Extensible Architecture for ConcernMapper

by:
Mohammad Usman Ahmed
Student #: 260044056

Supervisor: Martin Robillard

Overview

The project that I chose for Comp400 is directly related to an Eclipse plug-in called ConcernMapper. ConcernMapper is a plug-in that allows you to reorganize the modularity of a software system in a way that suits your needs, without altering its actual structure or behavior. It also allows you to keep a permanent record of the code associated with various concerns. A concern is a set of related methods and code-fragments of a large project that are related to a certain feature of the software but are scattered in multiple classes. These concerns can be monitored in a user-friendly view provided by the ConcernMapper plug-in where you can easily see the tree structure of the concern elements as they are found in your software code while also being able to keep together all elements that are part of one concern in the software design.

The ConcernMapper is an Eclipse plug-in for experimenting with techniques for advanced separation of Concerns. It supports software development involving scattered concerns and also paves way for new research methods in software investigation by observations of concern creation and usage during development of large software projects.

Background & Motivation

Although it provides a very good feature, there is a restriction to this useful plug-in that it only supports Java Elements such as methods and fields in the Concern Model. This means that a concern can only contain fields and methods defined in a Java Class, however many people in the industry and academia use languages other than Java and would like to have a similar tool for the language of their choice.

The problem with encoding other types of elements directly into the plug-in is that there are an unlimited number of useful types that one could want to add to a Concern Model, each with its own special features and properties and it would be infeasible to continuously hack into the code of an independently complete plug-in.

The extensible platform of Eclipse provides a plug-in architecture which suggested another solution to this problem. The existing ConcernMapper could

be changed into a generic plug-in implementing the ConcernMapper with support for a generic element node in its model. We could then write, plug-ins extending this generic ConcernMapper plug-in to specify the exact type for some new element that the Concern Model would then support. This would also allow further customization on the elements supported in the ConcernMapper depending upon the level of abstraction provided by the ConcernMapper. For the purpose of this project, this abstraction has been limited to the level where the existing functionality of the ConcernMapper can be reproduced for the existing supported types (i.e. Java Methods and Java Fields) as well as the addition of a new type unrelated to the Java Elements having complete and independent functionality of its own.

This required that the ConcernMapper host plug-in make no reference to any specific type of element throughout its code. Due to the extensive use of Java specific helper methods to provide special functionality for elements in the ConcernMapper caused by the assumption that the elements stored in the concern model are always one of two types of Java Elements, the ConcernMapper plug-in had to be re-visited throughout the code to replace the Java specific code with a general processing of all element types for each functionality at the locations found. Doing so required varying strategies depending upon the specific task being re-designed.

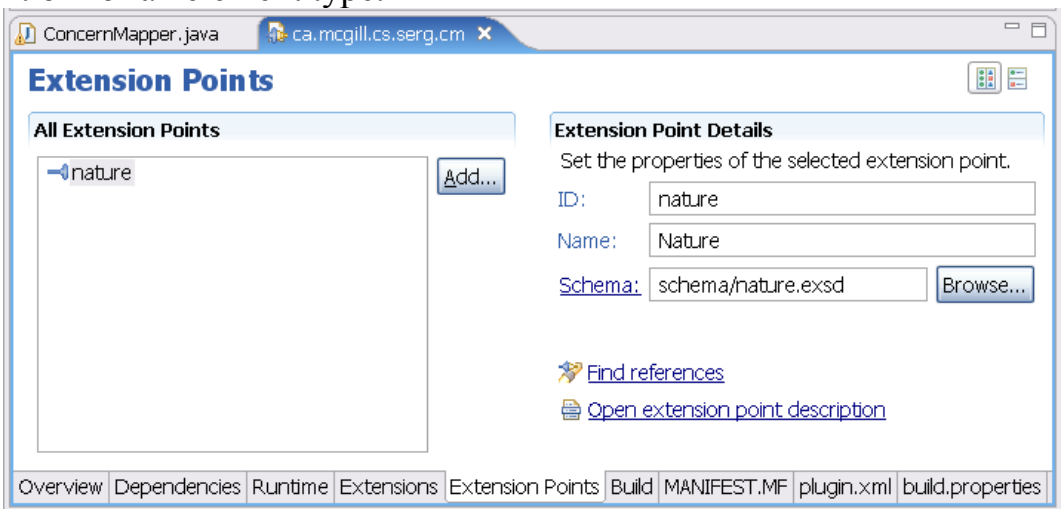
In the following pages I will be presenting the new extensible architecture of the ConcernMapper plug-in as outlined above.

