How to Develop Visual Paradigm Plug-in?

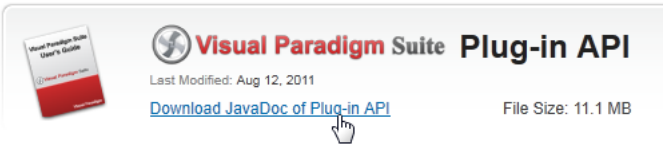
Written Date : September 1, 2011

In this tutorial, we are going to develop a plug-in that reads the flow of events of use cases and prints the collected information to an HTML file. We will use Eclipse as the Java IDE for implementing the plug-in. Therefore, please get Eclipse ready. You can, however, use any other Java IDE you prefer.

Getting the Plug-in JavaDoc

The Plug-in JavaDoc provides Visual Paradigm plug-in developers with definitions of classes, attributes, operations, and arguments in the plug-in. We recommend users getting the JavaDoc and incorporating it into the Java project that will be created in the next section. To get the plug-in JavaDoc:

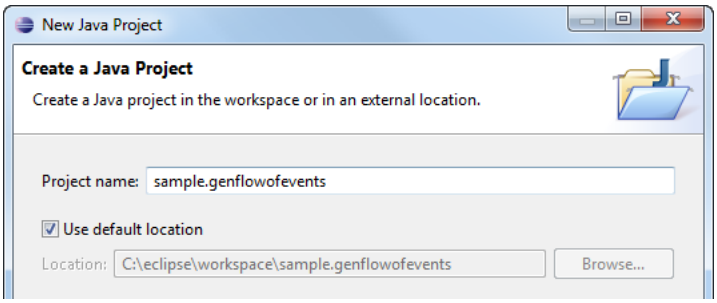
1. Visit the web page Plug-in API on the Visual Paradigm official site.
2. Click on the link **Download JavaDoc of Plug-in API**.



3. Save the zip file to a location on your machine.
4. Extract the zip file.

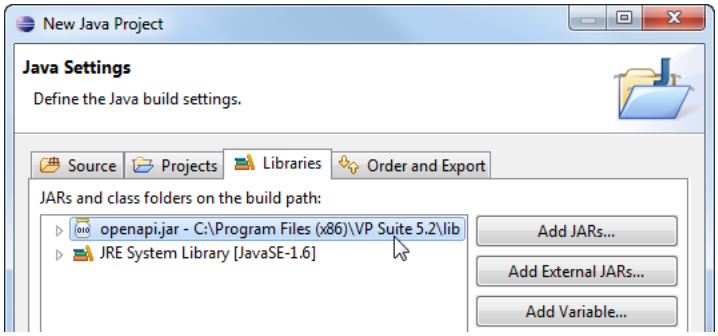
Implementing the Plug-in

1. Start Eclipse.
2. Create a Java project *sample.genflowofevents*.

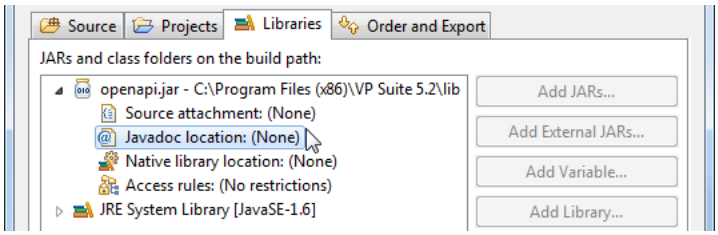


3. In the **Java Settings** page, set *src* to be the source folder and *classes* to be the class folder.

4. Open the **Libraries** tab. The jar file **openapi.jar** in **%VISUAL-PARADIGM-INSTALL-DIR% \lib** contains the API for plug-in development. Add it as a library, or else your work cannot be compiled.

