# Implementation of the application

The application is structured by different packages. Each of them providing a specific benefit to the program as a whole. Before speaking about those modules, we will talk about the actual requirements of the application. After that, each module will be explained in detail and how it works. After that, the last section will deal with problems and possible solutions.

## Requirements definition

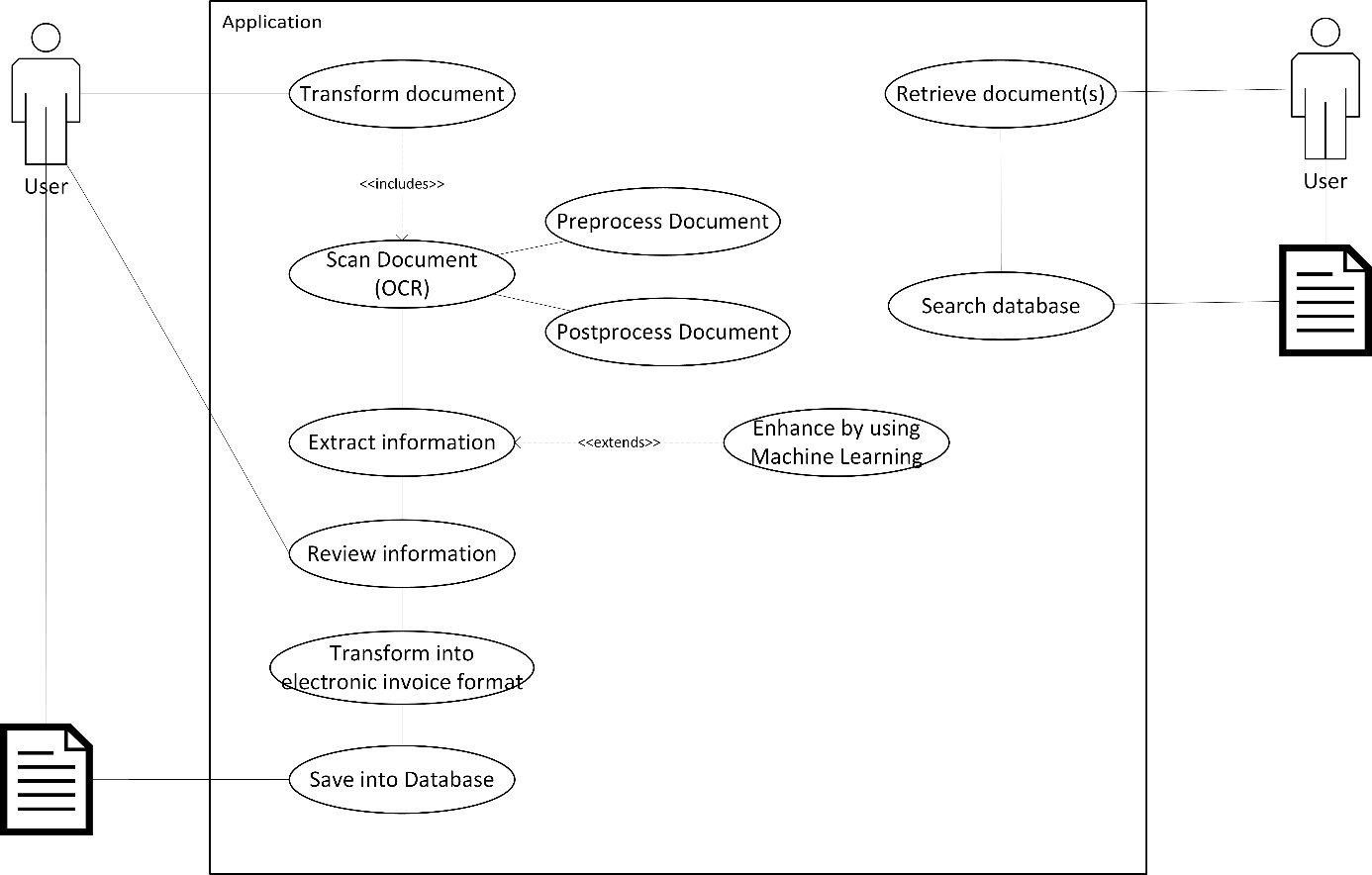
Focus of this application is the possibility to automatically process forms and retrieve the data of the forms. Hence the application should be able to deal with several files and process them without human help. But, since there is a big variety of forms and every company have different structures, retrieving all necessary information can fail. If this happens, a user has to review scanned documents that contain errors. If it doesn't fail, data should be stored without any additional help.

To improve the process of gathering data, a machine learning approach should be implement that facilitates retrieving data and to speed-up form processing over time.

