Глава 14

Расширение возможностей Microsoft Dynamics AX

В этой главе

- Введение
- Инфраструктура SysOperation
- Сравнение инфраструктур SysOperation и RunBase
- Инфраструктура RunBase
- Инфраструктура расширений
- Обработка событий

Введение

Microsoft Dynamics AX содержит несколько инфраструктур, которые вы можете использовать для расширения приложения. В версии 2012 инфраструктура SysOperation заменила инфраструктуру RunBase в части поддержки заданий обработки данных, таких как расчет курсовых разниц или закрытие склада. Microsoft Dynamics AX также предлагает два способа расширения: инфраструктура расширений, предназначенная для разработки новых модулей, и инфраструктура событий, основанная на концепции событий среды Microsoft .NET.

В первой части этой главы делается обзор инфраструктуры SysOperation и обсуждаются примеры, сравнивающие инфраструктуры SysOperation и RunBase. Следующий раздел более глубоко раскрывает информацию о классах RunBase с целью помочь понять существующую функциональность, разработанную с помощью инфраструктуры RunBase.

Заключительные разделы описывают инфраструктуры расширений и событий. Инфраструктура расширений уменьшает число или исключает зависимости между компонентами приложения и их расширениями. Инфраструктура событий является новой для Microsoft Dynamics AX. Методы в классе X++ могут вызывать события непосредственно перед своим

запуском (события *pre*) и после своего завершения (события *post*). Эти два события дают возможность вставки вашего произвольного кода в процесс выполнения программы с помощью обработчика событий.

Инфраструктура SysOperation

Используйте инфраструктуру SysOperation при необходимости создания логики приложения, поддерживающей интерактивный запуск операций или их работу на пакетном сервере Microsoft Dynamics AX. Возможности, предоставляемые этой инфраструктурой, похожи на особенности, имеющиеся в инфраструктуре RunBase.

Среда пакетных заданий, детально описанная в главе 18, предъявляет к выполняющимся в ней операциям специфические требования.

- Операция должна поддерживать сериализацию параметров для записи их в таблицу пакетных заданий.
- Операция должна иметь способ показать пользовательский интерфейс, позволяющий изменить параметры пакетного задания. Подробнее о пакетных заданиях в Microsoft Dynamics AX можно прочитать в главе 18 и в статье «Process batch jobs and tasks» по адресу: *http://technet.microsoft.com/en-us/library/gg731793.aspx.*
- Операция должна реализовывать интерфейсы, требуемые для интеграции с пакетным сервером во время их выполнения.

В то время как инфраструктура RunBase задает способы реализации, отвечающие этим требованиям, инфраструктура SysOperation идет дальше, реализуя базовую функциональность для многих интерфейсов и классов в своих шаблонах проектирования.

В отличие от RunBase, инфраструктура SysOperation реализует шаблон проектирования Model-View-Controller (MVC), отделяя уровень представления от бизнес-логики. Подробнее см. в статье «Model-View-Controller» по адресу: *http://msdn.microsoft.com/en-us/library/ff649643.aspx*.

Классы инфраструктуры SysOperation

Класс *SysOperationServiceController* содержит несколько полезных методов:

- ***getServiceInfo*** – возвращает действия службы;

- ***getDataContractInfo*** – возвращает контракты данных, используемые как параметры и возвращаемые значения действия службы, получает информацию о построении пользовательского интерфейса для каждого контракта данных;
- ***startOperation*** – осуществляет вызов службы в различных режимах, включая синхронный, асинхронный и пакетный.

Классы *SysOperationUIBuilder* и *SysOperationAutomaticUIBuilder* помогают создавать пользовательский интерфейс по умолчанию, исходя из определения контракта данных или из определения производной формы. Вы можете написать собственный построитель пользовательского интерфейса, унаследованный от этих базовых классов, для реализации задания значений по умолчанию и валидации, или чтобы вызвать какие-либо события. Вы можете перегрузить следующие методы.

- ***postBuild.*** Переопределение этого метода позволяет получить ссылки на элементы управления диалогового окна, если построитель интерфейса – динамический (другими словами, он не основан на какой-либо форме).
- ***postRun.*** Переопределение этого метода дает возможность подключить свои методы валидации.

Атрибуты инфраструктуры SysOperation

Атрибуты *SysOperation* указывают метаданные контрактов данных для организации слабого связывания с построителями пользовательского интерфейса. Доступны следующие атрибуты.

- ***DataContractAttribute.*** Указывает, что класс является контрактом данных.
- ***DataMemberAttribute.*** Указывает, что свойство является членом данных.
- ***SysOperationContractProcessingAttribute.*** Отмечает построитель пользовательского интерфейса по умолчанию для контракта данных.
- ***SysOperationLabelAttribute, SysOperationHelpTextAttribute*** и ***SysOperationDisplayOrderAttribute.*** Указывают атрибуты метки, подсказки и порядка отображения, соответственно, для членов данных.

Сравнение инфраструктур SysOperation и RunBase

Инфраструктуры SysOperation и RunBase разработаны для построения приложений с операциями, которые могут выполняться как на пакетном сервере, так и интерактивно. Для выполнения операции на пакетном сервере она должна поддерживать следующее.

- Сериализацию параметров при помощи интерфейса *SysPackable.*
- Стандартный метод *run,* как определено в интерфейса *BatchRunable.*
- Методы интеграции с пакетным сервером, расположенные в интерфейсе *Batchable.*
- Пользовательский интерфейс, позволяющий принять вводимые пользователем данные.

На рис. 14-1 показано, как все операции, которые должны выполняться посредством пакетного сервера, должны быть унаследованы от базовых классов *SysOperationController* или *RunBaseBatch.*

Примеры кодов в последующих разделах иллюстрируют основные возможности, предоставляемые двумя инфраструктурами. Эти примеры могут работать как интерактивно (с показом диалоговых окон), так и в пакетном режиме.

Чтобы ознакомиться с примерами, сделайте импорт проекта *PrivateProject_SysOperationIntroduction.xpo* и нажмите Ctrl+Shift+P для просмотра кода примеров в окне Projects. Следующие два класса примеров расположены в узле *Sample_1_SysOperation_Runbase_Comparison*:

- *SysOpSampleBasicRunbaseBatch;*
- *SysOpSampleBasicController.*

Эти классы позволяют сравнить функциональность инфраструктуры RunBase с функциональностью SysOperation.

Перед тем как вы запустите примеры на исполнение, откомпилируйте проект и сгенерируйте для них исполнимый код (CIL).

1. В среде разработки щелкните правой кнопкой мыши на названии проекта и выберите Компилировать.
2. Щелкните Сборка и затем выберите Создать инкрементный CIL (или нажмите Ctrl+Shift+F7).

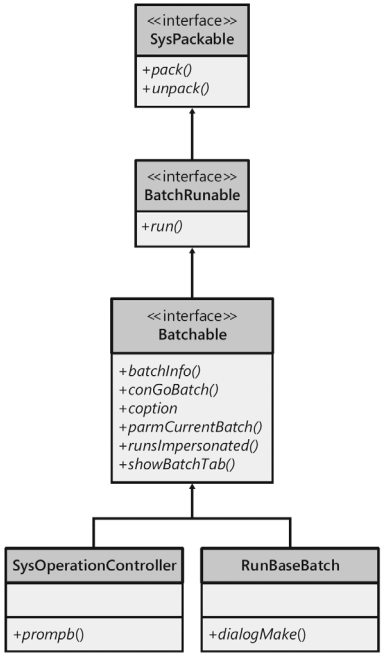

Рис. 14–1. Схема наследования для операций, поддерживающих выполнение на пакетном сервере

Пример RunBase: *SysOpSampleBasicRunbaseBatch*

Эта простейшая операция, основанная на базовом классе *RunBaseBatch*, должна реализовывать несколько переопределенных методов. В табл. 14-1 описаны переопределенные методы, реализованные в классе *SysOpSampleBasicRunbaseBatch*. Пример кода, расположенный после таблицы, иллюстрирует, как использовать эти методы.

