Document Indexer Persistence 1.0 Component Specification

1. Design

The Document Indexer Persistence component implements the persistence layer as required by the Document Indexer component. The pluggable framework allows different persistence mechanisms to be used. For the initial version, two mechanisms (XML and database) are provided.

This component will implement interfaces defined by the Document Indexer component to persist the necessary document index data. Developers should familiarize themselves with the overall design and API of that component.

Database persistence is straightforward; there is an ERD and a SQL file for creating the database in the "docs" folder. Each document is a single entry in the Document table. Each word in the document is an entry in the Word table. Each word only appears in the Word table once, but it could be included in multiple documents. The Location table holds values where each word can be found in each document. The user has the option of using a transaction when updating either index, through each class' AutoCommit property. If AutoCommit is false, a transaction will be used, committed using the Save method of each class.

XML persistence involves reading and writing XML files according to the schemas found in the "docs" folder. The only twist is that the user has the option of splitting out the XML for a document index into multiple files. This allows the user to get around file system size limits for particularly large documents.

1.1 Design Patterns

The **strategy** pattern is implemented by a combination of this design and the Document Indexer design. The IDocumentIndexSource interface and ICollectionIndexSource can be referenced generically, independent of whether they are an XML or DB based implementation.

1.2 Industry Standards

XML
SQL

1.3 Required Algorithms

Empty string:

An empty string is defined, in this component, as any string that is all white space, or is an empty string. That is, MyString.Trim().Equals(string.Empty);

The index implementations follow the example in the Document Indexer class, where the collection index has an IDictionary of indices, referenced by name, and the document index has an IDictionary of word locations, referenced by word string. These collections are loaded and saved using the Load and Save

XML Collection index

The collection index schema is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:topcoderprog="http://www.topcoder.com"
targetNamespace="http://www.topcoder.com">
    <xs:element name="Documents" type="topcoder:Documents"/>

    <xs:complexType name="Document">
        <xs:sequence>
            <xs:element name="Name" type="xs:string"/>
            <xs:element name="FilePath" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

We don't keep track of assembly names and classes for each document, since we only support creating XmlDocumentIndexSource implementations from the XmlCollectionIndexSource.

Loading Steps:

1. Create a new XmlDocument, loaded with the data from the file path given
2. Get the Documents node from the file
3. For each Document node in the Documents node
    a. Get the name of the document index
    b. Get the file path of the document index
    c. Create a new SimpleIndexParameters instance with the file path
    d. Create a new IndexInfo with the assembly name and class name of XmlDocumentIndexSource, and the SimpleIndexParameters instance created
    e. Add the IndexInfo to the indices IDictionary, with the document name as the key

Saving Steps:

1. Create a new XmlDocument
2. Add a Documents node to the root XmlDocument
3. For each document index in the indices collection
    a. Create a new Document node
    b. Create the FilePath and Name nodes and append them to the Document node
    c. Append the Document node to the Documents node
4. Save the Xml to the file pointed at by the file path.

XML Document Index

The document index schema is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:hermesprog="http://www.topcoder.com"
targetNamespace="http://www.topcoder.com">
    <xs:element name="Document" type="topcoder:Document"/>
    <xs:complexType name="Document">
        <xs:sequence>
           <xs:element name="CompareOptions" type="xs:string"/>
           <xs:element name="Culture" type="xs:string"/>
            <xs:element name="Words" type="topcoder:Words"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="Words">
        <xs:sequence>
            <xs:element name="Name" type="xs:string"/>
           <xs:element name="Words" type="topcoder:Words" minOccurs="0"/>
           <xs:element name="Files" type="topcoder:Files" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="Files">
        <xs:sequence maxOccurs="unbounded">
            <xs:element name="FilePath" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="Words">
        <xs:sequence maxOccurs="unbounded">
            <xs:element name="Word" type="xs:string"/>
            <xs:element name="Locations" type="topcoder:Locations"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="Locations">
        <xs:sequence maxOccurs="unbounded">
            <xs:element name="Location" type="xs:numeric"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

