

***icoya***  
***OpenContent***

**Aktive Websites  
schnell erstellen  
und einfach pflegen**

## **Zope Handbuch**

- ↘ **Structured Text**
- ↘ **Zope Page Templates**
- ↘ **METAL Ausdrücke**

---

## Deutsche Übersetzung von:

### **struktur AG**

Junghansstraße 5  
D-70469 Stuttgart  
Germany

E-Mail: [info@struktur.de](mailto:info@struktur.de)

[www.struktur.de](http://www.struktur.de)  
[www.strukturag.com](http://www.strukturag.com)

[www.icoya.de](http://www.icoya.de)  
[www.icoya.org](http://www.icoya.org)  
[www.icoya.net](http://www.icoya.net)  
[www.icoya.com](http://www.icoya.com)

## **COPYRIGHT**

The copyright to each Open Publication is owned by its author(s) or designee.

The Open Publication works may be reproduced and distributed in whole or in part, in any medium physical or electronic, provided that the terms of this license are adhered to, and that this license or an incorporation of it by reference (with any options elected by the author(s) and/or publisher) is displayed in the reproduction.

Proper form for an incorporation by reference is as follows:

This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v1.0 or later (the latest version is presently available at <http://www.opencontent.org/openpub/>).

The reference must be immediately followed with any options elected by the author(s) and/or publisher of the document (see section VI).

Commercial redistribution of Open Publication-licensed material is permitted.

Any publication in standard (paper) book form shall require the citation of the original publisher and author. The publisher and author's names shall appear on all outer surfaces of the book. On all outer surfaces of the book the original publisher's name shall be as large as the title of the work and cited as possessive with respect to the title.

Inhaltsverzeichnis

<b>1</b>	<b><i>structured text</i>-Anleitung</b>	<b>4</b>
1.1	<b><i>structured text</i>-Anleitung</b>	<b>4</b>
1.1.1	Grundlagen von <i>structured text</i>	4
1.1.2	Einrückung verwenden	4
1.1.3	Listen und Elemente	5
1.1.4	Beispielcode	5
1.1.5	Hyperlinks	6
1.1.6	Fortgeschrittene Verwendung	6
1.1.7	Zusammenfassung	6
1.1.8	Quellen	6
1.2	<b><i>structured text</i></b>	<b>7</b>
1.2.1	Einleitung	7
1.2.2	<i>structured text</i> Manipulation	7
1.2.3	<i>structured text</i> in DTML verwenden	8
1.2.4	Andere Quellen	8
<b>2</b>	<b>Page Templates</b>	<b>9</b>
2.1	<b>Zope Page Templates: Erste Schritte</b>	<b>9</b>
2.1.1	Einführung in Page Templates	9
2.1.2	Schon wieder eine neue Template Sprache?	9
2.1.3	Die Anwendung der Prinzipien	9
2.1.4	Page Templates erstellen	10
2.1.5	“Simple Expressions”	10
2.1.6	Text einfügen	10
2.1.7	Wiederholte Strukturen	10
2.1.8	Bedingte Elemente	11
2.1.9	Variablen definieren	11
2.1.10	Attribute ändern	12
2.1.11	Zusammenfassung	12
2.2	<b>Zope Page Templates: Fortgeschrittene Konzepte</b>	<b>12</b>
2.2.1	“Mixing”- and “Matching”-Anweisungen	12
2.2.2	Mehrteilige Anweisungen	12
2.2.3	String Ausdrücke	13
2.2.4	“Nocall”-Pfad-Ausdrücke	13
2.2.5	Python Ausdrücke	13
2.2.6	Standard-Variablen	13
2.2.7	Wechselnde Pfade	13
2.2.8	Platzhalter-Elemente	14
2.2.9	Einfügestrukturen	14
2.2.10	Python Grundlagen	14
2.2.11	Zope-Objekte referenzieren	14
2.2.12	Die Verwendung von Skripten	15
2.2.13	DTML aufrufen	15
2.2.14	Python Module	15
2.3	<b>TALES Spezifikation Version 1.3</b>	<b>16</b>
2.3.1	TAL Spezifikation Version 1.4	17
2.4	<b>Häufig gestellte Fragen zu Zope Page Templates</b>	<b>20</b>
<b>3</b>	<b>METAL Ausdrücke</b>	<b>22</b>
3.1	<b>METAL für Anfänger</b>	<b>22</b>
3.2	<b>Eine Basis-Anleitung für METAL</b>	<b>22</b>
3.2.1	Was ist METAL?	22
3.2.2	Mit METAL beginnen	22
3.2.3	Wofür Makros verwendet werden können	23
3.2.4	Links und andere Quellen	23
3.2.5	METAL Spezifikation Version 1.0	23
	<b>Open Publication License</b>	<b>25</b>

## 1.1 structured text-Anleitung

Von Paul Everitt

Ingenieure verbringen viel Zeit mit Kommunikation, vorzugsweise per Email, aber auch über Dokumentationen. Jedoch sind Dokumentationen von Ingenieuren aus einem einfachen Grund sehr kompliziert: Die Welt konsumiert Dokumentationen hauptsächlich in Präsentationsformaten wie HTML oder PDF. Theoretisch wäre das kein großes Problem, wenn wir alle fröhlich losmarschieren und in "DocBook" (oder vielleicht LaTeX) schreiben, der angeblichen "lingua franca" der Dokumentation. Allerdings unterstützen die meisten Werkzeuge "DocBook" (oder LaTeX) nicht besonders gut, und selbst wenn die Werkzeuge komplett durchentwickelt wären, würden die meisten sie ablehnen. Warum? Ingenieure verbringen die meiste Zeit damit, mit einfachem Text zu kommunizieren. Die Werkzeuge, die sie verwenden (vi und emacs) sind textorientiert. Der größte Wortanteil ihrer Kommunikation läuft über Email. Und schließlich ist alles an Dokumentation, was man aus einem Ingenieur herausquetschen kann, eine Anzahl von Kommentarzeilen in Quellcode.

Wäre es nicht nett, wenn es ein rein textorientiertes System ohne Tags dafür gäbe, die semantische Bedeutung auszudrücken? Dieses Problem wird mit *structured text* angegangen. Mit *structured text* wird formatunabhängige Texterstellung extrem natürlich und bequem, sobald einige wenige Regeln gelernt sind. Außerdem kann *structured text* so erweitert werden, daß er auch fortgeschrittenen und proprietären Anforderungen genügt. (Es soll jedoch nicht verschwiegen werden, daß es auch eine Schwäche gibt: die Verwendung von Umlauten. Umlaute können bisher nicht verwendet werden. In späteren Versionen wird dieses Problem sicher behoben sein.)

Um einen ersten Eindruck davon zu bekommen, was *structured text* ist: Der folgende Text:

Manchmal ist die *beste* Herangehensweise an die Komplexität die Einfachheit. Ein gutes *structured text*-System ist:

- o komfortabel
- o ergiebig

wird in HTML dargestellt als:

```
<p>
Manchmal ist die <em>beste</em> Herangehensweise an
die Komplexität die Einfachheit. Ein gutes structured text-
System ist:
</p>
```

```
<ul>
<li><p>komfortabel</p></li>
<li><p>ergiebig</p></li>
</ul>
```

und wie folgendes [DocBook](http://www.nwalsh.com/docbook/) (<http://www.nwalsh.com/docbook/>) XML:

```
<para>
Manchmal ist die <emphasis>beste</emphasis>
Herangehensweise an die Komplexität die Einfachheit. Ein
gutes structured text-System ist:
</para>

<itemizedlist>
<listitem><para>komfortabel</para></listitem>
<listitem><para>ergiebig</para></listitem>
</itemizedlist>
```

Tatsächlich wurde der Text dieses Artikels in *structured text* geschrieben. In diesem Artikel werden wir uns mit den Grundlagen von *structured text* beschäftigen und sehen, wie man große Texte in kleinere Abschnitte organisiert, fortgeschrittene Formatierungen macht und Metadaten verwendet.

### 1.1.1 Grundlagen von *structured text*

Lassen Sie uns anfangen und durch den Vergleich zwischen HTML und *structured text* einen Einblick in die Grundlagen erhalten. Das einfachste Konstrukt in *structured text* ist ein Paragraph. Der folgende Codeschnipsel:

Dies ist der erste Paragraph.

Dies ist der zweite Paragraph.

...wird in folgendes HTML transformiert:

```
<p>Dies ist der erste Paragraph.</p>
```

```
<p>Dies ist der zweite Paragraph.</p>
```

Also haben nichtdruckbare Zeichen (white spaces) in *structured text* eine Bedeutung. Dies ist eine sehr unmittelbare Idee. Zum Beispiel werden in Emails Paragraphen durch "white spaces" getrennt.

Um Hervorhebung einzuführen, verwendet *structured text* eine andere Konvention: Sternchen. Man beachte den folgenden Ausschnitt:

Dies ist der *\*erste\** Paragraph.

Dies ist der *\*\*zweite\*\** Paragraph.

Im daraus generierten HTML finden das `em`-Tag sowie das `strong`-Tag Verwendung:

```
<p> Dies ist der <em>erste</em> Paragraph.</p>
```

```
<p> Dies ist der <strong>zweite</strong> Paragraph.</p>
```

Dies ist ebenfalls ein gängiges Muster bei Emails. Es werden noch weitere bekannte Muster unterstützt, z. B. der Bezug auf einen Jargon:

Wenn Sie 'STX' sehen, wissen Sie, dies ist ein Abkürzung für 'structured text'.

Die HTML-Ausgabe sieht folgendermaßen aus:

```
<p>Wenn Sie <code>STX</code> sehen, dies ist ein
Abkürzung für <code>structured text</code>.</p>
```

### 1.1.2 Einrückung verwenden

Der vorige Abschnitt konzentrierte sich auf Konventionen die eine semantische Bedeutung transportieren. Diese semantische Bedeutung läßt den *structured text*-Prozessor bestimmte HTML-Tags erzeugen.

Bei *structured text* spielt die Einrückung ebenfalls eine große Rolle bei der Anzeige von semantischer Bedeutung. Die einfachste Idee ist die der Überschriften. Im folgenden Beispiel wird Einrückung verwendet, um eine Überschrift zu erzeugen.

Einrückung verwenden

Der vorhergehende Absatz konzentrierte sich auf Textkonvertierungen die semantische Informationen beinhalten. Diese semantische Bedeutung erzeugt bei der Verarbeitung mit 'structured text' bestimmte HTML Tags.

Das folgende HTML wird erzeugt:

```
<h1>Einrückung verwenden </h1>
```

```
<p> Der vorhergehende Absatz konzentrierte sich auf
Textkonvertierungen die semantische Informationen beinhalten.
Diese semantische Bedeutung erzeugt bei der Verarbeitung mit
'structured text' bestimmte HTML Tags.</p>
```

Nochmals, die Einrückung trägt also semantische Bedeutung. Der Paragraph ist der Überschrift direkt untergeordnet; diese Beziehung wird auch im HTML ausgedrückt. Die Einrückung kann weiter fortgesetzt werden:

```
Einrückung verwenden
```

```
Der vorhergehende Absatz konzentrierte sich auf Textkonvertierungen
die semantische Informationen beinhalten. Diese semantische Bedeutung
erzeugt bei der Verarbeitung mit 'structured text' bestimmte HTML Tags.
```

```
Grundlagen der Einrückung
```

```
In diesem Abschnitt werden wir die Grundlagen der Einrückung
untersuchen ...
```

```
Hyperlinks
```

Das folgende HTML wird erzeugt:

```
<h1>Einrückung verwenden</h1>
```

```
<p> Der vorhergehende Absatz konzentrierte sich auf
Textkonvertierungen die semantische Informationen beinhalten.
Diese semantische Bedeutung erzeugt bei der Verarbeitung mit
'structured text' bestimmte HTML Tags.</p>
```

```
<h2>Grundlagen der Einrückung</h2>
```

```
<p>In diesem Abschnitt werden wir die Grundlagen der
Einrückung untersuchen ...</p>
```

```
<h2>Hyperlinks</h2>
```

### 1.1.3 Listen und Elemente

Listen werden in *structured text* auch unterstützt, inklusive geordneten, ungeordneten und descriptiven Listen. Die ungeordnete Liste ist ein gebräuchliches Muster bei der Textkommunikation:

HTML beinhaltet drei Arten von Listen:

- o Ungeordnete Listen
- o Geordnete Listen
- o Beschreibende Listen

*structured text* erlaubt die Verwendung der Symbole \*, o und - um Listenpunkte zu erstellen. Das obige Beispiel erzeugt folgendes HTML:

```
<p>HTML beinhaltet drei Arten von Listen:</p>
```

```
<ul>
```

```
<li><p>Ungeordnete Listen</p></li>
```

```
<li><p>Geordnete Listen</p></li>
```

```
<li><p>Beschreibende Listen</p></li>
```

```
</ul>
```

Die Richtlinien von *structured text* für geordnete Listen sind wie folgt:

HTML beinhaltet drei Arten von Listen:

1. Ungeordnete Listen
2. Geordnete Listen
3. Beschreibende Listen

Dies produziert untenstehendes HTML:

```
<p>HTML beinhaltet drei Arten von Listen:</p>
```

```
<ol>
```

```
<li><p>Ungeordnete Listen</p></li>
```

```
<li><p>Geordnete Listen</p></li>
```

```
<li><p>Beschreibende Listen</p></li>
```

```
</ol>
```

Beschreibende Listen (descriptive lists) können durch die Verwendung eines doppelten Minus-Zeichens ebenfalls sehr einfach erzeugt werden:

Ungeordnete Listen -- Beinhaltet im allgemeinen eine Folge von Punkte im HTML.

Geordnete Listen -- HTML Browser erzeugen die Listenpunkte in eine numerierte Folge.

Beschreibende Listen -- Im allgemeinen für Definitionsklisten wie Glossare verwendet.

Das folgende HTML wird erzeugt:

```
<dl>
```

```
<dt>Ungeordnete Listen </dt>
```

```
<dd><p>Beinhaltet im allgemeinen eine Folge von Punkte
im HTML.</p></dd>
```

```
<dt> Geordnete Listen </dt>
```

```
<dd><p>HTML Browser erzeugen die Listenpunkte in
eine numerierte Folge.</p></dd>
```

```
<dt> Beschreibende Listen </dt>
```

```
<dd><p>Im allgemeinen für Definitionsklisten wie
Glossare verwendet.</p></dd>
```

```
</dl>
```

### 1.1.4 Beispielcode

Wie bereits oben erwähnt, gibt es für *structured text*-Autoren einfache Konventionen, um etwa die monotypische Semantik des Tags code zu erreichen:

Wenn der Dialog erscheint, klicken Sie auf den 'OK'-Button.

wird wie folgt in HTML dargestellt:

```
<p>Wenn der Dialog erscheint, klicken Sie auf den
<code>OK</code>-Button.</p>
```

Wie auch immer, manchmal hat man sehr lange Code-Passagen. Zum Beispiel könnte man eine Python-Funktion mitten in einem Artikel beschreiben wollen. Ein Codeblock kann auch dadurch angezeigt werden, daß der vorhergehende Paragraph mit '::' endet, und der bzw. die folgende(n) Paragraph(en) eingerückt werden.

