Redundanzen in der Projektkonfiguration zwischen Ant und Eclipse minimieren

Bei mir funktioniert's ...

VON NILS HARTMANN UND GERD WÜTHERICH

Es ist eine typische Situation: Das Softwareprodukt steht kurz vor der Fertigstellung, die Entwickler checken ihre letzten Änderungen in die Versionsverwaltung ein und der finale Build-Lauf kann beginnen. Da der Build automatisiert außerhalb von Eclipse ausgeführt werden soll, ist er in Ant implementiert. Quasi unter Quarantäne wird er auf einem eigenen Rechner regelmäßig ohne Eingriffe von außen ausgeführt und liefert zur Auslieferung fertige Artefakte. Das alles hat bereits diverse Male reibungslos geklappt und so macht sich der Verantwortliche für den Build-Prozess auch kurz vor der entscheidenden Auslieferung keine Gedanken, dass es diesmal anders verlaufen könnte. Doch dann meldet Ant „Build failed …".

Bei dem oben geschilderten Szenario wird sich beim Build-Verantwortlichen – nennen wir ihn Rob van der Built – eine leichte Unruhe bereitmachen. Er schaut in das Changelog von CVS und fragt bei den Entwicklern nach. „Bei mir kompiliert alles", „ja, ich habe alles neu ausgecheckt, und bei mir klappt es, vielleicht hat jemand anderes ..." – die Antworten der Entwickler sind Rob wohlbekannt. Ein vergessenes Commit scheidet als Problem aus. Im Gegenteil: Bei neuerlichem Hinsehen stellt Rob van der Built fest, dass ein Kollege einen neuen XML-Parser eingecheckt hat. Dieser aber taucht in den Build Scripts nicht auf, da diese in der Hoheit von Rob liegen und von den Entwicklern nicht bearbeitet werden. Kein Problem für Rob: Er schaut in die *.classpath*-Dateien jedes für den Build benötigten Eclipse-Projektes nach, ermittelt die geänderten Abhängigkeiten und aktualisiert den Classpath in seinen Ant-Skripten. Danach versioniert er die Build Scripts, kopiert sie auf den Build-Rechner und startet einen neuerlichen Build-Lauf. „Build successful" erscheint eine halbe Stunde später und nachdem auch die Tests durchgelaufen sind, beginnt die Lava-Lampe des Extreme Feedback Device [1] grünlich zu blubbern. Die Welt von Rob van der Built und seinen Kollegen aus der Anwendungsentwicklung ist wieder in Ordnung – fast.

Denn in der S-Bahn auf dem Nachhauseweg beschleicht Rob ein ungutes Gefühl: „Was ist, wenn die Entwickler die Klassenpfad-Einstellungen erneut ändern? Wenn sie zum Softwareprodukt neue Eclipse-Projekte hinzufügen? Oder bestehende entfernen?" Drei Stationen später begreift Rob, dass bislang alle Projektkonfigurationen doppelt gepflegt werden müssen: Da sind die Projekteinstellungen, die in Eclipse gesetzt werden. Die gleichen Informationen verwaltet Rob aber auch in seinen Ant-Skripten! Noch bevor Rob an der Station „Reeperbahn" aussteigt, fasst er einen Entschluss: Für die Zukunft wird er dafür sorgen, dass es möglichst keine Diskrepanzen von Klassenpfaden und anderen Konfigurationen zwischen den Umgebungen der Entwickler (Eclipse) und seiner Build-Umgebung (Ant) mehr gibt. „Redundanzen in der Projektkonfiguration zwischen Ant und Eclipse minimieren", schreibt er in die To-do-Liste seines Pocket-PCs für den nächsten Tag.

Klassenpfade von Eclipse-Projekten auslesen

Als Rob am nächsten Morgen sein Vorhaben in die Tat umzusetzen beginnt, muss er zunächst eine grundsätzliche Frage beantworten: Welches der eingesetzten Tools ist das „führende" Werkzeug – Eclipse oder Ant? Welches der beiden Werkzeuge gibt die Einstellungen vor und welches übernimmt sie nur? Die Antwort ist für Rob schnell klar: Da die Entwickler nicht gewillt sind, statt des inkrementellen Compilers, der grafischen Launcher und anderen Eclipse-Tools die von Rob betreuten Ant-Skripte zu nutzen, sieht er sich gezwungen, seine Ant-Skripte derart umzubauen, dass sie in der Lage sind, die Konfigurationen von Eclipse auszulesen.

