Institut für Softwaretechnik

UNIVERSITÄT KOBLENZ · LANDAU

Fachbereich 4: Informatik

Unicode-Migration eines komplexen Software-Systems

Entwurf eines modulisierbaren Umstellungsprozesses und Anwendung als Fallstudie am Dokumenten-Management-System *PROXESS*

Diplomarbeit

vorgelegt am 31. August 2012 von

Sebastian Plitt

Betreuer:
Prof. Jürgen Ebert
Dr. Volker Riediger
Tassilo Horn

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden.

Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.

Koblenz, den 31. August 2012 . . . .

Zusammenfassung

Diese Arbeit befasst sich mit der Migration von Software-Systemen hin zur Verwendung des im Unicode-Standard definierten Zeichensatzes. Die Arbeit wird als Fallstudie am Dokumenten-Management-System *PROXESS* durchgeführt. Es wird ein Umstellungsprozess entworfen, der die Arbeitsschritte der Migration für das gesamte System und eine beliebige Zerlegung des Systems in einzelne Module definiert. Die Arbeitsschritte für die einzelnen Module können zu großen Teilen zeitlich unabhängig voneinander durchgeführt werden.

Für die Umstellung der Implementierung wird ein Ansatz zur automatischen Erkennung von Verwendungsmustern eingesetzt. Im abstrakten Syntaxbaum werden Sequenzen von Anweisungen gesucht, die einem bestimmten Verwendungsmuster zugeordnet werden. Ein Verwendungsmuster definiert eine weitere Sequenz von Anweisungen, die eine Musterlösung für die Unicode-basierte Handhabung von Strings darstellt. Durch das Anwenden einer Transformationsregel wird die ursprüngliche Anweisungssequenz in die zum Verwendungsmuster gehörende Anweisungssequenz überführt. Dieser Mechanismus ist ein Ausgangspunkt für die Entwicklung von Tools, die Transformationen von Anweisungssequenzen automatisch durchführen.

Abstract

This work deals with the migration of software systems towards the use of the character set defined in the Unicode standard. The work is performed as a case study on the document-management-system *PROXESS*. A conversion process will be designed that defines the working-steps of the migration for the entire system as well as an arbitrary decomposition of the system into individual modules. The working-steps for each module can be performed chronologically independent of each other to a great extend.

For the conversion of the implementation, an approach of automatic recognition of usage patterns is applied. The approach aims at searching the abstract syntax tree for sequences of program instructions that can be assigned to a certain usage pattern. The usage pattern defines another sequence of instructions that acts as an sample solution for that usage pattern. The sample solution demonstrates the Unicode-based management of strings for that usage pattern. By applying a transformation rule, the original sequence of instructions is transferred to the sequence of instructions exposed by the sample solution of the related usage pattern. This mechanism is a starting point for the development of tools that perform this transformation automatically.

Inhaltsverzeichnis

I. Einleitung 1

1. Zielsetzung und Anforderungen 4

2. Aufbau der Arbeit und Schreibkonventionen 8
2.1. Überblick über die Gliederung 8
2.2. Schreibkonventionen 9

II. Grundlagen 11

3. Zeichensätze und Unicode 13
3.1. Definitionen 13
3.2. Historischer Überblick 17
3.3. Der Unicode-Standard 19
3.4. String-Längen-Problematik 29
3.5. Zusammenfassung 31

4. Einführung in *PROXESS* 32
4.1. Datenmodell 33
4.2. Services 36
4.3. *PROXESS* Server 39
4.4. Zusammenfassung 49

5. Weitere Grundlagen 51
5.1. C-Style-Strings 51
5.2. *Microsoft*-spezifische Grundlagen 53
5.3. Zusammenfassung 65

6. Verwandte Arbeiten 67

III. Prozessdefinition und -anwendung **71**

7. Definition und Entwurf des Umstellungsprozesses **73**
7.1. Definitionen 73
7.2. Entwurf des Umstellungsprozesses 75
7.3. Anwendung des Umstellungsprozesses an *PROXESS* 86

8. Analyse und Umstellung des Datenschemas und Migration des Datenbestandes **93**
8.1. Unicode-Unterstüzung der Datenbanksysteme 95
8.2. Migration des Datenschemas 97
8.3. Exploration: Migration des Datenbestandes 107
8.4. Zusammenfassung 110

9. Analyse und Umstellung der Schnittstellen **112**
9.1. Angebotene RPC-Schnittstelle 114
9.2. Untersuchung der verwendeten Schnittstellen 130
9.3. Zusammenfassung 138

10. Analyse und Umstellung der Implementierung **139**
10.1. Definitionen und Beispiele 139
10.2. Überblick: Verwendungsmuster im *DatabaseManager* 148
10.3. Tool zur grundlegenden Herangehensweise an die automatische Erkennung von Verwendungsmustern mit *clang* 150
10.4. Zusammenfassung 151

IV. Evaluation, Fazit und Ausblick **153**

11. Aufwandsabschätzung **155**

12. Bewertung der Ergebnisse **157**
12.1. Bewertung des Umstellungsprozesses 157
12.2. Bewertung des automatischen Erkennungsverfahrens 158

13. Fazit **159**

V. Anhang **160**

Literaturverzeichnis **161**

A. Interviewprotokoll zur Anforderungserhebung **173**

B. Beispielprogramm für Microsoft RPC **177**

C. Auflistung der Datenbanktabellen von *PROXESS* **181**

C.1. Tabellen der Master-Datenbank 181

C.2. Tabellen der StorageManager-Datenbank 184

C.3. Tabellen einer Benutzer-Datenbank 186

D. Verwendung des Tools *PROXESS - Database2WideChar* **194**

Teil I.

Einleitung

Diese Arbeit befasst sich mit der **Migration** von Software-Systemen auf die Verwendung des im Unicode-Standard definierten Zeichensatzes. Sie wird als Fallstudie am Dokumenten-Management-System (DMS) *PROXESS* durchgeführt, das von der Akzentum GmbH[1] entwickelt und gewartet wird. Ein DMS bietet im wesentlichen Services zum Speichern, Anzeigen, Abrufen und Archivieren von beliebigen elektronischen Dokumenten.

Der **Unicode-Standard**[2] ist ein internationaler Standard, der vom Unicode Konsortium seit 1991 entwickelt und in regelmäßigen Abständen in Form einer neuen Version erweitert wird. Der Unicode-Zeichensatz wurde nach der Maßgabe entworfen, alle Zeichen aus allen relevanten Schriftsystemen der Welt zu enthalten. Neben dem Zeichensatz spezifiziert der Standard eine Reihe von Kodierungen zur eindeutigen Speicherung von Strings, sowie Algorithmen zur Sortierung und Normalisierung von Strings. In der aktuellen Version – Unicode 6.1, veröffentlicht im Januar 2012 – unterstützt der Standard 100 verschiedene Schriftsysteme mit insgesamt 110.181 Zeichen[3].

Dieser Standard stellt eine Alternative zu „klassischen" Zeichensätzen wie z.B. US-ASCII (mit 127 Zeichen) oder den 8-Bit ISO-Zeichensätzen (mit jeweils 256 Zeichen) dar, die im überwiegenden Teil der Fälle für genau ein Schriftsystem entworfen worden sind. Da der Unicode-Standard viele verschiedene Schriftsysteme unterstützt, stellt sein Einsatz einen wichtigen Grundstein bei der Internationalisierung von Software dar. Dies wird durch eine breite Unterstützung des Unicode-Standards von Betriebssystemen, Internet-Browsern, Datenbanksystemen und Programmiersprachen gezeigt.

Es gibt heute, Stand 2012, noch viele Software-Systeme, die meist aus historischen Gründen Zeichensätze verwenden, die nur für ein Schriftsystem entworfen worden sind. Handelt es sich bei Anwendern solch eines Software-Systems um Unternehmen mit einer internationalen Ausrichtung, kann eine so deutliche Einschränkung bei der Zeichenkodierung einen Grund darstellen, zu einem Konkurrenzprodukt zu wechseln, das Unicode unterstützt. Für potentielle neue Anwender stellt sich das Software-System schon im voraus als nicht attraktiv dar.

Das **Software-System *PROXESS*** hat Anwender mit einer internationalen Ausrichtung und ist im jetzigen Zustand unter der Annahme programmiert, dass ein 8-Bit Zeichensatz zur Speicherung und zur Verarbeitung von Strings verwendet wird. Zudem ist das System nicht darauf ausgelegt, gleichzeitig mehrere verschiedene 8-Bit Zeichensätze zu

[1] `https://www.akzentum.de` [Letzter Zugriff am 25.07.2012]
[2] `http://www.unicode.org` [Letzter Zugriff am 25.07.2012]
[3] `http://de.wikipedia.org/wiki/Unicode` [Letzter Zugriff am 25.07.2012]

unterstützen, was die gleichzeitige Handhabung mehrerer Schriftsysteme unmöglich macht. Aus diesem Grund soll *PROXESS* in einer neuen Version so angepasst werden, dass zur Speicherung und zur Verarbeitung von Strings der Unicode-Zeichensatz verwendet wird.

Diese Arbeit erarbeitet und erprobt die Grundlagen und das Vorgehen für die Migration von *PROXESS*. Die Anwendung an *PROXESS* hat an vielen Stellen Auswirkungen auf die Herangehensweise und das konkrete Vorgehen in Bezug auf spezifische Details. Trotzdem sollen die grundlegenden Überlegungen und Entscheidungen dieser Arbeit einen wiederverwendbaren Charakter haben: Die Problemlösungen werden möglichst allgemein formuliert und dann konkret am Beispiel *PROXESS* angewendet.

Im folgenden werden die Anforderungen und Ziele aufgeführt, die die Rahmenbedingungen dafür festlegen, wie das Software-System *PROXESS* in eine Version zu überführen ist, in der Strings konform zum Unicode-Standard gehandhabt werden.

1. Zielsetzung und Anforderungen

Das **Ziel** dieser Arbeit ist es, die Grundlagen für eine geordnete Migration des Softwaresystems *PROXESS* hin zu der Verwendung des Unicode-Zeichensatzes zu schaffen. Die Aufgabe wird von der Akzentum GmbH und dem Institut für Softwaretechnik (IST) der Universität Koblenz-Landau gestellt. Die Anforderungen an die Arbeit von Seiten der Akzentum GmbH wurden in einem Interview erfasst, das im Anhang in Kapitel A protokolliert ist, und mit den zuständigen Mitarbeitern des IST abgestimmt. Die folgende Auflistung gibt die Anforderungen in komprimierter Form wieder.

1. Die Migration von *PROXESS* muss nebenläufig zur Weiterentwicklung und Wartung des Systems ablaufen.
2. Die Unicode-Unterstützung von *PROXESS* muss mit einem Release erstmals und vollständig implementiert werden.
3. Die alte „ANSI"-Schnittstelle von *PROXESS* muss für eine bestimmte Zeit parallel zur neuen Unicode-Schnittstelle betrieben werden.
4. Die umgestellten Teile des Systems sollen mit Unit-Tests getestet werden.
5. Der Aufwand für die Migration des gesamten Systems soll abgeschätzt werden.
6. Die Arbeit muss problematische Code-Stile aufdecken.
7. Die Arbeit muss Konventionen zur richtigen Handhabung von Unicode-Strings festlegen.
8. Die Ergebnisse sollen in einer Guideline niedergelegt werden.
9. Microsoft-Richtlinien und -Standards sind für die Lösung maßgeblich.

Aus diesen Anforderungen leiten sich drei primäre Zielsetzungen für diese Arbeit ab, die in den nächsten Abschnitten diskutiert werden.

1. Problematische Unterschiede zwischen 8-Bit Zeichensätzen und Unicode erkennen. Die Unterschiede zwischen der Handhabung von „klassischen" Strings, bei denen ein Byte immer genau einem Zeichen entspricht, und Unicode-Strings müssen erarbeitet werden. Diese Untersuchung ist wichtig um entscheiden zu können, was genau unter dem Begriff „problematische Code-Stile" aus Anforderung 6 zu verstehen ist. Dazu muss genau definiert werden, was unter einem Zeichensatz zu verstehen ist, und darauf aufbauend müssen Annahmen die für 8-Bit Zeichensätze gelten mit Unicode verglichen werden.

2. Entwurf eines Umstellungsprozesses. Es soll ein Umstellungsprozess entworfen werden, der die notwendigen Arbeitsschritte für die Migration des Systems festlegt. Der Umstellungsprozess wird möglichst allgemein formuliert und dann auf *PROXESS* angewendet. Der Umstellungsprozess wird nach den Maßgaben aus den Anforderungen 1 - 4 entworfen. Der Umstellungsprozess wird an einem **repräsentativen Teil des Systems** exemplarisch durchgeführt.

Die Anwendung des Umstellungsprozesses in dieser Arbeit umfasst die Analyse und Umstellung des Datenschemas von *PROXESS* und die Analyse und Umstellung der Schnittstellen des Programms *DatabaseManager*. Für die Migration des eigentlichen Quelltextes wird ein Ansatz zur Mustererkennung mit Hilfe von statische Code-Analyse entwickelt.

3. Analyse des Source-Codes zur Mustererkennung. Zur Realisierung der Anforderungen 5 - 8 wird Verständnis über den Zusammenhang von Quelltext und der Verarbeitung von Strings benötigt. Dazu ist es notwendig, alle Aufgaben, die im Zusammenhang mit Strings stehen, zu identifizieren. Mit dem Begriff **Verwendungsmuster** wird ein Begriff eingeführt, mit dem alle Verwendungsformen von Strings kategorisiert werden.

Durch Analyse des Quelltextes sollen Sequenzen von Anweisungen identifiziert werden, die ein bestimmtes Verwendungsmuster realisieren. Diese Sequenzen heißen **Ausprägungen von Verwendungsmustern**. Ausprägungen mit großer Ähnlichkeit werden zusammen als spezifische **Variante des Verwendungsmusters** aufgefasst. Eine Variante besitzt eine generische Beschreibung, die Menge von Ausprägungen einheitlich beschreibt.

Für jede Variante soll eine Transformationsregel erstellt werden. Eine **Transformationsregel** beschreibt eine Transformation, deren Anwendung den Quelltext von der alten

Form in die neue Unicode-fähige Form überführt.

In dieser Arbeit wird ein Ansatz zur **automatischen Erkennung** von Ausprägungen von Verwendungsmustern verfolgt. Die Analyse soll auf Basis des abstrakten Syntaxbaumes stattfinden und die Ergebnisse der Analyse sollen genaue Stellenangaben zum Quelltext liefern. Daher muss ein geeigneter C++-Parser gefunden werden, der einen abstrakter Syntaxbaum erzeugt, der die Struktur des (nicht-geparsten) Quelltextes ausreichend genau wiedergibt. Durch die automatische Erkennung soll der Suchraum für umzustellende Stellen im Quelltext eingeschränkt werden. Durch die Zuordnung von Anweisungen zu Varianten von Verwendungsmustern wird die Suche nach der „richtigen" Unicode-Lösung vereinfacht.

Diese Untersuchungen können als Ausgangspunkt für die **Entwicklung eines Tools** dienen, das in die Entwicklungsumgebung (im Fall von *PROXESS* ist das *Microsoft Visual Studio 2008*) integriert wird und die Entwickler, die die Migration durchführen, unterstützt.

Für das Ziel 3 wird ein Schwerpunkt auf die Formulierung der Verwendungsmuster und die automatischen Erkennung von Verwendungsmustern gelegt. Die Entwicklung eines Tools kann aus Zeitgründen nur im Anschluss an diese Arbeit verfolgt werden. Diejenigen Teile, die in dieser Arbeit behandelt werden, werden allerdings mit einem Blick auf die mögliche Entwicklung solch eines Tools durchgeführt.

Zusammenfassung des Erfolgs

Die Problematik in der unterschiedlichen Handhabung von Strings kann im Grundlagenteil erarbeitet werden und dient als Entscheidungsgrundlage für Festlegungen zur vereinheitlichten Handhabung von Strings.

Der im Hauptteil entworfene Umstellungsprozess zerlegt die Migration des Systems in überschaubare und getrennt durchführbare Teilprozesse und erweist sich als hilfreicher Leitfaden bei der Durchführung der Migration.

Die ersten beiden Ziele der Arbeit konnten erreicht werden.

Der Ansatz zur automatischen Erkennung von Verwendungsmustern, der in dieser Arbeit präsentiert wird (Kapitel 10) soll mit dem C++-Frontend *clang* implementiert wer-

den. Diese Implementierung konnte ansatzweise realisiert werden. Leider war keine Zeit um das Thema ausführlich auszuarbeiten.

2. Aufbau der Arbeit und Schreibkonventionen

Dieses Kapitel gibt einen Überblick über die Gliederung der Arbeit (Abschnitt 2.1) und legt Konventionen für Hervorhebungen, Abbildungen und Quellenangaben/Literaturverweise fest (Abschnitt 2.2).

2.1. Überblick über die Gliederung

Dieser Abschnitt gibt einen Überblick über die Gliederung dieser Arbeit. Der Haupttext gliedert sich in die folgenden vier Teile und einen Teil für den Anhang. Die vier Teile des Haupttextes sind die Folgenden:

1. Einleitung
2. Grundlagen
3. Prozessdefinition und -anwendung
4. Zusammenfassung, Fazit und Ausblick

Jeder Teil wird wiederum durch einzelne Kapitel gegliedert. Jeder Teil beginnt mit einem kurzen einleitenden Text, der einen Ausblick auf die Inhalte der enthaltenen Kapitel zusammenfasst.

Jedes Kapitel wird mit einem Text eingeleitet, der einen Überblick über die anschließende Diskussion gibt. Jedes (längere) Kapitel wird durch einen abschließenden Abschnitt zusammengefasst, der die Erkenntnisse und Ergebnisse des Kapitels reflektiert. Bei sehr kurzen Kapiteln – z.B. den Kapiteln der Teile „Einleitung“ und „Zusammenfassung, Fazit und Ausblick“ – ist einen zusammenfassender Abschnitt nicht sinnvoll.

2.2. Schreibkonventionen

Dieser Abschnitt legt die Schreibkonventionen fest, die für diese Arbeit gültig sind. Die Festlegungen beziehen sich zum einen auf Hervorhebungen im Fließtext (Abschnitt 2.2.1) und zum anderen auf Abbildungen und Bezeichnerwahl (Abschnitt 2.2.2).

2.2.1. Hervorhebungen im Fließtext

Dieser Abschnitt erläutert die verschiedenen Arten von Hervorhebungen im Fließtext, die in dieser Arbeit verwendet werden. Es werden drei Arten von Hervorhebungen im Fließtext verwendet:

1. Firmen- oder Produktnamen werden in *Kursivschrift* geschrieben, z.B. „*PROXESS* von der *Akzentum GmbH*" oder „*Microsoft SQL Server*".
2. Bezeichner und Literale aus Quelltext oder Abbildungen werden in `Typewriterschrift` geschrieben, z.B. „Ein Dokumenttyp wird in der Abbildung durch die Klasse `DocumentType` dargestellt." oder „Die Variable `length` hat den Wert `13`.".
3. Werden wichtige Begriffe eingeführt wird **Fettdruck** verwendet. Auf diese Weise werden auch einige wichtige Phrasen aus Abschnitten hervorgehoben. Beispiele: „Ein **Dokument** ist die zentrale Dateneinheit von *PROXESS*." oder „[...], wobei die Kommunikation durch ein **Client-/Server-Modell** realisiert wird."

Ggf. wird die Hervorhebungsart 3. mit einem der beiden anderen Typen kombiniert, z.B. „Ein **Dokument** (Abb. **`Document`**) ist die zentrale Dateneinheit von *PROXESS*." oder „***PROXESS*** ist ein Dokumenten-Management-System [...]. Ein **Dokumenten-Management-System** ist [...]".

2.2.2. Abbildungen und Bezeichnerwahl

Dieser Abschnitt beschreibt, welche Arten von Abbildungen und Diagrammen in dieser Arbeit verwendet werden.

Der überwiegende Teil der Abbildungen sind UML-Klassen- oder Aktivitäts-Diagramme ([Obj11a][Obj11b]). Sie werden bei der Beschreibung von *PROXESS*, dem Entwurf des Umstellungsprozesses und der Analyse und Umstellung der Implementierung verwendet. Diejenigen Abbildungen, mit denen Datenbanktabellen

beschrieben werden, werden durch eine Legende erläutert. Die übrigen Abbildungen haben einen skizzenhaften Charakter und ihre Bedeutung wird im Zusammenhang mit dem Fließtext deutlich.

In den Abbildungen werden i.d.R. englische Bezeichner gewählt. Im Fließtext werden allerdings deutsche Bezeichner verwendet, die beim erstmaligen Auftreten mit einem Verweis auf den englischen Begriff in der Abbildung versehen werden. So soll eine „verdenglischung" des Textes vermieden werden, die den Lesefluss behindern könnte. Die Aktivitätsdiagramme zur Beschreibung des Umstellungsprozesses stellen eine Ausnahme dar. Hier werden in den Diagrammen deutsche Bezeichner gewählt, weil die Namen der Aktivitäten teils lange Verbalphrasen sind. Eine Übersetzung dieser Verbalphrasen ins Englische erscheint an dieser Stelle nicht sinnvoll.

2.2.3. Literatur- und Quellenangaben

Dieser Abschnitt legt die zwei verwendeten Arten fest, in denen auf Literatur, Dokumentation oder andere Quellen verwiesen wird.

1. Quellen oder Literatur, aus denen/der wörtlich oder sinngemäß zitiert wird (z.B. Verweise auf die Unicode-Dokumentation [Uni11, DW12b, DW12a]), oder deren Inhalt für das Verstehen der Arbeit wichtig ist (z.B. Verweise auf die UML-Spezifikation [Obj11a, Obj11b]). werden im Anhang im Literaturverzeichnis aufgeführt. Im Fließtext finden sich an den (sinngemäß) zitierten Stellen Verweise auf die entsprechenden Einträge im Literaturverzeichnis.
2. Informative Verweise auf Homepages von Projekten oder Internet-Artikel, die einen für die Arbeit weniger relevanten Sachverhalt genauer erläutern, werden als Fußnoten auf der verweisenden Seite aufgeführt. Ein Beispiel dafür ist ein Wikipedia-Artikel, der den Aufbau eines Compilers allgemein beschreibt[1].

Alle Quellen, die auf Internetseiten verweisen, werden mit einem Hyperlink und einem Verweis auf das letzte Zugriffsdatum des Autors auf diese Internetseite versehen.

[1] Wikipedia, die freie Enzyklopädie - Compiler – `http://de.wikipedia.org/wiki/Compiler` [Letzter Zugriff am 20.08.2012]

Teil II.

Grundlagen

Dieser Teil der Arbeit führt die für das Verständnis der Arbeit notwendigen Grundlagen ein.

Kapitel 3 führt in das Thema Zeichensätze und Unicode ein und stellt einen „klassischen" 8-Bit-Zeichensatz dem im Unicode-Standard definierten Zeichensatz gegenüber. Dabei wird deutlich, dass einige grundlegende Annahmen über „klassische" Strings verworfen werden müssen, wenn Unicode verwendet wird.

Kapitel 4 führt den Begriff Dokumenten-Management-System (DMS) ein und stellt das Software-System *PROXESS* vor. Die von einem DMS angebotenen Services und verarbeiteten Daten werden in der Terminologie von *PROXESS* beschrieben. Das Datenmodell von *PROXESS* sowie eine Auswahl von spezifischen Aspekten von *PROXESS* werden näher beschrieben.

Kapitel 5 führt weitere – größtenteil *Microsoft*-spezifische – Grundlagen ein. Diese Einführungen werden mit einem besonderern Blick auf Zeichensätze und Strings geführt. Aus den Betrachtungen lassen sich bereits einige grundlegende Entscheidungen für die Unicode-basierte Version von *PROXESS* ableiten.

Zuletzt werden in Kapitel 6 verwandte Arbeiten vorgestellt. Dabei wird das Thema Unicode-Migration vom Thema Softwareinternationalisierung oder -lokalisierung abgerenzt. Die Entscheidung für die Verwendung des Unicode-Zeichensatzes, stellt lediglich eine sinnvolle Grundlage für die Internationalisierung eines Software-Systems dar.

3. Zeichensätze und Unicode

Dieses Kapitel führt in das Thema Zeichensatz und Zeichenkodierung in Rechen- bzw. Software-Systemen ein. Dazu werden in Abschnitt 3.1 zunächst der Begriff Zeichensatz und weitere verwandte Begriffe definiert.

In Abschnitt 3.2 wird ein Überblick über traditionelle und weit verbreitete Zeichensätze gegeben.

In Abschnitt 3.3 wird der Unicode-Standard eingeführt, der aus dem Unicode-Zeichensatz, mehreren Unicode-Kodierungen (Unicode-Encodings) und einer Reihe von Algorithmen besteht.

Abschließend wird in Abschnitt 3.4 ein „klassischer" 8-Bit-Zeichensatz dem Unicode-Zeichensatz (bzw. den relevanten Unicode-Kodierungen) gegenübergestellt. In diesem Zusammenhang wird festgestellt, dass viele Annahmen über Strings, die für 8-Bit-Zeichensätze gültig sind, bei der Verwendung des Unicode-Zeichensatzes verworfen werden müssen. Die wesentlichen Unterschiede in der Handhabung von String werden unter dem Begriff String-Längen-Problematik zusammengefasst.

3.1. Definitionen

Im Folgenden sei Y definiert als die Menge aller **Zeichen**. Zu Y gehören **Schriftzeichen**, die durch eine **Glyphe** graphisch dargestellt werden. Zu den Schriftzeichen gehören zum Beispiel lateinische Buchstaben, arabische Ziffern, Satzzeichen oder chinesische Schriftzeichen. Auch Zeichen, die keine Glyphe haben, oder die nur im Zusammenhang mit anderen Zeichen sinnvoll darstellbar sind, sind Teil von Y: Dazu gehören z.B. verschiedene Arten von **Leerzeichen** sowie **Steuerzeichen** wie Backspace (`BS`), Wagenrücklauf (carriage return, `CR`) oder Zeilenumbruch (line feed, `LF`). Diese Zeichen können je nach Zusammenhang ebenfalls eine explizite Darstellung in Form einer Glyphe besitzen: z.B. wird in Textverarbeitungsprogrammen das Zeichen ¶ häufig dazu

verwendet, um das Ende eines Paragraphen darzustellen. In Code-Beispielen wird das Zeichen ␣ verwendet, um das Vorhandensein eines Leerzeichens explizit darzustellen.

Ligaturen sind Glyphen, die mehrere Zeichen zu einer einzelnen graphischen Darstellung zusammenfügen. Auch Ligaturen werden als Elemente von Y aufgefasst. Beispiele für Ligaturen sind die Zeichen *Æ* oder *œ*.

Ein **diakritisches Zeichen** ist ein Zeichen, das an einer bestimmten Stelle eines Schriftzeichens angefügt wird, um dadurch ein neues Zeichen darzustellen. Beispiele für diakritische Zeichen sind die Zeichen ¨ (Punktierung wie in *ä*, *ö* oder *ü*) oder ´ (accent grave). Diakritische Zeichen sind Elemente einer Hilfsmenge Y_2. Diakritische Zeichen in Kombination mit einem Basis-Zeichen (z.B. *ä* oder *é*) können als **eigenständige Schriftzeichen** als Elemente von Y definiert werden oder als zusammengesetzte Zeichen. Ein **zusammengesetztes Zeichen** ist ein Tupel (B, C) mit $B \epsilon Y$ und $C \epsilon Y_2$. Z.B. werden diese Kombinationen in den Zeichensätzen der ISO-8859-Familie ausschließlich als eigenständige Schriftzeichen definiert. Im Gegensatz dazu werden im Unicode-Zeichensatz in vielen Fällen beide Alternativen definiert.

3.1.1. Zeichensatz, Codepoints und Codeunits

Ein **Zeichensatz** (engl. **Character-Set**) Z ist ein Tupel $(CSet, CNum)$ wobei

$$CSet \subseteq Y \bigcup Y_2$$

die Menge aller in Z definierten Zeichen definiert und die injektive Abbildung

$$CNum : CSet \to \mathbb{N}_0$$

jedem dieser Zeichen eine eindeutige nicht-negative Zahl zuordnet. Sinnvollerweise enthält ein Zeichensatz alle Zeichen eines oder mehrerer Schriftsysteme. Ein **Schriftsystem** ist ein System von darstellbaren Zeichen, das dazu verwendet werden kann, Ausdrücke einer Sprache in Symbolform darzustellen.

Elemente des Wertebereiches von $CNum$ heißen **Codepoints** (oder deutsch: **Codepunkte**). Der Wertebereich selbst wird **Code-Set** genannt. Dass $ran(\, CNum\,)$ n-elementig ist, kann sprachlich so ausgedrückt werden: *„Der Zeichensatz definiert n Codepunkte (bzw. Codepoints)“*. O.B.d.A. umfasst dann der Wertebereich der Abbildung $CNum$ das offene Intervall $[0, n[$.

Zum Beispiel wird für einen 8-Bit-Zeichensatz der Wertebereich von $CNum$ mit

$$ran(\, CNum\,) \;=\; \{\, x \,|\, x >= 0 \,\land\, x < 2^8 \,\}$$

eingeschränkt. Das bedeutet, dass ein 8-Bit-Zeichensatz 256 Codepunkte definiert, die im Intervall [0, 256[liegen.

3.1.2. Kodierung, Encoding und Decoding

Eine **Kodierung** K (auch **Codec**) für einen Zeichensatz Z wird als Tupel $K := (B, f)$ definiert. Die Kodierung überführt jedes Zeichen aus Z in eine Sequenz von **Codeunits** (deutsch: **Codewörtern**). Dabei ist B eine positive ganze Zahl, die die Bitbreite eines einzelnen Codewortes angibt. Die bijektive Funktion

$$f : ran(\, CNum\,) \rightarrow \{\, x \,|\, x >= 0 \,\land\, x < 2^B \,\}^{\mathbb{N}}$$

heißt **Kodierungsfunktion** und definiert, wie ein Codepunkt in eine Folge aus Codewörtern zu überführen ist. Das Anwenden der Kodierungsvorschrift wird **Encoding** genannt. Das Anwenden der Umkehrung der Kodierungsfunktion wird **Decoding** genannt.

Die Folge von Codewörtern, die den Codepunkt CP darstellt, wird **kodierte Darstellung** oder **Kodierung** von CP genannt.

Wird von einer B-Bit-Kodierung gesprochen, bedeutet dies, dass die durch das Encoding erzeugten Codewörter B-Bit groß sind. In dieser Arbeit werden 8-, 16- und 32-Bit-Kodierungen behandelt.

Unterscheidung zwischen Fixed-Width- und Variable-Width-Encoding. Der Begriff **Fixed-Width-Encoding** bezeichnet Encodings, bei denen die Codewort-Folgen aller Zeichen die gleiche Anzahl an Elementen haben. Gibt es mindestens zwei Zeichen, deren Codewort-Folgen unterschiedlich viele Elemente haben, wird von einem **Variable-Width-Encoding** gesprochen. Ist die Bitbreite des Encodings `8`, wird ein Variable-Width-Encoding auch **Multi-Byte-Encoding** genannt.

Beispiele für Kodierungen. **ISO-8859-15 *(*Latin-9*)*** ist ein 8-Bit-Zeichensatz, der (immer) mit einem 8-Bit Encoding und der Identitätsfunktion als Kodierungsvorschrift

dargestellt wird. Dieses Encoding `(8, ID)` ist ein Fixed-Width-Encoding.
UTF-8 ist ein 8-Bit-Encoding für den Unicode-Zeichensatz. Die Kodierungsfunktion schreibt vor, dass die Codepunkte `0` bis `127` mit jeweils einem Codewort dargestellt werden. Die übrigen Codepunkte werden mit mindestens zwei und höchstens vier Codewörtern dargestellt. Es handelt sich bei UTF-8 also um ein Multi-Byte-Encoding.

3.1.3. Strings

Ein **String** ist eine endliche Folge $(a_i) := \{a_1, a_2, ..., a_n\}$ von Zeichen aus einem Zeichensatz Z, sodass gilt $\forall i : a_i \epsilon CSet$. Die rechnerinterne Darstellung eines Strings (b_i) ist eine Folge von Codeunits, die durch das Aneinanderfügen aller derjenigen Folgen entsteht, die durch die Anwendung einer Kodierungsfunktion f auf jedes einzelne Zeichen a_i enstehen. Daher ist diese Darstellung immer an die zu f gehörige Kodierung K des Zeichensatzes Z gebunden.

$$(b_i) := f(a_1) \circ f(a_2) \circ ... \circ f(a_n)$$

Der Operator $\circ$ zum Aneinanderfügen von Folgen ist dabei frei an den Konkatenationsoperator für formale Sprachen aus der theoretischen Informatik angelehnt.

3.1.4. Darstellungsformen von Zeichen und Strings

Der Begriff **kodierte Darstellung** drückt die rechnerinterne Repräsentation eines Zeichens oder Strings als Folge von Codeunits aus, die aus der Anwendung der entsprechenden Kodierungsfunktion hervorgeht. Der Begriff **Ausgabedarstellung** drückt die Präsentation eines Zeichens als Glyphe (oder eines Strings als Menge von Glyphen) auf einem Ausgabegerät (Bildschirm oder Drucker) aus.

Wird der allgemeine Begriff **Darstellung** im Zusammenhang mit Zeichen oder Strings verwendet, ergibt sich aus dem Zusammenhang, ob es sich um die kodierte Darstellung oder um die Ausgabedarstellung handelt.

3.2. Historischer Überblick

Die *IANA*[1] pflegt ein Zuletzt Ende 2011 aktualisiertes Dokument[2], das die Namen (und Aliase) aller bekannten im Internet verwendeten Zeichensätze auflistet. In diesem Schriftstück werden **230 verschiedene Zeichensätze** identifiziert.

3.2.1. ASCII-Zeichensatz

Der wichtigste Standard ist sicherlich `ANSI_X3.4-1968`, besser bekannt unter den geläufigeren Namen **ASCII** oder **US-ASCII**. Der Zeichensatz wurde 1963 eingeführt und liegt seit 1968 in der bis heute gültigen Form vor. Es handelt sich um einen 7-Bit Zeichensatz, d.h. es sind 128 Zeichen definiert. Der Codebereich von `0x00` bis `0x20` (33 Stellen) sowie der Code 0x7F sind mit Steuerzeichen belegt. Die übrigen 94 Stellen (`0x21` bis `0x7E`) sind mit Schriftzeichen belegt. Darin sind das lateinische Alphabet in jeweils Groß- und Kleinschreibweise, die arabischen Ziffern, Satzzeichen und weitere Sonderzeichen (Klammern, Währungssymbole, grundlegende arithmetische Operatoren, etc.) enthalten.

3.2.2. 8-Bit Zeichensätze

Es gibt eine Reihe von 8-Bit Zeichensätzen, von denen die meisten abwärtskompatibel zu US-ASCII sind. Das bedeutet, dass die Codestellen von `0x00` bis `0x7F` identisch zu US-ASCII sind. Lediglich die Codestellen von `0x80` bis `0xFF` werden für neu definierte Zeichen verwendet.

ISO 8859-Familie. Die **ISO/EIC 8859-Familie** beinhaltet eine Reihe von Normen, die durch die ISO[3] bzw. die IEC[4] definiert werden. Insgesamt sind 15 Teilnormen definiert, von denen alle kompatibel zu US-ASCII sind und die für verschiedene sprachliche Regionen der Welt zusätzliche Zeichen definieren. Tabelle 3.1 listet alle gültigen Teilnormen auf.

[1] Internet Assigned Numbers Authority `http://www.iana.org/` [Letzter Zugriff am 31.05.2012]

[2] http://www.iana.org/assignments/character-sets

[3] International Organization for Standardization `http://www.iso.org/iso/home.html` [Letzter Zugriff am 31.05.2012]

[4] International Electrotechnical Commission `http://www.iec.ch/index.htm` [Letzter Zugriff am 31.05.2012]

Teilnorm	Bechreibung
-1	Latin-1, Westeuropäisch
-2	Latin-2, Mitteleuropäisch
-3	Latin-3, Südeuropäisch
-4	Latin-4, Nordeuropäisch
-5	Kyrillisch
-6	Arabisch
-7	Griechisch
-8	Hebräisch
-9	Latin-5, Türkisch
-10	Latin-6, Nordisch
-11	Thai
-13	Latin-7, Baltisch
-14	Latin-8, Keltisch
-15	Latin-9, Westeuropäisch
-16	Latin-10, Südosteuropäisch

Tabelle 3.1.: Teilnormen der ISO-8859-Familie

Für neue Schriftzeichen ist in allen Teilnormen nur der Codebereich `0xA0` bis `0xFF` vorgesehen. Analog zum US-ASCII-Standard sind die ersten 32 Zeichen (der oberen 128 Bit) `0x80` bis `0x9F` mit Steuerzeichen belegt. Die deutschen Umlaute zum Beispiel sind in allen Latin-Teilnormen enthalten. Das €-Zeichen wurde mit **ISO 8859-15 (Latin-9)** eingeführt.

Weitere Zeichensätze. Es gibt viele weitere 8-Bit Zeichensätze, die teilweise proprietär von Computer- oder Softwareherstellern definiert worden sind. Einige Beispiele:

- Microsoft definiert eine Reihe von Windows- und DOS-Codepages. Unter Windows ist **CP1252** der Standard für Westeuropa; unter DOS war es **CP850**. Diese Zeichensätze sind, wie der unterschiedliche Name schon andeutet, nicht kompatibel zueinander.
- Apple hat für Mac-Computer vor Mac OS X z.B. den Zeichensatz **MacRoman** für westeuropäische Länder verwendet.
- IBM definiert unter dem Namen **EBCDIC** eine Reihe von 8-Bit Zeichsätzen, die hauptsächlich auf IBM Großrechnern eingesetzt werden. EBCDIC ist ein Beispiel für ein Code-Set, dass **nicht zu US-ASCII kompatibel** ist. Der Codebereich 0x00 bis 0x7F ist hier mit gänzlich anderen Zeichen belegt.

- Für russischen Text ist **KOI8-R** sehr weit verbreitet (bzw. **KOI8-U** für ukrainischen Text).

3.2.3. Multibyte-kodierte Zeichensätze

Bei einigen Sprachen, insbesondere den CJK-Sprachen (China, Japan, Korea), ist eine Beschränkung der Sprache auf einen 8-Bit Zeichensatz (mit max. 256 Zeichen) nicht möglich. Wenn ein 8-Bit Datentyp für die Kodierung zu Grunde gelegt wird, hat das zur Folge, dass einige Zeichen mit mehr als einem Byte kodiert werden müssen.
Zwei Beispiele für große Zeichensätze, die eine Multi-Byte-Kodierung verwenden, sind:

- **Big5** – Traditionelles Chinesisch
- **Shift-JIS** – Für Japanisch; von Microsoft entworfen.

Der Begriff Multi-Byte-Kodierung lässt nicht auf den Zeichensatz schließen. Es gibt **Multi-Byte-Encodings**, die Unicode (siehe z.B. UTF-8) kodieren und es gibt Multibyte-Encodings, die andere Zeichensätze (wie z.B. Shift-JIS) kodieren.

3.2.4. Begriffsfestlegung: Codepage-basierter String

In dieser Arbeit wird ein String, der in einer von den in diesem Abschnitt vorgestellten Kodierungen vorliegt, mit dem Begriff **Codepage-basierter String** (oder auch: **ANSI-basierter String**) bezeichnet. In dieser Arbeit beziehen sich diese Begriffe verallgemeinernd auf Strings, die in der Kodierungen eines beliebigen 8-Bit-Zeichensatzes (oder US-ASCII) verarbeitet werden. Sie bilden das Gegenstück zu dem Begriff **Unicode-basierter String**.

3.3. Der Unicode-Standard

> The Unicode Standard is the universal character encoding standard for written characters and text. It defines a consistent way of encoding multilingual text that enables the exchange of text data internationally and creates the foundation for global software. (Aus der Dokumentation der Unicode-Version 6.0.0 [Uni11], S.1)

Der **Unicode-Standard** definiert einen Zeichensatz, der alle für die Weltbevölkerung relevanten Textzeichen enthalten soll. Er wird vom Unicode-Konsortium[5] seit 1991 entwickelt und in regelmäßigen Zeitabständen in Form einer neuen Version erweitert. Dieser Standard stellt die Grundlage für die Internationalisierung von Software dar. Dies wird durch eine breite Unterstützung von Unicode durch Betriebssysteme, Programmiersprachen, Datenbanksysteme oder Internet-Browser gezeigt. Zu Beginn des Jahres 2012 liegt der Unicode-Standard in der Version 6.0[6] vor. In der aktuellen Version sind 109.242 Zeichen enthalten; in der ersten Version waren es noch (oder schon) 7.161 Zeichen.
Die ISO bezeichnet mit der Norm **ISO 10646** praktisch den gleichen Zeichensatz. In der Unicode-Dokumentation [Uni11] wird auf den Seiten S.1347 ff. der geschichtliche Zusammenhang zu ISO 10646 erläutert. In der Einleitung dieses Absatzes wird verdeutlicht, dass sich beide Organisationen, das Unicode-Konsortium und die ISO, der Synchronisation ihrer Standarde verschrieben haben. Es heißt dort:

> *The Unicode Consortium maintains a strong working relationship with ISO/IEC/JTC1/ SC2/WG2, the working group developing International Standard 10646. Today both organizations are firmly committed to maintaining the synchronization between the Unicode Standard and 10646.*

Neben dem Zeichensatz selbst definiert der Unicode-Standard eine Reihe von Kodierungen (im englischen wird der Begriff Encodings verwendet), die unter dem Begriff **UTF** (**Universal Transformation Format**) zusammengefasst werden (siehe Abschnitt 3.3.2) und eine Reihe von Normalformen und Algorithmen, die eine einheitlich Behandlung von Texten verschiedenster Sprachen ermöglichen sollen. Teilzeichensätze von Unicode, die einem bestimmten Schriftsystem zu Grunde liegen, werden in der Unicode Terminologie als **Skript** bezeichnet.

Die folgenden Abschnitte behandeln die Gliederung des Unicode-Zeichensatzes (Abschnitt 3.3.1), die möglichen Kodierungen von Unicode (Abschnitt 3.3.2) und geben einen Überblick über weitere Festlegungen im Unicode-Standard (Abschnitt 3.3.3).

3.3.1. Gliederung des Zeichensatzes

Unicode definiert insgesamt 1.114.112 Codepunkte. Diese teilen sich in 17 Bereiche zu je 65.536 Zeichen auf. Diese Bereiche werden in der Unicode-Terminologie **Planes** ge-

[5] `http://www.unicode.org` [Letzter Zugriff am 31.01.2012]
[6] `http://www.unicode.org/versions/Unicode6.0.0` [Letzter Zugriff am 31.01.2012]

nannt. Über Aufteilung und Zuordnung bestimmter Teilzeichensätze zu einzelnen Planes wird in[Uni11]auf den Seiten S.32ff ein Überblick gegeben. An dieser Stelle wird die wichtigste Plane, die Basic Multilingual Plane eingeführt und die übrigen Planes aufgelistet. Das Konzept der Planes ist vor allem für die weit verbreitete Kodierung UTF-16 (siehe Abschnitt 3.3.2.2) von Bedeutung.

Basic Multilingual Plane (BMP). Die erste der 17 Planes, **Plane 0**, wird als **Basic Multilingual Plane** (Abk.: **BMP**) bezeichnet. Die ersten 128 Codepoints entsprechen dem US-ASCII- Zeichensatz. Die ersten 256 Codepoints entsprechen dem ISO 8859-1-Zeichensatz. Hauptsächlich ist Plane 0 mit Zeichensätzen von Sprachen belegt, die aktuell in Gebrauch sind. Das schließt auch diejenigen Teile der CJK-Sprachen ein, die häufig in Gebrauch sind. Daneben sind auch wichtige weitere Satzzeichen, Symbole und Steuerzeichen enthalten. Weiterhin gibt es Codepoint-Bereiche für Surrogate-Pairs, die bei einer UTF-16-Kodierung (siehe 3.3.2.2) dazu verwendet werden, Zeichen außerhalb der BMP zu kodieren. Ein weiterer Bereich ist für die private Nutzung vorgesehen. In diesem Bereich werden zum Beispiel die Zeichen der klingonischen Sprache definiert[7]. Solche Definitionen sind allerdings kein Teil des Unicode-Standards und daher nicht offiziell. Die privaten Code-Bereiche können beliebig oft und von Jedermann definiert werden.

Überblick über alle Planes. Die folgende Auflistung gibt einen Überblick über die 17 Planes des Unicode-Zeichensatzes.

- Plane 0 – BasicMultilingualPlane.
- Plane 1 – Supplementary MultilingualPlane (SMP). Enthält Zeichen für historische oder weniger relevante aktuelle Sprachen. Außerdem werden viele musikalische und mathematische Symbole und Piktogramme (z.B. Mahjongg-Steine) definiert.
- Plane 2 – Supplementary Ideographic Plane (SIP). Enthält viele weitere Zeichen aus den CJK-Sprachen.
- Plane 3-13 – Reserviert.
- Plane 14 – Reserviert. Enthält bereits einige weitere Steuerzeichen wie z.B. Zeichen für die Markierung der verwendeten Sprache.

[7] `http://www.evertype.com/standards/csur/klingon.html` [Letzter Zugriff am 31.01.2012]

- Plane 15, 16 – Beide Planes sind vollständig für die private Benutzung vorgesehen.

3.3.2. UTF: Kodierungen für den Unicode-Zeichensatz

Der Unicode-Standard definiert die offiziellen Kodierungen des Zeichensatzes zweischichtig ([Uni11], S. 24ff). Zunächst wird der Begriff Byte-Order-Mark eingeführt, der bei der Beschreibung der Unicode-Kodierungen bekannt sein muss.

Definition: Byte-Order Mark (BOM). Unicode definiert zwei Codepoints, die dazu verwendet werden, die **Byte-Order** (d.h. Big-endian oder Little-endian) zu identifizieren, die bei einer UTF-16- oder einer UTF-32-Kodierung verwendet wird. Das Zeichen mit dem Codepoint `0xFEFF` (zero width no-break space) wird als erstes Zeichen in einen Stream oder eine Datei geschrieben. Handelt es sich bei der Byte-Order um Big-endian, wird das Zeichen unverändert als `0xFEFF` kodiert. Handelt es sich hingegen um eine Little-endian Byte-Order, wird das Zeichen als `0xFFFE` kodiert (dieser Codepoint ist explizit als ungültiges Zeichen definiert). Der Leser des Stream kann anhand des ersten gelesenen Codepoints entscheiden, mit welcher Byte-Order der zu lesende Text kodiert wurde.

Im Folgenden werden die beiden Kodierungs-Schichten von Unicode aufgelistet.

1. Die **Encoding-Form** gibt den grundlegenden Datentyp von Codeunits an. Die Encoding-Form entspricht also der **Bitbreite der Kodierung**. Im Unicode-Standard werden folgende Basistypen definiert:

   a) UTF-8 – Ein 8-Bit großer Datentyp, z.B. `char`

   b) UTF-16 – Ein 16-Bit großer Datentyp, z.B. `char16_t` oder `unsigned short`

   c) UTF-32 – Ein 32-Bit Datentyp, z.B. `char32_t` oder `unsigned int`

2. Das **Encoding-Scheme** spezifiziert zusätzlich die **Endian Order** und legt damit fest, wie die kodierte Darstellung eines Zeichens letztendlich aussieht. Dies kann entweder explizit durch die Angabe eines Suffixes geschehen, d.h. `LE` (für Little-Endian) oder ein `BE` (für Big-Endian), oder implizit durch die Verwendung einer BOM (Byte-Order Mark). Für UTF-8 ist diese Spezifikation nicht anwendbar, weil es sich um eine Kodierung mit 8-Bit (ein Byte) großen Codeunits handelt und daher die Byte-Order keine Rolle spielt. Allerdings kann auch für UTF-8 eine BOM

Encoding Scheme	Endian Order	BOM Allowed?
UTF-8	N/A	yes
UTF-16	LE xor BE	yes
UTF-16LE	Little-endian	no
UTF-16BE	Big-endian	no
UTF-32	LE xor BE	yes
UTF-32LE	Little-endian	no
UTF-32BE	Big-endian	no

Tabelle 3.2.: Unicode Encoding Schemes

angegeben werden, die immer als Byte-Folge `0xEF 0xBB 0xBF` kodiert wird. Das Encoding-Scheme stellt einen **Aspekt der Kodierungsfunktion** der Kodierung dar.

Tabelle 3.2, die inhaltlich aus [Uni11] entnommen ist, listet alle möglichen offiziell definierten Kodierungen für den Unicode-Zeichensatz auf.

Festlegung. Wird im Folgenden z.B. von einer UTF-32-Kodierung gesprochen, sind damit zusammenfassend alle drei möglichen Variante von UTF-32 gemeint, also: die explizite Spezifikation einer Big-endian- bzw. Little-endian-Kodierung sowie die implizite Spezifikation mittels BOM.

Die unterschiedlichen Ansätze für eine Kodierung unterscheiden sich maßgeblich in den Punkten Speicherplatzbedarf, Kompatibilität zu US-ASCII (bzw. ISO 8859-1) und Rechenaufwand zur Laufzeit. Die Folgenden Unterabschnitte sollen die Vor- und Nachteile der einzelnen Kodierungen feststellen.

3.3.2.1. UTF-32-Kodierungen

Die **UTF-32-Kodierungen** sind die einzigen **Fixed-Width-Encodings** für den Unicode-Zeichensatz. Alle Codepoints werden durch genau eine 32-Bit große Codeunit und der Identitätsfunktion als Kodierungsfunktion dargestellt. Das bedeutet für englisch- oder europäischsprachige Texte einen entscheidenden Nachteil: Es wird im Vergleich zu der Kodierung eines „klassischen“ 8-Bit Zeichensatz verschwenderisch mit Speicherplatz umgegangen. Mit UTF-32 belegen alle Zeichen 32-Bit Speicher, also viermal so viel Speicher, wie unter Verwendung eines ISO-Zeichensatzes benötigt würde.

Der Vorteil ist allerdings, dass so alle Codepoints mit einer fixen Länge kodiert werden.

Jeder Unicode Codepoint belegt genau ein Element in einem Array; nur das in diesem Fall der Datentyp des Arrays viermal so groß ist, wie bei einem 8-Bit-Zeichensatz. Das verleitet zur Annahme, dass UTF-32-kodierte Strings genauso gehandhabt werden können, wie es von 8-Bit Strings bekannt ist. In Abschnitt 3.3.3 wird gezeigt, warum diese Annahme falsch ist.

21-Bit würden zur Kodierung der 1.114.112 definierten Codepunkte übrigens schon ausreichen. Bei der Anwendung einer UTF-32-Kodierung enthält folglich jede Codeunit 11 irrelevante Bits, oder anders gesagt: mindesten ein unbenutztes Byte.

3.3.2.2. UTF-16-Kodierungen

Die **UTF-16-Kodierungen** können als 16-Bit-Kodierungen genau 2^{16} Codepoints adressieren. Somit können **alle Zeichen der BMP** mit genau einer Codeunit dargestellt werden.

> For the BMP, UTF-16 can effectively be treated as if it were a fixed-width encoding form. ([Uni11], S. 27.)

Das hat den Effekt, dass für die große Mehrheit aller verwendeten Zeichen die Hälfte des Speicherplatzes von UTF-32 ausreichend ist. Speicherplatztechnisch stellt UTF-16 also einen Kompromiss zwischen UTF-32 und einer 8-Bit Kodierung dar.

UTF-16 Surrogate-Pairs. Um Zeichen außerhalb der BMP zu kodieren, werden sogenannte **Surrogate-Pairs** ([Uni11] S. 144, 145) verwendet. Ein Surrogare-Pair besteht immer aus einen **High-Surrogate gefolgt von einem Low-Surrogate**. In Kombination identifizieren die beiden Surrogates genau einen Unicode-Codepoint, der außerhalb der BMP liegt. Den Surrogates sind ausreichend große (d.h. jeder Unicode-Codepoint kann adressiert werden) und disjunkte Codepoint-Bereiche in der *BMP* zugeteilt. Surrogates sind keine Zeichen (weder Steuerzeichen noch Schriftzeichen) des Zeichensatzes, sondern nur eine Bezeichnung für diejenigen Codepunkte, die in einem speziellen Bereich enthalten sind, der vollständig belegt ist und keine Zeichendefinitionen enthält.

Aus Sicht der Kodierungsfunktion ist die BMP in drei Bereiche aufgeteilt:

1. Ein nicht-zusammenhängender Bereich für Codepoints, die mit genau einer UTF-16-Codeunit dargestellt werden können.
2. Ein zusammenhängender Bereich der Codepoints für High-Surrogates enthält.

3. Ein zusammenhängender Bereich der Codepoints für Low-Surrogates enthält.

Durch die Surrogate-Pairs wird UTF-16 zu einem Variable-Width-Encoding: Ein Zeichen wird entweder durch eine oder durch zwei 16-Bit-Codeunits dargestellt. UTF-16 hat einen Nachteil gegenüber UTF-32 in der Handhabung, wenn Zeichen außerhalb der BMP verwendet werden: Bei der Bearbeitung einer Zeichenkette muss immer mit dem Auftreten eines Surrogate-Pairs gerechnet werden.

