Table of Contents

TechNote - Maintaining configuration files

Software applications are getting more and more complex in all aspects. This is also the case for a particular aspect every programmer has to face and find a solution for: the configuration of applications. This document outlines a generic, easy to use and easy to implement approach for the maintenance of configuration files using Alaska Software technology.

Configuration data, what is it?

The term "Configuration data" describes data that is not processed by an application in the usual sense, such as creating a report from a database, for example, but is used to **control the behavior** of an application. What is the default directory for databases? Where to store the position of the application window so that the program "wakes up" at the same position when it is re-started? These are common questions to be answered in the context of configuring an application program.

The problem

The main problem in finding a solution to such questions is the lack of a standardized and convenient file format to be used for configuration data storage. Some developers tend to use the Windows registry for this purpose, others write a parser/reader for their own proprietary file format, and there is a group of programmers favoring the well known Windows INI file format which develop their own program routines to process such files.

As a result, programmers all over the world have developed scanners or parsers to suit their own needs, and it can be assumed that parsers or scanners are one of the most redundantly written pieces of code.

Configuration data can be "hard-coded" in PRG source code. However, there is an increasing demand for flexibility and this requires an application to become more and more data driven, i.e. the application must use external data. In addition, applications must integrate with other solutions for the purpose of data exchange. EDI or EDIFACT is a well known data exchange format in this context, for example.

The requirements

The need for a platform independent, easy to maintain and widely accepted file format for configuration data requires a configuration file to be readable/editable with a normal ASCII editor. This excludes any form of binary files, such as DBF or XPF files. In addition, configuration data must be accessible via symbolic names, it must be user-definable, covering complex data structures such as arrays of more than three dimensions in depth. Windows INI files meet two of these requirements (ASCII editor, symbolic names), and this is the reason for the wide acceptance of this file format, but INI files are unsuitable when complex data structures must be stored.

An appropriate file format

A file format for conveniently reading/editing configuration data with an ASCII editor that allows for storing complex data structures and accessing data via symbolic names is XML. In addition, the XML file format is platform independent since it is defined by a commonly accepted standard.

These are some reasons for the existence of the Alaska Software XML parser library that allows for easy processing of XML files from Xbase^{++} applications. The purpose of this document is to demonstrate the versatility of XML file usage for configuring an Xbase^{++} application. A short introduction into XML is followed by a discussion of how to design XML files and how to apply the Alaska Software XML technology in a common usage scenario.

Introduction to XML

This section gives you a brief introduction to the XML file format and demonstrates the design process for XML files from an Xbase^{++} programmer's point of view. How to define configuration data for an Xbase^{++} application is then discussed using a data driven approach for loading and building Database Engines.

What is XML?

To answer this questions, we will start with a definition found in one of the numerous books presenting XML introductions:

XML enables you to define a grammar for marking up documents in terms of semantic tags and their attributes.

This is a quite good definition of XML. However, don't be disappointed if it doesn't tell much to you. There are some terms that need an explanation:

Grammar	This term describes the words allowed in a language and how or where they may appear in a sentence.
Semantic	This term describes the meaning of the words allowed in a language. E.g. "Deutsch" is not part of the English language, it is part of the German language. The semantic of the words "Deutsch" and "German" is identical, depending on the language.
Mark-up	This term describes all characters in a text that define how text is displayed or printed. For example, *italic* or **bold** markers are included in a text but do not contribute to its contents. The markers are invisible when the text is displayed or printed. However, they are visible in an ASCII or HEX editor.
Tag	A tag is a word enclosed in an opening and closing delimiter. <HTML> is a tag existing in all HTML documents, for example.
Attribute	An attribute is part of a tag and defines additional information for mark-up. For example, <TR ALIGN="RIGHT"> is a string defining the HTML tag <TR> whose ALIGN attribute has the value "RIGHT".

With these "definition of terms", the XML definition above is much better to understand: XML allows you to define your own grammar for mark-up, i.e. in XML you can define the names for tags and their meaning, the context in which they may appear, and you can define attributes for tags and their meaning. As a matter of fact, XML is a meta-language that allows you to define your own language to describe documents, or data structures, and that is what makes XML the file format of choice for managing configuration data.

