Oracle AutoVue 20.2

AutoVue API Programmer's Guide

March 2012

Contents

1. Preface

The *AutoVue API Programmer's G*uide provides an overview of the concepts of the AutoVue API and its fundamental packages and classes.
For the most up-to-date version of this document, go to the AutoVue Documentation Web site on the Oracle Technology Network at http://www.oracle.com/technetwork/documentation/autovue-091442.html.

Audience

This document is intended for Oracle partners and third-party developers (such as integrators) who want to implement their own integration with AutoVue. Note that these developers are expected to have a good understanding of JAVA programming. This guide serves as a good starting point for developers and professional services to become more familiar with the AutoVue API.

Related Documents

For more information, see the following documents:

- *VueBean Javadocs*
- *Oracle AutoVue Installation and Configuration Guide*
- *AutoVue Planning Guide*
- *AutoVue Integration SDK Overview*
- *Oracle AutoVue Web Services Overview*

2. Introduction

The AutoVue Application Programming Interface (API)[1] is a Java-based toolset that provides tools to modify the functionality of Oracle's AutoVue client[2], and allows you to create your own customized Java applets/applications based on AutoVue API components.

Prior to developing your API integration, you should review the documentation for AutoVue API, AutoVue Web Services and AutoVue Integration SDK to find the integration tool that best fits your needs.

AutoVue API	This is the API described in this document.
AutoVue Web Services	You can integrate AutoVue's capabilities into your application regardless of platforms or programming languages. For more information, refer to the *AutoVue Web Services Overview*.
AutoVue Integration SDK	The AutoVue Integration SDK is a Java-based implementation of the Document Management Application Programming Interface (DMAPI) specifications published by Oracle. For more information, refer to the *AutoVue Integration SDK Overview*.

This document provides an overview of the concepts of the AutoVue API and its fundamental packages and classes. Additionally, basic and advanced applications of the AutoVue API are provided along with their source code. For detailed information on the packages and classes included in the AutoVue API, refer to the *VueBean Javadocs*.

[1] In previous releases of AutoVue, this API was referred to as the VueBean API.

[2] Throughout this document, the term "AutoVue client" is used interchangeably with term "JVue".

3. System Requirements

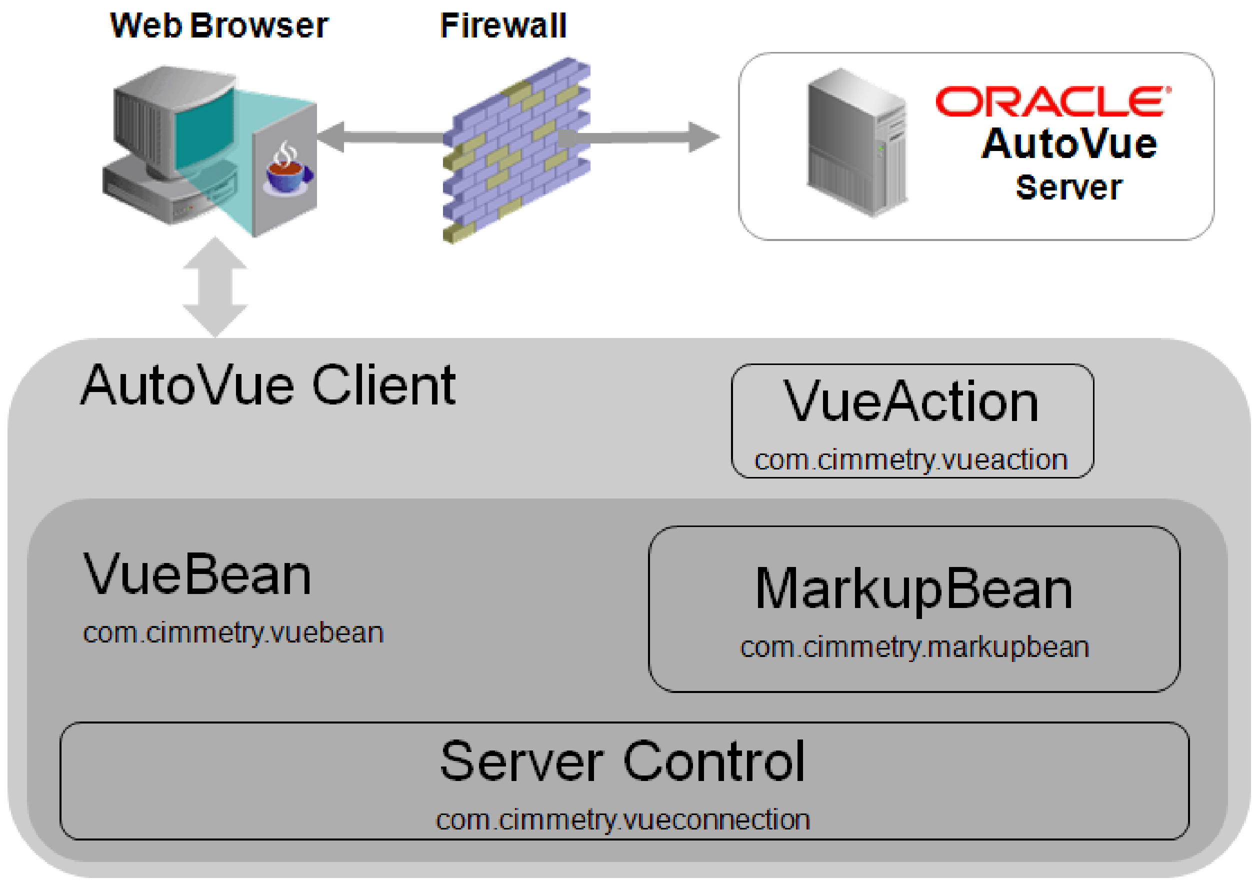
For a complete list of system requirements specific to your platform, refer to the *Oracle AutoVue Installation and Configuration Guide.*

4. Architecture of an AutoVue API Solution

The AutoVue API is an umbrella term for the APIs that the AutoVue client is built upon, with the VueBean API being the core component of the architecture. The client can be a Java application, a Java applet, or a Java servlet. The AutoVue client that ships with AutoVue Client/Server Deployment is an example of an applet-based AutoVue client. As seen in the following diagram, the AutoVue client is layered on top of the VueBean.

Note: It is possible to build a solution based on the JVue class (`com.cimmetry.jvue`) or based directly on the VueBean class (`com.cimmetry.vuebean`). If you build a solution based on the JVue class, then you are building from a class that already extends Applet, and you can take advantage of the functionality and the graphical user interface (GUI) that Oracle has built into the JVue layer. If you build a solution based on the VueBean class, then you must implement your own GUI.

There are a number of packages included in the AutoVue API. The following figure shows the most commonly used components.



VueAction: This component can add graphical user interface (GUI) elements to different contexts (such as menu bar, toolbar, status bar, and so on). For example, when a menu option is selected in the GUI, a VueAction is triggered. For more information, refer to VueAction Package.

VueBean: This component manages the representation of a file including the resources upon which the file depends. For more information, refer to VueBean Package.

MarkupBean: This component handles markup functionality. For more information, refer to MarkupBean Package.

Server Control: This component handles the communication with the AutoVue server and the session book keeping. For more information, refer to Server Control.

AutoVue API Design Options

With the AutoVue API you have three design options: modify the functionality of the client that is shipped with Oracle AutoVue, build your own customized application/applet, or implement pre-existing code from Oracle's AutoVue client to build your own customized client. It is recommended to review each option prior to developing your project.

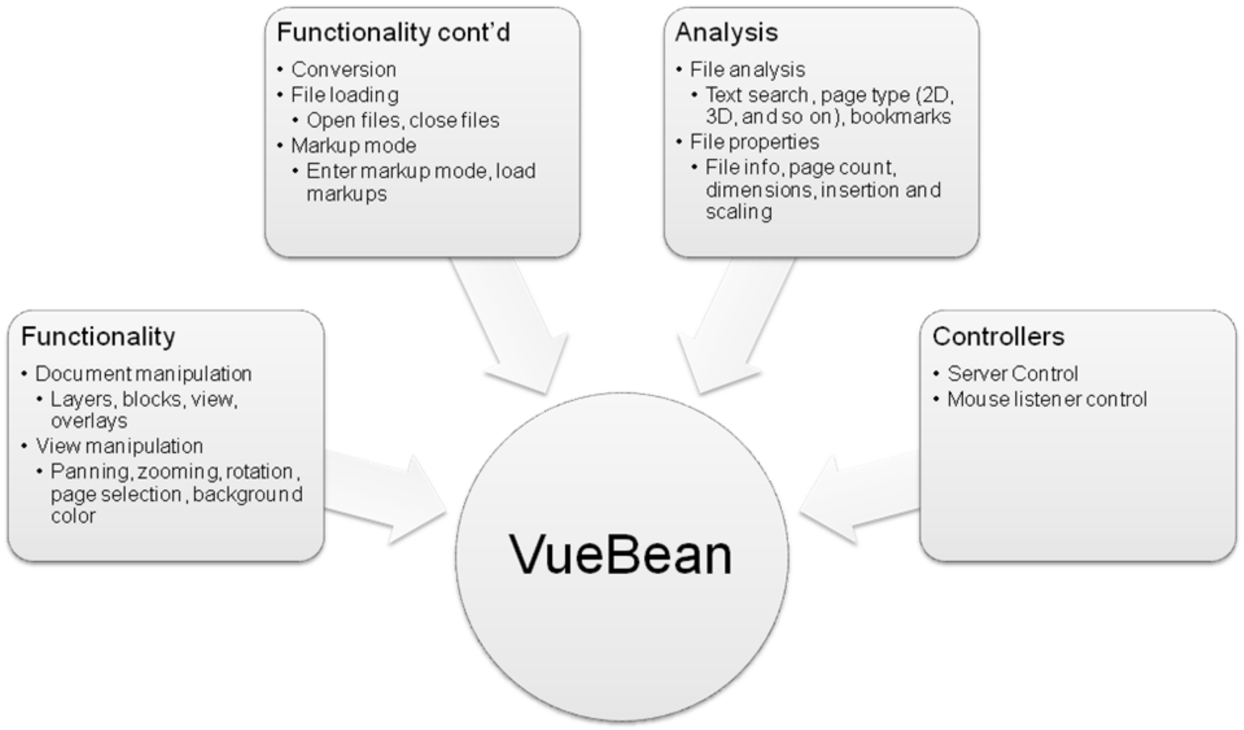
Design Option	Description
Adding Custom Actions to Oracle's AutoVue Client	This option is used to customize the existing AutoVue client's menus and toolbars. For an example, refer to Adding Customized Menu Items to the AutoVue Client.
Building a Custom Application/Applet	This option allows you to build an application/applet that makes calls to the VueBean package. You can leverage our viewing and markup technology while maintaining complete control of the behavior of the application/applet. For an example, refer to Building an AutoVue API Application.
Modifying the Behavior of Oracle's AutoVue Client Through JavaScript	This option is used to build additional menu and toolbars outside of the AutoVue client's interface. You can design a standalone application or a Java applet in a Web page. For an example, refer to Implementing Functions from AutoVue in a Second Applet.

