A new approach to structured document building

An analysis of 'automated document processing' at Kadaster

Master Thesis

Author:	Robbin Janssen
Student number:	s4005236
University:	Radboud University Nijmegen
Faculty:	Faculty of Science
Course:	Information Science

Graduation number:	182IK
Supervisor:	Prof.dr.ir. Th.P. van der Weide
Date:	17 April 2013

Abstract

Kadaster is a company that is responsible for the land registry in the Netherlands. Not only do they register information about the geography and the location of properties, they also provide information to the public about this.

They have designed and implemented a system to automatically process documents, provided by customers, containing such information. Since there is no standard format for supplying documents of this kind, they designed and implemented a system that accepts defined document models, which can be used by customers.

The creation of these document models is time-consuming; it causes a lot of discussion about such a model between the business- and IT-department. One of the reasons for this is that document models are created using a self-defined color-coding syntax and Microsoft Word. This thesis proposes a solution to this problem by introducing a model where all these document models can be based on, replacing the color-coding syntax

The basic concept is that the model slices a document into a sequence of smaller parts called blocks. A block can be as little as one character, but can also contain other blocks forming a complete document. Different types of blocks offer different functionality to create and compose a document.

Kadaster can gain a lot of benefits using a model to base structured document models on. It can be a starting point for automating the complete process.

Contents

Table of tables

Table of figures

Table of document models

1 Introduction

In this rapidly changing computer-aided world we are always looking for tools and methods to support us in the field of work. We want to perform tasks smarter, produce faster and make sure that the whole process is sustainable and reusable whenever we want to perform the same task over and over again.

There are a lot of work tasks like these that involve producing a document as the end result, and in a lot of work fields the same document is used over and over again but with different data (O'Leary, 2011).

This thesis focuses on a similar process within a company that is responsible for land registry in the Netherlands; *Kadaster BV*. Not only do they register information about the geography and the location of properties, they also provide information to the public about this. This applies to houses and buildings, but also for ships, aircrafts and (underground) transportation networks.

The data they use is mostly gathered from legal documents, the necessary data is extracted from these documents and processed in a system. However, the documents provided are no standard formatted documents. To cope with this problem Kadaster has designed, created and implemented a system that accepts 'standard' formatted documents to be processed automatically into their system. Using this method there is hardly any manual labor required to process a legal document.

What they have created is called a *document assembly* system. In short: they use self-written *document models* that are provided to an IT department. A working application is created based on a document model, which results in a document assembly system. Knowing which model created what document, they can easily extract data from such a document and process it into their system automatically, resulting in; *document automation*. Customers can use this assembly system or create own implementations of this system. (J. Vos, personal communication, Septemer 23th, 2012).

This thesis will take a closer look at the process described in short above. Kadaster indicated that the process of creating a *document model* that can be *assembled* and automatically processed takes to long.

1.1 Terminology

The terms *document assembly*, *document models* and *document automation* give a global idea of what Kadaster tries to achieve, but does not capture the full width of the actual process.

- *Document assembly* indicates that a document can be assembled based on given data, but not how the document is defined.
- *Document models* define what a model/document can do, but not how to use it.
- *Document automation* is a term that might be too broad for the subject at hand because it sometimes refers to managing, comparing and analyzing documents as well.

A term that potentially covers the full width of the system is a '*drafting system*'. A drafting system is an intelligent system that can draft documents based on structured document models and save the data of that model (Lauritsen, Current Frontiers in Legal Drafting Systems, 2007).

1.2 Focus of the research

Drafting systems help you draft documents. A mostly forgotten, or treated as least important, part is not the actual drafting of documents based on *structured document models*, but drafting or defining the *structured document models* **themselves** (Lauritsen, Current Frontiers in Legal Drafting Systems, 2007). Because the problem for Kadaster lies within this part of the drafting system, this thesis focuses on creating a general approach to such a (part of the) system.

To optimize the process of creating a document model within Kadaster, we need to overcome several obstacles.

> *Look into the current process of creating a document model and find flaws in the process.*

By looking into the complete process we get a clear view of how the current process works within the organization. By finding flaws in that process we can find limitations of the current method and find functionalities that Kadaster would like to have.

> *Define the requirements of a structured document model*

When there is a clear view of the current process we need to take a look at how Kadaster creates document model. Find out what kind of elements they use within these documents, what the current shortcomings are and how this document model gets translated into an application.

> *Propose a model for structured document models*

A model can be proposed for creating a structured document model once the requirements are clear. Within the proposed model the research will focus on documents in global, not just documents based on Kadaster.

Basic interactions and functionalities of this model have to be defined as well. A description of how an application can work with this model should give the user a notion of what the model is capable of and how they can interact with the model.

> *Storing and storage structure of structured document models*

Once the model has been proposed it is important to define how such a model can be stored, of course many solutions can be applied here to achieve this. But a general storage structure will be defined.

Throughout the research one example document of Kadaster will be used to test and validate the requirements, model and storage structure.

Summing this into a research question results in the following:

> *How do structured document models benefit the process of document automation within Kadaster?*

In the following chapters the statements above will be dealt with, together they will show what benefits this solution can have to the document automation process within Kadaster.

1.3 Method

Many papers exist about *document automation*; by reading and searching other papers about this subject we will try to understand the know-hows about document automation.

Kadaster has published all of their current document models and methods for creating and implementing those models online on their website. These sources will be used to define requirements and to get a global idea of how a company like Kadaster works with documents. Next to these documents we can contact Kadaster to provide any additional information next to these documents.

These are the sources for proposing a model and storage structure, to test the model and storage structure a small application will be written to test the functionality and validity of the model. An actual live document will be created using the model and storage structure.

The literature used in this research will be in the bibliography and organized according to the APA standard. These references will be visible throughout the whole research.

1.4 Relevance

The web is constantly evolving and the rise of web-applications has officially started a while ago, more and more web-based applications are being developed (NEM, 2012). In the field study of this thesis we can find some web applications that are similar to drafting systems as well. However, most of them are based on old studies and have to cope with legacies of previous applications (Lauritsen, Current Frontiers in Legal Drafting Systems, 2007).

Data within documents needs to get where it is going much faster then it used to. A clear separation between data and content should benefit this (O'Leary, 2011).

It is important to realize that in these new times technology is rapidly evolving. Being able to utilize these new technologies might give new insights on how to create more intelligent documents.

2 Requirements and relevance

Structured document models are used for creating and processing large amounts of automatically generated or drafted documents. The structure and principles where these document models, or any other objects for that matter, are based on is of utmost importance and is the base for all applications that want to work with these documents (Lauritsen & Gorden, 2009). These structured document models are a crucial part of a *drafting system*. Any crucial mistakes in the model and thus the structure can result in a terrible system and in an even more terrible user experience (Hafner & Lauritsen, 2007).

2.1 Why use a structured document model for documents

Imagine a ticket for a fine handed out by a police officer; to the human eye it appears to be a normal ticket or document, to a computer system however, it can be much, much more. A ticket contains a lot of information, for example details of the offender, the details of the officer, the amount of the fine, date, location, car information and much more. If the filled out ticket is based on a structured document model then we can automatically process all these details with another system that makes sure that the fine is assigned to the correct person, that the car is registered, the person has insurance etc.

When data is saved in a structured way a computer system can create, read, update and delete this data. If a structured document model is connected to this data, then we have the possibility to recreate documents based on given input. We can alter the document, including its structure as well. Because the data is stored in a structured way, we can process the document automatically. A system can perform actions based on the given data input from a structured document. In other words, structured document models can greatly benefit in automating business processes (Lauritsen & Gorden, Toward a General Theory of Document Modeling, 2009).

Systems used to generate structured document models are called '*drafting systems*'. These *drafting systems* are tools that help you create a structured document model that eventually can be filled in by a user (Hafner & Lauritsen, 2007). The ticket example used above looks like a simple document; it has few fields to fill in and has a pre-defined layout. Saving a structured document model of this ticket seems easy at first, however there are obstacles that need to be conquered. E.g. definition of what appears where when. Filled in fields that can be reused in the document; for example; a name that is reused in a later part of the document, or multiple choices that affect the layout of the document.

We can use structured document models for more any purpose, for example responses to customer complaint letters. This requires a structured document to fill in predefined data, offer choices of solutions, maybe add or remove parts of texts and changes to layout. Now the model already has properties, multiple values, positions and mandatory or optional status.

Note that the more functionality, freedom and possibilities the model has, the harder it becomes to create a tool that utilizes all these features and is easy useable for a user (Macleod, 1990). This thesis focuses on creating a model for basic, structured document models. The purpose of this model is to define a better insight in which these structured documents can be built.

2.2 Implementation of a document model

As described in chapter **Fout! Verwijzingsbron niet gevonden.** this thesis is closely based on legal document models that are created and used by Kadaster. Before it is possible to make assumptions, a definition and a technical description of a new model, it must be clear what the requirements are and what functionality the model needs.

To realize this it needs to be clarified what it is that Kadaster wants to achieve using a structured document model. As we know, Kadaster processes large amounts of documents and gathers data from these documents. The data gathered will be processed and used in their core business process system. Extracting this data should be quick and easy; integrity and validation of the data is important.

Kadaster uses a system that can (re-) generate documents based on content from an Extensible Markup Document (XML) file and a model identifier number[1], this number indicates that this XML file should contain content to generate a document *'Levering'* for example, in other words, the meta-information about the document. The XML Schema Document (XSD) file contains the document layout, document structure and document validation for all possible documents. The XSD is based on a proposed document model. The combination of these gives you the functionality to (re-) generate documents (M. Arfman, personal communication, September 9th, 2012).

A structured document that can be processed automatically by Kadaster contains:

1. Meta-information about the document such as version number and creation date,
2. A document model containing a set of elements that form the content of the document,
3. The validation of the document,
4. The layout of the document.

[1] Kadaster uses the word 'Depot number' in their documentation to indicate a document.

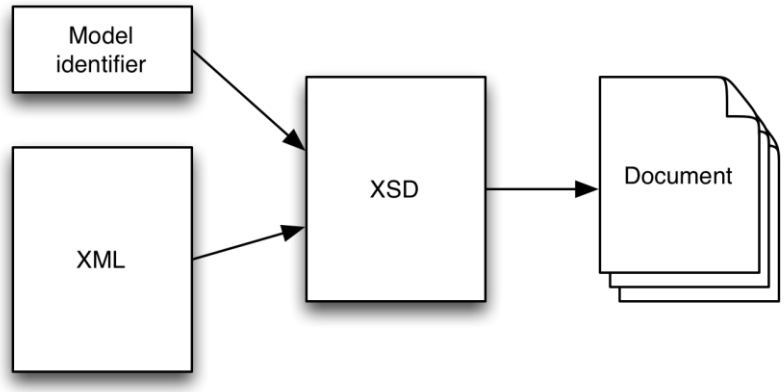

Figure 1 – Kadaster document processing

The XSD created and used in this application is based on **all** of the proposed document models that are defined internally; all models are defined in one file. These document models are made by hand, not by a *drafting system* or any other application of that kind.

Kadaster uses a text-processing tool (Microsoft Word) and a self-defined color-coding syntax to define the layout, content and rules for a document model. The biggest challenge here is that the model is not saved in a structured, ordered and reproducible way. Because of this a lot is unclear when another department tries to create an implementation of this document model. In chapter 2.5 the challenges and limitations with the current process and color-coding syntax are explained. For now the same functionality and elements as this syntax will be used to create the proposed model.

2.3 Creating a document model

When a new type of document or a change in an existing document is proposed, the document model for this document will be (re-) defined. As stated before, the document model is drawn through a self-defined color-coding syntax, not a *drafting system*. In chapter 2.3.2 the color-coding syntax defined by Kadaster and its flaws/limitations are explained in detail.

Domain experts for this particular type of document define the document model. These experts have to learn the syntax defined by Kadaster Because Kadaster uses a text-processing tool to define the model, **none** of the information, properties and values inside the model is stored in a defined, structured, digital way; the model is saved as a Microsoft Word document, but any text-processing tool that supports colors suffices. (J. Vos, personal communication, December 13th, 2012).

After many revisions and changes the document model is finalized and send to the IT-department of Kadaster An example of a finished document model can be found in chapter 2.4.

When the document is sent from the domain experts to the IT-department, it causes a lot of discussion and explanation about the model. Many parts of the document model are unclear, vital details are missing and there is a lot of room for different interpretations of parts of the document model.

Overcoming this barrier, the IT department creates a technical document for this document model and defines the structure and validation. It is then outsourced to a development company that will implement it into their main application. Here an application is formed that implements this structured document, but still, it is based on a document model written by hand. When a change is made, the whole process starts over.

2.3.1 Elements within the current model

Kadaster has defined their own syntax and semantics to create structured document models. Each document model has a version number and when the document changes, the version number is incremented as well.

As of now, document models can exist of the following elements: (Noort, 2011)

- Fixed and optional text
- Mandatory choice text
- Mandatory and optional variables
- Choice blocks
- (Optional) Text blocks

Most of these elements require more explanation; the document provided by Kadaster should take care of this. However, personal communication was needed to explain the different situations (J. Vos and M. Arfman). This indicates that the document that **should** provide a clear guideline on how to create document models is unclear.

