

You can extract the written XML documentation for an AOT project by using the Add-Ins menu option Extract XML Documentation. One XML file is produced that contains all of the documentation for the elements inside the project. You can also use this XML file to publish the documentation.

The following code example shows XML documentation for a static method on the *Global* class:

```
/// <summary>
/// Converts an X++ utcDateTime value to a .NET System.DateTime object.
/// </summary>
/// <param name="_utcDateTime">
/// The X++ utcDateTime to convert.
/// </param>
/// <returns>
/// A .NET System.DateTime object.
/// </returns>
static client server anytype utcDateTime2SystemDateTime(utcDateTime _utcDateTime)
{
    return CLRInterop::getObjectForAnyType(_utcDateTime);
}
```

Classes and interfaces

You define types and their structure in the AOT, not in the X++ language. Other programming languages that support type declarations do so within code, but Microsoft Dynamics AX supports an object layering feature that accepts X++ source code customizations to type declaration parts that encompass variable declarations and method declarations. Each part of a type declaration is managed as a separate compilation unit, and model data is used to manage, persist, and reconstitute dynamic types whose parts can include compilation units from many object layers.

You use X++ to define logic, including method profiles (return value, method name, and parameter type and name). You use the X++ editor to add new methods to the AOT, so you can construct types without leaving the X++ editor.

You use X++ class declarations to declare protected instance variable fields that are members of application logic and framework reference types. You can't declare private or public variable fields. You can declare classes as abstract if they are incomplete type specifications that can't be instantiated. You can also declare them final if they are complete specifications that can't be further specialized. The following code provides an example of an abstract class declaration header:

```
abstract class MyClass
{
}
```

You can also structure classes into single-inheritance generalization or specialization hierarchies in which derived classes inherit and override members of base classes. The following code shows an example of a derived class declaration header that specifies that *MyDerivedClass* extends the abstract base class *MyClass*. It also specifies that *MyDerivedClass* is final and can't be further specialized by another class. Because X++ doesn't support multiple inheritance, derived classes can extend only one base class.

```
final class MyDerivedClass extends MyClass
{
}
```

X++ also supports interface type specifications that specify method signatures but don't define their implementation. Classes can implement more than one interface, but the class and its derived classes should together provide definitions for the methods declared in all the interfaces. If it fails to provide the method definitions, the class itself is treated as abstract and cannot be instantiated. The following code provides an example of an interface declaration header and a class declaration header that implements the interface:

```
interface MyInterface
{
    void myMethod()
    {
    }
}
class MyClass implements MyInterface
{
    void myMethod()
    {
    }
}
```

Fields

A field is a class member that represents a variable and its type. Fields are declared in class declaration headers; each class and interface has a definition part with the name *classDeclaration* in the AOT. Fields are accessible only to code statements that are part of the class declaration or derived class declarations. Assignment statements are not allowed in class declaration headers. The following example demonstrates how variables are initialized with assignment statements in a *new* method:

```
class MyClass
{
    str s;
    int i;
    MyClass1 myClass1;

    public void new()
    {
        i = 0;
        myClass1 = new MyClass1();
    }
}
```

Methods

A method on a class is a member that uses statements to define the behavior of an object. An interface method is a member that declares an expected behavior of an object. The following code provides an example of a method declaration on an interface and an implementation of the method on a class that implements the interface:

```
interface MyInterface
{
    public str myMethod()
    {
    }
}
```

```
class MyClass implements MyInterface
{
    public str myMethod();
    {
        return "Hello World";
    }
}
```

Methods are defined with public, private, or protected access modifiers. If an access modifier is omitted, the method is publicly accessible. The X++ template for new methods provides the private access specifier. Table 4-13 describes additional method modifiers supported by X++.

TABLE 4-13 Method modifiers supported by X++.

Modifier	Description
abstract	Abstract methods have no implementation. Derived classes must provide definitions for abstract methods.
client	Client methods can execute only on a MorphX client. The *client* modifier is allowed only on static methods.
delegate	Delegate methods cannot contain implementation. Event handlers can subscribe to delegate methods. The *delegate* modifier is allowed only on instance methods.
display	Display methods are invoked each time a form is redrawn. The *display* modifier is allowed only on table, form, form data source, and form control methods.
edit	The *edit* method is invoked each time a form is redrawn or a user provides input through a form control. The *edit* modifier is allowed only on table, form, and form data source methods.
final	Final methods can't be overridden by methods with the same name in derived classes.
server	Server methods can execute only on an Application Object Server (AOS). The *server* modifier is allowed on all table methods and on static class methods.
static	Static methods are called using the name of the class rather than the name of an instance of the class. Fields can't be accessed from within a static method.

