|  |
| --- |
| Elmer Lukas, Heidt Christina, Treichler Delia  12. Dezember 2011 |

|  |
| --- |
| Studienarbeit |
| Entwurf |
|  |

****

# Dokumentinformationen

## Änderungsgeschichte

|  |  |  |  |
| --- | --- | --- | --- |
| Datum | Version | Änderung | Autor |
| 25.09.2011 | 1.0 | Erste Version des Dokuments | lelmer |
| 03.10.2011 | 1.1 | Review | dtreichl |
| 12.12.2011 | 1.2 | Schichtenarchitektur dokumentiert | lelmer |

## Inhaltsverzeichnis

[1 Dokumentinformationen 1](#_Toc311469272)

[1.1 Änderungsgeschichte 1](#_Toc311469273)

[1.2 Inhaltsverzeichnis 1](#_Toc311469274)

[2 Design Entscheide 1](#_Toc311469275)

[2.1 PDF auf Surface 2 Darstellen 1](#_Toc311469276)

[2.1.1 Varianten 2](#_Toc311469277)

[2.1.2 Nutzwertanalyse 2](#_Toc311469278)

[3 Architektur 2](#_Toc311469279)

[3.1 Interfaces 5](#_Toc311469280)

[3.1.1 Converter.Interfaces 5](#_Toc311469281)

[3.1.2 Services.Interfaces 6](#_Toc311469282)

[3.1.3 Loader.Interfaces 6](#_Toc311469283)

[3.2 Implementation 7](#_Toc311469284)

[3.3 Services.Loader 7](#_Toc311469285)

[3.4 Converter.Pdf 7](#_Toc311469286)

[3.5 Preparer 8](#_Toc311469287)

[3.6 Services 10](#_Toc311469288)

[3.7 UserInterface.Surface 11](#_Toc311469289)

[3.8 ProjectFlip 12](#_Toc311469290)

[4 Patterns 12](#_Toc311469291)

[4.1 MVVM 12](#_Toc311469292)

[4.2 Flyweight 12](#_Toc311469293)

[4.3 Dependency Injection mit Unity 12](#_Toc311469294)

# Design Entscheide

## PDF auf Surface 2 Darstellen

Eine sehr wichtige Aufgabe der Software ist es, PDF Dokumente auf dem Surface 2 darzustellen. Dazu gibt es verschiedene Möglichkeiten, die genauer analysiert wurden. Zur Entscheidung wurde schliesslich eine Nutzwertanalyse durchgeführt.

### Varianten

#### Variante 1: PDF direkt darstellen

Die naheliegende Lösung ist natürlich, das PDF direkt auf dem Surface darzustellen. Leider unterstützt das Surface 2 Framework diese Darstellung nicht direkt; es müsste noch eine PDF Library zur Darstellung zur Laufzeit geladen werden. Dadurch wird die Laufzeitumgebung von einer zusätzlichen Library abhängig, was die Installation erschwert und vielleicht gar nicht möglich ist (kann zurzeit nicht genauer beurteilt werden, da die zu testende Hardware noch fehlt).

#### Variante 2: Umwandlung zu XPS

Das Surface 2 SDK erlaubt es, ohne weitere Libraries XPS Dokumente anzuzeigen (DocumentViewer, <http://msdn.microsoft.com/en-us/library/system.windows.controls.documentviewer.aspx>). Da ein XPS Dokument ähnlich wie ein PDF Dokument ist, können Textinhalte als Text gespeichert werden, wodurch beliebig weit herangezoomt werden kann.

#### Variante 3: Umwandlung zu Bild

Die einfachste Möglichkeit, um ein PDF auf dem Surface 2 darzustellen, ist eine Umwandlung des PDF Dokuments in ein Bild (z.B. PNG). Natürlich können Bilder ohne Probleme auf dem Surface 2 dargestellt werden, und auch die Umwandlung eines PDFs zu einem Bild ist mit Open Source Libraries problemlos möglich (z.B. mit Image Magick oder GhostScript/GhostPDF/GhostXPS). Der grosse Nachteil ist, dass die Bilder sehr gross sein würden, da der Text als Bild dargestellt werden müsste. Dadurch wird auch der maximale Zoom für die Bilder festgelegt.

### Nutzwertanalyse

In der Nutzwertanalyse geht die Variante 2: Umwandlung zu XPS als knapper Sieger vor der Variante 3: Umwandlung zu Bild hervor.

Weitere Details zur Nutzwertanalyse sind dem Dokument NutzwertanalysePDFAufSurface.xlsx zu entnehmen.

# Architektur

Bei der Architektur wurde darauf geachtet, dass die einzelnen Komponenten möglichst unabhängig sind und so einfach ausgetauscht werden können.

