JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

Technisch-Naturwissenschaftliche
Fakultät

Zerlegung eines monolithischen Softwaresystems in ein Plug-In-basiertes Komponentensystem

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplomingenieur

im Masterstudium

Informatik

Eingereicht von:
Christian Mittermair

Angefertigt am:
Institut für Systemsoftware

Beurteilung:
o.Univ.-Prof. Dr. Dr. h.c. Hanspeter Mössenböck

Mitwirkung:
Mag. Dr. Reinhard Wolfinger

Linz, März 2010

Danksagung

Diese Diplomarbeit wäre nicht ohne die tatkräftige Unterstützung einiger Personen entstanden, die ich hier nennen möchte und bei denen ich mich nochmals herzlich bedanken will. Zuerst möchte ich mich beim Team des Instituts für Systemsoftware und dem Team für Automated Software Engineering unter der Leitung von Univ.-Prof. Dr. Hans-Peter Mössenböck bedanken, besonders aber bei Reinhard Wolfinger, Markus Löberbauer und Markus Jahn. Die Gespräche mit Euch haben mir immer wieder weitergeholfen. Ich freue mich schon auf eure Dissertationen.

Besonders möchte ich Martin Luger erwähnen, der mich immer wieder mit aufmunternden Worten im Studium begleitet hat. Vielen Dank dafür. Ich wünsche dir alles erdenklich Gute bei deiner akademischen Karriere.

Zuletzt gilt meiner Familie großer Dank, die mir den nötigen Rückhalt gegeben hat und mich trotzdem anspornte. Vielen Dank an euch alle. Ich hätte sonst nicht das machen können, was ich mir immer gewünscht habe.

Christian Mittermair, 2010

Kurzfassung

BMD Systemhaus GmbH entwickelt Business Software für mittelständische Betriebe. Das aktuelle Produkt trägt den Namen NTCS(New Technology Commercial System).

NTCS ist eine umfangreiche und komplexe Anwendung und besteht aus vielen verschiedenen Funktionen. Für Anfänger stellt, sich im Funktionsumfang zurecht zu finden. Die monolithische Architektur stellt ein zunehmendes Problem sowohl in der Entwicklung als auch Verteilung und die Installation beim Kunden dar. Weiters ist die verwendete Sprache Delphi und die Entwicklungsumgebung für Delphi nicht mehr mit aktuellen Entwicklungsumgebungen vergleichbar und konkurrenzfähig.

In dieser Arbeit wurde NTCS analysiert und in einem zweiten Schritt die Basisklassen von Delphi nach Delphi.Net portiert. Durch die Portierung konnte mit geringen Änderungen der bestehende Quelltext auf die Plattform Mirocsoft.NET migriert werden. In einem weiteren Schritt wurde ein Vorschlag erarbeitet, die monolithische Anwendung in einzelne Komponenten zu zerlegen, die als Plug-Ins des Frameworks Plux.NET vom Benutzer zusammengesetzt werden können. Das Ergebnis dieser Arbeit ist eine prototypische Implementierung. Anhand dieses Prototyps werden die Lösungen für die aufgestellten Probleme praktisch aufgezeigt.

Abstract

BMD Systemhaus GmbH develops business software for medium-sized enterprises. The current project is entitled NTCS (New Technology Commercial System).

NTCS is a comprehensive and complex application and consists of many different functions. For beginners it is difficult to successfully deal with the scope of functions. The monolithic architecture complicates the development as well as the distribution and installation at the customer. Furthermore, the used language Delphi and the development environment for Delphi are no longer comparable or competitive to current development environments.

In this work NTCS was analysed first, and in a second step, the basic classes were ported form Delphi to Delphi.NET. By this step the existing source code could be migrated to the platform Microsoft.NET with minor adjustments. In a further step a proposal was created for disassembling the monolithic application into components that can be assembled by the users as Plug-Ins with the Plux.NET Framework. The result of this thesis is a prototypical implementation. Based on this prototype the solutions for the given problems are shown practically.

Vorwort

Christian Doppler Labor for Automated Software Engineering

Diese Arbeit wurde im Rahmen des Christian Doppler Labors für Autmated Software Engineering am Institut für Systemsoftware der Johannes Kepler Universität Linz in Zusammenarbeit mit der Firma BMD Systemhaus GmbH in Steyr erstellt. Das Labor beschäftigt sich unter anderem mit der Architektur von Softwaresystemen, dabei speziell mit dem Erstellen von großen Anwendungen mit leichtgewichtigen Komponenten mit hoher Wiederverwendbarkeit und starker Kapselung.

BMD Systemhaus GmbH

BMD Systemhaus GmbH wurde 1972 in Steyr gegründet. BMD ("Bürocomputer mittlerer Datentechnik") entwickelt und vertreibt integrierte Unternehmenssoftware für den Bereich der klein bis mittelständischen Unternehmen.

BMD vertreibt gleichzeitig zwei Generationen von Unternehmenssoftware: Die ältere Generation lautet BMD 5.5, ist in Cobol implementiert und hat eine textbasierte, zeilenorientierte Benutzeroberfläche. Die aktuelle Generation trägt den Namen "New Technology

Commercial System" – kurz NTCS, ist in Delphi implementiert und hat eine grafische Benutzeroberfläche.

Um in Zukunft noch besser auf Kundenbedürfnisse eingehen zu können, arbeitet BMD mit dem Christian Doppler Labor gemeinsam an der Erforschung von Plug-In-Plattformen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Umfeld und Problemstellung

1.1 BMD - New Technology Commercial System

Seit 1997 arbeiten die Entwickler bei BMD am Nachfolger für die in Cobol implementierte Rechnungswesensoftware (Knasmüller, 2001). Die Oberfläche dieser Rechnungswesensoftware ist zeilenorientiert. Für die Neuentwicklung war eine grafische Benutzeroberfläche eine wesentliche Anforderung.

Eine weitere Anforderung an die Neuentwicklung war ein objektorientierter Entwurf und eine objektorientierte Architektur. Um diese Anforderungen zu erfüllen, gab es zum damaligen Zeitpunkt mehrere Entwicklungsumgebungen. Für BMD standen C++ und MFC, Java, Visual Basic und Borland Delphi zur Auswahl.

C++ und MFC wurden wegen der großen Komplexität ausgeschlossen und Visual Basic war nicht objektorientiert. Java wurde als noch nicht reif erachtet, um eine große Un-

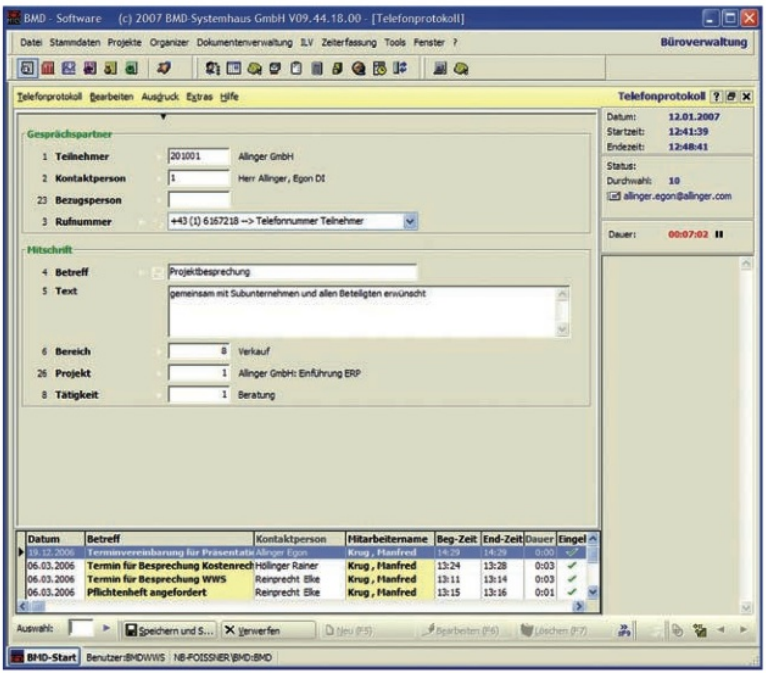

Abbildung 1.1: Benutzeroberfläche des ERP-Programmes

ternehmensanwendung zu erstellen. Borland Delphi war eine objektorientierte und reife Programmiersprache mit einer umfangreichen Klassenbibliothek.

Für Rechnungswesensoftware ist der Zugriff auf Datenbanken eine wichtige Funktion. Die Unterstützung dafür war in Delphi vergleichsweise besser als in anderen Technologien. Mit Delphi ist es einfach, Anwendungen mit einer graphischen Benutzeroberfläche zu erstellen. Diese beiden Vorzüge gaben bei der Neuentwicklung den Ausschlag zu Gunsten von Delphi. (Knasmüller, 2001)

Neben den Paketen zum Buchen und Bilanzieren verfügt das ERP-Programm unter anderem auch über Pakete zur Personalverrechnung, Büroverwaltung, Customer Relationship Management, Lagerhaltung, Produktionsplanung, Projektmanagement oder zur Archivierung von Dokumenten um nur die wichtigsten zu nennen. (BMD/Preisliste, 12.12.2008)

Das ERP-Programm bildet sämtliche Geschäftsfälle rund um den betrieblichen Alltag ab. Es ist auf dem Markt etabliert und bei über 18.000 Kunden in Verwendung. (BMD/Unternehmensprofil, 1.6.2009)

In der Softwareentwicklung hat sich in den letzten 10 Jahren viel verändert. Objektorientierte Plattformen mit automatischer Speicherbereinigung dominieren die Softwareentwicklung. Java war hier lange Zeit der wichtigste Vertreter solcher Plattformen. Mircosoft hat mit dem Microsoft.NET Framework und C# eine solche Plattform mit großem Potential auf den Markt gebracht.

Virtuelle Maschinen und Zwischensprachen erleichtern es, den Quelltext auf viele verschiedene Plattformen zu portieren. Die Übersetzer für solche Plattformen übersetzen den Quelltext in eine Zwischensprache für eine virtuelle Maschine und nicht mehr in direkt ausführbaren Maschinencode. Java und Microsoft.NET unterscheiden sich in ihrer Strategie.

In Java gibt es nur eine Programmiersprache, mehrere virtuelle Maschinen. In Microsoft.NET ist das anders. Mit der Common Language Runtime (CLR) gibt es nur eine Zielplattform, es gibt mehrere Programmiersprachen, aus denen ausführbarer Code für diese Zielplattform erzeugt werden kann. Die Hersteller verschiedener Programmiersprachen haben die Möglichkeit, so ausführbaren Code für diese Plattform zu erzeugen. Sie müssen dafür einen Übersetzer entwerfen, der den bestehenden Quelltext in die Zwischensprache (Common Language Infrastructure) von Microsoft.NET übersetzen kann. Borland ist mit Delphi.NET diesen Weg gegannen.

Das Ziel von Delphi.NET war es, die bestehenden Quelltexte mit geringem Aufwand auf diese Plattform zu portieren. Dieses Ziel konnte puncto Portierbarkeit der Sprache und der Programmierung der Benutzeroberfläche zufriedenstellend erreicht werden.

Bei der Sprache mussten einige Zugeständnisse an das Microsoft.NET Framework gemacht werden. Delphi.NET ist ein Teilmenge von Delphi und somit nicht komplett identisch. Besonders bei hardwarenaher Programmierung müssen bei der Portierung komplexe Anpassungen vorgenommen werden.

1.2 Problemstellung

Die Entwickler des ERP-Programmes sind seit 12 Jahren mit der Entwicklung und Wartung beschäftigt. In dieser Zeit ist der Umfang dieser Software stark gewachsen. Das ERP-Programm steht noch nicht am Ende des Produktlebenszyklus und für die Entwickler gibt es keine Notwendigkeit, nach einer Nachfolge für die aktuelle Entwicklung zu suchen. Trotzdem stehen für die Entwickler drei Fakten rund um die Entwicklung des ERP-Programmes fest:

- Das ERP-Programm ist eine umfangreiche und komplexe Unternehmenssoftware. Durch die große Anzahl an Funktionen leidet die Übersichtlichkeit und Erlernbarkeit der Software. Für die Benutzer ist es schwierig, die wenigen Funktionen, die sie häufig verwenden, wiederzufinden.

- Die Architektur des ERP-Programmes ist monolithisch. Für die Entwickler ist es schwierig, das ERP-Programm zu warten, Fehler zu finden oder das ERP-Programm zu erweitern. Durch die monolithische Architektur ist die Komplexität hoch.

- Die Entwicklungsumgebung und die Sprache Delphi sind mit aktuellen Entwicklungsumgebungen nicht mehr konkurrenzfähig. Aktuelle Entwicklungsumgebungen unterstützen die Entwickler in einem höheren Ausmaß als es mit Delphi jetzt möglich ist.

Die nächsten Abschnitte gehen detailiert auf diese Probleme ein.

1.2.1 Umfangreiches und komplexes ERP-Programm

Das ERP-Programm deckt nahezu jeden Aspekt von Geschäftsprozessen im betrieblichen Alltag ab. Dementsprechend viele Teilbereiche und Pakete gibt es in der Anwendung. Die Anzahl der Funktionen wächst mit der Anzahl an Paketen, wovon einige Pakete komplexe Geschäftsprozesse eines Unternehmens abbilden. Diese Komplexität spiegelt sich direkt in der Komplexität der Funktionen wieder und vergrößert somit die Komplexität des ERP-Programmes.

Oft ist die direkte Zuordnung einer Funktion zu einem gewissen Paket nicht möglich, da alle Pakete stark miteinander verwoben sind. Für die Anwender stellt der große Umfang des ERP-Programmes ein Problem dar. Die Fülle an Funktionen ist für den ungeschulten Benutzer auf den ersten Blick nur schwer erlern- und begreifbar. Unter diesen Umständen ist das Konzept “One-size-fits-all” hinderlich.

Dieses Konzept schränkt die Anwender stark ein. Man kann das ERP-Programm nämlich nur schwer für einen bestimmten Anwender individualisieren. Es bietet den Anwendern zwar alle notwendigen Funktionen, die Anwender haben durch die große Menge an Funktionen Schwierigkeiten, die richtige Funktion auszuwählen. Jeder Anwender bekommt beim Start die gleiche Benutzeroberfläche, egal für welchen Tätigkeitsbereich er zuständig ist. Der Lernaufwand ist daher hoch.

Darum versuchen die Entwickler des ERP-Programmes die Funktionalität klar und verständlich zu halten. Für einen definierten Geschäftsprozess gibt es daher nur eine Art, um diesen abzuarbeiten. Für die Anwender des ERP-Programmes kann sich das jedoch als schwierig herausstellen.

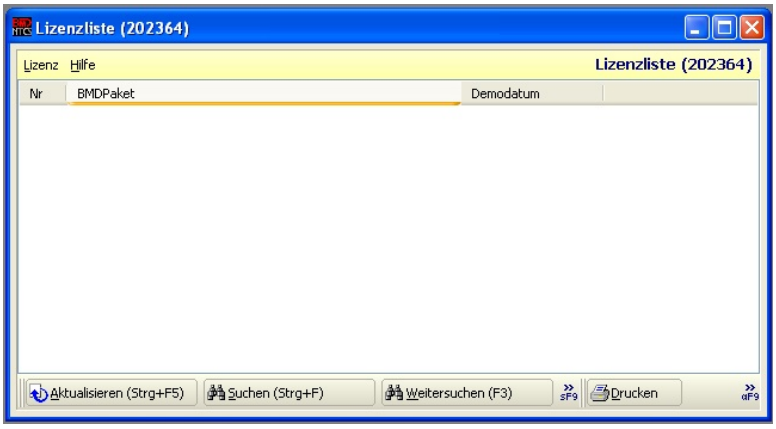

Abbildung 1.2: Lizenzliste für selektives Freischalten von Funktionen

So kann es sein, dass die Geschäftsprozesse des Anwenders sich an die Anwendung anpassen müssen. Für den einzelnen Anwender kann sich dies als unmöglich herausstellen. Für die Entwickler des ERP-Programmes ist es nicht möglich, für jeden Anwender spezielle Funktionen zur Abwicklung seiner speziellen Geschäftsprozesse zu implementieren. Dies gilt besonders, wenn ein Geschäftsprozess nur bei einem einzigen Anwender auftritt.

Könnten die Entwickler des ERP-Programmes die Komplexität der Geschäftsanwendung reduzieren oder umgestalten, dass nur die notwendigen Funktionen verfügbar sind, wäre es für neue Benutzer leichter, sich damit zurechtzufinden. Die Anwender könnten das ERP-Programm feingranular auf die wirklich benötigten Funktionen reduzieren. Für die Entwickler hingegen wäre es möglich, auf die Wünsche der Anwender einzugehen.

1.2.2 Monolithische Architektur

Das ERP-Programm hat als Unternehmenssoftware Funktionen, die sich über verschiedene Bereiche des Unternehmens erstrecken. Diese Funktionen lassen sich nur schwer einem gewissen Teil der Anwendung zuordnen. Diese Tatsache spiegelt sich direkt in den Quelltexten wieder. Delphi enthält erst seit Version 4 Konzepte wie Interfaces oder Mo-

dule. Weiters unterstützt Delphi die Aufteilung in Komponenten nicht zufriedenstellend. Das alles führte dazu, dass das ERP-Programm eine einzelne ausführbare Binärdatei mit mehr als 80 Megabyte ist.

Im Programmcode des ERP-Programmes sind neben dem Produktionsquelltext auch die Laufzeit- und die GUI Bibliothek von Delphi enthalten. Die Binärdatei enthält zusätzlich noch Bibliotheken anderer Hersteller, die für den Datenbankzugriff oder zur Kalenderanzeige benötigt werden.

Für die Entwickler des ERP-Programmes wirkt sich dies negativ auf die Wartbarkeit und Erweiterbarkeit aus. Das Übersetzen dauert auch bei kleinen Änderungen lange. Da die einzelnen Bereiche der Anwendung stark miteinander verwoben sind, haben kleine Änderungen große Auswirkungen auf andere Teile. Man kann Seiteneffekte nur durch intensives Testen finden.

Die Entwickler passen das ERP-Programm laufend an gesetzliche Anforderungen an. Neben diesen Änderungen beheben sie auch Fehler oder erweitern die Anwendung. Für die Entwickler ist es schwierig, die betreffenden Stellen im Quelltext zu finden, weil das ERP-Programm aufgrund der monolithischen Struktur schwer überschaubar ist. Finden die Entwickler einen Fehler und beheben diesen, kann diese Änderung nicht selektiv an die Anwender ausschickt werden. Die Anwender des ERP-Programmes müssen die gesamte Anwendung austauschen. Erst dann ist die Änderung auch bei den Anwendern aktiv.

Für die Anwender ist diese Vorgangsweise ein Problem. Wie beschrieben, installieren die Anwender Updates oder Anpassungen nur durch den Austausch der gesamten Anwendung. Das kann zu Problemen führen, weil unter Umständen gerade kritische Arbeitsabläufe wie das Bilanzieren stattfinden. Deshalb möchten die Anwender kein Update installieren, um nicht den wichtigen Arbeitsablauf zu stören oder gar zu verhindern.

Gleichzeitig ist es für eine andere Abteilung notwendig, die neue Version des ERP-Programmes, um arbeiten zu können. Die erste Abteilung will jedoch die alte Version behalten, weil die Abteilung erst ihre Arbeit fertigstellen muss.

Die monolithische Architektur verhindert die selektive Aktualisierung des ERP-Programmes. Die Anwender müssen entscheiden, ob sie die neue Version der Anwendung installieren oder noch warten wollen. Dadurch kann es zwischen verschiedenen Abteilungen zu einem Konflikt kommen, obwohl die Abteilungen wenig miteinander zu tun haben und auch unterschiedliche Aufgabengebiete betreuen.

Für die Entwickler des ERP-Programmes ist es interessant, dass Dritthersteller von Software spezielle Erweiterung des ERP-Programmes entwickeln können. Die Entwickler des ERP-Programmes könnten sich auf die Kernfunktionalität der Anwendung konzentrieren, während Dritthersteller autonom Erweiterungen für das ERP-Programm erstellen können. Die Anwender selbst können Dritthersteller beauftragen spezielle Funktionen zu programmieren.

Der Kern des ERP-Programmes als solches wird nicht verändert und bleibt für alle Anwender gleich. Trotzdem hat jeder Anwender die Möglichkeit, das ERP-Programm mit den verschiedenen Erweiterungen der Dritthersteller an die eigenen Geschäftsprozesse anzupassen.

Eine solche Zusammenarbeit mit Drittherstellern ist nur dann möglich, wenn es klar definierte Schnittstellen gibt. Aufgrund der monolithischen Architektur des ERP-Programmes fehlen solche Schnittstellen.

Könnten die Entwickler die monolithische Code-Basis so auftrennen, dass getrennt übersetzbare Module entstehen, wäre es für die Entwickler leichter die Anwendung zu warten und Updates für die Anwender anzubieten.

Für Anwender wäre es möglich, selektiv ein Update für eine bestimmte Abteilung zu installieren, ohne dass andere Abteilungen beeinträchtigt würden. Durch die Modularisierung würden klar definierte Schnittstellen entstehen, die Dritthersteller nutzen könnten, um das ERP-Programm zu erweitern.

1.2.3 Unsichere Zukunft der Entwicklungsumgebung

Die Programmiersprache Delphi und die zugehörige Entwicklungsumgebung verursachen verschiedene Probleme für die Entwickler des ERP-Programmes. Einerseits fehlen der Sprache bis Delphi 2009 verschiedene wichtige Konzepte wie zum Beispiel Generics. Andererseits hinkt die Entwicklungsumgebung anderen Produkten wie Visual Studio oder Eclipse hinterher.

Auch der *Build*-Prozess von Delphi stellt die Entwickler vor Probleme. Der Übersetzer von Delphi legt alle Zwischenkompilate in Dateien ab. Diese sollen die Dauer der Übersetzung verkürzen, weil nur mehr geänderte Quelltexte neu übersetzt werden. Diese Zwischenkompilate schaffen weitere Abhängigkeiten, die sich beim Übersetzen störend auswirken. Die Zwischenkompilate verursachen dann Probleme, die sich nicht intuitiv lösen lassen.

Neben dem Übersetzer braucht auch die Entwicklungsumgebung die Zwischenkompilate. Die Zwischenkompilate werden verwendet, um den Entwickler mit Code-Completion zu unterstützen. Treten beim Übersetzen Probleme auf, kann man diese nur lösen, indem man die Zwischenkompilate löscht, damit der Übersetzer diese Zwischenkompilate beim nächsten Mal neu erstellt.

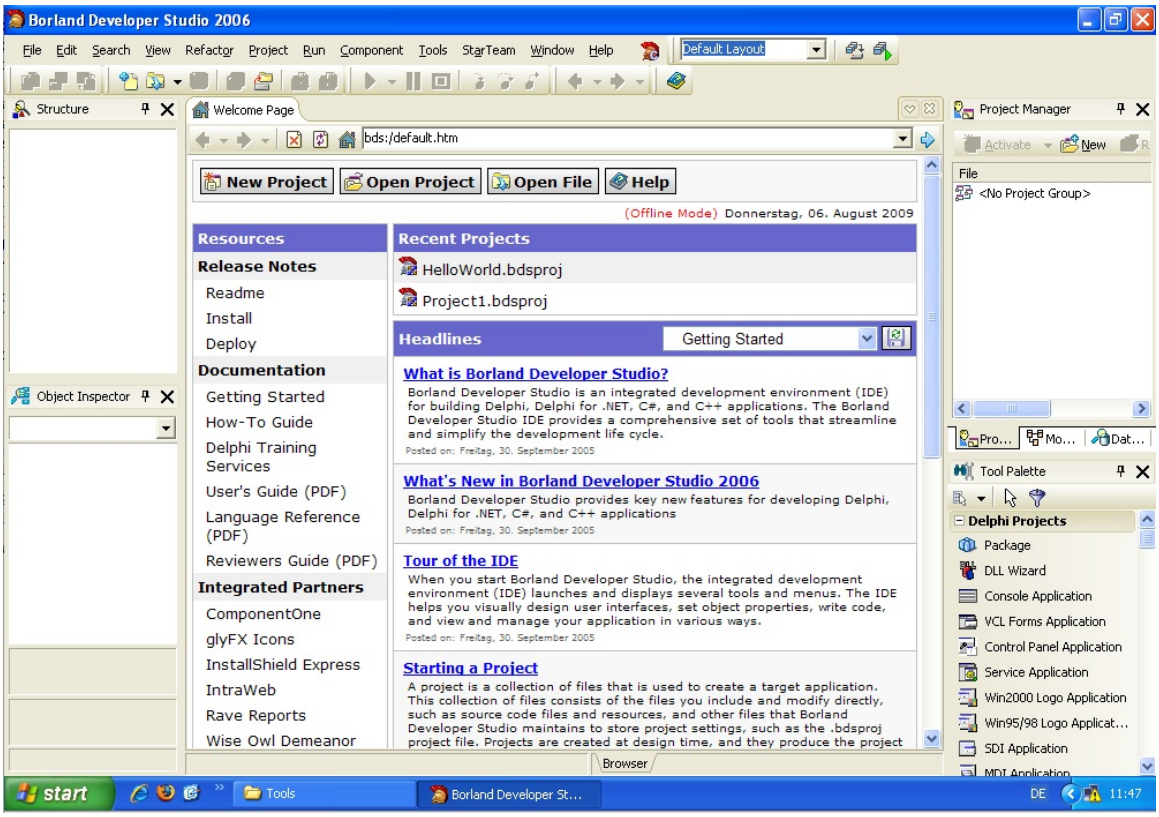

Abbildung 1.3: Startbildschirm von Borland Developer Studio 2006

Ein anderes Problem sind die häufigen Eigentümerwechsel von Delphi. In den letzten Jahren hat war Delphi im Besitz von 4 verschiedenen Firmen. Schon in den 1990er Jahren wurde Borland von Inprise übernommen und 2000 wieder in Borland umbenannt.

