Глава 20

Отражение

В этой главе

- Введение
- Системные функции отражения
- Прикладные интерфейсы отражения

Введение

Отражение (reflection) – это программный механизм раскрытия прикладной модели. С помощью интерфейсов отражения (APIs) системы Microsoft Dynamics AX можно в среде разработки запрашивать метаданные так, как если бы они хранились в таблице, объектной модели или древовидной структуре.

С помощью сведений, предоставленных механизмом отражения, можно делать много полезного. Прекрасным примером мощных функциональных возможностей отражения служит инструмент Обратная разработка. На основе определений элементов в MorphX данный инструмент генерирует UML модели и схемы сущностей и отношений (ERDs), которые можно просматривать в приложении Microsoft Office Visio. Отражение также позволяет вызывать методы объектов. Это имеет небольшое значение для разработчиков бизнес-приложений, которые создают иерархии классов, однако для разработчиков инфраструктур такая возможность крайне необходима.

Предположим, что нужно программным путем записать в XML-файл содержимое выбранной записи, включая все поля и display-методы. Отражение позволяет определить перечень полей и их значения, а также вызвать display-методы, чтобы получить возвращаемые ими значения.

Язык X++ предлагает набор системных функций, которые могут использоваться для отражения прикладной модели, а также три интерфейса программирования для этих же целей. Системные функции отражения приведены ниже.

- **Встроенные функции.** Набор функций, которые позволяют вам ссылаться на название или идентификатор прикладного элемента с проверкой существования элемента.
- ***Системная функция typeOf.*** Функция, которая возвращает базовый тип указанной переменной.
- ***Системная функция classIdGet.*** Функция, которая возвращает идентификатор класса на основании экземпляра объекта.

Интерфейсы программирования, реализующие отражение, включают в себя следующие элементы.

- **Представление модели в виде таблиц.** Набор таблиц, которые содержат определения всех прикладных элементов приложения. Таблицы предоставляют вам прямой доступ к содержимому .aod-файлов. Таким образом, можно проверить существование элементов и определенных свойств прикладных элементов, например значения свойств *created by* и *created datetime*. С помощью этого прикладного интерфейса нельзя извлечь сведения о содержимом или структуре каждого элемента.
- **Словарь данных (Dictionary).** Набор классов, которые предоставляют строго типизированный механизм для чтения метаданных из прикладной модели. Классы словаря данных предоставляют базовые и более абстрактные сведения об элементах с использованием строгой типизации. За несколькими исключениями данный API может использоваться только для чтения данных.
- **Узлы AOT (TreeNode).** Иерархия классов, которая предоставляет доступ к узлам репозитария прикладных объектов с возможностью создания, чтения, обновления и удаления любого участка метаданных и исходного кода. Этот программный интерфейс позволяет получить полную информацию о любом прикладном элементе в AOT. С помощью данного API вы перебираете узлы в AOT и запрашиваете метаданные, при этом не используется строгая типизация.

Эта глава посвящена детальному описанию данных прикладных интерфейсов и системных функций отражения.

Системные функции отражения

Язык X++ предлагает набор системных функций, которые могут использоваться для отражения определений элементов. Они описываются в следующих разделах.

Встроенные функции

Встроенные функции подстановки следует использовать всякий раз при необходимости ссылки на прикладной элемент из X++-кода. Функции подстановки предоставляют способ создания строго типизированной ссылки. Компилятор распознает ссылку и проверяет, существует ли указанный прикладной элемент. Если элемент не существует, то код компилируется с ошибкой. Поскольку элементы имеют собственный цикл жизни, то ссылка не остается корректной вечно; элемент может быть переименован или удален. Использование функций подстановки позволяет гарантировать, что уже в процессе компиляции станет известно обо всех нарушенных ссылках. Ошибка компиляции в цикле разработки всегда предпочтительней ошибки времени выполнения в приложении заказчика.

