Microsoft Dynamics® AX 2012

Using the refactored formletter framework

The formletter framework in Microsoft Dynamics AX is used when posting documents for sales or purchase orders. The framework has been refactored in Microsoft Dynamics AX 2012 to provide better support for customizations, extensibility, and performance. This document describes the changes to the framework and how to update your customizations to take advantage of them.

Date: May 2011

http://microsoft.com/dynamics/ax

Author: Kenneth Puggaard, Senior Development Lead

Send suggestions and comments about this document to adocs@microsoft.com. Please include the white paper title with your feedback.

Table of Contents

Introduction

The formletter framework in Microsoft Dynamics® AX is used when posting documents for sales or purchase orders. This framework has been refactored in Microsoft Dynamics AX 2012 to provide better support for customizations, extensibility, and performance. This document describes the Microsoft Dynamics AX 2012 version of the framework.

Background

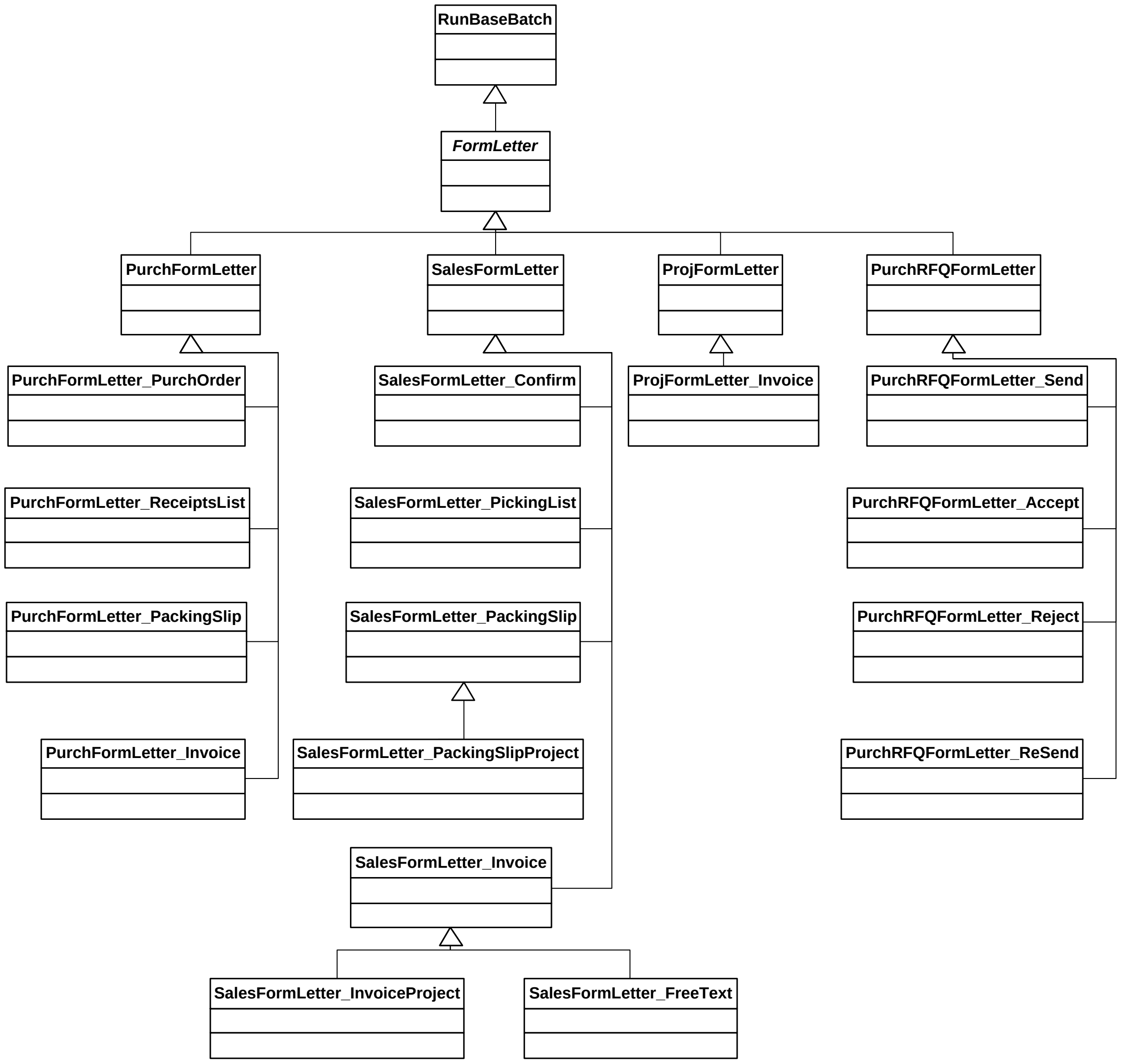
In Microsoft Dynamics AX 2009, the formletter framework consisted of the following objects:

- The **FormLetter** base class that extended **RunBaseBatch**.
- Module-specific classes that extended the **FormLetter** class, such as **SalesFormLetter**.
- A number of document-specific classes that extended the module class, such as **SalesFormLetter_PackingSlip**.

This set of classes had a number of functions, including the following:

- Interaction with the posting form, such as SalesEditLines.
- Creation and maintenance of posting data, such as records in SalesParmTable.
- Creation of journal data, such as records in CustPackingSlipJour/CustPackingSlipTrans.
- Validations.
- Updating subledgers, such as ledger and inventory.
- Controlling document outputs, such as printing and XML export.
- Client/server marshaling.

Figure 1 shows the formletter framework in Microsoft Dynamics AX 2009.



RunBaseBatch
FormLetter
PurchFormLetter
SalesFormLetter
ProjFormLetter
PurchRFQFormLetter
PurchFormLetter_PurchOrder
SalesFormLetter_Confirm
ProjFormLetter_Invoice
PurchRFQFormLetter_Send
PurchFormLetter_ReceiptsList
SalesFormLetter_PickingList
PurchRFQFormLetter_Accept
PurchFormLetter_PackingSlip
SalesFormLetter_PackingSlip
PurchRFQFormLetter_Reject
PurchFormLetter_Invoice
SalesFormLetter_PackingSlipProject
PurchRFQFormLetter_ReSend
SalesFormLetter_Invoice
SalesFormLetter_InvoiceProject
SalesFormLetter_FreeText

Figure 1: Class diagram: Formletter framework in Microsoft Dynamics AX 2009

Because the **FormLetter** classes had so much functionality and no well-defined APIs in Microsoft Dynamics AX 2009, they were complex for developers to understand and customize.

The formletter framework in Microsoft Dynamics AX 2012

In Microsoft Dynamics AX 2012, the formletter framework has been refactored to clearly separate the functionality needed for the posting process into different classes. This has led to a number of new class hierarchies. The following class diagram compares the new set of base classes in Microsoft Dynamics AX 2012 to Microsoft Dynamics AX 2009 and shows how the classes interact with each other in Microsoft Dynamics AX 2012.

