Separates Deployment von Produktdaten

24. August 2010

Einleitung

Faktor-IPS verwaltet Produktdaten während der Produktentwicklung in XML Dateien. Zur Laufzeit wird bisher (Version 2.5. und frühere Versionen) ebenfalls mit XML Dateien gearbeitet. Genau genommen muss man von XML-Ressourcen sprechen, da die einzelnen XML-Dateien meist in Bibliotheken (z.B. JAR-Dateien) zusammengefasst werden. Um die Produktdaten zu laden müssen sie zur Laufzeit im Classpath der Applikation verfügbar sein, entweder als einzelne Dateien oder in Form von Bibliotheken. Innerhalb einer Applikation kann auf die Produktdaten über ein Runtime Repository (z.B. *ClassloaderRuntimeRepository*) zugegriffen werden.

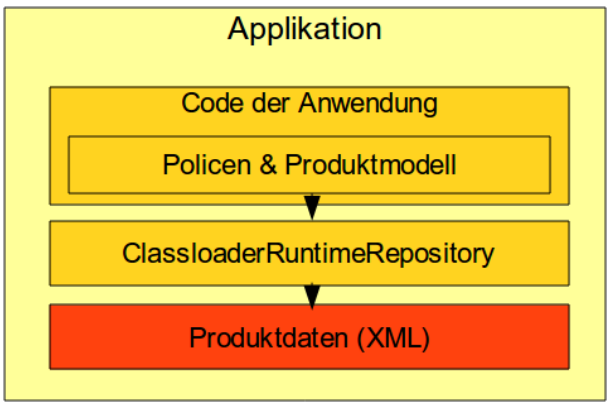

Abbildung 1: Bisher wurden Programmcode und Produktdaten in einer gemeinsamen Applikation ausgeliefert

Da sowohl Programmcode als auch Produktdaten als Dateien vorliegen, können sie zusammen in einem Werkzeug wie CVS oder Subversion gespeichert werden. Eine fertige Version von Programmcode und Produktdaten wird dann in eine Zielumgebung transportiert. Im JavaEE Umfeld wird dazu ein Enterprise Archive (EAR) bzw. ein Web Application Archive (WAR) erzeugt.

Werden die Produktdaten verändert oder erweitert muss dieses Archiv ausgetauscht werden. Da Produktdaten und Programmcode eine Einheit bilden, muss auch der Programmcode neu ausgeliefert werden.

Ziel des separaten Deployments von Produktdaten ist es, bei einer Änderung der Produktdaten nur die XML-Ressourcen auszutauschen ohne den Programmcode neu ausliefern zu müssen. Die dabei auftretenden Herausforderungen und die daraus entwickelten Lösungen werden in diesem Dokument vorgestellt.

Herausforderungen

1 Runtime Repository

Für den Zugriff auf die Produktdaten steht in Faktor-IPS zur Laufzeit das Interface *IRuntimeRepository* zur Verfügung. Wie bereits einleitend erwähnt wurde bisher meist die Implementierung *ClassloaderRuntimeRepository* verwendet. Dieses lädt die Produktdaten über den Klassenpfad als Ressourcen.

Um die Produktdaten unabhängig vom Programmcode zu halten muss zunächst ein Repository entwickelt werden, dass die Produktdaten unabhängig vom Klassenpfad der Applikation laden kann. Die später genauer vorgestellte Lösung greift dazu auf einen separaten Service (EJB 3.0 Stateless Session Bean) zu; die Verwendung einer Datenbank soll ohne großen Aufwand implementierbar sein.

2 Ausführen von Formeln

In Faktor-IPS ist es möglich Formeln in einer einfachen Formelsprache in einem Produktbaustein einzugeben. In den Modellklassen wird lediglich die Signatur der Formel festgelegt, die Berechnungsvorschriften liegen in den Produktdaten. Um diese Formeln zur Laufzeit auszuführen werden sie vom Faktor-IPS-Builder in Java-Code übersetzt.

