# How to design a Tabbed Properties View using the XA Common Diagram Properties framework™ 1.1

***delivery date: 2007/10/26***

Atos Worldline
Architecture & Methodology

| **Revision History** | |
|---|---|
| Revision 1.0 | 2007/10/02 |
| Initial release | |
| Revision 1.1 | 2007/10/26 |
| Added contents | |

**Table of Contents**



# Presentation

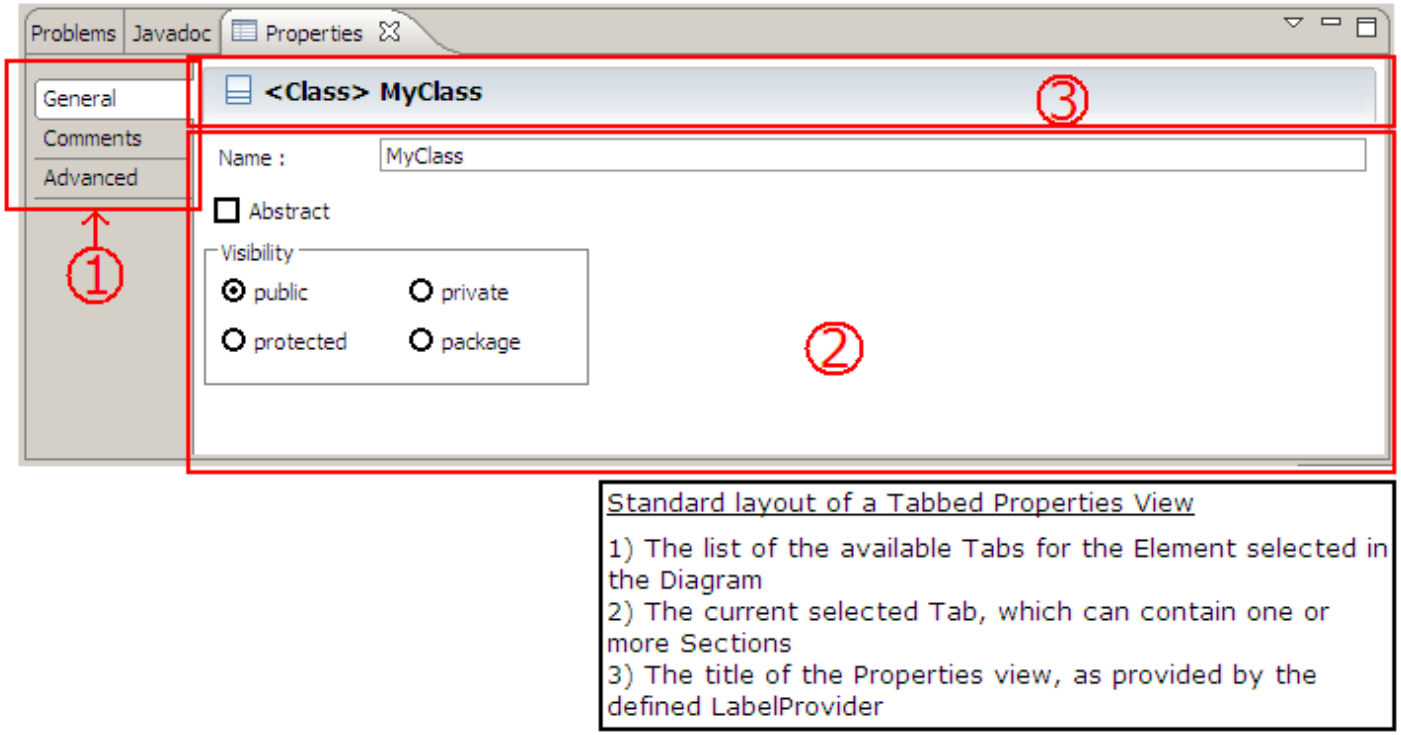
The XA Common Diagram Properties is an API which role is to provide tools, in order to ease the implementation of a Tabbed Properties View associated with

GMF Diagrams. This document will explain in details the fonctionalities provided by this tool, and how to use them.

In order to create a Tabbed Properties View, in a more general case, you have to deal with two kind of resources. The first kind of resources is the Java Classes you will implement, which will define all the graphical components, and the other one, is the `plugin.xml` file.

The `plugin.xml`
is as important as the JAVA Classes, because this resource will define the relationship between the graphical elements, and the Eclipse IDE. So, this document will explain first how to populate the `plugin.xml`
file in order to initialize the Tabbed Properties View, and then, how to use the API in order to fill the Tabbed Properties View with your graphical components.

