Prototype of a Fragmented Document Editor

Jenny Jutterström

Theses Report
Basic level

Mälardalen University
School of Innovation, Design and Engineering

Supervisors:
Damir Isovic
Mälardalen University,
Thomas Sandström
Signifikant Svenska AB

8/21/2009
Abstract
Signifikant Svenska AB supplies an information system called Assert, developed to facilitate the aftermarket sales in the manufacturing and subcontractor domains. The information system offers companies and organizations the possibility to gather their product information in a joint database in order to increase the information availability and distribution.

The management of the documents is an important part of the customer need and can be improved in order to also support document maintenance directly in Assert. At the moment, users only have the possibility to add and view documents in the database. By also providing users the possibility to create documents, update document contents, effectively reuse document sections and ease the translation of documents within Assert, the document management will be better facilitated.

The purpose of this thesis is to develop a prototype which shows the concept and benefits when providing the possibility to share document contents between several documents. The prototype is developed in C#/WPF and provides a word processor with features to reuse document contents and translation management.
# Table of Contents

Introduction ......................................................................................................................... 5  
Problem Background ............................................................................................................. 5  
Purpose and delimitations ..................................................................................................... 5  
Method ................................................................................................................................ 6  
Analysis of Problem ............................................................................................................ 6  
Customer Interview .............................................................................................................. 6  
Reuse Document Contents ..................................................................................................... 6  
Keep Translations up to Date ............................................................................................... 6  
Use Cases ............................................................................................................................. 7  
Objective Formulation .......................................................................................................... 8  
Solution ................................................................................................................................. 9  
Implementation Background .................................................................................................. 9  
  Windows Presentation Foundation ......................................................................................... 9  
  XAML .................................................................................................................................. 9  
  Control Templates and styles .............................................................................................. 10  
  Data Binding ...................................................................................................................... 10  
System Design Overview ......................................................................................................... 11  
Word Processor Control ......................................................................................................... 11  
  Functionality ...................................................................................................................... 11  
  Design .................................................................................................................................. 11  
  Implementation .................................................................................................................. 14  
Document Fragment Management ............................................................................................ 15  
  Functionality ...................................................................................................................... 15  
  Design .................................................................................................................................. 15  
  Implementation .................................................................................................................. 17  
Translation Management ........................................................................................................ 18  
  Functionality ...................................................................................................................... 18  
  Design .................................................................................................................................. 18  
  Implementation .................................................................................................................. 19  
Result .................................................................................................................................... 20  
Application Overview ............................................................................................................ 20  
Analysis of Result .................................................................................................................. 25
Introduction
This chapter provides a brief introduction to the thesis domain with information about the background, motivation, purpose and the used method.

Problem Background
Signifikant Svenska AB (Signifikant) supplies an information system called Assert, developed to facilitate the aftermarket sales in the manufacturing and subcontractor domains. The information system offers companies and organizations the possibility to gather their product information in a joint database in order to increase the information availability and distribution. The gathered product information includes document types such as service information, spare part catalogs with CAD drawings, educational documents and product manuals.

The management of the documents is an important part of the customer need and can be improved in order to also support document maintenance directly in Assert. At the moment, users only have the possibility to add and view documents in the database. By also providing users the possibility to create documents, update document contents, effectively reuse document sections and ease the translation of documents within Assert, the document management will be better facilitated.

Purpose and delimitations
Signifikant wishes to investigate how to handle and structure documents in order to effectively maintain the document contents. Signifikant has a solution proposal where the document is defined by an editable template, in which the content is divided in a number of parts which can be used in several documents. A concise description of the structure in the concept is showed in Figure 1. The purpose of this thesis is to focus on the smallest document elements which have the actual text contents, herewith called document fragments, and their structure and interaction. Important issues related to the document fragments is the ability to support the translation process, remain consistency of shared contents and to preserve a coherent look in different documents. The objective of the thesis is to develop a document fragment manager component, together with a graphical user interface. The component will be used in a larger system and because of the limited time it will only be a prototype showing the suggested concept.

![Diagram](image)

Figure 1. The documents contain a number of pages. Each page can be provided with containers where a document fragment can be placed. One document fragment can be used in several pages and documents.
Method
Assert is a client/server solution developed in C# and uses Microsoft’s .Net-framework. The prototype will therefore be implemented in C# using the .Net 3.5-framework. The main information source about the techniques will be the Internet where many articles are provided. The application will be developed in Microsoft Visual Studio, together with Microsoft Expression Blend as a help to enhance the graphical user interface.

The requirements of the application will be defined as use cases based on the customer needs and requirements decided with the company. The graphical user interface will be developed based on concept sketches and the functionality will be implemented in an iteration model.

Analysis of Problem
This chapter provides the problem background from the customer perspective and describes the defined use cases. The use cases resulted in a summarized objective of the prototype to develop, described in the last section.

Customer Interview
In order to get a better understanding of the customer need, a meeting was conducted with a potential customer regarding how they maintain their documents locally today. Because of the time limit, the problem domain must be limited and therefore are only the following two areas summarized.