The Extensible Architecture

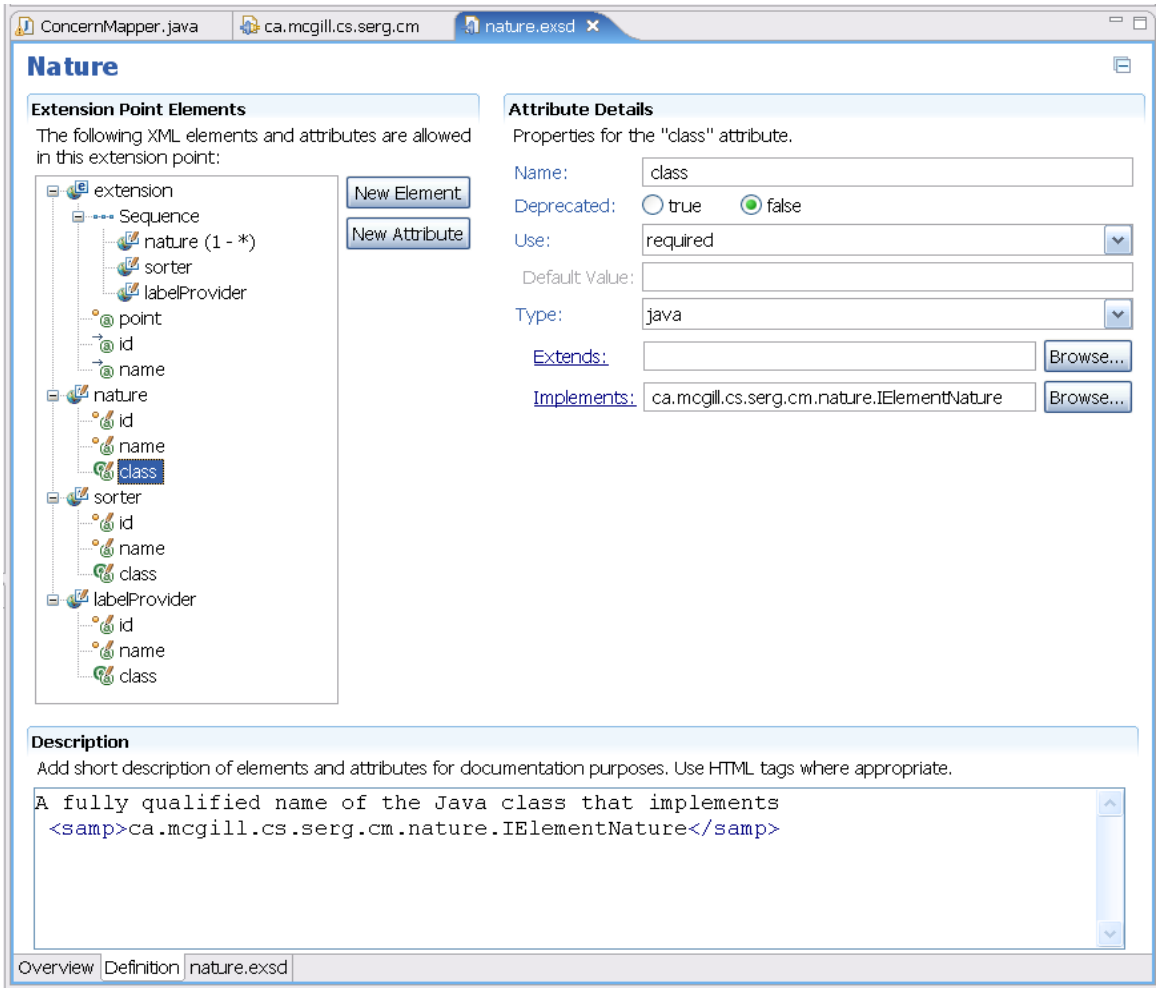
A major part of this project was spent in becoming familiar with the eclipse platform and development environment, the eclipse plug-in architecture and the ConcernMapper plug-in.
Following the extensive study of the eclipse architecture and the code of the existing ConcernMapper plug-in, I began to re-design the ConcernMapper to an extensible version. I did not set out to implement the change with a complete design for the extensible version in hand; instead the new design was gradually developed as I came across certain challenges during the process of extracting the code implementing Java specific functionality in ConcernMapper.

The final design is presented below.

The first step was to create an extension-point in the existing plug-in to define the requirements that an extending plug-in must fulfill when providing the definition for an element type.

