www.sensoria-ist.eu

The SENSORIA Development Environment

Creating a Sensoria SDE Tool

Philip Mayer
Programming and Software Engineering Research Group
Department of Informatics
LMU Munich
Germany

Information Society Technologies

Creating a Sensoria SDE Tool

- An **SDE-integrated tool** can take many forms
- The only requirement is some XML code to register a Java-Based API (might be a wrapper) to the SDE

- Mostly, however, SDE-Integrated Tools are fully-fledged **Eclipse Plug-Ins** as well
  - They provide their own UI to the Eclipse platform
  - Additionally, they offer API for scripting through the SDE

- We will follow this approach in creating our own tool for the SDE
  - We'll use the Eclipse PDE (Plug-In Development Environment) to create an Eclipse Plug-In
  - Later, we'll use the SDE Dev tools to add support for the SDE

The Tool

- We'll create a tool which enables us to check for "problems" (non-well-nestedness, cycles...) in UML4SOA model files
- We want to perform this check
  - At any time, i.e. using the UI for an arbitrary model
  - In an SDE workflow, for example before converting to BPEL.

- Input of the tool: UML4SOA UML model of an orchestration
- Output of the tool: Report on problems, if there are any, in the UI.

What we need...

- **Eclipse** (as our underlying platform). This includes
  - **Eclipse PDE** (Plug-In Development Environment) for writing **Plug-Ins**
  - **SWT/JFace**: The Standard Widget Toolkit is the Widget Toolkit used by Eclipse (instead of Swing). JFace is an extension of this toolkit.
- An **UML input model**. We'll use the output provided by MagicDraw, which is an XML-based format. In Eclipse, we can use the following tools/techniques to read the input and query it:
  - **EMF** (Eclipse Modelling Framework)
  - **UML2Tools** (UML2 Meta Model implemented in EMF)
- The **SDE**. We use the SDE Dev tools to create an integration of our tool into the SDE.

- We start by creating a new **Eclipse project** inside the workspace. To do this, we follow these steps:
  - Add a new Plug-In project
    - Select “Generate an Activator”
    - Select “This Plug-In will make contributions to the UI”
    - Do not select any template
  - Exploring the Plug-In
    - `MANIFEST.MF/plugin.xml` contains meta-settings
    - Activator class contains setup routines
  - Testing the Plug-In
    - Launch an Eclipse Runtime Workbench
    - (Nothing to see – no contributions to the UI yet)
- Now, we have an empty Plug-In we can use for our purposes.
- How can we use it?

- The Eclipse platform has a very open architecture – it enables plug-ins to "extend" it at many places by **contributing** to an **extension point**.
- Extension points exist for many purposes, for example for
  - …adding a new menu, or a menu entry (an **action**)
  - …adding a new editor
  - …adding a new view
  - …adding status line items, cool bar items, perspectives, etc.
  - …adding non-UI items like source code control etc.
  - …
- Plug-Ins contribute to an extension point by declaring an **extension** in their `plugin.xml` file.
- Each plug-in can also provide its own extension points and can therefore be extended itself, making the system very flexible.

Adding an entry point

- In our case, the user must be able to execute the problem analysis based on a UML model.
- UML models reside in a file with a `.uml` extension.

- To add an action to files, we need to extend the popup menu on files
- In `MANIFEST.MF/plugin.xml`, select the **Extensions** tab
- Add a new `org.eclipse.ui.popupMenus` extension with template **Popup Menu** to create an object contribution
- Enter some settings (stay with `IFile` as contribution object)
- Use Runtime Workbench to test

- Restricting the contribution to `.uml` files:
  - Add `"*.uml"` as name filter on the object contribution.

- UML models in `.uml` files are stored in an XML dialect which we can process using the **Eclipse Modelling Framework (EMF)**.

- EMF allows creation of **meta-models** of arbitrary domains; for example, the UML2 language.
- Concrete instances of these meta-models, i.e. **models**, can then be created, queried, and manipulated either dynamically, or through generated Java classes.

- The Eclipse project provides a complete meta-model of the UML2 based on EMF.
- This means we can load, store, create, and manipulate UML2 models using the EMF mechanisms.

