A Web Service for Annotation of Documents

Master's Thesis

Henrik Bartholdt Sønder

Kongens Lyngby 2013
M.Sc. 2013-109

DTU Compute,
Technical University of Denmark
Matematiktorvet, Building 303B
DK-2800 Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@compute.dtu.dk
www.compute.dtu.dk

Abstract

This master's thesis describes the design and implementation of a web-based service for viewing and annotating documents. The main purpose of the system is to allow authenticated users to view, comment and annotate documents, to support the discussion of reports, papers and theses. The vision for the system is to provide the necessary utilities and be as cost-effective about it as possible, in the inevitable process of future maintenance and develop of new features. The main priority of the project is therefore not to develop a fully functional production-ready system, but rather to design a system that is able to most effectively provide the necessary features in a manner which also compliments a well-designed system with a high quality of code. To properly assess the quality of code in the system, a model to support this assessment in the context of the system vision is developed. The model puts an emphasis on ensuring the maintainability and adaptability of the solution, and this is used to guide and support the design considerations throughout development of the system.

The result is a highly maintainable and extensible design, where all system-critical choices are based on thorough analysis and included in the thesis to support the assessment of their precision. The design and implementation of each system-critical component is covered with an emphasis on explaining the important design considerations made during development. Testability of the system is an important factor as well, and the testability of the final design will be discussed and exemplified using mock objects as well as supportive "system under test" builder objects for a highly configurable test environment. A thorough assessment of the security of the system is featured as well, testing the system against the top ten most critical flaws in web application security, as provided by the OWASP organization.

I believe the design of the system satisfies the necessary criteria for success, and the emphasis on cost-effective maintenance and further development should ensure the future success of this system. To definitively conclude this is difficult at best, but the contents of this thesis should provide an adequate assessment on the success of my efforts.

Preface

This master's thesis ”A Web Service for Annotation of Documents” was conducted in the period from 22nd of February 2013 to 13th of September 2013 at the Department of Applied Mathematics and Computer Science, Technical University of Denmark. The work corresponds to 35 ETCS points.

The project was conducted under the supervision of Associate professor Christian W. Probst to whom I owe great thanks for excellent guidance and interest in my project. Additionally, I would like to thank my loving family and friends for supporting me when really needed.

Henrik Bartholdt Sønder, s072655
Technical University of Denmark
13th of September 2013

Table of Contents

1 Introduction

This thesis covers the development of a web service for annotation of documents. The development of the system was proposed by Associate Professor Christian W. Probst from the Technical University of Denmark, answering my request of potential topics for a master's thesis in the subject of software development.

The main purpose of the system is to allow authenticated users to view, comment and annotate documents, to support the discussion of reports, papers and theses. The initial set of functional requirements is quite simple, but some requirements do present technological difficulties.

In particular, the process of rendering and storing annotations should be carefully considered. It is apparent that the system should be developed with consideration for possible future developments as well, and some suggestions for more advanced features have already been received. One such feature is the detection of hand-drawn annotations on real paper, meaning that the system should support the process of detecting hand-drawn annotations by analysing shapes in an image of a scanned piece of paper. While this is surely an interesting feature to develop, the primary focus has been kept towards ensuring a well-designed system.

In regards to the design of the system, it will be essential to ensure the potential for a cost-effective approach to the future maintenance of the system. The only utility of the system so far is to support annotation of documents, and if the system has high maintenance costs, as well as low utility, the fact that it is non-profit as well could easily result in its discontinuation. The main priority of the project is therefore not to develop a fully functional production-ready system, but rather to design a system that is able to most effectively provide the necessary features in a manner which also compliments a well-designed system with a high quality of code.

2 Requirements

This section covers the simple list of functional requirements for the system. The primary focus of the project is to ensure proper development of the non-functional requirements. A proper definition for what these few terms actually mean for the solution requires a more thorough analysis though, so for now this tiny list is the actual system requirements.

2.1 Users

- Users can register with a DTU email address
  - Such users have upload privileges per default.
- Users can register with an open authentication method
  - Such users have no upload privileges, initially.
- Users can share their own documents with other individual users.

2.2 Annotations

- Users can annotate documents using text comments and simple graphics.
- Users can reply to annotations with text annotations, essentially supporting discussions for each annotation.
- Users can upload documents.
  - Uploaded documents will be stored online.
  - Uploaded documents will have to detect and support existing annotations, and convert them if needed.
- Users can add all annotations from an existing local pdf file to an existing online document.
  - Users will be able to upload a pdf file, extract annotations from it and have the annotations added to an existing online document.

2.3 Non-Functional Requirements

- Cost-effective Maintenance
- Extensibility
- Adaptability

3 Analysis

This section will cover the analysis made during the early stages of development, where the primary concerns are to investigate and research the areas that are most critical to the success of this development.

The first section of this chapter will present and briefly discuss these critical areas and their purpose in this project, and these areas will provide the general structure of the rest of the analysis, as these areas are analyzed in detail in their separate sections.

The sections of this chapter will constitute an analysis of the viable solutions for annotating, rendering and managing documents, as well as discuss the assessment of quality of software in the system.

3.1 System-Critical Areas

This section covers an analysis of the most critical features of the application, to assist in planning the design and development of system accordingly. The areas are either essential components or more abstract requirements that should be thoroughly investigated. It will be a key priority to develop or at least conceptually prove areas related to functional requirements and system utility. Areas related to non-functional requirements will have to be properly assessed and quantified to provide some degree of measure for their successful development. A thorough investigation of these critical areas should help assess, and hopefully also ensure, the future success of this system.

The critical areas listed below will initially be developed or at least conceptually proven, and the most optimal solution to each area will be carefully considered.

- Adding annotations to documents
    - Creating and editing annotations
    - Synchronizing changes for multiple users
- Viewing documents and annotations
    - Updating the view when users annotate
    - Updating the view when collaborators annotate

- Managing documents and annotation data
    - Organization of documents and annotations
    - Access control and availability
- User experience
    - Intuitive design
    - Responsive interface
    - No-lag, asynchronous methods

This may initially function as a list of things to conceptually prove, but it should not be viewed as such; simply providing these critical features is not the goal of this project. The goal of this project is to research and develop a system able to most effectively provide these features in a manner which compliments a well-designed system with a high quality of code. As such, an analysis of how the quality of code should be assessed and how the system should be designed towards this goal will have to be provided as well.

3.2 Annotating Documents

This section covers one of the first analyses made: investigating the capabilities of existing software solutions and the different approaches used to successfully annotate documents. During the research of existing systems for document annotation, two main approaches were observed, and these will be analyzed in the context of the requirements of this system. The analysis ends in a very one-sided decision towards one approach, which will likely be obvious throughout the analysis. The analysis of the alternative option was kept even so, as several subtle strengths of the winning option were not as easily described without a second-best option of which to compare.

3.2.1 Existing Solutions

Software solutions for annotating various types of documents already exist, and they generally take one of two different approaches: One approach supports annotation by editing the document itself, while another approach annotates documents by applying document independent annotations on top of a read-only document. These two categories of solutions will now be referred to as document-dependent and document-independent, respectively.

To name a few successful applications in the document-independent category there is “A.nnotate”[1], “Crocodoc”[2] and “Mendeley”[3]. Mendeley is originally a reference manager for research papers, but it does support a few features for annotating documents and as such fits the description of an annotation application as well. The approach used by these tools do not require editing of the document itself, as they support annotation by annotating on top of other documents using independent annotations. As such, these annotations are naturally not stored in the document itself, and will have to be provided through some others means, such as a separate file or a web service connected to the client. The client application will then be responsible for displaying the annotations correctly on top of the document, along with other features. The fact that this approach does not require editing of the document is what distinguishes it from the document-dependent approach, and the document can effectively remain read-only. Most of these document-independent tools will be able to merge their annotations into a document at some point in time though, to be able to provide a user with an annotated document, but it is important to note that this approach does not require the application to actively edit the document each time an annotation is created or edited.

To name a few solutions in the document-dependent category there is the “PDF Annotator”[4] and the perhaps more commonly known “Adobe Acrobat”[5] PDF editor. These tools will easily annotate a document in many different ways and their features extend far beyond just annotating. The reason I refer to them as *document-dependent* is because they support annotation of documents by creating and editing annotations inside the document itself. This necessitates changes in the document itself every time an annotation is added or editing, and it also makes the solutions inherently dependent on the PDF format for any kind of editing or annotation. It is also worth noting that there are few, if any, online web applications using this approach for document annotation. This also seems natural though, that editing a PDF document might commonly be a job for

[1] A.nnotate:	http://a.nnotate.com/
[2] Crocodoc:	https://crocodoc.com/
[3] Mendeley:	http://www.mendeley.com/
[4] PDF Annotator:	http://www.grahl-software.com/en/pdfannotator/
[5] Adobe Acrobat:	http://www.adobe.com/products/acrobat.html

a desktop application, as this is common for most applications dealing with a lot of file manipulation.

3.2.2 The Document-Independent Approach

An initial assessment of the document-independent approach suggests that it has a lot of advantages. Several of the existing solutions are not only web-based, but also browser-based, meaning this approach allows for solutions that are provided entirely through a website, without installing any desktop application or browser plugins. The data required for all the annotations will have to be stored separately though, but since we are providing the solution as a web service to synchronize data anyway, this is not a problem at all – Quite the contrary actually. This separation between a read-only document and the collaboratively updated annotations will likely simplify the synchronization process a lot, especially in regards to continuously redrawing annotation updates. If we are not changing the document and simply adding an annotation of top of it, it should be possible to update this change by redrawing the annotations separately. In regards to viewing the document we are free to either use a plugin or convert the PDF to some other format - We just need to display a read-only document, and display the annotations on top. Being free to choose any method of displaying a PDF document should also further increase the probability of finding a good solution for updating the view correctly. Having annotations stored separately also gives an entirely different level of extensibility to the system: it could choose to present these annotations in alternative ways, customize the access controls for viewing them or even allow for account- or system-wide search on annotation content.

It is already apparent that the document-independent approach has some very attractive properties, but we will take a look at the other approach as well.

3.2.3 The Document-Dependent Approach

It is clear that taking a document-dependent approach has a number of immediate disadvantages in the context of our application requirements. Annotating documents by editing the document itself introduces a lot of potential errors which are avoided entirely if the documents are kept read-only. Direct editing of the document might be preferred if the solution required that the rendering of these annotations had to be exactly correct, but exact rendering is not much of a concern for these requirements. Editing a PDF document directly through a website might be possible, but a desktop application would probably be a better option. This approach is already awfully complicated compared to the alternative document-independent approach, especially when the options for synchronization are considered in the context of multiple directly manipulated PDF document files. Yikes.

3.2.4 Synchronizing Annotation Data

Having multiple clients annotate the same document at the same time presents a number of challenges beyond that of simply adding the annotations themselves. The solution has to ensure the integrity of the document at all times, and some degree of version control will have to be applied to ensure this. This will commonly be accomplished by locking sections of the document as users add or edit them, to ensure that others do not attempt to edit the same sections at the same time. Even with this in place there will likely still be race conditions to take care of, such as two users trying to edit the same section of a document at the exact same time.

An important factor in the difficulty of handling version control and race conditions for a source of data is how well structured that source of data is in regards to adding and editing data. For instance, it is most often not that difficult to ensure the integrity of data stored on an SQL server and its rows of data, as we can easily ensure that fields for 'LastModifiedDateTime' are validated and checked against the current time each time we attempt to update an entity. A data store with a data entry for each annotation would provide a nice degree of granularity, where any action create, edit or delete would only affect a single annotation entity. This degree of version control would not be difficult to support, and it would also reduce the possible scenarios for race conditions to the ones where two users actually edit the same annotation at the same time. This would likely be the way to go if we were to support document-independent annotations.

With this in mind, ensuring the data integrity of document-dependent annotations would be another matter entirely. We do not have the same fine degree of granularity in PDF documents, and adding, deleting or editing an annotation directly in the document would have ripple effects throughout the document; we would have to add or edit entries in the body element as well as the cross-reference table and trailer elements. We can also choose to use incremental updates and simply append any changes to the end of the PDF file, meaning that we would append another body, cross-reference table and trailer element to the end of the file every time an annotation was added, changed or deleted. This has the added benefit of providing a clear history of document annotations, as well as the disadvantage of having to download this history just to view the document. Then again, if annotations are rarely edited or deleted, this will not be a significant disadvantage in itself.

3.2.5 Conclusion

The existing solutions for annotating documents use one of two approaches; they are either dedicated PDF editors with tools for all kinds of PDF manipulation, or dedicated annotation-tools able to annotate documents more or less independent of the document file itself. The choice of which approach to embrace is very one-sided even at this early point in the analysis, as the advantages of the document-independent approach are overwhelming given the requirements of this solution.

3.3 Rendering Documents

This section covers an analysis of the available options for rendering and displaying documents and annotations. The methods to consider will be listed below and the following sections of the report will feature an analysis of each of the methods in their separate sections.

After an initial research of available methods for rendering PDF documents, the main methods to be considered are the following four:

- Using a browser plugin designed specifically for PDF files.
    - E.g. Adobe Acrobat Reader.
- Using Flash, a browser plugin.
- Using Adobe AIR, a browser plugin.
- Using HTML5.
- By converting to Images

In regards to rendering annotations, it should be possible to do so using an HTML overlay on top of the PDF viewer, and this should be possible for all the above methods as well. Flash and Adobe Air might benefit from having annotations rendered through their methods of rendering though.

3.3.1 Rendering using a PDF Browser Plugin

The first attempts at a proof of concept implementation using a PDF browser plugin was successful, in that a PDF document was loaded and displayed in a so called iframe element in HTML. This was done very easily, using only the following section of code on a web page:

```
<iframe src="http://samplepdf.com/sample.pdf"
        width="800px" height="600px" >
```

One fairly big problem encountered in this method of rendering is that the plugin used to render the PDF in the iframe will be decided by the client browser. Each browser handles PDF documents differently and even if one could manage to support a number of major browsers, a user might still run into problems by using a supported browser but have some unsupported PDF plugin installed. There are security concerns for this method of rendering as well, as plugins using native code introduce a whole new category of possible vulnerabilities to the system. In order to keep security up to date, regular updates to the plugin would also to be taken into consideration, and continuous updates to the plugin likely require occasional maintenance.

An HTML overlay was also implemented on top of the PDF plugin displaying the PDF, e.g. on top of the iframe. The PDF plugin attempts to force itself on top of everything, but this could be circumvented. The successful HTML proved that it would be possible to draw annotations on top of the PDF plugin, and this is essential because if this was not possible the rendering of annotations would likely have to be done through the PDF plugin. Relying on the PDF plugin for manipulation of annotations or developing a custom solution to support this are both options I barely even want to consider – At least not without thoroughly investigating other options. The next step here would be to test if a decent integration between PDF plugin and annotations was possible and show that mouse events could be handled properly along with synchronous scrolling of both the document in the viewer and the annotations overlay. However, even though this minor proof of concept implementation could be considered successful there are already many potential maintenance issues on the horizon. As such, other methods for rendering were investigated before heading any further along this path.

3.3.2 Rendering using HTML5

Rendering using HTML5 is done by rendering the document as a set of HTML elements. The process of creating and styling HTML elements to correspond exactly with the intended layout of the PDF document has been developed and polished for a while now, and without referring to any actual analysis I'd say the conversions have a high level of accuracy that should be satisfying for this solution. Given that this method of rendering is based exclusively on HTML, and possibly JavaScript, this option should also be easily available to any platform supporting HTML5.

Rendering using HTML has several immediate benefits in this solution, the first being that it should integrate rather easily with an HTML overlay for rendering annotations. Additionally the rendering using HTML uses HTML elements throughout the document and a proper classification of these elements could provide a high level of information about the contents of the document at any given time. The combination of the two aforementioned benefits even makes it possible to attach event handlers directly to the HTML elements of the read-only document and have them directly interact with the annotation overlay.

A disadvantage of rendering a PDF document using HTML is that the data processing and rendering process is a lot to handle for the average browser at run-time, especially given the fact that JavaScript's dynamic nature results in quite inefficiently compiled procedures. As a result this type of rendering has commonly not been efficient enough for real-time rendering but has been restricted to pre-rendering of documents, meaning that the document is converted into HTML through a one-time conversion process. However, the potential processing capabilities of JavaScript have increased in the last few years, and the development of just-in-time type specialization for dynamic languages (Gal) in particular has improved the computational speed of a broad set of instructions by several magnitudes.

This has led to the development of PDF.js: a PDF viewer built entirely in JavaScript. PDF.js is a community driven experiment supported by Mozilla Labs, and the experiment could be considered successful as PDF.js has been the default integrated PDF viewer in Firefox since version 19, February 19 2013. Another noteworthy benefit of PDF.js is the inherent security benefits of rendering without the use of native code. This makes the rendering process a lot less vulnerable to exploits compared to the method using rendering through a PDF plugin.

To summarize, it would be possible to render documents in HTML, either by converting documents once to serve an HTML document directly or by rendering the PDF document as HTML through PDF.js. It would also make the rendering process itself fairly secure, as no native code is necessary, and provide a high level of available across all platforms with HTML5.

3.3.3 Rendering using Images

Rendering using images is done by pre-rendering the document as a set of images. This has the advantage of keeping the process of displaying the document very simple and consistent across all platforms. The document will no longer be able to feature vector based graphics, and the process of rendering might have to be customizable in regards to quality and resolution to be able to satisfy the needs of a broad range of users.

A fairly big disadvantage of rendering a document as a set of images is the lack of support for kind of object recognition or interaction in the rendered document. This means the rendered document will not be able to support any kind of text search or selection, and other PDF features such as positional links or hyperlinks will likewise not be supported. Some existing solutions do use rendering through images though, and they work around this lack of support for text selection and search by providing a layer of text on top of the rendered image; as the document is pre-rendered they render both an image and a text overlay along with it, to be able to provide text selection and search.

In conclusion, rendering using images is a viable option. However, it would be essential to provide support for text selection and perhaps other document elements through an overlay as described in the analysis, as this is necessary to overcome most of the disadvantages of this rendering process. Similar to the rendering process using HTML, the rendering process using images should itself be fairly secure and provide a high level of availability across many platforms.

3.3.4 Rendering using Flash and/or Adobe AIR

The option of rending using Flash, perhaps assisted by Adobe AIR, is in the same category as the PDF plugin option in regards to the inherent disadvantages of a using 3rd party plugins. This requires installation of additional software, and exposes the application to an additional set of security vulnerabilities and maintenance costs. From a personal perspective it also requires that I learn at least the basics of a new scripting language, ActionScript, and I have no interest in that.

Without going into too much detail this option provides about the same benefits and disadvantages as the PDF plugin option. Using Flash for rendering might have been considered an interesting alternative to the option of rendering using a PDF plugin, but they are both at a disadvantage compared to both Image and HTML rendering. As such, it was quickly apparent that this option was not very promising, and little effort was made into investigating it further.

