

1.09

eclipse
MAGAZIN

Deutschland € 9,80
Österreich € 10,80, Schweiz CHF 19,20



eclipse

MAGAZIN

www.eclipse-magazin.de

Exklusiv für
Abonnenten
OSGi
POSTER



Mik Kersten über Mylyn

Video-Keynote von der
W-JAX 2008 „Less is More:
Redefining the 'I' of the IDE“

>> Tools, Open Source & More

Hibernate, Eclipse Mylyn, Eclipse BIRT,
Jasper-Reports, iReport, Jython, PyDev,
JyDT, jLead Community Edition

>> Eclipse Plug-ins

Eclipse ResourceBundle Editor,
ant4eclipse

Weitere Infos S. 3

GROOVY & ECLIPSE

Wie lässt sich eine Groovy-DSL
mit Eclipse-GMF visualisieren? >> 22

Eclipse Babel >> 86

Mythos und Wirklichkeit
der Eclipse-Internationalisierung

OSGi-Framework >> 12

Grundkurs: Eigene Plug-ins erstellen

Multimedia mit Eclipse >> 67

Ein Redaktionssystem auf RCP-Basis

SCHWERPUNKT: TESTEN

Callbacks mit JMock und EasyMock >> 33

QF-Test: Automatisierte SWT-Tests >> 43

Testpraxis mit SWTBot >> 37



D 68864

Datenträger enthält
Info- und
Lehrprogramme
gemäß § 14 JuSchG



Classpath-Container zur Integration verschiedener Build-Umgebungen

Harmonischer Gleichklang

In vielen Projekten existieren neben dem offiziellen Build-Prozess, der für das Erstellen der gesamten, unternehmensweiten Anwendung verantwortlich ist, die Build-Informationen der IDE des Entwicklers. Grundsätzlich sollte aber auch hier das Single-Source-Prinzip gelten und die notwendige Information nur an einer Stelle hinterlegt sein. Dieser Artikel zeigt einen Weg auf, mithilfe von Classpath-Containern dieses Ziel zu erreichen und die Arbeit für den Entwickler zu erleichtern.

>> DOMINIK HIRT

Wer kennt das nicht: Gerade im Anfangsstadium eines Softwareprojekts ist noch Vieles im Fluss und ständige Änderungen sind an der Tagesordnung: Welche Softwarekomponenten benötigt mein Beitrag noch, wollen wir nicht doch noch auf dieses neue Open Source Framework wechseln usw. Die Settings für den Classpath sind so wechselhaft wie das Wetter im Hochgebirge. Neben den Einstellungen in der eigenen IDE muss auch der zumeist ANT-basierte Build-Prozess der Gesamtanwendung zufrieden gestellt, sprich mit den richtigen Einstellungen versehen werden. An dieser Stelle sind also die gleichen Informationen an mindestens zwei Stellen zu pflegen. Hier bemerkt der aufmerksame Entwickler sofort ein bekanntes Muster, den Hinweis von Hunt und David aus ihrem „Pragmatischen Programmierer“ [1], sich doch bitte nicht selbst zu wiederholen, eine Information an einer Stelle zu hinterlegen, und nur dort. Doch wie soll man diese beiden Welten miteinander verheiraten?

Der erste Wurf

Als gläubiger Eclipse-Jünger will man natürlich gleich den Berg zum Propheten bringen: Mithilfe von ant4eclipse [2] ist es möglich, aus jedem normalen

ANT-basierten Build-Prozess die Classpath-Informationen zu benutzen, die Eclipse für seine Projekte in den *.classpath*-Dateien vorhält. Das funktioniert gut, wenn man mit einem Projekt startet, alle Entwickler Eclipse verwenden und der Build-Prozess der Gesamtanwendung auf diese Art und Weise gestaltet wurde. Im vorliegenden Fall traf jedoch keine dieser Voraussetzungen zu: Das Gesamtprojekt, ProClassic/Enterprise (PC/E) ist eine J2EE-Anwendung für den SB-Kanal von Banken mit Filialnetzwerken (Retail Banking). Die Anwendung besteht aus ca. 100 einzelnen Komponenten, die von einem verteilten Entwicklungsteam erstellt und gepflegt werden. Dabei soll es auch noch Entwickler geben, die nicht Eclipse benutzen. Alle Komponenten nutzen den seit Langem etablierten ANT-basierten Build-Prozess. Diesen auf ant4eclipse umzustellen, kam daher nicht in Frage. Der Prophet musste also zum Berg: Die Eclipse-Projekte sollen die Classpath-Informationen aus dem existierenden Build-Prozess beziehen. Dieser sieht für jede Komponente eine Datei *project-build.properties* vor. In dieser Datei, die im *antfile.xml* als *property*-File eingebunden ist, wird der Classpath definiert. Listing 1 zeigt die Ausgangslage in Form

einer bestehenden Datei aus diesem Build-Prozess.

