Versionierung von SOA Projekten mit Subversion im JDeveloper

Klaus Friemelt
MT AG
Ratingen

Schlüsselworte: Subversion, JDeveloper11g, Versionierung, SOA-Projekte

Einleitung

SOA-Projekte (und ebenso ADF-Projekte) erfordern -speziell bei größeren Entwicklerteams- mit ihren zahlreichen komplexen Artefakten den Einsatz eines Versionskontrollsystems. Der JDeveloper 11g als zentrales Entwicklungswerkzeug für die Oracle-basierte SOA-Welt unterstützt dies in vielfältiger Weise.

Hier stehen unter anderem CVS, IBM ClearCase, Serena Dimensions und Subversion zur Auswahl.

IBM ClearCase und Serena Dimensions bieten als SCM-Systeme (Software Configuration Management) deutlich mehr als eine schlichte Versionskontrolle, was sich allerdings auch in Komplexität und Lizenzkosten niederschlägt. CVS (Concurrent Versions System) ist letztlich veraltet und hat in Subversion seinen Nachfolger gefunden, der auch für zahlreiche Plattformen verfügbar ist. Durch diese hohe Verbreitung und geringe Kosten bietet sich zunächst Subversion als Lösung an.

Es wird zunächst das allgemeine Einrichten der Versionskontrolle im JDeveloper 11g erläutert. Des Weiteren wird gezeigt, welche Möglichkeiten Subversion für den Entwicklungsprozess in SOA-Projekten bietet.

Kurzüberblick über Versionskontrollsysteme

Die Versionskontrolle soll einen konsistenten Stand aller Quellcodes eines Softwaresystems sicherstellen. Technisch bestehen die meisten Systeme üblicherweise aus je einer Client- und Server-Komponente. Ausnahme wäre beispielsweise der früher sehr verbreitete PVCS Version Manager (nur Clients+Netzlaufwerk).

Abhängig von den Artefakten sind dabei zwei Ansätze sinnvoll. Für quasi binäre Quellformate wie z.B. *.fmb oder *.rdf (für Freunde der Oracle Traditional Tools) ist der Ansatz „Lock-Modify-Write“ sinnvoll. Hier werden die Objekte aus dem zentralen Repository in die lokale Arbeitsumgebung geschrieben und gleichzeitig eine Schreibsperre im Repository gesetzt. Für die Dauer der Bearbeitung können andere Entwickler diese Objekte nicht verändern (bzw. keine Version davon einchecken). Nach Abschluss der Bearbeitung werden die Objekte mit erhöhter Versionsnummer zurück in das Repository geschrieben.

Der alternative Ansatz läst sich durch „Copy-Modify-Merge“ beschreiben. Hier werden rein textbasierte Quellen wie *.java, *.c als Delta archiviert. Entwickler können zeitgleich die gleiche Quelle aus dem Repository in ihren Arbeitsbereich auschecken und bearbeiten, beim Zurückschreiben werden gegebenenfalls parallele Versionen der Quelle automatisch zusammengeführt („Merge“) oder Konflikte zur manuellen Auflösung angezeigt.

Parallele Entwicklungszweige („Branch“) werden in beiden Ansätzen unterstützt.

Mischformen der beiden Ansätze sind hier durchaus üblich. Auch Subversion als „Copy-Modify-Merge“-Vertreter unterstützt das Locking von Quellen.

Im JDeveloper 11g ist Subversion bereits als Versionskontrollsystem voreingestellt. Damit hat man die Möglichkeit, ein lokales SVN-Repository anzulegen.

Einrichtung von Versionierungs-Erweiterungen im JDeveloper 11g

Unter dem Menüpunkt „Versioning“ > „Configure...“ sind die verfügbaren Erweiterungen zur Versionisierung gelistet. Um weitere Erweiterungen zu ergänzen, startet unter dem Menüpunkt „Help“ > „Check for Updates...“ ein Assistent. Dieser sollte bekannt sein – schließlich wird der „Oracle SOA Composite Editor“ ja auf dem gleichem Wege integriert.