Reuse Document Contents
The interviewed company keeps local copies of all published documents, in which they make their updates before distribution. Many of the documents have sections and pages which are the same as in other documents, for instance, different variants of one product could share up to 50 % of the document contents. When a section is updated which is used in several documents, the changed text is manually copied to every document containing the text.

The update procedure would be more time efficient if there was a possibility to share document contents and thereby automatically update a shared text in all documents where it is used. The update would also guarantee the consistency of the shared contents when it only has to be performed at one place. A suggested work procedure is like in Cad, where components are gathered in a common database and can be inserted to drawings with different contexts. One concern related to the shared texts is preserving the current text format in the different documents when including a shared text, but inline formats such as text marked as bold should be kept the same in all documents.

Keep Translations up to Date
The interviewed company has copies of their documents in every translated language. When one text is updated or added, a check is performed to determine in which languages the text needs to be translated. For example, if the updated text is used in several documents, the different documents can have various languages translated which all must be updated. The updated text is then sent as plain text to different places for translation, depending on the target languages. When all translations have been received, the text is manually copied to all documents and the translations are copied to the respective language document.
The text translation would be more efficient if there was a possibility to automatically determine which languages the updated or added text needs to be translated to. The translation updates should also be better facilitated if new translations could be automatically included in the language documents. One suggestion was to be able to view the documents in several languages, but texts that miss translations in the current view should be clearly marked.

**Use Cases**
In order to capture the basic functionalities of the prototype, a use case diagram was modeled with the requirements gathered from the customer interview and the requirements decided with Signifikant. The use case diagram can be seen in Figure 2 and the following section describes each use case.

![Use Case Diagram](https://www.altova.com)

**Figure 2. Use case showing the basic functionalities of the prototype.**

A. **Create Document Sections** - By dividing the contents of a document into smaller parts, called Document Fragments, the document becomes more modular. Therefore, the prototype should offer the possibility to create document fragments which can be used in use case (B).

B. **Reuse Document Sections** – In order to effectively reuse the contents of a document, every document fragment should be easy to include in different documents. The building of a document is handled by an outside document builder and is not included in the prototype domain, but the structure of the document fragments should support the building process.

C. **Apply Text Formatting** – The user should be able to apply basic text formatting options to the text contents of a document fragment, such as text styles and text alignments.

D. **Translate Document Sections** – The document fragments should provide translation ability.

E. **Generate Document Paragraphs** – In order to make the document even more modular, every document part should automatically split its contents into smaller contents fractions. In that way, a document fragment consisting of three paragraphs will be divided to three content elements, called Document Paragraphs. The generated paragraphs should be possible to use only locally or in several document fragments by use case (F).
F. **Reuse Document Paragraphs** – The generated Document Paragraphs should be possible to include in other document fragments. Inline formatting information, such as text marked as bold, should be saved with the Document Paragraph in order to be consistent when the Document Paragraph is used in different places.

G. **Export Document Sections to Translation** – In order to make it easy to handle the text in a document fragment and a provided translation, the translation should be handled by using XML. Many translation companies offer to possibility to receive translations in XML format, which also facilitates to keep inline formatting information from the source text to the translation. Therefore, the prototype should be able to generate a XML file with the document fragments to translate.

H. **Import Translated Document Sections** – The prototype should be able to add translations by parsing a defined XML translation document, generated in use case (G).

**Objective Formulation**

With the defined use cases as a base, the functionality goals of the prototype are described as follows.

The document fragment prototype will provide a word processor where the user can open a document fragment to edit the text contents. When the document fragment is updated and saved, the contained text will automatically be divided in smaller parts, called document paragraphs (see figure 3). The generated document paragraphs represent every paragraph in the document fragment and new paragraphs will be placed in a library. The user can then reuse the available document paragraphs by including them from the library to any document fragment.

The document fragments will be translated through the using of XML. When the user selects a number of document fragments, the user is provided the option to translate their contents to a chosen target language. Document paragraphs in the selected document fragments which miss a translation in the target language will be extracted to a XML file. The translation can then be added manually to the XML file and afterward be imported to the application.
The user will have the opportunity to view a document fragment in an arbitrary language. All the included document paragraphs will then be displayed in the chosen language. Paragraphs which do not contain a translation in the current language view will be displayed in their source language and marked red.

**Solution**

*This chapter describes the design and implementation of the developed prototype. The chapter begins by describing a theoretical background of the implementation.*

**Implementation Background**

The prototype will be implemented in C# and use WPF (Windows Presentation Foundation) for the graphical user interface. This section presents a basic description of the techniques provided by WPF.

**Windows Presentation Foundation**

Windows Presentation Foundation (WPF) is Microsoft’s latest technology for building user interfaces in Windows based applications. The system was initially released as a part of .Net Framework 3.0 and included with Microsoft Vista, as a way to meet peoples increasing expectations on graphical user interfaces. WPF was therefore designed to make a clear separation between the appearance and behavior of the applications, and thereby provide the possibility to allow designers and developers to focus on separate domains during the development. In order to enable modern user interface features, WPF is built on the game programming API DirectX which uses the graphics hardware. The technique provides the possibility to use interactive 2D and 3D, multimedia, transparency, animations, gradients and much more in the user interfaces. WPF uses vector graphics and since the framework utilizes the graphics hardware acceleration, the user interfaces gets faster, scalable and resolution independent [NL01].

