Integration of Universe Type System Tools into Eclipse

Paolo Bazzi

Semester Project Report

Software Component Technology Group
Department of Computer Science
ETH Zurich

http://sct.inf.ethz.ch/

Summer Semester 2006

Supervised by:
Dipl.-Ing. Werner M. Dietl
Prof. Dr. Peter Müller

Software Component Technology Group
inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

The JML Tools developed at Iowa State University and the Runtime Inference Tool developed at ETH Zurich are both tools supporting a programmer to use the Universe Type System[8] in his Java programs. The Universe Type System is used to be able to restrict the access between different components of a program by adding annotations with a set of new type modifiers into the source code. This semester thesis presents the integration of both the JML and the Runtime Inference tools into Eclipse. This is achieved with two sets of plug-ins. Beside important implementation points of the plug-ins this thesis covers in detail the chosen plug-in architecture, the automatic building and deployment process and a few stumbling blocks creating an Eclipse plug-in.

Contents

Chapter 1

Introduction

1.1 Overview

The Universe Type System provides a way to structure the heap memory into so-called Universes. This structuring allows the programmer to specify and restrict the access to complex object structures. The structuring of the heap is performed by assigning an additional Universe type modifier to each reference. Using the JML Tools the programmer is able to add annotations to his source code using a special comment format. With the different JML Tools the Java source code with the annotations can be syntax-checked and compiled into byte code including instructions that check the specifications at runtime.

The Runtime Inference Tool on the other hand tries to infer the Universe type modifier automatically by tracing the execution of Java programs. The tool was written in former Master Thesis by Frank Lyner[7] and extended by Marco Bär[1].

Both the JML Tools and the Runtime Inference Tools are command line based[1]. The goal of this semester theses is to build a set of plug-ins around the existing tools to make them available and easy to use within Eclipse.

1.2 Outline

This report starts with an introduction to the Eclipse plug-in framework and it possibilities to be extended by user plug-ins. In the beginning of chapter 2 first a short Eclipse introduction is given, followed by a brief Hello World plug-in example. Furthermore in section 2.4 the chosen architecture to include and decouple the existing tools from the newly build plug-in code is presented.

Chapter 3 covers the points of integration of the JML plug-in and explains important implementation details. In the following sections the integration of the JML Checker, JML Compiler, JML Runtime Assertion Checker and the properties page to manage all the settings are presented. In the last section 3.3 two stumbling blocks creating an Eclipse plug-in are explained.

Chapter 4 starts with a short introduction of the Runtime Inference Tool. In the following sections the core plug-in which integrates the existing Runtime Inference tools into Eclipse is presented, followed by two sections about the graphical Runtime Inference Configuration editor and about the Annotations file editor.

To be able to efficiently use the implemented plug-ins, chapter 5 explains how to automatically build the plug-ins, how to pack them into deployable features and how to build an update site to make the new features available through the Web.

The final chapter 6 contains a summary of the achieved results and an outlook to possible extensions both for the JML and the Runtime Inference plug-in.

[1] There are standalone GUI s available for each of the JML Tools, but not as a whole and without Eclipse integration

Chapter 2

Plug into Eclipse

2.1 Eclipse Overview

The Eclipse Platform is an integrated development environment (IDE) for anything, and for nothing in particular. The platform consists only of a small kernel with the core functionality, known as the Platform Runtime. The core is independently of the different programming languages, which can be used within Eclipse. Except of the core, all Eclipse Platform's functionality is located in plug-ins. Figure 2.1 shows the platform UI when no special development tools are loaded.

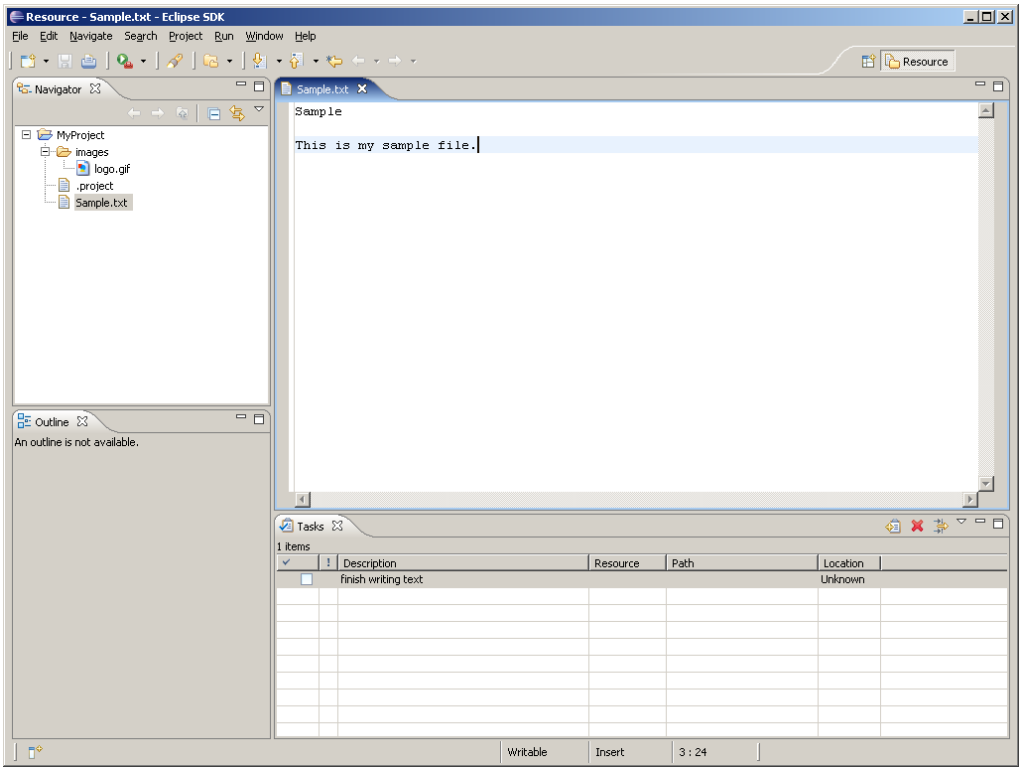


Figure 2.1: Eclipse Platform UI

The navigator view (top left) shows the files in the user's workspace; the text editor (top right) shows the content of a file; the tasks view (bottom right) shows a list of to-dos; the outline view (bottom left) shows a content outline of the file being edited (not available for plain text files).

2.2 Eclipse Plug-in Platform

The Eclipse Platform is built on a mechanism for integrating additional functionality and extending the platform with modules. Plug-ins can add new content types, provide new functions for existing content types and plug-in tool-specific UI contributions to the workspace. When the Eclipse Platform is launched, an integrated development environment (IDE), composed of the set of available plug-ins is presented to the user.

Every plug-in consists of several Java classes (or JAR files) and two additional files describing the plug-in's extension points and dependencies. The file `MANIFEST.MF` contains information about plug-in version, name and classpath. The file `plugin.xml` describes each contribution, which the plug-in adds to Eclipse. The interconnection model is very simple, a plug-in declares any number of extension points and any number of extensions to one or more extension points in other plug-ins. The `plugin.xml` file is omitted, when a plug-in adds no contributions to extension points but for instance only a provides a library file (see also section 2.4).

This separation of code and description of every plug-in facilitates lazy-loading of the plug-in code. An activated plug-in uses the plug-in registry to discover extensions contributed to its extension points by other plug-ins. For example, the plug-in which declares the user preference extension point, can search all contributed user preferences and access their names to construct the preferences dialog. All this can be done with the information from the plug-in registry and without having to load the code of all involved plug-ins. Only when a function provided by a plug-in is executed the necessary classes are loaded. This reduces both the startup time and the memory footprint of Eclipse.

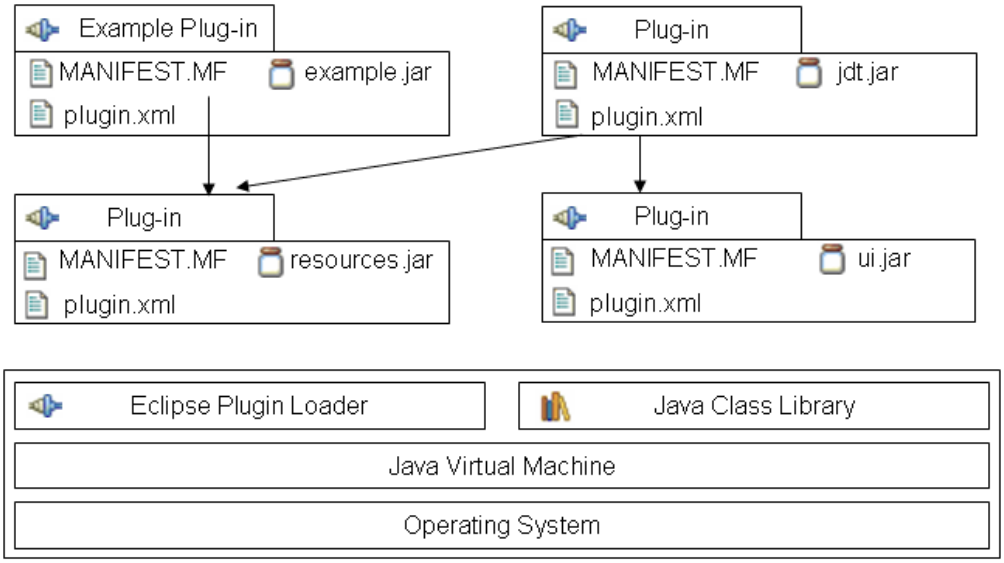


Figure 2.2: Eclipse Plug-in Architecture

Figure 2.2 shows an architectural overview of Eclipse with different plug-ins which are dependent on each other. The Eclipse core functionality is concentrated in the plug-in loader.

2.3 Hello World Plug-in

In the following chapter I will describe, which simple steps are necessary, to add a 'Hello World' button and menu item to the Eclipse UI. This and all further examples are based on Eclipse version 3.2. Compared with previous versions, the concept of using two separate files declaring the plug-in is new. In older versions only a `plugin.xml` was used storing all information together. For our Hello World example first of all we need the `MANIFEST.MF` and `plugin.xml` file.

Listing 2.1: MANIFEST.MF

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: HelloWorld Plug-in
Bundle-SymbolicName: HelloWorld; singleton:=true
Bundle-Version: 1.0.0
Bundle-Localization: plugin
Require-Bundle: org.eclipse.ui
Eclipse-LazyStart: true
```

As shown in listing 2.1 the file MANIFEST.MF declares general information about the plug-in and a list of required plug-ins to use this plug-in (in our case only the `org.eclipse.ui` plug-in).

Listing 2.2: plugin.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
   <extension point="org.eclipse.ui.actionSets">
      <actionSet
            label="Sample Action Set"
            visible="true"
            id="HelloWorld.actionSet">
         <menu label="Sample &amp;amp;Menu"
               id="sampleMenu">
            <separator name="sampleGroup">
            </separator>
         </menu>
         <action
               label="&amp;amp;Sample Action"
               icon="icons/sample.gif"
               class="helloworld.actions.SampleAction"
               tooltip="Hello, Eclipse world"
               menubarPath="sampleMenu/sampleGroup"
               id="helloworld.actions.SampleAction">
         </action>
      </actionSet>
   </extension>
</plugin>
```

The file `plugin.xml` shown in listing 2.2 declares a list of extension points, which this plug-in wants to contribute to. In our example the only extension point is `org.eclipse.ui.actionSets`, which are menu items and buttons on top of the Eclipse workbench. The `actionSet` element consists of a label, a menu, and an action declaration. The action element is the interface between the declarative `plugin.xml` file and the plug-in's implementation in Java code. The `class` attribute of the `action` element declares which class has to be instantiated, when a user clicks the new added button or menu item.