Bisher wurde dazu für jeden Anpassungsstufe eines Produktbausteins eine Subklasse erzeugt in der die Methode der Formel überschrieben wurde. Das Runtime Repository war so aufgebaut, dass es jeweils die korrekte Implementierung zu einem Produktbaustein lud.

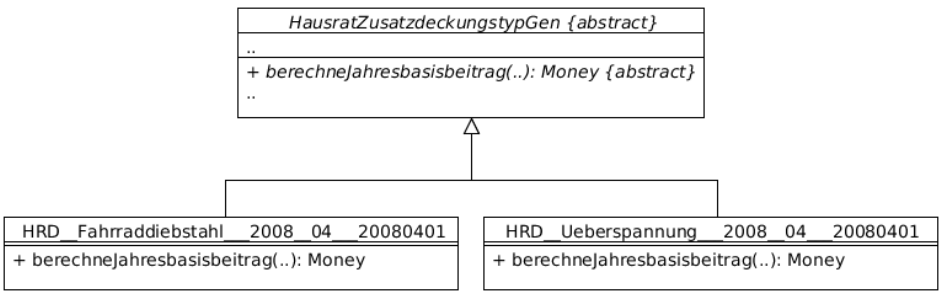

Abbildung 2: Für jeden Produktbaustein wird eine Subclass angelegt, in der die übersetzte Formel implementiert ist

Werden nun die Produktdaten durch einen separaten Service abgerufen, müsste nach die-

sem Ansatz auch die Implementierung über den Service abgerufen werden. Ändert sich in einer neuen Version der Produktdaten eine Formel, muss auch die Implementierung ausgetauscht werden. Der Classloader von Java sieht jedoch keinen Austausch von Klassen während des Betriebs vor. Da Application Server im JavaEE Umfeld selbst diverse Anpassungen am Classloader vornehmen ist eine eigene Anpassung an dieser Stelle nicht möglich.

Eine Herausforderung im separaten Deployment ist es daher, die Formeln nicht wie bisher in Subklassen zu kompilieren sondern zur Laufzeit zu interpretieren. Die Produktdaten werden dadurch frei von Programmcode.

3 Abarbeiten von Anfragen

Mit dem Separaten Deployment von Produktdaten soll es möglich sein, ohne Unterbrechung des Betriebs der Anwendung neue Produktdaten auszuliefern. Im Fall eines EJB Services wird also der Produktdaten Service per Hot-Deployment ausgetauscht.

Nehmen wir an, ein Client[1] möchte Produktdaten verarbeiten. Dazu muss er im allgemeinen eine Reihe von Anfragen an das Repository senden: z.B. fordert der Client einen Produktbaustein an, dann die aktuell gültige Generation und daraufhin alle assoziierten Produktbausteine. Für den Client ist dabei wichtig, dass er innerhalb seiner Anfrage auf einem konsistenten Datenstand arbeitet. Werden während solch einer Anfrage die Produktdaten ausgetatuscht, muss der Client entweder weiter mit den alten Daten versorgt werden oder muss eine Exception erhalten und seinen Request abbrechen.

Um bei einem neuen Versuch mit den neuen Daten weiter zu arbeiten, muss der Client den Beginn seiner Anfrage signalisieren. Das Runtime Repository kann daraufhin die neuen Produktdaten laden und an den Client weiterleiten. Dabei ist darauf zu achten, dass ein anderer Client, der sich noch in einer Anfrage befindet, immernoch eine Exception auslösen muss.

Lösungen

1 Überblick der Architektur

Um die im folgenden vorgestellte Lösung zu verwenden, werden die Runtime Addons[2] von Faktor-IPS beötigt. Das Archiv kann über die Download-Site[3] heruntergeladen werden.

Wie bereits in der Einleitung erwähnt, betrachten wir die Auslieferung in einem Java EE Umfeld. Die verschiedenen Programmteile werden als Services implementiert und in einem EAR ausgeliefert.