Табл. 14–1. Переопределенные методы класса *RunBaseBatch*

Методы	Описания
dialog	Заполняет диалоговое окно, созданное базовым классом, элементами управления, требующимися для принятия вводимых пользователем данных
getFromDialog	Переносит содержимое элементов управления диалогового окна во входные параметры операции
putToDialog	Переносит содержимое входных параметров операций в элементы управления диалогового окна

Табл. 14–1. Переопределенные методы класса *RunBaseBatch (окончание)*

Методы	**Описания**
pack	Сериализует входные параметры операции
unpack	Десериализует входные параметры операции
run	Выполняет операцию
description	Статический метод, возвращающий описание операции

В переопределенном методе *classDeclaration* наследника *RunBaseBatch* вы должны задать переменные для входных параметров элементов управления в диалоговом окне и макрокоманду LOCALMACRO, задающую список сериализуемых переменных:

```
class SysOpSampleBasicRunbaseBatch extends RunBaseBatch
{
    str text;
    int number;
    DialogRunbase       dialog;

    DialogField numberField;
    DialogField textField;

    #define.CurrentVersion(1)

    #LOCALMACRO.CurrentList
        text,
        number
    #ENDMACRO
}
```

Далее перегрузите метод *dialog*. Этот метод помещает в диалоговое окно, созданное базовым классом, два элемента управления, принимающие пользовательский ввод: текстовое поле и цифровое поле. Начальные значения из переменных – членов класса используются для инициализации элементов управления. Обратите внимание, что тип каждого элемента управления определен заданным именем расширенного типа данных (EDT):

```
protected Object dialog()
{
    dialog = super();
```

```
        textField = dialog.addFieldValue(IdentifierStr(Description255),
            text,
            'Текстовый параметр',
            'Введите текст');

        numberField = dialog.addFieldValue(IdentifierStr(Counter),
            number,
            'Числовой параметр',
            'Введите число');

        return dialog;
    }
```

Перегруженный метод *getFromDialog* переносит содержимое элементов управления диалогового окна во входные параметры нашей обработки:

```
    public boolean getFromDialog()
    {
        text = textField.value();
        number = numberField.value();

        return super();
    }
```

Перегруженный метод *putFromDialog* переносит содержимое входных параметров в элементы управления диалогового окна:

```
    protected void putToDialog()
    {
        super();

        textField.value(text);
        numberField.value(number);
    }
```

Перегруженные методы *pack* и *unpack* упаковывают и распаковывают входные параметры обработки:

```
    public container pack()
    {
        return [#CurrentVersion, #CurrentList];
    }
    public boolean unpack(container packedClass)
    {
        Integer version = conPeek(packedClass,1);
```

```
    switch (version)
    {
        case #CurrentVersion:
            [version,#CurrentList] = packedClass;
            break;
        default:
            return false;
    }
    return true;
}
```

Перегруженный метод *run* выполняет обработку. Приведенный ниже пример печатает входные параметры в Infolog. Также он печатает уровень, на котором выполняется обработка, и среду выполнения обработки.

```
public void run()
{
    if (xSession::isCLRSession())
    {
        info('Выполняется в сессии CLR.');
    }
    else
    {
        info('Выполняется в интерпретируемой сессии.');
        if (isRunningOnServer())
        {
            info('Работает на AOS.');
        }
        else
        {
            info('Работает на клиенте.');
        }
    }

    info(strFmt('SysOpSampleBasicRunbaseBatch: %1, %2', this.parmNumber(), this.
parmText()));
}
```

Метод *description* позволяет задать обработке какое-либо определенное описание. Перегрузите его, как показано в примере ниже, и используйте возвращаемое им описания для задания заголовка, отображаемого в списке пакетных заданий и в заголовке пользовательского интерфейса:

```
public static ClassDescription description()
{
    return 'Простой пример RunBaseBatch';
}
```

Перекройте метод *main*, выводящий запроса пользователя и выполняющий затем обработку или же добавляющий ее в очередь пакетных заданий, как показано в примере:

```
public static void main(Args _args)
{
    SysOpSampleBasicRunbaseBatch operation;

    operation = new SysOpSampleBasicRunbaseBatch();
    if (operation.prompt())
    {
        operation.run();
    }
}
```

Методы *parmNumber* и *parmText*, вообще говоря, необязательны. Но рекомендации по стилю кодирования в Microsoft Dynamics AX советуют следовать шаблону создания методов-свойств, дающих доступ к параметрам обработки, чтобы улучшить возможности тестирования и предоставить возможность доступа к переменным класса извне. Создайте эти методы, как в примере ниже.

```
public int parmNumber(int _number = number)
{
    number = _number;

    return number;
}
public str parmText(str _text = text)
{
    text = _text;

    return text;
}
```

Метод *main* в примере *RunBaseBatch* запрашивает у пользователя параметры обработки при вызове *operation.prompt*. Если метод *prompt* возвращает *true*, то *main* сразу вызывает метод *operation.run*. Если же *prompt* вернул *false*, это означает, что пользователь либо отменил запуск обработ-

ки, либо поставил ее в очередь для обработки в пакетном режиме. Для запуска примера в интерактивном режиме запустите метод *main*, щелкнув в окне редактора кода на кнопке Go, как показано на рис. 14-2.

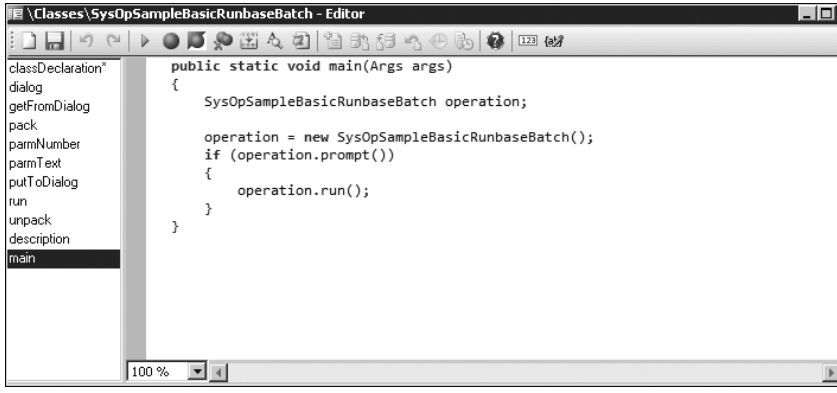

Рис. 14–2. Окно редактора кода с кодом *SysOpSampleBasicRunbaseBatch*

На вкладке Разное пользовательского диалогового окна примера введите информацию в поля Текстовый параметр и Числовой параметр, как показано на рис. 14-3.

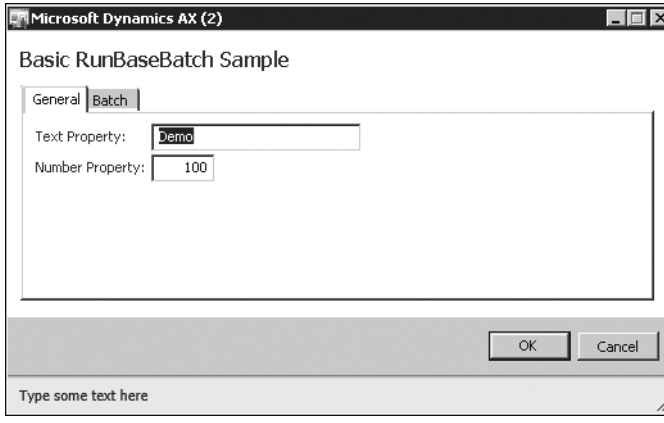

Рис. 14–3. Вкладка Общее пользовательского диалогового окна обработки *SysOpSampleBasicRunbaseBatch*

Убедитесь, что на вкладке Пакет снята отметка Пакетная обработка, как показано на рис. 14-4.