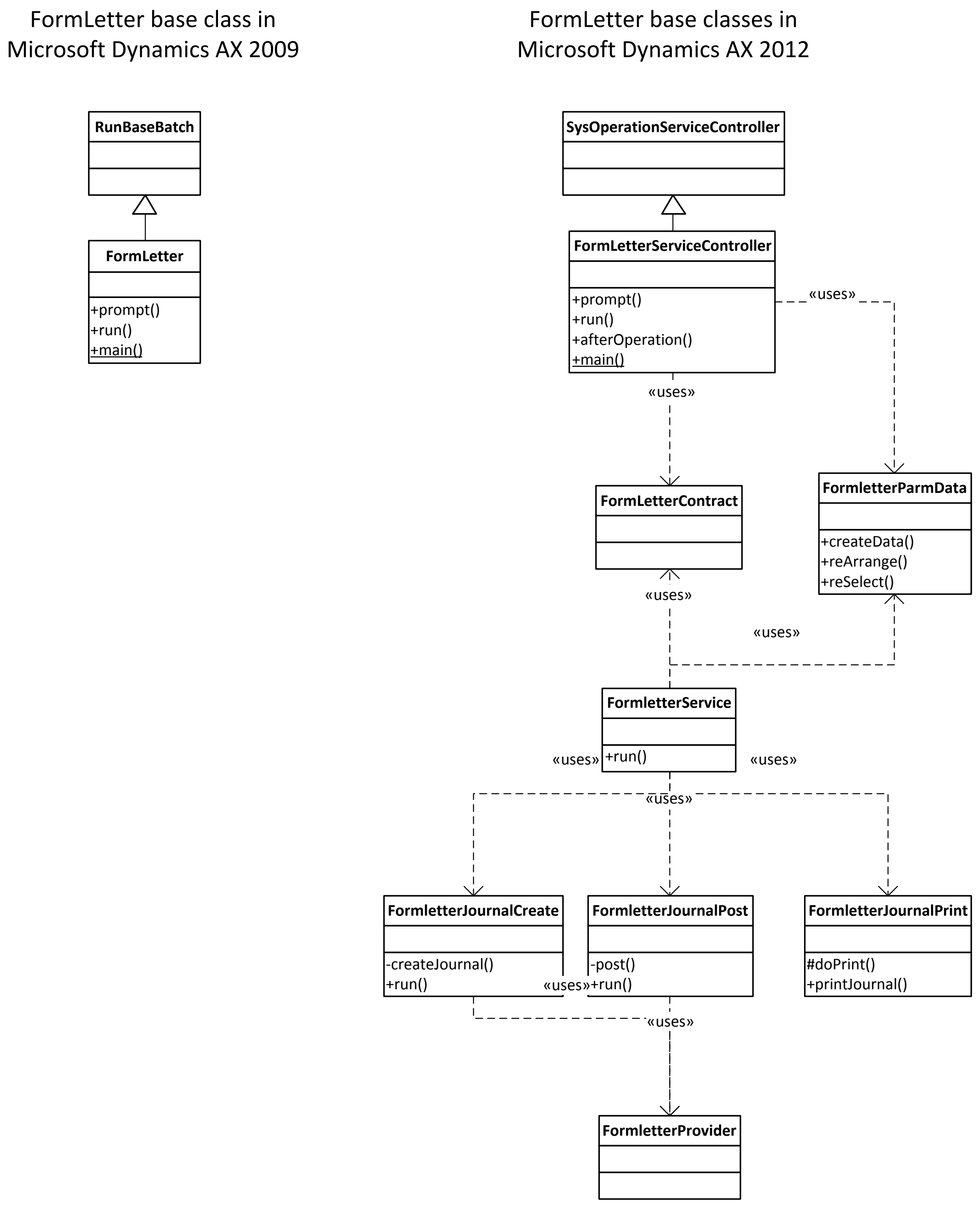
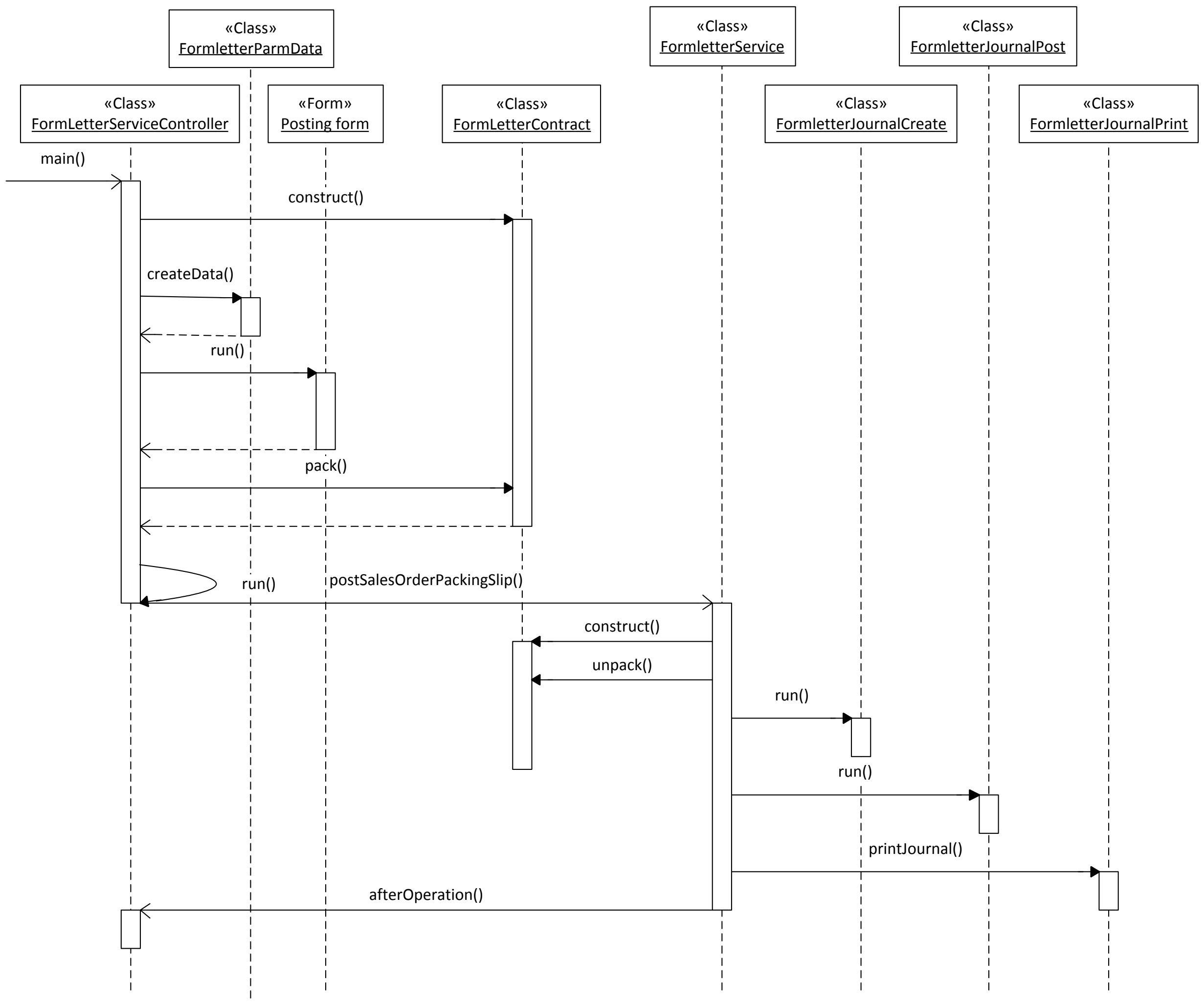


Figure 2: Class diagram: Formletter framework base classes

The Microsoft Dynamics AX 2009 base **FormLetter** class has been split into eight base classes. It also has been changed to run under the SysOperation framework. Switching to the SysOperation framework has the advantage of executing the code on the server tier during posting in IL (intermediate language). Because of the switch to using the SysOperation framework, client callbacks from code running on the server tier are no longer supported. Client callbacks result in an exception.