Listing 2.3: SampleAction.java

```
package helloworld.actions;
import org.eclipse.jface.*;
import org.eclipse.ui.*;

/**
 * Our sample action implements workbench action delegate. The action proxy will be created
 * by the workbench and shown in the UI. When the user tries to use the action, this delegate
 * will be created and execution will be delegated to it.
 * @see IWorkbenchWindowActionDelegate
 */
public class SampleAction implements IWorkbenchWindowActionDelegate {
    private IWorkbenchWindow window;
    /** The constructor. */
    public SampleAction() { }

    /**
     * The action has been activated. The argument of the method represents the 'real'
     * action sitting in the workbench UI.
     * @see IWorkbenchWindowActionDelegate#run
     */
    public void run(IAction action) {
        MessageDialog.openInformation(
            window.getShell(),
            "HelloWorld Plug-in",
            "Hello, Eclipse world");
    }

    /**
     * Selection in the workbench has been changed. We can change the state of the 'real'
     * action here if we want, but this can only happen after the delegate has been created.
     * @see IWorkbenchWindowActionDelegate#selectionChanged
     */
    public void selectionChanged(IAction action, ISelection selection) {
    }

    /**
     * We can use this method to dispose of any system resources we previously allocated.
     * @see IWorkbenchWindowActionDelegate#dispose
     */
    public void dispose() { }

    /**
     * We will cache window object in order to be able to provide parent shell for the msg dialog.
     * @see IWorkbenchWindowActionDelegate#init
     */
    public void init(IWorkbenchWindow window) {
        this.window = window;
    }
}
```

Listing 2.3 shows the actual implementation of the new menu action. As described in the Eclipse Platform Plug-in Developer Guide[5] a class providing a new action must implement the `IWorkbenchWindowActionDelegate` interface (which extends the `IActionDelegate` interface).

The important methods are the `run` and the `init` method. The `init` method is called, when the plug-in is initialized and the `run` method is the effective code, which is executed, when the user clicks on the Hello World button or Sample Action menu item (see figure 2.3).

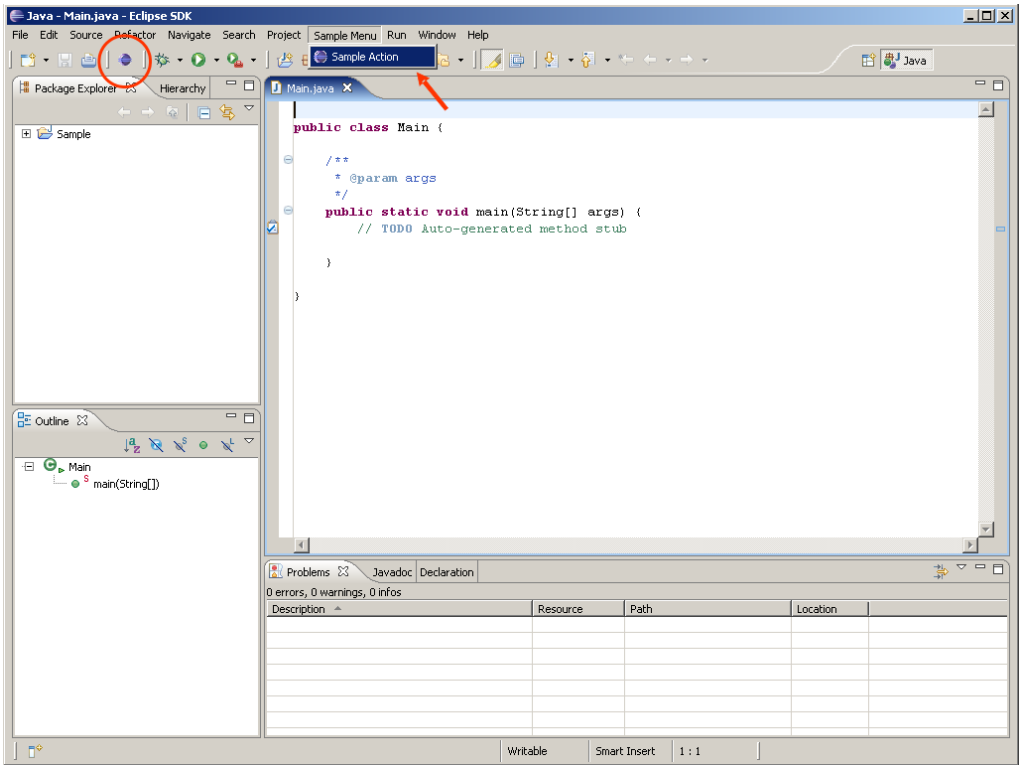


Figure 2.3: Eclipse with HelloWorld Plug-in

When the run method is executed, a message box is shown to the user (see figure 2.4):

Figure 2.4: Eclipse with HelloWorld MessageBox

To install a newly developed plug-in, so that Eclipse can find and load it, the plug-in classes (or a single JAR file) must be placed in a subdirectory of the `/eclipse/plugins` directory together with the files `plugin.xml` and `MANIFEST.MF`.

2.4 Plug-in Architecture Strategy

A plug-in is the smallest unit of Eclipse Platform function that can be developed and delivered separately. Usually a small tool is written with a single plug-in, whereas a complex tool has its functionality split into several plug-ins. As already mentioned in the introduction, except for a small kernel, all of the Eclipse Platform's functionality is located in plug-ins.

Because everything in Eclipse is a plug-in, the only way to add an existing Java class library to Eclipse, is to insert this library into a plug-in, and add the library to the plug-in's runtime classpath. For the plug-in's described further in the chapters 3 and 4 the starting situation was a JAR file containing the tool which had to be integrated into an Eclipse plug-in. In order to create a clean design and decouple the plug-in implementation from the existing tools, I created a resource wrapper plug-in for each new plug-in, which required existing JAR libraries.

For example, the resources plug-in for the Runtime Inference plug-in contains and exports several JAR libraries. Listing 2.4 shows the corresponding `MANIFEST.MF` file:

Listing 2.4: MANIFEST.MF

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Runtime Inference Resources Plug-in
Bundle-SymbolicName: ch.ethz.inf.sct.runtime_inference.resources
Bundle-Version: 1.0.1
Bundle-Vendor: Paolo Bazzi
Bundle-Localization: plugin
Require-Bundle: org.eclipse.core.resources,
 org.eclipse.core.runtime
Export-Package: ch.ethz.inf.sct.runtime_inference.resources,
 ch.ethz.inf.sct.runtime_typinfer
Bundle-ClassPath: ri-resources.jar,
 bcel-5.1.jar,
 commons-beanutils-1.6.1.jar,
 ...
 xmltypes.jar
Eclipse-LazyStart: true
Bundle-Activator: ch.ethz.inf.sct.runtime_inference.resources.RIResourcesPlugin
```

Providing those libraries together with the `MANIFEST.MF` file as a plug-in would suffice, if the plug-in using this resource plug-in would only use the classes provided in the library JARs programatically by calling methods. Because both the JML Plug-in and the Runtime Inference Plug-in use the JAR libraries also as command line parameter for their tools, I had to find another way to encapsulate and access the library JARs from outside the resource plug-in.

My solution for both plug-ins was to add two static classes to the corresponding resource plug-in, providing access to the libraries. For example the class `ResourceConstants` of the Runtime Inference Resources plug-in contains a public constant string with the name of each library:

Listing 2.5: ResourceConstants.java

```
public class ResourceConstants {
    /** name of Runtime Inference Resources Plugin */
    public static final String RUNTIME_INFERENCE_RESOURCES_PLUGIN_NAME
        = "runtime_inference.resources";

    public static final String RUNTIME_INFERENCE_JAR = "runtime_inference2.jar";
```

```
    public static final String BCEL_JAR = "bcel-5.1.jar";
    ...

    /** name of trace agent library (win32) */
    public static final String TRACE_AGENT_WIN32 = "win32_trace_agent.dll";

    /** name of trace agent library (linux) */
    public static final String TRACE_AGENT_LINUX = "uts_type_inferer.so";
}
```

Furthermore the class `RuntimeInferenceResources` in the Runtime Inference Resources plug-in provides several convenience methods to get paths to single libraries or even a classpath array with paths to all required libraries. Listing 2.6 and 2.7 show two example access functions, which locate the path to library files and return their os-specific string representation:

Listing 2.6: Path getter in class RuntimeInferenceResources

```
public static String getPath(String file ) {
    Bundle bundle = Platform.getBundle(
        ResourceConstants.RUNTIME_INFERENCE_RESOURCES_PLUGIN_NAME);
    IPath path = new Path(file);
    URL jarURL = FileLocator.find(bundle, path, null);
    IPath location = null;
    try {
        location = new Path(FileLocator.resolve(jarURL).getPath());
    } catch (IOException e) { /* error handling */ }
    if (location != null) return location.toOSString();
    else return null;
}
```

Listing 2.7: Classpath getter in class RuntimeInferenceResources