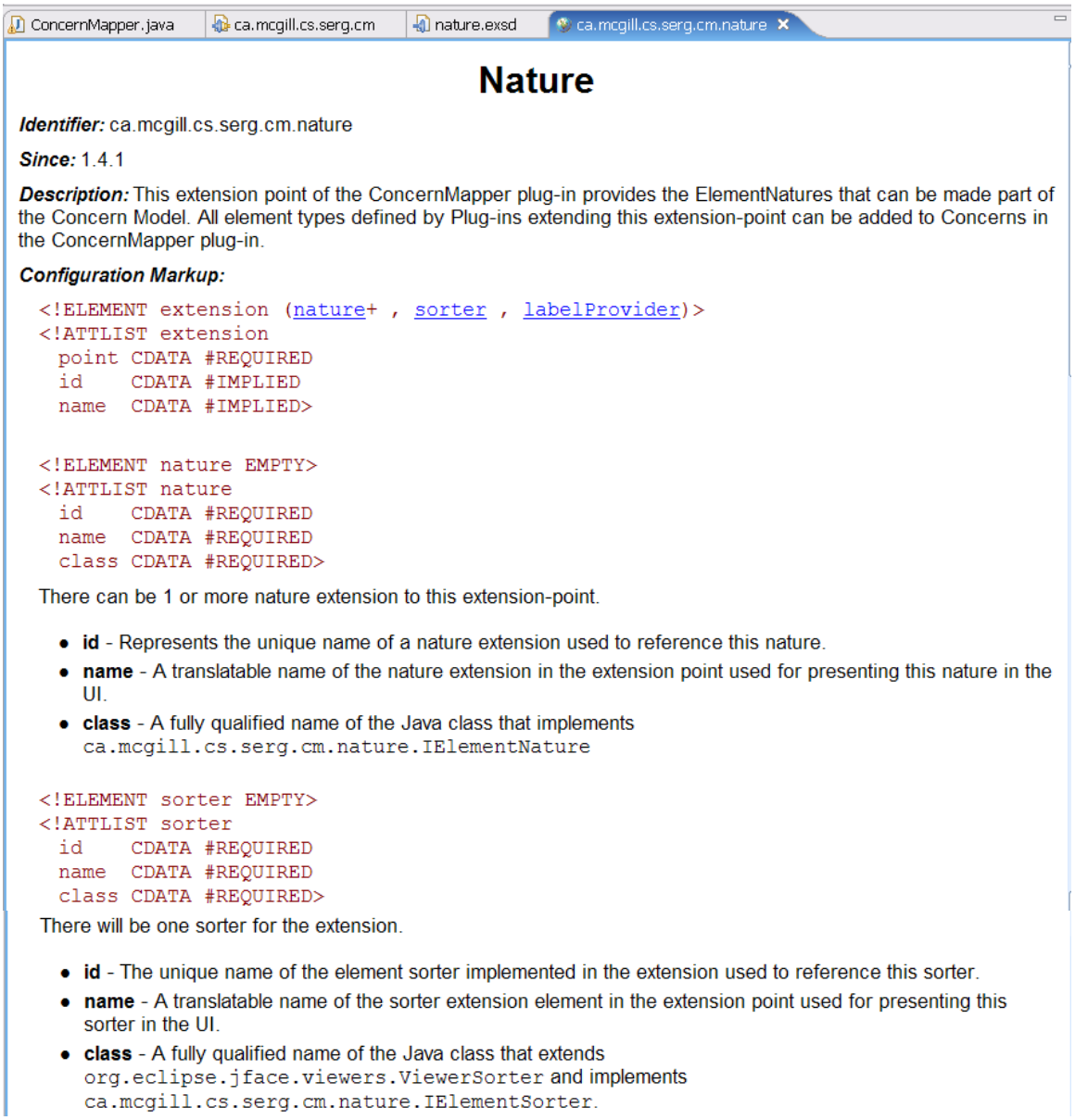


This schema is defined as shown below



The extension-point has three elements, namely, *Nature*, *Sorter*, and *LabelProvider*. The nature is the main extension-element with the class attribute that requires the extender plug-in to implement the '*IElementNature*' interface defined in the *nature* package. The extension-point uses the sequence compositor to force the extender plug-in to implement one instance of a *sorter* class implementing the '*IElementSorter*' interface, one instance of a *LabelProvider* and one or more instances of the *element nature*.

The extension-point description page shown below summarizes these properties.



```
<!ELEMENT labelProvider EMPTY>
<!ATTLIST labelProvider
  id    CDATA #REQUIRED
  name  CDATA #REQUIRED
  class CDATA #REQUIRED>
```

There will be one label provider for the extension.

- **id** - Represents the unique name of the label provider implemented in the extension used to reference this label provider.
- **name** - A translatable name of the extension to the labelProvider element in the extension point used for presenting this Label Provider in the UI.
- **class** - A fully qualified name of the Java class that extends org.eclipse.jface.viewers.LabelProvider that implements `ca.mcgill.cs.serg.cm.nature.ILabelProvider`.

