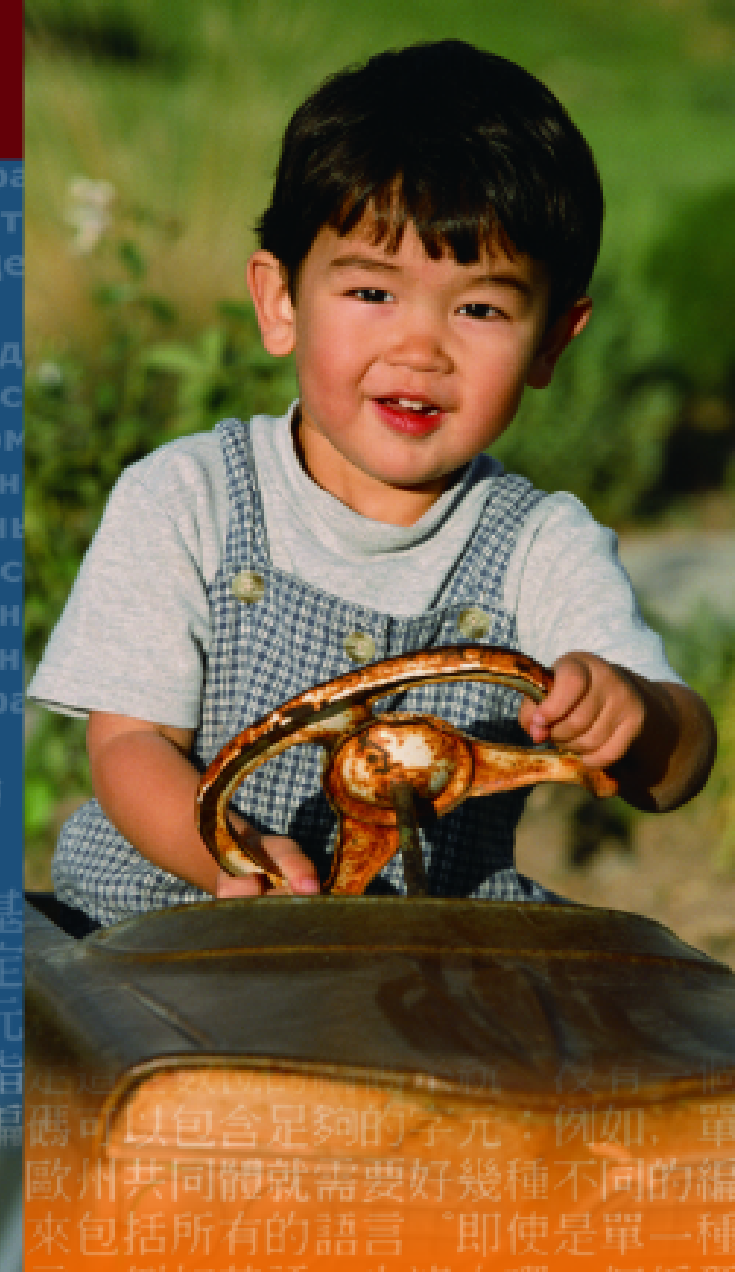
imperia

Guido Flohr

Unicode und Multi-Language



HOWTO

Imperia Unicode- und Multi-Language-HOWTO

Konzeption, Implementierung und Pflege mehrsprachiger Web-Sites mit Imperia

Guido Flohr

Imperia Unicode- und Multi-Language-HOWTO: Konzeption, Implementierung und Pflege mehrsprachiger Web-Sites mit Imperia

Guido Flohr

Veröffentlicht 25. September 2003

Copyright © 2003 Guido Flohr, Imperia AG [http://www.imperia.net/] Huerth/Germany

Alle Rechte vorbehalten.

Inhaltsverzeichnis

Einleitung ... viii
I. Allgemeines ... 1
1. Die Zeit vor Unicode ... 4
1. Text im Speicher des Computers ... 4
2. ASCII ... 4
2.1. ASCII-Codeset ... 4
2.2. Von Bits und Bytes ... 5
2.3. Wieviel Bits werden für ASCII verwendet? ... 5
3. 8-Bit-Codesets ... 6
3.1. ISO 8859 ... 7
3.1.1. ISO-8859-1 (Latin-1) ... 7
3.1.2. ISO-8859-2 (Latin-2) ... 7
3.1.3. ISO-8859-3 ... 8
3.1.4. ISO-8859-4 ... 8
3.1.5. ISO-8859-5 ... 8
3.1.6. ISO-8859-6 ... 8
3.1.7. ISO-8859-7 ... 9
3.1.8. ISO-8859-8 ... 9
3.1.9. ISO-8859-9 (Latin-5) ... 9
3.1.10. ISO-8859-10 (Latin-6) ... 9
3.1.11. ISO-8859-11 ... 9
3.1.12. ISO-8859-13 (Latin-7) ... 9
3.1.13. ISO-8859-14 (Latin-8) ... 9
3.1.14. ISO-8859-15 (Latin-9) ... 9
3.1.15. ISO-8859-16 (Latin-10) ... 9
3.2. KOI ... 10
3.3. Microsoft ... 12
3.3.1. CP1252 (Windows-1252) ... 12
3.3.2. ISO-8859-15 Revisited ... 13
3.3.3. Andere Sprachen, andere Windows-Codepages ... 15
3.4. Weitere Hersteller ... 15
3.5. Weitere Sprachen ... 15
3.5.1. Georgisch ... 16
3.5.2. Vietnamesisch ... 16
4. Multibyte-Codesets ... 16
4.1. CJK-Sprachen ... 17
4.1.1. Chinesisch ... 17
4.1.2. Japanisch ... 17
4.1.3. Koreanisch ... 17
4.2. Escaping ... 18
4.3. Multi-Byte-Encodings ... 19
4.4. Bekannte Multi-Byte-Encodings ... 21
5. Zusammenfassung ... 21

2. Die schöne Welt von Unicode ... 23
1. Was ist Unicode? ... 23
2. Der Wertebereich von Unicode ... 23
3. Unicode-Properties ... 24
4. Technische Repräsentation von Unicode ... 24
4.1. Wide Characters ... 24
4.1.1. Probleme mit Wide Characters ... 24
4.1.1.1. Auf- und Abwärtskompatibilität ... 25
4.1.1.2. Speicherverbrauch ... 27
4.1.1.3. Synchronisation ... 27
4.1.1.4. File-System-Safety ... 28
4.1.1.5. Was haben Eier mit Bytes zu tun? ... 28
4.1.2. UTF-16 ... 30
4.1.3. UCS-2 ... 31
4.1.4. UCS-4 ... 31
4.2. Multi-Byte-Encodings ... 31
4.2.1. UTF-7 ... 31
4.2.2. UTF-8 ... 32
4.2.2.1. ASCII-Transparenz ... 35
4.2.2.2. Selbstsynchronisierung ... 35
4.2.2.3. Speicherverbrauch ... 35
4.2.2.4. Nachteile von UTF-8 ... 36
5. Zusammenfassung ... 36
II. Unicode und Multi-Language im Web ... 38
3. Web-Standards ... 40
1. Hypertext Transfer Protocol HTTP ... 40
1.1. Funktionsweise von HTTP ... 40
1.2. Der Header Content-Type ... 43
1.2.1. Die Konfigurationsanweisung AddDefaultCharset ... 43
1.2.2. Die Konfigurationsanweisung AddCharset ... 44
1.3. Der Header Content-Language ... 44
1.3.1. Die Konfigurationsanweisung DefaultLanguage ... 46
1.3.2. Die Konfigurationsanweisung AddLanguage ... 46
1.4. Voreingestellte Werte ... 47
1.5. Der Header Accept-Charset ... 47
1.6. Der Header Accept-Language ... 48
1.7. Formulare ... 49
2. Extensible Markup Language XML ... 51
2.1. Perfomance-Erwägungen ... 52
2.2. XML in UTF-8 ... 53
2.3. Sprachbestimmung ... 53
3. Hypertext Markup Language HTML ... 54
3.1. Das HTML-Attribut lang ... 54
3.2. Das Attribut http-equiv des meta-Elements. ... 54
3.3. XHTML ... 56
4. Content-Negotiation ... 56
4.1. Implementierung mit dem Apache ... 56
4.2. Praktische Erwägungen ... 58

4.2.1. Sprachpersistenz 58
4.2.2. Sprachumschaltung 59
4.2.3. Vertikale oder horizontale Aufteilung 59
5. Zusammenfassung 60
III. Unicode und Multi-Language mit Imperia 61
4. Allgemeines 63
1. Copy-Pages 63
2. Unicode-Unterstützung in Imperia 63
2.1. System- und User-Charset 64
2.1.1. Wirkung der Sprach- und Charseteinstellungen 64
2.1.2. Stolpersteine 65
2.1.3. Templates 66
2.1.4. Meta-Dateien 67
2.1.5. Konfiguration des Web-Servers 67
2.2. Best Practice - eine Empfehlung 67
2.3. Weitere Unicode-Features 68
5. Ein Beispielszenario 69
1. Die Anforderungen 69
2. Vorbereitung 69
2.1. System-Einstellungen 70
2.2. Template und Meta-Datei 70
2.3. Eine Test-Rubrik 72
3. Basis-Implementierung 72
3.1. Mehrsprachige Eingabe 72
3.1.1. Verallgemeinerung der Spracheingaben 73
3.1.2. Parametrisierte Code-Includes 73
4. Dynamische Sprachwahl 75
4.1. Anpassung der Meta-Datei 75
4.2. Ein einfaches Workflow-Plug-In 77
4.2.1. Gerüst für das Plug-In 78
4.2.2. Workflow-Definition 78
4.2.3. Implementierung der Plug-In-Logik 79
4.3. Dynamische Einbindung der Code-Includes 82
5. Reduzierung der Templates 83
6. Charset-Konvertierung 85
7. Verfeinerungen 87
7.1. Problemfall Flexmodule 87
7.2. Der Flex-Flexxer 88
7.3. Mehr Komfort durch DHTML 90
7.3.1. Erzeugen der Layer 91
7.3.2. Die DHTML-Erweiterungen am Template 92
7.3.3. Das Layer-Menü 93
8. Zusammenfassung 94
Anhang. 95
A. Klingonisch (tIhIngan Hol) 97
Weiterführende Informationen 98

Tabellenverzeichnis

2.1. Anfangsbyte von UTF-8 32
2.2. Prinzip von UTF-8 33

Einleitung

Wer jemals mit Texten zu tun hatte, die weder in englisch noch in einer westeuropäischen Sprache verfasst sind, ist sicherlich schon einmal mit dem Begriff *Unicode* in Berührung gekommen. Mit Imperia ist es leicht, Content in beliebigen Sprachen zu erfassen und zu pflegen. Allerdings sind dabei einige Besonderheiten zu beachten, die möglichst schon *vor* Beginn des Projektes Beachtung finden sollten.

Dieses Dokument versucht alle Aspekte der Unicode-Unterstützung in Imperia zu behandeln. Dafür ist natürlich ein gewisses Grundverständnis dafür erforderlich, was sich hinter dem Begriff Unicode überhaupt verbirgt, und in welchem Zusammenhang Unicode mit der Publizierung von Content steht. Bei der Vermittlung dieser Hintergrundinformationen wurde mehr Wert auf Verständlichkeit denn auf hundertprozentige technische Exaktheit gelegt, um auch weniger technisch versierten Anwendern den Einstieg in die Thematik zu erleichtern.

Die Pflege mehrsprachiger Websites hat nicht notwendigerweise etwas mit Unicode zu tun. Das Thema ist jedoch eng genug verwandt, um eine Behandlung innerhalb desselben Dokumentes zu rechtfertigen.

Teil I. Allgemeines

Inhaltsverzeichnis

1. Die Zeit vor Unicode ... 4
1. Text im Speicher des Computers ... 4
2. ASCII ... 4
2.1. ASCII-Codeset ... 4
2.2. Von Bits und Bytes ... 5
2.3. Wieviel Bits werden für ASCII verwendet? ... 5
3. 8-Bit-Codesets ... 6
3.1. ISO 8859 ... 7
3.1.1. ISO-8859-1 (Latin-1) ... 7
3.1.2. ISO-8859-2 (Latin-2) ... 7
3.1.3. ISO-8859-3 ... 8
3.1.4. ISO-8859-4 ... 8
3.1.5. ISO-8859-5 ... 8
3.1.6. ISO-8859-6 ... 8
3.1.7. ISO-8859-7 ... 9
3.1.8. ISO-8859-8 ... 9
3.1.9. ISO-8859-9 (Latin-5) ... 9
3.1.10. ISO-8859-10 (Latin-6) ... 9
3.1.11. ISO-8859-11 ... 9
3.1.12. ISO-8859-13 (Latin-7) ... 9
3.1.13. ISO-8859-14 (Latin-8) ... 9
3.1.14. ISO-8859-15 (Latin-9) ... 9
3.1.15. ISO-8859-16 (Latin-10) ... 9
3.2. KOI ... 10
3.3. Microsoft ... 12
3.3.1. CP1252 (Windows-1252) ... 12
3.3.2. ISO-8859-15 Revisited ... 13
3.3.3. Andere Sprachen, andere Windows-Codepages ... 15
3.4. Weitere Hersteller ... 15
3.5. Weitere Sprachen ... 15
3.5.1. Georgisch ... 16
3.5.2. Vietnamesisch ... 16
4. Multibyte-Codesets ... 16
4.1. CJK-Sprachen ... 17
4.1.1. Chinesisch ... 17
4.1.2. Japanisch ... 17
4.1.3. Koreanisch ... 17
4.2. Escaping ... 18
4.3. Multi-Byte-Encodings ... 19
4.4. Bekannte Multi-Byte-Encodings ... 21
5. Zusammenfassung ... 22
2. Die schöne Welt von Unicode ... 23
1. Was ist Unicode? ... 23

2. Der Wertebereich von Unicode 23
3. Unicode-Properties 24
4. Technische Repräsentation von Unicode 24
4.1. Wide Characters 24
4.1.1. Probleme mit Wide Characters 24
4.1.1.1. Auf- und Abwärtskompatibilität 25
4.1.1.2. Speicherverbrauch 27
4.1.1.3. Synchronisation 27
4.1.1.4. File-System-Safety 28
4.1.1.5. Was haben Eier mit Bytes zu tun? 28
4.1.2. UTF-16 30
4.1.3. UCS-2 31
4.1.4. UCS-4 31
4.2. Multi-Byte-Encodings 31
4.2.1. UTF-7 31
4.2.2. UTF-8 32
4.2.2.1. ASCII-Transparenz 35
4.2.2.2. Selbstsynchronisierung 35
4.2.2.3. Speicherverbrauch 35
4.2.2.4. Nachteile von UTF-8 36
5. Zusammenfassung 36

Kapitel 1. Die Zeit vor Unicode

Der folgende Schnelldurchlauf soll kurz die Entwicklung von den auf frühen Computersystemen verwendeten Codesets zu Unicode beleuchten. Der Fokus liegt dabei eindeutig auf der Relevanz für Web-Authoring. Für detaillierte Informationen sei auf die ausgezeichneten Darstellungen auf [czyborra.com] von Rainer Czyborra und [Standards] von Dik T. Winter verwiesen.

1. Text im Speicher des Computers

Computer werden auch *Rechner* genannt, weil ihre Stärke im Rechnen, also im Umgang mit Zahlen liegt. Mit Sprache, ganz gleich ob gesprochen oder geschrieben, haben sie ihre Schwierigkeiten, und so ist es nicht verwunderlich, dass Computer Texte einfach als Zahlenfolge behandeln. Jeder Buchstabe eines Textes ist im Speicher des Computers als eine Zahl abgelegt.

Was passiert aber, wenn ein Text am Bildschirm dargestellt werden soll? Dafür muss der Computer zu jeder Zahl, zu jedem *Zeichencode* eine passende Grafik haben, damit er, wenn er im Speicher auf beispielsweise die Zahl 65 trifft, etwas am Bildschirm darstellen kann, das wie ein großes A aussieht. Solange man die Texte nur auf dem eigenen Rechner bearbeitet, ist die Zuordnung der nummerischen Codes zu ihrer graphischen Repräsentation (die auch als *Glyph* bezeichnet wird), willkürlich, sie sollte sich nur nicht ändern. Will man dagegen eine Textdatei auf einem anderen Rechner, mit womöglich einem anderen Betriebssystem, ebenfalls bearbeiten, tut eine Konvention Not, welcher Zahl denn welches Zeichen entsprechen soll.

2. ASCII

In den Urzeiten des Computers war Interoperabilität kein großes Thema, und so grenzt es fast an ein Wunder, dass sich tatsächlich eine Minimalkonvention für die Zuordnung von Codes zu Zeichen durchsetzen konnte, nämlich ASCII, der *American Standard Code for Information Interchange*. *Durchsetzen* heißt allerdings nicht, dass es der einzig verwendete Code ist. Es gibt tatsächlich Ausnahmen wie beispielsweise der auf Großrechnern und Mainframes verbreitete EBCDIC.

Die erste Version von ASCII stammt aus dem Jahre 1963. Sie hat bis zur heute gültigen Version aus dem Jahre 1967 einige Änderungen erfahren, die detailliert unter http://www.cwi.nl/~dik/english/codes/stand.html auf [Standards] beschrieben sind.

2.1. ASCII-Codeset

Wie sieht die Zuordnung in ASCII, jetzt genau aus? Die Codes 0-31 sind sogenannte Steuer- bzw. Controlcodes, die für nicht-druckbare Zeichen verwendet werden, beispielsweise Zeilenumbruch oder Tabulator. Das Leerzeichen hat die Nummer 32, die Ziffern 0-9 haben die Codes 30-39, die Großbuchstaben A-Z die Codes 65-90, und die Kleinbuchstaben a-z sind mit 97-122 codiert. Die Zwischenbereiche werden für Interpunkti-

onszeichen und Computerspezifisches wie den Klammeraffen „@“ verwendet.

2.2. Von Bits und Bytes

Was genau ist denn eigentlich ein *Byte*? Nun, Computer können eigentlich noch nicht einmal (Dezimal-)Zahlen speichern. Vielmehr besteht der Speicher des Computers aus Milliarden oder sogar Billionen winziger Schalter, die entweder ein- oder ausgeschaltet sind. Mit einem solchen System lassen sich daher keine normalen, also Dezimalzahlen, sondern nur *Binärzahlen*, Zahlen, die lediglich mit Nullen und Einsen dargestellt werden, repräsentieren. Diese einzelnen Kippschalter heißen *Bits*, und damit die Rechnerei nicht ganz so wüst wird, werden 8 Bits normalerweise zu einem *Byte* zusammengefasst. Steht an einer Speicherzelle im Computer also die Zahl 65, sieht das in Wirklichkeit so aus:

Die Zahl 65 in Binärdarstellung

Irgendwo im Speicher unseres Computers liegen 8 Birnchen, einige an, einige aus, und die Kombination von An und Aus steht für die Zahl 65. Aber es sind nur 8 Birnchen, und uns beschleicht der Verdacht, dass man mit diesen 8 Birnchen nicht allzuviele verschiedene Zahlen darstellen kann. Wieviele Kombinationen sind denn wirklich mit diesen acht Birnchen möglich? Mathematik leider nur ausreichend minus, also raten wir: Acht! Nein, Moment, kann nicht sein, die Kombination oben steht ja schon für 65. Also schauen wir im Lexikon nach. Aha, es sind $2^8 = 256$. Hab' ich doch gleich gesagt.

Wir müssen aber auch noch die Zahl Null darstellen können. Also kommen wir mit unseren 8 Birnchen/Bits pro Byte nur von 0-255, weil wir die 256 noch der Null opfern. Weit ist das ja nicht. Wenn man allerdings zwei Bytes (16 Bit) nebeneinander betrachtet, sieht die Sache schon anders aus, denn jetzt haben wir für jede der 256 Kombinationen beim rechten Byte noch 256 Kombinationen beim linken Byte, insgesamt also 256 x 256 = 2562 = 65536. Bei vier Bytes (32 Bit) auf einmal sind es schon 4.294.967.296, bei den heute schon gebräuchlichen acht Bytes (64 Bit) sogar 18.446.744.073.709.551.616. Es kommt Hoffnung auf.

2.3. Wieviel Bits werden für ASCII verwendet?

Speichern wir eine Text-Datei als blanken Text ab (also nicht mit einem Textverarbeitungsprogramm, sondern z. B. mit dem Notepad unter Windows oder vi unter Unix) stellen wir an der Dateigröße fest, dass pro Zeichen (Leerzeichen nicht vergessen!) ungefähr ein Byte an Speicherplatz verbraucht wird. Wenn wir beim Zählen auch Zeilenumbrüche und Tabulatoren nicht vergessen, stellen wir fest, dass es sogar genau ein Byte ist (unter DOS/Windows wird ein Zeilenumbruch allerdings aus Gründen, die das Geheimnis des Erfinders bleiben werden, als zwei Bytes abgespeichert, pro Zeile muss daher noch ein überflüssiges Byte dazugerechnet werden).

Also könn(t)en in einer Textdatei nur 256 verschiedene Zeichen verwendet werden. Ist das wirklich so? Nein, bei ASCII ist es tatsächlich nur die Hälfte, und davon gehen noch einmal die 32 Steuerzeichen (und genaugenommen auch noch die Nummer 127, das

Zeichen für die Taste **Entf** bzw. **Del**) ab. Bei ASCII werden nämlich nur 7 von 8 möglichen Bits verwendet, hauptsächlich wohl, weil zur Zeit der Definintion von ASCII die meisten Computer-Anwendungen das achte Bit für interne Zwecke verwendeten. Das gilt mit Einschränkungen sogar noch heute. SMTP (*Simple Mail Transfer Protocol*, siehe [RFC821], inzwischen ersetzt durch [RFC2821] oder auch [RFC822], inzwischen ersetzt durch [RFC2822]), das Netzwerkprotokoll, mit dem E-Mail verschickt wird, kann erst mit einer Erweiterung 8-Bit-Zeichen übertragen. Weil aber noch genügend Mail-Server ohne diese Erweiterung existieren, ist es noch heute üblich, 8-Bit-Mails vor der Übertragung in eine 7-Bit-Darstellung zu konvertieren.

Mit ASCII lassen sich also tatsächlich weniger als 100 Zeichen darstellen. Wir schauen noch einmal im Abschnitt 2.1, „ASCII-Codeset“ nach, welche Zeichen das etwa sind. Wie sieht es beispielsweise mit deutschen Umlauten, französischen akzentuierten Buchstaben, skandinavischen Zeichen aus? Oder griechischen, hebräischen, arabischen, kyrillischen, chinesischen Zeichen. Die sind allesamt nicht definiert, wohlgemerkt, auch die deutschen Umlaute nicht! Wer einen deutschen Text mit Umlauten also als ASCII-Text bezeichnet, weiß nicht, was ASCII ist.

3. 8-Bit-Codesets

Die meisten Computer-Anwendungen kommen schon seit geraumer Zeit blendend mit 8-Bit-Textdaten zurecht, und die Beschränkung auf 7 Bit scheint daher nicht mehr so ganz zeitgemäß. Allerdings müssen wir uns daran erinnern, dass es für die Interoperabilität entscheidend ist, dass eine eindeutige Konvention für die Zuordnung von nummerischen Codes zu entsprechenden Zeichen gegeben ist. Für die Zeichen 0-127 ist das durch ASCII weitestgehend gewährleistet, von 128-255 existiert keine solche Konvention.

Eine Zeitlang herrschte in diesem Bereich sogar völliges Chaos. Jeder Hersteller benutzte seine eigene Konvention für den 8-Bit-Bereich, meistens sogar mehrere, abhängig davon, für welches Land das jeweilige System hergestellt wurde. IBM beispielsweise belegte den Bereich mit einfachen Grafikzeichen (die sogenenannte *PC ruler*), um mit Winkeln, Strichen und Kreuzen einfache Tabellen, später sogar einfache Fenstersysteme darstellen zu können, in kleinen Teilbereichen waren länderspezifische Zeichen wie deutsche Umlaute, akzentuierte Buchstaben, skandinavische Zeichen etc. darzustellen. Der Atari war teilweise kompatibel zum IBM-PC, statt der Grafikzeichen wurden jedoch hebräische und griechische Zeichen verwendet, der Macintosh hatte ebenfalls seine eigene Zuordnung, auf HP-Systemen ist noch heute ein Codeset namens HP-Roman8 Standard, ...

Wer früher Texte zwischen Computersystemen austauschte, die nicht in Englisch verfasst waren, war daher daran gewöhnt, dass Texte entweder völlig verhunzt ankamen, oder in einer 7-Bit-Ersatzdarstellung verfasst wurden (z. B. „ae“ statt „ä“, „'e“ statt „é“, „c,“ statt „ç“ und so weiter). Verhunzt heißt nicht, dass die Daten tatsächlich beim Transport beschädigt wurden, lediglich die Zuordnung von Codes zu Zeichen stimmte nicht. Tippte ein Unix-Nutzer beispielsweise ein großes „Ä“ ein, wurde dies als Code Nummer 196 gespeichert, was ein Apple-Macintosh als großes F mit Haken („ƒ“) interpretierte.

3.1. ISO 8859

Die *International Standardization Organization* ISO versuchte mit der Norm ISO 8859 eine gewisse Ordnung in das Chaos zu bringen. In der Norm sind heute 14 Standard-Codesets definiert.

Alle Codesets der Norm ISO 8559 weisen mehr oder weniger große Überschneidungen auf. Im Bereich 128-159 liegen zusätzliche (nicht druckbare) Steuerzeichen, das Zeichen mit der Nummer 160 ist stets das *no-break space*, das wir in HTML als ` ` kennen, Nummer 173 ist ein *Soft Hyphen* - `` (Ausnahme, das noch nicht offiziell standardisierte, auf TIS-620 basierende ISO-8859-11). Auch andere Latin-1-Sonderzeichen, die in ASCII nicht enthalten sind, wie §, ©, ° liegen oft (aber nicht immer) an der gleichen Stelle.

Nicht alle Codesets sind in der Praxis wirklich relevant. Einige haben sich nur teilweise, einige gar nicht durchgesetzt.

3.1.1. ISO-8859-1 (Latin-1)

Geeignet hauptsächlich für westeuropäische und ähnliche Sprachen, zum Beispiel Afrikaans, Bretonisch, Katalanisch, Walisisch, Dänisch, Deutsch, Englisch, Spanisch, Estnisch, Baskisch, Finnisch, Färöisch (Färingisch), Französisch/Wallonisch, Irisch (Gälisch), Galizisch, Manx, Indonesisch (Bahasa Indonesia) , Isländisch, Italienisch, Eskimoisch (Westgrönländisch/Kalaallisut), Malaysisch, Niederländisch/Flämisch, Norwegisch (Bokmål) und Neunorwegisch (Nynorsk), Okzitanisch (->Frankreich), Portugiesisch, Albanisch, Schwedisch, Tagalog (->Philippinen), Usbekisch und viele mehr.

ISO-8859-1 ist heute ein De-Fakto-Standard für 8-Bit-Textdaten, und wird praktisch von allen Computersystemen unterstützt. Prominente Ausnahme im Unix-Bereich ist Hewlett-Packards HP-UX, das ein eigenes Codeset für westeuropäische Sprachen verwendet, Roman8 oder auch HP-Roman8. Bei den Micros relevant sind noch das MS-DOS-Codeset, das noch immer von DOS-Anwendungen unter MS-Windows verwendet wird (Beweis: DOS-Box öffnen und **`type umlaute.txt`** eingeben, wenn `umlaute.txt` eine Textdatei mit deutschen Umlauten ist). Inwieweit die Macintosh-Codesets auf dem Apple noch aktuell sind, ist nicht bekannt.

3.1.2. ISO-8859-2 (Latin-2)

Geeignet für mittel- und osteuropäische Sprachen mit lateinischen Alphabeten, insbesondere Bosnisch, Tscheschisch, Kroatisch, Ungarisch, Polnisch, Rumänisch, Slowakisch, Slowenisch, Serbisch (mit lateinischen Schriftzeichen), etc.

Auch ISO-8859-2 ist im osteuropäischen Raum (soweit keine kyrillischen Zeichen benutzt werden) weitgehend durchgesetzt. Für deutsche Anwender weist ISO-8859-2 die interessante Eigenschaft auf, dass die deutschen Umlaute und das Eszet nicht nur in ISO-8859-2 vorhanden sind, sondern auch an den gleichen Stellen liegen (also die gleichen nummerischen Codes haben). In der Praxis heißt das, dass man deutsche Texte auch in ISO-8859-2 darstellen kann, und somit in ein und derselben Datei deutsche und z. B. polnische Textpassagen haben kann.

3.1.3. ISO-8859-3

Dieses Codeset ist laut Standard für „südosteuropäische und sonstige Sprachen“ vorgesehen. Mir persönlich ist hier nur Maltesisch bekannt.

3.1.4. ISO-8859-4

Geeignet für skandinavische und baltische Sprachen.

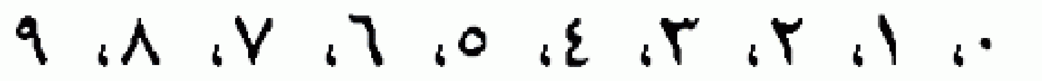
3.1.5. ISO-8859-5

Prinzipiell ist dieses Codest für Bulgarisch, Mazedonisch, Russisch, Serbisch (mit kyrillischen Schriftzeichen), Tadschikisch, Ukrainisch, Weißrussisch, etc. geeignet.

Im osteuropäischen Sprachraum, der kyrillische Schriftzeichen verwendet, steht ISO-8859-2 auf der Bedeutungsskala jedoch nur an dritter Stelle. Den ersten Platz in der Gunst der Anwender dürften sich die KOI8-Codesets und die Codesets der Firma Microsoft teilen (wobei die KOI8-Codesets etwas verbreiteter sein sollten). Beiden Familien von Codesets ist ein eigener Abschnitt gewidmet.

3.1.6. ISO-8859-6

Das Codeset ISO-8859-6 ist für Arabisch stark verbreitet, was allerdings nicht zu der Illusion verleiten sollte, dass sich damit arabischer Text adäquat darstellen ließe. Das arabische Alphabet kennt 28 Buchstaben mit jeweils vier, meist unterschiedlichen Schreibweisen. Hierfür braucht man schon 112 unterschiedliche Codes (die ISO-Codesets haben nur 96 Stellen frei, weil die ersten 32 für Steuerzeichen verwendet werden). Hinzu kommen noch die zehn Ziffern, denn die wirklichen arabischen Ziffern unterscheiden sich in der Darstellung erheblich von unseren „arabischen“ Ziffern:



Die arabischen Ziffern 0-9

Die Ziffern sind natürlich von rechts nach links zu lesen (die Null ist also der Punkt, und die Neun ist das Zeichen, das unserer Neun sehr ähnlich sieht) und sind durch (arabische) Kommata getrennt. Die Satzzeichen wie Punkt, Komma, Semikolon weichen ebenfalls mehr oder weniger stark von der uns geläufigen Darstellung ab.

Hinzu kommt eine Unzahl von Ligaturen, Buchstabengruppen also, die über- und untereinander und ineinander verwoben dargestellt werden. Diese Ligaturen werden keineswegs nur in kalligraphischen Schriften, sondern durchaus auch im Alltagsgebrauch verwendet. Prominentestes Beispiel ist die Darstellung für Gott ﷲ („Allah“), die praktisch durchweg in dieser Form geschrieben wird. Die arabische Sprache und Schrift hat für gläubige Muslime eine spirituelle Bedeutung, und somit sind solche Eigenheiten besonders ernst zu nehmen. ISO-8859-6 wird diesen Anforderungen nur begrenzt gerecht.

3.1.7. ISO-8859-7

Dieses Codeset ist Standard für Griechisch.

3.1.8. ISO-8859-8

Dieses Codeset enthält den hebräischen Zeichenvorrat.

3.1.9. ISO-8859-9 (Latin-5)

Latin-5 ist eine Variante von Latin-1 für Türkisch.

3.1.10. ISO-8859-10 (Latin-6)

Geeignet für Lappisch, sowie weitere nordische und Eskimo-Sprachen.

3.1.11. ISO-8859-11

ISO-8859-11 ist zur Zeit noch kein offizieller Standard. Es ist praktisch identisch mit dem Codeset TIS-620 für Thailändisch, enthält aber zusätzlich noch das No-Break-Space (aber kein Soft Hyphen).

3.1.12. ISO-8859-13 (Latin-7)

Mit ISO-8859-13 lassen sich interessanterweise nicht nur die baltischen Sprachen Lettisch und Litauisch, sondern auch Maori darstellen, eine Tatsache die allerdings mehr zum Schmunzeln denn zu sprachwissenschaftlichen Spekulationen anregen sollte.

3.1.13. ISO-8859-14 (Latin-8)

ISO-8859-14 enthält akzentuierte Zeichen für keltische Sprachen.

Hinweis: Irisch (*Gaeilge*) wird gewöhnlich in ISO-8859-1 bzw. ISO-8859-15 codiert. Die aus dem Mittelalter stammende traditionelle irische Schrift wird auch in Irland selbst mehr und mehr durch eine vom lateinischen Alphabet abgeleitete Schrift verdrängt (s. [Ó Siadhail], S. 9).

3.1.14. ISO-8859-15 (Latin-9)

ISO-8859-15 ist der designierte Nachfolger von ISO-8859-1 (s. Abschnitt 3.1.1, „ISO-8859-1 (Latin-1)“). Es ersetzt einige wenig benutzte Zeichen durch die in Latin-1 „vergessenen“ Zeichen für Französisch und Finnisch, nämlich die O-E-Ligaturen (Œ und œ), das große Y mit Trema (Ÿ), und die Konsonanten S und Z mit Caret (Š, š, Ž und ž). Die praktisch bedeutsamste Änderung ist die Ersetzung des allgemeinen Währungssymbols ¤ (das ohnehin kaum jemand zu deuten weiß) durch das Euro-Symbol €.

3.1.15. ISO-8859-16 (Latin-10)

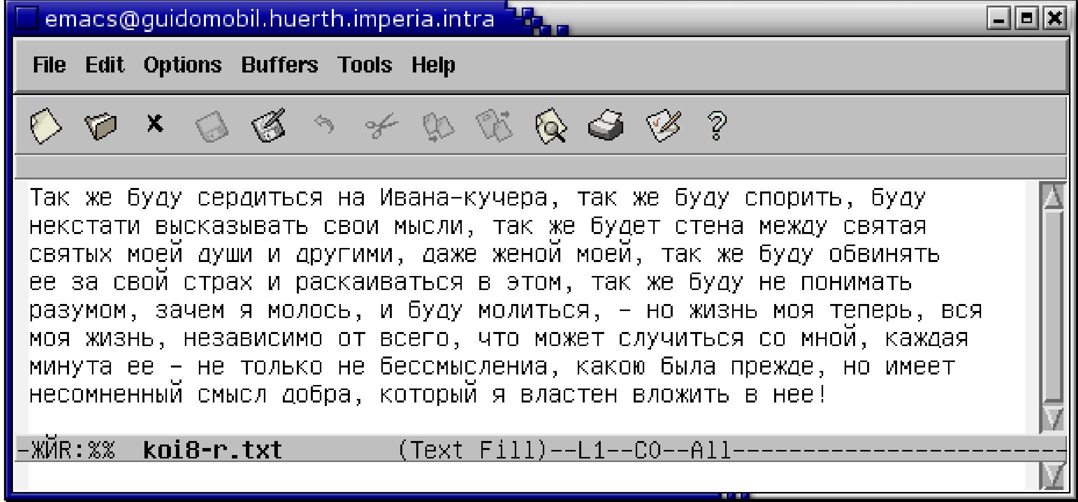
Dieser Standard ist für „sonstige osteuropäische Sprachen“ vorgesehen. Einziges bekanntes Beispiel ist Rumänisch.

3.2. KOI

Parallel zur Standardisierung im Westen, insbesondere mit ASCII, gab es ähnliche Bemühungen in Osteuropa, insbesondere in der ehemaligen Sowjetunion. Ungefähr zur gleichen Zeit wie die ersten Versionen von ASCII enstanden auch die ersten sowjetischen Codeset-Standards des GOST, des sowjetischen Normungsinstituts. GOST 10859 definierte die 10 Ziffern, 22 Interpunktions- und Sonderzeichen, gefolgt vom russischen Alphabet (nur Großbuchstaben), wiederum gefolgt von den 14 lateinischen Buchstaben ohne (graphische) kyrillische Entsprechung, wiederum gefolgt von weiteren Sonderzeichen und dem Härtezeichen *Twjordyj Znak*.

