Technical report

Importing JSP into Moose

David Gurtner
Supervised by: Tudor Gîrba
University of Bern, Switzerland
Software Composition Group

July 13, 2006

Java Server Pages (JSP) is an already established technology for web application development, and thus there is a big need for tools to support reverse engineering of JSP applications. A first step towards the analysis is creating the model by parsing JSP. We have built j2moose as an Eclipse plugin to parse JSP using the Eclipse capabilities. We have validated the approach by extending the Moose reengineering environment to load the exported models from j2moose.

1 Introduction

Java Server Pages (JSP) technology is an XML based language that enables Web developers and designers to rapidly develop dynamic web pages. JSP is part of the Java technology family and allows for integration with and extension through the Java developing language.

Additionally there are several big frameworks further faciliating the development of web applications. Among them the Apache Struts [HDF+02] Model-View-Controller framework and the rather new Java Server Faces technology [Mah04], which allow for easily building web application user interfaces.

Because of these major frameworks, the platform independence and the integration possibility with Java based backend solutions, like the Enterprise Java Bean (EJB) technology, JSP has grown to be one of todays most used web application development technologies.

Beeing widely used in the industry, there is a growing need for tools to analyse and reeingineer JSP applications.

We wanted to build such a tool. The fastest way to build it, was to modify an existing reengineering application to our needs. We decided to add JSP capabilities to the Moose reeingineering environment [NDG05].

Moose is language independent, meaning it supports any language, given that language can be parsed to a format understood by Moose.

Moose did not have support for JSP. For that, we built a parser and we extended FAMIX [TDDN00] with JSP entities. We named the parser j2moose.

J2moose can convert JSP and Java code to one (or more) of the formats, which can be imported into Moose.

2 J2moose

2.1 Concept of j2moose

J2moose is realised as an Eclipse [dRB06] plugin, because of those reasons:

- *Java model in Eclipse*
  Eclipse has features such as a package overview for a project, code completion and code validation. For these features Eclipse has built in models representing Java applications in different ways. All these models are intended to be used by plugins, to integrate with Eclipse, by providing access. It is easy to get meta information on Java code in Eclipse. There's no need to parse Java code directly, but just query Eclipse for information.

- *JSP conversion support through webtools*
  As the goal of j2moose is to not only export Java but also JSP code, there needs to be support for that too. The Eclipse webtools plugin does just that. Similar to how Eclipse has a representation of the Java code, the webtools plugin has a model for JSP. As JSP code can be precompiled to plain Java code, the webtools plugin can be querried for a Java representation of the JSP code, afterwards the Java code can be fed into one of the Eclipse Java models, which can then be used to acquire information.

- *Exporting from where you work on your code*
  Another nice feature of having j2moose as an Eclipse plugin is, that it runs from inside Eclipse. As most JSP applications are developed in Eclipse, the Java and JSP code can be exported from where its created, without the need for some additional application or tool.

2.2 Architecture of j2moose

The j2moose application consists of three main packages (see figure 1):

- *popup.actions*
  This package makes up the Eclipse plugin and contains the classes implementing the neccessary interfaces for the Eclipse plugin. This is where the main classes (the ones starting up j2moose) and the user interface is implemented. The name of this package was generated by the Eclipse plugin builder wizard. More detailed information on this package can be found in Section 3.

- *converters*
  The converters package contains the classes implementing the navigation through the Java project model, find all elements and convert them to a Moose format. More detailed information on this package can be found in Section 4.

- *writers*
  In this package are classes to write data to a file. The subpackage `elements` contains Java beans representing Moose entities to store the data before its written to a file. More detailed information on this package can be found in Section 5.

3 Building an Eclipse Plugin

Getting a running Eclipse plugin is done through providing a few files describing the plugin [AI03].

