SPS Order Writer Overview

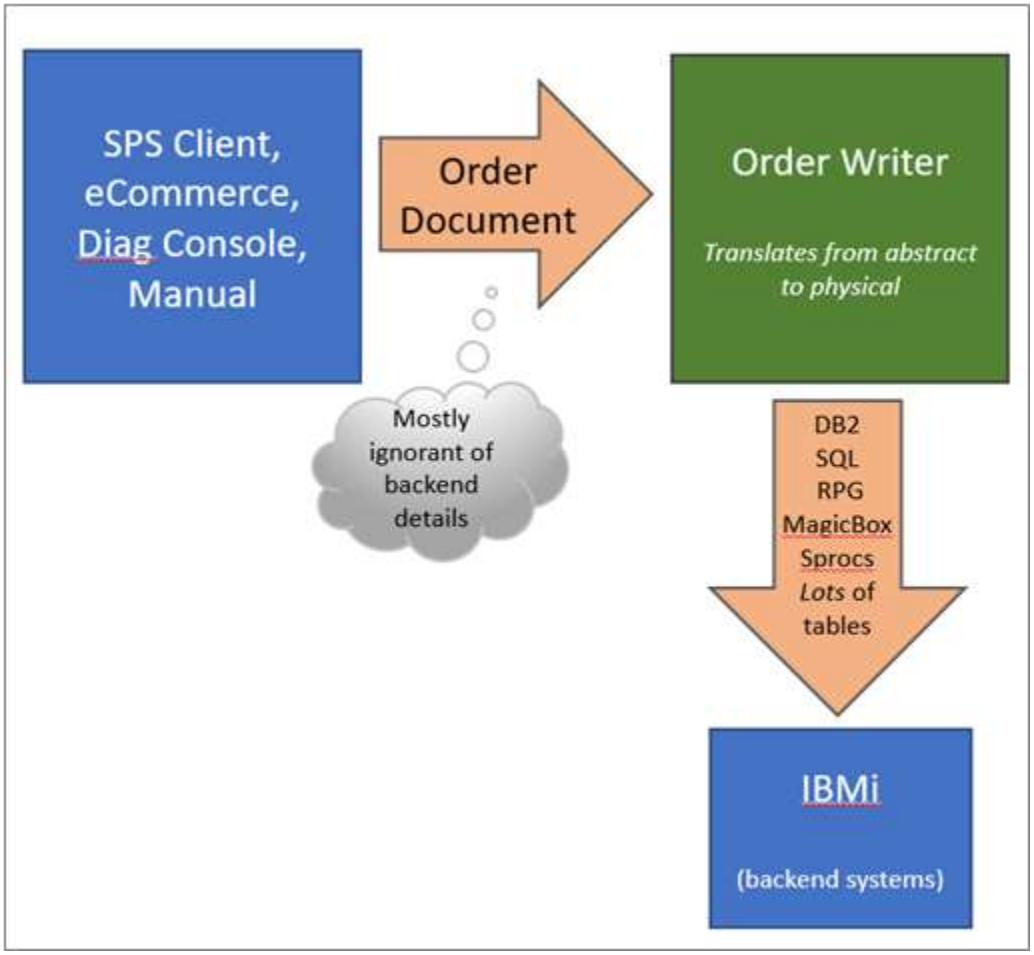
Capabilities and Usage

Table of Contents

Overview

Basic Description

The Order Writer (OW) is an intermediary between business clients and the corresponding backend systems. It is focused on order creation, amendment, and cancellation.



The order documents are in the form of XML documents. Though the primary client is currently SPS, any program can use OW to easily create orders. It is easy for clients to use, and really easy to add new backend pieces!

High-Level History

The OW started as a Java-based SOAP web service. It ran on the iSeries and was orchestrated by BizTalk. In order to make it more reliable and easier to maintain for our .NET-centric team it was translated to C# and turned into a Windows service.

The OW and surrounding code is 10-15 years old. Some the coding style and the any lack of use of more advanced C# constructs can be attributed to that.

SPS Client Integration

The actual order submission process requires two essential steps:

1. Formulating the XML document from the SaleBase instance, and
2. Writing to the Submission_XXX tables.

After the XML is written to the DB, alongside the appropriate control records, it becomes subject to the Order Writer's processing and may be picked up at any point.

Order Document Schema Definition

The order document schema is managed as part of the Order Writer solution. The XSDs are managed directly and skeleton classes are generated using the **xsd.exe** utility. This a simple process and there are instructions for doing this embedded in the *OrderWriter.Schemas* project.

Schema Assembly

Applications such as SPS can import the schema by referencing the *OrderWriterSchemas.dll* assembly. The stubs may then accessible using the following example:

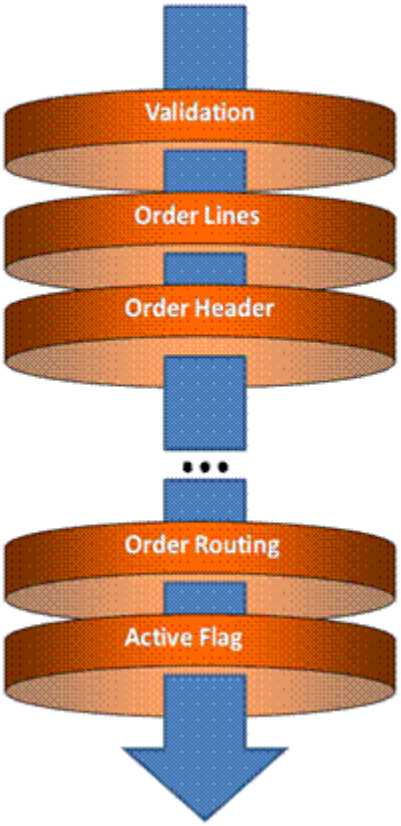
```
...
using OrderSchema = Cdw.XmlSchemas.Order;
using OrderExSchema = Cdw.XmlSchemas.OrderEx;
...

public class SaleBase
{
    ...
    OrderSchema.Order order = new OrderSchema.Order();
    order.Contact = new OrderSchema.Contact();
    order.Contact.FirstName = "Joe";
    order.Contact.LastName = "Sloppy";
    ...
}
```

Plugins

Overview

The main Order Writer process is not monolithic. Rather, it is broken into discrete steps or modules called *plugins*. This approach allows each feature to be treated as a small, mostly independent, program that is focused on a particular task. Is also enforces a degree of modularity that is helpful for a message-based environment.

An in-process order is passed through the appropriate series of plugins required to complete processing:

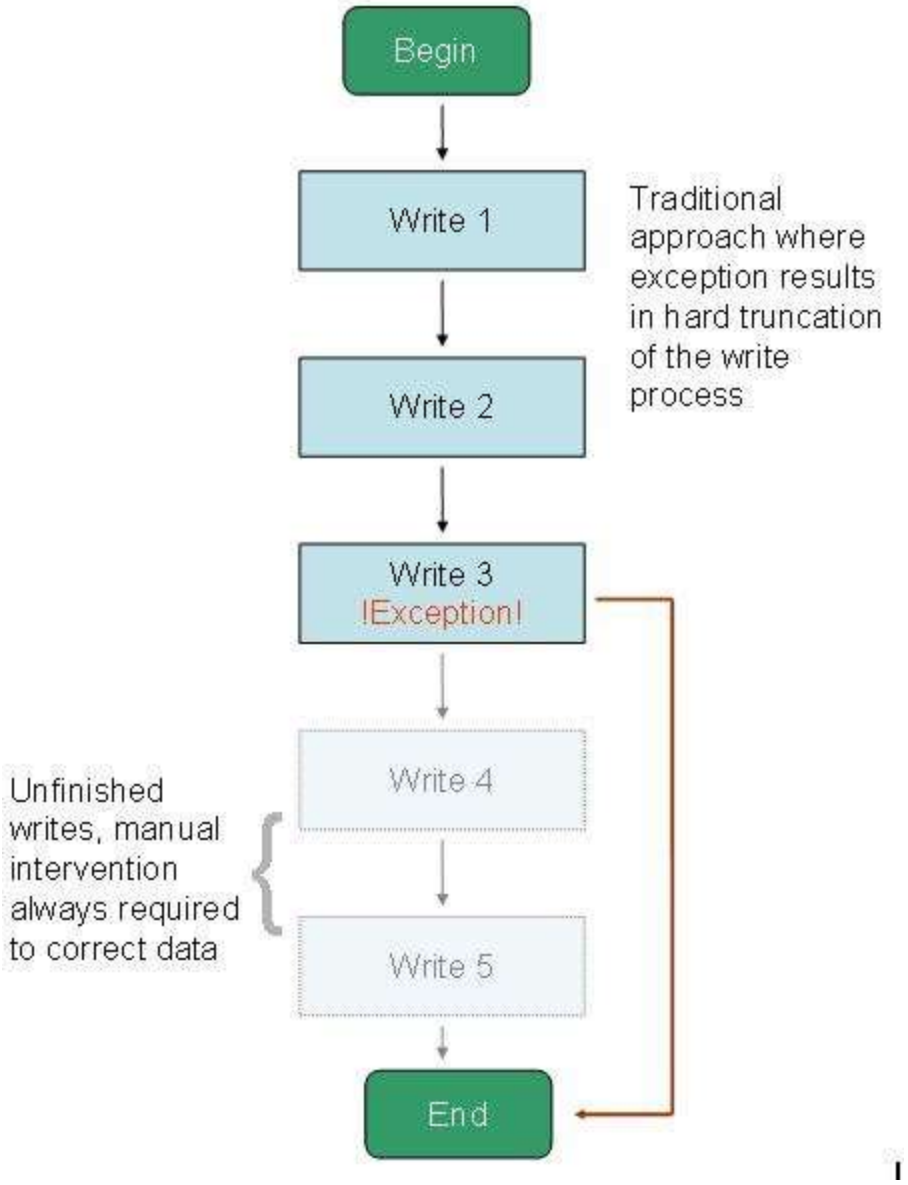


A different set of plugins results in different behavior. This is the basis of behavioral differences between Create, Amend, Cancel, and Bid Create - they have a different set of plugins but the engine is the same. The set of plugins to be executed for a given component is selected indirectly by the Initiator as the operation is started, and then directly by the *OrderUploadLegacyHandler* class.

Plugins and the State Machine

The Create/Amend process is complex and performs a lot of database writes that could fail before the entire order creation process has been completed. Though it is expected to be rare, this might result in orders that are only partially written and require manual intervention on the AS400 to correct.

