HowTo -- Adding additional Tools (aka Views)

Adding additional Tools (aka Views)

This document describes how to build a new Tool (For Eclipse users, a Tool in MPS is like a View in Eclipse) for MPS. This can serve as an example to build arbitrary additional tools into MPS. This text emphasizes to aspects of Tool development: how to add a new tool to a language and the menu system, and how to synchronize the view with whatever is currently edited.

In all other ways tools are just Swing UI programs. So for implementing sophisticated views, one needs Java Swing experience. In this tutorial we will not focus too much on the intricacies of building a tree view with Swing — an endeavour that is surprisingly non-trivial!

The Outline Tool itself

The MPS plugin language supports the definition of new Tools. Create a new instance of `Tool` and set the name. You can also give it a different caption and an icon. A default location (bottom, left, right, top) can also be defined.

We now add three fields to the Tool. The first one is used to remember the project, the second one connects the tool to MPS's message bus, and the third one remember the `ToolSynchronizer. The ToolSynchronizer` updates the outline view in case the active editor changes. The message bus is IDEA platform's infrastructure for event distribution. We use it for getting notified of selection changes in the currently active editor. More on these two later.

```
private Project project;
private MessageBusConnection messageBusConn;
private ToolSynchronizer synchronizer;
```

We are then ready to implement the three main methods of a Tool: `init`, `dispose and getComponent`. Here is the code (with comments, please read them!):

```
init(project)->void {
  this.project = project;
}

dispose(project)->void {
  // disconnect from message bus -- connected in getComponent()
  if ( this.messageBusConn != null ) this.messageBusConn.disconnect();
}

getComponent()->JComponent {
  // create a new JTree with a custom cell renderer (for rendering custom icons)
  JTree tree = new JTree(new Object[0]);
  tree.setCellRenderer(new OutlineTreeRenderer(tree.getCellRenderer()));
  // create a new synchronizer. It needs the tree to notify it of updates.
  this.synchronizer = new ToolSynchronizer(tree);
  // connect to the message bus. The synchronizer receives editor change events
  // and pushes them on to the tree. The synchronizer implements
  // FileEditorChangeListener to be able to make this work
  this.messageBusConn = this.project.getMessageBus().connect();
  this.messageBusConn.subscribe(FileEditorManagerListener.FILE_EDITOR_MANAGER,
        this.synchronizer);
  // we finally return a ScrollPane with the tree in it
  return new JScrollPane(tree);
}
```

An Action to Open the Tool

Actions are commands that live in the MPS UI. We have to add an action to open the outline view. Actions live in the plugins aspect of a language. Actions can define keyboard shortcuts, captions and icons, as expected. They also declare action parameters. These define which context needs to be available to be able to execute the action. This determines the presence of the action in the menu, and supports delivering that context into the action itself. In our case the context includes the currently opened project as well as the currently opened file editor.

```
action context parameters ( always visible = false )

    Project project key: PROJECT required

    FileEditor editor key: FILE_EDITOR required
```

In the `execute` method of the action we create and open the Outline tool. The `setOutline` method is an ordinary method that lives it the Outline Tool itseld. It simply stores the editor that's passed in.

```
execute(event)->void {
  tool<Outline> outline = this.project.tool<Outline>;
  outline.setEditor(this.editor);
  outline.openTool(true);
}
```

An Action Group

Menu items are added via groups. To be able to add the Open Outline Action to the menu system, we have to define a new group. The group defines its contents (the action, plus a two separators) and it determines where in the menu it should go. Groups live in the plugin aspect as well.

```
group OutlineGroup
is popup: false

contents
  <--->
  OpenOutlineAction
  <--->

modifications
  add to Tools at position customTools
```

Managing the Tool Lifecycle

The tool must play nicely with the rest of MPS. It has to listen for a number of events and react properly. There are two listeners dedicated to this task. The `EditorActivationListener` tracks which of the potentially many open editors is currently active, since the outline view has to show the outline for whatever editor is currently used by the user. It is also responsible to hooking up further listeners that deal with the selection status and model changes (see below). The `ModelLifecycleListener` tracks lifecycle events for the model that is edited by the currently active editor. The model may be replaced while it is edited, for example, by a revert changes VCS operation.