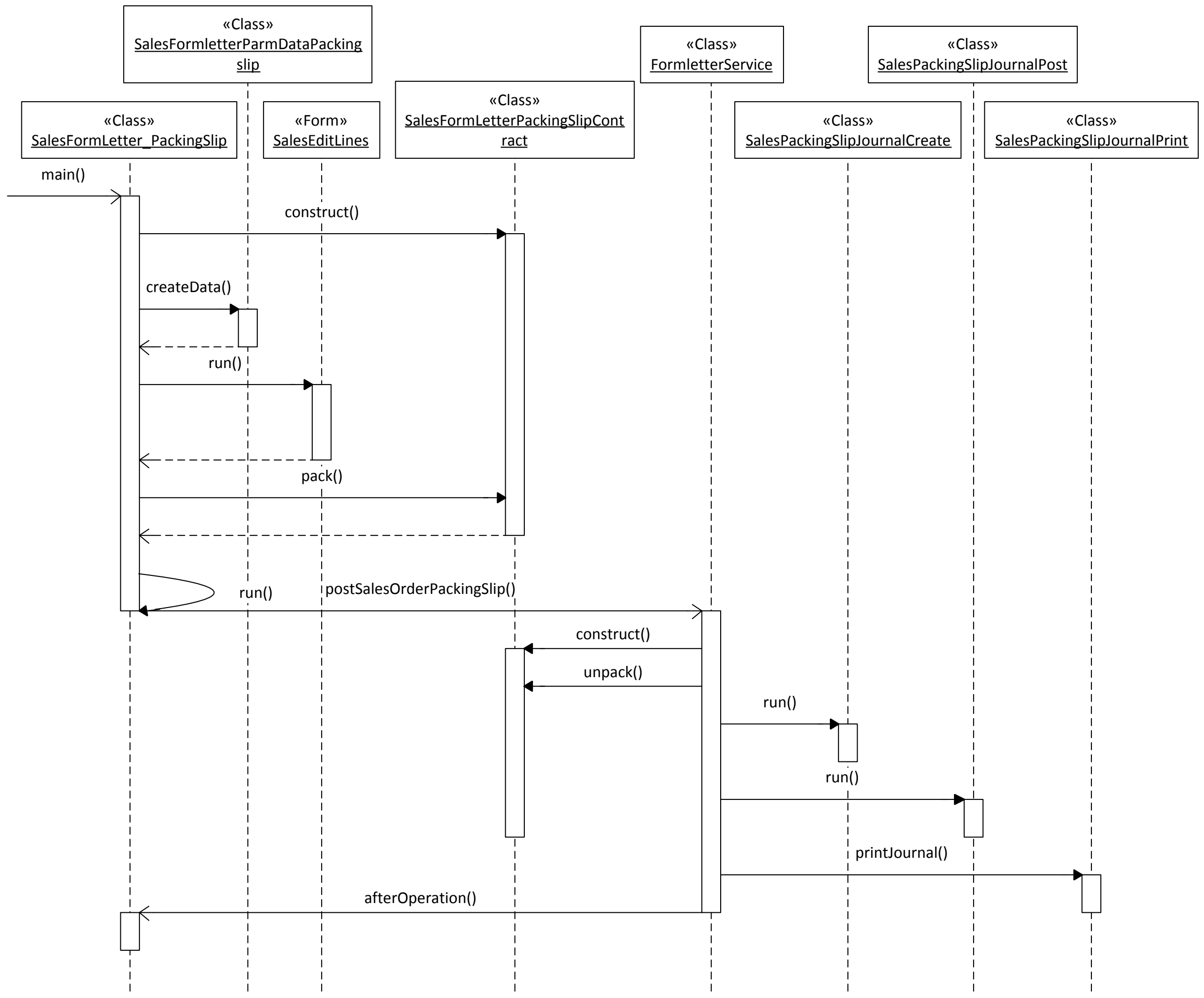
Figure 3 shows how the various base classes are used when posting a document for an order.



«Class» FormletterParmData
«Class» FormletterService
«Class» FormletterJournalPost
«Class» FormLetterServiceController
«Form» Posting form
«Class» FormLetterContract
«Class» FormletterJournalCreate
«Class» FormletterJournalPrint
main()
construct()
createData()
run()
pack()
run()
postSalesOrderPackingSlip()
construct()
unpack()
run()
run()
printJournal()
afterOperation()

Figure 3: Sequence diagram: Posting an order document, base class view.

Figure 4 shows how the various class hierarchies are used when posting a sales order packing slip document.



«Class» SalesFormletterParmDataPackingslip
«Class» FormletterService
«Class» SalesPackingSlipJournalPost
«Class» SalesFormLetterPackingSlipContract
«Class» SalesFormLetter_PackingSlip
«Form» SalesEditLines
«Class» SalesPackingSlipJournalCreate
«Class» SalesPackingSlipJournalPrint
main()
construct()
createData()
run()
pack()
run()
postSalesOrderPackingSlip()
construct()
unpack()
run()
run()
printJournal()
afterOperation()

Figure 4: Sequence diagram: Posting a sales order packing slip document

The FormLetterServiceController class

The **FormLetterServiceController** class has replaced the Microsoft Dynamics AX 2009 **FormLetter** class. The class is the entry point for the formletter framework and can be used to interact with the posting form (for example, the SalesEditLines form). The **FormLetterServiceController** class is executed on the client tier. The class gathers the information needed during the posting process and passes these values to the **FormletterService** class using the data contract class pattern. The **FormLetterServiceController** invokes the **FormletterService** class when the run method is called.

Figure 5 illustrates the **FormLetterServiceController** class hierarchy.

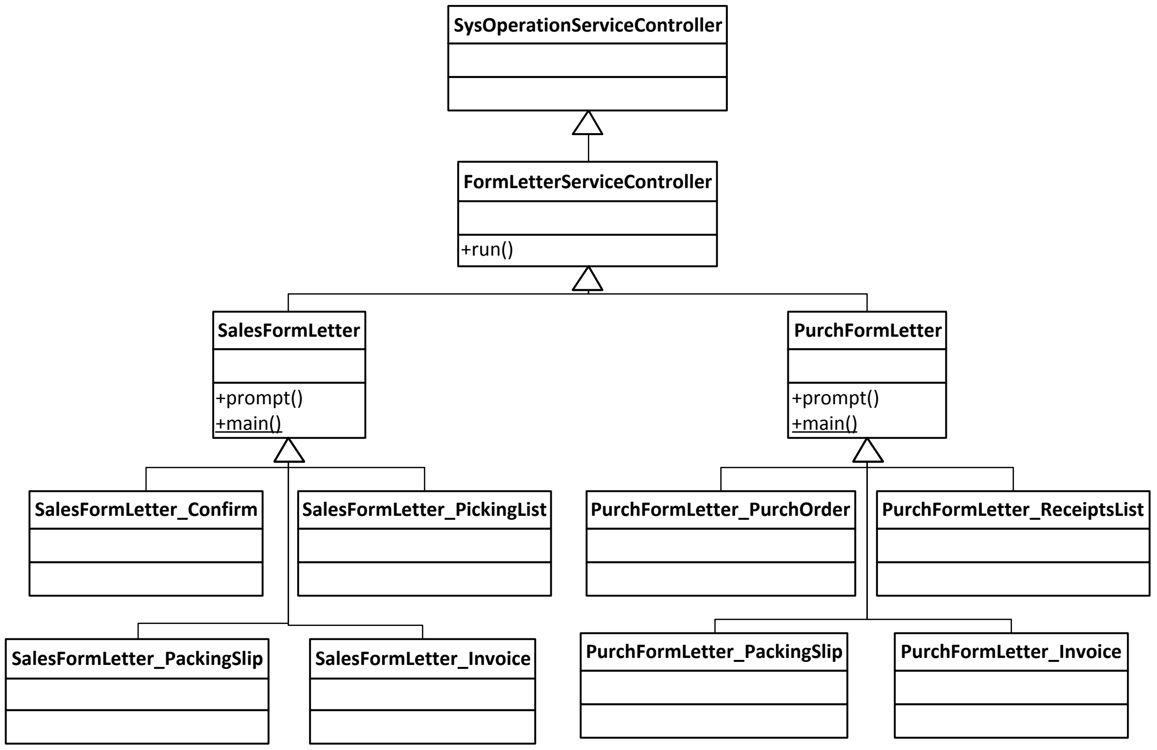

SysOperationServiceController
FormLetterServiceController
+run()
SalesFormLetter
+prompt()
+main()
PurchFormLetter
+prompt()
+main()
SalesFormLetter_Confirm
SalesFormLetter_PickingList
PurchFormLetter_PurchOrder
PurchFormLetter_ReceiptsList
SalesFormLetter_PackingSlip
SalesFormLetter_Invoice
PurchFormLetter_PackingSlip
PurchFormLetter_Invoice