```
public static String[] getRuntimeInferenceClasspathAsArray()
{
    ArrayList<String> classpath = new ArrayList<String>();
    Bundle bundle = Platform.getBundle(
        ResourceConstants.RUNTIME_INFERENCE_RESOURCES_PLUGIN_NAME);
    classpath.add(getPath(bundle,ResourceConstants.BCEL_JAR));
    classpath.add(getPath(bundle,ResourceConstants.COM_BEANUTILS_JAR));
    ...
    return (String[]) classpath.toArray(new String[classpath.size()]);
}
```

The main advantage of this design is the possibility to update just the corresponding resource plug-in, if for instance a new version of the Runtime Inference Tool is released. The whole update process is covered separately in section 5.4.

Chapter 3

Java Modeling Language Plug-in

3.1 JML Overview & Tools

The Java Modeling Language (JML)[2] is a behavioral interface specification language that can be used to specify the behavior and the detailed design of Java classes and interfaces including pre and postconditions for methods and class invariants. JML specifications are written in special annotation comments into the source code or in separate annotation files.

There are several tools, developed at Iowa State University, that support the programmer working with JML:

- The JML checker (*jml*) performs parsing and type checking of Java programs and their JML annotations.
- The JML compiler (*jmlc*) also known as the runtime assertion checker, was developed as an extension to the MultiJava[9] compiler. It compiles Java programs annotated with JML specifications into Java byte code. The compiled byte code includes instructions that check JML specifications.
- The JML runtime assertion checker (*jmlrac*) is a version of the *java* command that knows about the necessary runtime libraries for runtime assertion checking. Assertion violations are printed as messages on the console.
- The JML unit tool (*jmlunit*) generates JUnit test classes that rely on the JML runtime assertion checker.
- The JML doc tool (*jmldoc*) produces browsable HTML pages containing both the API and the specifications for Java code, in the style of pages generated by javadoc.

In the following sections I will describe, how the different JML tools were integrated into the Eclipse Java IDE. The goal was to make the JML tools available within Eclipse for normal Java projects, without having to change anything in existing projects in order to be able to use the JML tools. The Eclipse Java editor remains the standard editor for Java source files and is also used to insert the JML annotations into the source code. All known functionality like syntax highlighting and on-the-fly syntax correction for the Java code are remain available.

3.2 JML Integration into Eclipse

3.2.1 JML Checker

The JML Checker was made available in the context menu of each Java source code file. By right-clicking a Java file and choosing JML TOOLS | JML CHECKER the JML Checker is executed in the background (see figure 3.1). The checker is also available for the other known file extensions of JML files (*.jml, etc).

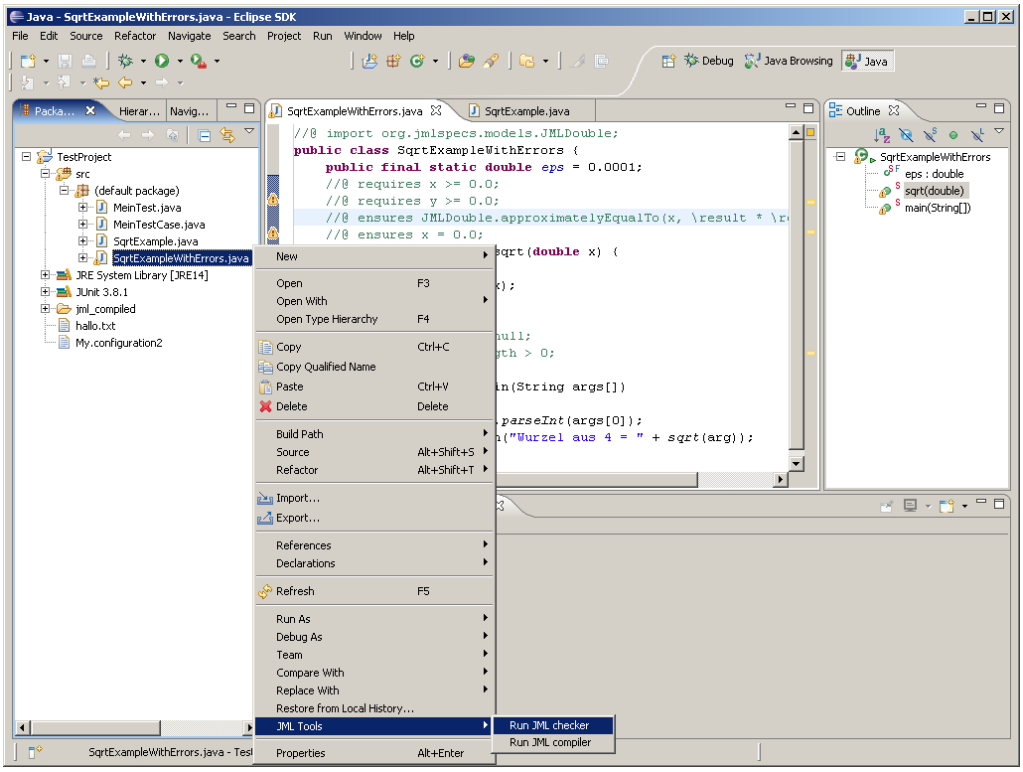
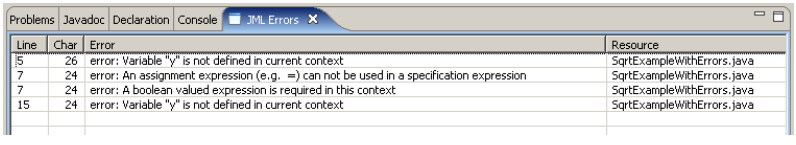
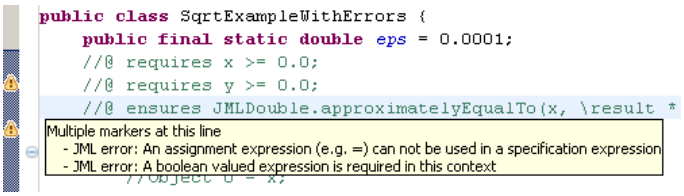


Figure 3.1: Run JML Checker on SqrtExampleWithErrors.java

After finishing the checks, the errors are parsed and shown in a JML Error View. The errors are also shown as warnings in the Eclipse problem view and by double-clicking the warning, the cursor is automatically positioned at the error location in the Java source code editor.



The JML errors are also visible in the Java editor as Eclipse warning marker:



But what happens in the background, when a user runs the JML compiler? As mentioned in chapter 2.3 a menu action contribution class must implement the interface `IObjectActionDelegate`. In the JML Plug in this is done in the class `jml.plugin.internal.ui.RunJmlCheckerAction`. To separate the menu item action from the actual execution of the JML checker, the `run` method of `RunJmlCheckerAction` just forwards the call to the `JmlPlugin` class, which is implemented as a singleton. In order to not block the Eclipse UI thread, the call to the JML checker tool is encapsulated in a workspace runnable, which is executed in an own thread. Listing 3.1 shows the main parts of `RunJmlCheckerAction.java`, some error handling and helper methods are omitted.

Listing 3.1: RunJmlCheckerAction.java

```
public class RunJmlCheckerAction extends RunAction {
    public void run(IAction action) {
        if (! (selection instanceof IStructuredSelection)) return;
        IStructuredSelection structured = (IStructuredSelection) selection; // actual selection
        try { addFiles(structured); } catch (CoreException e) { /* error handling */ }

        showView(JmlErrorView.ID); // open JML Error view

        ProgressMonitorDialog dialog= new ProgressMonitorDialog(getShell());
        try {
            dialog.run(true, true, new IRunnableWithProgress() {
                public void run(IProgressMonitor monitor) throws InvocationTargetException {
                    try {
                        JmlPlugin.getDefault().runChecker(getFiles());
                    } catch (CoreException e) {
                        throw new InvocationTargetException(e);
                    }
                }
            });
        } catch (InterruptedException e) { /* error handling */ }
        } catch (InvocationTargetException e) { /* error handling */ }
    }
}
```

To not concentrate all code in the `JmlPlugin` class, the call is again forwarded to a separate class, implementing the effective call to the JML checker. The `JmlPlugin` class is also responsible for handling the different listeners, which listen to the JML-error events.

Listing 3.2: JmlPlugin.java

```
public class JmlPlugin extends AbstractUIPlugin {
    public void runChecker(ArrayList<IFile> files) throws CoreException {
        if ( files . size () > 0) {
            addListener( listener );
            new JmlCheckerRunner().run(files);
            removeListener( listener );
        }
    }
}
```

The class `JmlCheckerRunner` (see listing 3.3) constructs the JmlOptions object, which contains all required options to run the JML Checker. The options are extracted from the project settings (see section 3.2.4). As already explained in section 2.4, the path's to the JML runtime libraries are accessed with helper methods and added to the JML checker classpath. After the effective call to `org.jmlspecs.checker.Main.compile`, the output is parsed and a `fireJmlFinishedEvent` is sent to all listeners.

Listing 3.3: JmlCheckerRunner.java

```
public class JmlCheckerRunner extends JmlRunner {
    public boolean run(IFile file ) throws CoreException {
        JmlPlugin.getDefault().fireJmlStarted( file );
        String [] arg = new String[] { file .getLocation().toOSString() };
        JmlOptions jmlo = new JmlOptions();
        IProject proj = file .getProject ();
        JmlPropertiesProvider checkerProps =
            JmlPropertiesProvider.getPropertiesProvider(
            ResourceConstants.CHECKER_DEF_PROPERTIES_FILE);

        // copy all options to JmlOptions object
        jmlo. set_admissibility (checkerProps.getProperty(
            JmlCheckerProperties.ADMISSIBILITY,proj));
        jmlo. set_assignable (checkerProps.getBooleanProperty(
            JmlCheckerProperties.aSSIGNABLE,proj));
         [...]

        // Add JML JARs & specs to JmlOptions classpath
        String classPath = checkerProps.getProperty(JmlCheckerProperties.CLASSPATH,proj);
        jmlo. set_classpath (jmlo.classpath () + ResourceConstants.PATH_SEPARATOR
            + getJmlClasspath());

        // add destination to JmlOptions
        String workingDir = file .getProject (). getLocation (). toOSString();
        String outDir = checkerProps.getProperty(JmlCheckerProperties.DESTINATION, proj);
        if (!outDir.startsWith(workingDir)) {
            jmlo. set_destination (workingDir + ResourceConstants.FILE_SEPARATOR + outDir);
        } else {
            jmlo. set_destination (outDir);
        }
        ByteArrayOutputStream baos = new ByteArrayOutputStream();