2006 spaltete Borland die gesamte Entwicklungssparte ab und gründete die Firma CodeGear. Der Hintergrund dafür war von Beginn an der Wunsch für die gesamte Entwicklungssparte zu verkaufen. Seit Sommer 2008 lautet nun der neue Eigentümer Embarcadero.

Der neue Eigentümer schlägt auch eine neue Richtung ein. Embarcadero gab kurz danach bekannt, Delphi.NET nicht mehr weiter zu entwickeln. Mit Version Delphi 2009 wurde Delphi.NET mitsamt der Laufzeit- und GUI Bibliothek eingestellt.

Anstatt von Delphi.NET setzt Embarcadero auf die syntaxähnliche Sprache. Das neue Produkt Delphi Prism ist nicht kompatibel mit Delphi.NET. Delphi Prism verwendet im Gegensatz zu Delphi.NET die Bibliotheken von Microsoft.NET.

Delphi wurde in den vergangenen Jahren nicht kontinuierlich weiterentwickelt und hat daher den Anschluss an andere Entwicklungsumgebungen verloren. Mit dem drastischen Sprung auf Delphi Prism versucht Embarcadero wieder aufzuschließen. Diese Abspaltung erleichtert nicht zuletzt auch die Weiterentwicklung von Delphi. Bei Embarcadero will man sich nicht festlegen, ob und wie weit sich die beiden Sprachen in Zukunft auseinander entwickeln.

Die Vergangenheit und die aktuelle Entwicklung von Delphi stärkt nicht das Vertrauen an diese Plattform. Für die Entwickler des ERP-Programmes wäre es ein Problem, wenn Delphi eingestellt würde. Sie hätten dann ihre Anwendung mit einer Entwicklungsumgebung geschaffen haben, die in Zukunft nicht mehr weiter entwickelt wird.

Könnten die Entwickler des ERP-Programmes den Quelltext so verändern, dass dieser auf einer anderen Plattform wieder verwendbar ist, wäre dieser Unternehmenswert für die Zukunft gesichert.

1.3 Zusammenfassung

Das ERP-Programm ist am Markt gut etabliert. Die Probleme sind sowohl für die Entwickler als auch für die Anwender des ERP-Programmes hinderlich. Wenn die Entwickler die angeführten Probleme lösen, entstehen sowohl für die Entwickler als auch für die Anwender viele Vorteile.

Durch die Reduktionen der Komplexität, könnten Anwendungen realisiert werden, die auf den Anwendungsfall und die aktuelle Rolle des Anwenders zugeschnitten sind. Könnten die Entwickler das ERP-Programm feingranular zerlegen, könnten dann nur gewisse Teile ausgewechselt werden. Stehen gesetzliche Neuerungen an oder wurde ein Fehler ausgebessert, könnte dann nur dieser kleine Teil ausgewechselt werden, ohne dass das gesamte ERP-Programm gestört wurde.

Neben der besseren Ausrichtung des ERP-Programmes auf den Anwender, haben die Entwickler selbst mehr Möglichkeiten in Zukunft flexibel und konzentrierter am ERP-Programmes zu arbeiten.

Mit der Zerteilung des ERP-Programmes und dem Aufstellen von Schnittstellen, wäre es möglich, dass Dritthersteller ganz spezielle Geschäftsprozesse oder maßgeschneiderte Lösungen für einen speziellen Anwender entwickeln. Für die Entwickler selbst vereinfacht sich der Entwicklungsprozess, da sich durch die definierten Schnittstellen Seiteneffekte reduzieren und genau lokalisieren lassen.

Bei einem Wechsel der Entwicklungsumgebung entstehen für die Entwickler des ERP-Programms neue Möglichkeiten und Perspektiven. Einerseits gibt es bei anderen Plattformen mehr Entwickler auf dem Arbeitsmarkt, die sofort ohne umfassende Einschulung bei der Entwicklung am ERP-Programm mitarbeiten können. Andererseits werden mit einer aktuellen Entwicklungsplattform die Entwickler besser bei der Arbeit unterstützt. Die Entwickler des ERP-Programmes befinden sich gegenüber den Mitbewerbern im Vorteil, wenn es gelingt, all diese Probleme zu beheben.

Kapitel 2

Ausgangssituation

Um ein Verständnis für das ERP-Programm zu bekommen, muss man es aus zwei Blickwinkeln betrachten: Die Sicht der Entwickler und die Sicht der Anwender auf das ERP-Programm. Für die Entwickler sind die technischen Hintergründe von Interesse. Für die Anwender sind die Funktionen und die Bedienbarkeit des ERP-Programmes wichtig. Für beide ist das ERP-Programm ein großes und umfangreiches Programmpaket.

2.1 Monolithisches ERP-Programm aus Anwendersicht

2.1.1 Strukturierung des ERP-Programmes in Pakete

Aus Sicht der Anwender gliedert sich das ERP-Programm primär in fünf verschiedene Pakete, die geauswählt werden können:

- `BMDCONSULT` für für Steuerberater und Wirtschaftsprüfer.

- `BMDCRM` für die Verwaltung von Kundenbeziehungen.
- `BMDACCOUNT` für das betriebliche Rechnungswesen.
- `BMDCOMMERCE` für ein Warenwirtschaftssystem.
- `BMDPPS` zur Produktionsplanung.
- `BMDPROJECT` für die Projektabwicklung.

Für den Anwender besteht zusätzlich die Möglichkeit, Teile aus den Paketen einzeln zu kaufen. Die Funktionen sind in die Bereiche Verkauf, Lager, Einkauf, Kassenlösung, Produktionsplanung, Finanzbuchhaltung, Zahlungsverkehr, Controlling, Kostenrechnung, Bilanzierung, Berichtswesen, Lohnverrechnung, Leistungserfassung, Zeitverrechnung, Human Resources Management und Büro- und Dokumentenmanagement eingeteilt (BMD/Preisliste, 12.12.2008).

Das ERP-Programm hat seine Wurzeln in einem zeilenorientierten Programm, bei dem die Funktionen hierarchisch angeordnet sind. Diese Benutzerführung ist bei den Anwendern des ERP-Programmes lange eingeführt und wohl bekannt. In der aktuellen Version des ERP-Programmes gibt es ebenfalls diese Ansicht. *BMD-Quickstart* zeigt dem Benutzer alle Funktionen in einer hierarchischen Weise (siehe Abbildung 2.2).

2.1.2 Funktionsumfang einschränken oder erweitern

Das ERP-Programm ist eine monolithische Applikation. Das heisst, dass bei jedem Kunden die gleiche Applikation installiert ist. Somit ist für jeden Kunden theoretisch der gesamte Funktionsumfang verfügbar. Damit der einzelne Kunde nur jene Funktionen

startet, für die er sich entschieden hat, werden je nach Lizenz gewisse Funktionen freigeschaltet.

Im ERP-Programm gibt es eine Lizenzliste, um für jeden Kunden die Funktionsfreigabe feingranular zu steuern. Nur jene Funktionen, die in dieser Lizenzliste über einen Lizenzschlüssel freigegeben wurden, kann der jeweilige Kunde dann auch ausführen. Für die Entwickler ist das die einzige Möglichkeit, die Anwendung als solche trotzdem in Bezug auf die Anforderungen des Kunden zu individualisieren.

2.1.3 Funktionen aufrufen

Beim ERP-Programm sind die Funktionen – wie auch im zeilenorientiertem System – hierarchisch gegliedert. In Abbildung 2.1 wird ein Auszug aus den Funktionen gezeigt, die durch *BMD-Quickstart* zum Start angeboten werden.

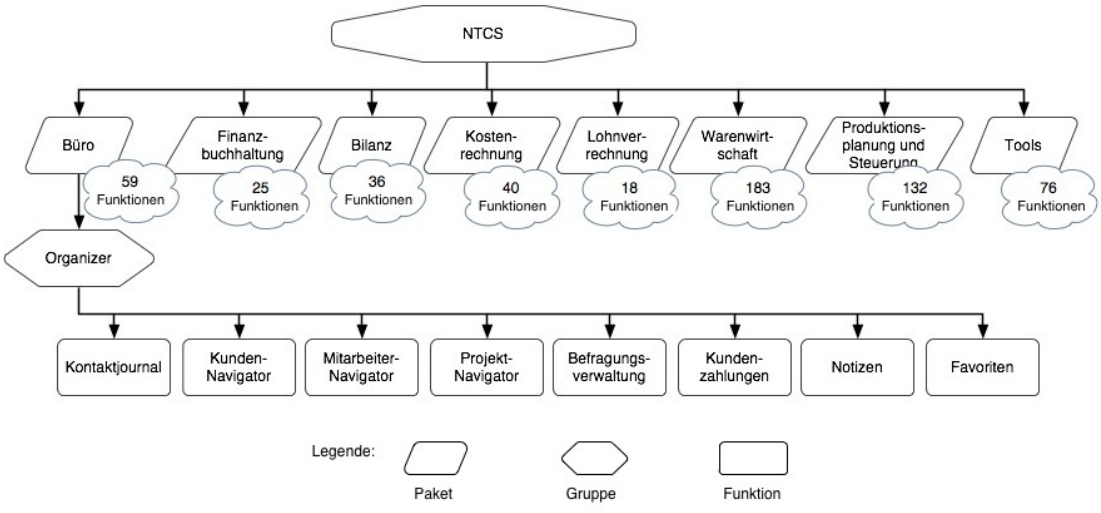

Abbildung 2.1: Auszug aus der Funktionsliste des ERP-Programmes

Neben *BMD-Quickstart* kann man Funktionen auch über die Menüleiste starten. Die Menüleiste stellt Funktion als inaktiv dar, wenn die Lizenz zum Ausführen der Funktion fehlt oder die Funktion aktuell nicht auswählbar ist. Aufgrund der großen Anzahl an Funktionen kann die Darstellung im Menü schnell unübersichtlich werden.

In *BMD-Quickstart* werden inaktive Funktionen zwar angezeigt, jedoch ist das Starten der Funktion nicht möglich. Die Benutzer sind durch diese Konvention verwirrt. Der Benutzer sieht die Funktion, ohne dass er sie ausführen kann.

Die Anwender haben die Anforderung, dass die Software komplett ohne Maus bedienbar sein soll. Der Benutzer muss jede Funktion über die Tastatur starten können. Darum gibt es zusätzlich zu den Menüleisten und *BMD-Quickstart* die Möglichkeit, jede Funktion über den *Matchcode* zu starten. Diese Anforderung stammt ebenfalls aus der Zeit der zeilenorientierten Benutzeroberfläche und gilt nach wievor für die aktuelle Version. Daher ist es auch im ERP-Programm wie in der zeilenorientierten Anwendung möglich, jede Funktion durch die Eingabe des *Matchcodes*, direkt zu starten. Dieser *Matchcode* ist für jede Funktion eindeutig. Jede Funktion kann über die Eingabe ihres *Matchcodes* gestartet werden (siehe Abbildung 2.2.b).

2.1.4 Benutzeroberfläche

Die Benutzeroberfläche des ERP-Programmes ist ein sogenanntes *Multiple Document Interface* (kurz: MDI). Das Hauptfenster enthält dabei viele verschiedenene innenliegende Fenster. Diese Fenster enthalten die Benutzeroberfläche für die gestarteten Funktionen.

Wie in Abbildung 2.2.a ersichtlich gibt es in der Statuszeile die Schaltfläche *BMD-Start*. Dieses Steuerelement startet die Funktion *BMD-Quickstart*. Neben *BMD-Quickstart* gibt es noch die Möglichkeit, Funktionen über das Menü zu starten. Um die Anzahl an Funk-

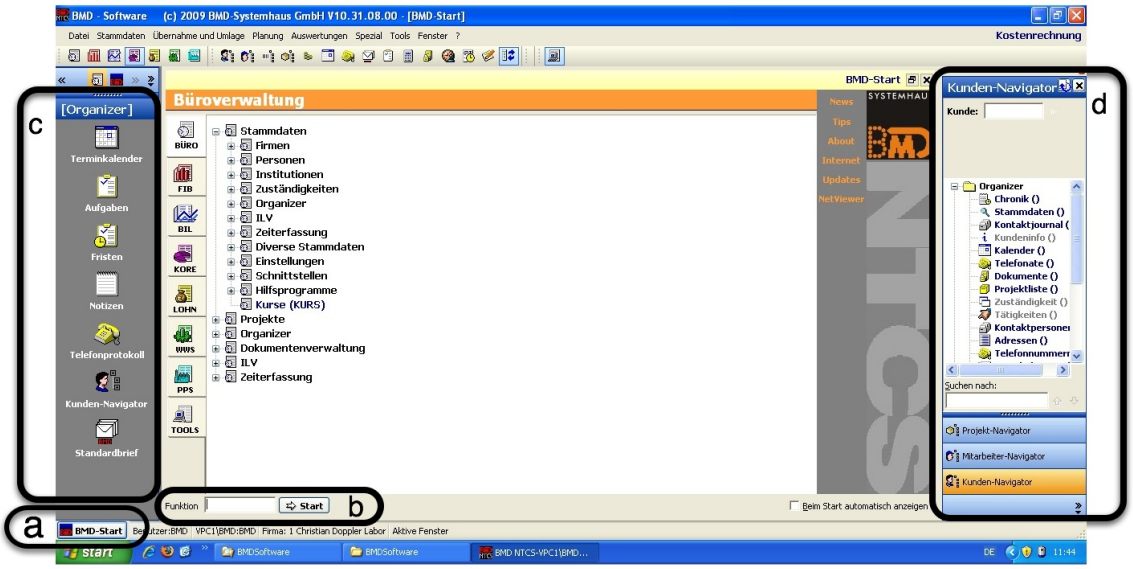

Abbildung 2.2: Das ERP-Programm mit der gestarteten Funktion BMD-Quickstart

tionen einzuschränken muss man das gewünschte Paket erst auswählen. Das jeweilige Paket kann man über die Menüleiste im Menü *Datei* umschalten.

Neben der Menüleiste gibt es sowohl links als auch rechts noch weitere Leisten: Am linken Bildschirmrand (Abbildung 2.2.c) befindet sich eine Outlook-ähnliche Leiste. Diese Leiste zeigt ausgewählte, häufig gebrauchte Funktionen zum Starten anbietet. In Abbildung 2.2.d sind rechten Bildschirmrand die *Navigatoren* zu sehen. Diese Navigatoren gruppieren die Arbeitsabläufe eines konkreten Aufgabengebiets.

2.2 Monolithisches ERP-Programm aus Entwicklersicht

Aus Entwicklersicht ist das ERP-Programm ein großes und komplexes Produkt. Aufgrund des Funktionsumfanges ist der Einlernaufwand für neue Entwickler groß. Das ERP-Programm ist eine einzelne ausführbare Datei, die mehr als 80 Megabyte groß ist.

Diese 80 Megabyte sind das Resultat von ca. 13.000 Quelltextdateien mit rund vier Millionen Codezeilen.

2.2.1 Monolithische Architektur

Das ERP-Programm wird immer in eine einzige Datei übersetzt. Daher präsentiert sich das ERP-Programm nach außen als eine große und monolithische ausführbare Datei. Der Grund dafür ist hauptsächlich die Entwicklungsumgebung Delphi. Delphi selbst unterstützt den Entwickler bei der Modularisierung des Codes nicht.

Es ist somit notwendig, selbst bei einer kleinen Änderung die gesamte Anwendung neu zu übersetzen. Dieser Umstand macht es für die Entwickler des ERP-Programmes unmöglich, mehrmals pro Stunde den Quelltext neu zu übersetzen und somit den programmierten Code zu testen. Häufiges Übersetzen eines Programmes hilft den Entwicklern, Fehler so früh wie möglich zu entdecken.

Durch die fehlende Modularisierung ist keine getrennte Übersetzung möglich, was die Produktivität eines einzelnen Entwicklers hemmt. Der gesamte Code wird in eine einzige, nativ ausführbare Datei übersetzt. Alle Pakete mit den dazugehörigen Funktionen sind darin enthalten. Die Entwicker haben versucht, die Architektur des ERP-Programmes so straff wie möglich zu halten. Die Entwicklungsplattform unterstützt die Entwickler nicht ausreichend bei der Modularisierung der Anwendung.

Der Architekturansatz des ERP-Programmes entspricht dem Top-Down Design, bei der das Problem immer weiter von oben nach unten verfeinert wird. Eine Top-Down Architektur reduziert die Komplexität des Problems und erleichtert so den Entwurf. Die Top-Down Architektur macht die Wiederverwendung von Code dafür deutlich schwie-

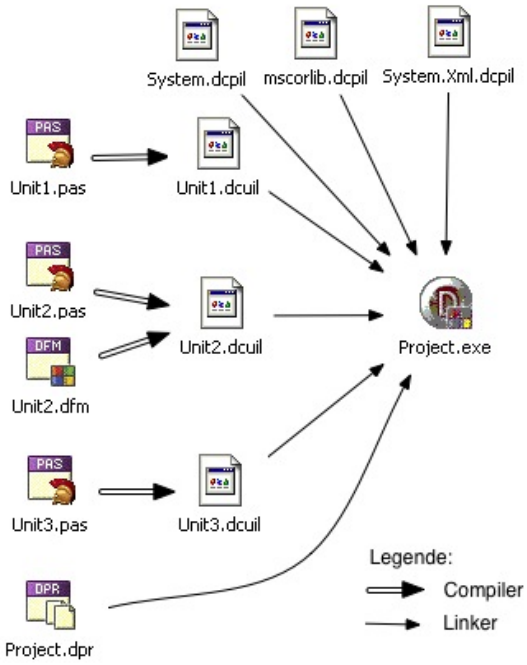

Abbildung 2.3: Übersetzung einer Delphi Unit mit Zwischenkompilaten und dem Ergebnis

riger. Die Abbildung 2.4 zeigt, dass in jeder Stufe sich immer wieder Quelltext für das Model-View-Controller-Architekturmuster (kurz: MVC) befindet.

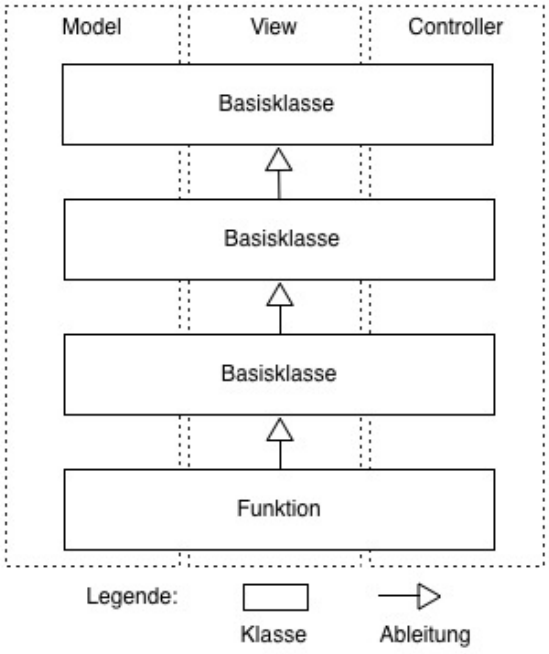

Abbildung 2.4: Implementierung der Model-, View- und Controllerklassen über vier Ebenen verstreut

Im Quelltext gibt es *Units* mit mehreren tausend Quelltextzeilen. Das ist für die Übersichtlichkeit und Verständlichkeit schlecht. Viele dieser *Units* sind für die zentrale Abarbeitung von Aufrufen zuständig. Diese *Units* enthalten viele statische Methoden. Diese statische Methoden sind für den Aufruf und Abarbeitung der einzelnen Funktionen zuständig. Der Code einer einzelnen Methode ist nicht groß und nicht schwer zu verstehen. Die Summe an verschiedenen Funktionen und auch die massive Codeverdopplung sind für das Verständnis und die Wartbarkeit schlecht.

2.2.2 Modularisierung in Delphi

Delphi unterstützt die Entwickler nicht bei der Modularisierung, da es in Delphi üblich ist, externe Bibliotheken nicht als getrennt übersetzbare Einheiten bereitzustellen. Stattdessen wird der Quelltext der Bibliothek direkt in die ausführbare Datei mitübersetzt. Das hat den Vorteil, dass diese Bibliothek zur Übersetzungszeit die gleiche ist, wie zur Ausführungszeit.

Es wäre möglich *Borland Delphi Packed Libraries* (kurz: BPL) oder *Dynamic Link Libraries* (kurz: DLL) zu erzeugen und diese mit Aufrufen einzubinden. Die Entwickler des ERP-Progeammes haben diese Möglichkeit nicht eingesetzt, weil sich bei Tests große Probleme für der Entwicklung herausstellten.

Im Quelltext vom ERP-Programm gibt es einzelne Quelltextdateien mit mehreren tausend Zeilen. Erst seit Version 4 unterstützt Delphi das Konzpet von Interfaces. Deshalb finden sich im gesamten Quelltext des ERP-Programmes trotz des großen Umfangs nur wenige Interfaces. Der gesammte Quelltext befindet sich in einem Ordner. Darin enthalten sind alle notwendigen Quelltextdateien, die durch ein einzelnes Projekt im Übersetzer zu einer ausführbaren Datei übersetzt werden.

Listing 2.1: BMDNTCS.dpr

```
program BMDNTCS;

uses
  BMDMenuBar, Forms, BMDLoginFrm, BMDGlobalCoordinator,
     BMDLoginDM,
  BMDForm, About, BMDDllManager, BMDMAINFRM, BMDStdUses,
  BMDVersionCheck;

{$R *.RES}
begin
  Application.Initialize;
  Application.UpdateFormatSettings := FALSE;
```

```
  Application.Title := 'BMD NTCS';
  if CheckLocalNTCSVersion then begin //  Mit Lizenzkey
    DoDatabaseLogin( BMDDMLogin, TheGlobCoordinator,
                     'BMDNTCS', TRUE, TRUE, TRUE );
    if not Application.Terminated then begin
      Application.CreateForm(TBMDFRMMain, BMDFRMMain);
      BMDFRMMain.Show;
      Application.Run;
    end;
  end;
end.
```

Dieses Codestück aus dem Listing 2.1 repräsentiert das Hauptprojekt, indem sich die *Main-Methode* befindet. Diese *Main-Methode* hat keine bestimmte Methodensignatur sondern steht nur in einem Block der zu `PROGRAM` gehört. Nach der Überprüfung der Lizenz und dem Einstieg in die Datenbank startet das Anwendungfenster, in dem die freigeschaltenen Pakete in Menüs dargestellt werden. Durch das Einbinden von `BMDGlobalCoordinator` und Aufrufen von `BMDFRMMain` werden notwendigen Quelltext-dateien in die Anwendung miteingebunden.

2.2.3 Entwicklermannschaft für die Basisklassen

Im Entwicklerteam gibt es zwei Gruppen von Entwicklern. Die erste Gruppe ist für Wartung und Erweiterung der Basisklassen zuständig. Die Entwickler gaben dieser Basisklassenbibliothek den Namen *BMD-Tools*. Die Entwickler der Basisklassen stellen mit dieser Klassenbibliothek eine Basisfunktionalität zur Verfügung.

In der Basisklassenbibliothek des ERP-Programmes befinden sich weiters Klassen für den Datenbankzugriff, rudimentäre Geschäftslogik, Berichte und Drucken, die Internationalisierung, der Fensterdarstellung und noch abgeleitete und erweiterte Standardbibliotheken.