5. AutoVue API Packages

The following sections provide an overview of common classes and interfaces that are used to create a solution based on the AutoVue API. For more information on classes/packages, refer to the *VueBean Javadocs*.

VueBean Package

The VueBean component is central to the AutoVue client architecture. An application can embed the VueBean component and use its API to provide comprehensive support for file viewing, markup, real-time collaboration, and so on. The following diagram provides a graphical overview of how the VueBean can be used when developing your own application/applet.


Functionality cont'd
• Conversion
• File loading
• Open files, close files
• Markup mode
• Enter markup mode, load markups
Analysis
• File analysis
• Text search, page type (2D, 3D, and so on), bookmarks
• File properties
• File info, page count, dimensions, insertion and scaling
Functionality
• Document manipulation
• Layers, blocks, view, overlays
• View manipulation
• Panning, zooming, rotation, page selection, background color
Controllers
• Server Control
• Mouse listener control
VueBean

Note: It is possible to have multiple instances of the VueBean class. For example, when AutoVue is in Compare mode there are three instances of the VueBean class.

A typical VueBean usage scenario is as follows:

1. Create a VueBean Object.
2. Create a server control or use the default one obtained from the VueBean.
3. Use the server control to connect to the server and open a session on it.
4. View a file by invoking the `VueBean.setFile(DocID)` method.

The following file types are supported by the VueBean:

- Vector files (2D and 3D)
- Raster files
- Document files (MS Word, and so on)
- Spreadsheet files
- Archive files

The file type can be queried through the `VueBean.getFileType()`method and file information can be retrieved through the `VueBean.getFileInfo()` method.

You may have to convert a file to another file type. To do so, use the `Vuebean.convert()` method.

In its various modes, such as viewing and markup, the VueBean manages the representation of a file including the management of overlays, layers, and external references to other files or resources upon which the file depends. Use the `VueBean.getResourceInfoState()` method to query for resources that are attached to a file.

To search for a particular string in the file use the `VueBean.search(PAN_CtlSearchInfo)` method. The following is an example of how to build the `PAN_CtlSearchInfo` object.

```
// Construct the search object with arguments (Search String, Search Multiple
// Occurrences, Search Downwards, Wrapped Search, Match Case, Whole Word),
// in this example we search for the word “line”.
PAN_CtlSearchInfo searchInfo = new PAN_CtlSearchInfo("line", true, true,
                                                     true, false, true);
```

Note: Since the VueBean is only a client-side component, the connection to the AutoVue server must be established before any operation can be performed on the VueBean. Refer to Server Control for more information.

Event Package

`com.cimmetry.vuebean.event`

For VueBean-specific events, the event delegation model of the VueBean is slightly different from the standard Java one. Listeners such as `VueViewListener`, `VueFileListener`, `VueMarkupListener`, or `VueStateListener` should register to the VueBean's `VueEventBroadcaster` object instead of to the VueBean itself.
For example: `vueBean.getVueEventBroadcaster().addFileListener(listener)`.

This package provides interfaces and classes for VueBean event broadcasting. Every VueBean object has an event broadcaster. Depending on the operation type, the broadcaster notifies listeners using an instance of `VueEvent` or `VueModelEvent`. The following types of events are supported:

- File events
- View events
- Markup events
- State events
- Model events

Every event type has a corresponding event listener interface which is registered to the broadcaster. Objects that are responsible for handling of events should implement one or more of the listener interfaces.

The following code sample defines and registers an event handler:

```
import com.cimmetry.vuebean.*;
import com.cimmetry.vuebean.event.*;
.
.
.
final VueBean vueBean = getVueBean();// Get the valid active VueBean
if (vueBean != null) {
    VueFileListener eventHandler = new VueFileListener() {
        public void onFileEvent(VueEvent ev) {
            switch (ev.getType()) {
            case VueEvent.ONSETFILE:
                System.out.println("Set file: " + vueBean.getFile());
                break;
            case VueEvent.ONSETPAGE:
                System.out.println("Set page: " + vueBean.getPage());
                break;
            }
        }
    };
    vueBean.getVueEventBroadcaster().addFileListener(eventHandler);
}
.
.
.
```

VueEvent

`com.cimmetry.vuebean.event.VueEvent`

`VueEvent` object encapsulates information for all notifications sent by VueBean and is generated for the `VueFileListener`, `VueViewListener`, `VueMarkupListener` and `VueStateListener` interfaces. The event type is used to differentiate between a view event, file event, markup event or state event.

VueModelEvent

`com.cimmetry.vuebean.event.VueModelEvent`

The `VueModelEvent` class handles all notifications for model-related events such as entity attributes, 3D transformation, and so on. It is generated for objects implementing `VueModelListener` interface.

VueEventBroadcaster

`com.cimmetry.vuebean.event.VueEventBroadcaster`

`VueEventBroadcaster` is used to manage event delegation model for the VueBean. Each listener has to register to a VueEventBroadcaster to be notified of events in the VueBean. By design, each VueBean owns its own `VueEventBroadcaster`. However, you may find it useful to use only one `VueEventBroadcaster` for all beans by using the `VueBean.setVueEventBroadcaster` method.

VueFileListener

`com.cimmetry.vuebean.event.VueFileListener`

Objects implementing this interface listen for file event notifications (such as setting file, setting page, and so on).

VueMarkupListener

`com.cimmetry.vuebean.event.VueMarkupListener`

Objects implementing this interface listen for markup event notifications (such as entering or exiting markup mode).

VueViewListener

`com.cimmetry.vuebean.event.VueViewListener`

Objects implementing this interface listen for view event notifications (such as zoom, begin and end paint, and so on).

VueStateListener

`com.cimmetry.vuebean.event.VueStateListener`

Objects implementing this interface listen for state event notifications (such as server error, file error, and so on).

VueModelListener

`com.cimmetry.vuebean.event.VueModelListener`

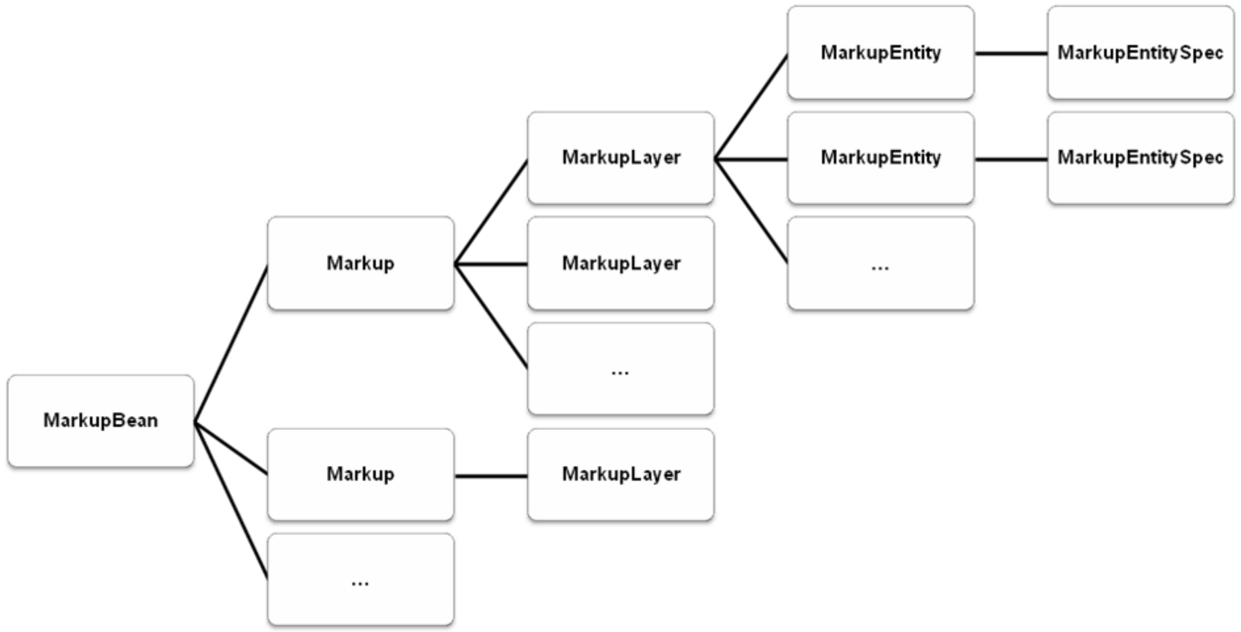
Objects implementing this interface listen for model event notifications (such as model attribute, selection, transformation changes, and so on).

MarkupBean Package

`com.cimmetry.markupbean`

The top–level class for the `com.cimmetry.markupbean` package is the MarkupBean class. MarkupBean represents the Markup functionality in the VueBean API. Each VueBean instance can contain only one MarkupBean instance, represented by a private member variable. Through the MarkupBean class, you can add/modify/remove Markup Files, Markup Layers, and Markup Entities, as well as open and save Markup Files.

The following diagram displays how the architecture of a Markup is structured into four separate levels: **Markups**, **Markup Layers**, **Markup Entities**, and **Markup Entity Specification**.