Since XML is a mark-up language, similar to HTML, there are some rules to comply with. The first rule is that all XML files must be "well-formed". This means that textual content in a file must be embedded in an opening and closing tag. The second rule requires all XML files to begin with an <?xml?> tag which identifies the file as being an XML file and includes version information. This is the only tag that has no closing pendant. The following XML code illustrates a simple, well-formed XML document used to describe the data structure for a customer address:

```
<?xml version="1.0"?>
<Customer>
 <Address>
  <Street>1562 FIRST AVE</Street>
  <City>New York City</City>
  <Zip>KOC 1HO</Zip>
  <State>NY</State>
  <Country>USA</Country>
 </Address>
</Customer>
```

This example demonstrates how data is organized in XML files. They contain information that can be represented as a tree, or as a parent/child hierarchy. <Customer> has a child tag <Address>, <Address> has child tags <Street>, <City>, <Zip> and so forth. The opening tag defines the meaning (semantic) of embedded information and the closing tag defines the end of data. The tag names define the allowed words (grammar) for a data structure.

As you can see, XML is a pretty simple file format. It is flexible enough to manage data of high complexity that can be stored in an ASCII compliant file format. XML files can be used anytime you have to store information outside a database, or when you have to exchange information between applications. As a matter of fact, whenever you need a file format to exchange data or to store configuration data - simply start drafting your own XML document. You are always on the safe side with XML.

Furthermore, because XML is already a well accepted standard, you can expect more and more tools to become available over time which ease the editing, parsing and data-exchange of information based on XML. There are a couple of tools available already to edit XML documents - one is Microsoft's XML Notepad (http://msdn.microsoft.com/xml/notepad/intro.asp), another more sophisticated one is XMLSPY (http://www.xmlspy.com).

XML data for Xbase++ applications

The task we are going to solve is to define configuration data for an Xbase++ application that tell a program which DatabaseEngines (DBE) to load and to build at program start. In other words, we will replace the DbeSys() procedure using a data driven approach. XML allows us to perfectly describe the data required for calling the functions DbeLoad() and DbeBuild(). Both functions must be called in an application program to accomplish the proposed task. The following code shows an example of an XML file that contains the required data. We will discuss its design process step by step in the next section.

```
<?xml version="1.0"?>

<Configuration>

 <DatabaseEngines>
  <load>DBFDBE</load>
  <load>NTXDBE</load>

  <build name="DBFNTX">
   <data>DBFDBE</data>
   <order>NTXDBE</order>
  </build>
 </DatabaseEngines>

</Configuration>
```

Designing a configuration file in XML

As always, the design stage is the most time consuming part of the software development process. This is also the case for our small configuration file project. So let's start with the design.

The first thing an XML file must contain is the <?xml?> tag identifying the XML version used in a file. The <?xml?> tag is required due to the XML file specification. An XML parser will fail to process an XML file correctly unless this specification is met.

The next thing to do is to define the tags, or the words, describing our configuration data in a way that allows us to extend configuration files in the future, whenever this may be required. We start with defining the root tag and name it <Configuration> because that's what our XML file is all about. The root tag is going to have child tags which identify various configuration data. At the moment, however, we are dealing only with Database Engines, which one to load and build, and the first child tag is <DatabaseEngines>. This tag will embed all DBE related data in the XML file. The design process leads us to a basic structure for the configuration file:

```
<?xml version="1.0"?>

<Configuration>

  <DatabaseEngines>
    <!-- our DBE specific tags will be here -->
  </DatabaseEngines>

</Configuration>
```

There is a dedicated section for Database Engines and we must fill it with additional child tags in order to support the load and build operations for DatabaseEngines. Therefore, we have to take a close look at the parameter interface of the specific functions we need to support.

```
// load a DatabaseEngine
   DbeLoad( <cDbeFile>,  [<lHidden>] ) --> lSuccess

// build a DatabaseEngine
   DbeBuild( <cCompoundDBE>  , ;
             <cDataDBE>      , ;
            [<cOrderDBE>]    , ;
            [<cRelationDBE>] , ;
            [<cDictionaryDBE>] ) --> lSuccess
```

Again, our first step is the naming of our child tags, which is a pretty easy task. We just call them <load> and <build> to give them the right semantic. Then we define the syntax of our child tags. Let's start with the DbeLoad() operation.