Method parameters can have default values that are used when parameters are omitted from method invocations. The following code sample prints "Hello World" when *myMethod* is invoked with no parameters:

```
public void myMethod(str s = "Hello World")
{
    print s;
    pause;
}

public void run()
{
    this.myMethod();
}
```

A constructor is a special instance method that is invoked to initialize an object when the *new* operator is executed by the Microsoft Dynamics AX runtime. You can't call constructors directly from X++ code. The sample on the next page provides an example of a class declaration header and an instance constructor method that takes one parameter as an argument.

```
class MyClass
{
    int i;

    public void new(int _i)
    {
        i = _i;
    }
}
```

Delegates

The purpose of delegates is to expose extension points where add-ons and customizations can extend the application in a lightweight manner without injecting logic into the base functionality. Delegates are methods without any implementation. Delegates are always public and cannot have a return value. You declare a delegate using the *delegate* keyword. You invoke a delegate using the same syntax as a standard method invocation:

```
class MyClass
{
    delegate void myDelegate(int _i)
    {
    }

    private void myMethod()
    {
        this.myDelegate(42);
    }
}
```

When a delegate is invoked, the runtime automatically invokes all event handlers that subscribe to the delegate. There are two ways of subscribing to delegates: declaratively and dynamically. The runtime does not define the sequence in which event handlers are invoked. If your logic relies on an invocation sequence, you should use mechanisms other than delegates and event handlers.

To subscribe declaratively, right-click a delegate in the AOT and then select New Event Handler Subscription. On the resulting event handler node in the AOT, you can specify the class and the static method that will be invoked. The class can be either an X++ class or a .NET class.

To subscribe dynamically, you use the keyword *eventhandler*. Notice that when subscribing dynamically, the event handler is an instance method. It is also possible to unsubscribe.

```
class MyEventHandlerClass
{
    public void myEventHandler(int _i)
    {
        ...
    }

    public static void myStaticEventHandler(int _i)
    {
        ...
    }

    public static void main(Args args)
    {
        MyClass myClass = new MyClass();
        MyEventHandlerClass myEventHandlerClass = new MyEventHandlerClass();

        //Subscribe
        myClass.myDelegate += eventhandler(myEventHandlerClass.myEventHandler);
        myClass.myDelegate +=
                        eventhandler(MyEventHandlerClass::myStaticEventHandler);

        //Unsubscribe
        myClass.myDelegate -= eventhandler(myEventHandlerClass.myEventHandler);
        myClass.myDelegate -=
                        eventhandler(MyEventHandlerClass::myStaticEventHandler);
    }
}
```

Regardless of how you subscribe, the event handler must be public, return *void*, and have the same parameters as the delegate.

Note Cross-tier events are not supported.

As an alternative to delegates, you can achieve a similar effect by using pre- and post-event handlers.

Pre- and post-event handlers

You can subscribe declaratively to any class and record type method by using the same procedure as for delegates. The event handler is invoked either before or after the method is invoked. Event handlers for pre- and post-methods must be public, static, void, and either take the same parameters as the method or one parameter of the *XppPrePostArgs* type.

The simplest, type-safe implementation uses syntax where the parameters of the method and the event handler method match.

```
class MyClass
{
    public int myMethod(int _i)
    {
        return _i;
    }
}

class MyEventHandlerClass
{
    public static void myPreEventHandler(int _i)
    {
        if (_i > 100)
        {
            ...
        }
    }

    public static void myPostEventHandler(int _i)
    {
        if (_i > 100)
        {
            ...
        }
    }
}
```

If you need to manipulate either the parameters or the return value, the event handler must take one parameter of the *XppPrePostArgs* type.

