TANNER

DHBW
Duale Hochschule
Baden-Württemberg
Ravensburg
Campus Friedrichshafen

Entwicklung eines modularen Systems zur regelbasierten und automatisierten Dokumentkomposition

Bachelorarbeit

für die Prüfung zum **Bachelor of Engineering**
im Studiengang **Netz- und Softwaretechnik**
an der Dualen Hochschule Baden-Württemberg in Ravensburg
Campus Friedrichshafen

Florian Peschka

19. September 2012

Bearbeitungszeitraum	KW 27 / 2012 - KW 39 / 2012
Matrikelnummer, Kurs	6192194, TIT09
Partnerunternehmen	TANNER AG, Lindau
Betreuer im Partnerunternehmen	Andreas Fessler
Gutachter der dualen Hochschule	Claudia Zinser

Erklärung

gemäß §*5(2)* der Studien- und Prüfungsordnung DHBW Technik vom 18. Mai 2009.

Hiermit erkläre ich, dass ich die vorliegende Arbeit mit dem Titel

Entwicklung eines modularen Systems zur regelbasierten und automatisierten Dokumentkomposition

selbständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt, keine anderen als die angegebenen Hilfsmittel benutzt und wörtliche sowie sinngemäße Zitate als solche gekennzeichnet habe.

Florian Peschka

Kurzfassung

Diese Bachelorarbeit befasst sich mit einer problematischen Situation, die in vielen Betrieben aus Industrie und Dienstleistung besteht. Dabei handelt es sich um die Problematik, aus einer Menge an verschiedenen Informationen eine fertige, auslieferbare Dokumentation herzustellen. Wenn ein Unternehmen mit technischen Dokumentationen umgeht, wird nach einiger Zeit klar, dass die Verwaltung kleinerer, gekapselter Informationseinheiten einfacher ist, als ein großes Dokument immer weiter zu verändern.

Dadurch entstehen oft große Mengen von einzelnen Dokumenten oder Datenquellen, die jeweils für sich selbst eine bestimmte Informationsmenge beinhalten. Die Frage ist, wie diese einzelnen Quellen zu einer Gesamtdokumentation zusammengefasst und ausschnitthaft konfiguriert werden können.

Dieser Frage geht diese Arbeit auf den Grund und versucht, eine praktikable Lösung in Form eines Rahmenprogramms zu entwickeln. Dabei steht die Flexibilität und Zukunftsfähigkeit der Lösung im Vordergrund, da das beschriebene Problem in vielen Unternehmen besteht, aber die Umstände in jedem Fall deutlich unterschiedlich sind.

Dafür konnte ein Konzept erarbeitet werden, mit welchem sich prinzipiell alle verschiedenen Quelldaten verarbeiten lassen und auf beliebige Art und Weise neu ordnen lassen, solange es sich an gewisse Schnittstellen und Konventionen hält. Dieses in *C#* entwickelte Framework kann durch die Schnittstellen in Form von *Plug-Ins* an viele verschiedene Anwendungsfälle angepasst werden.

Weiterhin wurden Beispiele der Verwendung dieses Rahmenprogramms erarbeitet, die die Verwendung des Frameworks und die Programmierung eigener Plug-Ins für die Verarbeitung von *Word-Dokumenten* und *Datenbank-Tabellen* erläutern.

Letztendlich wurde im Rahmen eines konkreten Kundenprojektes gezeigt, wie sich durch das Framework eine praktikable Lösung für das Problem der Komposition mehrerer Word-Dokumente realisieren lässt. Das gesamte Projekt wurde vor diesem Hintergrund entwickelt und konnte als Lösung für dieses Kundenprojekt erstellt werden, ist jedoch flexibel genug, um auch andere Anforderungen abdecken zu können.

Abstract

This Bachelor's thesis deals with a common problem of industry and service companies. The task those companies are often faced with is to condense many documents into a single, shippable technical documentation. As soon as a company handles technical documents and documentation, it becomes apparent, that processing small bits of information is easier than managing and reworking a large document over and over again.

This leads to a vast amount of small documents in different types of sources which contain the desired information. The question is, how to combine all these different sources into a single document.

This thesis examines that question and tries to create a practical solution in the form of a computer program. Flexibility and sustainability are the key aspects of this approach because the described problem is often found in different types of companies and environments.

In order to achieve this, a concept has been developed which can process different types of sources and arrange them in a free and modular way into a new, composed document. This concept, developed in *C#*, provides the desired functionality if all components conform to certain interfaces and conventions. Using those *plug-in-interfaces*, custom solutions can be based on of the proposed structure which add support for customer and company specific environments.

Example applications of those plug-ins have been developed which show how to use the framework and how to develop own plug-ins for composing *Word documents* and *database tables* in other output formats.

The proposed solution was then used to solve a specific customer request. The complete project was completed with the problem pf this customer in mind, making the solution primarily for this particular customer, while still being flexible enough to cover other requirements as well.

Danksagung

Diese Bachelorarbeit wäre ohne die Unterstützung bestimmter Personen nicht entstanden. Diesen möchte ich hier meinen Dank aussprechen und ihre Arbeit und ihren Einsatz in dieser Danksagung würdigen.

Vorerst bedanke ich mich bei der *TANNER AG* und der *DHBW Ravensburg* für das Ermöglichen meines Studiums und damit der Möglichkeit, die Bachelorarbeit zu einem positiven Abschluss bringen zu können.

Besonders danken möchte ich dabei meinem (ehemaligen) Segmentleiter *Bernd Lehmanski*, der mir die Chance gegeben hat, die TANNER AG als Praktikant kennenzulernen und mir später die Tore geöffnet hat, mein duales Studium in dieser aufregenden und interessanten Umgebung zu absolvieren.

Weiterhin danke ich meinen Arbeitskollegen, die mir während der gesamten Zeit des Studiums rat- und tatkräftig zur Seite standen und im Besonderen meinem Segmentleiter *Andreas Fessler*, der mir die Erstellung der Bachelorarbeit ermöglicht hat.

Für das Gelingen und den reibungslosen und angenehmen Ablauf der Bachelorarbeit selbst bedanke ich mich bei *Tomislav Matievic*, der besonderen Einfluss auf die Entwicklung der Kundenanwendung hatte und meine gesamte Arbeit hochkompetent getestet und bewertet hat.

Herrn *Erwin Fahr* danke ich besonders für seinen Einsatz und das Engagement, mit welchem er nicht nur mich, sondern alle Studenten unseres Kurses bei der Absolvierung der uns auferlegten Aufgaben unterstützt hat.

Des weiteren möchte ich meinen Studienkollegen *Daniel Menke*, *Christian Schulz*, *Matthias Flock*, *Steffen Gießmann*, *Andreas Kerner*, *Christian Mosbach*, *Alexander Schaaf* und *Johannes Kast* für die gemeinsam verbrachte Studienzeit danken, die durch ihre Anwesenheit zu einer Erfahrung wurde, die ich nicht so schnell vergessen werde.

Zu guter Letzt danke ich natürlich allen hier nicht erwähnten Personen, die mich auf meinem Weg durch drei interessante Jahre des Studiums begleitet haben.

Vorwort

Um dem Leser einen leichteren Einstieg in diese Arbeit zu erleichtern, müssen zunächst einige Formulierungen und Konventionen, die in dieser Arbeit verwendet werden, erklärt werden. Zunächst sei gesagt, dass trotz der Bemühungen des Autors, nicht zu sehr in Anglizismen zu verfallen, es gerade im Bereich der Computer- und Software-Branche schwer ist, diese komplett zu verhindern.

Um dies zu verdeutlichen, sind Fachbegriffe oder solche, zu denen sich keine sinnvolle deutsche Übersetzung finden lässt oder eine solche sogar die Bedeutung des Begriffes verändern oder verfälschen könnte, in *kursiver* Schreibweise zu erkennen.

Ebenso ist es unablässlich für Arbeiten im IT-Umfeld, über Variablen, Klassen, Schnittstellen und sonstige Elemente aus Programmiersprachen zu diskutieren. Um solche Elemente entsprechend zu kennzeichnen, wird ein `dicktengleiche` Schrift verwendet. Diese Begriffe beziehen sich meistens direkt auf vorher gezeigte Ausschnitte aus Quellcode.

Diese Code-Ausschnitte sind in hervorgehobenen größeren Blöcken zu finden, wie folgend beispielhaft gezeigt.

Beispiel eines Code-Blocks

```
1 // Hier steht Quellcode der Anwendung
2 public String doSomething();
```

Es ist zu beachten, dass diese Code-Ausschnitte lediglich zu demonstrativen Zwecken und der Veranschaulichung von Abläufen dienen sollen. Der darin enthaltene Code ist in den meisten Fällen für sich genommen nicht lauffähig und stellt nur Auszüge aus dem Gesamtprogramm dar.

Das Thema der Arbeit ordnet sich grundlegend im Bereich der Softwareentwicklung anhand von konkreten Kundenanforderungen ein. Gemessen an dem OSI-Standard-Modell[ISO96] kann man die Arbeit im *Application*-Bereich (Layer 7) einordnen. Damit ist dies keine streng wissenschaftliche Arbeit, sondern vielmehr die Dokumentation der Entwicklung eines Prototypen, welcher bestimmte Anforderungen erfüllen soll.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

In der technischen Dokumentation spielen seit jeher Dokumente jeglicher Art eine große, wenn nicht sogar die größte Rolle. Betriebs-, Montage- und Wartungsanleitungen sowie -beschreibungen werden in verschiedensten Dokumenten und Systemen abgelegt, gepflegt und vertrieben. Viele dieser Dokumente sind in Dokument-Formaten wie *Microsoft Office Word*, *OpenOffice Writer* oder *FrameMaker* gespeichert. Andere sind in abstrakten Repräsentationsformen vorhanden, z. B. in Datenbanktabellen. [web12]

Diese Dokumente sind das Zentrum der Bemühungen, hochwertige Dokumentationen jeglicher Form anzulegen, da sie das direkte Bindeglied zum Endkunden darstellen und ihm die darin erklärte Technik verständlich machen sollen. Um dies zu gewährleisten, müssen technische Dokumente jeder Art einer ständigen Qualitätssicherung unterliegen und dauerhaft kontrolliert werden.

Um diese Prozesse zu vereinfachen, werden oftmals Teile der Dokumentation als separate Dateien abgelegt und so einzeln gekapselt behandelt. Je nach Komplexität und Kompliziertheit der zu dokumentierenden Sache können diese Dokumentensammlungen schnell zu einer enormen Datenmenge anwachsen. Diese zu verwalten ist jedoch immer noch einfacher, als ein einzelnes Gesamtdokument ständig neu zu überarbeiten und diese Änderungen zu protokollieren.

Besonders hervorzuheben ist hierbei, dass die meisten Dokumentationen nicht aus einheitlichen Quellen entstehen. Grafiken werden aus CAD[1]-Programmen exportiert oder durch proprietäre Software wie *Adobe Illustrator* erstellt, wohingegen komplexe Dokumente meist in XML[2]-ähnlichen Strukturen und Editoren verwaltet werden.

Um auslieferungsfähige Dokumentationen zu produzieren, müssen demnach Prozesse gefunden werden, die aus verschiedenen Quellen ein fertiges, in sich schlüssiges Dokument produzieren. Diese Prozesse sind aufwändig und werden von geschulten Redakteuren in mühsamer Kleinarbeit durchgeführt.

Da sich diese Prozesse jedoch in den meisten Firmen sehr ähnlich sind, ist hier ein deutliches Optimierungspotential zu erkennen, da sie durch eine gemeinsame Schnittstelle automatisiert werden können. Diese Schnittstelle muss in der Lage sein, verschiedene Quellen auf eine den jeweiligen Eingangsdaten angepasste Art und Weise zu verarbeiten und diese in einem definiertem Ausgangsdokument zu bündeln.

[1] Computer Aided Design
[2] Extensible Markup Language

1.2 Problemstellung

Um einen einheitlichen Prozess zu gestalten, der Eingangsdaten aus verschiedenen Quellen verarbeiten kann und diese in einem neuen Ausgangsdokument zusammenfassen kann, soll ein modulares System entwickelt werden, welches diese Anforderungen erfüllt. Dabei ist besonders wichtig, dass das System selbst je nach Anwendungsfall flexibel konfiguriert werden kann.

Die manuelle Zusammenstellung einer Dokumentation aus einer Vielzahl von Datenquellen ist enorm fehleranfällig und führt häufig zu unerwünschten Effekten, die im Nachhinein einzeln kontrolliert und korrigiert werden müssen. Die Überführung der verschiedenen Ausgangsformate in eine einheitliche Datenstruktur ist hierbei eine der größten Herausforderungen für den zuständigen Redakteur.

Oftmals sind bestimmte Grafikformate nicht mit den Dokumentformaten kompatibel und es müssen aufwändige Konvertierungsverfahren verwendet werden, bis sich die Gesamtdokumentation als einheitliches Dokument ausliefern lässt.

Um diesen Problemen Abhilfe oder zumindest Beihilfe zu schaffen, soll das System durch eine geeignete Schnittstelle in der Lage sein, an jegliche Art von Eingangs- und Ausgangsdaten angepasst werden zu können.

1.3 Inhalt dieser Arbeit

Diese Arbeit soll als Dokumentation des kompletten Entwicklungsprozesses eines Systems mit den in Abschnitt 2 aufgeführten Anforderungen verstanden werden. Es wird der gesamte Prozess vom Erfassen der Anforderungen über die eigentliche Entwicklung des Systems bis hin zu einem fertigen Prototypen und einer beispielhaften Anwendung für ein konkretes Kundenprojekt beschrieben und dokumentiert.

Es wird auf die verwendeten Technologien, Vorgehensweisen und Rahmenprogramme eingegangen und eine detallierte Übersicht über die erarbeiteten Ergebnisse gegeben. Die Ergebnisse dieser Arbeit sollen im weiteren Verlauf als Grundlage für verschiedene Projekte dienen, deren Anforderungen sich in dem Sinne ähneln, als dass stets ein Gesamtdokument aus verschiedenen Quellen erstellt werden soll und dieser Prozess durch Automatisierung vereinfacht werden kann.

Die Arbeit wird vor dem Hintergrund eines konkreten Kundenprojektes erstellt, für welches mit Hilfe des entwickelten Systems eine akute Problemstellung beseitigt werden soll. Diese Lösung wird in der Arbeit als beispielhafte Anwendung des Systems vorgestellt.

Die Kundenanwendung stellt hierbei einen deutlichen Teil der Anforderungen an das fertige System, darf jedoch nicht als eigentliches Ergebnis dieser Arbeit angesehen werden, sondern lediglich als Muster für dessen konkreten Einsatz.

2 Aufgabenstellung

2.1 Gesamtbeschreibung

Wie bereits kurz in Unterabschnitt 1.2 erörtert, soll entsprechend dem Titel dieser Arbeit ein *modulares* System zur *regelbasierten* und *automatisierten Dokumentenkomposition* entwickelt werden. Diese grobe Anforderung lässt sich in vier wesentliche Bestandteile zerlegen, durch welche die Anforderungen an das System genauer beschrieben werden:

2.2 Modularität

Als Modularität im Sinne dieser Arbeit wird die Eigenschaft des Systems angesehen, durch möglichst kleine Änderungen an viele verschiedene Arten von Umständen anpassbar zu sein. Diese Anpassungen sollen, um als Modularität angesehen werden zu können, nicht nur in ihrem Umfang und Komplexität möglichst gering sein, sondern auch nur an bereits definierten Punkten im Gesamtsystem nötig und möglich sein.

Ein System, welches keine Möglichkeiten zur Erweiterung der Basisfunktionalitäten bietet, um an neue Umstände angepasst zu werden, kann nicht als modular angesehen werden. [Ngu] Im Bezug auf das zu entwickelnde System dieser Arbeit bedeutet dies insbesondere, dass das System Möglichkeiten bieten muss, durch die seine Funktionalitäten sowohl austauschbar als auch erweiterbar sind.

Es darf jedoch nicht davon ausgegangen werden, dass das fertige System an jegliche mögliche Konstellation von Anforderungen anpassbar sein kann. Die Hauptaufgabe des Gesamtsystems ist weiterhin die Möglichkeit der Dokumentenkomposition, wie in Unterabschnitt 2.5 definiert. Sollten Anforderungen an das System gestellt werden, die diese Definition nicht erfüllen, ist es kein Bruch der Modularität des Systems, wenn diese nicht erfüllt werden können.

2.3 Regelbasiertheit

Die allgemeine Definition regelbasierter Systeme (z. B. nach [IP96]) ist für diese Arbeit nur teilweise als gültig zu betrachten. Zum einen ist das zu entwickelnde System keines, welches aus bestimmten Regeln und Fakten Schlüsse ziehen kann, sondern vielmehr eines, welches anhand gewisser Regeln Aktionen durchführt, wie zum Beispiel das Zusammenführen von verschiedenen Quelldokumenten.

Die Regeln sollen hier möglichst abstrakt und allgemein sein, damit durch eine geeignete Anpassung das System an vielen verschiedenen Stellen wiederverwendet werden kann. Wie genau die Regeln aussehen, muss je nach Anforderung an das System spezifisch definiert werden.

Hier sind die Regeln mehr als Richtlinien zu verstehen, die dem System mitteilen, wie es sich in gewissen Situationen zu verhalten hat oder wie es mit bestimmten Eingangsdaten umgehen soll. Die größte Rolle spielen hierbei offensichtlich die Anweisungen zum genauen Zusammenstellen der Eingangsdokumente in einem Ausgangsdokument.

Die Anforderung an das System im Bezug auf dessen Regelbasiertheit ist demnach, dass es vorhersehbar und zuverlässig für gleichartige Eingangsdaten dieselben Schlüsse und Aktionen durchführt.

2.4 Automatisierung

Das System soll die Aufgabe möglichst automatisiert abarbeiten können. Das bedeutet, dass das System in einem gewissen Rahmen autonom arbeiten und entscheiden können muss. Dem Endanwender sollten einige essentielle Einstellungsmöglichkeiten gegeben werden, die den Ablauf des Programms steuern, ab dem Starten der Aufgabe muss das System jedoch in der Lage sein, selbst Entscheidungen zu treffen, um den Endanwender soviel Arbeit wie möglich abzunehmen. [Deu]

Hierzu muss das System jedoch die in Unterabschnitt 2.3 erstellten Regeln und Anweisungen bedienen und darf sich nicht über diese hinwegsetzen.

Zusätzlich soll eine Möglichkeit geschaffen werden, die das automatisierte Aufrufen des Systems durch andere Programme ermöglicht, ohne dass ein Benutzer Eingaben macht oder Einstellungen verändert. Dies ist vor allem dann nötig, wenn das System in verschiedenen Umgebungen eingesetzt werden soll. Hier muss der Ergebniszustand sowohl beim Steuern des Prozesses durch eine grafische Oberfläche als auch über einen automatisierten Aufruf aus einem anderen Programm heraus der gleiche sein.

2.5 Komposition

Die Grundaufgabe des Systems ist die Komposition, also das Zusammenstellen von mehreren Einheiten zu einer Gesamteinheit [Klu02], von mehreren Eingangsdokumenten in einem Ausgangsdokument. Diese Aufgabe muss durch das System in einer Art ausführbar sein, die auch mit verschiedenartigen Eingangsdokumenten umgehen kann (z. B. ein `.doc`-Dokument und ein `.xls`-Dokument).

Je nach den Regeln des Systems muss es in der Lage sein, diese verschiedenen Eingangsquellen in einer Art und Weise zu verarbeiten, die es dem Endanwender erlaubt, die Komposition auch in verschiedenartige Ausgangsdokumente durchzuführen.

Dabei sollen sich die Daten der Eingangsdokumente auch in dem Ausgangsdokument wiederfinden. Dies kann natürlich für bestimmte Eingangsquellen nicht unbedingt gewährleistet werden, hier müssen die durch Unterabschnitt 2.3 definierten Regeln greifen und einen Kompromiss finden.

Hier muss erwähnt sein, dass eine Veränderung der Eingangsdaten nur dem Grundsystem verboten ist. Sollten bestimmte, nach Kundenanforderungen programmierte Module (s. Unterabschnitt 2.2) Eingangsdaten weglassen oder anders ausgeben, als die Eingabe es vorgibt, überwiegt diese Anforderung die hier formulierte Forderung nach Gleichheit der Eingangsdaten zu den Ausgangsdaten.

3 Konzeption

3.1 Machbarkeit und Technologieentscheidung

Um die Machbarkeit der Aufgabe festzustellen und eine Technologieentscheidung treffen zu können, wurden zunächst die Anforderungen genau analysiert. Aus diesen geht vorerst nicht unmittelbar hervor, auf welchen Betriebssystemen das System Anwendung finden soll.

Wie jedoch in Unterabschnitt 2.5 bereits beispielhaft erwähnt, ist aufgrund des Hauptanwendungsgebietes der fertigen Programme, die auf diesem System basieren, davon auszugehen, dass es sich ausschließlich um *Microsoft Windows* handeln wird.

Prinzipiell ist es, ausgehend von den reinen Systemanforderungen auch möglich, ein ähnliches System für andere Betriebssysteme zu entwickeln. Die Unterstützung von geeigneten Bibliotheken, die den Zugriff auf die von Microsoft zur Verfügung gestellte API[3] ermöglichen, ist hier jedoch mangelhaft. Es ist nicht vorauszusehen, dass neue Anforderungen von diesen Bibliotheken unterstützt werden. Um das Risiko möglichst gering zu halten, wurde auf diese Alternative verzichtet. [Apab]

Die verwendeten Technologien wurden demnach sehr stark durch die Anforderungen der Kundenanwendung beeinflusst, da das entwickelte System durch direkte Anwendung in der Praxis evaluiert werden sollte.

Durch die sehr grob und allgemein formulierten Anforderungen an das Grundsystem lässt sich die Machbarkeit einer Umsetzung a priori nur schwer einschätzen. Grundlegende programmiertechnische Anforderungen wie Objektorientierung und verwendbare Design-Pattern sind durch nahezu jede moderne und ausreichende hohe Programmiersprache erfüllt.

Um die Anforderung nach dem möglichst einfachen und sicheren Umgang mit *Microsoft Word*-Dokumenten der Kundenanwendung bereits bei der Planung des Grundsystems zu berücksichtigen, wird als Hauptprogrammiersprache *C#* verwendet. Diese von Microsoft entwickelte Sprache für die *.NET*-Umgebung auf Windows-Betriebssystemen bietet die besten Möglichkeiten, Programme, die mit Office umgehen sollen, zu programmieren.

3.2 Evaluation bestehender Lösungen

Für das Zusammenstellen von mehreren Word-Dokumenten gibt es zwar bereits Lösungen, von denen jedoch viele auf manuelle Aktionen des Benutzers ausgelegt [wor08][wor08] oder nur als kleine Skripthilfen für bereits bestehende Word-Dokumente vorhanden sind. [wor]

Bestehende Programme zur Zusammenführung von Word-Dokumenten besitzen ein wenig anpassbares Interface und können nur schwer an die Anforderungen an diese Anwendung angepasst werden. Ebenso sind diese bestehenden Software-Lösungen nicht modular aufgebaut und können so nicht für verschiedene Arten von Daten die gleichen Aktionen durchführen. [Sob][3St]

[3] Application Programming Interface

Damit ist eine zufriedenstellende Lösung der Problemstellung einer modularen, automatisierten Lösung mit bereits bestehenden Software-Lösungen nicht wahrscheinlich. Eine Eigenentwicklung bietet hier deutliche Vorteile, da sie wesentlich besser auf die tatsächlichen Anforderungen (sowohl dieser Arbeit als auch eines Kunden) angepasst werden kann.

3.3 Rahmenprogramme

3.3.1 Versionskontrolle

Versionskontrolle ist ein sehr wichtiges Thema, sobald in einem Team oder in Kooperation programmiert werden soll. Zwar wird diese Arbeit und das daraus entstehende System in Einzelarbeit gefertigt, jedoch muss für die weitere Entwicklung und Fortführung ähnlicher Themen die Nachvollziehbarkeit des Programmierens sichergestellt werden.

Hierbei helfen sogenannte SCM[4]-Programme, die zum einen eine zentrale Anlaufstelle für die gesamte Code-Basis des Projekts darstellen, zum anderen auch eine Historie der Änderungen an jeder einzelnen Datei anlegen und protokollieren. [BCS11]

Die bekanntesten Vertreter dieser Gattung von Programmen sind SVN[5] und Git. Es existieren hierzu zahlreiche umfangreiche Rahmenprogramme, die sich der Möglichkeiten der jeweiligen API zunutze machen und so eine Umgebung für Programmierer schaffen, in welcher sie ohne Gefahr Änderungen am Code vornehmen können. Dies ist deshalb möglich, weil die SCM-Programme auch einen Mechanismus anbieten, der jegliche Änderungen an einer Datei (auch Löschung und Verschiebung) rückgängig machen kann.

Für dieses Projekt wurde ein proprietäres Programm zum Verwalten des Source-Codes verwendet: MKS[6]. Dieses basiert lose auf den Ideen der verbreiteten Systeme, bindet jedoch Änderungen am Code sehr viel stärker an dafür zwingend notwendige Anforderungen, die auch im MKS verwaltet werden.

So muss jede Änderung an der bestehenden Code-Basis durch eine Anforderung *begründet* sein - diese können wiederum nur durch definierte Prozesse in das System gelangen. So kann sichergestellt werden, dass niemand eine Änderung am Code durchführen kann, ohne dass dazu ein tatsächlicher Bedarf vorhanden ist.

Dies unterscheidet MKS stark von SVN oder Git - dort kann jeder mit Zugang zum System und den entsprechenden Rechten auch Änderungen am Code vornehmen. Um jedoch eine möglichst feine Kontrolle darüber zu haben, *warum* am Code etwas geändert wurde, ist dies bei MKS nicht möglich.

[4] Source Code Management
[5] Subversion
[6] Mortice Kern Systems Integrity & Source

3.3.2 Dokumentation

Zudem ermöglicht MKS die komplette Überwachung des *Lebens* eines Programmes durch die Implementierung von ALM[7]. Dies dient hauptsächlich zur Dokumentation der Anforderungen und aller nicht-code-bezogenen Teile eines Programms oder Systems.

Dies reicht von der Erfassung der Anforderungen des Kunden an das Programm (Lastenheft) über die Formulierung der Systemanforderungen (Pflichtenheft) bis hin zur genauen Dokumentation der Testabläufe und Ergebnisse. So wurden alle Anforderungen für die Kundenanwendung auf diese Weise erfasst und mit entsprechenden Änderungen im Code verknüpft. [Cha08]

Das Grundsystem (Framework) ist jedoch nicht Teil der im ALM-System abgelegten Dokumentation, da es nur als Basis für die Kundenanwendung dient und diese letztendlich an den Kunden ausgeliefert wird. Jedoch können deren Anforderungen auch für das Framework übernommen werden und damit abgedeckt werden. Die vorliegende Arbeit ist die Hauptdokumentation des Frameworks.

3.3.3 Entwicklungsumgebung

Um eine reibungslose Entwicklung zu ermöglichen, ist eine IDE[8] unabdingbar. Durch sie ist der Programmierer in der Lage, in einem einzigen System zu entwickeln, zu testen und die fertige Applikation in einen auslieferungsfähigen Zustand zu versetzen.

Durch die Technologieentscheidung für die Entwicklungssprache C# fiel die Wahl der verwendeten IDE unmittelbar auf das ebenfalls von Microsoft entwickelte und vertriebene *Visual Studio*.

Der Vorteil der Verwendung einer proprietären IDE war hier klar gegeben: Die enge Verzahnung des .NET-Frameworks in Windows, Visual Studio und auch C# ermöglicht eine sehr systemnahe Entwicklung. Dass dies selbstverständlich der Portierbarkeit der Lösung abträglich ist, muss hier nicht extra erwähnt werden.

Ebenso lässt sich mit Hilfe der IDE eine einfach zu bedienende Installationsroutine entwickeln, die das Endprodukt der Kundenanwendung auf den Kundenrechnern installieren lässt. Dies ist eine in Visual Studio integrierte Möglichkeit der Publikation von entwickelten Programmen.

Allerdings wird durch die Anforderungen an das Framework klar, dass vorerst ausschließlich Windows-Umgebungen zu beachten sind – zumindest für den ersten Entwurf und die Kundenanwendung.

[7] Application Lifecycle Management
[8] Integrated Development Environment

3.3.4 Bibliotheken

Um möglichst wenig Programmcode neu zu erfinden, wurde bei der Entwicklung stark auf bereits vorhandene Bibliotheken gesetzt. Bereits die Entscheidung zur Entwicklung in C# in der .NET-Umgebung von Microsoft ist im Grunde genommen bereits eine Verwendung von vordefinierten APIs und Bibliotheken.

Für die meisten Programme, die entwickelt werden, existieren ähnliche und immer wiederkehrende Anforderungen, z. B. für das Mitschreiben von Lognachrichten oder die Präsentation von Daten in einer grafischen Benutzeroberfläche – der GUI[9].

Es wäre sowohl ökonomisch als auch entwicklungstechnisch äußerst fragwürdig, all diese Komponenten für jedes neue Projekt aufs neue von Grund auf zu entwickeln und umzusetzen. Daher wurde bei der Entwicklung des Systems auf möglichst viele bereits bestehende Bibliotheken aufgebaut, um einerseits die Stabilität zu erhöhen (da bereits entwickelte Systeme vollständig getestet sind) und andererseits die Wiederverwendbarkeit und Zukunftssicherheit zu garantieren (da die Unterstützung und Weiterentwicklung von etablierten System meistens besser sichergestellt ist als bei internen Entwicklungen).

.NET-Framework Wie bereits erwähnt ist bereits die Entscheidung, mit C# zu programmieren, eine implizite Verwendung einer Bibliothek. Das .NET-Framework ist, vergleichbar mit der von Java bekannten JVM[10], eine virtuelle Laufzeitumgebung, die es Programmen, die deren API verwendet, ermöglicht, auf verschiedenen Hardwarekonfigurationen die gleichen Ergebnisse zu liefern.

Um dies zu ermöglichen, verwendet das .NET-Framework eine CLR[11], die den Code der Applikationen interpretiert und ausführt. Ähnlich wie der Bytecode Javas werden .NET-Programme nicht in Maschinencode kompiliert, sondern in die sogenannte CIL[12]. Diese *Sprache* wird durch ein Interface definiert, die CLI[13]. Diese Schnittstelle kann von jeder Programmiersprache verwendet werden, um gültigen CIL-Code zu produzieren. [The10]

Dadurch kann man, anders als bei Java, in verschiedenen Programmiersprachen entwickelte Programmme trotzdem in der gleichen Laufzeitumgebung ausführen, da sie alle auf die CIL zurückgeführt werden. Das ermöglicht es Programmierern, die verschiedene Sprachen beherrschen (C#, VB.NET oder JScript), an einem gemeinsamen Projekt mitzuwirken, ohne dass es zu groben Fehlern kommen wird.