Markup
`com.cimmetry.markupbean.Markup`

This interface represents an individual Markup file. The key functionalities are as follows:

- Get/set information regarding the Markup files, such as:
    - Name
    - Visibility
    - Whether Markup is modified
    - Whether Markup is read-only
- Get information regarding the base file
- Get the layers in the Markup

MarkupLayer
`com.cimmetry.markupbean.MarkupLayer`

This interface represents an individual Markup layer. The key functionalities are as follows:

- Get/set information regarding the specific layer, such as:
    - Name
    - Color
    - Visibility
    - Default line type and width
- Get the entities in the Markup layer

MarkupEntity
`com.cimmetry.markupbean.MarkupEntity`

This interface represents an individual Markup entity. The key functionalities are as follows:

- Name
- Author
- Date modified
- Color

- Line type and width
- Tooltip text
- Visibility
- Selection state
- Get children entities of the specific entity
- Perform actions when user double-clicks on entity

MarkupEntitySpec

`com.cimmetry.markupbean.MarkupEntitySpec`

This class represents an entity's specification. Each entity has its own specification class that is derived from this class defines the attributes specific to that entity's context.

For example, the specification for a rectangle entity includes attributes for the XY coordinates of all four corners, while the specification for a text entity includes attributes for the contained text as well as its alignment.

Server Control

`com.cimmetry.vueconnection.ServerControl`

The ServerControl class handles the server connection object and the user session. Prior to using the VueBean, you must first set its ServerControl properties, connect to the server via the `connect()` method, and then open a session via the `sessionOpen()` method.

For example:

```
import com.cimmetry.vuebean.*;
import com.cimmetry.vueconnection.ServerControl;
…
VueBean bean = new VueBean();
ServerControl control = bean.getServerControl();
try {
    control.setHost(<SERVER URL>);
    control.connect();
    control.setUser("scarlati");
    control.sessionOpen();
} catch (Exception e) {
    System.out.println("Failed to connect to JVueServer.");
}
…
```

Note: Set the `server` URL to the VueServlet URL.

For example, `http://<HostName>:5098/servlet/vueservlet`

VueAction Package

`com.cimmetry.vueaction`

This package provides a hierarchy of classes implementing the AutoVue action API. It can be used to add graphical user interface (GUI) elements to different contexts (such as menu bar, toolbar, status bar, and so on). For example, when a menu option is selected in the GUI, a VueAction is triggered.

To add a new action to the AutoVue client, create a new action class by extending VueAction.

Use the methods in this package to:

- Specify resources for an action. For example, menu item text, an icon, tooltip text, and so on.
- Specify which resource bundle (a properties file with resource mappings) to search in for the action's resources.
- Specify sub-actions (for example, Zoom In, Zoom Out, Zoom Previous, and so on) for the action if it can perform more than one function.
- Receive a message signifying that the action should be performed. If the action has sub-actions, the sub-action to perform is specified.
- Specify properties of the views of the action or its sub-actions that appear in the GUI in the menu bars, toolbars, and popup menus. For example, whether the view can be selected (behaves as a checkbox) and/or whether it is enabled.
- Specify groups of sub-actions (if the action includes sub-actions) in which selection is exclusive (that is, in which only one sub-action can be selected at a time).

AbstractVueAction

`com.cimmetry.vueaction.AbstractVueAction`

The abstract class `AbstractVueAction` is the super class of all action classes. All actions performed on the session must be derived from this class or a descendent of this class.

VueAction

`com.cimmetry.vueaction.VueAction`

`VueAction` is an abstract class that extends `VueActionMultiMenu`. It provides a simple yet powerful interface for creating actions.

To create a new action class, you must extend this class. There are two ways to do this depending on whether your action performs a single function or multiple functions. The following sections describe both scenarios.

Create an action that performs a single function

1. Make sure your class extends `VueAction`.

2. In the constructor of your class, call the appropriate super constructor.
   Note: Since your action performs only one function, the super constructor takes the two String arguments: resource key and resource bundle. The resource bundle identifies the set of text files (one for each locale your action supports) containing the resources identified by the resource key for your action.

3. Implement a `perform()` method to override the one in VueAction.
   Note: This method is called when your action has been fired. In this method, enter your action's code.

4. Implement event handlers `onFileEvent` and `onViewEvent` to ensure that your action is enabled or disabled when appropriate.
   For example, if no base file has been loaded yet, your action will be disabled. However, once a file has been reloaded, your action must be enabled.

5. Create one or more resource files (one resource file per language your action supports) containing the resource keys and their values needed by your action. Together with any icon files used by your action, these files are referred to as a *resource bundle*.
   For an example of a resource file, refer to `VueFrame_en.properties` file.

6. Create a copy of AutoVue's .gui file and insert the name of your new action in the appropriate location.

To view an example of implementing an action that performs a single function, refer to Action that Performs a Single Function.

Create an action that performs multiple functions

1. Make sure your class extends `VueAction`.

2. In the constructor of your class, call the appropriate super constructor.
   Note: Since your action performs multiple functions, the super constructor takes one String argument: the resource bundle name. The resource bundle name indentifies the set of text files (one for each language your action supports) containing the resources for your action.

3. After you call the super constructor, call `defineSubAction()` to define each sub-action your action can perform.
   Note: In each case, specify the name by which you want to refer to the sub-action and its resource key. The resource key identifies where to find the resources for your action (for example, menu item text, icon, tooltip text and so on) in your resource bundle. Optionally, you can call `defineExclusiveGroup()` to define a subset of your sub-actions that form an

exclusive group. That is, sub-actions that are selectable where only one can be selected at a time.

4. Implement a `performSubAction(String)` method to override the one in VueAction.
   Note: This method is called when your action's sub-action has been fired. The method is passed the name of the sub-action fired, so that you will know which one to perform. In this method, enter your sub-action's code.

5. Implement event handlers `onFileEvent` and `onViewEvent` to ensure that your sub-actions are enabled or disabled when appropriate.
   For example, if no base file has been loaded, your sub-action will be disabled. However, once a file has been reloaded, your sub-actions must be enabled.

6. Create one or more resource files (one resource file per language that your action supports) containing the keys and values needed by your action.
   Note: Together with any icon files used by your action, these files are referred to as a *resource bundle*.

7. Create a copy of AutoVue's .gui file and insert the name of your new action in the appropriate location. You must also specify the appropriate sub-actions.

To view an example of implementing an action that performs multiple functions, refer to Action that Performs Multiple Functions.

6. Hotspots

AutoVue includes a hotspot capability that allows system integrators to create links between objects in AutoVue's data model and objects in an external system. With this feature, a solution can be built that integrates AutoVue tightly into other applications. By clicking on an area of a document in AutoVue, an action is triggered and/or information displays in other applications. Additionally, you can expose data from enterprise systems visually by changing the hotspot color.

AutoVue provides the following capabilities around hotspots:

- Hotspot definitions
- Data connection information defining AutoVue hotspots linked to external objects
- Tooltip to display on the hotspots defined in AutoVue
- Customization for hotspot selection notification
- Customization for available actions for selected hotspot

AutoVue provides the ability to define the following kinds of hotspots:

- Text-based hotspots in 2D and EDA documents (based on AutoVue's text search capability)
- Hotspots in Web CGM files

- Box (rectangular region) hotspots in 2D, EDA, and graphics documents
- Hotspots based on 3D attribute names and/or values

Note: It is possible to extend the AutoVue applet using the `VueAction()` method to implement a hotspot action. Refer to section Custom VueAction for a `VueAction()` hotspot example.

Text Hotspots in 2D and EDA Documents

Text hotspot support in 2D and EDA documents is based on regular expressions filtering graphical text strings based on AutoVue's text search. You can use regular expressions in the hotspot definition. Since AutoVue uses the Java library, it relies on Java's regular expression guidelines. For more information, refer to the Java regular expression guidelines at http://java.sun.com/developer/technicalArticles/releases/1.4regex.

Note: Text hotspot support is not available for raster formats, archive formats, Microsoft Office, Excel, RTF, and Outlook formats.

Web CGM Hotspots

In Web CGM files, hotspots are defined in the native file. The hotspot information contains three attributes:

- Name
- ID
- URI

External systems can interact with these hotspots using the VueBean API using a given name. AutoVue matches the name to the ID property of the hotspot. If this fails, AutoVue matches the name to the Name property in order to highlight a specific hotspot.

3D Hotspots

In 3D files, hotspots are defined by the attribute name. Optionally, an attribute value can be defined. If no attribute value is provided, then AutoVue identifies all parts with the attribute name as a hotspot. That is, the attribute value is used by AutoVue as a key to identify the hotspot attached to the owner part.

Note the following when defining 3D hotspots:

- Hotspots are not supported on 3D PMI entities.
- Attribute name or value used in 3D hotspot definitions cannot contain regular expressions. The attribute name/value should not contain any leading or trailing spaces and should exactly match the attribute name/value in the model.
- AutoVue does not currently support attribute names/values that contain a semi-colon (;).

- Internal attributes that AutoVue displays in 3D models (for example, Mesh Resolution, Transparency, and Layers) should not used when defining hotspots.

- To prevent conflicts in highlight color, it is recommended to use the Bounding Box Highlight for a 3D selection (default AutoVue setting) instead of the Entity Highlight. The conflicts result because of dynamic rendering, conflict in highlights on measurements, and performance due to redundant rendering of the model part.

- If a hotspot is defined with density an attribute, then the specified density value must be the same value saved in the native file without measurement units.

- It is not recommended to define hotspots with attributes that the user can modify after the model loads (for example, Color, Transparency, Display/Render Mode, Visibility, Highlight Color, and Bounding Box Color). If these attributes are used, AutoVue uses the values defined at the time of hotspot initialization and not the value set by the user.

Regional Hotspots

Box Hotspots

A box hotspot can be drawn on 2D, EDA, and Raster files. The dimensions/extents of the box are based on the coordinates displayed in the AutoVue status bar. Optionally, a user key can be used by AutoVue as an identifying key for the hotspot.

Note the following when defining box hotspots:

- Box hotspots are not supported on archive formats, Microsoft Word, Microsoft Excel, Microsoft Outlook, RTF formats.

- Vector files and raster files do not use the same World Coordinate System in AutoVue. Vector files use the bottom-left corner of the client area as the origin and the Y-axis oriented down-top, while the Raster-Files use the Top-Left corner as the origin and the Y-axis oriented top-down. This mismatch is already exposed in AutoVue with the current user interface (UI) because the mouse position is reported in World Coordinates System on the Status Bar of the UI. Since box hotspots are provided relatively to World Coordinate System, the box definitions need to consider this difference between raster and vector files.