Editor Activation and Focus

The `EditorActivationListener` tracks which of the potentially many open editors is currently active. It is instantiated and hooked up to the MPS message bus by the tool as it is created by the following code (already shown above):

```
this.messageBusConn = this.project.getMessageBus().connect();
this.messageBusConn.subscribe(FileEditorManagerListener.FILE_EDITOR_MANAGER,
        this.synchronizer);
```

In its constructor, it remember the outline tree. Then it sets up a new outline tree selection listener that listens to selection changes made by the user in the tree itself.

```
this.outlineSelectionListener = new OutlineSelectionListener(this);
tree.addTreeSelectionListener(this.outlineSelectionListener);
```

It then sets up a listener to keep track of the model lifecycle (load, unload, replace) and it hooks up with the events collector (both explained below).

```
modelLifecycleListener = new ModelLifecycleListener(tree);
eventsCollector = new ModelChangeListener(tree);
```

The `EditorActivationListener implements FileEditorManagerListener`, so it has to implement the following three methods:

```
void fileOpened(FileEditorManager p0, VirtualFile p1);
void fileClosed(FileEditorManager p0, VirtualFile p1);
void selectionChanged(FileEditorManagerEvent p0);
```

In our case we are interested in the `selectionChanged` event, since we'll have to clean up and hook up all kinds of other listeners. Here is the implementation.

```
public void selectionChanged(FileEditorManagerEvent event) {
  // read action is required since we access the model
  read action {
    FileEditor oldEditor = event.getOldEditor();
    // grab the old editor and clean it up if there is one
    if (oldEditor != null) {
      this.cleanupOldEditor(oldEditor);
    }
    // call a helper method that sets up the new editor
    this.newEditorActivated(event.getNewEditor());
  }
}
```

The `cleanupOldEditor` method removes existing listeners from the old, now inactive editor:

```
private void cleanupOldEditor(FileEditor oldEditor) {
  // Downcast from IDEA level to MPS specifics and
  // grab the NodeEditor component
  IEditor oldNodeEditor = ((MPSFileNodeEditor) oldEditor).getNodeEditor();
  if (oldNodeEditor != null && this.editorSelectionListener != null) {
    // remove the selection listener from the old editor
    oldNodeEditor.getCurrentEditorComponent().getSelectionManager().
            removeSelectionListener(this.editorSelectionListener);
    // grab the descriptor of the model edited by the old editor
    // and remove the model listener (cleanup!)
    SModelDescriptor descriptor = oldNodeEditor.getEditorContext().getModel().
            getModelDescriptor();
    descriptor.removeModelListener(modelLifecycleListener);
    // remove the model edited by the old editor from the events collector
    // ...we are not interested anymore.
    eventsCollector.remove(descriptor);
  }
}
```

The next method hooks up the infrastructure to the newly selected editor and its underlying model. Notice how we use `SNodePointer` whenever we keep references to nodes. This acts as a proxy and deals with resolving the node in case of a model is replaced and contains a "weak reference" to the actual node, so it can be garbage collected if the model is unloaded. This is important to avoid memory leaks!

```
public void newEditorActivated(FileEditor fileEditor) {
  if (fileEditor != null) {
    // remember the current editor
    this.currentEditor = ((MPSFileNodeEditor) fileEditor);
    // grab the root node of that new editor...
    SNode rootNode = this.currentEditor.getNodeEditor().
            getCurrentlyEditedNode().getNode();
    // ...wrap it in an SNodePointer...
    SNodePointer treeRoot = new SNodePointer(rootNode);
    // and create a new outline tree model
    OutlineModel model = new OutlineModel(treeRoot);
    tree.setModel(model);

    // create a new selection listener and hook it up
    // with the newly selected editor
    this.editorSelectionListener = new EditorSelectionListener(tree,
            outlineSelectionListener);
    SelectionManager selectionManager = this.currentEditor.getNodeEditor().
            getCurrentEditorComponent().getSelectionManager();
    selectionManager.addSelectionListener(this.editorSelectionListener);

    // This is needed to detect reloading of a model
    ((MPSFileNodeEditor) fileEditor).getNodeEditor().
           getEditorContext().getModel().getModelDescriptor().
           addModelListener(modelLifecycleListener);
    eventsCollector.add(this.currentEditor.getNodeEditor().
           getEditorContext().getModel().getModelDescriptor());
  } else {
    tree.setModel(new OutlineModel(null));
  }
}
```