Output of the application should be a storage of the processed forms, appended with electronic invoice information that is valid against the basic- or comfort-level of the ZugfErd-Invoice standard.

## Use Cases

In order to specify exactly what the application should deal with, we will introduce use cases. The following image shows the use cases and how users can interact with the application:



Output of both processes is a document. Either the document that has been transformed into an electronic invoice, or one or more documents that have been stored in the database and that have been requested by the user.

The following part will now explain each use case in detail.

Transform document: If the user has an invoice to be transformed, he can enter the application and start the progress of transforming it. This will call several other cases until the user retrieves the transformed document.

Scan Document (OCR): The actual process of reading the document will take the invoice document and process it using OCR techniques. The progress of scanning documents does also include the preprocessing of the document as well as postprocessing it.

Preprocess Document: In order to retrieve the best output, the quality of the document should be improved before. As colour does not matter for the reading, we can greyscale the picture. Tesseract works best on 300 dpi images (TODO: Link), hence we can resize the image to the default pixel values of a DIN A4 paper on 300 dpi quality. Furthermore, other techniques like despeckling, noise reduction and deskewing will be applied on the image as well.

Postprocess Document: After the document has been scanned, it is very much likely that there are still some errors left. In this step, we will go through the document and try to improve the result by searching for known keywords that usually appear in invoices (such as “Invoice”, “Invoice date” or “Skonto”). If we find any of those keywords but slightly different or with some errors in it (for example “Inv0ce date” or “Skont.”) we will replace this word with the keyword in the list.

Extract information: This is the actual extraction part which covers the extraction of invoice information (debitor and creditor, grand total, skonto) as well as accounting record information (which positions are in the invoice? To which accounts are they related?). The exact process of information retrieval will be covered by the corresponding module part.

Enhance by using Machine Learning: As the invoice information are extracted, we want to improve the process by using Machine Learning. We are using a naïve bayes approach that learns about the positions and which accounts are involved in the accounting. Furthermore, we are using a word net in order to gain knowledge about string-siblings, which will enable us predictions on new cases that are not stored yet.

Review Information: Now the user gains the possibility to interact with the application. All files will be listed in a table. If any problems occurred during the process of information retrieval (such as missing information, the likelihood of the classification is below the user-defined level etc.) it will be showed to the user. Every document that lacks obligatory information for the transformation progress has to be manually revised by the user. However, if the user decides to save the documents, all documents that are able to be transformed will be saved.

Transform into electronic invoice format: Every document that is now

## Definition of modules

Following the Separation of Concerns principle (SoC) we want to separate all logical parts of the application into different modules.

What the application will do is to take an invoice, read it (1), extract the information out of it (2), improve the information by using machine learning techniques (3) and eventually convert it to the ZugFerd-format (4).

Hence we will define four different modules:

1. OCR: After the user has passed an document to the application, this module will process the document and read it using OCR techniques. Therefore, this module will be named \"OCR\".

2. Extraction: This module will deal with the business logic regarding the retrieval of information from the processed document. Therefore, it will also give input and get output from the third module.

3. ML: In this module we will implement methods to improve future information extraction.

4. Transformation: Eventually, the extracted information and the processed document will be transformed into a new electronic invoice that is conform with the ZUGfERD-format.

5. GUI: In order to facilitate the process of entering and retrieving invoices, another module will be used that deals with all sorts of user interaction. As this application will have an graphical user interface, we will call this module GUI.

## Architectural concept

The application will make use of several design patterns. One used pattern on an architectural level is the MVC-pattern. Hence the graphical user interface will be steered by a controller which retrieves data from the database and shows it to the user using javaFX and .fxml-Files. Input and changes the user makes in the view will be transported by the controller to the model which is stored in the database again.

To access the database we will use classes for each business object. The package BO contains classes that represent a table. A data-access-object (DAO) will be used to retrieve data from the database. To do that, this application will also make use of an object-relational mapping framework (Hibernate) which facilitates the conversation between table data and java objects.

## Module 1 - OCR

The OCR module deals with the processing of the document. Therefore, we will use Googles Tesseract as described in chapter 3. In order to use it, we use Tess4J as a Java wrapper. TesseractWrapper.java is the class that initiates a tesseract instance. With initOcr() the tesseract instance is getting called. It returns a String as result.

We set HOCR to true, which means that our output will not only be a String containing the processed words, but in a structured way. HOCR is a xml-structured document first proposed by (TODO: CITE). Using this output we are not only able to retrieve the processed words, but also their position in the document.

The package hocr contains necessary java classes to represent this document in an objective-oriented way. The string output of the TesseractWrapper class can be given to the constructor of the HocrDocument class, that completely parses the string and divides it into multiple HocrAreas, HocrParagraphs, HocrLines and HocrWords.