Fixed and optional text

Two types of text can exist within a document. There is fixed text that cannot be changed and is mandatory for the document to be valid. Then there's optional text, this text is valid to use for this document but is not mandatory. It is mostly used to create large amounts of fixed text where no variables, choices or text blocks are needed.

Mandatory choice text

Like the fixed text this choice text is mandatory for the document. The choice text exists of multiple sentences or words and one of these has to be chosen to make the document valid. A choice text can be made optional by using it within an optional text. It is used for creating multiple choices like 'Sir/Madam' for example.

Mandatory and optional variables
Like the text there are two types of variables as well. One type of variables is optional and one type is mandatory. The variable obtains a unique name within this document and can be used multiple times within the same document. A variable begins and ends with a § symbol. A variable can be used for example to use a name throughout the whole document.

Choice blocks
A choice block is a part of the document where the user eventually has to choose an instance of this block. These instances can contain for example all of the elements above. An instance of a choice block is created using the same syntax as a document model; this would indicate that a variant of a choice block is a document model itself. According to the documentation, a choice block is specific to the document model and applies only to this document model. In short, a choice block is a collection of mini documents where the user has to choose one of these mini documents.

(Optional) text blocks
A text block is a generic block of text that can be used within all the document models. Similar to an instance of a choice block, it is created using the same syntax as a document model as well. However, the main difference between these two is that a text block actually can be used within other document models where a choice block cannot. Because text blocks, like document models, can be modified and updated, these text blocks contain meta-information. When a text block is included in a document model the version number of this text block is specified as well.

2.3.2 Color-coding syntax to define elements in a document model

To use all the elements discussed above Kadaster has created it's own syntax. The syntax is defined by color-coding to show what kind of element is used (Noort, 2011).

Element	Example
Fixed text	Fixed text is colored red.
Mandatory choice text	Mandatory choice text is colored green
Mandatory variable	Mandatory variables are colored black and begin and end with a § symbol
Optional variable	Optional variables are colored purple and begin and end with a § symbol
Optional text	Optional text is colored purple
Fixed text within optional text	Fixed text within optional text is colored brown
Variable within a optional text	A variable within a optional text is colored blue
Mandatory choice text within a mandatory choice text	Mandatory choice text within a mandatory choice text is colored light blue
Choice block	A choice blocks has a green background with white text
Mandatory text block	A mandatory text block has a yellow background
Optional text block	An optional text block has a green background

Table 1 – Color-coding syntax

2.4 Example of a 'document model' by Kadaster

The syntax above can define a document model that is used to create a technical design. A document model created by Kadaster is nothing more than a text document file with text and colors. The text document gets a title and version number and this is how it is referred to in other document models. Because text blocks can contain other text blocks and can be used within multiple document models, it is the smallest document like element within a document model. A text block is a separate text document file as well.

A text block looks as following:

Tekstblok Aanhef
Versie 2.2, dd 10 januari 2011

Op/Heden,/Vandaag, ♣datum♣, verscheen/verschenen voor mij,/verklaart TEKSTBLOK PERSONALIA VAN NATUURLIJK PERSOON versie 1.0 hierna te noemen: 'notaris', als waarnemer van TEKSTBLOK PERSONALIA VAN NATUURLIJK PERSOON versie 1.0, notaris in de gemeente ♣gemeente♣ kantoorhoudende te / te / gevestigd te / met plaats van vestiging ♣woonplaats♣ als volgt

….

Document model 1 – "Tekstblok aanhef"

As seen in the example above, this text block already includes another text block. Below an example of a small part of a document model is displayed.

Modeldocument akte van levering - *conform model BTR01E/ Van Brug*

Versie 3.7 d.d. 22-02-2012

…

Kenmerk: § omschrijving §

TEKSTBLOK AANHEF versie 2.2:
§ 1. §TEKSTBLOK GEVOLMACHTIGDE versie 2.4:
§ a. §TEKSTBLOK PARTIJ NATUURLIJK PERSOON versie 2.7 / TEKSTBLOK PARTIJ NIET NATUURLIJK PERSOON versie 2.5;
§en voornemens het hierna te vermelden registergoed te gaan bewonen, § hierna zowel tezamen als ieder afzonderlijk te noemen: ("vervreemder"/"verkoper") / ("verkrijger"/"koper) / "partij [volgletter]";
en
…

Document model 2 – "Modeldocument akte van levering"

Looking at these live documents we already notice the following flaws:

1. The title and version definition lack consistency, this could indicate that no agreements have been made on how to define these.
2. Different symbols are used to indicate a variable, the spades and the paragraph symbol; one of these symbols is not defined in the color-coding syntax.
3. In the document model we see that a paragraph symbol is used in front of a paragraph, text block and fixed text, this is incorrect according to the color-coding syntax as well as this symbol is defined to use with variables.

A finished document model will be send to the IT-department, they will create a technical document and complement the XSD that can be used within an application. This thesis focuses on the creating of those document models that can be send to the IT-department. It will not focus on creating these XSD's and technical documents,

since this is a specific type of export for their system. However the storing of *structured document models* will be treated in this thesis.

2.5 Challenges with the Kadaster model

By using this syntax to define a document model, Kadaster has to cope with a couple of challenges. The most important challenges are mentioned and explained in this chapter.

1. The lack of functionality that a domain expert can use to define a document model. For example: a text or variable should only be shown if a particular choice has been made.
2. The syntax can be unclear to the IT-department on how a domain expert meant to define a part of a document model. For example: if two **optional** texts are set after each other (meaning they have the same color) it is unclear where the separation is intended by the domain expert.
3. It is not clear to the IT-department which variable is meant where, sometimes a document model uses the same variable in multiple places and it is not clear which of the two variables is meant where. For example a ‘surname’ can be defined twice in a document for the selling and buying party, the current syntax only uses the name ‘surname’ to indicate a party, so to the IT-department it’s unclear where surname ‘A’ is meant and where surname ‘B’ is meant.
4. When a document or text block changes the version number is updated. However all of the other documents that use this document or text block have to be manually updated as well. For one or two documents this is not a problem, but when a lot of documents are defined it is hard to maintain.
5. There is no hierarchical view that shows which documents are dependent on each other and where they are being used. Missing this view also harms the maintainability.

These and other challenges result in a lot of communication errors between the domain experts and the IT-department. There is a lot of room for different interpretations of a document model because of some of these challenges. This is not desirable when a syntax is defined. The syntax should make sure that there is *no* room for difference in interpretation (Miller, 1992).

What we need is a clear definition of a model for a structured document model, when this is defined a tool is needed to draft document models based on this structured document model.

2.6 Current drafting systems

Drafting system software is reasonably common in legal and insurance companies nowadays. Most of these systems require the user, for example a lawyer, paralegal or do-it-yourselfer, to work through a series of question dialogs. The answers to these questions are used to fill the document with data. Another example is that a

user picks particular forms and parts of documents from a library that eventually are merged into a full document.

These types of systems are all based on regularities within the structure of these documents. These regularities define which sections, paragraphs, sentences or words are placed where under what circumstances in a document. The terminology varies among these systems, most of them use a *template* that functions as a model for a particular kind of document. Using the terms described in chapter 1.1 a *template* of this kind can be described as a *document model* that can be *assembled* within a system. (Lauritsen, Current Frontiers in Legal Drafting Systems, 2007)

These systems are mainly focused on *assembling* a document based on a *document model*, not on creating an actual *document model* that can be *assembled*. This thesis focuses on creating the actual *document model*, not *assembling* it.

2.7 A sample session

Defining a *structured document model* requires another kind of system than a system that is used *assembling* a *document model*. This chapter describes the kind of interactions a user may have with such a system and a basic explanation about the steps of creating a *structured document model*.

The concept of such a system is that the content of a *document model* results in a defined *structured document model* that can be translated into a document, but also back into that same model. (Lehtonen, Petit, Heinonen, & Lindén) There will be no more room for errors in different interpretations about a field that has been defined between the domain experts and the IT departments, because it has been defined according to a model and is saved structured.

There can be many ways in which such a system allows you to create a *structured document model*. For example inputting a predefined grammar that forms your model or an application that allows you to drag and drop certain elements to form a structured document model.

The key element for such a grammar or application to work is a defined *model* where all the *structured document models* are based on. In chapter 3 a basic *model* will be proposed that meets the requirements stated in the previous chapters. A storage structure is needed to save the *structured document* model after creating it according to the proposed model. Once saved, the application needs to be able to reopen that model and continue editing.

An ideal session with such a system would require the following global steps:

1. Start with an empty structured document model.
2. Fill the structured document model with data based on the proposed model. This can be done with for example;
    a. Grammar input.
    b. An application allowing you to create a document by interacting with it.
3. Export the structured document and save it as a new version according to a defined storage structure.
4. The defined storage structure allows the IT department to (fully automatically) process the structured document model into a useable document.
5. Reopen a structured document model that is loaded from the storage.
6. Repeat from step two.

Most modern day systems have an application that works online, mobile and in the cloud. This can also be achieved when creating an application based on this concept. In chapter 3.3.3 interactions with the model will be explained in more detail, for now a global definition of such an application will be given.

Visualization

For starters the application should offer the functionality to a *what-you-see-is-what-you-get* principle. The user creating a *structured document model* should be able to see on the fly how a change, addition or deletion of text affects a document. However, a *structured document model* can be become very large and complex, a user should be able to 'zoom-in' on particular parts of the document that displays more information and editing options about that particular part of document (Lauritsen, Current Frontiers in Legal Drafting Systems, 2007).

Lifecycle of a structured document model

A document model can be updated, edited or even removed. In other words, a document model changes and evolves. When a document model is altered enough it might occur that previously assembled documents based on that document model no longer work because the data that is required as input for that document has changed. That's why a document model needs to have a version and support for rendering of previous versions (Lauritsen & Gorden, Toward a General Theory of Document Modeling, 2009).

Storing document models

When a user has finished working on a document model, he or she needs to be able to save the document. There are many solutions, in many formats and languages, on how one could store a document model in a structured storage. But the principle is the same; store the data and make sure it is reusable and loaded in the same way as it was stored. A user should be able to export the model in a defined storage structure based on conventions that allows other tools to work with these models.

3 The proposed model

In this chapter a model is proposed for document models based on the requirements and basic features defined in chapter **Fout! Verwijzingsbron niet gevonden.**. Assumptions about different elements and their functionalities will be made. Those elements will be modeled and described in details as well. Finally the complete model and functionalities will be proposed.

3.1 Assumptions

As described in chapter 2.2 this thesis will closely look at Kadaster to propose a model. The elements that will be proposed in this chapter have been derived from the syntax and documents provided on their website. With these basic elements it is possible to create a document that fulfills the needs for every structured document that Kadaster uses at the moment.

Assume that a structured document can contain the following elements:

1. Text,
2. New variables (Assignments)
3. Existing variables (Variable retrievals)
4. Multiple choices (Choices)
5. Other structured documents (Blocks)

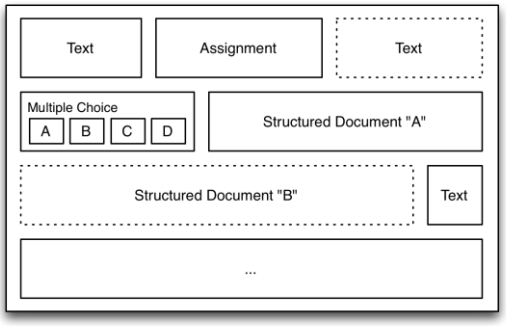

Figure 2 – A simple representation of a document model

These five elements are the basis for the newly proposed model. Each element has its own properties and functionality; of course some properties and functionalities are shared across elements.

The figure illustrated above illustrates the basic concept for the goal of this model. Creating a document by inserting small elements with their own functionality. Some are optional (striped lines) and some are mandatory. The end result will be a document that is created by ‘gluing’ all these small elements together in the correct **sequence**.

Lauritsen & Gorden define all these elements as ‘text’ in their general approach to a document model and do not make a clear distinction between these five (Lauritsen & Gorden, 2009). In the end everything can be reduced to text or characters, but it might be wise to make a clear distinction between elements because it allows you to divide and share functionality and properties of these elements. In another article written by Lauritsen and his colleague Hafner they *do* make a distinction between these elements when they zoom in further on the details of such a model (Hafner & Lauritsen, 2007). In the proposed model for this thesis a distinction between these elements is made as well. A lot of properties are shared, but not all. Some unique properties can influence the document vastly.

As illustrated in **Fout! Verwijzingsbron niet gevonden.** creating a document requires 'gluing' multiple elements together. From now on we will refer to each element as a *block*. We will start with the basic, fairly straightforward elements that a block can be.