Die benutzten Variablen werden von ANT aus anderen, hierarchisch eingebundenen *property*-Files aufgelöst oder programmatisch durch eigene ANT-Tasks (um-)gesetzt. Nachdem somit die Ausgangslage klar ist, wird nun ein Weg benötigt, diese Informationen den Eclipse-Projekten zur Verfügung zu stellen. Diesem Weg ist dieser Artikel gewidmet. Sie erfahren hier, in welcher Form innerhalb von Eclipse mit Classpath-Informationen umgegangen wird. Danach wird eine Möglichkeit aufgezeigt, diese Informationen von einer externen Quelle zu ermitteln und den Java-Projekten zur Verfügung zu stellen. Schließlich wird noch gezeigt, wie Veränderungen an dieser externen Quelle festgestellt und die Projekte entsprechend aktualisiert werden. Beginnen wir mit einem kurzen Ausflug ins Eclipse-Universum und dazu, wie hier Informationen zum Classpath vorgehalten werden.

Classpath Entries

Jedes Java-Projekt definiert seinen Classpath in einer Datei *.classpath* im Root des Projekts. In dieser Datei werden verschiedene Typen von Einträgen benutzt, um den Classpath zu definieren. Insgesamt gibt es fünf Typen, die alle in *org.eclipse.jdt.core.IClasspathEntry* definiert sind:

- Library – eine Bibliothek, also ein jar-File
- Project – ein anderes Eclipse-Projekt

Listing 1 ✕

```
classpath=${pce.lib.dir}/PceClientBase.jar;\
${pce.lib.dir}/PceBaseFlow.jar;\
${pce.lib.dir}/PceBusinessDomain.jar;\
${pce.lib.dir}/PceSmartClientConnector.jar;\
${jakarta-oro.jarfile};\
${uka.jarfile}
```



- Source – ein Verzeichnis mit Quellcode
- Variable – ein Pfad, der mit einer der Classpath-Variablen beginnt, die im Workspace deklariert wurden
- Container – eine Sammlung von jar-Files, Projekten und Variablen unter einem gemeinsamen Namen

Diese Typisierung findet sich auch im Eclipse User Interface wieder: Schauen Sie auf die ersten drei Tabulatoren/Reiter des Dialogs zum Einstellen des Java Build Path für ein Projekt. Der Classpath eines Java-Projekts kann sich aus diesen Typen zusammensetzen. Besonders jedoch der letzte Typ, der so genannte Classpath-Container, ist interessant, da dieser programmatisch definiert und verändert werden kann. Er erscheint somit als geeigneter Kandidat, um das gewünschte Ziel zu erreichen, den Bezug der Classpath-Informationen für Java-Projekte zu externalisieren. Schauen wir uns also diesen Typ etwas genauer an.

Classpath-Container

Sie sind alle schon einmal mit einem speziellen Classpath-Container in Berührung gekommen: der *JRE System Library*. Hierunter werden die Systembibliotheken der benutzten JRE verwaltet. Ein anderes Beispiel ist der Classpath für JUnit. Auch hierfür stellt das JUnit-Plug-in einen Classpath-Container zur Verfügung. Der dynamische Aspekt, der mit einem Container erreicht werden kann, zeigt sich hier bereits: Auf dem Dialog mit den Einstellungen, der so genannten Container Page, ist es möglich, sich zwischen einzelnen Versionen von JUnit zu entscheiden. Erst in Abhängigkeit dieser Entscheidung wird der Classpath aufgebaut, der dann als *JUNIT_CONTAINER* den Projekten zur Verfügung gestellt wird.

