

# Objektorientierte Konstruktion weiterer Bausteine

Wir haben bereits viele Merkmale gesehen, die von objektorientierten Sprachen ausgeborgt sind. In diesem Kapitel sehen wir uns Ersetzungsgruppen (die Unterklassen ähneln), abstrakte Elemente und Datentypen (die abstrakten Klassen ähneln) sowie finale Datentypen (die finalen Klassen ähneln) an.

## Ersetzungsgruppen

In vielen Fällen muß ein Vokabular in der Lage sein, ganz verschiedene Inhaltsmodelle zu akzeptieren. Wir haben hierfür zwei Möglichkeiten: Wir können versuchen, einen einzigen generischen Elementnamen zu verwenden, oder wir können ein Schema definieren, das intelligent genug ist, mit dem möglichen Inhaltsmodell zurechtzukommen. Da wir (wegen der Regel von der konsistenten Deklaration) nicht mehrere verschiedene Inhaltsmodelle für dasselbe Element definieren können, haben wir entweder die Möglichkeit,  `xsi:type` -Attribute in den Instanzdokumenten zu verwenden oder aber ein Inhaltsmodell zu definieren, das breit genug gehalten ist, um alle Möglichkeiten unterzubringen. Solch ein Modell wäre wahrscheinlich breit genug, auch unerwünschte Kombinationen zu akzeptieren.

Die einfachste Lösung, mit *W3C XML Schema* verschiedene Typen unterzubringen, besteht darin, für jeden Fall einen anderen Elementnamen zu verwenden. Wir haben bereits gesehen, daß der Kompositor  `xs:choice`  (außerhalb einer Gruppe) es erlaubt, Konstruktionen aufzubauen, bei denen ein Knoten in einem Instanzdokument ein aus einer Liste gewähltes Element akzeptiert. Diese Liste ist jedoch in der Definition des komplexen Typs festgeschrieben. Wir haben auch erörtert, daß diese Liste nicht erweitert werden kann, denn die Regeln für Ableitungen komplexer Typen durch Erweiterung lassen dies nicht zu. Ersetzungsgruppen bieten einen flexiblen Weg, einen Kompositor  `xs:choice`  (außerhalb einer Gruppe) aus einzelnen Elementen oder Verweisen zu konstruieren und ihn auch zu erweitern. Einfacher ausgedrückt, handelt es sich um Listen von Elementen, die untereinander austauschbar innerhalb von Dokumenten verwendet wer-

den können. Eine wichtige Einschränkung soll jedoch erwähnt werden, bevor wir beginnen: Ersetzungsgruppen sind nur bei globalen Elementen anwendbar.

Ersetzungsgruppen können als erweiterbare Elementgruppen angesehen werden. Bevor wir sie besprechen, sehen wir uns noch einmal die »traditionellen« Elementgruppen an, damit wir die Unterschiede zwischen diesen beiden Begriffen hervorheben können. Da die Empfehlung in bezug auf die Erweiterbarkeit von Elementgruppen und die Beschränkung von Ersetzungsgruppen besonders unscharf ist, habe ich es vorgezogen, eine konservative Interpretation zu geben, die keine Interoperabilitätsprobleme aufwerfen sollte. Ich werde die verschiedenen Interpretationen am Ende dieses Kapitels erörtern.

## Verwendung einer »traditionellen« Gruppe

Kehren wir zu Definition eines Namens zurück. (Schließlich sind universelle Namen eines der kontroversesten Themen im Bereich der Normalisierung. Daher ist es keine Überraschung, daß sie gute Beispiele abgeben.) Statt mit Datentypen zu spielen, können wir einfach verschiedene Elementnamen verwenden. Wir können dann sagen, daß ein Name entweder ein einfacher Name ist wie

```
<simple-name>
  Snoopy
</simple-name>
```

oder aber ein vollständiger Name wie

```
<full-name>
  <last>
    Schulz
  </last>
  <first>
    Charles
  </first>
  <middle>
    M.
  </middle>
</full-name>
```

Wir wissen bereits, wie wir ein flexibles Schema definieren können, das auf diese beiden Dokumente paßt. Es ist eine gute Idee, eine Gruppe mit einem `xs:choice`-Kompositor zu erzeugen, der eines dieser beiden Elemente zuläßt und in sämtlichen Elementen wieder verwendet werden kann, in denen ein Name angegeben werden muß. Die logischen Schritte sind die Definition der beiden Elemente (`full-name` und `simple-name`), die Erzeugung einer Gruppe und die Verwendung in der Definition der Elemente `author` und `character`:

```
<xs:element name="full-name">
  <xs:complexType>
    <xs:all>
      <xs:element name="first" type="string32" minOccurs="0"/>
```

```

        <xs:element name="middle" type="string32" minOccurs="0"/>
        <xs:element name="last" type="string32"/>
    </xs:all>
</xs:complexType>
</xs:element>

<xs:element name="simple-name" type="string32"/>

<xs:group name="name">
    <xs:choice>
        <xs:element ref="simple-name"/>
        <xs:element ref="full-name"/>
    </xs:choice>
</xs:group>

<xs:element name="author">
    <xs:complexType>
        <xs:sequence>
            <xs:group ref="name"/>
            <xs:element ref="born"/>
            <xs:element ref="dead" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute ref="id"/>
    </xs:complexType>
</xs:element>

<xs:element name="character">
    <xs:complexType>
        <xs:sequence>
            <xs:group ref="name"/>
            <xs:element ref="born"/>
            <xs:element ref="qualification"/>
        </xs:sequence>
        <xs:attribute ref="id"/>
    </xs:complexType>
</xs:element>