> Assumption 1
>
> *A block (b) is an instance of an element text (t), assignment (a), variable retrieval (vr) or choice (c).*

This can be described as following:

$$b = t \,||\, a \,||\, vr \,||\, c$$

Creating a document requires the use of multiple blocks. Assume that we have *Text* t_1, t_2, t_3 and *Assignment* a_1 and a_2. A document is an instance of the sequence of these blocks:

$$d = <t_1>, <t_2>, <a_1>, <t_3>, <a_2>$$

It is possible that we want to reuse the same 'sequence' of blocks in another document. To achieve this we need to be able to *copy* a set of blocks from one document to another. Now if a change occurs in this sequence we might want that sequence to be changed in all the documents that depend on this sequence. To achieve this we will assume that a *block* is a sequence of at least one or more blocks.

$$\boldsymbol{Seq_1} = <t_1>, <t_2>, <a_1>, <t_3>, <a_2>$$

> Assumption 2
>
> *A block contains a sequence of one or more blocks.*

Seq_1 is now defined as a sequence of blocks and can be used inside the same or other documents.

$$\boldsymbol{d} = <seq_1>, <t_4>, <t_5>, <t_6>$$

$$\downarrow$$

$$<t_1>, <t_2>, <a_1>, <t_3>, <a_2>$$

A block containing a sequence of blocks, which is actually a mini-document, can be related to documents in three ways (Lauritsen & Gorden, Toward a General Theory of Document Modeling, 2009):

1. Intra-documentary
The *block* is related specifically to a particular document, it is only used in one document and has no function in other documents.

2. Inter-documentary
The *block* is related to at least one document, but can serve other documents as well. An example can be a salutation in a letter. One should strive for a maximum amount of *inter-documentary blocks*. This can be achieved by keeping *blocks* small and thus reusable.

3. Extra-documentary
Blocks that have no relation whatsoever to any *documents*. These can be for example *blocks* that will be used later or are old and no longer used.

Can we still speak of a document? Not really; documents have been minimized to a level where a document can exist of a single block of text or even a character; a large document is nothing more than a very long sequence of blocks. There will still be made references to 'documents' in this paper. From now on when we speak of documents or sequences, we speak of a **sequence of blocks**.

> Assumption 3
>
> *A document exists of a sequence of blocks.*

Something else that can be noticed from this sequence of blocks is the tree structure that will eventually form the backbone of a document. This behavior is illustrated below.

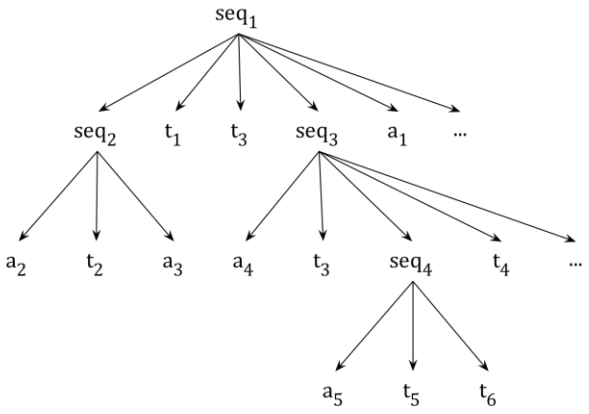

Figure 3 – Document as a tree

The tree structure can grow vastly and become complex when large documents are created. Smart methods are needed to quickly traverse within the tree. However when used properly, a tree structure can be a very powerful tool that allows you to reuse parts of a document.

In the next chapter a basic *block* element will be define, in the chapters following all type of elements that a *block* can be will be described in more detail.

3.1.1 Block

A *block* is the most important element in the proposed model. It's properties and functionalities are the basis for all other elements. A *block* exists of a *layout* and a *version*. Furthermore it can be set to *mandatory* or not, since we have optional parts in documents as well.

A *layout* is needed to define how the *block* displays inside a document. For now only little attention will be paid to the layout of a document. The model mainly focuses on the structure of a document, layout specific details like when a break or indentation is made can be added later on.

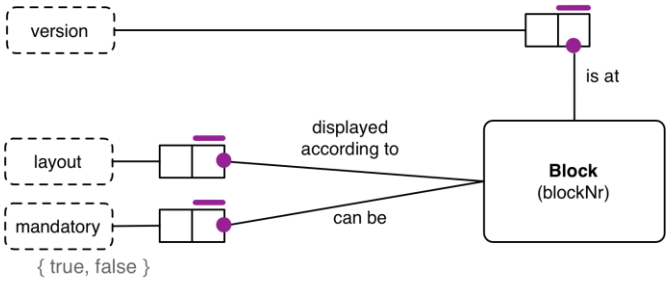

Figure 4 – Proposed model – Single Block

The version is needed for keeping track of revisions. It can also be used to regenerate a document at a given version with given data. For example: the data used inside an employment contract can be the same for the years 2012 and 2013, but the content can differ as discussed in chapter 2.7. If a contract needs to be renewed the next year, the data can be reused and it's still possible to regenerate the previous contract using that same data.

Given this, we can assume that a document is build up out of a sequence of specific versions of blocks. This means that a sequence of blocks is an actual block that points to other blocks. For example[2]:

$$\boldsymbol{Seq_1} = \langle t_{1,\,3.0}\rangle, \langle t_{2,\,2.1}\rangle, \langle a_{1,\,1.4}\rangle, \langle t_{3,\,0.7}\rangle, \langle a_{2,\,4.1}\rangle$$

We can now propose our next assumption based on this information.

> Assumption 4
>
> *a document exists of a sequence of blocks at a specific version.*

[2] *To improve readability the block version number will be omitted in the examples following.*

We can now extend the proposed model for a single block above with a sequence of blocks at a specific version. The following model is derived from these details and forms the basis for the proposed model.

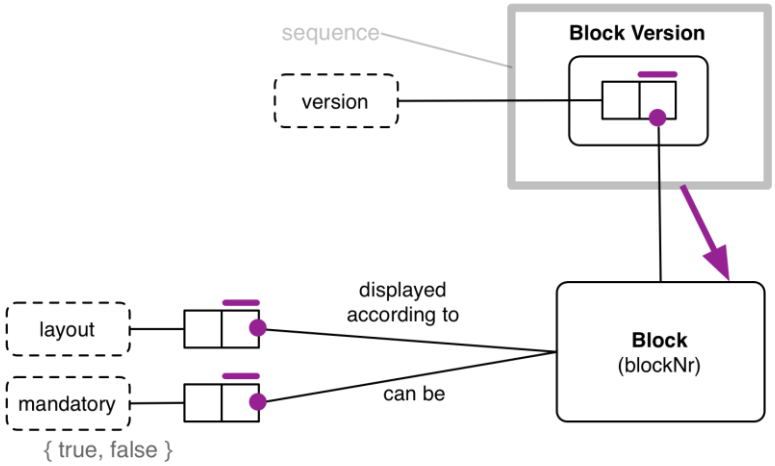

Figure 5 – Proposed model – Block

In the following chapters the other elements will be explained. All these elements are specializations of a block. They inherit all of its properties and functionalities.

3.1.2 Text

A text element is introduced to fill a document with content, as stated above all elements are specializations of a block and thus inherit properties and functionalities of a *block*. It has a *layout*, can be *mandatory* and the *version* is tracked. Because we want to propose the model in its simplest and most minimalistic form we will only add a *content* attribute to this element.

The content of a text can vary in length. It can be a single character, a word, a sentence or even a complete paragraph depending on the content of a document. For example:

$$<t_1, \text{“Vandaag verscheen voor mij”}>$$

If we want to the text to be optional, we add curly brackets to the example:

$$\{<t_1, \text{“Vandaag verscheen voor mij”}>\}$$

A user needs to be smart in defining *texts*; this can be very lucrative in reusing *blocks*. For example, reusable texts can be the closing part of a formal letter, or the address part of a letter.

Figure 6 below illustrates how the *text* element specializes the block element.

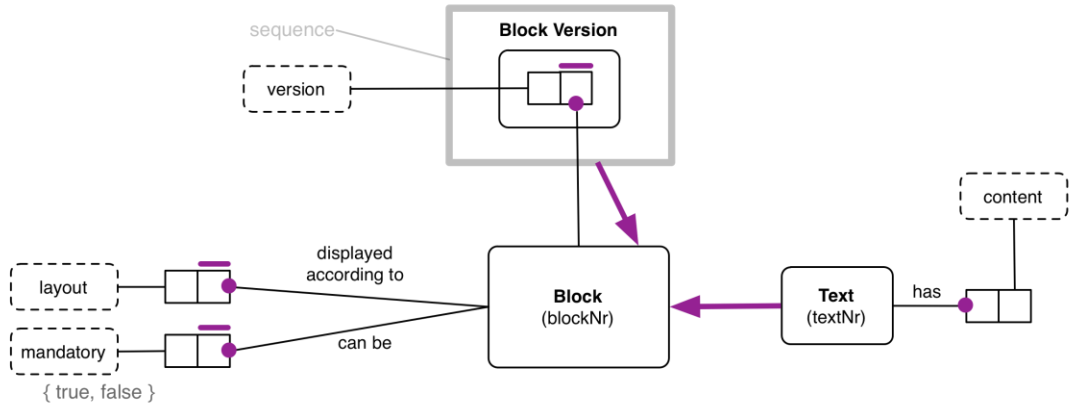

Figure 6 – Proposed model – Text

3.1.3 Assignment

It might occur that you want to introduce a *variable* in a document that can be filled with data, for example: the surname in the heading of a letter. The *variable* and data, from now on *value*, form a pair. This pair is called an *assignment*. *Assignments* can be reused in the same document; this functionality is explained in chapter 3.1.4.

A definition is needed of what type of *value* is accepted by the *variable* within an *assignment*. To achieve this an element called *variable type* is introduced. A variable type can be reused across other variables and defines the rules and validation for a variable. Types with their own rules can be dates, money, surnames, initials or email addresses for example.

$$<a_1, \text{“oplever datum”}, date, ?>$$

Assignment is also a specialization of the block element. It has no extra properties except for the relation with *variable* and *value*. The *variable type* defines for example the validation rule in a regular expression and the (max)length of a *variable*.

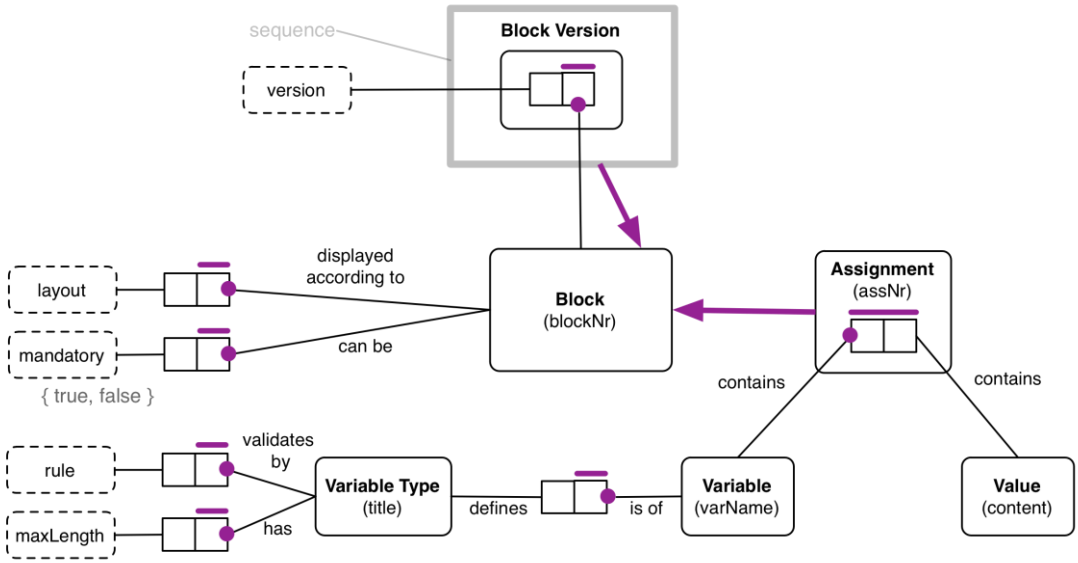

Figure 7 – Proposed model – Assignment

3.1.4 Variable retrieval

It is now possible to introduce *assignments* into documents. But it might occur that one wants to reuse the same filled-in data all over a document, for example when referring to a person's name. If we create a new *assignment* it will result in duplicate data. What is needed is an element that reuses an *assignment* and thus the *value* within this *assignment*. To achieve this the element *variable retrieval* is introduced.

The *variable retrieval* is a specialized element of block that has an extra property. That property is a derivative to an *assignment.* It defines which *assignment* exactly it needs to be retrieved.

When we request to retrieve a variable we know it is attached to an assignment (that does not imply that a value is assigned!). A simple grammar like $<vr_4>$ suffices because the actual value can be derivative through the assignment.