[9] Graphical User Interface
[10] Java Virtual Machine
[11] Common Language Runtime
[12] Common Intermediate Language
[13] Common Language Infrastructure

Log4NET Auch wenn Fehler und Probleme von vornherein möglichst ausgeschlossen werden sollen, kann nicht garantiert werden, dass die ausgelieferte Software und das zugrunde liegende System in jedem Fall fehlerfrei arbeitet. Um dem Benutzer aber unschöne Fehlermeldungen zu ersparen, muss das System in der Lage sein, seine Nachrichten über einen Kanal *nach draußen* zu bringen.

Hierfür sind Logdateien gedacht. In diese kann das System all seine Meldungen, seien es Hinweise, Warnungen oder schwerwiegende Fehler, hineinschreiben, ohne dass es den Benutzer stört. Diese Dateien können dabei helfen, im Nachhinein Fehlverhalten zu entschlüsseln und den Ursachen auf den Grund zu gehen.

Zu diesem Zweck wird in allen während dieser Arbeit entwickelten Systemen die Bibliothek *Log4NET* der *Apache Software Foundation* eingesetzt. Diese ermöglicht es, Lognachrichten nicht nur auf einfache Art und Weise zu schreiben, sondern auch z. B. in unterschiedlichen Ausgangsformaten oder auf mehreren unterschiedlichen Kanälen. [Apaa]

3.3.5 Vorgehensweise

Um das Projekt und die Enwicklung des Systems einen definierten Rahmen und Ablauf zu geben, ist eine Vorgehensweise unablässlich. Da sich die Anforderungen des Kunden, für den die Kundenanwendung entwickelt wird, möglicherweise während des Projekts ändern könnten, wurde ein agiler Ansatz gewählt, der durch wiederholte Iterationen eine Veränderung der Anforderungen ermöglicht und erlaubt, dass das Programm zyklisch getestet und durch den Kunden abgenommen werden kann.

Da die meisten agilen Vorgehensweisen für Teams ausgelegt sind, ließ sich keines finden, das den Anforderungen an einen Entwickler und einen Projektleiter bzw. Kunden entspricht. Dies führte dazu, dass ein eigener Vorschlag zur Vorgehensweise ausgearbeitet und umgesetzt wurde, der sich aber stark an bereits etablierten Systemen orientiert.

So geschieht auch hier die Entwicklung in einzelnen Iterationen, zwischen denen jeweils eine Rücksprache zwischen den einzelnen Mitgliedern des Projektes stattfindet, ob selbiges noch im Rahmen der Anforderungen verläuft oder Anpassungen gemacht werden müssen. Dafür wurde ein an bekannte agile Methoden angelehntes eigenes Projektablaufmodell gewählt, welches die in Anhang B dargestellte Projektplanung ermöglichte.

Während der als *Entwicklung* bezeichneten Phase war das Ziel, eine möglichst effiziente Programmierumgebung für den Umsetzungsteil der Arbeit zu erstellen, die schnelle Änderungen am Code erlaubt und gleichzeitig eine gute Nachvollziehbarkeit der erledigten Arbeit ermöglicht. Die gesamte Phase lief intern durch mehrere Iterationen von Entwicklung, Dokumentation, Tests und abschließender Revision durch den Projektleiter.

Die einzelnen Iterationen wurden hier nicht aufgelistet, da sie für den Abschluss der Bachelorarbeit nicht relevant sind. Diese Arbeit selbst wurde mit der in Anhang B gezeigten Planung durchgeführt.

Hierzu wurde das in Unterunterabschnitt 3.3.1 und Unterunterabschnitt 3.3.2 vorgestellte System tiefgehend in die Entwicklung und Anforderungserfassung integriert. Die Ergebnisse sind in Anhang C festgehalten.

3.4 Entwurf

3.4.1 Aufteilung der Projekte und Aufgaben

Um die Aufgaben der einzelnen Programmkomponenten möglichst gekapselt und voneinander unabhängig zu gestalten, wurde das System in vier verschiedene Einzelprojekte aufgeteilt. Diese sind im Einzelnen[14]:

DocomFramework Das Framework zur Komposition. Hier sollen die Schnittstellen für kundenspezifische Plug-Ins sowie die allgemeine Funktionalität gebündelt werden. Dies ist der Teil des Systems, der immer wieder verwendet werden soll und der die Grundlage für alle weiteren Kompositionsprogramme darstellt.

DocomGUI Die grafische Benutzeroberfläche, die eine allgemeine Möglichkeit bietet, Dokumente, Kompositionen und Plug-Ins zu verwalten. Sie soll ermöglichen, ein Verständnis dafür zu bekommen, wie sich die Plug-Ins verhalten und welche Ergebnisse die Kompositionen erzeugen.

CustomerDocom Die Kundenanwendung, die nach Kundenanforderungen spezifiziert wurde. Sie soll das Framework verwenden und durch eigene, den Anforderungen entsprechende Plug-Ins bestimmte Aufgaben erfüllen. Das Ergebnis dieses Projektes ist eine ausführbare Datei, die der Kunde auf seinem Rechner installieren und verwenden kann.

CustomerDocomPlugins Die Plug-Ins für die Kundenanwendung. Diese werden in einem extra Projekt erstellt, da sie losgelöst von der Programmlogik von *CustomerDocom* sein sollen. So lassen sich Erweiterungen oder Fehler in den Plug-Ins einfacher austauschen, da nur diese Ausgangsdateien ersetzt werden müssen und nicht die komplette Applikation.

Die Aufteilung auf mehrere verschiedene Projekte ermöglicht es, sie in verschiedenen Arten auszuliefern. So wird das Framework und die Plug-Ins als kompilierte (im .NET-Sinn, also in der CIL) Programmbibliotheken, in der Windows-Umgebung besser bekannt als DLL[15] bekannt, ausgeliefert. Dies erlaubt anderen Programmierern, die darin definierten Klassen und Namespaces zu verwenden.

Im Gegensatz dazu wird die GUI und die Kundenanwendung als EXE[16] ausgeliefert, da diese Projekte direkt angewendet werden sollen und nicht als Erweiterung oder Grundlage für andere Programm dienen.

[14] *Docom* ist der Arbeitstitel für das Projekt; eine Kombination aus *Doc*ument und *Com*poser
[15] Dynamic Link Library
[16] Executable

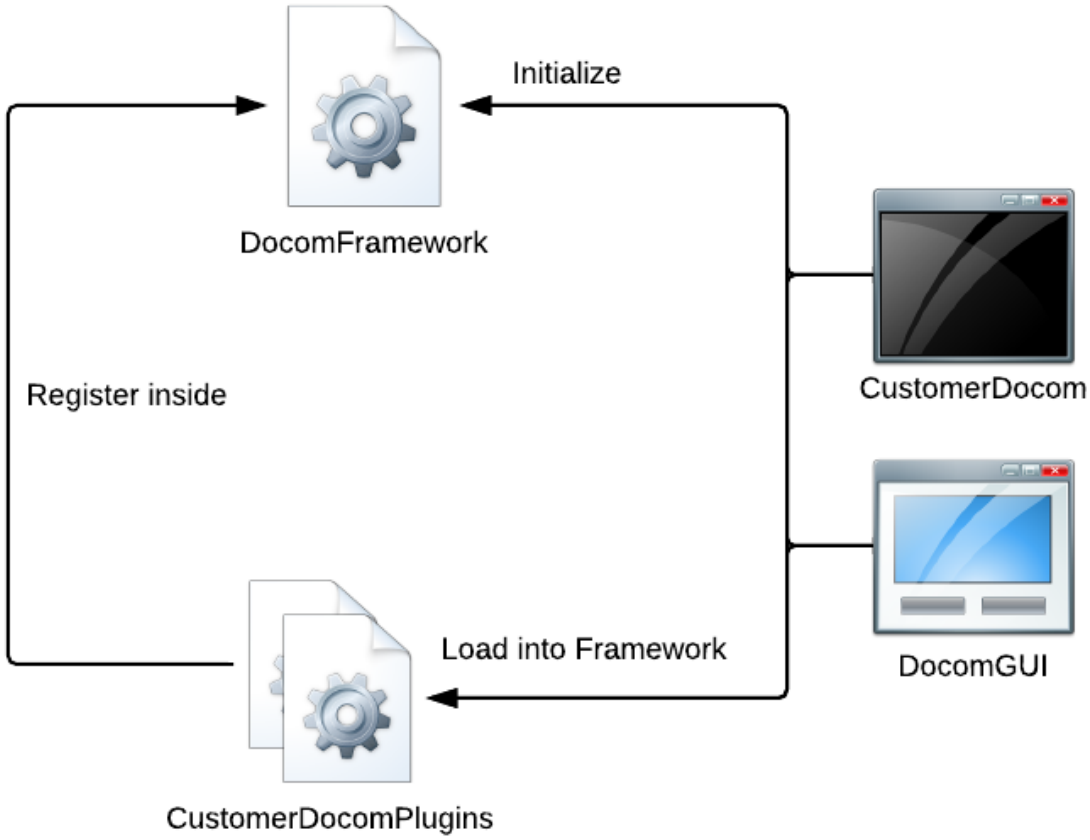


Abbildung 1: Aufgeteilte Projektstrukur und Interaktion zwischen den einzelnen Projekten

Wie in Abbildung 1 zu erkennen ist, initialisieren die ausführbaren Programme (DocumGUI oder CustomerDocom) das in Form einer Programmbibliothek eingebundene Framework. Ebenso registrieren sie die zu ladenden Plug-Ins (auch in Form von Programmbibliotheken) beim Framework, sodass dieses über die Existenz der Plug-Ins informiert ist.

3.4.2 Konzept zur Datenkapselung

Um das System im Hinblick auf die Modularität (s. Unterabschnitt 2.2) robust zu entwickeln, war Datenkapselung ein wichtiges Thema bei der Konzeption. Die Daten, die dafür in Frage kommen, wurden in einem Modell zusammengefasst. Dabei sind folgende Daten berücksichtig worden:

Dokumente sind sowohl alle Eingabedokumente sowie das Ausgabedokument, das durch die Komposition erstellt wird. Sie spiegeln ein physisch vorhandenes Dokument auf der Arbeitsmaschine wider.

Dokumentblöcke setzen ein Dokument zusammen. Die Eingangsdokumente können aus mehreren Dokumentblöcken bestehen, die vom Framework verwaltet werden können.

Ergebnisblöcke sind das Gegenstück zu den Dokumentblöcken auf der Ausgabeseite des Frameworks. Sie werden durch das Framework, bzw. dessen Plug-Ins, aus den Dokumentblöcken erzeugt und damit für die weitere Verarbeitung zur Verfügung gestellt.

Sie sind in verschiedenen Typen vorhanden, z. B. als Datei, Ordner oder Inhalt. Je nach Typ soll das für die Komposition zuständige Plug-In sie verschieden verarbeiten.

Weiterhin können sie Kinder enthalten, die ihrerseits wieder Ereignisblöcke eines bestimmten Typs sind. So entsteht eine Baumstruktur, die im Endeffekt das Ergebnis der Komposition widerspiegeln soll.

Diese Datenmodelle sind die Basis, auf welcher das Framework und dessen Plug-Ins ihre Arbeit verrichten sollen. Diese Arbeit besteht im Wesentlichen daraus, die *Dokumente* und deren *Dokumentblöcke* durch verschiedene Regeln neu zusammenzusetzen und in *Ereignisblöcke* umzuwandeln.

Wie diese Regeln definiert sind, wird durch die entsprechenden Plug-Ins selbst bestimmt. Hier gibt das Framework keine Vorgaben, da der Aktionsspielraum der Plug-Ins möglichst groß bleiben soll. Dies führt unweigerlich zu der Überlegung, wie das System als ganzes Stabilität sicherstellen kann, wenn den Plug-Ins keine Grenzen gesetzt werden, solange sie sich an die vom System bereitgestellten Schnittstellen halten.

3.4.3 Überlegungen zur Sicherheit

Wenn man Plug-Ins und Programme von Drittanbietern in ein Framework integriert und ausführt, ist die Sicherheit der zugrunde liegenden Umgebung immer ein wichtiger Bestandteil der Konzeption eines Systems. Für das Kompositionsframework ist diese Prämisse offensichtlich erfüllt, sodass zumindest kurz über die Sicherheit der Ausführung nachgedacht werden muss.

Es gibt verschiedene Ansätze, wie man die Sicherheit von modularen Systemen erhöhen oder garantieren kann. So ist es beispielsweise, wenn dies auch nichts mehr mit Sicherheit zu tun hat, durchaus eine valide Möglichkeit, komplett auf die Kontrolle der Sicherheit des Systems zu verzichten. Jedes Plug-In kann seine Arbeit ungehindert verrichten, und das umgebende Framework bietet den Funktionalitäten der Plug-Ins keinerlei Einschränkungen.

Dies ist sowohl die einfachste als auch unsicherste Möglichkeit, einem System die modulare Ausführung von Programmcode zu ermöglichen. Jedoch gibt es eine Möglichkeit, zumindest eine teilweise Sicherheit auch in einer solchen Ausführungsumgebung einzuführen – durch Inspektion des Programmcodes der Plug-Ins.

Dies wird vor allem bei Systemen, die große Zahlen an Drittentwicklern aufweisen und eine zentralisierte Möglichkeit besitzen, Plug-Ins zu beziehen, durchgeführt. Hierbei überprüfen Mitarbeiter des Grundsystems alle eingereichten Plug-Ins auf Unsicherheiten oder Schadcode. Nur wenn diese Prüfung ergibt, dass das Plug-In vertrauenswürdig und sicher ist, wird es für die Installation freigegeben.

Um wenigstens Schaden am Framework selbst zu verhindern, kann es zusätzlich eine einheitliche Schnittstelle für Plug-Ins zur Verfügung stellen, die den Zugriff auf interne Funktionen und Daten verbirgt. So könnten Plug-Ins zwar weiterhin prinzipiell schädlichen Code ausführen, jedoch würde dadurch zumindest die Integrität des Frameworks nicht beeinträchtigt werden.

Eine weitere Möglichkeit ist es, die Plug-Ins in einer Art abgekapseltem System auszuführen, in welchem sie nur beschränkte Rechte haben. Dies ist gerade bei Desktop-Applikationen besonders nützlich, da sich so sehr genau und streng kontrollieren lässt, welche Aktionen von welchen Plug-Ins ausgeführt werden können.

C# bietet in diesem Zusammenhang die Möglichkeit, Programme innerhalb einer sogenannten *AppDomain* auszuführen. Diese stellen isolierte Ausführungsbereiche dar, in welchem Programmen garantiert wird, dass sie keine anderen Programme durch Aktionen beeinflussen. Das gesamte Prozessmodell von .NET ist auf diese Art gekapselt und ermöglicht so das Programmieren ohne darauf achten zu müssen, ob und wie Daten und Methoden intern verarbeitet werden. [Micb]

Zusätzlich dazu ermöglicht *Code Access Security* die fein granulierte Kontrolle darüber, welche Aktionen das Programm ausführen darf. Dabei läuft eine Assembly in einem abgeschlossenem Sicherheitskontext, für welchen nur bestimmte Berechtigungen erteilt sind. [Kos04]

Bei der Konzeption des Systems wird zunächst auf die Implementierung von Sicherheitsmechanismen zugunsten der hier als erste aufgeführten Variante verzichtet. Zudem ist es für die prototypische Entwicklung nur von geringer Relevanz, das Framework gegen die Ausführung von Schadcode durch Drittanbieter zu schützen, da alle Entwicklungen von Plug-Ins durch interne Mitarbeiter durchgeführt werden sollen.

4 Umsetzung des Grundsystems

4.1 Das Framework

4.1.1 Verwaltung der Plug-Ins

Ausgehend von der Konzeption wurde zunächst mit der Entwicklung des in Unterunterabschnitt 3.4.1 beschriebenen *DocomFramework*s begonnen. Dieses sollte die Funktionen des Systems überwachen, die Zugriffe kapseln und eine zentrale Verwaltungsstelle für alle geladenen Plug-Ins und Daten darstellen.

Zu diesem Zweck benötigt das Framework zunächst öffentliche Eigenschaften, in welchen die Menge der geladenen Plug-Ins hinterlegt ist. Diese Eigenschaften – bei C# *Properties* genannt – sind ein integraler Bestandteil der Sprache. Sie ermöglichen es, Eigenschaften Intelligenz zu verleihen, ohne dass dies von außen unmittelbar sichtbar wird.

Listing 1: Eigenschaften der Frameworkklasse

```
1 public class Framework {
2     public IList<DocumentPlugin> DocumentPlugins {
3         get { return documentPlugins; }
4     }
5     private IList<DocumentPlugin> documentPlugins;
6     public IList<CompositionPlugin> CompositionPlugins {
7         get { return compositionPlugins; }
8     }
9     private IList<CompositionPlugin> compositionPlugins;
```

In Listing 1 erkennt man, dass der `get`-Block der beiden Eigenschaften `DocumentPlugins` und `CompositionPlugins` als eigene Methode gesehen werden kann. Dieser wird vom .NET-Framework automatisch aufgerufen, sobald eine Klasse einen Zugriff auf die entsprechende Eigenschaft aufruft.

So lassen sich, vor dem Zurückgeben des eigentlichen Wertes, z. B. Lognachrichten schreiben oder andere wichtige Operationen ausführen. Die Eigenschaften von C#-Klassen lassen sich damit direkt mit den bekannten `getX()` und `setX()`-Methoden von Java-Klassen vergleichen.

Auch das Setzen von Eigenschaften kann in C# in solche Blöcke gekapselt werden, in welcher der übergebene Wert überprüft und validiert werden kann. Im Unterschied zu Java jedoch sind diese Eigenschaftenmethoden von außen nicht direkt ersichtlich, d. h. die aufrufende Methode kann nicht erkennen, ob eine Eigenschaft direkt übernommen wird oder durch eine Eigenschaftsmethode läuft. Dies ist ein erheblicher Vorteil gegenüber dem klassischen Erstellen von *gettern* und *settern*, da keine aufrufende Methode Wissen darüber benötigt, wie die jeweils andere Klasse die Daten verarbeitet.

Durch das Weglassen eines `set`-Blocks für die beiden Eigenschaften sind sie automatisch als *nur lesen* markiert. So können diese beiden Eigenschaften niemals von außen durch andere Objekte ersetzt werden. Was dennoch funktioniert, ist die Manipulation des Objektes selbst, in diesem Fall einem, das die Schnittstelle `IList<T>` implementiert.

Dabei ist zu beachten, dass die Implementierung der Listen selbst nicht in den Eigenschaften festgeschrieben ist. `IList<>` ist ein generisches Interface, welches Listen von Objekten jedweder Art erzeugen kann. Wie genau die Implementierung der Liste aussieht, ist im Interface nicht definiert. Dies bezeichnet man als *Loose Coupling*, einem wesentlichen Konzept moderner objektorientierter Programmierung.

Durch das Verstecken der tatsächlichen Implementierung der Liste kann diese im Nachhinein problemlos ausgetauscht werden, da sich alle Programme und Klassen, die mit dieser arbeiten, rein auf die Schnittstelle verlassen und nicht auf die tatsächliche Liste, die das Interface nur implementiert. [Mil]

4.1.2 Hinzufügen von Plug-Ins

So lassen sich von außen problemlos weitere `DocumentPlugin`s oder `CompositionPlugin`s hinzufügen, wie in Listing 2 gezeigt. Diese Art des Hinzufügens funktioniert jedoch nur mit bereits instanzierten Objekten der jeweiligen Plug-Inklasse.

Listing 2: Hinzufügen von instanzierten Plug-Ins zum Framework

```
1 [...]
2 Framework docom = new Framework();
3 DocumentPlugin plugin = // Pluginobjekt erzeugen
4 docom.DocumentPlugins.Add(plugin);
```

Die Plug-Inklassen sind als abstrakte Klassen konzipiert, die als Schnitstelle für das Framework dienen. Damit eine Klasse immer entweder ein `DocumentPlugin` oder ein `CompositionPlugin` sein muss, wurden diese beiden Schnittstellen nicht als, dem Namen eigentlich entsprechende, `Interface`s erstellt, sondern als echte, wenn auch abstrakte, Klassen.

Auf diese Weise kann sichergestellt werden, dass keine Klasse sowohl als Plug-In für Dokumente als auch für Kompositionen zuständig sein kann. Dies sichert die Einhaltung des *SOLID*-Prinzips. Dies ist ein mnemonisches Akronym, das für die fünf grundlegenden Anforderungen an gute objektorientierte Software steht. [Lot10][Mar09]

Single Responsibility Principle — Jedes Objekt soll nur für eine einzige Aufgabe zuständig sein

Open/Closed Principle — Softwareeinheiten sollen offen für Erweiterung und geschlossen gegen Veränderung sein

Liskov Substitution Principle — Objekte müssen jederzeit durch Objekte ihrer Subklassen ersetzt werden können, ohne dass die Korrektheit des Programms dadurch verletzt wird

Interface Segregation Principle — Viele benutzerspezifischen Schnittstellen sind besser als ein allgemeingültiges Interface

Dependency Inversion Principle — Es ist besser, sich auf Abstraktionen zu verlassen als auf Konkretisierungen

All diese Prinzipien sind, sogar noch vor den allgemein bekannten *Design Patterns* für objektorientiertes Programmieren, wichtige Bestandteile des Konzepts und der Umsetzung komplexer Software. [Mar00]

Das Framework muss, neben der Verwaltung bereits instanzierter Plug-Ins, auch in der Lage sein, noch nicht instanzierte Plug-Ins, die als DLL vorliegen, zu laden. Um dies zu bewerkstelligen, kommt ein weiterer integraler Bestandteil moderner Programmiersprachen zur Anwendung: *Reflection*.

Als Reflection oder Introspection wird die Fähigkeit von Programmen bezeichnet, ihren eigenen Code zu kennen und sogar zu bearbeiten. In Hochsprachen wird damit auch die Möglichkeit in Verbindung gebracht, Informationen über andere Klassen abzufragen. Die Klassen, die dies ermöglichen, sind fester Bestandteil des .NET-Frameworks (wie auch in der JVM von Java). [Micd]

Wie in Unterunterabschnitt 3.4.1 beschrieben, sollen Plug-Ins für die Auslieferung als kompilierte Bibliotheken vorliegen. Um diese im Framework laden zu können, muss dieses nichts weiter als den Dateinamen der zu ladenden Assembly kennen. Mit diesem Wissen kann das Framework über Reflection und Introspection die DLL-Datei analysieren und die darin enthaltenen Plug-Ins anhand ihrer Signatur laden und dem ausführenden Programm zur Verfügung stellen.

Listing 3: Laden von in einer DLL kompilierten Plug-Ins

```
1  public void RegisterAssembly(String pluginFile) {
2      Assembly assembly = Assembly.LoadFile(pluginFile);
3      foreach (Type type in assembly.GetTypes()) {
4          if (type.IsSubclassOf(typeof(DocumentPlugin))) {
5              DocumentPlugin plugin = (DocumentPlugin)Activator.CreateInstance(
                   type);
6              DocumentPlugins.Add(plugin);
7          }
8          else if (type.IsSubclassOf(typeof(CompositionPlugin))) {
9              CompositionPlugin plugin = (CompositionPlugin)Activator.
                   CreateInstance(type);
10             CompositionPlugins.Add(plugin);
```

Listing 3 zeigt den vergleichsweise einfachen Prozess, der eine angegebene Datei anhand ihres Dateinamens als Assembly lädt. Zunächst müssen alle in der Assembly vorhandenen Typen einzeln überprüft werden (Zeile 3). Typen können sowohl Klassen, Schnittstellen, Arrays, Werte, Auflistungen oder Parameter sein[Mic10].

Daher muss im nächsten Schritt zwingend überprüft werden, ob der Typ eine Subklasse einer der beiden Plug-In-Klassen ist. Damit lässt sich in einem Schritt darauf prüfen, ob der Typ überhaupt eine Klasse ist und zu welchem der beiden Plug-Intypen er gehört (Zeile 4 und 8).

Je nachdem, welcher Klassentyp vorliegt, wird die Klasse über den sogenannten `Activator` instanziert. Diese Klasse „[enthält] Methoden, mit denen Objekttypen lokal oder remote erstellt und Verweise auf vorhandene Remoteobjekte abgerufen werden können.“[Mica]

So entsteht nach dem Ausführen der `CreateInstance()`-Methode ein neues Objekt der jeweiligen Subklasse der Plug-Ins. Diese wird dann der jeweiligen Liste der geladenen Plug-Ins hinzugefügt.

4.1.3 Plug-Ins auswählen und erhalten

Damit mit den im Framework geladenen Plug-Ins gearbeitet werden kann, müssen diese nach außen weitergegeben werden können. Da das Gesamtsystem nach Unterabschnitt 2.5 hauptsächlich mit Dokumenten in Dateiform umgehen muss, liegt es nahe, als Unterscheidungsmerkmal für Plug-Ins ihre Fähigkeit heranzuziehen, mit verschiedenen Dokumenttypen umgehen zu können.

So wurde für das Akquirieren von geladenen Plug-Ins eine Methode entwickelt, die je nach übergebenem Dokumenttyp ein dafür passendes, geladenes Plug-In zurückliefert. Da es in C# keine native Implementierung für einen Datentyp gibt, der die Endung eines Dateinamens darstellt, wurde diese für diesen Zweck entwickelt.

Dies ist besonders deshalb wichtig, da keine einheitliche Handhabung des Punktes bei Dateinamen besteht. Je nach Implementierung wird dieser zur Dateiendung gezählt oder eben nicht. Um diese Inkonsistenz zu beseitigen, werden alle Operationen, die mit der Dateiendung arbeiten, mit dieser neue Klasse arbeiten.

Gleichzeitig muss jedoch auch im Hinblick auf die Erweiterung des Systems die Möglichkeit in Betracht gezogen werden, dass spätere Plug-Ins möglicherweise nicht alleine mit Dateien, sondern auch mit anderen, abstrakten Datenquellen arbeiten müssen. Um auch dieses Szenario behandeln zu können, wird die Klasse möglichst allgemein gehalten.

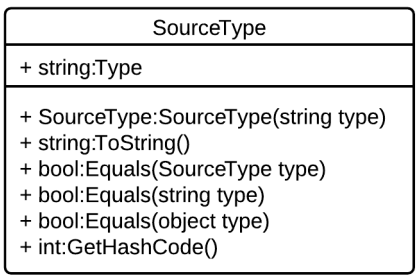


Abbildung 2: Die SourceType-Klasse

Mit der in Abbildung 2 gezeigten Klasse lässt sich nun ein Plug-In aus dem Framework erhalten, welches die Verarbeitung eines bestimmten Datentyps handhaben kann. Hierbei wird die Liste aller geladenen Plug-Ins nacheinander durchlaufen und das erste Plug-In zurückgegeben, welches (nach dessen *Aussage*) mit dem Dateityp umgehen kann.

Listing 4: Akquirieren eines Plug-Ins zur Verarbeitung von Dokumenten

```
1 public DocumentPlugin GetPluginFor(SourceType type) {
2     foreach (DocumentPlugin plugin in DocumentPlugins)
3         if (plugin.CanHandle(type))
4             return plugin;
5     throw new NotSupportedException("The type " + type + " cannot be handled by
          any plugin");
```

Eine weitere Möglichkeit, ein Plug-In aus dem Framework zu erhalten, ist, es direkt über seinen Typen anzufragen. Dies soll besonders bei kundenspezifischen Implementierungen helfen, ganz bestimmte Plug-Ins aus dem Framework zu erhalten, die anhand ihres Typs identifizierbar sind.

Hierfür wird ein Typvergleich zwischen dem Typ, der der Methode übergeben wird, und dem des geladenen Plug-Ins durchgeführt, der bei Erfolg zeigt, dass die beiden Typen auf dieselbe Klasse verweisen.

Listing 5: Akquirieren eines bestimmten Plug-Ins anhand seines Typs

```
1 public CompositionPlugin GetCompositionPlugin(Type ofType) {
2     foreach (CompositionPlugin plugin in CompositionPlugins)
3         if (ofType.AssemblyQualifiedName.Equals(plugin.GetType().
              AssemblyQualifiedName))
4             return plugin;
5     throw new ArgumentException("A composition plugin of type " + ofType.
          FullName + " could not be found");
```

Wird nun ein Plug-In für einen Kunden spezifisch programmiert, lässt sich über einen Aufruf dieser Methode das Plug-In aus dem Framework erhalten, wenn es vorher über eine der Methoden aus Listing 2 oder Listing 3 hinzugefügt worden ist.

Hierbei ist der If-Abfrage in Listing 5, Zeile 3 besondere Aufmerksamkeit zu widmen. Diese führt einen relativ simplen und daher auch unsicheren String-Vergleich zwischen den beiden Namen der Assemblies, welche die referenzierten Typen enthalten, durch. Dies ist aufgrund einer Eigenheit des .NET-Frameworks leider auf diese Art und Weise nötig, da ein Objektvergleich nicht erwartungsgemäß arbeitet.

Würde man die beiden `Type`-Instanzen `ofType` und `plugin.GetType()` direkt miteinander vergleichen, würde dieser Vergleich auf Gleichheit negativ verlaufen. Dies liegt daran, dass die Assembly, in welchem die Plug-In-Implementierung liegt, eine andere als diejenige des Frameworks ist.

Obwohl das ausführende Programm das Plug-In im Code referenziert, würde der Objektvergleich fehlschlagen, da das Framework ja, wie in Listing 3 gezeigt, die DLL-Assembly selbst lädt und instanziert. Somit ist für das Framework ein Objekt der Typ-Klasse für das Plug-In aus der DLL-Assembly ein anderes als jenes, das im Code der Applikation referenziert wird.

4.1.4 Freigeben von allokiertem Speicher

Auch wenn in C#-Programmen durch Garbage-Collection eigentlich keine expliziten Aufrufe der Destruktoren von Objekten durch das ausführende Programm selbst durchgeführt werden müssen und dadurch unvorhersehbar ist, wann ein Objekt aus dem Speicher entfernt wird[Küh09, S. 195], ist es dennoch in manchen Fällen angebracht, zumindest bestimmte Objekttypen durch das Aufrufen eines „Pseudo-Destruktors“selbst aufzuräumen.

Hierfür gibt es die `Dispose`-Methode, die im Gegensatz zum Destruktor einen expliziten Aufruf erfordert und damit „die weitere Vorgehensweise bei der Programmierung nachhaltig beeinflusst“[Küh09, S. 197]. Besonders bei der Programmierung mit der für die Kundenanwendung benötigten Office-Interop-Bibliothek ist das explizite Dereferenzieren von Objekten und das manuelle „Aufräumen“nach getaner Arbeit essentiell.

Aus diesem Grund implementieren sowohl das Framework selbst als auch alle Plug-In-Schnittstellen das `IDisposable`-Interface. Durch diese Schnittstelle muss eine Methode `Dispose()` implementiert werden, in welcher die allokierten Objektreferenzen gelöscht werden sollen und die Klasse für das *Aufsammeln* durch den Gargabe-Collector vorbereitet werden soll.

