The Service Development Environment (SDE)

Version 4.4

Executive Summary

This document describes the Service Development Environment, which is intended to support the development of Service-Oriented Software by integrating various tools across development, analysis, and deployment of service-oriented software systems.

In this document, the current state of the Development Environment is reported, along with hints and tutorials for using the Development Environment as well as creating new tools for this platform, and recommended future steps.

Contents

1 About the Development Environment

The Service Development Environment is intended to support the development of Service-Oriented Software by integrating tools across the development, analysis, and deployment of service-oriented software systems.

1.1 Aim of the Development Environment

The main aim of the Service Development Environment is to **provide a service-oriented platform for development tool integration**. On this platform,

- tools are services, and provide arbitrary functionality
- tools can be used as-is, or combined using orchestration mechanisms
- tools can be published and discovered

By integrating them into the Service Development Environment, tools become available to a broader user range and in a larger context, and are thus more usable by developers.

In the view of the Service Development Environment, the tools each consist of functions, which can be invoked in the Development Environment with or without User Interface (UI). The UI is not necessarily tied to a specific function, but can also be provided in a cross-function way. Tools are easy to write, add, and remove. Accessing remote or legacy applications is possible.

To enable composition, tools are intended to provide an Application Programming Interface (API) allowing tool orchestration with arbitrary orchestration languages. Included within the SDE is a graphical orchestration with data-driven activity diagrams and a JavaScript orchestrator.

1.2 High-Level Overview

As a tool for developing, analyzing, and deploying service-oriented software systems, the Development Environment must feature an IDE-like UI, while at the same time keeping requirements for tool builders low as not to hinder integration of such tools. To allow this, the Service Development Environment is itself built in a service-oriented way on top of the OSGi platform. The graphical part of the Development Environment and contributed tools – if available – is built on Eclipse technology.

This architecture is laid out in Figure 1.

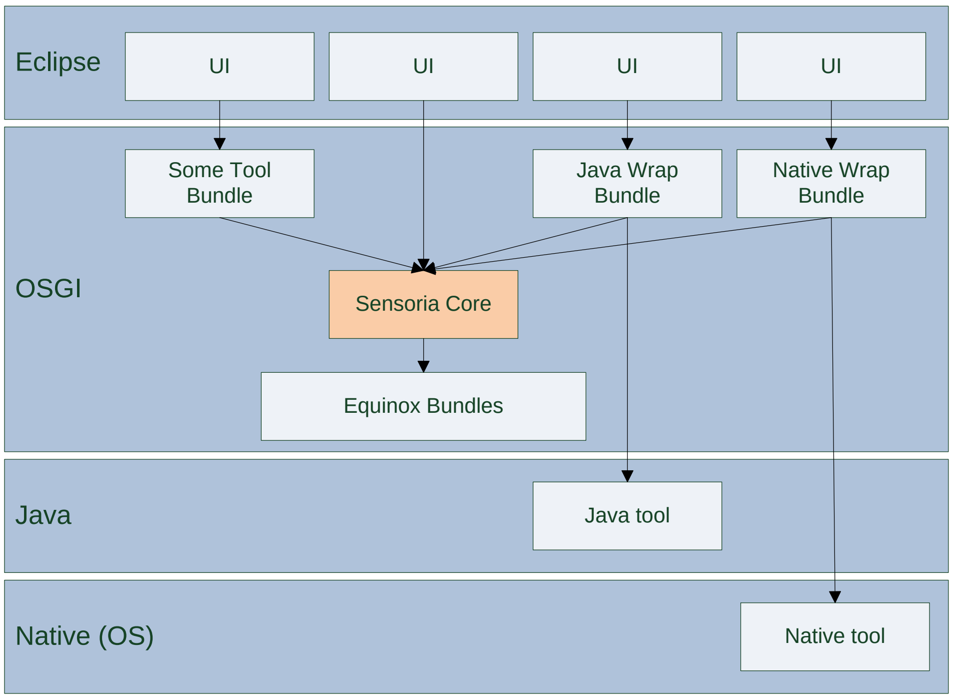


Figure 1: Service Development Environment architecture

As can be seen in the figure, the Development Environment core and the tools are based on OSGi only (or, more specifically, the Equinox implementation of OSGi). Tools may use existing Java implementations or native code as they wish. Being only based on OSGi, they can be invoked completely independently from Eclipse. If they additionally choose to provide a UI, this UI is integrated into and based on the Eclipse platform, as is the UI for the SDE core itself.

1.3 Basic Concepts of the Development Environment

As outlined above, the Development Environment provides an environment for using and orchestrating **tools**. The tools themselves are provided by various manufacturers and in general must be installed separately; however, some are already distributed with the Development Environment as examples.

The basic functionality of the core is as threefold:

- It provides access to all the registered tools by an API (which can be seen as a basic discovery service) and by a UI. The API allows retrieving tools based on their ID or name and is intended to be used from within Java, while the UI allows graphical browsing of registered tools directly for the end user.
- It provides access to the tool functions by API and by UI as well. The API allows calling arbitrary functions on the registered tools from within Java, while the user interface provides a generic UI for executing these functions, storing the results, and re-using results as input for other functions.
- It provides an orchestration mechanism using a) activity diagrams and b) JavaScript. Using such orchestrations, the API discussed in the previous two points can be accessed and tools can be orchestrated in a simple way.

For most tools, integration into the Service Development Environment is only one way of providing users with access to their functionality. Thus, the Service Development Environment only imposes a minimum number of requirements on tool writers. In particular, tools can be used in any of the following three ways by the user in the Development Environment:

- By using the generic wizards outlined above.
- By entering commands in a manual orchestrator like the SDE Shell.

- By using UIs provided by the tools themselves, which are independent of the Development Environment UI.

Besides simply using their functionality, tools can also be orchestrated with partial help from the platform. This can be achieved by different means as well. In particular, the following scenarios are envisioned:

- Orchestration using the built-in shell
- Orchestration using Java, i.e. within other tools
- Orchestration by using JavaScript-based tools
- Orchestration by using the graphical orchestration inside SDE

Tools to be used as part of the Development Environment must be implemented as OSGi bundles and contain a declarative description of their functionality but are otherwise unlimited in their implementation. In particular,

- tools may be written in Java and may consist of an arbitrary number of libraries, other Eclipse plug-ins, or external code
- tools may also wrap native code, thereby providing an interface to non-Java software
- tools may include functionality for calling remove services, thereby providing the link to Web services

2 Installing the Development Environment

As the Development Environment features both an OSGi and an (Eclipse)-UI component, the recommended way is to install it onto the Eclipse platform, which will be detailed here.

2.1 Requirements

The Development Environment is built on cutting-edge technology. It requires two components to be installed:

- **Java JDK 1.6**. Former versions, including 1.5, will NOT work as the Development Environment uses the Java 6 Scripting Engine. Note that you will need the JDK, not only the JRE. The JDK can be downloaded from http://java.sun.com/javase/downloads/index.jsp.
- **Eclipse version 3.4**. The newest version of Eclipse 3.4.X can be downloaded from http://download.eclipse.org/eclipse/downloads/.

It is recommended that the Java VM to use is explicitly specified when running Eclipse. This is achieved with the -vm command line argument (for example, -vm c:\jre\bin\javaw.exe). Without -vm, Eclipse will use the first Java VM found on the O/S path. Also, within Eclipse, the right JDK must be selected in the preferences before starting a Runtime Workbench.

The Java JDK 1.6 must be selected in Eclipse (Window > Preferences > Java > Installed JREs...) as the default.

2.2 Installing the Tool

2.2.1 Overview

Once Eclipse has been installed, the Development Environment core and many tools can be installed via update sites. The main update site for the Development Environment core is

`http://svn.pst.ifi.lmu.de/update/sde/`

This update site contains two features:

- The core itself
- The development feature which eases development of new SDE tools

If you are familiar with Eclipse update sites, simply point Eclipse to this URL and download all the items provided. A more detailed explanation can be found in the next section.

2.2.2 Detailed Steps

Begin the installation by selecting the following menu path from the Eclipse main menu:

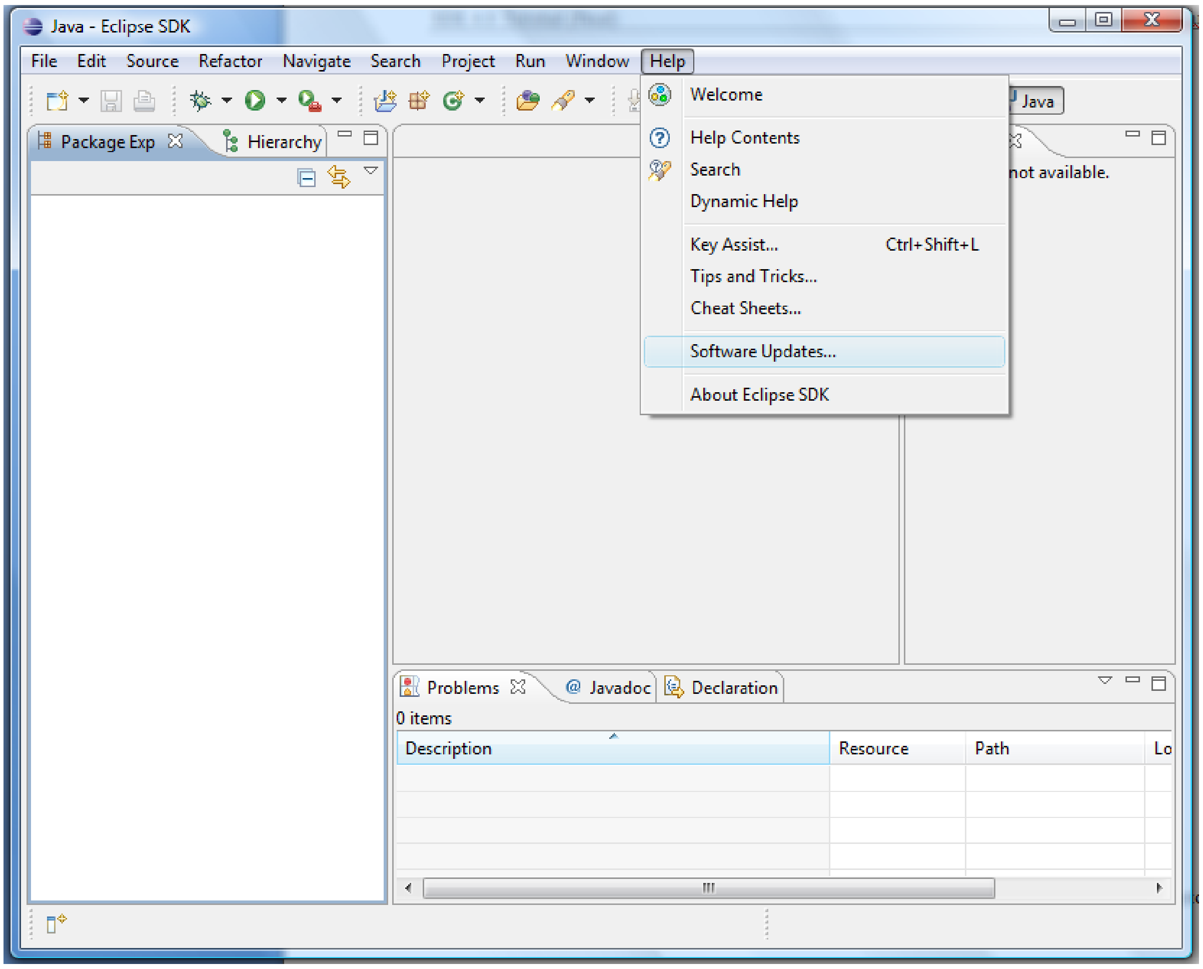
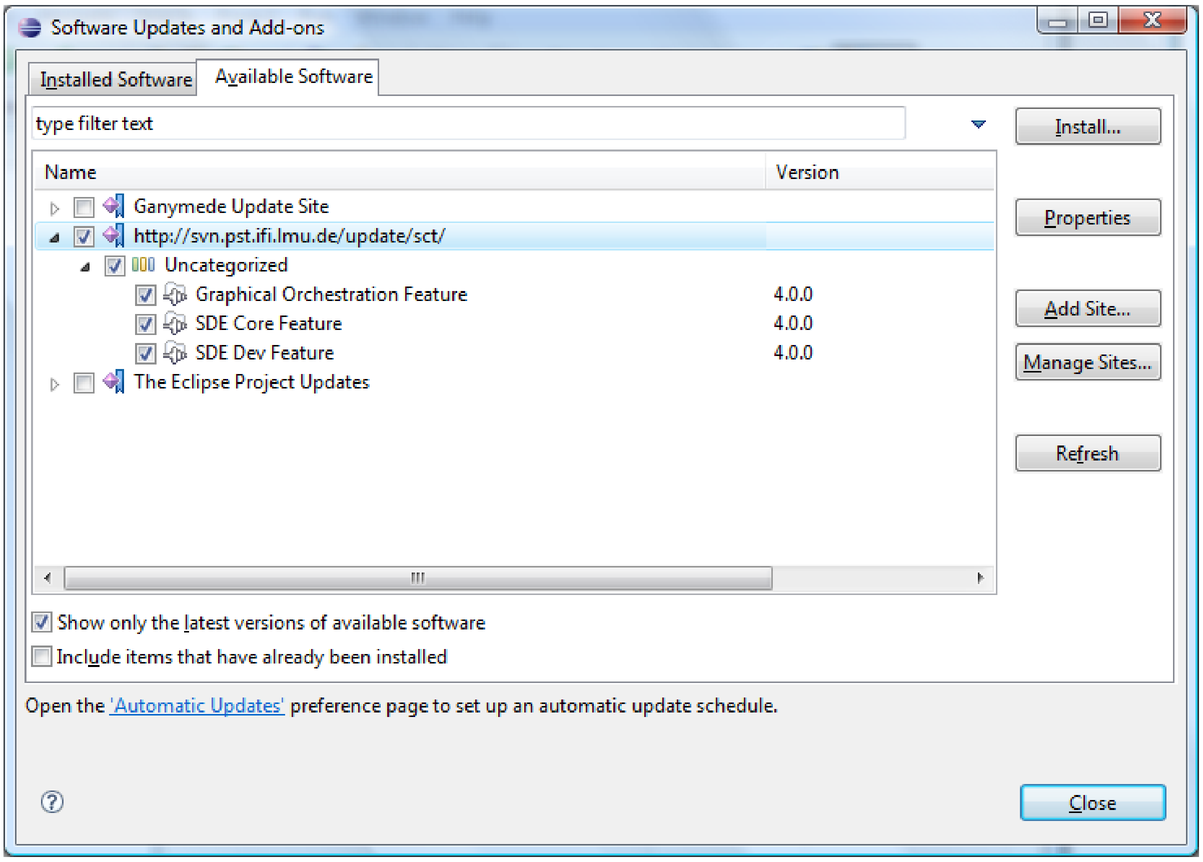


Figure 2: Selecting Software Updates

A dialog will show up with two tabs: **Installed Software** and **Available Software**. Select the **Available Software** tab. Then, click **Add Site...**, and enter the following URL:

`http://svn.pst.ifi.lmu.de/update/sde/`

Select OK. The dialog now shows the new update site. Select the complete site as shown in the next figure.


