Configo – Projektdokumentation

Juli 2005

0 Inhaltsübersicht

1 Configo Steckbrief

Anzahl Java Klassen

- 230

Lines of Code

- ~ 14598

Lines of JET Template Code

- ~ 470

Verwendete Technologien und Werkzeuge

- **Ant** – Automatisierter Build der Configo Distribution mit Hilfe des Ant Build Tools
- **Castor** – XML-to-Java Binding-Framework
- **Eclipse JDT** – Entwicklungsumgebung für Java
- **exe4J** – Erzeugt ausführbare Dateien aus jar-Files
- **FTP-Server** – File Share
- **JDOM** – API für DOM-basierte XML-Verarbeitung
- **JUnit** – Test der RichValidator-Funktionalität über das JUnit Test-Framework
- **Log4J** – Logging
- **SAX 2 API** – API für eventbasierte XML-Verarbeitung
- **Subversion** – Versionierung der Projektdaten

2 Anforderungsanalyse / Motivation

2.1 Problematik

Professionelle Anwendungen bieten heute ein hohes Maß an Flexibilität und Skalierbarkeit. Die hierzu erforderlichen Konfigurationsdaten werden dabei zumeist in externen XML-Dateien abgelegt, da eine einfache Vorgehensweise mit Key-Value Paaren (Beispiel: Java Property-File Mechanismus) in vielen Fällen einfach nicht reicht, um eine geeignete Konfiguration abzubilden (Beispiel: Konfigurationsfile Tomcat).

Um nun ein Konfigurationsfile in XML-Form zu verarbeiten, kann ein Java-Entwickler eine XML-API, wie DOM oder SAX nutzen, jedoch sind diese Werkzeuge sehr generisch und es erfordert einiges an Logik, um die Daten einzulesen. Des Weiteren hat der Entwickler das Problem, dass er den AW-Admin durch Fehlermeldungen dabei unterstützen will, eine gültige Konfiguration zu erzeugen. Jedoch sind die Fehlermeldungen, die einem die XML-APIs zur Verfügung stellen nur eingeschränkt nutzbar. Einem Laien auf XML-Gebiet werden diese Meldungen nicht sehr viel helfen. Außerdem muss ein XML-File durch eine geeignete DTD oder ein XML-Schema File beschrieben werden. Das heißt, der AW-Entwickler muss sich erst mit diesen Verfahren beschäftigen und das liegt oft nicht in deren Interesse.

2.2 Ideen und Anforderungen

Da stellt sich doch die Frage, warum für diese Problematik nicht ein Framework existiert, da sich die Erstellung und das Einlesen einer Konfiguration für eine Anwendung doch gut in einem solchen verpacken ließe.

Aus den oben genannten Problemen haben sich für uns folgende Anforderungen an ein solches Framework herauskristallisiert:

- **Für AW-Entwickler**
    - Eine einfache Möglichkeit, ein Konfigurationsfile zu beschreiben
    - Bereitstellung einer intuitiven Programmierschnittstelle
- **Für AW-Admins**
    - Unterstützung im Fehlerfall, durch aussagekräftige Meldungen
    - Weitere Unterstützung durch Metainformationen zum Konfigurationsfile

Ausgehend von diesen Anforderungen hatten wir uns als Ziel für dieses Projekt gesetzt, ein Konzept für ein solches Framework auszuarbeiten. Darüber hinaus wollten wir eine Referenz-Implementation realisieren, welche auf diesem Konzept aufsetzt. Um diese Implementation letztendlich zu demonstrieren, nahmen wir es uns zur Aufgabe, eine passende Beispielapplikation zu entwickeln.

2.3 Zielpersonen

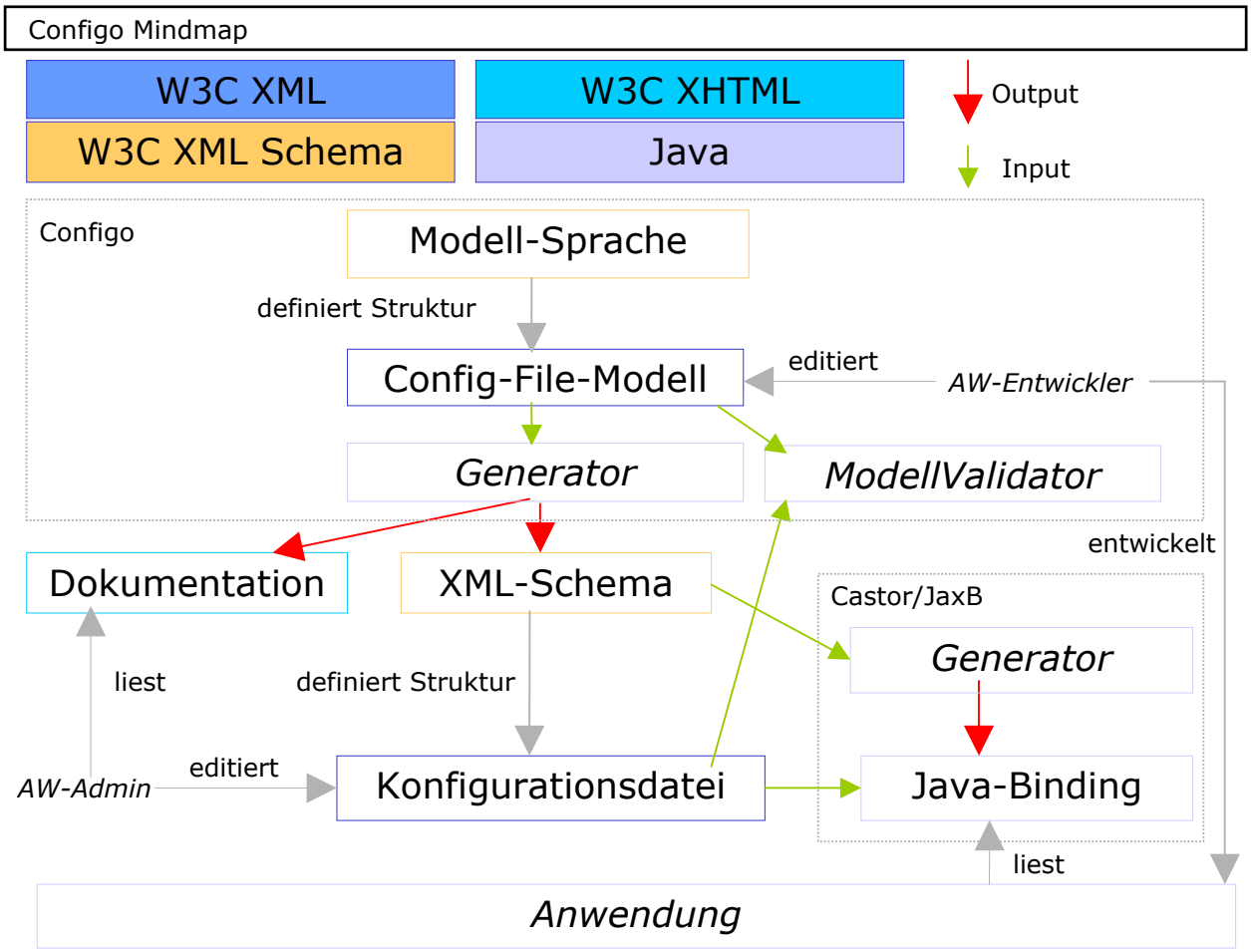
Im Folgenden wird mehrfach auf die Rollen AW-Entwickler, -Admin und -User Bezug genommen. Daher an dieser Stelle eine kurze Begriffsklärung:

- AW-Entwickler:
  Aufgaben des AW-Entwicklers ist die Konzeption und Programmierung einer Anwendung mit einem hohen Konfigurationsvolumen. AW-Entwickler stellen die eigentliche Zielgruppe des Configo Frameworks dar. (Selbstverständlich wird in größeren Entwicklungsprojekten diese Rolle weiter untergliedert. Aus Sicht des Frameworks spielt dieser Aspekt jedoch keine Rolle.)
- AW-Admin:
  Aufgabe des AW-Admins ist die Installation und Einrichtung/Konfiguration der Anwendung zwecks Benutzung durch den AW-User. Da der AW-Admin aus Sicht des AW-Entwicklers Stakeholder ist, sind dessen Anforderungen auch für Configo von hoher Bedeutung.
- AW-User:
  Der AW-User kommt mit der Anwendungs-Konfiguration über editierbare Konfigurationsdateien in diesem Szenario nicht in Berührung.

Selbstverständlich ist es möglich, dass eine Person mehrere Rollen in sich vereint. Insbesondere wird der AW-Entwickler in der Testphase des Entwicklungsprozesses auch in die Rollen des AW-Administrators (und des AW-Users) schlüpfen.

3 Die Configo Architektur

Zu Beginn des Praktikums entstand folgende Mindmap, die die Architektur des Configo Frameworks aufzeigt.



Zentraler Bestandteil der Architektur stellt die **Modell-Sprache** (die Configo Modeling Language) dar. Mittels dieser Sprache kann der AW-Entwickler ein Modell der von seiner Anwendung verwendeten **Konfigurationsdatei** erstellen. Das **Config-File-Modell** enthält hierbei sowohl die Struktur- als auch die anwendungsfachliche Beschreibung der Konfigurationsdatei. Weitere Informationen zur Configo Modeling Language sind in Kapitel 5 nachzulesen.

Aus dem Config-File-Modell wird mittels eines **Generators** eine **XML-Schema-Datei** generiert, welche lediglich die strukturelle Beschreibung der Konfigurationsdatei enthält. Erstellt oder editiert der AW-Admin seine Konfigurationsdatei mit seiner eigenen Umgebung, kann er diese XML-Schema-Datei als Hilfestellung verwenden.

Wichtiger ist jedoch die Möglichkeit, aus der XML-Schema-Datei mittels eines Generators ein **Java-Binding** zu genieren, das dem AW-Entwickler direkten Zugriff auf die Konfigurationsdatei gestattet. Configo verwendet hierzu das Castor Framework.

Darüber hinaus wird aus dem Config-File-Modell eine **Dokumentation** über die Konfigurationsdatei generiert. Diese stellt eine weitere Hilfestellung für den AW-Admin dar.

Neben der Modell-Sprache stellt der **ModellValidator** eine weitere zentrale Komponente der Configo Architektur dar. Basierend auf dem Config-File-Modell kann eine Validierung der Konfigurationsdatei stattfinden, welche dem AW-Admin im Fehlerfall verständliche Fehlermeldungen zur Verfügung stellt.

In folgenden Kapiteln werden die zentralen Komponenten näher beschrieben.

4 Die Konfigurationsdatei

4.1 Anforderungen

Folgende Anforderungen bestanden an das Konfigurationsfile:

- logische / hierarchische Strukturierung der Konfigurationsparameter
- Validierung des Konfigurationsfiles mit aussagekräftigen Fehlermeldungen
- Dokumentation zum Schema des Konfigurationsfiles

4.2 Format des Konfigurationsfiles

Der Fokus des Projektes liegt auf Anwendungen, die eine gewisse Komplexität aufweisen. Konfigurationsfiles, welche aus einfachen Key-Value-Paaren bestehen schienen uns für diesen Fall nicht geeignet. Wir standen nun vor der Überlegung, ob ein einfaches selbstdefiniertes Format eingesetzt werden sollte oder wir XML verwendeten.