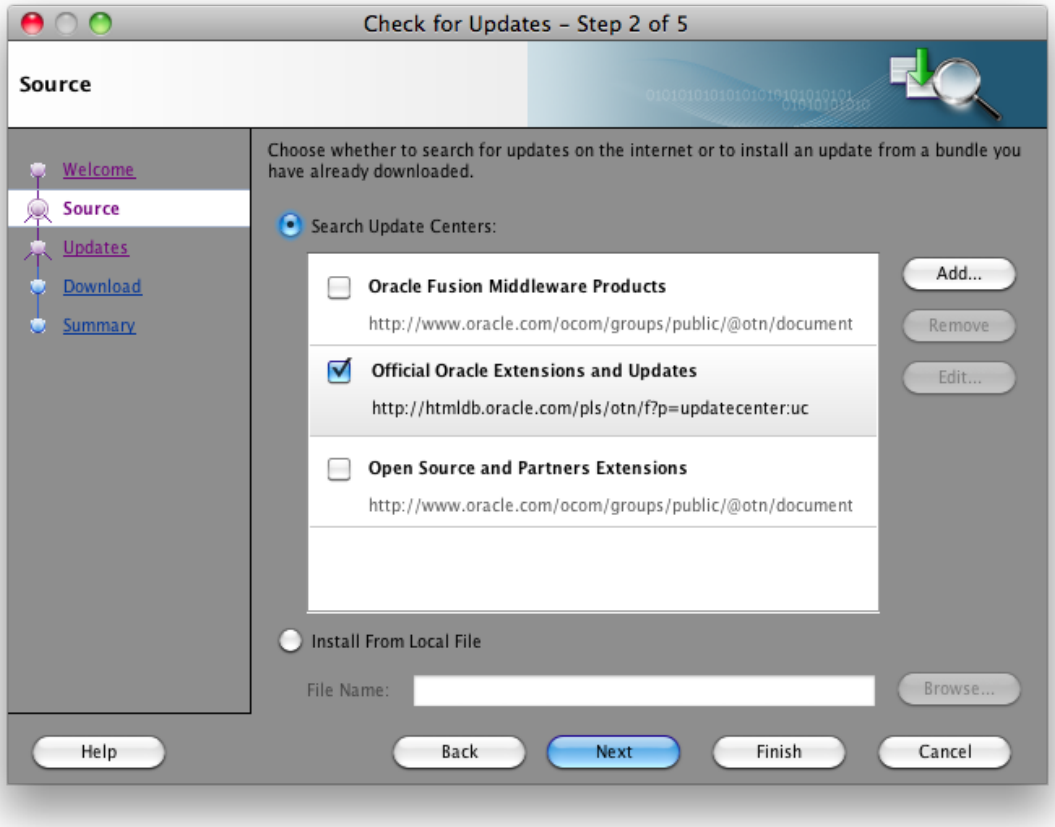

Abb. 1: Check for Updates: Auswahl des Update Centers

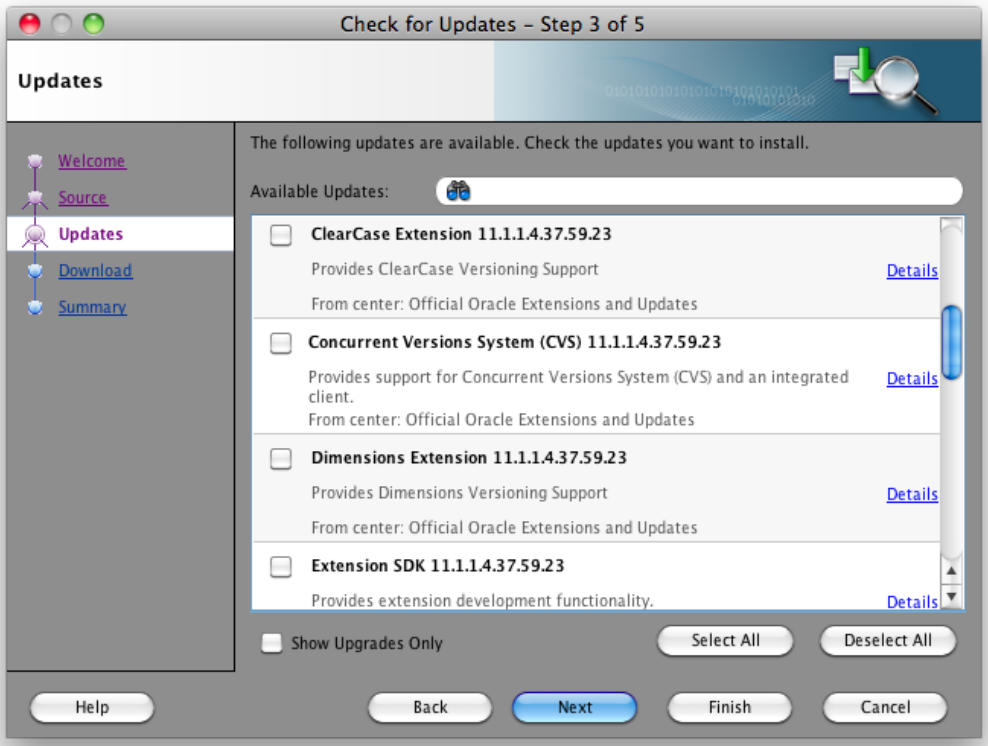

Abb. 2: Check for Updates: Auswahl der Erweiterung

Nach Auswahl der gewünschten Erweiterung und Abschluss des Assistenten stehen nach dem Neustart des JDevelopers die Zugriffe über den Menüpunkt „Versioning“ zur Verfügung.

Einrichtung eines lokalen SVN-Repositories im JDeveloper 11g

Da generell auch für 1-Mann-Teams die Verwendung einer Versionskontrolle sinnvoll sein kann, wird zunächst auf die