Глава 4

Язык программирования X++

В этой главе

- Введение
- Задания
- Система типов
- Синтаксис
- Classes and interfaces
- Code access security
- Compiling and running X++ as .NET CIL
- Design and implementation patterns

Введение

Язык X++ – это объектно-ориентированный язык программирования, предназначенный для разработки приложения и работы с данными. Язык является объектно-ориентированным, поскольку он поддерживает абстракции объектов, иерархии абстракций, полиморфизм и инкапсуляцию. Язык является ориентированным на разработку ERP-приложений, поскольку он включает ключевые слова, такие как client, server, changecompany и display, которые используются для написания клиент-серверных приложений планирования ресурсов предприятий (enterprise resource planning, ERP). Язык является ориентированным на работу с данными, поскольку он включает ключевые слова, такие как firstFast, forceSelectOrder и forUpdate, а также синтаксис запросов к базе данных, которые чрезвычайно полезны для программирования приложений баз данных.

Для редактирования структуры прикладных типов используются дизайнеры и инструменты Microsoft Dynamics AX. Поведение прикладных типов определяется путем написания X++-кода с использованием редактора X++. Компилятор X++ затем компилирует данный исходный код в промежуточный формат байт-кода. Модель данных, исходный код на X++,

промежуточный байт-код и .NET common intermediate language (CIL) хранятся в базе данных модели.

Среда времени выполнения Microsoft Dynamics AX динамически формирует объектные типы путем загрузки байт-кода из наивысшего прикладного слоя. Затем из этих динамических типов создаются экземпляры объектов. Упрощенно можно сказать, что компилятор генерирует .NET CIL из исходного кода X++ на самом наивысшем уровне. За дополнительной информацией о технологии слоев в Microsoft Dynamics AX обратитесь к главе 21.

В данной главе описывается система типов Microsoft Dynamics AX и основные возможности языка X++, используемые для написания ERP-приложений. Представленная здесь информация поможет в будущем избежать общеизвестных проблем программирования, которые существуют из-за специфики реализации языка X++. Для углубленного изучения системы типов и языка X++ обратитесь к руководству разработчика Microsoft Dynamics AX 2012 SDK на MSDN.

Задания

Задания – это глобально определенные функции, которые исполняются в рабочей среде разработки выполнения толстого клиента. Задания часто используются разработчиками, поскольку они легко запускаются из среды разработки MorphX (нажатием клавиши F5 или путем выбора пункта Перейти из меню Команда редактора). Задания не следует использовать для реализации функциональности приложения. Примеры из данной главы написаны в виде заданий.

Задания – это прикладные элементы модели, которые создаются при помощи Репозитария прикладных объектов AOT. Нижеприведенный код представляет собой пример задания, которое печатает строку «Hello World» в автоматически генерируемое окно сообщений. Оператор *pause* приостанавливает исполнение программы и ожидает ввода пользователя в диалоговом окне.

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

Система типов

Среда времени выполнения Microsoft Dynamics AX управляет размещением переменных значимых типов в стеке, а переменных ссылочных типов – в куче. Стек вызовов – это структура в памяти, в которой содержатся сведения об активных методах, вызванных в процессе исполнения программы. Куча – это область памяти, в которой выделяется место для размещения объектов. Объекты в куче автоматически удаляются средой времени выполнения Microsoft Dynamics AX.

Значимые типы

Встроенные примитивные типы данных, расширенные типы данных, перечислимые типы и встроенные типы-коллекции являются значимыми типами данных.

- Примитивные типы данных включают: *boolean*, *int*, *int64*, *real*, *date*, *utcDateTime*, *timeofday*, *str* и *guid*.
- Расширенные типы данных – это специализированные примитивные типы и специализированные перечислимые типы.
- Перечислимые типы – это собственно перечислимые типы и расширенные типы данных, основанные на них.
- Коллекции – это встроенные типы массива и контейнера.

По умолчанию Microsoft Dynamics AX инициализирует переменные значимых типов нулевым значением, соответствующим типу переменной. Переменным значимых типов не может быть присвоено значение *null*. Когда переменные значимых типов используются в качестве параметров при вызове методов или в операторах присваивания, производиться копирование значений переменных. Поэтому две переменных значимого типа не могут ссылаться на одно и то же значение.

Ссылочные типы

Ссылочные типы включают типы записи, класса и интерфейса.

- К типу Запись относятся *table*, *map* и *view*. Пользовательские типы записи динамически формируются из прикладных слоев. Типы записи среды времени выполнения Microsoft Dynamics AX доступны через системный программный интерфейс.

Примечание. Несмотря на то, что это не отображается в AOT, все типы записи реализуют методы, которые являются членами системного класса *xRecord*.

- Пользовательские типы классов динамически формируются на основании прикладных слоев и классов, доступных через системный программный интерфейс среды времени выполнения Microsoft Dynamics AX.
- Интерфейсы являются спецификациями типов и в среде времени выполнения Microsoft Dynamics AX экземпляры интерфейсов создаваться не могут. Однако классы могут реализовывать интерфейсы.

Переменные ссылочных типов содержат ссылки на объекты, экземпляры которых инициализирует среда времени выполнения Microsoft Dynamics AX, используя динамически сформированные на основании прикладных слоев типы. Среда времени выполнения Microsoft Dynamics AX также выполняет освобождение памяти (сборка мусора), занимаемой данными объектами, когда они выходят за пределы области видимости и на них больше нет ссылок из других объектов. Переменные типов записей ссылаются на объекты, экземпляры которых автоматически создаются средой времени выполнения Microsoft Dynamics AX. Экземпляры классов в свою очередь создаются с использованием оператора *new*. Копии ссылок на объекты передаются в качестве параметров в вызовы методов и присваиваются ссылочным переменным, поэтому две переменных могут ссылаться на один и тот же объект.

Дополнительная информация. Не все узлы в AOT являются объявлениями типов. Некоторые объявления типов классов – это просто синтаксически удобные для чтения выражения. Например, заголовок всех экранных форм толстого клиента Microsoft Dynamics AX содержит объявление класса *FormRun*. Однако *FormRun* также является названием класса системного API. Разрешение такого рода объявлений – это синтаксическое смягчение, поскольку технически в иерархии классов Microsoft Dynamics AX нельзя иметь два типа с одинаковым названием.

Иерархии типов

Язык X++ поддерживает определение иерархий типов, которые устанавливают отношения обобщения и специализации между разными типами классов. Например, метод оплаты чеком – это тип метода оплаты. Иерархия типов позволяет повторно использовать код. Повторно используемый код основывается на использовании базовых типов, определенных выше в дереве иерархии типов, и может повторно использоваться производными типами, определенными ниже в иерархии типов.

Совет. Для просмотра иерархии для любого типа в AOT можно использовать инструмент Иерархия объектов MorphX.

В данном разделе рассказывается о базовых типах, предусматриваемых средой времени выполнения Microsoft Dynamics AX, а также описывается, как они наследуются в иерархиях типов.

Внимание. Система типов Microsoft Dynamics AX известна как система со слабой типизацией, поскольку язык X++ принимает определенные присваивания типов, которые являются неправильными и ведут к ошибкам разработки. Внимательно изучите предостережения в последующих разделах и пытайтесь избегать конструкций, использующих слабую типизацию, при написании X++-кода.

Тип *anytype*

В системе типов Microsoft Dynamics AX не существует строгой иерархии типов с конкретным типом, базовым для всех остальных. Тем не менее *anytype* имитирует такой базовый тип. Переменные типа *anytype* ведут себя подобно значимым типам, когда им присваиваются переменные значимых типов, и подобно ссылочным типам, когда им присваиваются переменные ссылочных типов. Класс *SysAnyType* может использоваться для упаковки любых типов, включая значимые, превращая их в ссылочные типы.

Тип *anytype*, показанный в следующем примере кода, является «синтаксическим сахаром», который позволяет методам принимать параметр любого типа и возвращать значение любого типа.

```
static str queryRange(anytype _from, anytype _to)
{
```

```
    return SysQuery::range(_from,_to);
}
```

В коде можно объявлять переменные типа *anytype*. Однако действительный тип данных переменной типа *anytype* фиксируется при первом присваивании и не может быть изменен позже, как показано в следующем примере.

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

Тип *common*

Тип *common* – это базовый тип для всех типов записей. Подобно *anytype*, типы записей являются контекстно-зависимыми типами, переменные которых могут использоваться, как если бы они указывали на единичные записи или же на курсор записи, который может служить итератором по набору записей базы данных.

Использование типа *common* позволяет приводить один тип записи к другому (возможно несовместимому), как показано в следующем примере.

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

Таблицы в Microsoft Dynamics AX также поддерживают наследование и полиморфизм. Эта возможность предлагает безопасный метод предоставления общего доступа к методам и полям таблицы родителя. То есть возможно переопределить в наследнике только метод таблицы, а не поля. Базовая таблица-родитель может быть настроена в свойствах как *abstract* или *final*.

Карты соответствия в AOT являются более безопасным методом обобщения различных типов записей, которые следует использовать для предотвращения присвоения несовместимых типов записей. Карта соответствия определяет поля и методы, которые безопасно оперируют одним или более типами записей.