Figure 5: Class diagram: FormLetterServiceController class hierarchy

The module-specific classes from Microsoft Dynamics AX 2009, such as **SalesFormLetter**, have been changed so that they now extend the **FormLetterServiceController** class. All functionality that does not relate to the interaction with the posting form has been moved to other class hierarchies. The class variables that were assigned a value to be used during the posting process have been moved to the data contract classes. The pattern used for assigning values to the data contract classes is that the **parm** methods from Microsoft Dynamics AX 2009 have been changed to utilize the data contract class instead of using a global class variable. The following code sample is an example of a **parm** method:

```
public boolean parmDirectDeliveryUpdate(boolean _directDeliveryUpdate =
salesFormletterContract.parmDirectDeliveryUpdate())
{
    return salesFormletterContract.parmDirectDeliveryUpdate(_directDeliveryUpdate);
}
```

The FormLetterContract class

The **FormLetterContract** class is the base data contract class in the formletter framework. The data contracts follow the same class hierarchy structure as the FormletterServiceController classes. Therefore, each class in the **FormLetterServiceController** class hierarchy has a corresponding data contract class. The data contract classes pass data from the **FormLetterServiceController** to the **FormletterService**.

Figure 6 illustrates the **FormLetterContract** class hierarchy.

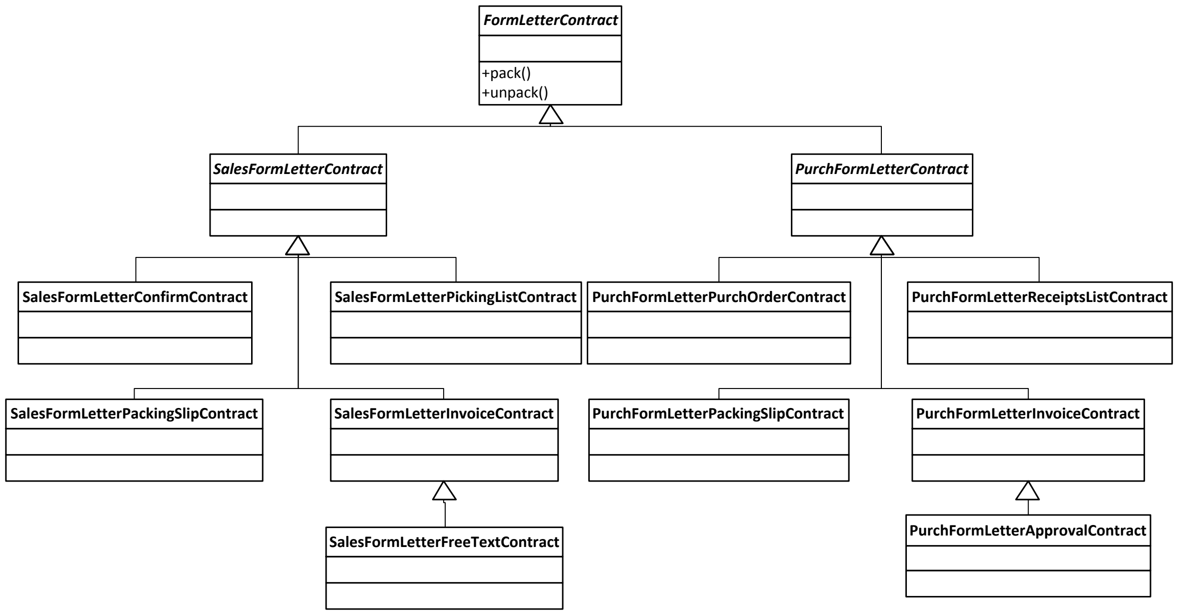

FormLetterContract
+pack()
+unpack()
SalesFormLetterContract
PurchFormLetterContract
SalesFormLetterConfirmContract
SalesFormLetterPickingListContract
PurchFormLetterPurchOrderContract
PurchFormLetterReceiptsListContract
SalesFormLetterPackingSlipContract
SalesFormLetterInvoiceContract
PurchFormLetterPackingSlipContract
PurchFormLetterInvoiceContract
SalesFormLetterFreeTextContract
PurchFormLetterApprovalContract

Figure 6: Class diagram: FormLetterContract class hierarchy

Parm methods are used to set and get the data from a contract class. The following code sample is an example of a **parm** method:

```
[DataMemberAttribute]
public NoYes parmDirectDeliveryUpdate(NoYes _directDeliveryUpdate = directDeliveryUpdate)
{
    directDeliveryUpdate = _directDeliveryUpdate;
    return directDeliveryUpdate;
}
```

The FormletterService class

The **FormletterService** class can be used to control the posting flow for a number of journals. The class is bound to the server tier and executes in IL, which means that there can be no client callbacks from this class. Client callbacks result in an exception. Figure 7 illustrates the **FormletterService** class hierarchy.

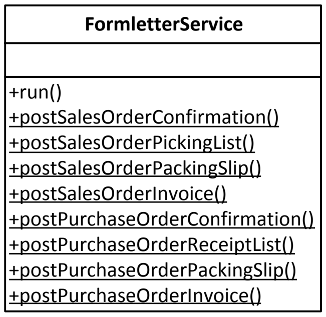

FormletterService
+run()
+postSalesOrderConfirmation()
+postSalesOrderPickingList()
+postSalesOrderPackingSlip()
+postSalesOrderInvoice()
+postPurchaseOrderConfirmation()
+postPurchaseOrderReceiptList()
+postPurchaseOrderPackingSlip()
+postPurchaseOrderInvoice()

Figure 7: Class diagram: FormletterService

The **run** method posts documents. The following code roughly shows the pattern used in the **run** method in the service.

```
…
If (lateSelection)//If the late selection option is used.
{
    formletterParmData.CreateData() //Create posting data.
}
While (next parmTable) //Loop over posting data, one parmTable record per journal.
{
    formletterJournalCreate.Run(); //Create one journal.
    formletterJournalPost.Run(); //Post one journal.
    If (printOut == Current)  //If printing after each posted journal.
    {
        formletterJournalPrint.PrintJournal(); //Print current posted journal.
    }
}
If (printOut == After) //If printing all journals after posting.
{
    formletterJournalPrint.PrintJournal(); //Print all posted journals.
}
…
```

The class also has a service operation method for each of the document types that can be posted by this service, such as **PostSalesPackingSlip**.

```
[SysEntryPointAttribute]
FormletterOutputContract postSalesOrderPackingSlip(SalesFormletterPackingSlipContract _contract)
{
    formletterContract = _contract;
    this.run();
    return outputContract;
}
```

The FormletterParmData class

The **FormletterParmData** class hierarchy can be used to create and maintain posting data in the parm tables, such as SalesParmUpdate, SalesParmTable, and SalesParmLine. The base class uses a template pattern, defining a template for how to create records in the parm tables. There are module-specific classes such as **SalesFormletterParmData**, and one class for each document type supported, such as **SalesFormletterParmDataPackingslip**.

