Fachhochschule Wiesbaden
Fachbereich Design Informatik Medien
Studiengang Allgemeine Informatik

Diplomarbeit

zur Erlangung des akademischen Grades
Diplom-Informatiker(FH)

Implementierung einer universellen, GUI-unterstützten Import/Export Schnittstelle für Stamm- und Bewegungsdaten in CAQ-Systemen mittels .NET und XML

vorgelegt von Oliver Lind
am 18.05.2007
Referent: Herr Prof. Dr. H. Werntges
Korreferent: Herr Dipl.-Phys. U. Bursik

Erklärung gem. Prüfungsordnung –Teil A- § 6.4.2

Ich versichere, dass ich die Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Ort, Datum Unterschrift Diplomand

Hiermit erkläre ich mein Einverständnis mit den im folgenden aufgeführten Verbreitungsformen dieser Diplomarbeit:

Verbreitungsform	**ja**	**nein**
Einstellung der Arbeit in die Bibliothek der FHW	X	
Veröffentlichung des Titels der Arbeit im Internet	X	
Veröffentlichung der Arbeit im Internet	X	

Ort, Datum Unterschrift Diplomand

Inhalt

Anhang

1 Einleitung

Schnittstellenarchitektur ist nicht zuletzt überall dort ein Thema, wo Daten im Sinne eines Electronic Data Interchange (EDI) zwischen verschiedenen Systemen ausgetauscht werden oder ausgetauscht werden sollen[1]. Inkompatible Datenformate, unsauber dokumentierte Schnittstellen und für den Betrieb notwendige Legacy-Formate, die nicht mehr dem aktuellen Stand der Technik entsprechen, sorgen in diesem Bereich häufig für Probleme. In der Theorie sollte diese Problematik frühzeitig durch ein einheitliches, erweiterbares Konzept mit kompatiblen Modellen vermieden werden. In der Praxis ist es jedoch häufig so, dass die in einer Firma verwendeten Formate und Schnittstellen modulare Abhängigkeiten besitzen, die sich über Jahre entwickelt haben. Neu hinzu kommende Anforderung haben neue Datenformate nötig gemacht. Einzelfalllösungen für bestimmte Kunden, spezifische Strukturen für ein bestimmtes Problem und persönliche Vorlieben der verschiedenen beteiligten Programmierer ergeben schnell eine sehr inhomogene Datenlandschaft, die mit fortschreitender Dauer und steigender Komplexität einen immer höheren Aufwand an Zeit und Geld für Wartung, Anpassungen und Pflege bewirken. Diese Entwicklung führt zwangsläufig zu einem Punkt, wo es aus Kosten-Nutzen-Sicht attraktiv erscheint, die bestehenden Datenstrukturen zumindest auf der Transportebene zu vereinheitlichen und die Schnittstellenlandschaft zu konsolidieren. Dabei sind verschiedene wichtige Punkte zu berücksichtigen. Die neue Schnittstelle muss flexibel und erweiterbar sein. Ist sie dies nicht, hat man zwar die bestehende Situation verbessert, ist allerdings für zukünftige Entwicklungen nicht gut aufgestellt. Außerdem muss sie robust sein und das bestehende Datenmaterial zuverlässig abbilden können. Fehlererkennung, Validierbarkeit und Tauglichkeit zur Automatisierung können ebenfalls wichtige Punkte darstellen. Diese Punkte machen das Entwickeln einer universellen Schnittstelle für mehrere Systeme zu einem nicht trivialen Problem. Ziel der Arbeit soll es sein, eine solche Schnittstelle zu entwerfen.

1.1 Ausgangslage

Zunächst zu betrachten sind die Rahmenbedingungen, die die Parameter der Arbeit vorgeben.

[1] Der Begriff „Schnittstelle“ ist sehr weit gefasst und es existieren diesbezüglich eine Vielzahl Definitionen. Gemeint ist der Teil eines Systems, der dem Austausch von Informationen im engeren Sinne dient.

1.1.1 Firma

Die IBS SINIC GmbH[2] (SINIC) ist ein Anbieter für Produkte im Bereich Computer Aided Quality (CAQ). SINIC entwickelt und vertreibt seit 15 Jahren Softwarelösungen für den Bereich Qualitätsmanagement. Die beiden wichtigsten Produkte sind CALVIN und das SINIC-CAQ System, das seinerseits aus etwa 17 verschiedenen Einzelmodulen besteht, die auch separat erworben werden können. Die meisten dieser Module verwenden, sofern erforderlich, ein eigenes ASCII-Datenformat zur Übermittlung von Daten. Ältere Module sind zum Großteil in Visual Basic umgesetzt, während jüngere Module die .NET-Plattform von Microsoft nutzen. SINIC ist Microsoft Certified Gold -Partner, daher finden alle Neuentwicklungen unter Zuhilfenahme von Microsoft-Produkten auf der .NET-Plattform statt.

Auf Seiten der Kunden, die primär im mittelständischen Umfeld angesiedelt sind, existieren eine große Zahl Softwaresysteme, die Daten in unterschiedlichsten Formaten bereithalten. Textdateien in verschiedenen Ausgestaltungen und Excel-Tabellen sind gängig, aber auch XML, Access oder andere, exotischere Formate sind möglich. Diese Konstellation macht es momentan nötig, für jedes neue Kundenformat einen eigenen Konverter zu entwickeln, der die Schnittstelle zwischen Kundensystem und SINIC-Modul bereitstellt. Endgültig findet die Speicherung der relevanten Daten in Datenbanken statt. Die Art der Datenbanken kann je nach Präferenz des Kunden variieren und ist anpassbar. Das Spektrum reicht von Access über SQL-Systeme bis Oracle.

1.1.2 Ziel der Arbeit

Die Arbeit wurde von der IBS SINIC in Auftrag gegeben. Als definiertes Ziel galt es, eine universelle Schnittstelle zu entwickeln, die die wichtigsten SINIC-Formate abdeckt und eine einheitliche programmatische Bearbeitung der eingehenden Kundendaten ermöglicht. Da die Datenformate der Kunden veränderlich sind und bei neu hinzukommenden Kunden keine Aussage über die bestehenden Datenformate gemacht werden kann, war eine graphische Oberfläche (GUI) zu entwickeln, die es dem Kunden erlaubt, sein vorliegendes

[2] http://www.sinic.de

Format auf das gewünschte SINIC Format zu mappen. Als Resultat dieses Mappings sollte der Datenimport vom Kunden zu SINIC in Zukunft ohne weiteren Userinput sowohl manuell auslösbar sein, als auch automatisiert ablaufen können. Hierzu musste ein neues Datenformat definiert werden, das die Forderung nach Flexibilität und Erweiterbarkeit erfüllt. Das Softwareprojekt hat den Namen „SINIC Universal Interface Tool“ (SUI-Tool).

Naturgemäß stehen wirtschaftliche Erwägungen als Hauptmotivation hinter der Arbeit. Die momentane Situation, die eine manuelle Anpassung des Programmcodes bei jeder auftretenden Änderung seitens bestehender Kunden bzw. dem Hinzukommen neuer Kunden erfordert, beansprucht einen substanziellen Aufwand an Ressourcen[3]. Die Codekomplexität steigt durch das Handling immer neuer Sonderfälle an, was zum einen die Fehleranfälligkeit erhöht, zum anderen die Übersichtlichkeit und Wartbarkeit reduziert. Auch sind die innerhalb der CAQ-Systeme anfallenden Daten bisher nicht ohne weiteres koppelbar und müssen durch eine Vielzahl kleiner Tools aufeinander angepasst werden. Dies erfordert wiederum mehr Aufwand, als eine Datenstruktur, die die Systeme aneinander koppelbar macht. Erfolgreiche Umsetzung der Arbeit könnte hier zu Einsparungen führen und so einen positiven wirtschaftlichen Effekt für SINIC erzielen. Desweiteren muss jede Änderung vom Kunden kommuniziert werden und die Umsetzung technisch und terminlich abgestimmt werden. Dieser Prozess ist naturgemäß den bekannten Störfaktoren unterworfen, die bei der Kommunikation zwischen mehreren Partner auftreten können. Missverständnisse, Probleme bei Terminabstimmungen, wechselnde personelle Zuständigkeiten zwischen den betroffenen Parteien und andere. Die zu entwickelnde Software verlegt den Anpassungsprozess von SINIC weg zum Kunden. Dieser kann seine eigenen Formate selbst pflegen, was einerseits einen Mehrwert für den Kunden darstellt, zum anderen die SINIC-Ressourcen schont.

1.1.3 Technische Gegebenheiten

Als Entwicklungssysteme werden PCs eingesetzt, die Microsoft Windows XP Professional als Betriebssystem nutzen. Entwicklungsumgebung ist Visual Studio 2005 Team Developer’s Edition mit Microsoft .NET 2.0. In Visual Studio eingebunden sind eine Reihe Komponenten von Drittanbietern, die zusätzliche Funktionen und Möglichkeiten

[3] Gemeint ist an dieser Stelle hauptsächlich das Personal als kritischer Faktor.

bereitstellen. Hier unter anderem Crystal Reports von Business Objects (Reporting), combit List&Label (Printservices) und zusätzliche Windows Controls von Infragistics, Component One und PlexityHide.

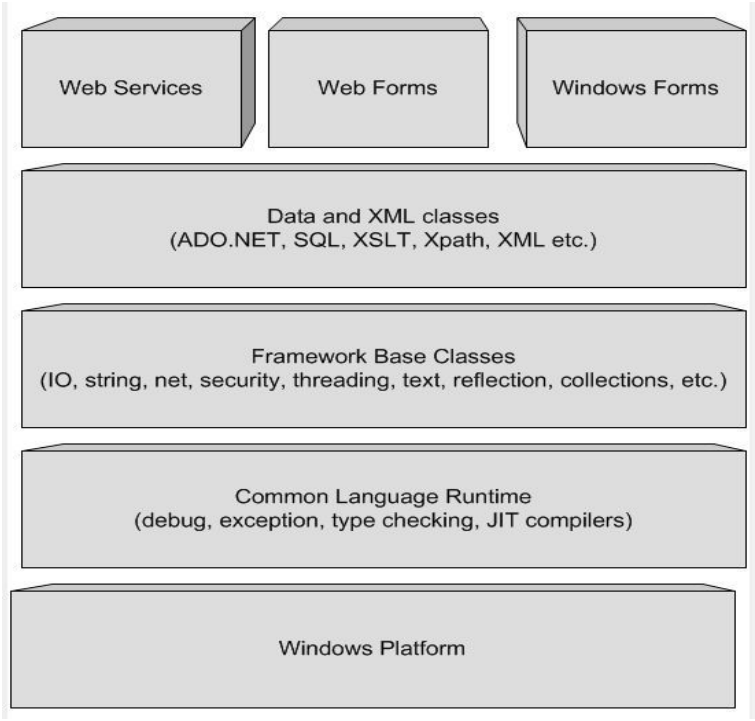
Jüngere Module stützen sich auf das SINIC-Framework für .NET. Dies ist ein umfangreiches, von SINIC entwickeltes Framework, das als vereinheitlichende Grundlage für alle Neuentwicklungen verpflichtend zu verwenden ist. Es verwendet die o.g. externen Komponenten. Als Entwicklungssprache wird C# verwendet. Aus diesen Gegebenheiten heraus resultiert die Vorgabe, die zu erstellende Software unter Verwendung von .NET und dem SINIC-Framework mittels C# zu programmieren. Eine entsprechende Einarbeitung ist Teil der Aufgabe.

1.2 Microsoft .NET[4]

„NET Framework ist ein Produkt, das die Entwicklungsgrundlage für die Microsoft .NET-Plattform bildet. .NET Framework und das geräteorientierte .NET Compact Framework stellen eine verwaltete, sichere Ausführungsumgebung für XML-Webdienste und Anwendungen sowie umfassende Unterstützung für XML zur Verfügung. Die Schlüsseltechnologien in .NET Framework sind Common Language Runtime, die Klassenbibliotheken und ASP .NET.“[5]

Es würde den Rahmen der Arbeit sprengen, detailliert über das .NET Framework (.NET) zu sprechen, von daher sollen kurz einige definierende Eckpunkte genannt werden, um einen Überblick zu vermitteln. Folgendes Schaubild verdeutlicht den generellen Aufbau der Frameworks:

[4] Alle Angaben beziehen sich auf die Version 2.0 des .NET Frameworks.
[5] MSDN, Microsoft .NET Framework



.NET Framework Architecture; Programming C#, S.5

Kernelement von .NET ist die Common Language Runtime (CLR). Diese Laufzeitumgebung stellt eine einheitliche Integrationsumgebung für die darüber liegenden Schichten dar. Die CLR enthält eine Virtual Machine, die der von Java bekannten Virtual Machine ähnelt und Just-in-time (JIT) Compiler.

Vom Ablauf her produzieren die Compiler des Frameworks zunächst aus dem Programmcode eine prozessorunabhängige Zwischensprache, die Microsoft Intermediate Language (MSIL). Die MSIL wird von der CLR ausgewertet und vom JIT-Compiler in plattformspezifischen Code übersetzt. Das bedeutet, dass theoretisch jeder Compiler die CLR nutzen kann, sofern er validen MSIL-Code erzeugt. Für Entwickler bietet das Vorteile. Es ist durch dieses System möglich, eine Software problemlos in verschiedenen Sprachen zu programmieren. Egal ob die Programmierung in C#, J# oder VB geschieht: schlussendlich wird nahezu identischer MSIL-Code erzeugt. Es ist auch möglich, Objekte, die in Sprache A definiert werden, von Objekten in Sprache B abzuleiten. Dem Objekt ist es gleich, in welcher Sprache die vererbende Klasse definiert wurde.

Damit eine Sprache als .NET-Sprache verwendet werden kann, muss sie die Common Language Specification (CLS) erfüllen. Diese ist in .NET beinhaltet und legt die grundlegenden Regeln zur Sprachintegration fest.

Eine weitere wichtige Komponente der CLR ist das Common Type System (CTS). Das Typsystem ist von den Sprachen losgelöst. Die Framework-Compiler mappen automatisch die verwendeten Typen in die korrekten CTS-Typen um. Da das CLR die erzeugte MSIL auswertet, können keine Inkonsistenzen zwischen den Typen der verschiedenen Sprachen auftreten.

Auf der CLR setzen die Framework Base Classes auf, die grundlegende Funktionalitäten zur Verfügung stellen (IO, String Handling, Netzwerkfunktionen etc.). Darüber sind die Klassen für Daten und XML, die sich um Persistenz kümmern. Hier finden sich hauptsächlich Klassen, die mit Manipulation und Interaktion bezüglich Daten und Backend zu tun haben. SQL und XML sind zwei Beispiele, die für die Arbeit erhöhte Bedeutung haben werden. Die oberste Schicht bilden die Klassen zur Darstellung von Web- und Windowsoberflächen, sowie Web Services. Durch die enge Integration mit Microsoft Visual Studio ist mit diesen Klassen eine schnelle Entwicklung von Anwendungen möglich, da Visual Studio eine graphische Möglichkeit zur Oberflächengestaltung vorsieht und Komfortfunktionen wie das Drag&Drop von Controls und interaktive Möglichkeiten zur Eventgenerierung und der Manipulation von Objekteigenschaften bietet.
Alle diese Klassen bilden zusammen die Framewor*k Class Library* (FCL), die über 4.000 Klassen enthält.

Ein für die Arbeit positiver Effekt ist die starke Unterstützung von XML. Es steht eine umfangreiche Sammlung an Klassen in der FCL zur Verfügung, die sich mit dem Lesen, -Verarbeiten und Schreiben von XML-Daten beschäftigen.

1.3 SINIC-Framework

1.3.1 Aufbau

SINIC verwendet zur Programmierung ein eigenes Framework. Darunter ist eine Klassenbibliothek zu verstehen, die helfen soll, wiederkehrende Aufgaben zu vereinfachen und zu vereinheitlichen.
Werden neue Module geschrieben oder bestehende erweitert, wird, wenn möglich, auf die entsprechenden Funktionalitäten zurückgegriffen.

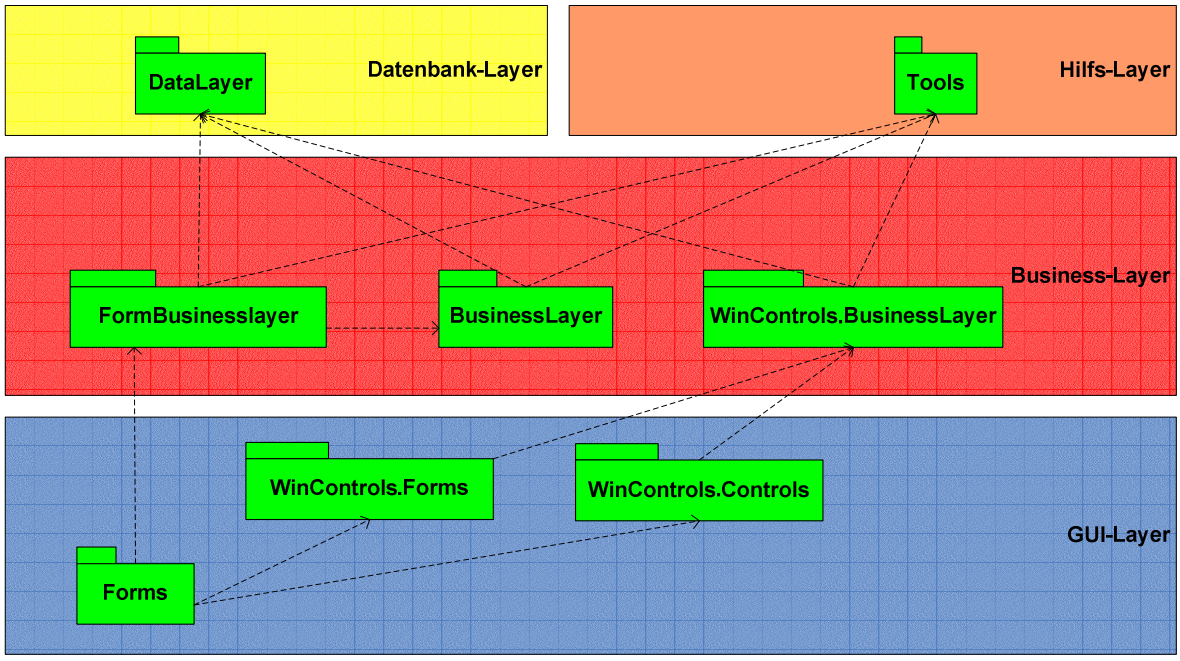
Aufgeteilt sind die Klassen in folgende neun Namespaces:

- Datalayer
- Forms
- FormBusinesslayer
- BusinessLayer
- Tools
- Tools.CTV
- WinControls.Forms
- WinControls.Controls
- WinControls.BusinessLayer

Diese Namespaces sind wiederum vier logischen Schichten zuzuordnen:

- Datenbank-Layer
- Hilfs-Layer
- Business-Layer
- GUI-Layer

Die Abhängigkeiten können dem folgenden Diagramm entnommen werden:



SINIC Framework Struktur Quelle: SINIC Infothek

1.3.2 Die Namespaces

Im Folgenden sollen die Namespaces kurz mit ihrer Funktion beschrieben werden.

1.3.2.1 Datalayer

Enthält Klassen, die sowohl den Zugang als auch die Interaktion zwischen Nutzer und Datenbank vereinfachen. Hierzu gehören die Anbindungen an verschiedene Datenbanken und Methoden zur Abfrage bestimmter Werte und Hilfsmethoden, die speziell auf die Bedürfnisse der SINIC-spezifischen Controls zugeschnitten sind.

Klassen: *SinicDBManager, SinicDBTools, SinicOleDBConnection, SinicOleDBShell*

1.3.3 Forms

Enthält die GUI-Oberflächen bzw. Controls. Die beiden Haupttypen im Framework sind Listen und Detailmasken. Die Detailmasken zeigen in der Regel die zu den jeweiligen Listenzeilen zugehörigen Datensätze an und bieten Funktionalitäten zum Ergänzen und Manipulieren der Gesamtliste. Diese Forms müssen entsprechend der gewünschten Anwendung programmiert werden und stellen den Bereich dar, in dem das Anwendungsfrontend hauptsächlich programmiert wird. Listen und Masken haben jeweils ein Pendant im FormBusinesslayer, wo die für die Anwendung erheblichen Routinen und Datenbankoperationen getätigt werden. (siehe unten)
Um korrekt zu funktionieren muss die richtige Anbindung/Integration zum Rest des Frameworks gewährleistet sein. Detail- und Listenansichten müssen von den unten angegebenen Klassen abgeleitet werden.

Klassen abgeleitet von*: frmList, frmHeadBase*

1.3.4 FormBusinesslayer

Die hier enthaltenen Klassen stellen die Anwendungslogik zu den Forms im Frontend dar. Die Klassen müssen von den unten angegebenen Klassen abgeleitet werden. Eine in diesen Klassen zwingend zu erledigende Aufgabe ist das Überschreiben der doDelete – Funktion. Dies ist nötig. um ein Löschen von Listenzeilen zu ermöglichen. Außerdem muss der Primärschlüssel für die mit der Liste verknüpfte Datenbanktabelle in der Funktion `SetPrimaryKeys` gesetzt werden.

Klassen abgeleitet von: *blListBase, blHeadBase*

1.3.4.1 BusinessLayer

Enthält allgemeine Klassen zur Geschäftslogik, die spezifisch für das jeweilige Modul sind. Konstanten, Enumerationen und andere Informationen, die modulweit gültig sind, sind hier zu finden.

1.3.4.2 Tools

Eine generelle Sammlung nützlicher Funktionen, Fehlermeldungen, Konstanten usw. Im Gegensatz zu den im BusinessLayer zu findenden Klassen und Aufrufen sind die Komponenten in diesem Namespace modulübergreifend gültig.

1.3.4.3 Tools.CTV

Tools.CTV(für Constants, Types, Variables) ist eine Ansammlung modulspezifischer Konstanten. Einige Anpassungen sind hier nötig, um eine neue Anwendung in das Framework integrieren zu können. Wichtig sind das Anlegen der für die Darstellung der Listen wichtigen Layout-IDs sowie die Festlegung eines Modulcodes, der dann für ein Matching gegen diverse andere Funktionen wie z.B. Datenbankzugriffe herangezogen wird. Prinzipiell wird in Tools.CTV die Identität der neuen Anwendung im Framework verankert und bekannt gemacht.

1.3.4.4 WinControls.Forms

Grundlegende Klassen für GUI-Elemente. Die allgemeine Natur macht eine modulübergreifende Verwendung möglich. Neben Klassen für Infoboxen, Blankofenster, Fortschrittsbalken, Container und andere generische Elemente findet sich hier auch die wichtige *frmHeadBase,* von der alle Kopfmasken abgeleitet werden müssen.

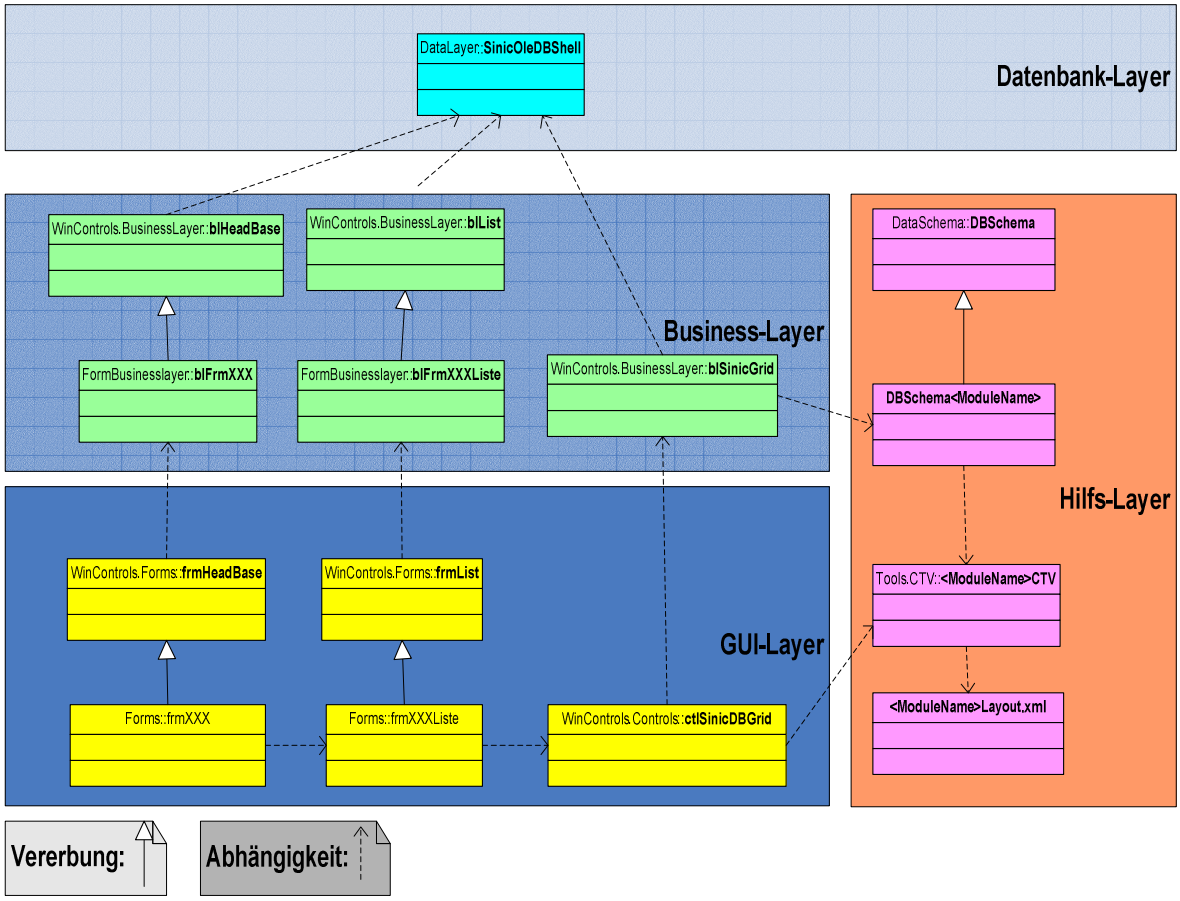
1.3.4.5 WinControls.Controls

Enthält von Standard-Forms abgeleitete SINIC-spezifische Controls. Das Spektrum reicht von Buttons über Checkboxen bis zur DBGrid-Control. Alle Controlklassen beginnen zur leichten Identifikation mit ctlSinic im Klassennamen. Besondere Relevanz hat das SinicDBGrid2, welches das Kernelement für Listen darstellt. Generell ist bei der Erstellung von Forms den spezialisierten SINIC-Controls Vorrang gegenüber den .NET–Standardcontrols zu geben.

1.3.4.6 WinControls.BusinessLayer

Enthält die für die in WinControls.Controls und .Forms existierenden Controls die entsprechende Kontrolllogik. Zum Teil sind diese Klassen abstrakt gehalten und müssen bei Verwendung in einem Modul entsprechend implementiert werden. Die unter BusinessLayer angesprochenen Basisklassen *blListBase* und *blHeadBase* sind hier zu finden.

Im folgenden Schaubild werden die Abhängigkeiten der Klassen für Listen und Masken verdeutlicht:



SINIC Layer Struktur Quelle: SINIC Infothek

Im Vergleich zur klassischen Model-View-Controller-Architektur ersetzt der BusinessLayer die Control-Schicht. Da es kein Modell im engeren Sinne abzubilden gibt, tritt an dessen Stelle der Datenbank-Layer, der auf unterster Ebene die Transaktionen zwischen Business-Layer und Datenbanken durchführt. Die Elemente der Hilfsschicht sind aus den anderen Layern ausgelagert, da sie modulübergreifende Funktionen zur Verfügung stellen. Die Auslagerung in eine eigene Schicht verhindert Cluttering und Redundanzen in den individuellen Modulen.

1.3.5 Datenzugriff

Zugriffe auf die Datenbanken erfolgen ausschliesslich über die im Framework verankerten Prozesse und Routinen. Eine zentrale Funktion übernimmt dabei die Klasse *SinicDBShell*.

Beispiel: Anbindung einer Datenbank an ein Datagrid.

In den Anzeigemasken werden häufig Daten in einer Tabelle angezeigt. Das folgende Schaubild zeigt den Prozess, wenn eine Windows-Form Daten aus einer Datenbank anfordert.

Der ursprüngliche Aufruf erfolgt aus der Form an die Klasse *SinicDBGrid2*.

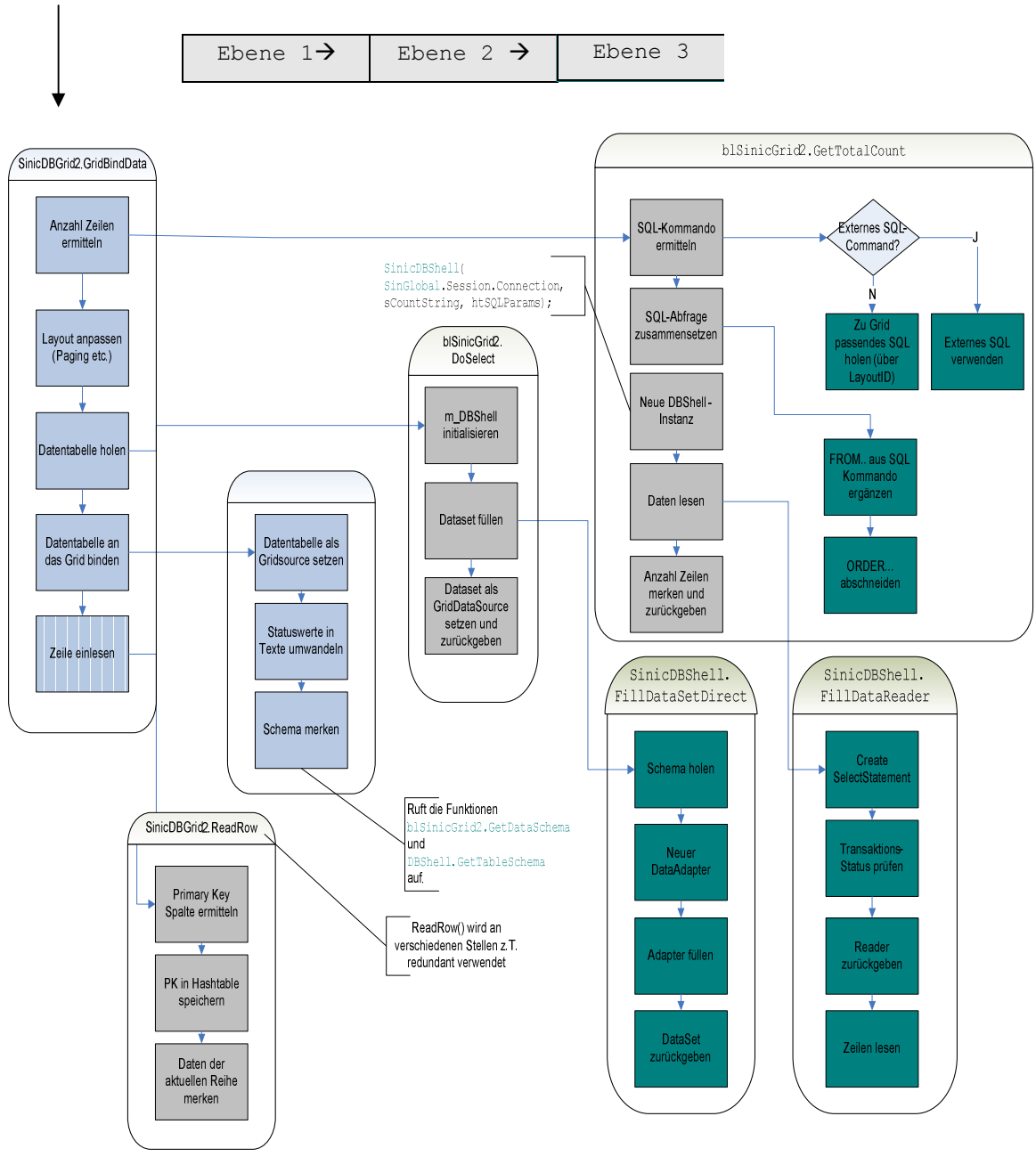
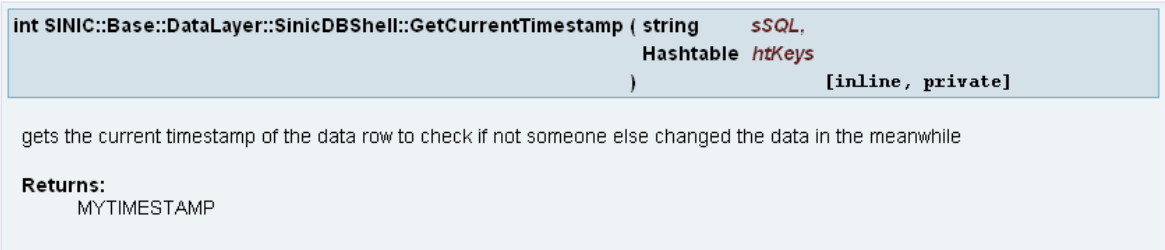


Schaubild: Relation der Datenbank zu einem DataGrid

1.3.6 Transaktionsschutz

Da in den SINIC-Modulen häufig Datenbanktransaktionen durchgeführt werden, ist es wichtig, dass die Daten konsistent valide sind und keine Fehler auftreten oder Verluste entstehen. Dies gilt gerade auch für den Multi-User-Betrieb. SINIC implementiert eine Variante des optimistischen Lockings. Der zu ändernde Datensatz wird inklusive eines als Integer abgelegten Timestamps gelesen.