        org.jmlspecs.checker.Main.compile(arg,jmlo,baos); // call JML tool

        ArrayList<JmlErrorElement> errors = new ArrayList<JmlErrorElement>();
        boolean result = parseOutput(baos, file, errors ); // parse result stream
        JmlPlugin.getDefault().fireJmlFinished( file , errors );
        return result ;
    }
}
```

3.2.2 JML Compiler

The JML Compiler is integrated into Eclipse in a very similar way as the JML Checker. It can also be called in the JML TOOLS context menu, executes in the background and puts his errors in the JML Error View.

Integrating the JML Compiler, I encountered one special problem: the output classes. If the output classes are put into the same directory, as Eclipse puts his compiled classes, every time the Eclipse builder runs, the files are overwritten. But if I would have disabled the Eclipse Java compiler, also all context sensitive error messages while editing a Java file would have disappeared. The solution I implemented is to specify an own JML Compiler output directory in the project settings of a JML project (see 3.2.4 for further information). With that solution, the classes are compiled both, automatically by the Eclipse Java compiler and manually by the JML Compiler. Advantage of this solution is, that the classes can still be executed as a normal Java application for testing purposes, or alternatively be executed with the JML runtime assertion checker (see chapter 3.2.3).

3.2.3 JML runtime assertion checker

The goal of integrating the JML runtime assertion checker, was to provide a very simple way to run jml-compiled classes within Eclipse. To accomplish this, a new launch configuration type was added. By right-clicking a class containing a main type and choose RUN-AS | JML RAC the selected class is launched with the necessary Java virtual machine arguments. The output and possible exceptions are printed in the normal Eclipse console (see figure 3.2.3).

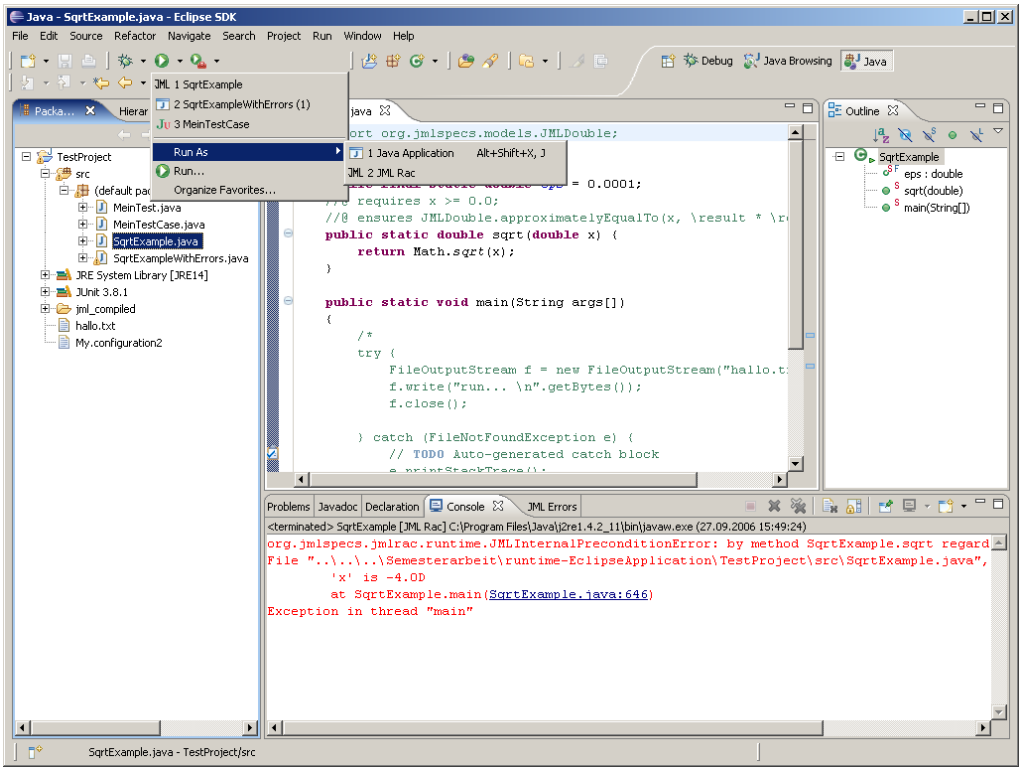


Figure 3.2: Run JML runtime assertion checker

To add the new JML Rac launch configuration in Eclipse, I used the extension point `org.eclipse.debug.core.launchConfigurationTypes`. Implementing the new launch configuration can start at different points, depending on how much existing code you want to reuse. Existing Eclipse plug-ins provide different abstract and normal classes with several convenience methods, ready to reuse. I used the class `JavaLaunchDelegate`, which already provides a lot of functionality to run local Java applications. Listing 3.4 shows an extract of the class `JmlRacLaunchConfigurationDelegate`. The methods I had to override were the getVMArguments method, to which I added the JML runtime library to the boot classpath and the getClasspath method, to which I added the directory with the JML-compiled classes to the classpath. Furthermore I added a pre-launch check, to assure that the class file of the class to be launched is available.

Listing 3.4: JmlRacLaunchConfigurationDelegate.java

```
public class JmlRacLaunchConfigurationDelegate extends JavaLaunchDelegate {
  public String getVMArguments(ILaunchConfiguration configuration) throws CoreException {
    StringBuffer arguments = new StringBuffer(super.getVMArguments(configuration));
    arguments.append("␣-Xbootclasspath/a:\""
        + JmlHelper.findJmlRuntimeJarLibraryPath() +"\"");
    return arguments.toString();
  }

  public String[] getClasspath(ILaunchConfiguration configuration) throws CoreException {
    IRuntimeClasspathEntry[] entries =
        JavaRuntime.computeUnresolvedRuntimeClasspath(configuration);
    entries = JavaRuntime.resolveRuntimeClasspath(entries, configuration);
    List userEntries = new ArrayList(entries.length);
    for (int i = 0; i < entries.length; i++) {
      if (entries[i].getClasspathProperty() ==
          IRuntimeClasspathEntry.USER_CLASSES) {
        String location = entries[i].getLocation();
        if (entries[i].getType() == IRuntimeClasspathEntry.PROJECT)
        {
          IJavaProject entryPro = (IJavaProject) JavaCore.create(entries[i].getResource());
          IJavaProject configurationPro = getJavaProject(configuration);
          if (entryPro.equals(configurationPro))
          {
            String workingDir = configurationPro.getProject().getLocation().toOSString();

            JmlPropertiesProvider props = JmlPropertiesProvider.getPropertiesProvider
              (ResourceConstants.COMPILER_DEF_PROPERTIES_FILE);

            String outDir = props.getProperty(JmlCompilerProperties.DESTINATION
              , configurationPro.getProject());

            location = workingDir + ResourceConstants.FILE_SEPARATOR + outDir;
          }
        }
        if (location != null) { userEntries.add(location); }
      }
    }
    return (String[]) userEntries.toArray(new String[userEntries.size()]);
  }
}
```

3.2.4 Project Properties

The JML Checker and Compiler support a large set of options to customize the execution. To simplify the configuration of these tools, the JML Plug-in adds two properties pages in the project settings page (see figure 3.4).
In this section I'll explain the adding and handling of project properties in detail. Adding properties support for a project consists of two parts. The first part, is to declare, load and store the properties to the project resource object. Listing 3.5 shows an example how to store a (persistent) property to a project object.

Listing 3.5: Reading a property

```
IProject p = getProject();
p.setPersistentProperty(
    new QualifiedName(JmlPlugin.PLUGIN_ID,JmlCheckerProperties.ASSIGNABLE),true);
```

To simplify the handling of getting the right properties in case of a project property which was not already set, I added a static class, handling the whole property mechanism. The `getProperty` method of the class `JmlCheckerProperties` provide the project property, if there is one which was set before or if no project property is set, it returns the default property. The class also contains convenience methods to directly get boolean or int properties.

The default properties are specified in a separate `.properties` file (see listing 3.6).

Listing 3.6: jmlchecker.default.properties

```
# Default options for JML Checker (JmlOptions)
#
#Check that a method with an assignable clause does not call methods that do not have an
#assignable clause [true]
jml.plugin.checker.default.Assignable=true

#Shuts off all typechecking informational messages [false]
jml.plugin.checker.default.Quiet=false
...
```

The second part of adding properties functionality to a project is to declare a new class extending `PropertyPage`, which provides a GUI to select and change the values of the properties. Two example properties pages are shown in figure 3.3 and 3.4.

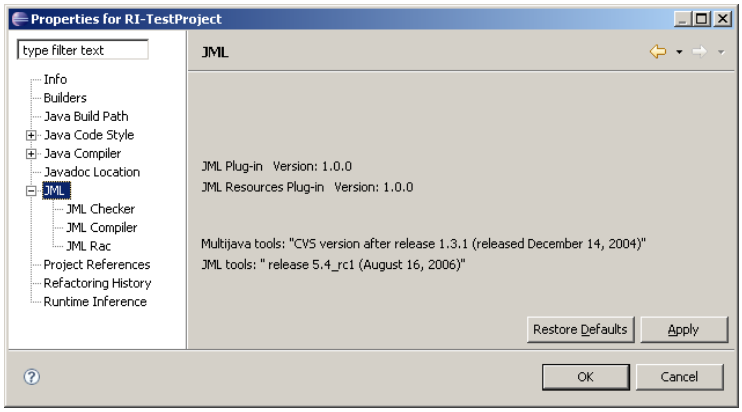


Figure 3.3: JML root properties page

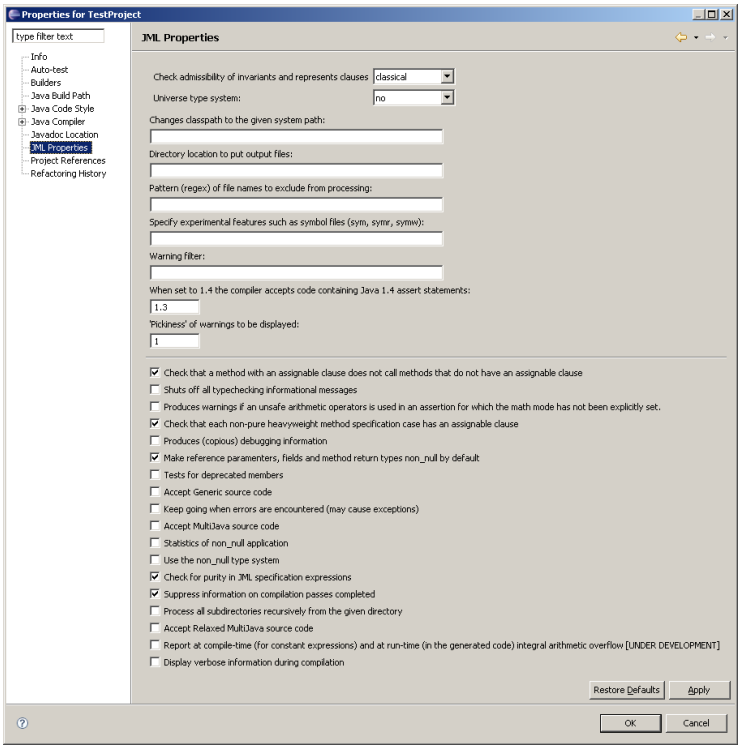


Figure 3.4: JML Checker properties page

3.3 Difficulties implementing a Plug-in

This section describes two exemplary difficulties I encountered while creating the JML Plug-in. The first one is a problem when contributing new object actions in a context-menu of an item and the second one is about how to contribute a new launch configuration which is also available in different context-menus.

3.3.1 Contribute a new Object Action

Providing a new context menu item in Eclipse is done with the extension point `org.eclipse.ui.popupMenus` and an object contribution. When declaring an object contribution in the `plugin.xml` file, you have to specify, for which objects the new menu item should appear. This sounds easy, but is a little bit tricky because of the different views in Eclipse. The most used views in the Eclipse Java development tools to browse the workspace are the Package Explorer and the Navigator view. Therefore it would be convenient that a contributed context menu item appears in both views. But for instance right-clicking on a file item in the Package Explorer and in the Navigator view doesn't provide the same object as event source. The reason is, that the Navigator view provides a direct view on the file system and returns `IResource` objects (`IFile`, `IFolder`, `IProject` etc). On the other hand, the Package Explorer provides a view into the Java project and returns `IJavaObjects` (`ICompilationUnit`, `IClassFile`, `IJavaProject` etc).

Nevertheless, there are several solutions to that problem in Eclipse. I will present three of them for the example of a new context menu for a file. A file in the Navigator view is represented as `IFile` object and in the Package Explorer corresponds to a `ICompilationUnit` object.

A) PlatformObject contribution

The simplest way to declare a menu contribution for both views would be to use their common super type `org.eclipse.core.runtime.PlatformObject`. See listing 3.7 for an example declaration:

Listing 3.7: plugin.xml with PlatformObject declaration