Dabei fallen ihm zunächst die gestrigen Classpath-Unterschiede wieder ein. Aus der manuellen Analyse des Klassenpfades in der Vergangenheit kennt Rob die technische Implementierung in Eclipse: Der Klassenpfad eines Java-Eclipse-Projektes wird in der *.classpath*-Datei beschrieben, die sich im Root-Verzeichnis eines jeden Projektes befindet. Es handelt sich dabei um eine einfache XML-Datei, die die einzelnen Klassenpfadeinträge beschreibt. Dabei kann ein Klassenpfadeintrag z.B. eine *.jar*-Datei oder ein (Classes-)Verzeichnis aus dem aktuellen Projekt sein. Ein Eintrag kann aber auch ein anderes Eclipse-Projekt aus dem gleichen Workspace referenzieren. An dieser Stelle wird dann der

Klassenpfad des referenzierten Eclipse-Projektes eingebunden. Diese Eigenschaft ist rekursiv, sodass ein referenziertes Eclipse-Projekt (bzw. dessen Klassenpfad) ebenfalls auf weitere Projekte verweisen kann. Hinzu kommt, dass ein Klassenpfadeintrag exportiert sein kann. In diesem Fall ist er für den Compiler auch in Projekten sichtbar, die dieses Projekt referenzieren. Ist er nicht exportiert, wird er nur von seinem Projekt erkannt.

Rob benötigt also zunächst einen Ant-Task, der ihm die Klassenpfaddefinitionen aus Eclipse einliest und für Ant aufbereitet. Bevor Rob beginnt, eigene Ant-Tasks zu schreiben, sucht er im Internet, ob nicht bereits solche Tasks existieren – möglichst natürlich als Open-Source-Projekt. Und tatsächlich: Auf Sourceforge.net stößt er auf das Ant4Eclipse-Projekt [2], welches unter anderem einen *getEclipseClasspath*-Task enthält. Diesem Task werden ein Pfad zu einem Workspace sowie der Name eines Projektes übergeben. Der Task liest dann die entsprechende Classpath-Datei ein und baut daraus ein Ant-Classpath-Objekt auf. Auf dieses Objekt kann Rob in seinen Skripten mittels des classpath-ref-Attributes überall dort zugreifen, wo ansonsten sein manuell erstellter Classpath genutzt würde. Nachdem sich Rob die aktuellen Produkt-Sourcen ausgecheckt hat, erstellt er einen ersten Entwurf einer *build.xml*-File (Listing 1), die den Classpath für ein Eclipse-Projekt einliest und das Projekt kompiliert.

Builds in Abhängigkeit von unterschiedlichen Projekttypen

Rob weiß, dass nicht alle Projekte einfach nur kompiliert werden müssen. Einige der Projekte brauchen bspw. einen abschließenden *rmic*-Aufruf, andere verlangen die Ausführung eines Code-Generators vor oder nach dem Kompilieren. In Eclipse werden solche zusätzlichen Build-Schritte durch die Definition von sog. Build-Kommandos (*BuildCommand*) realisiert. Build-Kommandos können projektindividuell angegeben werden und sind innerhalb des Eclipse-Projektes in der *.project*-Datei abgelegt. Jedes Build-Kommando ist in Eclipse mit einem Builder assoziiert, der das entsprechende Build-Kommando implementiert bzw. ausführt. So sind bspw. Java-Projekte in Eclipse mit einem Builder vom Typ *org.eclipse.jdt.core.javabuilder* assoziiert.

Um das Verhalten aus Eclipse in seinen Ant-Skripten zu übernehmen, nutzt Rob die *hasBuildCommand*-Condition von Ant4Eclipse. Diese Condition liefert *true* oder *false* zurück, je nachdem, ob das aktuelle Projekt das geforderte Build-Kommando besitzt oder nicht. Für die Auswertung der Abfrage nutzt Rob den *if*-Task aus dem *ant-contrib*-Projekt [3]. Um einige Zeilen erweitert ist sein Build Script jetzt in der Lage, *BuildCommand*-abhängig Build Scripts auszuführen (Listing 2).

In Eclipse können Projekten darüber hinaus so genannte Natures zugewiesen werden, die das Projekt als einen speziellen Typ kennzeichnen. So haben bspw. Java-Projekte in Eclipse den Typ *org.eclipse.jdt.core.javanature*. Auch zur Auswertung der Natures steht in Ant4Eclipse eine entsprechende Condition zur Verfügung.