The class has three public APIs:

- **CreateData**, which creates the needed data in the parm tables.
- **ReSelect**, which recreates the data based on the specQty in the posting form.
- **ReArrange**, which rearranges the data in the tables based on summary update settings.

Figure 8 illustrates the **FormletterParmData** class hierarchy.

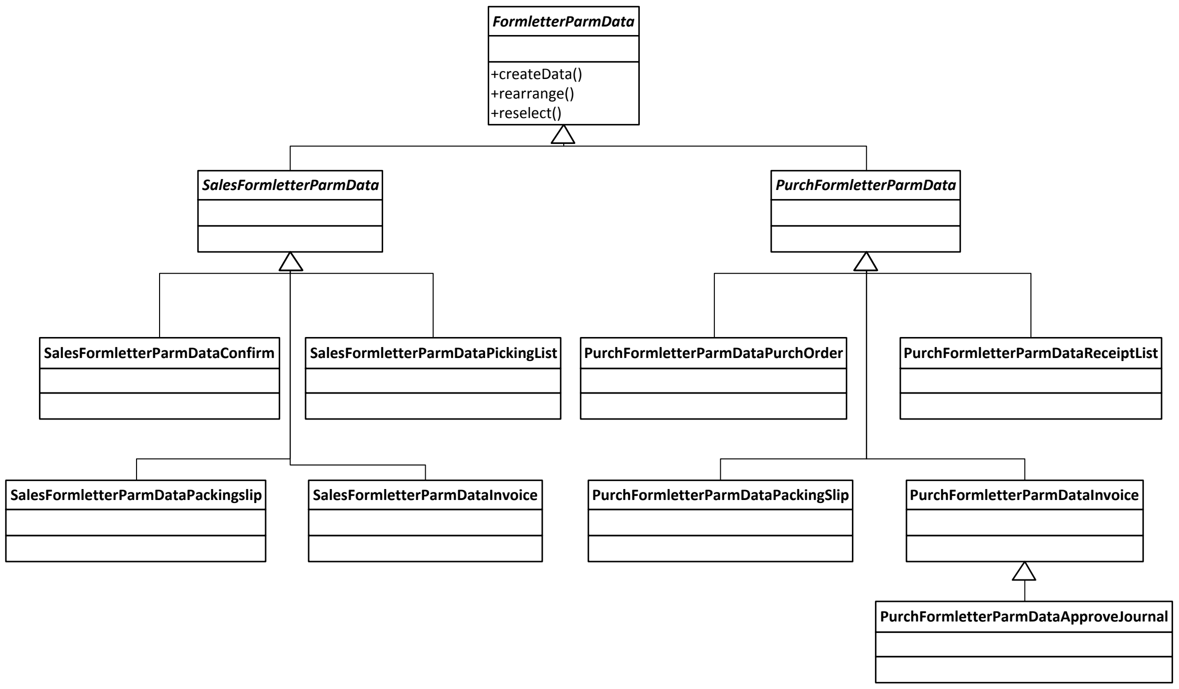

FormletterParmData
+createData()
+rearrange()
+reselect()
SalesFormletterParmData
PurchFormletterParmData
SalesFormletterParmDataConfirm
SalesFormletterParmDataPickingList
PurchFormletterParmDataPurchOrder
PurchFormletterParmDataReceiptList
SalesFormletterParmDataPackingslip
SalesFormletterParmDataInvoice
PurchFormletterParmDataPackingSlip
PurchFormletterParmDataInvoice
PurchFormletterParmDataApproveJournal

Figure 8: Class diagram: FormletterParmData class hierarchy

The FormletterJournalCreate class

The **FormletterJournalCreate** class hierarchy has the responsibility of creating one journal with a header, for example, CustPackingSlipJour, and a number of lines, for example CustPackingSlipTrans, and related journal data, for example, CustPackingSlipSalesLink. The base class uses a template pattern, defining the template for creating a journal. The template looks roughly like this

```
if (this.check())//Validate that the journal can be created.
{
this.initJournalHeader();//Initialize the journal header.
        this.createJournalHeader(); //Create the journal header.
        this.createJournalLines(); //Create journal lines.
        if (this.isJournalCreated()) //If a journal were successfully created.
        {
            this.insertRecordList();//Insert all records into the database.
            this.endCreate();//Finalize journal creation.
        }
}
```

The **FormletterJournalCreate** class hierarchy is illustrated in Figure 9.