Zum Beweis der Schichtenarchitektur wurde im Visual Studio 2010 ein Dependency Diagramm erstellt:

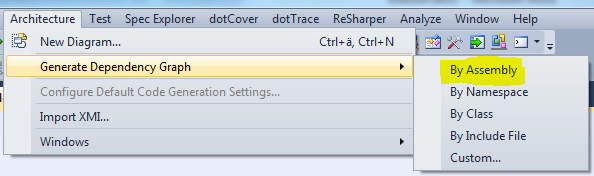


Abbildung - Generierung Dependency Graph

In einem zweiten Schritt wurden dann die „Generics“ und die „Externals“ entfernt, damit das Ganze übersichtlicher wird. Anschliessend wurden die Tests und Interfaces gruppiert und eingefärbt, damit das Diagramm übersichtlicher wird. Das sieht dann folgendermassen aus:

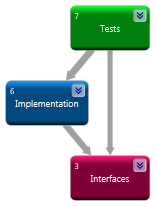


Abbildung - Übersicht Abhängigkeiten

Auf der folgenden Seite sind weitere Details zu den Abhängigkeiten ersichtlich:

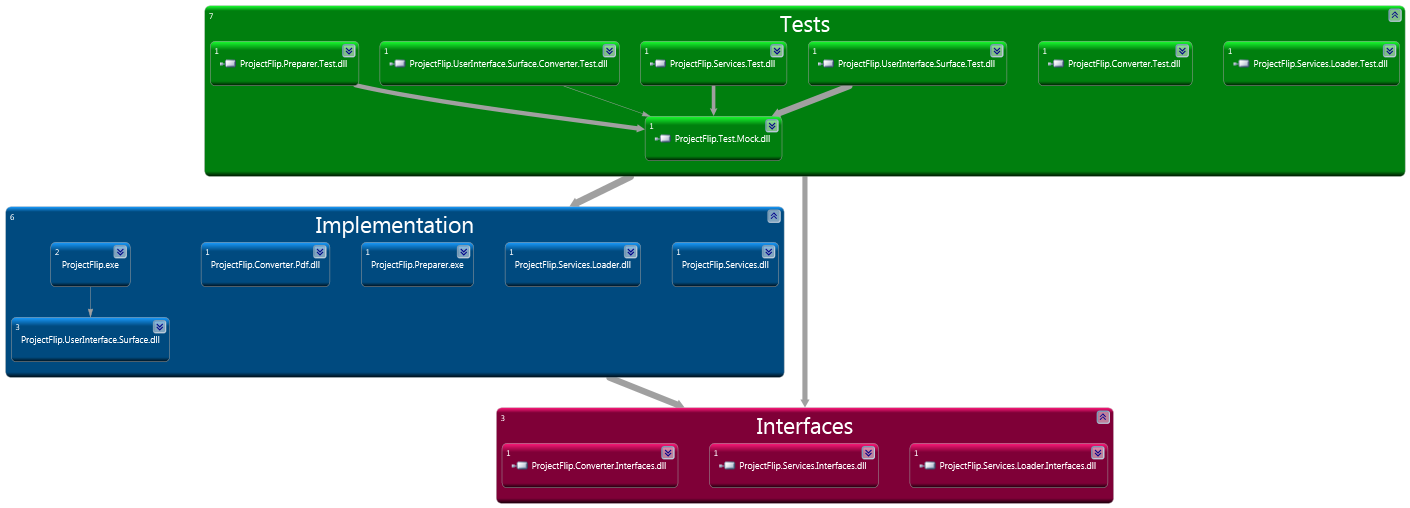
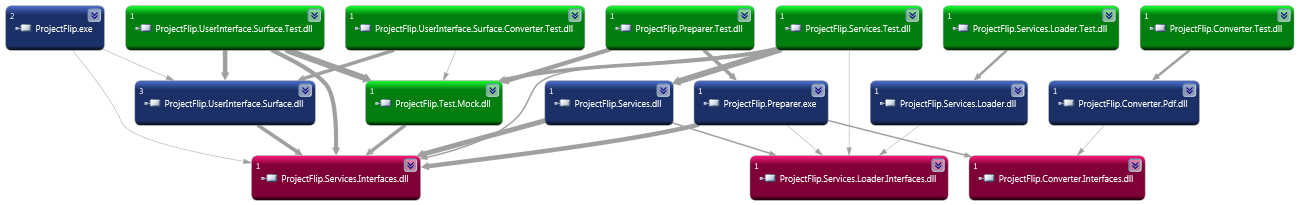