To create such an event handler, right-click the class, and then select New pre- or post-event handler. The *XppPrePostArgs* class provides access to the parameters and the return values of the method. You can even alter parameter values in pre-event handlers and alter the return value in post-event handlers.

```
class MyClass
{
    public int myMethod(int _i)
    {
        return _i;
    }
}

class MyEventHandlerClass
{
    public static void myPreEventHandler(XppPrePostArgs _args)
    {
        if (_args.existsArg('_i') &&
            _args.getArg('_i') > 100)
        {
            _args.setArg('_i', 100);
        }

    }

    public static void myPostEventHandler(XppPrePostArgs _args)
    {
        if (_args.getReturnValue() < 0)
        {
            _args.setReturnValue(0);
        }
    }
}
```

Attributes

Classes and methods can be decorated with attributes to convey declarative information to other code, such as the runtime, the compiler, frameworks, or other tools. To decorate the class, you insert the attribute in the *classDeclaration* element. To decorate a method, you insert the attribute before the method declaration:

```
[MyAttribute("Some parameter")]
class MyClass
{
    [MyAttribute("Some other parameter")]
    public void myMethod()
    {
        ...
    }
}
```

The first attribute that was built in Microsoft Dynamics AX 2012 was the *SysObsoleteAttribute* attribute. By decorating a class or a method with this attribute, any consuming code is notified during compilation that the target is obsolete. You can create your own attributes by creating classes that extend the *SysAttribute* class:

```
class MyAttribute extends SysAttribute
{
    str parameter;

    public void new(str _parameter)
    {
        parameter = _parameter;
        super();
    }
}
```

Code access security

Code access security (CAS) is a mechanism designed to protect systems from dangerous APIs that are invoked by untrusted code. CAS has nothing to do with user authentication or authorization; it is a mechanism allowing two pieces of code to communicate in a manner that cannot be compromised.

Caution X++ developers are responsible for writing code that conforms to Trustworthy Computing guidelines. You can find those guidelines in the white paper "Writing Secure X++ Code," available from the Microsoft Dynamics AX Developer Center (*http://msdn.microsoft.com/en-us/dynamics/ax*).

In the Microsoft Dynamics AX implementation of CAS, trusted code is defined as code from the AOT running on the Application Object Server (AOS). The first part of the definition ensures that the code is written by a trusted X++ developer. Developer privileges are the highest level of privileges in Microsoft Dynamics AX and should be granted only to trusted personnel. The second part of the definition ensures the code that the trusted developer has written hasn't been tampered with. If the code executes outside the AOS—on a client, for example—it can't be trusted because of the possibility that it was altered on the client side before execution. Untrusted code also includes code that is executed through the *runBuf* and *evalBuf* methods. These methods are typically used to execute code generated at run time based on user input.

CAS enables a secure handshake between an API and its consumer. Only consumers who provide the correct handshake can invoke the API. Any other invocation raises an exception.

The secure handshake is established through the *CodeAccessPermission* class or one of its specializations. The consumer must request permission to call the API, which is done by calling *CodeAccessPermission.assert*. The API verifies that the consumer has the correct permissions by calling *CodeAccessPermission.demand*. The *demand* method searches the call stack for a matching assertion. If untrusted code exists on the call stack before the matching assertion, an exception is raised. This process is illustrated in Figure 4-1.

The following code contains an example of a dangerous API protected by CAS and a consumer providing the correct permissions to invoke the API:

```
class WinApiServer
{
    // Delete any given file on the server
    public server static boolean deleteFile(Filename _fileName)
    {
        FileIOPermission    fileIOPerm;

        // Check file I/O permission
        fileIOPerm = new FileIOPermission(_fileName, 'w');
        fileIOPerm.demand();

        // Delete the file

        System.IO.File::Delete(_filename);
    }
}

class Consumer
{
    // Delete the temporary file on the server
    public server static void deleteTmpFile()
    {
        FileIOPermission    fileIOPerm;
        FileName            filename = @'c:\tmp\file.tmp';

        // Request file I/O permission
        fileIOPerm = new FileIOPermission(filename, 'w');
        fileIOPerm.assert();

        // Use CAS protected API to delete the file
        WinApiServer::deleteFile(filename);
    }
}
```