Wir betrachten ein Beispiel:

```
In unserem folgenden Python Beispiel , rechnen wir
Menschenjahre in Hundejahre um::
def hundeJahre(alter):
    """Umrechnung von Menschenjahre in Hundejahre"""
    return alter*7
```

Und in HTML:

```
<p>In unserem folgenden Python Beispiel , rechnen wir
Menschenjahre in Hundejahre um:</p>

<pre>
def hundeJahre(alter):
    """Umrechnung von Menschenjahre in Hundejahre"""
    return alter*7
</pre>
```

Die Verwendung von ':' als Endzeichen des letzten Satzes eines Paragraphen bedeutet mehr als die Anwendung der CODE-Tag-Semantik. Der eingerückte Block wird escaped, das bedeutet, daß Zeichen, die im Kontext von *structured text* oder HTML eine spezielle Bedeutung haben, diese Bedeutung verlieren und nur als ihre druckbare Repräsentation verstanden werden. Beispielsweise wird '<' als Kleinerzeichen ausgegeben und nicht als Tag-Anfang interpretiert. Genau so konnten auch die ganzen Code-Ausschnitte in diesem Artikel formatiert dargestellt werden. Noch ein Beispiel. Die Zeichen 'größer als', 'kleiner als' und 'Kaufmanns-Und' sind *escaped*:

Hier ist ein HTML-Beispiel::

```
<html>
<p>Dies ist eine Seite über Hunde und Katzen.</p>
</html>
```

... und folgendes HTML wird erzeugt:

```
<p>Hier ist ein HTML-Beispiel:</p>

<pre>
<html>
<p>Dies ist eine Seite über Hunde und Katzen.</p>
</html>
</pre>
```

### 1.1.5 Hyperlinks

In den vorherigen Abschnitten haben wir uns darauf konzentriert, wie man eine bestimmte Präsentations-Semantik durch die Verwendung einfacher Richtlinien erzielt. Aber das Web besteht nicht nur aus HTML. Die Verlinkung von Wörtern und Phrasen mit anderen Informationen und das Einfügen von Bildern ist ebenso wichtig. Glücklicherweise unterstützt *structured text* Hyperlinks und das Einfügen von Bildern. Wir beginnen mit einem einfachen Hyperlink. Haben wir etwa einen *structured text* Paragraphen über Python, wie:

Um mehr von Python zu erfahren, besuchen Sie bitte die "Python Website" :<http://www.python.org/>.

Daraus wird:

```
<p>Um mehr von Python zu erfahren, besuchen Sie bitte
die <a href="http://www.python.org/">Python
Website</a>.
```

Die Konventionen sind absolut einfach:

- ☞ Der Referenz-Text ist in Anführungszeichen gesetzt.
- ☞ Das zweite Anführungszeichen wird gefolgt von einem Doppelpunkt.
- ☞ Die URL darf unmittelbar von einem Satzzeichen gefolgt werden.

Die Grundregeln haben eine Anzahl von Variationen. So sind etwa auch relative URLs erlaubt, genauso wie 'mailto' URLs. (Bemerkung: im oberen Beispiel sollte eigentlich kein Leerzeichen zwischen dem Anführungszeichen und dem Doppelpunkt stehen. Dies war hier aber notwendig aufgrund eines Fehlers in der momentan bei Zope verwendeten Version. In neueren Versionen ist dieser Fehler bereits beseitigt.)

### 1.1.6 Fortgeschrittene Verwendung

Es gibt noch viele undurchsichtige Erweiterungen für *structured text* für die Behandlung von Querverweisen, Tabellen, Bildern und vielem anderen mehr.

Ein großer Vorteil von *structured text* ist die einfache Erweiterbarkeit, etwa wenn man mit den bisherigen Regeln nicht zufrieden ist. Dies wurde möglich durch eine kürzlich erschienene Neufassung von *structured text*, bekannt auch unter dem Namen "*structured text NG*". Zum Beispiel kann ein Modul für die Ausgabe von LaTeX hinzugefügt werden, oder die Regeln für die Erkennung von Hyperlinks kann umgeschrieben werden.

*structured text* ist in Zope verfügbar und es ist ebenfalls im Zope [Content Management Framework](http://cmf.zope.org) (<http://cmf.zope.org>) integriert. Aber es kann auch außerhalb von Zope verwendet werden. Um *structured text* in Zope zu verwenden, muß einfach ein Dokument bzw. eine Datei in *structured text* erstellt werden, und dann auf die folgende Art angesprochen werden:

```
<dtml-var my_document fmt=structured-text>
```

Diese Anweisung gibt die HTML-Repräsentation des Dokuments `my_document` aus. Das Buch "The [Zope Book](http://www.zope.org/Members/michel/ZB)" (<http://www.zope.org/Members/michel/ZB>) ist ein Beispiel für ein Projekt, das *structured text* außerhalb von Zope verwendet. Das gesamte Buch ist mit *structured text* gesetzt, mit einigen Erweiterungen für die Behandlung von Abbildungen und die Unterstützung des hauseigenen Markup-Formats des Verlegers. Python-Skripte parsen die Eingabe und erzeugen Ausgabedaten in den Formaten HTML und PDF.

Auch in Python-Kommentaren (docstring) kann *structured text* verwendet werden. Ein Anzahl von Dokumentationswerkzeugen für Python unterstützt *structured text*. Gegenwärtig wird weiter an docstring Konventionen und einem docstringverarbeitenden System gearbeitet. (<http://www.python.org/sigs/doc-sig>)

### 1.1.7 Zusammenfassung

Mit *structured text* hat man ein einfaches Werkzeug, mittels dem man sich durch einfachen Text ausdrücken kann. Die Implementierung von *structured text* erlaubt die Anpassung der Syntax und der Ausgabe. Außerdem ist *structured text* sowohl in Zope integriert, als auch außerhalb verwendbar.

### 1.1.8 Quellen

[Structured Text Wiki](http://www.zope.org/Members/jim/StructuredTextWiki/FrontPage) (<http://www.zope.org/Members/jim/StructuredTextWiki/FrontPage>) - behandelt *structured text* und STXNG.

[reStructuredText](http://structuredtext.sourceforge.net/) (<http://structuredtext.sourceforge.net/>) - Eine *structured text* Alternative, entwickelt als Python docstring Standard.

## 1.2 structured Text

Von [millejoh](http://www.zope.org/Members/millejoh) (<http://www.zope.org/Members/millejoh>).  
 Letzte Änderung 16.10.2000  
 Siehe auch <http://www.zope.org/Members/millejoh/structuredText>

### 1.2.1 Einleitung

*structured Text* ist ein verdammt gutes Werkzeug, und sollte von allen gekannt werden. Es eignet sich hervorragend für schnelles Prototyping bei der Erstellung von Webseiten in Zope. Irgendwie kann ich mich selbst nicht immer an alle Funktionen von *structured text* erinnern, ergo der 'raison d'être' für dieses HOWTO.

Es ist fast nur eine Sammlung von Dokumentation, die ich in Quellcode gefunden und zusammenkopiert habe. Als ich dieses Dokument das erste Mal zusammengestellt habe war mir nicht bewußt, daß ich keine Ahnung davon hatte, wie ich *structured text* in meine DTML Methoden und Dokumente einfügen kann. Für diejenigen, die sich ähnlichen Problemen gegenübersehen, habe ich am Schluß ein kleines Beispiel dafür eingefügt, wie man *structured text* in DTML Dokumente einbindet. Keine Angst - wenn man die magischen Zauberformeln einmal kennt, ist alles ganz einfach. Viel Spaß!

Vorschläge jeder Art sind jederzeit [willkommen](mailto:jmiller1@uop.com). (mailto:jmiller1@uop.com).

### 1.2.2 structured text Manipulation

Analysiere einen *structured text*-String so, daß man ihn mit strukturierten Formaten wie HTML verwendet kann. *structured text* ist Text, der Einrückung und eine einfache Symbolik verwendet, um die Struktur eines Dokuments zu bestimmen.

Ein strukturierter String besteht aus einer Folge von Paragraphen, die durch eine oder mehrere Leerzeilen voneinander getrennt sind. Jeder Paragraph hat einen Level, der definiert ist als die minimale Einrückung des Paragraphen. Ein Paragraph ist Unterparagraph eines anderen Paragraphen, wenn der andere Paragraph direkt vorausgeht und einen niedrigeren Level hat.

Eine spezielle Symbolik wird für die Anzeige spezieller Konstrukte verwendet:

- ☞ Ein einzeliger Paragraph dessen direkt folgende Paragraphen einen niedrigeren Level haben, wird als Überschrift behandelt.
- ☞ Ein Paragraph, der mit '-', '\*', oder 'o' anfängt, wird als Element einer ungeordneten Liste behandelt.
- ☞ Ein Paragraph, der mit einer Folge von Zahlen beginnt, gefolgt von einem Leerzeichen, wird als Element einer geordneten Liste betrachtet.
- ☞ Ein Paragraph, der mit einer Folge von Folgen beginnt, wobei jede Folge eine Folge von Zahlen oder Buchstaben ist, gefolgt von einem Punkt, wird als Element einer geordneten Liste betrachtet.
- ☞ Ein Paragraph, dessen erste Zeile aus etwas Text besteht, gefolgt von beliebigen Leerzeichen und '--' wird als beschreibendes Listenelement behandelt. Der führende Text ist der Titel des Elements.
- ☞ Unterparagraphen von Paragraphen die mit dem Wort 'example' oder dem Wort 'examples' oder '::' beginnen, werden als Beispielcode betrachtet und behalten ihre Formatierung bei.

☞ Text in einfachen Anführungszeichen, mit einem Leerzeichen vor dem linken und einem Leerzeichen oder einem Satzzeichen vor dem rechten, wird als Beispielcode betrachtet.

☞ Text der in '\*' eingeschlossen ist, mit einem Leerzeichen vor dem linken und einem Leerzeichen oder einem Satzzeichen vor dem rechten, wird hervorgehoben.

☞ Text der in '\*\*' eingeschlossen ist, mit einem Leerzeichen vor dem linken und einem Leerzeichen oder einem Satzzeichen vor dem rechten, wird stark hervorgehoben.

☞ Text der in '\_' eingeschlossen ist, mit einem Leerzeichen vor dem linken und einem Leerzeichen oder einem Satzzeichen vor dem rechten, wird unterstrichen.

☞ Text der in doppelte Anführungszeichen eingeschlossen ist, gefolgt von einem Doppelpunkt ':', wiederum gefolgt von einer URL und mit einem Satzzeichen abgeschlossen, wird als Hyperlink behandelt.

Zum Beispiel:

"Zope":<http://www.zope.org/> is ...

wird als

`<a href="http://www.zope.org/">Zope</a> is ....`

interpretiert. Dies gilt sowohl für relative als auch für absolute URLs.

☞ Text der in doppelte Anführungszeichen eingeschlossen ist, gefolgt von einem Komma ',', einem oder mehreren Leerzeichen gefolgt von einer absoluten URL und mit einem Satzzeichen und Leerzeichen abgeschlossen, wird als Hyperlink behandelt. Zum Beispiel

"Schreib mir", <mailto:amos@digicool.com>.

wird interpretiert als

`"<a href="mailto:amos@digicool.com">Schreib mir</a>."`

Dies gilt sowohl für relative als auch für absolute URLs.

☞ Text, der in eckige Klammern eingeschlossen ist die nur Buchstaben, Zahlen, '\_' und '-' enthalten, wird als Hyperlink innerhalb des Dokuments interpretiert. Ein Beispiel:

Wie A. Smith [a12] gezeigt hat, ist diese Technik sehr effizient. [r1]

(<http://www.zope.org/Members/millejoh/structuredText/#r1>)

wird interpretiert als

`"Wie A.Smith <a href="#a12">[12]</a> gezeigt hat, ..."`

Zusammen mit der nächsten Regel können so einfach Verweise und Endnoten eingefügt werden.

☞ Text in eckiger Klammer am Zeilenanfang, dem zwei Punkte und ein Leerzeichen vorangehen, wird als benannter Link behandelt. Zum Beispiel:

.. [a12] "Effektive Techniken" Smith, A.

wird interpretiert als `<a name="a12">[12]</a> "Effektive Techniken" Smith, A. .... [r2]` (<http://www.zope.org/Members/millejoh/structuredText/#r2>). Zusammen mit der vorigen Regel lassen sich so einfach Verweise und Links einfügen.

[r1] Gemäß der HTML 4.0 [Spezifikation](http://www.w3.org/TR/REC-html40/types.html#type-id) (<http://www.w3.org/TR/REC-html40/types.html#type-id>) müssen Bezeichner mit einem Buchstaben beginnen, also ist die Verwendung einer Referenz namens [12] eigentlich tabu, obwohl es funktioniert (wenigstens in allen Browsern, die getestet worden sind). Das StructuredText Python Modul sollte wahrscheinlich aktuali-

siert werden, um ein besseres Verhalten zu implementieren, etwa dem voranstellen von 'ref' bei Linkadressen, die nur aus Zahlen bestehen.

[r2] Der Verwendung des Attributes 'name' ist veraltet. Stattdessen sollte 'id' verwendet werden. Nicht daß man dagegen etwas unternehmen könnte, da *structured text* das HTML generiert (eigentlich sollte es geändert werden). Aber man sollte wenigstens Bescheid wissen, falls einer kleinlich ist.

### 1.2.3 *structured text* in DTML verwenden

Die härteste Zeit hatte ich, als ich versucht habe, *structured text* innerhalb von HTML zu verwenden und anzeigen zu lassen, bis ich folgenden Zauberspruch gelernt habe:

```
<dtml-var stx_doc_name fmt=structured-text>
```

Ohne den Teil 'fmt=structured-text' wird einfach der bloße Text ausgegeben, aber nicht das, was man sehen will.

[http://www.zope.org/Members/millejoh/structuredText/raw\\_text](http://www.zope.org/Members/millejoh/structuredText/raw_text)

### 1.2.4 Andere Quellen

Das [structured text Document](http://www.zope.org/Members/tseaver/STX_Document) ([http://www.zope.org/Members/tseaver/STX\\_Document](http://www.zope.org/Members/tseaver/STX_Document)) von Tres Seaver ist es auf jeden Fall Wert, angeschaut zu werden. Nützlicherweise wird auch eine ZClass bereitgestellt, die man als Wrapper für STX verwenden kann. Einfach besorgen, benutzen und glücklich damit sein.

Es gibt zwei kleinere Vorbehalte, die aber in 99% aller denkbaren Anwendungsfälle nicht zum Tragen kommen. In den Release Notes ist näheres darüber zu finden.

## 2.1 Zope Page Templates: Erste Schritte

### 2.1.1 Einführung in Page Templates

von Evan Simpson

Page Templates sind ein Werkzeug für die Erstellung von Webseiten. Sie unterstützen die Zusammenarbeit von Programmierern und Designern bei der Erzeugung dynamischer Webseiten für ZOPE Web-Applikationen. Designer können sie für die Pflege von Seiten nutzen, ohne auf Ihre gewohnten Werkzeuge und Editoren zu verzichten, und werden dabei vor dem Aufwand bewahrt, die Seiten in eine Anwendung einzubetten. In diesem Artikel betrachten wir die Grundlagen von Page Templates und wie Sie sie bei Ihrer Webseite für die einfache Erstellung von dynamischen Inhalten verwenden können. Die Zielsetzung von Page Templates ist ein natürlicher Arbeitsfluß. Ein Designer wird einen WYSIWYG HTML-Editor verwenden, um ein Template zu erstellen. Dann wird ein Programmierer es anpassen, um es in eine Anwendung zu integrieren. Bei Bedarf kann der Designer das Template wieder in seinen Editor laden und weitere Änderungen an der Struktur und dem Erscheinungsbild vornehmen. Durch mäßige Veränderungen, um die Arbeit des Programmierers zu erhalten, hält er die Anwendung am laufen.

Page Templates zielen in diese Richtung durch die Übernahme von drei Prinzipien:

1. Nettes Herumspielen mit Editierwerkzeugen
2. What you see is very similar to what you get. (In Anlehnung an das englische WYSIWYG (what you see is what you get). Es sagt aus, daß die Ausgabe von automatisch erzeugtem Code, der mit visuellen Werkzeugen erstellt worden ist, dem visuellen Design entspricht.)
3. Halte Programmcode aus den Templates fern, außer der strukturellen Logik

Ein Page Template ist wie ein Modell für die Seiten, die damit generiert werden. Insbesondere ist es eine gültige HTML Seite. Da HTML stark strukturiert ist, und WYSIWYG Editoren sorgfältig diese Struktur erhalten, gibt es strenge Richtlinien in welcher Art und Weise ein Programmierer eine Seite verändern kann und immer noch das erste Prinzip berücksichtigt. Page Templates sind für Programmierer und Designer, die zusammen arbeiten müssen, um eine Webseite mit dynamischem Inhalt gestalten zu wollen. Wenn Sie alle Ihre Webseiten mit einem Texteditor erstellen und pflegen, brauchen Sie sich nicht für Page Templates zu interessieren. Dann wiederum sind Sie einfacher zu gebrauchen und zu verstehen als die Alternative, DTML.