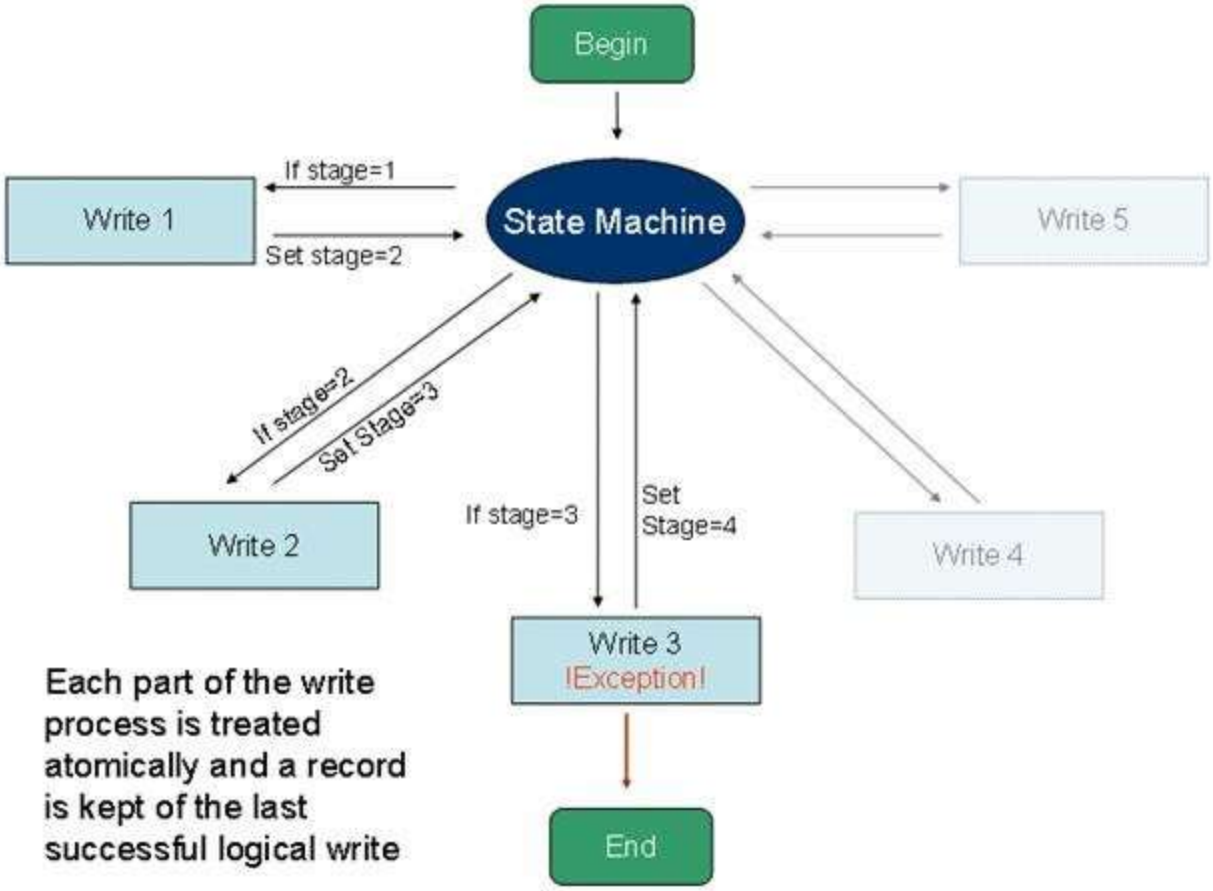
In a tradition sequential monolithic approach, an error in the middle is problematic:



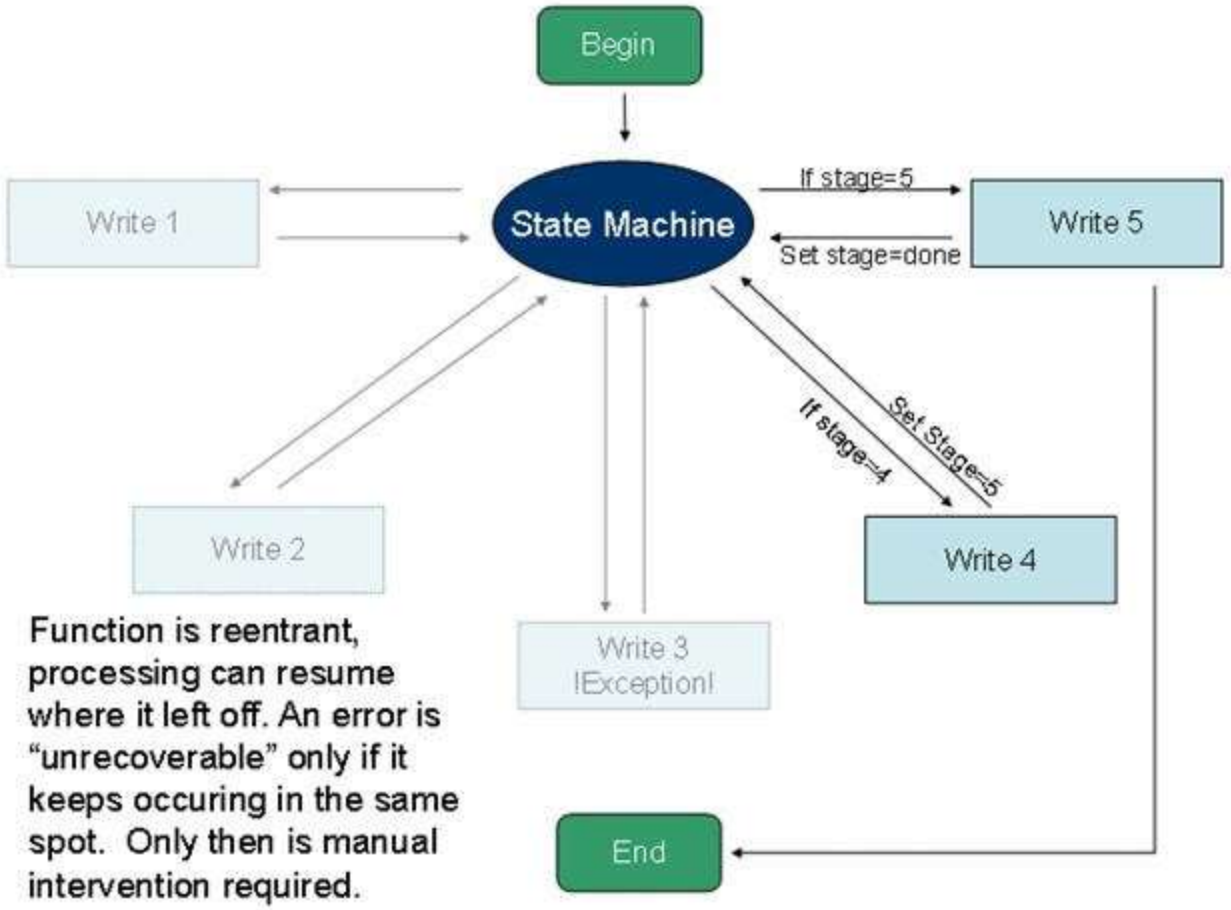
The Order Writer, however, takes advantage of the plugin approach and implements a simple *state machine*. This approach:

1. Allows messages to be submitted multiple times without damage.
2. Enables orders to be resumed and picked up where they left off.
3. Makes it easier to identify the source of problems

As each plugin completes its processing, the Order Writer records the progress of the operation. The next write to be performed for an order is determined based upon which one was last completed. If an exception is encountered while processing a given plugin, then, the Order Writer knows roughly where it left off in the processing sequence.



In many cases, after an environment fix, the processing for an order can be resumed and completed successfully due to the design:



Since most plugins are designed to handle both Create and Amend modes, they are generally intelligent enough to handle reentrancy. In fact, if an error occurs during a Create operation, it may sometimes automatically be converted to an Amend-like mode because Amend mode is "smarter" and more likely to succeed after an error.

Plugin Interface

There are different types of plugins that implement the plugin interface to different extents. Here is an example of a plugin designed to handles both create and amend for a business feature.

```
public interface ICommonPlugin : ICreateAmendBasePlugin
{
    ///<summary>
    /// Specifies whether the plugin should be executed even in the event of the exception on another plugin
    /// prior in the chain.
    ///</summary>
    bool alwaysExecute();

    ///<summary>
    /// Invoked on all plugins PRIOR TO initiating the write operation.  Gives
    /// each plugin a chance to reject the entire write operation due to semantic
    /// errors in the input.
    ///</summary>
    bool prevalidate(CreateAmendCommonContext ctx);

    ///<summary>
    /// Invoked on all plugins AFTER completing the write operation.  Gives
    /// each plugin a chance to perform sanity checks on what was created.  If
    /// this method throws an exception, the caller will receive an error.
    ///</summary>
    void postvalidate(CreateAmendCommonContext ctx);

    ///<summary>
    /// Invoked during the write process to handle its specific part of the write
```

```
    /// process.
    ///</summary>
    void execute(CreateAmendCommonContext ctx);

    /// <summary>
    /// Invoked on each plugin when there is an exception while processing
    /// the order.  Every plugin will have a chance to react if they implement this.
    /// </summary>
    void onException(CreateAmendCommonContext ctx);
}
```

Example Logic (Create/Amend Plugin)

Here is some simple logic to show some common aspects of a plugin that operates in both create and amend mode.

```
public void execute(CreateAmendCommonContext ctx)
{
    // Determine our action code based upon what's happening.
    string actionTaken;
    if (ctx.mode == BaseContext.MODE_CREATE)
    {
        actionTaken = "IE";
    }
    else if (ctx.mode == BaseContext.MODE_AMEND)
    {
        actionTaken = "AE";
    }
    else
    {
        return;
    }

    // Write our audit entry.

    // *** NOTE ***
    // If there is an exception anywhere, the OW will automatically route to WC/SLSREP to avoid
    // having the order go downstream.  This also gives the AM a chance to amend/fix the order.
    // Entries will also automatically be added to the appropriate logs so there is essentially
    // no logic that the plugin needs to implement unless we want something different.
    OEP87 oep87 = new OEP87();

    oep87.CompanyCode = ctx.order.Header.companyCode;
    oep87.OrderNumber = ctx.order.Properties.orderNumber;
    oep87.OldHoldCode = "";
    oep87.NewHoldCode = actionTaken;
    oep87.Program = "SPS";
    oep87.BatchActionDate = OrderEntryFunctions.getRPGDate(ctx.dbHelper).ToAS400DateDecimal();
    oep87.BatchActionTime = OrderEntryFunctions.getRPGDate(ctx.dbHelper).ToAS400TimeDecimal();
    oep87.ActionTaken = actionTaken;

    OEP87File.Insert(ctx.dbHelper, oep87);
    }
```

Plugin Services

Every plugin has access to an implementation of this interface full of some basic functions. **This interface is a rough model of low-level functions that we might need to provide for our plugin microservices.**

```
public interface IPluginServices : IDisposable
{
    // Methods for easy error/warning publishing via CdwExceptionManager.
    void PublishWarning(String msg);
    void PublishWarning(String msg, Exception ex);
    void PublishError(String msg);
    void PublishError(String msg, Exception ex);

    void PublishBeanWriteError(BeanUpdateType updateType, BeanBase bean, Exception ex);
    void PublishBeanWriteError(BeanUpdateType updateType, BeanBase bean, Exception ex, String activityThatBombed
);
    void PublishBeanInsertUpdateError(BeanBase bean, Exception ex, bool wasInsert);
    void PublishBeanInsertUpdateError(BeanBase bean, Exception ex, bool wasInsert, String activityThatBombed);
    void PublishOrderDocError(object subject);
    void PublishOrderDocError(object subject, String messsage);
```

```
    void PublishOrderDocError(object subject, Exception ex);
    void PublishOrderDocError(object subject, String messsage, Exception ex);

    // Methods to post errors in the order doc for easier debugging during support.
    void PostMessageToOrderDoc(String msg);
    void PostMessageToOrderDoc(List<String> msgList);

    // Post a completely new order to the queue.
    SubmitOrderToMOPRecord SubmitOrderDoc_NewProductOrder(
            Order order /* Should already have new order # populated */,
            bool queueUntilThisOperationFinishes,
            bool useSameOrderWriterInstance);

    SubmitOrderToMOPRecord SubmitOrderDoc_NewProductOrder(
            Order order /* Should already have new order # populated */,
            int componentIDWeWillWaitFor,
            bool useSameOrderWriterInstance);

    // Configuration file access.
    IServiceConfigReader ConfigReader { get; }

    // Order-level logger.
    EntityLog OrderLogger { get; }

    // Order-level logger.
    string UserIDForSubmission { get; }

    // Sets a context value used to identify order areas affected by error (in a friendly and human-
readable way).
    // In general, this value is set automatically and the plugin don't need to modify it directly.
    OrderProcessAreaAffected CurrentAreaAffected { get; set; }
    Type CurrentPluginType { get; set; }

    // Methods that update the actual order doc in the submission tables.  This is needed on
    // rare occasions when a plugin (such as CouponWS) modifies the order object.
    bool PersistUpdatedDoc(Order order);
    bool PersistUpdatedDoc(OrderCancel orderCancel);
    bool PersistUpdatedDoc(SPSBidCreate bidCreate);

    // Method to test whether the given error is present in theprocess state document.
    bool IsErrorPresent(OrderProcessAreaAffected? areaAffectedToCheck, OrderProcessErrorSource? errorSourceToChe
ck, bool lastAttemptOnly);

    // Method to see if a given plugin type is to be skipped.
    bool ShouldSkipPlugin(Type pluginType);

    // Causes the order/quote to be routed to WC/WV.  It goes in the rep's router.
    BaseContext PluginContext { get; set; }
    bool ForceSLSREPRouting { get; set; }

    // Request a retry for the operation at the OW level.
    bool RequestRetry { get; set; }

    void AddToHistory(string ErrorMessage);

    // Run diagnostic commands on the entire service like we can do from Order Console.
    ServiceDiagnosticEx DiagInterface { get; }
}
```

Requirements for a Plugins

This document outlines facilities currently used by the Order Writer for the purpose of determining what new microservices would require.