First we need to define that what we're creating is a plugin. We do this in the manifest file `MANIFEST.MF`:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: J2moose
Bundle-SymbolicName: ch.unibe.iam.scg.j2moose; singleton:=true
Bundle-Version: 0.1.3
Bundle-Activator: ch.unibe.iam.scg.j2moose.J2MoosePlugin
Bundle-Localization: plugin
Require-Bundle: org.eclipse.ui,
 org.eclipse.core.runtime,
 org.eclipse.jdt.core,
[...]
Eclipse-AutoStart: true
Bundle-ClassPath: ch.unibe.iam.scg.j2moose.jar
```

`Bundle-Localization` tells Eclipse that the manifest specifies a plugin, `Bundle-ClassPath` is where all plugin classes are found, `Require-Bundle` specifies library dependencies and `Bundle-Activator` finally is the mainclass of the plugin.

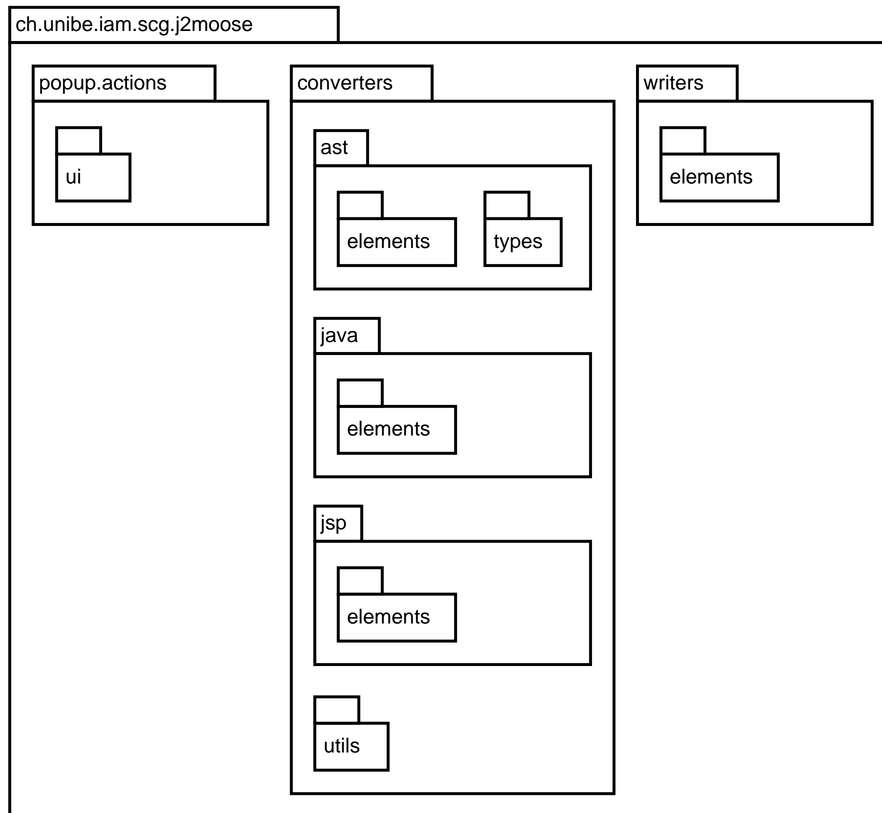

Figure 1: Package overview of j2moose

A `Bundle-Activator` class needs to implement `AbstractUIPlugin`. The one in j2moose looks like this:

```
package ch.unibe.iam.scg.j2moose;
import org.eclipse.ui.plugin.*;
[...]
public class J2MoosePlugin extends AbstractUIPlugin {
    private static J2MoosePlugin plugin;
    public J2MoosePlugin() {
        plugin = this;
    }
    public void start(BundleContext context) throws Exception {
        super.start(context);
    }
    public void stop(BundleContext context) throws Exception {
        super.stop(context);
        plugin = null;
    }
    public static J2MoosePlugin getDefault() {
        return plugin;
```

```
    }
    [...]
}
```

This class is a default implementation, and will look almost the same for all Eclipse plugins.

The next step is to add custom extensions, this is done in the `plugin.xml` file [Ars01]:

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin>
  <extension point="org.eclipse.ui.popupMenus">
    <objectContribution
          adaptable="false"
          id="ch.unibe.iam.scg.j2moose.jspFileContribution"
          nameFilter="*.jsp"
          objectClass="org.eclipse.core.resources.IFile">
      <menu
            id="ch.unibe.iam.scg.j2moose.jspFileMenu"
            label="J2Moose">
        <groupMarker name="ch.unibe.iam.scg.j2moose.jspFile"/>
      </menu>
      <action
            class="ch.unibe.iam.scg.j2moose.popup.actions.
              ResourceJsp2Moose"
            icon="icons/moosejsp.gif"
            id="ch.unibe.iam.scg.j2moose.FileJsp2Moose"
            label="Jsp2Moose"
            menubarPath="ch.unibe.iam.scg.j2moose.jspFileMenu/
              jspFile"/>
    </objectContribution>
     [...]
  </extension>
</plugin>
```

`point=org.eclipse.ui.popupMenus` tells Eclipse to add something to the popup menu of the element defined in `objectContribution`. The action to be executed on this element is defined in `action`. Finally an optional `menu` can be specified, where the extension will be located.