Die Klassen der Visual Component Library (kurz: VCL), die von Delphi zur Verfügung gestellt werden, wurden von den Entwicklern der Basisklassenbibliothek generell noch einmal abgeleitet. Diese Vorgehensweise vergrößert die Anzahl an Quelltextdateien und erschwert bei einer Aktualisierung der Standardbibliotheken die Portierung.

Diese Klassenbibliothek hat neben dem eben genannten Vorteil auch den dramatischen Nachteil, dass bei einer Änderung der Schnittstelle oder einer Änderung im Verhalten alle davon abgeleiteten Units angepasst und getestet werden müssen. Nur so kann überprüft werden, ob das Verhalten noch gleich ist.

Die wichtigsten Basisklassen in der Klassenbibliothek sind jene für das MVC-Architekturmuster. Alle Modell-Klassen leiten von `TBMDModel` indirekt ab. Alle Controller-Klassen werden von `TBMDDataSource` und `TBMDActionList` ebenfalls abgeleitet. Sämtliche View-Klassen sind indirekte Ableitungen von `TBMDFRMStdBase` und implementieren `IBMDView`.

2.2.4 Entwicklermannschaft für Funktionen

Die Anwendungsentwickler sind die zweite Gruppe von Entwicklern. Diese Entwickler programmieren die Funktionen des ERP-Programmes. Für diese Anwendungsentwickler wurde die Basisklassenbibliothek entwickelt, damit sie mit einem geringen Aufwand ihre Aufgabe erfüllen können.

Die Basisklassen haben eine breite Schnittstelle mit vielen Funktionen, Prozeduren, Feldern bzw. Properties. Aufgrund der breiten Schnittstelle der Basisklassen, wird die Schnittstelle in jeder abgeleiteten Klasse noch breiter. Dadurch wird es mit jeder Ableitung schwieriger die komplette Schnittstelle einer neu abgeleiteten Klasse zu verstehen.

Zu diesem Zweck finden sich in diesen Basisklassen auch Kopiervorlagen für immer wieder benötigte Bestandteile. Mit diesen Kopiervorlagen kann der Einlernaufwand für neue Entwickler vermindert werden. Die Trennlinie in Abbildung 2.5 grenzt die Kopiervorlagen von den Klassen aus der Klassenbibliothek ab. In diesen Kopiervorlagen wird von den Basisklassen abgleitet und die notwendigen Initialisierungen vorgenommen (Abbildung 2.5). Die Anwendungsentwickler können dann das neue Fenster mit den benötigten Funktionen befüllen.

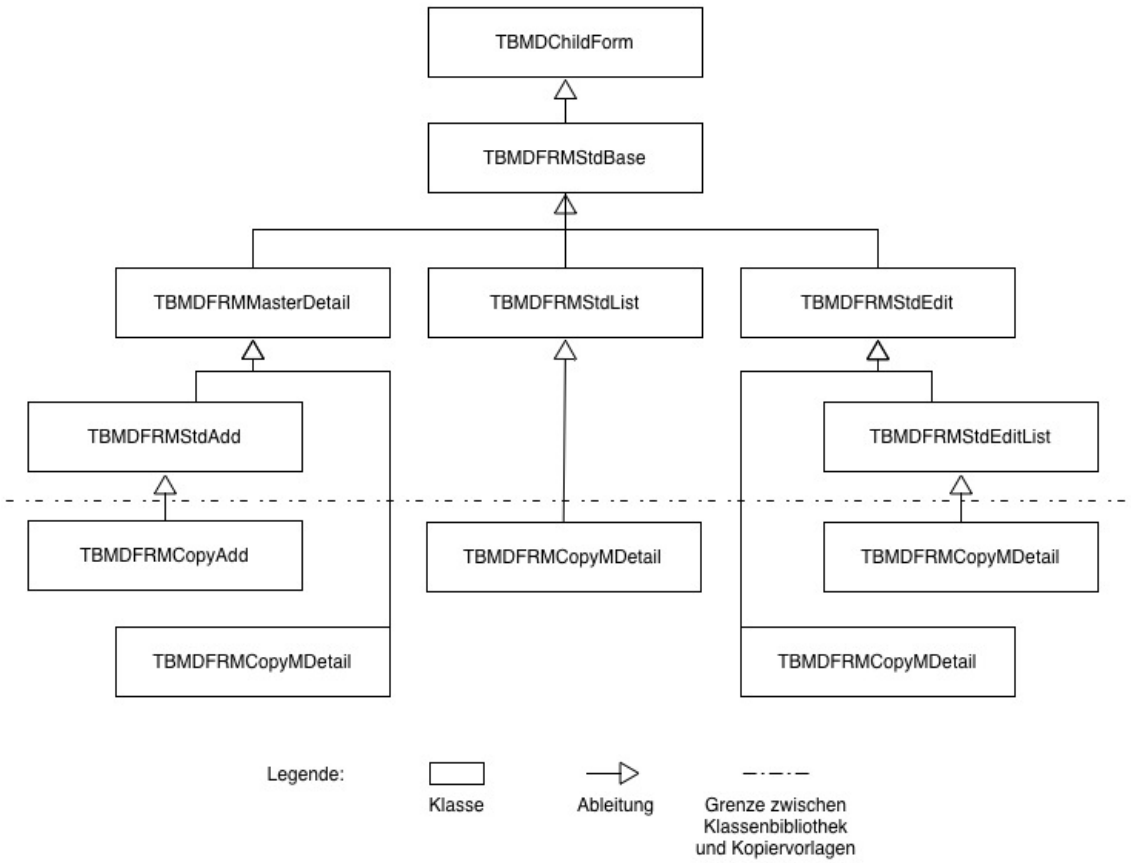

Abbildung 2.5: Basisklassen und davon abgeleitete Kopiervorlagen

Das ERP-Programm enthält viele Funktionen. Diese große Anzahl spiegelt sich in der Anzahl der Quelltextdateien wieder. Für jede Funktion gibt es eine eigene *Unit* im der

Klassenbibliothek, weshalb das vollständige Klassendiagramm unüberschaubar ist. Es können durch die fehlende Modularisierung keine Teilbäume isoliert werden.

Alle Klassen in der Basisklassenbibliothek haben ausschließlich englische Bezeichnungen. Ab der Ebene der Anwendungsprogrammierung befinden sich im Quelltext auch deutsche Begriffe für Methoden und Variablen. Es gibt auch Mischformen, bei denen die Bezeichner für Methoden und Variablen sowohl aus englischen als auch aus deutschen Wörtern bestehen.

2.2.5 Klassenbibliotheken anderer Hersteller

Der Quelltext des ERP-Programms enthält Klassenbibliotheken von anderen Herstellern. Als wichtigste Klassenbibliothek gilt jene für den Datenbankzugriff. Für das ERP-Programm ist die Anbindung an eine Datenbank ist ein wesentlicher Bestandteil. Als Datenbank stehen Oracle oder Microsoft SQL-Server zu Verfügung. Für diese Arbeit hat die Datenbankanbindung einen untergeordneten Stellenwert. Die Bibliotheken, die den Zugriff zur Datenbank herstellen, werden genau betrachtet.

Der Quelltext des ERP-Programmes enthält neben den Bibliotheken für den Datenbankzugriff noch diverse Bibliohteken anderer Hersteller. Diese Bibliotheken enthalten zum Beispiel den Programm-Code für Termin- und Kalenderfunktionalitäten, für String-Operationen und Binary Coded Decimal Arithmetik (kurz: BCD-Arithmetik).

Alle Klassenbibliotheken liegen für die Entwickler des ERP-Programmes im Quelltext vor. Für Entwickler ist es dabei praktisch und oft auch wichtig, Zugriff auf den Quelltext von Bibliotheken zu haben. Einerseits können Fehler ausgebessert werden, andererseits hilft der Quelltext die Funktion der Bibliothek zu verstehen.

Es ist verlockend und gleichzeitig auch problematisch, diese Bibliotheken fremder Hersteller zu verändern und anzupassen. Im ERP-Programm entstanden Abhängigkeiten der Bibliotheken auf den Produktions-Code. Das Resultat daraus waren Zyklen in der Abhängigkeit von Quelltextstücken. Eine andere Folge solcher Verweise sind unvorhergesehene Seiteneffekte.

Ein weiteres Problem ist die Inkompatibilität mit neuen Versionen der externen Klassenbibliotheken. Durch die soeben beschriebenen Änderungen kann man neue Versionen der externen Bibliotheken nicht sofort einsetzen, da die Entwickler erst wieder die neue Version der Bibliothek anpassen müssen. Die Adaptierung des Quelltextes von externen Bibliotheken muss mit großer Vorsicht erfolgen. Im Hinblick auf die Konsequenzen ist es nicht sinnvoll, Änderungen an externen Klassenbibliotheken durchzuführen.

2.2.6 Model-View-Controller Architekturmuster

Das *Model-View-Controller*-Architekturmuster (kurz: MVC) ist auch im ERP-Programm das Standard Muster, um Benutzeroberflächen zu gestalten. Die Entwickler haben versucht, streng zwischen dem *Model*, dem *Controller* und der *View* zu trennen. Bei den Datenmodellen ist dies am besten gelungen. Sie beinhalten neben den Daten aus der Datenbank auch die Geschäftslogik. Die tiefen Vererbungslinien verteilen allerdings den Code für *Model*, *Controller* und *View* auf verschiedene *Units*. Dadurch ist die Trennung von *Controller* und *View* nicht immer scharf.

Die Abbildung 2.5 zeigt wie aus den Basisklassen `TBMDChildForm` alle möglichen Fenster abgeleitet werden. Die Entwickler haben für neue Fenster Kopiervorlagen, die einfach als Basis für neue Fenster hergenommen werden. Sie stellen eine Rumpf-Implementierung dar. Die Aufgabe des Entwicklers ist es, das Gerüst mit den Funktionalitäten zu füllen. Solche Vorlagen begünstigen tiefe Vererbungslinien.

Die Verteilung von *Model*, *View* und *Conroller* auf verschiedene abgeleitete Klassen erschwert das Verständnis und die Implementierung des Musters ist schwer nachzuvollziehen.

In diesem Zusammenhang steht auch die *Unit* `BMDDBGrid`. Viele andere *Units* verwenden diese `BMDDBGrid`. Sie dient zur Darstellung von Daten der Datenbank in Listenform. Diese *Unit* ist extrem groß und hat eine breite Schnittstelle.

Die *Unit* `BMDDBGrid` wird bereits in der Basisklasse benutzt und in den Kopiervorlagen konfiguriert (siehe Abbildung 2.5). Durch die Vererbung ist es schwierig zu erkennen, wo genau und wie die `BMDDBGrid` mit dem Datenmodell verstrickt wird. Die Funktion und die Anwendung ist nicht klar verständlich.

2.2.7 Action-Architekturmuster

Das *Action*-Architekturmuster wird auch in der Welt von Delphi verwendet und wickelt den Aufruf und das Abarbeiten von Funktionen ab. Es wird auch in der Klassenbibliothek von Delphi und bei der Erstellung von Benutzeroberflächen verwendet. *Actions* dienen zur Kapselung eines Funktionsdelegaten. Eine *Action* kann mit einer visuellen Komponente – zum Beispiel einer Schaltfläche – gestartet werden (Steward, 19.09.2008). In der Entwicklungsumgebung ist das die Standard-Vorgehensweise, um mit visuellen Komponenten eine Funktion zu starten.

Die *Unit* `Action` aus der Klassenbibliothek von Delphi unterstützt noch keine Mehrsprachigkeit. Darum haben die Entwickler die abgeleitete *Unit* `BMDAction` dahingehend erweiter. Die Methode *Execute* der *Unit* `Action` ist der Auslöser, der den Funktionsdelegaten startet. In `BMDAction` werden noch andere *Actions* die in der Liste *ActionList*

abgelegt sind. Diese *Actions* werden je nach Konfiguration vor oder nach dem Start der Funktion der eigentlichen *Action* ausgeführt.

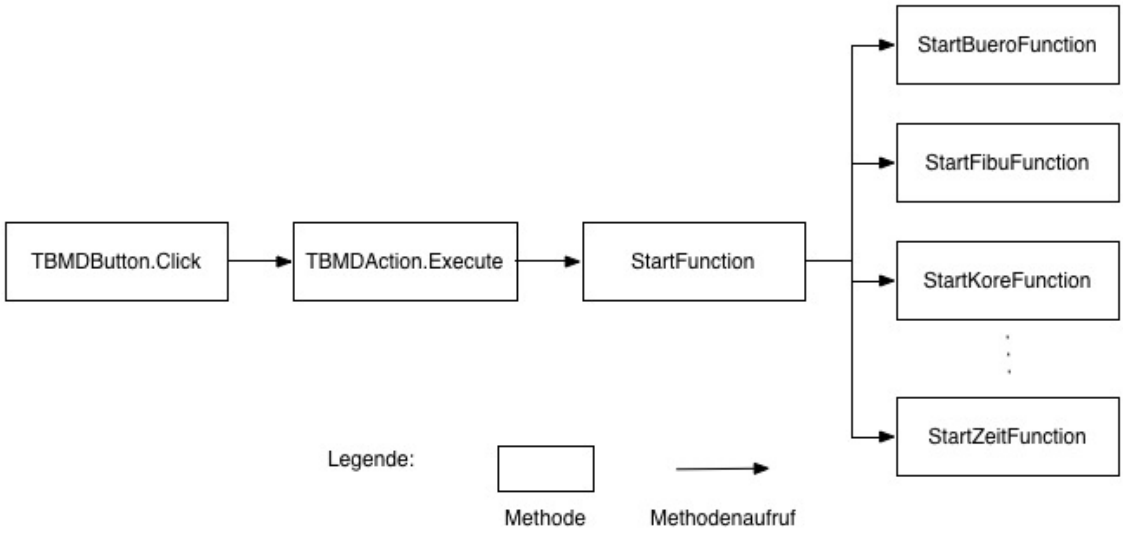

Abbildung 2.6: Schematische Darstellung der Implementierung von BMDAction

`BMDAction` ist ein zentraler Bestandteil des ERP-Programmes. Jede `BMDAction` besitzt eine *ID*, eine Bezeichnung und einem boolschen Feld ob die *Action* aktuell verfügbar ist. Alle *Actions*, die Funktionen der Pakete starten, sind in der Datenbank gespeichert. Beim Start des ERP-Programme für die Menüs und für Funktion *BMD-Quickstart* alle Funktionen aus der Datenbank geladen und aufgrund ihrer Zugehörigkeit zu den Paketen in einer hierarchischen *ActionList* abgelegt.

Wird eine Funktion gestartet, wird die *ID* an den zentralen statischen *Action*-Mechanismus übergeben, der aufgrund der ID weitere Methoden aufruft. Erst die aufgerufenen Methoden enthalten den Quelltext, um ein Fenster mit dem entsprechender Funktion zu starten.

2.2.8 In Textdokumenten ausgelagterte Bestandteile

Neben der ausführbaren Datei des ERP-Programmes gibt es noch über 3.000 Dateien für die Mehrsprachigkeit und Lokalisierung. In diesen Dateien sind die übersetzten Zeichenketten enthalten, die bei Bedarf aus den Dateien herausgelesen und angezeigt werden.

Weiters existieren für die verschiedenen Datenbanken (Oracle oder Microsoft SQL-Server) über 1.000 Textdateien, die SQL Anweisungen enthalten. Das ERP-Programm benötigt diese SQL Ausdrücke. Diese SQL Ausdrücke sind großteils `SELECT` Abfragen.

Das ERP-Programm liest diese Ausdrücke aus, verändert Parameter und schickt die Suche an die Datenbank. So sind die Abfragen nicht Teil des Anwendungscodes. Allerdings bedeuten diese Textdateien weitere Abhängigkeiten der Anwendung.

Bei einer Server-Installation werden diese Dateien nur auf dem Server abgelegt. Bei einer Einzelplatz-Installation müssen auch diese Dateien auch am Einzelplatz verfügbar sein.

2.2.9 Weitere Eigenschaften des ERP-Programmes

Die Sprache Delphi unterstützte bis zur Version 2009 keine generischen Typen und somit auch keine generischen Listen. Im Quelltext befinden sich auch keine allgemeinen *Collection*-Klassen. Stattdessen gibt es viele Listen, die einen bestimmten Objekttypen beinhalten. Diese *Collection*-Klassen sind auf den Anwendungsfall zugeschnitten und sind in ihren Methoden und Funktionen unterschiedlich.

Eine weitere Eigenschaft des ERP-Programmes sind die tiefen Vererbungslinien. Eine neue Funktion wird durch eine abgeleitete Klasse erzeugt, die zu einer weiteren *Blattklasse* in einem großen Vererbungsdiagramm wird.

Die Abbildung 2.7 zeigt einen Ausschnitt des Klassendiagramms zur Implementierung einer Liste aller Mitarbeiter. Die Abbildung 2.7 zeigt, dass diese Funktion eine abgeleitete Klasse einer anderen Funktion (hier einer allgemeinen Personenliste) ist.

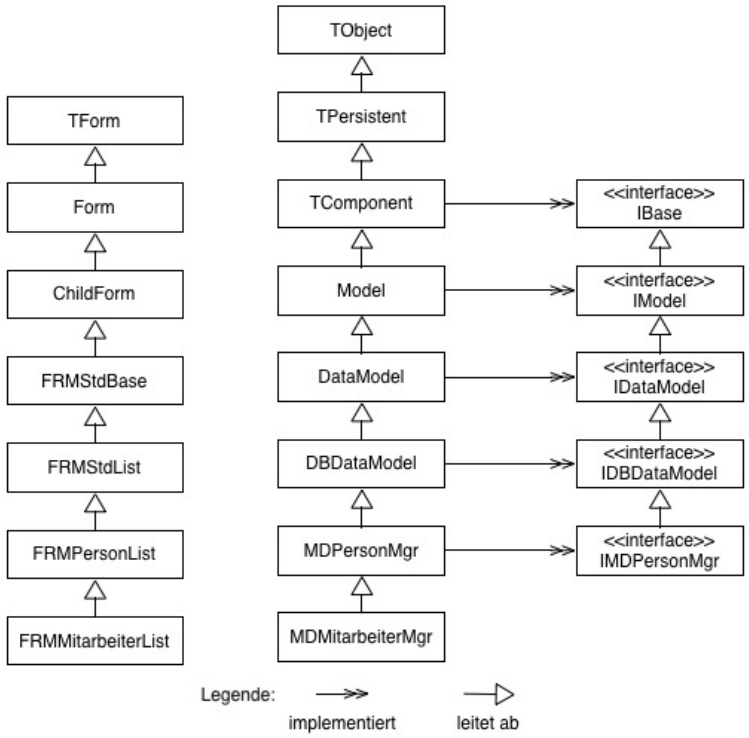

Abbildung 2.7: Tiefe Klassenhierarchie am Beispiel der Funktion Mitarbeiterverwaltung

Diese Architektur ermöglicht eine hohe lokale Wiederverwendung. Für die Fehlersuche kann dies große Probleme bedeuten. Die hohe Wiederverwendung führt zu einer Unübersichtlichekti.

Um eine Funktion zu verstehen, muss man auch die Klassen in der Vererbungshierarchie darüber kennen. Ohne Debugger ist es schwierig, Fehlerursachen zu finden, da die Methoden oft überschrieben werden. Aufrufe der überschriebenen Methode erschweren zusätzlich die Wartung des Quelltextes.

2.3 Zusammenfassung

Das ERP-Programm ist ein großes Softwareprodukt. Die Analyse und die Zerlegung der bestehenden Software ist umfangreich und schwierig. Ein Ziel für die Entwickler des ERP-Programmes ist die Reduktion der Komplexität, was nur durch die Zerlegung in kleinere Einheiten erzielt werden kann. Für die Entwickler des ERP-Programmes besteht der Wunsch, sowohl die Entwicklungsplattform als auch die Architektur auszutauschen.

In dieser Arbeit werden ausgewählte Funktionen aus dem Bereich Büro und Dokumentenmanagement näher untersucht. Es handelt sich dabei um das Paket `BMDOrganizer`. Es ist ein jüngerer Teil im ERP-Programm und weißt nur wenige Abhängigkeiten zu anderen Pakten ab. Dieses Paket wurde ausgewählt, um zu zeigen, wie eine Portierung von bestehendem Quelltext aussehen könnte. Daraus wurden Kernfeatures ausgewählt und diese als Komponente im Komponentenmodell von Plux.NET realisiert.

Kapitel 3

Problembeschreibung und Ziele

Die Softwarearchitektur schafft für die Entwickler und die Anwender des ERP-Programmes eine Reihe von Problemen. Dieses Kapitel beschreibt die Probleme zuerst aus Sicht der Anwender, danach aus Sicht der Entwickler.

3.1 Probleme aus Sicht der Anwender

3.1.1 Benutzerschnittstelle ist unübersichtlich

Die Benutzerschnittstelle der Anwendung zeigt in der Standardkonfiguration den vollen Leistungsumfang. Der typische Anwender hat Mühe, die Funktionen, die er benötigt, zu finden.

Diese Unübersichtlichkeit wirkt sich negativ bei der Schulung von neuen Anwendern aus. Bei der Benutzung des ERP-Programmes ist die Unübersichtlichkeit wieder ein Problem, weil die Anwender die Funktionen erst suchen müssen.

3.1.2 Mangelnde Anpassung an den Anwender und dessen Umfeld

Das ERP-Programm deckt als Komplettlösung für Betriebe alle wesentlichen Kundenszenarien ab. Trotzdem gibt es spezielle Kundenanforderungen, die die Entwickler des ERP-Programmes selbst nicht realisieren können.

Einerseits fehlen den Entwicklern des ERP-Programmes die Personalkapazitäten. Andererseits sind die speziellen Anforderungen eines Anwenders nicht auf andere Anwender übertragbar. Die Entwickler müssten diese Funktionen erneut anpassen. Weil die Anwendung als Monolith ausgeliefert wird, gibt es für Drittanbieter keine Möglichkeit, das ERP-Programm an die speziellen Anforderungen der Anwender anzupassen.

3.1.3 Akutalisierung von ausgewählten Teilen unmöglich

In großen Unternehmen tritt die Situation auf, dass eine Abteilung auf ein *Update* wartet, während eine anderere Abteilung eine neue Version zu diesem Zeitpunkt ablehnt. Die monolithische Auslieferung macht selektive *Updates* jedoch unmöglich. Kurze *Update*-Zyklen stellen sowohl die Entwickler als auch die Anwender des ERP-Programmes vor ein Problem.

3.2 Probleme aus Sicht der Entwickler

3.2.1 Wechselseitige Abhängigkeiten zwischen den Paketen

Das ERP-Programm ist in seiner Eigenschaft als Komplettlösung universell in einem Betrieb einsetzbar. Es ergeben sich für die Anwendung wechselseitige Abhängigkeiten zwischen den verschiedenen Paketen.

Diese Abhängigkeiten spiegeln sich direkt in der Komplexität der Anwendung wieder. Hinzu kommen noch Abhängigkeiten, die durch externe Bibliotheken entstanden sind.

3.2.2 Monolithische Architektur verhindert selektive Updates

Die monolithische Architektur des Systems erschwert für die Entwickler den Entwicklungsprozess. Die Suche auch das Beheben von Fehlern wird dadurch erschwert. Delphi als Entwicklungssprache hinkt aktuellen Konzepten der Softwareentwicklung hinterher. Im Bereich der Lohnverrechnung oder Buchhaltung gibt es Anpassungen, die durch gesetzliche Änderungen hervorgerufen werden. In anderen Paketen ist die Situation vergleichbar.

Für die Entwickler und die Anwender des ERP-Programmes besteht daher der Zwang die Anwendung auf den aktuellen Stand zu halten. Hier können kurze Updatezyklen eintreten. Bei diesen Update nehmen die Entwickler bei dem ERP-Programm keine Neuerungen sondern nur Anpassungen von Funktionen vor. Diese Änderungen werden dann die Anwender als neue Anwendung ausgeliefert.

3.2.3 Unzureichende Unterstützung durch Entwicklungsumgebung

Die Entwicklungsumgebung vom Delphi unterstützt die Entwickler nicht in dem Maß, wie man es von aktuellen Entwicklungsumgebungen gewohnt ist. Ein Punkt ist die Dauer der Übersetzung des Quelltextes.

Die Geschwindigkeit des Übersetzers von Delphi mit denen von anderen modernen Sprachen vergleichbar. Das Problem ist, dass immer der gesamte Quelltext in eine einzige ausführbare Datei übersetzt werden muss.

Um die Dauer zu reduzieren, verwirft der Übersetzer die Zwischenkompilate nicht. Der Übersetzer muss nicht wieder den gesamten Quelltext übersetzen, sondern nur die Teile, die durch die aktuelle Änderung betroffen sind.

