Inhaltsverzeichnis

Java WebApplications mit Eclipse

Version 1.00

17. Dezember 2003

Diesen Artikel gibt's online auf Stefan Rinke, Articles. Dieser Artikel wurde mit OpenOffice erstellt und anschließend nach DocBook konvertiert. Schließlich wurden mit dem DocBook−Framework die unterscheidlichen Ausgabeformate (siehe Kapitel 10, Download) erzeugt. Das DocBook−Framework selbst ist im Artikel: DocBook−Publishing beschrieben.

Kapitel 1. Einführung

Wer heute in einer J2EE Umgebung WebApplications entwickeln möchte, kann auf sehr leistungsfähige Tools zurückgreifen, die noch nicht einmal Lizenzgebühren kosten.

Mit Eclipse als IDE, Tomcat oder JBOSS als J2EE−Application−Server, CVS als Sourcecode−Verwaltung hat man alles zusammen, was zum professionellen Softwaredevelopment erforderlich ist.

Dieser Artikel soll zeigen, wie eine komplette Umgebung aufgebaut ist und welche Plugins für Eclipse benötigt werden, damit ein effizientes arbeiten möglich wird.

Eine weitergehende Einführung in Java, J2EE, Tomcat, JBOSS oder CVS ist nicht Gegenstand der folgenden Ausführungen. Hier sei im Anhang auf weiterführende Quellen verwiesen.

Kapitel 2. Eclipse: eine modulare Entwicklungsumgebung

Mit Eclipse hat IBM ein echtes Meisterstück abgeliefert. Durch das modulare Design läßt sie sich beliebig erweitern und somit auf unterschiedlichste Bedürfnisse anpassen. Als OpenSource−Produkt erfreut sie sich großer Beliebtheit, was sich vor allem in der grossen Zahl an Erweiterungen niederschlägt. So findet man für alle erdenklichen Sprachen und typischen Entwickleraufgaben Plugins [1], die einem das Leben erleichtern.

Da Eclipse selbst in Java geschrieben ist, spielt es vor allem im Javaumfeld seine Stärken voll aus. Hier sind auch die meisten Plugins verfügbar.

2.1. Installation

Eclipse lässt sich einfach aus dem Archiv auspacken und unter einem beliebigen Pfad installieren (unter Windows z.B. `C:\Programme\Eclipse`) unter Linux (Suse 8.1) unter `/opt/eclipse` . Wichtig für Suse 8.1: die Bibliothek gtk 2.0 muss installiert sein.

Alle Plugins liegen unter dem Installationsverzeichnis im Verzeichnis `plugins`, darunter die einzelnen Packages mit umgekehrten Domainnamen als Verzeichnisse (z.B. `org.eclipse.jdt&`).

Um weitere Plugins zu installieren genügt es, diese entsprechend in die Verzeichnishierarchie zu entpacken und dann Eclipse neu zu starten. Neue Plugins werden dann automatisch erkannt und eingerichtet.

2.2. Grundbegriffe, Aufbau

Der Umgang mit Eclipse gestaltet sich erheblich einfacher, wenn man ein paar grundlegende Begriffe und wie diese in Eclipse verwendet werden verstanden hat.

2.2.1. Workspace

Unter Workspace versteht man einfach alle Resourcen, Dateien, Projekte, die man gerade bearbeitet.

2.2.2. Resourcen

Resourcen sind eigentlich nichts weiter als Dateien, die zusammengenommen ein Projekt ausmachen.

2.2.3. Plugin

Ein Plugin erweitert Eclipse um eine Reihe von Funktionalitäten. Es kann neue Views oder Editoren hinzufügen, sowie neue Perspectives definieren oder einen neuen Wizard integrieren, der hilft eine bestimmte Datei zu erstellen.

2.2.4. Perspective

Eine Perspective fasst die Anordnung aller Views und Editoren auf der Workbench zusammen. So will man z.B. während des Debuggens andere Views benutzen, als beim Entwickeln. Auch die Werkzeugleisten und Menüeinträge werden immer an die aktuelle Perspective angepasst.

2.2.5. View

Ein View liefert eine bestimmte Sicht auf eine oder mehrere Resourcen oder Informationen aus der Umgebung selber. Je nach Anwendung kann so eine View ganz unterschiedlich aussehen. Um anzuzeigen welche Packages und Klassen

im Projekt vorhanden sind benutzt man z.B. den Package−Explorere. Will man Fehler aufspüren, dann ist der Variable−View hilfreich.

2.2.6. Editor

Mit einem Editor kann eine bestimmte Resource (eine Datei) bearbeiten / verändern. Je nach dem was man gerade öffnet , sieht so ein Editor ganz anders aus. Für Java, JSP, HTML oder XML−Dateien verwendet man normalerweise individuelle Editoren, die genau auf den Dateityp zugeschnitten sind.

---

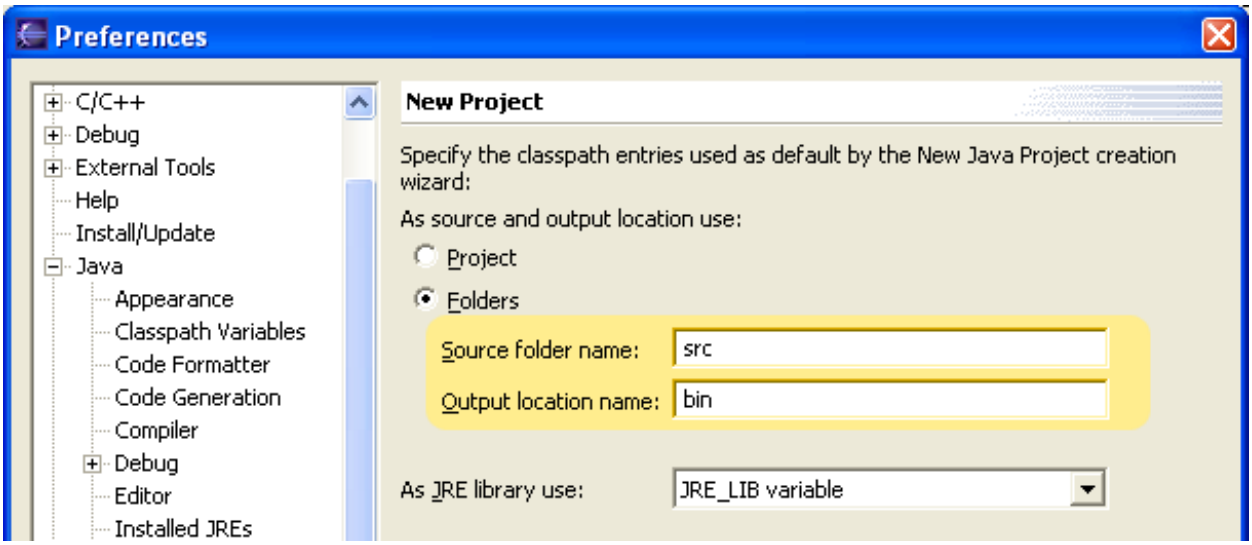
[1] Ein Verzeichnis der vielen Plugins findet man z.B. unter http://eclipse−plugins.2y.net/eclipse/index.jsp oder siehe Abschnitt 11.1.1

Kapitel 3. Javaprojekte mit Eclipse

3.1. Grundeinstellungen

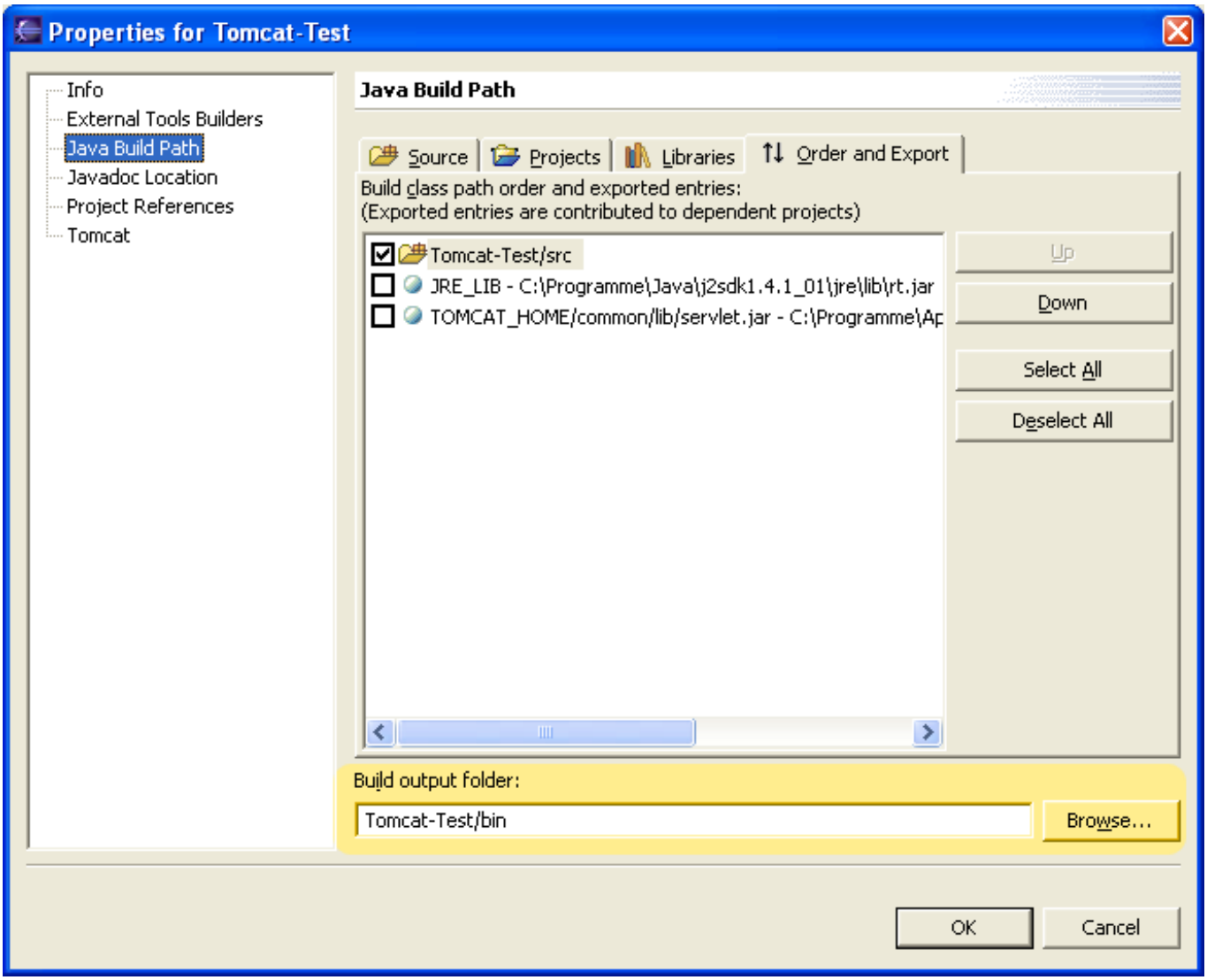
Zunächst einmal sollte man dafür sorgen, dass die Sourcecodes grundsätzlich von den erzeugten Class−Files getrennt bleiben. Hierfür ändert man im Dialog Windows − Preferences den Source−Folder−Name auf src und die Output−Location auf bin .

Abbildung 3.1. Preference New Project



Für Web−Applikationen ist es evtl. Geschickter das Output−Verzeichnis gleich so einzustellen, dass alles Class−Files dort landen, wo der Applicationserver sie erwartet (web/WEB−INF/classes). Das kann man aber auch in der Project−Preferences für jedes Projekt getrennt einstellen.

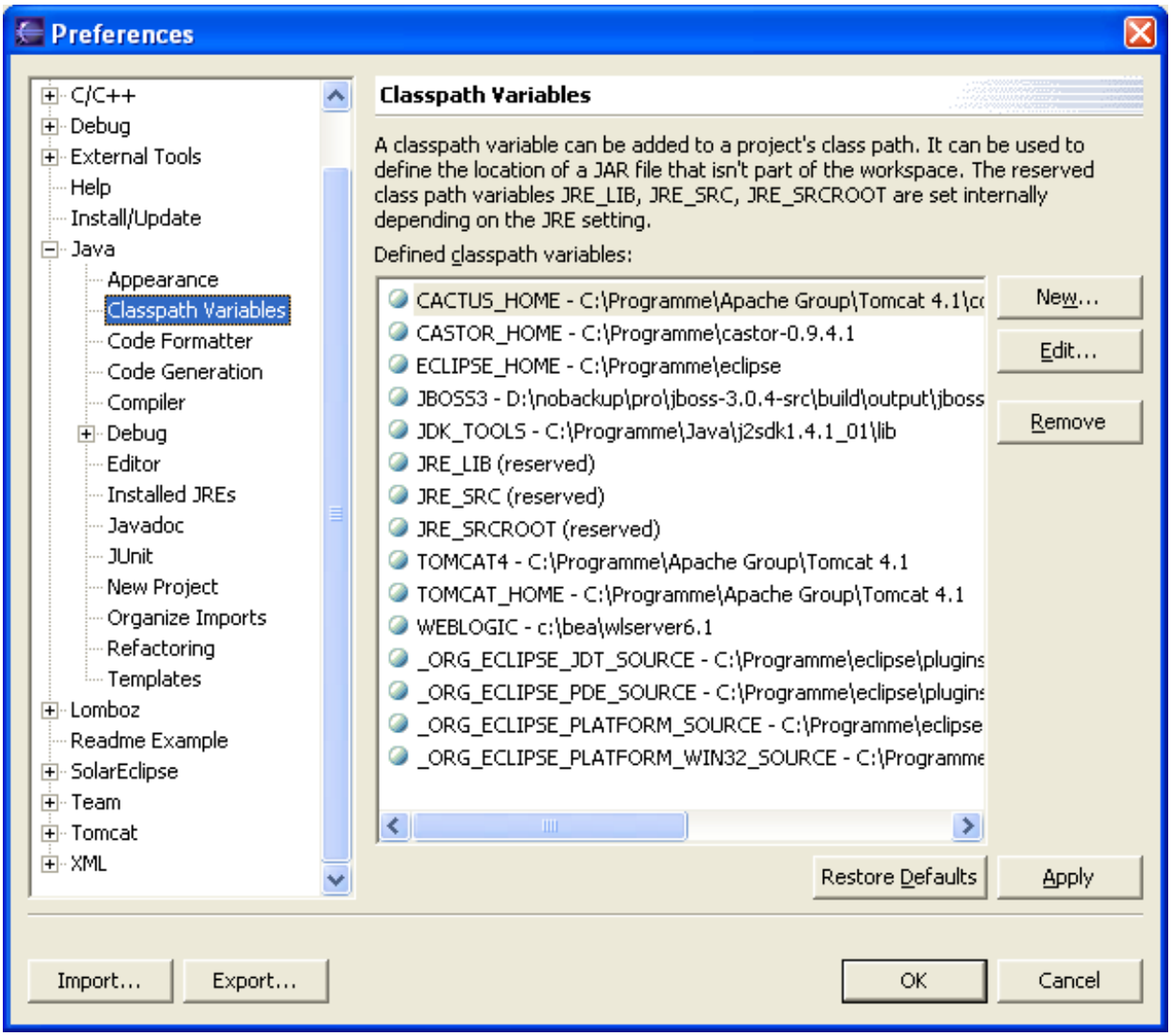
Abbildung 3.2. Properties Build output folder



3.1.1. Classpath−Variablen

Ausser der Laufzeit−Bibliothek wird man bei grösseren Projekten sehr schnell andere Bibliotheken mit einbinden müssen. Da diese bei jedem Entwickler auf unterschiedlichen Pfaden installiert sein können, benutzt man sogenannte Classpath−Variablen.

Abbildung 3.3. Preferences Classpath Variables



Im Projekt werden so nur symbolische Namen gespeichert, die jeder Entwickler auf seine lokale Installation anpassen kann. Braucht man z.B. servlet.jar von Tomcat, dann wird sie im Projekt als TOMCAT_HOME/common/lib/servlet.jar referenziert. Wo Tomcat dann genau installiert ist, wird vom Entwickler selbst festgelegt in dem er den Wert von TOMCAT_HOME entsprechend seiner lokalen Installation einstellt.

3.1.2. Versionierte Namen

Wenn man in einem Team entwickelt und relativ neue Technologien im Einsatz hat, dann ändern sich Versionen der zugehörigen Bibliotheken sehr schnell. Damit nicht das heillose Chaos ausbricht und seltsame Effekte auftreten, die auch noch von einem zum anderen Teammitglied unterschiedlich ausfallen, ist es sehr wichtig, dass alle immer die gleichen Versionen alle Komponenten verwenden.