- Code generator for POCOs to represent data from IBMi and other sources in a platform-independent manner.
- Logging mechanism to be shared among all the microservices, orchestrator, and other order-oriented logic.
- Database connection mechanisms and patterns.
- Common plan for exception handling.
- Easy access to core, shared business functions. An example includes routing to SLSREP upon exception.

Each of these mechanisms must be wrapped in a manner that they are simple to leverage and control.

Orchestrator

Plugin Sequencer

The Order Writer contains logic for "popping" orders from the queue and scheduling them, but at the core its job is to run a series of plugins. This following simplified code depicts how it is done. Invocations into plugins are highlighted to show where they fit in the logic. This logic is located in the file `PluginSequence.cs` of the OW solution.

```
///<summary>
/// This module controls the process of creating, amending, and cancelling
/// orders.  It will actually do any work that can be described using the
/// plugin architecture of this service.
///
/// The writeOrder() operation is re-entrant; If it is called multiple times
/// then it resumes where it left off on the last plugin for a given operation such
/// as creating an order.  Half-completed creations, for example, will be
/// resumed where they left off and if the creation was complete on another
/// call, the call has no effect.
/// no effect.
///</summary>
public class OrderSerializer
{
    // ---[ Member Variables ]---------------------------------------------
    IPluginServices pluginServices;

    // ---[ writeOrder() method --> main plugin control]-------------------
    void writeOrder( pluginCtx )
    {
        PluginInstanceGroup pluginSequence = pluginCtx.getMainPluginGroup();

        // ---[ Perform plugin-level prevalidations ]--------------------------------------
        pluginSequence.runPrevalidations(pluginCtx);

        // ---[ Invoke the plugins in sequence to perform the operation ]------------------
        // Record our initial DB nesting level for "bad nesting" detection.
        try
        {
            // Execute!  This delegates to another method.
            pluginSequence.executePlugins( pluginCtx );

            // ---[ Perform plugin-level postvalidations ]-------------------------------------
            //  This delegates to another method.
            pluginSequence.runPostvalidations(pluginCtx);
        }
        catch (Exception ex)
        {
            pluginSequence.runOnExceptionHandlers(pluginCtx);

            // Whoa dude!
            // Rethrow.
            throw;
        }
    }
}
```

Prevalidation / Postvalidation / Exceptions

Basic pre-validation logic.

```
public void runPrevalidations( pluginSvcs )
{
    // Prevalidate!
    foreach (PluginToRun plugin in _pluginInstancesToRun)
    {
        MethodInfo meth =
            OrderEntryFunctionsEx.findMethod(plugin, "prevalidate");

        meth.Invoke(plugin, new object[] { pluginSvcs });
    }
}
```

Basic post-validation logic:

```
public void runPostvalidations( pluginSvcs )
{
    foreach (PluginToRun plugin in _pluginInstancesToRun)
    {
        MethodInfo meth =
            OrderEntryFunctionsEx.findMethod(plugin , "postvalidate");

        meth.Invoke(plugin, new object[] { pluginSvcs });
    }
}
```

Exception consideration for plugins:

```
public void runOnExceptionHandlers( pluginSvcs )
{
    foreach (PluginToRun plugin in _pluginInstancesToRun)
    {
        MethodInfo meth =
            OrderEntryFunctionsEx.findMethod(plugin, "onException");

        if (meth != null) // Plugin not required to implement.
        {
            meth.Invoke(plugin, new object[] { pluginSvcs });
        }
    }
}
```

Main Plugin Sequencing

This is the main plugin “sequencer”.

```
public virtual void executePlugins( pluginSvcs )
{
    // Generate an ID for each plugin.  Unique only to this list.
    // This is used to track which ones have executed.
    pluginIDsByPlugin = BuildPluginIDsByPlugin(pluginIDsByPlugin);

    // Run the plugins!
    int pluginIdx = 0;
    try
    {
        bool anyPluginEncounteredUnhandledException = false;

        // Loop below avoids IEnumerable-based logic (dodges collection-was-modified exception)
        while ( more plugins to run )
        {
            // Get the plugin that we are gonna run.
            PluginToRun plugin = GetPluginToRun(pluginIdx, pluginSvcs, … );
            int pluginSymbol = GetPluginSymbol(pluginIDsByPlugin, plugin);

            // See if we should run the plugin.

            bool shouldRunPlugin;
            if (plugin.PluginInstance.AlwaysExecute)
        {
                shouldRunPlugin = true;

        }
            else if (hasPluginAlreadyExecuted(xactionState, pluginSymbol))
        {
                shouldRunPlugin = false;

        }
            else if (pluginSvcs.ShouldSkipPlugin(plugin.PluginInstance.GetType()))
        {
                shouldRunPlugin = false;
        }
            else if (anyPluginEncounteredUnhandledException &&
                     (plugin.ErrorMode == PluginErrorMode.DontRunAfterError))
        {
                shouldRunPlugin = false;
        }
            else
        {
                shouldRunPlugin = true;
```

```
        }

            if (shouldRunPlugin)
        {
                try
            {
                    // Run the plugin!  This delegates to another method for this.
                    executeSinglePlugin( pluginSvcs, … );
            }
                catch (Exception ex)
            {
                    // Is this a second uncaught exception?  Barf.  If so then
                    // this will stop the order processing completely until it can be examined
                    // manually.  These would go to pager if there are too many.
                    if (anyPluginEncounteredUnhandledException)
                {
                        throw;
                }
                    // A first failure just get published and we make note that there was an exception.
                    else
                {
                        // Force routing to SLSREP.
                        pluginCtx.pluginServices.ForceSLSREPRouting = true;

                        // Publish the error.  These would go to pager if there are too many.
                        pluginSvcs.PublishError("Uncaught error in Plugin", ex);

                        // Mark that we've seen an exception so we know how to handle subsequent plugins.
                        anyPluginEncounteredUnhandledException = true;

                        // If we are supposed to just kick out with an exception
                        // (which will show in Order Console), do so now.
                        if (plugin.ErrorMode == PluginErrorMode.StopProcessingOnError)
                    {
                            throw;
                    }
                }
            }
        }

            // Get ready for the next plugin.
            pluginIdx++;
        }
    }
    catch
    {
        // Rethrow.  Since there are so many layers of error handling, anything that gets here must
        // be a VERY serious problem.
        throw;
    }
}
```

```
public static void executeSinglePlugin( PluginToRun plugin, IPluginServices pluginSvcs, … )
{
     String currentActivity = "invoking plugin "
                 + pluginIDsByPlugin[plugin] + ": " + plugin.PluginInstance.GetType().GetShortName();

     // Record our initial DB nesting level for "bad nesting" detection.
     int startingDBNestingLevel = dbHelper.getNestingLevel();

     try
     {
          // Execute the action.
          MethodInfo meth =
              OrderEntryFunctionsEx.findMethod(plugin.PluginInstance, "execute", … );

          if (meth == null)
               throw new CdwException("Unable to find execute("
                             + pluginCtx.GetType().GetShortName()
                             + ") in plugin "
                             + plugin.PluginInstance.GetType().GetShortName());
          else
          {
               meth.Invoke(plugin.PluginInstance, new object[] { pluginCtx });
          }

                // Mark that execution of this plugin was completed.
```

```
                markPluginAsExecuted( plugin );
     }
     catch (Exception ex)
     {
          // Mark that execution of this plugin was completed in an error state.
          markPluginAsException( plugin, ex );
     }
}
```

IDBHelper

What is It?

All IBMi-based database actions in Order Writer are performed through the IDBHelper implementation (referred to in this document as just DBHelper for brevity). It offers shortcut methods for threads to perform SELECTs, INSERTs, UPDATEs, and DELETEs. Major features include:

- Simplifies DB-related code.
- Handles nested database-related methods efficiently.
- Provides easy-to-use wrappers for common fetch operations.

The ORM used by OW and all the Enterprise Web Services is built around DBHelper.

IDBHelper Interface

The DBHelper interface makes it very easy to do common stuff, corresponding roughly to a connection plus convenience methods. In this listing we've removed a few miscellaneous methods for clarity.

```
public interface IDBHelper
{
    IDbConnection connect();
    int getNestingLevel();
    void startTransaction();
    void close();
    void commit();
    void rollback();

    int exec(String strSQL, params Object[] ao);
    int quickExec(String sql);
    IQueryResult2D quickQuery(String sql);
    IQueryResult2D quickQuery(String sql, params Object[] parms);

    DBResultSet query(String strSQL);
    DBResultSet query(String strSQL, params Object[] ao);
    DBResultSet query(String strSQL, int cmdTimeout, params Object[] ao);
    DBResultSet query(String strSQL, int cmdTimeout, out Type connType, params Object[] ao);
    T QueryForOneSQLValue<T>(String sql, Object defaultValue);
    T QueryForOneSQLValue<T>(String sql, Object[] parms, Object defaultValue);
    Object QueryForOneSQLValue(String sql, Object defaultValue);
    Object QueryForOneSQLValue(String sql, Object[] parms, Object defaultValue);
    T[] QueryForOneSQLList<T>(String sql, params Object[] parms);
}
```

IConnectionFeeder

The IConnectionFeeder interface is implemented as a delegate which DBHelper uses to wrap new connections as needed. In practice, the OW uses several implementations of this interface with different purposes.

```
public interface ConnectionFeeder
{
    /**
     * Method to obtain a database connection.
     **/
    IDbConnection getConnection(string sql, params object[] sqlParms);

    /**
     * Method to release a database connection previously obtained.
```

```
     **/
    void releaseConnection(IDbConnection svr);

    /**
     * Pre- and post-processing hooks.
     **/
    void statementPreBind(int id, string sqlText);
    void statementPreProcess(int id, IDbCommand cmd);
    void statementPostProcess(int id, IDbCommand cmd);

    /**
     * Transactional methods.
     **/
    void beginTransaction(IDbConnection svr);
    void commitTransaction(IDbConnection svr);
    void rollbackTransaction(IDbConnection svr);
}
```

Obtaining Connections in Code

DBHelper gets its connections (by default) from the IConnectionFeeder implementation handed to it upon its creation. The following code is an example of how cleanly a simple DELETE can be done …

```
using (DBHelper dbHelper = new DBHelper())
{
    dbHelper.exec("DELETE MyTable Where ID=1");
}
```

… or as a parameterize statement …

```
using (DBHelper dbHelper = new DBHelper())
{
    dbHelper.exec("DELETE MyTable Where ID=?", 1);
}
```

The same code using only DB-level calls is not nearly as clean. There is no need to allocate connection variables, etc.

Nested Connections

An instance can also be passed to subroutines. Consider the following code where the connection is opened explicitly. Even though both the main method and the subroutine create a connection through DBHelper, only 1 connection is made! DBHelper remembers the nesting construct and acts accordingly. This allows you to write code that functions efficiently either nested, standalone, or both.

```
void mainFunc()
{
    using (DBHelper dbHelper = new DBHelper())
    {
        dbHelper.connect();  // Usually don't need to connect explicitly, shown to illustrate.

        for (int = 1; I <= 10; i++)
            subFunc(dbHelper);

        dbHelper.close();
    }
}

void subFunc(DBHelper dbHelper)
{
    dbHelper.connect();
    db.exec("INSERT .....blah.....);
    dbHelper.close();
}
```

Simple ORM

There are two classes to handle records from every IBMi table that we use. Their names follow this pattern:

OEP40 – POCO containing the data from a single row

OEP40File – ORM wrapper for accessing the table easily, plus some simple validation methods.

Both of these are machine-generated in order to keep them consistent and conformant to the base class specification.