Die entsprechende `Dispose()`-Methode des Frameworks ruft nichts weiter als die `Dispose()`-Methoden aller geladenen Plug-Ins auf, und löscht danach die Klasseneigenschaften des Frameworks selbst. So ist sichergestellt, dass keinerlei Referenzen verwaisen und das Programm ordnungsgemäß beendet werden kann.

Listing 6: Dispose-Methode des Frameworks

```
1 public void Dispose() {
2     foreach (DocumentPlugin docPlugin in DocumentPlugins)
3         docPlugin.Dispose();
4     foreach (CompositionPlugin compPlugin in CompositionPlugins)
5         compPlugin.Dispose();
6     documentPlugins = null;
7     compositionPlugins = null;
```

Diese Methode wird besonders bei der Entwicklung der Kundenanwendung interessant, da diese sehr stark mit Office-Interop-Objekten arbeitet.

4.2 Die Plug-Ins

4.2.1 Dokumenten-Plug-In

Ausgehend von der Anforderung der Handhabung verschiedener Dokumenttypen wurde das `DocumentPlugin`, welches Dokumente entgegennimmt und sie in eine interne Datenstruktur überführt, entwickelt.

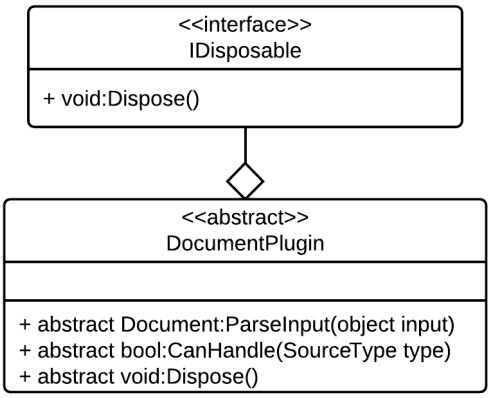


Abbildung 3: Die abstrakte DocumentPlugin-Klasse

Ein Plug-In, welches für das Einlesen von Dokumenten zuständig ist, muss von dieser abstrakten Klasse erben, damit es im Framework als `DocumentPlugin` erkannt werden kann. Hierbei zwingt die abstrakte Klasse die erbende, die drei ebenfalls abstrakten Methoden `Document:ParseInput()`, `bool:CanHandle()` und `Dispose()` zu implementieren. Nur, wenn alle diese Methoden in der Subklasse überschrieben werden, ist das Plug-In handlungsfähig und kann Arbeit leisten.

Dabei sieht die Schnittstelle für die Methoden (wie in Abbildung 3 gezeigt) das folgende Verhalten vor:

Die „ParseInput"-Methode Die `Document:ParseInput`-Methode ist für das Einlesen von Dokumenten zuständig. Sie muss ein `Document` zurückgeben, welches ein eigens definiertes Datenmodell ist und in Unterabschnitt 4.3 näher erläutert wird.

Wichtig ist hier zunächst nur, dass die Methode aus einem physisch vorhandenen Dokument ein abstraktes, nur im Datenmodell gehaltenes Dokument erzeugt. Dabei ist der Parameter, welcher der Methode übergeben wird, wie an der Signatur zu sehen, vom Typ `object`. Dies ist deshalb der Fall, weil es möglicherweise in Zukunft Plug-Ins geben könnte, die nicht ausschließlich mit auf dem ausführenden Rechner vorhandenen Dateien arbeiten sollen.

So ist es beispielsweise denkbar, dass ein Plug-In seine Daten aus einer Datenbank beziehen soll. Würde man die Signatur der Methode auf einen festen Übergabetyp wie `String` oder `FileInfo` ändern, würde man sie gegen diese Erweiterung verschließen und damit gegen das in Unterunterabschnitt 4.1.2 erläuterte *Open/Closed Principle* verstoßen.

Jedes Plug-In muss dem Entwickler, der mit ihm arbeitet, also in Form einer detaillierten Code-Dokumentation des Plug-Ins mitteilen, welche Art von Daten es für diese Methode erwartet.

Das Ergebnis hingegen ist immer ein abstraktes `Document`, in welchem das Plug-In alle relevanten Daten sammelt und kapselt. Wie mit diesen Daten umgegangen wird, liegt dann in der Entscheidung des Kompositions-Plug-Ins.

Die „CanHandle"-Methode Damit das Plug-In eine Möglichkeit hat, mit dem aufrufenden Programm zu kommunizieren und ihm mitzuteilen, welche Arten von Dokumenten bzw. Datenquellen es verarbeiten kann, ist diese Methode gedacht.

Mit ihr kann das Plug-In das übergebene `SourceType`-Objekt prüfen und entscheiden, ob es mit dieser Art von Datenquelle (oder Dateiendung) umgehen kann. Je nach der Entscheidung des Plug-Ins kann das aufrufendene Programm dann auf den Rückgabewert der Funktion reagieren, wie in Listing 7 gezeigt.

Listing 7: Beispiel der Verwendung der CanHandle-Methode

```
1 Framework docom = new Framework();
2 docom.RegisterAssembly("C:\\MyPlugins.dll");
3 SourceType type = new SourceType("doc");
4 DocumentPlugin pl-Plug-In docom.GetPluginFor(type);
5 if (plugin.CanHandle(type))
6     // Proceed handling
```

Der nachträgliche Aufruf von `CanHandle()` in Zeile 5 ist hier allerdings überflüssig, da die Framework-Methode `GetPluginFor()` diesen bereits auf alle geladenen Plug-Ins ausführt (s. Listing 4, Zeile 3), er soll lediglich zur Veranschaulichung einer einfachen Anwendung dienen.

Die „Dispose"-Methode Für die korrekte Freigabe von allokiertem Speicher und Ressourcen wird diese Methode implementiert, um dem Plug-In eine Möglichkeit zu geben, seine Arbeitsobjekte am Ende der Arbeit fachgerecht zu löschen. Diese Methode wird implizit durch die `Dispose()`-Methode des Frameworks aufgerufen, wie in Listing 6 gezeigt.

4.2.2 Kompositions-Plug-In

Der zweite wesentliche Bestandteil des Systems ist die Komposition der Dokumente, welche das *Dokumenten-Plug-In* in die interne Datenstruktur umgewandelt hat. Je nachdem, wie diese Komposition aussieht, wird das Ergebnisdokument eine andere interne Struktur haben.

Hierzu wurde ebenfalls eine abstrakte Klasse entwickelt, von der die konkreten Plug-Ins erben müssen (s. Abbildung 4).

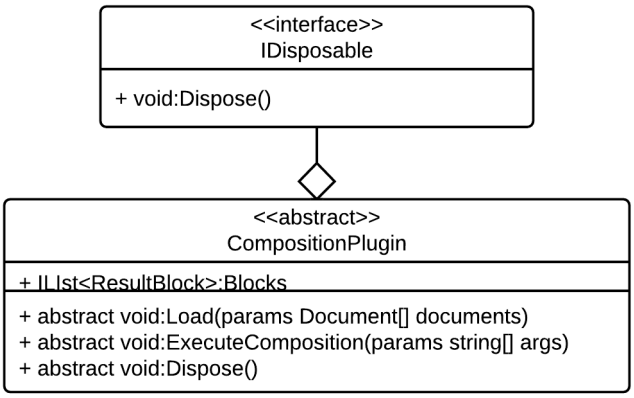


Abbildung 4: Die abstrakte CompositionPlugin-Klasse

Diese Plug-In-Klasse enthält neben den abstrakten Methoden, welche durch die konkrete Plug-In-Implementierung überschrieben werden müssen, noch ein weiteres, nicht abstraktes Feld: `Blocks`. Dieses ist eine Liste aus `ResultBlock`-Elementen. Diese spiegeln ein einzelne Einheit im Ergebnis der Komposition wieder und sind in Unterabschnitt 4.3 näher beschrieben.

Die „Load“-Methode Das Kompositions-Plug-In hat die Aufgabe, aus mehreren, durch das Dokumenten-Plug-In umgewandelten, `Document`-Objekten ein einzelnes Ergebnisdokument zu erzeugen. Hierfür müssen zunächst die `Document`-Elemente geladen werden.

Diesem Zweck dient die `Load`-Methode. Sie nimmt eine beliebige Menge an `Document`-Elementen entgegen. Diese Besonderheit ist durch das Schlüsselwort `params` möglich. Dieses Schlüsselwort vor einer Array-Deklaration ermöglicht es der Methode, nicht nur ein normales Array von `Document`-Elementen entgegenzunehmen, sondern auch eine kommaseparierte Liste derselbigen.

Somit sind die beiden in Listing 8 gezeigten Möglichkeiten des Methodenaufrufs im Inneren der Methode komplett identisch. Die Methode sowohl mit einem bereits deklarierten Array arbeiten als auch mit einem manuellen Aufruf der Methode mit einer eigens definierten Menge an Übergabewerten.

Listing 8: Beispiel der Auswirkungen von „params“

```
1 Document doc1 = new Document(), doc2 = new Document(), doc3 = new Document();
2 Document[] docArray = new Document []{ doc1, doc2, doc3 };
3 CompositionPlugin plugin = new CompositionPluginImpl();
4
5 // Beide Aufrufe erzielen das gleiche Ergebnis
6 plugin.Load(doc1, doc2, doc3);
7 plugin.Load(docArray);
```

Die „ExecuteComposition“-Methode Das tatsächliche *Ausführen* des Programms soll in dieser Methode geschehen. Sie nimmt, ähnlich wie die `Load()`-Methode eine beliebige Anzahl an `String`-Argumenten entgegen, da diese die einfachste Methode sind, Parameter an ein Programm zu übergeben.

Wie diese Parameter aussehen und was genau in ihnen stehen soll, muss das entwickelte Plug-In dem Entwickler über eine detallierte Code-Dokumentation selbst mitteilen. Hier wurde keine strenge Typprüfung eingeführt, um den Spielraum der Entwickler von Plug-Ins möglichst nicht einzuengen, die zu einem unsachgemäßen Verwenden des Frameworks führen könnten.

Die Methode sollte nach Möglichkeit ausschließlich mit den Daten in den `Block`-Element der `CompositionPlugin`-Klasse arbeiten und keine weiteren Einstellungsparameter benötigen. Da dies allerdings nur in seltenen Fällen möglich ist, liegt es in der Verantwortung des Plug-Inentwicklers, für die Beschaffung dieser Parameter zu sorgen, sei es über Konfigurationsdateien oder direkte Eingabe des Users.

Die „Dispose“-Methode vgl. Abschnitt 4.2.1, *Die „Dispose“-Methode*

4.3 Datenmodell

4.3.1 Modell für die Eingangsdaten

Um eine möglichst durchgängige Unabhängigkeit der Komposition von den Eingangsdaten zu erreichen, wurde eine spezielle interne Datenstruktur verwendet, in welche die Dokumenten-Plug-Ins die Eingangsdaten umwandeln. Die Kompositions-Plug-Ins müssen dann lediglich mit dieser einheitlichen Struktur umgehen können, um ihre Ergebnisse zu erzielen.

Grundlage des Datenmodells ist die Überlegung, dass jede Eingangsquelle als *Dokument* bezeichnet werden kann. Dass spätere Plug-Ins hier anstatt aus Dokumenten ihre Daten auch aus Datenbanken oder anderen Quellen beziehen könnten, schmälert die Gültigkeit dieser Aussage nicht, da auch diese Daten, abstrakt gesehen, Dokumenten mit Inhalten entsprechen werden. Davon kann ausgegangen werden, da sich das Gesamtsystem ja explizit auf die Komposition von Dokumenten spezialisiert.

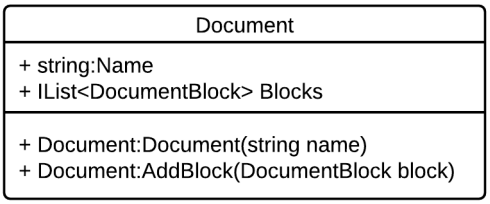


Abbildung 5: Die Document-Klasse

Hierzu wurde die Klasse des `Documents` erstellt. Ihre wesentlichen Bestandteile können Abbildung 5 entnommen werden. Interessant ist hierbei vor allem die Eigenschaft der `Blocks`.

Die „Blocks“-Eigenschaft Jedes Dokument besteht aus bestimmten Inhalten. Diese Inhalte müssen, ebenso wie der Bezeichner `Name` des Dokuments in eine interne Struktur überführt werden. Dafür wurden die `DocumentBlock`-Elemente erstellt. Sie setzen das gesamte Dokument zusammen und in ihnen ist der eigentliche Inhalt des Dokuments gespeichert.

Diese *Inhaltsblöcke* werden durch das Dokumenten-Plug-In aus dem Dokument extrahiert und in das Datenmodell überführt. Die einzelnen Blöcke sind dabei relativ einfache Datenklassen, wie in Abbildung 6 zu sehen ist.

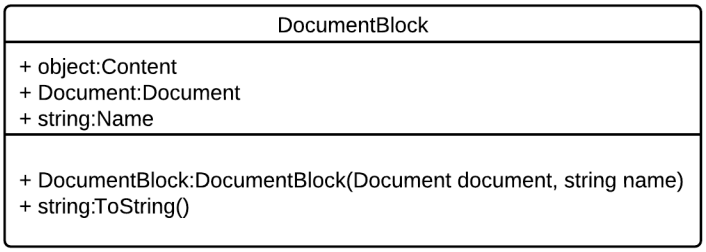


Abbildung 6: Die DocumentBlock-Klasse

Der tatsächliche Inhalt, der sich in den Dokumentblöcken befindet, wird im Datenmodell als `object` gespeichert. Dies soll die Möglichkeiten, die der Inhalt möglicherweise besitzen kann, möglichst nicht einschränken. Da jede Klasse und jeder Datentyp in C# von `object` erbt, kann so jeder mögliche Inhalt, den das .NET-Framework verarbeiten kann, in diesem Dokumentenblöcken gespeichert werden.

4.3.2 Modell für die Komposition

Beim Erstellen der Komposition aus den Eingangsdaten, bzw. den `Document`-Elementen, werden diese in eine neue Form gebracht, die dem Ausgangsdokument entsprechen soll. Hierbei wird die vorherige Struktur eines Dokumentes, welches aus vielen Dokumentblöcken besteht, aufgebrochen und auf einzelne Blöcke reduziert. Diese Blöcke sind ihrerseits verschachtelbar, sodass man eine baumartige Struktur in der Komposition erstellen kann.

Die Ausführung der Komposition wird ebenfalls durch das Plug-In anhand der intern gespeicherten Ergebnisblöcke, den `ResultBlocks` und gegebenenfalls weiteren übergebenen Parametern durchgeführt.

Die in Abbildung 7 gezeigte `ResultBlock`-Klasse besteht aus mehreren, für die Komposition relevanten Eigenschaften und Methoden.

Die „ChildBlocks“-Eigenschaft Um das Ergebnis einer Komposition und besonders dessen Verschachtelung in den Inhalten dynamisch zu halten, kann jeder `ResultBlock` beliebig viele Kindblöcke enthalten, die ihrerseits wieder vom `ResultBlock`-Typ sind. Dies ermöglicht eine beliebig tiefe Verschachtelung und die Erzeugung komplexer Strukturen in der Komposition.

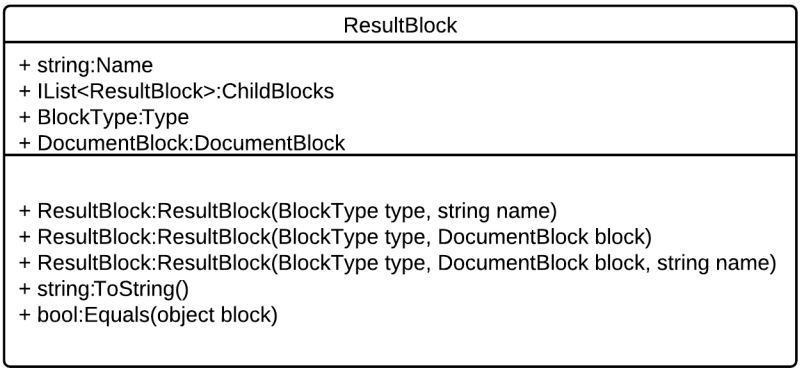


Abbildung 7: Die ResultBlock-Klasse

Wie die Blöcke durch das Kompositions-Plug-In verarbeitet werden können, hängt unmittelbar mit der `Type`-Eigenschaft des `ResultBlock`s zusammen.

Die „Type“-Eigenschaft Eine Komposition besteht zumeist aus einer einzelnen Ergebnisdatei. Um jedoch das System offen für spätere Veränderungen dieser Anforderung zu halten, wurde die `BlockType`-Enumeration eingeführt, die einem `ResultBlock` einen gewissen Typ zuordnet.

Diese sehr einfache Enumeration besitzt drei grundlegende Werte, die einen `ResultBlock` einem bestimmten Typ zu weisen:

FOLDER Der Block soll in der endgültigen Komposition als Ordner erstellt werden. Dies ist besonders nützlich, wenn die Kinder dieses Blocks weitere Dateien oder Ordner sind. Dies ermächtigt das Kompositions-Plug-In, nicht nur einfache Dateien, sondern komplexe Ordnerstrukturen zu erstellen.

FILE Der Block ist im Ergebnis eine physische Datei. Diese kann beispielsweise als Kind in einem `FOLDER`-Block liegen. `FILE`-Blöcke als Kinder von `FILE`-Blöcken sind jedoch widersprüchlich. Prinzipiell kann man sie zwar mit der vorhandenen Struktur erzeugen, jedoch muss das Kompositions-Plug-In damit vorhersehbar umgehen können.

BLOCK Ein tatsächlicher Inhaltsblock, der in einen `FILE`-Block geschrieben wird. Hierbei handelt es sich zumeist direkt um `DocumentBlock`-Elemente, die aus einem Eingangsdokument übernommen wurden.

Mit diesen Typen lassen sich sehr viele mögliche Kompositionen für die Eingangsdokumente erstellen. Diese können dann durch die Ausführung des Kompositions-Plug-Ins erstellt werden und in echte Dateien, Ordner und Inhalte *übersetzt* werden.

Die „DocumentBlock“-Eigenschaft Um einen festen Bezug zum ursprünglichen Inhalt des ResultBlocks zu halten, ist es ihm möglich (optional) eine Referenz auf seine Herkunft, in Form eines `DocumentBlock`s, zu halten. Dies ist besonders für die Blöcke des `BLOCK`-Typs interessant, da im entsprechenden `DocumentBlock.Content` der im Dokument enthaltene Inhalt des Blocks gespeichert ist.

Für die beiden anderen Block-Typen, `FOLDER` und `FILE`, ist dies jedoch weniger relevant. Nichtsdestotrotz können auch sie eine Referenz auf einen `DocumentBlock` halten, um die Erweiterbarkeit sicherzustellen.

4.4 Interaktion, Abläufe und Programminterna

4.4.1 Übersicht und genereller Ablauf

Der Gesamtablauf einer Komposition ähnelt dem Prozess, den Redakteure beim Zusammenstellen von Dokumenten selbst auch durchführen. So wird vorerst eine Auswahl getroffen, welche Dokumente für das Ergebnis überhaupt relevant sind. Diese werden dann anhand von bestimmten Regeln und Vorgaben in einer definierten Reihenfolge arrangiert und schlussendlich zu dem gewünschten Ergebnisdokument zusammengestellt.

Der in Abbildung 8 abgebildete Prozess teilt sich demnach in drei hauptsächliche Unterprozesse auf; dem *Überführen* der Eingabedokumente in die interne Datenstruktur, dem *Zusammenstellen* der Komposition und dem schlussendlichen *Ausführen* derselbigen.

Für die bessere Veranschaulichung der internen Abläufe und die Verwendung der Plug-In-Klassen werden in diesem Abschnitt zwei Beispiele erzeugt, die zur Demonstration der Möglichkeiten des Frameworks dienen werden. Dabei handelt es sich zum einen um ein sehr simples Plug-In, welches mehrere Word-Dokumente in einem Ergebnisdokument zusammenstellt und ein komplexeres Plug-In, das mit Datenbank-Tabellen und -Zeilen umgeht und diese in CSV[17]-Textdateien umwandelt.

4.4.2 Überführen in die interne Datenstruktur

Um die Anforderung nach der Verarbeitung unterschiedlicher Eingabetypen nachzukommen, muss zunächst jedes Dokument, bzw. jeder Eingabetyp, in die interne Datenstruktur überführt werden. Hierzu sind die bereits in Unterunterabschnitt 4.2.1 beschrieben Dokumenten-Plug-Ins zuständig.

Diese Plug-Ins müssen demnach in der Lage sein, eine Ressource eines bestimmten `SourceType`s (s. Abbildung 2) entgegen zu nehmen und diese auf eine vorhersehbare Art und Weise in `Document`-Objekte umzuwandeln.

[17] Comma Separated Values

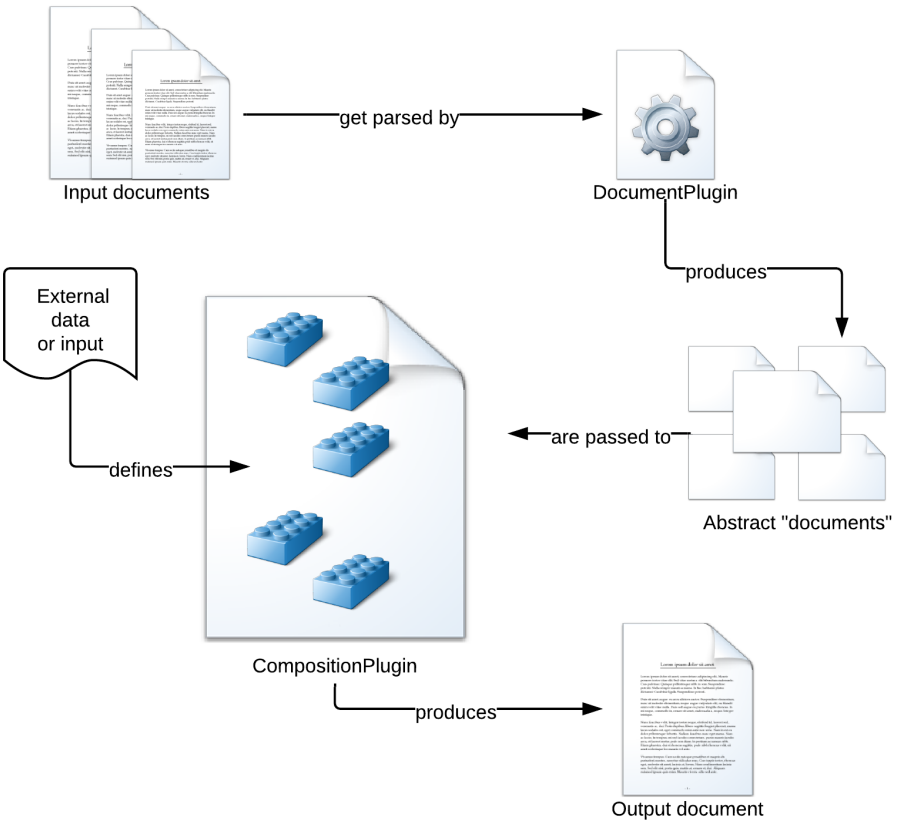


Abbildung 8: Ablauf einer Komposition

Beispiel: Word-Dokument Bei einem Word-Dokument könnte ein Plug-In z. B. den Dateinamen entgegennehmen und die entsprechende Datei dann zunächst in ein leeres `Document` umwandeln:

Listing 9: Beispiel eines DocumentPlugins für Word-Dokumente

```
1 public class DocPlugin : DocumentPlugin {
2   public override Document ParseInput(object file) {
3     FileInfo info = (FileInfo)file;
4     Document document = new Document(info.FullName);
5     return document;
6   }
7   public override Boolean CanHandle(SourceType type) {
8     return type.Equals("doc");
```

Das in Listing 9 gezeigte Beispiel demonstriert eine sehr vereinfachte Form der Verarbeitung, in welcher das erstellte Dokument keine weiteren Blöcke besitzt. Meistens sind jedoch Dokumente noch in einzelne Einheiten aufgeteilt.

Beispiel: Datenbank-Tabellen So ließe sich auch ein Plug-In schreiben, das aus einer Datenbank-Tabelle ein Dokument erstellt und aus den einzelnen Einträgen dieser Datenbank dann die Dokumentenblöcke erzeugt. Hierbei muss das Plug-In selbst definieren, wie genau die einzelnen Dokumentblöcke heißen und welchen Inhalt sie in sich tragen.

Listing 10: Beispiel eines DocumentPlugins für Datenbank-Tabellen

```
1  public class SqlPlugin : DocumentPlugin {
2    public override Document ParseInput(object input) {
3      DataTable tbl = (DataTable)input;
4      Document document = new Document(tbl.TableName);
5      int rowNum = 0;
6      foreach (DataRow row in tbl.Rows) {
7        DocumentBlock block = new DocumentBlock(document, "Row #" + rowNum++);
8        string[] content = new string[tbl.Columns.Count];
9        int colNum = 0;
10       foreach (DataColumn column in tbl.Columns)
11         content[colNum++] = row[column].ToString();
12       block.Content = String.Join("|", content);
13       document.AddBlock(block);
14     }
15     return document;
16   }
17   public override bool CanHandle(SourceType type) {
18     return type.Equals("DataTable");
```

Das Ergebnis des Plug-Ins aus Listing 10 ist dann ein Dokument mit dem Namen der Tabelle, welches für jeden Datensatz einen Dokumentenblock mit der Zahl der aktuellen Zeile (Zeile 7) und dessen gesamten Inhalt in Form einer kommaseparierten Liste (Zeile 12) hat.

Auch hier ist es wichtig, dass die Plug-Ins über die genaue Struktur der eingehenden Objekte Kenntnis haben, um sie von der universellen Basisklasse `object` in die für die Verarbeitung benötigte Klasse zu überführen (s. Zeile 3).

4.4.3 Zusammenstellen der Komposition

Nachdem alle Eingangsdokumente überführt wurden, muss aus diesen Blöcken eine Komposition erstellt werden. Dafür sind die in Unterunterabschnitt 4.2.2 erläuterten Kompositions-Plug-Ins zuständig. Sie enthalten unter anderem die Logik zum Laden von Dokumenten und deren Blöcken, um diese für die Komposition vorzubereiten.

Beispiel: Word-Komposition Um die vorherigen Beispiele weiterzuführen, könnte ein Kompositions-Plug-In, das die Dokumente des Word-Plug-Ins aus Listing 9 auf einfache Art und Weise in einem einzelnen Ergebnisdokument zusammenfasst, ähnlich wie in Listing 11 aussehen.

Listing 11: Laden einer Word-Komposition

```
1 public class WordCompositionPlugin : CompositionPlugin {
2   public override void Load(params Document[] documents) {
3     Blocks.Clear();
4     ResultBlock root = new ResultBlock(BlockType.FILE, "Root file");
5     foreach (Document document in documents) {
6       ResultBlock block = new ResultBlock(BlockType.FILE, document.Name);
7       root.ChildBlocks.Add(block);
8     }
9     Blocks.Add(root);
```

Dabei findet, wie auch bereits bei der in Unterunterabschnitt 4.4.2 beschriebenen Überführung der Eingangsdaten in die interne Datenstruktur eine weitere Überführung statt, diesmal jedoch von `Document`- und `DocumentBlock`-Objekte zu `ResultBlock`-Objekten eines bestimmtem Typs.

Da das entsprechende Dokumenten-Plug-In aus jedem einzelnen Word-Dokument ein an sich leeres `Document` gemacht hat, ist auch die Überführung zu einem `ResultBlock` sehr simpel: Jedes eingehende `Document` wird direkt zu einem `ResultBlock` des Typs `BLOCK` überführt. Diese Blöcke werden einem Wurzelblock als Kinder hinzugefügt. Dieser Wurzelblock spiegelt das im Endeffekt erzeugte Kompositionsergebnis, also eine einzelne Datei, deren Inhalte alle ausgewählten Eingangsdokumente enthält, wider.

Dies zeigt auf einfache Weise, wie aus einem physischen Dokument durch die Komposition ein abstraktes Dokument und schlussendlich nur ein einzelner Inhaltsteil eines bis dahin nicht existenten Dokumentes wird.

Dieser Wurzelblock wird dann der Liste von Blöcken des Plug-Ins selbst hinzugefügt. Nun kann die Komposition vom aufrufenden Programm durch einen Aufruf der `Execute()`-Methode ausgeführt werden.

Beispiel: Tabellen-Komposition Prinzipiell ist es auch möglich, komplexere Strukturen als Ergebnis der Komposition zu erstellen. Auch wenn das System vor dem Hintergrund entwickelt wurde, ein einzelnes Dokument zu erzeugen, lassen sich über die `BlockType`-Eigenschaft der `ResultBlock`-Objekte auch andere Strukturen erzeugen.

Als Beispiel hierfür soll die oben gezeigte und überführte Dokumentenstruktur aus einer Datenbanktabelle dienen. Ein Kompositions-Plug-In für diese Struktur ist in der Lage, für jede Tabelle ein eigenes Verzeichnis anzulegen und darin für alle 100 Datensätze jeweils eine weitere, einzelne Datei. Die dafür benötigte `Load()`-Methode ist in Listing 12 beschrieben.

Listing 12: Erzeugen einer komplexen Komposition mit Tabellendaten

```
1  public class SqlCompositionPlugin : CompositionPlugin {
2    public override void Load(params Document[] documents) {
3      foreach (Document document in documents) {
4        ResultBlock block = new ResultBlock(BlockType.FOLDER, document.Name);
5        int i = 0;
6        ResultBlock currentFile = new ResultBlock(BlockType.FILE, "0");
7        foreach (DocumentBlock docBlock in document.Blocks) {
8          ResultBlock content = new ResultBlock(BlockType.BLOCK, docBlock);
9          currentFile.ChildBlocks.Add(content);
10
11         if (++i % 100 == 0) {
12           currentFile = new ResultBlock(BlockType.FILE, i.ToString());
13           block.ChildBlocks.Add(currentFile);
```

Um dies zu erreichen, wird zunächst für jedes Dokument ein neuer `ResultBlock` des `FOLDER`-Typs erstellt. Darin sollen sich dann mehrere einzelne Dateien befinden, die über den Typ `FILE` erstellt werden.

Inhalt dieser Dateien sollen dann jeweils 100 Datensätze aus dem ursprünglichen Dokument sein, in welchem ja jeder Datensatz als einzelner `DocumentBlock` gespeichert ist. Um dies zu bewerkstelligen, wird die Anzahl der bereits verarbeiteten `DocumentBlock`-Objekte in einer Variable gespeichert, die mit jedem Eintrag inkrementiert wird (Zeile 11).