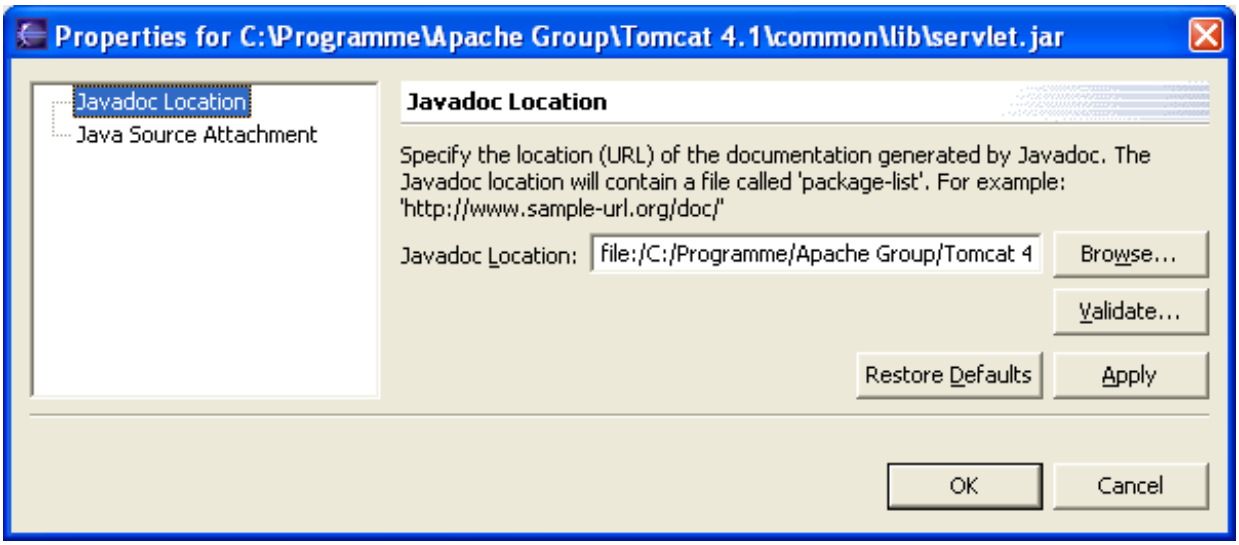
Deshalb empfehle immer Namen für Bibliotheken zu verwenden, die schon im Namen die Versionsnummer enthalten. z.B. castor−0.4.1.jar . Wenn alle Bibliotheken auch so im Projekt referenziert werden, ist sichergestellt, dass alle das Gleiche verwenden.

3.1.3. Erweitern der Dokumentation

Man kann die eingebaute Dokumentation sehr leicht erweitern mit eigenen Texten erweitern. Zum einen lassen sich alle mit JavaDoc erzeugten API−Dokumentationen direkt einbinden. Das gilt zu allererst natürlich für das Java−SDK selbst, aber dann natürlich auch für alle im Projekt benutzten Bibliotheken.

Zum anderen können Plugins ihre Dokumentation nahtlos integrieren, als Beispiel gibt's im Download−Bereich diesen Artikel auf als Plugin, wo durch er eben in der Eclipse Hilfe erscheint.

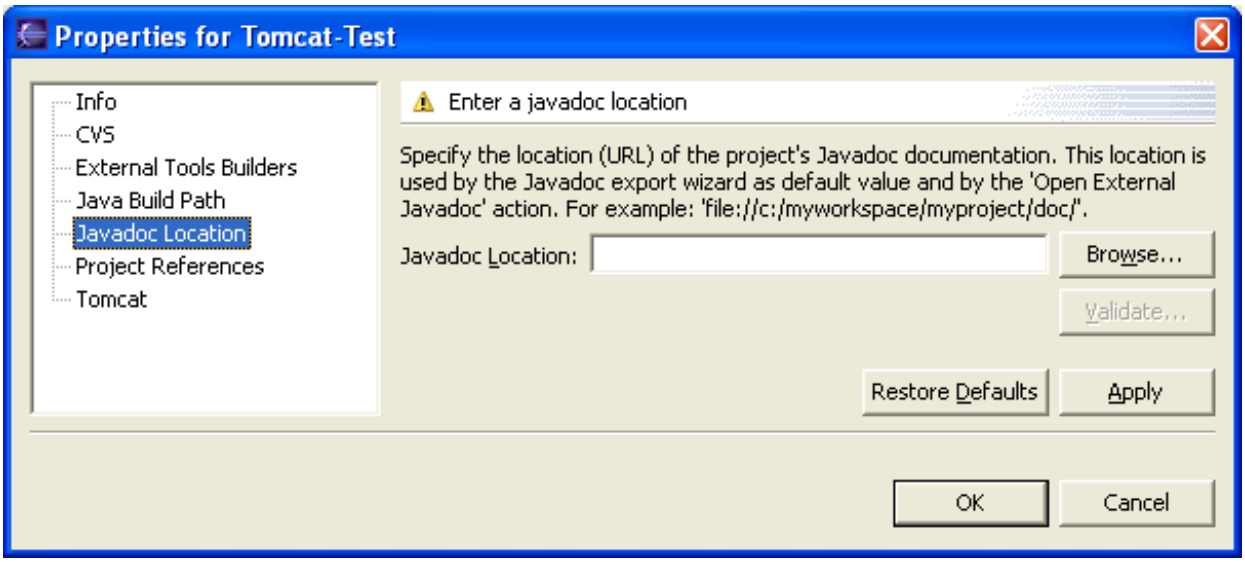
Abbildung 3.4. Properties für eine Bibliothek



Um die Möglichkeiten voll auszunutzen, sollte man zu jeder verwendeten Bibliothek (JAR−File) die entsprechende API−Dokumentation, sowie den Sourcecode (soweit verfügbar) zuordnen. Wenn man ein komplettes JDK installiert inklusive Source erkennt Eclipse das automatisch. Für alle individuellen Bibliotheken ordnet man über die Properties eines jeden JARs (erreichbar über das Kontextmenü) die Dokumentation und das Source−Archiv zu.

Entsprechend wird auch die API−Dokumentation des eigenen Projekts mit eingebunden:

Abbildung 3.5. Properties eigene Dokumentation einbinden

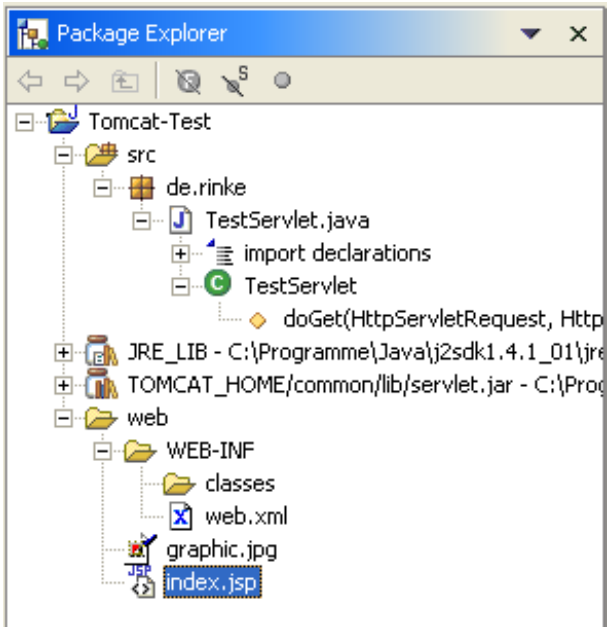


So vorbereitet liefert den Druck auf Shift−F2 jederzeit die Dokumentation zum Objekt unter dem Cursor, genauso wie F3 das betreffende Sourcefile öffnet.

3.1.4. Beispielprojekt für Tomcat

Dies Beispiel soll zeigen, wie man zunächst noch ohne spezielle Unterstützung von Plugins ein Projekt mit Tomcat aufbaut. Der Package−Explorer zeigt folgende Struktur:

Abbildung 3.6. Tomcat Projektstruktur



Neben dem `src` Verzeichnis gibt es bei WebApplications immer das Verzeichnis `web` und darin das `WEB−INF`, das alle Metainformationen und die zu erzeugenden Class−Files enthält. Die Output−Build−Location (siehe oben) wurde so umgestellt, dass die erzeugten Class−Files direkt im `classes` landen.

Daneben sind alle normalen Web−Resourcen wie HTML−Dateien, Grafiken usw. in `web` untergebracht. Auch die JSP−Seiten werden dort angelegt.

Grundsätzlich reicht das schon aus, um eine erste Anwendung mit Tomcat zu erstellen. Das Deployment ist allerdings noch nicht sehr komfortabel: man muss alle Dateien des Workspaces unter `web` in ein Verzeichnis unter das WebApps−Verzeichnis von Tomcat kopieren und anschließend Tomcat neu starten. Das geht entweder ganz manuell mit einem Script oder per Export mit Eclipse im File−Menü.

Während der Entwicklungsphase kann man sich das Deployment aber auch schenken und stattdessen einen Tomcat−Context erzeugen, der direkt auf die WebApplication im Workspace von Eclipse verweist. Hierzu wird ein Eintrag in der `server.xml` in `TOMCAT_HOME/conf` von Tomcat erstellt:

```
<Context reloadable = "true" displayName="Develop"
docBase="C:\Programme\eclipse\workspace\Test−Servlet\web" cookies="true"
path="/test"/>
```

Diesen Eintrag fügt man hinter die anderen Kontexte ein und startet Tomcat neu.

Was dann aber immer noch nicht geht, ist das Debuggen der Application. Diese speziellen Aspekte werden in den Kapiteln 7 und 8 erläutert.

Kapitel 4. Eclipse und CVS

Spätestens wenn man nicht mehr allein an einem Projekt arbeitet, muss der Zugriff auf die Sourcecodes richtig verwaltet werden. Ansonsten entsteht schnell ein grosses Durcheinander. Aber auch schon bei dem One−Man−Project (oder One−Woman) ergeben sich Vorteile, wenn man das Concurrent Version System (CVS) mit benutzt.

Das CVS verwaltet alle zum Projekt gehörenden Resourcen (also Dateien) und führt zu jeder einzelnen Datei eine Änderungshistorie. Alle Projekte werden an zentraler Stelle gehalten, dem Repository.

Die Unterstützung für CVS ist direkt in Eclipse integriert, man kann also sofort loslegen. Natürlich muss irgendwo ein CVS−Server zur Verfügung stehen. Zum Testen oder als lokales Repository kann man mit CVSNT (für Windows) oder CVS schnell einen CVS−Server einrichten.

Neben der Möglichkeit im Team an einem Projekt zu arbeiten, sind die wichtigsten Vorteile:

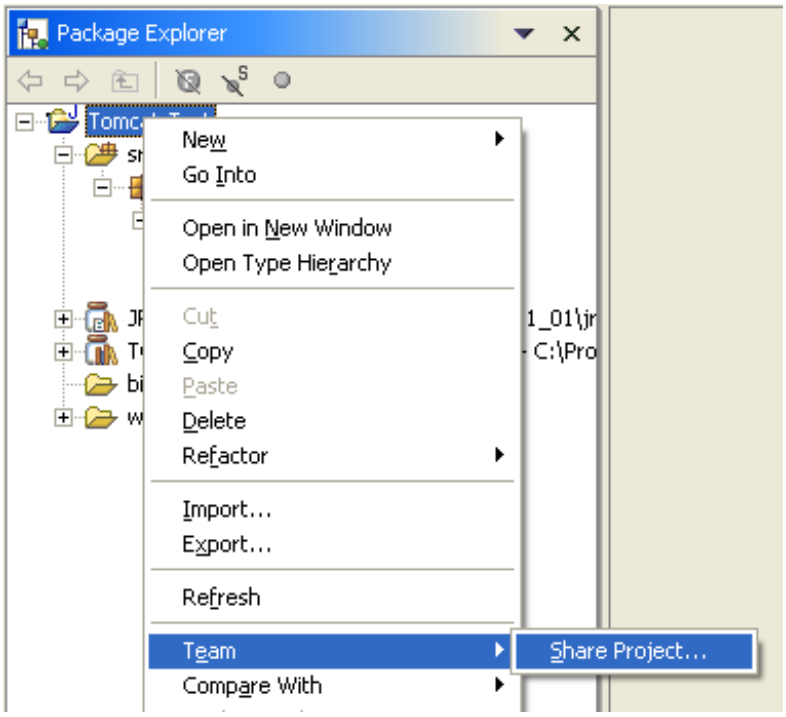
- Änderungshistorie zu jeder Datei
- Änderungskommentare, die die Motivation hinter jeder Änderung erläutern
- Übersichtliche Änderungsanzeige (was hat sich getan?)
- Verwalten von Tags (Versionsständen oder Releases)
- Verwalten von unterschiedlichen Entwicklungszweigen (Branches)

Unabhängig, ob im Team oder nicht gearbeitet wird, sollte man allein aus diesen Gründen immer mit CVS arbeiten.

4.1. Bestehendes Projekt einbinden

Im Context−Menü des Projekts unter Team Share Project auswählen:

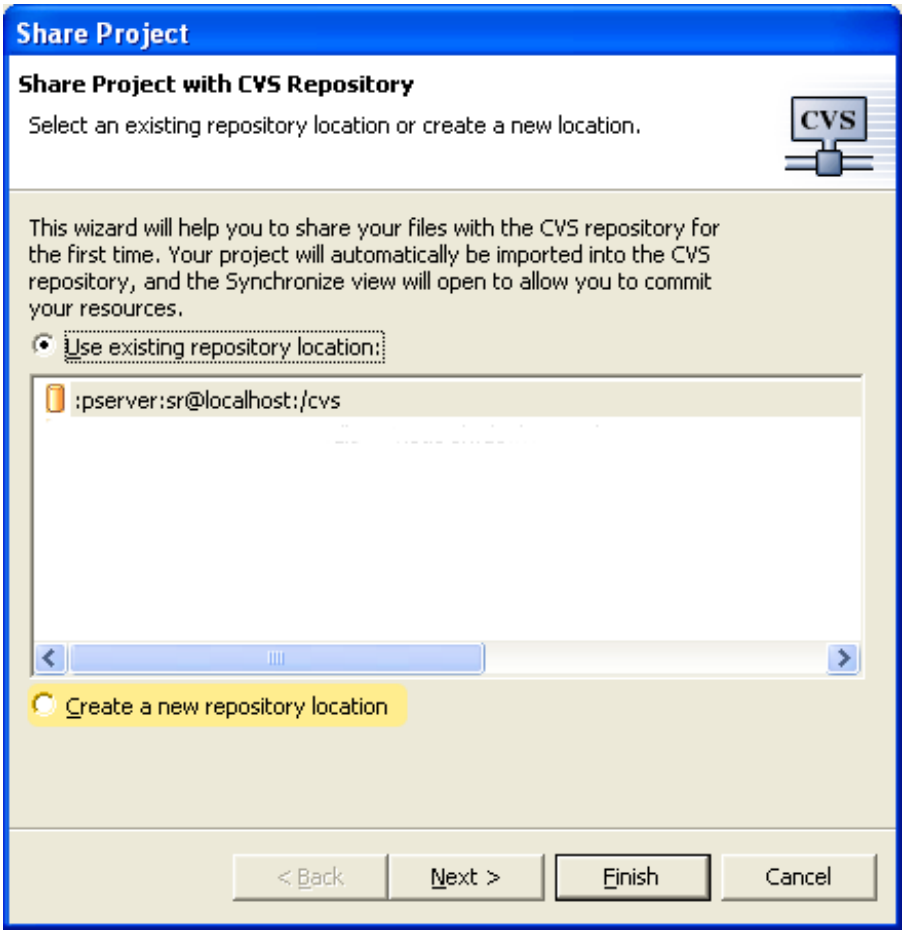
Abbildung 4.1. Projekt ins CVS einbinden



Wer zuvor noch nie mit CVS gearbeitet hat, muss nun zuerst eine Repository−Location anlegen, damit also bestimmen, welchen CVS−Server man verwenden möchte. Wenn schon mal mit CVS gearbeitet wurde, dann werden die bereits bekannten CVS−Server aufgelistet und man kann einfach einen auswählen.