Vor- und Nachteile des selbstdefinierten Formats:

+ einfachere Syntax/Struktur
+ leicht an unsere Bedürfnisse anpassbar, kein Überladen
- Erweiterbarkeit?
- keine Parser und sonstige Infrastruktur vorhanden

Vor- und Nachteile von XML:

+ gute Parser vorhanden (auch nicht validierende), Infrastruktur (z.B. Binding Frameworks wie Castor, JAX-B)
+ standardisiert, bekannt
+ Stabilität und Erweiterbarkeit
- overkill?
- Validierung genügt nicht unseren Ansprüchen

Die Vorteile von XML, speziell die Infrastruktur, die hier vorhanden ist, hat uns für XML entscheiden lassen. Die Möglichkeit, aus XML-Schema Binding-Klassen für die

Konfigurationsdatei zu generieren, führte zur Entscheidung, XML-Schema-Instanzen als Metamodelle zu verwenden.

4.3 Fehlermeldungen

Fehlermeldungen sollen dem AW-Admin über Fehler in der Konfigurationsdatei informieren und ihm eine Hilfestellung für die Behebung der Fehler geben. Hier kamen Überlegungen auf, die Fehlermeldungen von dem AW-Entwickler erstellen zu lassen. Dies ist für diesen jedoch ein bedeutender Mehraufwand. Deshalb sollten Fehlermeldungen wenn möglich vom Framework selbst zur Verfügung gestellt werden.

Um sinnvolle Fehlermeldungen generieren zu können, muss zunächst festgestellt werden, wo Fehlermöglichkeiten liegen. Betrachtet man das Konfigurationsfile als hierarchischen Baum, können die Fehler auf 2 Bereiche eingegrenzt werden:

- Verletzung des Content Models
    - Fehlende Elemente oder Attribute
    - Zu viele Elemente oder Attribute
- Blätter des Baumes
    - Hier können Fehler in der Wertevergabe gemacht werden, beispielsweise ein vorgegebener Wertebereich verletzt werden

Dies führte uns zur Überlegung, Default-Fehlermeldungen für die Content-Model-Verletzungen zu erstellen und diese im Fehlerfall mit genaueren Angaben zu füllen.

Der AW-Entwickler soll lediglich ins Spiel kommen, wenn Blätter, d.h. tatsächliche Werte im Konfigurationsfile fehlerhaft gesetzt sind. Hierfür kann er in der Beschreibung der Konfigurationsdatei mittels CML Constraints erstellen (z.B. reguläre Ausdrücke). Werden diese im Konfigurationsfile verletzt, erstellt das Framework eine entsprechende Fehlermeldung. Eine direkte Definition von Fehlermeldungen durch den AW-Entwickler findet also nicht statt.

Weitere Überlegungen, die Fehlermeldungen aussagekräftiger zu gestalten, führten zur Idee, Beispiele für gültige Werte durch den AW-Entwickler bereitstellen zu lassen. Wie dies umgesetzt wurde, ist im Abschnitt zur CML nachzulesen, sowie in der Dokumentation zur CML.

5 Die Configo Modeling Language (CML)

Die Configo Modeling Language stellt die Beschreibungssprache für Konfigurationsdateien dar und bildet somit das Herzstück von Configo.

5.1 Anforderungen

- Beschreibungssprache bietet
    - Hierarchie

    - Optionalität von Parametern
    - Typisierung von Parametern
- Einfach, intuitiv
- Dokumentation der Beschreibungssprache
- Unterstützung bei der Entwicklung des Modells
- leichter Zugriff auf Konfigurationsparameter durch AW über intuitive API

5.2 Format

XML-Schema ist sehr komplex, bietet aber keine elegante Möglichkeit, das Modell mit Metadaten anzureichern. Deshalb entschieden wir uns hier, selbst eine Sprache zu definieren. Wir entschieden uns für eine XML-basierte Sprache, da dies, wie in Kapitel 4.2 schon erläutert, einige Vorteile bringt.

Um die Sprache möglichst einfach und intuitiv zu halten und jeglichen Ballast zu vermeiden, bauten wir sie schrittweise auf. Anhand von Beispielen setzten wir zunächst die minimalen Anforderungen – wie Hierarchien – um. Später folgten Typsystem, Eindeutigkeit, Schlüssel etc. Die Sprache sollte hierbei natürlich erweiterbar gehalten werden. Dies musste auch in der Implementierung umgesetzt werden.

Hierbei entfachten einige Diskussionen, die im folgenden Abschnitt beschrieben sind. Die vorerst endgültige Version der CML ist in ihrer Dokumentation nachzulesen.

5.3 Diskussion

Content Modell

Zunächst entschieden wir uns, Elementverschachtelungen mittels Vater-Kind-Referenzierungen umzusetzen. Somit wird das Model flach gehalten und es ist leicht möglich, Parameter auszutauschen.

Die Reihenfolge der Kindelemente bereitete uns größeres Kopfzerbrechen. Nach längerer Diskussion orientierten wir uns an XML-Schema. Wir entschieden uns, „sequence“ und „choice“ auf oberster Ebene zuzulassen. Sequence schreibt die genaue Reihenfolge der Kindelemente vor. Choice ermöglicht es, eine Auswahl zwischen Kindelementen zu definieren. Sequence ist Default.

Wir entschieden uns, vorerst entweder Kindknoten oder Textinhalt zuzulassen.