Все ссылки, сделанные с использованием функций подстановки, обрабатываются средством Перекрестные ссылки. Это означает, что вы можете определить, где используется конкретный прикладной элемент, вне зависимости от того, указана ли ссылка в метаданных или в коде. Средство Перекрестные ссылки описывается в главе 2 .

Рассмотрим две реализации одного и того же кода, приведенные ниже:

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

Они приводят к абсолютно одинаковому результату: на экран выводиться строка "*MyClass*". С точки зрения ссылочной целостности, первая реализация является слабой.

В результате переименования или удаления этого класса ссылка будет нарушена, что приведет к дополнительным затратам времени на отладку кода приложения. Вторая реализация использует строгое соответствие типов и противостоит нарушениям целостности. При переименовании или удалении класса *MyClass* вы смогли бы использовать перекрестные ссылки для выполнения анализа влияния своих изменений.

С помощью функций подстановки вида <*ТипЭлемента*>*Str* можно ссылаться на все типы прикладных элементов в AOT по их именам. Некоторые

элементы также имеют идентификатор, с помощью которого на них можно ссылаться, используя функции подстановки вида *<ТипЭлемента>Num*. Функции подстановки не ограничиваются родительскими прикладными элементами. Они также могут существовать для методов классов, полей таблиц, индексов. Всего доступно более 50 функций подстановки. Ниже приводиться несколько примеров их использования.

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

Идентификатор элемента присваивается ему при создании и является последовательным номером, зависимым от выбранного слоя прикладной модели. В предыдущем примере *60001* – это идентификатор, присвоенный первому элементу поля таблицы *MyTable*, которое было создано в слое USR. Идентификаторы прикладных элементов объясняются в главе 21.

Следует также упомянуть еще две функции подстановки: *identifierStr* и *literalStr*. Первая позволяет ссылаться на элементы, когда недоступна более подходящая функция подстановки. *IdentifierStr* не используется для проверки существования элементов во время компиляции и не предоставляет сведений для системы перекрестных ссылок. Однако использование в коде функции *IdentifierStr* гораздо лучше, чем использование обычного текста, поскольку ясно указывается акцент на использовании ссылки на конкретный прикладной элемент. Если используется обычная текстовая константа, то такой акцент теряется – ссылка может быть текстом интерфейса пользователя, именем файла или чем-то совершенно другим. Инструмент Best Practices определяет использование функции *identifierStr* и генерирует предупреждение рекомендаций.

Среда времени выполнения Microsoft Dynamics AX автоматически преобразует любую ссылку на идентификатор метки в текст этой метки. В большинстве случаев это желаемое поведение. Однако вы можете запретить преобразование, используя функцию *literalStr*. Эта функция позволяет ссылаться на идентификатор метки, не преобразуя его в текст метки, как показано ниже.

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

В первой строке примера идентификатор метки (*@SYS1*) автоматически преобразуется в текст метки (Периодические проводки). Во второй

строке ссылка на идентификатор метки автоматически не преобразовывается.

Системная функция *typeOf*

Системная функция *TypeOf* принимает экземпляр переменной в качестве параметра и возвращает ее базовый тип. Пример использования функции показан ниже.

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

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

Результат выполнения этой функции является значением системного перечислимого типа *Types*. Этот перечислимый тип содержит перечень всех базовых типов в X++.

Системная функция *classIdGet*

Системная функция *ClassIdGet* принимает в качестве параметра объект класса и возвращает идентификатор класса данного объекта. Если в качестве параметра передается неинициализированный объект, то есть *Null*, то функция возвращает идентификатор класса для объявленного типа переменной, как показано ниже.

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

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

Данная функция особенно полезна при необходимости определения типа экземпляра объекта. Предположим, вам нужно определить, является ли переменная объектом определенного класса. В следующем примере показывается, как функция *ClassIdGet* может использоваться для определения идентификатора класса объекта *_anyClass*. Если переменная *_anyClass*

в действительности является объектом класса *MyClass*, ее можно присвоить переменной *myClass*.

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

Обратите внимание на то, как функция подстановки используется в процессе компиляции, а функция *classIdGet* – во время исполнения.