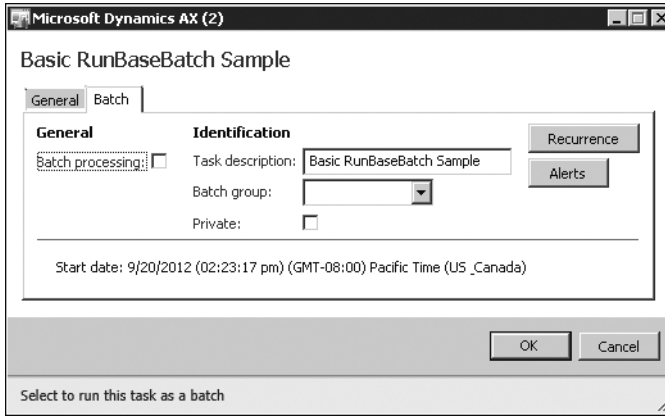

Рис. 14–4. Вкладка Пакет пользовательского диалогового окна обработки *SysOpSampleBasicRunbaseBatch*

Нажмите **OK для запуска обработки, и она выведет в Infolog результа**ты своего выполнения.

Посмотрите на выведенные в Infolog сообщения, показанные на рис. 14-5. Они показывают, что обработка выполнялась на сервере, так как класс примера *SysOpSampleBasicRunbaseBatch* имеет свойство *RunOn*, установленное в *Server*. Обработка выполнялась с помощью интерпретатора X++, что является поведением по умолчанию для кода X++.

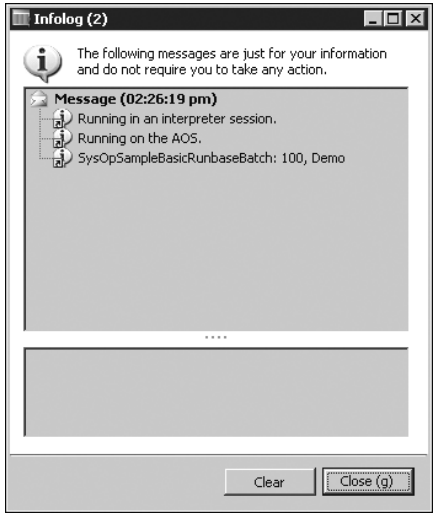

Рис. 14–5. **Окно** Infolog с выводом *SysOpSampleBasicRunbaseBatch*

Для запуска обработки в пакетном режиме запустите ее снова нажатием Go в редакторе кода и введите значения для полей Текстовый параметр и Числовой параметр на вкладке Разное пользовательского диалогового окна.

Далее поставьте отметку Пакетная обработка на вкладке Пакет, чтобы перезапустить обработку на пакетном сервере. При установленной отметке Пакетная обработка, в Infolog будет выведено сообщение, показанное на рис. 14-6, сообщающее о добавлении обработки в очередь пакетов. (Не забывайте, что вам нужно будет сделать инкрементную CIL-компиляцию обработки, чтобы она смогла выполниться на пакетном сервере через меню Сборка > Создать инкрементный CIL. – *Прим. перев.*)

Рис. 14–6. Окно Infolog, сообщающее о добавлении задания в пакетную очередь

Обработка может занять около минуты. Подождите немного и откройте форму BatchJob из узла *Forms* в дереве объектов приложения (AOT), как показано на рис. 14-7.

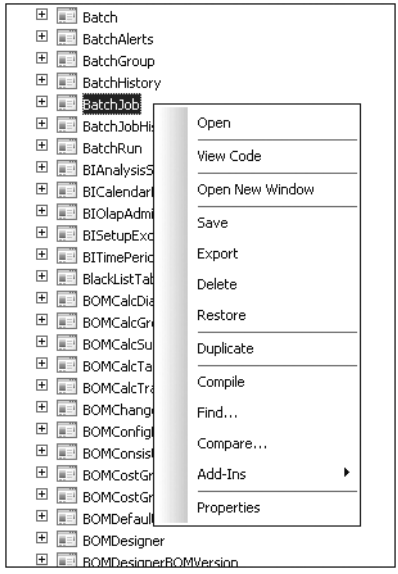

Рис. 14–7. Узел AOT *Forms*

Откроется форма Описание задания, изображенная на рис. 14-8.

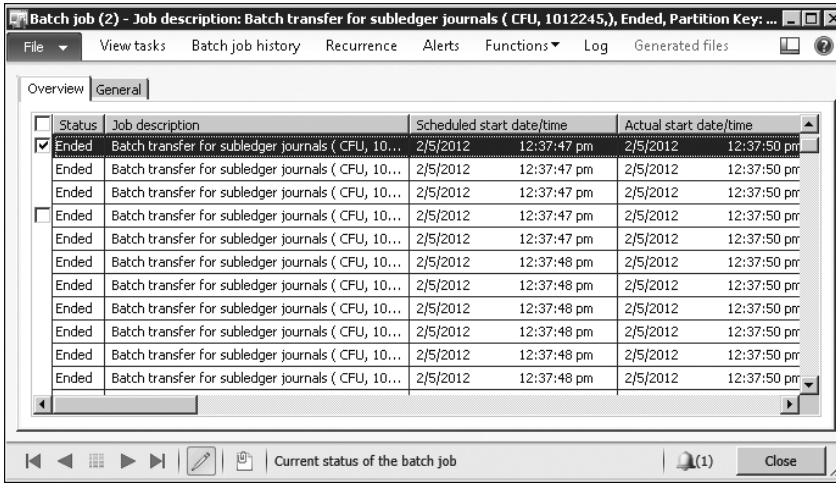

Рис. 14–8. Форма Описание задания, показывающая статус пакетных заданий

Нажмите **F5 для обновления содержания формы, и продолжайте нажимать** F5, пока в форме не будет показано, что ваше пакетное задание

завершилось. Отсортируйте форму по столбцу Запланированные дата/время начала, что поможет найти вашу обработку, если их отображается в списке слишком много.

Для просмотра журнала, выберите нужную обработку и нажмите на кнопку Журнал на панели инструментов.

Окно Infolog на рис. 14-9 показывает сообщения, информирующие о том, что обработка проводилась в сессии CLR, которая и является средой выполнения для сервера пакетной обработки.

Рис. 14–9. Сообщения класса *SysOpSampleBasicRunBaseBatch*

Пример SysOperation: *SysOpSampleBasicController*

Как говорилось ранее, инфраструктура SysOperation предлагает такие же возможности, что и инфраструктура RunBase, но с базовой реализацией наиболее часто переопределяемых методов. Инфраструктура SysOperation управляет созданием простого пользовательского интерфейса, упаковкой параметров и взаимодействием со средой выполнения CLR.

Приводимый пример SysOperation содержит два класса: управляющий класс *SysOpSampleBasicController* и контракт данных *SysOpSampleBasicDataContract*. В табл. 14-2 описаны переопределяемые методы, нужные для реализации такой же функциональности, что и в примере RunBase из предыдущего раздела. Заметьте, что вам не нужно перегружать методы *dialog, getFromDialog, putToDialog, pack, unpack* и *run* в классе *SysOpSampleBasicController*, так как инфраструктура SysOperation уже включает базовую функциональность для них. В коде примера далее будет показано, как использовать эти методы.

Табл. 14–2. Переопределяемые методы в классах *SysOpSampleBasicController* и *SysOpSampleBasicDataContract*

Метод	Описание
Класс *SysOpSampleBasicController*	
classDeclaration	Унаследован от базового класса инфраструктуры
new	Определяет класс и способ действия обработки
showTextInInfolog	Печатает входные параметры, уровень выполнения и среду выполнения в Infolog
caption	Возвращает описание обработки
main	Выполняет обработку
Класс *SysOpSampleBasicDataContract*	
classDeclaration	Атрибут контракта данных, используемый базовой инфраструктурой для отражения обработки
parmNumber	Атрибут члена данных отмечает этот метод-свойство как часть контракта данных. Атрибуты метки, подсказки и порядка отображения дают информацию для построения пользовательского интерфейса
parmText	См. описание *parmNumber*

Важно. Исходно класс *SysOpSampleBasicController* должен быть унаследован от *SysOperationServiceController*, который и содержит всю базовую функциональность для реализации обработок. Однако версия класса, входящая в Microsoft Dynamics AX 2012, содержит несколько известных ошибок, планируемых к исправлению в следующем пакете обновления. Для обхода этих ошибок есть новый общий класс – *SysOpSampleBaseController*. Для получения более подробной информации обратитесь к документу «Introduction to the SysOperation Framework» по адресу: *http://go.microsoft.com/fwlink/?LinkId=246316.*

В табл. 14-3 перечислены эти ошибки и описаны реализованные в классе *SysOpSampleBaseController* способы их решения.