WPF applications can be developed in .Net languages such as C# and Visual Basic, often in combination with the markup language XAML which is described in the next section. Microsoft has also developed a framework called Silverlight which enables a subset of the WPF resources to be used on the web.

**XAML**

XAML (eXtensible Application Markup Language) is used in WPF to describe the visual appearance of the user interface and was developed by Microsoft to enhance the separation between the user interface and the business logic. Everything that is created in XAML can be expressed by using common .Net languages, however XAML has many benefits since it is extensively created for user interface development [NL02]. One of these benefits is the possibility for developers and designers to work in parallel, without requiring a common compilation since the two areas are loosely coupled.

XAML is a XML-based language and therefore quite straightforward, see Figure 4 for a code example. Microsoft Visual Studio offers the visual designer WPF Designer to develop user interfaces with XAML, but there exists stand alone products which are focused on user interface development in WPF. Microsoft Expression Blend is one application that is compatible with Visual Studio and especially developed to build user interfaces in WPF. There are also many complementing tools available to process and generate XAML. For instance, Microsoft Expression Design which can
generate XAML code from created vector graphics and Zam3d that provides the possibility to export 3d models to XAML. The generated XAML code can then be included in WPF applications.

```xml
<Window x:Class="WpfApplication.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <StackPanel>
        <TextBlock Text="Some control examples:"></TextBlock>
        <Button Content="Button" />
        <TextBox Text="TextBox" />
        <ComboBox Text="ComboBox">
            <ComboBoxItem Content="ComboBoxItem" />
        </ComboBox>
    </StackPanel>
</Window>
```

Figure 4. XAML code presenting a stack panel with a TextBox, Button and a ComboBox control.

Control Templates and styles
The controls used in WPF are very customizable due to their control templates and control styles. All controls are defined by a control template, which is an element tree that defines the structure of the control. Each control has a default template which is available for all common windows themes and gives the control a basic appearance. However, the control template can easily be changed and contain almost any type of control as contents of another.

The control style is quite similar to Cascading Style Sheets (CSS) in web development. The control style gives the possibility to generally set appearance properties such as color background, font style, margin, etc of the controls. The control style can then be applied on a group of controls in order to not have to set these properties individually for each control when used in different contexts.

Both the control style and template enables features to easily change the visual appearance of the controls when for example, the mouse pointer is placed over a control or when a button is pressed. WPF also provides an option to automatically create animations from one appearance state to another.

Data Binding
One powerful concept in WPF is the flexible data binding which provide a simple and consistent way for WPF applications to present and interact with data. Data binding makes it possible to bind various .Net objects together by creating a source and target relationship. The following four types of data binding relationships exist in the .Net 3.5 framework:

- One time: The target object ignores updates in the source object.
- One way: The target object has a read-only access to the data in the source object.
- Two way: The target object can both read and write data to the source object.
- One way to source: The source object has a read-only access to the data in the target object.

When the data changes its value, elements that are bound to the data with correct settings get notified and reflect the changes automatically. Typically, the data binding relationship consists of four components; a binding target object, a target property, a binding source object and a path to
the value in the binding source to use (see Figure 5). For example, in order to bind the text in a TextBox control to the Name property of a Student business object, the target object is the TextBox, the target property is the Text property of the TextBox, the source object is the Student object and the value path is the property Name in the source object.

![Figure 5. The target-source relationship in WPF data binding.](image)

**System Design Overview**

The application is divided in three different parts; a word processor control, document fragment management and translation handling. The Word Processor control is responsible for managing the text contents of a document fragment. The document fragment management part includes the structure and handling of document fragments and their contents. The translation handling is responsible of the translation procedure. Therefore, the system parts are described in separate sections.

**Word Processor Control**

*This chapter describes the developed Word Processor control, which is a word processor that enables the user to edit the text of a document fragment.*

**Functionality**

The Word Processor control consists of an editable document and supports basic text formatting options to change the text appearance. When a text is selected in the document it can be marked as bold or italic. The document can also include lists, which can be presented as a numbered list or be styled with bullets. The control also provides the option to apply text styles defined by a style template and to include paragraphs in the document from a global library.

**Design**

An UML diagram of the Word Processor control can be seen in Figure 6. The control derives from the WPF RichTextBox control, the control document is a FlowDocument object and the control contains Dependency Properties and functions to manage its document contents. The four control parts are described in the following sections.
Figure 6. UML diagram of the WordProcessor control. The control derives from the RichTextBox control.