### 2.1.2 Schon wieder eine neue Template Sprache?

Es gibt eine Menge von Template-basierten Systemen; manche sehr populär, wie z. B. ASP, JSP und PHP. Schon von Anfang an enthielt Zope eine Template Sprache namens DTML. Warum jetzt eine neue einführen?

Erstens ist keines der Template-Systeme auf HTML-Entwickler zugeschnitten. Ist eine Seite erst mal in ein Template verwandelt, ist es kein gültiges HTML mehr, womit es schwer wird, außerhalb der Anwendung damit zu arbeiten. Jedes der angeführten Template Systeme verletzt entweder das erste oder das zweite Prinzip von Zope Page Templates auf die eine oder andere Art. Programmierer sollten eigentlich nicht die Arbeit der Designer entfremden, indem sie HTML Seiten in Software-Module verwandeln. XMLC, Teil des Enhydra Projekts, hat ebenfalls unser Ziel im Auge, aber es verlangt von einem Programmierer die Erstellung von einer nicht unwesentlichen Menge von Java Code für die Unterstützung jedes Templates.

Zweitens leiden all diese System unter der fehlenden Trennung von Präsentation, Logik und Dateninhalt. Die Verletzung des dritten Prinzips verringert die Skalierbarkeit von Content Management und der Anstrengungen bei der Entwicklung von Webseiten.

### 2.1.3 Die Anwendung der Prinzipien

Page Templates können einfach aus dem Internet von der [Page Template Seite](http://www.zope.org/Members/4am/ZPT) (<http://www.zope.org/Members/4am/ZPT>). heruntergeladen und in Ihrem Zope installiert werden. Dort finden sich ausführliche Anweisungen wie das Zope Page Template Produkt heruntergeladen und installiert werden kann, welches gebraucht wird, um die Beispiele hier im Text auszuprobieren. Die aktuelle Version enthält bereits Page Templates. Page Templates verwenden die Template Attribute Language (TAL). TAL besteht aus speziellen Tag-Attributen. So zum Beispiel kann ein dynamischer Seitentitel aussehen:

```
<title tal:content="here/title">Page Title</title>
```

Das ‚tal:content‘-Attribut ist ein TAL-Befehl. Da ein XML-Namensraum verwendet wird (der ‚tal:‘-Teil der Anweisung), stören sich die meisten Werkzeuge nicht daran, daß sie die Anweisung nicht verstehen und entfernen es daher nicht (Ein XML-Namensraum ist ein Bezeichner der den Attributen oder Tags durch einen Doppelpunkt getrennt voransteht). Die TAL-Anweisung ändert weder die Struktur noch die Erscheinung des Templates, wenn es in einen WYSIWYG Editor oder einen Webbrowser geladen wird. Der Name ‚content‘ zeigt an, das der Inhalt des title-Tags gesetzt werden soll. Der Wert ‚here/title‘ ist ein Ausdruck der den Text bereitstellt, der in das Tag geschrieben werden soll.

Für einen Designer, der einen WYSIWYG Editor benutzt, stellt sich der Code als absolut gültiges HTML dar. Die Darstellung des Titels im Editor oder Browser ist genau wie erwartet. Der Designer, der sich nicht um die anwendungsspezifischen TAL-Details kümmern will, sieht nur ein *Modell* des dynamischen Templates, zusammen mit Füllwerten wie z. B. ‚Page Title‘ für den Seitentitel.

Wenn dieses Template in Zope gespeichert und von einem Anwender betrachtet wird, verwandelt Zope diesen statischen Teil in dynamischen Inhalt und ersetzt Page Title mit was immer ‚here/title‘ auch zurückgeben mag. In diesem Fall wird ‚here/title‘ aufgelöst in den Titel des Objekts, dem das Template zugeordnet ist. Die Ersetzung geschieht dynamisch, sobald das Template angezeigt wird.

Diese Beispiel demonstriert auch das zweite Prinzip. Wenn das Template in einem Editor betrachtet wird, agiert der Titel Text als Platzhalter für den dynamischen Titel. Das Template ist ein Beispiel dafür, wie generierte Dokumente aussehen.

Es gibt Kommandos für die Ersetzung gesamter Tags, deren Inhalt oder nur einiger Attribute. Ein Tag kann mehrfach wiederholt oder ganz ausgelassen werden. Teile verschiedener Templates können vereinigt werden, und eine einfache Fehlerbehandlung kann spezifiziert werden. Alle diese Fähigkeiten werden zur Erzeugung von Dokumentstrukturen verwendet. Es können **keine** Unterprogramme erzeugt, Klassen definiert, Schleifen oder Mehrfach-Tests erzeugt oder gar komplexe Algorithmen einfach formuliert werden. Für diese Aufgaben sollte Python verwendet werden.

Die Template-Sprache ist bewußt nicht so mächtig und vielfältig wie sie sein könnte. Sie ist dafür gedacht, in Frameworks Verwendung zu finden (so wie Zope), wo andere Objekte die „business logic“ und Aufgaben übernehmen, die über die Darstellung der Seiten hinausgehen.

Beispielsweise ist die Template-Sprache hilfreich bei der Darstellung einer Fakturierungsseite; es wird eine Zeile pro Position erstellt und Text für die Beschreibung, die Menge und den Preis erstellt. Dafür wird sie sicher nicht gebraucht,

und den Datensatz in einer Datenbank zu speichern oder mit einer Abrechnungseinheit zu interagieren.

### 2.1.4 Page Templates erstellen

Wenn Sie Seiten erstellen, verwenden Sie wahrscheinlich FTP oder WebDAV anstelle des Zope Management Interfaces (ZMI) zur Erzeugung und Bearbeitung von Page Templates. Fragen Sie Ihren Administrator nach Einzelheiten.

Für die ganz kleinen Beispiele in diesem Artikel ist es wesentlich einfacher, das ZMI zu verwenden. Weitere Informationen über die Benutzung von WebDAV oder FTP mit Zope finden Sie in ["The Zope Book"](http://www.zope.org/Members/michel/ZB/) (<http://www.zope.org/Members/michel/ZB/>) oder in Jeffrey Shell's letzten Artikel über [WebDAV](http://www.zope.org/Documentation/Articles/WebDAV) (<http://www.zope.org/Documentation/Articles/WebDAV>).

Falls Sie ein Zope-Administrator oder Programmierer sind, verwenden Sie sicher gelegentlich das ZMI, sowie Editoren wie emacs oder cadaver. Schauen Sie in ["The Zope Book"](http://www.zope.org/Members/michel/ZB/) (<http://www.zope.org/Members/michel/ZB/>) nach Anleitungen für die Anpassung von Zope für das Arbeiten mit diversen Clients. Benutzen Sie Ihren Webbrowser für die Anmeldung beim ZMI wie sie es bei Zope machen würden. Wählen Sie einen Ordner (das Wurzelverzeichnis ist in Ordnung), und wählen Sie "Page Template" von der "Add"-Dropdownliste aus. Geben Sie 'simple\_page' in das Feld 'Id' auf der "Hinzufügen"-Seite ein. Bestätigen Sie dann mit dem "Add and Edit"-Button.

Nun sollten Sie die Bearbeitungshauptansicht für das neue Page Template vor sich haben. Der Titel ist leer, der Inhaltstyp (content type) ist 'text/html' und ein Standardtext befindet sich im Editierfeld. Nun werden Sie eine sehr einfache, dynamische Seite erstellen. Geben Sie die Worte 'a simple page' in das Titelfeld ein. Editieren Sie dann den Template Text, damit er folgendermaßen aussieht:

```
Dies ist <b tal:replace="template/title">der Titel</b>.
```

Betätigen Sie nun den "save changes"-Button. Es wird eine Bestätigung angezeigt, daß die Änderungen gespeichert wurden. Wird Text eingefügt, der mit '<-- Page Templates Diagnostics' beginnt, prüfen Sie, ob Sie das Beispiel korrekt abgeschrieben haben und speichern Sie es erneut.

Der Fehlerkommentar muß nicht entfernt werden - wurde der Fehler beseitigt, verschwindet er automatisch. Klicken Sie auf den Tabulator "Test" in der Oberfläche. Es sollte eine eine fast leere Seite erscheinen, auf der oben 'Dies ist eine einfache Seite' steht.

Gehen Sie zurück, und klicken dann auf den "Browser HTML Source"-Link unterhalb des Feldes 'content-type'. Der Quelltext des Templates wird angezeigt. Der Text lautet 'Dies ist der Titel'. Gehen Sie erneut auf die Bearbeitungsmaske zurück.

### 2.1.5 "Simple Expressions"

Der Text 'template/title' in dem einfachen Beispiel ist ein *Pfad-Ausdruck* (*path expression*). Dies ist der am häufigsten verwendete Ausdruckstyp der durch die TAL Expression Syntax (TALES) definierten Typen. Der Ausdruck 'template/title' liest die Titel-Eigenschaft des Templates aus. Hier ein paar weitere, häufig verwendete Ausdrücke:

- ✎ request/URL: Die URL des aktuelle "web request"
- ✎ user/getUserName: der Benutzername des angemeldeten Anwenders
- ✎ container/objectIds: Eine Liste mit Ids derjenigen Objekte, die im selben Verzeichnis liegen wie das Template

Jeder Pfad fängt mit dem Namen einer Variablen an. Enthält die Variable bereits die gewünschte Information, sind Sie fertig. Andernfalls folgt der Variablen ein Schrägstrich (/), gefolgt vom Namen der Untervariablen. Es ist sehr gut möglich,

das Sie sich den Pfad aus mehreren Hierarchiestufen von Untervariablen zusammenbauen müssen.

Es existiert eine kleine Menge fest eingebauter Variablen, wie beispielsweise 'request' und 'user'; sie werden zu einem späteren Zeitpunkt noch vollständig aufgelistet und erläutert. Sie werden auch lernen, eigene Variablen zu deklarieren.

### 2.1.6 Text einfügen

In dem "simple\_page"-Template, wurde die 'tal:replace'-Anweisung in dem Tag <b> verwendet. Beim Testen wurde das gesamte Tag durch den Titel des Templates ersetzt. Bei der Anzeige im Browser wurde der Titel in Fettschrift angezeigt. Das Tag <b> wurde von uns absichtlich gewählt, um den Unterschied hervorzuheben.

Um dynamischen Text innerhalb von statischem Text zu verwenden, wird üblicherweise die Anweisung 'tal:replace' in einem <span>-Tag verwendet. Fügen die die folgenden Zeilen zu dem Beispiel hinzu:

```
<br>
Die URL ist <span
tal:replace="request/URL">URL</span>.
```

Das <span>-Tag ist ein strukturierendes Tag, kein visuelles. Der Text lautet 'Die URL ist URL', wenn er in einem Editor oder Browser betrachtet wird. Betrachtet man die gerenderte Version, dann sieht der Text allerdings etwa so aus:

```
<br>
Die URL ist http://localhost:8080/simple_page.
```

Achten Sie unbedingt darauf, beim Editieren nicht die <span>-Struktur zu zerstören bzw. keine formatierenden Tags wie <b> oder <font> zu verwenden, da diese ebenfalls ersetzt werden würden.

Wollen Sie Text innerhalb eines Tags setzen, das Tag selbst jedoch unberührt lassen, müssen Sie die 'tal:content'-Anweisung verwenden. Um den Titel der Beispielseite zur Titel-Eigenschaft des Templates zu machen, müssen dem Beispiel folgenden Zeilen oberhalb des bisherigen Codes hinzugefügt werden:

```
<head>
<title tal:content="template/title">Der Titel</title>
</head>
```

Wenn Sie den "Test"-Tabulator in einem neuen Fenster öffnen, wird der Titel des Fensters "a Simple Page" lauten.

### 2.1.7 Wiederholte Strukturen

Nun werden wir etwas Kontext zu unserer Seite hinzufügen, in Form einer Liste aller sich im gleichen Verzeichnis befindlichen Objekte. Wir werden eine Tabelle mit nummerierten Zeilen erzeugen, mit Spalten für die Id, den meta-type und den Titel. Fügen Sie die folgenden Zeilen am Ende des Beispiels ein:

```
<table border="1" width="100%">
<tr>
<th>#</th><th>Id</th><th>MetaType
</th><th>Title</th>
</tr>
<tr tal:repeat="item container/objectValues">
<td tal:content="repeat/item/number">#</td>
<td tal:content="item/id">Id</td>
<td tal:content="item/meta_type">Meta-Type</td>
<td tal:content="item/title">Title</td>
</tr>
</table>
```

Die 'tal:repeat'-Anweisung in der Tabellenzeile bedeutet "wiederhole diese Zeile für jedes Element in der Liste der Objekt-Werte des Containers". Die 'repeat'-Anweisung steckt jeweils genau ein Objekt aus der Liste in die 'item'-Variable, und erzeugt eine Kopie der Zeile, die diese Variable (hier: 'item') verwendet. Der Wert 'item/id' in jeder Zeile ist die Id des jeweiligen Objekts der Zeile. Anstelle des Namens 'item' kann jeder beliebige Name verwendet werden, solange er mit einem Buchstaben beginnt und nur Zahlen, Buchstaben und den Unterstrich (\_) enthält. Die Variable existiert nur innerhalb des <tr>-Tags; wird außerhalb des Tags versucht, auf den Wert zuzugreifen, erhält man einen Fehler.

Der 'tal:repeat' Variablenname wird auch verwendet, um Informationen über die gegenwärtige Wiederholung zu erlangen. Wird der Variablenname in einem Pfad-Ausdruck der Variablen 'repeat' nachgestellt, so gibt 'index' die aktuelle Anzahl der Wiederholungen beginnend bei 0 zurück, 'number' die Anzahl beginnend bei 1, und 'Letter' erzeugt fortlaufende Buchstaben beginnend bei "A". Es gibt auch noch weitere Möglichkeiten. Also ergibt der Ausdruck 'repeat/item/number' aus unserem Beispiel eine 1 in der ersten Wiederholung, eine 2 in der zweiten usw.

Da 'tal:repeat' Schleifen ineinander verschachtelt werden können, können mehrere Schleifen gleichzeitig aktiv sein. Aus diesem Grund muß auch 'repeat/item/number' geschrieben werden anstelle von 'repeat/number'. Die gewünschte Schleife muß also durch den Schleifenamen spezifiziert werden.

### 2.1.8 Bedingte Elemente

Beim Betrachten der Tabelle werden wir feststellen, daß sie ziemlich langweilig aussieht. Wir können sie dadurch interessanter machen, wenn wir abwechselnde Zeilen durch Schattierung voneinander abheben. Kopieren Sie die zweite Zeile der Tabelle und editieren Sie sie so, daß sie folgendermaßen aussieht:

```
<table border="1" width="100%">
  <tr>
    <th>#</th><th>Id</th><th>Meta-Type
    </th><th>Title</th>
  </tr>
  <tbody tal:repeat="item container/objectValues">
    <tr bgcolor="#EEEEEE"
      tal:condition="repeat/item/even">
      <td tal:content="repeat/item/number">#</td>
      <td tal:content="item/id">Id</td>
      <td tal:content="item/meta_type">Meta-Type</td>
      <td tal:content="item/title">Title</td>
    </tr>
    <tr tal:condition="repeat/item/odd">
      <td tal:content="repeat/item/number">#</td>
      <td tal:content="item/id">Id</td>
      <td tal:content="item/meta_type">Meta-Type</td>
      <td tal:content="item/title">Title</td>
    </tr>
  </tbody>
</table>
```

Die 'tal:repeat'-Anweisung hat sich nicht verändert, wir haben sie nur in das <tbody>-Tag verschoben. Dies ist ein Standard-HTML-Tag für die Gruppierung der Tabellenzeilen, und genau so benutzen wir es auch. Es gibt zwei Zeilen in der Tabelle, die jeweils identische Spalten haben, aber die eine ist grau und die andere ist weiß.