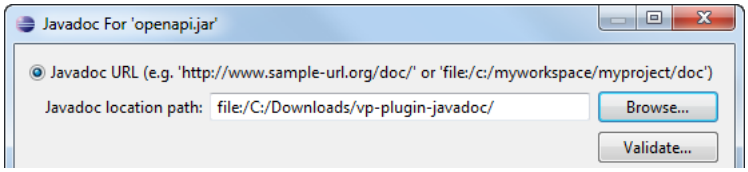


5. Expand the node **openapi.jar**. Select **Javadoc location**.



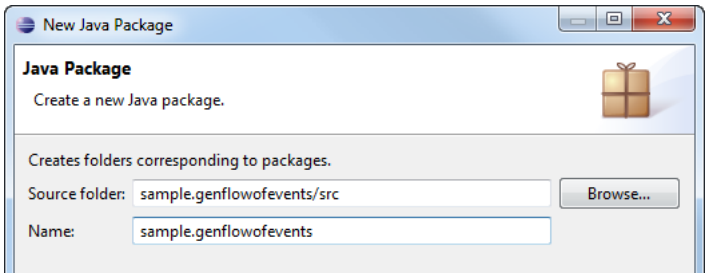
6. Click **Edit** on the right-hand side.

7. Click **Browse** in the popup window. Input the path of the extracted zip of JavaDoc. Click **OK**.

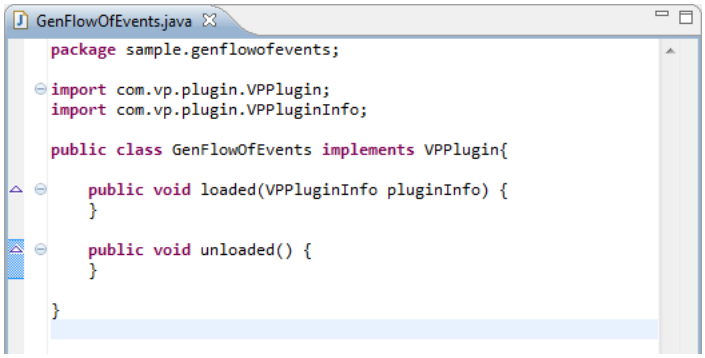


8. Click **Finish** in the **New Java Project** window.

9. Create a package *sample.genflowofevents*

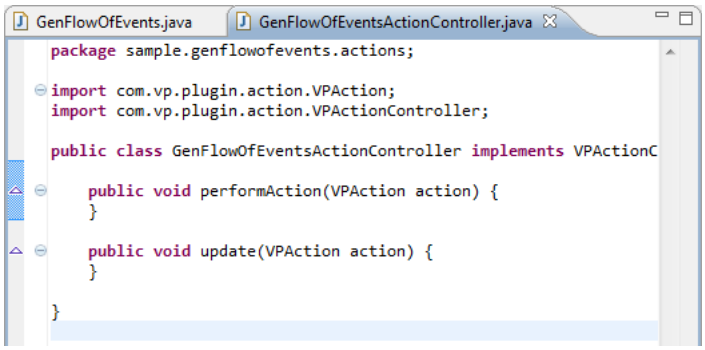


10. In package *sample.genflowofevents*, create a class *GenFlowOfEvents* which implements interface *VPPlugin*. If you do this by Eclipse's code completion feature, two methods `loaded` and `unloaded` will be added automatically. If not, add them yourself. If the editor cannot identify the interface *VPPlugin*, make sure you have added the **openapi.jar** to the project build path.



11. Create package *sample.genflowofevents.actions*.

12. In the package *sample.genflowofevents.actions*, create a class *GenFlowOfEventsActionController* which implements the interface *VPActionController*. This class is made for performing a certain action. In this case, it implements the logic of generating a flow of events to an HTML report. The action can be triggered from the user interface.



Here are some of the points you should be aware of:

1. The *performAction()* method contains the logic of the action you want to perform.

2. The *update()* method is called before showing the action on screen. You can make use of this method to disable the method before it gets shown. In this tutorial, we just leave it empty.

