A Web Based Medical Image Viewer

A Manuscript
Submitted to
the Department of Computer Science
and the Faculty of the
University of Wisconsin-La Crosse
La Crosse, Wisconsin

by
Christopher Treml
in Partial Fulfillment of the
Requirements for the Degree of
Master of Software Engineering
April, 2008
A Web Based Medical Image Viewer

By Christopher Treml

We recommend acceptance of this manuscript in partial fulfillment of this candidate’s requirements for the degree of Master of Software Engineering in Computer Science. The candidate has completed the oral examination requirement of the capstone project for the degree.

Dr. Kenny Hunt
Examination Committee Chairperson

Dr. Mark Headington
Examination Committee Member

Dr. Thomas Gendreau
Examination Committee Member
Abstract

Epic System’s software allows users viewing digital medical images to document their interpretations. However, at the project inception, Epic did not have an application that handles the actual viewing of the digital medical image. Currently Epic only interfaces with commercial medical image viewers used at their customers’ sites. Epic needs a viewer for their own internal development and testing. Prior to this project a viewer was developed over a summer internship for this purpose. This viewer was very simple and lacking in features compared to commercial and open source digital medical image viewers which are very large and complex software applications. This document describes the next version of this viewer. The viewer is a WPF Browser Application that interprets DICOM files and displays them to a user. In this version of the viewer functionality such as image filters, communication, and compression support will be developed.
Acknowledgements

I would like to thank my project Advisor Dr. Kenny Hunt for the support and guidance I received from him over the course of this project. I would also like to thank Davin Sannes, Dan Murphy, and Gillian Fortney at Epic for this continual support of this project. I would like thank Mathieu Malaterre and the GDCM community for all their help. Finally, I would like to thank my family and friends for their support and patience with my many long nights.
# Table of Contents

1. Background Information ................................................................. 1
2. Project Goals .................................................................................. 3
3. Software Development Process ...................................................... 5
4. Image Filters .................................................................................. 8
   4.1 Image Filters Functional Requirements ...................................... 8
   4.2 Image Filters GUI Requirements ............................................... 13
   4.3 Prototyping Image Filters ......................................................... 15
   4.4 Design for Image Filters ......................................................... 17
   4.5 Implementing and Testing Image Filters .................................. 19
5. Image Decompression ................................................................. 21
   5.1 Image Decompression Functional Requirements ...................... 21
   5.2 Image Decompression GUI Requirements ................................ 23
   5.3 Image Decompression Prototyping ........................................... 23
   5.4 Image Decompression Design .................................................. 24
   5.5 Implementing and Testing Image Decompression .................... 26
6. Communication .............................................................................. 28
   6.1 Viewing Information Communication Functional Requirements ...... 28
   6.2 Viewing Information Communication Requirements .................. 30
   6.3 Viewing Information Communication Prototyping ......................... 32
   6.4 Viewing Information Communication Design ............................. 35
   6.5 Viewing Information Communication Implementation and Testing .... 37
   6.6 Measurement Communication Functional Requirements .............. 38
   6.7 Measurement Communication GUI Requirements ....................... 39
6.8 Measurement Communication Prototyping ............................. 41
6.9 Measurement Communication Design ................................. 41
6.10 Measurement Communication Implementation and Testing ...... 46
7. Continuing Work ..................................................................... 48
8. Conclusion ............................................................................... 50
9. Bibliography ............................................................................ 51
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Image Filters Requirements</td>
<td>8</td>
</tr>
<tr>
<td>2. Image Filters Main Screen GUI Requirements</td>
<td>14</td>
</tr>
<tr>
<td>3. Image Decompression Requirements</td>
<td>22</td>
</tr>
<tr>
<td>4. Viewing Information Communication Requirements</td>
<td>29</td>
</tr>
<tr>
<td>5. Viewing Information Main Screen GUI Requirements</td>
<td>31</td>
</tr>
<tr>
<td>6. Study Closed Reason Dialog GUI Requirements</td>
<td>32</td>
</tr>
<tr>
<td>7. Measurement Communication Requirements</td>
<td>38</td>
</tr>
<tr>
<td>8. Add Text Measurement GUI Requirements</td>
<td>40</td>
</tr>
<tr>
<td>9. Delete Text Measurement GUI Requirements</td>
<td>40</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Window Center/Width Application Algorithm</td>
</tr>
<tr>
<td>2.</td>
<td>Window Center(c) = 128, Window Width(w) = 256</td>
</tr>
<tr>
<td>3.</td>
<td>Window Center(c) = 128, Window Width(w) = 128</td>
</tr>
<tr>
<td>4.</td>
<td>Window Center(c) = 192, Window Width(w) = 128</td>
</tr>
<tr>
<td>5.</td>
<td>Edge Enhancement Algorithm</td>
</tr>
<tr>
<td>6.</td>
<td>Edge Enhancement Pixel Map</td>
</tr>
<tr>
<td>7.</td>
<td>Main Screen with Image Filters GUI Requirements</td>
</tr>
<tr>
<td>8.</td>
<td>Image Filter Class Diagram</td>
</tr>
<tr>
<td>9.</td>
<td>Implemented Edge Enhancement Algorithm</td>
</tr>
<tr>
<td>10.</td>
<td>Image Decompression Class Diagram</td>
</tr>
<tr>
<td>11.</td>
<td>Main Screen with Viewing Information GUI Requirements</td>
</tr>
<tr>
<td>12.</td>
<td>Study Closed Reason Dialog</td>
</tr>
<tr>
<td>13.</td>
<td>Viewing Information Communication Class Diagram</td>
</tr>
<tr>
<td>14.</td>
<td>Main Screen with Add Text Measurement</td>
</tr>
<tr>
<td>15.</td>
<td>Main Screen with Delete Text Measurement</td>
</tr>
<tr>
<td>16-1.</td>
<td>Top of Final Class Diagram</td>
</tr>
<tr>
<td>16-2.</td>
<td>Middle of Final Class Diagram</td>
</tr>
<tr>
<td>16-3.</td>
<td>Bottom of Final Class Diagram</td>
</tr>
<tr>
<td>17.</td>
<td>General Measurement XML Format</td>
</tr>
<tr>
<td>18.</td>
<td>Text Measurement XML Format</td>
</tr>
</tbody>
</table>
Glossary