3.3.2.3. UTF-8

Die **UTF-8-Kodierung**[8] ist ein **Multi-Byte-Encoding**, das einen einzelnen Unicode-Codepoint mit bis zu vier Bytes darstellt. Dabei werden die 128 im ASCII-Zeichensatz definierten Zeichen jeweils mit einem Byte kodiert. Das hat den großen Vorteil, dass dadurch jeder ASCII-Text automatisch auch UTF-8 kodiert ist. Diese Eigenschaft wird auch als **ASCII-Transparenz** bezeichnet.
Alle weiteren Zeichen werden mit mindestens zwei und höchstens vier Bytes kodiert. Die Kodierungsvorschrit ([Uni11], S. 93-94) stellt sicher, dass sich ein kodiertes Byte immer in genau eine von drei Gruppen einordnen lässt:

- Kodiertes ASCII-Zeichen (Byte-Wert des Codewortes kleiner als `0x80`)
- Start-Element einer Multi-Byte-Sequenz (Byte-Wert des Codewortes >= `0xC0`)
- Sonstiges Element einer Multi-Byte-Sequenz (Byte-Wert des Codewortes >= `0x80` und <= `0xB7`)

Durch diese klare Trennung ist es leicht, einen Stream „mittendrin" aufzugreifen oder einzelne Bit-Fehler zu übergehen.
Wird eine ungültige Sequenz von UTF-8-Codeunits entdeckt, muss nur gewartet werden, bis ein Byte, dessen höchstes Bit den Wert `0` hat (der erste Fall) oder ein Byte, dessen zwei höchste Bits jeweils den Wert `1` haben (der zweite Fall) gelesen wird.
Ab diesem Zeitpunkt kann der Stream „normal" weitergelesen werden. Diese Eigenschaft von UTF-8 wird auch als **Selbst-Synchronisation** bezeichnet.
Bei vielen älteren Multi-Byte-Encodings ist diese Selbst-Synchronisation nicht gegeben.

UTF-16 bietet eine Selbst-Synchronisation, die auf dem selben Prinzip beruht: Die Trennung von Single-Codepoint-Characters und (High- bzw. Low-) Surrogates hat den selben Effekt bei einem unvollständigen oder fehlerhaften Lesevorgang.

---
[8] *http://tools.ietf.org/html/rfc3629*

Zusammengefasst bietet UTF-8 maximale ASCII-Kompatibilität auf Kosten einer rechenintensiveren[9] Kodierungsfunktion. Der Speicherplatzbedarf ist am geringsten, wenn viele ASCII-Zeichen verwendet werden. Werden viele „hohe“ Unicode Codepoints (z.B. Zeichen aus CJK-Sprachen) verwendet, kann UTF-16 ressourcenschonender sein. Im „worst case“ benötigt UTF-8 genau soviel Speicher wie UTF-32.

Tabelle 3.3 fasst die Vor- und Nachteile der einzelnen UTF-Kodierungen zusammen.

Encoding	Kompatibilität	Speicherplatzbedarf	Kodierungsaufwand
UTF-8	ASCII/8859-1	8-Bit bis 32-Bit	> UTF-16
UTF-16	-	16-Bit bis 32-Bit	> UTF-32
UTF-32	-	32-Bit	minimal

Tabelle 3.3.: Vor- und Nachteile des einzelnen *UTF*s

3.3.2.4. Weitere (teils inoffizielle) Encodings

UCS-2. **UCS-2** ist eine in ISO 10646 definiertes Kodierung für den Unicode-Zeichensatz, die – aus heutiger Sicht – die Unicode BMP so auffasst, dass jeder Codepoint genau einem Zeichen des Zeichensatzes entspricht. Diese Kodierung kennt also – im Gegensatz zu UTF-16 – keine Surrogate-Pairs und ist damit ein **Fixed-Width-Encoding**. Diese Kodierung wurde zu einer Zeit verwendet, als Unicode bzw. ISO 10646 jeweils noch weniger als 2^{16} Zeichen definierten. Zu dieser Zeit war UCS-2 dazu geeignet, alle definierten Zeichen zu kodieren. Daher trägt es sicherlich zu dem Heute noch verbreiteten Mythos bei, dass es sich bei Unicode um einen 16-Bit-Zeichensatz handelt.

CESU-8. **CESU-8**[10] ist eine **Variante von UTF-8**. Bei der Anwendung von CESU-8 wird der zu kodierende String zunächst in der UTF-16-Kodierung dargestellt. Die kodierte Darstellung enthält ggf. Surrogate-Pairs, die im folgenden Schritt allerdings als gültige (UCS-2-)Codepoints aufgefasst werden. Unter dieser Annahme wird der String erneut mit der UTF-8-Kodierung dargestellt. High- und Low-Surrogates werden bei CESU-8 als zwei getrennte Codepoints behandelt, und daher unterscheidet sich CESU-8 ganau dann von UTF-8, wenn Codepoints außerhalb der BMP kodiert werden.

[9] Im Vergleich zu anderen UTF-Kodierungen.
[10] http://www.unicode.org/reports/tr26/

UTF-7. UTF-7 ist ein inoffizielles 7-Bit-Encoding von Unicode das speziell für den Austausch von E-Mails konzipiert worden ist (z.B. ist die Textübertragung von SMTP auf 7-Bit Codeunits beschränkt). Es wird in RFC2152[11](was ebenfalls keine offizieller Standard ist) beschrieben.

UTF-EBCDIC. UTF-EBCDIC[12] ist ein Unicode-Encoding das UTF-8 sehr ähnlich ist. Es ist ebenfalls ein Multi-Byte-Encoding, nur dass es anstelle der ASCII-Transparenz von UTF-8 EBCDIC-Transparenz aufweist.

3.3.3. Weitere Festlegungen im Unicode-Standard

Der Unicode-Standard definiert neben dem Zeichensatz und möglichen Kodierung weitere Standard fest. Im Folgenden werden einige für diese Arbeit wichtigen Teile des Unicode-Standards vorgestellt.

Precomposition, Decomposition und Combining Characters

Im Unicode-Standard werden **Combining Characters** definiert. Dabei handelt es sich um diakritische Zeichen, die nur zusammen mit einem **Base Character** ein darstellbares Zeichen ergeben. Ein Combining Character definiert neben der Gestalt der Glyphe die relative Stelle, an welcher der Base Character dargestellt werden muss. In der Unicode-Dokumentation [Uni11] wird dieses Konzept auf den Seiten 41-46 eingeführt.

Ein einfaches **Beispiel** dafür sind **deutsche Umlaute**: In diesem Beispiel stellt z.B. das Zeichen `A` (mit dem Codepunkt `0x0041`) den Base Character dar und das aus den zwei „Umlautpunkten" bestehende diakritische Zeichen ¨ den Combining Character (mit dem Codepunkt `0x0308`). Die Codepointfolge *`0x0041 0x0308`* stellt das Zeichen `Ä` dar. Dieses Zeichen ist allerdings auch als selbstständiges Zeichen definiert (mit dem Codepunkt `0x00C4`). Diese kompakte Kodierungsform eines Zeichens wird **Precomposed Form** genannt, die Kodierungform als Kombination aus Base- und Combining Character wird **Decomposed Form** genannt.

[11] http://tools.ietf.org/html/rfc2152
[12] http://www.unicode.org/reports/tr16/

Unicode-Normalisierungsformen

Im Unicode-Standard werden **vier Normalisierungsformen für Strings** und die Algorithmen, mit denen Strings in eine normalisierte Form überführt werden können, definiert. Die Normalisierungsformen werden in einem offiziellen Anhangsdokument ([DW12a]) des Unicode-Standards (Verweise auf den Anhang in [Uni11] auf S. 146, 147) beschrieben. Im wesentlichen erfüllen die Normalisierungformen den Zweck, alle Zeichen eines Strings in die Precomposed Form (Normalization Form C und KC) bzw. die Decomposed Form (Normalization Form D und KD) zu überführen und in beiden Fällen eine einheitliche Reihenfolge der Combining Characters sicherzustellen[13]. Durch Normalisierungsformen lässt sich die Überprüfung auf Äquivalenz zweier Zeichen vereinfachen, was eine Grundlage für den Collation-Algorithmus (s.u.) darstellt.

Dabei stellen die Formen C und D strengere Bedingung an die Äquivalenz: Die Ausgabedarstellung von normalisierten und unnormalisierten Strings muss identisch sein. D.h. dass z.B. die Ligatur Æ in der normalisierten Form unverändert als Ligatur Æ beibehalten wird, oder dass eine hochgestellte ^{5} in der normalisierten Form eine hochgestellte ^{5} bleibt. Die Formen KC und KD erforden nicht, dass die Ausgabedarstellung identisch ist, sondern nur, dass die abstrakten Zeichen übereinstimmten. So wird bei der Normalisierung aus der Ligatur Æ die Zeichenfolge `AE` und aus einer hochgestellten ^{5} eine „normale“ `5`.

Der Unicode-Collation-Algorithmus

Der Unicode-Standard definiert in einem Anhangdokument ([DW12b]) den **Unicode Collation Algorithm** (**UCA**). Der UCA legt durch **Collation Element Tables** (**CET**) eine regions- bzw. sprachabhängige kontextsensitive Gewichtung von Zeichen fest, die der lexikographischen Anordnung von Zeichen dieser Region bzw. Sprache entspricht. Kurz: Der Collation-Algorithmus legt fest, wie String sortiert werden müssen.

Beim binären Vergleich der Strings `Hällo`, `Hallo` und `Hello` ergibt sich die Anordnung `Hallo < Hello < Hällo`. Der Grund hierfür ist, dass dem Zeichen `ä` ein Codepoint zugeordnet ist, dessen Wert größer ist als der Wert desjenigen Codepoints, der dem Zeichen `e` zugeordnet ist. Durch Anwendung des UAC mit dem passenden CET lassen sich verschiedene lexikographisch korrekte Anordnung realisieren, z.B. mit dem

[13] Auch Strings in der Normalisierungsform C können Combining Characters enthalten, und zwar dann, wenn keine Precomposed Form für die gegebene Kombination aus Base Character und Combining Character existiert.

CET `GermanDictionary` die Anordnung `Hallo < Hällo < Hello`, mit dem CET `GermanTelephone` die Anordnung `Hällo < Hallo < Hello` oder mit dem CAT `SwedishDictionary` die Anordnung `Hallo < Hello < Hällo`[14].

3.4. String-Längen-Problematik

Dieser Abschnitt stellt einen typischen 8-Bit-Zeichensatz[15], den verschiedenen Kodierungen des Unicode-Zeichensatzes gegenüber. In den folgenden Abschnitten werden Annahmen über Strings, die für einen typischen 8-Bit-Zeichensatz gültig sind, aufgeführt und aus der Sicht der verschiedenen Unicode-Kodierungen bewertet. Dass diese Annahmen für Unicode-Strings i.Allg. nicht gültig sind, wird mit dem Begriff **String-Längen-Problematik** bezeichnet. Dieses Problem bezieht sich nicht allein auf die Berechnung der Länge von Strings, sondern hat Auswirkungen auf die Speicherverwaltung und die korrekte Funktionsweise von String-Funktionen aus der C- bzw. C++-Standardbibliothek, die im Regelfall nur für Strings in einem Fixed-Width-Encoding sichergestellt ist. Tabelle 3.4 fasst diese Gegenüberstellung zusammen.

	ISO 8859-xx	UTF-8	UTF-16	UTF-32
3.4.1: $\Vert \mathit{Character} \Vert \sim \Vert \mathit{Codepoint} \Vert$	•	×	×	×
3.4.2: $\Vert \mathit{Codeunit} \Vert \sim \Vert \mathit{Codepoint} \Vert$	•	×	×	•

Tabelle 3.4.: Vergleich unterschiedlicher Kodierungen

3.4.1. Zusammenhang zwischen der Länge eines Strings und der Anzahl der Schriftzeichen

Für Strings, die in einer ISO-Kodierung dargestellt werden, gilt die Annahme, dass jedes Zeichen ein Steuerzeichen oder ein Schriftzeichen aus der Menge Y ist. Der Grund dafür liegt darin, dass die ISO-Familie von Zeichsätzen **keine diakritischen Zeichen** enthält. Um z.B. ein `ä` zu erkennen reicht es in allen Fällen aus, einen einzelnen Codepoint zu betrachten.

Für alle UTF-Encodings ist diese Annahme falsch. Ein `ä` kann hier entweder als eigenständiges Schriftzeichen oder als zusammengesetztes Schriftzeichen vorliegen. Damit

[14] Im schwedischen wird das Zeichen `ä` nach dem Zeichen `z` sortiert. Diese Information ist aus [DW12a] entnommen worden.
[15] Z.B. einen beliebigen Zeichensatz aus der ISO-8859-Familie.

diese Annahme für Unicode-Strings teilweise übernommen werden kann, muss der String in eine der Normalformen C und KC überführt werden. Dadurch wird allerdings nicht sichergestellt, dass der String keine Combining Characters – und damit zusammengesetzte Schriftzeichen – enthält. Sobald Schriftzeichen zu erwarten sind, die im Unicode-Zeichensatz durch mehr als einen Codepoint als zusammengesetzte Schriftzeichen dargestellt werden können, muss auf Unicode-Algorithmen zur weiteren Verarbeitung des Strings (z.B. Vergleich oder Längenberechnung) zurückgegriffen werden.

Die folgende Annahme muss auch dann betrachtet werden, wenn alle Codepoints des Strings bekannt sind und davon ausgegangen wird, dass der String keine zusammengesetzten Schriftzeichen enthält.

3.4.2. Zusammenhang zwischen der Länge eines Strings und der Speichergröße

Für Strings, die in einer ISO-Kodierung dargestellt werden, gilt die Annahme, dass die **Speichergröße proportional zur Länge des Strings** ist: Ein Zeichen entspricht in diesem Fall immer genau einem Byte.

Für die UTF-Kodierungen UTF-8 und UTF-16 ist diese Annahme falsch, da es sich bei beiden Kodierungen um ein Variable-Width-Encoding handelt. Die Länge eines String kann ohne algorithmische Messung nur grob nach oben und unten abgeschätzt werden: Die Länge L eines Strings mit der Speichergröße S liegt für UTF-8-kodierte Strings im Intervall $L\epsilon[\lceil \frac{S}{4} \rceil ; S]$ (jeder Codepoint belegt höchstens vier und mindestens eine Codeunit), für UTF-16-kodierte Strings im Intervall $L\epsilon[\lceil \frac{S}{2} \rceil ; S]$ (jeder Codepoint belegt höchstens zwei und mindestens eine Codeunit). Gelten jedoch für die Zeichen eines Strings bestimmte Einschränkungen, kann die Annahme aufrecht erhalten werden:

- Enthält ein UTF-8-kodierter String nur Zeichen aus dem ASCII-Zeichensatz, ist die Länge des Strings identisch zu der Speichergröße in Bytes. In diesem Fall ist der String wegen der ASCII-Transparenz nicht von einem ASCII- oder ISO-kodierten String zu unterscheiden.
- Enthält ein UTF-16-kodierter String nur Zeichen aus der BMP, entspricht die Länge des Strings genau der Hälfte der Speichergröße in Bytes. In diesem Fall ist die UTF-16-Kodierung identisch zur UCS-2-Kodierung.

Für UTF-32 gilt die Annahme der Proportionalität zwischen Speichergröße und Stringlänge, da es sich bei UTF-32 um ein Fixed-Width-Encoding handelt. Für die Län-

ge L eines Strings mit der Speichergröße S gilt $L = \frac{S}{4}$ weil jeder UTF-32 Codepoint durch genau eine 4-Byte-große Codeunit dargestellt wird.

3.5. Zusammenfassung

Die Begriffe Zeichen, Zeichensatz, Kodierung und String wurden definiert und in einen Zusammenhang gestellt. Darauf aufbauend wurden einige „klassische" Zeichensätze vorgestellt, von denen insbesondere die Zeichensätze der ISO-8859-Familie als Ausgangspunkt für diese Arbeit von Bedeutung sind. Unicode wurde als internationaler Standard eingeführt, der in erster Linie einen Zeichensatz definiert, der alle für die Weltbevölkerung relevanten Schriftzeichen enthält. Der Unicode-Zeichensatz entscheidet sich vom Umfang her deutlich – etwa in der Größenordnung 10^3 – von den anderen vorgestellten Zeichensätzen. Es wurde verdeutlicht, dass dieser Unterschied Auswirkungen auf grundlegende Annahmen über die Handhabung von ISO-8859-kodierten Strings hat. Diese Auswirkungen wurden unter dem Begriff String-Längen-Problematik zusammengefasst.

An dieses Kapitel schließt sich thematisch die Einführung von *Microsoft*-spezifischen Grundlagen mit einem besonderen Blick auf Strings und Unicode an (Kapitel 5). Zunächst wird jedoch das Dokumenten-Management-System *PROXESS* eingeführt.

4. Einführung in *PROXESS*

PROXESS ist ein **Dokumenten-Management-System** (**DMS**), das von der *Akzentum GmbH*[1] gepflegt wird. Als DMS wird ein System bezeichnet, das elektronische Dokumente datenbankgestützt und unabhängig von herkömmlichen hierarchischen Dateisystem verwaltet. Elementare Funktionen bilden der Zugriff, die Verwaltung und die Darstellung dieser Dokumente (frei nach [Kam99], S. 37-39).

PROXESS wird **ausschließlich für das Betriebssystem *Microsoft Windows*** entwickelt. Daher werden im folgenden Kapitel (Kapitel 5) die API von *Microsoft Windows* und *Microsoft*-spezifische Technologien zur Interprozesskommunikation eingeführt, die von *PROXESS* verwendet werden.

Den Kern von *PROXESS* bildet ein mehrschichtiges Serversystem (Produktname: ***PROXESS Server***), das die Kernfunktionalität des DMS implementiert. Weiterhin gehören einer Reihe von Client-Programmen zu *PROXESS*, die z.B. graphische Benutzerschnittstellen anbieten oder weitere DMS-Funktionalität realisieren. In dieser Arbeit wird das Serverprogramm *DatabaseManager*, ein Teil von *PROXESS Server* (Abschnitt 4.3), zur Anwendung des Umstellungsprozesses (Kapitel 7) verwendet.

Abbildung 4.1 gibt einen Überblick über den **Aufbau der Client-/Server-Architektur** von *PROXESS*. Das Subsystem *PROXESS Server* hat wiederum einen schichtenorientierten Aufbau, der an dieser Stelle zu Gunsten der Übersichtlichkeit vernachlässigt wird. Abschnitt 4.3 führt den Aufbau von *PROXESS Server* ein.

Die primäre Schnittstelle von *PROXESS*, die angebotene Schnittstelle des *DocumentManager*, dem Hauptprogramm von *PROXESS Server*, **ist eine RPC-Schnittstelle**. RPC wird in Kapitel 5 eingeführt. Diese Schnittstelle wird – mit Hilfe von Adaptern – zusätzlich für viele andere Technologien (*z.B.* .NET, COM+ und SOAP) und auf verschiedenen Detailstufen angeboten, was anderen Herstellern, sogenannten Lösungspartnern, beim Erstellen von Client-Software für *PROXESS* möglichst viel Flexibilität bieten soll.

[1] https://www.akzentum.de

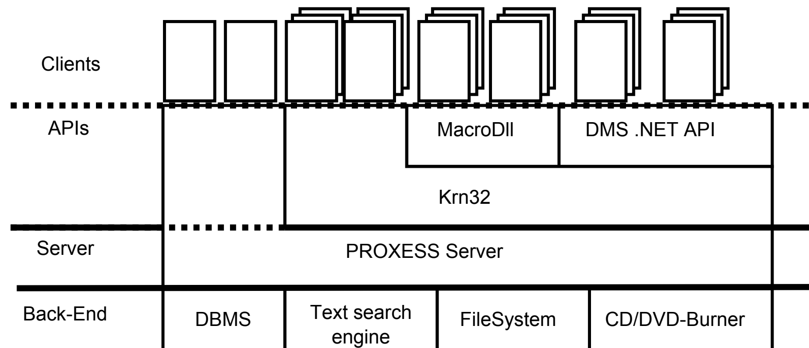

Abbildung 4.1.: Client-/Server-Schichtenmodell

Im Folgenden wird das Datenmodell, das zugehörige Datenschema und Metaschema von *PROXESS* eingeführt (Abschnitt 4.2). Darauf aufbauend werden die von *PROXESS* angebotenen Services eingeführt (Abschnitt 4.2). Im Anschluss wird der Aufbau des Subsystems *PROXESS Server* erläutert (Abschnitt 4.3). Bei diesen Einführungen wird die Terminologie von *PROXESS* verwendet. In anderen Dokumenten-Management-System können bestimmte Daten, Services oder Eigenschaften andere Namen haben.

Zu dem Text werden einige Abbildungen zur Unterstützung der Verständlichkeit gezeigt. Diese Abbildungen sind UML-Diagramme[2] und geben die Systemarchitektur aus der Logical View oder der Developement View nach Kruchtens „4+1“ Schichtenmodell [Kru95] wieder. Die Diagramme haben nicht das Ziel, Details der Implementierung aufzudecken, sondern sollen lediglich einen guten Überblick über die Systemstruktur geben.

4.1. Datenmodell

Dieser Abschnitt führt das von *PROXESS* verwendete Datenmodell und das zugehörige Datenschema ein.

Zuerst werden die Elemente des Datenmodells eingeführt (Abschnitt 4.1.1). Das Datenmodell besteht im wesentlichen aus Dokumenten und Dateien, die die grundlegenden Datenelemente von *PROXESS* darstellen.

Im Anschluss daran wird das Datenschema vorgestellt, das aus Benutzerdatenbanken, Dokumenttypen und Dateitypen besteht (Abschnitt 4.1.2). Das Datenschema wird bei

[2]Die *UML* (*Unified Modeling Language*) ist eine grafische Modellierungssprache zur Beschreibung von Software-Systemen. Die Sprache wird in zwei Dokumenten spezifiziert, der *UML Superstructure* [Obj11b] und der *UML Infrastructure* [Obj11a].

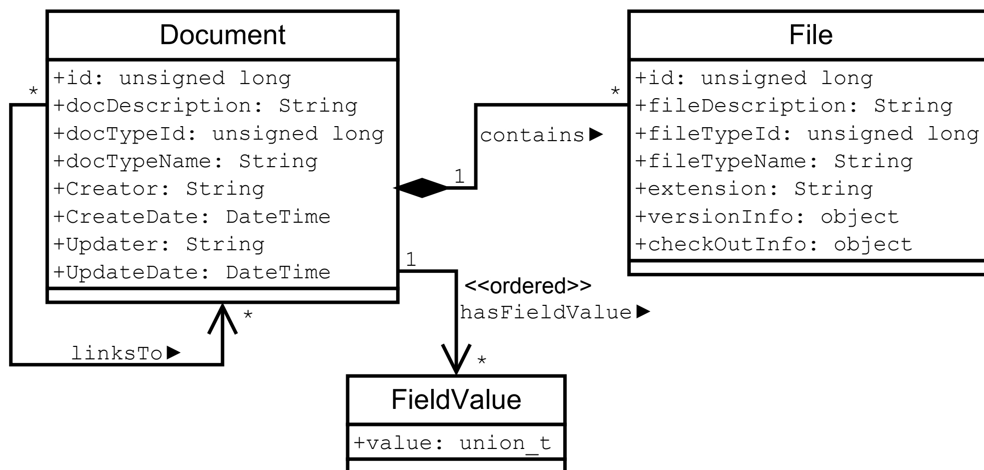

Abbildung 4.2.: Beziehungen zwischen Dokumenten und Dateien

der Einrichtung von *PROXESS* individuell konfiguriert. Eine typische Konfiguration des Datenschemas wird an einem Beispiel verdeutlicht (Abschnitt 4.1.3).

4.1.1. *Dokumente* und *Dateien*

Ein **Dokument** (**`Document`**) ist die zentrale Dateneinheit in *PROXESS.* Es ist ein mit Merkmalsfeldern und Verweisen zu anderen Dokumenten versehener Container für Dateien. In der *PROXESS* Terminologie entspricht eine **Datei** (**`File`**) dem in der Definition von DMS verwendeten Begriff eines elektronischen Dokuments. Eine Datei in *PROXESS* ist eine physische Datei im Dateisystem zusammen mit einer Menge von weiteren Eigenschaften, die für die Verwendung in *PROXESS* relevant sind. Eine Datei ist in der Praxis meist ein gescanntes Faksimile, eine Datei aus einer Office-Suite, eine E-Mail oder eine COLD-Datei (Computer Output on Laser Disk, siehe z.B. [3]). Allerdings macht *PROXESS* hier keinerlei Einschränkungen: Es kann theoretisch jede beliebige Datei aus dem Dateisystem archiviert werden, z.B. auch CD-Images, Programmdateien oder Swap-Files.

Abbildung 4.2 verdeutlicht den Zusammenhang zwischen Dokumenten (`Document`), Dateien (`File`) und den Merkmalsfeldern eines Dokumentes (`FieldValues`). Es wird deutlich, dass es durchaus Dokumente ohne Dateien, aber keine Dateien ohne Dokument geben kann.

[3] `http://de.wikipedia.org/wiki/Computer_Output_on_Laserdisk`

4.1.2. Datenbanken, Dokumenttypen und Dateitypen

Abbildung 4.3 zeigt das vereinfachte **Datenschema** von *PROXESS*. Das zentrale Element des Schemas ist die Benutzerdatenbank. Eine **Benutzerdatenbank** (**`UserDatabase`**) ist ein Kontainer für Dokumente, Dateien von Dokumenten und Typinformationen für Dateien und Dokumente. In der *PROXESS*-Terminologie heißen Benutzerdatenbanken einfach nur Datenbanken. In dieser Arbeit wird konsequent der Begriff Benutzerdatenbank verwendet, um eine klare Unterscheidung zu den relationalen Datenbanken zu treffen, die im Hintergrund als Datenspeicher verwendet werden.

Bei der Konfiguration von *PROXESS* können beliebig viele Benutzerdatenbanken definiert werden, die jeweils beliebig viele Dokument- und Dateitypen (s.u.) enhalten. Im folgenden Abschnitt 4.1.3 wird beispielhaft eine Konfiguration dieses Schemas vorgestellt.

Für jede Benutzerdatenbank wird eine Menge von **Dokumenttypen** (**`DocumentType`**) und **Dateitypen** (**`FileType`**) definiert. Dokumenttypen bzw. Dateitypen dienen zur Kategorisierung von Dokumenten bzw. Dateien innerhalb einer Benutzerdatenbank. Dokumente sind immer einem Dokumenttyp zugeordnet und Dateien sind immer einem Dateityp zugeordnet. Da Dokumenttypen (bzw. Dateitypen) nicht datenbankübergreifend definiert sind, stellt die Benutzerdatenbank selbst das erste Ordnungskriterium für Dokumente (bzw. Dateien) dar.

Dokumente dürfen nicht als Instanzen von Dokumenttypen verstanden werden: Der Dokumenttyp ist lediglich ein fest vorgegebenes Merkmalsfeld eines Dokuments.

In jeder Benutzerdatenbank werden alle verfügbaren **Merkmalsfelder** (**`Field`**) für Dokumente deklariert. Es gibt es eine Reihe von fest vorgegebenen Merkmalsfeldern, z.B. eine Dokumentenbeschreibung, der Benutzername des Erstellers, das Erstelldatum oder der Name und die Id des Dokumenttyps. Diese fest vorgegebenen Felder werden **Kernfelder** genannt. In Abbildung 4.2 werden die Kernfelder als Attribute der Klasse `Document` dargestellt. Im letzten Abschnitt wurde der Begriff Merkmalsfeld im Zusammenhang mit Dokumenten verwendet: In diesem Zusammenhang ist der Begriff Merkmalsfeld als konkrete Wertzuweisung an eines der in der Benutzerdatenbank definierten Felder zu verstehen.

Jeder Dokumenttyp kann beliebig viele Merkmalsfelder (außer Kernfelder) überschreiben. Zum Beispiel kann durch Setzen des `roleName`-Attributs ein für den Dokumenttyp spezifischer Anzeigename festgelegt werden (das `defaultRoleName`-Attribut des

Datenbankfeldes wird überschreiben), die Sichtbarkeit des Merkmalsfeldes geändert werden (durch Überschreiben des `visible`-Attributs) oder die Pflichteingabe für einen Feldwert (durch Überschreiben des `mandatory`-Attributs) geändert werden. Überschriebene Merkmalsfelder werden **Dokumenttypfelder** genannt. Abbildung 4.4 stellt Dokumenttypen, Dokumenttypfelder und Datenbankfelder in einen Zusammenhang.

In Abschnitt 4.3.2 wird skizziert, wie das Schema der relationalen Datenbanken aufgebaut ist, und wie es mit dem hier beschriebenen Datenmodell von *PROXESS* zusammenhängt.

4.1.3. Beispielschema: Datenbank „Personal"

Abbildung 4.5 stellt eine vereinfachte typische Konfiguration des Datenschemas von *PROXESS* dar. In der Benutzerdatenbank `Personal` sind die Felder `Name` und `Datum` und die Dokumenttypen `Vetrag`, `Mahnung` und `Gehaltsabrechnung` enthalten. Der Dokumenttyp `Vertrag` überschreibt das Datenbankdeld `Datum` und legt damit einen neuen Anzeigenamen für das Feld fest. In dem Beispiel fehlen Dateitypen, die typischerweise an „echte" Dateitypen, wie z.B. `.pdf`, `.docx` oder `.txt` angelehnt werden. Gibt es drei verschiedene Datenbanken muss z.B. der Dateityp `.pdf` bei Bedarf auch dreimal definiert werden.

Weitere typische Datenbanknamen für Anwender von *PROXESS* sind zum Beispiel: `WAWI` (Warenwirtschaft) oder `FIBU` (Finanzbuchhaltung).

4.2. Services

Im Folgenden werden die wichtigsten Services, die *PROXESS* anbietet, erläutert. Dabei werden die Services in zwei Kategorien eingeteilt: Services die vom Kernsystem *PROXESS Server* angeboten werden und Services die von Client-Software angeboten werden. Es wird nur ein Querschnitt durch die wirklich relevanten Services gezogen.

4.2.1. Im Serversystem implementierte Services

Im Serversystem von *PROXESS* (Produktbezeichnung: *PROXESS Server,* Einführung in Abschnitt 4.3) werden grundlegende Kernfunktionen des DMS realisiert, die allesamt über ein einzelnes Interface (Fassade) zugänglich gemacht werden.

Zugriff, Verwaltung und Darstellung von *Dokumenten*. *PROXESS* bietet Funktionen für das Anlegen, das Manipulieren und das Löschen von Dokumenten. Das Manipulieren von Dokumenten umfasst Änderungen der Merkmalsfelder und das Hinzufügen, Ändern und Löschen von Dateien. Die Benutzerdatenbank, in der das Dokument angelegt wird, der Dokumenttyp und die Merkmalsfelder stellen bei sachgemäßer Verwendung einen ordentlich verwalteten Dokumentenbestand sicher.

Die programmatische Darstellung eines Dokumentes ist ein Handle das als Parameter an Dokumentfunktionen des Interface übergeben wird, mit denen gezielt einzelne Informationen über das Dokument und seine Dateien abgerufen werden können. Handles im Zusammenhang mit *PROXESS* werden in Abschnitt 4.3.3 diskutiert.

Der Zugriff auf Dokumente wird durch mehrere Abfrageschnittstellen realisiert. Generell kann ein beliebiger Baum von Suchkriterien aufgebaut werden, bei dem innere Knoten eine logische `UND`- bzw. `ODER`-Verknüpfung darstellen und die Blätter einzelne Suchbedingungen. Suchbedingungen können Wertebeziehungen für ein Merkmalsfeld ausdrücken (z.B. `Name = 'Hugo Testmann'` oder `Datum >= 01.01.2012`) oder Volltextabfragen an den Dateibestand oder die Menge aller Dokumentenmerkmale sein.

Verschlüsselung von Merkmalsfeldern und Dateien in „Hochsicherheitsdatenbanken“. *PROXESS* bietet AES-Verschlüsselung von Merkmalsfeldern und Dateien von Dokumenten für speziell lizensierte Benutzerdatenbanken an.

Versionierung von *Dateien*. Wird eine im System verwaltete Datei überschrieben, wird dabei automatisch eine neue Version dieser Datei erstellt. Die Vorgängerversion bleibt im System erhalten und kann weiterhin abgerufen werden. Zusätzlich wird das Ändern mit Hilfe einer „Lock-Modify-Write“-Disziplin unterstützt, wobei bei einem Write automatisch eine neue Version der Datei entsteht.

Langfristige Archivierung von *Dateien;* elektronische Archivierung. Dateien können auf optische Medien ausgelagert werden. Das System führt Buch darüber, ob eine Datei ausgelagert ist und auf welchem Medium sie gespeichert ist. Einige JukeBox-Typen werden unterstützt, um Dateien auf verschiedenen Medien zeitnah bereitstellen zu können.

Benutzer, Gruppen und Zugriffsrechte. *PROXESS* unterstützt die Verwaltung von Zugriffsberechtigungen in From von ACLs (Access Control Lists). Rechte werden Benutzern oder Gruppen erteilt oder entzogen. Zugriffsrechte können für Benutzerdatenbanken, Dokumenttypen und Dateien erteilt/entzogen werden. Derzeit erfolgt die Authentifizierung von Benutzern über Passwörter oder Smartcards. Eine Integration von *Microsoft Active Directory*[4] in die *PROXESS* Benutzerauthentifizierung befindet sich derzeit in Entwicklung.

4.2.2. Von Clients implementierte Services

Einige Services, die von *PROXESS* bereitgestellt werden, werden von Programmen angeboten, die selbst Clients des Kernsystems *PROXESS Server* sind. Trotzdem sind diese Services ebenfalls ein integraler Bestandteil des Dokumenten-Management-Systems.

Scannen von Faksimile-Dokumenten mit Barcode-Erkennung. ***PROXESS Scan Link*** ist ein Client-Programm von *PROXESS* mit dem Papierdokumente gescannt und in *PROXESS* als Dokument archiviert werden. Dabei können Regeln festgelegt werden, wie die Merkmalsfelder des *Dokuments* zu belegen sind. Das Papierdokument kann vor dem Scannen mit einem Barcode-Sticker beklebt werden. Der Barcode wird von *Scan Link* erkannt und dient zur Identifikation des Dokumentes im System[5].

Direktes Archivieren aus Microsoft Office und dem Windows Explorer. ***PROXESS Office Link*** bzw. ***PROXESS Explorer Link*** integrieren die Fähigkeit, Office-Dokumente direkt aus Microsoft Office oder Dateien aus dem Dateisystem direkt aus dem Windows Explorer in *PROXESS* zu archivieren. Über eine Benutzermaske können vor dem Archivieren die Merkmalsfelder des Dokumentes spezifiziert werden.

Direktes Archivieren von PDF-Ausdrucken. ***PROXESS Printer Link*** bietet einen Druckertreiber an, der auf *PDFCreator*[6] basiert, um aus einer beliebigen Anwendung, die Ausdrucke unterstützt, eine PDF-Datei zu drucken und diese direkt in *PROXESS* zu archivieren. Die Merkmalsfelder für das resultierende Dokument können vor dem Archivieren über eine Benutzermaske eingegeben werden.

[4] `http://www.microsoft.com/en-us/server-cloud/windows-server/active-directory-overview.aspx` [19.03.2012]
[5] Der Barcode wird als Merkmalsfeld des *Dokuments* gespeichert.
[6] `http://sourceforge.net/news/?group_id=57796`[16.03.2012]

4.3. *PROXESS* Server

In diesem Abschnitt wird das Subsystem *PROXESS Server* näher betrachtet. Dieser Teil des Systems besteht aus drei Serverprogrammen, die zusammen die Kernfunktionalität des Dokumenten-Management-Systems realisieren, und aus einer Reihe von Konfigurationsprogrammen, die zur Einbettung des Systems in die Zielumgebung und zur Konfiguration des Datenschemas verwendet werden. Dieser Teil von *PROXESS* taucht in einer *PROXESS*-Installation genau einmal auf und dient als zentrale Logik- und Verwaltungseinheit des Systems.

Im Folgenden wird ein Überblick über die einzelnen Komponenten gegeben (Abschnitt 4.3.1), die Realisierung des Datenschemas mit relationalen Datenbanken besprochen (Abschnitt 4.3.2), das Konzept von Session- und Object-Handles erläutert (Abschnitt 4.3.3) und es wird das Mengengerüst der Serverprogramme von *PROXESS Server* aufgeführt (Abschnitt 4.3.4).

4.3.1. Server-Komponenten

Abbildung 4.6 zeigt ein Komponentendiagramm des Subsystems *PROXESS Server.* Die angebotenen Schnittstellen der drei Server *DocumentManager*, *StorageManager* und *DatabaseManager* sind RPC-Schnittstellen (Remote Procedure Call). RPC wird in Kapitel 5 eingeführt.

Im Folgenden werden die Aufgaben der einzelnen Komponenten erläutert.

4.3.1.1. Serverprogramme

Der ***DocumentManger*** ist das zentrale Serverprogramm von *PROXESS*. Er stellt für Client-Programme eine einzige Schnittstelle bereit, über welche die gesamte Funktionalität von *PROXESS Server* angeboten wird (Fassade). Alle Clients von *PROXESS* (oder genauer: *PROXESS Server*) sind – mit einer einzigen Ausnahme – Clients des *Document-Managers*. Der *DocumentManager* realisiert die gesamte Business-Logik des Systems: Erst hier werden Datenbank-Einträge zu Dokumenten und herkömmliche Dateien aus dem Dateisystem zu Dateien als Teil von Dokumenten. Außerdem ist der *DocumentManager* alleinig für die Umsetzung der Zugriffskontrolle zuständig, verwaltet demnach die Login-Informationen und stellt sicher, dass ein Benutzer für eine Aktion auch über die entsprechenden Rechte verfügt.

Der ***StorageManager*** ist für die Verwaltung von Dateien im herkömmlichen Sinne verantwortlich. Er identifiziert, speichert und ruft Dateien ab und realisiert im Wesentlichen die elektronische Archivierung durch Auslagerung von Datenbeständen in Form von Images auf optische Medien.

Der ***DatabaseManager*** ist für die Kommunikation mit dem Datenbank-Management-System (DBMS) und die Volltext-Suche und -Indexierung verantwortlich. Der *DatabaseManager* stellt alle Nutzdaten (d.h. Dokumente und Dokument-Datei-Mappings) und sämtliche Konfigurationsdaten (Dokumentypen und Dateitypen, Benutzer und Gruppen etc.) für den *DocumentManger* und den *StorageManager* in einer spezifischen Tabellendarstellung bereit. Diese Tabellendarstellung ist die systemweit gültige Form der Darstellung von Daten, die auch von Client-Programmen verwendet wird.

Der *DatabaseManager* existiert in drei verschiedenen Ausführungen, von denen jede eine Implementierung für eines der drei unterstützten *Datenbank-Systeme* ist: *Microsoft SQL Server*[7], *Intersystems Caché*[8] und *Oracle Database*[9].

4.3.1.2. Konfigurationsprogramme

Zur initialen Konfiguration von *PROXESS Server* wird das Programm ***PROXESS Registy Setup*** eingesetzt. Mit diesem Programm werden die Verbindungsdaten für die relationalen Datenbank, die Volltext-Datenbank und die Serverprogramme untereinander konfiguriert. Weiterhin können Logging-Einstellungen vorgenommen werden und bestimmte Grenzwerte wie z.B. Timeouts und maximale Trefferlistenlänge festgelegt werden. Dieses Programm läuft unabhängig von *PROXESS*, ist also im Gegensatz zu den anderen Konfigurationsprogrammen kein Client von *PROXESS.*

Zur Benutzer- und Gruppenverwaltung wird das Programm ***PROXESS Administrator Console*** eingesetzt. Benutzerdatenbanken, Merkmalsfelder und Dokument- und Dateitypen werden mit dem Programm ***PROXESS Administrator*** festgelegt. Für das initiale Aufbauen des Volltext-Index für einen bereits bestehenden Datenbestand wird das Tool ***PROXESS Fulltext Repair*** eingesetzt. Diese Programme sind Clients des *DocumentManagers*, wie auch alle anderen Client-Programme, die während des normalen Betriebs eingesetzt werden.

Der ***PROXESS StorageManager Explorer (SMX)*** stellt eine Besonderheit dar: Er ist das

[7] www.microsoft.com/server-sql
[8] http://www.intersystems.de/cache/index.html
[9] http://www.oracle.com/us/products/database/index.html?ssSourceSiteId=otnen

einzige Programm – neben dem *DocumentManger* – das direkter Client des *StorageManager* ist. Dieses Programm dient zur Steuerung, Konfiguration und Überwachung der elektronischen Archivierungsfunktionalität. Bei der Konfiguration werden mit diesem Programm die Festplatten-Caches und die optische Laufwerke für die Verwendung mit *PROXESS* konfiguriert. Während des Betriebs wird dieses Programm verwendet, um Images auf ein optisches Medium zu brennen. Diese Images werden vom System automatisch erstellt, wenn sich ein ausreichend großer Dateibestand angesammelt hat und sie enthalten die physischen Dateien aus Dokumenten. Die Dokumente selbst, die in der relationalen Datenbank abgelegt sind, werden nicht auf optische Medien ausgelagert.

4.3.2. Datenbank Backend

Aus der Sicht des DBMS gibt es für *PROXESS* mindestens drei Datenbanken. Es gibt pro Installation genau eine Datenbank für globale Konfigurationsdaten (`MasterDB`) und genau eine Datenbank für die Verwaltung von optischen Medien (`SMDB`). Für jede Benutzerdatenbank gibt es jeweils genau eine relationale Datenbank gleichen Namens, die alle für die Benutzerdatenbank spezifischen Konfigurations- und Laufzeitdaten enthält. Abbildung 4.7 gibt einen Überblick über die relationalen Datenbanken und die in ihnen enthaltenen Tabellen. Es soll erwähnt sein, dass es in jeder Datenbank mehr Tabellen gibt, als in der Abbildung gezeigt werden. Es sind nur diejenigen Tabellen aufgeführt, die für das beschriebene Metamodell von Bedeutung sind. Das macht etwas mehr als die Hälfte der tatsächlich existierenden Tabellen aus. Die Namen der Tabellen in der Abbildung entsprechen sinngemäß den Namen der tatsächlichen Tabellen. Die Namen wurden vereinheitlicht (in Bezug auf Groß- und Kleinschreibung) und ausgeschrieben (keine Abkürzungen).

Die **`MasterDB`** (im Regelfall heißt diese Datenbank `ProxessDB`) enthält Tabellen für die globalen Konfigurationsdaten. Es existieren Tabellen für Benutzerdatenbanken (`Databases`), Benutzer (`Users`), Gruppen (`Groups`) und Gruppenzugehörigkeit von Benutzern (`UserGroups`).

Die **`SMDB`** (*StorageManager DB*) enthält Tabellen für das Indexieren von Dokumentdateien (`Files`), sowie Tabellen für die Zuordnung von Dateien zu (abgelaufenen[10]) optischen Medien (`Volumes` und `ExpiredVolumes`).

[10] „Abgelaufen“ heißt, dass alle Dokumente dieses Mediums die Archivierungsfrist überschritten haben.

Jede **`UserDB`** (also jede Benutzerdatenbank, z.B. `Personal`, `FIBU` oder `WAWI`) enthält Tabellen für die Konfigurationsdaten der Benutzerdatenbank: Tabellen für die definierten Merkmalsfelder (`Fields`), für Dokument- und Dateitypen (`DocumentTypes` bzw. `FileTypes`) und für Dokumenttypfelder (`OverridenFields`). Für die Instanzdaten gibt es Tabellen für Dokumente (`Documents`), Dokumentverknüpfungen (`DocumentLinks`) und Dokument-Datei-Mappings (`Files`). Die in der Tabelle `Fields` deklarierten Merkmalsfelder finden sich in der Tabelle `Documents` als Spalten – mit entsprechendem Namen und Datentyp – wieder.

4.3.3. Session und Object Handles

Bei der Kommunikation mit Komponenten von *PROXESS Server*[11] werden RPC-Context-Handles (siehe Kapitel 5) intensiv eingesetzt. Durch Context-Handles kann ein serverseitiger Zustand zwischen Client und Server kommuniziert werden. In *PROXESS* heißen diese Handles **Session-Handle** (oder auch **Login-Handle**) wenn von dem Handle geprochen wird, dass die Benutzeranmeldung repräsentiert und **Object-Handle** wenn Handles, die geöffnete Objekte repräsentieren, gemeint sind. Darunter fallen insbesondere: **Document-Handles** für geöffnete Dokumente, **File-Handles** für geöffnete Dateien, **Search-Handles** für geöffnete Suchabfragen und **Result-Handles** für Trefferlisten. Listing 4.1 zeigt den schematischen Ablauf bei der Verwendung von Handles.

4.3.4. Mengengerüst

Für die Serverprogramme wird im Folgenden das Ergebnis einer Messung der **Lines of Code** (**LOC**) präsentiert. Für den *DocumentManager* wird im darauf folgenden Abschnitt zusätzlich ein Überblick über verschiedene Gruppen von Interface-Methoden gegeben.

4.3.4.1. LOC-Metriken

Die Messung setzt sich aus den Teilen Source Lines of Code (SLOC), Kommentare (Comments) und Leerzeilen (Blank lines) zusammen. Zur Ermittlung der Zahlen wurde das Tool *LOCCounter*[12] von *Microsoft* eingesetzt. Die Basis für die Messung liefert

[11] Das schließt auch die Kommunikation von Server-Programmen untereinander ein.
[12] *`http://archive.msdn.microsoft.com/LOCCounter`*

```
// connect to PROXESS with user digest "33e-6f2"
login_handle lh = KnlConnect( GetBinding(), Platform.IntelX86, "33e-6f2" );
// login user "admin"
SsnUserLogin( lh, "admin", "s3cr3t" );
// connect to database "Personal"
SsnConnectDatabase( lh, "Personal" );
// open document with id 1230711
object_handle oh = DocOpen( lh, 1230711 );
/* do stuff... */
// close doc. pass object handle by reference.
DocClose( lh, &oh);
// close connection. pass login handle by reference.
KnlDisConnect( &lh );
```

Listing 4.1: Schematischer Ablauf bei der Verwendung von Handles

ein Schnappschuss aus dem Revisionskontrollsystem, mit dem der Source-Code von *PROXESS* verwaltet wird. Dabei ist anzumerken, dass die automatisch generierten Interface-Stubs ebenfalls der Revisionskontrolle unterliegen und von daher vor der Messung händisch gelöscht worden sind. Bei den Messungen der drei Varianten des *DatabaseManager* wurden die Gemeinsamkeiten der Varianten ebenfalls von Hand ermittelt. Die Messung ist also nicht unbedingt exakt.

Tabelle 4.1 stellt die Ergebnisse der LOC-Messung pro Server-Modul dar. Eine Konfiguration von *PROXESS* beinhaltet genau eine Variante des *DatabaseManager*s, die je nach dem verwendetem Datenbanksystem ausgewählt wird. Der Umfang einer *PROXESS*-Konfiguration liegt also zwischen 280 und 290 kSLOC.

Modul	SLOC	Comments	Blank lines	**Total**
DocumentManager	150.072	22.078	36.291	**208.441**
StorageManager	74.289	17.619	15.954	**107.862**
DatabaseManager (Common)	36.378	5.455	8.155	**49.988**
DatabaseManager (MSSQL)	7.465	1.279	1.683	**10.427 (60.415)**
DatabaseManager (Oracle)	4.837	441	1.273	**6.551 (56.539)**
DatabaseManager (Caché)	8.676	1.003	2.092	**11.771 (61.759)**

Tabelle 4.1.: *LOC*-Metrik

Tabelle 4.2 gibt an, wie viele SLOC durchschnittlich auf eine Kommentarzeile entfallen und Tabelle 4.3 quantifiziert die Anzahl der Datein pro Server-Modul und gibt zusätz-

lich pro Modul die Anzahl an SLOC derjenigen Datei an, die die meisten SLOC*s* beinhaltet. Die minimale Anzahl an SLOC pro Datei liegt bei allen Modulen unter fünf und wird daher in der Tabelle nicht aufgeführt.

Modul	SLOC per Comment
DocumentManager	6.80
StorageManager	4.22
DatabaseManager (MSSQL)	8.97
DatabaseManager (Oracle)	9.59
DatabaseManager (Caché)	9.56

Tabelle 4.2.: *SLOC* per Comment

Modul	# Files	Average SLOC per File	Max SLOC per File
DocumentManager	281	534	8275
StorageManager	236	314	6099
DatabaseManager (Common)	113	322	3286
DatabaseManager (MSSQL)	13	339	2968
DatabaseManager (Oracle)	10	483	2858
DatabaseManager (Caché)	14	619	2950

Tabelle 4.3.: Anzahl Dateien und Dateigröße

4.3.4.2. Interface-Funktionen *DocumentManager*

Die exportierte Schnittstelle von *PROXESS* ist die angebotene Schnittstelle des *DocumentManager*s. Es ist eine einzelne RPC-Schnittstelle mit insgesamt 393 Funktionen (Fassade). Einzelne Objekte werden i.d.R. über eine systemweit eindeutige `ID` identifiziert, die durch eine 64-Bit große Ganzzahl dargestellt wird. Für den Zugriff auf Nutzdaten wie Dokumente oder Dateien wird zusätzlich das in Abschnitt 4.3.3 eingeführte Handle-Konzept verwendet: Zum Beispiel werden Dokumente anhand ihrer `ID` geöffnet und der *DocumentManager* gibt als Antwort einen Handle zurück. Mit Hilfe dieses Handle*s* wird das zu bearbeitende Dokument in folgenden Interface-Aufrufen identifiziert. Nach der Verwendung muss der Handle vom Benutzer geschlossen werden.

Tabelle 4.4 gibt einen Überblick über die funktionale Verteilung der Funktionen. Die einzelnen Kategorien enthalten jeweils sowohl Funktionen zur Bearbeitung der Nutzdaten als auch der Metadaten. Die Kategorisierung entstand durch das programmatische Zählen von Präfixen in den Methodennamen. Diese Trennung der Funktionalität

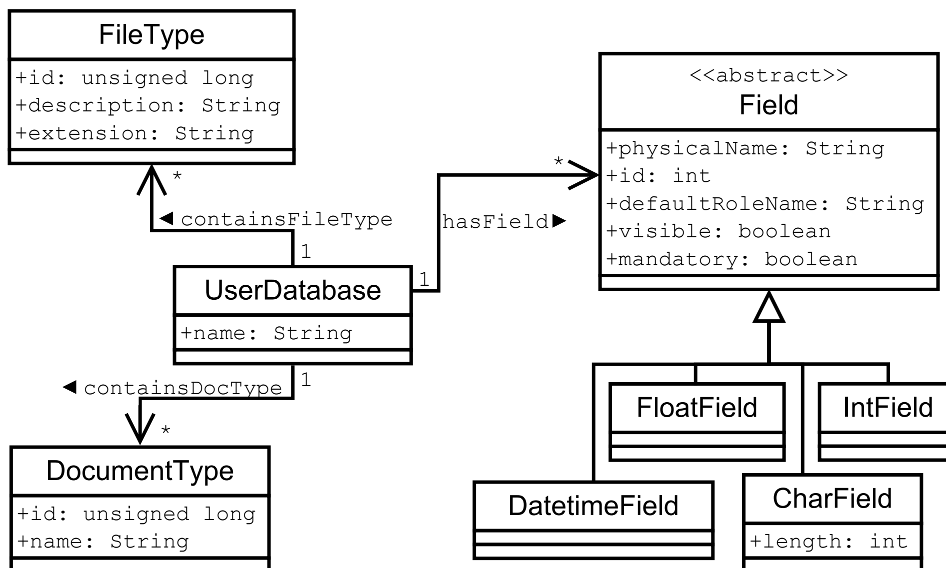

Abbildung 4.3.: Datenbanken, Typen und Felder

Verbindungsfunktionen	6
Session-Funktionen	72
Benutzer- / Gruppen-Funktionen	17
Dokument-Funktionen	89
Datei-Funktionen	49
Abfragefunktionen	39
Sonstige Funktionen	119
Gesamt	**391**

Tabelle 4.4.: DocumentManager Interface

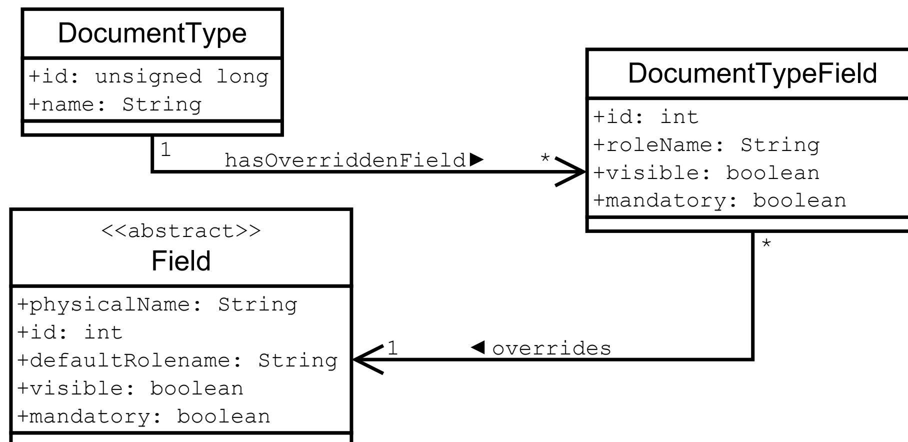

Abbildung 4.4.: Dokumenttypfelder

anhand von Präfixen ist nicht immer gültig, gibt aber einen schnellen und zu großen Teilen richtigen Überblick. Die einzelnen Kategorien werden im Folgenden kurz beschrieben:

Verbindungs-Funktionen, Präfix: `Krn`. Die Kategorie **Verbindungs-Funktionen** enthält Funktionen, die beim Verbindungsaufbau bzw. Verbindungsabbau zum *DocumentManager* eingesetzt werden. Hier erhält der Client das *Login-Handle* mit dem die Session bei allen Folgeaufrufen identifiziert wird.