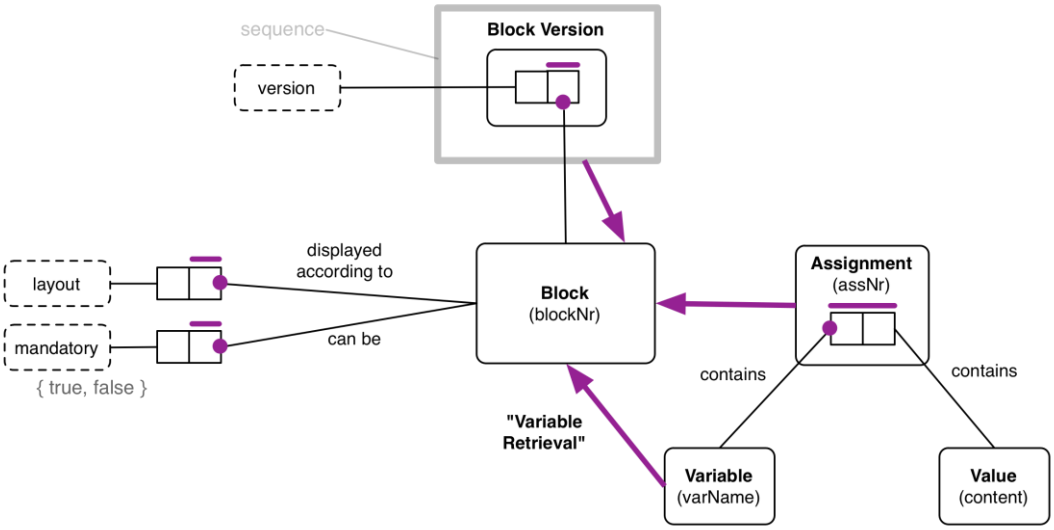

Figure 8 – Proposed model – Variable retrieval

To quickly retrieve the defined *assignment* within a document a *post-order* walk can be used to traverse to the *assignment*. This kind of walk traverses all child nodes before their respective parents are traversed. Details about this interaction with the model will be explained in the chapter 3.2.

3.1.5 Choice

It might occur that you want to provide a multiple *choice* within a document. For example when there's a document describing a person that is leasing a car, a *choice* can be the kind of lease or brand of car. A multiple *choice* can have unlimited *options*. The actual *choice* is one of the defined *values*.

An example of a rendered *choice* can be for example:

Favorite car brand

```
What is your favorite car brand?
   1. BMW
   2. Audi
   3. Mercedes
```

As you can see, a choice can have an indicator as a prefix for the choice. This can be set within the choice. Above example is translated as following:

$$<c_1, [1:"BMW", 2:"Audi", 3:"Mercedes"], ?>$$

The `?` at the end indicates that no choice has been made yet. If a choice would have been made the `?` would contain the actual chosen value, for example:

$$<c_1, [1:"BMW", 2:"Audi", 3:"Mercedes"], \text{“BMW”}>$$

What we need is a sequence of *choices* that form the *options* for a multiple *choice*. Next to that there needs to be a possibility to order those options in a specific way. A *choice* element has many *option* elements. These *option* elements are bound to the *choice* by an *indicator* that orders the *options*. An *option* exists of a value.

> Assumption 5
>
> *a choice exists of at least two options that contain a value.*

Once a choice is made between one of the *options* it will be saved as an assignment, by doing this the chosen option can be retrieved again in the document.

This can be visualized in the proposed model as following.

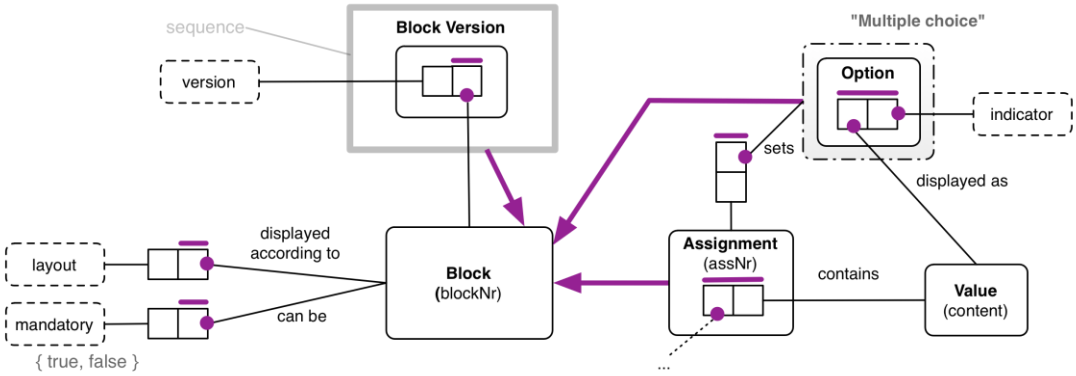

Figure 9 – Proposed model – Choice

3.1.6 Conditional choice

Based on the described *choice* element above we need to introduce another type of element: a *conditional choice block*. A *conditional choice block* is used when we want to make a decision within the document that vastly changes the content. In contrary with choices we 'choose' which text appears at that position.

For example: with a defined conditional choice block we can achieve the functionality that based on your earlier conditional choice, a specific sequence of blocks is added to the bottom of a document. Some other choice would have added another specified sequence of blocks at the bottom of the document. After a choice has been made the chosen *'option block'* will be set as a variable for this reference.

A conditional choice also makes it possible to create a choice between for example a variable and a sequence of blocks at the position where the element is defined.

Adding this we have to correct assumption 1.

> New assumption 1
>
> *A block (b) is an instance of an element text (t), assignment (a), variable retrieval (vr), choice (c) or conditional choice (cb).*

This can be described as following:

$$b = t \,||\, a \,||\, vr \,||\, c \,||\, cb$$

The grammar used for a conditional block is similar to the grammar used in choice, except for the options, they are now references to other blocks.

$$<cb_1, [1:<seq_{52}>, 2:<seq_{23}>], ?>$$

Visualizing it into a model looks as following:

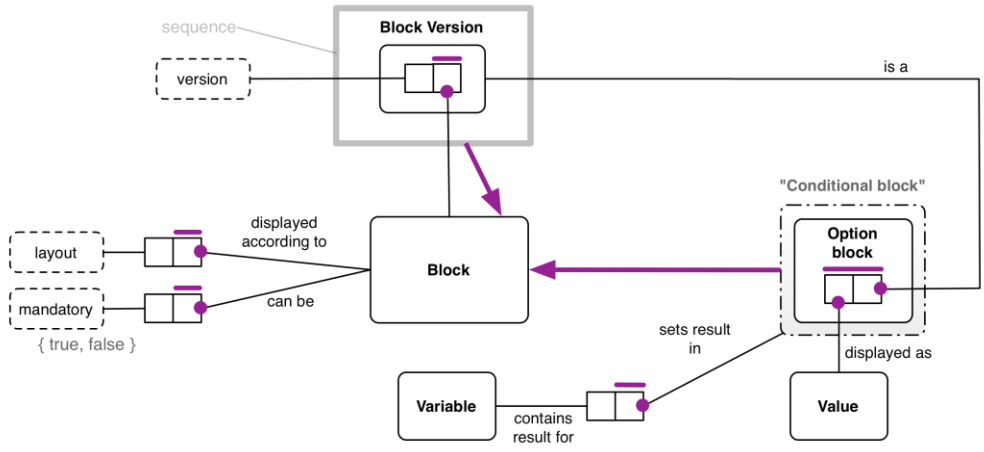

Figure 10 – Proposed model – Conditional choice

3.2 Definition of the model

All elements have been defined and modeled into a proposed model. When all these elements are put together in a model then the complete model scheme is formed, based on the assumptions and definitions made above. The scheme can be found in Figure 11. To keep the model readable and clear, some properties of elements have been disregarded in this figure.

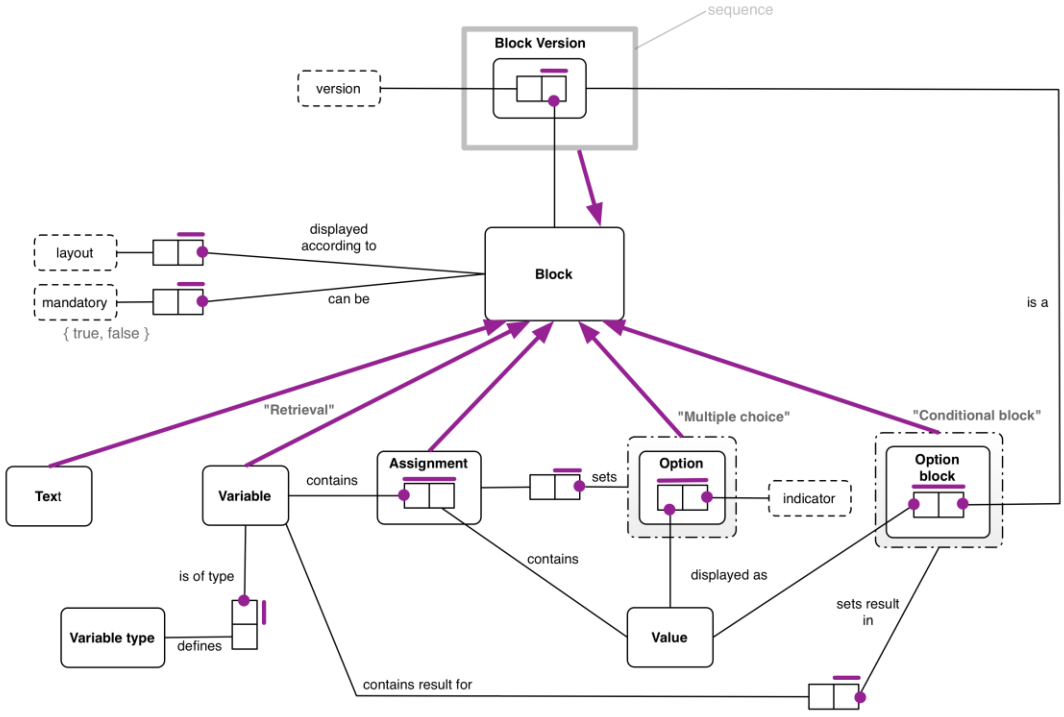

Figure 11 – Definition of the model

Some functionality can be derived from this scheme. As stated before a *block* can exist of a sequence of *blocks*. A sequence of *blocks* in fact already is a document.

The *block* element acts as a generalization for all other defined elements. They're a specialization of the *block* element. There's no need to define an *exclusive-or* constraint because a generalization/specialization forces the choice of a specialization.

A large document can exists of many, many of these *blocks* and *blocks* containing other sequences of *blocks*. As stated before a large document is build up in a tree structure. A method is needed to retrieve the value of a previously defined *assignment* with a *variable retrieval* element. To achieve this the model needs to be able to traverse through the tree.

With these kinds of documents it is most likely that a defined assignment is a (deep) child node. One of the traversal methods that we can apply here is a post-order walk. A post-order walk is a depth-first walk in which the node will traverse the children of all leaves/sub trees first before it will traverse its parents (Lerma, 2005).

In the case of the proposed model, it will traverse the deepest blocks first before it traverses their containers. Because of this, an assignment that is declared in a block node at the deepest level can be used in a block at the highest level. The traversal route of a post-order walk is shown in the figure below.

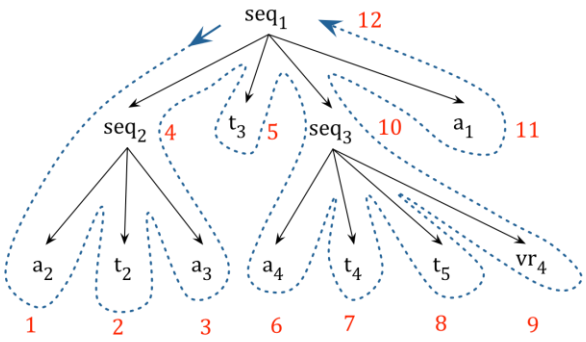

Figure 12 – Post-order walk for the retrieval of an assignment.

The *'root' block* is a *block* called seq_1, this is the main container for the *document* and the first node for our tree. It then has a sequence of blocks. *Block* seq_2, *text* t_3, *Block* seq_3, *assignment* a_1, etcetera. Each *block* contains more elements. The blue arrow describes the line in which the post-order walk traverses through all the elements. The red numbers show in which order the nodes are traversed.

At the red number 9 we find a *variable retrieval* vr_4. Assume that we want to retrieve *assignment* a_3 using this vr_4. Using a post-order walk we find, within 3 steps, the corresponding *assignment*. The traversal of the tree is as following: $a_2 \rightarrow t_2 \rightarrow a_2$.

The same method of traversal can be used when we want to locate a previously defined sequence of blocks.

3.3 Functionality

This chapter will describe the basic functionality for the model to work with. A 'real-life' example is introduced and an idea of the drafting system is presented.

3.3.1 Example based on real document using the model.

In this chapter a live document defined by Kadaster used in multiple of their documents will be translated into the model. All elements described for the model are used in this document. First the original document will be shown, followed by a table in which that particular sequence of the document is 'cut' into little parts that fit the model. These sequences can be reused within the document.