Обратите внимание, что вызовы методов для типа *common* не проверяются компилятором. Например, следующий вызов метода принимается компилятором несмотря на то, что указанный метод не существует.

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

По этой причине следует использовать отражение для проверки существования метода у переменной типа *common* перед вызовом этого метода. За дополнительной информацией обращайтесь к главе 20.

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

Тип *object*

Встроенный тип *object* – это слабый ссылочный тип, переменные которого ссылаются на объекты типа класса или интерфейса в иерархии классов Microsoft Dynamics AX. Система типов в Microsoft Dynamics AX позволяет программистам неявно приводить объекты базовых типов к объектам производных типов, а также приводить объекты производных типов к объектам базовых, как показано в следующем примере.

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

Обратите внимание, что тип *object* позволяет использовать оператор присваивания и приводить один класс к другому, даже несовместимому, как показано в следующем примере кода. Однако это, скорее всего, приведет к ошибке времени выполнения, когда будет использован оператор, не ожидавший такого типа объекта.

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

Лучше используйте операторы *is* и *as* вместо оператора присвоения для предотвращения приведения несовместимых типов. Оператор *is* определяет, является ли экземпляр определенным типом, а оператор *as* приводит экземпляр к определенному типу или приводит к значению *null*, если типы не совместимы. Операторы *is* и *as* используются при приведении типов в методах класса и таблицы.

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

Используйте тип *object* для позднего связывания методов аналогично ключевому слову *dynamic* в языке C#.

Примите к сведению, что произойдет ошибка выполнения, если вызываемый метод не существует.

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

Расширенные типы данных

Расширенные типы данных создаются с помощью AOT и используются для моделирования конкретных значений и иерархий данных. К примеру, расширенный тип данных *Name* является строковым, а типы данных *CustName* и *VendName* наследуются от типа данных *Name*.

Язык X++ поддерживает расширенные типы данных, но не предлагает никакой проверки преобразования типов в соответствии с иерархией расширенных типов данных. X++ рассматривает любой расширенный тип данных как его базовый тип, поэтому допускается код, подобный тому, который приведен ниже.

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

Однако при корректном использовании расширенные типы данных существенно улучшают читаемость X++-кода. Гораздо легче понять предназначение типа данных *CustName*, чем типа данных *string*, даже если оба используются для объявления строковых переменных.

Но расширенные типы данных предоставляют гораздо больше, чем просто улучшение читаемости кода. Для любого расширенного типа данных можно указать, как значения данного типа должны отображаться в интерфейсе пользователя. Помимо этого для расширенного типа данных можно указать отношение к определенной таблице базы данных. Инфраструктура экранных форм использует это отношение для построения формы выпадающего списка, связанной с выбранным управляющим элементом на форме, даже если этот управляющий элемент не связан с каким-либо источником данных. Для расширенного типа данных строкового типа также можно указать максимальный размер строки для этого типа. Указанный размер будет использован при создании полей таблиц, использующих данный расширенный тип данных, в базе данных. Так как размер в данном случае указывается только на одном прикладном элементе, это позволяет легко изменить этот размер в дальнейшем, если возникнет такая необходимость.

Синтаксис

Язык программирования X++ принадлежит к семейству языков, которые используют фигурные скобки для разделения синтаксических блоков (таких как C, C++, C# и Java). Если вы знакомы с любым из данных языков, то у вас не будет трудностей с чтением и пониманием синтаксиса языка X++.

В отличие от большинства других языков программирования, X++ является регистро-независимым. Однако рекомендуется использование нотации *PascalCasing* для названий классов и нотации *camelCasing* для названий переменных. (Более подробные рекомендации по написанию X++-кода доступны в руководстве разработчика Microsoft Dynamics AX 2012 SDK.) Для автоматической смены регистра символов в X++-коде в соответствии с рекомендациями Best Practices можно воспользоваться инструментом MorphX Смена регистра в исходном коде, который доступен из подменю Надстройки в AOT.

CLR-типы, которые чувствительны к регистру символов, являются важным исключением из рекомендаций по наименованию объектов в X++. Принципы работы с CLR-типами объясняются далее в этой главе.

Объявление переменных

Объявления переменных должны располагаться в начале кода методов. В табл. 4-1 приведены примеры объявления переменных значимых типов и ссылочных типов, а также примеры инициализации объявленных переменных. Примеры описания параметров в методах приведены в разделе «Классы и интерфейсы» далее в этой главе.

Табл. 4–1. Примеры объявления переменных в X++

Тип	**Примеры**
anytype	anytype type = null; anytype type = 1;
Перечислимые типы	NoYes theAnswer = NoYes::Yes;
boolean	boolean b = true;

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

date	date d = 31\12\2008;

Табл. 4–1. Примеры объявления переменных в X++ (*окончание*)

Тип	Примеры
Расширенные типы данных	Name name = "name";
guid	guid g = newguid();
int	int i = -5; int h = 0xAB;

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

Объектные типы	Object obj = null; MyClass myClass = new MyClass(); System.Text.StringBuilder sb = new System.Text.StringBuilder();
Real	real r1 = 3.14; real r2 = 1.0e3;
Типы записей	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;

Примечание. Значения строковым переменным могут назначаться с использованием одинарных или двойных кавычек. Считается хорошим тоном использовать одинарные кавычки для присвоения значений программным/системным переменным с типом строка, таким как название файла, а двойные кавычки использовать для переменных, используемых в пользовательском интерфейсе. Примеры в данной главе следуют этой рекомендации.

