AutoVue Integration SDK

Technical Guide

Cimmetry Systems, Corp.

Copyright Notice

Table of Contents

1. INTRODUCTION

Note: Before you start reading this document, it is strongly recommended that you first become familiar with the SDK by reading through the Overview, Installation Guide, and User Guide. These guides are in the **/docs** folder; you can access them from the readme file, **'Quick Start.html'**, located in the root folder where you installed the AutoVue Integration SDK.

The AutoVue Integration SDK is an interface between AutoVue Client-Server Edition (CSE) and a Document Management Systems (DMS). It enables users to add powerful viewing and markup capabilities to the DMS. The process of interfacing AutoVue with a particular DMS is referred to as an *integration*. The integration process itself is composed of several activities: requirements specification, analysis, design, implementation, test and maintenance. The development of the AutoVue Integration SDK fits into a larger vision of Cimmetry Systems Corp. to simplify and facilitate the whole integration process.

This document is intended for our partners and third-party developers (i.e., integrators) who want to implement their own integration with AutoVue. This guide serves as a good starting point for developers and professional services to become more familiar with the technical details of this SDK.

This document focuses on the implementation activity. It shows you in details how to implement your own integration based on the SDK Framework. To assist you with the integration, a sample (called Sample Integration for Filesys DMS) is included in this SDK. A detailed description of this sample is given in the appendix.

The objectives of this document are to help you to understand and become familiar with the SDK framework, and to help you build your own integration of AutoVue.

Document Management (DM) API is an XML specification of the request and responses exchanged between AutoVue and Framework. This framework is a Java based implementation of the XML specification. For detailed information about DMAPI, please refer to JavaDMAPI.pdf manual located under /docs folder. For complete list of Java Classes and API, please refer to index.html JavaDoc found under /docs/javadocs folder.

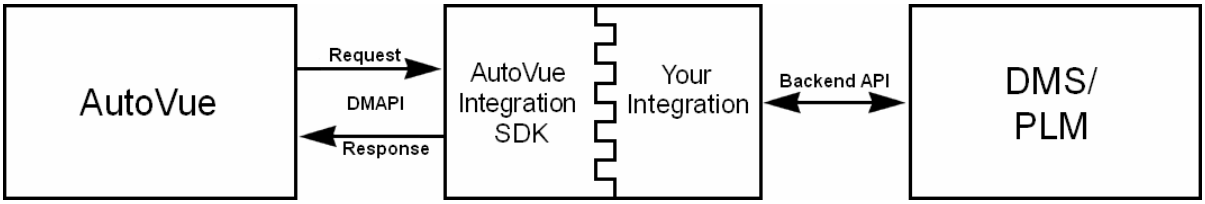


Figure 1: Integration with AutoVue

2. SYSTEM REQUIREMENTS

For a complete list of system requirements specific to your platform, refer the **Installation Guide**.

3. ARCHITECTURE

The following block diagram shows a typical configuration for integration between AutoVue and a DMS. This section describes how it works.

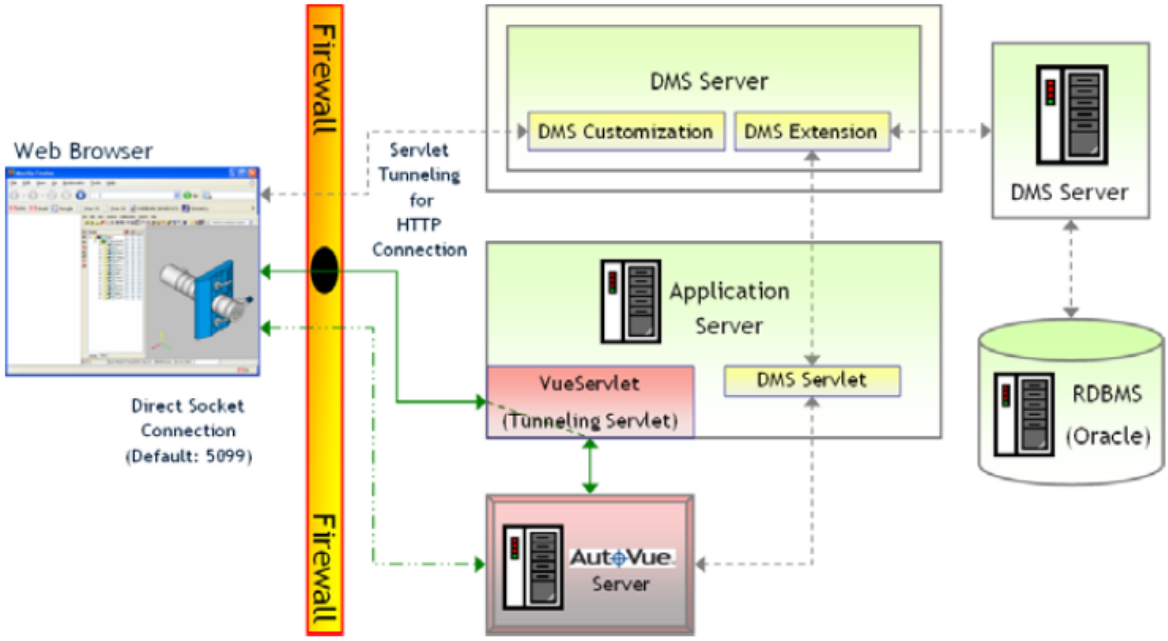


Figure 2: Typical configuration for AutoVue Integration with DMS server

3.1 How it Works

The **DMS Servlet** allows **AutoVue Server** to communicate with your **Document Management System (DMS), Electronic Document Management (EDM) system, or Product Data Management (PDM) system** using the standard HTTP protocol.

Here is a brief description of how the **DMS Servlet** works:

1. The client logs into the **DMS** through a Web browser, such as Microsoft Internet Explorer.

2. With **DMS Customization** in place, you are presented with a link labeled **'View'** next to each file stored inside **DMS**. The link lets you view files in the **AutoVue Applet** viewer.

3. When you click **'View'**, **AutoVue Applet** launches inside the Web browser window.

4. Depending on the **AutoVue** configuration, the **AutoVue Applet** communicates with **AutoVue Server** in one of the following two ways:

   - Through **Servlet Tunneling** for HTTP connection (**VueServlet**).

   - Through **Direct Socket** connection (default port is **5099**).

5. **The AutoVue Server** then communicates to **DMS Servlet** using a standard HTTP connection.

6. With the **DMS Extension** installed on the server machine, the **DMS Servlet** is able to talk to the **DMS** Server to handle any request made by the **AutoVue Server,** such as file fetching.

7. If you try to view a composite file (that is, a file having XRefs or font resource files), the **DMS Servlet** retrieves those files and makes them available to the **AutoVue Server**.

8. Once the file and all its related XRefs and/or resources are fetched out of the **DMS**, they are processed by the **AutoVue Server**, which renders the file(s) and streams the viewable to **AutoVue Applet** for display.

9. Once the file displays in the **AutoVue Applet**, you can redline it, create new Markups, save Markups into the **DMS,** and open Markups from the **DMS**.

3.2 Framework

The following block diagram shows the internal structure of a typical integration with a DMS. The framework included in the AutoVue Integration SDK provides you with the foundation you need to build your own integration. This framework handles all the plumbing for parsing XML requests received from the AutoVue Server, as well as constructing XML responses sent back to the AutoVue Server. This framework is provided so that you do not have to implement your integration from scratch.

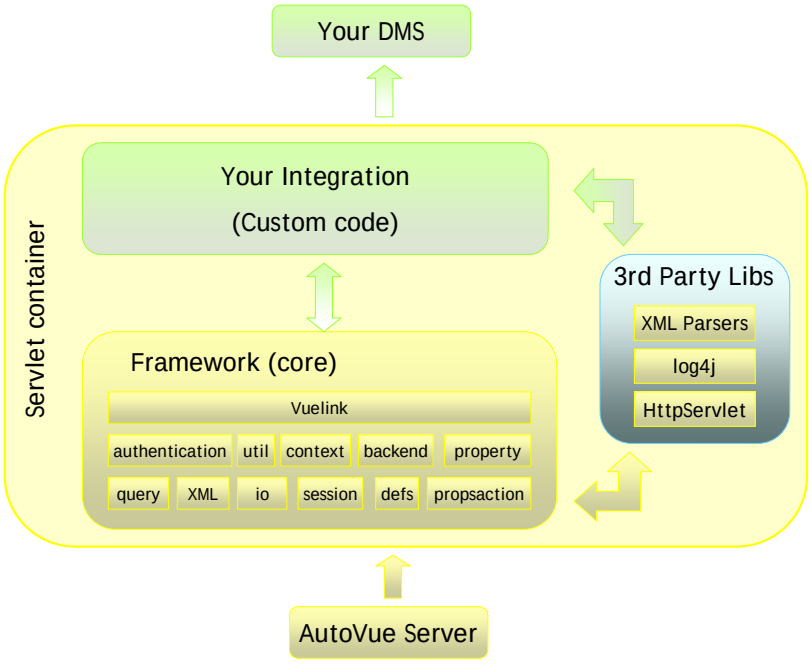


Figure 3. Internal Structure of the DMS Servlet

The AutoVue Integration SDK bundles some third-party Java libraries needed by the framework. These libraries are also available for you to call from your own code.

Your integration is responsible for interacting with your DMS. Depending on what type of SDK your DMS provides, such interaction can be as easy as calling your DMS Java libraries.

3.3 Sequence diagram

When a user selects a document to view, the **AutoVue** Server makes several requests to the DMS servlet. The DMS servlet provides a response for each request. The scenario of the exchanges established between the **AutoVue** Server and the DMS servlet are sketched in the following figure and can be summarized as follows:

- **AutoVue** Server asks for the docID of the selected document. This is done through the Action **Open,** which obtains the docID from the DMS.
- **AutoVue** Server asks for some properties of the document, such as document name, document size and date of the last modification (e.g., sequences 2 and 3 in the following figure). The reason is that the **AutoVue** Server maintains a cache repository of the document and needs to know if it already has the most recent copy of the document. In which case AutoVue uses the most recent copy rather than downloading the document.
- **AutoVue** fetches the document through the `Download` Action.

The following sequence diagram shows the flow of communication between AutoVue and your integration, for a typical case of viewing a file from your DMS. As you can see from this diagram, viewing a file triggers many calls to your integration.

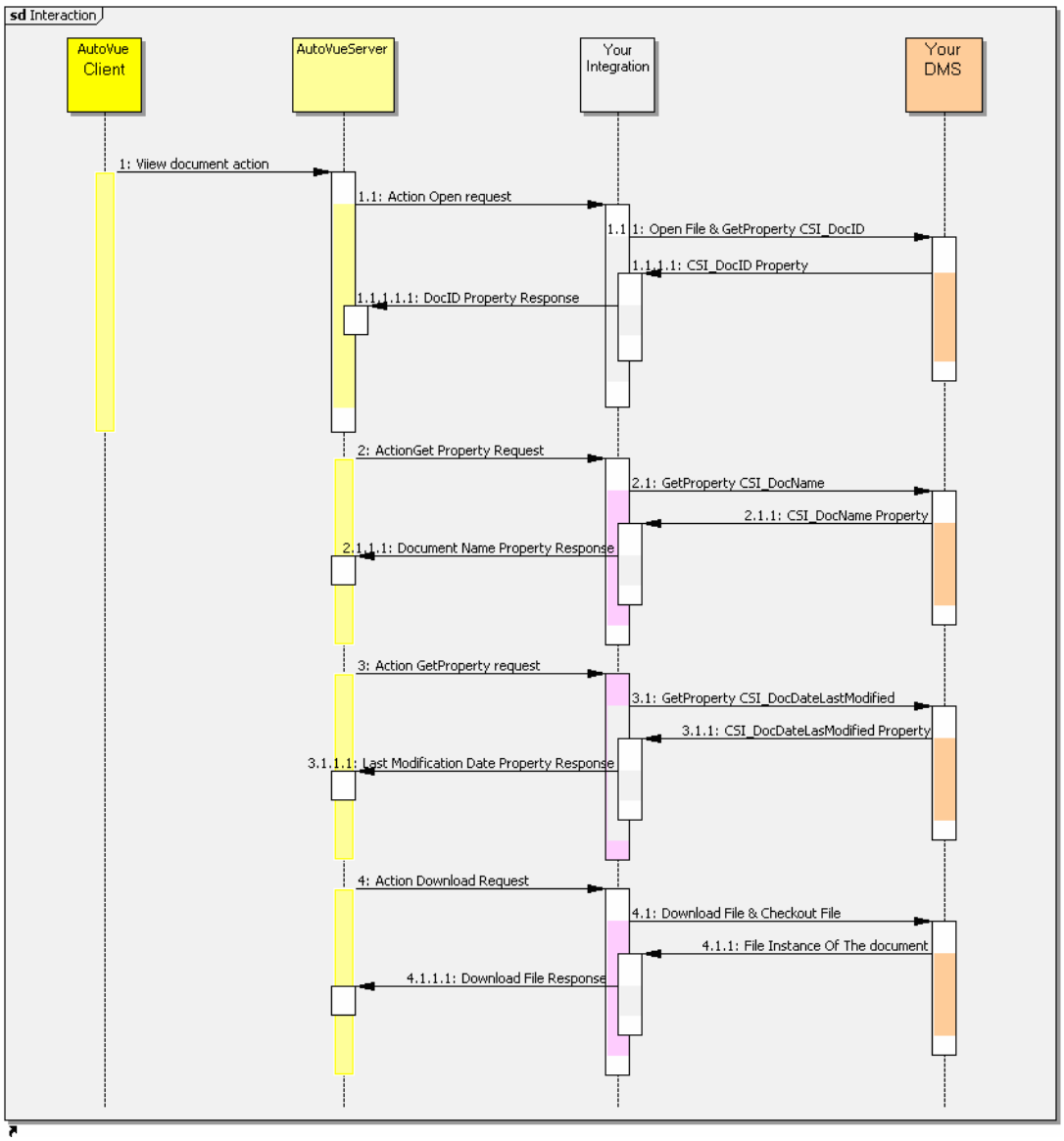


Figure 4. Sequence diagram for file view

.

4. INTEGRATION DESIGN

Integration is generally composed of two components: framework and your specific integration implementation.

Framework is a set of classes that can be used by your integration implementation. It provides you with all the needed functionalities to communicate with AutoVue Server and defines the key concepts to implement your new integration. Understanding these concepts is important for building accurate integrations. Here is a list of the most important classes and packages:

- **Vuelink servlet**: Base class for your DMS servlet (This is your main class)
- **DMSAction interface**: Represents an execution thread that handles a particular action (such as open, delete, download, save…).
- **DMSGetPropAction interface**: Represents an execution that handles the request for a specific property.
- **DocID interface**: Represents a DMS document ID.

All these concepts will be explained later in this section. For detailed information on these classes and packages, refer to **API Javadocs** located in the **/docs/javadocs folder**.

The second component is your specific integration, which is the code you add on the top of the framework in order to have a working integration. In other words, this is the main subject of this documentation.

Your integration must create a DMS servlet that extends Vuelink class and implements some actions and property actions.

4.1 Vuelink Class

The framework provides com.cimmetry.vuelink.`Vuelink` class which is an `HttpServlet` and is configured through the servlet initialization file. It performs important functionalities that establish the dialog between AutoVue and your integration. Your DMS servlet must extend this class.

- It sets up the log manager for enabling logging at runtime without modifying the application binary (log4j API).
- It registers the DMS Context action and DMS actions classes provided by your integration. Refer to **Javadocs** for more on the **context** package and the **propsactions** package.
- It parses the HTTP request using the `HttpRequestPart` class.
- It uses the `DMSXmlRequest` class, to parse the XML document that contains the actual request. Refer to **Javadocs** for more on the **xml** package.
- It builds a query object (i.e., *`DMSQuery`* object) containing all the document information and Properties that your integration needs. Refer to **Javadocs** for more on the **query** package.
- It also constructs some additional `DMSArguments` from an HTTP part or from some special data inside the XML document, such as the file content of a Save request for example. Refer to **Javadocs** for more on the **arguments** package.
- When **DMSQuery** is built, it calls the execute method of the appropriate `DMSAction`, and gets the result back or catches a `VuelinkException` when an error occurs. Refer to **Javadocs** for more information on the `defs` package.
- Finally, it uses the `DMSXmlResponse` class to construct the XML part of the HTTP response before sending it back. Refer to **Javadocs** for more on the **xml** package**.**

4.2 DMSActions Interface

AutoVue sends requests to your integration and expects responses from it through the framework interface. The framework implements a mechanism that routs requests to your DMS servlet and constructs responses back to AutoVue. The framework provides the `com.cimmetry.vuelink.propsaction.`*`DMSAction`* interface, which represents an execution thread that handles a DMS query. Your integration must define one *`DMSAction`* for each DMS action type (`Open`, `Save`, `Delete`, `Download`, `GetProperties` and `SetProperties`).

4.3 ActionGetProperties Interface

One of the most important requests that your integration must handle is `GetProperties`. This request covers a wide range of items. The implementation of this request as a single class with one monolithic '`execute`' method that handles all the properties has at least two limitations:

- **Understandability problem:** Too much code in one class, which makes it difficult to understand and then difficult to maintain.
- **Extendibility problem:** Since the class will perform lot of functionalities, it will be a hard task to extend it with new behavior.

One of the main objectives of the AutoVue Integration SDK is to make the framework open and easy to extend. Accordingly, instead of having a single class that takes care of the `GetProperty` request, we provide individual classes that handle individual properties. Each individual class will have its own `execute` method. When a `GetProperties` request is received, the framework will go through the list of properties. For each property, the framework will check if there is an appropriate action to handle it, using a mechanism similar to what is used for resolving the existing *`DMSAction`* classes. If such a class is found, its `execute` method is called, and its return property is saved. Any properties that do not have a specific handler class will be passed to a default class.

The framework provides a class for retrieving the individual classes that handles the properties contained in the `GetProperties` request. This class is called `com.cimmetry.vuelink.propsaction.ActionGetProperties` which implements the *`DMSAction`* interface. First, this class retrieves the class handler of the requested property, then it calls its '`execute`' method, and finally it returns an array of properties containing the response.