Wie bisher wird der Programmcode mit den Modellklassen in einem Archiv gekapselt.

[1] Als Client bezeichnen wir ein Programm, das Produktdaten abruft - also der Client des Runtime Repositories

[2] Dateiname des Archivs: faktorips-runtime-addons-[version].zip

[3] http://update.faktorzehn.org/faktorips/cgi/download.pl?subfolder=v3

Anstatt des *ClassloaderRuntimeRepositories* wird eine neue Implementierung des Interfaces *IRuntimeRepository* verwendet: das *DerivedContentRuntimeRepository*. Dieses ruft zum Laden der Produktdaten einen separaten Service auf, den Product-Data-Service. Im Klassenpfad des Product-Data-Service sind die Produktdaten weiterhin als XML-Ressourcen enthalten und können vom Service als Ressourcen geladen werden.

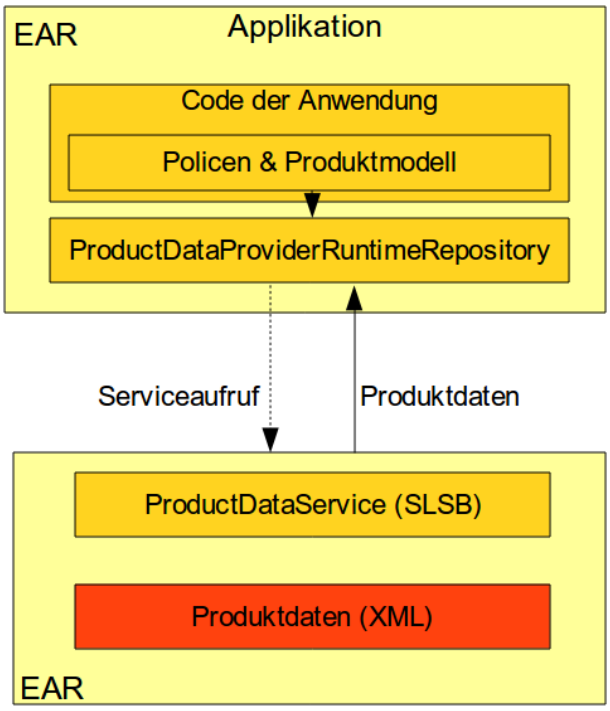

Abbildung 3: Die Produktdaten werden in einem separaten Service ausgeliefert der vom Runtime Repository verwendet wird

Der Product-Data-Service wird als Stateless Session Bean (SLSB) implementiert. Die Produktdaten werden als XML-Ressourcen geladen, der Inhalt wird an das Runtime Repository übergeben. Die Instantiierung findet weiterhin im Runtime Repository statt.

2 Interpretation von Formeln

Um die Formeln nicht in Subklassen zu implementieren muss ein Mechanismus zum Interpretieren des Formel-Codes geschaffen werden. Da das Compilieren der Formeln in Java-Code bereits existiert, ist die die naheliegenste Lösung diesen Mechanismus weiter zu verwenden. Der Java-Code wird dazu in das XML der Produktbausteine geschrieben. Zur Laufzeit wird der Code von Groovy[4] ausgeführt. Die Produktdaten enthalten dadurch keinen Programmcode und sind unabhängig vom Classloader der Applikation.

[4] http://groovy.codehaus.org/

3 Abarbeiten von Anfragen

Um innerhalb einer Anfrage einen konsistenten Datenstand zu gewährleisten, ist es notwendig, dass ein Client den Beginn seiner Anfrage ankündigt. Nach dem optimistischen Locking Verfahren kann die Anfrage abgearbeitet werden, bis es zu einem Fehler kommt. Wenn ein Fehler auftritt muss die Anfrage neu gestartet werden. Da der Austausch von Produktdaten nicht besonders oft auftritt, ist dieses Locking Verhalten völlig ausreichend.