Mit GOST 13052 folgte ein volles 7-Bit Codeset, in der unteren Hälfte entsprach es exakt dem ASCII-Codeset (das Dollarzeichen $ war natürlich durch das allgemeine Währungssymbol ¤ ersetzt), in der oberen Hälfte waren für die lateinischen Buchstaben jeweils die kyrillischen Zeichen angeordnet, die lautlich dem lateinischen Pendant am nächsten kamen, wobei allerdings Groß- und Kleinbuchstaben vertauscht waren. Wir erinnern uns kurz zurück an die Zeiten, als die ewigen Weltmeister aus der UdSSR noch die Aufschrift CCCP[1] auf ihren Eishockey-Leibchen trugen. Und kein Reporter verpasste während der Endspielübertragung die Gelegenheit, darauf hinzuweisen, dass diese Buchstabenfolge tatsächlich „SSSR“ bedeutete. Und der hungrige Moskau-Tourist lernte spätestens am zweiten Tag, dass sich in den Häusern mit dem Schild *PECTOPAH* ein „RESTORAN“ verbarg.

Dieser Kniff hatte im Alltag enorme Vorteile. Schickte ein Hacker aus Moskau seinem ins Silicon Valley geflüchteten Vetter den letzten Absatz aus Tolstois *Anna Karenina*, so sah das an seinem GOST-13052-Terminal zuhause so aus:



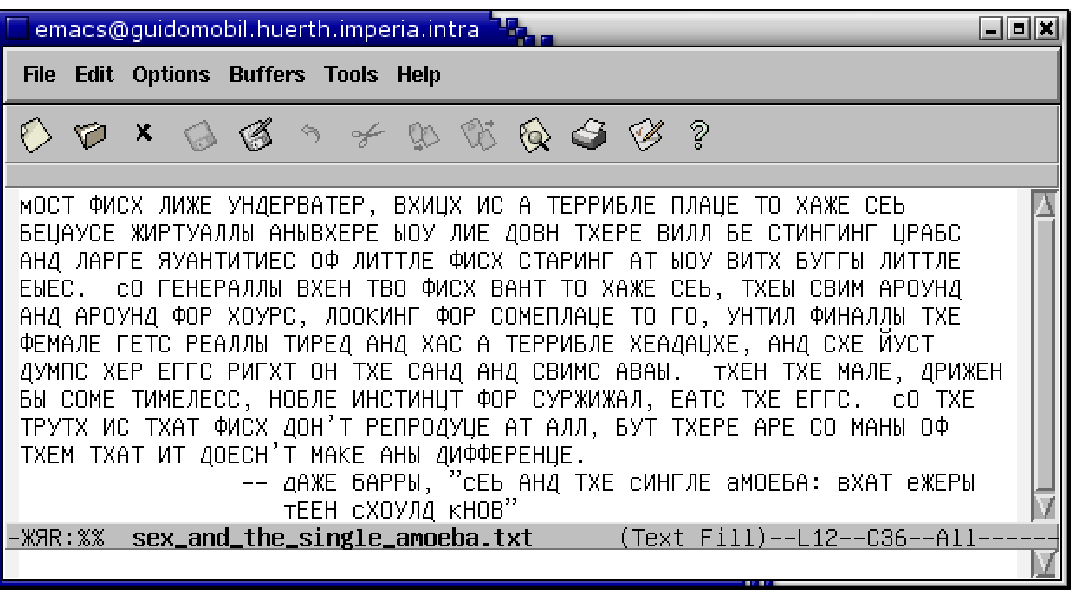
Die letzten Zeilen aus Anna Karenina (Tolstoi)

[1] Aufgrund der zahlreichen Anfragen bei Google: Die kyrillische Zeichenfolge CCCP entpricht tatsächlich den Buchstaben SSSR - ein kyrillisches C entspricht also dem deutschen S, und das Pendant des kyrillischen P ist ein deutsches R - und steht für (krude Transliteration) *Sojus Sowjetskich Sozialistitscheskich Respublik*, also *Union der Sozialistischen Sowjet-Republiken - UdSSR*, oder englisch *Union of Socialist Soviet Republics - USSR*. Um gleich weiteren Fragen vorzubeugen: Ein *Sowjet* ist einfach ein Rat, im Sinne von Beratungsgremium. Die Sowjetunion war also eine *Räterepublik*. Das russische Wort *Sojus* (normalerweise *Sajuus* ausgesprochen, also mit Betonung auf der zweiten langen Silbe, das O der erste Silbe wird zu A) bedeutet Union und war Taufpate für etliche sowjetische Weltraummissionen. Mehr Info? Lernt selber Russisch! Es gibt wohl kaum eine schönere und poetischere Sprache auf diesem Planeten.

Am ASCII-Terminal am anderen Ende der Welt, präsentierte sich dieser Text dann wiederum so:

> tAK VE BUDU SERDITXSQ NA iWANA-KU^ERA, TAK VE BUDU SPORITX, BUDU NEKSTATI WYSKAZYWATX SWOI MYSLI, TAK VE BUDET STENA MEVDU SWQTAQ SWQTYH MOEJ DU[I I DRUGIMI, DAVE VENOJ MOEJ, TAK VE BUDU OBWINQTX EE ZA SWOJ STRAH I RASKAIWATXSQ W \TOM, TAK VE BUDU NE PONIMATX RAZUMOM, ZA^EM Q MOLOSX, I BUDU MOLITXSQ, - NO VIZNX MOQ TEPERX, WSQ MOQ VIZNX, NEZAWISIMO OT WSEGO, ^TO MOVET SLU^ITXSQ SO MNOJ, KAVDAQ MINUTA EE - NE TOLXKO NE BESSMYSLENIA, KAKO@ BYLA PREVDE, NO IMEET NESOMNENNYJ SMYSL DOBRA, KOTORYJ Q WLASTEN WLOVITX W NEE!

Er sah eine krude, nach kurzer Übungszeit aber durchaus lesbare Transliteration des russischen Originals. Durch die Vertauschung von Groß- und Kleinbuchstaben war auch sofort erkennbar, welche Text-Teile in der jeweils anderen Sprache geschrieben waren. Antwortete der Vetter mit den neuesten sozialpsychologischen Erkenntnissen aus dem Land der unbegrenzten Möglichkeiten, kamen auch diese Weisheiten wieder lesbar am GOST-Terminal in Moskau an:



Left as an exercise to the reader...

Dieses Konzept hatte sich bewährt, und wurde auch im später auf 8 Bit (also 256 Zeichen) erweiterten Standard KOI8-R ([RFC1489]) beibehalten. Die untere Hälfte entsprach jetzt exakt ASCII, in der oberen Hälfte wurden parallel zu den lateinischen Buchstaben die phonetischen Pendants der kyrillischen Schrift gelegt. Der Trick hatte weiter seine Daseinsberechtigung: Viele Mail-Applikationen waren noch immer nicht in der Lage 8-Bit zu übertragen, und setzten das achte Bit einfach auf Null zurück. Technisch gesehen hieß das, dass alle 8-Bit-Zeichen um 128 Plätze nach vorne geschoben wurden. Konnte ein kyrillischer Text also nicht korrekt übertragen werden, wurde er also tatsächlich mit lateinischen Buchstaben transliteriert, und kam noch immer lesbar am Bestim-

mungsort an.

Leider wurde dieser intelligente Trick von KOI8-R nicht in ISO-8859-5, dem offiziellen Codeset für kyrillische Schrift, übernommen, was dazu führte, dass KOI8-R im russischen Sprachraum noch immer wesentlich verbreiteter ist, als ISO-8859-5. Es gibt auch noch weitere Varianten des KOI-Codesets, insbesondere KOI8-U für Ukrainisch und KOI8-T für Tadschikisch.

3.3. Microsoft

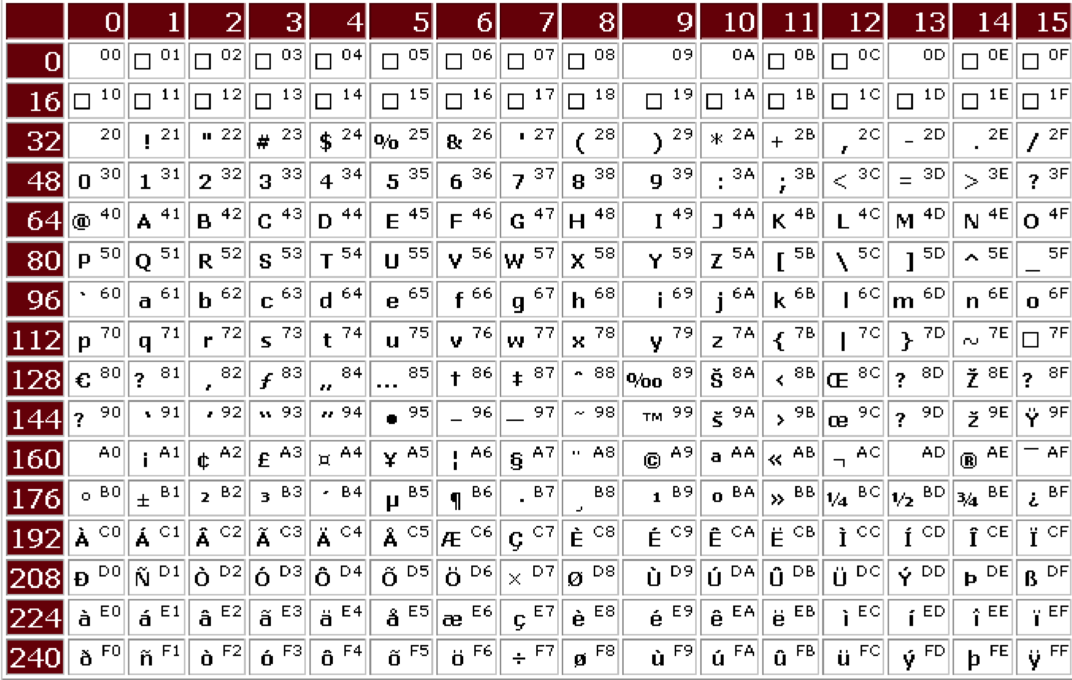
Die Firma Microsoft zeichnet sich traditionell durch eine originelle Auslegung von Normen aus. Die Interpretation von Codesets macht hier keine Ausnahme. Innerhalb der graphischen Oberfläche Windows™ ihres Betriebssystems MS-DOS™ schwenkte sie für den europäischen und amerikanischen Markt scheinbar von den in MS-DOS™ verwendeten IBM-Codesets auf das in der Unix-Welt etablierte ISO-8859-1 um.

Allerdings enthält ISO-8859-1, wie alle Codesets der Familie ISO 8859, eine Lücke, die von Microsoft ausgenutzt wurde. Die Zeichen 128-159 werden in ISO 8859 nicht verwendet. Hintergrund: Geht während der Datenübertragung das achte Bit verloren (in Wirklichkeit gibt es Applikationen, die das absichtlich machen, insbesondere Mail-Server und Mail-Reader), werden die 8-Bit-Zeichen faktisch um 128 Plätze nach vorne geschoben. Aus den Zeichen 128-159 würden dann die Steuerzeichen 0-31, was evtl. zu Problemen führen kann: Wird beispielsweise das 8-Bit-Zeichen mit der Nummer 138 nicht komplett übertragen, käme es beim Empfänger als Zeilenumbruch an, was die Lesbarkeit eines Textes nicht fördert. In pathologischen Fällen könnten auch die Vorder- und Hintergrundfarbe verstellt werden, Fenster auf dem Bildschirm verschoben werden, Pieptöne aus dem Lautsprecher kommen, und so weiter.

Auf Windows-Systemen war dieses Problem wenig relevant, auf Unix-Systemen eigentlich auch, die Problemzone wurde von ISO dennoch freigehalten. Um der Wahrheit die Ehre zu geben: Man muss schon ein ziemlich lausiger Programmierer sein, um eine Anwendung, die Texte verarbeitet, durch solche Control-Codes (also die Zeichen mit der Nummer 0-31) aus der Fassung zu bringen, denn immerhin könnte ein böswilliger Anwender diese bösartigen Steuerzeichen ja auch absichtlich z. B. als Mail verschicken. Die ISO nahm dennoch Rücksicht auf diese fehlerhaften Programme, und einige selbsternannte Gralshüter in der Unix-Welt verteidigen diese Einschränkung noch immer vehement, und verweisen bei den praktischen Erwägungen (es fehlten schlicht und ergreifend wichtige Zeichen in ISO-8859-1) auf das Allheilmittel *Unicode*. Sie übersehen dabei meist, dass praktisch alle Unicode-Anwendungen (sofern sie UTF-8 verwenden) hemmungslos, sogar in viel höherem Maße, genau diese Problemzone für textuelle Daten verwenden.

3.3.1. CP1252 (Windows-1252)

Was hat Microsoft jetzt genau gemacht? Microsoft benutzt ein gegenüber ISO-8859-1 leicht modifiziertes Codeset namens *CP1252* (CP wie CodePage) zur Darstellung westeuropäischer Texte:



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	00	□ 01	□ 02	□ 03	□ 04	□ 05	□ 06	□ 07	□ 08	09	0A	□ 0B	□ 0C	0D	□ 0E	□ 0F
16	□ 10	□ 11	□ 12	□ 13	□ 14	□ 15	□ 16	□ 17	□ 18	□ 19	□ 1A	□ 1B	□ 1C	□ 1D	□ 1E	□ 1F
32	20	! 21	" 22	# 23	$ 24	% 25	& 26	' 27	(28	) 29	* 2A	+ 2B	, 2C	- 2D	. 2E	/ 2F
48	0 30	1 31	2 32	3 33	4 34	5 35	6 36	7 37	8 38	9 39	: 3A	; 3B	< 3C	= 3D	> 3E	? 3F
64	@ 40	A 41	B 42	C 43	D 44	E 45	F 46	G 47	H 48	I 49	J 4A	K 4B	L 4C	M 4D	N 4E	O 4F
80	P 50	Q 51	R 52	S 53	T 54	U 55	V 56	W 57	X 58	Y 59	Z 5A	[5B	\ 5C	] 5D	^ 5E	_ 5F
96	` 60	a 61	b 62	c 63	d 64	e 65	f 66	g 67	h 68	i 69	j 6A	k 6B	l 6C	m 6D	n 6E	o 6F
112	p 70	q 71	r 72	s 73	t 74	u 75	v 76	w 77	x 78	y 79	z 7A	{ 7B	| 7C	} 7D	~ 7E	□ 7F
128	€ 80	? 81	‚ 82	ƒ 83	„ 84	… 85	† 86	‡ 87	ˆ 88	‰ 89	Š 8A	‹ 8B	Œ 8C	? 8D	Ž 8E	? 8F
144	? 90	‘ 91	’ 92	“ 93	” 94	• 95	– 96	— 97	˜ 98	™ 99	š 9A	› 9B	œ 9C	? 9D	ž 9E	Ÿ 9F
160	A0	¡ A1	¢ A2	£ A3	¤ A4	¥ A5	¦ A6	§ A7	¨ A8	© A9	ª AA	« AB	¬ AC	AD	® AE	¯ AF
176	° B0	± B1	² B2	³ B3	´ B4	µ B5	¶ B6	· B7	¸ B8	¹ B9	º BA	» BB	¼ BC	½ BD	¾ BE	¿ BF
192	À C0	Á C1	Â C2	Ã C3	Ä C4	Å C5	Æ C6	Ç C7	È C8	É C9	Ê CA	Ë CB	Ì CC	Í CD	Î CE	Ï CF
208	Ð D0	Ñ D1	Ò D2	Ó D3	Ô D4	Õ D5	Ö D6	× D7	Ø D8	Ù D9	Ú DA	Û DB	Ü DC	Ý DD	Þ DE	ß DF
224	à E0	á E1	â E2	ã E3	ä E4	å E5	æ E6	ç E7	è E8	é E9	ê EA	ë EB	ì EC	í ED	î EE	ï EF
240	ð F0	ñ F1	ò F2	ó F3	ô F4	õ F5	ö F6	÷ F7	ø F8	ù F9	ú FA	û FB	ü FC	ý FD	þ FE	ÿ FF

CP1252 (Windows-1252)

Erläuterung: Die Kästchen in den ersten beiden Reihen (0-31 oder hexadezimal 00-1F) sind die eigentlichen Control-Chars, die zum größten Teil nicht darstellbar sind. Die Control-Chars, die zur Formatierung von Text verwendet werden (Zeilenumbruch, Tabulatoren, Seitenvorschub, ...) sind einfach leer. Die „Problemzone“ erstreckt sich von der 9.-10. Zeile (128-159 bzw. hexadezimal 80-9F). Wie man sieht, wurde sie durchaus sinnvoll gefüllt: Das Euro-Zeichen ist relativ spät dazugekommen, ansonsten finden wir noch die französischen OE-Ligaturen, und Konsonanten mit Caret. „Gänsefüßchen“ in doppelter und einfacher Ausfertigung, Trademark-Zeichen, und das Y mit Trema.

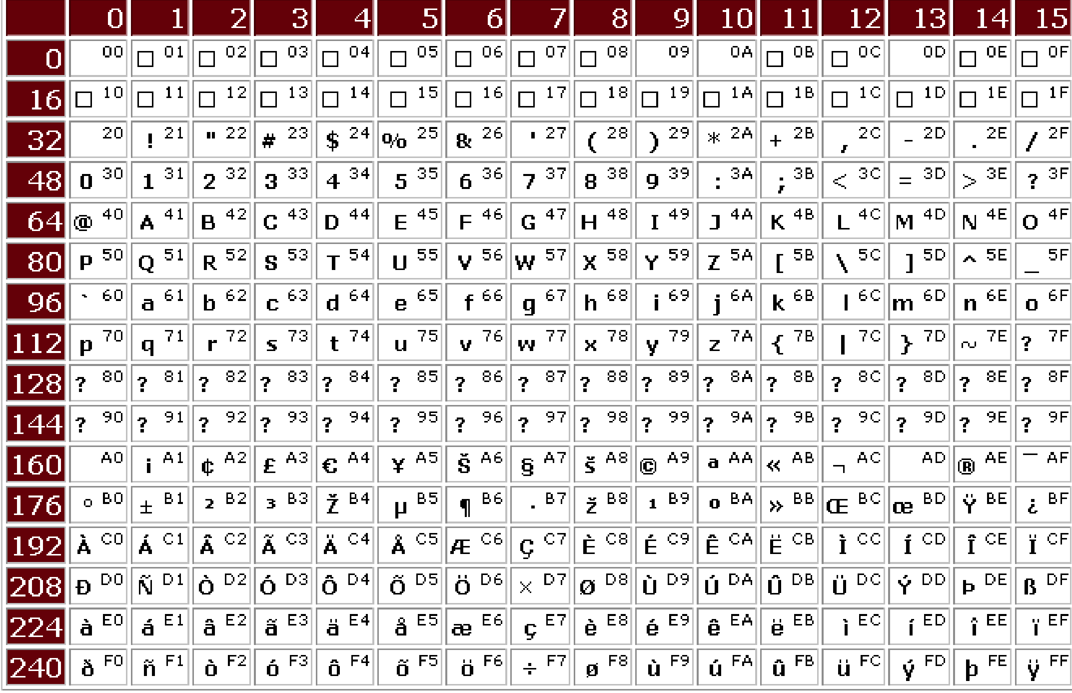
ISO-8859-1 ist absolut identisch mit Windows-1252 (das ist der gängige Alias für CP1252), nur sind eben die Zeichen 128-159 nicht definiert. Davor und dahinter sieht es jedoch völlig gleich aus.

3.3.2. ISO-8859-15 Revisited

Lange Zeit waren die Unterschiede zwischen ISO-8859-1 und Windows-1252 praktisch wenig relevant, schlicht und ergreifend, weil nicht ein Zeichen dabei war, das ohne größere Mausaktionen und Fingerverknotungen überhaupt in einen Text eingegeben werden konnte. Dann kamen die „intelligenten“ Textverarbeitungsprogramme, die z. B. in deutschen Umgebungen mit Heuristiken dafür sorgten, dass die Eingabe **`"Wörtliche Rede"`** vom Textverarbeitungsprogramm in „`Wörtliche Rede`“ gewandelt wurde (man vergleiche die "doppelten Anführungszeichen" mit den „deutschen Gänsefüßchen oben und unten“).

Die von Microsoft zusätzlich definierten Zeichen waren fast alle notwendig, das sah man auch im Unix-Lager ein, und spätestens mit dem Euro bestand akuter Handlungsbedarf, was die fehlenden Zeichen in ISO-8859-1 betraf. Statt aber einfach die Änderungen von Microsoft zu übernehmen (in den meisten Anwendungen hätte nicht eine Zeile

Code geändert werden müssen; es hätte lediglich eines systemweiten Updates der Zeichensätze bedurft), entschied man sich aus Halsstarrigkeit, Prinzipienreiterei, Besserwisserei, Trotz oder einem guten Grund, der mir persönlich nicht bekannt ist, dafür stattdessen ein neues Codeset ISO-8859-15 zu definieren:



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	00	□ 01	□ 02	□ 03	□ 04	□ 05	□ 06	□ 07	□ 08	09	0A	□ 0B	□ 0C	0D	□ 0E	□ 0F
16	□ 10	□ 11	□ 12	□ 13	□ 14	□ 15	□ 16	□ 17	□ 18	□ 19	□ 1A	□ 1B	□ 1C	□ 1D	□ 1E	□ 1F
32	20	! 21	" 22	# 23	$ 24	% 25	& 26	' 27	(28	) 29	* 2A	+ 2B	, 2C	- 2D	. 2E	/ 2F
48	0 30	1 31	2 32	3 33	4 34	5 35	6 36	7 37	8 38	9 39	: 3A	; 3B	< 3C	= 3D	> 3E	? 3F
64	@ 40	A 41	B 42	C 43	D 44	E 45	F 46	G 47	H 48	I 49	J 4A	K 4B	L 4C	M 4D	N 4E	O 4F
80	P 50	Q 51	R 52	S 53	T 54	U 55	V 56	W 57	X 58	Y 59	Z 5A	[5B	\ 5C	] 5D	^ 5E	_ 5F
96	` 60	a 61	b 62	c 63	d 64	e 65	f 66	g 67	h 68	i 69	j 6A	k 6B	l 6C	m 6D	n 6E	o 6F
112	p 70	q 71	r 72	s 73	t 74	u 75	v 76	w 77	x 78	y 79	z 7A	{ 7B	| 7C	} 7D	~ 7E	? 7F
128	? 80	? 81	? 82	? 83	? 84	? 85	? 86	? 87	? 88	? 89	? 8A	? 8B	? 8C	? 8D	? 8E	? 8F
144	? 90	? 91	? 92	? 93	? 94	? 95	? 96	? 97	? 98	? 99	? 9A	? 9B	? 9C	? 9D	? 9E	? 9F
160	A0	¡ A1	¢ A2	£ A3	€ A4	¥ A5	Š A6	§ A7	š A8	© A9	ª AA	« AB	¬ AC	AD	® AE	¯ AF
176	° B0	± B1	² B2	³ B3	Ž B4	µ B5	¶ B6	· B7	ž B8	¹ B9	º BA	» BB	Œ BC	œ BD	Ÿ BE	¿ BF
192	À C0	Á C1	Â C2	Ã C3	Ä C4	Å C5	Æ C6	Ç C7	È C8	É C9	Ê CA	Ë CB	Ì CC	Í CD	Î CE	Ï CF
208	Ð D0	Ñ D1	Ò D2	Ó D3	Ô D4	Õ D5	Ö D6	× D7	Ø D8	Ù D9	Ú DA	Û DB	Ü DC	Ý DD	Þ DE	ß DF
224	à E0	á E1	â E2	ã E3	ä E4	å E5	æ E6	ç E7	è E8	é E9	ê EA	ë EB	ì EC	í ED	î EE	ï EF
240	ð F0	ñ F1	ò F2	ó F3	ô F4	õ F5	ö F6	÷ F7	ø F8	ù F9	ú FA	û FB	ü FC	ý FD	þ FE	ÿ FF

ISO-8859-15

Na, endlich, der Euro war da (Nummer 164 bzw. Hex A4), auch die meisten anderen zusätzlichen Zeichen aus CP1252 wurden übernommen, aber der Bereich 128-159 wurde weiterhin freigehalten, und stattdessen wurden Zeichen aus ISO-8859-1 ersetzt. In der Theorie hatte das Unix-Lager also z. B. ein Euro-Zeichen, in der Praxis funktioniert es noch heute in den meisten (Unix-)Programmen nicht.

Der Grund dafür ist ganz einfach: ISO-8859-15 ist inkompatibel zu ISO-8859-1, und hat sich deshalb in der Praxis (noch) nicht durchgesetzt. Das wird wohl auch nie passieren, weil ISO-8859-15 auch inkompatibel zu CP1252 ist, und somit der vorher (eigentlich reibungslos funktionierende) Datenaustausch mit der Windows-Welt nicht mehr funktioniert: Vorher sahen Unix-Nutzer da, wo der Windows-Nutzer das Euro-Zeichen eingegeben hatte nur ein Fragezeichen, in die andere Richtung aber - von Unix zu Windows - wurde jedes Zeichen getreu der Intention des Schreibers wiedergegeben. Mit ISO-8859-15 wurde das Problem auf beide Richtungen ausgedehnt: Wo der Unix-Nutzer das neue Y mit Trema eingibt (Nummer 190, Hex BE) sieht der Windows-Nutzer „3/4", die OE-Ligaturen kommen als „1/4" bzw. „1/2" an. Das Euro-Zeichen aus der Unix-Welt mutiert doch wieder zum Currency-Symbol, die Windows-Gänsefüßchen bleiben Unix-Fragezeichen. Und innerhalb der Unix-Welt bestehen prinzipiell die gleichen Probleme, weil sich die Programme lange Zeit uneins bleiben werden, ob sie Zeichen Nummer 164/A4 als Euro- oder Currency-Symbol interpretieren sollen.

Hätte man stattdessen die Microsoft-Erweiterungen zu ISO-8859-1 übernommen, wäre es möglich gewesen, Programme schrittweise auf die neuen Codes umzustellen (sofern

das überhaupt notwendig gewesen wäre). Wie reibungslos das gehen könnte, sieht man an der Windows-Welt, in der die Euro-Einführung ohne größere Probleme vonstatten ging. Dass viele Drucker das Euro-Zeichen nicht kannten, führte nicht zu nennenswerten Irritationen, und unter Unix wird das selbst mit ISO-8859-15 noch lange Zeit üblich bleiben, weil es nicht ganz unwahrscheinlich ist, dass Dokumente, die das Euro-Zeichen verwenden, auf einem Windows-System erstellt wurden.

Fazit: Meiner Meinung nach ist ISO-8859-15 kompletter Unfug und landet hoffentlich bald auf dem Schrotthaufen der Computergeschichte.

3.3.3. Andere Sprachen, andere Windows-Codepages

Auch für nicht-westeuropäische Sprachen, die mit 8-Bit-Zeichen auskommen hat Microsoft durchweg eigene Standards definiert, oft nach der gleiche Linie wie bei ISO-8859-1/CP1252. Diese Codesets folgen durchweg dem gleichen Namensschema CPXXXX (X ist eine Ziffer von 0-9). Relativ weit verbreitet ist beispielsweise CP1251, auch als *Windows-Cyrillic* oder *Windows-1251* bekannt. CP1251 ist ein ernsthafter Konkurrent zu KOI8-R, weil die weite Verbreitung der Microsoft-Produkte die Verwendung des Codesets natürlich stark begünstigt. Dennoch ist KOI8-R für Kyrillisch, zumindest im World Wide Web, wohl noch immer erste Wahl.

Alle Windows-Codepages sind inkompatibel zu den entsprechenden Zeichensätzen aus ISO 8859, allerdings sind die spezifischen Kompatibilitäts-Probleme für andere Sprachen weit weniger ausgeprägt. ISO-8859-1 ist ein De-Facto-Standard; die meisten Applikationen gehen bei Text, der nicht ausdrücklich gekennzeichnet ist, davon aus, dass er in ISO-8859-1 bzw. Windows-1252 kodiert ist. Ist dies (ausnahmsweise) nicht der Fall, muss ohnehin eine Zusatzinformation mitgeliefert werden, die das verwendete Codeset spezifiziert. Mit dieser Zusatzinformation wiederum erfordert es aber nicht gerade Hexenwerk, die Texte vor der Darstellung von einem Codeset ins andere zu wandeln. Auch Unix-Applikationen sind in der Regel selbstverständlich in der Lage, in Windows-Zeichensätzen kodierte Dokumente korrekt zu interpretieren. Nur müssen sie dabei natürlich wissen, um welche Windows-Codepage es sich denn dabei handelt.

3.4. Weitere Hersteller

Nicht nur Microsoft benutzt proprietäre Codesets. Hewlett-Packard verwendet für sein Unix noch immer das - zu ISO-8859-1 völlig inkompatible - Codeset *HP-Roman8*. Der Macintosh scheint mit Max OS X standardmäßig auf ISO-8859-1/CP1252 umgeschwenkt zu sein (erlaubt aber auch noch die alten, proprietären Mac-Codesets), NextStep verwendet eigene Codesets, selbst IBM-Mainframes mit EBCDIC (das noch nicht einmal zu ASCII kompatibel ist) sind noch in Gebrauch. Alle diese Systeme sind allerdings nicht genügend verbreitet, um das Chaos noch nennenswert zu verschlimmern, und alle haben mittlerweile Methoden entwickelt, um den Datentransfer mit dem Rest der Welt über Standard-Codesets zu gewährleisten.

3.5. Weitere Sprachen

Es gibt natürlich noch eine Reihe weiterer Sprachen, die eine überschaubare Anzahl von

Schriftzeichen haben, und für die somit eigene ASCII-kompatible 8-Bit-Codesets definiert werden können. Um nicht mit ASCII in Konflikt zu kommen, ist eine Maximalzahl von 128 Zeichen möglich (die ersten 128 sind ja eben von ASCII belegt), aus ISO-8859-Sicht, maximal 96 (32 zusätzliche Control-Codes von 128-159).

Abgesehen von den diversen proprietären Codesets der Firmen Microsoft und Apple für Sprachen, die bereits behandelt wurden, sind noch einige weitere, teilweise durch Wildwuchs entstandene Codesets von Bedeutung.

3.5.1. Georgisch

Georgisch unterscheidet nicht zwischen Groß- und Kleinbuchstaben, und kommt mit 39 neuen Zeichen aus, die in eine ältere Version von CP1252 (ohne Euro, ohne Caret-Konsonanten) eingefügt wurden. Das Codeset ist als *Georgian-Academy* bekannt, meines Wissens allerdings nicht offiziell registriert.

3.5.2. Vietnamesisch

Vietnamesisch gehört wie Chinesisch, Japanisch, Koreanisch, etc. zu den *tonalen* Sprachen, Sprachen also, bei denen einsilbige Wortwurzeln (Töne) bedeutungstragend sind. Ursprünglich bediente sich das Vietnamesische der chinesischen Ideogramme. Im 13. Jahrhundert entstand daraus die *Nôm*-Schrift, die allerdings nur Intellektuellen verständlich war. Im 17. Jahrhundert entwickelten die christlichen Missionare das „Quôc-ngu“ („Landesschrift“), das auf dem lateinischen Alphabet aufbaut, und die Töne durch diakritische Zeichen kennzeichnet. Das Quôc-ngu löste offiziell 1910 die chinesische ideographische Schrift ab.

Vietnamesisch ist damit die einzige tonale Sprache, deren Schrift auf dem lateinischen Alphabet aufbaut (plus zwei spezielle Vokale O-Horn und U-Horn, O und U mit einem kleinen Horn rechts oben; weiterhin ein halbdurchgestrichenes D, das aus dem Polnischen bekannt ist). Es gibt allerdings insgesamt 134 Kombinationen aus diesen Grundbuchstaben und den zur tonalen Kennzeichnung dienenden diakritischen Zeichen, was bedeutet, das die von ASCII nicht definierte obere Hälfte des 8-Bit-Raumes nicht ausreichend für Vietnamesisch ist.

Die drei gebräuchlisten Codesets sind VSCII, VISCII, und VNCII (VPS). VSCII ist ein offizieller Standard, aber VISCII scheint am verbreitetsten zu sein. Alle drei Codesets haben die sechs überzähligen Zeichen im Bereich 0-31 für Control-Codes untergebracht, VSCII sogar deren zwölf, weil es auch Code-Positionen für die isolierten diakritischen Zeichen definiert.

Nach Informationen auf [Standards] sind mindestens sieben weitere Codesets in Verwendung, die allerdings nicht die Verbreitung von VSCII, VISCII und VNCII gefunden haben.

4. Multibyte-Codesets

Der Abschnitt 3.5.2, „Vietnamesisch“ ließ schon erahnen, dass viele Schriften niemals

mit den maximal 256 Zeichen, die der 8-Bit-Raum erlaubt, auskommen. Richtig haarig wird es bei den *CJK*-Sprachen Chinesisch, Japanisch und Koreanisch, die allesamt auf die chinesische Wortschrift, die sogenannten *Hanzi* (*Kanji* auf Japanisch, *Hanja* auf Koreanisch) zurückgehen. Wie der Name suggeriert, entstanden diese Ideogramme während der Han-Dynastie in China, und repräsentieren - im Gegensatz zur lateinischen Schrift - nicht die Lautung, sondern die Bedeutung, was z. B eine Japanerin in gewissem Maße in die Lage versetzt, auch chinesische Texte zu verstehen.

Chinesische Schriftzeichen und ihre in Japan und Korea verwendeten Derivate sind durchweg quadratisch und wurden ursprünglich von oben nach unten und rechts nach links gesetzt. Die heutige Praxis hat sich allerdings dem lateinischen Alphabet angepasst, es wird also von links nach rechts und oben nach unten geschrieben.

4.1. CJK-Sprachen

4.1.1. Chinesisch

In chinesischen Schulen differiert der Lehrplan für das erste Schuljahr daher signifikant von dem für deutsche Grundschulen vorgesehenen, was die Erschließung des Alphabets angeht. Etwas Ordnung in das Chaos kommt, wenn man in der Lage ist, die aus Grundwörtern gebildeten Kombinationen zu deuten: Wird das Ideogramm für die Sonne über dem Ideogramm für den Horizont angeordnet, entsteht das Schriftzeichen für „früh“, zwei ineinandergeschobene Bäume werden zum Wald, das gespiegelte Schriftzeichen für einen Beamten bedeutet Fürst.

Weder die im Jahre 1892 eingeführte chinesische Lautschrift (im Gegensatz zur von der Aussprache unabhängigen Wortschrift), noch die im Jahre 1926 versuchsweise eingeführte, auf lateinischen Buchstaben aufbauende Lautschrift vermochten sich durchzusetzen. Die chinesische KP ließ die Schriftzeichen zwar vereinfachen, hielt aber an der Wortschrift fest, um eine sprachliche Zersplitterung der Volksrepublik China zu vermeiden.

4.1.2. Japanisch

Die chinesische Schrift wurde vor ca. 2400 Jahren in Japan eingeführt. Als agglutinierende Sprache entzieht sich das Japanische jedoch einer adäquaten Darstellung mit chinesischen Ideogrammen. Daher wurden ca. 250 zusätzliche Kanji mit rein phonetischer Funktion, die sogenannten *Kana* eingeführt, die etwa im 9. Jahrhundert zu den noch heute gebräuchlichen *Katakana* weiterentwickelt wurde. Ungefähr zur gleichen Zeit entstanden aus häufig auftretenden Zeichenkombinationen die *Hiragana*, die zur Wiedergabe japanischer Silben dienten. Heute ist die japanische Schrift eine Mischschrift aus ca. 1850 Kanji, den Hiragana und den z. B. für Fremdwörter benutzten Katakana.

4.1.3. Koreanisch

Das Koreanische ist entfernt mit dem Japanischen verwandt, und bedient sich einer Silbenschrift, die durch wiederholte Vereinfachungen aus der chinesischen Schrift hervorgegangen ist.

4.2. Escaping

Angesichts der Vielzahl der darzustellenden Schriftzeichen erscheint die Idee, die ideographischen CJK-Schriften im 8-Bit-Raum darzustellen, natürlich völlig absurd. Scheinbar reicht ein Byte mit acht Bit also spätestens in Ostasien nicht mehr zur Darstellung eines Schriftzeichens aus. Man könnte natürlich einfach mehr Bits in ein Byte packen, um einen größeren Zahlenraum zu schaffen, oder die Eins-Zu-Eins-Beziehung zwischen Bytes und Schriftzeichen aufgeben. Allerdings ist genau diese Eins-Zu-Eins-Beziehung so stark in allen Computer-Betriebssystemen verwurzelt, dass eine generelle Änderung zu enormen Schwierigkeiten führen würde.

