Inklusive Teil 3 unserer Developer-Serie „Software-Entwicklungsprozesse“ S&S

Deutschland € 6,50 Österreich € 7,00 Schweiz sFr 13,40

entwickler magazin

Software, Systems & Development

CD-INHALT: Alle Infos auf Seite 5 | www.entwickler-magazin.de | Mai/Juni | 3.09

Entwickler-TV

Bonus Video

Bonus für Abonnenten der Profi-CD: Keynote der BASTA! Spring 2009

Testversion

Advantage Database Server 9

Software

Eclipse Rich Ajax Platform, Apache Tomcat 6.0.18, MySQL 6.0.9-alpha, PostgreSQL 8.3.7, Wireshark, NetBeans for C/C++

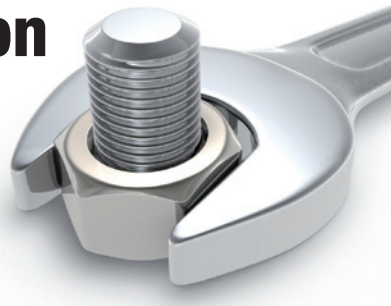
Weitere Artikel

- **SVG praktisch**
  Einsatz in der Geometrie
- **Per JDBC auf Datenbanken zugreifen**
- **WebOnDisk**
  Webanwendungen auf dem Desktop
- **PDF-Reporting**
  Optimierte Reports mit der Gnostice eDocEngine VCL

Datenträger enthält Info- und Lehrprogramme gemäß §14 JuSchG

Config im Griff

Softwarekonfiguration per Framework



Sicher tunneln: OpenVPN mit Delphi

Lose Kopplung

So werden SOAs einfacher und flexibler entwickelt

Auswahlstrukturen in C#

Grundlagen für Entwickler

SQLite für die Kleinen

Integration in Embedded-Systeme

Ein leichtgewichtiger Ansatz zur Softwarekonfiguration

Stellschrauben

Wer kennt das nicht: Man ist neu im Projekt und möchte der Konfiguration einfach nur einen zusätzlichen Parameter hinzufügen. Auf die Frage, wie das zu bewerkstelligen sei, drückt einem der altgediente Kollege mit mitleidvollem Blick die zwanzigseitige Anleitung zur projekteigenen Konfigurationslösung in die Hand. Oder er verweist auf ein Verzeichnis mit zwei Dutzend völlig unstrukturierten Property-Dateien. Wie wäre es stattdessen mit einem leichtgewichtigen Framework für einen einheitlichen Zugriff auf strukturiert abgelegte Konfigurationsparameter?

von Dr. Martin Jung und Dr. Thomas Walter

Eine flexibel einsetzbare Software muss konfigurierbar sein. Nur so kann sie ohne großen Aufwand an die jeweilige Einsatzumgebung angepasst werden. Häufig müssen z. B. Dateisystempfade zu externen Ressourcen festgelegt werden. Auch personalisierbare Software benötigt Konfigurationsparameter, um benutzerspezifische Einstellungen über eine Sitzung hinaus zu speichern. In kleinen Projekten verwendet man zur Speicherung der Konfigurationswerte meist einfach Textdateien aus Schlüssel-Wert-Paaren (so genannte *Property*- oder *Ini*-Dateien). Die Pflege von *Property*-Dateien wird mit wachsender Projektgröße jedoch schnell schwierig, da die Konfigurationsparameter nur anhand der Schlüsselnamen strukturiert werden können. Ab einer bestimmten Projektgröße geht man daher meistens dazu über, die Konfigurationsparameter in Form von XML-Dateien abzulegen. Für den Zugriff auf die Parameter wird dann entweder ein fertiges Framework eingesetzt oder eine eigene Komponente entwickelt. Kaum eine dieser Konfigurationslösungen bietet jedoch die Möglichkeit, Konfigurationsparameter mithilfe eines Modells zu verwalten. Diese Einschränkung führt im späteren Projektverlauf häufig zu Problemen, da mit steigender Komplexität der Software der Überblick über die Konfiguration verloren geht. Außerdem sind diese Frameworks und Komponenten oft schwergewichtig und erzeugen einen hohen Einarbeitungs- und Entwicklungsaufwand.

Vier Personengruppen kommen während des Lebenszyklus einer Software mit dem Thema Softwarekonfiguration in Berührung: Architekten, Entwickler, Administratoren und Anwender. Architekten und Entwickler entscheiden während der Design- und Implementierungsphase, in welchem Umfang und auf welche Art das Produkt später anpassbar sein soll. Dabei legen sie für viele Parameter auch gleich Standardwerte fest. Administratoren passen bei der Installation die Software an die jeweilige Einsatzumgebung an, indem sie geeignete Konfigurationswerte wählen. Schließlich richten die Anwender während des Betriebs die Software entsprechend ihren persönlichen Vorlieben ein und korrigieren diese Einstellungen gegebenenfalls von Zeit zu Zeit. Eine gute Konfigurationslösung sollte alle diese Personengruppen bei ihren Aufgaben unterstützen.

Wir stellen in diesem Artikel einen modellbasierten und leichtgewichtigen Ansatz zur Softwarekonfiguration vor.

Unser Ansatz unterstützt Architekten und Entwickler besser als bestehende Konfigurationslösungen. Zugleich bietet er Vorteile für Administratoren und Anwender, die Konfigurationswerte bevorzugt direkt mithilfe eines Editors bearbeiten.

Begriffsbestimmung und Aufbau des Artikels

In diesem Artikel geht es um Softwarekonfiguration und nicht um Konfigurationsmanagement. Als „Softwarekonfiguration" bezeichnet man die Flexibilisierung von Software durch die Verwendung von Konfigurationsparametern. Im Gegensatz dazu beschäftigt sich Konfigurationsmanagement mit der Herstellung von Softwareprodukten und der Erhaltung ihrer Eigenschaften über deren gesamten Lebenszyklus.

Unser Artikel ist wie folgt gegliedert: Zunächst zeigen wir die Stärken und Schwächen einiger bestehender Konfigurationslösungen. Danach beschreiben wir unsere Anforderungen an ein leichtgewichtiges Konfigurationsframework, und in den darauffolgenden Abschnitten stellen wir detailliert unseren modellbasierten Ansatz zur Softwarekonfiguration vor, der die zuvor aufgestellten Anforderungen bezüglich Leichtgewichtigkeit erfüllt. Schließlich erläutern wir die zusätzlichen Vorteile, die unser Ansatz in großen, komponentenbasierten Projekten mit architekturgetriebenem Entwicklungsansatz bietet. Alle in diesem Artikel vorgestellten Konzepte haben wir anhand eines Java-Prototyps validiert, auch die aufgeführten Code- und XML-Listings stammen aus diesem Prototyp. Allerdings haben wir alle Schema- und Namensraumangaben aus den XML-Listings entfernt, um die Lesbarkeit zu erhöhen.