POCO Base Class

Every POCO has a common base to tie them all together for operations that they have in common.

```
public abstract class BeanBase
{
    public abstract void Validate();
    public abstract void TrimAllStrings();
    public abstract void NormalizeNULLStrings();
    public abstract Dictionary<String, Object> TakeLogicalSnapshot();
    public abstract Dictionary<String, Object> TakePhysicalSnapshot();
    public abstract bool hasChanged();
    public abstract BeanBase Duplicate();


    #region ORM-Related Values
    private Dictionary<String, Object> logSnapshotAtLoadTime = new Dictionary<String, Object>();
    private Dictionary<String, Object> physSnapshotAtLoadTime = new Dictionary<String, Object>();
    #endregion
}
```

Example POCO

```
public abstract class OEP87 : BeanBase
{
    #region Properties
    public String CompanyCode { get; set; }
    public String OrderNumber { get; set; }
    public String ActionTaken { get; set; }
    public String NewHoldCode { get; set; }
    public String OldHoldCode { get; set; }
    public String Program { get; set; }
    public decimal BatchActionDate { get; set; }
    public decimal BatchActionTime { get; set; }
        #endregion

    #region Validation Methods
    /// <summary>
    /// Performs basic validation on the object.  It looks for NULL values
    /// and performs range checking based on the database metadata.
    /// </summary>
    public override void Validate()
    {
        if (ActionTaken == null)
        {
            throw new NoNullAllowedException("'OEP87.ActionTaken (ACTN87)' cannot be NULL");
        }
        else if (ActionTaken.Length > 2)
        {
            throw new InvalidColumnSizeException("'OEP87.ActionTaken (ACTN87)' cannot be wider than 2 char(s)");
        }

        if (CompanyCode == null)
        {
            throw new NoNullAllowedException("'OEP87.CompanyCode (CONO87)' cannot be NULL");
        }
        else if (CompanyCode.Length > 2)
        {
            throw new InvalidColumnSizeException("'OEP87.CompanyCode (CONO87)' cannot be wider than 2 char(s)");
        }

        if (BatchActionDate > 9999999L)
        {
            throw new InvalidColumnSizeException("'OEP87.BatchActionDate (DATE87)' cannot be greater than 9999999");
        }

        if (NewHoldCode == null)
        {
            throw new NoNullAllowedException("'OEP87.NewHoldCode (NEWH87)' cannot be NULL");
        }
        else if (NewHoldCode.Length > 2)
```

```
        {
            throw new InvalidColumnSizeException("'OEP87.NewHoldCode (NEWH87)' cannot be wider than 2 char(s)");
        }

        if (OldHoldCode == null)
        {
            throw new NoNullAllowedException("'OEP87.OldHoldCode (OLDH87)' cannot be NULL");
        }
        else if (OldHoldCode.Length > 2)
        {
            throw new InvalidColumnSizeException("'OEP87.OldHoldCode (OLDH87)' cannot be wider than 2 char(s)");
        }

        if (OrderNumber == null)
        {
            throw new NoNullAllowedException("'OEP87.OrderNumber (ORDN87)' cannot be NULL");
        }
        else if (OrderNumber.Length > 7)
        {
            throw new InvalidColumnSizeException("'OEP87.OrderNumber (ORDN87)' cannot be wider than 7 char(s)");
        }

        if (Program == null)
        {
            throw new NoNullAllowedException("'OEP87.Program (PRGM87)' cannot be NULL");
        }
        else if (Program.Length > 10)
        {
            throw new InvalidColumnSizeException("'OEP87.Program (PRGM87)' cannot be wider than 10 char(s)");
        }

        if (BatchActionTime > 999999L)
        {
            throw new InvalidColumnSizeException("'OEP87.BatchActionTime (TIME87)' cannot be greater than 999999");
        }
    }

    /// <summary>
    /// Ensures that all String fields do not exceed max length, and at the.
    /// same time trims any excess right whitespaces (which usually not significant).
    /// </summary>
    public override void TrimAllStrings()
    {
        if (ActionTaken != null)
        {
            ActionTaken = ActionTaken.TrimEnd();
            if (ActionTaken.Length > 2)
            {
                ActionTaken = ActionTaken.Substring(0, 2);
            }
        }

        if (CompanyCode != null)
        {
            CompanyCode = CompanyCode.TrimEnd();
            if (CompanyCode.Length > 2)
            {
                CompanyCode = CompanyCode.Substring(0, 2);
            }

        }

        if (NewHoldCode != null)
        {
            NewHoldCode = NewHoldCode.TrimEnd();
            if (NewHoldCode.Length > 2)
            {
                NewHoldCode = NewHoldCode.Substring(0, 2);
            }

        }

        if (OldHoldCode != null)
        {
            OldHoldCode = OldHoldCode.TrimEnd();
            if (OldHoldCode.Length > 2)
```

```
            {
                OldHoldCode = OldHoldCode.Substring(0, 2);
            }
        }

        if (OrderNumber != null)
        {
            OrderNumber = OrderNumber.TrimEnd();
            if (OrderNumber.Length > 7)
            {
                OrderNumber = OrderNumber.Substring(0, 7);
            }
        }

        if (Program != null)
        {
            Program = Program.TrimEnd();
            if (Program.Length > 10)
            {
                Program = Program.Substring(0, 10);
            }
        }
    }

    /// <summary>
    /// Ensures that no String fields are not NULL by replacing them with empty values if they are
    /// currently NULL.
    /// </summary>
    public override void NormalizeNULLStrings()
    {
        if (ActionTaken == null)
        {
            ActionTaken = "";
        }

        if (CompanyCode == null)
        {
            CompanyCode = "";
        }

        if (NewHoldCode == null)
        {
            NewHoldCode = "";
        }

        if (OldHoldCode == null)
        {
            OldHoldCode = "";
        }

        if (OrderNumber == null)
        {
            OrderNumber = "";
        }

        if (Program == null)
        {
            Program = "";
        }
    }

    #endregion

    #region ORM-Related Methods

    public override Dictionary<String, Object> TakeLogicalSnapshot()
    {
        Dictionary<String, Object> resultSet = new Dictionary<String, Object>();
        resultSet["ActionTaken"] = ActionTaken;
        resultSet["CompanyCode"] = CompanyCode;
        resultSet["BatchActionDate"] = BatchActionDate;
        resultSet["NewHoldCode"] = NewHoldCode;
        resultSet["OldHoldCode"] = OldHoldCode;
        resultSet["OrderNumber"] = OrderNumber;
        resultSet["Program"] = Program;
        resultSet["BatchActionTime"] = BatchActionTime;
```

```
        return resultSet;
    }

    public override Dictionary<String, Object> TakePhysicalSnapshot()
    {
        Dictionary<String, Object> resultSet = new Dictionary<String, Object>();
        resultSet["ACTN87"] = ActionTaken;
        resultSet["CONO87"] = CompanyCode;
        resultSet["DATE87"] = BatchActionDate;
        resultSet["NEWH87"] = NewHoldCode;
        resultSet["OLDH87"] = OldHoldCode;
        resultSet["ORDN87"] = OrderNumber;
        resultSet["PRGM87"] = Program;
        resultSet["TIME87"] = BatchActionTime;
        return resultSet;
    }

    /// <summary>
    /// Generates a list of physical column names that have changed in the bean since the record
    /// was first loaded.  It is based on content comparison.
    /// </summary>
    public virtual List<String> GetChangedColumns()
    {
        List<String> dirtyColNames = new List<String>();

        if (HasColumnChanged("ACTN87"))
            dirtyColNames.Add("ACTN87");
        if (HasColumnChanged("CONO87"))
            dirtyColNames.Add("CONO87");
        if (HasColumnChanged("DATE87"))
            dirtyColNames.Add("DATE87");
        if (HasColumnChanged("NEWH87"))
            dirtyColNames.Add("NEWH87");
        if (HasColumnChanged("OLDH87"))
            dirtyColNames.Add("OLDH87");
        if (HasColumnChanged("ORDN87"))
            dirtyColNames.Add("ORDN87");
        if (HasColumnChanged("PRGM87"))
            dirtyColNames.Add("PRGM87");
        if (HasColumnChanged("TIME87"))
            dirtyColNames.Add("TIME87");

        return dirtyColNames;
    }

    public override bool hasChanged()
    {
        return (GetChangedColumns().Count > 0);
    }

    /// <summary>
    /// Returns TRUE if the given physical column has changed since the record was loaded.
    /// It is based on content comparison of current vs. snapshot.
    /// </summary>
    public virtual bool HasColumnChanged(String columnName)
    {
        Dictionary<String, Object> physSnapshot = GetPhysicalSnapshotAtLoadTime();

        if (!physSnapshot.ContainsKey(columnName))
        {
            return true;
        }

        Object snapshotValue = physSnapshot[columnName];

        if (columnName == "ACTN87")
            return !snapshotValue.Equals(ActionTaken);
        else if (columnName == "CONO87")
            return !snapshotValue.Equals(CompanyCode);
        else if (columnName == "DATE87")
            return !snapshotValue.Equals(BatchActionDate);
        else if (columnName == "NEWH87")
            return !snapshotValue.Equals(NewHoldCode);
        else if (columnName == "OLDH87")
            return !snapshotValue.Equals(OldHoldCode);
        else if (columnName == "ORDN87")
```

```
            return !snapshotValue.Equals(OrderNumber);
        else if (columnName == "PRGM87")
            return !snapshotValue.Equals(Program);
        else if (columnName == "TIME87")
            return !snapshotValue.Equals(BatchActionTime);
        else
            return false;
    }
    #endregion
```

Core ORM Methods

There are two classes to handle records from every IBMi table that we use in OW. Their names follow this pattern:

OEP40 – POCO containing the data from a single row

OEP40File – ORM wrapper for accessing the table easily.

Reading an **OEP40** record looks something like this:

```
OEP40 oep40 = OEP40File.QueryByOrderNumber(dbHelper, "ABCD123", true);

Console.WriteLine("Customer Number is:");
Console.WriteLine(oep40.CustomerNumber);
```

Operations in the ORM are almost always key-based, though custom queries are possible.