Объявление переменных, которые называются так же, как и тип этих переменных, является общепринятой практикой. С первого взгляда это может сбить с толку. Проанализируйте данный класс и его метод установки/считывания значения поля.

```
Class Person
{
   Name name;

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

Поскольку язык X++ не различает регистр символов, слово *name* используется в данном примере восемь раз. Три из них являются расширенным типом данных, четыре раза это слово используется для обозначения переменной и один раз как название метода (слово_*name* используется дважды). Для улучшения читаемости кода переменную можно было бы назвать более конкретно, например *personName*. Однако использование более конкретных названий переменных требует и более конкретных названий типов (и их создания, если они еще не существуют). Изменение и названия типа, а также названия переменной на *PersonName* не улучшит читаемость кода. Преимущество данного подхода заключается в том, что если разработчик знает название переменной, то он, скорее всего, знает и тип этой переменной.

Примечание. В предыдущей версии Microsoft Dynamics AX требовалось использование точки с запятой для отделения завершения блока объявления переменных от основного кода. Теперь это не нужно, так как компилятор разрешает эту неоднозначность, читая один маркер вперед, за исключением использования первого вызова статической инструкции CLR. Компилятор по-прежнему принимает дополнительную точку с запятой, но вы вправе ее удалить.

Выражения

Выражения X++ представляют собой последовательность операторов, операндов, значений и переменных, которая приводит к определенному результату. В табл. 4-2 приводится общая информация о типах выражений, разрешенных в X++, совместно с примерами их использования.

Табл. 4–2. Примеры X++-выражений

Категория	Примеры	

Операторы доступа	this
	element
	<datasource>_ds
	<datasource>_q
	x.y
	E::e
	a[x]
	[v1, v2] = c
	Table.Field
	Table.(FieldId)
	(select statement).Field
	System.Type
	System.DayOfWeek::Monday
		//Доступ к членам текущего объекта
		//Доступ к членам формы
		//Доступ к членам источника данных
		//Доступ к членам запроса
		//Доступ к методу объекта
		//Доступ к перечислимому значению
		//Доступ к массиву
		//Доступ к контейнеру
		//Доступ к полю таблицы
		//Доступ к полю таблицы
		//Доступ к результату выполнения запроса
		//Доступ к CLR namespac
		// Доступ к перечислимому CLR-типу

Арифметические операторы	x = y + z
	x = y - z
	x = y * z
	x = y / z
	x = y div z
	x = y mod z
		// Сложение
		// Вычитание
		// Умножение
		// Деление
		// Целочисленное деление
		// Остаток от целочисленного деления

Побитовые операторы	x = y & z
	x = y | z
	x = y ^ z
	x = ~z
		// Логическое И
		// Логическое ИЛИ
		// Побитовое исключающее ИЛИ OR (XOR)
		// Побитовое дополнение

Условные операторы	x ? y : z	

Логические операторы	if (!obj)
	if (a && b)
	if (a || b)
		// Логическое НЕ
		// Логическое И
		// Логическое ИЛИ

Вызов методов	super()
	MyClass::m()
	myObject.m()
	this.m()
	myTable.MyMap::m();
	f()
		// Вызов метода базового объекта
		// Вызов статического метода класса
		// Вызов метода объекта
		// Вызов метода текущего объекта
		// Вызов метода карты соответствий
		// Вызов встроенной функции

Табл. 4–2. Примеры X++-выражений (*окончание*)

Категория	Примеры

Операторы создания объектов	new MyClass() // Создание объекта X++
	new System.DateTime() // Оболочка CLR-объекта и
	// Создание CLR-объекта
	new System.Int32[100]() // CLR создание массива

Круглые скобки	(x)

Операторы отношений	x < y // Меньше чем
	x > y // Больше чем
	x <= y // Меньше либо равно
	x >= y // Больше либо равно
	x == y // Равно
	x != y // Не равно
	select t where t.f like "a*" // Запрос с использованием шаблонов

Операторы сдвига	x = y << z // Сдвиг влево
	x = y >> z // Сдвиг вправо

Конкатенация строк	"Hello" + "World"
Переменные и значения	"string" myVariable

Операторы

Операторы X++ определяют состояние и поведение объектов системы. В табл. 4-3 приводятся примеры операторов языка X++, которые существуют и во многих других языках программирования. Более подробное описание каждого оператора выходит за рамки данной книги.

Табл. 4–3. Примеры операторов X++

Оператор	Пример

Оператор взаимодействия с .NET	System.Text.StringBuilder sb;
	sb = new System.Text.StringBuilder();
	sb.Append("Hello World");
	print sb.ToString();
	pause;

Табл. 4–3. Примеры операторов X++ (*продолжение*)

Оператор	**Пример**

Оператор присваивания	int i = 42;
	i = 1;
	i++;
	++i;
	i--;
	--i;
	i += 1;
	i -= 1;
	this.myDelegate += eventhandler(obj.handler);
	this.myDelegate -= eventhandler(obj.handler);

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

Оператор *breakpoint*	breakpoint; // Приводит к запуску отладчика
Оператор приведения типа	MyObject myObject = object as MyObject; boolean isCompatible = object is MyObject;

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

Табл. 4–3. Примеры операторов X++ (*продолжение*)

Оператор	**Пример**

Составной оператор	int i;
	{
	  i = 3;
	  i++;
	}

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

Оператор *do while*	int i = 4;
	do
	{
	  i++;
	}
	while (i <= 100);

Оператор *flush*	MyTable myTable; flush myTable;

Оператор *for*	int i;
	for (i = 0; i < 42; i++)
	{
	  print i;
	}
	pause;

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

Табл. 4–3. Примеры операторов X++ (*продолжение*)

Оператор	**Пример**

Лоальная функция	static void myJob(Args _args)
	{
	  str myLocalFunction()
	  {
	  return "Hello World";
	  }
	  print myLocalFunction();
	  pause;
	}

Оператор *pause*	print "Hello World"; pause;
Оператор *print*	int i = 42; print i; print "Hello World"; print "Hello World" at 10,5; print 5.2; pause;

Оператор *retry*	try
	{
	  throw error("Force exception");
	}
	catch(exception::Error)
	{
	  retry;
	}

Оператор *return*	int foo()
	{
	  return 42;
	}

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

Табл. 4–3. Примеры операторов X++ (*окончание*)

Оператор	Пример
Системная функция	guid g = newGuid(); print abs(-1);
Оператор *throw*	throw error("Error text");

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

Оператор *unchecked*	unchecked(Uncheck::TableSecurityPermission)
	{
	  this.method();
	}

Оператор *while*	int i = 4;
	while (i <= 100)
	{
	  i++;
	}

Оператор *window*	window 100, 10 at 100, 10; print "Hello World"; pause;

Операторы для работы с данными

Язык X++ содержит встроенную поддержку запросов манипуляции данными. Синтаксис операторов для работы с базой данных похож на язык структурированных запросов (Structured Query Language, SQL). В данном разделе предполагается, что вы на достаточном уровне знакомы с языком SQL. В примере кода ниже показано, как оператор *select* используется для

возврата только первой выбранной записи из таблицы *MyTable*, а также то, как содержимое поля *myField* выводится на экран.

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

Обратите внимание, что * from, часть оператора *select* в данном примере, не является обязательной. Символ звездочка (*) может заменяться списком полей, разделенных запятыми, например *myField2*, *myField3*. Все поля списка полей должны быть определены в таблице, а после ключевого слова *from* разрешается использовать только одну таблицу. Выражение *where* в операторе *select* может содержать любое количество логических операторов и операторов отношений. Ключевое слово *firstOnly* является необязательным и может заменяться другими, также необязательными ключевыми словами. В табл. 4-4 описываются все ключевые слова, которые могут использоваться в операторе *select*. За дополнительной информацией о ключевых словах для работы с базой данных обратитесь к главе 17.

Табл. 4–4. Ключевые слова для оператора *select*

Ключевое слово	Описание
crossCompany	Заставляет среду времени выполнения Microsoft Dynamics AX генерировать запрос без добавления выражения *where* по полю *dataAreaId*. Это ключевое слово используется для выборки записей из всех или только из подмножества компаний системы. К примеру, запрос `while select crosscompany:companies myTable { }` выбирает все записи из таблицы *myTable* из компаний, указанных в контейнере *companies*
firstFast	Возвращает первую запись быстрее, чем оставшиеся записи из выборки
firstOnly *firstOnly1*	Возвращает только первую выбранную запись
firstOnly10	Возвращает только первые 10 выбранных записей
firstOnly100	Возвращает только первые 100 выбранных записей

Табл. 4–4. Ключевые слова для оператора *select (продолжение)*

Ключевое слово	Описание
firstOnly1000	Возвращает только первые 1000 выбранных записей
forceLiterals	Заставляет среду времени выполнения Microsoft Dynamics AX генерировать запрос с использованием конкретных значений (литералов). Например, запрос, сгенерированный для предыдущего примера кода, выглядит в этом случае так: *select * from myTable where myField1='value'*. Если указан данный параметр, то планы запросов к базе данных не являются повторно используемыми. Данное ключевое слово не может использоваться с ключевым словом *forceplaceholders*
forceNestedLoop	Заставляет процессор запросов SQL Server использовать алгоритм с вложенными циклами для операций объединения таблиц. Следовательно, другие алгоритмы объединения, такие как хэш-объединение или объединение слиянием, не рассматриваются
forcePlaceholders	Заставляет среду времени выполнения Microsoft Dynamics AX генерировать запрос с использованием переменных. Например, запрос, сгенерированный для предыдущего примера кода, выглядит в этом случае так: *select * from myTable where myField1=?*. Если указан данный параметр, то планы запросов к базе данных могут быть использованы повторно. Это параметр используется по умолчанию для операторов *select*, которые не объединяют несколько таблиц в запросе. Данное ключевое слово не может использоваться совместно с ключевым словом *forceliterals*
forceSelectOrder	Заставляет процессор запросов Microsoft SQL Server обращаться к таблицам в том порядке, в котором они указаны в запросе
forUpdate	Выбирает записи для обновления
generateOnly	Заставляет процессор запросов SQL Server только генерировать SQL-запрос, не выполняя его. Для генерации SQL-запроса можно также использовать метод *getSQLStatement* основной таблицы
noFetch	Указывает среде времени выполнения Microsoft Dynamics AX не выполнять запрос сразу, поскольку записи требуются только для какой-либо другой операции

Табл. 4–4. Ключевые слова для оператора *select (окончание)*

Ключевое слово	Описание
optimisticLock	Переопределяет свойство таблицы *OccEnabled*, навязывая использование оптимистической блокировки в запросах. Данное ключевое слово не может использоваться с ключевыми словами *pessimisticlock* и *repeatableread*
pessimisticLock	Переопределяет свойство таблицы *OccEnabled*, навязывая использование пессимистической блокировки в запросах. Данное ключевое слово не может использоваться с ключевыми словами *optimisticlock* и *repeatableread*
repeatableRead	Блокирует все записи, прочитанные в одной транзакции. С помощью этого ключевого слова гарантируется согласованность данных, выбираемых идентичными запросами в одной транзакции, ценой блокировки обновления этих записей другими процессами. Недействительные данные при этом все равно могут появиться в случае, если другой процесс вставляет записи, соответствующие критериям выборки. Данное ключевое слово не может использоваться с ключевыми словами *optimisticlock* и *pessimisticlock*
reverse	Возвращает записи в обратном порядке
validTimeState	Заставляет процессор запросов SQL Server использовать предоставленную дату или диапазон дат вместо текущей даты. Например, запрос *while select validTimeState(fromDate, toDate) myTable { }* выбирает все записи в таблице *myTable,* которые действительны в период между *fromDate и toDate*

В следующем примере показано, как используется оператор для указания индекса таблицы, который сервер базы данных должен использовать в запросе. Среда времени выполнения Microsoft Dynamics AX добавляет оператор *order by* и поля индекса к первому из приведенных запросов. Поэтому записи упорядочиваются в соответствии с указанным индексом. Среда времени выполнения Microsoft Dynamics AX вставляет дополнительную подсказку для сервера БД во второй из запросов в том случае, если это имеет смысл.

```
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;
}
```

В следующем примере кода демонстрируется, как результаты запроса могут сортироваться или группироваться. Первый оператор *select* указывает, что результирующие записи должны быть отсортированы в возрастающем порядке по значениям поля *myField1* и в убывающем порядке по значениям поля *myField2*. Второй оператор *select* указывает, что результирующие записи должны группироваться по полю *myField1* и сортироваться в убывающем порядке по значениям этого поля.