Abbildung - Übersicht Abhängigkeiten mit Details

Abbildung - Details Abhängigkeiten

Um auch noch mit einem alternativen Tool die Schichtung zu beweisen, anbei ein Diagramm, das von NDepend generiert wurde:

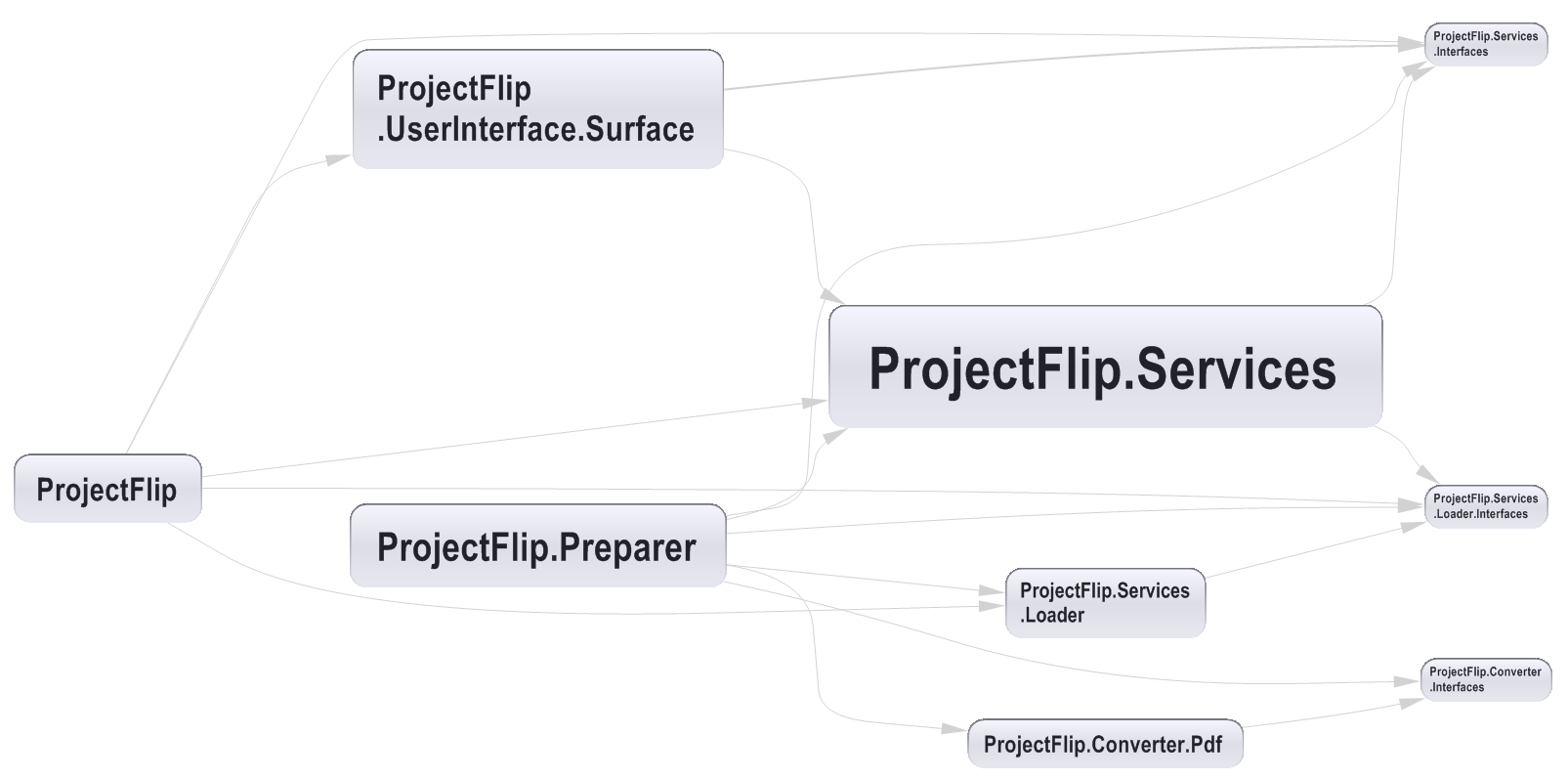


Abbildung - NDepend Dependency Graph

Ganz links befindet sich das Startup Projekt, dann folgen das User Interface und der Preparer, dann die Services, Loader und Converter und ganz rechts schliesslich noch die Interfaces.

Nun werden die einzelnen Projekte genauer beschrieben. Die Test Projekte werden nicht weiter dokumentiert, da ihre Namen schon aussagen, welche Projekte sie testen.