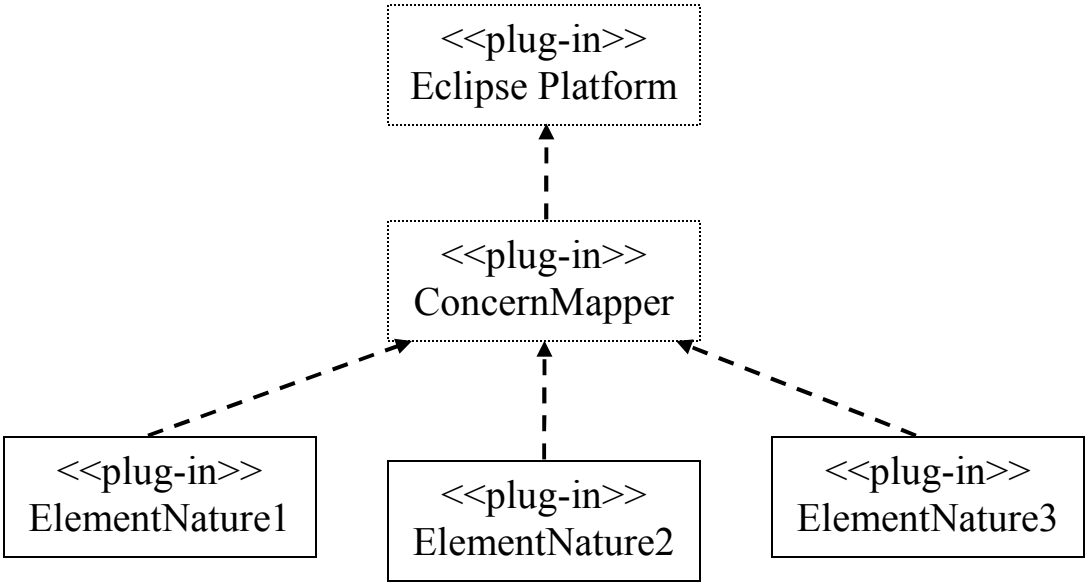
Examples: The following is an example of the extension point usage:

```
<extension
      point="ca.mcgill.cs.serg.cm.nature">
   <nature
         class="ca.mcgill.cs.serg.cm.nature.JavaField"
         id="ca.mcgill.cs.serg.cm.nature.jfield"
         name="ca.mcgill.cs.serg.cm.nature.jfield"/>
   <nature
         class="ca.mcgill.cs.serg.cm.nature.JavaMethod"
         id="ca.mcgill.cs.serg.cm.nature.jmethod"
         name="ca.mcgill.cs.serg.cm.nature.jmethod"/>
   <nature
         class="ca.mcgill.cs.serg.cm.nature.ResourceFile"
         id="ca.mcgill.cs.serg.cm.nature.file"
         name="ca.mcgill.cs.serg.cm.nature.file"/>
   <sorter
         class="ca.mcgill.cs.serg.cm.nature.ElementSorter"
         id="ca.mcgill.cs.serg.cm.nature.sorter"
         name="ca.mcgill.cs.serg.cm.nature.sorter"/>
   <labelProvider
         class="ca.mcgill.cs.serg.cm.nature.ConcernElementLabelProvider"
         id="ca.mcgill.cs.serg.cm.nature.labelProvider"
         name="ca.mcgill.cs.serg.cm.nature.labelProvider"/>
</extension>
```

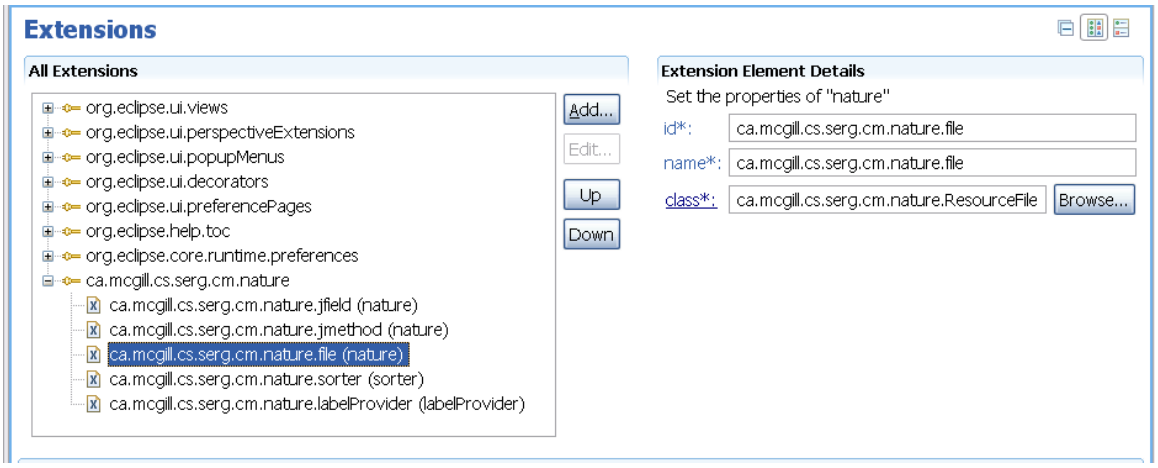
API Information: Plug-ins that want to extend this extension point must implement `ca.mcgill.cs.serg.cm.nature.IElementNature` , `ca.mcgill.cs.serg.cm.nature.IElementSorter` , and `ca.mcgill.cs.serg.cm.nature.ILabelProvider` interface.

Supplied Implementation: The current release of 1.4.1 provides default implementation of the nature for Java Methods, Java Fields, and Resource files.

This usage of the extension-point shown above was defined using the extension wizard for the plug-in definition file.
For the creation of the extension elements, I specified the id, name, and class as required in the definition above. The final extension is shown below



A diagram illustrating the ConcernMapper plug-in extension-point design



The main ConcernMapper class contains an instance of the sorter implementation class, a label provider class, and a list of the nature implementation classes provided by the extender plug-in. These 'callback objects' can be retrieved by the static call for the singleton and the following getter methods for these objects.

```
/**
 * This returns the ElementNatures implemented by extensions to the
 * 'nature' extension-point.
 * @return A list of natures implementing nature.IElementNature.
 */
public ArrayList<IElementNature> getElementNatures() {
    return aElementNatures;
}

/**
 * This returns the sorter provided by the nature extension.
 * @return An element sorter implementing nature.IElementSorter
 */
public IElementSorter getElementSorter() {
    return aElementSorter;
}

/**
 * This returns the label provider provided by the nature extension.
 * @return The label provider extending LabelProvider.
 */
public LabelProvider getConcernElementLabelProvider() {
    return aLabelProvider;
}
```