Software Updates and Add-ons

Installed Software | Available Software

type filter text

Name	Version
☐ Ganymede Update Site	
☑ http://svn.pst.ifi.lmu.de/update/sct/	
☑ Uncategorized	
☑ Graphical Orchestration Feature	4.0.0
☑ SDE Core Feature	4.0.0
☑ SDE Dev Feature	4.0.0
☐ The Eclipse Project Updates	

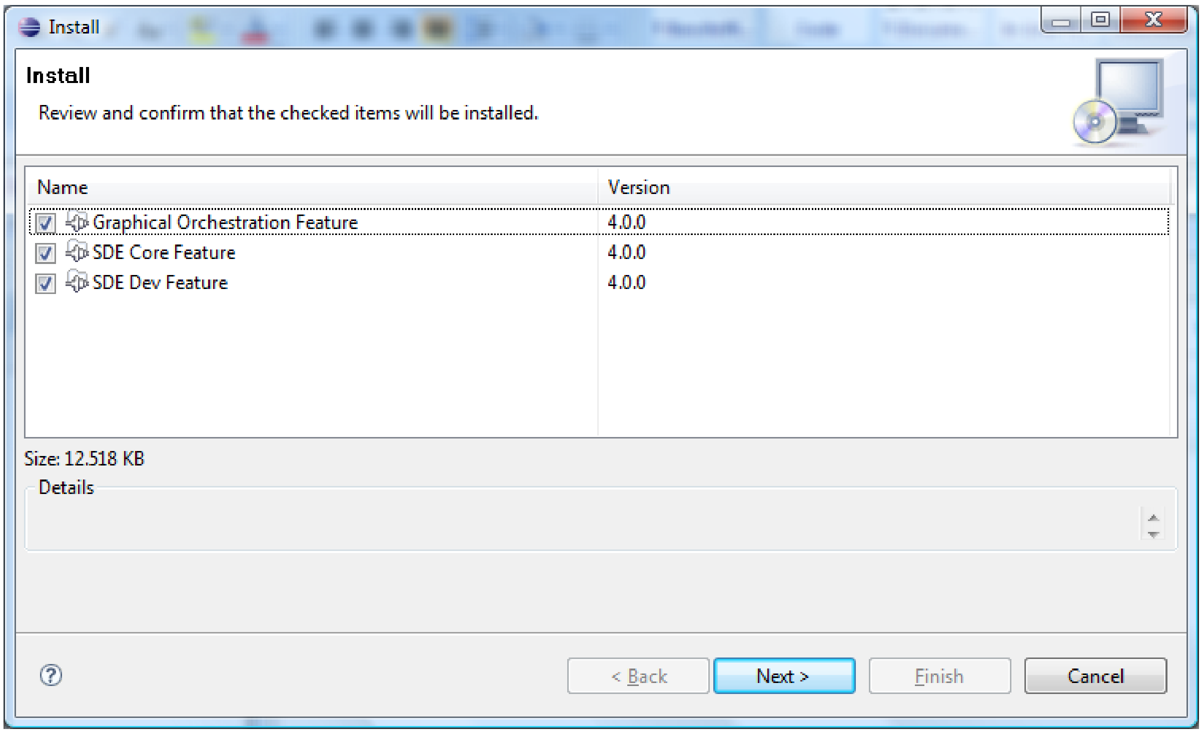
☑ Show only the latest versions of available software

☐ Include items that have already been installed

Open the 'Automatic Updates' preference page to set up an automatic update schedule.

Figure 3: Update Sites in Eclipse

Now select **Install...**. After a while, the following dialog is shown:


Install

Install

Review and confirm that the checked items will be installed.

Name	Version
☑ Graphical Orchestration Feature	4.0.0
☑ SDE Core Feature	4.0.0
☑ SDE Dev Feature	4.0.0

Size: 12.518 KB

Details

Figure 4: Finishing installation

You will be presented with more dialogs for accepting the license terms, agreeing to installing unsigned features, and finally restarting the workbench. Simply follow the steps until the workbench has rebooted.

3 User Guide

First and foremost, the Service Development Environment is an IDE extension to be used by developers for creating, analyzing, and deploying Service-Oriented Software Systems by providing access to tools. This user guide explains the meta functionality provided by the Development Environment core itself, and also contains examples for using installed tools.

3.1 Introduction

The Service Development Environment provides one new perspective, two new views, and one editor to the Eclipse platform. In addition, the manual orchestrator provided with the core (called the SDE Shell) provides an additional view and a launcher for executing Shell scripts; the graphical orchestration provides another editor and toolbar for creating graphical orchestrations.

Having installed the Development Environment core as outlined in the previous section, you can select the newly provided SDE perspective to display the contributed views. To do this, open the perspective using the **Window** menu:

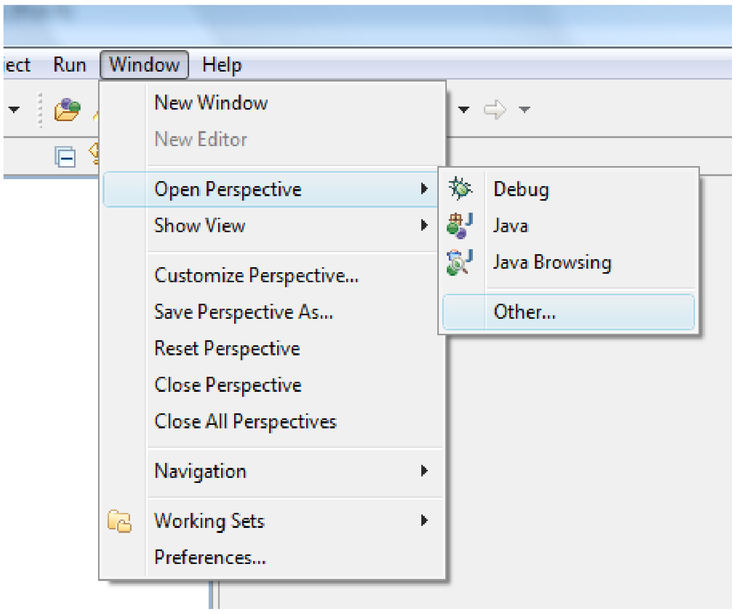


Figure 5: Opening the SDE perspective, take 1

In the following dialog, select the SDE perspective:

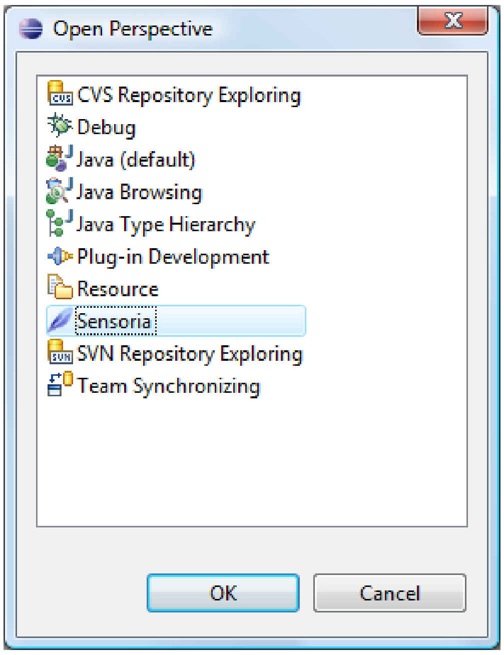
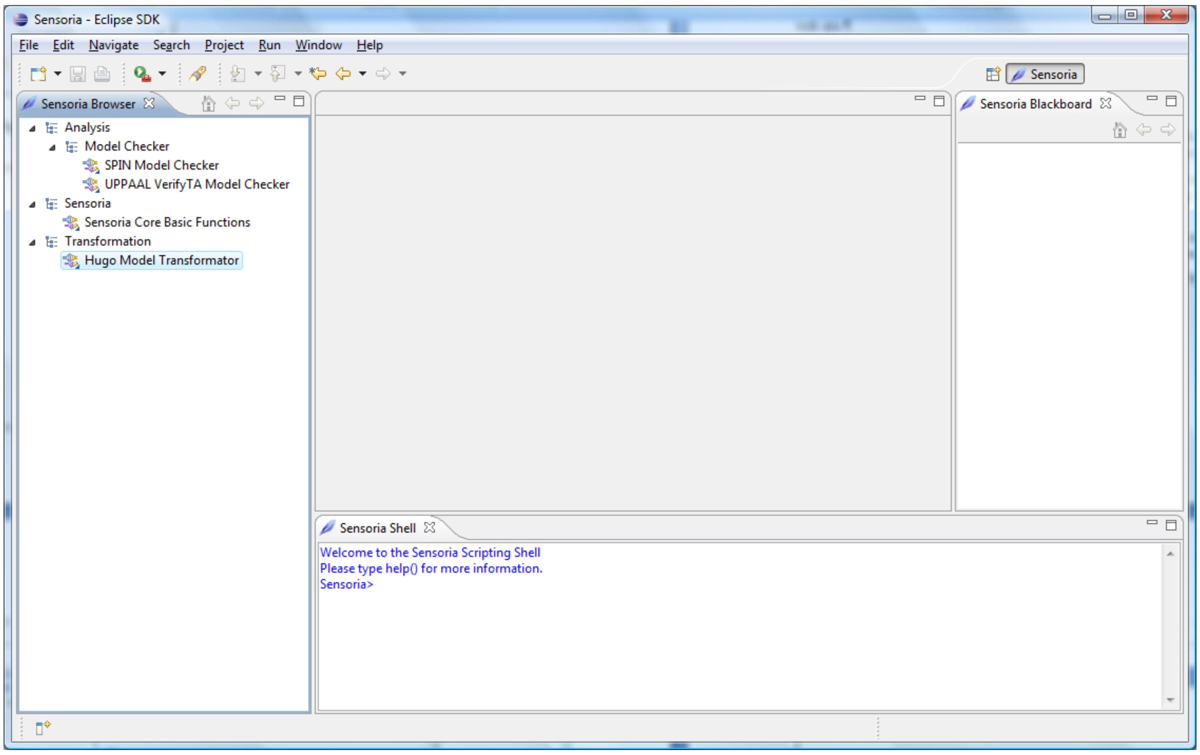


Figure 6: Opening the SDE perspective, take 2

Once opened, the perspective provides you with access to the complete functionality of the Service Development Environment.

3.2 SDE Perspective

In its initial form, the Service Development Environment perspective has the following layout. Note that in the following screenshot, some tools have already been installed by using other update sites available on the SDE web site.



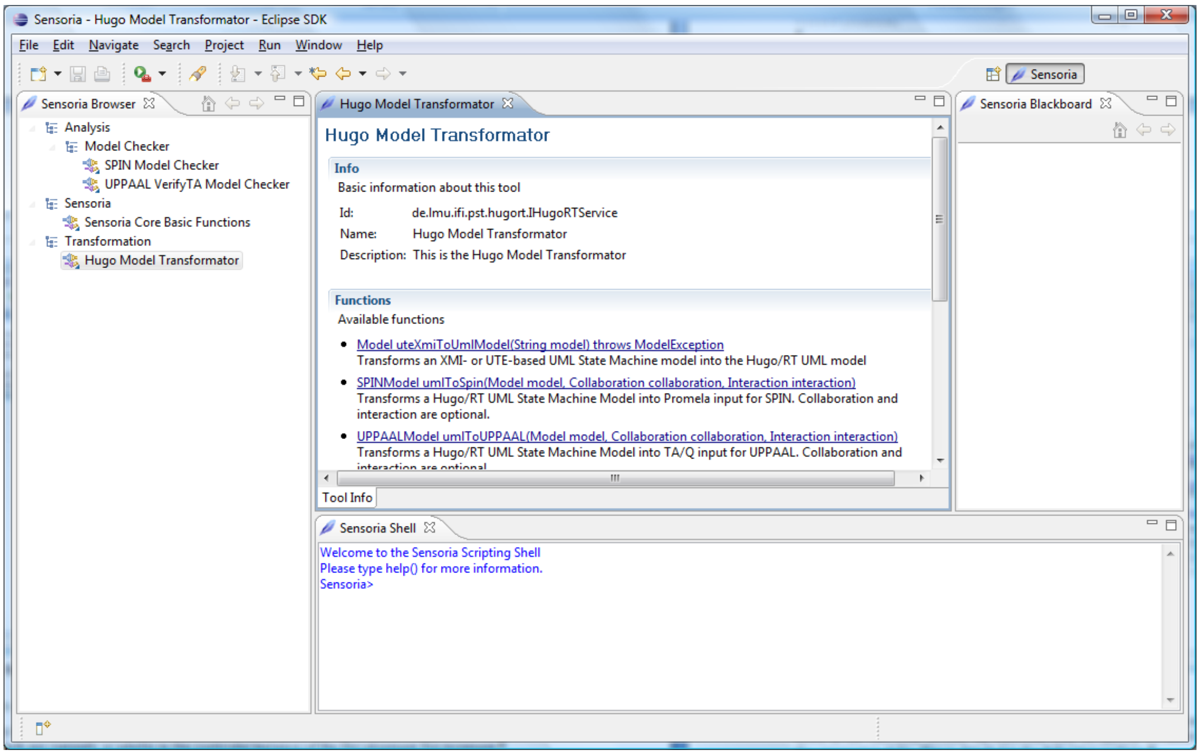
```
Welcome to the Sensoria Scripting Shell
Please type help() for more information.
Sensoria>
```

Figure 7: SDE perspective

Three views are visible:

- On the left-hand side, the **SDE Browser** is displayed. It contains a categorized listing of all tools which are currently available in this particular instance of the Development Environment.
- On the right-hand side, the **SDE Blackboard** is displayed. The blackboard is used to store Java object values in-between service invocations when using the manual generic UI to access tool functions.
- At the bottom, the **SDE Shell** is displayed. As pointed out above, the Shell is a manual orchestrator which can be used to employ JavaScript to call tool functions.

Double-clicking on a tool in the SDE Browser displays more information about the tool, for example the functions of the tool Hugo/RT.



Hugo Model Transformator

Info

Basic information about this tool

Id: de.lmu.ifi.pst.hugort.IHugoRTService

Name: Hugo Model Transformator

Description: This is the Hugo Model Transformator

Functions

Available functions

- Model uteXmiToUmlModel(String model) throws ModelException
  Transforms an XMI- or UTE-based UML State Machine model into the Hugo/RT UML model
- SPINModel umlToSpin(Model model, Collaboration collaboration, Interaction interaction)
  Transforms a Hugo/RT UML State Machine Model into Promela input for SPIN. Collaboration and interaction are optional.
- UPPAALModel umlToUPPAAL(Model model, Collaboration collaboration, Interaction interaction)
  Transforms a Hugo/RT UML State Machine Model into TA/Q input for UPPAAL. Collaboration and interaction are optional.

```
Welcome to the Sensoria Scripting Shell
Please type help() for more information.
Sensoria>
```

Figure 8: Hugo/RT

Besides some general information about the tool in the upper section, all available functions of the tool are listed in the functions section. All functions may be directly invoked using the generic wizard UI by selecting the appropriate links. This is detailed in the following sections.

Besides functions, a tool may also provide options. Options are necessary for example if the tool is actually a wrapper for some external tool like a Web services, which is installed at some particular URL. If available, options are displayed in another section beneath the functions section.

3.3 Using the Generic Wizard UI and the Blackboard

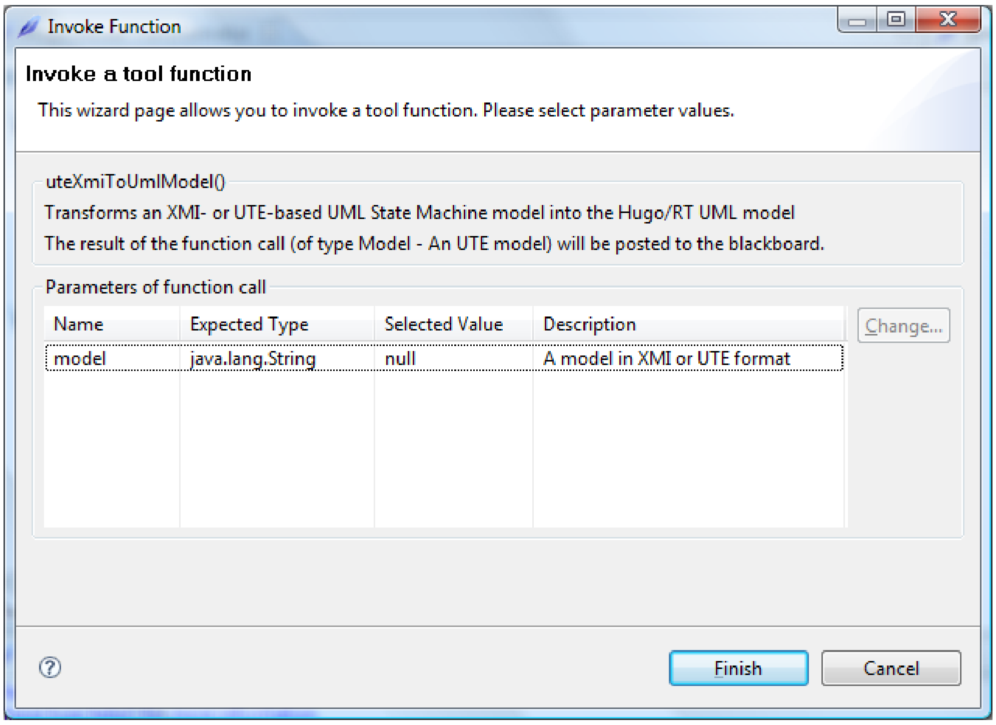
As pointed out in chapter 2, there are various ways for invoking tool functions. One way is using the generic wizard UI. This UI allows calling arbitrary functions of arbitrary tools, providing input to those functions from files, strings, or from the blackboard, and posting of the result of the invocation on the blackboard as well.

For example, consider the function `uteXmiToUmlModel()` of the Hugo/RT tool:

```
Model uteXmiToUmlModel(String) throws ModelException
```

This function transforms an XMI- or UTE-based UML State Machine model into a Hugo/RT UML model. The model is returned as a Java object of class `Model`; the function requires the UTE or XML specification to be given as a `String`.

Invoking the generic wizard on this function yields the following dialog:


Invoke Function

Invoke a tool function

This wizard page allows you to invoke a tool function. Please select parameter values.

uteXmiToUmlModel()

Transforms an XMI- or UTE-based UML State Machine model into the Hugo/RT UML model

The result of the function call (of type Model - An UTE model) will be posted to the blackboard.