Durch den Modulo-Operator kann dann festgestellt werden, ob diese Anzahl ein Vielfaches von 100 ist. Sollte dies der Fall sein, wird der aktuell beschriebene `ResultBlock` der Datei durch einen neuen überschrieben, welcher dann die nächsten 100 Datensätze enthalten soll. Das Ergebnis ist in Abbildung 9 dargestellt.

4.4.4 Ausführen der Komposition

Das letztendliche Ausführen und Speichern der Komposition geschieht durch die in Unterunterabschnitt 4.2.2 beschriebenen `ExecuteComposition`-Methode. Diese arbeitet mit den in der `Blocks`-Eigenschaft des Plug-Ins gespeicherten `ResultBlock`-Elementen und verwendet deren Verschachtelung und `BlockType`-Eigenschaft als *Anleitung* dafür, wie das Ergebnis der Komposition aussehen soll.

Hierfür müssen einige zusätzliche Parameter berücksichtigt werden, die das Ergebnis der Komposition nicht beeinflussen sollen, aber die Ausführung derselbigen erst ermöglichen.

Beispiel: Word-Komposition Beim Aufruf der `Execute()`-Methode in Listing 13 werden alle Blöcke, die im `Blocks`-Feld des Plug-Ins gespeichert sind, verarbeitet. Für jeden Block des `FILE`-Typs wird eine neue Word-Datei angelegt[18]. Die Kindblöcke dieses `FILE`-Blocks werden dann als Quelle des eigentlichen Inhalts des Dokuments herangezogen.

[18] Die Operationen der Word-Interop-Bibliotheken sind sehr umfangreich und platzraubend, daher werden sie hier nicht explizit ausgeschrieben.

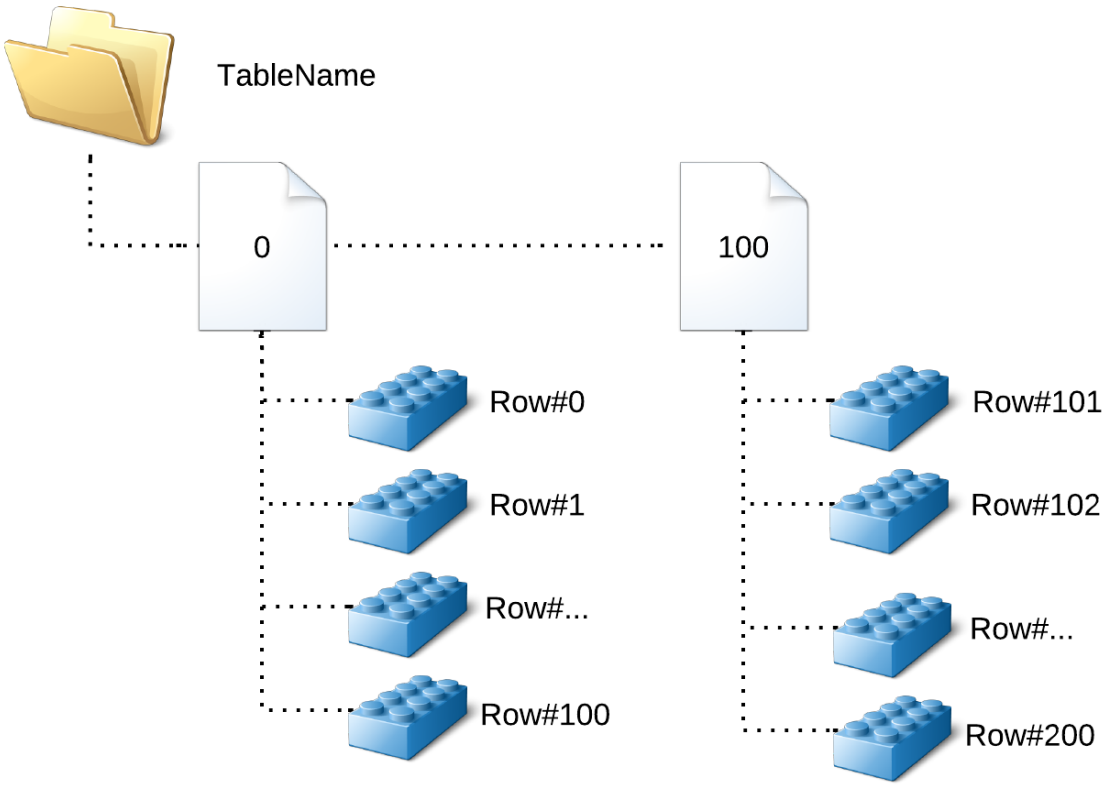


Abbildung 9: Ergebnis der beispielhaften Tabellen-Komposition

Dazu wird auch durch sie einzeln iteriert. Da jeder dieser Blöcke ein einzelnes Word-Dokument darstellt, inklusive dessen gesamtem Inhalt, wird zunächst das entsprechende Dokument anhand seines Namens (der gleich dem Blocknamen ist) geöffnet.

Listing 13: Ausführen der Word-Komposition

```
10 public override void ExecuteComposition(params string[] args) {
11   foreach (ResultBlock block in Blocks) {
12     if (block.Type != BlockType.FILE)
13       continue;
14     Word.Document doc = // Create new word document for the result
15     foreach (ResultBlock child in block.ChildBlocks) {
16       if (child.Type != BlockType.BLOCK)
17         continue;
18       Word.Document srcDoc = // Open the referenced word document
19       object srcDocStart = srcDoc.Content.Start;
20       object srcDocEnd = srcDoc.Content.End;
21       object targetDocStart = doc.Content.End - 1;
22       object targetDocEnd = doc.Content.End;
23       srcDoc.Range(ref srcDocStart, ref srcDocEnd).Copy();
24       Word.Range content = doc.Range(ref targetDocStart, ref targetDocEnd);
25       content.Paste();
26     }
27     // Save and close the word document as the block's name
```

Danach wird über die Eigenschaften `Content.Start` und `Content.End` der beiden offenen Word-Dokumente (das Quelldokument und das Zieldokument) ein Kopiervorgang eingeleitet (ab Zeile 19), der den kompletten Inhalt des Quelldokumentes in das Zieldokument kopiert. Dies ähnelt sehr stark dem Prozess, den man selbst auch manuell durchführen würde, indem man über `STRG-C` und `STRG-V` Inhalte zwischen zwei Dokumenten hin- und herkopiert.

Schlussendlich wird das Quelldokument geschlossen und es kann die nächste Quelldatei geöffnet werden. Das Zieldokument bleibt so lange geöffnet, bis alle Kopiervorgänge erfolgreich durchgeführt wurden, und wird danach erst geschlossen und mit dem Namen seines ursprünglichen `ResultBlocks` gespeichert.

Beispiel: Tabellen-Komposition Für die Tabellenkomposition wird die komplexe Struktur mit Ordnern und mehreren Dateien durch den in Listing 14 gezeigten Code erzeugt. Wichtig ist hier vor allem, dass das aufrufende Programm der Komposition einen Eingangsparameter übergibt, in dem Basisverzeichnis angegeben ist. In diesen wird das Ergebnis der Komposition abgelegt (s. Zeile 18).

Listing 14: Ausführen der Tabellen-Komposition

```
14 public override void ExecuteComposition(params string[] args) {
15   foreach (ResultBlock dirBlock in Blocks) {
16     if (dirBlock.Type != BlockType.FOLDER)
17       continue;
18     DirectoryInfo dir = new DirectoryInfo(args[0] + "\\" + dirBlock.Name);
19     if (!dir.Exists)
20       dir.Create();
21     foreach (ResultBlock fileBlock in dirBlock.ChildBlocks) {
22       FileInfo file = new FileInfo(dir.FullName + "\\" + fileBlock.Name + ".txt"
           );
23
24       if (!file.Exists)
25         file.Create();
26
27       StreamWriter writer = new StreamWriter(file.FullName);
28       foreach (ResultBlock block in fileBlock.ChildBlocks)
29         writer.Write((string)block.DocumentBlock.Content);
30       writer.Close();
```

Das Ergebnis im Dateisystem ist dann identisch mit der in Abbildung 9 gezeigten abstrakten Ansicht der Komposition. Die Daten in den einzelnen Dateien selbst werden einfach nacheinander in die Dateien geschrieben. Hier ist es denkbar, dass noch weitere Formatierungen vorgenommen werden, um sie beispielsweise in eine maschinenlesbare Struktur wie CSV oder XML zu bringen. Für die Zwecke dieses Beispiels soll dieses einfache Ergebnis jedoch ausreichend sein.

5 Umsetzung der Desktop-Anwendung

5.1 Motivation

Die Desktop-Anwendung wurde als Demonstrationsmöglichkeit entworfen, um den abstrakten Prozess der Dokumentenkomposition etwas sichtbarer und klarer zu gestalten. Dabei steht prinzipiell die Visualisierung von geladenen Kompositionen im Vordergrund und weniger das tatsächliche produktive Arbeiten. Diese Anwendung soll also weniger ein echtes Werkzeug sein als vielmehr eine andere Art der Darstellung der Kompositionsprozesse.

5.2 Konzept und grundlegender Aufbau

Für die Visualisierung eines Kompositionsprozesses sind zwei Komponenten besonders wichtig: Die Eingangsdokumente und die daraus entstehende Komposition. Beide Komponenten können als Listen verstanden werden. Die Eingangsdokumente spalten sich nach Unterunterabschnitt 4.3.1 in kleinere `DocumentBlock`-Elemente auf, in welchen der tatsächliche Inhalt der Datei gespeichert ist. Dies ist eine einzelne Hierarchie-Ebene, die bei der Anzeige beachtet werden muss.

Anders verhält es sich bei den Daten der fertigen Komposition. Wie aus Unterunterabschnitt 4.3.2 ersichtlich wird, kann jedes einzelne `ResultBlock`-Objekt eine beliebige Anzahl weiterer `ResultBlock`-Elemente als Kinder enthalten, die ihrerseits wieder weitere Kinder enthalten können. Diese beliebig tiefe Verschachtelung muss bei der Umsetzung einer grafischen Oberfläche unbedingt beachtet werden und geeignet visualisiert werden.

Da sowohl die Eingangsdaten als auch die Kompositionen durch extern ladbare Plug-Ins verarbeitet werden, muss die GUI auch eine Möglichkeit zum Laden dieser kompilierten Bibliotheken besitzen. Dies ermöglicht sowohl den direkten Vergleich zwischen zwei verschiedenen Kompositions-Plug-Ins als auch das direkte Einbinden vieler verschiedenener Typen von Eingangsdokumenten und die Evaluation, wie eine Komposition dieser Daten aussehen wird.

Schlussendlich sollte auch das Ausführen einer Komposition durch die Anwendung möglich sein. Da die `ExecuteComposition()`-Methode, wie in Unterunterabschnitt 4.2.2 ersichtlich, durch bestimmte Eingabeparameter beeinflusst werden kann, sind hier durchaus Einschränkungen in der Durchführung der Komposition möglich, da diese Parameter durch den Benutzer manuell eingegeben werden müssen, um dem Kompositions-Plug-In alle nötigen Informationen zukommen lassen zu können.

5.3 Verwendete Technologien

Die Desktop-Anwendung wurde, wie auch das Framework und die Plug-Ins selbst, in C# und .NET entwickelt. Diese Entscheidung fiel aufgrund der bereits in diesen Sprachen vorhandenen Code-Basis, die in der Desktop-Anwendung weiterverwendet werden sollte. Für die Umsetzung der Oberfläche wurde WPF[19] verwendet.

Das WPF-Framework stellt ein besonderes Programmiermodell für die Entwicklung von Windowsanwendungen dar, das tief in das .NET-Framework integriert ist und es erlaubt, aufwändige Windows-Anwendungen zu programmieren.

Um dem entwickelten System auch ein *Gesicht* zu geben, wurde auf dieser Bibliothek aufbauend eine GUI entwickelt, in welcher sich das Framework verwalten und auch mit entsprechenden Plug-Ins ausführen lässt. Sie wurde mit modernen Design-Pattern entwickelt und bietet vielfältige Möglichkeiten, die in dieser Arbeit allerdings nicht ausgeschöpft werden können.

Vielmehr wird die Bibliothek in dem kleinen Teil der GUI-Programmierung verwendet, um einen einfachen Überblick über die Funktionsweise des Systems zu liefern, den eine reine codebasierte Dokumentation nicht ermöglichen kann.

Eine sehr zentrale Komponente von WPF-Anwendungen ist die Layout-Sprache, die das Aussehen von Fenstern und Objekten definiert: XAML[20]. Aus den im XAML deklarierten XML-Elementen werden zur Laufzeit .NET-Objekte erzeugt, durch welche die Oberfläche entsteht[Hub08, S. 57].

Die Vorzüge dieses Ansatzes im Vergleich zu einer rein programmierten Oberfläche, wie es in Java mit *Swing* oder AWT[21] geschieht, liegen unter anderem in der besseren Trennung von Verhaltenslogik und Layout der Applikation und in der damit verbundenen einfacheren Verwaltung der beiden Teile eines Programms.

Außerdem können so bereits vor dem Kompilieren des Programms die Oberfläche angesehen und somit grobe Probleme in der Darstellung erkannt oder schlecht platzierte Objekte verändert werden. Würde die GUI ausschließlich programmiert werden, könnte dies stets nur beim Ausführen des Programms getan werden.

5.4 Oberflächen-Konzeption

5.4.1 Hauptfenster

Das Hauptfenster ist das erste Fenster, welches beim Ausführen der Desktop-Anwendung geöffnet wird. Darin werden die Eingabedaten, die Komposition und alle weiteren Fenster verwaltet. Der Aufbau ist dabei darauf ausgelegt, den Fokus des Benutzers auf die beiden wesentlichen Elemente des für ihn wichtigsten Prozesses zu legen; die Zusammenstellung einer Komposition aus mehreren Eingangsdokumenten.

[19] Windows Presentation Foundation
[20] Extensible Application Markup Language
[21] Abstract Window Toolkit

Abbildung 10 zeigt die Ansicht des Hauptfensters, die sich dem Benutzer unmittelbar nach dem Ausführen des Programms darstellt. Dabei sind noch keine besonderen Einstellungen vorgenommen worden, sodass keines der darin enthaltenen Felder mit Daten gefüllt ist. Dies geschieht erst durch das Hinzufügen von Eingabedaten.

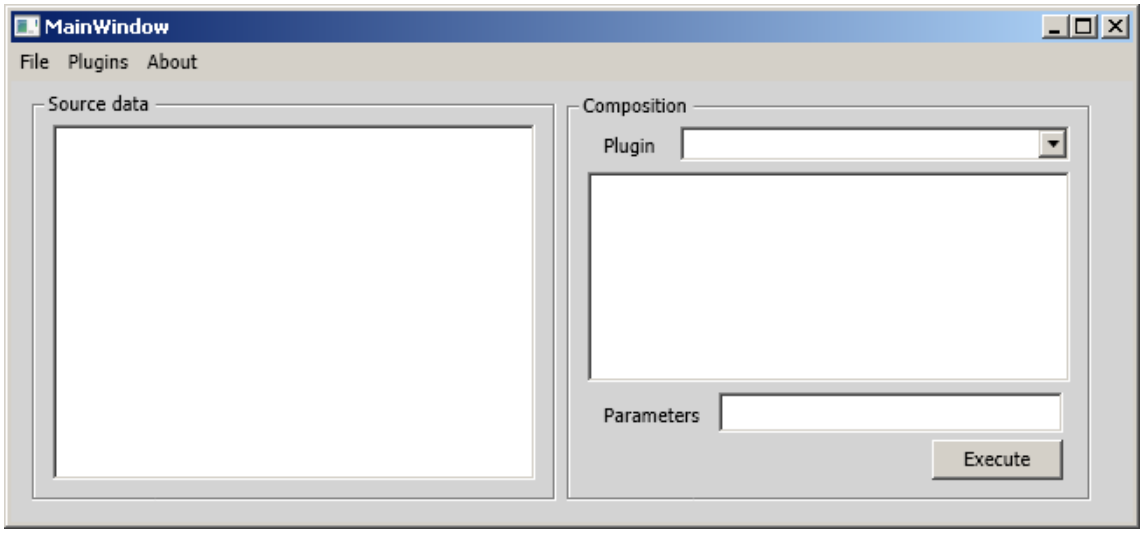


Abbildung 10: Das Hauptfenster der Desktop-Anwendung

Diese werden in der linken Hälfte des Fensters aufgelistet, nachdem sie durch die einzelnen Dokumenten-Plug-Ins verarbeitet und in die interne Datenstruktur überführt wurden. Hier werden auch sofort die Inhaltsblöcke der einzelnen Dokumente angezeigt, wie sie von dem jeweiligen Plug-In überführt wurden.

Die rechte Seite des Fensters ist für die Verwaltung der Komposition selbst zuständig. So muss zunächst ein passendes Kompositions-Plug-In ausgewählt werden. Dieses kann dann die Eingangsdaten laden (s. Unterunterabschnitt 4.2.2, *Die „Load“-Methode*) und das Ergebnis in dem darunter befindlichen Fenster als Vorschau anzeigen.

Da zum Ausführen einer Komposition, wie in Unterunterabschnitt 4.2.2 – *Die „ExecuteComposition“-Methode* erklärt, meistens zusätzliche Parameter von außen benötigt werden, können diese in einem dafür vorgesehenen Textfeld eingegeben werden, bevor die Komposition durch einen Klick ausgeführt wird.

5.4.2 Plug-In-Verwaltung

Um jedoch überhaupt eine Komposition starten oder Eingangsdaten eingeben zu können, müssen zunächst die dafür benötigten Plug-Ins in das Framework geladen werden. Dazu wurde ein Verwaltungsfenster erstellt, welches über den entsprechenden Menüpunkt *Plugins - Configure plugins* zu erreichen ist.

Hier werden in zwei separaten Listen jeweils die derzeit geladenen Dokumenten- und Kompositions-Plug-Ins mit ihren jeweiligen Assembly-Namen angezeigt. Beim Hinzufügen einer Assembly, die ein oder mehrere Plug-In-Klassen enthalten, werden diese durch das Framework hinzugefügt (vgl. Listing 3) und anschließend in der Plug-In-Verwaltung angezeigt.

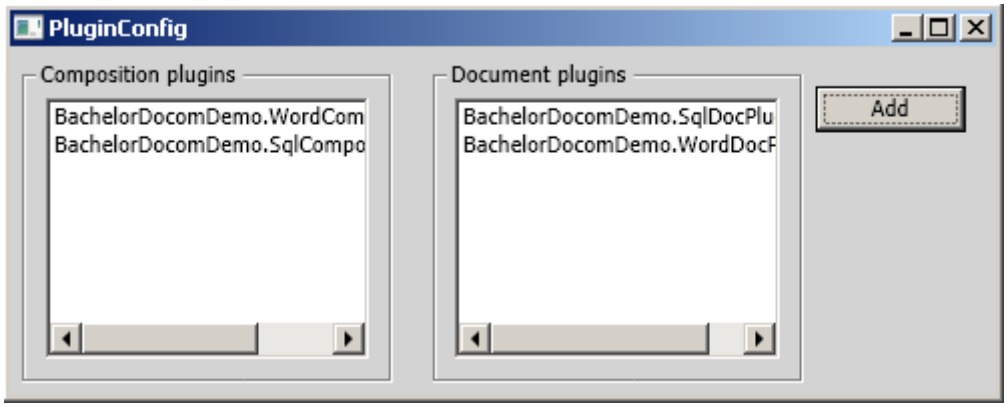


Abbildung 11: Plug-In-Verwaltung mit geladenen Plug-In-Assemblys

Die in Abbildung 11 angezeigten Plug-Ins sind die Umsetzungen der Beispiele, welche in Unterunterabschnitt 4.4.1 als Beispielanwendungen des Frameworks und der Plug-In-Klassen dienen. Diese Plug-Ins werden auch hier zur Darstellung der Abläufe in der Desktop-Anwendung verwendet.

5.4.3 Eingeben von Eingangsdaten

Jede Komposition arbeitet mit den internen `Document`-Objekten. Um diese von den geladenen Plug-Ins erzeugen zu lassen, müssen diese entsprechend der Definition der `Load()`-Methode (s. Unterunterabschnitt 4.2.2) mit Daten versorgt werden. Dies soll im Allgemeinen durch das aufrufende Programm passieren, in diesem Fall also durch die Desktop-Anwendung.

Um mit den beiden bereits vorgestellten Beispielen arbeiten zu können, werden zwei grundlegende Datentypen als Eingangsdaten benötigt: physikalisch vorhandene Dokumente und Datenbank-Tabellen. Dafür wurden zwei verschiedene Menüpunkte für das Hinzufügen von neuen Eingangsdaten erstellt, die in Abbildung 12 dargestellt sind.

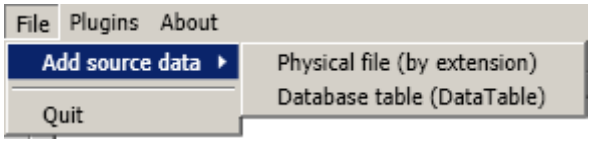


Abbildung 12: Menüpunkte zum Hinzufügen von Eingangsdaten

Der erste Menüpunkt, *Physical file (by extension)*, öffnet den *Datei öffnen*-Dialog von Windows. In ihm kann der Nutzer dann eine beliebige Datei auswählen und diese als Eingangsdokument hinzufügen. Dabei ist jedoch wichtig, dass ein Plug-In geladen wurde, welches mit dem Dateityp umgehen kann. Diese Art des Hinzufügens ist nicht nur für das Beispiel-Word-Plug-In nützlich, sondern kann für jede beliebige Komposition verwendet werden, die mit auf dem Arbeitsrechner gespeicherten Dateien umgehen kann.

Anders ist dies bei dem Menüpunkt *Database table (DataTable)*. Dieser wurde explizit für das Beispiel der Tabellen-Komposition eingeführt und öffnet einen weiteren Dialog, der die Auswahl einer bestimmten Datenbank-Tabelle ermöglicht und in Abbildung 13 dargestellt ist.

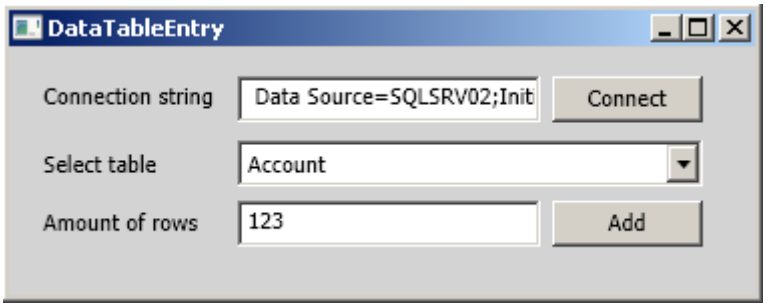


Abbildung 13: Hinzufügen einer Datenbank-Tabelle als Eingangsdatenquelle

In diesem Dialog muss zunächst ein sogenannter *Connection String* eingegeben werden. Diese besondere Art einer Zeichenkette wird von .NET zur Definition einer Datenbank-Verbindung verwendet[Micc]. Er ist in einer sehr einfachen Syntax definiert, die nur aus einer Reihe von Schlüsselwörtern und Werten besteht, die durch Semikola getrennt werden.

So definiert beispielsweise der Connection String aus Listing 15 eine Verbindung zur Datenbank *MyDB* auf dem Server *SQLSRV02* (der durch DNS[22] aufgelöst wird) mit dem Benutzernamen *root* und dem Password *xyz123ABC*.

Listing 15: Beispielhafter Connection String

```
1 string ConnectionString = "Data Source=SQLSRV02;Initial Catalog=MyDB;Persist
    Security Info=True;User ID=root;Password=xyz123ABC"
```

Durch die Eingabe dieser Zeichenkette kann dann die Verbindung anhand der darin definierten Parameter aufgebaut werden. Nun werden in der Auswahlliste *Table* alle in dieser Datenbank enthaltenen Tabellen aufgelistet. Der Benutzer muss daraus eine auswählen, die als Eingangsdatenquelle dienen soll. Zudem muss die Menge der auszuwählenden Datensätze angegeben werden, die als Daten herangezogen werden sollen.

Das Dokumenten-Plug-In, welches `DataTable`-Objekte verarbeiten kann, wird dann die Tabelle inklusive der bestimmten Anzahl an Datensätzen in die interne `Document`-Struktur überführen.

[22] Domain Name Server

5.4.4 Word-Komposition

Bei der Word-Komposition sollen mehrere Eingabe-Dokumente in ein Ergebnisdokument zusammengefasst werden. Dazu müssen zunächst die beiden verantwortlichen Plug-Ins geladen werden. Danach kann durch das Menü zum Hinzufügen von Quelldaten ein neues Word-Dokument hinzugefügt werden. Das Ergebnis der Word-Komposition wird dann im Vorschau-Fenster angezeigt und kann nach Eingabe der erforderlichen Parameter ausgeführt werden.

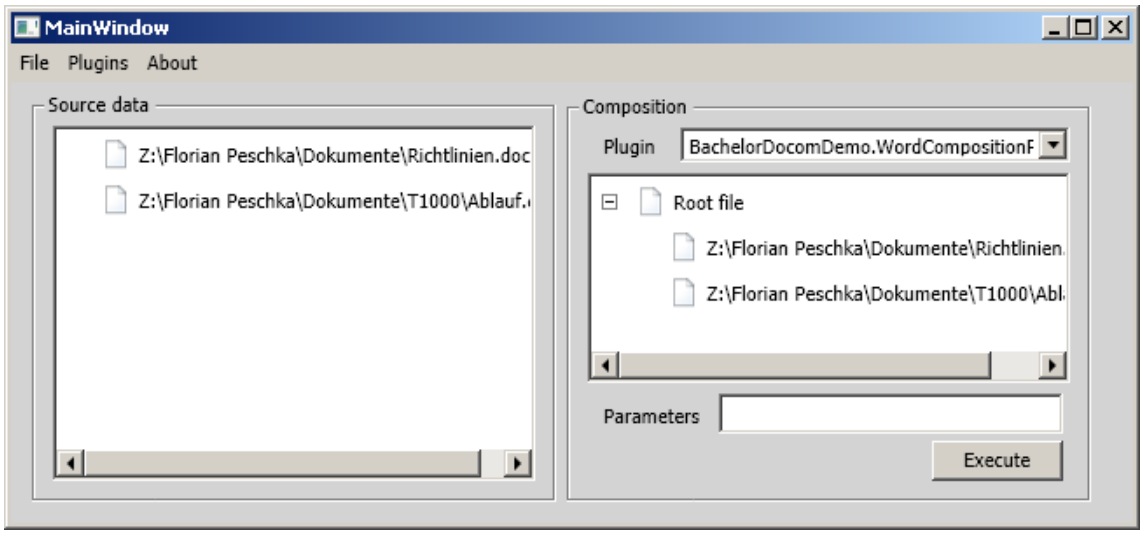


Abbildung 14: Einfache Word-Komposition durch die Beispiel-Plug-Ins

Wie in Abbildung 14 gut zu erkennen ist, wird für die Eingabedokumente keine weitere Verschachtelung in Form von `DocumentBlock`-Objekten benötigt. Jedes Dokument wird in seiner Gesamtheit in das Ergebnisdokument kopiert. Das Ergebnisdokument ist im *Composition*-Feld zu erkennen. Dieses Dokument beinhaltet nach der Komposition alle Eingangsdokumente in der Reihenfolge, in welcher sie hinzugefügt wurden.

5.4.5 Tabellen-Komposition

Für die Tabellen-Komposition wird das in Abbildung 13 dargestellte Fenster verwendet. Ein mögliches Ergebnis dieser Eingangsdaten ist in Abbildung 15 dargestellt[23]. In diesem Beispiel wird deutlich, dass die Eingangsdaten und das Ergebnis der Komposition nicht unbedingt gleich strukturiert sein müssen. Es wäre ebenso denkbar, die einzelnen Tabellenzeilen untereinander zu vermischen.

Auch könnten noch weitere Verschachtelungsebenen eingeführt werden, die z. B. die Eingabedaten auf Datenbanken ausweiten, aus denen das entsprechende Plug-In jede Tabelle einzeln herauszieht. Diese könnten in der Komposition dann ebenfalls in Unterordnern verwaltet werden.

[23] Aufgrund des Datenschutzes mussten die Inhalte der Tabelle geschwärzt werden.

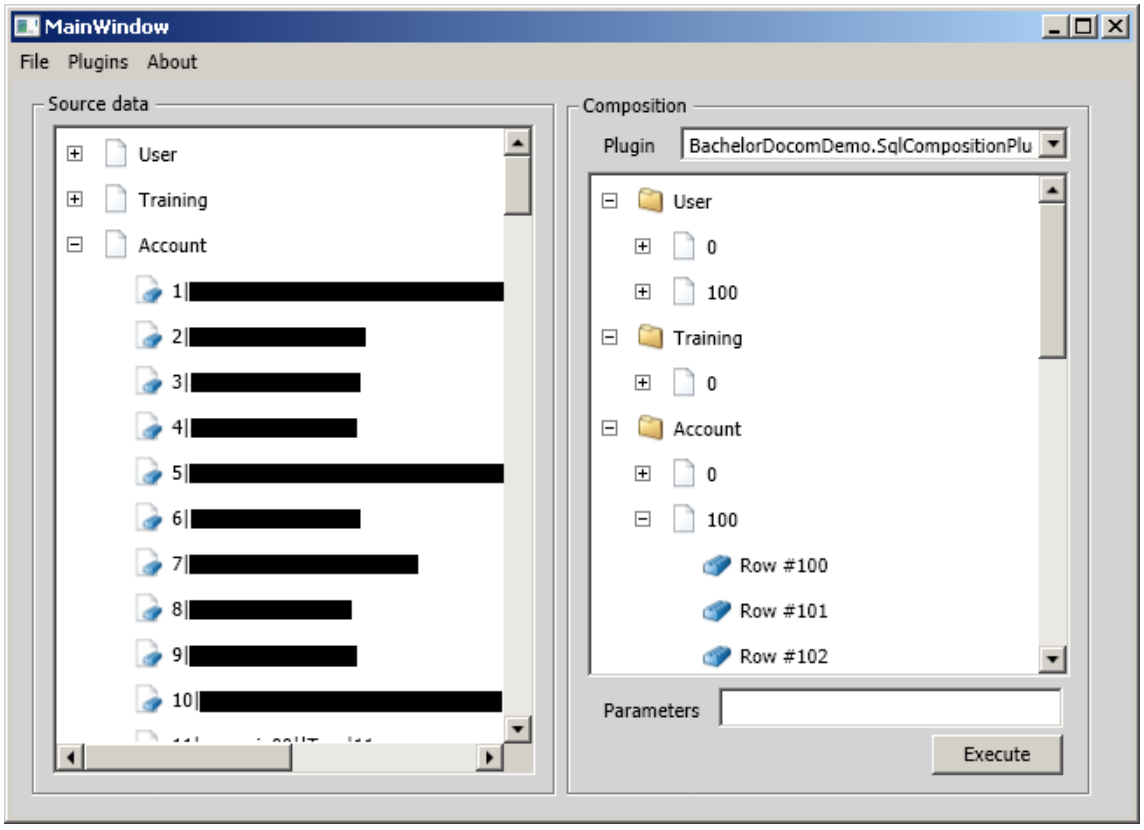


Abbildung 15: Einfache Tabellen-Komposition durch die Beispiel-Plug-Ins

5.5 Programmierung der Desktop-Anwendung

Für die Programmierung der vorgestellten Benutzeroberfläche kommt, wie in Unterabschnitt 5.3 bereits erläutert, WPF und XAML zur Verwendung. Ein Grundprinzip, was bei der Programmierung der Desktop-Anwendung im Vordergrund stand, war die deutliche Trennung von Code und Layout, wie sie im klassischen MVC[24] angestrebt wird.