Табл. 14–3. Ошибки и способы их обхода в *SysOperationServiceController*

Ошибка	**Код в *SysOpSampleBaseController***

При работе в режиме пакетного задания управляющий класс не должен распаковываться из таблицы *SysLastValue*	protected void loadFromSysLastValue()
	{
	  if (!dataContractsInitialized)
	  {
	  // Это ошибка в классе
	  // SysOperationController
	  // не следует загружать сохраненное
	  // в syslastvalue состояние при работе
	  // в пакетном задании. Это недопустимо.
	  // if (!this.isInBatch())
	  {
	  super();
	  }
	
	  dataContractsInitialized = true;
	  }
	}

Значение по умолчанию для свойства *parmRegisterCallbackForReliableAsyncCall* должно быть *false* во избежание ненужного опроса пакетного сервера	public void new()
	{
	  super();
	  // параметр по умолчанию для всех сценариев
	  // использования
	
	  // При использовании механизма надежных
	  // асинхронных вызовов не нужно ожидать
	  // завершения пакетного задания.
	  // Это лучше сделать на уровне приложения, т.к.
	  // момент изменения состояния при завершения
	  // пакетного задания непредсказуем
	  //
	  // this.parmRegisterCallbackForReliableAsyncCa
	  // ll(false);
	
	  ... код убран для простоты ...
	}

Табл. 14–3. Ошибки и способы их обхода в *SysOperationServiceController (окончание)*

Ошибка	**Код в *SysOpSampleBaseController***

Значение по умолчанию для свойства *parmExecutionMode* должно быть *Synchronous* во избежание ошибок во время создания исполняющихся задач	public void new()
	{
	... код убран для простоты ...
	
	// для управляющих классов в этих примерах
	// будет явно указано синхронное выполнение.
	// По умолчанию для SysOperationServiceController
	// указывается режим выполнения
	// ReliableAsynchronous.
	
	this.parmExecutionMode(SysOperationExecution Mode::Synchronous);
	}

Объявление класса *SysOpSampleBasicController* указывает, что он унаследован от базового класса инфраструктуры *SysOpSampleBaseController*, входящего в набор классов примера.

```
class SysOpSampleBasicController extends SysOpSampleBaseController
{
}
```

Метод *new* класса *SysOpSampleBasicController* определяет сам класс и способ действия обработки. В следующем примере метод *new* указывает на метод в управляющем классе. Но он может использовать любой метод класса. Инфраструктура использует отражение на этом классе и методе для создания пользовательского интерфейса и упаковки параметров.

```
void new()
{
    super();

    this.parmClassName(
        classStr(SysOpSampleBasicController));
    this.parmMethodName(
        methodStr(SysOpSampleBasicController,
            showTextInInfolog));

    this.parmDialogCaption(
        'Простой пример SysOperation');
}
```

В примере ниже метод *showTextInInfolog* печатает в окно **Infolog** входные параметры, уровень, на котором выполняется обработка, и среду ее выполнения.

```
public void showTextInInfolog(SysOpSampleBasicDataContract data)
{
    if (xSession::isCLRSession())
    {
        info('Выполняется в сессии CLR.');
    }
    else
    {
        info('Выполняется в интерпретируемой сессии.');
        if (isRunningOnServer())
        {
            info('Работает на AOS.');
        }
        else
        {
            info('Работает на клиенте.');
        }
    }

    info(strFmt('SysOpSampleBasicController: %1, %2', data.parmNumber(), data.parmText()));
}
```

Метод *caption* возвращает описание для обработки. Оно используется как значение по умолчанию для заголовка, показываемого в списке пакетных заданий и в интерфейсе пользователя.

```
public ClassDescription caption()
{
    return 'Простой пример SysOperation';
}
```

Метод *main* приглашает пользователя ввести параметры и запускает обработку или добавляет ее в очередь пакетных заданий.

```
public static void main(Args args)
{
    SysOpSampleBasicController operation;

    operation = new SysOpSampleBasicController();
    operation.startOperation();
}
```

Три переопределяемых метода в классе *SysOpSampleBasicDataContract* показаны в следующем примере. Инфраструктура использует атрибут контракта данных для отражения обработки в *class declaration*. Методы *parmNumber* и *parmText* используют атрибут члена данных для определения этих методов-свойств как частей контракта данных. Атрибуты метки, подсказки и порядка отображения дают информацию для построения интерфейса пользователя для обработки.

```
[DataContractAttribute]
class SysOpSampleBasicDataContract
{
    str text;
    int number;
}

[DataMemberAttribute,
SysOperationLabelAttribute('Числовое значение'),
SysOperationHelpTextAttribute('Введите некоторое число >= 0'),
SysOperationDisplayOrderAttribute('2')]
public int parmNumber(int _number = number)
{
    number = _number;

    return number;
}

[DataMemberAttribute,
SysOperationLabelAttribute('Текстовое значение'),
SysOperationHelpTextAttribute('Введите какой-нибудь текст'),
SysOperationDisplayOrderAttribute('1')]
public Description255 parmText(str _text = text)
{
    text = _text;

    return text;
}
```

Как и в примере **RunBase, щелкните Go на панели инструментов в редакторе кода**, находясь в методе *main* класса *SysOpSampleBasicController,* чтобы запустить наш пример обработки на базе **SysOperation, как показано** на рис. 14-10.

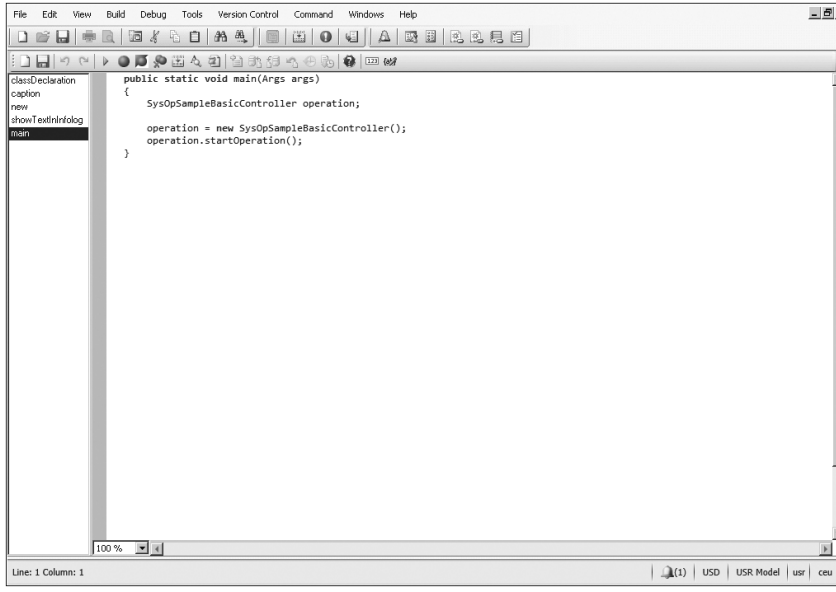

Рис. 14–10. Запуск примера SysOperation

Метод класса *main* вызывает *operation.startOperation*, который обрабатывает синхронный запуск обработки или постановку его в очередь пакетной обработки. Метод *startOperation* отображает пользовательский интерфейс обработки и затем вызывает *run*.

Для интерактивного запуска обработки заполните информацию на вкладке Разное пользовательского интерфейса обработки, как показано на рис. 14-11. Этот интерфейс, созданный инфраструктурой SysOperation, подобен созданному в примере RunBase.

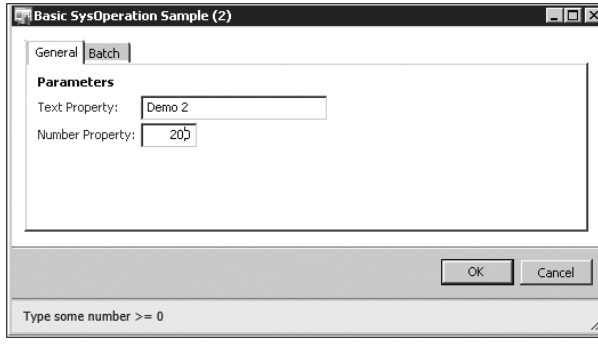

Рис. 14.11. Вкладка Общее примера использования инфраструктуры SysOperation

На вкладке Пакет убедитесь, что отметка Пакетная обработка снята, как показано на рис. 14-12.