The above example would thus add the menu `J2Moose` to the context menu of all gui elements implementing `IFile`. The `J2Moose` would have an entry `Jsp2Moose`, which would run the `FileJsp2Moose` action.

The actions we specify in the `plugin.xml` need then be defined. In the case of j2moose all actions are of the extension type `popupMenus`, thus they're defined in the `popup.actions` package.

As all actions do practically the same task (they only differ in their output, depending

on the element they are defined on), they are all extending a common class where the common parts are implemented. This class is called `AbstractJ2MooseAction`:

```
package ch.unibe.iam.scg.j2moose.popup.actions;
import org.eclipse.ui.IObjectActionDelegate;
[...]
public abstract class AbstractJ2MooseAction implements
    IObjectActionDelegate {
    protected Shell shell;
    protected J2MooseUI view;
    [...]
    public void run(IAction action) {
        [...]
        this.shell = workbenchWindow.getShell();
        this.view = new J2MooseUI(this, this.shell);
    }
    public void startExporting() {
        [...]
        ProgressMonitorDialog pmd = new ProgressMonitorDialog(
            this.shell);
        if (this.javaElement != null) {
            Java2Moose converter = new Java2Moose(this.javaElement,
                this.createJavaVisitor(writer));
            pmd.run(true, true, converter);
        }
        if (this.resourceElement != null) {
            Jsp2Moose converter = new Jsp2Moose(this.resourceElement,
                this.createJspVisitor(writer));
            pmd.run(true, true, converter);
        }
        [...]
    }
    [...]
}
```

In the `run` method the user interface gets instantiated, which presents the user with a GUI (see figure 2). In the GUI the user can select to start exporting Java and JSP code to a Moose format, which will run the `startExporting` method. Running `startExporting` will create and run a Java and a JSP exporter called `Java2Moose` and `Jsp2Moose` respectively.

4 Accessing the Eclipse Meta Model

When starting up Eclipse and looking at the Java perspective, the user is presented with different views, showing different parts of a Java application. There is the package explorer, with an overview of the projects, their packages and classes, even the the class attributes and methods. Next is the editor, which shows the sourcecode for the currently

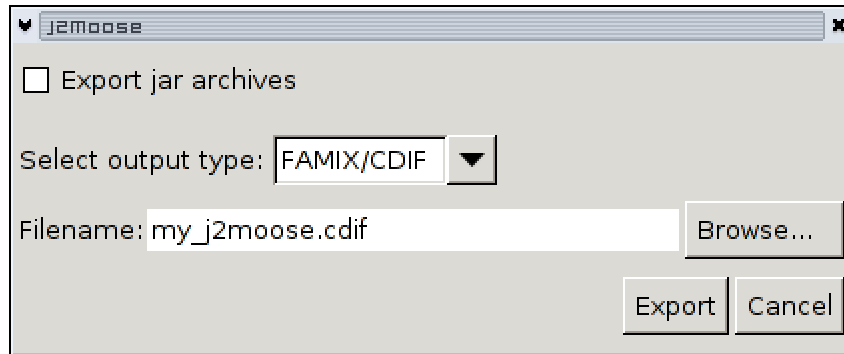

Figure 2: J2moose user interface

opened class. Another view shows an outline of the class displayed in the editor. Finally is a view for different kinds of output, generated by various Eclipse functions.

Now Eclipse does not only have all those different views, but a different internal representation for the data displayed in them. So to efficiently get information out of Eclipse, j2moose needs to access a few of these models, as none really represents the whole of an application.

In the following sections we present a detailed overview of how j2moose accesses the different models.