Bei der Verwendung von WPF wird jedoch ein abgewandeltes Pattern verwendet, welches „eine bessere Trennung von UI und Logik“[Hub08, S. 145] bietet und besonders für XAML-Dateien interessant ist: MVVM[25].

5.5.1 MVVM-Pattern und DataBinding

Die Grundlage für die Umsetzung des GUI-Konzeptes bilden hierbei die von Microsoft entwickelten Technologien des MVVM-Pattern und DataBinding. Bei ersterem handelt es sich um ein abgewandeltes Konzept, das auf dem klassischen MVC basiert. Es wurde vor dem Hintergrund entwickelt, dass die Oberfläche moderner Programme zunehmend von Designern gestaltet werden, im Gegensatz zur Programmierung durch Softwareentwickler.

Die *View* besteht aus den verschiedenen Elementen der Oberfläche, seien dies Buttons, Felder oder ganze Fenster. Diese sind zumeist in XAML ausgezeichnet[26] und werden von Designern in „What you see is what you get“-Editoren (WYSIWYG[27]) erstellt. Das *Model* beinhaltet alle Daten, unabhängig von deren Repräsentation in der *View*, und verwaltet die Konsistenz dieser Daten sowie den generellen Zustand des Programms.

Der Controller ist in MVVM etwas in den Hintergrund geraten. Die klassische *Programmlogik* ist bei WPF-Applikationen direkt in den View-Komponenten untergebracht. So sind Methoden, die auf Aktionen des Benutzers reagieren und Zustandsänderungen hervorrufen, in den gleichen Klassen untergebracht, die auch die Oberfläche selbst definieren.

Besonders interessant ist das *ViewModel*, welches in Abbildung 16 dargestellt ist. Dieses stellt eine weitere Schicht zwischen dem *Model* und der *View* dar, das für die Transformation der Daten des Models in eine von der View darstellbare Form zuständig. Nötig wird dies besonders dann, wenn das *Model* durch die View nur schwer oder garnicht dargestellt werden kann, da es z. B. durch eine vorherige Analyse erstellt wurde, die nicht auf die später erstellte Layout-Konzeption passt.

[24] Model View Controller
[25] Model View ViewModel
[26] *Programmiert* wäre hier irreführend. XAML ist eine Auszeichnungssprache, besitzt also nur wenige Eigenschaften einer klassischen Programmiersprache. Daher sind XAML-Elemente ausgezeichnet und nicht programmiert.
[27] What you see is what you get

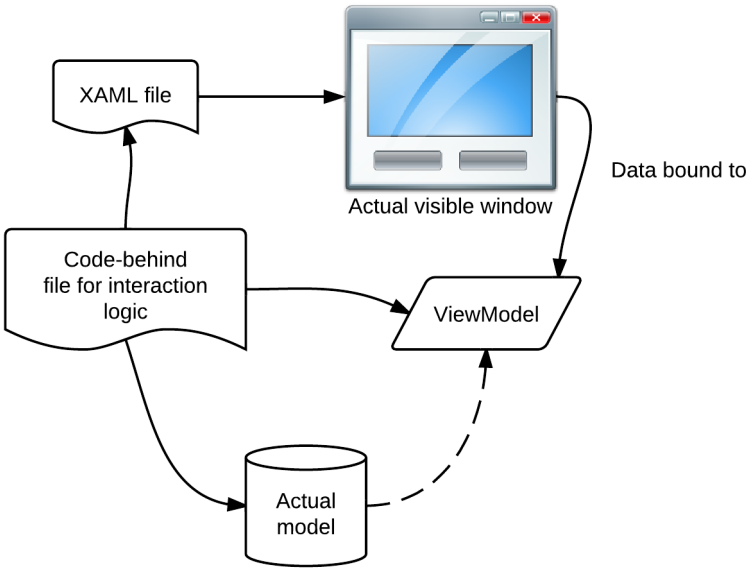


Abbildung 16: Das MVVM-Pattern in vereinfachter Darstellung

So gesehen ist das *ViewModel* eine spezialisierte Sicht auf das *Model* – daher der Name – die es der View erst ermöglicht, die im *Model* enthaltenen Daten darzustellen. Zudem ist durch *Data-Binding* ein direktes Manipulieren dieser Daten möglich. Darunter versteht man die Möglichkeit, Komponenten der View direkt an bestimmte Felder oder Eigenschaften von Klassen des Models zu binden. Dadurch entfällt das sonst nötige manuelle Programmieren von Methoden, die die Inhalte solcher Felder in die View bringen.

Um dem Programm das für DataBinding nötige *ViewModel* zuzuweisen, wird der sogenannte `DataContext` des jeweiligen Fensters, in diesem Fall dem Hauptfenster der Desktop-Anwendung, auf das Objekt des ViewModels gesetzt. Dieser Vorgang ist in Listing 16 dargestellt und wird direkt im Konstruktor der Hauptfenster-Klasse durchgeführt.

Listing 16: Setzen des DataContexts auf das ViewModel

```
1 public MainWindow(Framework framework, ViewModel model){
2   docom = framework;
3   this.model = model;
4   DataContext = model;
5 }
```

Nicht nur das Anzeigen ist über DataBinding möglich, sondern auch das Bearbeiten und sogar Löschen von Elementen. So entfallen auch diese Methoden, die in anderen Systemen einen deutlichen Aufwand darstellen. Durch DataBinding kann eine Liste in der View direkt auf ein Feld im *ViewModel* gebunden werden. Sobald sich einer der beiden durch Aktionen des Benutzers oder des Programms verändert, können die beiden Elemente die jeweils andere Seite synchronisieren und den neuen Zustand anzeigen.

Damit erleichtert DataBinding in Verbindung mit MVVM die Programmierung von GUI-Oberflächen enorm, da nahezu alle verwaltungstechnischen Aufgaben durch das zugrunde liegende Framework erledigt werden und nicht mehr manuell durch den Programmierer geschrieben werden müssen.

5.5.2 Hauptfenster

Jedes Fenster in WPF beginnt mit einer ähnlichen Definition. Da XAML durch XML beeinflusst ist, muss jede XAML-Datei ein einzelnes Wurzel-Element besitzen. Im Falle eines Fensters ist dieses Element `<Window>`. In Listing 17 sind die Eigenschaften des Wurzel-Elements des Hauptfensters der Applikation aufgelistet.

Listing 17: Wurzel-Element des Hauptfensters

```
1 <Window x:Class="DocomGUI.View.MainWindow"
2   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3   [...]
4   MaxWidth="650" MinHeight="300" Height="300" Width="650"
5   Title="MainWindow" Background="LightGray" mc:Ignorable="d">
```

Auf die Definition diverser Namensräume, die für weitere Funktionen innerhalb der Datei nötig sind, folgt die Angabe verschiedener Werte, die die Ausmaße des Fensters definieren; `MaxWidth`, `MinWidth` und `Width` geben die Grenzen und den Startwert der Breite des Fensters an. Dasselbe könnte auch für die Höhe des Fensters angegeben werden. Durch `Title` wird der Titel des Fensters, der oberhalb der Menüleiste angezeigt wird, festgelegt. Schlussendlich erhält das Fenster durch `Background` eine bestimmte Hintergrundfarbe aus einer Liste, die für viele häufig gebrauchte Farben innerhalb des WPF-Frameworks enhält[28].

Die Menü-Leiste Darauf folgt die Definition des sichtbaren Bereichs des Fensters. Zunächst erfolgt die relativ simple Definition eines Menüs, welches sich über die gesamte Breite des Fensters legt und die in Unterabschnitt 5.4 aufgeführten Funktionen abdeckt.

Listing 18: Definition des Hauptmenüs für die Desktop-Anwendung

```
1 <Menu Height="21" Name="MainMenu" VerticalAlignment="Top">
2   <MenuItem Header="File" Name="FileMenuItem">
3     <MenuItem Header="Add source data">
4       <MenuItem Header="Physical file (by extension)"
5         Click="AddDocumentMenuItem_Click" />
```

[28] Um die weiteren Code-Ausschnitte aus den XAML-Dateien klein zu halten, werden Styling-Angaben nicht weiter aufgeführt, da diese zur Funktion des Programms keinen nennenswerten Beitrag leisten

An diesem Menü wird bereits eine der wichtigsten Funktionalitäten jeder grafischen Oberfläche sichtbar: Die Reaktion auf Aktionen des Benutzers. Zu diesem Zweck wird z. B. in Listing 18, Zeile 5 ein sogenannter *EventHandler* angegeben. Diese EventHandler werden in der Datei definiert, die als Controller für dieses Fenster gilt. Als Parameter für die XAML-Datei werden stets die Namen der entsprechenden Methoden verwendet.

Diese Methode wird dann durch das Framework jedes Mal aufgerufen, wenn der Benutzer mit dem Element auf die durch den Parameternamen definierte Weise interagiert. Die in diesem Menü definierten Click-EventHandler sind allesamt sehr einfach, da sie lediglich andere Fenster öffnen oder das Programm schließen.

Ansicht der Eingangsdaten Die Komponente, welche die Darstellung der Eingangsdaten verwaltet, ist ein sogenanntes `TreeView`-Element. Da mehrere Dokumente jeweils mit ihren Dokumentenblöcken als Eingangsdaten dienen, ist eine Ansicht speziell für Baumstrukturen dafür besonders gut geeignet. Dabei kommt auch die in Unterabschnitt 5.5, Absatz *MVVM-Pattern und DataBinding* beschriebene DataBinding-Technik zur Anwendung.

Listing 19: TreeView-Ansicht der Eingangsdaten

```
1  <HierarchicalDataTemplate DataType="{x:Type docom:Document}" ItemsSource="{
       Binding Blocks}">
2    <StackPanel>
3      <Image Source="/DocomGUI;component/Resources/document_plain.png" />
4      <TextBlock Text="{Binding Name}" />
5    </StackPanel>
6  </HierarchicalDataTemplate>
7  <HierarchicalDataTemplate DataType="{x:Type docom:DocumentBlock}">
8    <StackPanel>
9      <Image Source="/DocomGUI;component/Resources/document_plain_building_block.
           png" />
10     <TextBlock Text="{Binding Content}" />
11   </StackPanel>
12 </HierarchicalDataTemplate>
```

Sichtbar wird dies in Listing 19. Hier werden zwei sogenannte `HierarchicalDataTemplate`s definiert. Ersteres ist ein Template für die Darstellung von Objekten des `Document`-Typs. Templates sind einzelne Entitäten, die für sich genommen keine Darstellung haben, sondern erst durch das Befüllen mit einem Objekt die Daten dieses Objektes auf die in ihnen definierte Art und Weise darstellen.

Im Falle eines `Document`-Objekts wird das in Zeile 1 definierte Template verwendet. Es zeigt das Dokument als Icon an, neben welchem der Name des Dokuments steht. Da der Name eine Eigenschaft des Objektes ist, welches dem Template übergeben wird, muss dies durch DataBinding geschehen. So wird die Eigenschaft `Text` des `TextBlock`-Elements an die Eigenschaft `Name` des zugrunde liegenden Objektes gebunden.

Eine Besonderheit der `HierarchicalDataTemplate`-Elemente ist die Eigenschaft `ItemsSource`. Diese ermöglicht das Angeben einer Eigenschaft des gebundenen Objektes, die als Quelle für Kindelemente zur Anzeige dient. Da jedes `Document` aus vielen einzelnen Dokumentblöcken besteht, die in der Eigenschaft `Blocks` gespeichert sind, wird dies in Zeile 1 auch dementsprechend angegeben.

Die Dokumentblöcke sollen auch in einer besonderen Art und Weise dargestellt werden. Dafür ist ein zweites Template definiert, das für den `DocumentBlock`-Typ zuständig ist. Die Darstellung ist ähnlich der des Templates für Dokumente, abgesehen von einem anderen Icon und einer anderen Eigenschaft für die Anzeige des Namens.

Das `TreeView`-Element entscheidet anhand dieser beiden Templates und den darin definierten `DataType`-Parametern, wie ein bestimmtes Objekt angezeigt werden soll. Die Liste aller Eingangsdokumente wird dem TreeView-Element über seinen eigenen `ItemsSource`-Parameter angegeben. Dieser ist wiederum im *ViewModel* des Programms gespeichert, welcher später von den Operationen zum Eingeben von Eingangsdaten modifiziert wird (s. Listing 26).

Hier wird der große Vorteil der Verwendung von DataBinding sichtbar. Es muss im TreeView-Element keinerlei weitere Logik in der Klasse definiert werden, wann und wie auf veränderte Daten reagiert werden soll. Das Einzige, was durch den Code ausgeführt werden muss, ist das gelegentliche Aktualisieren des ViewModels, beziehungsweise das Senden der Benachrichtigung über die Veränderung einer Eigenschaft, wie in Listing 26, Zeile 12 gezeigt.

Kompositionsvorschau Das gleiche Prinzip wird auch beim `TreeView`-Element, welches die durch das Kompositions-Plug-In umgewandelten Ergebnisblöcke enthält, verwendet. So wird in Listing 20 das `TreeView`-Element durch DataBinding an die Ergebnisblöcke des ViewModels gebunden, welche durch die Plug-Ins generiert werden.

Listing 20: Kompositions-Bereich des Hauptfensters

```
1 <GroupBox Header="Composition">
2   <Grid>
3     <TreeView Name="CompositionTree"
4       ItemsSource="{Binding ResultBlocks}"
5       ItemTemplateSelector="{StaticResource BlockTemplateSelector}" />
6     <ComboBox Name="CompositionListBox"
7       ItemsSource="{Binding CompositionPlugins}"
8       SelectionChanged="CompositionListBox_SelectionChanged" />
```

Ebenso ist die Liste, welche alle momentan geladenen Kompositions-Plug-Ins zur Auswahl durch den Benutzer enthält, an die `CompositionPlugins`-Liste gebunden (s. Zeile 7). Diese Liste wird durch das Laden von Plug-Ins aktualisiert, wie in Unterunterabschnitt 5.5.3 erläutert. Anders als bei der Anzeige der Eingabedokumente, bei welcher die Anzeige der Blöcke je nach deren tatsächlichem Objekttyp verändert wurde – ein leeres Dokument für `Document`-Objekte und ein Dokument mit Block für `DocumentBlock`-Objekte – muss bei der Anzeige der Komposition eine andere Methode verwendet werden.

Da alle Objekte in der Liste der Komposition vom Typ `ResultBlock` sind, können anhand des Objekttyps keine unterschiedliche Anzeigeicons geladen werden. Die Unterschiede der Ergebnisblöcke liegen in ihrer `Type`-Eigenschaft. Es muss demnach anhand dieses Typs entschieden werden, wie der Ergebnisblock dargestellt werden soll. Dafür wird dem TreeView-Element ein sogenannter `ItemTemplateSelector` zugewiesen, der, wie der Name schon sagt, je nach `Item` (Objekt) ein anderes Template auswählt, das für dessen Darstellung zuständig ist.

Damit dieser *Auswähler* überhaupt arbeiten kann, müssen ihm jedoch zuerst die Templates gegeben werden, die er dann anhand eines selbst definierbaren Entscheidungskriteriums auswählt und den entsprechenden Objekten zuweist. Für die Ergebnisblöcke wurden dafür die in Listing 21 gezeigten Templates definiert.

Listing 21: Templates für die Darstellung von Ergebnisblöcken

```
1  <Window.Resources>
2    <HierarchicalDataTemplate x:Key="BlockTemplate" ItemsSource="{Binding
         ChildBlocks}">
3      <Image Source="/DocomGUI;component/Resources/building_block.png" />
4    </HierarchicalDataTemplate>
5    <HierarchicalDataTemplate x:Key="FileTemplate" ItemsSource="{Binding
         ChildBlocks}">
6      <Image Source="/DocomGUI;component/Resources/document_plain.png" />
7    </HierarchicalDataTemplate>
8    <HierarchicalDataTemplate x:Key="FolderTemplate" ItemsSource="{Binding
         ChildBlocks}">
9      <Image Source="/DocomGUI;component/Resources/folder_closed.png" />
10   </HierarchicalDataTemplate>
11   <binding:BlockTemplateSelector
12      BlockTemplate="{StaticResource BlockTemplate}"
13      FileTemplate="{StaticResource FileTemplate}"
14      FolderTemplate="{StaticResource FolderTemplate}"
15      x:Key="BlockTemplateSelector" />
16 </Window.Resources>
```

Die einzelnen Templates sind denen der Ansicht der Eingangsdaten sehr ähnlich. Wichtig ist jedoch, dass alle Templates ihre `ItemsSource`-Eigenschaft auf die `ChildBlocks`-Eigenschaft des Objektes legen. Somit ist eine beliebig tiefe Verschachtelung der Kindblöcke möglich, da für jeden Block die Kindblöcke wieder als Quelle von weiteren Objekten dienen. Die Templates werden dem Auswähler in Zeile 11 zugewiesen. Somit kann dieser auf die Templates in seiner Logik verweisen und sie als Objekte verwenden.

Das Entscheidungskriterium kann im Auswähler frei definiert werden. Da er sowohl die Templates als auch das in Frage kommende Objekt erhält, kann er komplett frei entscheiden, welches Template dem Objekt zugewiesen werden soll. Im Falle der Ergebnisblöcke ist das Entscheidungskriterium klar definiert; der `BlockType` des Blocks. Anhand dieser Eigenschaft wird eines der drei Templates angewandt, wie in Listing 22 zu sehen ist.

Listing 22: Auswählen des richtigen Templates anhand der BlockType-Eigenschaft

```
if (block.Type == BlockType.BLOCK)
    return BlockTemplate;
else if (block.Type == BlockType.FILE)
    return FileTemplate;
else
    return FolderTemplate;
```

Die Templates aus Listing 21 sind als Eigenschaft der `BlockTemplateSelector`-Klasse definiert. Diese werden durch das Framework automatisch mit Objekten der Templates gefüllt. Somit kann der Auswähler anhand einer einfachen `if`-Abfrage entscheiden, welches der Templates für ein bestimmtes `ResultBlock`-Objekt zu verwenden ist. Dieses Template wird, ebenfalls als Objekt, zurückgegeben und das Framework kann dann dieses Template mit den Daten des Objektes füllen und in der Ansicht anzeigen.

5.5.3 Plug-In-Verwaltung

Auch für die Ansicht des Plug-In-Verwaltungsfensters (s. Abbildung 11) war DataBinding ein zentrales Element. Der in Listing 23 gezeigte XAML-Code ist für die Anzeige der Listen der Plug-In-Verwaltung zuständig. Die beiden Listen sind jeweils für die Anzeige der momentan geladenen Dokumenten- und Kompositions-Plug-Ins zuständig. Dabei ist diese Anzeige jeweils an die Liste dieser Plug-Ins im *ViewModel* gebunden.

Listing 23: XAML-Code des Plug-In-Verwaltungsfensters

```
<ListBox Name="CompositionPluginListBox" ItemsSource="{Binding
    CompositionPlugins}" />
<ListBox Name="DocumentPluginListBox" ItemsSource="{Binding DocumentPlugins}" />
```

Durch das Hinzufügen eines neuen Plug-Ins über einen *Add*-Button werden die Listen verändert. Dabei ist eine Eigenschaft des ViewModels sehr wichtig, die das Aktualisieren der Listen sehr einfach gestaltet. Wie in Listing 25 zu sehen ist, wird bei einem Setzen der `CompositionPlugins`-Eigenschaft die Methode zum Aktualisieren automatisch aufgerufen. Dies bedeutet, dass in der Methode der Plug-In-Verwaltung beim Laden eines neues Plug-Ins diese Methode nicht explizit aufgerufen werden muss, wie Listing 24 zeigt.

Listing 24: Laden von neuen Plug-Ins-Assemblys

```
docom.RegisterAssembly(openPlugin.FileName);
DocomGUI.Properties.Settings.Default.Plugins.Add(openPlugin.FileName);
model.CompositionPlugins = docom.CompositionPlugins;
model.DocumentPlugins = docom.DocumentPlugins;
```

Es reicht, das *ViewModel* durch eine erneute Zuweisung der `CompositionPlugins`- und `DocumentPlugins`-Eigenschaften auf den aktuellen Stand zu bringen. Da Plug-Ins stets durch das Framework selbst geladen werden und nicht durch das aufrufende Programm, kann hier die Liste der Plug-Ins direkt vom Framework kopiert und in das *ViewModel* eingesetzt werden (s. Zeile 3 und 4). Die in Listing 25 definierte Methode wird dann automatisch für das Aktualisieren der Ansicht sorgen.

Listing 25: Aktualisieren der Plug-Ins im ViewModel

```
1 compositionPlugins = new ObservableCollection<CompositionPlugin>(value);
2 OnPropertyChanged("CompositionPlugins");
```

Der wesentliche Unterschied zu dem Vorgehen in z. B. Listing 26 ist, dass dort nicht die Eigenschaft `SourceDocuments` selbst verändert wird. Viel mehr wird eine Methode des darin gespeicherten Objektes aufgerufen. Dies verändert zwar den Zustand des Objektes, das *ViewModel* erhält davon aber nie Kenntnis. Daher muss im Nachhinein die `RaiseProperty`-Methode manuell aufgerufen werden, um die Aktualisierung zu synchronisieren.

Würde man die `SourceDocuments`-Eigenschaft wie auch bei den Plug-Ins durch eine direkte Zuweisung verändern, würde das manuelle Aufrufen wegfallen, da dies bereits durch die aufgezeigte Methode gehandhabt werden würde.

5.5.4 Eingeben von Eingangsdaten

Für die Funktion, ein neues Dokument als Eingangsdatenquelle hinzuzufügen, wurde der Event-Handler in Listing 26 geschrieben, der dem entsprechenden Menüpunkt in Listing 18, Zeile 5 zugewiesen wird.

Listing 26: Der Click-EventHandler zum Hinzufügen von Dateien

```
1  private void AddDocumentMenuItem_Click(object sender, RoutedEventArgs e) {
2    OpenFileDialog openFile = new OpenFileDialog();
3    openFile.RestoreDirectory = true;
4    bool? result = openFile.ShowDialog(this);
5    if (result == true) {
6      FileInfo info = new FileInfo(openFile.FileName);
7      try {
8        DocumentPlugin pluginHandler = docom.GetPluginFor(info.Extension);
9        Document doc = pluginHandler.ParseInput(info);
10       DocomGUI.Properties.Settings.Default.SourceDocuments.Add(info.FullName);
11       model.SourceDocuments.Add(doc);
12       model.RaisePropertyChanged("SourceDocuments");
13     } catch (NotSupportedException) {
14       // Ausnahmen-Behandlung
15     }
16   }
17   CompositionListBox_SelectionChanged(sender, null);
```

Standard-Operationen, die in Programmen oft und wiederholt ausgeführt werden müssen, wie etwa das Auswählen einer bestimmten Datei durch den Benutzer, müssen dank der vorgefertigten Klassen von Microsoft nicht jedes Mal neu programmiert werden. So kann die in Zeile 2 instanzierte Klasse sofort verwendet werden, um den Benutzer um das Auswählen einer Datei zu bitten. Sobald das Ergebnis dieser Aktion vorliegt, kann mit dem Verarbeiten der Datei begonnen werden.

Dabei ist das Ergebnis des Dialogs von besonderem Interesse. Dieser wird in Zeile 4 mit einer besonderen Notation versehen; `bool?` anstatt dem üblichen `bool`. Diese Notation ist eine Kurzschreibweise für `Nullable<Boolean>`. Die generische Klasse `Nullable<T>` ermöglicht es dem generischen Typen `T`, einen zusätzlichen Zustand anzunehmen; den Null-Zustand. Dieser wird besonders im Umfeld von Datenbanken häufig dafür verwendet, nicht zugewiesene Felder zu markieren.

So kann ein boolesches Feld in einer Datenbank die Zustände `true`, `false` und `NULL` annehmen, wobei der letztere als dritte Möglichkeit zu wahr und falsch angibt, dass das Feld keines der beiden ist. Diese besondere Art der Logik ist besonders nützlich, um solche nicht-besetzten oder invaliden Felder zu markieren. Das Ergebnis des `OpenFileDialog`s ist auch von dieser Art. Der Benutzer kann entweder eine Datei auswählen, was in einem positiven Ergebnis des Dialogs resultiert oder den Vorgang abbrechen, was zum `false`-Ergebnis führt. Allerdings kann der Dialog auch in einen nicht definierten Zustand gebracht werden, was zu dem dritten Zustand, `NULL` führt.

Ist das Ergebnis positiv, wird die durch den Dialog angegebene Datei an das Framework zur Verarbeitung übergeben. Dabei muss zunächst ein für den Dateitypen zuständiges Plug-In gefunden werden, welches über die von den Plug-Ins implementierte Methode `CanHandle`, welche in Unterunterabschnitt 4.2.1 beschrieben wird, geschieht. Wird ein solches Plug-In gefunden, so wird diesem das `FileInfo`-Objekt als Parameter gegeben. Dieses Objekt enthält diverse Informationen über die Datei, z. B. den kompletten Dateinamen und den Pfad.

Hat das Plug-In seine Arbeit (das Parsen der Eingangsdaten in die interne Struktur `Document`) erledigt, wird das daraus entstandene Objekt in das *ViewModel* eingefügt. Nun muss dieses nur noch über die Veränderung der Eigenschaft benachrichtigt werden, was über die Methode `RaisePropertyChanged` geschieht. Dadurch wird die View dazu veranlasst, ihre Anzeige des ViewModels zu erneuern und den neuen Zustand dessen anzuzeigen.

Für das Hinzufügen von Quelldaten aus Datenbank-Tabellen konnte jedoch keine vorgefertige Methode verwendet werden. Hierfür wurde ein eigenes Fenster erstellt, das die Daten entgegennimmt. Diese ist in Abbildung 13 dargestellt. Der für dieses Fenster nötige XAML-Code ist sehr simpel und besteht nur aus einfachen Elementen und wird daher nicht explizit dargestellt.

Die wichtigen Elemente dieses Fensters sind die `TextBox`en und die `ComboBox`. In der ersten wird der in Unterunterabschnitt 5.4.3 beschriebene *Connection String* eingetragen, um die Verbindung zur Datenbank herzustellen. Dieser Vorgang wird durch die in Listing 27 gezeigte Methode bewerkstelligt.

Listing 27: Aufbau einer Datenbankverbindung mittels des Connection Strings

```
1 connection = new SqlConnection(ConnectionStringBox.Text);
2 connection.Open();
```

Darauffolgend sollen alle Tabellen, die in der im Connection String angegebenen Datenbank vorhanden sind, dem Benutzer zur Auswahl als Datenquelle zur Verfügung stehen. Da das Programm jedoch keine Information darüber haben kann, wie diese Tabellen in der Datenbank heißen und damit keine direkte Abfrage an sie möglich ist, muss diese Information aus einer Meta-Tabelle gezogen werden.

Diese Meta-Tabellen sind in jeder relationalen Datenbank vorhanden. Sie liefern Informationen über die Datenbank selbst, daher ihr Name. Im Falle einer MSSQL[29]-Datenbank heißt die Tabelle, die alle Informationen über die *echten* Tabellen der Datenbank enthält, treffenderweise `Tables`. Eine Abfrage des Inhaltes dieser Tabelle liefert dann diverse Informationen über die in der Datenbank definierten Tabellen. Diese Abfrage wird in Listing 28 gezeigt.

Listing 28: Abfrage der Tables-Metatabelle

```
1 DataTable table = connection.GetSchema("Tables");
2 TableBox.Items.Clear();
3 foreach (DataRow row in table.Rows)
4   if (row["TABLE_TYPE"].ToString().Equals("BASE TABLE"))
5     TableBox.Items.Add(row["TABLE_NAME"]);
```

Für die Abfrage spielt insbesondere der Wert der Spalte `TABLE_TYPE` eine wichtige Rolle. Darin ist der Typ der in dieser Zeile beschriebenen Tabelle gespeichert. Für die Tabellen-Komposition sollen nur vom Benutzer definierte Tabellen in Betracht kommen. Für diesen Tabellen-Typ ist der Wert *BASE TABLE* in der `Tables`-Tabelle vorgesehen. Durch die Schleife in Zeile 3 wird dann die `ComboBox` mit den Namen der jeweiligen Tabellen gefüllt.

Nachdem der Benutzer eine Tabelle als Datenquelle ausgewählt hat, muss er zusätzlich noch angeben, wieviele Zeilen aus dieser Tabelle als Quelldaten verarbeitet werden sollen. Dies ist wichtig, weil je nach Einstellung des SQL[30]-Servers nicht vorhergesehen werden kann, wie dieser mit einer Abfrage umgeht, die keine Begrenzung der Ergebnisse enthält. Nur wenn diese Angabe durch den Benutzer getätigt wird, kann die Tabelle als Datenquelle hinzugefügt werden.

Die Methode zum Hinzufügen einer Tabelle als Datenquelle nimmt zunächst die durch den Benutzer ausgewählte Tabelle als Parameter für die folgende Abfrage der Daten entgegen. Dabei wird das Ergebnis durch die Angabe der maximalen Anzahl an Zeilen limitiert. Die folgenden Aufrufe sind identisch mit der Methode, die einzelne Dateien als Quelle hinzufügt; es wird ein Plug-In für den Datentyp gesucht, die Daten werden von dem Plug-In überführt und dem *ViewModel* hinzugefügt.

[29] Microsoft SQL Server
[30] Standard Query Language

5.5.5 Ausführen der Komposition

Das Ausführen der Komposition über die GUI ist möglich, aber schlecht kontrollierbar. Zwar ist die Möglichkeit der direkten Eingabe von Parametern für die Komposition eine valide Möglichkeit, sie auszuführen, meistens wird dies aber durch automatisierte Prozesse und Programme deutlich effektiver ausgeführt.

Für die Demonstration der Funktionalität ist es aber ausreichend, die Komposition manuell auszuführen. Dabei werden die Parameter des Eingabefeldes direkt an die Komposition weitergegeben, sodass hier keinerlei Einschränkungen an die Korrektheit oder das Format der Parameter gestellt werden. Einziges Merkmal ist, dass die Eingabeparameter wie bei normalen, konsolenbasierten Programmen durch Leerzeichen getrennt werden.