Word Processor Document
WPF provides two categories of documents; fixed documents and flow documents. Fixed documents are intended to be used when the document display is fixed, because their contents does not adjust to the current context and will always appear exactly the same. In contrast, flow documents are designed to optimize the readability and therefore automatically adjust their contents based on runtime settings [MS01]. The dynamic adjustments include advantages such as the document font size and appearance can be reformed depending on the current device resolution, print settings will be maximized according to the current printer, et cetera. Therefore, the chosen document type to host the text in the Word Processor control is a FlowDocument.

FlowDocument Structure
FlowDocument is a FrameworkContentElement, which support data binding and other WPF mechanisms. The FlowDocument object accepts Blocks as its children, placed as a BlockCollection in the content property called Blocks. The content of the FlowDocument is defined by the used blocks and the following five different types of Blocks exist;

1. **Paragraph**: Contains a collection of Inlines, which are the actual text contents and are internally placed in the inline element Run.
2. **Section**: Groups several Blocks together.
3. **List**: Contains a collection of ListItems, and a ListItem is a collection of Blocks (for example, a Paragraph or another List).
4. **Table**: Collection of Blocks and elements defining the Tables structure.
5. **BlockUIContainer**: Embeds other WPF controls.
The Word Processor functionality is restricted to text and lists and the other Block types are therefore disregarded. In Figure 7 is a code example with the applicable blocks.

```xml
<FlowDocument>
  <Paragraph>
    <Bold>Some bold text in the paragraph</Bold>
    Some text that is not bold.
  </Paragraph>

  <List>
    <ListItem>
      <Paragraph>ListItem 1</Paragraph>
    </ListItem>
    <ListItem>
      <Paragraph>ListItem 2</Paragraph>
    </ListItem>
  </List>

</FlowDocument>
```

*Figure 7. XAML code presenting a FlowDocument with one Paragraph and one List block.*

**Word Processor Base Class**

WPF provides a couple of container controls which can host a FlowDocument, and most of them are only used for displaying FlowDocuments with built-in functionalities such as zoom, turn document pages and search possibilities. In order to have both view and edit abilities, the available container control is the *RichTextBox*, which stores its content in a Document property of type FlowDocument.

RichTextBox derives from the base class *TextBoxBase*, which has built-in support for features such as Copy, Paste, Undo, Redo and spell checking. However, the RichTextBox also provides additional functionality, such as a possibility to use a TextPointer while traversing the FlowDocument contents and a Selection property to fetch the currently selected FlowDocument text. The TextSelection type has many properties and provides for example the event Changed, which is fired when the selected text range covers a new content. The Word Processor control derives from the RichTextBox in order to make use of the RichTextBox functionality.

**Dependency Properties**

The WordProcessor control contains properties which sets and unsets Boolean text formatting options such as bold, italic, list and alignment alternatives to the selected text. The properties can be changed by an arbitrary WPF control in the user interface by creating data bindings. In order to notify any value changes from the binding source to the binding target, and the opposite, the properties are declared as *Dependency Properties*. Dependency properties are a new type of property introduced by WPF and is different than the common CLR properties. The value of a dependency property depends on providers and is determined by considering these factors at any point in time. These properties are widely used in WPF controls, where the Button control for example has 78 properties which are bindable [NL03]. When the value of a dependency property is changed, WPF automatically trigger a change notification which can be caught and handled. The WordProcessor control contains methods to handle every dependency property change, which updates the text formatting according to the current value status of the triggered property.

The WordProcessor document is set to a dependency property as well, which notifies when the document object is changed in order to set the Document property to the new FlowDocument object.
**Routed Commands**

The WordProcessor contains methods to include a paragraph in the FlowDocument and to apply a style to a selected text. In order to be able to execute these methods from the user interface without using tight coupling, they are defined as *Routed Commands*. The Routed Command implements the ICommand interface, which contains mechanisms to handle the command. The command is routed through the element tree to a specified command handler. This makes it possible to bind various WPF controls, called invokers, to the routed command by only specifying the command handler element in the user interface that is managing the command. For example, when binding a button command to the routed command ApplyStyle, the Command property of the button is set to “WordProcessor.ApplyParagraphStyle”, where WordProcessor is the command handler object fetched in the user interface and ApplyParagraphStyle is the command.