AutoVue Behavior on Hotspots

AutoVue handles the following user interactions around hotspots:

- When the mouse cursor is on top of a hotspot, a visual hint is displayed to the user to indicate that the hotspot can be interacted (for example, can be clicked). A tooltip is displayed to show its description.
  - The hotspot tooltips have the following priority ranking in the stack of tooltips precedence:

        - Markup tooltip
        - Measurement tooltip
        - Hotspot tooltip
        - EDA Entity Information tooltip
        - Hyperlink tooltip
- When a user clicks on a hotspot on the display, a notification is fired to the external system with the information identifying the clicked hotspot and the mouse action (Click vs. Double-Click) and keyboard modifiers (Ctrl, Shift, Alt)
    - The mouse click and double-click on a hotspot fires the notification to an external application following these precedence rules:
        - Markup: Consumes the click / double-click.
        - Measurement: Consumes the click / double-click.
        - Hotspot: Notifies the external application but does not consume the click / double-click and allows the subsequent layers to handle the click / double-click as well.
        - Hyperlink: Does not consume the click / double-click.
        - EDA Entity selection, 3D Entity selection, Entity properties on double-click, and so on.
- When a user right-clicks on the hotspot, a menu displays with the available actions on this hotspot pre-defined by the integrator. When the user clicks on one of the menu items, a notification is fired to the external system with the information identifying the clicked hotspot and the action selected by the user.

AutoVue API for External System Interaction

An external system can call the AutoVue API for manipulating hotspots from the following user actions:

- Highlight (Multiple Selection, Add/Remove)
    - Text Highlight as used in text search.
    - 2D Entity Highlight for Web CGM format.
    - 3D Entity Highlight for 3D formats.
    - Box Highlight for box hotspots.
- Zoom to a hotspot, or the hotspots associated with a specific external object.
- Browse the hotspots associated with a specific external object using **Zoom Previous**/**Zoom Next**.

Note: When a user selects a hotspot, all hotspots associated with the same external object may be selected by using the highlight mechanism provided above.

AutoVue Hotspot API

There are two methods in the jVue class that handle hotspots:

- `setHotSpotHandler()` to define hotspots
- `performHotSpot()` to perform an action on a hotspot

Define Hotspot

```
setHotSpotHandler (final String definitionType, final String definitionKey,
final String Definition)
```

This method sets the hotspot handler for a given hotspot definition. This should typically be called before opening the file. It initializes hotspots in the files opened in AutoVue based on external application data.

Parameter	Description
definitionType	The hotspot definition type. Specify if hotspot is a WebCGM hotspot, text search hotspot, box hotspot, or a 3D attribute-based hotspot. See Hotspots Handler Types
definitionKey	The hotspot definition key. This is the identifier for the hotspot.
definition	A string separated by semicolons specifying hotspot definition parameters. For example: name1 = value1; name2 = value2. See Hotspot Definition Types.

Hotspot Handler Types

The hotspot definition types supported in `setHotSpotHandler()`are:

DEFINITION_TYPE_NATIVE	Native Web CGM hotspot.
DEFINITION_TYPE_TEXT	Text search hotspot.
DEFINITION_TYPE_BOX	Rectangular box hotspot.
DEFINITION_TYPE_3D_ATTRIBUTE	3D entity hotspot.

Hotspot Definition Types

The hotspot definition parameters supported in the key-value string parameter (definition) of the method `setHotSpotHandler()` are:

Common Definition Parameters

DEFINITION_TOOLTIP	The tooltip that displays when a mouse point hovers over a hotspot defined by the handler.
DEFINITION_ONINIT	The java script method to call when page is loaded and ready to interact.
DEFINITION_FUNCTION	The JavaScript function to call when user performs an action on the hotspot.
DEFINITION_ACTIONS	Popup actions to show when user right-clicks on a hotspot.
DEFINITION_COLOR	The highlight color to use when user hovers the mouse cursor over a hotspot. Note that AutoVue parses the RGBA value as a string. **Example:** `(R, G, B, [A])`. Refer to 3D Hotspot and Rectangular Box Hotspot examples for more information.

Text Definition Parameters

DEFINITION_REGEX	Regular expression to use only in Text Search Hotspot handlers.
DEFINITION_MATCHCASE	Whether to handle case sensitivity in Text Search Hotspot handlers only.

3D Definition Parameters

DEFINITION_ATTRIB_NAME	The attribute name assigned to a 3D hotspot on the model.
DEFINITION_ATTRIB_VALUE	The attribute value assigned to a 3D entity on the model. (Optional)
DEFINITION_MATCHCASE	Whether to handle case sensitivity when searching

	name and value attributes assigned to 3D entities
Rectangular Box Definition Parameters	
DEFINITION_BOX	Define the bounds of the rectangular box given the minimum and maximum points. Where {X1, Y1} and {X2, Y2} are the coordinates of the box minimum and maximum points. **Syntax:** DEFINITION_BOX=#X1#Y2#X2#Y2
DEFINITION_USER_KEY	Define a user key for the rectangular box. This user key allows you to link multiple boxes with various definitions to the same external object. (Optional) **Syntax:** DEFINITION_USER_KEY=box1

Perform an Action on a Hotspot

```
performHotSpot (final String definitionKey, final String hotspotKey, final
String action, final String params)
```

Perform a hotspot action on the given hotspot. This method should be called during the file session when the hotspots have been already initialized (only after the external application notifies that hotspots have been initialized in the file).

Parameters	**Description**
definitionKey	The hotspot definition key (the hotspot identifier) provided at creation.
hotspotKey	The hotspot property key string found based on the definition key.
action	The action to perform on the hotspot. Refer to Hotspots Actions.
params	A string separated by semicolons specifying hotspot action parameters. For example: name1 = value1; name2 = value2.

Hotspot Actions

The hotspot actions supported in `performHotSpot()`and their arguments are:

Action Name	Description	Arguments
HIGHLIGHT	Perform a highlight	**HOTSPOT_COLOR:** The color for a highlight to add (RGBA Format). If this argument is not provided, the action is interpreted as a Highlight Removal.
ZOOMTO	Zoom to all hotspot instances	None
ZOOMNEXT	Zoom to the next hotspot instance	None
ZOOMPREV	Zoom to the previous hotspot instance	None

Interactions with Hotspots from JavaScript

The following is a code prototype for a custom JavaScript function call to initialize hotspots when the file/page loads:

```
initialization_script(String definitionKey)
```

The following is a code prototype for a custom JavaScript function call when a user interacts with hotspots:

```
notification_script(String definitionKey, String hotspotKey, String action,
int keyModifiers, String properties)
```

`action` may be a custom action sent during the definition of the hotspot handler (RMB actions) or one these two predefined actions:

OnHotSpotClicked	To send when user clicks on the hotspot
OnHotSpotDoubleClicked	To send when user double-clicks on the hotspot

`properties` that could be sent to the external application notification script are:

PROPERTY_ ID	ID of Native WebCGM Hotspots

PROPERTY_ NAME	Name of Native WebCGM Hotspots
PROPERTY_URI	URI of Native WebCGM Hotspots

Hotspot Samples

The following sections provide sample code on how to add hotspot capability to AutoVue and how to define text hotspots, rectangular box hotspots, and 3D hotspots.

Adding a Hotspot

The following hotspot example shows how the setHotSpotHandler() and performHotSpot() methods are implemented to add hotspot capability to AutoVue. This example only adds one definition, but it is possible to add multiple definitions.

1. Initialize the hotspots with the ONINIT applet parameter. This parameter is used to call the `onAppletInit()` method after the AutoVue applet has initialized.

   Note: If a newly added definition key already exists, then the existing definition is replaced by the new one.

```
<PARAM NAME="ONINIT" VALUE="onAppletInit();">

function onAppletInit() {
    var handlerStr = "DEFINITION_REGEX=AutoVue;
    DEFINITION_TOOLTIP=AutoVue 2D Professional";
    // The following function is called once when AutoVue is ready to
    // interact with a hotspot.
    handlerStr += ";DEFINITION_ONINIT=onHotSpotInit";
    // The following function is called each time a hotspot is fired.
    handlerStr += ";DEFINITION_FUNCTION=onHotSpot;
    DEFINITION_ACTIONS=Menu1, Menu2";
    color = ((128 & 0xFF) << 24) | ((0 & 0xFF) << 16) | ((0 & 0xFF) <<
    8) | ((255 & 0xFF) << 0);
    handlerStr += ";DEFINITION_COLOR=" + color;

    //The following call sets up the hotspot definition.

    window.document.applets["JVue"].setHotSpotHandler("DEFINITION
        _TYPE_TEXT", "AV2D", handlerStr);

}
```

2. Method `onHotSpotInit()` is called for each definition when the current page is loaded and ready for hotspot interactions. Note that the method name should be exactly the same as the one specified in the hotspot definition DEFINITION_ONINIT in step 1.

```
function onHotSpotInit(hotspotDefinitionKey) {
    alert("HotSpot definition initialized: " + hotspotDefinitionKey);
}
```

3. The following `onHotSpot()` method is invoked when a hotspot is fired when the user either clicks on the hotspot or by selecting one of the Hotspot menu items defined in variable DEFINITION_ACTION in step 1.

```
function onHotSpot(hotspotDefinitionKey, hotspotKey, action, modifiers,
properties) {
 if (equalsIgnoreCase(action, "onHotSpotClicked")) {
        alert("User clicked on hotspot: " + hotspotKey);
 } else if (equalsIgnoreCase(action, "onHotSpotDoubleClicked")) {
        alert("User double clicked on hotspot: " + hotspotKey);
 } else if (equalsIgnoreCase(action, "Menu1")) {
        alert("User Peformed Menu1 action: " + hotspotKey);
 } else if (equalsIgnoreCase(action, "Menu2")) {
        alert("User Peformed Menu2 action: " + hotspotKey);
    }
}
```

Note that the method name should be exactly the same as the one specified in the hotspot definition DEFINITION_FUNCTION in step 2. The `onHotSpotClicked` and `onHotSpotDoubleClicked` methods are predefined keys when the user clicks on the hotspot.