Einrichtung eines lokalen Repositories eingegangen (auch wenn dieses aus Sicherheits-Gründen vielleicht auf einem Netzlaufwerk mit geregelter Backup-Sicherung liegen sollte). Im Menüpunkt „Versioning“ > „Subversion“ > „Create Local Repository...“ startet ein Dialog zur Eingabe des Speicherorts und des Repository-Namens:

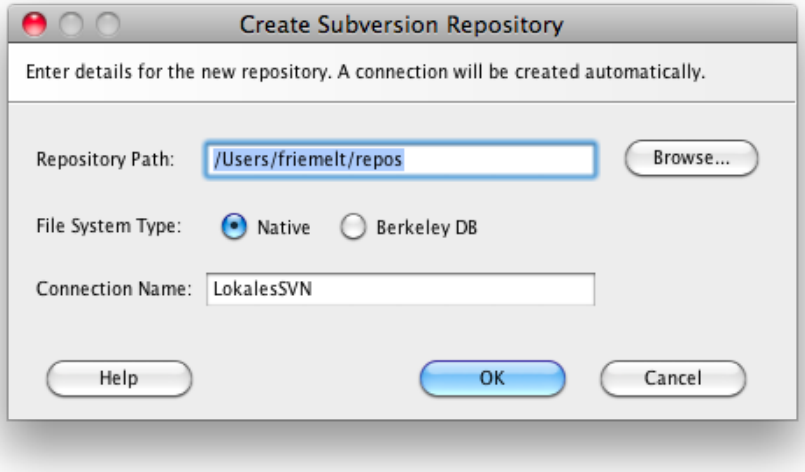

Abb. 3: Anlegen eines lokalen SVN-Repositories

Es wird im Dateisystem im ausgewählten Repository Path eine Ordnerstruktur angelegt, in die man niemals manuell eingreifen sollte.