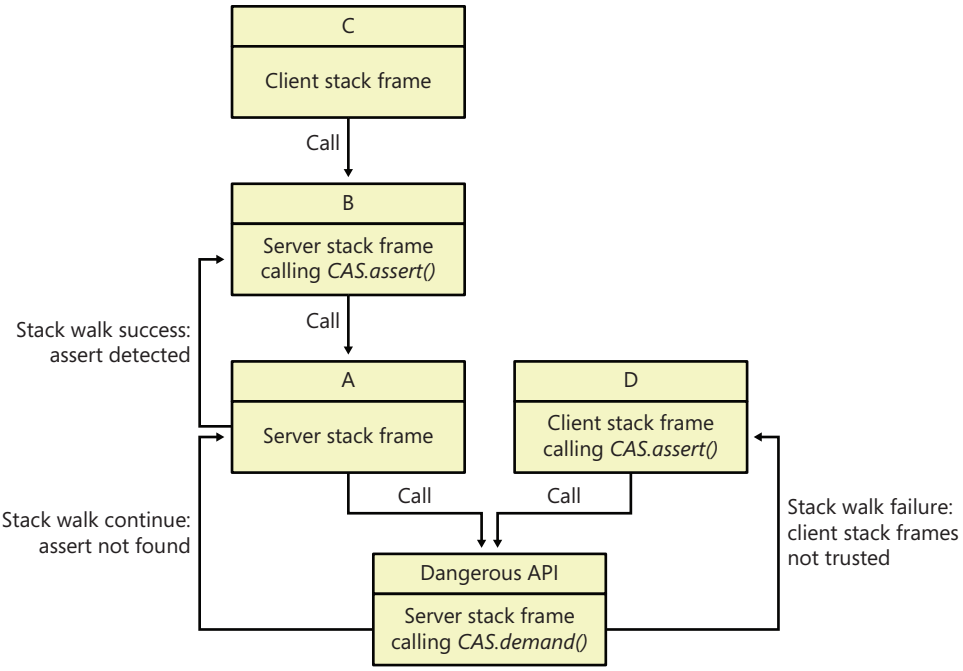

FIGURE 4-1 CAS stack frame walk.

WinAPIServer::deleteFile is considered to be a dangerous API because it exposes the .NET API *System.IO.File::Delete(string fileName)*. Exposing this API on the server is dangerous because it allows the user to remotely delete files on the server, possibly bringing the server down. In the example, *WinApiServer::deleteFile* demands that the caller has asserted that the input file name is valid. The demand prevents use of the API from the client tier and from any code not stored in the AOT.

Caution When using *assert*, make sure that you don't create a new API that is just as dangerous as the one that CAS has secured. When you call *assert*, you are asserting that your code doesn't expose the same vulnerability that required the protection of CAS. For example, if the *deleteTmpFile* method in the previous example had taken the file name as a parameter, it could have been used to bypass the CAS protection of *WinApi::deleteFile* and delete any file on the server.

Compiling and running X++ as .NET CIL

All X++ code is compiled into Microsoft Dynamics AX runtime bytecode intermediate format. This format is used by the Microsoft Dynamics AX runtime for Microsoft Dynamics AX client and server code.

Further, classes and tables are compiled into .NET common intermediate language (CIL). This format is used by X++ code executed by the Batch Server and in certain other scenarios.

The X++ compiler only generates Microsoft Dynamics AX runtime bytecode to generate CIL code; you must manually press either the Generate Full IL or Generate Incremental IL button. Both are available on the toolbar.

The main benefit of running X++ as CIL is performance. Generally the .NET runtime is significantly faster than the X++ runtime. In certain constructions, the performance gain is particularly remarkable:

- **Constructs with many method calls—**Under the covers in the X++ runtime, any method call happens through reflection, whereas in CIL, this happens at the CPU level.
- **Constructions with many short-lived objects—**Garbage collection in the Microsoft Dynamics AX runtime is deterministic, which means that whenever an instance goes out of scope, the entire object graph is analyzed to determine if any objects can be deallocated. In the .NET CLR, garbage collection is indeterministic, which means that the runtime determines the optimal time for reclaiming memory.
- **Constructions with extensive use of.NET interop—**When running as X++ code as CIL, all conversion and marshaling between the runtimes are avoided.

Note The capability to compile X++ into CIL requires that X++ syntax be as strict as the syntax in managed code. The most noteworthy change is that overridden methods must now have the same signature as the base method. The only permissible discrepancy is the addition of optional parameters.

One real-life example where running X++ code as .NET CIL makes a significant difference is in the compare tool. The compare algorithm is implemented as X++ code in the *SysCompareText* class. Even though the algorithm has few method calls, few short-lived objects, and no .NET interop, the switch to CIL means that within a time frame of 10 seconds, it is now possible to compare two 3,500-line texts, whereas the AX runtime can only handle 600 lines in the same time frame. The complexity of the algorithm is exponential. In other words the performance gain gets even more significant the larger the texts become.