**Implementation**

**Update of Text Format**

The WordProcessor uses Dependency Properties to keep track of the currently applied text format in the document. The Dependency Properties are registered to the WPF property system with information about the property type, property name, its default value, the property owner type, and a callback method which is invoked when the property value has changed. The registered text format Dependency Properties are listed in Table 1.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Default value</th>
<th>Owner type</th>
<th>Change callback</th>
</tr>
</thead>
<tbody>
<tr>
<td>SelectionIsBold</td>
<td>Bool</td>
<td>False</td>
<td>WordProcessor</td>
<td>SelectionIsBoldPropertyChanged</td>
</tr>
<tr>
<td>SelectionIsItalic</td>
<td>Bool</td>
<td>False</td>
<td>WordProcessor</td>
<td>SelectionIsItalicsPropertyChanged</td>
</tr>
<tr>
<td>SelectionIsBullets</td>
<td>Bool</td>
<td>False</td>
<td>WordProcessor</td>
<td>SelectionIsBulletsPropertyChanged</td>
</tr>
<tr>
<td>SelectionIsNumbering</td>
<td>Bool</td>
<td>False</td>
<td>WordProcessor</td>
<td>SelectionIsNumberingPropertyChanged</td>
</tr>
<tr>
<td>SelectionAlignmentLeft</td>
<td>Bool</td>
<td>True</td>
<td>WordProcessor</td>
<td>SelectionAlignmentLeftPropertyChanged</td>
</tr>
<tr>
<td>SelectionAlignmentRight</td>
<td>Bool</td>
<td>False</td>
<td>WordProcessor</td>
<td>SelectionAlignmentRightPropertyChanged</td>
</tr>
<tr>
<td>SelectionAlignmentCenter</td>
<td>Bool</td>
<td>False</td>
<td>WordProcessor</td>
<td>SelectionAlignmentCenterPropertyChanged</td>
</tr>
<tr>
<td>SelectionAlignmentJustify</td>
<td>Bool</td>
<td>False</td>
<td>WordProcessor</td>
<td>SelectionAlignmentJustifyPropertyChanged</td>
</tr>
</tbody>
</table>

Table 1. The created dependency properties with registration information.

When a Dependency Property changes its value, for example by a data bound control in the user interface, the belonging callback method is invoked which handles the text format change. The callback methods can be divided in two different types; *inline format* and *paragraph format*. The inline format can be applied to a single word or to a selected phrase inside a paragraph, for example the bold and italic property. The Paragraph format applies a text format to a whole paragraph where the current selection is, and includes the alignment and list properties.

When a callback method is executed, the value of the used Dependency Property is changed and any selected text or the paragraph object at the selected position in the FlowDocument is obtain as a TextSelection object. The TextSelection represents the selected text together with methods to handle it. In order to apply the format setting, the TextSelection method ApplyPropertyValue is used. The method has two parameters; one defining which property to apply and one with the new value of the property.
**Update of User Interface**
When the selected position in the document is changing, the formatting settings of the current position should always be updated in the user interface in order to inform the user. For example, if the currently selected position in the text is marked as bold, any data bound bold button should appear enabled. This is handled by the method called UpdateSelectionProperties, which is executed when the WordProcessor event SelectionChanged is triggered. The method inspects the format settings of the current selection and compares them to the Dependency Properties, if one value differs, the Dependency Property value is updated and notifies any data bound objects.

**Document Fragment Management**
This chapter describes the structure and management of Document Fragments and their contents.

**Functionality**
The document fragments contain a number of Document Paragraphs placed in a global paragraph library. The paragraphs received from the paragraph library are represented as a XAML text strings in order to preserve words marked with an inline format when used in different places. However, it is the document fragment that decides how each Document Paragraph should be presented, and thereby support an arbitrary text formatting of the text. For example, one Document Paragraph used in two document fragments can have the font family Arial in one document fragment and Times New Roman in the other. But the inline formatting is consistent.

The document fragment can be viewed in an arbitrary language. If one used Document Paragraph miss a translation in the current language view, the Document Paragraph text is showed in the source language instead and marked red.

When the text of a document fragment is changed and saved, all new Document Paragraphs are included in the global library and all changed Document Paragraphs are updated in the global library. The library notifies every document fragment ViewModel using an updated paragraph to reload its contents. If the text change is considered to be important, the user has the possibility to discard all translations of a Document Paragraph when performing the update.

**Design**
The fragment management design is influenced by the design pattern called Model-View-ViewModel (MVVM). MVVM is a variation of the Model-View-Controller (MVC) pattern and was developed by Microsoft to facilitate building of WPF applications that separates the appearance and behavior of the application. Therefore, the pattern is designed to be used in the WPF data binding structure [JS01].

MVVM includes three element types; Model, View and ViewModel, which are similar to the ones used in MVC. The Model refers to business objects and should be unaware of being used in a display or not. The View represents elements in the graphical user interface and the ViewModel is used to pass information between the View and the Model. The ViewModel exposes properties and commands from the Model that can be used by the View through data binding, where valid information and procedures gets forwarded from the ViewModel to the corresponding Model.
Model classes
The prototype contains the following classes representing the Model part. The Model classes are presented as an UML diagram found in Appendix 1.

A. DocumentFragment – The DocumentFragment class contains information about the contents of a DocumentFragment, the available ParagraphStyle objects of the document fragment and methods for management.
B. ParagraphStyle – The ParagraphStyle class represent a text formatting style.
C. FragmentRepository – The FragmentRepository class contains DocumentFragment objects and methods to handle them. The DocumentFragments are gathered in a Dictionary.
D. DocumentParagraph – The DocumentParagraph class represents a paragraph and contains a text string, which is used in the DocumentFragment by the DocumentParagraph id. The text is placed in a Dictionary together with any translations, listed by their languages.
E. ParagraphLibrary – The ParagraphLibrary contains a number of DocumentParagraphs, collected in a Dictionary, and methods to handle them.