Da wir ein CML-Modell in eine XML-Schema-Instanz transformieren, mussten wir einige dieser Entscheidungen von XML-Schema abhängig machen.

Kardinalität / Optionalität von Elementen und Attributen

Verwenden wir min/max oder lediglich die Möglichkeit, Elemente/Attribute als optional/required zu kennzeichnen?

Wie verhält sich dies im speziellen Fall bei Choice? Darf ein Element innerhalb einer Choice optional sein? Ist es dann möglich, keines der Kindelemente anzugeben? Diese

Fragen führten zu folgender Entscheidung: Bei Verwendung der Gruppe choice muss min > 0 sein, dies wird beim Einlesen des Modells geprüft.

Wir entschieden uns für min/max bei Elementen und für required=true/false bei Attributen, um den Zwang der Auflistung gleicher Elemente im Modell zu vermeiden.

Weitere Überlegungen mussten nun bei den Defaultwerten von min und max getroffen werden. Wir entschieden uns für den intuitivsten Weg und gingen dabei bewusst in eine andere Richtung wie XML-Schema. Wir legten folgendes fest:

- weder min noch max angegeben: min=1, max=1 (default)
- nur min angegeben: max = unbounded (default)
- nur max angegeben: min = 0 (default)

Schlüsselfunktionalität

Auch hier orientierten wir uns zunächst bei XML-Schema. Jedoch entschieden wir uns für eine abgespeckte Umsetzung, um die CML einfach zu halten.

Die Eindeutigkeit eines Parameters ist Elementtyp-global, d.h. innerhalb aller Elemente eines Typs. Wie die Schlüsselfunktionalität umgesetzt wird, ist in der CML-Dokumentation nachzulesen.

Datentypen / Typsystem

Mittels eines einfachen Typssystems will Configo dem AW-Entwickler komplexe Datentypen zur Verfügung stellen und die Möglichkeit bieten, eigene Datentypen zu entwickeln.

Nach einer Analyse wurde beschlossen, folgende Typen anzubieten:

- Basistypen wie String, Integer, Float
- Bounded Typen
- Reguläre Ausdrücke
- Auswahllisten
- User definierte um die Sprache erweiterbar zu halten

Der AW-Entwickler kann eigene Typen definieren, indem er die Basistypen einen Namen gibt und mit konkreten Werten initialisiert.

Beschreibung

Bei allen Elementen, Attributen und Datentypen wird ein Beschreibungselement erlaubt. Dieses dient zur Generierung von Dokumentation und Fehlermeldungen. Die Beschreibungen werden als Elemente an das zu beschreibende Element gehängt. Es wurde diskutiert, stattdessen ein Attribut zu verwenden, jedoch fiel die Entscheidung aus Gründen der Übersichtlichkeit des Modells bei langen Texten auf die Verwendung von Elementen.

Beispiele

Beispiele sind bei Datentypen direkt oder bei Properties erlaubt. Auch sie werden wie die Beschreibung als Elemente angehängt. Dies ermöglicht das Hinzufügen eines Kommentars zu dem Beispiel. Um den AW-Entwickler keine zusätzlichen Vorschriften zu machen, wird davon abgesehen, bei regulären Ausdrücken die Angabe eines Beispiels zu erzwingen.

Casesensitivität

Alle vergebenen Namen werden casesensitiv behandelt.

Endlosrekursion

Werden beim Parsen des CML-Modells abgefangen.

Defaultwerte

Weder für Elementinhalte noch für Attribute kann ein Defaultwert vorgegeben werden. Sollten tatsächlich Defaultwerte von Nöten sein, erstellt der AW-Entwickler ein Beispiel-Konfigurationsfile oder vermerkt diese Defaultwerte im Modell (in Beispielen oder der Beschreibung des Elements/Attributs).

Elementgruppen

Werden vorerst nicht erlaubt.

Nicht referenzierte Elemente

Sind nicht erlaubt.

Eindeutigkeit von Elementtypnamen

Elementtypnamen müssen dokumentweit eindeutig sein.

Nicht referenzierte Datentypen

Werden ignoriert.

6 Validierung von Konfigurationsdateien

Eine Kernfunktionalität von Configo besteht in der Validierung von Konfigurationsdateien gegen konkrete CML Modellinstanzen. Dieser Vorgang ist eng verwandt mit der Validierung von XML Dateien anhand einer Schemabeschreibung, wie etwa einer DTD oder eines XML Schemas. Im Folgenden sollen interessante Aspekte der Implementierung des Validators beleuchtet werden.

Eine aus unserer Sicht zentrale Anforderung an diese Validierung ist, dass erkannte Regelverletzungen mit möglichst aussagekräftigeren Fehlermeldungen behandelt werden kann. Wichtige Aspekte dabei sind:

- Lokalisierung von Fehlern in der Konfigurationsdatei
- Fortsetzung der Validierung nach einem erkennten Regelverstoß, um den Prozess der Fehlerkorrektur zu vereinfachen

6.1 Schnittstellen (1): Die Konfigurationsdatei API

Für die Validierung einer Konfigurationsdatei gegen eine konkrete Configo Modellinstanz benötigt der Configo RichValidator eine generische „readonly“ Java-Repräsentation der Datei, die deren Daten und Struktur beschreibt. Aus der Konfigurationsdatei werden hierzu alle relevanten Informationen wie Element- und Attributnamen und deren Werte, aber auch die jeweiligen Positionen in der Datei über einen SAX 2 Parser extrahiert und anschließend in Form eines Dokumentbaums ähnlich DOM zur Verfügung gestellt. Da die DOM API eine abstakte Repräsentation eines XML Dokuments darstellt, fehlen Informationen zu Positionen von Elementen im Bezug auf die Konfigurationsdatei. Aus diesem Grund kommt eine eigene Konfigurationsdatei API zum Einsatz.