```
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;
}
```

Пример кода, приведенный ниже, демонстрирует использование агрегатных функций *avg* и *count* в операторах *select*. Первый оператор *select* рассчитывает среднее значение в столбце *myField* и присваивает результат

полю *myField*. Второй оператор *select* подсчитывает число возвращаемых записей и присваивает результат полю *myField*.

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

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

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

Внимание. Компилятор не проверяет, являются ли параметры агрегатных функций числовыми типами, поэтому результат функции может быть присвоен полю строкового типа. Компилятор также выполняет округление в том случае, если, к примеру, рассчитанное среднее значение равно 1.5, а результат присваивается полю *myField* целочисленного типа.

В табл. 4-5 описываются агрегатные функции, которые поддерживаются в операторах *select* языка X++

Табл. 4–5. Агрегатные функции X++-оператора *select*

Функция	Описание
avg	Возвращает среднее арифметическое значений поля результирующей выборки
count	Возвращает количество записей результирующей выборки
maxOf	Возвращает максимальное значение поля в результирующей выборке
minOf	Возвращает минимальное значение поля в результирующей выборке
sum	Возвращает сумму значений поля результирующей выборки

В следующем примере кода показывается, как с использованием ключевого слова *join* объединяются таблицы в запросе к базе данных. Первый оператор *select* объединяет две таблицы по условию равенства указанных

полей. Второй оператор *select* объединяет три таблицы, чтобы показать, что операции объединения таблиц могут быть вложенными, а также то, как можно использовать ключевое слово *exists* для проверки существования связанной записи при объединении. Второй оператор *select* также демонстрирует, как оператор *group by* используется в условиях объединения таблиц.

```
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;
}
```

В табл. 4-6 приведено описание оператора *exists* и других операторов *join*, которые могут использоваться вместо оператора *exist*.

Табл. 4–6. Операторы объединения

Оператор	Описание
exists	Возвращает *true*, если в результирующей выборке после объединения таблиц есть хоть одна запись. В противном случае возвращает *false*
notExists	Возвращает *false*, если в результирующей выборке после объединения таблиц есть хоть одна запись. В противном случае возвращает true
outer	Возвращает левостороннее внешнее объединение двух таблиц

В следующем примере демонстрируется использование оператора *while select*, который на каждой итерации цикла перемещает позицию курсора переменной *myTable* на следующую запись выборки.

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

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

Для обновления и вставки записей в таблицы должны использоваться операторы транзакций *ttsbegin*, *ttscommit* и *ttsabort*. Оператор *ttsbegin* указывает начало транзакции базы данных. Операторы *ttsbegin-ttscommit* могут быть вложенными. В этом случае каждый вызов оператора *ttsbegin* увеличивает уровень транзакции, каждый вызов оператора *ttscommit* уменьшает уровень транзакции, а самый внешний оператор *ttscommit* (уменьшает уровень транзакции до нуля, соответственно) фиксирует в базе данных все вставки и обновления записей, выполненные с момента начала транзакции при вызове первого оператора *ttsbegin*. Оператор *ttsabort* выполняет откат всех вставок, обновлений и удалений записей, выполненных после оператора *ttsbegin*. В табл. 4-7 приведены примеры использования данных операторов для одиночных записей и для большого количества записей с использованием пакетных операций. В последних трех примерах в табл. 4-7 показано использование метода *RowCount*, который был добавлен в Microsoft Dynamics AX 2009. Данный метод используется при выполнении пакетных операций *insert_recordset*, *update_recordset* и *delete_from* для получения количества обработанных записей.

Метод *RowCount* облегчает использование пакетных операций с базой данных в прикладных сценариях обновления данных. Приложение проверяет значение, возвращаемое методом *RowCount*, после выполнения выражения с *update_recordset* для определения количества обработанных записей. Без метода *RowCount* приложение, для того чтобы получить количество обработанных записей, выполнило бы дополнительный запрос к базе данных, что ухудшило бы производительность системы.

Табл. 4–7. Примеры выражений для работы с транзакциями базы данных

Тип выражения	Пример
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;

Табл. 4–7. Примеры выражений для работы с транзакциями базы данных (*окончание*)

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

Обработка исключений

Вместо программного отката транзакции с использованием оператора *ttsabort* рекомендуется использовать инфраструктуру обработки исключений X++. Исключение (кроме исключения конфликта обновления), генерируемое внутри транзакции, останавливает исполнение этого блока, после чего происходит откат всех вставок и обновлений записей, выполненных с момента вызова первого оператора *ttsbegin*.

Генерация исключения имеет дополнительное преимущество, которое заключается в том, что она позволяет восстановить состояния объекта и сохранить целостность данных. В теле блока *catch* допускается вызов оператора *retry*, что приведет к повторному выполнению блока *try*. В следующем примере демонстрируется генерация исключения внутри транзакции базы данных.

```
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;
    }
}
```

Оператор *throw* генерирует исключение, которое приводит к остановке транзакции и откату внесенных изменений. Исполнение кода не может продолжаться внутри области действия транзакции, поэтому среда времени выполнения игнорирует операторы *try* и *catch* внутри транзакции. Это означает, что исключение, сгенерированное внутри транзакции, может быть перехвачено только за пределами транзакции, как показано в следующем примере.

```
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;
```

```
    }
}
```

Несмотря на то, что оператор *throw* принимает в качестве параметра перечислимый тип *Exception*, для генерации ошибок лучше использовать глобальный метод *error*. Для каждого блока оператора *try* может быть указано более одного блока *catch*. Первый блок *catch* в первом примере этого раздела перехватывает исключения типа *Error*.

Оператор *retry* выполняет переход на первый оператор самого внешнего блока *try*. Второй блок *catch* перехватывает все типы исключений, которые не были перехвачены другими блоками *catch*. В табл. 4-8 описываются значения системного перечислимого типа данных *Exception*, которые могут использоваться в *try-catch*-выражениях.

Табл. 4–8. Значения перечислимого типа *Exception*

Значение	**Описание**
Break	Генерируется, когда пользователь нажимает клавишу Break или Ctrl+C
CLRError	Генерируется, если происходит неисправимая ошибка в коде, использующем общеязыковую среду выполнения (CLR)
CodeAccessSecurity	Генерируется, если происходит неисправимая ошибка в методе *demand* объекта *CodeAccessPermission*
DDEerror	Генерируется, если происходит ошибка в использовании системных классов динамического обмена данными (DDE)
Deadlock	Генерируется, когда транзакция базы данных попадает в тупиковую ситуацию
DuplicateKeyException	Генерируется, когда происходит нарушение уникальности первичного ключа таблицы при вставке записи. В блоке *catch* можно изменить значение первичного ключа и вызвать оператор *retry* для повторной попытки зафиксировать прерванную транзакцию
DuplicateKeyExceptionNotRecovered	Генерируется, когда происходит неисправимая ошибка. нарушения уникальности первичного ключа таблицы при вставке записи. Блок *catch* не должен использовать оператор *retry* для попытки зафиксировать прерванную транзакцию

Табл. 4–8. Значения перечислимого типа *Exception* (*окончание*)

Значение	Описание
*Error**	Генерируется, если происходит неисправимая ошибка приложения. Необходимо помнить, что в блоке *catch* уже произошел откат всех транзакций базы данных и, соответственно, внесенных изменений
Internal	Генерируется, если происходит неисправимая внутренняя ошибка приложения
Numeric	Генерируется, если происходит неисправимая ошибка в системных функциях *str2int*, *str2int64* и *str2num*
PassClrObjectAcrossTiers	*Генерируется, если происходит попытка передать объект* CLR с клиента на сервер или наоборот. Среда Microsoft Dynamics AX не поддерживает маршалинг объектов CLR между уровнями исполнения
Sequence	Генерируется ядром Microsoft Dynamics AX, если происходит ошибка базы данных при выполнении какой-либо операции
Timeout	*Генерируется, когда происходит тайм-аут с базой данных*
UpdateConflict	Генерируется, когда происходит конфликт обновления в транзакции, если используется режим оптимистичной конкуренции. В блок *catch* следует поместить вызов оператора *retry* для повторной попытки зафиксировать прерванную транзакцию
UpdateConflictNotRecovered	Генерируется, когда происходит неисправимая ошибка в транзакции, если используется режим оптимистичной конкуренции. Блок *catch* не должен использовать оператор *retry* для попытки зафиксировать прерванную транзакцию

* Метод *error* является статическим методом X++ класса *Global*, для вызова которого компилятор X++ разрешает использовать сокращенный синтаксис. Выражение *Global::error("Текст ошибки")* эквивалентно выражению *error* в примерах кода выше. Данные глобальные методы являются обычными методами X++, которые не следует путать с системными функциями, такими как *newguid*.

Типы исключения *UpdateConflict* и *DuplicateKeyException* являются единственными, которые приложение Microsoft Dynamics AX может перехватить в области действия транзакции базы данных. При генерации исключения *DuplicateKeyException* транзакция не откатывается, при этом позволяя восстановить состояние приложения. Существование исключения

DuplicateKeyException упрощает прикладные сценарии (такие как Сводное Планирование), которые выполняют ресурсоемкие операции по обработке большого объема записей в пакетном режиме и обрабатывают исключения уникальности первичного ключа без отката транзакции базы данных.

В следующем примере кода показано использование исключения *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;
}
```

В примере блок *catch* обрабатывает исключение уникальности первичного ключа. Обратите внимание, что уровень транзакции *ttsLevel* не изменяется и равен 1, что указывает на то, что откат транзакции не выполнялся и приложение может продолжить обработку других записей.