Dieses Feld repräsentiert keine echte Zeit, sondern eine rein zur Zustandskontrolle dienende, fortlaufende Zahl.

Nach der Manipulation des Datensatzes wird kontrolliert, ob der Originaldatensatz in der DB noch den gleichen Timestamp wie zu Beginn der Transaktion aufweist. Ist dies der Fall, so werden die neuen Daten inklusive des inkrementierten Timestamps in die DB geschrieben. Stimmt der Timestamp nicht überein, so wird der Datensatz verworfen und die Transaktion neu gestartet. Dieses Verfahren erfüllt im Hinblick auf die Transaktionshäufigkeit und die Anzahl simultan arbeitender User zuverlässig seinen Zweck

1.3.7 Limitationen

Naturgemäß hat ein anwendungsorientiertes und auf eine bestimmte Thematik ausgerichtetes Framework auch einige Limitationen, die nun kurz dargestellt werden sollen. Momentan ist das Framework nur bedingt Webtauglich. Die Integration von ASP.NET ist noch nicht durchgeführt, allerdings bereits vorbereitet. Da die Klassen des Frameworks bewusst sehr allgemein und zentralisiert strukturiert sind, geht zum Teil die Übersichtlichkeit ein wenig verloren. Zum Beispiel werden Textbausteine zur Anzeige in Dropdown-Boxen zentral in wenigen Klassen gehalten. Dieser Zusammenhang erschließt sich nicht unmittelbar und bedarf einer gewissen Einarbeitungszeit. Ein weiterer Punkt, der

sich auch auf die Arbeit ausgewirkt hat, ist die Orientierung auf bestimmte SINIC-Controls. Da diese Controls im Framework fest verankert sind und häufig genutzt werden, richtet sich die GUI-Gestaltung zwangsweise nach diesen aus. Abweichende Oberflächen würden einen entsprechenden Mehraufwand erfordern, der nur schwer durch rein optische Gründe zu rechtfertigen ist. Ein weiterer Punkt ist die bisher kaum vorhandene Ausnutzung der speziellen .NET-Sicherheitskonzepte. .NET stellt eine Vielzahl an Mechanismen zur Verfügung (Rollenmodelle, Codezugriffssicherheit u.a.), die mangels zwingenden Bedarfs bisher noch nicht über das Framework abgebildet werden können.

Abschließend soll noch die Übersetzungsfähigkeit erwähnt werden. Das Framework bietet Konstrukte zur automatischen Übersetzung von Anwendungen in andere Sprachen an. Diese Übersetzungsfunktionalität ist auf die Oberflächen beschränkt und kann keine Daten übersetzen.

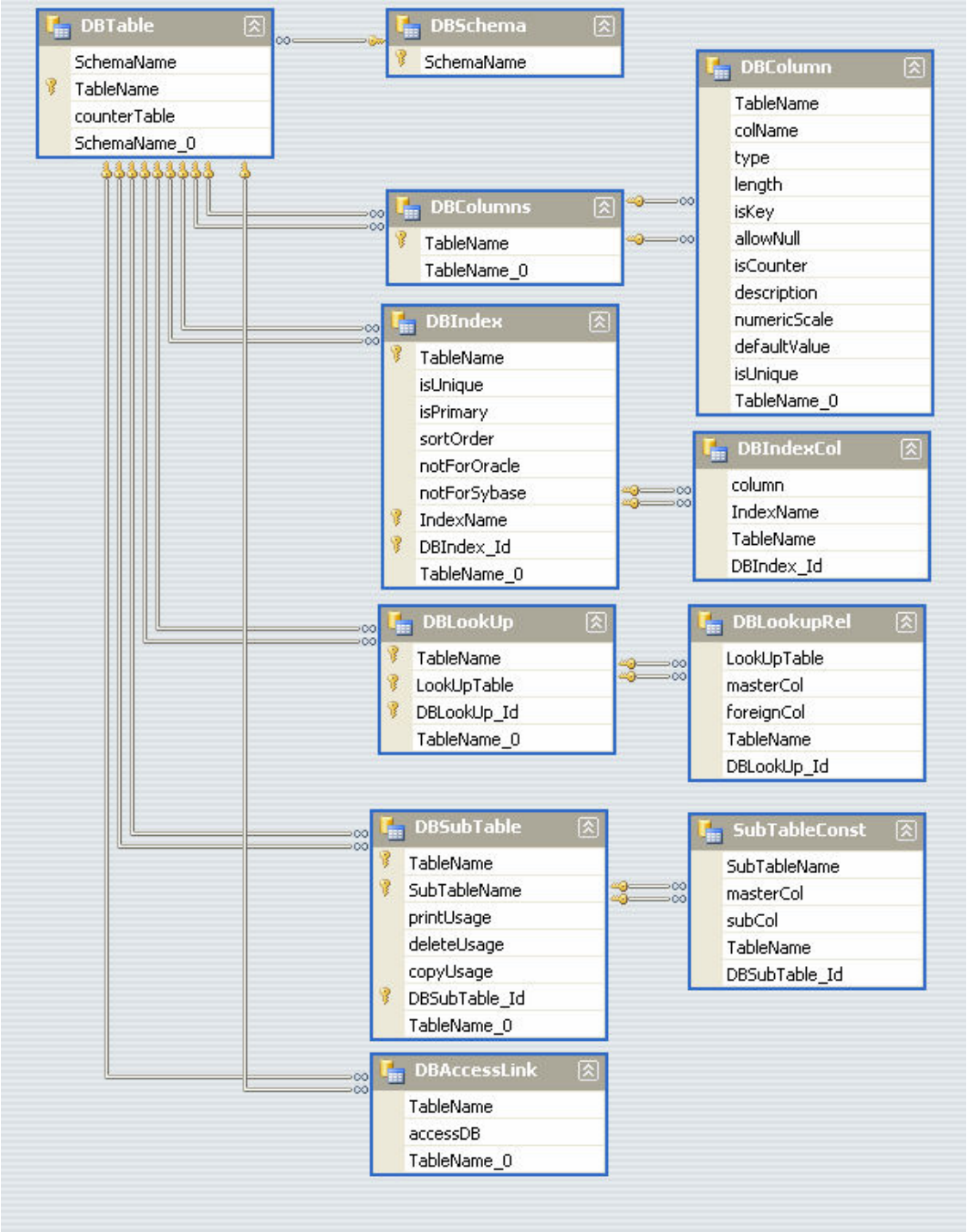
1.4 XML Datenbank-Schemas

Eine zentrale Komponente innerhalb des SINIC-Frameworks sind die XML-Dateien, die die Datenbankschemas enthalten (XmlDBSchemas). In ihnen ist jede Datenbank als XML-Beschreibung hinterlegt. Hier ein Auszug aus der Datei QSISTM.xml, die den Aufbau der darin enthaltenen Tabelle QSART verdeutlicht:

```
<DBTable d2p1:SchemaName="QSISTM" d2p1:TableName="QSART"
d2p1:counterTable="">
<DBColumns d2p1:TableName="QSART">
<DBColumn   d2p1:TableName="QSART" d2p1:colName="ARTBAUGRP"
                  d2p1:type="WChar" d2p1:length="35"
                  d2p1:isKey="false" d2p1:allowNull="true"
                  d2p1:isCounter="false" d2p1:description="Baugruppe"
                  d2p1:numericScale="0" d2p1:defaultValue=""
                  d2p1:isUnique="false"/>
<DBColumn   d2p1:TableName="QSART" d2p1:colName="ARTNR"
                  d2p1:type="WChar" d2p1:length="35" d2p1:isKey="true"
                  d2p1:allowNull="true" d2p1:isCounter="false"
                  d2p1:description="Artikelnummer" d2p1:numericScale="0"
                  d2p1:defaultValue="" d2p1:isUnique="true"/>
                        […  weitere Tabellenspalten…]
</DBColumns>
<DBIndex          d2p1:TableName="QSART" d2p1:isUnique="true"
                  d2p1:isPrimary="true" d2p1:sortOrder="asc"
                  d2p1:notForOracle="false" d2p1:notForSybase="false"
                  d2p1:IndexName="ART01">
<DBIndexCol d2p1:column="ARTNR" d2p1:IndexName="ART01"
                  d2p1:TableName="QSART"/>
</DBIndex>
</DBTable>
```

(siehe Anhang für das komplette Schema)

Der Präfix “d2p1” wurde beim Erstellen der Schemata willkürlich gewählt und hat keine inhaltliche Bedeutung. Die DB-Schemas als Ganzes wurden während der Entwicklung des SINIC- Frameworks durch ein Tool automatisiert erstellt. Wie sich während meiner Arbeit zeigte, sind manche noch nicht ausgiebig getestet. Dies liegt daran, dass ältere Module, die vor der Umstellung auf die .NET-Plattform entwickelt wurden, noch nicht auf die Schemas zugreifen und daher diesbezüglich wenig Bedarf bestand. Zunächst soll die Struktur des XML–Dokuments genauer untersucht werden. Der XML-Datei liegt ein XML-Schema zu Grunde, dass sich in der Visual Studio -Visualisierung folgendermaßen darstellt (siehe Abbildung nächste Seite):



Übersicht: XmlDbSchema

(Anmerkung: Attribute mit _0 am Ende werden aus technischen Gründen automatisch erstellt, werden aber für das dahinter liegende XML-Schema ignoriert.)

Im Folgenden werde ich auf die Teilbereiche der Struktur eingehen, die für die Arbeit größere Bedeutung hatten.

Zentraler Kern ist der Aufbau der Tabellen als eine Sammlung von *DBColumn* –Elementen. die in dem *DBColumns* -Element gesammelt sind. In ihnen sind alle für die Datenbank relevanten Informationen bezüglich einer Tabellenspalte erfasst. Besondere Bedeutung für den weiteren Verlauf haben die Attribute *TableName*, das den Namen der Datenbanktabelle angibt, *colName*, der Spaltenname, *isKey*, das kennzeichnet, ob es sich bei der Spalte um einen Primary-Key handelt oder nicht, und *description*, die Beschreibung der Tabellenspalte.

Description war zu Beginn der Arbeit leer und zugleich das einzige Feld, was bedenkenlos editiert werden konnte. Die im Auszug abgebildeten Inhalte stellen nachträgliche Änderungen meinerseits dar, die GUI-Informationen enthalten. Für meine Arbeit war es erforderlich, dass alle relevanten Informationen zur Interaktion mit SINIC-Datenbanken direkt diesen Schemas entnommen werden.

1.5 C#

Eine Vorgabe ist die Programmierung der Anwendung mit der Programmiersprache C#. Die Sprache ist in Standard ECMA-334 definiert.[6] C# ist eine stark typisierte Sprache, die Eigenschaften von Java und C++ kombiniert und eine objektorientierte, strukturierte Programmiermethodik unterstützt. In Version 2.0 wurde sie um einige fehlende Elemente ergänzt (z.B. *generics*, die in etwa den aus C++ bekannten *Templates* entsprechen). Da C# nur Hilfsmittel und nicht Kerngegenstand der Arbeit ist, beschränke ich mich auf die Angabe einiger Eigenschaften, die Auswirkungen auf Designentscheidungen haben oder zum Verständnis des beigefügten Codes hilfreich sind.

Klassen und Code – in C# wird alles, was zu einer Klassendeklaration gehört, in der eigentlichen Deklaration erfasst. Es gibt keine separaten Headerfiles, wie man sie aus C++ kennt.

[6] http://www.ecma-international.org/publications/standards/Ecma-334.htm

Eigenschaften/Properties – erlauben es dem Nutzer, auf den Zustand einer Klasse zuzugreifen, als würden Membervariablen direkt angesprochen, obwohl in Wahrheit Klassenmethoden verwendet werden[7]. Sie haben eine *Get*-Komponente zum Lesen und eine *Set*-Komponente zum Schreiben des gewünschten Members. Auf diese Art können die in Klassen enthaltenen Daten eingekapselt werden.

Keine Mehrfachvererbung – Ebenso wie in Java kann eine Klasse nicht von mehreren anderen Klassen erben.

Interfaces – Ein Interface ist ein bindender Kontrakt. Es kann u.a. die Signaturen von Methoden und Eigenschaften enthalten. Eine Klasse oder Struct, die diese Klasse implementieren will, muss sich an diesen Kontrakt halten.[8]

Beispiel

```
interface IFlugfaehig
{
    void Starten(float speed);
}
```

Das Interface IFlugfaehig schreibt eine Methode vor. Um dieses Interface implementieren zu können, muss die fragliche Klasse die geforderte Methode umsetzen. Nehmen wir eine Klasse JumboJet an, die von der abstrakten Klasse Fahrzeug erbt und IFlugfaehig implementiert.

Beispiel

```
public class JumboJet: Fahrzeug,IFlugfaehig
{
//IFlugfaehig implementieren
    public void Starten(float speed)
    {
        // hier wird gestartet
    }
//... weiter Eigenschaften, Methoden, etc.
}
```

[7] Programming C#, Seite 97
[8] MSDN, Interface(C#)

Andere Objekte können in ihrem Code mit Interfaces agieren, ohne die genaue Umsetzung innerhalb des exakten Objekts zu kennen. Für sie ist nur von Belang, dass das Interface umgesetzt wird. Nehmen wir weiter an, es gibt eine andere Klasse, die ein Array mit Fahrzeug-Objekten vorhält. Alle flugfähigen Objekte sollen gestartet werden:

Beispiel

```
foreach (Fahrzeug f in fahrzeugArray)
        {
            if (f is IFlugfaehig)
            {
                IFlugfaehig flieger = (IFlugfaehig) f;
                  f.Starten(float s);
            }
        }
```

Der Unterschied zwischen abstrakten Klassen und Interfaces wird sprachlich gut deutlich. JumboJet *ist-ein* Fahrzeug und *implementiert-ein* IFlugfaehig Interface. Eine Besonderheit von C# ist, daß eine Klasse mehrere Interfaces implementieren kann.
Dies kann für die Struktur der zu programmierenden Software Auswirkungen haben, da Interfaces in dieser Beziehung flexibler sind.

Keine Pointer – Grundsätzlich werden Pointer im C-Stil in C# unter dem Paradigma des „managed Code“ nicht verwendet. Die Möglichkeit ist zwar gegeben, allerdings müssen die entsprechenden Passagen als „unsafe“ gekennzeichnet werden. In meiner Arbeit habe ich diesen Weg jedoch bewusst nicht eingeschlagen.

Garbage collection – C# berücksichtigt, dass die CLR eine garbage collection besitzt. Diese ruft von sich aus die nötigen Destruktoren auf. Es finden sich daher keine „echten“ Destruktoren im Code. Es ist jedoch möglich, Objekte mit einem finalize-Block zu versehen, der destruktorähnliche Aufgaben erledigt. Der explizite Aufruf eines Destruktors ist nicht möglich.[9]

C# eröffnet vollen Zugriff auf die Möglichkeiten des umfangreichen .NET- Frameworks. Es ist davon auszugehen, dass diese Kombination die Entwicklung nach entsprechender Einarbeitung positiv begünstigt.

[9] Programming C#, Seite 85

1.6 Schnittstellen

Bevor mit der Entwicklung eines Konzepts begonnen werden kann, müssen zunächst die Formate betrachtet werden, die für die Entwicklung eine Rolle spielen. Hierzu werden einige typische Dateien herangezogen, die beim Kunden für Exportzwecke anfallen. Betrachtet werden sollen an dieser Stelle nur die technischen Details

1.6.1 Artikel im CAQ-System

Typ: Textdatei

Format: Key-Value Paare

Die erste Zeile enthält eine Angabe zur Versionsnummer, dann folgen 1-n Datensätze mit Artikeln. Jeder Artikel kann 0-n Lieferantenbeziehung enthalten. Mehr als drei Lieferantenbeziehungen sind allerdings selten.

Dateiauszug:

```
VER:AAI-200
A01:1.110.4315
A02:
A03:Ti Gr.4 ASTM F 68
A04:10
***
F02:7071801
F03:
F04:
```

Es gibt 18 verschiedene Feldbezeichner zur Beschreibung der Artikel (A**)und neun Felder zur Bezeichnung der Lieferanten (F**).

1.6.2 Messwerte

Typ: Microsoft Access Datei

Format: Datenbanktabelle

Jede Datei enthält 1-n mal Datensätze in folgendem Format:

Dateiauszug:

ARCHIV_10				
VarName	**TimeString**	**VarValue**	**Validity**	**Time_ms**
RT_OFF	08.04.05 09:31:13	0	2	38450396672,9398
BKNR	08.04.05 10:27:33	898951	1	38450435793,3102
MLSNr	08.04.05 10:27:33	123123	1	38450435793,3102
Leff	08.04.05 10:27:33	193274	1	38450435793,3102
dE	08.04.05 10:27:33	10	1	38450435793,3102
SteghEff	08.04.05 10:27:33	126	1	38450435793,3102
SteghdE	08.04.05 10:27:33	36	1	38450435793,3102
MKrafteff	08.04.05 10:27:33	133,101852416992	1	38450435793,3102
MKStart	08.04.05 10:27:33	177,604156494141	1	38450435793,3102
MKEnde	08.04.05 10:27:33	0	1	38450435793,3102
MaschNr	08.04.05 10:27:33	39	1	38450435793,3102
Mitarbeiter	08.04.05 10:27:33	3534	1	38450435793,3102
Storno	08.04.05 10:27:33	-1	1	38450435793,3102

1.6.3 Prüfdaten

Typ: Textdatei

Format: Fixed Record Length

Die ersten zwei Zeilen enthalten aus Gründen der Übersichtlichkeit die Spaltenheader, danach folgen n-Zeilen mit Datensätzen, wobei jede Zeile einen kompletten Datensatz repräsentiert.

Dateiauszug

		BED	BED	GEB	GEB			BED	…
SNR	EXNR	STANDORT	STT	STANDORT	STT	BENENN	SPUEKLASS	KOST	
0113404	D55	01	00	01	01	MESSSCHRAUBE	MSRB25A0,010	5370	…

Aus Gründen der Übersichtlichkeit sind nur die ersten neun der 15 Datenfelder dargestellt.

1.6.4 Artikel Import SAP/VDO

Typ: Microsoft Excel Datei

Format: Excel Datenbanktabelle mit Spaltenüberschriften als Feldbezeichner

Die Datei enthält 1-n Datensätze im folgenden Format. Die Spaltenüberschriften werden dabei nicht wiederholt. Jede Zeile enthält einen Datensatz.

Dateiauszug

SAP-Nr	WERK	VDO-Nr	Kundengeräte-Nr	Gerätebezeichnung	Typ
A2C53084979	24	110 080 230 014	98764110301	KOMBI-INSTRUMENT	987
…	…	…	…	…	…

1.6.5 Prüfpläne

Typ: Textdatei

Format: Fixed Record Length

Dies ist ein Fixed Record Length Format ohne beschreibende Überschriften. Die Feldbezeichnungen ergeben sich rein aus der Formatbeschreibung des Herstellers.

Dateiauszug

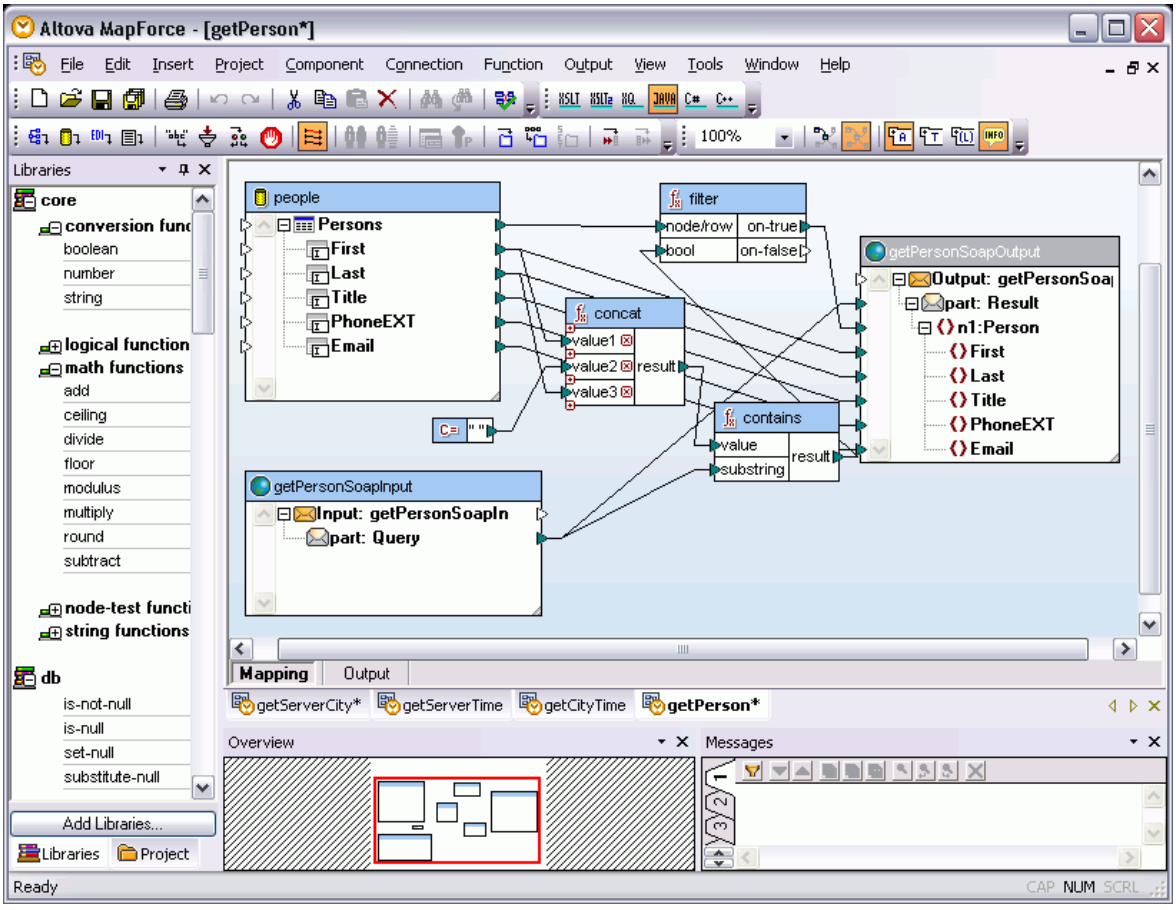
```
1QA0001I PP5121010PAL16L8ACN      REVI
Programmierbare Array Logik   14000000000000000000009700   1 71205
```

Diese Liste ist nicht abschließend, sondern soll einen Überblick über gängigere auftretende Formate vermitteln. Das SUI-Tool soll mit allen diesen Formattypen umgehen können und auch weitere Optionen wie Comma Separated Values und XML berücksichtigen.

1.7 Mapping

Zunächst stellt sich die Frage, wie das Mapping, d.h. die Abbildung von Quell- auf Zielformat, realisiert werden soll. Es gibt einige kommerzielle Tools, die sich auf

Datenmapping spezialisiert haben. Ein Beispiel wäre Mapforce® von der Firma Altova[10], die durch Tools wie XmlSpy bekannt geworden ist. Mapforce® ist ein sehr mächtiges Mapping-Tool, das in der Professional Edition unterschiedlichste Formate, von Textfiles über Datenbanken und XML bis EDIFACT, über ein graphisches Frontend mappen kann. Die lange Feature List des Produktes enthält auch durchaus attraktive Punkte, wie eine automatisierte Codeerzeugung in verschiedene Programmiersprachen, Umwandlung in WSDL-Transaktionen und Beschreibungen, Integration in Visual Studio und Eclipse und vieles mehr. Dieser Screenshot lässt erahnen, was alles hinter diesem Tool steckt:



Quelle: http://www.altova.com/products/mapforce/data_mapping.html

Ein Tool dieser Art wirft aber auch einige Probleme auf. Zunächst wären die anfallenden Lizenzkosten zu nennen, die nicht unerheblich sind. Zum anderen soll das SUI-Tool primär von SINIC-Kunden bedient werden. Dies wirft aber bestimmte Ansprüche an die Bedienerfreundlichkeit auf. Der Kunde soll ohne eine langwierige oder aufwändige Schulung in der Lage sein, seine Formate an die SINIC-Systeme anpassen zu können. Es ist

[10] http://www.altova.com

ein sehr spezialisierter Teilbereich. Ein Tool wie Mapforce® kann diesen leichten Zugang aber nicht bieten, da die gebotene Funktionsfülle vergleichsweise Overkill darstellt. Zudem soll das SUI-Tool voll integriert in die bestehenden SINIC-Module ausgeliefert und verwendet werden, was zum einen optische Konsequenzen im Hinblick auf das SINIC Corporate Design mit sich bringt, zum anderen auch Anforderungen an den Programmcode stellt, um eine volle Integration und nahtlose Anbindung an das oben bereits dargestellte SINIC-Framework zu gewährleisten.
Es wurde daher einer kompletten Eigenentwicklung unter Ausnutzung der bestehenden Ressourcen den Vorzug gegeben. Dies bedeutet, das geeignete, auf die Problemstellung bezogene Mappingverfahren zu entwickeln sind, die an eine übersichtlich zu bedienende graphische Oberfläche gekoppelt sein müssen. Besondere Berücksichtigung muss die Flexibilität und Erweiterbarkeit der verwendeten Verfahren erhalten, da neu hinzukommende Kunden eventuell Datenformate verwenden, die nur geringe Übereinstimmungen mit den bisher Existierenden aufweisen.

1.8 Sicherheit

Das Thema Sicherheit hat bei Datentransfers eine hohe Priorität. Es muss garantiert sein, dass nach der Umwandlung der Importdaten in ein anderes Format keine Daten verloren gehen und es zu keinen ungewollten Änderungen kommt, die dazu führen könnten, dass das gewandelte Datenmaterial nicht mehr valide ist. Dies gilt ebenso für das Übernehmen des Datenbestandes in die entsprechenden Datenbanken. Um Transparenz über die durchgeführten Aktionen der Software zu haben, ist ein Logging zu implementieren, dass eine nachvollziehbare Historie aller Transaktionen erlaubt. Der Transaktionsschutz auf Datenbankebene wird durch Integration des SINIC-Frameworks erreicht, das entsprechende Routinen bereits vorsieht.
Da das SUI-Tool als lokale Anwendung auf Kundenrechnern läuft und nicht in einer unsicheren Zone wie dem Internet, läuft der Schutz gegen dolose Dritteinwirkung über ein Zugriffsberechtigungssystem, dass SINIC für viele Anwendungen standardmäßig verwendet. Ein zusätzliches System, wie z.B. eine Verschlüsselung der Daten mit kryptographischen Mitteln, ist aufgrund des vorliegenden Risikoprofils zunächst nicht angedacht. Dieser Punkt müsste für den Fall einer Erweiterung auf internetverfügbare (Web-) Schnittstellen gegebenenfalls neu überdacht werden.

1.9 Performance

Datenimport- und Exportoperationen unterliegen von Haus aus Performanceüberlegungen. Diese werden maßgeblich durch die Datenverbindung zwischen den betroffenen Systemen, dem Datenumfang und den in den Prozess involvierten Algorithmen geprägt. Die größte Nenngröße hierbei sind die zu bewegenden Datenmengen. Laut den bei SINIC gewonnenen Erfahrungswerten sind die Datenmengen, die auf einmal im System einlaufen, überschaubar. Die zu verarbeitenden Datensätze können zwar mehrere Tausend stark sein, aber es handelt sich nicht um zeitkritische Prozesse. Diese Gegebenheit vorausgesetzt, zusammen mit der Tatsache, dass die Kundensysteme im eigenen Netzwerk interagieren und somit kein Leitungsengpass in der Übertragungsrate zu erwarten ist, lässt zunächst keinen Bedarf für eine außergewöhnliche Komprimierung der Daten oder einen besonders zeitoptimierten Algorithmus feststellen. Nichtsdestotrotz soll bei qualitativ äquivalenten Techniken die performantere den Vorrang erhalten.

2 Konzept

Zur Lösung der gestellten Aufgabe sind zwei große Komplexe zu betrachten. Zum einen die Gestaltung und Umsetzung der grafischen Benutzeroberfläche (GUI), zum anderen die Mechanismen zum Datenmapping und den automatisierten Transfer der Quelldaten in das Zielformat unter Zuhilfenahme der Generierung eines universell tauglichen, automatisiert lesbaren Zwischenformats.

2.1 GUI – das Frontend

Verschiedene Aspekte sind für das Oberflächendesign zu berücksichtigen. Zunächst muss die GUI benutzerfreundlich sein. Benutzerfreundlichkeit, oder auch „Useability“, ist wie folgt definiert:
„Das Ausmaß, in welchem ein Produkt durch bestimmte Benutzer in einem bestimmten Nutzungskontext genutzt werden kann, um bestimmte Ziele effektiv, effizient und zufriedenstellend zu erreichen.“[11]

[11] DIN- ISO 9241-11

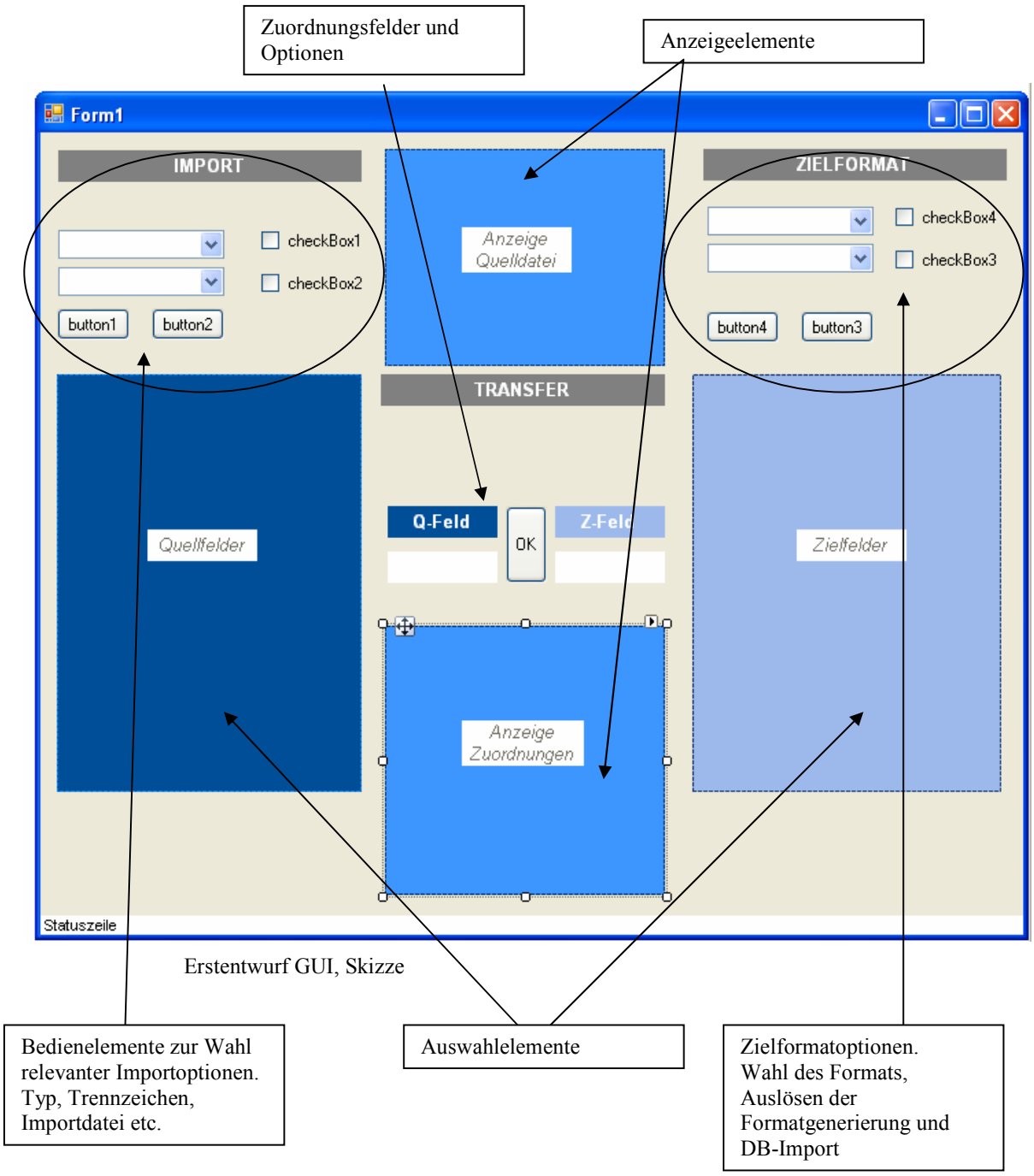
Im Kontext einer Software, die primär auf eine Nutzung im Firmenumfeld abzielt, ist diesem Aspekt m.E. Priorität gegenüber anderen Designaspekten zu geben.

Um dieses Ziel zu erreichen sind verschiedene Ansätze eingeflossen. Zum einen sollen die wichtigsten Funktionen wenn möglich zentral in einer Benutzeroberfläche zu finden sein, um ein häufiges und eventuell verwirrendes Umschalten zwischen mehreren Oberflächen zu minimieren. Zum anderen soll eine klare, intuitive optische Struktur gewählt werden, die dem User auf übersichtliche Weise deutlich macht, welche Aktionen er zu welchem Zweck durchführt.