**DICOM**
Digital Imaging and Communication in Medicine – The standard for handling, storing, and transmitting medical images developed by the National Electrical Manufacturers Association.

**RIS**
Radiology Information System – A software application responsible for managing the information used and created in a radiology department.

**CIS**
Cardiology Information System – A software application responsible for managing the information used and created in a cardiology department.

**WPF**
Windows Presentation Foundation - A .NET based GUI framework developed by Microsoft.

**PACS**
Picture Archiving and Communication System – An application suite dedicated to storing, retrieval, distribution, and presentation of digital medical images.

**Modality**
The devices used to obtain medical images, such as an X-Ray machine.

**WPF Browser Application**
A WPF application that is delivered to the user in the same manner as a web page,
it is hosted on a web server and displayed within a browser, but is a fully fledged .NET Application.

**Window Center**
Analogous to brightness, this is used in the DICOM standard, along with Window Width, to decide how to present an image.

**Window Width**
Analogous to contrast, this is used in the DICOM standard, along with Window Center, to decide how to present an image.

**Visual Studio 2008**
Visual Studio 2008 is an integrated development environment developed by Microsoft for the creation of .NET and a variety of other Microsoft technology applications. It includes tools for writing, compiling, refactoring, and debugging .NET code.

**Iterative Process**
A software development process where a project is in small sections called iterations. Each iteration includes the requirements gathering, design, implementation, and testing for its section of the project.

**Throwaway Prototyping**
In software development refers to the creation of a model that will be discarded rather than become part of the final software.

**Photometric Interpretation**
The intended interpretation of pixel data, for example grayscale or RGB.

**MONOCHROME1**
A grayscale photometric interpretation where a value of zero means white and the maximum possible pixel value means black. This term defined in the DICOM standard [3].
MONOCROME2
A grayscale photometric interpretation where a value of zero means black and the maximum possible pixel value means white. This term is defined in the DICOM standard [3].

HLSL
High Level Shading Language – A shading language developed by Microsoft for use in DirectX.

GDCM
Grassroots DICOM, an open source C++ library for DICOM files, supports wrapping in C#, python, and Java

.NET
A Microsoft created software framework and virtual machine.

C#
An object oriented language developed by Microsoft that runs on top of .NET

COM
Component Object Model – A low level standard interface for software development, developed by Microsoft, mostly replaced by .NET.

DLL
Dynamically Linked Library – A software library containing premade code that can be used in application.