Before the actual step of processing the image, we want to improve its quality. Therefore, we use the ImagePreprocesser class. Any kind of document inserted will first converted to a BufferedImage. Then preprocess() can be called which executes multiple algorithms on the image:

public BufferedImage preprocess() {

try {

…

BufferedImage outputFile = this.resizeImage(image);

…

outputFile = this.adjustDPI(image);

…

outputFile = this.deSkewImage(image);

…

outputFile = this.greyScaleImage(image);

…

outputFile = this.deSpeckleImage(image);

…

return outputFile;

}

…

}

Most of those calculations are made using ImageMagick, a powerful open source library with several useful commands to apply on images. It is licensed under the Apache 2.0 license. In order to use it inside our application we are using IM4Java which is cited by ImageMagick itself (here: <https://www.imagemagick.org/script/develop.php>) and is licensed under the LGPL license.

In order to increase the performance of the application, we want to be able to perform the optical character recognition by using multiple instances of the tesseract at the same time. Hence we need to implement the Runnable interface provided by the JDK. Seen from the outside, the TesseractWrapper class is just the Tesseract instance itself. So we need a worker class that can be given to a new Thread. The TesseractWorker class implements this interface. When we start a new Thread using start(), the run()-method of this worker is called internally. Run initiates a new Tesseract instance and executes OCR with the given ocr file:

/\*\*

\* Executes tesseract ocr using a wrapper

\* The result can be obtained using the getResultIfFinished() method

\*/

@Override

public void run() {

TesseractWrapper wrapper = new TesseractWrapper();

if (this.imgToScan == null) {

this.result = wrapper.initOcr(this.fileToScan, runWithHocr);

} else {

this.result = wrapper.initOcr(this.imgToScan, runWithHocr);

}

Logger.getLogger(this.getClass()).log(Level.INFO, "Finished OCR");

}

Since we want to be able to support not only pdf documents, but also images, we have to differentiate between this two. Depending what type of document, we have to parse it differently in order to get a BufferedImage out of it.

Postprocessor:

## Module 2 - Extraction

The core class that extracts the information from the hocr document is the DataExtractorService class. As we also want to retrieve information as fast as possible, we want to run it on different threads, so that we can extract the invoice information part on one thread and the accounting records information on another. Hence this class needs to implement the Runnable interface. When instantiated, a flag is set if this thread should extract the former or the latter:

@Override  
**public void** run() {  
 …  
 **if** (**this**.**extractInvoice**) {  
 **this**.**threadInvoice** = **this**.extractInvoiceInformationFromHocr();  
 } **else** {  
 **this**.**threadRecord** = **this**.extractAccountingRecordInformation();  
 }  
}

We will now start explaining the extractInvoiceInformationFromHocr() method in detail before continuing with the explanation of the extractAccountingRecordInformatio() method.

As we built our invoice information extraction process on similar invoices of the same creditor, the extractInvoiceInformationFromHocr() method starts with a search for the creditor:

…

result.setCreditor(**this**.getLegalPersonFromDatabase(

**this**.getHocrDoument(), **true**));  
**if** (result.getCreditor() != **null**) {  
 result = **this**.getCaseInformation(result);  
} **else** {  
 String invNo = **this**.findInvoiceNumber();  
 result.setInvoiceNumber(invNo);  
 result.setIssueDate(**this**.findIssueDate());

result.setDebitor(**this**.getLegalPersonFromDatabase(  
 **this**.getHocrDoument(), **false**));  
}

…

If we are not able to find the creditor in the database (because there was no invoice of this creditor yet) we will continue by searching for necessary invoice information by hand. This will be covered after the case information retrieval.

If a creditor is found, we get the case information of the corresponding creditor. A *DocumentCase* consists of a creditor to which it belongs as well as a keyword which relates the *DocumentCase* to one of the following:

* Document type: The *DocumentCase* contains information where to find a keyword that defines the document as an invoice, a proforma invoice or a credit note.
* Invoice number: ­­The *DocumentCase* contains information where to find the corresponding invoice number of the invoice.
* Invoice date: The *DocumentCase* contains information where the invoice date is being placed on the document.
* Creditor: The *DocumentCase* contains information where the name of the creditor usually is. This is being used for new documents that are not classified yet in order to improve the recognition of creditors.
* Debitor: The *DocumentCase* contains information where the name of the debitor usually is.

Besides the keyword and the creditor, there is also the position stored where one of those keywords can be found, as well as the creation date of the DocumentCase, which is being used so that newer cases get a higher priority. This way we can react on changing designs for example when a company decides to restructure their invoice documents.

In addition to that, a case id clusters all *DocumentCases* that are created on one document. With five keywords at hand, a maximum of five *DocumentCases* should be related to one document.