Im HTML-Code sehen wir beide Zeilen. Würden wir die 'tal:condition'-Anweisungen weglassen, so würden für jedes Objekt in 'item' zwei Zeilen in der Tabelle dargestellt werden. Die 'tal:condition'-Anweisung stellt sicher, daß die erste Zeile nur

für gerade, die zweite Zeile nur für ungerade Durchläufe ausgegeben wird.

Ist die Bedingung der Anweisung wahr, so passiert überhaupt nichts (Außer daß das Tag so wie es ist ausgegeben wird.). Ist die Bedingung unwahr, so wird das ganze Tag mit seinem gesamten Inhalt entfernt. Die Eigenschaften 'odd' und 'even' des 'repeat/item' Objekts sind entweder 0 oder 1. Die Zahl 0, ein leerer String, eine leere Liste und die Standardvariable 'nothing' representieren alle den Wahrheitswert 'false'. Beinahe jeder andere Wert bedeutet 'true', einschließlich von 0 verschiedene Zahlen und beliebige Strings, sogar wenn sie Leerzeichen enthalten.

### 2.1.9 Variablen definieren

Das Beispiel-Template wird immer mindestens eine Zeile anzeigen, da das Template selbst eines der aufgelisteten Objekte ist. Unter anderen Umständen will man vielleicht eine spezielle Behandlung für die Möglichkeit einführen, daß die Tabelle leer ist. Angenommen, die Tabelle soll überhaupt nicht angezeigt werden, falls sie leer ist. Dies kann dadurch erreicht werden, daß die 'tal:condition'-Anweisung zur Tabelle hinzugefügt wird.

```
<table border="1" width="100%"
  tal:condition="container/objectValues">
```

Sollten keine Objekte vorhanden sein, so wird kein Teil der Tabelle mit ausgegeben. Sind Objekte vorhanden, dann wird der Ausdruck 'container/objectValues' zweimal ausgewertet. Dies ist etwas ineffizient. Ebenso muß man darauf achten, daß im Falle einer Änderung des Ausdrucks der Ausdruck an beiden Stellen im Code geändert werden muß.

Um dieses Problem zu vermeiden, kann eine Variable deklariert werden, die den Inhalt der Liste enthält. Die Variable kann dann sowohl in 'tal:condition' als auch in 'tal:repeat' verwendet werden. Wir verändern die ersten paar Zeilen wie folgt:

```
<table border="1" width="100%"
  tal:define="items container/objectValues"
  tal:condition="items">
  <tr>
    <th>#</th><th>Id</th><th>Meta Type
    </th><th>Title</th>
  </tr>
  <tbody tal:repeat="item items">
```

Die 'tal:define'-Anweisung erzeugt die Variable 'items', die innerhalb des gesamten <table>-Tags verwendet werden kann. Beachten Sie auch, daß man sehr wohl zwei TAL-Attribute in einem Tag verwenden kann. Tatsächlich kann man so viele verwenden wie man will. Sie werden der Reihe nach ausgewertet. Zum Beispiel deklariert das erste Attribut eine Variable und weist ihr einen Wert zu, das zweite Attribut wertet die Variable in einer Bedingung aus und prüft auf wahr oder falsch.

Nehmen wir nun an, daß wir anstatt die leere Tabelle einfach auszulassen eine Meldung ausgeben wollen. Um das zu tun, schreiben wir folgendes über die Tabelle:

```
<h4 tal:condition="not:container/objectValues">There
  Are No Items</h4>
```

Die 'items'-Variable kann hier nicht verwendet werden, da Sie noch gar nicht definiert ist. Wird die Variablendeklaration in das <h4>-Tag verlagert, so kann es in der Tabelle nicht mehr verwendet werden, da es dann eine lokale Variable für das <h4>-Tag ist. Man kann die Definition in ein Tag einschließen, welches sowohl die Überschrift als auch die Tabelle klammert, aber es gibt eine einfachere Lösung. Durch das Voran-

stellen des Schlüsselworts 'global' vor den Variablennamen erlangt die Variable Gültigkeit ab der Deklaration bis zum Ende des Templates. Ein Beispiel:

```
<h4 tal:define="global items container/objectValues"
  tal:condition="not:items">There Are No Items</h4>
<table border="1" width="100%"
  tal:condition="items">
```

Das 'not:' in der ersten 'tal:condition'-Anweisung ist ein Präfix, das jedem Ausdruck vorangestellt werden kann. Ist der Ausdruck wahr, dann ist 'not:' falsch und umgekehrt.

### 2.1.10 Attribute ändern

Die meisten, wenn nicht gar alle Objekte, die im Beispiel-Template aufgelistet werden, besitzen die 'icon'-Eigenschaft, die den Pfad zu dem Icon für den jeweiligen Objekt-Typ enthält. Um diese Icons in der Spalte für den Meta-Typ anzuzeigen, müssen Sie diesen Pfad in das 'src'-Attribut eines <img>-Tags setzen. Die Spalte Meta-Typ muß wie folgt abgeändert werden:

```
<td>
  <span tal:replace="item/meta_type">Meta-Type</span>
</td>
```

Das 'tal:attributes'-Anweisung ersetzt das 'src'-Attribut des Bildes mit dem Wert von 'item/icon'. Der Wert des 'src'-Attributes im Template fungiert als Platzhalter, damit die Referenz auf das Icon im Zweifel nicht gebrochen ist und die Größe des Icons dann auch stimmt.

Da das 'tal:content'-Attribut in der Zellenzelle den gesamten Zelleninhalt mit dem Meta-Typ-Text ersetzt hätte (inklusive dem Bild), mußte es entfernt werden.

### 2.1.11 Zusammenfassung

Diese Einführung in Page Templates zeigt, wie man einfache Templates erstellt und bearbeitet, und wie man die TAL- und TALES-Sprache benutzt, ohne von HTML oder den gewohnten Werkzeugen und Editoren abzukommen. Wie gezeigt wurde, ermöglichen Page Templates die nahtlose Zusammenarbeit von Designern und Programmierern, ohne das Sie sich gegenseitig Zugeständnisse machen müssen.

Im nächsten Artikel dieser Serie werden wir fortgeschrittenere Funktionen und Eigenschaften von Page Templates untersuchen.

## 2.2 Zope Page Templates: Fortgeschrittene Konzepte

Von Evan Simpson

In diesem [Abschnitt](http://www.zope.org/Documentation/Articles/ZPT1) (<http://www.zope.org/Documentation/Articles/ZPT1>) betrachten wir fortgeschrittene Konzepte von TAL. In der zweiten Hälfte dieses Artikels beschäftigen wir uns mit der TAL Expression Syntax (TALES).

### 2.2.1 "Mixing"- und "Matching"-Anweisungen

Wie wir bereits im Beispiel-Template gesehen haben, kann mehr als eine TAL-Anweisung in einem Tag stehen. Allerdings sollte man sich dreier Einschränkungen bewußt sein.

1. Es darf nur jeweils eine Anweisung eines Anweisungs-typs in einem einzelnen Tag verwendet werden. Da HTML nicht die mehrfache Verwendung eines Attributes mit demselben Namen erlaubt, ist beispielsweise die mehrfache Verwendung von 'tal:define' nicht erlaubt.
2. Die beiden Anweisungen 'tal:content' und 'tal:replace' können nicht im selben Tag verwendet werden, da sich ihre Funktionalitäten überschneiden bzw. miteinander in Konflikt stehen.
3. Die Reihenfolge, in der die TAL-Attribute in einem Tag stehen, beeinflußt nicht die Ausführungsreihenfolge. Ungeachtet der Anordnung werden die Attribute immer in folgender, feststehender Reihenfolge ausgewertet: define, condition, repeat, content/replace, attributes

Um diese Reihenfolge zu umgehen, können weitere Tags hinzugefügt und die TAL-Attribute entsprechend darauf verteilt werden. Falls es kein passendes Tag gibt, kann einfach <span> oder <div> verwendet werden.

Soll beispielsweise eine Variable für jede Wiederholung eines Paragraphen definiert werden, so kann das 'tal:define'-Attribut nicht in das selbe Tag geschrieben werden wie die 'tal:repeat'-Anweisung, da die Definition vor der Wiederholung stattfinden würde. Stattdessen würde man eher etwas schreiben wie:

```
<div tal:repeat="p phrases">
  <p tal:define="n repeat/p/number">
    Phrase Nummer <span tal:replace="n">1</span> is
    "<span tal:replace="p">Phrase</span>". </p>
</div>
```

```
<p tal:repeat="p phrases">
  <span tal:define="n repeat/p/number">
    Phrase Nummer <span tal:replace="n">1</span> is
    "<span tal:replace="p">Phrase</span>". </span>
</p>
```

### 2.2.2 Mehrteilige Anweisungen

Sollen in einem Tag mehrere Attribute gesetzt werden, kann dies nicht einfach durch die wiederholte Verwendung der 'tal:attributes'-Anweisung erfolgen. Die Verteilung über mehrere Tags wie oben ist natürlich nutzlos. Sowohl die 'tal:attributes'-Anweisung als auch die 'tal:define'-Anweisung können aus mehreren Teilen zusammengesetzt sein. Die Teile werden durch ein Semikolon (;) voneinander getrennt. Somit muß aber auch jedes Semikolon innerhalb eines solchen Teils durch das Voranstellen eines zweiten Semikolons gekennzeichnet werden (;;). Hier ist ein Beispiel für das Setzen zweier Attribute in einem Tag:

```

```

Und hier eine Menge von Variabeldeklarationen:

```
<span tal:define="global logo here/logo.gif; ids
  here/objectIds">
```

### 2.2.3 String-Ausdrücke

String-Ausdrücke erlauben die Vermengung von Pfad-Ausdrücken mit normalem Text. Der gesamte Text hinter einem vorangestellten 'string:' wird nach Pfad-Ausdrücken durchsucht. Vor jedem Pfad-Ausdruck muß ein Dollar-Zeichen (\$) stehen. Hat es mehr als einen Teil, oder soll es von dem nachfolgenden Text unterschieden werden, muß der Ausdruck in geschweiften Klammern stehen ({,}). Da sich der Text innerhalb eines Attributwertes befindet, können doppelte Anführungszeichen nur durch die Verwendung der Entity-Syntax (" " ") eingefügt werden. Da das Dollarzeichen Pfadausdrücke kennzeichnet, muß das Buchstabensymbol für Dollar als doppeltes Dollarzeichen (\$\$) geschrieben werden. Ein paar Beispiele:

```
"string:Just text."
"string:© $year, by Me."
"string:Three ${vegetable}s, please."
"string:Your name is ${user/getUserName}!"
```

### 2.2.4 "Nocall"-Pfad-Ausdrücke

Ein gewöhnlicher Pfad-Ausdruck versucht das Objekt darzustellen, das er abrufen. Das bedeutet, falls das Objekt eine Funktion, ein Skript, eine Methode oder irgend eine andere Art von ausführbarem Objekt ist, wird der Ausdruck berechnet als Ergebnis des Objekt-Aufrufs. Für gewöhnlich will man das auch, aber eben nicht immer. Will man beispielsweise ein DTML-Dokument in eine Variable stecken, so daß man seine Eigenschaften abrufen kann, so kann kein einfacher Pfad-Ausdruck verwendet werden, da dadurch das Dokument "gerendert", also als String dargestellt werden würde. Stellt man das Prefix 'nocall:' vor einen Pfad-Ausdruck, wird der Aufruf unterdrückt und nur das Objekt zurückgegeben. Ein Beispiel:

```
<span tal:define="doc nocall:here/aDoc"
  tal:content="string:${doc/id}: ${doc/title}">
  Id: Title</span>
```

Dieser Ausdrucks-Typ ist ebenfalls nützlich, wenn Sie eine Variable definieren wollen, die eine Funktion oder Klasse eines Moduls für die Verwendung in Python enthält.

### 2.2.5 Python Ausdrücke

Ein Python-Ausdruck beginnt mit 'python:', gefolgt von einem in der Programmiersprache Python geschriebenen Ausdruck. Im Abschnitt über die Erstellung von Python-Ausdrücken finden Sie mehr darüber.

### 2.2.6 Standard-Variablen

Wir haben bereits Beispiele für Standard-Variablen wie 'template', 'user', 'repeat' und 'request' gezeigt. Hier folgt nun eine vollständige Liste aller verfügbaren Variablen und ihre Erklärung:

- ❏ *nothing*: ein Wert für 'false', gleichbedeutend einem leeren String, der verwendet werden kann in 'tal:replace' oder 'tal:content' um Tags oder deren Inhalt zu entfernen bzw. zu umgehen. Wird ein Attribut auf 'nothing' gesetzt, wird es aus dem Tag entfernt (bzw. gar nicht eingefügt), nicht wie bei einem leeren String.
- ❏ *default*: ein Spezialwert, der gar nichts ändert, wenn er in 'tal:replace', 'tal:content' oder 'tal:attributes' verwendet wird. Es läßt den Template-Text wo er ist.
- ❏ *options*: die 'keyword'-Argumente - falls es welche gibt, die dem Template übergeben werden.
- ❏ *attrs*: Ein Dictionary gefüllt mit den Attributen des gegenwärtigen Tags im Template. Die Schlüssel sind die Attribut-Namen, und die Werte sind die originalen Attribut-Werte innerhalb des Templates.
- ❏ *root*: das Zope root-Objekt. Verwenden Sie es, um Zope-Objekte an bestimmten Positionen anzusprechen, unabhängig von der Position Ihres Templates.
- ❏ *here*: das Objekt, über das das Template aufgerufen wurde. Oft ist es das selbe wie 'container', kann bei der Verwendung von "acquisition" aber auch ein ganz anderes sein. Benutzen Sie es zur Lokalisierung von Zope-Objekten, die Sie an unterschiedlichen Stellen zu finden erwarten, je nach dem wie das Template aufgerufen worden ist.
- ❏ *container*: für gewöhnlich ein Ordner, in dem das Template enthalten ist. Benutzen Sie den Container, um Zope-Objekte anzusprechen, die relative Positionen zu dem ständigen Ort des Templates haben.
- ❏ *modules*: eine Sammlung von Python-Modulen, die in den Templates verfügbar sind. Weiteres dazu im Abschnitt über Python-Ausdrücke.

### 2.2.7 Wechselnde Pfade

Der Pfad 'template/title' existiert garantiert während der gesamten Nutzung des Templates. Dagegen existieren manche Pfade wie etwa 'request/form/x' zeitweilig während der Darstellung des Templates nicht. Dies verursacht dann normalerweise einen Fehler, wenn der Pfad ausgewertet wird. Falls der Pfad nicht existiert, gibt es oft als Ausweichlösung einen anderen Pfad, der dann als Ersatz verwendet werden soll. Existiert der Pfad 'request/form/x' beispielsweise nicht, will man vielleicht den Pfad 'here/x' verwenden. Dies kann durch eine Auflistung der Pfade in der Reihenfolge ihrer Priorisierung erreicht werden. Die einzelnen Pfade werden durch den horizontalen Balken (|) getrennt:

```
<h4 tal:content="request/form/x | here/x">Header</h4>
```

Zwei in diesem Zusammenhang sehr nützliche Variablen sind 'nothing' und 'default' als jeweils letzter Eintrag in der Liste. Verwenden Sie 'nothing' um einen leeren Rückgabewert zu erhalten, falls keiner der Pfade aufgelöst werden kann, oder 'default', um den Beispieltext anstelle des Ergebnisses anzuzeigen.