ViewModel classes
The ViewModel classes expose properties and commands from the Models that can be used by Views in the user interface through data binding. The ViewModel classes are presented in Appendix 1 and are shortly described below.

A. FragmentViewModel – The FragmentViewModel class represents a DocumentFragment object. The class provides bindable properties together with a save contents command. The FragmentViewModel shows the DocumentFragment contents in a chosen language.
B. ParagraphStyleViewModel – The ParagraphStyleViewModel class represents a ParagraphStyle object.
C. AllFragmentsViewModel – The AllFragmentsViewModel class handles a group of FragmentViewModels and contains a method to create FragmentViewModels for every DocumentFragment in the FragmentRepository class.
E. AllParagraphsViewModel – The AllParagraphsViewModel class handles a group of ParagraphViewModels with a common translation. The class provides a method to create ParagraphViewModels to every DocumentParagraph in the ParagraphLibrary containing the chosen language.

Change Notifications
In order to notify clients participating in a ViewModel – View data binding relationship when a property value has changed, all ViewModel classes implements the INotifyPropertyChanged interface. INotifyPropertyChanged does not have any methods or properties, only one event named PropertyChanged. By triggering the PropertyChanged event when a property value has been changed, all clients data bound to the property get the notification and updates.

The Model classes FragmentRepository and ParagraphLibrary also contain events, which are used to inform the classes AllFragmentsViewModel and AllParagraphsViewModel when a new Model object has been added to their collections. The ViewModels thereby gets notified about the new Model object and can create a representing ViewModel.
When a DocumentParagraph is updated, the event ParagraphUpdated in ParagraphLibrary is fired which informs all FragmentViewModels that their contents has changed and needs to be reloaded. The ParagraphLibrary also contains an event which is triggered when a DocumentParagraph translation of a new language type is added to the library. The information is used by the AllParagraphsViewModel class which provides a bindable list of added languages to the Views.

Implementation

**Document Fragment Content**
The DocumentFragment content is presented as a FlowDocument object in the FragmentViewModel, but is saved as an XDocument in the DocumentFragment class. The XDocument class represents an XML document and can be effectively traversed by using LINQ. Therefore, the FlowDocument can contain random blocks with Paragraph objects at arbitrary positions and all Paragraph objects can be easily extracted. The Paragraphs are stored in the XDocument with their text formatting options, but the text content of every Paragraph element is empty, instead each Paragraph contains a Tag attribute with a DocumentParagraph ID. This enables the ViewModel to fill in the text according to a chosen language decided in the user interface, by reloading the text of the Paragraph objects to the wanted language to the FlowDocument. The possibility to use DocumentParagraph which are not shared is not implemented in the prototype, but in that case the ID Tag attribute also could contain information about in which library to obtain the DocumentParagraph from, the local or the global.

**Document Fragment Content Update**
When the document fragment content is updated and saved, every Paragraph object is inspected and the text is then cleared. If the checked Paragraph has an ID, the text of the Paragraph is compared to the text in the library to determine if the text has been updated or not. If the text has been changed, the text is updated in the global library. If the user wants to discard all translation when performing the update, the global library is informed to dismiss all translations during updates of Document Paragraphs. Paragraph objects which misses an ID contains a new text which is inserted in the global library, if the text is not empty.

**Save Command**
When the Save Command in a FragmentViewModel is executed, the FlowDocument is parsed before the DocumentFragments makes its update. This is because all new Paragraph blocks in the FlowDocument automatically receives the same ID Tag attribute as the one above. For example, when a text is entered between two paragraphs, the new paragraph gets the same ID as the one above. Therefore, the FragmentViewModel has a list of previous added Paragraph objects to keep track on the Paragraph objects which are already added. When a new Paragraph object is found, the ID tag is removed in order to inform the DocumentFragment during its update that it is a new text.

**Save Command Limitations**
Unfortunately, the prototype does not manage to recognize saved Paragraph objects when they are moved in the FlowDocument, by for example copy and paste or by pressing enter at a beginning of a paragraph. When a paragraph is copied and pasted into the FlowDocument, it is interpreted as new a new Paragraph object and the text is added to the library again. When moving a paragraph down in the document by pressing enter at the paragraph beginning, the FlowDocument contents is only shifted in the already existing Paragraph objects and new objects are created only for the last paragraphs, which therefore is interpreted as a new object and inserted in the library again. A better
technique for determine which Paragraph objects that are actually new during a save is needed, for example by catching the press enter event and process the information, but this is not implemented in the prototype due to the time limit.

**Translation Management**

*This section describes how the translation of document fragments is managed.*

**Functionality**

The user has the possibility to export selected document fragments for translation to a specified language. The used paragraphs in the document fragments are then extracted to a XML file, where each paragraph is presented as a paragraph node. The paragraph node has one Source element, containing the source text to be translated, and one target node where the translations can be provided. All inline formats is kept by using tokens in the text, which describes if the text format is normal (N), bold (B) or italics (I). The person performing the translation can then see which format option the words and phrases should have during the translation, which enables the inline formatting to be consistent in all languages, independent of the amount of words used.