Each individual class you provide to handle a specific property must realize the `DMSGetPropAction`, then implement the `execute` method. The execute method must get the response for the request from your DMS and return it as an array of properties.

The `GetPropAction` retrieves each property action using the init-parameters mechanism. If the class is not registered, the framework looks for a property action defined with a default name "GetProp<prop name>" in the DMS servlet location. If no class is found, the `GetPropDefault` class is called. In this framework, the `GetPropDefault` class is treated as any other property action. If `GetPropDefault` is not found, an exception is thrown. Also if the requested property is not handled in the `GetPropDefault` class, an exception must be thrown.

4.4 DocID Interface

The `DocID` in this framework always refers uniquely to a specific document or file in your DMS. You must be able to ask for the contents of the file referred to by its `DocID`, and get a uniquely-identified result. In a typical DMS, this can be a combination of the object ID of the document that contains the file along with library name where this document is stored.

The following block diagram shows the minimum components you need to add to your integration. Chapter 5 describes these components in more detail.

- Your DMS Servlet class (extended from Vuelink class)
- Your DocID class (implements DocID interface)
- Your ActionOpen (implements DMSAction interface)
- Your ActionDownload class (implements DMSAction interface)
- Your ActionGetProperties class (implements DMSAction interface)

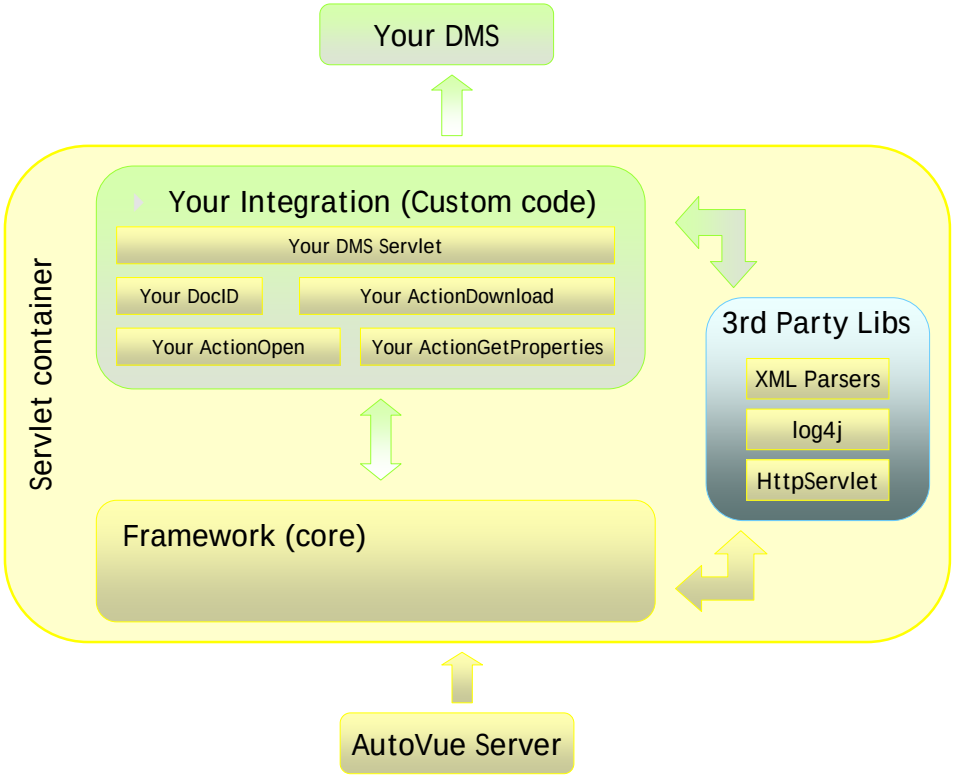


Figure 5. Your Integration

The following chapter describes the minimum set of steps you need to follow in order to implement the viewing functionality of files stored in your DMS using AutoVue.

5. STEPS FOR IMPLEMENTING FILE VIEW FUNCTIONALITY IN YOUR DMS

This chapter describes the minimum steps required to add file view capabilities using AutoVue with your DMS. Once you have completed these steps, you may proceed to the next chapter about adding other functionality. The next chapter describes the steps required to add advanced functionalities such as: searching the DMS, browsing the DMS, creating markups, performing conversions, and so on.

As mentioned in the **Overview** document, the AutoVue Integration SDK bundles a sample integration called Sample Integration for Filesys DMS. The purpose of this sample is to guide and help you in getting familiar with the integration framework. This sample also acts as a good starting point for building your own integration between AutoVue and your DMS.

To learn more about the sample integration, refer to the appendix in this document.

The following sections describe the steps you need to follow in order to implement basic file view functionality using AutoVue in your DMS. Each step has an excerpt of code to show you an example of how the Sample Integration for Filesys DMS was implemented.

5.1 Step 1: Create Your Main DMS Servlet by Extending the 'Vuelink' Class

As discussed in section 4, Integration Design, the framework provides a base class called `Vuelink`, which is a servlet implemented in the SDK in the `com.cimmetry.vuelink` package. It provides all the needed services to handle the request and responses. In most cases for implementing your DMS servlet, just deriving a new class from `Vuelink` class is sufficient.

The following excerpt of code shows the implementation of the `FilesysVuelink` servlet in the `com.cimmetry.vuelink.filesys` package.

```
package com.cimmetry.vuelink.filesys;

import com.cimmetry.vuelink.*;

/**   */
public class FilesysVuelink extends Vuelink {
}
```

If needed, you can extend the behavior of this servlet by it implementing new methods or overriding existing ones.

5.2 Step 2: Define Your Unique Document Identifier by Implementing Docid Interface

AutoVue and the DMS exchange several types of files, such as base document, XRefs, markups, renditions, and so on. To keep the correct mapping between the handled files and their original copies in the DMS repository, an identification mechanism is needed. For this purpose, the framework provides us with the `DocID` interface. This key concept must be implemented by any class responsible for building the unique identifiers of exchanged files. The framework encodes DocID in Base64 when sending it back to AutoVue server.

The `DocID` interface standardizes the serialization and deserialization of the DMS file identifiers by deriving the `java.io.Serializable` interface (See the following excerpt of code). You must implement your own `DocID` class containing anything you want, such as string, integer, binary data, and so on. Keep in mind that the `DocID` data members that are not essential to the uniqueness of the ID should be tagged as 'transient' so that they do not become serialized.

```
package com.cimmetry.vuelink.core;

/**   */
public interface DocID extends java.io.Serializable {
}
```

In the Sample Integration for Filesys DMS, we coded the `FilesysDMSDocID` class in the backend package (`com.cimmetry.vuelink.filesys.backend`) that implements the *`DocID`* interface and builds a unique identifier for each file.
Note: It may help to think of the backend class as a wrapper around your DMS API. Implementing the DMSBackend interface is optional. To learn more about the backend package, refer to the **Appendix**.
Inside the Filesys DMS repository the path for each file is obviously unique and can be used as an identifier. When constructing a `FilesysDMSDocID` object, the `m_id` member is set to the file path.

```
package com.cimmetry.vuelink.filesys.backend;

/**   */

public class FilesysDMSDocID implements  DocID, DMSDefs{

    /** File instance representing the document. */
    private transient File m_file;

    /** document ID*/
    private final String m_id;
•••
    public FilesysDMSDocID(final File file, …, …){
        m_id = file.getPath();
        …
    }
•••
}
```

5.3 Step 3: Create an Open Action class that returns your DocID

When you select a document to view, the first request the AutoVue Server sends is an `open` request asking for the `DocID` of this document. You must create the ActionOpen class in your integration by implementing the DMSAction interface. The framework will automatically find your class that handles this request and execute it. You must also implement the execute method which returns the unique DocID for the document being viewed.

In the Sample Integration for Filesys DMS, as shown in the following excerpt of code, the `ActionOpen` class realizes the `DMSAction` interface and implements the `execute` method. The `execute` method returns the docID obtained from `openFile` method of the backend's API. Although implementing the DMSBackend interface is optional, the Sample Integration for Filesys implemented this interface as an example to show how you can use it in your own integration. You can think of DMSBackend interface as a wrapper around API provided by your DMS SDK. Normally, any DMS provides an API which allows basic document check-in/out and getting/setting attributes.

```
package com.cimmetry.vuelink.filesys.actions;

/**   */

public class ActionOpen implements DMSAction, DMSDefs{
•••
public Object execute(final DMSContext    context,
                      final DMSSession    session,
                      final DMSQuery      query,
                      final DMSArgument[] args
                      ) throws VuelinkException {
•••
// open action returns the DocID
DocID docID =
       ((FilesysDMSBackend)context.getBackendAPI()).openFile
       (context.getBackendSession(session),params);
•••
return docID;
}
```

But how does the framework find the `ActionOpen` class? If you do not place your DMSAction classes in same package as your DMS Servlet, the framework retrieves it from the web.xml descriptor file. In this case, each action class should be registered in this file as an init-parameter. The `ActionOpen` class has `dms.action.`**`Open`** as a parameter name and its value should be a fully qualified class name. In the case of the Sample Integration for Filesys, this is `com.cimmetry.vuelink.filesys.actions.`**`ActionOpen`** as the parameter value. The `FilesysVulinkServlet` uses this init parameter to locate, register, and instantiate the `ActionOpen` class.

```
<init-param>
    <param-name>dms.action.Open</param-name>
    <param-value>com.cimmetry.vuelink.filesys.actions.ActionOpen</param-value>
</init-param>
```

For more information on the behavior of `ActionOpen` class, we advise you to (1) closely examine the source code and (2) run the **filesys** project in Eclipse in debug mode, set breakpoint as shown in the following figure, then follow the execution step by step. This will give you more insight on the behavior of this class.

```
53          // The action name in the query must be "open"
54          if (!"open".equalsIgnoreCase(query.getActionName())) {
55              throw new VuelinkException(DMS_ERROR_CODE_UNKNOWN_ERROR,
56                                         "Invalid action name within query:
57                                         query.getActionName());
```

Figure 6

In the Sample Integration for Filesys DMS, the `ActionOpen` class relies on the `openFile` method to obtain a docID of a file. This method has two parameters:

- The session information to connect to the backend.

- The information needed to open the file (i.e., Filesys DMS data repository and name of the file)

```
public DocID openFile(DMSBackendSession session, Hashtable<String, String> _params)
                                                        throws VuelinkException {
```

This method returns the docID of the file in the DMS. If it fails, it throws Vuelink exception.

Now, what does the `openFile` method do? It essentially asks the file from the filesys DMS repository through the `getFile` method by providing its name, its root directory, and its version (version is optional). Once the file is returned back by the DMS, it builds its unique document identifier and returns it back to the ActionOpen class.

Note: When the number of the version is not provided, the Filesys DMS system returns the latest version of this document.

```
 // Get the file from the Filsys DMS repository
 try{
    File docFile = m_filesysInfo.getFile(rootDir, fileName, version);
    if (docFile != null) {
    // Build a unique docID for the returned file
       docID = new FilesysDMSDocID(docFile, null, version);
    }
}
 catch(Exception e){
    throw new VuelinkException(DMSDefs.DMS_ERROR_CODE_ERROR, e.getMessage());
 }
return docID;
```

For more information, examine the code and use the debugger to learn more about the actual behavior of this class.

5.4 Step 4: Create a Get Property action to return Document Name

AutoVue sends several `GetProperties` requests to know if it already has the most recent copy of the document in its cache. The first request sent is about the name of the file identified by a docID. This is done through the CSI_DocName property.

To handle get property requests, you have two options. You can either define a single class called `ActionGetProperties` that implements DMSAction or you can have separate classes that implement the `DMSGetPropAction` interface.

Use the following excerpt of code to create your own ActionGetProperties class. You can retrieve the list of properties from the query object passed as parameter to execute method. You can then loop through the properties list and retrieve its value from your DMS.

Remember, you do not need to register your class as long as it is located in the same package as your DMS servlet class. Otherwise, you need to register it in the web.xml descriptor file.

```
package com.acme.actions;

/**   */

public class ActionGetProperties implements DMSAction, DMSDefs{
•••
public Object execute(final DMSContext    context,
                      final DMSSession    session,
                      final DMSQuery      query,
                      final DMSArgument[] args
                      ) throws VuelinkException {
•••
Property[] props = query.getProperties();
String propname = props[i].getNme();

// GetProperty action returns attribute values
        if(propname.equals(DMSProperty.CSI_DocName) {
                // return doc name
        } else if(propname.equals(DMSProperty.CSI_DocNameIsMultiContent) {
                // return is multi content
        } else if(propname.equals(DMSProperty.CSI_DocDateLastModified) {
                // return is date last modified
        } else if(propname.equals(DMSProperty.CSI_DocSize) {
                // return is doc size
        }

•••
}
```

In the Sample Integration for Filesys, we chose the latter approach of having separate classes that implement the `DMSGetPropAction` interface. For each property asked, we implemented a class to return it. But since this is not an optimal way to do things, we opted to use a single class instead (e.g., `GetPropDefault` class) which takes care of several individual properties. Refer to section 6.1.9 for more information.

The following excerpt of code illustrates the implementation of the `CustomGetDocName` class in the Sample Integration for Filesys DMS. It gets the document name from the `GetFilesysProperty` class, then returns it to the AutoVue Server.

```
package com.cimmetry.vuelink.filesys.propactions;

/**   */

public class CustomGetDocName extends GetFilesysProperty implements DMSGetPropAction {
…
        public DMSProperty execute(DMSContext context, DMSSession session,
                        DMSQuery query, DMSArgument[] args, Property property)
                        throws VuelinkException {
                final DocID  docID = query.getDocID();
                …
                Hashtable attrs = getAttrs(((FilesysDMSBackend)context.getBackendAPI()),
                                        context.getBackendSession(session),query, docID);
                DMSProperty retProp = new DMSProperty(Property.CSI_DocName,
                                                        (String)attrs.get("DocName"));
                …;
                return retProp;
        }
}
```

As explained in section 4.3 **ActionGetProperties Interface**, each individual property class realizes the framework interface `DMSGetPropAction` by implementing the `execute` method. Given a docID, the `getAttrs` method returns a Hashtable of attributes of the correspondent document. One of these attributes is the document name, which is returned as `DMSProperty` object. Refer to the **Appendix** for the reason for implementing the `GetFilesysProperty` class and for more details on this implementation.

Finally, to allow the framework to locate, register an instantiate the `CustomGetDocName`, we must register it in the web.xml file. As illustrated in the following code, this class is registered with the parameter name `dms.getprops.CSI_DocName` and the parameter value `com.cimmetry.vuelink.filesys.propactions.CustomGetDocName`.

```
<init-param>
  <param-name>dms.getprops.CSI_DocName</param-name>
  <param-value>com.cimmetry.vuelink.filesys.propactions.CustomGetDocName</param-value>
</init-param>
```

Note:
For this property class, we have chosen a different name from the one suggested by the framework. The default name has the format `GeProp<property name>`; however, in this case the name of the class could be `GetPropCSI_DocName`.

For more information, examine the code and use the debugger to learn more about the real behavior of this class.

5.5 Step 5: Create a GetProperty action to return Document Date Last Modified and Size

The AutoVue Server sends a second `GetProperties` request asking for the date of the last modification and the size of the document (i.e., `CSI_DocDateLastModified` and `CSI_DocSize` properties). The implementation of the class responsible for returning these properties is very similar to the `CSI_DocName` presented in the previous section.

Again, to handle get property requests, you have two options. You can either define a single class called `ActionGetProperties` that implements DMSAction, or you can have separate classes that implement the `DMSGetPropAction` interface.

Use the following excerpt of code to create your own ActionGetProperties class. You can retrieve the list of properties from the query object passed as parameter to execute method. You can then loop through the properties list and retrieve its value from your DMS.

Remember, you do not need to register your class as long as it is located in same package as your DMS servlet class. Otherwise, you need to register it in the web.xml descriptor file.

```
package com.acme.actions;

/**   */

public class ActionGetProperties implements DMSAction, DMSDefs{
•••
public Object execute(final DMSContext    context,
                      final DMSSession    session,
                      final DMSQuery      query,
                      final DMSArgument[] args
                      ) throws VuelinkException {
•••
Property[] props = query.getProperties();
String propname = props[i].getNme();

// GetProperty action returns attribute values
        if(propname.equals(DMSProperty.CSI_DocName) {
                // return doc name
        } else if(propname.equals(DMSProperty.CSI_DocNameIsMultiContent) {
                // return is multi content
        } else if(propname.equals(DMSProperty.CSI_DocDateLastModified) {
                // return is date last modified
        } else if(propname.equals(DMSProperty.CSI_DocSize) {
                // return is doc size
        }

•••
}
```

In the Sample Integration for Filesys, we chose latter approach of having separate classes that implement the `DMSGetPropAction` interface. The `GetPropCSI_DocDateLastModified` class realizes the `DMSGetPropAction` interface by implementing the execute method, and it is derived from the `GetFilesysProperty` class. The following code shows this method implementation in more detail.

```
package com.cimmetry.vuelink.filesys.propactions;

/**   */

public class GetPropCSI_DocDateLastModified extends GetFilesysProperty
                                                    implements DMSGetPropAction {
        …
        public DMSProperty execute(DMSContext context, DMSSession session,
                                DMSQuery query, DMSArgument[] args, Property property)
                                                    throws VuelinkException {

                final DocID docID = query.getDocID();
                Hashtable attrs = getAttrs(((FilesysDMSBackend)context.getBackendAPI()),
                                        context.getBackendSession(session),query, docID);
                final Date lastModifiedDate = (Date)attrs.get("DateLastModified");
                DMSProperty retProp = new DMSProperty(Property.CSI_DocDateLastModified,
                                        lastModifiedDate.toString());
                return retProp;
        }
}
```

Given the docID extracted from the *query* parameter, we get a set of attributes of the correspondent document through the inherited **`getAttrs()`**method. From the returned Hashtable, we get the date of the last modification attribute and we return it as `DMSProperty`.

For this property action, we chose a name that respects the naming convention suggested by the framework, and we registered it in the web.xml file as shown in the following code.

```
<init-param>
  <param-name>dms.getprops.CSI_DocDateLastModified</param-name>
    <param-value>
      com.cimmetry.vuelink.filesys.propactions.GetPropCSI_DocDateLastModified
    </param-value>
</init-param>
```

If we had chosen not to register the class in the web.xml file, the framework would have first looked for it in same package of the FilesysVuelinkServlet class. If that had failed, it would have considered the `GetPropDefault`, as presented in the section 6.1.9.