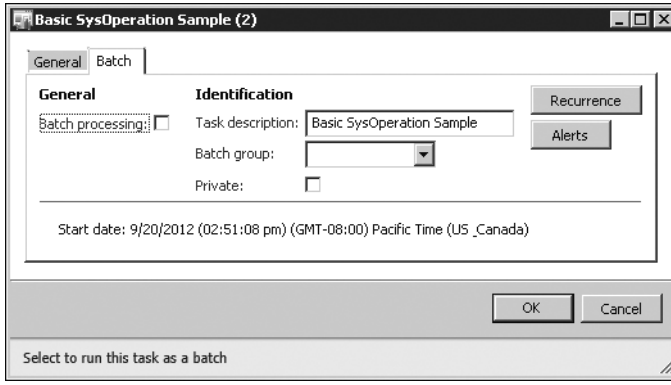

Рис. 14–12. Вкладка Пакет примера использования инфраструктуры SysOperation

Нажмите OK для запуска обработки и вывода в Infolog сообщений, показанных на рис. 14-13.

Рис. 14–13. Вывод в Infolog для примера SysOperation

Сообщения в Infolog показывают, что, в отличие от примера RunBase, обработка проводилась на сервере в сессии CLR.

Если вы повторите предыдущие шаги, но укажете отметку Пакетная обработка на вкладке Пакет, то обработка будет запущена на пакетном сервере, как и в примере с RunBase. (Не забывайте, что, возможно, вам нужно будет сделать инкрементную CIL-компиляцию обработки, чтобы

она смогла выполниться на пакетном сервере через меню Сборка > Создать инкрементный CIL. – *Прим. перев.*)

Ожидание начала обработки может занять до минуты, после чего откройте окно Описание задания из AOT.

Нажимайте F5, обновляя форму до тех пор, пока не увидите, что наше задание завершено. Сортировка по столбцу Запланированные дата/время начала поможет найти вашу обработку, если в форме содержится слишком много записей. Щелкнув на кнопке Журнал на нужной записи, вы можете открыть окно Infolog с выводом обработки.

На рис. 14-4 показано, что обработка шла в сессии CLR, что является средой выполнения по умолчанию для пакетного сервера.

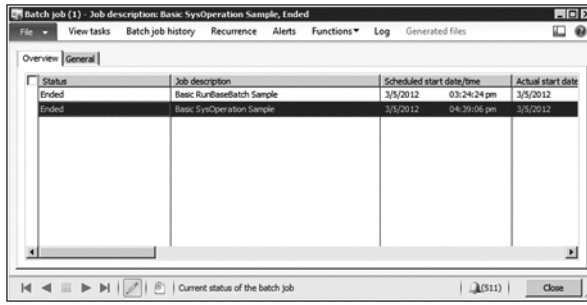

Рис. 14–14. Окно Infolog для примера SysOperation

Инфраструктура RunBase

Всюду в Microsoft Dynamics AX, там, где требуется организовать задачу обработки любой бизнес-операции, вы можете использовать инфраструктуру RunBase. Ее расширение позволяет реализовать бизнес-операцию, не существовавшую в системе ранее. Эта инфраструктура включает множество возможностей, такие как диалоговые окна, окна подтверждения начала обработки, индикаторы прогресса, оптимизацию клиент-серверного взаимодействия, упаковку/распаковку с поддержкой версионности и опциональную возможность пакетного выполнения в назначенную дату и время.

Примечание. Поскольку инфраструктура RunBase в основном была заменена фреймворком SysOperation, то изложение в последующих разделах предназначено больше для облегчения понимания того, как реализована уже существующая функциональность, использующая инфраструктуру RunBase.

Наследование в инфраструктуре RunBase

Классы, использующие инфраструктуру RunBase, должны наследоваться от класса *RunBase*, либо *RunBaseBatch*. Если класс наследует *RunBaseBatch*, он может быть использован в режиме пакетной обработки.

В хорошей модели наследования каждый класс имеет общедоступный механизм своего создания, кроме случая абстрактных классов. Если классу не требуется инициализация, то он должен иметь статический метод *construct*. Так как X++ не поддерживает перегрузку имен методов, вы должны использовать статический метод *new*, если класс должен быть проинициализирован перед созданием его экземпляра. Больше информации о конструкторах вы найдете в главе 4.

Статические методы *new* имеют следующие характеристики.

- Они общедоступны (public).
- Их имя начинается с *new*.
- Они имеют имя, логически вытекающее из их назначения или отвечающее получаемым аргументам. Примерами могут быть *newInventTrans* и *newInventMovement*.
- Обычно они не имеют параметров по умолчанию.
- Они всегда возвращают корректный объект типа класса, созданный и проинициализированный. В противном случае выбрасывается ошибка-исключение.

Примечание. Класс может иметь несколько методов *new* с разными наборами параметров. Класс *NumberSeq* может быть отличным примером класса со многими методами *new*.

Конструктор по умолчанию (метод *new*) должен быть защищенным (protected) для того, чтобы использующие класс применяли статический метод *construct* или методы *new*. Если в *new* присутствует какая-то допол-

нительная логика, которая должна выполняться каждый раз при инициализации класса, вынесите ее в отдельный метод *init*.

Совет. Чтобы облегчить последующую разработку, рекомендации по стилю кодирования советуют добавлять вновь создаваемую функциональность инициализации в виде новых подклассов (в вышележащих слоях), не смешивая код с методом *construct* в исходном слое.

Шаблон методов-свойств

Чтобы обеспечить работу остальной бизнес-логики приложения с вашей новой обработкой, может потребоваться запустить ее, не показывая никаких диалоговых окон пользователя. В случае такого решения, вам понадобится альтернативный способ задания параметров для переменных – членов класса вашей обработки.

В классах Microsoft Dynamics AX переменные – члены класса всегда имеют тип *protected*. Другими словами, они не могут быть доступны снаружи класса, а только внутри объектов этого класса или его подклассов. Для получения такого доступа к этим переменным внутри класса, вам придется написать методы доступа к ним. Эти методы могут получать, устанавливать или делать и то и другое со значениями переменных – членов класса. Имена таких методов доступа должны начинаться с *parm*. В Microsoft Dynamics AX методы доступа также часто называют *parm*-методами.

Совет. Рекомендации по стилю кодирования Microsoft Dynamics AX советуют не использовать отдельные методы доступа *get* и *set*. Эти методы объединяются в единый метод доступа, позволяющий одновременно получить и установить значение, путем использования шаблона проектирования *метод-свойство.* Методы доступа должны иметь то же имя, что и переменная, к которой они предоставляют доступ, но с префиксом *parm*.

Ниже приведен пример, как может выглядеть метод доступа, созданный по такому шаблону.

```
public NoYesId parmCreateServiceOrders(NoYesId _createServiceOrders =
createServiceOrders)
{
```

```
    createServiceOrders = _createServiceOrders;

    return createServiceOrders;
}
```

Если вы хотите, чтобы метод работал только как метод *get*, измените его примерно так:

```
public NoYesId parmCreateServiceOrders()
{
    return createServiceOrders;
}
```

Ну, а если вы хотите превратить его в метод *set*, сделайте так:

```
public void parmCreateServiceOrders(NoYesId _createServiceOrders = createServiceOrders)
{
    createServiceOrders = _createServiceOrders;
}
```

Когда переменные – члены класса содержат большие объемы данных (вроде больших контейнеров или **memo-полей), применяйте технику, по**казанную в примере ниже. Она определяет, изменился ли параметр. Обратной стороной такой техники являются добавочные накладные расходы при каждом вызове метода доступа.

```
public container parmCode(container _code = conNull())
{
    if (!prmIsDefault(_code)
    {
        code = _code;
    }

    return code;
}
```

Шаблон pack–unpack

Если вам требуется сохранить состояние объекта с возможностью его повторного пересоздания позже, используйте шаблон проектирования *pack-unpack*. Например, инфраструктура **RunBase требует от вас сле**дования этому шаблону для реализации переноса выполнения класса между клиентом и сервером и возможности отобразить в пользовательском диалоговом окне заданные при прошлом запуске значения. Если же ваша обработка расширяет класс *RunBaseBatch*, то этот шаблон

позволит также осуществить выполнение класса по расписанию в пакетном режиме.