Примечание. Специальный синтаксис, в котором табличная сущность включалась в блок *catch*, больше не поддерживается.

Взаимодействие

Язык X++ содержит операторы для взаимодействия со сборками Microsoft .NET CLR и COM-компонентами. Возможность взаимодействия с обеими технологиями достигается путем создания в Microsoft Dynamics AX интерфейсных классов для внешних объектов и передачи вызовов методов объектов этих классов тем внешним объектам, интерфейс для которых они предоставляют.

Взаимодействие с CLR

Есть два способа написания кода для взаимодействия с CLR: со строгим и со слабым контролем типов. Рекомендуется использования строго-типизированного подхода, так как он менее подвержен ошибкам и в результате получается код, который намного читабельнее, чем в подходе со слабым контролем типов. Более того, среда разработки MorphX обеспечивает поддержку *IntelliSense* при наборе кода.

В примерах данного раздела предполагается, что сборка .NET System.Xml была добавлена в узел References AOT (подробнее описание программной модели элементов см. в главе 1 «Обзор архитектуры»). Код для работы с CLR, написанный с использованием такого подхода, получается более громоздким, поскольку компилятор поддерживает вызов не более одного метода в выражении, а обращение к CLR-типу происходит с использованием полного имени типа в указанной сборке. Например, выражение *System.Xml.XmlDocument* является полным именем для типа XML-документа в Microsoft .NET Framework.

Внимание. X++ при работе с CLR-типами является регистро-зависимым языком!

В следующем примере показано взаимодействие со строгим контролем типов CLR, неявным преобразованием строковых типов Microsoft Dynamics AX в строки общеязыковой среды выполнения и обработкой CLR-исключений в 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 );
        }
    }
}
```

В следующем блоке кода показан пример вызова статических методов CLR из X++ с использованием синтаксиса (::) вызова статических методов X++.

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

Microsoft Dynamics AX 2012 также обеспечивает поддержку массивов CLR:

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

    myArray.SetValue(1000, 0);
```

```
    print myArray.GetValue(0);
}
```

Язык X++ поддерживает передачу параметров по ссылке в методы CLR. Изменения, которые происходят с передаваемым параметром в вызываемом методе, влияют и на значение соответствующей переменной в вызывающем методе. При передаче по ссылке значимых типов они временно оборачиваются в объект. Неявное выполнение этой операции, которую обычно называют «упаковкой», показано в коде ниже.

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

    MyNamespace.MyMath::Increment(byref myVar);

    print myVar;   // prints 6
}
```

Вызываемый метод на языке C# имеет следующую реализацию:

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

Примечание. Передача параметров по ссылке поддерживается только для CLR-методов и не поддерживается для X++-методов.

Вторым подходом при написании кода взаимодействия с CLR является подход со слабым контролем типов. В следующем примере выполняются операции, аналогичные первому примеру раздела, однако в этом случае все ссылки проверяются только во время выполнения и используются только явные преобразования типов.

```
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;
}
```

Первый оператор в предыдущем примере демонстрирует использование статического метода преобразования между примитивными типами Microsoft Dynamics AX и CLR-объектами. Оператор *print* показывает обратное преобразование значимых типов CLR к примитивным типам X++. В табл. 4-9 перечислены поддерживаемые Microsoft Dynamics AX преобразования типов.

Табл. 4–9. Преобразования типов, поддержка которых присутствует в Microsoft Dynamics AX

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

В предыдущем примере кода также демонстрируется синтаксис метода доступа к свойствам CLR-объекта, например *get_DocumentElement*. CLR поддерживает несколько операторов, которые не поддерживаются в X++.

В табл. 4-10 перечислены CLR-операторы и синтаксис альтернативных методов в X++.

Табл. 4–10. CLR-операторы и методы их использования

Операторы CLR	**Методы CLR**
Операторы обращения к свойствам	*get_<property>, set_<property>*
Операторы обращения по индексу	*get_Item, set_Item*
Математические операторы	*op_<operation>(arguments)*

Следующие возможности CLR не могут использоваться в X++.

- Открытые (Public) поля (доступ можно получить через классы отражения CLR).
- События и делегаты.
- Параметризованные типы (generics).
- Внутренние классы.
- Объявления пространств имен.

Взаимодействие с COM

В следующем примере кода демонстрируется взаимодействие с XML-документом путем использования COM-компонента Microsoft XML Core Services (MSXML) 6.0. Пример предполагает, что COM-компонент MSXML уже установлен. Первым шагом является создание объекта MSXML-документа, упакованного в оболочку COM-объектов Microsoft Dynamics AX. Затем для COM-строки создается оболочка вариантных типов COM с направлением передачи строки в COM-компонент. Переменные для корневого элемента и заголовка XML-файла объявляются как COM-объекты. В примере показывается, как заполнить переменную вариантного типа строкой X++, а затем использовать данную переменную в качестве аргумента в COM-методе *loadXml*.

Оператор, который создает элемент заголовка документа, демонстрирует, как среда времени выполнения Microsoft Dynamics AX автоматически преобразовывает примитивные типы Microsoft Dynamics AX в COM-объекты вариантного типа.

```
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;
}
```

Макросы

С помощью макросов в X++ можно определять и использовать константы макроподстановки, а также выполнять условную компиляцию кода. Макросы задаются без учета синтаксиса X++, и обрабатываются еще до компиляции исходного кода приложения. Макросы могут использоваться как в методах, так и в определении классов.

В табл. 4-11 показаны доступные для макросов директивы.

Табл. 4–11. Директивы макросов

Директива	Описание
#define *#globaldefine*	Определяет макроподстановку с указанным значением. `#define.MyMacro(SomeValue)` Определяет макрос *MyMacro*, который будет заменяться значением *SomeValue*

#macro
…
#endmacro
 	
#localmacro
…
#endmacro
	Определяет макроподстановку кода, которая занимает несколько строк.
	`#macro.MyMacro`
	`  print “foo”;`
	`  print “bar”;`
	`#endmacro`
	Определяет макрос MyMacro со значением для подстановки, которое занимает несколько строк

Табл. 4–11. Директивы макросов (*продолжение*)

Директива	Описание

#macrolib	Подключает указанную макробиблиотеку. В сокращенной записи этой директивы можно упускать ключевое слово *macrolib*.
	`#macrolib.MyMacroLibrary`
	`#MyMacroLibrary`
	Оба примера подключают макробиблиотеку *MyMacroLibrary*, которая должна быть определена в узле Macros в AOT

#MyMacro	Производит подстановку значения макроса.
	`#define.MyMacro("Hello World")`
	`print #MyMacro;`
	Определяет макроподстановку *MyMacro*, и выводит на экран ее значение. В приведенном примере, на экран будет выведена строка "Hello World"

#definc *#defdec*	Увеличивает или уменьшает значение макроса. Обычно используется для значений целочисленного типа. `#defdec.MyIntMacro` Уменьшает значение макроподстановки *MyIntMacro*
#undef	Удаляет определение макроподстановки. `#undef.MyMacro` Удаляет определение макроподстановки *MyMacro*

#if
…
#endif
	Условная компиляция кода. Если макрос, указанный после директивы *#if* определен или содержит указанное значение, то будет выполнена компиляция следующего за директивой кода.
	`#if.MyMacro`
	`print "MyMacro is defined";`
	`#endif`
	Если макрос *MyMacro* определен, то выражение с оператором *print* будет считаться частью исходного кода при компиляции.
	`#if.MyMacro(SomeValue)`
	`print "MyMacro is defined and has value: SomeValue";`
	`#endif`
	Если макрос *MyMacro* содержит значение *SomeValue*, то выражение с оператором *print* будет считаться частью исходного кода при компиляции

Табл. 4–11. Директивы макросов (*окончание*)

Директива	Описание

#ifnot
…
#endif
	Условная компиляция. Если макрос, указанный после директивы *#ifnot* не определен или не содержит указанное значение, то будет выполнена компиляция следующего за директивой кода.
	`#ifnot.MyMacro`
	`print "MyMacro is not defined";`
	`#endif`
	Если макрос *MyMacro* не определен, то выражение с оператором *print* будет считаться частью исходного кода при компиляции.
	`#ifnot.MyMacro(SomeValue)`
	`print "MyMacro does not have value: SomeValue; or it is not defined";`
	`#endif`
	Если макрос *MyMacro* не определен или не содержит значение *SomeValue*, выражение с оператором *print* будет считаться частью исходного кода при компиляции

В следующем примере кода показано определение макроподстановки и ее последующее использование.

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

   print #HelloWorld;
   pause;
}
```

Как уже упоминалось в табл. 4-11, глобальная макробиблиотека создается с помощью AOT в узле Macros и используется путем включения в заголовок (*classDeclaration*) или метод класса, как показано в следующем примере.