When the texts have been translated, the XML file can be imported and all translations are added to the Document Paragraphs in the library.

**Design**

The TranslationManager class is presented in Figure 8. The TranslationManager class has one method to export fragments for translation and one method to import translations, together with a number of help methods. The export method is called from the ExportFragments command in the AllFragmentsViewModel class, with a DocumentFragment ID list representing the DocumentFragments chosen to export and a file name decided by the user. The Import method is called from the ImportParagraphs command in the AllParagraphsViewModel class, with the chosen XML file name to open.

The property TargetLanguages represents available `CultureInfo` languages the user can choose to make the exportation to. The CultureInfo class provides information about different language and country cultures, including language names which are used in this application to represent different languages.

![Figure 8. UML diagram of the Translation Manager class.](image)
Implementation

Export Document Fragments for Translation
The export document fragments procedure begins by determine which document paragraphs the exported document fragments includes. The contained DocumentParagraph IDs are stored in an ID list, where an ID only is saved once to avoid the same paragraph to be translated more than once. If a DocumentParagraph already contains a translation in the target language, the paragraph is ignored and not included in the ID list for translation. When the ID list for translation is set, the XML document is created which is described in the following section.

Structure of the XML Translation File
The XML document has the following structure, displayed in Figure 9.

Figure 9. The defined structure of the generated XML file used for translation.

- TranslationFile
  - SourceLanguage Attribute – The CultureName definition of the source language.
  - TargetLanguage Attribute – The CultureName definition of the target language.
- Paragraph
  - ID Attribute – The Document Paragraph ID.
  - Source Node – The paragraph text to translate.
  - Target Node – The provided paragraph text translation.

The text of a DocumentParagraphs is saved as a string which represents the inline formats in XAML code (defined as Run elements). The inline tokens are relatively long and are therefore replaced with shorter representations in the translation file. This is achieved by using Regex, which is .Net framework’s technique for handling of regular expressions. With Regex, a text can be parsed to find character patterns which for example can be replaced, edited or extracted. When performing an export, the allowed XAML inline formats are replaced with the tokens <N> for normal text, <B> for bold text and <I> for italics. The end tag of inline XAML format is always the same (</Run>), and is therefore removed in the translation file in order to minimize the number of inline tokens. The inline formats are instead ended only by the start of another.
**Import Document Fragments from Translation**

When a translation XML file is imported, the TranslationFile node is checked to make sure it has a valid target language. All paragraph nodes are then extracted to a node list. Every paragraph in the node list is checked to see if it has an added translation, in that case the translation is added to the Document Paragraph in the library or else the user is notified about the missing translation and next node is checked. Before adding the translations, all inline tokens are replaced by their XAML value.

**Result**

The theses resulted in a prototype to manage document fragments. The prototype has support for creating document fragments, enter text, edit text format, include paragraphs from a shared library, export document fragments to translation and import translations from a XML file.

**Application Overview**

The application user interface is displayed in Figure 10 with a description of the different areas.

![Figure 10. The graphical user interface of the prototype.](image)
1. Top Toolbar – The top toolbar contains four areas called Fragment, Edit, Paragraph Style and Font. By the Fragment group the user can create a new document fragment or save the currently opened. The print and open alternatives showed are not implemented. In the Edit Toolbar group the user can use short cuts to copy, cut, paste, undo, redo and delete. The Paragraph Style and Font area provides the available text formatting alternatives. The Apply button is used to apply the selected paragraph style in the ComboBox.

2. Current Document Fragment – Shows the text of the currently opened document fragment where the user can edit the text.

3. Fragment Information – Information about the currently opened document fragment. The user can set the name of the document fragment and decide its width.

4. Fragment Language – Provides information about the source language and the user can change the language view in order to see the document fragment in another language. The “Discard Translation at Text Updates” checkbox is used to decide whether or not any translation is considered out of date and be removed when an update of the text is performed.

5. Fragment Library – Shows all document fragments and the user can select which one to open.

6. Export and import – Contains two buttons; one to export selected document fragments for translation to the chosen Target Language and one to import translations from file.

7. Paragraph Library – Shows all paragraphs which have a translation to the chosen language.

8. Insert Paragraph – Inserts a selected paragraph to the selected position in the text.

The following sections show some of the mentioned features above.

**Change the preferred width**

The preferred width of the document fragment can be set in the textbox, see Figure 11. The information is used when including the document fragment in documents, which is not covered in the thesis domain.

![Figure 11. The entered width is directly applied to the document fragment and showed to the user.](image-url)
Select Language View

The document fragment can be viewed in an arbitrary language, chosen by the Language View ComboBox to the left (Figure 12). Paragraphs that miss a translation in the current is showed in their source language and marked red (Figure 13). The document fragment can only be edited in its source language, therefore the document fragment is locked when displayed in another language and a lock symbol is showed next to the chosen language.

Figure 12. The user can view the document fragment in languages where the content is fully or partly translated. The language view is chosen by selecting one of the available languages in the ComboBox control.