It optionally can reference files in the following form:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:hermesprog="http://www.topcoder.com"
targetNamespace="http://www.topcoder.com">
    <xs:element name="Words" type="topcoder:Words"/>
    <xs:complexType name="Words">
        <xs:sequence>
            <xs:element name="Word" type="topcoder:Word"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="Word">
        <xs:sequence maxOccurs="unbounded">
            <xs:element name="Word" type="xs:string"/>
           <xs:element name="Locations" type="topcoder:Locations"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="Locations">
        <xs:sequence maxOccurs="unbounded">
            <xs:element name="Location" type="xs:numeric"/>
```

```
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

The second schema is used if the user has selected the option of saving a document index to multiple files. The second file has all words and locations for a number of words, with the actual number dependent on the number of total words and number of files they are split over.. The XmlDocumentIndexSource class has a SaveWordsToFile method that takes a collection of words given to it and saves them to a file in the Words schema above.

When saving to multiple files, we split the words into as many sub-files as requested by the user to create, and then we create a new XML document for each group of words. The group size is easily calculated by just getting the number of total words, and dividing it by the number of files to create, rounding up to the nearest integer. These files should be named with the name of the file we are saving to, appended with a GUID. So saving to Document.xml would result in the file Document.xml being created, along with Document(some GUID).xml, Document(some GUID).xml etc…

Loading Steps (single file):

1. Create a new XmlDocument, loaded with the data from the file path given
2. Get the Words node from the file
3. For each Word node in the Letters node
    a. Insert the word and locations into the words IDictionary

Loading Steps (multiple files):

4. Create a new XmlDocument, loaded with the data from the file path given
5. Get the Files node from the file
6. For each FilePath node in the Letters node
    a. Load the words in the file, using the GetWordsFromFile method.

Saving Steps (single file):

1. Create a new XmlDocument
2. Add the CultureInfo and CompareOptions nodes
3. Add a Words node to the root XmlDocument
4. For each word
    a. Create a new “Word” node for each word and the locations of that word in the document index
    b. Add the word node as a child of the Words node
5. Save the Xml to the file pointed at by the file path.

Saving Steps (multiple files):

1. Create a new XmlDocument
2. Add the CultureInfo and CompareOptions nodes
3.
4. Add a Files node to the root XmlDocument
5. For each group of words
    a. Get the group of words from the words IDictionary
    b. Save them to a file using the “SaveWordsToFile” method
    c. Create a FilePath node pointing to the file where the words were saved
    d. Append the FilePath node to the Files node
6. Save the Xml to the file pointed at by the file path.

Database Collection Index:

Each document is a different entry in the Document table, with an auto-assigned ID. The Location table holds the mapping between entries in the Word table and entries in the Document table. The Word table only holds unique words, to cut down on space used by the system.

All SQL statements should be done using parameterized SQL to help with efficiency.

Both the DB classes work directly with the database for adding, removing, clearing, loading and saving. They do keep references to indices and words, but when calling Add, Remove, Clear, Load, or Save, those operations are DB dependent.

Adding a document index to the collection (DBCollectionIndexSource.Add):

```
INSERT INTO Document(document_name)
       VALUES(@documentName)
```

Removing a document index from the collection (DBCollectionIndexSource.Remove):

```
DELETE FROM Document
       WHERE document_name=@documentName
```

Clearing the collection (DBCollectionIndexSource.Clear()):

```
TRUNCATE TABLE Document
TRUNCATE TABLE Word
TRUNCATE TABLE Location
```

Loading all document indexes in a collection (DBCollectionIndexSource.Load):

```
SELECT document_id, compare_options, culture, document_name FROM Document
```

For each record returned, create a new DBDocumentIndexParameters instance with the connection type, string, auto-commit and document_id from the database

Create a new IndexInfo with each DBDocumentIndexParameters and add it to the indices Dictionary with the document_name from the database.