Um dieses Locking zu realisieren wird ein Runtime Repository Manager eingeführt. Jeder Client erhält zunächst anstatt des Runtime Reposiories einen Manager. Möchte ein Client eine neue Anfrage starteten, ruft er die Methode *getActualRuntimeRepository*() am Manager auf. Der Manager liefert daraufhin ein Runtime Repository mit dem der Client auf den aktuell gültigen Produktdaten arbeiten kann. Vor der nächsten Anfrage holt sich der Client erneut das aktuelle Repository. Haben sich die Produktdaten nicht geändert, wird das gleiche Repository zurück gegeben. Dieses einfache Verhalten ist im Sequenzdiagramm in Abbildung 4 abgebildet.

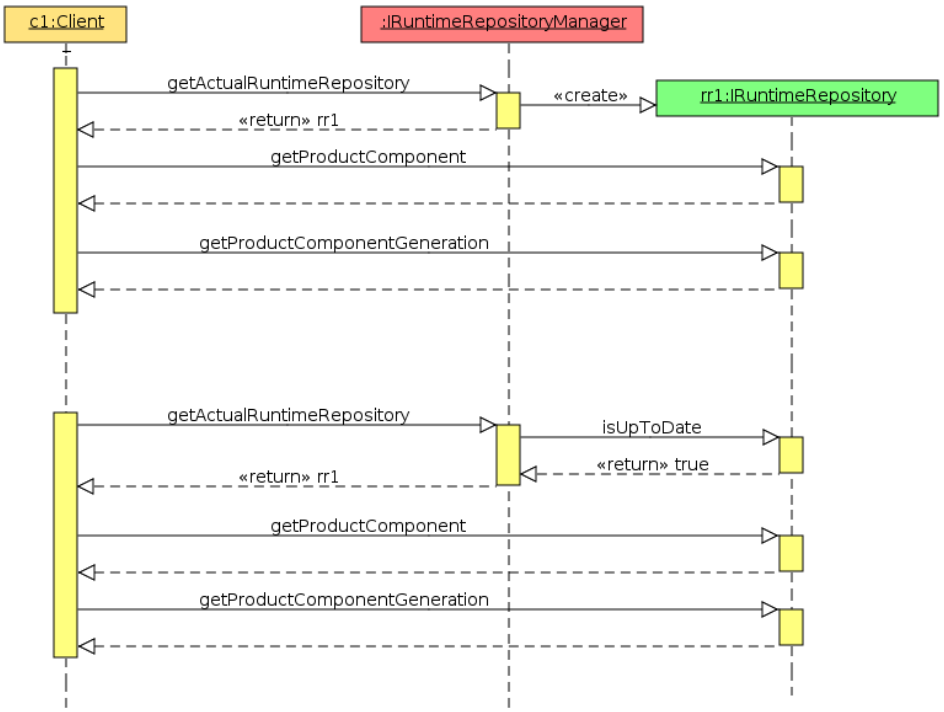

Abbildung 4: Der Client holt sich vor Beginn einer Anfrage das aktuelle Runtime Repository; solange sich keine Daten ändern arbeitet er auf dem gleichen Repository weiter.

Werden während der Anfrage die Produktdaten ausgetauscht, liefert das Runtime Repository beim nächsten Abruf von Produktdaten einen Fehler. Der Client muss dadurch seine Anfrage abbrechen.

Wiederholt der Client die Anfrage oder startet eine neue Anfrage, nachdem sich die Produktdaten geändert haben, stellt der Manager die geänderten Produktdaten fest. Es wird ein neues Runtime Repository erzeugt und an den Client zurück gegeben. Der Client arbeitet nun auf den neuen Produktdaten. Dieses Verhalten ist im Sequenzdiagramm in Abbildung 5 dargestellt.