Die Gültigkeit eines Pfades kann auch direkt mit dem 'exists:'-Präfix-Ausdruck überprüft werden. Ein Pfad-Ausdruck mit dem 'exists:'-Ausdruck ist entweder wahr, falls der Pfad existiert, oder falsch. Das folgende Beispiel zeigt eine Fehlermeldung nur an, wenn sie mit dem Request übergeben wurde:

```
<h4 tal:define="err request/form/errmsg | nothing"
  tal:condition="err" tal:content="err">Fehler!</h4>
```

```
<h4 tal:condition="exists:request/form/errmsg"
  tal:content="request/form/errmsg">Fehler!</h4>
```

### 2.2.8 Platzhalter-Elemente

Durch die Verwendung der Standard-Variablen 'nothing' Seiten-Elemente einfügen, die zwar im Page Template sichtbar sind, nicht jedoch im daraus generierten Text:

```
<tr tal:replace="nothing">
  <td>10213</td><td>Beispiel Position</td><td>$15.34</td>
</tr>
```

Dies kann sehr nützlich sein, um Teile einer Seite auszufüllen, die in der generierten Ausgabe mehr Platz einnehmen, als im Template selbst. Um ein Beispiel zu geben: eine Tabelle die für gewöhnlich etwa zehn Zeilen hat, hat im Template nur eine. Durch das Hinzufügen von neun Platzhalter-Zeilen entspricht das Template-Layout mehr der tatsächlichen Ausgabe.

### 2.2.9 Einfügestrukturen

Normalerweise quoten (Als Quoting wird die Umsetzung von Zeichen bezeichnet, die als Spezialzeichen zur Strukturierung von Text verwendet werden. So muß in Text auf einer HTML-Seite z. B. ein '<' als '&lt;' geschrieben werden.) die Anweisungen 'tal:content' und 'tal:replace' den Text, den Sie einfügen, beispielsweise wird ein '<' zu einem '&lt;'. Soll tatsächlich ungequoteter Text eingefügt werden, so muß dem ganzen Ausdruck das Wort 'structure' vorangestellt werden. Ist etwa eine Variable 'copyright' gegeben, so erzeugen die folgenden Zeilen Code etwa 'Copyright 2001' oder entsprechend "© 2001":

```
<span tal:replace="copyright">Copyright 2000</span>
<span tal:replace="structure copyright">
  Copyright 2000</span>
```

Diese Eigenschaft ist besonders dann von Nutzen, wenn ein HTML-Codefragment eingefügt werden soll, das in einer Variablen steht oder von einem anderen Zope-Objekt generiert worden ist. Zum Beispiel können Sie Newsitems haben, die einfaches HTML-Markup enthalten, etwa mit den Tags für "bold" oder "italic". Beim Anzeigen auf einer News-Seite sollen solche Textformatierungen natürlich beibehalten werden. In diesem Fall könnte man schreiben:

```
<p tal:repeat="article topnewsitems"
  tal:content="structure article">A News Article</p>
```

### 2.2.10 Python Grundlagen

Die Sprache Python ist einfach und ausdrucksstark. Sollten Sie noch nie mit Python in Berührung gekommen sein, dann lohnt es sich eines der hervorragenden Python Tutorial von der [Python Webseite](http://www.python.org) (<http://www.python.org>) zu lesen.

Ein Page Template kann all das enthalten, was in der Sprache Python als Ausdruck gilt. Kontrollstrukturen wie 'if' and 'while' können nicht verwendet werden. Es gelten die Sicherheits-einschränkungen von Zope.

#### Vergleiche

Eine Sache, bei der Python-Ausdrücke unverzichtbar sind, ist in 'tal:condition'-Anweisungen. Für gewöhnlich wollen wir zwei Zahlen oder Strings miteinander vergleichen, und dafür gibt es keine andere Möglichkeit. Verwendet werden dürfen die Operatoren '<' (less than), '>' (greater than), '==' (equal to) und '!' (not), sowie die bool'schen Operatoren 'not', 'and' und 'or'. Ein Beispiel:

```
<p tal:repeat="widget widgets">
  <span tal:condition="python:widget.type == 'gear'">
```

```
Gear #<span tal:replace="repeat/widget/number">1
</span>:
<span tal:replace="widget/name">Name</span>
</span>
</p>
```

Manchmal soll zwischen mehreren Werten entschieden werden, abhängig von einer oder mehrerer Bedingungen. Dies kann mit der "test"-Funktion gemacht werden:

```
You <span tal:define="name user/getUserName"
  tal:replace="python:test(name=='Anonymous User',
    'need to log in', default)">
are logged in as
  <span tal:replace="name">Name</span>
  </span>

<tr tal:define="oddrow repeat/item/odd"
  tal:attributes="class python:test(oddrow, 'oddclass',
    'evenclass')">
```

### Die Verwendung anderer Ausdrucks-Typen

In einem Python-Ausdruck können andere Ausdrucks-Typen verwendet werden. Jeder Typ hat eine zugehörige Funktion gleichen Namens, eingeschlossen 'path()', 'string()', 'exists()' und 'nocall()'. Dies erlaubt es, folgende Ausdrücke zu schreiben:

```
"python:path('here/%s/thing' % foldername)"
"python:path(string('here/$foldername/thing'))"
"python:path('request/form/x') or default"
```

Das letzte Beispiel hat eine leicht unterschiedliche Bedeutung zu dem Pfadausdruck 'request/form/x | default', da es 'default' verwenden wird, wenn 'request/form/x' nicht existiert oder falsch ist.

### 2.2.11 Zope-Objekte referenzieren

Ein großer Teil der Mächtigkeit von Zope rührt daher, daß spezielle Objekte miteinander verbunden werden. Ihre Page Templates können Skripte verwenden, SQL-Methoden, Kataloge, und eigene Inhaltsobjekte. Um sie benutzen zu können, muß man wissen, wie man auf sie zugreift.

Objekteigenschaften sind für gewöhnlich Attribute, also kann zum Beispiel auf den Titel des Templates mit dem Ausdruck 'template.title' zugegriffen werden. Die meisten Zope-Objekte unterstützen Aquisition, die es erlaubt, von Super-Objekten Eigenschaften zu erhalten. Das bedeutet, daß der Python-Ausdruck 'here.Control\_Panel' zum Control Panel Objekt des 'root'-Ordnern ausgewertet wird. Objekt-Methoden sind Attribute, wie in 'here.objectIds' und 'request.set'. Auf Objekte, die in einem Ordner enthalten sind, kann als Ordneureigenschaft zugegriffen werden. Da sie jedoch oftmals Ids haben, die keine gültigen Python-Identifizierer sind, kann die gewohnte Notation nicht verwendet werden. Beispielsweise muß anstatt 'here.penguin.gif' 'getattr(here, penguin.gif)' geschrieben werden. Manche Objekte wie 'request', 'modules' und Zope-Ordner unterstützen Direktzugriff:

```
request['URL'], modules['math'], and here['thing']
```

Wird der Direktzugriff auf einem Ordner verwendet, so wird der Name des Ordners nicht akquiriert; der Zugriff hat also nur dann Erfolg, wenn sich tatsächlich ein Objekt mit dem entsprechenden Namen in dem Ordner befindet.

Wie wir schon in früheren Kapiteln gesehen haben, erlauben Pfad-Ausdrücke das Verbergen von Details darüber, wie man von einem Objekt zum anderen gelangt. Zope versucht zuerst einen Attribut-Zugriff, dann einen Direktzugriff. Man kann so-

wohl 'here/images/penguin.gif' anstelle von 'python.getattr (here.images, penguin.gif)' als auch 'request/form/x' anstelle von 'python:request.form![x]' schreiben.

Der Kompromiss ist der, daß Pfad-Ausdrücke nicht diese Detailliertheit erlauben. Hat man zum Beispiel eine Form-Variable "get", muß man 'python:request.form!['get']' schreiben, da 'request/form/get' die "get"-Methode des Form-Dictionary repräsentiert.

### 2.2.12 Die Verwendung von Skripten

Skript-Objekte werden häufig zur Kapselung von "business logic" und für komplizierte Datenverarbeitung verwendet. Jedes mal, wenn Sie sich dabei ertappen, jede Menge TAL-Anweisungen zu schreiben mit komplizierten Ausdrücken darin, sollten Sie darüber nachdenken, ob die Aufgabe nicht besser mit einem Skript zu erledigen ist.

Jedes Skript hat eine Liste von Parametern, von denen es erwartet, daß diese beim Aufruf mitgegeben werden. Ist diese Liste leer, so kann das Skript über einen Pfad-Ausdruck aufgerufen werden. Sonst wird ein Python-Ausdruck benötigt, der wie folgt aussieht:

```
"python:here.myscript(1, 2)"
"python:here.myscript('arg', foo=request.form['x'])"
```

Soll mehr als nur ein einfaches Ergebnis von dem Skript an das Page Template zurückgegeben werden, ist es immer eine gute Idee, dies in einem Dictionary zu tun. Dann wird einfach eine Variable definiert, die alle Daten enthält und es werden Pfad-Ausdrücke verwendet, um auf die einzelnen Ergebniskomponenten zuzugreifen. Ein Beispiel:

```
getPerson returns this: {'name': 'Fred', 'age': 25}
```

```
<span tal:define="person here/getPerson"
  tal:replace="string:${person/name} is
  ${person/age}">
  Name is 30</span> years old.
```

### 2.2.13 DTML aufrufen

Anders als Skripte haben DTML-Methoden keine expliziten Parameterlisten. Stattdessen erwarten Sie die Übergabe eines Clients, eines Mappings und eines Schlüsselwort-Argumente. Diese werden für die Erzeugung eines Namensraums verwendet.

Wenn der ZPublisher ein DTML-Objekt publiziert, wird der Kontext des Objekts als Client und der REQUEST als Mapping übergeben. Ruft ein DTML-Objekt ein anderes auf, übergibt es seinen eigenen Namesraum als Mapping, und keinen Client. Wird ein Pfad-Ausdruck zur Darstellung eines DTML-Objekts verwendet, werden die Variablen mit einem Namespace aus 'here' und 'request' übergeben. Dies bedeutet, daß ein DTML-Objekt die selben Namen verwenden kann, so als wäre es im selben Kontext wie das Template publiziert worden, zusammen mit den Variablennamen, die im Template verwendet werden.

### 2.2.14 Python Module

Die Sprache Python wird mit einer Menge von Modulen geliefert, die Python-Programmen einen großen Funktionsumfang bereitstellen. Jedes Modul ist eine Sammlung von Python-

Funktionen, Daten und Klassen zu einem bestimmten Zweck, wie beispielsweise mathematische Berechnungen oder reguläre Ausdrücke.

Verschiedene Module, wie 'math' oder 'string', sind standardmäßig in Python-Ausdrücken verfügbar. Beispielsweise kann der Wert von pi aus dem 'math'-Modul ausgelesen werden mit 'python:math.pi'. Um von einem Pfad-Ausdruck darauf zuzugreifen, muß aber die 'modules'-Variable verwendet werden. Die Formulierung lautet 'modules/math/pi'. In "The Zope Book" oder entsprechenden DTML-Tutorials findet sich mehr Information zu diesem Thema.

Das 'string'-Module ist in Python-Ausdrücken durch die 'string'-Ausdruckstyp Funktion verdeckt, also muß hier nicht über die 'modules'-Variable zugegriffen werden. Dies kann direkt in dem Ausdruck geschehen, in dem es verwendet wird. Oder man deklariert eine globale Variable wie folgt:

```
tal:define="global mstring modules/string"
tal:replace="python:mstring.join(slist, ':)"
```

Module können in Pakete (packages) zusammengefaßt werden. Dies ist einfach eine Möglichkeit, zusammengehörige Module zu gruppieren und zu benennen. Zum Beispiel sind alle Python-basierten Skripte in Zope in einem Unterpaket "PythonScripts" des Pakets "Products" zusammengefaßt. Insbesondere bietet das Standard-Modul in diesem Paket eine Reihe nützlicher Formattierungsfunktionen die im DTML-Var-Tag standard sind. Der vollständige Name dieses Moduls ist "Products.PythonScripts.standard". Mit den folgenden Anweisungen kann also darauf zugegriffen werden:

```
tal:define="pps modules/Products.PythonScripts.standard"
tal:define="pps python:modules
['Products.PythonScripts.standard']"
```

Auf die meisten Python-Module kann über Page Templates, DTML oder Skripte solange nicht zugegriffen werden, bis ihnen Sicherheitserklärungen ("security assertions") hinzugefügt worden sind. Dies übersteigt jedoch den Umfang dieses Dokuments und wird im "Zope Security Guide" erläutert.

## 2.3 TALES Spezifikation Version 1.3

Die *Template Attribute Language Expression Syntax* beschreibt Ausdrücke für die Angabe von Eingabedaten für TAL und METAL. Dabei ist TALES nur eine der möglichen Grammatiken für diese Aufgabe. Außerdem kann TALES auch unabhängig von TAL oder METAL verwendet werden.

Die TALES-Ausdrücke sind unten ohne die Angabe von Trennzeichen oder Anführungszeichen, wie sie von höheren Sprachen verwendet werden, angegeben. Denn dies ist die Art und Weise, wie die Ausdruck-Strings an die TALES-Engine übergeben werden müssen. Es folgt die Basisdefinition der TALES-Grammatik:

```
Expression ::= [type_prefix ':'] String
type_prefix ::= Name
```

Die Regeln und Terminale finden sich in der *EBNF* (EBNF steht für ‚erweiterte Backus-Naur-Form‘ und ist eine Notationsart für Grammatiken. Mit EBNF werden Grammatiken durch eine Menge von Regeln beschrieben. Die Regeln ähneln Zuweisungen. Terminale sind dabei Symbole, die nicht weiter aufgelöst werden können, d.h., es gibt keine Regel mehr, bei der sie auf der linken Seite der Zuweisung stehen.). Hier sind einige Beispiele:

```
a/b/c
path:a/b/c
nothing
path:nothing
python: 1 + 2
string:Hello, ${username}
```

Das optionale Typenpräfix 'type\_prefix' bestimmt die Semantik und die Syntax des Ausdrucks, die ihm folgt. Eine gegebene TALES-Implementierung kann jede beliebige Anzahl von Ausdrucks-Typen definieren, mit jeder beliebigen Syntax. Sie bestimmt ebenfalls, welcher Ausdruckstyp verwendet wird, wenn das Präfix weggelassen wird.

### Benötigte Ausdruckstypen

Es werden verschiedene Ausdruckstypen benötigt.

- ☞ *not* - der Ausdruck wird vollständig (rekursiv) ausgewertet, und der negierte Wahrheitswert des Ergebnisses wird zurückgegeben. Kann der gegebene Ausdruck nicht zu einem Wahrheitswert ausgewertet werden, wird eine Warnung ausgegeben und das Ergebnis gezwungenermaßen als Wahrheitswert interpretiert. Dabei werden die folgenden Regeln angewendet:

1. Die Zahl 0 bedeutet *falsch*
2. Jede Zahl größer 0 bedeutet *wahr*
3. Ein leerer String oder eine leere Aufzählung bedeutet *falsch*
4. Ein leerer String oder eine nichtleere Aufzählung bedeutet *wahr*
5. Ein 'non'-Wert (z. B. none, null, nil, void, ...) bedeutet *falsch*
6. Jeder andere Wert hängt von der jeweiligen Implementierung ab

Ist gar kein Ausdruck angegeben, sollte eine Fehlermeldung generiert werden

- ☞ *path* - der Ausdruck wird als Pfad irgend eines Objekts verstanden. Die Syntax und die Semantik von Pfaden bedarf eines eigenen Kapitels und werden deshalb weiter unten ausführlicher besprochen. Ist kein Ausdruck angegeben, wird der Pfad als 'nothing' interpretiert.
- ☞ *string* - der Ausdruck wird einfach als Text verstanden. Wird kein Ausdruck angegeben, ist der Text leer (empty string). Der Text kann Variablenersetzungen der Form \$name oder \${name} enthalten, wobei 'name' ein Ausdruck des Typs Pfad ist. Der String-Wert des Pfades

Syntax:

```
string_expression ::= ( plain_string | [ varsub ] ) *
varsub             ::= ( '$' Path ) | ( '${' Path '}' )
plain_string       ::= ( '$$' | non_dollar ) *
non_dollar         ::= any character except '$'
```

### Optionale Typenpräfixe

- ☞ *python* - der Ausdruck wird als eingeschränkter Python-Code interpretiert. Der Code muß ein gültiger Python-Ausdruck sein.