4.1 The Java Representation

The package explorer shows you the package and class file structure of an application, so that is the place to look for the names and the hierarchical structure of things. Speaking from a Moose viewpoint, that is where the information about Namespaces, Classes , Methods and Attributes is found.

Eclipse has an interface for every Java element displayed in the package explorer. All those interfaces inherit from `IJavaElement`. Furthermore those elements containing others inherit from `IParent`. Lets clarify this with an example: `IJavaProject` represents the root node of a project, it contains elements of type `IPackageFragmentRoot`. So `IJavaProject` would implement `IJavaElement` and `IParent`.

To convert Java elements j2moose first gets the one element on which it was run (and which will be an instance of `IJavaElement`). Then it gets the exact type, asking all possible `IJavaElement` subinterfaces if the element is an instance of them (this is done in `Java2MooseConverterFactory`). Knowing the type of an element, j2moose then is able to convert it. Finally j2moose checks if it the element is an instance of `IParent` and if so, it gets all the children and converts them the same way as the original element. The following bit of code from `Java2Moose` does exactly that:

```
package ch.unibe.iam.scg.j2moose.converters;
```

```java
import org.eclipse.jdt.core.IJavaElement;
[...]
public class Java2Moose implements IRunnableWithProgress {
    [...]
    public void convertJava2Moose(IJavaElement element)
        throws InvocationTargetException, InterruptedException {
        [...]
        Java2MooseConverter converter = Java2MooseConverterFactory.
            createConverter(element);
        converter.convertJava2Moose(this.getConverterVisitor());
        if (IParent.class.isInstance(element)) {
            IParent elementAsParent = (IParent) element;
                [...]
                    IJavaElement[] children = elementAsParent.
                        getChildren();
                    this.convertChildren2Moose(children);
                [...]
        }
    }
    private void convertChildren2Moose(IJavaElement[] children)
        throws InvocationTargetException, InterruptedException {
        for (int i = 0; i < children.length; i++) {
            this.convertJava2Moose(children[i]);
        }
    }
    [...]
}
```

The `IJava2MooseConverterFactory` returns instances of a subclass of `Java2MooseConverter` holding a reference to the original element, and knowing its type. Those subclasses accept a visitor of type `IJava2MooseConverterVisitor` and use double dispatch to convert the original element to a Moose format. The visitor asks the instances of `Java2MooseConverter` subclasses for a typed reference of their element, and querries the element for information and writes it to a file. The writing is discussed in more detail in Section 5: "Writing Data to a Moose format". The visitor allows for the support of multiple formats. Different visitors convert to different formats.

The `IJava2MooseConverterVisitor` and implementations thereof are in the `ch.unibe.iam.scg.j2moose.converters.java` package.`Java2MooseConverter`, `Java2MooseConverterFactory` and elements created by the factory are in the `elements` subpackage as shown in figure 3.

4.2 The Resource Representation

Similar to the Java representation is the resource representation. Where the Java representation shows all Java elements, the resource representation shows all files and folders of a project. There is a relation between them, as every Java element is either a file or