## Interfaces

Um die Implementation untereinander zu entkoppeln, wurden Interfaces eingesetzt:

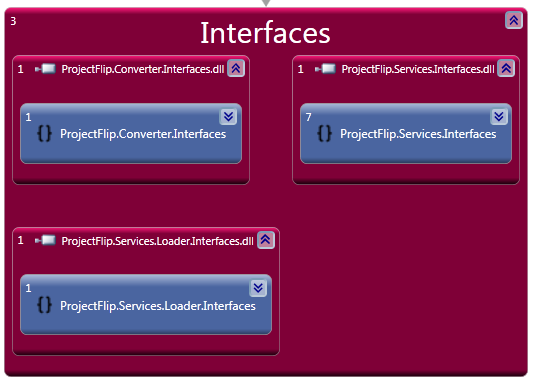


Abbildung - Übersicht Interfaces

### Converter.Interfaces

IConverter: Dieses Interface wird für die Konvertierung vom PDF Dokumenten in XPS Dokumente benutzt. So ist es einfach, diese Implementation auszutauschen. Dies wird in Zukunft, sobald mehr solche Konverter gratis zur Verfügung stehen und sich das XPS Dateiformat mehr durchgesetzt hat, auch gut möglich sein.

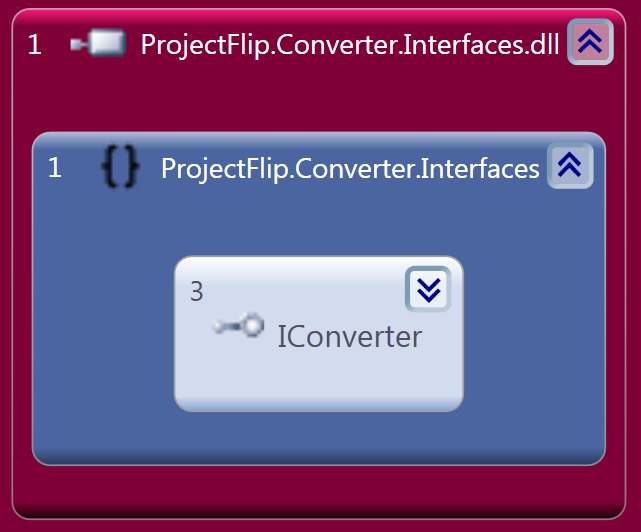


Abbildung - Converter.Interfaces

### Services.Interfaces

IAggregator, IMetadata, IMetadataType: Diese Interfaces sind für die Metadaten, die an den ProjectNotes hängen, verantwortlich. Es werden als Beispiel Dinge wie „C++“ als IMetadata und „Technologie“ als IMedatataType gespeichert. Der IAggregator ist für das Mapping verantwortlich, also wenn z.B. „JavaBeans“ nach „Java“ gemappt werden soll.

Der ICultureHelper ist dafür verantwortlich, dass eine Sprache registriert werden kann. Dies wird zur Anzeige eines XPS benötigt.

Die ICyclicCollectionView<T> ist eine zyklische Liste, auf der gefiltert werden kann. Es implementiert das INotifyPropertyChanged, damit sie die involvierten Komponenten benachrichtigen kann.

Die IProjectNote stellt eine ProjectNote dar. Neben den Properties (z.B. Title) stellt sie auch eine Preload Methode zur Verfügung, um das XPS Dokument mittels Eager Loading vorzuladen.

IProjectNoteService stellt den Service zur Verfügung. Es bietet zwei Property Getters – einer für die IProjectNotes und einer für die IMetadatas.