Im Abschnitt 2.2, „Von Bits und Bytes“ wurde ja schon beschrieben, wie man es trotz 8-Bit-Bytes hinbekommt, einen Computer mit Zahlen, die weitaus größer als 256 sind, rechnen zu lassen. Man muss lediglich mehrere Bytes (2, 4, 8, ...) zu logischen Einheiten zusammenfassen. Aber auch dieser Ansatz führt zu Schwierigkeiten: Wird ASCII-Text (oder auch Text mit beliebigen 8-Bit-Zeichen) von einem „neuen“ auf ein „altes“ System übertragen, kommt auf dem alten System ein Datenstrom an, der jeweils abwechselnd aus dem Zeichen mit der Nummer Null und dem eigentlichen Zeichen aufgebaut ist. Das Zeichen mit der Nummer Null hat allerdings auf praktisch allen Betriebssystemen, eine Sonderbedeutung, die der Programmiersprache C entstammt: Das sogenannte Null-Byte kennzeichnet das Ende einer Zeichenkette. Programme hören beim ersten Null-Byte also einfach auf, einen Text zu lesen.

Wir werden später sehen, dass dieser Ansatz trotzdem verfolgt wird, er hat aber seine spezifischen Schwierigkeiten. Es gibt aber eine weitere Möglichkeit, die 256-Zeichen-Grenze auch mit 8-Bit-Zeichen zu durchbrechen, das *Escaping*.

Nehmen wir an, wir hätten die Aufgabe nur mit den Buchstaben des lateinischen Alphabets die deutschen Umlaute Ä, Ö, Ü, ä, ö und ü sowie das ß darzustellen. Jeder kennt den Trick, man schreibt ae, oe, ue und ss, in Computer-Lingo: Wir kodieren bestimmte deutsche Schriftzeichen mit 2 Bytes. Nicht nur für Computer ist diese Codierung aber eigentlich unbrauchbar, denn wir können jetzt nicht mehr zwischen der zuweilen vorkommenden Buchstabenkombination ae und einem ä unterscheiden. Das Wort „Masse“ wäre nur noch aus dem Zusammenhang deutbar, denn es kann sowohl für die echte Masse, als auch für Maße stehen.

Es ist aber auch eine eindeutige Codierung möglich: In der deutschen Sprache taucht der Buchstabe q nur in der Kombination qu auf. Jedes andere Vorkommen eines q ist „illegal“. Eine „kugelsichere“ Codierung könnte also so aussehen: Ein ä wird als „qa“ codiert, ein Ä als „QA“, ö entsprechend als „qo“ und ü/Ü als „qy/QY“ (um die legale Kombination qu zu vermeiden). Aus dem Eszet könnte schließlich ein „qs“ werden, nicht sehr gut lesbar, aber eindeutig.

Nachteil dieser Methode wäre, dass wir keine Wörter aus anderen Sprachen darstellen können, die unsere Restriktion für die Verwendung des q nicht kennen. Das lässt sich aber leicht ausbügeln, indem man die Konvention erweitert: Ein „echtes“ q wird als „qq“ bzw. „QQ“ dargestellt.

Wenden wir die vollständige Regel auf den letzten Absatz an, würden wir folgendes se-

hen:

> Nachteil dieser Methode wqare, dass wir keine Wqorter aus anderen Sprachen darstellen kqonnen, die unsere Restriktion fqyr die Verwendung des qq nicht kennen. Das lqasst sich aber leicht ausbqygeln, indem man die Konvention erweitert: Ein „echtes“ qq wird als „qqqq“ bzw. „QQQQ“ dargestellt.

Lediglich die Darstellung der längeren „Q-Folgen“ erscheint etwas holprig. Ansonsten bleibt der Text aber durchaus lesbar, und selbst jemand, der unsere Konvention nicht kennt, wird das System spätestens beim zweiten Lesen verstanden haben.

Verallgemeinert lässt sich die Methode also so beschreiben: Man wählt ein (oder auch zwei) Escape-Zeichen aus, vorzugsweise eines, das in normalen Daten selten anzutreffen ist, legt eine Regel fest, dass die auf das Escape-Zeichen folgenden Zeichen eine Sonderbedeutung haben, und sorgt dafür, dass man das Escape-Zeichen noch immer darstellen kann, indem man es z. B. durch sich selber escapen lässt.

Der Trick ist ziemlich alt. In alten Drucker-Handbüchern findet man seitenweise solche Escape-Sequenzen. Wenn formatierte Textdateien ausgedruckt werden, müssen die Format-Informationen (Zeichensatz, fett, kursiv, unterstrichen, ...) ja irgendwie an den Drucker übermittelt werden. Traditionell schickte man dafür das Zeichen mit der Nummer 27 (im Bereich 0-31 für Control-Chars), gefolgt von Zeichenkombinationen, die eben im Druckerhandbuch dokumentiert waren. Dort konnte z. B. stehen, dass mit der Zeichenkombination „Zeichen Nummer 27 gefolgt von **`[31m`**“ die Schriftfarbe auf Rot gesetzt wird (bei Terminals ist das meistens sogar so: **`echo -e "\033[31m"`**).

Weil das Zeichen mit der Nummer 27 so gerne für diesen Zweck eingesetzt wird, heißt es auch *Escape* und liegt auf der Taste **Esc**. Man kann die Idee aber auch noch weiterführen, und neben dem Escape-Zeichen auch noch ein Zeichen definieren, das eindeutig das Ende der Escape-Sequenz kennzeichnet. Nimmt man als Escape-Zeichen das kaufmännische Und und als Ende-Markierung das Semikolon, hat man die HTML-Autoren geläufige XML-Entity-Darstellung, bei der ein ä zu `ä` wird, und (weil das Escape-Zeichen selber immer auch in das Escaping einbezogen werden muss) zwingendermaßen die Folge `&` für das eigentliche Kaufmanns-Und. Verwendet man als Escape-Zeichen „Kleiner-Als <“ und als Ende-Markierung „Größer-Als >“ hat man einen weiteren Weg gefunden, wie man beliebige Zusatzinformationen in einem Text unterbringen kann. Letztere Technik wird in SGML und daraus abgeleiteten Markup-Sprache wie HTML oder XML angewandt.

4.3. Multi-Byte-Encodings

Praktisch alle Kodierungen für Zeichen außerhalb des 7-Bit-Raums von ASCII bzw. des 8-Bit-Raums von Single-Byte-Codesets beruhen auf diesem Trick. Immer werden bestimmte Zeichenfolgen für die Erweiterung „missbraucht“, und die darauffolgenden Zeichen werden in einem anderen Kontext interpretiert. Der Standard ISO 8859 sieht z. B. einen Mechanismus vor, wie man durch Escape-Sequenzen aus dem 7-Bit-ASCII-Raum

auf die obere Hälfte des Codesets (das die Sonderzeichen enthält) um- und wieder zurückschalten kann. Dieser Vorgang wird auch *Shiften* genannt.

Die meisten Kodierungen für ideographische Schriften mit mehr als 255 Zeichen gehen ähnlich vor. Gängig ist eine Definition, in der alle ASCII-Zeichen für sich selber stehen, und alle 8-Bit-Zeichen (also in ISO-8859-1 die obere Hälfte) eine *Multi-Byte-Sequenz* einleiten. Eine fiktive Codierung sähe so aus: Die Zeichen 0-127 entsprechen exakt ASCII und werden *as is* transparent durchgereicht. Hat ein Byte einen Wert im Bereich 128 bis 255 wird ein weiteres Byte gelesen und der Zahlenwert beider Bytes multipliziert.

Wieviele Zeichen könnten wir damit darstellen. 128 Zeichen (0-128) sind schon in ASCII definiert. Das erste Multi-Byte-Zeichen ist Nummer 128. Wenn es von einem Null-Byte gefolgt ist, ergäbe sich nach der obigen Regel 128 x 0 = 0, also exakt ein Null-Byte. Folgt auf ein Byte mit dem Wert 128 eins mit dem Wert 25, wird damit Zeichen Nummer 3200 erzeugt. Maximal könnten zwei Bytes mit dem Wert 256 aufeinanderfolgen. Hier würde sich der Code aus (256 - 128) x 256 = 128,

Das Verfahren wäre einfach, hätte aber etliche Nachteile (in der Praxis geht man schlauer vor). Zunächst einmal ist die Codierung nicht eindeutig, ein und dasselbe Zeichen ließe sich auf mehrfache Art und Weise darstellen. Das erscheint auf den ersten Blick harmlos, wenn es aber darum geht, einen Text nach einer bestimmten Zeichenfolge zu durchsuchen, müsste nach sämtliche Bytefolgen gesucht werden, die für diese Zeichenfolge möglich sind. Das ist ein ziemlicher Nachteil.

Ein weiteres Problem ist die fehlende *Segregation* der Codierung. Stoßen wir in einem Datenstrom beispielsweise auf ein Byte mit dem Wert 65, lässt sich nicht festellen, ob es dem (ASCII-)Zeichen A entspricht, oder Teil einer Multi-Byte-Sequenz ist. Dieser Mangel ließe sich beheben, indem man fordert, dass auch das zweite Byte im Bereich 128-255 liegen muss. Dann könnte man bei jedem Byte an seinem Wert erkennen, ob es Teil einer Mulit-Byte-Sequenz ist, oder aber ein ASCII-Zeichen.

Weiterhin fehlt es an einer *Synchronisation*. Gesetzt den Fall, wir hätten die fehlende Segregation wie beschrieben behoben, lägen alle in Multi-Byte-Sequenzen erlaubten Bytes im Bereich 128-255. Wollen wir allerdings bei einem solchen Byte feststellen, wo der Anfang, und das Ende des Multi-Byte-Zeichens ist, stehen wir auf dem Schlauch. Wir müssen von Anfang an nachzählen. Beispiel:

```
Am Anfang ASCII, aber dann ÄÖéä§ßßüÖÖüßß§°µäü und
weiter im Text
```

Was ist jetzt ein Zwei-Byte-Zeichen? Die Kombination aus Grad und My, oder My und ä? In der Praxis ist das wichtig, denn dort kommen solche Situationen, in denen man einen Bytestrom verarbeiten muss (und nicht an den Anfang zurückkehren kann, um nachzuzählen) häufig vor.

Die Codierung hat aber auch einen Vorteil: ASCII-Zeichen werden transparent durchgereicht. Das ist einerseits für die Lesbarkeit wichtig, wenn ein Text vergleichsweise wenig Sonderzeichen (im Verhältnis zu ASCII) enthält. Liest man auf einer schlecht konfigu-

rierten Web-Site das Wort „MÃ¤rchen“ (das Beispiel ist UTF-8, das fälschlicherweise als ISO-8859-1 interpretiert wird), kann man trotzdem noch erkennen, dass von einem Märchen und nicht einem Mädchen die Rede ist. Noch wichtiger wird die ASCII-Transparenz, wenn Texte von Maschinen und nicht von Menschen interpretiert wird. Maschinen/Programme springen meist auf bestimmte Schlüsselwörter an, und sehr häufig sind diese Schlüsselwörter auf die Verwendung von ASCII-Zeichen beschränkt. Werden ASCII-Zeichen von der Codierung nicht angetastet, funktionieren diese Programme weiterhin. Weitergedacht ist dies auch ein starkes Argument für die Forderung, dass Multi-Byte-Codes grundsätzlich keine ASCII-Zeichen enthalten sollten, um eine zufällige Fehlinterpretation auszuschließen.

Beherzigen wir diesen Ratschlag bei unserer Codierung, fordern wir also, dass das zweite Byte unserer Multi-Byte-Sequenzen ebenfalls immer aus dem Bereich 128-255 stammen muss (damit „opfern“ wir natürlich die Hälfte des theoretisch möglichen Wertebereichs), ist unsere Codierung auch *file-system-safe*. Was verbirgt sich dahinter? Hätten wir definiert, dass beispielsweise die Kombination „ä/“ für ein japanisches Hiragana stehen soll, könnten wir die Codierung nicht für die Benennung von Dateien verwenden, denn unser Dateisystem kennt unsere Codierung höchstwahrscheinlich nicht, und wird sich am Slash (/) im Dateinamen stören.

4.4. Bekannte Multi-Byte-Encodings

Gerade für den asiatischen Raum werden bereits seit langem Multi-Byte-Encodings verwendet. Diese Codierungen sind meist erheblich intelligenter als die oben beschriebene Ad-Hoc-Idee, und sind auch in Unicode-Zeiten noch sehr verbreitet. Die perfekte Codierung gibt es nicht, und so wundert es nicht, dass etliche zueinander inkompatible Kodierungen in Gebrauch sind.

Noch nicht einmal offiziell standardisiert, trotzdem weit verbreitet ist beispielsweise *Big5*, meist als traditionelles Chinesisch bezeichnet. Vereinfachtes (*simplified*) Chinesisch ist unter dem Namen *GB2312* bekannt. Für Koreanisch ist eine Codierung unter dem Namen JOHAB verbreitet, für Japanisch wird oft das von Microsoft entworfene *Shift-JIS* verwendet.

Die sehr alten EUC-Codierungen existieren in Variationen für verschiedene Länder/ Sprachen unter den Namen *EUC-JP, EUC-CN, EUC-TW, EUC-KR*.

Perfekt wird das Chaos dadurch gemacht, dass für fast alle Codierungen noch etliche Alias-Namen definiert sind, *EUC-CN, CN-GB, csGB2312, EUCCN, EUC_CN, GB2312* bezeichnen alle ein und das selbe Codeset.

Und noch eine schlechte Nachricht: *Imperia* ist unicode-fähig, die oben beschriebenen Multi-Byte-Codesets sind aber eben nicht Unicode. Zur Zeit (April 2003) ist Imperia nicht in der Lage, diese Codesets korrekt zu verarbeiten (genauer gesagt: Imperia ist nicht in der Lage, zwischen solchen Codesest zu konvertieren). Um alle Imperia-Features für ostasiatische Sprachen auszunutzen, *muss* zur Zeit Unicode verwendet werden. Allerdings existieren etliche Tools, die Unicode in jedes noch so exotische Codeset konvertieren, so dass diese Einschränkung in der Praxis wenig Relevanz besitzt.

5. Zusammenfassung

Computer speichern Texte als Zahlenfolgen ab. Die Zuordnung von Zahlen zu Schriftzeichen wird als *Codeset* oder auch *Charset* bezeichnet, und normalerweise (8-Bit-Raum) liegen diese Zahlen im Bereich 0-255. Der allgemein anerkannte Standard ASCII legt die Zuordnung der Zeichen 0-127 fest und definiert Schriftzeichen, die zur Darstellung englischer Texte ausreichend ist. Für den Bereich der Zeichen 128-255 existiert eine Vielzahl von Standards, von den ISO-8859-1, das als *Superset* von ASCII zusätzlich die zur Darstellung von Texten in vielen westeuropäischen Sprachen erforderlichen Zeichen definiert, der wichtigste ist. Daneben existieren aber weitere Standards für spezielle Sprachen und spezielle Betriebssysteme.

Für Sprachen, die mehr als 256 Schriftzeichen benötigen, werden Nicht-ASCII-Schriftzeichen normalerweise in *Multi-Byte-Sequenzen* gespeichert. Ein konkretes Schriftzeichen wird also als Kombination von zwei oder mehr Bytes definiert.

Kapitel 2. Die schöne Welt von Unicode

Der Schuldige für die babylonische Codeset-Verwirrung im Computerbereich ist leicht ausgemacht: Es ist die historisch bedingte Konvention, dass Computer Schriftzeichen intern als ein einziges Byte repräsentieren. Diese Konvention impliziert eine Beschränkung des Wertebereichs für Zeichencodes auf die Spanne 0-255, mit dieser Beschränkung lassen sich viele Sprachen dieser Welt nicht adäquat repräsentieren, und an eine parallele Verwendung mehrerer Sprachen innerhalb eines einzigen Dokumentes ist überhaupt nicht zu denken.

1. Was ist Unicode?

Die Lösung des Problems liegt auf der Hand: Der Wertebereich muss vergrößert werden. Und nichts anderes macht Unicode: Genau wie ASCII eine Standardzuordnung für die Codes mit den Nummern 0-127 vorgenommen hat, definiert Unicode eine weitere Standardzuordnung. Dem Unicode-Standard liegt dabei das ehrgeizige Ziel zugrunde, eine Standardzuordnung für *jedes* Schriftzeichen der Erde zu schaffen (für Schriften aus dem extraterristrischen siehe den Anhang A, *Klingonisch (tlhIngan Hol)*), nicht nur für echte Schriftzeichen, sondern beispielsweise auch für technische Symbole, musikalische Zeichen, Lautschrift, und viele weitere Graphiken. Es dürfte klar sein, dass der Unicode-Standard deshalb permanenten Änderungen unterworfen ist, weil auch permanent Vorschläge für die Aufnahme neuer Zeichen gemacht werden. Allerdings werden Zeichen in aller Regel *zugefügt*, Neu-Zuordnungen werden strikt vermieden, wodurch eine Abwärtskompatibilität des Standards sichergestellt ist.

Unicode ist übrigens auch als offizieller Standard ISO/IEC 10646 bekannt. Der Unicode-Standard wird vom *Unicode Consortium*, dem Zusammenschluss vieler führender Softwarehersteller und öffentlicher Einrichtungen in einer gemeinnützigen Organisation, gepflegt und weiterentwickelt. Zur Zeit aktuell ist [Unicode 3.0]; [Unicode 4.0] ist auf dem Weg und wird für den September 2003 erwartet.

Weitere wissenswerte Informationen können dem Artikel *Was ist Unicode?* auf [www.unicode.org] entnommen werden.

2. Der Wertebereich von Unicode

Wieviele Zeichen lassen sich mit Unicode repräsentieren? In [Unicode 3.0] sind 49.149 Zeichen enthalten, und ein Unicode-Zeichen hat eine Größe von 16 Bits (im Gegensatz zu 7 Bits für ASCII-Zeichen, oder 8 Bits für Zeichensätze wie ISO-8859-1). 16 Bits oder zwei Bytes bedeutet, dass theoretisch die Codes 0-65.535 (entspricht 65.536 Zeichen) verwendet werden können (s. Abschnitt 2.2, „Von Bits und Bytes“). 6.400 Codes sind für private Zwecke reserviert, 2 Codes sind illegal. Der *UTF-16 Erweiterungs-Mechanismus* (die sogenannten *Surrogate*) erlaubt die Definition weiterer 917.476 „offizieller“ und 131.068 privater Codepunkte.

Wieviele Zeichen ließen sich also theoretisch mit Unicode repräsentieren? Begnügen wir uns mit der Antwort: Sehr viele. Die Frage ist tatsächlich schwer zu beantworten, weil Unicode auch Zeichenkombinationen vorsieht, Möglichkeiten, die Schreibrichtung mitten im Text zu ändern, etc.

3. Unicode-Properties

Texte (nicht nur auf dem Computer) bestehen nicht nur aus einer Aneinanderreihung von Buchstaben, sondern aus einer Vielzahl von Schriftzeichen: Buchtstaben, Ziffern, Interpunktionszeichen, Steuerzeichen (Zeilenumbrüche, Tabulatoren, ...), Sonderzeichen (Prozent, Gradzeichen, Dollarzeichen), Symbolen, etc.

Dies ist in der Praxis enorm wichtig. Wird zum Beispiel ein Text für eine spätere Volltextrecherche indiziert, ist es sicherlich sinnvoll, Interpunktions- und Steuerzeichen herauszufiltern. Erwartet eine Anwendung in einer Eingabe Zahlen, muss bekannt sein, ob die eingegebenen Zeichen Ziffern repräsentieren.

Der Unicode-Standard trägt dem Rechnung, indem für jedes im Standard enthaltene Zeichen bestimmte Meta-Informationen, also Eigenschaften (*Properties*) gepflegt werden, die von einer Anwendung abgefragt werden können. Es lässt sich daher ermitteln, ob es sich bei einem bestimmten Unicode-Zeichen um einen Buchstaben, eine Zahl, ein Symbol, ein Interpunktionszeichen, ein Trennzeichen, eine Markierung oder ein Steuerzeichen handelt (jede dieser Kategorien hat zahlreiche Unterkategorien wie zum Beispiel Groß- und Kleinbuchstaben bei Buchstaben). Auch die Schreibrichtung - Arabisch zum Beispiel wird ja von rechts nach links geschrieben - für ein Zeichen ist abrufbar, genauso wie sich die Schriftfamilie (Lateinisch, Kyrillisch, Thailändisch, etc.) ermitteln lässt.

4. Technische Repräsentation von Unicode

Der Unicode-Standard definiert lediglich Standard-Codes für *Zeichen*, nicht konkrete - vom verwendeten Zeichensatz wie Times, Helvetica oder Courier abhängige Darstellungen. Genausowenig schreibt der Unicode-Standard vor, wie die Zeichencodes intern von einem Computer gespeichert oder dargestellt werden müssen.

4.1. Wide Characters

Unicode-Zeichen sind 16 Bits groß, die natürliche Repräsentation eines Unicode-Zeichens hat demnach auch 16 Bits oder 2 Bytes, und die meisten Systeme definieren heute deshalb einen Datentyp *wide character*, im Gegensatz zum normalen *character* (oft auch zum *char* verkürzt), der mindestens 16 Bit groß ist. Wohlgemerkt, *mindestens* 16 Bits, nach oben hin sind vom Standard für die Repräsentation keine Grenzen gesetzt, und deshalb sind system- oder codierungsabhängig auch 32-Bit (4 Byte-)Typen gebräuchlich.

4.1.1. Probleme mit Wide Characters

Ein Schriftzeichen wird nicht mehr durch ein Byte, sondern durch zwei Bytes dargestellt, und alle Probleme sind gelöst. Oder doch nicht? In der Praxis wirft diese Umstellung erhebliche Probleme auf, neben denen die Euro-Einführung oder die Y2K-Problematik winzig anmuten.

4.1.1.1. Auf- und Abwärtskompatibilität

Nehmen wir als Beispiel einen aus dem Zusammenhang gerissenen winzigen Textausschnitt, die Zeichenkette „schreibt über", die von einem Programm verarbeitet werden soll. Der (Klein-)Buchstabe „s" hat den ASCII-Code 115, das kleine „c" hat die Nummer 99, der erste Stolperstein erwartet uns beim kleinen „ü", von dem wir der Einfachheit halber annehmen, es sei in ISO-8859-1 codiert, habe also die Nummer 252. Insgesamt speichert der Computer die Zeichenkette als Nummernfolge 115, 99, 104, 114, 101, 105, 98, 116, 32, 252, 98, 101, 114. Ist das Programm in C geschrieben, können wir sogar noch davon ausgehen, dass das Ende der Zeichenkette durch ein Pseudo-Zeichen mit der Nummer 0 gekennzeichnet ist.

Die nächsten Überlegungen sind einfacher anzustellen, wenn wir die Nummernfolge nicht mit Dezimalzahlen (also Ziffern von 0-9), sondern mit *Hexadezimalzahlen* (Ziffern von 0-9 und a-f) darstellen. Es ist nicht notwendig zu verstehen, wie Hexadezimalzahlen interpretiert werden, es reicht völlig aus, zu wissen, dass sie eine andere Darstellung für exakt die gleichen Zahlen sind. In Hex sieht unsere Zeichenfolge so aus:

```
 s  c  h  r  e  i  b  t     ü  b  e  r
73 63 68 72 65 69 62 74 20 fc 62 65 72
```

Jede Zweiergruppe von (Hex-)Ziffern entspricht einem Buchstaben (das Leerzeichen *Space* hat den Code 30, bzw. 20 als Hexzahl).

Jetzt stellen wir unser System auf Unicode-Wide-Characters um, verbraten pro Zeichen also 2 Bytes. Wenn wir annehmen, dass unser Text in einer Datei auf der Festplatte gespeichert ist, wird diese Datei doppelt so groß, denn pro Zeichen werden jetzt doppelt so viele Bits bzw. Bytes verwendet. Leider weiß unser Programm nichts von dieser Umstellung (es muss noch umprogrammiert werden), und wird beim nächsten Leseversuch folgendes sehen:

```
   s     c     h     r     e     i     b     t           ü     b
00 73 00 63 00 68 00 72 00 65 00 69 00 62 00 74 00 20 00 fc 00 62 00
```

Die Zeichen wurden doppelt so groß, aus Sicht unseres noch an Bytes gewöhnten Programms wurden sie also mit Nullen aufgefüllt, obwohl noch immer der gleiche Text gemeint ist. Computerprogramme sind dumm, unser Programm macht da keine Ausnahme und wird diese Änderung einfach hinnehmen. Aber was sehen wir, wenn das Pro-

gramm den Text ausgeben soll? Das hängt von Programminterna ab. Eine Möglichkeit lautet: *Nichts*! Weshalb? Ein Null-Byte ist in der am weitesten verbreiteten Programmiersprache, der Spache *C* mit der speziellen Bedeutung *End Of String*, also Ende der Zeichenkette belegt. Unglücklicherweise fängt unsere Unicode-Zeichenkette fast zwangsläufig schon mit genau diesem Zeichen an, und konsequenterweise gibt unser Programm jetzt eine leere Zeichenkette aus. Dieser Fall (wir sehen nichts) dürfte der häufigste sein, wenn zwei nicht-unicodefähige Programme wide characters austauschen.

Weitaus weniger wahrscheinlich ist, dass das Programm einfach über die nicht darstellbaren Null-Bytes hinweggeht, und alles andere ausgibt. Damit hätten wir erstmal keine Probleme, aber welche Länge wird unser Programm wohl für die Zeichenkette annehmen? 13 oder 26 Zeichen? Und wer weiß, an welchen anderen Stellen die Null-Bytes doch noch für Verwirrung sorgen, und Schaden anrichten?

Der häufigste Fall, wenn die Daten von einem Datenträger gelesen werden (dann gilt die Konvention mit dem Null-Byte nämlich nicht), dürfte der sein, dass das Programm die nicht-darstellbaren Null-Bytes Ersatzzeichen einfügt, wir sehen etwas wie „?s?c?h?r?e?i?b?t? ?ü?b?e?r“.

Wem das noch nicht erschreckend genug erscheint, möge sich vorstellen, dass wir Unicode jetzt richtig ausreizen wollen, und unseren Text endlich etwas weiterschreiben können: „schreibt über Anna Karenina“, wobei wir allerdings den Titel Tolstois Roman im russischen Original darstellen *wollen*:

```
schreibt über Анна Каренина
```

Schön wär's. Wahrscheinlicher wird uns unser Programm aber Zeichenmüll in der Art von „?s?c?h?r?e?i?b?t? ?ü?b?e?r^D?^D=^D=^Do^D ^D?^Do^D@^D5^D=^D8^D=^Do“ präsentieren. Der deutsche Part ist noch einigermaßen zu entziffern, der kyrillische Part ist ein Zeichensalat aus willkürlich anmutenden Schriftzeichen, die sich mit Steuerzeichen (meistens das ASCII-Zeichen mit der Nummer 4 *End of Transmission - EOT*, der Einfachheit halber durch die Zeichenfolge „^D“ ersetzt) abwechseln.

Alles nicht wünschenswert. Also werden wir zusehen, dass wir unsere Programme möglichst schnell auf Unicode umstellen, dass sie also intern alle Textdaten als wide characters erwarten. Versäumen wir es jedoch, den Rest der Welt von der Löblichkeit unseres Vorhabens zu überzeugen, wird eines Tages dennoch eine alte Datei mit 8-Bit-Zeichen den Weg in unser kleines Refugium finden. Bleiben wir bei unserem Beispiel, wir lesen die ursprüngliche, noch nicht nach Unicode konvertierte Datei:

```
 s  c  h  r  e  i  b  t     ü  b  e  r
73 63 68 72 65 69 62 74 20 fc 62 65 72
```

So war es jedenfalls einmal gemeint. Tatsächlich erwartet unser Programm aber, dass jedes Zeichen in zwei Bytes kodiert ist. Der erste Stein rollt uns jetzt in den Weg, weil wir eine ungerade Anzahl von Bytes einlesen, was nie passieren könnte, wenn wir zwei

Bytes pro Zeichen verwenden. Wenn wir Pech haben (das hängt vom Talent des Programmiers ab) fliegt uns unser Programm mit Karacho um die Ohren, wenn wir Glück haben, ignoriert das Programm das überflüssige Byte am Ende oder wird wieder stillschweigend ein Ersatzzeichen darstellen, und die (einzelnen) Bytes werden jeweils zu wide characters (doppelten Bytes) zusammengefasst. Das Programm liest:

```
 s c  h r  e i  b t   ü  b e
7363 6872 6569 6274 20fc 6265
```

Hex-Zahlen haben die angenehme Eigenschaft, dass für ein Byte nie mehr als zwei „Ziffern" gebraucht werden, und man die Bytes einfach nur zusammenschieben muss, um zu Multi-Byte-Darstellungen zu gelangen. Treudoof als Unicode interpretiert sähe unser (deutscher!) Text jetzt aber so aus:

獣桲?扴?扥

Die Bytefolge „schreibt übe" naiv als Unicode interpretiert

Nein, das ist nicht unser deutscher Text in japanischer Lautschrift. Noch weniger handelt es sich um eine computergestützte Übersetzung von deutsch auf japanisch. Dass hier ostasiatische Zeichen dargestellt werden, ist reiner Zufall, weil sie gerade im Bereich der falsch interpretierten Daten liegen. Tatsächlich steht hier Nonsens (hoffe ich doch jedenfalls).

All dies könnte man jedoch als Übergangsschwierigkeiten betrachten. Nach einer gewissen Zeit könnten sämtliche Softwareprogramme an die neue 16-Bit-Konvention angepasst werden, und wir wären da, wo wir hinwollen. Selbst dann wäre diese Konvention aber noch mit erheblichen Nachteilen belastet.

4.1.1.2. Speicherverbrauch

Auch jenseits allen westeuropäisch-amerikanischem Chauvinismus', ist die Tatsache nicht zu verleugnen, dass ein ganz großer Teil der (Text-)Dateien, die von Computern verarbeitet und zwischen ihnen ausgetauscht werden, mit dem 8-Bit-Codeset ISO-8859-1 (bzw. Windows-1252) kodiert ist, und diese Codierung auch für eine adäquate Darstellung geeignet ist. Viele weitere Sprachen kommen ebenfalls prinzipiell mit einem 8-Bit-Zeichensatz aus. In einer reinen Unicode-Welt, würden diese Daten plötzlich alle auf die doppelte Größe aufgebläht, würden doppelt so viel Platz auf Festplatten und im Speicher verbrauchen, würden bei der Übertragung im Netzwerk die doppelte Bandbreite verschlingen und würden auch (mindestens!) die doppelte Zeit für ihre Verarbeitung benötigen. Neben den Kompatibilitätsschwierigkeiten ist dieser Aspekt sicher der wichtigste Grund, der einer Migration von 8 nach 16 Bit für Textdaten entgegensteht.

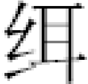
4.1.1.3. Synchronisation

Theoretisch ließen Computer sich vollständig auf die neue 16-Bit-Konvention umstellen. Praktisch wird dies unmöglich sein. Bei Netzwerkprotokollen interessiert es beispiels-

weise nicht wirklich, ob es sich bei den übertragenen Daten um Text oder Bilder, Sound, Video, Programmdateien oder was auch immer handelt. Die Daten werden weiterhin als Byteströme interpretiert werden. Das gleiche gilt, wenn Daten aus Gerätedateien (Festplatten, Sound-/Grafikkarten, etc.) gelesen werden. Auch hier wird sich in absehbarer Zeit nichts ändern, und diese Vorgänge werden weiterhin byteorientiert stattfinden.

Bei Daten*strömen* kommt es jedoch nicht selten vor, dass der Lesevorgang irgendwo, mitten im Datenstrom beginnt oder endet. Entsprechen die Bytegrenzen gleichzeitig Zeichengrenzen ist dies kein Problem. Bei wide characters, also 2-Byte-Zeichen wird das Lesen solcher Datenströme zur Herausforderung, denn die lesende Applikation kann immer nur Paare von Bytes sinnvoll interpretieren. Kann nicht ermittelt werden, ob das erste Byte eine gerade oder ungerade *Hausnummer* hat, können die Daten auf zwei unterschiedliche Arten interpretiert werden, mit anderen Worten: Es kann nicht ermittelt werden, ob das erste Byte im Datenstrom das vordere oder hintere Byte eines wide characters ist.

4.1.1.4. File-System-Safety



Diesem CJK-Silbenzeichen wurde im Unicode-Standard die Nummer 17.210 (hexadezimal 0x433a) zugewiesen. Darüber werden wir uns nicht wenig freuen, bis wir auf die Idee kommen, einer Datei auf einem Windows-Rechner einen Namen zu geben, der mit diesem Zeichen beginnt. Zerlegen wir die Zahl 17.210 nämlich in zwei einzelne Bytes, erhalten wir 67 und 58, und das sind die ASCII-Codes für die Zeichen „C" und „:". Es bleibt eine Übungsaufgabe für die Leserin, die Unicode-Zeichenfolge zu bestimme, die (fälschlicherweise) als ASCII interpretiert `C:\WINNT\system\very_important.dll` ergibt.

Dies ist natürlich kein Windows-Problem. Alle Dateisysteme haben „verbotene" Zeichen in Datei- und Verzeichnisnamen. Solange die Dateisysteme aber nicht unicodefähig sind, kann es zu bösen Überraschungen kommen, wenn eine Unicode-Byte-Folge zufällig solche verbotenen Zeichen enthält.

4.1.1.5. Was haben Eier mit Bytes zu tun?

Lemuel Gulliver, zunächst Schiffsarzt dann Kapitän auf einigen Schiffen, wurde folgendes auf seiner Reise in das Land Lilliput kolportiert (Quelle [Gulliver's Travels]):

> *It began upon the following Occasion. It is allowed on all Hands, that the primitive way of breaking Eggs, before we eat them, was upon the larger End: But his present Majesty's Grand-father, while he was a Boy, going to eat an Egg, and breaking it according to the ancient Practice, happened to cut one of his Fingers. Whereupon the Emperor his Father published an Edict, commanding all his Subjects, upon great Penaltys, to break the smaller End of their Eggs. The People so highly resented this Law, that our Histories tell us there have been six Rebellions raised on that account; wherein one Emperor lost his Life, and another his Crown. These civil Commotions were*

> *constantly fomented by the Monarchs of Blefuscu; and when they were quelled, the Exiles always fled for Refuge to that Empire. It is computed, that eleven thousand Persons have, at several times, suffered Death, rather than submit to break their Eggs at the smaller End. Many hundred large Volumes have been published upon this Controversy: But the books of the Big-Endians have been long forbidden, and the whole Party rendered incapable by Law of holding Employments. During the Course of these Troubles, the Emperors of Blefuscu did frequently expostulate by their Ambassadors, accusing us of making a Schism in Religion, by offending against a fundamental Doctrine of our great Prophet Lustrog, in the fifty-fourth Chapter of the Brundrecal (which is their Alcoran.) This, however, is thought to be a meer Strain upon the Text: For the Words are these: That all true Believers shall break their Eggs at the convenient End: and which is the convenient End, seems, in my humble Opinion, to be left to every Man's Conscience, or at least in the power of the Chief Magistrate to determine. Now the Big-Endian Exiles have found so much Credit in the Emperor of Blefuscu's Court, and so much private Assistance and Encouragement from their Party here at home, that a bloody War has been carried on between the two Empires for six and thirty Moons with various Success; during which time we have lost forty Capital Ships, and a much greater number of smaller Vessels, together with thirty thousand of our best Seamen and Soldiers; and the Damage received by the Enemy is reckon'd to be somewhat greater than Ours.*

Gut zwei Jahrhunderte später brach ein ähnlich heftiger Glaubenskrieg um die Frage aus, wie Zahlen größer als 255 (also Zahlen, zu deren Darstellung man mehr als ein Byte braucht) intern darzustellen seien. Aus Abschnitt 2.2, „Von Bits und Bytes" erinnern wir uns, dass der Trick prinzipiell darin besteht, diese großen Zahlen in Kombinationen von zwei, vier oder sogar acht Bytes darzustellen. So wie es nur die Ziffern 0-9 gibt, können wir dennoch Zahlen beliebiger Größe mit diesen Ziffern darstellen, wenn wir nur einfach genügend Ziffern aneinanderreihen. Die Zahl zwölf wird als Kombination aus eins und zwei dargestellt, wobei wir die erste Ziffer einfach mit zehn multiplizieren (12 = 1 x 10 + 2). Bei den Bits und Bytes gilt das gleiche, nur dass wir hier jeweils mit 256 multiplizieren müssen. Wenn wir die beiden Bytes 67 (binär 01000011) und 58 (binär 00111010) aneinanderreihen, erhalten wir also die Zahl 17.210 = 67 x 256 + 58. Oder - um bei den uns vertrauten Glühbirnen zu bleiben:

Die Zahl 17.210 in Binärdarstellung

Das scheint doch sehr klar. Oder doch nicht? Wie wäre es, wenn wir die beiden Bytes nicht von links nach rechts, sondern von rechts nach links anordnen?