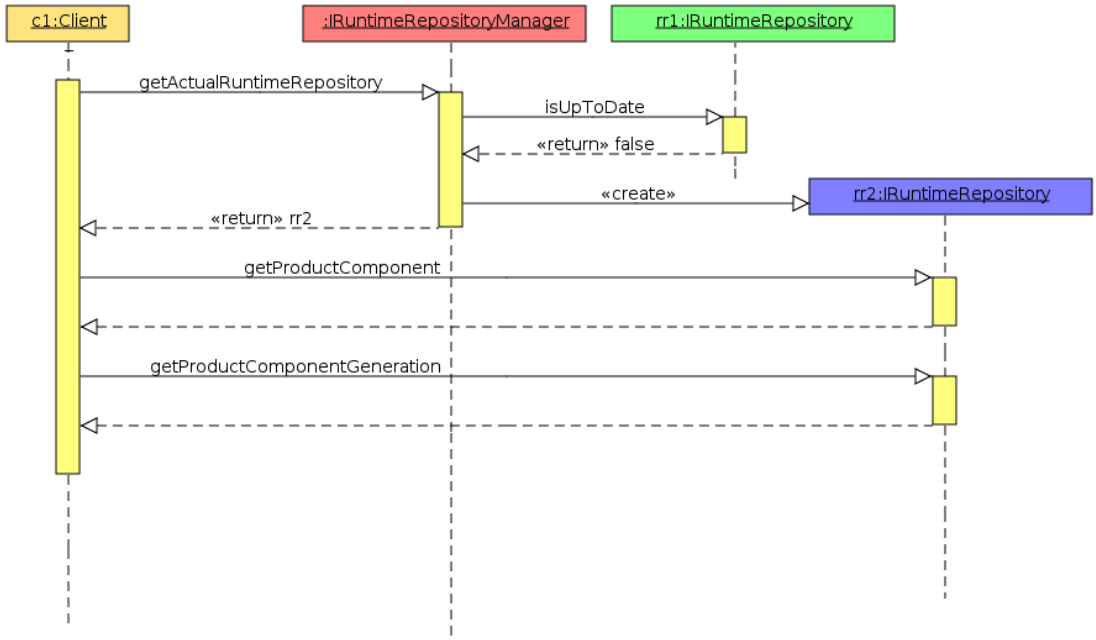

Abbildung 5: Der Manager stellt fest, dass sich die Produktdaten geändert haben und erstellt ein neues Runtime Repository

Um die Performance der Abfragen zu erhöhen, enthalten Runtime Repositories einen Cache und speichern darin bereits abgefragte Produktdaten. Um die Zahl der Abbrüche beim Austausch von Produktdaten weiter zu verringern wurde das Runtime Repository so entwickelt, dass es nur beim Abruf von nicht gespeicherten Produktdaten einen Fehler wirft - solange der Client nur auf Produktdaten im Cache zugreift, kann er dadurch seine Anfrage zu ende führen.

Auch der gleichzeitige Zugriff mehrerer Client ist mit dieser Lösung kein Problem. Lediglich der Abruf des aktuellen Runtime Repositories und der gemeinsam genutzte Cache müssen Threadsicher implementiert werden.

In der Client-Implementierung ist es sinnvoll, einen Manager an zentraler Stelle zu konfigurieren, damit alle Clients auf den gleichen Manager zugreifen. Zur Instantiierung des *DetachedContentRuntimeRepositoryManager* wird ein Builder[5] verwendet. Der Builder ist als innere Klasse des Repositories realisiert und erwartet eine Implementierung von

[5]Dieser Builder ist nach Item 2 im Buch „Effective Java“ von Joshua Bloch [1] erstellt und weicht vom bekannten Builder Design Pattern ab.