SWIG
Simplified Wrapper and Interface Generator – A software tool used to wrap libraries written in C/C++ with other languages, such as C#.

RIS
Radiology Information System – A software suite for handling the information gathered and analyzed by a radiology lab
1. Background Information

A PACS is the backbone of any digitally based cardiology or radiology department. This is the system that communicates with the image producing modalities, stores images, retrieves images, provides image viewing, and provides image filtering for analysis. The DICOM standard is the format that is used to encode almost all images created by medical modalities. Since the DICOM standard is widely used in the digital medical imaging field, a PACS must fully support the DICOM standard.

The DICOM standard is very diverse and allows for many proprietary customizations. This is because the DICOM standard was made to retroactively apply to digital medical images produced by modalities before the standard was created. The DICOM standard supports images of arbitrary color depth, multi-framed and single framed images of arbitrary sizes, and image formats using standard compression schemes, proprietary compression schemes, or raw image data.

The development of an image viewer for a PACS is an extremely large task. A viewer must be able to show multiple DICOM images simultaneously, including a heterogeneous mix of the different DICOM file variations. Medical image viewers must be able to play back multi-framed images at the encoded speed. They must also allow the user to apply filters to images in real time for analysis. A viewer must also be able to communicate with outside RIS or CIS
applications to pass data back and forth, data such as the patient currently being viewed and user created measurements.
2. Project Goals

Epic provides software applications that enable users to document their findings in medical images; however Epic does not have a PACS of their own. Instead Epic’s software interfaces with commercial medical image viewers (typically from a customer’s PACS) at customer sites. Epic previously did not have a medical image viewer of their own for vendor-agnostic PACS development and testing. To remedy this, prior to this project Epic developed a simple viewer over the course of an internship. This simple viewer allowed the user to view several DICOM files simultaneously without any image filters provided for analysis. The viewer did not have any functionality for communication with outside application. A DICOM parser was also created for this viewer that could read a very limited subset of the DICOM standard.

The goal of this project is to build upon Epic’s simple medical image viewer by adding needed functionality and support for a larger subset of the DICOM standard. These additional functionalities include the ability to properly display DICOM files with a provided Window Center/Width, allow the application of industry standard image filters, support a variety of DICOM files with compression schemes, and bi-directional communication with outside applications. These additions are very important to the project for a variety of reasons. For instance, a medical image viewer needs to support DICOM Window Center and Window Width to properly display images on screen. A user must also be able change image filters in real time to properly analyze a medical image.

A medical image viewer must be able to support DICOM files with compression schemes as DICOM images have a large variety of sizes, ranging
from a couple KB up to several GB. In larger DICOM files a compression scheme is almost always used and medical image viewers must be able properly interpret these compression schemes.

It is important to Epic for internal demonstrations to show how a medical image viewer integrates with their software suite. The addition of network communication to the viewer will allow them to properly demonstrate their ability to integrate with a customer’s PACS. This functionality will also allow Epic to better test their integration software with PACS systems.

Each of these functionalities must be designed to be easily extended in the future. This is especially important in this project as there are currently known functionalities that can and will be added to the viewer in the future, but are not included in this project due to time limitations. For instance, it is a well known fact that many modalities produce DICOM files with image data that is compressed using a proprietary compression scheme. The support of these proprietary compression schemes will not be part of this project, but is something that will be needed in a later version.
3. Software Development Process

There is a large selection of development processes in the software engineering field, each with its own unique strengths and weaknesses. The development process chosen for this project was the iterative process with some elements of throwaway prototyping added to it. The iterative process was chosen as similar requirements could be grouped together into iterations. Each iteration of functionality could then be completed before the next phase of development was begun. The iterative process also allows each iteration to be broken down into several smaller iterations if deemed necessary. This development process allowed the project to receive feedback from Epic early and often. This was important as it allowed the developer to quickly identify and fix mistakes that were found as the project was progressing. The iterative process fit this project well as the requirements could generally be easily grouped together (image filters, communication, etc...) allowing each chunk to be completed and fully tested. The iterations also supported regression testing as it was easy to test if new functionality broke any previously completely functionality.