Ein Classpath-Container stellt eine logische Referenz auf eine Sammlung von Classpath Entries zur Verfügung. Für die Entries selbst können die vier Entry-Typen *Library*, *Source*, *Project* oder *Variable* genutzt werden. Diese Menge von Classpath Entries wird als Container einem Java-Projekt zugewiesen und hilft somit, die abhängigen Referenzen zu strukturieren. Das wird auch im Package Explorer deutlich (Abb. 1). Alle Classpath Entries lassen sich unter dem Namen des Containers einklappen

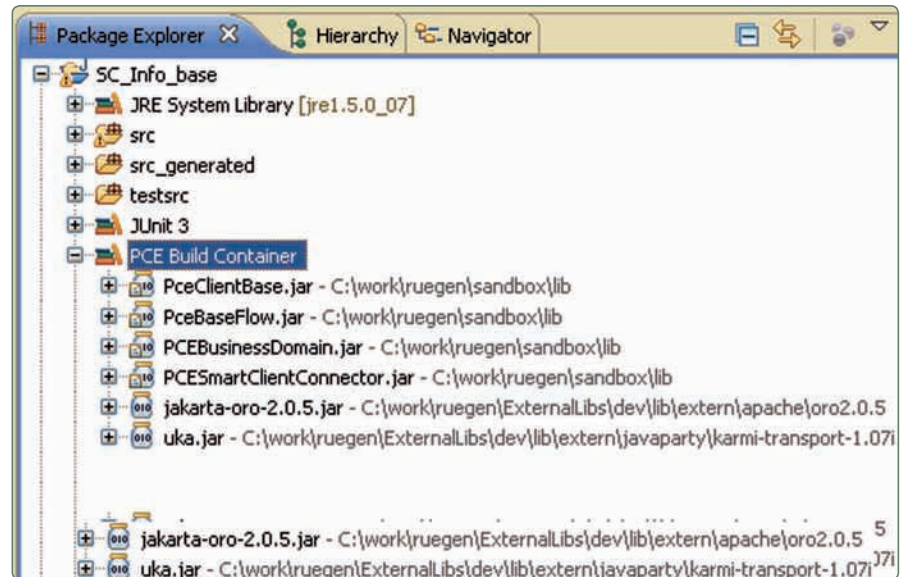


Abb. 1: Aufgeklappter Container im Package Explorer

und verbergen. Die Ansicht erscheint damit übersichtlicher.

Im Gegensatz zu den User Libraries (Kasten: „Container und Libraries“) werden keinerlei plattformspezifische Pfade gespeichert, was die Benutzung von Containern sehr portabel werden lässt. Die *.classpath*-Dateien können im Source Control Management eingetragt und von allen Entwicklern genutzt werden. Die größten Vorteile gegenüber allen anderen Typen von Classpath Entries sind aber Dynamik und Programmatik. Keine der Informationen über die Containereinträge werden in irgendeiner Form im Workspace oder im Projekt gespeichert. Erst in dem Moment, in dem sie wirklich benötigt werden, z.B. zum Zeitpunkt des Kompil-

lierens eines Java-Projekts, werden die Classpath-Container nach ihren Einträgen befragt. Die dazu notwendige Logik wird in Java programmiert.

Wenn es also gelingt, einen solchen Container zu etablieren und ihn mit Werten aus dem *projectbuild.properties* zu füllen, ist das Ziel fast erreicht. Es ist außerdem notwendig, diesen Container in den Java-Build-Path-Einstellungen dem Projekt hinzuzufügen. Wir müssen also Eclipse um einen Classpath-Container erweitern.

Der Weg

Bevor auf die einzelnen Teilaspekte eingegangen wird, soll zunächst die Gesamtlösung skizziert werden: Die Erweiterung wird als gewöhnliches Eclipse

Container und Libraries

Die Bezeichnungen innerhalb Eclipse werden nicht ganz konsistent gebraucht und wechseln zwischen Library und Container. Als Richtschnur kann gelten, dass nach außen hin häufiger von Library gesprochen wird, wenn intern ein Classpath-Container benutzt wird. Auch der Button zum Hinzufügen eines Containers ist mit *Add Library* beschriftet. Die User Libraries sind ein dem Classpath-Container verwandtes Prinzip. User Libraries können unter *Window | Preferences | Java | Build Path | User Libraries* verwaltet werden. Eine User Library stellt eine Liste von Archiven (jar-Files etc.) unter einem gemeinsamen Namen zur Verfügung. Diese kann dann dem Build Path eines Java-Projekts zugewiesen werden. Die Liste der

Archive ist jedoch rein statisch. Sie wird über einen Dialog mit Auswahlmöglichkeit auf jar- oder zip-Dateien erstellt. In den *prefs*-Dateien des Core Plug-ins *org.eclipse.jdt.core* werden die absoluten Pfade abgespeichert. Ist eine solche User Library also Änderungen unterworfen, müssen diese manuell über das UI nachgepflegt werden. Im Gegensatz dazu sind Classpath-Container dynamisch. Da sie programmatisch erstellt sind, stehen völlig andere Möglichkeiten zur Verfügung. Außerdem beginnt Eclipse erst zu dem Zeitpunkt, da ein Container durch ein konkretes Projekt benötigt wird, mit dem Produktionsprozess. Dem Container können nun beliebige Einträge hinzugefügt werden. Lazy Loading also auch auf dieser Ebene.