Parameters of function call

Name	Expected Type	Selected Value	Description
model	java.lang.String	null	A model in XMI or UTE format

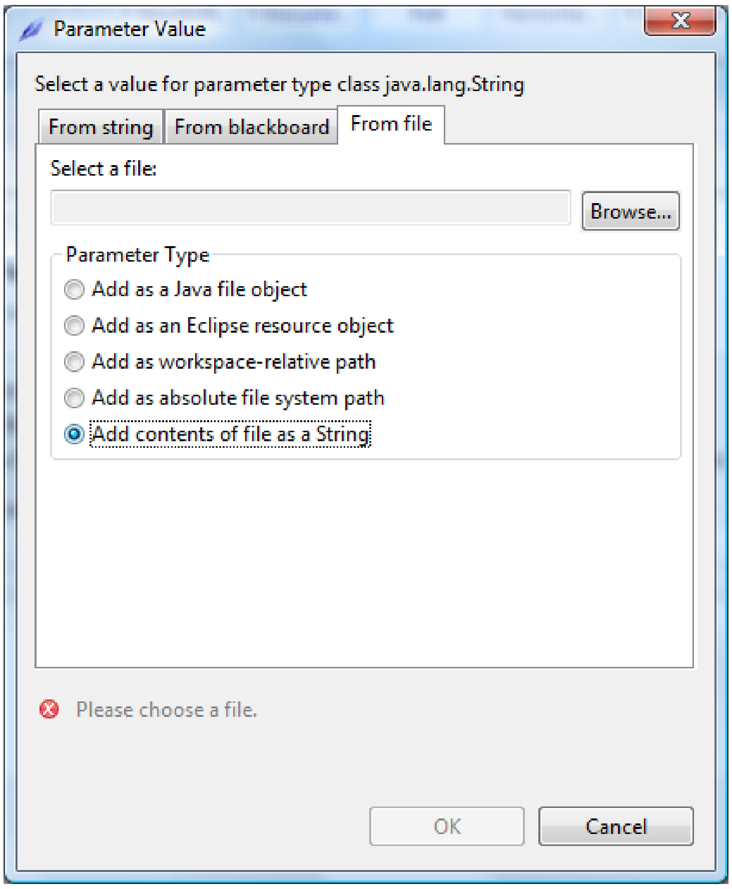
Figure 9: Invoking a function using the generic wizard UI

There is one parameter to be provided. By selecting the parameter and clicking **Change**, another dialog pops up which allows choosing a value for the parameter.

In this dialog, you have three choices for selecting a parameter value:

- You can simply provide text in a text field. This is only possible for String-typed parameters.
- You can select a value from the blackboard. This is possible for all kinds of parameter types.
- You can load the input for the parameter from a file.

In this particular example, the UTE or XMI model will probably be loaded from a file, so the dialog is used to select a file in the workspace. As the expected type is a String, we will select "Add contents of file as String".


Parameter Value

Select a value for parameter type class java.lang.String

From string | From blackboard | From file

Select a file:

Browse...

Parameter Type

- Add as a Java file object
- Add as an Eclipse resource object
- Add as workspace-relative path
- Add as absolute file system path
- [✓ Add contents of file as a String]

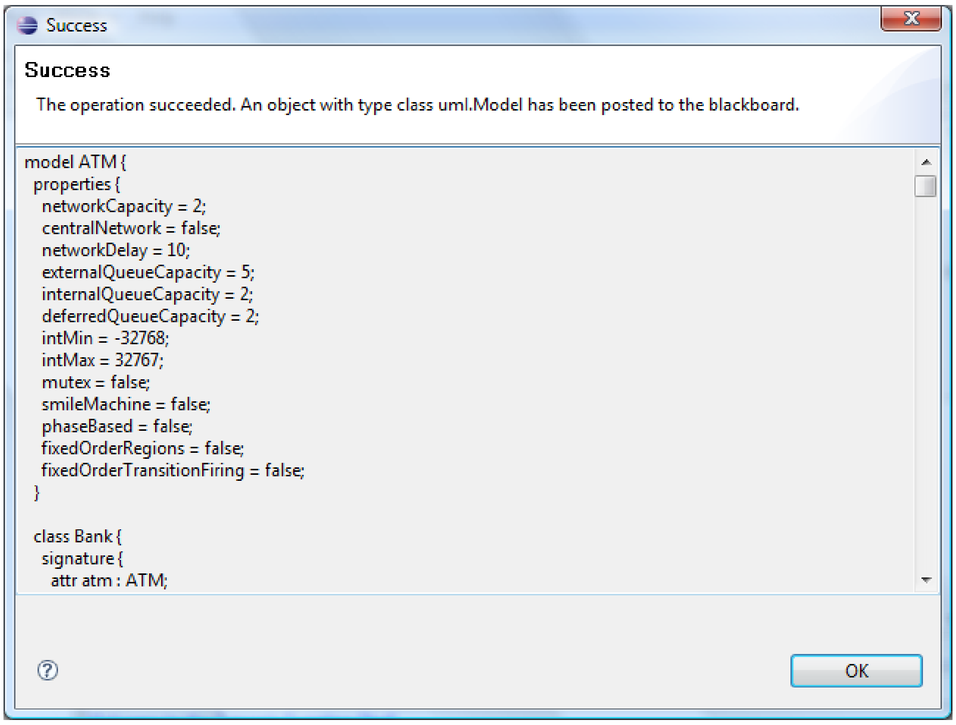
Please choose a file.

OK | Cancel

Figure 10: Choosing a file as a parameter

After executing a function, the result is either opened in the UI (for example, in an editor or a separate view), or posted on the blackboard for further use.

In this example, the result is shown and posted to the blackboard.


Success

Success

The operation succeeded. An object with type class uml.Model has been posted to the blackboard.

```
model ATM {
 properties {
  networkCapacity = 2;
  centralNetwork = false;
  networkDelay = 10;
  externalQueueCapacity = 5;
  internalQueueCapacity = 2;
  deferredQueueCapacity = 2;
  intMin = -32768;
  intMax = 32767;
  mutex = false;
  smileMachine = false;
  phaseBased = false;
  fixedOrderRegions = false;
  fixedOrderTransitionFiring = false;
 }

 class Bank {
  signature {
   attr atm : ATM;
```

Figure 11: A successful call

As can be seen in the following figure, objects on the blackboard are sorted by providing tool. An object is shown with its name and its class. With tools, objects are sorted by date.

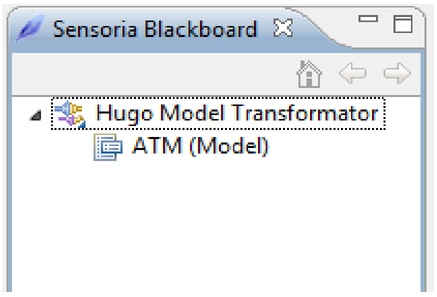
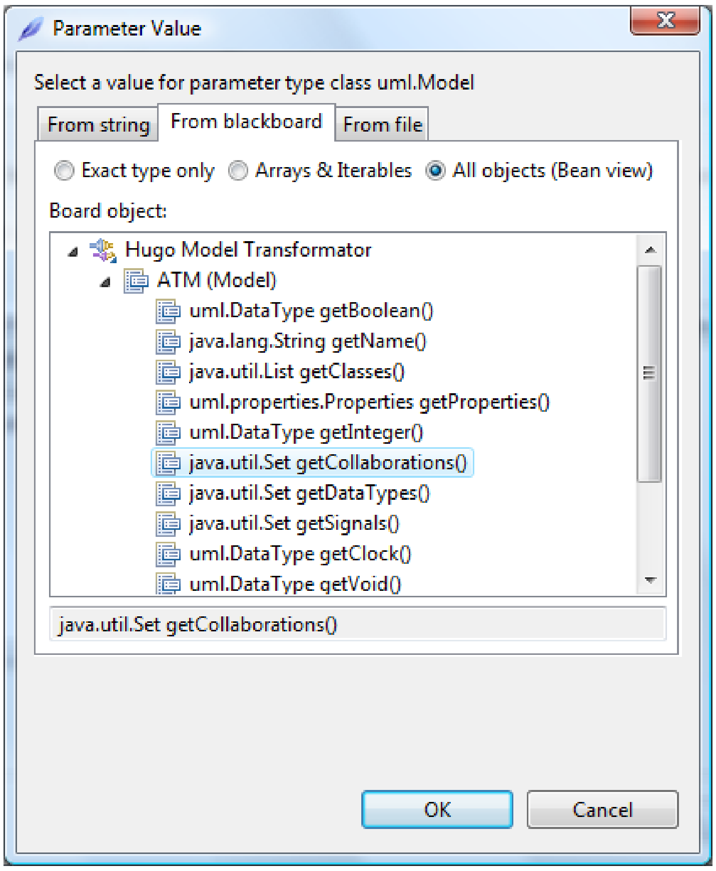


Figure 12: An object on the blackboard

When invoking another function which takes an `uml.Model` as input, this object can now be readily retrieved from the blackboard. The generic invocation wizard also allows more generic access to Java objects by providing a **Bean view**, i.e. allowing not only the blackboard objects themselves to be used as parameters but also objects returned by invoking zero-argument methods on these objects. The following screenshot shows an example of selecting such a method.


Parameter Value

Select a value for parameter type class uml.Model

Exact type only [Arrays & Iterables] [✓ All objects (Bean view)]

Board object:

```
Hugo Model Transformator
  ATM (Model)
    uml.DataType getBoolean()
    java.lang.String getName()
    java.util.List getClasses()
    uml.properties.Properties getProperties()
    uml.DataType getInteger()
    java.util.Set getCollaborations()
    java.util.Set getDataTypes()
    java.util.Set getSignals()
    uml.DataType getClock()
    uml.DataType getVoid()
```

java.util.Set getCollaborations()

Figure 13: Selecting a bean from the blackboard

To summarize, the generic wizard invocation UI along with the blackboard can be used to invoke tool functions with, or without a UI of their own.

3.4 Using the Orchestration Functionality

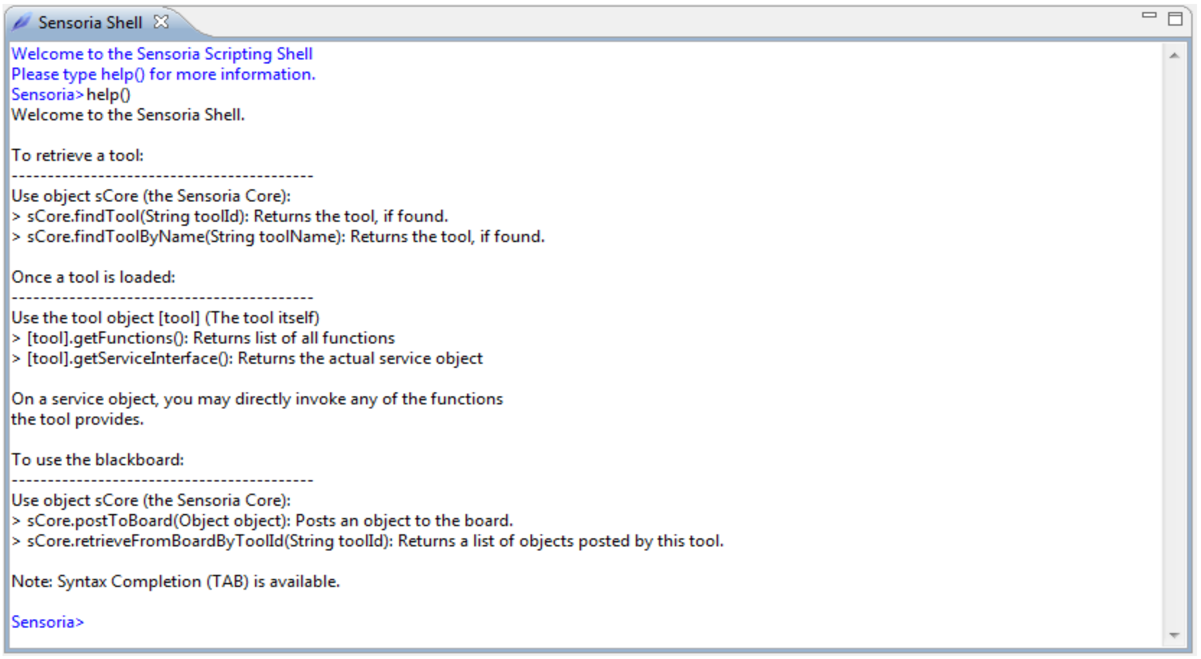
One of the main aims of the Development Environment is enabling orchestration of tools. As tools can be discovered using the SDE core and their interface functions invoked, orchestration can be provided by arbitrary tools on top of this service layer. The SDE provides three options for orchestrating tools:

- The **SDE Shell** provided within the core is a manual orchestrator which provides such an orchestration mechanism as a UNIX-like Shell with the additional ability to store, load, and execute scripts.
- **SDE Scripts** are written in JavaScript and are executed without parameterization. In order to create new tools, **SDE Tool Scripts** may be written which contain JavaScript functions. After converting them to tools, they can be used as any other tool through the tool browser.
- A graphical editor is provided to write **Graphical Orchestrations** as data-driven activity diagrams. Such orchestrations are again tools themselves.

In all orchestrations, the SDE core can be accessed directly; thus any tool can be retrieved from the core and its interface functions executed.

3.4.1 Orchestrating with the Shell

The SDE Shell is a manual orchestrator, i.e. it is intended for working with direct user input like a UNIX-like shell. A help function is provided which lists some of the available functionality. The following figure shows the SDE Shell with the help function invoked.

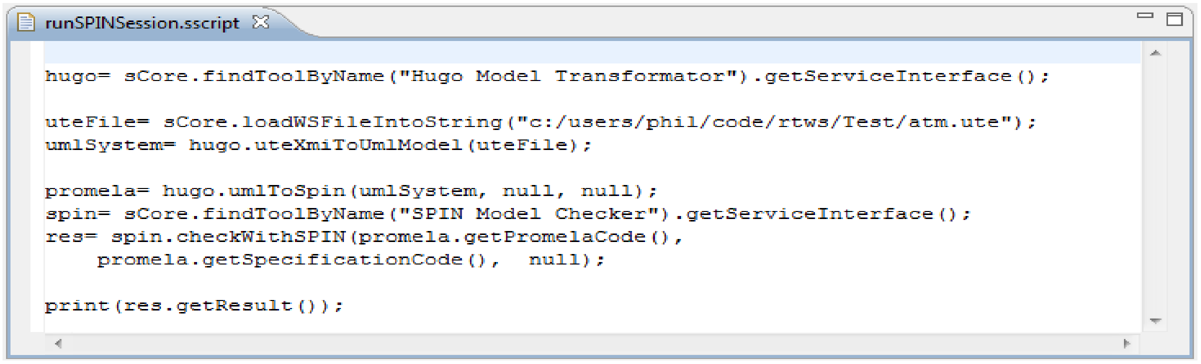


```
Welcome to the Sensoria Scripting Shell
Please type help() for more information.
Sensoria>help()
Welcome to the Sensoria Shell.

To retrieve a tool:
------------------------------------------
Use object sCore (the Sensoria Core):
> sCore.findTool(String toolId): Returns the tool, if found.
> sCore.findToolByName(String toolName): Returns the tool, if found.

Once a tool is loaded:
------------------------------------------
Use the tool object [tool] (The tool itself)
> [tool].getFunctions(): Returns list of all functions
> [tool].getServiceInterface(): Returns the actual service object

On a service object, you may directly invoke any of the functions
the tool provides.

To use the blackboard:
------------------------------------------
Use object sCore (the Sensoria Core):
> sCore.postToBoard(Object object): Posts an object to the board.
> sCore.retrieveFromBoardByToolId(String toolId): Returns a list of objects posted by this tool.

Note: Syntax Completion (TAB) is available.

Sensoria>
```

Figure 14: The SDE Shell

As can be seen, the SDE core is provided through the object `sCore` which is always available in SDE Scripts. The interface of the core can also be viewed in the tool browser – it is provided as the tool with the name **SDE Core Functions**.

The shell supports a history (up arrow/down arrow) and basic syntax completion with the TAB key. As an example for using the Shell, have a look at the following script which uses the two tools Hugo and SPIN to model-check an UTE-based model and outputting the result. In the shell, each line must be entered separately.



```
hugo= sCore.findToolByName("Hugo Model Transformator").getServiceInterface();

uteFile= sCore.loadWSFileIntoString("c:/users/phil/code/rtws/Test/atm.ute");
umlSystem= hugo.uteXmiToUmlModel(uteFile);

promela= hugo.umlToSpin(umlSystem, null, null);
spin= sCore.findToolByName("SPIN Model Checker").getServiceInterface();
res= spin.checkWithSPIN(promela.getPromelaCode(),
    promela.getSpecificationCode(),  null);

print(res.getResult());
```

Figure 15: An SDE Script

3.4.2 Using SDE Scripts

Figure 15 has already shown script code within the Eclipse text editor. The recommended file ending for such scripts is `.sscript` for **SDE Script**. To write such a script from scratch, select **File > New > Other…** in the Eclipse main menu, and then **File** under the **General** section. A dialog appears which allows you to choose a file name, which should end with `.sscript`.