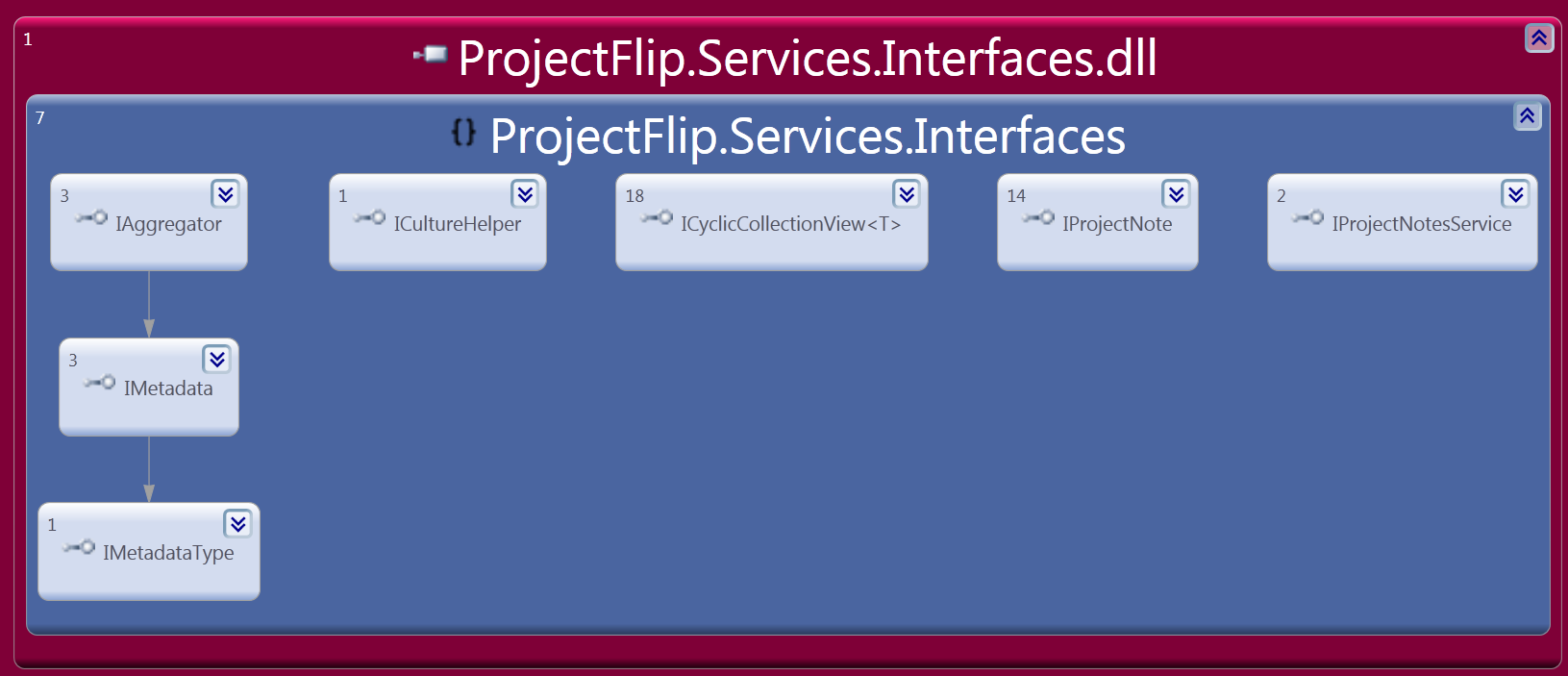


Abbildung - Services.Interfaces

### Loader.Interfaces

Der IProjectNotesLoader stellt das Interface zur Verfügung, um IProjectNotes zu laden. Dies ist deshalb in einem separaten Assembly, damit der Preparer und die eigentliche Applikation dieses gemeinsame Interface implementieren können.

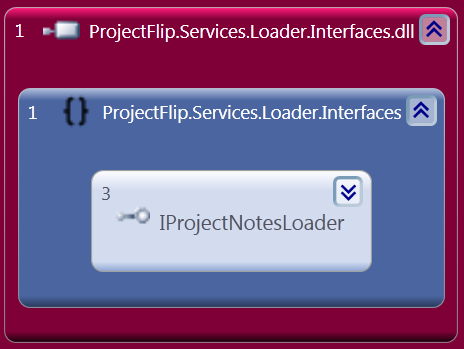


Abbildung - Loader.Interfaces

## Implementation

Die Implementation ist natürlich von den Interfaces abhängig. Dadurch können die Implementationen voneinander entkoppelt werden. Hier die Übersicht:

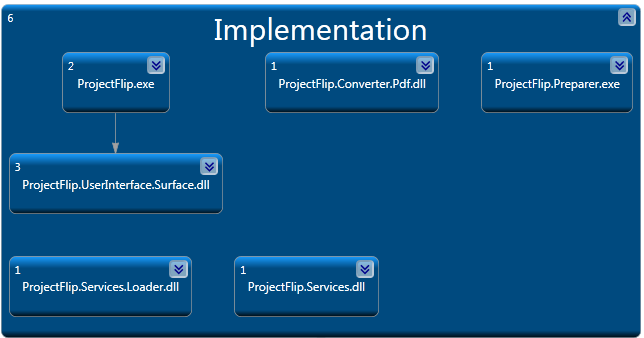


Abbildung - Implementation Übersicht

## Services.Loader

Der ProjectNotesLoader kann eine Liste von IProjectNotes laden. In der aktuellen Version werden die Daten von einem Tabulator getrenntem Textdokument gelesen, in Zukunft kann das aber auch von einer Datenbank oder vom SharePoint geladen werden.