The weakness of the iterative process is that the user will not see a fully completed project and may incorrectly believe that the software is completed earlier than it actually is. The iterative process also requires much more interaction with the client. Instead of one requirements gathering period with the client there are many throughout the project’s life cycle and not every client is willing to work with this. Epic agreed to have several requirement gathering session as required for an iterative process to work.
The project’s software development process also included some throwaway prototyping elements. Throwaway prototyping was selected in this project since many elements of the project problem domain were unfamiliar to the developer. After the analysis phase for each iteration a prototype was created to develop a working model for the design. One of the strengths of the throwaway prototyping process is that it gives the developer experience and knowledge in the area of development before starting design.

Like all software development processes, the throwaway prototyping process also has weakness which must be considered and managed. One of the most important weaknesses is the temptation of the developer to directly use the prototype in the final project. This must not be done as the prototype is just a proof of concept that is not properly designed, tested, or documented. If the prototype is used in the final project it is likely to cause many problems later in the software’s life cycle. Another problem with the throwaway prototype process is that it can give the clients a false impression that a feature is almost completed when it needs much more work and time. This weakness is shared by the iterative process, which made this weakness extremely prominent in this project and careful consideration was made to overcome this. To overcome this weakness the developer clearly defined to the project sponsor estimates for completely functionality and what demonstrations were prototypes.

In this project each of the main feature sets: image filtering, compression support, and communication were grouped into their own iterations. In this project only the communication iteration was subdivided into two iterations. This was done as the communication functionality to could be further divided into two categories: viewing information and measurement information. The viewing information was already present in previous version of the viewer which meant only the communication of this information needed to be developed. The measurement information required a measurements system to be developed in
addition to the communication. By subdividing this iteration into two, the communication framework developed in the first iteration could be leveraged by the second iteration.
4. Image Filters

This section provides an overview of the analysis, design, implementation, and testing of the image filter iteration. It was decided that this iteration should be done with no sub-iterations.

4.1 Image Filters Functional Requirements

Table 1 lists the functional requirements that were gathered for the image filter iteration. These functional requirements were gathered and reviewed over a series of meetings between the developer and Davin Sannes who was acting in his role of project sponsor. The following functional requirements were gathered for the Image Filter iteration.

<table>
<thead>
<tr>
<th>Requirement #</th>
<th>Requirement Name</th>
<th>Requirement Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF.1</td>
<td>Apply Window Center/Width</td>
<td>To apply window center and window with to a DICOM image.</td>
</tr>
<tr>
<td>IF.2</td>
<td>Apply Color Invert</td>
<td>To invert the color of a DICOM image.</td>
</tr>
<tr>
<td>IF.3</td>
<td>Apply Edge Enhancement</td>
<td>To “sharpen” the edges in a DICOM image so they stand out more.</td>
</tr>
</tbody>
</table>

Table 1. Image Filters Requirements

Each of these requirements, described in Table 1, was also required to fulfill several nonfunctional requirements. Each image filter must be able to be applied by the user in real time. For instance, a user must be able to change the Window Center/Width or turn off and on the color inversion in real time and see
the changes onscreen as they happen. These image filters must also be able to be applied to the same DICOM image at the same time in any combination, so for example, a user can have the color inverted and window center/window width being applied, but also have edge enhancement turned off.

The image filters must be applicable to DICOM movies and single frame images. IF.1 also has one extra stipulation in that it can only be applied to DICOM images with a photometric interpretation of MONOCHROME1 or MONOCHROME2. This is because Window Center/Width can only be applied to grayscale images and MONOCHROME1 and MONOCHROME2 are the grayscale photometric interpretations available in the DICOM standard.

Since the project developer was unfamiliar with the problem domain, a large amount of the requirements gathering process was devoted to understanding the image filters themselves. Window Center/Width is used to better view and analyze different parts in images. For instance, one Window Center/Width value might be best to view bone structures in an X-Ray image, while another value is best for viewing soft tissue. During the research of Window Center/Width it was discovered that Window Center/Width is somewhat analogous to brightness and contrast. However, Window Center/Width is only applicable to grayscale images as previously mentioned. The different Window Center/Width values are used to change the linear interpretation of grayscale values. The formal definition for the application of Window Center and Window Width is defined by the DICOM standard as the equation seen in Figure 1.

\[
\begin{align*}
if \left( x \leq c - 0.5 - \frac{w - 1}{2} \right) & \text{ then } y = y_{\text{min}} \\
else if \left( x > c - 0.5 + \frac{w - 1}{2} \right) & \text{ then } y = y_{\text{max}} \\
else & \quad y = \left( \frac{x - (c - 0.5)}{w - 1} + 0.5 \right) \ast (y_{\text{max}} - y_{\text{min}}) + y_{\text{min}}
\end{align*}
\]
In Figure 1, the term “c” means Window Center, “w” means Window Width, “y” means output, “x” means input, “ymax” means the maximum output value, and “ymin” means the minimum output value.