Am Ende des Tages hat Rob ein sehr schlankes, aber mächtiges Build Script entwickelt, das die Inkonsistenzen zwischen der Ant- und der Eclipse-Umgebung minimiert. Was ihm allerdings noch fehlt, ist die Möglichkeit, sich alle für den Build benötigten Eclipse-Projekte automatisch auszuwählen. Da jedoch die Erstellung des Build Script einige Zeit in Anspruch genommen hat und Rob bereits mehrfach von seinem Vorgesetzten angehalten wurde, die gewerkschaftlich vereinbarte 37-Stunden-Woche einzuhalten, beschließt er, diese Aufgabe am nächsten Tag zu lösen.

Wie werden die Projekte eines Team Project Set ausgecheckt?

Wie bei vielen Softwareprodukten, die mit Eclipse entwickelt werden, besteht auch das von Rob betreute Produkt aus mehreren Eclipse-Projekten. Diese Projekte liegen für gewöhnlich in einem eigenen Verzeichnis, dem *Workspace*. Die einzelnen Projekte sind darüber hinaus im zentralen CVS Repository eingestellt. Da das Produkt von Rob und seinen Kollegen schon etwas länger am Markt besteht, müssen die Entwickler in unterschiedlichen Versionen bzw. Branches arbeiten: So gibt es u.a. einen Branch für die Fehlerbehebung in der aktuell ausgelieferten Version und einen weiteren Branch für die aktuellen Weiterentwicklungen des Produktes. Zu jeder Version bzw. zu jedem Branch des Produktes gehören mehrere Eclipse-Projekte in ganz bestimmten Versionen bzw. Branches. Um diese Versionsinformationen zu verwalten, nutzen Robs Kollegen die Team Project Sets von Eclipse. Ein Team Project Set enthält eine Liste mit Projekten und deren Versionen im CVS. Für jede relevante Produktversion gibt es eine Team Project Set-Datei, die Aufschluss darüber liefert, welche Projektversionen in diesem Produkt eingesetzt werden.

Um in ihrem lokalen Workspace von einer Version zur anderen zu wechseln, nutzen Robs Kollegen das Team Project Set Plug-in [4]. Dieses Plug-in vergleicht den Inhalt eines Team Project Set mit den Projekten, die sich zurzeit im Workspace befinden, und stellt die Unterschiede in visueller Form dar. So kann ein Entwickler leicht erkennen, welche Projektversionen

Listing 1

```
<target name="compile-project">

 <!-- Einlesen des Eclipse-Classpath -->
 <getEcliseClasspath workspace="${basedir}/.."
  projectName="myproject"
  classpathId="build.classpath"/>

 <-- Kompilieren -->
 <javac srcdir="source"
  destdir="classes"
  classpathref="build.classpath"/>

</target>
```

Listing 2

```
<target name=" compile-project ">
 ...
 <if>
  <hasBuildCommand workspace="${basedir}/.."
   projectName="myproject"
   buildCommand ="org.mycompany.rmibuilder" />
 </if>
 <then>
  <!-- rmi-projekt -->
  <rmic base="classes"
   classpathref="build.classpath"
   includes="classes/**/Remote*.class"/>
 </then>

</target>
```

im Workspace von denen im Team Project Set abweichen und seine Projekte entsprechend aktualisieren. Wenn ein Kollege von Rob für eine bestimmte Produktversion entwickeln möchte, öffnet er mit dem Plug-in die entsprechende Team Project Set-Datei und tauscht gezielt die Projekte aus, die nicht in der richtigen Version in seinem Workspace vorliegen. Um sich die Arbeit noch weiter zu erleichtern, werden die Team Project Set-Dateien ebenfalls ins CVS eingecheckt, sodass alle Entwickler jederzeit Zugriff auf sie haben.

Rob sieht, dass er ebenfalls von den Team Project Sets profitieren könnte, wenn er in der Lage wäre, diese mittels Ant zu verarbeiten. Im Ant4Eclipse-Projekt findet er den *cvsGetProjectSet*-Task, der genau diese Arbeit für ihn erledigt. Diesem Task braucht er lediglich den Namen einer Project Set-Datei und ein Zielverzeichnis zu übergeben. Der Task checkt dann alle in der Project Set-Datei angegebenen Projekte – in den entsprechenden Versionen – in das Zielverzeichnis aus. Rob kann sich auf diese Art und Weise einen Ant Workspace nachbauen.