Once you have written and saved the script, you can invoke it by right-clicking the file in the Navigator or Package Explorer view, and selecting **Run As > SDE Shell Script**. Note that in both the Navigator and Package Explorer views, every `.sscript` file is annotated with a small green run button to show it is executable as a SDE script.

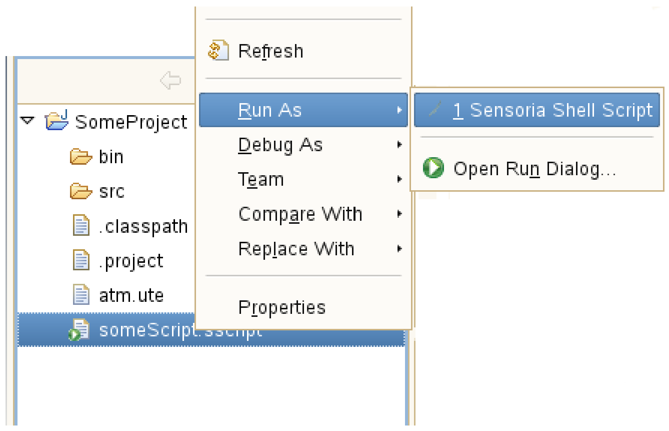
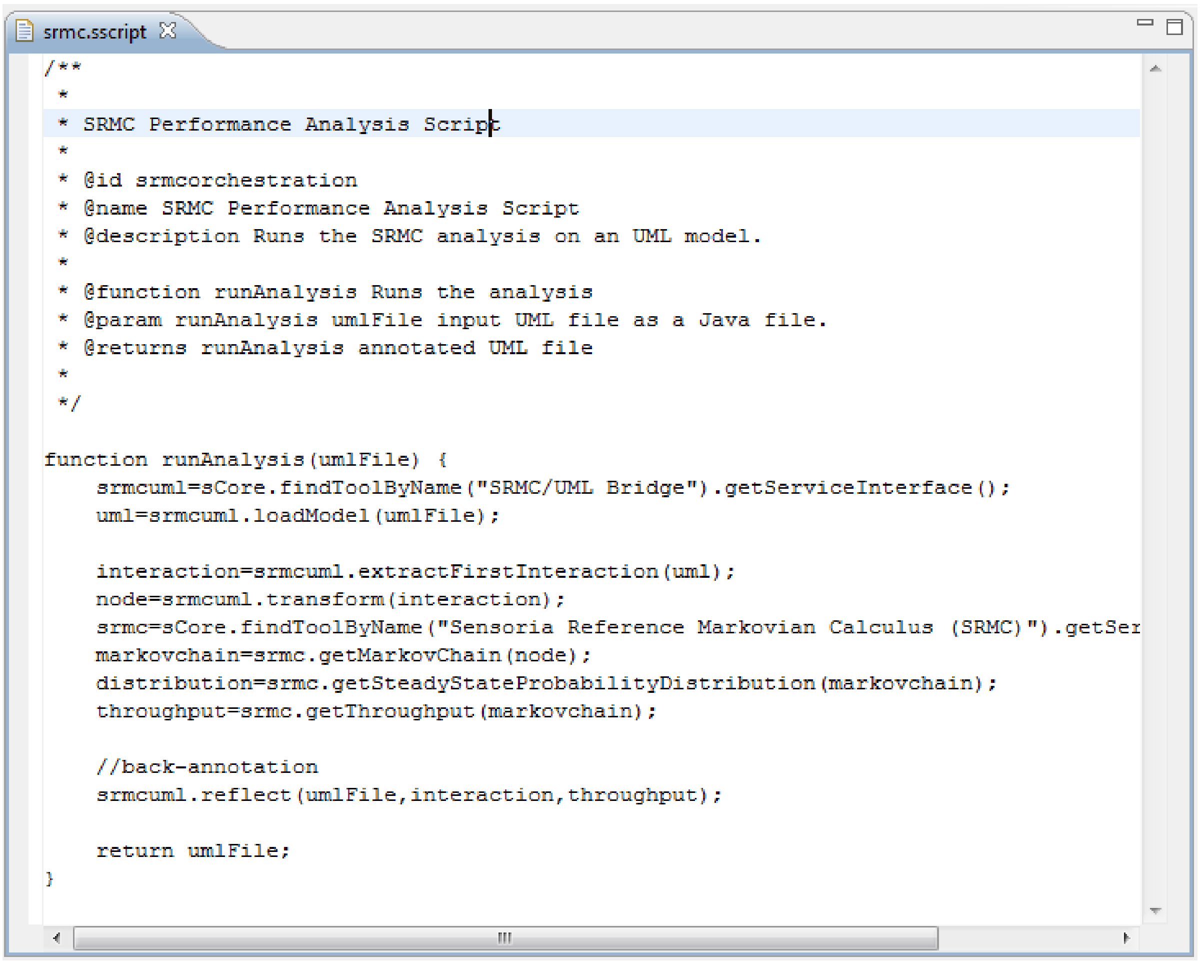


Figure 16: Running an SDE Shell script

The most powerful way of creating orchestrations within the SDE Development is writing **Tool Scripts**. These scripts are again based on JavaScript, but employ functions and certain comments for providing functionality, which allows the SDE to convert them to standard tools.



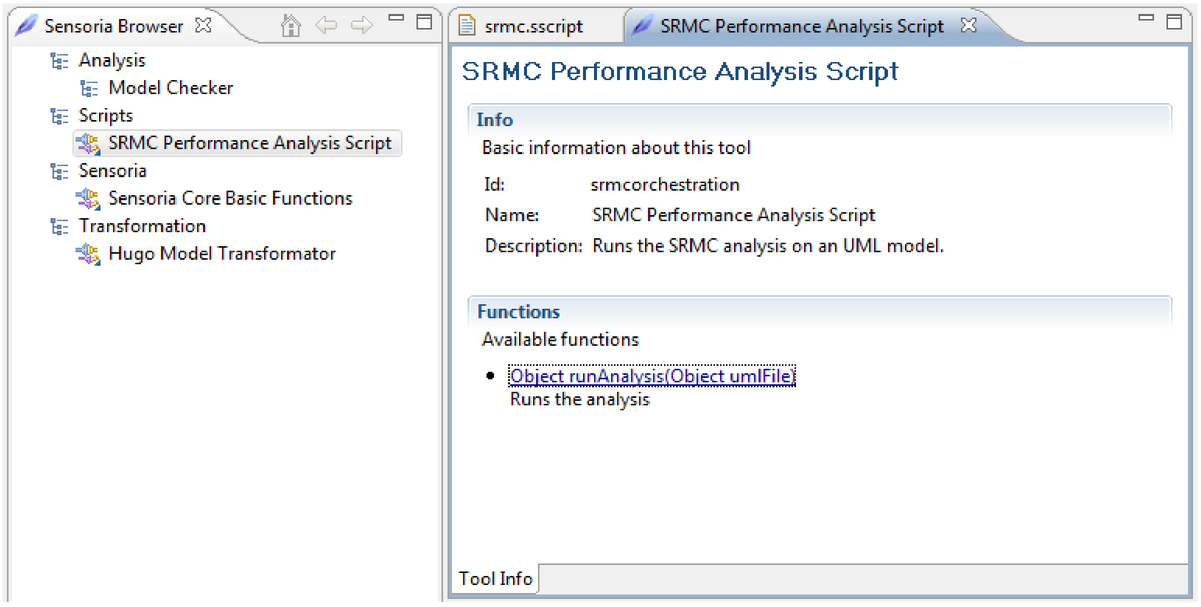
```
/**
 *
 * SRMC Performance Analysis Script
 *
 * @id srmcorchestration
 * @name SRMC Performance Analysis Script
 * @description Runs the SRMC analysis on an UML model.
 *
 * @function runAnalysis Runs the analysis
 * @param runAnalysis umlFile input UML file as a Java file.
 * @returns runAnalysis annotated UML file
 *
 */

function runAnalysis(umlFile) {
    srmcuml=sCore.findToolByName("SRMC/UML Bridge").getServiceInterface();
    uml=srmcuml.loadModel(umlFile);

    interaction=srmcuml.extractFirstInteraction(uml);
    node=srmcuml.transform(interaction);
    srmc=sCore.findToolByName("Sensoria Reference Markovian Calculus (SRMC)").getSer
    markovchain=srmc.getMarkovChain(node);
    distribution=srmc.getSteadyStateProbabilityDistribution(markovchain);
    throughput=srmc.getThroughput(markovchain);

    //back-annotation
    srmcuml.reflect(umlFile,interaction,throughput);

    return umlFile;
}
```

Figure 17: An SDE Tool Script

Figure 17 shows an example of a SDE Tool Script containing one function called `runAnalysis`, which takes one parameter named `umlFile`. The function orchestrates two tools – the `SRMC/UML bridge` and the `SRMC` tool itself – to provide some functionality. In the comment at the top of the function, various tags are used to provide descriptions to the content of the file. Only the id and name of the tool are required, however it is recommended to provide all of these tags to make the tool more usable.

- The `@id`, `@name`, and `@description` tags are used to describe the tool created from the script.
- Per function, use
    - `@function [functionName] [description]` to add a description to a function
    - `@param [functionName] [paramName] [description]` to add a description to a parameter of the function. Note that you must add these `@param` links in the same order as the parameters in the function.
    - `@returns [functionName] [description]` to add a description of the return value of the function.

After a Tool Script has been written, right-click on the file in the Navigator or Package Explorer and select `SDE: Add Script as Tool...`. The script will be converted to Java and shown as a new tool in the browser, as can be seen from Figure 18.



SRMC Performance Analysis Script

Info

Basic information about this tool

Id: srmcorchestration

Name: SRMC Performance Analysis Script

Description: Runs the SRMC analysis on an UML model.

Functions

Available functions

- Object runAnalysis(Object umlFile)
  Runs the analysis

Figure 18: A tool converted from JavaScript

Note that all return types and parameters are assumed to be Objects.

A link to the original script source will be kept in the Service Development Environment and the tool will be re-created upon start-up. If the original file is missing or the workspace has changed, the script will be silently ignored.

3.4.3 Graphical Orchestration

The other alternative to writing orchestrations is to employ the graphical activity diagram editor included in the SDE. To create a graphical orchestration, select **New** > **Other** > **SDE SDE Graphical Orchestration**.

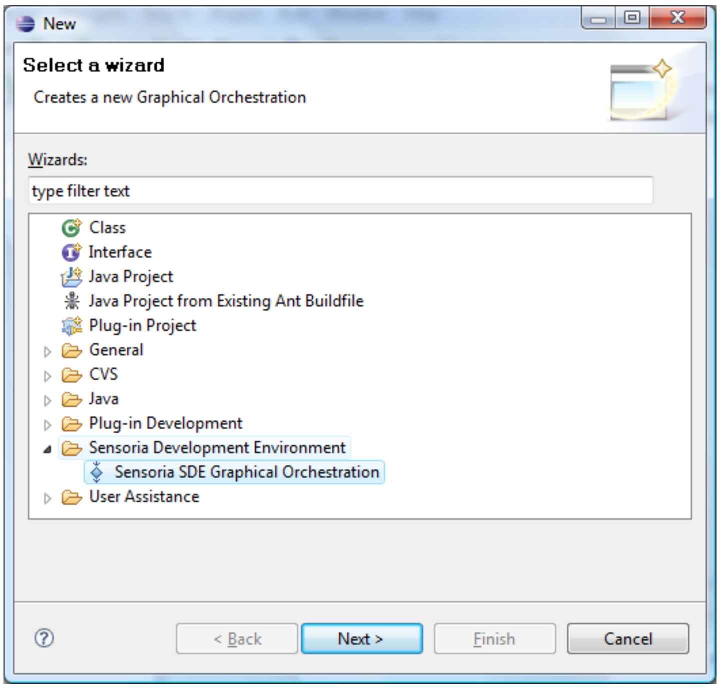


Figure 19: Creating a graphical orchestration

In the following dialogs, select a name for the two files which are created:

- The first file contains the diagram, and has the extension .god (for graphical orchestration diagram)
- The second file contains the orchestration itself, and has the extension .go (for graphical orchestration)

It is recommended to pick the same name for both files (excluding extension). Once the files have been created, the editor in Figure 20 is shown.

The canvas of the editor corresponds to a new tool to be created in the SDE. As each tool can contain multiple functions, a function needs to be added first. To do so, click on **Function** in the palette on the right hand side, and click onto the canvas to create a new function. Name it appropriately, for example **checkWithWSEngineer** (Figure 21).

Now, tool functions can be dragged into the new function as appropriate from the palette on the right-hand side, which contains all invokeable function of all installed tools. Additionally, the following meta tools may be used:

- Use **Link** to model data flow from an output of a function to the input of another
- Use **Input Pin** to add an input parameter to a function
- Use **Output Pin** to add an output parameter to a function

An example of a complete script is shown in Figure 22.

Before you can execute the function, you also need to name the tool. For this, right-click on the canvas and select properties. The properties view shows up, you need to fill in both the name and id (Figure 23).

Finally, to execute a function, simply click on the **green play button** in the upper right corner of a function. You may also use the SDE menu to convert the whole tool to a SDE tool.