```
<extension point="org.eclipse.ui.popupMenus">
    <objectContribution
        id="PopupExample.objectContribution1"
        objectClass="org.eclipse.core.runtime.PlatformObject">
        <action class="popupexample.popup.actions.NewAction"
            enablesFor="1"
            id="PopupExample.action1"
            label="New Action for all PlatformObjects"/>
    </objectContribution>
</extension>
```

The big disadvantage of this approach is, that the new menu item is visible for all PlatformObject. So it appears for each file, folder, project etc. Certainly after a click on the item, you can filter out all not relevant objects and run the real menu action only for files, but nevertheless the item appears in the whole workspace at points where it is not useful.

B) IFile adaptable contribution

The second alternative to achieve a solution for both views is to declare the contribution for `IFile` objects and additionally mark the extension as adaptable. Adaptable object contributions are visible for all types which adapts to an `IResource` resp. in this case `IFile` object using and implementing the Eclipse `IAdaptable` interface. See listing 3.8 for an example declaration:

Listing 3.8: plugin.xml with adaptable IFile declaration

```
<extension point="org.eclipse.ui.popupMenus">
    <objectContribution
        adaptable="true"
        id="PopupExample.contribution1"
        objectClass="org.eclipse.core.resources.IFile">
        <action
            label="New Action for IFile, adaptable"
            class="popupexample.popup.actions.NewAction"
            enablesFor="1"
            id="PopupExample.newAction2">
        </action>
    </objectContribution>
</extension>
```

This solution is easy to use, because there are just little changes needed in the `plugin.xml` file. The disadvantage of this solution is, that you only get a `IFile` object as event source in your action code. That means, that you loose the possibility to treat a `ICompilationUnit` object differently from a `IFile` object, depending on where the menu item was selected. But if the contributed menu action for instance needs only the file name, a IFile object is sufficient to accomplish the task. From an other point of view this solution can be very useful, when interested only in the file name, then because a lot of objects in the workspace are adaptable to an `IFile` object, with one single extension you contribute the menu item into a lot of different places, see figure 3.5.

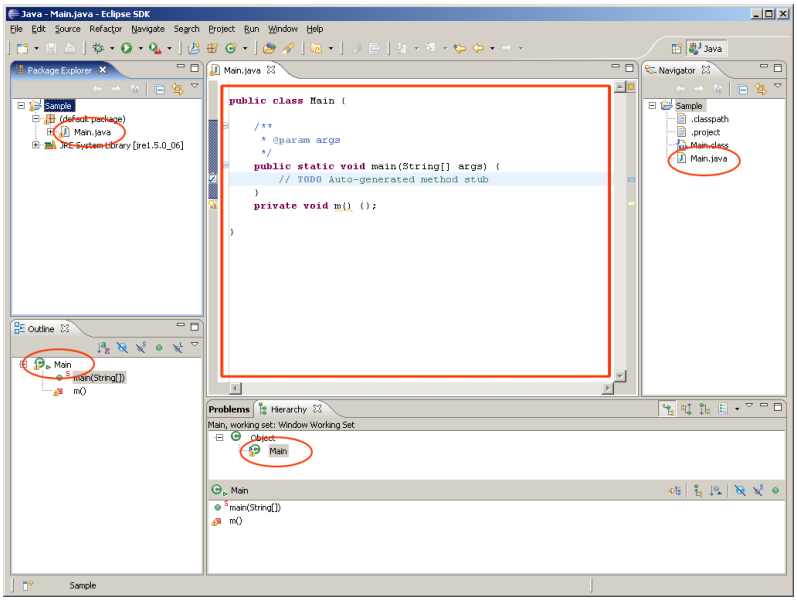


Figure 3.5: Extension points for IFile object contributions

C) IFile and ICompilationUnit contribution

The most complex solution is to declare the menu item contribution twice, once for `IFile` objects and once for `ICompilationUnit` objects. However, the class containing the real menu action can be the same in both cases. See listing 3.9 for an example declaration:

Listing 3.9: plugin.xml with declarations for IFile and ICompilationUnit

```
<extension point="org.eclipse.ui.popupMenus">
    <objectContribution id="PopupExample.contribution1"
        objectClass="org.eclipse.core.resources.IFile">
        <action label="New Action for IFile"
            class="popupexample.popup.actions.NewAction"
            enablesFor="1"
            id="PopupExample.action2"/>
    </objectContribution>
    <objectContribution id="PopupExample.contribution2"
        objectClass="org.eclipse.jdt.core.ICompilationUnit">
        <action label="New Action for ICompUnit"
            class="popupexample.popup.actions.NewAction"
            enablesFor="1"
            id="PopupExample.action1"/>
    </objectContribution>
</extension>
```

The main advantage of this solution is, that within the action code, you have full access to the object, which was selected. Listing 3.10 provides an example, how to access the selected object depending on its type.

Listing 3.10: Handling selection depending its type

```
public void run(IAction action) {
    if (curSelection instanceof IStructuredSelection)        {
        IStructuredSelection structSel = (IStructuredSelection)curSelection;
        Object obj = structSel.getFirstElement();
        if (obj instanceof IFile) {
            // handling IFile
        } else if (obj instanceof ICompilationUnit) {
            // handling ICompilationUnit
        }
    }
}
```

In my plug-ins depending on the situation and the requirements I used the solution described in section B or C. Apart from menu contributions the issue described in this chapter appears while declaring several other contributions. For instance nearly the same the problem occur when declaring properties pages for projects, which are represented as `IJavaProject` object in the Package Explorer and as `IProject` objects in the Navigator.

3.3.2 Launching any kind of resource

As described in section 3.2.3 adding a new launch configuration for Java applications is quite easy, the only thing to do is to extend the class `JavaLaunchDelegate` and override the necessary methods. But this is not the plain truth, providing a class extending the abstract class `JavaLaunchDelegate` indeed adds the launch configuration in the launch configuration menu, but to be able to launch programs with the RUN-AS context menu item, you need to provide an additional extension to the launch configuration shortcut extension point (`org.eclipse.debug.ui.launchShortcuts`).

A launch shortcut must implement the interface `ILaunchShortcut`, see listing 3.11.

Listing 3.11: Interface ILaunchShortcut

```
public interface ILaunchShortcut {
    /**
     * Locates a launchable entity in the given selection and launches
     * an application in the specified mode. [...]
     */
    public void launch(ISelection selection, String mode);

    /**
     * Locates a launchable entity in the given active editor, and launches
     * an application in the specified mode. [...]
     */
    public void launch(IEditorPart editor, String mode);
}
```

The first method is called for selections in the workbench like the Package Explorer or the Navigator view. The second method is called for selections in editors. The class implementing ILaunchShortcut is mainly responsible to figure out which main method shall be launched. The mechanism is like the one described in 3.3.1 B), the idea is to allow the user to launch an application from different points in Eclipse and then figure out programatically which method to launch exactly. Therefore the following steps are required:

- Search the main method of the selected items
- If there is more than one main method within the selected items, let the user choose which main method to launch
- Search if there is an existing launch configuration for this main method. If there is more then one, let the user choose which configuration to use
- If this method was never launched before, create a new launch configuration with default parameters
- Launch this launch configuration

For further detailed information, see the class `JmlRacLaunchShortcut` in the package `ch.ethz.inf.sct.jml.plugin.internal.ui`.

Chapter 4

Runtime Inference Plug-ins

4.1 Runtime Inference Tool Overview

Using the Universe Type System Java programs can be annotated to restrict the access between different components. The Runtime Inference tool developed by Frank Lyner [7] and extended by Marco Bär [1] infers these type annotations from executable Java programs. This is done through runtime observation and without static code analysis of the Java program. Figure 4.1 gives an overview of the execution order of the different parts of the Runtime Inference Tool.

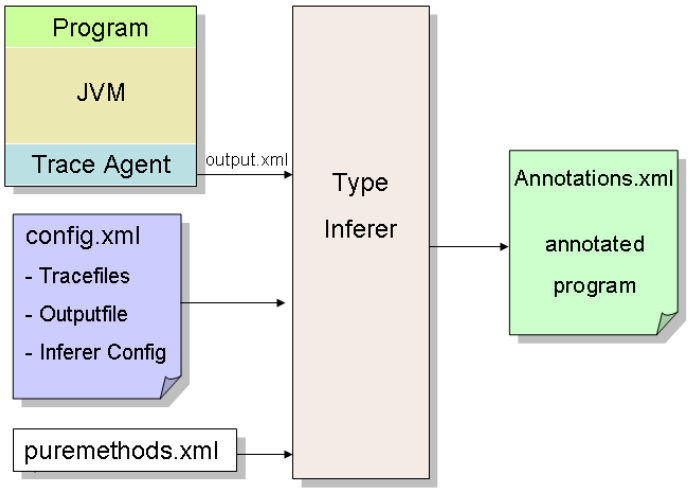


Figure 4.1: Runtime Inference Tool overview

- The first step of the Runtime Inference algorithm is to monitor the execution of a normal Java program. This is done with a special agent, attached to the Java virtual machine. The output of the agent is an XML files containing the trace information.
- After running the agent in a second step an XML file with configuration information has to be created.
- Together with the agent output file and the configuration file a pure methods file with a few purity annotations is provided as input to the Type Inferer.
- The final step is the Type Inferer, which implements the real logic to infer the annotations and provides another XML document with the inferred annotations as output.

The handling of the original Runtime Inference tool is command-line based and not very comfortable. The goal of the Runtime Inference Eclipse plug-in was to simplify the single steps of the tool and to make them available as new actions for Java projects within Eclipse. The Runtime Inference plug-in is divided into three parts. The part with the core functionality is described in section 4.2. Section 4.3 presents the configuration-related Runtime Inference Configuration plug-in and finally in section 4.4 the Annotations Editor plug-in is presented.

4.2 Runtime Inference Core Plug-in

4.2.1 Trace Agent

The Trace Agent was integrated as new launch configuration type Runtime Inference Agent. Running a Java program with this launch configuration automatically adds the necessary virtual machine arguments with the path to the trace agent to the launched program. Additionally the name of the class to trace and the name of the output file are automatically added as arguments for the trace agent. The user can specify the output folder and file name in an additional configuration tab, which was added to the new launch configuration dialog (see figure 4.2). Additionally the user can choose whether or not he wants to be asked for the name of the output file before every run of the trace agent.

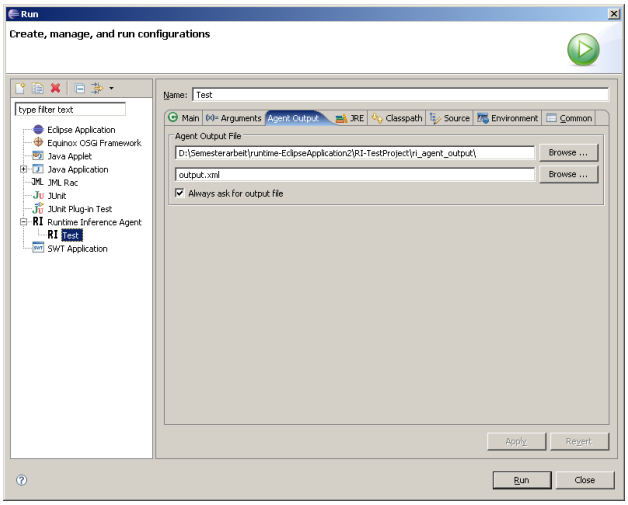


Figure 4.2: Launch Configuration with trace agent tab

4.2.2 Type Inferer

The Type Inferer can be started by right-clicking on a configuration file and selecting Runtime Inference | Run Type Inferer. The plug-in then automatically calls the Type Inferer with the selected configuration file as argument. Details about the runtime inference configuration files are explained in section 4.3.

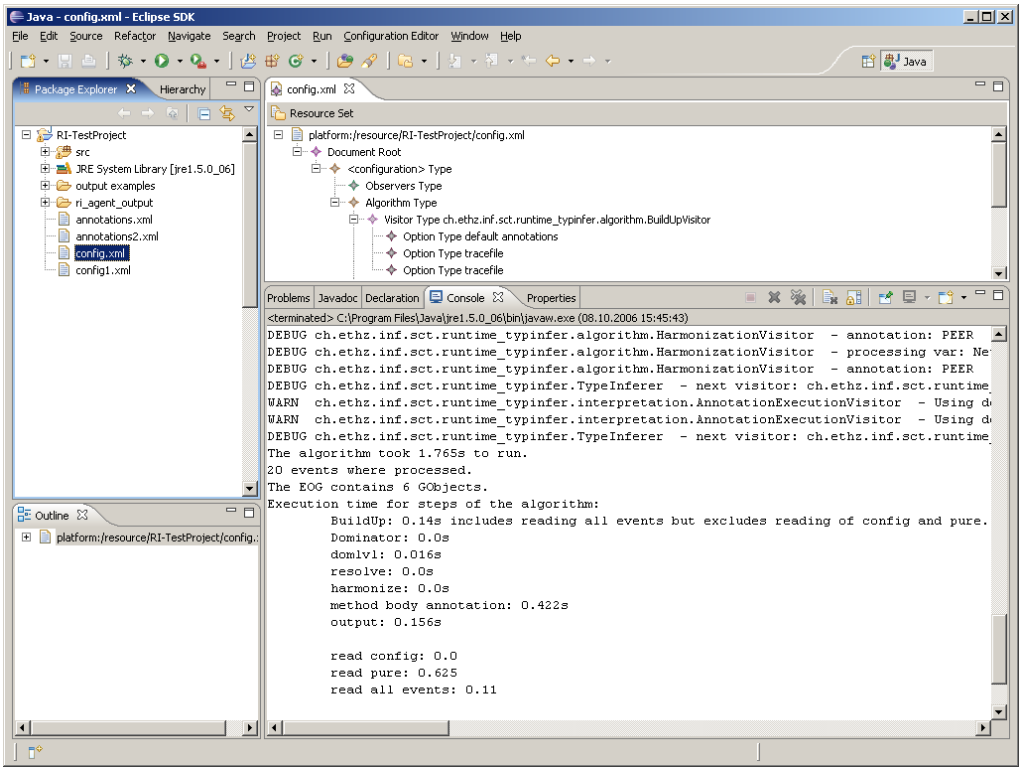


Figure 4.3: Type Inferer Output

The Type Inferer is called in an own virtual machine, in order that Eclipse containing the workspace is protected, if a crash should occur. Eclipse provides an easy to use framework to start a new virtual machine programatically. Listing 4.1 shows a simplified code example (error handling and a few details omitted).

Line 1 shows how to extract a reference to the Java Project from a workspace file reference. To launch a virtual machine you need a reference to a `IVMInstall` object, a `IVMRunner` object and a `VMRunnerConfiguration` object. In line 5 and 6 the default project classpath is computed and the virtual machine assigned to the project is stored in a local variable. Lines 12 to 18 shows how to configure the `VMRunnerConfiguration`. Finally a `ILaunch` object is created (line 20) and passed to the `IVMRunner` and the `LaunchManager` of the Eclipse DebugPlugin. The LaunchManager takes care of all registered launch configurations and notifies registered launch listeners.

Listing 4.1: Starting a virtual machine