Session-Funktionen, Präfix: `Ssn`. Die Kategorie **Session-Funktionen** enthält allgemeine Funktionen zur Benutzeranmeldung, der Verwaltung von Datenbanken und Datenbank-Feldern und zur Abfrage von weiteren öffentlichen Informationen wie z.B. Standard-Profileinstellungen für Client-Programme, Informationen zur Lizenzdatei oder Abfrage aller definierten Benutzer und Gruppen.

Benutzer- und Gruppen-Funktionen, Präfix: `Usr` bzw. `Grp`. In der Kategorie **Benutzer- und Gruppen-Funktionen** sind alle Funktionen zum Anlegen und Verändern von Benutzern bzw. Gruppen enthalten sowie Funktionen, mit denen Benutzer verschiedenen Gruppen zugewiesen werden können.

Dokument-Funktionen, Präfix: `Doc`. Die Kategorie **Dokument-Funktionen** fasst alle Funktionen zusammen, mit denen Dokumente oder Dokumenttypen erstellt oder be-

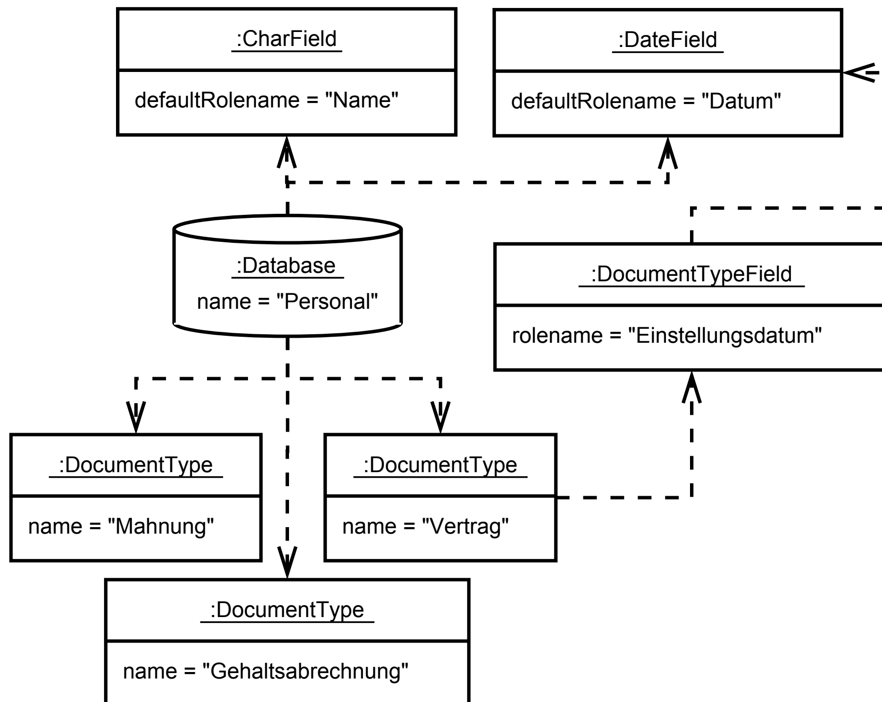

Abbildung 4.5.: Beispielschema

arbeitet werden. Die meisten Funktionen für Dokumenttyp*en* beginnen mit dem Präfix `DocType`, allerdings ist das nicht konsequent durchgehalten. Ein neuer Dokumenttyp wird z.B. mit der Funktion `DocNewType(...)` angelegt.

Datei-Funktionen, Präfix: `File`. Analog beinhaltet die Kategorie **Datei-Funktionen** alle Funktionen zum Abrufen und Bearbeiten von Dateien oder Dateitypen.

Abfrage-Funktionen, Präfix: `Gs`, `Qbe`, `Qs`, `Sqr` und `Rst`. Die Kategorie **Abfrage-Funktionen** enthält Funktionen zur Formulierung jeder Art von Suchabfragen und Funktionen zum Abholen von Trefferlisten (Funktionen mit dem Präfix `Rst`). Die Präfixe für die Abfrage-Schnittstellen haben folgende Bedeutung:

- **GeneralSearch**, Präfix `Gs`. Allgemeinste Abfrageschnittstelle. Ermöglicht die Formulierung von beliebigen Suchausdrücken.
- **QueryByExample**, Präfix: `Qbe`. Die Suchabfrage wird durch ein virtuelles Dokument formuliert. Alle Dokumente, deren Felder die selben Werte haben, werden in die Trefferliste aufgenommen.

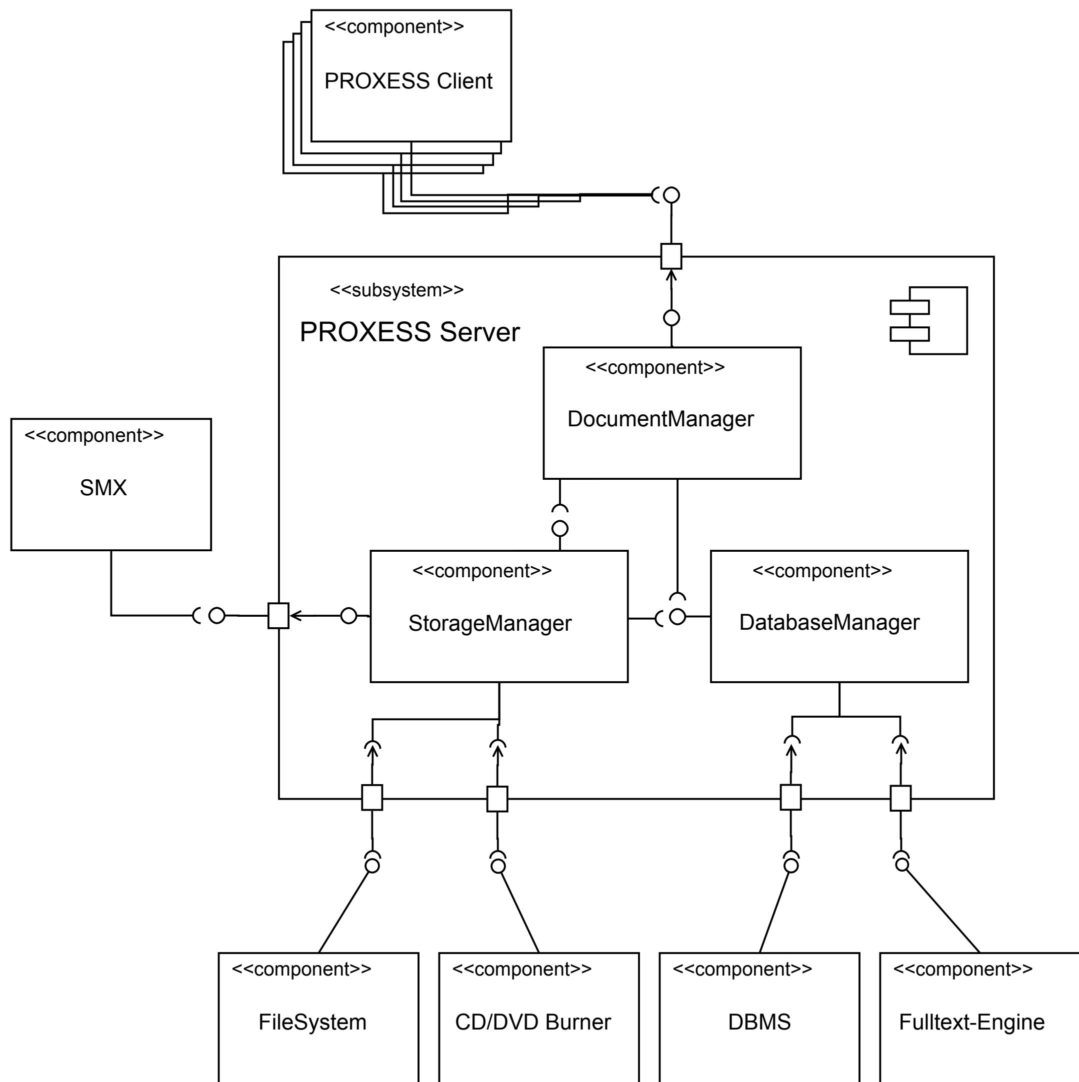

Abbildung 4.6.: PROXESS Komponentendiagramm

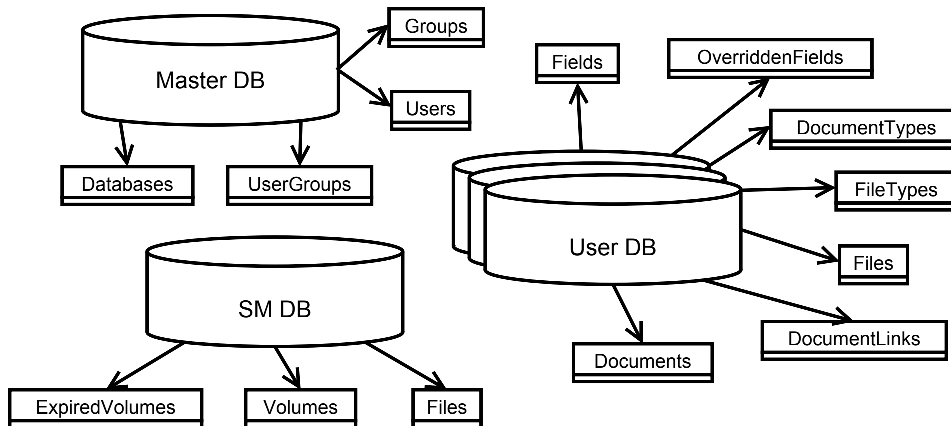

Abbildung 4.7.: Überblick: Relationale Datenbanken

- **QuickSearch**, Präfix: `Qs`. Schnittstelle für reine Volltext-Suchabfragen.
- **SavedQueries**, Präfix: `Sqr`. Die einer Trefferliste zu Grunde liegende Abfrage kann gespeichert werden. Mit dem `Sqr`-Interface können diese gespeicherten Suchabfragen erneut ausgeführt werden.

Sonstige Funktionen. Übrige Funktionen stellen von der Anzahl her den größten Teil des Interface dar. Darunter fallen z.B. folgende Funktionsgruppen:

- Validierungsregeln für Felder
- Benutzer- und Client-spezifische Profilinformationen
- Vorlagedateien für bestimmte Dateitypen
- Volltext-Trefferhighlighting in Dateien
- Abrufen von installierten Zertifikaten

Die Auflistung ist nicht vollständig, enthält aber die wichtigsten Kategorien.

4.4. Zusammenfassung

PROXESS ist ein Dokumenten-Management-System (DMS) mit einer Client-/Server-Architektur. Der Server-Teil des Systems, *PROXESS Server*, besteht neben einer Reihe von Konfigurationsprogrammen aus drei Server-Programmen, die die Kernfunktionalität des DMS implementieren: Die Verwaltung von Dokumenten und Dateien.

Der *DocumentManager* regelt die Benutzerverwaltung und die Zugriffskontrolle auf Dokumente und Dateien. Die gesamte Kernfunktionalität des DMS wird von *DocumentManager* über eine Fassade exportiert. Dokumente und Konfigurationsdaten werden in relationalen Datenbanken gespeichert, die durch den *DatabaseManager* abgekapselt werden. Der *DatabaseManager* überführt alle Daten aus der Datenbank in ein *PROXESS*-spezifisches und systemweit gültiges Format. Dateien von Dokumenten werden physisch im Dateisystem gespeichert, das durch den *StorageManager* abgekapselt wird.

Das Gesamtsystem besteht weiterhin aus einer Reihe von Client-Programmen, die die von *PROXESS Server* angebotenen Services dazu verwenden, um weitere Funktionalität des DMS realisieren.

Im Hauptteil dieser Arbeit wird ein Umstellungsprozess für die Unicode-Migration von *PROXESS* entworfen und exemplarisch am *DatabaseManager* angewendet.

5. Weitere Grundlagen

Dieses Kapitel führt weitere Grundlagen ein, die für das Verständnis der Arbeit wichtig sind.

Abschnitt 5.1 führt den Begriff C-Style-String bzw. `NUL`-terminierte Strings ein. Diese Art von Strings wird sowohl in der Implementierung von *PROXESS* als auch in der *Windows*-API verwendet.

Abschnitt 5.2 führt *Microsoft*-spezifische Grundlagen ein, wobei ein besonderer Blick auf Strings im Allgemeinen und auf die Unterstützung von Unicode im speziellen geworfen wird. In diesem Zusammenhang werden einige grundlegende Entscheidungen für die Unicode-basierte Version von *PROXESS* getroffen.

5.1. C-Style-Strings

Ein **`NUL`-terminierter** String (oder: **C-Style-String**) ist Array aus Zeichen, in dem das `NUL`-Zeichen (das Zeichen mit dem Byte-Wert `0x00`, als Literal `'\0'`) das Ende des Strings markiert. Die einfachste Form von C-Style-String ist die Deklaration eines statischen `char[]`. So wird z.B. die Deklaration

```
char string[] = "abc";
```

vom Compiler in ein statisches Array der Länge 4 umgeformt, nämlich

```
char string[4] = {'a','b','c','\0'};
```

. Alle String-Funktionen der C-Standardbibliothek (z.B. `strcpy`, `strcat` oder `strcmp`) funktionieren nur dann korrekt, wenn `NUL`-terminierte Strings verwendet werden. Daher muss bei der Reservierung von Speicher immer eine extra Speichereinheit Platz für das `NUL`-Zeichen eingeplant werden. Listing 5.2 zeigt, wie das Kopieren von C-Style-Strings mit der Funktion `strcpy` realisiert wird.

```
char source[] = "Copy me!";
char *dest = new char[strlen(source)+1];
strcpy(dest, source);
```

Listing 5.2: Kopieren von C-Style-Strings

Im Folgenden werden beispielhaft zwei problematische Eigenschaften von C-Style-Strings aufgezeigt. Beide Eigenschaften sind auf die manuelle Speicherverwaltung zurückzuführen, die stets eine große Sorgfalt erfordert. Diese Eigenschaften motivieren, dass die Unicode-basierte Version von *PROXESS* keine C-Style-Strings verwendet, sondern eine Stringklasse verwendet. Der Gedanke wird in Kapitel 7.3 wieder aufgenommen.

Gefahr von Pufferüberläufen (Buffer-Overflow). Wird im letzten Beispiel die `+1` bei der Reservierung des Speichers vergessen, kann durch das Kopieren mit `strcpy` kein gültiger C-Style-String erzeugt werden. In diesem Fall überschreibt die Funktion `strcpy` sogar das erste Byte hinter dem reservierten Speicher, was fast unausweichlich zu undefiniertem Programmverhalten führt. Dieses Fehlverhalten ist unter dem Begriff Buffer-Overflow bekannt und kann u.U. gezielt dazu eingesetzt werden, den Kontrollfluss zu manipulieren (indem durch einen Buffer-Overflow die Rücksprungadresse der aktuellen Funktion überschrieben wird). Bei der Verwendung von C-Style-Strings muss vor der Verwendung also immer ein ausreichend großer Speicherbereich reserviert worden sein, um solche Probleme zu vermeiden – sofern dies überhaupt möglich ist.

Um den Aufrufer vor einem Pufferüberlauf zu schützen, wurden sichere Varianten der C-Stringfunktionen in den C-Standard aufgenommen. Im Fall von `strcpy` heißt die sichere Variante `strcpy_s`, die einen zusätzlichen Parameter enthält, der die Größe des Zielpuffers spezifiziert.

Notwendigkeit mehrfacher Speicherfreigabe. Ein weiteres Problem stellt die manuelle Speicherverwaltung dar. Im obigen Beispiel wurde der String mit dem Operator `new` allokiert. Das bedeutet, dass der Speicher durch ein entsprechendes `delete` wieder freigegeben werden muss (das gilt analog für `malloc/free`). Die manuelle Speicherverwaltung erweist sich insbesondere bei der Fehlerbehandlung als aufwendig, was in Listing 5.3 am einem Beispiel mit C++-Exceptions demonstriert wird.

```
void func(){
 char *cpCompName = 0;
 try{
  size_t szCompName = ServiceA.GetMaxLength()+1;
  cpCompName = new char[szCompName];
  ServiceA.GetComputerName(cpCompName,szCompName)
  ServiceB.RegisterHost(cpCompName);
  // delete memory on success
  delete cpCompName;
 }catch(...){
  // delete memory on failure
  delete cpCompName;
 }
}
```

Listing 5.3: Fehlerbehandlung mit C-Style-Strings

Sowohl im Erfolgsfall als auch im Fehlerfall muss der Speicher des dynamisch allokierten Strings wieder freigegeben werden, damit kein Speicherleck entsteht.

5.2. *Microsoft*-spezifische Grundlagen

Dieser Abschnitt führt *Microsoft*-spezifische Grundlagen ein, die für diese Arbeit relevant sind. In Abschnitt 5.2.1 werden diejenigen Aspekte der **API von *Microsoft Windows*** erläutert, die im Hinblick auf Unterstützung und Handhabung von Unicode-Strings von Bedeutung sind.

In den folgenden Abschnitten werden zwei **Technologien zur Interprozesskommunikation** vorgestellt, die häufig zur Kommunikation zwischen *PROXESS*-Modulen, die in unterschiedlichen Prozessen laufen, eingesetzt werden. Die Technologie *Microsoft RPC* (Remote Procedure Call) wird in Abschnitt 5.2.2 eingeführt, die Technologie *Microsoft COM* (Component Object Model) wird in Abschnitt 5.2.3 eingeführt.

5.2.1. Unicode in der *Microsoft Windows* API

Dieser Abschnitt erläutert die Unterstützung und Handhabung von Unicode-Strings im Zusammenhang mit dem Betriebssystem *Microsoft Windows*. Zuerst wird die Unterstützung des Unicode-Standard durch das Betriebssystem *Microsoft Windows* unter-

sucht, und dabei wird die minimale Version von *Windows* ermittelt, die für den Einsatz der Unicode-fähigen Version von *PROXESS* sinnvoll erscheint. Im Anschluss daran wird ein Überblick über die Implementierung der wichtigsten Unicode-Algorithmen geben. Abschließend wird die programmiertechnische Verwendung von Strings im Allgemeinen und Unicode-Strings im Speziellen im Zusammenhang mit der *Windows*-API erläutert.

Allgemeine Unicode-Unterstützung von *Microsoft Windows*

Nach einem Artikel von *Microsoft* ([1]) verwendet *Windows NT* mindestens seit Version *3.1* intern Unicode zur Darstellung von Strings. Nach Allen, Kaplan und Wissink ([2]) wird ab der Version ***Windows NT 6.0*** (d.h. *Windows Vista* bzw. *Windows Server 2008*) der **Unicode-Standard** in der **Version 5.0 unterstützt**. Ältere Versionen von Windows unterstützen eine Auswahl von Features von Unicode 1.1 bis Unicode 3.1. Eine genaue Angabe darüber, welche Teile von Unicode in den älteren Versionen von *Windows*[3] unterstützt werden, konnte nicht ausfindig gemacht werden.

Weil sich erst ab Version *Windows NT 6.0* eine präzise Aussage zur Unicode-Unterstützung findet, sollte diese Version als minimale Voraussetzung für ein Unicode-fähiges *PROXESS* gelten. Daher können bei der Entwicklung der Unicode-fähigen Version von *PROXESS* auch solche Funktionen der API von *Microsoft Windows* eingesetzt werden, die erst mit der Version *Windows NT 6.0* eingeführt worden sind.

Windows verwendet intern für alle Strings die Kodierung UTF16-LE.

> *On Windows platforms, which are mostly little endian, UTF-16LE is just called "Unicode" and UTF-16BE is just called "Unicode (Big Endian)"* [Kap05]

Daher ist es für die Unicode-basierte Version von *PROXESS* sinnvoll, ebenfalls die Kodierung UTF16-LE für Strings zu verwenden. Dies wird dadurch unterstrichen, dass die von *PROXESS* verwendete Technologie *COM* (siehe Abschnitt 5.2.3) und das für *PROXESS* relevante Datenbanksystem *Microsoft SQL Server* (siehe Kapitel 8) Unicode ebenfalls mit der Kodierung UTF16-LE verbinden.

---

[1] `http://support.microsoft.com/kb/99884` [Letzter Zugriff am 30.05.2012]

[2] `http://msdn.microsoft.com/de-de/magazine/cc163490.aspx` [Letzter Zugriff am 30.05.2012]

[3] Mit „älteren Versionen" sind die Versionen von *Windows NT 3.1* bis ausschließlich *Windows NT 6.0* gemeint.

Allgemeine Handhabung von Strings bei Funktionen der *Windows*-API

Dieser Abschnitt führt die Handhabung von Strings im Zusammenhang mit Funktionen der API von *Microsoft Windows* ein. Die **Windows-API** (**Win32-API**) stellt eine Programmierschnittstelle zum Win32-Subsystem[4] von Windows dar und enthält alle Betriebssystemfunktionen, die von Windows-Programmen verwendet werden können. Diese API ist eine Schnittstelle für die Programmiersprache C. Alle Funktionen der *Windows*-API, die String-Parametern haben, erwarten diese Strings als `NUL`-terminierte Strings. Viele Funktionen mit String-Parametern, insbesondere ältere Funktionen, existieren in drei Varianten.

1. Eine „klassische“ Variante, die Strings in der Kodierung einer bestimmten *Windows*-Codepage erwartet und den Datentyp `LPSTR` (Typdefinition für den eingebauten Datentyp `char*`) verwendet. An den Namen der Funktion wird das Suffix `A` angefügt.
2. Eine Unicode-basierte Variante, die Strings in der Kodierung UTF16-LE erwartet und den Datentyp `LPWSTR` (Typdefinition für den eingebauten Datentyp `wchar_t*`, der für die Plattform *Microsoft Windows* als `unsigned short` definiert ist) verwendet. An den Namen der Funktion wird das Suffix `W` angefügt.
3. Eine generische Variante, die Strings je nach dem, ob das Präprozessor-Makro `UNICODE` definiert worden ist, in einer 8-Bit-Kodierung oder der UTF16-LE-Kodierung erwartet. In den aktuellen Headern von *Windows* wird die generische Funktion überwiegend durch ein Makro definiert, das, wenn `UNICODE` definiert ist, zu der `W`-Variante expandiert wird und andernfalls zu der `A`-Variante.

Die generischen Varianten von Stringfunktionen wurden vor allem eingeführt, um Software gleichzeitig für *Windows NT*, das schon früh Unicode unterstützt hat, und die *Windows*-Varianten *9x/Me*, die lange kein Unicode unterstützt haben, kompilieren zu können:

> Back when applications needed to support both Windows NT as well as Windows 95, Windows 98, and Windows Me, it was useful to compile the same code for either ANSI or Unicode strings, depending on the target platform ([Mic10]).

Die verschiedenen Varianten von Stringfunktionen werden in Listing 5.4 am Beispiel der Funktion `SetWindowText` aus der *Windows*-API ([Micd]) aufgeführt.

[4]Das Win32-Subsystem ist als User-Mode von Windows zu verstehen.

```
// function definition of Windows-Codepage-based variant
BOOL WINAPI SetWindowTextA( _In_  HWND hWnd,
  _In_opt_  LPCSTR lpString );
// function definition of Unicode-based variant
BOOL WINAPI SetWindowTextW( _In_  HWND hWnd,
  _In_opt_  LPCWSTR lpString );

// definition of generic function via macro
#ifdef UNICODE
#define SetWindowText  SetWindowTextW
#else
#define SetWindowText  SetWindowTextA
#endif
```

Listing 5.4: Beispiel für Funktionsvarianten in der *Windows*-API

Notation von String-Literalen

Die Notation von Stringliteralen ist im C-Standard festgelegt. Für Strings, die auf dem Datentyp `char` basieren, werden Literale in doppelten Anführungszeichen notiert, z.B. `"Hello String"`. Für Strings, die auf dem Datentyp `wchar_t` basieren, werden Literale in doppelten Anführungszeichen mit einem führenden `L` notiert, z.B. `L"Hello String"`. Das *Microsoft*-spezifische Makro `TEXT(x)` (bzw. `_T(x)`) produzieren die zum Präprozessor-Makro `UNICODE` passenden Stringliterale. Ist `UNICODE` nicht definiert, wird der Ausdruck `_T("Hello String")` zu `"Hello String"` ausgewertet. Ist `UNICODE` definiert, wird der Ausdruck `_T("Hello String")` zu `L"Hello String"` ausgewertet.

In der heutigen Zeit, in der jede Variante/Version von *Windows* Unicode-fähig ist, sind diese generischen Makros nicht mehr von großer Bedeutung:

> The TEXT and TCHAR macros are less useful today, because all applications should use Unicode ([Mic10]).

Die Verwendung der generischen Makros verleitet außerdem zu der Annahme, dass der Unterschied in der Handhabung von Strings zwischen einem typischen 8-Bit-Zeichensatz und dem Unicode-Zeichensatz ausschließlich vom verwendeten Datentyp

abhängt. Wie in Kapitel 3 verdeutlicht worden ist, müssen bei der Verwendung von Unicode grundlegende Annahmen über Strings verworfen werden.

Aus diesen Gründen soll auf die Verwendung der generischen Datentypen und Funktionen verzichtet werden. Die Unicode-API von *Windows* soll explizit verwendet werden.

Umsetzung von Unicode-Algorithmen in der API von *Microsoft Windows*

Dieser Abschnitt gibt einen kurzen Überblick über die Implementierung der wichtigsten Unicode-Algorithmen (Unicode-Collation-Algorithm und Unicode-Normalization-Algorithm) in der *Windows*-API. Die aufgeführten Funktionen sind weitere gute Beispiele für typische String-Funktionen in der *Windows*-API. In beiden Fällen sind die Funktionen nur für Unicode-Strings (Parameter vom Typ `LPCWSTR`) definiert, d.h. es gibt weder eine 8-Bit-Variante (Parameter vom Typ `LPCSTR`) noch eine generische Variante (Parameter vom Typ `LPCTSTR`).

Unicode-Collation-Algorithm (UCA). Seit der Version *Windows NT 6.0* wird der UCA durch die Funktion `CompareStringEx` realisiert. In älteren Versionen von *Windows* heißt die entsprechende Funktion `CompareString`, die zwar auch Unicode-Strings vergleichen kann, aber hauptsächlich für Codepage-basierte Strings ausgelegt ist. Nach [Mica] soll zum Vergleich von Unicode-Strings bevorzugt die Funktion `CompareStringEx` verwendet werden. Die Beschreibung der Funktion und ihre Signatur werden in Listing **??** aufgeführt und sind aus der *MSDN Library* ([Micb]) entnommen.

Die Funktion gibt im Erfolgsfall einen der vordefinierten Werte `CSTR_LESS_THAN`, `CSTR_EQUAL` und `CSTR_GREATER_THAN` zurück.

Der String-Parameter `LPCWSTR`[5] `lpLocaleName` spezifiziert die Region bzw. Sprache, die für den Vergleich maßgeblich ist. Dieser Parameter entscheidet, welcher Collation-Element-Table (CET) verwendet wird. Mögliche Werte sind z.B. `en-US`, `de-DE` oder `de-DE_phonebook`.

Der Parameter `DWORD dwCmpFlags` legt weitere Bedingungen für den Vergleich fest, z.B. ob beim Vergleich Groß- und Kleinschreibung ignoriert werden soll.

---

[5] „`Long Pointer const Wide String`“, eine Typdefinition für den Datentyp `wchar_t const *`.

```
// Compares two Unicode (wide character) strings,
// for a locale specified by name.
int CompareStringEx(
  _In_opt_  LPCWSTR lpLocaleName,
  _In_      DWORD dwCmpFlags,
  _In_      LPCWSTR lpString1,
  _In_      int cchCount1,
  _In_      LPCWSTR lpString2,
  _In_      int cchCount2,
  _In_opt_  LPNLSVERSIONINFO lpVersionInformation,
  _In_opt_  LPVOID lpReserved,
  _In_opt_  LPARAM lParam );
```

Listing 5.5: Funktion `CompareStringEx`

Die folgenden vier Parameter spezifizieren die Datenpuffer und Längenangaben der zu vergleichenden Strings.

Die letzten drei Parameter sind z.Zt. reserviert und müssen mit dem Wert `NULL` (Null-Pointer) belegt werden.

Unicode-Normalization-Algorithm. Seit der Version *Windows NT 6.0* wird der Unicode-Normalization-Algorithm durch die Funktion `NormalizeString` implementiert. In älteren Versionen von *Windows* gibt es keine Entsprechung für diese Funktion. Die Beschreibung der Funktion und ihre Signatur werden in Listing 5.6 aufgeführt und sind aus der *MSDN Library* ([Mic12c]) entnommen. Hier ist anzumerken, dass der Algorithmus anscheinend nur konform zum Unicode-Standard 4.0 implementiert ist.

Die Funktion gibt die Länge des Ausgabestrings in Codeunits zurück.

Der Parameter `NORM_FORM NormForm` erwartet einen Enumerationswert, der eine der vier Normalformen NFC, NFD, NFKC und NFKD spezifiziert (in der *Windows*-API werden die Normalformen lediglich mit C, D, KC und KD bezeichnet).

Die zwei folgenden Parameter spezifizieren den Puffer und dessen Größe für den Eingabestring.

Die letzten zwei Parameter spezifizieren den Puffer und dessen Größe für den Ausgabestring. Wird für den Parameter `LPWSTR lpDstString` der Wert `NULL` und für

```c
// Normalizes characters of a text string
// according to Unicode 4.0 TR#15.
int NormalizeString(
  _In_       NORM_FORM NormForm,
  _In_       LPCWSTR lpSrcString,
  _In_       int cwSrcLength,
  _Out_opt_  LPWSTR lpDstString,
  _In_       int cwDstLength );
```

Listing 5.6: Funktion `NormalizeString`

den Parameter `int cwDstLength` der Wert `0` übergeben, so gibt die Funktion die benötigte Größe des Ausgabepuffers in Codeunits zurück.

Die Betrachtung der Implementierung spezifischer Unicode-Algorithmen schließt dieses Thema. Im Anschluss werden zwei Technologien zur Interprozesskommunikation vorgestellt, *Microsoft RPC* und *Microsoft COM*. Von diesen Technologien ist insbesondere *Microsoft RPC* von Bedeutung für diese Arbeit.

5.2.2. *Microsoft RPC*

Dieser Abschnitt führt die Technik **Remote Procedure Call (RPC)** ein. RPC ist eine Grundlage für das Verständnis der Serverkommunikation von *PROXESS*, das Microsofts Implementierung von RPC einsetzt.

Definition und prinzipielle Funktionsweise

Remote Procedure Call (**RPC**) ist eine Technik, mit der das Aufrufen von Funktionen in verschiedenen Adressräumen ermöglicht wird. Es ist eine spezielle Form der Interprozesskommunikation (IPC), wobei die Kommunikation durch ein **Client-/Server-Modell** realisiert wird.

> The *remote procedure call* (*RFC*) mechanism is the simplest way to implement *client-server* applications, because it keeps the details of network communications out of your application code. The idea is that each side behaves, as much as possible, the way it would within a traditional application: the programmer on the client side issues a call, and the programmer on the server side writes a procedure to carry out the desired function. ([SR95], S. 1)

Bei der Verwendung von RPC gibt es mehrere **integrale Bestandteile**: Das Interface, den Server, den (die) Client(*s*), eine RPC-Runtime und einen Interface-Compiler, der verschiedene, an das Interface angepasste Stubs erzeugt, die jeweils für Server und Clients als Fassade zur RPC-Runtime dienen. Im Interface werden alle Funktionen deklariert und alle Datentypen definiert, die über die Schnittstelle kommuniziert werden können. Mit Hilfe eines speziellen Compilers werden aus dieser Interface-Definition sogenannte Interface-Stubs generiert. Sowohl der Client als auch der Server müssen diese Interface-Stubs implementieren, d.h. der Server muss für jede Stub-Methode die gewünschte Funktionaltität bereitstellen und der Client muss die Stub-Methoden an geeigneter Stelle aufrufen. Abbildung 5.1 verdeutlicht den engen Zusammenhang zwischen dem Interface und den Stubs.

Zur **Laufzeit** realisiert die RPC-Runtime die eigentliche Interprozesskommunikation auf der jeweils anderen Seite der Stubs und verbirgt somit die IPC (egal ob lokal oder über ein Netzwerk) vor beiden Kommunikationspartnern. Für den Client sieht es so aus, als rufe er beim Server direkt eine Interface-Funktion auf und für den Server sieht es so aus, als ob der Client direkt eine seiner Interface-Methoden aufruft. Abbildung 5.2 skizziert den Ablauf eines RPC-Aufrufs.

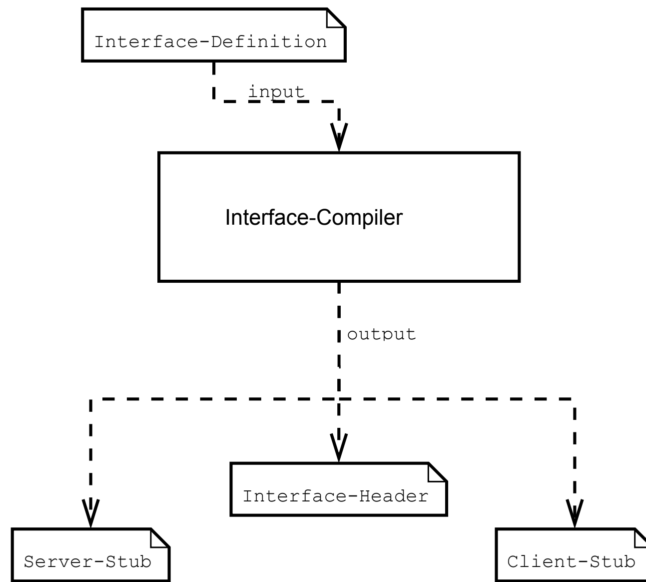

Abbildung 5.1.: RPC-Interface-Compiler

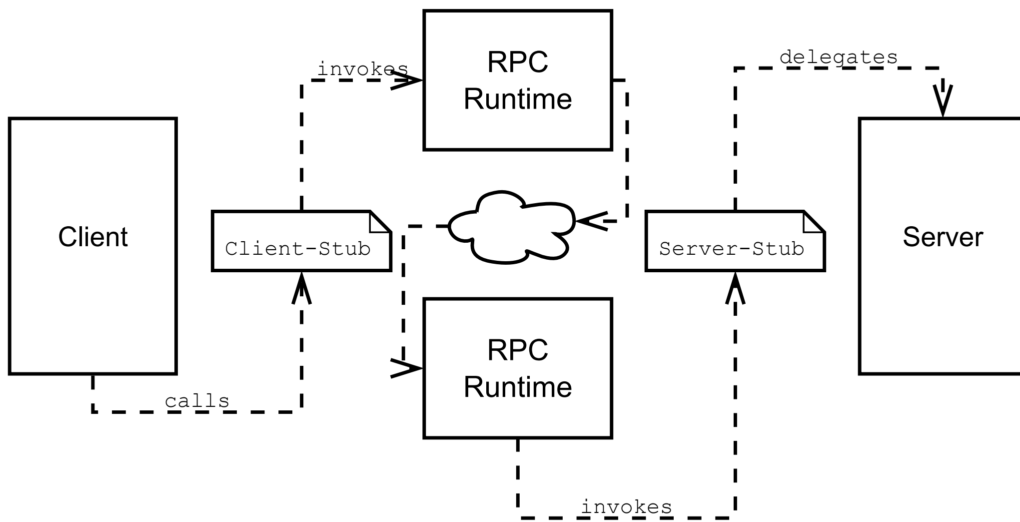

Abbildung 5.2.: RPC-Aufrufe

Die Idee wurde bereits im Jahre 1976 von James E. White formuliert ([Whi76]) und zuerst im Jahre 1984 als Teil der *Xerox Network Systems* implementiert ([BN84]). *Microsoft* bietet eine eigene Implementierung von RPC (*Microsoft RPC* oder *MSRPC*), die von John Shirley und Ward Rosenberry in dem Buch *Microsoft RPC Programming Guide* ([SR95]) ausführlich dokumentiert wird. Im Anhang wird in Kapitel B ein kleines Beispielprogramm vorgestellt, dass die Interface-Definition und die Client- und Server-seitige Implementierung verdeutlicht. Im Folgenden werden einige für das Verständnis der Arbeit wichtigen Eigenschaften von *Microsoft RPC* beschrieben.

IDL-Dateien, PODs und Attribute

Eine RPC-Schnittstelle wird durch eine IDL-Datei definiert. Die Syntax und Semantik einer IDL-Datei ist ähnlich zu der einer typischen C-Header Datei. Eine IDL-Datei enthält Typdefinitionen und Funktionsprototypen, die in einer erweiterten C-Syntax formuliert werden.

Bei den **Typdefinitionen** handelt es sich immer um POD-Objekte (Plain Old Data), d.h. durch Typdefinitionen werden lediglich die Daten und keine Funktionalität (Funktionen oder Methoden) deklariert.

Die Funktionalität des Interfaces wird durch **Prototypen für freie Funktionen** (d.h. global sichtbare Funktionen) realisiert.

Datentypen oder Funktionsprototypen können durch sog. **Attribute** annotiert werden, die dem Interface-Compiler Hinweise auf die Verwendungsart der Parameter geben. Solche Attribute werden in eckigen Klammern notiert, also `[attribute-name]`. Z.B. können Funktionsparameter mit den Attributen `in` oder `out` markiert werden, um dem Interface-Compiler Hinweise auf das Marshalling zu geben. Das Attribut `string` teilt dem Compiler mit, dass der Parameter als `NUL`-terminierter String interpretiert werden soll. Listing 5.7 zeigt einen Ausschnitt aus einer IDL-Datei, die diese Eigenschaften verdeutlicht.

Binding- und Context-Handles

In der Einführung von *PROXESS* wurde ein Handle-Konzept vorgestellt, mit dem auf geöffnete Objekte verwiesen wird. Dieses Handle-Konzept ist ein Teil von RPC und wird in diesem Abschnitt beschrieben.

```
1  // type definitions
2  typedef struct{
3    int length;
4    char *array;
5  } MyString;
6  typedef char * MyCharPointer;
7  // function declarations
8  void DoSomething(
9    [in] MyString stringData);
10 void DoSomethingElse(
11   [in,string] MyCharPointer stringData);
12 void GetSomething(
13   [out] MyString *stringData);
14 void GetSomethingElse(
15   [out,string] MyCharPointer *stringData);
16 void ModifySomething(
17   [in,out] MyString *stringData);
18 void ModifySomethingElse(
19   [in,out,string] MyCharPointer *stringData);
```

Listing 5.7: Beispiel: IDL-Datei mit Typdefinitionen und Attributen

Ein **Binding-Handle** ist ein Objekt, dass die Verbindung eines RPC-Clients zu einem RPC-Server identifiziert. Eine RPC-Schnittstelle kann so konfiguriert werden, dass entweder implizite Binding-Handles oder explizite Binding-Handles verwendet werden.

Im Beispiel, das in Listing 5.7 aufgeführt wird, wird ein **implizites Binding Handle** verwendet. Die Funktionen enthalten keinen Parameter, der das Binding zum RPC-Server spezifiziert. Auf diese Weise kann jeder Client höchstens eine Verbindung zu einem Server aufbauen.

Werden **explizite Binding-Handles** verwendet, muss jede Funktion einen zusätzlichen Parameter enthalten, der die Verbindung zwischen RPC-Client und RPC-Server spezifiziert. Dieser Parameter hat den vordefinierten Typ `handle_t` und muss mit dem Attribut `in` markiert werden. Durch die explizite Angabe des Bindings ist es Clients möglich, mehrere Verbindungen zum selben RPC-Server aufzubauen.

Ein Sonderform von expliziten Binding-Handles sind Context-Handles. Ein **Context-Handle** ist ein explizites Binding-Handle, das zusätzlich Zustandsinformationen einer Verbindung identifiziert. Ein Context-Handle wird durch eine Typdefinition deklariert, die mit dam Attribut `context_handle` markiert worden ist. In *PROXESS* werden

Context-Handles z.B. dazu verwendet, um Verweise auf geöffnete Dokumente und Dateien zu repräsentieren.

Listing 5.8 zeigt einen Ausschnitt aus einer IDL-Datei, die explizite Binding-Handles und Context-Handles verwendet.

```
1 // type definitions
2 typedef [context_handle] MyHandle;
3 // function declarations
4 void DoSomethingWithoutContext([in] handle_t hBinding);
5 void DoSomethingWithContext([in] MyHandle hContext);
```

Listing 5.8: Beispiel: IDL-Datei mit explizitem Binding-Handle und Context-Handle

In dem im Anhang in Kapitel B beschriebenen Beispielprogramm wird deutlich, wie ein Interface für die Verwendung einer bestimmten Art von Binding-Handles konfiguriert wird. Zusätzlich wird gezeigt, welche Gestalt die vom Interface-Compiler generierten Stubs haben, und wie diese Stubs implementiert werden.

5.2.3. *Microsoft COM*

COM (**Component Object Model**) wird in der MSDN-Dokumentation von *Microsoft* ([Mic12f]) als ein *„plattformunabhängiges, objektorientiertes System"* beschrieben, *„mit dem interagierende Objekte erzeugt werden können"*. COM definiert einen Standard für Binärdaten und eine API für die dynamische Erzeugung von Objekten. Die Einhaltung dieses Standards erlaubt es, COM-Klassen unabhängig von einer bestimmten Programmiersprache zu erstellen und zu verwenden.

Eine **COM-Klasse** wird durch eine Schnittstelle beschrieben, die technisch auf RPC basiert und daher ähnliche Eigenschaften hat: Die Schnittstelle einer COM-Klasse kann Typdefinitionen und freie Funktionen enthalten (eine COM-Klasse ist also streng genommen nicht objektorientiert). Die API zur dynamischen Objekterzeugung erlaubt das Erzeugen von Instanzen einer bekannten COM-Klasse, den sog. **COM-Objekten**.

Eine COM-Klasse ist immer in einem COM-Server enthalten. Ein **COM-Server** ist eine Linkage-Unit, die aus Binärdaten besteht, die konform zum COM-Standard sind. Ein COM-Server kann mehrere – jedoch mindestens eine – COM-Klasse enthalten. Ein COM-Server kann **In-Process** (in Form einer gelinkten DLL), **Local** (als lokaler Prozess) oder **Remote** (als DCOM-Server; **D**istributed **COM**) betrieben werden.

Zum System COM gehört ebenfalls die eindeutige Registrierung von COM-Klassen in der Windows-Registry. Somit wird eine COM-Klasse lediglich durch eine GUID (**g**lobally **u**nique **id**entifier) identifiziert, und daher wird zum Zeitpunkt der Objekterzeugung keine Kenntnis über den Speicherort der Binärdaten (d.h. den Speicherort des zugehörigen COM-Servers) benötigt.

Strings in Schnittstellen von COM-Klassen sollen – nach der Spezifikation ([Kin95]) – immer Unicode-Strings sein.

PROXESS verwendet intern einige COM-Server bzw. COM-Klassen, z.B. werden COM-Server zum Logging oder zum Extrahieren von Textinformationen aus formatierten Dateien eingesetzt. Bei allen Verwendungen von COM im Zusammenhang mit *PROXESS* handelt es sich um **In-Process COM-Server**, die jeweils **genau eine COM-Klasse** exportieren. Für diese Arbeit wird ein von *PROXESS* verwendeter COM-Server mit dem Begriff **COM-Modul** bezeichnet. Alle von *PROXESS* verwendeten COM-Module verwenden für Strings an den Schnittstellen den Datentyp `BSTR`. **`BSTR`** (**B**asic **Str**ing) ist eine Struktur, die eine Längenangabe (vom Typ `int`) und ein Array von Zeichen (vom Typ `wchar_t`) enthält. Objekte vom Typ `BSTR` liegen immer in der UTF16-LE-Kodierung vor.

Als letzter Teil der Grundlagen wird das C++-Compiler-Frontend *clang* eingeführt, dass in der Analyse- und Migrationsphase der Implementierung des *DatabaseManager* eingesetzt wird.

5.3. Zusammenfassung

Der Begriff C-Style-String (bzw. `NUL`-terminierter String) wurde als Begriff für eine in der Programmiersprache C typische Verwendungsart von Strings eingeführt. Die Verwendung von C-Style-Strings erfordert immer eine manuelle Speicherverwaltung. Die Unicode-basierte Version von *PROXESS* soll keine C-Style-Strings verwenden[6]. Im Hauptteil wird daher die Verwendung einer Stringklasse zur einheitlichen Handhabung von Strings diskutiert (Kapitel 7.3).

Die API von *Microsoft Windows* wurde im Hinblick auf die Unterstützung und die Handhabung von Unicode(-Strings) untersucht. Seit der Version *Windows NT 6.0* wird der Unicode-Standard in der Version 5.0 unterstützt. Aus den Betrachtungen der *Windows*-API wurden einige Folgerungen für die Unicode-basierte Version von *PROXESS*

[6]Es zeigt sich allerdings, dass an den RPC-Schnittstellen weiterhin C-Style-String verwendet werden.

gezogen, die an dieser Stelle zusammengefasst werden. Diese Folgerungen berücksichtigen insbesondere Anforderung 9 (siehe Kapitel 1), „Microsoft-Richtlinien und -Standards sind für die Lösung maßgeblich.“.

- Die API von *Windows* ist im Hinblick auf die Unicode-Unterstützung aktuell und die entsprechende Dokumentation ist transparent[7]. Daher wird die Unicode-Implementierung von *Microsoft* eingesetzt. Die Open-Source-Library *icu4c*, die als de-facto-Standard für Unicode angesehen werden kann ([ICU12]), hat die einzige wirkliche Alternative zur *Windows*-API dargestellt.
- Es sollte mindestens Version *Windows NT 6.0* (d.h. *Windows Server 2008* bzw. *Windows Vista*) eingesetzt werden. Die Funktionen, die explizit Unicode-Algorithmen implementieren, scheinen in älteren Versionen von *Microsoft Windows* noch nicht enthalten zu sein.
- UTF16-LE wird zur *Windows*-internen Kodierung von Strings verwendet. Es ist naheliegend, dass die Unicode-basierte Version von *PROXESS* die selbe Kodierung für Strings verwendet.
- Die Funktionen der *Windows*-API existieren häufig in drei Varianten. Es gibt eine Variante für Windows-Codepages, eine Variante für Unicode und eine generische Variante, die in Abhängigkeit eines Präprozessor-Makros die erste oder die zweite Variante auswählt. Die Unicode-basierte Version von *PROXESS* soll die expliziten Unicode-Datentypen und -Funktionen der *Windows*-API verwenden.

Die Einführung der Technologie *Microsoft RPC* hat spezifische Notationselemente der Sprache *MIDL* (Attribute) und technische Einschränkung (POD-Objekte) erläutert, die beim Entwurf der Unicode-basierten Version der angebotenen Schnittstelle des *DatabaseManager* (Kapitel 9.1) berücksichtigt werden müssen.

Die Technologie *Microsoft COM* baut ein teilweise objekt-orientiertes Datenmodell auf *Microsoft RPC* auf. Strings in *COM*-Schnittstellen sollen nach der Spezifikation immer Unicode-Strings sein. Bei der Untersuchung der benötigten Schnittstellen des *DatabaseManager* (Kapitel 9.2) wird deutlich, dass alle vom *DatabaseManager* verwendeten *COM*-Schnittstellen spezifikationsgemäß Unicode-Strings verwenden.

[7] Die Microsoft-Dokumentation ist zwar an manchen Stellen im Hinblick auf Unicode inkorrekt, unspezifisch oder irreführend. Allerdings ist die Dokumentation von Funktionen, die Unicode-Algorithmen implementieren, sehr genau und verwendet die Unicode-Terminologie.

6. Verwandte Arbeiten

Dieses Kapitel stellt die Arbeit in einen Zusammenhang mit den Begriffen **Software-Internationalisierung und -Lokalisierung** und verdeutlicht, dass die Unicode-Migration eines Software-System lediglich eine Grundlage für die Internationalisierung von Software darstellt.

Im Anschluss werden verwandte Arbeiten vorgestellt, die sich mit den Themengebieten **projektartige Durchführung einer Unicode-Migration** und **automatische Analyse von Quelltext** befassen.

Unicode-Migration, Software-Internationalisierung und -Lokalisierung

He und Bustard geben in [HBL02] eine kompakte Definition der Begriffe Internationalisierung und Lokalisierung:

> A **locale** represents a geographic, political, or cultural region. It determines the conventions in use in that region, such as sort order, keyboard layout, date, time, number and currency formats, in addition to the language itself. [...]
>
> **Internationalisation** is the process of developing or re-engineering a program by separating the program into culturally dependent and independent elements so that it can be easily adapted, without engineering changes, to various other locales. This process produces an internationalised program.
>
> **Localisation** is the process of adapting a program version, which may be the original or internationalised, for one or more other locales. This process produces a localised prograrn. ([HBL02])

Internationalisierung umfasst im wesentlich das **Programmieren ohne das Voraussetzen einer bestimmten locale** sowie das **neu-Programmieren von locale-sensitiven Teilen des Programms**. Darunter fällt zum Beispiel das Paketieren von für den Benutzer sichtbaren Strings in sog. Resource-Bundles. Die Strings in diesen Bundles werden für

jede unterstütze Kultur bzw. Sprache einmal übersetzt. Der Prozess der Übersetzung ist Teil der Lokalisierung. Zur Lokalisierung gehören u.a. auch das Vorbereiten von Texten für eine möglichst einfache und eindeutige Übersetzung, das Anpassen der Benutzeroberfläche an kulturelle Gegebenheiten ([HHSP04]) oder das Anpassen weiterer Ressourcen, wie Bild- oder Tondateien. Die Arbeitsschritte der Lokalisierung erfordern i.d.R. gute Kenntnise über die Sprache und Kultur, an welche die Software angepasst wird, und sind keine Teilaufgaben der Softwareentwicklung im engeren Sinne. Im Gegensatz dazu stellen die Arbeitsschritte der Internationalisierung Anforderungen an das softwaretechnische Vorgehen bei der Entwicklung der Software.

Hogan verdeutlicht dies in [HHSP04]: Internationalisierung von Software muss während des gesamten Entwicklungsprozesses bedacht werden.

> Internationalization issues need to be considered from conception through to packaging and delivery, and there are many distinct aspects to be addressed. ([HHSP04])

Die in dieser Arbeit behandelte Unicode-Migration von Software-Systemen stellt die Infrastruktur dafür bereit, Texte in beliebigen Sprachen verarbeiten zu können. Durch die Verwendung von Unicode-Algorithmen zur Sortierung wird eine locale-sensitive Sortierung von Text ermöglicht. Die Handhabung von Resource-Bundles oder die Anpassung von Datums-, Zeit- oder Währungsformaten werden nicht behandelt. Im Folgenden stehen die technischen Aspekte der projektartigen Durchführung einer Unicode-Migration im Vordergrund.

Projekt-artige Durchführung einer Unicode-Migration

Mit der **projektartigen Durchführung einer Unicode-Migration** befassen sich z.B. Arbeiten von Lindenberg ([LP08]), Nestved ([Nes94]) und Peng et al. ([PYZ09]).

[LP08] betrachtet die Umstellungsproblematik mit einem Blick auf das Gesamtsystem und diskutiert Problemstellungen, die sich überwiegend auf die Umstellung und die Einhaltung der Kompatibilität von Schnittstellen beziehen. Die von Lindenberg formulierten Ideen zum prinzipiellen Vorgehen dienen als Anregung für den Entwurf eines modularisierbaren und iterativen Umstellungsprozesses in dieser Arbeit.

Die Arbeiten von Nestved und Peng et al. befassen sich mit der konkreten Migration des Source-Codes.

[Nes94] stellt ein allgemein anwendbares, mehrstufiges Vorgehen zur Migration von C-Code vor, so dass die Schnittstellen sowohl Unicode-basierte als auch Codepage-basierte Datentypen unterstützen. Nestveds Artikel dokumentiert das von *Microsoft* für die API von *Microsoft Windows* angewendete Verfahren. Die aus diesem Verfahren resultierenden Schnittstellen werden in dieser Arbeit **duale Schnittstelle** genannt. Im Rahmen der Anwendung des Umstellungsprozesses wird die angebotene Schnittstelle des *DatabaseManager*, einem Teil des Software-Systems *PROXESS*, als duale Schnittstelle entworfen.

[PYZ09] stellen einen Ansatz zur halbautomatischen Migration von Quelltext vor. Es wird ein Verfahren zur Mustererkennung auf lexikalischer Ebene vorgestellt und evaluiert. Dabei werden Muster, die nicht automatisch migriert werden können, durch das manuelle Eintragen neuer Transformationsregeln in ein Repository automatisch migrierbar gemacht. Einige grundlegende Ideen aus [PYZ09] werden in dieser Arbeit aufgegriffen, um einen Ansatz zur satischen Analyse und Transformation von Quelltext auf Basis der abstrakten Syntax zu entwickeln.