4. The following code performs specific actions on the clicked hotspot such as Highlight Zoom and so on.

.

```
// Highlight the "AutoVue" hotspot, "AV2D" is the definition key.
// Color : alpha | red | green | blue
params = "HOTSPOT_COLOR=" + (((128 & 0xFF) << 24) | ((255 & 0xFF) <<
16) | ((255 & 0xFF) << 8) | ((0 & 0xFF) << 0));
window.document.applets["JVue"].performHotSpot("AV2D", "AutoVue",
"Highlight", params);

// To clear the hotspot highlight simply set the params (color) to
null.
window.document.applets["JVue"].performHotSpot("AV2D", "AutoVue",
"Highlight", null);

// To clear the definition highlights, set the hotspot key to null.
```

```
window.document.applets["JVue"].performHotSpot("Highlight", "AV2D",
null, null);

// To clear all hotspot highlights, set the definition key to null.
window.document.applets["JVue"].performHotSpot(null, null, "Highlight",
null);

// Zoom to the next "AutoVue" hotspot.
window.document.applets["JVue"].performHotSpot("AV2D", "AutoVue",
"ZoomNext", null);

// Zoom to the previous "AutoVue" hotspot.
window.document.applets["JVue"].performHotSpot("AV2D", "AutoVue",
"ZoomPrev", null);
```

3D Hotspot

The following example shows how to define a 3D hotspot.

1. Get the JVue Applet.

```
jApplet = window.document.applets["JVue"];
```

2. Define a 3D hotspot.
   This example defines a hotspot matching a part number in a 3D Unigraphics assembly file. The sample file is included with the AutoVue Client/Server Deployment installation: <AutoVue Installation Folder>`/samples/3D/Unigraphics/3DUnigraphics_iLearn-Assy.prt.`

```
item00003Def = "DEFINITION_ATTRIB_NAME=PART_NUMBER;
DEFINITION_ATTRIB_VALUE=ITEM-UG-00003;"
        + "DEFINITION_TOOLTIP=Board;"
        + "DEFINITION_ONINIT=onHotSpotInit;"
        + "DEFINITION_FUNCTION=onHotSpot;"
        + "DEFINITION_ACTIONS=Add Part, Remove Part;"
        + "DEFINITION_COLOR=(255, 0, 0)";
```

3. Set the 3D hotspot handler.

```
jApplet.setHotSpotHandler("DEFINITION_TYPE_3D_ATTRIBUTE", "item00003",
item00003Def);
```

Rectangular Box Hotspot

The following example shows how to define a rectangular box hotspot.

1. Get the JVue Applet.

```
jApplet = window.document.applets["JVue"];
```

2. Define a rectangular hotspot.
   This example defines a rectangular hotspot that encloses the Oracle logo included in the PDF sample file that is included with the AutoVue Client/Server Deployment installation: <AutoVue Installation Folder>`/samples/Desktop-Office/Basell_Autovue_Case_Study.pdf.`

Note: The rectangle coordinates are defined by `#minX #minY #maxX #maxY`. Note that each coordinate must be preceded by a dash (#).

```
oracleDef = "DEFINITION_BOX=#6.4 #0.7 #8.1 #0.4;
DEFINITION_USER_KEY=oracle;"
        + "DEFINITION_TOOLTIP=www.oracle.com;"
        + "DEFINITION_ONINIT=onHotSpotInit;"
        + "DEFINITION_FUNCTION=onHotSpot;"
        + "DEFINITION_ACTIONS=Open Link;"
        + "DEFINITION_COLOR=(0, 0, 255, 64)";
```

3. Set the rectangular box hotspot handler.

```
jApplet.setHotSpotHandler("DEFINITION_TYPE_BOX", "oracleBox",
oracleDef);
```

Text Hotspot

The following example shows how to define a text hotspot.

1. Get the JVue Applet.

```
jApplet = window.document.applets["JVue"];
```

2. Define a text hotspot. The following example defines a text hotspot (regular expression) matching the AutoVue string. The PDF sample from Rectangular Box Hotspot Sample includes the *AutoVue* string in multiple locations.

```
autovueDef = "DEFINITION_REGEX=AutoVue; DEFINITION_MATCHCASE=false;"
        + "DEFINITION_TOOLTIP=AutoVue Professional;"
        + "DEFINITION_ONINIT=onHotSpotInit;"
        + "DEFINITION_FUNCTION=onHotSpot;"
        + "DEFINITION_ACTIONS=AutoVue 2D, AutoVue 3D, AutoVue EDA,
           AutoVue Electro-Mechanical;"
        + "DEFINITION_COLOR=(0, 255, 0, 128)";
```

3. Set the text hotspot handler.

```
jApplet.setHotSpotHandler("DEFINITION_TYPE_TEXT", "AutoVue",
autovueDef);
```

7. Use Cases

The following sections provide information on typical use cases you may come upon when creating an AutoVue API applet/application or adding enhanced functionality to the AutoVue client. Refer to the *VueBean JavaDocs* for more information.

Note: Throughout this document, `m_vueBean` is used as a valid active `VueBean` object and `m_JVue` as a valid `JVue` applet object. This is done assuming that the methods or segments of code that use objects have access to a class which owns them.

Building an AutoVue API Application

A good starting point with the AutoVue API is to create an application that opens and displays a file. This section provides detailed steps for creating a file open application using the AutoVue API.

1. Import required packages.

```
import java.awt.*;
import java.awt.event.*;

import javax.swing.*;

import com.cimmetry.core.*;
import com.cimmetry.util.Messages;
import com.cimmetry.vuebean.*;
import com.cimmetry.vueconnection.ServerControl;
```

2. Create a Java class, `ApplicationSample`, that can be run as a stand-alone application, and declare all external parameters and internal data members.

```
public class ApplicationSample {
      private String  m_host = "socket://localhost:5099";
      private String  m_user = "guest";
      private String  m_fileName = null;
      private String  m_verbose = null;
      private String  m_format = "AUTO";

      // Internal data members
      private VueBean  m_vueBean = null;
      private ServerControl m_control = null;
      private static JFrame m_frame = null;

      private JMenu m_fileMenu = null;
```

3. Create stand-alone application support.

```
      public static void main(final String args[]) {
            ApplicationSample app  = new ApplicationSample();
            app.init(args);
      }
      public void init(final String[] args) {
        switch (args.length) {
            case 4:
                 m_verbose    = args[3];
            case 3:
                  m_fileName   = args[2];
            case 2:
                  m_user       = args[1];
            case 1:
                  m_host       = args[0];
            default:
                  break;
        }
        init();
     }
```

4. Initialize the application.

```
public void init() {
        // Setup verbosity
        if (m_verbose != null && m_verbose.length() > 0) {
                Messages.setVerbosity(m_verbose);
        }
…
```

Note: The `init()` method continues until step 13.

5. Establish a connection with the server.

```
m_control = new ServerControl();
try {
      m_control.setHost(m_host);
      m_control.connect();
} catch (Exception e) {
      System.out.println("Unable to connect to:"+m_host);
      e.printStackTrace();
      return;
}
```

6. Open the session.

```
try {
      m_control.setUser(m_user);
      m_control.sessionOpen();
} catch (Exception e) {
      System.out.println("Unable to open session for " + m_user);
      e.printStackTrace();
      return;
}
```

7. Initialize the frame.

```
m_frame = new JFrame("VueBean Sample");
m_frame.setBounds(100, 100, 640, 480);
m_frame.addWindowListener( new WindowAdapter() {
      public void windowClosing( WindowEvent e) {
            destroy();
      }
});
```

8. Set the menus and actions.

```
setMenuBar();
```

9. Create the bean.

```
m_vueBean = new VueBean(m_format);
m_vueBean.setServerControl(m_control);
m_vueBean.setBackground(Color.lightGray);
```

10. Set up the viewer as a model event listener.

```
m_vueBean.setVueEventBroadcaster(m_vueBean.getVueEventBroadcaster());
```

11. Add the VueBean to the frame.

```
m_frame.getContentPane().add(m_vueBean);
```

12. Display the frame.

```
m_frame.setVisible(true);
```

13. Show the file.

```
    updateFile();

}// Closing bracket for init() method
```

Note: This step marks the end of the `init()` method.

14. Close the session.

```
public void destroy() {
      try {
                  m_control.sessionClose();
      } catch (Exception ex) {
            ex.printStackTrace();
      }
      m_frame.setVisible(false);
      m_frame.dispose();
      System.exit(0);

}
```

15. Get the attached VueBean.

```
public VueBean getVueBean() {
        return m_vueBean;
}
```

16. Get the attached frame.

```
public JFrame getFrame() {
        return m_frame;
}
```

17. Get the file menu.

```
protected JMenu getFileMenu() {
        return m_fileMenu;
}
```

18. Get the frame. The following method sets the applet's menubar to File Open, Print, and Exit.

```
public void setMenuBar() {
            m_fileMenu = new JMenu("File");
            JMenuItem menuItem;

            // File open menu item
            menuItem = m_fileMenu.add(new JMenuItem("Open"));
            menuItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                showFile();
            }
            });

            // set the Applet's menu bar
            JMenuBar menu_bar = new JMenuBar();
            m_frame.setJMenuBar(menu_bar);
            menu_bar.add(m_fileMenu);
}
```

19. Load the file.

```
public void updateFile() {
            // Set the vuebean's file
            if (m_fileName != null && !m_fileName.equals("")) {
                  m_vueBean.setFile(new DocID(m_fileName));
                  m_vueBean.setBackground(Color.lightGray);
                  initMouseListeners();

              javax.swing.SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                    m_vueBean.getController().zoomFit();
                }
             });
            }
}
```

20. This method initializes listeners for mouse actions and sets the right mouse button to zoom out.

```
private void initMouseListeners() {
            MouseListener mouseSet = new MouseAdapter() {
            public void mouseClicked(MouseEvent me) {
                if ((me.getModifiers() & MouseEvent.BUTTON3_MASK) != 0)
{
                    if ( m_vueBean.getFileInfo() != null ) {
                        m_vueBean.getController().zoomFit();
                        me.consume();
                    }
                }
            }
        };
            VueMouseToolManager mtm = m_vueBean.getMouseToolManager();
            mtm.setMouseListener(VueMouseToolManager.TOP_LAYER,
mouseSet);
}
```

21. Display the client-side (upload) File Open dialog and set the selected file in the bean.

```
      public void showFile() {
            FileDialog openDlg = new FileDialog(m_frame, "File Open",
                                                FileDialog.LOAD);
            openDlg.setVisible(true);
            m_fileName = "upload://"+openDlg.getDirectory() +
                         openDlg.getFile();
            openDlg.dispose();
            updateFile();
      }
}
```

Note: End of class ApplicationSample. In order to run the application properly, an AutoVue server needs to be running on either a local or remote host that is specified through command line arguments. Refer to step 3 for the definition of each argument.