Eine zentrale Idee unseres Ansatzes besteht darin, die Konfigurationsparameter in einer XML-Datei zu speichern und den Entwicklern in Form eines einfach navigierbaren Objektbaums zur Verfügung zu stellen. In unserem Prototyp haben wir die JAXB-Implementierung des Glassfish-Projekts [1] verwendet, um die XML-Konfigurationsdatei einzulesen und den entsprechenden Objektbaum zu erzeugen. Zur Durchführung von XSL-Transformationen haben wir Xalan-J [2] eingesetzt. Nun haben wir unseren Prototyp zwar in Java erstellt, jedoch ist unser Lösungsansatz auch auf andere Programmiersprachen übertragbar. Die einzige Voraussetzung hierfür ist, dass in dieser Sprache eine mit JAXB vergleichbare Bibliothek zur Verfügung steht.

Beispiele bestehender Konfigurationslösungen

Da fast jede Software Konfigurationsmöglichkeiten benötigt, wird schon seit Längerem an Lösungen gearbeitet, die den Zugriff auf Konfigurationsparameter erleichtern. So bieten die Standardbibliotheken einiger Programmiersprachen eine Unterstützung beim Zugriff auf Textdateien (in Form der *Property*- bzw. *Ini*-Dateien), in denen die Konfigurationsparameter in Form von Schlüssel-Wert-Paaren gespeichert sind. Darüber hinaus wurden in verschiedenen Open-Source-Projekten auch ausgefeiltere Konfigurationsframeworks entwickelt. Ein für die praktische Entwicklungsarbeit wichtiges Merkmal von Konfigurationslösungen ist die Typsicherheit ihrer Zugriffsmethoden. Während manche Konfigurationslösungen nur Zugriffsmethoden besitzen, die eine Zeichenkette zurückliefern, stellen andere Lösungen Zugriffsmethoden für eine Vielzahl von primitiven Datentypen zur Verfügung. Im zweiten Fall kann der Compiler bei der Übersetzung eine statische Typprüfung durchführen, was die Fehleranfälligkeit eines Programms vermindert. Jedoch ist im Fall von Konfigurierungslösungen die statische Typprüfung nicht ausreichend, um Laufzeitfehler auszuschließen, die aufgrund falscher Typen entstehen können. So tritt ein Laufzeitfehler immer dann auf, wenn eine Datenquelle für einen Konfigurationsparameter, der einen Wert mit numerischem Typ besitzen sollte, eine Zeichenkette gespeichert hat. Um solche Fehler zu vermeiden, muss die Konfigurationslösung die Möglichkeit bieten, den Datentyp eines Parameters zu deklarieren und den Typ des tatsächlich gespeicherten Werts zu prüfen (dynamische Typprüfung).

Aus der großen Menge der bestehenden Konfigurationslösungen besprechen wir im Folgenden neben den *Property*-Dateien auch zwei bekanntere Open-Source-Frameworks aus der Java-Welt. Zur Veranschaulichung der verschiedenen Lösungen verwenden wir in diesem Artikel immer wieder das Beispiel einer Webbrowserkonfiguration.

Property-Dateien speichern Konfigurationsparameter, wie bereits angesprochen, mithilfe von Schlüssel-Wert-Paaren. Zur Strukturierung der Parameter verwendet man im Allgemeinen eine paketartige Notation. Die Konfiguration für die Verbindungsparameter eines Proxy-Servers würde also wie folgt aussehen:

```
proxy.name = proxyserv
proxy.port = 4242
```

Für den Zugriff auf die Parameterwerte steht in Java nur eine Methode mit Rückgabetyp *String* zur Verfügung. Beim Einlesen numerischer Werte ist daher der Entwickler dafür verantwortlich, die gelesene Zeichenkette in einen numerischen Wert umzuwandeln:

```
String proxyPort = properties.getProperty("proxy.name");
int proxyPort = Integer.parseInt(properties.
                                    getProperty("proxy.port"));
```

Die Benutzung dieser Konfigurationslösung ist sehr leicht erlern- und einsetzbar. Eine statische oder dynamische Typprüfung wird allerdings nicht durchgeführt.

Das in Java realisierte Open-Source-Framework *Obix* [3] speichert Konfigurationsparameter im XML-Format. Einzelne Konfigurationsparameter können hier mithilfe von Modulen zusammengefasst werden. Da Module neben Parametern auch andere Module enthalten dürfen, ist es möglich, die Konfiguration beliebig tief zu strukturieren. Zusätzlich können die Module auf mehrere Dateien aufgeteilt werden. *Obix* bietet auch Unterstützung für Entwickler von Web- und Enterprise-Anwendungen. Für Webanwendungen enthält das *Obix*-Framework z. B. eine spezielle *Listener*-Klasse. Deklariert man diesen *Listener* im Anwendungsdeskriptor, so wird er

beim Start der Anwendung vom *Container* aufgerufen. Daraufhin lädt er die Konfiguration und macht sie über JNDI (Java Naming and Directory Interface) zugreifbar.

Für den Zugriff auf Konfigurationswerte existiert wie bei den *Property*-Dateien nur eine einzige Methode mit Rückgabetyp *String*. Um einen ganzzahligen Wert zu erhalten, muss der gelesene *String* daher mittels des Java-API explizit umgewandelt werden:

```
int proxyport = Integer.parseInt(config.
                        getStringValue("proxy.port"));
```

Somit findet auch beim *Obix*-Framework weder eine statische noch eine dynamische Typprüfung statt.

Commons-Config [4] ist ebenfalls ein in Java implementiertes Open-Source-Framework. Die Stärke von *Commons-Config* liegt in der Unterstützung unterschiedlicher Datenquellen, z. B. *Property*- und XML-Dateien. Parameter aus Verzeichnisdiensten oder des Betriebssystems lassen sich ebenfalls einlesen. Das Framework ermöglicht es sogar, verschiedene Datenquellen zu kombinieren und mittels eines einheitlichen API anzusprechen. *Commons-Config* bietet Zugriffsmethoden für die verschiedenen primitiven Datentypen. Das Lesen eines ganzzahligen Wertes ist daher einfacher als mit *Obix*:

```
int proxyport = config.getInt("proxy.port");
```

Listing 1

```
<configuration>
 <bean>
  <name>proxy</name>

  <int-property>
   <name>port</name>
   <value>12345</value>
  </int-property>

  <string-property>
   <name>hostname</name>
   <value>proxyserv</value>
  </string-property>
 </bean>
</configuration>
```

Commons-Config besitzt damit zwar typsichere Zugriffsmethoden, eine dynamische Typprüfung findet aber nicht statt.