All services and batch jobs will automatically run as CIL. If you want to force X++ code to run as CIL in non-batch scenarios, you use the methods *runClassMethodIL* and *runTableMethodIL* on the *Global* class. The IL entry point must be a static server method that returns a container and takes one container parameter:

```
class MyClass
{
    private static server container addInIL(container _parameters)
    {
        int p1, p2;
        [p1, p2] = _parameters;
```

```
        return [p1+p2];
    }

    public server static void main(Args _args)
    {
        int result;
        XppILExecutePermission permission = new XppILExecutePermission();
        permission.assert();
        [result] = runClassMethodIL(classStr(MyClass),
                                    staticMethodStr(MyClass, addInIL), [2, 2]);
        info(strFmt("The result from IL is: %1", result));
    }
}
```

Design and implementation patterns

So far, this chapter has described the individual elements of X++. You've seen that statements are grouped into methods, and methods are grouped into classes, tables, and other model element types. These structures enable you to create X++ code at a higher level of abstraction. The following example shows how an assignment operation can be encapsulated into a method to clearly articulate the intention of the code.

```
control.show();
```

is at a higher level of abstraction than

```
flags = flags | 0x0004;
```

By using patterns, developers can communicate their solutions more effectively and reuse proven solutions to common problems. Patterns help readers of source code to quickly understand the purpose of a particular implementation. Bear in mind that even as a code author, you spend more time reading source code than writing it.

Implementations of patterns are typically recognizable by the names used for classes, methods, parameters, and variables. Arguably, naming these elements so that they effectively convey the intention of the code is the developer's most difficult task. Much of the information in existing literature on design patterns pertains to object-oriented languages, and you can benefit from exploring that information to find patterns and techniques you can apply when you're writing X++

code. Design patterns express relationships or interactions between several classes or objects. They don't prescribe a specific implementation, but they do offer a template solution for a typical design problem. In contrast, implementation patterns are implementation specific and can have a scope that spans only a single statement.

This section highlights some of the most frequently used patterns specific to X++. More descriptions are available in the Microsoft Dynamics AX SDK on MSDN.

Class-level patterns

These patterns apply to classes in X++.

Parameter method

To set and get a class field from outside the class, you should implement a parameter method. The parameter method should have the same name as the field and be prefixed with *parm*. Parameter methods come in two flavors: *get-only* and *get/set*.

```
public class Employee
{
    EmployeeName name;

    public EmployeeName parmName(EmployeeName _name = name)
    {
        name = _name;
        return name;
    }
}
```

Constructor encapsulation

The purpose of the constructor encapsulation pattern is to enable Liskov's class substitution principle. In other words, with constructor encapsulation, you can replace an existing class with a customized class without using the layering system. Just as in the layering system, this pattern enables changing the logic in a class without having to update any references to the class. Be careful to avoid overlayering because it often causes upgrade conflicts.

Classes that have a static *construct* method follow the constructor encapsulation pattern. The *construct* method should instantiate the class and immediately return the instance. The *construct* method must be static and shouldn't take any parameters.

When parameters are required, you should implement the static *new* methods. These methods call the *construct* method to instantiate the class and then call the parameter methods to set the parameters. In this case, the *construct* method should be private:

```
public class Employee
{
    ...
    protected void new()
    {
    }

    protected static Employee construct()
    {
        return new Employee();
    }

    public static Employee newName(EmployeeName name)
    {
        Employee employee = Employee::construct();

        employee.parmName(name);
        return employee;
    }
}
```

Factory

To decouple a base class from derived classes, use the *SysExtension* framework. This framework enables the construction of an instance of a class based on its attributes. This pattern enables add-ons and customizations to add new subclasses without touching the base class or the factory method:

```
class BaseClass
{
    ...
    public static BaseClass newFromTableName(TableName _tableName)
    {
        SysTableAttribute attribute = new SysTableAttribute(_tableName);

        return SysExtensionAppClassFactory::getClassFromSysAttribute(
            classStr(BaseClass), attribute);
    }
}

[SysTableAttribute(tableStr(MyTable))]
class Subclass extends BaseClass
{
   ...
}
```

Serialization with the *pack* and *unpack* methods

Many classes require the capability to serialize and deserialize themselves. Serialization is an operation that extracts an object's state into value-type data; deserialization creates an instance from that data.

X++ classes that implement the *Packable* interface support serialization. The *Packable* interface contains two methods: *pack* and *unpack*. The *pack* method returns a container with the object's state; the *unpack* method takes a container as a parameter and sets the object's state accordingly. You should include a versioning number as the first entry in the container to make the code resilient to old packed data stored in the database when the implementation changes.

```
public class Employee implements SysPackable
{
    EmployeeName name;
    #define.currentVersion(1)
    #localmacro.CurrentList
        name
    #endmacro
    ...

    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:   //The version number is unsupported
                return false;
        }
        return true;
    }
}
```

Table-level patterns

The patterns described in this section—the *find* and *exists* methods, polymorphic associations (Table/Group/All), and Generic Record References—apply to tables.