```

Wir können in diesem Fall `xs:all` verwenden, weil die betroffenen Elemente in dem Element `full-name` isoliert sind. An dieser Stelle sollte auch darauf hingewiesen werden, daß `xs:all` nicht bedeutet, daß es auf die Reihenfolge nicht ankommt, sondern nur, daß alle Kombinationen gültig sind. In diesem Fall kann

```

<full-name>
  <first>
    Eric
  </first>
  <last>
    van der Vlist
  </last>
</full-name>

```

bzw.

```
<full-name>
  <last>
    van der Vlist
  </last>
  <first>
    Eric
  </first>
</full-name>
```

vielleicht ausdrücken, ob ich es vorziehe, »Eric van der Vlist« oder aber »van der Vlist Eric« genannt zu werden. Anwendungen, die Zugang zu den Komponenten dieses full-name haben wollen, können ihn problemlos haben, nur müssen diejenigen, die einen full-name brauchen, die Reihenfolge im Dokument respektieren.

## Ersetzungsgruppen

### Verwendung von Ersetzungsgruppen

Wie können wir nun dasselbe Inhaltsmodell mit Hilfe von Ersetzungsgruppen definieren? Das erste, was zu tun ist, ist die Definition eines Elements, von dem sowohl full-name als auch simple-name abgeleitet werden kann. In diesem Fall haben wir einerseits einen einfachen Typ, andererseits einen komplexen Typ komplexen Inhalts. Daher können wir keinen Typ finden, der zu beiden erweitert werden kann. Wir haben keine Wahl, wir müssen mit dem universellen Typ beginnen, der jedes Inhaltsmodell akzeptiert. Dieser ganz besondere Typ namens xs:anyType ist auch der Standardwert, wenn kein Typ angegeben ist. Wir können ein generisches name-Element definieren, ohne irgendeine Typdefinition zu geben, um es so offen wie möglich zu halten:

```
<xs:element name="name"/>
```

Dieses Element wird zu dem werden, was man den Kopf der Ersetzungsgruppe nennt. Ohne in diesem Kopfelement etwas weiteres anzugeben, können andere Elemente erklären, daß sie überall dort verwendet werden können, wo das Kopfelement in dem Schema referenziert wird. Diese Elemente werden Mitglieder der Ersetzungsgruppe genannt. Die einzige Einschränkung für die Mitglieder liegt darin, daß ihre Typen gültige Ableitungen des Kopfelementtyps sein müssen. Diese Erklärung wird mit einem substitutionGroup-Attribut abgegeben, das in jedem austauschbaren Element das Kopfelement referenziert – beispielsweise so:

```
<xs:element name="simple-name" type="string32" substitutionGroup="name"/>
```

```
<xs:element name="full-name" substitutionGroup="name">
  <xs:complexType>
    <xs:all>
      <xs:element name="first" type="string32" minOccurs="0"/>
      <xs:element name="middle" type="string32" minOccurs="0"/>
```

```

        <xs:element name="last" type="string32"/>
    </xs:all>
</xs:complexType>
</xs:element>

```

Diese Deklarationen bewirken, daß die beiden Elemente überall dort verwendet werden können, wo immer ihr Kopf in dem Schema genannt wird, beispielsweise in der Definition der Elemente `character` und `author`:

```

<xs:element name="character">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name"/>
      <xs:element ref="born"/>
      <xs:element ref="qualification"/>
    </xs:sequence>
    <xs:attribute ref="id"/>
  </xs:complexType>
</xs:element>

<xs:element name="author">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name"/>
      <xs:element ref="born"/>
      <xs:element ref="dead" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute ref="id"/>
  </xs:complexType>
</xs:element>

```

## Abstrakte Elemente

Wenn wir unser Schema so belassen, wie wir es gerade gesehen haben, ist auch die Verwendung des Kopfes in den Instanzdokumenten erlaubt. Da unser Kopfelement beliebige Inhalte zuläßt, ist dies wahrscheinlich nicht das, was wir wollen. Wir benötigen einen Mechanismus ähnlich dem der abstrakten Typen, den wir gesehen haben, als wir in Kapitel 7 der gleichen Art von Problemen mit `xsi:type` begegnet sind. Wir werden das Kopfelement mit Hilfe des Attributs `abstract` in der Definition des Kopfelements als abstrakt definieren. Damit sieht die Definition so aus:

```

<xs:element name="name" abstract="true"/>