Angedacht ist zu diesem Zweck eine primäre Oberfläche in Form einer .NET Windows - Form, die den Fluss der Software steuert und den Anwender bei seiner Tätigkeit unterstützt. Andere GUI-Systeme wie TK[12], oder das bekannte QT von Trolltech,[13] sind an dieser Stelle nicht praktikabel, da durch die Fixierung auf .NET die Einhaltung des SINIC Look&Feel, und nicht zu vergessen die Integration des SINIC-Frameworks, die Verwendung der SINIC-Controls und -Forms firmenseitig zwingend vorgeschrieben ist.
Da es bei der Tätigkeit des Benutzers hauptsächlich um das Erstellen von Zuordnungen, Quellfeld zu Zielfeld, geht, sollen diese Bereiche klar voneinander getrennt sein und eine Übersicht über die bereits erstellten Zuordnungen als verbindendes Element zwischen Quelle und Ziel in der Bildschirmmitte zentral positioniert werden. Unterschiedliche Farbgebung soll diese Bereiche optisch noch deutlicher voneinander trennen. Zusätzliche Informationsfelder und Schalter für Optionen sollen nicht in Menüleisten auftauchen, sondern in unmittelbarer Nähe des bezogenen Bereichs befindlich sein, um dem Nutzer Suchvorgänge zu ersparen und benötigte Wahlmöglichkeiten kontextnah zu präsentieren.
Bei der Wahl der .NET-Controls soll, wenn möglich, eine einheitliche Darstellungsform für einander ähnelnde Daten gewählt werden. Hier sind besonders die Import- und Exportbereiche zu nennen, zwischen denen Bezüge hergestellt werden sollen. Bei größeren Informationsblöcken oder Ansichten, die keine unmittelbare Userinteraktion erfordern, können je nach Bedarf freie Bereiche in der Windows-Form verwendet oder neue Forms im modalen „Pop-up“-Stil verwendet werden.

[12] http://www.tcl.tk/
[13] http://www.trolltech.com/

Im Zusammenhang betrachtet stellt sich die Oberfläche in einem ersten Entwurf folgendermaßen dar:



Erstentwurf GUI, Skizze

Die Oberfläche teilt sich grob in drei Bereiche ein. Den Importbereich links, den Exportbereich rechts und den Zuordnungs- und Informationsteil in der Mitte. Teile der Elemente können als Reaktion auf Benutzeraktionen ein- oder ausgeblendet werden.
Ziel des Aufbaus ist es, dem User eine Bearbeitung in der in der westlichen Welt üblichen Leserichtung von links oben nach rechts unten, zu ermöglichen. Zuerst wird links oben das Quellformat gewählt, dann auf gleicher Höhe rechts das Zielformat, um schließlich etwas weiter unten in der Mitte die Zuordnungen beginnen zu können.

2.1.1 Importbereich

Der Importbereich stellt alle für den Import relevanten Elemente dar. Die Reihenfolge soll, von oben nach unten betrachtet, die Reihenfolge der Arbeitschritte des Users widerspiegeln. Zunächst besteht im oberen Bereich die Möglichkeit zur Wahl des Importformats und zum Öffnen einer Beispieldatei, die die für die Zuordnungen relevanten Felder enthält. Darunter erscheinen formatspezifische Controls zum Einstellen von Optionen. Hervorzuheben sind die Optionsfelder für den Import eines FRL-Formats. Bedienfelder für die Eingabe der Feldnamen und Feldlängen sind an dieser Stelle einzurichten.

Als Ergebnis der getroffenen Auswahl werden darunter die Feldelemente des Importformats angezeigt. Der User soll bei der Selektion der verfügbaren Felder unterstützt werden; eine möglichst einheitliche Darstellungsform wird daher angestrebt. Die Darstellung der Auswahlelemente soll, wenn praktikabel, in Form tabellarischer Listen erfolgen, die die Importfelder von oben nach unten gelistet anzeigen. Da diese Vorgehensweise für XML- und Access-Formate keine übersichtliche Darstellung erwarten läßt, wird für diese Formate eine Darstellung in Baumform angestrebt.

2.1.2 Zuordnungsbereich

Der obere Teil des Zuordnungsbereichs enthält den Bereich für die Anzeige der Quelldatei. Dieser soll bei Textdateien eingeblendet werden, um dem Benutzer eine Kontrollmöglichkeit zwischen Quelldatei und der durch das SUI-Tool identifizierten Importfelder zu ermöglichen. Darunter befinden sich die Felder, die das aktuell

ausgewählte Quell- und Zielfeld anzeigen. Der abgebildete „OK“-Button steht in der Skizze stellvertretend für die Buttons „zuordnen“, “speichern“ und „löschen“, hinter denen sich die grundsätzlichen Funktionen des Transaktionsbereichs verbergen. Ein Klick auf „zuordnen“ transferiert den Inhalt der Quell- und Zielfeldboxen in die darunter befindliche Zuordnungsliste und löst im Backend die Zuordnung aus, „löschen“ löscht einen Eintrag aus der Liste und „speichern“ löst die Speicherung der aktuellen Schnittstelle aus.

2.1.3 Exportbereich

Im Exportbereich werden die Informationen zum Zielformat dargestellt. Die User kann das gewünschte Zielformat im oberen Bereich wählen. Diese Auswahl führt dazu, dass die Felder des Zielformats angezeigt werden. Diese sind nach drei Ebenen gruppiert. Jede Ebene stellt eine für das Zielformat relevante Datenbanktabelle des Zielformats dar und wird daher auch in tabellarischer Form dargestellt. Die Inhalte der Tabellen werden automatisiert aus den SINIC-*XmlDBSchemas* ausgelesen. Ebenfalls zu finden sind die Buttons, die nach beendeter Zuordnung die Erzeugung des SINIC-Formats und den Systemimport auslösen.

2.1.4 Coupling

Die GUI muß, wie aus der o.g. Systematik hervorgeht, im engen Zusammenspiel mit den darunter liegenden Im- und Exportkomponenten funktionieren. Auf der Datenebene findet die Verknüpfung hauptsächlich über die verschiedenen Datencontainer statt.
Im Rahmen eines veränderlichen Umfelds und der Möglichkeit sich ändernder Formate, vor allem im Importbereich, ist eine zu enge Verdrahtung zwischen Container und GUI nicht wünschenswert. Die Objektverbindungen zwischen GUI und Container beziehen sich daher ausschließlich auf Interfaces. Nötige Verbindungen verschiedener Systemkomponenten werden weitgehend über Komposition realisiert. Vererbungsdependenzen sollen, wenn möglich, minimiert werden.

2.2 Verarbeitung und Systeme – das Backend

2.2.1 Anforderungen

Zuerst sollen die Anforderungen näher definiert werden, die die Software typischerweise zu bewältigen hat.

Allgemein ausgedrückt, müssen Importdateien in verschiedenen Formaten gelesen werden können. Gelesen bedeutet in diesem Falle mehr als das technische Einlesen. Um für ein Mapping tauglich zu sein, muss die Datei in einem weiteren Schritt analysiert und die enthaltenen Daten extrahiert werden. Aus diesem Grund ist eine geeignete Datenstruktur zu konzeptionieren, die die so ermittelten Daten aufnimmt. Die Ergebnisse der Analyse müssen in gefilterter Form auch der GUI verfügbar gemacht werden, um eine zweckmäßige Anzeige zu ermöglichen.

Für die Zielformate gilt ähnliches. Hier liegen die benötigten Daten in den im SINIC-Framework-Kapitel bereits erwähnten *XmlDBSchema*-Dateien vor. Diese Dateien müssen gelesen und die nötigen Formatinformationen extrahiert werden. Die ausgelesenen Informationen müssen nach entsprechender Aufbereitung der GUI zur Verfügung gestellt werden. Auch dieser Vorgang erfordert eine zur Bearbeitung taugliche und flexible Datenstruktur zur Aufnahme der Daten.

Liegen Import- und Zielformatdaten vor, muss die Zuordnung von Quell- zu Zielfeldern möglich sein. Diese Zuordnungen müssen geeignet erfasst werden und bearbeitet oder rückgängig gemacht werden können.

Sind alle Zuordnungen getroffen, muss dem User die Speicherung seiner Zuordnungen unter Angabe einiger zusätzlicher Informationen möglich sein. Die in diesem Schritt gespeicherten Daten stellen eine vollwertige Schnittstelle zwischen Quell- und Zielformat dar, die alle notwendigen Daten enthalten muss, um eine Übertragung von Quelle zu Ziel ohne zusätzlichen Input des Users möglich zu machen. Daher muß ein Datenformat entwickelt und technisch umgesetzt werden, das die Schnittstelle gemäß den

Anforderungen abbilden kann. Auch eine Bearbeitung bereits bestehender Schnittstellen muss möglich sein.

Zudem muss die Software als weitere, grundlegende Funktionalität mittels einer Importdatei und einer Schnittstellendatei eine Ausgabedatei generieren können, die ungeachtet des Zielformats automatisiert lesbar den Import in die SINIC-Datenbanken ermöglicht (SINIC-Systemimport). Zu diesem Zweck muss ein taugliches Format zur Beschreibung der in den SINIC CAQ -Systemen üblichen Formaten und Daten konzipiert und umgesetzt werden.

Gruppiert man diese Anforderungen inhaltlich, so erhält man folgende Gruppen:

1. Entwicklung geeigneter Formate zur
    a. Speicherung und Wiederherstellung von Schnittstellen,
    b. Speicherung eines universellen Formates für den SINIC-Systemimport
    c. Aufnahme der während der internen Prozesse anfallenden Import- und Exportdaten

2. Programmatisches Handling der
    a. Auswertung der Importdateien und XmlDBSchemas
    b. Mapping von Importformat zu Zielformat
    c. Interaktion des Backends mit der GUI
    d. Generierung und Serialisierung der unter 1a und 1b genannten Formate
    e. Durchführung des SINIC-Systemimports

Außerdem ist die Interaktion des Users mit der GUI durch entsprechende Routinen im Backend selbstverständlich so zu behandeln, daß ein Abweichen vom geplanten Ablauf nicht zu einem Absturz der Anwendung führt oder in einem nicht validen Datensatz resultiert.
Um diesen Anforderungen gerecht zu werden, ist zunächst generell über die einzusetzenden Techniken und die Architektur der Software nachzudenken.

2.2.2 Formatentwicklung

Zunächst zu betrachten ist die Datenstruktur für die zu serialisierenden Daten. Das Augenmerk fällt hierbei schnell auf die Extensible Markup Language, XML. Die Sprache liegt aktuell in der vom World Wide Web Consortium (W3C)[14] am 16. August 2006 vorgelegten und am 29. September 2006 überarbeiteten Recommendation vor[15]. XML erhält im Datenaustausch eine immer größere Bedeutung, da die Sprache für die strukturierte Abbildung von Daten gut geeignet ist. Es erscheint daher nicht sinnvoll, an diesem Trend vorbeizugehen und einen anderen Weg zu beschreiten, welcher z.B. in einem proprietären Text- oder Binärformat münden würde. Speicherung der Schnittstellen innerhalb einer Datenbank erscheint ebenfalls nicht sinnvoll. Zum einen widerspricht es der für die Schnittstellen geforderten flexiblen Struktur, zum anderen erhöht es den internen Verwaltungsoverhead.

Auch im Hinblick auf eine eventuell stattfindende Umsetzung der Software als Internet-Anwendung bietet sich XML an. Die Software ist mit diesem Format zukunftssicher aufgestellt und kann flexibel an neue Anforderungen angepasst werden. Diese Faktoren führen im Ergebnis zu einer Nutzung von XML zur Speicherung der Daten. Die Verwaltung der gespeicherten Schnittstellen erfolgt userverantwortlich auf Dateiebene.

2.2.3 XML-Dokumenttypen

Zu betrachten sind zwei XML-Dokumenttypen. Zum einen das Format, das die Schnittstellenbeschreibung abbildet, zum anderen das Format, das die Daten für den SINIC-Systemimport aufnimmt.

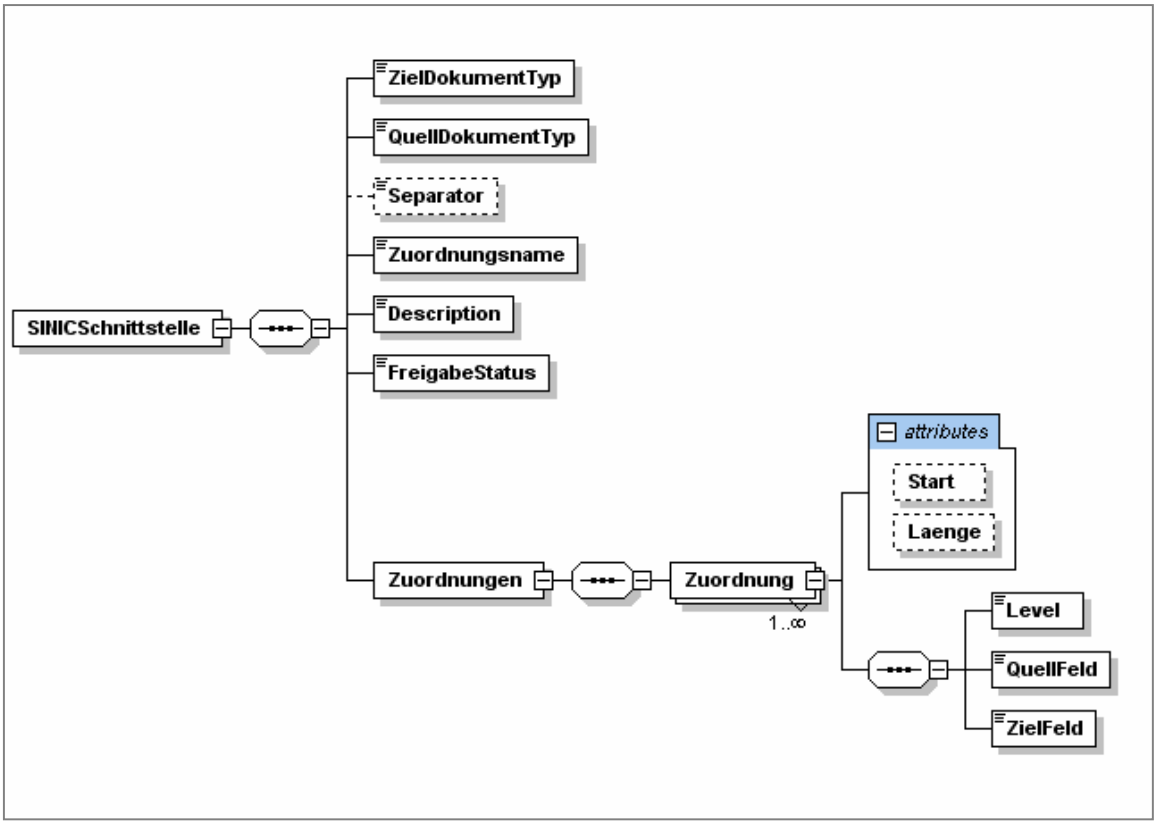
Dokumenttyp SINICSchnittstelle

Festzulegen sind die Daten, die zur Beschreibung der Schnittstelle nötig sind. Da auf Seiten von SINIC die Speicherung in Datenbanken erfolgt, liegt es nahe, dass Datenbankfelder in die Schnittstelle einfließen müssen. Da manche Importe mit mehr als einer Datenbank

[14] http://www.w3.org/
[15] http://www.w3.org/TR/REC-xml/

interagieren, muss ein entsprechender Indikator vorgesehen werden, der die „Ebene“ des Feldes beschreibt. Um hier nicht in ungewollte fixe Abhängigkeiten zu geraten, wird dieser Indikator abstrakt als Zahl gewählt. Auf der Seite der Datenquelle muss natürlich ebenfalls ein Bezeichner für das Quelldatenfeld vorhanden sein, damit eine Zuordnung möglich ist. Zu berücksichtigen ist hierbei, dass für FRL-Formate die Start- und Endpositionen der einzelnen Felder mit in die Schnittstelle einfließen müssen, da diese nicht aus der Quelldatei rekonstruiert werden können. Diese benannten Informationen müssen für jede Feldzuordnung gespeichert werden. Die Anzahl dieser Zuordnungen ist nach oben unbegrenzt und stellt in der Summe den Datenteil des Dokuments dar. Um den Datenteil in einen Kontext zu setzen, werden in einem vorangestellten Kopfteil (Head) einige Zusatzinformationen festgelegt. Diese liefern Informationen zum Typ des Zielformats (ZielDokumentTyp) und Quellformats (QuellDokumentTyp), einen beschreibenden Namen für diese spezifische Schnittstelle (Zuordnungsname), eine Beschreibung (Description) und eine Angabe zum Freigabestatus. Der Freigabestatus ist dafür vorgesehen Mitarbeitern zu ermöglichen, bestimmte Schnittstellen zu sperren oder freizugeben. Zugriff auf die Schnittstelle ist dann zwar im SUI-Tool möglich, nicht aber die automatisierte Verarbeitung. Auf diese Weise ist die Schnittstellendatei allgemein genug gehalten, um sehr unterschiedliche Schnittstellen zu ermöglichen, aber spezifisch genug, um den Datentransfer zu ermöglichen.

Es ergibt sich im Überblick folgende Struktur:



SinicSchnittstelle-Schema; die Schema-Beschreibung befindet sich im Anhang

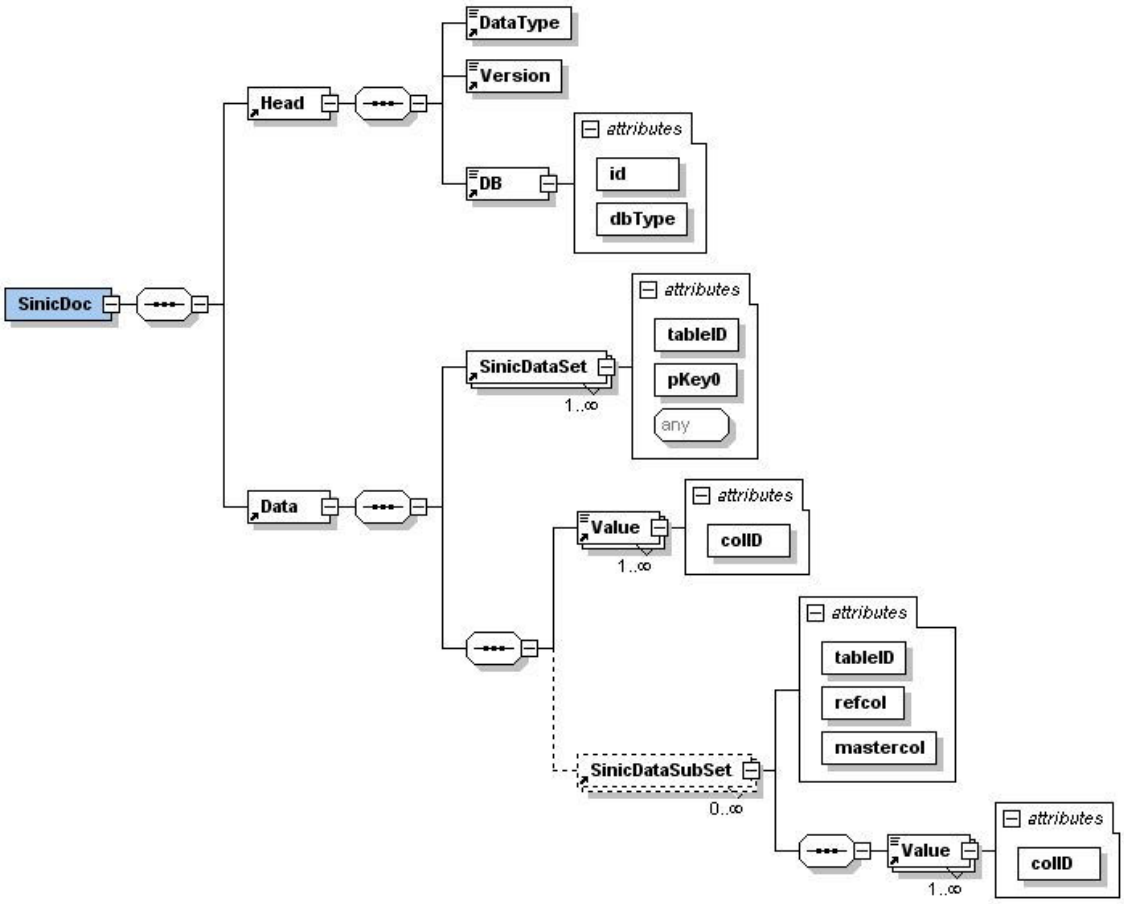
Dokumenttyp SinicDoc

Der SINIC-Dokumenttyp *SinicDoc* stellt das universelle Format dar, das den automatisierten SINIC-Systemimport ermöglichen soll. Um dieser Aufgabe und den zum Teil sehr unterschiedlichen SINIC-Formaten gerecht zu werden, ist er sehr allgemein gehalten. Die grundlegende Problematik hierbei liegt in der Struktur der abzubildenden Daten begründet. Eine zu importierende Datei kann verschiedene Datenebenen beinhalten. Als Beispiel soll der CAQ-Artikelimport dienen. Er enthält 1-n Artikel, von denen jedem 0-n Lieferanten zugeordnet sind. Es muss gewährleistet sein, dass zum einen die Daten korrekt erfasst werden, zum andern aber auch der interne Zusammenhang zwischen Lieferanten und Artikeln bestehen bleibt. Da es Zusammenhänge dieser Art in mehreren Variationen innerhalb der SINIC-Systeme gibt, muss die Datenstruktur möglichst abstrakt gehalten und von den spezifischen betroffenen Datenbankfeldern abgekoppelt werden. Das Resultat für die XML-Struktur ist, dass Felder nicht als Elemente angelegt werden können, sondern als Werte innerhalb von Elementen oder Attributen abgebildet werden müssen. Um

diese Schachtelung der Ebenen generell zu realisieren, gäbe es verschiedene Möglichkeiten. Eine wäre, jedem Element der ersten Ebene (Artikel) eine Id zuzuweisen, die sich dann in allen untergeordneten Elementen (Lieferanten) wiederfindet. Dies ergäbe prinzipiell eine Liste mit allen Elementen der ersten Ebene, gefolgt von einer Liste mit allen Elementen der zweiten Ebene. Eine andere Möglichkeit wäre, eine Struktur zu bilden, die der Struktur im Quellformat ähnelt. Dies hätte den Vorteil, dass sich die Struktur im Dokument selbst leichter nachvollziehen lässt und vom logischen Aspekt her stimmiger ist. Jeder Datensatz der ersten Ebene wäre nach seiner Beendigung inklusive aller Unterelemente komplett erfasst. Dies macht auch im Hinblick auf weitergehende Optionen Sinn. Zum Beispiel könnte ein System auf diese Weise Datensätze auch zeitlich versetzt nacheinander erfassen. Bei der ersten angesprochenen Variante wäre dies nur unter Schwierigkeiten möglich, da erst nach Erfassen aller Elemente die Ids für die Zuordnungen feststehen bzw. die Subelemente erst nach Erhalt aller Primärelemente übertragen werden. Aus diesem Grund habe ich mich für die zuletzt genannte Variante entschieden.

Die übergeordnete Einteilung erfolgt in ein *Head*- und ein *Data*-Segment. Im *Head* werden Informationen zur Art des Systemimports, der Versionsnummer und der bezogenen Datenbank erfasst. Das *Data*-Element enthält alle Importdaten. Die Daten der ersten Ebene werden in den Elementen mit der Bezeichnung *SinicDataSet* erfasst. Daten tieferer Ebenen werden in *SinicDataSubSet*-Elementen gespeichert. Ein *SinicDataSet*-Element kann 0-n *SinicDataSubSet*-Elemente enthalten. Diese Subsets enthalten die Daten zweiter oder dritter Ebene, die diesem Primärdatensatz zugeordnet sind. Ein Beispiel wäre ein bestimmter Artikel, dem mehrere Lieferanten als Bezugsquellen zugeordnet sind. Da jede Ebene unterschiedliche Datenbanktabellen darstellt, muss durch entsprechende Angaben im XML-Code die Zuordnung und Referenzierung ermöglicht werden. Zu diesem Zweck sind in den *SinicDataSet* und *SinicDataSubSet*-Elementen Attribute enthalten, die diese Zuordnung ermöglichen. *SinicDataSet* listet sowohl den Namen der Datenbanktabellen als auch die Keys der Tabelle auf. Die Subsets enthalten den Tabellennamen sowie die als Keys verknüpften Spalten der eigenen und übergeordneten Ebene. Durch die gewählte Struktur bedingt entstehen zwangsweise einige Redundanzen in den Datasets. Dies erzeugt zwar einen etwas längeren XML-Code, vereinfacht aber auf der anderen Seite die automatisierte Verarbeitung des Formats.

Visualisiert sieht die Struktur folgendermaßen aus:



SinicDoc-Schema; die Schemabeschreibung befindet sich im Anhang

Alle Inhalte, die in die Datenbanken importiert werden sollen, befinden sich in den *Value*-Elementen. Diese Struktur soll einen überschaubar zu handhabenden, gut automatisierbaren Import begünstigen.

2.2.4 Architektur

Ein vorrangiges Ziel bei dem Aufbau der Software ist die Möglichkeit der Erweiterbarkeit und der Flexibilität, auf neue Formate und Anforderungen schnell reagieren zu können.

Dies wird bestimmt durch stark veränderliche Formate, die die Kunden in ihren Systemen generieren. Glücklicherweise zielen einige softwaretechnische Prinzipien genau auf dieses

Erfordernis hin. Eines der Grundprinzipien des „guten Tons“ des Softwaredesigns ist das Prinzip:

„Identify the aspects of your application that vary and separate them from what stays the same”[16] .

Bereiche, die wahrscheinlich häufig Änderungen unterworfen sind, sollen von den restlichen Bereichen abgekapselt werden. Dadurch soll verhindert werden, dass sich Änderungen in der Implementierung des fraglichen Teilbereichs durch weit reichende Teile der Anwendung ziehen. Kapselung trennt diese Bereiche vom Rest der Implementierung.
Daraus folgt ein weiteres Grundprinzip:

„Program to an interface, not an implementation.“[17].

Eine Klasse, die sich auf die Implementierungsdetails einer anderen Klasse verlässt, ist zwangsläufig von dieser abhängig. Es macht daher Sinn, wichtige Eigenschaften und Algorithmen zu Interfaces zusammenzufassen welche die am Prozess beteiligten Klassen einhalten müssen. Auf diese Weise ist die Klasse, die mit dem Interface arbeitet, von den eigentlichen Objekten, die die Interfaces implementieren, entkoppelt. Dabei muss nicht zwingend ein Interface im engeren Sinne[18] verwendet werden, sondern es kann ebenso auch eine abstrakte Klasse realisiert werden.

Das Design-Prinzip „Favor composition over inheritance”[19] legt die Vorteile der Klassenkomposition über Vererbung dar. Systeme, die Kompositionen verwenden, sind flexibler und können ihr Verhalten zur Laufzeit ändern.

Da, wie bereits oben erwähnt, C# kein *multiple inheriting* von Klassen zulässt, eine Klasse aber mehrere *Interfaces* implementieren kann, stellt das Konstrukt *Interface* ein flexibles Hilfsmittel dar.

[16] HFDP, Seite 9
[17] HFDP, Seite 11
[18] Als Konstrukt „Interface“ in .NET zum Beispiel
[19] HFDP, S. 23

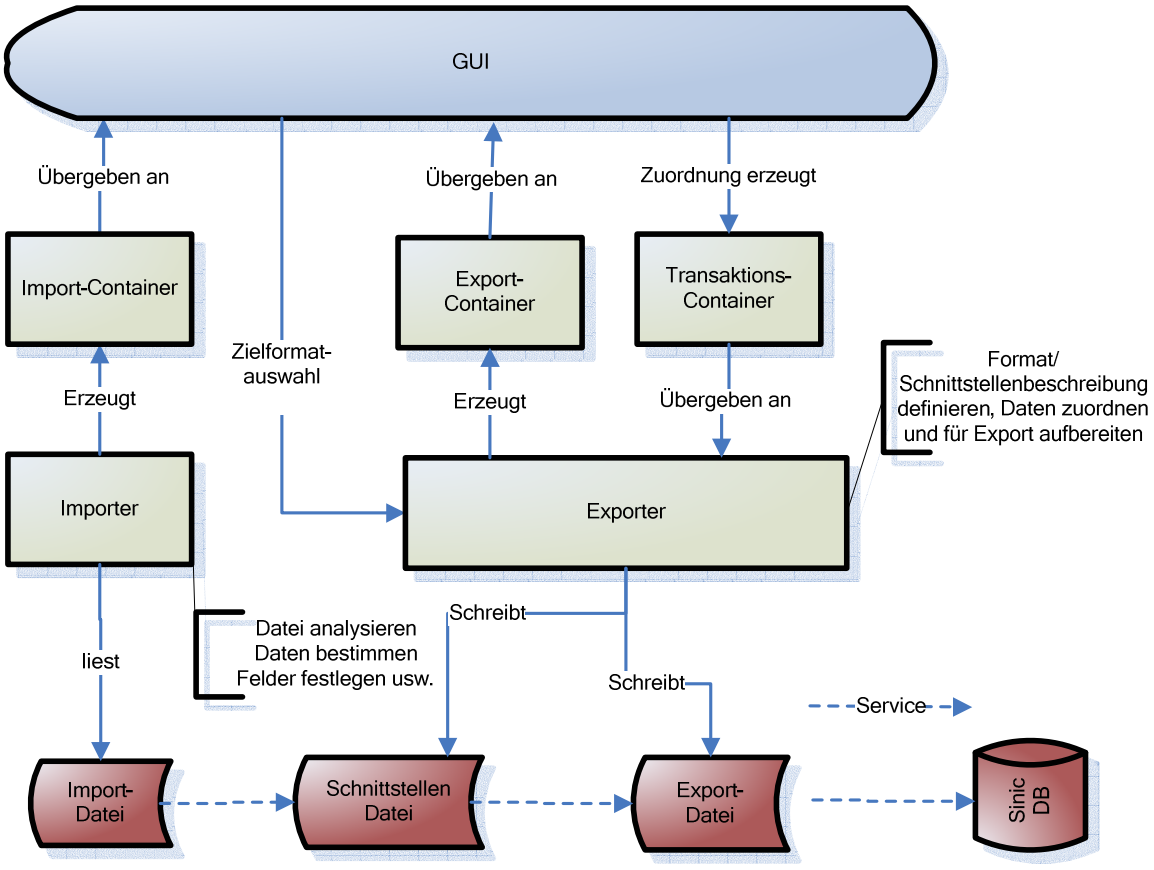
Es sollen, soweit möglich, die o.g. Design-Prinzipien beachtet werden. Sie finden in vielen Entwurfsmustern Verwendung und helfen, die Software gut wartbar und flexibel zu halten.

Zunächst müssen die kritischen Bereiche allerdings identifiziert werden. Stellt man sich die Situation im Hinblick auf die gegebenen Anforderungen vor, so stellt der Importbereich sicher die veränderlichste Komponente dar, auch im Hinblick darauf, dass SINIC auf diesen Bereich den wenigsten Einfluss hat. Andererseits ist aber auch der mit dem Zielformat verknüpfte Exportteil nicht frei von Änderungen. Es kommt laut Angabe der SINIC-Mitarbeiter auch immer wieder vor, dass in einzelnen Formaten neue Felder hinzukommen. Auch das Hinzukommen eines neuen internen Zielformats kann nicht ausgeschlossen werden. Dementsprechend muss darauf geachtet werden, dass die mit diesen Bereichen verknüpften Daten von den darunter liegenden Dateien und Formaten weitestgehend entkoppelt werden. Da diese Daten einander zugeordnet werden müssen, macht es Sinn, auch diese Zuordnung zu entkoppeln und in einer eigenen Datenstruktur zu speichern.

Ausgehend von dieser Idee sind drei verschiedene Typen von „Containern“ zu entwickeln: für die Importdaten, die Daten des Zielformates und die Daten. die die Zuordnungsinformationen aus beiden aufnimmt. Diese werden in jeweils eigenständigen Klassen entwickelt (blImpCon, blExpCon, blTrans), die die entsprechenden Interfaces *IImportable*, *IExportable* und *ITransformable* implementieren. Sie werden im Weiteren als Import-Container, Export-Container und Trans-Container bezeichet. Diese Container stellen das Bindeglied zwischen dem Backend und der GUI dar und entkoppeln die relevanten Systeme.

2.2.5 Systematik

Folgendes Schaubild zeigt die grundlegende Systematik.



Systematik

Wie man sehen kann, ist die Software in drei Ebenen unterteilt. Auf der untersten Ebene (rot) finden die Interaktionen mit Dateien und Datenbanken statt. Darüber, im gemäß der SINIC-Struktur bezeichneten *Business Layer* (beige), schlägt das Herz der Anwendung. Die Importer- und Exporter-Klassen, die die erforderlichen Kernfunktionalitäten der Software umsetzen, sind hier ebenso angesiedelt wie die oben erwähnten Containerklassen, die die Dateninteraktion zwischen Importer, Exporter und GUI kapseln. Die oberste Schicht (blau) bildet naturgemäß die GUI, realisiert als .NET-Forms.

Eine generelle Frage bei der Architektur einer Software ist immer die Anbindung des Frontends an die dahinter liegende Programmlogik. Es ist hier abzuwägen, inwieweit auf der einen Seite die softwaretechnisch geforderte Entkopplung zwischen Front- und Backend realisiert werden kann, zum anderen die im .NET-Framework integrierten Möglichkeiten zum direkten Anbinden von Windows-Controls an Datenquellen, die vor allem in zentralen Controls wie *Listboxes*, *DataGrids* und *DataGridViews* zum Tragen

kommen, zu nutzen. Der hier gewählte Weg versucht die Kontakte zu selbstentwickelten Klassen weitestgehend zu minimieren, während eingebaute Möglichkeiten, die .NET-Kernkomponenten verwenden, nach Möglichkeit ausgenutzt werden.