POCO / ORM Generation

The following is the base class for all **[xxx]File** such as **OEP40File**, which implements ORM methods for POCO **OEP40**. The abstract methods are machine-generated for each type of POCO with specific logic.

```
// Enum representing what we expect at the end of an UPDATE/DELETE.
// Failure of the operation to meet this expectation could result in an exception.
public enum UpdateExpectationMode
{
    IgnoreRowsAffected,
    ExpectOneRowOrFewerAffected, /* The default */
    ExpectOneRowAffected,
    ExpectMultipleRowsAffected
};

/// <summary>
/// Base class containing core functions for serializing beans to/from the the
/// database.
/// </summary>
/// <typeparam name="T">DB record POCO  derived from BeanBase that this instance can
/// serialize to/from the DB.</typeparam>
public abstract class BeanMappingBase<T> where T : BeanBase, new()
{
    #region Protected Values
    private static string _schemaName = "";
    #endregion

    #region Abstract Methods (implemented by bean-specific mapping classes)
    public abstract String GetTableName();
    public abstract String GetColumnNames();
    public abstract String GetKeyNames();
    protected abstract String GetColumnListParameterized(String delimiter);
    protected abstract String GetKeyListParameterized(String delimiter);
    protected abstract String GetInsertSQL();
    protected abstract UpdateParms GetUpdateSQL(T bean, bool changedFieldsOnly);
    protected abstract String GetCheckExistenceSQL();
    public abstract Object[] GetColumnValues(T bean);
    public abstract Object[] GetKeyValues(T bean);
    protected abstract bool IsKeyFullyPresent(T bean);
    protected abstract String WhichKeysMissing(T bean);
    protected abstract void SetValueByColumn(T bean, String colName, Object colValue, bool crashIfNoSuchColumn);
    public abstract DBHelper.QuickQueryResult GetTableMetadata();
    #endregion
```

```
    #region Insert/Update/Delete Methods

    /// Inserts the given record into the database.  Throws an error if any
    /// "key" fields are detected as missing (this is not full-proof).  Defaults
    /// NULL fields to 0's and blanks, as appropriate to the type.  Validates
    /// that no fields exceed their maximum range.
    public static void Insert(IDBHelper dbHelper, T bean)
    {
        //Stopwatch sw = new Stopwatch();
        //sw.Start();

        BeanMappingBase<T> dummyMapper = GetDummyMapperInstance(typeof(T));
        try
        {
            // Make sure no key fields are missing.
            String whichKeysAreMissing = dummyMapper.WhichKeysMissing(bean);
            if (whichKeysAreMissing != null)
            {
                throw new CdwException("One or more key fields for " + dummyMapper.GetTableName()
                                       + " are missing, including: " + whichKeysAreMissing);
            }

            // Protect against writing anything without a declared key.
            if (dummyMapper.GetKeyValues(bean).Length == 0)
            {
                throw new CdwException("Key fields not declared for " + dummyMapper.GetTableName());
            }

            // Format fields within the field. For "trim mode", hack off String fields
            // that are too long.
            bean.NormalizeNULLStrings();
            bean.TrimAllStrings();

            // Make sure we are okay now.
            bean.Validate();

            // Insert!
            try
            {
                dbHelper.exec(dummyMapper.GetInsertSQL(), dummyMapper.GetColumnValues(bean));
            }
            catch (Exception innerEx)
            {
                var param = dummyMapper.GetColumnValues(bean);
                StringBuilder sb = new StringBuilder();

                foreach (var obj in param)
                {
                    if (sb.Length > 0)
                    {
                        sb.Append(", ");
                    }
                    sb.Append(obj.ToString());
                }
                throw new CdwException("Error executing SQL:\n" + dummyMapper.GetInsertSQL() + "\r\n Column Values: " + sb
.ToString(), innerEx);
            }

            // Make sure we get the latest snapshots on this updated guy.
            // This ensures that subsequent updates on the same object work okay.
            bean.UpdateSnapshots();

            //sw.Stop();
            //Debug.WriteLine("InsertBean (" + bean.TableName + "): " + sw.Elapsed + "        ....       " + bean.GetInser
tSQL());
        }
        catch (Exception outerEx)
        {
            throw new CdwException("Error inserting " + dummyMapper.GetTableName(), outerEx);
        }
    }

    /// Updates the given bean in the database, setting the expectation that one
    /// or fewer rows will be affected by the operation.
    public static int Update(IDBHelper dbHelper, T bean)
```

```
    {
        return Update(dbHelper, bean, UpdateExpectationMode.ExpectOneRowOrFewerAffected);
    }

    /// Updates the given bean in the database.  The object is actually updated
    /// *only* if the underlying row has not changed since it was first read from
    /// the database.  Throws an error if the number of rows actually affected does
    /// not match the expectation passed in.
    public static int Update(IDBHelper dbHelper, T bean, UpdateExpectationMode errorMode)
    {
        if (!bean.hasChanged())
        {
            return 0;
        }

        //Stopwatch sw = new Stopwatch();
        //sw.Start();

        BeanMappingBase<T> dummyMapper = GetDummyMapperInstance(typeof(T));
        try
        {
            // Make sure no key fields are missing.
            String whichKeysAreMissing = dummyMapper.WhichKeysMissing(bean);
            if (whichKeysAreMissing != null)
            {
                throw new CdwException("One or more key fields for " + dummyMapper.GetTableName()
                                       + " are missing, including: " + whichKeysAreMissing);
            }

            // Format fields within the field. For "trim mode", hack off String fields
            // that are too long.
            bean.NormalizeNULLStrings();
            bean.TrimAllStrings();

            // Make sure we are okay now.
            bean.Validate();

            // No known key fields?  As a safety measure, bomb out so we don't
            // accidentally overwrite out the entire table.
            if (dummyMapper.GetKeyValues(bean).Length == 0)
            {
                throw new CdwException("Error updating " + dummyMapper.GetTableName()
                                       + ": No key fields defined");
            }

            // Update!
            UpdateParms updateParms = dummyMapper.GetUpdateSQL(bean, true); // Changed fields only!

            // A NULL parm value means no fields have changed.
            if (updateParms == null)
            {
                if (errorMode == UpdateExpectationMode.ExpectOneRowAffected)
                    throw new CdwException("No updates required but one row was expected to be expected");
                else if (errorMode == UpdateExpectationMode.ExpectMultipleRowsAffected)
                    throw new CdwException("No updates required but multiple rows were expected to be affected");
                else
                {
                    return 0;
                }
            }

            int rowsAffected;
            try
            {
                rowsAffected = dbHelper.exec(updateParms.updateSQL, updateParms.updateParms);
            }
            catch (Exception innerEx)
            {
                throw new CdwException(
                    string.Format("Error executing SQL:\n{0}\nUpdate Params Used: {1}\n", updateParms.updateSQL, string.Jo
in(",", updateParms.updateParms)),
                    innerEx);
            }

            // Make sure we get the latest snapshots on this updated guy.
            // This ensures that subsequent updates on the same object work okay.
```

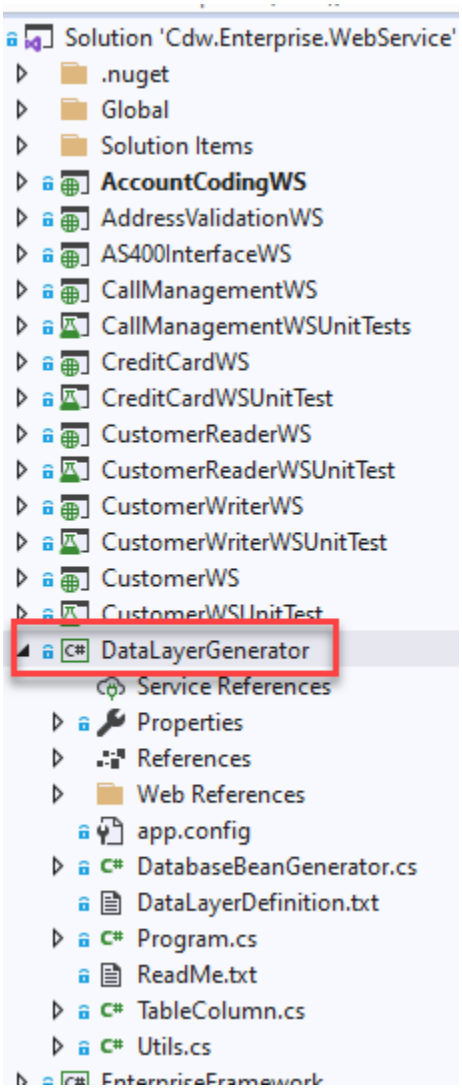
```
            bean.UpdateSnapshots();

            // Throw errors where appropriate.
            if ((errorMode == UpdateExpectationMode.ExpectOneRowOrFewerAffected) &&
                 (rowsAffected > 1))
            {
                throw new CdwException("Multiple rows affected when no more than 1 was expected");
            }
            else if ((errorMode == UpdateExpectationMode.ExpectOneRowAffected) &&
                 (rowsAffected == 0))
            {
                throw new CdwException("No rows affected when 1 was expected");
            }
            else if ((errorMode == UpdateExpectationMode.ExpectOneRowAffected) &&
                        (rowsAffected > 1))
            {
                throw new CdwException("Multiple rows affected when only 1 was expected");
            }
            else if ((errorMode == UpdateExpectationMode.ExpectMultipleRowsAffected) &&
                        (rowsAffected == 0))
            {
                throw new CdwException("No rows affected when multiple rows were expected");
            }
            else if ((errorMode == UpdateExpectationMode.ExpectMultipleRowsAffected) &&
                      (rowsAffected == 1))
            {
                throw new CdwException("Only 1 row affected when multiple rows were expected");
            }

            //sw.Stop();
            //Debug.WriteLine("UpdateBean (" + bean.TableName + "): " + sw.Elapsed + "        ....       " + updateParms.u
pdateSQL);

            // Return rows affected.
            return rowsAffected;

        }
        catch (Exception outerEx)
        {
            dbHelper.rollback();
            throw new CdwException("Error updating " + dummyMapper.GetTableName(), outerEx);
        }
    }

    /// Updates a business object in the database.  But only if they changed since the
    /// original snapshot, excluding change stamp fields.
    public static bool UpdateIfChanged(IDBHelper dbHelper, T bean)
    {
        bool shouldUpdate = true;

        // Get the original DB snapshot for this sucker.
        Dictionary<String, Object> originalSnapshot = bean.GetLogicalSnapshotAtLoadTime();
        if (originalSnapshot != null)
        {
            // Take a current snapshot.
            Dictionary<String, Object> currentSnapshot = bean.TakeLogicalSnapshot();

            // Remove fields that should not be part of the comparison.
            originalSnapshot.Remove("changeDate");
            currentSnapshot.Remove("changeDate");

            originalSnapshot.Remove("changeTime");
            currentSnapshot.Remove("changeTime");

            originalSnapshot.Remove("changeUser");
            currentSnapshot.Remove("changeUser");

            originalSnapshot.Remove("changeProgram");
            currentSnapshot.Remove("changeProgram");

            originalSnapshot.Remove("dateLastChanged");
            currentSnapshot.Remove("dateLastChanged");

            // See if they are otherwise the same.
            shouldUpdate = !currentSnapshot.Equals(originalSnapshot);
        }
```

```
        // Update as appropriate.
        if (shouldUpdate)
        {
            Update(dbHelper, bean);
        }
        return shouldUpdate;
    }

    /**
            * Deletes the given bean in the database, setting the expectation that
            * exactly one row will be affected by the operation.
            *
            * @param bean
            * @return Number of rows actually affected.
            * @
            */
    public static int Delete(IDBHelper dbHelper, T bean)
    {
        return Delete(dbHelper, bean, UpdateExpectationMode.ExpectOneRowAffected);
    }

    /// Deletes the given bean in the database.  The object is actually deleted
    /// *only* if the underlying row has not changed since it was first read from
    /// the database.  Throws an error if the number of rows actually affected does
    /// not match the expectation passed in.
    public static int Delete(IDBHelper dbHelper, T bean, UpdateExpectationMode errorMode)
    {
        BeanMappingBase<T> dummyMapper = GetDummyMapperInstance(typeof(T));
        try
        {
            // Make sure no key fields are missing.
            if (!dummyMapper.IsKeyFullyPresent(bean))
            {
                throw new CdwException("Error deleting " + dummyMapper.GetTableName() + ": One or more key fields is missi
ng");
            }

            // Format fields within the field.
            bean.NormalizeNULLStrings();
            bean.Validate();

            // No known key fields?  As a safety measure, bomb out so we don't
            // accidentally delete an the entire table!
            if (dummyMapper.GetKeyValues(bean).Length == 0)
            {
                throw new CdwException("Error deleting " + dummyMapper.GetTableName()
                                       + ": No key fields defined");
            }

            // Make a list of values with the full column list to be used in the
            // WHERE clause.  Using all the values (rather than just the keys)
            // is a safety measure.
            Dictionary<String, Object> physSnapshotAtLoadTime =
                 bean.GetPhysicalSnapshotAtLoadTime();

            String whereKeyList = "";
            List<Object> whereClauseValues = new List<Object>();

            foreach (String columnName in physSnapshotAtLoadTime.Keys)
            {
                if (whereKeyList.Length > 0)
                {
                    whereKeyList += " AND ";
                }
                whereKeyList += columnName + "=?";

                whereClauseValues.Add(physSnapshotAtLoadTime[columnName]);
            }

            Object[] whereClauseValueArray = whereClauseValues.ToArray();

            // Form the SQL.
            String sql = "DELETE FROM " + dummyMapper.GetTableName()
                         + " WHERE " + whereKeyList;
```

```
            // Delete!
            int rowsAffected = dbHelper.exec(sql, whereClauseValueArray);

            // Throw errors where appropriate.
            if ((errorMode == UpdateExpectationMode.ExpectOneRowOrFewerAffected) &&
                 (rowsAffected > 1))
            {
                throw new CdwException("Multiple rows affected when no more than 1 was expected");
            }
            else if ((errorMode == UpdateExpectationMode.ExpectOneRowAffected) &&
                 (rowsAffected == 0))
            {
                throw new CdwException("No rows affected when 1 was expected");
            }
            else if ((errorMode == UpdateExpectationMode.ExpectOneRowAffected) &&
                        (rowsAffected > 1))
            {
                throw new CdwException("Multiple rows affected when only 1 was expected");
            }
            else if ((errorMode == UpdateExpectationMode.ExpectMultipleRowsAffected) &&
                        (rowsAffected == 0))
            {
                throw new CdwException("No rows affected when multiple rows were expected");
            }
            else if ((errorMode == UpdateExpectationMode.ExpectMultipleRowsAffected) &&
                      (rowsAffected == 1))
            {
                throw new CdwException("Only 1 row affected when multiple rows were expected");
            }

            return rowsAffected;

        }
        catch (Exception ex)
        {
            throw new CdwException("Error deleting " + dummyMapper.GetTableName(), ex);
        }
    }
    #endregion

    #region General Query Methods
    … (more useful stuf) …
    #endregion
```