Before the generalized algorithm presented in Figure 1 could be implemented it was necessary that developer knew what the algorithm was trying to accomplish. To help understand what this algorithm is doing it is best to view a few examples and the output they produce. For the following examples we will be showing how different Window Center/Width values would affect an 8-bit image. Since this is an 8-bit image that means the input would be 8-bit (maximum pixel value of 255) and an 8-bit output (maximum pixel value of 255) different Window Centers and Window Widths look like the following:

![Diagram](image)

Figure 2. Window Center(c) = 128, Window Width(w) = 256

Figure 2 shows how Window Center/Width affects an image when Window Center is the median value and the Window Width covers the entire input range. This effect is the same as not applying Window Center and Window Width as no change occurs to the pixel values.
Figure 3. Window Center(c) = 128, Window Width(w) = 128

In Figure 3 Window Width has been reduced by one half. This means the top 25% of the input values are transformed to the maximum output value and the bottom 25% of the input values are transformed to the minimum output value. The middle 50% of the values are linearly transformed between the new start and end points. This change in Window Center/Width values from Figure 2 is analogous to increasing the contrast.

Figure 4. Window Center(c) = 192, Window Width(w) = 128

In Figure 4 Window Center has been increased to ¾ of the maximum input value, while the Window Width is still one half the maximum input value. Figure
4 is Figure 3 shifted to the right by 25%. This means the bottom 50% of the values are transformed to the minimum output value and the top 50% are linearly transformed between the new start and end values. This change in Window Center/Width values from Figure 3 is analogous to increasing the brightness of the image.

The meaning of edge enhancement was also a mystery in the beginning of this iteration. The process of edge enhancement is not defined by the DICOM standard as the application of Window Center and Window Width was. After researching what edge enhancement was it found to increase the differences between surrounding pixels that have different values. To do this the general algorithm was to take the differences of the pixels immediately surrounding a selected pixel and add the differences to the selected pixel’s value multiplied by a preset strength. While there are several algorithms for edge enhancement the chosen algorithm for edge enhancement can be seen in Figure 5.

\[
pixel = (\text{upper left pixel} - \text{bottom right pixel}) \times \text{strength} \\
+ (\text{top pixel} - \text{bottom pixel}) \times \text{strength} \\
+ (\text{left pixel} - \text{right pixel}) \times \text{strength} \\
+ (\text{bottom left pixel} - \text{upper right pixel}) \times \text{strength}
\]

Figure 5. Edge Enhancement Algorithm

The terms in Figure 5 refer to the pixels surrounding the pixel that the edge enhancement is being applied to. For instance, upper left pixel is the pixel that is to the immediate upper left of the pixel. Figure 6 is a visual aid of how the pixels used in Figure 6 relate. The term strength present in Figure 5 is a preset value used in the algorithm to determine how much the edge should be sharpened; a higher number causes the algorithm to have a greater effect.
The GUI requirements for the image filters needed to allow the user to change the Window Center and Window Width, turn edge enhancement off and on, and turn color inversion on and off. The GUI requirements must also follow the guiding UI principles from the previous version of the project. One principle was that the UI must not detract from the viewing of images themselves. The other principle was that users of this application will all be considered power users; this is because any user of a medical image viewer will most likely be using the application throughout most of their workday. From these requirements and guidelines the following GUI was conceived.
Figure 7. Main Screen with Image Filters GUI Requirements

<table>
<thead>
<tr>
<th>GUI Control</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Window Center Slider</td>
<td>Change the Window Center of an image.</td>
</tr>
<tr>
<td>Window Width Slider</td>
<td>Change the Window Width of an image.</td>
</tr>
<tr>
<td>Reset Button</td>
<td>Reset Window Center, Window Width, turns Edge Enhancement off, and turns Invert Color off.</td>
</tr>
<tr>
<td>Edge Enhancement Check Box</td>
<td>Turn Edge Enhancement on or off.</td>
</tr>
<tr>
<td>Invert Color Check Box</td>
<td>Turn Invert Color on or off.</td>
</tr>
</tbody>
</table>