Поскольку в приведенном выше примере не учитывается возможность наследования классов, такая реализация может привести к нарушению объектной модели. В большинстве случаев любой экземпляр класса, наследуемого от *MyClass*, может рассматриваться как экземпляр класса *MyClass*.

Простейшим способом учета возможности наследования является использование операторов *is* и *as*.

Более подробную информацию см. в главе 4.

Примечание. Эта книга развивает тему наследования по принципу подстановки Барбары Лисков.

Прикладные интерфейсы отражения

Системная библиотека X++ содержит три интерфейса прикладного программирования, которые могут использоваться для отражения определений элементов. Они описываются в следующих разделах.

Интерфейс представления в виде таблицы

Предположим, что необходимо найти все классы, названия которых начинаются с Invent. В следующем примере кода приведен способ, как это можно сделать.

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

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

Таблица *SysModelElement* предоставляет доступ ко всем прикладным элементам приложения. Поле *ElementType* содержит контекст поиска. Модель данных хранилища модели данных содержит девять таблиц, которые показаны на рис. 20-1.

Примечание. Таблица *UtilElements* все еще доступна для целей обратной совместимости. Она реализована как агрегированное представление над таблицами *SysModel*. В целях повышения производительности вам следует ограничить использование этой возможности и применять новый интерфейс.

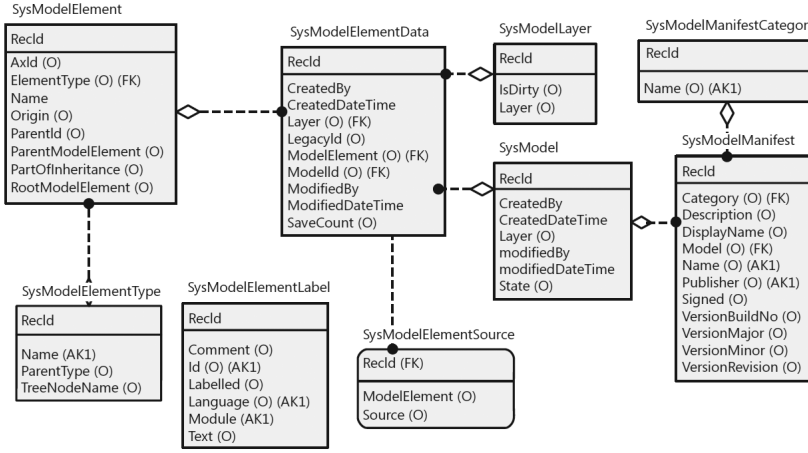


Рис. 20–1. Модель данных хранилища модели данных

По функциональности таблицу *SysModel* можно использовать так же, как источник данных форм и отчетов. Форма, показывающая данные из таблицы, доступна по следующему пути: Сервис > Управление моделью > Элементы Модели. В этой форме вы можете использовать стандарт-

ные запросы для фильтрации и поиска данных.Таблица *SysModelElement* содержит все элементы модели данных; присутствует связь с таблицей *SysModelElementData*, которая содержит определения каждого элемента. Каждая запись в *SysModelElement* – это как минимум одна запись в *SysModelElementData* и, возможно, дополнительно до 16 элементов по одному на каждый слой. Другими словами, эта таблица определяет гранулярность модификации элементов модели данных. Вы не можете модифицировать менее чем один элемент модели данных. Например, если вы измените одно свойство элемента модели данных, то вставится новая запись в таблицу *SysModelElementData*, в которой содержатся все свойства элемента.

Примечание. Системные элементы, которые представлены в узле *System Documentation* дерева AOT, **не представлены в этих таблицах.**

Элементы иерархически структурированы. В основании находится корневой элемент, например форма. Форма содержит источник данных, элемент управления и метод элемента. Иерархия может включать несколько уровней, например, элемент формы может иметь несколько методов. Корневой элемент, родительский элемент представлены в полях *RootModelElement* и *ParentModelElement* таблицы *SysModelElement*.