For more information, examine the code and use the debugger to learn more about the real behavior of this class.

5.6 Step 6: Create a Download action to return Document Content

The AutoVue Server checks in its cache repository if it has a more recent copy of the document. If no more recent copy exists, AutoVue Server tries to fetch the file from the DMS repository by calling the Download Action.
You must create the `ActionDownload` class in your integration by implementing DMSAction interface. You must also implement the execute method which returns `FileInputStream object`. The framework automatically streams the file content back to the AutoVue Server.

In the Sample Integration for Filesys DMS, the following excerpt of the code presents the implementation of the `ActionDownload` class. Note that like any action class, this class realizes the `DMSAction` class and implements the `execute()` method. Given the docID of the document to download, the `execute()` method calls the `checkout()` method and downloads the file as `FileInputStream` object, and returns the stream. The rest is done by the Vuelink class before passing it back to the AutoVue Server. If the download operation fails, a `VuelinException` is thrown.

```
package com.cimmetry.vuelink.filesys.actions

/**   */

public class ActionDownload implements DMSAction, DMSDefs{
•••
public Object execute(final DMSContext    context,
                      final DMSSession    session,
                      final DMSQuery      query,
                      final DMSArgument[] args
                      ) throws VuelinkException {
•••
// chekout the instance file of the document
final FileInputStream doc =
       ((FilesysDMSBackend)context.getBackendAPI()).
              checkout(context.getBackendSession(session),docID);
•••
return doc;
}
```

The registration of the action download in the web.xml file is done, as indicated in the following excerpt of code.

```
<init-param>
    <param-name>dms.action.Download</param-name>
    <param-value>com.cimmetry.vuelink.filesys.actions.ActionDownload</param-value>
</init-param>
```

For more information, examine the code and use the debugger to learn more about the real behavior of this class.

This method gets a copy of a file from the DMS repository by invoking the Filesys DMS `getFile()` method. It has two parameters:

- The session information to connect to the DMS
- The docID of the file to get

```
public FileInputStream checkout(DMSBackendSession session, DocID docID) {
       FilesysDMSDocID fsID = (FilesysDMSDocID)docID;
       FileInputStream fis = null;
       try {
              fis = new FileInputStream(m_filesysInfo.getFile(fsID));
       } catch (FileNotFoundException e) {
              System.out.println("File not found" + fsID.getName());
       }catch(Exception e){
              m_logger.error(DMSDefs.DMS_ERROR_CODE_ERROR + e.getMessage());
       }
       return fis;
}
```

The copy of the document is returned as `FileInputStream` object.

5.7 Step 7: Implement Rest of Actions and Register in web.xml

Implement DMSAction interface to create a skeleton for the following action classes in your integration:

- ActionDelete
- ActionSave
- ActionSetProperties

For each action, you must add the `execute` method. At this point, you can leave the `execute` method empty; that does nothing. Actually implementing these actions is optional and are explained in more detail in the next chapter. For example, if you plan to add delete functionalioty to your integration, you can refer to section 6.11 Implementing File Delete Action.

Review the following code excerpt:

```
public class ActionDelete
        implements DMSAction, DMSDefs {

        public Object execute(final DMSContext    context,
                              final DMSSession    session,
                              final DMSQuery      query,
                              final DMSArgument[] args
                              ) throws VuelinkException {
                // TODO…

        }
}
```

As with ActionOpen and ActionDelete, if you place ‘ActionDelete’/ ’ActionSave’/ ’ActionSetProperties’ in the same package as your DMS Servlet, the framework will automatically find them. Otherewise, you need to register them in web.xml. In the case of the sample integration for Filesys, these actions were under the ‘actions’ package and therefore had to be registered in web.xml

```
 <init-param>
     <param-name>dms.action.Delete</param-name>
     <param-value>com.cimmetry.vuelink.filesys.actions.ActionDelete</param-value>
 </init-param>
```

6. STEPS FOR IMPLEMENTING ADVANCED INTEGRATION FUNCTIONALITY IN YOUR DMS

This section describes additional functionality that you can choose to add to your integration but whose implementation is optional.

Note: The following sections assume that you have already implemented file view functionality in your DMS as outlined in previous chapter.

6.1 Handling Document Attributes

Each document has a set of attributes which characterizes it. When we work on one these documents we often need one or several of its attributes. Asking the DMS to provide the value of each attribute each time we need it is certainly not the best solution. A more effective solution consists of acquiring, in one shot, the whole set of attributes from DMS the first time they are needed. Thereafter, we can use this set of attributes each time we need an attribute.

This is the reason why we decided to implement the class `GetFilesysProperty` in the Sample Integration for Filesys.

```
package com.cimmetry.vuelink.filesys.propactions;

/**   */

public class GetFilesysProperty implements DMSDefs {
        …
        protected Hashtable getAttrs(final FilesysDMSBackend be, DMSBackendSession beSession,
        final DMSQuery query, DocID docID) throws VuelinkException {
        …
        Hashtable attrs = (Hashtable) query.getQueryData("attrs");
        …
        attrs = be.getAttributes(null, docID);
        ..
        return attrs;
        }
}
```

Note that this class is not a property class and does not realize the `DMSProperty` interface and does not implement the `execute()` method; we therefore do not need to register it in the web.xml file. This class supports all the property classes that need to use attributes document. This class gets the attributes from the DMS repository by means of the `getAttributes()` method of the Filesys DMS backend class (i.e., `FilesysDMS` class).

This method returns a list of attributes of a document file in the DMS repository. These attributes are returned in `Hashtable` object. Here is the list of the attributes:

- **`DocName`:** The name of the file. It is returned as a `String` object.
- **`DateLastModified`:** The date the document file was last modified. It is returned as a `java.util.Date` object
- **`DocSize`:** The size of the file. It is returned as an `Integer`.
- **`DocFormat`:** Document format (i.e., "document" or "folder"). It is returned as `String` object.
- **`Version`:** The version number of a document. It is returned as `String` object.
- **`VersionsNumber`:** The number of versions of a document. It is returned as a `String` object.

Given the docId of the document, this method asks the FilesysDMS systems to provide it with the set of attributes. As shown in the following code, this is done through calling the *`m_filesysInfo`*`.getAttributes()` method.

```
package com.cimmetry.vuelink.filesys.backend;

/**   */

public Hashtable getAttributes(DMSBackendSession session, DocID docID) {
        FilesysDMSDocID fsID = (FilesysDMSDocID)docID;
        Hashtable attrs = null;
        try{
                attrs = m_filesysInfo.getAttributes(fsID);
        }catch(Exception e){
                m_logger.error(DMSDefs.DMS_ERROR_CODE_ERROR + e.getMessage());
        }
        return attrs;
}
```

6.2 Returning External References (XREFS)

Chapter 5 discussed the case of viewing single document. Often, documents are compound and have many associated files or External Reference files (XRefs). In this case, the AutoVue Server asks for XRefs by passing `CSI_XREFS` within the `GetProperties` request. The response to this request is provided by `GetCSI_XREFS`, the XRefs property class. Refer to the JavaDMAPI.pdf manual for the actual format of an XRefs response.

In the Sample Integration for Filesys, since `GetCSI_XREFS` is a property class it realizes the `DMSGetPropAction` and implements the `execute()` method. The following code shows all the imported classes from the AutoVue Integration SDK framework. All these classes are referenced in the `execute()` method parameters. Refer to the **Appendix** for more information on these parameters.

```
package com.cimmetry.vuelink.filesys.propactions;

import com.cimmetry.vuelink.context.DMSContext;
import com.cimmetry.vuelink.defs.DocID;
import com.cimmetry.vuelink.defs.VuelinkException;
import com.cimmetry.vuelink.filesys.backend.FilesysDMSBackend;
import com.cimmetry.vuelink.filesys.backend.FilesysDMSDocID;
import com.cimmetry.vuelink.property.Property;
import com.cimmetry.vuelink.propsaction.DMSGetPropAction;
import com.cimmetry.vuelink.propsaction.DMSProperty;
import com.cimmetry.vuelink.propsaction.arguments.DMSArgument;
import com.cimmetry.vuelink.query.DMSQuery;
import com.cimmetry.vuelink.session.DMSBackendSession;
import com.cimmetry.vuelink.session.DMSSession;

public class GetPropCSI_XREFS  implements DMSGetPropAction {
```

The following excerpt of code shows how the `execute()` method builds a ***CSI_XREFS*** DMSProperty from the list of XRefs files returned by the `dmsListXrefs` method of the FilesysDMS backend class. The ***CSI_XREFS*** DMSProperty is returned to the `Vuelink` servlet which will do the rest and provides the response to the AutoVue Server.

```
public DMSProperty execute(DMSContext context, DMSSession session,
        DMSQuery query, DMSArgument[] args, Property property)
        throws VuelinkException {

        DMSProperty retProp =
                new DMSProperty(Property.CSI_XREFS,
                buildXREFSProperty(((FilesysDMSBackend)context.getBackendAPI()),
                context.getBackendSession(session), query.getDocID()));
        m_logger.debug("got the xrefs property: " + retProp);
        return retProp;
}
```

But how are the XRefs files returned from the DMS repository to the `GetCSI_XREFS` class? This is done through the `dmsListXRefs` of the FilesysDMS backend class, which talks to DMS repository and gets from it the list of the XRefs file as vector. For each element of the vector, we build a `DMSProperty` as specified in the DMAPI specification.

```
private Property[] buildXREFSProperty(FilesysDMSBackend be,
                                      DMSBackendSession beSession, DocID docID) {
        Vector xrefsDocIds = be.dmsListXRefs(beSession, docID);
        DMSProperty[] xrefs = new DMSProperty[xrefsDocIds.size()];
        …
        for (int i = 0; i < xrefsDocIds.size(); i++)
        {
                DMSProperty xrefProp[] = new DMSProperty[2];
                xrefProp[0] = new DMSProperty(Property.CSI_DocID,
                                                        (DocID)xrefsDocIds.get(i));

                xrefProp[1] = new DMSProperty("Name",
                                ((FilesysDMSDocID)((DocID)xrefsDocIds.get(i))).getName());
                xrefs[i] = new DMSProperty(Property.PROP_XREF,xrefProp);
        }
        return xrefs;
}
```

The `GetPropCSI_XREFS` is registered in the web.xml file as indicated in the following code excerpt.

```
<init-param>
    <param-name>dms.getprops.CSI_XREFS</param-name>
    <param-value>com.cimmetry.vuelink.filesys.propactions.GetPropCSI_XREFS</param-value>
</init-param>
```

For more information, examine the code and use the debugger to learn more about the real behavior of this method.

This method asks the Filesys DMS system for the list of XRefs associated to a given document by providing its `docID`. After it receives the vector of XRefs files elements, it builds a docID for each element. Finally, it returns the list of the docIDs as a vector.

For more information, examine the code and use the debugger to learn more about the real behavior of this method.

```
public Vector<DocID> dmsListXRefs(DMSBackendSession session, DocID docID) {
        FilesysDMSDocID fsDocID = (FilesysDMSDocID) docID;
        …
        xrefsInfos = m_filesysInfo.listXRefs(fsDocID);

        xrefs = new Vector<DocID>();
        …
        for (int i = 0 ; i < xrefsInfos.size() ; ++i) {
                xrefs.add(new FilesysDMSDocID((DocInfo)xrefsInfos.get(i)));
        }
        return xrefs;
}
```

6.3 Handling Markups

When users try to view any Redlines/Markups, the AutoVue Server sends a `GetProperties` request asking for the list of markups of this document by passing *`CSI_Markups`* property within the request. The response to this request is done through the `GetPropCSI_Markups property` class. The response must be specified in two parts: a 'GUI' part and a 'Markup' part. Refer to the JavaDMAPI.pdf manual for the actual format of the Markups request and response.

For the 'GUI' part, we must specify the structure of the GUIs with which the user will interact. The 'GUI' part is itself composed of three sections: 'Display Options', 'Edit', and 'Display'. The 'Display Options' specifies whether or not users are allowed to perform some operations on markups.

In the Sample Integration for Filesys, the following excerpt of code builds several properties and sets their value to `true` or `false`. Each of these properties is dedicated to a particular

operation. For instance, in the property *AllowDelete*, Markups is set to `true`, which allows user to delete markups. The last line of the code shows us how all the properties are grouped in a single property labeled *DisplayOptions*.

```
package com.cimmetry.vuelink.filesys.propactions;

DMSProperty guiProps[] = new DMSProperty[3];
DMSProperty DispOptArr[] = new DMSProperty[7];
DispOptArr[0] = new DMSProperty("AllowDelete","true");
DispOptArr[1] = new DMSProperty("ShowPreviousVersions","true");
DispOptArr[2] = new DMSProperty("AllowNew","true");
DispOptArr[3] = new DMSProperty("AllowImport","false");
DispOptArr[4] = new DMSProperty("AllowExport","false");
DispOptArr[5] = new DMSProperty("AllowNewLayers","false");
DispOptArr[6] = new DMSProperty("AllowModifyLayers","false");

guiProps[0] = new DMSProperty("DisplayOptions", DispOptArr);
```

The ‘Edit’ section specifies the GUI Elements with which we want to populate the ‘Save Markup Dialog Box’. The ‘Save Markup Dialog Box’ contains two GUI Elements: an ‘Edit Box’ and a ‘Drop Down Combo Box’.
The label of the ‘Edit Box’ is ‘Name’ and its control ID is *CSI_DocName*.
The label of the ‘Drop Down Combo Box’ is ‘Markup Type’ and its control ID is `CSI_MarkupType`. The ‘Drop Down Combo Box’ element contains three selections: `normal`, `master` and `consolidated`, with the default value set to `normal`.

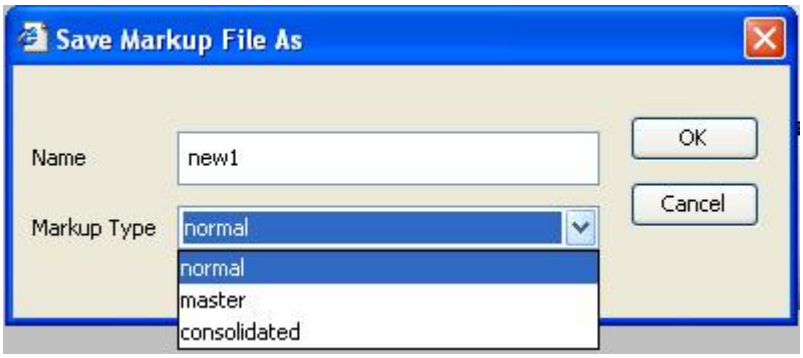


Figure 7. Save Markup dialog

In the Sample Integration for Filesys, the following excerpt of code builds the `GUIElementCombo` property, which specifies a ‘Drop Down List’ that contains three selections: `normal`, `master` and `consolidated`. The default selection is set to `normal`. Notice that the last line of code attaches the `GUIElementCombo` property in a `DMSProperty` labeled *Edit*.

```
String comboVals[] = new String[3];
comboVals[0] = "normal";
comboVals[1] = "master";
comboVals[2] = "consolidated";
EditArr[1] = new GUIElementCombo("CSI_MarkupType", "Markup Type", "normal",
                                                                    comboVals, false);
guiProps[2] = new DMSProperty("Edit",EditArr);
```

The 'Display' section specifies 'Read-Only' properties to be displayed in tabular format inside 'Open Markup Dialog Box'.

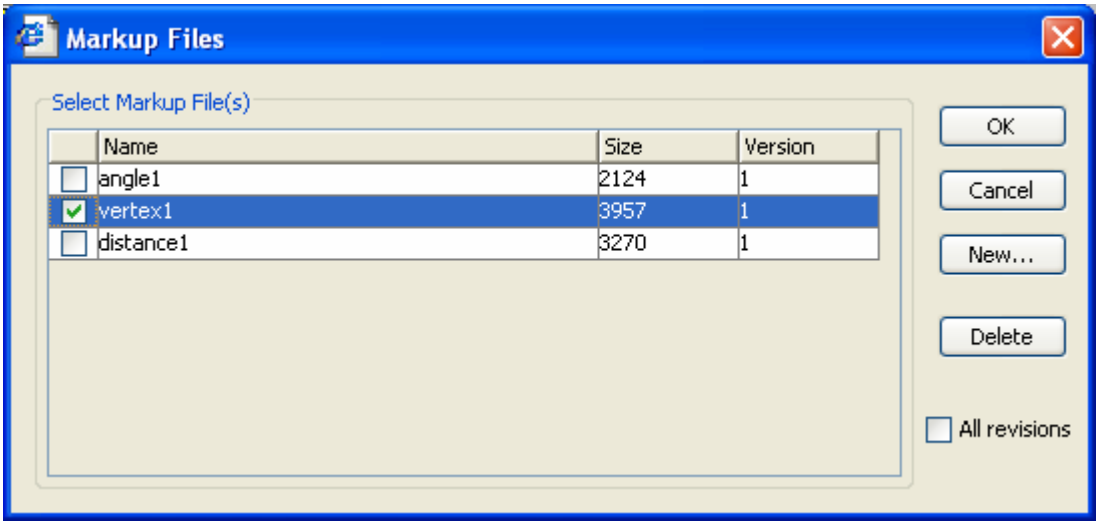


Figure 8. Markup files dialog

In the Sample Integration for Filesys, the following code defines three GUI elements that compose the 'Open Markup Dialog Box': document name, document size and the version of the document. Each of these element is encapsulated as a `DMSProperty` labeled respectively: ***`CSI_DocName`***, ***`CSI_DocSize`***, and ***`CSI_Version`***. Finally all these properties are attached to a `Display` DMSProperty object.

```
DMSProperty DispArr[] = new DMSProperty[3];
DispArr[0] = new DMSProperty(Property.CSI_DocName,"40");
DispArr[1] = new DMSProperty(Property.CSI_DocSize,"10");
DispArr[2] = new DMSProperty(Property.CSI_Version,"10");

guiProps[1] = new DMSProperty("Display", DispArr);
```

Note: All 'GUI' properties (i.e., ***`DisplayOptions`***, ***`Display and Edit`***) must be attached to a `DMSProperty` object with ***`PROP_GUI`*** identification.