The element sorter and label provider is retrieved when setting the sorter and label provider for the tree viewer in the ConcernMapperView class.

```
aViewer.setSorter( (ViewerSorter) ConcernMapper.getDefault().getElementSorter());

aViewer.setLabelProvider( ConcernMapper.getDefault().getConcernElementLabelProvider() );
```

A list of callback objects for all the natures in the plug-in extension is created when the ConcernMapper constructor is called

```
private ArrayList<IElementNature> setElementNatures(){
    ArrayList<IElementNature> natures = new ArrayList<IElementNature>();
    IExtensionRegistry r= Platform.getExtensionRegistry();
    String pluginID= "ca.mcgill.cs.serg.cm";
    String extensionPointID= "nature";
    IExtensionPoint p= r.getExtensionPoint(pluginID, extensionPointID);
    IConfigurationElement[] c= p.getConfigurationElements();

    if (c != null) {
       for (int i= 0; i < c.length; i++) {
          IElementNature nature= null;
          try {
             IConfigurationElement ce= (IConfigurationElement) c[i];
             if(ce.getName().equals("nature")) {
                    nature= (IElementNature) c[i].createExecutableExtension("class");
             }
              if(nature != null)
                    natures.add((IElementNature)nature);
          } catch (CoreException x) {
             System.out.println(x.getMessage());
          }
       }
    }
    return natures;
}
```

We retrieve the *Extension Registry* from the *Platform* and then provide it with the plug-in id and the extension-point ('*nature*') that we are looking for. That gives us the extension-point from which we can then get a list of all its configuration elements. These are the elements we saw above that were defined in the extension-point schema. Each configuration element has a name which was defined above and here during extension-processing we distinguish between them using this name. In the case shown above, we are interested in the '*nature*' element so for each configuration element of type '*nature*', we instantiate the class implemented by the extender plug-in to create a callback object for this nature.

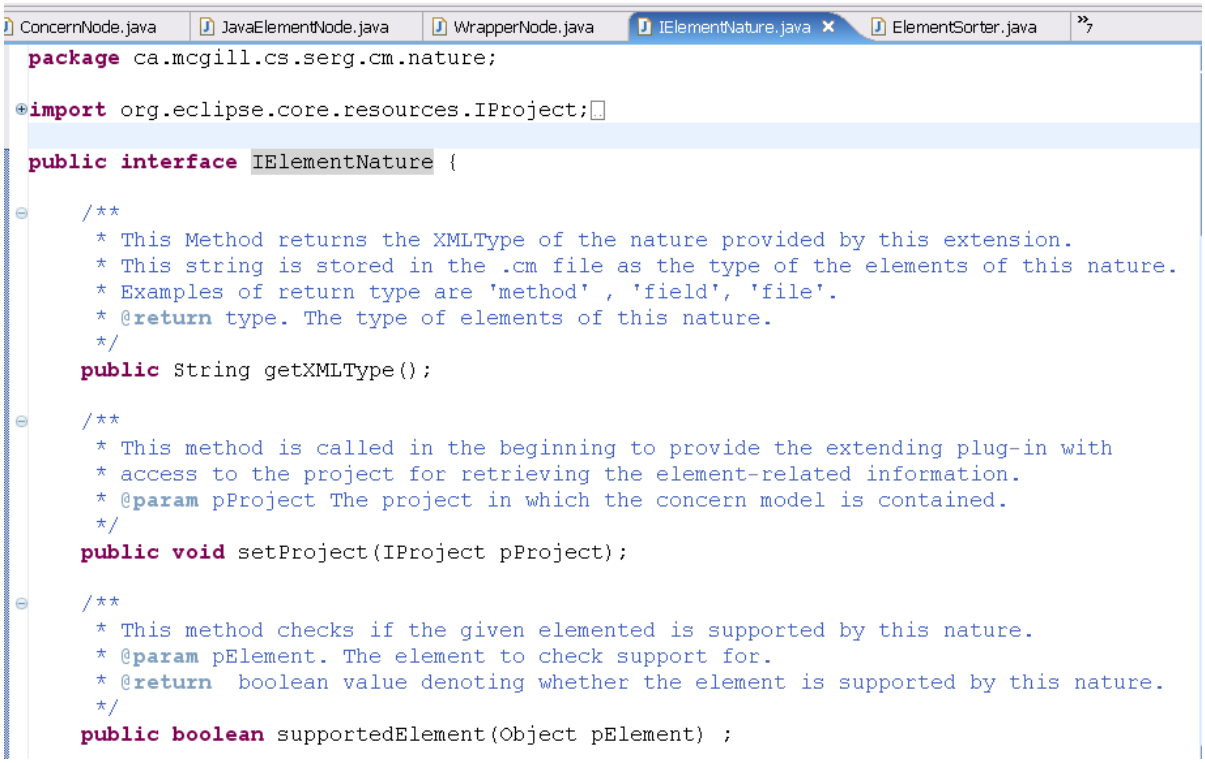
The most common use of the callback objects for the 'nature' element is to check for supported types wherever the elements were assumed to be a Java element in the previous plug-in code.
For example, in the '*AddToConcernAction*' class we replace the previous *supportedElement(IJavaElement pElement)*

```
/**
 * Determines if pElement can be included in a concern model.
 *
 * @param pElement
 *            The element to test
 * @return true if pElement is of a type that is supported by the
 *         concern model.
 */
private boolean supportedElement( IJavaElement pElement )
{
	boolean lReturn = false;
	if( ( pElement instanceof IField ) || ( pElement instanceof IMethod ) )
	{
		try
		{
			if( ( (IMember) pElement ).getDeclaringType().isAnonymous() ||
			(     (IMember) pElement ).getDeclaringType().isLocal())
			{
				lReturn = false;
			}
			else
			{
				lReturn = true;
			}
		}
		catch( JavaModelException lException )
		{
			ProblemManager.reportException( lException );
			lReturn = false;
		}
	}
	return lReturn;
}
```