Ein großer Vorteil dieser eigenen Konfigurationsdatei API ist die Möglichkeit, das Visitor Pattern [Gamma] umzusetzen, wodurch die Verarbeitung des Dokument-Baums von dessen Repräsentation getrennt wird. Ein pre-order Traversierungsmechanismus wird bereitgestellt, bei dem sich ein passender Visitor registrieren kann, um auf jedem Baumknoten eigene Operationen durchführen zu können. Kennzeichnend für das Zusammenspiel von Baumtraversierer und Visitor ist, dass der Visitor durch entsprechende Kennzeichnung von Knoten den Abstieg in Unteräste verhindern kann. Dieser Mechanismus dient der Ignorierung von Regelverletzungen, welche die Fortsetzung der Validierung behindern würden.

Anmerkung:
Die generische Konfigurationsdatei API kommt während des Validierungsprozesses zum Einsatz. Der Zugriff einer Anwendung auf die Daten Konfigurationsdatei erfolgt vorzugsweise über nicht-generische (weil generierte) Castor API. Die Tatsache dass das Konfigurationsfile dadurch zweifach geparst werden muss wird in Kauf genommen.

6.2 Schnittstellen (2): Die Configo Modell API

Zweiter Baustein der RichValidation ist eine Java-Repräsentation einer konkreten Configo Modellinstanz. Das Modell wird mit Hilfe der Configo Modelling Language (CML) in einer XML-Datei beschrieben. Die Struktur der Modell-Datei ist durch CML definiert. Da CML erst im Rahmen des Projekts entwickelt wird, erfolgt eine Aufteilung in Interface und Implementierung der Configo Modell API. Dadurch wird der gegen die API programmierte Code von syntaktischen Änderungen der CML unabhängig. (Durch diese Trennung würde selbst der Wechsel von einer XML-basierten CML zu einer propritären Syntax ohne Anpassung von Validatoren und Generatoren ermöglicht). Im Rahmen des Projekts wird eine Implementation der Modell API für eine XML-basierte CML bereitgestellt.

Es ist wichtig, sich den Unterschied der beiden beschriebenen APIs zu vergegenwärtigen. Während die Konfigurationsdatei API die Konfigurationsdatei an sich repräsentiert, ist die Configo Modell API als Repräsentation des durch die CML-Datei beschriebenen Configo Modellinstanz aufzufassen. Als Konsequenz ergibt sich, dass die Modell API auch Zugriff auf in der Datei nur implizit enthaltene Informationen ermöglicht, und beispielsweise nicht nur benutzerdefinierte, sondern auch vordefinierte Configo Datentypen enthält. Ein anderes Beispiel ist das Verhalten von BoundedTypes bei der Angabe von nur einer Intervallgrenze. Dieses Verhalten muss nur an einer Stelle implementiert werden. Generatoren und Validatoren können sich gleichsam auf die Configo Modell API stützen, alle Constraints der Configo Modellinstanz explizit bereitstellt.

Daraus ergibt sich darüber hinaus die Notwendigkeit, die Modell-Datei vor dem Aufbau des Java Modells zu validieren. Diese Validierung erfolgt, soweit möglich, mit Hilfe der XML-Schema Definition der CML, komplexere Abhängigkeiten werden mit Java geprüft.

Die Modell API wird vom RichValidator und von allen Configo Generatoren verwendet. Sie stellt die einzige Schnittstelle zu den Daten des mit CML beschriebenen Modells dar.

6.3 Das CML-Validierungswerkzeug RichValidator

Der RichValidator ermöglicht die Validierung einer Configo Konfigurations-Datei gegen ein konkretes Configo Modell. Dabei kommen die oben beschriebenen APIs für die Konfigurationsdatei und das Configo Modell zum Einsatz. Zentrales Merkmal des RichValidators ist die Verwendung des Listener Patterns [Gamma]. Damit definiert das Configo Framework einen Erweiterungspunkt, der durch beliebige Anwendungen genutzt werden kann. Somit ist beispielsweise die Integration des RichValidators mit einem Eclipse JDT Plugin prinzipiell möglich (siehe auch Kapitel 10). Im Rahmen des Projekts wird ein einfacher Listener bereitgestellt, der die Validierungsmeldungen lediglich auf Konsole ausgibt (SimpleErrorPrinter).

6.4 Ein Erweiterungspunkt: Benutzerdefinierte Typvalidatoren

Das Typsystem von Configo definiert einen zusätzlichen Erweiterungspunkt des Configo-Frameworks. Für jeden in CML „eingebauten“ Basisdatentyp, der in einer Modellinstanz deklariert und dabei mit konkreten Werten (z.B. der Intervallgrenzen für BoundedTypes) initialisiert werden kann, stellt Configo einen sogenannten TypeValidator zur Verfügung. Jeder TypeValidator definiert zusammen mit der konkreten Initalisierung der Werte bei der Deklaration im Modell das Verhalten bei der Validierung. Nun ist davon auszugehen, dass mit den Configo Basisdatentypen nicht alle Anforderungen an die Gültigkeitsprüfung von Konfigurationsparametern abgedeckt werden können. So ist es beispielsweise eine recht schwierige Aufgabe eine e-Mail Adresse mit Hilfe eines regulären Ausdrucks korrekt auf Gültigkeit zu überprüfen, weil hier bereits recht komplexe Regeln gelten. Ein anderes Beispiel wäre die Prüfung einer Kontonummer auf innere Konsistenz. Aus diesem Grund definiert Configo den Basistyp „userType“, der zur Validierung eine per Reflection zur Laufzeit geladene (vom Anwendungsentwickler bereitgestellte) Klasse zur Validierung heranzieht. Einzige Voraussetzung an diese Klasse ist, dass sie das selbe Interface implementiert, wie die „eingebauten“

TypeValidators. Dank dieses Mechanismus können eigendefinierte TypeValidators in den Validierungsprozess eingebunden werden.