Listing 3

```
<target name="build-all">
 ...
 <!-- Ermitteln der Build-Reihenfolge -->
 <getBuildOrder workspace="${basedir}/.."
  projectSet="${basedir}/../projectsets/
  myprojectset.psf" buildorderProperty="buildorder"/>

 <!-- Aufruf des compile-Targets für jedes einzelne
                                        Projekt -->
 <foreach list="buildorder"
  target="compile-project"
  param="project.name"/>

</target>

<target name="compile-project"
 <!-- kompiliert das Projekt ${project.name} -->
 ...
</target>
```

Listing 4

```
<target name="launch">

 <!--Starten einer Applikation -->
 <launch workspace="${basedir}/.."
  launchFile="${basedir}/../myProject/
                          Applikation.launch" />

 <!--Starten eines JUnit-Tests -->
 <launchjunit workspace="${basedir}/.."
  launchFile="${basedir}/../myProject/
                          AllJUnitTests.launch" >
  <formatter type="xml" usefile="false"/>
 </launchjunit>

</target>
```

```
<target name="build-all">

 <!-- Erstellen des Workspaces -->
 <cvsGetProjectSet workspace="${basedir}/.."
  projectSet="${basedir}/../projectsets/
                  myprojectset.psf" command="checkout"/>
 ...
</target>
```

Die richtige Build-Reihenfolge eines Team Project Set berechnen

Zusammen mit seinen gestrigen Build Scripts ist Rob nun in der Lage, einen kompletten Workspace auszuchecken und einzelne Projekte darin zu bauen. Was ihm allerdings zur vollständigen Automatisierung eines kompletten Build-Laufs fehlt, ist das Wissen über die Reihenfolge, in der die Projekte kompiliert werden müssen: Zunächst sollen die Projekte gebaut werden, die keine Abhängigkeiten zu anderen Projekten besitzen. Danach sollen sukzessive die Projekte gebaut werden, die nur von bereits gebauten Projekten abhängig sind.

Um die notwendige Build-Reihenfolge zu ermitteln, benutzt Rob van der Built den *getBuildOrder*-Task, der für einen Workspace die erforderliche Build-Reihenfolge berechnet. Die Projektnamen werden in der Reihenfolge, in der sie gebaut werden müssen, in ein angegebenes Ant Property geschrieben, auf das Rob in seinem Build Script zugreifen kann. Über den *foreach*-Task aus dem *antcontrib*-Projekt kann auf diese Liste zugegriffen werden. Dieser Task ruft für jeden Eintrag in der Liste ein angegebenes Ant Target auf. Rob erweitert sein Build Script also erneut, um komplette Workspaces auschecken und bauen zu können (Listing 3).

Am Ende des zweiten Tages mit Ant-4Eclipse kann Rob das von ihm betreute Produkt in den unterschiedlichen Versionen mittels der neuen Tasks auschecken und die zugehörigen Projekte auf Basis der Eclipse-Definitionen bauen. Zufrieden begibt sich Rob in den Feierabend.

Applikationen mithilfe von Eclipse-Launch-Konfigurationen starten

Der nächste Arbeitstag beginnt für Rob mit einer Bestandsaufnahme: Das Bauen eines einzelnen Projektes auf Basis der Eclipse-Konfigurationen ist realisiert. Ebenfalls implementiert ist das Auschecken und Bauen von ganzen Produkten, die aus mehreren Projekten bestehen. Was Rob noch zu tun bleibt, ist die Anpassung der Testausführung: Im nächtlichen Build wird nämlich zusätzlich zu den einfachen Unit-Tests ein Integrationstest ausgeführt, für den die komplette Applikation gestartet wird. Innerhalb von Eclipse ist das Starten der Anwendung denkbar einfach: Für jede Applikation stehen so genannte Java Application Launch Configurations zur Verfügung. Eine Launch-Konfiguration beschreibt, wie eine Applikation bzw. ein JUnit-Test gestartet werden sollen. Neben den Klassenpfadeinstellungen können bspw. auch Kommandozeilen- oder Systemparameter gesetzt werden. Und damit nicht jeder Entwickler eigene Launch-Konfigurationen definieren muss, sind auch die Launch-Konfigurationen in die Versionsverwaltung eingestellt und damit fester Bestandteil der von Rob betreuten Projekte.

Um die Redundanzen zwischen Eclipse- und Ant-Konfigurationen noch weiter zu minimieren, überlegt Rob, ob es nicht möglich wäre, die Launch-Konfigurationen innerhalb des Ant-Builds auszulesen. Und auch das – es mag den Leser kaum verwundern – ist mithilfe des *launch*-Tasks aus Ant4Eclipse möglich.