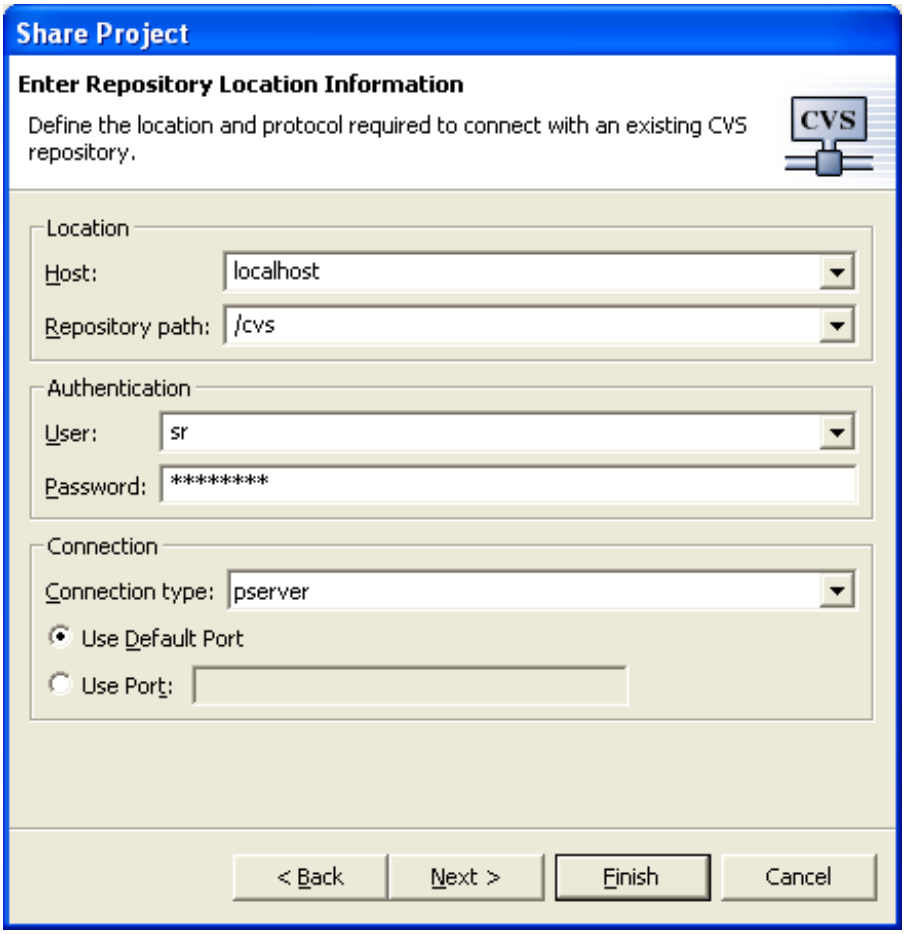
> **Anmerkung: Anmerkung**
> Das Löschen einer Repository−Location geht nur in der CVS−Perspective im CVS−Locations−View: im Kontextmenü auf Discard−Location klicken.

Abbildung 4.2. Repository wählen



Das Erstellen einer Repository−Location fragt nun eine Reihe von Parametern ab:

Abbildung 4.3. Repository−Location neu erstellen

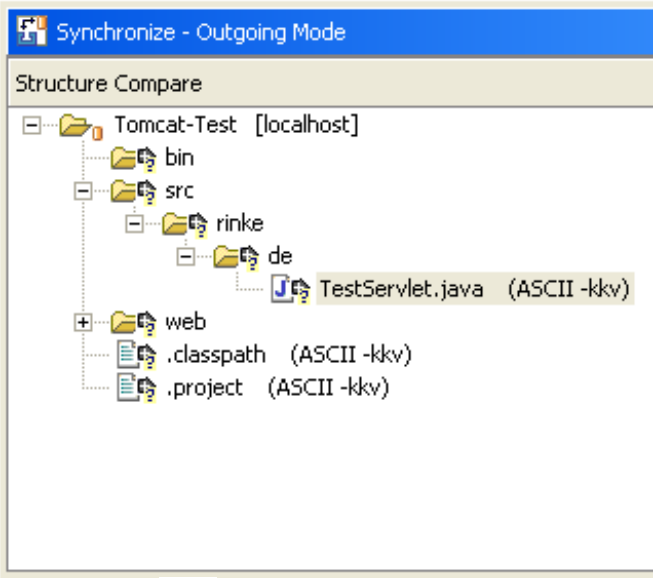


Host und Repository path definieren, wo das Projekt zu finden ist bzw. wo es gespeichert werden soll. Authentication legt die Informationen für das Login zum CVS−Server fest. Connection spezifiziert die Art der Verbindung (normalerweise im pserver).

Dann bleibt nur noch auszuwählen, ob der Projektnamen auch als CVS Modulnamen verwendet werden soll. CVS verwaltet alles in Modulen, was im Repository einem Verzeichnis auf oberer Ebene entspricht. Normalerweise wählt man Projektnamen und Modulnamen identisch, dann kann keine Verwechslung passieren.

Anschließend wird das Projekt im Synchronize−View angezeigt:

Abbildung 4.4. Neues Projekt im Synchronize−View



Die Symbole bedeuten, dass die lokalen Verzeichnisse und Dateien im Repository noch nicht bekannt sind und daher mit Add to Version−Control und anschließendem Commit endgültig unter die Verwaltung des CVS gestellt werden.

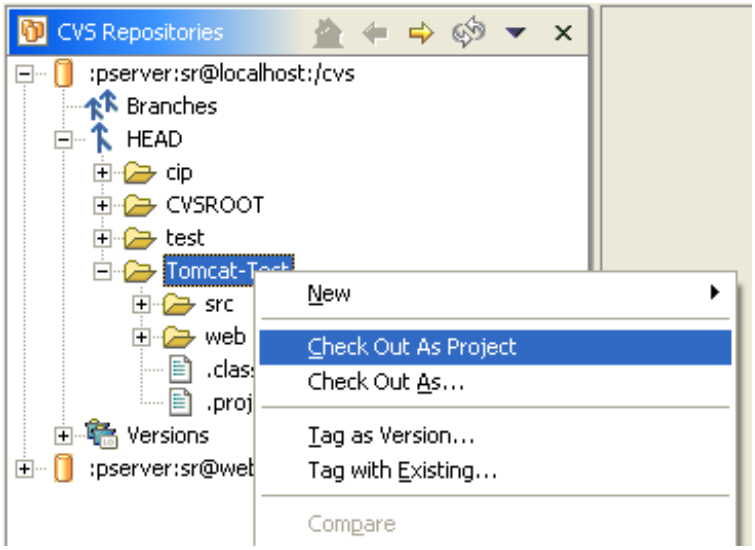
Für das ganze Projekt geschieht dies im Kontextmenü des Projekts. Beim Commit wird abschließend noch ein Dialog angezeigt, der jedem Übertragen ins Repository das Eintragen eines Kommentars erlaubt.

> **Anmerkung: Anmerkung**
> Die Klammern neben den Dateinnamen (ASCII −kkv) oder auch (Binary) kennzeichnen die Dateiart, wie sie vom CVS gesehen werden. Im ASCII−Modus können Änderungen sehr komfortabel angezeigt und verwaltet werden. Die Option −kkv bewirkt, dass bestimmte Platzhalter in den Dateien vom CVS automatisch ersetzt werden. So kann z.B. die Versionsnummer immer aktuell in den Text einer Ausgabe eingebaut werden.

In den beiden Dateien .project und .classpath verwaltet Eclipse die Metainformationen zu jedem Projekt. Es ist daher wichtig, dass diese Dateien immer mit eingecheckt werden. Wenn man sich an die Grundeinstellungen von Kapitel 3.1.1 hält und nur mit Classpath−Variabeln arbeitet und absolute Bezüge ganz vermeidet, dann kann jeder im Team das Projekt aus dem CVS auschecken und sofort damit arbeiten.

In Eclipse geht das im Repository−View besonders komfortabel mit Checkout as Project was bei jedem Verzeichnis im Kontextmenü angeboten wird.

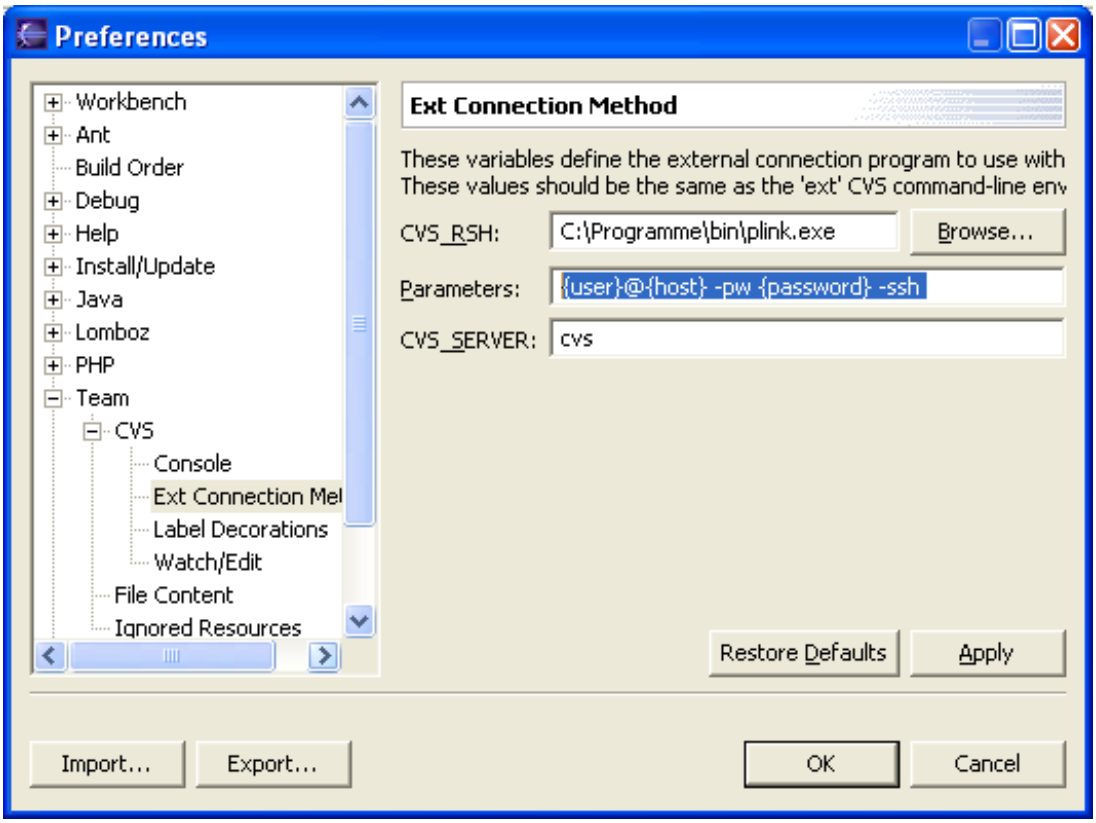
Abbildung 4.5. Projekte auschecken



4.2. CVS mit SSH

Kann mit auf das Repository nur per SSH zugreifen, so muss man, besonders unter Windows, einen SSH−Client zusätzlich installieren, der die Verbindung herstellt. Für Windows gibt es seit langem den freien Client Putty, der mehrere Serviceprogramme[2] mitbringt. Was für den Remotezugriff per SSH gebraucht wird, ist `plink.exe`. Unter Windows/Preferences/Team/CVS/Ext Connection Method wird es konfiguriert:

Abbildung 4.6. Ext Connection konfigurieren



Als Parameter muss eingetragen werden {user}@{host} −pw {password} −ssh . Damit lassen sich sichere Verbindungen zum Repository aufbauen, so dass man auch übers Internet bequem arbeiten kann.

[2] u.a. zur Schlüsselgenerierung, zum Kopieren per SSH

Kapitel 5. Eclipse und XML

WebApplications zu erstellen erfordern immer auch einen guten XML−Editor. Zwar kann man einen Teil der benötigten XML−Files für die WebApplication Entwicklung auch generieren, aber trotzdem ist ein manuelles Bearbeiten früher oder später unumgänglich.

Ein sehr gutes XML−Plugin für Eclipse ist XML−Buddy (aktuell ist die Version 0.2.8).

5.1. XML−Buddy installieren

Wie jedes andere Plugin wird das Zip−File in die Verzeichnishierarchie von Eclipse entpackt.

Achtung: Die `plugin.xml` Datei im Verzeichnis `ECLIPSE_HOME/plugins/com.objfac.xmleditor_0.2.8` enthält eine Importanweisung für das PDE−Plugin[3] von Eclipse (was man aber eigentlich nur als Entwickler von Plugins braucht). Wenn das PDE nicht installiert ist, startet XML−Buddy deshalb nicht. Man kann die Importanweisung aber gefahrlos auskommentieren.

```
   ...
 <requires>
     <import plugin="org.eclipse.ui"/>
     <import plugin="org.eclipse.core.resources"/>
     <import plugin="org.eclipse.core.runtime"/>
     <import plugin="org.eclipse.swt"/>
     <import plugin="org.apache.xerces"/>
     <import plugin="org.eclipse.jdt.ui"/>
     <import plugin="org.eclipse.search"/>
     <!-- sr <import plugin="org.eclipse.pde"/> -->(1)
 </requires>
 ...
```

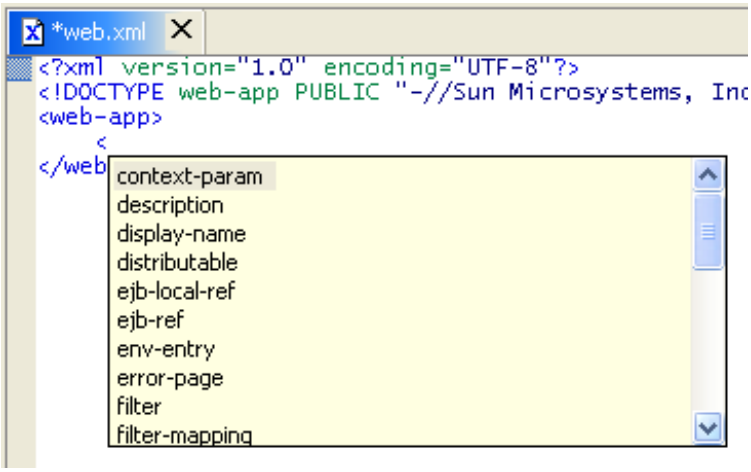
(1) Diese Zeile wird wie im Ausschnitt der `plugin.xml` auskommentiert.

5.2. Gültiges XML

XML−Buddy kann mit Hilfe einer DTD oder eines Schemas die editierten XML−Dateien auf ihre Gültigkeit überprüfen. Er unterstützt einen Cache für schon verwendete DTDs, so dass diese nicht immer aus dem Internet herunter geladen werden müssen.

5.3. Automatische Vervollständigung

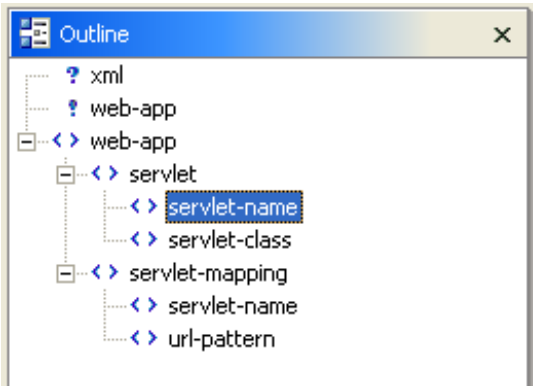
Abbildung 5.1. XML−Buddy Vervollständigung von Child−Elementen



Die DTD bzw. das Schema definiert die Struktur eines XML−Dokuments, was XML−Buddy dazu nutzt beim Editieren automatisch Vorschläge anzuzeigen wie der Code komplettiert werden könnte.

5.4. Strukturanzeige von XML−Dokumenten

Abbildung 5.2. XML−Buddy Strukturanzeige



Als Navigationshilfe liefert XML−Buddy einen Outline−View , der die Struktur des XML−Dokuments mit einer Baumansicht darstellt.