Table 2. Image Filters Main Screen GUI Requirements

As seen in Figure 7 and described by Table 2, the image filter controls are very minimal. The controls also have a few properties that cannot be displayed in a static image. Each of the individual controls, such as the Window Center slider is described using a tool tip. It was decided to use tooltips, rather than the normal method of labels, for descriptions to preserve space and keep save as much of the
screen’s real estate as possible for the images. The image filter controls also fade into transparency when the mouse is not in the over the control, once again this is to keep as much of the screen’s real estate in use for viewing the images. The Reset Button is also provided to allow the user to quickly reset the image filters to the values they were the image was first loaded, this means the Window Center/Width values are set to their defaults, color inversion is turned off, and edge enhancement is turned off. Last, as can be seen in the bottom right image in Figure 7, non-grayscale images do not have the Window Center and Window Width sliders.

4.3 Prototyping Image Filters

The purpose for prototyping the image filters was to decide what techniques and technology would best fulfill the requirements. Two technologies were evaluated for use in fulfilling the image filtering requirements: WPF Effects and writeable bitmaps. WPF Effects were a new addition to WPF, with the first version only being completed several months before this project began. Writeable bitmaps are a WPF technology that allows a programmer to directly modify a pixel matrix as it is shown on screen. It was decided to look into WPF Effects first and then writeable bitmaps if WPF Effects proved to be insufficient.

WPF Effects use a DirectX technology known as HLSL. HLSL is most commonly used in graphic intensive programs such as video games. WPF Effects are used by WPF to apply pixel shaders developed in HLSL to affect the appearance of WPF controls. The benefits of using WPF Effects are the graphical changes are processed on the GPU whenever possible making sure the image filters will not slow down the application. Since WPF Effects are integrated into the basic WPF controls the WPF Effect technology they will also integrate well with the previous viewer’s design. A WPF Effect is essentially a .NET wrapper
for a HLSL module, with no logic of its own. This meant the algorithms the filters would be written in HLSL.

During the prototyping a WPF Effect for color inversion and edge enhancement were developed. These two prototypes helped to discover the strengths and weaknesses of WPF Effects. The color inversion prototype would be done first as the algorithm is simpler and will allow the developer to learn HLSL easier. The edge enhancement prototype would be done next, as this prototype would help to understand the more advanced features of HLSL.

One strength of HLSL was that it automatically handled the looping over the matrix of pixels. This helped to simplify the logic when writing each filter. One downside stemming from this is that HLSL does not provide a way to programmatically discover the width and height of the image is applied to as this is done internally. This meant the height and width of the image it was being applied to would need to be explicitly passed into it.

Pixel values and coordinates are represented in a normalized form in HLSL. This normalization proved to be a strength of HLSL as it allowed the filter algorithms to be simplified.

The largest limitation that was found is that only one WPF Effect could be applied to a WPF control at any given time and this project needed three. After researching possible solutions it was found that the best solution was to put all necessary effects into one module. During the research of this limitation it was found that the next version of WPF would allow multiple effects to be applied to one control.

After evaluating the WPF Effect technology it was decided that it would be used for the image filter functionality. Its benefits of solid integration and GPU support far outweighed the limitations that were found during the prototyping phase.
4.4 Design for Image Filters

In designing the image filters it was necessary to take into account the knowledge gained in the prototyping. Each filter algorithm needed to be implemented in HLSL and the WPF Effect module would need to have properties such as image width and height to compensate for the areas HLSL was lacking.

It was also decided that the image filters should only affect the image and not any measurements that are on the image. This required the annotation image control to have image effect property where it would make sure the effect was only applied to the image and not the measurements.

Another design decision was that the image filters should also be applied in the images’ thumbnail selectors as the thumbnail selector control exists to convey to the user what images are showing and what images are not. If the image filters did not apply to the thumbnail selector control it would not properly reflect the image it is representing, making the purpose of the thumbnail selectors moot.

During the design process it was discovered that not all grayscale DICOM files contained a value for Window Center and Window Width. For these files it was decided to give them default values of Window Center = ½ Maximum Pixel Value and Window Width = Maximum Pixel value. These default values will make the Window Center/Width filter have the appearance of no effect initially. However, since the Window Center/Width filter is still being applied their values can be changed to affect the image.