Figure 13. Paragraphs that miss a translation in the current language view are marked red. The document fragment can only be edited in its source language; therefore a lock symbol is showed to the left in order to notify the user.
**Update Paragraph**

When a paragraph is updated, the text is updated in the library and thereby in every document fragment where it is used. Figure 14 and 15 shows an example.

![Figure 14. The document fragment and paragraph library before making the update.](image)

![Figure 15. The first paragraph has been updated from “several modules” to “42 modules” with italic style. The document fragment was saved and the text is updated in the paragraph library together with the new inline information.](image)

**Include Paragraph**

The user can include a paragraph from the library by selecting the paragraph in the library and then press the Insert Paragraph button. The paragraph is then inserted after the selected position in the text. Figure 16 and 17 shows an example.

![Figure 16. The user has selected the paragraph “Create templates for products...” in the paragraph library and holds the mouse pointer over the Insert Paragraph button which makes the button turn blue.](image)
View Paragraphs in Library

The paragraph library filters the shown paragraphs by language. The user can select which paragraphs to show by a ComboBox control (Figure 18 and 19).

Figure 17. The user has pressed the Insert Paragraph button and the paragraph was inserted to the document fragment. The user has then applied a bulleted list style to the paragraph.

Figure 18. The user can change the paragraphs to show by using the Show Paragraphs ComboBox control.

Figure 19. The user has selected “Dutch” and all paragraphs with a Dutch translation are shown.
Analysis of Result
The thesis work proceeded well and the majority of the use cases were implemented which shows the suggested concept of document fragments. The exception is in use case (E): support for both local and shared paragraphs, the prototype only support shared paragraphs.

The initial time plan turned out to be a little time optimistic and a test session planned to the end was dismissed due to report writing. The performed test cases would have been a useful way to find and correct problems not showed during the development.

Future Work
The application developed in this thesis was only a prototype; therefore there is much work left which can be improved and features that can be added. The following section presents some of the areas which were defined during the development.

- Support for local paragraphs which are not included in a global library. The text in the document should then clearly state which of the paragraphs that are shared and which that are local. The user may also have different alternatives when including a paragraph from the global library. For example, a possibility to include a paragraph from the paragraph library as a local copy or a shared paragraph.
- A better technique for determine which paragraphs that are new and which are only updated when performing a save. For example, when the user copy and paste a shared paragraph within the document fragment, the pasted text should be recognized and not added to the library once again.
- Support for removing paragraphs and document fragments. The remove procedure should consider where a removed document fragment or document paragraph is used in order to notify the user about possible consequences.
- Support for drag and drop to enhance the usability. For example, include a paragraph from the shared library by drag the selected item to the document and drop it at the wanted position. The possibility to move paragraphs within a document fragment by using drag and drop would also increase the user experience
- Search possibilities. The document fragments and paragraphs would be easier to find with a search feature.
- Support for additional text format alternatives, such as tables.
- Support for images in document fragments, both shared and local.
Summary and Conclusions
The objective of this thesis was to develop a prototype which shows the concept and benefits when providing the possibility to share document contents between several documents. The prototype was developed in C# / WPF as a part of a larger prototype system. The concept of the complete system is to make the document more modular by dividing the document in smaller parts, called document fragments, and building the documents by putting together available document fragments. The domain of this thesis is the management of the document fragments. In order to make the documents even more modular, the paragraphs in a document fragment can be shared between several document fragments. The prototype also provides text formatting alternatives and a translation procedure.

One of the noticed benefits by the share text concept is the preserving of text consistency when texts are used in several documents. Because the shared texts are gathered in one single library, text updates will automatically project to all document fragments where it is used. This prevents misses which may occur while updating a text manually in all documents where it is used. The concept of shared texts is also more time efficient, since the update only needs to be performed at one place.

Another benefit when gathering shared texts at once place is the simplification of the translation process. Instead of sending a whole document to translation or manually determine which document parts that needs to be translated, the paragraphs that miss a translation can be easily extracted from the library and sent. The work procedure is therefore more effective and avoids texts which already have a translation to be retranslated.

The ability to share paragraphs between document fragments also spare the user from having to create a new document fragment for every paragraph that is going to be used in several documents. The user can for example easily use the same section headings in several document fragments or create lists where only some of the list items are unique in different document fragments.

But some features needs to be added in order to prevent the user from making changes that can lead to unexpected consequences. Paragraphs in a document fragment that are shared should be clearly marked in order to notify the user that any changes of the text will affects all document fragments were it is used. The user may for example have to unlock a shared text before making any changes, or approve the changes to be performed by a confirmation dialog with the affected document fragments.

The system also needs good search possibilities in order to avoid creation of document fragments and document paragraph that already exists. For example, if the library contain similar or identical items it can be hard to determine which one to use in which context. The possibility to get detailed information about shared document fragments and paragraphs would also be useful from the user perspective to separate resembling items, by for example provide a support to add comments about created items. The fragment and paragraph library would also be better facilitated with filter options, since the amount of information will be substantial. The available items could for example be filtered by category, type and document.
References


Appendix 1