The second part 'Markup' of the response specifies the list of markups associated with the current document. Each element of the list must be encapsulated in a Markup `DMSProperty`. For more information, refer to the JavaDMAPI.pdf manual for the actual format of Markup response. The list of the markups is returned by the `dmsListMarkups` method of the FilesysDMS backend class.

In the Sample Integration for Filesys, the following excerpt of code shows all the needed information about each markup. This information includes: the docID, the name of the markup, the size of the markup, and the version of its base document. Each information is built inside a single DMSProperty object respectively labeled: *CSI_DocID*, *CSI_DocName*, *CSI_DocSize*, and *CSI_Version*. As indicated before last line in the code excerpt, all this DMSProperties are gathered in a single `DMSProperty` property labeled *PROP_MARKUP*.

```
DMSProperty guiProps[] = buildMarkupGui(be, beSession, docID);
//Gets the list of markups from the DMS
Vector mrkDocIds = be.dmsListMarkups(beSession, docID);
DMSProperty markup[] = new DMSProperty[mrkDocIds.size()+1];
markup[0] = new DMSProperty(Property.PROP_GUI,guiProps);

for (int i = 0; i < mrkDocIds.size(); i++)
{
        DMSProperty mrkProp[] = new DMSProperty[4];
        mrkProp[0] = new DMSProperty("CSI_DocID", (DocID)mrkDocIds.get(i));
        mrkProp[1] = new DMSProperty("CSI_DocName",
        ((FilesysDMSDocID)((DocID)mrkDocIds.get(i))).getName());
        mrkProp[2] = new DMSProperty(Property.CSI_DocSize,
                        new Long(((FilesysDMSDocID)
                        ((DocID)mrkDocIds.get(i))).getFile().length()).toString());
                       Hashtable attrs = getAttrs(be, beSession,query, docID);
        mrkProp[3] = new DMSProperty(Property.CSI_Version,
                                                        (String)attrs.get("Version"));

        markup[i+1] = new DMSProperty(Property.PROP_MARKUP,mrkProp);
}
        …
return markup;
```

Finally, the `execute()` method attaches the *PROP_GUI* and *PROP_MARKUP* properties in a *CSI_Makups* property and returns it to the `Vuelink` servlet.

The registration of the `GetPropCSI_Markups` class is done as indicated below.

```
<init-param>
    <param-name>dms.getprops.CSI_Markups</param-name>
    <param-value>
       com.cimmetry.vuelink.filesys.propactions.GetPropCSI_Markups
    </param-value>
</init-param>
```

For more information, examine the code and use the debugger to learn more about the real behavior of this method.

Note: For saving and deleting Markups, refer respectively to sections Action Save and Action Delete.

The dmsListMarkups method asks the Filesys DMS repository for the list of the Markups associated to a given document by providing its `docID`.

```
public Vector<DocID> dmsListMarkups(DMSBackendSession session, DocID docID) {
        Vector<DocID> markups = null;
        FilesysDMSDocID fsDocID = (FilesysDMSDocID) docID;
        Vector marksInfos = null;
        try{
                marksInfos = m_filesysInfo.listMarkups(fsDocID);
        }catch(Exception e){
                m_logger.error(DMSDefs.DMS_ERROR_CODE_ERROR + e.getMessage());
        }
        markups = new Vector<DocID>();
        for (int i = 0 ; i < marksInfos.size() ; ++i) {
                markups.add(new FilesysDMSDocID((DocInfo)marksInfos.get(i)));
        }
        return markups;
    }
```

After it receives the vector of Markups files, it builds a docID for each file. Finally, it returns the list of the docIDs as a vector.

6.4 Handling Renditions

The AutoVue Server allows you to view more than 400 file formats. The viewed files are often large and time-consuming. To enhance performance, AutoVue can generate files in a lightweight format called Cimmetry Metafile (CMF), which is a true replica of the original files. AutoVue also can generate renditions in the TIF format.

When a user wants to view a file, the AutoVue Server sends several requests to the DMS systems through the integration interface. One of these requests is about metafiles. The AutoVue Server sends a `GetProperies` request by passing the `CSI_Renditions` property in it. This means that AutoVue is asking if the DMS already has a metafile associated to the base document. The response to this question will be provided by the `GetPropCSI_Renditions`. (A description of how this response is built is provided later in this section). If the response is yes, the AutoVue Server sends requests to download the original file and the metafile. Next, it verifies if they are a true replica, in which case AutoVue displays the metafile instead the original one.

If the DMS does not have a metafile or the metafile it has is not a true replica of the original, the client (i.e., the applet) makes a request to the AutoVue Server, asking it to generate a metafile of the original file. When the user decides to quit the viewed file, AutoVue sends a request to the DMS asking it to save the generated metafile. Refer to the Action Save section for information on how to build the response for this case.

In the Sample Integration for Filesys, the following excerpt of code shows how the `GetPropCSI_Renditions` class encapsulates the docID returned by the `getMetaRendition()` method of the FilesysDMS backend class in the ***`CSI_DocID`*** DMSProperty object.

```
package com.cimmetry.vuelink.filesys.propactions;

/**   */

public class GetPropCSI_Renditions implements DMSGetPropAction {

private DMSProperty[] buildRenditionProperty(FilesysDMSBackend be,
                    DMSBackendSession beSession, DocID docID) throws VuelinkException{

        DocID rendDocIds = be.getMetaRendition(beSession, docID);

        if (rendDocIds == null) return null;
        //
        DMSProperty[] metaRend = new DMSProperty[1];
        metaRend[0] = new DMSProperty(DMSProperty.CSI_DocID, rendDocIds);
        m_logger.debug("got the docID: " + metaRend);
        return metaRend;
}
```

As illustrated in the following code, the `execute()` method builds a *`CSI_Rendition`* DMSProperty and attaches to it the *`CSI_DocID`* `DMSProperty.` Finally, it returns it to the `Vuelink` servlet, which will do the rest and provide the AutoVue Server with the response.

```
DMSProperty retProp = new DMSProperty(DMSProperty.CSI_Renditions,
        new String[] {"PCRS_TIF","CSI_META"},
        buildRenditionProperty((FilesysDMSBackend)context.getBackendAPI(),
                            context.getBackendSession(session), query.getDocID()));
```

The `GetPropCSI_Renditions` is registered in the web.xml file as indicated in the following code.

```
 <init-param>
     <param-name>dms.getprops.CSI_Renditions</param-name>
     <param-value>
          com.cimmetry.vuelink.filesys.propactions.GetPropCSI_Renditions
     </param-value>
 </init-param>
```

For more information, examine the code and use the debugger to learn more about the real behavior of this class.

The `getMetaRendition` method asks the FilesysDMS repository for the metafile associated with the base document identified by the `docID` passed as parameter. After it receives the metafile, it builds the `docID` and it returns it.

```
public DocID getMetaRendition(DMSBackendSession session, DocID docID) {
        FilesysDMSDocID fsDocID = (FilesysDMSDocID) docID;
        DocInfo metafile = null;
        try{
                metafile = m_filesysInfo.getMetaInstance(fsDocID);

        }
        catch(Exception e){
                m_logger.error(DMSDefs.DMS_ERROR_CODE_ERROR + e.getMessage());
         }
        if (metafile == null){
                return null;
        }else{
                return new FilesysDMSDocID(metafile);
        }
    }
```

6.5 Returning the List of All Properties of the DMS Document

When users select File > Properties, then click the DMS tab (see Figure 9), the AutoVue Server asks for some attributes of the current document by passing the `CSI_ListAllProperties` property element within the `GetProperties` request. The response to this request is done through a property class called `GetPropCSI_AllProperties`. Refer to the JavaDMAPI.pdf manual for the actual format of the ListAllProperties response.

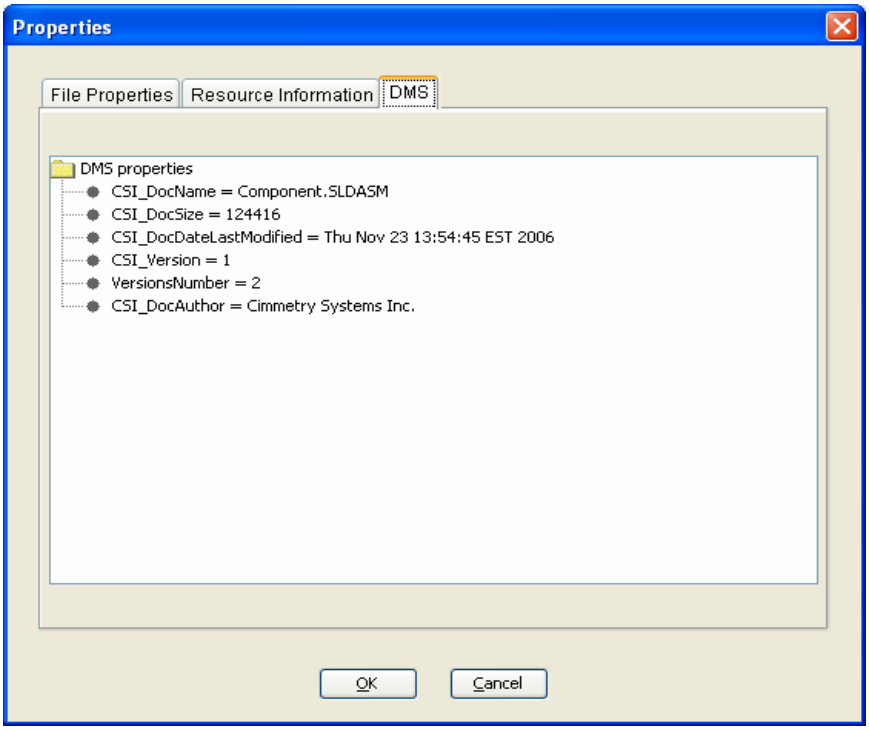


Figure 9. Properties dialog

In the case of the Filesys DMS, all the requested attributes are returned by the `GetFilsysProperty` class. This is why we have simply derived the `GetPropCSI_AllProperties` class from the `GetFilesysProperty` class. Calling the `getAttribues` method of the latter class returns the document attributes. This is shown in the following excerpt of code. After getting the attributes, we build a `DMSProperty` object for each attribute. For instance, we build a DMSProperty named *`CSI_Version`* for the number of document versions. Finally, from these sets of properties, the `execute()` method builds a DMSProperty with the value set to *`CSI_LIstAllProperties`* and returns it to the `Vuelink` servlet.

```
package com.cimmetry.vuelink.filesys.propactions;
….

public class GetPropCSI_ListAllProperties extends GetFilesysProperty
                                          implements DMSGetPropAction {
…
  atttrs = ((FilesysDMSBackend)context.getBackendAPI()).getAttributes(null, docID);
  //List of attributes
  DMSProperty props[] = new DMSProperty[4];
  props[0] = new DMSProperty(DMSProperty.CSI_DocName, );
  props[1] = new DMSProperty(DMSProperty.CSI_DocSize);
  props[2] = new DMSProperty(DMSProperty.CSI_DocDateLastModified);
  props[3] = new DMSProperty(DMSProperty.CSI_Version);
  return props;
```

This `GetPropCSI_AllProperties` class is registered in the web.xml file, as indicated in the following code excerpt.

```
 <init-param>
     <param-name>dms.getprops.CSI_ListAllProperties</param-name>
     <param-value>com.cimmetry.vuelink.filesys.propactions.GetPropCSI_ListAllProperties
 </param-value>
 </init-param>
```

For more information, examine the code and use the debugger to learn more about the real behavior of this class.

6.6 Implementing File Browse

Users may want to browse the DMS repository to select documents for viewing or comparison. In this case, the AutoVue Server sends two `GetProperties` requests. The first request is about the GUIs that will support the definition of the browse operation. The second request is about the result of the browse action performed by the user.

First Request

In the first request, AutoVue asks for the ‘Browse Dialog Box’ structure by passing the ‘GUI’ property with a value set to ‘Browse’ within the request. The response to this first request is done through a property class called `GetPropGUI`. Refer to the JavaDMAPI.pdf manual for the actual format of the GUI property request and response. The GUI section is

identified by the ‘Display’ which specifies the ‘Read-Only' properties to be displayed in tabular manner inside the ‘Browse Dialog Box’.

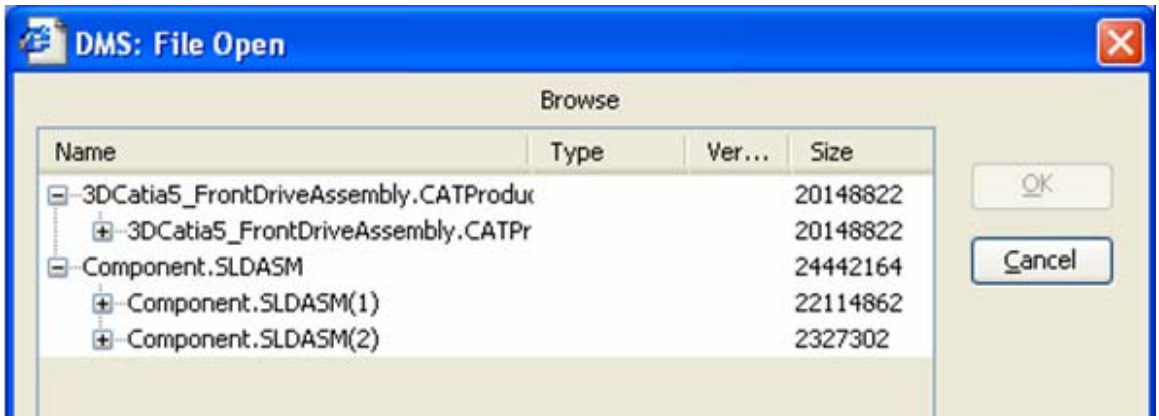


Figure 10. DMS Browse dialog

In the Sample Integration for Filesys, the following excerpt of the code shows how the ‘Browse Dialog Box’ elements are defined. All the elements are specified are DMSProperty objects and it includes: the document name *CSI_DocName*, the document type folder or file *SP_TYPE*, the document version *SP_FileVersion* and the document size *CSI_DocSize*. Each element is a field with a certain size. Note that the ‘Name’ field is a GUI tree where the folders are nodes that can be expanded by the user. All these properties are returned as single property labeled *PROP_GUI_DISPLAY.*

```
final String SP_Type                = "Type";
final String SP_FileVersion = "Version";

DMSProperty[] guiValue = new DMSProperty[4];
guiValue[0] = new DMSProperty(DMSProperty.CSI_DocName,"35");
guiValue[1] = new DMSProperty(SP_Type,"10");
guiValue[2] = new DMSProperty(SP_FileVersion,"6");
guiValue[3] = new DMSProperty(DMSProperty.CSI_DocSize,"14");

DMSProperty[] gui = {new DMSProperty(DMSProperty.PROP_GUI_DISPLAY,guiValue)};
m_logger.info("building GUI for browsing: " + guiValue);
return gui;
```

Second request

The second request sent by the AutoVue Server is about the list of items that are direct children of a node. This request is done by `GetProperties` by passing `CSI_Browse` property within it. The response to this request is handled by `GetProp_Browse` class. This class must return the data that will populate the ‘Browse Dialog Box’.

In the Sample Integration for Filesys, all this information is obtained from calling the `dmsListItemsForBrowse` method of the FilesysDMS backend class. This method returns a vector of docIDs of direct children of the expanded document.

In the Sample Integration for Filesys, the following excerpt of code shows how the `GetProp_Browse` builds properties for the returned document. For each document we return this information about it: Type of document folder or a file *`CSI_ItemType`*, document name *`CSI_DocName`* date of lat modification *`CSI_DocDateLastModified`*, document size *`CSI_DocSize`*, and the version of the document *`CSI_Version`*. Finally, the `execute()` method gathers all the built properties in a single property labeled *`CSI_Browse`* and returns it to the `Vuelink` servlet.

```
package com.cimmetry.vuelink.filesys.propactions;

…

public class GetPropCSI_ListItems implements DMSGetPropAction {
  …
  Vector listItemsInfos = be.dmsListItemsForBrowse(be, beSession, rootID);
  DMSProperty listItems[] = new DMSProperty[listItemsInfos.size()];
  …
  for (int i = 0 ; i < listItemsInfos.size() ; ++i) {

    DocID instId = (DocID) listItemsInfos.get(i);
    Hashtable docAttrs = be.getAttributes(beSession,instId);

    DMSProperty props[] = new DMSProperty[5];
    …
    props[0] = new DMSProperty(DMSProperty.CSI_ItemType,
                                                    DMSProperty.CSI_Folder);
    props[1] = new DMSProperty(DMSProperty.CSI_DocName,
                                                  (String)docAttrs.get("DocName"));
    props[2] = new DMSProperty(DMSProperty.CSI_ItemType,
                                                  (String)docAttrs.get("DocFormat"));
    props[3] = new DMSProperty(DMSProperty.CSI_Version,
                                                  (String)docAttrs.get("Version"));
    props[4] = new DMSProperty(DMSProperty.CSI_DocSize,
                                           ((Long)docAttrs.get("DocSize")).toString());
    …
    listItems[i]= new DMSProperty(DMSProperty.CSI_DocID, instId,props);
```

When the list of items is constructed, the execute method constructs `CSI_Browse` and returns it to `Vulelink` servlet.

The classes `GetPropGUI` and `GetPropCSI_Browse` are registered respectively in the web.xml as indicated below.

```
<init-param>
    <param-name>dms.getprops.CSI_GUI</param-name>
    <param-value>com.cimmetry.vuelink.filesys.propactions.GetPropCSI_GUI</param-value>
</init-param>
```

```
<init-param>
    <param-name>dms.getprops.CSI_ListItems</param-name>
    <param-value>com.cimmetry.vuelink.filesys.propactions.GetPropCSI_ListItems</param-value>
</init-param>
```

For more information, examine the code and use the debugger to learn more about the real behavior of these classes.