Шаблон состоит из двух методов *pack* и *unpack*. Эти методы используются инфраструктурой SysLastValue, которая сохраняет и восстанавливает настройки пользователя или данные, однажды заданные им, в промежутках между выполняемыми им действиями.

Примечание. Восстановленный объект не является тем же самым объектом, который был сохранен. Это его копия, имеющая те же значения, что были упакованы и затем распакованы обратно.

Метод *pack* должен быть способен прочитать состояние объекта и вернуть его в контейнере. Чтение состояния объекта включает чтение значений его переменных, которые требуются для упаковки и распаковки объекта. Переменные – члены класса, использующиеся только во время выполнения (т.е. не требующие сохранять их состояние для восстановления состояния объекта), не нужно включать в метод *pack*. Первым элементом контейнера должно быть число, указывающее версию сохраненной структуры. Код ниже показывает пример метода *pack*.

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

В объявлении класса должны быть определено несколько макросов. *CurrentList* – это макрос, объявляемый в *ClassDeclaration* и перечисляющий список переменных для упаковки. Если этот список в макросе *CurrentList* будет изменен, то следует также изменить номер версии, чтобы обеспечить безопасную версионированную распаковку. Метод *unpack* может также поддерживать распаковку предыдущих версий класса, как показано в следующем примере.

```
class InventCostClosing extends RunBaseBatch
{
    #define.maxCommitCount(25)

    // Parameters

    TransDate               transDate;
    InventAdjustmentSpec    specification;
```

```
        NoYes                   prodJournal;
        NoYes                   updateLedger;
        NoYes                   cancelRecalculation;
        NoYes                   runRecalculation;
        FreeTxt                 freeTxt;
        Integer                 maxIterations;
        CostAmount              minTransferValue;
        InventAdjustmentType    adjustmentType;
        Boolean                 collapseGroups;
        ...

        #DEFINE.CurrentVersion(4)
        #LOCALMACRO.CurrentList
            TransDate,
            Specification,
            ProdJournal,
            UpdateLedger,
            FreeTxt,
            MaxIterations,
            MinTransferValue,
            adjustmentType,
            cancelRecalculation,
            runRecalculation,
            collapseGroups
        #ENDMACRO

    }
    public boolean unpack(container packedClass)
    {
        #LOCALMACRO.Version1List
            TransDate,
            Specification,
            ProdJournal,
            UpdateLedger,
            FreeTxt,
            MaxIterations,
            MinTransferValue,
            adjustmentType,
            del_minSettlePct,
            del_minSettleValue
        #ENDMACRO

        #LOCALMACRO.Version2List
            TransDate,
```

```
    Specification,
    ProdJournal,
    UpdateLedger,
    FreeTxt,
    MaxIterations,
    MinTransferValue,
    adjustmentType,
    del_minSettlePct,
    del_minSettleValue,
    cancelRecalculation,
    runRecalculation,
    collapseGroups
#ENDMACRO

Percent         del_minSettlePct;
CostAmount      del_minSettleValue;

Boolean         _ret;
Integer         _version = conpeek(packedClass,1);

switch (_version)
{
    case #CurrentVersion:
        [_version, #CurrentList]  = packedClass;
        _ret = true;
        break;

    case 3:
        // List has not changed, just the prodJournal must now always be updated
        [_version, #CurrentList]  = packedClass;
        prodJournal               = NoYes::Yes;
        updateLedger              = NoYes::Yes;
        _ret = true;
        break;

    case 2:
        [_version, #Version2List] = packedClass;
        prodJournal               = NoYes::Yes;
        updateLedger              = NoYes::Yes;
        _ret = true;
        break;

    case 1:
        [_version, #Version1List] = packedClass;
```

```
            cancelRecalculation     = NoYes::Yes;
            runRecalculation        = NoYes::No;
            _ret = true;
            break;

        default:
            _ret = false;
    }
    return _ret;
}
```

Если любая переменная – член класса не может быть упакована, то и сам класс не может быть упакован и восстановлен к исходному состоянию. Если его членами являются другие классы, записи, курсоры или временные таблицы, они также должны быть упаковываемыми. Другие классы, не унаследованные от *RunBase,* также могут реализовывать методы *pack* и *unpack* путем реализации интерфейса SysPackable.

Во время повторного создания объектов должна быть доступна возможность вызова метода *unpack,* читающего сохраненное состояние и присваивающего значения переменным – членам класса. Метод *unpack* может присваивать корректные значения переменным класса, основываясь на номере сохраненной версии, как показано в примере ниже.

```
public boolean unpack(container _packedClass)
{
    Version version = conpeek(_packedClass, 1);

    switch (version)
    {
        case #CurrentVersion:
            [version, #CurrentList] = _packedClass;
            break;

        default:
            return false;
    }
    return true;
}
```

Метод *unpack* возвращает Boolean-значение, указывающее, была ли повторная инициализация успешной.

Как говорилось ранее в этом разделе, методы *pack* и *unpack* отвечают за три аспекта.

- Перенос класса-наследника *RunBase* между клиентом и сервером.
- Вывод выбранных в предыдущем запуске класса значений параметров.
- Выполнение класса по расписанию в пакетном режиме.

В некоторых сценариях бывает необходимо выполнить некоторую логику, зависящую от контекста, в котором были вызваны методы *pack* или *unpack*. Для этого можно использовать метод *isSwappingPrompt* объекта *RunBase* для определения, был ли вызов *pack* или *unpack* сделан в контексте переноса между клиентом и сервером. Метод *isSwappingPrompt* возвращает *true* при вызове в рамках такого контекста. Кроме того, существует метод *isInBatch* на *RunBaseBatch* для определения, вызван ли метод *unpack* в контексте выполнения класса в пакетном режиме.

О взаимодействии клиент/сервер

На практике обычно требуется выполнять обработки данных на уровне сервера приложений, так как они почти всегда состоят из нескольких транзакций базы данных. Однако при этом отображать пользовательское диалоговое окно (для минимизации вызовов между клиентом и сервером со стороны сервера) лучше на клиенте. К счастью, инфраструктуры, как *SysOperation*, так и *RunBase,* могут помочь вам запустить диалоговое окно на клиенте, а основную обработку провести на сервере.

Чтобы сделать это, нужно обратить внимание на два параметра. На пункте меню, вызывающем обработку, следует установить свойство *RunOn* в *Server*; а в свойствах самого класса – свойство *RunOn* в *Called From*. Об этих свойствах подробнее можно прочитать в разделе «Написание кода, который учитывает нюансы клиент-серверного взаимодействия» в главе 13.

Когда такая обработка будет запущена, она начнется на сервере, а фреймворк RunBase упакует ее внутренние переменные и создаст новый экземпляр класса на клиенте. Тот распакует сохраненные внутренние переменные и отобразит диалоговое окно. После нажатия пользователем кнопки OK в диалоговом окне, RunBase упакует переменные экземпляра на клиентской стороне и снова распакует в серверном экземпляре класса.

Инфраструктура расширений

Инфраструктура расширений – это шаблон расширения функциональности, уменьшающий или вовсе устраняющий связывание между ком-

понентами приложения и их расширениями. Многие основополагающие фреймворки приложения написаны с использованием инфраструктуры расширений. Эта инфраструктура использует системы классов-атрибутов и фабрик классов для разделения базовых и производных классов в два приема.

Создание расширения

Сначала создайте метод класса-атрибута, унаследовав его от класса *SysAttribute*.

Пример такого класса можно найти в модуле Управление сведениями о продукте в классе *PCAdaptorExtensionAttribute*.

```
class PCAdaptorExtensionAttribute extends SysAttribute
{
    PCName modelName;

    public void new(PCName _modelName)
    {
        super();
        if (_modelName == '')
        {
            throw error(Error::missingParameter(this));
        }
        modelName = _modelName;
    }

    public PCName parmModelName(PCName _modelName = modelName)
    {
        modelName = _modelName;
        return modelName;
    }
}
```

Далее, используйте класс-атрибут *PCAdaptorExtensionAttribute,* чтобы добавить метаданные в производный класс.