17.210 binär, aber verkehrt herum?

Das erscheint zunächst ziemlich abwegig, aber es gab Hardwarehersteller, die genau diesen Weg gingen, bei Byte-Folgen also das weniger signifikante Byte (*Least Significant*

Byte - LSB) vor das *Most Significant Byte* (*MSB*) setzten. Nein, diese Hardwarehersteller kamen nicht aus der arabischen Welt, sondern aus dem Silicon Valley in Kalifornien. Und, nein, LSB-Prozessoren sind keine exotische Ausnahme; die Intel- (oder Intel-kompatiblen) Chips, die sich in gängigen PCs finden, gehören allesamt zur LSB-Fraktion, würden die Zahl zwölf - um bei der Analogie zu bleiben - also als „21“ darstellen.

Die prominentesten Vertreter des anderen, des MSB-Lagers, sind die Prozessoren der Firmen Sun Microsystems und Motorola, sowie die meisten Prozessoren der *Risc*-Architecktur, und weil die Standpunkte der Verfechter der einen und der anderen Fraktion ähnlich unvereinbar, und ihre Auseinandersetzungen ähnlich unversöhnlich wie die der Big- und Little-Endians in Lilliput waren, bürgerte es sich ein, die beiden *Byte-Sex*-(sic! Auch dies ist ein Terminus Technicus!) Varianten als *big-endian* (MSB) und *little-endian* (LSB) zu bezeichnen. In der Praxis muss also bei jeder Zahl über 255 dazugesagt werden, welcher Konvention folgend sie denn abgespeichert ist, was einigen Rechenaufwand verursachen kann. In praktisch allen Netzwerk-Protokollen wird eine Big-Endian-Darstellung postuliert, was dazu führt, dass Little-Endian-Intel-Boxen alle Zahlen intern an den Schnittstellen zum Netzwerk drehen müssen, selbst, wenn sie unter sich bleiben.

Einigermaßen unangenehm ist es, wenn der Byte-Sex (weniger zweideutig auch als *Endianness* bezeichnet) von Daten nicht bekannt ist, weil dies natürlich, zu einer Ambivalenz bei der Interpretation der Daten führt. Speichern wir einen Text mit Wide Characters auf einem Intel-PC ab, und übertragen ihn auf einen Sparc-Server der Firma Sun, haben wir ein handfestes Problem, weil die beiden Rechner unterschiedliche Vorstellungen über die Interpretation der Daten haben.

4.1.2. UTF-16

Allen Schwierigkeiten zum Trotz hat die Darstellung von Unicode-Zeichen als 2-Byte-Folgen eine relativ große Verbreitung gefunden, nicht zuletzt, weil dies die native Darstellung textueller Daten in der Programmiersprache *Java* ist. Folge dieser Designentscheidung in Java ist ein erhöhter Speicherverbrauch, andere negative Effekte sind weniger relevant, solange es sich um ein geschlossenes System handelt, bei dem die Interpretation von Daten (Bytes) eindeutig festgelegt ist.

Tatsächlich gibt es aber zwei (weitere) Varianten von UTF-16 (16-Bit *Unicode Transformation Format*), nämlich eine Big-Endian- und eine Little-Endian-Variante. In Abwesenheit anderer Anhaltspunkte sollte von Big-Endianness ausgegangen werden. Ein konkreter Anhaltspunkt ist das Unicode-Zeichen mit der Nummer 65.279 (hexadezimal feff). Dieses Zeichen ist „illegal“, es ist das sogenannte *Byte Order Mark - BOM*.

Der Nutzen des BOM erschließt sich erst dann, wenn man sich überlegt, wie das Zeichen bei „falschem“ Byte-Sex interpretiert wird. Liest ein Big-Endian-System ein BOM aus Daten, die auf einem Little-Endian-System erzeugt wurde, sieht es nämlich nicht die Zahl 65.279 (Hex feff), sondern - da die beiden Bytes vertauscht sind - die Zahl 65.534 (Hex fffe), und das Unicode-Zeichen mit der Nummer 65.534 ist im Unicode-Standard ebenfalls als illegal gekennzeichnet. Konsequenz: Bei allen folgenden Byte-Paaren müssen die Bytes jeweils vertauscht werden, und das gilt praktischerweise sowohl für Big-

Endian- als auch für Little-Endian-Systeme, was wiederum die Portabilität von Programmen auf Quelltextebene erleichtert.

Es ist nicht verwunderlich, dass das BOM normalerweise das erste Zeichen in einer Textdatei ist. Clevere Programme interpretieren es aber an beliebigen Stellen (Rationale: Die Datei könnte vom User aus Einzeldateien zusammengesetzt worden sein).

Bei den sogenannten *Surrogaten* handelt es sich um eine Kombination mehrerer Unicode-Zeichen, die speziell interpretiert werden, um die Beschränkung auf theoretisch maximal 65.536 Zeichen zu sprengen. In der Praxis ist dies wenig relevant, weil die Surrogate in den meisten Fonts - die ja zur endgültigen Darstellung der Textdaten notwendig sind - schlichtweg fehlen.

4.1.3. UCS-2

Die Unterschiede zwischen UTF-16 und UCS-2 sind mehr akademischer Natur. Der Name ist aus *Universal Character Set* und *2 Bytes* zusammengesetzt.

4.1.4. UCS-4

Mit 2 Bytes/8 Bits lassen sich theoretisch nur 65.536 Zeichen repräsentieren; Unicode sieht über den Extension-Mechanismus aber mehr als eine Million weiterer Zeichen vor. Diesen Zeichen wurde natürlich genauso eindeutige Nummern zugewiesen, und konsequenterweise umgehen viele Systeme den zusätzlichen Aufwand, den der Extension-Mechanismus verursacht, indem sie gleich 4 Bytes (32 Bits) für Wide Characters reservieren. Tatsächlich ist dies nicht die Ausnahme, sondern die Regel. Das Codeset UCS-4 (und die Varianten UCS-4BE und UCS-4LE) ist dort die „natürliche" interne Repräsentation von Unicode-Daten, und wer als Programmiererin auf den Datentyp Wide Character (in C `wchar_t`) stößt, sollte eher mit 4-Byte- bzw. 32-Bit-Daten rechnen.

4.2. Multi-Byte-Encodings

Da die „natürliche" Repräsentation von Unicode-Daten durch *Wide Characters* unter etlichen Nachteilen leidet, werden solche Daten oft mit variablen Zeichenfolgen, Multi-Byte-Encodings, kodiert. In diesen Kodierungen hat ein Zeichen keine fixe Länge in Bytes; vielmehr werden Zeichen durch einen Escaping-Mechanismus in eine Byte-Folge variabler Länge konvertiert.

4.2.1. UTF-7

UTF-7 steht für *Universal Transformation Format 7 Bits*, und erfreut sich mäßiger Beliebtheit, jedoch einiger Verbreitung insbesondere im E-Mail-Verkehr. Wir wissen ja bereits, dass E-Mail strenggenommen nur ASCII-Zeichen, also 7-Bit-Zeichen akzeptiert, sprich noch nicht einmal Umlaute, akzentuierte oder andere im Standard für 8-Bit-Zeichensätze ISO-8859-1 definierte Zeichen klarkommt. UTF-7 umgeht diese Schwierigkeiten, indem beliebige (Unicode-)Zeichen mithilfe von ASCII-Zeichen repräsentiert werden. Erreicht wird dies durch eine Sonderinterpretation des Pluszeichens. Ein Pluszeichen leitet immer eine Escape-Sequenz ein, die ein oder mehrere Nicht-AS-

CII-Zeichen repräsentiert, ein Minuszeichen beendet sie. Die Zeichenkette „Große Füße“ stellte sich in ASCII beispielsweise als `Gro+AN8-e F+APwA3w-e` dar. Dies lässt sich mit einiger Fantasie noch entschlüsseln. Fragen wir hingegen auf Russisch `Что делать?`, stellt sich dies schon als `+BCcEQgQ+ +BDQENQQ7BDAEQgRM?` dar. Viel mehr als das Leer- und Fragezeichen dürfte hier nicht entzifferbar sein.

UTF-7 wurde in [RFC2152] standardisiert.

Der schlechte Ruf dieser Codierung rührt nicht zuletzt daher, dass einige minderwertige Mail-Programme für MS-DOS-Derivate dieses Codeset oft ganz und gar überflüssigerweise verwenden, obwohl wesentlich verbreitertere und elegantere Alternativen existieren). Ein ganz entscheidender Nachteil ist weiterhin die Tatsache, dass eine UTF-7-Codierung nicht eineindeutig ist, das heißt, ein und die selbe (Unicode-)Zeichenfolge lässt sich auf verschiedene Arten in UTF-7 kodieren. Es wurde bereits dargelegt, weshalb dies beispielsweise eine Volltextsuche über einen Text erheblich verkompliziert.

4.2.2. UTF-8

UTF-8 ist der De-Fakto-Standard für den Austausch von Unicode-Daten. Diese Kodierung ist auch unter den Namen FSS_UTF (*file-system-safe Universal Transformation Format*), oder UTF-2 (Version 2 des Universal Transformation Format) bekannt. Die Kodierung ist in [RFC2279] standardisiert.

Der Alias UTF-FSS deutet bereits an, dass diese Kodierung etliche Nachteile der anderen Methoden umgeht, dies sogar auf eine außerordentliche clevere Art und Weise. Wie funktioniert UTF-8 also?

In UTF-8 wird jedes ASCII-Zeichen durch sich selbst dargestellt, Sonderzeichen durch eine Folge von Nicht-ASCII-Zeichen. Der Aufbau dieser Multi-Byte-Folgen ist wiederum äußerst geschickt definiert. Eine solche Folge beginnt immer mit einem Byte, dessen höchstes und zweithöchstes Bit immer *gesetzt*, also Eins sind. Die Anzahl der nachfolgenden gesetzten Bits bestimmt die Länge der Multi-Byte-Folge:

Tabelle 2.1. Anfangsbyte von UTF-8

Erstes Byte	Länge des Zeichens in Bytes
0xxxxxxx	1
110xxxxx	2
1110xxxx	3
11110xxx	4
111110xx	5
1111110x	6

Eine Anwendung muss also lediglich die Anzahl der gesetzten Anfangs-Bits eines Bytes „zählen“, um die Länge des jeweiligen Zeichens zu ermitteln. Bei ASCII-Zeichen ist das erste (von acht) Bits nicht gesetzt (ASCII ist eine 7-Bit-Codierung), das Zeichen hat eine

Länge von eins, es repräsentiert sich - im ASCII-Sinne - selbst. Sind mindestens zwei Anfangsbits gesetzt, lässt sich durch Zählen dieser Bits die Länge des Zeichens ermitteln.

Die nachfolgenden Bytes einer Multi-Byte-Sequenz entsprechen stets dem Muster `10xxxxxx`, also einem gesetzten, gefolgt von einem nichtgesetzten Bit. Die bisher nicht besprochenen Bits (durch „x“ dargestellt) werden zur Codierung des eigentlich gemeinten Unicode-Zeichens verwendet. Bei 2-Byte-Sequenzen, die dem Muster `110xxxxx 10xxxxxx` entsprechen müssen, zählen wir insgesamt elf mal „x“, haben also elf Bits zur Codierung des Zeichens zur Verfügung. Mit diesen elf Bits lassen sich schon 2047 (hexadezimal 7ff) Zeichen darstellen. In der Übersicht:

Tabelle 2.2. Prinzip von UTF-8

Muster	Bits	Wertebereich
		Hexadezimal
0xxxxxxx	7	0-7f
110xxxxx 10xxxxxx	11	80-7ff
1110xxxx 10xxxxxx 10xxxxxx	16	800-ffff
11110xxx 10xxxxxx 10xxxxxx 10 xxxxxx	21	10000-1fffff
111110xx 10xxxxxx 10xxxxxx 10 xxxxxx 10xxxxxx	26	200000-3ffffff
1111110x 10xxxxxx 10xxxxxx 10 xxxxxx 10xxxxxx 10xxxxxx	31	4000000-7fffffff

Wie werden Zeichen konkret in UTF-8 kodiert? Nehmen wir als Beispiel die bereits erwähnte arabische Ligatur für Gott bzw. Allah ﷲ, der in Unicode die Nummer 65.010 (Hexadezimal fdf2) zugewiesen wurde. Wir werden jetzt schrittweise die Kodierung dieses Zeichens in UTF-8 entwickeln. 65.010 liegt im Bereich 2.048-65.535, wir können obiger Tabelle also entnehmen, dass das Zeichen in insgesamt drei Bytes kodiert wird, die folgendem Muster entsprechen:

```
1110xxxx 10xxxxxx 10xxxxxx
```

Das erste Byte beginnt mit drei gesetzt Bits, die gesamte Multi-Byte-Folge hat also die Länge 3 (in Bytes). Es bleiben noch 16 Bits für die Kodierung des Zeichens übrig. Als nächstes wandeln wir die Dezimalzahl 65.010 in eine Binärdarstellung um:

```
                    65.010  (dezimal)
```

```
                fd f2   (hexadezimal)
   f    d     f    2   (hexadezimal)
1111 1101  1111 0010   (binär)
```

Diese 16 Bits müssen jetzt in die Schablone für die Zeichen 2.048-65.535 „eingesetzt werden“.

```
    11111101  11110010
1110xxxx 10xxxxxx 10xxxxxx

     Neugruppiert:
    1111    110111   110010
1110xxxx 10xxxxxx 10xxxxxx

     Eingesetzt:
11101111 10110111 10110010
   e   f    b   7    b   2
```

Das Zeichen wird also in die Drei-Byte-Folge `ef b7 b2` transformiert. Diese drei Bytes würden im geläufigeren ISO-8859-1 als „ï·²“ dargestellt, also als kleines i mit zwei Punkten, als Multiplikationspunkt, gefolgt von einer hochgestellten zwei. Dass unsere Konversion korrekt ist, lässt sich mit einem einfachen Experiment feststellen: Wenn wir die drei Zeichen `ï·²` per Cut & Paste in eine Text-Datei in kopieren, und im Browser darstellen, wird der Browser sie zunächst als ISO-8859-1 interpretieren. Die meisten Browser bieten im Menü Ansicht/View die Möglichkeit, das Codeset für ein Dokument manuell zu setzen. Wählen wir dort Unicode (UTF-8) müsste unsere arabische Ligatur erscheinen, vorausgesetzt, der eingesetzte Zeichensatz enthält dieses Zeichen.

Versuchen wir das Zeichen mit der Maus zu selektieren, stellen wir fest, dass der Browser es wirklich als ein einziges Zeichen interpretiert, es lässt sich nur komplett selektieren.

Falls das nicht klappt, liegt es vermutlich daran, dass der Browser beziehungsweise das Betriebssystem nicht in der Lage sind, arabische Texte darzustellen. In diesem Falle wiederholen wir das Experiment mit der Zeichenkette „übermütig“. Das kleine ü wird in UTF-8 durch die zwei Bytes `c3 bc` (hexadezimal) repräsentiert. Diese beiden Bytes entsprechen einem großen A mit Tilde und dem Zeichen für ein Viertel, also „Ã¼“, insgesamt wird `übermütig` in UTF-8 also zu `Ã¼bermÃ¼tig`. Pasten wir diese Zeichenkette in eine Text-Datei, und verstellen das Browser-Codeset auf UTF-8, können wir den Effekt ebenfalls beobachten.

Welche Eigenschaften von UTF-8 können wir ausmachen?

4.2.2.1. ASCII-Transparenz

ASCII-Zeichen bleiben unangetastet. Sie stehen für sich selbst, das heißt, das UTF-8 *filesystem-safe* ist, und z. B. auch Programmcode (der in aller Regel zwingendermaßen in US-ASCII geschrieben ist) voll gültig bleibt. Dies gilt für alle Anwendungen, in denen traditionell die Verwendung von Sonderzeichen vermieden wird, wie Dateinamen (File-System-Safety), E-Mail-Headern, Anweisungen in Imperia-Templates, ...

Besonders wichtig ist auch, dass ein Null-Byte in UTF-8-Daten für sich selber steht, es kann also niemals Teil eines gültigen UTF-8-Zeichens sein, und ist weiterhin geeignet zur Kennzeichnung des Endes einer Zeichenkette.

Texte, die in Obermengen von US-ASCII verfasst sind, wie z. B. westeuropäische Texte bleiben zumindest noch deutbar. Die Zeichenkette „übermutig“ stellt sich - fälschlicherweise als ISO-8859-1 interpretiert - als `Ã¼bermÃ¼tig` dar. In Texten mit weniger Sonderzeichen und mehr Kontext-Informationen ist der Negativ-Effekt deutlich geringer ausgeprägt.

Die ASCII-Transparenz von UTF-8 ist für den praktischen Gebrauch enorm wichtig. Etliche Anwendungen (Beispiele hierfür sind der Linux-Kernel, C, oder auch Imperia) werden dadurch unicodefähig, dass sie Unicode größtenteils schlichtweg *ignorieren*, also einfach nichts in die zu verarbeitenden Daten hineininterpretieren, sie weiterhin als Byteströme betrachten, weil sie wissen, dass die für die Anwendung „interessanten“ Bytes, die traditionell im ASCII-Bereich liegen, weiterhin eindeutig zu erkennen sind. Anwendungen bzw. Systeme, die den gegenteiligen Ansatz verfolgen (Beispiele sind Java, XML oder auch Perl), also die interne Interpretation eines „Zeichens“ ändern, erkaufen sich diesen Ansatz mit erheblichen Performanceeinbußen und umständlichen Schnittstellen, ohne Gewinne bei der Funktionalität.

4.2.2.2. Selbstsynchronisierung

Auch das Problem der Synchronisationsfähigkeit in Byte-Strömen ist auf elegante Weise gelöst. Stößt eine Anwendung in einem UTF-8-Strom auf ein Byte, dessen erstes Bit nicht gesetzt ist, kann es sofort aufsetzen, weil es sich um ein normales ASCII-Zeichen handeln muss. Sind mindestens die ersten beiden Bits gesetzt, weiß die Anwendung, dass sie den Anfang einer Multi-Byte-Folge erwischt hat, und ist ebenfalls sofort synchronisiert. Ist dagegen das erste Bit gesetzt und das zweite nicht, sind wir auf den *casus irreducibilis* gestoßen, auf eine unvollständige Multi-Byte-Sequenz, und die Anwendung muss die folgenden Bytes ignorieren, bis sie auf ein Byte stößt, dass entweder für sich selber steht (7-Bit) oder den Anfang einer neuen Multi-Byte-Sequenz kennzeichnet. Das ist nicht wirklich tragisch, die Anwendung hat ja tatsächlich mitten in einem Buchstaben angefangen zu lesen, und in dieser Situation ist es sicher legitim, das unvollständig gelesene Zeichen zu ignorieren.

4.2.2.3. Speicherverbrauch

Auch die Bilanz beim Speicherverbrauch ist durchweg positiv. Bei ASCII-Texten/-Zeichen gibt es keinen Unterschied. Bei Texten, die traditionell in 8-Bit-Zeichensätzen dargestellt wurden (das ist in etwa der Bereich von 128-2.047) wer-

den statt bisher maximal 8 Bit jetzt 16 Bit pro Zeichen verbraucht. Dies relativiert sich allerdings, wenn - wie in den meisten europäischen Sprachen - der ASCII-Anteil überwiegt (deutsche Texte werden typischerweise um weniger als 5 Prozent „aufgebläht“). In nichtlateinischen Alphabeten (z. B. kyrillisch) liegt die Rate knapp unter 100 % (unter der Annahme, dass die Interpunktions- und Steuerzeichen identisch sind), bei anderen Sprachen, die mit der *Basic Multilingual Plane - BMP* (Unicode 0-65534) darstellbar sind, hängt der Faktor vom Referenz-Codeset ab, dürfte in der Regel jedoch knapp unter 50 % liegen (2 Bytes vs. 3 Bytes). Die Gesamtbilanz ist durchaus annehmbar.

4.2.2.4. Nachteile von UTF-8

Die Nachteile von UTF-8 sollen jedoch auch nicht verschwiegen werden. Beim Speicherverbrauch begünstigt UTF-8 Sprachen mit lateinischen Alphabeten und „diskriminiert“ andere, auch wenn die tatsächlichen Faktoren einen ziemlich vorteilhaften Kompromiss darstellen.

Außerdem ist die Dekodierung von UTF-8 mit einigem Rechenaufwand verbunden, die oben beispielhaft durchgeführte Kodierung der arabischen Ligatur für Allah sieht nicht nur kompliziert aus, sie ist es auch. Noch komplizierter ist es jedoch aus der UTF-8-Darstellung zurückzugelangen, also UTF-8-Ströme zu dekodieren. Auch triviale Aufgaben wie die Bestimmung der Länge einer Zeichenkette (in byteorientierten Anwendungen läuft das auf stupides Zählen der Bytes hinaus) werden plötzlich zur Herausforderung.

Zusätzlich ergibt sich die Schwierigkeit, dass UTF-8 „Lücken“ aufweist, illegale Multi-Byte-Sequenzen, die in kein gültiges Unicode-Zeichen umgewandelt werden können. Bei der Dekodierung müssen ständig Vorkehrungen für diesen Fall getroffen werden, was eine Anwendung erheblich verlangsamen kann.

Schließlich ist auch die relative Ähnlichkeit von UTF-8 zu ASCII gleichzeitig Fluch und Segen: Beginnt eine Datei mit eine der 2-Byte-Sequenzen feff (Hex) oder fffe (Hex) gehen viele Programme stillschweigend davon aus, dass es sich beim Inhalt der Datei um Daten in UTF-16 handeln muss. Bei UTF-8 fehlt dagegen eine solche automatische Erkennungsmöglichkeit.

Trotz dieser Nachteile ist UTF-8 heute der De-Fakto-Standard für den Austausch und die Speicherung von Unicode-Daten. Beim Datenaustausch treten die Schwierigkeiten ja nur an den Schnittstellen auf, und können durch eine einmalige Konvertierung in performantere Formate für den internen Gebrauch leicht umgangen werden.

5. Zusammenfassung

Unicode erlaubt die Definition von mehr als einer Million Schriftzeichen. Intern können diese Zeichen entweder als *wide characters* mit einer mehr oder weniger konstanten Größe von zwei oder vier Bytes oder mit *UTF-8* ([RFC2279], einer Multi-Byte-Kodierung repräsentiert werden. Wide Characters werfen beim Datenaustausch etliche Probleme auf, sind aber auf Kosten eines stark erhöhten Speicherverbrauchs in geschlossenen Systemen sehr performant zu verarbeiten. UTF-8 zeichnet sich durch einen nur mäßig erhöh-

ten Speicherverbrauch und maximale Abwärtskompatibilität zu ASCII und ISO-8859-1 aus, stellt aber höhere Anforderungen an die Rechenleistung.

Teil II. Unicode und Multi-Language im Web

Inhaltsverzeichnis

3. Web-Standards 40
1. Hypertext Transfer Protocol HTTP 40
1.1. Funktionsweise von HTTP 40
1.2. Der Header Content-Type 43
1.2.1. Die Konfigurationsanweisung AddDefaultCharset 43
1.2.2. Die Konfigurationsanweisung AddCharset 44
1.3. Der Header Content-Language 44
1.3.1. Die Konfigurationsanweisung DefaultLanguage 46
1.3.2. Die Konfigurationsanweisung AddLanguage 46
1.4. Voreingestellte Werte 47
1.5. Der Header Accept-Charset 47
1.6. Der Header Accept-Language 48
1.7. Formulare 49
2. Extensible Markup Language XML 51
2.1. Perfomance-Erwägungen 52
2.2. XML in UTF-8 53
2.3. Sprachbestimmung 53
3. Hypertext Markup Language HTML 54
3.1. Das HTML-Attribut lang 54
3.2. Das Attribut http-equiv des meta-Elements. 54
3.3. XHTML 56
4. Content-Negotiation 56
4.1. Implementierung mit dem Apache 56
4.2. Praktische Erwägungen 58
4.2.1. Sprachpersistenz 58
4.2.2. Sprachumschaltung 59
4.2.3. Vertikale oder horizontale Aufteilung 59
5. Zusammenfassung 60

Kapitel 3. Web-Standards

Die meisten modernenen textbezogenenem Standards sind von vorneherein so angelegt, dass sie Texte in beliebigen Sprachen adäquat verarbeiten können. Die für Web-Autoren relevanten Standards HTTP, XML und HTML werden in den nachfolgenden Abschnitten einer näheren Betrachtung unterzogen.

1. Hypertext Transfer Protocol HTTP

Das Akronym HTTP steht für *Hypertext Transfer Protocol*. Das Protokoll ist in [RFC2616] standardisiert. Obwohl der Name etwas anderes suggeriert, beschränkt sich seine Anwendung keineswegs auf die Übertragung von Hypertext, oder anderen Textformaten. Es dient heute vielmehr der Übermittlung beliebiger Daten, kann also Bilder, Sound, Video, Vektorgraphiken, ja ganze Programme oder Software-Archive übertragen. Im folgenden beschränken wir uns allerdings weitestgehend auf die Übertragung von Textdateien über HTTP.

Es sei noch angemerkt, dass sich alle Beispiele nur auf den Web-Server Apache beziehen. Andere Web-Server implementieren HTTP oftmals fehlerhaft, manche Features sind nicht vorhanden oder nur mit unverhältnismäßigem Aufwand oder durch Erwerb von Zusatzmodulen zu realisieren, und schließlich legt die große Verbreitung des Apache-Servers eine Beschränkung in der Darstellung auf diesen einen Server nahe.

1.1. Funktionsweise von HTTP

HTTP ist ein textbasiertes Protokoll, der prinzipielle Ablauf ist also von Menschen lesbar. Das Protokoll ist stets zweistufig: Der Web-Browser (*Client*) schickte eine Anfrage (einen *Request*) zum *Server*, woraufhin der Server eine Antwort zurückschickt. Sowohl die Anfrage als auch die Antwort bestehen aus zwei Teilen, einem *Header* mit Meta-Informationen über die Anfrage bzw. die Antwort, und einem *Body*, der die eigentlichen Informationen enthält.

Spielen wir als Beispiel einmal durch, was passiert, wenn wir mit einem Web-Browser die imaginäre Web-Seite http://www.gips.nicht/mein/home.html aufrufen. Der Browser wird in diesem Falle ein Datenpaket schicken, das in etwa so aussieht:

```
GET /mein/home.html HTTP/1.1                      ❶
❷❸
Host: www.gips.nicht                              ❹
User-Agent: Microscape Browser 6.15               ❺
Connection: close                                 ❻
                                                  ❼
```

❶ Die erste Zeile beginnt stets mit dem Request-Typ. Außer GET ist hier für praktische Belange noch POST interessant, dass zur Übermittlung von Formular-Daten verwendet wird (auch HEAD zur Ermittlung von Meta-Informationen und PUT zum Datei-Upload sind in der Praxis noch relevant).
❷ Durch Leerzeichen getrennt folgt der Pfad zum angeforderten Dokument.
❸ Die erste Zeile wird abgeschlossen durch die Angabe des verwendeten Protokolls (hier HTTP) und der Protokoll-Version (zur Zeit aktuell ist 1.1).
❹ Der (virtuelle) Hostname des Servers *muss* ab HTTP-Version 1.1 immer mitgesendet werden. Dies ist notwendig, weil der Server sonst nicht wissen kann, unter welchem Namen er aufgerufen wurde, denn er sieht nicht etwa die symbolische Adresse „www.gips.nicht", sondern lediglich eine nummerische IP-Adresse wie 192.168.6.29, die aber beliebig vielen symbolischen Adressen zugeordnet sein kann. Durch den Host-Header erfährt der Server, welcher virtuelle Host vom Browser „gemeint" war. Mit Hilfe dieser virtuellen Hosts ist es also möglich, dass ein und derselbe physikalische Server beliebig viele symbolische Adressen und sogar Domains bedienen kann. Um bei unserem Beispiel zu bleiben: Würden wir für die gleiche IP-Adresse auch noch die Namen „www.gips.doch" und „das.gips.nicht" vergeben, könnte unser Server - bei entsprechender Konfiguration - noch für zwei weitere komplette Sites eingesetzt werden.
❺ Ein String, der den Browser identifiziert.
❻ Der Verbindungstyp. In diesem Falle weisen wir den Server an, die Verbindung nach der Antwort zu schließen. Alternativ könnten wir mit dem Wert *Keep-Alive* den Server „bitten", die Verbindung für weitere Anfragen offenzuhalten, weil wir beispielsweise wissen, dass wir noch einige Dutzend Bilder vom gleichen Server holen müssen.
❼ Die abschließende Leerzeile teilt dem Server mit, dass der Request hier endet.

Man kann dieses Beispiel übrigens durchaus am eigenen Computer nachvollziehen, sofern das Programm **telnet** auf dem Rechner installiert ist. Unter Unix ruft man es in der Form **`telnet`** ***`www.servername.com`*** **`80`** auf (80 ist die Port-Nummer für HTTP) und tippt die obige Protokollausgabe ein. Unter Windows öffnet man eine DOS-Box und tut das gleiche wie unter Unix. Im Gegensatz zur Unix-Version sieht man allerdings nicht, was man eintippt. Am besten kopiert man den Text also aus einem anderen Fenster in die DOS-Box.

Der Server wird auf die Anfrage in jedem Fall antworten. Hat man sich nicht vertippt, wird er sogar das gewünschte Dokument ausliefern, in unserem Falle also `/mein/home.html` (auf Vertipper wird der Server mit einer entsprechenden Fehlermeldung antworten, die vom Format aber gleich aufgebaut ist). Allerdings ist es etwas mühselig, im Telnet-Fenster zu scrollen, um die Server-Antwort zu lesen. Bequemer geht es mit dem Programm **wget**, das auf den meisten Unix-Systemen vorinstalliert ist (unter Windows ist es in der Cygwin-Umgebung enthalten). Mit **wget** lassen sich Web-Dokumente über die Kommandozeile herunterladen, für unser Beispiel ließe sich das mit **`wget -s`** ***`http://www.gips.nicht/mein/home.html`*** bewerkstelligen (die Option `-s`, bzw. `--save-headers` weist **wget** an, die Header des Servers in der Datei mit abzuspeichern). Falls **wget** mit **`...:`** `Connection refused` antwortet, müssen wir wahrscheinlich noch einen Proxy-Server angeben. Das geht, indem man in das Terminalfenster das Kommando **`http_proxy=http://`*****`proxy:3128/`*****`;`** **`export`**

`http_proxy` eingibt. Statt *proxy* und *3128* muss natürlich der Name des Proxy-Servers und dessen Port eingegeben werden. Beide Informationen bekommt man zur Not von seiner Systemadministration.

Die Antwort des Servers sollte jedenfalls ungefähr wie folgt aussehen:

```
HTTP/1.1 200 OK                                                    ❶
❷
Date: Mon, 09 Jun 2003 12:53:37 GMT
Server: Apache/1.3.26 (Linux/SuSE) PHP/4.2.2 mod_perl/1.27
Last-Modified: Mon, 09 Jun 2003 12:53:30 GMT
ETag: "f368c-f1-3ee4834a"
Accept-Ranges: bytes
Content-Length: 280
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html                                            ❸

<html>                                                             ❹
<head>
  <meta http-equiv="Content-Type"
        content="text/html; charset=iso-8859-1" />                 ❺
  <title>Meine Homepage</title>
</head>
<body>
  <h1>Willkommen auf meiner Homepage</h1>
  <p>
Die ist eine schöne Homepage, weil der Autor hat sie selbst gemacht.
  </p>
</body>
</html>
```

❶ Die Antwort des Servers beginnt wiederum mit der Angabe des Protokolls und der Version.

❷ Auf die Protokollangabe folgt ein nummerischer Status-Code (hier 200) und ein formfreier Text, der diesen nummerischen Status-Code im Klartext (hier: „OK“) beschreibt. Welche Status-Codes möglich sind, ist in [RFC2616], Abschnitt 10 genau festgelegt.

❸ Hier beschreibt der Server, um welchen Typ von Datei es sich handelt, und genau dieser Header „Content-Type“ ist für uns interessant und wird unten noch näher betrachtet.

❹ Wiederum durch eine Leerzeile abgetrennt folgt die eigentliche Datei, die wir angefordert haben.

❺ Bemerkenswert ist, dass der Server - obwohl es sich beim angeforderten Dokument um eine Text-Datei handelt - keinerlei Angaben darüber macht, in welchem Codeset die Datei vorliegt. Dies ist erst dem Dokument selber zu entnehmen.

Tatsächlich wäre es allerdings durchaus möglich (durch eine entsprechende Konfiguration des Servers), dass dieser das Codeset doch schon in den Headern mitsendet. Wie dies auszusehen hätte, ist im folgenden Abschnitt beschrieben.

1.2. Der Header *Content-Type*

Das Format des Headers Content-Type ist in [RFC2616] genau beschrieben. Es besteht aus der Angabe eines Medientyps, optional gefolgt von zusätzlichen Parametern, die durch ein Semikolon vom Medientyp abgetrennt sein müssten. Medientypen bestehen jeweils aus einer allgemeinen Typbezeichnung (application, audio, image, message, model, multipart, text, video) und einem Untertyp; die vollständige Liste kann [IANA-MEDIATYPES] entnommen werden. Selbstdefinierte Typen und Untertypen müssen mit „x-“ beginnen (diese Einschränkung scheint für Microsoft nicht zu gelten, denn dort brilliert man gerne mit der Erfindung von Fantasietypen wie „image/pjpeg“).

Uns interessiert vornehmlich der Typ „text“ mit dem Untertyp „html“. In einer vollständigen Typbezeichnung müssen die beiden Bestandteile durch einen Schrägstrich getrennt werden, es würde sich also `text/html` als Bezeichnung ergeben. Die Kodierung eines Dokumentes ist in einem optionalen Parameter *`charset`* zum Medientyp anzugeben. Die vollständige Bezeichnung für ein HTML-Dokument, dass in UTF-8 kodiert ist, wäre also `text/html; charset=utf-8`.

Groß-/Kleinschreibung spielt keine Rolle, für maximale Portabilität sollte man die Medientyp-Angabe allerdings immer klein schreiben. Ein häufiger Fehler ist es, den Wert des Parameters *`charset`* in Anführungszeichen zu setzen (wie in `text/html; charset="utf-8"`. Das ist falsch! Zwar sind Anführungszeichen prinzipiell in Parameterwerten erlaubt, jedoch nicht bei *`charset`*.

Wie bekommen wir jetzt aber den Server dazu, diesen Parameter bei der Auslieferung des Dokumentes mitzuschicken? Bei dynamisch generierten Dokumenten ist dies kein Problem, denn dort gibt das Programm, dass den Inhalt generiert, meistens zumindest den Header Content-Type ohnehin selber aus. Sofern der Server diesen Header nicht später wieder überschreibt, was durchaus möglich ist:

1.2.1. Die Konfigurationsanweisung `AddDefaultCharset`

Diese Konfigurationsanweisung gibt es in drei Varianten:

```
AddDefaultCharset charset     (1)
AddDefaultCharset Off         (2)
AddDefaultCharset On          (3)
```

(1) Allen Dokumenten, deren Charset nicht auf anderem Wege ermittelt werden kann, wird im Header der angegebene *`charset`*-Parameter mitgegeben.

(2) Diese Anweisung schaltet das Feature ab.

❸ Dies hat beim Apache die gleiche Wirkung wie **`AddDefaultCharset iso-8859-1`**.

Die Verwendung der Direktive ist eine beliebte Fehlerquelle, weil es selbst bei dynamisch generiertem Content die Charset-Angabe überschreibt. Wenn Browser also penetrant von einer falschen Kodierung der Dokumente auf dem eigenen Server ausgehen, sollte man die Server-Konfiguration nach dieser Anweisung durchsuchen.