2.2.6 Import und Formate[20]

Kern der Importroutinen sind die oben genannten Import-Container. Alle Importmethoden implementieren Import-Container und geben selbige dann auch zur Darstellung an die GUI zurück. Die relevanten Importformate und die Ideen zum Umgang mit selbigen sollen hier kurz dargestellt werden.

2.2.6.1 Textformate

Textdateien sind per Definition unstrukturiert. Da sie sehr einfach zu generieren sind und nahezu von jedem System gelesen werden können, erfreuen sie sich gerade im Bereich Produktion großer Beliebtheit und haben daher für den Bereich CAQ eine große Relevanz. Die Software unterscheidet in der Bearbeitung die Formate Key:Value, Fixed Record Length (FRL) und Comma Separated Values (CSV).

Key:Value-Formate sind in der Regel dadurch gekennzeichnet, dass sie in dem Format:

{Feld}{Separator}{Wert}

vorliegen, und nur eines dieser Paare pro Textzeile vorkommt. Die Analyse der Kundenformate ergab, dass in allen vorliegenden Fällen ein Doppelpunkt (:) als Separator verwendet wurde, daher wird diese Einstellung als default verwendet. Der Benutzer hat allerdings die Möglichkeit, diese Vorgabe in der GUI manuell zu ändern, sollte er mit dem Ergebnis des Parsens unzufrieden sein.

Fixed Record Length-Formate sind vom Aufbau her meist so strukturiert, dass ein Datensatz pro Zeile gespeichert wird. Wie der Name schon sagt, sind die Datenfelder an

[20] Importformate bezeichnen die kundenseitig vorliegenden Quellformate.

bestimmten Positionen innerhalb der Zeilen festgelegt. Diese Festlegungen müssen der Formatbeschreibung des Herstellers entnommen werden, da es keine realistische Möglichkeit gibt, Feldgrenzen zuverlässig zu „erraten“.

Comma Separated Values (CSV) ist ein sehr beliebtes Format, da die meisten gängigen Tabellenkalkulationen dieses Format direkt im- und exportieren können. Die Vielfalt an Varianten ist bei diesem Format etwas größer als bei den Vorgängern. Generell ist die Grundidee, Daten von einander mittels eines Trennzeichens wie z.B. einem Komma von einander zu separieren. Dies ist zunächst ein ähnliches Verfahren wie bei der Key:Value - Formatierung, unterscheidet sich aber dadurch, das zum einen mehrere Werte in einer Zeile stehen können und zum anderen die Feldnamen meist nicht den Werten vorangestellt sind, sondern in einer separaten ersten Zeile aufgeführt werden. Ist die Feldreihenfolge bekannt, kann auf diese „Überschriftenzeile“ auch ganz verzichtet werden.

Da Kommas für numerische Werte als Trennzeichen nicht ideal sind, findet häufig das Semikolon (;) Verwendung. Als Variante ist ebenfalls möglich, dass die Datensätze nicht zeilenweise getrennt sind, sondern ein bestimmtes Zeichen als Datensatztrenner verwendet wird.

Programmatisch müssen diese Dateitypen unterschiedlich geparst werden, wofür reguläre Ausdrücke besonders geeignet erscheinen.

2.2.6.2 Microsoft Excel

Im korporativen Umfeld spielt die Tabellenkalkulation Excel von Microsoft eine große Rolle. Es ist ein häufig anzutreffendes Format wenn es um die Speicherung und Bearbeitung von Daten aller Art geht. Da viele Anwendungen mit Excel-Formaten arbeiten und diese auch zum Teil schreiben können (z.B. Open Office), ist der Import von Excel(*.xls)-Dateien Teil der Anforderungen. Da die .NET-Plattform ebenso wie Excel von Microsoft entwickelt wurde, liegt es nahe, zunächst nach fertigen bzw. bereits in .NET integrierten Lösungen zu suchen. In der Tat bietet Microsoft einige Bibliotheken an, um mit ihren MS-Office-Produkten zu interagieren. Ein Kernelement der programmatischen Verarbeitung von MS-Office-Daten bzw. der Steuerung von MS-Office-Produkten bilden

die Microsoft Primary Interop Assemblies (PIA): „These primary interop assemblies (PIAs) enable the solution developer to leverage all the new capabilities of the Microsoft .NET Framework and to access Microsoft Office in a reliable and consistent way."[21]. Diese Bibliotheken stehen auf den Microsoft-Webseiten als kostenloser Download zur Verfügung und belasten daher das Projekt nicht finanziell. Nach erfolgreicher Installation integrieren sich die PIA in die unter Windows verfügbaren .COM-Bibliotheken. Im Visual Studio können sie dann über die Referenzverwaltung hinzugefügt werden.

2.2.6.3 Microsoft Access (.mdb)

Im Umfeld der SINIC-Systeme sind sowohl intern als auch extern häufig Access-Datenbanken zu finden. Dementsprechend muss auch für diese Daten ein Import möglich sein. In den meisten Fällen werden pro Format eine bis drei Tabellen angesprochen, die sich in einer einzigen Datenbank befinden. Für Access gilt ähnliches wie für Excel – da Microsoft maßgeschneiderte Bibliotheken für den Umgang mit den Office-Produkten zur Verfügung stellt, sollen, wenn möglich, die in .NET integrierten Möglichkeiten zur Behandlung Verwendung finden. Eine Untersuchung der vorhandenen Möglichkeiten ergab, dass Access-Dateien in .NET prinzipiell wie eine gewöhnliche Datenbank angesprochen werden können. Es sind zwar kleine Besonderheiten bei der Implementierung zu beachten, diese scheinen aber nicht kritisch zu sein. Da die Struktur Datenbank->Tabellen–>Spalten als Baumstruktur abbildbar ist, soll die visuelle Darstellung in eine ähnliche Richtung gehen.

2.2.6.4 XML

Das SUI-Tool soll die Grundlagen schaffen, in einem späteren Schritt auch eine Vielzahl von XML-Formaten auf das SINIC-Format mappen zu können. Momentan findet XML kundenseitig in den für SINIC relevanten Systemen noch kaum Verwendung. Nichtsdestotrotz soll das SUI-Tool auf einen zukünftigen Umgang mit XML vorbereitet werden. Verschiedene Strategien existieren zum Handling von XML-Daten. Programme können das XML-Dokument als reinen Text, als Folge von Ereignissen, als Baum oder als

[21] MSDN, A Primer to the Office XP Primary Interop Assemblies

Serialisierung einer anderen Struktur auffassen.[22] Alle diese Verfahren sind mit Rücksicht auf die Möglichkeiten des .NET-Frameworks zu untersuchen. Das Framework bietet eine umfangreiche Methoden- und Klassensammlung zum spezialisierten Umgang mit XML-Dokumenten an, die kaum Wünsche offen lässt. Vor diesem Hintergrund scheint eine Verarbeitung als reiner Text unwahrscheinlich. Problematischer ist jedoch die Darstellung in einer Benutzeroberfläche. Die automatisierten Möglichkeiten, so wie z.B. das *Databinding* an ein DataGrid, liefern in diesem Fall keine Lösung, da eine unbekannte XML-Struktur viele Ebenen tief verschachtelt sein kann. Die genannte Methodik liefert aber nur die zweidimensionale Abbildung einer Ebene in Tabellenform. Ein DataSet ist nicht als XML-Speicher entwickelt worden und eignet sich nicht für generelle XML-Manipulationen[23]. Sinnvoll erscheint eine Verarbeitung des XML als Baum, da .NET spezialisierte Controls zur Darstellung von Baumstrukturen in Form der *TreeView*-Controls anbietet und sich so die Möglichkeit bietet, die XML-Baumstruktur ohne großen Mehraufwand zu visualisieren. Um auf einzelne Elemente der Struktur gezielt zugreifen zu können, soll die XML Path Language (XPath) verwendet werden. XPath ist eine Nicht-XML-Sprache, die eingesetzt wird, um bestimmte Teile von XML-Dokumenten zu identifizieren.[24] Sie ist in der Version 1.0 in der W3C Recommendation vom 16. November 1999 definiert.[25] Mit ihr ist es möglich, Knoten innerhalb des XML-Dokuments anhand der (relativen) Position, dem Inhalt oder anderen Eigenschaften zu selektieren[26]. XPath wird von .NET nativ unterstützt. Ein Nachteil dieser Methode ist, dass die *XmlDocument*-Klasse von .NET zunächst komplett in den Speicher geladen werden muss, bevor die Verarbeitung mit XPath beginnen kann. Dies kann zu Problemen bei sehr großen Dateien führen, vor allem wenn Bandbreite eine Rolle spielt. Dies ist im vorliegenden Szenario allerdings nicht der Fall.

2.2.7 Zuordnungen

Ein zentrales Element der Anwendung ist die Zuordnung von Quell- zu Zielfeldern. Wie oben beschrieben, findet die Entkopplung der verschiedenen Anwendungskomponenten auf der Datenebene durch die Bereitstellung der verschiedenen Containertypen statt. Die

[22] XMLiaN, S. 316
[23] MSDN, Improving XML Performance
[24] XMLiaN, S. 164
[25] http://www.w3.org/TR/xpath
[26] XMLiaN,S. 164

Transformationscontainer (TransCon) stellen das Bindeglied zwischen Import- und Export-Container dar. Diese Container müssen, ergänzt um die nötigen Formatinformationen, ihren Weg in eine valide Schnittstellendatei finden.

Erforderlich für eine Zuordnung sind zunächst die relevanten Felder des Zielformats. Diese müssen entsprechend der Benutzerauswahl gelesen und angezeigt werden. Programmatisch ist das Handling des Zielformats auf der Ausgabeseite, also im Exporter, angesiedelt. Als Grundlage dienen die im SINIC-Framework verankerten XmlDBSchemas.

2.2.8 Mapping

Nach dem Öffnen und der Darstellung des Zielformats und dem unter *Import* beschriebenen Öffnen des Quellformats ist der nächste logische Schritt die Umsetzung der Zuordnung. Die beiden abstrakten Konstrukte Import-Container und Export-Container müssen einander zugeordnet werden. Zur Erfassung dieser Zuordnung werden keine Properties in den beteiligten Containern verändert, sondern Instanzen einer weiteren Klasse verwendet, die sich ausschließlich um die Zuordnungen kümmert: die Transformationscontainer (Trans-Container). Somit werden die Zuordnungen von den Import- und Exportdaten entkoppelt.

Die Hauptaufgabe der Trans-Container liegt in der Speicherung der Zuordnung. Diese wird in ihren Membervariablen abgelegt. Zudem ist es für die Bearbeitung der Zuordnungslisten nötig, Import- und Export-Container aus den Angaben der Trans-Container heraus zuordnen zu können. Daher gibt es entsprechende Ids, die über Methoden, die im Exporter definiert sind, auf verschiedene Weisen abgefragt werden können.

2.2.9 Serialisierung | Deserialisierung

Zum Zwecke der Serialisierung soll die Möglichkeit von .NET genutzt werden, ein Objekt automatisch als XML serialisieren zu lassen. Dabei werden alle „public“ Membervariablen und Properties gespeichert. Enthält das Objekt andere Objekte, so werden diese unter bestimmten Umständen ebenfalls serialisiert. Dies soll so ausgenutzt werden, dass eine Klasse „Schnittstelle“ die allgemeinen Schnittstelleninformationen wie Name, Quell- und Zielformat aufnimmt und eine Liste mit Trans-Containern enthält.

Alle für das Speicherformat in den beteiligten Klassen relevanten Variablen bekommen ein Property zugeordnet, das, wenn nötig, mit XML-Serialisierungsanweisungen bestückt ist. Diese dienen verschiedenen Funktionen. Anweisungen zum Ignorieren des Properties sind ebenso vorhanden wie Umbenennungen und die Möglichkeit, ein Property statt als Element als Attribut des übergeordneten Elements zu serialisieren. Zu beachten ist an dieser Stelle, dass Variablen, die nur für die Dauer dieses spezifischen Zuordnungsvorgangs Relevanz haben, beim Speichern nicht serialisiert werden. Es ist daher erforderlich, nach dem Lesen einer Schnittstelle einen Synchronisierungsvorgang zu initialisieren, um die Schnittstelle mit den vorhandenen Import- und Export-Containern zu synchronisieren.

Ein Vorteil dieses Verfahrens ist die sehr einfache und sichere Deserialisierung. Ein auf die oben genannte Weise gespeichertes Objekt lässt sich mit sehr geringem Aufwand wiederherstellen.

2.2.10 Synchronisation

Ist die Deserialisierung abgeschlossen und die Daten der Schnittstelle liegen im Exporter vor, so muss im SUI-Tool noch ein wichtiger Schritt vorgenommen werden. Die Zuordnungen und Formatinformationen sind zwar jetzt vorhanden, sie müssen aber noch mit den Export-Containern des Zielformates und, sobald diese ins System einlaufen, mit den Import-Containern „verdrahtet“ bzw. synchronisiert werden. Dies ist erforderlich, da der Benutzer im Bearbeitungsmodus die Zuordnungen ändern, löschen oder neu definieren kann und eine Überwachung der zugeordneten Pflichtfelder erforderlich ist.

2.2.11 Export

Der Export aus dem SUI-Tool erzeugt den oben erläuterten Dokumentyp *SinicDoc*. Der zweite Teil des Exports ist die Übertragung der Daten des SinicDoc-Dokuments in die SINIC-Datenbanken.

2.2.12 Serialisierung des Formats SincDoc

Um einen Systemimport durchführen zu können, muss zunächst das universelle XML-Format generiert werden. Dieser Vorgang ist im Sinne der gewählten Entkopplung der

einzelnen Komponenten lose verknüpft mit Import-Containern, die die Daten enthalten, und den Trans-Containern, die das Mapping beinhalten. Schwierigkeiten entstehen bei der Generierung der XML-Datei. Da das SINIC XML-Format mehrere Ebenen tief verschachtelte XML-Elemente enthalten kann , muss gewährleistet werden, dass die Serialisierungsroutine alle Elemente an den richtigen Stellen mit dem korrekten Elementbezeichner schließt. Ein Fehler führt zwangsläufig zu einer defekten XML-Datei, die beim nächsten Schritt im Prozessablauf, dem eigentlichen Systemimport, für einen Abbruch sorgen oder zu Inkonsistenzen führen kann. Um diesen Prozess einfacher und übersichtlicher zu gestalten, wird aus den gelesenen Daten eine Tabellenmatrix generiert, die die Elemente zunächst ordnet. Danach wird aus dieser Matrix die eigentliche XML-Datei erzeugt.

2.2.13 SINIC-Systemimport

SINIC-Systemimport bezeichnet den Vorgang der Datenübertragung vom dem durch die Software produzierten, universellen SINIC-Format in die SINIC-Datenbanken.

Dieser Import soll sowohl automatisiert, als auch manuell über eine Windows-Oberfläche durchgeführt werden können. Der manuelle Trigger ist in das SUI-Tool integriert. Was die Automatisierung betrifft, so ist hierzu ein Windows-Dienst angedacht, der einige Routinen aus dem Exporter verwendet und um zusätzliche Reporting-Funktionalitäten erweitert. Um seine Aufgabe zu verrichten ist das Vorhandensein von zwei Komponenten nötig; zum einen eine Datei, die die Quelldaten für den Systemimport enthält, zum anderen eine auf diese Daten passende, mit dem SUI-Tool generierte Schnittstellendatei. Der Systemimport ist dann ein zweistufiger Prozess. Zunächst wird aus diesen beiden Dateien das oben erläuterte SINIC-Format erzeugt. Danach wird es im zweiten Schritt gelesen und die Daten gemäß der erfassten Informationen zu Zielformat und Feldzuordnungen in die entsprechenden Tabellen der SINIC-Datenbanken geschrieben. Liegt das SINIC-Format bereits vor, so kann auf den ersten Schritt verzichtet werden und unmittelbar mit dem Systemimport begonnen werden. Um das Lesen der XML-Datei effizient durchzuführen, soll erneut XPath eingesetzt werden.

2.2.14 Reporting

Ein wichtiger Punkt sowohl für die Nachvollziehbarkeit der durch den Service durchgeführten Transaktionen als auch die Vorgänge während der Verwendung des SUI-Tools betrifft ein verlässliches Reporting. Das Reporting des Service erfolgt in zwei Schritten. Zum einen wird während des Transformations- und Importvorgangs ein Logfile im XML-Format erstellt. Dieses Logfile enthält eine Zusammenfassung der verschiedenen durchlaufenen Prozess-Schritte und einen Überblick über die Anzahl der importierten Daten. Zum anderen wird eine Email mit einer kurzen Zusammenfassung an die angegebene Supportemailadresse geschickt. Diese enthält dann nicht das komplette Logfile, sondern nur den Status des Programmablaufs mit eventuellen Fehlercodes und eine mengenmäßige Angabe über den Umfang der importierten Daten. Das SUI-Tool verzichtet auf eine Emailbenachrichtigung, da der User direkt mit der Software arbeitet.

Ein Problem bei der Realisierung eines Reportings ist die Gefahr eines inkonsistenten oder nebenläufigen Reportingmechanismus. Es muss gewährleistet sein, dass - ungeachtet von der Erzeugung neuer reportingbedürftiger Objekte - immer nur ein einziges Objekt mit der Erzeugung des Logfiles beauftragt ist. Wird dies nicht beherzigt, kann es dazu führen, dass mehrere unterschiedliche Logfiles für den gleichen Vorgang geschrieben werden oder Datenelemente im „richtigen“ Logfile fehlen. Um dieses Problem zu umgehen, wird das Singleton-Schema für die Realisierung des Loggings verwendet. Das *Singleton*-Schema ist folgendermaßen definiert:

„The Singleton Pattern ensures a class has only one instance and provides a global point of access to it. “[27]

Zu diesem Zweck hat die Klasse, die das Logging realisiert, die Kontrolle über die Erzeugung ihrer Instanzen. Auf diese Weise ist gewährleistet, dass alle beteiligten Objekte in dem gleichen Logfile erfasst werden. Eine Möglichkeit, dieses System auszuhebeln, wäre die Verwendung von Multithreading. Da dies aber nicht vorgesehen ist, besteht diesbezüglich keine Gefahr.

[27] HFDP, S. 177

2.2.15 Problemfelder

Nachdem das grundlegende Konzept nun dargestellt worden ist, sollen hier die wahrscheinlichen Problemfelder sowohl im konzeptionellen als auch im Realisierungsteil untersucht werden.

Verwendete Technologien

Um das Projekt erfolgreich umzusetzen, sind umfangreiche Kenntnisse in den Bereichen C#, .NET, dem SINIC-Framework und XML von Nöten.

Untersuchung der betroffenen Themen ist essentiell, kann Erfahrung jedoch nicht ersetzen. Eine fundierte Einarbeitung ist wichtig, und es ist davon auszugehen, dass mit fortschreitender Entwicklung neue Erkenntnisse gewonnen werden, die sich verschieden auswirken können. Im günstigen Fall können diese Erkenntnisse unmittelbar zur Verbesserung der Software führen. Im weniger günstigen, aber akzeptablen Fall kann es sein, dass ein bereits fertig gestellter und nicht ohne großen Aufwand zu ändernder Teil der Software bei einer Reevaluierung anders besser hätte umgesetzt werden können. Im ungünstigsten Fall läuft die gewählte Programmiermethode in eine Sackgasse und es muss ein Backtracking erfolgen, entweder mit einer konzeptionellen Abweichung einhergehend oder verbunden mit einem größeren Zeitverlust. Die größte Gefahr geht hier von den beiden verwendeten Frameworks aus. .NET ist sehr umfangreich und mächtig, jedoch auch äußerst ausführlich dokumentiert. Das SINIC-Framework ist ebenfall ein komplexes Gebilde, wobei einiges an Detailwissen zum effizienten Arbeiten weniger schriftlich dokumentiert, als in den Köpfen der SINIC-Mitarbeiter verankert ist.

Was C# betrifft, so sind einige Monate sicher nicht ausreichend, um Expertenlevel im Umgang mit dieser mächtigen Sprache zu erreichen. Ziel wird es sein, sich von vornherein auf die o.g. Problematik einzustellen und nicht zu strikt an einmal gewählten Realisierungsmethoden festzuhalten, wenn sich zu einem späteren Zeitpunkt andere Optionen auftun.

Offenes System

Wie aus der Aufgabenbeschreibung hervorgeht, handelt es sich bei dem zu entwickelnden Tool um ein recht offenes System, das eine Vielzahl von Möglichkeiten für weitere Entwicklungen bietet. Das Konzept erfasst die Kernfunktionalitäten. In Diskussionen mit den in der Entwicklung tätigen SINIC-Mitarbeitern traten jedoch noch eine Vielzahl an weiteren Möglichkeiten zu Tage, wie das System sich in Zukunft weiter entwickeln könnte bzw. welche Funktionalitäten denkbar oder interessant sein könnten. Dies ist nicht zuletzt von einer Analyse der Kundeninteressen und Bedürfnisse abhängig. Es ist daher nicht ausgeschlossen, dass während der Entwicklung Schwerpunktverschiebungen auftreten, die möglicherweise mit dem Grundkonzept inkompatibel sein könnten.

2.2.16 Zielsetzung und Limitationen

Verschiedene Ziele sollen mit der Software verwirklicht werden. Zum einen geht es darum, generell die Frage zu klären, ob eine Vereinheitlichung der innerhalb von CAQ-Systemen verwendeten Daten in einem universellen Format möglich ist. Dies soll am praktischen Beispiel zunächst als „Proof of Concept“ geprüft und umgesetzt werden. Laut Aussage des SINIC-Entwicklungsleiters, Herrn J. Schneider, ist eine solches vereinheitlichendes Format bisher in der Branche nicht vorhanden.

Das Ergebnis soll, falls es positiv ausfällt, Grundlage für die weiterführende Entwicklung der Software dienen. Im Idealfall sollen alle SINIC-Formate über das Universalformat abgebildet und vereinheitlicht weiterbearbeitet werden können.

Explizit ausgeklammert ist für die Software beim Handling der Im- und Exporte der Bereich Validierung. Einige SINIC-Systeme verwenden für die bezogenen Formate ein sehr umfangreiches Netz an Prüfroutinen, die die eingehenden Daten validieren. Eine Einbeziehung dieser umfangreichen Routinen, die zum Teil als Visual Basic vorliegen, würde den Rahmen der Diplomarbeit sprengen. Ähnliches gilt für eine komplette Integration aller existierenden SINIC-Formate und das Portfolio an angedachten Funktionen.

Bezieht man dies auf die Eingangs genannten Zielsetzungen, so ergibt sich daraus eine nach oben offene Aufgabenstellung, die sich in dem Ausspruch „wir fangen mal an, und schauen wie weit wir kommen“[28] zusammenfassen lässt. Ergänzende Elemente zu Teilbereichen können demnach jederzeit einfließen oder wieder verworfen werden, ohne allerdings die grundlegenden Zielsetzungen in Frage zu stellen.

3 Implementierung

3.1 Import

Der Import selbst ist ein mehrstufiger Prozess. Im ersten Schritt wird die Quelldatei geöffnet. Die Auswahl erfolgt über eine Dialogbox in der GUI. Entsprechend des in der GUI gewählten Zielformats, bzw. der in der Schnittstellendatei hinterlegten Information, wird die relevante Importer-Klasse erstellt. Diese extrahiert je nach Format auf unterschiedliche Weise die Daten und ordnet sie den Feldnamen im Quellformat zu. (Details in den jeweiligen Formaterläuterungen). Es werden zwar alle Daten gespeichert, aber für das Mapping in der GUI wird eine Liste zurückgegeben, in der jedes Feld nur jeweils einmal angezeigt wird. Zum Dateihandling selbst werden die I/O-Operationen des .NET-Frameworks verwendet.

3.1.1 Formate

Die grundlegenden Importformate sollen anhand der zugrunde liegenden Syntax, also dem Aufbau der zu Grunde liegenden Datei, unterschieden werden. Dieser Input wird vom User an die GUI übermittelt, die dann entsprechende Schritte für den Importvorgang einleitet.

3.1.2 Key:Value

Das eigentliche Matching erfolgt über einen regulären Ausdruck (Regex). .NET 2.0 hat die Behandlung regulärer Ausdrücke stark ausgebaut, so dass ein Matching komfortabel möglich ist.

[28] interne Kommunikation zu Projektbeginn; Herr Bursik, betreuender Entwickler

Codeauszug:

```
vergleich = "(^.*)" + kvSep + "(.*)";
regx = new Regex(vergleich);
// ...
m = regx.Match(s); //s enthält den zu durchsuchenden String
if (m.Success) //fall das Matching Erfolg hatte
     {
        string key = m.Groups[1].Value.ToString();
        string value = m.Groups[2].Value.ToString();
```

Zunächst wird der Matchstring zusammengesetzt. Die Variable *kvSep* enthält den Separator als String. Mit diesem String wird ein neues Regex-Objekt initialisiert. Die Variable m ist vom Typ *Match*. Ein *Match*-Objekt dient in .NET dazu, die Ergebnisse der *Regex.match* - Funktion, also des Vergleichs String zu Regular Expression, aufzunehmen. Über Match.Groups[Index] können die zuvor über runde Klammern gruppierten Ergebnisse entsprechend ihrer Reihenfolge abgefragt und zugewiesen werden.

Ein besonderes Augenmerk wird auf die Zeilen gelegt, die nicht durch die Regex gematcht werden. Oft sind in den *Key:Value*-Textdateien Datensatztrenner enthalten, die die zueinander gehörenden *Key:Value*-Paare auf Datensatzebene voneinander trennen. Alle Zeilen, die nicht in das Raster der Regex passen, werden in einer anderen Liste gespeichert. Am Ende des Parsings werden die enthaltenen Strings untereinander verglichen und der am häufigsten vorkommende Kandidat in einer Variable als der wahrscheinlichste Datensatztrenner gespeichert. Momentan wird dieser Wert noch nicht verwendet; er wird aber bei einigen der selteneren SINIC-Formate zum Tragen kommen.

3.1.3 Fixed Record Length

Enthalten, wie bereits im Konzept erläutert, die Felder an bestimmten, fix definierten Positionen innerhalb eines Strings.

Beispiel:

```
1QA0001I   PP5121010   PAL16L8ACN REVI    Prog Array Logik    4009700    1
71205
```

Das Parsen ist in diesem Fall relativ einfach. Die Datei wird zeilenweise ausgelesen und der String entsprechend der vorliegenden Formatinformation zerhackt. Dies ist mittels Stringoperationen relativ unproblematisch möglich. Die Formatinformation wird entweder

aus der GUI übergeben oder aus der Schnittstellendatei ausgelesen. Zu beachten ist an dieser Stelle, dass ein sauberes Fehlerhandling für den Fall implementiert werden muss, dass die Datei kurze Zeilen enthält, die nicht die volle Länge der Formatbeschreibung aufweisen. Ein in der Praxis nicht unüblicher Sonderfall ist die Situation, dass der User zwar das eigene Format „im Kopf“ hat, aber die exakte Formatdefinition momentan nicht vorliegt. Um diese Situation ebenfalls abzudecken ist ein Algorithmus implementiert, der es im Zusammenspiel mit der GUI ermöglicht, unter Zuhilfenahme einer Beispieldatei einen String Schritt für Schritt zu zerlegen und die Teilstücke mit den entsprechenden Feldbezeichnern zu versehen. Nachdem die Zuordnung erfolgt ist, kann sie mit der Beispieldatei sofort getestet werden, um das Ergebnis unmittelbar zu überprüfen.

3.1.4 Comma Separated Values (CSV)

Dieses Format tritt - wie bereits im Konzeptteil erwähnt - in verschiedenen Varianten mit und ohne Überschriftenzeile auf:

Beispiel:

```
Artikelnummer,LieferantenID,Status, Index
103AD2,101,1,5
7700W3,89,1,14
7701W3,89,1,15
```

Die CSV-Importmethode parst die Datei zeilenweise unter Verwendung eines wählbaren Separators. Analog zu den anderen Textformaten ist der Separator über GUI oder Schnittstellendatei wählbar. Wie in den Beispielen zuvor finden reguläre Ausdrücke zum Parsing Verwendung.

3.1.5 Excel (.xls)

Zur Implementierung des Excel-Imports finden die Microsoft PIAs Verwendung. Laut Dokumentation soll es durch Einbinden der Bibliothek *Microsoft.Office.Interop.Excel* möglich sein, mit der Excel-Datei bzw. dem darin enthaltenen Workbook[29] zu arbeiten. In der Praxis treten zur Laufzeit allerdings Fehler beim Öffnen des Workbooks auf. Der Grund hierfür ist in einem Fehler in der Library zu suchen. Der Fehler entsteht durch eine

[29] Dt. “Arbeitsmappe”. Eine Exceldatei enthält eine Arbeitsmappe mit 1-n Tabellenblättern.

Inkonsistenz bezüglich der vom Benutzer installierten Windows-Sprachversion. Die Library erwartet fälschlicherweise die Einstellung US/Englisch (Parameter ‚en-US'), ohne lokal abweichende Betriebssysteminstallationen zu berücksichtigen. Um dieses Problem zu umgehen, muss zunächst das „Current Culture"-Property des verwendeten System-Threads auf „en-US" umgestellt werden.[30] Danach kann mit dem Workbook, wie in der Dokumentation angegeben, gearbeitet werden. Nach Bearbeitung des Workbooks muss der Thread allerdings wieder auf die korrekte Ländereinstellung zurückgesetzt werden, um Inkonsistenzen und schwer zu findende Fehler zu vermeiden. Im Code stellt sich die Prozedur folgendermaßen dar:

```
//Umstellung
System.Globalization.CultureInfo CurrentCI =
System.Threading.Thread.CurrentThread.CurrentCulture;
System.Threading.Thread.CurrentThread.CurrentCulture = new
System.Globalization.CultureInfo("en-US");
// Bearbeitung des Workbooks …..
// …
//Zurücksetzen
System.Threading.Thread.CurrentThread.CurrentCulture = CurrentCI;
```

Der eigentliche Import läuft in folgenden Schritten ab:

1. Zunächst wird eine Instanz von Excel gestartet. Um die Performance zu verbessern wird über Parameter angegeben, dass die Excel-GUI nicht mitgestartet wird.

2. Das Workbook wird geöffnet und der Name des Tabellenblatts ausgelesen. Dieses wird zur späteren Verwendung gespeichert.

3. Die Excelinstanz wird geschlossen und das Workbook Objekt wieder freigegeben.

4. Ein spezieller Connection-String wird erstellt, der Datenbankzugriffe auf die Tabelle des Workbooks erlaubt. Verwendet wird der Microsoft Jet OLEDB -Treiber mit dem Workbook als Datenquelle.

```
string sConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;" +"Data
Source=" + filename + ";" +   "Extended Properties=Excel 8.0;";
```

[30] MSDN please

5. Die Verbindung wird erstellt und über SQL-Befehle der Inhalt der Tabelle abgefragt. Das Ergebnis der Abfrage wird von einem *DataAdapter* in ein *Dataset* gefüllt. Die Ergebnisse der Abfrage stehen nun in der ersten Tabelle des Datasets.

6. In zwei Schleifen wird über die Tabelle iteriert und alle Import-Container per Zuordnung der Zeileninhalte zu den Spaltenüberschriften erfasst.

7. Die Datenbank-Verbindung wird geschlossen und die Liste Import-Container an die GUI zurückgegeben.

Sieht man von der eingangs erwähnten Ländereinstellungsproblematik ab, so ist die .NET-Plattform für den Umgang mit Excel gut ausgerüstet und stellt einen komfortablen Zugriff auf das Format zur Verfügung. Ein Problem, welches bisher nicht angegangen worden ist, aber nicht unerwähnt bleiben soll, betrifft variierende Excel-Versionen. Da nicht gewährleistet ist, dass der Kunde die gleiche Excel-Version besitzt wie der Entwickler, kann es zu Problemen kommen. Um dieses Problem auszuschalten, kann man auf „Late Binding“ setzen, ein Verfahren, das grundlegende Excel-Funktionen versionsübergreifend zur Laufzeit zugänglich macht.

3.1.6 Microsoft Access (.mdb)

Zunächst wird die Access (.mdb) -Datei analog zur Exceldatei mittel des Microsoft Jet.OleDB-Treibers konnektiert. Das erste Problem, dass sich nun stellt, ist, dass unklar ist, wie viele verschiedene Datenbanktabellen in der Datei enthalten sind und wie diese heißen. Zwar ist es in den allermeisten Fällen so, dass die Dateien der SINIC-Kunden nur jeweils eine Tabelle enthalten, trotzdem muss der Möglichkeit Rechnung getragen werden, dass Ausnahmen auftreten können. .NET stellt für dieses Problem die Funktion *GetOleDBSchemaTable* zur Verfügung.

Code

```
public DataTable GetOleDbSchemaTable (Guid schema,Object[] restrictions)
```

Diese Funktion liefert eine *DataTable* zurück, die umfangreiche Informationen über die Datenbank beinhaltet. Der Funktionsaufruf selbst beinhaltet unter anderem auch die Möglichkeit, Filter auf bestimmte Datenbankeigenschaften zu setzen. So ist es z.B. möglich, nur die Tabellennamen herauszufiltern.

Beispiel:

```
//Table Schema holen, Filter auf Tabellennamen setzen
schemaTable = cn.GetOleDbSchemaTable(
OleDbSchemaGuid.Tables, new Object[] { null, null, null,"TABLE"});
```

Nun wird über die DataTable iteriert und eine Verbindung zu den enthaltenen Tabellen hergestellt. Per SELECT Statement werden die Felder der Tabellen ausgelesen und als Nodes in eine *SinicTreeView*-Control eingehängt. Ein eigentlicher Import findet an dieser Stelle noch nicht statt, die Aufbereitung der Baumstruktur und dadurch die Ermöglichung von Feldselektion und Zuordnung stellt allerdings die nötige Vorstufe für eine weitere Bearbeitung dar.