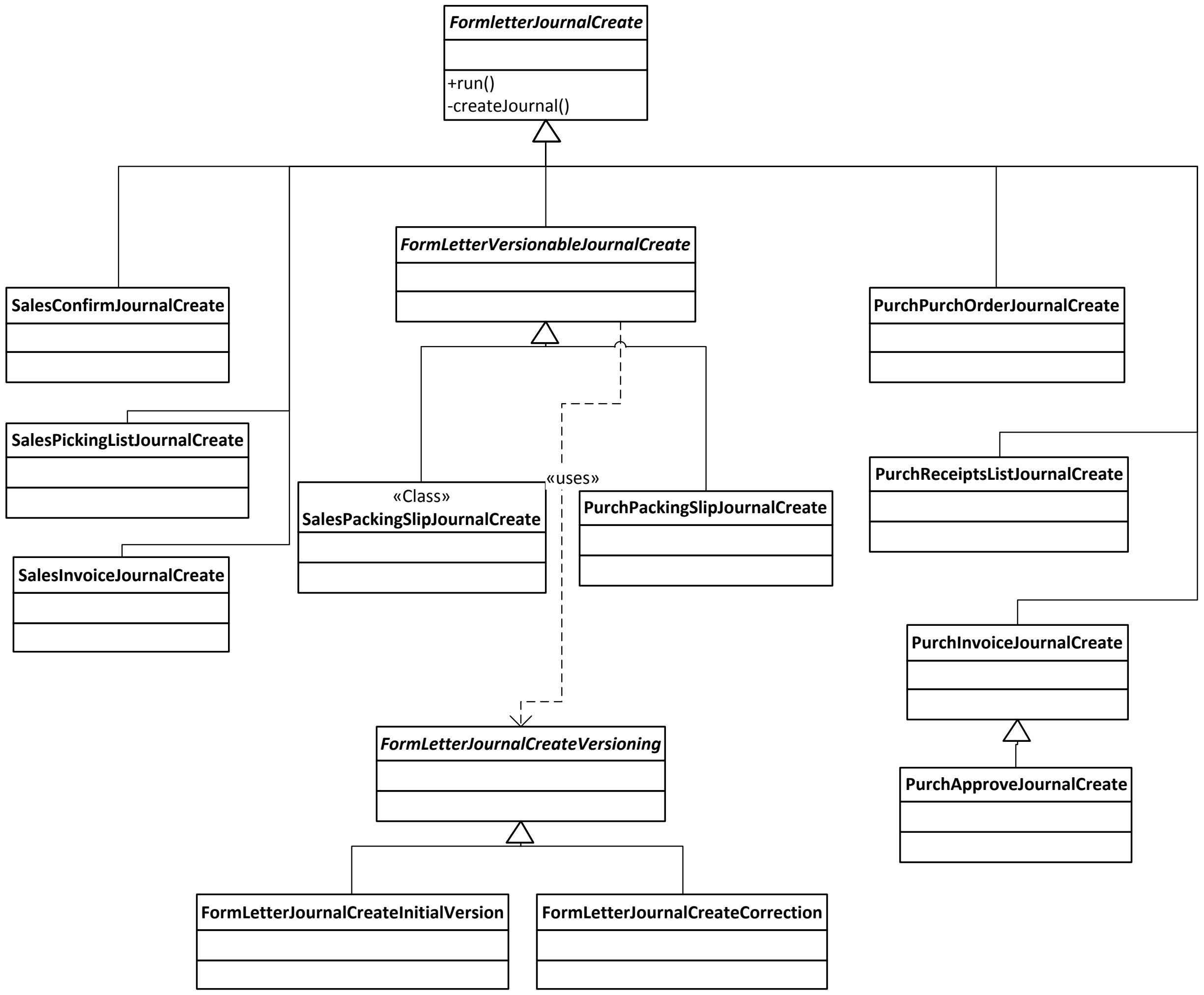

FormletterJournalCreate
+run()
-createJournal()
FormLetterVersionableJournalCreate
SalesConfirmJournalCreate
PurchPurchOrderJournalCreate
SalesPickingListJournalCreate
PurchReceiptsListJournalCreate
«uses»
«Class»
SalesPackingSlipJournalCreate
PurchPackingSlipJournalCreate
SalesInvoiceJournalCreate
PurchInvoiceJournalCreate
FormLetterJournalCreateVersioning
PurchApproveJournalCreate
FormLetterJournalCreateInitialVersion
FormLetterJournalCreateCorrection

Figure 9: Class diagram: FormletterJournalCreate class hierarchy

There are also document specific classes for each of the documents that can be created by the framework, such as **SalesPackingSlipJournalCreate**. These classes hold the logic specific for a document. Those document classes that support having multiple versions of the same document extend the **FormletterVersionableJournalCreate** class.

The **FormletterService** class instantiates this class hierarchy for each of the journals it needs to create and calls **run**.

The FormletterJournalPost class

The **FormletterJournalPost** class hierarchy can be used to post one journal, for example, updating ledger and inventory. The base class uses a template pattern, defining the template for posting a journal.

There is also a document specific class for each of the documents that can be posted by the framework, for example **SalesPackingSlipJournalPost**. These classes hold the logic specific for a document. This class hierarchy requires that a journal has been created and passed in.

The **FormletterService** class instantiates this class for each of the journals it needs to post and calls **run**.

Figure 10 is an illustration of the **FormletterJournalPost** class hierarchy.

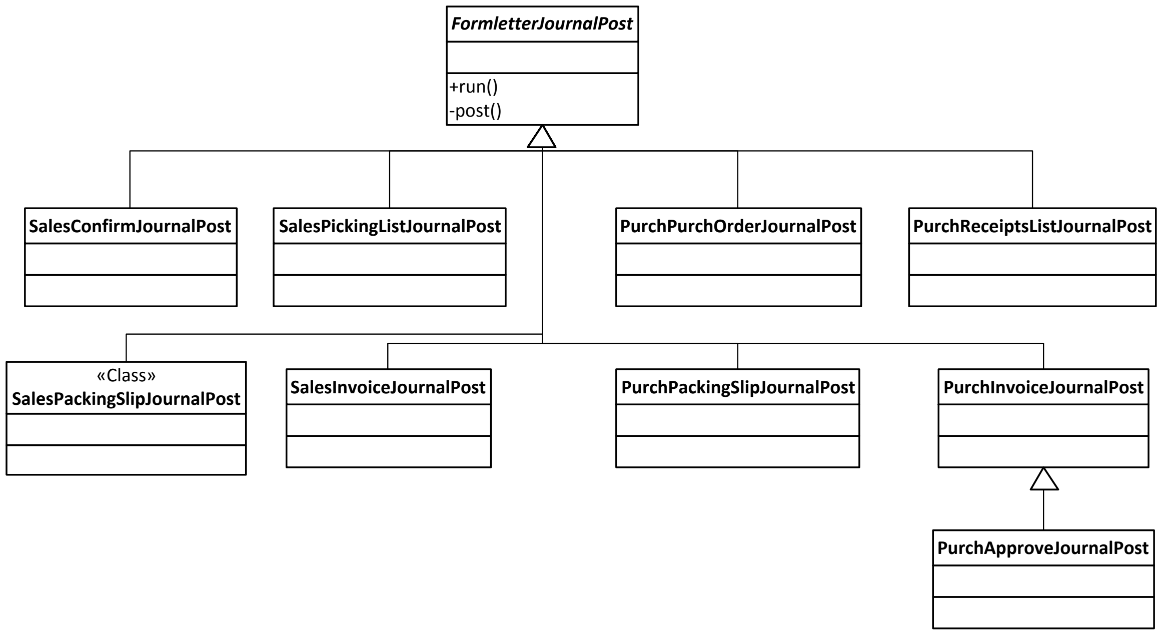

FormletterJournalPost
+run()
-post()
SalesConfirmJournalPost
SalesPickingListJournalPost
PurchPurchOrderJournalPost
PurchReceiptsListJournalPost
«Class»
SalesPackingSlipJournalPost
SalesInvoiceJournalPost
PurchPackingSlipJournalPost
PurchInvoiceJournalPost
PurchApproveJournalPost

Figure 10: Class diagram: FormletterJournalPost class hierarchy

The FormletterJournalPrint class

The **FormletterJournalPrint** class hierarchy can be used to control the printing of one or more journal documents. There is a document-specific class for each the documents that can be printed by the framework.

The **FormletterService** class instantiates this class either for each of the journals being posted, or once with a list of all posted journals.

Figure 11 is an illustration of the **FormletterJournalPrint** class hierarchy.

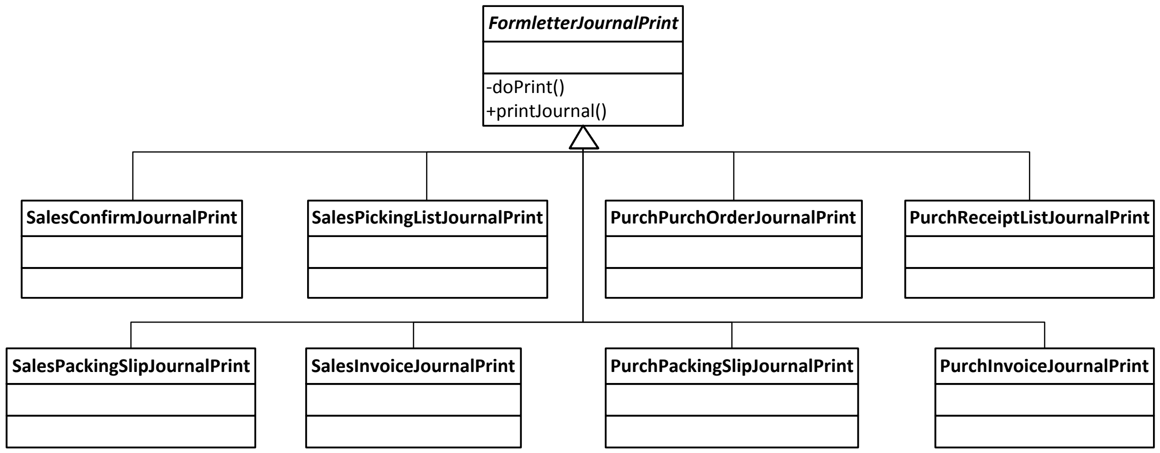

FormletterJournalPrint
-doPrint()
+printJournal()
SalesConfirmJournalPrint
SalesPickingListJournalPrint
PurchPurchOrderJournalPrint
PurchReceiptListJournalPrint
SalesPackingSlipJournalPrint
SalesInvoiceJournalPrint
PurchPackingSlipJournalPrint
PurchInvoiceJournalPrint

Figure 11: Class diagram: FormletterJournalPrint class hierarchy

The FormletterProvider class

The **FormletterProvider** class can be used to provide module specific data to each of the other class hierarchies. There is one child class per module, for example, **SalesFormletterProvider**. Figure 12 is an illustration of the **FormletterProvider** class hierarchy.