Due to the nature of the DbeLoad() function, the <load> tag is simple. The DbeLoad() function expects at least a single parameter: the name of the DBE. As a consequence, we decide to design the tag as <load>....</load> where the DBE name to be loaded is embedded in the opening and closing tag. The following code shows the resulting syntax of the new child tag embedded in a parent tag:

```
<DatabaseEngines>
  <load>DBFDBE</load>
  <load>NTXDBE</load>
</DatabaseEngines>
```

The build operation is more complex. The function prototype accepts many different parameters with a different semantic. However, to keep things simple in our example, we limit the supported parameters and call DbeBuild() as follows:

```
DbeBuild( cNewDbeName, cDataDbe, cOrderDbe )
```

By taking a close look at these parameters and their semantic, we can isolate the *cNewDbeName* parameter as the name of the DBE to be built. To reflect this in our <build> tag, we add an attribute to the tag. The syntax is then as follows:

```
<build name="MYDBE">.
```

The interface of DbeBuild() function needs *cDataDbe* as the Data provider and *cOrderDbe* as the Order provider specified. However, there could be more DBEs and we want to stay flexible and keep all options open. Therefore, we add those DBEs as child tags to the <build> tag and use the provider type <order> and <data> as the names for child tags. The following code illustrates our new build tag.

```
<build name="DBFNTX">
  <data>DBFDBE</data>
  <order>NTXDBE</order>
</build>
```

Now, let us put the pieces together and see how the entire configuration file looks like:

```
<?xml version="1.0"?>

<Configuration>

  <DatabaseEngines>
    <load>DBFDBE</load>
    <load>NTXDBE</load>
    <load>CDXDBE</load>

    <build name="DBFNTX">
      <data>DBFDBE</data>
      <order>NTXDBE</order>
    </build>

    <build name="DBFCDX">
      <data>DBFDBE</data>
      <order>CDXDBE</order>
    </build>
  </DatabaseEngines>

</Configuration>
```

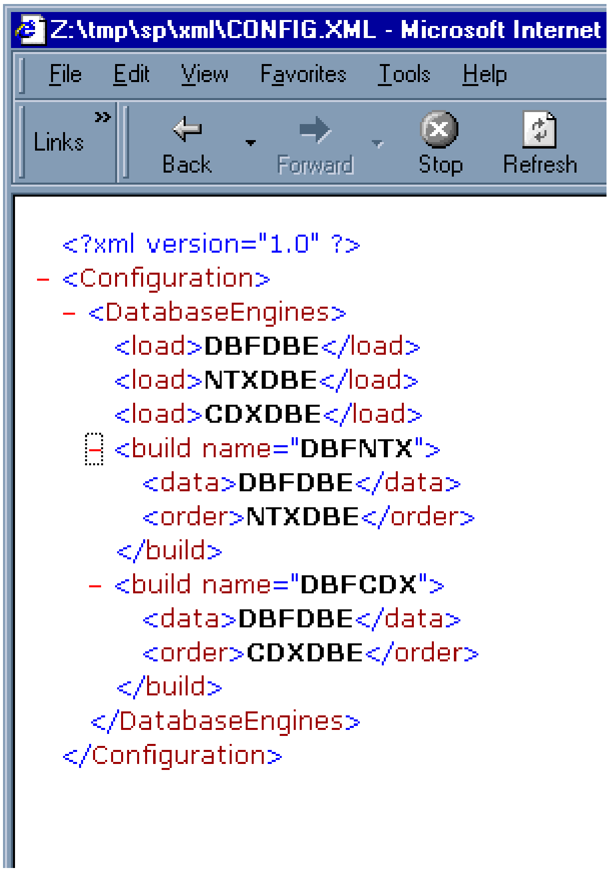
The section holding configuration data starts with <Configuration> as the root node. It embeds a <DatabaseEngines> tag which defines data related to DBE configuration. Child tags recognized for this tag are <load> and <build>. The <load> tags embeds the name of the DBE to be loaded. The <build> tag, in turn, requires additional child tags to specifiy the DBE components for the resulting DBE, while the name of the new DBE is reflected in the "NAME" attribute of the <build> tag.

By processing this XML configuration file, an Xbase^{++} application would load three component Database Engines and build two compound DBEs. If there is another DBE required, we would modify only the configuration file and add <load> and <build> tags embedding appropriate data. There is no need to recompile or relink an application.