Unser Ansatz: Modellbasiert und leichtgewichtig

Die im letzten Abschnitt besprochenen Konfigurationslösungen verfügen alle nur über „generische" Zugriffsmethoden. Der Name des Konfigurationsparameters, auf den zugegriffen werden soll, muss als Argument übergeben werden, wie z. B. in *config.getInt("proxy.port")*. Bei dem von uns vorgeschlagenen Ansatz werden die Konfigurationsparameter dagegen in einem Modell gepflegt. Dadurch kann man auch den Datentyp eines Parameters deklarieren und auf Basis des Modells datenspezifische Zugriffsmethoden generieren. Um den Konfigurationswert für den Port eines Proxy-Servers zu lesen, könnte daher in etwa folgende Zuweisung verwendet werden:

```
int proxyport = config.getProxy().getPort();
```

Ein solches API aus datenspezifischen Zugriffsmethoden ist aus folgenden Gründen wesentlich entwicklerfreundlicher als die üblichen generischen APIs:

- Aufgrund der datenspezifischen Methoden kann die Codevervollständigungsfunktion der Entwicklungsumgebung genutzt werden, um die Konfiguration nach einem bestimmten Parameter zu durchsuchen. Bei Verwendung der generischen Methoden muss dagegen der pfadartige Ausdruck, der den Konfigurationsparameter identifiziert, in der Dokumentation nachgeschlagen oder sogar anhand der Konfigurationsdatei ermittelt werden.
- Bei den datenspezifischen Methoden ist der Typ des Konfigurationsparameters aufgrund des generierten APIs bekannt. Eine statische Typprüfung durch den Compiler ist daher möglich. Wie wir später zeigen werden, ermöglicht unser Ansatz außerdem eine dynamische Typprüfung.
- Selbst ein einfacher Tippfehler beim Methodenargument macht sich bei den generischen Methoden erst zur Laufzeit bemerkbar. Bei unseren datenspezifischen Methoden würden Tippfehler beim Methodennamen dagegen bereits beim Übersetzen des Quellcodes erkannt werden.

Das Problem ist, dass modellbasierte Lösungen häufig schwergewichtig sind und von Entwicklern daher ungern genutzt werden. Wie wir in den folgenden Abschnitten zeigen werden, haben wir jedoch einen modellbasierten und trotzdem leichtgewichtigen Ansatz zur Softwarekonfiguration gefunden. „Leichtgewichtig" bedeutet hierbei für uns:

- Der Einarbeitungsaufwand für einen neuen Entwickler im Team beträgt weniger als dreißig Minuten.
- Das Hinzufügen, Ändern oder Löschen eines Konfigurationsparameters im Modell und die anschließende Codegenerierung beanspruchen weniger als drei Minuten.
- Die Modellierung und Codegenerierung lassen sich nahtlos in die Entwicklungsumgebung integrieren. Insbesondere werden keine außergewöhnlichen Werkzeuge benötigt.

Bei Modellierung denkt man immer zuerst an UML. Damit ist es jedoch schwierig, die obigen Anforderungen zu erfüllen. So hat nicht jeder Entwickler ein UML-Werkzeug auf seinem Rechner installiert. Eigentlich ist der UML-Sprachumfang für die Modellierung von Konfigurationsdaten auch unnötig groß. Daher haben wir uns stattdessen für XML als Modellierungssprache entschieden. Der XML-Sprachumfang ist völlig ausreichend und XML im Vergleich zu UML leichter zu erlernen und anzuwenden.

Die leichtgewichtige Modellierung

Als Einführungsbeispiel zeigt Listing 1 das Modell, das dem oben aufgeführten Ausdruck *config.getProxy().getPort()*, der dem Zugriff auf den Proxyport dient, zugrunde liegt.

Das *<configuration>*-Element ist das Wurzelelement der XML-Modelldatei. Mit dem *<bean>*-Element kann man Konfigurationselemente gruppieren und

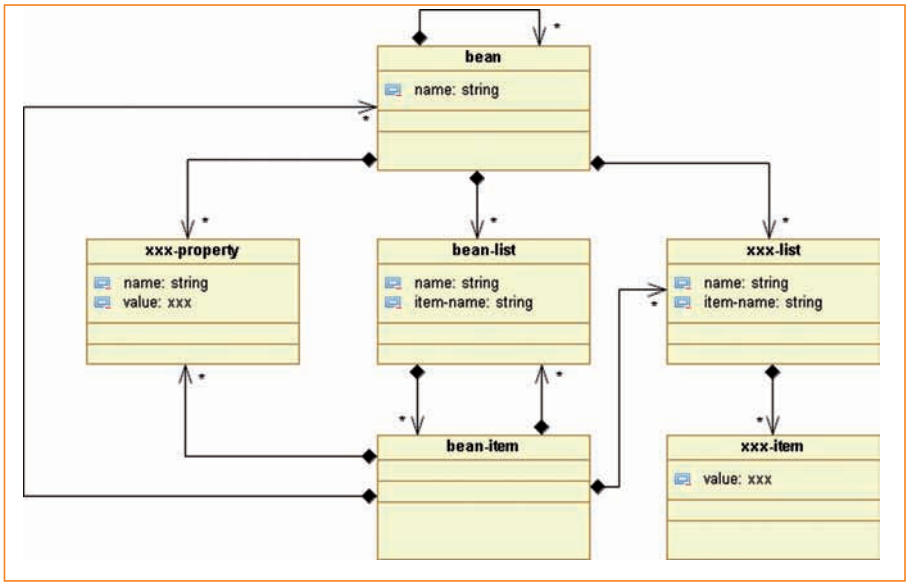


Abb. 1: Das Metamodell als Klassendiagramm

somit die Menge der Konfigurationsdaten strukturieren. In dem generierten API werden aus dem *<configuration>*- und dem *<bean>*-Element Java-Bean-Klassen, mit deren Getter- und Setter-Methoden man auf die umschlossenen Konfigurationselemente zugreifen kann. Der Wert des *<name>*-Elements (*proxy*) dient zur eindeutigen Identifikation der Bean. Mithilfe des *<string-property>*-Elements wird dem Konfigurationsparameter *hostname* der Wert *proxyserv* zugewiesen. Analog wird der Parameter *port* mittels des *<int-property>*-Elements auf den Wert *12345* gesetzt.

Manchmal möchte man statt eines einzelnen Konfigurationswertes eine ganze Reihe von Werten mittels eines Parameters abspeichern. Dies ist vor allem dann notwendig, wenn sich die Anzahl der Werte zur Laufzeit ändern kann. So bieten die meisten Browser die Möglichkeit, die URLs von Webauftritten zu speichern, die Cookies ablegen dürfen. Die Anzahl der Werte in dieser Liste kann beliebig groß werden. In einem solchen Fall kann das *<bean>*-Element nicht verwendet werden. Stattdessen benötigt man einen listenartigen Container. Listing 2 zeigt die Modellierung der oben beschriebenen Liste aus der Browserkonfiguration.