Tools zur statischen Code-Analyse

Zur statischen Analyse wird das Tool *clang* eingesetzt. *Clang* ([claa]) ist ein C++-Frontend für den Compiler *LLVM*. Beide Tools werden in Open-Source-Projekten entwickelt. *clang* besitzt einen Library-basierten Aufbau und bietet die Frontend-Funktionalität über eine Menge von C++-Modulen an. *Clang* bietet neben den typischen Komponenten eines C++-Frontend – Lexer, Präprozessor und Parser – u.a. Möglichkeiten zur Traversierung, Modifikation und zum Zurückschreiben[1] des AST. Die Online-Dokumentation [clab] gibt einen Überblick über den Aufbau der Libraries von *clang*.

Alternativen zu *clang* haben das Sofware-Reengineering-Toolkit DMS ([Sem]) von Semantic Design und das Columbus-Framework ([rBFG05]) von *FronendART* dargestellt. Die Untersuchungen in dieser Arbeit werden mit *clang* durchgeführt, da sich die anderen Alternativen als nicht verfügbar dargestellt haben: Es ist nicht möglich, DMS kostenlos zu evaluieren und die Kontaktaufnahme mit *FrontendART* ist gescheitert.

[1] Zurückschreiben in C++-Quelltext.

Teil III.

Prozessdefinition und -anwendung

Dieser Teil stellt den Hauptteil der Arbeit dar. Der Name „Prozessdefinition und -anwendung" verdeutlicht, dass der Schwerpunkt auf dem Entwurf des Umstellungsprozesses für die Unicode-Migration von *PROXESS* liegt, was der Zielsetzung 2 der Arbeit entspricht. Zielsetzung 3 der Arbeit wird als Teil der Anwendung des Umstellungsprozesses in Kapitel 10 behandelt.

Kapitel 7 definiert den Umstellungsprozess als eine Reihe von Aktivitäten, die bei der Migration durchgeführt werden müssen. Es gibt zwei Arten von Teilprozessen. Zum einen diejenigen Teilprozesse, die genau einmal für das gesamte System durchgeführt werden müssen, und zum anderen diejenigen Teilprozesse, die für jedes Element einer sinnvollen Zerlegung des Software-Systems durchgeführt werden müssen.

Kapitel 8 dokumentiert die Analyse und Umstellung des Datenschemas von *PROXESS*, das einer Menge von Schemata der relationalen Datenbank entspricht. In diesem Zusammenhang werden auch die von *PROXESS* unterstützten Datenbank-Management-Systeme im Hinblick auf Unicode-Unterstützung untersucht. Aufbauend auf den Erkenntnissen aus der Analyse des Datenschemas wird zur Unterstützung der Datenmigration ein Tool zur automatischen Erzeugung von Update-Skripts für Benutzerdatenbanken von *PROXESS* vorgestellt. Da *PROXESS*-Benutzerdatenbanken für jede Installation unterschiedliche Stringfelder enthalten können – und daher für jede Installation ein individuelle Update-Skript ausgeführt werden muss –, ist ein (teilweise) automatisiertes Verfahren zur Migration des Datenbestandes sinnvoll.

Kapitel 9 dokumentiert die Analyse und Umstellung der Schnittstellen des *DatabaseManager*, einem Serverprogramm von *PROXESS*. Die angebotene RPC-Schnittstelle des *DatabaseManager* ist repräsentativ für die global exportierte Schnittstelle des gesamten Systems. Daher wird – auch wenn die angebotene Schnittstelle des *DatabaseManager* selbst nicht exportiert wird – eine duale Schnittstelle entworfen, die sowohl Unicode-basierte als auch Windows-Codepage-basierte Clients bedienen kann. Bei der Untersuchung der verwendeten Schnittstellen des *DatabaseManager* zeigt sich, dass diese zu großen Teilen schon Unicode-basiert sind.

Kapitel 10 behandelt die Umstellung des Quelltextes des *DatabaseManager*. Es wird ein Vorgehen beschrieben, um Ausprägungen von Verwendungsmustern automatisch zu erkennen, und diese Ausprägungen durch das Anwenden einer Transformationsregel in eine semantisch äquivalente Form zu überführen, die Unicode-String verarbeitet.

7. Definition und Entwurf des Umstellungsprozesses

Dieses Kapitel führt den Begriff **Umstellungsprozess** ein, und präsentiert einen modularisierbaren und iterativen Umstellungsprozess, der konkret auf das Software-System *PROXESS* angewendet wird.

In Abschnitt 7.1 werden die Begriffe (Unicode-)Umstellung, Migration und Umstellungsprozess definiert und in einen Zusammenhang gestellt.

In Abschnitt 7.2 wird ein allgemein verwendbarer Umstellungsprozess entworfen, der die Migration in eine Reihe von Aktivitäten mit einfachen Abhängigkeiten zerlegt.

In Abschnitt 7.3 wird die Anwendung des Umstellungsprozesses auf das Software-System *PROXESS* konkretisiert.

7.1. Definitionen

Ein **Software-Modul** ist ein identifizierbarer Teil eines Software-Systems. Der Begriff kann je nach Zusammenhang das gesamte System oder eine zusammenhängende Teilmenge einer (hierarchischen) Zerlegung des Systems bezeichnen. Eine solche Zerlegung muss sinnvoll sein, d.h. jedes Modul einer Zerlegung muss eine eigene Funktionalität besitzen und eine Schnittstelle anbieten, über welche die Funktionalität anderen Modulen zur Verfügung gestellt wird. Ein Beispiel ist die Zerlegung von *PROXESS* in die Menge aller Client-Programme und das Sub-System *PROXESS Server*. *PROXESS Server* kann weiter in aufgeteilt werden in die drei Serverprogramme (*DocumentManager*, *StorageManager* und *DatabaseManager*) und die zugehörigen Konfigurationsprogramme.

Eine **angebotene Schnittstelle** (**provided interface**) eines Software-Moduls `S` ist eine Schnittstelle, die von anderen Software-Modulen zur Kommunikation mit `S` verwendet

werden kann. Zum Beispiel sind die RPC-Schnittstellen der drei Serverprogramme von *PROXESS* angebotene Schnittstellen.

Eine **exportierte Schnittstelle** ist eine angebotene Schnittstelle, die über die Systemgrenze hinaus sichtbar ist.

Verwendet ein Software-Modul `S` die angebotene Schnittstelle eines Software-Moduls `T`, dann heißt diese Schnittstelle aus der Sicht von `S` **benötigte Schnittstelle** (**required interface**).

Ein Software-Modul, das mehrere Sub-Module zusammenfasst, wird als **zusammengesetztes Modul** bezeichnet. Ein zusammengesetztes Modul muss nicht zwangsläufig über eine eigene Implementierung verfügen. Die Komponente *PROXESS Server* ist z.B. ein zusammengesetztes Modul: Es beinhaltet neben einer Reihe von Konfigurationsprogrammen die Programme *DocumentManager*, *StorageManager* und *DatabaseManager*. *PROXESS Server* besitzt keine eigene Implementierung: Der Programm-Code teilt sich vollständig auf die Sub-Module auf.

Ein **externes Software-Modul** ist ein Software-Modul, dass von Fremdanbietern stammt, z.B. die APIs von *Microsoft Windows*, *log4net* sowie die *Java Virutal Machine*.

Unter einer **Unicode-Umstellung** wird eine bestimmte **Zustandsänderung** eines Software-Moduls `S` verstanden. Im **Vorzustand** verwendet `S` zur Kodierung von Strings überwiegend einen nicht-Unicode Zeichensatz. Damit ist gemeint, dass an einigen Stellen durchaus schon Unicode verwendet werden kann. Im **Nachzustand** verwendet `S` zur Kodierung von allen Strings den Unicode Zeichensatz. Davon sind Strings ausgeschlossen, die an Schnittstellen zu Modulen verwendet werden, die Unicode nicht unterstützen. Durch die Änderung der Kodierung eines Strings darf dessen Semantik nicht verändert werden. Alle übrigen Datenrepräsentationen sind im Vorzustand und Nachzustand identisch.

Wird im Folgenden von einer **Umstellung** gesprochen, ist damit eine Unicode-Umstellung gemeint.

Mit dem Begriff **Migration** wird die projektartige Durchführung einer Umstellung bezeichnet.

Ein **Umstellungsprozess** definiert die notwendigen Arbeitsschritte für die Migration eines Software-Systems und legt die zeitliche Reihenfolge fest, in der diese Arbeitschritte durchgeführt werden müssen.

7.2. Entwurf des Umstellungsprozesses

Dieser Abschnitt entwirft einen Umstellungsprozess, der die Migration eines Software-Systems in einzelne Teilprozesse zerlegt. Dieser Umstellungsprozess wird exemplarisch auf einen Teil des Software-Systems *PROXESS* angewendet.

[LP08] legt die Zerlgeung der Migration in Teilprojekte für nicht-triviale Anwendungen nahe und führt die wesentlichen Arbeitsschritte auf. Die von [LP08] vorgestellten Arbeitsschritte dienen als Ausgangspunkt für den Entwurf des Umstellungsprozesses.

Durch die Einführung eines Teilprozesses für die Migration eines (evtl. zusammengesetzen) Software-Moduls wird der Prozess **modularisierbar**. Dieser Teilprozess wird auf jedes Software-Modul **genau einmal** angewendet, und gibt dem Umstellungsprozess einen **iterativen** Charakter: Während der Anwendung des Umstellungsprozesses wird der Teilprozess für die Migration eines Software-Moduls solange auf einen spezifischen Teil des Systems angewendet, bis das gesamte System migriert ist. Mehrere Iterationen des Teilprozesses können zu großen Teilen **parallel** oder **zeitlich unabhängig** voneinander ausgeführt werden.

Der gesamte **Ablauf des Umstellungprozesses** wird in Abbildung 7.1 als UML Aktivitätsdiagramm dargestellt. Die Teilprozesse werden als Aktivitäten dargestellt. Die Teilprozesse gliedern sich in zwei Kategorien – globale und lokale Teilprozesse:

1. Ein **globaler Teilprozess** muss für das gesamte System nur einmal ausgeführt werden.

2. Ein **lokaler Teilprozess** muss für jedes Modul einer sinnvollen Zerlegung des Software-Systems durchgeführt werden. Die Umstellung eines einzelnen Moduls kann bei Bedarf ebenfalls durch eine feinere Zerlegung in Sub-Module realisiert werden.

Der Ablauf des Prozesses gliedert sich in drei Phasen: **Migration**, **Systemtest** und **Installation**. Zunächst werden die einzelnen Teilprozesse diesen Phasen zugeordnet. Die Teilprozesse der Phasen Systemtest und Installation werden direkt an dieser Stelle näher betrachtet. Eine detaillierte Betrachtung der Teilprozesse der Phase Migration wird in den anschließenden Abschnitten betrachtet. An die Zuordnung der Teilprozesse zu den Phasen des Umstellungsprozessen schließt sich eine Überlegungen an, wie die Entfernung von alten, nicht Unicode-basierten Versionen von angebotenen Schnittstellen handzuhaben ist.

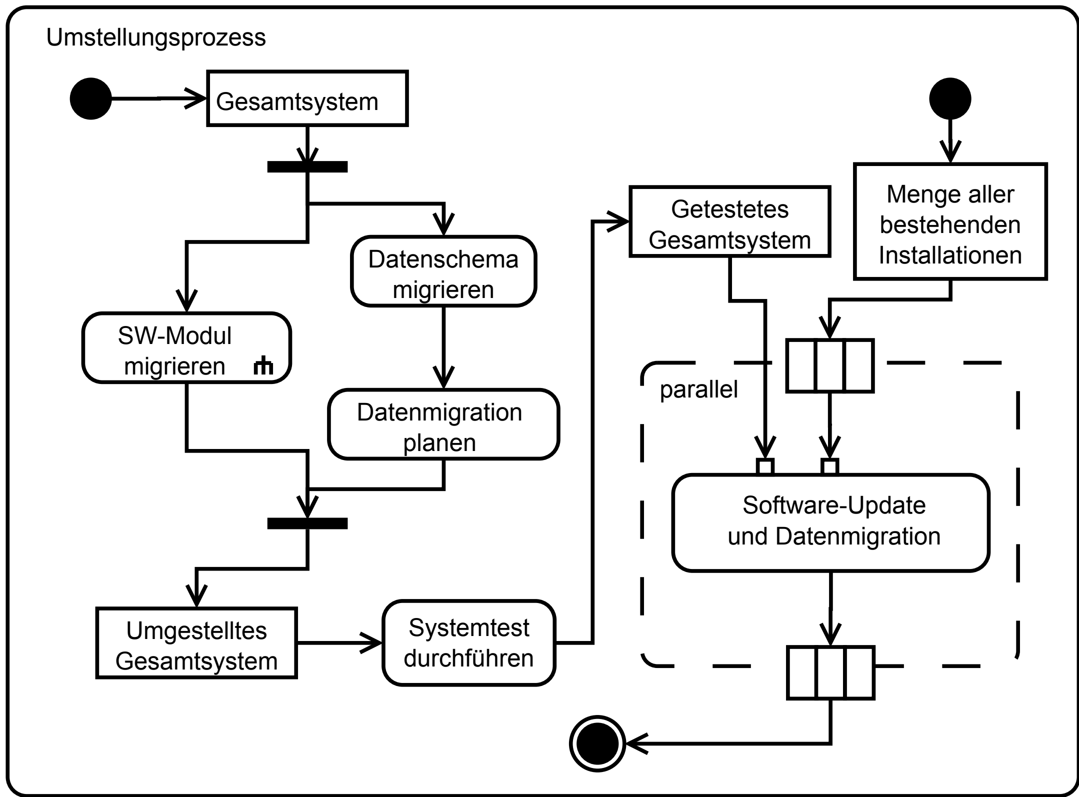

Abbildung 7.1.: Ablauf des Umstellungsprozesses

Umstellungsphase: Migration

Zu der Phase **Migration** gehören die Teilprozesse `SW-Modul migrieren`, `Datenschema migrieren` und `Datenmigration planen`. In dieser Phase wird eine Unicode-basierte Version der Software entwickelt. Die Teilprozesse `Datenschema migrieren` und `Datenmigration planen` sind globale Teilprozesse, die in den Abschnitten 7.2.1 bzw. 7.2.2 näher beschrieben werden. Der Teilprozess `SW-Modul migrieren` ist ein lokaler Teilprozess. Das Gesamtsystem wird als zusammengesetztes Software-Modul aufgefasst und – aus Sicht des gesamten Umstellungsprozesses – in einem Schritt migriert. Dieser Teilprozess wird in Abschnitt 7.2.3 genauer betrachtet.

Umstellungsphase: Systemtest

Die Phase **Systemtest** besteht aus dem globalen Teilprozess `Systemtest durchführen`. Ein Systemtest besteht aus Testfällen, die die Gesamtfunktionalität des Systems anhand der Anforderungen testen. Der Systemtest als Teil des Umstellungsprozesses glidert sich in zwei Teile:

1. **Regressionstest** – Der Systemtest des alten Systems wird für die umgestelle Unicode-Version des Systems ohne Veränderungen durchgeführt. Die Testausführungen müssen für beide Versionen die gleichen Ergebnisse liefern. Damit wird sichergestellt, dass die Migration das geforderte Systemverhalten nicht verändert hat.

2. **Erweiterung um spezielle Unicode-Testfälle** – Der Systemtest wird um Testfälle erweitert, die zusätzlich die richtige Handhabung von Unicode-Strings testen. Dabei soll sichergestellt werden, dass sich das System konform zu allen benötigten Aspekte des Unicode-Standards verhält.

Umstellungsphase: Installation

Die Phase **Installation** besteht aus dem globalen Teilprozess `Software-Update und Datenmigration`. Dieser Teilprozess wird für alle Installationen des Systems angewendet. Er besteht aus einem Update der Software auf die Unicode-basierte Version und der Anwendung der Datenmigration, die im Teilprozess `Datenmigration planen` erarbeitet wurde.

Enterfernen der alten Schnittstellen

Das **Entfernen der alten Schnittstellen** ist im Umstellungsprozess nicht enthalten. Ob, und wann die alten Schnittstellen entfernt werden hängt davon ab, wie schnell die neuen Unicode-Schnittstellen vollständig in das Systemumfeld integriert werden. Das gilt insbesondere für Systeme mit offenen Schnittstellen (wie z.B. *PROXESS*), bei denen die Integration der neuen Schnittstellen von externen Personen oder Organisationen abhängt. Wie lange die Kompatibilität zu den alten Schnittstellen erhalten bleiben muss oder soll, kann zu diesem Zeitpunkt nicht vorausgesagt werden. In der folgenden Auflistung werden einige Feststellungen zu dieser Thematik gemacht.

1. **Exportierte Schnittstellen** – Für exportierte Schnittstellen muss u.U. für einen langen Zeitraum die alte, nicht Unicode-basierte Version der Schnittstelle bereitgestellt werden. Die Schnittstelle kann erst dann entfernt werden, wenn sichergestellt ist, dass die alte Schnittstelle von keiner externen Software verwendet wird. Das Auslaufen der Unterstützung dieser Schnittstelle muss eine ausreichend lange Zeit vorher terminiert werden (z.B. zwei oder drei Major-Releases), und alle Anwender des Systems müssen von dieser Änderung rechtzeitig in Kenntnis gesetzt werden.
   Ggf. kann die alte Schnittstelle gar nicht entfernt werden, wenn es von der Schnittstelle abhängige externe Programme gibt, die nicht an die neue Schnittstelle angepasst werden können (z.B. weil der Anbieter die Wartung für das Programm eingestellt hat; oder weil der Quelltext eines Programms nicht (mehr) verfügbar ist).

2. **Angebotene, nicht-exportierte Schnittstellen** – Nicht-Unicode-basierte Versionen von angebotenen Schnittstellen, die nur system-intern verwendet werden, können entfernt werden, sobald alle Module, die diese Schnittstelle benötigen, umgestellt worden sind. Der richtige Zeitpunkt, zu dem eine nicht Unicode-basierte Version einer angebotenen Schnittstelle entfernt wird, ergibt sich implizit aus dem Umstellungsprozess. Im Gegensatz zu den exportierten Schnittstellen muss das Entfernen der nicht-exportierten Schnittstellen nicht „nach außen" mitgeteilt werden.

In den folgenden Abschnitten werden die Teilprozesse der Phase Migration des Umstellungsprozesses detaillierter betrachtet.

7.2.1. Datenschema migrieren

Das Datenschema muss analysiert werden um festzustellen, welche Teile von einer Umstellung betroffen sind. Dabei sind vor allem zwei Fragen zu klären:

1. Werden alle Schema-Elemente, die als String-Elemente deklariert sind, auch dazu verwendet, Strings zu speichern?
2. Werden Strings in solchen Schema-Elementen gespeichert, die nicht als String-Element deklariert sind?

Die richtige Interpretation der Daten erfordert detaillierte Kenntnisse über das System. Es ist möglich, dass Daten, die wie ein String aussehen, nicht als String interpretiert werden dürfen (z.B. Dateidaten, die ASCII-kodiert sind und das auch bleiben müssen), oder dass Daten, die zunächst nicht nach einem String aussehen, doch ein String sind (oder einen String enthalten) und damit von der Umstellung betroffen sind.

Dieser Teilprozess wird für das Datenschema von *PROXESS* durchgeführt. Die Durchführung wird in Abschnitt 7.3 skizziert, und die Ergebnisse der Durchführung werden in Kapitel 8 dokumentiert. Dieser Arbeitsschritt wird mit den Datenbankschemata für diejenige Variante von *PROXESS* durchgeführt, die *Microsoft SQL Server* als Back-End verwendet. Die Umstellung für die anderen Datenbanksystem wird analog durchgeführt.

7.2.2. Datenmigration planen

Sobald das neue Datenschema bekannt ist, kann ein Vorgehen zur Migration des Datenbestandes entwickelt werden. Ziel der Datenmigration ist die verlustfreie Überführung aller Daten in das geänderte Datenschema. Da die Datenmigration für jede Systeminstallation durchgeführt werden muss, soll bei der Bearbeitung dieses Teilprozesses ein weitgehend automatisiertes Verfahren zur Datenmigration entstehen.

Dieser Teilprozess wird exemplarisch für diejenige Variante von *PROXESS* durchgeführt, die das Datenbanksystem *Microsoft SQL Server* als Back-End verwendet. Die Durchführung wird in Abschnitt 7.3 skizziert, und die Ergebnisse der Durchführung werden in Kapitel 8 dokumentiert.

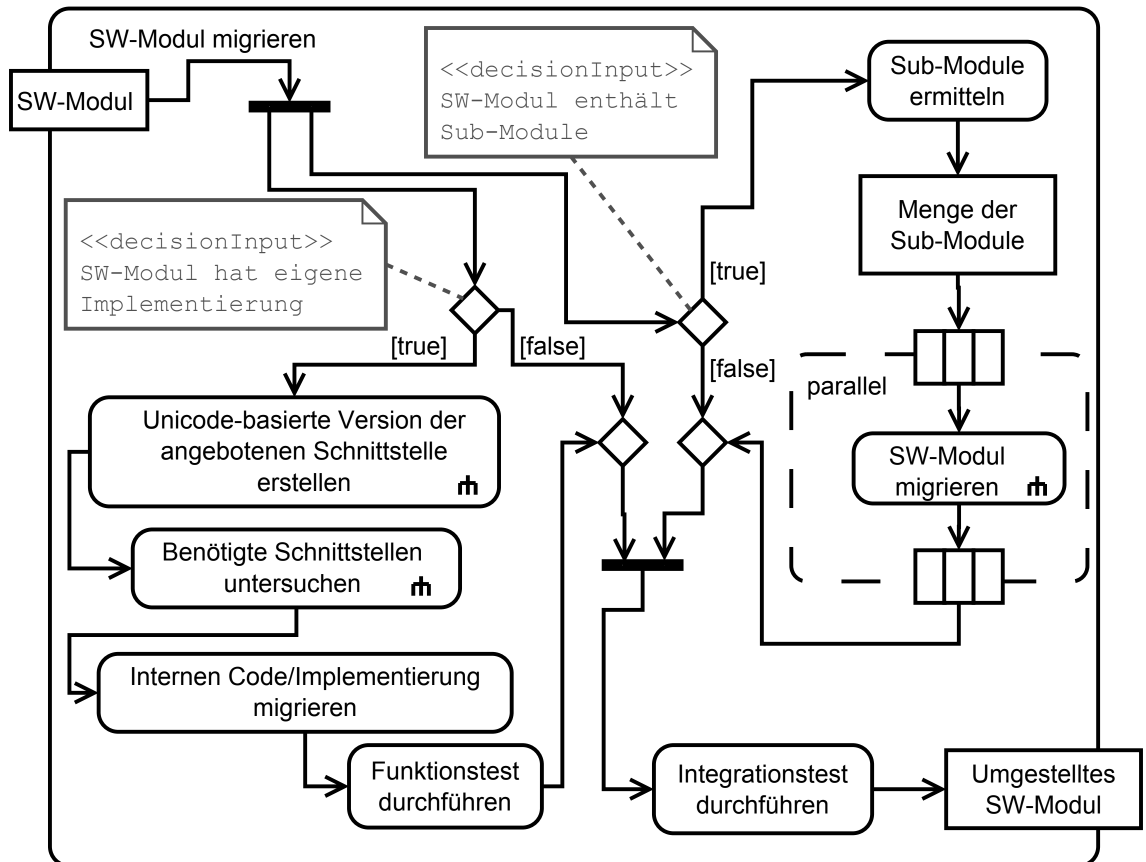

Abbildung 7.2.: Teilprozess für ein Software-Modul

7.2.3. Software-Modul migrieren

Der Teilprozess `SW-Modul migrieren` wird für jedes Modul einer sinnvollen Zerlegung des Software-Systems genau einmal durchgeführt. Abbildung 7.2 zeigt den Ablauf dieses Teilprozesses als Aktivitätsdiagramm, der sich in zwei Teile gliedert:

1. Handelt es sich bei dem Software-Modul um ein **zusammengesetztes Modul** (`SW-Modul enthält Sub-Module`), dann wird die Aktivität `SW-Modul migrieren` rekursiv für alle Elemente der Menge von Sub-Modulen aufgerufen. Der rekursive Aufruf spiegelt die kompositionale Struktur von Software-Modulen wieder. Die Migrationsschritte für die Sub-Module müssen nicht in der durch die rekursiven Aufrufe entstehenden Reihenfolge abgearbeitet werden.

2. Besitzt das Software-Modul eine **eigene Implementierung** (`SW-Modul hat eigene Implementierung`) wird zuerst eine Unicode-basierte Version der angebotenen Schnittstelle erstellt. Dieser Teilprozess wird in Abschnitt 7.2.3 näher betrachtet. Anschließend werden die benötigten Schnittstellen des Software-Moduls untersucht. Dieser Teilprozess wird in Abschnitt 7.4 näher betrachtet. Die Arbeitsschritte der Migration des internen Codes werden im Umstellungsprozess nicht in Form von Aktivitäten konkretisiert. An dieser Stelle muss eine Transformations-Strategie gewählt werden. Dieser Aspekt des Umstellungsprozesses wird in Abschnitt 7.2.3 weiter behandelt. Die Migration der Implementierung eines Software-Moduls wird durch einen Funktionstest abgeschlossen, dessen Eigenschaften in Abschnitt 7.2.3 weiter behandelt werden.

In beiden Fällen wird als letzter Schritt ein **Integrationstest** durchgeführt, der das richtige Zusammenspiel der Schnittstellen zu anderen Modulen sicherstellen soll. Für den Integrationstest liegt der Fokus vor allem auf Schnittstellen-Funktionen mit Stringparametern oder -rückgabewerten.

Der Teilprozess `SW-Modul migrieren` wird für den *DatabaseManager* durchgeführt. Die Durchführung wird in Abschnitt 7.3 skizziert, und die Ergebnisse der Durchführung werden in den Kapitel 9 und 10 dokumentiert.

Bereitstellen von Unicode-basierten Versionen der angebotenen Schnittstellen

Bei der Migration eines Software-Moduls `S` sollen bereits in einer frühen Phase der Migration Unicode-basierte Versionen der **angebotenen Schnittstelle**(n) bereitgestellt werden. Das erleichtert es, die Migration mehrerer Software-Module zu parallelisieren:

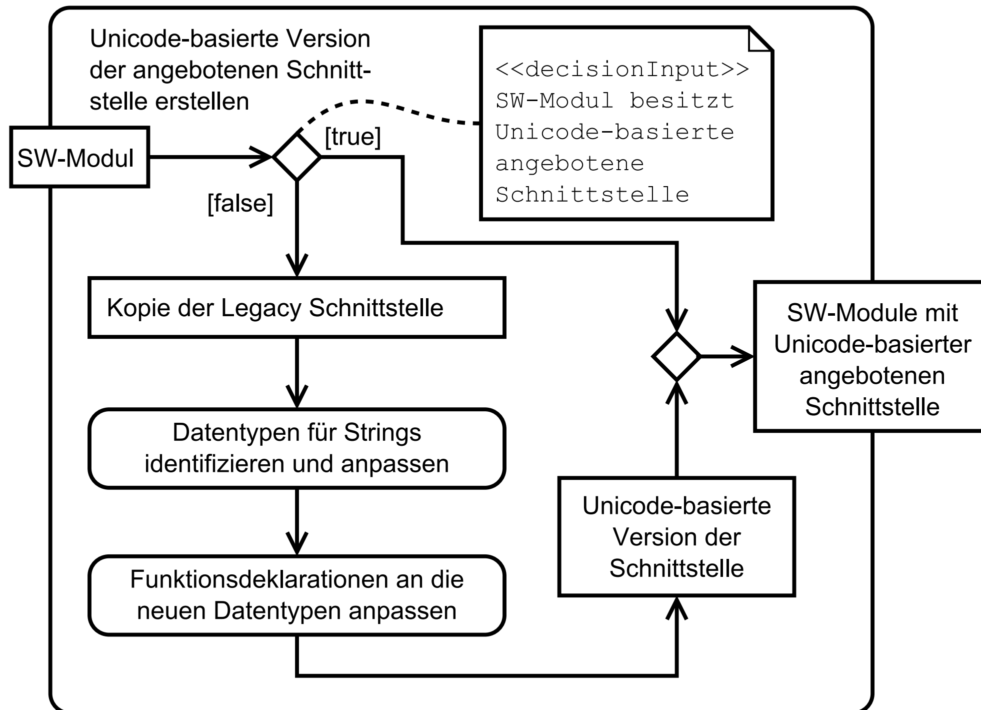

Abbildung 7.3.: Teilprozess für das Erstellen einer Unicode-basierten Version der angebotenen Schnittstelle

Während ein Entwickler die weitere Migration von `S` durchführt, kann ein anderer Entwickler bereits mit der Migration eines Moduls beginnen, das die externe Schnittstelle von `S` benötigt.

Abbildung 7.2.3 zeigt die Arbeitsschritte für das erstellen einer Unicode-basierten Version der angebotenen Schnittstelle. Es wird eine Kopie der alten Schnittstelle (`Legacy Schnittstelle`) verwendet, weil davon ausgegangen wird, dass die alte Schnittstelle für eine gewisse Zeit weiter verwendet wird.

Benötigte Schnittstellen untersuchen

Dieser Abschnitt zeigt die notwendigen Arbeitsschritte bei der Untersuchung der **benötigten Schnittstellen**. Abbildung 7.4 stellt die Arbeitsschritte als Aktivitätsdiagramm dar. Die Aktivitäten werden im folgenden Fließtext textuell beschrieben.

Es seien `S` und `T` Software-Module. Die benötigten Schnittstellen von `S` werden auf die Verfügbarkeit von Unicode-basierten Versionen untersucht. Für jede benötigte

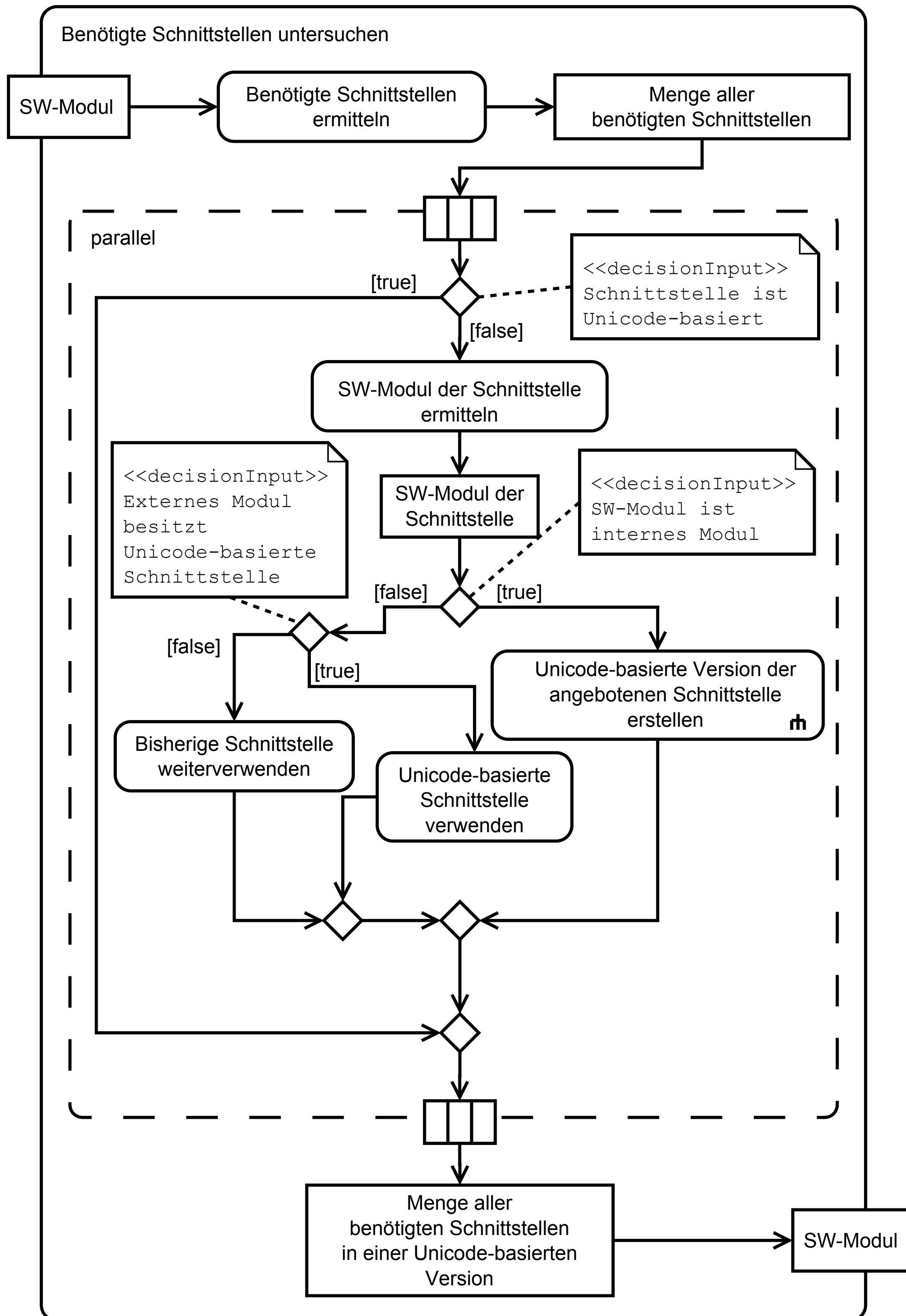

Abbildung 7.4.: Teilprozess für die Untersuchung der benötigten Schnittstellen

Schnittstelle von `S` gibt es ein Software-Modul `T`, das diese Schnittstelle anbietet. Dabei ist zu unterscheiden, ob es sich bei `T` um ein internes Modul handelt, das Teil des zu migrierenden Systems ist, oder ob es sich um ein externes Modul handelt, das von Drittanbietern bereitgestellt wird.

1. Handelt es sich bei `T` um ein **internes Modul** muss überprüft werden, ob bereits eine Unicode-basierte Version der angebotenen Schnittstelle erstellt worden ist. Ist dies nicht der Fall, wird eine solche Schnittstelle bereitgestellt. Das heißt, dass der Arbeitsschritt `Bereitstellen von Unicode-basierten Versionen der externen Schnittstellen` für das Modul `T` durchgeführt wird, wenn dies nicht schon vorher geschehen ist.

2. Handelt es sich bei `T` um ein **externes Modul** muss festgestellt werden, ob die angebotene Schnittstelle bereits Unicode-basiert ist. Ist dies der Fall, kann die Schnittstelle unverändert beibehalten werden. Ist die Schnittstelle nicht Unicode-basiert, muss überprüft werden, ob es überhaupt eine Unicode-basierte Version dieser Schnittstelle gibt. Gibt es eine Unicode-basierte Version der Schnittstelle, muss `S` im Laufe der Migration so verändert werden, dass es die Unicode-basierte angebotene Schnittstelle von `T` verwendet. Gibt es keine Unicode-basierte Version der angebotenen Schnittstelle von `T`, bleibt keine andere Wahl, als die ASCII- oder Codepage-basierte Version der angebotenen Schnittstelle von `T` beizubehalten.

Die benötigten Schnittstellen von `S` sollen also möglichst früh an Unicode-basierte Versionen angepasst werden. Insbesondere wenn `T` ein internes Modul ist, soll gelten, dass `S` direkt die Unicode-basierte Version der angebotenen Schnittstelle von `T` verwendet. So kann `S` „im ersten Anlauf" in die finale Form gebracht werden; es müssen also keine nachträglichen Anpassungen gemacht werden, wenn die angebotene Schnittstelle von `T` erst zu einem späteren Zeitpunkt auf eine Unicode-basierte Version umgestellt wird.

Es sei angemerkt, dass das Bereitstellen der Unicode-basierten Version der angebotenen Schnittstelle von `T` nur der erste Schritt der Migration von `T` ist. Für `T` selbst muss eine Iteration des Umstellungsprozesses ebenfalls durchlaufen werden. D.h., dass auch die von `T` benötigten Schnittstellen analysiert und ggf. angepasst werden müssen, und anschließend der interne Code migriert werden muss. Dies geschieht allerdings zeitlich unabhängig von der Migration von `S`.

Implementierung migrieren

Die Migration des internen Codes wurde im Umstellungsprozess bisher als ein einzelner Arbeitsschritt dargestellt, der nicht näher spezifiziert wurde. Der Umstellungsprozess legt fest, wie ein in (Sub-)Module zerlegtes Software-System an den Schnittstellen zwischen den Modulen migriert werden soll. Ab einer gewissen Granularität ist eine weitere Zerlegung nicht mehr sinnvoll. An dieser Stelle muss eine Transformation `TRANS` auf das Software-Modul `S` angewendet werden, sodass das resultierende Software-Modul `S'=TRANS(S)` Unicode-fähig ist. D.h. `S'` empfängt an allen angebotenen Schnittstellen Unicode-Strings, verarbeitet diese Strings intern ggf. konform zum Unicode-Standard und gibt an den benötigten Schnittstellen Unicode-Strings an die entsprechenden Module weiter (das gilt analog für die umgekehrte Datenflussrichtung).

Der **einfachste Ansatz** ist die schrittweise Bearbeitung von jeder Datei, jeder Zeile und jeder Anweisung des Software-Moduls. Bei diesem Ansatz besteht die Transformation darin, den Datentyp von allen Strings an die neue Kodierung anzupassen und alle Aufrufe von String-Funktionen durch Unicode-basierte Varianten zu ersetzen. Bei diesem Ansatz handelt es sich um eine vollständig manuelle Migration. Ein deartiges Vorgehen demonstrieret Nestved für C-Programme [Nes94].

Um die Migration des internen Code zu unterstützen, werden in dieser Arbeit **Transformationsregeln für Verwendungsmuster** eingeführt. Eine Transformationsregel beschreibt die konkreten Umwandlungsschritte für typische syntaktische Ausprägungen von bestimmten Verwendungsformen von Strings. Durch das Anwenden der Transformationsregel wird sichergestellt, dass Strings

1. konform zum Unicode-Standard und

2. einheitlich

gehandhabt werden. In Kapitel 10 werden die Begriffe Verwendungsmuster und Transformationsregel genau eingeführt und der Teilprozess `Internen Code/Implementierung umstellen` des Umstellungsprozesses wird dementsprechend verfeinert. Die Vorgaben zur einheitlichen Handhabung von Strings werden in diesem Kapitel in Abschnitt 7.3.2 gegeben.

Funktionstest durchführen

Dieser Abschnitt beschreibt, wie der **Funktionstest** für ein migriertes Software-Modul durchgeführt werden soll. Prinzipiell ist der Ablauf des Funktionstests identisch mit dem für den Systemtest beschriebenen Ablauf. Der Unterschied besteht darin, dass existierende Unit-Tests für einzelne Bausteine des Software-Moduls vorausgesetzt werden – anstatt Testfälle, die sich an den Systemanforderungen orientieren.

Der Testablauf gliedert sich ebenfalls in zwei Phasen:

1. **Regressionstest** – Der bestehende Unit-Test wird ohne Änderungen mit dem migrierten Software-Modul durchgeführt. Die Testausführung muss für die alte nicht-Unicode-basierte Version, und für die neue Unicode-basierte Version des Software-Moduls identische Ergebnisse liefern.
2. **Erweiterung um spezielle Unicode-Testfälle** – Es werden neue Testfälle hinzugefügt, die String-verarbeitende Bausteine mit verschiedenen Unicode-Strings testen. Bei Unicode-Strings muss im schlimmsten Fall von einer gänzlich unterschiedlichen Anzahl an Bytes, Codeunits, Codepoints und darstellbaren Zeichen (Combining Characters) ausgegangen werden. Daher muss vor allem sichergestellt werden, dass Größenangaben der Spezifikation entsprechen.

Der Umstellungsprozess wurde in den wesentlichen Punkten detailliert beschrieben. Der folgende Abschnitt skizziert die Anwendung des Umstellungsprozesses auf das Software-System *PROXESS*.

7.3. Anwendung des Umstellungsprozesses an *PROXESS*

Dieser Abschnitt beschreibt, wie der Umstellungsprozess in dieser Arbeit auf das Software-System *PROXESS* angewendet wird.

In Bezug auf die Anwendung der Teilprozesse, die sich auf das Testen der migrierten Software(-Module) beziehen, müssen deutliche Einschränkungen gemacht werden, die in Abschnitt 7.3.1 aufgeführt werden.

In Abschnitt 7.3.2 wird ein Überblick über die Anwendung des Umstellungsprozesses auf das Software-System *PROXESS* gegeben, und es werden diejenigen Teilprozesse aufgeführt, welche in dieser Arbeit exemplarisch durchgeführt werden.

In Abschnitt 7.3.3 werden Festlegung zur einheitlichen Handhabung von String getroffen.

7.3.1. Einschränkungen bei der Testbarkeit

Der Umstellungsprozess beinhaltet Teilprozesse für das Testen von umgestellten Modulen, welche die korrekte Funktionsweise der Software sicherstellen sollen. Das Sicherstellen der richtigen Funktionsweise kann nur durch einen Regressionstest erfolgen, d.h. die selben Testfälle werden vor und nach der Migration durchgeführt, und beide Testausführungen müssen zu den gleichen Ergebnissen führen. Für einen Regressionstest müssen bereits vor der Migration rekonstruierbare Tesfälle existieren.

Für *PROXESS* gibt es id.R.[1] keine Unit-Tests und auch keine andersartig rekonstruierbaren Testfälle, die gezielt einzelne Aspekte der Funktionsweise des Systems überprüfen. Deshalb stellt sich ein (automatisierter) Regressionstest im jetzigen Zustand als nicht realisierbar dar.

Damit Testfälle erstellt werden können, müssen

- die Komponenten identifiziert werden, die getestet werden sollen,
- die Testbarkeit jeder Komponente sichergestellt werden und
- Anwendungsfälle mit Szenarien erstellt werden, die von einer Komponente realisiert werden sollen.

Diese Aufgaben gehen über das Thema dieser Arbeit hinaus, und werden daher nicht bearbeitet. Die richtige Funktionsweise kann nur durch manuelle Integrations- und Systemtests validiert werden. Diese Tests können durch eine schrittweise Integration von umgestellten Modulen in das nicht-umgestelle Gesamtsystem nach jeder Iteration des Umstellungsprozesses durchgeführt werden.

Die schrittweise Integration von umgestellten Modulen stellt eine Anforderungen an die angebotene Schnittstelle des Moduls, die im wesentlichen Anforderung 3 an die Arbeit[2] (siehe Kapitel 1) entspricht. Bei der schrittweisen Integration bezieht sich diese Anforderung auf alle angebotenen Schnittstellen, und nicht nur auf die exportierte(n) Schnittstelle(n).

In Kapitel 9 wird ein duales Interface entworfen, das alle String-Funktionen und -Datentypen sowohl in einer Codepage-basierten Variante als auch in einer Unicode-basierten Variante anbietet.

[1] Zumindest für die zentralen Komponenten gibt es keine Unit-Tests.

[2] „Die alte „ANSI“-Schnittstelle von *PROXESS* muss für eine bestimmte Zeit parallel zur neuen Unicode-Schnittstelle betrieben werden.“

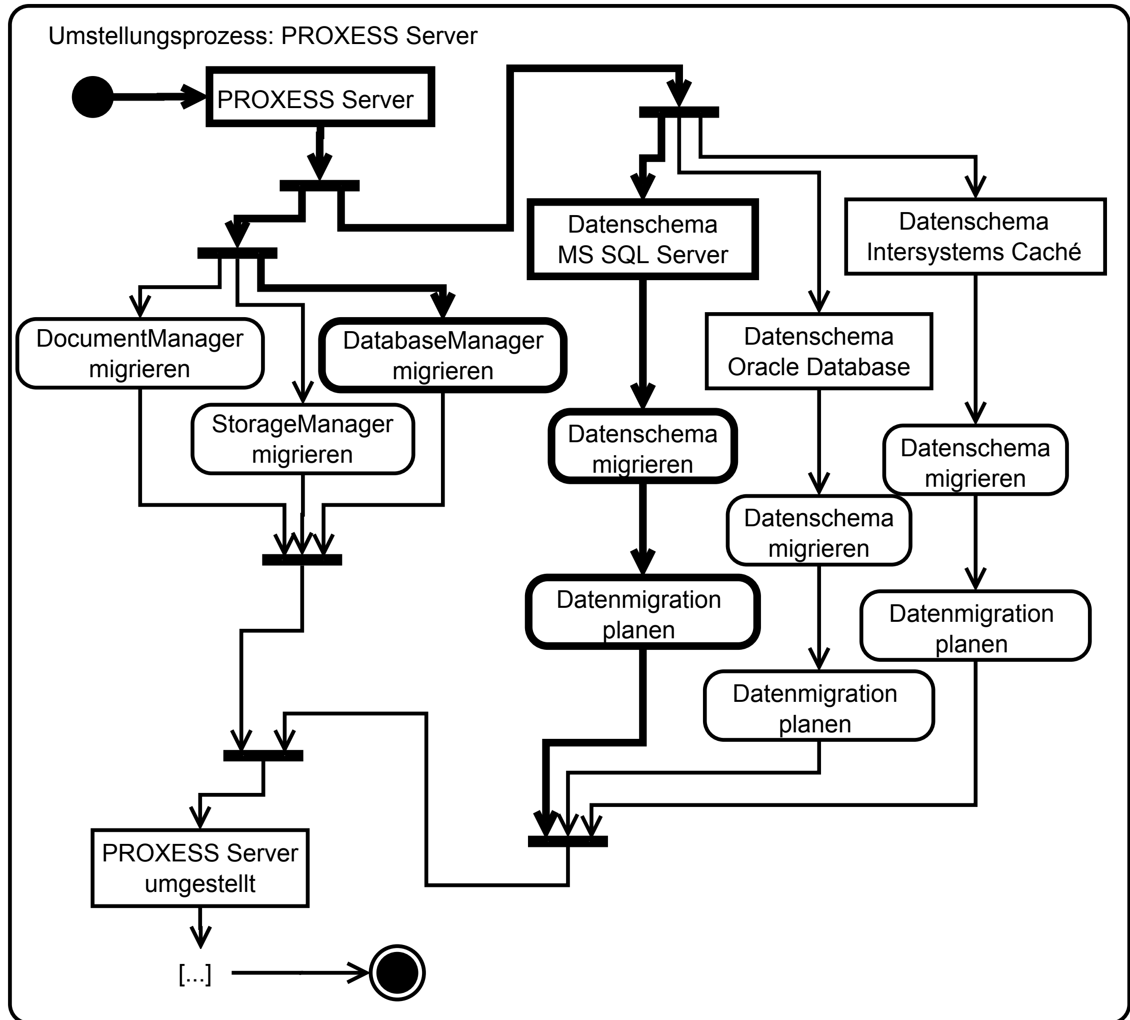

Abbildung 7.5.: Übersicht Migrationsschritte *PROXESS Server*

7.3.2. Angewendete Teilprozesse

Dieser Abschnitt beschreibt, welche Teilprozesse des Umstellungsprozesses auf das Software-System *PROXESS* angewendet werden. Abbildung 7.5 stellt die Teilprozesse für das Sub-System *PROXESS Server*, auf das sich diese Arbeit beschränkt, als Aktivitätsdiagramm dar. Es wird also nur ein Teil einer Zerlegung des gesamten Systems betrachtet: Die Client-Programme (inkl. der Konfigurationsprogramme, die auch zu *PROXESS Server* gehören) und die verschiedenen APIs für *PROXESS* werden nicht dargestellt.

Diejenigen Teilprozesse des Umstellungsprozesses, die in dieser Arbeit bearbeitet wer-

den, sind durch eine größere Linienstärke hervorgehoben. Die konkrete Ausprägung wird im Folgenden für jeden dieser Teilprozesse aufgeführt.

1 Datenschema migrieren

Die Migration des Datenschemas (Teilprozess `Datenschema migrieren`) ist notwendig, damit

1. bekannt ist, welche Daten tatsächlich als Unicode-Strings gespeichert werden müssen, und
2. damit das System direkt mit einem Datenbanksystem, das Unicode-basierte Datentypen verwendet, getestet werden kann.

Die Analyse und Umstellung des Datenschemas bedeutet für *PROXESS*, die Schemata der relationalen Datenbanken zu analysieren, die von *PROXESS* verwendet werden. Die Untersuchung wird exemplarisch für diejenige Variante von *PROXESS* verwendet, die das Datenbanksystem *Microsoft SQL Server* als Back-End verwendet. Die Umstellung der Datenbankschemata lässt sich analog auf die anderen beiden Datenbanksysteme übertragen. Dabei müssen ggf. lediglich die Namen für Datentypen angepasst werden.

Für die Migration des *DatabaseManager* (siehe Schritt 3 der Anwendung des Umstellungsprozesses) ist diese Analyse und Umstellung des Datenschemas wichtig, damit bekannt ist, welche neuen Datentypen an der Schnittstelle zum Datenbanksystem zu erwarten sind. So kann bei der Migration des *DatabaseManager* die Schnittstelle zum Datenbanksystem direkt mit den neuen Datentypen angesprochen werden.

Die Ergebnisse der Analyse werden zusammen mit einer Erörterung der Umstellungsalternativen und den entsprechenden Entscheidungen in Kapitel 8 präsentiert.

2 Datenmigration planen

Die Migration von Bestandsdaten soll nach Möglichkeit mit möglichst wenig manuellen Eingriffen erfolgen. Es wird untersucht, in wie weit sich dieser Prozess automatisieren lässt.

Ein mögliches Vorgehen zur Migration des Datenbestandes wird ebenfalls am Beispiel des Datenbanksystems *Microsoft SQL Server* in Kapitel 8 vorgestellt.

Für alle *PROXESS*-Installationen werden im wesentlichen die selben Migrationsschritte durchgeführt: Für die meisten Tabellen werden einmalig Migrations-Skripts erstellt, die für alle Datenbestände wiederverwendet werden können.

Eine Ausnahme stellen die Tabellen für Dokumente der individuellen Benutzerdatenbanken dar, da in diesen Tabellen beliebig viele, individuell benannte String-Felder existieren können. Es wird ein Tool präsentiert, das ein Migrations-Skript für diese individuellen Tabellen automatisch erstellt.

3 Software-Modul migrieren: *DatabaseManager*

Die Migration eines Software-Moduls wird exemplarisch am *DatabaseManager* durchgeführt.

Die Wahl des *DatabaseManager* ist zum einen dadurch begründet, dass er das innerste Modul des Systems ist. Er ist das einzige Modul, das die Datenbanken direkt verwendet. Der *DatabaseManager* stellt alle Daten in einer einheitlichen Form dar, die systemweit gültig ist. Das hat zur Folge, dass die Datentypen im Interface des *DatabaseManager* – zumindest in Bezug auf Strings – identisch mit den Datentypen der primären exportierten Schnittstelle des Systems[3] sind.

Zum anderen muss der *DatabaseManager* z.Zt. nicht weiterentwickelt werden. Der *DocumentManager*, der die Business-Logik des Systems implementiert und dadurch sicherlich repräsentativer als der *DatabaseManager* ist, wird von zwei Entwicklern in zwei unterschiedlichen elementaren Bereichen weiterentwickelt, während diese Arbeit geschrieben wird.

Die Migration des *DatabaseManager* gliedert sich nach dem Umstellungsprozess in drei Teile:

3.1 Unicode-basierte Version der angebotenen Schnittstelle erstellen.

Für das Erstellen der Unicode-basierten Version der angebotene Schnittstelle des *DatabaseManager* wird eine duale Schnittstelle entworfen: Die IDL-Datei der RPC-Schnittstelle des *DatabaseManager* wird so angepasst, dass sowohl Unicode-basierte Versionen als auch Codepage-basierte Versionen aller Stringdatentypen und -funktionen deklariert werden. Das ist sinnvoll, weil die Datentypen in gleicher Form auch in der

[3] Die angebotene Schnittstelle des *DocumentManager*.

RPC-Schnittstelle des *DocumentManager* verwendet werden, der auf jeden Fall für einige Zeit sowohl Unicode- als auch Codepage-basierte Clients bedienen können muss. Erweist sich der Ansatz im internen Gebrauch als praktikabel, kann er für die Schnittstelle des *DocumentManager* ohne große Anpassungen wiederverwendet werden. Die Analyse, Umstellung und Implementierung der angebotenen RPC-Schnittstelle wird in Kapitel 9 behandelt.

3.2 Benötigte Schnittstellen untersuchen

Für die Migration des *DatabaseManager*, müssen dessen benötigte Schnittstellen dahingehend untersucht werden, ob es Unicode-fähige Versionen von ihnen gibt. Diese Untersuchung wird zusammen mit den Untersuchungen zur angebotenen Schnittstelle des *DatabaseManager* in Kapitel 9 dokumentiert. Somit werden alle Schnittstellen zusammen in einem eigenen Kapitel behandelt werden.

3.3 Internen Code/Implementierung migrieren.

Um die Migration des Codes zu unterstützen, wird ein Ansatz zur erkennung von Verwendungsmustern formuliert. Verwendungsmuster sollen mit Hilfe von statischer Code-Analyse diejenigen Stellen im Quelltext erkennen, die von der Unicode-Migration betroffen sind. In diesem Zusammenhang werden zwei Tools vorgestellt, welche die Anwendbarkeit von *clang* auf dieses Verfahren experimentell untersuchen und visualisieren.

Die Anwendung dieses Teilprozesses wird in Kapitel 10 dokumentiert.

7.3.3. Einheitliche Handhabung von Strings

Dieser Abschnitt dokumentiert die Festlegung einer **einheitlichen Handhabung von Strings** in der Unicode-basierten Version des *DatabaseManager*.

Die folgenden Abschnitte dokumentieren den **aktuellen Zustand**, den **geplanten Zielzustand** und **Einschränkungen** für die angebotene RPC-Schnittstelle, an der diese einheitliche Form nicht umgesetzt werden kann.