An important aspect in the XML tag design is that DBE related data is embedded in the <DatabaseEngine> tag, which does not contribute to DbeLoad() or DbeBuild() related data. However, we gain the flexibility of re-using existing tags in the future in another context. For example, we could create a new child tag for <Configuration> and name it <DynamicDLL>. This section could support DllLoad() functionalities and could configure additional DLLs to be loaded at start-up. In this case, the <load> tag could be re-used since it has the same semantic for loading DLLs implemented in Xbase^{++} or other languages.

XML and Internet Explorer 5.0

A nice feature of Internet Explorer 5 is its capability to browse XML files. You can load an XML file into IE5 by dragging it over the browser. You then see a tree displaying your XML file structure. This is an easy way to inspect XML files. You can even click with the mouse on the "-" or "+" icons on each node to collapse or expand a subtree.


Z:\tmp\sp\xml\CONFIG.XML - Microsoft Internet
File Edit View Favorites Tools Help
Links
Back
Forward
Stop
Refresh

```
  <?xml version="1.0" ?>
- <Configuration>
  - <DatabaseEngines>
      <load>DBFDBE</load>
      <load>NTXDBE</load>
      <load>CDXDBE</load>
    - <build name="DBFNTX">
        <data>DBFDBE</data>
        <order>NTXDBE</order>
      </build>
    - <build name="DBFCDX">
        <data>DBFDBE</data>
        <order>CDXDBE</order>
      </build>
    </DatabaseEngines>
  </Configuration>
```

Processing XML configuration files

This section discusses XML file usage from the application's point of view. It describes the technology of the Alaska Software XML parser, how to use it und how to process configuration data for an Xbase^{++} application defined in XML files.

Alaska XML parser technology

The Alaska Software XML (ASXML) implementation provides an extremly fast lightweight XML processor. To give you an idea of its performance: it can process approximately 250.000 XML tags per second on a Dell Inspiron Laptop with a 433 MHz CPU.

The major design goals for the XML processor have been: simple usage, speed and a callback processing architecture. Callback processing simplifies the development of XML readers tremendously. Unlike Microsoft's XML implementation, where the parser creates a complete tree structure representing an entire XML document, a callback parser allows you to register own functions to process specific tags in a specific hierarchy.

For example, with a hierarchical parser you have to "walk through" the tree structure until you reach the node/tag of interest. The following pseudo code shows you the steps necessary if you would want to extract the contents of the <load> tag from an XML file using a hierarchical parser:

```
oRoot   := getRootTag()
oParent := oRoot:getChild( "DatabaseEngines" )
aChild  := oParent:getChildren( "load" )
FOR i:=1 TO Len( aChild )
  oChild := aChilds[i]
  // do whatever you want to to with the <load> tag
NEXT i
```

With a hierarchical parser you would extract the start, or root, tag and iterate through the tree until the desired tag is reached. In contrast, a callback processing parser uses a different approach. It assumes the user to register a function for a specific group of tags, or nodes, and passes the tags of interest along with associated data to that function. This means, the user associates an action with a tag and the parser starts traversing the tree and would execute all callback functions registered for specific tags. The following example demonstrates this technique and shows how easy it is to process the <load> tag using a callback processing approach.

```
registerFunction( "//DatabaseEngines/load", "myFunction" )
processDocument()

FUNCTION myFunction(oTag)
  // do whatever you want to do with the <load> tag
RETURN(.T.)
```

The program code for the callback processing approach is not only much easier to read, it is also easier to maintain if the XML tag definition changes. We just have to register a function to be called for a specific node in the XML document structure ("//DatabaseEngines/load") and then we implement this the callback function to process the tag we are interested in. If the structure of a XML configuration file changes, there is no need to adapt PRG source code to reflect the changes, because the code is executed on a "per tag" basis and is independent of the physical structure of the XML document.

Now that we had a little look into the architecture of the ASXML library let us use it to process our XML configuration file.

XML function overview

The ASXML library provides a callback processing parser for XML documents. A generic usage pattern for the parser can be split into the following steps:

1. Open the XML document with the XMLDocOpenFile() function.
2. Register functions to process the tags of interest, i.e. we have to define the action the parser should take for a node in the XML structure. This is done using the XMLDocSetAction() function, which allows us to associate a code block with a particular XML tag. We can set as many actions for a document as we want, as long as there is only one action per node.
3. Start the callback processor using the XMLDocProcess() function.
4. Close the document using the XMLDocClose() function.