### Pfad-Ausdrücke

Ein Pfad-Ausdruck besteht aus einem oder mehreren **Pfaden**, die durch die Pipe ('|') getrennt werden.

Ein **Pfad** besteht aus einem oder mehreren Strings, die durch einen Schrägstrich ( '/') getrennt werden. Dabei muß der erste String ein Variablenname sein, und die restlichen Strings - die **Pfadsegmente** - dürfen Zahlen, Buchstaben und Leerzeichen enthalten, sowie die Zeichen Unterstrich ( '\_' ), Minus ( '-' ), Punkt ( '.' ), Komma ( ',' ) und Tilde ( '~' ).

Die Syntax lautet:

```
PathExpr ::= Path [ '|' Path ] *
Path      ::= variable [ '/' URL_Segment ] *
variable  ::= Name
```

Ein Beispiel:

```
request/cookies/oatmeal
nothing
here/some-file 2001_02.html.tar.gz/foo
root/to/branch | default
```

Wenn ein TALES Pfadausdruck ausgewertet wird, wird versucht, die Pfade von links nach rechts abzuarbeiten bis entweder ein Pfad gültig ist oder keine Pfade mehr vorhanden sind. Um einen Pfad zu durchlaufen, wird erst einmal das Objekt, das an erster Stelle steht, eingelesen. Für jedes Pfadsegment wird dann das dadurch benannte Unterobjekt geladen. Wurde ein Pfad erfolgreich abgearbeitet, ist das gegenwärtig aktuelle Objekt das Ergebnis des Ausdrucks. Ist das Objekt ausführbar - wie eine Funktion oder eine Klasse - so wird es ausgeführt. Die Semantik der Pfadtraversierung und was ausführbar bedeutet ist implementierungsspezifisch.

Misslingt ein Traversierungsschritt, wird die Auswertung sofort beim nächsten Pfad fortgesetzt. Gibt es keinen weiteren Pfad, ist das Ergebnis ein Fehler.

Da jeder Pfad mit einem Variablennamen beginnen muß, benötigt man eine Grundmenge von Objekten, über die andere Objekte lokalisiert und angesprochen werden können. Page Templates definieren die unten aufgelisteten Variablennamen. Da Variablen zuerst lokal, dann global und dann in dieser Liste gesucht werden, verhalten sich die aufgelisteten Variablen wie eingebaute Standardvariablen in Python. Sie sind jederzeit verfügbar, können aber durch die Deklaration globaler oder lokaler Variablen ausgeblendet werden. Die Variablen können jedoch jederzeit angesprochen werden, wenn man dem Namen das Wort 'CONTEXTS' voranstellt, also beispielsweise 'CONTEXTS/root' oder 'CONTEXTS/nothing'.

### Standardvariablen in Page Templates

- ☞ *nothing* - einwertig, repräsentiert einen non-Wert in TAL (wie void, nil, none, ...)

- ✎ *default* - einwertig, bedeutet bei TAL, daß existierender Text nicht ersetzt werden soll.
- ✎ *options* - keyword-Argument, das dem Template übergeben wird
- ✎ *repeat* - die Schleifenvariable (siehe auch Repeat - Variable).
- ✎ *attrs* - ein Dictionary, welches die Attribute des aktuellen Anweisungs-Tags enthält.
- ✎ *CONTEXTS* - die Liste der Standardnamen (diese Liste). Kann verwendet werden, um auf eine durch eine globale oder lokale Variable verdeckte Standardvariable zuzugreifen.

### Optionale Namen in Page Templates

- ✎ *root* - das höchste Objekt des Systems.
- ✎ *here* - das Objekt, auf welches das Templates angewendet wird.
- ✎ *container* - das Container-Objekt des Templates.
- ✎ *template* - die Vorlage selbst.
- ✎ *request* - das Objekte für den anfragenden Request.
- ✎ *user* - der angemeldete Benutzer.
- ✎ *modules* - ein Sammelobjekt, über das alle Python Module und -Pakete angesprochen werden können. Abhängig von den Sicherheitsmechanismen und -einstellungen werden einige oder sogar viele nicht mit TALES verwendet werden können.

### 2.31 TAL Spezifikation Version 1.4

Diese Spezifikation ersetzt die [TAL Spezifikation Version 1.2](http://www.zope.org/Wikis/DevSite/Projects/ZPT/TAL%20Specification%201.2) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/TAL%20Specification%201.2>).

Die *Template Attribute Language* ist eine [Attributsprache](http://www.zope.org/Wikis/DevSite/Projects/ZPT/AttributeLanguage) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/AttributeLanguage>) zur Erstellung dynamischer Templates. Sie erlaubt die Ersetzung, Wiederholung oder Auslassung von Elementen eines Dokuments. Die Anweisungen von TAL sind XML-Attribute im TAL-Namensraum. Diese Attribute können entweder in einem XML-Dokument oder einem HTML-Dokument verwendet werden, damit es sich wie ein Template verhält. Eine **TAL-Anweisung** hat einen Namen (den Attribut-Namen) und einen Rumpf (den Attribut-Wert). Zum Beispiel könnte eine 'content'-Anweisung aussehen wie 'tal:content="string:Hello"'. Das Element, auf dem die Anweisung definiert ist, ist das Anweisungselement. Die meisten **TAL-Anweisungen** benötigen Ausdrücke, aber die Syntax und Semantik dieser Ausdrücke sind nicht Teil von TAL. [Die Seite](http://www.zope.org/Wikis/DevSite/Projects/ZPT/TALES) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/TALES>) ist die empfohlene Quelle hierfür. Der TAL-Namensraum-URI und der Name dafür sind gegenwärtig festgelegt als:

```
xmlns:tal="http://xml.zope.org/namespaces/tal"
```

Dies ist keine URL, lediglich ein eindeutiger Bezeichner. Erwarten Sie nicht, daß irgendein Browser diese Adresse erfolgreich auflösen kann.

### TAL Anweisungen

Unter [EBNF](http://www.zope.org/Wikis/DevSite/Projects/ZPT/EBNF) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/EBNF>) sind die Regeln und Terminale beschrieben. Unter [AttributeLanguage](http://www.zope.org/Wikis/DevSite/Projects/AttributeLanguage) (<http://www.zope.org/Wikis/DevSite/Projects/AttributeLanguage>)

findet sich mehr über Anweisungen in Attributsprachen.

Folgende Befehle sind TAL-Anweisungen: define, attributes, condition, content, replace, repeat, on-error, omit-tag (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/TAL%20Specification%201.4>). Jede [Anweisung](http://www.zope.org/Wikis/DevSite/Projects/ZPT/TAL%20Specification%201.4#oop) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/TAL%20Specification%201.4#oop>) wird unten zusammen mit ihrer Syntax und den Argumenten beschrieben. Obwohl in (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/TAL>) nicht die Syntax von Nichtterminalen in Ausdrücken festgelegt ist - dies bleibt der jeweiligen Implementierung überlassen - ist in [TALES](http://www.zope.org/Wikis/DevSite/Projects/ZPT/TALES) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/TALES>) eine allseits anerkannte Syntax für Ausdrücke in [TAL](http://www.zope.org/Wikis/DevSite/Projects/ZPT/TAL) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/TAL>) beschrieben. Ausdrücke, wie sie in Anweisungen verwendet werden, können jeden Ergebnistyp liefern, obwohl die meisten Anweisungen nur Strings akzeptieren, bzw. andere Werte zu Strings konvertieren. Die Ausdruckssprache muß einen Wert names 'nothing' definieren (siehe [TALES](http://www.zope.org/Wikis/DevSite/Projects/ZPT/TALES)) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/TALES>)), der kein String ist. Im speziellen dient dieser Wert dem Löschen von Elementen und Attributen. Die Auswertung eines Ausdrucks kann den Abbruch der Aktion (der Wirkung der Anweisung) zur Folge haben. Dieser Fall verhält sich genau so, als ob Teile der Anweisung oder die gesamte Anweisung nie existiert hätten.

### Define

Syntax:

```
argument ::= define_scope [';' define_scope]*
define_scope ::= ('local' | 'global') define_var
define_var ::= variable_name expression
variable_name ::= Name
```

*Merke:* Soll in eine Anweisung ein Semikolon eingefügt werden, muß es durch Verdoppelung escaped werden (';').

Es können zwei Arten von TAL-Variablen definiert werden: lokale und globale. Wird in einer Anweisung eine lokale Variable definiert, kann sie nur innerhalb dieses Elements und den darin enthaltenen Elementen verwendet werden. Wird eine Variable in einem inneren Element neu definiert, so wird die alte Variable innerhalb des Geltungsbereichs der neuen Variable durch diese überdeckt. Eine globale Variable kann nach dem Ort ihrer Definition in jedem Element verwendet werden. Wird eine globale Variable irgendwo neu definiert, wird sie im gesamten Template ersetzt.

Ergibt die Auswertung eines Ausdrucks 'nothing', dann hat die Variable den Wert 'nothing', und kann so in weiteren Ausdrücken verwendet werden. Beendet die Auswertung eines Ausdrucks die Ausführung der Anweisung, ist die entsprechende Variable nicht (neu) definiert. Falls die Variable eine andere überdeckt bzw. überdecken sollte, bleibt der Wert trotzdem unverändert, falls nicht, wird sie eben gar nicht erzeugt. Jede Variablendefinition ist unabhängig, also können Variablendefinitionen in Anweisungen stattfinden, in denen andere Variablen aufgelöst werden.

Beispiele:

```
tal:define="mytitle template/title; tlen python:len(mytitle)"
tal:define="global company_name string:Digital Creations, Inc."
```

### Attributes

Syntax:

```
argument ::= attribute_statement
           [';' attribute_statement]*
attribute_statement ::= attribute_name expression
attribute_name ::= [namespace ':' ] Name
namespace ::= Name
```

*Merke: Soll in eine Anweisung ein Semikolon eingefügt werden, muß es durch Verdoppelung escaped werden (';').*

Soll ein Attributwert durch -oder ein neues Attribut mit einem dynamischen Wert ersetzt bzw. erzeugt werden, verwendet man die 'attributes'-Anweisung. Ein Attributnamen kann durch ein Namensraum-Präfix ausgezeichnet werden -- zum Beispiel 'html:table' - wenn in einem XML-Dokument mehrere Namensräume existieren. Der Wert von jedem Ausdruck wird - falls notwendig - in einen String konvertiert.

Wird der zugewiesene Wert zu 'nothing' ausgewertet, so wird das Attribut aus dem Element entfernt. Bricht der Ausdruck die Anweisung ab, so bleibt das Attribut unverändert. Jede Attribut-Zuweisung ist unabhängig, d. h. in einer Anweisung können manchen Attributen Werte zugewiesen werden, während andere gelöscht oder unverändert bleiben.

Beispiele:

```
<a href="/sample/link.html"
  tal:attributes="href here/sub/absolute_url">
<textarea rows="80" cols="20"
  tal:attributes="rows request/rows;cols request/cols">
```

Wird obige Anweisung in einem Element mit einer aktiven 'replace'-Anweisung benutzt, kann je nach Implementierung die 'attributes'-Anweisung ignoriert werden. Falls nicht, muß die Ersetzung den 'structure'-Type verwenden. Die zurückgegebene Struktur muß mindestens ein Element enthalten, und die Ersetzung wird auf dem ersten entsprechenden Element ausgeführt. Zum Beispiel ist für die erste Zeile unten jede der beiden folgenden Lösungen akzeptabel:

```
<span tal:replace="structure an_image"
  tal:attributes="border string:1">


```

Wird es in einem Element mit einer 'repeat'-Anweisung verwendet, wird die Ersetzung auf jedem wiederholten Element ausgeführt, und die Ersetzungs-Anweisung wird für jede Wiederholung neu ausgewertet.

## Condition

Syntax:

```
argument ::= expression
```

Um einen bestimmten Teil eines Templates nur unter bestimmten Bedingungen einzuschließen, und es anderenfalls auszulassen, wird die 'condition'-Anweisung verwendet. Ist das Ergebnis der Auswertung 'true', wird ganz normal mit der Verarbeitung fortgefahren. Ist das Ergebnis 'unwahr', wird das Anweisungselement sofort aus dem Dokument entfernt. Die Bestimmung des Wahrheitswerts ist die Angelegenheit der Schnittstelle zwischen TAL und der 'expression engine'. Zu diesem Zweck hat 'nothing' den Wert 'false', und die Beendigung einer Aktion hat denselben Effekt wie die Rückgabe des Wertes 'true'.

Beispiel:

```
<p tal:condition="here/copyright"
  tal:content="here/copyright">(c) 2000</p>
```

## Replace

Syntax:

```
argument ::= ([ 'text' ] | 'structure' ) expression
```

Um ein Element durch dynamischen Inhalt zu ersetzen wird die 'replace'-Anweisung verwendet. Diese ersetzt die Anweisung entweder durch Text oder durch eine Struktur (structu-

re, unescaped HTML Markup). Der Rumpf der Anweisung ist ein Ausdruck mit einem optionalen Typenpräfix. Der Wert des Ausdrucks wird in einen String konvertiert, der escaped ist, falls das Typenpräfix 'text' oder gar kein Typenpräfix angegeben ist. Der Wert wird unverändert eingefügt, falls das 'structure'-Präfix verwendet wird. Escapen bedeutet die Ersetzung von '&' zu '&amp;', '<' zu '&lt;' und '>' zu '&gt;'. Ist der Wert 'nothing', wird das Element einfach entfernt. Wird die Aktion abgebrochen, bleibt das Element unverändert (siehe auch TALEs default value).

*Merke: Das Standardersetzungsverhalten ist 'text'.*

Beispiele:

```
<span tal:replace="template/title">Title</span>
<span tal:replace="text template/title">Title</span>
<span tal:replace="structure table" />
<span tal:replace="nothing">Dieses Element ist ein
  Kommentar.</span>
```

## Content

Syntax:

```
argument ::= ([ 'text' ] | 'structure' ) expression
```

Anstatt ein ganzes Element zu entfernen, kann mit der 'content'-Anweisung auch Text oder HTML Markup anstelle des Kindknotens eingefügt werden. Die Argumentsyntax ist genau wie bei 'replace', und wird auch genau gleich interpretiert. Ist der Wert des Ausdrucks 'nothing', bleibt das Ausdruckselement kinderlos. Wird die Ausführung abgebrochen, bleibt der Elementinhalt unverändert.

*Merke: Das Standardersetzungsverhalten ist 'text'.*

Beispiel:

```
<p tal:content="user/name">Fred Farkas</p>
```

## Repeat

Syntax:

```
argument ::= variable_name expression
variable_name ::= Name
```

Soll für jeden Eintrag einer Liste ein Unterbaum des Dokuments repliziert werden, verwenden wir 'repeat'. Der in 'repeat' verwendete Ausdruck sollte als Wert eine Liste haben. Ist die Liste leer, wird das Anweisungselement entfernt, andernfalls wird es für jeden Wert in der Liste wiederholt. Wird die Ausführung abgebrochen, bleibt das Element unverändert und es werden keine neuen Variablen deklariert.

Die Variable 'variable\_name' wird dazu verwendet, eine lokale Variable und eine Schleifenvariable zu deklarieren. Für jeden Schleifendurchlauf wird die lokale Variable auf das aktuelle Listenelement gesetzt, die Schleifenvariable ist ein Iterationsobjekt. Iterationsobjekte werden dazu verwendet, Zugriff auf Informationen über den gegenwärtigen Schleifendurchlauf zu haben (so wie den aktuellen Index). Iterationsobjekte werden genauer in (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/TALES>) beschrieben. Die Schleifenvariable hat genau den selben Namen wie die lokale Variable, aber auf sie kann nur über die Standardvariable 'repeat' zugegriffen werden (näheres unter [RepeatVariable](http://www.zope.org/Wikis/DevSite/Projects/ZPT/RepeatVariable) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/RepeatVariable>)).