Aktuelle Handhabung von Strings

In der aktuellen Version werden im *DatabaseManager* hauptsächlich C-Style-String und eine Reihe von Stringklassen verwendet. Zu den verwendeten Stringklassen gehören die Klasse `std::string` der C++-Standardbibliothek sowie zwei individuelle Stringklassen `CStr` (eine einfache Wrapper-Klasse für C-Style-String) und `CDynamicString` (eine Klasse mit String-Builder Funktionalität). An Schnittstellen zu COM-Klassen wird die Stringklasse `_bstr_t` verwendet (eine Wrapper-Klasse für den Datentyp `BSTR`).

Alle Strings mit Ausnahme der `_bstr_t`-Strings werden in der Kodierung der aktuellen System-Codepage dargestellt. `_bstr_t`-Objekte werden in der UTF16-LE-Kodierung dargestellt.

Einheitliche Handhabung mit einer Stringklasse

In der Unicode-basierten Version des *DatabaseManager* soll einheitlich die ATL/MFC-Klasse `CStringW`[4]

8. Analyse und Umstellung des Datenschemas und Migration des Datenbestandes

Dieses Kapitel dokumentiert die Anwendung der beiden Teilprozesse `Datenschema migrieren` und `Datenmigration planen` des Umstellungsprozesses. Dies sind die globalen Teile des Umstellungsprozesses, die für das gesamte System nur einmal angewendet werden müssen. Der Teilprozess `Datenschema migrieren` wird angewendet,

1. um eventuelle Probleme im Datenschema früh aufzudecken und
2. damit die ggf. geänderten Datentypen bei der Migration des *DatabaseManager* bekannt sind.

Die Anwendung des Teilprozesses `Datenmigration planen` hat einen explorativen Charakter und soll prinzipell klären,

1. wie aufwendig sich die Migration von Bestandsdaten gestaltet und
2. in wie weit sich der Prozess automatisieren lässt.

Abbildung 8.1 stellt die in diesem Kapitel angewendeten Teilprozesse in einen Zusammenhang mit dem Gesamtprozess.

In der Einführung von *PROXESS* wurde gezeigt, dass das Datenschema durch eine Menge von Datenbankschemata realisiert wird (siehe 4.3.2). Demnach beziehen sich die folgenden Abschnitte auf die Datenbankschemata eines Datenbanksystems.

Zunächst wird in Abschnitt 8.1 die Unterstützung des Unicode-Standards der einzelnen von *PROXESS* unterstützten Datenbanksysteme untersucht.

Abschnitt 8.2 dokumentiert die Ergebnisse der Analyse aller von *PROXESS* verwendeten Datenbanktabellen. An dieser Stelle werden zur Verdeutlichung der Sachverhalte

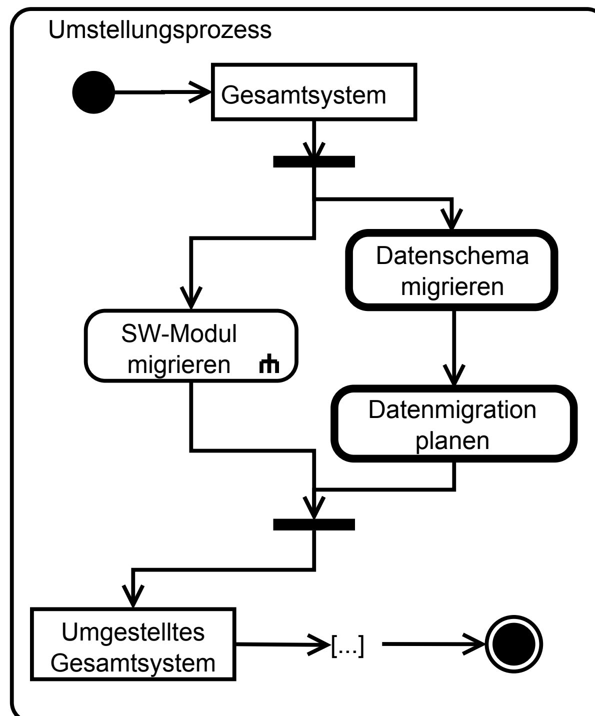

Abbildung 8.1.: Teilprozesse `Datenschema migrieren` und `Datenmigration planen`

nur repräsentative Teile des Schemas gezeigt. Eine vollständige Auflistung aller Datenbanktabellen wird im Anhang in Kapitel C aufgeführt.

Abschnitt 8.3 dokumentiert die Untersuchungen zur Migration des Datenbestandes am Beispiel der *PROXESS*-Variante für das Datenbanksystem *Microsoft SQL Server*. Es werden verschiedene Ansätze vorgestellt und es wird ein Tool präsentiert, mit dem ein konkreter Ansatz umgesetzt wird.

8.1. Unicode-Unterstüzung der Datenbanksysteme

Dieser Abschnitt identifiziert diejenigen Versionen der von *PROXESS* unterstützten Datenbanksysteme, seit denen Unicode unterstützt wird. Da *Microsoft SQL Server* das am häufigsten eingestzte Datenbanksystem ist, fällt dessen Betrachtung etwas detaillierter aus, als für die beiden anderen Systeme.

Microsoft SQL Server. Das Datenbanksystem *Microsoft SQL Server* bietet eine teilweise Unicode-Unterstützung seit dem Jahr 2000 (Version *SQL Server 2000*, [Mic12j]). In dieser Version und den Folgeversionen *SQL Server 2005*, *SQL Server 2008* und *SQL Server 2008 R2* wird eine UCS-Kodierung als datenbankinternes Encoding verwendet. Diese Einschränkungen wird an vielen Stellen in der Dokumentation übergangen, z.B.:

> *SQL Server* supports the Unicode Standard, Version 3.2. [...] *SQL Server* stores all textual system catalog data in columns having Unicode data types. The names of database objects, such as tables, views, and stored procedures, are stored in Unicode columns. (Microsoft MSDN Library [Mic06])

In der technischen Dokumentation werden diese Einschränkungen allerdings aufgeführt[1]:

> The LEN() function returns the value 2 for each supplementary character. [...] The LEFT, RIGHT, SUBSTRING, STUFF, and REVERSE functions may split any surrogate pairs and lead to unexpected results. [...] Supplementary characters are not supported for use in metadata, such as in names of database objects. (Microsoft TechNet Library [Mic12l])

Mit der Version *SQL Server 2012* werden Supplementary Characters für die SQL-Stringfunktionen unterstützt, wenn neue, für diesen Zweck eingeführte Collation-

[1] Zumindest für die Versionen SQL Server 2005, SQL Server 2008 und SQL Server 2008 R2.

Algorithmen[2] verwendet werden. Für diese Version wird in der Dokumentation erstmals UTF-16 als datenbankinternes Encoding angegeben. Allerdings können Supplementary Characters immer noch nicht für Datenbank-, Tabellen- oder Spaltennamen verwendet werden (Microsoft MSDN Library [Mic12i]).

Oracle Database. Das Datenbanksystem *Oracle Database* unterstützt Unicode-fähige Datenbanken seit Version *Oracle 7*, die 1992 erschienen ist (Oracle Whitepaper [Ora05]) und eine UTF-16-Kodierung (AL16UTF16 in der Terminologie von *Oracle*) für Tabellenspalten seit der Version *Oracle 9i*, die 2001 erschienen ist (Oracle Whitepaper [Ora05]). Hier ist anzumerken, dass *Oracle* für Systemtabellen in allen Versionen, die älter als Version *9i* sind, immer eine CESU-8-Kodierung (UTF8 in der Terminologie von *Oracle*) verwendet. Seit Version *9i* kann für Systemdatenbanken auch eine echte UTF-8-Kodierung (AL32UTF8 in der Terminologie von *Oracle*) verwendet werden. UTF-16 wird als Kodierung für Systemdatenbanken nicht unterstützt.

Da UTF-16-Kodierte Spalten erst seit Version *9i* unterstützt werden, ist diese Version auch die Mindestanforderung für den Betrieb mit *PROXESS*.

Intersystems Caché. Der „Upgrade Checklist für Version 2008.2“ ([Int08]) kann entnommen werden, dass das Datenbanksystem *Intersystems Caché* eine UTF-16-Kodierung mit Unterstützung von Supplementary Characters erstmals seit der Version *2008.2* (erschienen im Jahr 2008) unterstützt. Die Zeichen der BMP wurden – in Form von einer UCS-2-Kodierung – seit der Version 3.0 unterstützt. Das genau Release-Datum dieser Version konnte nicht ermittelt werden. Zwei Presseberichte[3][4] lassen den Zeitraum auf die Jahre 1998-2000 eingrenzen.

Folgerungen. Alle genannten Datenbanksysteme unterstützen mindestens seit dem Jahr 2001 Tabellenspalten für Strings mit 16-Bit weiten Codeunits, so dass zumindest UCS-2-kodierte Strings verarbeitet werden können. Echtes UTF-16 – mit richtiger Behandlung von Surrogate Pairs – wird von *Oracle* seit 2001, von *Intersystems* seit 2008 und von *Microsoft* erst seit 2012 unterstützt. Tabelle 8.1 gibt einen Überblick über die

[2]Dabei handelt es sich um datenbankseitige Collation-Algorithmen, die nicht mit der Implementierung des Unicode-Collation-Algorithmus in der *Windows*-API verwechselt werden dürfen.

[3]Pressemitteilung zum Release von Caché 4, `http://www.intersystems.com/press/2000/cache4.html` [Letzter Zugriff am 06.05.2012]

[4]Pressemittelung zum Support von Unicode 2.0, `http://www.intersystems.com/press/1998/encode.html` [Letzter Zugriff am 06.05.2012]

Datenbanksystem	UCS-2 (Version)	(Jahr)	UTF-16 (Version)	(Jahr)
Microsoft SQL-Server	2000	2000	2012	2012
Oracle Database	9i	2001	9i	2001
Intersystems Caché	3.0	1998 - 2000	2008.2	2008

Tabelle 8.1.: Unicode-Unterstützung der Datenbanksysteme

Version und das Erscheinungsjahr der einzelnen Datenbanksysteme im Hinblick auf UCS-2- bzw. UTF-16-Unterstützung.

Zur Anordnung von Strings implementieren alle Datenbanksysteme den Unicode-Collation-Algorithm individuell. Der Vergleich von Strings in der Datenbank ist wichtig, wenn eine Textspalte indexiert wird oder Abfrageergebnisse nach einer Textspalte sortiert werden sollen (mit dem SQL-Schlüsselwort `ORDER BY`). Bei der Verwendung des Datenbanksystems durch *PROXESS* können Abfrageergebnisse durch Verwendung des Schlüsselwortes `ORDER BY` nach Textspalten sortiert werden.

Für Anwender, die voraussichtlich keine Texte mit Zeichen außerhalb der BMP verarbeiten, ist eine Unterstützung von UTF-16 Surrogate Pairs nicht zwingend erforderlich. Das gilt in jedem Fall für alle in europäischen Sprachen verfasste Texte. Für diese Anwender ist ein Upgrade der Datenbank wegen mangelnder Unterstützung des UCA also nicht unbedingt zu empfehlen. Ein Upgrade des Datenbanksystems auf eine Version mit vollständiger Unterstützung von UTF-16 ist nur Anwendern zu empfehlen, die auch Texte mit Zeichen außerhalb der BMP verarbeiten müssen – das sind im wesentlichen in ostasiatischen Sprachen verfasste Texte. Diese Anwender benötigen einen funktionierenden UCA für Zeichen außerhalb der BMP.

8.2. Migration des Datenschemas

Dieser Abschnitt dokumentiert die Durchführung des Teilprozesses `Datenschema migrieren`. Es müssen alle Schemaelemente des Datenschemas identifiziert werden, die zum Speichern von Strings verwendet werden. Diese Elemente müssen durch solche Elemente ersetzt werden, die UTF-16LE-kodierte Strings speichern können. Im Fall von *PROXESS* handelt es sich bei dem Datenschema um eine Menge von Datenbankschemata für ein bestimmtes Datenbanksystem. Es müssen also die Schemata für jede Datenbank und damit auch die Schemata für jede in der Datenbank enthaltene Tabelle dahingehend untersucht werden, in welchen Tabellenspalten Strings gespeichert werden.

Abschnitt 8.2.1 führt alle von *PROXESS* verwendeten Datentypen für Tabellenspalten der Datenbankschemata ein und legt fest, welche Datentypen von der Umstellung betroffen sind. Bei der Betrachtung der Datentypen werden die Datentypnamen des Datenbanksystems *Microsoft SQL Server* verwendet. Für das Datenbanksystem *Intersystems Caché* sind die Namen aller Datentypen identisch. Für das Datenbanksystem *Oracle Database* gibt es Abweichungen, die ebenfalls in diesem Abschnitt aufgeführt werden.

Abschnitt 8.2.2 diskutiert alternative Möglichkeiten bei der Umstellung des Datenschemas und dokumentiert die jeweiligen Entscheidungen.

8.2.1. Verwendete Datentypen in der Datenbank

In diesem Abschnitt werden alle von *PROXESS* verwendeten Datentypen für Spalten von Tabellen der relationalen Datenbank aufgelistet. Die Datentypen lassen sich in drei Kategorien einteilen:

1. Datentypen, deren Verwendungen in keinem Fall von einer Umstellung betroffen sind.
2. Datentypen, deren Verwendungen in jedem Fall von einer Umstellung betroffen sind.
3. Datentypen, deren Verwendungen teilweise von einer Umstellung betroffen sind.

Alle verwendeten Datentypen werden in den folgenden Abschnitten einer dieser Kategorien zugeordnet. Zur Veranschaulichung werden zwei in Bezug auf die Unicode-Migration repräsentative Tabellen aufgeführt: Abbildung 8.3 stellt die Tabelle `CProfiles` (Tabelle für Benutzer-individuelle Profildaten) aus der Master-Datenbank von *PROXESS* und die Tabelle `DocTypes` (Tabelle für Dokumenttypen) aus einer beliebigen Benutzerdatenbank dar. Die (potentiellen) String-Spalten sind farblich hervorgehoben. Abbildung 8.2 ist eine Legende für Abbildung 8.3.

Für eine Auflistung aller Datenbanktabellen und eine Beschreibung ihrer Funktionen wird auf das Kapitel C im Anhang verwiesen.

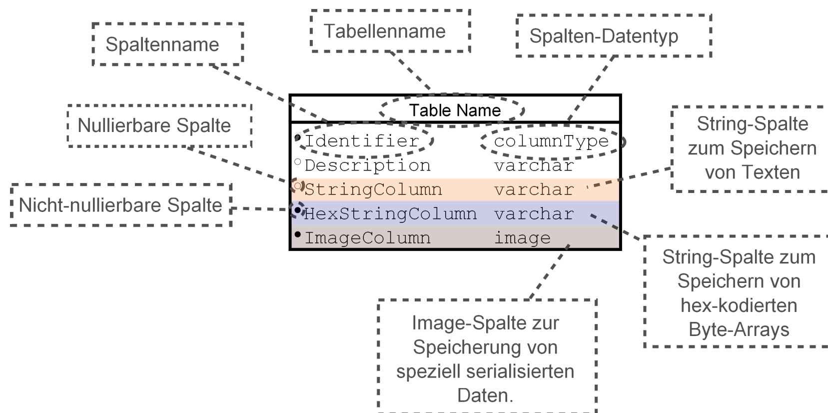

Abbildung 8.2.: Legende für Abbildungen von Datenbanktabellen

DocType	
DocTypeID	money
DocTypeName	varchar(31)
DocTypeACL	image
CollectionFlag	int
CollectionFormat	image
DefaultKeepForDays	int
DefaultMediaType	int
FullText	int
OCRFlag	int
Datahash	varchar(41)
CryptFlag	int

CProfiles	
UserGroupId	money
Modulename	varchar(100)
VProduct	int
VModule	int
Revision	int
Datahash	varchar(50)
DataXml	text

Abbildung 8.3.: Repräsentative Datenbanktabellen

Nicht von einer Umstellung betroffene Datentypen

Für einem Teil der Datentypen ist offensichtlich, dass sie nicht von einer Umstellung betroffen sind. Darunter fallen z.B. der Datentyp **`money`**[5], der zur Speicherung von IDs[6] verwendet wird oder der Typ **`datetime`**, der zur Speicherung von Zeitstempeln verwendet wird. Die Typen **`int`**, **`smallint`**, **`tinyint`**[7] und **`float`**[8] vervollständigen die Menge von Datentypen, die nicht von einer Umstellung betroffen sind.

Die Unterschiede bei den Datentypnamen für *Oracle Database* werden im Anschluss an den nächsten Abschnitt in Tabelle 8.2 gezeigt.

Von einer Umstellung betroffene Datentypen

Die übrigen Datentypen sind prinzipiell von einer Umstellung betroffen; dabei handelt es sich um die Typen `char`, `varchar`, `text` und `image`.

Die Typen **`char`**, **`varchar`**, **`text`** (im Folgenden unter dem Namen `varchar` zusammengefasst) sind dafür vorgesehen, Strings mit einer 8-Bit-Kodierung in der Datenbank zu speichern. Im Fall von *Microsoft SQL Server* entspricht diese 8-Bit-Kodierung einer bestimmten Windows-Codepage, für welche die Datenbank konfiguriert worden ist.

Diese Typen haben gemeinsam, dass alle unterstützten Datenbanksysteme Datentypen anbieten, die die Speicherung von 16-Bit-kodierten Strings erlauben, nämlich **`nchar`**, **`nvarchar`** und **`ntext`**. Spalten von diesen Typen sind als Unicode-Pendant[9] zu den „klassischen“ `varchar`-Spalten zu verstehen.

Zwar können in den 8-Bit-weiten Spalten natürlich UTF-8 Strings – und damit auch vollwertige Unicode Strings – gespeichert werden, das Datenbanksystem muss in diesem Fall aber auch den Unicode-Collation-Algorithmus für diese Spalten unterstützen. Das ist z.B. bei dem Datenbanksystem *Microsoft SQL Server* nicht der Fall. `varchar`-Typen werden in *PROXESS* in zwei verschiedenen Zusammenhängen verwendet:

1. In der betroffenen Spalte wird ein Textstring gespeichert. Beispiele für diese Verwendungsart sind Objektnamen (Datenbankname, Dokumenttypname oder

[5]Ein 8 Byte großer Datentyp zur Speicherung von Dezimalzahlen mit vier Nachkommastellen.
[6]Die IDs sind eigentlich 8 Byte große ganze Zahlen. Z.B. wird die ID 12034 mit dem Wert 12.034 gespeichert.
[7]Das sind Datentypen für ganze Zahlen, die einen 4, 2 bzw. 1 Byte großen Speicher haben.
[8]Ein Datentyp für Gleitkommazahlen mit 4 Byte großen Speicher.
[9]Genaugenommen: Ein UTF-16-Pendant.

Feldname) und Feldwerte von Dokumenten. Jede dieser Spalten soll auf den entsprechenden `nvarchar`-Typ umgestellt werden. Die Spalten in den Beispieltabellen aus Abbildung 8.3, auf welche dieser Anwendungsfall zutrifft, sind rot unterlegt.

2. In der betroffenen Spalte wird ein als String kodiertes Byte-Array (i.d.R. in hexadezimaler Notation) gespeichert. Beispiele für diese Verwendungsart sind Hashes von Benutzerpasswörtern, Datenbanken, Dokumenttypen und Dokumenten. In diesem Fall muss der Datentyp der Spalte nicht unbedingt umgestellt werden, da der zur Zeit verwendeten Zeichensatz in jedem Fall ausreicht, um alle möglichen Werte zu kodieren. Die Spalten in den Beispieltabellen aus Abbildung 8.3, auf welche dieser Anwendungsfall zutrifft, sind blau unterlegt.

Diese beiden Verwendungsformen können nur bei Kenntnis des Datenbankschemas unterschieden werden. Allerdings gibt der Name der Spalte meistens einen Hinweis auf deren Verwendungszweck. Z.B. werden Spalten, die in ihrem Namen die Wörter `Name` oder `Des` (Abk. für Description) enthalten, durchgehend zur Speicherung von Text verwendet. Die häufig auftretende Tabellenspalte `Datahash` wird immer zur Speicherung von hexadezimal-kodierten Byte-Arrays eingesetzt.

Spalten vom Datentyp **`image`** werden durchgehend dazu verwendet, um speziell kodierte Strings als BLOB[10] zu speichern. Die Spalten der Beispieltabellen in Abbildung 8.3, welche den Datentyp `image` verwenden, sind grau unterlegt.

Bei den „speziell kodierten Strings" handelt es sich um eine serialisierte Form eines Byte-Arrays, das keine Null-Bytes mehr enthält. Diese serialisierte Form kann als `NUL`-terminierter String interpretiert werden. Der Serialisierungsmechanismus wird im folgenden Abschnitt (Abschnitt 8.2.2) näher erläutert.

Auch die Verwendung des Typs `image` lässt sich in zwei Kategorien aufteilen:

1. In der betroffenen Spalte werden Daten gespeichert, die keine Textinformationen enthalten. Ein Beispiel dafür sind die ACLs für Benutzerdatenbanken, Dokumenttypen und Dateien, die als Liste von Rechte-Tupeln[11] serialisiert werden. Dieser Verwendungsfall macht den größten Teil der Vorkommnisse von `image`-Spalten aus.
2. In der betroffenen Spalte werden Daten gespeichert, die Textinformationen enthalten. Von diesem Fall ist genau eine Datenbankspalte betroffen. Diese Spalte

[10] **B**inary **L**arge **Ob**ject, eine u.U. große Menge an Binärdaten, die in der Datenbank gespeichert ist.

[11] Ein Rechte-Tupel besteht aus der Benutzer- oder Gruppen-ID (8-Byte Ganzzahl) und einem Rechte-Flag (4-Byte Ganzzahl, Bitmaske) das die Zugriffsrechte beschreibt.

SavedQueries	
•QueryID	money
•QueryName	varchar(63)
•Usersgroups	image
•SqlWhereComponent	image
◦FormatString	image
•CreatorID	money
◦Options	int
◦OptionsParams	varchar(255)

Abbildung 8.4.: Datenbanktabelle `SavedQueries`

wird dazu verwendet, um die SQL-Syntax einer individuellen Suchabfrage zu speichern (In der Tabelle `SavedQueries` der zu einer Benutzerdatenbank gehörenden relationalen Datenbank). Die entsprechende Datenbanktabelle wird in Abbildung 8.4 dargestellt.

Die genaue Verwendungsform einer `image`-Spalte kann anhand des in ihr gespeicherten Wertes ermittelt werden. Durch einen TYP-Marker (s.u.) am Beginn des serialisierten Byte-Streams kann die Art der gespeicherten Daten identifiziert werden.

Unterschiede bei Datentypnamen zwischen den Datenbanksystemen

Dieser Abschnitt listet die Namen für Datentypen der drei unterstützten Datenbanksysteme auf. Tabelle 8.2 zeigt die Namen aller verwendeten Datentypen für alle drei Datenbanksysteme. Bei der Beschreibung der Datentypen `char` und `varchar` wird eine variable Größenangabe `N` gemacht. Der konkrete Wert unterscheidet sich von Fall zu Fall, ist aber niemals größer als `255`.

Da *PROXESS* die native Schnittstelle von *Oracle Database* verwendet, werden auch die *Oracle*-spezifischen Datentypen verwendet[12]. Der Datentyp `NUMBER` wird sowohl zum Speichern von Ganzzahlen als auch zum Speichern von Gleitkommazahlen verwendet. Dabei stellen die Größenangaben (in der Tabelle die Zahlen `20`, `10`, `3` und `1`) die maximale Anzahl an Dezimalstellen dar und nicht etwa die Anzahl von Bytes.

Spalten, die in den Datenbanksystemen *Microsoft SQL Server* und *Intersystems Caché* den Datentyp `image` haben, werden in *Oracle Database* als Spalten vom Typ `VARCHAR2(2000)` dargestellt. Für dieses Datenbanksystem dürfen also nicht alle

[12] Die Internetseite `http://ss64.com/ora/syntax-datatypes.html#precision` gib einen guten Überblick über die wichtigsten Datentypen von *Oracle Database*.

Microsoft SQL Server	*Intersystems Caché*	*Oracle Database*
`money`	`money`	`NUMBER(20)`
`int`	`int`	`NUMBER(10)`
`smallint`	`smallint`	`NUMBER(3)`
`tinyint`	`tinyint`	`NUMBER(1)`
`float`	`float`	`NUMBER`
`datetime`	`datetime`	`DATE`
`char(N)`	`char(N)`	`VARCHAR2(N)`
`varchar(N)`	`varchar(N)`	`VARCHAR2(N)`
`text`	`text`	`LONG`
`image`	`image`	`VARCHAR2(2000)`

Tabelle 8.2.: Unterschiede bei Datentypnamen zwischen den Datenbanksystemen.

`VARCHAR2`-Spalten umgestellt werden. Die betroffenen Spalten können durch die Größenangabe von `2000` identifiziert werden, da alle `VARCHAR2`-Spalten, die tatsächlich Strings speichern, eine Größenangabe haben, die kleiner als `256` ist.

8.2.2. Entscheidungen

Im letzten Abschnitt wurde gezeigt, dass es Daten gibt,

1. die Strings sind, obwohl der Datentyp der zugehörigen Spalte kein String-Datentyp ist
2. die eigentlich keine Strings sind, aber in einer kodierten Form in Spalten mit einem String-Datentyp gespeichert werden.

Für beide Fälle wird die Entscheidung getroffen, die betroffenen Spalten so behandeln, wie Spalten, die kein Sonderfall sind. Das ist sinnvoll, da die intern verwendeten *PROXESS*-spezifischen Datentypen zur Darstellung von Datenbankdaten fest an die Datentypen der Datenbank gebunden sind. Dies wird bei der Betrachtung der Schnittstellen des *DatabaseManager* in Kapitel 9 verdeutlicht. Die Änderung des Datentyps einer Tabellenspalte hätte zur Folge, dass sich auch der *PROXESS*-interne Typ der Daten verändert. Daher müsste vor allem die Implementierung im *DocumentManger* in einer Weise an das geänderte Datenschema angepasst werden, die deutlich über die eigentliche Unicode-Migration hinausgeht. Das würde die Erfolgsausichten für eine Migration verschlechtern. Wegen der eingeschränkten Testbarkeit (siehe Kapitel 7.3) kann der Erfolg der Migration ohnehin nur schwer validiert werden kann.

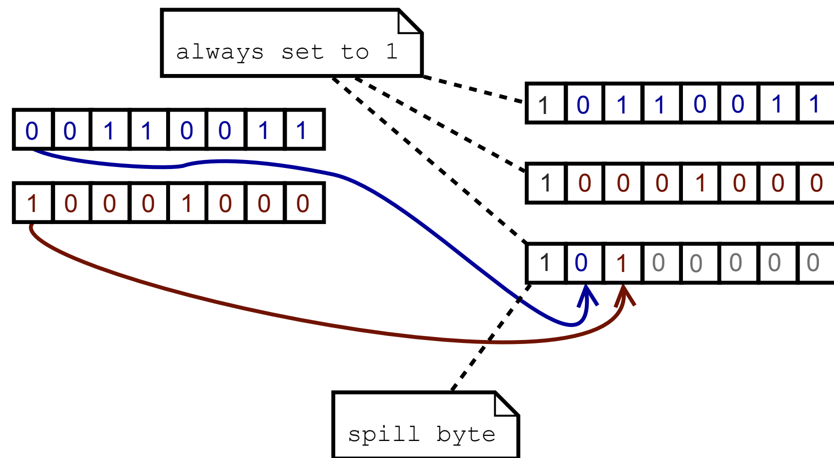

Abbildung 8.5.: rights_s-Serialisierung

Im Folgenden werden die beiden Fälle näher erläutert und die entsprechenden Lösungen präsentiert.

Fall 1. Der erste Fall tritt genau einmal auf und betrifft gespeicherte, individuelle Suchabfragen, die in der Tabelle SavedQueries in einer Spalte vom Typ image abgelegt werden. Der Typ image soll beibehalten werden. Um die Handhabung von Unicode-Strings in Spalten vom Typ image zu erläutern, wird im Folgenden der Serialisierungsmechanismus für die Darstellung von image-Spalten beschrieben.

Der Datenbank-Datentyp image ist an den *PROXESS*-internen Datentyp rights_s gebunden, der primär zur Speicherung von ACLs in der Datenbank eingesetzt wird. Der Typ rights_s ist eine Typdefinition für char*, und wird an den RPC-Schnittstellen als NUL-terminierter String interpretiert. ACLs sind Listen von Integer-Tupeln und enthalten somit i.d.R. Null-Bytes. Zur Darstellung einer ACL als rights_s-Objekt werden durch einen Serialisierungsmechanismus alle Null-Bytes entfernt. Die Funktionsweise dieses Serialisierungmechanismus wird in Abbildung 8.5 skizziert.

Für jedes Byte eines beliebig großen Byte-Stroms setzt der Serialisierungsmechanismus das High-Bit auf den Wert 1 und schreibt das modifizierte Byte[13] in den Ausgabestromg. Der ursprüngliche Wert des High-Bits wird an eine feste Position in einem sog. Spil-Byte geschrieben. Das High-Bit des Spil-Byte wird selbst immer auf den Wert 1 gesetzt. Somit kann ein Spil-Byte die Werte der High-Bits von 7 Bytes verlustfrei

[13]Das modifizierte Byte hat sicherlich nicht den Wert 0.

aufnehmen. Ist der zu serialisierende Byte-Strom länger als 7 Byte, wird das Spil-Byte in den Ausgabestring geschrieben und es wird ein neues Spil-Byte für die nächsten 7 Byte erzeugt. Ist der zu serialisierende Byte-Strom vollständig gelesen, wird das aktuelle Spil-Byte in den Ausgabestrom geschrieben und die Serialisierung ist abgeschlossen. Wird z.B. ein 32-Bit großer Integer serialisiert, werden dazu 5 Bytes (4 Bytes + 1 Spil-Byte) benötigt; wird ein 64-Bit großer Integer serialisiert, werden dafür 10 Bytes (8 Bytes + 2 Split-Bytes) benötigt. Abbildung 8.5 zeigt, dass für die Serialisierung eines 16-Bit großen Integers – mit denen u.a. UTF16-Codepoints dargestellt werden – 3 Bytes (2 Bytes + 1 Spil-Byte) benötigt werden.

Damit der Deserialisierer erkennen kann, welche Daten er deserialisieren muss, wird an den Beginn des serialisierten Streams ein sog. TYP-Marker geschrieben. Dieser besteht aus den drei ASCII-kodierten Zeichen `'T'`,`'Y'` und `'P'` gefolgt von einer Ziffer, die den Typ der enthaltenen Daten spezifiziert. Z.B. haben ACLs den Typ-Code `TYP1`; gespeicherte Suchabfragen haben den Typ-Code `TYP4`.

Für den Fall der gespeicherten Suchabfragen wird derjenige Teil des Serialisierungsmechanismus mit dem Spil-Byte nicht benötigt, weil die gespeicherte Suchabfrage selbst ein String ist und in der jetzigen Form als `NUL`-terminierter String vorliegt.

Im Rahmen der Unicode-Migration wird die Kodierung für Strings auf UTF-16LE umgestellt werden. Weil die gespeicherte Suchabfrage in diesem Fall nicht mehr als `NUL`-terminierter String vorliegt, muss der Serialisierungsmechanismus zur Übertragung der gespeicherten Suchabfrage eingesetzt werden. Das führt zu einem Overhead von einem Spil-Byte pro 3,5 Codeunits. Die Serialisierung von UTF16-LE-String muss einen eigenen TYP-Marker erhalten.

Alternativ kann der String vor der Serialisierung in die UTF-8-Kodierung konvertiert werden. In dieser Form handelt es sich wieder um einen einfachen `NUL`-terminierten String. Auch im Fall einer UTF-8-Konvertierung muss ein neuer TYP-Marker eingeführt werden, damit entschieden werden kann, ob die Daten bereits migriert worden sind.

Welche der beiden Kodierungen für die Übertragung (und Speicherung in der Datenbank) von gespeicherten Suchabfragen eingesetzt wird, muss während der Migration des *DocumentManager* entschieden werden. In beiden Fällen muss die Datenmigration einmal pro Installation durchgeführt werden. Das Datenbankschema muss für diesen Fall nicht angepasst werden.

Fall 2. Der zweite Fall tritt an mehreren Stellen auf und umfasst Verwendungen von `varchar`-Spalten zur Speicherung von hexadizmal-kodierten Hashes (bzw. Byte-Arrays) von Datensätzen oder Benutzerpasswörtern. Da die Hashes den gleichen Datenbank-Datentyp verwenden wie herkömmliche Strings (z.B. Benutzernamen, Feldnamen, etc.), sind die Hashes aus Sicht der *PROXESS*-spezifischen Datentypen nicht von herkömmlichen Strings zu unterscheiden.

Im Rahmen der Unicode-Umstellung werden die Datentypen der Spalten für herkömmliche Strings zu den entsprechenden `nvarchar`-Typen geändert. Im Hinblick auf das Datenmodell gibt es drei Möglichkeiten für die Handhabung von String-kodierten Hashes:

1. Der Datentyp für Spalten, in denen Hashes gespeichert werden, wird ebenfalls zu `nvarchar` geändert. So werden herkömmliche Strings und String-kodierte Hashes in der Unicode-basierten Version von *PROXESS* weiterhin gleich behandelt.

2. Der Datentyp für Spalten, in denen Hashes gespeichert werden, wird nicht geändert. In diesem Fall werden String-kodierte Hashes weiterhin als Codepage-basierte Strings gespeichert. Folglich muss in der Unicode-basierten Version von *PROXESS* ein neuer *PROXESS*-spezifischer Datentyp für Codepage-basierte Strings eingeführt werden.

3. Der Datentyp für Spalten, in denen Hashes gespeichert werden, wird zu `image` geändert. Somit wird ein Hash in der Datenbank als Byte-Array modelliert. Eine Spalte vom Datentyp `image` wird in der *PROXESS*-internen Darstellung allerdings immer als `rights_s`-Objekt interpretiert. Folglich würden die betroffenen Hashes in der Unicode-basierten Version von *PROXESS* ebenfalls als `rights_s`-Objekte interpretiert werden. Die Objekte könnten – ohne weitere Vorkehrungen – nicht deserialisiert werden, weil der TYP-Marker zu Beginn des serialisierten Strings fehlen würde.

Aus Datenbanksicht scheint die Wahl des Datentyps `image` (Punkt 3) am natürlichsten zur Speicherung von Hashes zu sein, da der Datentyp den tatsächlich gespeicherten Daten am besten entspricht.

Wird der Datentyp unverändert beibehalten, wird nur die Hälfte des Speicherplatzes gebraucht. Diese Tatsache spricht für die Wahl des Datentyps `varchar` (Punkt 2).

Die Folgen, die jede der beiden Änderung für das interne Datenmodell von *PROXESS* nach sich zieht, macht aber deutlich, dass Hashes am besten in der jetzigen Form als

String-kodierte Hashes beibehalten werden, und dass sie – genau so wie alle anderen Strings – in Unicode-Strings umgewandelt werden.

8.2.3. Zusammenfassung

Dieser Abschnitt fasst die Erkenntnisse aus der Analyse des Datenschemas und den Entscheidungen bezüglich der Alternativen bei der Umstellung zusammen.

- Alle Spalten in Datenbanktabellen, in denen Strings gespeichert werden, sind vom Typ `char`, `varchar` oder `text`. Diese Typen werden zusammenfassend `varchar`-Typen – bzw. die zugehörigen Spalten `varchar`-Spalten – genannt. Es gibt eine Ausnahme, in der ein String in einer Tabellenspalte vom Typ `image` gespeichert wird. Dabei handelt es sich um den Suchausdruck für eine gespeicherte individuelle Suchabfrage in der Tabelle für gespeicherte Suchabfragen (`SavedQueries`).
- Alle `varchar`-Spalten werden zu entsprechenden `nvarchar`-Spalten umgestellt. Das schließt auch die Spalten ein, in denen hexedezimal-kodierte Hashes als Strings gespeichert werden.
- Diejenige Spalte der Tabelle `SavedQueries`, in welcher der Suchausdruck gepspeichert wird, ist vom Typ `image`. Dieser Typ wird in der Unicode-basierten Version von *PROXESS* beibehalten. Der Serialisierungsmechanismus muss für die Handhabung von UTF16-LE String angepasst werden (entweder durch eine Konvertierung des Suchausdrucks in einen UTF-8-kodierten String, oder durch eine Serialisierung der Menge von 16-Bit Codeunits).

Diese Festlegungen wurden so getroffen, dass die Datentypen des Schemas so wenig (bzw. so einheitlich) wie möglich geändert werden. Eine Änderung der Datentypen hat unmittelbare Auswirkungen auf die *PROXESS*-interne Darstellung der Daten. Bei nicht einheitlichen Änderungen führt das zwangsläufig zu Refactorings, die weit über die eigentliche Unicode-Migration hinausgehen.

8.3. Exploration: Migration des Datenbestandes

Dieser Abschnitt beschreibt die explorative Durchführung des Teilprozesses `Datenmigration planen`. In diesem Rahmen wurde das Tool *PROXESS - Data-*

base2WideChar[14] (kurz: *PDB2WC*) entwickelt, das Update-Skripts für alle Datenbanktabellen automatisch erstellt. Im Folgenden wird die Entwicklung des Tools motiviert und die Umsetzung skizziert. Der C#-Quelltext von *PDB2WC* befindet sich auf der CD, die dieser Arbeit beiliegt. Im Anhang (Kapitel D) wird die Verwendung des Tools beschrieben.

Motivation.

Die Ergebnisse des letzten Abschnittes zeigen, dass genau alle `varchar`-Spalten zu `nvarchar`-Spalten umgestellt werden müssen, um die von *PROXESS* verwendeten Datenbankschemata Unicode-fähig zu machen.

Die Mengen der `varchar`-Spalten sind für alle Tabellen mit Ausnahme der in jeder Benutzerdatenbanken enthaltenen Tabelle `Docs` (der Tabelle für Dokumente) für alle *PROXESS*-Installationen identisch. In Zahlen bedeutet das: **Die durchzuführenden Migrationsschritte sind für 35 von 36 Tabellen identisch.**

Die Tabelle `Docs` enthält alle bei der Konfiguration einer Benutzerdatenbank angelegten Felder. Abbildung 8.6 zeigt eine Tabelle `Docs`, in der drei Spalten für individuelle Felder (das sind die Spalten, deren Name mit `CUSTOM_FIELD` beginnt) enthalten sind. Die Namen dieser Felder dienen als Identifier und werden bei der Konfiguration frei gewählt[15]. **Die konkrete Migration für jede `Docs`-Tabelle gestaltet sich unterschiedlich.**

Das Tool *PDB2WC* wird dazu verwendet, die Update-Skripts für `Docs`-Tabellen automatisch zu erstellen, und die Update-Skripts für die alle anderen Tabellen einmalig zu erstellen.

Umsetzung.

[Ued08] zeigt, dass sich die Unicode-Migration einer Datenbanktabelle für das Datenbanksystem *Microsoft SQL Server* am bestem mit einem Staging-Table in Kombination mit dem SQL-Befehl `INSERT SELECT` zum Überspielen der Daten realisieren lässt. Dieses Verfahren wird durch die von *PDB2WC* erzeugten Update-Skripts implementiert. Das Tool verwendet die für *Microsoft SQL Server* spezifischen Object Catalog

[14] Der Name ist eine Homage an die *Windows*-Funktion `MultiByteToWideChar`, die zur Konvertierung von Codepage-basierten Strings zu Unicode-Strings verwendet wird.

[15] Mit Einschränkungen bezgl. der maximalen Länge und der erlaubten Zeichen.

Docs	
•DocID	money
◦DocTypeID	money
◦DocDes	varchar(63)
◦CreateDate	datetime
◦DocACL	image
◦Creator	money
◦UpdateDate	datetime
◦Updater	money
◦KeepForDays	int
◦DocsDocTypeName	char(31)
◦DocState	int
◦Datahash	varchar(41)
◦SecData	text
◦CUSTOM_FIELD	varchar(61)
•CUSTOM_FIELD2	int
◦CUSTOM_FIELD3	varchar(255)

Abbildung 8.6.: Datenbanktabelle `Docs`

Views[16], um die benötigten Informationen über die Datenbankschemata (Tabellennamen, Spaltennamen und -typen, Indexnamen und -typen) während der Erzeugung der Skripts abzurufen. Die folgende Auflistung erläutert die Schritte, die bei der Migration durchgeführt werden müssen. Die Struktur der erzeugten Update-Skripts ist analog aufgebaut.

1. **Staging-Table T′ erzeugen.** Die Staging-Tabelle `T'` wird neu erzeugt. Jede Spalten `S` aus der Quelltabelle `T` wird betrachtet. Ist `S` keine `varchar`-Spalte, wird `S` unverändert (mit gleichem Namen und Datentyp) in `T'` deklariert. Ist `S` eine `varchar`-Spalte, wird nicht `S` sondern eine dem Datentyp von `S` entsprechende `nvarchar`-Spalte `S'` mit dem gleichen Namen wie `S` in `T'` deklariert.

2. **Daten aus der alten Tabelle T mit dem Befehl `INSERT SELECT` in T′ überspielen.** Mit dem Befehl `INSERT INTO T' SELECT * FROM T` wird die eigentliche Migration durchgeführt. Die Konvertierung von `varchar`-Spalten zu `nvarchar`-Spalten findet implizit statt ([Mic12b]).

3. **T löschen.** Die Quelltabelle `T` wird mit dem Befehl `DROP T` gelöscht.

4. **T′ nach T umbenennen.** Die Staging-Tabelle `T'` wird umbenannt, so dass sie den gleichen Namen wie `T` hat. Dazu wird die Stored Procedure `sp_rename`([Mic12e]) eingesetzt: `sp_rename 'T'' 'T' 'Object'`.

[16] `http://msdn.microsoft.com/de-de/library/ms189783` [Letzter Zugriff am 21.08.2012]

5. **Alle Indexe von T für T′ neu erzeugen.** Alle Indexe (z.B. der Primary Key einer Tabelle) der Quelltabelle `T` werden für die (jetzt umbenannte) Tabelle `T'` neu erzeugt. Das Abrufen der Indexe von `T` ist möglich, weil `T` in Schritt 4 nicht wirklich gelöscht wurde, sondern der Löschbefehl nur in das Skript geschrieben wurde.

Anmerkung. *PROXESS* verwendet keine Foreign Key Constraints (FKC). Würden FKCs verwendet, müssten alle Tabellen nach FKCs durchsucht werden, die auf den Primärschlüssel von `T` verweisen. Bevor `T` in Schritt 3 gelöscht wird, müssten die alten FKCs gelöscht werden. Die FKCs würden dann (im 6. Schritt) mit einem Verweis auf die umbenannte Staging-Tabelle `T'` neu erzeugt werden.

Die Update-Skripts, die von *PROXESS - DatebaseToWideChar* erzeugt werden, können fehlerfrei ausgeführt werden und liefert die gewünschten Ergebnisse. Die richtige Funktionsweise der Skripts wurde an verschiedenen von *PROXESS* verwendeten Datenbanktabellen validiert. Die überprüften Datenbanktabellen wurden initial für die *Windows*-Codepage 1252 erstellt. Bei der Migration wurden auch deutsche Umlaute in die richtigen Unicode-Codepoints überführt. Die Migration einer `Docs`-Tabelle mit etwa 6000 Einträgen dauert auf einem betagten *Intel Core 2 Duo* mit 2 Ghz Prozessor und 2 GB RAM etwas weniger als eine Sekunde. [Ued08] hat für die Migration von ca. 130 Millionen Datensätze, die insgesamt etwa 20 GB Speicherplatz belegen, auf einem 2,33 Ghz Quad-Core mit 16 GB RAM eine Zeit von 61 Minuten gemessen.

8.4. Zusammenfassung

Die Unterstützung von Unicode durch die Datenbanksysteme (insbedondere für UTF16-LE-kodierte Strings) wurde untersucht und die minimalen unterstützenden Versionen wurden identifiziert.

Die von *PROXESS* verwendeten Datenbankschemata wurden analysiert und dabei wurden diejenigen Datentypen identifiziert, mit denen Strings gespeichert werden. Es wurde gezeigt, wie diese Datentypen zu ändern sind, damit UTF16-LE-kodierte Strings in den zugehörigen Spalten gespeichert werden können. Die *PROXESS*-interne Repräsentation dieser Datentypen muss im weiteren Verlauf der Migration an die UTF16-LE-Kodierung angepasst werden.

Der überwiegende Teil der Bestandsdaten kann durch statische Update-Skripts migriert werden, da die Datenbank- und Tabellenstruktur für alle *PROXESS*-

Installationen zu großen Teilen identisch ist. Die Tabelle, in der Dokumente einer Benutzerdatenbank gespeichert werden, kann beliebig viele individuelle String-Spalten enthalten[17]. Für diese Tabellen muss ein individuelles Update-Skript ausgeführt werden. Es wurde ein Tool vorgestellt, mit dem diese Update-Skripts für das Datenbanksystem *Microsoft SQL Server* automatisch erzeugt werden.

Die Erkenntnisse dieses Kapitels helfen bei der Analyse und Umstellung der Interfaces und der Implementierung des *DatabaseManager*, die in den folgenden Kapiteln 9 und 10 behandelt werden.

[17]So eine Spalte gibt es für jedes in der Benutzerdatenbank definierte String-Feld.

9. Analyse und Umstellung der Schnittstellen

Dieses Kapitel dokumentiert die Anwendung der Teilprozesse `Unicode-basierte Version der angebotenen Schnittstelle erstellen` und `Benötigte Schnittstellen untersuchen` für den *DatabaseManager.* Der betroffene Ausschnitt aus dem Umstellungsprozess wird in Abbildung 9.1 dargestellt. Abbildung 9.2 gibt einen Überblick über die verschiedenen Schnittstellen des *DatabaseManager*.

Die Anwendung des Teilprozesses `Unicode-basierte Version der angebotenen Schnittstelle erstellen` wird in Abschnitt 9.1 behandelt. Es wird eine duale Schnittstelle entworfen, die sowohl Codepage-basierte Varianten als auch Unicode-basierte Varianten von allen String-Datentypen und -Funktionen anbietet. Die Signatur der jetzigen Schnittstelle bleibt in Form von Typdefinitionen erhalten, die in Abhängigkeit eines Präprozessor-Makros auf die Unicode-basierte oder die Codepage-basierte Variante der dualen Schnittstelle verweisen. Dadurch wird es möglich, den umgestellten *DatabaseManager* in das alte System zu integrieren. Zusätzlich sollen die Erfahrungen, die bei der Implementierung und der späteren Verwendung der dualen Schnittstelle gemacht werden, zeigen, ob es sinnvoll ist, diesen Ansatz bei der Entwicklung der Unicode-basierten Version der angebotenen (und externen) Schnittstelle des *DocumentManager* wiederzuverwenden.

Die Anwendung des Teilprozesses `Benötigte Schnittstellen untersuchen` wird in Abschnitt 9.2 behandelt. Die benötigten Schnittstellen des *DatabaseManager* umfassen die API von *Microsoft Windows*, den *OLE DB Provider for MSSQL Server*, die *Java Invocation API* (die native C-Schnittstelle von Java), sowie eine Reihe von Akzentum entwickelte *COM*-Module.

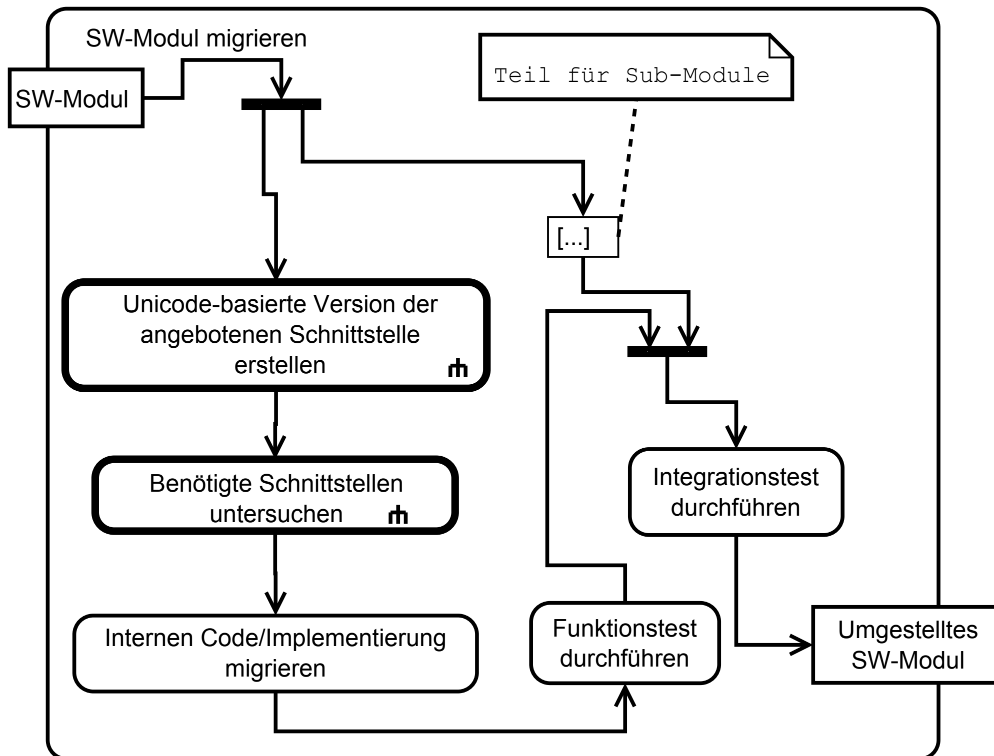

Abbildung 9.1.: Teilprozesse `Unicode-basierte Version der angebotenen Schnittstelle erstellen` und `Benötigte Schnittstellen untersuchen`.

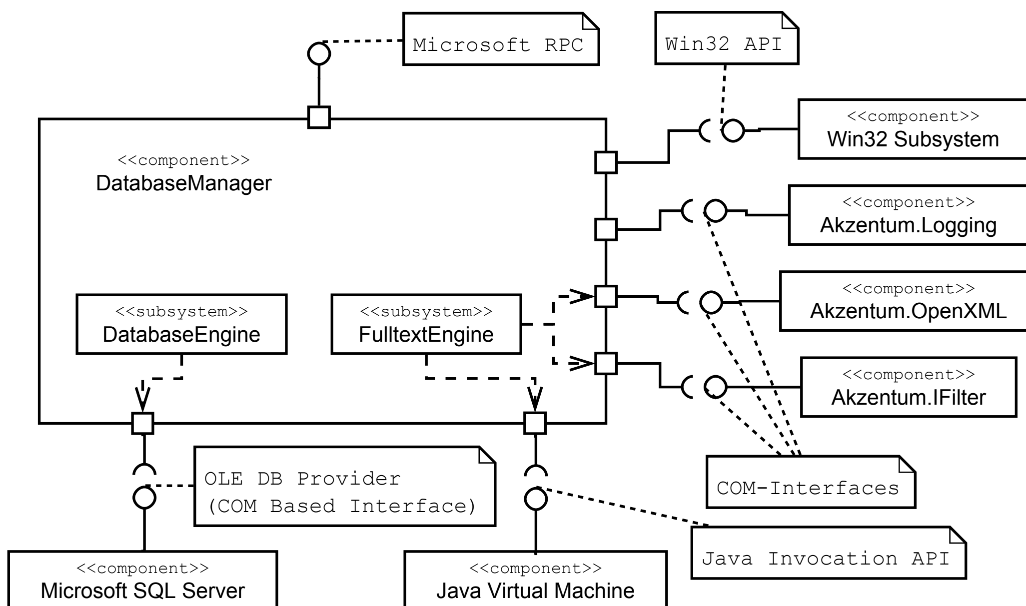

Abbildung 9.2.: Schnittstellen des *DatabaseManager*

Kategorie	Anzahl Funktionen
Verbindungsaufbau und -abbau	7
Funktionen zur Datenbankkonfiguration	10
Funktionen für Datenbankabfragen	7
Funktionen der Volltext-Schnittstelle	14
Sonstige Funktionen	8
Gesamt	**46**

Tabelle 9.1.: Funktionen der angebotenen Schnittstelle des *DatabaseManager.*

9.1. Angebotene RPC-Schnittstelle

Die angebotene Schnittstelle des *DatabaseManager* ist eine RPC-Schnittstelle. Sie wird hauptsächlich vom *DocumentManager* aber auch vom *StorageManager* verwendet. Die folgenden Abschnitte dokumentieren die Analyse der aktuellen Schnittstelle und den Entwurf und die Implementierung der dualen Schnittstelle für die Unicode-basierte Version des *DatabaseManager*. Zuvor wird ein Überblick über die Funktionen der angebotenen Schnittstelle des *DatabaseManager* gegeben.

Abschnitt 9.1.1 gibt einen Überblick über alle Verwendungsformen von Strings in der angebotenen Schnittstelle.

Abschnitt 9.1.2 führt die Datentypen `header_t` und `union_array_t` ein und beschreibt den engen Zusammenhang zwischen diesen Datentypen und den Datentypen des Datenbanksystems.

Abschnitt 9.1.3 dokumentiert den Entwurf der dualen Schnittstelle für die Unicode-basierte Version des *DatabaseManager*.

Abschnitt 9.1.4 dokumentiert die Implementierung der dualen Schnittstelle. Dies umfasst die Implementierung von Adapter-Funktion und die Anpassung des internen Codes an geänderte Datentypen. Dieser Zustand ist der Ausgangspunkt für weitere Analyseschritte (Kapitel 10).