In anderen Fällen kann es jedoch durchaus sinnvoll sein, die Anweisung zu benutzen. Gerade in Umgebungen, in denen viele Dokumente durch Authoring-Tools unter MS-Windows erzeugt werden, können damit fehlerhafte Angaben im Dokument (z. B. iso-8859-1 statt windows-1252) auf einen Schlag vom Server „repariert“ werden.

`AddDefaultCharset` ist ab Version 1.3.12 im Apache verfügbar.

1.2.2. Die Konfigurationsanweisung `AddCharset`

```
AddCharset utf-8 .utf8
AddCharset koi8-r .koi
AddCharset iso-8859-1 .iso
```

Diese Anweisung erfordert, dass das Apache-Modul `mod_mime` in den Server einkompiliert und in der Konfiguration aktiviert wurde. Dies ist in der Voreinstellung der Fall.

Bei `AddCharset` gehen wir schon etwas weniger mit dem Holzhammer an das Problem heran. Sofern man eine Namenskonvention hat, kann man den Server einfach anweisen, den Parameter *`charset`* in Abhängigkeit vom Dateinamen, genauer gesagt der Dateiendung, zuzufügen. Bei allen Dateien, deren Name auf `.utf8` endet, wird der Server UTF-8 als Charset angeben, bei `.iso` ISO-8859-1 und bei `.koi` KOI8-R.

Problematisch ist dabei allerdings, dass der Dateiname in gewissem Sinne „verhunzt“ wird. Würden wir beispielsweise auf die Idee kommen, hier die Endung `.gif` zu „missbrauchen“, werden sich gerade ältere Browser evtl. nur mit Mühe davon abbringen lassen, die Datei als Bild anzuzeigen, selbst, wenn der Server den korrekten Medientyp `text/html` mitgegeben hat.

Das Problem lässt sich jedoch umgehen, wie wir im Abschnitt 4, „Content-Negotiation“ über *Content-Negotiation* sehen werden.

1.3. Der Header *Content-Language*

Nimmt ein Browser eine falsche Kodierung für ein Dokument an, hat das in der Regel ziemlich augenfällige Konsquenzen. Entweder ist der Text von Fragezeichen oder kleinen Rechtecken durchsetzt, oder man sieht scheinbar völlig sinnlosen Zeichenmüll. Der Grund ist natürlich, dass der Browser den Byte-Strom fehlinterpretiert, und nicht in die korrekten Schriftzeichen umwandelt.

Weniger augenfällig ist die Notwendigkeit, auch die Sprache eines Dokumentes als Meta-Information mitzuliefern. Fangen wir mit einem (leider untauglichen) Beispiel in HTML an:

```
    <p>
They kill babies in the crib, and say <q>only the good die young ...</q>
    </p>
```

Aber hallo! Das Element `q` gibt es doch gar nicht in HTML, das muss doch `"` heißen! Weit gefehlt, aber selbstverständlich gibt es das! Allerdings erst seit Dezember 1997 (siehe [HTML-4.0]), und heute, keine sechs Jahre später kann man natürlich nicht verlangen, dass schon alle neuen Browserversionen dieses Feature unterstützen. Der mit der größten Marktdurchdringung ignoriert es jedenfalls schlichtweg ...

Gedacht ist das Element für Zitate innerhalb eines Textblocks. Ein `"` ist nämlich *immer* das Zeichen mit dem ASCII-Code 34, "doppelte" Anführungszeichen. Es gibt jedoch keine bekannte Sprache, in der dies tatsächlich die typographisch korrekte Darstellung für Zitate wäre, und für diesen Zweck wurde das HTML-Element `q` eingeführt. Je nach Dokumentensprache soll der Browser hier beispielsweise “englische Quotes”, „deutsche Gänsefüßchen“ oder «französische Guillemots» verwenden. Das macht allerdings nicht ein einziger gängiger Browser. Dass dies aber technisch durchaus machbar ist, beweist *dieses* Dokument. Es ist in DocBook-XML verfasst, und der fragliche Absatz sieht im Quelltext so aus:

```
    eingeführt. Je nach Dokumentensprache soll der Browser hier
    beispielsweise <quote lang="en">englische Quotes</quote>, <quote
    lang="de">deutsche Gänsefüßchen</quote> oder <quote
    lang="fr">französische Guillemots</quote> verwenden. Das macht
```

Die Idee sollte dennoch klar sein. Auch jenseits der reinen Zuordnung zu Schriftzeichen kann die Dokumentensprache durchaus von Bedeutung für die Darstellung sein. Praktisch noch bedeutsamer ist dies für barrierefreien Zugang zu Dokumenten: Ein sogenannter *Screen-Reader*, der von Blinden genutzt werden kann, um textuelle Inhalte vorzulesen, sollte natürlich schon wissen, ob ein Dokument in englischer, deutscher oder chinesischer Sprache verfasst ist, weil die negativen Konsequenzen sonst von unfreiwilliger Komik bis zur völligen Unverständlichkeit führen können.

Die größte praktische Bedeutung besitzt diese Information allerdings dadurch, dass Browser in der Lage sind, gezielt Dokumente in einer bestimmten Sprache vom Server anzufordern, und dafür muss der Server natürlich ebenfalls ermitteln können, in welcher Sprachen ein Dokument jeweils vorhanden ist, um die gewünschte Version ausliefern zu

können. Für Details dazu sei wiederum auf den später folgenden Abschnitt 4, „Content-Negotiation“ über *Content-Negotiation* verwiesen.

1.3.1. Die Konfigurationsanweisung `DefaultLanguage`

Die Möglichkeiten, den Server zur Übermittlung des Headers Content-Language zu bewegen, sind denen für Content-Type sehr ähnlich.

```
DefaultLanguage Sprachkürzel
```

`DefaultLanguage` (Achtung: Es heißt `AddDefaultCharset`, aber `DefaultLanguage`!) ist ab Version 1.3.4 im Apache verfügbar und erfordert `mod_mime`, das standardmäßig vorhanden ist.

Welche Sprachkürzel genau erlaubt sind, lässt sich [RFC3066] entnehmen; einige erlaubte, aber eher exotische Beispiele kann man sich auf [IANA-LANGUAGE-TAGS] anschauen. In der Praxis sollte man sich auf die gängigen Kürzel beschränken, und das sind die jeweils aus zwei Buchstaben bestehenden Sprachkürzel nach ISO 639, wie de für Deutsch, fr für Französich, en für Englisch, ru für Russisch, und so weiter. Optional, durch einen Bindestrich getrennt, kann auch noch nach Ländern variert werden. Ländercodes sind in ISO 3166 festgelegt, Beispiele sind US für die USA, FR für Frankreich, CH für die Schweiz, und so fort. Mit dem Kürzel de-CH würde man somit die Sprache Deutsch für die Schweiz identifizieren, mit en-US amerikanisches Englisch. Obwohl bei Sprachkürzeln Groß-/Kleinschreibung keine Rolle spielt, hat es sich eingebürgert, die Sprachbezeichnung klein, das optionale Land dagegen groß zu schreiben. Allerdings werden diese Angaben mehr und mehr komplett kleingeschrieben. Praktisch ist dies jedoch eine rein ästhetische Frage.

Am einfachsten macht man es sich, wenn man in einem einigermaßen modernen Browser nachschaut, welche Sprachen effektiv in welchen Varianten erkannt werden. Bei Mozilla (bzw. Netscape 6, 7, ...) geht das über Bearbeiten -> Voreinstellungen -> Navigator -> Sprachen, beim Internet Explorer über Extras -> Internet-Optionen -> Allgemein -> Sprachen.

1.3.2. Die Konfigurationsanweisung `AddLanguage`

```
AddLanguage en    .en
AddLanguage de    .de
AddLanguage de-ch .de-ch
```

Diese Anweisung erfordert, dass das Apache-Modul `mod_mime` in den Server einkompi-

liert und in der Konfiguration aktiviert wurde. Dies ist in der Voreinstellung der Fall.

Ähnlich wie bei `AddCharset` (siehe Abschnitt 1.2.2, „Die Konfigurationsanweisung AddCharset“) wird hier eine Zuordnung von Sprachen zu Dateinamensendungen über eine Namenskonvention eingeführt. Heißt eine Datei `print.html.de-ch`, geht der Server also davon aus, dass es sich um ein Dokument für die deutschsprachige Schweiz handelt.

1.4. Voreingestellte Werte

Interessant ist noch die Frage, welche Werte für Sprache und Kodierung angenommen werden, wenn eine explizite Angabe fehlt. Für die Sprache ist das leicht zu beantworten, es gibt nämlich keinen Standardwert. In der Praxis wird der tatsächlich angenommene Wert entweder en-US, also amerikanisches Englisch, oder aber die Systemsprache sein. Beides hat dann allerdings eher programmiertechnische Gründe.

Bei der Dokumentenkodierung sieht es nicht ganz so einfach aus. Abschnitt 3.7.1 in [RFC2616] lässt sich eindeutig entnehmen, dass für diesen Fall ISO-8859-1 angenommen werden muss, jedenfalls soweit es sich um Textdokumente handelt. Tatsächlich werden viele Browser jedoch als Kodierung CP1252 bzw. Windows-1252 annehmen, was einen jedoch nicht gleich zu einem Aufschrei des Entsetzens veranlassen sollte: CP1252 *erweitert* ISO-8859-1 lediglich, und nur in Bereichen, die in ISO-8859-1 ohnehin nicht definiert sind. Selbst, wenn der Browser also mit seiner stillschweigenden Annahme danebenliegen sollte, ist das wenig schädlich. Wenn tatsächlich Bytes aus dem „verbotenen“ Bereich ankommen, dann ist eine fälschliche Darstellung zum Beispiel als Euro-Zeichen auch kaum weniger lästig oder gar gefährlich, denn Fragezeichen, Kästchen oder andere Ersatzdarstellungen.

Allerdings sollte dies *nicht* dazu verleiten, dieses zweifelhafte Feature auszunutzen. Etliche Browser auf etlichen Betriebssystemen werden bei CP1252-Dokumenten nämlich tatsächlich statt der unbekannten Zeichen Müll darstellen (Netscape 4.x unter älteren Linux-Versionen ist ein Beispiel dafür, wenngleich das seinen Grund auch im verwendeten X-Server haben könnte).

Ebenfalls relevant ist das Problem, das in [RFC2616] in Abschnitt 3.4.1 noch genannt wird, dass nämlich ältere Browser das Fehler einer Kodierungsangabe zum Anlass nahmen, die Kodierung zu „raten“. In Gebieten, in denen ISO-8859-1 Standard ist, wird man das Problem nur schwer nachvollziehen können, in anderen Sprachräumen sind Browser dagegen häufig darauf eingestellt, das für die jeweilige Region gängige Charset zu verwenden.

Und schließlich haben wir noch die „übereifrigen“ Browser, die meinen, weil Unicode doch das Allheilmittel gegen alles Übel dieser Welt sei, müssten Dokumente ohne Charset-Angabe auch unbedingt in Unicode, bzw. UTF-8 kodiert sein. Sie schießen damit leicht übers Ziel hinaus, und frustrieren höchstens die Benutzer, weil die Kodierung in der Praxis doch recht häufig fehlt.

1.5. Der Header *Accept-Charset*

Wir erinnern uns, dass HTTP ein Frage- und Antwort-Spiel ist. So, wie der Server angegeben kann, in welcher Form ein Dokument genau vorliegt, kann der Browser Wünsche dazu äußern, also mitteilen, wie er (bzw. die Benutzerin) die Dokumente gerne erhalten möchte. Der Browser übermittelt diese Vorlieben durch Header in seiner Anfrage an den Server. Diese Header beginnen allesamt mit dem Wort *Accept*.

Für die Dokumentenkodierung heißt dieser Header *Accept-Charset*. Ein praktisches Beispiel sieht so aus:

```
Accept-Charset: ISO-8859-1, utf-8; q=0.7,*;q=0.7
```

Prinzipiell verständlich, ..., aber im Detail? Es wird klarer, wenn wir den Inhalt des Headers jeweils an den Kommata aufsplitten. Danach werden also scheinbar die drei Charsets ISO-8859-1, utf-8 und „*" akzeptiert. Übrigens: Der Internet Explorer scheint keine Accept-Charset-Header an den Server zu übermitteln.

Der erste Teil ist klar, der Browser bevorzugt Dokumente in ISO-8859-1. Weiterhin akzeptiert er auch utf-8. Was bedeutet aber die Angabe „`; q=0.7`"? Jedem akzeptierten Typen kann eine Präferenz im Bereich zwischen 0 und 1 mitgegeben werden, wobei 0 die niedrigste und 1 die höchste Präferenz ausdrückt. Eine fehlende Angabe ist als 1 anzunehmen.

Im vorliegenden Fall drückt der Browser daher aus, dass Dokumente in ISO-8859-1 für ihn eine Qualität (daher die Abkürzung q) von 1 haben, Dokumente in utf-8 dagegen nur eine von 0,7.

Das letzte Charset * gibt es natürlich nicht. Es ist eine Auffangbedingung, der Stern bedeutet, dass der Browser mit *allen* Charsets etwas anfangen kann. Man darf getrost davon ausgehen, dass es sich bei dieser Pauschalaussage um eine glatte Lüge handelt. Und genau diese Pauschalaussagen haben den praktischen Nutzen des Verfahrens sehr stark eingeschränkt:

Vorsichtshalber schicken nämlich praktisch alle Browser bei allen Accept-Headern dieses Sternchen mit, weshalb man die Angabe in der Praxis mit Vorsicht genießen muss. Auch Browser, die keine PNGs darstellen können, werden z. B. meistens behaupten, sämtliche Bildformate darstellen zu können.

Als Ausschlusskriterium taugen die vom Browser übermittelten Präferenzen also nichts. Die beigefügten Qualitätswerte sind schon wesentlich nützlicher, weil sie zumindest eine optimale Auswahl des ausgelieferten Dokuments durch den Server erlauben.

1.6. Der Header *Accept-Language*

Dieser Header wird von praktisch allen Browser geschickt und sieht beispielsweise so aus:

```
Accept-Language: de, fr;q=0.8, en-us;q=0.6, en;q=0.4, ru;q=02
```

Wir sollten jetzt in der Lage sein, diesen Header selbst zu deuten. Auf den Inhalt dieses Headers hat man auch in praktisch jedem Browser direkten Zugriff. Bei Mozilla (bzw. Netscape 6, 7, ...) geht das - wie bereits erwähnt - über Bearbeiten -> Voreinstellungen -> Navigator -> Sprachen, beim Internet Explorer über Extras -> Internet-Optionen -> Allgemein -> Sprachen. Man kann das nicht nur konfigurieren, es funktioniert sogar, wie man beispielsweise auf der Webseite http://www.debian.org/ testen kann. Je nachdem, wie der Browser eingestellt ist, wird die Seite in der bevorzugten Sprache ausgeliefert. Am Fuß der Seite findet sich eine Liste aller unterstützten Sprachen, und auch ein Link der weiterhilft, wenn die automatische Spracherkennung nicht funktioniert.

Mit dem Apache ist das keine Hexerei. Wir werden im Abschnitt 4, „Content-Negotiation“ über *"Content-Negotiation* lernen, wie wir es genauso schön hinbekommen.

1.7. Formulare

HTTP ist bidirektional, es lassen sich also nicht nur Daten vom Server auf den Client übertragen, sondern auch umgekehrt. In der Theorie gibt es für die umgekehrte Richtung zum Beispiel die Request-Methode PUT, mit der Dateien auf den Web-Server gespielt werden können, in der Praxis ist dies aus Sicherheitsgründen jedoch meist abgestellt. Relevant ist dagegen die Übertragung von Formulardaten mit der Methode POST (oder auch GET, wobei die Formularwerte durch ein Fragezeichen abgetrennt im URI übermittelt werden).

In welcher Sprache die übermittelten Daten vorliegen, ist in der Praxis wenig bedeutsam, das Charset dagegen spielt eine ziemlich entscheidende Rolle. Allerdings werden Formulardaten mit dem Medientyp `application/x-www-form-urlencoded` übermittelt, der keinen Parameter *`charset`* kennt.

Die Formulardaten werden einfach als Byte-Strom verschickt, wobei alle Sonderzeichen in der Form %*XX* kodiert werden (*XX* entspricht dabei dem Hexadezimalwert des jeweiligen Bytes). Sonderzeichen ist hier eigentlich der falsche Ausdruck, weil tatsächlich Bytes, also Binärdaten gemeint sind, die nur durch die Prozent-Form in 7-Bit-ASCII kodiert werden.

Die tatsächliche Kodierung der Daten entspricht dabei dem Charset der Seite, von der aus das Formular abgeschickt wurde. Geben wir in eine ISO-8859-1-Seite ein großes Ü ein, kommt beim Server die Folge %DC an (DC ist der Hexadezimal-Code des großen Ü). Stammt die Eingabe jedoch von einer Seite, die in UTF-8 kodiert ist, sieht das Skript, das auf dem Server die Daten verarbeitet, dagegen die Folge %C3%9C, denn in UTF-8 wird ein großes Ü durch zwei Bytes mit den Hexadezimalwerten C3 und 9C dargestellt.

Wie kann unser Skript aber nun feststellen, welches Charset gemeint ist? Die Antwort ist

relativ einfach: Gar nicht! Die Konsequenz ist, dass man *niemals* ohne weitere Vorkehrungen das gleiche `action`-Attribut in Formularseiten, die in unterschiedlichen Charsets kodiert sind, verwenden sollte, denn je nach Charset wird exakt der gleiche Text in unterschiedlicher Kodierung im auswertenden Skript ankommen.

Eine naive Entgegnung könnte lauten, dass der Web-Browser ja einen Referer-Header mitsendet, der die zuletzt besuchte Seite angibt. Dieser Header wird von vielen Browsern aber mittlerweile der informationellen Selbstbestimmung der Benutzer zuliebe unterdrückt, und lässt sich außerdem nach Belieben fälschen, und das bereits mit einfachsten Mitteln. Dem Inhalt dieses Headers sollte man daher immer mit sehr großer Skepsis begegnen.

Jede Hoffnung, eine bestimmte Kodierung mit Sicherheit annehmen zu können, wird man aber spätestens dann aufgeben, wenn man einmal in einem Formular die Seitencodierung von Hand abändert. Der Browser wird in diesem Falle zwar die Formulareingaben in der Regel auf die ursprünglichen Werte zurücksetzen, die Daten dann aber trotzdem in der manuell veränderten Kodierung übermitteln. Es bleibt zu hoffen, dass zukünftige Browser das manuelle Verstellen des Charsets auf Formularseiten wenigstens mit einer Warnung quittieren.

In der Praxis kann man dem Problem also nur mit Schulterzucken und einem lapidaren „selber schuld!“ begegnen. Allerdings sollte man zumindest dafür sorgen, dass bei normaler Benutzung der Web-Site nichts passieren kann. Ein Negativ-Szenario sähe so aus: Der Web-Auftritt hat einen deutschen, einen russischen und einen chinesischen Bereich. In jedem Bereich gibt es ein Kontaktformular, das eine Mail versendet, und für alle drei Kontaktformulare wird das gleiche Skript auf dem Server verwendet (also das gleiche action-Attribut im Element `form`). Die versendete Mail ist zumindest für Russisch und Chinesisch definitiv zerschossen, wird im Mail-Client des Empfängers also nur als Zeichenmüll ankommen. Abhilfe: Entweder für jedes Charset ein separates Skript verwenden, oder aber einen versteckten Parameter, der das angenommene Charset (es sollte ja bekannt sein, welches Charset das Formular zumindest haben *sollte*) mit übergeben.

Angesichts der grundsätzlichen Schwierigkeiten, die die Übermittlung von Nicht-ASCII-Daten in HTML-Formularen verursacht, mutet es direkt harmlos an, dass alle Browser neuerer Bauart in allen ISO-8859-1-Formularen ohne jeden Skrupel auch den Zeichenvorrat von CP1252 (Windows-1252) verwenden. Insbesondere beim Euro-Zeichen heißt es also höllisch aufzupassen, denn in Windows-1252 hat es den Hexadezimalcode 80, während es in ISO-8859-15 (also Latin-9) den Code A4 hat. Der Hexadezimal-Code A4 ist wiederum in ISO-8859-1 (Latin-1) das allgmeine Währungssymbol ¤.

Nur vermeintliche Abhilfe schafft das Attribut `accept-charset` des HTML-Elements `form`. Wert dieses Attributes ist eine durch Kommata oder Leerzeichen getrennte Liste erlaubter Charsets. Default für dieses Attribut ist der reservierte Wert `UNKNOWN` und dies ist kein Zufall. Auch die Interpretation des Attributs bleibt nämlich im Dunkeln. Wird als Wert des Attributs zum Beispiel ISO-8859-1 angegeben, ist es keineswegs sicher, ob der Browser Daten aus anderen Charsets ignoriert, ablehnt, mit einer wohlgemeinten Warnung an den Benutzer bedenkt, selber konvertiert, oder aber das Attribut als Ganzes schlichtweg ignoriert. Wie bei der Angabe einer Liste zu verfahren sei, er-

schließt sich genausowenig. Enthält das Attribut zum Beispiel den Wert `iso-8859-9, utf-8, koi8-r`, hilft das nicht wirklich weiter, denn die Sache wird nicht dadurch besser, dass nur noch zwischen drei sich gegenseitig ausschließenden Kodierungen gewählt werden muss. Die Verwendung des Attributes schadet sicher nicht, aber wirkliche Sicherheit bei der Interpretation der Formulardaten lässt sich auch damit nicht erreichen.

Spätestens jetzt sollte es jedem dämmern, weshalb Unicode erfunden wurde ... Sind alle Seiten, und damit auch Formulare, durchgehend in UTF-8 kodiert, gibt es natürlich auch keine Verwirrung, was die Kodierung der Formulardaten angeht. Dafür muss sich die Webmasterin jedoch gelegentlich mit Beschwerden herumschlagen, weil Leute mit älteren Browsern Schwierigkeiten haben, die Seiten zu lesen.

2. Extensible Markup Language XML

XML entstand wohl hauptsächlich aus der Motivation heraus, ein besseres HTML zu schaffen, und dürfte jedem, der sich bis hierhin durchgekämpft hat, ein Begriff sein. Ansonsten sei auf die Übersicht auf [XML-W3C] oder die offizielle Spezifikation [XML-Specification] verwiesen. Wir versuchen die für unser Thema interessanten Dinge also lieber direkt anhand eines konkreten Beispiels herauszubekommen:

```
<?xml version='1.0' encoding="iso-8859-1"?>                          ❶
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html lang="EN">
  <head>
    <title>XML-Beispiel</title>                                      ❷
  </head>
  <body>
    <h1>XML-Beispiel</h1>                                            ❷
    <img src="photo.png" alt="Blick aus meinem Fenster" />           ❸
  </body>
</html>
```

Dieses Beispiel hat den Vorzug, dass es sowohl gültiges XML als auch gültiges (X)HTML ist.

❶ Das Attribut *`encoding`* der XML-Deklaration bestimmt die Kodierung des Dokuments. Fehlt dieses Attribut, ist das Dokument in UTF-8, der Standardkodierung für XML kodiert.

❷ An diesen beiden Stellen enthält das Dokument Text im konventionellen Sinne, auch Content genannt.

❸ Der Wert des Attributs *`alt`* des Elements `img` ist insofern etwas Besonderes, als es ein Mittelding zwischen *Markup* (also Element- und Attributnamen) und Content darstellt.

Die Bestandteile eines XML-Dokuments lassen sich nach dem obigen Schema grob klassifizieren. Markup bildet das abstrakte Gerüst des Dokuments, die eigentliche Information ist in der Regel als Inhalt von Elementen vorhanden, Attributwerte können irgendwo dazwischen liegen. Worauf bezieht sich jetzt die Angabe `encoding` in der XML-Deklaration? Lediglich auf die eigentliche Information? Nein, auf das gesamte Dokument, inklusive Markup. Wer dabei jubelt hat es noch nie unternommen, einen Parser für XML zu schreiben, denn genau hier liegt einer der Gründe, weshalb XML-Applikationen so elend langsam, und daher in der Praxis selten so zu gebrauchen und einzusetzen sind, wie man beim W3C träumt.

Die Intention ist klar: Es soll möglich sein, XML-Formate in beliebigen natürlichen Sprachen zu definieren, während man sich bei derlei Unternehmungen normalerweise auf US-ASCII beschränkt. Die Krux des Features tritt allerdings zutage, wenn man tiefer in die Spezifikation einsteigt. So sind für Elementnamen so ziemlich alle Unicode-Zeichen außer *Whitespace* (nicht druckbare Zeichen wie Leerzeichen, Tabulatoren, Zeilenumbrüche, etc.) erlaubt. Dazu gehören allerdings auch ideographische Leerzeichen, also Zeichen, denen in Unicode die entsprechende Eigenschaft zugeordnet wurde.

2.1. Perfomance-Erwägungen

Welchen Unicode-Zeichen jedoch welche Eigenschaften zugeordnet werden, ist keineswegs für alle Ewigkeit klar. Bei einigen Zeichen gibt es Uneinigkeit über die Klassifikation, und die Zuordnung kann sich auch durchaus einmal ändern. Ein XML-Dokument, das heute noch gültig ist, könnte also nach einer Änderung des Unicode-Standards plötzlich ungültig sein. Selbst, wenn dies nicht so wäre, stellt das Parsen von XML aber noch immer eine echte Herausforderung dar, weil die für die einzelnen Bestandteile erlaubten und nicht-erlaubten Zeichen keineswegs in kontinuierlichen Bereichen liegen, sondern bunt über das Spektrum verteilt sind. Die Definition der sogenannten Basis-Zeichen in http://www.w3.org/TR/REC-xml#NT-BaseChar mag als abschreckendes Beispiel genügen.

Anhänger des XML-Hypes wenden gegen derlei Blasphemie gerne ein, dass eines der Design-Ziele von XML die Lesbarkeit durch Menschen sei, Knappheit ist ausdrücklich kein Design-Ziel. Die meisten XML-Dokumente werden allerdings auch dann nicht wirklich zur unterhaltsamen Lektüre, wenn man die Elementnamen ohne weiteres versteht ...

Der Wert von XML soll hier keineswegs pauschal in Abrede gestellt werden. Die hehren Ziele, die beim Design von XML verfolgt wurden, schränken allerdings die praktische Brauchbarkeit stark ein, und man sollte sich vor der Verwendung von XML darüber im Klaren sein, dass es sich bei den fantastischen Anwendungsmöglichkeiten meistens noch um Tagträumereien handelt, die in der Praxis einfach an Performance- und Lastproblemen scheitern. Eine gern beschriebene Anwendungsmöglichkeit sieht so aus, dass Web-Server alle Daten nur noch in XML vorhalten, und dann bei Bedarf durch ein Stylesheet jagen, und in fast beliebige Zielformate wandeln. Nun, *dieses* Dokument *ist* in XML geschrieben, und wird via XSLT in HTML und PDF gewandelt. Die Umwandlung in HTML dauert zur Zeit 35 Sekunden, wird zusätzlich noch eine PDF-Version generiert, dauert es schlappe anderthalb Minuten. Auf Webseiten, die mithilfe solcher Techniken generiert werden, sollte man es daher nicht versäumen, die Nummer eines Telefonan-

schlusses anzugeben, der von sachkundigen Mitarbeiterinnen betreut wird, die sich darauf verstehen, den ignoranten Besuchern der Webpräsenz die Vorzüge von XML zu vermitteln. An mangelnder Zeit dafür soll es nicht scheitern ...

Wie gesagt, XML kann sicher nutzbringend eingesetzt werden, es erleichtert insbesondere den Datenaustausch, nicht zuletzt weil die Kodierung von Textdaten stets eindeutig festzustellen ist. Seine Tauglichkeit als Speicherformat darf allerdings zum jetzigen Zeitpunkt bezweifelt werden. An Schnittstellen zu anderen Applikation ist XML sehr nützlich. Um Konfigurationsdateien, die ohnehin nur von der eigenen Software gelesen und verstanden werden, abzuspeichern, wird man sicher besser geeignete Formate finden.

2.2. XML in UTF-8

Ein großer Vorzug von XML ist die Eindeutigkeit der Daten. Allerdings wird die heile Welt durch ein zweifelhaftes Feature auf Windows-Systemen doch etwas getrübt. In Abschnitt 4.1.2, „UTF-16“ wurde das *Byte Order Mark* BOM erwähnt. In 2- und 4-Byte-Kodierungen ist diese Markierung nützlich, um die Byte-Ordnung der Daten zu ermitteln, für Multi-Byte-Encodings wie UTF-8 ist es überflüssig, und sollte ignoriert werden.

Dieser Umstand wird von Windows-Applikationen (der Minimal-Editor Notepad ist ein Beispiel hierfür) „missbraucht“, um beliebige Textdateien als utf-8-kodiert zu kennzeichnen, indem nämlich genau die UTF-8-Darstellung des BOM am Anfang der Datei eingefügt wird. Daran ist prinzipiell nichts auszusetzen, es ist ein cleverer Trick.

Für XML hat dies allerdings fatale Konsequenzen: Editiert man eine XML-Datei, die in UTF-8 kodiert ist (oder es sein sollte) mit einem solchen Editor, fügt dieser das Zeichen ebenfalls ein, obwohl dies nach dem XML-Standard nicht erlaubt ist (XML-Dateien müssen *immer*, sofern sie nicht in 2- oder 4-Byte-Kodierungen vorliegen, mit dem Zeichen „<“ beginnen); das BOM ist aber kein Kleinerzeichen.

Sowohl der Internet Explorer als auch Mozilla nehmen diese Verletzung des Standards stillschweigend hin, selbst der aus dem Unix-Bereich stammende Editor **vim** stört sich nicht daran, und interpretiert das BOM in Notepad-Manier (und speichert es auch wieder mit ab). Echte XML-Parser brechen jedoch bereits zu Anfang mit einem fatalen Fehler ab, und das zurecht.

Man mag das Problem als wenig relevant abtun, weil etliche Standard-Applikationen diesen Fehler ja akzeptieren. Problematisch ist jedoch, dass sich der Fehler aus den Dateien nicht mehr eliminieren lässt. Hat man eine XML-Datei einmal mit dem Notepad verunstaltet, wird es mit dem selben Programm nicht gelingen, den Fehler wieder zu entfernen. Erkennbar ist er ebenfalls nicht, weil die Programme, die den Standard an dieser Stelle ignorieren, die illegalen Zeichen ja noch nicht einmal darstellen.

Welche Programme sonst noch diesen Fehler verursachen, ist nicht bekannt. Es darf jedoch vermutet werden, dass das Notepad nicht der einzige Bösewicht ist und bleibt.

2.3. Sprachbestimmung

Die Bestimmung der Sprache geschieht in XML durch das Attribut `xml:lang`. Dieses spezielle Attribut ist unabhängig von der konkreten DTD *immer*, und mit der gleichen Semantik erlaubt. Inhalt des Attributs ist ein Sprachkürzel, wie es auch in HTTP verwendet wird, also beispielsweise de, en-us, fr-ch. Groß-Kleinschreibung spielt bei der Interpretation des Attributes keine Rolle.

Das Attribut gilt jeweils für das Element, in dem es verwendet wird und für alle Unterelemente. Unterelemente „erben" den Wert also. Mithilfe dieses hierarchischen Konzeptes ist es daher möglich, beliebige Teilbäume in verschiedenen Sprachen zu verfassen und entsprechend zu kennzeichnen (für das Charset fehlt eine solche Möglichkeit leider, obwohl dies in der Praxis, insbesondere bei der manuellen Erstellung von XML-Daten durchaus nützlich wäre).

Einen Standardwert für das Attribut `xml:lang` kennt XML ebensowenig wie HTTP.

Wird ein XML-Dokument über HTTP ausgeliefert (mit dem Medientyp `text/xml`) dürfte man davon ausgehen, dass die empfangende Applikation einen Content-Language-Header ebenfalls beachtet, und dessen Wert auf das komplette Dokument bezieht. Da alle gängigen Web-Browser die Dokumentensprache zur Zeit noch ignorieren, lässt sich diese Annahme jedoch nicht eruieren.

3. Hypertext Markup Language HTML

Bei HTML-Dateien stellen sich im Zusammenhang mit dem Thema dieses Dokumentes zwei Fragen. Wie kann ich die Sprache eines Dokumentes, und wie kann ich die Kodierung der Textdaten angeben?

3.1. Das HTML-Attribut *lang*

Die Kennzeichnung der Sprache beliebiger Teile eines HTML-Dokuments geschieht in der gleichen Form, wie bei XML, also durch Angabe eines `lang`-Attributs, allerdings ohne Angabe des XML-Namespaces (`xml:lang`) (siehe Abschnitt 2.3, „Sprachbestimmung").