Borland.Vcl.Themes.dcuil	Borland.Vcl.WinHelpViewer.dcuil
Borland.Vcl.ToolWin.dcuil	Borland.Vcl.WinInet.dcuil
Borland.Vcl.Types.dcuil	Borland.Vcl.WinSpooler.dcuil
Borland.Vcl.TypInfo.dcuil	Borland.Vcl.WinSvc.dcuil
Borland.Vcl.UrlMon.dcuil	Borland.Vcl.WinUtils.dcuil
Borland.Vcl.UxTheme.dcuil	Borland.Vcl.XPActnCtrls.dcuil
Borland.Vcl.ValEdit.dcuil	Borland.Vcl.XPMan.dcuil
Borland.Vcl.VarCmplx.dcuil	Borland.Vcl.XPStyleActnCtrls.dcuil
Borland.Vcl.VarConv.dcuil	Borland.Vcl.ActnBand.dcuil
Borland.Vcl.Variants.dcuil	Borland.Vcl.AdoDB.dcuil
Borland.Vcl.VDBConsts.dcuil	Borland.Vcl.BdeRtl.dcuil
Borland.Vcl.Win32.dcuil	Borland.Vcl.DbCtrl.dcuil
Borland.Vcl.Windows.dcuil	Borland.Vcl.VclDbRtl.dcuil

Tabelle 3.1: Auszug der Dateiliste von Zwischenkompilaten

Tabelle 3.1 einen Auzug aus den Zwischenkompilaten, die für die *VCL*-Klassenbibliotheken von Delphi erstellt wurden. Diese Zwischenkompilate verursachen im Entwicklungsbetrieb Schwierigkeiten. Diese Probleme können nur dadurch behoben werden, wenn die Zwischenkompilate gelöscht werden. Die Zwischenkompilate werden dann beim Linken zu einer einzigen ausführbaren Datei zusammengestellt.

Diese Datei enthält den gesamten Programmcode und wird als solche an die Anwender ausgeliefert. Aktuelle Sprachen und deren Übersetzer übersetzen die durch eine Änderung im Quelltext betroffenen Dateien neu und verwerfen die Zwischenkompilate nach dem Übersetzen. Heute ist es nicht mehr üblich, den gesamten Quelltext in eine einzige ausführbare Datei zu packen.

Um über viele Quelltextdateien den Überblick zu behalten, benötigt man neben Funktionen wie *Code-Completion*, *Syntax-Highlighting* auch Unterstützung bei der Modularisierung.

Moderne Entwicklungsumgebungen bieten hier ausgefeiltere Möglichkeiten. Sie sind in diesen Punkten mächtiger als es die aktuelle Entwicklungsumgebung ist. Auch der Übersetzungsprozess mit dem massiven Einsatz von Zwischenkompilaten verursacht große Probleme für die Entwickler.

Ein weiterer Grund, die Entwicklungsplattform zu wechseln, ist der Arbeitsmarkt. Es gibt für moderne Sprachen fertig ausgebildete Programmierer. Für Delphi ist der Markt wesentlich keiner, weshalb erst neue Entwickler ausgebildet werden müssen.

3.3 Ziele

3.3.1 Modularisierung des Quelltextes

Das Aufbrechen der Anwendung in Module erzeugt sowohl für die Entwickler als auch für die Anwender einen erheblichen Mehrwert. Das Ergebnis für beide Seiten sind stärkeres Eingehen auf die Bedürfnisse der Anwender, besser ausgebildete Anwender und vereinfachter Support im Fehlerfall. So könnte im Fall eines *Updates* nur der betreffende Teil an die Kunden ausgeliefert werden. Teile der Anwendung wären durch die Modularisierung getrennt versionierbar.

Das Aufbrechen der komplexen Geschäftsanwendung wird im Zerlegen der monolithischen Code-Basis weiter fortgesetzt und spiegelt sich in der Modularisierung der Anwendung auf Code-Ebene wieder. Den Entwicklern bringen die hier sich ergebenen Möglichkeiten große Vorteile und eine gesteigerte Produktivität:

- Kürzere Übersetzungszeiten
- Vereinfachte Architektur
- Vereinfachte Aufteilung der Pakete zu Entwicklerteams
- Isoliertere Entwicklung innerhalb eines Entwicklerteams
- Vereinfachte Zusammenarbeit zwischen den Entwicklerteams

All diese Punkte erleichtern dem Entwickler die eigentliche Arbeit. Einfache Dinge, wie das häufige Übersetzen der Programme helfen dem Entwickler, rascher Fehler zu finden und diese zu beseitigen. Die Vereinfachung der Architektur und somit die Reduktion der Komplexität erleichtern wiederum den Entwicklern die Arbeit.

3.3.2 Zerlegung des ERP-Programmes in Plug-Ins

Für die Entwickler gibt es Anforderungen, die erst mit einer in kleineren Einheiten zerlegten Anwendung möglich sind.

Es ergeben sich für die Kunden noch weitere Vorteile. So kann für die Anwender eine individuelle Anwendungen aufgebaut werden, die für die aktuelle Arbeitssituation angepasst ist. Eine Rekonfiguration der Anwendung zur Laufzeit erlaubt es dem Anwender die Arbeitssituation einfach umzuschalten. Der Anwender hat zu jeder Zeit eine minimale Konfiguration ohne die Anwendung beenden zu müssen.

Weiteres ergeben sich neue Möglichkeiten in der Zusammenarbeit mit Drittanbietern. So wäre es möglich, dass Drittanbieter oder auch die Kunden selbst die Anwendung mit eigenen Funktionen erweitern. Dadurch können Anwendungen zusammengestellt werden, die schlanker und kundenspezifischer sind.

3.3.3 Wechsel der Entwicklungsumgebung

Mit dem Wechsel der Entwicklungsplattform eröffnen sich für die Entwickler des ERP-Programms neue Möglichkeiten. Hinter dem Microsoft.NET Framework steht mit Microsoft ein großes und starkes Softwarehaus. Microsoft selbst benutzt für aktuelle und zukünftige Entwicklungen diese Plattform.

Es gibt Arbeitsmarkt fertig ausgebildete Entwickler für Microsoft.NET. Würde das ERP-Programm mit Microsoft.NET entwickelt, könnten neue Entwickler sofort eingesetzt werden. Im Gegensatz dazu müssen für Delphi erst Entwickler ausgebildet werden.

Für die Entwickler des ERP-Programms stellt der Wechsel der Entwicklungsumgebung keine Herausforderung dar. Dieser Wechsel und die damit verbunden Änderungen sind durchaus gewünscht.

Auch der Wechsel der Programmiersprache stellt für die Entwickler kein Problem dar. Ein Wechsel der Entwicklungsplattform bringt viele Vorteile:

- Technologisch Stand der Technik
- Bessere Integration
- Unterstütze Softwarearchitektur
- Vereinfachter Build-Prozess
- Verbesserte Unit-Test-Bedingungen

Kapitel 4

Portierung des Monolithen und Zerlegung in Plug-Ins

4.1 Lösungsansätze zur Weiterverwendung des Quelltextes

Ein wichtiger Punkt für die Entwickler ist die Erhaltung des Quelltextes. Diese Quelltextbibliothek stellt einen großen Wert dar. Sie soll mit möglichst geringem Änderungsaufwand portiert werden.

Dazu werden hier vier Varianten besprochen. Diese Varianten machen es möglich, dass Quelltext in Zukunft auf derMicrosoft.NET Plattform verfügbar ist.

4.1.1 Dynamic Link Libraries

Delphi kann normale native *Dynamic Link Libraries* (kurz: DLL) erstellen. Diese DLLs enthalten Maschinen-Code, der direkt auf auf der Rechner ausgeführt werden kann. Wird

Programmcode direkt auf dem Rechner und nicht von von einer virtuellen Maschine ausgeführt, nennt Mircosoft das *unmanaged*. Das bedeutet, dass solcher Programmcode ohne automatische Speicherbereinigung und auch ohne andere Mechanismen wie Versionierung, *Security* oder *Reflection* ausgeführt wird.

In solchen DLLs kann Programmcode ausgelagert werden, der über statische Aufrufe gestartet werden kann. Man kann hinter solchen Aufrufen beliebig komplexen Programm-Code ausführen.

Mit dieser Variante können komplexe Quelltexte herausgelöst und getrennt übersetzt werden. In diesen Spezialfällen kann man mit dieser Art der Portierung den Programmcode geringem Aufwand portieren.

Die Entwickler können parallel neben der Weiterentwicklung des ERP-Programms mit dem Herauslösen von Codeteilen beginnen. Ein zu grobes Zerlegen bringt keinen Vorteil. Man kann diese Art der Portierung für die Gesamtanwendung nicht benutzen, da hinter wenigen statischen Methoden sich die gesamte Funktionalität verbirgt und nicht verändert oder gesteuert werden kann. Hier ist diese Art der Portierung nicht sinnvoll. Im Anhang (Seite 93) finden sich zwei Quelltexte, die zeigen, wie mit Delphi eine DLL erzeugt werden kann.

4.1.2 Component Object Model

Das *Component Object Model* (kurz: COM) ist die nächste Möglichkeit, den bestehenden Quelltext für neue Entwicklungen zu konservieren. COM ist sprach-, versions- und plattformunabhängig. Man kann auf Objekte zugreifen und diese benutzen, da Objekte zusammen mit dem Code plattformunabhängig zur Verfügung gestellt werden. Die Wiederverwendung von Objekten ist unabhängig von einer speziellen Sprache.

Mit dem Microsoft.NET Framework ist es einfach, auf diesen Code zuzugreifen. Es wird von der Entwicklungsumgebung automatisch ein *Wrapper-Assembly* erstellt, um die Funktionalität in Microsoft.NET transparent zur Verfügung zu stellen.

Für das ERP-Programm bedeutet eine Portierung mittels COM einen weiteren Zwischenschritt, der mit erheblichem Aufwand verbunden ist. Diese Art der Portierung wird in dieser Arbeit nicht eingesetzt. Trotzdem zeigt der Anhang auf den Seiten 94 und 95 anhand von drei Quelltexten, wie mit Delphi ein Component Object Model erzeugt wird.

4.1.3 Portierung per Cross-Compiler

Jahn (2009) berschreibt einen *Cross-Compiler*, mit dem beliebiger Quelltext in einer Quellsprache in eine bestimmte Zielsprache übersetzt wird. Für die Sprachen Delphi und C# gibt es eine Implementierung.

Mit diesem *Cross-Compiler* wäre es möglich, den gesamten Delphi Quelltext automatisiert zu C# zu konvertieren. Diese Möglichkeit ist interessant, weil der gesamte bestehende Quelltext in einer neuen Programmiersprache verfügbar wäre, und ohne weiteren Aufwand benutzt werden könnte.

In der Praxis stellt sich dieser Prozess schwieriger dar. Diese Art der Portierung kann nicht parallel zur eigentlichen Entwicklung am ERP-Programm durchgeführt werden. Einige Konstrukte von Delphi sind nicht auf die Konstrukte des Microsoft.NET Frameworks überführbar. Der Quelltext des ERP-Programmes ist stark mit den Laufzeitbibliotheken von Delphi gekoppelt. Hier existiert auf der Seite vom Microsoft.NET Framework kein Gegenstück.

Die Entwickler müssten zusätzlich Quelltexte schrieben, der die Funktionalität der Delphi-Laufzeitbibilotheken auf die Funktionalität des Microsoft.NET Framework abbildet. Schwierig und aufwändig wird das bei den grafischen Benutzeroberflächen.

Wurde die Anwendung mit dem *Cross-Compiler* übersetzt, bleiben spätere Änderungen im Originalquelltext unbelassen. Das bedeutet, dass man solange auf dem Quelltext in der Quellsprache arbeiten muss, bis sich die Entwickler des ERP-Programms sicher sind, dass die Übersetzung per *Cross-Compiler* vollständig ist.

Werden Fehler im Ergebnis des *Cross-Compilers* gefunden, muss man den Quelltext in der Quellsprache anpassen und dann überprüfen, ob dieser vom *Cross-Compiler* richtig übersetzt wird.

Ist dieser Prozess einmal abgeschlossen, haben die Entwickler des ERP-Programms den gesamten Quelltext in einer zeitgemäßen Umgebung. Ab dann gilt nur mehr der Quelltext in der Zielsprache. Der Quelltext in der Quellsprache dient nur mehr als Dokumentation. Erst dann ist es vernünftig, am Quelltext in der Zielsprache zu arbeiten und dort neue Funktionen zu programmieren.

4.1.4 Portierung zu Delphi.NET

2002 präsentierte Borland Delphi.Net, um ebenfalls Anwendungen für die Microsoft.NET Plattform erstellen zu können. Die Sprache Delphi.NET ist eine Untermenge der Sprache Delphi.

Mit Delphi.NET ist es möglich, bestehende Delphi-Anwendungen zu einer Delphi.NET-Anwendung zu portieren. Bis zur Version 2006 der Entwicklungsumgebung von Delphi wurden *Assemblies* für das Microsoft.NET Framework 1.1 erstellt. Zusätzlich konnte über die Kommandozeile der Übersetzer instruiert werden, auch *Assemblies* für das Mi-

crosoft.NET Framework 2.0 zu erzeugen. Erst mit der Version 2007 der Entwicklungsumgebung ist es möglich, *Assemblies* für das Microsoft.NET Framework 2.0 zu erstellen.

Die Entwickler können diese Art der Portierung parallel zur eigentlichen Entwicklung durchführen, indem sie mit Übersetzungsdirektiven Quelltexte ein- oder ausblenden. So kann je nach Projekt der Übersetzer ein Delphi-Programm oder ein Delphi.NET-Programm erzeugen.

Ein Vorteil dieser Art der Portierung gegenüber den anderen liegt darin, dass nur ein Mal der Aufwand für die Portierung notwendig ist. Bei allen anderen Portierungen ist es notwendig, sowohl beim ursprünglichen Quelltext als auch in der Zielumgebung Anpassungen vorzunehmen.

4.2 Portierung des Quelltextes

Für die Portierung des Quelltextes wurde ein Mix aus den eben genannten Techniken herangezogen. Schon hier wurde darauf geachtet, dass der hier erstellte Programm-Code getrennt von einander übersetzbar ist. Im Quelltext fanden sich Abschnitte, die nur sehr schwierig portiert werden konnten. Für diese Teile wurde die Portierung zu Dynamic Link Libraries gewählt. Für den übrigen Teile wurde die direkte Portierung zu Delphi.NET gewählt, weil hier der Quelltext im großen Maß erhalten bleibt. Man kann zusätzlich mit einem CLRDebugger den übersetzten Code überprüfen.

Die verwendeten Techniken sollen nun erläutert werden.

4.2.1 Portierung von hardwarenahen Quelltexten

Im Quelltext des ERP-Programmes befinden sich Teile, die hardwarenahe programmiert sind. Dieser Quelltext lässt sich nur mit großem Aufwand portieren, da es hier Konstrukte wie Pointer-Arithmetik verwendet wurden. Diese Teile können statisch aufgerufen werden und eignen sich daher für die Portierung als Dynamic Link Library.

Der Quelltext für die BCD-Arithmetik und der Teil der Klassenbibliothek für den Datenbankzugriff wurden daher in zwei separaten Dateien für die Verwendung konserviert. Hier war eine Portierung zu einer anderen Sprache – selbst zu Delphi.NET – nur schwer oder mit großem Aufwand möglich.

4.2.2 Portierung des Quelltextes zu Delphi.NET

Für die Portierung des Quelltextes wurde die Entwicklungsumgebung von Delphi in der Version 2006 benutzt. Das bedeutete, dass die anfängliche Portierung hin zum Micosoft.NET Framework 1.1 durchgeführt wurde. Plux.NET benutzt als Plattform das Microsoft.NET Framework 2.0.

Um mit der Plug-In-Plattform Plux.NET kompatibel zu sein, war es notwendig, dass Microsoft.NET 2.0 *Assemblies* erstellt werden. Dies war über die Entwicklungsumgebung nicht direkt möglich. Eine mögliche Lösung waren *Wrapper Assemblies* im Microsoft.NET Framework 2.0, die die Funktionen der portierten *Assemblies* für Plux.NET verfügbar machen. Solche *Wrapper Assemblies* bedeuteten eine Codeverdopplung und wurden deshalb nicht eingesetzt.

Der Übersetzer von Delphi.NET in der Version 2006 kann mit einem Parameter auch *Assemblies* für das Microsoft.NET Framework 2.0 erstellen. Diese Funktion wird jedoch

nicht von der Entwicklungsumgebung unterstützt, sondern ist nur über die Kommandozeile verfügbar. Der Übersetzer erzeugt dabei nur *Assemblies* für das Microsoft.NET Framework 2.0, erweitert nicht die Funktionalität der Sprache um das Konzept von Generics, welches mit Micosoft.NET Framework 2.0 hinzugekommen sind.

Neben dem Quelltext des ERP-Programms mussten auch die Laufzeitbibliotheken von Delphi portiert werden. Zu Beginn dieser Arbeit lagen diese Laufzeitbibliotheken nur für das Microsoft.NET Framework 1.1 vor. Der Aufwand dafür war nicht unerheblich, dafür war es gewährleistet, dass sich der portierte Programm-Code zum ursprünglichen Programm-Code nahezu identisch verhält.

Die getroffene Entscheidung hatte auch weitreichende negative Auswirkungen. So war es nicht mehr möglich, den portierten Quelltext über die Entwicklungsumgebung von Borland zu entwickeln. Das bedeutete somit den kompletten Wegfall der Unterstützung der Entwicklungsumgebung wie *Syntax-Highlighting*, *Code-Completion*, *Debugging* und der gleichen. Stattdessen mussten andere Werkzeuge gefunden werden um diesen Mangel auszugleichen. Einfache Texteditoren konnten die Funktionalität des Entwicklungsumgebung nachbilden. Für das *Debugging* wurde der CLR-Debugger aus dem Microsoft.NET Framework benutzt. Dieser Schritt wurde trotzdem gemacht, weil er für die Zukunft vielversprechender zu sein schien.

Mit der Entwicklungsumgebung von Borland für Delphi 2007 war es erstmals möglich, *Assemblies* für das .Net 2.0 Framework zu erstellen. Allerdings wurde in dieser Version der Entwicklungsumgebung ebenfalls die Sprache Delphi erweitert und auch die Visual Class Library verändert. Daher wäre es notwendig gewesen, den Quelltext auf Delphi 2007 zu bringen um ihn dann auf Delphi.NET zu portieren. Daher wurde der Quelltext so belassen.

4.2.3 Portierung der Delphi-Klassenbibiliotheken

Zu Beginn der Portierung waren die Klassenbibliotheken von Delphi.NET nur für Microsoft.NET Framework 1.1 verfügbar. Um die Anwendung für das Microsoft.NET Framework 2.0 zu portieren, war es wünschenswert, dass diese Klassenbibliotheken ebenfalls für das Microsoft.NET Framework 2.0 portiert werden. Dies bedingt eine eigene Übersetzung der Bibliotheken wie bei Wischnewski (19.04.2008) beschrieben. Durch die Portierung der Delphi-Klassenbibliotheken konnten *Wrapper*-Klassen eingespart werden.

Leider hatte diese Entscheidung auch gravierende Nachteile. Erstens verliert man damit die *Release*-Fähigkeit zu neueren Klassenbibliotheken. Andererseits erzeugt man dadurch eine starke Abhängigkeit auf das installierteMicrosoft.NET Framework. Der Übersetzer von Delphi.NET erzeugt für das Microsoft.NET Framework eigene Zwischenkompilate. Ändert sich dasMicrosoft.NET Framework durch ein Update, müssen die Delphi-Klassenbibliotheken aufgrund der veränderten Zwischenkompilate ebenfalls erneut übersetzt werden. Macht man das nicht, kann man nicht Programm-Code übersetzen, der die Delphi-Klassenbibliothek benutzt. Das Problem sind die Zwischenkompilate, die nicht mehr mit den *Assemblies* aus demMicrosoft.NET Framework zusammenpassen.

Dieser Fehler zeigt sich nur, wenn man Quelltext neu übersetzen will, der diese Delphi Klassenbibliotheken benutzt. Die Fehlermeldung des Übersetzers ist nicht aussagekräftig. Dies war die Ursache dafür, dass es nach der Portierung immer wieder Schwierigkeiten gab, Teile neu zu übersetzen.

Im Hinblick auf die Version 2007 von Delphi.NET ist die Portierung der Delphi-Klassenbibliothek nicht mehr notwendig, da dieser Version von Delphi.NET liegt die Delphi-Klassenbibliothek schon für das Microsoft.NET Framework 2.0 vorliegt. Allerdings war ohne die Unterstützung der Entwicklungsumgebung nur mit großem Aufwand

machbar gewesen, die neuen Klassenbibliotheken, die mit Delphi 2007 erneuert und auch verändert wurden, für diese Arbeit zu verwenden.

In dieser Arbeit wurde nur die Machbarkeit geprüft. Daher ist es verschmerzbar, dass nicht die Laufzeitbibliotheken von Delphi.NET 2007 verwendet wurden. Für eine Portierung des Produktionscodes würde ohnehin mit Delphi 2007 gearbeitet werden.

Mit Delphi 2009 ist das ganze Thema obsolet geworden. Alle Klassenbibliotheken, die geschaffen wurden, um vorhandenen Delphi Quelltext mit Delphi.NET sanft zu migrieren, wurden nicht mehr weiter geführt.

Eigentlich wurde Delphi.NET komplett eingestellt und anstatt dessen Delphi Prism veröffentlicht, das zwar einen pascalartigen Syntax hat, jedoch gravierende Unterschiede zu Delphi oder sogar Delphi.NET aufweist, wenn es um die Grammatik geht.

Delphi Prism nimmt auch nicht mehr die Klassenbibliotheken von Delphi sondern baut komplett auf die Klassenbibliothek von Microsoft.NET auf.

4.2.4 Portierung externer Klassenbibliotheken

Im Quelltext des ERP-Programmes befinden sich externe Klassenbibliotheken. Diese Klassenbibliotheken regeln den Zugriff zu den Datenbanken von Microsoft SQL-Server und Oracle. Diese Bibliotheken sind eng mit dem Produktionscode gekoppelt.

Die externen Bibliotheken waren früher im Programm-Code des ERP-Programmes enthalten. Durch die Portierung und Zerlegung wurden diese Bibliotheken herausgelöst.

Von den Herstellern dieser Klassenbibliotheken gibt es neuere Versionen, die ebenfalls für das Microsoft.NET Framework 2.0 verfügbar sind. Weil die Entwickler des ERP-

Programmes den Quelltext dieser Bibliotheken verändert haben, konnten nicht die aktuellen Klassenbibliotheken verwendet werden.

4.2.5 Einführung von Namespaces

Der Quelltext des ERP-Programmes ist umfangreich. Um nach der Portierung nicht ein großes, schwach strukturiertes *Assembly* zu erhalten, müssen die einzelnen Klassen zu *Namespaces* gruppiert werden.

Es gibt eine Zuordnung der einzelnen *Units* zu Paketen. Diese ist nicht explizit sondern implizit durch den Namen der *Unit* bzw. den Namen der Quelltextdatei gegeben. Für die Aufteilung ist es notwendig, sich einen Plan zurechtzulegen, nach welchem Schema die resultierenden Klassen in einem *Namespace* aufgeteilt werden. Der Quelltext wurde auf 16 verschiedene *Namespaces* aufgeteilt.

Delphi selbst kennt keine *Namespaces*. Stattdessen gibt es das Konzept von *Units*. Dieses Konzept wurde in Delphi.NET ebenfalls übernommen. *Units* haben ähnlich wie *Namespaces* die Aufgabe, zusammengehörigen Quelltext zu gruppieren. Eine *Unit* kann wie ein *Namespace* mehrere Klassen enthalten (Doberenz und Gewinnus, 2005).

Der wesentlichste Unterschied zwischen Klassen in Delphi.NET und beispielsweise C# ist, dass Klassen in Delphi.NET keine statischen Methoden haben können. In Delphi.NET sind statische Methoden Teil der *Unit*.