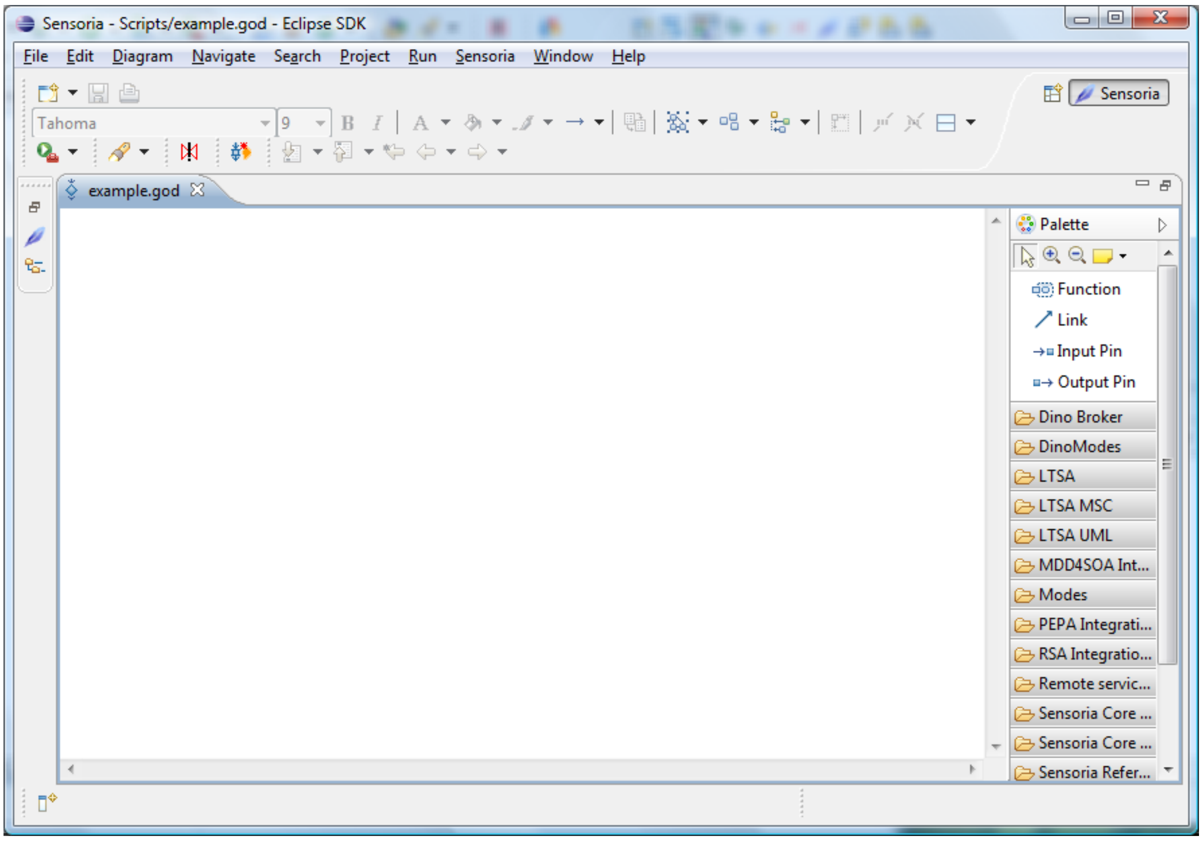


Figure 20: Graphical Orchestration editor

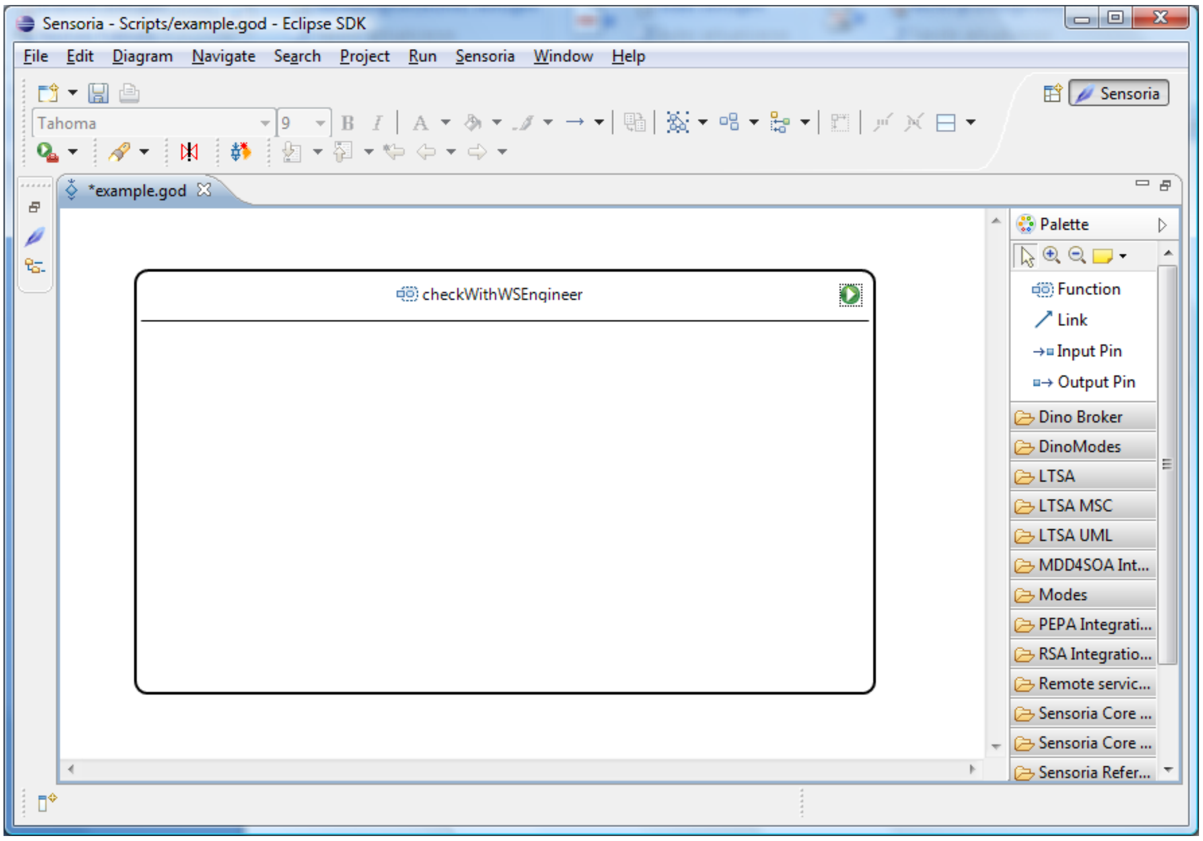


Figure 21: Creating a function

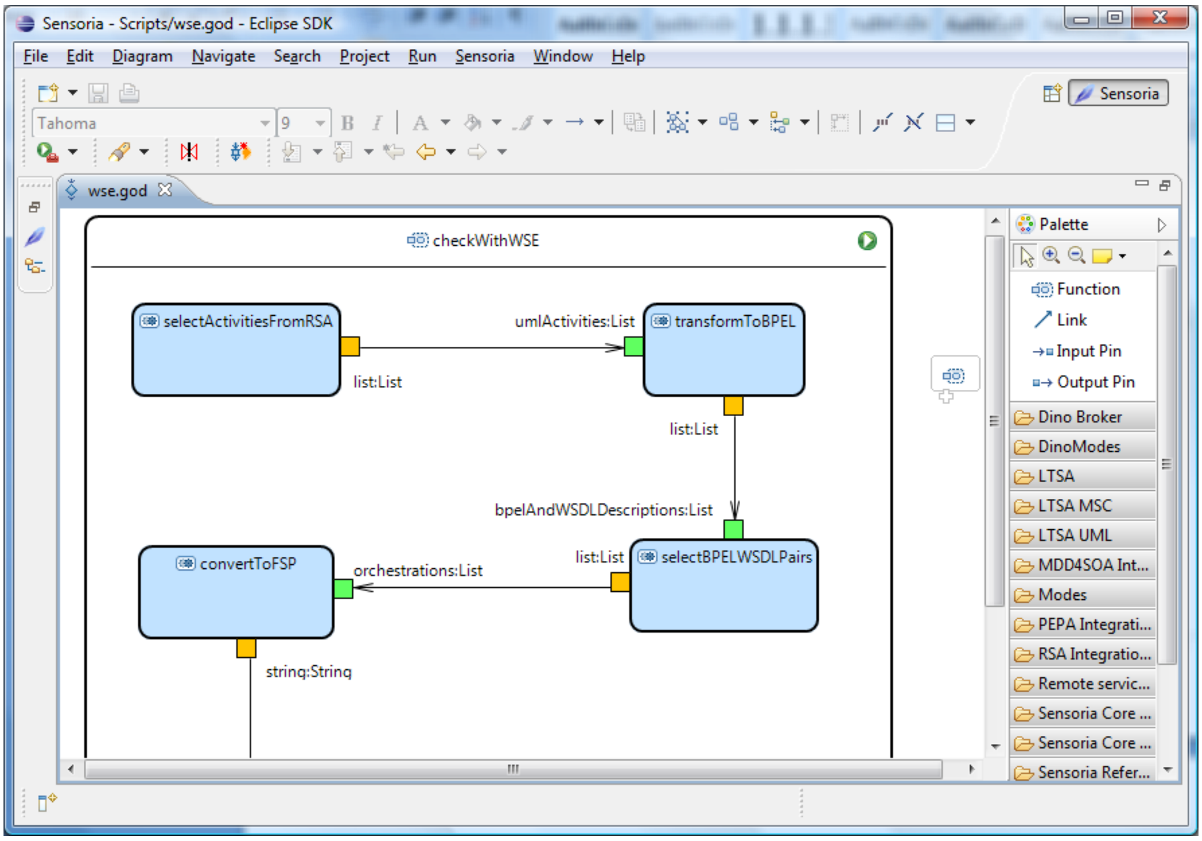
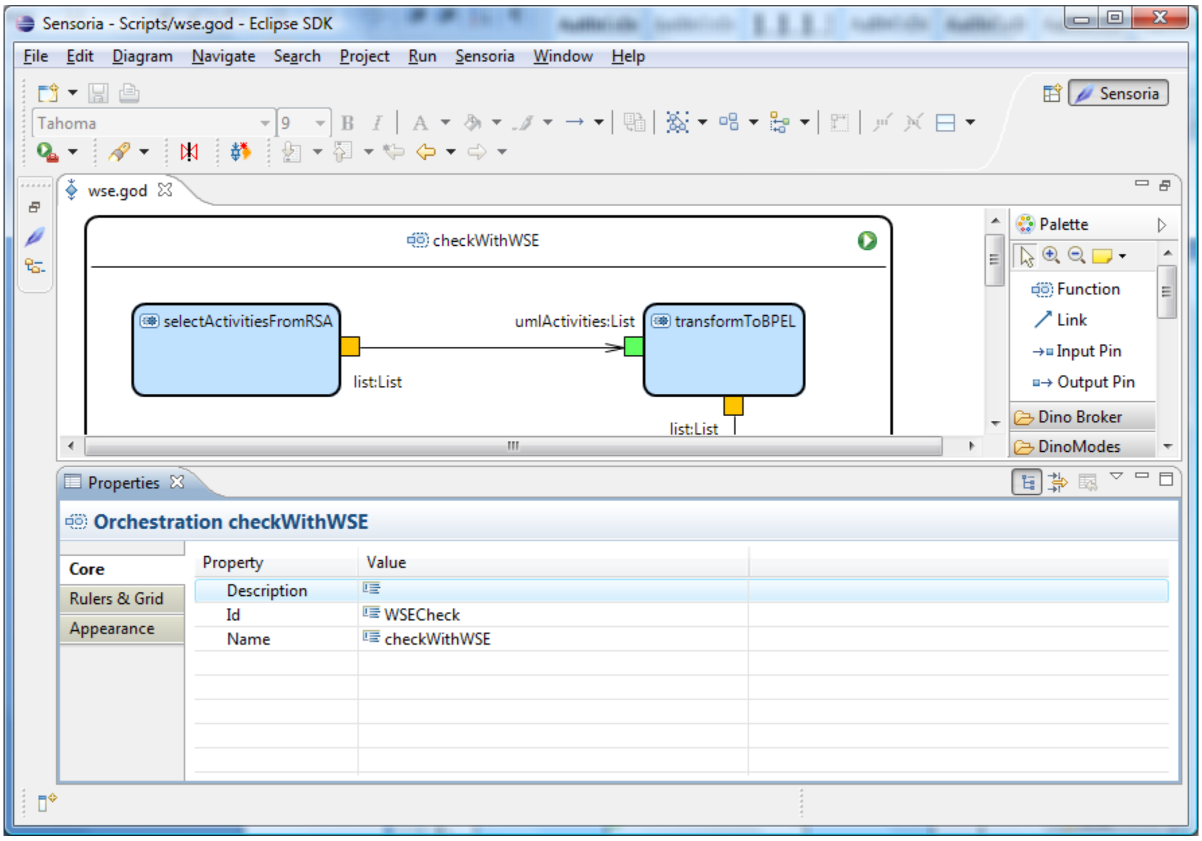


Figure 22: A complete function



Orchestration checkWithWSE

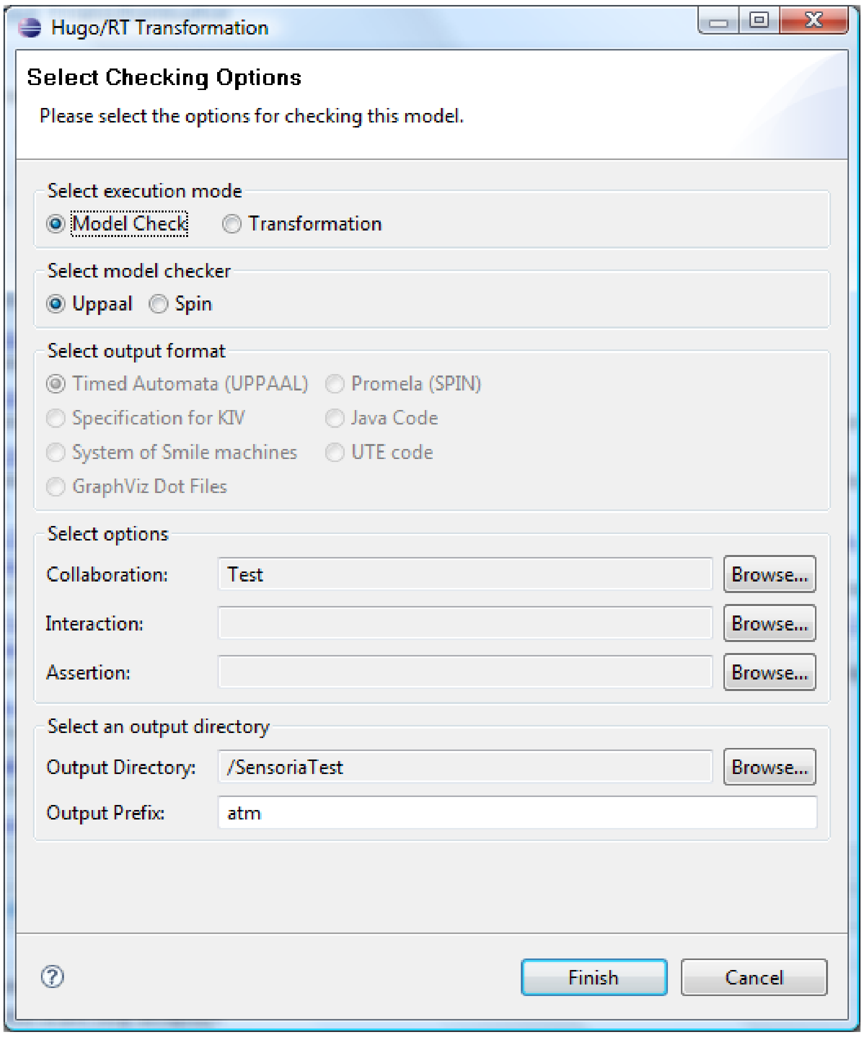
Property	Value
Description	
Id	WSECheck
Name	checkWithWSE

Figure 23: Tool Name and ID

3.5 Custom Tool UIs

As mentioned before, each SDE tool can also provide its own custom UI to the Eclipse platform. From within this UI, the SDE tool may be used as such, but this is not necessary.

As an example for such an UI, the Hugo/RT tool has been equipped with a wizard for model checking or transforming UTE- or XMI files. The wizard can be invoked by right-clicking `.ute` or `.xml` files in the Navigator or Package Explorer views. A wizard opens, which looks as follows.



Select Checking Options
Please select the options for checking this model.

Select execution mode
[✓ Model Check] Transformation

Select model checker
[✓ Uppaal] Spin

Select output format
[✓ Timed Automata (UPPAAL)] Promela (SPIN)
Specification for KIV Java Code
System of Smile machines UTE code
GraphViz Dot Files

Select options
Collaboration: Test
Interaction:
Assertion:

Select an output directory
Output Directory: /SensoriaTest
Output Prefix: atm

Figure 24: The Hugo/RT wizard

Tools are not limited in their usage of Eclipse UI elements and may provide whatever is needed in addition to allowing access to their core functionality via the Service Development Environment.

4 Developers Guide

The Development Environment contains all the relevant functionality to enable the vision described in the introduction. The architecture has been implemented as laid out in the first chapter. This chapter will provide some more technical insights into this implementation and offer a guide for developing new tools for this platform.

4.1 Development Environment Architecture

The Service Development Environment – as provided by the core feature from the update site – has been implemented as two OSGi bundles – the core itself, which is only based on Equinox, and the UI, which is based on Eclipse.

The following figure shows how the various components of the Service Development Environment – as well as some examples – fit together.

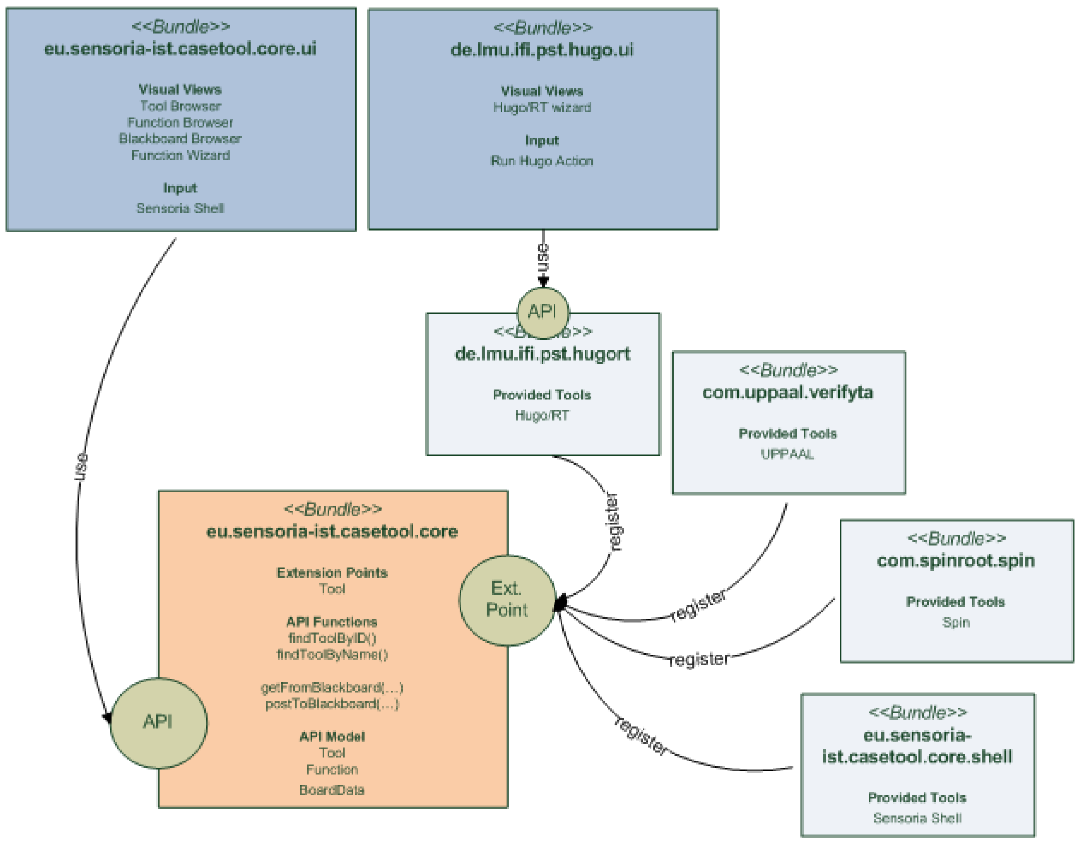


Figure 25: Development Environment Technical Architecture

In the lower left corner, the SDE core bundle is displayed in orange. It provides two externally accessible interfaces:

- An **Equinox extension point** (on the right-hand side), which can be used for tool registration.
- A **Java API** (on the left hand side), which can be used by orchestrators, discoverers, and UI to access the registered tools.