- In our case, we can load and query a UML2 model in a `.uml` file through EMF.

Loading EMF models

- In our action, we get the selected UML file through the selection:
  - `(IFile) ((IStructuredSelection)selection).getFirstElement();`

- A `.uml` file contains a serialized UML model in XMI format. The specific XMI format is the one defined by EMF.
- Therefore, we can use EMF deserialization classes to load the model back into memory.
- We need the EMF classes to do this, so add the EMF plugins as dependencies
  - `org.eclipse.emf`
  - `org.eclipse.emf.ecore.xmi`

- EMF serialization puts objects into **resources** (basically, files).
- A resource loads data from a file, the resource can then be queried for the actual model.

Resource Contents

- Load the EMF resource
  - `ResourceSet set= new ResourceSetImpl();`
  - `Resource resource= set.getResource(URI.createFileURI(theUMLFile.getLocation().toString()), true);`
- Get actual model elements:
  - `EList<EObject> contents= resource.getContents();`
  - or
  - `TreeIterator<EObject> allContents= resource.getAllContents();`
- The contents of an EMF resource are `EObjects` which are related to one another (by associations, compositions, ...).
- They belong to an EMF metamodel, which is defined somewhere and registered with the EMF core
- In our case, we need the UML2 meta model to deal with these `EObjects`.

UML2 Meta Model

- To use the UML2 EMF metamodel, add a dependency:
  - `org.eclipse.uml2.uml`
- You can also import the model with Import > Plug-Ins and Fragments.

- The metamodel defines `EObjects` like Activity, Class, Pin, ...
- To use these, cast the `EObjects` to the actual classes, e.g.

```
If (someObject instanceof Activity)
  doSomethingWith((Activity)someObject);
```

- We are interested in UML Activities (the UML4SOA base construct for modelling orchestrations). Collect these, then
- **Perform Problem Analysis**
- This is done by querying the in-memory EMF model.

- **Note**. There are other metamodels written in EMF for many purposes and languages – check Google before creating your own.

Querying EMF models #1

- EMF model objects are all based on the `EObject` root class
- They contain dynamic query/manipulation methods and (if a concrete type is available) static query/manipulation methods

- Dynamic Methods are similar to the Java Reflection mechanism. They start with “e”, for example:
  - `eContainer()`
  - `eContents()`
  - `eCrossReferences()`
- Static Methods are directly based on the meta model, for example in UML2 ActivityNodes:
  - `getIncomingEdges()`
  - `getOutgoingEdges()`
  - `getName()`
  - `...`
- We’ll use the static methods, as they are easier to read.

Querying EMF models #2

- Static querying and manipulation methods are based on the models' metamodel, for example, the UML2 meta model (MOF).
- They are named follow a certain pattern
  - For simple attributes, there are getter and setter methods
  - For associations and compositions, the list of elements can be retrieved with a getter and then manipulated directly
- The available methods are therefore easily inferred from the meta model (and, of course, from syntax completion).
- Example: Querying UML2 activity nodes for outgoing edges:

```
if (object instanceof ActivityNode) {
  List<ActivityEdge> edges=
            ((ActivityNode)object).getOutgoings();
  for (ActivityEdge activityEdge : outgoings) {
    ...
```

Reporting

- We'd like to report on "problems" found in the UML2 activities.
- In particular, we're interested in well-nestedness, i.e.
  - Nodes which have several outgoing edges, but are no decision/merge nodes,
  - Nodes which have several incoming edges, but are no decision/merge nodes,
  - cycles not involving decision/merge nodes
  - …
- We can get this information easily using the UML2 objects

- We'll then put them into a `ProblemReport` for reporting
- A `ProblemReport` contains several `Problems`. A problem consists of
  - A problem description ("what is wrong?")
  - The node causing the problem
  - The activity the node belongs to
  - The trace (from the beginning of the activity) to the problematic node

- We can then show the `ProblemReport` to the user.