Funktionsüberblick

Tabelle 9.1 teilt die 46 Funktionen der angebotenen Schnittstelle des *DatabaseManager* in Kategorien ein, deren Bedeutung im Folgenden kurz erläutert wird.

Die Kategorie **Verbindungsaufbau und -abbau** enthält Funktionen zum Aufbauen und Abbauen von RPC- und Datenbank-Verbindungen.

Die Kategorie **Funktionen zur Datenbankkonfiguration** enthält Funktionen zum Registrieren und Löschen von Datenbanken und zum Hinzufügen und Löschen von Feldern und Indexen.

Die wichtigste Kategorie (an der Aufrufhäufigkeit gemessen) stellt die Kategorie **Funktionen für Datenbankabfragen** dar. Darunter fallen Funktionen, mit denen Datenbankabfragen abgesetzt werden und Funktionen, mit denen die Ergebnisse von Abfragen abgeholt werden. Die aus der Datenbank abgeholten Daten werden in ein einheitliches Format gebracht, das systemweit gültig ist.

Die Kategorie **Funktionen der Volltext-Schnittstelle** enthält Funktionen zum Hinzufügen und Löschen von Volltext-Indexeinträgen und zum Abrufen von Highlight-Informationen[1]. Suchabfragen an den Volltext werden immer mit den Funktionen für Datenbankabfragen formuliert. Es gibt also keine speziellen Volltext-Abfragefunktionen.

Die Kategorie **Sonstige Funktionen** enthält Hilfsfunktionen wie z.B. Funktionen zum Abrufen von reservierten Wörtern, der bevorzugten RPC-Datenblockgröße oder zum Abrufen der möglichen Feldtypen[2].

9.1.1. Verwendungsformen von Strings

Es gibt drei verschiedene Verwendungsformen von Strings in der angebotenen Schnittstelle des *DatabaseManager*. Diese werden in den folgenden Abschnitten aufgeführt.

Strings als direkte Ein-/Ausgabeparameter von Funktionen

Strings als direkte Ein- bzw. Ausgabeparameter von Funktionen haben Parameter vom Typ `p_string_t` (Eingabeparameter) bzw. `p_string_t*` (Ausgabeparameter). Der Definition des Typs `p_string_t` wird in Listing 9.9 aufgeführt.

```
1 typedef /* [string] */ unsigned char *p_string_t;
```

Listing 9.9: Definition des Datentyps `p_string_t`

[1]Das sind Informationen darüber, an welcher Position im indexierten Text Suchbegriffe gefunden worden sind.

[2]Dabei handelt es sich um die möglichen Datentypen der Merkmalsfelder von Benutzerdatenbanken, die im übrigen für alle *PROXESS*-Installationen, unabhängig vom Datenbanksystem, gleich sind.

Ein Beispiel für eine Funktion, in der ein String als Eingabeparameter verwendet wird, ist die Funktion `DbConnectDatabase`. Die Signatur dieser Funktion wird in Listing 9.10 aufgeführt.

```
1  // connect session hDbil to database pDbName.
2  // returns a connection_handle.
3  connect_handle DbConnectDatabase(
4    /* [in] */ dbil_handle hDbil,
5    /* [unique][in] */ p_string_t pDbName,
6    /* [in] */ long Privileged );
```

Listing 9.10: Beispiel: Funktion mit Eingabeparameter vom Typ `p_string_t`

Die einzige Funktion, in der ein String als Ausgabeparameter verwendet wird, ist die Funktion `FtGetVccRules`, die im Zusammenhang mit Texthervorhebung von Volltext-Suchbegriffen in Dateien verwendet wird. Die Signatur dieser Funktion wird in Listing 9.11 aufgeführt.

```
1  // get highlight marker rules.
2  void FtGetVccRules(
3    /* [in] */ dbil_handle hDbil,
4    /* [out] */ p_string_t *ppVccRules );
```

Listing 9.11: Beispiel: Funktion mit Ausgabeparameter vom Typ `p_string_t`

Arrays von Strings

Die zweite Verwendungsform von Strings ist in Form eines String-Arrays. Ein String-Array wird durch die Struktur `string_array_t` gekennzeichnet, deren Definition in Listing 9.12 aufgeführt wird.

Die einzige Funktion, die den Typ `string_array_t` verwendet, ist die Funktion `DbGetIndexedField`. Die Funktion verwendet den Typ als Ausgabeparameter und gibt in dem String-Array die Menge der in der Datenbank `pTableName` indexierten Felder zurück. Die Signatur der Funktion wird in Listing 9.13 aufgeführt.

```
// wraps up an array of strings
typedef struct string_array_t {
  long size; // element count
  long padding; // unused
  unsigned char *array[ 1 ]; // the array
} string_array_t;
```

Listing 9.12: Definition des Datentyps `string_array_t`

```
// get indexed fields of database pTableName.
void DbGetIndexedFields(
  /* [in] */ connect_handle hConnection,
  /* [in, unique] */ p_string_t pTableName,
  /* [out] */ string_array_t **ppFields );
```

Listing 9.13: Beispiel: Funktion mit Ausgabeparameter vom Typ `string_array_t *`

Strings in UnionArrays

Die dritte Verwendungsform von Strings fasst String-Parameter als Teil der Typen `union_array_t` und `header_t` zusammen. Die Strukturen bilden Datenbanktabellen ab. Die Struktur dieser Typen und ihr enger Zusammenhang mit den Datentypen des Datenbanksystems werden im folgenden Abschnitt erläutert.

9.1.2. UnionArrays und Zusammenhang zu den Datentypen der relationalen Datenbank

Es sei DBT die Menge der Datentypen des verwendeten Datenbanksystems und $t : DBT \rightarrow PDT$ eine Abbildung auf die Menge PDT. Elemente aus PDT sind Tupel (TI, DT), wobei TI ein Typ-Indikator ist und DT ein Datentyp, der in den *PROXESS*-internen Datenstrukturen definiert ist. Ein **Typ-Indikator** ist ein Enumerationswert, der genau einem *PROXESS*-internen Datentyp zugeordnet ist. Verschiedene Typ-Indikatoren können auf den selben Datentyp verweisen.

Ein **`UnionArray`** (IDL-Struktur `union_array_t`) ist ein Datentyp, der primär zur Darstellung von Tabellen der relationalen Datenbanken verwendet wird. In diesem Zusammenhang wird er zusammen mit einem `Header` verwendet.

Abbildung 9.3.: Struktureller Zusammenhang der Typen `Header` und `UnionArray`

Ein **`Header`** (IDL-Struktur `header_t`) beschreibt die Spalten einer Datenbanktabelle und besteht aus einem Array von `HeaderFields`.

Ein **`HeaderField`** (IDL-Struktur `header_field_t`) beschreibt eine einzelne Spalte `s` einer Datenbanktabelle mit dem Datentyp $dbt \in DBT$. Ein `HeaderField` hat einen Namen, der dem Namen von `s` entspricht, und einen Typ-Indikator, der dem Typ-Indikator des Bildes von dbt unter der Abbildung t entspricht.

Ein **`UnionArray`** beschreibt eine Menge von Zeilen einer Datenbank und besteht – analog zu einem `Header` – aus einem Array von `Unions`. Das `UnionArray` teilt das eindimensionale Array von `Unions` durch die Spezifikation der Anzahl der in ihm enthaltenen Zeilen (`rows`) implizit auf.

Ein **`Union`** `u` (IDL-Struktur `union_t`) beschreibt eine einzelne Zelle einer Datenbanktabelle, deren zugehörige Spalte durch das `HeaderField` hf beschrieben wird. `u` besteht aus einem Typ-Indikator ti und einem Datenwert vom Typ dt. Wenn die zu `u` gehörige Datenbankzelle einen Wert hat (SQL `NOT NULL`), so gilt $(ti, dt) = t(hf)$, andernfalls (SQL `NULL`) gilt $(ti, dt) = (8, undefined)$.

Abbildung 9.3 verdeutlicht den strukturellen Zusammenhang der Typen `UnionArray`, `Union`, `Header` und `HeaderField`. Tabelle 9.2 führt die Typzuordnung zwischen Datentypen des Datenbanksystems und *PROXESS*-internen Datentypen auf. Stehen in einer Zelle der Spalte *DBT* mehrere, durch ein Komma getrennte Werte, bedeutet dies eine Mehrfachzuordnung.

Diese feste Zuordnung von Datentypen wird vom *DocumentManager* angenommen.

DBT	*PDT*
int, smallint, tinyint	(e_long=0, long)
-	(e_int=1, long)
varchar, char, text	(e_string=2, unsigned char*)
double	(e_double=3, double)
money	(e_hyper=4, uid_t[3])
datetime	(e_datetime=5, double)
image	(e_rights_s=6, unsigned char*)
-	(e_rights_f=7, rights_t[4])
-	(e_novalue=8, <undefined>)[5]

Tabelle 9.2.: Zuordnung *t* von Datenbank-Datentypen zu *PROXESS*-internen Datentypen

Uneinheitliche Änderungen bestimmter Datentypen in der Datenbank (z.B. Spalten, die Hashes speichern bleiben vom Typ varchar und werden nicht zu nvarchar geändert) haben zur Folge, dass sich die Typ-Indikatoren für bestehende Daten ändern und daher jede Verwendung der betroffenen Spalten im *DocumentManager* an die geänderten Typ-Indikatoren angepasst werden muss. Das macht eine vorzeitige Integration des *DatabaseManager* in das bestehende System unmöglich. Die Integration wäre erst dann möglich, wenn auch der *DocumentManager* vollständig an das geänderte Schema angepasst ist. Vor diesem Hintergrund wird deutlich, warum in Kapitel 8 die Entscheidung getroffen wurde, Datentypen von Spalten – außer der einheitlichen Transition von varchar zu nvarchar – unverändert zu lassen. Der Entwurf des Unicode-basierten Interface in Abschnitt 9.1.3 zeigt, dass alle Typ-Indikatoren für bestehende Daten unverändert bleiben.

Im RPC-Interface werden die Klassen in der Abbildung als C-Strukturen (structs) realisiert und nur die Assoziationen hasHeaderField und hasUnion werden programmatisch explizit abgebildet (durch Aggregation). Das bedeutet, dass sowohl die Zugehörigkeit eines Headers zu einem UnionArray als auch die Zugehörigkeit eines HeaderFields zu einem Union aus dem Zusammenhang erschlossen werden muss.

Die Definitionen der Typ-Indikatoren und der C-Strukturen werden in den Listings 9.14,9.15 und 9.16 aufgeführt. Im Anschluss daran werden Beispiele für Funktionen gegeben, welche diese Datentypen verwenden.

```
// definitions of type indicators
#define  e_long       ( 0 )  // int32
#define  e_int        ( 1L ) // int32
#define  e_string     ( 2L ) // char*
#define  e_double     ( 3L ) // double
#define  e_hyper      ( 4L ) // int64
#define  e_datetime   ( 5L ) // double
#define  e_rights_s   ( 6L ) // char*
#define  e_rights_f   ( 7L ) // rights_t
#define  e_novalue    ( 8L ) // <undefined>
// type indicators are reffered to by field_type_t
typedef long field_type_t;
```

Listing 9.14: Definition der Konstanten für Typ-Indikatoren

```
// describes the columns of a database table
typedef struct header_t {
 long size; // number of columns
 long padding; // unused
 header_field_t field[ 1 ]; // columns
} header_t;

// describes a single column of a database table
typedef struct header_field_t {
 unsigned char name[ 256 ]; // the column's name
 field_type_t type; // the column's type indicator
 long length; // max. length for strings, else sizeof()
 long decimal; // unused
 long padding; // unused
} header_field_t;
```

Listing 9.15: Definition der Strukturen `header_t` und `header_field_t`

```
// describes a set of rows of a database table
typedef struct union_array_t {
 long rows; // number of rows
 long size; // number of cells
 union_t field[ 1 ]; // array of cells
} union_array_t;

// describes a single cell of a database table
typedef struct union_t {
 // the type indicator of the cell
 field_type_t type;
 // value of the cell
 union {
  unsigned char padding[ 12 ];
  rights_t u_rights_f;
  uid_t u_hyper;
  double u_double;
  double u_datetime;
  long u_long;
  long u_int;
  unsigned char *u_rights_s;
  unsigned char *u_string;
  } value;
} union_t;
```

Listing 9.16: Definition der Strukturen `union_array_t` und `union_t`

Bespiele für Funktionen.

An dieser Stelle werden die Kernfunktionen des *DatabaseManager*, die Funktionen `DbExecSqlCmdValue` (`union_array_t` als Eingabeparameter), `DbGetSqlResultHdr` (`header_t` als Ausgabeparameter) und `DbGetSqlResultRows` (`union_array_t` als Ausgabeparameter), aufgeführt.

Die Funktion `DbExecSqlCmdValue` führt ein parametriertes SQL-Kommando aus. Das `UnionArray` `pCmdValues` enthält die Parameter, die an das SQL-Kommando gebunden werden. Die Signatur der Funktion wird in Listing 9.17 aufgeführt.

```
1 // execute SQL-query pCmdLine.
2 // bind parameters pCmdValues to pCmdLine.
3 // returns 1 or throws an exception.
4 long DbExecSqlCmdValue(
5   /* [in] */ connect_handle hConnection,
6   /* [unique][in] */ p_string_t pCmdLine,
7   /* [in] */ union_array_t *pCmdValues );
```

Listing 9.17: Beispiel für eine Funktion mit einem Eingabeparameter vom Typ `union_array_t *`.

Die Funktion `DbGetSqlResultHdr` gibt, nachdem eine SQL-Abfrage ausgeführt worden ist, die Spalteninformation der Ergebnistabelle in einem `Header` an den Client zurück. Die Signatur der Funktion wird in Listing 9.18 aufgeführt.

```
1 // get description of the result set's columns.
2 // returns 1 or throws an exception.
3 long DbGetSqlResultHdr(
4   /* [in] */ connect_handle hConnection,
5   /* [out] */ header_t **ppResultHeader )
```

Listing 9.18: Beispiel für eine Funktion mit einem Ausgabeparameter vom Typ `header_t`.

Die Funktion `DbGetSqlResultRows` gibt eine spezifizierte Anzahl von Zeilen in einem `UnionArray` an den Client zurück. Die Signatur der Funktion wird in Listing 9.19 aufgeführt.

```
1  // get the next RequestRows rows from the result set.
2  // returns the number of retrieved rows
3  //   or throws an exception.
4  long DbGetSqlResultRow(
5    /* [in] */ connect_handle hConnection,
6    /* [in] */ long RequestRows,
7    /* [out] */ union_array_t **ppResultRows )
```

Listing 9.19: Beispiel für eine Funktion mit einem Ausgabeparameter vom Typ `union_array_t *`.

9.1.3. Entwurf der dualen Schnittstelle

Dieser Abschnitt dokumentiert den Entwurf einer dualen Schnittstelle für die angebotene RPC-Schnittstelle des *DatabaseManager*, die sowohl Codepage-basierte als auch Unicode-basierte Clients bedienen kann. Der Entwurf setzt die Verfahrensweise für Funktionen der *Windows*-API um. Dieses Verfahren wird auf die Typdefinitionen und Funktionsdeklarationen angewendet.

Definitionen der Varianten von Datentypen

Dieser Abschnitt dokumentiert die **Definition der Varianten für Datentypen** der angebotenen RPC-Schnittstelle des *DatabaseManager*. Die folgende Datentypen sind direkt von der Umstellung betroffen: Die Strukturen `p_string_t`, `string_array_t`, `header_field_t` und `union_t`.

Diese Strukturen müssen sowohl in einer Codepage-basierten Variante als auch in einer Unicode-basierten Variante definiert werden. Angelehnt an das Prinzip für Funktionsnamen derr *Windows*-API erhalten die drei Typ-Varianten einen Unicode-spezifischen, einen Codepage-spezifischen sowie einen generischen Namen. Das selbe Verfahren wird auch auf diejenigen Typen angewendet, die einen der vier obigen Typen aggregieren, also die Typen `header_t` und `union_array_t`. Die resultierenden Typ-Namen werden in Tabelle 9.3dargestellt.

Der Unterschied in der Typdefinition besteht im wesentlichen darin, dass der Typ `unsigned char*` zu `wchar_t*` geändert wird. Die generische Variante wird in Abhängigkeit des Präprozessor-Makros `AK_UNICODE` entweder als Unicode-basierte Variante oder als Codepage-basierte Variante definiert. Dies wird in Listing 9.20 am Bei-

Generischer Name	Unicode-basierter Name	Codepage-basierter Name
p_string_t	p_stringW_t	p_stringA_t
string_array_t	string_arrayW_t	string_arrayA_t
header_field_t	header_fieldW_t	header_fieldA_t
union_t	unionW_t	unionA_t
header_t	headerW_t	headerA_t
union_array_t	union_arrayW_t	union_arrayA_t

Tabelle 9.3.: Neue Namen für Datentypen an der Schnittstelle

spiel des Datentyps p_string_t verdeutlicht.

```
1  // define unicode variant
2  typedef wchar *p_stringW_t;
3  // define codepage variant
4  typedef unsigned char *p_stringA_t;
5  // define generic variant
6  #ifdef AK_UNICODE
7  typedef p_stringW_t p_string_t;
8  #else
9  typedef p_stringA_t p_string_t;
10 #endif
```

Listing 9.20: Definitionen der Varianten von Datentypen

Für die übrigen Typen werden diese Definitionen analog durchgeführt. Bei den Typen header_field_t und union_t ist jeweils eine Besonderheit zu beachten.

In der Struktur header_field_t wird das Feld name als statisches char[256] deklariert. Prinzipiell müsste dieses Feld nicht umgestellt werden, weil auf Unicode-basierte Spaltennamen verzichtet wird (siehe Kapitel 8). Damit alle String in der Schnittstelle einheitlich Unicode-Strings sind, wird dieses Feld trotzdem umgestellt. Die Deklaration wird zu wchar_t[256] geändert, was die Speichergröße zwar verdoppelt, dafür aber die gleiche Zahl an Codeunits erlaubt.

In der Struktur union_t darf das Feld u_rights_s vom Typ unsigned char* nicht geändert werden. Dieses Feld speichert diejenigen Byte-Arrays, die mit dem in Kapitel 8 beschriebenen Serialisierungsmechanismus als NUL-terminierte Codepage-Strings dargestellt werden. Das Feld u_string vom Typ unsigned char* speichert hingegen alle Strings-Spalten aus der Datenbank und wird daher in der Unicode-basierten Variante als wchar_t* deklariert. Listing 9.21 stellt die relevanten Teile der

neuen Definitionen dar.

```
typedef struct unionA_t {
 field_type_t type;
 union {
  // ...
  unsigned char *u_rights_s;
  unsigned char *u_string;
  } value;
} unionA_t;

typedef struct unionW_t {
 field_type_t type;
 union {
  // ...
  unsigned char *u_rights_s;
  wchar_t *u_string;
  } value;
} unionW_t;
```

Listing 9.21: Definitionen der Varianten des Datentyps `union_t`

Die Definitionen für die aggregierte Typen `header_t` und `union_array_t` werden ebenfalls analog durchgeführt. Das Verfahren wird in Listing 9.22 am Beispiel der Struktur `union_array_t` exemplarisch aufgeführt.

Definitionen der Varianten von Funktion

Dieser Abschnitt dokumentiert die **Definition der Varianten von Funktionen** der RPC-Schnittstelle des *DatabaseManager*. Für diejenigen Funktionen der RPC-Schnittstelle, welche einen Parameter eines neu definierten Typs haben, werden drei Varianten eingeführt. Für diese Funktionen wird die selbe Verfahrensweise für die Namensgebung angewendet, wie für die Funktionen der *Windows*-API:

1. Der Name der Unicode-Variante erhält das Suffix `W`.
2. Der Name der Codepage-basierten Variante erhält das Suffix `A`.
3. Der Name der generischen Variante erhält kein Suffix, ist selbst eine Makro-Definition, und wird durch das Präprozessor-Makro `AK_UNICODE` entweder zu der `W`-Variante oder der `A`-Variante expandiert.

```c
typedef struct union_arrayA_t {
 long rows;
 long size;
 unionA_t field[ 1 ];
} union_arrayA_t;

typedef struct union_arrayW_t {
 long rows;
 long size;
 unionW_t field[ 1 ];
} union_arrayW_t;
```

Listing 9.22: Definitionen der Varianten des Datentyps `union_array_t`

Die Funktionen ohne einen solchen Parametertyp bleiben unverändert. Die Anwendung des Verfahrens wird in Listing 9.23 am Beispiel der Funktion `DbExecSqlCmdValue` aufgeführt.

```c
long DbExecSqlCmdValueA(
  /* [in] */ connect_handle hConnection,
  /* [unique][in] */ p_stringA_t pCmdLine,
  /* [in] */ union_arrayA_t *pCmdValues );

long DbExecSqlCmdValueW(
  /* [in] */ connect_handle hConnection,
  /* [unique][in] */ p_stringW_t pCmdLine,
  /* [in] */ union_arrayW_t *pCmdValues );

#ifdef AK_UNICODE
#define DbExecSqlCmdValue DbExecSqlCmdValueW
#else
#define DbExecSqlCmdValue DbExecSqlCmdValueA
#endif
```

Listing 9.23: Definition der Varianten von Funktionen

Realisierung in der IDL-Datei

Dieser Abschnitt dokumentiert die **Realisierung der dualen Schnittstelle in der IDL-Datei**. Die Code-Beispiele aus den letzten Abschnitten haben Definitionen aus der Header-Datei aufgeführt, die vom MIDL-Compiler aus der IDL-Datei erzeugt wird.

Der MIDL-Compiler bearbeitet den Präprozessor-Output einer IDL-Datei und daher tauchen z.B. die `#ifdefs` nicht in der generierten Header-Datei auf, wenn sie direkt in die IDL-Datei geschrieben werden. Ein Work-around dafür stellt die MIDL-Direktive `cpp_qoute()` dar, mit der ein beliebiger Text in die Ausgabe-Datei (die generierte Header-Datei) geschrieben werden kann ([Mic12a]).

Der Quelltext der IDL-Datei für die Präprozessor-Direktiven wird in Listing 9.24 am Beispiel der Funktion `DbExecSqlCmdValue` aufgeführt, das die Verwendung der `cpp_quote`-Direktive verdeutlicht.

```
1 long DbExecSqlCmdValueA(...);
2 long DbExecSqlCmdValueW(...);
3 cpp_quote("#ifdef AK_UNICODE")
4  cpp_quote("#define DbExecSqlCmdValue DbExecSqlCmdValueW")
5 cpp_quote("#else")
6  cpp_quote("#define DbExecSqlCmdValue DbExecSqlCmdValueA")
7 cpp_quote("#endif")
```

Listing 9.24: Realisierung von `#define`-Direktiven in der IDL-Datei

Bei der Erzeugung der Präprozessor-Direktiven für die Varianten von String-Datentypen wird analog verfahren.

9.1.4. Implementierung der dualen Schnittstelle

Die Implementierung des *DatabaseManager* wurde an die neue RPC-Schnittstelle angepasst. Die durchgeführten Arbeitsschritte werden im Folgenden dokumentiert.

Implementierung der generierten Server-Stubs. Die vom MIDL-Compiler aus der modifizierten IDL-Datei generierten Dateien, die Header-Datei und der Server-Stub für die RPC-Schnittstelle, enthalten Definitionen für beide Varianten von String-Funktionen. Die jetzige Implementierung wird für die Unicode-basierten Varianten

von Funktionen übernommen. Für die Codepage-basierten Varianten werden Adapter-Funktionen implementiert.

Die restliche Implementierung verwendet (ohne vorherige Anpassung) die generischen Datentypen (ohne `A` oder `W` Suffix) der Schnittstelle. Damit die Datentypen mit der Unicode-basierten Variante der Schnittstelle übereinstimmen, wird das Präprozessor-Makro `AK_UNICODE` definiert.

Im folgenden wird die Anpassung der internen Implementierung an die gänderten Datentypen und die Implementierung der Adapter-Funktionen dokumentiert.

Anpassung der Implementierung an die geänderten Datentypen.

Die Implementierung des *DatabaseManager* wird an die geänderten Datentypen der Unicode-basierte Schnittstelle minimal angepasst. Das ist notwendig, damit das Programm in einer semantisch korrekten Form für die folgenden Analyseschritte vorliegt. Die Änderungen werden im Folgenden kurz zusammengefasst.

Untersuchung der Funktionen der RPC-Speicherverwaltung. Zur Allokation von Speicher für Objekte, die über RPC gesendet werden, wird eine spezifische Funktion verwendet. Die Signatur dieser Funktion, `void RpcSsAllocate(size_t size)`, ist identisch zu der Signatur von `malloc`. Im *DatabaseManager* ist für jeden Datentyp der RPC-Schnittstelle eine Wrapper-Funktion implementiert, die zuerst die benötigte Speichergröße berechnet und dann die Funktion `RpcSsAllocate` aufruft. Für alle RPC-Strings – sowohl–`p_string_t`-Objekte als auch Strings in `union_t`-Objekten und `string_array_t`-Objekten – wird die Wrapper-Funktion `SsAllocString(size_t len, p_string_t *pString)` verwendet. Listing 9.25 zeigt die ursprüngliche und die angepasste Implementierung dieser Funktion.

Konvertieren von Strings. Strings werden konvertiert, sobald die Datentypen aus der RPC-Schnittstelle an Funktionen übergeben werden, die Signaturen für Codepage-basierte Strings haben. Das Gleiche wird für die umgekehrte Richtung gemacht. Zur Konvertierung werden die Konvertierungs-Konstruktoren der Klassen `CStringW` bzw. `CStringA` verwendet.

Im Rahmen dieser Anpassung hat sich gezeigt, dass für das Bindung von Daten aus der Datenbank an Datentypen aus der RPC-Schnittstelle keine Konvertierungen not-

```
// old version
void SsAllocString(
  size_t len, p_string_t *pString){
  *pString = RpcSsAllocate(len+1);
}

// new version
void SsAllocString(
  size_t len, p_string_t *pString){
  *pString = RpcSsAllocate(sizeof(wchar_t)*(len+1));
}
```

Listing 9.25: Implementierung der RPC-Allokationsfunktion für Strings

wendig sind. Für die umgekehrte Richtung gilt das nicht. Von Clients geschickte SQL-Kommandos werden, bevor sie an die Datenbank gesendet werden, von einer Funktion bearbeitet, die keine Datentypen aus der RPC-Schnittstelle verwendet.

Implementierung der Adapter-Funktionen.

Für die Codepage-basierten Varianten werden Adapter-Funktionen geschrieben, welche die entsprechende Unicode-basierte Variante der Funktion mit den richtigen Parametertypen aufrufen.

Für Eingabe-Parameter werden für alle Datentypen werden C++-Klassen eingeführt, welche die notwendige Speicherverwaltung der C-Datentypen der RPC-Schnittstelle abkapseln. Die Wrapper-Objekte werden durch eine statische Factory-Methode erzeugt, die zuerst die Konvertierung der Codepage-basierten Strings in die UTF16-LE-Kodierung durchführt und dann ein Objekt mit den konvertierten Daten erzeugen, das den Speicher bei der Destruktion wieder freigibt. Listing 9.26 verdeutlicht das Verfahren für Eingabe-Parameter am Beispiel der Funktionen `DbExecSqlCmdValue`. Dieses Verfahren wird analog für alle Datentypen und Funktionen angewendet.

Ausgabeparameter von Funktionen werden an der RPC-Schnittstelle durch zweifache Pointer realisiert. Eine Adapter-Funktion übergibt eine Referenz auf einen Pointer an die Unicode-basierte Funktion, die so ausgeführt wird, als ob ein RPC-Client die Funktion aufrufen würde. Im Anschluss an den Funktionsaufruf wird das resultierende Objekt durch eine statische Konvertierungsfunktion in den erforderlichen Datentyp der Codepage-basierten Schnittstelle konvertiert. Dabei können Daten verloren

```
long DbExecSqlCmdValueA(
  /* [in] */ connect_handle hConnection,
  /* [unique][in] */ p_stringA_t pCmdLine,
  /* [in] */ union_arrayA_t __RPC_FAR *pCmdValues ){
  // [...] set structured exception handler function
  try{
    CWrapper_p_stringW_t wcpCmdLine =
      CWrapper_p_stringW_t(pCmdLine);
    CWrapper_union_arrayW_t uapCmdValues =
      CWrapper_union_arrayW_t::ConvertFrom(pCmdValues);
    return DbExecSqlCmdValueW(hConnection,wcpCmdLine,uapCmdValues);
  }catch(CDException &Ex){
    // [...] reset seh and raise exception
  }
  return 0;
}
```

Listing 9.26: Beispiel: Implementierung einer Adapter-Funktion mit Eingabeparametern

gehen, wenn die Unicode-Daten Zeichen enthalten, die keine entsprechung in der aktuellen System-Codepage haben. Zuletzt wird das temporäre Objekt durch den Aufruf einer Funktion freigegeben, welche die spezifische Freigabefunktion für die RPC-Speicherverwaltung aufruft.

Listing 9.27 verdeutlicht das Verfahren für Ausgabe-Parameter am Beispiel der Funktion `DbGetSqlResultRow`.

Die Betrachtung der angebotenen RPC-Schnittstelle ist jetzt abgeschlossen. Im folgenden werden die Ergebnisse der Analyse der benötigten Schnittstellen des *DatabaseManager* dokumentiert.

9.2. Untersuchung der verwendeten Schnittstellen

In diesem Abschnitt werden die vom *DatabaseManager* verwendeten Programmschnittstellen so untersucht, wie es der Teilprozess `Benötigte Schnittstellen untersuchen` des Umstellungsprozesses beschreibt.

Die Schnittstellen von intern entwickelten Modulen sollen so angepasst werden, dass sie eine Schnittstelle anbieten, die Strings in der UTF-16LE-Kodierung erwarten und

```
long DbGetSqlResultRowA(
  /* [in] */ connect_handle hConnection,
  /* [in] */ long RequestRows,
  /* [out] */ union_arrayA_t **ppResultRows){
  // [...] set seh translator function for exception handling
  try{
    union_arrayW_t *wpUArray;
    long result = DbGetSqlResultRowW(
      hConnection,RequestRows,&wpUArray);
    OutConv::Convert(wpUArray,ppResultRows);
    SsFreeConfArray(&wpUArray);
    return result;
  }catch(CDException &Ex){
    // [...] reset seh and raise exception
  }
```

Listing 9.27: Beispiel: Implementierung einer Adapter-Funktion mit Ausgabeparametern

verarbeiten. Bei der Umstellung wird davon ausgegangen, dass die verwendeten Module bereits umgestellt sind. So kann auf Fallunterscheidungen im *DatabaseManager* verzichtet werden, die je nach dem, ob ein verwendetes Modul bereits umgestellt worden ist oder eben noch nicht, zu unterschiedlichen Aktionen führen. Die eigentliche Umstellung dieser Module erfolgt in einer eigenen Iteration des Umstellungsprozesses, die zeitlich unabhängig von der Umstellung des *DatabaseManager* erfolgt.

Für externe Schnittstellen wird untersucht, ob es überhaupt eine Unicode-basierte Schnittstelle gibt, und ob diese die Kodierung UTF16-LE unterstüzt. Wenn es eine Unicode-fähige Schnittstelle gibt, so muss die Implementierung im *DatabaseManager* an diese neue Schnittstelle angepasst werden. Ggf. müssen Konvertierungen von bzw. nach UTF16-LE in die Kodierung der benötigten Schnittstelle implementiert werden.

9.2.1. *Windows*-API

Die API von *Microsoft Windows* und ihre Eigenschaften in Bezug auf die Handhabung von Strings wurden in Kapitel 5 eingeführt. Es wurde gezeigt, dass die meisten Funktionen der *Windows*-API in drei Varianten existieren:

1. Eine Variante für explizit Codepage-basierte String, die am Namen durch das Suffix `A` zu erkennen ist.

2. Eine Variante für explizit Unicode-basierte Strings, die am Namen durch das Suffix `W` zu erkennen ist.

3. Eine generische Variante, die am Namen durch das Fehlen eines Suffixes zu erkennen ist. Dieser Name definiert keine Funktion, sondern ist ein Präprozessor-Makro, das, wenn das Präprozessor-Makro `UNICODE` definiert ist, zu der `W`-Variante expandiert wird, und andernfalss zu der `A`-Variante expandiert wird.

Es wurde motiviert, warum die Unicode-basierten Varianten explizit aufgerufen werden sollen: Die generischen Varianten wurden zu einer Zeit eingeführt, als es sowohl relevante *Windows*-Varianten gab, die Unicode unterstützen (*Windows NT* ab Version *3.1*), als auch welche, die kein Unicode unterstützen (*Windows 9x/Me*). Nach [Mic10] sind die generischen Varianten von Funktionen (im Jahr 2010) nicht mehr von großer Relevanz, da jede aktuelle Version von *Windows* auf der Variante *Windows NT* basiert.

Beispiele für die aktuelle und die zukünftige Verwendung von Funktionen der *Windows*-API

An dieser Stelle wird ein Beispiel für die aktuelle Verwendung von Funktionen der *Windows*-API gegeben. Generell werden die generischen Funktionen (ohne Suffix im Namen) mit expliziten Typen (`char (const)*` bzw. `LP(C)STR`) aufgerufen. Das wird in Listing 9.28 am Beispiel der Funktion `GetComputerName` verdeutlicht.

```
unsigned long szCmptrName = MAX_COMPUTER_NAME;
char pCmptrName[szCmptrName+1];
GetComputerName(&pCmptrName, &szCmptrName);
```

Listing 9.28: Beispiel: Aktuelle Form der Aufrufe von Funktionen der *Windows*-API

Der Aufruf dieser Funktion ist nur gültig, wenn das Präprozessor-Makro `UNICODE` nicht definiert ist. Ansonsten würde die Funktion einen Unicode-String erwarten, und damit einen Parameter vom Typ `LPWSTR (wchar_t*)` haben. Listing 9.29 werden die drei möglichen Aufrufvarianten aufgeführt, welche die Datentypen konsequent verwenden.

Bei der Migration des *DatabaseManager* wird Variante 2 implementiert.

```
unsigned long szCmptrName = MAX_COMPUTER_NAME;
// var. 1 -- explicit ANSI
char pCmptrName[szCmptrName+1];
GetComputerNameA(&pCmptrName, &szCmptrName);
// var. 2 -- explicit Unicode
wchar_t pCmptrName[szCmptrName+1];
GetComputerNameW(&pCmptrName, &szCmptrName);
// var. 3 -- generic
TCHAR pCmptrName[szCmptrName+1];
GetComputerName(&pCmptrName, &szCmptrName);
```

Listing 9.29: Korrekte Alternativen für Aufrufe von Funktionen der *Windows*-API

9.2.2. Schnittstellen zu den Datenbanksystemen

Im Folgenden werden die verwendeten Schnittstellen der von *PROXESS* unterstützten Datenbanksysteme betrachtet.

Microsoft SQL Server

Der DatabaseManager verwendet den **OLE DB Provider for MSSQL Server** als Schnittstelle zu dem Datenbanksystem *Microsoft SQL Server*.

> OLE DB is a set of Component Object Model (COM) interfaces that provide applications with uniform access to data stored in diverse information sources and that also provide the ability to implement additional database services. These interfaces support the amount of DBMS functionality appropriate to the data store, enabling it to share its data. ([Mic12d])

OLE DB ist ein *COM*-basierter, generischer Adapter für beliebige Datenspeicher. Daten werden durch einen Anbieter bereitgestellt. Ein Anbieter heißt in der *OLE DB*-Terminologie *Provider*. Ein Datenspeicher kann z.B. eine Datenbank, ein E-Mail-Server oder eine Excel-Datei sein. Programme, welche die Daten eines *Providers* verwenden, heißen *Consumer*. Im vorliegenden Fall ist der *DatabaseManager* ein *Consumer* des *Providers* für das Datenbanksystem *MSSQL Server*.

Die folgende Betrachtung teilt sich in zwei Teile auf: 1) Das Senden von Daten über die *OLE DB*-Schnittstelle an das Datenbanksystem und 2) das Empfangen von Daten des Datenbanksystems über die *OLE DB*-Schnittstelle.

Senden von Daten an das Datenbanksystem. Daten vom *DatabaseManager* werden immer in Form eines parameterlosen SQL-Kommandos an das Datenbanksystem geschickt. Die dazu verwendete *OLE DB*-Schnittstelle bietet die überladene Funktion `Create` an, mit dem ein Kommando erzeugt wird, das an den *Provider* übermittelt werden soll. Eine Überladung erwartet das SQL-Kommando in einer Codepage-basierten Kodierung (Parametertyp: `LPCSTR`). Die andere Überladung erwartet das SQL-Kommando in einer Unicode-basierten Kodierung (Parametertyp: `LPCWSTR`). Das bedeutet, dass allein durch die Anpassung des Funktionsarguments die Unicode-basierte Version dieser Funktion aufgerufen wird. Die *OLE DB*-Schnittstellenfunktion wird im *DatabaseManager* in drei Funktionen (`ExecSql`, `ExecSpecialCmd`, `SqlCmdExec`) der Klasse `CDbConnection` aufgerufen.

Der *DatabaseManager* kann auch – von Clients geschickte – parametrierte SQL-Kommandos mit Parameterliste verarbeiten. Die Elemente aus der Parameterliste werden in der Methode `FilterCommand` der Klasse `CDbConnection` in das übermittelte parametrierte SQL-Kommando eingefügt. Das zusammengesetze SQL-Kommando enthält keine Parameter mehr und wird in dieser Form an das Datenbanksystem gesendet.

Clients, welche die Unicode-basierte Version des *DatabaseManager* verwenden und ein parameterloses SQL-Kommando ausführen wollen, müssen sicherstellen, dass Stringliterale in Suchkriterien für String-Felder mit dem Präfix `N` versehen werden (analog zu dem Präfix `L` für Wide-String-Literale in den Programmiersprachen C/C++).

> When dealing with Unicode string constants in SQL Server you must precede all Unicode strings with a capital letter N [...]. The "N" prefix stands for National Language in the SQL-92 standard, and must be uppercase. If you do not prefix a Unicode string constant with N, SQL Server will convert it to the non-Unicode code page of the current database before it uses the string. ([Mic07])

Empfangen von Daten aus dem Datenbanksystem. Über *OLE DB* empfange Daten sind vom Typ `void*` und der Typ der Daten wird mit Hilfe eines sog. *Type Indicator* gekennzeichnet[6] ([Micc]). In Tabelle 9.4 wird die Übersetzung zwischen *PROXESS*-internen Datentypen, den *OLE DB Type Indicator*s und den Datentypen des Datenbanksystems *MSSQL Server* aufgeführt.

[6] Das ist zumindest kein Beleg für die Eigenschaft, dass *COM* objekt-orientiertiert ist.

PROXESS-Datentyp	*OLE DB*-Datentyp	*MS SQL Server*-Datentyp
u_long	DBTYPE_I1 DBTYPE_I2 DBTYPE_I4	tinyint smallint int
u_hyper	DBTYPE_CY	money
u_double	DBTYPE_R4 DBTYPE_R8	float
u_datetime	DBTYPE_DBTIMESTAMP	datetime
u_string	DBTYPE_STR DBTYPE_WSTR DBTYPE_BSTR	char varchar text
u_rights_s	DBTYPE_BYTES	image

Tabelle 9.4.: Mapping zwischen *PROXESS*-Datentypen, OLE DB und *MS SQL Server*

Die Tabelle zeigt für die Zeilen, in denen die *PROXESS*-Datentypen `u_double` und `u_string` aufgeführt sind, keine genaue Zuordnung zwischen *OLE DB Type Indicators* und Datentypen des Datenbanksystems. Die gezeigten *OLE DB Type Indicators* werden im Quelltext des *DatabaseManager* zur Fallunterscheidung verwendet. Die in der Datenbank verwendeten Datentypen lassen sich in dieser Fällen auch durch eine Teilmenge der angegebenen *OLE DB Type Indicators* erkennen. Z.B. werden alle Datenbankstring durch den *OLE DB Type Indicator* `DBTYPE_STR` erkannt.

Der im *DatabaseManager* für Strings verwendete *Type Indicator* `DBTYPE_WST` erkennt alle Unicode-Strings in der Datenbank (Spalten vom Typ `nchar`, `nvarchar` oder `ntext`). Die Verwendung dieses *Type Indicator* geht übrigens auf dieser Arbeit vorangegangene Experimente mit Unicode-Datenbankspalten zurück.

Die Zuordnung von *PROXESS*-Datentypen zu *OLE DB Type Indicators* muss im Rahmen der Unicode-Umstellung also nicht geändert werden.

Oracle Database

Der *DatabaseManager* verwendet **OCI** (**Oracle Call Interface**) zur Kommunikation mit dem Datenbanksystem *Oracle Database*. Dieses Interface ist die elementarste Schnittstelle zu einem Oracle Datenbanksystem[7] und unterstützt alle von Oracle angebotenen Datentypen und Kodierungen, einschließlich UTF-16LE. Die Anpassung der Implementierung an diese Schnittstelle wird in dieser Arbeit nicht behandelt.

[7] Alle anderen Schnittstellen, wie z.B. ODBC, bauen auf OCI auf.

Intersystems Caché

Der *DatabaseManager* verwendet die von *Intersystems* angebotene **ODBC-Schnittstelle** zum Datenbanksystem *Intersystems Caché*. Diese Schnittstelle bietet nach [Int12] „nativen Unicode-Support“. Die Anpassung der Implementierung an diese Schnittstelle wird in dieser Arbeit nicht behandelt.

9.2.3. Schnittstelle zur Java Virtual Machine

Die Volltext-Engine im *DatabaseManager* verwaltet eine *Java Virtual Machine* (*JVM*) in der *Apache Lucene*[8] zur Volltext-Indexierung und -Suche betrieben wird. *Apache Lucene* verwendet intern die Unicode-Library *ICU4J* zur Einhaltung des Unicode-Standards. Bei allen *PROXESS* Installationen wird entweder die Java-Version 1.6 oder 1.7 eingesetzt. Nach [Ora10] unterstützt Java mindestens seit der Version 1.5 Unicode 4.0 inklusive Supplementary Characters in Form von Surrogate Pairs. Um mit der Java Runtime zu kommunizieren verwendet der *DatabaseManager* die Java Invocation API. Die Verwendung der Java Invocation API spaltet sich in zwei Teile auf:

1. **Steuerung der JVM.** Die *Java Invocation API* bietet eine Reihe von Funktionen, die dazu eingesetzt werden, die Java Runtime mit den richtigen Parametern zu starten, Java-Klassen und -Referenzen für native Threads[9] zu registrieren und die native Behandlung von Exceptions, die im Java-Code nicht behandelt worden sind. Diese Funktionen der Invocation API sind in der aktuellen Version von Java nicht Unicode-fähig. Alle String-Parameter dieser Funktionen verwenden ausschließlich den Typ `char*`[10] und interpretieren Strings immer in der Kodierung der System-Codepage des Host-Computers. Das hat z.B. zur Folge, dass Elemente aus dem `classpath` auf keinen Fall in Verzeichnissen gespeichert sein dürfen, deren Namen Zeichen enthalten, die mit der Codepage des Systems nicht kodiert werden können[11].

2. **Verwendung der Java Runtime.** Die Java Runtime wird von allen Volltext-Klassen im *DatabaseManager* verwendet, um Java-Methoden aufzurufen. Der überwiegende Teil der aufgerufenen Methoden hat String-Parameter

[8] `http://lucene.apache.org/` - Timestamp: 23.04.2012

[9] Das sind alle Threads, die nicht von der Java Runtime erstellt worden sind.

[10] Es wäre vom Datentyp her durchaus möglich, UTF-8 Strings zu verwenden; aber das wird von der JVM nicht unterstützt.

[11] Konkret ist davon das Installationsverzeichnis von *PROXESS* betroffen.

(`java.lang.String`), die Strings in der UTF-16LE-Kodierung erwarten. Aus diesem Grund wird z.Zt jeder übergebene oder zurückgegebene String in einer Konvertierungsfunktion an die im *DatabaseManager* verwendete Codepage angepasst. Während der Migration des *DatabaseManager* müssen diese Aufrufe der Konvertierungsfunktionen entfernt werden.

Die Anpassung des *DatabaseManager* beschränkt sich im Hinblick auf Java also nur auf die Verwendung der Java Runtime, genaugenommen auf das Entfernen der Konvertierungsfunktionen für Strings, die an die JVM übergeben bzw. von der JVM zurückgegeben werden. Die Steuerungsfunktionen der Invocation API für die JVM müssen unverändert beibehalten werden, weil es keine Unicode-basierten Varianten der entsprechenden Funktionen gibt.

Diese Anpassungen wurden zusammen mit der kompletten Implementierung des Volltext-Moduls durchgeführt, damit der Aufwand für die gesamte Migration abgeschätzt werden kann. Es wurden 8 Klassen mit insgesamt etwa 3000 LOC umgestellt. Die Anpassung konnte innerhalb von 1,5 Arbeitstagen durchgeführt werden.

9.2.4. Schnittstellen zu COM-Modulen

Der *DatabaseManager* verwendet vier von Akzentum entwickelte *COM*-Module. Im Grundlagenteil dieser Arbeit wurde in Kapitel 5 verdeutlicht, dass Strings in *COM*-Schnittstellen nach der Spezifikation immer Unicode-Strings – d.h. UTF-16LE-kodierte Strings – sein sollen. Die Schnittstellen aller verwendeten Module deklarieren Strings als Objekte des Typs `BSTR`. Das ist ein Datentyp, der mit einer Längenangabe versehene UTF16-LE-kodierte Strings darstellt. Im *DatabaseManager* wird die Wrapper-Klasse `_bstr_t` verwendet, um `BSTR`-Objekte abzukapseln. Diese Klasse bietet sowohl Codepage-basierte als auch Unicode-basierte Konstruktoren an. Die Strings werden also an den Schnittstellen aller *COM*-Module bereits in der richtigen Kodierung verarbeitet und müssen im Rahmen der Migration nicht angepasst werden.

Dass die COM-Module auch konform zum Unicode-Standard mit den Strings umgehen, muss für jedes einzelne Modul – in einer eigenen Iteration des Umstellungsprozesses – sichergestellt werden. Im Folgenden wird die Funktionalität der vom *DatabaseManager* verwendeten COM-Module kurz vorgestellt.

Das Modul **Akzentum.Logging** wird (von allen *PROXESS Server* Programmen) zum Erstellen von Log-Files verwendet und ist ein Adapter für die Logging-Library

log4net[12].

Das Modul **Akzentum.Smartcard** stellt ist eine Fassade für alle Funktionen zur Verschlüsselung von Daten, den sicheren Schlüsselaustausch und die Kommunikation mit Smartcards zur Verfügung. Es wird sowohl von den *PROXESS Server* Programmen als auch von denjenigen Client-Programmen, welche eine Smartcard-Anmeldung erlauben, verwendet. Im *DatabaseManager* wird dieses Modul von der Volltext-Engine verwendet, um die Zuordnung von Dokument- oder Datei-IDs zu den zugehörigen Volltext-Informationen zu verschleiern. Die Verschlüsselung von Feldern in Hochsicherheits-Datenbanken findet nicht im *DatabaseManager*, sondern im *Document-Manager* statt.

Die Module **Akzentum.OpenXML** und **Akzentum.IFilter** stellen Funktionen zur Textextraktion aus *Microsoft Office* Dokumenten bereit und werden ebenfalls ausschließlich von der Volltext-Engine verwendet.

9.3. Zusammenfassung

Für die angebotene RPC-Schnittstelle des *DatabaseManager* wurde eine duale Schnittstelle entworfen und implementiert, die sowohl Unicode-basierte Clients als auch Codepage-basierte Clients bedienen kann. Bei der Implementierung der Schnittstelle wurde deutlich, dass die Daten aus der Datenbank direkt an die Datentypen der RPC-Schnittstelle gebunden werden. Der Datenfluss von der Datenbank zur RPC-Schnittstelle des *DatabaseManager* für die Übertragung von Unicode-Strings konnte demnach mit geringen Anpassungen realisiert werden. Die umgekehrte Datenflussrichtung erfordert die Anpassung weiterer Funktionen im *DatabaseManager* sowie die Anpassung aller Clients.

Bei der Anpassung der Aufrufe von Funktionen der *Windows*-API werden die expliziten Unicode-Varianten verwendet.

Strings an COM-Schnittstellen liegen schon in der richtigen Form vor und müssen nicht angepasst werden.

Das Volltext-Modul des *DatabaseManager* wurde testweise manuell migriert. In diesem Zusammenhang wurde die Benutzung der JVM an die Verwendung von Unicode-Strings angepasst.

[12] `http://logging.apache.org/log4net/` - Timestamp: 23.04.2012

10. Analyse und Umstellung der Implementierung

Dieses Kapitel behandelt den Teilprozess `Internen Code/Implementierung migrieren` des Umstellungsprozesses. Der Name dieses Kapitels ist analog zu den anderen Anwendungen des Umstellungsprozesses gewählt worden. Dieses Kapitel befasst sich, im Gegensatz zu den anderen Kapiteln, nicht mit der konkreten Migration bestimmter Teile des Systems, sondern untersucht die automatische Erkennung von Stellen im Quelltext, die von einer Migration betroffen sind.

Abschnitt 10.1 führt die Begriffe Verwendungsmuster, Ausprägungen und Transformationsregeln ein.

Abschnitt 12.1 gibt einen Überblick über die Verwendungsmuster im *DatabaseManager*.

Abschnitt 10.3 präsentiert ein Tool, das die grundlegende Herangehensweise an die statische Analyse mit dem C++-Frontend *clang* demonstriert.

10.1. Definitionen und Beispiele

Ein **Verwendungsmuster** (`UsagePattern`) beschreibt den Zweck und den Zusammenhang, in dem ein String verwendet wird. Ein Verwendungsmuster besteht aus einem eindeutigen Bezeichner (`Identifier`), einer Beschreibung (`Description`) und einer Menge von (Verwendungs-)Varianten. Beispiele für Verwendungsmuster sind das Kopieren von Strings (`CopyString`), dass Konkatenieren von Strings (`BuildString`) oder das Vergleichen von Strings (`CompareString`). Ein Verwendungsmuster verweist immer auf ein(e) oder mehrere Deklarationen oder String-Literale.

Eine **Variante** (`Variant`) eines Verwendungsmusters ist ein Klassifizierer für eine Menge von konkreten syntaktische Ausprägungen eines Verwendungsmusters. Eine Variante besteht aus einem eindeutigen Bezeichner (`Identifier`) und einer sog. Template-Beschreibung (`TemplateDescription`).

Eine **Template-Beschreibung** beschreibt eine Menge von konkreten syntaktischen Ausprägungen (s.u.) eines Verwendungsmuster, indem String-Literale oder Bezeichner für Variablen durch Template-Bezeichner generisch beschrieben werden. Ein **Template-Bezeichner** ist ein Bezeichner, der durch ein führendes `$`-Zeichen gekennzeichnet wird.

Eine **syntaktische Ausprägung** ist eine Menge von Deklarationen und Anweisungen im Quelltext, die mit der Realisierung eines Verwendungsmusters in einem direkten Zusammenhang stehen.

Beispiel: Auswertung von Fehlern an der OLEDB-Schnittstelle

Am Beispiel eines kleinen Ausschnitts der zentralen Klasse `CDbConnection` – diese Klasse kapselt eine Verbindung zum Datenbanksystem ab – wird der Unterschied zwischen syntaktischen Ausprägungen und den Template-Beschreibungen als Teil von Varianten eines Verwendungsmusters aufgeführt.

In dem Beispiel schreiben alle Ausprägungen von Verwendungsmuster in den String `m_MsgText` vom Typ `CDynamicString` – eine individuelle String-Builder-Klasse. Die Variable `m_MsgText` wird als Member der Klasse `CDbConnection` deklariert:

```
1  class CDbConnection {
2    // ...
3    private: CDynamicString m_MsgText;
4    // ...
5  }
```

Diese Deklaration wird in der Funktion `CDbConnection::HrOLEDB_Error` verwendet. Der relevante Teil aus dieser Funktion wird im folgenden Listing aufgeführt:

```
1  void CDbConnection::HrOLEDB_Error( /* ... */ ){
2   // ... stripped param check, exception handling
3   string sSource, sErrText; // local declarations
4   // ... ole db call
5   if(ErrOLE.GetMyErrorInfo(sSource, sErrText)){
6    m_MsgText.Copy(sSource.c_str(), sSource.length());
7    m_MsgText.Cat(" - ", 3);
8    m_MsgText.Cat(sErrText.c_str(), sErrText.length());
9   }else{
10   // ...
```

```
11     }
12     // ... stripped exception handling
13   }
```

In den Zeilen `6` bis `8` finden sich Funktionsaufrufe der Klasse `CDynamicString`, die als Ankerpunkt für die Erkennung einer syntakischen Ausprägung dienen.