Code Generator

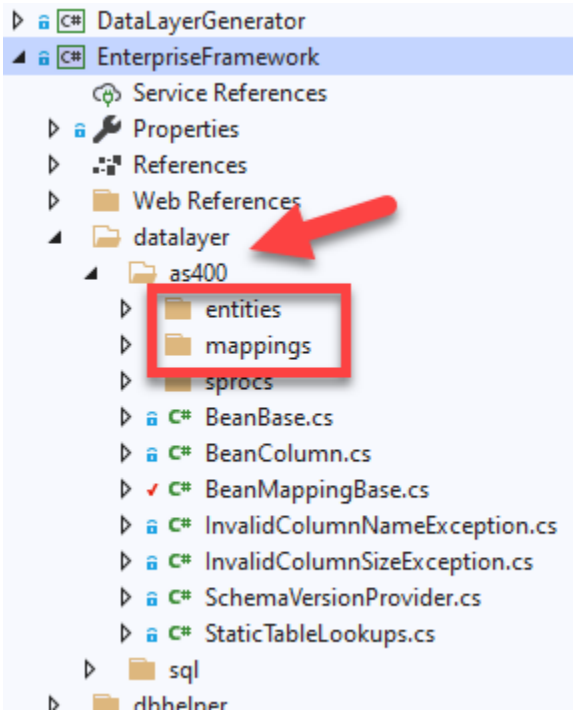
The existing POCO generator exists inside the EnterpriseWebServices solution, the same one where Order Reader lives. The project is called DataLayerGenerator:



Specifying POCOs to Generate

If the application is run with a specific param, such as OEP40…

… it will generate **OEP40.cs** and **OEP40File.cs**. They are automatically placed in these directories in the Enterprise Framework code, overlaying them if already present:



Because of this behavior, they can simply be added to the EF project if not already present, then checked in.

Executing Stored Procedures

The RPG Wrapper Class

Each stored procedure used by OW is encapsulated in a dedicated class designed to be intuitive and easy to use, all of which derive from `RPGWrapperBase`. The `RPGWrapperBase` class contains most of the mechanics of calling the sproc so that the derived type has little actual work to perform directly. Parameter representation…

```
public enum SprocParmTransferMode { Input, Output, InputOutput };

public class StoredProcedureParm
{
    public SprocParmTransferMode xferMode { get; set; }
    public DbType sqlType { get; set; }
    public String parmName { get; set; }
    public Object value { get; set; }
}
```

… and definition for the sproc itself:

```
public class StoredProcedure : IDisposable
{
    public String sprocName { get; set; }
    public String sprocDescription { get; set; }
    public List<StoredProcedureParm> appParmList { get; set; }
    public bool returnRawResultSet { get; set; }
    public IQueryResult2D resultSet { get; set; }   // This one is wrapped by dbHelper.
    public List<IQueryResult2D> multiResultSet { get; set; }
    public DBResultSet resultSetRaw { get; set; }


    public void addInputParameter(String parmName, Object parmValue)
    {
        StoredProcedureParm parm = new StoredProcedureParm();
        parm.xferMode = SprocParmTransferMode.Input;
        parm.parmName = parmName;
        parm.value = parmValue;
        appParmList.Add(parm);
    }
```

```
public void addOutputParameter(String parmName)
{
    StoredProcedureParm parm = new StoredProcedureParm();
    parm.xferMode = SprocParmTransferMode.Output;
    parm.parmName = parmName;
    appParmList.Add(parm);
}

public void addInputOutputParameter(String parmName, Object parmValue, DbType sqlType)
{
    StoredProcedureParm parm = new StoredProcedureParm();
    parm.xferMode = SprocParmTransferMode.InputOutput;
    parm.sqlType = sqlType;
    parm.parmName = parmName;
    parm.value = parmValue;
    appParmList.Add(parm);
}

public Dictionary<String, Object> run(IDBHelper dbHelper, String libraryOverride, bool returnsResultSet)
{
    // Now the main stuff.
    IDbConnection conn = null;
    IDbCommand spCommand = null;
    IDataReader dataReader = null;

    try
    {
        // Set up the SQL for the call.
        String absSprocName;
        if (libraryOverride != null)
            absSprocName = libraryOverride + getSQLSeparator() + sprocName;
        else
            absSprocName = sprocName;

        String callSQL = "CALL " + absSprocName + "(";
        for (int i = 0; i < appParmList.Count; i++)
        {
            if (i > 0)
            {
                callSQL += ",";
            }
            callSQL += "?";
        }
        callSQL += ")";

        // Set up parms.

        // if dbHelper already has a connection this won't open a new one
        conn = dbHelper.connect(callSQL);
        spCommand = conn.CreateCommand();
        spCommand.CommandText = callSQL;
        spCommand.CommandTimeout = (int) _sprocTimeoutInterval.TotalMilliseconds;
        dbHelper.getConnectionFeeder().statementPreProcess(0, spCommand);

        for (int i = 0; i < appParmList.Count; i++)
        {
            StoredProcedureParm appParm = appParmList[i];

            IDbDataParameter spParm = spCommand.CreateParameter();
            spParm.ParameterName = appParm.parmName;

            if (appParm.xferMode == SprocParmTransferMode.Output)
            {
                spParm.Direction = ParameterDirection.Output;
            }
            else if (appParm.xferMode == SprocParmTransferMode.InputOutput)
            {
                spParm.Direction = ParameterDirection.InputOutput;
                spParm.Value = appParm.value;
                spParm.DbType = appParm.sqlType;
            }
            else if (appParm.xferMode == SprocParmTransferMode.Input)
            {
                spParm.Direction = ParameterDirection.Input;
```

```
                spParm.Value = appParm.value;

                if (appParm.value is decimal)
                    spParm.DbType = DbType.Decimal;
                else if (appParm.value is int)
                    spParm.DbType = DbType.Int32;
                else
                    spParm.DbType = DbType.String;
            }

            spCommand.Parameters.Add(spParm);
        }

        // Run the procedure!
        if (returnsResultSet)
        {
            DateTime startTime = DateTime.Now;
            dataReader = spCommand.ExecuteReader();
            dbHelper.AddExecHistoryEntry(conn, callSQL, startTime, DateTime.Now);
            dbHelper.AddBlackBoxStat(absSprocName, sprocDescription, startTime, DateTime.Now);

            // Return a raw ResultSet if requested.
            if (returnRawResultSet)
            {
                this.resultSetRaw = new DBResultSet(dbHelper, conn, spCommand, dataReader, conn.GetType(), null);
            }
            // Convert to a wrapped result set.
            else
            {
                this.resultSet = DBHelper.convertDateReaderToQuickResult(spCommand, dataReader);
                multiResultSet.Add(this.resultSet);

                // We support multiple result sets.
                while (dataReader.NextResult())
                {
                    IQueryResult2D rs = DBHelper.convertDateReaderToQuickResult(spCommand, dataReader);
                    multiResultSet.Add(rs);
                }
            }
        }
        else
        {
            DateTime startTime = DateTime.Now;
            spCommand.ExecuteNonQuery();
            dbHelper.AddExecHistoryEntry(conn, callSQL, startTime, DateTime.Now);
            dbHelper.AddBlackBoxStat(absSprocName, sprocDescription, startTime, DateTime.Now);
            this.resultSet = null;
        }

        // Invoke post-processing.
        if (PostCallProcess != null)
        {
            PostCallProcess();
        }

        dbHelper.getConnectionFeeder().statementPostProcess(0, spCommand);

        // Process the output values.
        Dictionary<String, Object> results = new Dictionary<String, Object>();

        foreach (IDbDataParameter parm in spCommand.Parameters)
        {
            if ((parm.Direction == ParameterDirection.Output) ||
                (parm.Direction == ParameterDirection.InputOutput))
            {
                results[parm.ParameterName] = parm.Value;
            }
        }

        // Sign on the dotted line...
        spCommand.Parameters.Clear();
        spCommand.Dispose();
        spCommand = null;

        // Kick outta here!
        return results;
```

```
        }
        catch (Exception ex)
        {
            // DB connection cleanup.
            if (spCommand != null)
            {
                spCommand.Parameters.Clear();
                spCommand.Dispose();
                spCommand = null;
            }

            // Form an exception message that includes parms and input values.
            StringBuilder sb = new StringBuilder();
            sb.AppendLine("Error calling " + sprocName + " with parms:");
            for (int i = 0; i < appParmList.Count; i++)
            {
                String displayValue = (appParmList[i].value == null) ? "null" : appParmList[i].value.ToString();
                sb.AppendLine("   " + appParmList[i].parmName + " (" + appParmList[i].sqlType + ", " + appParmList[i].xfer
Mode + ") = <" + displayValue + ">");
            }

            // Raaaalllffff!
            throw new RPGWrapperException(sb.ToString(), ex);
        }
        finally
        {
            if (dataReader != null)
            {
                dataReader.Close();
                dataReader.Dispose();
            }

            if (spCommand != null)
            {
                spCommand.Dispose();
            }

            if (conn != null)
            {
                // DBHelper only closes the connection if was opened here.  If was opened prior nothing happens here
                dbHelper.close();
            }
        }
    }
}
```

Example Derived Class for Allocation

Most of these derived classes are currently hand-coded but, because of RPGWrapper plus existing samples, this is typically very easy.

```
public class BO008 : RPGWrapperBase
{
    public class Parms
    {
        public decimal priority { get; set; }     // DECIMAL (2,0),
        public decimal orderDate { get; set; }    // DECIMAL (7,0),
        public decimal orderTime { get; set; }    // DECIMAL (6,0),
        public String companyCode { get; set; }   // CHAR (2),
        public String orderNumber { get; set; }   // CHAR (7),
        public decimal orderLine { get; set; }    // DECIMAL (3,0),
        public String requestType { get; set; }   // CHAR (1),
        public String processor { get; set; }     // CHAR (3),
    }

    public class Result
    {
        public String returnCode { get; set; }  // CHAR (7)

        public String response { get; set; }         // CHAR (7)

        public Result()
        {
            returnCode = "";
```

```
        }
    }

    public override string Description
    {
        get
        {
            return "SOA";
        }
    }

    public static Result execute(IDBHelper dbHelper, String lib, Parms parms)
    {
        StoredProcedure sproc = new StoredProcedure("DBBO008");

        // Get the description of the sproc.  This is a hack - should not use reflection ideally.
        Type t = MethodBase.GetCurrentMethod().DeclaringType;
        sproc.sprocDescription = GetDescriptionField(t);

        // Set up in/out parameters.  The order of parms should be the
        // same as declared in the sproc definition in order to avoid issues.
        Result result = new Result();

        sproc.addInputOutputParameter("RETURNCODE", result.returnCode, DbType.AnsiString);
        sproc.addInputParameter("PRIORITY", parms.priority);
        sproc.addInputParameter("ORDERDATE", parms.orderDate);
        sproc.addInputParameter("ORDERTIME", parms.orderTime);
        sproc.addInputParameter("COMPANYCODE", parms.companyCode);
        sproc.addInputParameter("ORDERNUMBER", parms.orderNumber);
        sproc.addInputParameter("ORDERLINE", parms.orderLine);
        sproc.addInputParameter("REQUESTTYPE", parms.requestType);
        sproc.addInputParameter("PROCESSOR", parms.processor);
        sproc.addOutputParameter("RESPONSE");

        // run the sproc.
        Dictionary<String, Object> retVal = sproc.run(dbHelper, lib);

        // Obtain and return the result.
        result.returnCode = (String)retVal["RETURNCODE"];
        result.response = (String)retVal["RESPONSE"];

        // Throw an exception on error.
        if ((result.returnCode != null) && (result.returnCode.Trim().Length > 0))
            throw new RPGWrapperException("Error code returned from " + sproc.sprocName + ": " + result.returnCode);
        else
            return result;
    }
}
```

Example Stored Procedure Usage in Code

Here is an example of how to call Allocation:

```
BO008.Parms parms = new BO008.Parms
{
    priority = 50,
    orderDate = 1210101,
    orderTime = 50101,
    companyCode = “01”,
    orderNumber = “ABCD123”,
    orderLine = 0,
    requestType = “A”,
    processor = “SOA”
};

BO008.Result result = BO008.execute(dbHelper, null, parms);

if (result.response.Trim().ToLower() != "pass")
{
    throw new Exception("Allocation failed! The response was " + result.response.Trim());
}
else
{
    throw new Exception("Allocation succeeded!”);
```

```
}
```

Message Queueing Architecture and Features

One of the defining features of the existing OW queue is that it uses a single queue, filtered by properties and status codes. This could instead be handled using multiple, simpler queues.