3.1.7 XML

Die vorliegende Implementierung verwendet zum Lesen und Bearbeiten der XML-Struktur die *XmlDocument*-Klasse des .NET-Frameworks.

Zunächst wird das *XmlDocument*-Objekt mit der vom User angegebenen XML-Datei initialisiert. Die XML-Struktur ist damit in dem Objekt abgebildet. Dann wird das Ursprungselement des Dokuments gelesen. Um per *Xpath* auf Elemente zugreifen zu können, müssen vorhandene XML-Namespaces dem *XMLNamespaceManager* des .NET-Frameworks hinzugefügt werden. Eine Besonderheit, die dabei zu beachten ist, ist die Tatsache, dass auch Elemente ohne Präfix im Namespace Manager ein solches erhalten müssen[31]. Dieser Faktor sorgte bei der Implementierung zunächst für einige Probleme. Es muss ein beliebiges Präfix für die „namenlosen“ Elemente gewählt werden und diese in die Instanz des Namespace-Managers eingetragen werden. Eine Missachtung dieser Vorgehensweise führt dazu, dass ansonsten tadellos funktionierende Xpath-Ausdrücke plötzlich keine Ergebnisse mehr zurückliefern. Als nächster Schritt wird

[31] MSDN, HOW TO: Specify Namespaces When You Use an XmlDocument to Execute XPath Queries in Visual C# .NET

XMLDocument.SelectNodes aufgerufen. Ihr wird ein Xpath-Ausdruck übergeben und ein Verweis auf den Namespace-Manager. Als Ergebnis wird eine Nodelist zurückgegeben, die alle zur Xpath-Expression passenden Knoten (Nodes) der XML-Struktur zurückliefert.

Code:

```
XmlDocument dom   = new XmlDocument();//Neues XMLDoc
dom.Load(strXMLFile); // Load XML file.
//Instantiate an XmlNamespaceManager object.
System.Xml.XmlNamespaceManager xmlnsManager = new
System.Xml.XmlNamespaceManager(dom.NameTable);
//Add the namespaces to the XmlNamespaceManager.
xmlnsManager.AddNamespace("ol", "QML");
//...
XmlNodeList oNodes = dom.SelectNodes(strXPath,xmlnsManager);
XmlNode     xNode  = oNodes.Item(0).ParentNode;
AddNode(ref xNode, ref tNode,tRootNode); //Start recursive Node walking
```

Diese Nodes ähneln den Nodes, die in den *TreeView*-Controls der Windows Forms Verwendung finden und können daher direkt in eine *TreeView* eingehängt werden. Als letzter Schritt bleibt, in einer rekursiven Funktion die Nodelist zu durchlaufen und die XML-Nodes an richtiger Stelle in die *TreeView* einzuhängen. Um diese potenziell lange dauernde Rekursion abzukürzen, kann ein Limit der darzustellenden Nodes eingegeben werden. Das Ergebnis ist die Darstellung des XML-Dokuments als Baumstruktur in der GUI. Dort können dann Knoten zwecks Zuordnung ausgewählt werden. Eine Zuordnung ist dann relativ einfach möglich, da aus den ausgewählten Knoten entsprechende Xpath-Ausdrücke generiert werden können, die gezielt Elemente aus dem XML-Dokument herausziehen können.
Die Grundlage zur Verwendung in einem späteren Prozess ist somit gegeben.

3.2 Zuordnung

3.2.1 Zielformate

Die Daten, die die GUI zur Darstellung und der Exporter zur Verwaltung braucht stellt die Funktion *readXMLFormat* zur Verfügung.

Code

```
public List<blExpCon> readXMLFormat(int Zielformat)
```

Als Technik zum Auslesen dient ein XmlReader (Reader). Zunächst werden die Einstellungen des Readers auf das Ignorieren von Whitespace und Kommentaren gesetzt. Dann wird eine neue Instanz unter Angabe der Settings und des Dateinamens der relevanten XmlDBSchema-Datei angelegt. Zum Parsen sind einige Angaben nötig, die nicht unmittelbar aus diesen Dateien hervorgehen. Diese werden mittels der übergebenen Variable *Zielformat* durch eine neue Instanz der *blTableRelationConstants*-Klasse bereitgestellt.

In einer while – Schleife liest der Reader linear vorwärts durch das XML-Dokument. Ein switch–case Block übernimmt die Steuerung zur Analyse und dem Verteilen der Daten anhand der gelesenen Node-Types.

Es werden zunächst drei Fälle unterschieden. Zwei davon, nämlich die Node-Types *Text* und *EndElement* werden nur der Protokollierung wegen erfasst. Sind die Nodes von diesen Typen, enthalten sie gemäß der Struktur der XmlDbSchemas keine relevanten Daten. Der interessante Fall ist der Node-Type *Element* und hierbei speziell die *DBColumn*-Elemente, da sie die eigentlichen Spaltendefinitionen der Datenbanktabellen enthalten.

Zunächst wird geprüft, ob das *DBColumn*-Element Attribute enthält. Falls ja, werden diese der Reihe nach gelesen. Elemente werden ohne Bearbeitung verworfen, sofern sich der Wert des ersten Attributs des Elements *d2p1:TableName* nicht mit dem für dieses Format erforderlichen Tabellennamen deckt. Ist das korrekte Element auf diese Weise identifiziert, werden die Attribute *d2p1:colName*, *d2p1:description* und *d2p1:isKey* ausgelesen. *colName* bestimmt den Feldnamen, *description* enthält Anzeigeinformationen für die GUI und *isKey* gibt an, ob diese Spalte für die Datenbank als ein Schlüssel fungiert. Geht aus der Formatbeschreibung hervor, dass es sich bei bestimmten Spalten um reservierte Felder handelt, oder wenn Felder aus anderen Gründen nicht in der GUI auftauchen sollen, enthält die *description* den Wert *noShow*. Ist *isKey* „true“, wird zum einen der Spaltenname einer Liste von Keys hinzugefügt, die der jeweiligen Datenbankebene (1-3) entspricht, zum andern wird eine Variable gesetzt, die dafür sorgt, dass das Feld in der GUI besonders gekennzeichnet wird.

Zu guter Letzt werden die Daten dieses *DBColumn-Elements* zur Erzeugung eines Export-Containers verwendet, der zur Liste *exportFelder* hinzugefügt wird. Die Funktion endet mit der Rückgabe der Liste *exportFelder* an die GUI.

Das Auslesen der Daten über einen *XmlReader* mag zunächst nicht so elegant erscheinen wie die individuelle Knotenselektion über Xpath-Ausdrücke, hat aber den Vorteil, dass ein XmlReader sehr schnell liest und mit geringem Overhead auskommt[32].

3.2.2 Mapping

Das Mapping wird durch einen Klick auf den „Zuordnen"-Button der GUI gestartet. Zunächst werden die ausgewählten Quell- und Zielfelder ermittelt. Dies geschieht über einen Abgleich der momentan aktiven Zelle im DataGridView bzw. der ausgewählten Node im TreeView mit den Import- und Export-Containern. Dann werden die ermittelten Container zur Erstellung eines neuen Trans-Containers verwendet.

Code

```
trans = new blTrans(import[importEngine.currentICID],
export[selectedExpID]);
```

Die Klasse *blTrans* entscheidet in ihrem Konstruktor, welche Elemente der übergebenen Container sie verwendet.

Der gewählte Export-Container wird zudem auf „assigned" gesetzt, um eine Pflichtfeldprüfung möglich zu machen. Als letzter Schritt erfolgt eine Prüfung, ob das gewählte Importfeld vorher bereits zugewiesen wurde. Ist dies der Fall, wird der User darauf hingewiesen und kann entscheiden, ob das Feld neu zugewiesen werden soll. Entsprechend dieser Entscheidung wird der neue Trans-Container gegen den alten ausgetauscht, oder es geschieht nichts. Im anderen Falle wird der Trans-Container den Zuordnungen hinzugefügt.

[32] MSDN, Improving XML Performance

3.2.3 Transformations-Container

Die Objekte der Klasse *blTrans* bilden das Bindeglied zwischen Import- und Export-Containern. Sie implementieren das Interface *ITransformable* und speichern das Mapping des Users. Bei der Serialisierung stellen sie das Kernelement dar.

3.2.4 Serialisierung

Zur Serialisierung der neuen Schnittstelle wird die Technik des XmlSerializers verwendet.

Code

```
XmlSerializer mySerializer = new XmlSerializer(typeof(blSchnittstelle));
```

Initialisiert wird der *XmlSerializer* (Serializer) mit der zu speichernden Klasse. Die Klasse *blSchnittstelle* dient diesem Zweck. Wie im Konzept beschrieben hat sie zwei Aufgaben. Zum einen nimmt sie die Header-Informationen für die neue Schnittstelle auf, zum anderen bekommt sie von dem Exporter-Objekt die Liste mit Trans-Containern übergeben, die die Zuordnungen enthält.

Nun ist nur noch ein Streamwriter erforderlich, den der Serializer für den Speichervorgang verwenden kann. Ein Streamwriter erstellt und öffnet eine Datei zum Schreiben und erwartet einen Stream der geschrieben werden kann. Dieser Streamwriter wird zusammen mit der zu speichernden Objektinstanz dem Serializer als Parameter übergeben, um die Serialisierung zu starten. Das Objekt muss dabei zwangsweise vom Typ der Klasse sein, die bei der Initialisierung des Serializers angegeben worden ist.

Code

```
blSchnittstelle Schnittstelle = new blSchnittstelle();
Schnittstelle.setHeader(mFormatName, mTyp, mDesc, mStatus);
Schnittstelle.addFelder(mTrans);
StreamWriter myWriter = new StreamWriter(filename);
mySerializer.Serialize(myWriter, Schnittstelle);
```

Serialisiert werden alle Public-Membervariablen und Properties, die im Default-Fall in ein gleichnamiges XML-Element umgewandelt werden. Dieser Vorgang kann durch die

Angabe von Attributen beeinflusst werden.[33] Die für die Speicherung relevanten Properties und Attribute sollen nun kurz beschrieben und der entstehenden Ausgabe gegenübergestellt werden.

Code

```
[XmlRootAttribute("SINICSchnittstelle", Namespace =
"http://www.sinic.de/SUI/Schnittstelle",
IsNullable = false)]
```

Das XmlRootAttribute wird direkt vor die Klassendeklaration gesetzt. Es bestimmt das Basiselement des XML-Files und den dazugehörigen Namespace.

Output

```
<?xml version="1.0" encoding="utf-8"?>
<SINICSchnittstelle xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="http://www.sinic.de/SUI/Schnittstelle">
```

Code

```
        [XmlElement("ZielDokumentTyp")]
        public long Typ {... }

        [XmlElement("QuellDokumentTyp")]
        public long Quellformat{... }

        [XmlElement("Separator")]
        public string FWTrenner{... }

        [XmlElement("Zuordnungsname")]
        public string FormatName{... }

        [XmlElement("Description")]
        public string Desc{... }

        [XmlElement("FreigabeStatus")]
        public bool Status{... }
```

Diese Properties werden serialisiert, da sie Public sind. Die Steuerattribute [XmlElement] bewirken, dass sie bei der Serialisierung die angegebenen XML-Elementnamen erhalten.

[33] MSDN, Steuern der XML-Serialisierung mit Attributen

Output

```
  <ZielDokumentTyp>0</ZielDokumentTyp>
  <QuellDokumentTyp>0</QuellDokumentTyp>
  <Zuordnungsname>Artikelimport</Zuordnungsname>
  <Description>aus KV</Description>
  <FreigabeStatus>false</FreigabeStatus>
```

Code

```
[XmlArrayItem("Zuordnung")]
public blTrans[] Zuordnungen {...}
```

Ein weiteres Property. Da es sich um ein Array handelt, wird es vom Serializer so behandelt, dass alle enthaltenen Objekte vom Typ *blTrans* einzeln geparst werden und die Objekte ihrerseits auf die gleiche Art nach Properties und Public-Variablen durchsucht werden. Das Attribut [XmlArrayItem] gibt an, unter welchem Elementnamen die einzelnen Array-Items serialisiert werden sollen. Eine Untersuchung der *blTrans*-Klasse ergibt folgende relevante Properties:

Code

```
public int Level  {  …   }
[XmlIgnore]
public int ExpConID {    …   }
[XmlIgnore]
public int TransID{  …   }
[XmlElement(ElementName = "QuellFeld")]
public string ImpFeld {  …   }
public string ZielFeld{  …   }
[XmlIgnore]
public string Wert {  …   }
```

XmlIgnore bewirkt, dass das folgende Property ignoriert wird. *XmlElement* legt einen vom Namen des Properties abweichenden Elementnamen fest. Dies ergibt für das *blTrans*-Array folgendes Ergebnis:

Output

```
<Zuordnungen>
    <Zuordnung>
      <Level>1</Level>
      <QuellFeld>K001</QuellFeld>
      <ZielFeld>PLANID</ZielFeld>
    </Zuordnung>
    ...hier folgen die weiteren Zuordnungs-Tags....
</Zuordnungen>
```

Auf diese Weise werden alle für die Schnittstelle bedeutsamen Werte auf übersichtliche Weise in der Zieldatei gespeichert.[34]

3.2.5 Deserialisierung

Die gespeicherten Daten müssen natürlich auch wieder ausgelesen werden können. Sowohl das SUI-Tool als auch der dazugehörige Dienst müssen dazu in der Lage sein. Die entsprechende Routine ist in der *Exporter*-Klasse implementiert. Das verwendete Verfahren nutzt ebenfalls die *XmlSerializer-Klasse*. Dies bietet sich an, da ein mit *XmlSerializer.serialize* gespeichertes Objekt mittels *XmlSerializer.deserialize* wiederhergestellt werden kann. Die grundlegende Vorgehensweise ist dabei identisch zum Serialisieren. Benötigt wird ein *XmlSerialize*- Objekt, das mit dem Typ *blSchnittstelle* initialisiert wird. Dann wird ein lesender Stream auf die Schnittstellendatei geöffnet und die *XmlSerializer.deserialize*- Methode aufgerufen, deren Rückgabeobjekt auf *blSchnittstelle* gecastet und einem entsprechenden Objekt zugewiesen wird.

Code

```
XmlSerializer serializer = new XmlSerializer(typeof(blSchnittstelle));
Stream sreader = new FileStream(sFilename, FileMode.Open);
blSchnittstelle mySchnittstelle;
mySchnittstelle = (blSchnittstelle)serializer.Deserialize(sreader);
sreader.Close();
```

Um eine hohe Sicherheit bei der Deserialisierung zu gewährleisten, ist für den Fall eines Fehlers bei diesem Vorgang ein zweites Verfahren als Backup implementiert. Dabei wird die Schnittstellendatei noch einmal neu mittels eines *XmlReader* gelesen und die relevanten Elemente „von Hand“ per if-then Abfragen in das *blSchnittstelle*- Objekt eingelesen. Auf diese Weise können auch defekte Dateien gelesen werden, solange die Kerninformationen innerhalb der Zuordnungs-Elemente intakt sind. In der Log-Datei wird ein entsprechender Eintrag geschrieben, da ein Versagen der *deserialize*-Methode einen schwerwiegenden Fehler in der geschriebenen Schnittstellendatei bzw. die komplette Abwesenheit derselben indizieren würde. Es ist denkbar, dieses Backupverfahren in verschiedene Richtungen auszubauen: eigenes Fehlermenü, Hilfetipps etc.

[34] Siehe Anhang für eine komplette Schnittstellendatei

3.2.6 Synchronisation

In einem dreistufigen Prozess werden folgende Dinge erledigt:

Zunächst wird ein Abgleich zwischen dem *Feld*-Property von Trans-Containern und vorhandenen Import-Containern durchgeführt. Wird der passende Import-Container gefunden, wird die TransId des Trans-Containers mit der Id des Import-Containers synchronisiert. Gleichermaßen wird mit den Export-Containern verfahren, wobei die ExpConID des Trans-Containers mit der ID des Export-Containers gleichgesetzt wird. Dabei werden identische Feldbezeichungen in unterschiedlichen Tabellen entsprechend berücksichtigt.

Wird ein übereinstimmender Container nicht gefunden, so wird in einem dritten Schritt der verwaiste Trans-Container in eine Liste aufgenommen und nach Beendigung der Überprüfung aller Container aus der Zuordnungsliste gelöscht. Programmatisch ist dies eine der interessanteren Konstruktionen, welche die Möglichkeiten der .NET-*List* benutzt, um einen eleganten und kompakten Code zu erhalten.

Code

```
private void synchronizeTransIDs()
        {
            List<int> notFound = new List<int>();
            int nCounter = 0;
            foreach (blTrans blT in transList)
            {
            try
                {
            //Sync Import
blT.TransID = import.Find(delegate(blIImportable blic) { return blic.Feld
== blT.ImpFeld; }).vFeld;
                  //Sync Export
                    blT.ExpConID = export.Find(delegate(blIExportable
blex) { return blex.Feld == blT.ZielFeld && blex.Level == blT.Level;
}).ID;
                    exportEngine.setAssignedById(blT.ExpConID, true);
                }
      // weiterer Code…
            }
```

Aus der Nähe betrachtet:

Code

```
blT.TransID = import.Find(delegate(blIImportable blic) { return blic.Feld
== blT.ImpFeld; }).vFeld;
```

Hier geschieht folgendes:

Die *Find*-Methode der *List< blIImportable> import* gibt ein Objekt zurück, auf das bestimmte Bedingungen zutreffen. *Delegate(blIImportable blic)* dient dazu, ein temporäres Objekt zur Definition der Bedingung zur Verfügung zu stellen. Die eigentliche Bedingung ist dann *blic.Feld==blT.ImpFeld*, welche die Feldnamen von Import-Container (*blic*) und Trans-Container (*blT*) vergleicht. Wird diese Bedingung erfüllt, so wird das entsprechende Objekt zurückgegeben. Von dem gefundenen Objekt wird das *vFeld*-Property ausgelesen und an *blT.TransID* zugewiesen. Wird kein Objekt gefunden, so löst die Zuweisung eine *Exception* aus, die mit *Catch* gefangen wird. Dort wird die Indexnummer des auslösenden Containers gespeichert, um ihn später mit der Listenmethode *RemoveAt(Index)* aus der Liste löschen zu können.

3.3 Export

Für den Export sind zwei Stufen nötig. Die Erstellung des SINIC-Formats und den Systemimport aus diesem Format in die Datenbanken.

3.3.1 Format

Als erste Stufe des zu tätigenden Systemimports wird zunächst das SINIC-Format generiert. Der grundsätzliche Aufbau ist im Konzept dargelegt worden. Als Ergebnis, am einfachen Beispiel demonstriert, soll folgendes XML stehen:

Code

```
<?xml version="1.0" encoding="utf-8"?>
<SinicDoc xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="http://www.sinic.de/SUI/Export">
    <Head>
        <DataType>ArtikelImport</DataType>
        <Version>AI-100</Version>
        <DB id="" dbType="Access">QSISTM</DB>
    </Head>
    <Data>
    <SinicDataSet tableID="QSART" pKey0="ARTNR" pKey1="INDEX">
        <Value colID="ARTNR">AIX 5</Value>
        <Value colID="ARTBEZ">AI-Xen.Zub.5-69</Value>
        <Value colID="ZINDEX">0</Value>
        <SinicDataSubSet tableID="QSARTLIF" mastercol="ARTNR"
refcol="ARTNR">
            <Value colID="ARTNR">AIX 5</Value>
            <Value colID="LIEFID">001</Value>
            <Value colID="STATUS1">0</Value>
        </SinicDataSubSet>
    </SinicDataSet>
    <SinicDataSet tableID="QSART" pKey0="ARTNR" pKey1="INDEX">
        <Value colID="ARTNR">AIX 8</Value>
        <Value colID="ARTBEZ">AI-Xenotype Mk. 8</Value>
        <Value colID="ZINDEX">100</Value>
        <SinicDataSubSet tableID="QSARTLIF" mastercol="ARTNR"
refcol="ARTNR">
            <Value colID="ARTNR">AIX 8</Value>
            <Value colID="LIEFID">002</Value>
            <Value colID="STATUS1">1</Value>
        </SinicDataSubSet>
    </SinicDataSet>
    </Data>
</SinicDoc>
```

3.3.2 Serialisierung

Die Vorgehensweise zur Umformung der vorliegenden Daten in das SINIC-Format ist wie folgt:

Zunächst wird durch alle Import-Container iteriert. Jeder Container wird mit seiner *Feld*-Eigenschaft gegen die *ImpFeld*-Eigenschaft der Trans-Container verglichen. Liegt eine Übereinstimmung vor, so wird die *ZielFeld*-Eigenschaft des Trans-Containers gegen die Indikatoren der TRC-Klasse verglichen. Das Ergebnis dieses Vergleichs wird zusammen mit einigen anderen Werten in eine Tabelle eingetragen. Der Aufbau dieser Tabelle (*Exportmatrix)* ist wie folgt:

Export-Typ „PRIME“, „SEC“, „TERT“, „NORM“	Zähler Level 1	Zähler Level 2	Zähler Level 3	Zielfeld	Wert
Daten	Daten	Daten	Daten	Daten	Daten
Daten	Daten	Daten	Daten	Daten	Daten
...					

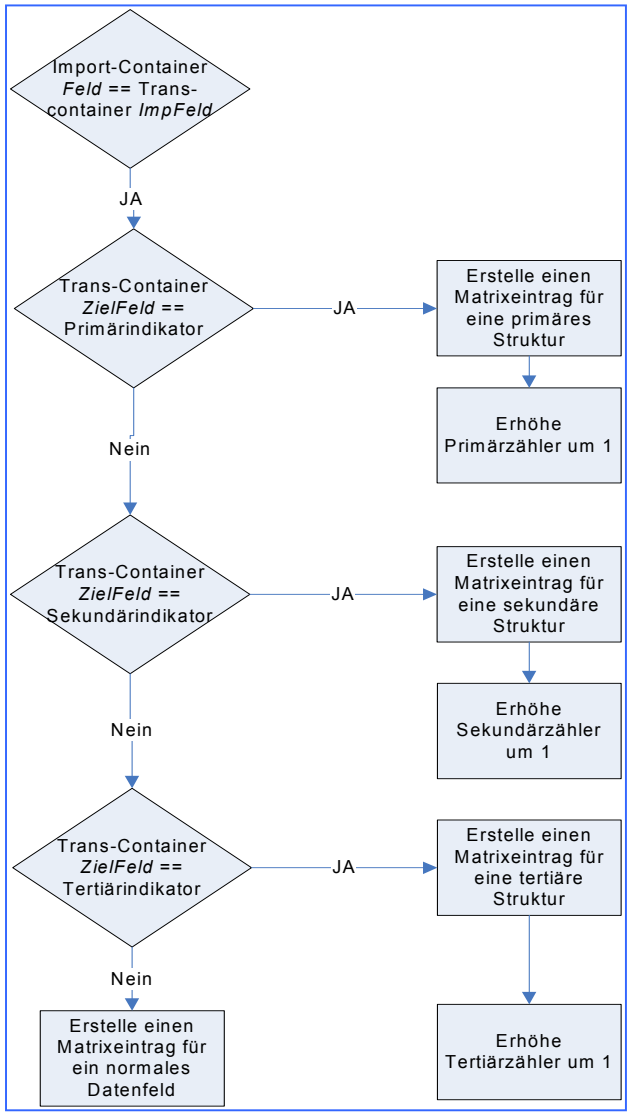
Je nachdem, welche Ergebnisse die o.g. Abfragen ergeben, werden die Import-Container der Reihe nach in diese Matrix eingetragen. Der Algorithmus ist wie folgt:

1. Ist die Feld-Eigenschaft des Import-Containers in keiner ImpFeld-Eigenschaft der Trans-Container, wird nichts getan.

2. Ansonsten:

   a) Entspricht die Zielfeld-Eigenschaft des passenden Trans-Containers dem TRC-Primärindikator, dann erstelle einen neuen Matrixeintrag der Kategorie "PRIME" und erhöhe PrimeCounter um 1.

   b) Entspricht die Zielfeld-Eigenschaft des passenden Trans-Containers dem TRC-Sekundärindikator, dann erstelle einen neuen Matrixeintrag der Kategorie „SEC“ und erhöhe SecCounter um 1.



Algorithmus Exportmatrix

c) Entspricht die Zielfeld-Eigenschaft des passenden Trans-Containers dem TRC-Tertiärindikator, dann
erstelle einen neuen Matrixeintrag der Kategorie „TERT“ und erhöhe TertCounter um 1.

d) Ansonsten erstelle einen neuen Matrixeintrag der Kategorie "NORM“.

Somit ergibt sich eine Matrix, die über ihre Indizes in den Spalten zwei bis vier jedem Element eine Position zuweist. Daten die als „NORM“ gekennzeichnet in die Matrix eingehen, haben keinen eigenen Zähler, da ihre Position untereinander keine Relevanz hat. Mit den Informationen aus dieser Matrix wird nun das eigentliche Schreiben der Exportdatei in Angriff genommen.

Zunächst wird eine neue Instanz der Klasse *WFAlpha* initialisiert. Diese regelt die eigentliche Serialisierung. Im Laufe der Initialisierung richtet die Klasse eine Instanz der .NET-Klasse *XmlWriter* ein. Unter Angabe des Dateinamens und der *XmlWriter*-Settings wird das Schreiben der Exportdatei begonnen und das Startelement *SinicDoc* geschrieben. Danach ist die Initialisierung beendet.

Nun werden Prime-, Sec- und Tert-Counter zurückgesetzt. Ein Flag für ein geschriebenes Subset, also ein Element der Ebene zwei oder drei, wird angelegt und auf *false* gesetzt.

Der Schreibvorgang des Dokuments läuft nun in folgenden Schritten ab:

1. Der Dokument-Header wird geschrieben. Dieser enthält das Element Head mit allen untergeordneten Elementen.

2. Nacheinander werden die Elemente der Matrix durchlaufen.

3. Zuerst erfolgt die Prüfung auf den Primärindikator(„PRIME“). Ist diese positiv, wird geprüft. ob bereits ein Subset geschrieben worden ist. Falls ja, wird dieses Element geschlossen. Danach wird das Prime-Element geschrieben. Auf Datenebene wird in *WfAlpha* ein neues *SinicDataSet* begonnen, die TabellenID wird ergänzt und es werden alle nötigen Keys im Format *pKey# =“ Name“* als Attribute

hinzugefügt. *XmlWriter.WriteAttributeString* erzeugt aus zwei übergebenen Stringvariablen einen Attribut-String des Formats *(Variable a)="(Variable B)"*.

Code:

```
writer.WriteAttributeString("pKey"    +    z.ToString(),    mPKeys[z]);
```

Danach wird das eigentlich Datenelement als "normales" Element mittels der Methode *addData(string Field, string Value)* innerhalb von *SinicDataSet* hinzugefügt. Innerhalb dieser Methode werden zusätzlich die Werte der Keys zur späteren Verwendung gespeichert.

4. Falls die Prime-Prüfung erfolglos war, wird nun als nächstes auf den Sekundärindikator geprüft. Falls erfolgreich, wird analog zu Punkt 3 zunächst auf ein bereits geschriebenes Subset getestet und dieses beendet. Dann wird SubsetWritten auf true gesetzt und der SubsetCounter um eins erhöht. Auf Datenebene wird ein *SinicDataSubSet* geschrieben. Attribute für den Tabellennamen sowie die für die Db-Tabelle zweiter Ebene geltenden Keys werden ergänzt. Dabei werden die Werte des zuletzt geschriebenen Datensatz-Primarys, die zu diesem Zweck gespeichert wurden (siehe Punkt 3), den Attributen zugeordnet. Die Verbindung zwischen erster und zweiter Datenbank-Ebene ist somit hergestellt. Danach wird das Datenelement analog zu Punkt drei mittels der Methode *addData* geschrieben.

5. Analog zur Prüfung auf den Sekundärindikator verläuft die Prüfung auf Ebene 3. Der anschliessende Prozess unterscheidet sich ebenfalls nur minimal.

6. Zuletzt wird der Fall abgedeckt, dass es sich um ein „normales" Datenelement ohne Indikatoreigenschaft handelt. In diesem Falle wird auf Datenebene direkt die Methode *addData* aufgerufen. Der Bezeichner für die Datenbankspalte wird innerhalb des Elements *Value* als Attribut *colID* ergänzt; der eigentliche Wert folgt dann im Element selbst.

7. Ist die Matrix abgearbeitet, werden noch offene XML-Tags geschlossen und die Methode *finishXML*() aufgerufen. Diese schreibt den End-Tag für das Root-Element und schliesst dann den *XmlWriter*.

3.3.3 Systemimport

Abschließender Schritt ist der Datenbankimport der im SINIC-Format abgelegten Daten. Hierfür ist die Klasse blReadUniversalXML verantwortlich. Zunächst müssen die Daten aus dem SINIC-Format gelesen werden, dann folgt die Verarbeitung. Als grundlegende Technik findet Xpath in Kombination mit XpathDocument Verwendung. Den Einstieg in den Prozess bildet die Methode *blReadUniversalXML.QueryXML*, welcher der Name der zu lesenden Sinic-Format-Datei übergeben wird.

Zunächst wird ein Streamreader auf die übergebene Datendatei angelegt. Mit diesem Streamreader wird dann ein *XpathDocument* instanziiert. Um mit den Elementen des *Xpath*-Documents umgehen zu können, werden zudem drei zusätzliche Objekte benötigt. Ein *XpathNavigator*, der die Erzeugung von *Xpath*-Ausdrücken erlaubt, eine Variable des Typs *XpathExpression*, die diese Ausdrücke speichern kann, und ein *XpathNodeIterator* (Iterator), mit dem über die im Dokument per *Xpath*-Expression ausgewählte Knotengruppe iteriert werden kann.

Ablauf

1. Zunächst wird per Xpath der Inhalt des <DB> -Tags selektiert und als Datenbankname gespeichert. Die Vorgehensweise soll exemplarisch für die nachfolgenden Xpath-Selektionen gezeigt werden:

```
//Datenbank ermitteln
xpathQuery = "//DB/text()";                //Query-String angeben
xpathExpr = xpathNav.Compile(xpathQuery); //in ein XpathExpr. umwandeln
xpathIter = xpathNav.Select(xpathExpr);  //die Knoten der XpathExpr.
auswählen
xpathIter.MoveNext();                      //den ersten der Knoten lesen
qRes = xpathIter.Current.Value;            //DB Name in qRes speichern
```

Als nächstes werden alle <SinicDataSubSet> -Tags selektiert. Falls solche vorhanden sind, werden in einem weiteren Schritt die Attribute der Elemente ausgelesen und die Inhalte in entsprechende Variablen für Tabellennamen und Keys eingetragen. Zum annavigieren der Attribute stellt der *XpathNodeIterator* die Methode *MoveToNextAttribute*() zur Verfügung.

2. Nun werden die <SinicDataSet> -Tags selektiert und ihre Attribute ausgelesen. um auch für diese primäre Ebene die Informationen über benötigte Tabellen und Keys zu erhalten. Die vorbereitenden Schritte sind damit abgeschlossen.

3. Der nächste Schritt ist, den Iterator auf den Anfang des *SinicDataSet* – Knotensatzes zurückzusetzen. Über diesen Knotensatz, welcher den Ausgangspunkt für alle enthaltenen Datensätze bildet, wird im folgenden iteriert.