---

[3] PlugIn Development Environment

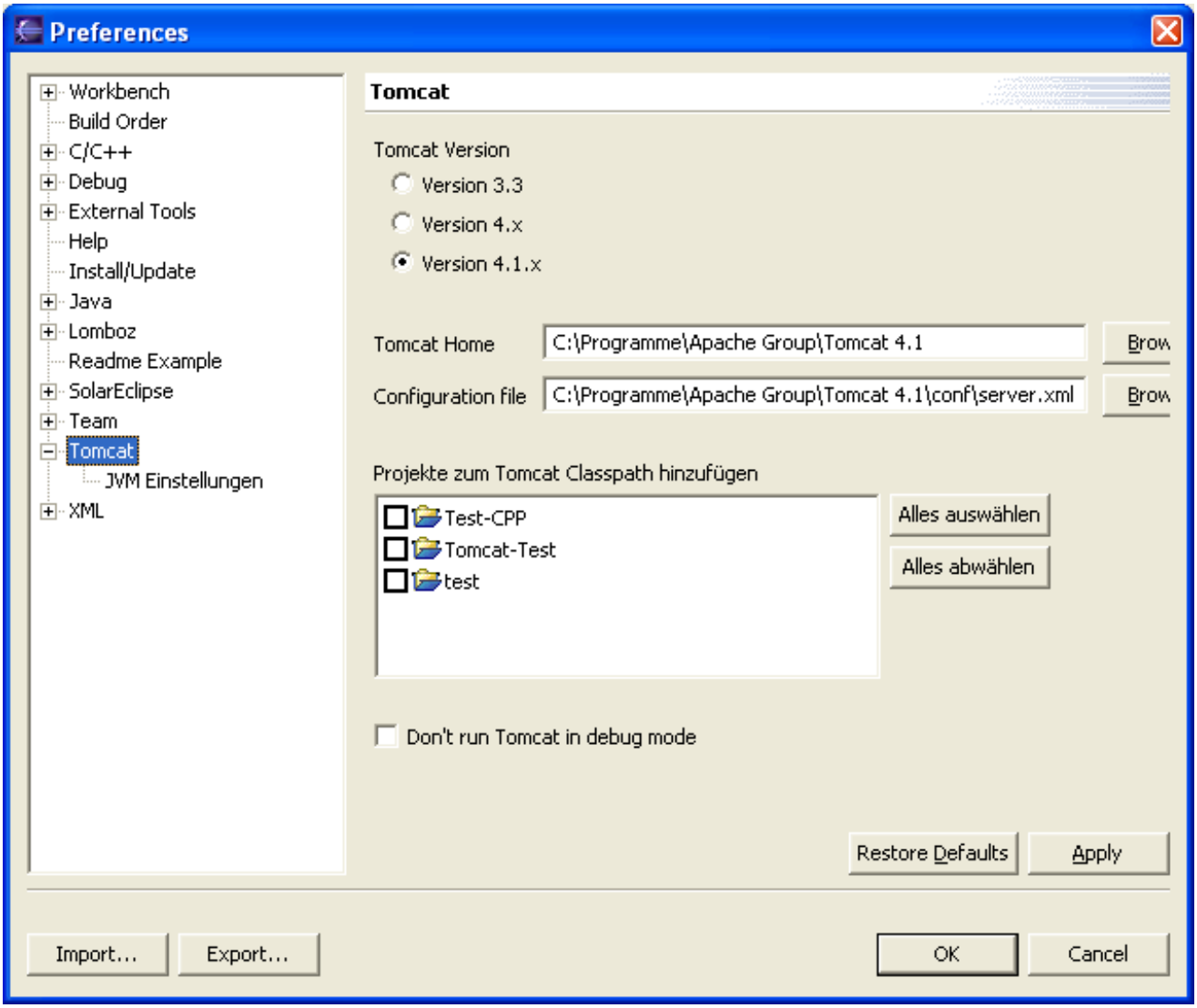
Kapitel 6. Eclipse und J2EE

Um WebApplications mit Eclipse zu entwickeln sind mehrere Plugins erhältlich. Das Tomcat−Plugin von Sysdeo ist für etwas einfachere Projekte mit Tomcat als Applicationserver schon sehr gut geeignet. Wesentlich mehr Funktionalitäten liefert aber das Lomboz−Plugin von ObjectLearn.

6.1. Tomcat−Plugin von Sysdeo

Zunächst wird das ZIP−Archiv in die Verzeichnishierarchie von Eclipse entpackt. Nach dem Neustart von Eclipse läßt sich das Plugin unter Windows / Preferences konfigurieren:

Abbildung 6.1. Tomcat−Plugin Konfigurieren

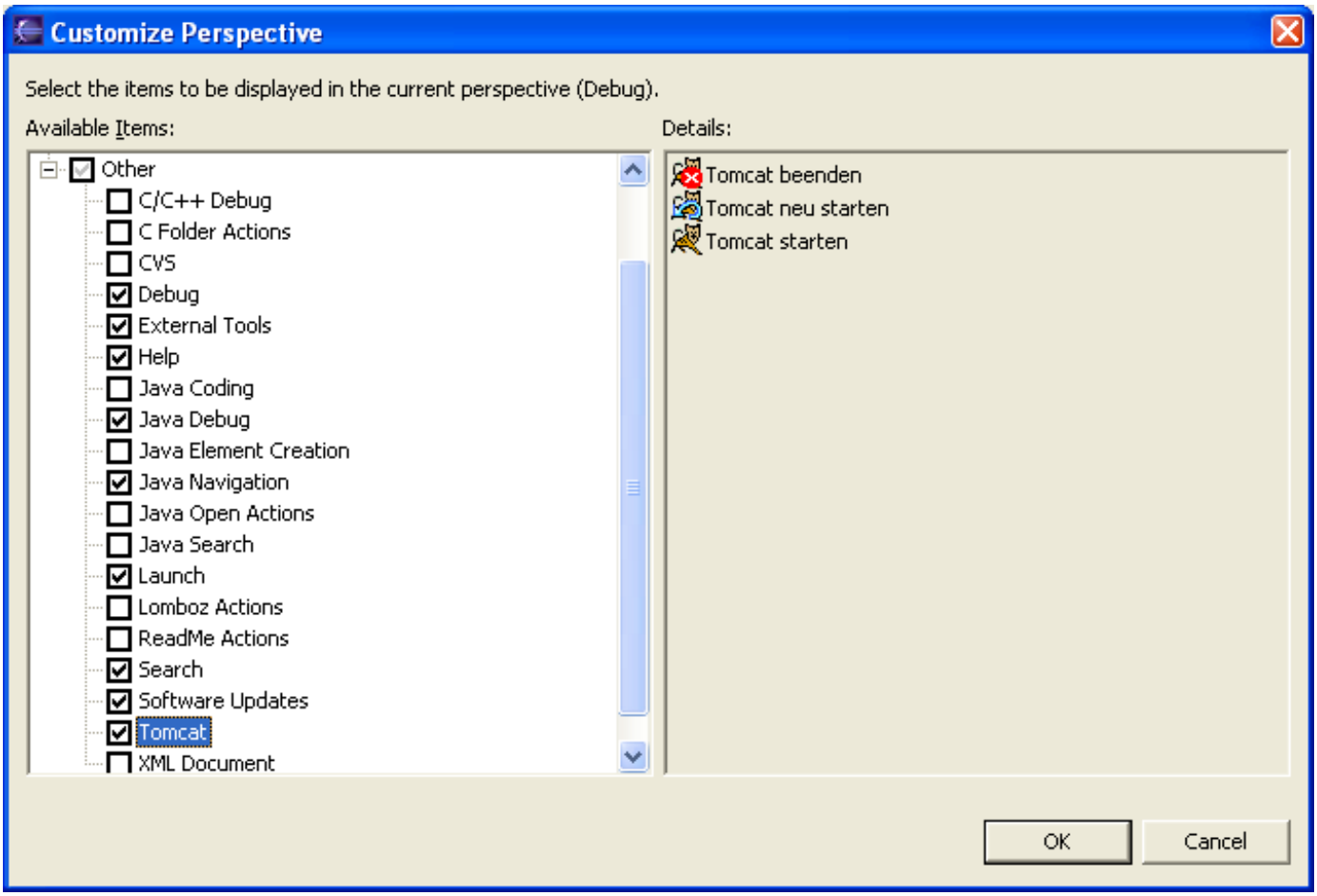


Die wichtigsten Einstellungen sind:

- Tomcat−Version
- Tomcat−Home−Verzeichnis
- Tomcat−Konfigurationsdatei

Damit kann das Plugin bereits benutzt werden. Unter Windows / Customize Perspective aktiviert man nun in der Debug−Perspective das Tomcat−Plugin:

Abbildung 6.2. Tomcat−Plugin aktivieren



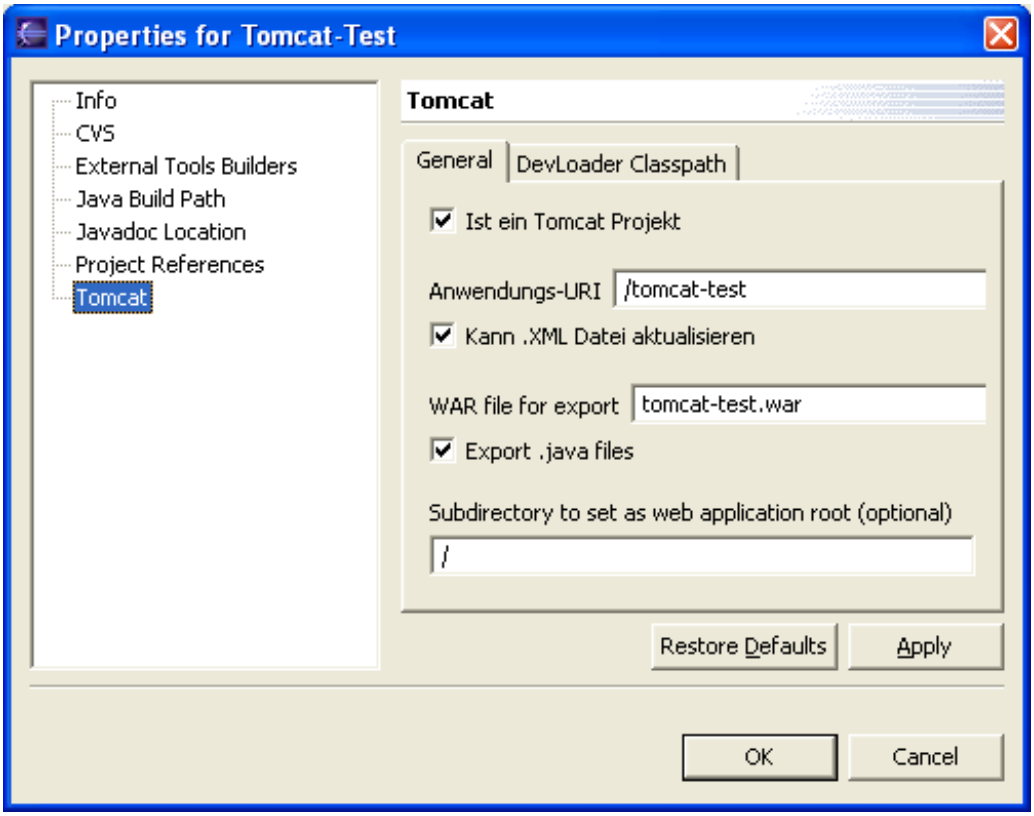
Nun erscheinen in der Toolbar drei neue Buttons  über die man Tomcat direkt unter Eclipse starten und stoppen kann.

Das interessante daran ist, dass man so eine komplette WebApplication inklusive Tomcat debuggen kann. Man kann also ohne weiteres in die Methoden des TestServlets einen Breakpoint setzen und wenn man nun mit einem Browser das Servlet aufruft, bleibt der komplette Request genau in der Servlet−Methode stehen.

6.1.1. Projekt als Tomcat−Projekt

Wenn man ein Project neu anlegt ist fortan Tomcat−Projekt neben Java−Projekt als neuer Projekttyp verwendbar. Ein bestehendes Projekt läßt unter Projekt−Properties zum Tomcat−Projekt machen:

Abbildung 6.3. Tomcat−Projekt Einstellungen

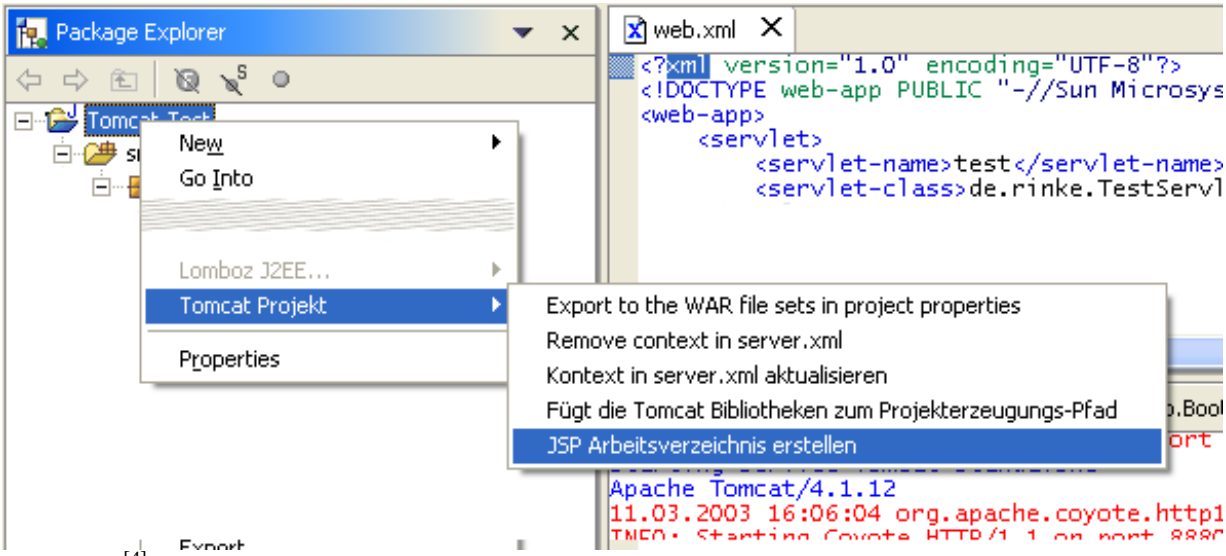


- Zunächst muss Ist ein Tomcat Project angekreuzt sein.

- Bei Anwendungs URI wird der Context−Pfad eingetragen mit dem der Tomcat−Context an gelegt werden soll.
- Wenn Kann .XML aktualisieren angekreuzt ist, dann wird der Tomcat−Context automatisch in die server.xml eingetragen.
- WAR file for export legt den Namen des WAR−Files fest, wenn das Projekt exportiert wird. Der Name ist allerdings ein absoluter Pfadnamen, d.h. der Pfad verweist direkt auf das Verzeichnis in das deployed werden soll.

Wenn alles so vorbereitet ist, dann ist im Kontextmenü des Projekts ein zusätzlicher Eintrag vorhanden:

- **Abbildung 6.4. Tomcat−Projekt Optionen im Kontextmenü**



Das WAR[4]−File läßt sich ausliefern.

- Der Context in der Tomcat−Konfigurationsdatei kann entfernt bzw. aktualisiert werden.
- Die Tomcat−Bibliotheken können auf Knopfdruck alle zum Projekt hinzugefügt werden.
- Es wird ein Arbeitsverzeichnis erzeugt, welches die aus JSP−Seiten erzeugten Servlets aufnimmt (wozu das gut ist, sieht man später unter).

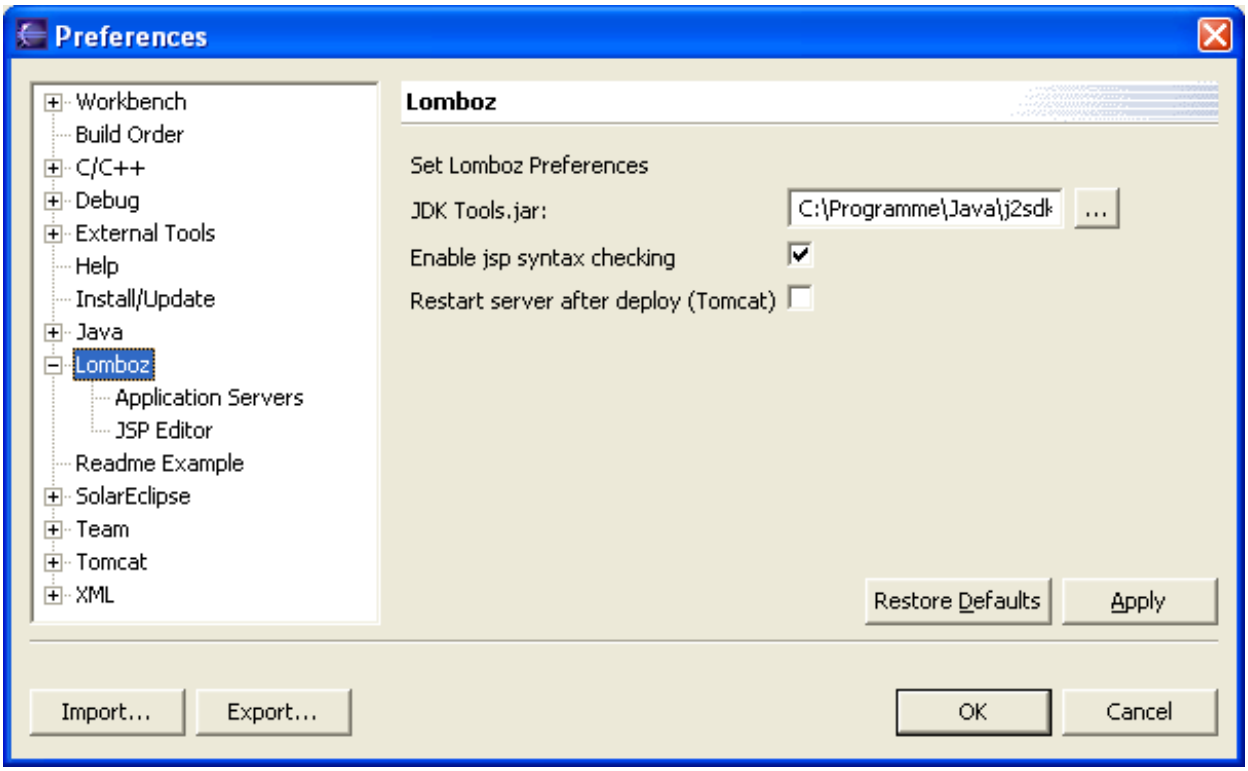
6.2. Lomboz−Plugin von ObjectLearn

ObjectLearn hat mit seinem Plugin ein sehr mächtiges Werkzeug zum Erstellen von J2EE Applicationen vorgelegt. Was hier vorgestellt wird ist nur ein Ausschnitt aus der gesamten Funktionalität, der sich mit WebApplications beschäftigt. Der ganze EJB−spezifische Teil wird hier zunächst ausgespart.