Eine Einschränkung des TypeValidator-Konzepts ist, dass Konfigurationsparameter nur auf innere Konsistenz geprüft werden können. Dabei wäre eine Überprüfung von Abhängigkeiten zwischen Konfigurationsparameter sinnvoll und wünschenswert. Beispielsweise könnte geprüft werden, ob eine angegebene Kontonummer zu einer angegebenen BLZ passt (siehe auch Kapitel 10).

7 Der Schema-Generator

Ein Einsatzzweck von Configo ist die Bereitstellung einer direkten Zugriffsschicht (Binding) auf die Konfigurationsdatei für den AW-Entwickler. Hier existieren einige Frameworks, die aus XML-Schema-Instanzen Binding Frameworks generieren. Um diese Generatoren nutzen zu können muss jedoch eine XML-Schema-Datei vorliegen. Deshalb generiert Configo die benötigte XML-Schema-Datei aus dem CML-Modell der Konfigurationsdatei.

Dies geschieht mittels einer Java-Klasse, die das CML-Modell in eine XML-Schema-Instanz transformiert. Vor der Implementierung wurde diskutiert, ob XSL geeigneter für diese Transformation ist. Jedoch wurde im Zuge Configos eine Modell-API implementiert, die direkten Zugriff auf die Struktur des CML-Modells bietet. Mögliche Änderungen müssen somit nur in der Implementierung der Modell-API nachgezogen werden. Dies war der Grund für die Entscheidung zu Gunsten einer Java-Implementierung.

Ein positiver Nebeneffekt der Generierung der XML-Schema-Datei ist, dass die Validierung der Konfigurationsdatei mit „herkömmlichen“ XML-Validatoren ermöglicht wird.

8 Der Doku-Generator

8.1 Beschreibung

Damit der AW-Admin nicht die Modellierungssprache CML lernen muss, um korrekte Konfigurationsdateien erzeugen zu können, wurde ein Generator entwickelt, der die Informationen aus konkreten Modell-Instanzen durch Transformation ins HTML-Format visualisieren kann. Das hierbei erzeugte Dokument kann dem AW-Admin als Dokumentation zum Konfigurationsfile bereitgestellt werden.

Aus der Tatsache, dass die Dokumentation direkt aus dem Modell generiert wird, ergibt sich der Vorteil, dass die Dokumentation leichter auf aktuellem Stand gehalten werden kann, als beispielsweise ein manuell erstelltes Dokument.
In gewisser Weise besteht hier eine Ähnlichkeit zur Generierung von Java Quellcode-Dokumentationen mit JavaDoc. Der Erfolg von JavaDoc ist ein wichtiges Indiz für die Bedeutung dieses Nutzens.

Die erzeugte HTML-Seite gliedert sich inhaltlich in 4 Teile:

- Ganz oben kommt zunächst der Kopf mit allgemeinen Informationen und der Hauptnavigation.

- Die Struktur (XML-Struktur) des Files wird unter dem Punkt „Document Structure“ dargestellt, wodurch die Abhängigkeiten, also Referenzen auf Kind-Elemente oder Fremdverweise auf einen Blick sichtbar sind und auch alle Attribute des jeweiligen Element werden dort angezeigt. Diese Darstellung soll dem AW-Admin eine Hilfestellung geben, um die Verschachtelungen und Zugehörigkeiten zu verstehen.

- Im dritten Teil der Dokumentation werden die einzelnen Elemente und Attribute im Detail beschrieben. Für jedes Element werden die Regeln für das Auftreten von Kind-Elementen beschrieben (Content Model). Wurde dem Element eine anwendungsfachliche Beschreibung angefügt, so wird diese wiedergegeben. Sind Attribute vorhanden, so werden deren technischen und anwendungsfachlichen Metadaten in der Dokumentation visualisiert (Datentyp, Beschreibung, Beispiele). Gleiches gilt für Elemente mit Textinhalt (Data Elements).

- Zudem gibt es am Ende des HTML-Dokuments noch eine kleine Legende, in der wichtige Schlüsselwörter (z.B. unique) in ihrer Funktion beschrieben werden.

Vielfach angebrachte, interne Verweise unterstützen bei der Navigation durch die Dokumentation.

8.2 Implementation

Wie oben schon angesprochen, verwendet der Doku-Generator die Modell API, um das CML-Schema zu verarbeiten. Der Baum von Elementen wird anhand der API durchlaufen, wobei jedes Element nur einmal verarbeitet wird.

Die Darstellung geschieht durch JET-Templates, damit man eine Trennung von Darstellungs- und Logikschicht erhält. Zu Anfang war geplant, dass der AW-Entwickler die Templates an seine Vorstellungen anpassen kann. Jedoch ist dieser Code auch nicht sehr einfach, da es einige Iteratoren, etc. zu verarbeiten gibt. Aus diesem Grund haben wir uns auf statische Templates geeinigt.

9 Die Beispielanwendung – der Birthday Manager

9.1 Beschreibung

Der BirthdayManager ist ein kleines Tool, welches den Benutzer (hier auch gleichzeitig AW-Admin) dabei unterstützen soll, die Geburtstage von Freunden, Verwandten oder auch Kollegen nicht zu vergessen. Es werden dem Benutzer die kommenden und vergangenen Geburtstage in schöner Weise angezeigt, wodurch man immer den Überblick über diese wichtigen Ereignisse behält.