That's all about it. There are only four functions required from the ASXML library to process our document.

XML functions used to process a XML document

Function	Purpose
XMLDocOpenFile(cFileName)	Load a XML document from file
XMLDocSetAction(nHandle, cNode, bCallback)	Sets the action per node
XMLDocProcess()	Process the document and execute all callback functions
XMLDocClose()	Close the document an release the parser

The source code below implements the XML document loader and processing part. For illustrational purposes we have left out error handling. In our configuration file, we have specified two primary tags, <load> and <build>. Both tags must be located between the opening and closing <DatabaseEngines> tag, i.e. they are child tags. That means we have to register our callback functions at "//DatabaseEngines/load" and "//DatabaseEngines/build" to process these tags. The first slash in this node-string is a place holder for all tags preceding the <DatabaseEngines> tag in the XML file structure. The following slashes delimit the names of the tags we are interested in. The implementation how to process each tag is programmed in our callback functions handleLoad() and handleBuild().

```
/*
 * Document Processor cFilename must be a XML document
 * according to our configuration file syntax sample.
 */
FUNCTION processConfig(cFileName)
  LOCAL nXMLDoc,nActions

  // load XML document
  nXMLDoc := XMLDocOpenFile(cFileName)

  // register functions by setting actions.
  nActions := XMLDocSetAction(nXMLDoc, "//DatabaseEngines/load",;
                              {|n,c,a,ch|handleLoad(n,c,a,ch)})

  nActions := XMLDocSetAction(nXMLDoc, "//DatabaseEngines/build",;
                              {|n,c,a,ch|handleBuild(n,c,a,ch)})

  // If there is something to process, we start the processor
  // to call our callback functions handleLoad() and handleBuild()
  IF nActions != 0
    XMLDocProcess(nXMLDoc)
    XMLDocResetAction(nXMLDoc)
  ELSE
    RETURN(.F.)
  ENDIF
  XMLDocClose(nXMLDoc)
RETURN(.T.)
```

As you can see, writing the processor for our own XML document format is a relatively simple task. The most important part is the design of the XML document, and the specification of the nodes in the tree we are interested in.

To process each tag, we have to implement the callback functions handleLoad() and handleBuild(). We start with the handleLoad() function which is easy since the name of the DBE is the content of the tag. Therefore, we simply retrieve the tag from its handle and use the tag's content (the string between the opening and closing tag) as the name for the DBE to be loaded. The source code of this task is outlined next.

```
/* Callback function to handle the DbeLoad action
 *
 * Tag: <load>#PCDATA</load>
 */
FUNCTION handleLoad(cTag,cContent,aMember,nHandle)
  LOCAL aCH
  XMLGetTag(nHandle,@aCH)
  DbeLoad(aCH[XMLTAG_CONTENT],.F.)
RETURN (XML_PROCESS_CONTINUE)
```

The XMLGetTag() functions retrieves an array of values for a specific tag determined by its unique handle. The array is comprised of the following attributes.

#define constants for the TAG attribute array

#define	Content
XMLTAG_NAME	Name of the tag
XMLTAG_CONTENT	The string between the opening and closing tag
XMLTAG_CHILD	Array of handles for the child tags
XMLTAG_ACTION	Codeblock if action was registered
XMLTAG_ATTRIB	Array of attributes for tag

The next sample-code illustrates the implementation of our <build> tag processing, which is more complex. Remember, we have used an attribute for the tag to specify the name of the DBE to be built, and we have added a couple of child tags to specify the Data and/or Order component of the DBE. Therefore, we have to obtain the attributes of an XML tag and we have to access specific child tags.

```
01:/* Callback function to handle DbeBuild action
02: *
03: */
04:FUNCTION handleBuild(cTag,cContent,aMember,nHandle)
05:  LOCAL aCH :={}
06:  LOCAL nHChild,cName,cOrder,cData
07:
08:  cName := XMLGetAttribute(nHandle,"name")
09:
10:  nHChild := XMLGetChild(nHandle,"data")
11:  XMLGetTag(nHChild,@aCH)
12:  cData := aCH[XMLTAG_CONTENT]
13:
14:  nHChild := XMLGetChild(nHandle,"order")
15:  XMLGetTag(nHChild,@aCH)
16:  cOrder:= aCH[XMLTAG_CONTENT]
17:
18:  DbeBuild(cName,cData,cOrder)
19:RETURN (XML_PROCESS_CONTINUE)
```

Our callback function handleBuild() is executed each time the parser finds a node "//DatabaseEngine/build" in our XML document. To retrieve an attribute of a tag by name, we can use the XMLGetAttribute() function (line #8), which returns the value of the attribute. This is the name of the DBE to be built.