With the following implementation using the '*element nature'* callback objects

```
/**
 * Determines if pElement can be included in a concern model.
 *
 * @param pElement
 *            The element to test
 * @return true if pElement is of a type that is supported by the
 *         concern model.
 */
private boolean supportedElement( Object pElement )
{
boolean lReturn = false;
for(IElementNature nature : ConcernMapper.getDefault().getElementNatures()) {
        if(nature.supportedElement(pElement)) {
                lReturn = true;
                break;
        }
}
return lReturn;
}
```

In the rest of the code replacement we use a number of methods that are provided by the *IElementNature* interface. This interface is shown below to give a picture of what the extender plug-in needs to implement for creating an extension to this ConcernMapper plug-in apart from providing a sorter and a label provider.



```java
/**
 * This method provides a broader test of the support for the element. This would
 * return true even if the element is wrapped in a node.
 * @param pElement The element to needs to be checked for support.
 * @return true/false whether the element is supported by the implementing nature object.
 */
public boolean supportedType(Object pElement) ;

/**
 * This method checks for the given element whether it exists in the resource model.
 * @param pElement The element that needs to be checked for existence.
 * @return The boolean value representing the existence of the element.
 */
public boolean elementExists( Object pElement) ;

/**
 * Given a concern element object, this method returns an instance of a wrapper Node
 * containing the given object.
 * @param pObject The object to be wrapped.
 * @return The concern view node wrapping the given object.
 */
public IConcernMapperViewNode getElementNode(Object pObject);

/**
 * This method returns the parent of this element in the hierarchy of the View.
 * Example, the parent of a method element would be a Class and the parent
 * of a non-inner Class would be a ConcernNode.
 * @param pElement The element for which the parent is needed.
 * @return The parent of the given element in the tree.
 */
public Object getStructureParent(Object pElement);

/**
 * This method returns a set of parent-structure for the given element
 * to be decorated in the various views. The integer encoding for the
 * decoration limit element is provided as an argument.
 * @param pElement The element for which the parent-structure is being retrieved.
 * @param pTopParent The top most element to be decorated in the parent-structure.
 * @return The set of parent-elements to be decorated.
 */
public Set<Object> getParentsToDecorate(Object pElement, int pTopParent);


/**
 * This method provides a way to convert the string representation of a concern
 * element into the object that it represents from the project resource.
 * This method is used when loading concerns from a file.
 * @param pId the element's string id.
 * @return The object represented by this string.
 * @throws ConversionException
 */
public Object convertStringToElement(String pId)  throws ConversionException;
/**
 * This method provides a way to convert the given element into a string
 * representation of the element. This method is used when saving concerns to file.
 * @param pElement The concern element that is being stored in file.
 * @return The string representation of the element to be stored in the .cm file
 * @throws ConversionException
 */
public String convertElementToString(Object pElement) throws ConversionException;

/**
 * This method provides context content for the menu.
 * @param pManager The menu manager where the new menu group will be added
 * @param pElement The element for which the context menu is needed.
 * @return  The Menu Manager with the context Menu group added.
 */
public IMenuManager fillContextMenu(IMenuManager pManager, Object pElement);
```

```
    /**
     * This method implements a way to load the given object in the corresponding
     * editor.
     * @param pObject The object to be revealed in the editor.
     */
    public void revealInEditor(Object pObject);

    /**
     * This is a helper method to provide initialization data that an extension
     * might use when populating the context menu. For example, the Java extension
     * would use the ViewPart to initialize a JavaSearchActionGroup.
     * @param pViewPart
     */
    public void initSearchDataForViewPart(ViewPart pViewPart);
}
```

The methods for converting from ‘element to string’ and ‘string to element’ are used when writing to a file and reading from a file in the ‘*model.io*’ package.

```
    private Element createConcernNode( Document pDocument, String pConcern ) throws
ModelIOException
    {
        …
        Set lElements = aModel.getElements( pConcern );
        for( Iterator lI = lElements.iterator(); lI.hasNext(); )
        {
                …
                for(IElementNature nature : ConcernMapper.getDefault().getElementNatures()){
                        try {
                                if( nature.supportedElement(lNext)) {
                                        natureFound = true;
                                        lElement.setAttribute( XMLElementTypes.ATTRIBUTE_TYPE,
                                        nature.getXMLType() );
                                        lElement.setAttribute( XMLElementTypes.ATTRIBUTE_ID,
                                        nature.convertElementToString(lNext ));
                                        lElement.setAttribute( XMLElementTypes.ATTRIBUTE_DEGREE,
                                        new Integer( aModel.getDegree( pConcern, lNext
                                        )).toString());
                                }
                        }
                        catch(ConversionException lException){
                                lReturn.removeChild( lElement );
                        }
                }
                …
                /* Similarly done for inconsistent elements */
                …
    }
```

Here we go through the list of ‘*natures*’ and whenever we find a match for the object we are comparing, we request the callback object to provide a string representation for this object.