В примере ниже наследник класса *PCAdaptor* расширяет его, создавая объект *MyPCAdaptor* вместо объекта *PCAdaptor*, при обработке конфигурации продукта, созданного по модели конфигурации продукта *Computers*:

```
[PCAdaptorExtensionAttribute('Computers')]
class MyPCAdaptor extends PCAdaptor
{
    protected void new()
```

```
    {
        super();
    }
}
```

Вы можете протестировать это расширение, пройдя следующие шаги.

1. Нажмите Ctrl+Shift+W, чтобы открыть рабочую область разработчика.
2. Добавьте класс *MyPCAdaptor* в AOT и скомпилируйте его.
3. Поставьте точку останова в методе *PCAdaptorFactory.getAdaptorFromModelName*.
4. В Windows-клиенте Microsoft Dynamics AX выберите Управление сведениями о продукте > Обычный > Модели конфигурации продукта.
5. На панели действий в группе Создать щелкните Модель конфигурации продукта. Появится диалоговое окно Новая конфигурация модели продукта.
6. В поле Название введите **Computers**.
7. Введите название в поле Создать компонент – Название и нажмите OK. Откроется окно Сведения о модели конфигурации продукта на основе ограничений.
8. В секции Атрибуты формы добавьте атрибут в корневой компонент, например атрибуты размера и цвета.
9. На панели действий в группе Запуск щелкните Проверить.
10. Выберите значение для атрибута.

Запустится отладчик Microsoft Dynamics AX, стоящий на точке останова, установленной вами на шаге 3.

По мере прохождения кода вы увидите, как фабрика *SysExtensionAppClassFactory* используется для создания экземпляра класса *MyPCAdaptor*:

```
adaptor = SysExtensionAppClassFactory::getClassFromSysAttribute(
    classStr(PCAdaptor), extensionAttribute);
```

Работа метода *getClassFromSysAttribute* заключается в поиске по классам, унаследованным от PCAdaptor. Он возвращает экземпляр найденного класса, имеющего атрибут *PCAdaptorExtensionAttribute*, который возвращает название модели, совпадающее с названием модели, переданным в метод поиска. В этом случае будет создан экземпляр для модели конфигурации продукта с названием *Computers*.

Создаваемый вами код может получить определенные преимущества от использования этой модели расширения за счет разделения базовых и производных классов и сокращения количества кода, нужного для увеличения возможностей Microsoft Dynamics AX.

Пример расширения

Ниже показан законченный пример написания расширяемых классов. Также здесь представлено несколько примеров расширений.

Вначале нужно создать класс, унаследованный от *SysAttribute* с названием *CalendarExtensionAttribute*, чтобы затем с его помощью помечать классы как расширяемые.

```
public class CalendarExtensionAttribute extends SysAttribute
{
    str calendarType;
}

public void new(str _calendarType)
{
    super();
    if (_calendarType == '')
    {
        throw error(error::missingParameter(this));
    }
    calendarType = _calendarType;
}

public str parmCalendarType(str _calendarType = calendarType)
{
    calendarType = _calendarType;
    return calendarType;
}
```

Затем используйте вновь созданный класс-атрибут для добавления метаданных в расширяемый класс *Calendar* и его производные классы:

```
[CalendarExtensionAttribute(«Default»)]
public class Calendar
{
}

public void new()
{
```

```
}

public void sayIt()
{
    info("All days are work days except for weekends!");
}
```

Следующий код иллюстрирует два примера расширений – *Financial-Calendar* и *HolidayCalendar*. Оба класса переопределяют метод *sayIt*.

```
[CalendarExtensionAttribute("Financial")]
public class FinancialCalendar extends Calendar
{
}

public void sayIt()
{
    super();
    info("Financial Statements are available on the last working day of June!");
}
[CalendarExtensionAttribute("Holiday")]
public class HolidayCalendar extends Calendar
{
}

public void sayIt()
{
    super();
    info( "Eight public holidays including New Year's Day!");
}
```

И, наконец, создадим новый класс-фабрику для генерации необходимых экземпляров класса *Calendar*. Эта фабрика использует метод *SysExtensionAppClassFactory.getClassFromSysAttribute,* который производит поиск среди наследников класса *Calendar* совпадения параметров метаданных его атрибута с параметрами его вызова. Приведенный ниже код показывает класс *CalendarFactory,* создающий экземпляр календаря.

```
public class CalendarFactory
{
}

public static Calendar instance(str _calendarType)
{
```

```
    CalendarExtensionAttribute extensionAttribute =
        new CalendarExtensionAttribute(_calendarType);
    Calendar calendar =
        SysExtensionAppClassFactory::getClassFromSysAttribute(classStr(calendar),
extensionAttribute);

    if (calendar == null)
    {
        calendar = new Calendar();
    }

    return calendar;
}
```

Здесь приведен код задания (job), содержащий возможные сценарии создания календаря.

```
static void CreateCalendarsJob(Args _args)
{
    Calendar calendar = CalendarFactory::instance("Holiday");
    calendar.sayIt();
    calendar = CalendarFactory::instance("Financial");
    calendar.sayIt();
    calendar = CalendarFactory::instance("Default");
    calendar.sayIt();
}
```

Обработка событий

Обработка событий – это еще один шаблон расширения функциональности, уменьшающий число или устраняющий зависимости между компонентами приложения и их расширениями. В отличие от инфраструктуры расширений, оперирующей довольно большими порциями бизнес-логики, обработка событий может быть более тонким инструментом. С ее помощью вы сможете эффективно дополнить или изменить существующее поведение приложения.

Следующие понятия относятся к событиям в X++.

- **Источник.** Логика, содержащая код, производящий изменения в данных. Это сущность, порождающая события.
- **Потребитель.** Код приложения, выражающий запрос на получение уведомления о конкретном состоявшемся событии. Это сущность, получающая события.

- **Событие.** Представляет изменение, произошедшее в источнике.
- **Полезная нагрузка события.** Информация, которую несет событие. Например, при найме нового сотрудника это может быть его имя и дата рождения.
- **Делегат.** Задает определение информации, которая передается от источника к потребителю, когда происходит событие.

Используя события, вы потенциально можете снизить стоимость создания и доработки ваших решений. Если вы создаете код, который затем часто изменяется кем-либо, то можете создать события в тех местах, где обычно делаются изменения. Тогда разработчик, делающий доработку функциональности на другом слое, может подписаться на получение событий. И когда реализованная функциональность связана с событием, нижележащий код приложения может быть переписан с минимальным влиянием на эту функциональность до тех пор, пока от версии к версии одни и те же события происходят в том же самом порядке. Использование событий дает возможность следовать следующим парадигмам программирования.

- **Наблюдение.** События могут отслеживать возникновение исключительных ситуаций и оповещать о них сразу после их возникновения. Например, такой тип событий может быть использован в системе отслеживания за соблюдением некоторых правил. И если со счета на счет переводится больше оговоренной суммы, будет сгенерировано событие, так что его обработчики смогут адекватно среагировать на него. К примеру, они могут отклонить транзакцию и оповестить управляющего этим счетом.
- **Распространение информации.** События могут доносить нужную информацию нужным потребителям точно в нужное время. Эта информация может быть распространена путем опубликования события всем, кто пожелает на него реагировать. Например, при приеме на работу нового работника, сотрудникам отдела кадров может быть необходимо провести его ознакомление с организацией.
- **Разделение.** События, порожденные в одной части приложения, могут быть обработаны другой его частью. Источник события не обязан знать о своих потребителях, и наоборот. Одно события источника может быть обработано множеством потребителей и, напротив, потребители могут реагировать на любое число событий от многих ис-

точников. Так, создание нового работника в системе может быть обработано модулями Управления по Проектам и Главной книгой, если вы хотите сразу добавлять его в проектные команды по умолчанию.