```
1  public void run(IFile file ) throws CoreException {
2      IJavaProject project = (IJavaProject)JavaCore.create((IResource)file.getProject());
3      String[] projectClasspath = null;
4      IVMInstall vmInstall = null;
5      try {
```

```
6           projectCP = JavaRuntime.computeDefaultRuntimeClassPath(project);
7           vmInstall = JavaRuntime.getVMInstall(project);
8       } catch (CoreException e) { /* error handling */ }
9       if (vmInstall == null) vmInstall = JavaRuntime.getDefaultVMInstall();
10      IVMRunner vmRunner= vmInstall.getVMRunner(ILaunchManager.RUN_MODE);
11
12      String mainClass = ResourceConstants.TYPEINFERER_CLASS;
13      VMRunnerConfiguration vmConfig= new VMRunnerConfiguration(mainClass, projectCP);
14      vmConfig.setWorkingDirectory(file.getProject().getLocation().toOSString());
15
16      String [] vmArgs = { "-D" + ResourceConstants.TYPEINFERER_CONFIG_PARAM
17              +"="+ file.getLocation().toOSString() };
18      vmConfig.setVMArguments(vmArgs);
19
20      ILaunch launch= new Launch(null, ILaunchManager.RUN_MODE, null);
21      try {
22          vmRunner.run(vmConfig, launch, null);
23          DebugPlugin.getDefault().getLaunchManager().addLaunch(launch);
24      } catch (CoreException e) { /* error handling */    }
25  }
```

4.2.3 Project Properties

Like the JML Plug-in (described in section 3.2.4) the Runtime Inference Plug-in has also properties which can be managed in an own properties page (see figure 4.4). The properties set in the properties page are valid as long as they are not overwritten in a specific Runtime Inference launch configuration. The default values are stored in the Runtime Inference Resources Plug-in within the file `runtimeinference.default.properties`.

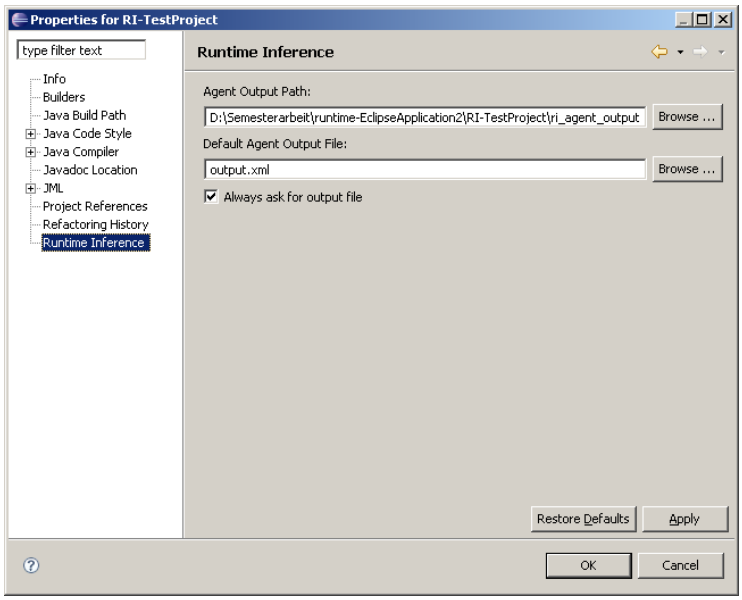


Figure 4.4: Runtime Inference Project Properties

4.3 Configuration Editor Plug-in

In order to easily manage the XML-based configuration file for the Runtime Inference Tool, a special editor was added to the Runtime Inference Plug-in. The editor is based on the Eclipse Modeling Framework (EMF)[4]. This framework allows you to automatically generate both, an Eclipse plug-in with a Java representation model of an XML schema and two other plug-ins providing am example editor to edit the specific XML. Starting from this generated editor, I changed and reduced the graphical interface to one single view and added two functions to the context menu of the configuration file. The functions simplify the often used tasks of adding a trace file to the configuration and adding the default purity annotations file to the configuration. Figure 4.5 shows the configuration editor with the associated properties view, which allows to edit the selected element.

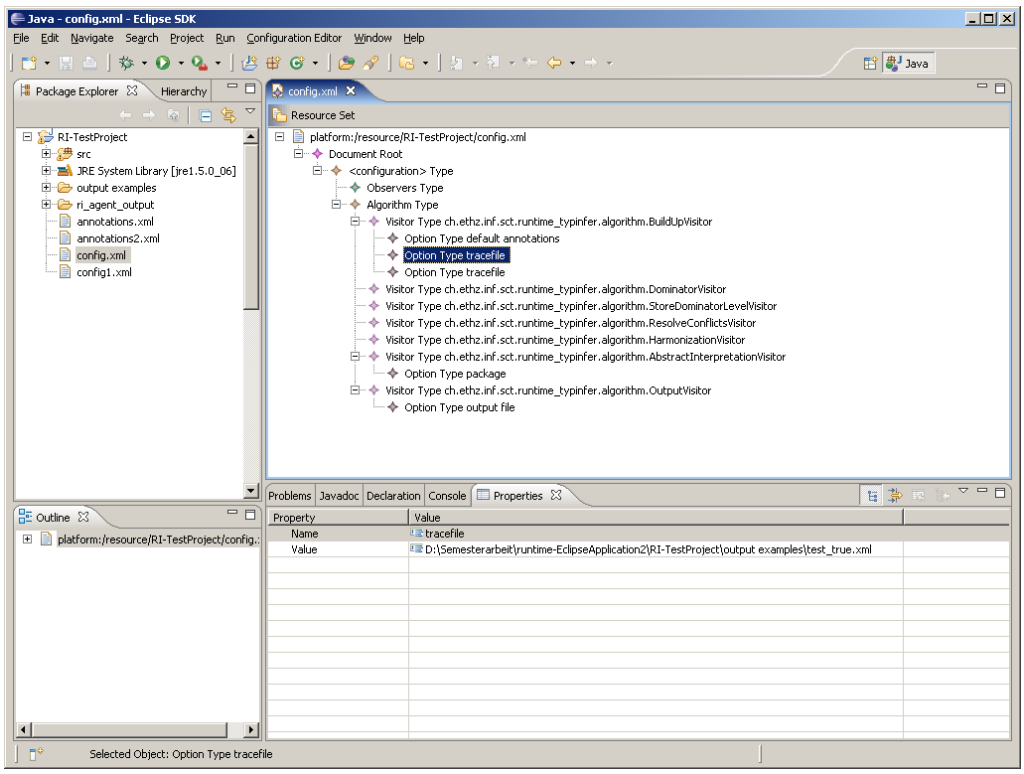


Figure 4.5: Runtime Inference Configuration Editor

4.3.1 Content-sensitive Eclipse contributions

Extending the EMF-generated editor I also used a new Eclipse 3.2 feature, the content-sensitive enablement of additional context-menu items. In previous Eclipse versions to add a new context-menu contribution you had to specify the object class and the file extension for which the menu item should be visible. Unfortunately a contribution for a certain XML file appeared in the context-menu of every XML file in the workspace. Eclipse 3.2 newly provides a way to specify content-sensitive contributions to file types.

To use of this feature, first of all you have to declare the new content type. The content type description contains the file extension and a reference to a content describer class. For XML types there is an already provided default content describer class, which is able to compare the root element name with a given parameter name in order to determine if a file matches the content type. Listing 4.2 shows how the configuration content type is declared in the Runtime Inference plug-ins `plugin.xml` file. In this case, the value of the root element must be `configuration`;

Listing 4.2: Declaring a new content type