Следующее задание находит все элементы формы *CustTable* и выводит в виде списка название и тип элемента, название родительского элемента и путь в AOT из класса *TreeNode*.

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

    while select name from modelElement
        join Name from modelElementType
            where modelElementType.RecId == modelElement.ElementType
        join name from parentModelElement
            where parentModelElement.RecId == modelElement.ParentModelElement
        exists join rootModelElement
            where rootModelElement.RecId == modelElement.RootModelElement
                && rootModelElement.Name == formStr(CustTable)
                && rootModelElement.ElementType == UtilElementType::Form
```

```
    {
        info(strFmt("%1, %2, %3, %4",
            parentModelElement.Name, modelElementType.Name, modelElement.Name,
            SysTreeNode::modelElement2Path(modelElement)));
    }
}
```

Обратите внимание на использования поля *ElementType* в этих двух примерах. Если тип элемента *UtilElement*, то вы найдете соответствие в перечислимом типе *UtilElementType*; альтернативно вы можете присоединять таблицу *SysModelElementType*, в которой содержится информация о всех типах элементов.

В ней есть все корневые элементы и некоторые бывшие подчиненные элементы *Utilelements*. Вы можете получить доступ к ним доступ через таблицу *UtilElements*. Модели данных были представлены в этой версии для поддержки более гранулярных модификаций элемента приложения, что, помимо всего прочего, способствует более легкому обновлению приложения и обеспечивает возможность независимой установки различных моделей данных.

Дополнительную информацию см. в главе 21.

В табл. 20-1 дана информация по таблицам и представлениям. На рис. 20-1 вы можете наблюдать, как эти таблицы связаны друг с другом.

Табл. 20–1. Таблицы отражения и представления

Название таблицы или представления	**Описание**
SysModel	Таблица, содержащая модели в хранилище моделей
SysModelElement	Таблица, содержащая элементы в хранилище моделей. Точное соответствие одно записи элементу в AOT несмотря на доработки
SysModelElement-Data	Таблица, содержащая определения элементов в хранилище моделей. Точное соответствие одной записи для каждого элемента на каждом слое
SysModelElement-Label	Таблица, содержащая все тексты меток и комментарии
SysModelElement-Source	Таблица, содержащая весь исходный код X++

Табл. 20–1. Таблицы отражения и представления (*окончание*)

Название таблицы или представления	Описание
SysModelElementType	Таблица, содержащая определения типов элементов. Информация в таблице статична и заполняется во время установки
SysModelLayer	Таблица, содержащая слои. Информация в таблице статична и заполняется во время установки. Одна запись на каждый из 16 слоев
SysModelManifest	Таблица, содержащая информацию о содержимом моделей, такую как название, издатель и номер версии. Одна запись создается на каждую модель
SysModelManifestCategory	Таблица, содержащая категории, к которым принадлежит модель. Каждая модель принадлежит к одной из следующих категорий: Стандарт, Hotfix, Виртуальная или Временная
UtilElements, UtilIdElements	Агрегированные представления над таблицами *SysModel*. Используются для обратной совместимости
UtilModels	Представление над таблицами *SysModel, SysModelManifest* и *SysModelLayer,* которое упрощает использование запросов по модели

Примечание. Существуют альтернативные версии таблиц, описанных в табл. 20-1. Если вы запустите поиск по объектам с окончанием имени *old*, то получите доступ к базовой модели вместо основной. Например, таблица *SysModelElementOld* содержит элементы в базовой модели. Базовая модель используется в сценариях обновления.

Используйте пространство имен *Microsoft.Dynamics.AX.Framework.Tools.ModelManagement* библиотеки сборки *AxUtilLib.dll* для создания, импорта, экспорта или удаления моделей. Эта сборка доступна из кода на X++ с помощью класса-обертки *SysModelStore,* который используется для упрощенного доступа.

Примечание. При использовании интерфейса отражения таблиц с поддержкой контроля версий, значения некоторых полей сбрасываются в процессе сборки. Для файло-ориентированной

системы контроля версий процесс построения импортирует файлы .xpo в пустые слои системы Microsoft Dynamics AX. Значения полей *CreatedBy*, *CreatedDateTime*, *ModifiedBy* и *ModifiedDateTime* устанавливается во время процесса импорта, и поэтому меняется от сборки к сборке.

Прикладной интерфейс словаря данных

API словаря данных приложения – это строго типизированный интерфейс прикладного программирования, который может отражать множество прикладных элементов. Следующий пример кода является модифицированной копией предыдущего примера, который находит классы модуля управления запасами, но уже с использованием интерфейса словаря данных. Данный интерфейс не может быть использован для извлечения сведений о дате последней модификации элемента. Вместо этого данный пример отражает немного больше информации о классах, при этом перечисляются только абстрактные классы:

```
static void findAbstractInventoryClasses(Args _args)
{
    Dictionary dictionary = new Dictionary();
    int i;
    DictClass dictClass;
    for(i=1; i<=dictionary.classCnt(); i++)
    {
        dictClass = new DictClass(dictionary.classCnt2Id(i));

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

Класс *Dictionary* предоставляет сведения о существующих прикладных элементах. С помощью данной информации вы можете создать объект класса *DictClass*, который предоставляет определенные сведения о классах, например, является ли класс абстрактным (abstract), завершенным (final) или же интерфейсом (interface), от какого класса он наследуется, реализует ли он другие интерфейсы, какими атрибутами он помечен, а также

перечень его методов. Обратите внимание на то, что класс *DictClass* также может работать с интерфейсами, и на то, как счетчик классов преобразуется в идентификатор класса; такое преобразование требуется, поскольку идентификаторы в AOT перечислены не последовательно.

При запуске данного задания вы увидите, что оно выполняется гораздо медленнее, чем реализация, использующая интерфейс с табличным представлением – как минимум при первом запуске! Задание станет выполняться быстрее после того, как требуемая информация будет закэширована.

На рис. 20-2 показана объектная модель для интерфейса словаря данных. Как вы можете видеть, некоторые элементы не могут быть отображены с использованием данного API.

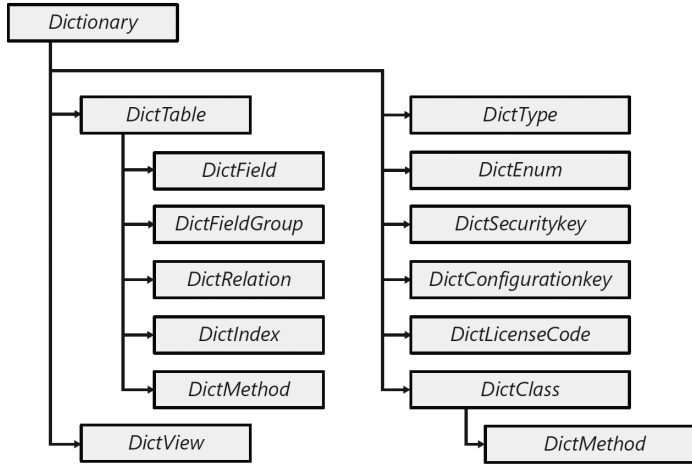


Рис. 20–2. Объектная модель интерфейса словаря данных *Dictionary*

В следующем примере приводится обработка статического метода на таблице *CustTable*.

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

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

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

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

Как упоминалось ранее, отражение также может использоваться для вызова методов объектов. В данном примере выполняется вызов статического метода *Find* таблицы *CustTable*.

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

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

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

Обратите внимание на использование функции подстановки *tableStaticMethodStr* для строго типизированной ссылки на метод *Find* таблицы.

Данный API может также использоваться для создания объектов классов и таблиц.

Предположим, что необходимо выбрать все записи в таблице с указанным идентификатором. В следующем примере показано, как это можно реализовать.