Beispiele:

```
<p tal:repeat="txt python:'one', 'two', 'three'">
  <span tal:replace="txt" />
</p>
<table>
  <tr tal:repeat="item here/cart">
```

```

<td tal:content="repeat/item/index">1</td>
<td tal:content="item/description">Ein Ding</td>
<td tal:content="item/price">EUR 1.50</td>
</tr>
</table>

```

## On-Error

Syntax:

```
argument ::= ([ 'text' ] | 'structure') expression
```

Mit der Anweisung 'on-error' kann eine Fehlerbehandlung in das Dokument eingebaut werden. Wenn eine TAL-Anweisung einen Fehler hervorruft, versucht der TAL-Interpreter zuerst im selben Element eine 'on-error'-Anweisung zu finden, dann im einschließenden Element, und so weiter. Die erste 'on-error'-Anweisung die gefunden wird, wird ausgeführt. Die 'on-error'-Anweisung verhält sich genau wie die 'content'-Anweisung.

Die einfachste 'on-error'-Anweisung hat als Ausdruck einen String oder 'nothing'. Eine etwas komplexere Fehlerbehandlung ruft etwa ein Skript auf, das den Fehler genauer untersucht und dann entweder eine Fehlermeldung ausgibt oder einen Ausnahmefehler (exception) erzeugt um den Fehler nach außen weiterzuleiten. Weitere Informationen sind zu finden unter [RenderErrorHandlingStrategies](http://www.zope.org/Wikis/DevSite/Projects/ZPT/RenderErrorHandlingStrategies) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/RenderErrorHandlingStrategies>).

Beispiele:

```

<p tal:on-error="string: Fehler! Dieser Paragraph ist fehlerhaft!"> Mein Name ist
<span tal:replace="here/SlimShady" />.<br/>
(Mein Login ist <b tal:on-error="string: Benutzername ist nicht definiert!"
tal:content="user">unbekannt</b>)
</p>

```

Im oberen Beispiel wird 'here/SlimShady' einen Fehler auslösen, die 'on-error'-Anweisung fängt diesen ab und ersetzt den ganzen Paragraphen mit dem Text 'Fehler! Dieser Paragraph ist fehlerhaft!'. Kann 'here/SlimShady' korrekt ausgewertet werden, aber bei der Auswertung von 'user' tritt ein Fehler auf, dann ersetzt 'Benutzername ist nicht definiert!' das Wort 'unbekannt', aber der Rest des Paragraph wird völlig normal behandelt.

## Omit-Tag

Syntax:

```
argument ::= [expression]
```

Um den Inhalt eines Tags unverändert zu lassen aber die einschließenden Tags selbst zu entfernen wird die 'omit-tag'-Anweisung verwendet. Wird der zugehörige Ausdruck zu 'false' ausgewertet, wird das Tag normal abgearbeitet. Ist der Wert des Ausdrucks 'true' ,oder ist gar kein Ausdruck vorhanden, wird das Anweisungselement einfach durch seinen Inhalt ersetzt. Die Bestimmung des Wahrheitswerts ist die Angelegenheit der Schnittstelle zwischen TAL und der 'expression engine'. Zu diesem Zweck hat 'nothing' den Wert 'false', und die Beendigung einer Aktion hat denselben Effekt wie die Rückgabe des Wertes 'false'.

Beispiele:

```

<div tal:omit-tag="" comment="Dieses Tag wird entfernt">
<i>...aber dieser Text wird bleiben</i>
</div>

<b tal:omit-tag="not:bold">Ich darf nicht fettgedruckt sein.</b>

```

## Ausführungsreihenfolge

Bei nur einer TAL-Anweisung pro Element ist die Ausführungsreihenfolge einfach. Beginnend mit dem Wurzelement werden die Anweisungen jedes Elements ausgeführt und dann die Kind-Elemente der Reihe nach besucht bzw. abgearbeitet. Jede beliebige Kombination von Anweisungen kann in einem Element vorkommen, mit der Ausnahmen, daß 'replace' und 'content' nicht im selben Element stehen dürfen. Enthält ein Element mehrere Anweisungen, so werden sie entsprechend der folgenden Prioritätenliste abgearbeitet:

- define
- condition
- repeat
- content or replace
- attributes
- omit-tag

Da die 'on-error'-Anweisung nur dann ausgeführt wird wenn ein Fehler auftritt, wird sie in der Liste nicht geführt.

Der Sinn hinter dieser Prioritätenverteilung ist folgender: Die Deklaration von Variablen kann auch andere Anweisungen betreffen, deshalb kommt 'define' zuerst. Dann gilt es zu entscheiden, ob das aktuelle Element überhaupt eingefügt werden soll, also muß jetzt 'condition' folgen. Da 'condition' bereits von Variablen abhängen kann, die mit 'define' deklariert wurden, muß es danach kommen. Es ist sehr nützlich, verschiedene Teile eines Elements bei jeder Wiederholung einer Schleife mit unterschiedlichen Daten zu ersetzen, deshalb kommt jetzt 'repeat'. Es macht wenig Sinn, die Attribute eines Elements zu ersetzen und sie dann zu löschen, also kommt 'attributes' zum Schluß. Die übrigen Anweisungen stehen im Konflikt zueinander, weil sie entweder das Anweisungselement verändern oder ersetzen.

Soll die Ausführungsreihenfolge verändert werden, müssen die Anweisungen in andere Elemente gepackt werden, etwa <div> oder <span>.

Beispiel:

```

<p tal:define="x /a/long/path/from/the/root"
tal:condition="x"
tal:content="x/txt"
tal:attributes="class x/class">Ex Text</p>

```

## 2.4 Häufig gestellte Fragen zu Zope Page Templates

### Q1: Was sind Page Templates (ZPT)?

Bei [VisionStatement](http://www.zope.org/Wikis/DevSite/Projects/ZPT/VisionStatement) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/VisionStatement>) findet sich ein Überblick.

### Q2: Ersetzen Page Templates DTML?

ZPT ist eine Alternative zu DTML und kann einen großen Teil wenn nicht sogar den kompletten Gebrauch von DTML für eine bestimmte Anwendergruppe ersetzen. Sollten Sie [PresentationDesigner](http://www.zope.org/Wikis/DevSite/Projects/ZPT/PresentationDesigner) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/PresentationDesigner>) sein, werden Sie ZPT nützlicher als DTML empfinden.

### Q3: Ist DTML überholt?

Wir ermutigen Entwickler bei neuen Projekten zu prüfen, ob ZPT in Frage kommt. Hier bei DC werden wir ZPT verwenden, wo immer es in Frage kommt. Aber wie auch immer, wenn DTML besser paßt, ....

### Q4: Kann ich ZPT verwenden, ohne mich mit XML auszukennen?

Ja, man kann ZPT ohne jede XML-Vorkenntnisse verwenden. Die [Attribut-Sprache](http://www.zope.org/Wikis/DevSite/Projects/ZPT/AttributeLanguage) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/AttributeLanguage>) von ZPT wurde so entworfen, daß sie mit der XML-Grammatik kompatibel ist, aber das bedeutet nur, daß sie einfach und gut definiert ist.

### Q4a: Warum also überhaupt XML?

Wenn sich das Web in Richtung XHTML entwickelt (vorausgesetzt, Sie glauben daran), dann wird XML das allgemein zugrunde liegenden Markup-Format sein. Es bietet eine hohe Flexibilität durch die Verwendung von Namensräumen, so daß wir innerhalb der Strukturen von XML alles erledigen können, was wir mit [DTML](http://www.zope.org/Wikis/DevSite/Projects/ZPT/FAQ/edit-form?page=DTMLish) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/FAQ/edit-form?page=DTMLish>) tun wollten. Außerdem erwarten wir, daß es mehr HTML-Editoren geben wird die XML unterstützen, als solche, die DTML unterstützen.

### Q4b: Warum ist man auf das Arbeiten mit Attribute eingeschränkt?

Eigene Elemente definieren wäre sicher eine Möglichkeit. Unglücklicherweise erwarten wir, daß die meisten HTML-Editoren eher weiterhin mit reinem HTML arbeiten, als beliebiges XML zu unterstützen. Die Definition einer eigenen Elementmenge (durch eine DTD oder ein Schema) erfordert eine Menge Verarbeitungsaufwand seitens des Editors, um irgendwas nützliches mit der Markupsprache anzufangen. Ein Teil der Kernidee von ZPT ist, den PresentationDesignern zu erlauben, sich eher auf die Darstellung (also HTML) zu konzentrieren als auf neudefinierte Elemente, die sich im Editor nicht vernünftig visualisieren lassen.

### Q5: Benötigt ZPT XHTML?

ZPT arbeitet mit gewöhnlichem HTML. Das ZPT-Markup besteht in den Attributen der HTML-Tags. Stört sich der verwendete HTML-Editor nicht an unbekanntem Attributen, kommt er auch mit Page Templates zurecht.

### Q6: Was ist XHTML (kurze Antwort, bitte)?

Informell gesehen ist XHTML einfach HTML mit einer strengeren Syntax und Semantik. Ein paar Regeln sind zum Beispiel, daß alle Tags geschlossen werden müssen, oder daß jedes Attribut der Syntax „attributName=„wert immer in Anführungszeichen“ folgen muß. Außerdem dürfen bestimmte Elemente bestimmte andere Elemente nicht als Kinder enthalten. Das ist es im Großen und Ganzen (wir entschuldigen uns bei allen XML-Verfechtern).

### Q7: Mein bevorzugter Editor kann nur einfaches HTML. Kann ich damit ZPT verwenden?

Ja, da gibt es mehrere Möglichkeiten. Die beste ist, einen Editor zu verwenden der XML unterstützt (was einfach bedeutet, daß es keine Probleme mit eigenen Elementen oder Namensräumen gibt). Wenn Ihr Editor mit Code wie dem untenstehenden klarkommt, dann können Sie wahrscheinlich ZPT verwenden:

```
<span tal:define="x str:1" />
```

Sie können jedoch auch einfach ein Werkzeug wie HTML Tidy verwenden, um Ihr HTML in XHTML zu konvertieren (was von ZPT bedingungslos verstanden wird). Der einzige Seiteneffekt ist, daß die Ausgabe nicht mehr Ihrem originalen HTML-Code entspricht! Dies ist keine optimale Lösung.

### Q8: Welche HTML-Editoren arbeiten gut mit ZPT?

Wir sind noch dabei, eine Liste zusammenzustellen, aber auf kommerzieller Seite arbeiten sowohl [GoLive](http://www.zope.org/Wikis/DevSite/Projects/ZPT/GoLive) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/GoLive>) als auch Dreamweaver 4 gut mit ZPT. Frühere Versionen von Amaya arbeiten im HTML-Modus ebenfalls gut.

### Q9: Wie verwendet man einen HTML-Editor am besten mit ZPT?

Ein Ziel von ZPT ist die nahtlose Integration in Ihren bisherigen Estellungsprozeß. Im Moment bedeutet dies entweder die Verwendung des HTML-Editor-Formulars (Igit!), oder die Verwendung eines HTML-Editors der mit [WebDAV](http://www.zope.org/Wikis/DevSite/Projects/ZPT/WebDAV) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/WebDAV>) oder FTP arbeiten kann. Da ZPT erwartet, daß die zu bearbeitenden Quellen aus Zope ausgelesen werden, ist dies der natürlichste Weg zu arbeiten: Zope enthält die anerkannten Quellen.

### Q10: Warum spricht man bei ZPT über TAL und METAL als nicht Zope-spezifisch?

Wir dachten diese Ideen sind so großartig, daß wir sie nicht für uns behalten wollten. Natürlich glauben wir, daß wir die absolut beste Implementierung der Spezifikationen haben, aber wir wollten auch sicherstellen, daß die Spezifikationen so allgemein sind, daß jeder, der diese Idee stiehlt (jede gute Idee sollte gestohlen werden!), nicht eine total abweichende Implementierung hat. Wir brauchen Annäherung, nicht Auseinanderentwicklung.

**Q11: Bedeutet dies, daß TAL und METAL einem Gremium wie dem W3C zur Standardisierung vorgeschlagen werden soll?**

Im Augenblick gibt es keine derartigen Bestrebungen. Im Gegensatz zu vielen anderen Standardvorschlägen wollten wir zuerst eine funktionierende Implementierung haben.

**Q12: Wie führe ich die Tests aus? Funktionieren die Beispiele alle?**

Die Datei `.TAL README.txt` beschreibt, wie man die Testumgebung für TAL laufen läßt. Haben Sie `.expat` nicht installiert, werden die XML-Beispiele fehlschlagen. Die ZPT-Tests können durch das Ausführen von `.run.py` im Testverzeichnis durchgeführt werden.

**Q13: Funktioniert Acquisition mit den Pfadausdrücken aus TALES?**

Pfadausdrücke wie zum Beispiel `"here/master_page/macros/copyright"` müssen immer mit einer Standardvariablen anfangen (in diesem Fall `"here"`). Jeder Abschnitt des Pfads zieht Vorteile aus der Acquisition, wenn das angesprochene Objekt dies unterstützt; `"master_page"` könnte von `"here"` acquiriert worden sein. Hier liegt ein Unterschied zu DTML, wo jeder Name in einer großen, mannigfaltigen Menge von Orten gesucht wird, von denen viele Acquisition unterstützen.

**Q14: Wie schreibe ich Kommentare, die in der Darstellung der Seite nicht angezeigt werden?**

Verwenden Sie `tal:replace="nothing"` or `tal:condition="nothing"`, wie folgt:

```
<span tal:replace="nothing">A Comment</span>
<span tal:replace="nothing"><-- A Commented
  Comment --></span>
```

**Q15: Wenn ich tal:attributes verwende um das src- oder das href-Attribut zu setzen, werden meine Kaufmannsunds ('&') in &amp;s verwandelt! Wie kann ich das Verhindern?**

Das können und sollen Sie nicht! Dem Standard entsprechend ist escapen absolut korrekt. Es funktioniert auch mit jedem Browser, den wir ausprobiert haben.

### 3.1 METAL für Anfänger

Von [tone](http://www.zope.org/Members/tone) [http://www.zope.org/Members/tone].  
Letzte Aktualisierung am 01.08.2001.

### 3.2 Eine Basis-Anleitung für METAL

Dieser Artikel ist Teil eines Versuchs, etwas an die ZPT-Gemeinde zurückzugeben. Ich bin sicherlich kein ausgesprochener ZPT-Experte, sollten sich also irgendwo Fehler eingeschlichen haben, so teilt mir das einfach mit. Ich werde es sofort verbessern.

Dies ist keine Einführung in Page Templates - es gibt genügend andere Quellen, die das abdecken.