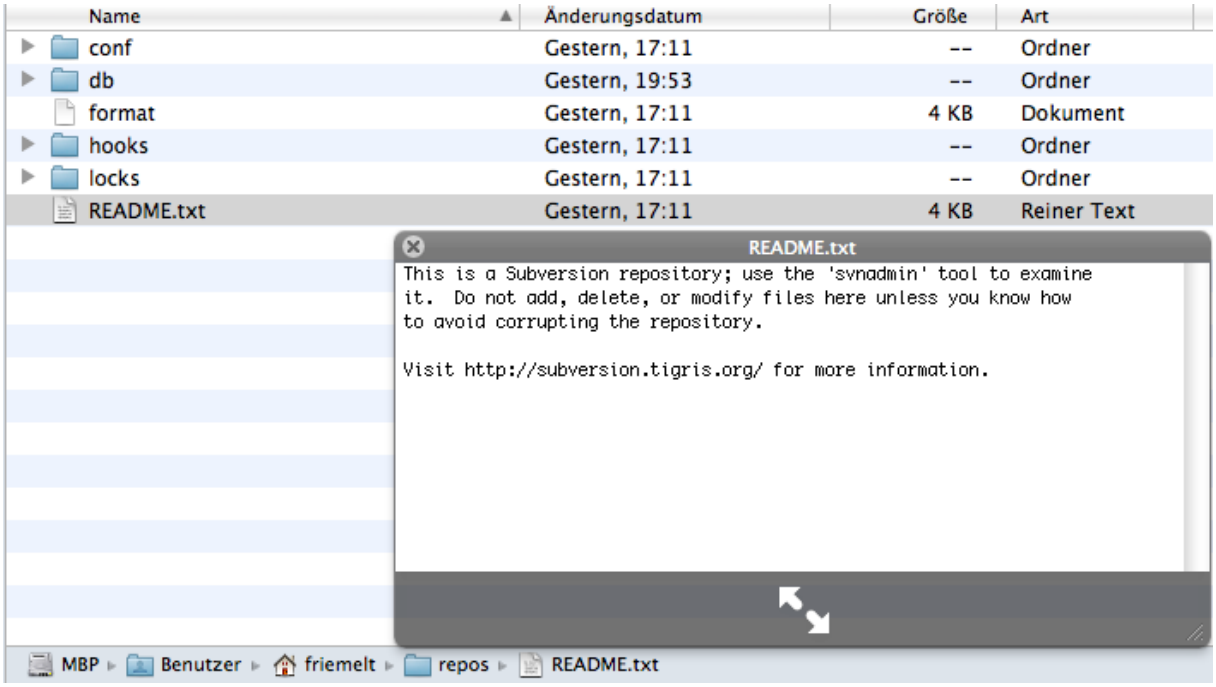

Abb. 4: Dateistruktur des lokalen SVN-Repositories

Einrichtung des Zugriffs auf ein entferntes SVN-Repository im JDeveloper 11g

Den Zugriff auf ein entferntes Repository geschieht über das Anlegen einer Verbindung. Dies geschieht wahlweise über den allgemeinen Weg im JDeveloper, also „File“ > „New“ mit der

Kategorie „Connections“ und Auswahl von „Subversion Repository Connection“ oder direkt im Versioning Navigator:

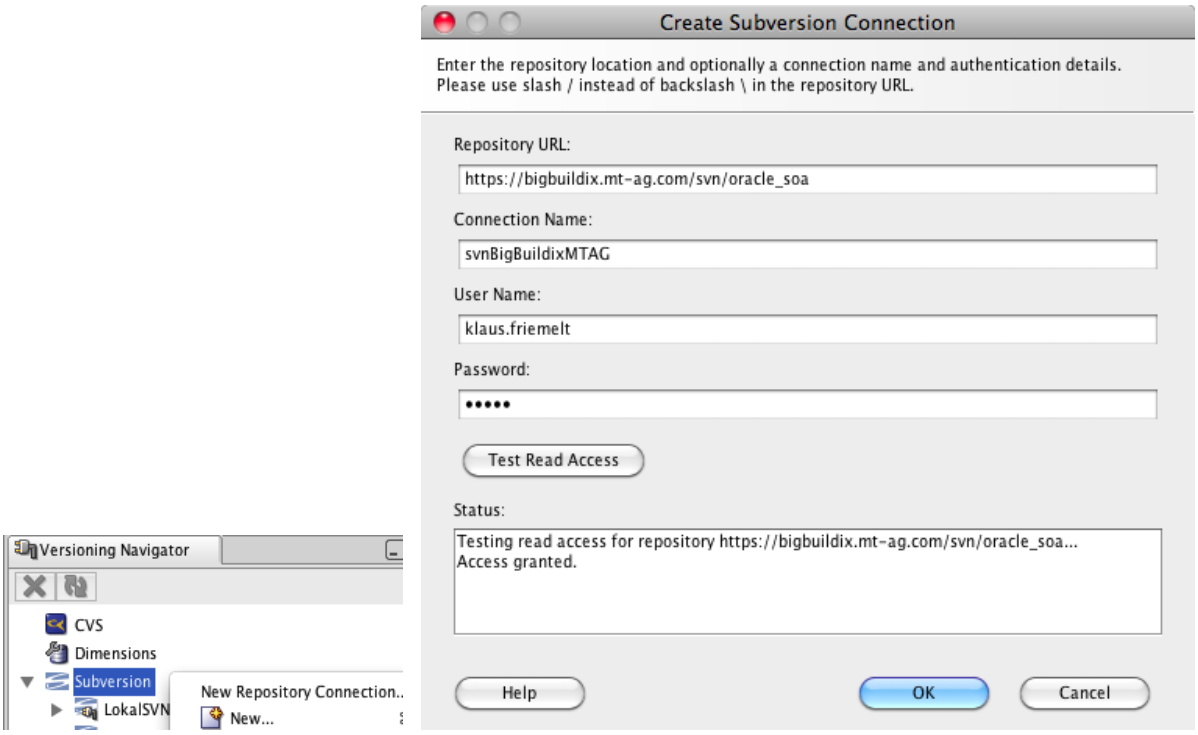

Abb. 5: Anlegen einer entfernten SVN-Verbindung

Mit „New Repository Connection...“ werden die Verbindungsdaten zu einem bestehenden Subversion-Repository abgefragt. Das Beispiel zeigt den Zugriff via https auf ein Subversion-Repository innerhalb eines Buildix-Servers. Dieser läuft über Apache und SSL, damit ist das Repository beispielsweise auch per Browser über Internet erreichbar um Quellen anzusehen.

In diversen Linux-Distributionen, wie beispielsweise dem hier verwendeten Oracle Linux 5.6, ist bereits Subversion enthalten. Bei einem vereinfachten Szenario - ohne Webzugriff und nur mit wenigen Entwicklern, die dann als Betriebssystembenutzer angelegt sind - wäre das Zugriffsprotokoll für die anzugebende Repository-URL dann „svn+ssh“, also z.B. svn+ssh://192.168.244.135/svn.

Projekt in die Versionierung überführen

Unter dem Menüpunkt „Versioning“ > „Version Application...“ lässt sich die aktuelle Applikation in die Versionierung übernehmen. Ein Assistent leitet wieder in mehreren Schritten durch den Vorgang. Zunächst wählt man ein Repository aus der Liste der bestehenden Repository Verbindungen aus. Optional und kann man noch einen neuen Ordner im SVN anlegen.
Im nächsten Schritt gibt man an das Quellverzeichnis der zu importierenden Dateien an. Dies ist mit der aktuellen Applikation vorbelegt, kann hier aber noch geändert werden.

Ganz wichtig, ist es den Importfilter im Schritt 4/6 zurückzusetzen. Der voreingestellte Filter schließt beispielsweise *.class aus. Ein anschließendes Deployment auf den SOA Server schlägt damit fehl! Überführt man beispielsweise eine bereits lokal versionierte Applikation in ein zentrales Repository, wäre der Filter auf „/.svn/“. zu setzen. Dahinter verbergen sich versteckte Ordner, die Subversion zur internen Steuerung überall im Arbeitbereich anlegt.
Der „.DS_Store“-Filter schließt die allgegenwärtigen und normalerweise versteckten Darstellungsoptionen des Finders unter Mac OS X aus.