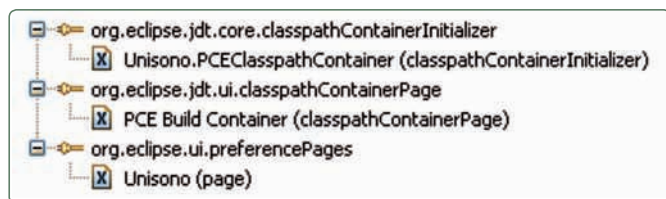


Abb. 2: Die Extension Points

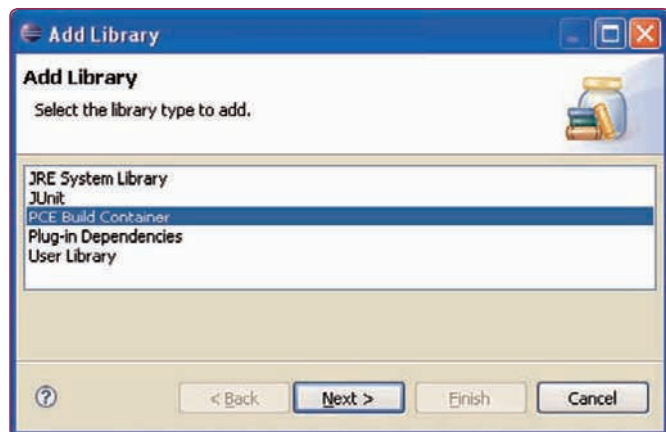


Abb. 2: Die verfügbaren Classpath-Container

Mit dessen Hilfe definiert man einen eigenen Wizard, der den Benutzer durch die Einrichtung des Containers führt. An dessen Ende wird eine neue Instanz unseres Classpath-Containers erzeugt und dem entsprechenden Java-Projekt hinzugefügt.

Einige grundsätzliche Einstellungen für das Plug-in erfolgen unter **WINDOWS | PREFERENCES | UNISONO**. Die hierzu notwendige Seite wird mithilfe des Extension Points `org.eclipse.ui.preferencePages` deklariert. Auf ihr werden zwei Werte vom Benutzer erbeten: das *sandbox*-Verzeichnis des zugrunde liegenden Source Control Managements und das Verzeichnis mit allen externen Bibliotheken. Beide Informationen sind gleichsam die festen Punkte im Universum des PC/E-Build-Prozesses. Nach dieser kurzen Skizzierung der Gesamtlösung sollen nun die einzelnen Schritte detailliert geschildert werden.

Der Container

Listing 2 zeigt ein Fragment der Implementierung des Classpath-Containers. Die Konstante *ID*, als eindeutige Kennung dieses Containers, wird später in allen *.classpath*-Dateien als Hinweis auf unseren Container genutzt. Sie ist vom Typ `org.eclipse.core.runtime.Path`. Durch die Benutzung dieses Typs wäre die Verknüpfung zusätzlicher Informationen, z.B. der Pfad zur *projectbuild.properties* oder Ähnliches, möglich. Das Unisono-Plug-in übernimmt jedoch das Speichern dieser Pfadangaben – neben anderen Daten – selbst und verwendet einen eigenen PreferencesStore. Dem Konstruktor wird ein Objekt des Typs `org.eclipse.jdt.core.IJavaProject` übergeben. Das ist das Java-Projekt, dem der Container zugewiesen wurde. Die Ermittlung der benötigten Classpath Entries aus der ANT-Property-Datei wird an eine Klasse ausgelagert, die unseren ANT-Build-Prozess modelliert. Die Methode *getDescription()* liefert schließlich den Wert, der in der Liste aller Container angezeigt wird.

Der Container Initializer

Ein Container wird durch einen Container Initializer erzeugt und im System bekannt gemacht. Der Extension Point `org.eclipse.jdt.core.classpathContainerInitializer` stellt einen solchen Container Initializer zur Verfügung. Über

Plug-in realisiert. Das Plug-in, der Autor taufte es *Unisono*, wurde mit dem Plug-in Development Environment (PDE) erstellt. Abbildung 2 zeigt die verwendeten Extension Points.