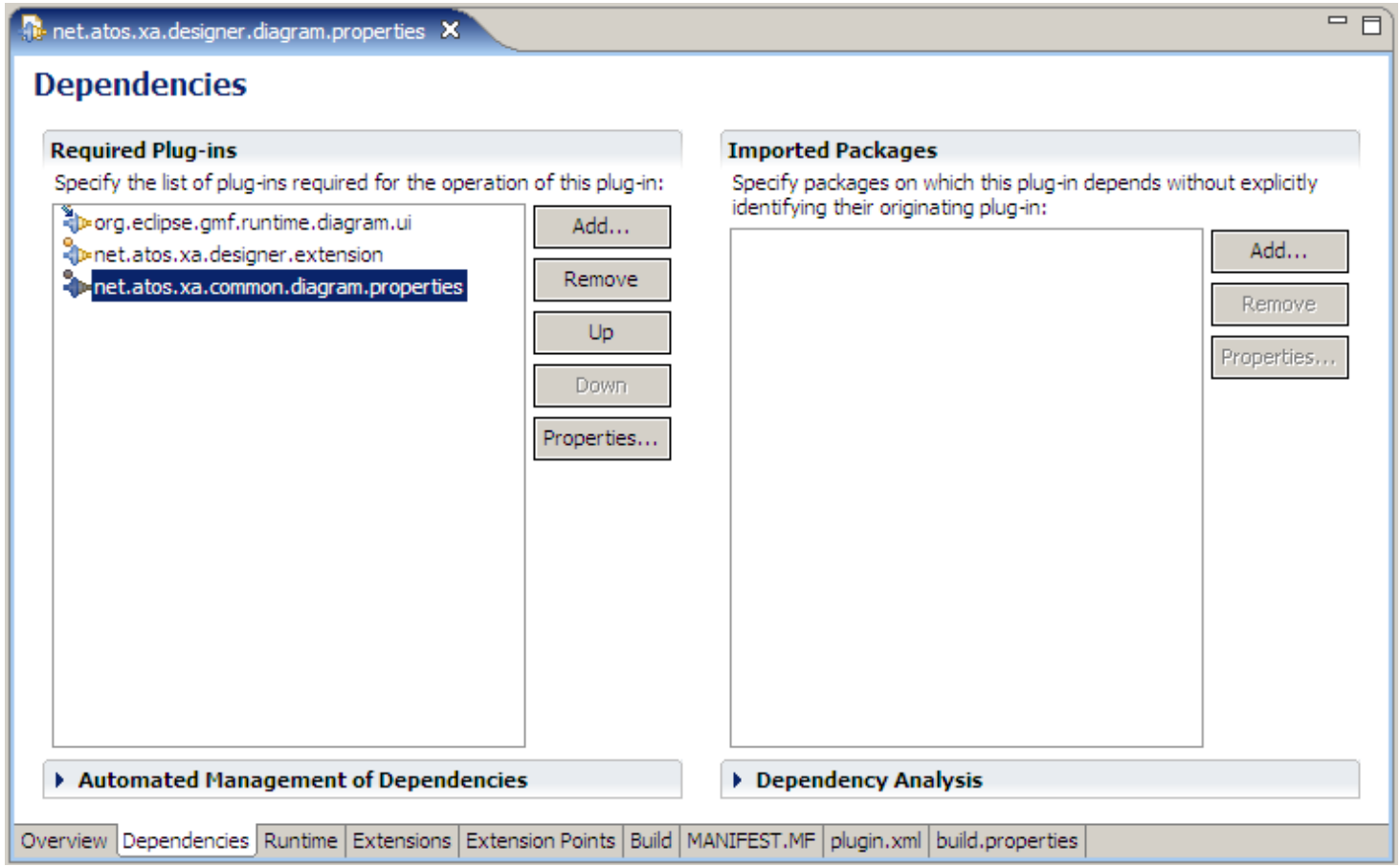
The following image shows the default structure of a Tabbed Properties view. Some of the terms will be explained later.




# Defining an environment.

First of all, you have to create a new Eclipse Plugin, or select a Plugin which will contain all your implementation of a Tabbed Properties View. This plugin

will need the `net.atos.xa.common.diagram.properties` plugin in its classpath (which contains the XA Common Diagram Properties Framework API). Once the plugin is defined, and the classpath is fixed, you are ready to implement your Tabbed Properties View.




# Populating the plugin.xml file

As said before, the `plugin.xml` resource will contain the binding between the graphical objects, and the Eclipse IDE. This binding is defined in the *Extensions* tab.