Listing 2

```
<configuration>
 <string-list>
  <name>cookies_allowed</name>

  <item-name>url</item-name>

  <string-item>www.example.org</string-item>
  <string-item>www.company.com</string-item>
 </string-list>
</configuration>
```

Jedes *<string-item>*-Element enthält einen Listenwert. Das *<string-list>*-Element bildet die Klammer um die einzelnen Konfigurationswerte. Das *<name>*-Element dient der eindeutigen Identifizierung der Liste. Der Wert des Elements *<item-name>* charakterisiert die einzelnen Konfigurationswerte. Die eigentliche Bedeutung dieses Elements wird jedoch erst im nächsten Abschnitt deutlich, wenn wir die Generierung der Zugriffsklassen erläutern.

Die formale Spezifikation der im Modell erlaubten XML-Elemente und ihrer Beziehungen untereinander erfolgt durch das so genannte *Metamodell*. Da wir uns für XML als Modellierungssprache entschieden haben, wurde das Metamodell konsequenterweise in Form eines XML Schemas realisiert. Das Metamodell kann dadurch direkt zur Validierung der Modelle verwendet werden. Darüber hinaus stellen professionelle XML-Editoren dem Benutzer eine auf dem Schema basierende Codevervollständigung zur Verfügung, die das Modellieren weiter vereinfacht. Abbildung 1 zeigt das Metamodell aufgrund der höheren Übersichtlichkeit jedoch in Form eines UML-Klassendiagramms. Das Wurzelelement *<config>* wird in der Abbildung nicht dargestellt. Es kann dieselben Kindelemente besitzen wie *<bean>*. Die Klassen in der Abbildung repräsentieren XML-Elemente, die Kindelemente besitzen können. Attribute einer Klasse stehen dagegen für XML-Elemente, die nur einen primitiven Wert in Form einer Zeichenkette enthalten. Die Variable *xxx* bezeichnet einen der im Text beschriebenen Datentypen.

In Listing 2 haben wir die Modellierung einer Liste anhand des *<string-list>*-Elements gezeigt. Es existieren jedoch auch XML-Elemente für Listen anderer Datentypen, z. B. *<boolean-list>*, *<int-list>* und *<float-list>*. Im Metamodell werden alle diese Listentypen durch die Klasse *xxx-list* repräsentiert, wobei die Variable *xxx* für einen der folgenden, in der XML-Spezifikation definierten Datentypen steht:

- boolean
- byte, short, int, long
- float, double
- string

Analog kann *xxx* auch an allen anderen Stellen des Metamodells durch einen dieser Typen ersetzt werden. Mithilfe des Metamodells (XML Schema) lassen sich die Eltern-Kind-Beziehungen zwischen den XML-Elementen des Modells einschließlich der spezifizierten Multiplizitäten validieren. Indem für jeden der oben beschriebenen Datentypen eine eigene Ausprägung des *<xxx-property>*-, *<xxx-list>*- und *<xxx-item>*-Elements geschaffen wurde, kann zusätzlich noch überprüft werden, ob

- der Wert eines *<value>*-Elements im Wertebereich des Datentyps *xxx* liegt, der durch das umschließende *<xxx-property>*- bzw. *<xxx-item>*-Element vorgegeben ist,
- eine Liste nur *<xxx-item>*-Elemente des Datentyps enthält, der durch das *<xxx-list>*-Element festgelegt ist.

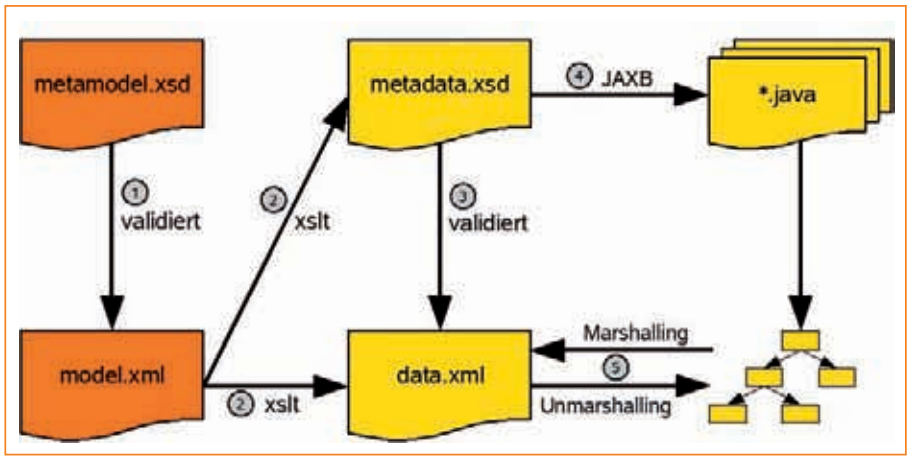


Abb. 2: Die für die Generierung des Zugriffs-API notwendigen Arbeitsschritte

Das *<bean-list>*-Element dient zur Modellierung von Listen aus *<bean>*-Elementen. Das *<bean-item>*-Element besitzt analog zum *<xxx-item>*-Element kein *<name>*-Kindelement, da das *<item-name>*-Element der Liste die einzelnen Einträge charakterisiert. Ansonsten entspricht ein *<bean-item>*-Element vollständig einem *<bean>*-Element. Wie man am Metamodell erkennt, kann ein *<bean>*-Element nicht nur *<xxx-property>*-Elemente enthalten (Listing 1), sondern auch *<xxx-list>*-, *<bean>*- und *<bean-list>*-Elemente. Durch die Möglichkeit, *<bean>*-Elemente zu schachteln, können die Konfigurationsparameter beliebig stark strukturiert werden.

Wir haben in diesem Abschnitt einige Spezialfälle der Modellierung, z. B. Aufzählungstypen, aus didaktischen Gründen beiseite gelassen. Dementsprechend zeigt Abbildung 1 auch nur ein vereinfachtes Metamodell. Die fehlenden Spezialfälle werden wir im Abschnitt „Spezialfälle der Modellierung" weiter hinten nachreichen.