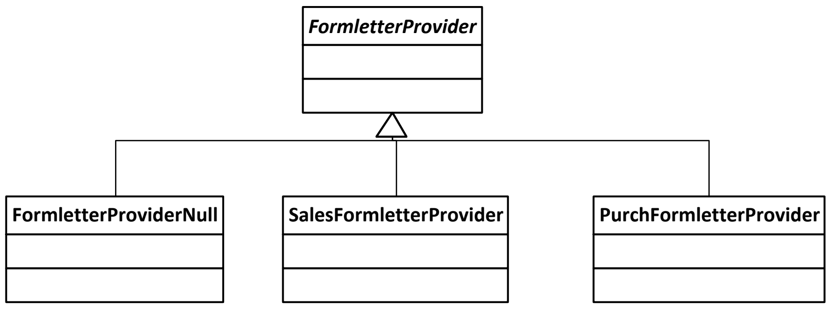


Figure 12: Class diagram: FormletterProvider class hierarchy

Where to place Microsoft Dynamics AX 2009 customizations

The following table shows where customizations made to the Microsoft Dynamics AX 2009 formletter framework should be moved in the Microsoft Dynamics AX 2012 formletter framework.

Microsoft Dynamics AX 2009 object	Customization made to logic for	Microsoft Dynamics AX 2012 object
Formletter	Creating parm data	FormletterParmData
	Creating journal data	FormletterJournalCreate
	Posting a journal	FormletterJournalPost
	General posting	FormletterService
	Printing	FormletterJournalPrint
	Interacting with posting form	FormLetterServiceController
SalesFormletter	Creating parm data	SalesFormletterParmData
	Interacting with SalesEditLines form	SalesFormLetter
SalesFormletter_Confirm	Creating parm data	SalesFormletterParmDataConfirm
	Creating journal data	SalesConfirmJournalCreate
	Posting a journal	SalesConfirmJournalPost
	Printing	SalesConfirmJournalPrint
	Interacting with SalesEditLines form	SalesFormLetter_Confirm
SalesFormletter_PickingList	Creating parm data	SalesFormletterParmDataPickingList
	Creating journal data	SalesPickingListJournalCreate
	Posting a journal	SalesPickingListJournalPost
	Printing	SalesPickingListJournalPrint
	Interacting with SalesEditLines form	SalesFormLetter_PickingList
SalesFormletter_PackingSlip	Creating parm data	SalesFormletterParmDataPackingslip
	Creating journal data	SalesPackingSlipJournalCreate
	Posting a journal	SalesPackingSlipJournalPost
	Printing	SalesPackingSlipJournalPrint
	Interacting with SalesEditLines form	SalesFormLetter_PackingSlip
SalesFormletter_Invoice	Creating parm data	SalesFormletterParmDataInvoice
	Creating journal data	SalesInvoiceJournalCreate
	Posting a journal	SalesInvoiceJournalPost
	Printing	SalesInvoiceJournalPrint
	Interacting with SalesEditLines form	SalesFormletter_Invoice
PurchFormletter	Creating parm data	PurchFormletterParmData

Microsoft Dynamics AX 2009 object	Customization made to logic for	Microsoft Dynamics AX 2012 object
	Interacting with PurchEditLines form	PurchFormLetter
PurchFormletter_PurchOrder	Creating parm data	PurchFormletterParmDataPurchOrder
	Creating journal data	PurchPurchOrderJournalCreate
	Posting a journal	PurchPurchOrderJournalPost
	Printing	PurchPurchOrderJournalPrint
	Interacting with PurchEditLines form	PurchFormLetter_PurchOrder
PurchFormletter_PickingList	Creating parm data	PurchFormletterParmDataReceiptsList
	Creating journal data	PurchReceiptListJournalCreate
	Posting a journal	PurchReceiptListJournalPost
	Printing	PurchReceiptListJournalPrint
	Interacting with PurchEditLines form	PurchFormLetter_ReceiptsList
PurchFormletter_PackingSlip	Creating parm data	PurchFormletterParmDataPackingSlip
	Creating journal data	PurchPackingSlipJournalCreate
	Posting a journal	PurchPackingSlipJournalPost
	Printing	PurchPackingSlipJournalPrint
	Interacting with PurchEditLines form	PurchFormLetter_PackingSlip
PurchFormletter_Invoice	Creating parm data	PurchFormletterParmDataInvoice
	Creating journal data	PurchInvoiceJournalCreate
	Posting a journal	PurchInvoiceJournalPost
	Printing	PurchInvoiceJournalPrint
	Interacting with PurchEditLines form	PurchFormLetter_Invoice

Batch parallelism enhancements

The batch parallelism strategy used by the formletter framework has been enhanced in Microsoft Dynamics AX 2012.

The formletter framework can now use batch parallelism both when using late selection to create posting data and when posting the journals. The default batch task size has been change to include five orders or journals per batch task. The batch size is customizable in the parameter forms for sales and purchase orders.

When a posting is executed in batch and late selection is chosen, the framework will start a number of "create posting data" tasks that create data for the orders in the posting data tables (for example, one record per sales order in the SalesParmTable table). It also starts one "rearrange posting data" task that rearranges the posting data based on the summary update settings when all the "create posting data" tasks have completed. When the posting data creation is completed, or late selection is not chosen, then the framework starts a number of "post journal" tasks, each handling a number of journals, and an "end posting" task that finalizes the posting when all "post journal" tasks have completed.

Figure 13 shows the batch parallelism posting process.