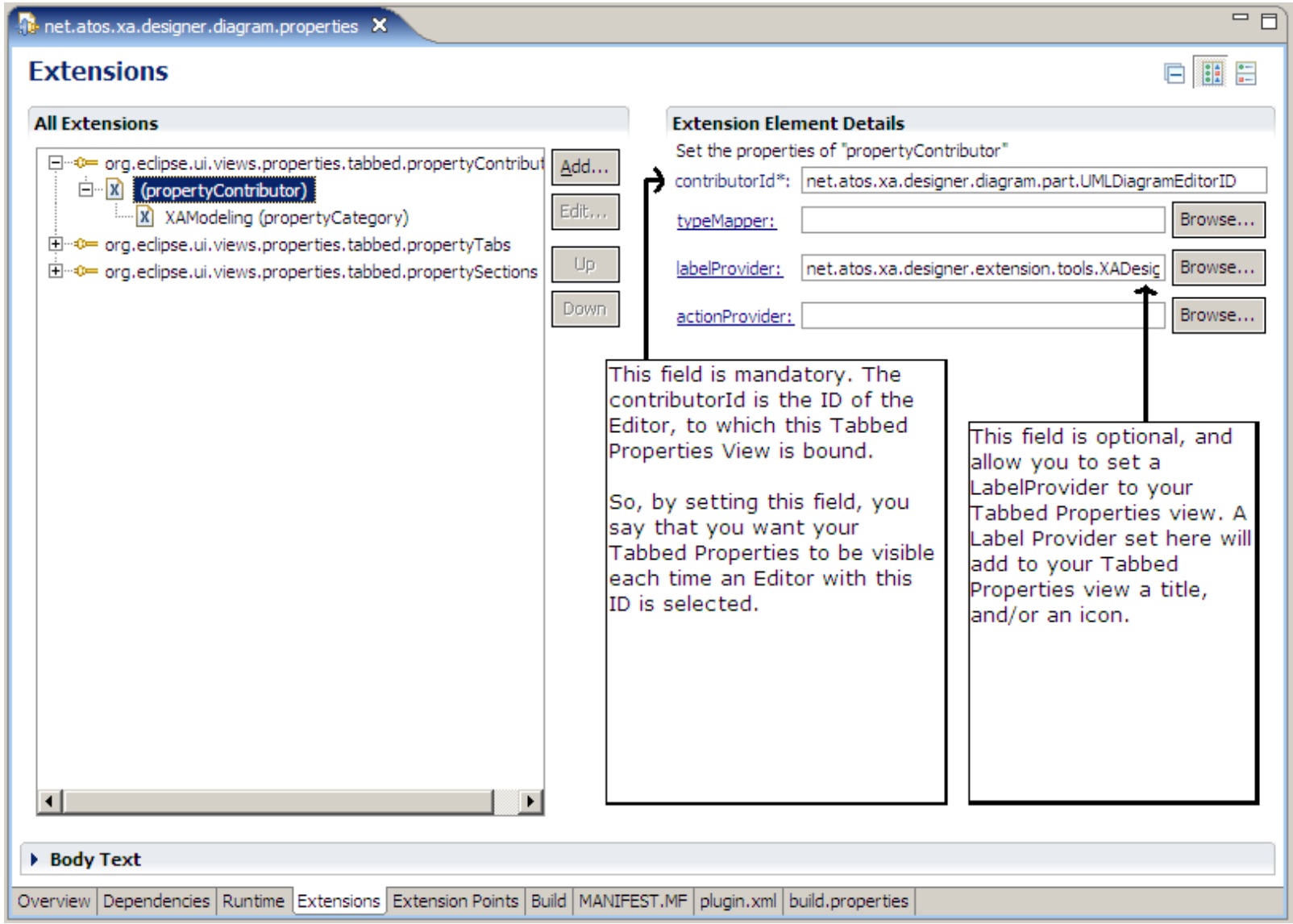
This binding process contains 3 different steps, which are the definition of the 3 Tabbed Properties Extension Points. (See Eclipse tutorials on how to create Eclispe plugins, to have more information about ExtensionPoints)

Here are the names of the 3 extension points you have to create:

- `org.eclipse.ui.views.properties.tabbed.propertyContributor`
- `org.eclipse.ui.views.properties.tabbed.propertyTabs`
- `org.eclipse.ui.views.properties.tabbed.propertySections`

## Defining the PropertyContributor Extension Point

This Extension Point, which full name is `org.eclipse.ui.views.properties.tabbed.propertyContributor` , defines the relationship between your Tabbed Properties View and a particular type of Editor. The screenshot below explains the meaning of the Extension Points parameters, and how they should be filled.




NOTE : In order to create the `PropertyContributor` from the Extension Point, or the `PropertyCategory` from the `PropertyContributor` , right click on the element, and pick the right one in the *new* menu.

In case of GMF Diagrams, the `contributorID` can be found in the GMF generated source code, in the class named `XXXDiagramEditor` from the package `xxx.part` (It is the value of the generated `ID` field.