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

```
    #MyMacroLibrary2


    #MacroFromMyMacroLibrary1
    #MacroFromMyMacroLibrary2
}
```

Макроподстановки могут принимать на вход параметры. При этом компилятор вставляет параметры вместо заполнителей, указанных в определении макроса. В следующем примере кода показано использование макроподстановки с параметрами.

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

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

Комментарии

В языке X++ используются однострочные и многострочные комментарии. Однострочные комментарии начинаются с символов // и заканчиваются символом конца строки. Многострочные комментарии начинаются с символов /* и заканчиваются символами */. Вложенные многострочные комментарии не допускаются. В комментарии можно помещать напоминания, которые обрабатываются компилятором и выводятся в качестве заданий в окне сообщений компилятора. Для этого нужно добавить однострочный комментарий, содержащий слово TODO (все буквы должны быть заглавными). Обратите внимание, что задания, указываемые внутри многострочных комментариев, рассматриваются как закомментированные, и поэтому компилятором не обрабатываются.

Ниже приводится пример использования комментариев, напоминающих разработчику о том, что ему необходимо сделать, а также многострочных комментариев, с помощью которых отключена существующая функциональность.

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

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

XML-документация

Microsoft Dynamics AX поддерживает написание документации к коду методов и классов непосредственно в коде X++. Документация, оформляемая в структурированном XML-формате, добавляется в начало кода метода, который она описывает, путем написания символов /// (тройная косая черта), за которыми следует описание метода.

Структура документации должна строго соответствовать формату и тому коду, который она описывает. Инструмент рекомендаций Best Practices содержит набор правил, с помощью которых проверяется корректность XML-документации. Допустимые теги приведены в табл. 4-12.

Табл. 4–12. Теги, допустимые в тексте документации XML

Тег	Описание
<*summary*>	Описывает назначение метода или класса
<*param*>	Описывает параметры метод
<*returns*>	Описывает возвращаемое методом значение
<*remarks*>	Содержит дополнительную информацию о методе/классе, которая расширяет описание, приведенное в теге <*summary*>
<*exception*>	Описывает исключения, которые могут генерироваться в методе
<*permission*>	Описывает права доступа, необходимые для работы метода *CodeAccessSecurity.demand*
<*seealso*>	Перечисляет ссылки на логически связанную документацию

Документация XML автоматически показывается функцией *IntelliSense* в редакторе X++.

Документацию XML можно экспортировать для конкретного проекта в AOT, если выбрать в меню Надстройки пункт Извлечь документацию XML. В результате будет сгенерирован один XML-файл, содержащий описание всех элементов выбранного проекта. Этот файл может использоваться для публикации документации проекта.

Ниже приведен пример кода статического метода в классе *Global*, для которого написана документация XML.

```
/// <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);
}
```

Классы и интерфейсы

Все определения типов и их структуры производятся в AOT, а не в коде X++, как это делается в других языках программирования, которые поддерживают объявления типов. Это обусловлено тем, что Microsoft Dynamics AX поддерживает технологию слоев прикладных объектов, поддерживающую изменение исходного кода приложения и типов данных, которые используются в объявлениях переменных и методов. Каждая часть объявления типа обрабатывается как независимая единица компиляции, а данные прикладной модели используются для управления, поддержки и воссоздания динамических типов, которые могут состоять из единиц компиляции из нескольких прикладных слоев.

Язык X++ используется для определения логики приложения, в том числе для определения методов (возвращаемое значение, название метода, названия и типы параметров). Редактор X++ позволяет добавлять новые методы в AOT, поэтому при конструировании прикладных типов вы можете продолжать использовать только редактор.

Заголовки X++-классов используются для объявления защищенных (*protected*) переменных, которые являются членами прикладной логики и

типов ссылочных структур. Вы не можете объявлять закрытые (*private*) или открытые (*public*) переменные классов. Классы могут объявляться абстрактными (*abstract*), если они являются неполными спецификациями типов, объекты которых не могут быть созданы. Классы также могут быть объявлены как завершенные спецификации (*final*), если не планируется дальнейшее их расширение. В следующем примере кода приведен пример объявления абстрактного класса.

```
abstract class MyClass
{
}
```

Классы также могут быть сгруппированы в иерархии единичного наследования, в которой производные классы наследуют и перекрывают члены базового класса. Ниже представлен пример объявления производного класса, который указывает, что класс *MyDerivedClass* является подклассом абстрактного базового класса *MyClass*. В нем также указывается, что класс *MyDerivedClass* является завершенной спецификацией (*final*), то есть дальнейшее расширение класса не допускается. Производный класс может быть подклассом только одного базового класса, поскольку X++ не поддерживает множественное наследование.

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

Язык X++ также поддерживает определения интерфейсов, которые указывают сигнатуры методов, но не содержат их реализации. Классы могут реализовывать более чем один интерфейс, но класс и его производные классы должны предоставить реализацию для методов, объявленных во всех реализуемых ими интерфейсах. Если класс не предоставляет реализации методов, то он должен быть объявлен абстрактным. В следующем коде представлен пример объявления интерфейса, а также объявления класса, который реализует данный интерфейс.

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

```
{
    void myMethod()
    {
    }
}
```

Поля

Поле – это член класса, который представляет переменную и ее тип. Поля объявляются в заголовках классов (метод *classDeclaration*, присутствующий в определении любого класса или интерфейса в AOT), как показано в следующем примере кода. Поля классов доступны только из методов самого класса, а также из методов его наследников. В заголовках классов операторы присваивания не допускаются. В следующем примере кода продемонстрировано, как переменные инициализируются с помощью операторов присваивания в методе *new* класса.

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

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

Методы

Методы – это члены класса, которые определяют поведение объекта с помощью операторов X++. Метод интерфейса – это член, который декларирует ожидаемое поведение объекта. В следующем коде приведен пример объявления метода интерфейса и его реализации в классе, который реализует данный интерфейс.

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

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

Методы определяются с указанием модификаторов доступа: *public* (открытый), *private* (закрытый) или *protected* (защищенный). По умолчанию методы определяются с модификатором доступа *public*. Дополнительные модификаторы методов, поддерживаемые X++, представлены в табл. 4-13.

Табл. 4–13. Модификаторы методов, поддерживаемые в X++

Модификатор	**Описание**
abstract	Абстрактные методы не имеют реализации. Производные классы должны предоставить реализацию для абстрактных методов
client	Клиентские методы могут исполняться только на клиенте Microsoft Dynamics AX. Модификаторы *client* разрешается использовать только для статических методов
delegate	Методы с модификатором *delegate* не могут содержать реализацию. Обработчики событий можно подписывать на методы с модификатором *methods*. Модификатор *delegate* допускается использовать только в методах объектов
display	Дисплей-методы вызываются каждый раз при перерисовке формы или отчета. Модификатор *display* разрешается использовать только для методов таблиц, форм, источников данных форм, отчетов и дизайна отчетов
edit	*edit*-методы вызываются каждый раз при перерисовке формы или при вводе данных пользователем в управляющие элементы формы. Модификатор *edit* разрешается использовать только для методов таблиц, форм и источников данных форм
final	Конечные методы не могут быть перекрыты в производных классах
server	Серверные методы могут исполняться только на сервере приложения AOS. Модификатор *server* разрешается использовать только для статических методов
static	Доступ к статическим методам выполняется посредством определения класса. Доступ к полям класса из статических методов невозможен

В методах могут использоваться параметры со значениями по умолчанию, которые используются, когда при вызове метода данные параметры опускаются. В следующем примере кода строка "Hello World" печатается в случае вызова метода *myMethod* без параметров.

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

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

Конструктор – это специальный объектный метод, который вызывается средой времени выполнения Microsoft Dynamics AX для инициализации объекта при исполнении оператора *new*. Конструкторы не могут напрямую вызываться из кода X++. Ниже представлен пример объявления класса и его конструктора, который принимает в качестве аргумента один параметр.

```
class MyClass
{
    int i;

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

Делегаты

Цель использования делегатов – вставка точек расширения функциональности системы, при которых надстройки и дописанная бизнес-логика проще применяются без модифицирования основного кода базовой функциональности. Делегаты – это методы без реализации. Делегаты – это всегда открытые методы, не возвращающие значения. Метод делегат объявляется в помощью ключевого слова *delegate*. Метод делегат вызывается так же, как и стандартный метод.

```
class MyClass
```

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

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

При вызове делегата среда исполнения автоматически вызывает все обработчики событий, подписанные на этот делегат. Существуют два способа подписки на делегаты: декларативно и динамически. Среда исполнения не проверяет последовательность вызова обработчиков событий. Если ваша логика опирается на определенную последовательность вызовов, то используйте другой механизм вместо делегатов и обработчиков событий. Чтобы создать подписку на событие в контекстном меню узла метода, выберите пункт Создать Event Handler Subscription. В созданном обработчике события репозитария дерева AOT определяете класс и статический метод, который будет вызываться при обработке события. Класс, обработчик события может быть реализован в X++ или в .NET.

Для динамической подписки на событие используйте *eventhandler*. Обратите внимание, что при динамической подписке обработчик события является методом. Также возможно динамически отписаться от обработки события.