Features

Dual FIFO/Random-Access Access

The OW queue is handled as a FIFO queue by the OW itself. Messages are popped from the queue in order and "acquired" by an OW. From the perspective of Diag Console, however, the queue is handled as a table, and this is how it can provide a higher-level view of the activity.

Guaranteed delivery

The OW queue features guaranteed delivery of messages, assuming SQL Server remains reliable. It leverages the transactional nature of the database to ensure this. Internally this is implemented using a Status column in the queue table that change as the message travels through processing steps.

Ordered delivery

Messages are generally delivered in order, with a few exceptions that are made to provide enhanced functionality:

- A message might be "paused" due to an exception or some other condition. In this case any other messages can "flow around" it.

- When the message is reactivated, its position in the queue is also restored. Therefore, ordered delivery is observed for any messages were delayed at the same time. This is so that, during widespread OW outages or delays, messages are still processed in a FIFO manner to be fair to customers in terms of when processes such as Allocation occur.

- Subscription order operations are "serialized" and only processed one-at-a-time and only by OW1 (via config). Other non-subscription messages will "flow around" subscriptions. Can be thought of as a separate serialized queue.

Resume/Abort

The mechanism allows messages to be "aborted", which effectively pauses the message but leaves it in queue. When "resumed", the message becomes active and resumes its priority in queue.

Editing Message Content In-Place

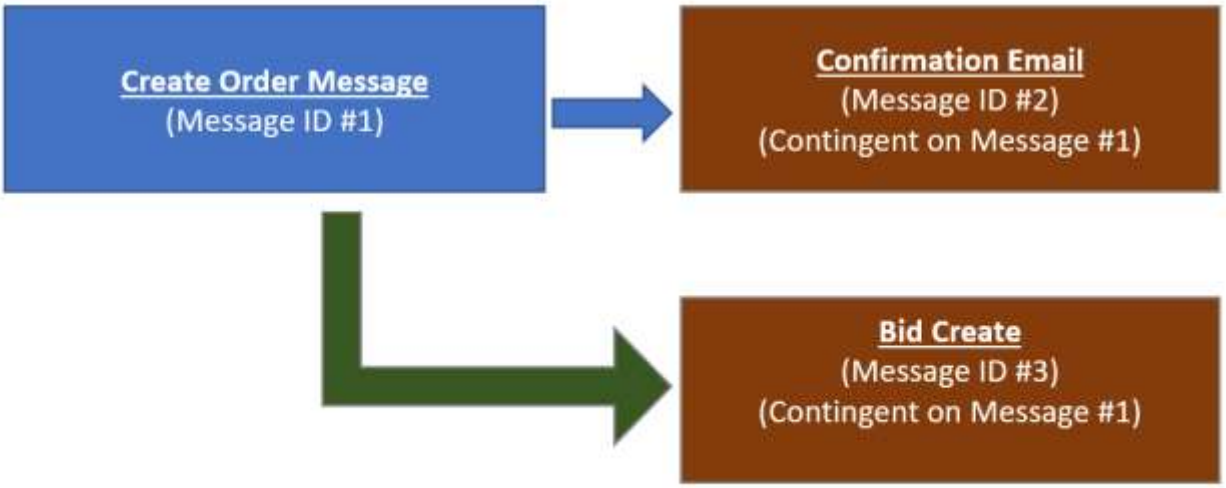
Using Diag Console, the contents of a message (XML) can be edited in-place without affecting its place in the queue. Thus, the queue is both FIFO as well as random-access for Diag Console.

Message Metadata

The core of an OW message is the XML representing the operation being performed as well as the data that it requires. A new order, for example, contains XML representing the order as constructed in SPS as well as properties of the operation.

Message Dependency

The OW queue supports simple dependency (chaining) between messages. Dependent messages will not be processed until processing of the prior message is completed.



In the OW Queue, the message ID is called ComponentID.

Message Idempotence

Messages can be processed more than once if there are errors with their processing. An example of this is where an exception occurs inside the OW during an order creation. The condition can sometimes be fixed and then resumed.

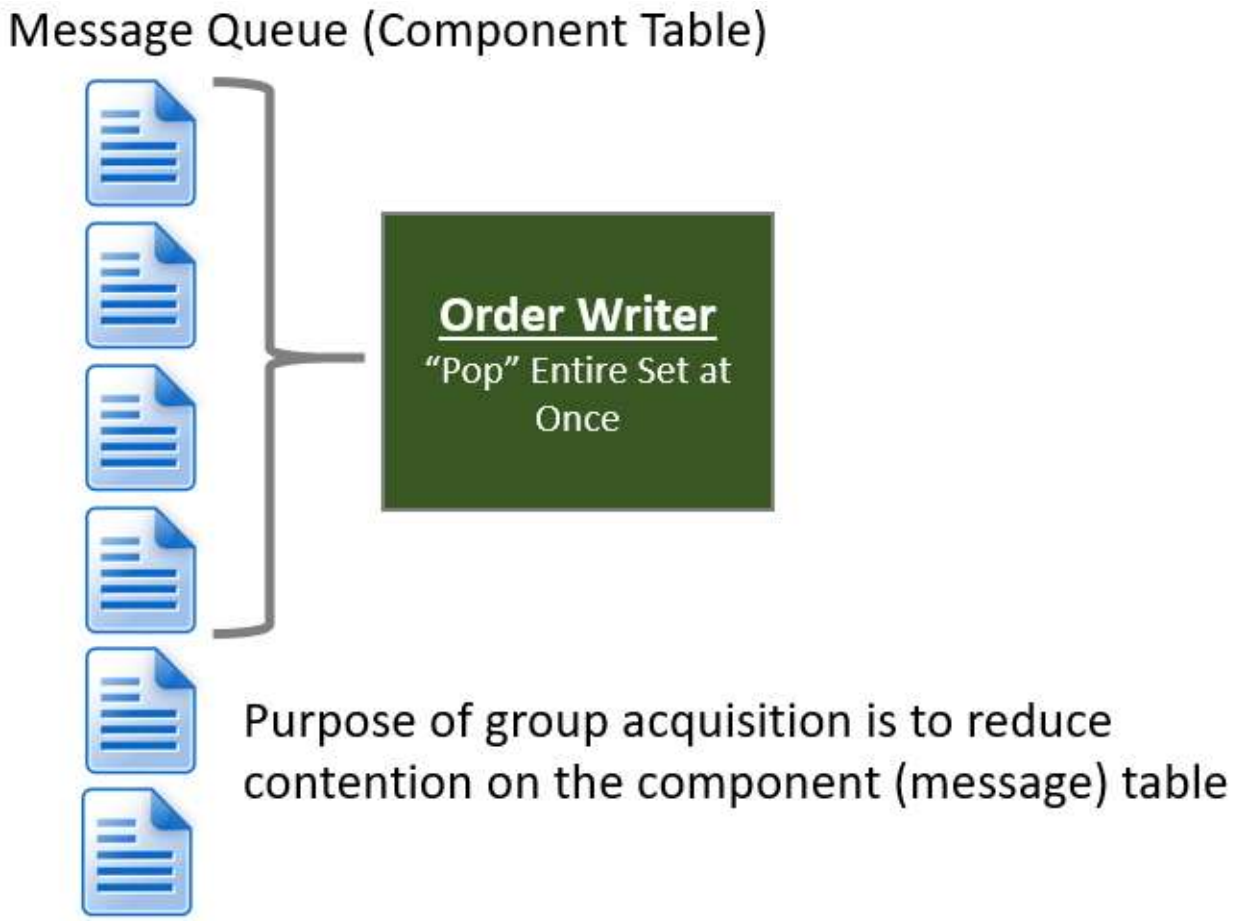
The ability to reprocess a message is supported by the queuing mechanism but could cause problems such as Duplicate Keys inside the OW. However, sometimes it is appropriate because the OW also has logic to skip problematic plugins. Knowledge of the order process is required in order to know when it is okay.

Atomic Message Processing

Messages in the OQ queue are atomic, meaning that OW's obtain them in a transactional-like manner. It is built upon a Status code fields and SQL Server transactions. When "popping" a message from the queue, an OW marks it as "Acquired" transactionally. Other OW's will ignore it once another OW has acquired it.

Group Message Acquisition

Message are currently acquired from the in groups, as depicted below:



Queue concurrency

The default for OW is that it can process 10 messages at one time though this is dynamically configurable. This means that OW can create 10 orders (or other operations) at one time.

Throttling

Message throttling is controlled by the OW and the rate at which it Acquires them from the queue. They are not popped one-by-one.