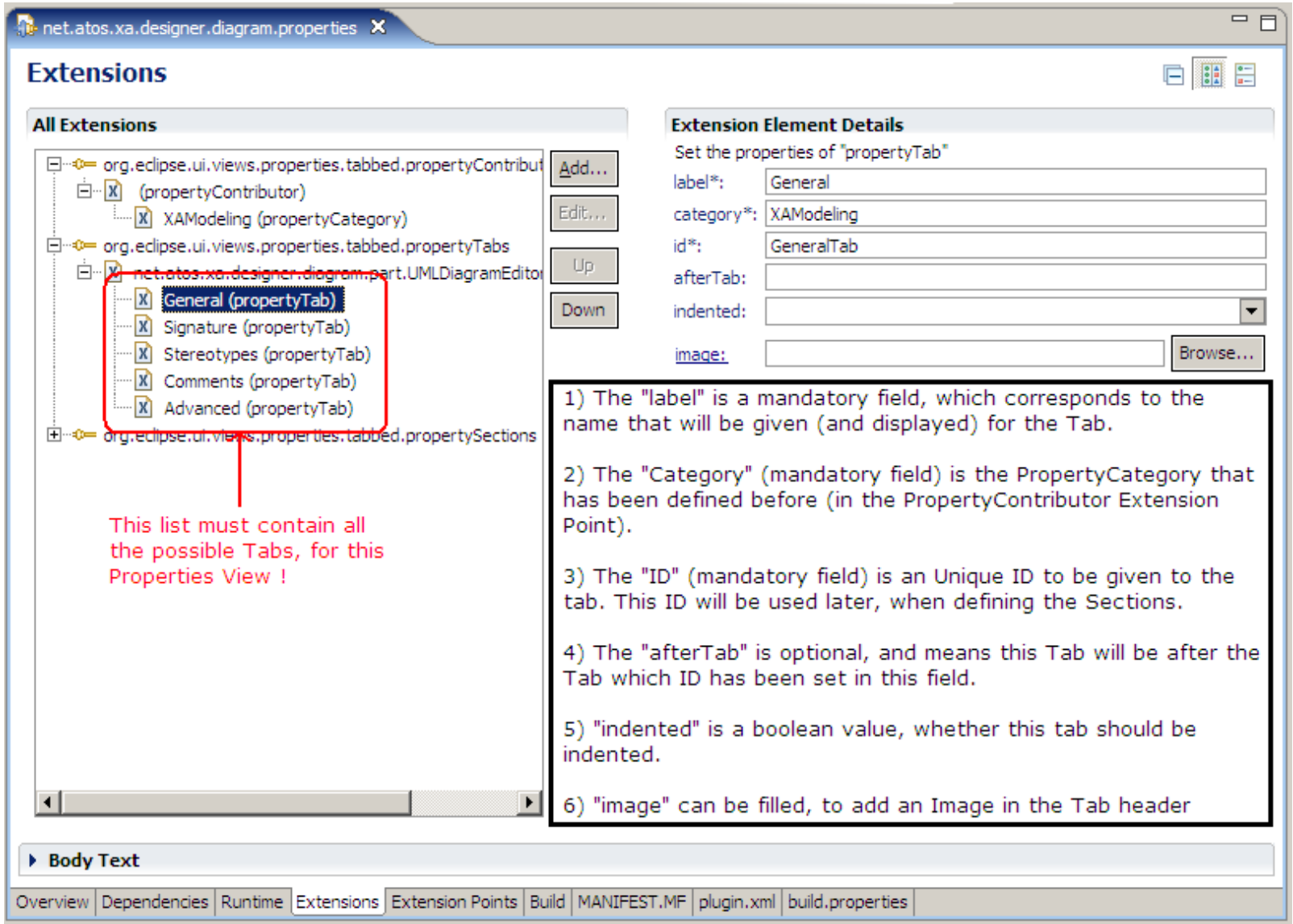
Concerning the `propertyCategory` (on the screenshot, under the `propertyContributor` item in the tree view), its only field to populate is a name of a category. You can define this name as you want, and it will be used later in the other Extension Points.

## Defining the PropertyTabs Extension Point

This Extension Point, which full name is `org.eclipse.ui.views.properties.tabbed.propertyTabs` , will contain the list of the Tabs that will be present in the Tabbed Properties View.

A Tab must contain a Graphical Element (named **Section** , see next paragraph) to be visible. By the way, all the `Tabs` won't automatically be visible on the Tabbed Properties View at the same time. This means that for a Selected Element on a Diagram, if there is no `Section` to be displayed for a `Tab` , this `Tab` won't appear

Such as the `PropertyContributor` for the previous Extension point, you have to create a new `PropertyTabs` item, which parameter will be the `contributorId` of the Editor (see previous Extension Point), and under this `PropertyTabs` item, you have to create all the `PropertyTab` items (one per `Tab` ), and populate the fields as shown on the screenshot below.