Tracking the Model Lifecycle

The `ModelLifecycleListener extends the SModelAdapter` class provided by MPS. We are intereted in the model replacement and hence overload the `modelReplaced` method. It is called whenever the currenly held model is replaced by a new one, e.g. during a VCS rever operation.

In the implementation, we create a new new tree model for the same root. While this code looks a bit nonsensical, note that we use an `SNodePointer` internally which automatically re-resolves the proxied node in the new model. We also add ourselves as a listener to the new model's descriptor to be notified of subseqent model replacement events.

```
@Override
public void modelReplaced(SModelDescriptor descriptor) {
  tree.setModel(new OutlineModel(((OutlineModel) tree.getModel()).getRealRoot()));
  descriptor.addModelListener(this);
}
```

Synchronizing Node Selections

Tracking Editor Selection

This one updates the selection (and expansion status) of the tree as the selected node in the currently active editor changes. We have already made sure (above) that the outline view's tree is synchronized with the currently active editor.

The class extends MPS' `SingularSelectionListenerAdapter` because we are only interested in single node selections. We overwrite the `selectionChangedTo` method in the following way:

```
protected void selectionChangedTo(EditorComponent component,
                                  SingularSelection selection) {
  // do nothing is disabled -- prevents cyclic, never ending updates
  if (disabled) { return; }
  // read action, because we access the model
  read action {
    // gran the current selection
    SNode selectedNode = selection.getSelectedNodes().get(0);
    // ... only if it has changed...
    if (selectedNode != lastSelection) {
      lastSelection = selectedNode;
      // disable the tree selection listener, once again to prevent cyclic
      // never ending updates
      treeSelectionListener.disable();
      // select the actual node in the tree
      tree.setSelectionPath(((OutlineModel) tree.getModel()).
            gtPathTo(selectedNode));
      treeSelectionListener.enable();
    }
  }
}
```

Tracking Changes in the Model Structure

The tree structure in the outline view has to be updated if nodes are added, changed (renamed), moved or deleted in the editor. Tree change events can be quite granular, and to avoid overhead, MPS collects them into batches related to more coarse-grained commands. By using MPS' `EventsCollector` base class, we can get notified when a significant batch of events has happened, and then inspect the event list for those that we are interested in using a visitor.

The `ModelChangeListener` performs this task. To do so, we have to implement the `eventsHappened` method. We get a list of events, and use a an inner class that extends `SModelEventVisitorAdapter` to visit the events and react to those that we are interested in.

```
protected void eventsHappened(List<SModelEvent> list) {
  super.eventsHappened(list);
  foreach evt in list {
    evt.accept(new ModelChangeVisitor());
  }
}
```

The `ModelChangeVisitor` inner class, which acts as the visitor to notify the tree, overrides `visitPropertyEvent` to find out about nodes whose properties have changed in the current batch. It then notifies all the listeners of the tree model.

```
public void visitPropertyEvent(SModelPropertyEvent event) {
  OutlineModel outlineModel = ((OutlineModel) tree.getModel());
  foreach l in outlineModel.getListeners() {
    l.treeNodesChanged(new TreeModelEvent(this,
          outlineModel.getPathTo(event.getNode())));
  }
}
```