Implementing Functions from AutoVue in a Second Applet

When creating your own customized Java applets/applications based on AutoVue API components, it is sometimes easier to implement pre-existing code from AutoVue. Many AutoVue and VueBean methods can be easily called through JavaScript in your HTML page by first getting a handle to the AutoVue object with the following JavaScript call:

```
document.applets["JVue"]
```

However, some functionality may be difficult to implement directly through JavaScript and must be written in Java. An efficient way to do this is through a separate Applet that references the AutoVue applet. The steps are as follows:

1. Create your own Java class (for example, App2.java) that extends Applet.
2. Import the appropriate packages and classes (such as `java.applet.Applet`, `com.cimmetry.vuebean.VueBean`, `com.cimmetry.jvue.JVue`, and so on).
3. Add the following two variables to your class:

   ```
   private Applet m_applet;
   private JVue m_jv;
   ```

4. Define an `attach()` method for your class and add the following two lines of code to obtain a handle to the AutoVue (JVue) applet instance:

   ```
   m_applet = getAppletContext().getApplet("JVue");
   m_jv = (JVue)m_applet;
   ```

   You can now call AutoVue methods on the `m_jv` variable, and can also obtain a handle to the VueBean instance with `m_jv.getActiveVueBean()`.
   Note: For more information regarding the `getAppletContext()` method, refer Java documentation for the `AppletStub` interface in the `java.applet.package`.

5. Compile your class (make sure to include jvue.jar in the classpath) and place your Java class file in your CODEBASE location.
   Note: If your custom applet has inner classes and generates additional class files upon compilation, you should combine those classes in a JAR file and set the JAR files as your second applet's archive parameter.

6. In your HTML page, declare your Applet as follows:

```
  <APPLET
      NAME="App2"
      CODE="App2.class"
      ARCHIVE="jvue.jar,jogl.jar,gluegen-rt.jar"
      CODEBASE="http://<SERVERNAME>/jVue"
      HEIGHT="0%" WIDTH="0%"
      MAYSCRIPT>
  </APPLET>
```

   You can either modify frmApplet.html in the AutoVue root directory or use it as a template to create your own HTML page.

   Note: Make sure to set the `CODEBASE` and parameter appropriately based on your Web server or application server hosting the Applet.

   For example: `CODEBASE="http://localhost:80/jVue"`

7. In your HTML page, initialize your new Applet in the on`AppletInit()` method for the AutoVue Applet by adding the following line:

```
document.applets[“App2”].attach();
```

This is the easiest way to initialize the second Applet in this particular example, since the frmApplet.html page already contains the on`AppletInit` method.

Custom VueAction

Action that Performs a Single Function

The following example shows how to implement a custom action for AutoVue that displays a dialog that lists all components of a drawing that are represented by hotspots and that were double-clicked by the user.

Note: The following are segments of the source code of the VueAction example to illustrate the essential steps of creating a custom action, it may not compile if you just copy and paste the code here. For the complete source code, refer to *PartListAction.java*.

1. Import all required packages.

```
import java.awt.*;
import java.awt.event.*;
import java.util.Vector;

import com.cimmetry.vuebean.*;
import com.cimmetry.vuebean.event.*;
import com.cimmetry.vueframe.*;
import com.cimmetry.vueframe.hotspot.*;
import com.cimmetry.core.*;
import com.cimmetry.dialogs.VueBasicDialog;
import com.cimmetry.vueaction.VueAction;
import com.cimmetry.gui.*;
```

2. Make your class extend `VueAction`.

```
public class PartListAction extends VueAction { …}
```

3. In the constructor of your class, call the appropriate super constructor. Since this action only performs a single function, a call to the super-constructor of VueAction takes this action's resource key as well as its resource bundle name.

```
public PartListAction() {
      super("LIST_PARTS",RESOURCE_BUNDLE_NAME);
      setViewListener(true);
}
```

Note: The resource bundle name here is the common part of resource bundle files for different languages. The actual name of a resource bundle file should include the language suffix and file extension. For example, `PartListAction_en.properties` is the resource bundle file for English.

4. Implement a perform method for this action.

```
public void perform() {
      PartInfo[] parts = new PartInfo[m_cart.size()];
      m_cart.copyInto(parts);
      PartListDialog dialog = new PartListDialog(getFrame(), parts);
      dialog.show();
```

5. Implement the event handlers `onFileEvent` and `onViewEvent` to notify when a file has changed and to update the user-interface.

```
public void onFileEvent(VueEvent e) {
        switch (e.getEvent()) {
        case VueEvent.ONPAGELOADED:
            setEnabledByCurrentState();
            break;
        }
    }
public void onViewEvent(VueEvent e) {
      switch(e.getEvent()) {
      case VueEvent.ONLINKCLICKED:
      Object[] params = (Object[]) e.getParameter();
      MouseEvent me = (MouseEvent) params[0];
      if (me.getClickCount() == 2) {
            Object link = params[1];
            if (link instanceof HotSpot) {
                  HotSpot hotspot = (HotSpot) link;
                  PartInfo part = getPartInfo(hotspot);
                  m_cart.addElement(part);
            }
      }
            break;
      default:
          super.onViewEvent(e);
          break;
      }
}
```

6. The dialog that lists all components of a drawing extends `VueBasicDialog`. You must implement your own constructor that calls the super-constructor and over-rides buildDialog() and `buttonAction(int).`

```
public static class PartListDialog
extends
      VueBasicDialog
implements
      ActionListener (…)

protected void buildDialog() {
      super.buildDialog();
…
}

protected void buttonAction(int index){…}
```

7. You must define a model for the table that represents the displayed product parts list.

```
public static class PartListModel implements CTableModel { …}
```

8. Close the `PartListDialog` method.

9. Get a `PartInfo` associated with a given hotspot.

```
private PartInfo getPartInfo (HotSpot hotspot) {
      return new PartInfo(hotspot.getDefinitionKey(),
            hotspot.getHotSpotKey(),
            hotspot.getProperty(HotSpot.PROPERTY_DESCRIPTION));
)
```

Action that Performs Multiple Functions

The following example shows how to implement a custom action for AutoVue that performs multiple tasks. The custom action consists of several related sub-actions that access information about parts of a model. One sub-action permits the user to order a part, another permits the user to display part information, and another sub-action displays a list of all the model’s parts.

Note: The following are segments of the source code of the VueAction example to illustrate the essential steps of creating a custom action, it may not compile if you just copy and paste the code here. For the complete source code, refer to *PartCatalogueAction.java*.

1. Make your class extend `VueAction`.

```
public class PartCatalogueAction extends VueAction {
    private static final String RESOURCE_BUNDLE_NAME =
                                "/PartCatalogueAction";

    // Names of the sub-actions used in *.gui file
    private static final String ORDER_SUBACTION = "Order";
    private static final String LIST_PARTS_SUBACTION = "ListParts";
    private static final String SHOW_INFO_SUBACTION = "ShowInfo";
    …
}
```

2. In the constructor of your class, call the appropriate super constructor.

```
public PartCatalogueAction() {
        super(RESOURCE_BUNDLE_NAME);
```

Note: The resource bundle name here is the common part of resource bundle files for different languages. The actual name of a resource bundle file should include the language suffix and file extension. For example, `PartCatalogueAction_en.properties` is the resource bundle file for English.

3. Call `defineSubAction` to define each sub-action your action can perform.

```
       defineSubAction(ORDER_SUBACTION,"ORDER_PART");
       defineSubAction(LIST_PARTS_SUBACTION,"LIST_PARTS");
       defineSubAction(SHOW_INFO_SUBACTION,"SHOW_PART_INFO");
}
```

4. Implement a `performSubAction(String)` method to override the one in VueAction.

```
public void performSubAction(String subActionName) {
        if (subActionName.equals(ORDER_SUBACTION)) {

            //Code for performing the “Order” subaction
            …
        } else if (subActionName.equals(LIST_PARTS_SUBACTION)) {

            //Code for performing the “List Parts” subaction
            …
          }
…
}
```

5. Implement the event handlers on`FileEvent` and on`ViewEvent` to ensure that your sub-actions are enabled or disabled when appropriate.

```
public void onFileEvent(VueEvent e) {
        switch (e.getEvent()) {
        case VueEvent.ONSETFILE:
            //Code for handling ONSETFILE event
            …
        case VueEvent.ONPAGELOADED:
            //Code for handling ONPAGELOADED event
            setEnabledByCurrentState();
            …
            break;
        }
    }
public void onViewEvent(VueEvent e) {
        switch(e.getEvent()) {
        case VueEvent.ONVIEWCHANGED:
            //Code for handling ONVIEWCHANGED event
            setEnabledByCurrentState();
            …
            break;
        case VueEvent.ONOPTIONSCHANGED:
            //Code for handling ONOPTIONSCHANGED event
            …
            break;
        }
    }
```

6. Create one or more resource files, one per language your action supports, containing the keys and values needed by your action. For example:

```
…
FILE_MARKUP_NEW_MARKUP=&New Markup, 32_new_markup_red.png, New Markup
FILE_MARKUP_OPEN=&Open..., 57_markup_red.png, Open Markup(s)
FILE_MARKUP_SMALL=     &Markup, 57_markup_red_small.png, Markup
FILE_MOCKUP=&Import File for Mockup..., 115_dmu.png, Import File for
Mockup
FILE_MRU=Recent Files
FILE_NOTFOUND=File not found.
FILE_NOTSUPPORTED=This file format is not supported by your server.
FILE_NOTUPLOADED=Failed to upload file.
FILE_OPEN=&Open...\\tCTrL+O, 59_open.png, Open File
FILE_OPEN_SERVER=Open from &Server..., , Open a file from the server
…
```

Similarly, in our resource bundle file for English language `PartCatalogueAction_en.properties`, it should contain the resource keys for the `PartCatalogueAction` shown in the following:

```
…
ORDER_PART = &Order Part, order_part.png, Order a part
LIST_PARTS = &List Parts, list_parts.png, List product parts
SHOW_INFO_SUBACTION = &Show Part Info, show_info.png, Show part
information
…
```

Note: Each resource key has three entries separated by a comma ",". The first entry (for example, `&Order Part`) is the text displayed on the GUI item (such as a menu item or toolbar button) and the ampersand "&" defines a shortcut key. The second entry (for example, `order_part.png`) is the file of the icon displayed on its GUI item. The third entry is the tooltip text for the GUI item. The second and third entries are optional. You should get the icon files ready if needed and add them to the JAR file for your custom action.