The dmsListItemsForBrowse method asks the Filesys DMS repository for the list of direct children of a node given by its `docID`. After it receives the vector of the direct children of the document, it builds a docID for each child. Finally, it returns the list of the docIDs as a vector.

```
public Vector<DocID> dmsListItemsForBrowse(FilesysDMSBackend be, DMSBackendSession beSession,
DocID docID) {

        FilesysDMSDocID fsDocID = (FilesysDMSDocID) docID;
        // modify and rename the the implementation of getInstanceIDs
        Vector browseItemsIDs = null ;
        try{
                browseItemsIDs = m_filesysInfo.listItemsForBrowse(fsDocID);
        }catch(Exception e){
                m_logger.error(DMSDefs.DMS_ERROR_CODE_ERROR + e.getMessage());
        }
        Vector<DocID>  docIDs = new Vector<DocID>();
                for (int i = 0 ; i < browseItemsIDs.size() ; ++i) {
                        docIDs.add(new FilesysDMSDocID((DocInfo)browseItemsIDs.get(i)));
                }
                return  docIDs;
        }
```

6.7 Implementing File Search

Users may want to search for documents in the DMS repository for viewing or comparison. In this case, the AutoVue Server sends two `GetProperties` requests: one is about the GUIs that will support the definition of the search operation and the other is about the result of the search operation that will be displayed on these GUIs. Generally, there are to dialog boxes to define, in the first one we define the search criteria elements and in the second one we define the structure where will be displayed the returned information elements.

Firs Request

In the first request AutoVue asks for the structures of the two dialog boxes discussed above by passing ‘GUI’ property with a value set to ‘Search’ within the request. The response to this first request is done through a property class called `GetPropGUI` (This class is presented in the Browse section above). Please refer to JavaDMAPI.pdf manual for the actual format for GUI property request and response.

The response is specified by two parts: the ‘EDIT’ which specifies GUI Elements of the dialog box with which we want to specify the search criteria. This box includes two GUI Elements: a ‘Drop Down List’ whose label is ‘Criteria’ and control ID is *`CSI_Criteria`* and contains two selections ‘Name’ and ‘Type’ with default value set to ‘Name’ and, an ‘Edit Box’ whose label is ‘value’ and control ID is *`CSI_Entry`* .

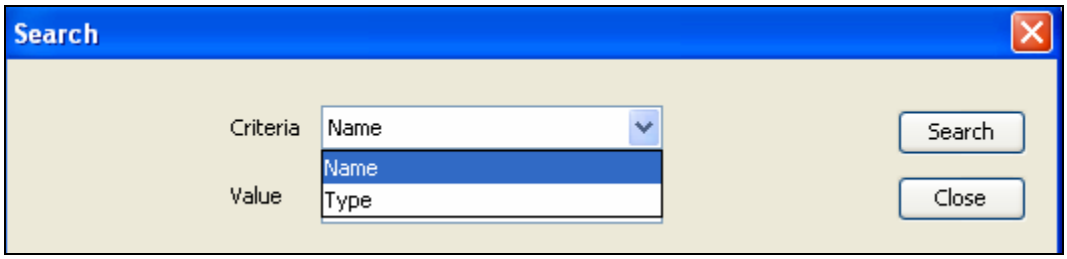


Figure 11. Search dialog

In the Sample Integration for Filesys, the following excerpt of code prepares information for building the first part of the response. It builds a `GUIElementCombo` property for specifying the 'Drop Down List' and it builds a `GUIElementEdit` property for specifying the 'Edit Box'. The two properties are returned in a single property labeled ***PROP_GUI_EDIT.***

```
String [] values = {"Name","Type"};
GUIElementCombo comboForType  =  new GUIElementCombo("CSI_Criteria",
                                                     "Criteria", null, values, true);
…
GUIElementEdit editForName = new GUIElementEdit("CSI_Entry", "Value", null, false);

Property [] p = new Property[2];
p[0] = comboForType;
p[1] = editForName;
…
props = new DMSProperty(Property.PROP_GUI_EDIT, p );
```

In the second part of the response, 'Display' specifies 'Read-Only' properties to be displayed in tabular manner inside the 'Search Dialog Box'.

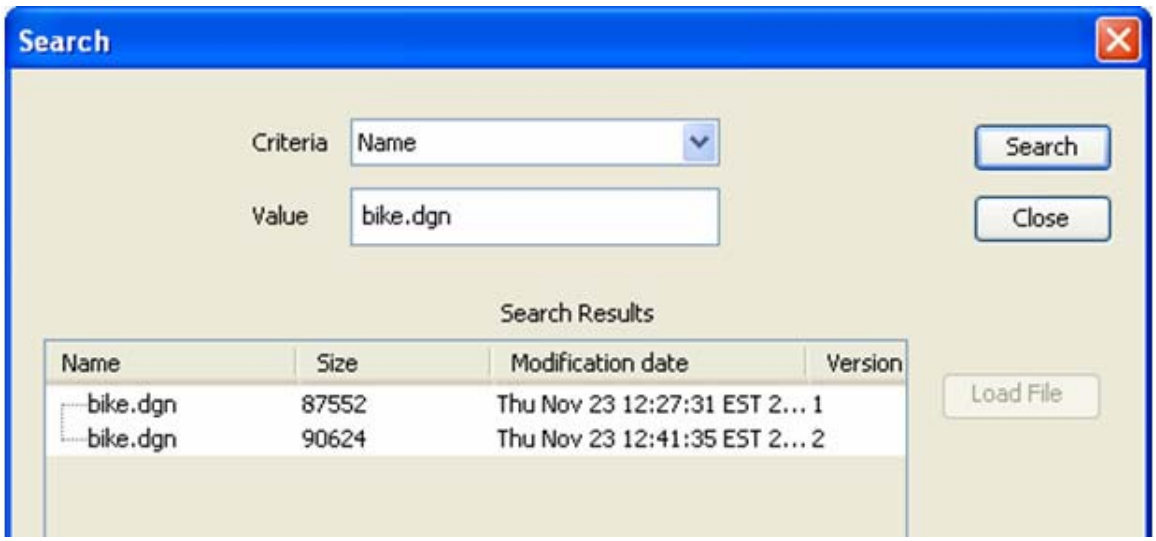


Figure 12. Search results

In the Sample Integration for Filesys, the following excerpt of the code shows how this box is defined. It includes the following information by building the properties: the document

name *CSI_DocName*, the document size *CSI_DocSize*, the date of last modification *CSI_DocDateLastModified* and the version of the document *CSI_Version*. All these properties are returned in a single property labeled *PROP_GUI_DISPLAY*.

```
        DMSProperty[] props = new DMSProperty[4];
        props[0] = new DMSProperty(DMSProperty.CSI_DocName, "18");
        props[1] = new DMSProperty(DMSProperty.CSI_DocSize, "18");
        props[2] = new DMSProperty(DMSProperty.CSI_DocDateLastModified, "18");
        props[3] = new DMSProperty(DMSProperty.CSI_Version, "4");

        return new DMSProperty(Property.PROP_GUI_DISPLAY, props );
```

Second Request

The second request sent by AutoVue server is about list of items that matches the criteria value. This is done through a `GetProperties` request by passing within it *CSI_Search* property. The response to this request is handled by `GetProp_Search` class. This class must return the data that will populate the 'Search Dialog Box'.

In the Sample Integration for Filesys, the result information is obtained in appropriate formats from the method `dmsListItemsForSearch` of the FilesysDMS backend class. The following excerpt of code shows how the `GetProp_Search` builds properties for the returned document. For each document, we return this information about it: Type of document folder or a file *CSI_ItemType*, document name *CSI_DocName* date of last modification *CSI_DocDateLastModified*, document size *CSI_DocSize* and, the version of the document *CSI_Version*. Finally the `execute()` method gathers all the built properties in a single property labeled *CSI_Search* and return it to the `Vuelink` servlet.

```
Vector items = be.dmsListItemsForSearch(docID, rootDir, creteria, type);

for (int i = 0; i < items.size(); i++){
        …
        Hashtable attrs = (Hashtable) query.getQueryData( "attrs");

        attrs = ((FilesysDMSBackend)context.getBackendAPI()).getAttributes(null, docID);
        …
        sProp[0] = new DMSProperty(DMSProperty.CSI_ItemType, DMSProperty.CSI_Folder);
        …
        sProp[0] = new DMSProperty(DMSProperty.CSI_ItemType, DMSProperty.CSI_Document);
        ….
        sProp[1] = new DMSProperty(Property. , (String)attrs.get("DocName"));
        sProp[2] = new DMSProperty(Property.CSI_DocSize, (Long)attrs.get("DocSize"));

        sProp[3] = new DMSProperty(Property.CSI_DocDateLastModified,
                                                (Date)attrs.get("DateLastModified"));
        sProp[4] = new DMSProperty("CSI_Version", (String)attrs.get("Version"));
        sItems[i] = new DMSProperty(Property.CSI_DocID, docID, sProp);
```

The `GetPropCSI_Search` class is registered in the web.xml file as indicated below.

```
<init-param>
    <param-name>dms.getprops.CSI_Search</param-name>
    <param-value>
       com.cimmetry.vuelink.filesys.propactions.GetPropCSI_ Search
    </param-value>
</init-param>
```

For more information, examine the code and use the debugger to learn more about the real behavior of this class.

The dmsListItemsForSearch method asks the Filesys DMS repository for the list of documents that match the search criteria by providing the criteria type, criteria value and the repository root.

```
public Vector<DocID> dmsListItemsForSearch(DocID docID, String root,
                                           String creteria, String type) {
        FilesysDMSDocID fsDocID = (FilesysDMSDocID) docID;
        // comments
        Vector searchItemsIDs = null;
        …
        searchItemsIDs = m_filesysInfo.listItemsForSearch(fsDocID,root,
                                                          creteria, type );
        …
        Vector<DocID>  docIDs = new Vector<DocID>();
        for (int i = 0 ; i < searchItemsIDs.size() ; ++i) {
                docIDs.add(new FilesysDMSDocID((DocInfo)searchItemsIDs.get(i)));
        }
        return  docIDs;
}
```

When it receives the vector of the searched files from the DMS repository, for each element it builds a docID and it returns the list of the docIDs as a vector.

6.8 Handling Versions

When users want to compare the viewed document with another version of the document, they check the 'Document Versions' box in the **File versions** dialog box. At this moment, the AutoVue Server sends a `GetProperties` requests asking the DMS repository for all versions of the current document by passing ***`CSI_Versions`*** property within it. The response request is done through a property class called `GetPropCSI_Versions.` Refer to the JavaDMAPI.pdf manual for the actual format for ***`CSI_Versions`*** property response.

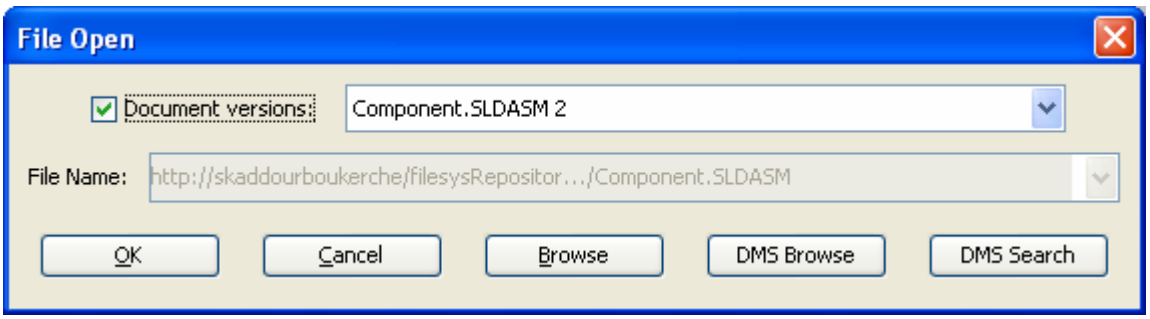


Figure 13. File Open dialog with Document versions for compare

In the Sample Integration for Filesys, to build the response, the `GetPropCSI_Versions` class first receives from `dmsListVersions()` method, which is implemented in the FilesysDMS backend class, a vector of docIDs of all the versions of the document. Second, it builds *`CSI_Version`* property.

The following excerpt of code shows how we build the content of the *`CSI_Versions`* property. For each version of a document we create a *`PROP_VERSION`* property and we attach to it the docID *`CSI_DocID`*, the name *`CSI_DocName`* and the version number *`CSI_Version`* properties. Finally, the list of *`PROP_VERSION`* properties are attached to *`CSI_Versions`* property and returned to the `Vuelink` servlet.

```
Vector versionsDocIDs = be.dmsListVersions(beSession, docID);
DMSProperty[] versions = new DMSProperty[versionsDocIDs.size()];

for (int i = 0; i < versionsDocIDs.size(); i++){
        DMSProperty[] aVersion = new DMSProperty[3];
        DocID doc = (DocID)((Vector) versionsDocIDs.get(i)).get(0) ;
        aVersion[0] = new DMSProperty(DMSProperty.CSI_DocID, doc);
        aVersion[1] = new
                DMSProperty(DMSProperty.CSI_DocName,((FilesysDMSDocID)doc).getName());

        aVersion[2] = new DMSProperty(DMSProperty.CSI_Version, (String)((Vector)
                                                versionsDocIDs.get(i)).get(1));
        versions[i] = new DMSProperty(DMSProperty.PROP_VERSION, aVersion);
        }

return versions;
```

The `GetPropCSI_Versions` class is registered in the web.xml file as indicated above.

```
 <init-param>
     <param-name>dms.getprops.CSI_Versions</param-name>
     <param-value>
        com.cimmetry.vuelink.filesys.propactions.GetPropCSI_Versions
     </param-value>
 </init-param>
```

For more information, examine the code and use the debugger to learn more about the real behavior of this class.

The dmsListVersion method asks the Filesys DMS repository for the list of document versions by providing the docID of the current document.

```
public Vector<DocID> getVersions(DMSBackendSession session, DocID docID) {
        Vector<DocID> versions = null;
        FilesysDMSDocID fsDocID = (FilesysDMSDocID) docID;
        Vector versionsInfos =  null;
        try{
                versionsInfos = m_filesysInfo.listVersions(fsDocID);
        }
            catch(Exception e){
               m_logger.error(DMSDefs.DMS_ERROR_CODE_ERROR + e.getMessage());
            }
            versions = new Vector<DocID>();
        for (int i = 0 ; i < versionsInfos.size() ; ++i) {
               versions.add(new FilesysDMSDocID((DocInfo)versionsInfos.get(i)));
        }
        return versions;
}
```

After it receives the vector of the document versions, it builds a docID for each element. Finally, it returns the list of the docIDs as a vector.

6.9 Implementing handler for Default Property

When the AutoVue Server sends a `GetProperties` request with a property that does not have a class for handling it, the framework runs the `GetPropDefault` class. This latter class must not return an exception if it does not handle the requested property. Instead, it should return empty value. The `GetPropDefault` class is not dedicated to a particular property and there is no property called `Default`, so when you register the web.xml file you must use `dms.getprops.Default` as the parameter name. Of course, you can give the class a different name from the default one. However, if you choose not to register the class, then you must name it `GetPropDefault`.

```
<init-param>
  <param-name>dms.getprops.Default</param-name>
    <param-value>com.cimmetry.vuelink.filesys.propactions.GetPropDefault</param-value>
</init-param>
```

Later we will discuss when to use individual classes for handling properties and when to use `GetPropDefault` class. Also we will discuss how you can avoid implementing the `GetPropDefault` by implementing a class for each request property.
For more information, refer to the source code of this class and run this class in debug mode for more information on its behavior.

```
public DMSProperty execute(final DMSContext    context,
                        final DMSSession    session,
                        final DMSQuery      query,
                        final DMSArgument[] args,
                        final Property property
                        ) throws VuelinkException {

    // Insert here the code that handles properties

    // CSI_Redirected Property
    if (DMSProperty.CSI_Redirected.equals(propName)) {
        retProp = new DMSProperty(DMSProperty.CSI_Redirected,
        new Boolean(false));
    // Number of versions of a document property
    // NB : this is not a CSI propety but a specific filesys property
    }else if ("VersionsNumber".equals(propName)) {
                    retProp = new DMSProperty("VersionsNumber",
                            attrs.get("VersionsNumber"));
     // If the property in not handled throw an exception
        } else {
            throw new VuelinkException(DMS_ERROR_CODE_UNKNOWN_ERROR,
                    "Unsupported property: " + propName);
        }
    }
    return retProp;
}
```

Figure 14: The code of the execute method of the GetPropDefault class

6.10 Implementing File Save Action

Users can create and modify markups and convert documents to other formats as TIFF and PDF. When they save these documents in the DMS repository by selecting the 'Save' or 'Save As' actions from AutoVue's File menu, the AutoVue Server sends an Action Save request. The response of this request is done through the `ActionSave` class. Refer to the JavaDMAPI.pdf manual for the actual format for Action Save response.

In saving Markups, there are two cases to handle. The first case is when the user tries to save a new Markup file. In this case, and as shown in the following excerpt of code, the Save action must return a valid docID of the newly created Markup. The second case is when the user tries to save an existing Markup file. In this case the Markup file keeps its old docID. For saving Markups the ActionSave class relies on the service of the `saveMarkup()` method of the Filesys DMS backend class.

When the user performs conversion of a document by selecting the Convert action from the File menu, AutoVue exhibits the same behavior as for saving Markups. But this time the `ActionSave` invokes the `saveRendition()` method of the Filesys DMS backend class.

In the Sample Integration for Filesys, the getDMSArgsProperties API is useful for both save methods. This API provides properties about the docID of the base document, the docID of

the Rendition or the Markup document if it exists, the Markup and Rendition types, and the Markups and Renditions files name. This information lets the FilsysDMS repository locate where the documents are saved, and is therefore very important.

```
package com.cimmetry.vuelink.filesys.propactions;

…

public class ActionSave implements DMSAction, DMSDefs{
•••
public Object execute(final DMSContext    context,
                      final DMSSession    session,
                      final DMSQuery      query,
                      final DMSArgument[] args
                      ) throws VuelinkException {
•••
// Get the file instance as DMSArgument object
final DMSArgument fileArg = args[0];
•••
// document dms properties
final Property[] props = query.getDMSArgsProperties();
•••
  // save document
  if (markType != null){
    newDocID = be.saveMarkup(beSession,baseID, outName, markType, fIn);
  }else if(rendType != null){
    newDocID = be.saveRendition(beSession,baseID, outName, rendType, fIn);
  }
•••
return newDocID;}
```

In the Sample Integration for Filesys, the `saveMarkup` method uploads the Markup file as an `InputStream` object and invokes the `saveMarkup` method from the FiesysDMS backend to save the in the repository. The parameters are: docID of the base document, the markup type, and the markup file as `InputStream.`

```
public DocID saveMarkup(DMSBackendSession session, DocID docID,
                        String filename, String markupType, InputStream fIn) throws
FileNotFoundException, IOException{
              FilesysDMSDocID fsDocID = (FilesysDMSDocID) docID;
              return new FilesysDMSDocID(m_filesysInfo.saveMarkup(fsDocID,
                                                    markupType, filename, fIn));
}
```

It returns the `docID` of the saved Markups if it fails it throws an exception.