Der *launch*-Task ist eine Erweiterung des *Java*-Task von Ant, dem aber nicht alle zum Start einer Java-Anwendung notwendigen Parameter einzeln übergeben werden. Es reicht aus, beim Aufruf des *launch*-Tasks eine Eclipse-Launch-Konfiguration zu spezifizieren. Die zum Start einer Applikation notwendigen Parameter wie bspw. die Klassenpfadeinstellungen oder Kommandozeilenparameter werden aus der angegebenen Konfiguration ausgelesen. Zum Starten von JUnit-Tests kann

der spezialisierte *launchjunit*-Task verwendet werden. Rob kann damit auf die Nutzung des *junit*-Tasks von Ant verzichten und die von den Entwicklern ohnehin erstellten und gepflegten JUnit-Konfigurationen aus Eclipse nutzen (Listing 4).

Alternativen zu Ant4Eclipse

Nachdem Rob seine neuen Skripte entwickelt, getestet und schließlich auch eingeführt hat, kann er sich bequem zurücklehnen und beschließt, einige Tage Urlaub zu machen. Er weiß, dass seine Skripte alle neuen Bibliotheken etc. korrekt mit in den Classpath aufnehmen. Er weiß auch, dass die Anwendungen und JUnit-Tests, die aus dem Build heraus gestartet werden, mit den richtigen Laufzeitklassenpfaden und Kommandozeilenargumenten versehen sind. Zufrieden genießt Rob die wohlverdienten freien Tage.

Für uns stellt sich allerdings die Frage, ob Rob nicht auch andere Ansätze zur Minimierung der Redundanzen in den Projektkonfigurationen hätte wählen können. In der Tat: Wie so oft im Leben führen auch hier mehrere Wege zu dem gewünschten Ziel. Mindestens zwei Alternativen – beide mit ganz anderen konzeptionellen Ansätzen – bleiben hier nicht unerwähnt.

Die erste Alternative, die Rob hätte wählen können, ist die automatische Generierung von Ant-Build-Skripten aus Eclipse heraus. Mithilfe des *eclipse2ant*-Plug-ins [5] kann für jedes Projekt ein individuelles Build-File auf Basis der Eclipse-Projekteinstellungen erzeugt werden. Mithilfe dieses Build Script können Projekte auch außerhalb von Eclipse gebaut werden. Die Generierung der Build Scripts erfolgt über einen Exportfilter direkt aus Eclipse heraus. Seit der Eclipse-Version 3.1 M6 ist das Plug-in zur Generierung von Ant-Build-Skripten Bestandteil des offiziellen Eclipse-Releases.

Eine zweite Alternative, die allerdings weit reichendere Konsequenzen für den gesamten Build-Prozess von Rob und seine Kollegen hätte, ist die Umstellung auf Maven [6]. Bei der Verwendung von Maven wird die Definition der Projekteigenschaften nicht länger in Eclipse vorgenommen, sondern innerhalb von Maven. Damit verliert Eclipse seine Funktion als führendes System. Mithilfe des Maven-Eclipse-Plug-in können aus der Maven-Projekt-Beschreibung *.classpath*- und *.project*-Dateien für Eclipse erzeugt werden. Der Build außerhalb von Eclipse findet dann nicht mehr mit Ant, sondern mit Maven statt.

Fazit

Mit Ant4Eclipse steht eine leistungsfähige Sammlung von Ant-Tasks zur Verfügung, mit der sich Eclipse-Projekt-Informationen in Ant-Skripten auslesen und verarbeiten lassen. Unterstützt werden das Bauen und das Ausführen von Applikationen, die als Eclipse-Projekte vorliegen. Ob der Ant4Eclipse zugrunde liegende Ansatz, Eclipse bei der Definition der Projekteigenschaften als das „führende" System zu definieren, im konkreten Einzelfall sinnvoll und praktikabel ist, muss individuell entschieden werden. Ist dies jedoch der Fall, dann stellt Ant4Eclipse ein sehr brauchbares Werkzeug zur Minimierung der Redundanzen in der Projektdefinition zwischen Ant und Eclipse dar.

Nils Hartmann arbeitet als Softwareentwickler in Hamburg. Er beschäftigt sich dort u.a. mit der Optimierung von Entwicklungs- und Build-Prozessen von Java EE-Anwendungen.
Gerd Wütherich arbeitet als Softwarearchitekt und Softwareentwickler im Java EE-Umfeld. Zu seinen Interessen zählt u.a. die effiziente Gestaltung von Build- und Anwendungsentwicklungsprozessen.

Links & Literatur

[1] www.developertesting.com/archives/month200404/20040401-eXtremeFeedbackForSoftwareDevelopment.html
[2] ant4eclipse.sourceforge.net
[3] ant-contrib.sourceforge.net
[4] www.ejbprovider.de
[5] www.geocities.com/richard_hoefter/eclipse2ant/
[6] maven.apache.org