Loading all words for a document index from the database (DBDocumentIndexSource.Load):

```
SELECT location, location_id, document_id, word_id FROM Location
INNER JOIN Word ON Location.word_id=Word.word_id
WHERE document_id=@documentID
```

For each record returned, find the word value in the words IDictionary, if it exists, and add the location found to the IList value. If the word doesn't exist in the words IDictionary, create a new ArrayList with the location of the record and add the key and value to the words collection

Removing a Word from the database (DBDocumentIndexSource.Remove):

First, get the word_id from the database.

```
SELECT word_id from Word where word=@word
```

Then, delete all Location rows with the same document_id and word_id.

```
DELETE FROM Location
WHERE document_id=@documentID and word_id=@wordID
```

Clearing Words from the database for a given document (DBDocumentIndexSource.Clear):

```
DELETE FROM Location WHERE document_id=@documentID
```

Adding a Word to the database (DBDocumentIndexSource.Add):

First, check if the word exists in the Word table

```
SELECT word, word_id FROM Word WHERE word =@word
```

Then, insert the word if it doesn't exist.

```
INSERT INTO Word(word) VALUES(@word)
```

```
Next, insert a Location record for the location of the word

INSERT INTO Location(
                            word_id,
                            document_id,
                            location)
VALUES(
                            @wordID,
                            @documentID,
                            @location)
```

1.4 Component Class Overview

XmlDocumentIndexSource:
This class is an Xml based implementation of IDocumentIndexSource. It saves to and loads from one or more Xml files, depending on a limit defined by the user. Besides just the standard saving to an XML file, this class also provides the option to split the saving of the document index to multiple XML files, all linked from the main file. This provides us with a way to split the output file into any number of different files. This allows us a way to get around file system size limits on a single file. This class is thread safe, as the access to the words dictionary is synchronized.

XmlCollectionIndexSource:
This class implements the ICollectionIndexSource to provide the sources in an Xml document. This implementation saves and parses files according to the Documents.xsd file in the docs directory. This class provides functionality for getting information about a particular document, as well as saving back out all documents in the collection. The indices member variable in this class is mutable, but access to it is synchronized, leading to a thread safe implementation

DBDocumentIndexSource:
This class implements the IDocumentIndexSource in a database related manner. The saving and loading is all done to a database, based on connection parameters given. This class has the additional option to auto commit changes directly to the database, or use a transaction to queue up changes and then commit them all at once. This class is mutable and is not thread safe.

DBPersistenceParameters:
This class contains parameters that are used to initialize a DBCollectionIndexSource class. Contained are properties for connection type and connection string, which are used to get a connection from the Connection Factory. The AutoCommit property is used to tell the DBCollectionIndexSource whether or not each update goes directly to the database or whether it is added to a transaction to be committed later. This class is immutable and is thread safe.

DBDocumentIndexParameters:
This class extends the DBPersistenceParameters class to provide an extra DocumentID property as a supplement to the properties contained by its parent class. This class is immutable and is thread safe.

DBCollectionIndexSource:
This class provides a DB specific implementation of the ICollectionIndexSource interface. It holds properties for connection parameters, either passed in or loaded from the configuration manager. It only works with the Document table in the database, for loading in document indices. This class is mutable.

1.5 Component Exception Definitions

DBPersistenceConfigurationException:
This exception is thrown by both the DB implementations in this package if a given namespace doesn't contain the necessary information to load in all configuration values for each class.

1.6 Thread Safety

This component is not 100% thread safe. The XML persistence mechanism synchronizes its collections and is thread safe. The only two places in this part to worry about are the words and indices collection, and access to them is synchronized, providing thread safety.

The DB persistence is not thread safe. The collections are not synchronized, but even if they were, the inherent thread instability of the database operations would still cause these particular classes to be non-thread safe. Each class contains a single connection that is opened and closed, but multiple methods in each class use data readers to retrieve data from the database, and since only one data reader can be open at a time on any given connection, it is infeasible to make this layer thread safe. Too much locking on the connection would cause efficiency to be adversely affected.

2. Environment Requirements

2.1 Environment

- C# .NET 1.1
- Informix database
- Informix ODBC drivers -- http://www.dotnet247.com/247reference/msgs/13/68131.aspx

2.2 TopCoder Software Components

- Document Indexer 1.0: This component provides new persistence functionality for the Document Indexer component.
- Configuration Manager 2.0: The DB persistence layer loads connection information from the configuration manager.

- Connection Factory 1.0: The DB persistence layer loads in connection information from the Connection Factory based on configuration parameters.

NOTE: The default location for TopCoder Software component jars is../lib/tcs/COMPONENT_NAME/COMPONENT_VERSION relative to the component installation. Setting the tcs_libdir property in topcoder_global.properties will overwrite this default location.