Unser Classpath-Container muss eine Implementierung der Schnittstelle `org.eclipse.jdt.core.IClasspathContainer` sein. Die wichtigste Methode ist hierbei `IClasspathEntry[] getClasspathEntries()`. Diese Methode wird

vom Eclipse Java Model immer dann gerufen, wenn für ein Projekt, dem dieser Classpath-Container zugeordnet ist, die Liste der Classpath Entries benötigt wird, z.B. um das Projekt zu kompilieren. An dieser Stelle müssen wir also die Informationen aus unserer *projectbuild.properties* in die Objekte der Typen *IClasspathEntry* konvertiert haben. Damit der Container überhaupt benutzt werden kann, muss er zuvor initialisiert werden. Der Extension Point `org.eclipse.jdt.core.classpathContainerInitializer` hilft dabei, die Initialisierung immer genau dann automatisch erfolgen zu lassen, wenn der Container benötigt wird. Eine eindeutige ID ist dabei das Bindeglied zwischen dem Projekt, das den Container benötigt, und einer Instanz unseres Containers, die vom Initializer erzeugt wird. Dieser ist auch dafür verantwortlich, ob und wie der Container nach seiner Erzeugung noch manipuliert werden, d. h. seine Einträge verändern kann. Die eigentliche Zuordnung von Classpath-Container zum Java-Projekt erfolgt dann über die Einstellungen für den Java Build Path des Java-Projekts. Dort gibt es den Button **ADD LIBRARY**, der eine Liste aller verfügbaren Container öffnet (Abb. 3). Soll unser Container auch auftauchen, muss das Plug-in noch ein UI für diesen Schritt mitbringen. Das erfolgt durch die Benutzung des Extension Points `org.eclipse.jdt.ui.classpathContainerPage`.

Listing 2

```
public class PCEContainer implements
    IClasspathContainer
{
    public final static IPath ID = new Path
        ("Unisono.PCEClasspathContainer");

    public PCEContainer(IPath containerID,
        IJavaProject javaProject)
    {
        this.entries = Model.getInstance().
            getPCEClasspathEntries(javaProject);
    }

    public IClasspathEntry[] getClasspathEntries()
    {
        return entries;
    }

    public String getDescription()
    {
        return "PCE Build Container";
    }
}
```

den grafischen Plug-in-Manifest-Editor wird der Extension Point dem Plug-in hinzugefügt. Das Extension-Point-Schema definiert zwei Attribute, *id* und *class*, die wir angeben müssen: Als *id* ist exakt die gleiche Kennung zu verwenden, die auch für die eigentliche Containerklasse vereinbart wurde. Der Wert des Attributs *class* beschreibt den voll qualifizierten Klassennamen unserer Implementierung des Classpath Initializers, eine Klasse, die von *org.eclipse.jdt.core.ClasspathContainerInitializer* abgeleitet ist. Listing 3 zeigt die konkrete Implementierung.

Die Methode *initialize* wird in zwei Situationen aufgerufen: wenn der Container am Ende des *Add-Library*-Wizards zum ersten Mal erzeugt wird und später beim Initialisieren der Java-Projekte, denen dieser Container bereits zugewiesen wurde, während des Starts von Eclipse. Der Container wird erzeugt und dem JDT über die Methode *setClasspathContainer* bekannt gemacht. Ein Überschreiben der Methode *canUpdateClasspathContainer* ermöglicht es später, einen einmal zugewiesenen Container zu editieren. Eine solche

Änderung wird dann mit *requestClasspathContainerUpdate* wieder in das JDT propagiert.

Die Container Page

Nach diesen sehr übersichtlichen Implementierungen wird es nun umfangreicher: Wir begeben uns in den Bereich der UI-Programmierung. Ein UI ist in Form eines Wizards nötig, der alle notwendigen Informationen für die Zuordnung unseres Containers zu einem konkreten Java-Projekt vom Benutzer abfragt. In unserem Fall sind dies zwei Dinge: Als Erstes benötigen wir den Pfad zur komponentenspezifischen Datei *projectbuild.properties*, die bekanntlich die Liste der Classpath Entries enthält. Wie oben bereits erwähnt, sind die Einträge in dieser Datei häufig mithilfe von ANT-Properties referenziert. Diese sind im Allgemeinen statisch und können somit leicht ermittelt werden. Eine Ausnahme stellt allerdings der Wert der Variablen *#{pce.lib.dir}* dar: Er kann vom PC/E-Build-Prozess dynamisch geändert werden. Wir müssen daher den Wert dieser Variablen ebenfalls vom Benutzer erfragen. Nach diesen Vorüberlegun-