IProductDataProviderFactory zum instantiieren des Product-Data-Providers. Optional kann man im Builder weitere Voreinstellungen treffen. Insbesondere eine Implementierung von *IFormulaEvaluatorFactory*[6] zum ausführen der Formeln sollte gesetzt werden. Ist der Builder fertig konfiguriert, wird die Methode *build*() aufgerufen - als Ergebnis erhält man einen *DetachedContentRuntimeRepositoryManager*. Ein Beispiel zur Instantiierung des Managers ist in Listing 1 gegeben.

```
repositoryManager = new DetachedContentRuntimeRepositoryManager
  .Builder(pdpFactory)
  .setFormulaEvaluatorFactory(new GroovyFormulaEvaluatorFactory())
  .build();
```

Listing 1: Initialize DetachedContentRuntimeRepositoryManager

3.1 Verknüpfen mehrerer Repositories

In den meisten Fällen werden Modell- und Produktdaten in mehrere Projekte untergliedert. Für den Zugriff auf die Daten muss zur Laufzeit für jedes dieser Projekte ein eigenes Runtime Repository instantiiert werden. Um direkt mit einer Abfrage an alle relevanten Daten zu gelangen gibt es die Möglichkeit Runtime Repositories zu verbinden. Die Implementierung des Runtime Repositories durchsucht automatisch alle verbundenen Repositories um die gewünschten Daten zu finden.

Mit der Einführung des Runtime Repository Managers ist es nun Aufgabe des Managers die Repositories zu erstellen. Es muss damit ebenfalls sichergestellt werden, dass in einem neuen Repository alle notwendigen Verknüpfungen gesetzt sind. Da sich auch in verknüpften Repositories Produktdaten ändern können, muss der Manager außerdem überprüfen, ob sich in einem referenzierten Repository die Daten geändert haben.

Um diese Aufgaben zu bewältigen werden die Manager analog zu den Runtime Repositories miteinander verknüpft. Ein Manager kann dadurch alle verbundenen Manager nach dem aktuellen Repository fragen und entsprechend das eigene Repository zusammen bauen.

4 Implementierung eigener Product-Data-Providers

Ziel des Separaten Deployments von Produktdaten war es, die Produktdaten unabhängig von der Applikation ausliefern zu können. Dabei soll es nicht nur möglich sein, die Daten von einem Service abzurufen. Auch die Abfrage einer Datenbank sollte mit möglichst geringem Aufwand ermöglicht werden. Dazu wurde die eigentliche Abrage der Daten unabhängig vom *DerivedContentRuntimeRepository* implementiert. Dieses ruft über das Interface *IProductDataProvider* die Methoden zur Abfrage der Produktdaten auf. Zur Instantiierung des konkreten Product-Data-Providers erhällt das Runtime Repository eine *ProductDataProviderFactory*.

Das Interface *IProductDataProvider* enthält Methoden um die unterschiedlichen Produktdaten (ProductComponent, TableContent, EnumValue, ...) abzufragen. Außerdem

[6] z.B. GroovyFormulaEvaluatorFactory zur Ausführen der Formeln mit Groovy (in den Addons enthalten)

kann das Repository die aktuelle Table-Of-Content laden und die Version der Produktdaten abfragen.

Um festzustellen, ob zwei Versionen kompatibel sind, wird in der Abstrakten Implementierung *AbstractProductDataProvider* ein *IVersionChecker* verwendet. Die Überprüfung der Kompatibilität wird dadurch ausgelagert und kann je nach Umfeld verändert werden. Denkbar wäre beispielsweise, dass zwei Versionen kompatibel sind, wenn Major- und Minor-Version gleich sind, jedoch die Micro-Version abweicht.

Zusammenfassend müssen zur Implementierung eigener Product-Data-Providers folgende Implementierungen erstellt werden:

- *IProductDataProvider* zur Abfrage der Daten und der Version - optional kann die Abstrakte Klasse *AbstractProductDataProvider* verwendet werden
- *IProductDataProviderFactory* zum Erstellen des eigenen Product-Data-Providers
- Optional eine eigene Implementierung von *IVersionChecker*

Literatur

[1] Joshua Bloch, *Effective java (2nd edition) (the java series)*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2008.