```
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);
    }
}
```

Независимо от подписки, обработчик события должен содержать префикс *public*, ничего не возвращать (*void*) и иметь тот же набор параметров, что и делегат.

Примечание. Передача между сервером и клиентом не поддерживается.

В качестве альтернативы делегатам вы можете получить аналогичный результат, используя предварительные и завершающие обработчики событий.

Предварительные и завершающие обработчики событий

Вы можете декларативно подписаться на события любого класса и любого метода таблицы с помощью аналогичной процедуры, использующей делегат. Обработчик события вызывается до или после вызова метода. Обработчики событий должны быть открытыми, статическими, ничего не возвращающими, а также содержать тот же набор параметров, что и основной метод или одни параметр типа *XppPrePostArgs*.

Упрощенная и безопасная по типу реализация использует синтаксис, в котором параметры метода и обработчика событий совпадают.

```
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)
        {
            ...
        }
    }
}
```

Если вам нужно управлять параметрами или возвращаемым значением, обработчик события должен принимать один параметр типа *XppPrePostArgs*.

Для создания обработчика событий в контекстном меню метода класса выберите Создать > Обработчик перед или после выполнения события. Класс *XppPrePostArgs* предоставляет доступ к параметрам и возвращаемому значению метода. Вы можете изменить значение параметра в обработчике *pre-event handlers* и изменять значение в *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);
        }
    }
}
```

Атрибуты

Классы и методы могут содержать атрибуты для передачи декларативной информации в другой код, который отрабатывается во время исполнения, компилятор, другие структуры или другие инструменты. Для этого необходимо вставить атрибут в метод *classDeclaration*. Для передачи декларативной информации в метод вставьте атрибут перед объявление метода.

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

Первый созданный атрибут в Microsoft Dynamics AX 2012 это – *SysObsoleteAttribute*. При обрамлении класса или метода этим атрибутом любой код-потребитель уведомляется во время компиляции, что цель устарела (*obsolete*). Вы можете создавать свои атрибуты, унаследовавшись от класса *SysAttribute*:

```
class MyAttribute extends SysAttribute
{
    str parameter;
```

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

Разграничение доступа кода

Разграничение доступа кода (Code access security, CAS) – это механизм, разработанный для защиты систем от опасных интерфейсов программирования, которые вызываются непроверенным кодом. CAS не имеет ничего общего с авторизации и аутентификацией пользователей; это механизм, позволяющий защищенно вызывать код из другого кода.

Внимание. Разработчики X++ ответственны за написание кода, следующего рекомендациям защищенных информационных систем (Trustworthy Computing). Данные рекомендации изложены в техническом документе «Writing Secure X++ Code», доступном для загрузки с сайта Microsoft Dynamics AX (*http://msdn.microsoft.com/en-us/dynamics/ax*).

В реализации разграничения доступа кода, доступного в Microsoft Dynamics AX, проверенным кодом считается тот, который присутствует в AOT и исполняется на сервере приложения AOS. Присутствие кода в AOT означает, что он был написан разработчиком X++, который имеет наивысшие полномочия в системе. Такие полномочия должны даваться только проверенному персоналу. Выполнение кода на сервере приложения гарантирует, что он не был модифицирован перед исполнением.

При выполнении кода вне сервера приложения – к примеру, на толстом клиенте – нет гарантии, что этот код не модифицировался перед исполнением, а соответственно такому коду верить нельзя. К непроверенному коду относится также и код, который выполняется через функции *runBuf* и *evalBuf*. Эти методы обычно используются для выполнения кода, сгенерированного во время выполнения на основании ввода пользователя.

CAS предоставляет возможность безопасного взаимодействия прикладного интерфейса и его потребителя с помощью подтверждения безопасности вызова. Только подтвержденные потребители имеют возможность вызова прикладного интерфейса. Любой другой вызов генерирует исключение.

Подтверждение безопасности выполняется с помощью класса *CodeAccessPermission* или одного из его наследников. Потребитель API должен запросить разрешение для вызова прикладного интерфейса путем вызова метод *CodeAccessPermission.assert*. Прикладной интерфейс в свою очередь проверяет полномочия потребителя с помощью вызова метода *CodeAccessPermission.demand*. Метод *demand* просматривает стек вызовов до соответствующего вызова метода assert в поиске непроверенного кода, исполняемого между двумя вызовами. Если такой код найден, генерируется исключение. Данный процесс показан на рис. 4-1.

```
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);
   }
}
```

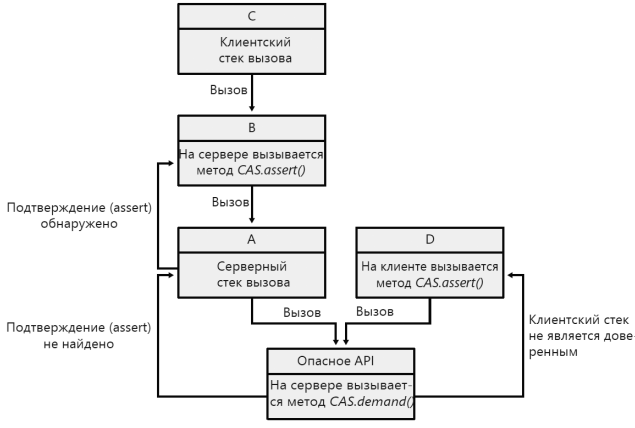

Рис. 4–1. Структура стека CAS

WinAPIServer::deleteFile считается опасным прикладным интерфейсом, поскольку он предоставляет доступ к API *.NET System.IO.File::Delete(string fileName)*. **Это опасно, поскольку позволяет пользователю интерфейса удалить** любой файл на сервере приложения удаленно, что может нарушить работу сервера. В данном примере *WinApiServer::deleteFile* требует, чтобы вызывающий код проверил правильность имени файла, который будет удален. Такое требование предотвращает использование этого интерфейса с клиента приложения и из любого кода, которые не присутствует в AOT.

Внимание. При использовании метода *assert*, разработчик должен убедиться, что его код не является таким же опасным интерфейсом API, как и тот код, который защищен CAS. При вызове *assert* разработчик гарантирует, что его код не позволяет воспользоваться той же уязвимостью, что и защищенный прикладной интерфейс. Так, если бы в предыдущем примере метод *deleteTmpFile* принимал название файла в качестве параметра, то он мог бы использоваться злоумышленником для обхода разграничения доступа кода для метода *WinApi::deleteFile* и удаления любого файла на сервере приложения.

Компиляция и исполнение кода X++ как .NET CIL

Весь код X++ компилируется в промежуточный код, формат среды исполнения Microsoft Dynamics AX. Это формат используется средой Microsoft Dynamics AX как серверный и клиентский код Microsoft Dynamics.

Кроме того, классы и таблицы также компилируются в .NET common intermediate language (CIL). Это формат используется кодом X++, выполняемым в пакетной обработке и в некоторых других сценариях. Компилятор X++ только лишь производит исполняемый байт-код Microsoft Dynamics AX для генерации CIL-кода; вы можете в ручную выбрать опцию Создать полный IL или Создать инкрементный IL. Оба варианта доступны на панели инструментов.

Основным преимуществом исполнения X++ как CIL является улучшенная производительность. Вообще исполнение .NET значительно быстрее, чем X++. В некоторых конструкциях прирост производительности особенно значителен.

- **Конструкции с множественными вызовами методов.** Внутри среды исполнения X++ вызов любого метода происходит через отражение, в то время как в CIL это происходит на уровне процессора.
- **Конструкции с использование объектов с коротким временем жизни.** Работа сборщика мусора в среде исполнения Microsoft Dynamics AX всегда предопределена. Это означает, что, когда объект выходит за рамки использования, анализируется весь граф объектов, чтобы определить, какие объекты могут быть освобождены из памяти. В среде .NET CLR работа сборщика мусора не предопределена. Это означает, что сама среда определяет оптимальное время жизни объекта в памяти.
- **Конструкции с использованием .NET взаимодействия.** Когда код X++ выполняется как CIL, все преобразования и маршалинг между средами отсутствуют.

Примечание. Компиляции X++ в CIL требует, чтобы синтаксис в X++ был достаточно строгим, как и в управляемом коде. Значительным изменением является то, что переопределенный метод должен иметь ту же подпись, что и базовый. Единственное допустимое расхождение – добавление новых параметров.

Пример из реальной жизни: когда используется код X++ и .NET CIL, то происходит значительное различие в производительности при сравнении объекта в инструменте сравнения. Алгоритм сравнения реализован на X++ в классе *SysCompareText*. Хотя сам алгоритм имеет всего несколько методов, несколько объектов с коротким временем жизни и отсутствует .NET-взаимодействие, переключение в CIL означает, что в течении 10 секунд возможно сравнить два текста длиной в 3500 строчек, в то время как среда AX может обработать за тот же период времени только 600. Сложность алгоритма экспоненциальная. Другими словами, разница в производительности в значительной степени зависит от объема обрабатываемого текста.