3.2. Das Attribut *http-equiv* des meta-Elements.

Für die Angabe der Kodierung eines HTML-Dokumentes fehlt eine solche einfache Möglichkeit. Sie lässt sich nur über einen Umweg bewerkstelligen, über das Attribut `http-equiv` des `meta`-Elements:

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" /
<meta http-equiv="Content-Language" content="de-DE" />
```

Mit dem Attribut lassen sich also beliebige HTTP-Header „emulieren". Welche Bedeu-

tung hat dies aber konkret?

Der HTML-Standard ([HTML-4.01]) sieht eigentlich vor, dass der Web-Server diese Meta-Information auswertet und zur Erzeugung von HTTP-Headern verwendet. In der Praxis lassen die Web-Server (der Apache verhält sich jedenfalls so) genau dies jedoch aus gutem Grunde sein, weil es bedeuten würde, dass jedes Dokument vor der Auslieferung durch den Server nach den entsprechenden Meta-Informationen durchsucht werden müsste, was sich wiederum sehr negativ auf die Performance auswirkt.

Tatsächlich werten die Clients, also die Browser, die Informationen aus, und Autoren „missbrauchen" das Attribut dazu, fehlende Server-Informationen zu ergänzen oder zu überschreiben. Das zeigt aber auch das grundsätzliche Problem dieser Technik auf, weil es nämlich keineswegs klar ist, wie widersprüchliche Informationen zu interpretieren sind, wenn beispielsweise der Server behauptet, ein Dokument wäre in UTF-8 kodiert, während das Dokument selber die Angabe ISO-8859-1 macht.

Ein solcher Zustand lässt sich relativ einfach herbeiführen, wenn man nur etwas vorschnell entsprechende Konfigurationsdirektiven (zum Beispiel `AddDefaultCharset` oder `DefaultCharset`) in die Server-Konfiguration einfügt. Dieser Fehler liegt aber wenigstens noch im eigenen Verantwortungsbereich und lässt sich auch dort beheben. Gefährlicher sind schon die Schwierigkeiten, die sich bei Zwischenschaltung eines *Proxy-Servers* ergeben können, wenn das Dokument also über mehrere Stationen ausgeliefert wird. Ein Proxy-Server hat durchaus das Recht, Dokumente *on the fly* bei der Auslieferung zu konvertieren. Geht der Proxy beispielsweise davon aus (weil das Attribut *`charset`* im Header Content-Type fehlt), dass ein Dokument in ISO-8859-1 kodiert ist, tatsächlich ist es aber in KOI8-R kodiert, und wandelt der Proxy es unter dieser (falschen) Annahme in UTF-8 um, kommt das Dokument in unlesbarer Form beim Browser an.

Das beschriebene Verhalten des Proxy-Servers wäre völlig legal, aber natürlich nicht wünschenswert. In der Praxis wird man solche Fehler dadurch zu vermeiden versuchen, dass man den eigenen Web-Server sorgfältig konfiguriert, insbesondere also sicherstellt, dass alle notwendigen HTTP-Header mitgeliefert werden, und so zwischengeschaltete Proxy-Server nach Möglichkeit davon abhält, solche automatischen Umwandlungen durchzuführen.

In homogenen Umgebungen ist dies bei entsprechender Planung sicher immer möglich, in heterogenen Umgebungen aber mit einigen Schwierigkeiten verbunden. Ein Horrorszenario bietet sich in dieser Hinsicht zum Beispiel den Administratoren von Universitäts-Servern. Hier hat man es oft mit einer Vielzahl von eigenverantwortlich erstellten, sprich nach eigenem Gutdünken erzeugten Dokumenten zu tun, an eine einheitliche Namenskonvention ist nicht zu denken, eine korrekte Server-Konfiguration ist damit schlechterdings unmöglich. Einziger Ausweg ist hier der Einsatz eines leistungsfähigen Content-Management-Systems, das die notwendige Homogenität sicherstellt, und dennoch das wünschenswerte Maß an Gestaltungsfreiheit bietet. Glücklicherweise gibt es da ja schon ein gutes ... ;-)

In der Praxis hat es sich bewährt, in dieser Hinsicht nach dem Grundsatz „doppelt gemoppelt ..." vorzugehen. Einerseits sollte man durch die entsprechende Serverkonfigu-

ration die korrekte Kennzeichnung des Contents sicherstellen, andererseits schadet es aber auch nicht, über *http-equiv* noch eins draufzulegen, und bei Manipulation des Contents auf dem Übertragungsweg zum Browser zumindest darauf zu hoffen, dass der die Informationen, die direkt aus dem Dokument gewonnen werden, mit Vorrang behandelt.

3.3. XHTML

XHTML (siehe dazu [XHTML-1.0]) ist eine Neuformulierung von HTML in XML, gültige XHTML-Dokumente sind also auch stets gültige XML-Dokumente (aber natürlich nicht umgekehrt). Gültige XHTML-Dokumente sind - von Details abgesehen - aber auch gültiges HTML. Insofern gilt das für HTML gesagte weitgehend analog.

Was die Dokumentkodierung angibt, ergibt sich bei XHTML aus dem Erfordernis der XML-Konformität natürlich, dass das Dokument in dieser Hinsicht dem XML-Standard genügt, also entweder in UTF-8 kodiert ist, oder eine abweichende Kodierung im Attribut *encoding* der XML-Deklaration entsprechend angegeben ist. Ältere Browser werden die XML-Deklaration jedoch als sogenannte *Processing Instruction* ignorieren, und bekommen von der Kodierung also auch nichts mit. Daraus ergibt sich dann allerdings die Notwendigkeit, die Kodierung zusätzlich mit den konventionellen Mitteln von HTML und HTTP anzugeben.

Für die Sprache empfiehlt das W3C die Angabe von *zwei* Attributen, sowohl von *lang* (für ältere Clients) und *xml:lang*, wobei das Attribut mit Namespace-Angabe Vorrang genießt.

4. Content-Negotiation

Aus Abschnitt 1, „Hypertext Transfer Protocol HTTP“ über HTTP haben wir erfahren, wie schön das Web sein könnte. Browser beschreiben ihre Wünsche mit Accept-Headern, der Server wählt liebevoll das geeignete Dokument aus, kennzeichnet es durch Content-Header und schickt die Daten in genau der Form zurück, in der der geneigte Besucher der Webseite sie am liebsten sieht. Auf derlei leere Versprechungen fällt kein realistisch denkender Mensch hinein, aber dank des Web-Servers Apache und des Server-Moduls `mod_negotiation` funktioniert genau dies fast reibungslos, und dankenswerterweise völlig kostenlos.

Der Nutzen liegt auf der Hand, die Informationen werden in optimaler Form ausgeliefert, der Aufwand dafür hält sich stark in Grenzen, eine simple Namenskonvention reicht.

4.1. Implementierung mit dem Apache

Gehen wir von einer Web-Site aus, die in Englisch, Deutsch, Russisch und Chinesisch verfügbar sein soll. Die Dokumente in Englisch und Deutsch sind - so wollen wir hier als Beispiel annehmen - in CP1252 (Windows-1252/Windows-Westeuropäisch) kodiert, die russischen Seiten liegen in KOI8-R vor, der chinesische Content in Unicode als UTF-8.

In Abschnitt 1, „Hypertext Transfer Protocol HTTP“ haben wir bereits gelernt, wie wir über eine entsprechende Namenskonvention erreichen können, dass der Web-Server immer die richtigen Header für Content-Language und Content-Type mitschickt. Nehmen wir an, wir hätten folgende Zeilen in unsere Serverkonfiguration aufgenommen:

```
AddLanguage en .en                 ❶
AddCharset windows-1252 .en        ❷
AddLanguage de .de                 ❸
AddCharset windows-1252 .de        ❹
AddLanguage ru .ru                 ❺
AddCharset koi8-r .ru              ❻
AddLanguage zh .zh                 ❼
AddCharset utf-8 .zh               ❽
Options +MultiViews                ❾
```

❶ Hier legen wir fest, dass alle Dateien, die auf `.en` enden, in englisch verfasst sind.
❷ Alle englischen Dokumente sind in Windows-1252 kodiert (pfui!).
❸ Deutsche Dokumente enden auf `.de` ...
❹ ... und liegen ebenfalls in Windows-1252 vor.
❺ Russische Dokumente (`*.ru`) ...
❻ ... sind in KOI8-R kodiert.
❺ Chinesische Dokumente (`*.zh`) ...
❻ ... in UTF-8.
❾ Und schließlich das Zauberwort `MultiViews`.

Hinsichtlich der Namenskonvention sind nur zwei Dinge bemerkenswert: Was die Sprachzuordnung angeht, ist es sehr wahrscheinlich, dass genau diese Anweisungen bereits in der Server-Konfiguration enthalten sind. Die Zuordnung zu Charsets entspricht dagegen genau unserer (fiktiven) Umgebung. Wir *wissen*, dass alle unsere chinesischen Dokumente in UTF-8 kodiert sind. Wir können daher Namensungetüme wie `index.html.zh.utf-8` umgehen, und die Sprachzuordnung gleichsam für das Charset missbrauchen.

Wenn wir nun beispielsweise eine Datei mit dem Namen `testpage.html.ru` auf dem Server anlegen, und mit einem Tool, das die HTTP-Header bewahrt, von dort holen, werden wir (nach einem Neustart bzw. Neuladen der Konfiguration des Apache) feststellen, dass der Server die gewünschten HTTP-Header geliefert hat:

```
Content-Type: text/html; charset=koi8-r
Content-Language: ru
```

Der spannende Teil kommt allerdings noch. Dazu müssen wir unser Dokument in allen vier Versionen erzeugen, also die Dateien testpage.html.en, testpage.html.de, testpage.html.ru und testpage.html.zh. In jede Datei sollten wir zur eindeutigen Wiedererkennung etwas wie „Ich bin die englische Version.“ oder „Ich bin die russische Version.“ hineinschreiben (für unsere Testzwecke müssen die Dateien weder tatsächlich in der vermeintlichen Sprache verfasst sein, noch müssen sie das entsprechende Charset haben). Wir rufen die vier Dateien jetzt jeweils mit einem Web-Browser ab, und überprüfen so, dass wir die richtige Adresse verwenden.

Und jetzt ist endlich Weihnachten: Wir lassen beim Abruf der Datei die Endung mit dem Sprachkürzel einfach weg, fordern also statt http://meinserver/testpage.html.de die Seite http://meinserver/testpage.html ab. Und, siehe da, obwohl die Datei gar nicht existiert, kommt eine Antwort, genaugenommen eine der von uns erzeugten Dateien. Welche das ist, hängt von der Browserkonfiguration ab.

Testweise verstellen wir unsere Spracheinstellungen im Browser (bei Mozilla bzw. Netscape 7 über Bearbeiten -> Voreinstellungen -> Navigator -> Sprachen, beim Internet Explorer über Extras -> Internet-Optionen -> Allgemein -> Sprachen), schieben Deutsch, Englisch, Russisch und Chinesisch hoch und runter und sehen, wie der Server unsere neuen Sprachvorlieben umgehend honoriert, und jeweils das Dokument in der gewünschten Sprache ausliefert.

Damit das funktioniert müssen verschiedene Voraussetzungen erfüllt sein. Gerne vergessen, insbesondere, wenn unsere Datei index.html.* heißt: Die eigentlich „gemeinte“ Datei darf nicht existieren. Wollen wir also Content-Negotiation für eine Datei index.html ermöglichen, dann darf genau diese Datei am entsprechenden Ort auf dem Server nicht existieren. Stattdessen dürfen nur die Dateien mit den entsprechenden Sprachzusatz vorhanden sein.

Weiterhin muss das Server-Modul mod_negotiation in den Server einkompiliert und aktiviert sein. Wenn man es nicht explizit abgeschaltet hat, sollte dies jedoch der Fall sein.

Und schließlich müssen die Endungen, die wir für Content-Negotiation verwenden, wirklich am *Ende* des Dateinamens stehen. Hätten wir zum Beispiel zusätzlich die Endung .utf-8 für UTF-8-Dateien konfiguriert, wäre es prinzipiell egal, ob die tatsächliche Datei nun als testpage.html.zh.utf-8 oder testpage.html.utf-8.zh auf dem Server liegt. Sofern die Endung .html jedoch nicht ebenfalls der Content-Verhandlung zwischen Client und Server unterliegt, funktioniert der Mechanismus für testpage.utf-8.zh.html nicht.

4.2. Praktische Erwägungen

4.2.1. Sprachpersistenz

Bei aller Begeisterung für das neu erlernte Feature sollte man es jedoch vermeiden, gleich übers Ziel hinauszuschießen. Dies lässt sich allerdings durch ungeschickte Verlin-

kung innerhalb der Site leicht erreichen. Wir hatten ja bereits festgestellt, dass eine Voraussetzung für die erfolgreiche dynamische Bestimmung des Contents darin besteht, dass die vom Browser eigentlich angeforderte Datei auf dem Server nicht existiert (insofern kann man sich Content-Negotiation als eine Art Fehlerbehandlung vorstellen). Dies ist tatsächlich keine Einschränkung, sondern ein Feature, denn dadurch lässt sich leicht erreichen, dass die einmal gewählte Sprache beibehalten wird. In englischen Dokumenten sollte man stets auf die jeweils englische Version anderer Dokumente verlinken, um den Benutzer nicht durch ständiges Wechseln der Sprache zu verwirren. In einer nach obigem Schema konzipierten Site würden Links aus englischen Dokumenten also nach `/anderes/verzeichnis/seite.html.en` und nicht nach `/anderes/verzeichnis/seite.html` zeigen. Dadurch, dass der Dateiname im Link vollständig angegeben ist, wird der Server auf Content-Negotiation verzichten, und ohne jede weitere Konfiguration die gewählte Sprache beibehalten.

4.2.2. Sprachumschaltung

Auf die gleiche Art und Weise lässt sich eine Sprachumschaltung realisieren. Es muss lediglich ein expliziter Link auf die anderen Sprachversionen ins Dokument eingefügt werden. Hierbei sollte man sich Tricks mit JavaScript tunlichst verkneifen, weil dies niemandem viel nützt, allerdings Besucherinnen mit softwaretechnischen und erst recht solchen mit körperlichen Nachteilen (Stichwort Barrierefreiheit) sehr schadet.

Besonders schlecht ist eine Realisierung der Sprachumschaltung mit Hilfe von *onchange*-Handlern für `select`-Elementen. Selbst wenn man in Kauf nimmt, dass die eigene Web-Site nur mit JavaScript bedienbar ist (und damit körperlich Behinderte faktisch von der Benutzung ausschließt), hat dies einen ganz entscheidenden weiteren Nachteil: Nicht nur Behinderte, auch Suchmaschinen verfügen nicht über JavaScript. Ist der Content in anderen Sprachen aber nur über JavaScript (die Verwendung von Cookies ist in diesem Kontext genauso schlecht) erreichbar, und nicht über reguläre Links, führt das dazu, dass die Teile der Web-Site, die in anderen Sprachen verfügbar sind, von externene Suchmaschinen nicht indiziert werden können.

4.2.3. Vertikale oder horizontale Aufteilung

Aus dem bisher Gesagten, insbesondere den Hinweisen zur Sprachpersistenz ergibt sich, dass es prinzipiell ausreichend ist, lediglich an wohldefinierten Ansprungstellen der Web-Site Content-Negotiation zu aktivieren. Innerhalb fester Content-Bereiche ist es eher verwirrend, wenn die Sprache gewechselt wird.

Daraus ergibt sich allerdings auch die Möglichkeit eines prinzipiell anderen Aufbaus der Site. Es wäre zum Beispiel denkbar, dass man lediglich an den Ansprungstellen die oben beschriebene Namenskonvention einhält, und von diesen Ansprungstellen aus, in abgeschlossene Bereiche verzweigt, also zum Beispiel Teilbäume unterhalb von Verzeichnissen `en`, `de`, `ru` und `zh`.

Welche Struktur vorzuziehen ist, hängt von den konkreten Gegebenheiten und Anforderungen ab. In der Regel ist ein Aufbau mit getrennten Verzeichnisbäumen eher komplizierter: Ein Link von einem (englischen) Dokument `/en/a/b/c/index.html` auf die

entsprechende deutsche Version `/de/a/b/c/index.html` erfordert stets Kenntnis des kompletten Pfades. In statischen Webauftritten kann genau das jedoch zu erheblichen Problemen führen. Auch eine Verschiebung von Teilbereichen innerhalb der Site wird bei einem solchen Aufbau erheblich erschwert. Liegen die unterschiedlichen Sprachversionen jedoch immer innerhalb desselben Verzeichnisses, genügt stets ein konstanter Link auf die jeweils anderen Sprachen. Ein Link auf `./index.html.de` funktioniert auch dann noch, wenn die Website komplett umstrukturiert wird.

5. Zusammenfassung

Bei der praktischen Umsetzung mehrsprachiger Web-Sites kommt der eindeutigen Kennzeichnung von Dokumentensprache und Dokumentenkodierung große Bedeutung zu. Dies sollte bevorzugt durch eine sorgfältige Konfiguration des Web-Servers sichergestellt werden. Für HTML-Dokumente können diese Informationen, hilfsweise, zusätzlich noch innerhalb der Dokumente vorgehalten werden.

Mit Content-Negotiation lässt sich die Erreichbarkeit von Content über einfache Namenskonventionen stark vereinfachen, weil bereits durch einfache Maßnahmen eine Auslieferung des Contents entsprechend den Vorlieben der Besucher erreicht werden kann.

Teil III. Unicode und Multi-Language mit Imperia

Inhaltsverzeichnis

4. Allgemeines 63
1. Copy-Pages 63
2. Unicode-Unterstützung in Imperia 63
2.1. System- und User-Charset 64
2.1.1. Wirkung der Sprach- und Charseteinstellungen 64
2.1.2. Stolpersteine 65
2.1.3. Templates 66
2.1.4. Meta-Dateien 67
2.1.5. Konfiguration des Web-Servers 67
2.2. Best Practice - eine Empfehlung 67
2.3. Weitere Unicode-Features 68
5. Ein Beispielszenario 69
1. Die Anforderungen 69
2. Vorbereitung 69
2.1. System-Einstellungen 70
2.2. Template und Meta-Datei 70
2.3. Eine Test-Rubrik 72
3. Basis-Implementierung 72
3.1. Mehrsprachige Eingabe 72
3.1.1. Verallgemeinerung der Spracheingaben 73
3.1.2. Parametrisierte Code-Includes 73
4. Dynamische Sprachwahl 75
4.1. Anpassung der Meta-Datei 75
4.2. Ein einfaches Workflow-Plug-In 77
4.2.1. Gerüst für das Plug-In 78
4.2.2. Workflow-Definition 78
4.2.3. Implementierung der Plug-In-Logik 79
4.3. Dynamische Einbindung der Code-Includes 82
5. Reduzierung der Templates 83
6. Charset-Konvertierung 85
7. Verfeinerungen 87
7.1. Problemfall Flexmodule 87
7.2. Der Flex-Flexxer 88
7.3. Mehr Komfort durch DHTML 90
7.3.1. Erzeugen der Layer 91
7.3.2. Die DHTML-Erweiterungen am Template 92
7.3.3. Das Layer-Menü 93
8. Zusammenfassung 94

Kapitel 4. Allgemeines

Mit Hilfe Imperias ist es bereits seit langem möglich, mehrsprachige Web-Sites zu erstellen und zu warten. Das Kern-Feature, dass diese Möglichkeiten erlaubt, sind die sogenannten *Copy-Pages*. Die Unicode-Unterstützung in Imperia ist dagegen ein relativ junges Feature, das erst mit Imperia 6.1 eingeführt wurde.

1. Copy-Pages

Imperia speichert Content und Meta-Informationen über Dokumente intern in sogenenannten Meta-Variablen. Dabei macht das System keinen Unterschied zwischen tatsächlichem Content, und dem, was klassischerweise als Meta-Information eines Dokumentes verstanden wird (Titel, Schlüsselwörter, URI, etc.). Was tatsächlich in den publizierten Dokumenten landet, wird alleine durch das Template bestimmt, mit dessen Hilfe gezielt Meta-Informationen aus dem das Dokument konstituierenden Datensatz extrahiert, und in die zu publizierenden Dateien eingesetzt werden können. Imperia ist jedoch in der Lage, während des Publikationsprozesses mehrere Templates sukzessive auf den gleichen Datensatz anzuwenden, und somit mehrere Dateien gleichzeitig aus dem jeweils gleichen Datenbestand zu erzeugen.

Eine klassische Anwendung für Copy-Seiten sind Druckversionen von HTML-Dateien, bei denen der Content einmal für das Web, und parallel dazu in einer für den Druck optimierten Version publiziert wird. Im Regelfall werden beide Dokumente dabei jedoch ähnlichen, wenn nicht sogar identischen Content enthalten, und sich lediglich mehr oder weniger stark im Layout unterscheiden.

In Multi-Language-Anwendungen wird das gleiche Feature in unterschiedlicher Weise eingesetzt. Solange das Dokument in Bearbeitung ist, werden alle Sprach-Versionen über ein einziges, sogenanntes *Master-Template* eingepflegt. Für die Publikation werden dagegen sprachspezifische Templates eingesetzt, die jeweils nur auf einem Teilbereich des Contents operieren, gleichsam sprachspezifische *Views*, Ansichten auf den mehrsprachigen Datenbestand generieren. Für den klassischen Fall der Web-Publikation entstehen dabei zwar physikalisch getrennte Dateien, die jedoch von Imperia logisch als eine Einheit betrachtet werden.

2. Unicode-Unterstützung in Imperia

Intern behandelt Imperia alle Meta-Informationen und Assets als (8-Bit-)Binärdaten, macht also keinerlei Annahmen darüber, in welchem Codeset textuelle Daten eingegeben oder gespeichert werden. Diese Vorgehensweise erlaubt die größtmögliche Flexibilität, da sie nicht nur die gleichzeitige Speicherung von Informationen in beliebigen Kodierungen, sondern daneben sogar die zwanglose Verwaltung von echten Binärdaten erlaubt. Eine Klassifizierung der Meta-Informationen kann aber jederzeit über geeignete Namenskonventionen für Meta-Variablen erreicht werden, auch wenn die Praxis zeigt, dass dieser Aufwand meist gar nicht nötig ist.

Interne Informationen (zum Beispiel Menü-Titel, Grid-Parameter oder aber auch die Beschriftungen von Buttons in der OneClickEdit-Toolbar) speichert Imperia mittlerweile bevorzugt in UTF-8 ab, und wandelt sie erst für die Anzeige in ein geeignetes Codeset um. Es soll allerdings nicht verschwiegen werden, dass dieser zu bevorzugende Weg von Imperia noch nicht durchgehend beschritten wird, eine Tatsache, die bei der Umsetzung von Projekten bereits frühzeitig bedacht werden sollte.

2.1. System- und User-Charset

Jeder in Imperia angemeldete User besitzt die Möglichkeit, in seinen persönlichen Einstellungen sowohl die Sprache, als auch das zur Anzeige verwendete Charset frei zu wählen. Trifft ein einzelner User keine Auswahl, werden stattdessen die systemweiten Einstellungen (über den Menüpunkt „Allgemeine Einstellungen“ zugänglich) verwendet. In der Regel wird es sinnvoll sein, hier eine funktionierende systemweite Einstellung zu wählen, um Fehlkonfigurationen der Redakteure vorzubeugen. In zukünftigen Imperia-Versionen wird es genau aus diesem Grunde möglich sein, eine abweichende persönliche Konfiguration durch die einzelnen User zu unterbinden. Eine ungeeignete Einstellung kann faktisch dazu führen, dass sich User aus dem System aussperren, weil sie zum Beispiel als Sprache Russisch und als Charset Latin-1 wählen. Da Latin-1 keinerlei russische Schriftzeichen enthält, wird sich die Imperia-Oberfläche daraufhin jedoch mit Fragezeichen (der Ersatzdarstellung für nicht-darstellbare Zeichen) übersät präsentieren, was eine Korrektur der Fehleinstellung natürlich stark erschwert.[2]

Natürlich wäre es möglich, solcherlei Fehleingaben abzufangen, allerdings nur auf Kosten der Flexibilität des Systems. Da die Software in beliebige Sprachen übersetzt werden kann, ist es schlechterdings unmöglich, mit Softwaremitteln zu entscheiden, ob eine bestimmt Kombination aus Sprache und Charset zu einer sinnvollen Darstellung führt. Die in der Praxis nicht relevante Gefahr, dass bestimmte Einstellungen zu temporären Einschränkungen in der Bedienbarkeit führen, wird daher bewusst in Kauf genommen. Der jetzige Zustand ist ohnehin lediglich als Übergang zu betrachten. Mittelfristig wird Imperia intern komplett auf Unicode umgestellt, sobald davon ausgegangen werden kann, dass alle unterstützten Browsertypen und -versionen Unicode problemlos verarbeiten können.

2.1.1. Wirkung der Sprach- und Charseteinstellungen

Welche Wirkung haben diese Einstellungsmöglichkeiten aber nun, wo Imperia Content doch in neutraler Binärform behandelt? Tatsächlich wird von diesen Parametern lediglich die Darstellung der Imperia-Oberfläche beeinflusst. Als Web-Applikation erzeugt Imperia selbst ja permanent HTML-Seiten bzw. HTML-Formulare, die über HTTP zum Browser übertragen werden. Die Charset-Einstellung bewirkt dabei zweierlei: Erstens werden alle lokalisierten Texte der Imperia-Oberfläche in dieses Charset konvertiert, und zweitens sorgt das System durch die Übermittlung der korrekten HTTP-Header dafür,

[2] Eine Zurücksetzung auf eine sinnvolle Einstellung ist natürlich weiterhin möglich, da die entsprechenden Menüpunkte noch immer über aussagekräftige Icons erreichbar sind. Die Sprach- und Charsetauswahl wiederum wird von Imperia nicht in die bevorzugte Sprache übersetzt, und die entsprechenden Drop-Downs sind ausschließlich mit ASCII-Zeichen beschriftet, wordurch die Darstellbarkeit in allen unterstützten Charsets gewährleistet ist.

dass die Browser diese Texte dann auch richtig darstellen.

Eine wichtige Ausnahme stellen die Seiten dar, die direkt aus Templates generiert werden. Bei diesen Seiten übermittelt Imperia keine entsprechenden HTTP-Header, um den Templateprogrammierern die volle Freiheit bei der Wahl des Dokumenten-Charsets zu überlassen. Wie dies im Detail erfolgen sollte, wird weiter unten noch beschrieben.

Dem bisher Gesagten mag man entnehmen, dass eine Fehlkonfiguration des Charsets lediglich zu Unregelmäßigkeiten in der Darstellung der Oberfläche führen kann. Dem ist leider nicht so, weil Imperia bidirektional mit den Benutzern interagiert: Das System übermittelt (Text-)Daten zum Benutzer, die Benutzer übermitteln umgekehrt (Formular-)Daten zurück zum System. Der Rückkanal birgt dabei einige Schwierigkeiten.

Die Übermittlung von Daten vom Benutzer/Client zum Imperia-Server erfolgt durchweg über HTML-Formulare. Wird in Imperia zum Beispiel UTF-8 als Charset eingestellt, führt dies dazu, dass Imperia dieses Charset auch in den HTTP-Headern der HTML-Seiten übermittelt, worauf die Browser entsprechend reagieren (sollten): Geht der Browser davon aus, dass die HTML-Seite in UTF-8 kodiert ist, wird er auch die Formulareingaben in UTF-8 zum Server zurückschicken. Bei einem UTF-8-Formular wird der Browser also für das Zeichen „ä" ein Multi-Byte-Sequenz in UTF-8 zurücksenden. Geht der Browser dagegen von Latin-1 aus, „sieht" Imperia ein 8-Bit-Zeichen (ein Byte, die „Hausnummer" des „ä" in Latin-1). Dieses Verhalten erweist sich an einzelnen Stellen, an denen Imperia aus Kompatibilitätsgründen intern noch nicht auf Unicode umgestellt wurde, als erheblicher Stolperstein.

2.1.2. Stolpersteine

In erster Linie treten diese Schwierigkeiten noch bei der Datenhaltung für die Rubrik- und User- bzw. Rolleninformationen auf, genauergesagt bei allen Daten, die direkt oder mittelbar in Templates ausgewertet werden können. Diese Schwierigkeiten lassen sich derzeit nicht vermeiden, da ansonsten die Abwärtskompatibilität des Systems nicht mehr gewährleistet wäre.

Konkret stellt sich das Problem so dar: User U hat als System-Charset UTF-8 eingestellt, und erzeugt eine Rubrik, die deutsche Umlaute im Namen enthält, und legt weiterhin einen neuen User L an, und vergibt für L auch gleich ein Passwort, ebenfalls mit Umlaut. Der Browser wird die entsprechenden Formulardaten in UTF-8 übermitteln, die Umlaute kommen bei Imperia also als Multi-Byte-Sequenzen an, und Imperia wird sie - so wie sie sind - als Binärdaten abspeichern.

Wenn User L sich jetzt versucht, in Imperia einzuloggen, kommt es zum ersten Problem: Imperia hat das Passwort in UTF-8 gespeichert, übermittelt wird es jetzt jedoch in Latin-1, dem Charset, das für User L eingestellt wurde. Der (binäre) Vergleich der Passwörter wird fehlschlagen, und als Konsequenz kann User L sich nicht erfolgreich anmelden. Nicht nur aus diesem Grunde sollte man sich bei Passwörtern grundsätzlich auf ASCII-Zeichen beschränken.

Das nächste Problem, dem L sich gegenübersieht, ist weniger kritisch, aber noch immer

ärgerlich: Auch die Informationen der Rubrik, die von U angelegt wurde, sind in Unicode (bzw. UTF-8) abgelegt worden. Die Anzeige des Rubrikenbaums für L erfolgt jedoch in Latin-1. Sonderzeichen in UTF-8 sind Multi-Byte-Sequenzen, diese werden von Imperia ohne Modifikation übertragen, und L wird statt dieser Sonderzeichen nur eine Folge von zwei bis sechs 8-Bit-Zeichen sehen.

Die Moral der Geschichte ist, dass administrativ unbedingt sichergestellt werden sollte, dass die Wahl des Charsets einheitlich erfolgt. Dies bedeutet nicht zwangsläufigerweise, dass *alle* User mit dem gleichen Charset arbeiten müssen; denkbar wäre zum Beispiel, bei allen Mitgliedern der deutschen Redaktion ISO-8859-1 einzustellen, bei allen Mitgliedern der russischen Redaktion dagegen KOI8-R. Für diesen Fall muss allerdings entweder durch eine geeignete Rechtevergabe sichergestellt werden, dass die Mitglieder der einen Redaktion die Daten der anderen gar nicht erst sehen, oder aber zumindest darüber aufgeklärt werden, dass die (Read-Only-)Daten der jeweils anderen Redaktion „entstellt“ übermittelt werden.

2.1.3. Templates

Auf template-basierte Seiten hat die Wahl des in Imperia konfigurierten Charsets *keinen* Einfluss, und dies aus gutem Grund. Das System erlaubt die Erzeugung von Dokumenten in beliebigen, auch innerhalb einer Imperia-Installation voneinander abweichenden Charsets, und deshalb übermittelt das System hier keine entsprechenden HTTP-Header; die Charset-Information entnimmt der Browser lediglich den Informationen aus dem Template selbst. Hieraus ergibt sich, dass Templates *grundsätzlich* ein entsprechendes Meta-Tag enthalten sollten, z. B.

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
```

Bei neuen Projekten sollte als Charset im (Eingabe-)Template vorzugsweise UTF-8 verwendet werden, da dies dazu führt, dass Imperia auch den eingegebenen Content in UTF-8 abspeichert. Wir werden später sehen, dass es leicht möglich ist, die Daten - völlig unabhängig von der Wahl des Charsets im Eingabetemplate - später in beliebigen anderen Charsets zu publizieren. Umgekehrt gilt dies leider nicht: Hat man sich einmal im Template auf beispielsweise Latin-1 festgelegt, ist es später nicht mehr ohne weiteres möglich, das Charset zu wechseln oder evtl. weitere Sprachen zu unterstützen, die nicht mehr in Latin-1 dargestellt werden können. Unicode steht für das *uni*verselle *Code*set, aus dem praktisch alle anderen Kodierungen erzeugt werden können. Wenn irgend möglich, sollte man diese universelle Wiederverwendbarkeit auch ausnutzen.

Aber auch bei der Wahl des Template-Charsets kann es zu vereinzelten (unangenehmen) Wechselwirkungen mit der Wahl des System- bzw. User-Charsets kommen: So ist die Editieransicht teilweise mit Elementen der Imperia-Bedienoberfläche durchsetzt, so zum Beispiel bei den Flexmodul-Boxen oder auch den Tooltips für die Buttons zum Speichern oder die Voransicht. Diese Elemente werden entsprechend den Charset-Einstellungen dargestellt, da Imperia (noch) nicht erkennen kann, auf welches

Charset ein Template ausgelegt ist. Typisches Beispiel ist die Beschriftung „Ausführen“ in den Flexmodul-Boxen. Weicht das Imperia-Charset vom Template-Charset ab, wird es hier zu (harmlosen!) Fehldarstellungen kommen.

Problematischer wird es bei Anzeigen, die aus Imperia-Strukturdaten gewonnen werden, z. B. User- oder Rubrikdaten:

```
Bearbeiter: <!--USER_CONF:fname--> <--USER_CONF:name-->
Rubrik: <--SECTION:NAME:0-->
```

Auch hier vermag Imperia weder festzustellen, in welchem Charset diese Daten eingepflegt wurden, noch in welchem Charset sie ausgegeben werden sollen, und wird sie deshalb unmodifiziert als Binärdaten einfügen. Sollte jedoch das Charset des Eingabetemplates vom in Imperia eingestellten Charset abweichen, wird es hier zwangsläufig zu Fehlern kommen, sobald die entsprechenden Daten nicht aus dem Zeichenvorrat von US-ASCII stammen.

2.1.4. Meta-Dateien

Meta-Dateien werden von Imperia im konfigurierten Charset dargestellt. Es ist also darauf zu achten, dass beschreibende Zusatztexte ebenfalls in diesem Charset kodiert sind. Andererseits werden aus Meta-Dateien natürlich auch Daten über Usereingaben gewonnen, die dann ebenfalls im eingestellten Charset abgespeichert werden. Werden diese Daten später im Template verwendet, ist dementsprechend sicherzustellen, dass sie im Template-Charset darstellbar sind.

2.1.5. Konfiguration des Web-Servers

Wie bereits erwähnt, trifft Imperia alle Vorkehrungen, um die HTML-Dateien bzw. Formulare im redaktionellen Workflow so zu übermitteln, dass der Browser das erwartete Charset korrekt interpretieren kann. Damit dies auch funktioniert, ist es natürlich unbedingt erforderlich, dass der Web-Server die entsprechenden Daten auch unverändert übermittelt, die jeweiligen HTTP-Header also nicht überschreibt. Insbesondere der Web-Server Apache wird jedoch oft so ausgeliefert, dass ein Standard-Charset für alle ausgelieferten Daten eingestellt wird (siehe hierzu Abschnitt 1.2.1, „Die Konfigurationsanweisung AddDefaultCharset“), was diese Bemühungen zunichte macht. Sollte Imperia trotz aller Bemühungen also die eigenen HTML-Dateien/-Formulare nicht den Einstellungen entsprechend ausliefern, hilft hier oft die Direktive **`AddDefaultCharset Off`** in der Serverkonfiguration.

2.2. Best Practice - eine Empfehlung

Die Ausführungen der vorhergehenden Abschnitte lassen erahnen, dass die Vermischung verschiedener Charsets zu teilweise recht erheblichen Problemen führen kann. Wann immer dies möglich ist, sollte man sich daher von vorneherein auf ein geeignetes

Charset festlegen, und dieses *durchgehend* in Imperia verwenden. Prinzipiell kann dies jedes Charset sein, sofern es nicht mit der bevorzugten Benutzersprache in Konflikt steht, aus praktischen Erwägungen sollte jedoch UTF-8 hier stets bevorzugt in Betracht gezogen werden, selbst für den klassischen Fall einer rein deutschsprachigen Site. Unicode bzw. UTF-8 als universelles Charset lässt sich verlustfrei in jedes beliebige andere Charset umwandeln, Imperia liefert schon mit Bordmitteln alle notwendigen Werkzeuge hierfür, eine Festlegung auf UTF-8 stellt also gerade *keine* Festlegung dar, sondern lässt für die Zukunft alle Freiheiten, sei es für die Erweiterung des Auftritts um weitere Sprachen oder auch für die Syndizierung des eigenen Contents in sprachneutraler Form. Wurde als einheitliches Charset UTF-8 gewählt, beschränkt sich die Konvertierung des kompletten Auftritts auf eine triviale Änderung der Templates und einen Lauf des Template-Reparsers, während die kurzsichtige Festlegung auf (veraltete) 8-Bit-Zeichensätze doch zu mehr oder weniger schmerzhaften Migrationsprozessen bei der Annahme neuer Herausforderungen führen kann.

2.3. Weitere Unicode-Features

Der vorhergehende Abschnitt sollte verdeutlicht haben, dass Imperia intern über sehr mächtige Funktionen zur Verarbeitung und Konvertierung textueller Daten in beliebigen Charsets verfügt. Es vermag daher nicht zu überraschen, dass diese Funktionen an verschiedenen Stellen des Systems „durchscheinen“ und auch für eine redaktionelle Verwendung nutzbar gemacht werden. Dies reicht von Erweiterungen der Template-Syntax (beispielsweise der lokalisierten Generierung von Datum bzw. Zeit über die Template-Direktive **`Xstrftime`**) bis hin zur automatischen Konvertierung von Content zwischen beliebigen Charsets. Diese Features sind in den mit der Software ausgelieferten Handbüchern dokumentiert, und werden teilweise im weiteren Verlauf einer eingehenderen Betrachtung unterzogen.

Kapitel 5. Ein Beispielszenario

Im folgenden werden wir Schritt für Schritt die Umsetzung einer mehrsprachigen Web-Site mit Imperia skizzieren. Zum Verständnis ist ein gewisses Imperia-Know-How vonnöten, weswegen im weiteren Verlauf der Darstellung lediglich die spezifischen Teile der Implementierung einer näheren Betrachtung unterzogen werden. Bei den verwendeten Techniken wurde auf Wiederverwendbarkeit Wert gelegt, wenngleich der beschriebene Weg weder den einzig möglichen, noch den einzig richtigen darstellen will. Es werden jedoch alle relevanten Aspekte angesprochen, und die konkrete Umsetzung entspricht dem, was bei Imperia derzeit als Best Practice angesehen wird.

1. Die Anforderungen

Bei der Umsetzung werden wir versuchen, die folgenden Anforderungen zu erfüllen:

- Die Site soll zunächst in vier Sprachen, English, Französisch, Italienisch und Deutsch erstellt werden. Eine Erweiterung auf weitere Sprachen soll mit geringstmöglichem Aufwand machbar sein.
- Dokumente müssen nicht zwingend in allen Sprachen erstellt werden. Es soll möglich sein, die Sprachversionen sukzessive zu erstellen, also zum Beispiel erst eine englische Version fertigzustellen, und zu publizieren, und weitere Sprachvarianten erst in späteren Bearbeitungsschritten hinzuzufügen.
- Content und Meta-Informationen sollen in UTF-8 erfasst werden, um eine Erweiterbarkeit in beliebige Sprachen zu gewährleisten. Die Publikation der Dokumente soll hingegen im für die jeweilige Sprache gebräuchlichsten Charset erfolgen.
- Der Einsatz von Perl-Code sollte auf das Notwendigste beschränkt sein, um das erforderliche Know-How für die Pflege der Site auf niedrigem Niveau zu halten.
- Innerhalb der Templates soll eine möglichst weitgehende Trennung zwischen programmierter Anwendungslogik und Layout eingehalten werden. Dies gewährleistet einerseits die Wiederverwendbarkeit der Anwendungslogik für ähnliche Projekte und erleichtert Änderungen am Layout durch Fachkräfte mit geringen oder gar nicht vorhandenen Programmierkenntnissen.