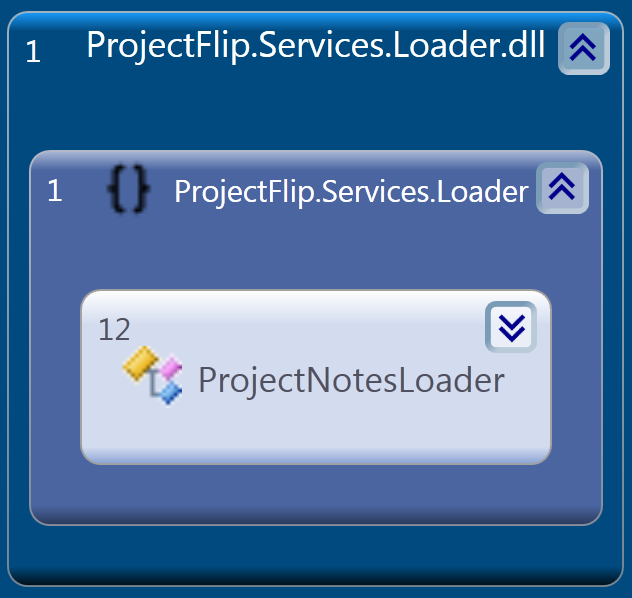


Abbildung - Services.Loader

## Converter.Pdf

Der PdfConverter ist dafür verantwortlich, das PDF Dokument in ein XPS Dokument zu konvertieren.

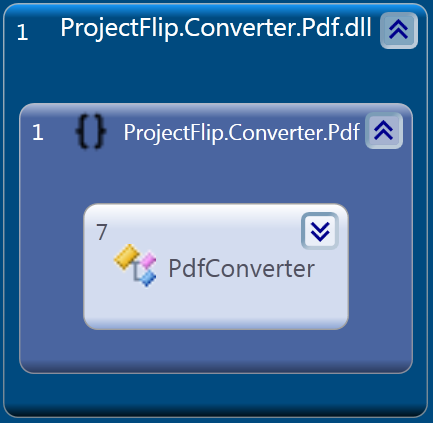


Abbildung - Converter.Pdf

## Preparer

Der Preparer ist dafür verantwortlich, dass die ProjectNotes der Applikation im entsprechenden Format zur Verfügung steht. Konkret heisst das pro Project Note: ein XPS Dokument und ein Bild (243x243 Pixel gross). Dazu verwendet der Preparer einen IProjectNotesLoader und einen IConverter.

Da der Preparer vor der Applikation und unabhängig ausgeführt wird, ist er als Executable (exe) kompiliert.

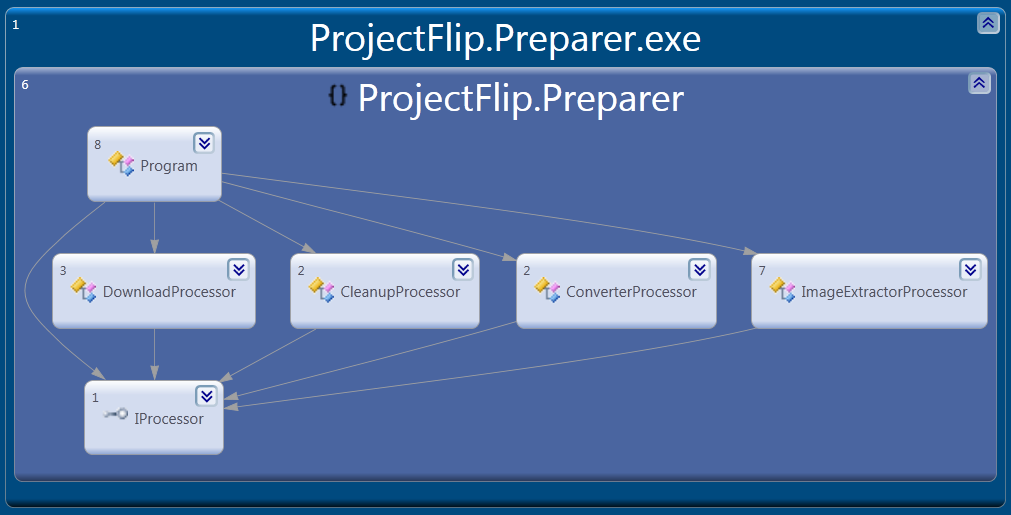


Abbildung - Preparer

Beim Preparer sind die einzelnen IProcessors wichtig. Als erstes wird der DownloadProcessor verwendet, der das PDF Dokument herunterlädt und lokal speichert. In einem zweiten Schritt wird der ConverterProcessor verwendet, der das PDF in ein XPS Dokument konvertiert. Der ImageExtractorProcessor extrahiert schliesslich noch ein Bild aus dem XPS Dokument. Sollte etwas schief gegangen sein und sich die Dateien in einem fehlerhaften Zustand befinden, räumt der CleanupProcessor schliesslich noch auf.