Show the report: Eclipse UI Integration

- For showing the report in the UI, we have several options
- We can use
  - **Dialogs,** blocking user input until dealt with
  - **Editors**, for example for a specific file type, or a non-file-based editor. Editors support the concept of “dirtiness”, save, saveAs etc.
  - **Views**, which is displayed in certain perspectives and is “dockable” to each side of the workbench
  - ...other mechanisms...
- In our case, we’d like to add a new view **which** shows the problems
- A view is better than a dialog as it does not block user actions
- We do not have a concept of dirtiness here

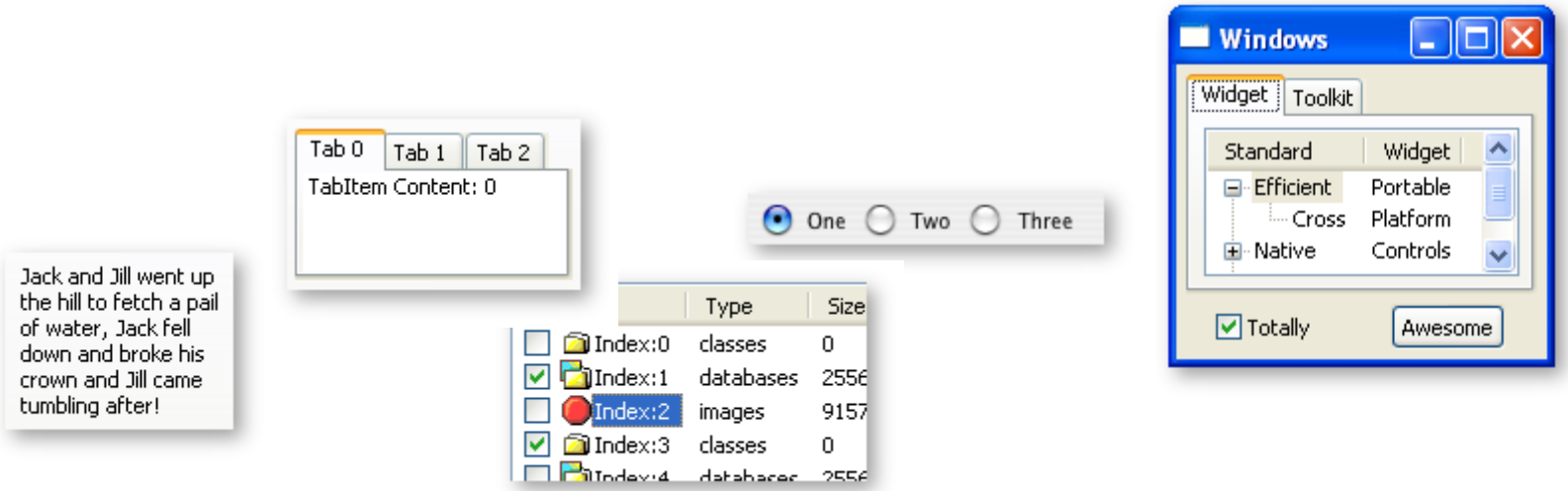
Adding a view

- To add a new view, we need to extend the workbench (again)
- In `MANIFEST.MF/plugin.xml`, select the **Extensions** tab
- Add a new `org.eclipse.ui.views` extension without a template.

- Use Runtime Workbench to test the new view
  - Use `Window > Show View` to test the view
  - Should be empty

- Now, the view should display the `ProblemReport`
- I.e., have a table which displays details about each problem that was found

- The Eclipse platform is based on SWT (The Standard Widget Toolkit) and JFace for its UI
- SWT is a widget toolkit similar to Swing, but more directly based on native widgets of the underlying operating system
- JFace is built on top of SWT and adds more advanced controls like `TableViewers`, `ListViewers`, `TreeViewers` etc.



TableViewer #1

- We would like to display a table with several columns to indicate the problem, the node which caused the problem, the trace to the node etc.
- A JFace `TableViewer` displays such a complete table in several `TableColumns`. It uses two additional classes for display:
  - A `ContentProvider`, which provides the content (i.e. rows) for the table
  - A `LabelProvider`, which provides the labels for each row and each column of the row based on the content.