4. Die Funktion *getConnected* wird aufgerufen.

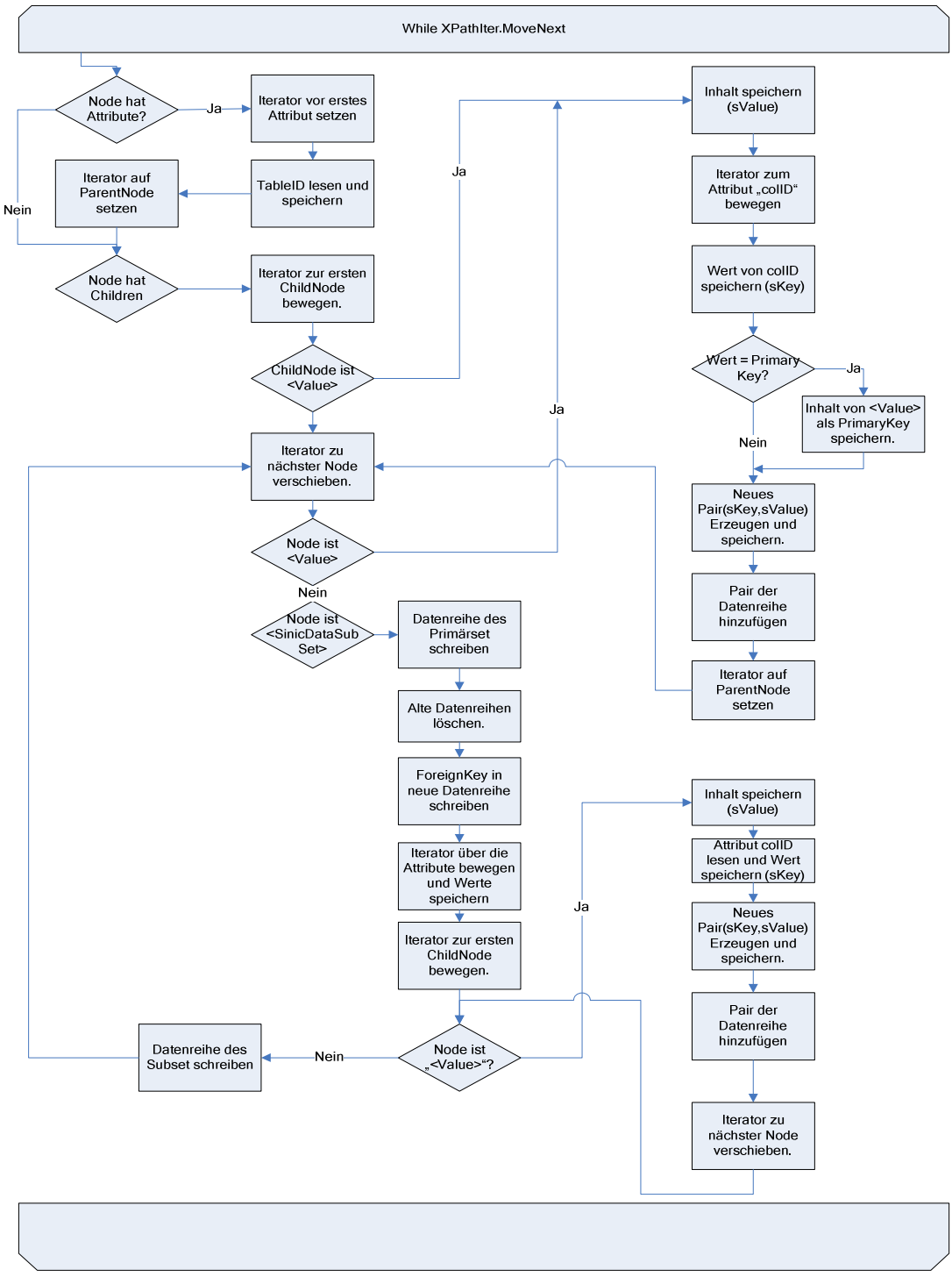
Code

```
public void getConnected(string dbName, string pTable, string sTable)
```

Diese Funktion ist dafür verantwortlich, die korrekte Verbindung zur Datenbank herzustellen. Hierzu wird zunächst eine neue *SinicDBShell* geöffnet, der ein SELECT-Statement zur Selektion der kompletten Daten der primären Zieltabelle übergeben wird. Mit diesen Daten wird ein *DataSet* gefüllt, das nun eine Abbildung der Zieltabelle darstellt. Analog wird auch für Tabellen der zweiten Ebene verfahren.

Dann werden die aus der XML-Datei gelesenen Informationen verwendet, um die Tabellen innerhalb des Datasets mit den korrekten Keys zu versehen.
Die Manipulation der Daten kann nun begonnen werden.

5. Das Iterieren über die Datensätze beginnt. Ziel ist es zunächst, aus der XML-Struktur die kompletten Datensätze des Quellformats wiederherzustellen. Der Ablauf kann folgendem Schaubild entnommen werden:



6. Als Ergebnis des abgebildeten Algorithmus' entstehen *List<Pair<string>>* -Objekte. Dies bedarf einer Erklärung. Die Klasse *Pair* ist eine Klasse zur Aufnahme zweier generischer Objekte. In diese Objekte werden die zu einem Datensatz gehörenden Feld-Wert Zuordnungen gespeichert und einer *List<>* hinzugefügt. Ein *List<Pair<string>>* -Objekt stellt somit einen kompletten Datensatz (mRow) dar. Dieses Objekt wird dann in die *updateData*-Methode übergeben, um das in der *getConnected*-Methode erzeugte DataSet zu manipulieren.

Code

```
public void updateData(string sTable, List<Pair<string>> kv)
```

7. Die *updateData*–Methode übernimmt, wie bereits erwähnt, die eigentliche Datenmanipulation. Als erster Schritt werden temporäre Objekte der Typen *DataRow* und *DataColumn* erzeugt. Nach Auswahl der passenden Tabelle mittels der übergebenen *sTable*-Variable werden diesen Objekten zusammen mit einer *DataTable* und einer *DataColumnCollection* die Strukturdaten der zu ändernden Tabellen zugewiesen. Das Gerüst der Tabellen und ihrer Komponenten liegt nun in den Objekten vor. Dann wird *browseDataRow* aufgerufen.

Code

```
private void browseDataRow(ref DataRow oDart, DataColumnCollection mCC,
List<Pair<string>> kv)
```

8. *browseDataRow* ist eine kurze Methode, die eine einzige Funktion übernimmt: sie füllt das als Referenz übergebene, lokale *DataRow*-Objekt mit Daten. Um dies zu erreichen iteriert sie in einer doppelten Schleife über alle *DataColumns* der *TableRow* und alle *Pair*-Objekte der *List<Pair>*. Wird eine Übereinstimmung zwischen Spaltenbezeichner des *Pairs* und Name der *DataColumn* entdeckt, wird der Wert des Pair-Objekts der *DataColumn* als Inhalt zugewiesen.

9. Zurück in der updateData-Methode liegt nun das mit Daten gefüllte *DataRow*-Objekt vor. Es wird als nächstes ermittelt, ob bereits eine Zeile mit diesem Primary Key vorliegt oder nicht.

Code

```
oTemp = mPrimaryDS.Tables[tb].Rows.Find(oDart[sPKey]);
```

Wird eine passende Zeile gefunden, so werden die Inhalte der alten Zeile mit den Inhalten der neuen Zeile überschrieben. Wird eine solche Zeile nicht gefunden, so wird die neue Zeile dem Dataset hinzugefügt.

10. Der Ablauf, der unter Punkt 6 begonnen wurde, wird solange durchgeführt, bis über alle *SinicDataSets* iteriert worden ist.

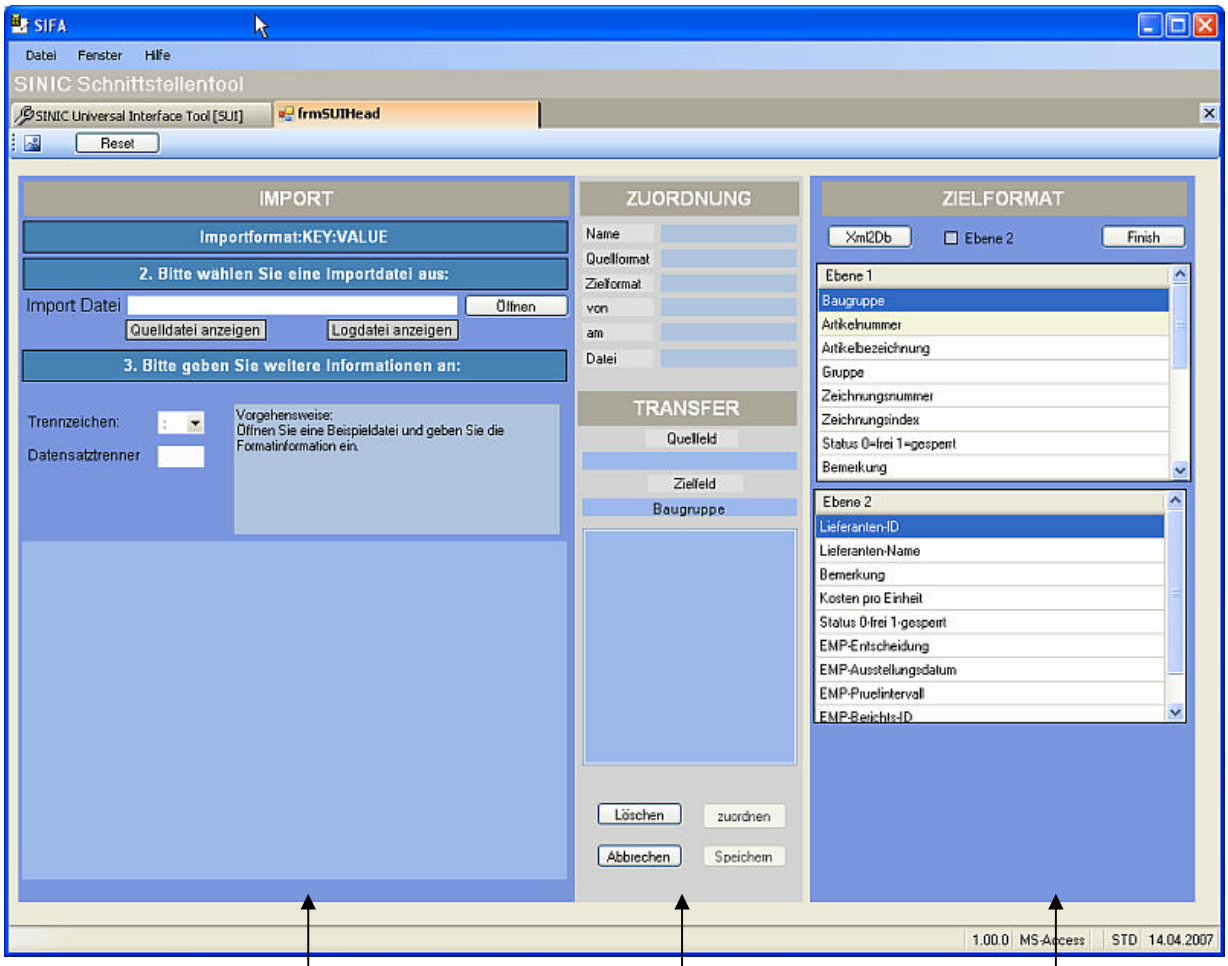
11. Der abschliessende Punkt ist das Schreiben der Daten in die Datenbank. Bisher liegen die Daten nur in dem Dataset vor, das aus der betroffenen Datenbank erzeugt worden ist. Die Methode *writeData*() übergibt das veränderte DataSet an die *DoUpdate*-Methode der *SinicDBShell*. Ab diesem Zeitpunkt kümmern sich die im Framework verankerten Routinen um das korrekte Schreiben in die Datenbank.

Mit Punkt zwölf ist der Importvorgang abgeschlossen. Es bleibt nur, den Vorgang ordnungsgemäß mit den Ergebnissen in eine Logdatei zu schreiben.

3.4 GUI

3.4.1 Darstellungsform/Grundsätzlicher Aufbau

Die Windows-Form *frmSUIHead* ist als „Tab“ in einer Basisform eingehängt, die von *SINIC.Base.WinControls.Forms.frmMainBase2* abgeleitet ist. Diese stellt die Menüstruktur zur Verfügung und ist integraler Bestandteil zur Einbindung der neuen Anwendungen in das SINIC-Framework. Die Kopfmaske *frmSUIHead* leitet von *SINIC.Base.WinControls.Forms.frmSINIC* ab. *frmSinic* stellt allgemeinverwendbare Funktionen wie z.B. ein Schließen der Maske per Tastendruck, eine Übersetzungsfähigkeit der Oberflächentexte und ein automatisches Befüllen von *Comboboxes* zur Verfügung.



Screenshot, frmSUIHead

IMPORTBEREICH **ZUORDNUNGS-/INFOBEREICH** **EXPORTBEREICH**

Die grundsätzliche graphische Darstellung ist umgesetzt über die Gruppierung der Controls in farblich voneinander abgesetzte *Panels*. .NET-Panels bieten den Vorteil, dass sie flexibel einsetzbar sind, ohne eine großen organisatorischen Overhead zu erfordern. Prinzipiell sind es - einfach gesprochen - Flächen, auf denen andere Elemente angeordnet werden können. Der Visual Studio -Designer ordnet automatisch alle auf einem Panel befindlichen Controls als Child-Elemente unter das Panel ein. Dadurch kann man komfortabel größere Mengen an Windows-Elementen verschieben, die Sichtbarkeit ändern und an unterschiedlichen Stellen je nach Bedarf positionieren.

3.4.2 Importbereich

Der Importbereich ist gemäß des Konzepts von oben nach unten in Reihenfolge des benötigten Userinputs gestaffelt. Die Anzeige des gewählten Importformats wird als Label realisiert, dessen Text automatisch durch die Importauswahl bestimmt wird. Der darunter liegende „Öffnen"-Button ist mit einer speziellen Windows-.NET-Control, *openFileDialog,* gekoppelt. Ein *openFileDialog* wird nicht an eine bestimmte Stelle platziert, sondern kann einer Windows-Form generell als Ressource hinzugefügt werden. *OpenFileDialog* bietet die komplette Funktionalität des bekannten „Datei öffnen" -Dialogfeldes. Über Parameter können Voreinstellungen wie unterstütze Dateitypen, Defaults etc. eingestellt werden. Das gleiche gilt analog auch für die verwendeten *saveFileDialogs.*

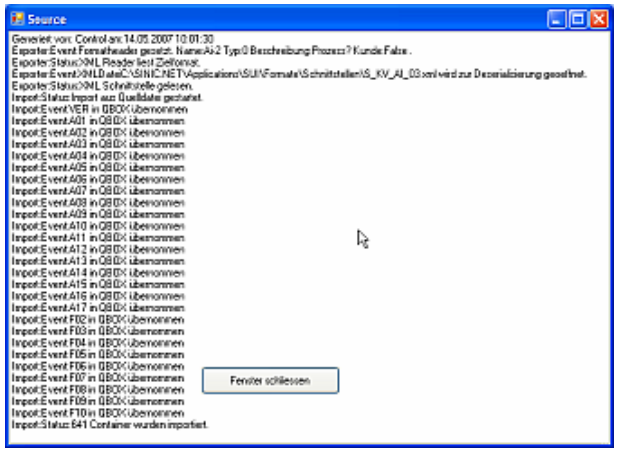
Code

```
saveFileDialog1.Filter = "(*.xml)|*.xml";
saveFileDialog1.FilterIndex = 1;
saveFileDialog1.RestoreDirectory = true;
if (mEditMode.Equals("Copy") || mEditMode.Equals("Edit"))
                    saveFileDialog1.FileName = formatheader.Filename;
                    if (saveFileDialog1.ShowDialog() == DialogResult.OK)
```

Im Beispiel wird zunächst die Liste „Dateityp" befüllt. Die Property „Filter" zeigt als Beschreibung (*.xml) an und filtert nach Auswahl die Anzeige im aktuellen Verzeichnis nach *.xml. Diese Liste könnte mit weiteren Einträgen versehen werden - falls sinnvoll. Die Property *FilterIndex* wählt den ersten (und in unserem Fall einzigen) Eintrag der Liste und setzt ihn als Default. *RestoreDirectory* auf *true* zu setzen bewirkt, dass sich die Anwendung das zuletzt ausgewählte Verzeichnis merkt und bei erneutem Aufruf des Dialogs dieses wieder anzeigt.

Darunter finden sich zwei Labels, „Quelldatei anzeigen“ bzw. „Logdatei anzeigen“, die aus optischen Gründen den Vorzug vor regulären Windows-Buttons bekommen haben. Die Funktion ist allerdings die Gleiche. Beide öffnen ein neues Fenster, das die geloggten Ereignisse bzw. die Quelldatei, sofern es sich um eine Textdatei handelt, anzeigt. Beides geschieht in einem neuen Fenster, das im modalen Modus geöffnet wird.

Button „Quelldatei anzeigen“



Button „Logdatei anzeigen“

Dies stellt eine Abweichung von der im Konzept geplanten integrierten Ansicht im Zuordnungsbereich dar. Motiviert wurde die Abweichung durch die zum Teil sehr „sperrigen“ Importformate. Große Zeilenbreiten und sehr lange Dateien machten die Darstellung für einen Teil der importierten Formate nicht praktikabel. Die jetzige Lösung ist zum einen in der Darstellung flexibler, und zum anderen wird mehr Platz für die anderen Elemente des Zuordnungsbereichs frei. Im mittleren Bereich der Importsektion finden sich unter dem Label „Weitere Informationen“ noch einige formatabhängige Controls, über welche Separatoren, Fixed-Record-Felder und ähnliche Dinge eingestellt werden können. Sie werden ja nach Format ein- oder ausgeblendet. Es handelt sich dabei um Textboxen und Comboboxen (eine Windows-Combobox entspricht den aus dem Internet bekannten Dropdown-Boxen).

Das hellblaue Panel ganz unten ist der reservierte Platz für die Darstellung der zwecks Vornahme der Zuordnung geöffneten Beispieldatei. Was dort angezeigt wird entscheidet sich erst zur Laufzeit. Möglich sind zwei verschiedene Controls. Hat der User als Importformat Key:Value, FRL ,CSV oder Excel gewählt, so erscheint an dieser Stelle eine

DataGridView. Falls die Wahl auf Access oder XML gefallen ist, dann erscheint eine *SinicTreeView*.

3.4.2.1 DatagridView

Betrachten wir zunächst den Fall, dass eine *DataGridView* dargestellt werden soll. Problematisch ist an dieser Stelle, dass die unterschiedlichen Formate in ein überschaubares und - wenn möglich - ähnliches Format gebracht werden sollen. Wie im Konzept beschrieben liegt die Vorgabe in einer Feldliste, die in vertikaler Reihenfolge die zuzuordnenden Felder auflistet. Dies ist am einfachsten für die Formate umzusetzen, die in der Ursprungsform bereits eine ähnliche Struktur aufweisen, z.B. das Key:Value Format.

Die Hauptarbeit für die Darstellung des Grids haben zwei Funktionen, *PopulateDataGridView* und *SetUpDataGridView*.

```
private void SetUpDataGridView(ref DataGridView dgv, ref List<string[]>
r, string sCaller)
```

```
private void PopulateDataGridView(ref DataGridView dgv, List<string[]> r,
string sCaller)
```

Die *DataGridView* ist eine der mächtigsten Windows-Controls im .NET-Framework und daher äußerst umfangreich mit Properties und Methoden ausgestattet. *SetUpDataGridView* legt zunächst Optik, Größe und Position des Grids fest, sowie einige andere Eigenschaften. Wichtig an dieser Stelle ist, den EditMode des Grids auf *EditProgrammatically* zu setzen. Nur so ist es möglich, das Grid programmatisch „on the fly“ zu befüllen. Entsprechend des anzuzeigenden Formats werden dann Spalten angelegt und die Spaltenüberschriften gesetzt. Um den Darstellungsbesonderheiten für FRL-Formate Rechnung zu tragen, werden Start und Endindizes ein- oder ausgeblendet.

Einige Möglichkeiten, die später zu Problemen führen können, werden deaktiviert, so z.B. die Eigenschaft Multiselect, die Mehrfachauswahlen erlaubt. Da einzelne Zellen ausgewählt werden sollen, aber nicht ganze Zeilen oder Teile des Zellinhalts, wird die Eigenschaft *SelectionMode* zudem auf *Cellselect* gestellt. Die Properties des *DataGridViews* müssen genau untersucht werden. Erst nach genauer Prüfung der erlaubten und gewünschten Interaktion zwischen User und GUI können die Einstellungen so

vorgenommen werden, dass der Programmablauf nicht durch unerwünschte Nebeneffekte gestört wird.

Nachdem das *DataGrid* nun vorbereitet ist, muss es noch befüllt werden. Hierzu ist *PopulateDataGridView* in zwei Varianten befähigt. Die Funktion bekommt unter anderem eine Liste gefüllt mit String-Arrays übergeben (siehe Aufruf). Dort sind die im Grid anzuzeigenden Daten enthalten.

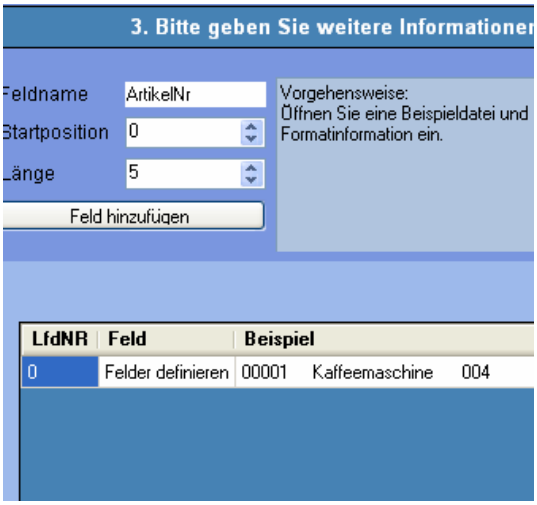
Zeilen können einer DatagridView relativ einfach hinzugefügt werden. Die *DataGridView* beinhaltet ein *DataGridViewRowCollection*, die eine Auflistung *DataGridViewRow*-Objekte enthält. Diese Collection verfügt über eine *Add*-Methode, der man ein String-Array übergeben kann. Die Strings des Arrays werden der Reihe nach in die in der *DataGridView* enthaltenen Spalten eingeordnet. Auf die Länge des Arrays ist zu achten, um eine Exception zu vermeiden.

Es muss zwischen drei unterschiedlichen Fällen entschieden werden. Handelt es sich um ein zeilenorientiertes Format wie *Key:Value*, können die Werte relativ einfach übernommen werden. Ist es jedoch ein spaltenorientiertes Importformat wie *CSV* oder *.XLS*, so müssen die Daten erst „um 90 Grad gedreht“ werden. Bei den spaltenorientierten Formaten stehen die Feldbezeichner in der Regel in den Spaltenüberschriften mit den entsprechenden Werten darunter, während das Grid laut Konzept eine vertikal verlaufende, zeilenweise Anordnung anzeigen soll. Folglich muss eine Umwandlung erfolgen und die *RowHeader* gesetzt werden, bevor dem Grid neue Reihen hinzugefügt werden können.

Codeauszug

```
int nI =0;
//...
//Daten stehen in List<string[]> r
//erste Zeile enthält Feldbezeichnungen
int nFelder = r[0].Length;
//Daten um 90 Grad drehen, Zeilen in Spalten umwandeln
for (int counter = 1; counter <= nFelder; counter++)
    {
     rows.Add(new string[5] { nI.ToString(), r[0][counter - 1],"-","-",
r[1][counter - 1] });
if (sCaller.Equals("Importdaten"))rows[nI].HeaderCell.Value =
r[0][counter-1];
     nI++;
  }
```

Einen Sonderfall stellt die *DatagridView* für bisher nicht spezifizierte FRL-Formate dar. Um die im Konzept erläuterte interaktive Eingabe des Userformats möglich zu machen, wird analog zur gerade beschriebenen Vorgehensweise die DataGridView gefüllt. Allerdings bleibt das Grid zunächst bis auf die erste Zeile leer. In dieser Zeile wird ein aus der Beispieldatei importierter String angezeigt, der einen Datensatz des FRL-Formats repräsentiert. Im GUI-Bereich „weitere Informationen“ stehen Eingabefelder zur Verfügung, um die Länge und Bezeichnungen der Formatfelder einzugeben.



Die Variable *bSetFieldsInteractive* wird auf “true“ gesetzt, um die alternative Bearbeitung einzuleiten. Nachdem der User ein Feld definiert hat, wird der String aus dem Grid an der angegebenen Stelle zerlegt. Die erste Zeile enthält dann nur noch den definierten Abschnitt unter der angegebenen Bezeichnung, während der Rest des Strings in die nächste Zeile des Grids verschoben wird.

Um dies zu realisieren wird ein geänderter Aufruf von *PopulateDataGridView* verwendet.

```
private void PopulateDataGridView(ref DataGridView dgv, string[] r, int
nOffset)
```

Er enthält eine Referenz auf die *DataGridView*, die einzutragende neue Zeile und einen Offset. Der Inhalt der neuen Zeile wird vor dem Aufruf berechnet und innerhalb der Routine nicht mehr weiter bearbeitet. Der Offset ist nötig, da die vom User eingegebenen Werte sich in dieser Variante auf die optisch dargestellten Stringpositionen beziehen. Diese Werte müssen um die echten Positionen bereinigt werden, um intern eine korrekte Speicherung des Formates zu erreichen.

Ein Beispiel:
Als erste Wahl bestimmt der User Feldposition 0, Länge 10, unter der Bezeichnung Artikelnummer. Artikelnummer steht nun mit dem entsprechenden Stringausschnitt in Zeile eins. Für das zweite Feld wählt der User Position 0, Länge 6 der zweiten Reihe und speichert sie unter Warengruppe. Um eine korrekte Formatinformation zu erlangen, muss die Eingabe auf die reale Position des Feldes gewandelt werden (Startposition:11). Dazu ist ein wenig Codejonglage nötig; sowohl um den Inhalt der Felder korrekt darzustellen, als auch um die Positionen in GUI und Format korrekt zu führen.
Um Fehler bei der Stringbehandlung zu vermeiden, werden alle Eingaben des Users zunächst gegen die Längen der Strings geprüft, bevor ein Funktionsaufruf eingeleitet wird.

Um die angelegten Felder zu überprüfen, kann der User durch das erneute Öffnen einer Beispieldatei einen Import starten und die Anzeige im DataGridView inhaltlich mit dem Quellformat vergleichen. Dazu wird das im Import-Kapitel beschriebene Verfahren zum Einlesen von Textformaten (FRL) verwendet.

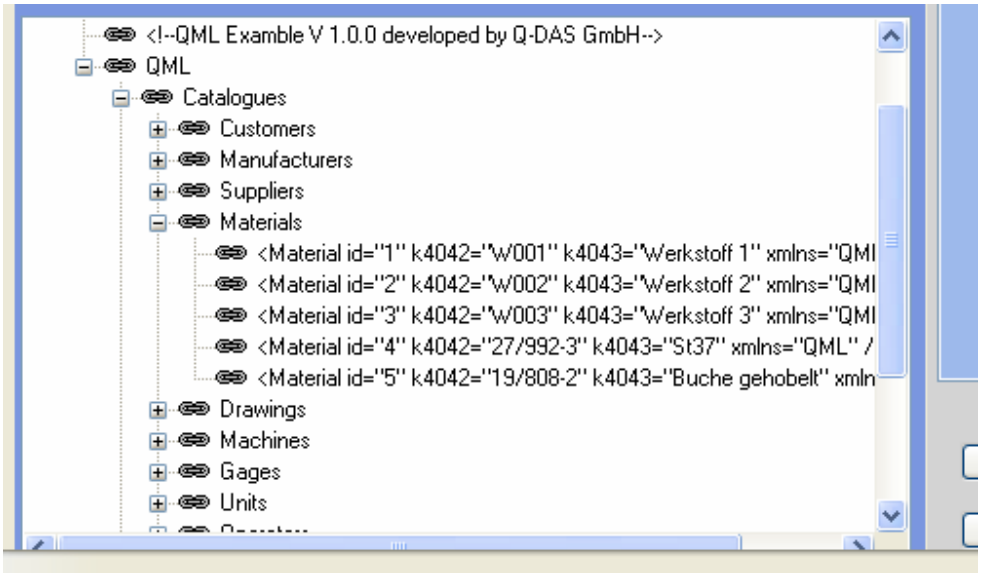
3.4.2.2 DatagridView Events

Die Event-Behandlung in einer *DataGridView* ist äußerst umfangreich. Da es sehr viele unterschiedliche Events gibt, die sich für den Benutzer aber nicht zwangsläufig unterscheiden, müssen die behandelten Events sorgfältig gewählt werden. Zum Beispiel ist der Klick auf den Inhalt einer Zelle ein anderes Event (*cellContentClick*), als das klicken

auf die Zelle selbst (*cellClick*). Um den Benutzer an dieser Stelle nicht zu verwirren, werden ähnliche Events zusammengefasst und dem gleichen Eventhandler zugeführt.

3.4.2.3 TreeView

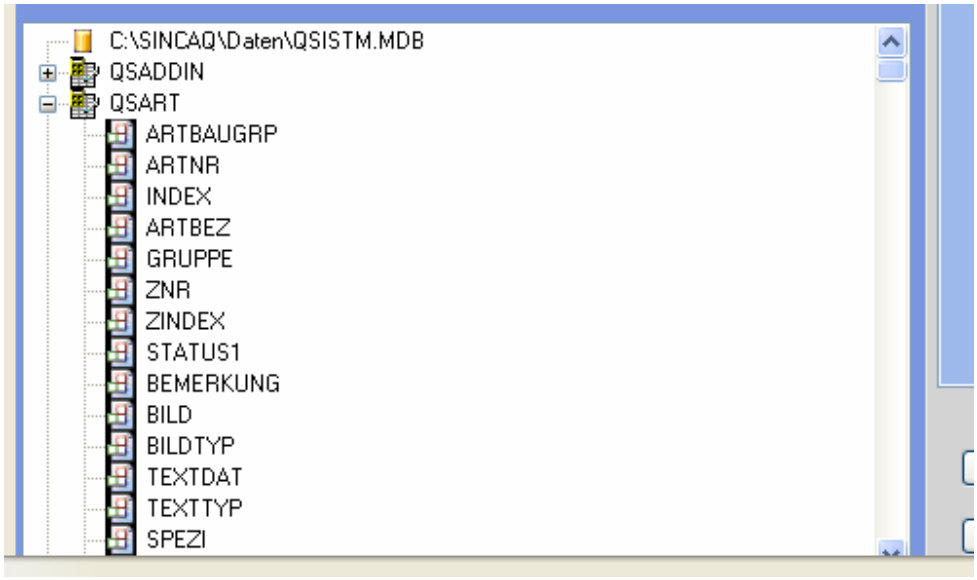
Für die Variante zur Darstellung von XML-Daten, wird die Struktur des XML-Dokuments in eine Baumstruktur übersetzt. Hierzu werden XML-Nodes, wie bereits im Kapitel Import beschrieben, als Tree-Nodes in den Baum eingehängt. Funktionalitäten - wie auf- und zuklappen des darunter liegenden Astes - sind in der Control bereits integriert.



XML-Importdarstellung

Ein wenig anders ist es beim Darstellen von Datenbankstrukturen. Eine Nodes-Struktur wie bei XML ist dort nicht gegeben. Als Darstellungsform wurden drei Ebenen gewählt, der Name der Datenbank-Datei ist der Wurzelknoten (Root-Node). In Ebene zwei werden alle Tabellen dargestellt und in Ebene drei dann die Spalten der Tabellen. Um das ganze optisch ansprechender aufzubereiten wurde eine Windows-*ImageList* der Form hinzugefügt. Dies ist eine Ressource ähnlich der im Importteil beschrieben *FileDialogs*. Eine *ImagLlist* enthält Grafiken, die zur Verwendung in verschiedenen Controls zur Verfügung stehen. In der TreeView ist das entsprechende Property der *ImageIndex*. Der *ImageIndex* bezieht sich auf die Indexnummer der der TreeView zugeordneten *ImageList*. Es können der TreeView

bestimmte Indizes als Default eingestellt werden, die dann programmatisch je nach Situation geändert werden können.



Access-Importdarstellung

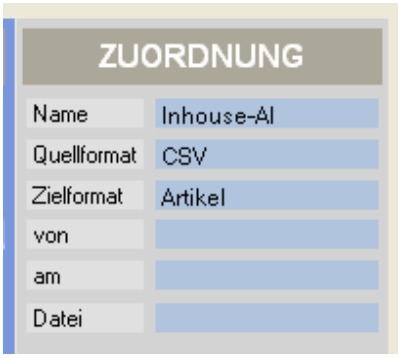
3.4.2.4 TreeView Events

Als Event zur weiteren Bearbeitung wird *NodeMouseClick* verwendet. Im Eventhandler wird abgefragt, auf welcher Ebene der Baumstruktur die auslösende Node lag, bevor entsprechend reagiert wird. Im vorliegenden Fall lösen nur Klicks auf unterster Node-Ebene eine Zuordnung aus, da dort die Datenbankfelder verankert sind.

3.4.3 Zuordnungs- und Informationsbereich

Der obere Bereich des mittleren Panels wird durch Labels gekennzeichnet, die Informationen zu der vom User gewählten Schnittstelle anzeigen. Die Labels übernehmen die vom User in *frmSchnittstelle* angelegten Daten. Die Beschriftungen für „von“, „am“ und „Datei“ werden gesetzt, wenn der User in der Listenmaske „bearbeiten“ oder „kopieren“ gewählt hat; ansonsten sind sie leer.

frmSchnittstelle



Darstellung frmSUIHead

Der Platz für diese Informationsfelder wurde durch die Ausgliederung der Quelldatei-Anzeige in eine eigene Form frei (s.o.). Darunter werden die eigentlichen Zuordnungen angezeigt und durchgeführt.

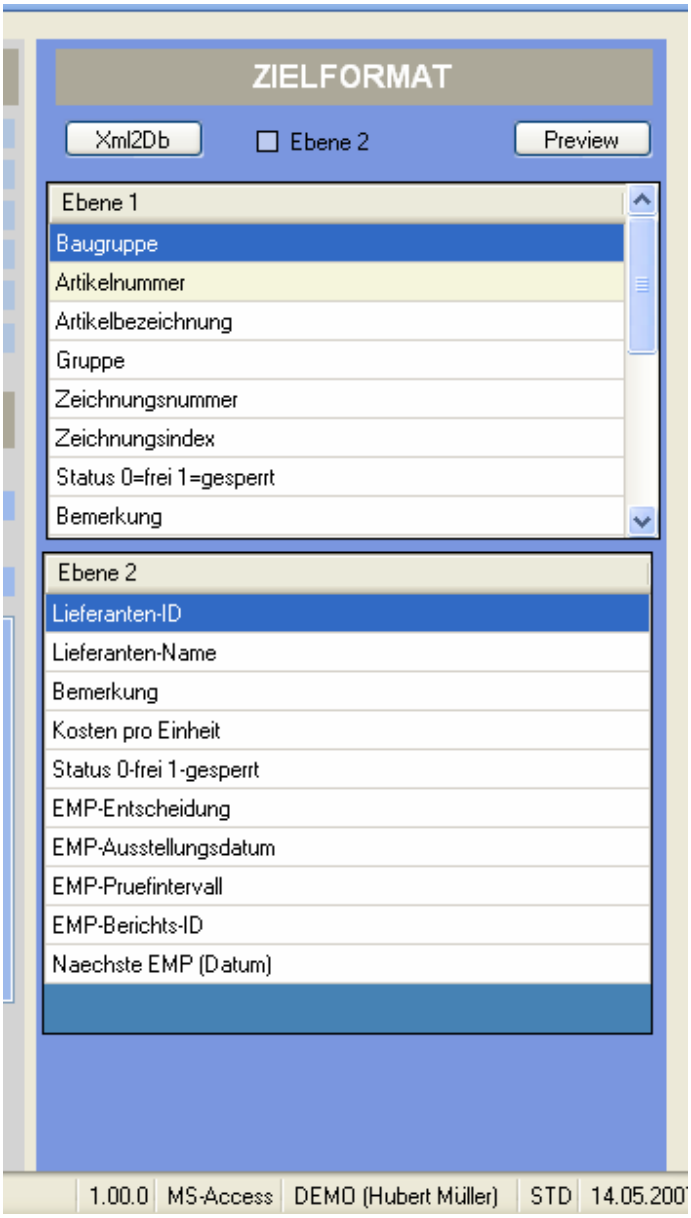
Quellfeld und Zielfeld zeigen die in Quell- und Zielformat gewählten Felder an. Existiert ein gewähltes Zielfeld bereits, so springt die Anzeige für das Quellfeld auf das entsprechende Quellfeld um. Umgekehrt ist dieser Automatismus nicht implementiert, da die Möglichkeit gegeben sein soll, ein zugeordnetes Feld bei falscher Zuordnung neu zu belegen, ohne den Weg über Löschen und Neuanlage zu gehen.