```

## Bäume von Ersetzungsgruppen

Was, wenn unsere deutsche Niederlassung ein Element `composed-name` definiert, die dem `full-name` ähnlich ist, jedoch ohne das Unterelement `middle`? Wir können dieses Element einfach unmittelbar unserer Ersetzungsgruppe hinzufügen. Wenn wir so jedoch das Element `name` als Kopf definieren, werden die Ähnlichkeiten zwischen diesem neuen Element und dem Element `full-name` nicht ersichtlich. Außerdem müssen einige Anwendungen

vielleicht angeben, daß sie entweder full-name oder aber composed-name akzeptieren. Die Lösung besteht darin, full-name als Kopf einer neuen Ersetzungsgruppe zu verwenden. Dafür müssen wir den Typ des Elements full-name als global definieren, um die ausdrückliche Ableitung zwischen den beiden Elementen zu zeigen:

```
<xs:complexType name="full-name-type">
  <xs:all>
    <xs:element name="first" type="string32" minOccurs="0"/>
    <xs:element name="middle" type="string32" minOccurs="0"/>
    <xs:element name="last" type="string32"/>
  </xs:all>
</xs:complexType>

<xs:element name="full-name" substitutionGroup="name"
  type="full-name-type"/>

<xs:element name="composed-name" substitutionGroup="full-name">
  <xs:complexType>
    <xs:complexContent>
      <xs:restriction base="full-name-type">
        <xs:all>
          <xs:element name="first" type="string32" minOccurs="0"/>
          <xs:element name="last" type="string32"/>
        </xs:all>
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

Wir haben jetzt nicht nur zwei Ersetzungsgruppen (mit name bzw. full-name als Köpfen) definiert, sondern auch einen Baum von Ersetzungsgruppen, da die erlaubten Ersetzungen für name sowohl full-name als auch simple-name umfassen, zusätzlich jedoch composed-name.

## Traditionelle Deklarationen oder Ersetzungsgruppen?

Wenn wir die beiden Lösungen, mit denen wir ein und dasselbe Problem gelöst haben, noch einmal betrachten, sehen wir, daß Ersetzungsgruppen besser erweiterbar als traditionelle Gruppen mit einem xs:choice-Kompositor sind. Während von der Elementgruppe nur durch Einbindung mit xs:redefine abgeleitet werden kann, kann die Ersetzungsgruppe mit neuen möglichen Elementen erweitert werden, indem man sie einfach definiert. Diese Elemente können in jedem beliebigen Namensraum definiert werden. Die einzige Einschränkung ist, daß ihr Typ der gleiche wie der des Kopfelements oder zumindest eine gültige Ableitung davon sein muß. (Diese Einschränkung ist gerechtfertigt, um sicherzustellen, daß Anwendungen durch unerwartete Inhaltsmodelle nicht allzu stark überrascht werden.)

Man sollte jedoch noch einen Unterschied anmerken. Wir haben gesehen, daß die Ableitung eines Inhaltsmodells mit einem `xs:choice`-Kompositor den Umfang der Auswahl nicht erweitern kann, indem weitere Alternativen hinzugefügt werden. Die Lage bei Ersetzungsgruppen ist beinahe umgekehrt. Auch wenn die Recommendation besagt, daß Ersetzungsgruppen als Auswahlen validiert werden sollten, definiert sie nicht die Reihenfolge der Elemente in der äquivalenten Auswahl. Ich empfehle, Ersetzungsgruppen in der Praxis nicht einzuschränken, da dies zu Problemen der Interoperabilität zwischen den einzelnen Schema-Prozessoren führen kann.

Wir haben also die paradoxe Situation, daß einer der beiden Mechanismen (`xs:choice`) nur eingeschränkt werden kann, während der andere (Ersetzungsgruppen) nur erweitert werden kann, und daß die Recommendation dennoch feststellt, daß diese beiden Mechanismen äquivalent sind, was die Validierung angeht. Diese Eigenschaft muß in Betracht gezogen werden, wenn man zwischen den beiden Ansätzen wählt.

Die Unterschiede zwischen den beiden Konstruktionen sind in Tabelle 12-1 zusammengefaßt. »Nicht empfehlenswert« steht für »kann mit einigen Schema-Prozessoren funktionieren, beruht jedoch auf einer freien Interpretation der Recommendation, was zu Interoperabilitätsproblemen führen kann.«

Tabelle 12-1: Elementgruppen und Ersetzungsgruppen im Vergleich

| Konstruktion   | Elementgruppen mit dem Kompositor <code>xs:choice</code> (außerhalb einer Gruppe)                               | Ersetzungsgruppen   |
|--|---|---|
| <b>Definition</b>  | zentralisiert, verwendet <code>xs:group</code> (Definition) und <code>xs:choice</code> (außerhalb einer Gruppe) | über die einzelnen Definitionen globaler Elemente verteilt; verwendet das Attribut <code>substitutionGroup</code> |
| <b>Beschränkungen für die Auswahlmöglichkeiten</b>                   | keine Beschränkungen; die Elemente können komplett unterschiedlich sein   | der Typ der Elemente muß eine explizite Ableitung des Kopftyps sein   |
| <b>Läßt globale Elemente zu</b>                                      | ja  | ja  |
| <b>Läßt lokale Elemente zu</b>                                       | ja  | nein  |
| <b>Einschränkung, um Auswahlmöglichkeiten zu entfernen</b>           | ja, mit <code>xs:redefine</code>  | nicht empfehlenswert  |
| <b>Erweiterung, um Auswahlmöglichkeiten hinzuzufügen</b>             | nicht empfehlenswert  | ja, durch Hinzufügen neuer Elemente mit demselben Kopfelement   |
| <b>Erweiterung, um neue Elemente in der Reihenfolge hinzuzufügen</b> | ja, mit <code>xs:redefine</code>  | nein  |

## Unschärfe in der Recommendation

Sowohl die Erweiterung von `xs:choice` durch Elementgruppen-Redefinitionen als auch die Einschränkung von Ersetzungsgruppen sind in der Recommendation sehr unscharf dargestellt und bedürfen der Erläuterung.

### Erweiterung von `xs:choice` durch Gruppen-Redefinitionen

Wenden wir uns nochmals unserer wie folgt definierten Gruppe zu:

```
<xs:group name="name">
  <xs:choice>
    <xs:element ref="simple-name"/>
    <xs:element ref="full-name"/>
  </xs:choice>
</xs:group>
```

Es scheint in der Empfehlung nichts zu geben, was einer Redefinition dieser Gruppe ausdrücklich verbietet, der Auswahl ein anderes Element etwa so hinzuzufügen:

```
<xs:redefine schemaLocation="foo.xsd">
  <xs:group name="name">
    <xs:choice>
      <xs:group ref="name"/>
      <xs:element ref="bar"/>
    </xs:choice>
  </xs:group>
</xs:redefine>
```

Das Ergebnis dieser Redefinition wäre jedoch, daß ein neues Element (`bar`) anstelle von `simple-name` und `full-name` akzeptiert würde. Das wäre hübsch. Allerdings sind die Grundsätze einer Redefinition durch *Einschränkung* (d.h., wenn der Inhalt der Gruppe während einer Redefinition eingeschränkt wird) dieselben wie die der Ableitung komplexer Typen durch Einschränkung. Wir können daraus schließen, daß die Working Group auch die Absicht hat, die Eigenschaften von Redefinitionen durch *Erweiterung* entsprechend der Ableitung komplexer Typen durch Erweiterung festzulegen, und diese verbietet das Hinzufügen neuer Partikeln bei `xs:choice` (außerhalb einer Gruppe) ausdrücklich.

Auch wenn einige Schema-Prozessoren diese Konstruktion unterstützen und manche Spezialisten das gut finden, rate ich nicht dazu, es zu verwenden, da es die Absicht (wenn auch nicht den Wortlaut) der Empfehlung zu verletzen scheint.

### Einschränkung von Ersetzungsgruppen

Die Einschränkung der Ersetzungsgruppen ist dem ziemlich entgegengesetzt. Die Absicht der Working Group scheint zu sein, solche Einschränkungen zuzulassen, wobei der Wortlaut der Recommendation das Ergebnis undefiniert läßt.

Die Empfehlung legt klar fest, daß Ersetzungsgruppen bei der Prüfung, ob eine Partikel eine gültige Einschränkung einer anderen Partikel ist, wie `xs:choice` behandelt werden sollten. Dies ist ein klarer Hinweis darauf, daß Ersetzungsgruppen wie bei der Ableitung komplexer Typen durch Einschränkung ebenfalls eingeschränkt werden können. Um dies zu veranschaulichen, nehmen wir die Definition für den komplexen Typ des Elements `author`, die die Ersetzungsgruppe mit dem Kopf `name` verwendet, wie wir es oben festgelegt hatten:

```
<xs:complexType name="authorType">
  <xs:sequence>
    <xs:choice>
      <xs:element ref="name"/>
      <xs:element ref="simple-name"/>
      <xs:element ref="full-name"/>
    </xs:choice>
    <xs:element ref="born"/>
    <xs:element ref="dead" minOccurs="0"/>
  </xs:sequence>
  <xs:attribute ref="id"/>
</xs:complexType>
```

Wenn Ersetzungsgruppen wie `xs:choice` behandelt werden, ist diese Definition äquivalent zu der folgenden, vorausgesetzt, der Kopf ist nicht als abstrakt definiert:

```
<xs:complexType name="authorType">
  <xs:sequence>
    <xs:element ref="name"/>
    <xs:element ref="born"/>
    <xs:element ref="dead" minOccurs="0"/>
  </xs:sequence>
  <xs:attribute ref="id"/>
</xs:complexType>
```

Es sollte möglich sein, diesen komplexen Typ durch Einschränkung abzuleiten, indem man beispielsweise schreibt:

```
<xs:complexType name="restrictedAuthorType">
  <xs:complexContent>
    <xs:restriction base="authorType">
      <xs:sequence>
        <xs:choice>
          <xs:element ref="simple-name"/>
          <xs:element ref="full-name"/>
        </xs:choice>
        <xs:element ref="born"/>
        <xs:element ref="dead" minOccurs="0"/>
      </xs:sequence>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
```

Die Recommendation besagt jedoch auch, daß es während der Ableitung durch Einschränkung bei einem `xs:choice`-Kompositor »eine vollständige, die Reihenfolge bewahrende funktionale Abbildung« zwischen den Partikeln gibt, die für die Definition der abgeleiteten und der ursprünglichen `xs:choice` verwendet werden. Sie legt allerdings die Reihenfolge der Partikeln nicht fest, wenn Ersetzungsgruppen auf `xs:choice` abgebildet werden. Je nach der vom Schema-Validierer gewählten Reihenfolge bei der Konstruktion der `xs:choice` aus der Ersetzungsgruppe kann unsere Ableitung daher entweder gültig oder aber ungültig sein.

## Kontrolle über Ableitungen

Es ist an der Zeit, zu den bereits erörterten Konstruktionen zurückzukehren – und neue einzuführen –, um zu erklären, wie man die Verwendung der Ableitungen der verschiedenen Kompositoren steuert. Im wesentlichen bestehen diese Konstruktionen aus Attributen, die es ermöglichen, weitere Ableitungen entweder zu blockieren oder aber Ableitungen vorzuschreiben. Die Feinheit der Abstufung hängt jedoch vom jeweils betroffenen Kompositor ab.

### Attribute

Es gibt keine Ersetzungsgruppen für Attribute und keinen Mechanismus, um sie abzuleiten oder ihren Typ in Instanzdokumenten zu definieren. Das bedeutet auch, daß keine Konstruktionen benötigt werden, um ihre Ableitung zu steuern. Um ganz genau zu sein: Es ist möglich, Attribute indirekt abzuleiten, indem ihr Elternelement durch Einschränkung abgeleitet wird oder indem Attributgruppen redefiniert werden.

### Elemente

Wenn wir von einem `xs:element` sprechen, müssen wir zwischen globalen und lokalen Definitionen sowie Referenzen unterscheiden, die sich bezüglich Ableitungen unterscheiden. Eine Kontrolle über Ableitungen wird daher ausgeübt, wie in Tabelle 12-2 gezeigt.

Tabelle 12-2: Kontrolle über Elementableitung

| Attribut              | Globales Element | Lokale Elementdefinition | Element-Referenz |
|-----------------------|------------------|--------------------------|------------------|
| <code>block</code>    | Ja               | Ja                       | Nein             |
| <code>abstract</code> | Ja               | Nein                     | Nein             |
| <code>final</code>    | Ja               | Nein                     | Nein             |

## Block-Attribut

Das Attribut `block` steuert Typersetzung in den Instanzdokumenten durch das Attribut `xsi:type` und durch Ersetzungsgruppen. Dieses eine Attribut enthält eine Whitespace-getrennte Liste von Tokens – mit den möglichen Werten »restriction«, »extension« und »substitution« – oder den Spezialwert `#all` (was für alle drei Werte gemeinsam steht). Der Vorgabewert für das Attribut kann durch das Attribut `blockDefault` des `xs:schema`-Dokumentenelements festgelegt werden.

Die ersten beiden Werte (»restriction« und »extension«) steuern jede Ersetzung durch `xsi:type` oder Ersetzungsgruppen und blockieren die Ersetzung durch Datentypen, die durch Einschränkung oder Erweiterung aus dem Datentyp abgeleitet wurden, der in der Deklaration des Elements definiert worden ist. Der dritte Wert (»substitution«) bezieht sich nur auf die Ersetzungsgruppen und legt fest, ob ein Element aus einer Ersetzungsgruppe (für die das Element der Kopf ist) zulässig ist. Da nur globale Elemente an einer Ersetzungsgruppe teilnehmen können, hat dieser letzte Wert offensichtlich nur für globale Definitionen eine Bedeutung.

Die Tatsache, daß das Attribut `block` sowohl für Typersetzung als auch für Ersetzungsgruppen verwendet wird, kann in die Irre führen, besonders bei den Werten »restriction« und »extension«, die bei beiden Aspekten Auswirkungen zeigen. Ein einfaches Beispiel konkretisiert dies. Nehmen wir an, wir haben die Definition eines komplexen Typs (`personType`), der eine Person mit einem Namen, einem erforderlichen Geburtsdatum und optional einem Sterbedatum und einer Charakterisierung angibt:

```
<xs:complexType name="personType">
  <xs:sequence>
    <xs:element ref="name"/>
    <xs:element ref="born"/>
    <xs:element ref="dead" minOccurs="0"/>
    <xs:element ref="qualification" minOccurs="0"/>
  </xs:sequence>
  <xs:attribute ref="id"/>
</xs:complexType>
```

Wir können durch Erweiterung aus diesem komplexen Typ einen Datentyp ableiten, der einen Autor als Person mit einer Buchliste beschreibt:

```
<xs:complexType name="authorType">
  <xs:complexContent>
    <xs:extension base="personType">
      <xs:sequence>
        <xs:element name="book" type="xs:token" minOccurs="1"
          maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Wir können durch Einschränkung einen Datentyp ableiten, der eine Buchfigur als Person ohne Sterbedatum, aber mit einer erforderlichen Charakterisierung angibt:

```
<xs:complexType name="characterType">
  <xs:complexContent>
    <xs:restriction base="personType">
      <xs:sequence>
        <xs:element ref="name"/>
        <xs:element ref="born"/>
        <xs:element ref="qualification"/>
      </xs:sequence>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
```