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

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

Во-первых, обратите внимание на вызов метода *makeRecord*, который создает объект буфера записей таблицы, указывающий на корректную таблицу. После этого можно использовать оператор *select* для выбора записей из таблицы. При необходимости можно аналогично вставить записи с использованием этого буфера записей таблицы. Обратите внимание на синтаксис, используемый для получения значения поля из объекта буфера записей; данный синтаксис позволяет обращаться к любому полю по его идентификатору. В данном примере просто печатается содержимое поля первичного ключа. Для создания объекта заданного класса может использоваться метод *makeObject* класса *DictClass*.

Все классы в API словаря данных, которые обсуждались до сих пор, являются классами системного API. Поверх каждого из этих классов присутствует определенный в приложении класс, который предоставляет еще больше возможностей отражения. Данные классы называются *SysDict<ТипЭлемента>*, а каждый конкретный класс является наследником своего аналога в системном API. Например, класс *SysDictClass* является наследником класса *DictClass*.

Рассмотрим следующий пример. Поля таблицы имеют свойство, которое указывает, является ли поле обязательным к заполнению. Класс *DictField* возвращает значение данного свойства, как набор битов, возвращаемый методом *flag* этого класса. Проверка набора битов – это громоздкое и некрасивое решение, и, если реализация флагов изменится, то клиентские приложения будут работать некорректно. Класс *SysDictField* инкапсулирует логику проверки битов в методе *mandatory*. Ниже показано, как используется данный метод.

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

    while (enum.moveNext())
    {
        sysDictField = enum.current();
        if (sysDictField.mandatory())
        {
            info(sysDictField.name());
        }
    }
}
```

Возможно имеет смысл просмотреть статические методы классов *SysDict**. Многие из них предоставляют дополнительные сведения отражения и улучшенные интерфейсы. К примеру, класс *SysDictionary* содержит метод *classes*, который возвращает коллекцию экземпляров класса *SysDictClass* для всех классов приложения. Данный метод может использоваться для упрощения примера задания *findAbstractInventoryClasses* из предыдущего примера.

Интерфейс узлов AOT (класс Treenode)

Оба интерфейса отражения, обсуждаемые до сих пор, имели определенные ограничения. API табличного представления может отражать только существующие элементы и небольшое подмножество метаданных этих прикладных элементов. API словаря данных может отражать сведения только о тех типах элементов, интерфейс для которых существует в данном API.

API узлов репозитария прикладных элементов может отражать всю информацию обо всех прикладных элементах, но, как всегда происходит, за такую мощь приходится платить. API узлов репозитария прикладных элементов труднее в использовании, чем другие интерфейсы отражения, обсуждаемые в данной главе. Он может вызвать проблемы производительности и требуемой памяти, а также не является строго типизированным интерфейсом.

Следующий код является модификацией примера кода из раздела «Интерфейс представления в виде таблицы», в котором выполняется поиск классов модуля управления запасами с использованием API узлов репозитария прикладных элементов.

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

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

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

Во-первых, обратите внимание на то, как производится поиск конкретного узла в AOT на основе пути, представленного в виде строки. Макрос AOT содержит определения для основных путей в AOT. В целях улучшения читаемости кода примеры в данной главе не используют этот макрос. Обратите внимание на использование класса *TreeNodeIterator* для просмотра классов в цикле.

Следующее небольшое задание выводит на экран исходный код метода *find* на таблице *CustTable*, вызывая метод *AOTgetSource* объекта *treenode* для метода *find*.

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

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

API узлов репозитария прикладных элементов предоставляет доступ к исходному коду узлов в AOT. Вы можете использовать класс *ScannerClass* для преобразования строки, которая содержит исходный код в последовательность маркеров компиляции. В следующем коде выполняется поиск полей таблицы *CustTable*, обязательных к заполнению.

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

    TreeNode field = fieldsNode.AOTfirstChild();

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

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

Обратите внимание на альтернативный способ циклического прохода по подузлам дерева. И этот подход, и подход, использующий итератор, прекрасно работают. Единственный способ определить, что поле является обязательным к заполнению с помощью данного API, – это иметь информацию о том, что выбранный узел моделирует поле таблицы и что узлы табличных полей содержат свойство с именем *Mandatory*, которое для обязательных полей установлено в значение Yes (а не True).

При ссылке на имена свойств используйте глобальный макрос *Properties*. Он содержит текстовые определения для всех имен свойств. Применяя данный макрос, вы избежите использования констант в качестве имен, как при ссылке на поле *Mandatory* в предыдущем примере.

В отличие от API словаря данных, который может отражать не все типы прикладных элементов, API узлов AOT может отражать все что угодно. Данный факт используется в классе *SysDictMenu*, который предоставляет строго типизированный способ отражения меню и пунктов меню путем конвертации сведений, предоставленных интерфейсом узлов AOT, в строго типизированный API. В следующем задании выполняется печать структуры меню *MainMenu*, которое обычно показывается в области переходов.

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