Original: “Tekstblok aanhef”

Op/Heden,/Vandaag, ♣datum♣, verscheen/verschenen voor mij,/verklaart TEKSTBLOK PERSONALIA VAN NATUURLIJK PERSOON versie 1.0 hierna te noemen: ‘notaris’, als waarnemer van TEKSTBLOK PERSONALIA VAN NATUURLIJK PERSOON versie 1.0, notaris in de gemeente ♣gemeente♣ kantoorhoudende te / te / gevestigd te / met plaats van vestiging ♣woonplaats♣ als volgt

Document model 3 – Content for “Tekstblok aanhef”

Defined as: Seq_1 - “Tekstblok aanhef”

Document	Model
Op/Heden,/Vandaag,	`<c1, [1: "Op", 2: "Heden," 3: "Vandaag,"], ?>`
♣datum♣	`<a1, "datum", date, ?>`
,	`<t1, ",">`
verscheen/verschenen voor mij,/verklaart	`<c2, ["verscheen voor mij,", "verschenen voor mij,","verklaart"], ?>`
TEKSTBLOK PERSONALIA VAN NATUURLIJK PERSOON versie 1.0	`<seq2>`
hierna te noemen: ‘notaris’, als waarnemer van TEKSTBLOK PERSONALIA VAN NATUURLIJK PERSOON versie 1.0	`{<seq3>}`
, notaris	`<t4, ", notaris">`
in de gemeente ♣gemeente♣ kantoorhoudende te / te / gevestigd te / met plaats van vestiging	`<seq4>`
♣woonplaats♣	`<a10, "woonplaats", string, ?>,`
als volgt	`{<t7, "als volgt">},`

Document model 4 - Grammar of content for "Tekstblok aanhef"

Original: “Tekstblok personalia van natuurlijk persoon”

professor §adellijke titel§ §titel§ §voornamen§ §adellijke titel§ §voorvoegsels§ §achternaam§ §titel§

Document model 5 - Content for “Tekstblok personalia van natuurlijk persoon”

Defined as: Seq_2 - “Tekstblok personalia van natuurlijk persoon"

Document	Model
professor	{<t_2, "professor">}
§adellijke titel§	{<a_2, "adelijkte titel", string, ?>}
§titel§	{<a_3, "titel", string, ?>}
§voornamen§	<a_4, "voornamen", string, ?>
§adellijke titel§	{<a_5, "adelijke titel", string, ?>}
§voorvoegsels§	{<a_6, "voorvoegsels", string, ?>}
§achternaam§	<a_7, "achternaam", string, ?>
§titel§	{<a_9, "titel", string, ?>}

Document model 6 – Grammar for content for “Tekstblok personalia van natuurlijk persoon”

Original:

hierna te noemen: ‘notaris’, als waarnemer van TEKSTBLOK PERSONALIA VAN NATUURLIJK PERSOON versie 1.0

Document model 7 – Content for second “Tekstblok personalia van natuurlijk persoon”

Defined as: Seq_3

Document	Model
hierna te noemen: ‘notaris’, als waarnemer van	<t_3, "hierna te noemen: 'notaris', als waarnemer van">
TEKSTBLOK PERSONALIA VAN NATUURLIJK PERSOON versie 1.0	<seq_2>

Document model 8 – Grammar for content for second “Tekstblok personalia van natuurlijk persoon”

Original:

in de gemeente ♣gemeente♣ kantoorhoudende te / te / gevestigd te / met plaats van vestiging

Document model 9 – Content for conditional choice

Define: Seq_4

Document[3]	Model
1. **in de gemeente ♣gemeente♣ kantoorhoudende te /** 2. **te / gevestigd te / met plaats van vestiging**	`<cb1,[1:<seq5>, 2:<seq6>], ?>`

Define: Seq5

Document	Model
in de gemeente	`<t5, "in de gemeente">`
♣gemeente♣	`<a9, "gemeente", string, ?>`
kantoorhoudende te	`<t6, "kantoorhoudende te">`

Define: Seq6

Document	Model
te / gevestigd te / met plaats van vestiging	`<c3, ["te", "gevestigd te", "met plaats van vestiging"], ?>`

Document model 10 – Grammar for content for conditional choice

3.3.2 Generating the example document

When the grammar is complete it is possible to generate a document from it. This can be achieved by 'parsing' the grammar.

For this thesis a JavaScript application[4] was written that parses the grammar into a JSON object. A HTML form allows you to input the grammar; a parser script will parse this grammar into a JSON object. This JSON object is composed into a HTML DOM structure that actually shows the structured document model.

[3] The numbers leading these sentences do not appear in the original document, it is purely shown here to indicate the separation between seq5 and seq6

[4] Live example at http://www.doccy.nl/thesis/

The application[5] responsible for parsing the grammar follows these steps:

1. Clean the grammar (where needed) from special characters like new line, tabs etcetera.
2. Parse the cleaned grammar according to a strict set of rules, this will be done token by token. It will eventually form an array of 'blocks'
3. For every element in the array transform it into a valid JSON object.
    a. Detect the element type
    b. Create JSON based on the element type and content from the array
    c. Return that element
4. A large JSON object is the end result.

That JSON object forms a tree that we can illustrate as following:

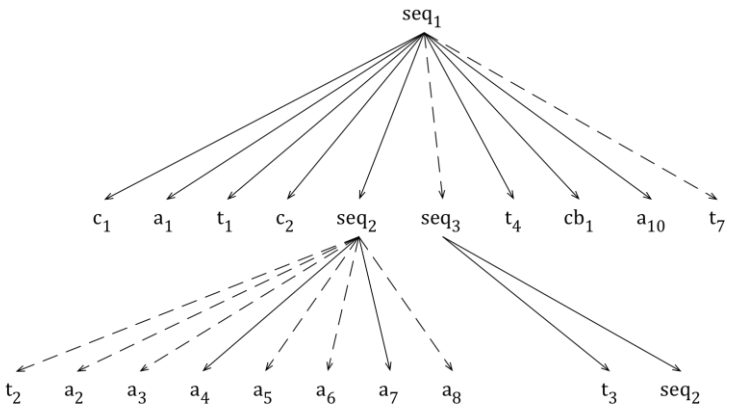

Figure 13 – JSON object tree for "Tekstblok aanhef"

This parser is a limited parser that expects perfect input, once it finds an error it will quit parsing and log about the error.

Another file was written[6] that composes a document based on a given JSON object. It will create a simple HTML DOM structure that illustrates the separation between the different elements within the grammar as show in Figure 14.

Figure 14 – Parsed grammar into HTML

[5] Source code for parser can be found at: http://www.doccy.nl/parser.txt, jQuery (http://www.jquery.com) is required to run the script.

[6] Source code for composing the grammar can be found at: http://www.doccy.nl/compose.txt, jQuery (http://www.jquery.com) is required to run the script.

Having a JSON object that has been parsed from the grammar makes it fairly easy to compose a HTML DOM structure based on that grammar. The compose script follows these steps:

1. Start at the root node of the JSON object as figured in Figure 13.
2. For each child node walk 'compose' the element into a HTML DOM object.
    a. Variable Retrieval elements will post-order traverse through the tree and find the referred assignment.
    b. Conditional Blocks will post-order walk through the tree and find and set the referred variable/assignment.
3. If a child node has child nodes itself:
    a. If blocks are defined, repeat step 2 for this node.
    b. If no blocks are defined, post-order walk through the tree and find the referred node.
4. Display the HTML DOM object.

The function composing the grammar needs to be recursive, because one sequence can contain another sequence containing another sequence, etc. The function responsible for this in the script is called parseSequence.

```
/**
 * Parse a given sequence of blocks
 *
 * @param object sequence The sequence we want to parse
 *
 * @return void
 */
parseSequence = function(sequence) {
  // Create a container where the rendered HTML will be put in
  var container = $(document.createElement('div'));
  // Check if there are any blocks defined in this sequence.
  if (sequence.blocks !== undefined && sequence.blocks.length) {
    // Walk through all of the provided blocks, and parse them
    $.each(sequence.blocks, function(i, block) {
      // Make sure the object is defined.
      if (block.object !== undefined && block.object.length) {
        block.previous = sequence;
        // Parse the element and append it to the document container
        element = parseBlock(block);
        container.append(element);
      } else {
        console.log('Error: undefined object');
      }
    });
  } else {
    // No blocks found, we need to check if this sequence was defined before,
    // if not, throw error.
    sibling = getSequence(sequence);
    if (sibling) {
      // Parse the given sequence (again) and return it.
      element = parseSequence(sibling);
      return setLayout(sequence, element);
    } else {
      console.log('Error: sequence ' + sequence.id + '
                   has not been defined within this scope');
    }
  }
  return setLayout(sequence, container);
};
```

This application demonstrates how a user can interact with the model, but for an application to function in a work environment like Kadaster it will need more functionality.

3.3.3 The functionalities drafting system

As described in chapter 2.6 most drafting systems don't take the actual *defining* of structured document models into account.

This chapter describes the minimum functionality that such a drafting system needs to successfully work with the proposed model.

1. List all existing structured document models

The system needs to be able to display all *structured document models* that have been created using the model. This means that the system will need to fetch all 'root' *blocks*. These are *blocks* that are not contained by any other *block* but do contain other *blocks*. The models shown should always be the latest version of the document. There should be a link called 'add document' where the user can start creating a new document model.

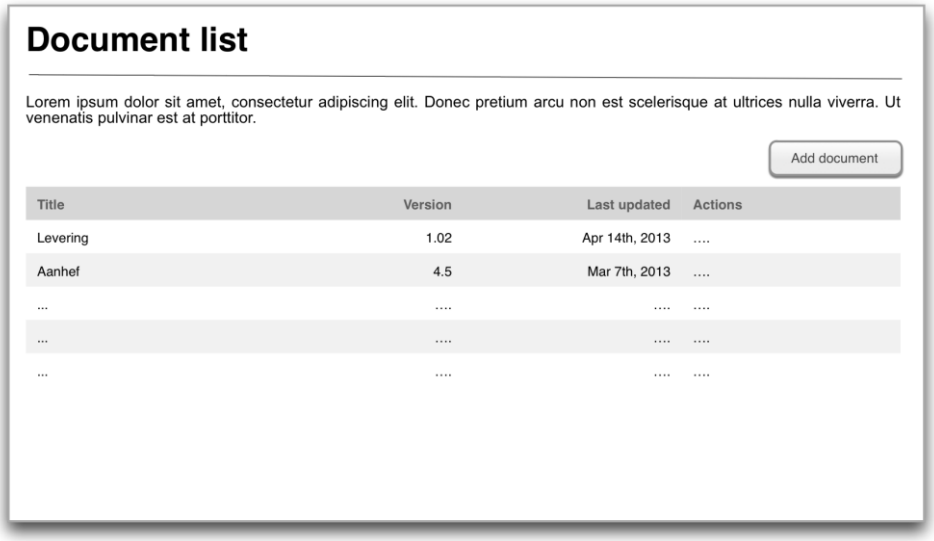

Figure 15 – Illustration of 'List documents'

2. Viewing an existing structured document model

When clicking on a link within the list created in Figure 15 the system should navigate to the selected *structured document model*. The *model* has to be shown as a 'live' document like we know it from for example Microsoft Word, as stated in chapter 2.7. This screen has three modes. The first mode is viewing, the second mode is editing and the third mode is revision viewing. The views are mostly the same, however the actions will differ.

Figure 16 – Illustration of 'view document'

Blocks that contain other *blocks* will be highlighted when the mouse hovers them. Clicking on such a *block* will result in zooming in on this *block*, in other words, navigating to the viewing page of a *block*. The viewing page of a *block* however is the same as the page currently described because a *document* is nothing more then a *block*.

Figure 17 – Illustration of 'highlighted block'

None mandatory elements in a *document* should be partly greyed out. This way a user knows that it is an optional element.

The scope of an *assignment* or *variable retrieval* has to be clearly visible. This could be achieved by using a different font and/or color with a leading symbol. For example: '$'. When an *assignment* or *variable retrieval* is hovered, the origin of this *assignment* will be highlighted in a specific color as well as all other places where the *assignment* occurs (the *variable retrievals*) in another color. This way the scope of an assignment will be clear.

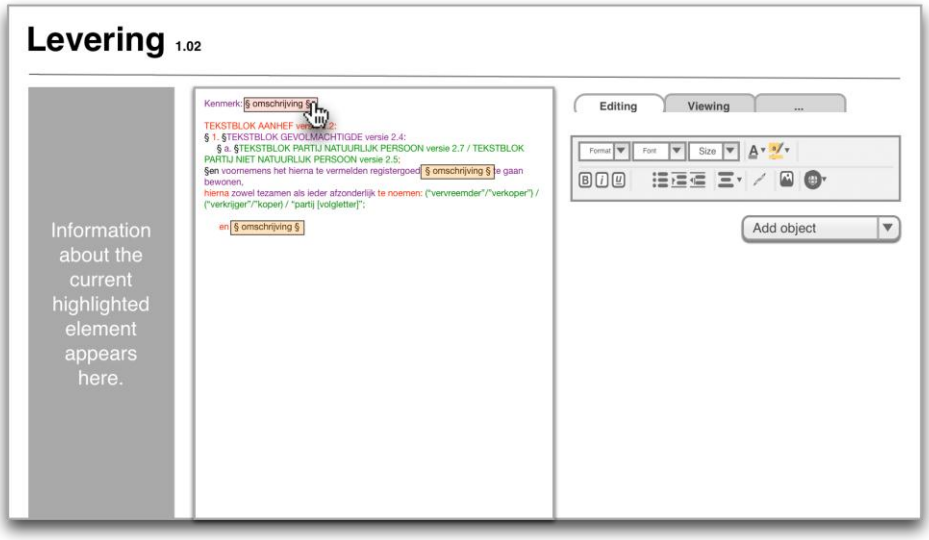

Figure 18 – Illustration of 'highlighted assignment'

3. Editing an existing structured document model.

When a structured document model is set to editing mode all elements inside this model will be highlighted when a mouse hovers them. Clicking on an element will display a small tooltip that has three options. View, edit and delete. Clicking on a block containing other *blocks* will still result in navigating to the view of that *block* as described at Figure 17. The elements within this *block* can be edited from here.