6.2.1. Installieren und Einrichten

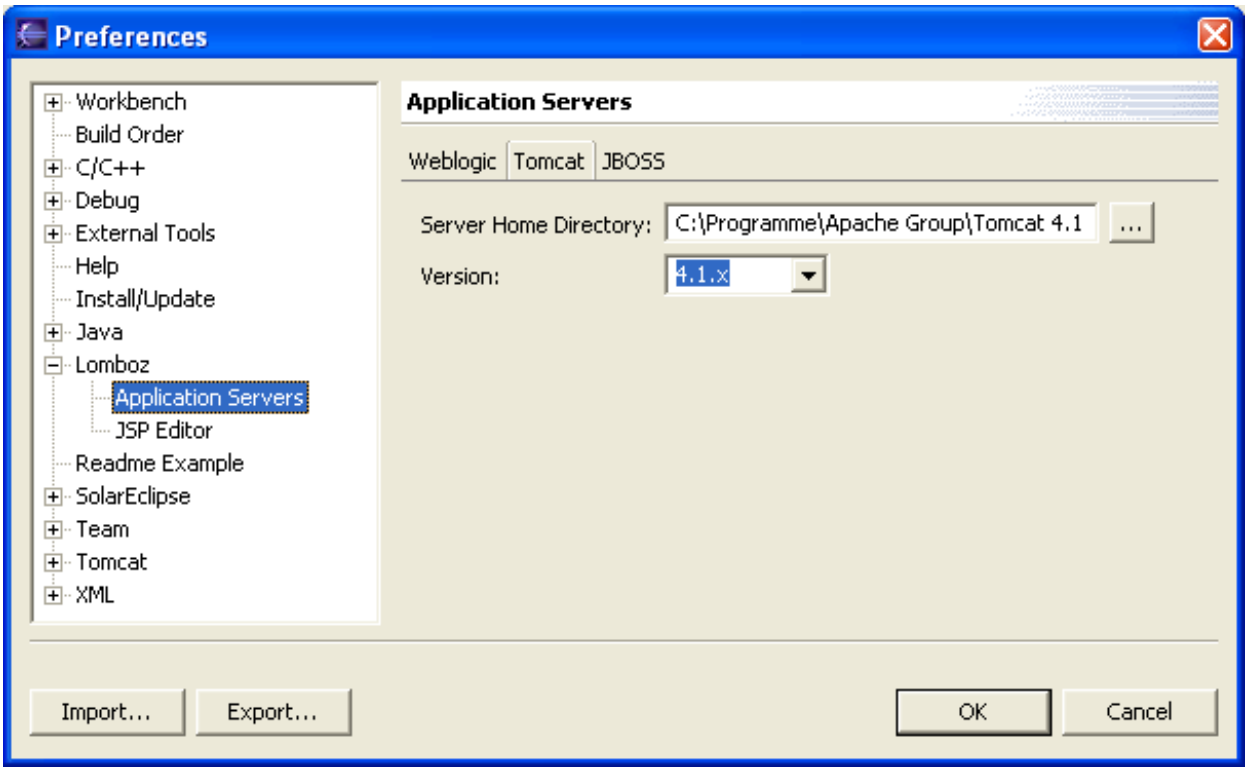
Zunächst wird das ZIP−Archiv in die Verzeichnishierarchie von Eclipse entpackt. Nach dem Neustart von Eclipse läßt sich das Plugin unter Windows / Preferences konfigurieren:

Abbildung 6.5. Konfiguration von Lomboz 1



Zunächst gibt man den Pfad zur `tools.jar` vor. Das ist die Bibliothek aus dem JDK welche den Compiler enthält. Falls man mehrere JDKs installiert hat, sollte man darauf achten, dass der Compiler des JDKs verwendet wird, welches auch für das Projekt eingestellt ist.

Abbildung 6.6. Konfiguration von Lomboz 2

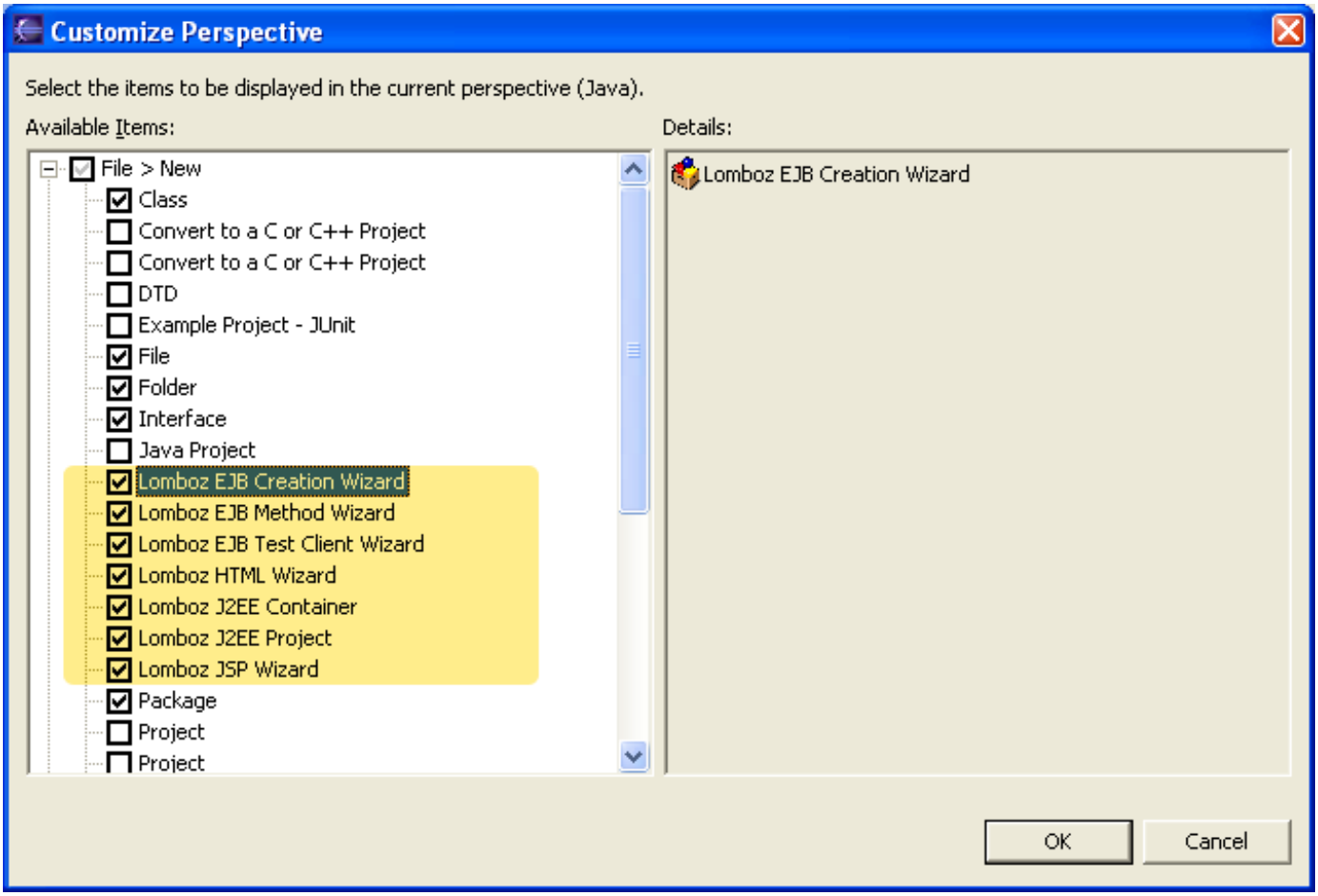


Dann wählt man die passenden Parameter für die Tomcat−Installation.

Die Einstellungen des JSP−Editor sind eher kosmetischer Natur und sollen hier nicht weiter erwähnt werden.

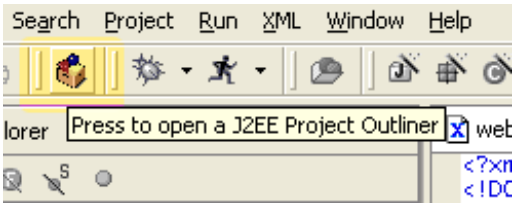
Damit die verschiedenen Views und Actions sichtbar werden, muss man in der Java−Perspective über Customize Perspective (genau wie oben beim Tomcat−Plugin) einige der Lomboz−Optionen aktivieren:

Abbildung 6.7. Konfiguration von Lomboz 3



Entsprechend werden im Zweig Windows / Show View der Lomboz J2EE View aktiviert und im Zweig Other die Lomboz Actions .

Neben den Wizards zur Erstellung neuer Resourcen, kommt in der Taskleiste damit ein neuer Button J2EE Project Builder hinzu.

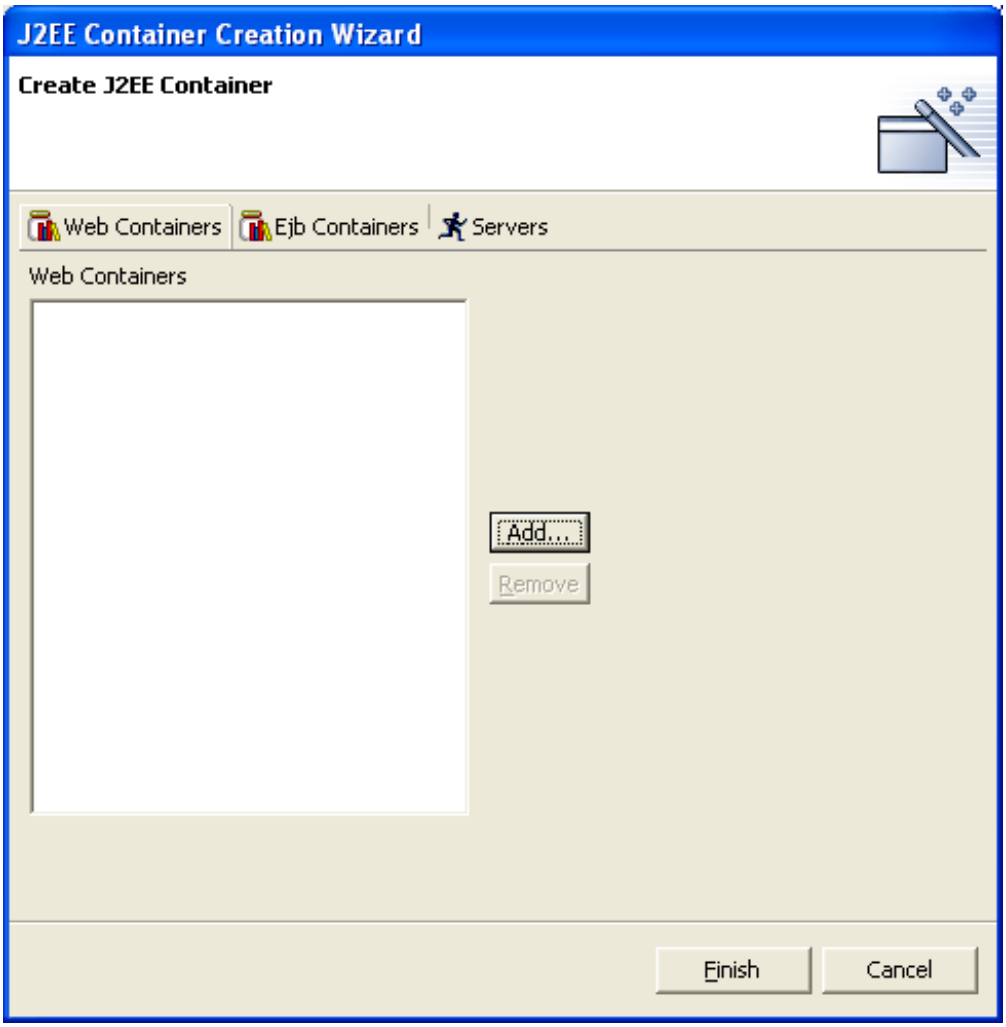


Damit lassen sich sehr schnell sogenannte Container erstellen. Unter Container versteht Lomboz z.B. Einen WebContainer also genau das, was in unserem Beispielprojekt das Verzeichnis `web` schon ist.

Damit Lomboz den vorhandenen Web−Container erkennt, muss eine Datei `build.xml` und `servers.xml` im Verzeichnis `WEB-INF` vorhanden sein (neben `web.xml` die sowieso immer da liegt).

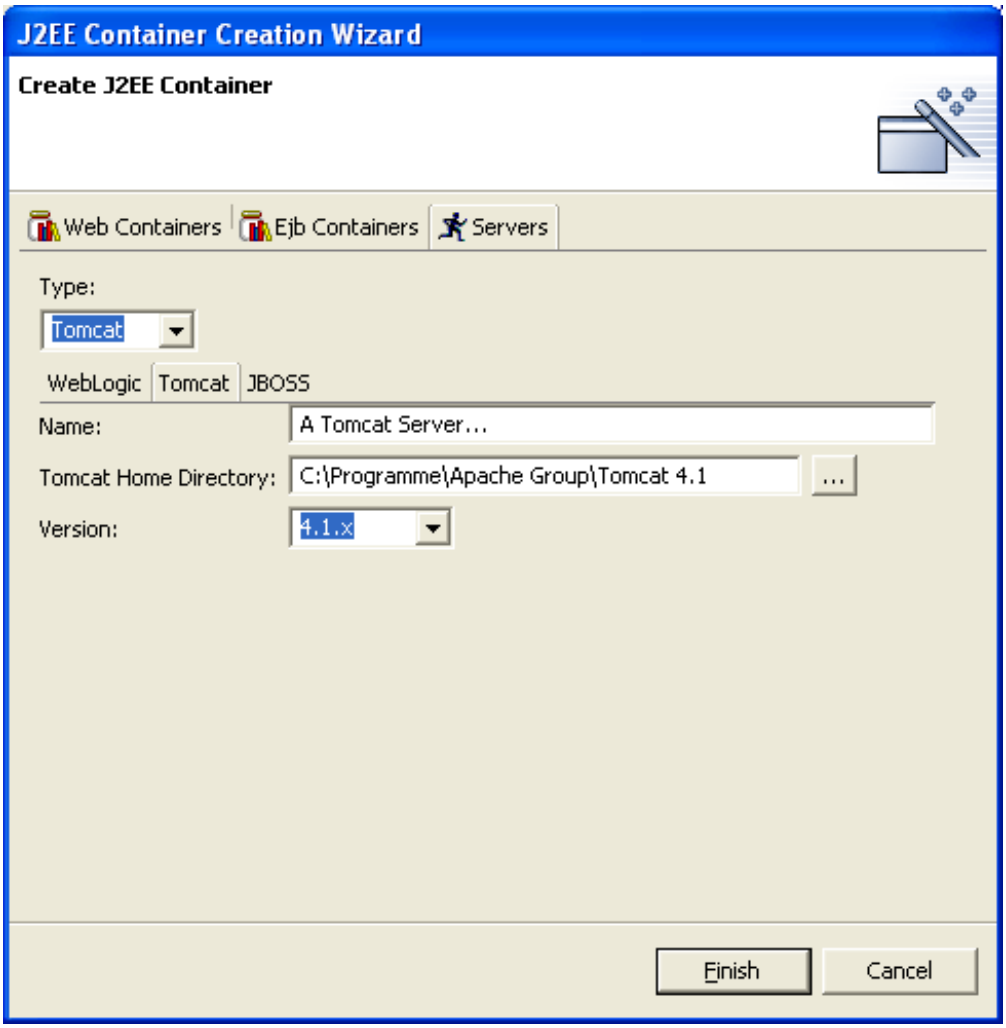
Was in den beiden Dateien drinstehen muss, erfahren wir am besten, wenn wir einfach einen neuen Web−Container anlegen. Man öffnet den J2EE Project−Outliner und klickt Add Container . Der folgende Dialog erscheint:

Abbildung 6.8. Hinzufügen eines Containers



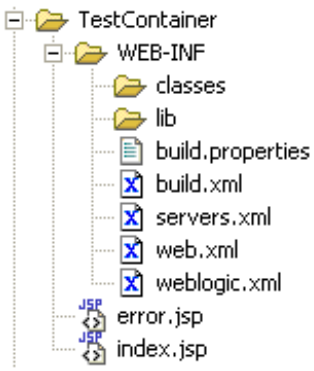
Nun fügen wir einen Web−Container hinzu und geben ihm den Namen test (web gibt es ja schon). Dann schaltet man zunächst weiter auf den Reiter Servers und definiert als Applicationserver Tomcat:

Abbildung 6.9. Servertyp für den Container



Sobald der Dialog mit Finish verlassen wird, sehen wir im Package−Explorer den neu erstellten Container.