#### 3.2.1 Was ist METAL?

METAL ist eine in Zope integrierte Makro-Sprache. Evan Simpson - die Lichtgestalt in der Zope Gemeinschaft - beschreibt den Nutzen der Makros folgendermaßen:

*Makros sind dazu da, Präsentationseinheiten zu teilen, und die gemeinsam genutzten Darstellungen in Page Templates eingebettet darzustellen. Sie werden nicht dazu verwendet, um Daten einzubauen; dafür sind ,tal:replace' und ,tal:content' da. - Evan Simpson*

#### 3.2.2 Mit METAL beginnen

Zuerst muß ein Page Template Dokument mit dem Name ,mymacro' erzeugt werden. Das Standarddokument sieht ungefähr folgendermaßen aus:

```
<html>
<head>
<title tal:content="template/title">The title</title>
</head>
<body>

<h2><span tal:replace="here/title_or_id">Titel oder Id
des Inhalts</span>
<span tal:condition="template/title"
tal:replace="template/title">optionale Template
Id</span></h2>

Die ist ein Page Template <em tal:content="template/id">
Template Id</em>.
</body>
</html>
```

Nun verändern wir das <html>-Tag wie folgt:

```
<html xmlns:tal="http://xml.zope.org/namespaces/tal"
xmlns:metal="http://xml.zope.org/namespaces/metal"
metal:define-macro="master">
```

Am Anfang mögen die ganzen XML-Namesräume etwas verwirrend erscheinen. Eigentlich können sie auch weggelassen werden und man schreibt:

```
<html
metal:define-macro="master">
```

Jetzt fangen wir an, Makros zu benutzen. Wir werden eine Navigationsleiste und eine Karteikartenansicht hinzufügen. Das Page Template wird geändert zu:

```
<html metal:define-macro="master">
<head>
<title tal:content="template/title">Der Titel</title>
</head>
<body>
<table border="1">
<tr><td>Ich bin eine Navigationsleiste - fügen Sie hier
Links hinzu</td></tr>
<tr><td>
<h2><span tal:replace="here/title_or_id">Titel oder Id
des Inhalts</span>
<span tal:condition="template/title"
tal:replace="template/title">optionale Template Id
</span></h2>
<div metal:define-slot="main">
Verwenden Sie innerhalb dieses Makros ein 'fill-
slot="main"-Attribut in einem Tag, wird das Tag den
Text ersetzen. Dies ist der Haupttrumpf. Er enthält alles.
</div>
</td>
</tr>
</table>
</body>
</html>
```

Über die Option ,test' kann man sich das Ergebnis ansehen. Um dieses Makro effektiv einzusetzen, muß es von einem anderen Page Template aus aufgerufen werden. Ein Page Template ,use\_mymacro' würde ungefähr so aussehen:

```
<html metal:use-macro="here/mymacro/macros/master">
<head>
<title tal:content="template/title">Der Titel</title>
</head>
<body>
<p>Ich weiß nichts</p>
</body>
</html>
```

**Ein wichtiger Tipp:** Die ,Expand Macros when Editing'-Option sollte nicht gewählt werden. Sonst werden alle Makros ausgedehnt und die klare Dokumentstruktur geht verloren. Vielleicht habe ich auch missverstanden, wie ,Expand Macros ...' arbeitet.

Wenn Sie das ,use\_mymacro' Page Template testen, werden Sie eine große Überraschung erleben: der Text "Ich weiß nichts" wird nicht angezeigt.

Wie auch immer, die Seite gibt einen das Verständnis dafür, was Sie als nächstes zu tun haben: *Wenn Sie bei der Bearbeitung des Makros ein Tag mit einem ,fill-slot="main"-Attribut eingeben, wird das Tag den Text ersetzen. Dies ist der ,main'-Rumpf. Er enthält alles.* In Ordnung, bearbeiten Sie das Makro so, daß es folgendermaßen aussieht:

```
<html metal:use-macro="here/mymacro/macros/master">
<head>
<title tal:content="template/title">Der Titel</title>
</head>
<body>
<p metal:fill-slot="main">Ich weiß nichts</p>
</body>
</html>
```

Zeit zu testen - viel Erfolg! Anstelle des statischen Contents im <p>-Tag können auch die Anweisungen, 'tal:content' oder 'tal:replace' verwendet werden.

Ein paar Worte über Slots. Evan Simpson hat es wie folgt formuliert:

*Man sollte Makros nur verwenden, wenn man die Möglichkeit haben will, Teile davon in Page Templates zu überschreiben, die das Makro verwenden. Um das zu tun,*

gibt man mit der Anweisung `.metal:define-slot` in der Makrodefinition einem Tag ein Slot. Bei der Verwendung des Makros wird der Inhalt des Slots ganz normal mit dem Rest des Makros verwendet, solange, bis eine `'metal:fill-slot'`-Anweisung unter dem Namen des Slots hinzugefügt wird. Die `'metal:fill-slot'`-Anweisung ersetzt das Slot-Tag aus der Makrodefinition vollständig, und zwar samt dem Namen des Tags und allen Attributen.

### 3.2.3 Wofür Makros verwendet werden können

Makros können verwendet werden, um Schnittstellenspezifische Information zu verstecken. Außerdem kann durch Änderungen des Attributs `.metal:use-macro="here/mymacro/macros/master"` im `<html>`-Tag völlig voneinander verschiedene Layouts erzeugt werden, z. B. können für Sehbehinderte total andere Ausgaben erstellt werden.

### 3.2.4 Links und andere Quellen

Auf der ZPT-Seite sind viele Informationen verfügbar; hier sind diejenigen aufgelistet, die ich am häufigsten verwende.

- 🔗 [The ZPT site](http://dev.zope.org/Wikis/DevSite/Projects/ZPT)  
(<http://dev.zope.org/Wikis/DevSite/Projects/ZPT>)
- 🔗 [Recent changes to the ZPT site](http://dev.zope.org/Wikis/DevSite/Projects/ZPT/RecentChanges)  
(<http://dev.zope.org/Wikis/DevSite/Projects/ZPT/RecentChanges>)
- 🔗 [Evan Simpsons' ZPT tutorial \(4 parts\)](http://dev.zope.org/Wikis/DevSite/Projects/ZPT/SimpleTutorial)  
(<http://dev.zope.org/Wikis/DevSite/Projects/ZPT/SimpleTutorial>)
- 🔗 [Peter Bengtssons' DTML -> ZPT page](http://www.zope.org/Members/peterbe/DTML2ZPT)  
(<http://www.zope.org/Members/peterbe/DTML2ZPT>)

### 3.2.5 METAL Spezifikation Version 1.0

METAL (Macro Expansion Template Attribute Language) ist eine [Attributsprache](http://www.zope.org/Wikis/DevSite/Projects/ZPT/AttributeLanguage) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/AttributeLanguage>) für das strukturierte Preprocessing von Makros. METAL kann gemeinsam mit oder unabhängig von [TAL](http://www.zope.org/Wikis/DevSite/Projects/ZPT/TAL) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/TAL>), [TALES](http://www.zope.org/Wikis/DevSite/Projects/ZPT/TALES) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/TALES>) und [ZPT](http://www.zope.org/Wikis/DevSite/Projects/ZPT) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT>) verwendet werden. Makros bieten die Möglichkeit, ein Stückchen Präsentation in einem Template zu definieren und mit anderen zu teilen, so daß die Änderung gleichzeitig an allen Stellen stattfindet, die sich das Makro teilen. Makros werden immer komplett ausgeschrieben, auch im Quelltext eines Templates, so daß das Template seiner tatsächlichen Darstellung immer sehr nahe kommt.

Der METAL-Namensraum-URI und der symbolische Name davon sind gegenwärtig definiert als:

```
xmlns:metal="http://xml.zope.org/namespaces/metal"
```

### METAL Anweisungen

Die Terminale und Regeln sind in [EBNF](http://www.zope.org/Wikis/DevSite/Projects/ZPT/EBNF) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/EBNF>) aufgeführt. Eine Beschreibung für Anweisungen in der Atribut-Sprache findet sich in [AttributeLanguage](http://www.zope.org/Wikis/DevSite/Projects/ZPT/AttributeLanguage) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/AttributeLanguage>)

Die folgenden TAL 1.0 Anweisungen werden benötigt: `define-macro`, `use-macro`, `define-slot`, `use-slot` (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/METAL%20Specification%201.0>).

Jede Anweisung wird unten zusammen mit seiner Syntax noch genauer beschrieben. Obwohl METAL nicht die Syntax von `'expression'`-Nichtterminalen definiert - dies ist Implementierungsspezifika - wird eine kanonische `'expression'`-Syntax für METAL-Argumente in [TALES Specification 1.0](http://www.zope.org/Wikis/DevSite/Projects/ZPT/TALES%20Specification%201.0) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/TALES%20Specification%201.0>) beschrieben.

### Define-Macro

Syntax:

```
argument ::= Name
```

Um ein Makro zu definieren, muß zuerst ein Name für das Makro festgelegt und dann das `.define-macro`-Attribut zum Dokumentelement hinzugefügt werden - mit dem Name als Argument. Der Unterbaum des Elements ist der Rumpf des Makros. Ein Beispiel:

```
<p metal:define-macro="copyright">
  Copyright 2001, <em>Foo</em> Inc.
</p>
```

### Use-Macro

Syntax:

```
argument ::= expression
```

Um ein Makro anzuwenden, muß zuerst das Dokumentenelement ausgewählt werden, das ersetzt werden soll. Man fügt ein `.use-macro`-Attribut hinzu, und setzt als Wert des Attributs einen Ausdruck, der die Makrodefinition zurückliefert. Für gewöhnlich ist der Ausdruck ein Pfad oder ein Template Id und ein Makroname. Es ist wichtig zu verstehen, daß dieser Ausdruck unabhängig von jeglichem TAL-Kommando ausgewertet wird, um somit auch nicht davon abhängen kann.

*Bemerkung:* [Page Templates](http://www.zope.org/Wikis/DevSite/Projects/ZPT/PageTemplates) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/PageTemplates>) verwenden [TALES](http://www.zope.org/Wikis/DevSite/Projects/ZPT/TALES) (<http://www.zope.org/Wikis/DevSite/Projects/ZPT/TALES>) für diesen Ausdruck, und es können alle Standardnamen des Kontexts verwendet werden. Da die Sammlung von Makrodefinitionen nach außen über das Attribut `.macros` verfügbar gemacht werden, kann auf einzelne Makros auch einfach zugegriffen werden.

Ein Beispiel:

```
<hr />
<p metal:use-macro="here/master_page/macros/
  copyright">
</hr />
```

### Makro-Entfaltung

Der Effekt einer Makro-Entfaltung ist das Einpflanzen eines Unterbaums eines anderen Dokuments oder von irgendwo aus dem aktuellen Dokument an die Stelle, wo das Anweisungselement steht. Dabei wird der bestehende Unterbaum ersetzt. Teile davon können als Unterbaum des neuen bestehen bleiben, wenn das Makro Slots enthält. Werden im Rumpf eines Makros weitere Makros verwendet, so werden diese zuerst entfaltet.

Wenn ein Makro entfaltet wird, wird das `.define-macro`-Attribut durch das `.use-macro`-Attribut aus dem Anweisungselement ersetzt. Dadurch wird die Wurzel des entfalteten Makros ein gültiges `.use-macro`-Anweisungselement.

## Define-Slot und Fill-Slot

Syntax:

```
argument ::= Name
```

Makros sind noch viel nützlicher, wenn man Teile von ihnen während der Benutzung überschreiben kann. Beispielsweise könnte man eine komplizierte Tabelle mit jeweils verschiedenem Inhalt wiederverwenden wollen. Nun wäre es möglich, die Tabellendaten irgendwo im Template zu speichern, eine Variable dafür zu deklarieren, und die Variable dann irgendwo im Rumpf des Makros zu verwenden. Damit wird allerdings die Präsentationsstruktur verletzt; die gewünschten Zelleninhalte würden in der Tabelle nicht gesehen werden können.

Um Elemente im Rumpf des Makros überschreibbar zu machen, müssen an der jeweiligen Stelle ‚define-slot‘-Attribute eingefügt und als Wert einen Slotnamen zugewiesen bekommen. Egal wo das Makro verwendet wird, man wählt einfach die entsprechenden Unterelemente des Anweisungselements aus, und fügt ein ‚define-slot‘-Attribut hinzu und setzt es auf den Namen eines Slots. Wenn das Makro entfaltet wird, ersetzen die ‚fill-slot‘-Elemente die ‚define-slot‘-Elemente im Rumpf derjenigen Makros, die den selben Slotnamen verwenden.

Slotnamen müssen innerhalb eines Makros und innerhalb einer Makroanwendung eindeutig sein. Es ist jedoch erlaubt, in einem Makro einen Slot zu definieren aber nicht zu füllen. Dies hat dann einfach zur Folge, daß die Standard-Inhalte der Slotdefinition in das Makro kopiert werden. Bezieht sich eine ‚fill-slot‘-Anweisung auf einen Slot der im Rumpf des Makros nicht gefunden werden kann, verursacht dies einen Fehler.

Es folgt ein Beispiel:

Erstes Dokument:

```
<table metal:define-macro="sidebar">
  <tr><th>Links</th></tr>
  <tr><td metal:define-slot="links">
    <a href="/">Ein Link</a>
  </td></tr>
</table>
```

Zweites Dokument:

```
<table metal:use-macro="here/doc1/macros/sidebar">
  <tr><th>Links</th></tr>
  <tr><td metal:fill-slot="links">
    <a href="http://www.goodplace.com">Guter Platz
  </a><br>
    <a href="http://www.badplace.com">Schlechter Platz
  </a><br>
    <a href="http://www.otherplace.com">Anderer Platz
  </a>
  </td></tr>
</table>
```

Man beachte, daß im zweiten Dokument, welches das Makro aus dem ersten Dokument verwendet, den ganzen Code des Sidebar-Makros enthält außer den Slot namens ‚links‘. Das kommt daher, daß das Makro jedesmal eingefügt wird, wenn das zweite Dokument editiert wird.

## Open Publication License

Draft v1.0, 8 June 1999

### I. REQUIREMENTS ON BOTH UNMODIFIED AND MODIFIED VERSIONS

The Open Publication works may be reproduced and distributed in whole or in part, in any medium physical or electronic, provided that the terms of this license are adhered to, and that this license or an incorporation of it by reference (with any options elected by the author(s) and/or publisher) is displayed in the reproduction.

Proper form for an incorporation by reference is as follows:

Copyright (c) <year> by <author's name or designee>. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, vX.Y or later (the latest version is presently available at <http://www.opencontent.org/openpub/>).

The reference must be immediately followed with any options elected by the author(s) and/or publisher of the document (see section VI).

Commercial redistribution of Open Publication-licensed material is permitted.

Any publication in standard (paper) book form shall require the citation of the original publisher and author. The publisher and author's names shall appear on all outer surfaces of the book. On all outer surfaces of the book the original publisher's name shall be as large as the title of the work and cited as possessive with respect to the title.

### II. COPYRIGHT

The copyright to each Open Publication is owned by its author(s) or designee.

### III. SCOPE OF LICENSE

The following license terms apply to all Open Publication works, unless otherwise explicitly stated in the document.

Mere aggregation of Open Publication works or a portion of an Open Publication work with other works or programs on the same media shall not cause this license to apply to those other works. The aggregate work shall contain a notice specifying the inclusion of the Open Publication material and appropriate copyright notice.

SEVERABILITY. If any part of this license is found to be unenforceable in any jurisdiction, the remaining portions of the license remain in force.

NO WARRANTY. Open Publication works are licensed and provided "as is" without warranty of any kind, express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose or a warranty of non-infringement.

### IV. REQUIREMENTS ON MODIFIED WORKS

All modified versions of documents covered by this license, including translations, anthologies, compilations and partial documents, must meet the following requirements:

1. The modified version must be labeled as such.
2. The person making the modifications must be identified and the modifications dated.
3. Acknowledgement of the original author and publisher if applicable must be retained according to normal academic citation practices.
4. The location of the original unmodified document must be identified.
5. The original author's (or authors') name(s) may not be used to assert or imply endorsement of the resulting document without the original author's (or authors') permission.

## V. GOOD-PRACTICE RECOMMENDATIONS

In addition to the requirements of this license, it is requested from and strongly recommended of redistributors that:

1. If you are distributing Open Publication works on hardcopy or CD-ROM, you provide email notification to the authors of your intent to redistribute at least thirty days before your manuscript or media freeze, to give the authors time to provide updated documents. This notification should describe modifications, if any, made to the document.
2. All substantive modifications (including deletions) be either clearly marked up in the document or else described in an attachment to the document.
3. Finally, while it is not mandatory under this license, it is considered good form to offer a free copy of any hardcopy and CD-ROM expression of an Open Publication-licensed work to its author(s).

## VI. LICENSE OPTIONS

The author(s) and/or publisher of an Open Publication-licensed document may elect certain options by appending language to the reference to or copy of the license. These options are considered part of the license instance and must be included with the license (or its incorporation by reference) in derived works.

A. To prohibit distribution of substantively modified versions without the explicit permission of the author(s). "Substantive modification" is defined as a change to the semantic content of the document, and excludes mere changes in format or typographical corrections.

To accomplish this, add the phrase 'Distribution of substantively modified versions of this document is prohibited without the explicit permission of the copyright holder.' to the license reference or copy.

B. To prohibit any publication of this work or derivative works in whole or in part in standard (paper) book form for commercial purposes is prohibited unless prior permission is obtained from the copyright holder.

To accomplish this, add the phrase 'Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.' to the license reference or copy.