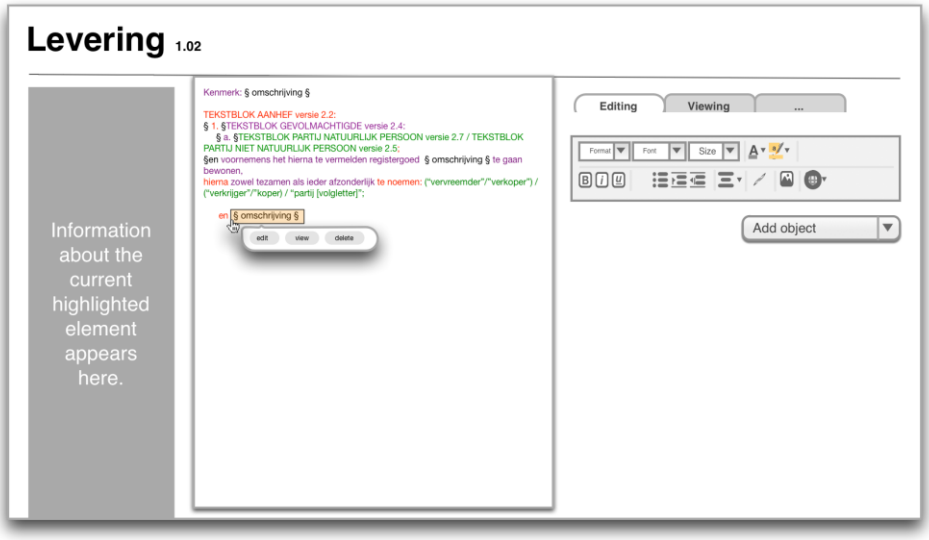

Figure 19 – Illustration of 'edit tooltip'

The box on the left side of the document will contain all the information concerning this element. For example an assignment has to display the variable, variable type, value and number of variable retrievals depending on this assignment. A choice will display all of its options (in short) and indicators. Next to this information it has to display the rendered blocks before and after this element, the dependencies and version of the element.

Clicking on the edit option in a tooltip will load a screen where a specific element can be edited. Each element has it's own edit form, however some fields appear on all forms. The version field can be updated, this triggers an action called 'Archive document', specified later in this chapter. The mandatory field can be updated as well.

A text simply has a field where the text can be edited. The variable has more details, the type can be chosen and the value can be used. A choice will allow you to add unlimited options. These options can be created on the fly or, in case of a conditional block, existing elements can be chosen. An indicator is set as well.

Clicking on the delete option in a tooltip of an element will ask for confirmation if you are sure you want to delete this element from the document. If no other documents depend on this element an option will be given to remove the element from the system.

Inserting elements can be achieved by right clicking in a document. It will place a marker where the element will appear and the system will ask what kind of element the user wants to add. The options for this will be a new element (one of the elements discussed in the previous chapter) or an existing element. When a user clicks on one of these options the system will load a popup or screen where the user can add this element. The form for adding an element will be similar to editing an element.

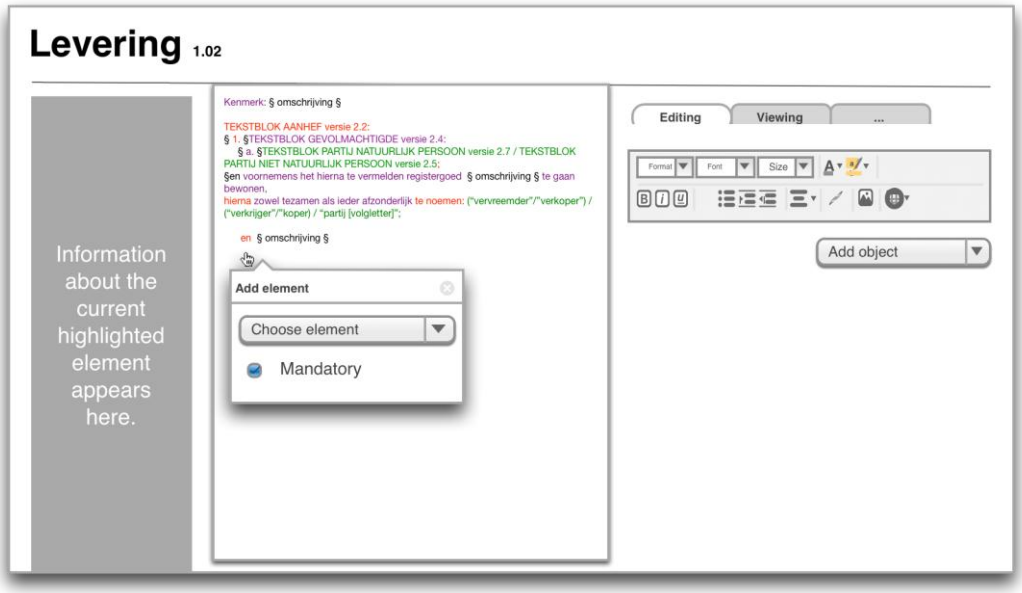

Figure 20 – Illustration of 'add element'

4. Creating a new document

When a user clicked on the 'add document' button a screen will show with a blank document. The functionality of this screen is similar to editing a document. The user can right click in a document to start adding elements.

5. Archiving documents

A user can specifically archive a document at a given time. The structured document model will be saved as a version that the user can provide through an input field. The system will automatically duplicate all elements from this document to make sure that their integrity is saved.

6. Compare document versions

This screen allows you to compare models with their previous version. It can be seen as a time machine where the user can navigate through the different revisions of a model and see exactly what changed, where and when. Elements that have changed, have been added or deleted will be highlighted.

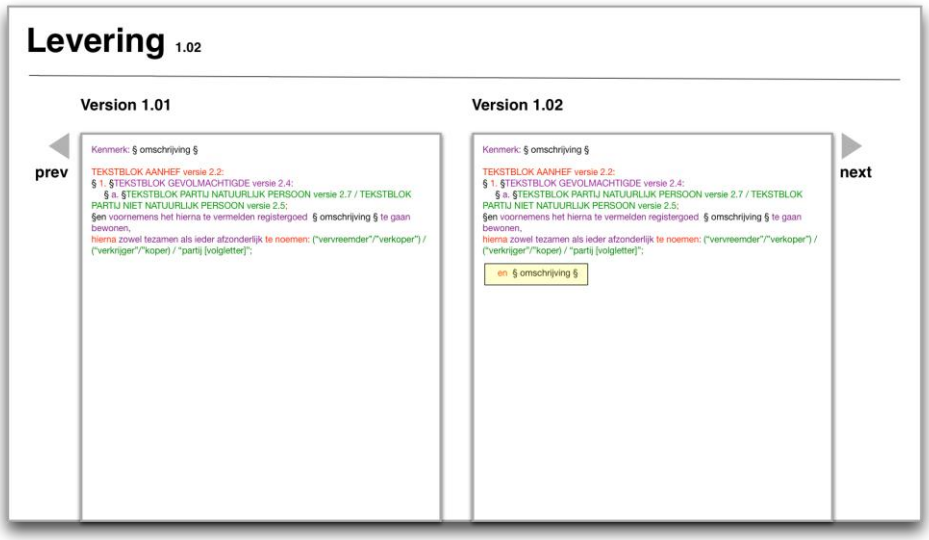

Figure 21 – Illustration of 'Revision comparing'

7. List all existing blocks

A view, similar to the 'list all existing documents' view will show all the blocks containing other blocks. The blocks that appear here are based on how many dependencies it has. The more dependencies it has, the more important it is and it will be shown at the top of the list. Clicking on a link of this block will navigate the user to the view block page.

4 Technical setup

Storing can be done using many structures and techniques. For many years XML has been the leading language for these kinds of structures, but this is starting to change slowly. An alternative that we can apply here is a human-readable open standard derived from JavaScript, called JSON[7].

There are many ongoing discussions about the advantages and disadvantages of XML versus JSON (Cormack, 2010), but for this thesis JavaScript is applied to store a *document (model)*. Our document solitary exists of elements on particular positions, making JSON a good fit. It is language-independent with parsers available in many languages, making it a good starting point for implementing our model.

JSON supports the following basic types (Crockford, 2006).

- A `number` - can be integer or floating point.
- A `string` - value should be inside double quotes, escape with backslash.
- A **`Boolean`** - **`true`** or **`false`**.
- An `array` – is defined in square brackets: `[]`.
- An `object` – is defined in curly brackets: `{}`.
- `null` – an empty value.

The type '`object`' can contain one or more members/properties. These properties are defined in pairs of name/value. An object can have many properties separated by a comma. The name is always defined as a string surrounded by double quotes. The value can be any one of the types defined above.

- Members of objects are defined as: `"property": value`

These types, together with the examples displayed in chapter 3.3.1, have been used to create a JSON schema based on the proposed model. In the schema the following useable elements have been defined: a sequence element, a text element, an assignment element, a variable retrieval element and a conditional-choice element. The schema can be found in Appendix A – JSON Schema.

4.1 Storage structure

Before we start saving an actual document model we need to take into account that, eventually, we want to store every single object separately. To securely save these objects it is needed to save the objects in pairs of a unique identifier and the version. When the version is updated we need to keep the previous version to make sure that 'old' versions of a document model work as well.

[7] JSON, or JavaScript Object Notation, is a text-based open standard designed for human-readable data interchange.

For this thesis we will create a basic structure that will suffice to save the dataset. However advanced storing structures might improve speed and storage usage (Sacks-Davis, Arnold-Moore, & Zobel, 1994).

In this example a simple MySQL database was created that stores all the data in a structured way. For now every object has it's own table(s). The storage structure can be found in Appendix C – MySQL Storage structure.

We need to take in account that a single document can and most likely will exist of many, many records. Adding indexes to tables might help improving speed, however most of the speed will rely on properly written queries and sufficient hardware for the database (Oracle, 2013).

Still, this storage structure is merely an example; it all depends on how the application works. There are a lot of frameworks, client side as well, that can work with JSON as an input. For example Backbone.js[8] accepts JSON as input, in this case we can just save data as a JSON object converted to a string. But by doing this maintainability will be harmed because the complete document is saved as an entire string, making it harder to replace or update particular parts.

4.2 Storing a document model

Storing a model based on the previously proposed document model can be done using many structures and techniques.

As shown in chapter 3.3.1 the document 'tekstblok aanhef' has been decomposed into the model. This example will be stored using JSON. Start by defining a root object for this. The object contains an object type, an identifier, a title and a version. Next to those properties, it contains a property called 'blocks' that is able to contain other (sequences) of blocks.

Root object	{
	"object": "sequence",
	"id": "seq_1",
	"title": "Tekstblok aanhef",
	"version": 1.0,
	"blocks": [...]
	}

A document always starts with a sequence containing at least one block, all of the following elements defined are an item in the `"blocks": [ ... ]` array. In other words, they are child nodes/objects of the root node/object.

[8] http://www.backbonejs.org - Backbone.js gives structure to web applications by providing **models** with key-value binding and custom events, **collections** with a rich API of enumerable functions, **views** with declarative event handling, and connects it all to your existing API over a RESTful JSON interface.

For this example the first six elements of the ‘Tekstblok aanhef’ will be defined, the complete JSON can be found in Appendix B – ‘Tekstblok aanhef’ in JSON storage.

Document	Model
Op/Heden,/Vandaag,	<c_1, [1: "Op", 2: "Heden," 3: "Vandaag,"], ?>
♣datum♣	<a_1, "datum", date, ?>
,	<t_1, ",">
verscheen/verschenen voor mij,/verklaart	<c_2, ["verscheen voor mij,", "verschenen voor mij,","verklaart"], ?>
TEKSTBLOK PERSONALIA VAN NATUURLIJK PERSOON versie 1.0	<seq_2>
hierna te noemen: ‘notaris’, als waarnemer van TEKSTBLOK PERSONALIA VAN NATUURLIJK PERSOON versie 1.0	{<seq_3>}

If the schema defined in Appendix A – JSON Schema is used to translate the grammar into JSON we get the following elements that we can store.