In Zeile `6` wird die Funktion `CDynamicString::Copy` aufgerufen. Dies ist eine Ausprägung des Verwendungsmusters `CopyString`: Der Inhalt des Strings `sSource` vom Typ `std::string` wird komplett in den String `m_MsgText` kopiert. Die Variante von `CopyString`, die diese Verwendungsform beschreibt, wird im Folgenden aufgeführt:

Verwendungsmuster: `CopyString`
Beschreibung: Der Inhalt des Strings `$srcStr` wird in den String `$dstStr` kopiert.
Variante: `Var13`
Constraints: Der Bezeichner `$dstStr` ist eine Member-Variable vom Typ `CDynamicString`. Der Bezeichner `$srcStr` ist eine lokale Variable vom Typ `std::string`.
Template-Beschreibung:

```
$dstStr.Copy($srcStr.c_str(),$srcStr.length());
```

In Zeile `7` und `8` finden sich zwei Ausprägungen des Verwendungsmusters `BuildString`. Die entsprechenden Varianten werden im folgenden aufgeführt:

Verwendungsmuster: `BuildString`
Beschreibung: Der Inhalt des Strings `$srcString` wird an den String `$dstStr` angehängt.
Variante: `Var1a`
Constraints: Die Bezeichner `$dstStr` ist eine Member-Variable vom Typ `CDynamicString`. Der Bezeichner `$srcStr` ist ein String-Literal, `$size` ist ein Integer-Literal und entspricht der Länge von `$srcStr`.
Template-Beschreibung:

```
$dstStr.Cat($srcStr, $size);
```

Verwendungsmuster: `BuildString`
Beschreibung: Der Inhalt des Strings `$srcString` wird an den String `$dstStr` angehängt.
Variante: `Var1a`
Constraints: Der Bezeichner `$dstStr` ist eine Member-Variable vom Typ `CDynamicString`. Der Bezeichner `$srcStr` ist eine Variable vom Typ `std::string`.
Template-Beschreibung:

```
$dstStr.Cat($srcStr.c_str(), $srcStr.length());
```

In der Beschreibung werden Bezeichner mit einem führenden `$`-Zeichen verwendet. Diese Bezeichner beschreiben einen String ohne Verweis auf konkrete Variablennamen, Typen oder Deklarationsformen. So wird die Beschreibung des Musters möglichst allgemein gehalten. Diese allgemeine Beschreibung wird in sog. Varianten des Verwendungsmusters weiter spezifiziert. Eine Variante klassifiziert eine Menge von syntaktischen Ausprägungen des Verwendungsmusters.

Eine **syntaktische Ausprägung** (`Manifestation`) besteht aus einer Sequenz von Anweisungen und Deklarationen, die in einem direkten Zusammenhang mit der Realisierung eines Verwendungsmusters stehen. Im folgenden Code-Listing sind mehrere syntaktische Ausprägungen verschiedener Verwendungsmuster enthalten, die unter dem Listing erläutert werden.

In Zeile `3` wird ein Objekt vom Typ `CDynamicString` – eine individuelle String-Builder-Klasse – als Member der Klasse `CDbConnection` deklariert. In Zeile `9` werden zwei loakle Variablen vom Typ `std::string` deklariert. Die Deklarationen allein sind bereits Ausprägungen des nicht-operationalen Verwendungsmusters `DeclareString`. In den Zeilen `11 - 14` finden sich Ausprägungen von der operationalen Verwendungsmuster `CopyString` und `BuildString`.

Eine **Variante eines Verwendungsmusters** (`Variant`; Kurzform: **Variante**) klassifiziert eine Menge von syntaktischen Ausprägungen (s.u.) des Verwendungsmusters.

Verschiedene syntaktische Ausprägungen, die bis auf Bezeichner und Formatierung größtenteils identisch sind, werden als **Varianten** des Verwendungsmuster abstrakt beschrieben. Zu jeder Variante eines Verwendungsmusters wird eine Musterlösung zugeordnet. Die Musterlösung eines Verwendungsmusters ist eine spezielle Variante, die den Zustand des Programmcodes nach Abschluss der Migration beschreibt. Abbildung 10.1 verdeutlicht den Zusammenhang zwischen Verwen-

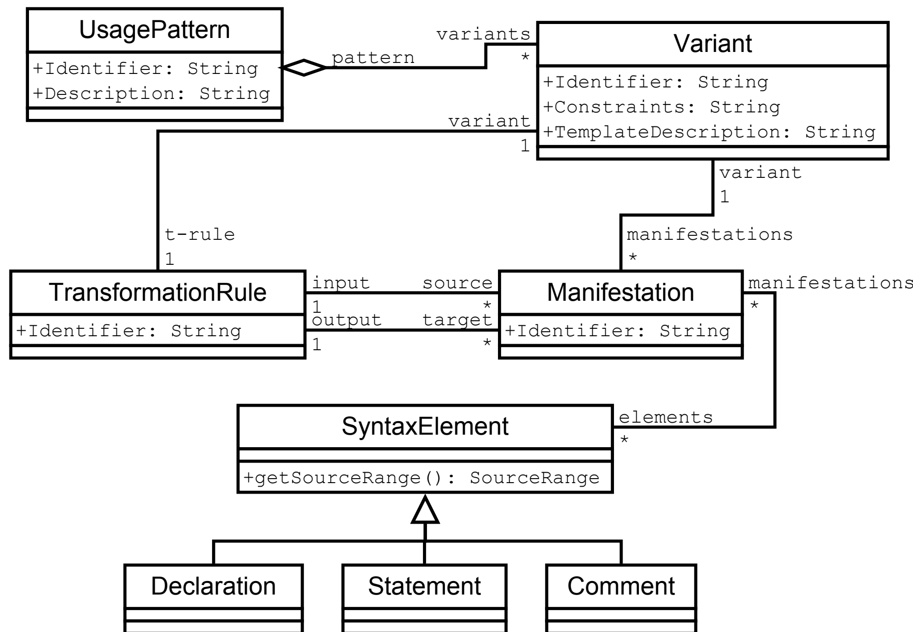

Abbildung 10.1.: Verwendungmuster und Ausprägungen

dungsmustern (`UsagePattern`), Varianten (`Variant`), syntaktischen Ausprägungen (`Manifestation`) und Musterlösungen (`SampleSolution`).

Eine syntaktische Ausprägung kann als spezifische Teilmenge des abstrakten Syntaxbaumes (AST, Abstract Syntax Tree) aufgefasst werden. Diese Teilmenge heiße `PAST` (Part of Abstract Syntax Tree). Bei `PAST` handelt es sich nicht um einen Unterbaum des AST, sondern um eine Teilmenge, die mit einem Programm-Slice vergleichbar ist: `PAST` enthält alle für die Realisierung des Verwendungsmusters relevanten Teile des Codes. Theoretisch besitzt ein Verwendungsmuster unendlich viele syntaktische Ausprägungen. Da sich viele Ausprägungen aber nur marginal unterscheiden, können Mengen von syntaktisch sehr ähnlichen Ausprägungen mit Templates für Bezeichner und/oder Literale einheitlich als Variante eines Verwendungsmusters zusammengefasst werden. Diese Anweisungen mit Platzhalten für Variablennamen und Literale heißen Templateanweisungen. In dieser Arbeit werden für Templateanweisungen gewählte Bezeichner (kurz: Templatebezeichner) mit einem führenden `$`-Zeichen notiert. Auch die zu einer Variante gehörige Musterlösung wird mit Hilfe dieser Templateanweisungen dargestellt und dabei werden die gleichen Templatebezeichner wie in der Beschreibung der Variante gewählt. Abbildung 10.2 verdeutlicht den Zusammenhang zwischen syntaktischen Ausprägungen (`Manifestation`), der zugehörigen Sequenz `PAST` von Anweisungen und Deklarationen (`SyntaxElement`) und stellt diesen die

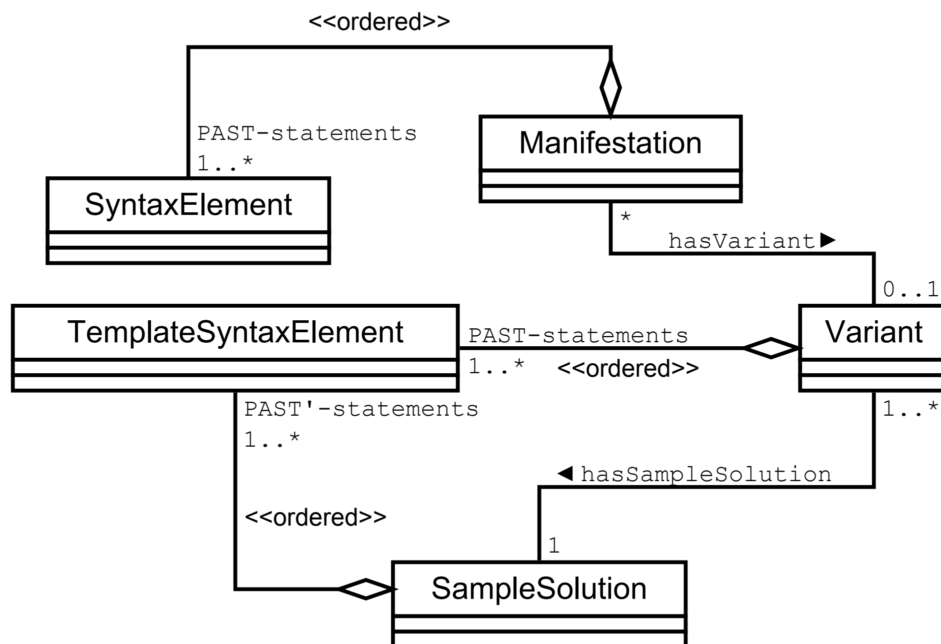

Abbildung 10.2.: SyntaxElemente, Ausprägungen und Musterlösung

Beschreibung von Varianten (`Variant`) und Musterlösungen (`SampleSolution`) mit Hilfe von Templateanweisungen (`TemplateSyntaxElement`) gegenüber.

Eine **Transformationsregel** beschreibt eine Transformation `TRANS`, die für jede syntaktische Ausprägung einer Variante eines Verwendungsmusters gültig ist. Abbildung 10.3 verdeutlicht den Zusammenhang zwischen Transformationsregeln (`TransformationRule`) und der zugehörigen Variante (`Variant`) und Musterlösung (`SampleSolution`). Die Assoziation `TransformsFrom` stellt die Beziehung zwischen einer Transformationsregel und `PAST` dar. Die Assoziation `TransformsTo` stellt die Beziehung zwischen einer Transformationsregel und `PAST'` dar.

Jede syntaktische Ausprägung `PAST` einer Variante eines Verwendungsmusters wird durch die Anwendung von `TRANS` in die modifizierte Form `PAST'` überführt. `PAST'` hat folgende Eigenschaften:

1. `PAST'` realisiert die Musterlösung der zu `PAST` gehörenden Verwendungsvariante.
2. `PAST'` ist syntaktisch korrekt.
3. `PAST'` ist semantisch äquivalent zu `PAST`.
4. `PAST'` verarbeitet Strings in der Kodierung UTF-16LE.

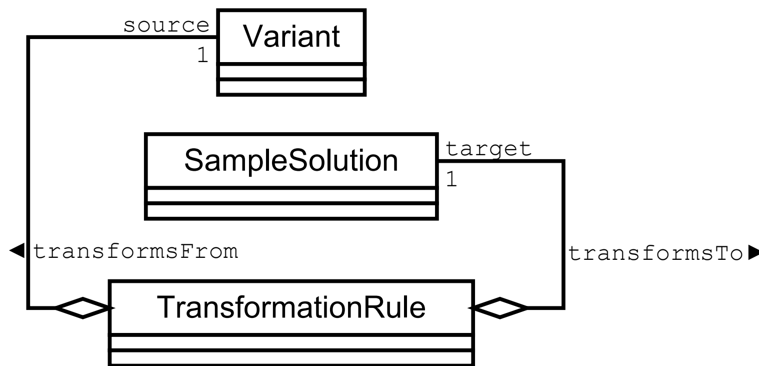

Abbildung 10.3.: Transformationsregeln und Ausprägungen

Dabei sind die Punkte 1 und 2 Anforderungen an die Transformation `TRANS`, und die Punkte 3 und 4 Anforderungen an die Gestalt der Musterlösung.

10.1.1. Textuelle Darstellung von Verwendungsmustern und Transformationsregeln

Dieser Abschnitt legt die Darstellung von Verwendungsmustern, Varianten und Transformationsregeln dar. Die Verwendung dieser Templates stellt eine einheitliche Beschreibung der Muster sicher. Stellt sich im Laufe der über diese Arbeit hinausgehenden Migration des Systems heraus, dass nicht alle Transformationsmuster entdeckt worden sind, können neue Verwendungsmuster, Varianten und Transformationsregeln mit diesen Templates beschrieben werden.

10.1.1.1. Aufbau der Beschreibung von Verwendungsmustern und Varianten

Die Beschreibung eines Verwendungsmusters wird in Abbildung 10.4 am Beispiel `CopyString` skizziert. Jedes Verwendungsmuster hat einen **Identifier**, eine **Problembeschreibung** und eine Reihe von **Varianten**. Eine Variante legt Einschränkungen für die Problemstellung fest und beschreibt die betroffene Syntax mit Templateanweisungen. Es gibt mindestens zwei Varianten, von denen zwei eine besondere Rolle einnehmen: Die Variante **VC** (**V**ariant **C**ommon) beschreibt die am häufigsten auftretende Variante; die Variante **VS** (**V**ariant **S**olution) beschreibt die Variante der Musterlösung. Die übrigen Varianten werden mit den nummerierten Bezeichnern **V1** bis **VN** identifiziert.

Name. `CopyString`
Problembeschreibung. Der Inhalt eines Strings `$srcStr` vom Typ `$srcType` wird ganz oder teilweise in einen evtl. neu allokierten String `$dstStr` vom Typ `$dstType` kopiert. Zum Kopieren wird die Funktion `$funct` verwendet.
VC. Verwendet zum Kopieren die C-Standardfunktion der `strcpy`-Familie. Die Quell- und Zieltypen sind Varianten von `char*`. D.h. der Quelltyp kann optional `unsigned char` sowie `const`-Qualified sein. Der Zieltyp kann optional `unsigned char*` sein.

```
char *$dstStr, *$srcStr;
$dstStr = new char[strlen($srcStr)+1];
strcpy($dstStr,$srcStr);
```

VS. Die Lösung setzt den Zuweisungsoperator der Klasse `CStringW` ein. Sowohl der Quell- als auch der Zieltyp ist vom Typ `CStringW`.

```
CStringW $dstStr, CStringW $srcStr;
$dstStr = $srcStr);
```

Abbildung 10.4.: Beispiel Verwendungsmuster `CopyString`

Dieses Verwendungsmuster hat insgesamt 20 Varianten, wenn allein von der Anzahl der unterschiedlichen Funktionen, Konstruktoren und Operatoren ausgegangen wird. Würde sich die Beschreibung noch in Deklarations- und Typvarianten aufteilen, würde die Menge an Varianten unverhältnismäßig umfangreich werden. Es wären unterschiedlichste Kombinationen von Funktionen mit lokalen oder globalen Deklarationen, Deklarationen von Funktionsparametern oder Klassenmembern und Typvarianten wie `unsigned char`, `unsigned char const`, `unsigned char const &` zu beachten.

10.1.1.2. Aufbau der Beschreibung für Transformationsregeln

In diesem Abschnitt wird an einem Beispiel der Aufbau der Beschreibung von Transformationsregeln verdeutlicht. Die Beschreibung für Transformationsregeln teilt die Beschreibung in zwei Teile ein:

1. **Name** – Name des Transformationsmusters. Der Name besteht aus dem Namen des Verwendungsmusters gefolgt von einem `.` gefolgt von dem Namen der Variante für die diese Transformationsregel gültig ist.
2. **Transformationsregel** – Transformation der syntaktischen Ausprägung `PAST` in die modifizierte syntaktische Ausprägung `PAST'`. Die T-Regel wird als zwei-

spaltige Tabelle dargestellt: Auf der linken Seite wird der Code von `PAST` aufgeführt. Auf der rechten Seite wird der Code von `PAST'` aufgeführt. Auf beiden Seiten werden die gleichen Bezeichner für Variablen verwendet. Steht in einer Zeile auf der rechten Seite keine Anweisung, muss die entsprechende Anweisung auf der linken Seite ersatzlos gestrichen werden. Steht in einer Zeile auf der linken Seite keine Anweisung, muss die entsprechende Anweisung auf der rechten Seite unbedingt hinzugefügt werden.

Im folgenden Beispiel wird das Transformationsmuster `CopyString.VC` beschrieben.

Name. `CopyString.VC`

Transformationsregel.

Source	Target
`char *$dstStr;`	`CStringW $dstStr;`
`char *$srcStr;`	`CStringW $srcStr;`
`$dstStr = new char[strlen($srcStr)+1];`	
`strcpy($dstStr,$srcStr);`	`$dstStr = $srcStr;`

CopyString	#
string &operator=(const string &)	85
string &operator=(const char *)	65
strcpy(char*,const char*)	61
string(const string &)	43
CStr(const unsigned char *)	41

Tabelle 10.1.: Häufigste Varianten des Verwendungsmuster `CopyString` im *DatabaseManager*

10.2. Überblick: Verwendungsmuster im *DatabaseManager*

Dieser Abschnitt beschreibt die relevanten Verwendungsmuster, die im *DatabaseManager* vorkommen. Die Menge von Varianten von Verwendungsmustern und die Anzahl der Vorkomnisse wurde aus der Ausgabe des in Abschnitt 10.3 vorgestellten Tools abgeleitet. Das Tool gibt u.a. eine XML-Datei aus, die Knoten für alle aufgerufenen Funktionen enthält. Jede Funktion ist mit der Aufrufhäufigkeit und Informationen zu den Aufrufern versehen.

Im Folgenden werden die wichtigsten Verwendungsmuster im *DatabaseManager* beschrieben. Die häufigsten Varianten werden zusammen mit ihrer Aufrufhäufigkeit in Tabellenform dargestellt.

Verwendungsmuster: `CopyString`

Beschreibung: Kopiert den Inhalt von String `$srcStr` in den String `$dstStr`.

Tabelle 10.1 gibt die fünf am häufigsten vorkommenden Varianten des Verwendungsmusters `CopyString` an, und gibt deren Aufrufhäufigkeit an.

Verwendungsmuster: `BuildString`

Beschreibung: Fügt den Inhalt von String `$srcStr` an den String `$dstStr` an.

Tabelle 10.2 gibt die fünf am häufigsten vorkommenden Varianten des Verwendungsmusters `BuildString` and, und gibt deren Aufrufhäufigkeit an.

Verwendungsmuster: `FormatString`

Beschreibung: Benutzt den Format-String `$fmt` zum Formatieren der variablen Argumentliste `...` und weist den resultierenden dem String `$dstStr` zu. Die Deklarationen aller String-Argumente müssen transformiert werden.

BuildString	#
CDynamicString::Cat(const char *)	89
char *strcat(char *,const char *)	15
LPTSTR lstrcat(LPSTR,LPSTR)	10
CDynamicString::Cat(const char*,const char*)	10
std::string::append(const char *)	5

Tabelle 10.2.: Häufigste Varianten des Verwendungsmuster `BuildString` im *DatabaseManager*

FormatString	#
_snprintf(char*,size_t,const char *,...)	103
sprintf(char*,const char *,...)	61
sprintf_s(char *,size_t,const char *,...)	12

Tabelle 10.3.: Häufigste Varianten des Verwendungsmuster `FormatString` im *DatabaseManager*

Das Verwendungsmuster `FormatString` zeigt einen Fall, in dem die Transformationsregel nicht vollständig durch die implizite Transformationsregel der Template-Beschreibung spezifiziert wird. Die Beschreibung stellt natürlichsprachlich zusätzliche Bedingungen an die Transformation, die bei der Implementierung des Verwendungsmusters berücksichtigt werden müssen. Tabelle 10.3 gibt die fünf am häufigsten vorkommenden Varianten des Verwendungsmusters `FormatString` and, und gibt deren Aufrufhäufigkeit an.

Es gibt weitere Verwendungsformen von Strings, die sich nicht auf diese einfache, auf einen Funktionsaufruf beschränkte Weise beschreiben lassen, oder für die eine solche Beschreibung, gemessen an der Aufrufhäufigkeit, nicht sinnvoll ist. Durch die drei aufgeführten Verwendungsmuster werden die am häufigsten aufgerufenen Funktion[1] abgedeckt. Die am häufigsten aufgerufenen Funktionen, die nicht durch die aufgeführten Varianten der Verwendungsmuster abgedeckt sind, werden in Tabelle 10.4 dargestellt.

Die Funktion `printf` muss nicht durch ein Verwendungsmuster abstrahiert werden, da sie die einzige Funktion ist, die einen formatierten String direkt in die Standardausgabe schreibt. Der überwiegende Teil der Aufrufe des Operators `std::ostringstream::operator<<(const char*)` finden in einer Klasse statt, und können daher besser manuell migriert werden.

Im folgenden Abschnitt wird ein Tool vorgestellt, dass die grundlegende Herangehens-

[1] Mit Ausnahme der Funktion `printf`

Weitere Funktionen	#
printf	202
std::ostringstream::operator<<(const char*)	61
GetProcAdress	36
OutputDebugString	24
_strnicmp	16

Tabelle 10.4.: Häufig aufgerufene Funktionen im *DatabaseManager*, die nicht durch ein Verwendungsmuster beschrieben werden

weise an die automatische Erkennung von Verwendungsmustern mit *clang* demonstriert.

10.3. Tool zur grundlegenden Herangehensweise an die automatische Erkennung von Verwendungsmustern mit *clang*

Dieser Abschnitt stellt das Tool `cpp-clangtool.exe` vor, das die grundlegende Herangehenweise an die automatische Erkennung von Verwendungsmustern mit *clang* demonstriert.

Das Tool wird über die Kommandozeile bedient und schreibt die Analyseergbnisse in eine XML-Datei, die mit einem separaten Viewer, dem Tool `ASTStringView.exe`, betrachtet werden kann.

Die folgenden Abschnitt beschreiben die Kommandozeilen-Parameter des Tools `cpp-clangtool.exe`, den Inhalt der XML-Ausgabe und die Verwendung des Tools `ASTStringView.exe`.

Kommandozeilen-Parameter des Tools `cpp-clangtool.exe`

- Parameter `-i` spezifiziert den absoluten Pfad der Eingabedatei. (das Programm unterstützt nur SCUs - Single Compilation Units)
- Parameter `-o` spezifiziert das Ausgabeverzeichnis
- Der Parameter `-ast-source cmdline` ist obligatorisch zu setzen.
- Der Parameter `@clang` signalisiert, dass die folgenden Parameter an das Frontend *clang* geschickt werden.

Die Datei `clangtool.cmd` auf der beiliegenden CD-ROM gibt einen Überblick über einige mögliche Kommandos.

XML-Ausgabe des Tools `cpp-clangtool.exe`

Es werden folgende Informationen in die Ausgabedatei `FullXML.log` geschrieben, die im Ausgabeverzeichnis gespeichert wird:

- Eine Liste aller Deklarationen von Stringtypen
- Eine Liste aller Funktionen mit Verweisen auf alle Statements, die String-Funktionen aufrufen, auf String-Deklarationen verweisen oder ein String-Literal verwenden.
- Zwei Callmaps.
    - Die erste Callmap stellt Aufrufe von internen Funktionen zu internen Funktionen mit String-Parametern dar.
    - Die zweite Callmap stellt Aufrufe von internen Funktionen zu externen Funktionen mit String-Parametern dar.

Die meisten XML-Elemente sind mit der entsprechenden SourceRange der Deklaration oder Anweisung verknüpft, die im Tool `ASTStringView.exe` zur Anzeige des Source-Codes verwendet werden. Der `ASTViewer` ist von der Verwendung her selbsterklärend. Es gibt eine TreeView, in der die XML-Daten angezeigt werden, und ein Hauptfenster, in dem der Source-Code angezeigt wird. Zusätzlich verfügt das Programm über einen `Open`-Button, um eine XML-Datei zu öffnen.

Die XML-Ausgabe für die Analyse des *DatabaseManager* befindet sich auf der CD-ROM in der Datei `DbilAnalysis.xml`.

Alle Dateien, einschließlich Source-Code, liegen im Unterverzerzeichnis `cpp-clang`

10.4. Zusammenfassung

In diesem Kapitel wurden die Begriffe Verwendungsmuster, Variante, Ausprägung und Transformationsregel eingeführt.

Die Begriffe sollten dazu verwendet werden, um ein Verfahren zur automatischen Erkennung und Transformation von Verwendungsmustern zu entwickeln.

Aus zeitlichen Gründen kann das Thema nicht gründlich ausgearbeitet werden. Der Inhalt wurde deswegen bewusst auf ein Minimum gekürzt.

Teil IV.

Evaluation, Fazit und Ausblick

Der Teil Evaluation, Fazit und Ausblick bewertet die Ergebnisse der Arbeit und gibt einen Ausblick auf mögliche anschließende Arbeiten. Der benötigte Aufwand für die Migration des gesamten Systems wird Anhand der durchgeführten Teilprozesse des Umstellungsprozesses abgeschätzt.

Kapitel 11 diskutiert die Aufwandsabschätzung für die Migration von *PROXESS*.

Kapitel 12 bewertet die Ergebnisse der Arbeit. Die Betrachtung teilt sich in eine Bewertung des Umstellungsprozesses selbst und eine Bewertung der Ergebnisse der statischen Code-Analyse zur automatischen Erkennung von Verwendungsmustern auf.

Kapitel 13 zieht ein abschließendes Fazit.

11. Aufwandsabschätzung

Dieses Kapitel dokumentiert die Aufwandsabschätzung für die Migration der Server-Programme von *PROXESS*. Die Abschätzung wird nur für die Server-Programme gemacht, da

1. die Konfigurationsprogramme von *PROXESS Server* oder der *PROXESS Standard Client* Programme mit einer grafischen Benutzeroberfläche sind, für die sich die Migration vermutlich deutlich komplizierter darstellen wird, und
2. die übrigen Client-Programme einen deutlich abweichenden grundlegenden Aufbau haben können, so dass eine Abschätzung nicht sinnvoll ist.

Die Abschätzung gliedert sich analog zum Umstellungsprozess in globale und lokale Teilabschätzungen.

11.0.1. Abschätzung für die globalen Teilprozesse

Die Teilprozesse `Datenschema migrieren` und `Datenmigration planen` wurden exemplarisch für das Datenbanksystem *Microsoft SQL Server* durchgeführt. Die Ergebnisse der Anwendung des Teilprozesses `Datenschema migrieren` sind maßgeblich für den auf jedes Datenbanksystem anzuwendenden Teilprozess `Datenmigration planen`.

Im Rahmen dieser Arbeit wurde das Tool *PDB2WC* implementiert, das Update-Skripts für Datenbanktabellen des Datenbanksystems *MSSQL Server* erzeugt. Für die Realisierung wurden Recherchen durchgeführt, wie das Abrufen von benötigten Metadaten der Datenbank umzusetzen ist. Damit das Tool *PDB2WC* für die Datenbanksysteme *Oracle Database* und *Intersystems Caché* angepasst werden kann, müssen die gleichen Recherchen für diese Datenbanksysteme durchgeführt und *PDB2WC* angepasst werden.

Gemessen an der Dauer der Umsetzung für das Datenbanksystem *MSSQL Server* sollte für die Recherche und Anpassung des Tools *PDB2WC* pro Datenbanksystem eine Arbeitswoche eingeplant werden.

11.0.2. Abschätzung für die lokalen Teilprozesse

Die Abschätzung für den Aufwand der lokalen Teilprozesse eines Software-Moduls beruht auf den Erfahrungen, die bei der Anwendung der lokalen Teilprozesse für das Modul *DatabaseManager* gemacht wurden.

Anpassung der angebotenen Schnittstelle

Die **Anpassung der IDL-Datei** und die **Implementierung der Server-Stubs** hat für die 46 Funktionen der RPC-Schnittstelle des *DatabaseManager* etwa einen Arbeitstag in Anspruch genommen. Hochgerechnet auf die 391 Funktionen der RPC-Schnittstelle des *DocumentManager* müssen für die Anpassung der angebotenen RPC-Schnittstelle etwa 8 - 9 Arbeitstage eingeplant werden.

Abschätzung für eine vollständig manuelle Migration der Implementierung der Module von *PROXESS Server*

Das Verfahren zur automatischen Erkennung von Verwendungsmustern wurde bisher nicht dazu verwendet, um ein praktisch anwendbares Tool zu entwickeln, das Entwickler bei der Migration unterstützt. Daher kann nur eine **sehr grobe Abschätzung** für eine **vollständig manuelle Migration** auf Basis der Migration der Volltext-Engine vorgenommen werden. Bei der Migration der Volltext-Engine wurden 3000 SLOC in 1,5 Arbeitstagen bearbeitet.

Tabelle 11.1 stellt die benötigten Arbeitstage für die Migration der Module von *PROXESS Server* als Hochrechnung auf Basis der SLOC-Messungen von *PROXESS* und einer Bearbeitungsgeschwindigkeit von 2000 SLOC pro Tag dar.

Modul	SLOC	2000 SLOC/D
DocumentManager	150.072	76 T
StorageManager	74.289	38 T
DatabaseManager (Common)	36.378	19 T
DatabaseManager (*MSSQL*)	7.465	4 T
DatabaseManager (*Oracle*)	4.837	3 T
DatabaseManager (*Caché*)	8.676	5 T

Tabelle 11.1.: Aufwandsabschätzung für die Server-Programme von *PROXESS* in Arbeitstagen

12. Bewertung der Ergebnisse

Dieses Kapitel bewertet die Ergebnisse dieser Arbeit. Die Betrachtung gliedert sich in zwei Teile.

Abschnitt 12.1 bewertet den Umstellungsprozess und dessen Anwendbarkeit.

Abschnitt 12.2 bewertet das Verfahren zur automatischen Erkennung von Verwendungsmustern.

12.1. Bewertung des Umstellungsprozesses

Dieser Abschnitt bewertet den in Kapitel 7 entworfenen Umstellungsprozess.

Die **Aufteilung der Migration in modularisierte Teilprozesse** lässt die gesamte Migration als gut strukturierte und überschaubare Menge von Teilaufgaben erscheinen.

Der Teilprozess `Unicode-basierte Versionen der angebotenen Schnittstelle erstellen` ist der einzige Teilprozess, der eine feste Reihenfolge der Arbeitsschritte zwischen unterschiedlichen Software-Modulen erfordert. Die Anwendung dieses Teilprozesses hat sich als unproblematisch herausgestellt. Nachdem die Unicode-basierte Version der angebotenen Schnittstelle des *DatabaseManager* erstellt worden ist (Dokumentation in Kapitel 9), hätte prinzipiell schon mit der Migration des *DocumentManager* oder des *StorageManager* begonnen werden können.

Die Modularisierung des Umstellungsprozesses unterstützt die Realisierung der Anforderung

- Die Migration von *PROXESS* muss nebenläufig zur Weiterentwicklung und Wartung des Systems ablaufen.

Die gleichzeitige Migration und Weiterentwicklung eines Moduls wird allerdings nicht berücksichtigt.

Durch den **Entwurf der dualen Schnittstelle** für den *DatabaseManager* wurde gezeigt, dass die Anforderung

- Die alte „ANSI"-Schnittstelle von *PROXESS* muss für eine bestimmte Zeit parallel zur neuen Unicode-Schnittstelle betrieben werden

im Einklang mit dem Umstellungsprozess realisiert werden kann.

Die Anforderung

- Die umgestellten Teile des Systems sollen mit Unit-Tests getestet werden.

hat sich als nicht-realisierbar herausgestellt.

Insgesamt stellt der Umstellungsprozess einen guten Leitfaden für die Migration eines beliebigen Software-Systems dar. Die Ergebnisse sind zufriedenstellend.

12.2. Bewertung des automatischen Erkennungsverfahrens

Dieser Abschnitt sollte den in Kapitel 10 dokumentierten Entwurf eines automatischen Erkennungsverfahrens für Verwendungsmuster von Strings bewerten.

Leider war keine Zeit für eine genaue Ausarbeitung des Themas und daher wurde der Inhalt auf ein Minimum gekürzt. Es wurden zwei Tool vorgestellt, welche die praktischen Ergebnisse der Untersuchungen demonstrieren.

13. Fazit

Das Ziel dieser Arbeit war es, die Grundlagen für die Unicode-Migration des Dokumenten-Management-Systems *PROXESS* zu legen.

Zu diesem Zweck wurden grundlegende Begriffe im Zusammenhang mit Zeichensätzen eingeführt, um

1. die Unterschiede zwischen dem Unicode-Zeichensatz und einem klassischen 8-Bit-Zeichensatz zu erarbeiten, und
2. deren Interpretation in der Zielplattform von *PROXESS*, *Microft Windows*, festzustellen.

Die aus diesen Betrachtungen gewonnenen Erkenntnisse stellen eine Grundlage für wichtige Entscheidungen in Bezug auf die Form und Handhabung von Strings in der Unicode-basierten Version von *PROXESS* dar.

Um die Durchführung der Migration zu strukturieren wurde ein modularisierbarer Umstellungsprozess entworfen, der allgemein für komplexe Software-Systeme anwendbar ist. Dieser Umstellungsprozess wurde an *PROXESS* angewendet. Dabei hat sich gezeigt, dass der Umstellungsprozess die Migration in gut nachvollziehbare und überschaubare Teilaufgaben zerlegt. Die Modularisierbarkeit des Umstellungsprozesses erlaubt es, die Migration einzelner Teile des Gesamtsystems zu großen Teilen zeitlich unabhängig voneinander durchzuführen.

Es sollte ein Verfahren zur automatischen Erkennung von Verwendungsmustern entworfen, und darauf aufbauend sollte ein Tool entwickelt werden, das die Software-Entwickler bei der Migration unterstützt. Es konnten nur die grundlegenden Begriffe eingeführt werden und ein Überblick über die vorhandenen Verwendungsmuster im *DatabaseManager* gegeben werden. Die Implementierung der automatischen Erkennung mit dem Compiler-Frontend *clang* konnte nur ansatzweise erreicht werden.

Insgesamt stellen die Erkenntnisse und Ergebnisse dieser Arbeit eine gute Grundlage für die Unicode-Migration von *PROXESS* dar.

Teil V.

Anhang

Literaturverzeichnis

[Ale79] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, New York, 1979.

[BN84] A.D. Birrell and B.J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, Vol.2 No.1:39–49, 1984. `http://www.cs.yale.edu/homes/arvind/cs422/doc/rpc.pdf` [Letzter Zugriff am 17.04.2012].

[claa] *clang - project hompage*. `http://clang.llvm.org/` [Letzter Zugriff am 19.08.2012].

[clab] *Clang Internals – The AST Library*. `http://clang.llvm.org/docs/InternalsManual.html`libast [Letzter Zugriff am 19.08.2012].

[DW12a] Mark Davis and Ken Whistler. *Unicode Standard Annex 15 – Unicode Normalization Forms*. Unicode, Inc, 2012. `http://unicode.org/reports/tr15/` [Letzter Zugriff am 05.08.2012].

[DW12b] Mark Davis and Ken Whistler. *Unicode Technical Standard 10 – Unicode Collation Algorithm*. Unicode, Inc, 2012. `http://unicode.org/reports/tr10/` [Letzter Zugriff am 05.08.2012].

[fS07] International Organization for Standardization. *9899:TC3*. ISO/IEC, 2007.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994.

[HBL02] Z. He, D. W. Bustard, and X. Liu. Software internationalisation and localisation: practice and evolution. In *Proceedings of the inaugural conference on the Principles and Practice of programming, 2002 and Proceedings of the second workshop on Intermediate representation engineering for virtual machines, 2002*, PPPJ '02/IRE '02, pages 89–94, Maynooth, County Kildare, Ireland, Ireland, 2002. National University of Ireland.

[HHSP04] James M. Hogan, Chris Ho-Stuart, and Binh Pham. Key challenges in software internationalisation. In *Proceedings of the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation - Volume 32*, ACSW Frontiers '04, pages 187–194, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[ICU12] *ICU Project - Homepage*, 2012. http://site.icu-project.org [Letzter Zugriff am 10.02.2012].

[Int98] Intersystems Corp. *Intersystems Press Releases*, 1998. http://www.intersystems.com/press/1998/encode.html [Letzter Zugriff am 31.03.2012].

[Int08] Intersystems Corp. *Intersystems Caché Documentation*, 2008. http://training.intersystems.com/tutorials/DocBook.UI.Page.cls?KEY=GCNV_R2008_2 [Letzter Zugriff am 05.06.2012].

[Int12] Intersystems Corp. *Introduction to Caché - Connectivity*, 2012. http://docs.intersystems.com/cache20121/csp/docbook/DocBook.UI.Page.cls?KEY=GIC_Connectivity [Letzter Zugriff am 23.04.2012].

[iso03] *ISO/IEC 14882:2003: Programming languages: C++*. 2003.

[Kah06] Steffen Kahle. JGraLab: Konzeption, Entwurf und Implementierung einer Java-Klassenbibliothek für TGraphen. Diplomarbeit, Universität Koblenz-Landau, Campus Koblenz, Juni 2006.

[Kam99] U. Kampffmeyer. *Dokumenten-Management*. Project Consult GmbH, 1999. http://books.google.de/books?id=QuwakDA7xeAC [Letzter Zugriff am 17.04.2012].

[Kap05] Michael Kaplan. What is the difference between big endian and little endian unicode?, 2005. http://blogs.msdn.com/b/michkap/archive/2005/02/09/369958.aspx [Letzter Zugriff am 06.04.2012].

[Kin95] Charlie Kindel. *Microsoft MSDN Library – The Rules of the Component Object Model*. Microsoft Corp., 1995. http://msdn.microsoft.com/en-us/library/ms810016 [Letzter Zugriff am 18.08.2012].

[Kru95] P. Kruchten. Architectural blueprint - the '4+1' view model of software architecture. *IEEE Software 12*, 6:42–50, 1995.

[LP08] Lindenberg and Phillips. Migrating to unicode. *W3C Internationalization*, 2008. http://www.w3.org/International/articles/unicode-migration/ [Letzter Zugriff am 17.04.2012].

[LR97] M M Lehman and J F Ramil. Metrics and laws of software evolution - the ninties view. pages 20–32, 1997.

[Mica] Microsoft Corp. *Microsoft MSDN Library – CompareString function*. http://msdn.microsoft.com/en-us/library/windows/desktop/dd317759%28v=vs.85%29.aspx [Letzter Zugriff am 18.08.2012].

[Micb] Microsoft Corp. *Microsoft MSDN Library – CompareStringEx function*. http://msdn.microsoft.com/en-us/library/windows/desktop/dd317761%28v=vs.85%29.aspx [Letzter Zugriff am 18.08.2012].

[Micc] Microsoft Corp. *Microsoft MSDN Library – Ole DB Type Indicators*. http://msdn.microsoft.com/en-us/library/windows/desktop/ms711251%%28v=vs.85%%29.aspx [Letzer Zugriff am 23.08.2012].

[Micd] Microsoft corp. *Microsoft MSDN Library – SetWindowText function*. http://msdn.microsoft.com/en-us/library/windows/desktop/ms633546%28v=vs.85%29.aspx [Letzter Zugriff am 19.08.2012].

[Mice] Microsoft Corp. *Microsoft MSDN Library – Type Indicator in a Microsoft SQL Server Provider*. http://msdn.microsoft.com/en-us/library/windows/desktop/ms714373%%28v=vs.85%%29.aspx [Letzter Zugriff am 23.08.2012].

[Mic06] Microsoft Corp. *Microsoft MSDN Library – SQL Server 2005: Working with Unicode Data*, 2006. http://msdn.microsoft.com/en-us/library/ms187828%28v=sql.90%29.aspx [Letzter Zugriff am 31.03.2012].

[Mic07] Microsoft Corp. *Microsoft Knowledge Base Article – You must precede all Unicode strings with a prefix N when you deal with Unicode string constants in SQL Server*, 2007. http://support.microsoft.com/kb/239530/en-us [Letzter Zugriff am 23.08.2012].

[Mic10] Microsoft Corp. *Microsoft MSDN Library – Working with Strings (Windows)*,

2010. http://msdn.microsoft.com/en-us/library/ff381407%28v=VS.85%29.aspx [Letzter Zugriff am 18.08.2012].

[Mic12a] Microsoft Corp. *Microsoft Dev Center – Dealing with defines in IDL Files*, 2012. http://msdn.microsoft.com/en-us/library/windows/desktop/aa366780%%28v=vs.85%%29.aspx [Letzter Zugriff am 23.08.2012].

[Mic12b] Microsoft Corp. *Microsoft MSDN Library – CAST and CONVERT (Transact-SQL)*, 2012. http://msdn.microsoft.com/en-us/library/ms187928.aspx [Letzter Zugriff am 21.08.2012].

[Mic12c] Microsoft Corp. *Microsoft MSDN Library – NormalizeString function*, 2012. http://msdn.microsoft.com/en-us/library/windows/desktop/dd319093%28v=vs.85%29.aspx [Letzter Zugriff am 18.08.2012].

[Mic12d] Microsoft Corp. *Microsoft MSDN Library – Overview of OLE DB*, 2012. http://msdn.microsoft.com/en-us/library/windows/desktop/ms718124%28v=vs.85%29.aspx [Letzter Zugriff am 23.08.2012].

[Mic12e] Microsoft Corp. *Microsoft MSDN Library – sp_rename (Transact-SQL)*, 2012. http://msdn.microsoft.com/de-de/library/ms188351.aspx [Letzter Zugriff am 21.08.2012].

[Mic12f] Microsoft Corp. *Microsoft MSDN Library - ComponentObjectModel*, 2012. http://msdn.microsoft.com/de-de/library/aa286559.aspx [Letzter Zugriff am 23.04.2012].

[Mic12g] Microsoft Corp. *Microsoft MSDN Library - CString Argument Passing*, 2012. http://msdn.microsoft.com/de-de/library/acttytz3%28v=vs.90%29 [Letzter Zugriff am 01.06.2012].

[Mic12h] Microsoft Corp. *Microsoft MSDN Library - Microsoft RPC: IDL and ACF files.*, 2012. http://msdn.microsoft.com/en-us/library/windows/desktop/aa378708%28v=vs.85%29.aspx [Letzter Zugriff am 12.03.2012].

[Mic12i] Microsoft Corp. *Microsoft MSDN Library - SQL Server 2012: Collation and Unicode Support*, 2012. http://msdn.microsoft.com/en-us/library/ms143726.aspxSupplementary_Characters [Letzter Zugriff am 06.04.2012].

[Mic12j] Microsoft Corp. *Microsoft MSDN Library - UCS-2 support in SQL Server 2000*, 2012. http://msdn.microsoft.com/en-us/library/aa276823%28SQL.80%29.aspx [Letzter Zugriff am 31.03.2012].

[Mic12k] Microsoft Corp. *Microsoft MSDN Library - Unicode in the Windows API*, 2012. http://msdn.microsoft.com/en-us/library/windows/desktop/dd374089%28v=vs.85%29.aspx [Letzter Zugriff am 06.05.2012].

[Mic12l] Microsoft Corp. *Microsoft TechNet Library - Supplementary Characters in SQL Server 2008 R2*, 2012. http://technet.microsoft.com/en-us/library/ms180942%28v=sql.105%29.aspx [Letzter Zugriff am 12.03.2012].

[Nes94] Timothy D Nestved. Migrating c code to unicode. *Dr Dobbs Journal*, 19(8):28,30,32,91–93, 1994.

[Obj11a] Object Management Group, Inc. *UML Infrastructure Specification*, 2011. http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF [Letzter Zugriff am 31.01.2012].

[Obj11b] Object Management Group, Inc. *UML Superstructure Specification*, 2011. http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF [Letzter Zugriff am 31.01.2012].

[Ora02] Oracle Corp. *Oracle9i Database Globalization Support Guide - Programming with Unicode*, 2002. http://docs.oracle.com/cd/B10500_01/server.920/a96529/ch6.htm [Letzter Zugriff am 23.04.2012].

[Ora05] Oracle Corp. *Oracle unicode database support*, 2005. http://www.oracle.com/technetwork/database/globalization/twp-appdev-unicode-10gr2-129234.pdf [Letzter Zugriff am 31.03.2012].

[Ora10] Oracle Corp. *Java Internationalization FAQ*, 2010. http://java.sun.com/javase/technologies/core/basic/intl/faq.jsp [Letzter Zugriff am 23.04.2012].

[PYZ09] Wenlin Peng, Xiaohu Yang, and Feng Zhu. Automation technique of software internationalization and localization based on lexical analysis. In *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human*, ICIS '09, pages 970–975, New York, NY, USA, 2009. ACM.

[rBFG05] Árpád Beszédes, Rudolf Ferenc, and Tibor Gyimóthy. Columbus: A reverse engineering approach. In *In STEP 2005*, pages 93–96, 2005.

[Sem] Semantic Designs. *The DMS Software Reengineering Toolkit – Project Homepage*. http://www.semdesigns.com/Products/DMS/DMSToolkit.html [Letzter Zugriff am 30.08.2012].

[Sne87] Harry M. Sneed. *Software-Management*. R. Müller, 1987. S.42, S.171.

[SR95] J. Shirley and W. Rosenberry. *Microsoft RPC programming guide*. O'Reilly & Associates, Inc., 1995.

[Ued08] Kohei Ueda. Sql server 2005 – best practices for migrating non-unicode data types to unicode. 2008. http://download.microsoft.com/download/d/9/4/d948f981-926e-40fa-a026-5bfcf076d9b9/SQL_bestpract_MigrationToUnicode.docx [Letzter Zugriff am 21.08.2012].

[Uni11] *The Unicode Standard, Version 6.0.0*, 2011. ISBN 978-1-936213-01-6.

[Whi76] J.E. White. A high-level framework for network-based resource sharing., 1976. http://tools.ietf.org/html/rfc707 [Letzter Zugriff am 06.04.2012].

Abbildungsverzeichnis

4.1. Client-/Server-Schichtenmodell 33
4.2. Beziehungen zwischen Dokumenten und Dateien 34
4.3. Datenbanken, Typen und Felder 45
4.4. Dokumenttypfelder 46
4.5. Beispielschema 47
4.6. PROXESS Komponentendiagramm 48
4.7. Überblick: Relationale Datenbanken 49

5.1. RPC-Interface-Compiler 61
5.2. RPC-Aufrufe 61

7.1. Ablauf des Umstellungsprozesses 76
7.2. Teilprozess für ein Software-Modul 80
7.3. Teilprozess für das Erstellen einer Unicode-basierten Version der angebotenen Schnittstelle 82
7.4. Teilprozess für die Untersuchung der benötigten Schnittstellen 83
7.5. Übersicht Migrationsschritte *PROXESS Server* 88

8.1. Teilprozesse `Datenschema migrieren` und `Datenmigration planen` 94
8.2. Legende für Abbildungen von Datenbanktabellen 99
8.3. Repräsentative Datenbanktabellen 99
8.4. Datenbanktabelle `SavedQueries` 102
8.5. `rights_s`-Serialisierung 104
8.6. Datenbanktabelle `Docs` 109

9.1. Teilprozesse `Unicode-basierte Version der angebotenen Schnittstelle erstellen` und `Benötigte Schnittstellen untersuchen`. 113
9.2. Schnittstellen des *DatabaseManager* 113

9.3. Struktureller Zusammenhang der Typen `Header` und `UnionArray` . . 118

10.1. Verwendungmuster und Ausprägungen 143
10.2. SyntaxElemente, Ausprägungen und Musterlösung 144
10.3. Transformationsregeln und Ausprägungen 145
10.4. Beispiel Verwendungsmuster `CopyString` 146

C.1. Legende für die Abbildungen von Datenbanktabellen. 182
C.2. Tabellen der Master-Datenbank, Teil 1 183
C.3. Tabellen der Master-Datenbank, Teil 2 184
C.4. Tabellen der *StorageManager*-Datenbank 185
C.5. Tabellen für Datei- und Dokumenttypen 187
C.6. Tabellen für Feldinformationen 188
C.7. Tabellen für Validierungsregeln 189
C.8. Tabellen für Dokumente und Dateien 190
C.9. Sonstige Tabellen 192

D.1. Hauptfenster von *PDB2WC* 195
D.2. Hauptfenster von *PDB2WC* 196
D.3. Hauptfenster von *PDB2WC* 197

Tabellenverzeichnis

3.1. Teilnormen der ISO-8859-Familie 18
3.2. Unicode Encoding Schemes 23
3.3. Vor- und Nachteile des einzelnen *UTF*s 26
3.4. Vergleich unterschiedlicher Kodierungen 29

4.1. *LOC*-Metrik 43
4.2. *SLOC* per Comment 44
4.3. Anzahl Dateien und Dateigröße 44
4.4. DocumentManager Interface 45

8.1. Unicode-Unterstützung der Datenbanksysteme 97
8.2. Unterschiede bei Datentypnamen zwischen den Datenbanksystemen. . . 103

9.1. Funktionen der angebotenen Schnittstelle des *DatabaseManager*. 114
9.2. Zuordnung *t* von Datenbank-Datentypen zu *PROXESS*-internen Datentypen 119
9.3. Neue Namen für Datentypen an der Schnittstelle 124
9.4. Mapping zwischen *PROXESS*-Datentypen, OLE DB und *MS SQL Server* 135

10.1. Häufigste Varianten des Verwendungsmuster `CopyString` im *DatabaseManager* 148
10.2. Häufigste Varianten des Verwendungsmuster `BuildString` im *DatabaseManager* 149
10.3. Häufigste Varianten des Verwendungsmuster `FormatString` im *DatabaseManager* 149
10.4. Häufig aufgerufene Funktionen im *DatabaseManager*, die nicht durch ein Verwendungsmuster beschrieben werden 150

11.1. Aufwandsabschätzung für die Server-Programme von *PROXESS* in Arbeitstagen 156

List of listings

4.1. Schematischer Ablauf bei der Verwendung von Handles 43

5.2. Kopieren von C-Style-Strings 52
5.3. Fehlerbehandlung mit C-Style-Strings 53
5.4. Beispiel für Funktionsvarianten in der *Windows*-API 56
5.5. Funktion `CompareStringEx` 58
5.6. Funktion `NormalizeString` 59
5.7. Beispiel: IDL-Datei mit Typdefinitionen und Attributen 63
5.8. Beispiel: IDL-Datei mit explizitem Binding-Handle und Context-Handle 64

9.9. Definition des Datentyps `p_string_t` 115
9.10. Beispiel: Funktion mit Eingabeparameter vom Typ `p_string_t` 116
9.11. Beispiel: Funktion mit Ausgabeparameter vom Typ `p_string_t` 116
9.12. Definition des Datentyps `string_array_t` 117
9.13. Beispiel: Funktion mit Ausgabeparameter vom Typ `string_array_t *` 117
9.14. Definition der Konstanten für Typ-Indikatoren 120
9.15. Definition der Strukturen `header_t` und `header_field_t` 120
9.16. Definition der Strukturen `union_array_t` und `union_t` 121
9.17. Beispiel für eine Funktion mit einem Eingabeparameter vom Typ `union_array_t *`. 122
9.18. Beispiel für eine Funktion mit einem Ausgabeparameter vom Typ `header_t`. 122
9.19. Beispiel für eine Funktion mit einem Ausgabeparameter vom Typ `union_array_t *`. 123
9.20. Definitionen der Varianten von Datentypen 124
9.21. Definitionen der Varianten des Datentyps `union_t` 125
9.22. Definitionen der Varianten des Datentyps `union_array_t` 126
9.23. Definition der Varianten von Funktionen 126
9.24. Realisierung von `#define`-Direktiven in der IDL-Datei 127

9.25. Implementierung der RPC-Allokationsfunktion für Strings 129
9.26. Beispiel: Implementierung einer Adapter-Funktion mit Eingabeparametern 130
9.27. Beispiel: Implementierung einer Adapter-Funktion mit Ausgabeparametern 131
9.28. Beispiel: Aktuelle Form der Aufrufe von Funktionen der *Windows*-API . 132
9.29. Korrekte Alternativen für Aufrufe von Funktionen der *Windows*-API . . 133

B.30. HelloRPCDemo.idl 177
B.31. HelloRPCDemo.acf 178
B.32. ifglobal.h 178
B.33. HelloRPC_h.h 179
B.34. HelloRPCDemoImpl.cpp 180
B.35. HelloRPCDemoClient.cpp 180

A. Interviewprotokoll zur Anforderungserhebung

Dieses Protokoll fasst die Ergebnisse aus dem Interview vom 19. Januar 2012 zusammen. Das Gespräch fand um 13 Uhr statt. Die befragten Personen waren

- Heinz **Pretz**, Geschäftsführer der Akzentum GmbH und
- Dr. Roman **Becker**, Leiter der Entwicklungsabteilung der Akzentum GmbH.

Das Gespräch dauerte 33 Minuten. Das Protokoll wurde mit den Interviewpartnern abgestimmt.

Ziel des Interview war es, die *Motivation* für eine Unicode-Umstellung und die *Anforderungen* an die Diplomarbeit aus der Sicht der Akzentum GmbH zu ermitteln.
Das Gespräch wurde mit Hilfe eines Sprachmitschnitts ausgewertet. So war es möglich, freien Redefluss entstehen zu lassen und viele Details anzuschneiden. Eine genaue Abschrift oder der Sprachmitschnitt selbst sollen in Einvernehmen mit den Interviewpartnern nicht dokumentiert werden.
Herr Pretz beleuchtete die Thematik aus einer eher unternehmerischen Sicht und Herr Becker eher aus einer entwicklungstechnischen Sicht. Allerdings haben beide Interviewpartner Argumente von beiden Sichtweisen aus vorgetragen, was das Interview zu einem belebten Gedankenaustausch machte.

Zusammenfassung

Im Folgenden werden die Interviewergebnisse anhand der gestellten Fragen[1] zusammengefasst. Die Texte unter den Fragen geben die Kernaussagen der Interviewpartner sinngemäß und in komprimierter Form wieder.

[1]Wenn sinnvoll, werden mehrere Fragen zusammengefasst. Die Fragen wurden nicht in der selben Reihenfolge gestellt. Einige technische Nachfragen, die gestellt wurden, werden nicht aufgeführt.