13. Let's fill in the method *performAction()*. First, we need to prompt the user for the destination HTML path. Enter the following code inside *performAction():*

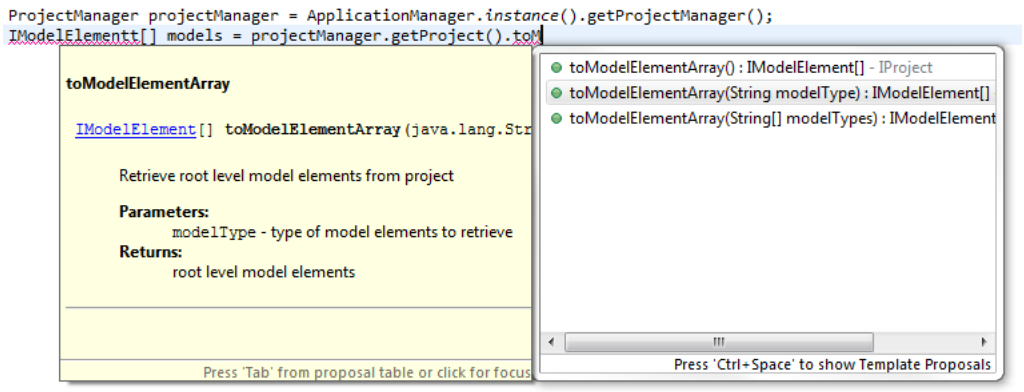
```
// get the view manager and the parent component for the modal dialog.
ViewManager viewManager = ApplicationManager.instance().getViewManager();
Component parentFrame = viewManager.getRootFrame();
 // popup a file chooser for choosing the output file
JFileChooser fileChooser = viewManager.createJFileChooser();
fileChooser.setFileFilter(new FileFilter() {

  public String getDescription() {
    return "*.htm";
  }

  public boolean accept(File file) {
    return file.isDirectory() || file.getName().toLowerCase().endsWith(".htm");
  }

});
fileChooser.showSaveDialog(parentFrame);
File file = fileChooser.getSelectedFile();
```

As we have incorporated the JavaDoc, we can read the description of classes and methods when using code completion.



Here are some of the points you should be aware of:

1. *ViewManager* is responsible for working with the user interface. For example, you may call it to print data to the message pane.
2. Instead of initiating a File Chooser directly, we recommend you use *ViewManager.createJFileChooser* instead.

14. After we have prompted the user for the path of the HTML, we can proceed with retrieving the flow of events of use cases and generating the data to HTML. Fill in the remaining part of the performAction() method.

```
if (file!=null && !file.isDirectory()) {
  String htmlContent = "";
  String result = "";

  // Retrieve all use cases from project
  ProjectManager projectManager =
ApplicationManager.instance().getProjectManager();
  IModelElement[] models =
projectManager.getProject().toModelElementArray(IModelElementFactory.MODEL_TYPE_USE_CASE);
  // Retrieve an HTML string of flow of events info from every use case
  for (int i = 0; i < models.length; i++) {
    IModelElement modelElement = models[i];
    IUseCase useCase = (IUseCase)modelElement;
    htmlContent += generate(useCase);
  }

  // write to file
  try {
    FileWriter writer = new FileWriter(fileChooser.getSelectedFile());
    System.out.println(file.getAbsolutePath());
    writer.write(htmlContent);
    writer.close();
    result = "Success! HTML generated to "+file.getAbsolutePath();
  } catch (IOException e) {
  }

  // show the generation result
  viewManager.showMessageDialog(parentFrame, result);
}
```

Here are some of the points you should be aware of:

1. *ProjectManager.getProject().toModelElementArray(IModelElementFactory.MODEL_TYPE_USE_CASE)* enables you to get an array of a specific type of model element, such as a use case.
2. *IModelElementFactory* contains constants of element type names.
3. *IModelElement* refers to a model element in a project, such as a use case, an actor, etc.
4. You may type-cast an *IModelElement* into a concrete type class, such as *IUseCase*, *IActor*, etc.