Choice: Op/Heden,/Vandaag,	{ "object": "choice", "id": "c_1", "version": 0.1, "options": ["Op", "Heden,", "Vandaag,",], "value": null, },

Assignment:
♣datum♣
	{
	"object": "assignment",
	"id": "a_1",
	"version": 0.1,
	"variable": {
	"name": "datum",
	},
	"variableType": {
	"name": "date",
	"rule": "/^(0?[1-9]|[12][0-9]|3[01])[\/\-](0?[1-9]|1[012])[\/\-]\d{4}$/",
	},
	"value": null,
	},

Text: ,	{ "object": "text", "id": "t_1", "version": 0.1, "content": ",", },
Choice: verscheen/verschenen voor mij,/verklaart	{ "object": "choice", "id": "c_2", "version": 0.1, "options": ["verscheen voor mij",

	<pre>"verschenen voor mij",
	"verklaart"
	],
	"value": null,
	},</pre>

Sequence:
TEKSTBLOK PERSONALIA VAN NATUURLIJK PERSOON versie 1.0
	<pre>{
	"object": "sequence",
	"id": "seq2",
	"title": "Tekstblok personalia van Natuurlijk Persoon",
	"version": 1.0,
	"blocks": [
	see Appendix B – 'Tekstblok aanhef' in JSON storage
	]
	},</pre>

Sequence:
hierna te noemen: 'notaris', als waarnemer van
TEKSTBLOK PERSONALIA VAN NATUURLIJK PERSOON versie 1.0
	<pre>{
	"object": "sequence",
	"id": "seq3",
	"version": 1.0,
	"mandatory": **false**,
	"blocks": [
	{
	"object": "text",
	"id": "t3",
	"version": 0.1,
	"content": "hierna te noemen: 'notaris', als
	waarnemer van"
	},
	{
	"object": "sequence",
	"id": "seq2"
	},
	]
	},</pre>

In the example above the definition of the sequence of blocks within *"Tekstblok personalia van natuurlijk person versie 1.0"* has been omitted, the complete definition can be found at Appendix B – 'Tekstblok aanhef' in JSON storage.

The sequence following, called seq_3, includes the previously defined sequence seq_2. We see that in the second appearance the blocks have not been defined at seq_2, which his valid according to the schema that states that "blocks" is **not** a required property. This means the tree will be traversed using a post-order walk as defined in chapter 3.2 to fetch the content of the actual seq_2.

The following tree can be derived from the objects defined in JSON:

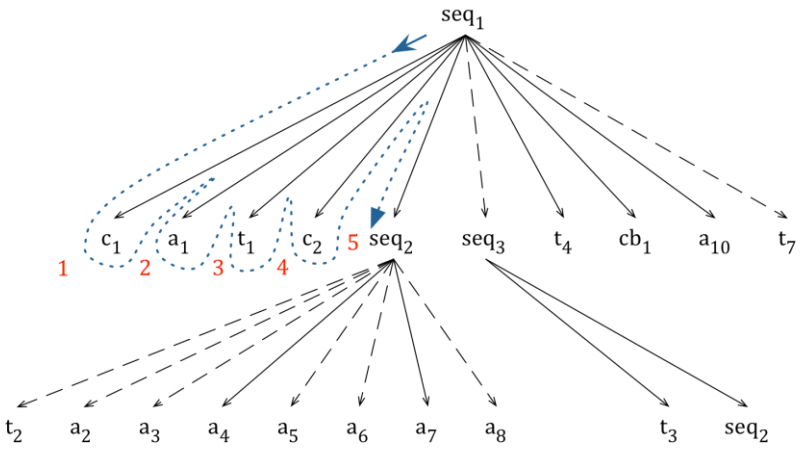

Figure 22 – Tree for "Tekstblok aanhef" with post-order walk traversal

During the parsing of the document we find inside the sequence with id seq_3 a text object with id t_7 and a sequence with id seq_2. As stated above the second time the sequence is mentioned no blocks are defined. The traversal order has been drawn to fetch the content of the actual sequence with id seq_2. We see that it takes 5 steps to fetch the actual content.

Because the sequence has already been defined within the document there's no need to actually define it twice in the document and save it. Once a document is stored we can reopen it using the application defined in chapter 3.3.2. The document can be reopened and edited from storage.

5 Conclusion

Kadaster indicated that the process of creating a *document model* that can be *assembled* and *automatically processed* takes to long. This thesis provides a solution on how to cope with this problem.

We started by describing the requirements, functionalities and flaws of the current method used, which is a color-coded syntax in combination with Microsoft Word. The main problem with this method is the room for different interpretations between the business- and IT-department. The syntax could become quite confusing, because structure and logic was missing.

Another problem that occurred frequently was maintainability, document models were relying on each other but there was no clear view of which document model is used where. When a frequently used document model was updated all other document models had to be checked manually if it included this model (Noort, 2011).

Once the requirements for a new model were clear and functionalities have been derived, a model was proposed. This model is based on all elements that are currently available in the document models of Kadaster. It copes with different revisions, reusing and maintainability of structured document models. It knows which structured document models are used where, so no more manually checking if a structured document model uses the latest version of another structured document model.

The model has been setup to decompose a document into ‘blocks’. Each block has properties and functionalities. A block can contain other blocks and thus create mini-documents. These ‘mini-documents’, which are sequences of blocks, can be reused in the same or another document.

The key element behind the use of the model is a solid application that supports and offers a lot of functionality to work with a structured document model. The main functionalities for such an application have been described, and a basic application has been created to test a structured document model.

After creating a structured document model we have to actually save the structured document model on a storage platform. The defined JSON scheme shows which properties are required for saving the model, and a limited MySQL database has been defined to save the structured document models.

This completes the cycle of creating a structured document model based on the proposed model, saving it and being able to reopen the structured document model after saving.

5.1 Implementation in the organization

The next step is to successfully setup and implement a new application based on the proposed model in the organization. Five key ingredients are essential for such a project to succeed. (Laurtisen & Soudakoff, 2005).

1. **An appropriate concept is needed.**
   An application based on the proposed model that leads to better, faster, more efficient and less error-prone documents.
2. **The right mixture of people.**
   Make sure that domain experts from the document field are involved as well as IT experts. It might also help to collaborate with some customers.
3. **Adequate tools.**
   Make sure open-standards are used to ensure endurance of the software. Create documentation about the software that is written to ensure that no knowledge is lost in case in a change of employees for example.
4. **Relevant knowledge.**
   Basically three kinds of knowledge are needed. Domain experts on the documents themselves are crucial. Experts about storing such large amounts of data and off course the IT experts who can actually implement such a system. All three need to closely collaborate to ensure the durability and efficiency of the system.
5. **An effective process.**
   Choose a process where you are constantly improving the software. Short cycles of implementation could help here. Start by creating the core functionality and keep adjusting it until it works perfectly. Once this works the core functionality can be used to build the other functionalities.

If Kadaster would implement such a model and application in their process, it would not only be a time-saver, because there's no more room for difference in interpretation, but also a starting point for automating the complete process. If a structured document model has been defined and saved, then the next step is automating the assembly functionality of these structured documents.

When this is implemented properly there will be less cause for communication and discussion between the business and IT-department. When done correctly, a business-department employee can create or update a structured document model useable by customers without any interaction with the IT-department. This should benefit the company greatly.

If done correctly, this could be a first step for creating an environment where customers themselves can define structured document models in order to give them complete freedom in how a document looks and is filled in.

5.2 Recommendation and follow-up

Further research is needed about interactions between structured document models and users. When users are interactively working with structured document models and actually seeing what impact a change might have on the structured document model is not only important, but will also make working with structured document models a lot easier.

Creating structured document models has caught the interest of me and graduated master student Daan Schraven. Together we would like to start building a prototype of an application that can work with the proposed model to generate structured document models.

In the conversations with Kadaster we've already discussed this possibility and Kadaster is very enthusiastic about the idea. A workshop will be organized to get a clear picture of the interactions with such an application. Employees from business-, IT- and other departments will join this workshop to contribute and think about the application.

Bibliography

Cormack, J. (2010, 01 27). *JSON vs XML*. Retrieved 04 01, 2013, from Technology of Content: http://blog.technologyofcontent.com/2010/01/json-vs-xml/

Crockford, D. (2006, 07 01). *The application/json Media Type for JavaScript Object Notation (JSON)*. Retrieved 04 02, 2013, from The Internet Engineering Task Force (IETF): http://www.ietf.org/rfc/rfc4627.txt

Hafner, C. D., & Lauritsen, M. (2007). Extending the power of automated legal drafting technology. In A. R. Lodder, & L. Mommers (Ed.), *Legal Knowledge and Information Systems - JURIX 2007. 165*, pp. 59-68. Amsterdam: IOS Press.

Lauritsen, M. (2007). Current Frontiers in Legal Drafting Systems. *11th International Conference on AI and Law* (pp. 1-15). California: Stanford University.

Lauritsen, M., & Gorden, T. F. (2009). Toward a General Theory of Document Modeling. *ICAIL '09: Proceedings of the 12th International Conference on Artificial Intelligence and Law* (pp. 202-211). Barcelona: ACM.

Laurtisen, M., & Soudakoff, A. (2005). *Keys to a Successful Document Assembly Project.* New York: Capstone Practice Systems.

Lehtonen, M., Petit, R., Heinonen, O., & Lindén, G. *A Dynamic User Interface for Document Assembly.* University of Helsinki, Department of Computer Science, Helsinki.

Lerma, M. A. (2005, 04 01). *Tree Traversal.* Retrieved 04 02, 2013, from Mathematical Foundations of Computer Science: http://www.math.northwestern.edu/~mlerma/courses/cs310-05s/notes/dm-treetran

Macleod, I. (1990). Storage and Retrieval of Structured Documents. *Information Processing & Management , 26* (2), 197-208.

Miller, D. (1992). Abstract Syntax and Logic Programming. *Lecture notes in Computer Science , 592*, 332-337.

NEM. (2012). Digital Growth through web-based applications and services . *13th NEM General Assembly* (pp. 65-74). Brussel: NEM.

Noort, v. H. (2011, 02 01). Tekstblok - Algemene afspraken modeldocumenten en tekstblokken. Apeldoorn, Gelderland, Netherlands.

O'Leary, D. (2011, 06 19). *5 Myths about Document Automation and Electronic Document Creation*. Retrieved 04 02, 2013, from Digital Landfill: http://www.digitallandfill.org/2011/07/5-myths-about-document-automation-and-electronic-document-creation.html

Oracle. (2013, 03 24). *Optimization and Indexes.* Retrieved 04 01, 2013, from MySQL 5.5 Reference Manual: http://dev.mysql.com/doc/refman/5.5/en/column-indexes.html

Sacks-Davis, R., Arnold-Moore, T., & Zobel, J. (1994). Database Systems for Structured Documents. *International Symposium on Advanced Database Technologies and Their Integration.* Nara, Japan: ADTI.

Appendix A – JSON Schema

Sequence	{
	  "name": "sequence",
	  "properties": {
	  "id": {
	  "type": "string",
	  "valid": "\bseq[0-9]+\b",
	  "required": true
	  },
	  "version": {
	  "type": "number",
	  "default": 0.1,
	  "required": true
	  },
	  "mandatory": {
	  "type": "boolean",
	  "default": true
	  },
	  "title": {
	  "type": "string"
	  },
	  "blocks": {
	  "type": "array",
	  "items": {
	  "type": "object",
	  },
	  }
	  }
	},

Text	{
	  "name": "text",
	  "properties": {
	  "id": {
	  "type": "string",
	  "valid": "\bt[0-9]+\b",
	  "required": true
	  },
	  "version": {
	  "type": "number",
	  "default": 0.1,
	  "required": true
	  },
	  "mandatory": {
	  "type": "boolean",
	  "default": true
	  },
	  "content": {
	  "type": "string",
	  "required": true
	  }
	  }
	},

Assignment	{
	  "name": "assignment",
	  "properties": {
	  "id": {
	  "type": "string",
	  "valid": "\ba[0-9]+\b",
	  "required": true
	  },

```
    "version": {
      "type": "number",
      "default": 0.1,
      "required": true
    },
    "mandatory": {
      "type": "boolean",
      "default": true
    },
    "variable": {
      "type": "object",
      "properties": {
        "name": {
          "type": "string",
          "required": true
        }
      },
      "required": true
    },
    "variableType": {
      "type": "object",
      "properties": {
        "name": {
          "type": "string",
          "required": true
        },
        "rule": {
          "type": "string",
          "required": true
        },
        "maxLength": {
          "type": "number",
          "default": 0
        }
      },
      "required": true
    },
    "value": {
      "type": "string",
    }
  }
},
```
 |
| Variable Retrieval |
```
{
  "name": "variable-retrieval",
  "properties": {
    "id": {
      "type": "string",
      "valid": "\bvr[0-9]+\b",
      "required": true
    },
    "version": {
      "type": "number",
      "default": 0.1,
      "required": true
    },
    "mandatory": {
      "type": "boolean",
      "default": true
    },
    "retrieve": {
      "type": "string",
      "valid": "\ba[0-9]+\b",
      "required": true
    }
  }
```
 |

	},

Choice	{
	  "name": "choice",
	  "properties": {
	  "id": {
	  "type": "string",
	  "valid": "\bc[0-9]+\b",
	  "required": true
	  },
	  "version": {
	  "type": "number",
	  "default": 0.1,
	  "required": true
	  },
	  "mandatory": {
	  "type": "boolean",
	  "default": true
	  },
	  "options": {
	  "type": "array",
	  "items": {
	  "type": "string",
	  },
	  "required": true
	  },
	  "value": {
	  "type": "string",
	  }
	  }
	},

Conidtional Block	{
	  "name": "conditional-block",
	  "properties": {
	  "id": {
	  "type": "string",
	  "valid": "\bcb[0-9]+\b",
	  "required": true
	  },
	  "version": {
	  "type": "number",
	  "default": 0.1,
	  "required": true
	  },
	  "mandatory": {
	  "type": "boolean",
	  "default": true
	  },
	  "options": {
	  "type": "array",
	  "items": {
	  "type": "object",
	  "properties": {
	  "indicator": {
	  "type": "string",
	  },
	  "option": {
	  "type": "object",
	  "required": true
	  },
	  }
	  },
	  "required": true
	  },
	  "value": {

	<pre>"type": "string",
	}
	}
	}</pre>

Table 2 – JSON schema for proposed model

Appendix B – 'Tekstblok aanhef' in JSON storage

Root object

The root object of the document, it contains a title and a sequence of blocks.