События Microsoft Dynamics AX основаны на концепциях событий в .NET. Больше информации об этом можно узнать в статье «X++, C# Comparison: Event» по адресу: *http://msdn.microsoft.com/en-us/library/gg881685(v=ax.60).aspx*.

Делегаты

В X++ можно добавлять делегатов как членов классов. Синтаксис, используемый для этого, такой же, что и синтаксис определения метода, со следующими исключениями.

- Используется ключевое слово *delegate*.
- В объявлении делегатов не может быть использовано модификаторов доступа, так как все они являются защищенными (protected) членами класса.
- Возвращаемый тип должен быть *void*.
- Тело должно быть пусто, то есть оно не может содержать ни объявлений, ни операторов.
- Делегат может быть объявлен только как член класса. Он не может быть членом таблицы.

Например, делегат для события, генерирующегося при приеме на работу нового сотрудника, может выглядеть так:

```
delegate void hired(str personnelNumber, UtcDateTime startingDate)
{
    // Делегаты не имеют никакого кода в своем теле
}
```

Параметры, определенные в списке параметров, могут быть любого типа, допустимого в X++. В частности, удобно передавать в них экземпляр объекта и изменять в обработчике события его состояние. При таком способе использования источники могут получать значения от подписчиков.

События *pre* и *post*

События *pre* и *post* – это предопределенные события, происходящие при вызове методов. Обработчики событий *pre* вызываются перед началом вы-

полнения указываемого метода, а обработчики события *post* – после завершения его выполнения. Вполне можно думать об этих обработчиках, как о дополнениях существующего метода новыми методами, вызывающимися перед ним и после него. Обработчики этих сообщений *pre* и *post* видны в AOT как подузлы методов, к которым они относятся.

Следующий псевдокод иллюстрирует метод без обработчиков событий:

```
void someMethod(int i)
{
    --тело метода --
}
```

А следующий пример – тот же метод после добавления обработчиков событий:

```
void someMethod(int i)
{
    preHandler1(i);
    preHandler2(i);
    --тело метода–
    postHandler1(i);
    postHandler2(i);
}
```

Отсутствие обработчиков событий для какого-либо метода оставляет его нетронутым. Следовательно, для методов без каких-либо обработчиков событий *pre* и *post* отсутствуют накладные расходы.

Если в обработчике события *pre* произойдет исключение, то не произойдет вызова ни оставшихся обработчиков событий, ни самого метода. Если метод, имеющий любые обработчики событий *pre* или *post,* бросит исключение, то оставшиеся обработчики событий *post* также не будут вызваны. То же относится и к выбрасыванию исключений в обработчике события *post* – оставшиеся обработчики событий не вызовутся.

Каждый обработчик события *pre* может получить доступ к исходным значениям параметров и изменить их в нужную сторону. Обработчик события *post* может изменить возвращаемое значение метода.

Обработчики событий

Обработчики событий – это методы, вызывающиеся в момент вызова делегатов либо напрямую через код (или закодированными событиями),

либо из среды исполнения (для созданных событий, находящихся в AOT). Связи между делегатом и обработчиками событий могут поддерживаться как в коде, так и через AOT.

Для декларативного задания обработчика события в AOT вы должны создать статический метод для обработки сообщений от делегата и затем просто перетащить его на узел делегата, представляющего обрабатываемое событие. Можно также удалить обработчик события, используя пункт меню Удалить, доступный для любых узлов в AOT. В таком контексте вы можете использовать только статические методы.

Чтобы создать в коде статический обработчик события, требуется использовать специальный синтаксис X++, как показано в следующем примере.

```
void someMethod(int i)
{
    this.MyDelegate += eventhandler(Subscriber::MyStaticHandler);
}
```

Имя делегата появляется слева от оператора +=. Справа вы можете видеть ключевое слово *eventhandler*, вместе с указанным именем добавляемого обработчика. Компилятор проверяет профили параметров делегата и обработчика на совпадение. Указанное имя обработчика в примере использует двойное двоеточие (::) для разделения имени типа и делегата, что указывает на статический обработчик события.

Для того чтобы вызвать метод конкретного экземпляра объекта, используйте синтаксис из следующего примера.

```
void someMethod(int i)
{
    EventSubscriber subscriber = new EventSubscriber();
    this.MyDelegate += eventhandler(subscriber.MyInstanceHandler);
}
```

Можно убрать (отсоединить) обработчик события от делегата, использовав оператор -= *вместо оператора* +=. Пример отсоединения статического обработчика приведен ниже.

```
void someMethod(int i)
{
    this.MyDelegate -= eventhandler(Subscriber::MyHandler);
}
```

О событиях стоит помнить следующие вещи:

- компилятор X++ **не позволит вам породить событие вне класса, в котором** определен делегат;
- среда выполнения не гарантирует порядок вызова обработчиков события;
- обработчики событий могут быть реализованы как на управляемом коде, так и на X++. Обработчики на управляемом коде создаются в проекте Microsoft Visual Studio, добавляемом в AOT.

Пример обработки событий

Приводимый здесь пример иллюстрирует использование как воплощенных в коде обработчиков событий, так и смоделированных в AOT. Пример также показывает способы передачи аргументов в обработчики событий.

Код представляет собой массив с событиями, возникающими при изменении элемента.

```
public class arrayWithChangedEvent extends Array
{
}

delegate void changedDelegate(int _index, anytype _value)
{
}

public anytype value(int _index, anytype _value = null)
{
    anytype paramValue = _value;
    anytype val = super(_index, _value);
    boolean newValue = (paramValue == val);
    if (newValue)
        this.changedDelegate(_index, _value);

    return val;
}
```

Следующий динамический обработчик добавляется во время выполнения.

```
public class arrayChangedEventListener
{
    arrayWithChangedEvent arrayWithEvent;
}
```

```
public void new(ArrayWithChangedEvent _arrayWithEvent)
{
    arrayWithEvent = _arrayWithEvent;

    // Регистрируем обработчик события в делегате
    arrayWithEvent.ChangedDelegate += eventhandler(this.ListenToArrayChanges);
}

public void listenToArrayChanges(int _index, anytype _value)
{
    info(strFmt("Array changed at: %1 - with value: %2", _index, _value));
}

public void detach()
{
    // Отсоединяет обработчик от делегата
    arrayWithEvent.changedDelegate -= eventhandler(this.listenToArrayChanges);
}
```

Этот пример состоит из двух статических обработчиков.

```
public static void ArrayPreHandler(XppPrePostArgs args)
{
    int indexer = args.getArg("_index");
    str strVal = "";
    if (args.existsArg("_value") && typeOf(args.getArg("_value")) == Types::String)
    {
        strVal = "Pre-" + args.getArg("_value"); // Отметим значение как  pre-обработанное
        args.setArg("_value", strVal);
        // Изменения в значениях параметров могут основываться на
        // состоянии записи или переменных среды
    }

}
public static void ArrayPostHandler(XppPrePostArgs args)
{
    anytype returnValue = args.getReturnValue();
    str strReturnValue = "";

    if (typeOf(returnValue) == Types::String)
    {
        strReturnValue = returnValue + "-Post"; // Отмечаем возвращаемое значение как
post- обработанное
        args.setReturnValue(strReturnValue);
```

```
    }
}
```

Чтобы проверить пример обработки событий, добавим обработчики событий *pre* и *post* к методу *value* класса *ArrayWithChangedEvent* в AOT и запустим следующее задание.

```
static void EventingJob(Args _args)
{
    // Создадим массив
    ArrayWithChangedEvent arrayWithEvent = new ArrayWithChangedEvent(Types::String);

    // Создадим слушатель для массива
    ArrayChangedEventListener listener = new ArrayChangedEventListener(arrayWithEvent);

    // Проверим добавление элементов в массив
    info(arrayWithEvent.value(1, "Blue"));
    info(arrayWithEvent.value(2, "Cerulean"));
    info(arrayWithEvent.value(3, "Green"));

    // Отключим слушатель от массива
    listener.Detach();

    // Последующие вставки в массив не должны вызвать слушателя,
    // за исключением появления событий pre и post
    info(arrayWithEvent.value(4, "Orange"));
    info(arrayWithEvent.value(5, "Pink"));
    info(arrayWithEvent.value(6, "Yellow"));
}
```