Listing 29: Ausführen der Komposition in der Desktop-Anwendung

```
1 CompositionPlugin plugin = (CompositionPlugin)CompositionListBox.SelectedItem;
2 plugin.ExecuteComposition(ParameterTextBox.Text.Split(' '));
```

Ab dem Moment, in dem die `ExecuteComposition`-Methode des Plug-Ins in Listing 29 aufgerufen wird, geht auch die komplette Kontrolle des Prozesses an das Plug-In über. Sollte die Komposition korrekt abgelaufen sein, wird der Benutzer darüber in einer kleinen Informationsbox in Kenntnis gesetzt.

Ist dies nicht der Fall, tritt die Fehlerbehandlung in Kraft. Hier ist für den Benutzer zunächst nur die Information wichtig, dass etwas nicht korrekt abgelaufen ist. Tiefgehendere Informationen müssen dann der Logdatei entnommen werden, die über Log4Net geschrieben wird.

6 Entwicklung einer Kundenanwendung

6.1 Einleitung und Motivation

Ein solches System wird vor allem entwickelt, um für die oft anzutreffende Anforderung einer automatisierten Dokumentenkomposition eine Lösung zu finden. Eben diese Anfrage wurde von einem Kunden gestellt, für den das System nun entwickelt und für einen Beispieleinsatz an dessen Umgebung angepasst wird.

Die Anforderungs- und Testspezifikationen für dieses System sind dem Dokument angehängt. Aus Datenschutzgründen wurden kundenbezogene Daten geschwärzt. Prinzipiell geht es darum, Redakteuren das Zusammenstellen einer Dokumentation zu erleichtern.

Diese Dokumentation besteht aus vielen einzelnen Kapitel-Dateien, welche in einer Ordnerstruktur abgelegt und in einem Excel-Dokument aufgelistet sind. Dabei wird in diesem Excel-Dokument die letztendliche Struktur der Dokumentation festgelegt und danach manuell aus den Einzeldokumenten zusammengestellt. Dieser Prozess soll durch die Verwendung des hier entwickelten Frameworks vereinfacht werden.

Dabei ist unter anderem eine Struktur zu erstellen, die es dem Redakteur ermöglicht, die Struktur der Dokumentation in einer Excel-Tabelle zusammenzustellen. Diese Struktur soll dann durch die Kundenanwendung unter Zuhilfenahme des Frameworks in ein einheitliches Word-Dokument umgewandelt werden. Wie bereits in Unterabschnitt 4.4 im Word-Beispiel dargestellt, soll dabei aus mehreren Word-Dokumenten ein einzelnes entstehen.

6.2 Umsetzung

6.2.1 Kompositions-Tabelle

Um das entwickelte Framework für die Kundenanwendung zu verwenden, muss ein Rahmenprogramm entwickelt werden, das den Ablauf steuert und das Framework verwendet. Das Rahmenprogramm besteht aus einer ausführbaren Datei und einem Excel-Dokument, welches als Eingabeparameter dient und definiert, welche Einzeldokumente zur Erstellung des Gesamtdokuments benötigt werden.

Die Excel-Tabelle muss für die Funktion des Rahmenprogramms und der kundenspezifischen Plug-Ins dabei unbedingt folgende Informationen enthalten:

- Den Ordner, aus welchem die Einzeldokumente gelesen werden sollen
- Die Dateinamen der Einzeldokumente
- Eine Markierung, die angibt, ob ein Einzeldokument zur Komposition beitragen soll
- Einen Order (optional mit Dateiname), den das Gesamtdokument erhalten soll

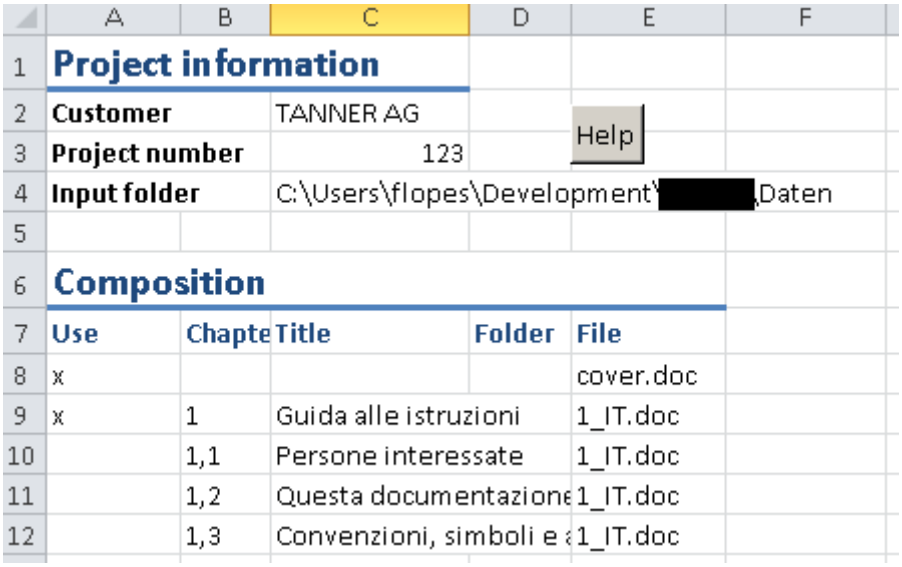


Abbildung 17: Kompositionszusammmenstellung in der Excel-Tabelle

Zu diesem Zweck wurde ein Entwurf der Excel-Tabelle entwickelt, in der ein Redakteur alle nötigen Informationen eingeben kann. Darin wird über eine bestimmte Spalte definiert, ob ein Einzeldokument enthalten sein soll oder nicht. Ist die Spalte mit einem x markiert, wird es in die Komposition aufgenommen, andernfalls nicht.

Diese Kompositionstabelle ist in Abbildung 17 dargestellt. Die *Chapter*- und *Title*-Spalten sind für die Funktionalität des Programms nicht relevant. Sie dienen lediglich der besseren Übersicht für den Redakteur.

Weiterhin muss für das erfolgreiche Ausführen der Komposition definiert werden, wohin das Ergebnisdokument gespeichert werden soll und mit welchem Namen. Dafür wurde am Ende der Tabelle, welche alle Einzeldokumente enthält, ein weiterer Abschnitt zur Konfiguration dieser Parameter eingeführt, der in Abbildung 18 gezeigt wird.

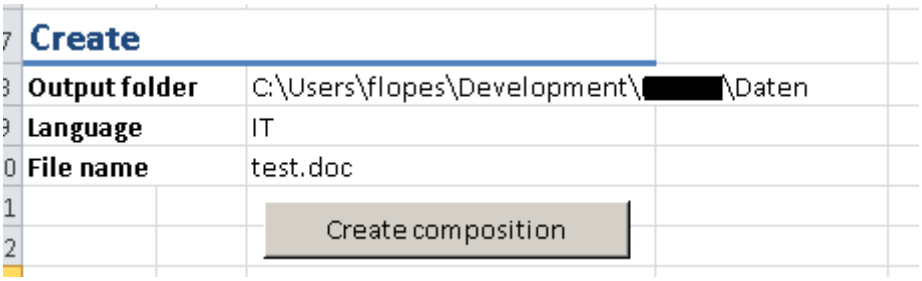


Abbildung 18: Ergebnisparameter für das Erstellen der Komposition

Das Ausführen der Komposition wird durch das Klicken des *Create Composition*-Buttons ausgelöst. Dies startet das ebenfalls entwickelte Rahmenprogramm, welches die Daten der Excel-Tabelle entgegennimmt und das Framework für die Erstellung der Komposition verwendet.

6.2.2 Konfiguration des Rahmenprogramms

Um dies zu ermöglichen, ist es jedoch nötig, den genauen Ort zu kennen, in welchem die ausführbare Datei des Rahmenprogramms zu finden ist. Daher ist in einer zweiten Tabelle in der Kompositions-Datei eine Konfiguration erstellt worden, die es dem Benutzer ermöglicht, einige Parameter des Programms anzupassen, unter anderem auch den Pfad zur EXE-Datei.

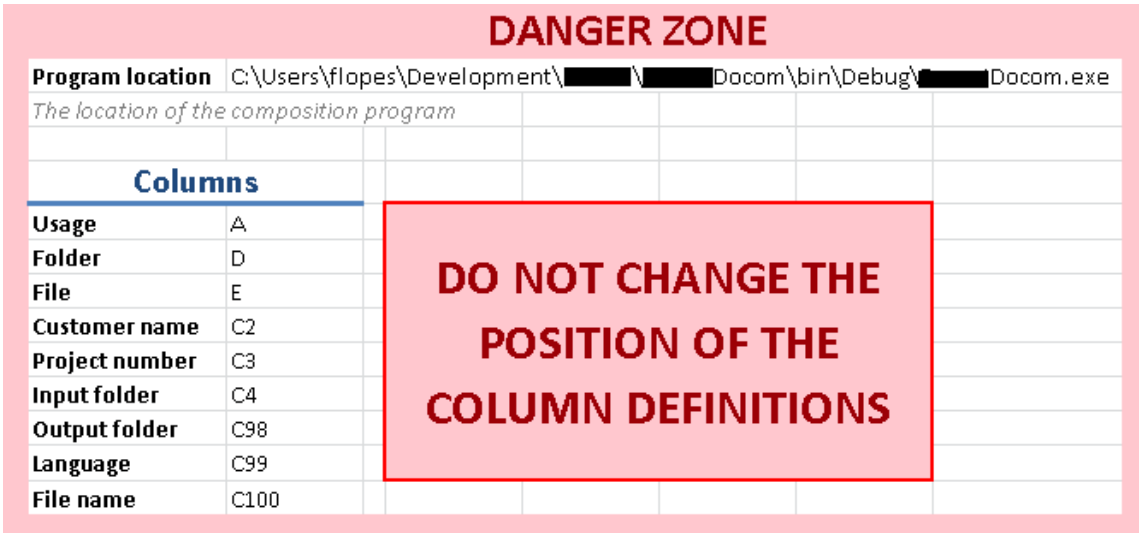


Abbildung 19: Die Konfigurationstabelle für die Kompositionsdatei

Diese in Abbildung 19 gezeigte Konfigurationstabelle enthält noch weitere Einstellungsmöglichkeiten, wie die Definitionen der Spalten und Zellen, aus welchem das Programm seine Eingangsdaten beziehen soll. Dies wurde in die Konfiguration aufgenommen, damit normale Benutzer oder Administratoren so wenig Aufwand wie möglich in die Konfiguration des Rahmenprogrammes investieren müssen.

Sollte nun z. B. die Spalte, in welcher die Dateinamen der Einzeldokumente stehen, verändert werden, muss dies nur einmal, in der Konfigurationstabelle, geändert werden. Diese Änderung ist für die meisten Nutzer wesentlich einfacher verständlich als das Bearbeiten einer XML-Datei (die gemeinhin für Konfigurationen verwendet werden).

Wichtig ist jedoch, wie es die Warnung in der Konfigurationstabelle bereits andeutet, dass die Position der Definitionen selbst nicht verändert werden. Diese sind nämlich direkt im Code des Rahmenprogrammes eingebettet, da dieses sonst keinerlei Möglichkeit hat, zu wissen, in welchen Spalten diese Definitionen stehen. Sollten diese Definitionen verschoben werden, kann das Rahmenprogramm nicht mehr korrekt arbeiten.

Die Konfiguration von zwei weiteren wichtigen Elementen ist jedoch weiterhin in einer XML-Datei untergebracht, da diese nur von erfahrenen Benutzern geändert werden können sollen; der Pfad zur Plug-In-Assembly und die Logging-Konfigurationen. Diese werden in einer besonderen XML-Datei abgelegt, der `app.config.xml`. Diese kann vom Rahmeprogramm geladen werden und dann zum Auslesen der darin definierten Einstellungen verwendet werden. In diese XML-Datei kann man jegliche selbst definierten Einstellungen schreiben und diese mit speziellen Klassen und Methoden auswerten.

Listing 30: XML-Konfigurationsdatei des Rahmenprogrammes

```
<configuration>
  <configSections>
    <section name="log4net"
             type="log4net.Config.Log4NetConfigurationSectionHandler, log4net"
                />
    <section name="docom"
             type="Docom.Config.DocomConfigurationSectionHandler, Docom" />
  </configSections>
  <docom>
    <!-- The file that contains the plugin classes -->
    <PluginAssembly file="C:\Users\flopes\Development\DocomPlugins\bin\Debug\
        DocomPlugins.dll" />
  </docom>
```

Listing 30 zeigt den Ausschnitt der Konfigurations-XML-Datei, der die selbst definierten Abschnitte zeigt. Um einen eigenen Konfigurationsabschnitt zu definieren, muss dafür zunächst eine `section` erstellt werden, die einer speziellen Klasse, dem `ConfigurationSectionHandler` zugewiesen ist. Diese Klasse enthält Eigenschaften, die den jeweiligen XML-Elementen innerhalb dieser Sektion entsprechen.

So wird das XML-Element `PluginAssembly` auf die Eigenschaft `PluginAssembly` vom Typ `PluginAssemblyElement` der Konfigurations-Handler-Klasse abgebildet, wie in Listing 31 zu sehen ist.

Listing 31: Konfigurations-Handler des Rahmenprogrammes

```
class DocomConfigurationSectionHandler : ConfigurationSection {
    [ConfigurationProperty("PluginAssembly", IsRequired = true)]
    public PluginAssemblyElement PluginAssembly {
        get { return (PluginAssemblyElement)this["PluginAssembly"]; }
```

Jedes einzelne Element innerhalb der selbst definierten Sektion der XML-Konfiguration ist wiederum eine eigene Klasse, in welcher die Struktur des XML-Elements abgebildet ist. So ist in Listing 32 die sehr einfache Klasse gezeigt, welche das `PluginAssembly`-XML-Element abbildet. Da dieses nur einen einzelnen Parameter, `file`, enthält, ist dieser auch die einzige Eigenschaft der Klasse, die abgebildet werden muss.

Listing 32: PluginAssembly-Klasse zur Abbildung des XML-Elements

```
1 class PluginAssemblyElement : ConfigurationElement {
2     [ConfigurationProperty("file", IsRequired = true)]
3     public string File {
4         get { return (string)this["file"]; }
```

Durch diese Abbildung von XML-Elementen in Klassen und Eigenschaften kann im Rahmenprogramm auf sehr einfache Weise auf die in der XML-Datei konfigurierten Elemente und deren Werte zugegriffen werden, wie beispielhaft in Listing 33 gezeigt.

Listing 33: Zugriff auf XML-Konfigurationen über Abbildungsklassen

```
1 Config.DocomConfigurationSectionHandler config = Config.
      DocomConfigurationSectionHandler.GetConfig();
2 logger.Info("Registering assembly " + config.PluginAssembly.File);
```

Durch den Aufruf dieser Methoden hat man direkten Zugriff auf den im XML-Element `PluginAssembly` und dessen Paremeter `File` gespeicherten Wert. Alle weiteren sonst manuell durchzuführenden Aktionen (XML-Dateien laden, parsen und nach der gewünschten Information durchsuchen) entfallen und werden durch das .NET-Framework durchgeführt.

Eine weitere wichtige Aufgabe der XML-Konfiguration ist die Einstellung des Loggings der log4Net-Bibliothek. Diese ist ein wesentlicher Bestandteil des Projektes und oft die einzige Möglichkeit, Fehler im Nachhinein erkennen und lösen zu können.

Durch geeignete Einstellungen werden Log-Nachrichten, die im Programm ausgegeben werden, sowohl auf der Konsole (durch den `ConsoleAppender`) ausgegeben als auch in eine Logdatei geschrieben (durch den `FileAppender`). Die dargestellten Einstellungen sind für den Entwicklungsbetrieb optimiert, da Lognachrichten unabhängig von ihrem Grad mitgeschrieben werden.

Für den produktiven Einsatz sollte der Grad auf `INFO` gesetzt werden, sodass Nachrichten mit dem `DEBUG`-Level nicht mehr angezeigt werden. Diese sind meistens nur für Entwickler interessant und helfen nur bedingt bei dem Auffinden von fehlerhaftem Verhalten des Programms.

Dies funktioniert bei Log4Net durch den Mechanismus einer einfachen Level-Hierarchie. Jede Nachricht ist mit einem bestimmten Grad versehen, die ihre *Wichtigkeit* widerspiegelt. Der Logger-Klasse ist ebenfalls ein Grad zugewiesen. Nur, wenn eine Nachricht gleich oder größer dem Grad des Loggers ist, wird diese auch durch die `Appender` auf die für sie definierte Art und Weise geschrieben. Dabei gilt das in Gleichung 1 definierte Verhältnis zwischen den einzelnen Log-Graden.

$$DEBUG < INFO < WARN < ERROR < FATAL \tag{1}$$

So wird eine Nachricht des Grades *WARN* nur dann geschrieben, wenn der entsprechende Logger ein Level kleiner oder gleich *WARN* hat. Eine *INFO*-Nachricht würde in so einem Logger nicht geschrieben werden, eine *ERROR*-Nachricht hingegen schon.

6.2.3 Funktionsweise

Das Rahmenprogramm nimmt zur Ausführung einen Parameter entgegen; den Dateipfad der Kompositions-Tabelle. Darin sucht das Programm nach allen Einzeldokumenten, die zur Verwendung markiert sind. Dies wird anhand der in der Konfigurationstabelle definierten Spalten und Zellen entschieden. Die dadurch gesammelten Einzeldokumente werden durch das dafür zuständige Dokumenten-Plug-In in `Document`-Objekte umgewandelt. Die Liste all dieser Dokumente wird dem zuständigen Kompositions-Plug-In übergeben, welches sie in Ergebnisblöcke umwandelt und anordnet. Das Kompositions-Plug-In kann dann durch Angabe eines Dateinamens für das Ergebnisdokument die Komposition ausführen.

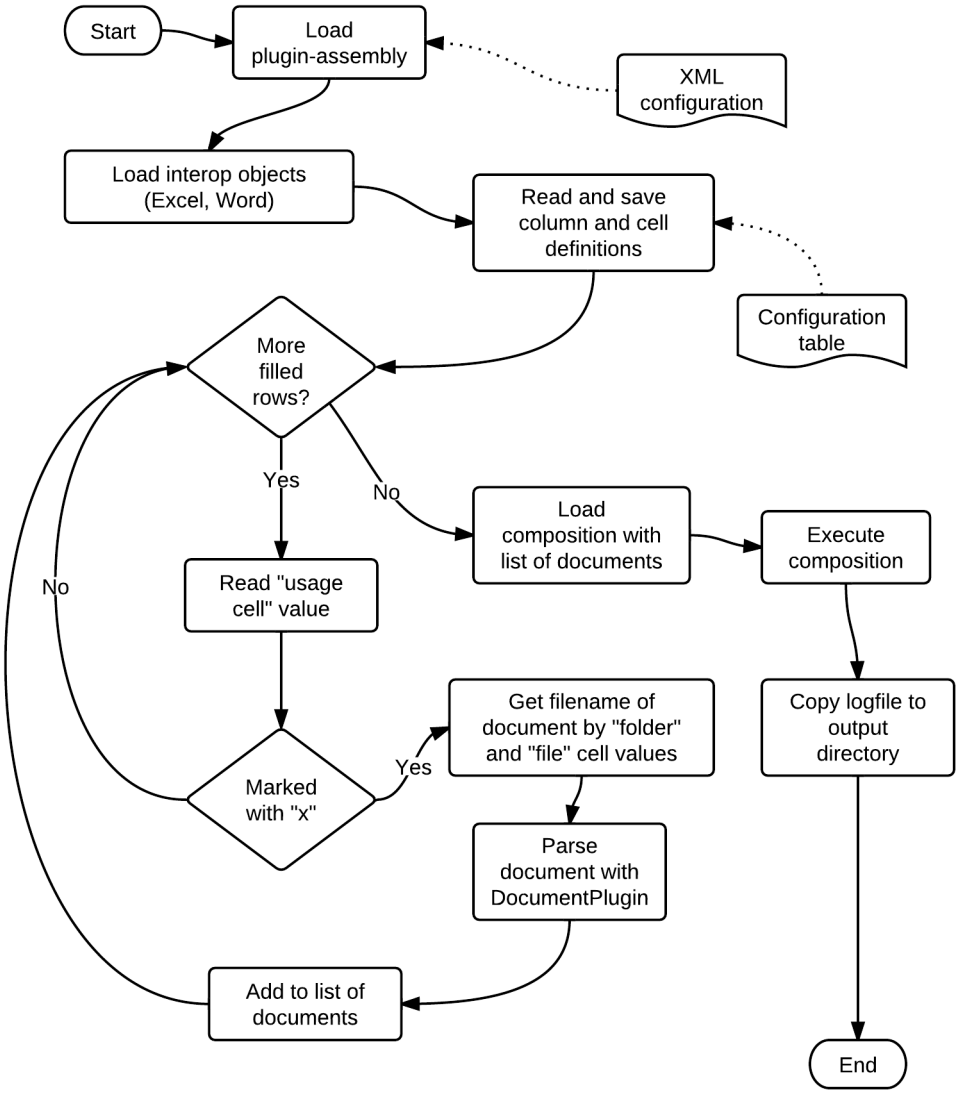


Abbildung 20: Ablauf des Rahmenprogramms

Der Gesamtprozess ist in Abbildung 20 dargestellt. Sollten bei der Ausführung des Programms Fehler auftreten, wird der Benutzer darüber benachrichtigt. Nachdem das Kompositions-Plug-In die Komposition ausgeführt hat, wird die durch Log4Net erstellte Log-Datei in das Ausgabe-Verzeichnis kopiert. Dies soll es dem Benutzer ermöglichen, bei etwaigen Fehlern direkt die Log-Datei zu öffnen und so möglicherweise den Fehler selbst zu beseitigen.

Wenn ein Einzeldokument nicht gefunden werden kann, ist es dem Benutzer freigestellt, das Programm abzubrechen oder den Fehler zu ignorieren und die Komposition ohne dieses Einzeldokument fortzuführen[31]. Die Plug-Ins wurden in Abbildung 20 nicht näher dargestellt, da ihre Funktion differenzierter in Unterabschnitt 6.3 erläutert wird.

6.3 Plug-Ins

6.3.1 Dokumenten-Plug-In

Das Dokumenten-Plug-In für die Kundenanwendung soll, ähnlich wie das bereits in Unterabschnitt 4.4 gezeigte Word-Beispiel, Word-Dokumente entgegennehmen und diese als kleinste zu verarbeitende Einheit behandeln und in `Document`-Objekte überführen. Dabei sollen die Einzeldokumente, die in der Kompositions-Tabelle markiert sind, zu einem großen Gesamtdokument zusammengestellt werden.

Der gesamte Inhalt des Word-Dokuments soll dabei im Ergebnisdokument wieder erscheinen, daher ist eine Untergliederung des `Document`-Objekts in weitere `DocumentBlock`-Elemente nicht sinnvoll. Das Dokument wird daher ,wie in Listing 34 gezeigt, nur anhand seines Dateinamens erkannt. Dieser wird in Form eines `FileInfo`-Objektes als `Name`-Eigenschaft des `Document`-Objektes gespeichert.

Listing 34: Dokumenten-Plug-In für die Kundenanwendung

```
1 public override Tanner.Docom.Document ParseInput(object file) {
2     FileInfo fileInfo = (FileInfo)file;
3     Document document = new Document(fileInfo.FullName);
4     logger.Debug("Created new document: " + document);
5     return document;
6 }
7 public override Boolean CanHandle(SourceType type) {
8     return type.Equals("doc");
```

Dadurch ist der Name des `Document`-Objekts nun gleich dem Dateipfad der originalen Word-Datei. Diese Objekte können nun dem Kompositions-Plug-In zur Überführung in Ergebnisblöcke übergeben werden.

[31] Diese Entscheidung ist in Abbildung 20 nicht dargestellt, da sie keinen wesentlichen Einfluss auf die Funktionsweise des Rahmenprogramms hat.

6.3.2 Kompositions-Plug-In

Das Kompositions-Plug-In nimmt im ersten Schritt eine Liste aller `Document`-Objekte entgegen, die durch das Rahmenprogramm erkannt und dem Dokumenten-Plug-In überführt wurden. Dabei ist das Überführen der `Document`-Objekte in `ResultBlock`-Objekte sehr einfach, da keinerlei tiefere Verschachtelung stattfinden muss.

Listing 35: Überführen der Document-Objekte in ResultBlock-Objekte

```
1 public override void Load(params Document[] documents) {
2     ResultBlock result = new ResultBlock(BlockType.FILE, "Composition result");
3     foreach (Document document in documents) {
4         ResultBlock block = new ResultBlock(BlockType.BLOCK, document.Name);
5         logger.Debug("Adding new block: " + block);
6         result.ChildBlocks.Add(block);
7     }
8     Blocks.Add(result);
```

In Listing 35 wird die Komposition erstellt. Alle Einzeldokumente werden direkt in `ResultBlock`-Objekte mit dem Typ `BLOCK` überführt. Weiterhin wird ein *Wurzel*-Block erstellt, der als einziger vom `FILE`-Typ ist. In diesem werden alle Blöcke der Einzeldokumente als Kinder hinzugefügt. Damit ist das Grundgerüst der Komposition vollständig erstellt.

Es wird dabei keinerlei Abfrage auf eventuell bereits vorhandene und damit doppelt hinzugefügte Einzeldokumente getätigt, da die Verhinderung dieses Umstandes in den Aufgabenbereich des Benutzers fällt.

Die Ausführung der Komposition wird dann anhand der geladenen Ergebnisblöcke durchgeführt. Dabei wird jedes Einzeldokument anhand seines Namens (der gleich dem Dateipfad der Originaldatei ist), mit den durch Word zur Verfügung gestellten Methoden in das Ergebnisdokument kopiert. Wie bei dem in Unterabschnitt 4.4 gezeigten Word-Beispiel geschieht dies durch die Simulierung eines echten Kopiervorganges.

Der Vorgang ist dabei identisch zu dem, der bereits für das Beispiel in Listing 13 verwendet wurde. Zusätzlich wurden jedoch die in Listing 36 dargestellten Methoden erstellt.

Listing 36: Ausführen der Komposition

```
1 foreach (Word.TableOfContents toc in doc.TablesOfContents) {
2     toc.Update();
3 }
4 int fieldErrors = 0;
5 foreach (Word.Field field in doc.Fields) {
6     field.Update();
7     if (field.Result.Text.StartsWith("Fehler!"))
8         ++fieldErrors;
9 }
```

Bei der Erstellung des Ergebnisdokumentes werden auch Überschriften und dokumentinterne Referenzen kopiert. Da diese jedoch in den Einzeldokumenten andere Pfade und Kapitelkennzeichnungen enthalten, müssen sie nach dem Zusammenstellen aktualisiert werden. Dazu werden zum einen alle Kapitel neu nummeriert (Zeile 1) und zum anderen alle Referenzen (z. B. auf Bilder, andere Kapitel oder Tabellen), welche in der Word-API als `Fields` benannt werden, aktualisiert (Zeile 6).

Dabei ist leicht ersichtlich, dass möglicherweise nicht alle Referenzen innerhalb des Gesamtdokumentes korrekt aufgelöst werden können. So ist es z. B. zwar möglich, in einem Einzeldokument die Referenz auf ein anderes Kapitel anzugeben, wenn das Einzeldokument, in welchem sich dieses referenzierte Kapitel befindet, aber nicht zum Gesamtdokument hinzugefügt wird, kann die Referenz folgerichtig nicht aufgelöst werden.

Um den Benutzer über solche Fehler zu informieren, werden nach der Akutalisierung aller Felder diese auf Fehler untersucht. Nicht auflösbare Referenzen werden nach dem Aktualisieren durch Word selbst mit einer Fehlermeldung ersetzt, die mit dem Schlüsselwort *Fehler!* beginnt. Nachdem alle diese fehlerhaften Referenzen gefunden wurden, kann der Benutzer darüber informiert werden.

Schlussendlich wird, wie in Listing 13 auch, das Dokument abgespeichert. Der Name des neuen Ergebnisdokumentes wird dabei als erster Parameter für die `Execute`-Methode erwartet.

6.4 Verwendung der Desktop-Anwendung

Die Tatsache, dass für die Kundenanwendung ein Rahmenprogramm entwickelt wurde, ist vor allem mit der Anforderung der direkten Verwendung aus der Excel-Tabelle heraus zu erklären.

Durch diese Art der Verwendung ist ein konsolenbasiertes Rahmenprogramm wesentlich effektiver einzusetzen und verlangt dem Benutzer weniger zusätzlichen Einarbeitungsaufwand ab als die Verwendung der Desktop-Anwendung. Praktisch ist es jedoch ebenso gut möglich, die Desktop-Anwendung mit den für den Kunden entwickelten Plug-Ins zu verwenden.

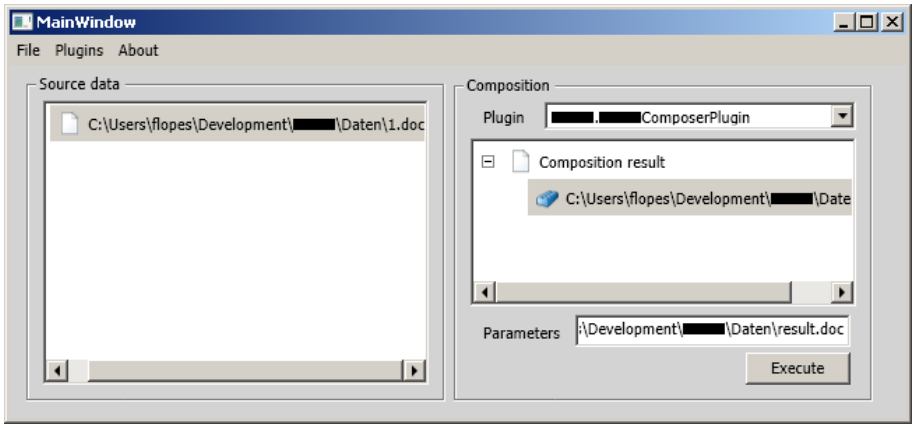


Abbildung 21: Ansicht einer Komposition mit den Kunden-Plug-Ins in der Desktop-Anwendung

Dabei müssen diese lediglich über die Plug-In-Verwaltung geladen werden. Danach können die Einzeldokumente ebenso wie in der Kompositions-Tabelle einzeln ausgewählt und für die Komposition vorbereitet werden.

Dabei ist der Umweg, jedes mal die Datei selbst über den Dialog auszuwählen, natürlich deutlich größer, als nur die Zeile der Datei mit einem x zu markieren.

Abbildung 21 zeigt die Ansicht, die sich in der Desktop-Anwendung bei Verwendung der kundenspezifischen Plug-Ins ergibt. Dabei kann auch hier die Komposition direkt ausgeführt werden, indem als Parameter für das Kompositions-Plug-In der Dateipfad angegeben wird, an dem das Ergebnisdokument abgelegt werden soll.