Все сервисы и пакетные обработки автоматически запускаются в CIL. Если вы хотите принудительно использовать X++ вместо CIL в сценариях, не использующих пакетные операции, то применяйте методы *runClassMethodIL* и *runTableMethodIL* класса *Global*. Точка входа для IL должна быть статическим серверным методом, который возвращает контейнер и принимает один параметр-контейнер.

```
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));
    }
}
```

Шаблоны проектирования

До сих пор в этой главе обсуждались отдельные элементы языка X++. Было показано, как выражения X++ группируются в методы, а методы – в классы, таблицы и другие прикладные элементы. Эти структуры позволяют разработчику X++ анализировать код на более высоком уровне абстракции. В следующем примере кода показано, как операция присваивания может быть инкапсулирована в метод для повышения уровня абстракции.

```
control.show();
is at a higher level of abstraction than

flags = flags | 0x0004;
```

Шаблоны проектирования позволяют использовать существующие решения известных проблем, а также упрощают понимание предназначения конкретного блока кода. Не стоит забывать, что разработчики кода обычно тратят большую часть времени именно на чтение кода, а не на его написание.

Реализации шаблонов проектирования обычно можно опознать по названиям классов, методов, параметров и переменных, используемых в данной реализации. Выбор названий для этих элементов с целью четкой передачи замысла разработчика – это, возможно, самая сложная его задача. Большинство литературы, в которой освещен вопрос шаблонов проектирования, посвящено объектно-ориентированным языкам программирования, а соответственно вы можете получить выгоду от прочтения данной литературы в поисках шаблонов и приемов, которые потом можно будет использовать при написании кода X++. Шаблоны проектирования раскрывают отношения и взаимодействие классов и объектов. Они не навязывают конкретной реализации, однако предлагают шаблонное решение для типичных задач проектирования.

Шаблоны реализации обычно завязаны на конкретную реализацию и затрагивают единичное выражение.

В данном разделе описываются наиболее часто используемые в X++ шаблоны проектирования. Более подробное описание доступно в руководстве разработчика Microsoft Dynamics AX SDK на сайте MSDN.

Шаблоны на уровне класса

Шаблоны, описанные ниже, применимы к классам в X++.

Методы доступа к полям класса

Для получения и установки значения поля класса извне необходимо реализовать метод доступа. Метод должен называться аналогично полю, доступ к которому он предоставляет, с добавлением префикса *parm*. Существуют два типа методов доступа: только для получения значения поля класса и как для установки, так и для получения.

```
public class Employee
{
    EmployeeName name;

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

Инкапсуляция конструктора

Целью шаблона инкапсуляции конструктора является выполнение принципа подстановки Барбары Лисков. Другими словами, инкапсуляция конструктора позволяет разработчику заменить существующий класс его наследником без использования технологии прикладных слоев. Как и в системе прикладных слоев, этот шаблон проектирования позволяет изменить логику работы класса без необходимости обновления всех ссылок на этот класс. Перекрытия определения прикладных элементов на текущем слое стоит по возможности избегать, так как это в дальнейшем может привести к конфликтам обновления кода.

Классы, содержащие статический метод *construct*, используют шаблон инкапсуляции конструктора. Метод *construct* должен создать объект класса и вернуть созданный объект в вызывающий метод. Метод *construct* должен быть статическим и не должен принимать параметров.

Если для инициализации класса нужны параметры, следует создавать для этого статические методы *new*. Данные методы должны вызывать метод *construct* для инициализации класса, а затем с помощью методов доступа устанавливать значения требуемых параметров класса. В данном случае метод *construct* должен иметь модификатор доступа *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;
    }
}
```

Фабрика

Для отделения базовых классов от производных используйте структуру *SysExtension*. Эта структура позволяет создавать экземпляры базового класса на основе его атрибутов. Этот шаблон допускает использование дополнений и настроек в наследуемых классах без упоминания базового класса или фабричного метода.

```
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
{
    ...
}
```

Сериализация с помощью методов *pack* и *unpack*

Многим классам необходима возможность сериализации и десериализации. Сериализация – это операция перевода состояния объекта в формат данных значимого типа; десериализация восстанавливает состояние объекта на основе этих данных.

В X++ классы, которые реализуют интерфейс Packable, поддерживают сериализацию. Интерфейс Packable содержит два метода: *pack* и *unpack*. Метод *pack* возвращает контейнер, содержащий текущее состояние объекта; метод *unpack*, соответственно, восстанавливает состояние объекта на основании переданного в качестве параметра контейнера. В создаваемом контейнере следует сохранять также и текущую версию объекта (на первой позиции в контейнере), что позволит сохранить поддержку устаревших данных о состоянии объекта при модификации определения этого класса.

```
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;
    }
}
```

Шаблоны на уровне таблицы

Шаблоны, описанные в данном разделе – методы *Find* и *Exists*, полиморфные отношения (Таблица/Группа/Все) и общие отношения таблиц, – применимы ко всем таблицам в AOT.

Методы *find* и *exists*

Каждая таблица должна содержать два статических метода *find* и *exists*. Оба принимают на вход значения полей первичного ключа и возвращают запись таблицы, соответствующую этим значениям, и булевое значение типа Boolean, соответственно.

Помимо значений полей первичного ключа, метод *find* содержит дополнительный параметр типа Boolean, указывающий на необходимость выбора записи на обновление. На примере таблицы *CustTable* показана сигнатура этих двух методов.

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

Полиморфные отношения

Шаблон Таблица/Группа/Все используется для моделирования полиморфной связи с конкретной записью, набором записей или всеми записями в другой таблице. К примеру, выбранная запись может быть ассоциирована с конкретной номенклатурой, с группой номенклатур или со всеми номенклатурами.

Реализация шаблона Таблица/Группа/Все выполняется путем создания двух полей и двух отношений на таблице. Рекомендуется добавить к названию первого поля суффикс *Code*, как в названии *ItemCode*. Это поле должно использовать перечислимый тип *TableGroupAll*. К названию второго поля обычно добавляют суффикс *Relation*, к примеру, *ItemRelation*. Это поле должно использовать расширенный тип данных первичного ключа связанной таблицы. Оба отношения, создаваемые на таблице, используют тип отношения Поле фиксировано. Первое отношение определяет, что для значения поля *Code*, равного 0 (*TableGroupAll::Table*), поле *Relation* указывает на первичный ключ в главной связанной таблице. Второе отношение определяет, что для значения поля *Code*, равного 1 (*TableGroupAll::Group*), поле *Relation* указывает на первичный ключ в групповой связанной таблице.

На рис. 4-2 показан пример определения этих отношений.

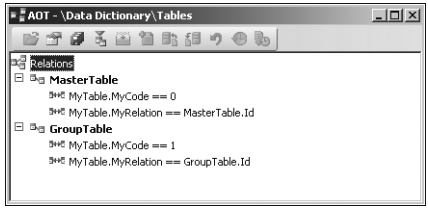

Рис. 4–2. Полиморфные ассоциации

Общие отношения таблиц

Шаблон общих отношений таблиц является вариацией шаблона полиморфных отношений. Он используется для моделирования связей с внешними таблицами.

Существуют три разновидности применения шаблона: (а) связь с любой записью в одной выбранной связанной таблице, (б) связь с любой записью в ограниченном наборе связанных таблиц и (в) связь с любой записью в любой таблице.

Все три разновидности реализуются путем создания поля на основе расширенного типа данных *RefRecId*. Для моделирования связи с любой записью в одной конкретной таблице (разновидность *а*) на таблице создается отношение по полю типа *RefRecId* на поле *RecId* связанной таблицы, как показано на рис. 4-3.

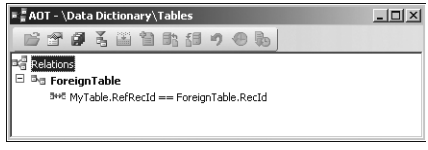

Рис. 4–3. Пример отношения к определенной таблице

Для реализации разновидностей *б* и *в* требуется создание дополнительного поля. Это поле должно быть основано на расширенном типе данных *RefTableId*. Для моделирования связи с любой записью в фиксированном наборе таблиц (разновидность *б*) на таблице создается новое отношение для каждой таблицы из выбранного набора: по полю типа *RefTableId* на поле *TableId* связанной таблицы и по полю типа *RefRecId* на поле *RecId* связанной таблицы, как показано на рис. 4-4.

Рис. 4–4. Пример отношения к любой записи из выбранного набора таблиц

Для моделирования связи с любой записью в любой таблице (разновидность *в*) на таблице создается отношение по полю типа *RefTableId* на поле *TableId* базового типа для всех таблиц, *Common* и по полю типа *RefRecId* на поле *RecId* таблицы *Common*, как показано на рис. 4-5.

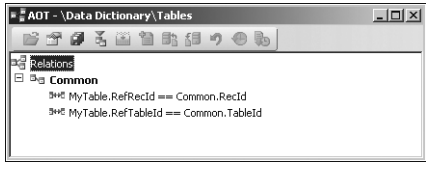

Рис. 4–5. Связь с любой таблицей