3.3.5 Conclusion

The chosen method of rendering is HTML5. The following few paragraphs states the decisions made based on this analysis and summarizes the facts those decisions were based on. The two options for rendering using either a PDF plugin or Flash were tested and analyzed, and a minor prototype was completed for the PDF plugin. These two options have many similarities, but they have both been discarded in favor of rendering through either HTML or Images. This is done primarily based on the disadvantages related to security, maintenance and the requirement of one or more browser plugins.

The two options for rendering using either HTML or images have many similarities as well. Rendering using images is a promising option, but HTML is the method better overall. Rendering using images inherently removes all information about the content of the document, such as text, images and other objects, as the process converts each page into a single image. To provide this information, rendering using images would require the development of an additional information-layer to display text and other object indicators on top of images. Rendering using images does not support vector based graphics.

HTML rendering inherently provides a lot of information about the content of the document through the resulting structure of HTML elements which can even be classified to provide additional object-specific information. HTML rendering will provide what is required of the viewer immediately, without any additional developments necessary. HTML rendering supports vector based graphics where applicable and fonts are displayed in crisp quality even at high levels of zoom. Rendering using HTML also provides a choice for rendering either in run-time or as a one-time conversion process when a document is uploaded.

3.4 Document Management

The requirements does not cover how the documents should be presented, but simply presenting a list of the documents the user has access to is probably not a great long-term solution. As the number of documents per user increases the need for some kind of document organization system will soon arise.

The decision regarding what would be the best set of methods for organizing documents in this solution was postponed to wait for additional input in regards to the future of this project. The analysis of the options for organizing documents will still be included though, concluding that the optimal structure for organizing documents in this solution depends a lot on how accessible and searchable these annotated documents should be.

There are several well-known and proven techniques when it comes to organizing documents or articles, and the methods we are going to analyze are organization by folders and categorization by tags. The first thing we notice in researching the subject is that we need to consider whether we want to provide organization, categorization, or both.

Organization based on folders

The folder structure is the fundamental structure of most files system, where the documents are organized in any number of folders and sub-folders. This design emphasizes a hierarchical structure where a document is contained in a single folder, and this puts a limit on the system in regards to categorization, as using folders as categories means that a document can be categorized by only a single category. The folder structure does allow for sub-folders though, effectively introducing the concept of sub-categories for documents, but it is still limited by that fact that a document can only have a single folder-based category. In conclusion this design is great in regards to organization, but lacks behind when it comes to categorization.

Categorization based on tags
Tags are a non-hierarchical keywords or terms assigned to any kind of object or set of data to describe and categorize the contents of that object. A real-world example of this would be Twitter, where hash tags are used to categorize messages so that users are able to filter the massive amount of messages based on the tags describing the content of the messages. This method of organization would categorize documents by tagging a document with any number of tags, where each tag effectively functions a category. This is a very flexible approach in regards to categorization, as it allows for a document to be included in an arbitrary number of categories. Compared to the simple and personal folder structure, organization using tags is in general less structured and more chaotic. The chaos could be limited by providing a list of categories to choose from, but ensuring a useful list will require continuous re-evaluation. The one great strength of this approach is the potential for an open and searchable library of documents, and whether or not this is wanted should be the deciding factor of this approach. While tags could be used as the primary method for organizing documents, it is also well suited as an additional tool to improve search ability in documents, while still providing a simple folder structure for personal organization needs.

A question to consider alongside with the choice of organizational structure is the scope of the structure: Should it be personal or not? Folders, groups and tags could be strictly personal, and some users would doubtless prefer this and be able to have full control over the tags on their documents and the structure of groups and folders. There are however obvious administrative advantages in being able to share folders or groups, and thereby make them not strictly personal, as this gives users the ability to effortlessly share entire collections of documents.

3.4.1 Conclusion

A folder structure provides a simple and well-known structure for document organization. The structure will provide sufficient organizational utilities for most users, and allows for some level of categorization as well. The structure could be accompanied by the structure of tags as well, in an effort to provide greater support for categorization and search.

Tags provide a high degree of categorization, and should definitely be considered if the system should support any kind of open and easily accessible platform for document access. Designing towards an open platform will have an enormous influence on the system, in particular in regards to many aspects of performance, and careful consideration should be made if this kind of system is of any interest.

The scope of the organizational structure should be considered as well. If this is considered a user centric system, a folder structure for each user might be the best solution. If this is considered a university-centric system, a single system-wide folder structure for all users might be the best solution. The folder structure could be based on some set of metrics, such as the year published along with the department from which it originated.

3.5 Assessing the Quality of Software

The quality of the software in this system is a top priority, and this section will provide a common ground for which to discuss the topic of software quality in the following design sections of the report. This section will first present and explain the current characterization of software quality, and then discuss how this model was developed.

Designing towards a goal of maintaining a high level of code quality will help ensure the future success of the application, but what code quality attributes do we need to focus on?

3.5.1 Characterization of Software Quality Attributes

The software solution designed and delivered in this thesis is a working prototype at best, and the quality of code must be considered throughout the design process to ensure that the future development and delivery of the system is successful.

After careful consideration, the model to use for quality assessment of software attributes was designed to assess the quality of code in regards to the following three questions:

- Ability to cost-effectively maintain and develop the system.
- Ability to adapt as a result of outside influences.
- Ability to provide the necessary utility, in the best possible way.

The three questions above are the basis for the three perspectives developed, and the perspectives are referred to as Maintainability, Adaptability and Utility, respectively. The details of the perspectives are listed below, and this constitutes the final model used for assessment of software quality:

Maintainability
This perspective identifies quality factors that influence the ability to maintain the system. This definition of maintainability also includes the ability to cost efficiently develop additional smaller improvement and features over time. The most critical quality factors are:

- Flexibility, the ability to make changes as dictated by the business.
- Simplicity / Understandability, the ease of understanding the system.
- Extensibility, the ability to continuously extend the system.
- Testability, the ability to cost-effectively test the system.

Adaptability
This perspective identifies quality factors that influence the ability to adapt the system to new environments. This includes adaption to critical changes as a result of outside influences such as the introduction of new and superior technologies. The most critical quality factors are:

- Reusability, the ease of using existing software components in a different context.
- Interoperability, the extent, or ease, to which software components work together.

Functionality / Utility

This perspective identifies quality factors that influence the ability to provide its features in a way that gives the user a satisfying user experience. The most critical quality factors are:

- Utility, the extent of which the system provides necessary features.
- Usability, ease of use.
- Reliability, the extent to which the system fails.
- Integrity, protection from unauthorized access.

Having presented the model, the last part of this section will briefly cover how the perspectives are used to guide the development of this project. Having these few perspectives to consider during development has proven an effective tool in analysing the design and development of the system along the way.

Adaptability is considered additionally important in the early stages of development, in order to be as adaptable as possible when the system is most prone to change. Adaptability should always be a concern, and the design section features an assessment of component adaptability for each of the system-critical components developed.

Maintainability could be considered less of a current concern in these early stages, whereas the important thing in this perspective is that the current design decisions ensure the future maintainability of the system. Lower maintenance costs will increase the potential for further development of the system, and will at the very least provide some level of insurance that the system will not die a cruel death at the hands of too high maintenance costs. As such, any design decisions will be carefully considered in order to ensure a high level of maintainability that is well designed for future developments as well. The most critical areas of the system in terms of maintainability will feature thorough discussions of the system design and its effect on maintainability.

Utility, or functionality, is not a prominent concern for the currently implemented system, but it is of course very important to ensure that future development will be able to provide all the necessary features. The prototype developments covered in this report will therefore feature critical assessments of the future capabilities of the components they develop. Utility is not as well covered in the following design sections, but this is mostly a result of low requirements in terms of utility, such as performance and reliability which are not much of a concern. One important aspect of utility is the usability or user experience, and several of the client components have been developed with usability as a top priority. The design section will also feature several examples of the more technical requirements for user experience, the effects of which will be covered in the design section as well as the security section – Some of the technical requirements for providing a good user experience did result in additional security concerns.

3.5.2 Discussion

Designing towards a goal of maintaining a high level of code quality in the future will help ensure the future success of the application, but what code quality attributes do we need to focus on?

When it comes to software quality, the different aspects of it are commonly discussed in many different categories, such as robustness, efficiency and responsiveness. There are also many existing models of which to classify software quality, most of them modeled to best support their type of system. While many categorizations are quite similar, some amount of overlap and inconsistency in the meaning and scope of these categorizations undoubtedly occur between them. For the purpose of the following design section it would therefore be beneficial to discuss the meaning of the concept of software quality beforehand, to best ensure the existence of a common ground for this topic. This discussion states the critical aspects of software quality in the context of this system, and explains the absence of consideration for some less important aspects in the following design discussions. Before states the most critical areas, let us start the discussion by presenting a few different models for software quality, and discuss them in the context of this system.

To begin with, the FURPS model is presented below. This model was developed by Hewlett-Packard more than 30 years ago and it is designed to classify large, enterprise software solutions. We will not be discussing this model in much detail, but simply note its structure along with the fact that each of these main categories contain between 5 and 10 additional sub-categories for a total of almost 30. Naturally, a FURPS+ model has been developed as well, with additional categories.

Classifying software quality using FURPS (Grady & Casswell):

- **F**unctionality
  - Feature set, Capabilities, Generality, Security
- **U**sability
  - Human factors, Aesthetics, Consistency, Documentation
- **R**eliability
  - Frequency/severity of failure, Recoverability, Predictability, Accuracy, Mean time to failure
- **P**erformance
  - Speed, Efficiency, Resource consumption, Throughput, Response time
- **S**upportability
  - Testability, Extensibility, Adaptability, Maintainability, Compatibility, Configurability, Serviceability, Installability, Localizability, Portability

Classifying software quality according to this model could be beneficial at some point, but the categories are unnecessarily detailed at this point in development. Simpler models exist, that would serve the system better.

To exemplify, it is beneficial that the analysis makes a brief point about the benefits of the fact that our browser based application requires no installation of any plugins. However, it does improve the understandability of the discussion to be discussing this benefit in terms of it providing “a high level of installability, which benefits the level of supportability of the system”. While it is not useful for this system, such terms would surely benefit more complex systems, where installability could be quantified to a meaningful unit of measure and used to set goals for the software in regards to this aspect.

To provide a contrast to the very granular categorization of the FURPS model, the list below features a much simpler model for software quality assessment.

- Product Level Quality
    - Flexibility
    - Simplicity
    - Utility
- Code Level Quality
    - Modularity
    - Extensibility
    - Maintainability

This model is used by Drupal[6], an open source content management platform powering millions of websites around the world. I personally like this model a lot, in the context of the Drupal development, and I believe they have done a good job in this regard. The model is useful in assessing the important aspects of the quality of software in the system, but the fact that it is simple as well makes it a lot more useful as a guideline for further development. It should be clear that the FURPS model severely ruins its ability to act as any sort of guideline for further development, simply because it defines categories for just about any possible aspects of the system. In conclusion it would be preferable to either find or develop a fairly simple model that is simple enough to act as a guideline for further development, but still detailed enough to be useful in assessing the most critical areas of the system.

In the effort to develop such a model, let us take a look at another well-known model, commonly referred to as McCall's model (McCall, Richards, & Walters, 1977); shown below in Figure 1. The model is presented with a high level of detail for each characterization, but the details are not important for this discussion. The important thing to note in this model is its emphasis on separating the quality attributes into three main perspectives: Revision, Transition and Operations.

[6] https://drupal.org/

McCall's Model for Classification of Software Quality – 1977

McCall identified three main perspectives for characterizing the quality attributes of a software product:

- Product revision (ability to change).
- Product transition (adaptability to new environments).
- Product operations (basic operational characteristics).

Product revision

The product revision perspective identifies quality factors that influence the ability to change the software product, these factors are:-

- Maintainability, the ability to find and fix a defect.
- Flexibility, the ability to make changes required as dictated by the business.
- Testability, the ability to validate the software requirements.

Product transition

The product transition perspective identifies quality factors that influence the ability to adapt the software to new environments:-

- Portability, the ability to transfer the software from one environment to another.
- Reusability, the ease of using existing software components in a different context.
- Interoperability, the extent, or ease, to which software components work together.

Product operations

The product operations perspective identifies quality factors that influence the extent to which the software fulfils its specification:-

- Correctness, the functionality matches the specification.
- Reliability, the extent to which the system fails.
- Efficiency, system resource (including cpu, disk, memory, network) usage.
- Integrity, protection from unauthorized access.
- Usability, ease of use.

Figure 1: McCall's model for classifying software quality.

Compared to the FURPS model, this model takes the first four categories (Functionality, Usability, Reliability, Performance) and categorizes them all as part of the product operations perspective. The remaining category, Supportability, is then split into two perspectives: product revision and transition. It should be clear that McCall's model has less emphasis on the functional aspects of the system, and more emphasis on the non-functional aspects such as maintenance and how well the system adapts to change. Out of these two models, FURPS and McCall's, I would much prefer to use McCall's model in any discussions or assessments of the quality of software. To best explain why, let me first state a few personal opinions in regards to the development of the system.

In the early stages of system development the design and technology choices made are more likely than ever to meet unexpected, system-critical issues that cannot be easily circumvented. There are several such cases in this project alone, as several technologies for PDF rendering and data storage has been prototyped and later discarded. This could necessitate changes to the core design of components or require the introduction of alternative technologies, and the initial system design would do well to take this into consideration. An assessment of the system's ability to adapt to such changes would focus entirely on the product transition perspective in the McCall model. It is clear that the McCall model supports this assessment quite well, and compared to the FURPS model it sure has its advantages. A similar assessment in the FURPS model would focus on the Supportability section of the FURPS model, but this section contains at least ten sub-categories for supportability, many of which are not that relevant for the assessment of a product's ability to transition or adapt.

In conclusion, the concept of the product transition perspective of McCall's model supports a type of assessment that is central to the system: An assessment of the system's ability to transition or adapt to new environments. In comparison, using the FURPS model for this type of assessment would not provide any benefit; the model does not define any categorization useful for the assessment of product transition or adaptability of the system.

The two other perspectives of McCall's model provide support for two other useful assessments as well. Each of the three perspectives supports an assessment of the system, in the context of three quite intuitively asked questions:

- Revision: It the system easy to maintain?
- Transition: Is the system able to adapt to unexpected changes?

- Operation: Is the system good at what it does?

It provides an adequate answer to these questions as well, by listing three to five aspects of software quality that should be considered. Asking the question "Is the system easy to maintain?" is quite another matter in the FURPS model, as that would be answered by checking the "Supportability" section which includes:

> "Testability, Extensibility, Adaptability, Maintainability, Compatibility, Configurability, Serviceability, Installability, Localizability, Portability"

Following this analysis it was decided that McCall's model was a good foundational model to build upon, and the concept of assessing the software in the three perspectives of McCall was adapted as well.

3.5.3 Conclusion

The primary conclusion to this section, as well as the discussion, is the resulting characterization of quality attributes developed, which is presented in the introduction of this section. Besides the resulting characterization, several different models for the categorization and assessment of software quality attributes were discussed and compared. It was argued that the general structure of McCall had many benefits and the benefit of having a simplistic model was explained as well. A model for assessment of software quality attributes was designed specifically to support the development of the system. The model emphasises three perspectives of the system: Maintainability, Adaptability and Utility. The three perspectives of the model support assessments of the quality of the code in the system, in regards to the three chosen focus areas:

- Ability to cost-effectively maintain and develop the system.
- Ability to adapt as a result of outside influences.
- Ability to provide the necessary utility, in the best possible way.

The quality of software characterization developed is primarily based on McCall's three perspectives of Product Revision, Product Transition and Product Operations. The naming of these perspectives was changed to Maintainability, Adaptability and Utility respectively. The sub-categories of quality attributes for each perspective was changed slightly to better fit this development, as the first sub-section of this chapter show, but the core idea behind the three perspectives was kept largely the same.

3.6 Conclusion

A selection of system-critical areas to analyze is developed initially, and this selection provides the basic structure for the rest of the analysis. The selection of critical areas developed in this stage is one of the primary concerns of this report, and the areas related to functional requirements are to be developed or at least conceptually proven. The areas related to non-functional requirements will be subject to critical assessment during development, and methods to best measure these qualities and develop the system accordingly was analyzed and developed to support this. Simply providing solutions to the critical areas is not the goal, and the areas covered will be subject to thorough research and consideration throughout the development of this application. This should ensure the development of a system that is able to most effectively support these critical areas, in a manner which compliments a well-designed system with a high quality of code.

In order to provide a high quality of code in the system, a method and model supporting assessments of the quality of software in the system was analyzed and developed. The model supports assessment of the system in three different perspectives: Maintainability, Adaptability, Utility. The perspectives provide an effective and consistent assessment the system and effectively act a guideline for development of non-functional requirements throughout the design and development process.

Viable solutions to the critical areas of the system are analyzed and the most optimal solutions are discussed, argued and chosen for further development. It was decided that annotations were best supported using a document-independent approach, where the annotations for documents is stored separately from the document. This has immediate inherent benefits in regards to synchronization, in particular considering the not-well-designed-for-editing nature of the PDF format.

Using a document-independent approach, it was decided that the solution would benefit from separate rendering of annotations as well. HTML is as a good initial option for rendering annotations, as it would support most rendering methods. As such, it was chosen to ensure a high level of adaptability.

To support rendering of documents, HTML5 was chosen as the best option. To assist in this process a library for rendering PDF files in HTML5 at run-time, PDF.js, was the initial choice for rendering the document. It was also decided that the option for exchanging this viewer should be carefully considered, as the rendering method using one-time HTML conversion of documents was considered almost on-par with run-time rendering of HTML5.

4 Design: The Client

This section will cover the layout of the client-side of the solution and explain cover the set of components most critical to the client. The components developed will be covered and their purpose in the solution will be explained as well as their design intent and implementation.

The following list presents a brief view of the client as seen from the user's perspective, in that these are the three main pages used by the client in the most common use-cases of the application.

- **Main Website**
    - Login and site navigation.
    - Registration and account management.
    - Could also provide:
        - Related news on the front page.
        - A help page with instructions.
- **Documents Page**
    - List documents
    - Manage documents
    - Open documents in the viewer
- **Document Viewer Page**
    - Renders PDF documents using PDF.js.
    - Renders annotations separately, or through PDF.js.
    - Provides tools for managing annotations.

The following sections will each feature a discussion of the components developed, where the three main components are the documents page as well as the two extensions made to the viewer. The topic of JavaScript and its effect on the solution will briefly be covered first though, in the following section.

4.1 JavaScript

This section features a brief discussion of JavaScript in general and how it currently influences the system. JavaScript is an essential part of this solution, as it is used extensively throughout the client. The structure and quality of the scripts developed will have an impact on the system as whole, especially in regards to further development and maintainability.