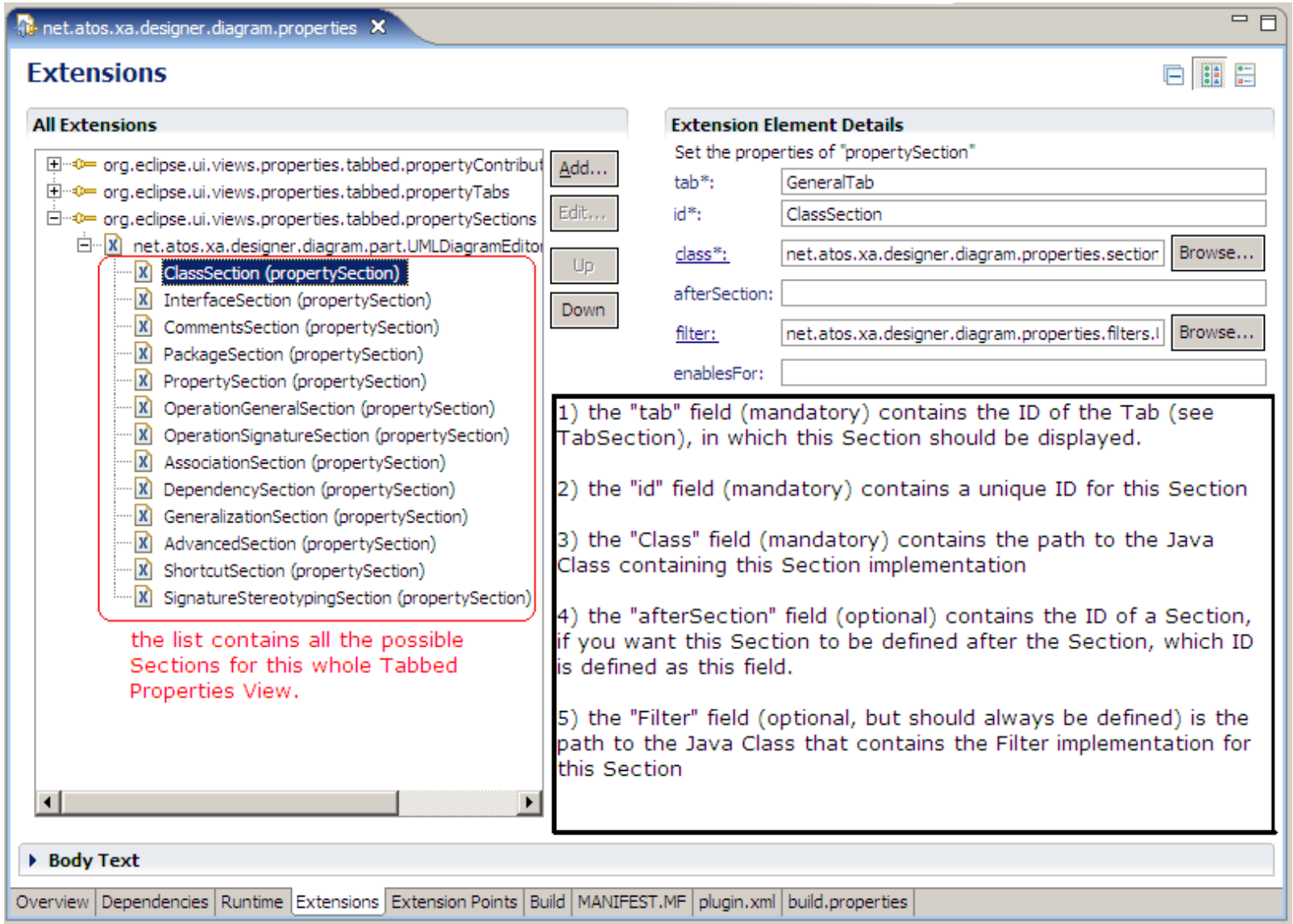

## Defining the PropertySections Extension Point

This extension point, which full name is `org.eclipse.ui.views.properties.tabbed.propertySections` defines all the Graphical Elements (named **Sections** ) for a `Tab`.

Like before, the Extension point contains a `PropertySections` item, which will contain all the possible `PropertySection` that can be displayed in all `Tabs` .

A PropertySection (or `Section` ) is closely bound to a `Filter` . This `Filter`
(which is a Java Class, which will be explained in details in the API part) tells, according to the Object selected on the Diagram, if a section has to be displayed or not. So, even if all the `Sections` have to be defined here, the `Filter` will tell if the `Section` is visible or not. And, as said before, if no `Section` is visible for a `Tab` , this `Tab` won't be visible. Moreover, more that one `Section` can be displayed for a Tab at the same time.

The screenshot below explains the role of each useful parameter for the `PropertySection` item.




# Using the XA Common Diagram Properties API

## Introduction

The XA Common Diagram Properties API role is to provide tools to ease the JAVA implementation that stands behind a Tabbed Properties View. The JAVA