7 Diskussion

7.1 Qualitative Bewertung

7.1.1 Vollständigkeit

Sowohl das Frameworks als auch die Desktop-Anwendung können nicht als endgültig *fertig* bezeichnet werden. Um diesen Zustand zu erreichen, muss im Bereich der Verwaltungsaufgaben sowohl im Framework als auch in der Desktop-Anwendung noch nachgearbeitet werden.

So fehlt in der Framework-Klasse eine zufriedenstellende Möglichkeit, Dokumente oder Plug-Ins wieder zu entfernen. Dabei ist das Fehlen vor allem damit zu erklären, dass zur Zeit der Entwicklung keine ausreichend zuverlässige und sinnvolle Methode gefunden werden konnte, die diese Aufgabe auch in Zukunft lösen könnte.

Dabei ist gerade das Entfernen auf Ebene des Frameworks ein Problem, da hier keine Information darüber besteht, welche konkreten Plug-Ins aktuell geladen sind, da alles über Schnittstellen-Definitionen gehandhabt wird. In der GUI-Anwendung hingegen wäre die Lösung durch das Fenster der Plug-In-Verwaltung relativ einfach zu gestalten.

Diese beiden Gegensätze konnten zur Entwicklungszeit nicht zufriedenstellend vereint werden, weshalb sie nicht in das hier beschriebene Produkt übernommen wurden. Die vergleichsweise niedrige Priorität des Entfernens von Dokumenten und Plug-Ins spielte dabei ebenfalls eine nicht unwesentliche Rolle.

7.1.2 Robustheit

Sowohl Framework als auch Desktop-Anwendung sind für sich genommen durch geeignete Ausnahmenbehandlung stabil lauffähig. Sollten dennoch Fehler auftreten, können diese durch die Log4Net-Lognachrichten nachvollzogen werden. Dabei verwenden beide Komponenten ausschließlich Ausgaben, die über die von Log4Net definierten Kanäle ausgegeben werden.

Voraussetzung dafür ist allerdings, dass das verwendete Rahmenprogramm diese Kanäle ausreichend konfiguriert. In der Desktop-Anwendung ist dies bereits getan. Wird das Framework ohne eine solche Konfiguration verwendet, kann keine Aussage darüber getroffen werden, wie Lognachrichten an die Umgebung weitergegeben werden.

Allgemein lässt sich sagen, dass alle entwickelten Komponenten in einem für ihren Bereich ausreichendem Maße zuverlässig arbeiten und keine groben Speicher-, Zugriffs- oder Sicherheitsfehler auftreten.

Auch wenn Plug-Ins Abstürze verursachen, ist die Desktop-Anwendung weiterhin lauffähig. Da das Framework selbst keine ausführbaren Komponenten besitzt, sondern hauptsächlich zu Verwaltungszwecken besteht, ist hier keine Aussage über die Robustheit möglich.

7.1.3 Einsetzbarkeit

Das Framework und die Desktop-Anwendung können in der Form, die in dieser Arbeit beschrieben und erkärt wurde, zur Erfüllung vieler verschiedener Anforderungen eingesetzt werden. Besonders lohnend sind dabei verständlicherweise Aufgaben der Prozessautomatisierung, die unmittelbar mit Dokumenten oder Datenquellen im weiteren Sinne zu tun haben.

Die Desktop-Anwendung kann aufgrund ihrer sehr generischen Implementierung mit allen Plug-Ins umgehen, die die vom Framework angebotenen Schnittstellen einbinden. Hier muss lediglich eine zusätzliche Möglichkeit für das Hinzufügen von speziellen Datentypen, die nicht direkt in Dateien vorhanden sind, entwickelt werden (wie es im Beispiel für Datenbanktabellen getan wurde).

Dadurch ist das entwickelte System für die Erfüllung sehr vieler erdenklicher Aufgaben im Bereich der Dokumentenkomposition optimal gerüstet. Auch die komplette Automatisierung solcher Kompositionen ist, sofern dafür ein Rahmenprogramm entwickelt wird, einfach möglich.

7.1.4 Korrektheit

Das Framework liefert bei korrekter Verwendung stets vorhersehbare Ergebnisse, die ausschließlich von den eingegebenen Parametern beeinflusst werden. Gleiches gilt auch für die Desktop-Anwendung. Hier spielen selbstverständlich auch die verwendeten Plug-Ins eine große Rolle. Sind diese nicht ausreichend getestet oder unzuverlässig programmiert, wird auch das Framework bzw. die Desktop-Anwendung schlechte Ergebnisse zeigen.

Dabei kann jedoch nicht die korrekte Arbeitsweise des Frameworks oder der Desktop-Anwendung in Mitleidenschaft gezogen werden, da diese keinerlei Einfluss auf die von den Plug-Ins durchgeführten Aktionen haben können (und sollen).

7.1.5 Zukunftsfähigkeit

Durch die Plug-In-Architektur des Frameworks ist auch für zukünftig anstehende Anforderungen die Erweiterbarkeit des Systems sichergestellt. Da beide Komponenten auf modernen Programmiersprachen und Umgebungen basieren, ist zumindest mittelfristig keine schwerwiegende *Veralterung* der Code-Basis zu erwarten.

Auch ist die Portierung des entwickelten Systems auf neue Versionen der .NET-Umgebung durch die Struktur von .NET und die Entkopplung des Codes von der Plattform, auf welcher er ausgeführt wird, sehr einfach und mit vergleichsweise geringem Aufwand verbunden.

7.1.6 Wartbarkeit

Für die Wartbarkeit des Systems sind gute Grundvoraussetzungen geschaffen worden. Durch die dezentrale Organisation in mehrere kleine Komponenten, die unabhängig voneinander ausgetauscht werden können und nur mit Schnittstellen anstelle von konkreten Implementierungen arbeiten, können Fehler in Komponenten gezielt beseitigt und durch Austauschen der fehlerhaften Komponente beseitigt werden.

Durch die Verwendung gängiger Dritt-Komponenten wie Log4Net, welche separat getestet und entwickelt werden, sind auch die von ihnen angebotenen Funktionalitäten einfach durch robustere und neuere Versionen austauschbar.

Mittels durchgehender Code-Dokumentation (s. Anhang D) ist auch für andere Entwickler das Verstehen des entwickelten Systems einfach, was die Wartbarkeit durch andere Personen sichert.

7.2 Anforderungsabdeckung

7.2.1 Modularität

Die Modularität, wie in Unterabschnitt 2.2 definiert, ist durch die Plug-In-Struktur des Frameworks ausreichend erfüllt. Neue Plug-Ins, die Methoden zur Verarbeitung von neuen Dokumenttypen oder andere Arten der Komposition bereitstellen, können separat entwickelt werden und über ihre DLL-Assemblies geladen werden.

Unzureichend ist jedoch die Kapselung der Plug-Ins und die damit verbundene Abhängigkeit zwischen den Dokumenten- und Kompositions-Plug-Ins. Das wird besonders dann deutlich, wenn versucht wird, von einem Dokumenten-Plug-In umgewandelte `Document`-Objekte mit einem Kompositions-Plug-In zu verarbeiten, welches nicht direkt mit dem verwendeten Dokumenten-Plug-In entwickelt wurde.

Die Problematik liegt hier darin, dass das Kompositions-Plug-In keinerlei Informationen darüber haben kann, wie das Dokumenten-Plug-In die Daten aus der Quelle speichert und verarbeitet hat. Grund für dieses Problem ist die Speicherung des Inhaltes eines Dokumentblockes als `object` -Typ, wie in Abbildung 6 zu sehen ist.

Da `object` die Basisklasse *aller* Klassen in C# ist, kann ein Kompositions-Plug-In ohne weiteres Wissen über die genaue Art des Inhaltes von `Content` keine sinnvolle Aktion durchführen, die den Inhalt dieser Eigenschaft der `DocumentBlock`-Klasse verarbeitet.

Umstrukturierung zu generischen Klassen Eine ursprünglich in Erwägung gezogene Lösung, die Umwandlung des Datenmodells in generische Klassen, hat sich bei genauerer Prüfung als nicht praktikabel erwiesen. So wäre es prinzipiell zwar möglich, das gesamte Datenmodell in generische Klassen in der Form `DocumentBlock<T>` umzuwandeln, jedoch ist damit die Problematik nur verschoben und nicht beseitigt.

Ein generisches Plug-In zur Verarbeitung von Word-Dokumenten würde über `CompositionPlugin <Word._Document>` gekennzeichnet werden. Der Inhalt dieses Word-Dokumentes kann sich jedoch erheblich von dem eines anderen Dokumentes unterscheiden. Mit generischen Datentypen wäre damit keine Unterscheidung möglich, wie genau der Inhalt der jeweiligen Objekte verarbeitet wird. Ein Plug-In, welches Word-Dokumente auf eine Weise verarbeitet, könnte demnach mit anderen Word-Dokumenten, die sich in ihrer Struktur unterscheiden, nicht mehr umgehen.

Demzufolge ist die Unterscheidung durch `SourceType`-Objekte besser geeignet, da hier vollkommene Freiheit in der Definition der Quelltypen besteht, da diese auf `string`s basieren.

7.2.2 Regelbasiertheit

Das System kann durch Regeln, die von außen in die Plug-Ins hineingegeben werden, kontrolliert werden. Es wird bei gleichen Eingaben die gleichen Ergebnisse liefern, sofern die Plug-Ins selbst ebenfalls deterministisch aufgebaut sind (s. Unterunterabschnitt 7.1.4).

Eine Problematik besteht auch hier vor allem in der Entwicklung der Plug-Ins. Da diese den Hauptteil der Arbeit des gesamten Systems erledigen, sind auch sie verantwortlich für die korrekte Verarbeitung der Eingangsdaten und das korrekte Erstellen der Ergebnisse.

7.2.3 Automatisierung

Das System ist durch geeignete Programmierung von Rahmenprogrammen, welche die Verwendung des Frameworks und der Plug-Ins übernehmen, komplett automatisierbar. Dabei liegt auch hier wieder viel der Verantwortung bei den Entwicklern der Plug-Ins. Sind diese abgeschlossen und korrekt programmiert, lassen auch sie sich ausschließlich durch automatisierte Prozesse steuern.

7.2.4 Komposition

Die Komposition ist, sofern das Kompositions-Plug-In und das Dokumenten-Plug-In aufeinander abgestimmt sind, korrekt funktionsfähig. Das Framework bietet alle benötigten Methoden und Schnittstellen für die Plug-Ins an, damit diese korrekt arbeiten können.

Auch hier besteht das bereits in Unterunterabschnitt 7.2.1 und Unterunterabschnitt 7.1.4 angesprochene Problem von unterschiedlichen Plug-In-Implementierungen. Nur wenn die beiden Plug-Ins die Daten auf gleiche Weise verarbeiten, kann eine Komposition fehlerfrei erstellt werden.

8 Ausblick

8.1 Framework

Beim entwickelten Framework ist, wie in Unterunterabschnitt 7.1.1 beschrieben, gerade im Bereich der Verwaltungsaufgaben für Plug-Ins und Dokumente deutliches Optimierungspotenzial sichtbar. Die Entwicklung der fehlenden Basisoperationen zum Entfernen von Plug-Ins und Dokumenten könnten dem Framework noch deutliche Vorteile bringen. Ebenso würden diese das Arbeiten mit der Desktop-Anwendung weiter vereinfachen.

Weiterhin ist eine zufriedenstellende Lösung für das in Unterunterabschnitt 7.1.4, Unterunterabschnitt 7.2.1 und Unterunterabschnitt 7.2.4 angesprochene Problem der unterschiedlichen Verarbeitungsweise von Dokumenten-Plug-Ins und Kompositions-Plug-Ins problematisch. Einige Ansätze könnten hier gute Ergebnisse liefern, konnten in der knappen Zeit des vorliegenden Projektes jedoch nicht weiter verfolgt werden.

So könnte gerade die bereits für das Erkennen der Plug-In-Assemblies verwendete Reflection dazu verwendet werden, genauere Informationen über die im `Content` eines `DocumentBlock`s gespeicherten Werte zu erhalten. Damit wäre ein Kompositions-Plug-In in der Lage, variabel auf diese Informationen zu reagieren und je nach Objekttyp unterschiedliche Aktionen durchzuführen.

Eine andere Möglichkeit ist das Vereinheitlichen der Eingangsdaten in ein spezielles Datenmodell. Dieses Modell könnte anstelle des aktuellen treten und die Verarbeitung von Daten für die Kompositions-Plug-Ins vereinheitlichen. Problematisch an dieser Lösung ist, dass es möglicherweise Situationen geben kann, in welchen dieses vorgefertige Datenmodell nicht ausreichend fein granulierte Einstellungsmöglichkeiten bietet, um die für die Komposition nötigen Daten zu speichern und weiterzugeben.

Ein Beispiel hierfür wäre die bereits in Unterunterabschnitt 7.2.1 erwähnte unterschiedliche Verarbeitung eigentlich identischer Datentypen. Es müsste hier eine Lösung erarbeitet werden, die für alle möglichen Situation unterschiedliche Sichten auf die gleichen Basisdaten liefert, von denen ausgehend die Kompositions-Plug-Ins die für sich nötigen Daten ermitteln können.

8.2 GUI

Bei der Desktop-Anwendung ist ebenso deutliches Optimierungspotenzial sichtbar. Dieses liegt besonders im Bereich der Usability, da hierauf während der Entwicklung wenig Wert gelegt wurde.

Funktionalität stand hier eindeutig im Vordergrund. So wäre es für das Hinzufügen von neuen Quelldaten denkbar, die Möglichkeit anzubieten, vorhandene Dokumente durch Drag&Drop zu den Quelldokumenten hinzuzufügen. Dies ist besonders bei mehreren Dokumenten eine deutliche Zeitersparnis für den Benutzer.

Auch das Speichern und Wiederherstellen von bereits geladenen Plug-Ins und Quelldokumenten zwischen einzelnen Sitzungen steigert die Benutzerfreundlichkeit erheblich.

Gepaart mit der zusätzlichen Funktion des Löschens von Dokumenten und Plug-Ins, wie in Unterunterabschnitt 7.1.1 erwähnt, würde die Desktop-Anwendung zu einem mächtigen Programm werden, mit welchem sich Kompositionen jedweder Art komfortabel erstellen und verwalten ließen.

Zusätzlich ist eine Art *Projektdatei* denkbar, in welcher geladene Plug-Ins, Quelldokumente und ausgewählte Kompositionen speicherbar sind. Diese Dateien könnten von Redakteuren ausgetauscht werden und beim Öffnen direkt alle benötigten Dokumente und Plug-Ins in die GUI laden und damit die Komposition direkt ausführbar machen, ohne dass manuell alle Einstellungen getroffen werden müssen.

Die Plug-In-Struktur des Frameworks lässt sich ebenso auf die Desktop-Anwendung abbilden, in dem z. B. dafür spezialisierte GUI-Plug-Ins die Verwaltung von Kompositionen anbieten. Diese könnten mit den normalen Kompositions- oder Dokumenten-Plug-Ins gebündelt werden und somit beim Laden des *Basis-Plug-Ins* sofort die Desktop-Anwendung um entsprechende Felder, Tabs oder Fenster erweitern, die das Eingeben der Daten vereinfachen.

9 Fazit

Die Anforderung, ein modulares System zur regelbasierten und automatisierten Dokumentenkomposition wurde durch die vorliegende Arbeit sowohl in den Bereichen des Grundsystems als auch beispielhaft für eine Kundenanwendung bearbeitet. Dabei konnte eine Lösung gefunden werden, die modulares Entwickeln und damit die Anpassung an unterschiedliche Anforderungen ermöglicht.

Eine speziell für die Verwendung dieser modularen *Plug-Ins* entwickelte Desktop-Anwendung ermöglicht die Veranschaulichung und Vorschau der Zusammenstellung von Dokumenten. Es wurde jedoch festgestellt, dass für eine rein automatisierte Lösung die Entwicklung eines spezialisierten Rahmenprogrammes für den jeweiligen Anwendungsfall effektiver ist als die Verwendung der Desktop-Anwendung.

Mit dem entwickelten Framework lassen sich viele Aufgaben, die sich mit der Zusammenstellung von Informationen aus verschiedenen Quellen in verschiedene Ausgabeformate beschäftigen, in einer strukturierten und stabilen Umgebung lösen. Dabei ist die Entwicklung robuster Plug-Ins jedoch unablässig, da ein Großteil der Kontrolle und damit Verantwortung des reibungslosen Ablaufs an die Plug-Ins übergeht.

Problematisch war vor allem die komplette Abkapselung der Eingabedaten von den Kompositions-Plug-Ins. Dies konnte nicht zufriedenstellend gelöst werden, ist jedoch in Anbetracht der zumeist gleichzeitig entwickelten Dokumenten- und Kompositions-Plug-Ins nur eine geringfügige Einschränkung der Funktionalität.

Schlussendlich lässt sich sagen, dass die Anforderungen an das Basisprogramm und an die Kundenanwendung in zufriedenstellendem Maße erfüllt wurden. Einschränkungen mussten vor allem beim Komfort der Verwaltung von Dokumenten und Plug-Ins gemacht werden, sowie bei der Modularität, die nur scheinbar eine komplett flexible Umgebung realisiert.

Jedoch könnnen das Framework und die Desktop-Anwendung durch Weiterentwicklung deutlich verbessert werden. Das entwickelte System stellt dafür alle benötigten Schnittstellen bereit und kann in diesem Zustand, wie an der Kundenanwendung erkennbar, bereits produktiv eingesetzt werden.

A Verzeichnisse

Quellen

[3St] 3Steps Software. Merge Word Files. Dokumentation. http://www.3stepsoftware.com/merge-word-files.html.

[Apaa] Apache Software Foundation. Apache log4net Features. Dokumentation. http://logging.apache.org/log4net/release/features.html.

[Apab] Apache Software Foundation. Apache poi. Dokumentation. http://poi.apache.org/index.html.

[BCS11] C. Michael Pilato Ben Collins-Sussman, Brian W. Fitzpatrick. Version Control with Subversion, 2011. For Version 1.7 (compiled from r4304).

[Cha08] David Chappell. What is application lifecycle management, Dezember 2008. http://www.microsoft.com/global/applicationplatform/en/us/RenderingAssets/Whitepapers/What%20is%20Application%20Lifecycle%20Management.pdf.

[Deu] Deutsches Institut für Normung e. V. DIN V 19233: Leittechnik - Prozessautomatisierung - Automatisierung mit Prozessrechensystemen.

[Hub08] Thomas Claudius Huber. Windows Presentation Foundation, 2008. 1. Auflage.

[IP96] Jocelyn Ireson-Paine. What is a rule-based system?, Februar 1996. http://www.j-paine.org/students/lectures/lect3/node5.html.

[ISO96] ISO/IEC. 7498-1: Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model. Standard, Juli 1996.

[Küh09] Andreas Kühnel. Visual C# 2008, 2009. 4. Auflage, 1. korrigierter Nachdruck.

[Klu02] Seebold Elmar Kluge, Friedrich. Etymologisches Wörterbuch der deutschen Sprache, 2002. 24. durchges. und erw. Auflage.

[Kos04] Andreas Kosch. Auswirkungen der Code Access Security, 2004. http://msdn.microsoft.com/de-de/library/cc405437.aspx.

[Lot10] Mladen Lotar. The principles of SOLID Programming. Blog, November 2010. http://inchoo.net/tools-frameworks/the-principles-of-solid-programming/.

[Mar00] Robert C. Martin. Design Principles and Design Patterns, 2000. http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf.

[Mar09] Robert C. Martin. Getting a solid start. Blog, Februar 2009. http://blog.objectmentor.com/articles/2009/02/12/getting-a-solid-start.

[Mica] Microsoft Developer Network. Activator-Klasse. Code-Referenz. http://msdn.microsoft.com/de-de/library/System.Activator.aspx.

[Micb] Microsoft Developer Network. Anwendungsdomänen. Reference. http://msdn.microsoft.com/de-de/library/ms173138%28v=vs.80%29.

[Micc] Microsoft Developer Network. ConnectionString-Eigenschaft. Code-Referenz. http://msdn.microsoft.com/de-de/library/system.data.sqlclient.sqlconnection.connectionstring%28v=vs.80%29.aspx.

[Micd] Microsoft Developer Network. Reflektion. Code-Referenz. http://msdn.microsoft.com/de-de/library/ms173183%28v=vs.80%29.aspx.

[Mic10] Microsoft Developer Network. Type Class. Code-Referenz, Dezember 2010. http://msdn.microsoft.com/en-us/library/system.type.aspx.

[Mil] Jeremy Miller. Cohesion and Loose Coupling. Tutorial. http://msdn.microsoft.com/en-us/magazine/cc947917.aspx.

[Ngu] Binh Nguyen. Modular Programming. http://www.tldp.org/LDP/Linux-Dictionary/html/m.html.

[Sob] Sobolsotf. MS Word Join Multiple Documents Software. Dokumentation. http://www.sobolsoft.com/wordjoin/.

[The10] Thomas Theis. Einstieg in Visual C# 2010, 2010. 1. Auflage, 1. korrigierter Nachdruck.

[web12] webmasterpro. Verbreitung von Office-Software bei Internetnutzern in Deutschland im Januar 2010. Statistik, Januar 2012. http://de.statista.com/statistik/daten/studie/77226/umfrage/internetnutzer---verbreitung-von-office-software-in-deutschland/.

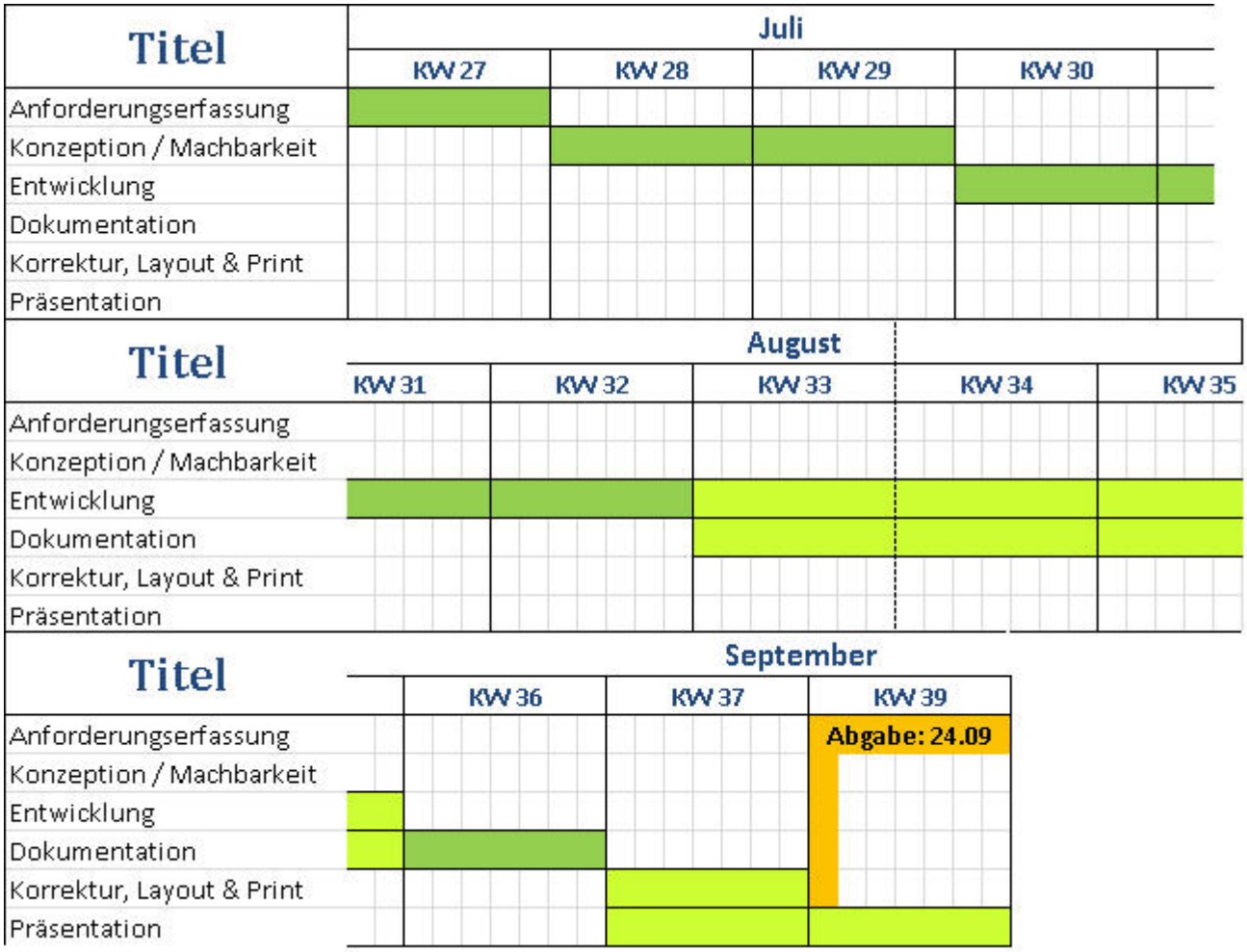
[wor] How to combine multiple Word Documents into one Document. Blog. http://www.gaebler.com/How-to-Combine-Multiple-Word-Documents-Into-One-Document.htm.

[wor08] How to combine multiple Word Documents. Blog, Oktober 2008. http://www.pcworld.com/article/152620/how_to_combine_multiple_word_documents.html.

Abbildungsverzeichnis

Listings

B Projektplan



C Kundenanforderungen

TANNER

Requirements Specification
DocumentComposer Requirements

for the company

Created by
Florian Peschka
TANNER AG
Lindau

Publication date: 17.09.2012 11:39 AM CEST
Document State: Final
Page count: 9
Delivery to customer: 17.09.2012

DocumentComposer Requirements

Summarizes all user requirements for the DocumentComposer project

Part of

Topic	Title	State	Category
Small Project 46685	DocumentComposer	In Progress	

1 Glossary

Requirement 46686	*State:* Approved	*Priority:* -
Version: 1.2		*Changed at:* Aug 20, 2012 11:03 AM
Category: Glossary		*External ID:* -

Excel
: The Microsoft Excel program in the 2003 version

Word
: The Microsoft Word program in the 2003 version

Document
: A document that was created and saved using Word

Composition
: The abstract structure that combines the input documents in a new document

Sheet
: A document that was created and saved using Excel

Table
: A table inside a sheet

Cell
: A cell inside a table

Column
: A complete column inside a table

Row
: A complete row inside a table

PDF
: A portadle document format file

2 Environment

Requirement 46689	*State:* Approved	*Priority:* -
Version: 1.1		*Changed at:* Sep 14, 2012 11:02 AM
Category: Non Functional-Understandability		*External ID:* -

The program should be implemented inside Excel and the .NET-Environment in version 3.5

Fit criterion The user can create the output document from a control inside the composition sheet or inside excel

Rationale The usability needs to be in the focus. The users will not accept a complicated program that has to be configured in a different location than the composition sheet

3 Document composition

Requirement 46690	*State:* Approved	*Priority:* -
Version: 1.1		*Changed at:* Sep 14, 2012 11:02 AM
Category: Functional		*External ID:* -

The program should compose a single output document from a selection of several input documents

Fit criterion After creating the output document, a single document is placed on a defined location.

Rationale Overhead of having multiple documents for a single documentation purpose is extremely high. By providing only one document, the user is more likely to accept, understand and use the documentation for its purposes

3.1 Input documents

Requirement 46692	*State:* Approved	*Priority:* -
Version: 1.2		*Changed at:* Sep 10, 2012 11:53 AM
Category: Functional		*External ID:* -

The input documents are the smallest possible structure the program has to deal with. Every input document should not be smaller than a single chapter or subchapter. A single document must be able to also contain more than only one chapter or only one subchapter (see also example attachment).

Fit criterion When composing the output document, all input documents are treated as single chapters inside the output document. The position of an input document in the output document is determined by the composition table.

Rationale While it would be possible to treat one input document as a collection of several chapters, the input documents are already arranged chapter-wise. To keep the developing easy to understand, every input document is treated as one chapter. This conforms with the existing document structure of the input documents. .Every chapter can contain more subchapters.

3.1.1 Localization of input documents

Requirement 46694	*State:* Approved	*Priority:* -
Version: 1.1		*Changed at:* Sep 10, 2012 11:53 AM
Category: Functional		*External ID:* -

The program must choose the correct localization for the project. This is configured in the composition sheet. If a document is not available in the selected language, the user has to be notified about it.

Fit criterion When selecting a language localization for the output document, the input documents are selected depending on their localization attribute. If a document is not available in the selected language, the program produces an error and lets the user select an action:

- Use the default language instead
- Abort the process

Rationale Documentation has to be produced in different languages. To provide an easy way to compose a localized documentation and still maintain a good overview in the composition table, the program should deal with localization just by defining the language once.

3.2 Output document

Requirement 46693	*State:* Approved	*Priority:* -
Version: 1.2		*Changed at:* Sep 10, 2012 11:53 AM
Category: Functional		*External ID:* -

The output document has to contain all input documents in the order defined by the composition sheet.

Fit criterion All selected documents are composed into one output document.

Rationale To make specific versions of the same documentation or a documentation for a differant variant of a product, the output document has to contain different chapters from the same input document base For every documentation, this process is easier when the chapters that are used for one documentation are selected dynamically.

3.2.1 Optional PDF composition

Requirement 46691	*State:* Approved	*Priority:* -
Version: 1.0		*Changed at:* Sep 14, 2012 11:12 AM
Category: Functional		*External ID:* -

The program should provide an optional function to export the final document into a PDF

Fit criterion

The user gets to choose wether he wants to create PDF version of the output document upon starting the composition process.

- When he chooses yes, a PDF is generated at the same location as the output document, containing the exact same content as the output document does
- When he chooses no, only the output document is generated

Rationale

As PDF is a more portable format than DOC, it's easier to ship it. The output document will in most cases be transfered to PDF directly, yet to provide the outmost flexibility for the user, the decision to do so has to remain on his side

3.2.2 Composition sheet

Requirement 46695	*State:* Approved	*Priority:* -
Version: 1.3		*Changed at:* Sep 10, 2012 11:53 AM
Category: Functional		*External ID:* -

The composition sheet defines how the input documents are combined into the output document. Therefore, the following information is provided inside a table within the sheet:

- Project specific settings
- Customer name
- Project number
- Input folder where the documents are
- Output path where to put the output document
- The name of the output file (defaults to yyyy-mm-dd_HH-MM-SS)
- The path to the folder where the specific document is placed (either as a root path like "C:..." or a relative path starting at the input folder path)
- Composition information
- The filename of that particular chapter's document including the extension
- A mark wether the chapter is to be combined into the output document

Any other information added is merely information needed by the user.

Fit criterion The composition sheet provides all information defined above. The input documents that are marked (and only these) are nested in the corresponding level into the output document.