Umgesetzt ist die Zuordnungsliste als eine *Listbox*-Control. Zunächst findet eine Überprüfung statt, ob das Quellfeld bereits zugeordnet ist. Falls ja, wird eine Alertbox angezeigt und der User um Auswahl gebeten. Falls nicht, dann wird ein neues String-Item der Listbox hinzugefügt, welches aus den beiden Feldbezeichnungen und einem Verbindungspfeil besteht. Alle Listeneinträge sind anklickbar und können auf Knopfdruck

auch wieder entfernt werden. In punkto Darstellung gibt es in diesem Bereich insgesamt wenig Besonderheiten. Wird eine Schnittstelle zur Bearbeitung geöffnet, so wird die Zuordnungslistbox mit den Inhalten der in der Schnittstelle gespeicherten Zuordnungen befüllt. Technisch läuft das so ab, dass die Methode *readXMLSchnittstelle* eine Liste mit Trans-Containern zurückgibt, aus der sich die GUI die entsprechenden Listeneinträge generiert.

Nachdem dies geschehen ist, findet nach dem Öffnen einer Beispieldatei noch ein Abgleich mit den tatsächlich existierenden Quellfeldern statt, um Inkonsistenzen bei der Bearbeitung der Zuordnungsliste zu vermeiden.

3.4.4 Exportbereich



Im Exportbereich werden primär die Felder des Zielformats angezeigt. Diese sind als *DataGridView*-Controls implementiert. Befüllt werden sie automatisiert duch das Lesen der *XmlDBSchemas*.

Zunächst ruft die GUI die Methode *readXMLFormat* von *blExporter* unter Angabe der Indexnummer des gewünschten Zielformats auf. *blExporter* parst das passende *XmlDBSchema* und liefert eine Liste mit Export-Containern an die GUI zurück. Diese *List<blExport>* stellt die Grundlage zur Befüllung der DataGridViews dar.

Die GUI legt nun lokal je Ebene eine *List<string[]>* -Variable an,

die entsprechend der Level-Eigenschaft der Export-Container mit den in den Containern enthaltenen Strings befüllt werden. Dabei werden nur die für die GUI relevanten Daten berücksichtigt: der Feldname, die ID des Containers und das Flag für die Kennzeichnung von Pflichtfeldern. Analog zur im Abschnitt Importbereich beschriebenen Vorgehensweise, werden für die drei DataGridViews die Routinen *SetupDataGridView* und *PopulateDataGridView* aufgerufen, die als Parameter die jeweilige *List<string[]>* übergeben bekommen.

Das Verfahren zur Population unterscheidet sich nur in Details von der Vorgehensweise für das Importgrid. Zum einen werden Pflichtfelder in den Grids farbig markiert, zum anderen werden die Spaltenanzahl und Überschriften auf die etwas andere Darstellung angepasst. Die Eventhandler sind analog zu dem *DataGridView* im Importbereich behandelt.

Ein Problem, dass in dieser Form bisher nicht auftrat, ist der verfügbare Platz. Zwar muss Platz für drei Ebenen vorgesehen werden, in der Praxis ist dies jedoch mit Abstand der seltenste Fall. Eine bis zwei Ebenen sind die Regeln. Das Problem wird dadurch abgemildert, dass den Grids dynamisch unterschiedlich viel Platz zugewiesen wird. Entsprechend des Vorhandenseins von Ebene drei wird die Größe der anderen Grids zur Laufzeit angepasst, um den Benutzerkomfort zu erhöhen. Programmatisch ist dies durch ein neues Setzen der entsprechenden Eigenschaften Position, Width und Height möglich.

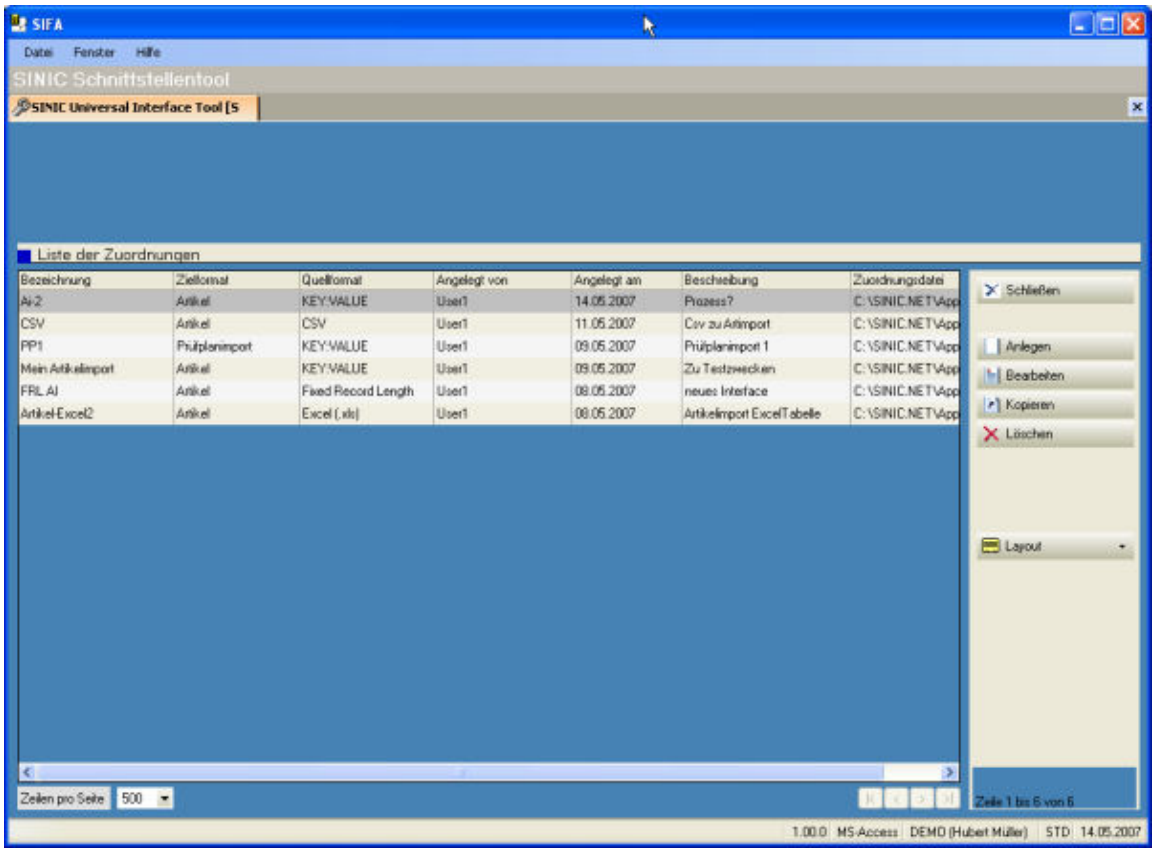
Im oberen Bereich sind die im Konzept skizzierten Buttons umgesetzt. *Xml2Db* löst den SINIC-Systemimport aus, Preview erzeugt das SINIC-Format. Beide Verfahren sind bereits ausführlich beschrieben worden. Als Optionsfeld ist die „Ebene 2“ Checkbox ein Schalter, der sich auf die Plausibilitätsprüfung bezüglich der Zielfelder auswirkt. Ein Häkchen in dieser Box bedeutet, dass die zweite Ebene in die Prüfung mit einbezogen wird. Dies ist deshalb bedeutsam, da für manche Formate die Existenz der zweiten Ebene nur optional ist, was berücksichtigt werden mußte.

Abweichend zum Konzept ist, analog zur Wahl des Quellformats, auch die Wahl des Zielformats aus dieser Maske verschwunden. De facto machte eine Umstellung des Zielformats nach Beginn der Zuordnung einen kompletten Neubeginn des Prozesses nötig. Durch die Umsetzungen in der jetzigen Form wirkt der Exportbereich aufgeräumter. Die

zum Handling der Userinteraktionen nötigen Prozesse und die Handhabung im Backend wurden durch diesen Schritt vereinfacht. Die Wahl des Zielformats wird vor dem Öffnen dieser Oberfläche durch die unter 3.4.3 abgebildete *frmSchnittstelle* erledigt.

3.4.5 Schnittstellenverwaltung

Die Verwaltung der Schnittstellen erfolgt über die Klasse *frmSUIListe* des SUI-Tools. Diese Oberfläche kam erst relativ spät im Projektverlauf hinzu, daher ist sie im Anfangskonzept, das ein Schnittstellenmanagement rein auf Dateiebene vorsah, nicht enthalten. Der Benutzer hat an dieser Stelle die Möglichkeit, seine bereits angelegten Schnittstellen nach verschiedenen Kriterien zu sortieren, sie zu bearbeiten, neu anzulegen oder zu löschen. Die Liste selbst kann nach Benutzervorgaben umstrukturiert werden.
Sobald eine Eintragung innerhalb der Liste ausgewählt wurde, oder ein neuer Eintrag erstellt werden soll, öffnet sich die eigentliche Bearbeitungsoberfläche, die sog. Kopfmaske oder Detailmaske. Die Listenmaske *frmSUIListe* ist von der Basisklasse *SINIC.Base.WinControls.Forms.frmList* abgeleitet - eine zwingende Vorgabe für Listenmasken, die *SinicDBGrids* in ihrer Struktur verwenden.



Screenshot, frmSUIListe

Der Kern der Oberfläche ist eine *SinicDBGrid*-Control. Diese Control beinhaltet primär die Anzeige einer Liste, die programmatisch an eine Datenbank angebunden werden muss. Das SUI-Tool speichert alle Schnittstellen in der Datenbanktabelle QSSUIHEAD, die in einer Access-Datenbank gespeichert wird. Der Aufbau der Datenbank ist wie folgt:

	Field Name	Data Type	Description
🔑	RECNO	Number	ID needed for Sinic-Framework.
	SUINAME	Text	Name of the interface.
	SUITARGET	Number	Name of the SINIC format /function we made the field assignments for.
	SUISOURCE	Number	File name and path of the saved SINIC interface file.
	CREATEUSER	Text	User who saved the interface.
	CREATEDATE	Date/Time	File creation date.
	DESCRIPTION	Text	General comments and description regarding the saved interface.
	SUIFILE	Text	File name under which the SinicSchnittstelle was saved.
▶			

Tabellenansicht. Datenbank: SINQM.mdb, Tabelle: QSSUIHEAD

Die Felder sind relativ selbsterklärend. Quell- und Zielformat werden gespeichert, ebenso der erstellende User, das Erstelldatum, eine Beschreibung und der Pfad zur Schnittstellendatei. Das Feld „RECNO“ (Record Number) ist der Primärschlüssel der Tabelle, wobei die Feldbezeichnung innerhalb des SINIC-Frameworks gängig ist.

Der Inhalt der Tabelle wird nach Programmstart automatisch angezeigt, wobei die Anbindung einer Datenbank an eine solche SINIC-Control nicht ganz trivial ist. Das SINIC-Framework stellt einige nützliche Funktionen in Verbindung mit einer derartigen Control zur Verfügung. So z.B. die Möglichkeit, eine vom User gewählte Sortierung der Spalten zu speichern; ein Paging für den Fall, dass die Liste sehr viele Einträge enthält; Filterung nach spezifischen Kriterien, um die Anzeige der Daten einzuschränken und einige mehr. Viele dieser Möglichkeiten sind jedoch optional verwendbar und können je nach Bedarf in den Code eingebunden oder entfernt werden. Da die Anzahl der Schnittstellen pro Kunde in einem überschaubaren Bereich liegt, wurde auf die Integration eines Pagings und einer Filterung verzichtet. Nichtsdestotrotz wäre eine Integration in kurzer Zeit möglich, da die Methoden bereits vorhanden sind und nur an die gegebene Datenstruktur der Anwendung anzupassen sind.

Zusammengefasst bietet die Oberfläche dem User folgende Möglichkeiten:

3.4.5.1 Anlegen

„Anlegen“ löst die Anlage einer neuen Schnittstelle aus. Zunächst wird *frmSchnittstelle* geöffnet, die dem User erlaubt, die generellen Informationen zu seiner Schnittstelle einzugeben. Hierzu gehören Quell- und Zielformat, eine Beschreibung, der Name und ein Freigabestatus. Der Freigabestatus ist dahingehend von Belang, da nur freigegebene Schnittstellen für einen Import verwendet werden dürfen.
Danach wird *frmSUIHead* in einem neuen Tab mit den angegebenen Informationen geöffnet. Die Klasse *blExporter* liest zunächst, wie im Kapitel Export beschrieben, das Zielformat aus den *XmlDBSchemas*. Das Erstellen der Zuordnungen kann nun wie oben beschrieben beginnen.

3.4.5.2 Bearbeiten

Öffnet zunächst die Oberfläche *frmSchnittstelle* mit den Eckdaten der zur Bearbeitung gewählten Schnittstelle. An dieser Stelle können bereits Anpassungen vorgenommen werden. Nach Bestätigung wird die Kopfmaske *frmSUIHead* in einem neuen Tab geöffnet. Die Klasse *blExporter* liest zunächst, wie auch bei „Anlegen“, das Zielformat. Dann wird die Schnittstellendatei, deren Pfadangabe und Name in der Schnittstellendatenbank gespeichert wurde, geöffnet und die Zuordnungen wiederhergestellt. Der User kann dann mit der Bearbeitung beginnen.
Sobald er die Bearbeitung beendet, kann er die geänderte Schnittstelle unter Verwendung des gleichen Dateinamens und des gleichen Datenbankeintrags abspeichern. Sollte der Benutzer sich für einen anderen Dateinamen entscheiden, wird die Ursprungsdatei gelöscht.

3.4.5.3 Kopieren

Kopieren funktioniert vom Procedere prinzipiell wie das Bearbeiten. Einziger Unterschied ist, daß zum einen der Name der neuen Schnittstelle per Default den um den Zusatz „Copy“ ergänzten Namen der kopierten Schnittstelle erhält, zum anderen muss die Schnittstellendatei zunächst kopiert werden. Ansonsten sind die Arbeitsschritte identisch zu denen beim „Bearbeiten“.

3.4.5.4 Löschen

Die ausgewählte Schnittstelle wird aus der Datenbank gelöscht und die zugeordnete Datei ebenfalls.

3.4.5.5 Layout

„Layout“ ist eine in der *SinicDBGrid*-Control bereits verankerte Funktionalität, die nicht gesondert angepasst werden mußte. Der User kann die Tabellenspalten in der Reihenfolge umsortieren und in der Darstellungsbreite anpassen. Diese Funktion ermöglicht es dem User, die gewählte Listendarstellung abzuspeichern.

3.5 Coupling

Die beiden Klassen, die die Hauptarbeit im Backend verrichten, *blImporter*(Importer) und *blExporter* (Exporter), sind als Komposition mit der *frmSUIHead* verbunden. Die Hauptkontaktpunkte zwischen der GUI und diesen beiden Klassen sind wie folgt:

1. Die GUI bekommt nach Auswahl einer Quelldatei eine Liste mit Import-Containern zur Ermöglichung der Anzeige im Importpanel zurückgeliefert.

2. Analog liefert der Exporter nach Auswahl des Zielformats eine Liste mit Export-Containern zurück, aus denen die Anzeige in den Export-*DataGridViews* ermöglicht werden.

3. Tracking der Userzuordnungen und Übergabe einer getroffenen Zuordnung an den Exporter in Form eines Trans-Containers.

4. Steuerung des Programmablaufs gemäß dem Userinput durch entsprechende Methodenaufrufe an Importer und Exporter.

Um effektiv zu sein hält *frmSUIHead* jeweils eine Liste für Import-, Export- und Trans-Container vor. Diese werden für die Darstellungen in fast allen Bereichen der GUI

verwendet. Gemäß der im Konzept erwähnten Regel: „Don't program to an implementation, program to an interface" sind die Variablen, die Containerlisten von Importer und Exporter entgegennehmen, als Interfacevariablen implementiert. Auf diese Weise interessiert sich die GUI nicht für die konkrete Implementation der Klassen, solange das entsprechende Interface von selbiger ordnungsgemäß implementiert wurde.

Was die Klasse frmSUIListe betrifft, die den Einstiegspunkt in die Anwendung darstellt, so sind die Kopplungen mit den darunter liegenden Klassen durch das SINIC-Framework vorgegeben und weitestgehend standardisiert. Um die zentrale Control der Klasse *ctlSinicDBGrid* an die neue Applikation zu binden, sind eine Vielzahl Anpassungen nötig, die sich in Ergänzungen bestehender Klassen äußert. An einigen Stellen, wie z.B. der Button zum Löschen bestehender Zeilen im *DataGridView*, ist das Überschreiben von Funktionen nötig. Allerdings werden hierbei keine Eingriffe in die zugrundeliegende Architektur, die bereits im Kapitel SINIC-Framework dargelegt worden ist, vorgenommen.

Die Kommunikation zwischen den verschiedenen Forms der GUI erfolgt in zwei Varianten. Zwischen *frmSUIListe* und *frmSUIHead* werden die benötigten Werte konform des SINIC-Frameworks über eine Hashtable mit Parametern übermittelt. frmSUIHead verwendet eine eigene Datenklasse zum Austausch der Daten zwischen sich und frmSchnittstelle. Diese Vorgehensweise ist generell empfohlen, wenn zwischen zwei Forms mehrere Daten oder umfangreichere Daten getauscht werden sollen. Der Vorteil liegt vor allem darin, dass zum kompletten Austausch aller Daten nur ein Funktionsaufruf benötigt wird. Da dieses Datenobjekt eine eigene Klasse darstellt, sind Anpassungen der Oberfläche sehr leicht zu verwirklichen. Zum Beispiel müssen für die Einführung eines zusätzlichen Feldes nur die GUI selbst und die Datenklasse verändert werden; die Aufrufe zur Interaktion zwischen den Forms werden davon aber nicht betroffen.

3.6 Automatisierung - SinicUIServ

Das SUI-Tool zur Erstellung einer Schnittstelle ist der Teil der Software, der gegenüber dem User den Kern der Software darstellt. Es muss jedoch auch noch einen Automatismus geben, der die beiden Prozessschritte - Umwandlung der Quelldatei ins SINIC-Format und Import in die Datenbank - ohne zusätzlichen Userinput durchführt. Zu diesem Zweck ist ein Windows-Service Teil der Software, der diese Arbeitsschritte entsprechend durchführt. Er stellt die automatisierte Umwandlungs- und Importkomponente des Systems dar und kann über die Windows-Dienstverwaltung gemäß der Userwünsche verwaltet werden. Sobald der Service gestartet wird, liest er zunächst seine Initialisierungsdatei aus. Dort ist sowohl das Zielverzeichnis - also das Verzeichnis, wo die Importdaten zu finden sind - hinterlegt, als auch der Name und Pfad der nötigen Schnittstellendatei und die Emailadresse, an die eine Reportemail gesendet werden soll. Der Service öffnet zunächst Schnittstellendatei und liest die Daten für Art des Imports und die getroffenen Zuordnungen aus. Verwendet werden hierzu die gleichen Algorithmen wie beim Bearbeiten einer bestehenden Schnittstelle im SUI-Tool. Danach wird die Importdatendatei geöffnet und die Zuordnungen mit den Import-Containern synchronisiert. Auch dies verläuft analog zu den Mechanismen im SUI-Tool. Danach erfolgt die Erstellung des SINIC-Formats gemäß dem im Kapitel Export angegebenen Verfahren. Alle laut getroffener Zuordnung relevanten Daten werden in eine XML-Datei vom Typ *SinicDoc* geschrieben. Diese wird dann im nächsten Schritt ausgelesen und die Werte in die entsprechenden Datenbanken geschrieben. Eine Email mit Statusbericht wird an die Serviceadresse gesandt und das Logfile geschlossen. Schlußendlich wird der Dienst beendet und wartet auf den nächsten Einsatz.

Codeauszug

```
            mExport=new blExporter();
            mImport = new blImporter();
            //Inidatei lesen
            mExport.readServiceIni(sIniFile);
            //Zielpfad merken
            sFinisher = mExport.ExportFile;
            //Schnittstelle lesen
            transform =
            mExport.readXMLSchnittstelle(mExport.InterfaceFile);
            //Quelldateityp bestimmen
            quellTyp=mExport.getTyp();
             //Zielformat lesen
            mExport.readXMLFormat((int)quellTyp);
            //K:V or CSV
            if (quellTyp == 0 || quellTyp == 2)
            {
            //Separator aktualisieren
            mImport.kvSep = mExport.Separator;
            import = mImport.importTextFile(quellTyp, "Export",
mExport.SourceFile);
            }
            else if ...hier folgen weitere Typunterscheidungen...
            }
            //SinicDoc erstellen
           //Universalformat schreiben
            mExport.finish("Export",sFinisher , import);
            //DB-Import
            //Systemimport durchführen
            mExport.finish("XML2DB", sFinisher, null);
            //Email mit Report verschicken
            report.writeReportElement("Status","Vorgang abgeschlossen.");
            report.sendMail(sRecipient);
            //logfile schliessen
            report.writeReportFooter();
    }
```

3.7 Reporting und Logging

Das Reporting erfolgt gemäß den im Konzept besprochenen Richtlinien. Betrachtet werden soll im Speziellen die Umsetzung der Logfile-Generierung.

3.7.1 Logging

Die Umsetzung des Loggings erfolgt, wie im Konzept erwähnt, über eine eigene Klasse, *blReport*, die gemäß dem Singleton-Schema implementiert ist. Sie verwaltet sich komplett selbst, d.h. sie legt sowohl die Reportdatei als auch alle Eintragungen an und verwaltet die

Erstellung ihrer Instanzen. Programmatisch wird das so gelöst, dass ihr Konstruktor nicht *Public* sondern *Private* ist. Da es mit dieser gegebenen Konstruktion nicht möglich wäre, überhaupt eine Instanz der Klasse zu erzeugen, wird eine *public*-Methode *getInstance()* als *static* definiert.

Code

```
public static blReport getInstance()
```

Eine *static*-Methode ist bekanntermaßen eine *Klassenmethode*, die ohne Vorhandensein einer Objektinstanz aufgerufen werden kann. Jedes Objekt, das durch das Logging erfasst werden will, ruft die Methode *getInstance()* auf. Die Methode prüft, ob bereits eine Instanz von *blReport* existiert. Wenn ja, wird diese zurückgegeben, wenn nicht, wird eine neue Instanz angelegt und dann zurückgegeben.

Eine typische, während der Arbeit mit dem SUI-Tool erstellte Logdatei, sieht folgendermaßen aus:

Code

```
<?xml version="1.0" encoding="utf-8"?>
<siniclog date="15.05.2007" time="13:30" source="Control"
xmlns="http://www.sinic.net/SUI/logging">
      <Event>Exporter:Formatheader gesetzt. Name:Ai-2 Typ:0
Beschreibung:Prozess Kunde:False .</Event>
      <Status>Exporter:XML Reader liest Zielformat.</Status>
      <Event>Exporter:XMLDatei
C:\SINIC.NET\Applications\SUI\Formate\Schnittstellen\S KV AI 03.xml wird
zur Deserialisierung geoeffnet.</Event>
      <Status>Exporter:XML Schnittstelle gelesen.</Status>
      <Status>Import:Import aus Quelldatei gestartet.</Status>
      <Event>Import:VER in GUI-Importpanel übernommen.</Event>
      <Event>Import:A01 in GUI-Importpanel übernommen.</Event>
      <Event>Import:A03 in GUI-Importpanel übernommen.</Event>
      <Event>Import:A07 in GUI-Importpanel übernommen.</Event>
      <Event>Import:F02 in GUI-Importpanel übernommen.</Event>
      <Event>Import:F05 in GUI-Importpanel übernommen.</Event>
      <Status>Import:11 Container wurden importiert.</Status>
      <Status>Exporter:Written Elements(P|S|T|D): (1|-1|-1|4) .</Status>
      <Status>Exporter:XML written.15.05.2007 13:30:43 .</Status>
      <Event>Control:Programm beendet.</Event>
</siniclog>
```

Quellcodeansicht: Logdatei

Die Einteilung der geloggten Ereignisse erfolgt in drei Kategorien - Status, Event und Error. Dies vereinfacht die Lesbarkeit und gestattet eine übersichtlichere Formatierung für den Fall, das eine HTML-Ansicht im Browser gewünscht ist. Gesteuert wird das Erzeugen der Einträge im Logfile durch entsprechende Methodenaufrufe in den Klassen, die die Import- und Exportfunktionalitäten übernehmen. Sie sind einheitlich gehalten und verwenden folgende Signatur:

Code

```
public void writeReportElement(string sErzeuger, string sElementTyp,
string sInhalt)
```

Das eigentliche Schreiben übernimmt dann ein *XmlWriter* unter Zuhilfenahme der *XmlWriter.WriteElementString*-Methode, dem die Parameter der *writeReportElemen*-Methode übergeben werden.

3.7.2 Email-Benachrichtigung

Der Service implementiert eine Emailbenachrichtigung. Bei der Umstellung von .NET 1.1 auf .NET 2.0 haben sich einige der für Emails vorgesehenen Klassen geändert. Verwendet werden die .NET 2.0 -konformen Klassen aus dem *System.Net.Mail* – Namensraum. Die Methode *sendMail(string sRecipient)* ist in der *blReport*-Klasse implementiert. Sie erhält als Parameter die Emailadresse des Empfängers, die zuvor aus der XML-Initialisierungsdatei ausgelesen wurde. Es wird zunächst eine Instanz der Klasse *SmtpClient* angelegt, die die Informationen für den Smtp-Versand aufnimmt. Dann werden *MailAddress*-Objekte mit Absender- und Empfängeradressen bestückt und mit diesen ein *MailMessage*-Objekt initialisiert. Zusätzliche Angaben zu weiteren Aspekten des Email-Verkehrs, wie z.B. der Titelzeile, können nun über die Eigenschaften des *MailMessage*-Objekts gesetzt werden. Als eigentliche Nachricht wird über alle eingegangenen Report-Messages iteriert und daraus ein String zusammengesetzt, welcher der *MailMessage.Body* – Eigenschaft übergeben wird. Dieses Objekt wird dann als UTF-8-codierte HTML-Email an den *SmtpClient* übergeben und verschickt.

4 Diskussion

Am Ende der Arbeit angekommen, sollen die Ergebnisse der Aufgabenstellung gegenüber gestellt und diskutiert werden. Ich werde zunächst thematisch geordnet auf die Teilbereiche eingehen, bevor ich eine abschließende Gesamtbewertung vornehme.

4.1 Leistungsumfang

4.1.1 Importformate

Zunächst sollen die behandelten Importformate rekapituliert werden.

4.1.1.1 Textbasierte Formate

Das SUI-Tool ist momentan in der Lage, Textformate zu verarbeiten und die enthaltenen Daten im Sinne der Aufgabenstellung durch alle Prozessschritte hindurch bis in die SINIC Datenbanken zu importieren. Die Software funktioniert in diesem Bereich zuverlässig sowohl für Key:Value-, als auch für CSV- und Fixed Record Length (FRL) -Formate. Die Auswertung dieser Formate wurde durch die Verwendung von regulären Ausdrücken erleichtert, die glücklicherweise von .NET gut unterstützt werden. Ein wenig knifflig war die Behandlung der FRL-Formate. Auch wenn diese Formate konzeptionell zunächst als die einfachsten erscheinen, so ist die Verarbeitung durch das naturgemäß zu implementierende Index-basierende Stringhandling fehleranfällig. Eine Variante der FRL-Formate, die in der ersten Zeile die Feldüberschriften enthält, wird momentan noch nicht gesondert behandelt. Eine entsprechende Option zum Überspringen der ersten Zeile ist jedoch vorgesehen und auch relativ einfach in GUI und Backend zu implementieren.

4.1.1.2 Excel

Auch Excel-Dateien lassen sich in vollem Umfang verarbeiten. Es machte sich an dieser Stelle bezahlt, dass als Strategie auf die Microsoft Primary Interop Assemblies gesetzt wurde. Der Zugriff auf die enthaltenen Daten wird so wesentlich vereinfacht.

Ein theoretischer Schwachpunkt ist, dass zur Weiterverarbeitung das erste in der Arbeitsmappe enthaltene Tabellenblatt herangezogen wird. Es wäre denkbar, dass ein Importformat auftaucht, das die Daten auf einem anderen Tabellenblatt vorhält oder über mehrere Blätter verteilt. In diesem Fall wäre es nötig, die Bearbeitung nach Erkennen der umfangreicheren Arbeitsmappenstruktur anzuhalten und über einen Auswahldialog ein bestimmtes Arbeitsblatt oder einen bestimmten Bereich auswählen zu lassen. Das grundlegende Verfahren zur weiteren Bearbeitung würde sich dadurch allerdings nicht ändern. Inwieweit dieser Fall in der Praxis zum Tragen kommt, bleibt abzuwarten. Im SINIC-Umfeld ist er bisher nicht aufgetreten.

4.1.1.3 Access

Die Access-Implementierung ist für den Importteil fertig gestellt. Auch Zuordnungen können vorgenommen werden. Noch nicht umgesetzt ist die weiterführende Verarbeitung. Die Frage, die sich hier stellt, ist, wie generell weiter verfahren werden soll. Da das Quellformat aus einer Datenbank besteht und das Ziel des Systemimports ebenfalls Datenbanken sind, ist der Zwischenschritt in das SINIC-Format eventuell nicht effizient. Denkbar wäre eine direkte Datenübertragung von Quell- zu Zieldatenbank mittels entsprechender SQL-Kommandos. Eine entsprechende Anwendung ist bei SINIC bereits vorhanden; diese ist jedoch für eine bestimmte Formatumwandlung konzipiert. Beide Varianten hätten ihre Daseinsberechtigung. Für die Einhaltung des Weges, der für die anderen Formate gewählt wurde, spricht die einheitliche Handhabung des SINIC-Formats für die nachfolgenden Prozessschritte. Ebenfalls müsste der Windows-Service nicht erweitert werden, um diesen Spezialfall abzudecken. Auf der Gegenseite fallen Performancegedanken ins Gewicht, da ein direkter Datentransfer schneller durchgeführt werden kann. Beide Möglichkeiten sind weiter verfolgbar und können anhand der bestehenden Situation abgewogen werden.

4.1.1.4 XML

Ziel war es, die Grundlagen für eine XML-Verarbeitung zu schaffen und die Möglichkeiten für eine universelle Verarbeitung von XML-Importdaten zu prüfen. Der erste Punkt ist erfüllt worden. XML wird gelesen und entsprechend der inhärenten Baumstruktur dargestellt. Auch eine Auswahl der einzelnen Nodes ist möglich, um die Elemente

Zielfeldern zuordnen zu können. Probleme macht hingegen die weitere Verarbeitung bzw. ein universeller Ansatz zur Behandlung der XML-Formate.

Das erste Problem, das auftritt, ist die Form der Darstellung. Kommerzielle Tools, wie z.B. XMLSpy von Altova, analysieren die Struktur eines XML-Dokumentes und erzeugen daraus eine komplexe Baumstruktur, die alle Elemente mit ihren Eigenschaften und Attributen darstellt und vielfältige Möglichkeiten zur Manipulation dieser Struktur bietet. Eine solche Umsetzung ist im Rahmen einer Anwendung, die nicht primär für den Import von XML-Daten vorgesehen ist, vom Aufwand her kaum zu rechtfertigen. Die vereinfachte Darstellung in einer *TreeView*-Control birgt allerdings ebenfalls Probleme. Da ein XML-Dokument n Ebenen tief verschachtelte Elemente enthalten kann, von denen jedes m Attribute enthalten kann, n und m aber im Sinne der Darstellung nicht nach oben begrenzt sind, muss die Übersicht zwangsweise leiden, wenn n oder m gegen große Werte gehen. Die gewählte Darstellungsform ist diesbezüglich nicht ideal und als Kompromiss zu sehen. Ein weiteres Problem bei sehr großen XML-Dokumenten ist der rekursive Algorithmus zum Navigieren des XML-Baums. Bei komplexen, tiefen Strukturen steigt sowohl der Speicherplatzverbrauch als auch die Bearbeitungsdauer stark an, was eine Limitierung auf die Zahl maximal auszuwertender XML-Nodes nahelegt. Je nach Struktur der zu lesenden Datei können aber durch eine Limitierung der Nodes Informationen verloren gehen. Als Ausweg aus dieser Situation bieten sich zwei Möglichkeiten an. Zum einen könnte man den rekursiven Algorithmus durch einen iterativen ersetzen. Dies würde zwar in einem sperrigeren Code resultieren, sollte aber die o.g. Performanceprobleme lösen. Es wäre ebenfalls möglich, von einem universellen Handling der XML-Dateien ein Stück weit Abstand zu nehmen und eine Spezialisierung auf in der Branche verwendete Formate vorzunehmen. Dies würde zum einen den Vorteil bieten, eine bessere graphische Darstellung für die jeweiligen XML-Dokumente realisieren zu können, und zum anderen eine optimierte Weiterverarbeitung ermöglichen. Andererseits würde die Beschreitung des zweiten Weges eine gewisse konzeptionelle Abkehr vom Grundgedanken der universellen Schnittstelle bedeuten.