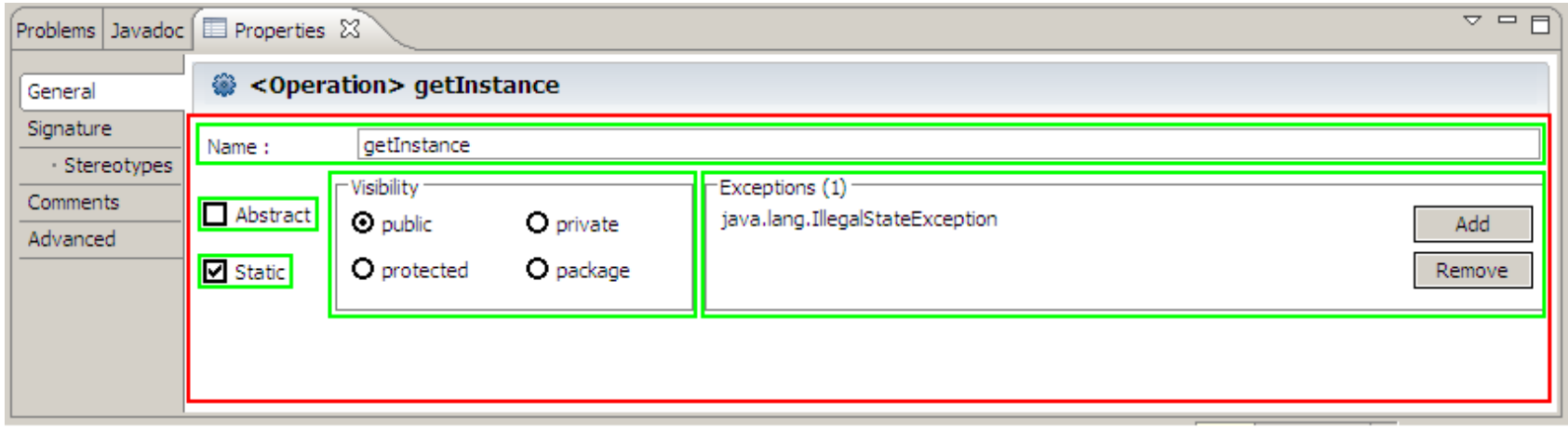
Implementation, as seen before, consists in the implementation of the `Sections` and the `Filters` .

Moreover, in order to ease the implementation of pure graphic parts, the `Section` has been divided into more smaller and specific groups, the `Zones` . Normally, the `Sections` do not contain any technical implementation, but just the definition of the `Zones` , while those `Zones` contain the technical implementation. The `Zones`
are most oftenly defined in order to manage only some properties of the selected Object on the Diagram (and not the whole objects, to make the job less tedious). The `Zones` visibility condition is, as the `Section` , defined by the filter, and indirectly, by the selected element of the Diagram.

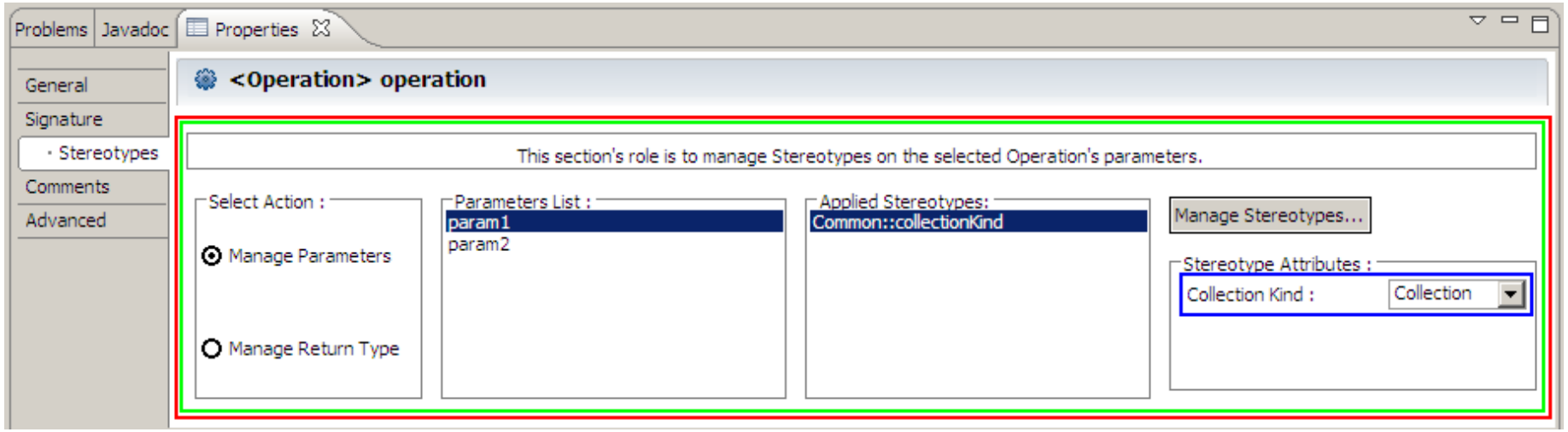
Finally, for graphical parts, which visibility is defined by some attributes of the selected element of the Diagram, the `Item` object has been defined. The `Item` objects should be defined in the `Zones` . The particularity of the `Items`
is to appear or disappear, depending on some properties of the Selected Object, and not only on the Type of the Selected Object.

More than that, this API provides `ChangeHelpers` . Those `ChangeHelpers`
are in fact advanced Listeners, in order to catch the "Human" interaction with the user, and spread the information.