```
        SysMenuEnumerator enumerator;

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

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

Обратите внимание на то, как функция *setPrefix* используется для отображения иерархии меню и как рекурсивно вызывается функция *reportLevel*.

API узлов AOT также позволяет отражать формы и отчеты, их структуру, свойства и методы. Инструмент Сравнение в среде MorphX использует данный API для сравнения выбранного узла с другими узлами. Класс *SysTreeNode* содержит переменную класса *TreeNode* и реализует набор интерфейсов, которые позволяют использовать класс *TreeNode* в инструментах разработки Сравнение и Контроль версий. Класс *SysTreeNode* также содержит большой набор полезных статических методов.

Класс *TreeNode* в действительности является базовым классом еще большей иерархии прикладных элементов. Вы можете привести объекты к специализированным классам *TreeNode*, которые предоставляют намного более конкретную функциональность. Вы можете просмотреть иерархию наследников *TreeNode* в AOT, выбрав System Documentation, затем Classes, щелкнув правой кнопкой мыши на классе *TreeNode*, выбрав Надстройки, затем Иерархия объектов.

Несмотря на то что в данном разделе обсуждается только функциональность отражения API узлов репозитария прикладных объектов, вы можете использовать данный интерфейс еще и как AOT-дизайнер. Вы

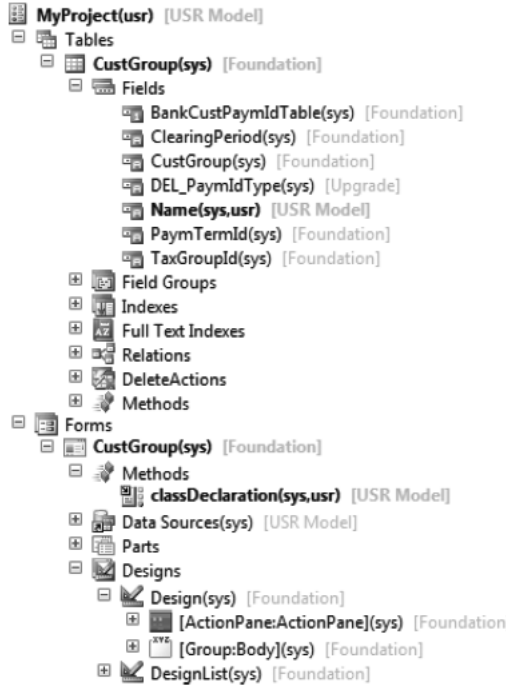
можете создавать новые элементы и изменять свойства и исходный код существующих. Мастер мастеров использует API узлов AOT для генерации проекта, формы и класса, реализующего функциональность мастера. Вы также можете компилировать и получать определения узлов с других прикладных слоев и узлов базовой модели. Данные возможности, которые выходят за рамки отражения, представляют мощную функциональность, которую следует использовать с особым вниманием. Извлечение информации в не строго типизированной манере требует внимания, но написание кода в такой манере может привести к катастрофическим последствиям.

TreeNodeType

Различные типы узлов имеют различный набор свойств и возможностей. Класс *TreeNodeType* может использоваться для анализа узла дерева AOT. Класс *TreeNodeType* обеспечивает надежную альтернативу возможностям класса *treenode*, основанную на свойствах. В предыдущей версии Microsoft Dynamics AX можно найти лишь немногие примеры его использования в базовом коде; например, считалось, что *treenode* поддерживает контроль версий, только если имеет свойство *utilElementType* и не имеет родительского ID.