Find and e*xists* methods

Each table must have the two static methods *find* and *exists*. They both take the primary keys of the table as parameters and return the matching record or a Boolean value, respectively. Besides the primary keys, the *Find* method also takes a *Boolean* parameter that specifies whether the record should be selected for update.

For the *CustTable* table, these methods have the following profiles:

```
static CustTable find(CustAccount _custAccount, boolean _forUpdate = false)
static boolean exist(CustAccount _custAccount)
```

Polymorphic associations

The Table/Group/All pattern is used to model a polymorphic association to either a specific record in another table, a collection of records in another table, or all records in another table. For example, a record could be associated with a specific item, all items in an item group, or all items.

You implement the Table/Group/All pattern by creating two fields and two relations on the table. By convention, the name of the first field has the suffix *Code*; for example, *ItemCode*. This field is modeled using the base enum *TableGroupAll*. The name of the second field usually has the suffix *Relation*; for example, *ItemRelation*. This field is modeled by using the extended data type that is the primary key in the foreign tables. The two relations are of the type *Fixed* field relation. The first relation specifies that when the *Code* field equals 0 (*TableGroupAll::Table*), the *Relation* field equals the primary key in the foreign master data table. The second relation specifies that when the *Code* field equals 1 (*TableGroupAll::Group*), the *Relation* field equals the primary key in the foreign grouping table.

Figure 4-2 shows an example.

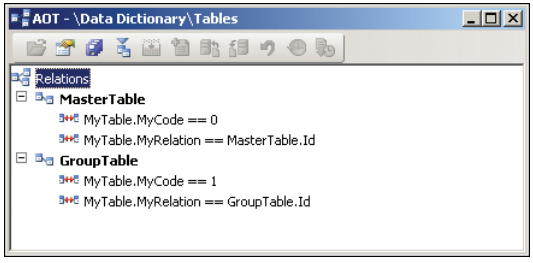

FIGURE 4-2 A polymorphic association.

Generic record reference

The Generic Record Reference pattern is a variation of the Table/Group/All pattern. This pattern is used to model an association to a foreign table. It comes in three flavors: (a) an association to any record in a specific table, (b) an association to any record in a fixed set of specific tables, and (c) an association to any record in any table.

All three flavors of this pattern are implemented by creating a field that uses the *RefRecId* extended data type.

To model an association to any record in a specific table (flavor a), a relation is created from the *RefRecId* field to the *RecId* field of the foreign table, as illustrated in Figure 4-3.

FIGURE 4-3 An association to a specific table.

For flavors b and c, an additional field is required. This field is created by using the *RefTableId* extended data type. To model an association to any record in a fixed set of specific tables (flavor b), a relation is created for each foreign table from the *RefTableId* field to the *TableId* field of the foreign table, and from the *RefRecId* field to the *RecId* field of the foreign table, as shown in Figure 4-4.

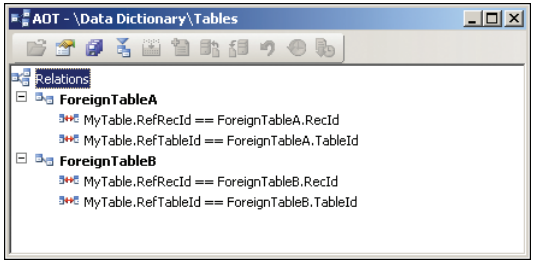

FIGURE 4-4 An association to any record in a fixed set of tables.

To model an association to any record in any table (flavor c), a relation is created from the *RefTableId* field to the generic table *Common TableId* field and from the *RefRecId* field to *Common RecId* field, as shown in Figure 4-5.

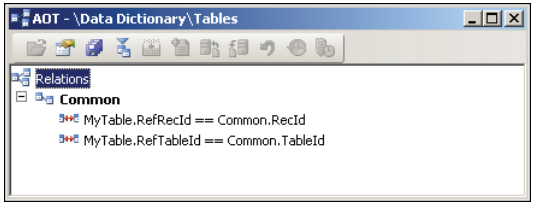

FIGURE 4-5 An association to any record in any table.