I have almost no prior experience in JavaScript besides a couple of minor implementations purely done out of necessity. These implementations were done primarily using the well-proven method of copy-pasting JavaScript made by smart people from the internet immediately followed by praying, where praying may very likely have been the deciding factor in my success.

With that said, I did spend a lot of time researching the JavaScript of the PDF.js development, which I am fairly convinced should be considered an example of very well-designed and structured JavaScript. I have learned a lot along the way, and that might be apparent in the scripts, as this has likely resulted in some overall design-differences or inconsistencies along the way. The design-patterns used may also be different between components, where I believe I have used closures and other common JavaScript patterns in a couple of different ways throughout the system. I am fairly sure this does not pose any concern at all, and that most of this could be refactored fairly quickly, and not require complete redevelopment of any components. Still, it should be noted that some of the scripts may include the potential conundrums for any experienced JavaScript developer. I apologize for this – Rest assured it was not done intentionally.

A primary concern in regards to *my* JavaScript code in general, is the potential for oversights that might result in poor memory management. This is rarely a problem in web pages, given their stateless nature and frequent memory resets, but the viewer page is uncommon in that regard. The viewer in particular will be open for long periods of time, contrary to most web pages, and poor memory management could have serious impact in regards to performance, reliability and utility in general. In conclusion, special care should be taken in the future to ensure proper memory management in the JavaScript code, in particular because my experience in preventing this in JavaScript is lacking.

4.2 The Documents Page

This section will cover the development of the documents page, allowing a user to view a list of documents. This is an essential part of the system, providing the user with many features for managing the list of documents in the system as well as providing entry into the viewer by opening documents.

4.2.1 Purpose

The main purpose of the application is to allow users to upload and annotate documents. As such, a very essential feature of this application is to allow a user to upload documents and view a list of uploaded documents as well, and that is the purpose of this documents page. This list of available documents is one of the first things a user wants to see when he logs in, and its primary responsibility is to allow a user to open documents in the viewer, to begin reading and annotating a document.

In an effort to provide a good user experience in the application, the list of documents in the documents page has been assigned a set of secondary responsibilities as well. It currently allows a user to delete and share documents as well, and a few options for additional features have been discussed as well.

To conclude this section, the purpose of the documents page is to provide a single page for all things related to document management.

4.2.2 Requirements

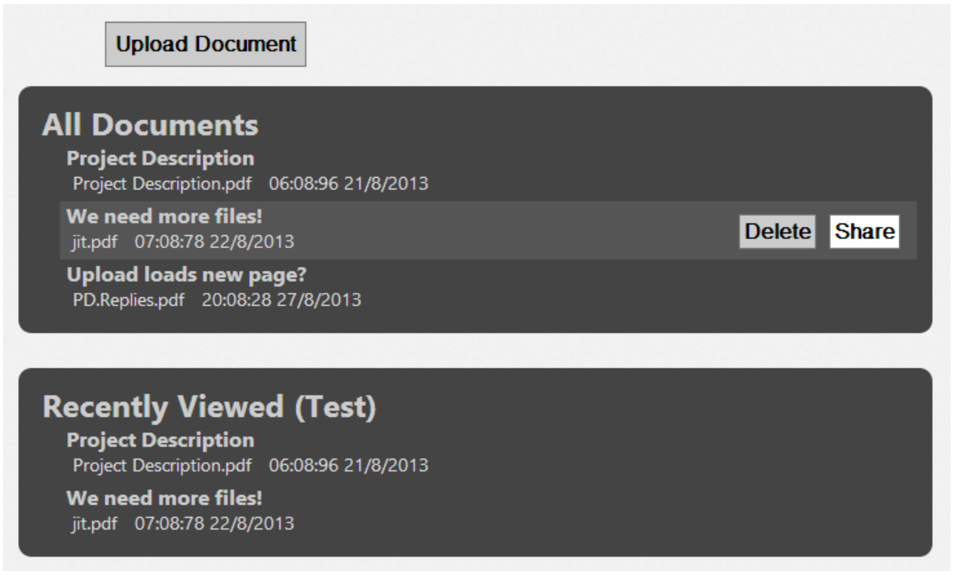
The functional requirements for this page are few, and quite simple:

- Allow a user to upload documents.
- Present a list of documents the authenticated user has access to.
- Allow a user to delete a document.
- Allow a user to share a document.

The non-functional requirements are another matter entirely, as a design with a “good user experience” is requested. Another non-functional requirement is a request to design towards a well-structured and maintainable solution that is not difficult to extend with a few additional features in the future.

4.2.3 Design Considerations

In order to provide at least some support for document organization we have introduced the concept of document lists. This means that the client receives and displays a list of document lists instead of just a list of all documents, which makes it possible to group documents in separate and even overlapping lists. An example of this is displayed in Figure 2 below, where a "Recently Viewed" list is displayed along with the list of all documents.



All Documents

Project Description
Project Description.pdf 06:08:96 21/8/2013

We need more files!
jit.pdf 07:08:78 22/8/2013

Upload loads new page?
PD.Replies.pdf 20:08:28 27/8/2013

Recently Viewed (Test)

Project Description
Project Description.pdf 06:08:96 21/8/2013

We need more files!
jit.pdf 07:08:78 22/8/2013

Figure 2: The documents page, providing tools for managing documents.

This concept of document lists is currently only present in the UI scripts of the client as well as the web API responsible for providing the list of documents details, and the "Recently Viewed" list was added simply by having the web API return an extra list with a few document details inside of it. The concept of document lists does not extend beyond the scripts and the web API yet though, and as such it is not possible to manage these lists e.g. by allowing a user to create a list, add documents to it and then store this information in the database. The choice of how to best design and provide this organization service is still kept up for consideration, and as soon as the back-end data structure is decided upon and provided for the client should be ready for it. The current concept of lists in the client should be easily adapted whether the structure is based on lists or folders;

the implemented concept of document lists could very easily be extended to allow nested lists, and as such support a folder-based structure for document organization.

4.2.4 Implementation

The components used to support the documents page is a single web page, along with three main JavaScript components in three separate files. The JavaScript components will be covered first, and they define a data context component, as well as a model and a view model. The inclusion of a model as well as a view model definition was done because the documents page uses a 3rd party JavaScript library, Knockout.js, to support the Model-View-ViewModel design pattern.

The concept of a view model is to provide an object specifically designed for presenting a domain or model object. This allows for view-specific data objects, which effectively provides a level of abstraction between the presentation logic and the domain model. This abstraction might not be strictly necessary in this JavaScript development, and were I to implement this again I might not be too concerned about developing a structure for the models themselves. With that said, the view models themselves are very helpful in creating a well-structured and intuitive presentation layer. The presentation layer is only further enhanced by Knockout.js, as this has proven a great tool for providing proper integration with the HTML layer of the page.

To begin presenting some code, all the JavaScript components will be first in line. It will take a while before Knockout.js and its integration-neatness is explained, as this will be done along with the HTML page at the very end of this section. To best present the structure of a view model to begin with, Figure 3 below shows the general structure of the view model responsible for presenting the entire documents page.

```
DocumentList.ViewModel = (function DocumentListViewClosure() {
function DocumentListView(ko, dataContext) {
    var self = this;
    this.documentLists = ko.observableArray();
    this.error = ko.observable();
    this.getDocuments = function ()...
    this.showDocumentList = function (documentList)...
    // Share document dialog
    this.documentToShare = ko.observable();
    this.shareDocumentModel = function ()...
    this.shareDocumentDialog = function (document) ...
    this.shareDocument = function (formElement) ...
    // Upload document dialog
    this.documentToUpload = ko.observable();
    this.documentUploadViewModel = function () {
        this.id = ko.observable();
        this.title = ko.observable();
        this.fileName = ko.observable();
        this.file = ko.observable();
        this.uploadDocument = function (formElement)...
        this.close = function ()...
    }
    this.uploadDocumentDialog = function () {
        var viewModel = new this.documentUploadViewModel();
        this.documentToUpload(viewModel);
        $('#uploadDialog').dialog('open');
    }
}
return DocumentListView;
})();
```

Figure 3: The ListViewModel of the Documents Page.

I have omitted most of the code from the view model of Figure 3 to best present the overall structure of the view model, but a few methods remain to show how the view model includes several other view models as well. One of the remaining methods is the documentUploadViewModel method, which provides an object to use when presenting the dialog to upload documents. Besides providing variables necessary for the upload process, it also assists by providing a few methods for submitting the upload request and closing the dialog. An essential part of the list view model is naturally to present the list of documents, and to provide this the view model needs to request some data, so let us take a look at the data context component.

The data context component is responsible for providing the set of necessary data access methods for the documents page, and the general structure of this component is shown below in Figure 4.

```
DocumentList.dataContext = (function () {
    function getDocumentLists(list, errorObservable)..
    function createDocumentItem(data)..
    function createDocumentList(data)..
    function shareDocument(formData, documentToShare)..
    function removeDocument(documentItem)..
    function saveNewDocument(formData, documentItem) {
        clearErrorMessage(formData);
        return ajaxUploadRequest(documentItemUrl(), formData)
            .done(function (result) {
                documentItem.id = result.id;
                alert("Document Uploaded! (ID:" + result.id + ")");
            })
            .fail(function () {
                documentItem.errorMessage("Error adding document.");
            });
    }

    // Private
    function clearErrorMessage(entity)..
    function ajaxRequest(type, url, data, dataType)..
    function ajaxUploadRequest(url, data)..
    // Routes
    function documentListUrl(id) {
        return "/api/DocumentList/" + (id ? "?id="+ id : ""); }
    function documentItemUrl(id) {
        return "/api/Document/" + (id ? "?id=" + id : ""); }
})();
```

Figure 4: The data context components for document data.

The selection of methods for sharing and removing documents are self-explanatory, and the contents of the saveNewDocument has been kept to show how the asynchronous Ajax calls handles call-backs. The fact that the data context class contains the code for what happens on call-backs is not the most optimal solution though. Having the data context classes accept parameters for call-back functions instead would be the more flexible solution, and it would provide a better separation of concern. This would make the data context class responsible for running the correct call-back method, instead of it being responsible for some presentation specific set of events. So, that should be considered for any further development.

Having covered the view model and the data context, Figure 5 below shows the necessary code for initializing the documents page.

```
document.addEventListener('DOMContentLoaded', function (evt) {
    var dataContext = DocumentList.dataContext;
    DocumentList.InitializeModel(ko, dataContext);
    var listView = new DocumentList.ViewModel(ko, dataContext);
    ko.applyBindings(listView);
    listView.getDocuments();
});
```

Figure 5: Initializing the documents page.

The InitializeModel method made during initialization is responsible for injecting the model into the data context. The model consists of just a couple of functions for creating document model objects, with little responsibility except for the variables they contain. As mentioned earlier, having a model might not even be strictly necessary in this design.

The last component to cover is the web page responsible for presenting all we have covered so far, and Figure 6 shows the section of the page responsible for presenting the list of documents.

```
<section id="lists" data-bind="foreach: documentLists,
                               visible: documentLists().length > 0">
    <article class="documentList">
        <header>
            <h2 data-bind="text: title"></h2>
        </header>
        <ul data-bind="foreach: documents">
            <li data-bind="click: openDocument">
                <h3 data-bind="text: title"></h3>
                <button data-bind="click: $root.shareDocumentDialog,
                                   clickBubble:false">Share</button>
                <button data-bind="click: $parent.removeDocument,

clickBubble:false">Delete</button>
                <span class="fileName" data-bind="text: fileName">
                </span>
                <span class="addedDate" data-bind="datetimetext:

addedDate"></span>
                <p class="error" data-bind="visible: errorMessage,
                                            text: errorMessage"></p>
            </li>
        </ul>
        <p class="error" data-bind="visible: errorMessage,
                                    text: errorMessage"></p>
    </article>
</section>
```

Figure 6: The combined HMTL and Knockout.js code, for presenting the list of documents.

As the figure shows, there are a lot of data-bind properties, and these are used by Knockout.js to integrate the HTML page with the underlying JavaScript model. Note that the first line features a data-bind stating “foreach: documentLists”, while line 7 has a similar one for “foreach: documents”. This is how the list of documents list is created, by going through each list of documents and then through each document in that list. The insides of the “foreach: documents” data-bind in line 7 features a few different data-binds itself, which are worth explaining:

- Click: openDocument
- Click: $root.shareDocumentDialog
- Click: $parent.removeDocument

These data-binds will activate a method when the data-bound HTML object is clicked, which is nothing special, but the fact the data-binds support methods for $root and $parent makes it very easy to build well-structured presentation layers. The $root and $parent methods makes it easy to call methods of other JavaScript objects, based on the underlying view model structure defined in JavaScript. The model supporting this underlying structure is shown below in Figure 7, showing the structure of a document item and a document list. I hope it is clear now that the underlying data context of each document in the list is a document item, and that the openDocument method of the document item class is accessed using the “click: openDocument” data-binding. Similar to this, the $parent.removeDocument data-bind refers to the parent item of document item, which would be a document list item, and this list item contains the method to remove an item from the list. Finally, the $root.shareDocumentDialog data-bind refers to the root view model, which is the DocumentListViewModel described earlier.

```
function documentItem(data) {
        self.id = data.id;
        self.title = ko.observable(data.title);
        self.fileName = data.fileName;
        self.addedDate = data.addedDate;
        self.errorMessage = ko.observable();
        self.openDocument = function ()...
    };

    function documentList(data) {
        self.id = data.id;
        self.title = ko.observable(data.title || "<Unnamed List>");
        self.documents = ko.observableArray(
                            importDocuments(data.documents));
        self.errorMessage = ko.observable();

        self.removeDocument = function (document)...
    };
```

Figure 7: The model for documents and document lists.

For a more thorough explanation of this underlying model, this last paragraph explains how the data context of HTML elements changes as a result of data-bind iterations. Using Knockout.js, each HTML element has a data context element it references, and it will attempt to use this data context if any data-bind events are activated, such as the click event. The data context of the main page is the view model covered earlier, the DocumentListViewModel. It was defined as the data context by the method call “ko.applyBindings(listView)” when initializing the page, as shown previously in Figure 5. The same data context will be used by many HTML elements, but each time a data-bind such as “foreach” is called, the data context of the resulting HTML elements will be that of the object iterated. This means that the HTML elements generated by each iteration in the “foreach: document” will be assigned the data context of that single document item. Similarly, the HTML elements generated by each iteration in the “foreach: documentLists” will be assigned the data context of each of the document lists items inside the array “documentLists”.

4.2.5 Conclusion

A web page was developed allowing users to view a list of their currently accessible documents. The page also allows the user to upload documents, and it supports other features related to document management as well. The purpose of the page was to provide a single page for all things related to document management, and it does so successfully. The underlying models and scripts supporting the page are well-structured and should be fairly easy to understand and further develop. Several methods for improving the user experience were demonstrated, and have been used effectively in an effort to enable the page to handle all current management requirements on a single page while still keeping an intuitive design.

If the implementation simply tracked when a user last accessed a document, the current implementation could also support for a list of recently viewed documents, which would likely be an appreciated feature.

4.3 Extending the Viewer: Supporting Tools

This section covers the design and implementation of the toolset developed and explains the purpose of the components involved. The design intent behind the toolset will also be explained along with how it is designed in regards to further development.

4.3.1 Purpose

The purpose of this extension will be to provide a basic framework for a set of tools to create and edit annotations. As stated in the analysis, the development of this toolset is one of the critical areas of this system, as it is critical to determine that it is possible to develop a satisfactory framework for a set of editing tools for this viewer.

The initial development of tools will have little emphasis on providing a wide selection of tools, and only a single tool is built so far. Instead, the primary focus is to develop a proper framework to support the future development of annotation tools, and the framework should be designed to assist the process of developing and integrating the set of tools to come.

The framework will be responsible for providing methods for tools to draw on top of the document, to be able to provide the visual aid necessary to support a good user experience when using the tools. It will also be responsible for managing a set of available tools, as well as options for changing the tool, as is common in most editors.

4.3.2 Design Considerations

An intuitive way to add or edit items in a document is to interact with the document displayed on the screen and this method is commonly used by editors of all sorts, especially when manipulating documents or images. This is also the preferred way of manipulating documents in this system, and to accomplish this the framework should provide a semi-transparent overlay canvas on top of the document when a tool is activated, effectively allowing a user to draw on top of the document.

A suitable element for the overlay canvas would be an actual canvas element, which is an element in HTML5, made for containment and rendering of graphics. This should allow the tools to provide a great deal of visual aid when manipulating annotations, and it easily supports HTML objects as well, effectively allowing tools to draw buttons or other intuitively useful visual aids. Providing this canvas still necessitates a bit of math before annotations can correctly be added to the document though. This is necessary because it is simply most convenient to have the overlay canvas cover the whole "View" area of the viewer, meaning it covers the page, the space in-between pages and the surrounding background as well. As such, the coordinates of objects drawn on the canvas are all relative to the canvas itself and has no relation to the rendered document and its pages. In order to insert annotations and other objects into the document at the correct scale and position it is therefore necessary to convert the size and coordinates of the objects relative to the canvas into a corresponding set of size and coordinates relative to the active document page.

To correctly convert the canvas coordinates to page coordinates in the document it is essential to be able to define where the target page of the document is located relative to the canvas. There are several ways to go about this problem and the currently implemented solution makes the framework responsible for gathering the necessary data whenever one of its tools is activated. The current implementation is neither the most efficient nor intelligent solution, but it is ok for a prototype and it succeeds in gathering the necessary data, as Figure 8 attempts to show; an explanation follows below.

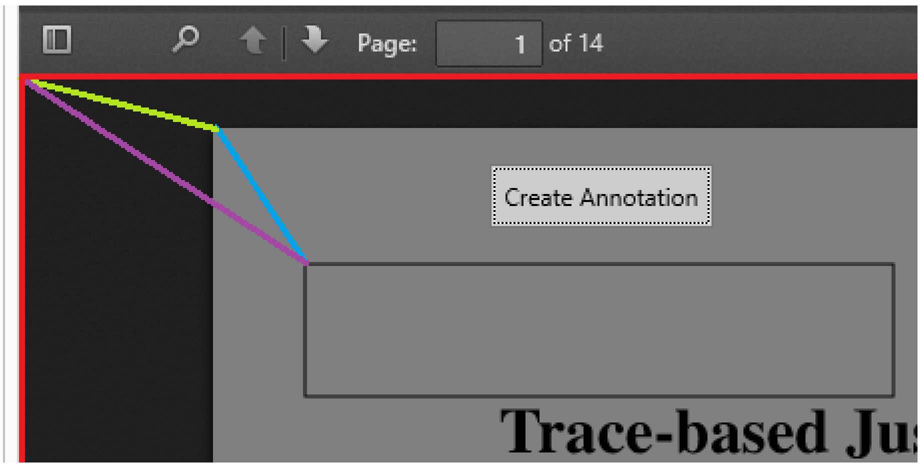


Figure 8: Converting canvas coordinates. Red line defines the area of both the viewer and the canvas overlay (On top of each other).