Класс *TreeNodeType* содержит метод, возвращающий информацию о типе узла, плюс семь методов, которые возвращают тип *Boolean* и содержат информацию о свойствах узла. Применение этих методов описывается далее в этой главе. На рис. 20-3 представлена информация с помощью класса *TreeNodeType* для каждого узла некого проекта, содержащего таблицу и форму. Слева располагается снимок экрана свойств самого проекта. Справа находится таблица-список по каждому узлу с ID типа и набором свойств.

- ▪ ***ID.*** Идентификатор типа узла. Определен в системе и доступен в макросе *TreeNodeSysNodeType*. Узлы с одинаковыми идентификаторами имеют одинаковое поведение.
- ▪ ***isConsumingMemory.*** В дереве MorphX есть узлы, которыми система Microsoft Dynamics AX во время исполнения не управляет, и их память не освобождается автоматически. Для каждого узла, у которого стоит признак *isConsumingMemory*, вам следует вызывать метод *treenodeRelease* для освобождения памяти, если вы более не имеете ссылок на вложенные подузлы. Также вы можете использовать класс ***TreeNode-***



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

Рис. 20–3. Информация, полученная с помощью класса *TreeNodeType* узлов таблицы и формы

Traverser, так как он выполняет ту же задачу за вас. Пример использования вы найдете в методе *traverseTreeNodes* класса *SysBpCheck*.

- ***isGetNodeInLayerSupported.*** С узлами дерева, которые поддерживают метод *getNodeInLayer*, вы можете получать информацию о версии узла в других слоях. Другими словами, с помощью этого метода вы можете получать доступ к узлам в нижележащих слоях.

- ***isLayerAware.*** Узлы дерева, которые зависят от слоя, отображают индикатор слоя в AOT; например, *SYS* или *USR*. Вы можете получить слой, на котором находится узел, с помощью метода *AOTLayer*: Чтобы получить информацию обо всех доступных слоях, используйте метод *AOTLayers*. Заметим, что метод *AOTLayers* не разворачивает слои для подузлов: этот метод возвращает информацию, которая отображается в AOT. Информация о развертывании слоев доступна через вызов метода *ApplObjectLayerMask*, который используется в AOT для определения, какой узел должен быть выделен жирным шрифтом. Если узел выделен жирным шрифтом в AOT, либо сам узел, либо один из подчиненных подузлов изменен на текущем слое.
- ***isModelElement.*** Узлы дерева являются элементами модели и представлены в таблице SysModelElement.
- ***isRootElement.*** Корневой элемент располагается в основании иерархии дерева объектов, а также поле *RootModelElement* для всех элементов подмодели ссылается на *recid корневого* элемента.
- ***isUtilElement.*** Если данный узел дерева – это *UtilElement*, то соответствующая запись будет найдена в представлении *UtilElements*. Более того, информация о первичном ключе может быть получена из метода элемента *utilElement*.
- ***isVCSControllableElement.*** Вы можете использовать метод *isVCSControllableElement* для определения подробной информации, основанной на файловых артефактах, которые хранится в системе контроля версий. В большинстве случаев в системе контроля версий поддерживается гранулярность до уровня корневого элемента; другими словами, в системе контроля версий вы работаете с целым объектом формы, класса и таблицы. Однако для элементов Microsoft Visual Studio уровень гранулярности отличен от обычного, и вы можете работать на уровне файлов Visual Studio, например, с файлами .cs.

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

В следующем примере показано, как получить доступ к информации для узла дерева.

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

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

Примечание. *Используйте класс TreeNodeType для отражения метамодели. Эти функции класса находятся на более высоком уровне абстракции — вместо отражения элементов* AOT **они отражают типы элементов.** *Класс SysModelMetaData предоставляет другой путь для отражения метамодели.*

.