15. Add and implement another method *generate()*, which takes *IUseCase* as a parameter, prepares the HTML string, and returns the result back to the method that calls it.

```
public String generate(IUseCase useCase) {
  String content = "";
```

```
  // Retrieve flow of events sets from use case. Each IStepContainer is a set
of flow of
events
  IStepContainer[] stepContainers = useCase.toStepContainerArray();
  for (int i = 0; i < stepContainers.length; i++) {
    IStepContainer stepContainer = stepContainers[i];

    // Print the name of use case and flow of events to HTML string
    content += "<table border=\"1\" width=\"500\"><tr><th>" +
useCase.getName()
+ " - " + stepContainer.getName() + "</th></tr>";

    // Print the flow of events content to HTML string
    IStep[] stepArray = stepContainer.toStepArray();
    for (int j = 0; j < stepArray.length; j++) {
      IStep step = stepArray[j];
      content += "<tr><td>"+ (j+1) + ". " + step.getName()+"</td></tr>";
    }
    content += "</table><br>";
  }

  return content;

}
```

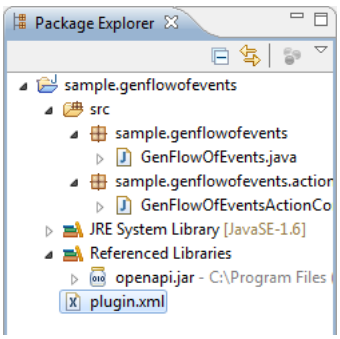
Here are some of the points you should be aware of:

1. *IStepContainer* refers to a set of flow of events. Each use case can have zero or more than one set of flow of events.
2. *IStep* refers to a step (i.e., a row) inside a flow of events set.

16. The coding part is done. If you find the code cannot be compiled, please make sure you have added **openapi.jar** into the build path.

Deploying the Plug-in

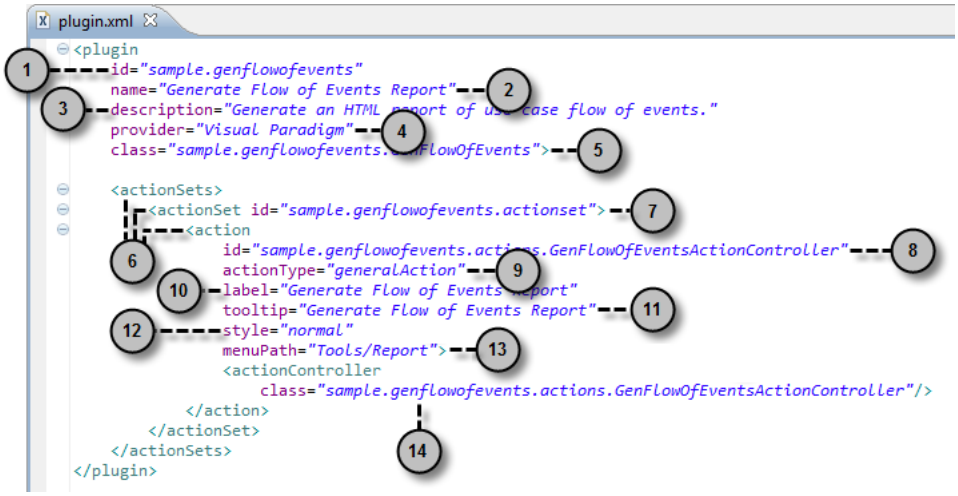
1. Create an XML file **plugin.xml** under the project root.



2. Fill in the XML file as shown below:

```
<plugin
  id="sample.genflowofevents"
  name="Generate Flow of Events Report"
  description="Generate an HTML report of use case flow of events."
  provider="Visual Paradigm"
  class="sample.genflowofevents.GenFlowOfEvents">
  <actionSets>
    <actionSet id="sample.genflowofevents.actionset">
      <action
        id="sample.genflowofevents.actions.GenFlowOfEventsActionController"
        actionType="generalAction"
        label="Generate Flow of Events Report"
        tooltip="Generate Flow of Events Report"
        style="normal"
        menuPath="Tools/Report">
        <actionController
class="sample.genflowofevents.actions.GenFlowOfEventsActionController"/>
      </action>
    </actionSet>
  </actionSets></plugin>
```