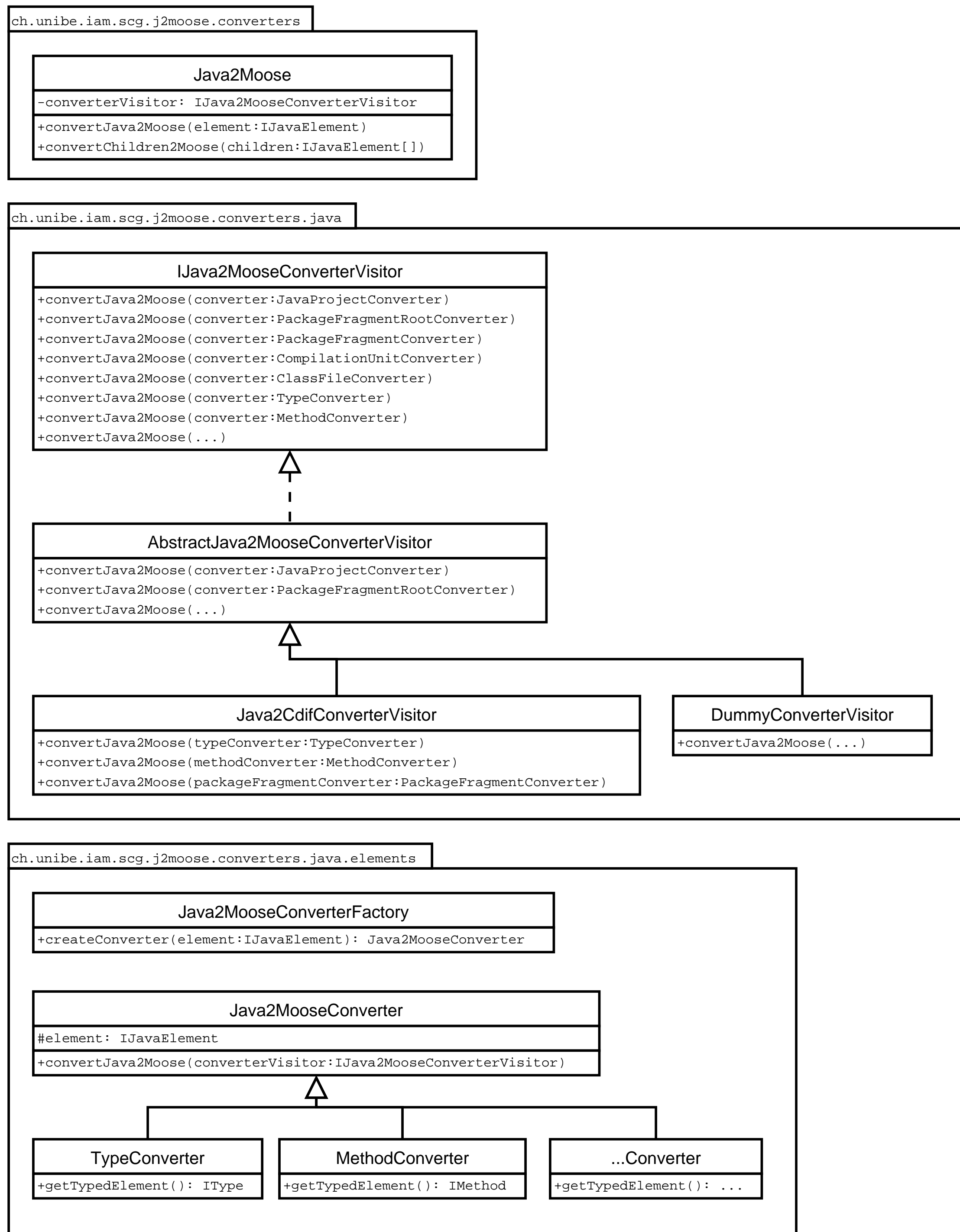

Figure 3: Java converter classes

a folder. But not necessarily the other way round, as a JSP file for example is only in the resource representation. So to find and access JSP files, the resource representation is the correct place.

In the resource representation, every element is an instance of `IResource`. There are only three different kinds of `IResource` elements:

- *IProject*
  Every root node of an Eclipse project is of this type. `IProject` extends the `IContainer` interface.
- *IFolder*
  Represents all folders other than the root node of a project. `IFolder` extends the `IContainer` interface.
- *IFile*
  All files are of this type. `IFile` knows its own file extension.

The `IContainer` interface has a method `members()` returning all resources contained.

Now similar to `Java2Moose` is `Jsp2Moose`. It does exactly the same, just for `IResource` elements:

```
package ch.unibe.iam.scg.j2moose.converters;
import org.eclipse.core.resources.IResource;
[...]
public class Jsp2Moose implements IRunnableWithProgress {
    private IJsp2MooseConverterVisitor converterVisitor;
    [...]
    public void convertJsp2Moose(IResource element)
            throws InvocationTargetException, InterruptedException {
        [...]
        Jsp2MooseConverter converter = Jsp2MooseConverterFactory
                .createConverter(element);
        converter.convertJsp2Moose(this.converterVisitor);
        if (IContainer.class.isInstance(element)) {
            IContainer container = (IContainer) element;
            [...]
                IResource[] members = container.members();
                this.convertMembers2Moose(members);
            [...]
        }
    }
    private void convertMembers2Moose(IResource[] members)
            throws InvocationTargetException, InterruptedException {
        for (int i = 0; i < members.length; i++) {
            this.convertJsp2Moose(members[i]);
        }
    }
```

```
    [...]
}
```