- Adding a TableViewer with some columns:

```
fProblemViewer= new TableViewer(parent, ...)
fProblemViewer.setContentProvider(new ViewContentProvider());
fProblemViewer.setLabelProvider(new ViewLabelProvider());

TableColumn col1= new TableColumn(...);
col1.setText("Problem"); col1.setWidth(200);

TableColumn col2= new TableColumn(...);
col2.setText("Node"); col2.setWidth(200);
```

TableViewer #2

- Adding content to a table viewer (use sparingly...)
  - `viewer.setInput(someInput)`
- Content Display with
  - **IStructuredContentProvider**

```
public Object[] getElements(Object parent) {
    return ((ProblemReport) parent).getProblems().toArray();
}
```

  - **ITableLabelProvider**

```
public String getColumnText(Object obj, int index) {
    Problem p= (Problem) obj;
    switch (index) {
        case 0: ...
        case 1: ...
    }
    return null;
}
```

Binding action and view together

- Right now, we have the `ProblemReport` in our action
- We need to get it to be displayed in our view.

- Use the workbench to show the view:
  - ```
    ProblemView view= (ProblemView) getSite().getPage()
        .showView("umlproblemanalysis.views.ProblemView");
    ```

- Then, deliver the `ProblemReport`

  ```
  view.setProblemReport(report);
  ```

- The report should then be added to the viewer:

  ```
  fProblemViewer.setInput(report)
  ```

- **Done!**
  ```

Integration into the SDE

- The SDE is an integration platform for (headless) scripting and orchestration of several tools
- We therefore need to think about which functions of our tool might be of help in such an environment

- We might add the following functions:
  - Performing problem analysis, i.e. creating a `ProblemReport` from a set of activities.
  - Serializing the `ProblemReport` to a `String` for storage or later use
  - Showing the `ProblemReport` to the user

- The first two functions are headless, the last is not.

- To provide these functions to the SDE, we need to write a Facade class encapsulating our tool.

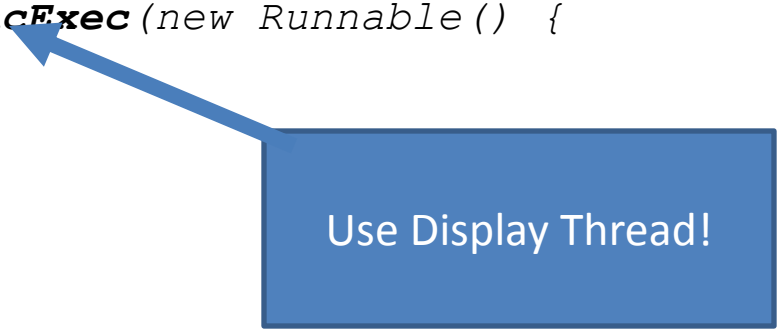
Writing a Facade class


```
public class UMLProblemAnalysisService {

    public ProblemReport analyseUMLList<Activity> activities) {
       return ProblemAnalysis.perform(activities);
    }

    public String serializeProblemReportProblemReport report) {
       return report.serialize();
    }

    public void showReportInUI(ProblemReport report) {
       Display.getDefault().asyncExec(new Runnable() {
         public void run() {
           //show view
         }
       }
    }
}
```



SDE Integration

- In order to use the Facade class as a tool within the SDE, we need to **annotate** it with additional metadata.
- The metadata is added through **Java annotations**.
- To use these annotations, we need to add the SDE plugin to our list of dependencies

```
eu.sensoria_ist.sde.core
```

- Add annotations to the class:

```
@SensoriaTool(name= "UML Problem Analysis Service",
  categories= "Analysis", description= "...")
```

- Add annotations to each method:

```
@SensoriaToolFunction(description= "...")
@SensoriaToolFunctionReturns(description= "A string")
```