Der erste Zweck des Attributs `block` in Elementdefinitionen liegt darin, die Typersetzung durch `xsi:type` zu steuern, wodurch Ersetzungen in den Instanzdokumenten wie etwa die folgende gesteuert werden (wenn für das Element `person` der Typ `personType` definiert worden ist):

```
<person xsi:type="authorType" id="CMS"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <name>
    Charles M. Schulz
  </name>
  <born>
    1922-11-26
  </born>
  <dead>
    2000-02-12
  </dead>
  <book>
    Auf den Hund gekommen
  </book>
</person>

<person xsi:type="characterType" id="Snoopy"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <name>
    Snoopy
  </name>
  <born>
    1950-10-04
  </born>
  <qualification>
    extrovertierter Beagle
  </qualification>
</person>
```

Die erste Ersetzung verwendet den Typ `authorType`, der durch Erweiterung von `personType` abgeleitet ist, und kann blockiert werden, indem ein `block`-Attribut angegeben wird, das den Wert `extension` enthält. Die zweite Ersetzung verwendet den Typ `characterType`,

der durch Einschränkung von `personType` abgeleitet ist, und kann blockiert werden, indem ein `block`-Attribut angegeben wird, das den Wert `restriction` enthält. Beide zusammen können blockiert werden, indem man ein `block`-Attribut mit dem Wert `#all` angibt.



Dieses Beispiel zeigt, daß sich die Auswirkungen der beiden Arten von Ableitung auf Anwendungen stark unterscheiden. Eine Typersetzung durch Einschränkung hat praktisch keine Auswirkung auf Anwendungen, da das Inhaltsmodell, das zu dem eingeschränkten Typ paßt, auch zu dem ursprünglichen Typ passen muß. Eine Typersetzung durch Erweiterung hingegen läßt Inhaltsmodelle zu, die der ursprüngliche Typ nicht erlaubt (wie die Hinzufügung des Elements `book` in dem vorangehenden Beispiel), daher ist die Gefahr größer, daß Anwendungen, die diese Hinzufügungen nicht erwarten, nicht mehr laufen. Eine konservative Möglichkeit wäre also, in Ihren Schemas `extension` in den Vorgabewert für `block` aufzunehmen.

Nachdem das gesagt ist, haben wir immer noch erst eine Seite des Attributs `block` gesehen. Es dient noch einem weiteren, unterschiedlichen, aber nicht unabhängigen Zweck und beschränkt auch die Verwendung von Ersetzungsgruppen. Um diesen Verwendungszweck zu veranschaulichen, definieren wir drei verschiedene Elementmitglieder einer Ersetzungsgruppe, deren Kopf `person` ist. Das dritte Elementmitglied ist einfach ein Synonym für das Element `person` und hat denselben Typ:

```
<xs:element name="person" type="personType"/>
<xs:element name="author" type="authorType" substitutionGroup="person"/>
<xs:element name="character" type="characterType" substitutionGroup="person"/>
<xs:element name="human" type="personType" substitutionGroup="person"/>
```

Nachdem wir diese Ersetzungsgruppe definiert haben, ohne irgend etwas zu blockieren, lassen wir nicht nur `block` und die Typersetzungen zu, die wir oben gesehen haben, sondern auch Elementersetzungen wie die folgenden:

```
<author id="CMS">
  <name>
    Charles M. Schulz
  </name>
  <born>
    1922-11-26
  </born>
  <dead>
    2000-02-12
  </dead>
  <book>
    Auf den Hund gekommen
  </book>
</author>
```

```

<character id="PP">
  <name>
    Peppermint Patty
  </name>
  <born>
    1966-08-22
  </born>
  <qualification>
    kühn, dreist und draufgängerisch
  </qualification>
</character>
<human id="CMS.">
  <name>
    Charles M. Schulz
  </name>
  <born>
    1922-11-26
  </born>
  <dead>
    2000-02-12
  </dead>
</human>

```

Die erste Ersetzung verwendet ein Element der Ersetzungsgruppe, dessen Typ durch Erweiterung vom Kopf abgeleitet ist. Sie kann blockiert werden, indem man ein `block`-Attribut angibt, das den Wert `extension` enthält. Als zweites folgt eine Ersetzung durch ein Element der Ersetzungsgruppe, dessen Typ durch Einschränkung vom Kopf abgeleitet ist. Sie kann blockiert werden, indem man ein `block`-Attribut angibt, das den Wert `restriction` enthält. Diese beiden sind den Substitutionen, die wir als Beispiele für Typsubstitutionen angegeben haben, ähnlich. Der Unterschied besteht darin, daß hier der Elementname verwendet wird, um den Typ zu unterscheiden, und nicht das Attribut `xsi:type`. Die dritte Ersetzung ist neu, da der Typ derselbe wie der des Kopfes ist, und kann nur blockiert werden, indem man ein `block`-Attribut angibt, das den Wert `substitution` enthält. Diesen Wert in ein `block`-Attribut aufzunehmen blockiert jede Elementersetzung, sagt aber gleichzeitig, daß nicht alle Kombinationen ausgedrückt werden können. Um alles zu blockieren, würde man `author` so definieren:

```
<x:element name="author" type="authorType" block="#all"/>
```

Ich will bei Ihnen nicht den Eindruck hinterlassen, Sie könnten Typersetzung nicht blockieren, ohne Ersetzungsgruppen zu blockieren. Erwähnenswert ist also, daß Typersetzung auch bei der Definition komplexer Typen blockiert werden kann. Dies wird später in diesem Kapitel im Abschnitt »Komplexe Typen« besprochen.