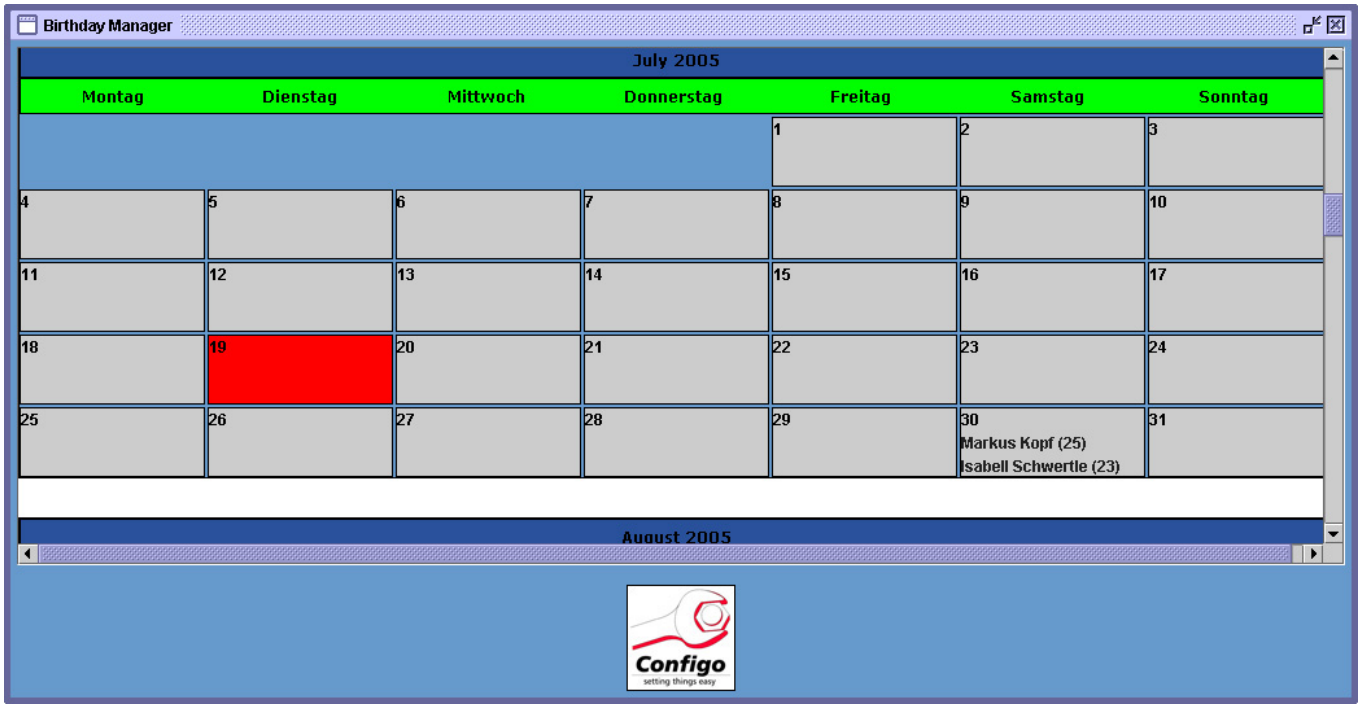
Dabei hat der Benutzer die Möglichkeit, die Daten (Geburtsdatum, Name, etc.) der einzelnen Personen in einer passenden Konfigurationsdatei abzulegen und die Personen

definierten Gruppen zuzuteilen. Des Weiteren werden in der Konfigurationsdatei Werte für die Darstellung der Geburtstage und für die zeitlichen Einschränkungen für die angezeigten Geburtstage festgelegt.

Der Benutzer hat hierbei die Wahl zwischen einer Kalender- oder einer Listenansicht der Geburtstage. Entscheidet sich der Benutzer für den Kalender, muss der Start- und Endtag des Kalenders festgelegt werden. Nach Start der Applikation präsentiert sich dem Benutzer dann ein schicker Kalender, in dem die Geburtstage der Personen eingetragen sind. Wird wiederum in der Konfiguration die Liste aktiviert, muss die Anzahl der vorhergehenden und folgenden Geburtstage angegeben werden. Hinter dieser Option verbirgt sich eine einfache Liste, welche die eingegrenzten Geburtstage enthält.

Macht der Benutzer Fehler in der Konfigurationsdatei, werden ihm diese Fehler mit leicht verständlichen Fehlermeldungen aufgezeigt und die Applikation wird nicht weiter ausgeführt.

Zu Beginn des Projekts hatten wir geplant, dass der BirthdayManager als Hintergrundprozess in Windows läuft, um Benachrichtigungen zu erzeugen, wenn ein Geburtstag bestimmter Gruppen näher rückt. Jedoch stellte sich schnell heraus, dass diese Idee aus Zeitgründen nicht realisierbar war.



9.2 *Vorgehen bei der Entwicklung*

Die Überlegungen zu der Beispielapplikation begannen schon sehr bald nach Projektstart. Das Ziel für eine Beispielanwendung war es, möglichst viele Daten in das Konfigurationsfile zu packen, um die Möglichkeiten unseres Frameworks Configo aufzeigen zu können. Nach einigen Ideen kristallisierte sich dann der BirthdayManager als Beispielapplikation heraus, da die weiterführenden Vorschläge sehr viel versprechend im Bezug auf ein Konfigurationsfile waren.