In the upper left corner, the SDE UI bundle is displayed in blue. It provides several visual views and uses the API of the core to retrieve tools and their functionality as well as blackboard resources.

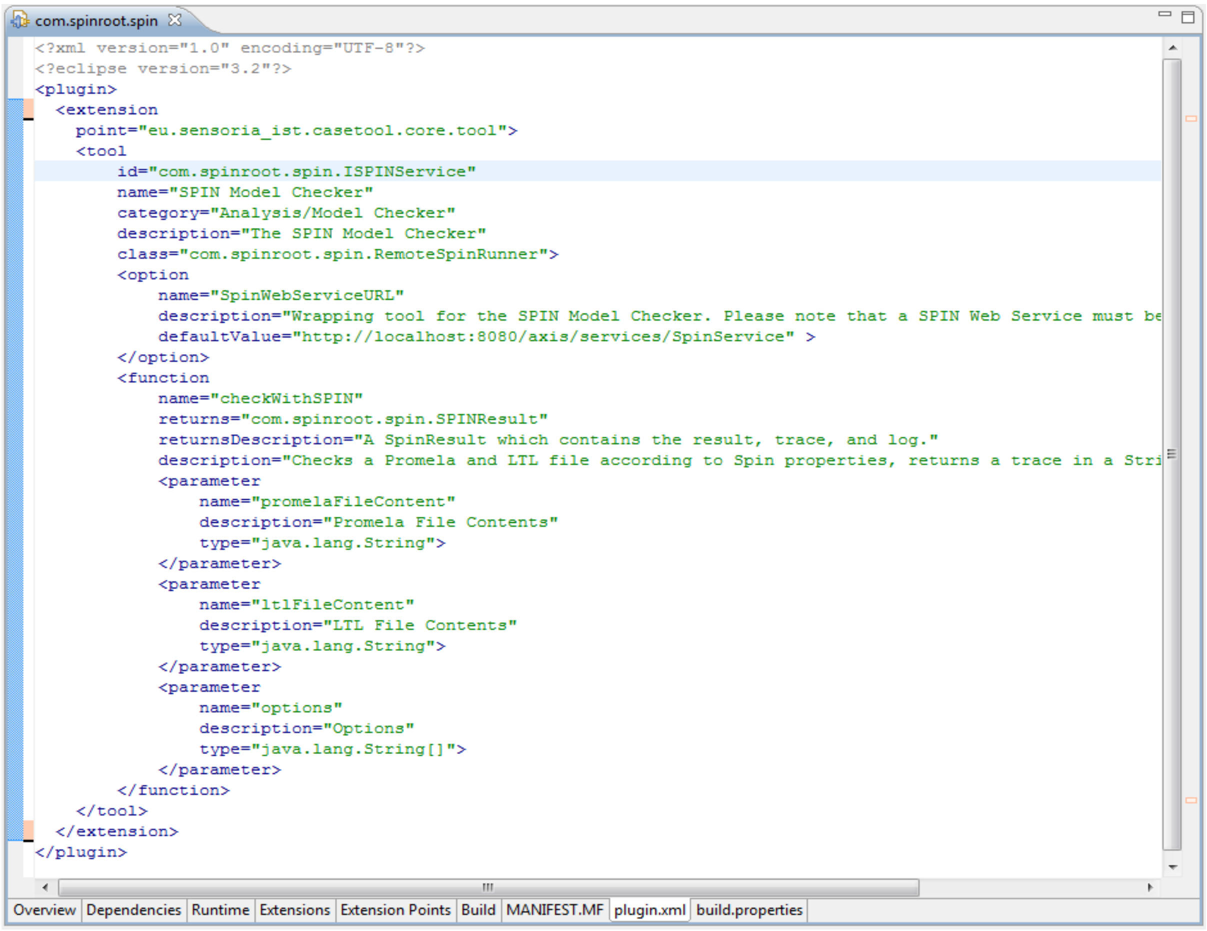
On the right-hand side, four tools are displayed (Hugo/RT, UPPAAL, SPIN, and the Shell); one of which has an additional UI bundle available (Hugo).

The complete source for all these bundles and plug-ins is available via Subversion. The Subversion URL is

http://svn.pst.ifi.lmu.de/svn/sde/

4.1.1 Core Extension Point

The next figure shows an example of a tool registration by using the Equinox extension point of the core. In the example, the SPIN bundle is registered with the SDE core. This will be explained in more detail in the following sections.



```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
  <extension
    point="eu.sensoria_ist.casetool.core.tool">
    <tool
        id="com.spinroot.spin.ISPINService"
        name="SPIN Model Checker"
        category="Analysis/Model Checker"
        description="The SPIN Model Checker"
        class="com.spinroot.spin.RemoteSpinRunner">
        <option
            name="SpinWebServiceURL"
            description="Wrapping tool for the SPIN Model Checker. Please note that a SPIN Web Service must be
            defaultValue="http://localhost:8080/axis/services/SpinService" >
        </option>
        <function
            name="checkWithSPIN"
            returns="com.spinroot.spin.SPINResult"
            returnsDescription="A SpinResult which contains the result, trace, and log."
            description="Checks a Promela and LTL file according to Spin properties, returns a trace in a Stri
            <parameter
                name="promelaFileContent"
                description="Promela File Contents"
                type="java.lang.String">
            </parameter>
            <parameter
                name="ltlFileContent"
                description="LTL File Contents"
                type="java.lang.String">
            </parameter>
            <parameter
                name="options"
                description="Options"
                type="java.lang.String[]">
            </parameter>
        </function>
    </tool>
  </extension>
</plugin>
```

Figure 26: Registering a tool

4.1.2 Core API

Besides the extension point for registering tools, the core also provides API to orchestrators, discoverers, and its own UI. The most important API functions are listed in the next figure.

```
public interface ISensoriaCore {

       public IToolStore getTools();

       public IBlackboard getBlackboard();

}

public interface IToolStore {

       public Set<Tool> getTools();

       public ToolElement getRoot();

       public Tool findToolById(String toolID);

       public Tool findToolByName(String toolName);

}

public interface ITool {

       public String getId();

       public String getDescription();

       public String[] getCategory();

       public Object getServiceInterface();

}
```

The functions displayed allow for retrieving tools, which can then be used to get the service interface for invoking functions.

4.2 Creating Tools

Creating new tools or adapting existing tools for use in the Service Development Environment requires the following three steps:

- Creating or reusing an OSGi bundle.
- Creating a tool class/interface and functions within this bundle, which implement the tool functionality.
- Registering the tool with the Service Development Environment core.

Depending on the type of tool to be implemented, an OSGi bundle may already be available or needs to be created from scratch:

- If the tool to be integrated is an Eclipse plug-in (starting from version 3.0), it is already implemented as an OSGi bundle.
- If the tool is written in Java, the code can probably be used from within an OSGi bundle or Eclipse plug-in as well, either in source or as a .jar library.
- If the tool is written in a native language or provided as a Web service, the recommended way is to create a wrapping OSGi bundle which forwards all functionality.

Publishing a tool to the Service Development Environment requires three steps:

- adding the SDE core as a dependency to the tool bundle,
- writing some XML to register the tool, and the provided functions of the tool, with the Service Development Environment core (the XML can be generated from Java 5 annotations, see below),

- exporting the necessary packages and libraries so the core is able to find the implementation code.

Additionally, it is of course possible to contribute to the Eclipse UI as appropriate.

4.2.1 Walkthrough for an Example Tool

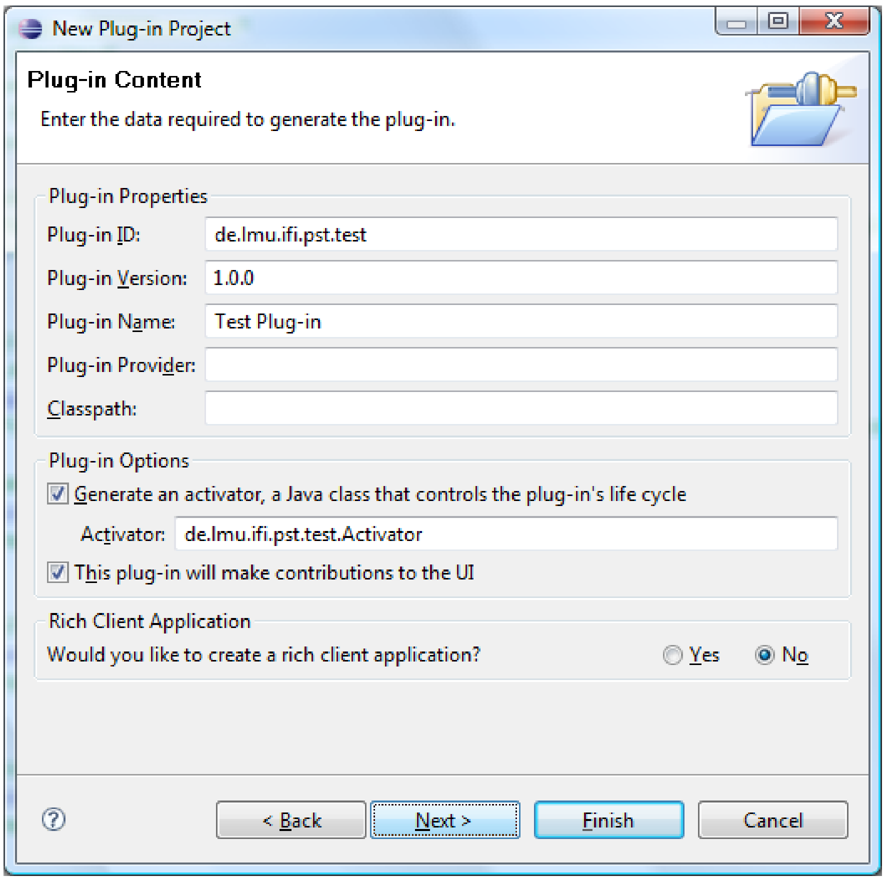
Before writing a new tool for the Service Development Environment, you need to have the Development Environment installed, which includes the **development package**. One installed, start the Eclipse workbench.

Normally, a tool should be split into at least two Eclipse projects – one OSGi bundle for the core functionality which exposes the functions on an OSGi level, and one for the Eclipse UI (if necessary). However, some existing tools may already be provided as bundles, so the Service Development Environment parts can either be added to those bundles or extracted into separate bundles to allow the tools to work without the Development Environment present. Also, there might be tools which are so dependent on Eclipse that an OSGi component is not possible. In this simple example, we will also create a bundle as an Eclipse plug-in project for simplicity.

4.2.1.1 First Step: Creating the project

To create a new plug-in project, select "**File > New > Project…**" in the Eclipse main menu. In the wizard which follows, select **Plug-In project**.

Choose a name for the project (it is recommended to use a fully qualified domain name in reverse order, i.e. `de.lmu.ifi.pst.test`); click **Next**, then **Finish**.



Plug-in ID:	de.lmu.ifi.pst.test
Plug-in Version:	1.0.0
Plug-in Name:	Test Plug-in
Plug-in Provider:	
Classpath:	

[✓] Generate an activator, a Java class that controls the plug-in's life cycle

Activator: de.lmu.ifi.pst.test.Activator

[✓] This plug-in will make contributions to the UI

Would you like to create a rich client application? Yes [✓ No]

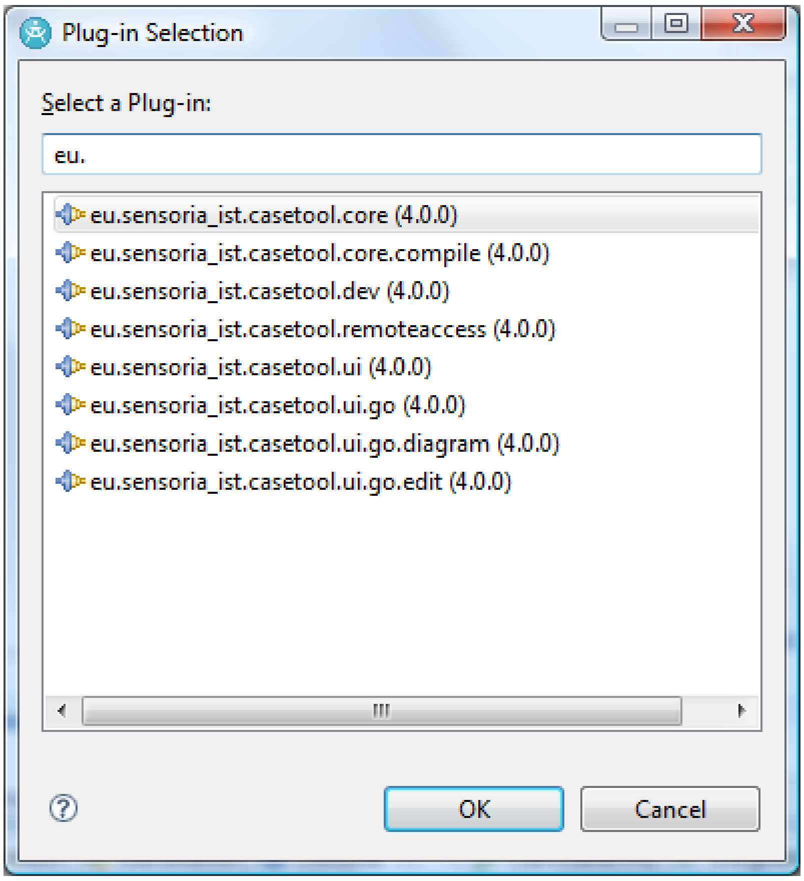
Figure 27: Creating a new plug-in project

Once created, the project will show up in the package explorer.

4.2.1.2 Second Step: Adding the SDE core as a Dependency

The plug-in contains a `MANIFEST.MF` file in the `META-INF` folder which contains the OSGi settings for this plug-in. Double-clicking on the `MANIFEST.MF` file opens the **PDE** (Plugin Development Environment) **editor** for this bundle.

Within the PDE editor, there are several pages for the various options. Right now, we are interested in the **Dependency** page. On this page, select **Add**… to add a new dependency; from the list presented, select the Service Development Environment core `eu.sensoria-ist.casetool.core`.



eu.sensoria_ist.casetool.core (4.0.0)
eu.sensoria_ist.casetool.core.compile (4.0.0)
eu.sensoria_ist.casetool.dev (4.0.0)
eu.sensoria_ist.casetool.remoteaccess (4.0.0)
eu.sensoria_ist.casetool.ui (4.0.0)
eu.sensoria_ist.casetool.ui.go (4.0.0)
eu.sensoria_ist.casetool.ui.go.diagram (4.0.0)
eu.sensoria_ist.casetool.ui.go.edit (4.0.0)

Figure 28: Adding the dependency to the core

Once the dependency has been added, save the `MANIFEST.MF` file.

4.2.1.3 Third Step: Creating a Tool Class and/or Interface

Each **tool** corresponds to exactly **one** implementing class with an **arbitrary number of functions** (of course, there may be more than one tool per plug-in, if necessary). Of the methods of the class, all or only a part can be published for the Development Environment. It is recommended to create an interface for these methods to ensure that:

1. All required methods are indeed published.
2. All methods are public.

The use of an interface is also practical when using Java annotations for describing the tool interface (see next step). The class implementing the methods or functions of the tool must fulfil the following requirements:

a) it must have a no-argument default constructor for instantiation.

b) it must implement one of the two SDE core base interfaces:

   a. `eu.sensoria-ist.casetool.ISensoriaTool` (for normal tools – the interface is empty)

b. `eu.sensoria-ist.casetool.ISensoriaConfigurableTool` (for tools with options, see advanced topics)

c) (Optional) The interface `eu.sensoria-ist.casetool.ISensoriaModelHandler` may be implemented additionally. This interface allows tools to add functionality for handling their model objects in the UI. This will be detailed later.

Each tool class will be instantiated only once by the Development Environment and then used for all consequent function calls.