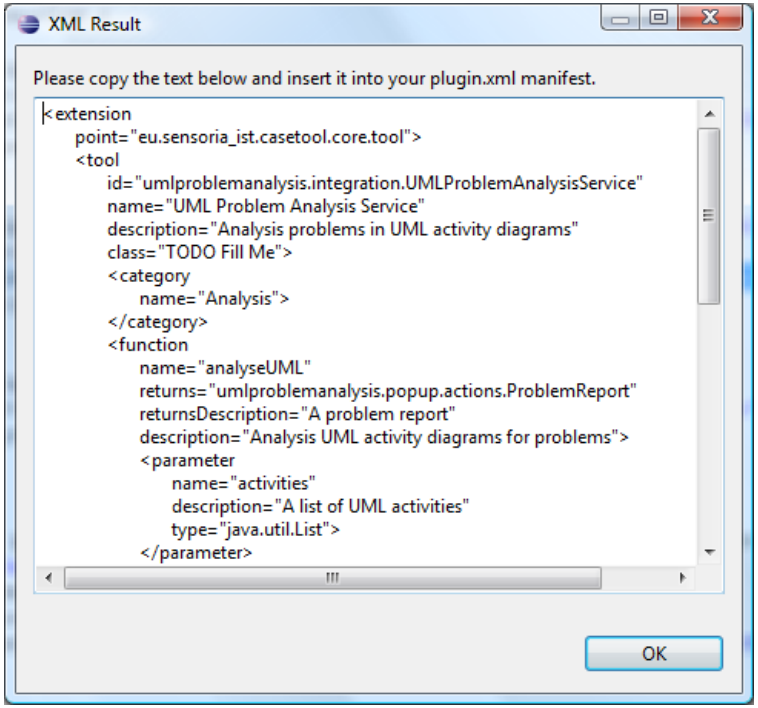
- And parameter

```
@SensoriaToolFunctionParameter(description= "...")
```

Adding the SDE extension

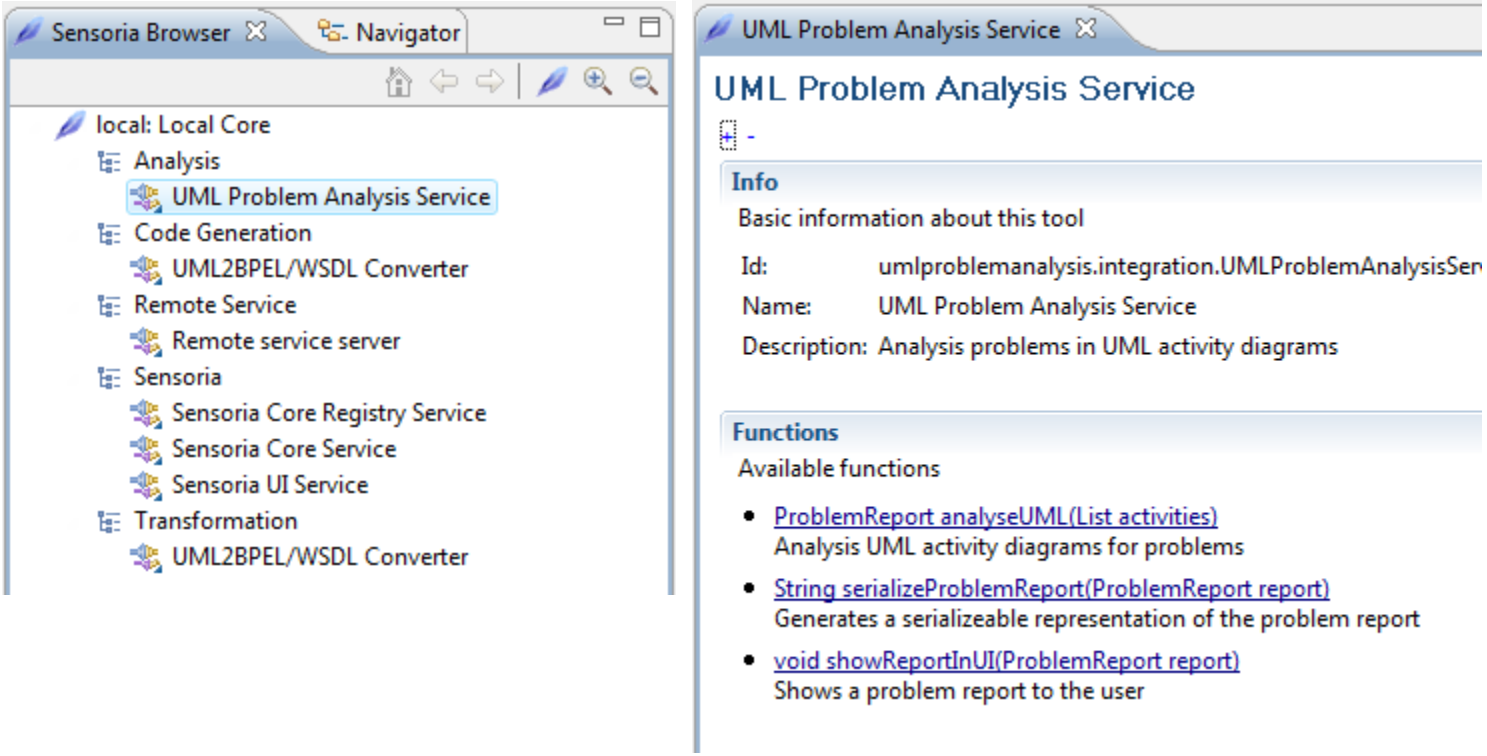
- Registering a tool with the SDE is the same process as adding an extension to Eclipse – i.e., the SDE provides an **extension point** for tools
- SDE contributions do not need to be written by hand, but can be generated from annotated classes
- To do so, right-click on the class in package explorer and select "Convert to Sensoria Tool".