Um diese übersetzen zu können, erstellt der Übersetzer von Delphi.NET für jede Unit zwei *Namespaces*. Tabelle 4.1 zeigt die verschiedenen *Namespaces*. Der erste *Namespace* lautet gleich wie die Unit enthält alle Klassen. Der zweite *Namespace* enthält eine Klasse, die alle statischen Methoden der *Unit* enthält. Der Name des *Namespaces* ist wie der

NCTCS.Attributes	NCTCS.Attributes.Units
NCTCS.Attributes.Forms	NCTCS.Attributes.Forms.Units
NCTCS.Base	NCTCS.Base.Units
NCTCS.Core	NCTCS.Core.Units
NCTCS.Database	NCTCS.Database.Units
NCTCS.Database.Forms	NCTCS.Database.Forms.Units
NCTCS.Forms	NCTCS.Forms.Units
NCTCS.Forms.StdDialogs	NCTCS.Forms.StdDialogs.Units
NCTCS.Forms.StdForms	NCTCS.Forms.StdForms.Units
NCTCS.Formulas	NCTCS.Formulas.Units
NCTCS.Formulas.Forms	NCTCS.Formulas.Forms.Units
NCTCS.Models	NCTCS.Models.Units
NCTCS.Models.Forms	NCTCS.Models.Forms.Units
NCTCS.Network	NCTCS.Network.Units
NCTCS.Packages	NCTCS.Packages.Units
NCTCS.Packages.Forms	NCTCS.Packages.Forms.Units

Tabelle 4.1: Ergebnis der Aufteilung in Namespaces

Name der *Unit* mit dem Postfix `Units`. Darum enthält die Abbildung 4.1 32 verschiedene *Namespaces* obwohl ursprünglich der Quelltext in nur 16 *Namespaces* eingeteilt wurde.

Der Übersetzer von Delphi.NET fügt den zusätzlichen *Namespace* `$compiler_internal$` hinzu. Mit diesem *Namespace* wird unter anderem dem Übersetzer signalisiert, dass es sich um ein Microsoft.NET *Assembly* handelt, welches mit Delphi.NET erstellt wurde.

Aufgrund dieses *Namespaces* erkennt der Übersetzer, dass er kein weiteres Zwischenkompilat erzeugen darf, sondern die bestehenden Zwischenkompilate aus vorhergegangen Übersetzungen wiederverwenden muss.

Für Microsoft.NET *Assemblies* von anderen Übersetzern (zum Beispiel vom C#-Übersetzer) erzeugt der Delphi.NET-Übersetzer Zwischenkompilate, damit diese in Delphi.NET verwendet werden können.

4.2.6 Aufteilung getrennt übersetzbare Einheiten

Wenn man von Zerlegung einer Anwendung spricht, sind zwei Begriffe ganz wichtig – Kopplung und Kohäsion. Unter gutem Software-Entwurf versteht man, wenn ein Modul über eine hohe Kohäsion und eine schwache Kopplung verfügt.

Zu Beginn wurde die Kernfunktionalität des ERP-Programmes in ein *Assembly* gepackt, welches den Namen NTCS.CommonLib.Net.dll trägt. Darin sind alle Klassen enthalten, die den *BMD-Tools* zuzurechnen sind. Dieses *Assembly* enthält keine direkte Funktionalität des Produkts. Die portierten Funktionen wurden in das *Assembly* OrganizerLib.dll gegeben. Die portierten Funktionen stammen aus dem Paket `BMDOrganizer`.

Das *Assembly* NTCS.CommonLib.Net.dll weist eine hohe Kopplung auf. Darum wurde versucht, in weiteren Schritten NTCS.Commonlib.Net.dll noch weiter zu zerteilen. Abbildung 4.1 zeigt Teile des Quelltextes, die sich für eine weitere Zerlegung eignen.

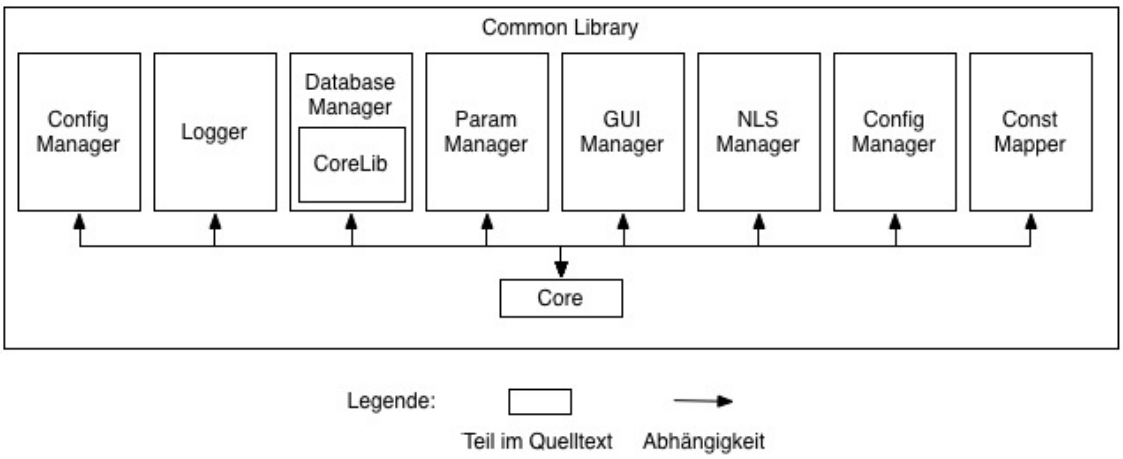

Abbildung 4.1: Schematische Zerlegung der Komponenten des ERP-Programmes

Besonderes Interesse galt dem *DatabaseManager*. Dieser Bereich des Quelltextes kapselt den Zugriff auf die Datenbanken. Es gibt im Quelltext des ERP-Programmes eine

zyklische Abhängigkeit zwischen den externen Bibliotheken von *CoreLib* und der Unit *BMDQuery* im Produktions-Code des ERP-Programmes. Um diesen Bereich getrennt übersetzen zu können, war es notwendig in NTCS.CommonLib.Net.dll Interfaces einzuarbeiten. Diese Interfaces werden dann vom NTCS.DatabaseManager implementiert. Hier gibt es keine zyklischen Abhängigkeiten mehr.

Erst durch die Auflösung der zyklischen Abhängigkeiten wie in Abbildung 4.2, wird eine getrennte Übersetzung der einzelnen *Assemblies* erzielt. Alle *Assemblies* haben eine Abhängigkeit auf das *Assembly* NTCS.CommonLib.Net.dll, welches als erstes übersetzt werden muss. Danach folgen die externen Bibliotheken und das *Assembly* NTCS.DatabaseManager.dll.

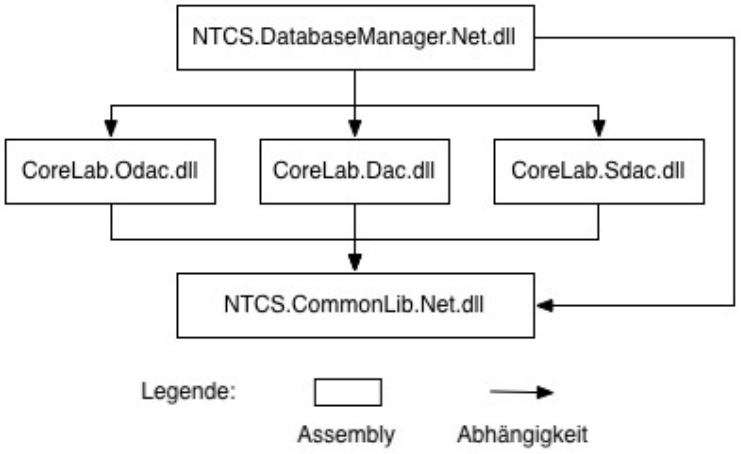

Abbildung 4.2: Darstellung Abhängigkeiten zwischen den Assemblies des ERP-Programmes und CoreLab

Das *Assembly* NTCS.DatabaseManager.dll ist relativ klein und hat nur wenige Klassen. Es war aber notwendig, um getrennt übersetzbare Einheiten zu schaffen. Die zyklischen Abhängigkeiten konnten so aufgelöst werden.

Tabelle 4.2 zeigt das Resultat der Zerlegung von NTCS.CommonLib.Net.dll. Es sind 14 *Assemblies* entstanden, die nur Produktions-Code des ERP-Programmes enthalten. Weiters gibt es noch Verzeichnisse mit den externen Bibliotheken für den Datenbankzugriff und die Laufzeit- und GUI-Bibliotheken von Delphi.

Dateiname	Größe
INI	
LOG	
NTCS.Calculator.dll	49 KB
NTCS.CommonLib.NET.dll	7.993 KB
NTCS.ConfigManager.dll	35 KB
NTCS.ConstMapper.dll	16 KB
NTCS.DatabaseManager.dll	452 KB
NTCS.ExceptionHandler.dll	33 KB
NTCS.FormatManager.dll	99 KB
NTCS.GUIManager.dll	23 KB
NTCS.Logger.dll	18 KB
NTCS.NativeCommonLib.NET.dll	438 KB
NTCS.NativeDatabaseManager.dll	80 KB
NTCS.NLSManager.dll	65 KB
NTCS.ParamManager.dll	182 KB
OrganizerLib.dll	5660 KB
BMD.ini	2 KB
BMDGLOBAL.ini	1 KB
BMDConstAttrID.xml	1353 KB

Tabelle 4.2: Dateiübersicht der getrennt übersetzten Assemblies des ERP-Programmes

Für die spätere Verwendung dieser portierten Anwendung liegen diese *Assemblies* in einem Unterordner der Plug-In Plattform und sind so für den *Assembly-Loader* verfügbar. In Summe sind dies 48 *Assemblies* und Konfigurationsdateien in 7 verschiedenen Ordnern.

Mit dieser Aufteilung wurden getrennt übersetzbare Einheiten geschaffen. Um eine Aussage treffen zu können ob ein gutes Maß an Kopplung und Kohäsion erreicht ist, be-

darf es einer Werkzeugunterstützung. Eine Abschätzung ohne Werkzeugunterstützung ist schwierig, weil die Anwendung einerseits groß ist und andererseits die Zyklen zwischen den Modulen eine Abschätzung unmöglich machen.

In der Entwicklungsumgebung von Delphi gibt es dazu direkt keine Werkzeugunterstützung. Sangal u. a. (2005) beschreiben eine Möglichkeit um Quelltexte auf ihre Abhänigkeit zueinander zu untersuchen. Untersucht wird dabei, wie oft Klassen eines *Namespace* von den Klassen eines anderen *Namespaces* benutzt werden. Da der Quelltext nach der Portierung auch als übersetzte Microsoft.NET *Assemblies* vorliegt, konnten Aussagen über das *Assembly* NTCS.CommonLib.Net.dll mit der *Dependency Structure Matrix*-Erweiterung getroffen werden.

	DatabaseManager	Network	Formulas	Packages	Models	Database	Forms	Attributes	Core	Base
DatabaseManager										
Network					8	51				
Formulas					52		64	19		
Packages			2		7			8		
Models	49		919	116		139	1296	886	49	5
Database	779		249	149	688		1174	682	31	2
Forms	17		166		137	199		1019	13	2
Attributes	36	1	702	141	172	117	3065		30	5
Core	1621	173	542	156	1298	1123	3291	3469		452
Base	345	44	432	84	851	579	3047	3887	61	

Abbildung 4.3: Dependeny Structure Matrix des Assembly NTCS.CommonLib.Net.dll

In Abbildung 4.3 sieht man die Anzahl von Verwendungen eines *Namespaces* in einem anderen. Graue Felder markieren zyklische Verwendungen. Diese Abbildung zeigt, dass NTCS.CommonLib.Net.dll ohne erheblichen Arbeitsaufwand nicht weiter zerlegbar ist.

Im Abbildung 4.4 zeigt die Abhängigkeiten des Programm-Codes gruppiert nach dem *Namespace*. In dieser Abbildung ist ebenfalls der *Namespace* Organizer zu sehen. Es beinhaltet die portierten Funktionen des Pakets `BMDOrganizer`.

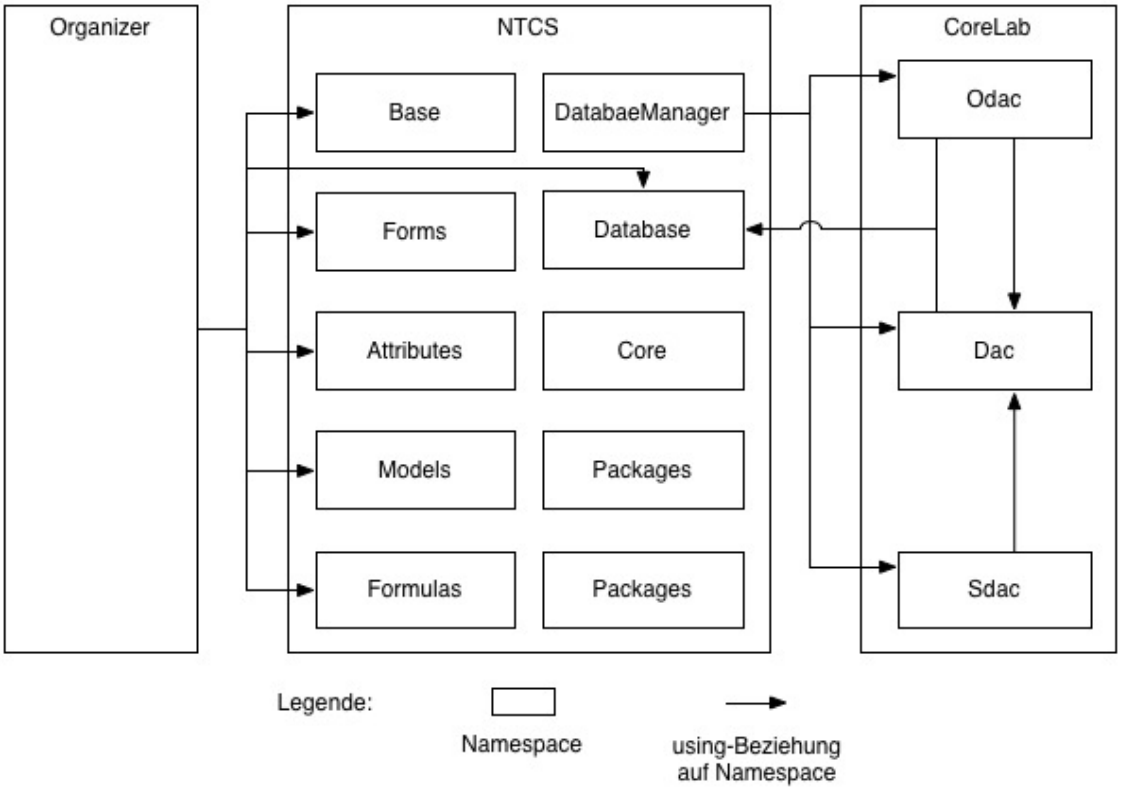

Abbildung 4.4: Abhängigkeiten zwischen den Namespaces im ERP-Programmes

4.2.7 Benutzte Hilfsmittel

Der Übersetzer von Delphi.NET kann *Assemblies* für das Microsoft.NET Framework 2.0 erzeugen. Das geschieht über den Parameter `--clrversion:v2.0.50727`. Dieser Parameter exisitiert nur als Kommandozeilenparameter und kann nicht über die Entwicklungsumgebung gesetzt werden. Um trotzdem den Quelltext verhältnismäßig einfach zu übersetzen, wurden die Aufrufe des Übersetzers durch Stapeldateien.

Listing 4.1: Stapeldatei zur Übersetzung der Basisklassen

```
del NTCS.NativeCommonLib.NET.cfg
dcc32 -H- -W- -M -EC:\work\output -LEC:\work\output -LNC:\
   work\dcu
      -NOC:\work\dcu -UK:\NTCS\libraries\systools
      -DNATIVEDOTNETLIB NTCS.NativeCommonLib.NET.dpr
del NTCS.CommonLib.NET.cfg
dccil -V --clrversion:v2.0.50727 --no-config -u"D:\vcl2\lib"
      -nsBorland.Vcl -M -H- -W-SYMBOL_PLATFORM
         -W-UNIT_PLATFORM
      -w-UNSAFE_CODE -w-SYMBOL_DEPRECATED -LEC:\work\output
      -LNC:\work\dcu -NOC:\work\dcu
      -UK:\NTCS\libraries\tms NTCS.CommonLib.NET.dpk
```

Das Quelltextbeispiel 4.1 zeigt den Inhalt der Stapeldatei, die zur Übersetzung der Dynamic Link Library NTCS.NativeCommonLib.Net.dll und des *Assemblies* NTCS.CommonLib.Net.dll dienen.

Die einzelnen Parameter sind zum Teil nur ungenau beschrieben und spezifiziert. Mit diesen Stapeldateien werden sowohl die schwer portierbaren Quelltexte als auch die Microsoft.NET *Assemblies* übersetzt. Neben den Quelltexten des ERP-Programms werden auch die Laufzeitbibliotheken mit diesen Stapeldateien übersetzt.

4.2.8 Ursachen für den hohen Aufwand beim Portieren des Quelltextes

Die Portierung zu Delphi.NET brachte neben den beschriebenen Vorteilen auch die Nachteile, dass es keine Unterstützung durch die Entwicklungsumgebung mehr gab. Das bedeutete in erster Linie keine Unterstützung beim Übersetzen und keine Unterstützung bei der Organisation der Zwischenkompilate.

Dies wirkte sich negativ aus, da durch normale *Windows-Updates* einige *Assemblies* des Microsoft.NET Frameworks verändert wurden. Danach war es nicht mehr möglich, den Delphi.NET Quelltext neu zu übersetzen. Die ausgegebene Fehlermeldung des Übersetzers war nicht verständlich.

Erst nachdem die selbst portierten Standardbibliotheken neu übersetzt wurden, war der Fehler behoben. Als späterer Grund zeigte sich, dass die Zwischenkompilate für die Bibliotheken des Microsoft.NET Frameworks nach einem Update nicht mehr mit den installierten *Assemblies* des Microsoft.NET Frameworks kompatibel waren.

Dieser Fehler war nicht aus der Fehlermeldung des Übersetzers erkennbar. Weiters stand das zu übersetzende Quelltextstück in keiner Weise mit den veränderten *Assemblies* des Microsoft.NET Frameworks in Verbindung. Diese Vorfälle zeigten, dass diese Stapeldateien den gesamten Prozess des Übersetzens rudimentär unterstützen und Fehlerquellen hinzukommen.

Die verschiedenen und unzureichend dokumentierten Parameter, mit denen der Übersetzer instruiert wird, machen es zusätzlich schwierig, Fehler zu finden. Die Meldungen des Übersetzers sind kryptisch und nicht verständlich. Diese Meldungen haben bei der Arbeit erheblich Zeit in Anspruch genommen.

Ein weiterer Nachteil aus dem Entschluss, Assemblies für das Microsoft.NET Framework 2.0 zu erstellten war, dass es ab dann auch keine Unterstützung durch die Entwicklungsumgebung mehr gab. Das bedeutete keine Unterstützung bei der Entwicklung durch *Code-Completion*, *Syntax-Highlighting*, *Debugging* oder dergleichen. Neuer Quelltext für die Erstellung von Plug-Ins konnte in Delphi nur mit normalen Editoren geschrieben werden.

Code-Completion ist bei einer großen Standardbibliothek und häufig abgeleiteten Klassen hilfreich. Es kann nicht durch das Nachschauen in den betreffenden Quelltextdateien kompensiert werden. Das Fehlen von *Syntax-Highlighting* war bei langen Methoden störend.

Bei den großen Quelltextdateien war es schwierig, sich zu orientieren. Eine Entwicklungsumgebung zeigt alle vorhandenen Methoden und Funktionen auf. Ein Entwickler kann sich dadurch in den Quelltextdateien zurechtfinden. Die Suchfunktion eines Editors kann diese Unterstützung nur mangelhaft ersetzen.

Weitere Probleme warf die Verwendung der GUI-Biblbiothek *Visual Class Library* von Delphi.NET auf. *VCL*-Fenster und *Windows.Forms* Fenster sind nicht kompatibel. In Delphi kann nur das erste geöffnete Fenster ein *MDI*-Fenster sein. Alle weiteren *VCL*-Fenster können nur normale Fenster sein. Plux.NET öffnet bereits beim Start ein *Windows.Forms* Fenster. Daher kann kein weiters *MDI*-Fenster der *VCL* erstellt werden.

Diese Einschränkung gibt es bei *Windows.Forms* Fenstern nicht, daher fiel die Entscheidung, das *MDI*-Fenster in *Windows.Forms* zu entwickeln. Die Funktionen des ERP-Programms öffnen aber und die *VCL*-Fenster. Deshalb wurde versucht, mittels Wrapperklassen die *VCL*-Fenster in das *Windows.Forms MDI*-Fenster einzubinden.

Dazu musste das *VCL*-Fenster unsichtbar erzeugt werden und danach die Referenz des Fensters in das *MDI*-Fenster, hineingezogen werden. Dies konnte mit *Win32*-Befehlen bewerkstelligt werden. Diese Lösung funktioniert, kann aber keines Falls als sauber gelten.

4.3 Anpassen an ein Komponentenmodell

Mit dem Abschluss der Portierung und Zerteilung sind für die Entwickler zwei Ziele erfüllt. So gibt es keine Abhängigkeit mehr zu der Entwicklungsumgebung von Delphi und die monolithische Codebasis ist aufgebrochen und kann weiter zerlegt werden.

Eine Reihe von wichtigen Zielen sind noch offen. Mit der portierten Anwendung kann keine personalisierte und rollenspezifische Anwendung erzeugt werden. Das Problem der Erweiterbarkeit der Anwendung ist ebenfalls noch nicht gelöst.

Um die aufgestellten Ziele zu erreichen, muss eine Umgebung gefunden werden, bei der die einzelnen Komponenten zur Laufzeit zu einer neuen Anwendung zusammengestellt werden können. Gruhn und Thiel (2000) beschreiben die Anforderungen an eine solche Umgebung:

> "Ein Komponentenmodell legt einen Rahmen für die Entwicklung und Ausführung von Komponenten fest, der strukturelle Anforderungen hinsichtlich Verknüpfungs- bzw. Kompositionsmöglichkeiten sowie verhaltensorientierte Anforderungen hinsichtlich Kollaborationsmöglichkeiten an die Komponenten stellt. Darüber hinaus wird durch ein Komponentenmodell eine Infrastruktur angeboten, die häufig benötigte Mechanismen wie Verteilung, Persistenz, Nachrichtenaustausch, Sicherheit und Versionierung implementieren kann."

Es gibt für solche Zwecke in der Welt von Java einige Beispiele wie zum Beispiel die OSGI-Implementierung Equinox oder Netbeans. Für dieMicrosoft.NET Plattform gab es nichts Vergleichbares. Deshalb wurde am Christian Doppler Labor von Mag. Reinhard Wolfinger die Plug-In-Plattform Plux.NET entwickelt. Bei Plux.NET handelt es sich um ein solches Komponentenmodell.

4.3.1 Software-Komponenten

Das Komponenten-Modell bietet ausschließlich Dienstleistungen für Software-Komponenten an. Die gesamte Funktionalität der Anwendung findet sich in den Software-Komponenten. Diese Software-Komponenten müssen gewisse Anforderungen wie folgend von Heineman und Councill (2001) beschrieben wird:

> "Eine Software-Komponente ist ein Software-Element, das konform zu einem Komponentenmodell ist und gemäß einem Composition Standard ohne Änderungen mit anderen Komponenten verknüpft und ausgeführt werden kann."

In Plux.NET heißen solche Software-Komponenten *Extensions*. In diesen *Extensions* befindet sich die gesamte Logik der Programme. Plux.NET verbindet die *Extensions* miteinander und stellt den Nachrichtenaustausch zur Verfügung.

Eine Extension ist in der Diktion von Plux.NET ein für den Anwender wahrnehmbarer Teil der Software. Die Teile der Benutzeroberfläche gehören eindeutig zu dieser Definition. In der Praxis hat es sich auch gezeigt, dass Datenmodelle oder Elemente zur Steuerung des Kontrollflusses (*Actions*) sich als Extension eignen.

Die Mechanismen zur Verknüpfung der *Extensions* sind *Slots* und *Plugs*. Um die Schnittstelle zwischen den *Extensions* typsicher zu definieren, wird für jeden *Slot* ein Kontrakt definiert.

Listing 4.2: Kontrakt von BMD.Plux.MainWindow

```
using Plux;
[SlotDefinition("BMD.Plux.MainWindow")]
public interface IBMDPluxMainWindow {
  void Show();
}
```

Das Interface `IBMDPluxMainWindow` definiert alle Methoden die eine Extension aufweisen muss, um sich in einen *Slot* mit dem Namen BMD.Plux.MainWindow anstecken zu können.

Listing 4.3: Implementierung von MainWindow

```
using System;
using Plux;
[Extension("BMD.Plux.MainWindow")]
[Plug("BMD.Plux.MainWindow")]
public class BMDPluxMainWindow : TBMDMainWindow,
   IBMDPluxMainWindow {
  public void Show() {
    setVisible(true);
  }
}
```

Die Klasse `BMDPluxMainWindow` leitet von der Klasse aus der portierten Klassenbibliothek des ERP-Programmes ab und implementiert das Interface `IBMDPluxMainWindow`.