4.1.1.5 Generelle Anmerkungen zum Import

Die Import-Container haben sich bewährt. Durch die Entkopplung der GUI von den Datenstrukturen der Quellformate ist eine einheitliche Darstellung der verschiedenen Formate möglich, ohne dass größere Anpassungen nötig sind. Eine grundsätzliche Unterteilung findet nur in zwei große Kategorien statt. Entweder sind die Importdaten besser in einer Tabellenstruktur dargestellt, oder in einer Baumstruktur. Problematisch waren bei der Entwicklung die grundlegenden Importroutinen zum Lesen der Dateien. Nachdem diese Probleme überwunden waren, ließen sich zusätzliche Formate relativ einfach hinzufügen.

4.1.2 Zielformate

Drei Zielformate sind bisher implementiert: der Artikelimport, der Lieferantenimport und der Prüfplanimport. Dies sind natürlich nicht alle bei SINIC vorkommenden Formate. Sie sind aber zum einen durchaus gängig und zum anderen demonstrieren sie, dass das implementierte System in dieser Form funktioniert. Während der Lieferantenimport ein Format mit nur einer Ebene darstellt, existieren sowohl bei Artikel- als auch Lieferantenimport zwei verschachtelte Datenebenen. Der Artikelimport war der Import, der als Musterimport als erstes implementiert worden ist. Nachdem dieser funktionierte, wurde die Software auf das jetzt vorliegende Modell einer dynamisch auslesbaren Zielformatssystematik umgestellt. Als ergänzende, manuelle Tätigkeit sind nur wenige Konstanten anzulegen, die den Namen des neuen Formats und einige formatspezifische Felder angeben. Die restlichen Informationen werden zur Laufzeit aus den XmlDBSchemas ausgelesen. Um zu testen, ob dieses System funktioniert, wurde zunächst der Lieferantenimport unter dem neuen System hinzugefügt. Es zeigte sich hierbei, dass an einigen Stellen die Systematik noch nicht vollständig durchdacht war, was zu zwei Änderungen führte. Zum einen wurden die einzugebenden Konstanten erweitert, und zum anderen Stellen, an denen noch Code vorhanden war, der spezifisch auf den Artikelimport zugeschnitten war, durch allgemeineren Code ersetzt. Weitere Testings ergaben, dass die neu ergänzten Konstanten im aktuellen Code prinzipiell nicht mehr nötig sind. Nichtsdestotrotz werden sie als Fallback für den Fall beibehalten, dass Probleme mit den XmlDBSchemas auftreten und eine schnelle Notfalllösung bis zur Behebung des Problems

vorhanden sein muss. Nachdem der Lieferantenimport erfolgreich im neuen System funktionierte, wurde der kompliziertere Prüfplanimport ergänzt. Hier zeigte sich, dass das System den Anforderungen gerecht wird. Die Integration des Prüfplanimports ließ sich ohne größere Probleme in einem Zeitraum von weniger als zwei Stunden, incl. Testing, realisieren. An dieser Stelle trat ein unerwartetes Problem auf: der Artikelimport funktionierte nicht mehr. Trotz umfangreichem Testing gelang es zunächst nicht, das Problem zu identifizieren. Schlussendlich konnte der Fehler an den XmlDBSchemas festgemacht werden. Ein falscher Eintrag sorgte für eine falsche Zuordnung der Keys im Datenbanklayer des SINIC-Frameworks. Dieses Ereignis zeigt zugleich eine Stärke und eine Schwäche der Software. Auf der positiven Seite läuft die Software wie gefordert in engem Zusammenspiel mit den XmlDBSchemas. Auf der negativen Seite bedeutet das aber auch, dass es für eine reibungslose Funktion der Software absolut essentiell ist, dass die Inhalte dieser Dateien valide sind. Dies ist vor Implementierung neuer Formate auf jeden Fall ausgiebig zu testen. In besonderem Maße gilt dies für die älteren Formate, bei denen dies nicht als gegeben anzunehmen ist. Sie existierten bereits vor Generierung der Schemas und verwenden sie nur zum Teil oder in manchen Fällen noch gar nicht. Die neueren Formate sind in der Regel gut getestet, und die Angaben in den Schemas funktionieren zuverlässig.

4.1.3 Mapping

In dem Moment, in dem der User die Mappings vornimmt, ist die Hauptarbeit im Hintergrund bereits passiert. Die Zuordnung erfolgt anhand der abstrakten Container und ist von der Datenebene losgelöst. Das eigentliche Matching im Import und Export, und damit die Umsetzung des erfolgten Mappings, wurde im Implementierungsteil bereits beschrieben und soll hier nicht wiederholt werden. Interessanter sind die Möglichkeiten des vom User durchführbaren Mappings, bzw. deren Limitationen. Im jetzigen Stand der Software ist eine reine Feldzuordnung ohne optionale Möglichkeiten implementiert. Dies reicht in den meisten Fällen aus, lässt aber Wünsche an die Flexibilität offen. Während der Entwicklung war im Gespräch, dass der User seine Feldzuordnungen auf verschiedene Weisen modifizieren können sollte. So sollte z.B. eine Zuordnung der Art „Feld A“ + Feld B“ => Feld C möglich sein. Eine Zuordnung dieser Art ist zur Zeit nicht möglich. SINIC verwendet in einem ihrer Module bereits eine Komponente, die ähnliche Funktionalitäten

implementiert. Dieser „Formeleditor“ könnte, sofern er sich in die Anwendung integrieren lässt, diese Funktionen übernehmen. Dies hätte zwangsweise Auswirkungen sowohl auf die GUI, als auch auf das Backend. Durch die Architektur der Anwendung und die Entkopplung des Mappings von den Quell- und Zieldaten wären allerdings keine Änderungen beim Verarbeiten auf der Dateiebene nötig. Lediglich die Verfahren zum Erzeugen des SINIC-Formats müssten die zusätzlichen Funktionalitäten berücksichtigen. Im Hinblick auf diese Gegebenheit erscheint eine Integration durchaus machbar zu sein, ohne einen unverhältnismäßigen Mehraufwand im *Business Layer* zu erzeugen.

Das Erstellen eines FRL-Formates „on-the-fly“ ist eine der auffälligeren Möglichkeiten, die dem User beim Arbeiten mit der Software zur Verfügung steht. Es stellt einen Teilbereich des Mappings dar, der nur die Importseite betrifft, und ist in dieser Form ein praktisches Hilfsmittel. Trotzdem soll ein Nachteil nicht unerwähnt bleiben. Das Fehlen einer „Undo“-Funktionalität, die den User seine letzte Zuordnung noch einmal revidieren lässt. Da die Daten alle vorliegen, wäre es möglich, eine solche Funktion umzusetzen; es wurde jedoch aus Zeitgründen zunächst darauf verzichtet.

In der Gesamtbetrachtung erfüllt das realisierte Mapping seinen Zweck und ist durch die Entkopplung von der Datenebene gut erweiterbar. Dies ist auszunutzen, um einen höheren Grad der Flexibilität bei den Zuordnungen zu erreichen.

4.2 GUI

Ziel der GUI war es, eine komfortable und übersichtliche Arbeitsoberfläche für den Benutzer zu schaffen, die intuitiv bedienbar ist. Dieses Ziel wurde weitestgehend erreicht. Abweichend vom Grundkonzept wurde eine eigene Oberfläche zur Verwaltung der bereits angelegten Schnittstellen angelegt. Die verwendeten Controls sind fest im SINIC-Framework verankert und lassen von daher nur geringfügige optische Anpassungen zu, was allerdings für die Listendarstellung der vorhandenen Schnittstellen kein Nachteil ist. Ein weiterer Vorteil ist, dass durch den Aufbau Listenmaske -> Detailmaske ein Wiedererkennungswert zu anderen SINIC-Modulen gegeben ist, was die Orientierung erleichtert. Zudem vereinfacht es den Betrieb für den Fall, dass mehrere User mit dem

System arbeiten. Diese können sich auf einen Blick einen Überblick über die angelegten Schnittstellen verschaffen, ohne sich anhand von Dateinamen orientieren zu müssen.

Was die Form betrifft, die den Arbeitsbereich für die Zuordnungen darstellt, so sind die Bereiche Import, Export und Zuordnungen durch den Aufbau und die farbliche Absetzung gut voneinander zu unterscheiden. Zuordnungen werden über Mausklick getroffen und wieder gelöscht. Das ganze Userinterface basiert auf dem Prinzip „Point and Click“. Dies ist sicherlich eine Geschmacksfrage. Viele Funktionen hätten auch per rechte Maustaste, Keyboard Shortcut oder Drag&Drop gelöst werden können. Es wäre eine reine Zeitfrage, diese Funktionalitäten noch zusätzlich zu implementieren. Insgesamt ist die Oberfläche schnörkellos und funktional, was ihrem zugedachten Einsatzgebiet gerecht wird und den im Einleitungssatz genannten Zielen entspricht.

Nichtsdestotrotz sind auch einige Punkte feststellbar, die noch zu verbessern wären. Zum einen ist die Darstellung von XML-Dateien nicht optimal gelöst, was bereits erläutert worden ist. Zum anderen ist der verfügbare Platz für die Darstellung der Zielformatfelder nicht sehr groß. Selbst wenn die *DataGridViews* sich entsprechend der verfügbaren Ebenen vergrößern oder verkleinern, so ist der Platz doch nicht sehr üppig. Dies tritt vor allem bei Formaten mit sehr vielen Feldern, wie z.B. dem Prüfplanimport, zu Tage. Die momentane Lösung ist zwar funktional voll ausreichend, aber die doch sehr langen Scrollbalken, die entstehen können, sind sowohl optisch als auch von der Useability her nicht optimal. Hier wäre grundsätzlich über eine andere Darstellungsform nachzudenken. Eine mögliche Lösung könnte sein, die verschiedenen Zielformatebenen als komplett eigenständige Windows-Forms zu realisieren, die im Stil von Anwendungen, wie z.B. Adobe Photoshop, frei positionierbar und aneinander andockbar sind. Fraglich ist jedoch, ob der ästhetische Mehrwert und ein kleines Plus an Useability den dafür nötigen Mehraufwand rechtfertigt.

4.2.1 Useability

Es ist davon auszugehen, dass ein ungeschulter User mit durchschnittlichen Kenntnissen der PC-Benutzung in kurzer Zeit effizient mit dem SUI-Tool arbeiten kann. Die Oberfläche ist relativ minimalistisch und unterstützt den User durch Ein- und Ausblendung relevanter Schaltflächen je nach Kontext und der Anzeige von situationsabhängigen Hilfetexten. Ein

relativ niedrig zu priorisierendes „nice to have“-Feature wäre die Ermöglichung von Drag&Drop bei der Zuordnung von Quell- zu Zielfeldern. Auch wenn nach meiner persönlichen Einschätzung aus über zehn Jahren Erfahrung als PC-Trainer das Drag&Drop-Verfahren bei ungeübten Usern einen Quell großer Verwirrung darstellen kann, so wäre es doch in diesem speziellen Fall ein durchaus legitimes, da intuitives Verfahren, über das man nachdenken könnte.

4.3 Logging und Reporting

Der Komplex Logging und Reporting funktioniert zuverlässig und ist eine relativ elegante Umsetzung, die durch die Implementierung als Singleton nur dann in Schwierigkeiten gerät, wenn das SUI-Tool auf Multithreading-Betrieb umgestellt werden sollte. Da es dafür keine Veranlassung gibt, ist das Gefahrenpotenzial diesbezüglich gering. Ein nettes Feature, das bisher aus Zeitgründen nicht implementiert wurde, aber angedacht ist, ist das userseitige Setzen von „Log-Leveln“. Diese Log-Level bestimmen den Umfang des Loggings und können nach Wunsch des Users angepasst werden. Denkbar wäre eine Einteilung in drei Stufen, die von ausführlich bis zur Kurzzusammenfassung der wichtigsten Ereignisse reicht.

4.4 Erweiterbarkeit und Entwicklungspotenzial

Die Anwendung bietet momentan einiges an Entwicklungspotenzial. Durch die offene Struktur ist eine gute Erweiterbarkeit gegeben. An dieser Stelle sollen nur Entwicklungsmöglichkeiten dargestellt werden, die ohne größere konzeptionelle und technische Vorbereitung umgesetzt werden könnten.

4.4.1 Neue Formate

Zunächst könnte über neue Importformate nachgedacht werden.

4.4.1.1 QDX und QML

Interessant ist das XML-Format QDX (Quality Data eXchange), das vom Verband der deutschen Automobilindustrie e.V., Quality Management Center entwickelt worden ist[35]. Laut Aussage der Entwickler ist QDX der „Standard für die Beschreibung und den Austausch von Qualitätsdaten zwischen Geschäftspartnern in der Automobilindustrie“. Namhafte Vertreter unter den Automobilherstellern und Zulieferfirmen (BMW, Volkswagen, Bosch ...) haben sich diesem Standard verschrieben und auch die Mutterfirma von SINIC, die IBS AG, findet sich unter den Förderern. Bei Erstellung der Arbeit waren allerdings viele wichtige SINIC-Formate nicht durch QDX Dokumenttypen erfasst, so dass eine Integration zunächst zurückgestellt wurde.

QML (Quality Markup Language)[36] ist ein Format der Firma Q-DAS. Es stellt eine Alternative zu dem ebenfalls von dieser Firma vertriebenen DFD/DFQ-Format, welches als Q-DAS ASCII Transfer -Format bekannt ist, dar. Auch wenn der Bereich CAQ-Systeme weit von einem einheitlich akzeptierten und verwendeten Datenformat entfernt ist, so hat das Q-DAS ASCII doch eine gewisse Verbreitung. Es ist anzunehmen, dass die XML-Version dieses Formates zunehmend an Bedeutung gewinnt, von daher war das SUI-Tool auf eine Verarbeitung von QML vorzubereiten. Die aktuelle XML-Import-Implementation wurde aus diesem Grund mit einer QML-Datei getestet.

4.4.1.2 Microsoft Word

Die Textverarbeitung Word ist sicher nicht unter den üblichen Verdächtigen, was das Thema Datentransferformate betrifft. Es gibt aber eine ganze Reihe SINIC-Kunden, die Teile ihrer Daten als Inhalte von Word-Formularen erfassen[37]. Diese Felder können, sofern sie einen eindeutigen Namen erhalten haben, programmatisch mit .NET ausgewertet werden. Somit fallen Word-Formulare in den Kreis der wahrscheinlicheren Formate, die in Zukunft mit dem SUI-Tool erfasst und bearbeiten werden könnten.

[35] http://www.vda-qmc.de/de/index.php?catalog=1101
[36] http://www.qml-org.com/
[37] Interne Kommunikation, Herr J. Schneider, Entwicklungsleiter

Was die Zielformate betrifft, so sind nach momentanem Wissensstand alle weiteren, relevanten SINIC-Formate durch das SUI-Tool verarbeitbar. Diese sollen hier nicht im Einzelnen aufgeführt werden. Die Praxis wird zeigen, inwieweit einzelne Formate den Weg in das SUI-Tool finden werden.

4.4.2 Frontend 2.0

Das momentane Frontend reizt den verfügbaren Bildschirmplatz fast voll aus. Man könnte daher über eine andere Darstellungsform im Bereich der entwickelten Kopfmaske nachdenken. Während der Entwicklungen sind einige Ideen hinzugekommen, die zum Teil auch schon oben dargestellt worden sind. Ein neues Layout macht sicherlich erst dann Sinn, wenn die Anzahl der Zielformate gesteigert wurde und sich abzeichnet, wohin der Trend im Bereich XML geht. Da die Kopfmaske relativ locker an den *Business Layer* gekoppelt ist, gäbe es in diesem Bereich keine unüberwindlichen Schwierigkeiten.

4.5 Gesamtbewertung

Die Implementierung der Software war eine interessante und abwechslungsreiche Aufgabe. Das Resultat soll nun noch einmal im Gesamtkontext kritisch beleuchtet werden.

Wie bereits an einigen Stellen der vorhergegangenen Bewertung zu Tage getreten ist, sind viele kleinere Funktionen noch nicht umgesetzt, obwohl der benötigte Zeitraum dafür als relativ gering angegeben wird. An dieser Stelle tritt der Charakter einer Grundsatzarbeit klar zu Tage. Die entwickelte Software baut nicht auf etwas bestehenden auf, sondern stellt eine echte Neuentwicklung dar. Ein beträchtlicher Teil der zur Verfügung stehenden Zeit musste daher für eine Einarbeitung in die relevanten Technologien verwendet werden. Besonders zu nennen sind an dieser Stelle das SINIC-Framework, das .NET-Framework und die feineren Punkte der Sprache C#. Auch der XML-Komplex mit den dazugehörigen Elementen, die unmittelbar relevant waren, also Xml, Xml Schema und Xpath, benötigten eine gewisse Einarbeitungszeit. Ebenfalls nicht zu unterschätzen ist die Entwicklung eines Gefühls für die Anforderungen und Besonderheiten, die die CAQ-Branche mit sich bringt.

Betrachtet vor diesem Kontext wurde mit der Software die vorgegebene Zielsetzung erreicht. Es ist ein System entstanden, das die grundlegenden Erfordernisse funktional zuverlässig erfüllt und den Boden für Weiterentwicklungen bereitet. Das Entstehen der Software war, bezogen auf Zielsetzung und exakter Ausgestaltung, ein iterativer Prozess. Ideen mussten verworfen, neu entwickelt und einem Realitätscheck unterzogen werden. Dies war oft nur dadurch möglich, etwas anzuprogrammieren , auch auf die Gefahr hin, dass sich der gewählte Weg als Sackgasse entpuppte. Aus dieser Entwicklungshistorie heraus resultiert, dass die Software an einigen Stellen kleine Ecken und Kanten aufweist. Diese sind aber keinesfalls kritisch und betreffen weniger den User als das interne Handling mancher Probleme.

Da die Basis gelegt ist, sind Änderungen und Erweiterungen nun zwangsläufig schneller umsetzbar als zu Beginn, als das Grundgerüst für die Anwendung noch nicht verankert war. Gegen Ende des Projekts wurde dem Testing bestehender Funktionen der Vorrang gegenüber dem Hinzufügen neuer Features gegeben. Als Endergebnis steht eine stabile Plattform für vielfältige, weitere Entwicklung.

5 Zusammenfassung und Ausblick

Fasst man die Punkte der Diskussion zusammen, so kommt man zu dem Ergebnis, dass die entstandene Software ein funktionales, ausbaufähiges Tool mit ansprechender, leicht zu bedienender Benutzeroberfläche ist. Die der Software zugrunde liegende Architektur hat sich als robust erwiesen, ebenso wie die entwickelten XML-Formate.

Wie bereits festgestellt, lässt sich das SUI-Tool in vielfältige Richtungen weiterentwickeln. Hier sollen zum Abschluss einige der interessanteren Möglichkeiten betrachtet werden, die über die o.g. Möglichkeiten hinausgehen und das SUI-Tool in umfangreicherem Maße ergänzen bzw. erweitern könnten.

5.1.1 Webschnittstelle

Die momentane Umsetzung des automatisierten Systemimports als Windows-Service trägt dem Umstand Rechnung, dass die SINIC-Systeme lokale Installationen beim Kunden sind.

Sowohl Datenbanken als auch die Systeme, die dem User zur Verfügung stehen, liegen in den meisten Fällen auf dem gleichen Computer bzw. im gleichen Netzwerk. Ein alternatives Angebot seitens SINIC, wie z.B. die Option, dass Datenbanken und Systeme bei SINIC für die Kunden gehosted werden, könnte eine andere Struktur nahe liegen. Es wäre denkbar, beim Kunden eine Anwendung zu installieren, die zwar die Bedienerführung und Logik lokal vorhält, deren Datenhaltung aber bei SINIC stattfindet. Die Kommunikation und der Datentransfer könnten so vonstatten gehen, dass ein Webserver eine Webschnittstelle implementiert, die durch die Anwendung beim User angesprochen werden kann. Da die relevanten Daten, die beim automatischen Systemimport des SUI-Tools generiert und verwendet werden, bereits im internet-freundlichen XML-Format vorliegen, würde es sich anbieten, diese per Webserver weiter zu bearbeiten. Methodenaufrufe könnten angepasst und als SOAP oder RPC realisiert werden. Es wäre zu untersuchen, inwieweit man Anwendungsteile beim User belässt oder aus Gründen der einfacheren Wartbarkeit auf den Server verlagert.

In elementarer Ausprägung könnte ein Internetformular erzeugt werden, das eine Auswahl der Quelldatei und der Schnittstellendatei zulässt. Es müssten dann lediglich die Daten übertragen und den Im- und Exportroutinen zugeführt werden.

Dies wäre allerdings durch ein Sicherheitskonzept abzusichern und es stellen sich auch besondere Anforderungen an die Infrastruktur. Serverseitige Erreichbarkeit wäre ebenso zu gewährleisten wie Ausfallsicherheit und ein belastbares Backupkonzept.

Dies kann im Bereich Hardware und Lizenzen zu deutlichen Kosten führen, und auch entsprechendes Know-how für eine solche Infrastruktur muss erst gebildet bzw. eingekauft werden. Andererseits könnten Pflege- und Supportkosten minimiert werden, nicht zuletzt durch Verwendung einer kundenunabhängigen, einheitlichen Datenbankstruktur im Backend, auf die direkt zugegriffen werden kann.

Eine Beschreitung dieses Weges unter Berücksichtigung einer Kosten-/Nutzen-Abwägung wäre daher zunächst vorrangig eine betriebswirtschaftliche Entscheidung.

5.1.2 XSLT-Generator

Ein Faktor, der bei der Implementierung eine Rolle gespielt hat, war die Frage nach einer universellen, adaptiven Verarbeitung von XML-Daten. Verschiedene Möglichkeiten für eine Umsetzung wären denkbar, die Stoff für weitere Untersuchungen bilden könnten. Eine der interessanteren Varianten wäre die Entwicklung eines XSLT[38]-Generators. XSLT ist eine XML-Anwendung, die Regeln festlegt, anhand derer ein XML-Dokument in ein anderes XML-Dokument transformiert wird.[39]Das Matching von XML-Tags und die Umwandlung in eine andere XML-Struktur ließe sich zwar auch programmatisch analog zu einer normalen Textumwandlung durchführen, aber da XSLT ein spezialisiertes Werkzeug zur XML-Transformation ist (wie der Name schon sagt), wäre es interessant zu untersuchen, wie man dynamisch erstellte XSLT-Stylesheets in den SUI-Tool-Kontext integrieren könnte. Die Grundidee ist, zunächst den Benutzer das XML-Quellelement bzw. Attribut des Ursprungsformats auswählen und dem Zielfeld zuordnen zu lassen. Dies ist im SUI-Tool auf Elementebene bereits möglich.

Die Herausforderung wäre nun, zu untersuchen, ob mit dieser Angabe und der Kenntnis des Quell- und des Zielformats eine XSLT-Datei generiert werden kann, die allen Vorgaben einer gültigen, wohlgeformten XML-Datei entspricht und in Verbindung mit einem XSLT-Prozessor wie z.B. SAXON[40] aus der Quelldatei ohne weiteres programmatisches Handling innerhalb des SUI-Tools eine gültige SINIC-Format-Datei erzeugt. Im Hinblick auf die zwar stark strukturierten aber im Aufbau sehr große Gestaltungsfreiheit zulassenden XML-Dokumente, ist dies sicher keine triviale Aufgabe. Ein Vorteil wäre, dass man bei Erfolg nur eine sehr schlanke Anwendung für die eigentliche Formatkonvertierung benötigen würde. Prinzipiell genügte der XSLT-Prozessor, der die bezogene Quelldatei, XSLT-Schema und Zieldatei als Kommandozeilenoption übergeben bekommt. In diesem Fall wäre eine kleine Anwendung, die den Output des Prozessors auf Fehler oder ähnliches untersucht, bereits genug. Auch ein minimalistisches Webformular, aus dem die Kommandozeilenparameter generiert werden, wäre denkbar.

[38] Kurz für: XSL Transformations
[39] XMLiaN, S. 146
[40] http://saxon.sourceforge.net/

Eine solche Erweiterung wäre umso mehr interessant, da zu erwarten ist, das XML-Formate eine immer stärkere Verbreitung im Bereich des elektronischen Datenaustauschs erlangen werden.

5.1.3 SAP-Connector

Dieser letzte Punkt soll nur kurz beleuchtet werden. Software-Riese SAP[41] durchdringt fast jeden Bereich der korporativen Softwarelandschaft. Vielfältige Module von der Automatisierung, Prozessoptimierung, Produktionssteuerung bis zum HR-Management existieren und erfreuen sich großer Beliebtheit. SINIC-Kunden, die SAP-Systeme verwenden, erzeugen momentan Excel-Tabellen, die die für SINIC relevanten Daten enthalten. Diese werden vom SUI-Tool entsprechend behandelt und umgewandelt. Da es sich bei SAP um eine sehr starke Marke handelt, wäre es eventuell interessant, eine Direktanbindung ohne den Zwischenschritt Excel-Tabelle zu realisieren. Hierzu müsste eine Einarbeitung in die SAP-Systeme und bezogene Schnittstellen erfolgen. um die Machbarkeit in Relation zum Aufwand zu prüfen.

[41] http://www.sap.de

ANHANG

A) Literaturverzeichnis

Print

- Head First Design Patterns; Eric Freeman, Elisabeth Freeman , Kathy Sierra; O'Reilly Media - Beijing, Cambridge, Köln, Paris, Sebastopol, Taipei, Tokyo; First Edition October 2004;
  zitiert: HFDP S. ..
- XML in a Nutshell. Deutsche Ausgabe; Elliotte R. Harold, W. Scott Means; O'Reilly, 3. Auflage 2005;
  zitiert: XMLiaN S. ..
- Programming C#; Jesse Liberty; O'Reilly Media - Beijing, Cambridge, Köln, Paris, Sebastopol, Taipei, Tokyo; 4. Edition February 2005;
  zitiert. Programming C# Seite ..
- Deutsches Institut für Normung, e.V.;
  zitiert: DIN - ISO xxxx

Digital

- Microsoft Developer's Network Library, zitiert: MSDN, Seitentitel
  - Microsoft .NET Framework
    https://www.microsoft.com/germany/msdn/library/net/MicrosoftNETFramework.mspx?mfr=true
  - Primary Interop Assemblies – Chris Kunicki, OfficeZealot.com, October 2002
    http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dno2k3ta/html/OfficePrimaryInteropAssembliesFAQ.asp
  - HOW TO: Specify Namespaces When You Use an XmlDocument to Execute XPath Queries in Visual C# .NET
    http://support.microsoft.com/kb/318545/en-us
  - Steuern der XML-Serialisierung mit Attributen
    http://msdn2.microsoft.com/de-de/library/2baksw0z(VS.80).aspx

- Improving .NET Application Performance and Scalability: Chapter 9, Improving XML Performance; J.D. Meier, Srinath Vasireddy, Ashish Babbar, and Alex Mackman ;Microsoft Corporation; May 2004
  http://msdn2.microsoft.com/en-us/library/ms998559.aspx

- C# Interface Based Development; Matthew Cochran ; March , 2006,
  zitiert: C# Interface Based Development
  http://www.c-sharpcorner.com/UploadFile/rmcochran/csharp_interrfaces03052006095933AM/csharp_interrfaces.aspx?ArticleID=cd6a6952-530a-4250-a6d7-54717ef3b345

B) Schema-Beschreibung für den Dokumenttyp SINICSchnittstelle

Code

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns="http://www.sinic.de/SUI/Schnittstelle"
xmlns:xs="http://www.w3.org/2001/XMLSchema" targetNamespace="http://www.sinic.de/SUI/Schnittstelle"
elementFormDefault="qualified" attributeFormDefault="qualified">
	<xs:element name="SINICSchnittstelle">
		<xs:complexType>
			<xs:sequence>
				<xs:element name="ZielDokumentTyp" type="xs:string"/>
				<xs:element name="QuellDokumentTyp" type="xs:string"/>
				<xs:element name="Separator" type="xs:string" minOccurs="0"/>
				<xs:element name="Zuordnungsname" type="xs:string"/>
				<xs:element name="Description" type="xs:string"/>
				<xs:element name="FreigabeStatus" type="xs:string"/>
				<xs:element name="Zuordnungen">
					<xs:complexType>
						<xs:sequence>
							<xs:element name="Zuordnung"
maxOccurs="unbounded">
								<xs:complexType>
									<xs:sequence>
										<xs:element
name="Level" type="xs:string"/>
										<xs:element
name="QuellFeld" type="xs:string"/>
										<xs:element
name="ZielFeld" type="xs:string"/>
									</xs:sequence>
									<xs:attribute name="Start"
type="xs:string" form="unqualified"/>
									<xs:attribute
name="Laenge" type="xs:string" form="unqualified"/>
								</xs:complexType>
							</xs:element>
						</xs:sequence>
					</xs:complexType>
				</xs:element>
			</xs:sequence>
		</xs:complexType>
	</xs:element>
</xs:schema>
```

Datei: SinicSchnittstelle.xsd

C) Schema-Beschreibung für den Dokumenttyp SinicDoc

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
	<xs:element name="Version" type="xs:string"/>
	<xs:element name="Value">
		<xs:complexType>
			<xs:simpleContent>
				<xs:extension base="xs:string">
					<xs:attribute name="colID" type="xs:string" use="required"/>
				</xs:extension>
			</xs:simpleContent>
		</xs:complexType>
	</xs:element>
	<xs:element name="SinicDoc">
		<xs:complexType>
			<xs:sequence>
				<xs:element ref="Head"/>
				<xs:element ref="Data"/>
			</xs:sequence>
		</xs:complexType>
	</xs:element>
	<xs:element name="SinicDataSubSet">
		<xs:complexType>
			<xs:sequence>
				<xs:element ref="Value" maxOccurs="unbounded"/>
			</xs:sequence>
			<xs:attribute name="tableID" type="xs:string" use="required"/>
			<xs:attribute name="refcol" type="xs:string" use="required"/>
			<xs:attribute name="mastercol" type="xs:string" use="required"/>
		</xs:complexType>
	</xs:element>
	<xs:element name="SinicDataSet">
		<xs:complexType>
			<xs:attribute name="tableID" type="xs:string" use="required"/>
			<xs:attribute name="pKey0" type="xs:string" use="required"/>
			<xs:anyAttribute processContents="skip"/> <!--erforderlich, da mehrere Keys
folgen können-->
		</xs:complexType>
	</xs:element>
	<xs:element name="Head">
		<xs:complexType>
			<xs:sequence>
				<xs:element ref="DataType"/>
				<xs:element ref="Version"/>
				<xs:element ref="DB"/>
			</xs:sequence>
		</xs:complexType>
	</xs:element>
	<xs:element name="DataType" type="xs:string"/>
	<xs:element name="Data">
		<xs:complexType>
			<xs:sequence>
				<xs:element ref="SinicDataSet" maxOccurs="unbounded"/>
				<xs:sequence>
					<xs:element ref="Value" maxOccurs="unbounded"/>
					<xs:element ref="SinicDataSubSet" minOccurs="0"
maxOccurs="unbounded"/>
```

```
                                    </xs:sequence>
                                </xs:sequence>
                            </xs:complexType>
                        </xs:element>
                        <xs:element name="DB">
                            <xs:complexType>
                                <xs:simpleContent>
                                    <xs:extension base="xs:string">
                                        <xs:attribute name="id" type="xs:string" use="required"/>
                                        <xs:attribute name="dbType" type="xs:string" use="required"/>
                                    </xs:extension>
                                </xs:simpleContent>
                            </xs:complexType>
                        </xs:element>
</xs:schema>
```

Datei: SinicDoc.xsd

D) CD-Inhalt

Pfad	Inhalt
SUI-Tool	Start-Verzeichnis des SUI-Tools
SUI-Tool\BusinessLayer	C#-Dateien - BusinessLayer
SUI-Tool\Forms	C#-Dateien - GUI
SUI-Tool\Formate	SINIC-Formatbeschreibungen
SUI-Tool\Formate\Testdaten	Importformate, mit denen getestet wurde.
SUI-Tool\Formate\Export	XML-Output,SinicDoc-Dokumenttyp
SUI-Tool\Formate\Schnittstellen	XML-Output,SINICSchnittstelle-Dokumenttyp
SUI-Tool\logs	Generierte Logfiles.
SinicUIServ	Startverzeichnis des UI-Service
SinicUIServ\ BusinessLayer	C#-Dateien
SinicUIServ\ logs	Generierte Logfiles
SinicUIServ\ Data	Testdateien und ini.xml
XML	SinicSchnittstelle.xsd, SinicDoc.xsd, QSISTM.xml
Grafiken und Screenshots	Verwendete Grafiken