Um die Ausführung zu beschleunigen, wurde das Verarbeiten einzelner Project Notes parallelisiert. Das folgende Sequenzdiagramm zeigt den Ablauf auf (in der Liste „processors“ sind Instanzen der einzelnen IProcessors enthalten):



Abbildung - Sequenzdiagramm Preparer

## Services

Der Service ist dafür verantwortlich, die IProjectNotes und die aggregierten IMetadata Objekte zur Verfügung zu stellen. So erhält er zuerst die Rohdaten vom IProjectNotesLoader und baut dann aufgrund der Eingaben die Objektstruktur mit den IProjectNotes und IMetadatas auf. Die Metadaten werden gleich beim Import aggregiert. Weiter implementiert das Service Projekt bestimmte Hilfsklassen, wie zum Beispiel den MetadataComparer.

Damit nach Project Notes gefiltert werden kann, stellt der Service eine CyclicCollectionView<T> zur Verfügung. Diese implementiert das Interface ICyclicCollectionView<T>.

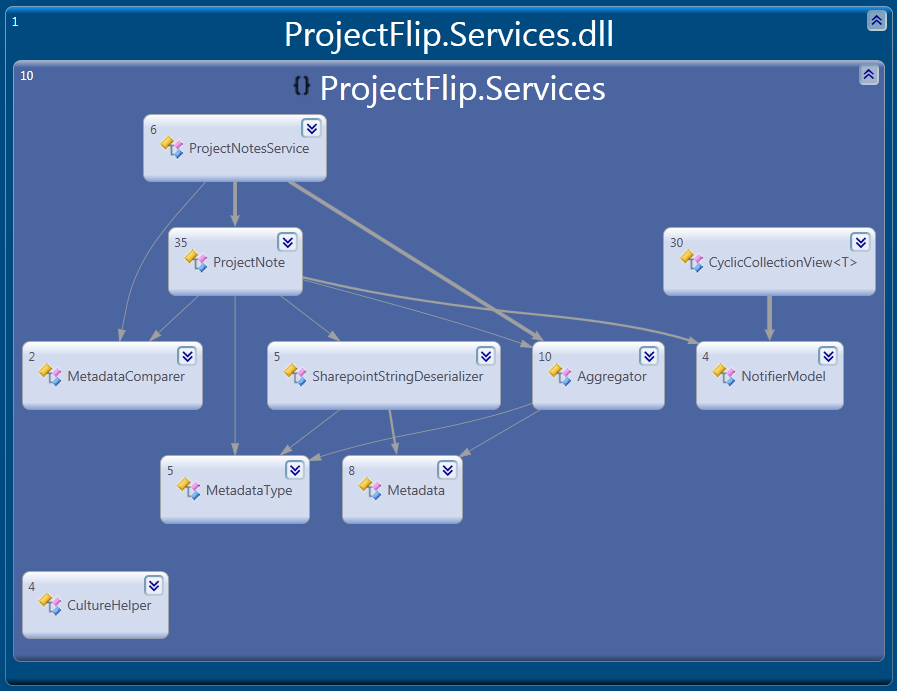


Abbildung – Services

## UserInterface.Surface

Das UserInterface.Surface Assembly ist für das GUI verantwortlich. Es benutzt zur Entkopplung der View und des Models das MVVM Pattern (siehe 4.1 MVVM).

Das OverviewWindowViewModel ist dafür verantwortlich, die Commands der View entgegenzunehmen, eine Aktion auszuführen (Bsp. einen Filter setzen), und dann die View zu notifizieren (Bsp. mithilfe von INotifyPropertyChanged).

Für die ICommand Implementation wird die Klasse Command implementiert, welche dann vom OverviewWindowViewModel benutzt wird.

Das OverviewWindow stellt das Hauptfenster dar, während die Klassen AboutView, DetailView, FilterDetailView, FilterView, Gravatar, ZoomInIcon und ZoomOutIcon einzelne UserControls darstellen, die vom OverviewWindow benutzt werden.

Das ScrollToTopBehaviour wird benötigt, damit bei einer Änderung der Elemente im Container automatisch nach oben gescrollt wird.