A flag *isCorrect* is also existing but set to false in the beginning. After the user has reviewed missing information and wants to store the revised documents, the case is compared with the given information. If there are no changes, we expect the case to be correct. Hence at this time we set *isCorrect* to true.

The getCaseInformation() method first retrieves all cases from the found creditor. Then, it sorts them to the corresponding cases.

For each keyword the corresponding cases contain position information of older documents where the keyword has been found. With that position at hand, the current HOCR document is being searched for a value at that position. The method findInCase() deals with this process:

**private** HocrElement findInCase(List<DocumentCase> cases) {**for** (DocumentCase docCase : cases) {  
 **if** (docCase.getIsCorrect()) {  
 String[] position = docCase.getPosition().split(**"\\+"**);  
 *// 0: startX, 1: startY, 2: endX, 3: endY* **int**[] pos = **new int**[] {

Integer.*valueOf*(position[0]),

Integer.*valueOf*(position[1]),

Integer.*valueOf*(position[2]),

Integer.*valueOf*(position[3])};  
  
 HocrElement possibleArea = **this**.**document**.getPage(0)

.getByPosition(pos, 50);  
 **if** (possibleArea != **null**) {  
 HocrParagraph possibleParagraph = (HocrParagraph)

possibleArea.getByPosition(pos, 30);  
 **if** (possibleParagraph != **null**) {  
 HocrLine possibleLine = (HocrLine)

possibleParagraph.getByPosition(pos, 30);  
 **if** (possibleLine != **null**) {  
 HocrWord possibleWord = (HocrWord)

possibleLine.getByPosition(pos, 10);  
 **if** (possibleWord != **null**) {  
 **return** possibleWord;  
 } **else** {  
 *// refine to multiple words, pixel threshold only a few pixels since we are searching for word* possibleWord = possibleLine

.getWordsByPosition(pos, 10);  
 **return** possibleWord;  
 }  
 }  
 }  
 }  
 }  
 }  
 **return null**;  
}

We are only using the cases that have the flag *isCorrect* set to true. Then we compare all HocrElements in the document with the stored position. But, as there could also be some small differences (e.g. because the scans are hand-made and the document has not been placed on the exact same position every time) we apply a threshold value. Every element that is more or less consistent with the given position will be returned. Eventually, we will a word that matches the position, or, if the position stored contained multiple words, a combination of words. Those are concatenated and returned. If any of those steps fail, the method will return null.

This is repeated for each keyword. A new DocumentCase is created and the position added. Every keyword that has not been found will result in missing DocumentCases. After that, the invoice filled with the retrieved information will be returned.

As mentioned before, if we are unable to find a creditor, then we proceed with the document manually. Which means we are looking for keywords such as “Rechnungsnummer” (invoice no.) or “Rechnungsdatum” (invoice date) which are usually followed by the corresponding value. This is a fallback practice and will yield more errors due to missing position information. An invoice object with the found values will be returned all the same.

The extractAccountingRecordInformation() method deals with the problem of information retrieval with a different approach: It uses the extracted table information if a table has been found (TODO: Include in text). If not, the HocrDocument is searched for keywords that are usually appear in invoice tables. If we find those information, we iterate over the following lines until we find table end information, such as “Gesamtbetrag” (total value), “Lieferdatum” (delivery date) and others. Both, the table header words as well as table end words are stored in two textfiles (tablecontents.txt and tableendings.txt) which allows the user to add more words to improve the accuracy.

Now, every line will be processed the following way:

Record r = **new** Record();  
String recordLine = **this**.removeFinancialInformationFromRecordLine(nextLine);  
**double** value = **this**.getValueFromLine(nextLine);  
  
Model m = service.getMostLikelyModel(recordLine);**if** (m == **null**) {  
 r.setEntryText(nextLine);  
} **else** {  
 r.setEntryText(m.getPosition());  
 r.setRecordAccounts(m.getAsAccountRecord(value));  
 r.setProbability(m.getProbability());  
}  
records.add(r);  
index++;

We first want to remove all those additional information from the position so that we are able to store / retrieve it if it comes again more precisely. This is done by the removeFinancialInformationFromRecordLine() method. After that, we also retrieve the total amount of the position by searching in the line again for the financial information, but this time searching for the last numeric value that is proceeded by “EUR” or “€”.

After that, the machine learning module is called. What exactly happens there will be covered by the next section. We will retrieve a possible Model that applies to our position. We can assign the found value to every involved account as the Model also contains the percentual values of each account and add a probability value to the Record which will later presented to the user in order to facilitate his decision if the automatically made decision is correct or not.

## Module 3 – ML

## Modue 4 – Transformation

## Module 5- GUI

## Problems during the implementation