Wir haben dann die Implementierung in die folgenden 3 Schritte aufgeteilt:

- In einem ersten Schritt diente ein Java Property-File als Datenquelle, damit bereits ein Großteil der Funktionalitäten des BirthdayManagers erstmal umgesetzt werden konnte. Wobei das Konfigurationsfile inhaltlich bereits alle Ideen, die wir anfangs gesammelt hatten enthielt.

- In der zweiten Iterationsstufe stand ein Konfigurationsfile in XML-Format bereit, welches über, von Castor generierten Klassen von der Applikation ausgelesen wurde. Zu diesem Zeitpunkt waren die Spezifikationen der CML noch nicht vollständig abgeschlossen. Deshalb war diese Struktur nur vorläufig.

- Im letzten Schritt wird nun das Konfigurationsfile durch ein gültiges CML-Modell beschrieben und die Applikation greift über die Configo Schnittstelle auf die Konfiguration zu.

9.3 Was wird hierbei durch Configo gewonnen?

Würde ein Entwickler versuchen, diese Daten in einem Java Property-File zu halten, müsste er viel Logik in die Anwendung stecken, um die gleichen Regeln auf das File überprüfen zu können. Zum Beispiel gibt es in der Anwendung die weiter oben angesprochenen Ansichten (Kalender/Liste) und für jede dieser Ansichten gibt es jeweils unterschiedliche Parameter, um die Geburtstage zeitlich einzuschränken. Diese Paramter dürfen jeweils nur unter Angabe der passenden Ansicht auftauchen. Damit sind die Möglichkeiten der Java Property-Files bereits erschöpft. Es wäre möglich, diese Technik für eine solche Anwendung einzusetzen, der Aufwand für den Entwickler jedoch wäre um ein vielfaches höher.

Mithilfe eines XML-Files in Kombination mit einer passenden DTD oder eines XML-Schema Files wäre es für einen Entwickler möglich, die gewünschten Regeln abzubilden, jedoch würden diese Möglichkeiten wieder einige Probleme mit sich bringen. Zum einen müsste der Entwickler über eine geeignete XML-API, zum Beispiel DOM oder SAX das XML Dokument einlesen, was nicht sehr intuitiv ist und zum anderen sind die Fehlermeldungen dieser APIs nur schlecht verwertbar. Diese sind für einen Benutzer, der sich nicht besonders mit XML auskennt, nicht sonderlich hilfreich. Diese Problematik löst Configo, indem es eine intuitive API zur Verfügung stellt und sehr schöne Fehlermeldungen produziert, die dann auch auf einfache Weise in der Applikation verarbeitet werden können.

10 Ideen für Nachfolgeprojekte

10.1 Datenbankbasierte Anwendungskonfiguration mit Configo

Die „Configo Modelling Language“ (CML) beschreibt zum jetzigen Zeitpunkt Anwendungskonfigurationsdaten, die in XML-Dateien abgelegt werden. Die Ablage von Konfigurationsdaten im lokalen Dateisystem ist jedoch auch mit Einschränkungen verbunden. Eine XML-Datei, wie sie durch CML beschrieben wird, lässt sich auch als eine Menge geschachtelter Container auffassen, die jeweils Properties, also Key-Value-Paare, besitzen. Properties sind dabei Attribute sowie der Inhalt reiner „Daten Elemente“,

das sind Elemente, die nur Text und keine weiteren Elemente enthalten. Eine solche Datenstruktur könnte auch ohne weiteres in einer Datenbank abgebildet werden. Aus Sicht der Anwendung könnte daher das Configo-Framework über die physikalische Ablage der Konfigurationsdaten abstrahieren, oder zusätzliche Features anbieten.

10.2 Integration von Configo in Eclipse

Ein weiteres spannendes Projekt stellt aus unserer Sicht die Integration von Configo in die Java Entwicklungsumgebung Eclipse dar. Die Werkzeuge für Anwendungsentwickler sind stand heute alle kommandozeilenbasierte Tools. Dabei könnten verschiedene Erweiterungspunkte von Eclipse gezielt genutzt werden, um die Anwendungsentwicklung mit Configo besser zu unterstützen. So wäre z.B. eine spezialisierte „Configo Nature“ denkbar und wünschenswert, welche die verschiedenen Generierungsschritte aus einer CML Modellinstanz im Rahmen des „normalen“ Project Builds automatisiert. Andere Anknüpfungspunkte wäre die Bereitstellung von Editoren, für die Entwicklung von CML Modellinstanzen oder für die Arbeit mit XML Konfigurationsdateien unter Verwendung des Configo Validierungsmechanismus gegen CML Modellinstanzen.

10.3 Formulierung von XML Covarianzen mit Configo

Die Formulierung von Covarianzen, also von komplexeren Constraints für XML Dateien bereitet heute große Schwierigkeiten. Selbst mit umfangreichen Schemabeschreibungssprachen wie XML Schema stößt man mit Anforderungen aus der Praxis häufig an Grenzen. Ein Beispiel für eine Covarianz wäre, wenn die Gültigkeit des Inhalt eines Attributes vom Inhalt eines anderen Attributes im Dokument abhängt, oder der Inhalt eines Attributs das Content-Model des betroffenen Elements (also z.B. gültige Kindelemente) bestimmt. Solche Anforderungen sind nach unserer Einschätzung auch im Bereich der Anwendungskonfiguration von brennender Bedeutung. Wo komplexen, generischen Beschreibungssprachen buchstäblich „die Luft ausgeht“, da wäre zu evaluieren, ob mit dem generativen Ansatz von Configo nicht ein großer Mehrwert geschaffen werden könnte. Im Rahmen dieses High-End Projekts könnte der Versuch unternommen werden, die Configo Modellierungssprache CML um Covarianzen zu erweitern und die Validierungsmechanismen von Configo entsprechend auszubauen.