The packages `ch.unibe.iam.scg.j2moose.converters.jsp` and `ch.unibe.iam.scg.j2moose.converters.jsp.elements` are very similar to their Java pendants (`ch.unibe.iam.scg.j2moose.converters.java` and `ch.unibe.iam.scg.j2moose.converters.java.elements`). In the `jsp` package is an interface for visitors called `IJsp2MooseConverterVisitor` and in the `elements` subpackage is a factory called `Jsp2MooseConverterFactory` creating subclasses of the abstract class `Jsp2MooseConverter`. The subclasses of `Jsp2MooseConverter` contain the original element, and if it is a file, folder or project root node. The conversion of files and folders works the same as the conversion of Java elements.

Because `Jsp2Moose` finds all files, not only the JSP ones, visitors have to ensure that files they convert have a `jsp` ending.

4.3 JSP Translation

As the main idea of j2moose is to provide as much insight into JSP programs as possible, it needs to be able to get more information out of JSP pages than is possible with just the mechanisms introduced in Section 4.2 "The Resource Representation". As JSP can be translated to plain Java, j2moose uses this to get additional information.

To translate JSP code to Java, j2moose uses another Eclipse plugin called webtools. The webtools plugin has a class `JSPTranslation` which can hold both the JSP and the Java version. To get an instance of `JSPTranslation`, j2moose has the following implementation in `Jsp2CdifConverterVisitor`:

```
package ch.unibe.iam.scg.j2moose.converters.jsp;
import org.eclipse.jst.jsp.core.internal.java.JSPTranslation;
[...]
public class Jsp2CdifConverterVisitor extends
    AbstractJsp2MooseConverterVisitor {
    public void convertJsp2Moose(FileConverter converter) {
        if (converter.getElement().getFileExtension() != null
                && converter.getElement().getFileExtension().
                    toLowerCase()
                        .equals(Jsp2MooseConverter.JSP_FILE_EXTENSION
                            )) {
            IFile file = (IFile) converter.getElement();
            [...]
            // extract Java Information from the Jsp code
            this.convertJSPContent(file);
        }
    }
    [...]
    public void convertJSPContent(IFile file) {
        IDOMModel model = null;
```

```
        try {
            // get jsp model, get tranlsation
            model = (IDOMModel) StructuredModelManager.
                getModelManager()
                    .getModelForRead(file);
            if (model != null) {
                JSPTranslationAdapterFactory factory = new
                    JSPTranslationAdapterFactory();
                model.getFactoryRegistry().addFactory(factory);
                IDOMDocument xmlDoc = model.getDocument();
                JSPTranslationAdapter translationAdapter = (
                    JSPTranslationAdapter) xmlDoc
                        .getAdapterFor(IJSPTranslation.class);
                JSPTranslation translation = translationAdapter
                        .getJSPTranslation();
                this.convertJSPTranslation(translation);
                translation.release();
            }
        } catch (IOException e) {
            e.printStackTrace();
        } catch (CoreException e) {
            e.printStackTrace();
        } finally {
            if (model != null)
                model.releaseFromRead();
        }
    }
    private void convertJSPTranslation(JSPTranslation translation)
            throws JavaModelException {
        ICompilationUnit javaTranslation = translation.
            getCompilationUnit();
        [...]
    }
    [...]
}
```

In `convertJSPContent(IFile file)` the `JSPTranslation` is created. This makes very close use of the webtools plugin, und might get outdated with future versions of webtools.

As can be seen in `convertJSPTranslation(JSPTranslation translation)` the `JSPTranslation` has a getter method for an `ICompilationUnit` which is one of the subinterfaces of `IJavaElement`.

`ICompilationUnit` is the class representing a Java source file, and is therefor not only present in the `IJavaElement` hierarchy, but also the root node of Eclipse abstract syntax tree, which represents sourcecode inside a file. So `ICompilationUnit` contains additional information.

Lets have a look at how to get to this information.

4.4 The Abstract Syntax Tree

As one of the ideas of j2moose is to find method invocations in JSP code, j2moose needs the abstract syntax tree (AST), as it is the only place where this information can be found.