Abbildung 6.10. Struktur des Web−Containers



Er enthält neben den normalen Verzeichnissen `WEB−INF` und `classes` auch gleich das Verzeichnis `lib`. Zusätzlich zum Deploymentdescriptor `web.xml` erzeugt Lomboz auch noch die Dateien `build.xml`, `build.properties`, `servers.xml` und `weblogic.xml`.

Die build.xml ist eine Ant−Build−Datei mit drei Targets deploy , init und undeploy . Ähnlich wie zuvor beim Sysdeo−Plugin wird hierüber das Deployment gesteuert. Man kann allerdings durch abändern der build.xml und der build.properties beliebig erweitern oder an eigene Bedürfnisse anpassen.

Die Datei servers.xml beschreibt den Applicationserver für diesen Container. In unserem Fall also wie zuvor eingestellt den Tomcat. Welche Parameter sich genau dahinter verbergen, kann man durch öffnen der Datei nachschauen.

weblogic.xml schließlich ist der Deploymentdescriptor für einen Bea−WebLogic−Server, da wir diesen zunächst nicht weiter betrachten, kann die Datei weblogic.xml wieder gelöscht werden.

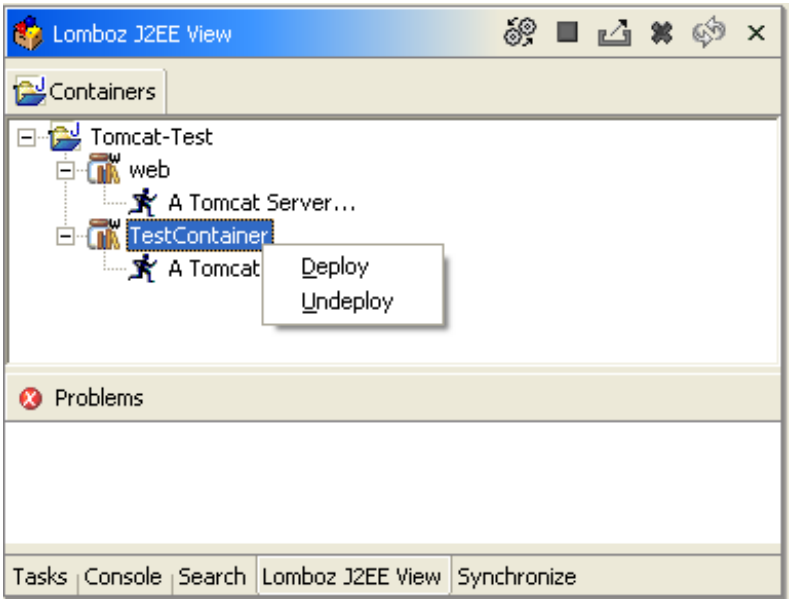
Weiterhin werden im web Verzeichnis selbst gleich zwei JSP−Seiten angelegt: index.jsp und error.jsp .

Wie man sieht kann mit diesem Wizard schnell ein kompletter WebContainer erzeugt werden.

Um nun das bestehende Verzeichnis web in unserem Projekt für Lomboz zum WebContainer zu machen, kopieren wir einfach die Dateien build.xml, build.properties und servers.xml in das web −Verzeichnis. Damit wird web für Lomboz auch zum WebContainer.

Nun kann man den Lomboz J2EE View nutzen und sich die Container darin anschauen:

Abbildung 6.11. Lomboz J2EE View



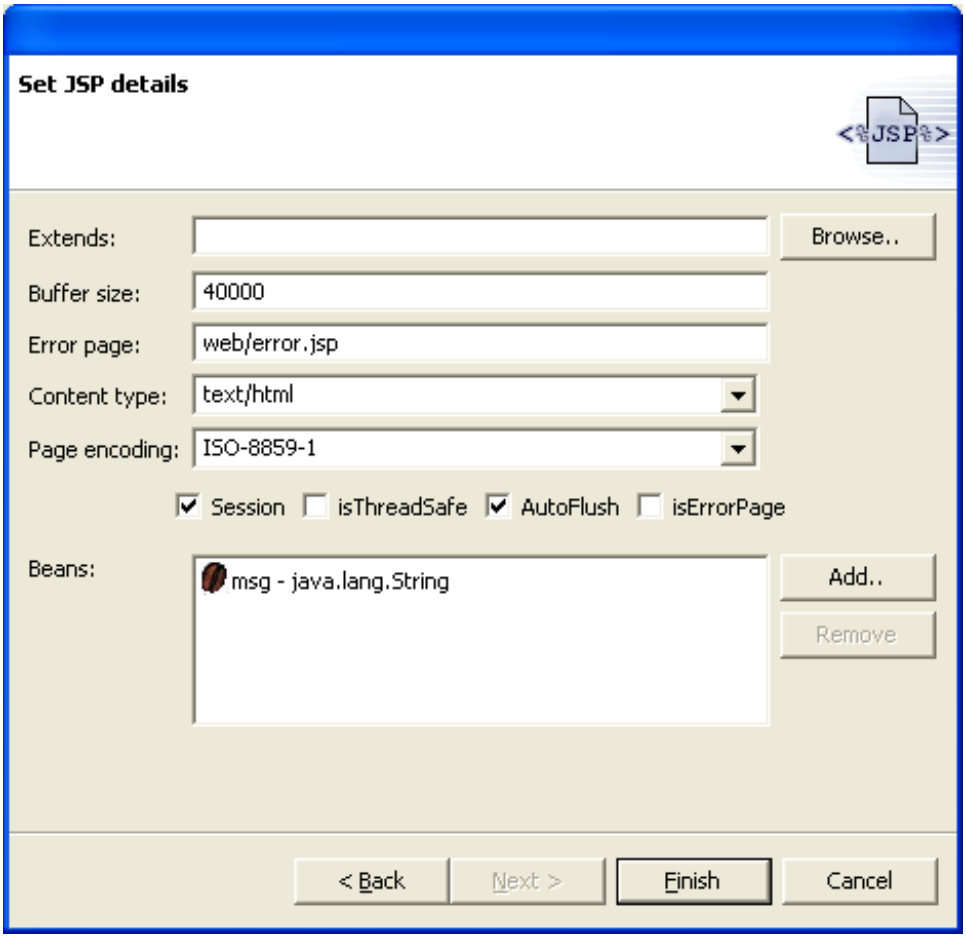
Über die Kontextmenüs und die Pushbuttons am rechten, oberen Fensterrand lassen sich nun verschiedene Aktionen starten, die zuvor in der build.xml definiert wurden. Was über die Standard−Target hinaus in der build.xml integriert wurde, muss per Run Ant direkt aufgerufen werden.

6.3. Wizards des Lomboz−Plugins

Neben dem Verwalten von J2EE Containern, kann Lomboz noch mehr Unterstützung leisten. Es gibt Wizards zum Erzeugen von diversen Objekten. Ausser den EJB bezogenen sind dies folgende:

- Lomboz J2EE Project: erzeugt die komplette Projektstruktur in einem Rutsch, das Ergebnis sieht dann so ziemlich genauso aus, wie das was in den vorherigen Abschnitten Stück für Stück aufgebaut wurde.
- Lomboz HTML Page: erzeugt ein Gerüst für eine HTML−Seite
- Lomboz J2EE Container: erzeugt einen neuen J2EE (Web) Container (wie oben schon erläutert)
- Lomboz JSP Page: erzeugt eine neue JSP Seite. Neben der Definition der Fehler−Seite, lassen sich hier bereits zu benutzenden Beans, mit Id, Scope usw. eintragen.

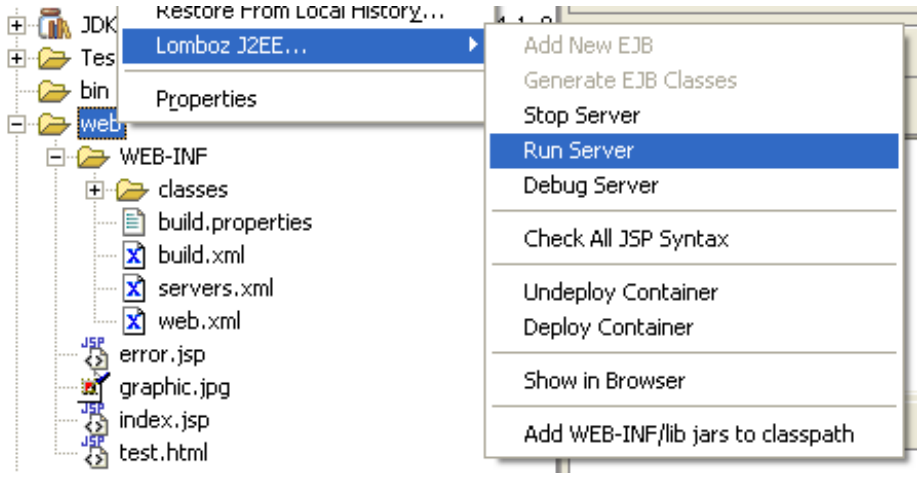
Abbildung 6.12. Lomboz JSP Wizard



6.4. Actions des Lomboz−Plugins

Eine weitere Funktionalität sind die Actions, also Befehle die im Kontextmenü von WebContainern zur Verfügung stehen:

Abbildung 6.13. Lomboz J2EE Actions im Kontextmenü



Die meisten dieser Befehle sind selbsterklärend und müssen nicht nochmal beschrieben werden. Eine Besonderheit ist

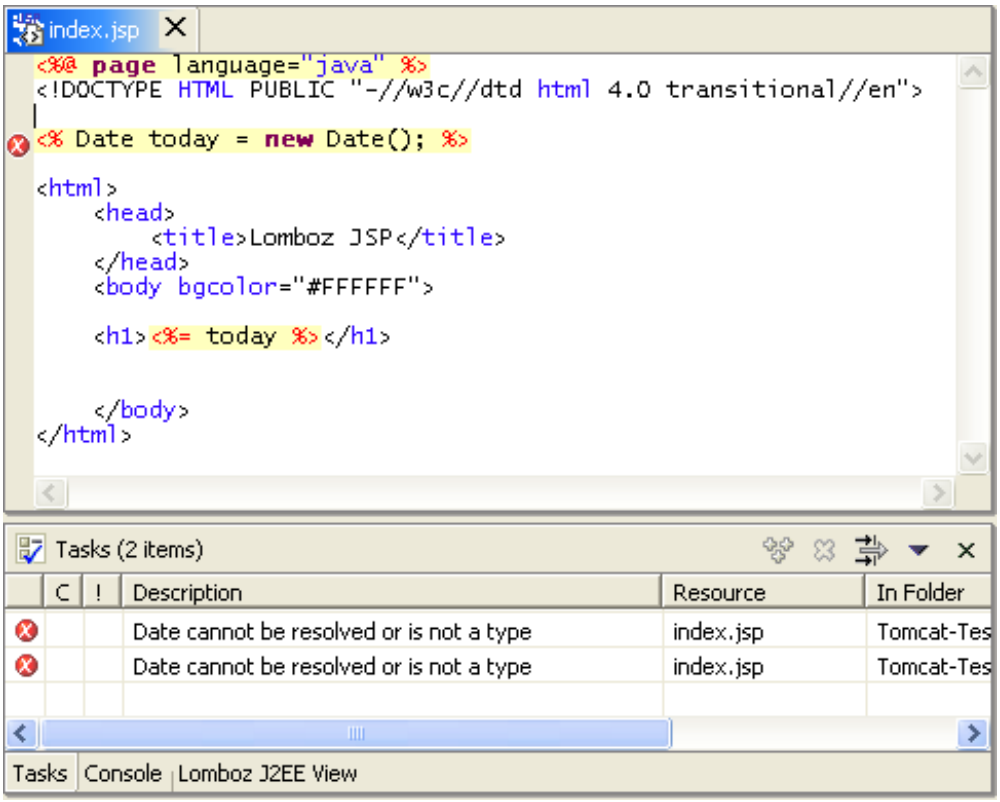
allerdings der Syntaxcheck von JSP−Seiten. Den gibt es nicht nur als Action im Menü, sondern der ist auch im Editor eingebaut.

6.5. JSP−Seiten editieren mit Lomboz

Das Problem kennt jeder: JSP−Seiten fehlerfrei zu erstellen ist viel schwerer als normalen Servletcode. Das kommt einfach daher, dass die Unterstützung mit Autovervollständigung, inkrementellem Compiler, Syntaxcheck und Syntaxhighlightning bei pure Java viel besser ist als bei JSP−Seiten. Mit Lomboz stimmt das alles nicht mehr.

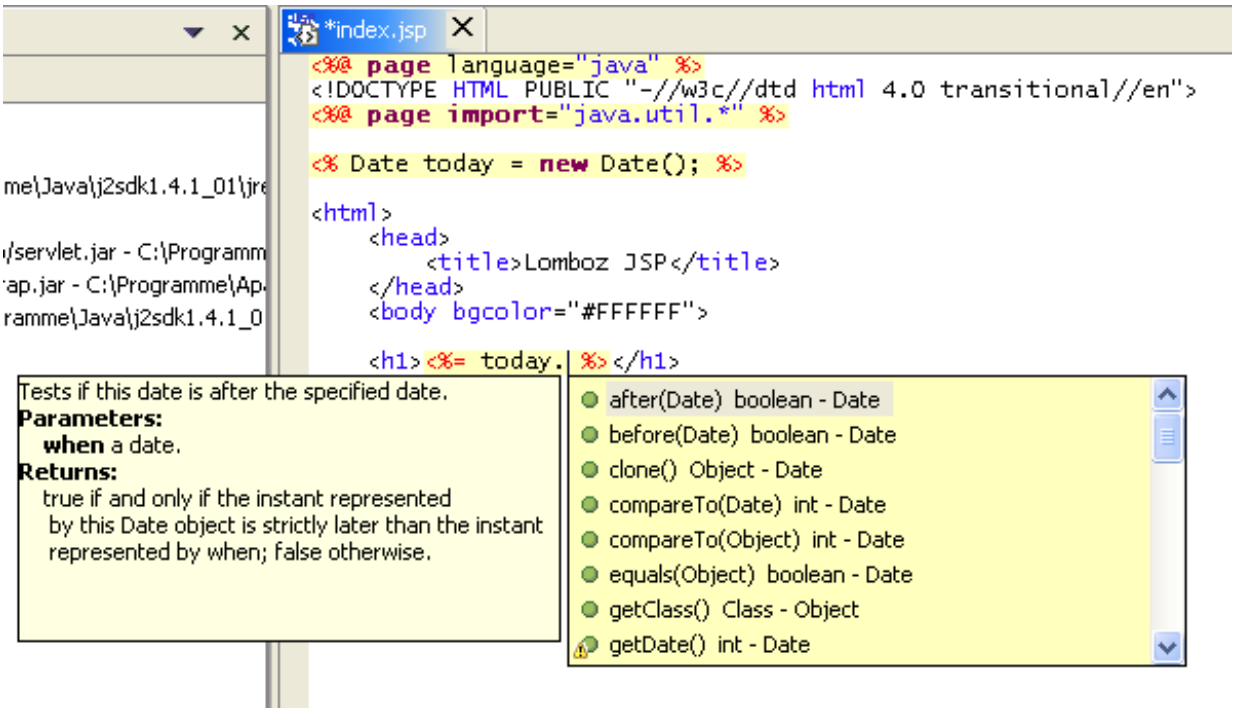
Fehler werden durch inkrementellem Compilieren auch der JSP−Seiten gleich beim abspeichern erkannt:

Abbildung 6.14. Lomboz JSP Editor Fehler in der JSP−Seite



Selbst die vom Java−Quelltexteditor gewohnte Unterstützung mit Autovervollständigung und Popuphelp funktioniert wie gewohnt:

Abbildung 6.15. Lomboz JSP Editor mit Autovervollständigung und Popuphelp



Dadurch ergeben sich natürlich grosse Effizienzsteigerungen, weil die Fehlerwahrscheinlichkeit deutlich zurück geht.

6.6. Zusätzliche Tasks über Ant

Zusätzlich ein build.xml für externe Builds und spezielle Builds, wie z.B. Deploy, install, doc oder ähnliche. build.properties für lokale Buildumgebung. Siehe Lomboz build.xml.

---

[4] (W)eb−(A)pplication−A(R)chive

Kapitel 7. Servlets debuggen

Servlets direkt in Eclipse zu debuggen ist beiden vorgestellten Plugins (Tomcat−Plugin von Sysdeo oder Lomboz von ObjectLearn kein Problem. Beide bieten schon von Hause aus Optionen an, mit dem sich Tomcat direkt in Eclipse starten läßt.