Rationale The documentation for different parts may be different depending on what variant of a product is documented. To ensure the outmost flexibility, the user has to select how the final documentation is combined in the output document.

3.2.2.1 Configuration sheet

Requirement 46698	*State:* Approved	*Priority:* -
Version: 1.3		*Changed at:* Sep 10, 2012 11:53 AM
Category: Functional		*External ID:* -

Additionally to the basic information, the composition sheet needs to provide meta information the program should respect:

- A default directory where all input documents reside
- The default directory to put the output document

Fit criterion The configuration sheet provides all information defined above. The composition sheet uses the information above as default value for the corresponding fields.

3.2.2.2 Program configuration sheet

Requirement 47172	*State:* Approved	*Priority:* -
Version: 1.0		*Changed at:* Sep 10, 2012 11:53 AM
Category: Functional		*External ID:* -

A configuration sheet for the program that includes

- Path to the executable program
- The column definitons where the program can find the composition information and configuration variables

Fit criterion The program configuration sheet provides all information above. The program reads all values correctly and is fully operational. If some information is not provided, a readable and informational error message informs the user.

Rationale To make the program as flexible as possible, all values have to be changeable by the user, if he needs to do so.

3.2.3 Images

Requirement 46700	*State:* Approved	*Priority:* -
Version: 1.2		*Changed at:* Sep 14, 2012 11:09 AM
Category: Functional		*External ID:* -

The output document has to maintain any images that are also in the input document.

Fit criterion The output document contains all images that the input documents have contained.

3.2.4 References

Requirement 47180	*State:* Approved	*Priority:* -
Version: 1.1		*Changed at:* Sep 10, 2012 11:53 AM
Category: Functional		*External ID:* -

The output document has to maintain any references from input documents to each other and convert them into chapter references that link inside the output document.

Fit criterion When an input document has a reference to another input document and both of them are composed into the output document, the reference is updated to link to another chapter inside the same document rather than to a different document. If the referenced document isn't composed into the output document, the reference produces an error in the output document.

Any errors in the references have to produce an error in the result message after the composition is finished.

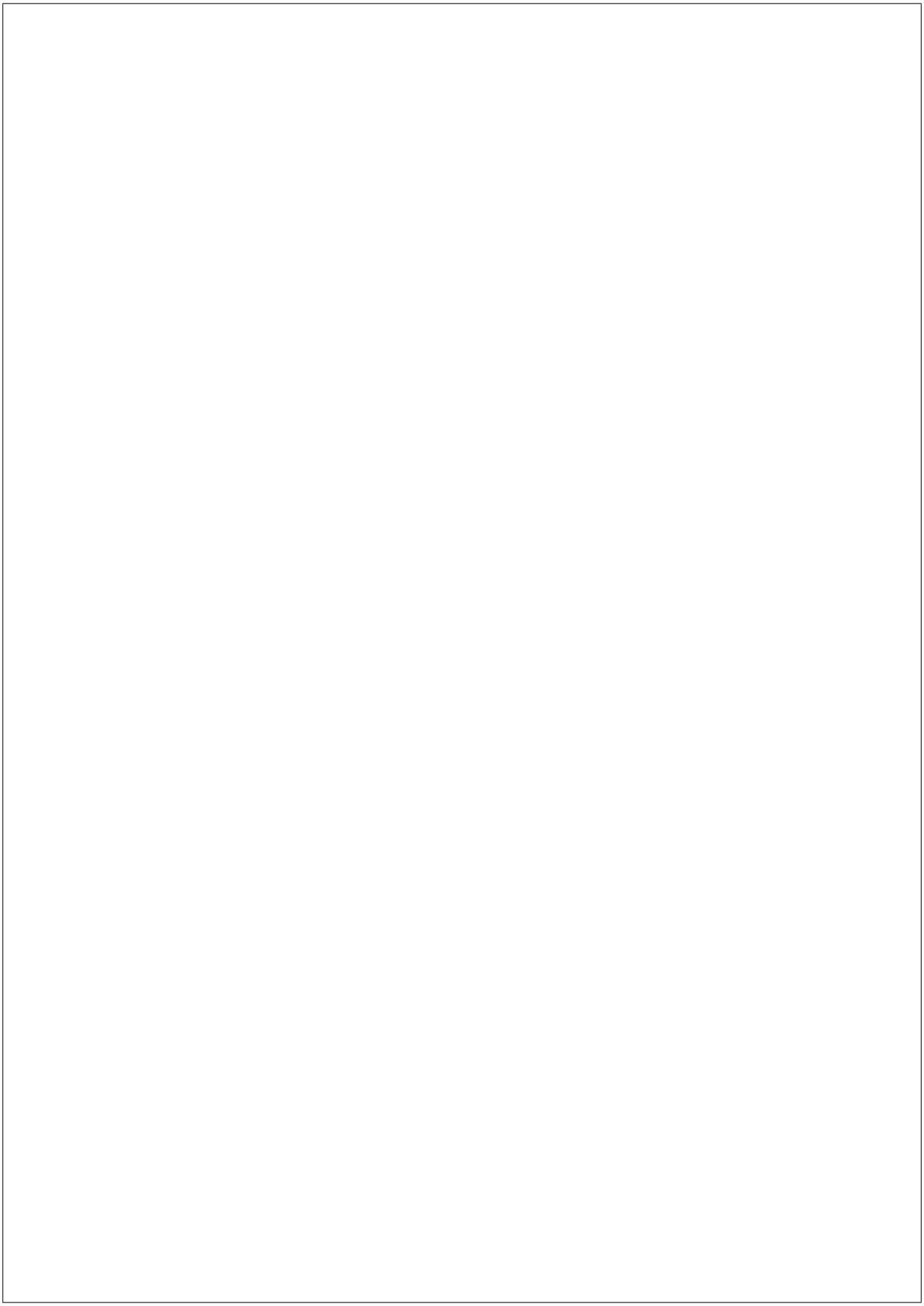
Rationale For a customer reading the documentation, it is vital that the references are valid, so he can easily navigate throughout the document. Errors in the references have to be resolved by the user manually, because there ought to be none.

D Code-Referenz

Docom

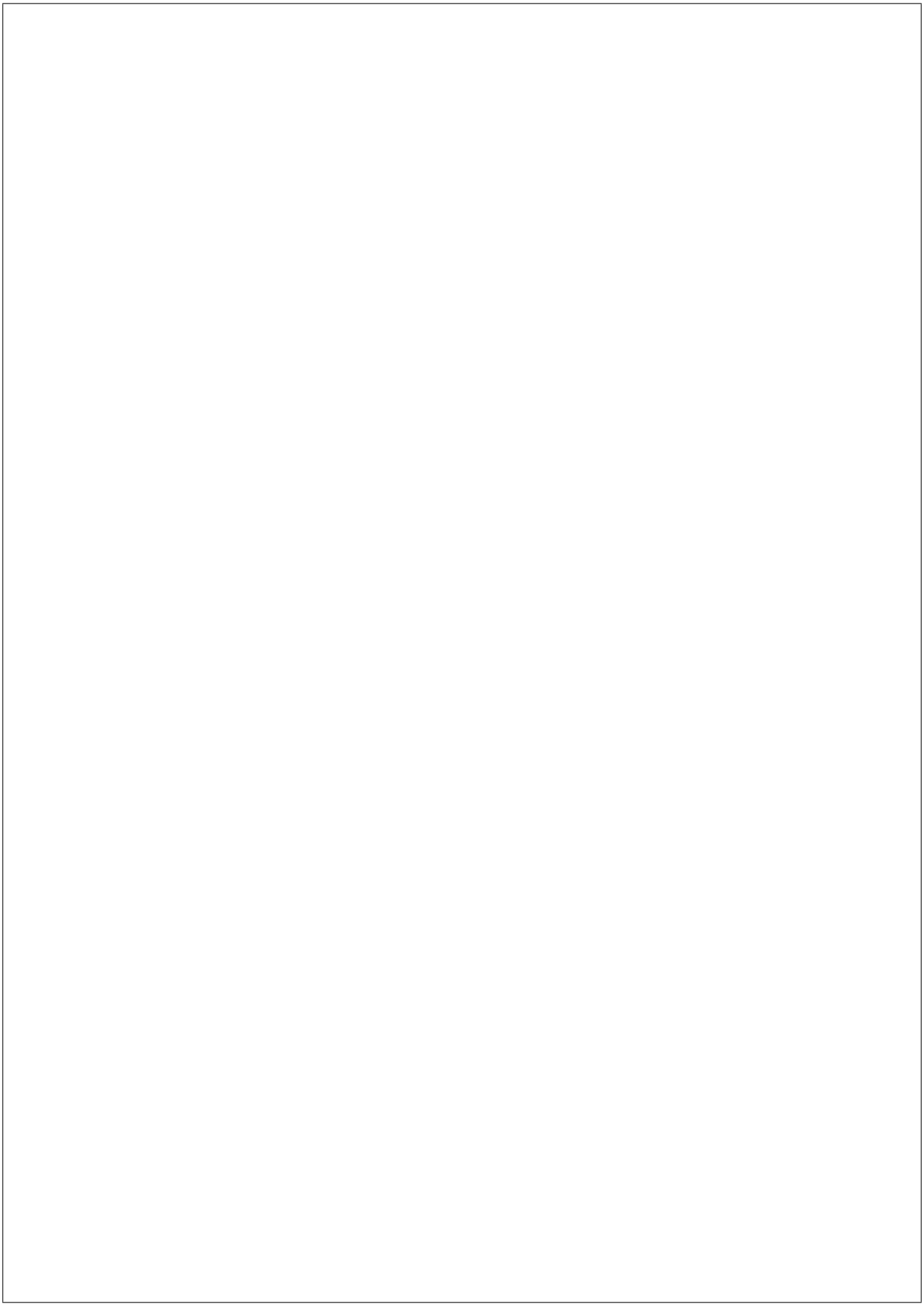
Generated by Doxygen 1.8.2

Mon Sep 17 2012 11:36:04



Contents

Generated on Mon Sep 17 2012 11:36:03 for Docom by Doxygen



Chapter 1

Namespace Index

1.1 Packages

Here are the packages with brief descriptions (if available):

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

Chapter 5

Namespace Documentation

5.1 Package Tanner

Namespaces

- package **Docom**

5.2 Package Tanner.Docom

Namespaces

- package **Model**

Classes

- class **Framework**

  *The basic framework when working with **Docom** (p. 9). Provides methods to load and retrieve plugins so they can be used by your program.*
- class **CompositionPlugin**

  *An abstract interface for plugins that deal with accepting several **Document** (p. 12) objects and compose them into a collection of **ResultBlock** (p. 19) objects that can be used to create a result composition*
- class **DocumentPlugin**

  *An abstract interface for plugins that deal with accepting sources and convert them into **Document** (p. 12) objects*
- class **Document**

  A representation of a single document that contains information
- class **DocumentBlock**

  A block that encapsulates a portion of a document's contents
- class **ResultBlock**

  *A representation of a block of a specific type that is either derived from a **DocumentBlock** (p. 14) or an empty block that servers structuring purposes*

Enumerations

- enum **BlockType** { **FOLDER**, **FILE**, **BLOCK** }

  A type that defines how a Block should be rendered

5.2.1 Enumeration Type Documentation

5.2.1.1 enum Tanner.Docom.BlockType

A type that defines how a Block should be rendered

Enumerator:

FOLDER The block represents a folder, having a name but no futher content other than a collection of child blocks

FILE The block represents a single file that has a name and can contain several child blocks or directly maps to a single **DocumentBlock** (p. 14)

BLOCK A block that contains content from a **DocumentBlock** (p. 14) and has a name

5.3 Package Tanner.Docom.Model

Classes

- class **SourceType**

  Represents the type of a source for document information, e.g. a specific programmable Type or a file extension

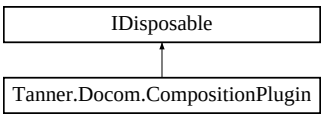
Chapter 6

Class Documentation

6.1 Tanner.Docom.CompositionPlugin Class Reference

An abstract interface for plugins that deal with accepting several **Document** (p. 12) objects and compose them into a collection of **ResultBlock** (p. 19) objects that can be used to create a result composition

Inheritance diagram for Tanner.Docom.CompositionPlugin:



Public Member Functions

- **CompositionPlugin** ()

  Creates a new composition plugin with an empty list of ResultBlocks
- abstract void **Load** (params **Document**[] documents)

  Loads the specified Documents and transfers them into a collection of ResultBlocks
- abstract void **ExecuteComposition** (params string[] args)

  Executes a composition, based on the Blocks-property of this plugin using the specified parameters
- abstract void **Dispose** ()

  Disposes all resources the plugin may have used during its lifetime

Properties

- IList< **ResultBlock** > **Blocks** `[get, set]`

  A list of ResultBlocks that this plugin has composed and can use for executing the composition

6.1.1 Detailed Description

An abstract interface for plugins that deal with accepting several **Document** (p. 12) objects and compose them into a collection of **ResultBlock** (p. 19) objects that can be used to create a result composition

6.1.2 Constructor & Destructor Documentation

6.1.2.1 Tanner.Docom.CompositionPlugin.CompositionPlugin ( )

Creates a new composition plugin with an empty list of ResultBlocks

6.1.3 Member Function Documentation

6.1.3.1 abstract void Tanner.Docom.CompositionPlugin.Dispose ( ) [pure virtual]

Disposes all resources the plugin may have used during its lifetime

6.1.3.2 abstract void Tanner.Docom.CompositionPlugin.ExecuteComposition (params string[] *args*) [pure virtual]

Executes a composition, based on the Blocks-property of this plugin using the specified parameters

Parameters

args	Optional plugin parameters to execute the composition

6.1.3.3 abstract void Tanner.Docom.CompositionPlugin.Load (params **Document**[] *documents*) [pure virtual]

Loads the specified Documents and transfers them into a collection of ResultBlocks

Parameters

documents	The documents to use

6.1.4 Property Documentation

6.1.4.1 IList<**ResultBlock**> Tanner.Docom.CompositionPlugin.Blocks [get],[set]

A list of ResultBlocks that this plugin has composed and can use for executing the composition

The documentation for this class was generated from the following file:

- **CompositionPlugin.cs**

6.2 Tanner.Docom.Document Class Reference

A representation of a single document that contains information

Public Member Functions

- **Document** (String name)

  Creates a new document having the specified name
- **Document AddBlock** (**DocumentBlock** block)

  *Adds the specified **DocumentBlock** (p. 14) to this document*
- override String **ToString** ()

  Returns the name of this document and the amount of DocumentBlocks in it

Properties

- String **Name** [get, set]
  The name of the document
- IList< **DocumentBlock** > **Blocks** [get, set]
  A list of all DocumentBlocks of this document

6.2.1 Detailed Description

A representation of a single document that contains information

6.2.2 Constructor & Destructor Documentation

6.2.2.1 Tanner.Docom.Document.Document (String *name*)

Creates a new document having the specified name

Parameters

name	The name of the document

6.2.3 Member Function Documentation

6.2.3.1 Document Tanner.Docom.Document.AddBlock (DocumentBlock *block*)

Adds the specified **DocumentBlock** (p. 14) to this document

Parameters

block	The block to add to this document

Returns

This Documents instance

6.2.3.2 override String Tanner.Docom.Document.ToString ( )

Returns the name of this document and the amount of DocumentBlocks in it

Returns

The name of this document and the amount of DocumentBlocks in it

6.2.4 Property Documentation

6.2.4.1 IList<DocumentBlock> Tanner.Docom.Document.Blocks [get],[set]

A list of all DocumentBlocks of this document

6.2.4.2 String Tanner.Docom.Document.Name [get],[set]

The name of the document

The documentation for this class was generated from the following file:

- **Document.cs**

6.3 Tanner.Docom.DocumentBlock Class Reference

A block that encapsulates a portion of a document's contents

Public Member Functions

- **DocumentBlock** (**Document** document, string name)

  *Creates a new **DocumentBlock** (p. 14) that is part of the document and has the given name*
- override string **ToString** ()

  Returns the name of this block along with a string representation of its contents

Properties

- object **Content** `[get, set]`

  The content of the document block. This is an arbitrary object, the plugins that deal with it have to decide which Type it resolves to
- **Document Document** `[get, set]`

  *The **Document** (p. 12) this block belongs to*
- string **Name** `[get, set]`

  The name of this block. May be unique, but doesn't have to be

6.3.1 Detailed Description

A block that encapsulates a portion of a document's contents

6.3.2 Constructor & Destructor Documentation

6.3.2.1 Tanner.Docom.DocumentBlock.DocumentBlock (Document *document*, string *name*)

Creates a new **DocumentBlock** (p. 14) that is part of the *document* and has the given *name*

Parameters

document	The **Document** (p. 12) this block is part of
name	The name of this block

6.3.3 Member Function Documentation

6.3.3.1 override string Tanner.Docom.DocumentBlock.ToString ( )

Returns the name of this block along with a string representation of its contents

Returns

The name of this block along with a string representation of its content

6.3.4 Property Documentation

6.3.4.1 object Tanner.Docom.DocumentBlock.Content [get],[set]

The content of the document block. This is an arbitrary object, the plugins that deal with it have to decide which Type it resolves to

6.3.4.2 Document Tanner.Docom.DocumentBlock.Document [get],[set]

The **Document** (p. 12) this block belongs to

6.3.4.3 string Tanner.Docom.DocumentBlock.Name [get],[set]

The name of this block. May be unique, but doesn't have to be

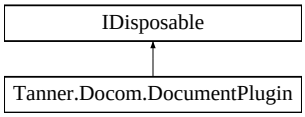
The documentation for this class was generated from the following file:

- **DocumentBlock.cs**

6.4 Tanner.Docom.DocumentPlugin Class Reference

An abstract interface for plugins that deal with accepting sources and convert them into **Document** (p. 12) objects

Inheritance diagram for Tanner.Docom.DocumentPlugin:



Public Member Functions

- abstract **Document ParseInput** (object input)
  *Parses an object and transfers its content into a **Document** (p. 12)*
- abstract bool **CanHandle** (**SourceType** type)
  Determines wether this plugin can handle a source of the SourceType type
- abstract void **Dispose** ()
  Disposes all resources the plugin may have used during its lifetime

6.4.1 Detailed Description

An abstract interface for plugins that deal with accepting sources and convert them into **Document** (p. 12) objects

6.4.2 Member Function Documentation

6.4.2.1 abstract bool Tanner.Docom.DocumentPlugin.CanHandle (SourceType *type*) [pure virtual]

Determines wether this plugin can handle a source of the SourceType type

Parameters

type	The source type to check

Returns

Wether this plugin can handle this type of source

6.4.2.2 abstract void Tanner.Docom.DocumentPlugin.Dispose ( ) [pure virtual]

Disposes all resources the plugin may have used during its lifetime

6.4.2.3 abstract Document Tanner.Docom.DocumentPlugin.ParseInput (object *input*) [pure virtual]

Parses an object and transfers its content into a **Document** (p. 12)

Parameters

input	The object to parse

Returns

The parsed **Document** (p. 12) object

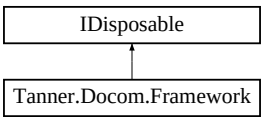
The documentation for this class was generated from the following file:

- **DocumentPlugin.cs**

6.5 Tanner.Docom.Framework Class Reference

The basic framework when working with **Docom** (p. 9). Provides methods to load and retrieve plugins so they can be used by your program.

Inheritance diagram for Tanner.Docom.Framework:



Public Member Functions

- **Framework** ()

  Creates a new instance of the Docom-Framework
- void **RegisterAssembly** (string pluginFile)

  Registers an assembly to the framework. All plugins that reside in that assembly are loaded as the respective plugins they represent.
- **DocumentPlugin GetPluginFor** (String extension)

  Gets a plugin that can handle the specified file extension
- **DocumentPlugin GetPluginFor** (**SourceType** type)

  Tries to get the document plugin that can handle a specific source type
- **DocumentPlugin GetDocumentPlugin** (Type ofType)

  Tries to get a document plugin that is of the given type and currently loaded in the framework
- **CompositionPlugin GetCompositionPlugin** (Type ofType)

  Tries to get a composition plugin that is of the given type and currently loaded in the framework

- void **Dispose** ()

  Dispose all resources that the plugins and the framework have used. Should always be called before shutting down the application that uses the framework

Properties

- IList< **DocumentPlugin** > **DocumentPlugins** `[get]`

  All currently loaded document plugins
- IList< **CompositionPlugin** > **CompositionPlugins** `[get]`

  All currently loaded composition plugins

6.5.1 Detailed Description

The basic framework when working with **Docom** (p. 9). Provides methods to load and retrieve plugins so they can be used by your program.

6.5.2 Constructor & Destructor Documentation

6.5.2.1 Tanner.Docom.Framework.Framework ( )

Creates a new instance of the Docom-Framework

6.5.3 Member Function Documentation

6.5.3.1 void Tanner.Docom.Framework.Dispose ( )

Dispose all resources that the plugins and the framework have used. Should always be called before shutting down the application that uses the framework

6.5.3.2 CompositionPlugin Tanner.Docom.Framework.GetCompositionPlugin (Type *ofType*)

Tries to get a composition plugin that is of the given type and currently loaded in the framework

Parameters

ofType	The type of the plugin to get

Returns

A plugin that is of the given type

6.5.3.3 DocumentPlugin Tanner.Docom.Framework.GetDocumentPlugin (Type *ofType*)

Tries to get a document plugin that is of the given type and currently loaded in the framework

Parameters

ofType	The type of the plugin to get

Returns

A plugin that is of the given type

6.5.3.4 DocumentPlugin Tanner.Docom.Framework.GetPluginFor (String *extension*)

Gets a plugin that can handle the specified file extension

Parameters

extension	The file extension to check

Returns

A document plugin that can handle this extension

6.5.3.5 DocumentPlugin Tanner.Docom.Framework.GetPluginFor (SourceType *type*)

Tries to get the document plugin that can handle a specific source type

Parameters

type	The source type to check for

Returns

The document plugin for that source type

6.5.3.6 void Tanner.Docom.Framework.RegisterAssembly (string *pluginFile*)

Registers an assembly to the framework. All plugins that reside in that assembly are loaded as the respective plugins they represent.

Parameters

pluginFile	The full path to the assembly file

6.5.4 Property Documentation

6.5.4.1 IList<CompositionPlugin> Tanner.Docom.Framework.CompositionPlugins [get]

All currently loaded composition plugins

6.5.4.2 IList<DocumentPlugin> Tanner.Docom.Framework.DocumentPlugins [get]

All currently loaded document plugins

The documentation for this class was generated from the following file:

- **Framework.cs**

6.6 Tanner.Docom.ResultBlock Class Reference

A representation of a block of a specific type that is either derived from a **DocumentBlock** (p. 14) or an empty block that servers structuring purposes

Public Member Functions

- **ResultBlock** (**BlockType** type, String name)

  *Creates a new **ResultBlock** (p. 19) of the BlockType type and with the specified name*
- **ResultBlock** (**BlockType** type, **DocumentBlock** block)

  *Creates a new **ResultBlock** (p. 19) of the BlockType type corresponding to the specified **DocumentBlock** (p. 14)*
- **ResultBlock** (**BlockType** type, **DocumentBlock** block, String name)

  *Creates a new **ResultBlock** (p. 19) of the BlockType type, corresponding to a **DocumentBlock** (p. 14) with the specified name*
- override String **ToString** ()

  Returns the type of this block, its name and the amount of child blocks in it
- override Boolean **Equals** (object obj)

  Determines if this block is equal to the specified block
- override int **GetHashCode** ()

  *Returns a hash code that identifies this instance of **ResultBlock** (p. 19)*

Properties

- String **Name** `[get, set]`

  *The name of this **ResultBlock** (p. 19)*
- IList< **ResultBlock** > **ChildBlocks** `[get, set]`

  A list of all child blocks
- **BlockType Type** `[get, set]`

  The BlockType of this block
- **DocumentBlock DocumentBlock** `[get, set]`

  *The corresponding **DocumentBlock** (p. 14) that this **ResultBlock** (p. 19) was created from*

6.6.1 Detailed Description

A representation of a block of a specific type that is either derived from a **DocumentBlock** (p. 14) or an empty block that servers structuring purposes

6.6.2 Constructor & Destructor Documentation

6.6.2.1 Tanner.Docom.ResultBlock.ResultBlock (BlockType *type*, String *name*)

Creates a new **ResultBlock** (p. 19) of the BlockType type and with the specified name

Parameters

type	The type of the new block
name	The name of the new block

6.6.2.2 Tanner.Docom.ResultBlock.ResultBlock (BlockType *type*, DocumentBlock *block*)

Creates a new **ResultBlock** (p. 19) of the BlockType type corresponding to the specified **DocumentBlock** (p. 14)

Parameters

type	The type of the new block
block	The **DocumentBlock** (p. 14) the new **ResultBlock** (p. 19) corresponds to

6.6.2.3 Tanner.Docom.ResultBlock.ResultBlock (BlockType *type*, DocumentBlock *block*, String *name*)

Creates a new **ResultBlock** (p. 19) of the BlockType type, corresponding to a **DocumentBlock** (p. 14) with the specified name

Parameters

type	The type of the new block
block	The **DocumentBlock** (p. 14) the new **ResultBlock** (p. 19) corresponds to
name	The name of the new block

6.6.3 Member Function Documentation

6.6.3.1 override Boolean Tanner.Docom.ResultBlock.Equals (object *obj*)

Determines if this block is equal to the specified block

Parameters

obj	The block to check

Returns

Wether this block is equal to *obj*

6.6.3.2 override int Tanner.Docom.ResultBlock.GetHashCode ( )

Returns a hash code that identifies this instance of **ResultBlock** (p. 19)

Returns

A hash code that identifies this instance of **ResultBlock** (p. 19)

6.6.3.3 override String Tanner.Docom.ResultBlock.ToString ( )

Returns the type of this block, its name and the amount of child blocks in it

Returns

The type of this block, its name and the amount of child blocks in it

6.6.4 Property Documentation

6.6.4.1 IList<ResultBlock> Tanner.Docom.ResultBlock.ChildBlocks [get],[set]

A list of all child blocks

6.6.4.2 DocumentBlock Tanner.Docom.ResultBlock.DocumentBlock [get],[set]

The corresponding **DocumentBlock** (p. 14) that this **ResultBlock** (p. 19) was created from

6.6.4.3 String Tanner.Docom.ResultBlock.Name [get],[set]

The name of this **ResultBlock** (p. 19)

6.6.4.4 BlockType Tanner.Docom.ResultBlock.Type [get],[set]

The BlockType of this block

The documentation for this class was generated from the following file:

- **ResultBlock.cs**

6.7 Tanner.Docom.Model.SourceType Class Reference

Represents the type of a source for document information, e.g. a specific programmable Type or a file extension

Public Member Functions

- **SourceType** (string type)
  Creates a new source type from the specified string
- override string **ToString** ()
  Returns the string representation of the source type
- bool **Equals** (string type)
  Determines if this source type is equal to the specified one
- override bool **Equals** (object type)
  Determines if this source type is equal to the specified one
- bool **Equals** (**SourceType** type)
  Determines if this source type is equal to the specified one
- override int **GetHashCode** ()
  *Returns a hash code that identifies this instance of **SourceType** (p. 21)*

Properties

- string **Type** [get, set]
  The string representation of this source type

6.7.1 Detailed Description

Represents the type of a source for document information, e.g. a specific programmable Type or a file extension

6.7.2 Constructor & Destructor Documentation

6.7.2.1 Tanner.Docom.Model.SourceType.SourceType (string *type*)

Creates a new source type from the specified string

Parameters

type	The type to create

6.7.3 Member Function Documentation

6.7.3.1 bool Tanner.Docom.Model.SourceType.Equals (string *type*)

Determines if this source type is equal to the specified one

Parameters

type	The source type to check

Returns

Wether this source type is equal to *type*

6.7.3.2 override bool Tanner.Docom.Model.SourceType.Equals (object *type*)

Determines if this source type is equal to the specified one

Parameters

type	The source type to check

Returns

Wether this source type is equal to *type*

6.7.3.3 bool Tanner.Docom.Model.SourceType.Equals (SourceType *type*)

Determines if this source type is equal to the specified one

Parameters

type	The source type to check

Returns

Wether this source type is equal to *type*

6.7.3.4 override int Tanner.Docom.Model.SourceType.GetHashCode ( )

Returns a hash code that identifies this instance of **SourceType** (p. 21)

Returns

A hash code that identifies this instance of **SourceType** (p. 21)

6.7.3.5 override string Tanner.Docom.Model.SourceType.ToString ( )

Returns the string representation of the source type

Returns

The string representation of the source type

6.7.4 Property Documentation

6.7.4.1 string Tanner.Docom.Model.SourceType.Type [get],[set]

The string representation of this source type

The documentation for this class was generated from the following file:

- **SourceType.cs**

Chapter 7

File Documentation

7.1 AssemblyInfo.cs File Reference

7.2 BlockType.cs File Reference

Namespaces

- package **Tanner.Docom**

Enumerations

- enum **Tanner.Docom.BlockType** { **Tanner.Docom.FOLDER**, **Tanner.Docom.FILE**, **Tanner.Docom.BLOCK** }

  A type that defines how a Block should be rendered

7.3 CompositionPlugin.cs File Reference

Classes

- class **Tanner.Docom.CompositionPlugin**

  *An abstract interface for plugins that deal with accepting several **Document** (p. 12) objects and compose them into a collection of **ResultBlock** (p. 19) objects that can be used to create a result composition*

Namespaces

- package **Tanner.Docom**

7.4 Document.cs File Reference

Classes

- class **Tanner.Docom.Document**

  A representation of a single document that contains information

Namespaces

- package **Tanner.Docom**

7.5 DocumentBlock.cs File Reference

Classes

- class **Tanner.Docom.DocumentBlock**

  A block that encapsulates a portion of a document's contents

Namespaces

- package **Tanner.Docom**

7.6 DocumentPlugin.cs File Reference

Classes

- class **Tanner.Docom.DocumentPlugin**

  An abstract interface for plugins that deal with accepting sources and convert them into ***Document*** (p. 12) *objects*

Namespaces

- package **Tanner.Docom**

7.7 Framework.cs File Reference

Classes

- class **Tanner.Docom.Framework**

  The basic framework when working with ***Docom*** (p. 9)*. Provides methods to load and retrieve plugins so they can be used by your program.*

Namespaces

- package **Tanner.Docom**

7.8 ResultBlock.cs File Reference

Classes

- class **Tanner.Docom.ResultBlock**

  A representation of a block of a specific type that is either derived from a ***DocumentBlock*** (p. 14) *or an empty block that servers structuring purposes*

Namespaces

- package **Tanner.Docom**

7.9 SourceType.cs File Reference

Classes

- class **Tanner.Docom.Model.SourceType**

  Represents the type of a source for document information, e.g. a specific programmable Type or a file extension

Namespaces

- package **Tanner.Docom.Model**