7. Make a copy of AutoVue's `default.gui` file located in the <AutoVue Installation Root>\bin directory, and insert the name of your new action in the appropriate locations of your GUI file. Note that for an action that performs multiple tasks, you must also specify the appropriate sub-actions.
   Note: For information on how `PartCatalogueAction` sub-actions are inserted into a menubar, toolbar, and custom pop-up menu, refer to `default.gui` and the custom.gui file located in the "<AutoVue Installation Root>\examples\VueActionSample\ directory.

8. To allow the custom action to take effect, you may need to create a JAR file with your custom VueAction classes and all resource files they depend on.
   For example, for the resource bundle files for different languages and icon files, if any, place your JAR file under AutoVue's bin directory or its web root directory and include your JAR file in the classpath of the stand-alone JVue application or ARCHIVE list of the JVue applet.

9. You must specify the name of the modified GUI file through Applet or Command line parameters.
   For more information, refer to the "Customizing the GUI" section of the *Installation and Configuration Guide*.

Directly Invoking VueActions

It is possible to develop your own customized user interface in an HTML page that incorporates AutoVue functionality. To do so, you must call `invokeAction()` of the `com.cimmetry.jvue.JVue` applet from the HTML page. This call to the action can be done purely through JavaScript.

Markups

The following show some ways to execute common Markup actions.

Note: Various MarkupBean functionalities (and various functionalities throughout the AutoVue API) require the use of the *Property* class. This class is used to define various property hierarchies for other classes in the API.

Entering Markup Mode

`VueBean.setMarkupModeEnable(true)`

Checks whether the MarkupBean member is null, and if so:

- Instantiates a new MarkupBean object
- Gets the markup settings from the user's profile
- Sets the markup-specific mouse listeners
- Points the VueBean's MarkupBean member to the new instance
- Broadcasts `VueEvent.ONENTERMARKUPMODE`

Checking Whether Markup Mode is Enabled

`VueBean.isMarkupModeEnabled()`

Checks whether the MarkupBean member is null.

Exiting Markup Mode

`VueBean.setMarkupModeEnabled(false)`

Checks whether the MarkupBean member is null, and if not:

- Sets the MarkupBean member to null
- Removes markup-specific mouse listeners

- Saves markup settings into the user's profile
- Broadcasts `VueEvent.ONEXITMARKUPMODE`

Adding an Entity to an Active Markup/Layer

`MarkupBean.setMarkupEntityClass(`*`<class name of desired markup entity>)`*
`MarkupBean.setActionMode(MarkupBean.ACTION_MODE_ADD)`

Adds a new markup entity to the active layer in an active Markup (based on the class name provided) through user input from the GUI. To programmatically add a markup entity, you must call: `MarkupBean.addMarkupEntity(MarkupEntitySpec spec)`

Enumerating Entities

`MarkupLayer.getEntities()`
or
`MarkupBean.getMarkupEntities(MarkupLayer layer)`

Returns an array of `MarkupEntity` objects in a markup layer.

Getting Entity Specification of a Given Entity

`MarkupBean.getMarkupEntityFullSpec(MarkupEntity ent)`

You must pass in the specific entity for MarkupBean to return its specification.

Changing Specification of an Existing Entity Programmatically

`MarkupBean.exchangeMarkupEntity(MarkupEntity a, MarkupEntity b)`

Allows you to dynamically change the properties of an existing entity. That is, it replaces markup entity *a* with markup entity *b*. Some properties can be directly changed via the following set methods of `MarkupEntitySpec` inherited from the `MarkupGraphicSpec` parent class:

- setColor
- setFillColor
- setFilled
- setFilltype
- setFont
- setLineType
- setLineWidth

For other properties, such as the entity position, entity size, entity text content, and so on, there are no `set` methods directly on the specification. As a result, you must do the following:

1. Create a new specification instance (with the new properties).
2. Create a new entity instance (with the new specification).
3. Use `exchangeMarkupEntity` to replace the existing entity.
4. Make a call to `MarkupBean.repaint()`.

Adding a Text Box Entity

The following code shows how to add a text box entity programmatically.

```
import com.cimmetry.markupbean.*;
import com.cimmetry.gui.*;
.
.
.
public void addTextBox(String text){


  m_vueBean.setMarkupModeEnabled(true);

  CTextPane textPane = GUIFactory.createTextPane();
  textPane.setText(text);
  byte[] textRTF = textPane.getRTF();
  PAN_CtlRange rect = new PAN_CtlRange(m_vueBean.getViewExtents());
  rect.scale(0.2);
  TextBoxSpec spec = new
  TextBoxSpec(m_vueBean.getMarkupBean().getMarkupEntitySpec(),
              rect.min, textRTF,
rect.max,TextBoxSpec.MRK_ALIGN_BOTTOMCENTER);

  m_vueBean.getMarkupBean().setMarkupEntityClass(spec.getEntityClassName());
  m_vueBean.getMarkupBean().addMarkupEntity(spec);
}
```

Open Existing Markup

`MarkupBean.readMarkup(InputStream is)`

`InputStream` can be relative to the client (for example, a locally-saved Markup), relative to the AutoVue server (for example, managed by AutoVue's markups.map file) or from a DMS/PLM/ERP.

To read a Markup from the AutoVue server, you first must get the `InputStream` by reading the Markup *Property* from the VueBean, and then choose a child property (that represents a Markup file) you want to read into the stream. The following code illustrates how to create a markup, save it, and then read it into the MarkupBean.

```
import com.cimmetry.markupbean.*;
.
.
.
Property[] name = {new Property(Property.PROP_DOC_NAME, <your Markup name>)};
Property prop = new Property(Property.PROP_MARKUP, name);
ByteArrayOutputStream os = new ByteArrayOutputStream();
m_markupBean.writeMarkup(os);
m_vueBean.writeMarkup(prop, os);
Property masterMarkup = m_vueBean.getMarkupProperty();
Property[] listMarkups =
masterMarkup.getChildrenWithName(Property.PROP_MARKUP);
Property aMarkup = listMarkup[0];
InputStream is = m_vueBean.readMarkup(aMarkup);
m_markupBean.readMarkup(is);
…
```

Saving Markups to a DMS/PLM

Note: This example is not applicable if you are building an ISDK-based application.

The following example uses the same concept as saving a Markup back to the AutoVue server; you must set the appropriate *Property* and build the `OutPutStream`. In order to build the Markup property, you need to first read the `CSI_Markups` property so that you can retrieve the values that the user sets in the Markup Save dialog.

```
private void saveMarkupToDMS() {
  // Gets the master markup property for the current file, that is,
  // the property containing the GUI and the markup list
  Property propMaster = m_vueBean.getMarkupProperty();

  // If none, we have a problem
  if ( propMaster == null ) {
      System.out.println("Could not get master markup property!");
      return;
  } else {
      // Get the GUI child property under master markup property
      Property[] listGuiProp
=propMaster.getChildrenWithName(Property.PROP_GUI);
      if (listGuiProp == null || listGuiProp.length != 1) {
          System.out.println("No valid GUI property!");
          return;
      }

      Property propGui = listGuiProp[0];

      // Get the user field (Edit) child property under GUI property
   Property[] listEditProp
=propGui.getChildrenWithName(Property.PROP_GUI_EDIT);
      if (listEditProp == null || listEditProp.length != 1) {
          System.out.println("No valid GUI edit property!");
          return;
      }

      Property propGuiEdit = listEditProp[0];

      // Get the list of user fields from save dialog
      // all children items under GUI edit property
      Property [] itemsEdit = propGuiEdit.getChildren();

      // ToDo: Use the list of edit items (GUI element) to construct a
      // save dialog to get user input for properties under PROP_GUI_EDIT.
      // Assume the input for attribute "CSI_DocName" we got from the dialog
      // is "myMarkup" and the input for attribute "CSI_MarkupType" is
      // "Normal", now the following code using the inputs to construct
      // the markup property contains these two attributes. In reality
      // there can be more than two attributes.

      Property [] listProp = {
          new Property("CSI_DocName", "myMarkup"),
          new Property("CSI_MarkupType", "Normal")
      };

      // Create a Markup property with the specified name & type properties
      Property propMarkup = new Property(Property.PROP_MARKUP, listProp);

      // Save the Markup
      try {
          ByteArrayOutputStream os = new ByteArrayOutputStream();
          m_vueBean.getMarkupBean().writeMarkup(os);
          m_vueBean.writeMarkup(propMarkup, os);
      } catch (MarkupIOException e) {
          System.out.println("Markup IO Exception!");
      }
  }
}
```

Adding a Markup Listener to Your Application

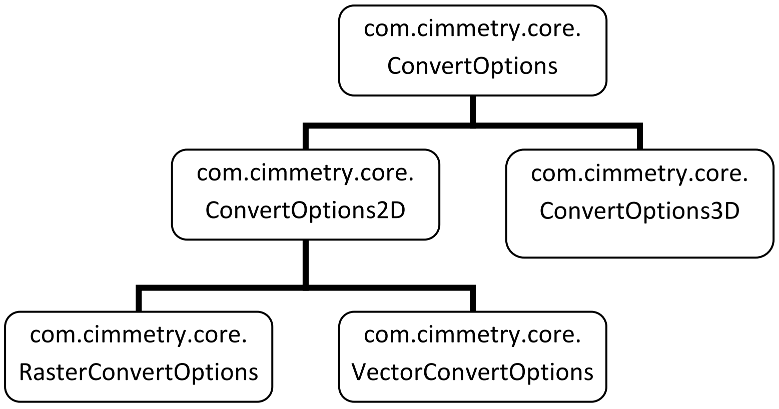
```
MarkupBean.getMarkupBroadcaster().addMarkupEventListener(MarkupEventListener
mel);
```

A Markup listener listens for Markup events related to creating/saving/deleting Markups, Markup entities, Markup file information, fonts, Markup status, and so on. Note that you must implement the `com.cimmetry.MarkupBean.event.MarkupEventListener` interface (thereby implementing the `onMarkupEvent` method).