## Finale Elemente

Genau wie `block` hat auch `final` Auswirkungen auf Ersetzungsgruppen, diese Konstruktion arbeitet jedoch auf einer anderen Ebene: Sie beschränkt das Schema selbst, nicht jedoch die Instanzdokumente. `final` kann eine Liste aus den Werten `restriction` und

extension oder den Spezialwert #all annehmen, der Standardwert kann mit Hilfe des Attributs finalDefault von xs:schema vorgegeben werden. Ein Wert substitution ist jedoch nicht notwendig, denn im Gegensatz zu block geht es bei final um Ersetzungsgruppen, und #all kann alle Ersetzungen (und nur diese) blockieren.

Die Auswirkung von final ist radikaler als die von block: Während block die Verwendung von Ersetzungsgruppen blockiert, verbietet final die Verwendung des Elements als Kopf einer Ersetzungsgruppe. Wenn das Element person mit gesetztem Attribut final definiert ist, ist es nicht möglich, es als Kopf einer Ersetzungsgruppe für author, character oder beide gemeinsam zu verwenden.

### **Abstrakte Elemente**

Das letzte Attribut ist abstract, das das Gegenteil zu block darstellt. Es verhindert, daß das Element unmittelbar in einem Instanzdokument verwendet wird. Statt dessen muß es durch eine Ersetzungsgruppe ersetzt werden. Das Element person als abstract zu definieren verbietet seine Verwendung in den Instanzdokumenten. Sie müssen eines der Elemente aus seiner Ersetzungsgruppe (also etwa author, character oder sogar human) verwenden.

### **Komplexe Typen**

Die Attribute für komplexe Typen sind dieselben wie die für Elemente, ihre Bedeutung unterscheidet sich jedoch ein wenig, denn eines der Attribute (block) operiert mit der Ersetzung von Elementen, die den Datentyp verwenden (wie wir es im vorigen Abschnitt gesehen haben), die anderen hingegen (final und abstract) betreffen die Ableitung des komplexen Typs selbst.

### **Blockieren komplexer Typen**

Wenn das block-Attribut bei Definitionen komplexer Typen angewendet wird, sind Typersetzen bei den mit diesem Typ definierten Elementen immer noch blockiert. Der Unterschied besteht darin, daß sich block in diesem Fall nicht auf Elementersetzen durch Ersetzungsgruppen, sondern auf Typersetzen bezieht. Diese block-Attribute kann man sich wie in Serie liegende elektrische Schalter vorstellen, von denen jeder die Ableitung auf seiner eigenen »Leitung« blockieren kann, wie in Abbildung 12-1 gezeigt.

Jeweils für restriction und extension können Ersetzungen zunächst für Element- und Typersetzen zugleich durch das block-Attribut der Elementdefinition abgeschaltet werden, und zusätzlich können Typersetzen durch das block-Attribut in der Definition des komplexen Typs blockiert werden. Der Wert substitution im block-Attribut der Elementdefinition wirkt wie ein globaler Schalter für alle Elementersetzen, einschließlich einer dritten »Leitung«, die Elementersetzen mit dem gleichen Typ zuläßt und die nicht getrennt blockiert werden kann.

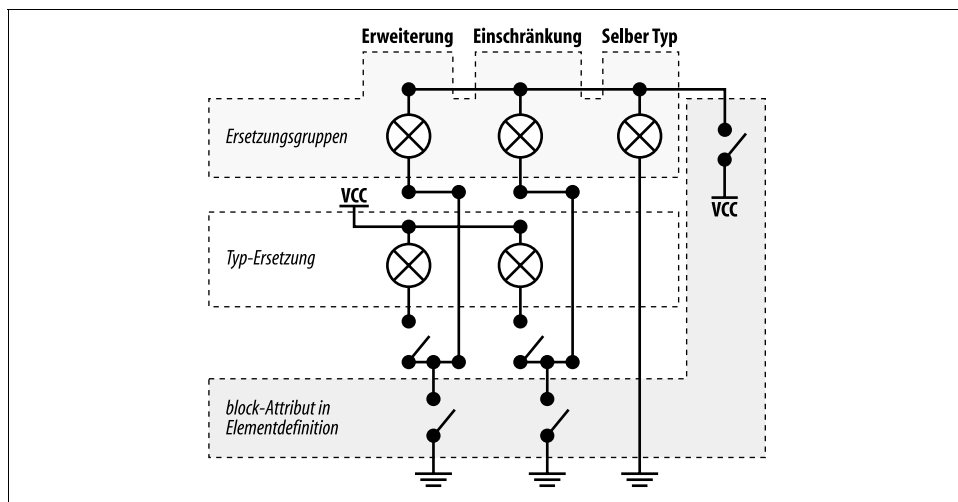


Abbildung 12-1: block-Attribut für die Definition von Elementen und komplexen Typen