The abstract syntax tree is as the name implies a tree representation of Java source code. An abstract syntax tree can be built out of an `ICompilationUnit` or an `IClassFile`. One being a representation of a Java source file, the other a representation of a compiled Java class. So j2moose creates an AST out of the `ICompilationUnit` it gets by translation JSP to Java as seen in Section 4.3.

In `ch.unibe.iam.scg.j2moose.converters.ast` is the necessary infrastructure to build an abstract syntax tree and afterwards browse it for information.

As seen in figure 4 the `ASTConverter` is implemented as a Singleton that accepts a visitor, when an instance is requested. It then accepts either an `ICompilationUnit` or an `IClassFile`. `ASTConverter` then builds an AST, and calls a `CompilationUnitASTConverter` with the AST (or its root node respectively) to start traversing the tree. The `CompilationUnitASTConverter` will call the `Jsp2CdifASTConverterVisitor` to convert its `ASTNode` and then call a `TypeASTConverter` for all its children. The `TypeASTConverter` will then convert its `ASTNode` with the `Jsp2CdifASTConverterVisitor` and call a `MethodASTConverter` for all its children. Finally the `MethodASTConverter` will convert its `ASTNode`, again using `Jsp2CdifASTConverterVisitor`.

5 Writing Data to a Moose format

J2moose collects various informations in lots of different places. This information gets processed and converted into a format which can be imported into Moose. To store information and pass it to a writer, j2moose has Java beans storing the same information as Moose FAMIX entities [DTD01]. The Java beans are defined in the package `ch.unibe.iam.scg.j2moose.writers.elements`. The Java bean class `MooseClass` for example has the exact same attributes as a FAMIX class entity.

All converters/visitors in j2moose store all information they gather in such an object, and pass this object to a writer.

At the moment j2moose only supports the FAMIX/CDIF [NTD98] format through `J2CdifWriter`, which is implementing the `IJ2MooseWriter` interface:

```
package ch.unibe.iam.scg.j2moose.writers;
import ch.unibe.iam.scg.j2moose.writers.elements.MooseClass;
[...]
public interface IJ2MooseWriter {
    [...]
    public void writeClass(MooseClass mooseClass);
```