Anzeige

Listing 3

```
public class PCEContainerInitializer extends ClasspathContainerInitializer
{
    public PCEContainerInitializer()
    {
    }

    public void initialize(IPath containerID, IJavaProject javaProject) throws CoreException
    {
        PCEContainer pceContainer = new PCEContainer(containerID, javaProject);
        JavaCore.setClasspathContainer(containerID, new IJavaProject[] { javaProject }, new IClasspathContainer[]
                                                                    { pceContainer }, null);
    }

    @Override
    public boolean canUpdateClasspathContainer (IPath containerID, IJavaProject project)
    {
        return true;
    }

    @Override
    public void requestClasspathContainerUpdate (IPath containerID, IJavaProject project, IClasspathContainer
                                                containerSuggestion) throws CoreException
    {
        JavaCore.setClasspathContainer(containerID, new IJavaProject[] { project }, new IClasspathContainer[]
                                                                    { containerSuggestion }, null);
    }
}
```

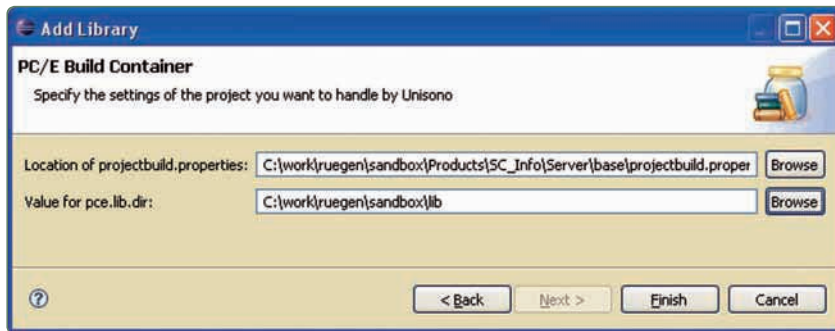


Abb. 3: Die Container Page

gen geht es nun an die konkrete Implementierung. Der Extension Point, den wir hierfür nutzen, ist *org.eclipse.jdt.ui.classpathContainerPage*. Er benötigt wieder die *ID* unseres ursprünglichen Containers, einen symbolischen Namen als Attribut *name* und die implementierende Klasse als *class*. Sie muss von *org.eclipse.jdt.ui.wizards.IClasspathContainerPage* abgeleitet sein. Listing 4 zeigt einen Auszug der Unisono-Implementierung.

Die Deklaration der UI-Elemente erfolgt in der Methode *createControl(Composite parent)* und soll hier nicht näher besprochen werden. Sie erfolgt gemäß den üblichen SWT-Mechanismen. Es werden zwei Textfelder für den Pfad zur *projectbuild.properties* und

den Wert von *pce.lib.dir* erzeugt. Daneben gibt es noch Browse-Buttons für die komfortable Suche im File-System, falls die automatisch ermittelten Werte einmal nicht stimmen sollten (Abb. 3).

Wichtig ist die Methode *finish()*: Diese wird vor Verlassen des Wizard



aufgerufen, und soll zum Speichern der eingegebenen Werte genutzt werden. Das Unisono-Plug-in speichert diese Werte in einem eigenen Preference-Store. Kommt der Aufruf von *finish()*

mit *true* zurück, wird die Methode *getSelection()* aufgerufen. Hier erzeugen wir unseren Container. Der Aufruf von *JavaCore.newContainerEntry()* führt in der weiteren Folge zu unserem Initializer und dadurch zum Konstruktoren unseres Containers. Hier wird das Erstellen der eigentlichen Entries an eine Implementierung delegiert, die die grundlegenden Daten, den Pfad zur *projectbuild.properties* und den Wert für *pce.lib.dir* aus dem PreferenceStore übernimmt.