2. Vorbereitung

Um die folgenden Beispiele nachzuvollziehen, wird eine funktionierende Imperia-Installation vorausgesetzt. Es sind keinerlei Manipulationen am System erforderlich, man kann also ohne größeres Risiko auch an einem Produktivsystem arbeiten, sofern die erforderlichen Systemeinstellungen (siehe Abschnitt 2.1, „System-Einstellungen“) nicht mit den vorhandenen Einstellungen in Konflikt stehen.

Alle Code-Beispiele sind bewusst einfach gehalten, und beschränken sich auf das Wesentliche, um die Source-Listings übersichtlich zu halten. In der Praxis werden sicherlich an allen Komponenten noch Verbesserungen vorzunehmen sein (auf konkrete Verbesserungsmöglichkeiten wird im jeweiligen Kontext hingewiesen), um die Stabilität und Usability zu erhöhen.

2.1. System-Einstellungen

Da der Content in Unicode eingepflegt werden soll, ist das System-Charset (und die jeweiligen User-Charsets) auf UTF-8 einzustellen. Die Systemsprache ist für unser Beispiel unerheblich. Wenn wir die Systemsprache über den Menüpunkt System -> Allgemeine Einstellungen auf UTF-8 konfigurieren, sollten wir gleichzeitig noch die Verwendung von Formularen und JavaScript innerhalb der Editieransicht des Template-Prozessors aktivieren:

Templateprozessor

☑ Formulare zulassen
☑ Skripten zulassen
☑ Frames, Iframes und Layers zulassen

Systemeinstellungen für den Template-Prozessor über System->Allgemeine Einstellungen

Dies ist keine allgemeine Voraussetzung für die Verwendung der Multi-Language-Features in Imperia. Im weiteren Verlauf werden wir jedoch teilweise DHTML und JavaScript einsetzen, um die Content-Eingabe komfortabler zu gestalten.

2.2. Template und Meta-Datei

Im ersten Schritt sollte ein Rumpf-Template und eine Rumpf-Metadatei erstellt werden. Diese Dateien legen wir unter den Namen `templatemultilang.htms` und `multilang.meta` an den entsprechenden Stellen im System ab. Das Rump-Template könnte in etwa so aussehen:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title><!--XX-title--></title>
    <meta http-equiv="Content-Type"
          content="text/html; charset=utf-8" />                  ❶
  </head>
```

```
  <body>

<!--formstart-->
<!--hidden-->                                                       ❷
<!--template-description:Beispiel-Template für Multi-Language-->

<h1>Fortsetzung folgt...</h1>

<!--controls-->
<!--essentials-->
<!--formend-->

</body>
</html>
```

❶ Wie bereits erwähnt, ist dieses `meta`-Tag im Template unbedingt notwendig, um sicherzustellen, dass der Browser die vom User später eingegebenen Formulardaten (sprich den Content) in UTF-8 an Imperia übermittelt.

❷ Hier lauert beim kleinen „ü“ schon die erste Falle: Die Template-Beschreibung wird in den Template-Auswahlboxen von Imperia angezeigt, und - da wir als Charset UTF-8 eingestellt haben - muss diese Beschreibung natürlich ebenfalls in UTF-8 eingegeben werden, da es sonst zu Fehldarstellungen bei der Template-Auswahl kommt. Falls kein Editor zur Hand ist, der UTF-8 abspeichern kann, muss man entweder auf 8-Bit-Zeichen verzichten, oder aber den rudimentären Template-Editor innerhalb Imperias verwenden (da dieser als HTML-Formular die Sonderzeichen korrekt übermittelt).

Auch bei der Meta-Datei `multilang.meta` beschränken wir uns zunächst auf das absolut Notwendige:

```
TITLE = "Multi-Language"
AUTHOR = "Imperia AG"

INPUT "40:title::Title"

#IF ("<!--XX-METAMODE-->")
HIDDEN "directory:<!--XX-directory-->/<!--count-->"
#ELSE
HIDDEN "directory:<!--XX-directory-->"
#ENDIF
```

Auch hier ist natürlich wieder darauf zu achten, dass das Meta-File in UTF-8 abgespeichert, da Imperia aufgrund der Charset-Einstellungen von dieser Kodierung ausgeht, und die entsprechenden HTTP-Header bei der Darstellung des entsprechenden Meta-Edit-Formulars übermittelt.

2.3. Eine Test-Rubrik

Die verwendete Meta-Datei (siehe Abschnitt 2.2, „Template und Meta-Datei“) „verlässt“ sich darauf, dass sowohl das Template als auch das Verzeichnis bereits bei der Definition der Rubrik sinnvoll vorbelegt sind. Als Zielverzeichnis geben wir deshalb einen beliebigen Testpfad auf dem Server an, als Template wählen wir unser soeben erstelltes Beispiel aus.

Für den mit der Rubrik assoziierten Workflow können wir zunächst jeden beliebigen Workflow, zum Beispiel den mit Imperia ausgelieferten Minimal-Workflow `default` wählen. Wir können aber auch gleich einen eigenen Minimal-Workflow `multilang` erstellen, da wir später ohnehin eine kleine eigene Plug-In-Erweiterung erstellen werden.

Das Grundgerüst für unsere Experimente sollte jetzt stehen. Es sollte möglich sein, ein Dokument in unserer neuen Rubrik zu erzeugen, und durch den Workflow zu schieben.

3. Basis-Implementierung

3.1. Mehrsprachige Eingabe

Im nächsten Schritt müssen wir dafür sorgen, dass unsere Redakteure den Content für jede Sprache einpflegen können. Wir beginnen zunächst mit der englischen Version, und erweitern unser Template um die folgenden Anweisungen:

```
<h1><input name="IMPERIA:title_en"
           value="title_en" /></h1>                    ❶

<p>
<textarea name="IMPERIA:teaser_en" cols="60" rows="5">  ❶
teaser_en
</textarea>
</p>
```

❶ Bei der Benennung der Meta-Variablen befolgen wir die Namenskonvention, dass wir Meta-Felder, die lediglich für eine Sprache Bedeutung haben, durch den angehängten 2-Buchstaben-Code kennzeichnen.

Den Part für die englische Sprachredaktion müssen wir nun für die drei weiteren Spra-

chen kopieren, wobei wir natürlich das Kürzel `en` durch die entsprechenden Kürzel `fr`, `it` und `de` sinngemäß ersetzen müssen.

3.1.1. Verallgemeinerung der Spracheingaben

Die Eingabemasken für die jeweiligen Spracheingaben per Cut & Paste zu duplizieren, ist natürlich keine optimale Lösung. Das Verfahren ist fehleranfällig, weil es bei der Anpassung der einzelnen Teile zu Vertippern kommen kann. Weiterhin entspricht das Kopieren nicht unserer Forderung nach leichter Erweiterbarkeit: Im Moment ist unser Template noch sehr übersichtlich. Sobald es aber von der Designabteilung mit HTML vollgestopft wurde, wird das Hinzufügen einer neuen Sprache doch zu einer ziemlichen Herausforderung.

Unser Ziel sollte es also sein, den Teil, der sich wiederholt, „auszuklammern“, und das Mittel der Wahl heißt *Code-Includes*.

3.1.2. Parametrisierte Code-Includes

Wir erzeugen also aus dem englischen Teil der Spracheingabe im Template `templatemultilang.htms` ein Code-Include `multilang.htms`, und fügen im eigentlichen Template stattdessen die entsprechende Include-Angabe ein.

```
<!--CODEINCLUDE:multilang.htms-->
```

Natürlich müssen wir unserem Code-Include noch in irgendeiner Form mitteilen, für welche Sprache es jeweils gerade eingesetzt wird. Am einfachsten lässt sich dies durch einen Parameter im Aufruf realisieren. Wir ändern also nochmals unser Template:

```
<!--CODEINCLUDE:multilang.htms:PARAMETERS=en-->
<!--CODEINCLUDE:multilang.htms:PARAMETERS=fr-->
<!--CODEINCLUDE:multilang.htms:PARAMETERS=it-->
<!--CODEINCLUDE:multilang.htms:PARAMETERS=de-->
```

Da wir vier Sprachen gleichzeitig pflegen, müssen wir die Include-Datei auch viermal einbinden, jeweils für die entsprechende Sprache parametrisiert. Im Code-Include selber muss der Parameter wiederum abgefragt werden, was uns zur nächsten Version der Include-Dateien führt:

```
<h1><input name="IMPERIA:title_<!--CI_PARAM1-->"
           value="title_<!--CI_PARAM1-->" /></h1>

<p>
```

```
<textarea name="IMPERIA:teaser_<!--CI_PARAM1-->" cols="60" rows="5">
teaser_<!--CI_PARAM1-->
</textarea>
</p>
```

Wir müssen also lediglich unser bisher hartkodiertes Sprachkürzel `en` durch den Parameter, mit dem das Code-Include aufgerufen wurde, ersetzen. Unsere heile Welt der Wiederverwendbarkeit wird lediglich etwas gestört, wenn wir innerhalb dieses Code-Includes wieder sprachabhängige Teile einführen. Wann sich hier eine rekursive - wieder parametrisierte - Einbindung lohnt, ist Ermessenssache, bzw. hängt von den Gegebenheiten ab:

```
<!--CODEINCLUDE:multilang_recursive.htms:PARAMETERS=<!--CI_PARAM1-->-->
```

Alternativ kann natürlich auch eine Fallunterscheidung vorgenommen werden:

```
#IF ("<!--CI_PARAM1-->" EQ "it")
<h3>Non è mica difficile, nevvero?</h3>
#ELSIF ("<!--CI_PARAM1-->" EQ "fr")
<h3>Ce n'est pas difficile après tout, n'est-ce pas?</h3>
#ELSIF ("<!--CI_PARAM1-->" EQ "en")
<h3>It isn't hard to do, is it?</h3>
#ELSIF ("<!--CI_PARAM1-->" EQ "de")
<h3>Das ist doch gar nicht schwer, oder?</h3>
#ELSE
<h3>Parameter im Aufruf des Code-Includes fehlt!!!</h3>
#ENDIF
```

Wir sind jetzt in der Lage, Inhalte in allen vier geforderten Sprachen einzupflegen. Das weitere Vorgehen fuer diesen einfachen Fall ist im Imperia-Programmierhandbuch beschrieben. Im nächsten Schritt, würden zunächst sprachspezifische Templates erstellt, die sich vom Haupttemplate eigentlich nur insoweit unterscheiden, als dass sie statt aller vier Code-Includes nur das Code-Include für die jeweilige Sprache einbinden. Auch an verschiedenen weiteren Stellen wären noch kleinere Anpassungen vonnöten, so würden wir als Dokumententitel natürlich den Titel für die jeweilige Sprache einfügen.

Weiterhin muss natürlich für die Erzeugung der Copy-Pages gesorgt werden. Das Meta-File wäre demnach entsprechend zu erweitern:

```
HIDDEN "no_publish:1"
HIDDEN "copy_en:<!--KK-directory-->/index.html.en:TEMPLATE=\
templatemultilang_en.htms>"
HIDDEN "copy_fr:<!--KK-directory-->/index.html.fr:TEMPLATE=\
templatemultilang_fr.htms>"
HIDDEN "copy_it:<!--KK-directory-->/index.html.it:TEMPLATE=\
templatemultilang_it.htms>"
HIDDEN "copy_de:<!--KK-directory-->/index.html.de:TEMPLATE=\
templatemultilang_de.htms>"
```

Die Anweisungen müssen jeweils in einer Zeile stehen, also an den Backslashes jeweils zusammengesetzt werden. Die Backslashdarstellung ist lediglich für den Druck erforderlich.

Das Verfahren sollte eigentlich geläufig sein. Die Meta-Variable `no_publish` sorgt dafür, dass Imperia nicht versucht, eine Seite auf Basis des Master-Templates zu erstellen und zu publizieren, da dieses Template ja lediglich zur Eingabe des Contents verwendet wird.

Da es sich hierbei um konventionelle Imperia-Funktionalität handelt, werden wir hierauf auch nicht weiter eingehen, und uns stattdessen damit beschäftigen, wie wir unsere Struktur von vorneherein wesentlich flexibler auslegen können.

4. Dynamische Sprachwahl

Eine unserer Anforderungen haben wir derzeit noch nicht erfüllt: Jedes Dokument wird in allen vier Versionen erstellt. Unser Ziel ist es jedoch, der Redaktion ein Mittel an die Hand zu geben, nur einzelne Sprachen auszuwählen, und weitere Sprachen später bei Bedarf nachzureichen. In klassischer Imperia-Manier fügen wir eine solche Sprachauswahl in unser Meta-File ein. Wir erweitern es, wie folgt:

4.1. Anpassung der Meta-Datei

```
HTML "<h4>Languages</h4>"

#IF (NOT ("<!--XX-lingua_en-->"))
CHECKBOX "linguas:en::English"
#ELSE
HIDDEN "linguas:en"
#ENDIF
```

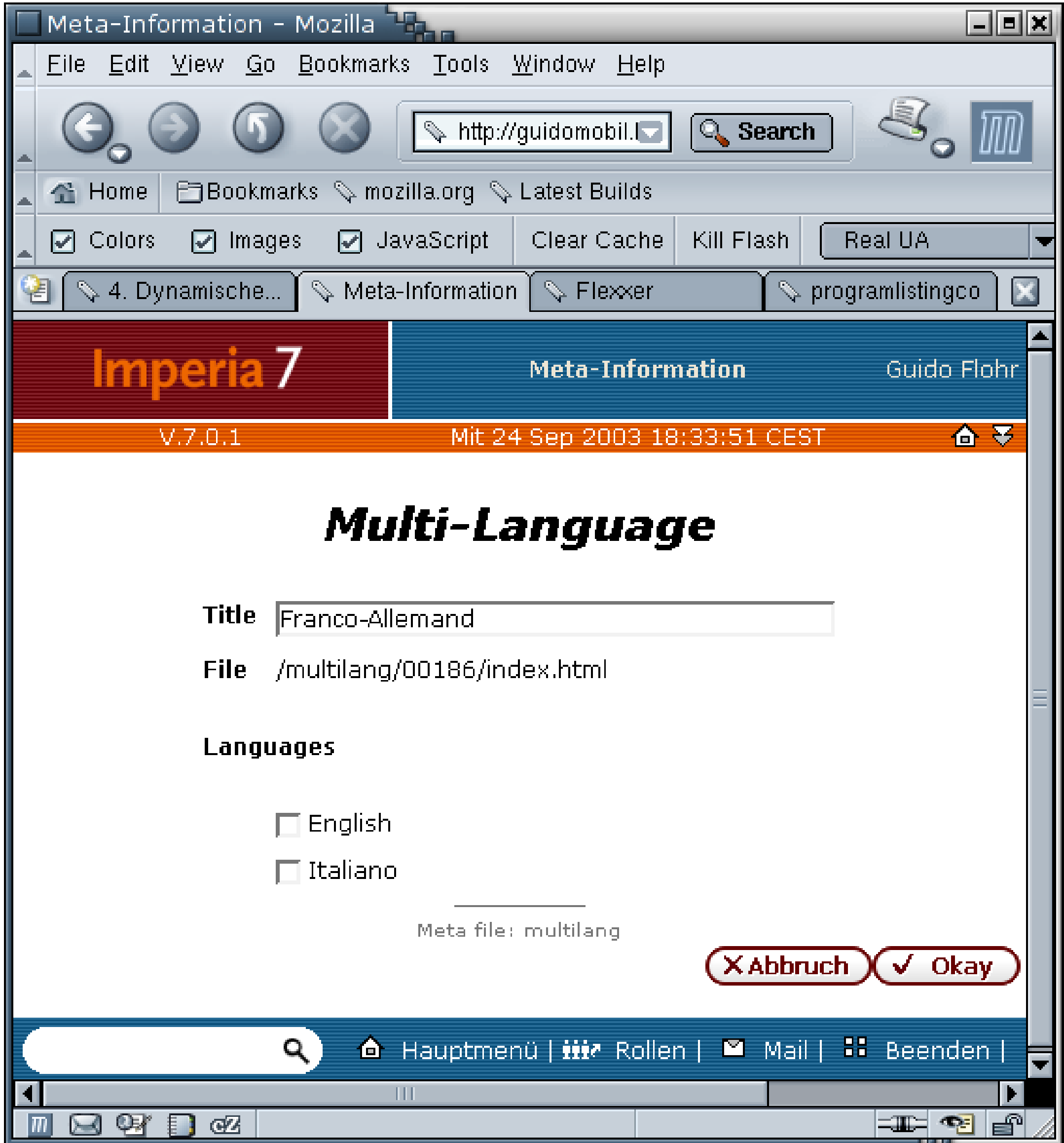
```
#IF (NOT ("<!--XX-lingua_fr-->"))
CHECKBOX "linguas:fr::FranÃ§ais"
#ELSE
HIDDEN "linguas:fr"
#ENDIF

#IF (NOT ("<!--XX-lingua_de-->"))
CHECKBOX "linguas:de::Deutsch"
#ELSE
HIDDEN "linguas:de"
#ENDIF

CHECKBOX "linguas:it::Italiano"
#ELSE
HIDDEN "linguas:it"
#ENDIF
```

Bemerkenswert ist an diesem Fragment unserer Meta-Datei allenfalls, dass wir stets dieselbe Meta-Variable `linguas` verwenden. Führt dies nicht dazu, dass der Inhalt der Variablen sukzessive überschrieben wird? Nein! Tatsächlich erzeugen wir durch diesen Trick eine Liste, `linguas` enthält also nach Abarbeitung des Meta-Edit-Schrittes eine Liste der bereits vorhandenen und neu hinzugewählten Sprachen.

Aus Redakteurrsicht präsentiert sich diese Meta-Seite zum Beispiel so:



Sprachauswahl über Meta-Datei

In diesem Falle handelt es sich offensichtlich um ein bereits bearbeitetes Dokument, für das schon eine französische und deutsche Version erstellt wurde. Die Auswahl für den Redakteur beschränkt sich also nunmehr darauf, noch eine englische und/oder italienische Version zu erstellen.

4.2. Ein einfaches Workflow-Plug-In

Für jede vom Redakteur gewählte Sprache wird nun eine Meta-Variable (`lingua_en`, `lingua_fr`, ...) belegt, die im Template abgefragt werden kann, und zur dynamischen Einbindung der Code-Includes verwendet werden kann. Allerdings hätte diese einfache Variante mit einigen Problemen zu kämpfen. Es sollte zum Beispiel verhindert werden, dass überhaupt keine Sprache ausgewählt wurde. Weiterhin muss natürlich für jede Sprach-Version eine Copy-Seite erzeugt werden, mithin ein entsprechendes Metafeld befüllt werden. Dies ließe sich prinzipiell alles auch ohne Perl-Code realisieren. Im Endeffekt ist eine Perl-Erweiterung für unsere Zwecke aber die flexiblere und gleichzeitig weni-

ger komplexe Lösung.

Da unser Thema nicht die Programmierung von Workflow-Plug-Ins für Imperia ist, wird das Vorgehen hier nur soweit skizziert, wie es nötig ist, um die weiteren Schritte nachvollziehen zu können.

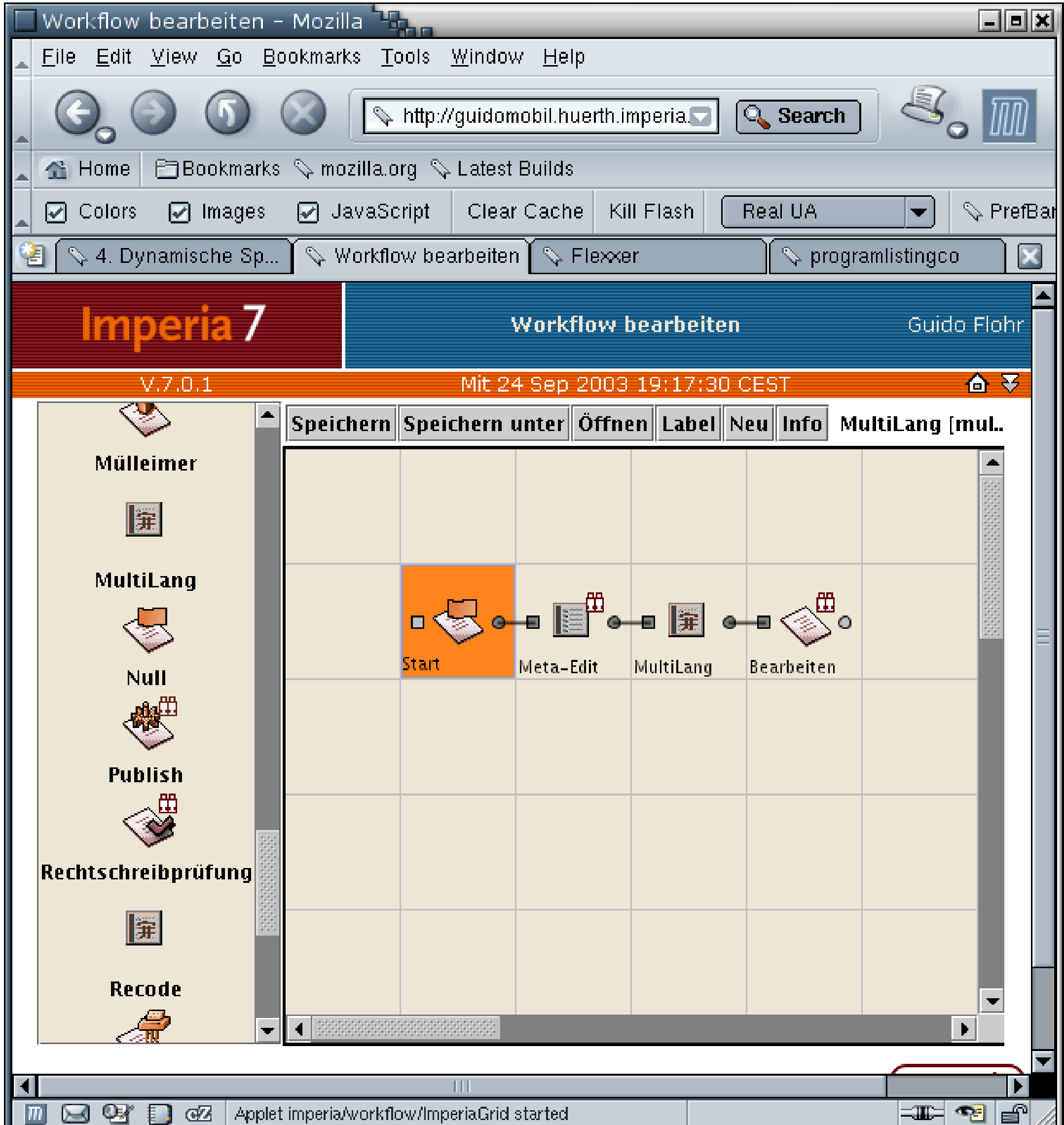
4.2.1. Gerüst für das Plug-In

Der Einfachheit halber werden wir für die Erstellung unseres neuen Multi-Language-Plug-Ins hemmungslos bei den Plug-Ins, die bereits mit Imperia ausgeliefert werden, „klauen“. Zunächst brauchen wir ein Icon, mit dem unser Plug-In im Workflow visualisiert wird. Bei neueren Versionen sollten im Verzeichnis `site/workflow/stepdefs` und `htdocs/imperia/images/worfklow` jeweils ein Icon namens `Recode.gif` liegen, das für unsere Zwecke geeignet ist; jedes andere Icon aus den Verzeichnissen erfüllt aber den gleichen Zweck. Wir erstellen jeweils im gleichen Verzeichnis eine Kopie, und speichern sie unter dem neuen Namen `MultiLang.gif` ab.

Ebenfalls im Verzeichnis `site/workflow/stepdefs` findet sich die Definitionsdatei `Null.xml`, von der wir jetzt eine Kopie `MultiLang.xml` erzeugen. Die XML-Datei laden wir in einen Editor, ersetzen jedes Vorkommen von „Null“ durch „MultiLang“, und machen uns keine weiteren Gedanken um das Format der Datei, da wir ja für heute andere Pläne haben.

4.2.2. Workflow-Definition

Die (Perl-)Implementierung unseres neuen Plug-Ins fehlt noch. Für das Workflow-Grid sind jedoch bereits alle notwendigen Daten vorhanden. Falls noch nicht geschehen, erstellen wir einen neuen Worfklow mit dem Namen „multilang“, und definieren ihn so:



Worfklow-Definition im Imperia-Grid

Der Workflow beginnt mit einem Null-Plug-In, das wir mit „Start“ beschriften.[3] Darauf folgt ein normaler Meta-Edit-Schritt, an dessen Ost-Anschluss wir unser neues Multi-Lang-Plug-In andocken. Der letzte Schritt ist der Bearbeitungsschritt, weitere Mätzchen wie Genehmigungsschritte schenken wir uns, weil wir ja während der Testphase möglichst schnell unsere Dokumente erzeugen wollen.

Wurde der Workflow erfolgreich abgespeichert (eine Konfiguration der einzelnen Plug-Ins ist für unsere Zwecke nicht erforderlich), rufen wir die Rubrikendefinition auf, und ändern sie dergestalt, dass unser neuer Workflow verwendet wird. Wir können jetzt bereits ein neues Dokument in unserer Rubrik erzeugen[4], und werden mit einem Fehler der Workflow-Engine konfrontiert, weil die Implementierung des Plug-Ins noch fehlt.

4.2.3. Implementierung der Plug-In-Logik

[4]*Achtung!* Dokumente, die sich derzeit noch im Workflow befinden, werden von der Änderung der Rubrikendefinition nicht erfasst. Die neue Workflowauswahl wirkt sich nur auf neue oder re-importierte Dokumente aus

Der Perl-Code für unser Plug-In fehlt also noch. Er gehört in die Datei `site/modules/core/Dynamic/Workflow/MultiLang.pm`:

```
#! /bin/false

package Dynamic::Workflow::MultiLang;

use strict;

use base qw (Dynamic::Workflow::Null);                                    ❶

sub edit
{
        my ($self, $loader, $cgiform) = @_;
❷

        my $metainfo = $loader->getMetaInfo
                or die $loader->errorSuck;

        my @linguas = $metainfo->getValues ('linguas');
❸
        unless (@linguas) {
❹
                @linguas = ('en');
                $metainfo->setValues (linguas => @linguas);
        }

        $metainfo->deleteValues ('copy');
❺
        foreach my $lingua (@linguas) {
❻
                $metainfo->setValues ("lingua_$lingua" => 1);
❼
                my $template = $metainfo->getValues ('template');
                $template .= '_' . $lingua;
                my $uri = $metainfo->getValues ('directory') . '/';
                $uri .= $metainfo->getValues ('filename') . ".$lingua";
                my $copy = $uri . ":TEMPLATE=$template";
                $metainfo->pushValues (copy => $copy);
        }
                                                                          ❽
        $metainfo->setValues (no_publish => 1);

        $self->_flushMeta ($loader, $metainfo);
```

```
        return 'READY';
}

1;
```

❶ Wir leiten unser Plug-In vom Null-Plug-In ab, weil letzteres sich dadurch auszeichnet, dass es nichts tut. Es kann also auch nichts Falsches machen.

❷ Diese und die folgenden Zeilen sind Standard. Der Parameter `$loader` ist ein Objekt vom Typ `Document::Loader`, über das wir die Meta-Informationen unseres Dokumentes als `Meta::Info`-Objekt ermitteln können.

❸ Wir rufen die Meta-Information `linguas` im Listenkontext ab, und erhalten somit auch in Perl eine Liste (ein Array) `@linguas`. Die Meta-Variable `linguas` wurde im vorhergehenden Meta-Edit-Schritt auf die gewählten Sprachen initialisiert (siehe Abschnitt 4.1, „Anpassung der Meta-Datei") und enthält entweder die Liste der gewählten Sprachen oder gar nichts, für den Fall, dass der Redakteur keine Sprache gewählt hat.

❹ Falls die Liste leer war (der Redakteur also keine Sprache gewählt hat), initialisieren wir sie auf Englisch. Genausogut könnten wir hier eine Fehlermeldung erzeugen, oder eine Rückverweisung zum Meta-Edit-Schritt anstoßen.

❺ Die Metavariable `copy` füllen wir später dynamisch, und initialisieren sie hier lediglich auf eine leere Liste, um die Informationen aus früheren Durchläufen zu überschreiben.

❻ Als nächstes durchlaufen wir die Liste aller gewählten Sprachen, und erzeugen zunächst die Hilfsvariablen `lingua_en`, `lingua_fr`, etc., die in der Meta-Datei (siehe Abschnitt 4.1, „Anpassung der Meta-Datei") ausgewertet werden, um festzustellen, ob die jeweilige Sprache bereits angewählt wurde, oder nicht.

❼ Für den Rest der Schleife sind wir damit beschäftigt, die Meta-Variable `copy` (ebenfalls eine Liste!) für jede gewählte Sprache der Syntax für Copy-Seiten entsprechend zu befüllen. Die dafür notwendigen Informationen gewinnen wir aus den Meta-Variablen `directory`, `filename` und `template`.

❽ Nach dem vollständigen Durchlauf der Schleife setzen wir sicherheitshalber noch die Meta-Variable `no_publish` um eine versehentliche Publikation des Dokuments auf Basis des Editiertemplates zu verhindern, sorgen über die Plug-In-Methode `_flushMeta`[5] dafür, dass die modifizierten Meta-Informationen in die Imperia-Datenhaltung zurückgeschrieben werden, und signalisieren der Workflow-Engine schließlich durch den Rückgabewert `READY`, dass wir die Abarbeitung dieses Workflowschrittes für beendet erachten.

Spielen wir kurz durch, was zum Beispiel für den Fall passiert, dass ein Redakteur die Sprachen Französisch und Deutsch ausgewählt hat. Die Schleife sorgt dafür, dass die Hilfs-Variablen `lingua_fr` und `lingua_de` gesetzt sind, und macht insgesamt zwei Einträge in die Liste `copy`. Falls wir annehmen, dass unser (Master-/Editier-)Template den Namen `multilang` hat, sähe die Liste `copy` so aus:

[5] Diese Methode wird von der Basisklasse geerbt.

```
@copy = (
     '/multilang/00123/index.html.fr:TEMPLATE=multilang_fr',
     '/multilang/00123/index.html.de:TEMPLATE=multilang_de',
)
```

Durch unser Plug-In gelangen wir mit wenigen Zeilen Perl-Code zu einer flexiblen, wiederverwendbaren und leicht zu erweiternden Implementierung. Die Lösung ist zum Beispiel völlig unabhängig vom konkreten Template, oder auch dem URL der späteren Dokumente. Es wird lediglich eine Namenskonvention vorausgesetzt.

Statt eine Liste `copy` zu befüllen, hätten wir natürlich auch einzelne Variablen, zum Beispiel `copy_en`, `copy_fr`, etc. entsprechend belegen können (das gleiche gilt für die Liste `linguas`). Die Verwendung von Listen besitzt gegenüber Skalaren jedoch den Vorteil, dass der Inhalt der Listen mit einem einzigen Statement abgerufen werden kann, während bei der Lösung mit Skalaren stets eine Schleife mit einem Namensvergleich notwendig ist.

Speziell für die Copy-Variablen, hat die Listenlösung noch einen weiteren Vorteil, der später in der Editieransicht zutage tritt. Liegen die Informationen für die Copy-Seiten in einer Liste statt in Skalaren, bietet Imperia bei der Aktivierung der Voransicht gleich eine Auswahl für das jeweilige Template:

Copy-Template-Auswahl für die Voransicht

Der Redakteur kann somit wählen, ob er eine Voransicht für das vollständige Dokument oder aber (das wird der Normalfall sein) nur für eine einzelne Sprachversion sehen möchte.

4.3. Dynamische Einbindung der Code-Includes

Auf unserem jetzigen Stand werden jedoch noch immer vier Code-Includes mit hartkodierten Parametern vom Template eingebunden. Auch diesen Zustand müssen wir noch ändern, denn es sollen ja lediglich für die tatsächlich gewählten Sprachen, die Editieransicht ins Template gezogen werden. Dies erreichen wir dadurch, dass wir das Master-Template so ändern, dass es statt der vier Code-Include-Aufrufe, nur noch ein einziges Perl-Include verwendet, das die weitere Steuerung übernimmt. Im Template sehen wir also nur noch:

```
<--CODEINCLUDE:multilang_setup.pl-->
```

Die Steuerlogik legen wir in der Datei `site/include/multilang_setup.pl` ab:

```
my @linguas = $metainfo->getValues ('linguas');                    ❶

foreach my $lingua (@linguas) {
        $new .= <<EOF;
❷
<!--CODEINCLUDE:multilang.htms:PARAMETERS=$lingua-->
EOF
}
```

❶ Wir ermitteln die Liste der gewählten Sprachen.

❷ Für jede Sprache binden wir rekursiv unser ursprüngliches HTML-Code-Include ein, wobei wir den Parameter dynamisch setzen.

Wir befinden uns jetzt auf einem funktionsfähigen Stand. Die Redakteure können sowohl bei der Erstellung, als auch bei der Wiederbearbeitung eines Dokumentes die Menge der zu bearbeitenden Sprachen frei wählen. Die Einbindung der jeweiligen Editieransichten und die Erzeugung der Sprachversionen erfolgt im weiteren Verlauf automatisch.

5. Reduzierung der Templates

Unsere Implementierung leidet derzeit noch unter einem Schönheitsfehler. Das Mastertemplate und die Ausgabetemplates für die einzelnen Sprachen unterscheiden sich nur in ganz geringem Maße. Es sollte doch möglich sein, für die Editierung und die Ausgabe ein und dasselbe Template zu verwenden. Auch schon bei der Voransicht sind die Ausgabetemplates größtenteils redundant.

Die Herausforderung besteht jetzt darin, innerhalb der Templates dynamisch zu ermitteln, ob alle Sprachversionen eingebunden werden müssen (in der Editieransicht), oder aber nur die relevanten Teile für eine Sprache. Diese Unterscheidung ist relativ einfach möglich, da wir lediglich abfragen müssen, ob wir uns im Editier-Modus befinden oder nicht.

Auch bei der Publikation der Seiten sind die notwendigen Fallunterscheidungen trivial. Neuere Imperia-Versionen setzen stets die Meta-Variablen `__imperia_parent_directory` und `__imperia_parent_filename`, die Verzeichnis und Dateiname der Master-Datei enthalten (selbst, wenn diese später bei der Publikation unterdrückt wird). Durch einen Vergleich dieser Informationen mit den Meta-Variablen `directory` und `filename`, die mit den entsprechenden Informationen für die konkrete Kopie gefüllt sind, ist leicht festzustellen, ob es sich bei einer Datei um das Master-Dokument oder eine Seitenkopie handelt.

Für die Voransicht ist diese Unterscheidung dagegen nicht so trivial zu machen, denn

hier werden die Meta-Informationen stets unverändert belassen, ganz gleich, welches Template zur Anzeige verwendet wird (das heißt, das selbst die Meta-Variable `template` den Namen des Master-Templates und nicht etwa den des für die Anzeige gerade verwendeten Templates enthält). Es gibt also keinerlei Anhaltspunkt dafür, ob es sich bei der Erzeugung der aktuellen Preview-Ansicht um eine Sprach-Version handelt oder nicht. Und selbst, wenn diese Unterscheidung getroffen werden könnte, ist es noch immer nicht möglich, festzustellen, *welche* Sprache zur Anzeige kommt.

Daraus folgt, dass wir unser Ziel derzeit [6] nicht vollständig erreichen können. Das Master-Template und die Ausgabetemplates für die einzelnen Sprachen müssen sich zumindest im Namen unterscheiden. Dies ist aber auch aus einem weiteren Grunde wünschenswert, da wir ja in der Previewansicht auch eine sinnvolle Template-Auswahl durch den Redakteur ermöglichen wollen. Für den Redakteur ist der Template-Name aber ebenfalls das einzig sichtbare Unterscheidungsmerkmal.

Auf Systemen, die symbolische oder harte Links unterstützen, behelfen wir uns daher damit, dass wir das Template unter mehreren Namen gleichzeitig ansprechbar machen - also Links vom Mastertemplate zu den Ausgabetemplates legen - und somit mit einer einzigen physikalischen Datei arbeiten. Auf Systemen ohne Links (insbesondere also Windows) bleibt uns nichts anderes übrig, als nach einer Änderung des Master-Templates die jeweiligen Kopien zu erzeugen.

Um das neue Feature zu implementieren, müssen wir lediglich die Steuerlogik im Perl-Include `multilang_setup.pl` modifizieren.