In the Sample Integration for Filesys, the saveRendition methods uploads the Rendition file as an `InputStream` object and invokes the `saveRendition` method from the FiesysDMS backend to save the file in the repository. The parameters are: docID of the base document, the rendition type, and the rendition file as `InputStream.`

```
public DocID saveRendition(DMSBackendSession session, DocID docID, String filename,
        String rendType, InputStream fIn) throws FileNotFoundException, IOException{
                FilesysDMSDocID fsDocID = (FilesysDMSDocID) docID;
        return new FilesysDMSDocID(m_filesysInfo.saveRendition(fsDocID, rendType,
                                                                filename, fIn));
}
```

It returns the docID of the saved Markups. If it fails, it throws an exception.

The SaveAction class is registered in the web.xml file as indicated in the following code excerpt.

```
<init-param>
    <param-name>dms.action.Save</param-name>
    <param-value>com.cimmetry.vuelink.filesys.actions.ActionSave</param-value>
</init-param>
```

Refer to the source code of this class for more information. You can also run this class in debug, as shown in the following figure to help you learn more about the dynamic behavior of this class.

```
66          /* Sanity checks */
67          if (!"save".equalsIgnoreCase(query.getActionName())) {
68              throw new VuelinkException(DMS_ERROR_CODE_UNKNOWN_ERROR,
69                                         "Invalid action name within query: " +
70                                         query.getActionName());
```

6.11 Implementing File Delete Action

Users have the option of deleting Markups from within the AutoVue Applet. When they try to delete existing markups, AutoVue Server sends an Delete Action request. The response to this request is handled by the `ActionDelete` class. The document to be deleted is indicated by the `docID` parameter.

In the Sample Integration for Filesys, to be deleted effectively from DMS repository this class sends its request through the `deleteMarkup()` method of FilesysDMS backend class..

```
package com.cimmetry.vuelink.filesys.propactions;

/**   */

public class ActionDelete implements DMSAction, DMSDefs{
•••
public Object execute(final DMSContext    context,
                      final DMSSession    session,
                      final DMSQuery      query,
                      final DMSArgument[] args
                      ) throws VuelinkException {
•••
// delete markup document
if (! ((FilesysDMSBackend)context.getBackendAPI()).
        deleteMarkup(context.getBackendSession(session), docID)) {
       throw VuelinkException(DMS_ERROR_CODE_ERROR,
                              DMS_ERROR_MSG_DELETE);}
return null;}
```

In the Sample Integration for Filesys, the `deleteMarkup` method sends a request to the DMS repository to delete the markups identified by a 'docID' passed in the parameter. If the document is deleted, it returns true. Otherwise it returns false.

```
public boolean deleteMarkup(DMSBackendSession session, DocID docID) {
              FilesysDMSDocID fsDocID = (FilesysDMSDocID) docID;
              boolean deletedDoc = false;
              try{
                     deletedDoc = m_filesysInfo.deleteDocument(fsDocID);
              }catch(Exception e){
              m_logger.error(DMSDefs.DMS_ERROR_CODE_ERROR + e.getMessage());
       }
              return deletedDoc;
       }
```

The `ActionDelete` class is registered in the web.xml file as shown in the following excerpt of code.

```
<init-param>
    <param-name>dms.action.Delete</param-name>
    <param-value>com.cimmetry.vuelink.filesys.actions.ActionDelete</param-value>
</init-param>
```

For more information, examine the code and use the debugger to learn more about the behavior of this method.

6.12 Creating your Context

Each VueLink has a context that holds various environment settings that remains constant throughout the VueLink servlet lifetime. This context is initialized during the VueLink servlet initialization and is passed to actions every time VueLink handles a request.

The framework publishes the `com.cimmetry.vuelink.context.DMSContext` interface which describes a set behavior that a context handler must exhibit, which include:

- Initializes this `DMSContext` by fetching the appropriate information within the DMS servlet initialization parameters.
- Finds, registers, and locates the appropriate Backend API class for the current DMS servlet.
- Finds the backend session object corresponding to the `DMSSession`.
- Creates a new backend session if an existing session cannot be found.

The framework provides the `com.cimmetry.vuelink.context.GenericContext` class which is a default implementation of the *`DMSContext`* interface. You must provide your own implementation of the *`DMSContext`* interface only if the `GenericContext` does not satisfy your needs.

For each DMS servlet, the context action is registered during the initialization of the DMS servlet and loaded by the framework as follows:

1. It fetches the init parameters looking for whether a custom action context is provided.
2. It looks for `ActionContext` class in the same location as the DMS VueLink servlet.
3. It looks for `ActionContext` class in the same location as the DMS VueLink servlet.
4. It looks for `GenericContext` in the same location as the DMS Vuelink servlet.
5. It throws an exception if does not succeed to find a class to handle the context.

In the Filesys DMS application, we did not need to implement a custom class to handle the context because the `GenericContext` class does the work. But remember that we have refactored the framework code, and the `GenericContext` class is no longer in the same location as the `FilesysVuelink` servlet. The framework will fail to find an action context class and will throw an exception. We encountered this problem by deriving and registering a subclass, which does absolutely nothing, of the `GenericContext` class.

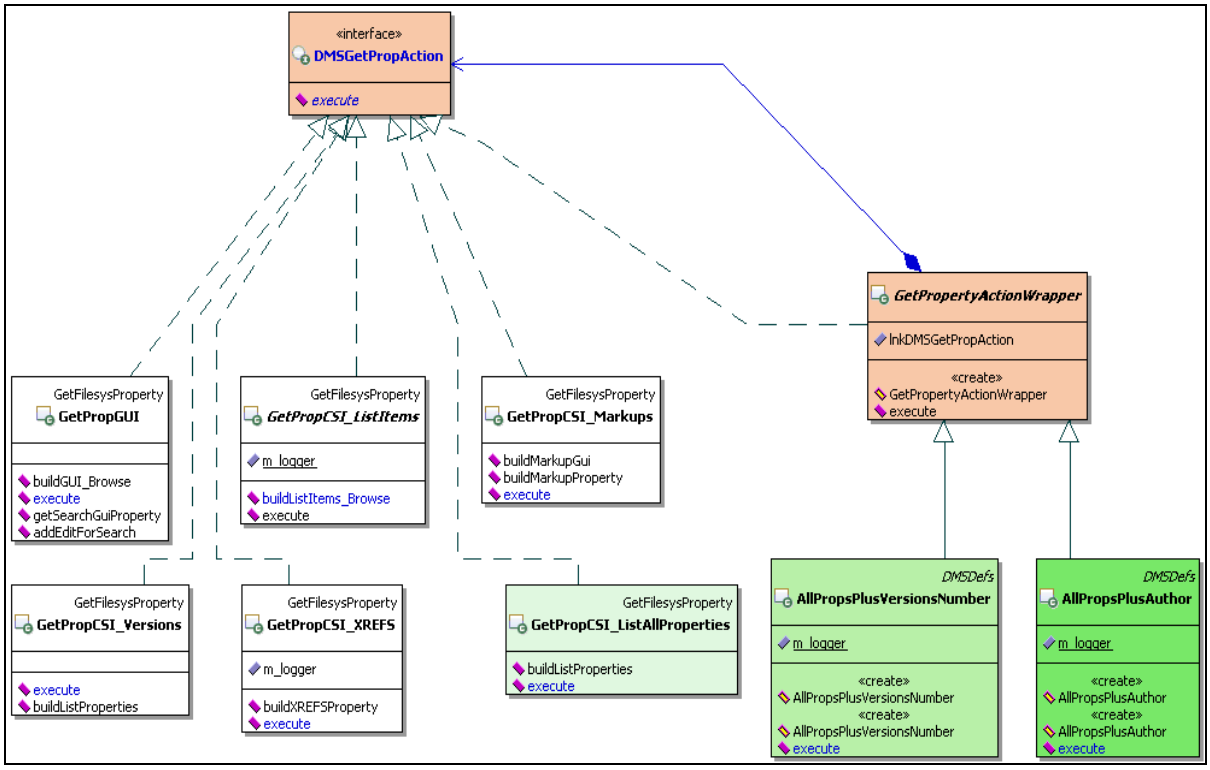
```
package com.cimmetry.vuelink.filesys.actions;

/**   */

public class ActionContext extends GenericContext {
…
…
}
```

```
<init-param>
  <param-name>dms.actions.Context</param-name>
    <param-value>com.cimmetry.vuelink.filesys.actions.ActioContext</param-value>
</init-param>
```

6.13 Overriding GetProp<CSI Property> classes

You may want to extend the response provided by a property class. For instance, you may add the number of all existing versions of a document to the ListAllProperties response. There are several ways to implement a mechanism that lets you extend the behavior of property classes. One mechanism you may consider is inheritance. A second one may be similar to the mechanism already implemented in the framework, such as simply implementing a new class that implements the `DMSGetPropAction` interface and registers it in the web.xml file.

The inheritance has two limitations: First limitation is that the new behaviors are added statically (i.e., at compilation time) The second is that for each new behavior, we must derive a new class and we know that the multiplication of the number of classes can be a maintenance nightmare. The second mechanism consists of replacing the old class by a new one which implements the new behaviors. A better solution is to add new behaviors to existing ones. This is simply because we do not need to rewrite existing code that has been tested and proven to be bug-free.



We designed an advanced integration mechanism that allows integrators and professional services to extend the handling of specific DMAPI messages without recompiling or rebuilding the entire integrations by just adding the overriding code. As illustrated in the

previous figure and the excerpt of the following code, we designed a class called `com.cimmetry.vuelink.propsaction.GerPropertyActionWrapper` which implements the `com.cimmetry.vuelink.propsaction.`*`DMSGetPropAction`* interface and has a variable that references any object that implements this interface. Note that the wrapper class implements the same interface as the classes it is going to wrap.

```
package com.cimmetry.vuelink.propsaction;

/**   */

public abstract class GetPropertyActionWrapper implements DMSGetPropAction {
        /**
         * {@link com.cimmetry.vuelink.core.DMSGetPropAction} object instance
         */
        protected DMSGetPropAction propertyAction ;

        /**
         * Constructs a decorator from the object to extend
         *
         * @param propAction object to extend
         */
        public GetPropertyActionWrapper(DMSGetPropAction propAction){
                this.propertyAction = propAction;
        }
}
```

To add a new behavior we just have to add a new class derived from the wrapper class. This mechanism allows third-party integrators to easily upgrade their solutions.

For example in the Sample Integration for Filesys, to add the number of versions of a document to the ListAllProperties class we just have to created a new `AllPropsPlusVersionsNumber` class that wraps the `GetPropCSI_ListAllProperties` and adds to it the number of versions of a document.

```
public class AllPropsPlusVersionsNumber extends GetPropertyActionWrapper
                                                implements DMSDefs  {
        /**
          * Wrapp the existing object
          */
        public AllPropsPlusVersionsNumber(){
                super(new GetPropCSI_ListAllProperties());
        }
        public DMSProperty execute(DMSContext context, DMSSession session,
                                DMSQuery query, DMSArgument[] args, Property property)
                                                        throws VuelinkException {
                // add the new behavior
                …
```

Finally, we have to register this class as indicated in the following excerpt of code. Remember that the wrapper is still using the services of the object it wraps.

```
 <init-param>
   <param-name>dms.getprops.CSI_ListAllProperties</param-name>
   <param-value>
      com.cimmetry.vuelink.filesys.propactions.GetPropCSI_AllPropsPlusVersionsNumber
   </param-value>
 </init-param>
```

The major advantage of this mechanism is its capability to dynamically compose wrapper classes. For example, you may add a new behavior to the same class by adding the document author property you just have to follow the same steps previously explained. But in this case, wrap a wrapper class as shown in the following excerpt of code.

```
public class AllPropsPlusAuthor extends GetPropertyActionWrapper implements
              DMSDefs {

       public AllPropsPlusAuthor(){ sper(new AllPropsPlusVersionsNumber(new
                                                    _ListAllProperties()));
                                                    }
```

You can also decide that a document has two authors. In that case, you need to compose the new behavior as indicated in the following line of code without adding any line of code.

```
new AllPropsPlusAuthor(new AllPropsPlusAuthor(new AllPropsPlusVersionsNumber(new
CSI__GetListAllProperties())))
```

7. APPENDIX – SAMPLE INTEGRATION

The Sample Integration for Filesys DMS) included in the AutoVue Integration SDK acts as an example and a starting point for new integration implementation and for getting familiar with the integration framework. The following figure shows the use case diagrams of possible actions available from within the AutoVue interface. A user logs into FilesysDMS through a Web browser and selects a file to view in AutoVue. Once the file is loaded in AutoVue, the user can perform other actions such as markup, conversion, compare, search, browse, and so on.

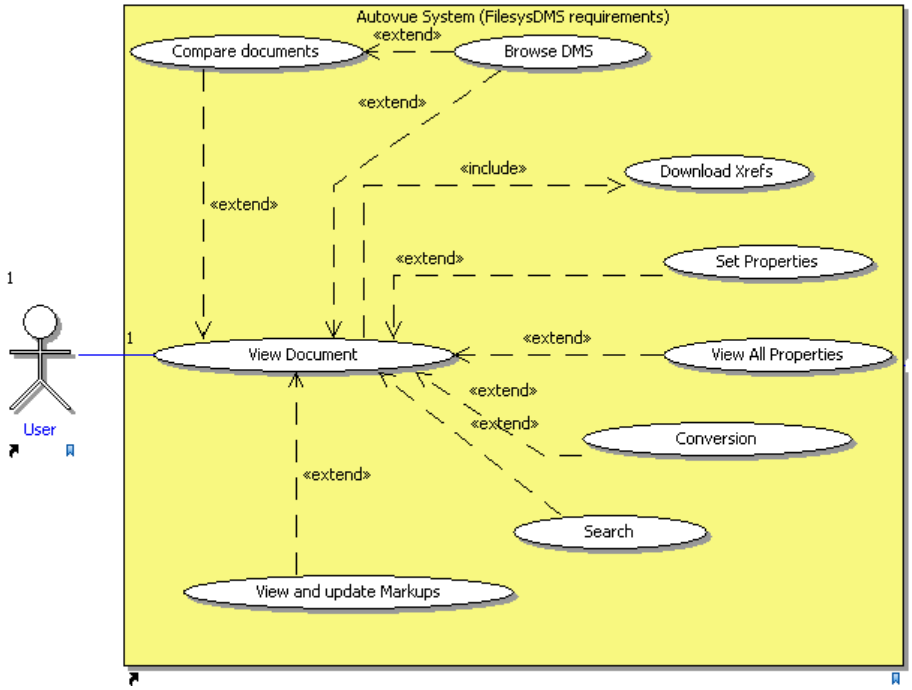


Figure 15: Use cases diagram for the FilesysDMS sample

As illustrated in the following figure, we have designed the Vuelink servlet class and three packages for the FilsysDMS integration, as follows:

1. The first package is called `com.cimmetry.vuelink.filesys.actions` and contains all action classes. The common characteristic of theses classes is that they all implement the DMSAction interface.
2. The second package is called `propactions` and contains a set of classes that all implement the DMSGetPropAction interface.
3. The third package is called `backend` and has three classes: the FilesysDMSBackend class that implements the DMSBackend interface, the FilesysDMS class which is the backend API that talks to FilesysDMS repository, and the FilesysDocID class which implements the DocID interface and defines the document ID.

Refer to chapters 5 and 6 for more information on the design of the Sample Integration for Filesys DMS.

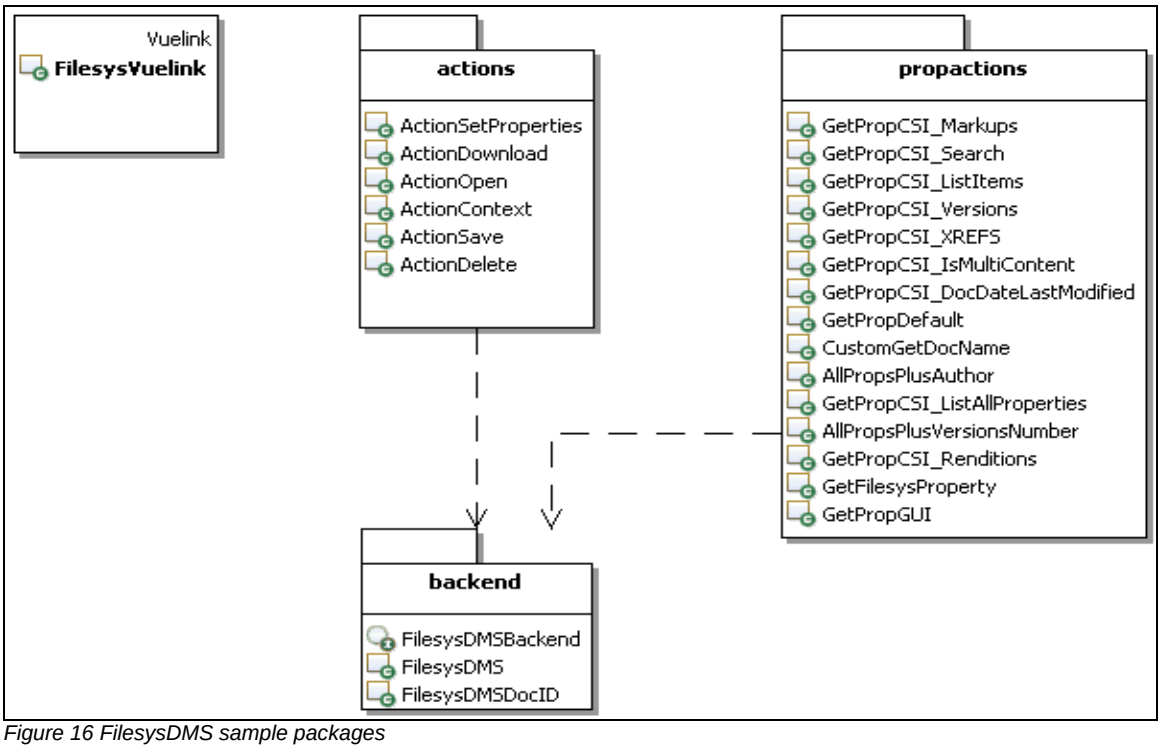


Figure 16 FilesysDMS sample packages

The data used by the sample integration is based on a very simple file system that has a simple data structure to store and retrieve files (the data structure is described in the next section of this Appendix). The file system includes three packages, as shown in figure 4.