The red area shows the area that is common to both the viewer and the canvas, and the green line denotes the vector from the page to the viewer while the purple line denotes the vector from the selection rectangle to the canvas. Given these two vectors the blue vector can now be calculated, and as such we are able to place new annotations correctly on the page. In order to handle rectangles and zoom levels we also need to know the size of a page and its current level of scale, and when we begin scrolling down we of course need to take the current scroll position into account as well. To best support the tools in this regard, it would be very beneficial for the framework to provide one or several methods to properly convert coordinates from canvas to document coordinates, and vice versa.

It should also be considered if the coordinates used for the document and its annotations should be defined as percentages of the page height and width, or if some other measurement is more desirable. Using percentage measurements would be a universally applicable unit of measure for any document, but there might be advantages in used other, perhaps more exact, units of measurement.

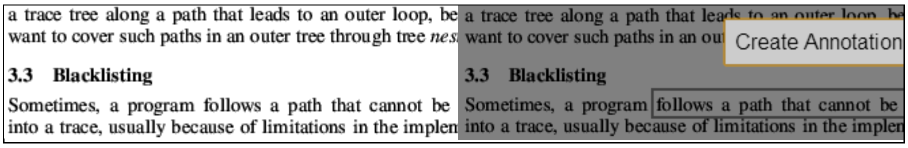
In regards to maintainability, the development of both the framework and any new tools will want to ensure that any direct interaction between the viewer and any tool is carefully considered. Dependencies towards the viewer should preferably be supported by the framework alone, to keep tools more easily maintained. Enforcing the convention that integration of tools happens through the framework alone will ensure that the tools are dependent on the framework, and not the viewer. This will ensure that most adaptability concerns occur between the framework and the viewer exclusively, and this should enable developers to more efficiently be able to debug and detect integration errors. It should also ensure that such integration errors will have to be taken care of only once, in the framework, and not multiple times in multiple tools – some of which the error might not have been reported yet. This will be increasingly beneficial as the number of tools increases.

In regards to adaptability, this extension could work for any viewer that is able to support an HMTL5 overlay with Javascript. In order to correctly fit objects to document pages the extension also needs to have access to the location and size of the document page relative to the canvas, including its currently level of zoom and scroll position. Both of the aforementioned requirements can be met with any of the PDF rendering methods covered in the analysis, and they are easily supported in the two most promising methods: Rendering through either HTML5 or images. As such, it would be beneficial to implement the framework with an emphasis on keeping the essential points of integration with the viewer to a minimum.

4.3.3 Implementation

The current implementation consists of two main classes called AnnotationTools and SelectionTool, which is referred to as the framework and the selection tool, respectively, throughout this section. The former is responsible for providing a framework for the set of tools to support, while the latter is a tool itself: a tool for selecting a point or an area on the document.

Before covering the implementation, let us present a quick demonstration of what it looks like when the components are actually in use. Figure 9 below shows how an active selection tool has activated the overlay canvas provided by the framework, which the tool then uses to draw a selection-rectangle around a section of text.


a trace tree along a path that leads to an outer loop, be
want to cover such paths in an outer tree through tree *nes*

3.3 Blacklisting

Sometimes, a program follows a path that cannot be
into a trace, usually because of limitations in the implen

a trace tree along a path that leads to an outer loop, be
want to cover such paths in an ou

3.3 Blacklisting

Sometimes, a program follows a path that cannot be
into a trace, usually because of limitations in the implen

Figure 9: Left image shows the default view. Right image shows the view with the tools canvas overlay, with an active rectangle selection and a context menu above it.

Figure 9 also reveals that the selection tool uses the canvas to draw a context-menu above the selection-rectangle as well, featuring a single button for creating an annotation. Being able to create context menus on the canvas along with the methods for drawing basic shapes should allow the tools to be a lot more flexible. For instance, the selection-tool in use here could be used to create new annotation rectangles sized according to the selection-box, simply by adding that options to a context menu. Having explained and exemplified the basics of the toolset, let us take a closer look at the implementation.

The design intent of the framework is provide a set of basic methods that most tools will need, such as the methods for creating and removing the overlay canvas used for drawing on top of documents. When the overlay canvas is activated by a tool, the framework is responsible for propagating mouse events from the overlay canvas through to the currently active tool. Having provided a canvas along with continuous mouse events from it, the last missing piece is now a way to draw on the overlay canvas as well. The framework naturally takes care of this as well, by exposing the drawing context of the canvas which gives the active tool access to methods for drawing a number of different geometrical shapes. The framework class contains a few other features as well, such as managing the set of available tools, but the core set of methods it exposes are the ones that allow tools to easily activate the overlay canvas and draw on it.

Figure 10 below shows the main methods of the main framework class along with the event listener function for initializing the AnnotationTools class and adding an instance of the SelectionTool class to its set of available tools.

```
document.addEventListener('DOMContentLoaded', function (evt) {
        var annTools = new AnnotationTools();
        var selectionTool = new SelectionTool(annTools);
        annTools.addTool(selectionTool, "selectionTool");
        annTools.init();
    }, false);

var AnnotationTools = (function AnnotationToolsClosure() {
    function AnnotationTools() {
        // Some variables omitted
        this.getCanvas = function ()...
        this.getViewerContainer = function ()...
        this.get2dContext = function ()...
        this.getViewPort = function ()...
        this.init = function ()...
        this.addTool = function (func, funcName)...
        this.createToolsCanvas = function ()...
        this.removeToolsCanvas = function ()...
        function mouseEventHandler(event)...
    }
    return AnnotationTools;
})();
```

Figure 10: The AnnotationTools class, providing a framework for other editor tools.

The .addEventListener() method in the top of Figure 10 shows how the selection tool is added to the AnnotationTools class, and it is the intent that any additional tools developed should be added and ready for use as simple as that. This covers the basics of the framework, and having developed this set of basics methods for tools along with a virtual toolbox should make it easier to develop the necessary tools for this application. Having covered the basics of the framework, let us move on to the selection tool and see how a tool is integrated with the supporting methods of the framework.

The selection tool is the first implementation of a tool designed to interact with the set of supportive methods provided by the framework. To interact with the framework there are a few essential methods that a tool must implement, as they will be expected and invoked by the framework. These methods are the tryActivate method along with methods for receiving three types of mouse events: mousedown, mousemove and mouseup. Figure 11 below shows the methods of the selection tool class, and based on the method names it should be fairly easy to see that the purpose of this tool is to draw a context menu along with either a selection rectangle or a selection point.

```
var SelectionTool = (function SelectionToolClosure() {
    function SelectionTool(annTools) {
        // Some variables omitted
        this.getCanvas =  annTools.getCanvas;
        this.get2dContext = annTools.get2dContext;

        this.tryActivate = function (event)...
        this.deactivate = function ()...
        this.mousedown = function (event)...
        this.mousemove = function (event)...
        this.mouseup = function (event)...
        this.drawContextMenu = function (event)...
        this.clearContextMenu = function ()...
        this.drawSelectionPoint = function (event)...
        this.drawSelectionRect = function (event)...
        this.clearSelection = function ()...
        this.reset = function ()...
        this.onCreate = function ()...
    }
    return SelectionTool;
})();
```

Figure 11: The SelectionTool class, enabling a user to select a point or an area on a document.

As mentioned, one of the benefits of this integration is that the tool has no dependencies or even any knowledge of anything except the framework, so perhaps this integration should be explained further.

When a tool is created the constructor includes a reference to the framework class, giving the tool all the necessary references it needs to access the canvas and all the other supportive methods of the framework. When a tool is the currently active tool of the framework, the framework may attempt to activate it using the tryActivate method. If the tool allows itself to be activated, it will announce so and will begin receiving mouse event through its three mouse event methods. This is all the active input a tool will receive, and all of the tool methods for drawing selections rectangles and context menu are activated based on the incoming mouse events. On a final note, the onCreate() method of the tool is a method accessed by the context menu drawn by the tool itself. When a selection is drawn by the user, the context menu is displayed as well, and this method could then be called to create a new annotation. The current selection-rectangle and the conversion methods of the framework could then be used to calculate the document-relative coordinates of the selection-rectangle and create an annotation at that exact position on the document.

4.3.4 Conclusion

A framework was designed and developed to support the future development of a set of tools for annotating documents. The framework provides a set of methods deemed necessary for most tools as well as integration with the overlay canvas used to provide visual aid for the tools to most effectively accomplish their intended purpose. A selection tool was developed and integrated successfully with the framework class, and both the development and integration of the selection tool was intuitive and should be easy to understand in future developments.

Additionally, the framework successfully decouples the dependencies of tools from the rest of the application. This should ensure that maintenance of the tools framework and its tools is kept at a minimum, and that any changes or improvements to the viewer should affect the framework exclusively and not the tools themselves. As such, the framework might need occasional maintenance, but proper maintenance of the framework alone should ensure that the tools work as well.

On a final note, the choice of what unit of measurement to use for the position of annotations in the document has neither been further researched yet, nor finally decided.

4.4 Extending the Viewer: Rendering Annotations

This section covers the extension developed for the rendering annotations, and will discuss the components involved and why they were developed in the first place. Finally a section will be dedicated to discussing how these extensions affect the adaptability and the future development of this solution.

4.4.1 Purpose

The purpose of this extension is to separately download and render annotations, instead of having them merged into the document and then rendered by PDF.js along with the document itself. There are several immediate benefits to this design, both in terms of adaptability and maintainability. The first big benefit in terms of adaptability is that this will make it possible to exchange PDF.js as our method of rendering. Since we did go for a read-only document on the client side, we could choose to convert PDF documents into HTML5 server-side and then serve HTML directly instead of rendering it client-side through PDF.js. Given this extension, this option is now a very viable choice and it would not require that much to integrate it.

Besides the benefit of adaptability, this extension will also make it possible to provide a custom implementation of annotations separately from the existing implementation PDF.js. This is quite important as PDF.js is made primarily as a viewer, and as such it does not provide any functionality towards creating or editing annotations, or manipulating any other part of the document for that matter. This is of course makes it well suited for our read-only requirements in regards to documents, but when it comes to annotations the solution would benefit a lot from a set of well-designed tools for creating and editing annotations, and the viewer just does not provide that. A solution for separately downloading and rendering annotations will allow for a viewer-independent development of annotation rendering and editing. This should lower the maintenance costs of the combined viewer and annotation solution as a whole, as it will be easier to maintain the annotation solution while keeping the 3rd party viewer up to date if they are well separated.

4.4.2 Implementation

The two main components in this extension are called the ListView and the AnnotationsCache. The ListView is responsible for displaying a list of annotations in the right side-bar, while the annotations cache provides methods for asynchronously retrieving annotations, either by downloading them or retrieving them from its cache.

The annotations cache is a wrapper for the data context class, responsible for handling the download of annotations more efficiently than the basic data context class. It currently uses a concept of promises which is a rather simple class responsible for asynchronous retrieval of data. When retrieving annotation data from the cache the object returned is actually a promise, and the retrieval of annotations will be followed by an asynchronous .then() method to be activated when the promise is resolved. This is quite similar to the ajax post and get methods used throughout the application, where asynchronous methods for ‘done’ or ‘fail’ are included as callback methods to the asynchronous request. After having requested a promise, the promise will be resolved as soon as the data cache has the data available. While a cache might not be necessary for providing the simple annotations we have at this point in time, it will likely be necessary for large documents with many annotations. However, if the data requirements for annotations should increase at some point in the future, e.g. if the solution is to support image annotations using any user-provided image, having a layer responsible for data management, such as this cache, could even be essential.

The ListView is responsible for displaying a list of annotations separately from the document viewer, and this list is currently found in a side-bar to the right of the viewer. At first glance this could be considered a feature introduced simply for convenience, but the fact that we want to support replies and/or discussions for each for annotation almost necessitates a list of annotations like this.

When it comes to options for displaying annotations there are many to choose from, and this solution attempts to display annotations passively, without user interaction, while at the same time not obstructing the document itself. The annotations themselves are displayed in the document without any text content displayed, and then the text content of all annotations is displayed in the right side-bar, when it is active.

The current implementation does actively display annotation text as well though, as it displays the text of an annotation if the user points at an annotation with the cursor. If we were to display whole discussions directly on the viewer in the same way it would need to be done with some kind of annotation-expand button, so a user could expand an annotation to view the attached replies to it. Enabling a user to expand an annotation still requires a user to interact with the document in order to read the discussion, so while expanding might be a nice feature to add at some point, some sort of passive display of annotations, such as the list view extension, is to be preferred.

4.4.3 Conclusion

As discussed in the section covering the purpose of this extension, separating the rendering of annotations from the PDF viewer itself has benefits in regards to adaptability, in that we can more easily exchange the current PDF viewer for just about any of the alternatives discussed in the analysis. Rendering annotations will naturally be a very essential aspect of this solution, and the current implementation of this extension should make for a good starting foundation. The current implementation is well separated from the PDF viewer with only a few lines of code needed for integration, and the asynchronous data access methods of the cache developed should provide a smooth experience, even as the amount of data required for rendering annotations increases.

4.5 Conclusion

A single page for all things related to document management was developed. The beauty of the page might be lacking, but the page implements and exemplifies the technological necessities of a smooth user interface. Scripts are well-structured, maintainable and adaptable, and should provide a good foundation for development of additional features.

A documents viewer was implemented, and the viewer is primarily based on the 3rd party viewer PDF.js. PDF.js renders a PDF using HTML5 exclusively – No native code involved, less security concerns. The viewer could easily be exchanged with one-time html conversion instead, with little effort needed to re-integration tools and custom rendering of annotations. This exchange has always been an interesting option for the viewer, and this has been considered throughout development in an effort to keep the view adaptable enough to support such an exchange.

The viewer has been extended to provide a basic framework to support tools and another extension was developed to support rendering of annotations separately from the rendering of the document. Despite little-to-no experience in JavaScript, the scripts are considered fairly well-structured and maintainable. Both extensions were developed with great emphasis on the future maintainability and adaptability of components, and both should provide good foundations for future development.

Ensuring a good user experience throughout the system will require proper attention to the visual design of the page, but the technological necessities for providing a responsive user interface have been successfully developed and exemplified.

5 Design: The Server

This section will cover the design and implementation of the server side of the solution. The components developed will be covered and their purpose in the solution will be explained as well as their design intent and implementation. The design of the server will feature thorough discussions of the most influential design considerations made, such as the design of the domain layer and the data access layer.

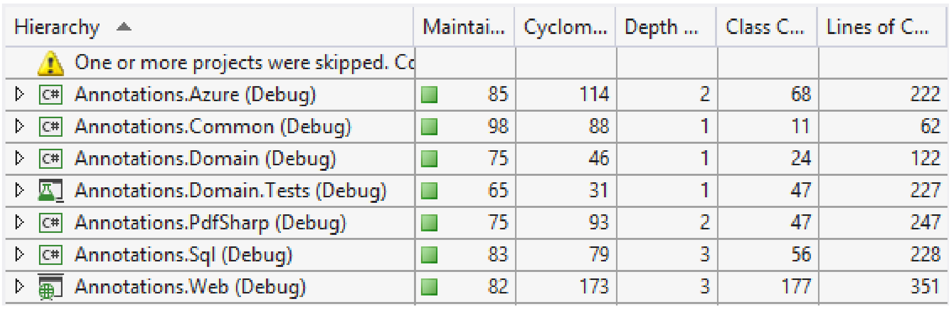
The following list presents a brief overview of the topics to be discussed in this section, along with a few details regarding the contents of each topic. Following the list, the current code metrics will be presented to give an indication of how the current design fares so far.

- **Main Web Server**
  - Serves web pages.
  - Handles routing for pages and web API.
  - Provides a web service API for a set of data access endpoints.
- **Authentication and Authorization**
  - Provides simple forms authentication.
  - Provides OAuth using Facebook ID.
  - Custom SQL service handles document permissions.
- **Solution Structure**
  - Common library manages dependencies.
  - Provides common data objects and interfaces.
- **Data Access Layer**
  - Separates data access dependencies from the domain logic.
- **Domain Logic Layer**
  - Handles domain logic and authorization.
  - Uses repositories for data access.
  - Only dependent on interfaces defined in the common library.
  - As a result, service classes are well designed for unit tests.
- **Data Storage**
  - Blob storage for files.
  - Table storage for annotations.
  - SQL storage for relational data.

- **PDF Support**
    - Custom PDF Editor provides a few necessary features.
    - Extracts and removes annotations from documents.
    - Merges annotations into documents.

One of the primary concerns in the development of the server architecture and design has been the development of a highly maintainable system, and this will be a main topic in the design discussions throughout this chapter. Developing towards a high level of maintainability is perhaps the most important topic of this chapter, and I have chosen to begin the chapter by presenting a measure of success in this regard. The code metrics of the final solution is shown below in Figure 12, and the maintainability index is quite satisfying across the board. The maintainability index is officially assessed as such:

"A green rating is between 20 and 100 and indicates that the code has good maintainability. A yellow rating is between 10 and 19 and indicates that the code is moderately maintainable. A red rating is a rating between 0 and 9 and indicates low maintainability."[7]



Hierarchy	Maintai...	Cyclom...	Depth ...	Class C...	Lines of C...
⚠ One or more projects were skipped. Cc					
Annotations.Azure (Debug)	🟩 85	114	2	68	222
Annotations.Common (Debug)	🟩 98	88	1	11	62
Annotations.Domain (Debug)	🟩 75	46	1	24	122
Annotations.Domain.Tests (Debug)	🟩 65	31	1	47	227
Annotations.PdfSharp (Debug)	🟩 75	93	2	47	247
Annotations.Sql (Debug)	🟩 83	79	3	56	228
Annotations.Web (Debug)	🟩 82	173	3	177	351

Figure 12: Code metrics of the final solution. (Maintainability, Cyclomatic Complexity, Depth of Inheritance, Class Coupling, Lines of Code)

[7] Code Metrics Values: http://msdn.microsoft.com/en-us/library/bb385914.aspx

5.1 The MVC4 Framework

Using a web application framework is a great way to get a lot of features for new your web application using very little effort. As such it is well worth considering using one and in our case there are few reasons not to.

The framework of choice will be the ASP .NET MVC4 framework (MVC4). The MVC4 framework has gotten a lot of popularity in the last few years, primarily in the crowd of .NET developers, as it enables a .NET developer with little-to-no experience in html and JavaScript to get started developing a website rather effortlessly, at least compared to other options.

Most of the required features related to authentication are provided by the MVC4 framework right out of the box. This covers user registration, login and a few administrative features such as changing the password of your account. Some of these features are not that well implemented though, and some adjustments will likely be required to improve the maintainability and testability of the solution in general.