Listing 4.4: BMD.Plux.Core als Anbieter von BMD.Plux.MainWindow

```
[Extension("BMD.Plux.Core")]
[Plug("Startup")]
[Slot("BMD.Plux.MainWindow", OnPlugged="SetMainWindow")]
public class BMDPluxCore: IStartup {

  IBMDPluxMainWindow mainwindow = null;

  public void Run() {
  }
  public void SetMainWindow(object s, PlugEventArgs args) {
```

```
    mainwindow = (IBMDPluxMainWindow) args.Extension;
    mainwindow.Show();
  }
}
```

Schließlich gibt es die Klasse `BMDPluxCore`, die als *Host-Extension* den Slot `BMD.Plux.MainWindow` anbietet. Jede *Contributor-Extension* implementiert das Interface `IBMDPluxMainWindow` und erfüllt noch weitere Eigenschaften, die bei der Definition des Slots im Kontrakt spezifiziert sind.

Plux.NET überprüft diese Eigenschaften und meldet die *Contributor-Extension* bei der *Host-Extension* an, indem die Methode bei der *Host-Extension* aufgerufen wird, die im Attribut *Slot* definiert wurde.

Dieses Beispiel zeigt den einfachsten Fall, um eine Anwendung mit Plux.NET herzustellen. Nach diesem Prinzip lassen sich beliebig komplexe Anwendungen erstellen.

Die hier gezeigten Parameter von *Extensions* und *Slots* sind nur eine grobe Skizzierung der Möglichkeiten, die Plux.NET bietet. Mit Plux.NET ist es möglich, *Extensions* als *Singleton* zu definieren, damit diese nur einmal existieren. Oder man kann gewisse *Extensions* ablehnen und erst anstecken, wenn gewisse Anforderungen erfüllt sind.

Weiters ist es möglich, *Slots* so zu definieren, dass nur eine einzige *Extension* darin eingesteckt werden kann. Die Möglichkeiten sind vielfältig und können in Wolfinger (2010) nachgelesen werden.

4.3.2 Generizität von Extensions

Wie in den Zitaten von Gruhn und Thiel (2000) und Heineman und Councill (2001) angeführt sollen Komponenten unabhängig von einander zusammengestellt werden können,

ohne dass es notwendig ist, die Komponenten aufgrund der Zusammenstellung intern zu verändern. Dies ist eine wichtige Tatsache, die bei der Entwicklung der *Extensions* bedacht werden muss. Hierzu ein Beispiel:

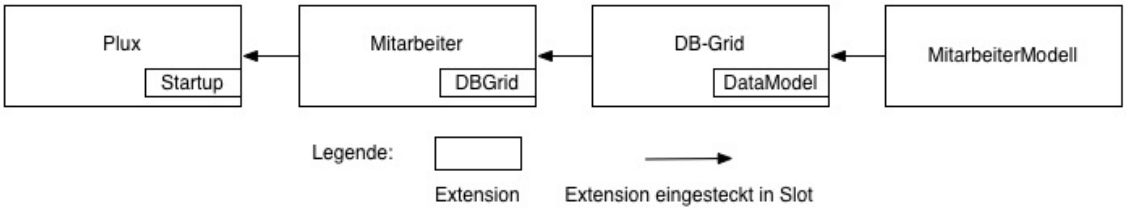

Abbildung 4.5: Schematische Darstellung der Extension einer Mitarbeiterliste

In der Abbildung 4.5 wird schematisch die Plug-Ins zur Anzeige von Mitarbeiter Datensätzen dargestellt. Der intuitive Ansatz besteht aus den Komponenten `Mitarbeiter` als Controller, `DB-Grid` zur Darstellung und `MitarbeiterModell` als Datenmodell.

Die Verknüpfung der Komponenten ist intuitiv richtig. Plux.NET ermöglicht es dem Entwickler, dass für jeden neu geöffneten *Slot* `DBGrid` eine neue Extension `DB-Grid` erzeugt wird. Das Verknüpfen des Datenmodells ist schwierig.

`DB-Grid` als generische Komponente kann ebenfalls nur generische *Slots* haben. Die generische Definition des *Slots* `DataModel` ermöglicht es allen Datenmodellen sich hier zu verbinden. Dieser Kontrakt muss eine generische Schnittstelle sein, damit die *Extension* `DB-Grid` mit verschiedenen Datenmodellen arbeiten kann.

Zur Lösung dieses Problems gibt es verschiedene Varianten. So wäre es denkbar, dass das Einstecken des Datenmodells über Code in der *Extension* `Mitarbeiter` und nicht durch die Automatismen von Plux.NET erledigt wird.

Eine weitere Möglichkeit wäre die Konfiguration der *Extension* `DB-Grid` über Parameter, sodass nur solche Datenmodelle automatisch angesteckt werden können, die ein

bestimmtes Attribut oder Namen haben. Ein solches Attribut muss im Konktrakt des *Slots* spezifiziert.

Eine weitere Lösung wäre, dass `DB-Grid` keinen *Slot* mehr anbietet. Dafür müsste sich das Datenmodell direkt in die Extension `Mitarbeiter` einstecken. Die Extension `Mitarbeiter` ihrerseits hätte dann die Aufgabe, Daten zwischen den *Extensions* `DB-Grid` und `MitarbeiterModell` auszutauschen.

Beachtet man diese Fakten, kann man eine große Wiederverwendung erzielen. Die Unit `BMDDBGrid` wird im Quelltext des ERP-Programms häufig benötigt. Sie ist ein wichtiger Bestandteil im ERP-Programm. Ist diese Komponente als austauschbare Extension verfügbar, kann diese an vielen Orten eingesetzt werden.

4.3.3 Funktionen des ERP-Programms als Extension

Die Architektur des ERP-Programmes enthält tiefe Vererbungslinien. Das bedeutet, dass die Entwickler sich stark an der Wiederverwendung von Code der Basisklassen von *BMD-Tools* orientiert haben. Die Wiederverwendung wird durch das Ableiten von eben diesen Basisklassen abgewickelt.

So gibt es in den Basisklassen Felder für Modelle, dem Menü oder für die Verarbeitung von Benutzereingaben. In den daraus abgeleiteten Klassen werden diese Felder nur benutzt. So verteilt sich der Code für eine Funktion auf viele Klassen.

Der Code für das Initialisieren oder Befüllen der Felder befindet sich in verschiedenen Quelltextdateien. Diese Vorgangsweise erleichtert es den Entwicklern das Erstellen von neuen Funktionen, indem Basisklassen bereits große Funktionalität haben und erst in den einzelnen abgeleiteten Klassen diese Funktionalität genau angepasst wird. Diese Vorgangsweise verhindert massiv die globale Wiederverwendbarkeit von Quelltexten.

Für die Zerlegung in Extensions ist das Verteilen von Funktionalität auf mehrere vererbende Klassen hinderlich und kann dies sogar unmöglich machen. So müsste für eine Funktion Quelltext aus verschiedenen Klassen extrahiert werden, um diesen Quelltext dann in eine Extension verpacken zu können.

Ohne Programmieraufwand und totem Code kann hier also keine neue Extension entstehen. Man muss im Detail klären ob es sinnvoll ist, den alten Code zu verwenden, oder ob es nicht besser wäre die Funktionalität neu zu entwickeln. In vielen Situationen steht der Portierungsaufwand in keinerlei Relation zu einem sauberen Neuentwurf.

Diese Problematik zeigt sich ganz stark bei den Klassen für die Controller und für die Fensterdarstellung. Hier ist es fast unmöglich, aus den Fenstern spezifische Teile herauszuextrahieren, diese in einer Extension zu verpacken und dann wieder zu der Funktion zusammenzusetzen.

Das Ziel dieser Arbeit war, die Portierung mit möglichst geringem Aufwand zu bewerkstelligen. Man kann sich nur mit den oben genannten Nachteilen arrangieren, und die Funktionen, die Blätter im Vererbungsbaum sind, als Gesamtes zu einer Extension portieren. Das bedeutet, dass man entweder die zu portierende Funktion mit dem Code für das Plug-In-Framework versieht, oder die Klasse dieser Funktion noch einmal ableitet und in dieser dann den Code für das Plug-In-Framework platziert.

Das Ergebnis ist eine vom Quelltext her kleine Extension, die auf die Klassenbibliothek zugreift. So ist es später möglich, diese Extension isoliert neu zu entwicklen. Die Klassenbibilothek wird nach und nach nicht mehr notwendig und kann schlussendlich ausgeschieden werden.

Listing 4.5: Daten Modell für die Funktion Mitarbeiter

```
[Extension("BMD.Mitarbeiter.Model", Singleton = true)]
[Plug("BMD.Mitarbeiter.DataModel")]
```

```
public class MitarbeiterModelProxy : IMitarbeiterDataModel{
  TBMDMDMitarbeiterMgr model;
  int Konzernnummer;
  int Jahr;
  int Firmennummer;
  int modelPosition;
  public MitarbeiterModelProxy(int nr, int j, int fnr) {
    Organizer.Units.Models.RegisterModels();
    Konzernnummer = nr;
    Jahr = j;
    Firmennummer = fnr;
  }
  public void Init() {
    model = new TBMDMDMitarbeiterMgr(null,true, true);
    model.Load(Konzernnummer, Jahr, Firmennummer);
    model.Open();
  }
  public int Position {
    get { return model.DataSet.RecNo;}
    set { model.DataSet.RecNo = value; }
  }
  public int Count {
    get { return model.DataSet.RecordCount; }
  }
}
```

Ganz im Gegensatz dazu stehen die Datenmodelle. Hier ist die Ausgangssituation gleich, für die spätere Verwendung treten die oben besprochenen Probleme nicht auf. Die Erklärung liegt darin, dass ein Modell als abgeschlossene Einheit aufgenommen werden kann. Die Funktionalität ist hier stark gekapselt und kann einfach in neue Funktionen integriert werden.

So kann man Datenmodelle als fertige Extension mit einer abgeschlossenen Funktionalität sehen. Das kommt dem Bottom-Up Entwurf entgegen. Vom Archtitekturstandpunkt ist das Entwicklen von Extensions eher dem Bottom-Up Entwurf zuzurechnen, weil einzelne Extensions erstellt und dann zu einer Funktion zusammmengesetzt werden. In Tests

wurden die bestehenden Modelle einfach mit dem Code für das Plug-In-Framework versehen und mit neuen Extensions zur Anzeige der Daten verbunden.

4.3.4 Ursachen für den hohen Aufwand beim Zerlegen des Quelltextes

In der Architektur des ERP-Programmes wurden Architekturentscheidungen getroffen, die es schwierig machen, das ERP-Programm auf die Plug-In-Plattformen zu migrieren. Die tiefen Vererbungslinien machen es schwierig, die benötigten Programmteile zu isolieren.

Ein weiteres Problem, das sich nicht auf die Portierung jedoch auf die Zerlegbarkeit auswirkte, waren die wenigen Interfaces gepaart mit tiefen Vererbungslinien. So entstand eine unklare Trennung zwischen der API und SPI. Anstatt eines klaren API und eines klaren SPI, welches die Anwendungsentwickler eigentlich erweitern müssen, wurde API und SPI durch Vererbung vermischt und aufgeweicht.

4.4 Bei der Zerlegung verwendete Architekturmuster

Architekturmuster helfen Entwicklern dabei, ihre Programme mit einer guten Architektur zu entwickeln. Diese Architekturmuster können auch bei der Arbeit mit Plug-Ins helfen. In der Arbeit wurden einige davon auf ihre Tauglichkeit untersucht und im Prototypen eingesetzt.

Dabei gibt es Muster, die in Prototypen getestet wurden, für die Zerlegung des ERP-Programms nicht eingesetzt werden konnten, weil die Verwendung des Musters gravierende Änderungen im Code nötig machen würde.

Bei der Arbeit mit Extensions wurden verschiedene Muster verwendet, welche die Architektur der Anwendung verbesserten. Einige der Muster konnten direkt Verwendung in der Anwendung finden.

Im Folgenden sollen trotzdem alle Varianten Erwähnung finden, da es sich um Muster handelt, die für Plug-In-Plattform Plux.NET erfolgreich funktionieren.

4.4.1 Action

Das Architekturmuster *Action* ist eine Abwandlung des schon bekannten Musters aus der Delphi Welt. Dieses Muster lässt sich für die Portierung des ERP-Programmes benutzen. Abbildung 4.6 zeigt schematisch das Zusammenstecken der einzelnen *Extensions*.

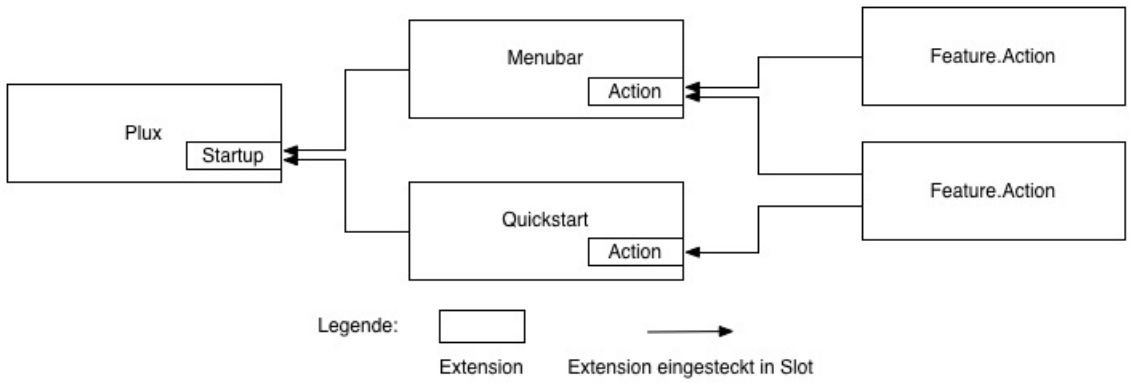

Abbildung 4.6: Schematische Darstellung der Extensions für das Action Pattern

Dieses Muster bietet eine dezentrale Abarbeitung und Ausführung der damit auszulösenden Funktion. Weiterer Vorteile ist die granulare und exakte Steuerung, mit der bestimmt werden kann, welche *Action* sich in welchem *Slot* einstecken und dort somit ausgelöst werden kann. Dieses Muster bringt Vorteile zum aktuellen Ablauf des Auslösens einer Funktion.

Dieses Muster wurde beim Prototyp im Fenster von *BMD-Quickstart* bereits eingesetzt und ermöglicht es, mit nur geringen Anpassungen in bestehendem Quelltext dieses Musters einzusetzen. Mit diesem Muster konnte das zentrale, dafür komplexe, *Action-Management* dezentralisiert und näher zur eigentlichen Ausführung der Funktion gebracht werden.

4.4.2 Abstract Factory

Das Architekturmuster *Abstract Factory* ist ein erzeugendes Muster. Das Muster wird eingesetzt wenn Objekte erzeugt werden sollen die erst zur Laufzeit bekannt sind. Wird zur Laufzeit ein bestimmtes Objekt einer Klasse benötigt, kann die Factory eine Instanz davon erzeugen.

Eine *Factory* für *Extensions* würde eine ähnliche Funktion erfüllen. Eine *Extension* kann zur Laufzeit andere *Extensions* benötigen, die der Plug-In-Plattform nicht bekannt sind.

Bei *Extensions* wird zur Übersetzungszeit der Quelltext und die Schnittstelle definiert. Die Schnittstelle sind die *Slots* und *Plugs* einer *Extension*. Die Plug-In-Plattform Plux.NET steckt – wenn nicht anders konfiguriert – die *Extensions* automatisch zusammen. Eine Instanz einer generischen *Extension* würde in jeden passenden *Slot* eingesteckt werden. Für viele Einsatzgebiete ist das falsch. Eine generische *Extension* für eine Schaltfläche wird in den *Slot* eingesteckt, der zur Übersetzungszeit angegeben wird. Gibt es mehrere gleichlautende *Slots* mit der gleichen Schnittstelle, steckt die Plug-In-Plattform die Instanz der generischen *Extension* überall ein. Eine Schaltfläche darf aber nur einmal in eine andere *Extension* eingesteckt werden.

Eine *Extension-Factory* erzeugt aus generischen *Extensions* konkrete *Extensions* mit einer veränderten Schnittstelle, die dann über die automatischen Mechanismen der Plug-In-Plattform Plux.NET zusammengesteckt werden. Die konkrete *Extension* hat das gleiche Verhalten wie die generische *Extension*. Der Name der *Extension*, der *Plugs* und *Slots* wird verändert.

Die *Extension-Factory* ist selbst eine *Extension*. Sie öffnet zwei *Slots*. Im *Slot* `Factory.Register` werden alle generischen *Extension* eingesteckt. Die generischen *Extensions* sind somit der *Extension* `Factory` bekannt. Am *Slot* `Factory.Request` werden alle *Extensions* eingesteckt, die generische *Extensions* benötigen.

Die generischen *Extensions* enthalten zusätzliche Attribute, die der *Factory* die Informationen geben um eine konkrete *Extension* zu erzeugen.

Listing 4.6: Ausschnitt aus der generischen Extension Toolbar

```
[ExtensionPrototype("Toolbar")]
[Plug("Factory.Register")]
[SlotPrototype("Toolbar", OnAttached = "OnAttached",
   OnDetached = "OnDetached")]
public partial class Toolbar : UserControl, IRegister {   }
```

In Listing 4.6 wird ein Ausschnitt des Quelltextes der generischen *Extension* `Toolbar` gezeigt. Das Attribut `[ExtensionPrototype("Toolbar")]` gibt den Namen für die generierten *Extensions* vor. Das Attribut `[SlotPrototype("Toolbar")]` definiert ebenfalls die Namen der *Slots* in den generierten *Extensions*. Zusätzlich werden dort auch die Methoden angegeben die bei den Ereignissen der Plug-In-Plattform Plux.NET aufgerufen werden.

Listing 4.7: Ausschnitt aus der Extension Mitarbeiter

```
[Extension("Mitarbeiter")]
[Plug("Factory.Request")]
```

```
[Slot("Mitarbeiter.Toolbar", DerivedFrom="Panel",
   MultipleExtensions = false)]
[PrototypeRequest("Toolbar", PlugInTo = "Mitarbeiter.Panel",
    Prefix = "Mitarbeiter")]
public class Mitarbeiter : IRequest {   }
```

In Listing 4.7 wird ein Ausschnitt aus der *Extension* Mitarbeiter gezeigt. Diese *Extension* fordert mit dem Attribut `[PrototypeRequest("Toolbar")]` die generische *Extension* `Toolbar` an. Zusätzlich wird spezifiziert, in welchen *Slot* die zu erzeugende *Extension* eingesteckt werden muss.

Aus den Informationen der generischen *Extension* erstellt die *Factory* eine konkrete *Extension*, die nach den Informationen aus der anfordernden *Extension* konfiguriert ist. Dazu analysiert die *Factory* die generische *Extension* durch *Reflection*. In dem neuen *Assembly* wird die Klasse der generischen *Extension* abgeleitet. Das Ergebnis speichert die *Factory* in einem neuen *Assembly*.

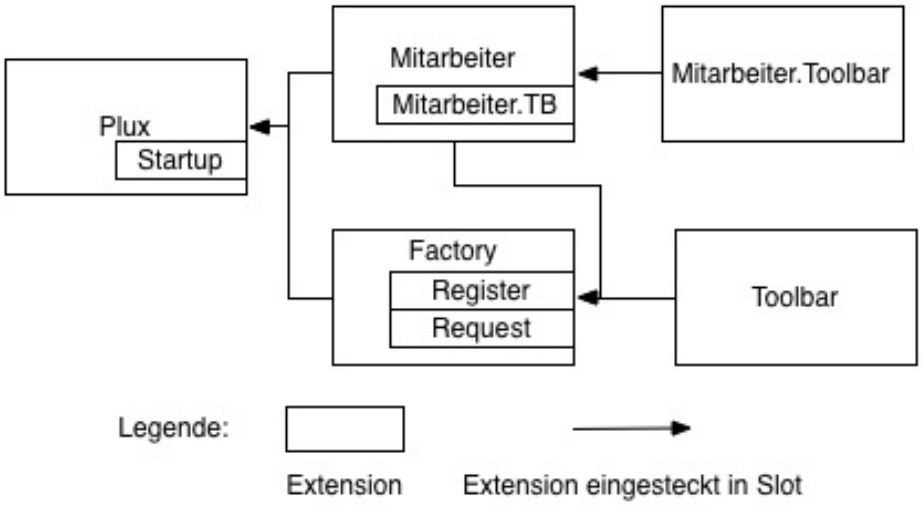

Abbildung 4.7: Schematische Darstellung des Architekturmusters Factory

Abbildung 4.7 zeigt, wie die verschiedenen *Extensions* zusammengesteckt sind. Im Laufe des Projekts haben sich andere Techniken entwickelt, um ein äquivalentes Verhalten zu erzielen. Dadurch wird der Mechanismus des automatischen Erzeugens von *Extensions* überflüssig.

4.4.3 Observer

In Plux.NET ist es möglich, eine *Extension* über alle Ereignisse, die das Anstecken und Ausstecken von *Extensions* betrifft, zu benachrichtigen. Mit dieser Technik kann ein *Security-Manager* implementiert werden, der nur bestimmte *Extensions* zulässt. Im Prototypen wurde damit die Anmeldung an die Datenbank realisiert.

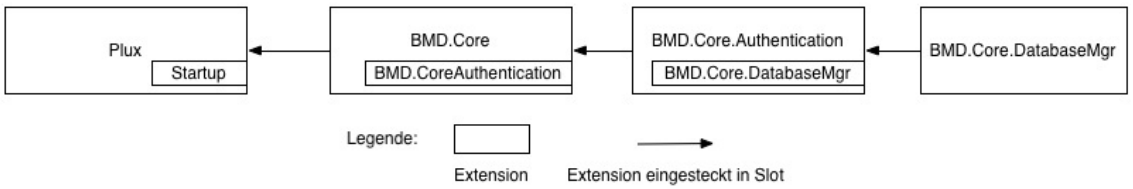

Abbildung 4.8: Schematische Darstellung der Extensions beim Datenbank-Login

Erst wenn der Einstieg über die Datenbank erfolgreich durchgeführt wurde, öffnet die *Extension* `BMD.Core` den *Slot* `BMD.Core.Authentication`. Durch dieses Öffnen des *Slots* und das Anstecken von Plug-Ins wird erst die Benutzeroberfläche angesteckt und für den Benutzer angezeigt.

Das Observer-Muster eignet sich für die Arbeit mit Extensions. Man kann dieses Muster auch benutzen, um komplexe Benutzeroberflächen zusammenzusetzen. Das größte Problem bei einer Anwendung, deren grafische Benutzeroberfläche auf einem Komponenten-Modell basiert, ist die Reihenfolge, in der die einzelnen Komponenten eingefügt werden.

Die Reihenfolge gibt an, in welcher Beziehung die einzelnen Benutzeroberflächen-Komponenten zueinander stehen. Daher kann das Aussehen der Anwendung nicht garantiert werden. Ohne zusätzliche Daten kann dies nicht bewerkstelligt werden. Mit dem Observer Muster ist ein Layout-Manager denkbar, mit dem die verschiedenen Komponenten richtig ineinander gesteckt werden.

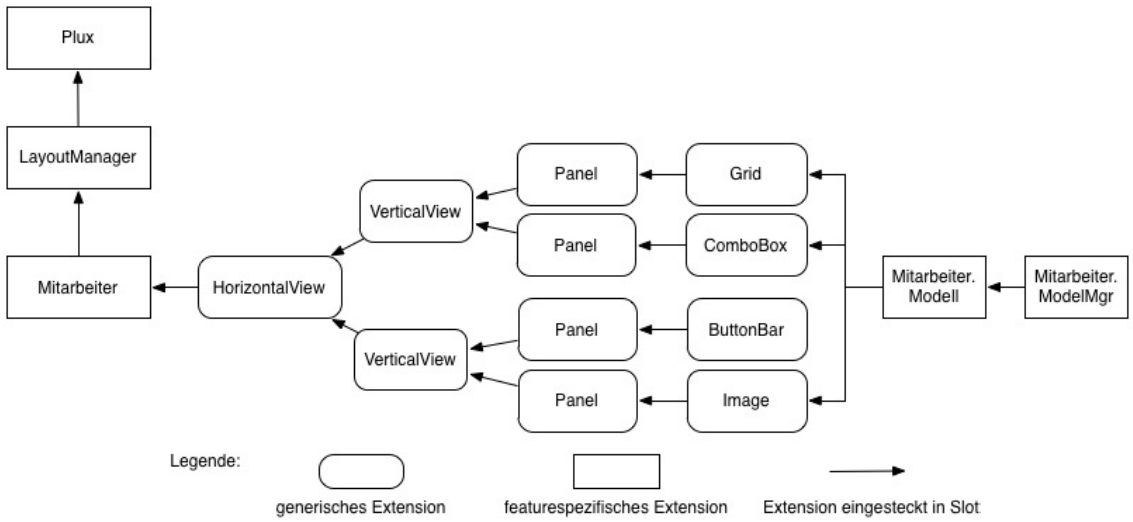

Abbildung 4.9: Schematische Darstellung der Anordnung von Extensions durch einen Layout-Manager

Mit diesem Muster lassen sich auch generische Extensions der Benutzeroberfläche mit einander zu verbinden. In Abbildung 4.9 wird beispielhaft eine Mitarbeiterliste dargestellt. Der Entwickler verwendet dort die generischen Plug-Ins zur Anzeige der Daten aus dem Datenmodell. Für diese Aufgabe müssen nur die Datenmodell-*Extension* und die *Extension* für die Funktion programmiert werden. Die *Extension* Mitarbeiter enthält die Information, welche generischen Extensions benötigt werden und wie und wo sie dargestellt werden.