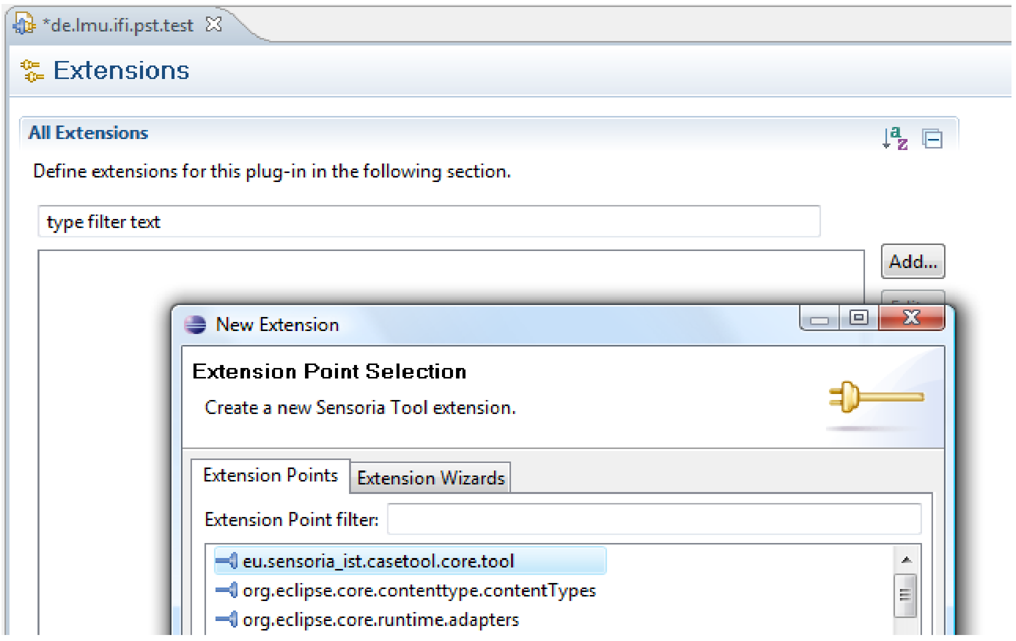
Once a tool class and/or interface have been created, the tool can be published.

4.2.1.4 Fourth Step: Publishing the Tool

The tool bundle and the implementing class must be registered with the Service Development Environment in order for the tool to be listed in the core API and displayed in the core UI. Publishing a tool is done by creating an Equinox extension similar to registering other extensions in Eclipse, i.e. by registering functionality at an extension point.

Such an extension is either written by hand in XML inside the `plugin.xml` file, or added via the PDE wizard, which in turn writes the XML. However, with the Development Environment Development package installed, there is a third way (see below).

While writing the XML by hand is not recommended, the PDE wizard can be reached by opening the `MANIFEST.MF` file again, selecting the **Extension** tab, clicking **Add…**, and selecting the appropriate extension point to be extended.



Define extensions for this plug-in in the following section.

type filter text

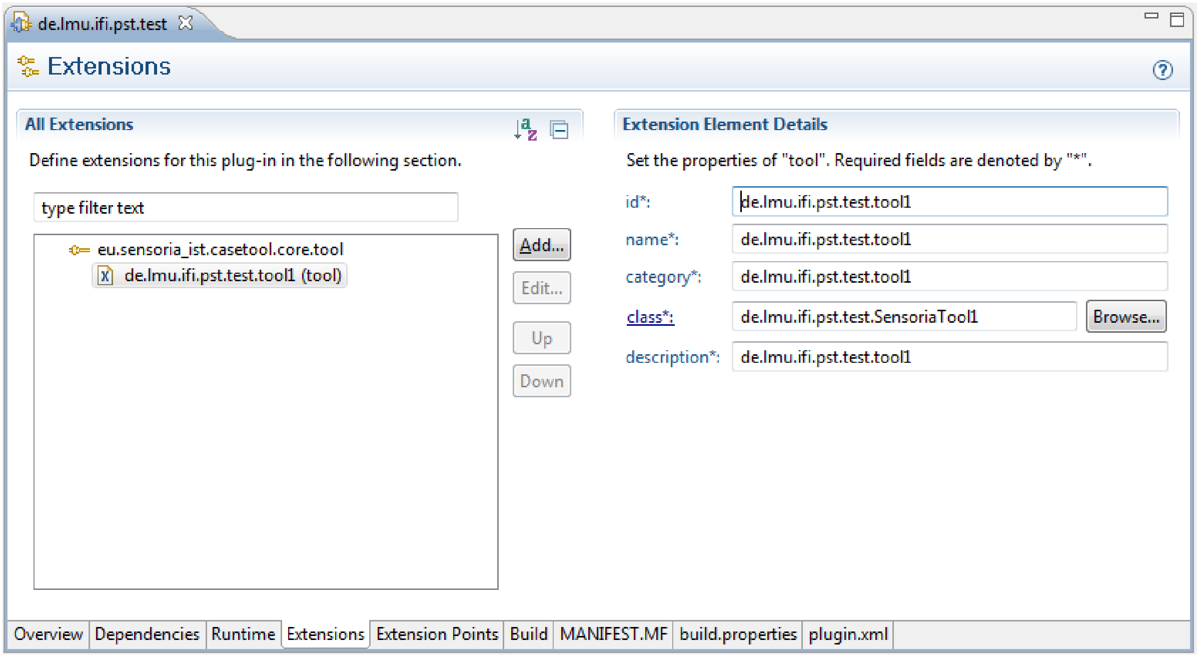
Create a new Sensoria Tool extension.

Extension Point filter:

eu.sensoria_ist.casetool.core.tool
org.eclipse.core.contenttype.contentTypes
org.eclipse.core.runtime.adapters

Figure 29: Adding a new extension, 1

The extension point for adding new Service Development Environment tools has the ID `eu.sensora_ist.casetool.core.tool`. Once added, an extension is displayed as follows:



Define extensions for this plug-in in the following section.

type filter text

eu.sensoria_ist.casetool.core.tool
de.lmu.ifi.pst.test.tool1 (tool)

Extension Element Details

Set the properties of "tool". Required fields are denoted by "*".

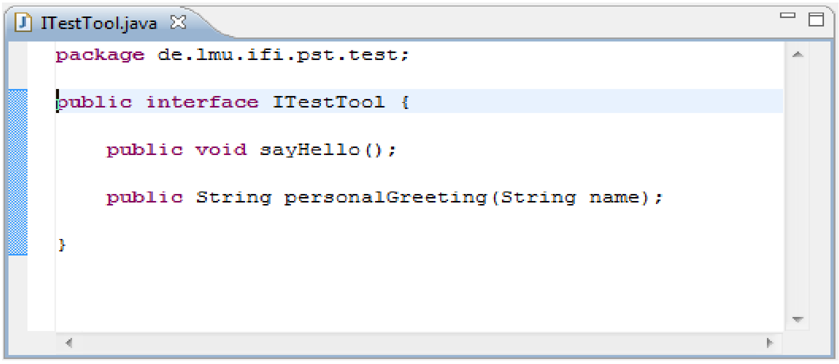
id*:	de.lmu.ifi.pst.test.tool1
name*:	de.lmu.ifi.pst.test.tool1
category*:	de.lmu.ifi.pst.test.tool1
class*:	de.lmu.ifi.pst.test.SensoriaTool1
description*:	de.lmu.ifi.pst.test.tool1

Figure 30: Adding a new extension, 2

A new tool with the ID `de.lmu.ifi.pst.test.tool1` has been added automatically; its properties are displayed on the right. The one function beneath it has been added by right-clicking the tool and selecting the appropriate menu item to create new function.

All the properties of the tool and all of the functions must be provided in order to publish the tool and the selected functions. Rather than doing this by hand, the development package of the Service Development Environment can automatically create this information from Java source code and annotations. This is the recommended way and explained in the following.

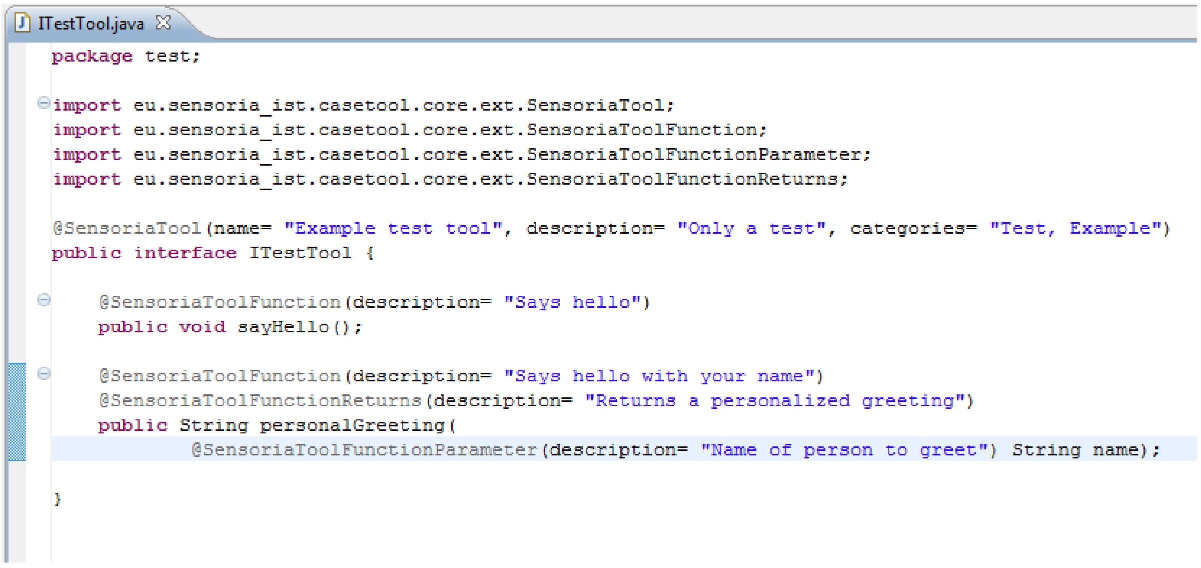
Open the Development Environment implementing class or interface created in the previous step in the Java editor. For example, the interface might look like this:



```
package de.lmu.ifi.pst.test;

public interface ITestTool {

    public void sayHello();

    public String personalGreeting(String name);

}
```

Figure 31: An interface for a tool

The interface may now be annotated as follows:



```
package test;

import eu.sensoria_ist.casetool.core.ext.SensoriaTool;
import eu.sensoria_ist.casetool.core.ext.SensoriaToolFunction;
import eu.sensoria_ist.casetool.core.ext.SensoriaToolFunctionParameter;
import eu.sensoria_ist.casetool.core.ext.SensoriaToolFunctionReturns;

@SensoriaTool(name= "Example test tool", description= "Only a test", categories= "Test, Example")
public interface ITestTool {

    @SensoriaToolFunction(description= "Says hello")
    public void sayHello();

    @SensoriaToolFunction(description= "Says hello with your name")
    @SensoriaToolFunctionReturns(description= "Returns a personalized greeting")
    public String personalGreeting(
            @SensoriaToolFunctionParameter(description= "Name of person to greet") String name);

}
```

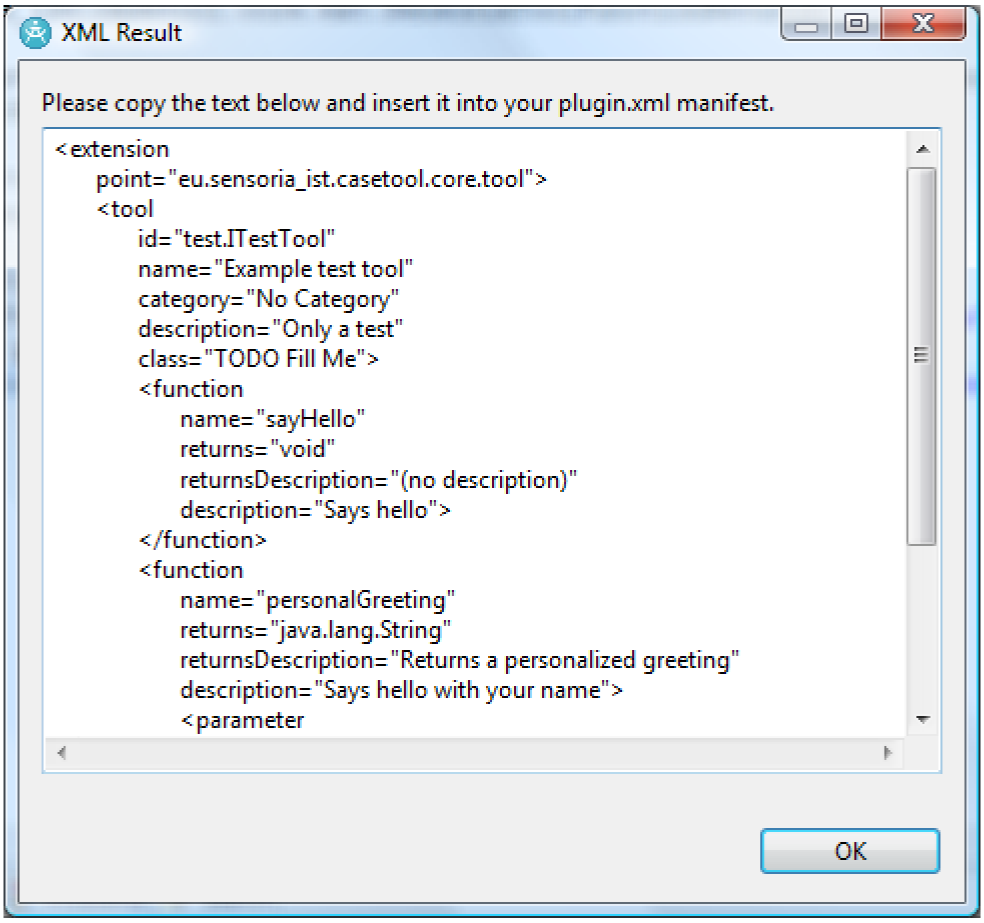
Figure 32: Java annotations for an SDE tool

As can be seen, the annotations are added both to the interface itself and to every function. The tool interface is annotated with three options:

- A **name** for the tool. This should be human readable.
- A **description** for the tool. This should be one sentence, at most two, which describe what this tool does.
- A **category**. Tools are categorized into a categorization tree which is built on-the-fly by the Development Environment. A path in this tree is given here, separated by forward slashes; for example **`Analysis/Model Checking`**. The SDE Tools Wiki features a list of predefined categories.

Every function is annotated with a **description** which should describe, in one or two sentences, what the function does. Additionally, a **description of the return parameter**, and **descriptions of each of the parameters** should be specified.

Once the annotation is complete, select the interface or class in the package explorer and right-click. A new menu item named **Convert to SDE tool** shows up. Selecting this menu item yields the following dialog:



Please copy the text below and insert it into your plugin.xml manifest.

```
<extension
    point="eu.sensoria_ist.casetool.core.tool">
    <tool
        id="test.ITestTool"
        name="Example test tool"
        category="No Category"
        description="Only a test"
        class="TODO Fill Me">
        <function
            name="sayHello"
            returns="void"
            returnsDescription="(no description)"
            description="Says hello">
        </function>
        <function
            name="personalGreeting"
            returns="java.lang.String"
            returnsDescription="Returns a personalized greeting"
            description="Says hello with your name">
            <parameter
```

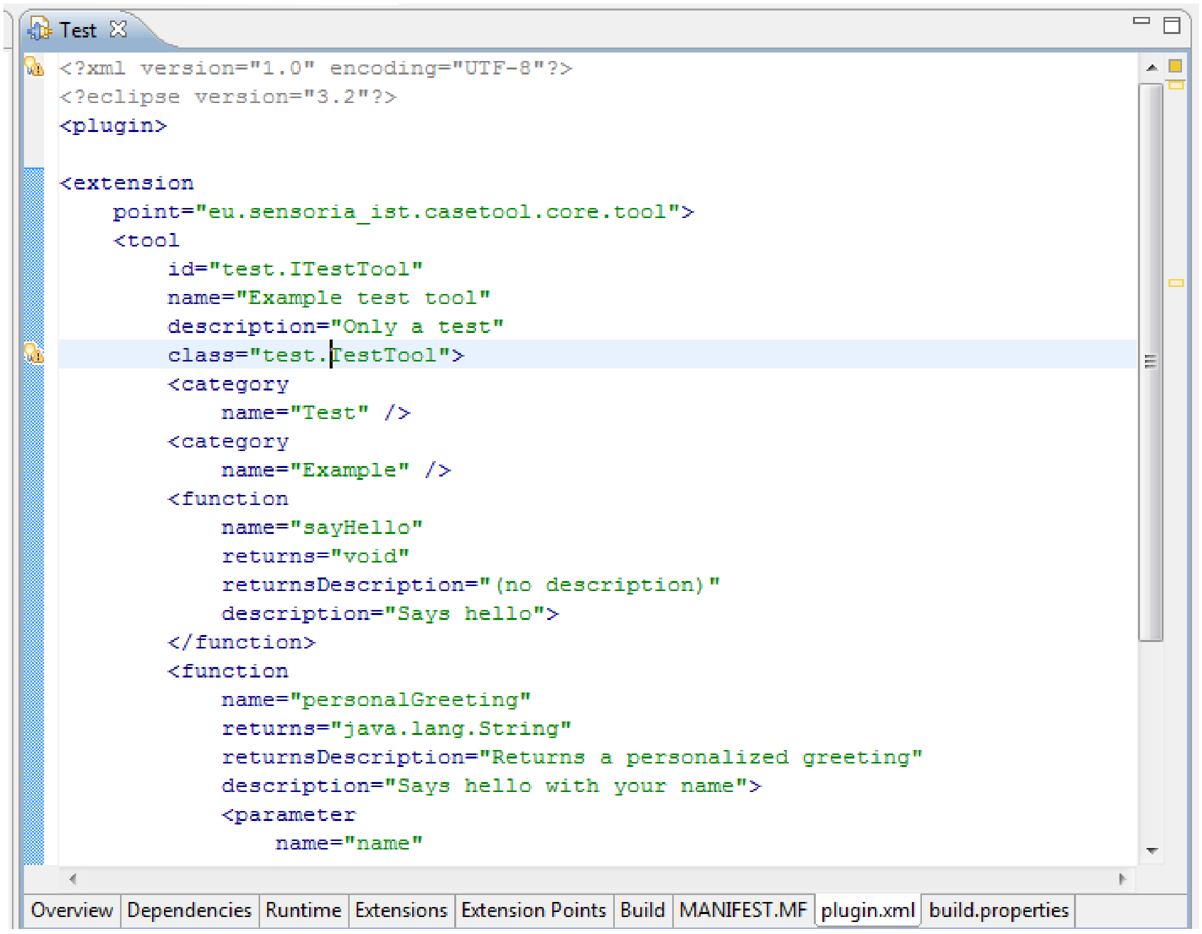
Figure 33: XML for the plugin.xml

The generated XML code is displayed in the dialog and can now be copied and pasted directly into the `plugin.xml`. Only one option still needs to be filled out: The actual implementation class is currently tagged with “**TODO Fill Me**” in the code, as the action cannot infer the actual implementing class.