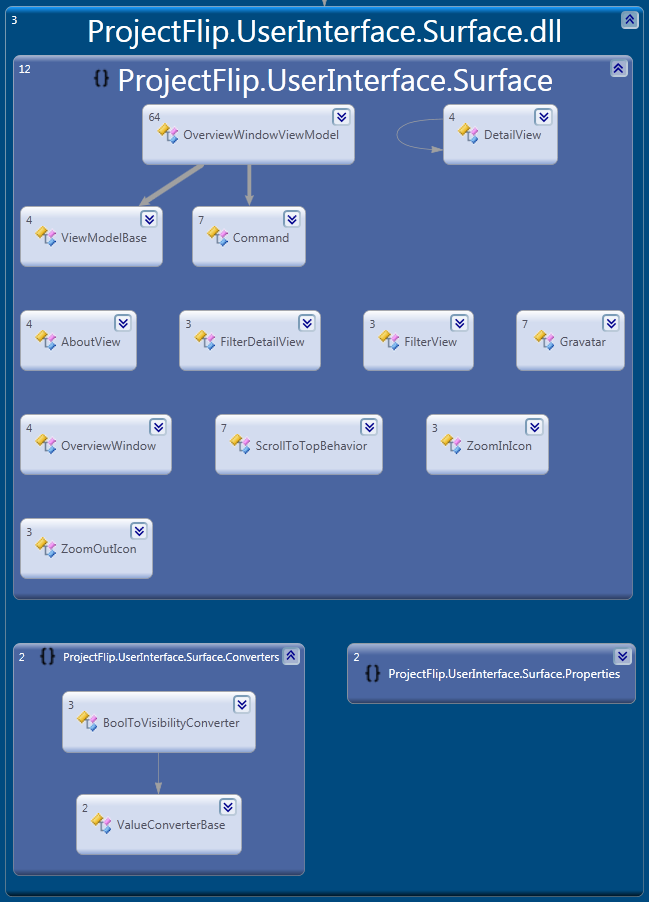


Abbildung - UserInterface.Surface

## ProjectFlip

Dieses Projekt ist das Startup Projekt. Es ist dafür verantwortlich, die richtigen Komponenten zusammenzufügen und schlussendlich das GUI zu starten. Hier wurde stark Dependency Injection eingesetzt, und zwar mit Unity (von Microsoft). So kann an einem Ort konfiguriert werden, welche Komponenten instanziiert werden sollen.

Über die Settings kann die Applikation schliesslich konfiguriert werden.

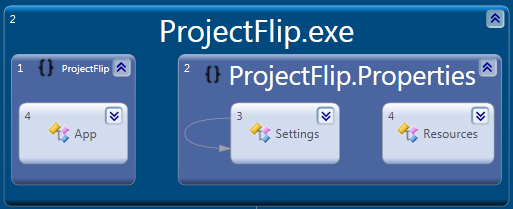


Abbildung - ProjectFlip

# Patterns

Die drei wichtigsten Patterns, die wir angewendet haben und hier beschreiben möchten, sind MVVM, Flyweight und Dependency Injection mit Unity.

## MVVM

Das MVVM („Model“, „View“, „ViewModel) Pattern wird benötigt, um die View von dem Model zu entkoppeln. Deshalb wird als Zwischenglied ein ViewModel erzeugt, das die Commands des GUIs abarbeitet und die verfügbaren Elemente dem GUI zur Verfügung stellt.

Die Idee dahinter ist, dass sich das GUI schneller ändert als die Businesslogik und deshalb macht es Sinn, diese zwei Komponenten möglichst stark abzutrennen. Ein weiterer wichtiger Punkt ist, dass das ViewModel mit Unit Tests getestet werden kann.

Weitere Informationen sind unter [<http://msdn.microsoft.com/en-us/magazine/dd419663.aspx>] zu finden.

## Flyweight

Flyweight (Deutsch: Fliegengewicht) wird für die Metadaten (Metadata) eingesetzt. Die Idee hinter dem Flyweight ist, dass viele Objekte andere gleiche Objekte benötigen, zum Beispiel ein Metatag mit dem Namen „Java“. Damit nicht sehr viele Objekte erstellt werden, referenzieren verschiedene ProjectNotes einen gleichen Metatag.

So werden das Starten und auch der Betrieb der Applikation performanter, weil nicht so viele Objekte erstellt werden müssen. Zusätzlich wird auch die Suche effizienter.

Der Nachteil an der Sache ist, dass die Flyweight Objekte immutable Objects, also unveränderlich, sein müssen. Deshalb hat das Metadata Model auch keine Setter.

## Dependency Injection mit Unity

Damit die Komponenten jederzeit und einfach ausgetauscht werden können, wurde mit Unity Containern gearbeitet, um Dependency Injection zu ermöglichen. Für dieses Projekt mag dass ein Overhead sein, der nicht unbedingt nötig wäre, aber speziell in grösseren Projekten ist Dependency Injection sehr wichtig. Denn so können z.B. auch verschiedene Software Versionen ausgeliefert werden, indem die Container ausgetauscht werden.