Root object:	{
	"object": "sequence",
	"id": "seq$_1$",
	"title": "Tekstblok aanhef",
	"version": 1.0,
	"blocks": [see Table 4 below]
	}

Table 3 – JSON – Document root for "Tekstblok aanhef"

Tekstblok aanhef

Contains all objects for the document.

Choice: Op/Heden,/Vandaag,	{ "object": "choice", "id": "c$_1$", "version": 0.1, "options": ["Op", "Heden,", "Vandaag,",], "value": null, },

Assignment:
♣datum♣
	{
	"object": "assignment",
	"id": "a$_1$",
	"version": 0.1,
	"variable": {
	"name": "datum",
	},
	"variableType": {
	"name": "date",
	"rule": "/^(0?[1-9]|[12][0-9]|3[01])[\/\-](0?[1-
	9]|1[012])[\/\-]\d{4}$/",
	},
	"value": null,
	},

Text: ,	{ "object": "text", "id": "t$_1$", "version": 0.1, "content": ",", },
Choice: verscheen/verschenen voor mij,/verklaart	{ "object": "choice", "id": "c$_2$", "version": 0.1, "options": ["verscheen voor mij", "verschenen voor mij", "verklaart"], "value": null, },

Sequence: TEKSTBLOK PERSONALIA VAN NATUURLIJK PERSOON versie 1.0	{ "object": "sequence", "id": "seq_2", "title": "Tekstblok personalia van Natuurlijk Persoon", "version": 1.0, "blocks": [see Table 5 below] },

Sequence:
hierna te noemen: 'notaris', als waarnemer van
TEKSTBLOK PERSONALIA VAN NATUURLIJK PERSOON versie 1.0
	{
	"object": "sequence",
	"id": "seq_3",
	"version": 1.0,
	"mandatory": **false**,
	"blocks": [
	{
	"object": "text",
	"id": "t_3",
	"version": 0.1,
	"content": "hierna te noemen: 'notaris', als waarnemer van"
	},
	{
	"object": "sequence",
	"id": "seq_2"
	}
	]
	},

Text: , notaris	{ "object": "text", "id": "t_4", "version": 0.1, "content": ", notaris", },

Conditional Choice:
in de gemeente
♣gemeente♣
kantoorhoudende te / te / gevestigd te / met plaats van vestiging
	{
	"object": "conditional-choice",
	"id": "cb_1",
	"version": 0.1,
	"options": [
	{
	"indicator": "1",
	"option": {
	"object": "sequence",
	"id": "seq_4",
	"version": 1.0,
	"blocks": [
	{
	"object": "text",
	"id": "t_5",
	"version": 0.1,
	"content": "in de gemeente",
	},
	{
	"object": "assignment",
	"id": "a_9",
	"version": 0.1,
	"variable": {
	"name": "gemeente",
	},
	"variableType": {
	"name": "alphaNumeric",
	"rule": "/^[-\sa-zA-Z]+$"
	},

	"value": null
	},
	{
	"object": "text",
	"id": "t_6",
	"version": 0.1,
	"content": "kantoorhoudende te",
	},
	]
	},
	},
	{
	"indicator": "2",
	"option": {
	"object": "choice",
	"id": "c_3",
	"version": 0.1,
	"options": [
	"te ",
	"gevestigd te",
	"met plaats van vestiging"
	],
	"value": null,
	},
	},
	]
	},

Assignment:
♣woonplaats♣
	{
	"object": "assignment",
	"id": "a_{10}",
	"version": 0.1,
	"variable": {
	"name": "woonplaats",
	},
	"variableType": {
	"name": "alphaNumeric",
	"rule": "/^[-\sa-zA-Z]+$"
	},
	"value": null,
	},

Text: als volgt	{ "object": "text", "id": "t_7", "version": 0.1, "mandatory": **false**, "content": "als volgt", },

Table 4 – JSON – Tekstblok aanhef

“TEKSBLOK PERSONALIA VAN NATUURLIJK PERSOON versie 1.0”
Contains the objects that belong to the sequence “Tekstblok personalia van natuurlijk person versie 1.0”

Text: professor	{ "object": "text", "id": "t_2", "version": 0.1, "mandatory": **false**, "content": "professor" },

Assignment:
§adellijke titel§
	{
	"object": "assignment",
	"id": "a_2",
	"version": 0.1,
	"mandatory": **false**,
	"variable": {
	"name": "adelijke titel",
	},
	"variableType": {
	"name": "alphaNumeric",
	"rule": "/^[-\sa-zA-Z]+$"
	},
	"value": null
	},

Assignment:
§titel§
	{
	"object": "assignment",
	"id": "a_3",
	"version": 0.1,
	"mandatory": **false**,
	"variable": {
	"name": "titel",
	},
	"variableType": {
	"name": "alphaNumeric",
	"rule": "/^[-\sa-zA-Z]+$"
	},
	"value": null
	},

Assignment:
§voornamen§
	{
	"object": "assignment",
	"id": "a_4",
	"version": 0.1,
	"variable": {
	"name": "voornamen",
	},
	"variableType": {
	"name": "alphaNumeric",
	"rule": "/^[-\sa-zA-Z]+$"
	},
	"value": null
	},

Assignment:
§adellijke titel§
	{
	"object": "assignment",
	"id": "a_5",
	"version": 0.1,
	"mandatory": **false**,
	"variable": {
	"name": "adelijke titel",
	},
	"variableType": {
	"name": "alphaNumeric",
	"rule": "/^[-\sa-zA-Z]+$"
	},
	"value": null
	},

Assignment:
§voorvoegsels§
	{
	"object": "assignment",
	"id": "a_6",
	"version": 0.1,
	"mandatory": **false**,
	"variable": {
	"name": "voorvoegsel",

	<pre> },
	  "variableType": {
	  "name": "alphaNumeric",
	  "rule": "/^[-\sa-zA-Z]+$"
	  },
	  "value": null
	},</pre>

Assignment:
§achternaam§
	<pre>{
	  "object": "assignment",
	  "id": "a7",
	  "version": 0.1,
	  "variable": {
	  "name": "achternaam",
	  },
	  "variableType": {
	  "name": "alphaNumeric",
	  "rule": "/^[-\sa-zA-Z]+$"
	  },
	  "value": null
	},</pre>

Assignment:
§titel§
	<pre>{
	  "object": "assignment",
	  "id": "a8",
	  "version": 0.1,
	  "mandatory": **false**,
	  "variable": {
	  "name": "titel",
	  },
	  "variableType": {
	  "name": "alphaNumeric",
	  "rule": "/^[-\sa-zA-Z]+$"
	  },
	  "value": null
	}</pre>

Table 5 – JSON – Tekstblok personalia van natuurlijk persoon versie 1.0

Appendix C – MySQL Storage structure

Sequence	`-- Create syntax for TABLE 'sequences'`
	`CREATE TABLE `sequences` (`
	`  `id` varchar(10) NOT NULL DEFAULT '',`
	`  `version` float(10,1) NOT NULL DEFAULT '0.1',`
	`  `mandatory` tinyint(1) NOT NULL DEFAULT '1',`
	`  `title` varchar(255) DEFAULT NULL,`
	`  PRIMARY KEY (`id`, `version`)`
	`) ENGINE=InnoDB DEFAULT CHARSET=utf8;`
	
	`-- Create syntax for TABLE 'sequence_blocks'`
	`CREATE TABLE `sequence_blocks` (`
	`  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,`
	`  `sequence_id` varchar(10) NOT NULL DEFAULT '',`
	`  `sequence_version` float(10,1) NOT NULL DEFAULT '0.1',`
	`  `block_id` varchar(10) NOT NULL DEFAULT '',`
	`  `block_version` float(10,1) NOT NULL,`
	`  `mandatory` tinyint(1) NOT NULL DEFAULT '1',`
	`  PRIMARY KEY (`id`)`
	`) ENGINE=InnoDB DEFAULT CHARSET=utf8;`

Texts	`-- Create syntax for TABLE texts`
	`CREATE TABLE `texts` (`
	`  `id` varchar(10) NOT NULL DEFAULT '',`
	`  `version` float(10,1) NOT NULL DEFAULT '0.1',`
	`  `mandatory` tinyint(1) NOT NULL DEFAULT '1',`
	`  `content` text NOT NULL,`
	`  PRIMARY KEY (`id`, `version`)`
	`) ENGINE=InnoDB DEFAULT CHARSET=utf8;`

Assignments	`-- Create syntax for TABLE assignments`
	`CREATE TABLE `assignments` (`
	`  `id` varchar(10) NOT NULL DEFAULT '',`
	`  `version` float(10,1) NOT NULL DEFAULT '0.1',`
	`  `mandatory` tinyint(1) NOT NULL DEFAULT '1',`
	`  `name` varchar(255) NOT NULL DEFAULT '',`
	`  `variable_type_id` int(11) NOT NULL,`
	`  `value` varchar(255) DEFAULT NULL,`
	`  PRIMARY KEY (`id`, `version`)`
	`) ENGINE=InnoDB DEFAULT CHARSET=utf8;`
	
	`-- Create syntax for TABLE variable_types`
	`CREATE TABLE `variable_types` (`
	`  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,`
	`  `name` varchar(255) NOT NULL DEFAULT '',`
	`  `rule` text DEFAULT NULL,`
	`  `max_length` int(11) DEFAULT NULL,`
	`  PRIMARY KEY (`id`)`
	`) ENGINE=InnoDB DEFAULT CHARSET=utf8;`

Variable Retrieval	`-- Create syntax for TABLE variable_retrievals`
	`CREATE TABLE `variable_retrievals` (`
	`  `id` varchar(10) NOT NULL DEFAULT '',`
	`  `version` float(10,1) NOT NULL DEFAULT '0.1',`
	`  `mandatory` tinyint(1) NOT NULL DEFAULT '1',`
	`  `assignment_id` varchar(10) NOT NULL DEFAULT '',`
	`  `assignment_version` float(10,1) NOT NULL,`
	`  PRIMARY KEY (`id`, `version`)`
	`) ENGINE=InnoDB DEFAULT CHARSET=utf8;`

Choices	`-- Create syntax for TABLE choices` `CREATE TABLE `choices` (`

	`id` varchar(10) NOT NULL DEFAULT '', `version` float(10,1) NOT NULL DEFAULT '0.1', `mandatory` tinyint(1) NOT NULL DEFAULT '1', `value` varchar(255) NOT NULL DEFAULT '', PRIMARY KEY (`id`, `version`)) ENGINE=InnoDB DEFAULT CHARSET=utf8; -- Create syntax for TABLE choice_options CREATE TABLE `choice_options` (`id` int(11) unsigned NOT NULL AUTO_INCREMENT, `choice_id` varchar(10) NOT NULL DEFAULT '', `choice_version` float(10,1) NOT NULL, `value` varchar(255) NOT NULL DEFAULT '', PRIMARY KEY (`id`)) ENGINE=InnoDB DEFAULT CHARSET=utf8;
Conditional blocks	-- Create syntax for TABLE conditional_blocks CREATE TABLE `conditional_blocks` (`id` varchar(10) NOT NULL DEFAULT '', `version` float(10,1) NOT NULL DEFAULT '0.1', `mandatory` tinyint(1) NOT NULL DEFAULT '1', `value` varchar(255) NOT NULL DEFAULT '', PRIMARY KEY (`id`, `version`)) ENGINE=InnoDB DEFAULT CHARSET=utf8; -- Create syntax for TABLE conditional_block_options CREATE TABLE `conditional_block_options` (`id` int(11) unsigned NOT NULL AUTO_INCREMENT, `conditional_block_id` varchar(10) NOT NULL DEFAULT '', `conditional_block_version` float(10,1) NOT NULL, `block_id` varchar(10) NOT NULL DEFAULT '', `block_version` float(10,1) NOT NULL, PRIMARY KEY (`id`)) ENGINE=InnoDB DEFAULT CHARSET=utf8;