It also overwrites `visitChildEvent` to get notified of child additions/deletions of nodes. Except that the API in `JTree` is a bit annoying, the following commented code should be clear about what it does:

```
@Override
public void visitChildEvent(SModelChildEvent event) {
  // grab the model
  OutlineModel outlineModel = ((OutlineModel) tree.getModel());
  // we need the following arrays later for the JTree API
  Object[] child = new Object[]{event.getChild()};
  int[] childIndex = new int[]{event.getChildIndex()};
  // we create a tree path to the parent notify all listeners
  // of adding or removing children
  TreePath path = outlineModel.getPathTo(event.getParent());
  if (path == null) { return; }
  // notify the tree model's listeners about what happened
  foreach l in outlineModel.getListeners() {
    if (event.isAdded()) {
      l.treeNodesInserted(new TreeModelEvent(this, path, childIndex, child));
    } else if (event.isRemoved()) {
      l.treeNodesRemoved(new TreeModelEvent(this, path, childIndex, child));
    }
  }
}
```

The way back: Tracking Tree Selection

Tracking a `JTree`'s selection happens by implementing Swing's `TreeSelectionListener and overwriting it's valueChanged` method the following way:

```
public void valueChanged(TreeSelectionEvent event) {
  // don't do anything if disabled --- preventing cyclic updates!
  if (!(disableEditorUpdate)) {
    JTree tree = ((JTree) event.getSource());
    if (editorActivationListener.currentEditor != null &&
            tree.getLastSelectedPathComponent() instanceof SNodePointer) {
      // grab the selected treee node
      SNodePointer pointer = ((SNodePointer) tree.
            getLastSelectedPathComponent());
      // disable the editor selection listener to prevent
      // cyclic, never ending updates
      editorActivationListener.editorSelectionListener.disable();
      // update the selection in the editor
      editorActivationListener.currentEditor.getNodeEditor().
             getCurrentEditorComponent().selectNode(pointer.getNode());
      editorActivationListener.editorSelectionListener.enable();
    }
  }
}
```

Swing's Artifacts: Tree Model and Renderer

In this section I want to describe a couple of interesting MPS-specific aspects of the implementation of the Swing artifacts.

Tree Cell Renderer

The tree cell renderer is responsible for rendering the cells in the tree. It uses the `getPresentation method on the nodes, and the IconManager` to get (a cached version of) the icon for the respective node.

```
public Component getTreeCellRendererComponent(JTree tree, Object object,
            boolean selected, boolean expanded,
            boolean leaf, int row, boolean hasFocus) {
  if (object instanceof SNodePointer) {
 string presentation;
 DefaultTreeCellRenderer component;
 read action {
   SNode node = ((SNodePointer) object).getNode();
   presentation = node.getPresentation();
   component = ((DefaultTreeCellRenderer) renderer.getTreeCellRendererComponent(
                  tree, presentation, selected, expanded, leaf, row, hasFocus));
   component.setIcon(IconManager.getIconFor(node));
 }
  } else {
    return renderer.getTreeCellRendererComponent(tree, object, selected,
          expanded, leaf, row, hasFocus);
  }
}
```

Tree Model

The tree model is interesting since we include only those children in the tree node whose concept has been annotated with the

`ShowConceptInOutlineAttribute` attribute. It is stored in the `storeInOutline` role. In the tree model, we have to filter the children of a node for the presence of this attribute. Here is the respective helper method:

```
private List findAllChildrenWithAttribute(SNodePointer pointer) {
  List result = new ArrayList();
  SNode node = pointer.getNode();
  if (node == null) { return new ArrayList(); }
  foreach child in node.getChildren() {
    SNode attribute = AttributeOperations.getNodeAttribute(
                      child.getConceptDeclarationNode(), "showInOutline");
    if (attribute != null) {
      result.add(child);
    }
  }
  return result;
}
```