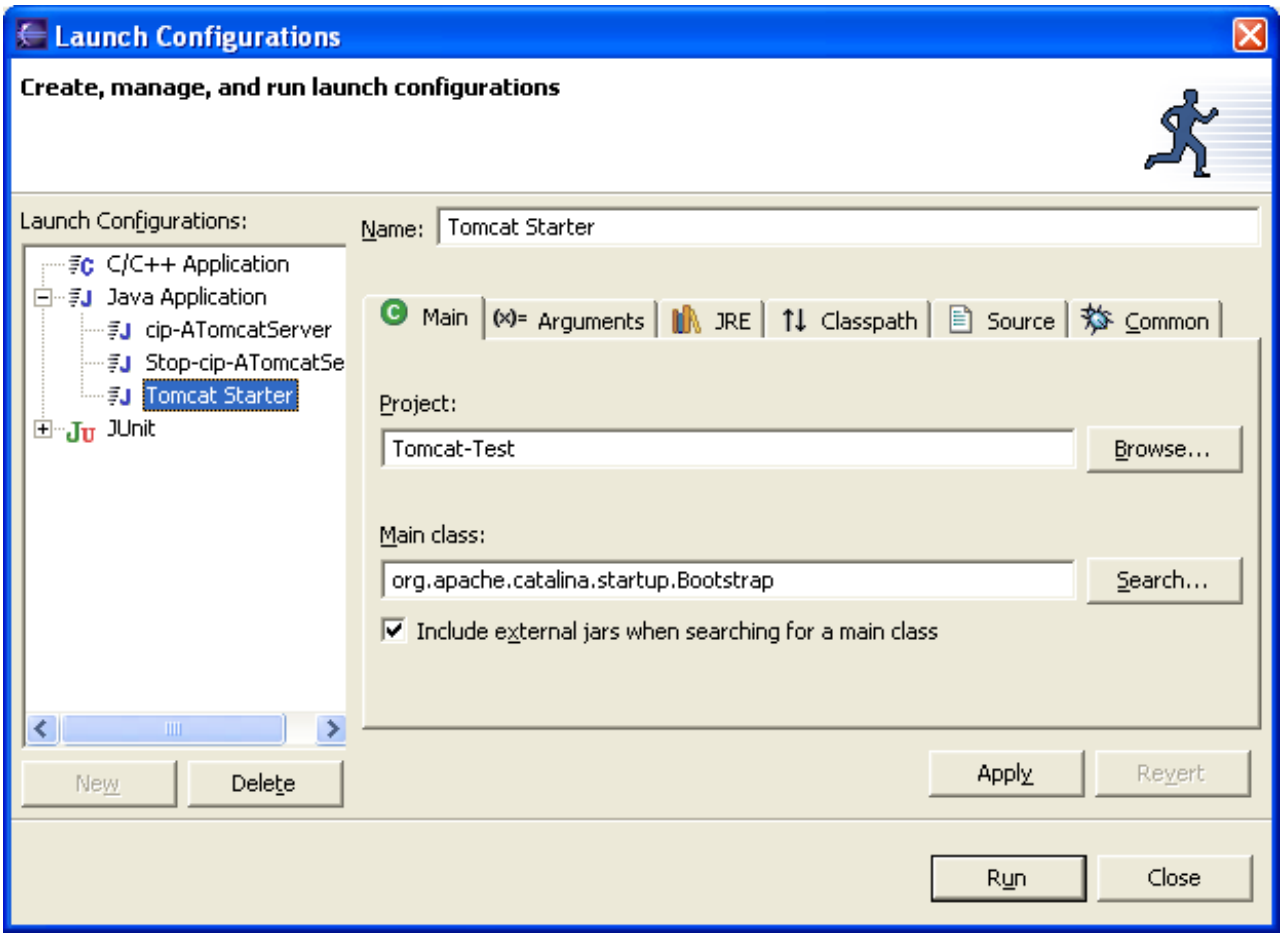
Man kann allerdings auch manuell eine Start−Konfiguration erstellen, die ein Debuggen von Servlets möglich mancht.

7.1. Start−Konfiguration erstellen

Eine eigene Start−Konfiguration zu erstellen, geht wie folgt:

Im Run−Menü, die Option Run... wählen und folgenden Dialog erscheint:

Abbildung 7.1. Start−Konfiguration erstellen



Neben Zuordnung zu einem Projekt und dem Benennen muss vor allem der Start−Punkt festgelegt werden. Im Falle von Tomcat ist das org.apache.catalina.startup.Bootstrap das im bootstrap.jar im bin−Verzeichnis von Tomcat liegt. Als Argument muss man start übergeben und falls noch nicht geschehen, muss das Archiv bootstrap.jar auch in den Classpath aufgenommen werden. Dies alles erledigt man in den unterschiedlichen Abschnitten des Dialogs und hat damit eine Startkonfiguration für Tomcat erstellt.

Die Argumente an die VM (zweiter Reiter Arguments) müssen so aussehen:

```
−DJAVA_HOME="C:/Programme/Java/j2sdk1.4.1_01" −Dcatalina.base="C:/Programme/Apache Group/Tomcat 4.1" -
```

Wobei man das Verzeichnis C:/Programme/Apache Group/Tomcat 4.1/ durch das gültige Installationsverzeichnis ersetzen muss.

Analog kann man zum Herunterfahren des Servers vorgehen. Der einzige Unterschied: als Argument wird stop eingetragen.

7.2. Debuggen mit dem Tomcat−Plugin

Das Tomcat−Plugin von Sysdeo macht das Starten und Stoppen von Tomcat noch etwas einfacher. Wenn in der Debug−Perspective das Plugin aktiviert ist, dann stehen in der Taskleiste automatisch entsprechende Buttons für Start , Stop und Restart zur Verfügung.

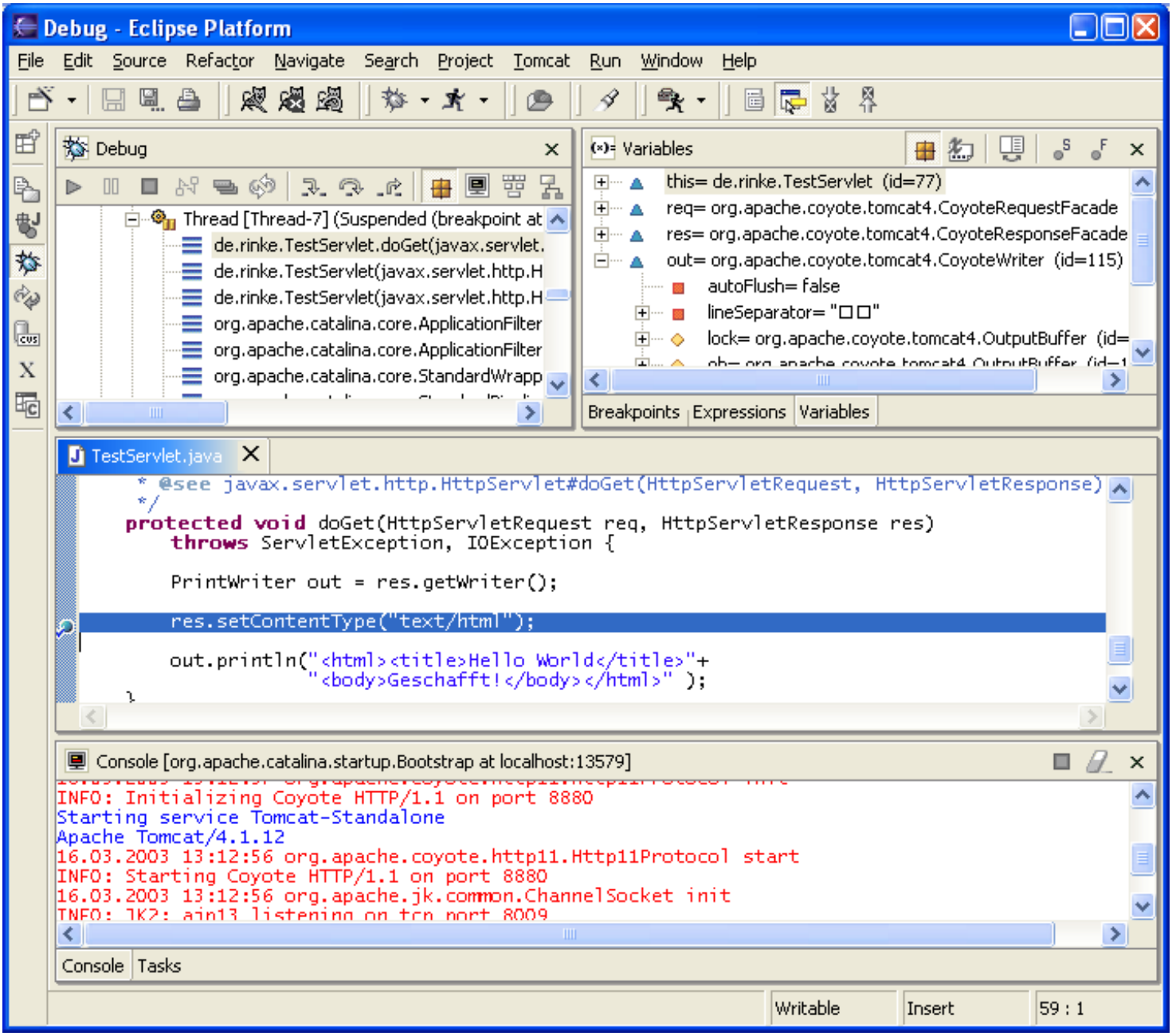
7.3. Lomboz−Actions zum Starten von Tomcat

Entsprechend der Vereinfachung mit Sysdeo Tomcat−Plugin gilt auch für Lomboz: für jeden WebContainer ist Start und Stop schon vordefiniert und direkt als Action ausführbar. Entweder im Contextmenü oder im J2EE−View in der Titelleiste (siehe Abbildung)

7.4. Debuggen

Mit einem so innerhalb von Eclipse gestarteten Tomcat kann man nun nach Belieben Breakpoints setzen und Schritt für Schritt, durch den Code des Servlets durch marschieren. Natürlich muss der Aufruf von aussen auch noch angestossen werden, im einfachsten Fall in dem wir die Seite einfach im Browser aufrufen.

Abbildung 7.2. Servlet im Debug−Modus auf einem Breakpoint



7.5. Hot Codereplacement

Ab der Version 1.4.1 der Java VM funktioniert das Codereplacement auch endlich recht gut. D.h. Man kann einer laufenden Anwendung neuen Code unterschieben. Wenn man z.B. während des Debuggens einen Fehler feststellt und gleich korrigiert, dann wird nach dem Abspeichern in Eclipse dieser Code sofort in die laufende Application übernommen.

Kapitel 8. JSP Seiten debuggen

Das Lomboz−Plugin liefert ja bereits beim Erstellen von JSP−Seiten sehr viel Unterstützung, so dass viele Fehler vermieden werden können. Allerdings kann es trotzdem vorkommen, dass man einem Fehler nur im Debugger auf die Spur kommt. Das ist bislang gerade bei JSP−Seiten relativ schwierig, weil sie sich nicht so ohne weiteres mit Breakpoints versehen und Codestellen inspizieren lassen.

Bekanntlich werden JSP−Seiten spätestens vor dem ersten Aufruf zu einem ganze normalen Servlet übersetzt. Wenn man dafür sorgt, dass diese generierten Servlets im Workspace von Eclipse auftauchen, dann kann man genau wie in Kapitel .. beschrieben mit dem Debugger arbeiten.

8.1. Schritt für Schritt

Damit alles richtig funktioniert sind folgende Schritte notwendig:

1. Eigenes Source−Verzeichnis einrichten, welches die Servlets aufnimmt.
2. Arbeitsverzeichnis von Tomcat so abändern, dass Servlets genau in diesem Verzeichnis erzeugt werden.
3. Tomcat Servlet−Compiler Jasper patchen, weil in diesem Szenario eine Schwierigkeit zu Tage tritt, die sonst eigentlich keine Rolle spielt.

Punkt 1 wird vom Lomboz−Plugin automatisch erledigt. Das Verzeichnis j2src ist Ihnen vielleicht schon aufgefallen.

Punkt 2 erfolgt analog wie die Änderung des Contextes in der Tomcat−Konfigurationsdatei (siehe Abschnitt 3.1.4)

```
<Context reloadable = "true" displayName="Develop"
docBase="C:\Programme\eclipse\workspace\Test−Servlet\web" cookies="true"
path="/test"/
workDir="C:\Programme\eclipse\workspace\Test−Servlet\j2src\org\apache\jsp" />(1)
```

(1) hier wird genau auf das j2src −Verzeichnis verwiesen, damit Jasper die generierten Servlets darin ablegt.

Der Grund warum das workDir auf org\apache\jsp endet ist derselbe, der es auch erforderlich macht Tomcat zu patchen[5]:

Damit die Servlets in Eclipse compiliert werden können müssen die Package−Deklarationen zu den Verzeichnissen passen. Das Standard−Package für JSP−Servlets ist immer org.apache.jsp und der Tomcat−Patch sorgt nun dafür, dass für Seiten, die nicht auf der obersten Ebene des Webservers liegen, die Package−Deklaration von Jasper entsprechend erweitert wird.

Eine JSP−Seite mit der URI /test/index.jsp erhält demnach das Package org.apache.jsp.test .

Zunächst muss man sich die zwei geänderten Class−Files bei Sysdeo herunterladen (findet man im unteren Teil der Seite http://www.sysdeo.com/eclipse/tomcatPlugin.html.

> **Anmerkung: Anmerkung**
> *Achtung*: Für die Version 4.1.12 und 4.1.18 werden unterscheidliche Patches benötigt.

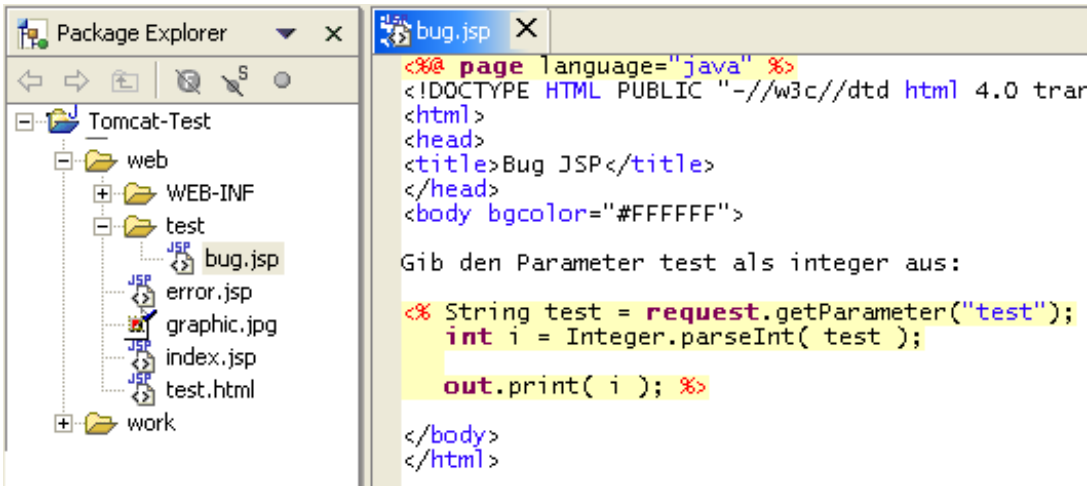
Um den Patch zu installieren muss man drei Class−Files an die richtig Stelle kopieren bzw. in das bestehende JAR−File integrieren. Hat man die Class−Files einzeln unter TOMCAT_HOME/common/classes, kopiert man die aus dem ZIP−Archiv einfach darüber.

Liegen die Class−Files unter TOMCAT_HOME/common/lib als JAR−Archiv vor, so muss man die dort Class−Files des Archivs ersetzen (z.B. mit WinZip). Bitte machen Sie vorher eine Sicherheitskopie von den betreffenden Dateien, damit nichts schiefgehen kann.

8.2. JSP−Seite im Debugger

Wenn alles so weit vorbereitet ist, dann können wir loslegen: zwei JSP−Seiten haben wir schon und zum Testen erzeugen wir im Verzeichnis test eine weitere Seite bug.jsp .

Abbildung 8.1. neue JSP−Seite zum Debuggen



Nun rufen wir die Seite mit dem Browser auf und tatsächlich erhalten wir nicht das gewünschte Ergebnis.

Da Eclipse von den im Hintergrund durch Jasper kompilierten Seiten nichts mitbekommt, müssen wir zunächst im Contextmenü von j2src auf Refresh klicken.