Dazu ist es im Kontrakt notwendig, Parameter und Einstellungen anzugeben, wie sich eine neue Extension in die bereits bestehenden einfügen soll. Alle überwachten *Extensions* weisen sowohl einen *Plug* als auch einen *Slot* mit derselben *Slot-Definition* auf.

Wird die *Extension* Mitarbeiter entdeckt, unterbindet der *Layout-Manager* das Anstecken der generischen *Extensions* und untersucht die Parameter. Gemäß den Parametern aus der *Extension* für die Funktion hängt der *Layout-Manager* die generischen *Extensions* richtig in die Benutzeroberfläche ein.

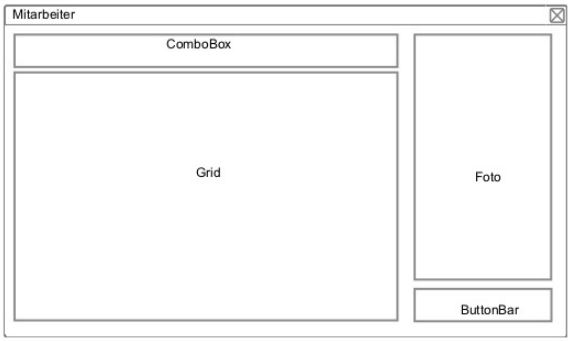

Abbildung 4.10: Schematische Darstellung des Fensters nach Konfiguration durch Layout-Managers

Abbildung 4.10 zeigt das mögliche Ergebnis eines *Layout-Managers*, der die generischen Extensions so zu einer Benutzeroberfläche zusammenstellt.

Für die konkrete Umsetzung im Prototyp konnte dieses Muster nicht eingesetzt werden, da dieses Muster noch nicht implementiert war. Aufgrund der tiefen Vererbungslinien wäre es ohnehin schwierig geworden, es einzusetzen. Die Vererbungslinien machen es schwierig, einzelne Teile der Benutzeroberfläche herauszutrennen.

4.4.4 Facade

Das Architekturmuster Facade ist für das Portieren einer monolithischen Anwendung hilfreich. Es kapselt die Schnittstelle der zu portierenden Klasse und bietet nach außen hin eine einfachere Schnittstelle.

Listing 4.8: Taschenrechner von BMD

```
using System.Windows.Forms;
using Plux;
using NTCS.Calculator;
[Extension("BMD.Plux.Tools.Calculator.View", OnCreated = "
   OnCreated")]
[Plug("Plux.Tool")]
[ParamValue("Name","Calculator")]
[ParamValue("OrderIndex","0.0")]
public class BMDCalculator : ITool
{
  TBMDFRMCalculator calculator;
  ExtensionInfo me;
  public BMDCalculator() {
  calculator = new  TBMDFRMCalculator(null);
  }
  public void OnCreated(object sender, ExtensionEventArgs
     args){
    me = args.ExtensionInfo;
  }
  #region ITool Members
  #endregion
    public int Handle { get { return (int)calculator.Handle;
        } }
    public void Show(){
       calculator.StartAlone(false);
    }
    public void Hide(){
       calculator.Hide();
    }
    public void Close(){
      calculator.Close();
      me.Release();
    }
```

```
}
```

Mit diesem Muster beherrscht man das Problem der langen Vererbungslinien von Klassen gut. Man muss nur wenig neuen Quelltext erzeugen, um eine Funktion als Extension zur Verfügung zu stellen. Zum Anderen braucht man keine Verhaltensänderungen fürchten. Eine solche Extension kann dann in der Plug-In-Plattform ganz normal eingesetzt werden.

Das *Assembly* für die *Extension* ist klein und stellt einfach die Funktion aus der Bibliothek zur Verfügung. Der gesammelte bestehende Quelltext bleibt dabei in der Bibliothek gekapselt. Ist es einmal nötig diese Funktion neu zu schreiben, stehen die Kontrakte fest und die neue *Extension* fügt sich in die Plattform ein und ersetzt die alte transparent.

4.4.5 Model-View-Controller

Das Model-View-Controller-Muster (kurz : MVC) ist ein in der Literatur und in der Praxis weit verbreitetes Architekturmuster. Es trennt die drei Teile, aus denen jede Funktion besteht: Das *Model*, welches die Daten beinhaltet, die der Benutzer abfragen oder verändern möchte, die *View*, die die Daten für den Benutzer darstellt, und der *Controller*, welcher auf die Eingaben des Benutzers reagiert und die Daten verändert.

Dieses Muster fügt sich schon jetzt gut in eine Plug-In-Architektur ein. Bei der Plug-In-Architektur kommen noch das Thema der Austauschbarkeit von *Extension* hinzu. Abbildung 4.12 zeigt die schematische Implementierung einer Funktion mit *Extensions* nach dem MVC-Architekturmuster. So kann durch die Implementierung mit *Extensions* das *Model* ausgetauscht werden oder andere *View* eingesetzt werden.

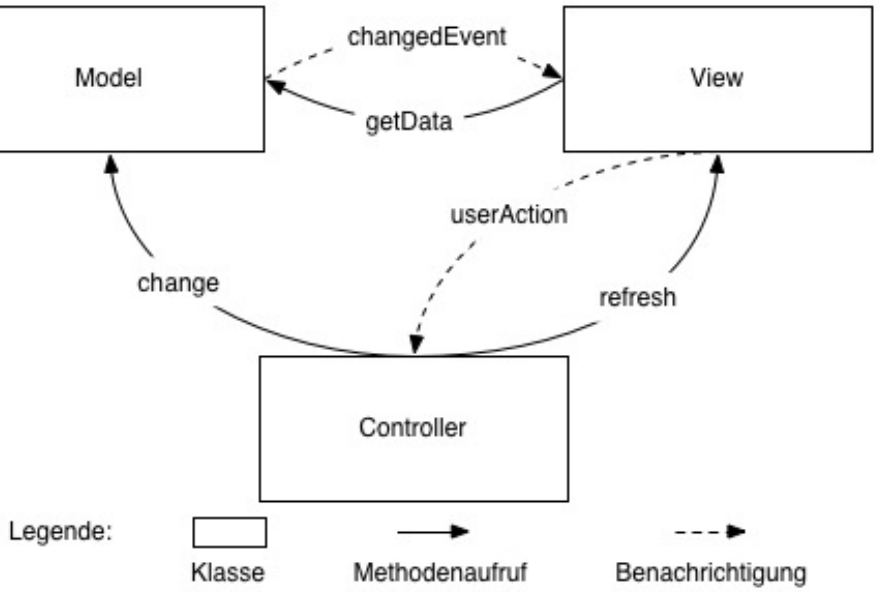

Abbildung 4.11: Model-View-Controller-Architekturmuster aus der Literatur

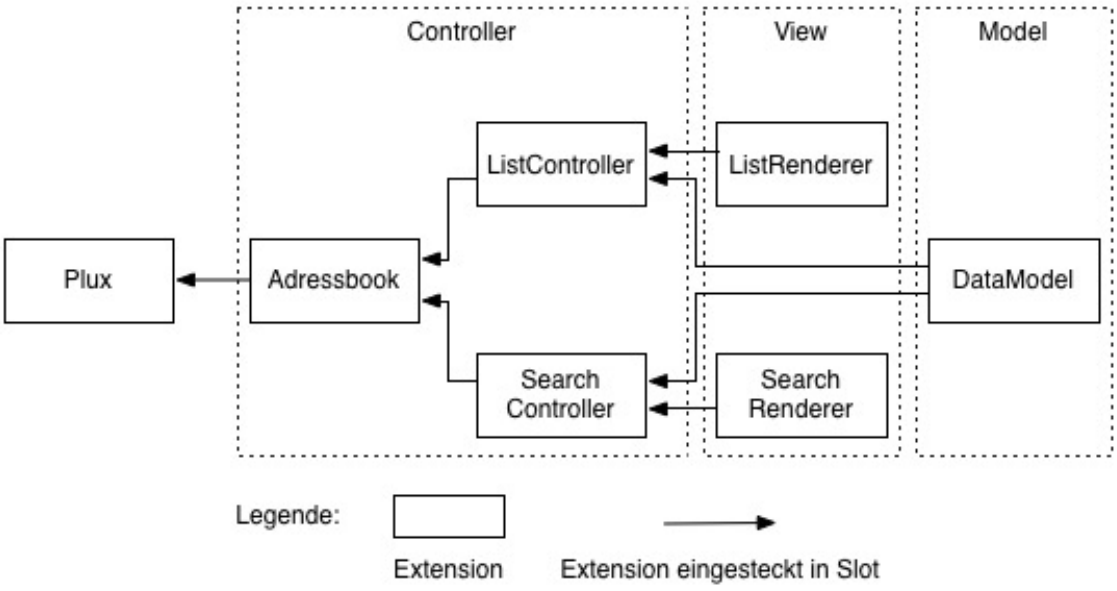

Abbildung 4.12: Schematische Darstellung der Extensions für MVC-Muster

Der einzige Unterschied zwischen dem Architekturmuster aus der Literatur und dem in Abbildung 4.12 beschriebenen Muster in Abbildung 4.11 ist, dass die *View* nicht mehr mit dem *Model* direkt verbunden ist. Diese Architektur ist mit Extensions auch realisierbar. In dieser Systematik der Plug-In-Plattform können ganz einfach Teile der *View*, das *Model* oder die *Controller* ausgetauscht werden. Zu diesem Grunde wurde eine Funktion auf mehrere Teile der Benutzeroberfläche aufgeteilt. So kann einer dieser Teile ein Menü darstellen, ein anderer eine Liste mit den Daten. Ein solcher Teil kann als einzelnes MVC-Muster gesehen werden.

Mit einer weiteren *Slot-Extension* Verbindung zwischen *Model* und *View* kann es zu zusätzlichen Problemen kommen. Die *View-Extensions* müssten genau wissen, was sie anzeigen sollen und wie sie es aus dem *Model* extrahieren können. Der *Controller* hingegen kennt das *Model* und auch die *View-Extensions* und kann wesentlich besser und zielgerichteter Daten zustellen.

Für die Portierung des ERP-Programmes ist dieses Muster nicht durchführbar. Grund dafür sind die tiefen Vererbungslinien, denen eine einzelne Funktion unterliegt. Die Funktionalität für *Controller* und *View* sind auf verschiedene Klassen verteilt und können nur mit großem Aufwand in separate *Extensions* extrahiert werden. Ohne ein Neudesign, bei dem schon von Beginn an mit *Extensions* gearbeitet wird, lässt sich hier das MVC-Muster nicht einführen. Wenn dieser Schritt vollzogen ist, eröffnen sich dafür neue Möglichkeiten beim Aufbau einer Benutzeroberfläche. Das bedeutet eine höhere Wiederverwendung von Komponenten wie Menüs oder der Listendarstellung für Datenbankinhalten.

4.4.6 Schichtenarchitektur

Momentan existiert im ERP-Programm keine konkrete und strenge Schichtenarchitektur, daher gibt dieser Abschnitt nur schematisch Auskunft über die Möglichkeiten, die mit

der Portierung zu Plug-Ins entstehen. Wesentlichste Erkenntnis ist, dass Extensions sich transparent in eine Schichtenarchitektur integrieren.

Dieser Umstand wirkt sich positiv auf den Einsatz eines Applikationsservers aus. Das ERP-Programm verbindet sich immer direkt mit der Datenbank und führt darauf Abfragen aus. In einer Schichtenarchitektur wird strikt zwischen der Speicherung der Daten, sowie der Präsentations- und Geschäftslogik unterschieden.

Das wesentliche an der Schichtenarchitektur ist, dass Teile einer Schicht nur mit der direkt darunterliegenden Schicht kommunizieren. Es wird nie eine Schicht übersprungen. Die Konzepte rund um Plux.NET sind gleich und fügen sich somit transparent ein eine Schichtenarchitektur ein.

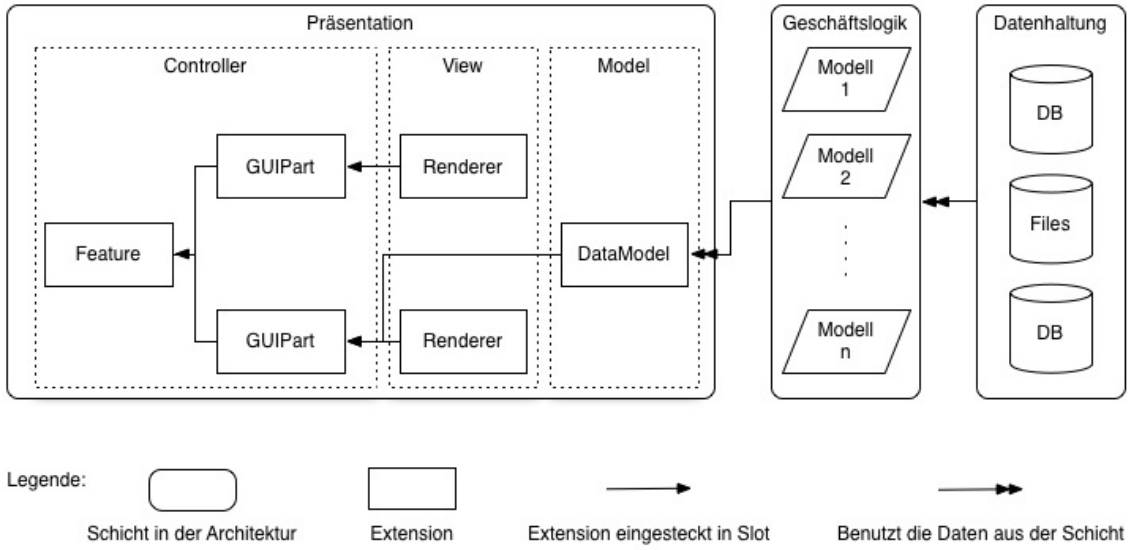

Abbildung 4.13: Schichtenarchitektur mit Plug-Ins

Wie schon besprochen wird die Präsenationsschicht mit Extensions realisiert. Sie beinhaltet jenen Teil der Gesamtanwendung, der für den Benutzer sichtbar ist. Die Extensions, die den Zugriff auf Daten bereitstellen, greifen nicht direkt auf die Datenhaltung in

der Datenbank zu. Vielmehr stellt die Logikschicht die Daten für diese Extensions zur Verfügung.

In dieser Logikschicht werden die Daten für die Extensions aggregiert. Vor der Persistierung werden die Daten mit Konsistenzprüfungen aus der Geschäftslogik überprüft. Erst in der Datenhaltungschicht, die zum Beispiel eine Datenbank darstellt, werden dann die Daten wirklich persistiert.

4.5 Funktion des Prototyps

Das Ergebnis dieser Arbeit ist eine prototypische Implementierung einer Bürosoftware, die auf dem Code des ERP-Programmes basiert und mit den Techniken und Mitteln rund um die Plug-In-Plattform Plux.NET zerlegt und modularisiert wurden. Die Plug-In-Plattform Plux.NET ermöglicht es, Programme Schritt für Schritt zusammenzustecken. Auf diese Weise kann man erkennen, wie der Prototyp im inneren funktioniert. Um die schematischen Darstellungen einfach zu halten wurden Präfixe und nicht benötigte *Slots* in den Abbildungen ausgelassen.

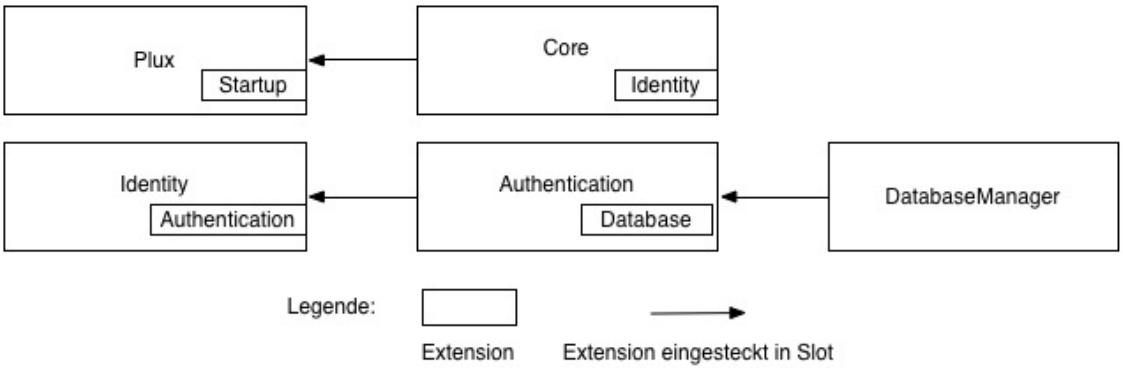

Abbildung 4.14: Schematische Darstellung der Extensions vor Einstieg in die Datenbank

Am Beginn ist nur die *Extension* Plux mit dem *Slots* `Startup` instanziert. Als nächstes wird die *Extension* `Core` an die *Extension* `Plux` angesteckt. Diese *Extension* hat zu diesem Zeitpunkt nur den *Slot* `Identity` geöffnet. In diesem *Slot* soll eine *Extension* eingesteckt werden, die die Identität des Benutzers prüft.

Abbildung 4.15: Benutzerdialog des ERP-Programms zur Eingabe des Benutzernames und des Kennwortes

In Abbildung 4.14 sieht man bereits eine Instanz einer *Extension* `Identity`. Diese ist aber nicht eingesteckt, weil die *Extension* `Authentication` das Einstecken nicht erlaubt. Durch die Eingabe des Benutzernamen und des Kennwortes wie in Abbildung 4.15, wird in der *Extension* `Identity` der Benutzername und das Kennwort gespeichert. Die *Extension* `Authentication` überprüft mit der *Extension* `Database` die Gültigkeit der Identität. Wird von der *Extension* `Database` die Identität bestätigt, gibt die *Extension* `Authentication` das Einstecken der *Extension* `Identity` in der *Extension* `Core` frei.

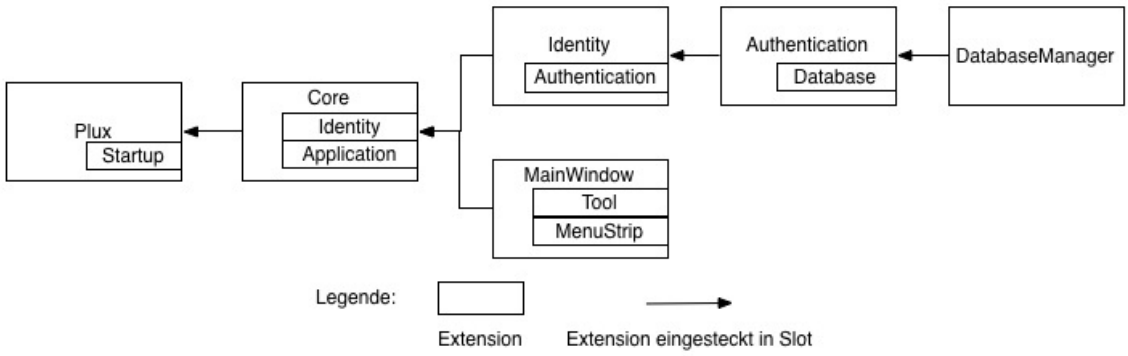

Abbildung 4.16: Schematische Darstellung der Extensions nach Einstieg in die Datenbank

Abbildung 4.16 zeigt schematisch den Zustand der *Extensions* nach der Eingabe des Benutzernamens und des Kennwortes. Die *Extension* `Identity` ist in die *Extension* `Core` eingesteckt. Dadurch öffnet die *Extension* `Core` den *Slot* `Application`. Durch das Öffnen des *Slots* `Application` wird automatisch die *Extension* `MainWindow` eingesteckt. Diese *Extension* enthält das *MDI*-Fenster, in dem später die Fenster der gestarteten Funktionen angezeigt werden.

Die *Extension* `MainWindow` öffnet zwei *Slots*. In den *Slot* `Tool` werden alle gestarteten Funktionen gesteckt. In diesem *Slot* können *Windows.Forms*-Fenster und *VCL*-Fenster gesteckt werden. Die *Extension* `MainWindow` sorgt dafür, dass beide Fensterarten angezeigt werden. Der zweite *Slot* lautet `MenuStrip`. In diesen *Slot* werden *Extensions* für Menüzeilen eingesteckt. Es sind zu diesem Zeitpunkt noch keine Steuerelemente in dieses Fenster eingesteckt.

Die Abbildung 4.17 zeigt den Startbildschirm des Prototyps, nach der Anmeldung an die Datenbank. Dieses Fenster ist ein `Windows.Forms` *MDI*-Fenster. In diesem Fenster werden später die Funktionen angezeigt. Durch *Wrapper*-Klassen ist es möglich in diesem Fenster auch die Benutzeroberfläche der portierten Funktionen aus dem ERP-Programm

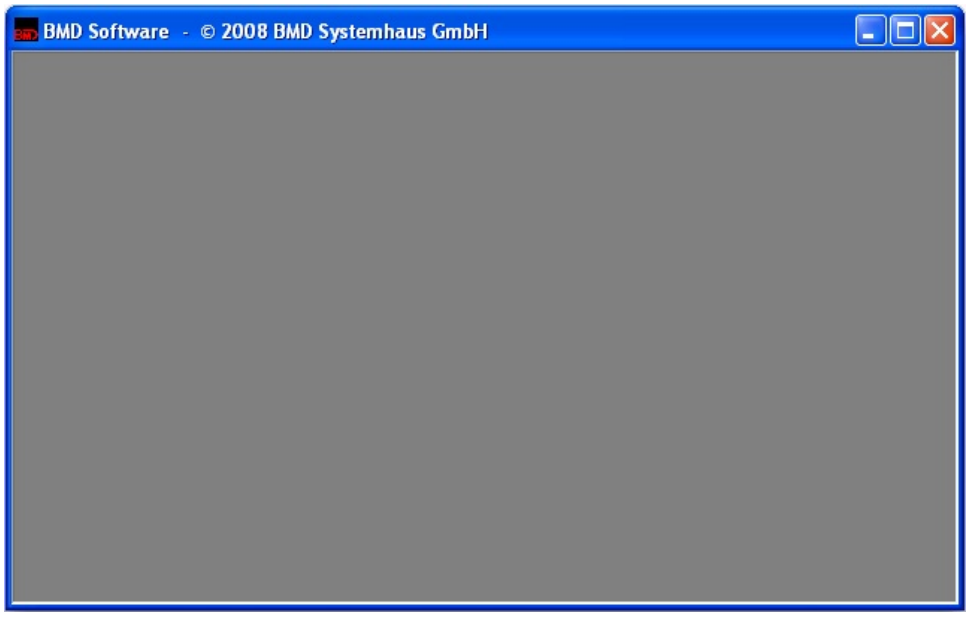

Abbildung 4.17: Geöffnetes *MDI*-Fenster ohne weitere Steuerelemente

anzuzeigen. Mit Win32-Befehlen können die *VCL*-Fenster gestartet und dann in das *MDI*-Fenster hineingezogen werden.

Abbildung 4.18 zeigt den Zustand am Ende des Startvorganges. Die *Extension* `MenuStrip` wurde angesteckt. Diese *Extension* öffnet neben dem *Slot* für *Action-Extensions* noch einen Slot um lokalisierte Bezeichnungen für die *Actions* anzuzeigen.

Von der *Extension* `NLSManager` wird automatisch eine Instanz erzeugt und eingesteckt. Weiters zeigt die Abbildung 4.18 die *Extension* der Funktion `Quickstart`. Diese Extension wurde in den *Slot* `Tool` eingesteckt.