The implementation of the required methods from the interface in the Java Extension to the ConcernMapper plug-in is very similar to the actual implementation for Java Elements in the previous ConcernMapper plug-in.

A good example for that would be the '*fillContextMenu(IMenuManager)*' method in the *ConcernMapperView* class.

```
/**
 * Fills the context menu based on the type of selection.
 * @param pManager
 */
private void fillContextMenu( IMenuManager pManager )
{
	ISelection lSelection = aViewer.getSelection();
	if( lSelection instanceof IStructuredSelection )
	{
		IStructuredSelection lStructuredSelection =
		(IStructuredSelection)lSelection;
		if( lStructuredSelection.size() == 1 )
		{
			Object lNext = lStructuredSelection.iterator().next();
			if( lNext instanceof ConcernNode )
			{
				pManager.add( new RenameConcernAction( this,
				((ConcernNode)lNext).getConcernName() ));
			}
			else
			{
				for(IElementNature nature :
				ConcernMapper.getDefault().getElementNatures()){
					if(nature.supportedType(lNext)) {
						pManager =
						nature.fillContextMenu(pManager, lNext);
						break;
					}
				}
			}
		}
		pManager.add( getDeleteActionFromSelection( lStructuredSelection ));
	}
	pManager.add( new Separator( IWorkbenchActionConstants.MB_ADDITIONS ));
}
```

Here first we get the selected element in the Concern Model for which the user is requesting the context menu. Then we search for that element's type in the list of callback objects from the extender plug-in if the selected element wasn't a ConcernNode itself. Once a match is found for a supported element in a callback object, we populate the *MenuManager* by passing the *MenuManager* as an argument to the callback object's *fillContextMenu(IMenuManager, Object)* method which returns a filled version of the *MenuManager* as demonstrated in the code below.

```
/**
 * This method provides context content for the menu.
 * @param pManager The menu manager where the new menu group will be added
 * @param pElement The element for which the context menu is needed.
 * @return     The Menu Manager with the context Menu group added.
 */
public IMenuManager fillContextMenu(IMenuManager pManager, Object pElement){
	IJavaElement lElement = ((JavaElementNode)pElement).getElement();
	if( lElement.exists() )
	{
		aJavaSearchActions.setContext( new ActionContext( new
		StructuredSelection( lElement )));
		GroupMarker lSearchGroup = new GroupMarker("group.search");
```

```
                pManager.add( lSearchGroup );
                aJavaSearchActions.fillContextMenu( pManager );
                aJavaSearchActions.setContext( null );
                pManager.remove( lSearchGroup );
            }
            return pManager;
        }
```

The *SearchActionGroup* for Java was initialized with the *ConcernMapperView's ViewPart* using the following helper method in the *ConcernMapperView* class' *createPartControl(Composite)* method.

```
        /**
         * This is a helper method to provide initialization data that an extension
         * might use when populating the context menu. For example, the Java extension
         * would use the ViewPart to initialize a JavaSearchActionGroup.
         * @param pViewPart
         */
        public void initSearchDataForViewPart(ViewPart pViewPart){
            aJavaSearchActions = new JavaSearchActionGroup(pViewPart);
        }
```

As a *doubleClickAction* for the *Resource File* we had to provide a *revealInEditor* method that would open the file that was clicked upon, in the editor view of Eclipse.
This *doubleClickAction* is created in the *makeActions()* method of the ConcernMapperView class by creating an instance of the *Action* class that calls the corresponding *nature callback object* to reveal in the editor the given object.

```
aDoubleClickAction = new Action()
  {
        public void run()
        {
            ISelection lSelection = aViewer.getSelection();
            Object lObject = ((IStructuredSelection)lSelection).getFirstElement();

            for(IElementNature nature : ConcernMapper.getDefault().getElementNatures()) {
                if(nature.supportedType(lObject)) {
                    nature.revealInEditor(lObject);
                       break;
                }
            }
        }
  };
```

This method for revealing the object in the editor is then implemented by the extending plug-in in the implementation for a *Resource File* element type

```
  /**
   * This method implements a way to load the given object in the corresponding
   * editor.
   * @param pObject The object to be revealed in the editor.
   */
  public void revealInEditor(Object pObject) {
        IFile lElement = (IFile)((WrapperNode)pObject).getElement();
      if( lElement.exists() )
      {
```

```
        IWorkbenchPage lPage =
                PlatformUI.getWorkbench().getActiveWorkbenchWindow().getActivePage();
                try
                {
                        IDE.openEditor( lPage, lElement);
                }
                catch( PartInitException lException )
                {
                        ProblemManager.reportException( lException );
                }
    }
}
```