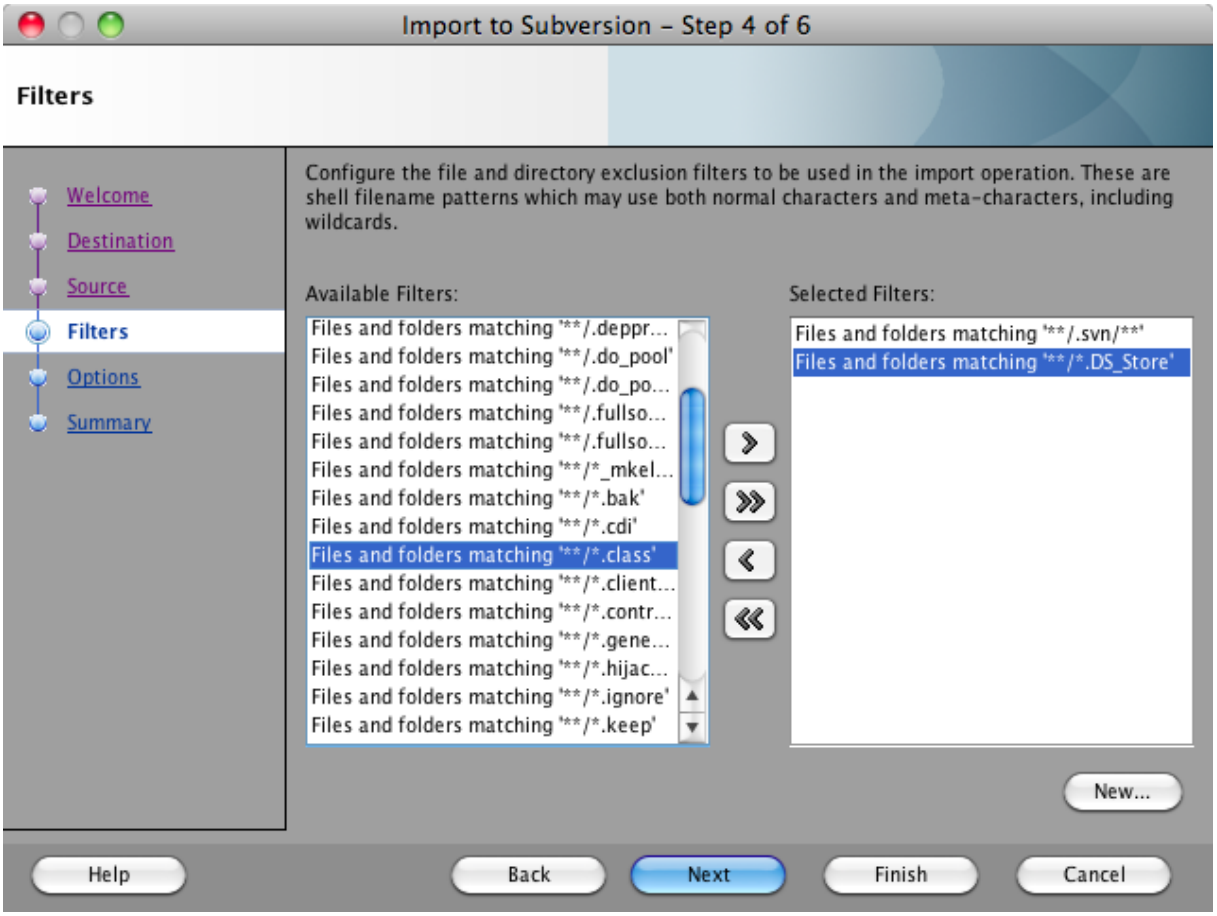

Abb. 6: Filterdialog beim SVN-Import

Im nächsten Schritt kann man die Hierarchietiefe der zu importierenden Dateien beschränken (Checkbox „Do Not Recurse“, diese ist standardmäßig nicht aktiviert und bleibt auch so) sowie nach dem Import einen direkten Checkout anfordern (Checkbox „Perform checkout“, ist standardmäßig aktiviert). Der letzte Schritt gibt als Zusammenfassung eine Übersicht über gewählte Quellen, Zielordner, Filter und Optionen und bietet den Importstart an.