```
<extension point="org.eclipse.core.runtime.contentTypes">
    <content-type base-type="org.eclipse.core.runtime.xml"
        file-extensions="xml"
        id="runtime-inference-plugin.configuration.content-type"
        name="runtime-inference-plugin.configuration.content-type"
        priority="high">
        <describer class="org.eclipse.core.runtime.content.XMLRootElementContentDescriber">
            <parameter name="element" value="configuration"/>
        </describer>
    </content-type>
</extension>
```

In a second step within the visibility element of the new contribution the declared content type is assigned to the new content-menu contribution, see listing 4.3

Listing 4.3: Declaring a new content-sensitive menu action

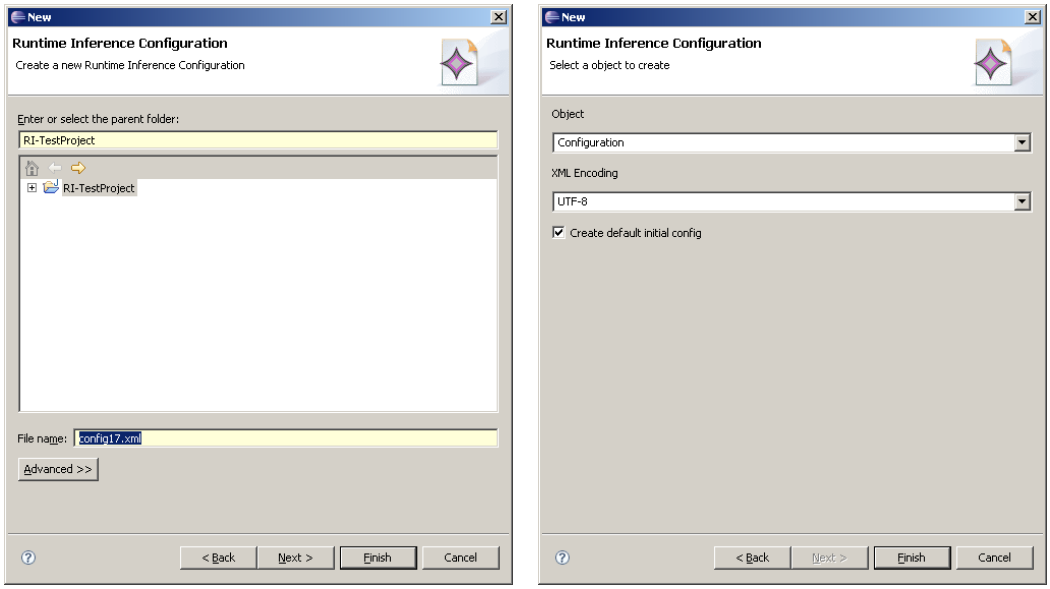
```
<extension point="org.eclipse.ui.popupMenus">
    <objectContribution adaptable="true"
        id="runtime_inference.plugin.runtypinferer"
        nameFilter="*.xml"
        objectClass="org.eclipse.core.resources.IFile">
    <visibility>
        <objectState name="contentTypeId"
            value="runtime-inference-plugin.configuration.content-type"/>
    </visibility>
    <action
        class="ch.ethz.inf.sct.runtime_inference.plugin.internal.ui.AddDefaultAnnotationsAction"
        enablesFor="1"
        id="runtime_inference.plugin.runtypinferer.addDefaultPuremethodsAction"
        label="Add default puremethods.xml"/>
</extension>
```

The content-sensitive enablement of Eclipse contributions is not only available for menu items but also for a large range of other extension points. For further information see also [3].

4.3.2 New configuration wizard

The editor plug-in generated with EMF also provides a wizard for creating a new blank configuration file. Starting from this wizard I added the functionality to generate a new configuration file from a configuration template, contained in the Runtime Inference Resources Plug-in.

By right clicking a project and select New | Others | Runtime Inference Configuration with a few steps you can add a new configuration file to the project, see figure 4.6.



(a) Specify filename (b) Specify encoding and initial config

Figure 4.6: New Configuration Wizard

4.4 Annotations Editor Plug-in

In common with the Configuration Editor plug-in, the Annotations Editor plug-in consists also of three plug-ins (model plug-in, edit plug-in providing model edit capabilities and the editor UI plug-in), build with the Eclipse Modeling Framework (EMF)[4]. I made a few changes to the look-and-feel in the graphical interface of the editor, to achieve a better usability for the visualisation of annotations, see figure 4.7.

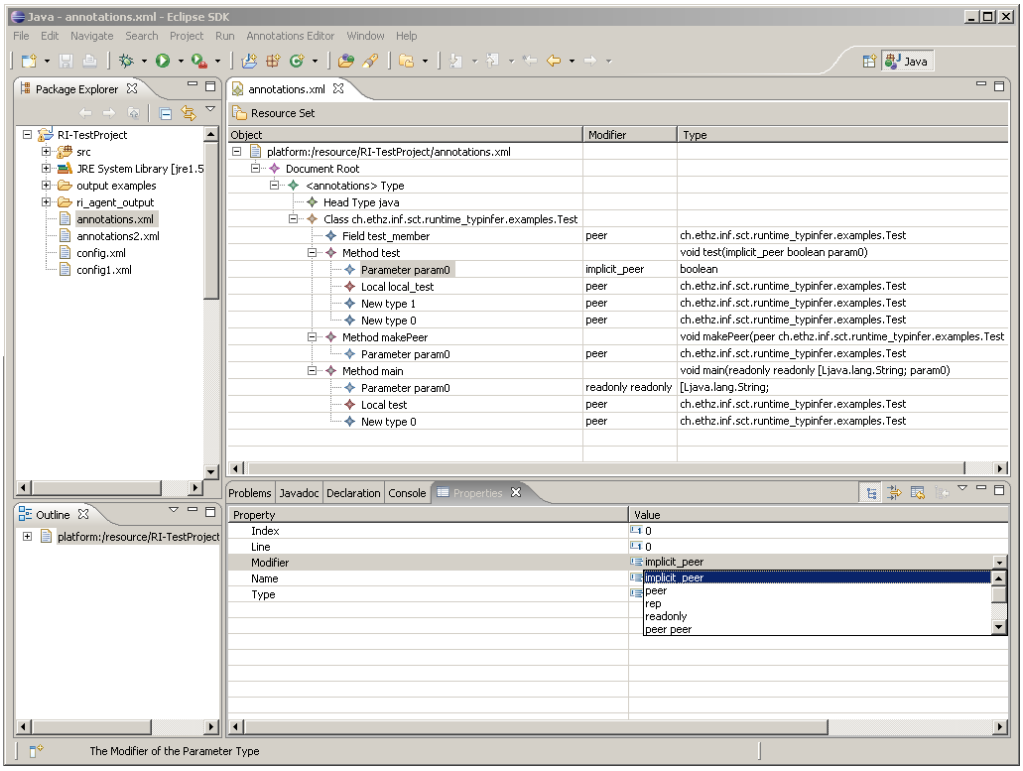


Figure 4.7: Annotations Editor

The Properties View in the bottom shows additional informations about a selected item and allows to change its universe type modifier. The main editor window shows the annotations tree of an annotated class. For each method, a short signature summary is displayed in the TYPE column. Additional information become visible, if the tree items are unfolded. To add or remove annotations, there is a context menu for each tree item, see figure 4.8.

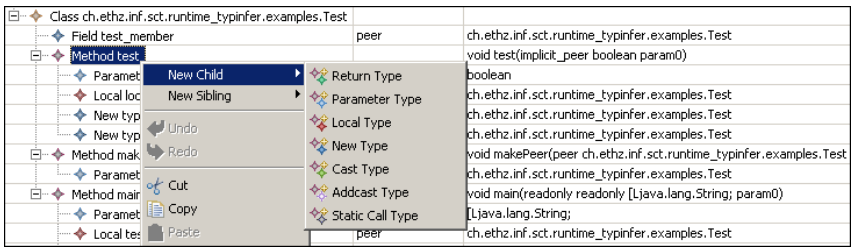


Figure 4.8: Annotations Editor

Chapter 5

Building and Deployment

5.1 Building a Plug-in

A plug-in developed in an Eclipse workspace must be compiled and packed into a JAR file or an own, specially labeled directory in order that it can be installed in any other Eclipse. There are two possibilies to build a plug-in, the first one is a UI-based wizard to export plug-ins provided by the Eclipse plug-in development environment (FILE | EXPORT | *Deployable plug-ins and fragments*). The second possibility is to build the plug-in with an Ant script. Both methods in common use a file named `build.properties` to specify which files belong to the plug-in and must be contained in the plug-in for delivery. The build file distinguishes between a source- and a binary build. For each build type the files to include must be specified. See listing 5.1 for the build file of the JML Plug-in.

Listing 5.1: Buildfile build.properties

```
source.. = src/
output.. = bin/
bin.includes = META-INF/,\
               .,\
               plugin.xml,\
               icons/,\
               about.ini
src.includes = plugin.xml,\
               icons/,\
               src/,\
               META-INF/,\
               about.ini
```

When exporting a plug-in with the Export wizard of the plug-in development environment, there is also the possibility to save the export as an Ant task. Listing 5.2 shows the exported Ant build script for the JML Plug-in (path and delete task added separately). There are a few parameter that can be specified, the most important are `exportType` which determines if the plug-in is exported into a directory or is packed into a zip file and `useJARFormat` which determines if the plug-in is exported as a single JAR file or as directory containing all the class files. A plug-in packed into one single JAR is more comfortable to deliver and handle, but has the disadvantage, that included resources are also packed into the JAR and are not directly accessible from outside.

For that reason the JML Resources and the Runtime Inference Resources plug-in are delivered and installed as directory, while all other plug-ins are packed into single JAR files.

Listing 5.2: Buildfile buildPlugin.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project default="plugin_export" name="JML Plugin">
    <target name="plugin_export">
        <delete dir="${basedir}\dist"/>
        <pde.exportPlugins destination="${basedir}\dist"
            exportSource="false"
            exportType="directory"
            plugins="ch.ethz.inf.sct.jml.plugin"
            useJARFormat="true"/>
    </target>
</project>
```

5.2 Building a Feature

One or more Eclipse plug-ins can be grouped together into an Eclipse feature. A feature can easily be installed and removed in Eclipse as a single unit, in order that a user don't need to care about installing and uninstalling a large set of single plug-ins. Another advantage of building a feature out of a set of plug-ins is that license informations must only be provided once for all plug-ins together.

A feature actual consists of a `feature.xml` file, describing the properties of the feature and several optional files i.e. an image for the about dialog or a HTML file containing separate license information. The `feature.xml` file declares the name, version and provider of the feature. Additionally copyright and license informations are provided. In a second part, all plug-ins contained in the feature are listed together with their download- and install size. And finally an attribute `unpack` specifies whether the JAR file should be unpacked into a directory after installation. This attribute is needed, because a deployable feature contains only JAR files and no ZIP files. Only after installation the plug-ins marked to be unpacked are decompressed in a single folder. Listing 5.3 shows the `feature.xml` file of the JML Plug-in feature. To deploy a feature, all files belonging to the feature (without the plug-in JAR files) are packed into a feature JAR file. The plug-in development environment has a wizard for this task too: FILE | EXPORT | *Deployable features*. Alternatively a feature can also be build with an Ant script. But if features are used together with update sites (see section 5.3), the process of build the feature is needless, because the building process of the update site contains the automatically building of the included features.