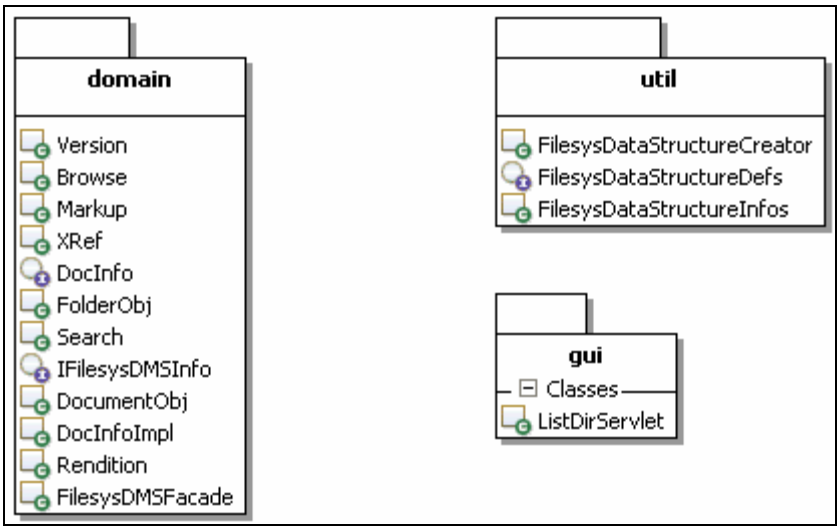


Figure 4: FilesysDMS Data repository manager packages

1. The first package is called `domain` and contains all the classes dedicated to managing the data repository. When we implemented our actions to retrieve and store files in the repository, we did it through the `com.cimmetry.vuelink.filesys.dms.domain.IFilesysDMSInfo` interface. This interface is our plugin point to the FilesysDMS repository manager.
2. The second package is called `util` and allows us to add new data to the repository. The instructions on how to add new data are described in the **User Guide**.
3. The last package is called `gui` and is essentially a servlet which allows us navigate the sample files through a dynamic HTML page.

7.1 DMSActions

A `DMSAction` has only one method to implement: `execute`. It takes four parameters:

- **`DMSContext`:** Represents the context of execution of a *`DMSAction`* and holds various environment settings.
- **`DMSSession`:** Represents the session of execution of a *`DMSAction`* for an arbitrary set of DMS queries.
- **`DMSQuery`**: Represents a query that a `DMSAction` must handle and holds parameters such as the original document URL (FILENAME param passed in the AutoVue applet page), the document ID, the collaboration session ID, the collaboration session data, the Authorization and a set of Properties.
- **`DMSArgument`**: Represents list of objects used to hold special arguments specific to a given DMS action type.

The `execute` method returns an object instance (the type of the instance depends on the DMS action but it is generally either null, a `DocID,` a File or a Property list). To report failures, execute can throw a `VuelinkException` containing the error code and error message (defined in the `DMSDefs` public interface) that the Vuelink servlet uses to build the `<ERROR>` HTTP response.

One important goal of the AutoVue Integration SDK is to make the integration open to extensions and modifications. We achieved that by registering the action classes in the `web.xml` file in init-parameters. The Vuelink servlet checks the init-parameters and registers the actions. Each action parameter name has the prefix `dms.action` followed by the name of the action as `dms.action.open` (e.g., for Open Action). The value parameter specifies the action name and its location (e.g., `com.cimmetry.vuelink.filesys.actions.ActionOpen`). This mechanism allows us to drop any obsolete class and replace it by a new one simply by updating the init-parameter.

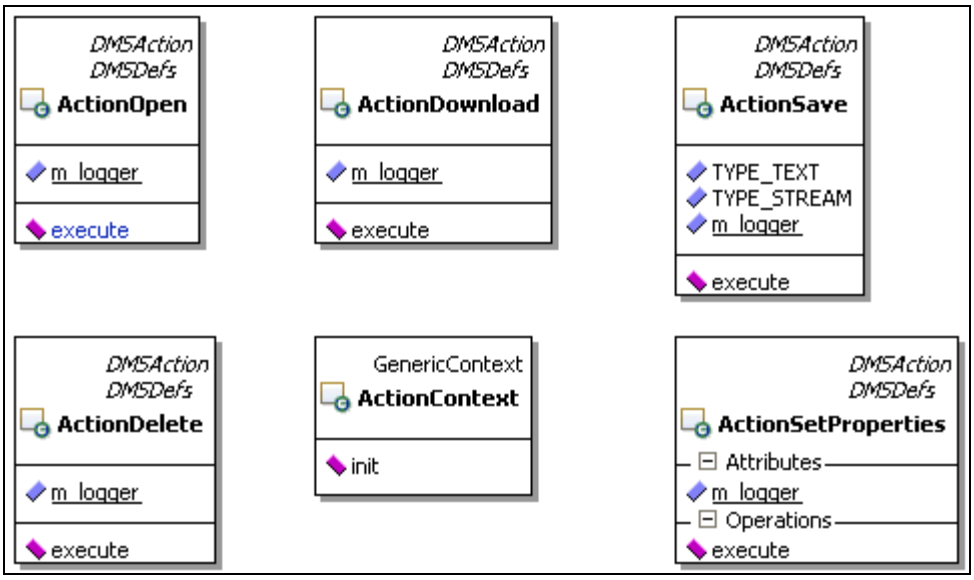


Figure 17: Action classes

In the Filesys DMS, we designed the `com.cimmetry.Vuelink.filesys.actions` package which implements all the needed actions. In this section we discuss the `Open`, `Download`, `Save`, and `Delete` actions. The `SetProperties`, `GetProperties` and `ActionContext` are discussed in the following sections.

Each individual class must be registered in the `web.xml` (web descriptor for your J2EE web application) file init parameters. The name of the parameter has the format dms.getprops.<property name> (e.g., dms.getprops.CSI_Markups). The value of the parameter contains the full qualification of the class and has the format "com.<yourCompany>.<package>.<class name>". You can choose the class name you want. Also, if you prefer, you can choose the default name proposed by framework "GetProp<prop name>" (e.g., `GetPropCSI_Markups`).

This makes the code easier to maintain and, more importantly, makes customization a lot easier. If changes to markup handling are required, the `GetPropCSI_Markups` class can be re-implemented without affecting the handling of any of the other properties. This will make the customization easier in the first place, and the customizations will be easier to update when the framework is updated. This will also allow the easy mix-and-match of functionality. For example, if a customized markup handler is done for Customer A, and later Customer B needs similar functionality, the class written for A can be dropped into B's install without impacting any other customizations done for B.

For the Filesys DMS we designed the `com.cimmetry.vuelink.filesys.propactions` which contains the following property action classes:

- **`CustomGetDocName`**: Handles the `CSI_DocName` property and returns the document name.
- **`GetPropCSI_DocDateLastModified`**: Handles the `CSI_DocDateLastModified` property and returns the date of the last modification of a document.

- **GetPropCSI_IsMultiContent**: Handles the CSI_IsMultiContent property.
- **GetPropCSI_ListAllProperties**: Handles the CSI_ListAllProperties property and returns an array DMS properties.
- **GetPropCSI_XREFS**: Handles the CSI_XREFS property and return an array of properties concerning the xrefs documents.
- **GetPropCSI_Markups**: Handles the CSI_Markups property and returns an array of properties concerning markups documents.
- **GetPropCSI_Renditions**: Handles the CSI_Renditions property and returns an array of properties concerning renditions documents.
- **GetPropCSI_Versions**: Handles the CSI_Versions property and returns an array of document versions properties.
- **GetPropGUI**: Handles the GUI property and returns an array of properties for building the browse GUI or the search GUI.
- **GetPropCSI_ListItems**: Handles the CSI_ListItems and returns an array of items to be displayed in the browse GUI.
- **GetPropCSI_Search**: Handles the CSI_Search property and returns an array of properties of documents that match the criteria search.
- **GetPropDefault**: Handles the properties that do not have dedicated individual classes.
- **GetFilesysProperty**: Returns additional document attributes (attributes that do not match any of the CSI properties). That class serves as support to some of the previously-cited classes.

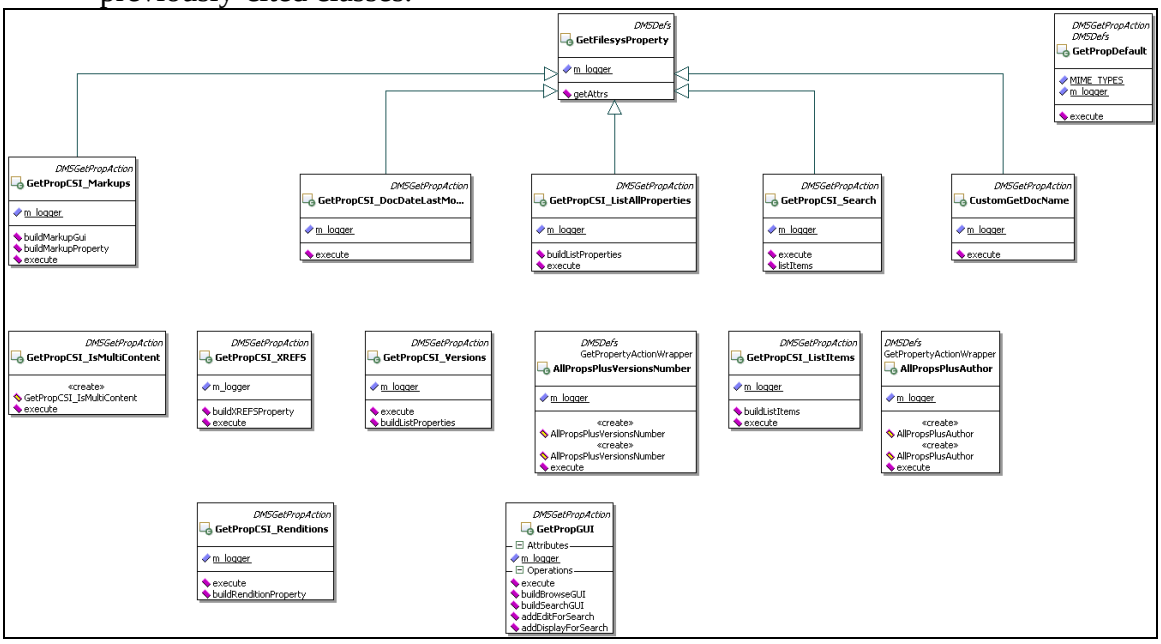


Figure 18: Property actions classes

7.2 Backend API

Note: Implementing the DMSBackend interface is optional. It is intended as an entry point to your custom code for handling communication with your DMS/PLM system. You can think of the Backend class as a wrapper around your DMS API.

The backend API allows the integration interface to properly talk with the DMS. This API is intended to gather all the custom code for handling communication with the DMS. Our backend class that implemented the `DMSBackend` interface also implemented the `connect` method which allows AutoVue to reuse existing user sessions with the DMS.

The framework locates the object that implements the backend API for an integration inside the `com.cimmetry.vuelink.context.DMSContext` object. During the initialization of the Vuelink servlet, a `DMSContext` object is created which in its turn initializes and registers the backend object. This allows you to get a reference to the backend API from a `DMSContext` object. This is always possible since all the `DMSAction` objects and `com.cimmetry.vuelink.propsaction.DMSGetPropAction` objects hold a `DMSContext` object. If custom registration, saving and loading of the backend API object are needed, you must derive the `GenericContext` class and implement the new overriding methods.

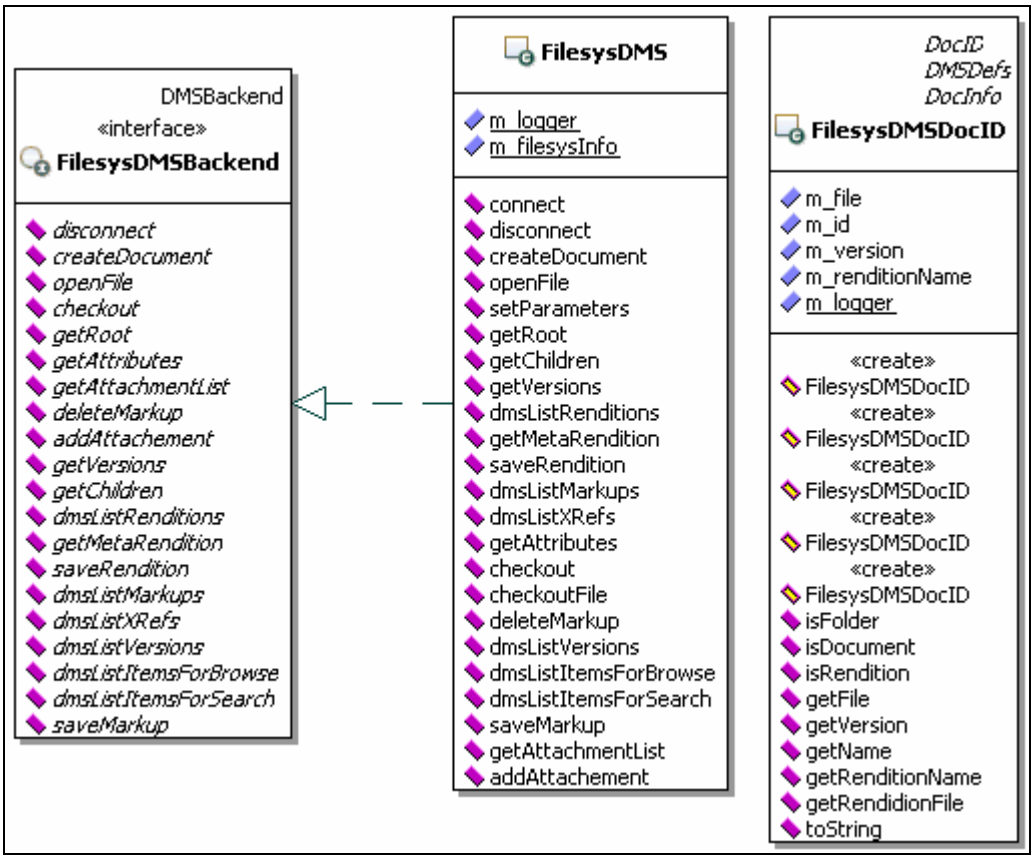


Figure 19 Backend classes

First the framework fetches the init parameters for `dms.backend`, the name of the init parameter, and then instantiates the class specified in the value parameter. If it fails, then it looks for the `DMSBackendImp` class as the default name in the current package (That is, in the same location where your DMS servlet is located).

In the Filesys DMS application, we registered the backend API as shown in the following excerpt of code.

```
<init-param>
    <param-name>dms.backend</param-name>
    <param-value>com.cimmetry.vuelink.filesys.backend.FilesysDMS</param-value>
</init-param>
```

The following excerpt of the code shows how to get an instance of the plugin point to Filesys DMS repository.

```
/** Instance of FilesysDMS object (singleton) responsible for communicating and providing
Vuelink with the required information */

private static final IFilesysDMSInfo m_filesysInfo = FilesysDMSFacade.getFilesysInstance();
```

7.3 Filesys DMS Repository Structure

The data used by this sample is based on a file system. This system has a simple structure to store and retrieve files. This structure consists of folders and document objects. Folder objects represent directories and document objects represent files. Folder objects can contain a list of document objects and a list of folder objects (the subfolders).

The access to the root of the FilesysDMS system structure is done through a given specific path. Inside the FilesysDMS structure we categorize the documents to allow flexible and easy document management. Each category is simply represented by an access path. Thus, the FilesysDMS system structure contains all the categories of documents to manage. For example, in figure 5, filesysDatabase is the root directory which contains 2 documents categories "2DRepository" and "3DRepository".

Inside a category one finds several folders, a folder per document. Each directory has the same name as the base document that it represents, and contains all the versions of this document. Each version is represented by a folder which has the same name as its base document concatenated to the number of the version enclosed between parentheses. For example, in figure 13, the category "*2DRepository*" contains three folders which correspond to the base documents *bike.dgn*, *main.dgn* and *myacad12.dgn*. The folder *myacad12.dgn* contains three versions of the base document and *myacad12.dgn(3)* represents its third version.

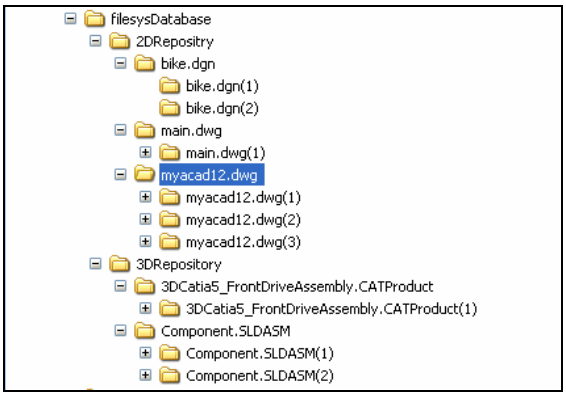


Figure 20: FilesysDMS data structure

Each version folder contains all related information (xrefs, markups renditions, and so on). For example, as illustrated in figure 14, under the folder representing version 2 of the document *myacade12.dgn*, there is the base document and the folders which contain the external references (i.e., in the case of a composite document), the markups, and renditions. The **xrefs** folder contains all files which constitute external references. The **markups** folder contains three subfolders which correspond to the different types of markups supported by AutoVue: normal, master and consolidated (see figure 15). Each of these subfolders contain all corresponding markups. Finally, the **renditions** folder contains all conversions supported by AutoVue and the metafiles. For example, in figure 15, the tiff folder contains the TIFF rendition. Note that the rendition subfolders have the same names as the rendition types.