Now we must access the child tags <data> and <order> and retrieve their contents which is a little bit more complex. First we have to retrieve the child tag by its tag name. This is done using the XMLGetChild() function (line #10) which accepts the numeric handle of the <build> tag currently being processed (the parent tag). This function returns a numeric handle for the requested child tag. The retrieval of the contents of the <order> and <data> tags is then accomplished by the XMLGetTag() function, just as we have done it already with the <load> tag.

That's all! To process XML documents, we have implemented the document loader and processor and we have registered two callback functions to process two specific tags. Then we have implemented both callback functions to process each tag. With this code you have a full working reader and processor for the example configuration file to handle loading and building of DatabaseEngines.

For your convenience, we have added the full source code and a sample XML file to this document's appendix. Try it out and adapt it to your specific needs. We are sure that XML will now find its place in your code-library of ready-to-use solutions.

A final word

We have explored how to design and process a XML document which fits a specific purpose - here a configuration file for our mission critical database application. We have left out error handling and how to ensure that our XML document conforms with the invented syntax and structure. We have also left out XML usage scenarios related to database export/ import functionalities. This and more will be a part of one of our next Alaska Software TechNote articles. So please stay tuned for the next TechNote from Alaska Software.

Appendix

Source code

```
#include "asxml.ch"
#pragma library("ASXML10.LIB")

/* sample main procedure
 */
PROCEDURE MAIN()
  ? DbeList()
  processConfig("test.xml")
  ? DbeList()
RETURN

FUNCTION processConfig(cFileName)
  LOCAL nXMLDoc,nActions

  nXMLDoc := XMLDocOpenFile(cFileName)

  nActions := XMLDocSetAction(nXMLDoc, "//DatabaseEngines/load",;
                              {|n,c,a,ch|handleLoad(n,c,a,ch)})

  nActions := XMLDocSetAction(nXMLDoc, "//DatabaseEngines/build",;
                              {|n,c,a,ch|handleBuild(n,c,a,ch)})

  IF nActions != 0
    XMLDocProcess(nXMLDoc)
    XMLDocResetAction(nXMLDoc)
  ENDIF
  XMLDocClose(nXMLDoc)
RETURN

/* Callback function to handle the DbeLoad action
 */
FUNCTION handleLoad(cTag,cContent,aMember,nH)
  LOCAL aCH
  XMLGetTag(nH,@aCH)
  DbeLoad(aCH[XMLTAG_CONTENT],.F.)
RETURN (XML_PROCESS_CONTINUE)

/* Callback function to handle DbeBuild action
```

```
 */
FUNCTION handleBuild(cTag,cContent,aMember,nH)
  LOCAL aCH :={}
  LOCAL nHChild,cName,cOrder,cData

  cName := XMLGetAttribute(nH,"name")

  nHChild := XMLGetChild(nH,"data")
  XMLGetTag(nHChild,@aCH)
  cData := aCH[XMLTAG_CONTENT]

  nHChild := XMLGetChild(nH,"order")
  XMLGetTag(nHChild,@aCH)
  cOrder:= aCH[XMLTAG_CONTENT]

  DbeBuild(cName,cData,cOrder)
RETURN (XML_PROCESS_CONTINUE)

/* since we load the DBEs using external data,
 * we overload DBESYS and load nothing
 */
PROCEDURE DBESYS()
RETURN
```

Sample XML document

```xml
<?xml version="1.0"?>

<Configuration>

  <DatabaseEngines>
    <load>DBFDBE</load>
    <load>NTXDBE</load>
    <load>CDXDBE</load>

    <build name="DBFNTX">
      <data>DBFDBE</data>
      <order>NTXDBE</order>
    </build>

    <build name="DBFCDX">
      <data>DBFDBE</data>
      <order>CDXDBE</order>
    </build>
  </DatabaseEngines>

</Configuration>
```