Die eigentliche Arbeit

Doch wie erfolgt nun die Umwandlung der Einträge aus der *projectbuild.properties* in Classpath Entries des Containers? Dazu bedarf es eigentlich nur der Auflösung der Variablen, die in dieser Datei benutzt und außerhalb der IDE vom ANT-Build-Prozess der Gesamtanwendung aufgelöst werden. Und warum das Rad neu erfinden? Auch in unserem Plug-in können wir diesen Mechanismus nutzen. Wir bilden ihn mithilfe der ANT-API nach und lassen ANT einfach auch innerhalb unseres Plug-ins das Gleiche tun. Listing 5 demonstriert die wichtigsten Schritte: Wir erzeugen ein ANT-Projekt und setzen die vom Benutzer erfragten Werte für die „festen Punkte“ im PC/E-Universum als Properties. Anschließend werden zwei weitere Dateien in unser ANT-Projekt eingebunden, die der Auflösung von Properties wie *{jakarta-oro.jarfile}* dienen. Wenn nun mit *antProject.getProperty("classpath")* der Wert aus der *projectbuild.properties*-Datei erfragt wird, wurden alle symbolischen Namen in dieser Liste von ANT bereits aufgelöst und wir erhalten absolute Pfadangaben auf unsere benötigten jar-Files zurück.

Wir können nun über die einzelnen Tokens im Classpath iterieren und für jedes einen neuen Classpath Entry vom Typ *Library* erzeugen. Damit füllen wir dann die Liste der Classpath Entries unseres Containers.

```
org.eclipse.core.runtime.Path lib = new Path(tokenizer.  
nextToken().trim());  
entries[i++] = JavaCore.newLibraryEntry(lib,  
sourceAttachment, null, true);
```

Noch eine kurze Anmerkung zu den *sourceAttachments*: Diese dienen dazu, referenzierten binären jar-Files mit ihren originalen Quelldateien zu verlinken.

Listing 4

```
public PCEContainerPage()
{
    super("PCEBuildContainer");
    setTitle("PC/E Build Container");
    setDescription("Please define your settings");
    setImageDescriptor(JavaPluginImages.  
DESC_WIZBAN_ADD_LIBRARY);
}

public void initialize(IJavaProject project,  
    IClasspathEntry[] currentEntries)
{
    if (project != null)
    {
        this.javaProject = project;
        IWorkspaceRoot myWorkspaceRoot =  
            ResourcesPlugin.getWorkspace().getRoot();
        IProject myProject = myWorkspaceRoot.  
            getProject(project.getElementName());
        projectRootDir = myProject.getLocation();
    }
}

public void createControl(Composite parent)
{
    // create the SWT controls
}
```

```
}

/**
 * Called, if button finish is clicked on wizard page
 */
public IClasspathEntry getSelection()
{
    return JavaCore.newContainerEntry(PCEContainer.ID,  
true);
}

public boolean finish()
{
    UnisonoPlugin.getDefault().getPreferenceStore().  
setValue(javaProject.getElementName(),  
tfProjectBuildPropsPath.getText());

    UnisonoPlugin.getDefault().getPreferenceStore().  
setValue(javaProject.getElementName() +  
"pceLibDir", tfPceLibDir.getText());

    UnisonoPlugin.getDefault().getPreferenceStore().  
save();

    return true;
}
```



Dadurch kann sich der Entwickler auch die Implementierungen dieser referenzierten Archive im Sourcecode ansehen. Um auch bei der Benutzung von Classpath-Containern nicht auf den Komfort von *sourceAttachments* für binäre *jar*-Files verzichten zu müssen, können alle Classpath Entries vom Typ *Library* mit einer Pfadangabe erzeugt werden, die auf den Java-Quellcode des *jar*-Files verweist. Diese Information, d.h. das Paar aus *jarfile* und Source-Verzeichnis innerhalb der Sandbox, ist natürlich bekannt, relativ statisch und wurde daher in eine *properties*-Datei ausgelagert, die dem Plug-in beiliegt. Diese wurde mit einem kleinen Skript initial erstellt und wird regelmäßig aktualisiert. Diese Aktualisierungen erscheinen relativ wichtig, da sich schon bei der kleinsten Abweichung merkwürdige Fehler bei der Eclipse-internen Suche nach den *sourceAttachments* ergaben.