Converting Files

The following sections discuss how to execute common Conversion actions such as making a call to convert, converting an image to a JPEG using a custom conversion, and converting a vector file to a PDF. In some cases, there are additional methods to achieve the same functionality. Refer to the *VueBean Javadocs* for more information.

The class hierarchy for conversion is as follows:



Note that the classes represent the format which you are converting a file to. For example, if you are converting to a vector format, you should define a `VectorConvertOptions` and pass it into the conversion method.

Calling to Convert

```
com.cimmetry.vuebean.VueBean.convert(ConvertOptions opts)
```

or

```
com.cimmetry.jvue.JVue.convertFile(ConvertOptions opts)
```

Once the convert options are defined, you must call one of the methods to convert.

Note: When making a call from the VueBean you must call `VueBean.convert`. When making a call from the AutoVue applet layer, you must call `JVue.convertFile.`

Converting to JPEG (Custom Conversion)

To convert an image to a JPEG, you must use the `encode()` method that Java provides as part of the `com.sun.image.coded.jpeg.JPEGImageEncoder` interface. This method encodes buffers of the image data in JPEG data streams. To use this interface, you must provide the image data in raster format or a `BufferedImage`. The following example illustrates how to use this interface with the AutoVue API:

```
import java.io.*;
import java.awt.*;
import java.awt.image.*
import com.cimmetry.core.*;
import com.sun.image.codec.jpeg.*;
…

double scaling=0.5;
BufferedImage bi = new BufferedImage((int)( m_vueBean.getWidth()*scaling),
(int)(m_vueBean.getHeight()*scaling), BufferedImage.TYPE_INT_RGB);

//Create or get Graphics and RenderOptions object here
Graphics2D g = bi.createGraphics();
RenderOptions optsRender = new RenderOptions();

// TODO: Initialize the Graphics object and RenderOptions object properly
such //       as setting the source and destination.

try {
   m_vueBean.renderOntoGraphics(g,optsRender);

   FileOutputStream out = new FileOutputStream("c:\\temp\\my.jpeg");
   JPEGImageEncoder encoder = JPEGCodec.createJPEGEncoder(out);
   JPEGEncodeParam param = encoder.getDefaultJPEGEncodeParam(bi);

   // TODO: Use the JPEGEncodeParam Interface to set the encoder parameters.

   encoder.encode(bi, param);
   out.flush();
   out.close();
} catch (Exception e) {
   System.out.println("Exception while converting to JPEG ");
   return;
}
…
```

Converting to PDF

To convert a vector file to a PDF you must perform the following steps:

- Create new `VectorConvertOptions()` object
- Set all appropriate convert options
- Call `VueBean.convert` and pass in the convert options

The following `convertToPDF()` method converts a vector file to a PDF.

```
public void convertToPDF() {

      VectorConvertOptions opts = new VectorConvertOptions();

      opts.setStepsPerInch(1);
      PAN_CtlFileInfo fi = m_vueBean.getFileInfo();
      PAN_CtlRange ps = m_vueBean.getPageSizeEx();


      if (fi.getType() == fi.PAN_DocumentFile) {
            ps = fi.getPageSize();
        }

      opts.setInputRange(ps);
      opts.setArea(ConvertOptions2D.AREA_EXTENTS);
      opts.setScaleFactor(100);
      opts.setScaleType(ConvertOptions2D.TYPE_SCALE);
      opts.setWidth(Math.abs(ps.width()));
      opts.setHeight(Math.abs(ps.height()));
      opts.setUnits(Constants.UNITS_INCH);
      opts.setPages(ConvertOptions2D.PAGES_ALL);
      opts.setFromPage(1);
      opts.setToPage(fi.getPagesNumber());
      opts.setFormat("PCVC_PDF");
      opts.setSubFormatID(0);
      opts.setFileName("c:\\output.pdf");

      Property[] p = m_ vueBean.uploadMarkups();
      //Uploads all currently loaded markups to the AutoVue server

      opts.setProperties(p);
      m_ vueBean.convert(opts);
}
```

Printing a File to 11x17 Paper

The following code prints a file to 11x17 paper size using the `com.cimmerty.common.PrintProperties` and `com.cimmetry.commonPrintOptions` classes.

```
import com.cimmetry.common.PrintProperties;
import com.cimmetry.common.PrintOptions;

public void printFile()
       {
       PrintProperties paramPrintProperties = new PrintProperties();
       PrintOptions po = new PrintOptions();

       po.setPrinter("AutoVue Document Converter");
       po.setPaperSize(po.PAPER_11X17);
       paramPrintProperties.setOptions(po);

       // The second parameter will enable the bypass of the Windows dialog
       m_JVue.printFile(paramPrintProperties, true);
       }
```

Monitoring Event Notifications

`com.cimmetry.vuebean.event`

If you have a requirement to programmatically execute specific file actions (such as rotation, zooming, and so on) as soon as a file has finished loading, you must monitor for the appropriate event notifications. If you do not check for file load completion, you might call a file action too early which may lead to errors.

The VueBean includes a set of notifications known as VueEvents. You can set up a listener to catch VueEvents, and catch the specific events that represent the completion of a file loading. In order to catch file loading completion, you must use a file listener, with the VueFileListener interface.

The steps are as follows:

1. Implement your own `VueFileListener` (for example, in a second applet).
2. In the `onFileEvent` method, check for occurrence of the `Vue.Event.ONSHOWINGFILE` event directly followed by the `VueEvent.ONACTIVEVIEW` event.
   Note: These two events in this particular order indicate that VueBean has finished loading a file.
3. Implement your code to be executed when these two events are detected in order.
4. Add your file listener to the VueBean.
5. Add this to your second applet. See Implementing Functions from AutoVue in a Second Applet.

Retrieving the Dimension and Units of a File

The following sample code shows how to get the dimensions and units of a file.

```
PAN_CtlDimensions pctlDim = m_vueBean.getFileInfo().getDimensions();
double width = pctlDim.getWidth();
double height = pctlDim.getHeight();
double depth = pctlDim.getDepth();
int units = m_vueBean.getFileInfo().getInsertion().units;
```

8. Cleanup Problems

Session Close

The following code is for a session close when using the VueBean. Note that when closing the session you do not need to close the document as it is done automatically by the session close call.

Note: This code is not required when using the AutoVue applet as it is done automatically by the applet's *destroy()* method.

```
//Disable restoration to speedup the shutdown
getServerControl().setRestorable(false);
if (getServerControl().isSessionClosed()) {
      // Already closed
      return;

try {
      //Close session on the server
      getServerControl().sessionClose();
} catch (VueRemoteException ex) {
      //Failed to close session
}

//Close connection to the server
getServerControl().disconnect();

//Stop all the active worker threads (threadpool)
JobGroup.killRootJobGroup(VueFrame.this);
```

9. FAQ

MarkupBean

Q: How do you determine the layer that a given entity is in?
A: Get the entity's spec and then get the layer from the spec.

Q: Do I have to implement the entire text editing dialog for the Text/Leader/Note entity?
A: No. The text editing dialog is inherent to these entities.

Q: An entity spec is tied to a given entity. Why was it decided to have an entity spec tied to the MarkupBean?
A: The entity spec on the MarkupBean was designed to be a reference to the most recent spec settings. When you create a new Markup entity, it defaults much of its spec attributes to the current spec in the MarkupBean. To retrieve the most recent spec settings, you can call `MarkupBean.getMarkupEntitySpec().`
Note: The other two methods `MarkupBean.getMarkupEntitySpec(MarkupEntity ent)` and `MarkupBean.getMarkupEntityFullSpec(MarkupEntity ent)` are for when you need to get the spec of a specific entity.

Q: What is the difference between `MarkupGraphicSpec` and `MarkupEntitySpec`? Why are the specs such as `ArcSpec` subclass not derived directly from `MarkupGraphicSpec`?
A: The MarkupGraphicSpec is a top-level specification that manages visual attributes such as color, fill type, and so on. The `MarkupEntitySpec` is a top-level spec that has access to the overall structure such as the MarkupBean, Markups, layers, pages, and so on.

Q: Can you work with MarkupBean independent of VueBean?
A: In theory it is possible to instantiate and work with MarkupBean without having a VueBean. However, there are not many use cases or practical reasons where this would be valuable.

Q: Are the Markup tree and Markup toolbars from the AutoVue Applet accessible if I am building a custom application from VueBean/MarkupBean?
A: No. The UI such as toolbars and Markup tree are part of the "JVue" class. If you build your solution using the JVue class you can use or customize this UI. However, if you build your solution directly from VueBean you need to implement your own UI.

Q: Is it possible to add AutoVue markup capabilities to a third-party application?
A: Yes. There are two primary ways to add markup entities using MarkupBean:
1. With user input, using `MarkupBean.setActionMode(MarkupBean.ACTION_MODE_ADD)`
2. Programmatically, using `MarkupBean.addMarkupEntity(MarkupEntitySpec spec)`

Printing

Q: What is the purpose of `com.cimmetry.core.PrintInfo` class?
A: It is used to pass information between the client and server.

General

Q: Can I perform file type-dependent operations?
A: Yes. You can do so by using the `getFileInfo()` method. The `PAN_CtlFileInfo` object that is returned can be queried to determine file format (such as vector, raster, spreadsheet, document, archive, or a database file).

Q: Can I delete server-side Markups when using the VueBean API?
A: No. It is not currently possible to programmatically delete server-managed Markups (referenced in the `markups.map` file on the server) using the VueBean API.

10. Feedback

We appreciate your feedback, comments or suggestions. Contact us by e-mail or telephone. Let us know what you think.

For any questions regarding a particular class or method, please contact Oracle Customer Support or post your question to the My Oracle Support AutoVue Community Web site. Customer Support can answer all questions related to specific topics documented in the VueBean Javadocs.

General Inquiries:

Telephone:	+1.514.905.8400 or +1.800.363.5805
E-mail:	autovuesales_ww@oracle.com
Web Site:	http://www.oracle.com/us/products/applications/autoVue/index.html

Sales Inquiries:

Telephone:	+1.514.905. 8400 or +1.800.363.5805
E-mail:	autovuesales_ww@oracle.com

Oracle Customer Support:

Web Site:	http://www.oracle.com/support/index.html

My Oracle Support AutoVue Community:

Web Site:	https://communities.oracle.com/portal/server.pt