Nach Abschluss des Imports stellen sich die Icons im Application Navigator verändert dar mit kleinen schwarzen Kreisen, um zu zeigen, dass es sich um ein versioniertes Projekt handelt.

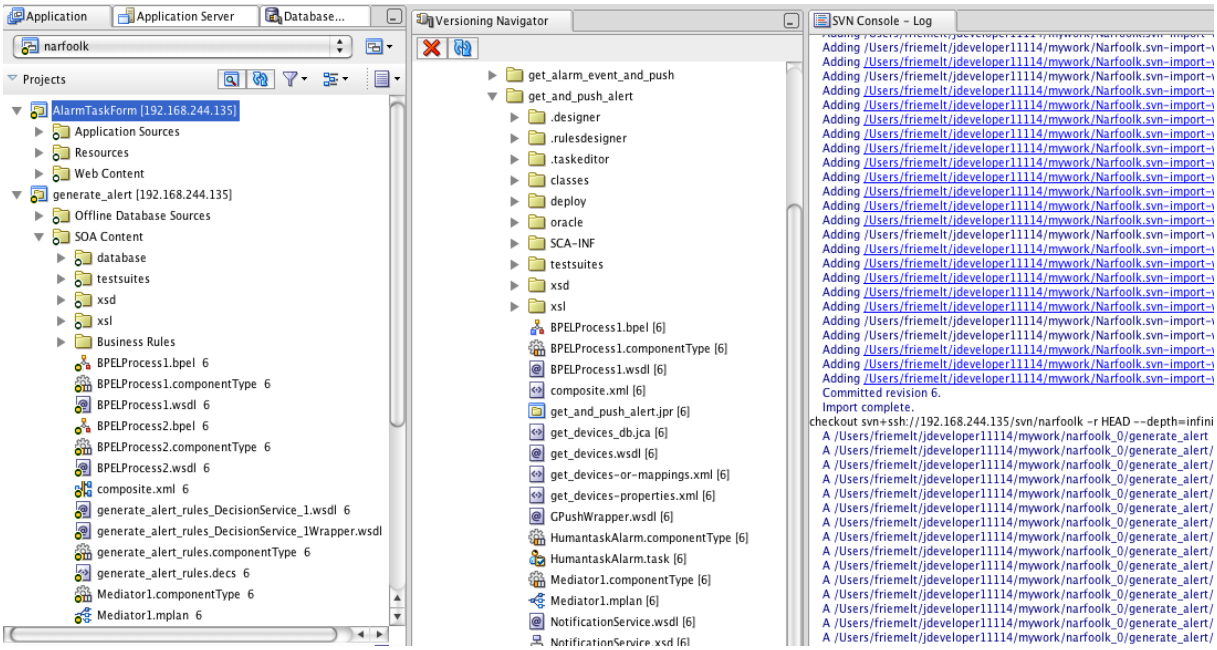

Abb. 7: Ansicht einer versionierten Applikation mit dem Versioning Navigator und dem SVN Console Log

Nach dem Import stehen im Arbeitsbereich zwei Ordner (<Applikationsname>.svn-import-backup und (<Applikationsname>.svn-import-workarea), die nach dem erfolgreichen Import überflüssig sind und gelöscht werden können.

SOA Composite Projekte und rekursives Locking

Rund um das composite.xml als zentrale Quelle eines SOA Composite Projektes entstehen je nach involvierter Komponente (Human Task, BPEL, Mediator, Rules, Webservices) zahlreiche Artefakte mit gegenseitigen Referenzen untereinander. Speziell bei der Verwendung der Human Task-Komponente hat man die Möglichkeit, sich die passenden Eingabedialoge dazu als separates ADF-Projekt innerhalb seiner Applikation generieren zu lassen. Damit bestehen auch projektübergreifende Abhängigkeiten.