A noteworthy benefit of MVC4 is that it is well structured and quite adaptable, with nice separation of concern in part due to its implementation of the model-view-controller pattern. This fits well with our preference for adaptability in this early stage of development, and it would not be too difficult to develop the rest of this solution to not be too dependent on this particular web application framework.

It also support the dependency inversion principle used throughout the design, by providing a framework for dependency injection. Depedency injection is supported through Unity, and a section of the code for registering types for dependency injection is shown below in Figure 13.

```
// Repositories
container.RegisterType<IAnnotationRepository, AnnotationRepository>();
container.RegisterType<IDocumentDetailsRepository,
DocumentDetailsRepository>();
container.RegisterType<IDocumentRepository, DocumentRepository>();

container.RegisterType<DocumentRepository>(
    new InjectionConstructor(typeof(CloudBlobClient),
        AzureConstants.Blobs.Documents));
```

```
container.RegisterType<AzureTable<AnnotationTableEntity>>(
    new InjectionConstructor(typeof(CloudStorageAccount),
        AzureConstants.Tables.Annotations));
// Services
container.RegisterType<IAuthenticationService, AuthenticationService>();
container.RegisterType<IDocumentService, DocumentService>();
container.RegisterType<IAnnotationService, AnnotationService>();
```

Figure 13: Dependency injection bootstrapper.

The MVC framework also provides support for easy bundling of scripts and style sheets, as shown below in Figure 14. Bundling scripts and style sheets can be used to effortlessly combine and minimize both scripts and style sheets. This has great effect on the speed of serving web pages, and can easily reduce the amount of requests in a typical get request to only a fraction.

```
public static void RegisterBundles(BundleCollection bundles)
{
    bundles.Add(new ScriptBundle("~/bundles/documents").Include(
        "~/Scripts/App/documents*"));
    bundles.Add(new ScriptBundle("~/bundles/annotations").Include(
        "~/Scripts/App/annotations*"));
    bundles.Add(new ScriptBundle("~/bundles/pdfjs").Include(
        "~/Scripts/Pdfjs/compatibility.js",
        "~/Scripts/Pdfjs/debugger.js",
        "~/Scripts/Pdfjs/l10n.js",
        "~/Scripts/Pdfjs/pdf.js",
        "~/Scripts/Pdfjs/viewer.js",
        "~/Scripts/App/pdfjs.extensions.js"));
}
```

Figure 14: Registering script and style sheet bundles.

And that concludes the most noteworthy benefits of the MVC framework.

5.2 Solution Structure

In the interest of keeping the server-side solution as adaptable as possible in these early stages of development, the solution structure is designed with a heavy emphasis on dependency management. In these early stages of development it makes a lot of sense though, as it gives us a bit more freedom to play around, change components and test alternative technology choices without having to handle a series of ripple effects throughout the solution. It is important to keep in mind that the solution structure is a result of an iterative design process, and the structure presented here is a result of all the design considerations to be discussed in this chapter.

Figure 15 below shows the solution structures, as defined in a layer diagram in Visual Studio (“VS”). The top-right numbers in each layer shows the number of projects in that layer, and VS provides methods to validate this layer model against the dependencies/references used in the included projects.

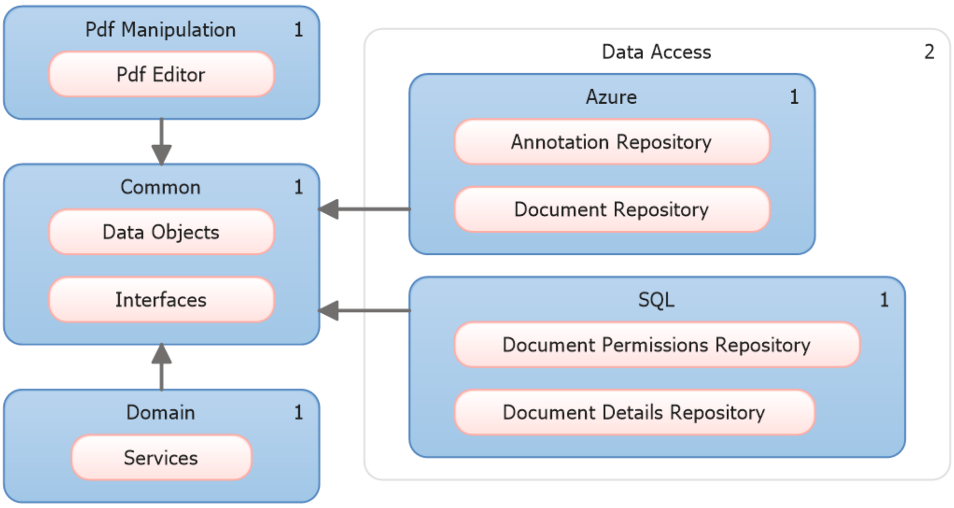


Figure 15: Solution structure emphasizing dependency management.

The blue layers are layers with only a single project, meaning we could just as well think of blue layers as actual projects. The white “Data Access” layer denotes all the data access projects of which there are currently two. The red “layers” have no function in this layer diagram in regards to validation and they are not really layers at all; they denote either a class or a set of classes and they simply serve to improve the immediate understanding of the diagram by specifying the most important high-level elements of each layer.

The intent of the design is fairly straightforward in that we provide a set of data objects and interface definitions in the common library to be used when implementing components with 3rd party dependencies. This ensures that we can fairly effortlessly prototype multiple different 3rd party technologies by implementing the technology in components based on the interfaces defined for them in the common library and simply choose which component-implementation we would like to use at any given time.

This layer of separation means that we can exchange components fairly effortlessly, even at runtime if we wish, and still be fairly confident that this will not break any other components or tests dependent on those components.

5.3 Data Access Layer

This section covers the design of the data access layer and discusses the choices made during development as well as the thoughts behind them. A primary concern in the system is to keep a strict separation between data access and domain logic, as this has several benefits to the system design in general. This has led to the development of series of conventions, restrictions and responsibilities split among the main classes of these two application layers, and this section will cover the design intent behind the data access layer and its primary set of classes: the repositories.

5.3.1 Responsibility and Purpose

The primary responsibility of data access components developed in this layer is to support the domain layer of the application, and the components will have to be designed with this in mind. The data access components must therefore be able to provide data access to the domain services while ensuring that the domain services are still kept de-coupled from any outside influences.

The actual purpose of the data access layer is much more than simply providing data access though. The design of the data access layer can have a great influence on the quality of code in the system as a whole, especially in regards to the overall maintainability and adaptability of the solution. In general the data access layer is a volatile layer that is easily affected by outside influences, and it is important to consider the possible implications of improper design at this level.

To conclude, the responsibility of the data access classes is to provide direct database access. The purpose of the data access layer itself is to provide data access throughout the application, in a way that encourages code re-use and ensures a high level of maintainability in the application. A high level of adaptability for data access components is preferred, as this will simplify the process of making changes and optimizations to the data access components.

5.3.2 Using Object-Relational Mappers

The processes of accessing a database can be simplified immensely by using a data access framework or an object-relational mapper such as the Entity Framework, which provides a set of classes and helpful methods to reduce the amount of data-access code a developer would otherwise needs to write. The following statement answers the question "What is Entity Framework?" as stated on an official Entity Framework site:

"Entity Framework (EF) is an object-relational mapper that enables .NET developers to work with relational data using domain-specific objects. It eliminates the need for most of the data-access code that developers usually need to write."[8]

Not having to write most of the usual data access code greatly speeds up the development of data access components, but in some cases it also cripples both the maintainability and adaptability of a system immensely. This is especially true for frameworks attempting to support the entire data access process from the structure of data in the database, to the data access and domain layers, and all the way to the process of displaying the data on the screen. Entity Framework is one such framework; it is able to easily support this whole process, and it is able to do so with barely any help from the developer.

This solution uses the Entity Framework, but in order to not cripple the solution as mentioned, the data access components of Entity Framework are responsible for, and restricted to, direct database access only. The components responsible for direct database access through Entity Framework are referred to as repositories. Repositories are entirely de-coupled from the rest of the application, accessed only through interfaces defined in the common library of the solution. This ensures that any framework dependencies along with any of the internal structures used for data access are kept separated from domain logic.

[8] http://msdn.microsoft.com/en-us/data/ef.aspx

5.3.3 Design Considerations

The primary responsibility of data access layer is to provide data access, and that is perhaps the only functional requirement of the data access layer. There are a lot of other aspects to consider when designing components for the data access layer though, as data access is a very essential part of most software systems. The design of this layer will heavily influence the system in regards to most aspects of software quality, and the data access components should be designed with this in mind.

Ensuring a high level of adaptability is especially critical in the early stages of development, as this allows for cost-efficient prototyping and testing of different storage methods. To ensure adaptability, the model developed for assessing the quality of software in this system identifies reusability and interoperability as critical factors. To best ensure that the data access components are reusable, each component or class should be responsible for handling just a single, logical subset of methods for handling data in the application. In this case it might be beneficial to restrict the repository to a single entity of data, meaning that one such logical subset of methods could be a few methods for the storage and retrieval of documents, and documents alone; this component should not include methods related to document details as well, or its annotations, or its permissions. Keeping the data access components simple and focused on a small set of methods for a single type of data should ensure that the components are relatively well-designed in terms of re-usability.

To ensure a decent level of interoperability it is absolutely essential that the components provide a proper abstraction layer which other components can depend upon and use for integration. This will allow other components to utilize data access components by depending on the abstraction of these data access components, instead of the actual implementation of the components. This makes it possible to seamlessly exchange the implementation of a data access component with another one, as long as they provide methods for the same abstraction. This level of abstraction is particularly important because it is the intended purpose of the data access components to provide data access for the domain layer. Some components could perhaps use data access components without proper abstraction layers, but the components in the static and fixed nature of a domain layer should not submit to such outside influences.

Besides the already discussed benefits, this abstraction layer also has the benefit of de-coupling the dependencies of the data access components from the rest of the application. This does wonders for both the maintainability and testability of the domain logic responsible for data access, as any data access technology changes will have no impact on the domain logic and all the tests related to it. This is a great level of insurance for a developer as well, as this decoupling makes it a simple task to change and optimize the data access methods of repositories or even the database itself, without having to worry about ripple effects in any layer other than that of the repositories. This would not have been possible if an object relational mapper had been used as the source of both the database and domain objects, and this is exactly why we do not want to the ability of Entity Framework to *"work with relational data using domain-specific objects"*. We do appreciate the part about eliminating most of the necessary data access code though.

5.3.4 Implementation

Repositories are the classes closest to the actual database, at least if we just consider our own classes and not that of the data access technology of choice.

As discussed in the design section of this component, a repository is responsible for handling the data access of a small logical subset of data in the application, such as the storage and retrieval of documents. To exemplify this, Figure 16 shows an interface definition for the document repository:

```
public interface IDocumentRepository
{
    Stream GetDocument(Guid documentId);
    void AddDocument(Guid documentId, Stream document);
    void DeleteDocument(Guid documentId);
}
```

Figure 16: The interface definition for the document repository.

As the interface definition shows, the methods exposed are kept quite simple and dependencies are avoided as much as possible. To illustrate how dependencies are avoided the use of Stream classes for input and output of documents makes the document repository quite flexible in regards to the type of document the system is able to support. Earlier prototype development might get away with exposing dependencies such as using a PdfSharp Document class for input and output instead of the Stream class, but as the solution matures dependencies should be handled more carefully and refactored if necessary.

The current implementation of the document repository uses an “Azure Cloud Blob” to store documents in the cloud, but since this interface wraps and abstracts the document repository class, the dependencies of the implemented class is confined within the class itself. This means we can easily add to or change our storage providers by implementing another document repository class, e.g. for local file system storage, and as such this makes our repository-dependent classes more adaptable.

This might seem like an odd time to conclude the implementation of the data access layer, but I will do so for the sake of adding dramatic effect. The important and system-critical aspects of the data access layer have been explained, and an essential part of this design is the fact that the system knows nothing of the data access layer except for a few repository interfaces. If these interfaces have been properly designed as to not require changes in the future, the data access layer has served its purpose well. The implementation of any repository classes will not be covered in this section, but some data storage methods are covered in chapter 5.5: Data Storage Options.

5.3.5 Conclusion

The data access layer was discussed in terms of requirements and design, along with the influence this has on the system. The most important aspects of the design is its emphasis on ensuring that data access components retain a high level of adaptability and re-usability while also ensuring that domain services are highly decoupled from outside influences, to keep them as static and fixed as possible.

Keeping the responsibility of each repository restricted to small subsets of data while keeping them all simple and focused solely on data access has made them easy to re-use for multiple purposes. Several of the simple repository methods are currently used by multiple different service classes.

Creating repository classes to handle direct database access keeps a nice separation of concern between the domain layer and the data access technology of choice. This makes it possible to optimize or completely change sections of database access code seamlessly, without compromising the domain logic or the tests validating it in any way. This has already been beneficial to the development of this project, as I have both exchanged and/or optimized different sections of the data access code several times now, with very little to no effort made in regards to integration of the changes made.

The design makes good use of most of the SOLID principles and the design benefits greatly from it, which should indicate at least some level maintainability and adaptability. The single responsibility principle is a key player in keeping the responsibilities of components well separated, and the interface segregation principle is used to great extend as well, as we choose to implement only a small logical subset of data access features for each repository. Of course, as we depend upon interfaces for dependencies throughout this design, the dependency inversion principle is applied as well.

5.4 Domain Logic Layer

This section covers the domain logic of the system, and explains the design and conventions applied to this layer of the application. The design of the domain logic layer has great influence on the quality of code throughout the application, and it benefits the system immensely in terms of adaptability and maintainability.

This section will first state the functional responsibilities of the domain layer components as well as the purpose of the domain layer itself. Following this will be a lengthy discussion of the design considerations, where I will explain the division of application critical responsibilities among the surrounding layers as well as the conventions made in this regard. The design section will help clarify the design decisions made in this process, especially in regard to the quality of code in the application. It will be a focus of the design section to pinpoint what was done to achieve certain quality-of-code benefits and how the design helps encourage or enforce these benefits, as well as how this will affect the future development of the system.

5.4.1 Responsibility and Purpose

The domain logic layer is responsible for enforcing the rules of the domain throughout the system, and it does so with a great deal of support from the data access layer. The data access layer of this system includes a number of simple repository classes responsible for data access, as explained in section 5.3: Data Access Layer.

The domain services are designed to be responsible for data access as well, but whereas the repositories are design for simple database access, the domain services are designed to handle data access according to domain logic. Initially domain services might sound like better versions of repositories, but they each serve their purpose and domain service classes are very much dependent on repositories. Domain services are not allowed to access data themselves, and are therefore required to access data through repositories, and through repositories only. As such, the responsibility of a service class is to enforce domain logic using repositories for data access. This effectively adds another application layer to the data access layer: a layer where data access is handled according to domain to logic.

The purpose of the domain layer is to keep domain logic easily testable and maintainable. The domain layer should be one of the most static and fixed layers in the application, susceptible to outside influences when the rules of the domain or business changes. One of the primary concerns of this layer should be to ensure a high level of decoupling between the domain logic and any potential outside influences, to ensure a static and fixed domain.

5.4.2 Design Considerations

This section covers a few different topics considered during the design of the domain layer and its service classes. Most topics are related to domain driven design and will cover a discussion of its use in the system along with a few areas where this kind of design is not a great fit for the system. Lastly this section will discuss a part of the design where the use of domain language and its activities could benefit the understandability of the system in regards to authorization.

5.4.2.1 Domain Models and Services

A common approach when building a domain logic layer is to build a series of domain models and services and have them interact and change according to domain logic. The current domain layer contains very little need for object interaction or state and as such an actual domain model has little use, at least in the current state of the system; there is simply not anything to model, as very little needs to be changed or verified between the server and the presentation layer. The development of more feature-rich annotations might change this, e.g. to more effectively support annotations with more advanced behavior. An example that might warrant a domain model could be the support for annotations designed for marking problems, where annotations would support the ability to request answers and solutions to problems as well as methods for marking a satisfying answer. This structure could easily be complex enough to warrant a proper model, especially in an open environment where efforts, such as answering and solving problems, are often rewarded with some amount of virtual credit or symbol of status.

Despite having little use of a domain model in the current requirements, there are still many operational aspects in the domain logic, e.g. rules for what happen when users upload, share or delete documents. Since the current domain logic is mostly operational, it will primarily be handled by domain services.

5.4.2.2 Enforcing the Use of Domain Language

The domain services are naturally closely tied to the domain, and most of their methods will be named according to what their activity is referred to in the domain language. What a data access class might refer to as “Adding permissions for a user to read and annotate a document”, a service class might instead refer to as “Sharing a document”. This comes natural as the service class implements domain logic while a data access class implements more in terms of the database, and this should make the solution easier to understand – At least if you are aware of it. If “Sharing a document" is what the system does according to the domain logic, then this should be reflected in the implementation of the domain logic as well.

Just the convention of having these domain specific services implement methods directly related to domain activities has several immediate benefits. To begin with, the service methods are implemented based entirely on the rules of each separate domain activity. This should make the implementation of them easy to understand in the context of the domain rules. It also gives a very clear definition for where this logic should be enforced, and clearly defined responsibilities are usually a plus for the design of system. Maintainability should be improved by the fact that errors related to the domain logic should be very easy to locate, as all the rules for any one domain activity are clearly defined in one place.

5.4.2.3 Domain Language and Activity-based Authorization

The current implementation of service classes makes them responsible for both authentication and authorization, according to domain logic. In the case of authentication, the responsibility merely defines that services are responsible for retrieving the user id of the current user. This should ensure that all service calls are correctly authenticated and that the authentication process is not circumvented by poorly designed parameters allowing a user to pose as another user using simple parameter manipulation. The design of having services provide authentication and authorization while repositories are simple data access classes is a convention used throughout the application. This will hopefully make it easy for a developer to know when to use the different classes for data access, and how to separate domain logic from database-centric data access.

Authorization is still a design in progress though, for the sake of adaptability, and while the current solutions supports the necessary authorization features, the services might not be the ones responsible for enforcing this logic in the future. The preferred degree of authorization in this solution is authorization based on activities, e.g. authorizing actions based on whether or not a user is allow to access "DeleteDocument(id)". Because of this, the domain services will likely be involved in this process, as the access rules are often closely tied to the domain specific methods.

The current implementation authorizes a user in a single line of code in the beginning of each restricted domain service method. The fact that it is so simple makes it easy to change at a later date if an actual authorization layer is desired. The implementation can easily be modified to be applied as a method attribute instead. Whether or not method attributes are desired for authorization enforcement should be decided based on the effect this would have on testability of authorization in general.