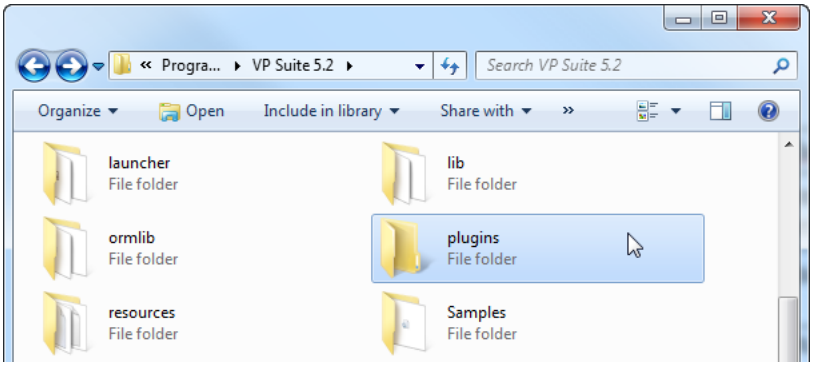
Here is a description of the content of plugin.xml:



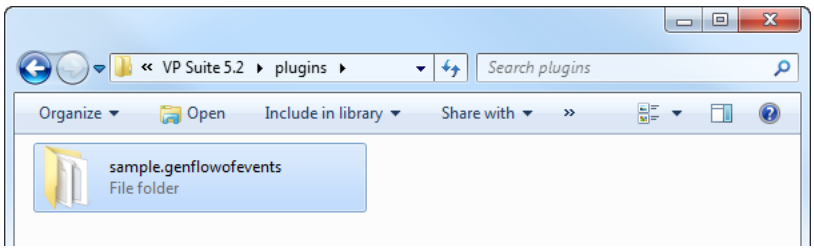
No.	Description
1	A text for identifying this plug-in. You may develop multiple plug-ins. Each plug-in must have a unique ID defined.
2	The name of the plug-in.
3	The description of the plug-in.
4	The person/organization who developed this plug-in.

5	The fully qualified class name of the plug-in class, which is the class that implements the VPPlugin class, offering the load and unload logic.
6	Action sets describe the possible ways of invoking the plug-in. An action can be added into the MenuBar and Toolbar by setting @menuPath=... and @toolbarPath=... If your plugin has actions on the menu bar and the diagram's popup menu: for menu bar's actions, the actions are defined in <actionSet>; for toolbar's actions, the actions are defined in <contextSensitiveActionSet>.
7	A unique value for identifying the action set.
8	A unique value for identifying the action.
9	Define the type of action: [generalAction | shapeAction | connectorAction]. generalAction: The simple action that may be added on the menu bar/toolbar/diagram popup menu. You are required to specify the <actionController...> for this action (refer to point 14). shapeAction: The action used to create a CustomShape. You are required to specify the <shapeCreatorInfo...>. connectorAction: The action used to create a CustomConnector. You are required to specify a set of <connectionRule...>.
10	The menu item text.
11	Tooltip is the text that displays when moving the mouse pointer over a toolbar button.
12	Specify whether the menu item is a Normal button or a Toggle (Checkbox) button. [normal | toggle]
13	The position where the menu/toolbar button will be presented. In this case, the menu will be placed under the Tools menu, after the Report menu.
14	The fully qualified class name of the class that implements VPActionController, for showing what and how to perform the action.

3. It's time to deploy the plug-in into your Visual Paradigm installation to make it available in Visual Paradigm. Create a folder `plugins` in the Visual Paradigm installation folder (e.g., *C:\Program Files\Visual Paradigm\plugins*).



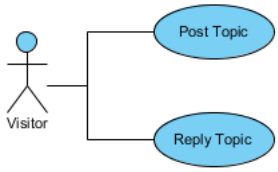
4. Copy the Java project folder *sample.genflowofevents* into the **plugins** folder. You should obtain a folder structure like this: *C:\Program Files\Visual Paradigm\plugins\sample.genflowofevents \classes*.



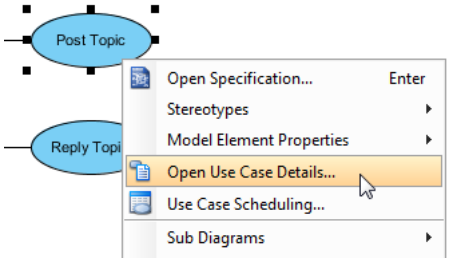
Running the Plug-in

1. Create a new project in Visual Paradigm by selecting **Project > New** from the application toolbar. In the **New Project** window, enter *Test Plug-in* as the project name and click **Create Blank Project**.