Die Menüzeile enthält schon Einträge, um die *Extensions* der Funktionen zu starten. Diese Einträge wurden aus *Action-Extensions* der Funktionen erstellt. Funktionen, die zum Paket *Organizer* gehören, werden im Menü unter diesem Punkt zusammengefasst. Die portierte Funktion Taschenrechner des ERP-Programms befindet sich im Menü *Tools*.

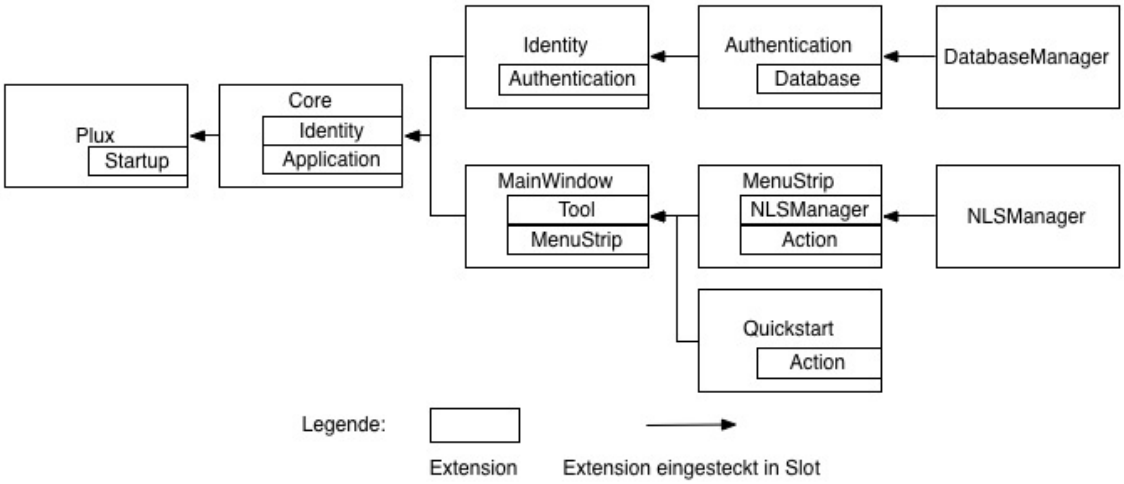

Abbildung 4.18: Schematische Darstellung der Extensions am Ende des Startvorganges

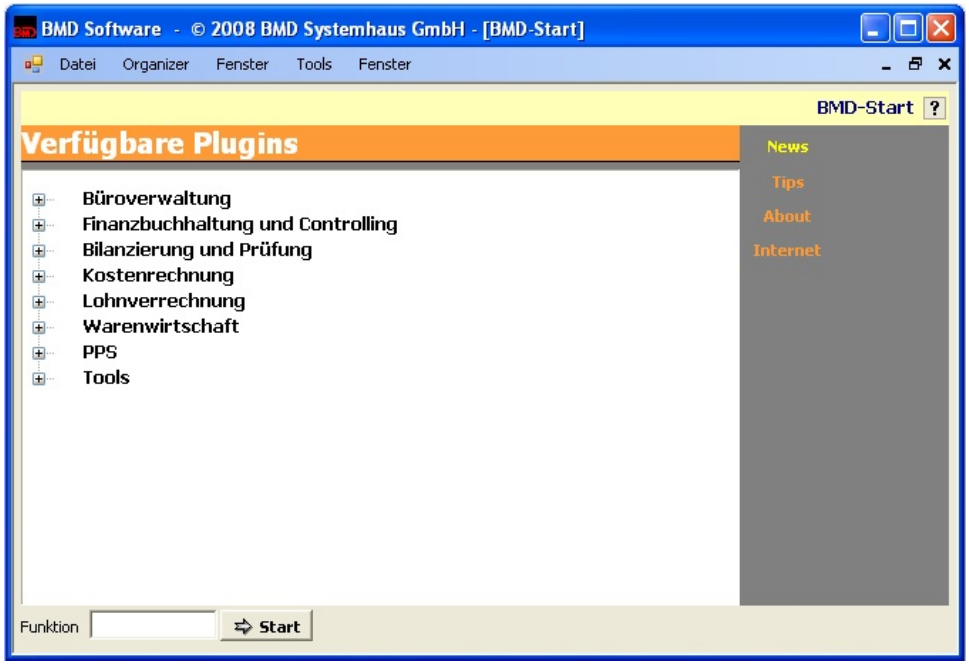

Abbildung 4.19: Benutzeroberfläche des Prototyps mit der Funktion Quickstart

In Abbildung 4.19 ist die Funktion *Quickstart* gestartet. Wie auch im ERP-Programm sind alle Pakete und die dazugehörigen Funktionen zu sehen. Allerdings lassen sich nur jene aufrufen, die als Extension portiert wurden. Die portierten Funktionen des ERP-Programms können auch über den *Matchcode* gestartet werden.

Die Menüleiste und Statusleiste sind als *Extension* realisiert. Diese *Extensions* sind von den Klassen aus dem Microsoft.NET Framework 2.0 abgeleitet. Abbildung 4.20 zeigt, wie sich die Statusleiste schematisch in das laufende Programm einfügt. Die *Extension*

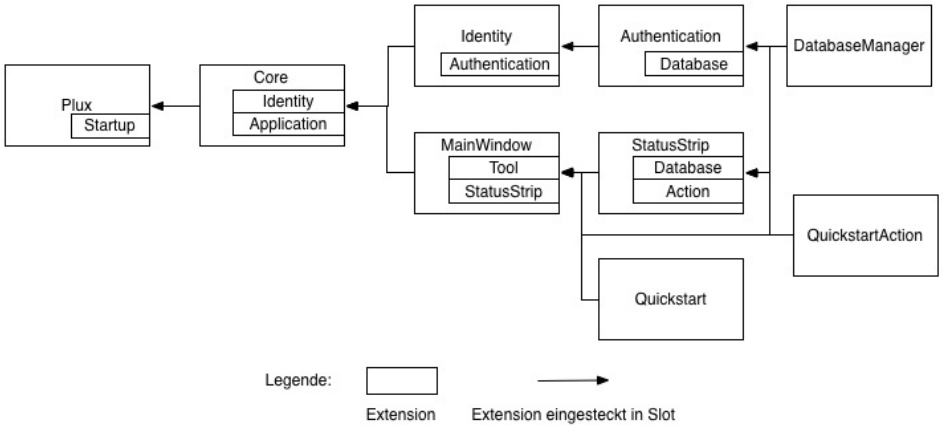

Abbildung 4.20: Ausschnitt der schematischen Darstellung zur Implementierung der Extension StatusStrip

`MenuStrip` öffnet den *Slot* `Action`. Die Plug-In-Plattform benachrichtig die *Extension* über alle *Action-Extensions*. Die *Extension* `MenuStrip` lehnt aber alle *Extensions* bis auf `Quickstart` ab. Wird die Extension `Quickstart` angesteckt, zeigt die Statusleiste die Schaltfläche *Start* an. Die Statusleiste öffnet zusätzlich noch den *Slot* `Database`.

Im ERP-Programm enthält der Statusleiste Informationen über den aktuellen Benutzer, den Mandanten und den Namen der Datenbank. Die *Extension* `Database` wird bei

der Instanzierung der *Extension* `StatusStrip` automatisch durch die Plug-In-Pattform eingesteckt.

In Abbildung 4.21 werden alle portierten Funktionen gestartet. Die *Wrapper*-Klassen der *Extension* `MainWindow` ist in der Lage, die VCL-Fenster kaskadiert und nebeneinander anzuzeigen.

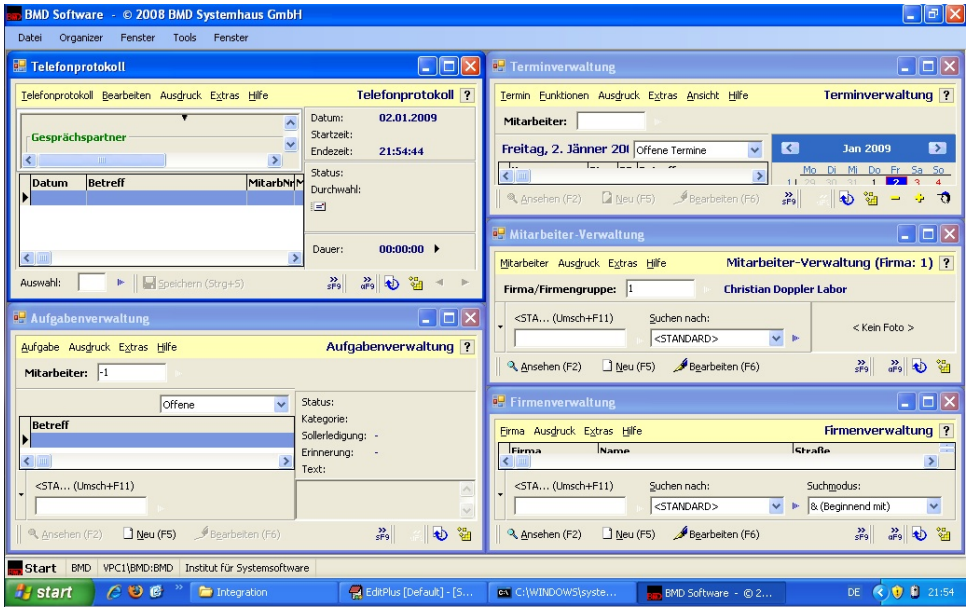

Abbildung 4.21: Benutzeroberfläche des Prototyps mit portierten Funktionen

Kapitel 5

Diskussion und Ausblick

5.1 Schlussfolgerung aus dem Zerlegen

Der Anfang ist gemacht. Für die Entwickler des ERP-Programmes steht eine prototypische Implementierung zur Verfügung. Der erstellte Prototyp ist in einer virtuellen Maschine verfügbar, die den Entwicklern zur Verfügung gestellt werden kann. Darin sind der Prototyp, der Quelltext und alle verwendeten Programme enthalten. Neben dieser prototypischen Implementierung entstanden noch andere Artefakte, die bei der Erstellung der Arbeit entstanden sind. Die verschiedenen Zerlegungsgrade sind in verschiedenen *Extensions* vorgestellt worden.

Zum aktuellen Zeitpunkt hat die Firma Embarcadero die Entwicklung an Delphi.NET eingestellt. Die Entwicklung der Visual Class Library wurde ebenfalls eingestellt. Stattdessen gibt es Delphi Prism, eine Delphi-ähnliche Sprache für das .Net Framework.

Delphi Prism hat schon jetzt einige Inkompatibilitäten auf Quelltextebene. Hinzu kommt, dass Delphi Prism die Klassenbibliothek des Microsoft.NET Frameworks benutzt und nicht wie früher eine eigene Klassenbibliothek mitbringt. Das erweist sich gerade bei der Zerlegung von GUI-Komponenten als schwierig.

Gerade im Hinblick darauf, dass Embarcadero in Zukunft keinen eigenen Zweig für die Microsoft.NET Plattform mehr anbietet, ist es auch nicht ratsam, hier weiter Zeit zu investieren. Die sauberste Möglichkeit wäre das Nachimplementieren der Funktionalität. Diese Art wäre sehr resourcenintensiv. Für die Entwickler des ERP-Programmes wäre dies ein frischer Start, es wäre notwendig, dass diese Neuentwicklung von einer zusätzlichen Entwicklermannschaft durchgeführt wird, um die aktuelle Entwicklung noch weiter zu warten.

Großer Nachteil ist hier, dass diese Neuimplementierung nicht Schritt für Schritt sondern komplett unabhängig von der bisherigen Entwicklung entsteht. Hinzu kommt, dass es sich hier um komplett neue Technologien handelt, die erst einmal verstanden werden müssen.

Die einfachste Möglichkeit wäre es, einfach den kompletten Code auf Delphi.NET zu portieren und daraus ein großes monolithisches *Assembly* zu erstellen. Für die zukünftigen Entwicklungen dient dieses *Assembly* nur als große Codebasis.

Am Beginn aller Faktorisierung zu Plug-Ins steht die Portierung. Die Funktionalität der verschiedenen Pakete können dann mit Plux.NET zu einer Architektur verwoben werden, die alle Anforderungen an die Zukunft erfüllt. Hinzu kommt, dass es leichter ist, neue Entwickler für das Microsoft.NET Framework zu finden.

5.2 Neue Erkenntnisse aus dem Zerlegen

In dieser Arbeit wurden Techniken erarbeitet und besprochen, um Delphi Code zu portieren und aus diesen portierten *Assemblies* Teile herauszulösen. Die Schwierigkeit dabei lag in der Tatsache, dass der Code eine starke Kohäsion aufwies und daher das Aufteilen nur schwer durchführbar war.

Weiters wurden die verschiedenen Ansätze zur Faktorisierung vom bestehenden Code zu Extension besprochen und durch eine prototypische Implementierung dargestellt.

Es wurden für die Arbeit mit der Plug-In-Plattform Muster identifiziert und verwendet, die helfen, zukünfigte Anwendungen mit guter Architektur zu erzeugen. Manche Muster sind sehr erfolgreich, können aber nicht aus dem bestehenden Quelltext einfach erzeugt werden.

5.3 Mögliche Schritte nach der Zerlegung

Für BMD stellt sich die aktuelle Lage zwiespältig dar. Durch die Übernahme von Embarcadero wurde die Plattform für native Entwicklung gestärkt. Sie hat das Potential wieder konkurrenzfähig zu werden.

Die Unterstützung der Microsoft.NET Plattform durch Delphi.NET wurde hingegen komplett verworfen. Delphi Prism ist eine mächtige Entwicklungsumgebung und ermöglicht Entwicklern mit Delphi-Erfahrung, in Zukunft neue Programme für die Microsoft.NET Plattform zu erstellen.

Für bestehende Programme hingegen bedeutet Delphi Prism kein Schritt in Richtung vonMicrosoft.NET. Für eine bestehende Applikation sind viele Wege denkbar, um sie auf die Microsoft.NET Plattform zu portieren.

Mit Plux.NET werden die bestehenden Funktionen des ERP-Programmes mit einfachen Action-Extensions gestartet. Ist es möglich oder notwendig, eine Funktion neu zu entwickeln, werden alle Verbindungen zur alten Codebasis gekappt und ein neues Plug-In mit der gesammten Funktionalität erstellt.

Hier ist eine große Disziplin bei der Neuentwicklung notwendig. Die Entwickler müssen planvoll vorzugehen, da sonst die alte Codebasis nie wirklich obsolet wird.

Am Besten wäre einer händische Neuentwicklung und Überarbeitung des Kerns in Delphi.NET, Delphi Prism oder gleich in C#. Parallel dazu kann eine Überarbeitung der Anwendungsteile stattfinden, sodass diese möglichst komfortabel und hochautomatisisert – zum Beispiel per Cross-Compiler – ebenfalls portiert werden können.

Hier liegt der große Vorteil, dass dies Schritt für Schritt durchgeführt werden kann. Die neu erstellten *Assemblies* können mittels COM und Interop Microsoft.NET *Assemblies* im nativen Code von Delphi eingebunden werden.

Dazu wird der Quelltext einfach vom C# Übersetzer in ein Microsoft.NET *Assembly* übersetzt. Mit `regasm` kann eine Typenbibliothek erstellt werden die dann in Delphi einfach eingebunden und einfach aufgerufen werden kann.

So können Neuentwicklungen bereits als Microsoft.NET *Assembly* erstellt werden und trotzdem gleich im bestehenden Altcode verwendet werden. Im Anhang finden sich auf Seite 98 2 Quelltexte, in der ein Beispiel für diese Wiederverwendung von CLR Code in Delphi gezeigt wird.

Literaturverzeichnis

[BMD/Preisliste 12.12.2008] *Preisliste für Wirtschaftskunden.* 12.12.2008. – http://www.bmd.at/downloads/preislisten/BMD_PL_WIRTSCHAFT.pdf

[BMD/Unternehmensprofil 1.6.2009] *Unternehmensprofil.* 1.6.2009. – http://www.bmd.at/desktopdefault.aspx/tabid-3/89_read-38/

[Doberenz und Gewinnus 2005] Doberenz, Walter. ; Gewinnus, Thomas: *Borland Delphi 2005: Microsoft .NET Framework-Entwicklung.* Hanser Fachbuchverlag; Auflage: 1, May 2005. – ISBN 3446402020

[Gruhn und Thiel 2000] Gruhn, Volker ; Thiel, Andreas: *Komponentenmodelle. DCOM, Javabeans, Enterprise Java Beans, CORBA.* Addison-Wesley, 2000. – ISBN 382731724X

[Heineman und Councill 2001] Heineman, George T. ; Councill, William T.: *Component-Based Software Engineering: Putting the Pieces Together (ACM Press).* Addison-Wesley Professional, June 2001. – ISBN 0201704854

[Jahn 2009] Jahn, Markus: *Entwurf und Implementierung eines Cross-Compilers von Delphi nach C.* Linz, Österreich, Johannes Kepler Universität, Dissertation, 2009

[Knasmüller 2001] Knasmüller, Markus: *Von Cobol zu OOP: Umsteigen auf objektorientierte Programmierung.* dpunkt.verlag GmbG, 2001. – ISBN 3932588959

[Sangal u. a. 2005] Sangal, Neeraj ; Jordan, Ev ; Sinha, Vineet ; Jackson, Daniel: Using dependency models to manage complex software architecture. In: *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.* New York, NY, USA : ACM, 2005, S. 167–176. – ISBN 1-59593-031-0

[Steward 19.09.2008] Steward, Bob: *Dr. Bob on Delphi Actions and Action Lists.* 19.09.2008. – http://www.drbob42.com/delphi4/actions.htm

[Wischnewski 19.04.2008] Wischnewski, Daniel: *Eager to use Delphi.NET for .NET 2.0? Start now!* 19.04.2008. – http://delphi-notes.blogspot.com/2006/03/eager-to-use-delphinet-for-net-20.html

[Wolfinger 2010] Wolfinger, Reinhard: *Dynamic Application Composition with Plux.NET: Composition Model, Composition Infrastructure.* Linz, Österreich, Johannes Kepler Universität, Dissertation, 2010

Anhang

Portierungsmöglichkeiten

In diesen Quelltextbeispielen soll gezeigt werden, wie der bestehende Code möglichst behutsam ohne große Veränderung im Quelltext für eine neue Plattform konserviert werden kann.

Portierung zu nativen Dynamic link Library

Listing 1: Bereitstellung von Delphi Code mit Dynamic Link Libraries

```
library Add;

uses SysUtils, Classes, Dialogs;

function Add(a, b :Integer): Integer; export; stdcall;
begin
  Result := a + b;
end;

exports Add index 1;

begin
end.
```

Listing 2: Benutzung Dynamic Link Libraries

```
program DLLClient;

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
     Forms, Dialogs,
  StdCtrls;

interface

function Add(a, b : Integer):Integer;stdcall;

implementation

function Add;external 'Add.dll' Name 'Add';stdcall;

var
 c : Integer;
begin
  c := Add(1, 2);
end.
```

Portierung zu Component Object Model

Listing 3: Definition der globalen Variablen und des Interfaces

```
unit ICOMTest;

interface

const
  Class_COMTest: TGUID = '{857DD02-E24A-11D4-BDE0-00
     A024BAF736}';
  IID_ICOMTest : TGUID = '{857DD02-E24A-11D4-BDE0-00
     A024BAF736}';

type
  ICOMTest = interface ['{8576CE04-E24A-11D4-BDE0-00
     A024BAF736}']
    function Add(a, b: Integer): Integer;
  end;
```

```
implementation

end.
```

Listing 4: Code des COM Servers

```
unit COMTest;

interface
  uses Windows, ActiveX, Classes, ComObj, DSGlobals;

type
  TCOMTest = class(TComObject, ICOMTest)
  protected
    function Add(a, b: Integer): Integer; stdcall;
  end;

implementation
  uses ComServ;

  function TTestCOM.Add(a, b: Integer): Integer;
  begin
    result := a + b;
  end;

begin
end;

initialization
  TComObjectFactory.Create(ComServer,
                           TCOMTest,
                      Class_COMTest,
                'COMTest', 'COM Objekt addiert',
                ciMultiInstance,
                tmApartment);
end.
```

Listing 5: Verwendung des Codes beim Client

```
program COMTestClient;

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
     Forms, Dialogs,
  StdCtrls, ICOMTest, ComObj, ActiveX;
```

```
var
  FCOMTest : ICOMTest;
  c : Integer;
begin
  OleCheck(CoCreateInstance(Class_DisplaySomething,
                            nil,
                            CLSCTX_ALL,
                            ICOMTest,
                            FCOMTest));
  c := FCOMTest.Add(1, 2);
end.
```

Quelltextbeispiele

Kernsystem von NTCS

Dieser Quelltext zeigt den auf Delphi.NET portierten und mit Plux.NET zerteilten Code, mit dem das Kernsystem von NTCS als Plugin zur Verfügung gestellt wird.

Listing 6: NTCS.Core

```
unit BMD.Plux.Core.Core;

interface

uses Plux, BMD.Plux.Core.Contracts.ICore, NTCS.Core.
   Interfaces;

type
  [Extension('BMD.Core', OnCreated = 'OnCreated', OnReleased
       = 'OnRelease', Singleton = true)]
  [Plug('Startup')]
  [Plug('BMD.Core.Database')]
  [Slot('BMD.Core', AutoOpen = FALSE)]
  TBMDCore = class(TObject, IStartup, ICoreDatabaseManager)
  private
    me : ExtensionInfo;
    FInitialized : Boolean;
  public
```

```
    procedure OnCreated(s : TObject; args :
       ExtensionEventArgs);
    procedure OnRelease(s : TObject; args :
       ExtensionEventArgs);
    procedure Run;
   function DatabaseManager() : IBMDDatabaseManager;
  end;

implementation

uses NTCS.Core.HostSide, System.IO, Forms;

procedure TBMDCore.Run;
var
  RootPath : String;
begin
 RootPath := Path.Combine(Path.GetDirectoryName(Application.
    ExeName), 'plugins\NTCS');
  //  if not InitializeCore(RootPath, 'BMD', true, true, '
     DBSERVER\BMD:BMD', 'bmd', 'bmd') then begin
  while not(FInitialized) do
  begin
    if not InitializeCore(RootPath, 'BMD', true, true, '', '
       BMD', 'BMD') then begin
      //cancel was pressed
      FInitialized := False;
      Runtime.Shutdown();
      exit;
    end
    else begin
      try
        //NTCS.Core.HostSide.Core.DatabaseManager.Database.
           Connection.Open;
  me.OpenSlots();
        FInitialized := True;
      except
        FInitialized := False;
      end;
    end;
  end;
end;
procedure TBMDCore.OnCreated(s : TObject; args :
   ExtensionEventArgs);
```

```
begin
  me := args.ExtensionInfo;

end;
function TBMDCore.DatabaseManager() : IBMDDatabaseManager;
begin
  Result := NTCS.Core.HostSide.Core.DatabaseManager;

end;
procedure TBMDCore.OnRelease(s : TObject; args :
   ExtensionEventArgs);
begin
  if FInitialized then begin
    CleanUpCore;
    me.CloseSlots();
    FInitialized := False;
  end;
end;

end.
```

Benutzung von CLR Assemblies in Delphi

Listing 7: Interface und Implementierung in C#

```
using System.Runtime.InteropServices;

// Interface
public interface IAdd
{
    int Add(int i1, int i2);
}

[ClassInterface(ClassInterfaceType.None)]
public class AddOp : IAdd
{
    public int Add(int i1, int i2)
    {
        return i1+i2;
    }
```

```
}
```

Listing 8: Benutzen des Assemblies in Delphi

```
function AddOperation(a, b : Integer) : Integer;
var
  refAddOp: IAdd;
begin
  refAddOp := CreateComObject(CLASS_AddOp_) as IAdd;
  Result := refAddOp.Add(2, 2);
end
```

Christian Mittermair

Hochbuchegg 24
4644 Scharnstein

Telefon: (07615) 7533
Mobil: (0699) 18 24 55 00
Email: mi.ch@gmx.net

Persönliches

Geboren am 11. Dezemer 1979 in Gmunden

Österreichischer Staatsbürger

Ausbildung

Volksschule Mühldorf, 1986–1990.

Hauptschule Scharnstein, 1990–1994.

Handelsakademie Gmunden, 1994–1999.

Johannes Keppler Universität, 2003–2010

Akademische Grade

Bakk.techn. Informatik, Johannes Keppler Universität, 2007

Beschäftigungsverhältnisse

Grüne Erde GmbH, 2000–2003.

CD Labor für Automated Software Engineering, 2007–2009.

Grüne Erde GmbH, 2009–

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Linz, am 22. März 2010

Christian Mittermair