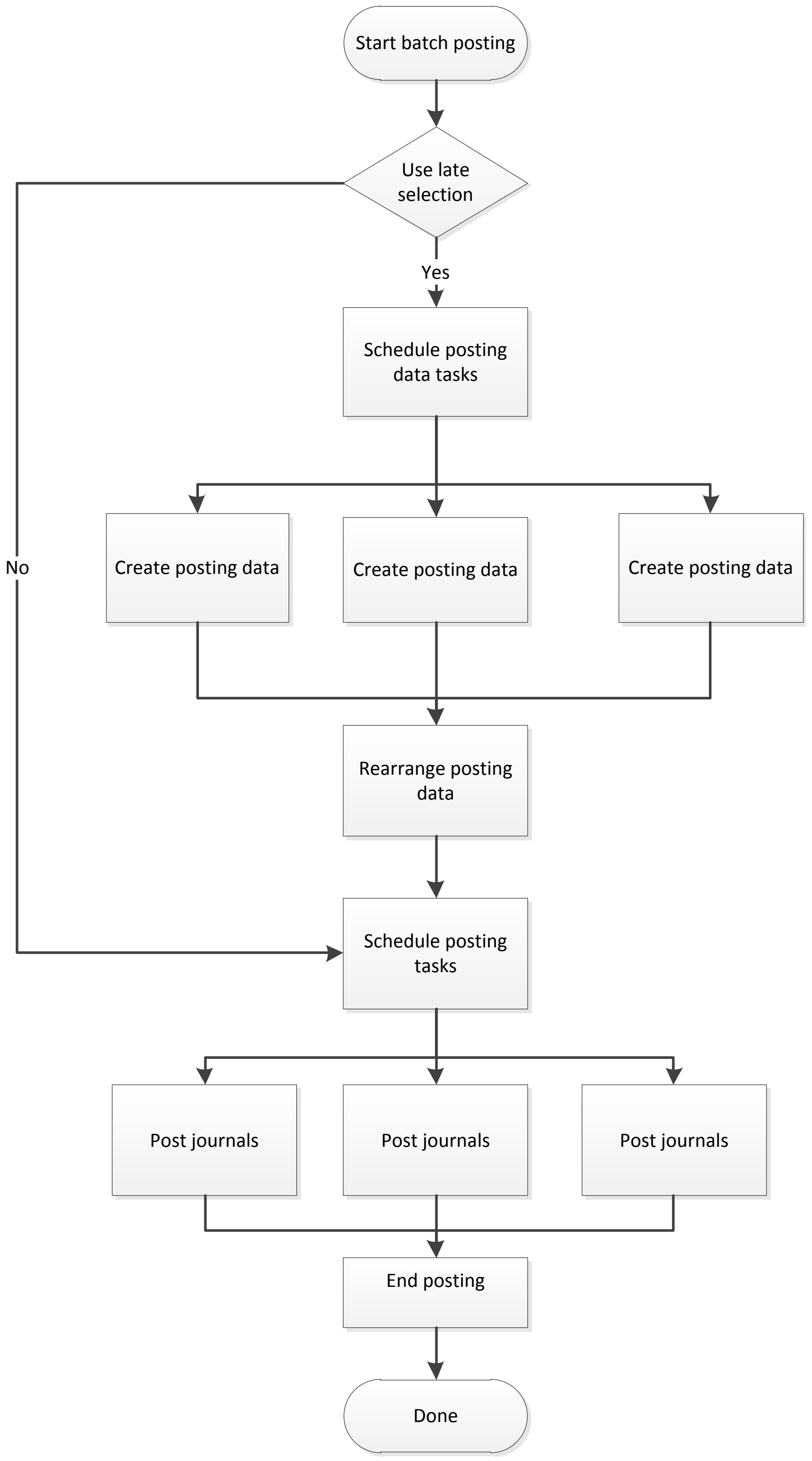


Figure 13: Batch parallelism

The FormletterServiceBatchTaskManager class

The **FormletterServiceBatchTaskManager** class can be used to create batch tasks when the formletter framework is set to execute in batch. Figure 14 illustrates the **FormletterServiceBatchTaskManager** class hierarchy.

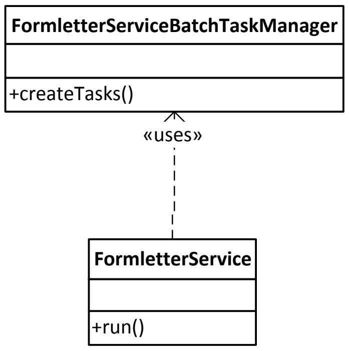


Figure 14: Class diagram: FormletterServiceBatchTaskManager

The FormletterServiceMultithread class

The **FormletterServiceMultithread** class is a specialization of the **FormletterService** class. The class is invoked from the **FormletterServiceBatchTask** class and ensures that the service handles only the journals that the tasks have responsibility for. Figure 15 illustrates the **FormletterServiceMultithread** class hierarchy.

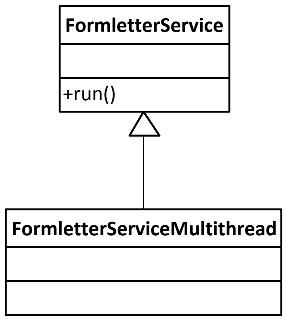


Figure 15: Class diagram: FormletterServiceMultithread

Batch task classes

The formletter framework contains four batch task classes that extend the base **FormletterBatchTask** class. The following table describes what each batch task class can be used for.

Batch task class	Use
FormletterParmDataCreateDataBatchTask	Creating posting data for each order passed to the class.
FormletterParmDataRearrangeBatchTask	Rearranging the posting data based on the summary update settings.
FormletterServiceBatchTask	Utilizing the **FormletterServiceMultithread** class to post the journals passed to the class.
FormletterServiceEndBatchTask	Finalizing the posting process.

Figure 16 illustrates the **FormletterBatchTask** class hierarchy.

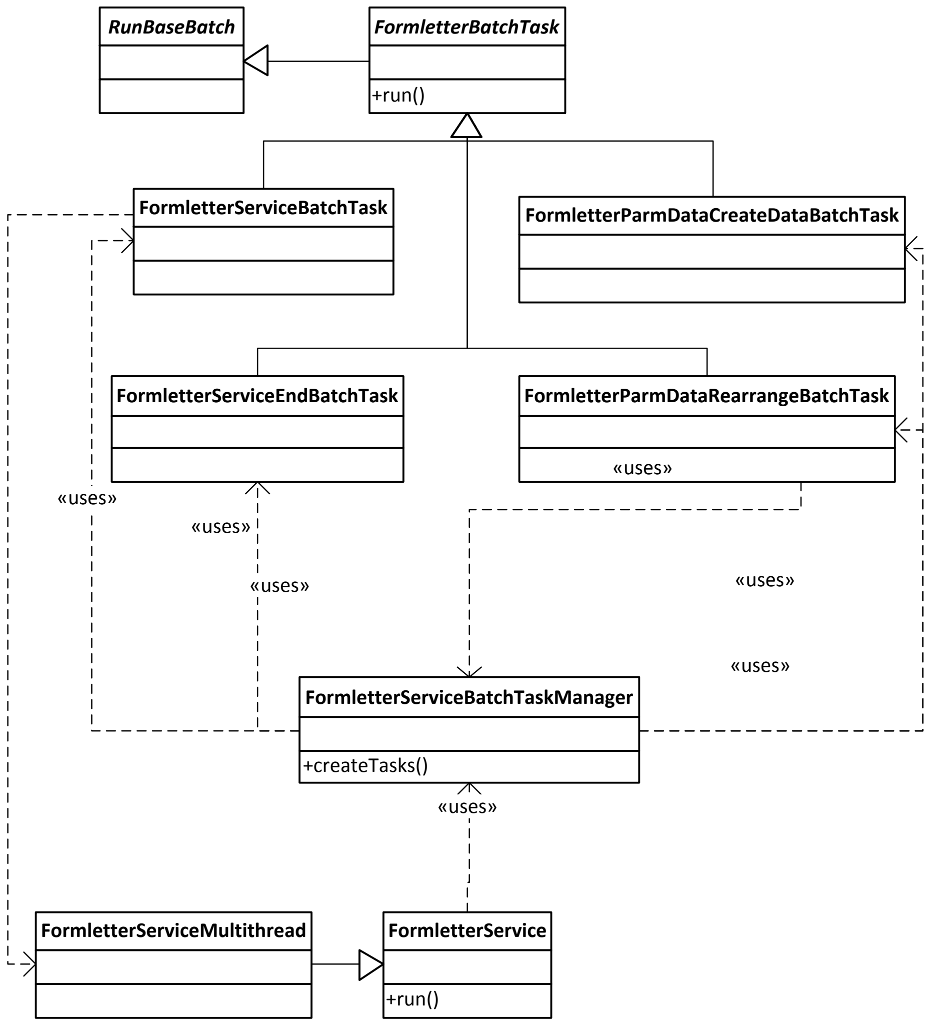

RunBaseBatch
FormletterBatchTask
+run()
FormletterServiceBatchTask
FormletterParmDataCreateDataBatchTask
FormletterServiceEndBatchTask
FormletterParmDataRearrangeBatchTask
«uses»
«uses»
«uses»
«uses»
«uses»
«uses»
FormletterServiceBatchTaskManager
+createTasks()
«uses»
FormletterServiceMultithread
FormletterService
+run()

Figure 16: Class diagram: Formletter batch tasks