Warum ist das Thema Unicode für das Unternehmen wichtig? Wurden schon Schritte in Richtung einer Unicode-Umstellung unternommen? Der Bedarf an einer Unicode-Unterstützung ist in erster Linie kundengetrieben. Anwender des Systems, was i.d.R. Unternehmen sind, expandieren über den deutschsprachigen Markt hinaus. So ist insbesondere der osteuropäische Markt von Interesse, aber auch China ist ins Gespräch gekommen.
Eine Umstellung des Systems auf Unicode Stringkodierung wurde schon im Rahmen von allgemeinen Renovierungsaufgaben angedacht, ist jedoch bisher nicht strategisch verfolgt worden.
Von der entwicklungstechnischen Seite werden neue Module ggf.[2] mit dem von Microsoft definierten Datentyp `TChar`[3] implementiert, um eine zukünftige Umstellung zu erleichtern. Einen systematischen, flächendeckenden Versuch einer Umstellung hat es bisher noch nicht gegeben.

Wie wird die Unicode-Umstellung Ihrer Einschätzung nach ablaufen? Eher „am Stück" oder nebenläufig zum normalen Entwicklungsbetrieb? Beide befragte Personen machten deutlich, dass die Personalkapazitäten eine rasche Umstellung nicht möglich machen. Das Vorhaben soll *kontinuierlich* verfolgt werden, nebenläufig zur den tagesaktuellen Entwicklungen.
Wie die Umstellung genau ablaufen soll, muss im Rahmen der Diplomarbeit geklärt werden. Als sinnvoller Ansatz wurde erwägt, das System „von unten nach oben"[4] umzustellen.
Daher ist es vor allem wichtig, dass im Rahmen der Diplomarbeit *problematische Code-Stile* (insbesondere im Hinblick auf Stringkodierung) aufgedeckt werden und dass in einer Coding-Convention niedergelegt wird, wie diese problematischen Stile zu vermeiden sind. Es ist wichtig, dass die Unicode-Problematik allen Entwicklern klar wird, und dass bestehende Probleme in Zukunft schon im voraus vermieden werden.
Außerdem sollen Lösungspartner[5] von den Erfahrungen profitieren können, die Akzentum bei der Unicode-Umstellung gemacht haben wird, d.h.: Die Erfahrungen werden in Form einer Guideline an die Lösungspartner weitergegeben, damit ihre Entwicklungsabteilungen einen leichteren Einstieg bei einer evtl. Umstellung auf Unicode haben.

[2] Das heißt, wenn es sich um C/C++ Projekte handelt.
[3] `http://msdn.microsoft.com/en-us/library/cc842072.aspx`
[4] d.h. es wird mit dem *DatabaseManager* angefangen, danach kommt der *StorageManager*, dann der *DocumentManager* und dann erst die *Client-Programme*
[5] Lösungspartner sind Hersteller von Software, die auf PROXESS aufbaut.

Ist eine schrittweise Umstellung des Funktionsumfangs denkbar? Anmerkung: '*Schrittweise Umstellung*' bezieht sich hier auf eine Umstellung, die sich über mehrere Releasezyklen erstreckt.
So eine Umstellung wird nicht beabsichtigt. Es wird für eine zukünftige Version von PROXESS erstmalige und dann auch vollständige Unicode-Unterstützung beabsichtigt. Eine Auslieferung einer teilweise umgestellten Version ist irreführend, schwer nachvollziehbar und führt zu absurden Situationen. Z.B. könnte es in einer teilweise umgestellten Version Benutzer[6] geben, die chinesischen Namen haben aber nur deutschsprachige Dokumente[7] bearbeiten können.

Ist eine komplette Abkündigung der ANSI-Schnittstelle geplant? Die alte Schnittstelle soll nur für eine begrenzte Zeit gepflegt werden. Nach einem Zeitraum „von vielleicht 4 Jahren“ solle man sich von der alten Schnittstelle trennen können.
Am besten soll durch ein zusätzliches Interface sichergestellt werden, dass alte Client das neue System weiterhin verwenden können.
In jedem Fall soll nach einer gewissen Zeit nur noch die Unicode-Schnittstelle weitergepflegt werden.

Gibt es besondere Richtlinien oder Coding-Conventions, die der migrierte Code einhalten soll? Richtlinien des Unternehmens Microsoft sind für Softwareprodukte der Akzentum GmbH maßgeblich. Die ganze Entwicklungsabteilung ist darauf ausgerichtet und wird in diese Richtung weitergebildet. Gegen diese Richtlinien „anzuprogrammieren“ wird zu dem Problem führen, dass die Lösung in der Entwicklergemeinschaft nicht verfestigt werden kann.
Der bestehende Code verwendet durchgehend englische Bezeichnerwahl, auch bei Abkürzungen. Weiterhin wird von der ungarischen Namenskonvention[8] Gebrauch gemacht. Diese Namenskonvention soll, auch wenn aus der Mode gekommen, weiterhin eingehalten werden.
Die Verwendung von Unicode betreffende Konventionen sollen in der Arbeit festgelegt werden.

---

[6] Gemeint sind hier Benutzer, die in PROXESS definiert werden, um sich am System authentifizieren zu können.
[7] Hier sind die PROXESS Dokumente gemeint.
[8] http://de.wikipedia.org/wiki/Ungarische_Notation#Systems_Hungarian

Soll der Migrationsprozess auch Tests definieren, mit denen die migrierten Artefakte überprüft werden? Bei Migrationstests muss es sich um interne Tests handeln, eine teilweise umgestellte Version soll nicht an Anwender weitergegeben werden. Die migrierten Artefakte sollen mit *Unit-Tests* getestet werden. Langfristig gesehen soll das gesamte System mit Unit-Tests getestet werden können.

Ist es von Interesse, dass im Rahmen der Diplomarbeit eine Abschätzung für den Gesamtaufwand der Migration abgegeben wird? In der Arbeit soll der Aufwand für die gesamte Migration von PROXESS abgeschätzt werden.

Gibt es von Ihrer Seite schon Vorstellungen dazu, welcher Teil von PROXESS in der Diplomarbeit migriert werden soll? Kein spezieller Teil des Systems drängt sich für die Anwendung des Migrationsprozesses absolut auf. Vielmehr wurde zwischen Durchgängigkeit[9] und Vollständigkeit[10] abgewogen. Dabei wurde deutlich, dass sich für das Unternehmen ein Ansatz, bei dem ein Modul möglichst vollständig migriert wird, als interessanter darstellt. Auf Nachfrage wurde dies bestätigt: Ein vertikaler Prototyp[11] ist nur interessant, wenn es ein gravierendes Problem bei der Kommunikation auf der Netzwerkschicht gibt.

[9] Im Sinne von Datenfluss durch das gesamte System.
[10] Im Sinne von „Vollständigkeit eines Modules“ und nicht „Vollständigkeit einer Funktionalität“
[11] Also ein Prototyp, der nur einen Teil der Gesamtfunktionalität realisiert, dafür aber durch alle Schichten.

B. Beispielprogramm für Microsoft RPC

An einem einfachen Beispiel soll die Funktionsweise und Verwendung von *MSRPC* demonstriert werden. Dazu wird ein Hello-World-Programm vorgestellt, das dem Benutzer einen Gruß in einer von vier Sprachen schickt (Englisch, Deutsch, Griechisch oder Chinesisch). Das Beispiel soll *MSRPC* möglichst ähnlich verwenden wie *PROXESS* und deswegen werden *explizite Handles* und *Context-Handles* verwendet. Nachdem diese beiden Begriffe eingeführt worden sind, werden das *Interface* und die Implementierungen der *Stubs* kurz erläutert.

```
[ uuid(f2130d50-1313-1313-1313-131313130000),
  version(1.0), pointer_default(unique) ]
interface hellorpc {
 cpp_quote("#ifndef __RPCTYPE__")
 cpp_quote("#define __RPCTYPE__")
 #include "ifglobal.h"
 cpp_quote("#endif")

void StopListening( [in] handle_t hBinding );

login_handle HelloRPCRegister(
   [in] handle_t hBinding,
   [in] e_language language );

void HelloRPCUnregister(
   [in, out] login_handle *lh );

void SayHello(
   [in] login_handle lh,
   [in, string] wchar_t *name,
   [out, string] wchar_t **response);
}
```

Listing B.30: HelloRPCDemo.idl

B.0.1. Interface

Das *RPC-Interface* besteht im Wesentlichen aus einer *IDL-Datei* (*Interface Definition Language*). In der *IDL-Datei* werden alle Methoden des *Interface* definiert und es können beliebige C-Header eingebunden werden, die Typdefinitionen für Parameter- und Rückgabetypen enthalten. Des weiteren gehört zur Beschreibung eines *Interface* eine *ACF-Datei* (*Application Configuration File*), die spezifische Eigenschaften des *Interfaces* in Bezug auf die Einsatzumgebung beschreibt.

> „The purpose of dividing this information into two files is to keep the software interface separate from characteristics that affect only the operating environment.“ ([Mic12h])

Listing B.30 zeigt die *IDL-Datei* des Interface, Listing B.31 die *ACF-Datei* und Listing B.32 zeigt den in der *IDL-Datei* eingebundenen Header `ifglobal.h`. Aus diesen drei Dateien generiert der *MIDL-Compiler* (der *Interface-Compiler* von *Microsoft RPC*) eine Header-Datei für Client und Server sowie jeweils eine C-Datei, die den Client- bzw. Server-Stub enthält.

```
1 [ explicit_handle, enable_allocate ]
2 interface hellorpc { }
```

Listing B.31: HelloRPCDemo.acf

```
1 typedef [context_handle] void * login_handle;
2 typedef enum e_language {
3   e_lang_en,
4   e_lang_de,
5   e_lang_gr,
6   e_lang_zh } e_language;
```

Listing B.32: ifglobal.h

B.0.2. Implementierung der Stubs

Listing B.33 zeigt einen Auschnitt aus der generierten Datei `HelloRPC_h.h`, die sowohl vom Client als auch vom Server verwendet wird. In der Header Datei werden die in `ifglobal.h` definierte Enumeration `e_language` und die in der *IDL-Datei* enthaltenen Funktionen deklariert. Die generierten Stubs bestehen hauptsächlich aus Arrays

für das Marshalling von Parametern und Rückgabewerten. Im Folgenden wird gezeigt, wie die Stubs auf der Server- bzw. Client-Seite implementiert werden. Zusätzlich zum angegebenen Code sind einige Aufrufe zum RPC-Verbindungsaufbau notwendig. Im Server-Code muss das *Interface* bei der *RPC-Runtime* registriert werden und im Client-Code muss das registrierte *Interface* lokalisiert werden. Der entsprechende Code ist in den Listings nicht aufgeführt.

```
1  [...]
2  typedef  enum e_language {
3    e_lang_en         = 0,
4    e_lang_de         = ( e_lang_en + 1 ) ,
5    e_lang_gr         = ( e_lang_de + 1 ) ,
6    e_lang_zh         = ( e_lang_gr + 1 ) } e_language;
7  [...]
8  login_handle HelloRPCRegister(
9    /* [in] */ handle_t hBinding,
10   /* [in] */ e_language language );
11 [...]
12 void SayHello(
13   /* [in] */ login_handle lh,
14   /* [string][in] */ wchar_t *name,
15   /* [string][out] */ wchar_t **response );
```

Listing B.33: HelloRPC_h.h

Im Server-Stub bleiben die Interface-Funktionen undefiniert, so dass die Funktionen vom Anwendungsprogrammierer implementiert werden müssen. Listing B.34 zeigt die Server-Implementierung für die Funktionen `HelloRPCRegister` und `SayHello`. Die Klasse `Session` ist eine typische C++-Klasse, die eine Methode `SayHello` enthält. Sie gehört zum Server und nicht zu irgendeinem Teil von *RPC*. Der Aufruf der Funktion `midl_user_allocate` in Zeile 15 dient zur Speicherallokation für die *RPC-Runtime*. Jeder *RPC*-Server bzw. -Client muss diese Funktion implementieren. Üblicherweise führt sie ein einfaches `malloc` aus. Analog gibt es eine Funktion `midl_user_free`, die üblicherweise `free` aufruft und die zur Freigabe von Speicher für die *RPC*-Runtime verwendet wird.

Im generierten Client-Stub sind die Interface-Methoden bereits definiert, so dass der Anwendungsentwickler die Interface-Funktionen direkt aufrufen kann. Listing B.35 zeigt, wie der Client-Stub verwendet wird.

```
// implements interface method
login_handle HelloRPCRegister( handle_t hBinding,
   e_language language) {
   // create new session
   Session *pSession = new Session(language);
   // return context handle
   return (login_handle)pSession;
}
[...]
void SayHello( login_handle lh,
   wchar_t *name, wchar_t **response ) {
   // convert param 'name' to wstring
   std::wstring wstrName( name );
   // delegate call to session
   std::wstring *greeting = ((Session*)lh)->SayHello(wstrName);
   // convert return value and assign out parameter
   *response = (wchar_t*)midl_user_allocate(
      sizeof(wchar_t)*(str->length()+1));
   std::copy(str->begin(), str->end(), *response);
   (*response)[str->length()] = L'\0';
   delete str;
}
```

Listing B.34: HelloRPCDemoImpl.cpp

```
// aquire session handle
login_handle lh = HelloRPCRegister(GetRpcBinding(), e_lang_gr);
// say hello
wchar_t *res;
SayHello(lh, L"Sebastian", &res );
std::cout << res << std::endl;
// unregister session
HelloRPCUnregister(&lh);
```

Listing B.35: HelloRPCDemoClient.cpp

C. Auflistung der Datenbanktabellen von *PROXESS*

Dieses Kapitel dokumentiert die Analyse aller von *PROXESS* verwendeten Datenbanktabellen. Im Hauptteil der Arbeit wurden in Kapitel 8 die Umstellungsalternativen, die sich aus dieser Analyse ergeben haben, diskutiert.

Die Tabellen werden auf Spalten untersucht, die von einer Umstellung betroffen sind. Bei der Betrachtung werden, wie schon im Hauptteil, die Datentypnamen von *Microsoft SQL Server* verwendet. Jede Tabelle wird durch eine Abbildung sowie durch einen kurzen Text beschrieben. Die von einer Umstellung betroffenen Spalten werden in der Abbildung farblich hervorgehoben: Spalten, die einen expliziten Datentyp für Strings haben (Datentypen `char`, `varchar` oder `text`) werden rot unterlegt. Spalten, die Binärdaten speichern (Datentyp `image`) werden blau unterlegt, wenn sie dazu verwendet werden, um Strings zu speichern. Andernfalls werden die Spalten für Binärdaten grau unterlegt. Abbildung C.1 stellt eine Legende für die Abbildungen der Datenbanktabellen dar.

Im folgenden werden die Tabellen jeder Datenbank in einem eigenen Abschnitt behandelt. Abschnitt C.1 behandelt die Tabellen der *Master*-Datenbank, Abschnitt C.2 die Tabellen der *StorageManager*-Datenbank und in Abschnitt C.3 die Tabellen für eine relationale Datenbank für eine *Benutzerdatenbank* von *PROXESS*.

C.1. Tabellen der Master-Datenbank

Die *Master*-Datenbank enthält 9 Tabellen, in denen globale (system-weite) Konfigurationsdaten enthalten sind. Diese Tabellen werden in den Abbildungen C.2 und C.3 dargestellt. Die Tabelle `NextUID` (Abbildung C.2) speichert lediglich die nächste zu vergebende systemweit eindeutige ID und muss bei einer Umstellung nicht beachtet werden. In mehreren Tabellen wird die Spalte `Datahash` vom Typ `varchar` deklariert.

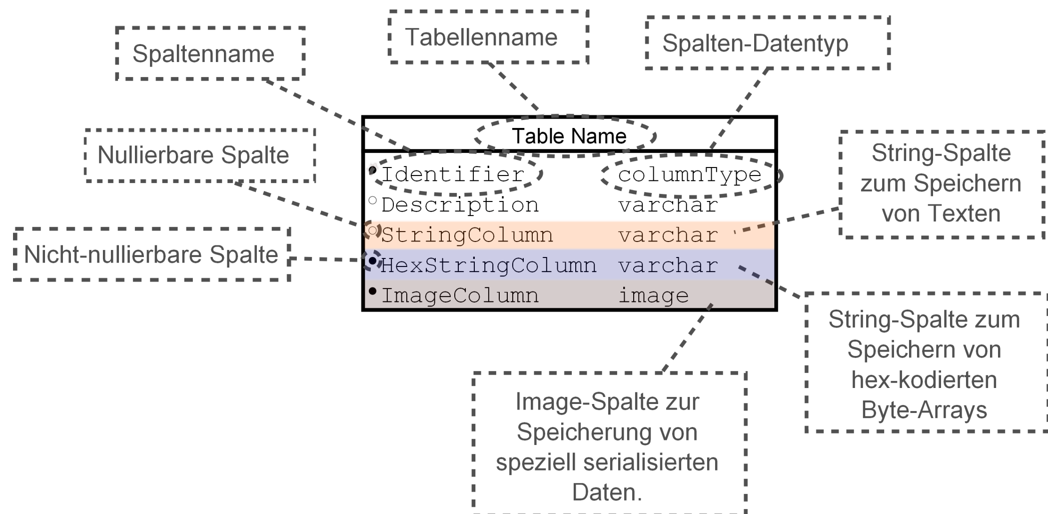

Abbildung C.1.: Legende für die Abbildungen von Datenbanktabellen.

Sie speichert eine Hash-Signatur des Datensatzes und wird als hexadezimal kodierter String gespeichert. Aus diesem Grund muss diese Spalte keinen Unicode unterstützen, und es ist abzuwägen, ob die Spalte umgestellt wird, um den Umstellungsprozess zu vereinfachen[1] oder ob sie in dieser Form erhalten bleibt, um eine Verdoppelung des benötigten Speicherplatzes zu vermeiden. Diese Überlegung gilt für jedes Auftreten der `Datahash`-Spalte und daher wird sie in der Betrachtung der einzelnen Tabellen lediglich mit einem Verweis auf diesen Absatz erwähnt.

Die Tabelle `Databases` (Abbildung C.2) enthält einen Eintrag für jede definierte Benutzer-Datenbank. Die Attribute `DBName` und `Description` sind beide von einer Umstellung betroffen. Das Attribut `DBName` speichert den Namen der Datenbank und das Attribut `Description` speichert eine Beschreibung der Datenbank. Beide Spalten müssen Unicode-Strings unterstützen. Den Datenbanknamen als Unicode-String zu speichern ist sinnvoll, weil alle unterstützten Datenbanksystem – zumindest teilweise – Unicode-fähige Systemtabellen verwenden (siehe Abschnitt 9.2.2).

Die Tabelle `Users` sowie die Tabelle `Groups` (beide in Abbildung C.2) enthalten die Definitionen von Benutzern bzw. Gruppen von *PROXESS*. Bei beiden Tabellen muss die Spalte `Name`, in der der Name des Benutzers/der Gruppe gespeichert wird und in der Tabelle `Users` zusätzlich die Spalte `ShortName`[2] umgestellt werden. Die Spalte `Passwd` enthält einen Hash des Benutzerpasswortes, der hexadezimal als String ko-

[1] Es kann auf eine Fallunterscheidung verzichtet werden.
[2] `ShortName` ist der Login-Name eines Benutzers.

NextUID	
• ID	money

Databases	
• DBName	varchar
○ Description	varchar
○ AllowedForGroups	image
• Datahash	varchar
• DocSign	int

Users	
• UserID	money
• ShortName	varchar(15)
○ Name	varchar(63)
• Passwd	varchar(47)
○ AcctDisabled	int
○ PasswdChgd	datetime
○ PasswdNExp	int
○ PTrans	int
• Datahash	varchar(41)

Certificates	
• RecordID	int
• SerialNo	varchar(40)
• FingerPrint	varchar(100)
• Issuer	varchar(255)
• IssueDate	datetime
• OwnerID	money
• CreatorID	money
• CertStatus	int
○ CertStatusText	varchar(255)
• CertAct	int
• DataHash	varchar(41)

Groups	
• GroupID	money
• Name	varchar(31)
○ Description	varchar(63)
• Datahash	varchar(41)

UserGroups	
• UserID	money
• GroupID	money
• Datahash	varchar(41)

Abbildung C.2.: Tabellen der Master-Datenbank, Teil 1

diert wird, und muss daher nicht umgestellt werden.

Die Tabelle `UserGroups` (Abbildung C.2) beinhaltet lediglich Zuordnungen von Benutzer-IDs zu Gruppen-IDs und muss bei einer Umstellung nicht weiter beachtet werden.

In der Tabelle `Certificates` (Abbildung C.2) werden die registrierten Zertifikate[3] gespeichert. Die Spalte `Issuer` speichert den Namen des Aussteller des Zertifikates und sollte deshalb umgestellt werden. Die Spalten `SerialNo` und `Fingerprint` sind hexadezimal-kodierte Byte-Arrays und müssen nicht umgestellt werden.

Die Tabelle `UserIni` (Abbildung C.3) speichert Profileinstellungen von Benutzern oder Gruppen für beliebige Client-Programme. Die Spalte `IniSection` identifiziert das Client-Programm und die Spalte `IniKey` die Programmeigenschaft, der in der Spalte `IniValue` ein Wert zugewiesen wird. Die Spalte `IniValue` ist auf jeden Fall von einer Umstellung betroffen, weil hier u.a. Dokumenttypnamen oder Datenbanknamen gespeichert werden. Die anderen Spalten enthalten nur system-interne Namen und müssen nicht umgestellt werden.

[3] X.509-Zertifikate `http://de.wikipedia.org/wiki/X.509` [31.03.2012]

CProfiles	
◦UserGroupId	money
•Modulename	varchar(100)
•VProduct	int
•VModule	int
•Revision	int
•Datahash	varchar(50)
•DataXml	text

UserIni	
•UserID	money
•IniSection	varchar(31)
•IniKey	varcahr(210)
◦IniValue	varchar(255)

CProfileGroupSeq	
•UserId	money
•Modulename	varchar(100)
◦VProduct	int
•VModule	int
•GroupId	money
•GRank	int

Abbildung C.3.: Tabellen der Master-Datenbank, Teil 2

Die Tabelle `CProfiles` (Abbildung C.3) speichert ebenfalls Profileinstellungen von Benutzern/Gruppen für beliebige Client-Programme und ist eine neue Variante der `UserIni`-Tabelle. Für die Spalte `Modulename` gilt das gleiche, wie für die Spalten `IniValue` und `IniKey` der Tabelle `UserIni`. Die Spalte `DataXml` enthält die XML-Konfigurationsdaten für das Benutzerprofil. Die XML-Daten sind immer als *UTF-16* deklariert, werden aber z.Zt. in einem 8-Bit weiten Textfeld (Datentyp `text`) gespeichert. Im Rahmen der Umstellung sollte diese Diskrepanz behoben werden und die Konfigurationsdaten in einem 16-Bit weiten Textfeld (Datentyp `ntext`) abgelegt werden. Die Tabelle `CProfileGroupSeq` (Abbildung C.3) legt eine Rangordnung für Profile fest, falls mehrere Profile für einen Benutzer definiert sind.

C.2. Tabellen der StorageManager-Datenbank

Die relationale Datenbank des *StorageManager* speichert hauptsächlich Informationen über den Speicherort von physischen Dateien. Abbildung C.4 zeigt die Tabellen dieser Datenbank. Die Tabelle `Files` enthält die Spalte `Db`, die auf die Datenbank verweist, die das Dokument speichert, zu dem eine Datei gehört. Die Tabelle `Packs` speichert die definierten CD/DVD-Brenner und die Spalte `Name` speichert einen system-interne Namen des Brenners. Diese Spalten sind also von der Umstellung betroffen. Die übrigen `varchar`-Spalten enthalten nur interne Informationen, wie z.B. Signatur[4] (die

[4] Für Dateien aus Hochsicherheitsdatenbanken.

Vols

•	VolID	money
•	Copy	int
•	Type	int
•	KeepFor	int
•	TotalKB	int
•	Location	char(30)
•	State	int
○	Source	money
•	sides	tinyint
○	MigVolId	money

VolsExpired

•	VolID	money
•	Copy	int
•	Type	int
•	KeepFor	int
•	TotalKB	int
•	Location	char(30)
•	State	int
○	Source	money
•	Sides	tinyint
○	MigVolId	money

NextVolID

•	LastId	money

OldPath

•	FileId	money
○	Path	varchar(255)

Packs

•	Name	varchar(32)
•	Id	varchar(36)
•	Capacity	smallint

Files

•	FileId	money
○	VolId	money
○	Db	char(16)
•	SizeKB	int
•	Cached	tinyint
○	HasPath	tinyint
○	LastUsed	money
○	Side	tinyint
○	SizeB	money
○	SignCnt	char(64)
○	SignLoc	char(64)
○	SignStd	char(64)

Abbildung C.4.: Tabellen der *StorageManager*-Datenbank

Spalten der Tabelle `Files`, die mit dem Präfix `Sign` beginnen), Laufwerksbuchstaben der Brenner (die Spalte `Location` in den Tabellen `Vols` und `VolsExpired`) oder alte Migrationspfade (die Spalte `Path` in der Tabelle `OldPath`). Für diese Tabellenspalten besteht also kein Bedarf an einer Umstellung des Datentyps.

C.3. Tabellen einer Benutzer-Datenbank

Für jede Benutzer-Datenbank in *PROXESS* gibt es eine relationale Datenbank. Jede dieser Datenbanken enthält 15 Tabellen, die im folgenden grob kategorisiert werden, um den Überblick zu erleichtern. Es gibt Tabellen für Metadaten, die nur für die Benutzer-Datenbank gültig sind. Darunter fallen Tabellen für Dokument- und Dateitypen, Felddefinitionen, Validierungsregeln für Felder und Vorlagendateien und Editoren für bestimmte Dateitypen. Diese Tabellen werden in den drei Abbildungen C.6, C.5 und C.7 dargestellt. Daneben steht eine Reihe von Tabellen, die Laufzeit- oder Instanzdaten repräsentieren. Darunter fallen Tabellen für Dokumente, Dateien oder gespeicherte Suchabfragen. Diese Tabellen werden in den Abbildungen C.8 und C.9 dargestellt.

C.3.1. Tabellen für Metadaten

Die Tabelle `DocType` enthält zwei `varchar`-Spalten, zum einen die Spalte `DocTypeName`, die den Namen des Dokumenttyps speichert und daher ein Umstellungskandidat ist, und zum anderen die Spalte `Datahash`, für die die selbe Überlegung wie für die `Datahash`-Spalten in Metadaten-Tabellen gilt. Für die zwei Spalten mit dem Datentyp `image`, nämlich `DocTypACL` und `CollectionFormat` gelten die allgemeinen Überlegungen zur Umstellung von `image`-Spalten.

Die Tabelle `DataTypes` speichert Dateitypdefinitionen und enthält die `varchar`-Spalten `DataTypeDes` und `FileExtension`. Die Spalte `DataTypeDes` speichert eine Beschreibung des Dateityps (z.B. „Adobe Portable Document Format“, „Log-Datei“ oder „Microsoft Word 2007 Dokument“) und ist daher ein Umstellungskandidat. Die Spalte `FileExtension` speichert die Windows-Dateierweiterung (z.B. „pdf“, „log“ oder „docx“) und ist nur dann von der Umstellung betroffen, wenn auch Unicode-Dateinamen unterstützt werden sollen.

Die Tabelle `Editors` speichert Beschreibungs- und Aufrufinformationen für Programme mit denen Dateien eines bestimmten Dateityps angesehen, editiert oder gedruckt werden können. Die Spalte `EditorDes` speichert eine textuelle Beschreibung des

DocType	
• DocTypeID	money
• DocTypeName	varchar(31)
○ DocTypeACL	image
○ CollectionFlag	int
○ CollectionFormat	image
○ DefaultKeepForDays	int
○ DefaultMediaType	int
○ FullText	int
○ OCRFlag	int
• Datahash	varchar(41)
○ CryptFlag	int

DataTypes	
• DataTypeID	money
○ DataTypeDes	varchar(63)
○ TemplateFileID	money
○ FullText	int
○ FileExtension	char(15)
○ OCRFlag	int
○ OCROutput	int
○ TGeneric	int

KeepForTimes	
• DocTypeID	money
• DataTypeID	money
○ KeepForDays	int
○ MediaType	int

Editors	
• DataTypeID	money
• EditorType	int
• Platform	int
○ EditorDes	varchar(63)
• InvocationMethod	int
• CommandLine	varchar(255)
○ DdeString	varchar(255)
○ ServiceName	varchar(31)

TemplateFiles	
• TemplateFileID	money
• TemplateFileDes	varchar(63)
○ TemplateVolID	money

Abbildung C.5.: Tabellen für Datei- und Dokumenttypen

Editor-Programms und ist daher ein Umstellungskandidat. Die übrigen `varchar`-Spalten `CommandLine`, `DdeString` und `ServiceName` werden dazu benötigt um das Programm aus *PROXESS* heraus aufzurufen. Diese Spalten sind Umstellungskandidaten, wenn Editor-Programme mit Unicode-Programmnamen oder generell Unicode-Datei- oder Pfadnamen unterstützt werden sollen.

Die Tabelle `TemplateFiles` speichert einen Verweis auf eine Vorlagendatei, die bei Neuanlage einer Datei eines bestimmten Dateityps verwendet werden soll. Diese Tabelle besitzt ebenfalls eine `varchar`-Spalte, `TemplateFileDes`, in der eine Beschreibung der Vorlagendatei gespeichert wird. Daher ist diese Spalte ebenfalls ein Umstellungskandidat.

Die Tabelle `KeepForTimes` speichert die Lebensdauer (d.h. die Archivierungsfrist) für Dateitypen in Bezug auf einen bestimmten Dokumenttyp. Diese Tabelle enthält keine `varchar`- oder `image`-Spalten und kann daher bei einer Unicode-Umstellung unverändert beibehalten werden.

Tabellen für Felddefinitionen. Die Tabellen `Fields` und `FieldRoles` in Abbildung C.6 enthalten Felddefinitionen für Benutzerdatenbanken (in der Tabelle `Fields`) bzw.

Fields	
● PhysFieldName	varchar(47)
○ ValidationRuleID	money
● DefaultFieldName	varchar(47)
○ DefaultFieldDes	varchar(63)
○ DefaultEditMask	varchar(255)
● Datatype	int
● Length	int
● Digits	int
○ FieldX	int
○ FieldY	int
○ FieldWidth	int
○ FieldHeight	int
○ LabelX	int
○ LabelY	int
○ LabelWidth	int
○ LabelHeight	int
○ Page	int
○ TabOrder	int
○ Visible	int
○ FieldID	int
○ Mandatory	int
○ CryptFlag	int
● Datahash	varchar(41)

FieldRoles	
● PhysFieldName	varchar(47)
● DocTypeID	money
○ ValidationRuleID	money
○ FieldRoleName	varchar(47)
○ FieldRoleDes	varchar(63)
○ EditMask	varchar(255)
○ FieldX	int
○ FieldY	int
○ FieldWidth	int
○ FieldHeight	int
○ LabelX	int
○ LabelY	int
○ LabelWidth	int
○ LabelHeight	int
○ Page	int
○ TabOrder	int
○ Visible	int
○ FieldID	int
○ Mandatory	int
○	

Abbildung C.6.: Tabellen für Feldinformationen

ValidationRules	
•ValidationRuleID	money
•ValidationType	int
∘ValidationDes	varchar(63)

ValidationWordlists	
•ValidationRuleID	money
∘UserUpdateAllowed	int
∘DefinedByPhysField	varchar(47)
∘Length	int

ValidationRanges	
•ValidationRuleID	money
∘DateTimeLower	datetime
∘DateTimeUpper	datetime
∘DoubleLower	float
∘DoubleUpper	float
∘IntLower	int
∘IntUpper	int

ValidationWords	
•ValidationRuleID	money
•ValidationWord	varchar(119)
∘IfDefiningFieldContains	varchar(119)

Abbildung C.7.: Tabellen für Validierungsregeln

von Dokumenttypen überschriebene Felder (in der Tabelle `FieldRoles`). In beiden Tabellen bezeichnet die Spalte `PhysFieldName` den für die Datenbank eindeutigen Namen des Feldes. In der Tabelle `Docs` (siehe Abschnitt C.3.2) existiert eine Spalte gleichen Namens in der die Wertzuweisungen für jedes Dokument gespeichert werden. Die Spalte `PhysFieldName` muss Unicode-Feldnamen speichern können und ist daher von der Umstellung betroffen. Das gleiche gilt für die Spalten `DefaultFieldName` und `DefaultFieldDes` (bzw. `FieldRoleName` und `FieldRoleDes`), die optional einen anderen Anzeigenamen[5] und eine textuelle Beschreibung für das Feld festlegen. Die Spalte `Length` in der Tabelle `Fields` gibt die Länge eines String-Feldes in Zeichen an. Das muss bei der Umstellung ebenfalls berücksichtigt werden. Die `varchar`-Spalten `DefaultEditMask` bzw. `EditMask` sind für ein Feature hinzugefügt worden, dass nicht implementiert worden ist. Diese Spalten können also ignoriert werden.

Tabellen für Validierungsregeln für Felder. Die Tabelle `ValidationRules` fasst alle Eingaberegeln für Felder zusammen. Es gibt z.B. Bereichseinschränkungen für `int`-, `float`- und `datetime`-Felder, die in der Tabelle `ValidationRanges` definiert werden oder Einschränkungen in Form von Listen von erlaubten Wörtern für `string`-Felder, die in den Tabellen `ValidationWords` und `ValidationWordlists` definiert werden. In der Tabelle `ValidationRules` ist die Spalte `ValidationDes`, die eine textuelle Beschreibung der Validierungsregel enthält, umzustellen. Die Tabelle `ValidationRanges` enthält gar keine Spalten, die von einer Umstellung betroffen sind. In der Tabelle `ValidationWords` ist die Spalte `ValidationWord` umzustellen,

[5] Für grafische Benutzeroberflächen.

Docs	
• DocID	money
∘ DocTypeID	money
∘ DocDes	varchar(63)
∘ CreateDate	datetime
∘ DocACL	image
∘ Creator	money
∘ UpdateDate	datetime
∘ Updater	money
∘ KeepForDays	int
∘ DocsDocTypeName	char(31)
∘ DocState	int
∘ Datahash	varchar(41)
∘ SecData	text

SeeAlso	
• SourceDoc	money
∘ TargetDoc	money

FilesCheckOut	
• FileID	money
• CheckOutPath	varchar(255)
• CheckOutHost	varchar(32)
∘ CheckOutComment	varchar(255)

Files	
• FileID	money
• DataTypeID	money
• DocID	money
• FileDes	varchar(63)
∘ VolID	money
∘ Creator	money
∘ CreateDate	datetime
∘ MinorVerNo	int
∘ MajorVerNo	int
∘ FileComment	varchar(250)
∘ ActualFileID	money
∘ CheckOutDate	datetime
∘ CheckOutBy	money
∘ CheckInDate	datetime
∘ CheckInBy	money
∘ FileState	int
∘ ExternInfo	varchar(250)
∘ ExternId	varchar(63)
∘ Datahash	varchar(41)
∘ FCrypt	int

Abbildung C.8.: Tabellen für Dokumente und Dateien

weil in dieser Spalte ein möglicher Feldwert festgelegt wird.

Die Spalten `IfDefiningFieldContains` in der Tabelle `ValidationWords` sowie die Spalte `DefinedByPhysField` in der Tabelle `ValidationWordlists` können ignoriert werden, weil sie für ein nicht implementiertes Feature von *PROXESS* eingeführt worden sind.

C.3.2. Tabellen für Laufzeit- oder Instanzdaten

Tabellen für Dokumente und Dateien. Abbildung C.8 zeigt die `Docs`-Tabelle, in der alle Dokumente einer Benutzer-Datenbank gespeichert werden. Die Abbildung zeigt nur die Spalten, die für Dokumente in allen Benutzer-Datenbanken gleich sind. Dazu zählen ein systemweit eindeutiger Identifier, die Kernfelder, die individuellen Zugriffsrechte oder der Dokumentenstatus[6]. In der Abbildung werden keine Felddefinitionen

[6] Der Status gibt an, ob das Dokument aus dem System gelöscht worden ist. Gelöschte Dokumente bleiben (zunächst) in der Datenbank bestehen.

angegeben, da diese für jede Benutzer-Datenbank spezifisch sind. Spezifische Felder können vom Typ `int`, `float`, `datetime` oder `(var)char` sein. Diese werden in der Tabelle `Fields` definiert und ggf. in der Tabelle `FieldRoles` überschrieben.

Beide Spalten vom Typ `varchar`, nämlich `DocDes` (Titel des Dokuments) und `DocsDocTypeName` (Name des Dokumenttyps) müssen umgestellt werden. Für die Spalte `Datahash` gelten die vorangegangenen Überlegungen, ebenso wie für die Spalte `DocACL`, in der individuelle Zugriffslisten für das Dokument abgelegt werden.

Mit der Tabelle `SeeAlso` werden Verknüpfungen zwischen Dokumenten realisiert, indem die Identifier der beiden Dokumente verknüpft werden. Da diese Tabelle lediglich zwei Spalten für die beiden Identifier enthält, muss sie bei der Umstellung nicht beachtet werden.

In der Tabelle `Files` werden Metadaten für alle Dateien von Dokumenten gespeichert. Dazu gehören ein Verweis auf das Dokument, zu dem die Datei gehört und auf das Volume, in dem die physische Datei gespeichert ist, ein Zeitstempel für das Erstellungsdatum, eine zweistufige Nummerung für die Dateiversion und Check-Out- und Check-In-Informationen[7]. Von einer Umstellung betroffen sind die `varchar`-Spalten `FileDes` (der Titel der Datei) und `FileComment` (eine textuelle Beschreibung der Datei), nicht akut betroffen sind die Spalten `ExternInfo` und `ExternId`, die nur in speziellen Szenarien zur Interoperabilität eingesetzt werden.

Die Tabelle `FilesCheckOut` speichert weitere Informationen für jede Datei, die z.Zt. „aus-gecheckt“ ist. Die Spalte `CheckOutHost` speichert den Namen des Computers, auf dem die Datei aus-gecheckt wurde und muss nur dann umgestellt werden, wenn Unicode-Computernamen unterstützt werden sollen. Die Spalte `CheckOutPath` speichert das lokale Verzeichnis, in dem die Datei aus-gecheckt wurde und muss nur dann umgestellt werden, wenn Unicode-Dateinamen unterstützt werden. Die Spalte `CheckOutComment` speichert einen Benutzerkommentar und muss auf jeden Fall umgestellt werden.

Sonstige Tabellen. In der Tabelle `SavedQueries` werden beliebige von Benutzern gespeicherte Suchabfragen abgelegt. Die Spalte `QueryName` speichert den Anzeigenamen der gespeicherten Suchabfrage und muss umgestellt werden. Die Spalte `OptionsParams` wird intern zur Speicherung von Anzeigeoptionen verwendet und

[7] Das Check-Out entspricht dem „Lock“ und das Check-In entspricht dem „Write“ in „Lock-Modify-Write“ beim exklusiven Bearbeiten einer Datei (siehe Abschnitt 4.2).

SavedQueries	
• QueryID	money
• QueryName	varchar(63)
• Usersgroups	image
• SqlWhereComponent	image
◦ FormatString	image
• CreatorID	money
◦ Options	int
◦ OptionsParams	varchar(255)

DefaultFormat	
• UserID	money
◦ FormatString	image

LogInfo	
• EType	int
• EDate	datetime
◦ DocTypeID	money
• DocID	money
◦ FileID	moeny
◦ DFDes	varchar(63)
• UserID	money
• UserName	varchar(15)
◦ ClientType	varchar(70)

Folders	
• FolderID	money
• FName	varchar(31)
◦ FDescription	varchar(63)
◦ Groups	image
◦ Field	varchar(47)
◦ AllowedChars	varchar(255)
◦ ConstLevels	int

Registers	
• RegisterID	money
• FolderID	money
• RName	varchar(31)
◦ RDescription	varchar(63)
• SQLWhereComponent	varchar(255)

Abbildung C.9.: Sonstige Tabellen

muss nicht umgestellt werden. Die Spalte SqlWhereComponent vom Typ image enthält die konkrete Suchabfrage. Da die Suchabfrage durchaus Unicode-Feldnamen und Unicode-Feldwerte enthalten kann, muss entweder der Datentyp geändert werden oder der Serialisierungsmechanismus so angepasst werden, dass er Unicode-Strings verarbeiten kann.

In der Tabelle Registers werden vorkonfigurierte Suchabfragen gespeichert. Sie werden in Ordnern (Folders) gruppiert. Die Spalten für den Namen und die Beschreibung beider Tabellen FName, FDescription, RName und RDescription müssen umgestellt werden. In der Tabelle Registers ist ebenfalls die Spalte SQLWhereComponent betroffen, die – obwohl sie einen anderen Datentyp hat – genau so wie die Spalte SqlWhereComponent in der Tabelle SavedQueries verwendet wird. Die Spalten Field und AllowedChars in der Tabelle Folders müssen ebenfalls umgestellt werden.

In der Tabelle LogInfo werden Löschungen von Dokumenten und Dateien protokolliert. Zu einer Löschung wird immer die Beschreibung (bzw. Titel) des gelöschten Dokumentes oder der gelöschten Datei festgehalten (in der Spalte DFDes) und optional der Name des Benutzers, der die Löschung ausgeführt hat (in der Spalte UserName) und/oder der Name des Client-Programms mit dem die Löschung durchgeführt wurde (in der Spalte ClientType). Die Spalten DFDes und UserName sind also von einer Umstellung betroffen, die Spalte ClientType nicht.

Die Tabelle DefaultFormat ist für ein Feature eingeführt worden, dass nicht implementiert worden ist und kann daher bei der Umstellung ignoriert werden.

D. Verwendung des Tools *PROXESS - Database2WideChar*

Dieses Kapitel erläutert die Verwendung des im Rahmen dieser Arbeit entwickelten Tools *PROXESS - Database2WideChar.* Abbildung D.1 zeigt das Hauptfenster von *PDB2WC.*

Bedienung des Hauptfensters

Nachdem eine Verbindung zum Datenbanksystem hergestellt wurde (s.u.), werden im oberen Panel in der linken Liste die verfügbaren Datenbanken angezeigt. In der Abbildung sind die Datenbanken `ProxessDB` (die Master-Datenbank von *PROXESS*), `SMDB` (die Verwaltungsdatenbank des *StorageManager*) und die zwei Benutzerdatenbanken `ArchivDB` (eine normale Benutzerdatenbank) und `SecurDB` (eine Hochsicherheitsdatenbank) zu erkennen.

Sobald eine Datenbank ausgewählt wird, werden in der rechten Liste alle Tabellen dieser Datenbank angezeigt. In der Abbildung sind einige Tabellen der Master-Datenbank `ProxessDB` zu sehen, darunter die ausgewählte Tabelle `Databases` in der die Benutzerdatenbanken von *PROXESS* definiert sind.

Sobald eine Datenbank ausgewählt ist kann durch einen Klick auf den Button `Create Update Script` ein Update-Skript erzeugt werden. Ist keine Tabelle ausgewählt, wird ein Update-Skript für alle Tabellen der ausgewählten Datenbank erzeugt. Ist zusätzlich eine Tabelle ausgewählt, wird das Update-Skript nur für diese Tabelle erzeugt. In der Abbildung wird das erzeugte Update-Skript für die Tabelle `Databases` angezeigt. Es sei auch an dieser Stelle angemerkt, dass *PDB2WS* z.Zt. nur für Datenbanken des Systems *Microsoft SQL Server* implementiert ist.

Der Button `Execute Update Script` ist dazu gedacht, das Skript direkt auszuführen. Diese Funktion ist nicht implementiert. Um das Skript auszuführen, muss der Text markiert werden und z.B. im *SQL Server Management Studio* ausgeführt werden.

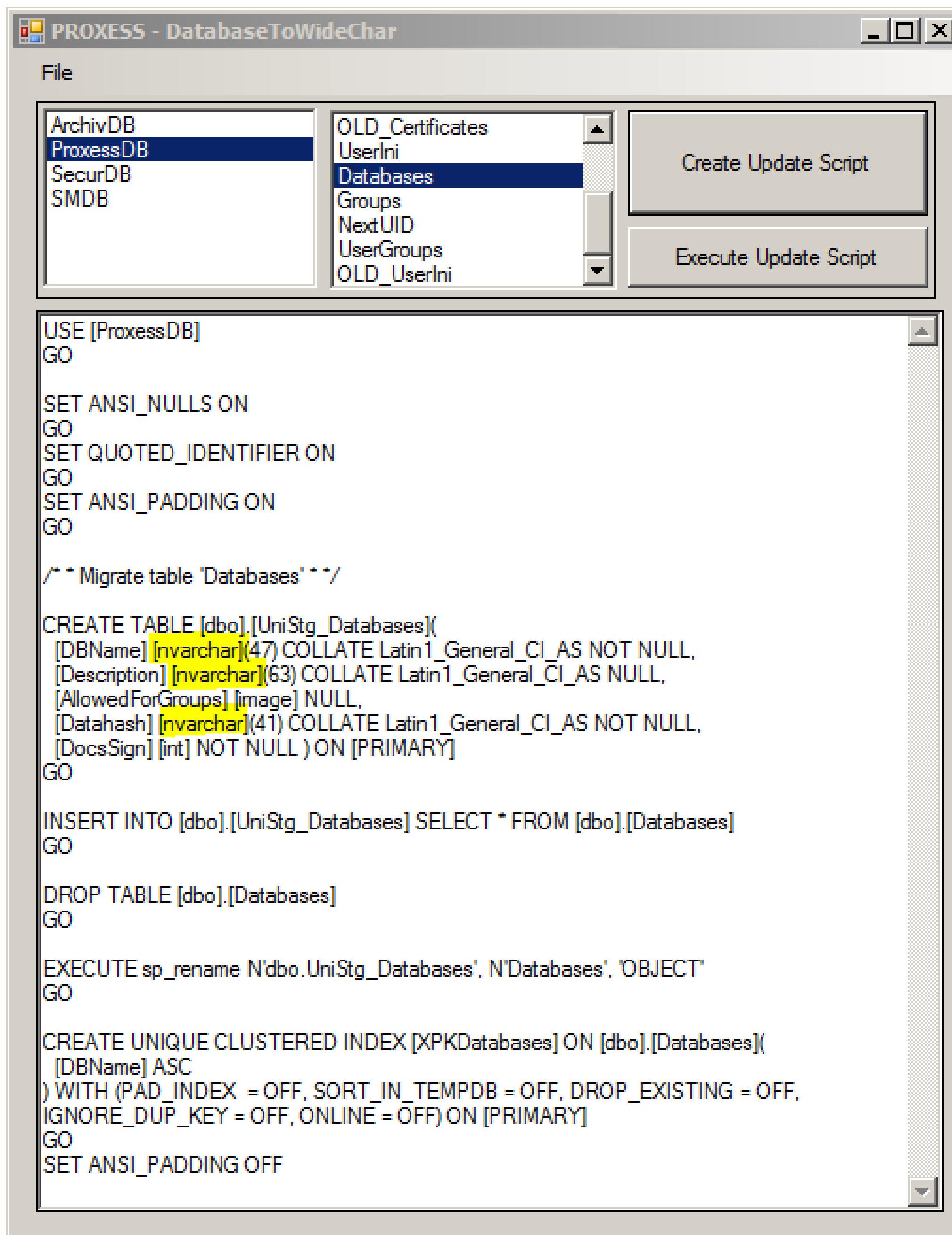

Abbildung D.1.: Hauptfenster von *PDB2WC*

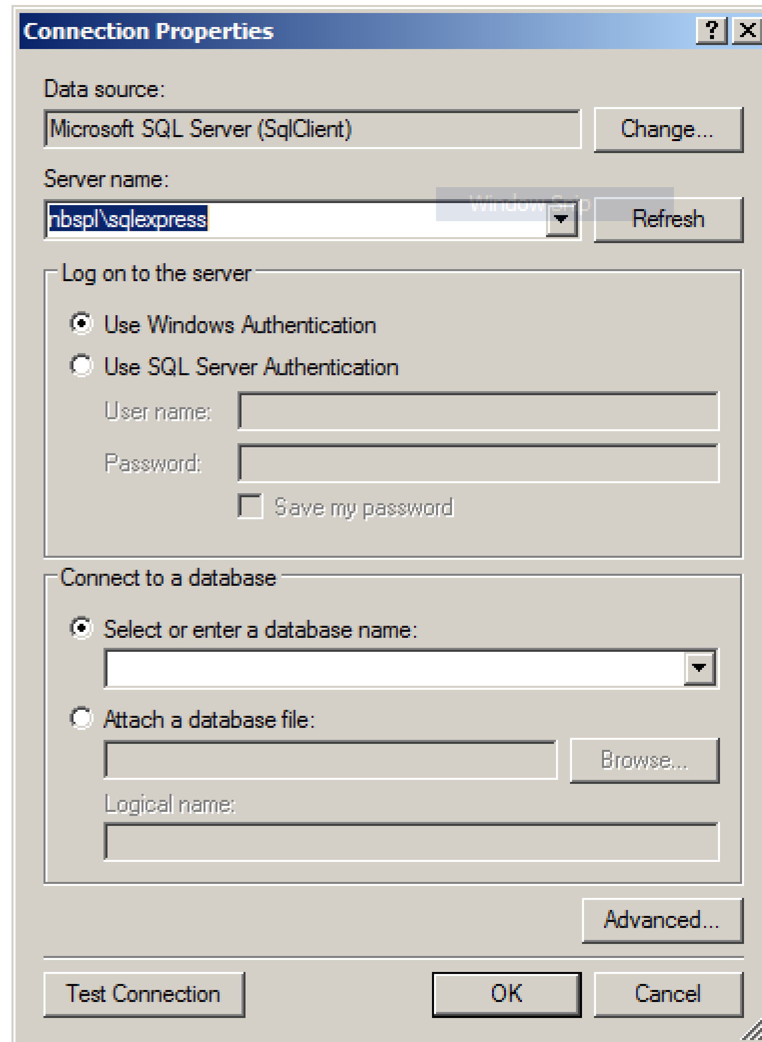

Abbildung D.2.: Hauptfenster von *PDB2WC*

Verbindungsdialog

Der in Abbildung D.2 dargestellt Verbindungsdialog stammt von *Microsoft* und wird unter der *Microsoft Public License* zur kostenlosen Nutzung veröffentlicht und zum Download angeboten[1]. Der Dialog öffnet sich bei der Auswahl des Menüpunktes `File -> Connect`.

Bei dem Dialog handelt sich um einen generischen Verbindungsdialog für viele Arten von Datenquellen. In der Abbildung wird die Dialogvariante für die Datenquelle *Mi-*

[1] urlhttp://archive.msdn.microsoft.com/Connection [Letzter Zugriff am 21.08.2012]

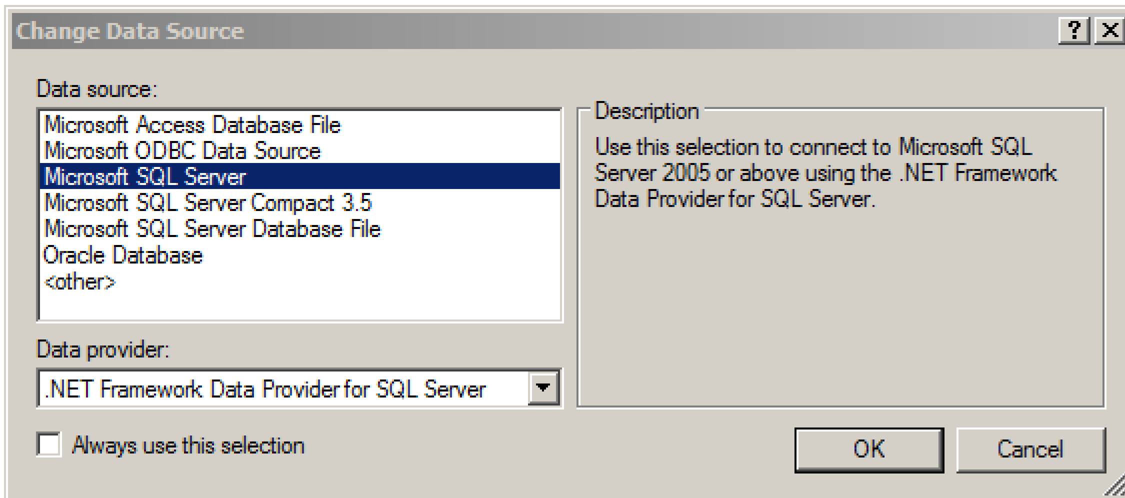

Abbildung D.3.: Hauptfenster von *PDB2WC*

crosoft SQL Server angezeigt.

Die Datenquelle wird in einem separaten Dialog ausgewählt, der durch einen Klick auf den `Change...`-Button oben rechts geöffnet wird. Abbildung D.3 stellt den Dialog zur Auswahl der Datenquelle dar. Der Verbindungsdialog aus Abbildung D.2 passt seinen Inhalt an die spezifischen Erfordernisse der Datenquelle an.

Weil auch Datenquellen für *Oracle* und *ODBC* (über diese Schnittstelle wird *Intersystems Caché* angesprochen) vorhanden sind, lässt sich das Tool bei Bedarf schneller an die anderen beiden Datenbanksysteme anpassen, weil die Verbindungsfunktionalität nicht implementiert werden muss. So muss nur noch der Zugriff auf die Metadaten und die Generierung der spezifischen Skript-Syntax implementiert werden.