Listing 3

```
<proxy>
 <port>12345</port>
 <hostname>proxyserv</hostname>
</proxy>
```

Listing 4

```
..
<element name="proxy" type="proxyType" />
...
<complexType name="proxyType">
 <sequence>
  <element name="port" type="int" />
  <element name="hostname" type="string" />
 </sequence>
</complexType>
```

Generierung des API aus dem Modell

Nachdem wir im letzten Abschnitt die Erstellung eines Modells besprochen haben, werden wir in diesem Abschnitt die Generierung des Zugriffs-API aus dem Modell erörtern. Abbildung 2 zeigt die hierfür notwendigen Arbeitsschritte (1)–(4), die an der Generierung beteiligten und die generierten Dateien. In der rechten unteren Ecke (5) ist außerdem die Erzeugung des Objektbaums aus der Datei *data.xml* (Unmarshalling), bzw. das Zurückschreiben des Objektbaums (Marshalling) zu sehen. Alle diese Schritte können mittels eines Build-Skripts automatisiert und in die Entwicklungsumgebung integriert werden. Bevor wir die notwendigen Arbeitsschritte im Detail erläutern, wollen wir einen kurzen Überblick über die Abfolge dieser Schritte geben:

1. Das Modell wird gegen das Metamodell validiert.
2. Mithilfe zweier XSLT-Skripte werden aus dem Modell die eigentlichen Konfigurationsdaten extrahiert und in die Datei *data.xml* geschrieben. Auf analogem Weg werden die Typinformationen des Modells in der Datei *metadata.xsd* gespeichert.
3. Es wird geprüft, ob *metadata.xsd* ein gültiges Schema darstellt. Anschließend wird die Datei *data.xml* mittels des Schemas *metadata.xsd* validiert.
4. Mithilfe der externen JAXB-Bibliothek wird aus der Schemadatei *metadata.xsd* das Zugriffs-API generiert.

Im ersten Arbeitsschritt wird das Modell anhand des Metamodells validiert. Bei unserem Lösungsansatz besteht dieser Schritt nur aus der Validierung einer XML-Datei anhand des zugehörigen XML Schemas. Hierfür stehen viele freie Werkzeuge und Bibliotheken zur Verfügung. Formale Fehler im Modell werden durch diese Validierung sofort entdeckt und können dem Entwickler in verständlicher Form gemeldet werden. Würde man diese Fehler nicht aufspüren, so würden sie in späteren Arbeitsschritten zu kryptischen Folgefehlern führen.

Im Modell sind für jeden Konfigurationsparameter zwei Arten von Informationen vorhanden: Der Datentyp und der Konfigurationswert. Als Ausgangspunkt der API-Generierung wird jedoch eine Schemadatei benötigt, die nur die Typinformationen enthält. Diese Schemadatei mit Namen *metadata.xsd* wird mittels einer XSL-Transformation aus dem Modell generiert. Analog werden die Konfigurationswerte per XSL-Transformation in die Datei *data.xml* geschrieben. Listing 3 zeigt den für das *<bean>*-Element aus Listing 1 generierten Abschnitt der Datei *data.xml*.

Wie man erkennt, wurden aus den Werten der verschiedenen *<name>*-Elemente die Namen der XML-Elemente, die die Konfigurationswerte umschließen. Die Datei wird dadurch schlanker und leichter lesbar. Listing 4 zeigt den entsprechenden Abschnitt der generierten Datei *metadata.xsd*.

Diese Zeilen definieren den Typ *proxyType*, der den Inhalt des *<proxy>*-Elements festlegt. Das Element muss demnach genau ein *<port>*- und ein *<hostname>*-Element enthalten.

Die XSL-Transformationen des letzten Teilschritts sind so aufeinander abgestimmt, dass die Schemadatei *metadata.xsd* verwendet werden kann, um die Datei *data.xml* zu validieren. Vor Aus-

führung der Validierung wird zusätzlich noch geprüft, ob *metadata.xsd* überhaupt eine gültige Schemadatei ist. Einige wenige Kategorien von Modellfehlern, die nicht bei der Validierung des Modells erkannt werden können, offenbaren sich in diesem Arbeitsschritt. So wird bei der Validierung des Modells nicht erkannt, ob zwei *<bean>*-Elemente mit dem gleichen Namen deklariert worden sind. Bei der Validierung des Schemas fällt dies jedoch auf, weil die XSL-Transformation in diesem Fall zwei komplexe Typen mit dem gleichen Namen erzeugt.

Im letzten Schritt wird das Zugriffs-API auf Basis des Schemas *metadata.xsd* generiert. Dafür verwenden wir die JAXB-Referenzimplementierung aus dem Glassfish-Projekt [1]. Diese externe Bibliothek enthält unter Anderem einen so genannten *Schema Compiler*, der diese Aufgabe übernimmt. Als Ergebnis erhält man für jeden komplexen Typ der Schema-Datei eine Java-Klasse. Im Fall des Typs *proxyType* entsteht eine Bean mit Namen *ProxyType*, die die Methoden *int getPort()*, *String getHostname()*, *setPort(int)* und *setHostname(String)* enthält.

Wir können jetzt auch die eigentliche Bedeutung des *<item-name>*-Elements nachliefern, das bei der Modellierung von Listen Verwendung findet. Dazu betrachten wir in Listing 5 den Abschnitt der Datei *data.xml*, der sich durch die XSL-Transformation von Listing 2 ergibt.

Wie man erkennt, bildet der Wert des *<name>*-Elements die Klammer um die gesamte Liste, während der Wert des *<item-name>*-Elements zum Namen des Elements wird, das einen einzelnen Wert umschließt. Bei der API-Generierung wird dann eine Klasse *CookiesAllowedType* erzeugt, die die Methode *public List<String> getUrl()* besitzt. Das *List*-Objekt, das diese Methode zurückgibt, enthält dann die mittels der Liste modellierten Werte.

Bei Betrachtung der Abbildung 2 kommt die Frage auf, warum man nicht gleich die Dateien *data.xml* und *metadata.xsd* erstellt, statt mit dem Modell *model.xml* zu beginnen. Hierfür gibt es zwei Gründe: Erstens müsste der Entwickler dann zwei Dateien pflegen statt einer und zweitens Kenntnisse über die Erstellung eines XML Schemas besitzen. Das würde unseren Grundsätzen zur Leichtgewichtigkeit entgegenstehen.

Allerdings kann dieser Zwischenschritt auch nicht weggelassen werden, da die Schemadatei *metamodel.xsd* die Grundlage für die Generierung des API bildet. Würde man das Metamodell zur Generierung heranziehen, würde man nur generische Klassen, z. B. *BeanType*, erhalten, die wiederum nur generische Methoden, z. B. *getIntProperty()*, *getFloatList()* usw. enthielten, aber nicht die von uns angestrebten datenspezifischen Methoden.