Der im Schema-Element angegebene Standardwert wird auf beiden Ebenen benutzt und kann für Definitionen von Elementen bzw. von komplexen Typen unterschiedlich sein, wenn sie zu unterschiedlichen Schemata gehören und einen unterschiedlichen `xs:schema`-Ahnen haben. Daß der Standardwert, wenn er im `xs:schema`-Ahnen definiert ist, auf beide Ebenen angewendet wird, bedeutet, daß er möglicherweise die Ableitung für diese beiden Ebenen blockiert (indem er zwei Schalter »öffnet«). Um einen solchen nicht-leeren Standardwert außer Kraft zu setzen, müssen Sie bei der Definition sowohl von Elementen als auch von komplexen Typen ein `block`-Attribut angeben.

### Finale komplexe Typen

Das Attribut `final` steuert bei der Definition komplexer Typen, ob der Typ durch Einschränkung oder Erweiterung abgeleitet werden kann, um neue komplexe Typen zu erzeugen.

Im Gegensatz zum `block`-Attribut, das mit seinem Gegenstück bei der Elementdefinition verbunden ist, gilt das Attribut `final` bei der Definition eines komplexen Typs nur für den komplexen Typ selbst. Ähnlich wie das Attribut `abstract` bei der Elementdefinition wird es auf der Schema-Ebene selbst angewendet und hat keine Auswirkungen auf die Instanzdokumente.

### Abstrakte komplexe Typen

Das Attribut `abstract` operiert bei der Definition komplexer Typen auf den Instanzdokumenten. Wenn ein abstrakter Datentyp verwendet wird, um Elemente zu definieren, muß der Typ in den Instanzdokumenten durch ein `xsi:type`-Attribut ersetzt werden.

Wir müssen deutlich darauf hinweisen, daß `abstract` für sich genommen nicht heißt, daß der komplexe Typ nicht in einem Inhaltsmodell verwendet werden kann. Wenn er jedoch verwendet wird, muß er in den Instanzdokumenten durch `xsi:type` ersetzt werden. Um einen komplexen Typ zu definieren, der nicht in Inhaltsmodellen verwendbar ist, müssen wir sowohl das `final`- als auch das `block`-Attribut angeben; dann kann dieser komplexe Typ ausschließlich als Basistyp für Ableitungen verwendet werden.

## Einfache Typen

Einfache Typen sind tatsächlich einfacher, was die Kontrolle über ihre Ableitungen angeht, denn sie können höchstens `final` sein. Ihr `final`-Attribut kann die Werte `list`, `restriction`, `union` oder den Spezialwert `#all` annehmen, was für alle drei Werte zugleich steht. Der Standardwert wird durch das Attribut `finalDefault` des Schema-Elements vorgegeben, der auch für Elemente und komplexe Typen gilt.

Daß einfache Typen nicht `abstract` oder `blockiert` sein können, vermeidet mögliche Probleme, wenn sie für die Definition von Attributen verwendet werden, für die diese Begriffe bedeutungslos sind. Falls man sie doch für die Definition von Elementen benötigt, kann man einen komplexen Typ einfachen Inhalts erzeugen, der den einfachen Typ als Basis verwendet. Dieser komplexe Typ kann die Attribute `abstract` und `block` aufnehmen.

Auch wenn `final` das einzige Attribut ist, das Ableitungen von Elementen einfachen Typs steuert und für das Element `xs:simpleType` (globale Definition) verfügbar ist, kann eine feinere Steuerung durch das Attribut `fixed` erzielt werden, das für jede der Facetten zur Verfügung steht (wie in Kapitel 5 erläutert).

## Andere Komponenten und Redefinitionen

Andere Komponenten, beispielsweise Attribute sowie Element- und Attributgruppen, können nicht unmittelbar abgeleitet werden. Attribute können überhaupt nicht abgeleitet werden, Gruppen nur durch Redefinition. Daher gibt es für diese Fälle keine Kontrolle über die Ableitung. Dementsprechend überschreitet auch die Redefinition von Gruppen durch `xs:redefine` die Möglichkeiten dieser Konstruktion; sie kann ebenfalls überhaupt nicht gesteuert werden. Auch wenn die Empfehlung in dieser Hinsicht unscharf ist, ist es sicherer anzunehmen, daß das `final`-Attribut der komplexen Typen auch für ihre Redefinition gilt, die wie implizite Ableitungen behandelt werden.



Die Interpretation des `block`-Attributs ist nicht einheitlich. Die aktuelle Ansicht der *W3C XML Schema Working Group* finden Sie in der Errata-Liste für die Spezifikation unter <http://www.w3.org/2001/05/xmlschema-errata>.