3. Installation and Configuration

3.1 Package Name

TopCoder.Document.Index.Persistence.DB
TopCoder.Document.Index.Persistence.Xml

3.2 Configuration Parameters

Parameter	Description	Values
DBConnectionName	The name of the connection to get from the Connection Factory component	Any valid connection name
DocumentID (DBDocumentIndexSource only)	The ID of the document in the database to load words for	Any valid document_id value in the database

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Follow Dependencies Configuration.
- Execute 'nant test' within the directory that the distribution was extracted to.

4.2 Required steps to use the component

4.3 Demo

XML:

```
//Load from an XML file
ICollectionIndexSource xmlSource=new XmlCollectionIndexSource();
IParameters parameters=new SimpleIndexParameters("C:\\collection.xml");
xmlSource.Load(parameters);

//Enumerate through the indices
IDictionaryEnumerator enumerator=xmlSource.GetEnumerator();

while(enumerator.MoveNext())
{
      do something with each IndexInfo…
}

//Add a new index
IParameters documentParams=new SimpleIndexParameters("C:\\document.xml");
IndexInfo info=new IndexInfo("TopCoder.Document.Index.Persistence.Xml",
"XmlDocumentIndexSource", documentParams);

xmlSource.Add("Document", info);
```

```
//Save the changes
xmlSource.Save();

//Remove the index
xmlSource.Remove(“Document”);

//Get an index
IndexInfo retrieved=xmlSource[“Document2”];

//Clear all indices
xmlSource.Clear();

//Save the changes
xmlSource.Save();

//Create a new XmlDocumentIndexSource
IDocumentIndexSource source=new XmlDocumentIndexSource();
source.Load(documentParams);

//Add a new word
WordInfo word=new WordInfo(“Cat”, 102);
source.Add(word);

//Remove a word
source.Remove(“Dog”);

//Save the changes
source.Save();

//Save to multiple files
source.NumberOfSubFiles=2;
source.Save();

//Check if the document contains a word
if(source.Contains(“Snake”))
{
      Console.Writeline(“Snake exists, eeeek!”);
}

//Clear all words
source.Clear();

//Set a new file path
source.FilePath=”C:\\Empty.xml”;

//Save the empty file
source.Save();
```

Database:

```
//Load from a database
ICollectionIndexSource dbSource=new DBCollectionIndexSource();
IParameters parameters=new DBPersistenceParameters(“odbc”,
“Database=source; Server=test-server”, false);

dbSource.Load(parameters);

//Enumerate through the indices
IDictionaryEnumerator enumerator=dbSource.GetEnumerator();

while(enumerator.MoveNext())
```

```
{
      do something with each IndexInfo…
}

//Add a new index
IParameters documentParams=new DBDocumentIndexParams(“odbc”,
“Database=source; Server=test-server”, false, 8);

IndexInfo info=new IndexInfo(“TopCoder.Document.Index.Persistence.DB”,
“DBDocumentIndexSource”, documentParams);

dbSource.Add(“Document”, info);

//Save the changes
dbSource.Save();

dbSource.AutoCommit=true;
//Remove the index
dbSource.Remove(“Document”);

//Get an index
IndexInfo retrieved=dbSource[“Document2”];

//Clear all indices
dbSource.Clear();

//No need to commit the changes, they have already been auto committed,
//calling “Save()” is unnecessary.

//Create a new DBDocumentIndexSource
IDocumentIndexSource source=new DBDocumentIndexSource();
source.Load(documentParams);

//Add a new word
WordInfo word=new WordInfo(“Cat”, 102);
source.Add(word);

//Remove a word
source.Remove(“Dog”);

//Save the changes
source.Save();

//Set auto commit to true
source.AutoCommit=true;

//Get a specific word’s locations
IList locations=source[“Horse”];

//Check if the document contains a word
if(source.Contains(“Snake”))
{
      Console.Writeline(“Snake exists, eeeek!”);
}

//Clear all words
source.Clear();
```

5. Future Enhancements

The loading of parameters and classes could be made more extensible, but this will require changes to the IParameters interface. More database implementations could be supported. The ability to change the save location of each individual sub file for Xml persistence to multiple files could be useful.