Lesen und Modifizieren von Konfigurationswerten

Indem man die generierten Klassen verwendet, kann man die Datei *data.xml* mit den hierarchisch strukturierten Konfigurationsdaten einlesen (Abb. 2, Arbeitsschritt 5). Das Framework erzeugt dann einen Objektbaum, der das eingelesene Dokument repräsentiert. Dieser Vorgang wird als „Unmarshalling" bezeichnet. Mithilfe der Getter-Methoden des generierten API fällt es anschließend leicht, durch den Objektbaum zu navigieren und Konfigurationswerte zu lesen. Mittels der Setter können die Konfigurationswerte auch geändert werden. Im Fall von Listen ist es zusätzlich möglich, neue Listenelemente hinzuzufügen oder existierende zu löschen. Falls Konfigurationswerte geändert wurden, so kann der Objektbaum auch wieder in eine XML-Datei zurückgeschrieben werden (Marshalling). Dabei wird geprüft, ob das Zieldokument dem XML Schema *metadata.xsd* entspricht. Treten bei der Validierung Fehler auf, so wird eine Ausnahme geworfen. Listing 6 zeigt das Einlesen der Datei *data.xml*, das Lesen und Modifizieren von Parameterwerten, das Anlegen eines neuen Listenelements und das Zurückschreiben des Objektbaums.

Hier werden die mithilfe von JAXB generierten Methoden zum Einlesen und Zurückschreiben der Parameterdatei verwendet. Um die Arbeit des Entwicklers weiter zu vereinfachen, kann man die Vielzahl der dazu benötigten Anweisungen noch hinter einer Fassade [5] mit der Schnittstelle

```
public ConfigurationType getConfiguration()
public void writeComponentConfiguration()
```

Listing 5

```
<cookies_allowed>
 <url>www.example.org</url>
 <url>www.company.com</url>
</cookies_allowed>
```

Listing 6

data.xml

```
// Die Datei mit den Konfigurationsparametern wird
//eingelesen.
JAXBContext jAXBContext =
  JAXBContext.newInstance("de.developgroup.japetos");
Unmarshaller unmarshaller =
                    jAXBContext.createUnmarshaller();
JAXBElement<ConfigurationType> jAXBElement =
  (JAXBElement) unmarshaller.unmarshal(
    new File("xml_gen/data.xml"));
ConfigurationType config = jAXBElement.getValue();

// Die Werte für die Parameter Port und Host werden
// ausgegeben.
System.out.println("port   : " + config.getProxy().
                                        getPort());
System.out.println("hostname : " + config.getProxy().
                                    getHostname());

// Die URLs aus der Liste cookies_allowed werden
// ausgegeben.
CookiesAllowedType cookiesAllowed =
                        config.getCookiesAllowed();
List<String> urls = cookiesAllowed.getUrl();
for (String url : urls) {
  System.out.println("URL: " + url);
}

// Der Wert des Ports wird geändert und ein neuer URL
// zur Liste
// hinzugefügt.
config.getProxy().setPort(5500);
urls.add("www.develop-group.de");

// Die aktuellen Konfigurationsparameter werden
// zurückgeschrieben.
// Dabei werden sie anhand des Schemas metadata.xsd
// validiert.
SchemaFactory schemaFactory =
  SchemaFactory.newInstance(XMLConstants.W3C_
                              XML_SCHEMA_NS_URI);
Schema schema = schemaFactory.newSchema(
  new File("xml_gen/metadata.xsd"));
Marshaller marshaller = jAXBContext.createMarshaller();
marshaller.setSchema(schema);
marshaller.marshal(jAXBElement, new File("xml_gen/
                                        data.xml"));
```

verstecken. Wird diese Fassade gleichzeitig noch als Singleton [5] realisiert, so ist sichergestellt, dass verschiedene Programmteile nicht inhaltlich unterschiedliche Objektbäume verwenden.

Die Tatsache, dass wir die Parameterdatei *data.xml* mittels des Schemas *metadata.xsd* validieren können, bietet noch einen weiteren Vorteil. So kann ein Administrator oder Benutzer, der Konfigurationswerte manuell mithilfe eines XML-Editors geändert hat, prüfen, ob die Parameterdatei noch gültig ist. Ohne diese Möglichkeit würde ein solcher Fehler erst durch Folgefehler im laufenden System auffallen.

Spezialfälle der Modellierung

In diesem Abschnitt besprechen wir einige Spezialfälle der Modellierung, die wir im Abschnitt *„Die leichtgewichtige Modellierung“* aus didaktischen Gründen unterschlagen haben. Diese Spezialfälle umfassen die Modellierung von Einschränkungen des Wertebereichs, Aufzählungstypen und so genannte *Dummy-Bean-Listen*.

Wie wir im letzten Abschnitt gezeigt haben, besteht nach programmatischen oder manuellen Konfigurationsänderungen die Möglichkeit, die modifizierte Parameterdatei gegen das Schema *metadata.xsd* zu validieren. Dabei wird unter anderem geprüft, ob die Werte der Konfigurationsparameter im Wertebereich der zugehörigen XML-Datentypen liegen. Manchmal möchte man den Wertebereich eines Parameters jedoch auf eine Teilmenge des Wertebereichs eines XML-Datentyps einschränken. Die XML-Schema-Spezifikation erlaubt es bereits, eine solche Wertebereichseinschränkung mithilfe des Schemas festzulegen. Die gewünschte Einschränkung muss also nur im Modell verankert und dann mittels der XSL-Transformation in das Schema *metadata.xsd* übertragen werden. Bei einer Validierung von *data.xml* wird die Einhaltung des Wertebereichs dann automatisch geprüft. Listing 7 zeigt z. B., wie im Modell der Wertebereich für einen Port auf Werte größer oder gleich 5000 und kleiner 6000 festgelegt werden kann:

Im Prinzip können auf diese Weise alle Wertebereichseinschränkungen verwendet werden, die die XML-Spezifikation definiert. (In unserem Prototyp haben wir außer *minInclusive* und *maxExclusive* nur noch die Wertebereichseinschränkung von Zeichenketten durch reguläre Ausdrücke umgesetzt.) Es ist auch möglich, Aufzählungstypen zu modellieren. Listing 8 zeigt, wie man die möglichen Werte für den Hostnamen des Proxies auf die beiden Zeichenketten *proxyserv1* oder *proxyserv2* beschränkt.

Die XSL-Transformation trägt die Werte der Aufzählung als *<enumeration>*-Elemente in das XML Schema ein. Bei der Codegenerierung wird dann auf Basis dieser Elemente ein echter Java-Aufzählungstyp erzeugt. Dadurch entsteht die Möglichkeit, bereits bei der Zuweisung eines Wertes den Wertebereich zu überprüfen und nicht erst beim Marshalling.