```
my @linguas = $metainfo->getValues ('linguas');

if ($mode ne 'EDIT') {                                  ❶
        my $id = $params->{template}->id;               ❷
        if ($id =~ /_([a-z][a-z])$/) {                  ❸
                @linguas = ($1);
        } else {
                $#linguas = 0;                          ❹
        }

}
foreach my $lingua (@linguas) {
          # etc.
```

❶ Die Variable `$mode` wird vom Template-Prozessor bereits initialisiert.

❷ Schwarze Imperia-Magie. Die Template-ID ist nichts anderes als der Name des tatsächlich verwendeten Templates. Die Meta-Variable `template` würde uns hier nicht weiterhelfen, da sie den Namen des Master-Templates enthält.

[6] Dieses Manko wird in zukünftigen Imperia-Versionen behoben.

❸ Weiße Perl-Magie. Wir kennen unsere eigene Namenskonvention, und erkennen am Sprachkürzel des Templates die aktuelle Sprache. Da wir uns nicht im Edit-Modus befinden, wollen wir auf jeden Fall nur eine einzige Sprache anzeigen, und reduzieren die Liste `@linguas` auf den gerade ermittelten Wert.

❹ Auch der Fall, dass das Master-Template (das keinen Sprachkürzel im Namen enthält) für die Voransicht ausgewählt wurde, will behandelt werden. Hier entscheiden wir uns dafür, die Liste auf den ersten Wert zurechtzustutzen, wodurch die erste Sprache in der Liste automatisch ausgewählt wird. Alternativ könnten wir die Variable `@linguas` auch unangetastet lassen, und dadurch erreichen, dass alle Sprachen gleichzeit angezeigt werden.

Damit ist der nächste Milestone erreicht. Faktisch brauchen wir nur noch ein einziges Template zu pflegen. Die Einbindung der sprachspezifischen Teile erfolgt dann volldynamisch.

6. Charset-Konvertierung

Unserer Anforderungsliste entnehmen wir, dass wir ursprünglich vorhatten, die Seiten nicht in UTF-8 zu publizieren, sondern stattdessen das für die jeweilige Sprache gebräuchlichste Charset zu wählen. Diese Anforderung ist fast trivial in der Umsetzung.

Da wir unser Master-Template weiterhin von sprachspezifischen Details freihalten wollen, ergibt sich fast zwangsläufig wieder ein Code-Include als Lösungsansatz. Prinzipiell könnten wir die Logik natürlich in eines der bereits vorhandenen Code-Includes integrieren. Allerdings müssen wir jetzt das `meta`-Tag im HTML-`head` dynamisch setzen, und dort das richtige Charset eintragen, während die anderen Code-Includes jeweils innerhalb des HTML-`body` eingebunden werden. Wir kommen also nicht umhin, für diese Anforderung ein separates Code-Include zu schreiben. Unser Master-Template müssen wir aber zuerst vorbereiten:

```
#IF ("<!--XX-editmode-->")
    <title><!--XX-title--></title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" /
#ELSE
    <!--CODEINCLUDE:multilang_charset.pl-->
#ENDIF
```

Das Code-Include sieht folgendermaßen aus:

```
use constant CHARSETS => {                                          ❶
        en => 'ISO-8859-1',
        fr => 'Windows-1252',
        de => 'ISO-8859-15',
```

```
        it => 'UTF-8',
};

my $id = $params->{template}->id;                                    ❷
$id =~ /_([a-z][a-z])$/;
my $lingua = $1;
$lingua = 'en' unless defined $lingua;

my $charset = CHARSETS->{$lingua};                                   ❸
$new .= <<EOF;                                                       ❹
<meta http-equiv="Content-Type" content="text/html; charset❺=$charset"
<title><!--XX-title_$lingua--></title>
<!--postconvert:Recode(UTF-8|$charset)-->
EOF
```

❶ Welches Charset, für welche Sprache gebräuchlich ist, kann Imperia noch nicht raten. Wir verwenden daher eine hartkodierte Liste. Die vorliegende Liste ist natürlich relativ sinnfrei; wahrscheinlich würde man für unsere Beispielsprachen durchgehend das gleiche Charset wählen.

❷ Die Ermittlung der aktuellen Sprache gehört jetzt ja schon zu unseren leichtesten Übungen. Eine Abfrage des Modus können wir uns im vorliegenden Falle schenken, weil dies bereits im Template erfolgt.

❸ Für das zu verwendende Charset machen wir einen Lookup in unserer Tabelle. Für Real-World-Anwendungen würden wir hier natürlich einen Fallback-Wert einsetzen, falls kein Charset gefunden werden kann.

❹ Das Ergebnis schreiben wir in Template-Syntax heraus. Neben dem Charset nutzen wir die günstige Gelegenheit, und setzen auch noch gleich den Titel auf den sprachspezifischen Wert.

❺ Die Postconvert-Direktive realisiert die Konvertierung.

Die eigentliche Konvertierung von UTF-8 ins Ziel-Charset wird über die Postconvert-Direktive erreicht. Diese Anweisung kann an beliebiger Stelle im Template stehen, die Syntax sieht folgendermaßen aus:

```
<--postconvert:Recode(FROM|TO)-->
```

Für *FROM* und *TO* sind das Quell- und Zielcharset einzutragen, den Rest erledigt Imperia. Sollten Quell- und Zielcharset identisch sein (bei genauerem Hinschauen wird man feststellen, dass dies bei uns für Italienisch der Fall ist), ist dies harmlos. Welche Charsets unterstützt werden, ist systemabhängig, eine vollständige Liste erhält man z. B. bei der Charset-Auswahl in den persönlichen Einstellungen.

Wichtig! Die Konvertierung sollte *nie* (außer, man weiß genau, was man da tut) im Edit-Modus erfolgen, da sich die Konvertierung auf das vollständig expandierte Template bezieht. Dies kann zu schweren Fehlern führen!

Wir testen die neue Funktionalität, und stellen fest, dass die Ausgabe-Seiten sowohl im Preview-Modus als auch bei der Publizierung unseren Anforderungen entsprechen. Das `meta`-Tag ist korrekt gesetzt, und die Seiten werden fehlerfrei dargestellt, was ein Zeichen dafür ist, dass die Konvertierung erfolgreich war.

7. Verfeinerungen

Wir haben unser Lastenheft somit erfolgreich abgearbeitet, und können uns nach der Absolvierung der Pflicht nun der Kür zuwenden.

7.1. Problemfall Flexmodule

Unsere Redakteure schreien nun nach dem Komfort der Imperia-Flexmodule, an die sie sich in der Zeit vor der Mehrsprachigkeit unserer Site gewöhnt hatten. Vor der Verwendung von Flex-Modulen mit Copy-Seiten gilt es jedoch einige Hürden zu überspringen. Enthält ein Flex-Modul zum Beispiel eine Variable `longtext` zur Aufnahme von Fließtext, muss Imperia in der Lage sein, den Variableninhalt auch dann noch zu unterscheiden, wenn dieses Flex-Modul mehrfach eingefügt wurde. Dies wird dadurch erreicht, dass diese Variablen intern um eine Nummerierung erweitert werden, die wieder eine eindeutige Unterscheidbarkeit zwischen den einzelnen Instanzen erlaubt.

Diese interne Nummerierung gerät natürlich durcheinander, wenn Flexmodulaufrufe plötzlich verschwinden oder hinzukommen, und genau dies ist in Multi-Language-Szenarien regelmäßig der Fall. Erstellen wir eines unserer Dokumente beispielsweise in Englisch und Französisch, werden die englischen Flex-Inhalte in der Editier-Ansicht den Index 0 haben, während die französischen Flex-Inhalte mit 1 indiziert werden.

Bei der Publizierung und in der Voransicht werden dagegen stets alle Sprachen bis auf eine ausgeblendet, somit auch die Flex-Aufrufe im Template. Für jede Sprache würde daher auf die Flex-Inhalte mit dem Index 0 zurückgegriffen, was aber leider nur für die erste Sprache in der Liste zum gewünschten Ergebnis führt. In diesen Fällen dürfen wir uns daher nicht mehr auf die automatische Zählung verlassen, sondern müssen dem Imperia-Template-Prozessor durch die Angabe eines expliziten Index auf die Sprünge helfen. Die Index-Vergabe sollte sich während der gesamten Lebendauer der Site natürlich niemals ändern, weshalb wir uns dazu entschließen, zu diesem Zweck eine zentrale Liste zu führen, die für neue Sprachen dann nach oben erweitert werden kann.

Wir geben uns ans Werk, und fügen in unser Code-Include `multilang.htms` am Ende einen Flexmodul-Aufruf ein:

```
<!--INSERT_FLEXMODULE:INDEX=<!--CI_PARAM2-->-->
```

Wir sehen, dass wir unserem Code-Include einen weiteren Parameter (**<!--CI_PARAM2-->**) für die Flex-Zählung übergeben müssen. Dementsprechend müssen wir natürlich auch den Aufruf im anderen Include-File `multilang_setup.pl` abändern:

```
                                                              ❶
use constant LANGNUMBERS => {
        en => 0,
        fr => 1,
        de => 2,
        it => 3,
};

my @linguas = $metainfo->getValues ('linguas');

# ...

foreach my $lingua (@linguas) {
        my $lang_number = LANGNUMBERS->{$lingua};             ❷
        $new .= <<EOF;
<!--CODEINCLUDE:multilang.htms:PARAMETERS=$lingua/$lang_number-->
EOF
}
```

❶ Auch hier benötigen wir eine hartkodierte Liste der Sprachkürzel, um eine Abbildung von Sprachkürzeln auf die jeweils verwendeten Flex-Indizes zu erhalten.
❷ Den jeweils ermittelten Index übergeben wir nun als zusätzlichen Parameter an das Code-Include (Parameter werden durch Slashes getrennt).

Unser Konzept trägt sich also auch noch bei Verwendung von Flex-Modulen.

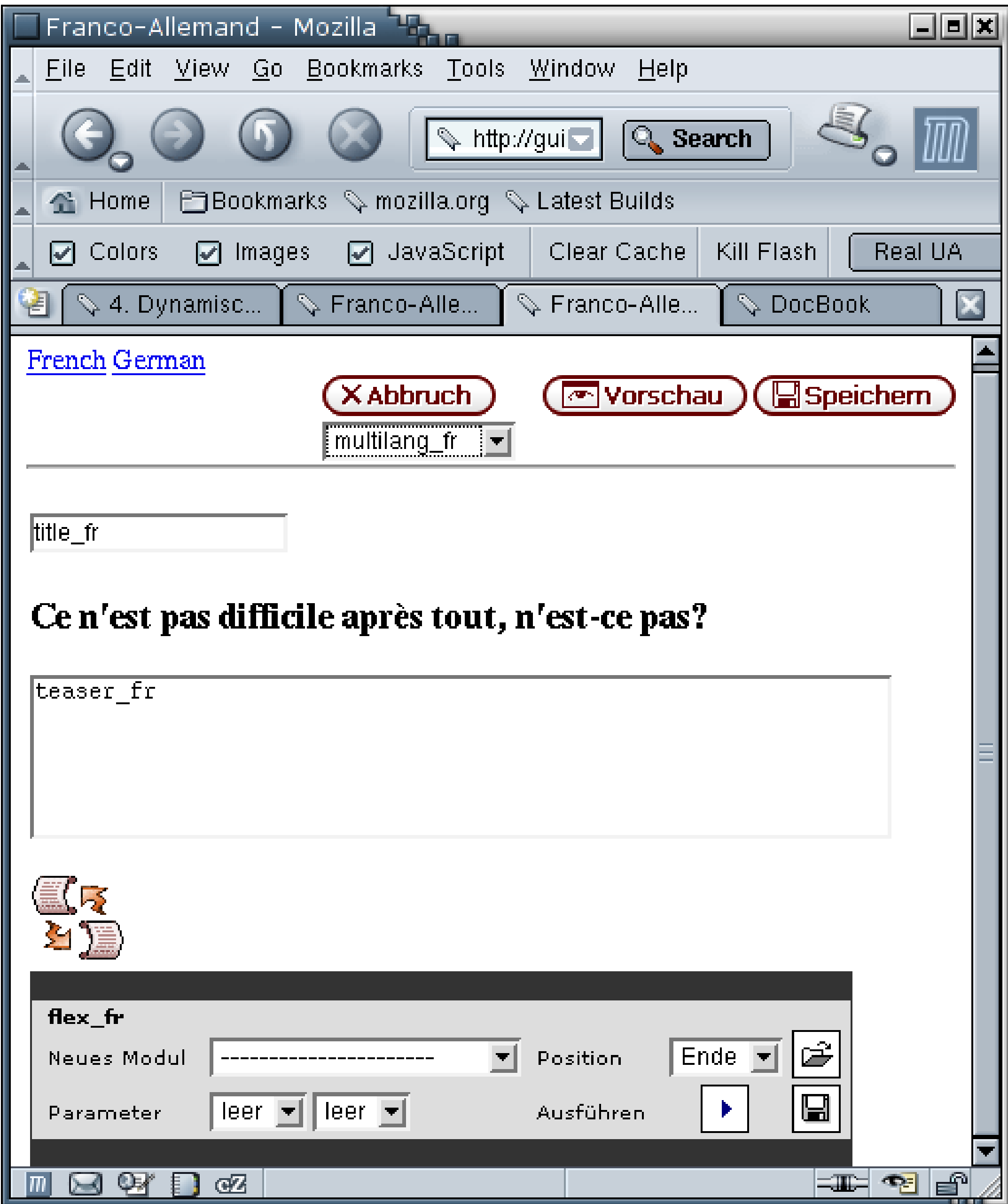
7.2. Der Flex-Flexxer

Die Verwendung von Flex-Modulen birgt eine weitere Schwierigkeiten im Zusammenhang mit Multi-Language-Seiten. Nehmen wir an, dass in der englischen Redaktion ein komplizierter Aufbau der englischen Seite mit Flex-Modulen erfolgte, und die anderen Sprachredaktionen jetzt vor dem Problem stehen, überhaupt erst einmal diesen Aufbau (inklusive der Parameter, die bei der Ausführung des Flex-Moduls gesetzt wurden) zu reproduzieren, bevor an eine Übersetzung des Contents zu denken ist.

Der Flex-Flexxer ist ein sehr junges Feature des Imperia-Template-Prozessors, das hier Abhilfe schaffen kann. Fügen wir zunächst einmal den entsprechenden Aufruf in unser Template (besser gesagt unser Code-Include) `multilang.htms` ein:

```
<!--INSERT_FLEX_FLEXXER-->
<!--INSERT_FLEXMODULE:INDEX=<!--CI_PARAM2-->:LABEL=flex_<!--CI_PARAM1-->
```

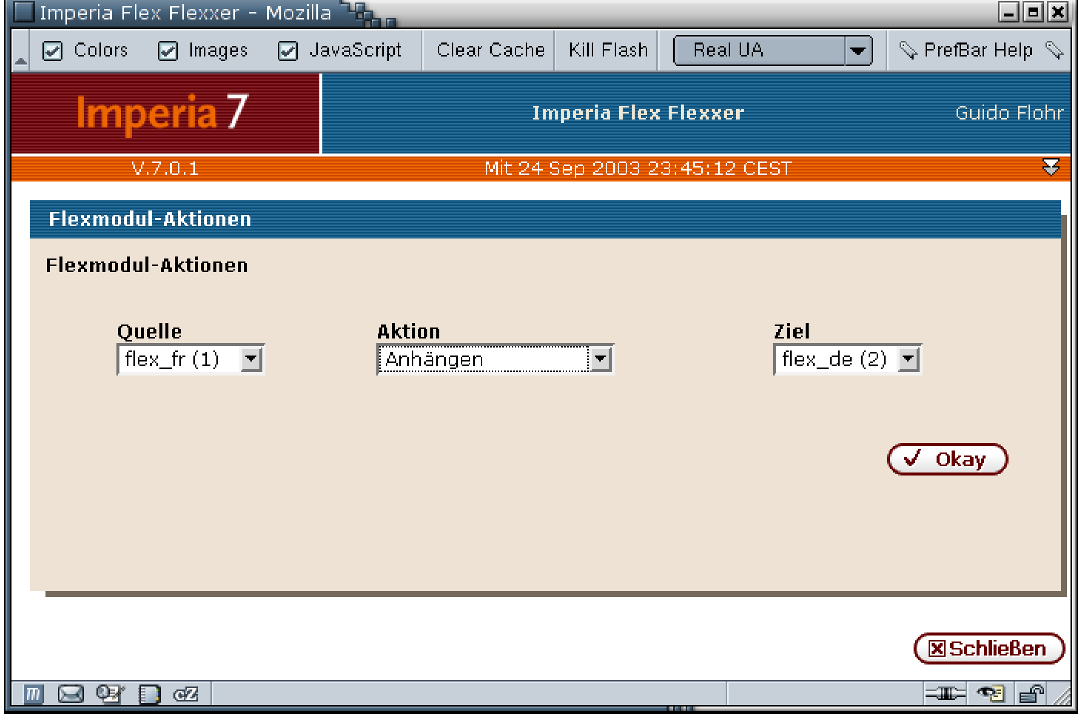
Rufen wir jetzt ein Dokument im Editier-Modus auf, stellen wir eine kleine Veränderung fest:



Der Flex-Flexxer in der Editieransicht

Die kleine Veränderung zuerst: Die Flex-Modul-Box enthält links oben plöztlich die Beschriftung „flex_fr“. Wenn wir uns noch einmal den Flex-Modul-Aufruf genauer anschauen, stellen wir fest, dass dies exakt dem zusätzlichen Parameter `LABEL` entspricht, den wir übergeben haben. Natürlich dürfen hier auch anwenderfreundlichere Namen verwendet werden.

Weiterhin fällt natürlich das Icon auf, dass der Template-Prozessor an der Stelle des Flex-Flexxer-Aufrufs eingefügt hat. Klicken wir auf das Icon, öffnet sich ein neues Fenster:



Erweiterte Flex-Aktionen durch den Flex-Flexxer

Die Bedienung des Flex-Flexxers dürfte relativ selbsterklärend sein. Es lassen sich komplette Flex-Modul-Sets (inklusive der Parameter und des Contents) kopieren, verschieben und löschen. Redakteure sind dadurch in der Lage, Flex-Module-Aufbauten aus anderen Sprachversionen als Grundlage der eigenen Version zu verwenden. Näheres kann dem Imperia-Programmierhandbuch entnommen werden.

7.3. Mehr Komfort durch DHTML

Die folgende Technik, mit der die Pflege mehrsprachiger Seiten in Imperia noch komfortabler wird, beruht auf einer Idee des Imperia-Partners Seitenbau (http://www.seitenbau.com/). Bislang liegen die einzelnen Sprachversionen in der Editieransicht vertikal untereinander, was gerade bei größeren Dokumenten mit vielen Sprachen sehr leicht zur Unübersichtlichkeit führt. Durch den Einsatz von DHTML lässt sich der Komfort hier stark verbessern. Die Idee von Seitenbau sieht vor, die Content-Pflege-Masken für die einzelnen Sprachen jeweils auf Layer zu verteilen, die über ein Menü am oberen Seitenrand angesteuert werden können, sprich, die Layer werden durch die

Anwahl der entsprechenden Menüpunkte sichtbar bzw. unsichtbar gemacht.

Die Vorteile dieser Technik werden wir gleich in vivo kennenlernen. Die Nachteile wollen wir aber natürlich auch nicht verschweigen:

- Eine vollkommen browserunabhängige Implementierung ist nur mit sehr großem Aufwand möglich. Da dies aber nur das Redaktionsinterface, und nicht etwa die publizierten Seiten betrifft, ist dies in der Regel zu verschmerzen, da man für die Redaktion(en) fast immer relativ hohe Mindestanforderungen an die Browsersoftware stellen kann.
- Gerade bei längeren Editiersessions kann diese Technik auf Kosten der Transparenz gehen. Eine unbeabsichtigte Änderung von Teilen der Seite kann leicht im Eifer des Gefechts übersehen werden, weil diese Änderung nicht unbedingt sichtbar ist.
- Die erhöhte Komplexität der Template-Programmierung kann leichter zu Fehlern führen. Wie eingangs erwähnt, erfordert der Einsatz von DHTML die Zulassung von JavaScript und Layern im Edit-Modus des Imperia-Template-Prozessors, was bei unsachgemäßer Verwendung zu Fehlfunktionen führen kann.
- Schließlich steigen durch die zusätzlichen DHTML-Bestandteile die Ladezeiten der Seiten im Redaktionsalltag. Diese Steigerung sollte nicht weiter ins Gewicht fallen, allerdings ist gerade bei einer sehr großen Zahl von parallel gepflegten Sprachversionen für die Redakteure natürlich weniger ersichtlich, dass die dargestellte Seite tatsächlich erheblich größer ist, als sie vermuten (da ja der größte Teil der Seite ausgeblendet wird).

Der Gewinn an Usability wird die erwähnten Nachteile in aller Regel leicht aufwiegen. Dennoch sollte jeder Punkt natürlich genau geprüft werden, bevor der zusätzliche Aufwand bei der Template-Programmierung in Kauf genommen wird.

Im Folgenden wollen wir eine solche Lösung skizzenhaft darstellen. Da dieses Dokument keine Anleitung zur browserunabhängigen DHTML-Programmierung sein will, wird eine Lösung dargestellt, die mit modernen Browsern (Mozilla 1.x, Internet Explorer 6.x, etc.) funktionieren wird, aber natürlich erheblicher Verfeinerungen bedarf, wenn die zu unterstützende Browser-Palette weiter gefasst wird.

7.3.1. Erzeugen der Layer

Die im Code-Include `multilang.htms` realisierten Eingabemasken für den Content müssen zunächst in Layer verpackt werden:

```
#IF ("<!--XX-editmode-->")
<div class="multi" id="div<!--CI_PARAM1-->">
#ENDIF
... Normaler Template-Code ...
```

```
#IF ("<!--XX-editmode-->")
</div>
#ENDIF
```

Unsere DHTML-Spielereien dienen lediglich der Verbesserung des Redaktionsinterfaces, und sollten demnach nur im Edit-Modus des Template-Prozessors aktiviert sein. Dies gilt durchgehend auch für alle weiteren Modifikationen, die ebenfalls durch Abfragen auf den Edit-Modus geschützt werden sollten, um die publizierten Seiten nicht unnötig aufzublähen.

Eine eindeutige ID des Layers gewinnen wir aus dem Sprachkürzel, das als Parameter an unser Code-Include übergeben wird. Aus Gründen der Browserkompatibilität sollte die ID keinen Unterstrich enthalten!

7.3.2. Die DHTML-Erweiterungen am Template

Der notwendige CSS- und JavaScript-Code ist sprachunabhängig und wird deshalb im Master-Template eingefügt. Wir ändern also die Datei `templatemultilang.htms` ab, und fügen folgenden Code innerhalb des HTML-`head`-Elements ein:

```
#IF ("<!--XX-editmode-->")
    <style type="text/css">
div.multi {
        position: absolute;
        left:10px;
        top: 60px;
        visibility: hidden;
}
    </style>
    <script language="Javascript">
function show_layer (id)
{
        id = 'div' + id;
        var all_layers = document.getElementsByTagName ('div');

        for (i = 0; i < all_layers.length; ++i) {
                var the_class = all_layers[i].className;
                if (the_class == null || the_class != 'multi')
                        continue;
                var the_layer = all_layers[i];
                the_layer.style.visibility = 'hidden';
        }

        var el = document.getElementById (id);
```

```
        el.style.visibility = 'visible';
}
    </script>
#ENDIF
```

Bei der Positionierung der Layer wird man eventuell etwas mit der vertikalen Lage herumexperimentieren müssen. Der gewählte Wert von 60 Pixeln sollte in Ordnung sein. Wird allerdings festgestellt, dass die Editierlayer die Imperia-Steuerelemente (insbesondere die Buttons für Speichern, Abbruch und Voransicht) überdecken, muss die Position verschoben werden, bis wieder eine einwandfreie Bedienbarkeit erreicht ist.

Das CSS-Attribut `visibility` ist zunächst für alle Layer auf `hidden` gesetzt, was für die Redakteure lästig und schwer zu durchschauen ist. Nachdem die Layer über die Code-Include-Kaskade eingebunden wurden, fügen wir am Ende des Templates daher nochmals ein wenig JavaScript ein, um sicherzustellen, dass stets ein Editierlayer sichtbar ist:

```
#IF ("<!--XX-editmode-->")
<script language="Javascript">
show_layer ('<!--XX-linguas-->');
</script>
#ENDIF
```

Die oben definierte JavaScript-Funktion `show_layer` erwartet als Argument ein Sprachkürzel, so wie sie in der Meta-Variablen `linguas` (als Liste!) abgelegt sind. Was wird hier nun aber vom Template-Prozessor eingefügt, wenn die Listen-Variable mit der bekannten XX-Syntax expandiert wird? Aus Kompatibilitätsgründen zu früheren Imperia-Versionen werden diese Variablen bei der XX-Expansion im Template im skalaren Kontext interpretiert, und statt der kompletten Liste wird das erste Element der Liste eingefügt. Da unser Workflow-Plug-In sicherstellt, dass die Liste `linguas` niemals leer ist, wird hier also immer der zuerst definierte Layer in den Vordergrund geholt.

An dieser Stelle wird man das System später sicher aufbohren müssen, da die Redakteure sehr bald bemängeln werden, dass sie nach jeder Aktion bei der ersten Sprache landen, und die gewünschte Sprache erst umständlich nach oben holen müssen. Die Behebung dieses Mangels ist eine reine Fleißangelegenheit: Man speichert die ID des aktuellen Layers in der JavaScript-Funktion `show_layer` in einer versteckten Formularvariable, und holt am Anfang diesen Layer in den Vordergrund. Imperia wird die Formularvariable - wie alle CGI-Variablen - als Metavariable abspeichern und persistent machen, so dass sich das System auch nach dem Speichern der Seite den zuletzt aktiven Layer „merkt".

7.3.3. Das Layer-Menü

Es fehlt noch das „Menü“ für die Umschaltung zwischen den Layern. Auch dieser Teil ist sprachunabhängig, aber da wir hier noch einmal etwas Perl-Code benötigen, setzen wir ihn nicht ins Template, sondern in unser Steuer-Include-File `multilang_setup.pl`. Den Anfang der Datei ändern wir dazu folgendermaßen:

```
my @linguas = $metainfo->getValues ('linguas');

use Locale::Language;
if ('EDIT' eq $mode) {
    foreach my $lingua (@linguas) {
        my $full_language = Locale::Language::code2language ($lingua);
        $full_language = $lingua unless defined $full_language;
        $new .= <<EOF;
<a href="javascript:show_layer ('$lingua');">$full_language</a>
EOF
    }
}
```

Das Perl-Modul `Locale::Language` gehört nicht zum Perl-Core wird aber mit Imperia zusammen ausgeliefert. Es erlaubt eine Zuordnung der von uns benutzten Sprachkürzel zu (englischen!) Sprachnamen, die wir für die Beschriftung unseres Menüs verwenden.

In der Praxis wird man sicher auf schönere Darstellungen für das Menü, z. B. durch die Verwendung von Länderfähnchen verfallen. Auch hier wird man sicherlich die Konvention mit den Sprachkürzeln zu schätzen wissen, und mit geringem Aufwand Bildlinks einfügen können.

8. Zusammenfassung

Das obige Szenario sollte zeigen, dass es mit einer gewissen Portion Imperia-Erfahrung möglich ist, eine flexibel aufgebaute, mehrsprachige Web-Site mit Hilfe von Imperia in wenigen Stunden aus dem Boden zu stampfen. Selbst, wenn man nicht alle Teile vollständig verstanden hat - insbesondere die in Perl geschriebenen Teile erschließen sich natürlich nur bei entsprechendem Perl-Know-How - sollte es doch relativ klar sein, an welchen Stellen man das Konzept aufbohren muss, um es an die eigenen Projekterfordernisse anzupassen. Im Laufe der Zeit wird man sicherlich eigene Bausteine hinzufügen, eigene Erfahrungen und Verbesserungen einfließen lassen, die prinzipielle Vorgehensweise dagegen dürfte sich auch in weitaus komplexeren Umgebungen bewähren.

Inhaltsverzeichnis

A. Klingonisch (tIhIngan Hol) 97

Anhang A. Klingonisch (tlhIngan Hol)

Das Unicode-Konsortium verweigert sich bislang beharrlich der Forderung, die Schriftzeichen der klingonischen Sprache *tlhIngan Hol* () in den Standard aufzunehmen. Die Geschichte dieser lang anhaltenden Debatte ist auf [Klingon-Unicode] dokumentiert. Allerdings reserviert der Unicode-Standard den Bereich von hexadezimal e000-f8ff für privaten Gebrauch. Beherzte Mitglieder der Linux-Gemeinde nutzten diesen Umstand aus, und schufen der diskriminierten Sprache schließlich doch eine Heimstatt, indem sie einen Teil der sogenannten *Linux-Zone* - den Teilbereich der privaten Unicode-Zone, dessen Verwendung unter allen Linux-Usern koordiniert wird - für *pIqaD* (), die reformierte klingonische Schrift, reservierten (vgl. [Anvin]). Das von [Schoen] beschriebene ursprüngliche Mandel-System wird scheinbar nicht mehr unterstützt.

Diese Zuordnung gilt heute als De-Fakto-Standard für *pIqaD*, und wurde zwischenzeitlich auch vom *Klingon Language Institute* (siehe [KLI]), dem Institut für Klingonische Sprache, geleitet von Dr. Lawrence Schoen, anerkannt. Nur wenige Zeichensätze enthalten die vollständigen Glyphen, die zur Darstellung des pIqaD nötig sind, weshalb die meisten verfügbaren Dokumente auf Klingonisch heute nur in der lateinischen Okrand-Transliteration verfügbar sind, oder vor der Publikation in Grafiken umgewandelt werden. Zu weiteren Aspekten von pIqaD, vgl. im übrigen die ausgezeichnete Darstellung von Zrajm C Akfohg von der Klingonska Akademien Uppsala (siehe [Klingonska]) in [Akfohg]).

Weiterführende Informationen

Referenzen wurden - soweit möglich - aus frei zugänglichen Quellen im Internet gewonnen.

World Wide Web

[Akfohg] Zrajm C Akfohg. *pIqaD, And How to Read It*. http://www.klingonska.org/piqad/ .

[Anvin] H. Peter Anvin. http://www.lanana.org/docs/unicode/unicode.txt .

[Czyborra] Rainer Czyborra. *czyborra.com*. http://czyborra.com/ [http://czyborra.com/] .

[HTML4.0] Dave Raggett. Arnaud Le Hors. Ian Jacobs. *HTML 4.0 Specification*. W3C Recommendation, revised on 24-April-1998. World Wide Web Consortium W3C®. 18. Dezember 1997. Copyright © 1997-1998 W3C®. http://www.w3.org/TR/1998/REC-html40-19980424/ .

[HTML4.01] Dave Raggett. Arnaud Le Hors. Ian Jacobs. *HTML 4.01 Specification*. W3C Recommendation 24 December 1999. World Wide Web Consortium W3C®. Copyright © 1997-1999 W3C®. http://www.w3.org/TR/html401/ [http://www.w3.org/TR/html401/] .

[IANA-LANGUAGE-TAGS] *LANGUAGE TAGS*. http://www.iana.org/assignments/language-tags .

[IANA-MEDIATYPES] *MEDIA TYPES*. ftp://ftp.rfc-editor.org/in-notes/iana/assignments/media-types/media-types . 16. Oktober 2001.

[KLI] *Klingon Language Institute*. http://www.kli.org/ .

[Klingon-Unicode] *Klingon-FAQ*. Is Klingon included in Unicode?. http://higbee.cots.net/~holtej/klingon/faq.htm#2.17 .

[Klingonska] *Klingonska Akademien*. http://www.klingonska.org/ .

[RFC821] Jonathan B. Postel. *SIMPLE MAIL TRANSFER PROTOCOL*. http://www.faqs.org/rfcs/rfc821.html . August 1982.

[RFC822] David H. Crocker. *STANDARD FOR THE FORMAT OF ARPA INTERNET TEXT MESSAGES*. http://www.faqs.org/rfcs/rfc822.html . 13. August 1982.

[RFC1489] A. Chernov. *Registration of a Cyrillic Character Set*. http://www.faqs.org/rfcs/rfc1489.html . Juli 1993.

[RFC2152] D. Goldsmith und M. Davis. *UTF-7*. A Mail-Safe Transformation Format of Unicode. http://www.faqs.org/rfcs/rfc2152.html . Mai 1997.

[RFC2279] F. Yergeau. Copyright © 1998 The Internet Society. *UTF-8*. a transformation format of ISO 10646. http://www.faqs.org/rfcs/rfc2279.html . Januar 1998.

[RFC2616] R. Fieldings, UC Irvine, J. Gettys, Compaq/W3C, J. Mogul, Compaq, H. Frystyk, W3C/MIT, L. Masinter, Xerox, P. Leach, Microsoft, T. Berners-Lee und W3C/MIT. Copyright © 1999 The Internet Society. *Hypertext Transfer Protocol -- HTTP/1.1*. http://www.faqs.org/rfcs/rfc2616.html . Juni 1999.

[RFC2821] J. Klensin. Copyright © 2001 The Internet Society. *Simple Mail Transfer Protocol*. http://www.faqs.org/rfcs/rfc2821.html . April 2001.

[RFC2822] P. Resnick. Copyright © 2001 The Internet Society. *Internet Message Format*. http://www.faqs.org/rfcs/rfc2822.html . April 2001.

[RFC3066] H. Alvestrand. Copyright © 2001 The Internet Society. *Tags for the Identification of Languages*. http://www.faqs.org/rfcs/3066.html . Januar 2001.

[Schoen] Lawrence M. Schoen. *Some Comments on Orthography*. http://www.kli.org/pdf/Orthography.pdf .

[Swift1726] Jonathan Swift. Lee Jaffe. *Gulliver's Travels*. http://www.jaffebros.com/lee/gulliver/contents.html [http://www.jaffebros.com/lee/gulliver/contents.html] .

[Unicode] Unicode Inc. . *Was ist Unicode?*. http://www.unicode.org/standard/translations/german.html [http://www.unicode.org/standard/translations/german.html] .

[Unicode3.0] Unicode Inc. . *Unicode 3.0*. http://www.unicode.org/book/u2.html [http://www.unicode.org/book/u2.html] .

[Unicode4.0] Unicode Inc. . *Unicode 4.0*. http://www.unicode.org/versions/Unicode4.0.0/ [http://www.unicode.org/versions/Unicode4.0.0/] .

[Winter] Dik T. Winter. *Standards*. http://www.cwi.nl/~dik/english/codes/stand.html [http://www.cwi.nl/~dik/english/codes/stand.html] .

[XHTML1.0] W3C HTML Working Group. *XHTML(TM) 1.0 The Extensible HyperText Markup Language (Second Edition)*. (Second Edition). A Reformulation of HTML 4 in XML 1.0. W3C Recommendation 26 January 2000, revised 1 August 2002. World Wide Web Consortium W3C®. 26. Januar 2000. Copyright © 2002 W3C®. http://www.w3.org/TR/xhtml1/ .

[XML-W3C] *Extensible Markup Language (XML)*. Copyright © 1996-2003 W3C®. http://www.w3.org/XML/ .

[XML-Specification] Tim Bray. Jean Paoli. C. M. Sperberg-McQueen. Eve Maler. *Extensible Markup Language (XML) 1.0*. W3C Recommendation 6 October 2000. World Wide Web Consortium W3C®. Copyright © 1997-1999 W3C®.

http://www.w3.org/TR/REC-xml/ .

Literaturhinweise

[De Bhaldraithe] M.A. Ph.D. D.Litt. M.R.I.A.. Tomás De Bhaldraithe. *English-Irish Dictionary*. With terminological additions and corrections. Copyright © 1959 Rialtas na hÉireann. An Deichiú Cló 1987. An Gúm, An Roinn Oideachais.

[Gaeilge-Béarla] *Gearrfhoclóir Gaeilge-Béarla*. Richview Browne & Nolan Ltd.. Baile Átha Cliath . Copyright © 1981 Rialtas na hÉireann.

[Hourani1991] Albert Hourani. *Die Geschichte der arabischen Völker*. Copyright © 1992 S. Fischer Verlag GmbH, Frankfurt am Main. ISBN 3-10-031832-3. 1992. 2. Auflage.

[Ó Siadhail] Mícheál Ó Siadhail. *Lehrbuch der irischen Sprache*. Copyright © 1985 Helmut Buske Verlag Hamburg. ISBN 3-87118-619-8.

[Tolstoi] Лев Николаевич Толстой. *Анна Каренина*. 1875-1878. ISBN 2-87714-264-7. © 1994, Bookking International, Paris.