Änderungen überwachen

Schon bald nach der Einführung des Unisono-Plug-ins kam der Wunsch auf,

Änderungen an der zugrunde liegenden *projectbuild.properties*-Datei sofort im Classpath zu reflektieren. Bis dahin musste zuerst die Referenz auf den Container gelöscht und danach wieder neu angelegt werden. Aber auch für diese Problemstellung gibt es in Eclipse bereits eine Lösung: den *ResourceChangeListener*. Dieser ist beim Workspace zu re-

Im Handumdrehen
sind die fremden
Eclipse-Projekte
kompilierfähig.

gistrieren und muss in seiner Implementierung dem Interface *org.eclipse.core.resources.IResourceChangeListener* genügen, was uns zu einer Methode *public void resourceChanged(IResourceChangeEvent event)* führt. Eclipse nutzt eine Resource-Delta-Hierarchie, um effizient Änderungen zwischen zwei Zeitpunkten propagieren zu können. Da wir uns konkret nur für eine Datei interessieren, wird eine *convenience*-Methode

findMember(IPath path) genutzt, um nicht den ganzen Baum manuell traversieren zu müssen. Hat es Änderungen an genau dieser Datei gegeben, ermitteln wir wieder die Classpath Entries für unseren Container und aktualisieren mit diesem unser Eclipse-Java-Projekt. Listing 6 zeigt die ganze Implementierung.

Fazit

Mithilfe von Classpath-Containern ist es sehr effektiv möglich, Classpath-Informationen aus externen Quellen für Eclipse-Java-Projekte zur Verfügung zu stellen. Die Informationen über den Classpath müssen nur an einer Stelle gepflegt werden und sind in der IDE automatisch immer aktuell. Fehler beim Build-Prozess werden minimiert. Darüber hinaus ergibt sich ein weiterer, nicht zu unterschätzender Vorteil: Da die Eclipse-Projekte vieler Komponenten wiederum andere Komponentenprojekte referenzieren, müssen unter Umständen zig Projekte importiert werden, um eine fremde Funktionalität mindestens kompilieren zu können. Das setzt dann auch voraus, dass man gerade den richtigen, also kompilierfähigen, Stand ausgecheckt hat. Bei rund 100 Komponenten manchmal ein hoffnungsloses Unterfangen. Mithilfe des Classpath-Containers ist dies nicht mehr notwendig. Man stellt ein gewünschtes Eclipse-Java-Projekt auf das Referenzieren von *jar*-Files um. Die dazu notwendigen Archive werden vom Container automatisch ermittelt. Hat man sie nicht lokal vorrätig, müssen sie zuvor nur noch vom zentralen Komponentenverzeichnis kopiert werden. Und im Handumdrehen sind die fremden Eclipse-Projekte kompilierfähig.

Listing 5

```
Project antProject = new Project();
antProject.init();
antProject.setProperty("root.dir", sandboxDir);
antProject.setProperty("pce.lib.dir", pceLibDir);
antProject.setProperty("extern.lib.dir", externLibDir);

// DailyBuild/pcebuild.properties
Property pcebuildProperties = new Property();
pcebuildProperties.setFile(new File(PFAD_ZU_
                                pcebuild.properties));
pcebuildProperties.setProject(antProject);
pcebuildProperties.execute();

// DailyBuild/projectHomesAndExternallibs.
properties
Property projectHomes = new Property();
projectHomes.setFile(new File(PFAD_ZU_
                                projectHomes.properties));
projectHomes.setProject(antProject);
projectHomes.execute();

// projectbuild.properties
Property projectbuildProperties = new Property();
projectbuildProperties.setFile(new File
                                (projectProps));
projectbuildProperties.setProject(antProject);
projectbuildProperties.execute();

String classpath = antProject.
getProperty("classpath");
```

Listing 6

```
public void resourceChanged(IResourceChangeEvent
                           event)
{
    if (event.getDelta() != null)
    {
        IResourceDelta delta = event.getDelta().
            findMember(projectBuildProps);
        if (delta != null)
        {
            this.entries = Model.getInstance().
                getPCEClasspathEntries(javaProject);
            try
            {
                JavaCore.setClasspathContainer(ID,
                    new IJavaProject[] {javaProject}, new
                    IClasspathContainer[] {this}, null);
                javaProject.setRawClasspath(javaProject.
                    getRawClasspath(), null);
            }
            catch (JavaModelException e)
            {
                logger.error(e.getMessage());
            }
        }
    }
}
```



Dominik Hirt ist Diplom-Informatiker und bereits seit 1995 Java-Entwickler. Er arbeitet als Software Engineer bei der Wincor Nixdorf AG. Dort ist er für Softwarekomponenten im Rahmen der PC/E Suite verantwortlich. Darüber hinaus gilt sein Interesse praktischen Verbesserungen am Softwareentwicklungsprozess und Model-driven Software Development.
Kontakt: dominik.hirt@wincor-nixdorf.com.

>> Links & Literatur

- [1] Andrew Hunt, David Thomas:
Der Pragmatische Programmierer.
- [2] ant4eclipse.sourceforge.net