5.4.2.4 Conclusion

Keeping the service classes solely responsible for handling domain logic ensures that domain logic is well separated from any outside influences, and this is further ensured if the domain logic does not have any 3rd party dependencies. The domain layer should not have to be changed for any reason except when the rules of the domain need to be adjusted - Even domain/business logic evolves sometimes.

A static, fixed domain layer makes the domain logic very affordable to test, as the tests should be equally static. This should encourage thorough testing of the domain-layer and improve the maintainability of the solution in general. The design of the service layer and repository layer makes the implementation of domain logic in service classes easy to read and understand, at least in the currently implemented tests, which is yet another benefit to both the testability and maintainability of the domain layer and the solution as a whole.

This design should lead to a set of highly de-coupled domain services, exclusively responsible for enforcing domain logic. Ensuring that the domain classes are exclusively dependent on well-designed abstractions should be strictly enforced, especially in regards to the repositories. Whether or not authorization should be a part of the domain is often considered a grey area, with reasonable arguments made for both cases, and the domain services should be kept adaptable to decisions in this regard, as the method for authorization has not been finally decided.

5.4.3 Implementation

The relationship between the repositories as simple data access tools and services as tools to enforce domain logic is the most important aspect of the domain layer. To best explain this relationship the next couple of figures will exemplify the relationship between two such classes.

The repository for document permissions shown in Figure 17 provides no authentication and it is easy to deduct that this method is able to check the permissions of any user, not just the user currently logged on, as the HasPermissions methods includes a userId parameter.

```
public interface IDocumentPermissionsRepository
{
    bool HasPermissions(int userId, Guid documentId,
                        DocumentPermissions permissions);
    void AddPermissions(int targetUserId, int grantedByUserId,
                        Guid documentId, DocumentPermissions
permissions);
    // Omitted some interface methods for clarity
}
```

Figure 17: Simple data access methods of a repository.

The same goes for the AddPermissions method, in which the grantedByUserId parameter is available even though domain logic requires that the "grantedByUserId", when adding permissions, is that of the currently authenticated user. The simplicity of the data access class and its emphasis on direct data access, without regards for the domain rules, makes it a great, re-useable tool for domain classes that should be easy to understand as well. The single method for adding permissions in the repository is used by domain classes to handle permissions when adding, sharing or deleting documents, and future development should only increase this number of uses.

Having shown the simple data access class, Figure 18 shows an example of how these classes are used to provide authentication and data access according to domain logic. The document authorization service class featured in this figure is one of the services that utilize the aforementioned document permissions repository.

```
public class DocumentAuthorizationService :
IDocumentAuthorizationService
{
    private readonly IAuthenticationService _authenticationService;
    private readonly IDocumentPermissionsRepository
_permissionsRepository;
```

```
    public bool AuthorizeUser(Guid documentId,
                              DocumentPermissions
requiredPermissions)
    {
        var currentUserId = _authentication.CurrentUserId;
        var hasPermissions =
_permissions.HasPermissions(currentUserId,
                                                documentId,
requiredPermissions);
        if(!hasPermissions)
            throw new UnauthorizedAccessException(
                "User does not have the required Document
Permissions: " +
                requiredPermissions.ToString());
        return hasPermissions;
    }
    public void ShareDocument(Guid documentId, int userToShareToId)
    {
        AuthorizeUser(documentId, DocumentPermissions.Owner);
        var currentUserId = _authenticationService.CurrentUserId;
        _permissionsRepository.AddPermissions(userToShareToId,
            currentUserId, documentId, DocumentPermissions.Shared);

    }
    // Some methods omitted for clarity.
}
```

Figure 18: Using simple repository methods to implement domain specific methods.

To begin with, note that neither of the two methods in the service class accepts any parameters for a user ID, as opposed to the methods of the permissions repository. The user ID is provided by an authentication service and used as input in the following call to the permission repository, and as such the service handles that part of the domain logic. This gives a layer of insurance that authentication is handled properly, and that a user will not be able to circumvent authentication by simple methods such as parameter manipulation. The only way to sneak an incorrect user id into this code would be to compromise the authentication service, so this should be considered when investigating the security of the system. The service also provides authorization, an example of which can be seen in the ShareDocument method in Figure 18, as the method performs a check to ensure that the currently authenticated user has “Owner” permissions before being allowed to share a document. On a final note for Figure 18, note that the methods of the service class have names that are closely tied to the domain logic of the application, such as the ShareDocument method.

```
public class DocumentAuthorizationService :
IDocumentAuthorizationService
{
    private readonly IAuthenticationService _authentication;
    private readonly IDocumentPermissionsRepository _permissions;

    public DocumentAuthorizationService(
        IAuthenticationService authenticationService,
        IDocumentPermissionsRepository permissionsRepository)
    {
        if (authenticationService == null)
            throw new
ArgumentNullException("authenticationService");
        if (permissionsRepository == null)
            throw new
ArgumentNullException("permissionsRepository");
        _authentication = authenticationService;
        _permissions = permissionsRepository;
    }
}
```
Figure 19: Dependency Inversion

Before concluding the implementation I will first exemplify how a complex domain action is implemented according to the design discussed so far, to show how testable and readable the resulting code turns out to be. To do this I will cover the handful of tasks it actually takes to upload a document in this application, and explain how this complex method is built. The process of uploading a document is not that simple, and even if the client side of things is ignored it is still quite a long to-do list, as the method needs to take care of the following:

- Upload document details, including the ID of the user
- Extract annotations from the document
- Upload annotations
- Remove annotations from the document prior to upload
- Upload the document
- Add permissions for the user to use the document

Now this might sounds like a lot to handle for a single method, especially if we take the single responsibility principle into account. However, given the combination of the simple data access repositories along with the services mainly being responsible for handling repositories according to domain logic, the result is actually quite neat. The code in Figure 20 below shows the service method for uploading a document: the AddDocument method of the DocumentService class.

```
public Guid AddDocument(Stream documentStream, string fileName,
string title)
{
    // Create and store document details.
    var currentUserId = _authenticationService.CurrentUserId;
    var documentId = Guid.NewGuid();
    var addedDate = DateTime.Now;
    var documentDetails = new DocumentDetailsDto(documentId,
title, fileName,
                                                 currentUserId,
addedDate);
    _documentDetailsRepository.AddDetails(documentDetails);

    // Extract annotations from document and store them.
    var annotationDtos =
_pdfEditor.GetAnnotationsFromDocument(documentStream);
    var textAnnotations =
_pdfEditor.FilterTextAnnotations(annotationDtos);
    _annotationsRepository.AddAnnotations(documentId,
textAnnotations);

    // Remove annotations from document and store document.
    var newDocumentStream =
_pdfEditor.RemoveAnnotationsFromDocument(documentStream);
    _documentsRepository.AddDocument(documentId,
newDocumentStream);
    newDocumentStream.Close();

    // Add owner permissions to document.
    _permissionsRepository.AddPermissions(currentUserId,
currentUserId, documentId,
                DocumentPermissions.Owner |
                DocumentPermissions.Read |
                DocumentPermissions.Annotate);
    return documentId;
}
```

Figure 20: The AddDocument method of the DocumentService class.

The bold texts highlights services and repositories used for data access, and besides that the method is also dependent on the document manipulation methods of the IPdfEditor interface. I will admit that the amount of service and repository dependencies in this method could at least potentially indicate a code smell, and that the single responsibility principle might be stretched a bit in this method. However, it is not a complicated method to understand; on the contrary it is very well readable even without the comments and it should even be easy to understand the domain logic the code enforces, just by reading the code. It is also an easily testable method and I will show how this is done, bit by bit, in the chapter covering tests, chapter 7.

5.4.4 Conclusion

This design creates a domain logic layer with the primary purpose of separating the dependencies and general volatility of data access classes from the static and fixed nature of the classes in the domain layer.

Similar to the data access layer, the design makes good use of most of the SOLID principles, which should indicate at least some level maintainability and adaptability. The single responsibility principle is a key player in keeping the responsibilities of components well separated, and the interface segregation principle is used to great extend as well, as we choose to implement only a small logical subset of data access features for each repository. Of course, as we depend upon interfaces for dependencies throughout this design, the dependency inversion principle is applied as well, and a dependency injection framework manages the creation of these classes as well.

The most important aspects of the design is its emphasis on ensuring that repositories retain a high level of adaptability and re-usability while also ensuring that domain services are highly decoupled from outside influences, to keep them as static and fixed as possible. This ensures that the domain level should not require changes for any reason except when the rules of the domain need to be adjusted. This should also encourage thorough domain-level testing and improve the maintainability of the solution. The design of the domain layer makes the implementation of domain logic in service classes easy to read and understand, and this should further improve both the testability and maintainability of the domain layer and the solution as a whole. Finally, services currently enforce authorization themselves, but the implementation supporting this is kept highly adaptable.

5.5 Data Storage Options

This section covers the topic of data storage in which several different storage techniques are used. The requirements for data storage varies a lot depending on the data and how it is used and accessed in the system, and we will cover how different sets of data is stored in this solution, and explain the reasoning behind the choices made along the way. The solution uses the Microsoft Azure cloud for storage exclusively as this was a simple way to get access to these different types of data storage, but precautions have been taken towards minimizing our dependency on this one provider and we will cover these precautions as well.

The storage of document files which utilize a type of storage called a “blob” in the azure cloud. Secondly we have the document annotations which are stored in a simple key-value store; a storage method referred to as “table storage” in the azure service. And lastly we have data entities such as users, permissions and document details which are all related to other data entities; these are naturally well suited for a relational database structure, such as an SQL server.

Blob Storage for Files
Blob storage is optimized for storing large amounts of unstructured data, and is commonly used for streaming video or serving images or documents directly to a browser.

Key-Value Storage for Annotations
A simple key-value store is used for storing annotations. There are no requirements for any relational comparisons between annotations, so a relational storage is not necessary.

SQL Storage for Relational Data
Relational databases are often the storage method of choice when it comes to storing user profiles, as the need for managing relationships between users and specific sets of data often arises.

5.6 Supporting PDF Documents

PDF is perhaps the most widely supported format for displaying documents. The fact that it is widely supported makes it a very valuable and arguably essential format to support, at least for a document-centric system such as this. Supporting the Portable Document Format (PDF) format for input and output of documents necessitates that the system is able to manipulate PDF documents in a few specific ways. This section covers the requirements for proper integration of the PDF format, as well as how the system manages to provide these requirements.

5.6.1 Purpose

As mentioned in the introduction, PDF is a widely supported format and this makes it a very valuable and arguably essential format to support. It may very likely be the only format necessary to support, as almost any other document format can be converted into a PDF document with very little effort. In fact most popular document editors support this conversion to PDF by default, meaning that most users with a document in just about any format is just a click away from a document in PDF format. As such, providing support for PDF effectively provides support for a wide selection of other formats as well, and that is exactly the purpose of supporting PDF documents.

5.6.2 Functional Requirements

To support input and output of documents in PDF format, the system must be able to manipulate a PDF document in a few specific ways. PDF documents are inherently not well designed in regards to editing the document, so to assist in this process a 3rd party PDF library was used: PdfSharp. This chosen library was one of only a handful of C# implementations, and was chosen simply because it was most convenient; a brief analysis did conclude that it met the necessary requirements though.

Uploading a document requires that the system is able to read annotations from the document. This allows the system to convert the annotations to the system-specific format and store them separately in the database, in order to present them later. Uploading a document also requires that all annotations are removed from the document prior to upload, as the viewer needs to display a clean document without any PDF annotations. When a document is displayed, the annotations previously extracted from the document and stored in the database will be rendered separately in the viewer, on top of the clean document.

The system only requires one additional method at this point, and that is to be able to merge annotations into document. This is necessary when the system needs to serve a document to the user that includes the separately stored annotations, if a user wants both the document and the annotations in PDF format.

To summarize, these are the methods required to support the PDF format for each of the two given scenarios:

To receive documents, the system must be able to:

- Read annotations from the document.
- Remove annotations from the document.

To serve documents, the system must be able to:

- Merge annotations into the document.

5.6.3 Design Considerations

A quick analysis of the functional requirements concludes that the system needs to be able to convert annotation data from the format of the system into PDF format, and vice versa. To assist in this process a 3rd party library for processing PDF documents is used, as this provides the most essential document processing capabilities with very little effort.

Conversion from the format of PdfSharp to the system format is a simple process. All the property elements of PdfSharp are able to convert their internal data into the equivalent string in a PDF document, meaning that any property such as a Date or a Rectangle can be converted into a string that corresponds to that property in the PDF format. Using these conversion methods the PdfSharp properties are easily converted into strings and used to populate data of the system-specific annotation structure.

Converting from the system format to the format of PdfSharp is another matter entirely though. To convert annotation data into PdfSharp proprietary annotations it is necessary to build the annotations manually, and this requires setting a number of necessary properties such as name, title, rectangle and so on. This process is complicated by the fact that the property types of PdfSharp are very similar to those of the PDF format, and those properties naturally need to be of the correct type. Some type conversions are naturally necessary and also expected, but even mistaking a String type property with a Name type property can lead to display errors in the document.

This process is of course further complicated by the choice made earlier in development: that the data structure used for storing annotation data in the system is different to that of the PDF format. I still believe this decision is the better alternative though, as the system should benefit from having its own custom annotation structure. Not being dependent on the PDF format and not having to support all aspects of it should be beneficial to the system as a whole. Being dependent on the PDF format could also introduce dependencies towards one or more proprietary PDF properties. Taking care of support for PDF properties in the conversion process exclusively would also constitute an intuitive and well-defined set of responsibilities in this regard.

To conclude the discussion so far, the conversion process both to and from system format should be carefully considered. The data structure of annotations should be designed to support our own interpretation of what annotations should be capable of, but the conversion processes have to be considered as well in order to properly support both import and export in PDF format. There are also a few considerations to be made in regards to the development of conversion methods:

- The conversion process from PDF to system format is quite simple.
    - This should be easy to implement as well as change entirely at a later time – Very little code is required.
- The conversion process from system format to PDF is complicated.
    - This component should be developed with emphasis on a high level of maintainability and extensibility.

5.6.4 Implementation

This section will cover the components developed to provide support for the PDF format. As stated in the requirements section, only a few specific methods are required to support PDF documents in this system, and the interface defining the three necessary methods is presented in Figure 21 below.

```
public interface IPdfEditor
{
    List<AnnotationDto> GetAnnotationsFromDocument(Stream documentStream);
    Stream RemoveAnnotationsFromDocument(Stream documentStream);
    Stream AddAnnotationsToDocument(Stream documentStream,
                                    IEnumerable<AnnotationDto> annotations);
}
```

Figure 21: The IPdfEditor interface, exposing methods for manipulating PDF documents.

This interface decouples all dependencies related to PDF manipulation from the rest of the application, according to the design of solution structure. This ensures that no dependencies related to the PDF format leak into the application, and it also makes for an intuitive integration point for supporting any additional document format. If any additional document formats were to be supported in the future, this could be accomplished by simply implementing another IPdfEditor-based component specifically for the new document format.

Most of the implementation of the IPdfEditor interface will not be covered in detail, as there are few noteworthy design considerations or benefits in the implementation itself. The process of creating annotations in PDF format has been well-designed though, and this should benefit the system overall and in particular in regards to extensibility. As mentioned in previous discussions, creating an annotation in PdfSharp requires the creation of an annotation class along with a series of property classes for that annotation. The property classes have to be of the correct type, which are similar to the types available in the PDF format. The PDF format has an official set of definitions for the naming of properties, e.g. a property with the key "/Subtype" must be of type Name, the property of key "/Contents" must be of type String and a property with the key "/Rect" must be of type Rectangle. The correct type to convert to when converting to PDF format is therefore defined by the key or name of the property. This is the main method used to decide how to convert system properties into PDF properties, as I have no method of detecting the correct type to convert to besides adhering to the standards of the PDF format. The PdfAnnotationBuilder class shown in Figure 22 provides a builder class or a factory to support this process, and it is quite well-designed in terms of extensibility and further development, if I may say so.

```
public class PdfAnnotationBuilder
{
    public Dictionary<string, PropertyConverterBase>
PropertyConverters
                                                        { get; set; }

    public PdfAnnotation CreatePdfAnnotation(
                                  Dictionary<string, string>
properties)
    {
        var annotation = new PdfTextAnnotation();
        AddPropertiesToPdfAnnotation(annotation, properties);
        return annotation;
    }

    public PdfAnnotation AddPropertiesToPdfAnnotation(
       PdfAnnotation annotation, Dictionary<string, string>
properties)
    {
        foreach (var property in properties)
        {
            var key = property.Key;
            var value = property.Value;
            if(PropertyConverters.ContainsKey(key))
            {
                var propertyConverter = PropertyConverters[key];
                propertyConverter.AddProperty(annotation, key,
value);
            }
        }
        return annotation;
    }
}
```
Figure 22: A builder class for converting annotations into PDF format.

The important thing to note in this class is the dictionary of property converters at the very top of the class. This contains a number of property converters able to convert the string data of the system format into the correct property type in the PDF format. Future developments may want to provide support for converting objects into PDF format, instead of just strings, but in the current system it is much simpler to have strings as the main source of data.

To provide a property converter for each different property type, an abstraction is necessary, and this is provided by the PropertyConverterBase class, presented in Figure 23.

```
public abstract class PropertyConverterBase
{
    public bool ThrowOnFormatException { get; set; }
    public abstract void AddProperty(PdfAnnotation annotation,
                                     string key, string value);
    // Several supportive methods have been omitted.
}
```

Figure 23: The base class for property converters.

The base class provides the necessary abstraction in regards to the AddProperty method, and besides that a few methods for easily handling exceptions is included as well. The intuitive approach is to avoid exceptions in the conversion layer entirely, but if a method to enable them is easily supported, it could be useful to have exception enabled for testing purposes.

Figure 24 below features one of the property converters, the one responsible for converting a rectangle string such as this “[20,20,40,40]” into a proper PdfSharp rectangle.

```
public class RectanglePropertyConverter : PropertyConverterBase
{
    public override void AddProperty(PdfAnnotation annotation,
                                     string key, string value)
    {
        PdfRectangle rect = GetRectangle(value);
        if (rect == null)
            ThrowFormatException(key, value, "Rectangle");
        annotation.Elements.SetRectangle(key, rect);
    }

    public static PdfRectangle GetRectangle(string rectString)
    {
        var trimmed = rectString.TrimEnd(']').TrimStart('[');
        var points = trimmed.Split(' ');

        var values = new List<double>();
        foreach (var str in points)
        // Simple math omitted.
    }
}
```

Figure 24: The property converter for converting a date.

It should be clear that the GetRectangle() method converts a rectangle string into a PDF rectangle and the AddProperty method then adds this rectangle to the list of elements in the PdfAnnotation object.