Ein weiterer Spezialfall der Modellierung ist für *Bean*-Listen notwendig. Bei Listen kann es vorkommen, dass die Liste anfänglich kein einziges Element enthalten soll. Bei den *<xxx-list>*-Elementen ist dies kein Problem, man lässt einfach die *<xxx-item>*-Elemente weg. Die API-Generierung funktioniert trotzdem, da der Typ der Listenelemente durch das *<xxx-list>*-Element gegeben ist. Anders verhält es sich bei *Bean*-Listen. Ohne ein *<bean-item>*-Element kann das API nicht generiert werden, da die interne Struktur der *Beans* in der Liste unbekannt ist. Daher muss bei der Modellierung einer *<bean-*

Listing 7

```
<int-property>
 <name>port</name>
<value>5000</port>

 <restriction>
  <minInclusive>5000</minInclusive>
  <maxExclusive>6000</maxExclusive>
 </restriction>
</int-property>
```

Listing 8

```
<string-property>
 <name>hostname</name>
<value>proxyserv1</value>

 <restriction>
  <value>proxyserv1</value>
  <value>proxyserv2</value>
 </restriction>
</string-property>
```

Listing 9

```
public class ConfigurationAccess {
private ConfigurationType config;
private JAXBContext jAXBContext;
private Unmarshaller unmarshaller;
private JAXBElement<ConfigurationType> jAXBElement;

public ConfigurationAccess() throws JAXBException {
// Die Datei mit den Komponentenkonfigurationspara-
// metern wird
// eingelesen.
this.jAXBContext =
  JAXBContext.newInstance("de.developgroup.japetos");
 this.unmarshaller = this.jAXBContext.
                                    createUnmarshaller();
this.jAXBElement = (JAXBElement) this.unmarshaller.
                                              unmarshal(
  new File("xml_gen/data.xml"));
 this.config = this.jAXBElement.getValue();

// Die Datei mit den Systemkonfigurationsparametern wird
// eingelesen.
 JAXBElement<SystemConfigurationType>
                                    sc_jAXBElement =
   (JAXBElement) this.unmarshaller.unmarshal(
      new File("xml_gen/sc_data.xml"));
 SystemConfigurationType systemConfig =
                                    sc_jAXBElement.getValue();

// Einhängen der Systemkonfiguration in die
// Komponentenkonfiguration.
 this.config.setSystemConfiguration(systemConfig);
}

public ConfigurationType getConfiguration() {
 return this.config;
}

public void writeComponentConfiguration()
  throws SAXException, JAXBException {
// Aushängen der Systemkonfiguration aus dem
// Objektbaum der
// Komponentenkonfiguration.
 config.setSystemConfiguration(null);

 // Zurückschreiben der Komponentenkonfiguration wie
// in Listing 6.
}
}
```

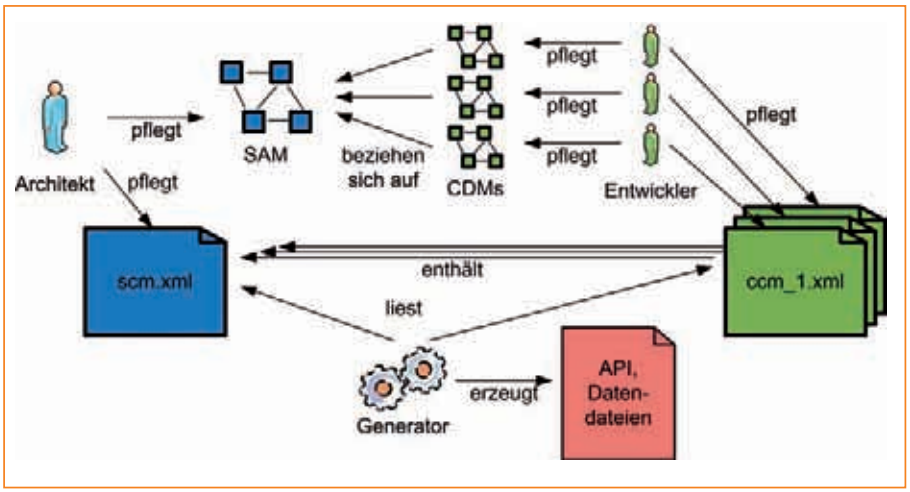


Abb. 3: Rollen, Artefakte und Beziehungen bei der architekturgetriebenen Entwicklung und Konfiguration

list> immer mindestens ein <*bean-item*>-Element samt dessen vollständiger innerer Struktur aufgeführt werden. Damit dieses <*bean-item*>-Element nicht in der Datei *data.xml* als Listenelement erscheint, fügt man dem <*bean-list*>-Element das Attribut *dummy* mit dem Wert *true* hinzu. Die XSL-Transformationen erzeugen dann nur die notwendigen Typen im Schema *metadata.xsd*, der Eintrag in der Datei *data.xml* wird dagegen unterdrückt.

Framework in der architekturgetriebenen Entwicklung

Unser Ansatz zur Softwarekonfiguration, wie wir ihn bisher vorgestellt haben, ist für kleine und mittelgroße Softwareprojekte gut geeignet. In großen, komponentenbasierten Projekten ergeben sich jedoch noch weitere Anforderungen an eine Konfigurationslösung, die unser Ansatz in seiner beschriebenen Form noch nicht erfüllt. So möchte man in einem komponentenbasierten Projekt systemweit gültige und komponentenspezifische Parameter getrennt voneinander verwalten. Außerdem sollte ausschließlich der Architekt die systemweit gültigen Parameter ändern können, während die Komponentenentwickler nur die Parameter ihrer eigenen Komponente pflegen dürfen. Durch die bewusste Zuordnung der Parameter zu einzelnen Komponenten werden implizite Abhängigkeiten zwischen den Komponenten vermieden.

In großen Projekten wird häufig ein architekturgetriebener Entwicklungsansatz verfolgt. Wie in [6] beschrieben, haben wir dabei die besten Erfahrungen damit gesammelt, dass der Anwendungsarchitekt ein globales Modell der Software erstellt, das Softwarearchitekturmodell (SAM). Dieses betrachtet die einzelnen Komponenten lediglich als „Blackbox" und definiert deren Schnittstellen und die dynamischen Interaktionen. Das *SAM* ist bindend für die Komponentenentwickler und wird nur vom Architekten geändert. Die Komponentenentwickler erstellen Komponentenentwurfsmodelle (Component Design Models – CDM), die auf den Schnittstellendefinitionen des *SAM* aufbauen und die Realisierung der Komponenten beschreiben. Die Konsistenz des *SAM* und der *CDMs* untereinander wird werkzeuggestützt sichergestellt. Aus allen Modellen zusammen wird dann ein Teil des Anwendungscodes automatisch generiert.

Damit sich eine Konfigurationslösung in ein komponentenbasiertes Softwareprojekt mit architekturgetriebenem Entwicklungsansatz integrieren lässt, muss sie also folgende zusätzliche Anforderungen erfüllen:

- Pro Komponente soll ein Komponentenkonfigurationsmodell (Component Configuration Model – CCM) existieren. Der Komponentenentwickler modelliert mithilfe des *CCM* die für seine Komponente spezifischen Konfigurationsparameter.