Figure 21 : A document version structure

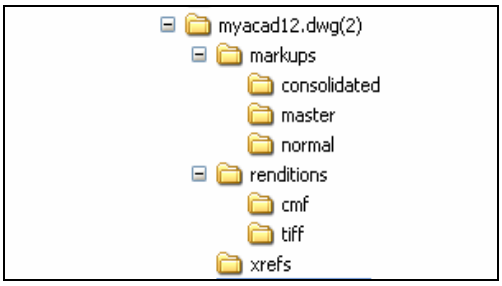


Figure 22: Content of version subfolders

This simple structure represents a good starting point when building your own integration based on the integration framework. Managing documents in this structure requires creating folders and copying files.

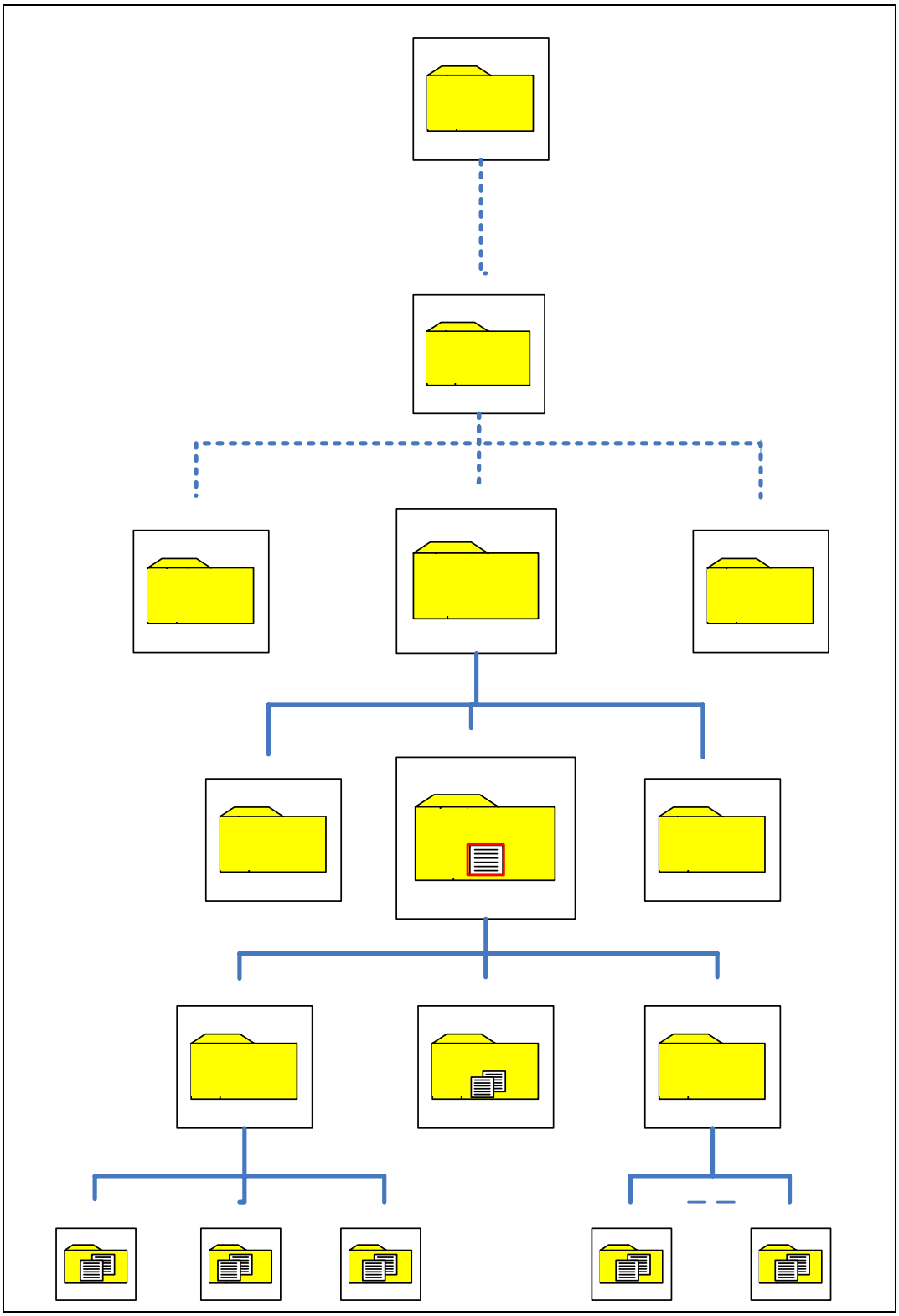


Figure 23: Filesys DMS repository structure

7.4 Sample Integration for Filesys DMS Use Cases

The implementation of the Sample Integration for Filesys DMS involved the implementation of the following functionalities:

- Implementing *DMSAction* interface for open/download/save/getproperties, etc.
- Implementing the backend interface for communicating with the Filesys DMS system.

The requirements for the *DMSAction* interface are presented in the "DMAPI Use Cases" section and those for the backend interface are presented in the "Backend Use Cases" section.

7.4.1 DMAPI use cases

Following six classes implement the DMSAction interface:

- ActionOpen
- ActionDownload
- ActionGetProperties
- ActionSetProperties
- ActionSave
- ActionDelete

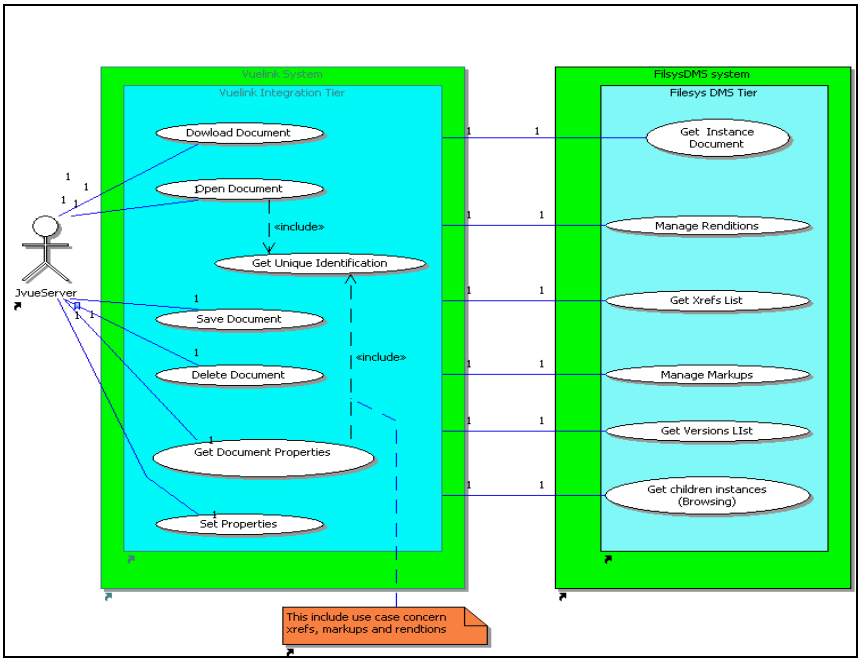


Figure 24 DMSAction interface for FilesysDMS and functionalities provided by the Filesys DMS

7.4.1.1 ActionOpen

The exchanged documents between the sample integration and the AutoVue Server must have unique identifiers. This is why sample integration must build a unique docid for each document sent to the AutoVue Server.

Use case: Get unique document identification
Description: The get unique identification use case builds a unique identification for each different document (i.e., base document and Xrefs documents, etc) sent to AutoVue Server.
Precondition: sample integration receives an open document request from AutoVue *Server*
Deployment constraints: none
Normal flow of events:

1. Sample integration builds a unique identification for each different document returned by *FilesysDMS* and sends it to AutoVue Server.

Activity diagram: none
Nonfunctional requirements: none
Open issues: none

7.4.1.2 ActionDownload

Sample integration processes the download request when *FilesysDMS* user wants to view a file from *filesysDMS* repository.

Use case: Download document
Description: The download document use case communicates to *FilesysDMS* the document to download.
Precondition: *Vuelink* receives a download document request from *AutoVue Server*
Deployment constraints: none
Normal flow of events:

1. *Sample integration* sends a download request to *FilesysDMS* system specified by a unique identifier
2. *Sample integration* returns to *AutoVue Server* the downloaded document

Exception flow of events:

1. *Sample integration* receives the message indicating that the document can not be downloaded
2. *Sample integration* sends the message to *AutoVue Server*
3. Add the exception to a log

Activity diagram: none
Nonfunctional requirements: none
Open issues: none

7.4.1.3 ActionDelete

Sample integration processes the delete request when *FilesysDMS* user wants to delete markups. The use case below describes this functionality.

Use case: Delete document
Description: The delete document use case communicates to *FilesysDMS* the document to delete.
Precondition: *Sample integration* receives a delete document request from *AutoVue Server*
Deployment constraints: Only markups documents can be deleted.
Normal flow of events:

1. *Sample integration* send a request to *FilesysDMS* system to delete the document specified by a unique identifier

Exception flow of events:

1. *Sample integration* receives a message indicating that the document can not be deleted
2. *Sample integration* sends the message to *AutoVue Server*
3. Add the exception to a log

Activity diagram: none
Nonfunctional requirements: none
Open issues: none

7.4.1.4 ActionSave

Sample integration processes the save request when *FilesysDMS* user wants to save markups or creates a rendition. When user saves document, *AutoVue Server* sends a request to integration servlet which relays this request to *FilesysDMS* to save the document. The following use case describes this functionality.

Use case: Save document
Description: The save document use case communicates to *FilesysDMS* system the document to save.
Precondition: *Sample integration* receives a save document request from *AutoVue Server*
Deployment constraints: Only markups and renditions (including metafiles) can be saved
Normal flow of events:

1. *Sample integration* sends a request to *FilesysDMS* system

asking to save a document specified by a unique identifier

Exception flow of events:

1. Sample integration receives a message indicating that the document can not be saved

2. Add the exception to a log

Activity diagram: none

Nonfunctional requirements: none

Open issues: none

7.4.1.5 ActionGetProperties

Sample integration processes the get properties request when *FilesysDMS* user wants to view a file. In this case, the *AutoVue Server* sends several requests to Sample integration asking for information about markups, XRefs, renditions, document properties, etc. The use case below describes this functionalities.

Use Case: Get properties

Description: The get properties use case takes in charge multiple requests of *Sample integration*. The requests concern a set of predefined properties that *Sample integration* must return to *AutoVue Server*. These requests are about Xrefs, markups, renditions, GUIs and other information concerning the base document (e.g., name, size, etc.)

Precondition: *AutoVue Server* sends to *Sample integration* request about:

1. Base document properties:
    a. Unique identifier
    b. Last modification date
    c. Size
    d. Name
    e. Author
    f. Type document (i.e., folder or file)
    g. Multi content document
2. Xrefs properties
    a. Documents unique identifiers of the external references in case of a composite document
3. Markups Properties
    a. Documents unique identifiers and types (normal, master, consolidated) of markups
    b. Markups for all revisions
4. Renditions properties
    a. Unique document identifier when returning a metafile
    b. A converted document and its type (i.e., type rendition)

5. Versions properties
    a. Document Identifier, name, size and version number of the document
6. GetAllProperties property action: a set of properties that characterize a base document (i.e., name, size and last modification date, etc.)
7. GUI properties
    a. The properties that composes the browsing GUI and the search GUI respectively: (1) name, type (folder or file), size and last modification date and (2) document name and extension type.
    b. The property that allow browse functionality
    c. The property that allows search functionality
8. Search result: get documents that match the search criteria
9. Browse result: the content of the root repository folder and the content of the expanded folder until reaching the base document

Deployment constraints: none.
Normal flow of events:

The flow depends on the request of *AutoVue Server*. For each one of the above property request *Sample integration* must provide a response.

Exception flow of events:

a. *Sample integration* is unable to process the request
b. Add the exception to a log

Activity diagram: none
Nonfunctional requirements: none
Open issues: none

7.4.1.6 ActionSetProperties

Sample integration processes the set properties request when *FilesysDMS* user wants to print a file. In this case, *AutoVue Server* sends notification messages when each printed page and when whole document is done printing.

Use Case: Set properties
Description: The set properties use case sends notification messages to *Sample integration*.
Precondition: *AutoVue Server* sends to *Sample integration* notifications about printing.
Deployment constraints: none.
Normal flow of events:
Exception flow of events:
Activity diagram: none

Nonfunctional requirements: none
Open issues: CSI_Notifications problem in the JvueServer request is not specified according to the DMAPI XML document.

7.4.2 Backend use cases

To provide responses to the AutoVue Server, integration servlet interacts with FilesysDMS which must provide integration servlet with appropriate information.

7.4.2.1 Get Document Instance

To view a document, *FilesysDMS* must be able to return an instance of the document to *Sample integration*. The use case below describes this functionality.

Use Case: Get document instance
Description: The get document instance use case returns the file instance of a document.
Precondition: *Sample integration* sends to *FilesysDMS* a get document request
Deployment constraints: none
Normal flow of events:
1. *FilesysDMS* finds the document.
2. *FilesysDMS* returns the document to *Sample integration*

Exception flow of events:
1. *FilesysDMS* is unable to find the document
2. Add the exception to a log.

Activity diagram: none
Nonfunctional requirements: none
Open issues: none

7.4.2.2 Manage Renditions

FilesysDMS must be able to manage conversion operations done by the user. It must be able to save a converted documents and metafiles. This functionality is described by the manage renditions use case.

Use case: Manage Renditions
Description: The manage renditions use case manages all operations concerning renditions (i.e., (1) save conversions and (2) save and return metafiles).
Precondition: *Sample integration* sends to *FilesysDMS* one of

the following renditions requests:

1. Get metafile instance or
2. Save rendition instance (i.e., converted file or metafile)

Deployment constraints: none
Normal flow of events:

1. *FilesysDMS* finds and returns the metafile document.

Alternate flow of events:

1. FilesysDMS saves the rendition document (metafile or converted file) .

Exception flow of events:

1. FilesysDMS is unable to find the metafile document.
2. FilesysDMS is unable to save the rendition document.
3. Add the exceptions to a log.

Activity diagram: none
Nonfunctional requirements: none
Open issues: none

7.4.2.3 Get Xrefs List

In the case of composite document, *FilesysDMS* must provide *Sample integration* with the list of its external references. The use case below describes this functionality.

Use case: Get Xrefs list
Description: The get xrefs use case returns a list of external references of a composite document.
Precondition: *Sample integration* sends a request to *FilesysDMS* asking for XRefs list documents.
Deployment constraints: none
Normal flow of events:

1. *FilesysDMS* returns the list of XRefs documents.

Exception flow of events:

1. *FilesysDMS* is unable to find the XRefs.
2. Add the exception to a log.

Activity diagram: none
Nonfunctional requirements: none
Open issues: none

7.4.2.4 Manage markups

FilesysDMS must be able to provide responses all the requests about markups (i.e., return markups list of a document, return markups list of all revisions document, save and delete markups). All these functionalities are described in the following use case.

Use case: Manage markups
Description: The manage markups use case manages all the operations concerning markups.
Precondition: *Sample integration* sends to *FilesysDMS* one of the following requests:

- Get list of markups
- Get list of markups for all revisions
- Save a markup
- Delete a markup

Deployment constraints: none
Normal flow of events:

1. *FilesysDMS* returns the list of markups.

Alternate flow of events:

1. *FilesysDMS* returns the list of markups for all revisions

Alternate flow of events:

1. *FilesysDMS* saves a markup

Alternate flow of events:

1. *FilesysDMS* deletes a markup

Exception flow of events:

1. *FilesysDMS is unable* to build the list of markups.
2. *FilesysDMS* is unable to build the list of markups of all revisions.
3. *FilesysDMS* is unable to save markup.
4. *FilesysDMS* is unable to delete markup.
5. Add the exceptions to a log.

Activity diagram: none
Nonfunctional requirements: none
Open issues: none

7.4.2.5 Get versions list

FilesysDMS must be able to return all the versions of a document when a user needs them to perform a comparison operation. The use case below describes this functionality.

Use case: Get versions list
Description: The get versions list use case returns the list of different versions of a document.
Precondition: *Sample integration* sends a request to *FilesysDMS* asking for the list of versions:
Deployment constraints: none
Normal flow of events:

1. *FilesysDMS* returns a list of items representing the different versions of the base document.

Exception flow of events:

1. *FilesysDMS* is unable to return the list of versions.
2. Add the exception to a log.

Activity diagram: none
Nonfunctional requirements: none
Open issues: none

7.4.2.6 Get children instances

The user must be able to browse the *FilesysDMS* data structure by expanding folders. This is why it must provide *Sample integration* by the children documents of each expanded folder. The use case Get children instances describes this functionality.

Use case: Get children instances
Description: The get children instances returns a list of items contained in a folder. The user browses the *FilesysDMS* database structure by expanding folders.
Precondition: *Sample integration* sends a request to *FilesysDMS* asking for the list of items contained in the selected folder.
Normal flow of events:

1. Get List of items contained in the specified folder.

Deployment constraints: none
Normal flow of events:

1. *FilesysDMS* returns the list items contained in a specified folder.

Exception flow of events:

1. *FilesysDMS* is unable to return the list of items.

2. Add the exceptions to a log.

Activity diagram: none
Nonfunctional requirements: none
Open issues: none

8. FEEDBACK

Cimmetry Systems Corp. products are designed according to your needs. We would appreciate your feedback, comments or suggestions. Contact us by fax, e-mail or telephone. There is a feedback button on our Web page that activates an easy-to-use feedback form. Let us know what you think.

General Inquiries:

Telephone:	+1 514-735-3219
Fax:	(514) 735-6440
E-mail:	info@cimmetry.com
Web Site:	http://www.cimmetry.com

Sales Inquiries:

Telephone:	+1 514-735-3219 or 1-800-361-1904
Fax:	(514) 735-6440
E-mail:	sales@cimmetry.com

Customer Support:

Telephone:	+1 514-735-9941
Fax:	(514) 735-6440
E-mail:	sales@cimmetry.com