Listing 5.3: feature.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<feature id="ch.ethz.inf.sct.jml.plugin" label="JML Plugin"
    version="1.0.0" provider-name="Paolo Bazzi">
<description>Plugin for JML Tools</description>
<copyright> This file is part the Runtime Universe Type Inference Project.
    Copyright (C) 2003-2006 Swiss Federal Institute of Technology Zurich [...]
</copyright>
<license>This library is free software; you can redistribute it and/or modify it under the terms
    of the GNU Lesser General Public License as published by the Free Software Foundation; [...]
</license>
```

```
<plugin id="ch.ethz.inf.sct.jml.plugin" download-size="73" install-size="73"
    version="0.0.0" unpack="false"/>
<plugin id="ch.ethz.inf.sct.jml.plugin.resources" download-size="3845" install-size="5204"
    version="0.0.0"/>
</feature>
```

5.3 Building an Update site

An Eclipse update site is a specially constructed Web site designed to provide one or more features. Like features which groups single plug-ins together, an update site groups different features together. Furthermore the update site helps to manage the delivery, installation and eventual update of the contained features. An update site contains the all the needed plug-ins (packed as JAR files) and features and describes them with a special site manifest file (the `site.xml` file). The Eclipse Update Manager can read this site manifest file and automatically load and install any updates or new products that it finds. Listing 5.4 shows the site.xml file used for the JML- and Runtime Inference plug-in features, declaring which features and versions of them are contained and in which category they should appear on the update site.

Listing 5.4: site.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<site>
    <category-def name="Universe␣Type␣System␣Tools" label="Universe␣Type␣System␣Tools"/>
    <feature url="features/ch.ethz.inf.sct.jml.plugin_1.0.0.jar"
        id="ch.ethz.inf.sct.jml.plugin" version="1.0.0">
        <category name="Universe␣Type␣System␣Tools"/>
    </feature>
    <feature url="features/ch.ethz.inf.sct.runtime_inference.plugin_1.0.0.jar"
        id="ch.ethz.inf.sct.runtime_inference.plugin" version="1.0.0" os="linux,win32">
        <category name="Universe␣Type␣System␣Tools"/>
    </feature>
</site>
```

Figure 5.1 shows an overview of all files contained in the update site:

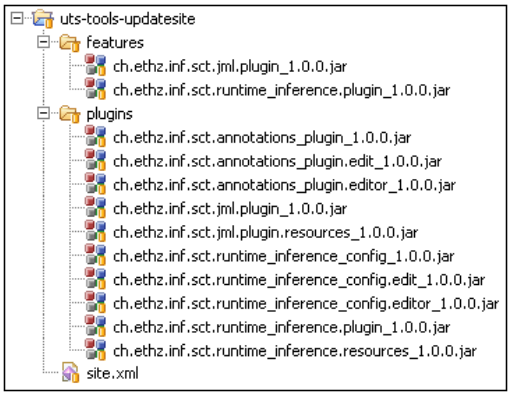


Figure 5.1: Update site files

To build an update site and prepare it for a deployment, Eclipse again offers a wizard. Opening the site.xml file in an update site project, brings up a special editor to easily manage all information. Clicking on the button BUILD ALL builds all plug-ins and features and copy them into the update site project. Everything left to be done to provide the features to interested users is to copy the content of the update site project into any desired folder on a Web server.

To install the features in any Eclipse, first a new remote update site has to be added: HELP | SOFTWARE UPDATES | FIND AND INSTALL | SEARCH FOR NEW FEATURES TO INSTALL | NEW REMOTE SITE. Then by clicking the FINISH button, the newly added update site is searched for new features and the user is prompted to choose the features to install, see figure 5.2.

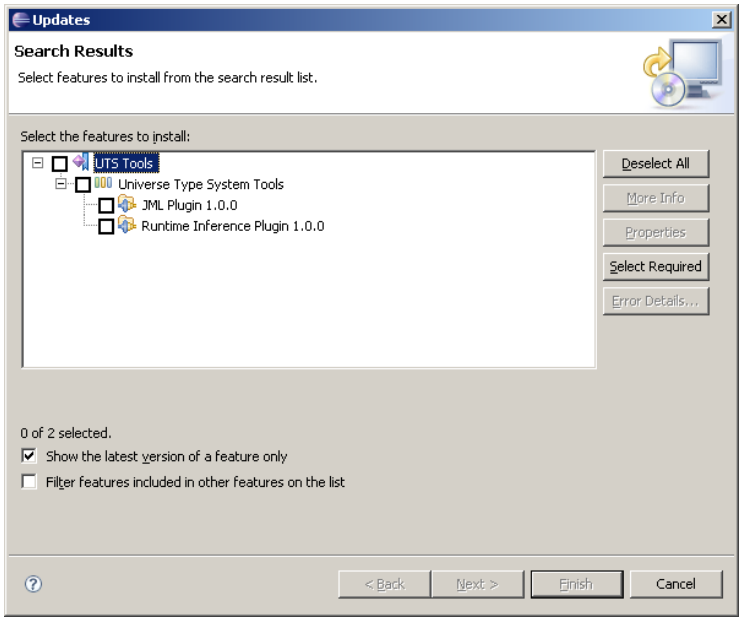


Figure 5.2: Install new Features

After selecting the features and accepting the displayed license the chosen features with their associated plug-ins are automatically downloaded and installed.

5.4 Updating a Plug-in

Using the example of the JML Resources Plug-in in this section I will describe how to correctly update a resources plug-in.

The following steps are necessary:

- Check out latest CVS version of the resources plug-in
- Replace libraries or other files with their new version
- If the file names have changed, rename them in the class `ResourceConstants`
- Open the file `MANIFEST.MF` and increase the plug-in version number
- Use the `build_plugin.xml` to build a new version of the resources plug-in
- Copy the new plug-in file from the `/dist` folder into the Eclipse `/plugin` folder and test it
- Check out the latest CVS version of the feature project containing the resources plug-in to update
- Increase the version number of the feature
- Check out the latest CVS version of the update site project
- Open the file `site.xml`, synchronize the changed feature and rebuild the feature containing the plug-in to update
- Copy the new feature and plug-in files together with the changed `site.xml` file to the Web server providing the update site.

After installing the new feature, the Feature Details dialog (Help | About Eclipse SDK | Feature Details) shows the new version of the installed feature:

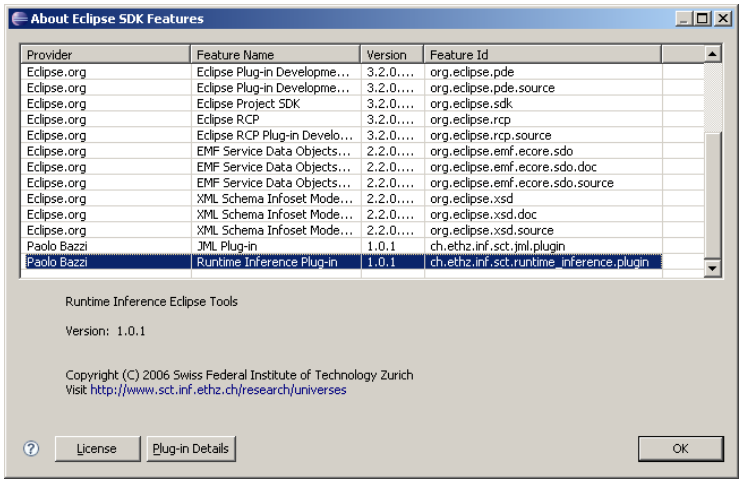


Figure 5.3: Installed Features

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this semester thesis I presented the integration of the JML Tools and the Runtime Inference Tools into the Eclipse development environment. As result both tools are now fully integrated in Eclipse and can easily be used in normal Java projects within Eclipse.

With my JML plug-in a programmer using the Universe type system is now able to check and compile his source code with just a couple of mouse clicks. Using the new JML launch configuration he can easily launch his programs in Eclipse and with the hole support of all other Eclipse tools like browsing the source code or jump into type hierarchies. Properties pages for all important JML Tools settings helps to manage the plenty of attributes which can be varied.

The Eclipse Universe type system programming environment is furthermore extended by the Runtime Inference Plug-in allowing the programmer to automatically analyse his running programs and infer the correct type modifiers. The tracing of running programs is done with a new launch configuration, in order to leave open all existent Eclipse possibilies like declaring special environment properties or provide additional arguments to the program or the Java virtual machine.
Continuing the Runtime Inference execution order a new editor helps to manage the configuration files for the Type Inferer in a graphical way. Finally the inferred annotations can be viewed in a special Annotations viewer and editor.

Finally to be able to use the Eclipse Update Manager to download and install the new plug-ins or eventual updates, an automatic building process for both features and a ready-to-use Eclipse update site were created.

6.2 Future Work

JML Plug-in

There are different major and minor issues that could be extended by future thesis.

- Integrate the two remaining JML Tools (JML unit and JML doc) into the plug-in
- Add a JML project nature with a default JML builder associated to it, and like Java projects using a JML build path to specify which classes to build with the JML compiler.
- Add debug capabilities for JML compiled classes with linked source code (at the moment the generated source code with the JML annotations is only uses temporarily by the JML compiler)

Runtime Inference Plug-in

The Runtime Inference plug-in was like a proof-of-concept for the Eclipse integration of the other Universe type system tools. In future thesis its imaginable, that the whole process of annotating Java programs, check this annotations, change and recheck them will be integrated into Eclipse. In the following enumeration I list a few possible extensions directly related to the Runtime Inference plug-in.

- The configuration editor needs further improvement, for instance editable values in the top list view would increase the usability.
- The annotations editor also could be extended in future, in order to be able to add the inferred annotations directly back into the source code and then reevaluate them.
- Finally the Runtime Inference Project and with the project the according Runtime Inference plug-in could be merged with the Static Inference project[6] into one set of Eclipse plug-ins connecting and sharing their functionality.

Acknowledgment

I would like to thank my supervisor Werner M. Dietl and Prof. Dr. Peter Müller for their assistance.

Bibliography

[1] M. Bär. Practical runtime Universe type inference. Master's thesis, ETH Zurich, 2006.

[2] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212 – 232, June 2005.

[3] Eclipse.org. Eclipse extension points reference. Available from http://help.eclipse.org/help30/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/extension-points/index.html.

[4] Eclipse.org. Eclipse modeling framework. Available from http://www.eclipse.org/emf/.

[5] Eclipse.org. Eclipse platform plug-in developer guide. Available from http://help.eclipse.org/help32/index.jsp.

[6] Nathalie Kelleberger. Static universe type inference. http://www.sct.inf.ethz.ch/projects/student_docs/Nathalie_Kellenberger/, 2005. Master Project.

[7] F. Lyner. Runtime Universe type inference. Master's thesis, ETH Zurich, 2005.

[8] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001. Available from `www.informatik.fernuni-hagen.de/pi5/publications.html`.

[9] MultiJava Team. The multijava project. Available from http://www.multijava.org/.