- Zusätzlich existiert ein Systemkonfigurationsmodell (System Configuration Model – SCM). Darin werden vom Architekten die systemweit gültigen Konfigurationsparameter gepflegt.
- Pro Komponente existiert ein API, mit dessen Hilfe man auf alle Konfigurationsparameter der Komponente und zusätzlich auf die systemweit gültigen Parameter zugreifen kann. Das API gestattet auch ein Zurückschreiben modifizierter Komponentenkonfigurationsparameter. Modifizierte Systemparameter lassen sich dagegen nicht zurückschreiben, da sonst die Kapselung der Komponenten durchbrochen werden würde.

Wir werden im Folgenden zeigen, wie unser Ansatz zur Softwarekonfiguration so erweitert werden kann, dass er die oben genannten Anforderungen erfüllt. Bei der Entwicklung einer komponentenbasierten Software verwendet man in unserem erweiterten Ansatz für jedes Komponentenkonfigurationsmodell eine eigene XML-Datei. Die Modellierung erfolgt dabei für jede Komponente exakt nach den Regeln, die wir in den Abschnitten *„Die leichtgewichtige Modellierung“* und *„Spezialfälle der Modellierung“* besprochen haben. Dadurch können alle Komponentenkonfigurationsmodelle weiterhin mithilfe des Metamodells validiert werden.

Das Systemkonfigurationsmodell wird ebenfalls in einer eigenen XML-Datei gepflegt. Es unterscheidet sich im syntaktischen Aufbau von den Komponentenkonfigurationsmodellen nur dadurch, dass das Wurzelelement *<system-configuration>* lautet. Ein entsprechendes Metamodell, das diesen Unterschied berücksichtigt, existiert ebenfalls, sodass auch das Systemkonfigurationsmodell jederzeit validiert werden kann. Das globale Systemkonfigurationsmodell wird, wie im Abschnitt „Generierung des API“ beschrieben, durch XSL-Transformationen in die Dateien *sc_metadata.xsd* und *sc_data.xml* überführt (wiederum mit der Ausnahme, dass das Wurzelelement der Datei *sc_data.xml* jetzt *<system-configuration>* heißt). Einen schematischen Überblick der Konfigurationslösung im architekturgetriebenen Umfeld zeigt Abbildung 3.

Bei der Erzeugung der Schemata *metadata.xsd* für die einzelnen Komponentenkonfigurationsmodelle müssen zwei zusätzliche Zeilen erzeugt werden. Zum einen wird das Schema *sc_metadata.xsd* durch ein *<include>*-Element in die Komponentenschemata eingebunden. Dadurch werden im Paket der Komponente neben den komponentenspezifischen Klassen auch Klassen für die Systemkonfiguration generiert. Zum anderen wird festgelegt, dass das erste Element im *<configuration>*-Wurzelelement ein optionales Element mit dem Namen *<system-configuration>* ist. Durch diese Zeile erhält die Klasse *ConfigurationType* eine Setter- und eine Getter-Methode für den Zugriff auf den Objektbaum der Systemkonfiguration.

Listing 9 zeigt einen Ausschnitt aus der Fassade, die den Entwicklern für den lesenden und schreibenden Zugriff auf die Konfiguration bereitgestellt wird. Diese Klasse sollte man eigentlich als Singleton implementieren (Abschnitt *„Lesen und Modifizieren von Konfigurationswerten“*). Wir haben das jedoch unterlassen, um die Lesbarkeit zu erhöhen.

Im Konstruktor der Klasse werden die System- und die Komponentenkonfiguration getrennt eingelesen. Anschließend wird die Systemkonfiguration in den Objektbaum der Komponentenkonfiguration eingehängt. Dadurch kann der Entwickler über das *ConfigurationType*-Objekt, das er durch Aufruf von *getConfiguration()* erhält, auch direkt auf die Systemkonfigurationsparameter zugreifen. Mittels der Methode *writeComponentConfiguration()* kann man die Komponentenkonfiguration zurückschreiben. Innerhalb dieser Methode wird vor dem eigentlichen Schreibvorgang die Systemkonfiguration wieder aus dem Objektbaum entfernt. Ohne diese Maßnahme würden sonst die Systemkonfigurationsparameter in die XML-Datei für die Komponentenkonfiguration geschrieben.

Fazit

Verwaltet man die Konfigurationsparameter einer Software mithilfe eines Modells, so kann man auf dessen Basis ein datenspezifisches Zugriffs-API generieren. Ein solches Zugriffs-API bietet viele Vorteile gegenüber den generierten APIs bestehender Konfigurierungslösungen, insbesondere typsichere Zugriffsmethoden. Wir haben hier einen leichtgewichtigen Ansatz für eine solche modellbasierte Softwarekonfiguration entwickelt. Unser Ansatz sieht vor, XML als Modellierungssprache zu verwenden und das Zugriffs-API mithilfe des JAXB-Frameworks zu generieren. Durch die Verwendung von XSL-Transformationen werden die Entwickler außerdem vom Schreiben der zur Generierung benötigten XML Schemas befreit.

Unser Ansatz lässt sich auch in großen, komponentenbasierten Projekten mit architekturgetriebenem Entwicklungsansatz verwenden. In diesem Fall wird für jede Komponente ein eigenes Modell erstellt und der Zugriff auf die Komponentenparameter mithilfe einer komponentenspezifischen Fassade gekapselt. Die systemweit gültigen Konfigurationsparameter, die ausschließlich der Architekt festlegt, werden in einem separaten Modell verwaltet und vor einer Änderung durch Komponenten geschützt.


Dr. Martin Jung *und* ***Dr. Thomas Walter*** *sind als Berater beim Erlanger Softwaredienstleister* ***develop group*** *tätig. Beide beschäftigen sich schwerpunktmäßig mit modellgetriebenen Entwicklungsmethoden und deren Einbettung in praxistaugliche Softwareentwicklungsprozesse.*
Kontakt: connect@develop-group.de

Links & Literatur

[1] JAXB Reference Implementation Project, Version 2.1.7: https://jaxb.dev.java.net/

[2] Xalan-Java, Version 2.7.1: http://xml.apache.org/xalan-j/

[3] The Obix-Framework, Version 1.1 (Stand Dezember 2006): http://obix-framework.sourceforge.net/

[4] Apache Commons Configuration, Version 1.6 (Stand Dezember 2008): http://commons.apache.org/configuration/

[5] Fassade bzw. Singleton bezeichnen die gleichnamigen Design Patterns aus Gamma, Helm, Johnson, Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley/Prentice Hall, 1995

[6] Al-Hilank, Samir; Jung, Martin: Toolunterstützung für die architekturgetriebene Entwicklung, Objektspektrum 03/2007