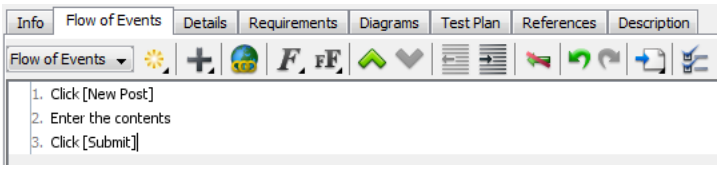
2. Draw a simple use case diagram with 2 use cases *Post Topic* and *Reply Topic*, like this:



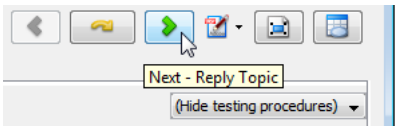
3. Right-click on the use case *Post Topic* and select **Open Use Case Details...** from the pop-up menu.



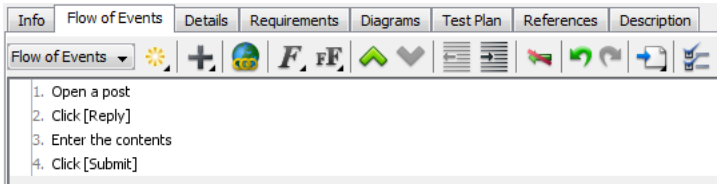
4. Open the **Flow of Events** tab. Enter the steps: *1. Click [New Post], 2. Enter the contents, 3. Click [Submit]*



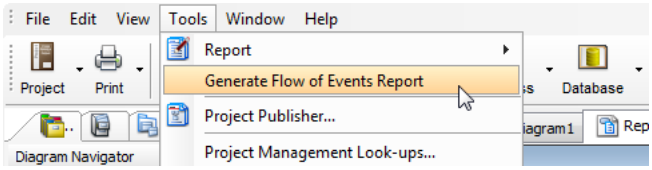
5. At the top right of the flow of events, click the **Next** button to move to the use case *Reply Topic*.



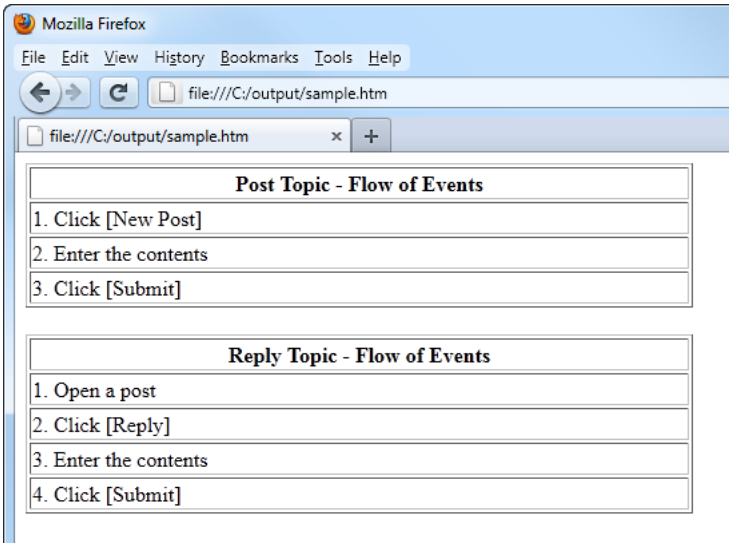
6. Enter the flow of events for use case *Reply Topic: 1. Open a post, 2. Click [Reply], 3. Enter the contents, 4. Click [Submit]*



7. Let's test our plug-in. Select **Tools > Generate Flow of Events Report** from the main menu.



8. You are prompted for the output path of the HTML. Specify the file path in the **Save** window and click **Save**. The HTML file will be saved to the specified location.



Resources

1. Java Project (folder) of this plug-in

Related Links

- Debug your plug-in with Eclipse
- Visual Paradigm Plug-in User's Guide
- Visual Paradigm Plug-in Sample

Visual Paradigm home page
(https://www.visual-paradigm.com/)

Visual Paradigm tutorials
(https://www.visual-paradigm.com/tutorials/)