To paste the code into the plugin.xml,

1. Open the **plugin.xml** file from the package explorer (if it has not yet been created, select the **MANIFEST.MF** file)
2. In the PDE editor which is opened, select the **`plugin.xml`** tab
3. Paste the code in-between the `<plug-in>` tags.

The following screen shot shows the result. Note that a class **`TestTool`** has been created in beforehand which implements **`ITestTool`**.



```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>

<extension
    point="eu.sensoria_ist.casetool.core.tool">
    <tool
        id="test.ITestTool"
        name="Example test tool"
        description="Only a test"
        class="test.TestTool">
        <category
            name="Test" />
        <category
            name="Example" />
        <function
            name="sayHello"
            returns="void"
            returnsDescription="(no description)"
            description="Says hello">
        </function>
        <function
            name="personalGreeting"
            returns="java.lang.String"
            returnsDescription="Returns a personalized greeting"
            description="Says hello with your name">
            <parameter
                name="name"
```

Figure 34: Finished tool definition

4.2.1.5 Fifth Step: Exporting Packages

The extension created in the last step points the Development Environment to the implementing class, which in turn probably uses more classes from the plug-in as well as libraries. These classes and libraries must be marked as available to other plug-ins if the Development Environment is to find them at runtime.

This can be done in the PDE editor as well. Selecting the **Runtime** tab yields the editor tab displayed in the next figure.

On this tab,

- add all relevant (which are normally all available) packages defined in the plug-in as exported packages in the list on the left.
- add all relevant (which are normally all available) jars used in the plug-in to be included on the runtime classpath in the list on the bottom right.

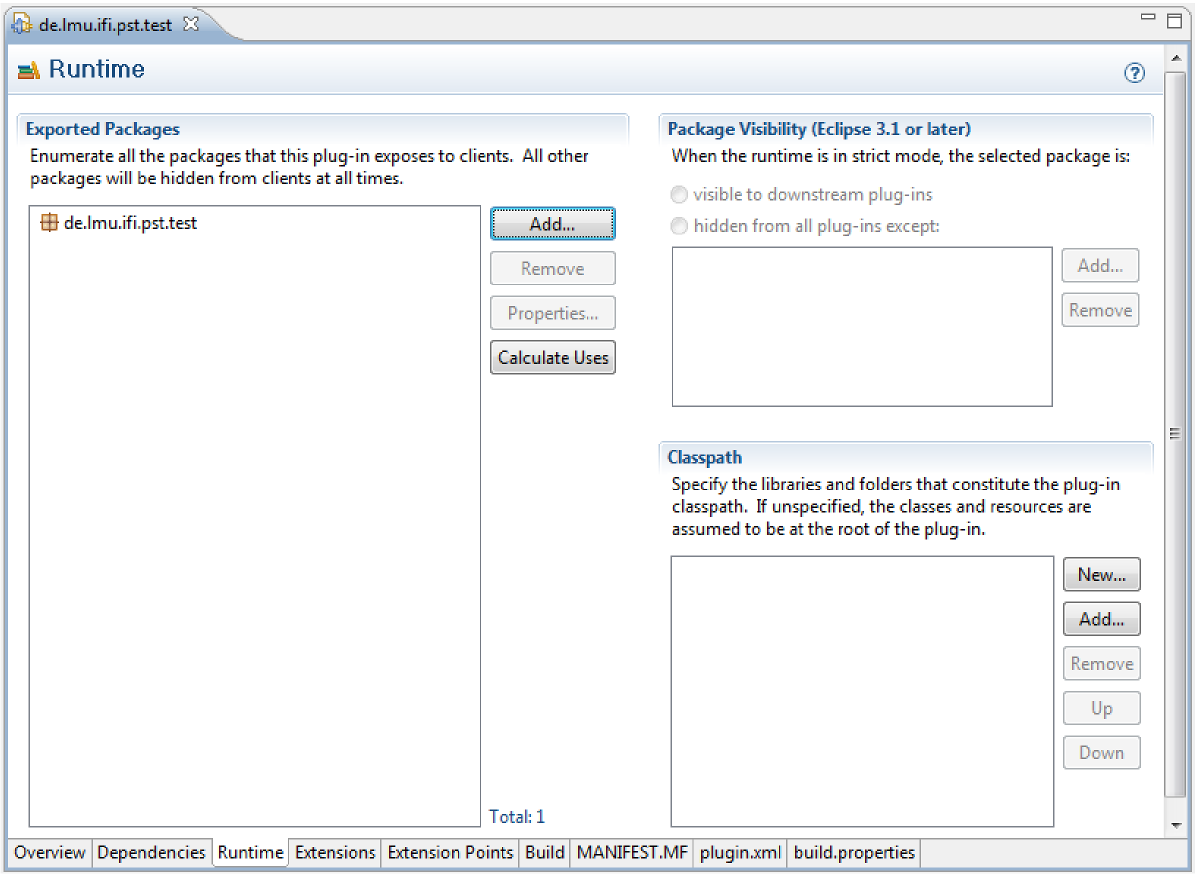
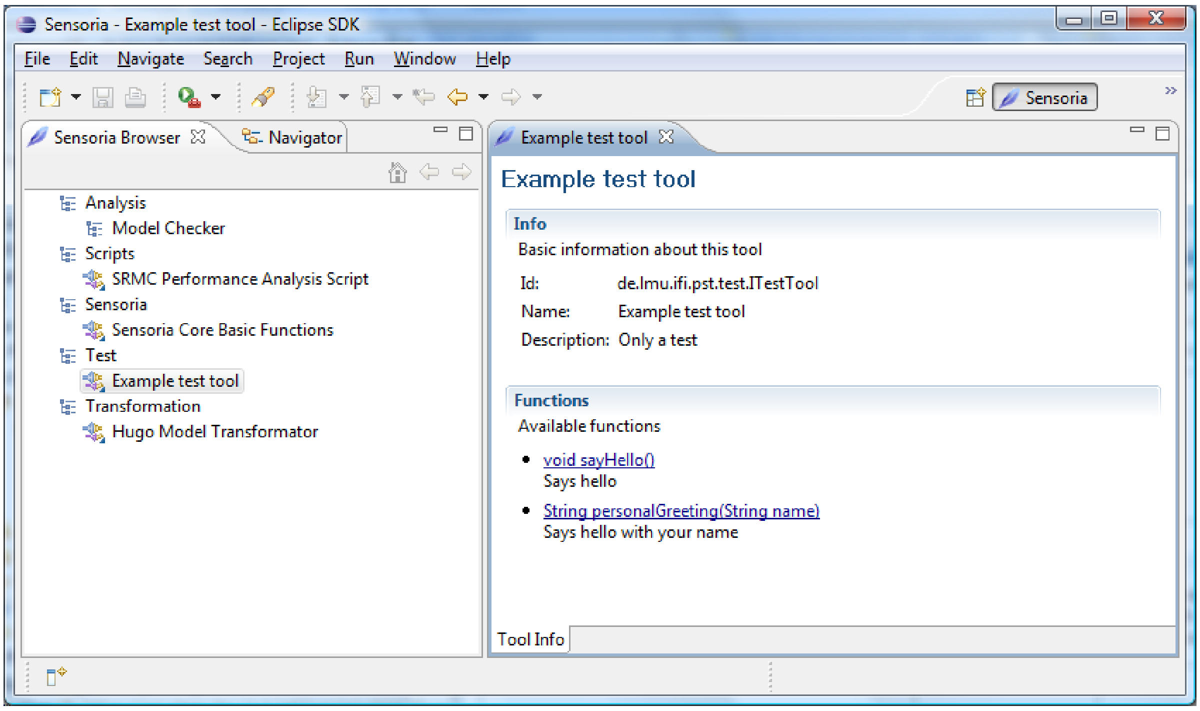

de.lmu.ifi.pst.test
Runtime
Exported Packages
Enumerate all the packages that this plug-in exposes to clients. All other packages will be hidden from clients at all times.
de.lmu.ifi.pst.test
Add...
Remove
Properties...
Calculate Uses
Total: 1
Package Visibility (Eclipse 3.1 or later)
When the runtime is in strict mode, the selected package is:
visible to downstream plug-ins
hidden from all plug-ins except:
Add...
Remove
Classpath
Specify the libraries and folders that constitute the plug-in classpath. If unspecified, the classes and resources are assumed to be at the root of the plug-in.
New...
Add...
Remove
Up
Down
Overview
Dependencies
Runtime
Extensions
Extension Points
Build
MANIFEST.MF
plugin.xml
build.properties

Figure 35: Runtime Tab

4.2.1.6 Final Step: Testing

Testing a plug-in or OSGi bundle requires starting a runtime workbench. To do this, open the **Overview** tab in the PDE editor and **select Launch an Eclipse application** (on the right-hand side).

A new Eclipse instance is started. Open the SDE perspective, which should allow access to the newly created tool:


Sensoria - Example test tool - Eclipse SDK
File Edit Navigate Search Project Run Window Help
Sensoria
Sensoria Browser
Navigator
Example test tool

- Analysis
  - Model Checker
- Scripts
  - SRMC Performance Analysis Script
- Sensoria
  - Sensoria Core Basic Functions
- Test
  - Example test tool
- Transformation
  - Hugo Model Transformator

Example test tool

Info

Basic information about this tool

Id: de.lmu.ifi.pst.test.ITestTool

Name: Example test tool

Description: Only a test

Functions

Available functions

- void sayHello()
  Says hello
- String personalGreeting(String name)
  Says hello with your name

Tool Info

Figure 36: New tool installed

4.2.2 Adding Model Handling

The Service Development Environment contains an additional interface, `ISensoriaModelHandler`, which can be implemented by tool providers to add advanced functionality for working with model objects used or returned by functions of this tool. In particular, implementing this interface will allow:

- Displaying a name and description for model objects in the blackboard and within the generic wizard UI.
- Automatically opening model objects in an editor or a custom tool UI when it is returned from a function, instead of simply posting it on the blackboard.

The interface, which must be implemented by the tool class defined in the extension definition, includes the methods displayed in **Fehler! Verweisquelle konnte nicht gefunden werden.**.

```
public interface ISensoriaModelHandler extends ISensoriaTool {

        public List<Class<?>> getHandledModelObjectClasses();

        public String getLabel(Object modelObject);

        public String getContent(Object modelObject);

        public boolean openInUI(Object modelObject);

}
```

Figure 37: ISensoriaModelHandler interface

The first method is used by the SDE core to determine which model objects will be handled by this tool. For example, the Hugo/RT tool defines a model object with class `uml.Model` and uses it extensively in the tool functions. Therefore, the tool returns a list which includes `uml.Model` here.

The second and third methods are called by the SDE core to retrieve a short label and a long complete String of the contents of an object.

Finally, the third method is called when the use requests opening of a model object. The tool may return false here if it cannot handle the given object.

Note that the classes String, File, and IFile are already handled by the SDE UI.

4.2.3 Advanced Topics

This section contains some more advanced topics related to tool creation. Additionally, a good starting point is the source code of other tools, like Hugo/RT, SPIN, and UPPAAL. The source code is available via SVN; see the SDE Tools Wiki for more information.

4.2.3.1 Options

As pointed out before, a tool may need options to be set by the user like the URL of a Web service or the location of a local script to be executed.

To enable options in your tool, you need to follow two steps:

1. Declare the options on the tool by using Java annotations.
2. Implement the `ISensoriaConfigurableTool` interface in the tool class.

The code written for the first step might look like this:

```
@SensoriaTool(
    name= "SPIN Model Checker",
    description= "The SPIN Model Checker",
    category= "Analysis/Model Checker")

@SensoriaToolOptions(
    @SensoriaToolOption(
        name= "SpinWebServiceURL",
        description= "Wrapping tool for the SPIN Model Checker",
        defaultValue= "http://localhost:8080/axis/services/SpinService")
)
public interface ISPINService { ... }
```

Figure 38: Defining options as Java annotations

For the second step, the interface `ISensoriaConfigurableTool` must be implemented. This is necessary for the Service Development Environment core to retrieve and set the options of this tool. A typical implementation of the two methods of this interface might look like this:

```
private Map<String, String> fOptionMap= new HashMap<String, String>();

public String getOption(String option) {
  return fOptionMap.get(option);
}

public void setOption(String option, String value) {
  fOptionMap.put(option, value);
}
```

Figure 39: Implementing ISensoriaConfigurableTool

4.2.3.2 Creating OSGi Bundles

Creating OSGi bundles is similar to creating normal Eclipse plug-ins. On the first page of the **New Plug-In Wizard**, select an OSGi framework as the target instead of an Eclipse platform.

Note that you might need to create a `plugin.xml` file by hand when using this option, as the **Extensions** tab is hidden by default for OSGi bundles.

4.2.3.3 Keeping Code Independent from the Development Environment

Normally, tools should be usable both within the context of the Development Environment and without. This can easily be achieved by creating separate plug-ins for the main code and the Development Environment integrator:

1. Create one or more bundles or plug-ins for the main code. These bundles or plug-ins contain the code to be executed both within the Development Environment and without.
2. Create an additional bundle which depends both on the SDE core and on the main bundles, and which contains the extension definition and service interface for the SDE core.
3. Add this additional bundle to your Development Environment distribution, but not to your normal distribution.

4.2.3.4 Providing UI

If you would like to provide additional UI to the rather basic generic invocation wizard of the Development Environment, you can use the full power of Eclipse to add such UI as the Development Environment does not impose any restrictions.

However, please ensure that all relevant functions can also be reached via the Development Environment, as it should be possible to orchestrate the Development Environment without UI.

5 Where to Go from Here

The development process of the core Development Environment is supported by an installation of the TRAC tool, which provides an integration of a Wiki, SVN browser, timeline, and bug tracker.

`http://svn.pst.ifi.lmu.de/trac/sde/`

5.1 Publishing Your Own Tools

The recommended way of publishing tools for others to use is via the Eclipse update site mechanism. Please set up an update site for your tools as soon as they are ready. There is a specific section on the SDE Tools Wiki to list existing tools and their update sites; please add your site there.

5.2 Bug Reporting and Enhancement Requests

The recommended way for reporting bugs or enhancement request is by using the SDE Tools Wiki, which contains a tracker for exactly this purpose. This way you can also check if others are having the same issues, or comment on their proposals.

6 References

6.1 Figures

6.2 Links

Internal Links

Name	**Link**
Service Development Environment SVN	http://svn.pst.ifi.lmu.de/svn/sde/
Service Development Environment Update Site	http://svn.pst.ifi.lmu.de/update/sde/
Service Development Environment Wiki	http://svn.pst.ifi.lmu.de/trac/sde/

External Links

Name	**Link**
Eclipse 3.4 Downloads	http://download.eclipse.org/eclipse/downloads/
Eclipse Project	http://www.eclipse.org/
Java SDK	http://java.sun.com/javase/downloads/index.jsp
OSGi	http://www.osgi.org/
PDE	http://www.eclipse.org/pde/
TRAC	http://trac.edgewall.org/