Jetzt sieht man alle bereits kompilierten JSP−Servlets im Projekt−Workspace. An der interessanten Stelle setzen wir nun einen Breakpoint und rufen die Seite nochmal im Browser auf: et vóila.

Abbildung 8.2. JSP−Seite im Debugger

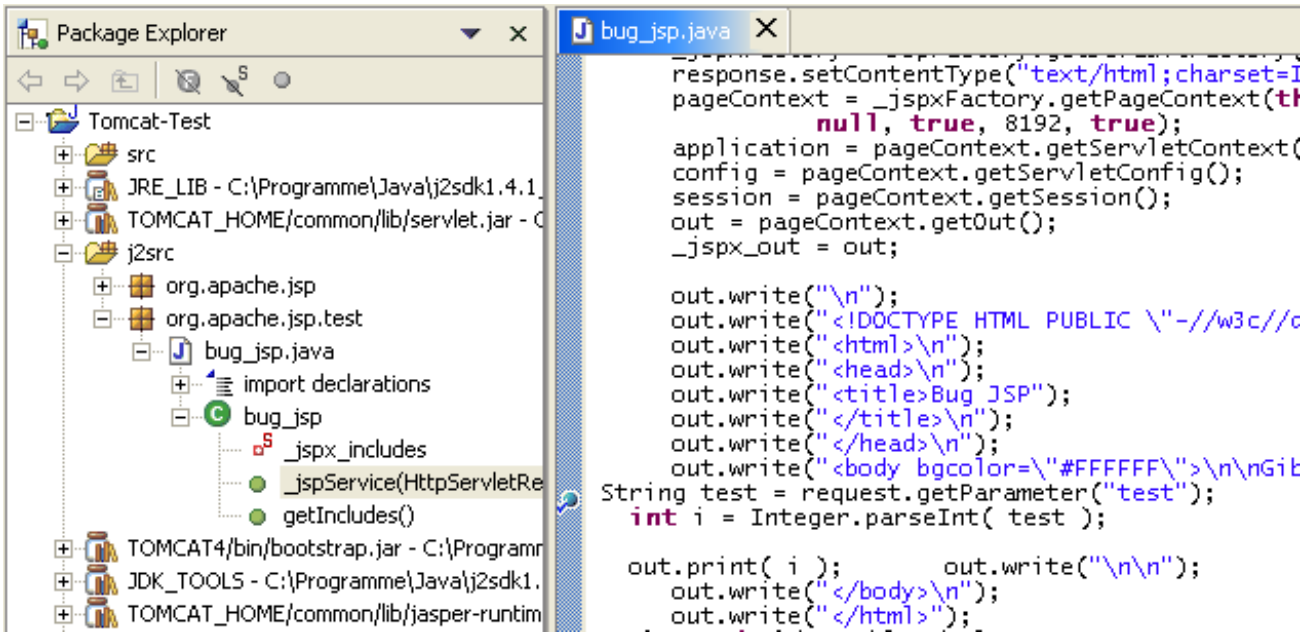


Abbildung.

[5] Nur die Versionen 4.x bei 3.x kann man auf den Patch verzichten

Kapitel 9. Unittests

Unter Unittests versteht man eine standardisierte Testmöglichkeit (am besten) aller Module und Methoden einer Applikation. Diese Tests sollten einfach durchführbar und jederzeit zu wiederholen sein. So kann jederzeit sicher gestellt werden, dass die Applikation die durch die Tests festgelegten Funktionalitäten erfüllt.

Für die Javawelt steht mit JUnit ein erweiterbares Framework zur Verfügung, welches sich grosser Beliebtheit erfreut und in Eclipse direkt über ein entsprechendes Plugin unterstützt wird.

9.1. Unittest mit Eclipse

Junit wird von Eclipse durch einen speziellen Typ von Run−Configuration und einer eigenen View unterstützt, welche die Testergebnisse anzeigt und den letzten Test erneut starten kann.

Um einen Unittest zu schreiben erweitert man einfach junit.framework.TestCase und fügt lauter Methoden ein, die mit dem Namen test beginnen (z.B. testMethod1). Mit Hilfe einiger Methoden aus dem Junit−Framework überprüft man anschließend die Ergebnisse.

Beispiel:

```
import junit.framework.TestCase;

/**

* A simple TestCase to show how junit works.

* @author sr

*/

public class MyFirstTest extends TestCase {

public MyFirstTest( String name ) {

super( name );

    }


public void testFoo( ) {

        Foo foo = new Foo();

        String s = foo.getString( 1 );


        assertNotNull( s );

    }

}
```

9.2. Unittests mit Servlets

Damit Unittests mit Servlets durchführbar sind, bedarf es einiger Vorbereitungen. Zu einen laufen Servlets nur in einer ganz bestimmten Umgebung, dem Servlet−Container (z.B. Tomcat), zum anderen werden Servlets nicht direkt aufgerufen, sondern indirekt über einen HTTP−Request.

Sie liefern auch nicht immer direkt Ergebnisse in Form eines Wertes oder einer HTML−Seite zurück, sondern verändern vielleicht nur den Zustand der serverseitigen Session.

Um all diesen Randbedingungen gerecht zu werden, wurde Cactus geschrieben.

9.3. Unittests mit Cactus

Cactus erweitert JUnit und liefert alles was zum Testen von Servlets, JSP−Seiten und Filtern gebraucht wird:

- Cactus läuft selbst auf dem Server (dort wo das Servlet auch läuft).
- Cactus bildet alle wichtigen APIs nach, so dass alle Parameter, die ein Servlet normalerweise über die Configuration oder den Request erreichen, vom TestCase gesteuert werden.
- Cactus kann das Testergebnis im Browser oder direkt in Eclipse[6] anzeigen.

9.3.1. Installation

Nach dem Download des Cactus−Archivs (aktuell ist z.Z. Cactus−1.5) muss man zunächst dafür sorgen, dass alle Jar−Files, die auf der Server−Seite gebraucht werden im jeweiligen Container gefunden werden. Mit Tomcat geht das am einfachsten so, dass man die JARs `aspectjrt.jar`, `cactus.jar`, `commons−logging.jar`, `commons−httpclient.jar` und `junit.jar` (jeweils mit Versionsnummer) in den `common/lib` −Pfad von Tomcat kopiert. Alternativ kann man all diese JARs auch jeweils ins Lib−Verzeichnis der Web−Applikation kopieren. Dann steht die Testmöglichkeit mit Cactus eben nicht allen Web−Applikationen automatisch zur Verfügung, sondern immer nur der einen, die die JARs auch mitbringt.

Anschließend wird ein spezielles Servlet, der ServletTestRunner , entweder in der globalen web.xml oder in der web.xml der jeweiligen Applikation konfiguriert.

Die globale web.xml von Tomcat findet sich in conf/web.xml . Zwei Abschnitte sind einzufügen [7]:

Am Anfang:

```
<servlet>

   <servlet−name>ServletTestRunner</servlet−name>

   <servlet−classcolor='blue'>gt;

     org.apache.cactus.server.runner.ServletTestRunner

   </servlet−classcolor='blue'>gt;

</servlet>
```

Und am Ende:

```
<servlet−mapping>
```

```
    <servlet-name>ServletTestRunner</servlet-name>

    <url-pattern>/ServletTestRunner</url-pattern>

</servlet-mapping>
```

9.3.2. Ein erster Test

Als zweiten Schritt muss man nun noch einen Test schreiben, der ein Servlet testet und seiner Web-Applikation hinzufügen. Hierzu wird einfach eine Klasse erzeugt, die ServletTestCase erweitert und ähnlich wie bei JUnit eine oder mehrere testXxx-Methoden enthält. Beispiel:

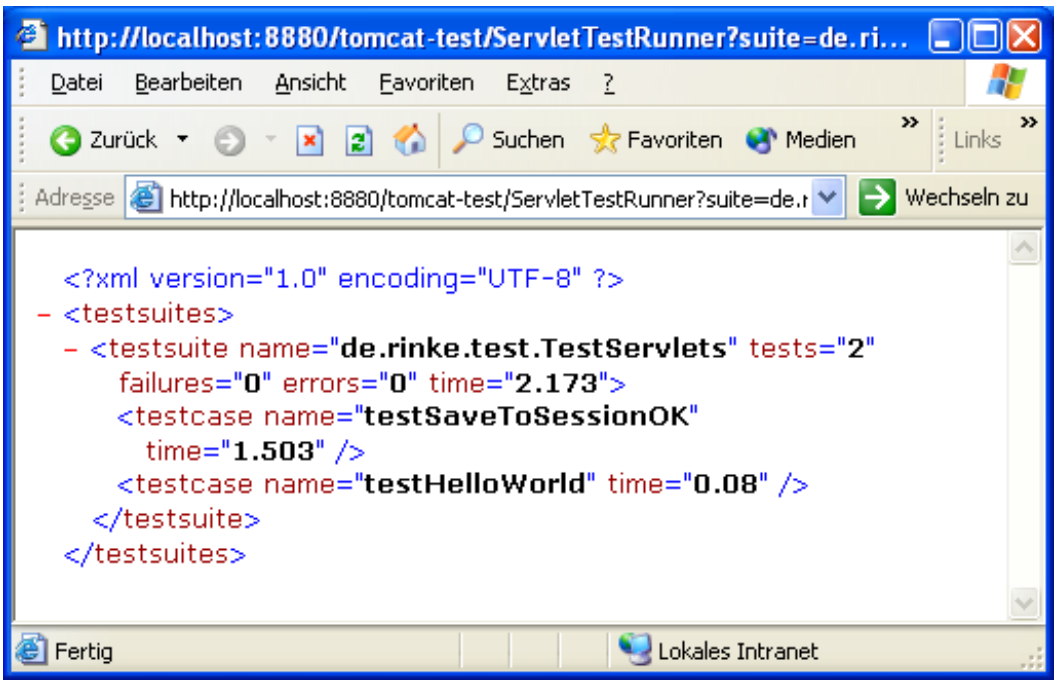
```
import junit.framework.Test;

import junit.framework.TestSuite;

import org.apache.cactus.ServletTestCase;

import org.apache.cactus.WebRequest;

import de.rinke.SaveToSessionServlet;

public class TestServlets extends ServletTestCase

{

public TestServlets(String theName)

    {

super(theName);

    }

public static Test suite()

    {

return new TestSuite(TestServlets.class);

    }

public void beginSaveToSessionOK(WebRequest webRequest)(1)

    {

        webRequest.addParameter("testparam", "it works!");

    }

public void testSaveToSessionOK()(1)

    {

        SaveToSessionServlet servlet = new SaveToSessionServlet();

        servlet.saveToSession(request);
```

```
        assertEquals("it works!", session.getAttribute("testAttribute"));

    }


public void testHelloWorld() {(1)

        assertNotNull( null );

    }

}
```

- Als Erweiterung zu normalen Unittest mit JUnit wird der Aufruf des Servlets mit beginXxx Methoden vorbereitet. Hier werden bestimmte Parameter in den Request eingesetzt , die das Servlet als Aufrufparameter erwartet. Normalerweise kommen diese Parameter aus dem Request des Browsers im TestCase werden die Parameter extra erzeugt.
- Dieser Test wird dann erfolgreich sein, wenn das Servlet ein bestimmtes Attribut in der Session erzeugt. Das passende Servlet zu schreiben sollte nicht schwer fallen.
- Dieser Test wird immer scheitern, weil eine der Assert−Methoden hier assertNotNull also stelle sicher, dass ungleich null direkt mit null aufgerufen wird. Dies ist hier nur zum Ausprobieren eingesetzt, um zu demostrieren, wie ein TestCase scheitert.

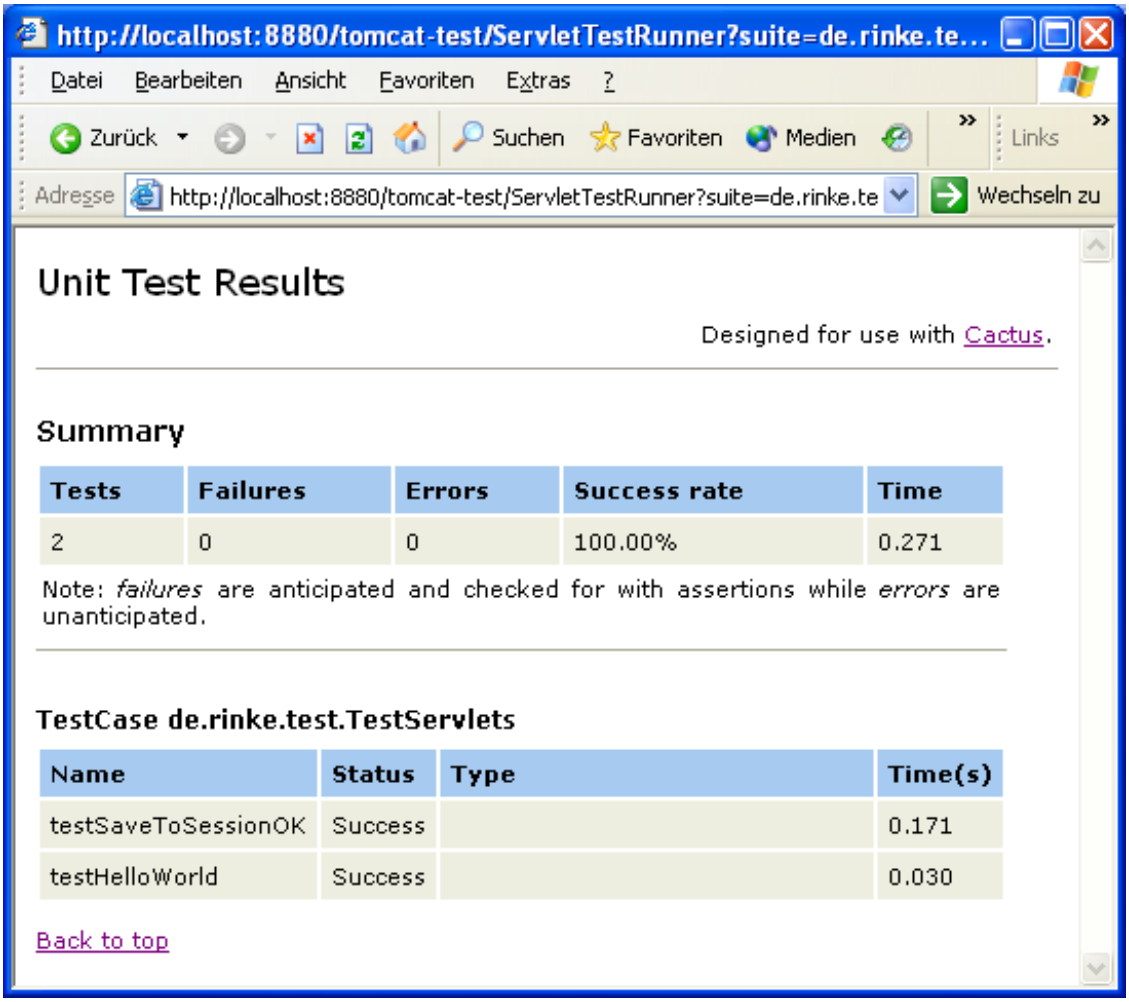
Wenn man ein solches Projekt anlegt und die richtige URL im Browser aufruft, wird der TestCase auf dem Server ausgeführt und das Ergebnis angezeigt
(`http://localhost:8080/tomcat-test/ServletTestRunner?suite=de.rinke.test.TestServlets`)

Abbildung 9.1. Testergebnis in XML



Wenn man das Ganze noch etwas hübscher aussehen lassen will, kann man noch ein Stylesheet hinzufügen und schon sieht das so aus
(`http://localhost:8080/tomcat-test/ServletTestRunner?suite=de.rinke.test.TestServlets`)

Abbildung 9.2. Testergebnis mit einem Stylesheet formatiert



Das komplette Projekt kann man hier herunterladen.

[6] Leider funktioniert das aktuelle Cactus−Plugin nicht richtig mit Eclipse 2.1

[7] Siehe auch: http://jakarta.apache.org/cactus/integration/howto_tomcat.html

Kapitel 10. Download Java WebApplications mit Eclipse

- PDF
- HTML
- Microsoft Word
- Windows Help
- Plaintext
- HTML (eine Datei)
- XML (DocBook)
- OpenOffice
- Eclipse−Plugin
- Build Framework (um selbst mit DocBook solche Artikel zu schreiben)

Kapitel 11. Anhang

11.1. Online−Referenzen

11.1.1. Eclipse Plugin Verzeichnisse

- http://www.crionics.com/products/opensource/eclipse/eclipse.html
- http://eclipse−plugins.2y.net/eclipse/index.jsp

11.1.2. Andere Online−Artikel

- http://www.oio.de/public/warum−eclipse.htm
- http://www.3plus4software.de/eclipse/index.html

11.1.3. Allgemeine Online Resourcen zu Eclipse

- http://www.eclipse.org/
- http://www.eclipseproject.de/