The development of this set of property converters has one great benefit, and that is the fact that this design allows for a great deal of code re-use. This is easily illustrated when the annotation builder class is initialized, as shown in Figure 25. The property converters are mapped to each of the properties they are responsible for converting.

```
var nameConverter = new NamePropertyConverter();
PropertyConverters.Add(PdfAnnotation.Keys.Subtype, nameConverter);
PropertyConverters.Add(PdfAnnotation.Keys.Type, nameConverter);
PropertyConverters.Add("/Name", nameConverter);

var dateConverter = new DateTimePropertyConverter();
PropertyConverters.Add(PdfAnnotation.Keys.M, dateConverter);
PropertyConverters.Add("/CreationDate", dateConverter);
```

Figure 25: Mapping property converters to keys.

The property converters currently support seven of the most essential property types, and are used to convert 15 different properties. As additional features are increased and more PDF properties are used, few additional converters should be necessary, while the additional properties will be easily support by the existing property converters.

5.6.5 Future Development

A minor section discussing the future development is included, because there are a few things to keep in mind in this regard.

The data structure for annotations in the system is not yet complete, and should be designed iterative and with the system itself as the primary concern. Any introduction of data formatted according to the PDF format should be carefully considered and not done simply to correspond with the PDF format. With that said, choosing a similar structure when it makes sense to do so is not discouraged at all; it does make integration easier and if the structure is used by other vendors it is likely not a bad design.

The annotation converters should be considered essential components in the development of new annotation features. In order to support anything new, changes or at least additions to the data structure of annotations will likely be required, and this affects the converters. It is essential that the data structure of any new annotation features can be properly integrated in both the viewer and the conversion process, and the conversions necessary to support both import and export of any new annotation artefacts should be considered early in such developments. Prior to development of new features, existing solutions with similar features should be researched, as the structure used by other popular vendors in the PDF editor business might need to be supported – We might even learn a thing or two as well.

The current implementation using PdfSharp has a minor problem in its current state though; it cannot handle documents with positional-links in them, such as links in the index allowing a user to click the headline in the index to have the document re-positioned to the position of that headline further down in the document. There is a simple fix for this problem though, that involves using another 3rd party library, iTextSharp, to provide this functionality.

5.6.6 Conclusion

Support for PDF documents was successfully integrated and the system is able to manipulate PDF documents as required. The system supports conversion of annotations from system format to PDF format, and vice versa. The conversion process has been well designed in terms of extensibility in particular, and the maintainability of the conversion layer should be relatively high. The 3rd party framework assisting the processing of PDF documents should assist in providing the necessary features to further extend the conversion process.

The adaptability of the system is ensured by the fact that the PDF dependent components are decoupled from the rest of the system, and other format should easily be supported by implementing the few simple features of the IPdfEditor. The IPdfEditor should actually be renamed to IDocumentEditor, because it is obviously not related to the PDF format or the PDF library in any way.

The PDF format could be further decoupled from the internal data of annotations, as some notions such as the string for rectangles remain, but this should happen iteratively as the structure for annotation data evolves to best support the system.

5.7 Conclusion

The server provides for all its functional requirements, and is in general a well-structured system. The solution keeps a high level of code quality across all components, and the maintainability index scores of the code metrics indicate a very maintainable server-side. Dependencies are also very well de-coupled throughout the system, which improves the overall adaptability of all components.

The domain logic is kept well de-coupled from any outside influences, and it should be able to remain as static and fixed as possible. Changes in domain logic rules will naturally still require changes, but the fact that the layer has no dependencies makes it easy to change in this regard. It is important to note that the layer is fixed and static only by choice, and that it is easily adaptable to any necessary changes. The relationship developed between domain services and data access repositories is quite successful, and provides several critically important benefits for both layers. The repositories are allowed to maintain a high level of adaptability, and the integrity of the domain level is still ensured. The implementation of some data access repositories has already been changed several times, with no effect on the domain layer in any way.

Authentication and Authorization are kept adaptable to change, even without affecting domain classes, access control or tests. Data storage supports a few different types of storage, which should help ensure optimal storage options for current and future requirements. PDF Manipulation tools are well-designed in regards to further development, and the PDF format is kept largely separated from the system. Support for new document formats should be seamlessly integrated, and the PdfSharp implementation can be exchanged is necessary in case a good 3rd party tool becomes available instead.

6 Security Analysis

This section will cover an analysis of the solution in regards to security. The analysis will focus on the top ten most critical web application security flaws as stated in the list provided by the Open Web Application Security Project (OWASP). OWASP is a non-profit, vendor-independent IT security organization formed in 2001, and is well-known for its top ten list of most critical web application security flaws.

"The OWASP Top Ten provides a powerful awareness document for web application security. The OWASP Top Ten represents a broad consensus about what the most critical web application security flaws are. Project members include a variety of security experts from around the world who have shared their expertise to produce this list."[9]

To begin with, here are the top ten most critical security flaws in web applications:

- A1 Injection
- A2 Broken Authentication and Session Management
- A3 Cross-Site Scripting (XSS)
- A4 Insecure Direct Object References
- A5 Security Misconfiguration
- A6 Sensitive Data Exposure
- A7 Missing Function Level Access Control
- A8 Cross-Site Request Forgery (CSRF)
- A9 Using Components with Known Vulnerabilities
- A10 Un-validated Redirects and Forwards

The security of the solution is analysed in regards to these security flaws, and most of the flaws will be discussed in detail in their separate sub-sections below. Some flaws in the top ten are quite similar though, at least in the context of this solution, and the following six flaws will therefore be discussed together in only three different subsections:

[9] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

A2 and A10 are not significantly relevant to the solution in its current state, but both of them relate to the MVC framework and will be discussed together in section 6.2: Improper Authentication and Redirects (A2+A10).

A4 and A7 are similar in that they are both covered in the topic of improper access control, and will as such be covered together in section 6.6: Improper Access Control (A4+A7).

A5 and A9 are also related, and touch the topics of security configurations and updates; these will be discussed together in section 6.4: Security Misconfiguration (A5+A9).

The rest of the vulnerabilities will be discussed in their separate sections, and each section will include a verdict in the end to conclude the analysis of each separate vulnerability.

6.1 SQL Injection (A1)

SQL Injection occurs when an intruder is able to inject an SQL command into an input parameter and trick the server into interpreting the input parameter as an actual SQL command. This effectively allows an intruder to run commands on the server, which is a serious vulnerability to say the least, as an attacker with enough knowledge of the database will be able to run commands on any set of data. Being exposed to just a single SQL injection vulnerability could allow an intruder to drop entire tables or do something more subtle like gain access to another user’s account and personal data.

Whether an intruder wants to delete entire database tables or do something more subtle like gaining access to another user’s account or personal data, being exposed to just a single SQL injection vulnerability could allow for this to happen.

The current implementation uses an SQL database for some of the storage requirements, and as such SQL injection vulnerabilities should be considered. These vulnerabilities can be protected against by sanitizing input, either by validating input prior to running queries or by using parameterized queries to ensure that no input is interpreted as anything but text.

The current implementation uses Entity Framework as the framework for data access to relational data, and as such all SQL queries are built by the framework. The framework uses parameterized queries for all the basic data access methods it provides, meaning that any potentially malicious input will be interpreted as text and harmlessly stored in the database without question. This offers an effective layer of protection against injection vulnerabilities, and the fact that potentially malicious input is stored as text in the database is not a concern – If it becomes a concern at some point, input validation could be used to reject the input entirely.

6.1.1 Verdict

The solution is considered well-protected against SQL injection attacks as long as the data access framework is used correctly. If any custom data access features are made that directly builds or manipulates a SQL query string, special care should be taken to ensure the integrity of that query.

6.2 Improper Authentication and Redirects (A2+A10)

One of the primary recommendations against this set of vulnerabilities is to have a single set of strong authentication and session management tools. I have chosen not to put a lot of effort into researching how well the default MVC framework performs in this regard, and little-to-no effort has been made towards improving this area of the solution. This is in part because the choice of which method or framework to use for authentication and authorization in general have not yet been settled.

6.2.1 Verdict

Little effort has been made into researching this topic, but the current solution does pose some concerns regarding unencrypted connections and the management of sessions ID's. These areas should be thoroughly investigated at some point, to ensure the solution is secure. Besides these concerns the solution handles many other possible vulnerability indicators in this area, such as providing a secure storage for passwords and not exposing session ID in the URL. A10 vulnerabilities concerning un-validated redirects and forwards are taken care of by the extensive use of routing in the MVC framework. The solution is considered protected against redirects to other sites when the routing methods of MVC are used, as they will not allow for cross-site redirects. With that said, I am not completely confident that it is not possible for a developer of this solution to mistakenly implement a redirect that could enable an intruder to circumvent this protection.

As is common to several other vulnerabilities, the protection against the vulnerabilities discussed in this section will not be able to ensure any degree of protection if the site is vulnerable to XSS.

6.3 Cross-Site Scripting (A3)

This section will feature a thorough analysis of Cross-Site Scripting (XSS), as this is one of the most critical vulnerabilities to protect against. The solution happens to be a potential target for several different categories of XSS, and as such this vulnerability receives some extra attention to properly explain where and why this is the case. This section will first cover a few different categories of XSS and then discuss how the solution fares against them.

XSS exploits are possible when an intruder is able to inject a script into the client of another user and trick that client into running that script. There are several variations of XSS using different attack vectors and there are two important distinctions to be made in this regard. One is whether or not the attack is persistent; persistent being when the malicious script is stored on the server and then served by the server to any number of other users. Another is whether or not the attack is reflected; reflected being when the attack interacts with the server in some way, e.g. when a server receives malicious input and sends it back again. The non-persistent and non-reflected vulnerabilities are entirely client-side vulnerabilities, as the vulnerability itself does not interact with the server. The protection will therefore have to be done on the client exclusively, as the server will not be able to help prevent or even detect such attacks. In reflected attacks the server is involved, which gives another set of options for detecting and preventing the attacks. In persistent attacks the malicious input is being stored on the server and the options for detecting and preventing attacks are the same as that of reflected attack.

6.3.1 Persistent XSS

XSS vulnerabilities are persistent if they are able to store a script or some other type of malicious input on the webserver. If this malicious data happens to be retrieved from the database and used by the server in the process of rendering of webpages, then the script could be served by the server itself. In the worst case scenario the persistent input is not only served back to the user making the attack, but served to many other users as well, effectively turning the server into an attack vector for whoever put the script there. Truthfully, there might be even worse worst-case scenarios, and the scary thing about this type of vulnerability is its potential for turning servers into attack vectors and going viral by itself.

A simple example of a persistent XSS vulnerability could be if a user Alice changes her username to "Alice <script>Alert('I put a script on your site!')</script>". In this case the attack will likely popup a JavaScript Alert each time the server presents her name to her or anyone else for that matter, as this would make the server serve the script as well. Alice would not even see the <script /> part, as this would be interpreted as a script.

Figure 26: A persistent XSS vulnerability.

6.3.2 Non-persistent XSS

This section will cover the non-persistent attacks, both reflected and non-reflected. Non-persistent and reflected attacks are probably the least dangerous and/or tricky attacks, as they do not have the scary viral abilities of the stored attacks nor the stealth of the client-based attacks. As soon as they go into the category of non-reflected they get a bit trickier to deal with though, as the server will not be involved and as such it is gets a bit more difficult to detect and guard against these attacks.

A prime example of a reflected vulnerability is through an input parameter for a search, where a search for something malicious is sent to the server, but not stored, and when the server replies to the user it will state something along the lines of "Your search for: Kittens <script>…</script> returned 100.000 results".

Figure 27: A non-persistent, reflected vulnerability.

The thing about non-persistent and reflected attacks is that it is not that common to have input parameters sent to the server, not stored, and then returned. However, it is a lot more common to have input parameters sent to the server and at the same time immediately used in the client as well, especially in pages using Ajax and asynchronous calls in general. This occurs due to the fact that asynchronous requests often act upon user actions immediately and then await confirmation from the server to acknowledge that the action was indeed successful.

To exemplify a non-reflected attack, the current solution immediately creates a new document item in the list of documents when a document is uploaded. This is done to give the user immediate feedback and provide a smooth user experience. When the document has been uploaded successfully the server then responds and the document item in the list can be used to open a document. However, before the server responds the client creates the document item and displays the title and filename of the uploaded document, and if any of those contain any malicious input the client will simply run the script and the server will not be able to react in time.

Figure 28: A non-persistent, non-reflected attack.

This type of vulnerability inherits attributes from both reflected and non-reflected XSS; the server is involved and as such it is reflected, but the client will immediately display the input, without waiting for response from the server, and as such that part of the attack is considered non-reflected. The server will therefore not be able to assist in detecting or sanitizing the malicious input before the client is exposed to it, so client-side protection is essential.

In conclusion, the smooth user experience supported by using asynchronous methods has the disadvantage of changing some potential reflected XSS vulnerabilities into non-reflected vulnerabilities as well. The smooth user experience will therefore have to be accompanied by some client-side protection against XSS, as server-side sanitation will not be sufficient to handle the combined reflected and non-reflected XSS vulnerabilities.

6.3.3 Discussion

This section will discuss how the system fares against XSS vulnerabilities and explain why this is a high-priority vulnerability to protect against. When it comes to XSS vulnerabilities there are two intuitive ways to guard against them. One is to prevent malicious input from being stored and subsequently reflected by the server, and to do this input needs to be sanitized to not allow scripts. Another is to prevent any input from being interpreted as scripts on the client, and to accomplish this all text variables needs to be html encoded when displayed in the client.

In regards to html encoding the system is well-covered and this is currently the primary layer of protection keeping the system safe from XSS attacks – Without html encoding there would be XSS vulnerabilities all over the place, because the solution severely lacks input sanitation. Html encoding is currently handled almost exclusively by a 3rd party library, Knockout.js, which is included primarily to assist in developing well-structured JavaScript components using the Model-View-ViewModel (MVVM) pattern. Besides the text in the PDF document, just about every bit of text displayed on the website is html encoded by the Knockout.js library, and as such all the scripts that are not caught by input sanitation are simply displayed as text.

The fact that the solution is not vulnerable to XSS because we html encode everything is of course great, but as mentioned the lack of input sanitation should be a concern. Even though the solution could be considered well-protected at this point, as the malicious input is not being interpreted as scripts anywhere, the solution would be a lot better protected if it did not store malicious input on the server. While storing SQL injection inputs was not a great concern, storing and serving malicious scripts should be a concern, as they will continue to challenge the client-side protection against them as long as they are stored and served. Having such data stored is just an XSS vulnerability waiting to happen, and the fact that users can inject scripts into document titles and then freely share these documents to other users means that a persistent script could easily turn the server into an attack vector, e.g. if the text of a document title suddenly gets displayed somewhere without being html encoded.

The MVC framework providing the base of the web server actually supports input sanitation and by default refuses to accept query parameters with malicious input:

“System.Web.HttpRequestValidationException: A potentially dangerous Request.QueryString value was detected from the client”

The reason most of the input parameters are not sanitized even though MVC attempts to sanitize input is because MVC does not check Json data by default, and most of the data sent to the server is in Json format. An extra layer of protection could be provided by using proper input sanitation and validation of Json input, and this should definitely be looking into at some point. Having this extra layer would prevent scripts from being stored on the server, effectively providing two layers of protection against one of the most critical vulnerabilities: Cross-Site Scripting. As explained in some of the other analysis sections, XSS vulnerabilities can easily make it possible to circumvent the protection for several other vulnerabilities, which naturally makes XSS a high-priority vulnerability to protect against.

6.3.4 Verdict

XSS is perhaps the most important vulnerability to protect against, especially given the high number of potential areas this solution could be exposed to XSS. The solution is currently well-protected against XSS attacks, but this is based primarily on a single layer of protection: html encoding. Given the implications of XSS vulnerabilities it is highly recommended that the protection against XSS is further enhanced by adding another layer of security: input sanitation. Input sanitation is currently active for query parameters only and as such does not include the Json data received by the server; Json data constitutes most of the incoming data.

6.4 Security Misconfiguration (A5+A9)

This brief analysis covers the two list entries A5 “Security Misconfiguration” and A9 “Using Components with Known Vulnerabilities”, as they are quite similar in the context of this system. On a side note A9 was actually part of A5 in previous list from 2010; the 2013 list introduced A9 as separate entry and yet it immediately made the top ten. This is yet another reason to keep the solution well-structured and easily maintainable, as keeping 3rd party components up to date could be a serious security concern.

6.4.1 Verdict

Both of these security concerns are difficult to assess in a prototype environment such as this. It would not be very productive to go through the details of security configurations and making sure all components are up-to-date and safe to use, and as such the current solution is likely severely misconfigured.

It is highly recommended that proper security configurations for the system are investigated and validated prior to deployment. The current solution is not well configured in regards to security, and there are a number of additional configurations and safety measures that should be investigated and taken care of prior to deployment of this system.

6.5 Sensitive Data Exposure (A6)

Sensitive data exposure deals with the protection of critically sensitive data such as credit card information, passwords and other personal data that could be damaging to the user if it was to be exposed. The current requirement specification makes no mention of any particular security requirements in regard to document or annotation data. The solution might need to ensure some degree of confidentiality for more or less sensitive documents at some point in the future, but in the current state of the solution there are no sensitive data to worry about, except of course for the list of user passwords.

The list of user passwords should not be taken lightly though, especially due to the fact that the application encourages or even forces users to register and authenticate with their DTU mail account. This would effectively make the list a set of email and password combinations, and in these situations it is very likely that some users have others accounts with the exact same email-password combination. As such this information should be considered sensitive, and precautions should be taken to secure the information so that is it not exposed in the event of a security breach.

It is important to note that this solution will be used on a university campus, and as such there will likely be more than a few people in the vicinity with the skills to exploit simple vulnerabilities. The fact that the application will likely be accessed through a wireless network connection most of the time should further encourage an attention to detail in regards to security. To ensure the integrity of user accounts and passwords on the wireless network it will be necessary to provide some kind of secure connection, at least when negotiating user authentication.

When storing passwords it is important to ensure that they are stored with algorithms specifically designed for password protection; securing passwords can be a tricky process and with improper protection they can be surprisingly easy to guess using brute force. This task is currently taken care of by the MVC framework, and the default implementation of the MVC framework salts the password and then hashes it using the SHA-1 hash algorithm. At the very moment of writing this part of analysis, I realize that this is almost embarrassingly inadequate[10] – The SHA-1 hash algorithm is not even designed for password protection. I have to admit that I did not know the default password protection did not even use an actual password protection algorithm before writing this section of the report – I guess the devil really is in the details.

The password protection method will definitely have to include a proper password protection algorithm, and it is possible to support this using a 3rd party security library[11] and a single line of code to add the algorithm to the cryptography configuration of the MVC framework.