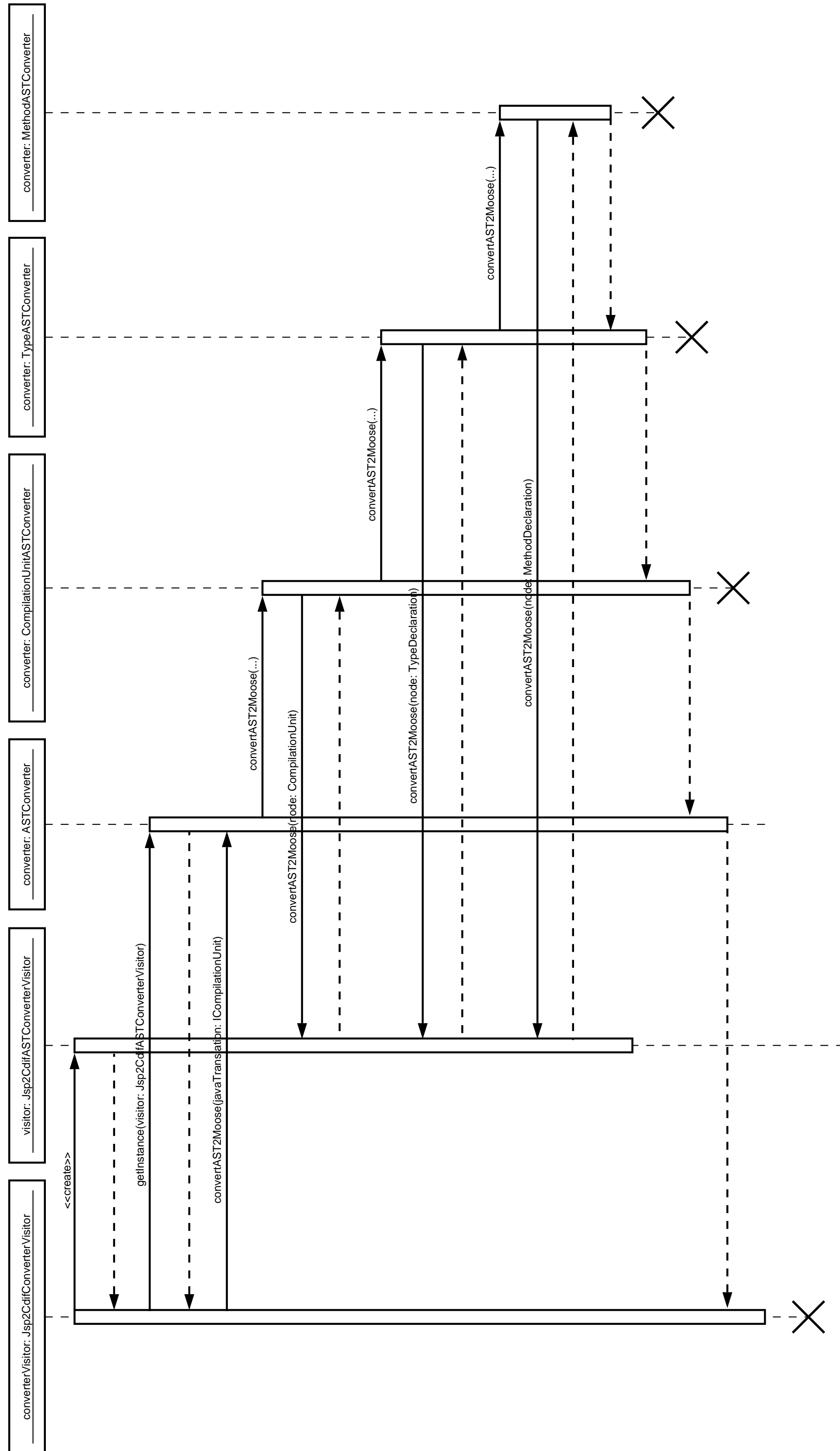

Figure 4: The flow of a sample abstract syntax tree conversion

```
    public void writeMethod(MooseMethod mooseMethod);
    public void writeInheritanceDefinition(
            MooseInheritanceDefinition mooseInheritanceDefinition);
    public void writeNamespace(MooseNamespace mooseNamespace);
    public void writeJSPPage(MooseJspPage mooseJspPage);
    public void writeJSPAttribute(MooseJspAttribute mooseJspAtribute)
        ;
    public void writeJSPInvocation(MooseJspInvocation
        mooseJspInvocation);
    public void close();
}
```

6 Conclusion

To import JSP into Moose various tasks needed to be done: parsing JSP and Java code, writing FAMIX entities to file and adding support for JSP FAMIX entities to Moose.

We built j2moose to parse JSP and to output in the FAMIX/CDIF format. Now we will add support for the new FAMIX/MSE format. We will also enhance the Java support with additional FAMIX entities and metrics.

References

[AI03] Jim Amsden and Andrew Irvine. Your First Plug-in. Technical report, Object Technology International, Inc., 2003.

[Ars01] Simon Arsenault. Contributing Actions to the Eclipse Workbench. Technical report, Object Technology International, Inc., 2001.

[dRB06] Jim des Riviers and Wayne Beaton. Eclipse Platform Technical Overview. Technical report, International Business Machines Corp., 2006.

[DTD01] Serge Demeyer, Sander Tichelaar, and Stéphane Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.

[HDF+02] Ted Husted, Cedric Dumoulin, George Franciscus, David Winterfeldt, and Craig R. McClanahan. *Struts in Action: Building Web Applications with the Leading Java Framework.* Manning Publications Company, 2002.

[Mah04] Qusay H. Mahmoud. Developing Web Applications with JavaServer Faces. Technical report, Sun Microsystems, Inc., 2004.

[NDG05] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software*

Engineering Conference (ESEC/FSE 2005), pages 1–10, New York NY, 2005. ACM Press. Invited paper.

[NTD98] Oscar Nierstrasz, Sander Tichelaar, and Serge Demeyer. CDIF as the interchange format between reengineering tools. In *OOPSLA '98 Workshop on Model Engineering, Methods and Tools Integration with CDIF*, October 1998.

[TDDN00] Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A Meta-model for Language-Independent Refactoring. In *Proceedings of ISPSE '00 (International Symposium on Principles of Software Evolution)*, pages 157–167. IEEE Computer Society Press, 2000.