In the *Label Provider* the, for the *Resource File* type, the default 'file image' is returned by the *getImage(Object)* method.

```
        /**
         * Provides the image for an object in a concern model.
         * @param pObject The object to provide the image for.
         * @return The image
         */
        public Image getImage( Object pObject )
        {
                …
                else if( pObject instanceof WrapperNode )
                {
                        Object lElement = ((WrapperNode)pObject).getElement();
                        if(lElement instanceof IFile)
                        {
                                lReturn = PlatformUI.getWorkbench().getSharedImages().getImage(
              ISharedImages.IMG_OBJ_FILE );
                        }
                }
                …
        }
```

A major part of the Java element code extraction was to be able to dynamically create a structure for the Concern Model from the elements that get added to a concern. For this purpose the extending nature provides the host plug-in with a method called *getElementNode(Object)* which takes as an argument the element that is being added to the concern, and it wraps it into a wrapper node and returns it as an *IConcernMapperViewNode* so that from the Concern Models point of view, all elements are wrapped in the same form, no matter what the element type turns out to be.

This concludes the implementation summary of the extensible version of ConcernMapper.

Unsolved Challenges

During the course of this project there were a couple of design issues that came up which I wasn't able to deal with satisfactorily.

When a concern is loaded or elements are added to a new concern, the ConcernMapper decorates these elements and other elements related to the one added (i.e. the immediate parents in the hierarchical structure within the project from which the element was dragged into the concern). These decorations include changing the font of the included element and its immediate parents to **bold** as well as displaying the name of the **concern** that the element has been added to, next to the name of the element in the other views in the workbench.

In the current version, the ConcernMapper plug-in provides the choice of the limit to which the parents of an element in the hierarchical tree-structure in the workbench views should get the **bold** font. The way this feature is implemented in the plug-in requires the comparison of the encoded integer value of the possible element types in the hierarchical tree-structure within a Java project with the value stored as a preference (which can be set by the user).

In the new version of ConcernMapper, the user is no longer limited to adding only Java elements and therefore the elements added to the Concern Model could possibly have different integer encoding than those of Java elements hence making comparison between the saved preferences and the actual values of the element and their hierarchical parents infeasible.

As a temporary fix in the final version, the plug-in is being released with an added facility to add any type of resource file into the Concern Model and the decoration of the element in the workbench views is limited the actual element itself.

A further complexity in this issue occurs once we are able to return the immediate parent of the resource file being dragged-in (which would be either a folder or the project root). The way the decoration works is by performing a search over all the available elements of the project (i.e. project root, package root, package fragments, and classes) where as the only way/'light' in which the resource file sees its parent, is as a 'folder' and *not* as a 'package fragment' within a larger project since a simple resource file cannot perceive the larger view which includes the project that it belongs to. Therefore the search is

looking at the same element as a package fragment and the file element claims it to be its parent folder, leading to a mismatch, making the two concepts incomparable.
One possible solution to this issue could include re-defining the search method for finding the parents provided by the element by including a separate check for cases like this mismatch.

Another issue that I wasn't able to solve is the implementation of the *ConcernMapperElementChangeListener* class which implements the *IElementChangedListener* interface which implements an element changed listener that receives notification of changes to Java elements maintained by the Java model. This implementation relies mainly on

org.eclipse.jdt.core.ElementChangedEvent

An element changed event describes a change to the structure or contents of a tree of Java elements. The changes to the elements are described by the associated delta object carried by this event.

This class is not intended to be instantiated or subclassed by clients. Instances of this class are automatically created by the Java model.

See Also:

IElementChangedListener
IJavaElementDelta

This class uses the IJavaElementDelta which encodes in it the changes made to a Java Element in the Java Model as a Set containing the original element description and the new element description, be it an addition, removal or a static change to an existing Java element.

Related Work and Availability

The ConcernMapper plug-in was developed and is currently maintained by Professor Martin Robillard and Frédéric Weigand-Warr. The source code for the plug-in is available at *http://www.cs.mcgill.ca/~martin.cm* and it may be worked upon to experiment further extensions to the existing plug-in.

This extensible version of the plug-in is available for testing and experimentation for anyone interested in creating an extension for a new type of Concern Element.

Conclusion

While this project is only a small addition to the existing ConcernMapper plug-in, there is still a lot to be gained from this plug-in through further extensions. It provides good grounds for research in patterns of concern usage during the software development lifecycle. Further extensions to this plug-in could provide the user with statistical analysis of the usage of this plug-in during the active stage of a project in an eclipse environment. This could include the number of concerns created, the correlation between their contents, the frequency of their usage by the developer as compared to the conventional navigational paths in eclipse, and other statistical data produced in the form of graphical correlation. This could help in the analysis of the current design of a large project and suggest ways for design improvement by motivating towards an aspect oriented design.

Thus there is much to be benefited from use of this tool in Software Engineering Research, and any further enhancements to the current functionalities could greatly help the research community in that regard.