6.5.1 Verdict

The current solution does not contain any sensitive data to protect, except for user passwords. To ensure the integrity of passwords when connected through a wireless network, it will likely be necessary to provide an encrypted connection, at the very least during user authentication. To ensure the integrity of passwords in the event of a server breach, the passwords are now protected by proper password protection algorithms, and the solution currently supports protection through either of the following two algorithms: PBKDF2 or bcrypt. Both of these algorithms are computationally expensive and they are also adaptive in that regard,

[10] http://www.troyhunt.com/2012/06/our-password-hashing-has-no-clothes.html

[11] https://github.com/skradel/Zetetic.Security

meaning they can be adjusted to be increasingly expensive, computationally, to ensure they remain consistently resistant to brute force attacks as computational power increases.

6.6 Improper Access Control (A4+A7)

A7 vulnerabilities cover any sort of failure to provide access control for any given function, while A4 vulnerabilities cover the more specific topic of authorization bypasses through manipulation of insecure object parameters.

Common to both of these vulnerabilities is that they allow an intruder to circumvent the access control for a given function in the application. This type of vulnerability could occur if a web application attempts to validate a set of data client-side and then neglects to provide similar server-side validation, or if the server-side checks are entirely dependent on information which could potentially be manipulated and provided by the user. The latter of these two examples is exactly when A4 would occur; authorization bypass though insecure parameters.

To exemplify both of these, consider a web service method retrieving a document, which would likely accept a single parameter: the document id. The client application presents a list of documents the user has to access, and by using the client and playing nice, the user is not able to click his way into an unauthorized document. However, if no server-side validation is done when retrieving a document, an intruder would be able to simply change the document id parameter of the request and receive a document he or she should not have access to.

Figure 29: A vulnerable service, as a result of improper access control.-

6.6.1 Discussion

To begin with a slightly personal note, the vulnerabilities of A4 and A7 are not technical vulnerabilities on the same level as most of the other vulnerabilities of the OWASP Top Ten. They are a result of forgotten or mismanaged access control, not due to technical vulnerabilities but more so due to poor design or a lack of proper system maintenance, with developers struggling to find their way around a difficult-to-manage system. On the other hand, most of the other top ten vulnerabilities are more or less technical vulnerabilities that most developers, lacking any prior knowledge to that particular vulnerability, could easily fall victim to. The fact that A4 and A7 vulnerabilities do not have quite the same "Oh, interesting!" feel of the other top ten vulnerabilities do not make them any less of a security risk though, so they should of course be discussed, analysed and prepared for just as well. The primary layer of protection against these vulnerabilities is a well-structured and easily understandable system design along with a set of tests and methods to validate that nothing is missing, so while the vulnerabilities themselves might not be interesting, developing protection against them sure is.

On the topic of easily understandable design, a good rule of thumb is to always enforce server-side validation. In fact, providing client-side validation in a web-application should be considered a user experience related feature in most cases, as the server will most likely not be able to trust the client-side result anyway. Validating a set of data instantly client-side makes for a great user experience, but rest assured that set of data will need to be validated server-side as well to ensure any kind of integrity.

Having stated that the primary layer of protection is a well-structured and easily understandable system, let us take a look at exactly where in this system this helps prevent A4 and A7: Improper access control. Proper separation of concern is a concern in this system, and responsibilities are well defined and intuitive, or at the very least, they should be easy to remember. The domain services are the ones responsible for authenticating and authorizing users according to domain logic.

As such, our rules for access control are enforced in these classes, and in these classes only, and this design should make it rather simple to ensure that the domain logic rules in regards to access control are enforced correctly in all service classes. As the service classes are very much related to the domain logic, it should be intuitive to check that the rules of the domain are enforced correctly, and a thorough set of unit tests for these service classes will provide further validation.

On a side note, the access control features of MVC could have handled most of the access control requirements for this solution. This was decided against though, in order to have this core responsibility well-separated and restricted to the service classes only. The MVC framework does enforce some level of access control though, as it immediately denies any unauthenticated user access to the web API exposed by the solution, except of course for the method to log on.

6.6.2 Verdict

In conclusion, it is difficult to ensure that forgotten or mismanaged access control does not occur, as it is mostly a result of human error. It is possible to design a system that reduces that chance of such errors happening though, and the design of the solution defines a clear set of responsibilities in this regards, as domain services are responsible for enforcing access control. The domain services are designed to be very well testable as well, and it should not be very demanding to test and verify the well-separated domain services sufficiently enough to prevent this vulnerability from occurring.

6.7 Cross-Site Request Forgery (A8)

Cross-Site Request Forgery (CSRF) occurs when a site is able to successfully execute a request on another site. The vulnerability is most effectively exploited by specifically targeting users that are likely to be authenticated on the desired site to attack.

An example attack features an intruder sending an email with a link to some facebook page along with a link to a site the intruder has built or made into an attack vector. In a lucky coincidence the user first visits the facebook page and logs in, and subsequently visits the other site containing a script that executes a cross-site request targeting facebook. Since the user is now logged in to facebook, the cross-site request automatically includes the user's authentication information and because of this the request will be successfully authenticated by facebook even though the request was sent by a script on another domain. Facebook will believe the user sent the request, and the user will have no idea an attack just happened, besides the fact that his or her facebook page now features cats. Everywhere.

Figure 30: A Cross-Site Request Forgery (CSRF) featuring cats.

There are many ways in which a server can attempt to detect CSRF attacks by analysing the incoming request, but common to almost every one of them is that the request can be forged well enough to avoid detection. There is one commonly preferred method that works really well against

CSRF though, and it uses the concept of validation tokens. To authenticate requests based on validation tokens the server must construct and serve an unpredictable token, commonly a string, to use when sending a request. This token is usually generated along with the form element that submits requests and the token is then returned again along with the request. This allows the server to validate requests by checking the attached token, and the protection lies in the fact that a request from another domain will not be able to guess the correct token.

6.7.1 Verdict

Validation tokens are used for requests throughout the application, and the solution should be easily secured against CSRF. There are currently no tests to ensure that the one essential requirement is met: that service endpoints demand validation tokens. In order to ensure protection against CRSF in a production environment, the use of a method to ensure that this requirement is met is highly recommended; this could be accomplished either manually or through automatic tests.

On a final note, this is yet another vulnerability which cannot be effectively prevented if the site is vulnerable to XSS.

7 Test

The development and design of this system has had quite a lot of emphasis on ensuring proper maintainability and testability of the domain logic layer. It has been one of the critical areas of interest in the system, in an attempt to ensure the cost-effectiveness of the future development of the system. The maintainability and testability of the domain components have been considered throughout the development of the system, and have therefore been an important topic in most of the design considerations discussed throughout the server design chapter. Since the system design so heavily encourages proper testing of the domain layer and its components, the proper testing of a single class of the domain layer will be the focus points of this entire test chapter.

This section will first cover the concept of mocks, briefly, and then cover the components developed to assist in configuring a system under test. Following this will be a section covering the tests themselves, and the final section will feature a conclusion for the test chapter.

The single class to cover throughout this section is the document service class. Testing the document service class allows for the coverage of many aspects of testing, including mocks, authentication and authorization, as well as validation of proper data movement between internal dependencies. Testing of the document service class also benefits from the development of supportive classes to assist in configuring its many dependencies, and this is covered as well. This should cover most of the aspects of testing the tests and the supporting classes designed specifically for the document service.

7.1 Mocks

Mocks are simulated objects that are able to mimic the behavior of real objects in a controlled way. Mocks enable a developer to more easily test classes, by exchanging any class dependency with a mock that does exactly what is expected. This enables a simpler and more focused testing process, as the testing process of a class can be de-coupled from the influences of real dependencies, which will not always do exactly as expected.

In order to properly test the domain logic, a proper set of mock objects are needed to remove any outside influences entirely. The use of mock objects, or simply mocks, is particularly important for the document service class of the domain layer, as its method for adding a document has a high number of abstract dependencies – A topic also discussed in the design and implementation of this method, in sub-section 5.4.3 of the Domain Logic Layer section.

7.2 Configuring the System under Test

When testing the document service class a set of mock objects are essential, and these mocks will need to be configured according to the purpose of the test. To be able to create many different tests, the service will therefore have to be configured in many different ways to effectively support each test without affecting other tests. Many different configurations are therefore necessary to support proper test coverage, so a document service builder class has been implemented to assist the process of configuring mocks. This document service builder class will simply be referred to as the builder for the purpose of this discussion, and the document service class will be referred to as the service; the builder builds and configures the service. The builder is responsible for injecting a default configuration of mock objects into the document service class, as well as providing a set of methods to support additional configuration of mocks. This will simplify the build process of a testable service class, and the additional methods for further configuration of mocks will provide sufficient flexibility to configure the behavior of mocks according to the requirements of the tests.

The code for the builder class is shown below in Figure 31, and the core structure of the builder class is quite simple to explain. The builder contains a mock for each of its dependencies, which effectively removes all outside influences simple as that. To provide behavior for the set of abstracted dependencies which are now reduced to mocks, a set of variables are stored in the builder class to act as replacement return-values for the mocks to use. The mocks are now able to be properly configured to return certain values and objects when the service class attempts to call some of their methods. This explanation covers the core structure of the builder, and the code for just this structure is shown below in Figure 31.

```
public class DocumentServiceBuilder
{
    public Mock<IPdfEditor> _pdfEditor = new Mock<IPdfEditor>();
    public Mock<IAuthenticationService> _authenticationService =
new..
    public Mock<IDocumentAuthorizationService>
_authorizationService..
    public Mock<IDocumentRepository> _documentsRepository = new..
    public Mock<IDocumentDetailsRepository> _detailsRepository =
new..
    public Mock<IDocumentPermissionsRepository>
_permissionsRepository
    public Mock<IAnnotationRepository> _annotationsRepository = new
..

    private Guid _documentId = Guid.NewGuid();
    private Stream _document;
    private Stream _annotatedDocument;
    private int _userId = 0;
    private List<DocumentDetailsDto> _details;
    private List<AnnotationDto> _annotations = new
List<AnnotationDto>
    {
        new AnnotationDto(),
        new AnnotationDto()
    };

    public DocumentService Build()
    {
        _documentsRepository
            .Setup(d => d.GetDocument(_documentId))
            .Returns(_document);
        _pdfEditor
            .Setup(pe => pe.FilterTextAnnotations(
                           It.IsAny<List<AnnotationDto>>()))
            .Returns(_annotations);
        // 5 other default mock setup configurations omitted.
        return new DocumentService(
            _authenticationService.Object,
            _authorizationService.Object,
            _documentsRepository.Object,
            _detailsRepository.Object,
            _permissionsRepository.Object,
            _annotationsRepository.Object,
            _pdfEditor.Object);
    }
}
```

Figure 31: The core of the builder class. Additional methods are revealed in later figures.

As seen in Figure 31, the builder contains a mock for each dependency along with a set of variables to provide the necessary return values for mocks to use. The bottom half of the figure contains the Build method, which is responsible for providing the default mocks configurations before it builds the service. The builder finally instantiates the service as seen in the very bottom of the figure; the fact that the service follows the dependency inversion principle makes this a simple process, as all the mocked dependencies are simply injected the same way the actual dependencies would be.

So far the core structure of the builder provides a default configuration of mocks, and the default configuration should simply be built to provide the simplest approach for the selection of tests to cover. As mentioned the builder is also responsible for providing a set of methods to assist in configuring the default behaviour of mocks, to allow for configurations that more specifically target each test. A small selection of the methods responsible for providing additional configuration options is shown below in Figure 32.

```
public class DocumentServiceBuilder
{
    // Continued from previous figure
    public DocumentServiceBuilder WithAnnotations(
        List<AnnotationDto> annotations)
    {
        _annotations = annotations;
        return this;
    }
    public DocumentServiceBuilder WithReadPermissions() {
        _authorizationService
            .Setup(a => a.AuthorizeUser(_documentId,
                        DocumentPermissions.Read))
            .Returns(true);
        return this;
    }
    public DocumentServiceBuilder WithUserId(int userId) {
        _userId = userId;
        _authenticationService
            .SetupGet(a => a.CurrentUserId)
            .Returns(userId);
        return this;
    }
}
```

Figure 32: Another section of the builder class, providing additional options for configuration.

The methods presented take care of configuration in slightly different ways, and the WithUserId method both sets a variable in the builder and configures the authentication service to use this variable. The authentication service will therefore return the userId specified in the builder, when the property for CurrentUserId is accessed. This allows for easy configuration of access control tests, where the userId can be set correctly, incorrectly or perhaps even result in an exception. The WithReadPermissions method sets no variables, but simply configures the authorization service mock to return true if/when the service attempts to authorize the user and check for read permissions for the current document id. The WithAnnotations, makes no setup adjustments, but simply sets the value of a variable containing list of annotations. In this case the configuration is already done by the default configuration, which is shown in a previous figure: Figure 31. As defined by the default configuration, the variable set by the WithAnnotations method determines the value returned by the PdfEditor when extracting annotations from the document. This is a good example of reducing the potential for outside influences, as there is great potential for errors and exceptions in the process of extracting annotations from a PDF document. However, the test successfully prevents any such potential influences.

This is used by one version of the GetDocument method, where annotations are stored separate in the database

7.3 Testing

Figure 33 shows a test validating one of the requirements of the DeleteDocument method of the document service class. Note where the builder is initialized and configured, and how easily readable the configuration options are.

```
[TestMethod]
public void DeleteDocumentRemovesDocumentPermissions()
{
    var documentId = Guid.NewGuid();
    var userId = 222222;
    var inititalDocument = GenerateDocumentStream();
    var expectedPermissions = DocumentPermissions.All ^
                              DocumentPermissions.Owner;
    var builder = new DocumentServiceBuilder()
        .UsingDocument(documentId, inititalDocument)
        .WithOwnerPermissions()
        .WithUserId(userId);
    var service = builder.Build();

    service.DeleteDocument(documentId);

    builder._authorizationService
        .Verify(a => a.RemovePermissions(userId, documentId,
                                        expectedPermissions),
Times.Once);
}
```

Figure 33: Validating that the delete document method correctly removes document permissions.

Based on the builder configurations made, it should be easily understood that the current system under test is a service, and that the service is configured to simulate an authenticated user with a specific user id. Additionally, the user also has owner permissions for the currently used document. Having properly configured the system under test, the test proceeds to call the DeleteDocument method and then verifies that the proper methods with the expected set of parameters. The Verify method is a part of the Moq framework, and the mocks tracks the incoming methods calls they receive in order to provide these verification methods. As such, the test verifies that the RemovePermission call was made once, with the correct parameters, and that is the single responsibility for this test.

There will commonly be many tests just to cover single methods, even for a simple method such as DeleteDocument. Another important aspect that increases the amount of tests is to handle cases for exceptions as well, or in the following case, improper authorization. The test for validating that the DeleteDocument method throws an exception when attempting to authorize the user is shown below in Figure 34.

```
[TestMethod]
[ExpectedException(typeof(UnauthorizedAccessException))]
public void DeleteDocumentThrowsUnauthorizedAccess()
{
    var documentId = Guid.NewGuid();
    var inititalDocument = GenerateDocumentStream();

    var service = new DocumentServiceBuilder()
        .UsingDocument(documentId, inititalDocument)
        .Build();

    service.DeleteDocument(documentId);
    Assert.Fail("Did not throw on invalid permissions");
}
```
Figure 34: Validating that an exception is thrown on improper authorization.

Again, it should be easy to read the builder configuration and see that the service it not configured to authorize the user. The tests is set to expect an exception, by using a method attribute as seen above the method definition in Figure 34. If the exception is not throw, the test fails when reaching the assert statement.

Being able to configure the exact return values of all internal dependencies simplifies the process of validating that data is handled correctly. This is exemplified in Figure 35, where the process of extracting and storing annotations is validated.

```
[TestMethod]
public void AddDocumentExtractsAndAddsAnnotations()
{
    var userId = 2211;
    var document = GenerateDocumentStream();
    var annotations = new List<AnnotationDto>()
    {
        new AnnotationDto(),
        new AnnotationDto(),
        new AnnotationDto(),
    };
    var builder = new DocumentServiceBuilder()
        .UsingDocument(Guid.NewGuid(), document)
        .WithAnnotations(annotations)
        .WithUserId(userId);
    var service = builder.Build();

    service.AddDocument(document, "My FileName", "My title");

    builder._annotationsRepository.Verify(r =>
        r.AddAnnotations(It.IsAny<Guid>(), annotations),
Times.Once);
}
```
Figure 35: Verifying the process of extracting and storing annotations.

The test validates that the AddAnnotations method of the annotation repository is called correctly, and that the annotations stored are the ones received from the extraction method of the PDF editor. As explained previously in Figure 32, the WithAnnotations call defines the list of annotations returned from the PDF editor, when it extracts annotations from the document. Since the return value of the PDF editor is defined by configuring the builder during setup, the test is able to very effortlessly validate that the value used to call AddAnnotations is the correct value received from the PDF editor’s extraction method. If the test had to reliantly build a document with a set of annotations, and then have them extracted, and then verify that those annotations match the ones in the document, then the test method would be a lot more complicated and it would be testing more than just the document service.

8 Conclusion

The goal of this project was to develop a foundation for a web-based system for annotation of documents. The primary focus was to research and design a system able to most effectively provide the necessary features in a manner which compliments a well-designed system with a high quality of code.

In the analysis, the most critical areas of the system were assessed and a set of requirements necessary to ensure success in these areas was defined. Potential solutions to these critical areas when then thoroughly researched and tested in order to find the best set of solution to satisfy the requirements stated. As a result of the analysis, a set of system-critical components was chosen for further development and a model for how the quality of code should be assessed throughout design considerations and development was designed as well. This has proven critical in the design and development of the system, and has had a great influence in the resulting design.

The security of the system was tested against the top ten most critical flaws in web application security, and system was able to properly prevent most critical attacks. The analysis features a thorough discussion of the system and how well it fares against these attacks, and also provides recommendations towards future development and potential areas of concern. This should provide a good foundation for maintaining a high level of security throughout the system.

The testing section features a complete walkthrough of how a domain class is properly tested, and should provide confidence that the domain layer maintains a high level of testability. It should also help explain the responsibilities and purpose of the classes developed to support the testing process, and why they are necessary to support the highly configurable test environment.

9 References

Boehm, B. W. (1976). Quantitative Evaluation of Software Quality. *IEEE Computer Society Press Los Alamitos, CA, USA*.

Gal, A. e. (u.d.). *Trace-based Just-in-Time Type Specialization for Dynamic*. Hentet fra ACM Digital Library: http://dl.acm.org/citation.cfm?id=1542528

Grady, R., & Casswell, D. (u.d.). Software Metrics: Establishing a Company-wide Program. *Prentice Hall. p. 159*.

McCall, J. A., Richards, P. K., & Walters, G. F. (1977). Factors in Software Quality. Volume I. Concepts and Definitions of Software Quality. *General Electrics Co*.

10 List of Figures