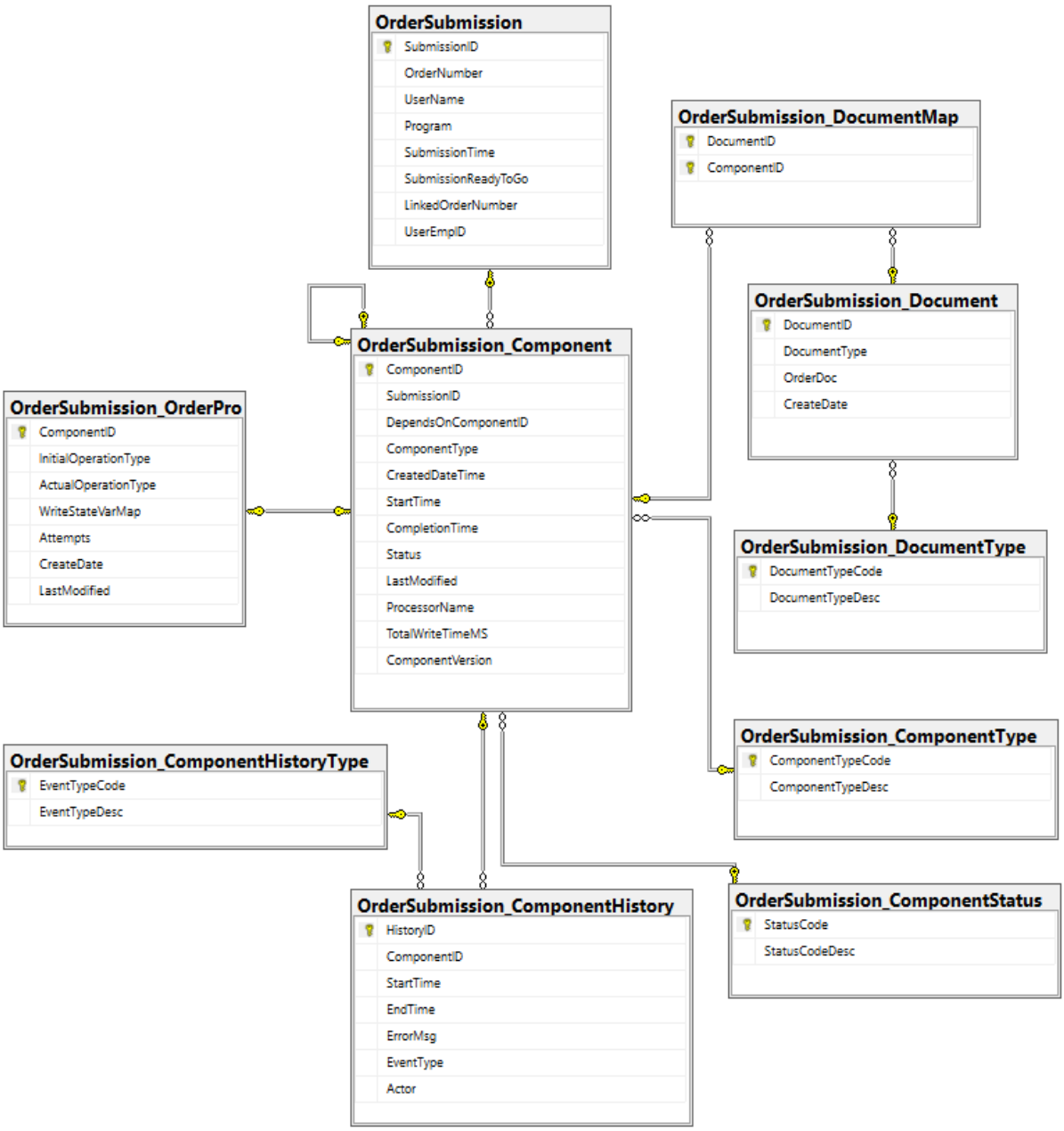
Automatic Retry

OW features an automatic retry mechanism that will "steal" a message that has been left unprocessed for an unexpectedly long period of time by an OW and that OW appears to be done. This can happen if an OW acquires a group of messages and then crashes.

If a message is reprocessed automatically, it retains its original position in the queue and is not placed the back. Therefore things like allocation priorities remain closer to their original values.

Data Structures

The following diagram depicts the queue implementation in SQL Server. There is a `Submission` table acting as an umbrella for an operation such as creating a new order. The `OrderSubmission_Component` table is the main part of the queue while the rest are supporting tables. It contains the metadata for the queue entry, while the `OrderSubmission_Document` holds the actual message contents.



Message Types

ComponentTypeCode	ComponentTypeDesc	Description
A	Amend	An order Amend request. An order is expected to be present in the document table.
B	Bid Create	A Bid Create request. This is normally, but not necessarily, dependent upon another Create or Amend component.
C	Create	A new order creation request. Note that a new Create operation can possibly be turned into an "Amend" by the Order Writer in some rare circumstances. Those situations are described further in this document. An order document is expected to be present in the document table.
E	Email Confirmation	Email confirmation generation request. Normally dependent upon another Create or

		Amendcomponent, but not necessarily. The details of the email (Send To, etc) are contained in the order document XML but can in theory be extracted into a distinct document type.
X	Cancel	An order Cancellation request. An Order Cancel XML document must be present in the document table to successfully process the request.

Message Status Codes

StatusCode	StatusCodeDesc	Description
A	Acquired	An Order Writer instance has "claimed" the operation and will process it as soon as it can. Submission components are assigned in "groups" in order tominimize polling on SQL Server.
C	Complete	Processing on the given component has been completed.
E	Error	The component is in an error state and cannot be processed without intervention from the SPS Order Console and possibly other remediation.
I	In Progress	The Order Writer is currently processing thecomponent.
P	Pending	The component is queued but waiting for completion of another component such as the quote/order portion of an Auto Q2O operation. The status will change to 'R' once the root component's processing has been completed, allowing the secondary component to be Acquired freely.
R	Ready for Processing	The component is clear for processing and ready to be Acquired by an Order Writer instance.
T	Transferred	The component was Transferred to the 'Old MOP',perhaps due to an issue in the 'new MOP'. *This code is no longer used*.
X	Canceled	The component is abandoned. It will not be processed by any Order Writer instances unless it is resume via the SPS Diagnostic Console.

Document Types

DocumentTypeCode	DocumentTypeDesc	Description
BID	Bid Create	A bid creation XML document.
CAN	Cancel	An order cancellation XML document.
EX	Order Ex	An order extension document, used by Create and Amend components. *This type is no longer used*.
ORD	Order	A primary order document, used by Create and Amend components.

Stored Procedures

All MOP integration points, such as submitting orders, are exposed as SQL Server stored procedures. The following major stored procedures comprise the MOP's high-level "API":

Stored Procedure Signature	Description	Used By
PROCEDURE OrderSubmission_SubmitOrderToMOP (@OrderNumber char(7), @LinkedOrderNumber char(7), @OperationType char(1), @RootComponentID int, @UserName char(20), @Program char(20), @OrderDoc xml, @OrderExDoc xml, @BidDoc xml, @IsEmailPresent bit, @GeneratedSubmissionID int OUT, @GeneratedOrderComponentID int OUT, @GeneratedBidComponentID int OUT, @GeneratedEmailComponentID int OUT, @GeneratedOrderDocumentID int OUT, @GeneratedOrderExDocumentID int OUT, @GeneratedBidDocumentID int OUT)	Encapsulates logic required for submitting an order document to MOP. The GeneratedXXX output values are information can be ignored unless the caller needs to establish dependencies between multiple operations.	SPS Client
PROCEDURE OrderSubmission_SubmitCancelToMOP (@OrderNumber char(7), @RootComponentID int, @UserName char(20), @Program char(20), @CancelDoc xml, @GeneratedSubmissionID int OUT, @GeneratedCancelComponentID int OUT, @GeneratedCancelDocumentID int OUT)	Encapsulates logic required to submit an order cancellation request to MOP. The GeneratedXXX output values are informational and can be ignored in most cases.	SPS Client
PROCEDURE OrderSubmission_GetNextOpenSubmissionComponent (@HowManyToAcquire int, @ProcessorName varchar(30), @ReturnCreates bit, @ReturnAmends bit, @ReturnCancels bit, @ReturnBids bit, @ReturnEmails bit)	Locates the next Submission Component to be processed by the Order Writer. A "component" is a Create, Amend, Cancel, Bid Create, or Email Confirmation. This procedure honors dependencies between components such as for Auto Q2O.	Order Writer

The following additional stored procedures perform supporting functions in MOP and are not used directly by the SPS client. Many are used by the SPS Diagnostic Console for supporting the order processing system:

Stored Procedure Signature	Description	Used By
PROCEDURE OrderSubmission_AddComponent (@SubmissionID int, @DependsOnComponentID int, @ComponentType char(1), @ComponentStatus CHAR(1), @ComponentID int OUT)	Adds a component to a submission tied to the click on the SPS "place order" button. A submission can have several parts, such as order/bid/email. One component exists for each part of the submitted package the components are	Used internally by the SubmitOrderToMOP sto red procedure.

	treated separately but as related.	
PROCEDURE OrderSubmission_AddComponentHistory (@ComponentID int, @EventType char(5), @ErrorMsg varchar(4000), @Actor varchar(20), @HistoryID int out)	Adds a history entry to a given submission component for tracking what happened during order/email/bid processing.	SubmitOrderToMOP, Order Writer, SPS Diag Console
PROCEDURE OrderSubmission_AddDocument (@DocumentType char(3), @OrderDoc xml, @DocumentID int OUT)	Adds document to the cache, which will be tied to an order/bid/email component of a Place Order submission.	Used internally by the SubmitOrderToMOP sto red procedure.
PROCEDURE OrderSubmission_AddSubmission (@OrderNumber char(7), @LinkedOrderNumber char(7), @UserName char(20), @Program char(20), @SubmissionID int OUT)	Defines an order "submission", which corresponds to a click on the SPS Place Order button	Used internally by the SubmitOrderToMOP sto red procedure.
PROCEDURE OrderSubmission_GetAllDocsByComponent (@ComponentID int)	Fetches all XML documents associated with a component of a Place Order submission. For example, the Order document or Bid Create document.	Order Writer, Diag Console
PROCEDURE OrderSubmission_GetComponentsByDateRange (@CreationDate datetime, @LastActivityTime datetime, @ComponentTypesToReturn varchar(50), @StatusesToReturn varchar(50))	Fetches all "components" (order/bid/email) of a given SPS Place Order submission.	Diag Console
PROCEDURE OrderSubmission_GetComponentsByOrderNumber (@OrderNumber varchar(20))	Fetches all "components" (order/bid/email) of a given SPS Place Order submission by the order number under which they were placed.	Diag Console
PROCEDURE OrderSubmission_GetComponentsForSubmission (@SubmissionID int)	Fetches all "components" (order/bid/email) of	Diag Console

	a given SPS Place Order submission.	
PROCEDURE OrderSubmission_GetComponentsStuck (@StuckIntervalInMinutes int)	Fetches all "components" (order/bid/email) of a given SPS Place Order submission that appear to be "stuck". This can happen if an order writer service instances crashes during a write. "Stuck" is determined by using an interval without activity.	Order Writer
PROCEDURE OrderSubmission_MapDocumentToComponent (@ComponentID int, @DocumentID int)	Maps a submission component to a corresponding XML document where appropriate. The mapping is many-to-many.	Used internally by the SubmitOrderToMOP sto red procedure.
PROCEDURE OrderSubmission_MarkSubmissionAsReady (@SubmissionID int)	Called when all of the components of a submission are in place and the Place Order operation is ready to be processed. It simply marks the submission record as being "ready".	Used internally by the SubmitOrderToMOP sto red procedure.
{{tab}}PROCEDURE OrderSubmission_PurgeOldSubmissions (@PurgePriorTo datetime)	Called periodically to "cull" old submission data from the database	Order Writer
PROCEDURE OrderSubmission_ReplaceDocument (@DocumentID int, @OrderDoc xml)	Used to replace an XML document that is tied to an existing submission component. This can be used, for example, to "fix" a bad order document.	Diag Console

PROCEDURE OrderSubmission_SetComponentStateCanceled (@ComponentID int, @Actor VARCHAR(20))	Sets a 'Cancelled/Abandoned' state for the given submission component. This causes the order writer process to ignore the component. Records history of the activity.	Diag Console
PROCEDURE OrderSubmission_SetComponentStateCompleted (@ComponentID int, @Actor varchar(20))	Sets a 'Completed' state for the given submission component. This signals the order writer process to ignore further processing the component because it is done. It also triggers any other components that depend upon completion of this part of the submission. Records history of the activity.	Order Writer
PROCEDURE OrderSubmission_SetComponentStateDirectly (@ComponentID int, @NewStatus char)	Sets an arbitrary state for the given submission component. A history record is NOT record.	Order Writer, Diag Console
PROCEDURE OrderSubmission_SetComponentStateError (@ComponentID int, @ErrorMsg varchar(4000), @Actor varchar(20))	Sets an 'Error' state for the given submission component, while creating a history record containing an error message detailing the nature of the problem. The Order Writer will not attempt further processing on this	Order Writer

	component unless it is resumed.	
PROCEDURE OrderSubmission_StartComponentProcessing (@ComponentID int, @ProcessorName varchar(30))	Signals that the component is being "grabbed" by an order writer instance and returns a non-zero row count if the component is secured. Once "grabbed" then the writer can safely assume that another writer instance won't work on the same order document concurrently.	Order Writer
PROCEDURE OrderSubmission_UpsertOrderProcessState (@ComponentID int,{{tab}} @InitialOperationType char(1),{{tab}} @ActualOperationType char(1),{{tab}} @WriteStateVarMap varchar(1000),{{tab}} @Attempts smallint,{{tab}} @LastModified datetime)	Adds an 'order process state' record to the DB which is used to track an order's progress through the Order Writer. It can be used to carry state information between plugin in a manner that can surviv disruption s in processing.	Order Writer
PROCEDURE OrderSubmission_GetOrderProcessStateByComponent (@ComponentID int)	Returns processing state for a given component by ComponentID.	Order Writer