Alle diese Quellen liegen im JDeveloper im xml-Format vor, tragen aber die entsprechenden aussagekräftigen Dateiendungen, wie z.B. *.bpel, *.mplan, *.wsdl. Dies sieht man immer, wenn man jeweils unten im entsprechenden Editor von der Design-Ansicht auf die Source-Ansicht umschaltet. Nun könnte man diese als textbasiert ansehen und den Ansatz „copy-modify-merge“ verfolgen. Das birgt aber wegen der angesprochenen Abhängigkeiten Gefahr, sich das gesamte Projekt durch Inkonsistenzen unbrauchbar zu machen. Es empfiehlt sich daher im Vorfeld möglichst kleine Projekte zu definieren und später beim Versionieren immer mit Locking zu arbeiten.

Hierbei stößt man auf das Problem, das Subversion von Natur aus kein rekursives Locking beherrscht (und auch das einfache Locking erst in späten Versionen gelernt hat). Ein Lock bezieht sich immer nur explizit auf ein Objekt. Will man alle abhängigen Artefakte eines Projektes sperren, muss man diese mühsam einzeln selektieren und sperren.

Diverse SVN-Clients wie TortoiseSVN oder das SVN-Plugin für das aktuelle Eclipse (Versionsstand „Helios“) bieten jedoch trotzdem ein rekursives Locking an auf alle Objekte innerhalb einer Ordnerhierarchie. Rekursives Locking ist dort einfach eine clientseitige Schleife über alle untergeordneten Objekte. Der JDeveloper als SVN-Client (ausgewiesene Version „SVNKit/1.3.0 with JNA disabled“) bietet leider auf Ordnerebene die Menüpunkte Lock/Unlock gar nicht erst an.

Eine Möglichkeit, dieses Manko zu umschiffen, wäre der Einsatz von „SVN-Hooks“. Dabei handelt es sich um Ereignistrigger, die bei bestimmten Vorgängen auf dem Repository ausgeführt werden. Wie in Abb. 4 zu sehen gibt es im SVN-Repository einen Unterordner namens „hooks“. Dieser enthält Templates für ausführbare Skripte. Als geneigter Shell-Programmierer kann man so ein post-lock schreiben, welches für untergeordnete Objekte ebenfalls ein Lock setzt.

```
/Users/friemelt/repos
friemelts-MacBook-Pro:repos friemelt$ ls hooks/
post-commit.tmpl                pre-lock.tmpl
post-lock.tmpl                  pre-revprop-change.tmpl
post-revprop-change.tmpl        pre-unlock.tmpl
post-unlock.tmpl                start-commit.tmpl
pre-commit.tmpl
```

Abb. 8: Template-Skripte für SVN-Ereignistrigger

Alternativ könnte man mit organisatorischen Vorgaben arbeiten und sagen, jeder Entwickler im Team prüft und respektiert gesetzte Locks auf Ebene der Projekt (*.jwr) Datei.

Zusammenfassung

Der JDeveloper 11g bietet als die zentrale Entwicklungsumgebung für SOA Composites mit seiner leicht zu integrierenden Versionskontrolle schon eine gewisse Sicherheit, definierte Projektstände zu archivieren. Wegen der weit verzweigten Abhängigkeiten der Artefakte wäre es für größere Entwicklerteams wünschenswert, ein über rekursives Locking zu verfügen. Sind organisatorische Anweisungen oder Ereignistrigger keine Option, bliebe die Hoffnung, dass der SVN-Client des JDevelopers ein Update erhält. Oder – falls bereits verfügbar oder im Budgetrahmen enthalten – die Verwendung von Dimensions als Versionskontrollsystem.

Kontaktadresse:

Klaus Friemelt
MT AG
Balcke-Dürr-Allee 9
D-40882 Ratingen

Telefon: +49(0)2102-309 61-0
Fax: +49(0)2102-309 61-50
E-Mail klaus.friemelt@mt-ag.com
Internet: www.mt-ag.com