- A dialog is shown with the complete extension as XML code
- The code can be pasted into `plugin.xml`.
- Don‘t forget to fill in the class name!



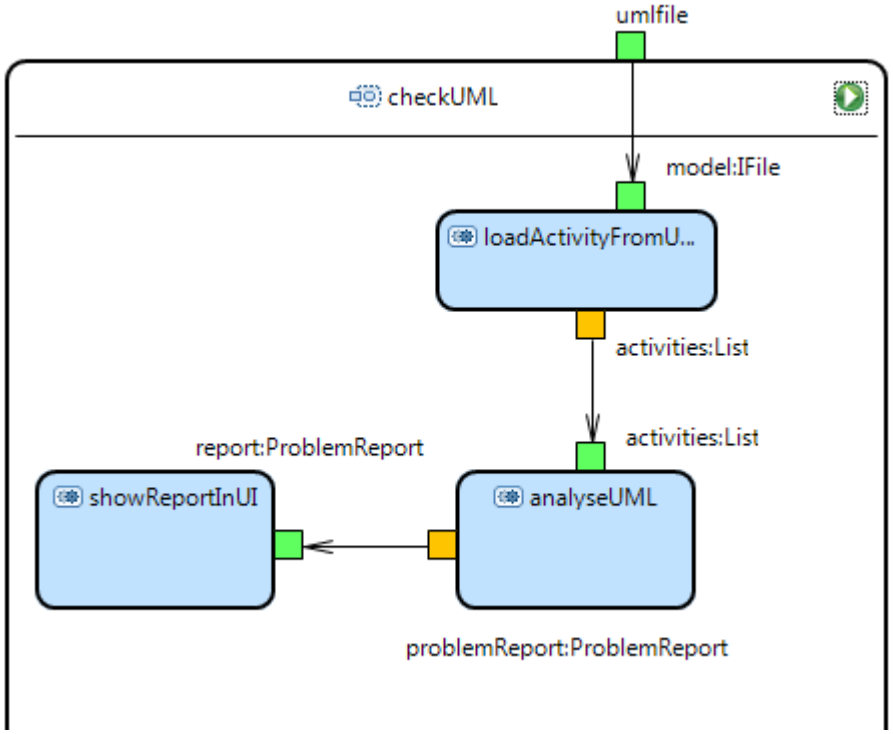
Finally...

- We can use the runtime workbench to check our contribution
- Direct invocation in the SDE UI is now possible



...and more

- We can also add our own functions to a graphical orchestration to create a complete workflow:



Thank You!