The following screenshots give examples of properties view in XA Designer, with the different Graphical Elements.




On the picture above, there is one `Section` (in red), in which are defined 5 `Zones`
(in green). As you can see, each zone manages a specific part of the selected Object (here, an Operation).




On the picture above, there is one `Section` , in which is defined only one `Zone` (There is only one `Zone` , because all elements in this `Zone` interact with each other, and interaction wouldn't be possible if there was several `Zones` , and a `Zone` is a totally independant Object). But in blue, there is an `Item` . An `Item` is defined here because this graphical part should only appear when the CollectionKind stereotype is applied to the selected object in the list (Here param1). The visibility is so defined by a property of the Object, and not the Object itself. (Here, the selection is from a the parameters list, but it would be exactly the same for the selected Element in the diagram).

## Section Object

A `Section` is defined as the main component of a Tab. It activates and deactivates according to the `Filters` . All the sections are defined in the plugin.xml file, and will be displayed as rows in the Tab.

To define a `Section` Class, just make your Class extend the `Section`
abstract Class, available in the Common Properties API. The Class inheritance will make you create and implement a default constructor, and two methods. Here is the description of the two methods to implement.

### initParts()

In this method, you will define all the sub-elements for this `Section` , i.e. the `Zones` . In order to ease the `Zones` management, some methods are given in the API. For the `Zones` creation, it is recommended to use the following syntax.

```
getZones().put("myZone",new MyZone(getBackground());
```

Here, the method `getZones()` gives you all existing `Zones` , while `myZone` will be the ID of the `Zone` you are created, while `MyZone` is the Class where the

implementation of the Zone is. (The implementation of the Zones will be described later).

**addLayoutsToParts()**

Zones don't have a predefinite layout within the Section . So this method if provided in order to define a layout between each Zones in a defined Section . In order to create layout for Sections , you have to create FormData , since a Section uses a FormLayout .

The specific syntax is as follows:

```
fData = new FormData();
fData.left = new FormAttachment(0, 5);
fData.right = new FormAttachement(100, -5);
fData.top = new FormAttachment(getZone("myOtherZone"), 5);
fData.height = 20;
getZone("myCurrentZone").setLayoutData(fData);
```

Concerning the previous code, fData is a FormData created instance you can reuse. To apply values to a FormData, see FormData javadoc, and in order to apply the FormData to the Zone, just use the getZone("myCurrentZone").setLayoutData(fData) , where "myCurrentZone" is the ID corresponding to Zone which you are applying the Layout.

## Zone Object

A Zone is a graphical element used to split a Section in smaller part. Indirectly, a Section will be displayed/undisplayed according to the Filters too. Of course, a Section can be reused in more than one Section. A Section contains SWT Objects.

To define a Zone class, you just have to make your class extend the Zone
class, provided in the Common Properties API. This will make you create and implement a default constructor and four methods. The four methods are described below.

**addItemsToZone()**

Like initParts() method for the Sections, this method is used to define all the SWT Objects to be displayed in the Properties View. It is recommented to use the getWidgetFactory() method (provided in the Zone API), in order to create the SWT elements, because the WidgetFactory applies some graphical enhancements to the default SWT Objects.

Here is some examples you can reuse, to create SWT Objects (the list is not exhaustive).

```
Group listGroup = getWidgetFactory().createGroup(getZone(), "Parameters List");
```

```
List list = getWidgetFactory().createList(this.listGroup, SWT.V_SCROLL | SWT.BORDER);
ListViewer listViewer = new ListViewer(list);

Group paramGroup = getWidgetFactory().createGroup(getZone(), "Selected Parameter details");

Label nameLabel = getWidgetFactory().createCLabel(this.paramGroup, "Name : ");
Text nameText = getWidgetFactory().createText(this.paramGroup, "", SWT.BORDER);

Button typeButton = getWidgetFactory().createButton(this.paramGroup, "Edit Type", SWT.NONE);

Button javadocButton = getWidgetFactory().createButton(this.paramGroup, "Comments", SWT.NONE);

Label multiplicityLabel = getWidgetFactory().createCLabel(this.paramGroup, "Multiplicity");
CCombo multiplicityCombo = getWidgetFactory().createCCombo(this.paramGroup, SWT.READ_ONLY | SWT.BORDER);
multiplicityCombo.add("0..1");
multiplicityCombo.add("0..n");
multiplicityCombo.add("1..1");
multiplicityCombo.add("1..n");
```

### addLayoutsToItems()

The management of Layout for the SWT Objects within a `Zone` is very similar to the management of `Zones` into a `Section`. the only difference is that you have to apply the `FormData` directly to the SWT Object, as shown by the example below :

```
fData = new FormData();
fData.left = new FormAttachment(0, 5);
fData.top = new FormAttachment(0, 5);
fData.right = new FormAttachment(25, -5);
fData.height = 13 * 6 + 5;
listGroup.setLayoutData(fData);

fData = new FormData();
fData.left = new FormAttachment(0, 5);
fData.top = new FormAttachment(0, 5);
fData.right = new FormAttachment(100, -5);
fData.bottom = new FormAttachment(100, -5);
list.setLayoutData(fData);
```

### addListenersToItems()

This method should contain the listeners to be applied to the SWT Objects, to define human interactions with the Property View. Those listeners are already defined (and enhanced) in the `ChangeHelpers` provided with the Common Properties API (`ControlChangeHelper` and `TextChangeHelper`). Each action defined in a `ChangeHelper` should call the `refreshZoneAndDiagram()` method, in order to keep the `Zone` and Diagram up-to-date. Some examples concerning

the `ChangeHelpers` will be given in the paragraph dedicated to this kind of Object

**updateItemValues()**

This method will be called when the `Zone`
and Diagram will be refreshing. So, you should define here the actions to perform i.e. refresh of SWT elements, when the refresh process is called.

Example : for a check-box, when the dedicated object owns the "static" property:

```
boolean isStatic = (Boolean) getEObject().eGet(UMLPackage.eINSTANCE.getFeature_IsStatic());
button.setSelection(isStatic);
```

## Item Object

The `Item` Object is used to define sub-parts of `Zones` (Kind of group of SWT Objects). The particularity of the `Item`, is to be able to appear and disappear regarding to some properties of the selected Object (an not regarding the type of the Object only). The `Item` should be defined in a `Zone`.

Four methods work as the `Section` methods. This methods are `initElements()`, `addLayoutToElements()`, `addListenersToElements()` and `updateElements()`. Those methods should contain the same kind of information and implementation as the `Section` object (described above).

Moreover, there is one more method, which can be overriden, named `load()`. This method allow you to override the default loading process of the graphical elements. This `load()` method should be called directly from the `Zone` containing this `Item`.

The `Item` objects also own methods to help with the `Item` visibility. Those methods are `setVisible(boolean)` and `getVisible()`.

Example of `load()` overrid method : (visibility depending on the presence of a particular sterotype on a class.)

```
public void load(Element element, Stereotype stereotype) {
        this.element = element;
        if (element == null || stereotype == null)
                setVisible(false);
        else
                setVisible(hasStereotype(element) && stereotype.getQualifiedName().equals(getStereotypeQualifiedName()));
        super.load();
}
```

## Filter Object

The `Filters` are objects to reference into the plugin.xml file. Their role is to define the visibility of the `Section`, depending on the Type of the selected Object

on the Diagram. For each `Section`, you must define a `Filter`.

To define a `Filter` class, make your class extend the `Filter` abstract class provided in the Common Properties API, and create and implement the `select(Object)` method. (The object passed as parameter will be the Object that will be currently selected in the diagram.

To help you implement the `select()`
method, some methods are provided in the Common Properties API. Here is an example. The following example Filters on UML Operations:

```
@Override
public boolean select(Object toTest) {
        return super.select(toTest, Operation.class);
}
```

Note that `super.select(Object, Class)` is one of the provided methods from the Common Properties API.

## ChangeHelper Object

The `ChangeHelpers`
are tools, which ease the management of Listeners. (To have information from human interaction, some SWT Object will need listeners, to catch a user's action.)

The common properties API will provide two `ChangeHelpers` : the `SelectionChangeHelper`, and `TestChangeHelper`. The `SelectionChangeHelper` is the helper to apply to object that can be selected, i.e. the buttons (which can be standard buttons, radio buttons ,checkbox buttons...), and the `TextChangeHelper`, which is the helper to apply to text fields.

Those Helpers are defined in the dedicated methods in `Zone` and `Item`. They may be used as follows :

```
SelectionChangeHelper selectionChangeHelper = new SelectionChangeHelper() {

        @Override
        public void buttonSelected(Control control) {
                // implement your action here
                refreshZoneAndDiagram();
        }

};
selectionChangeHelper.startListeningTo(button);
```

In this case, the `SelectionChangeHelper` is applied to a button (last line of code), and will react (the `buttonSelected()` method will be called) when the user clicks on the button.

Note that the `refreshZoneAndDiagram()` method (from the provided Common Properties API) is called, as it should always be called at the end of an action.

Here is another example, with the other Listener:

```
TextChangeHelper listener = new TextChangeHelper() {

        @Override
        public void textChanged(Control control) {
                // implement your action here
                refreshZoneAndDiagram();
        }

};
listener.startListeningTo(nameText);
listener.startListeningForEnter(nameText);
```

As previously, the `refreshZoneAndDiagram()` method is called after the action.

Here, the listening process is applied on two cases : the first one (`startListeningTo()`
method) will in fact listen to test modification when the SWT Test object will be selected or unselected. The second one (`startListeningToEnter()`)will also listen, for the selected SWT Text object, to the "Enter" key. By the way, hitting the "Enter" key when editing a SWT Text value will also run the action in the Listener.