The image filter classes will need to be implemented so that the code used in each filter is clearly labeled and sectioned. This needs to be done so that once WPF supports multiple effects on one control each filter can be easily moved to its own WPF Effect class.
As can be seen in Figure 8, four main classes were created or modified. The DICOMEffects class is the C# wrapper class for the HLSL module which will contain the filter algorithms. The AnnotatedImage View was modified to have an imageEffect as a public property. This property when set will make sure the WPF effect is only applied to the image and not the measurements. The DICOMImage view was modified to contain the visual controls for the DICOMEffect class. These controls use WPF’s binding system to bind their values to the properties of the DICOMEffect class such as windowCenter. The ThumbnailSelector class was also modified to have a DICOMEffect be applied to
the image it is showing. The ThumbnailSelector’s effect is the same
DICOMEffect object as the one being applied to the AnnotatedImageView; this
will make sure the thumbnail will always reflect what the actual image looks like.

4.5 Implementing and Testing Image Filters

The implementation of the image filters deviated slightly from the original
design. The largest differences occurred in computing an edge-enhanced image
where the formula given in Figure 5 was replaced by the formula shown in Figure
9.

\[ \text{pixel}^+ = (\text{upper left pixel} - \text{bottom right pixel}) \times \text{strength} \]

Figure 9: Implemented Edge Enhancement Algorithm

In Figure 9 pixel refers to the pixel the algorithm is currently being
applied to. The term upper left pixel refers to the pixel that is up one row and left
one column from pixel. Bottom right pixel refers to the pixel that is down one row
and right one column from pixel. Strength refers to a preset value that determines
how strong the effect should be. The algorithm seen in Figure 9 produced an
output with no notable visual difference from the output produced from the
original algorithm seen in Figure 5. The implemented algorithm was then changed
to algorithm shown in Figure 9 as it required less computation making it more
efficient.

For testing the image filters several different methods were used. For
testing the color inversion filter an image was exported as a bitmap without the
color inversion filter applied, and then exported again as a bitmap with color
inversion filter applied. These two exported bitmaps were then compared to make
sure the values were in fact inverted. This was done for several of the sample
DICOM files provided by Epic. For testing the Window Center and Window
Width filter comparisons were performed with the free viewer UniPACs[5] to
make sure the same values would produce the same image in both viewers. Hand calculations of the provided DICOM examples of Window Center and Window Width were compared with the hand calculations of the final implementation algorithm to make sure the values were correct. The testing of edge enhancement was done by hand calculating the algorithm to make sure the correct values were being computed. All the image filters were also tested by human eye by the developer and the project sponsors at Epic.
5. Image Decompression

This section provides an overview of the analysis, design, implementation, and testing of the image decompression iteration. It was decided this iteration should be developed with no sub-iterations.

5.1 Image Decompression Functional Requirements

The following functional requirements were gathered for the Image Decompression iteration. These functional requirements were gathered from the DICOM standard and through a series of meetings between the developer and Davin Sannes acting in his role as the project’s sponsor.

<table>
<thead>
<tr>
<th>Requirement #</th>
<th>Requirement Name</th>
<th>Requirement Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID.1</td>
<td>Decompress JPEG Lossless Process 14 First Order</td>
<td>To decompress pixel data that has been compressed using JPEG Lossless Process 14 First Order</td>
</tr>
<tr>
<td>ID.2</td>
<td>Decompress JPEG Lossless Process 14</td>
<td>To decompress pixel data that has been compressed using JPEG Lossless Process 14</td>
</tr>
<tr>
<td>ID.3</td>
<td>Decompress JPEG2000</td>
<td>To decompress pixel data that has been compressed using JPEGLossless 2000</td>
</tr>
<tr>
<td>ID.4</td>
<td>Decompress JPEG2000 Part 2</td>
<td>To decompress pixel data that has been compressed using JPEGLossless 2000 Part</td>
</tr>
</tbody>
</table>
In prioritizing the requirements described in Table 3 it was found that the most important requirements were ID.1 and ID.2 as they are the most commonly used compression schemes in DICOM files. The next requirements in priority were ID.6 and ID.7. The other requirements were considered to have the same priority below ID.1, ID.2, ID.6, and ID.7. These requirements also had a several nonfunctional requirements that applied to them: the decompression must be done in memory and as this is a web based application may not need access to write to the file system.

While researching about these compression types it was found that the JPEG Lossless scheme is an almost unsupported standard. JPEG Lossless was developed as a late addition to JPEG at the behest of DICOM. JPEG Lossless is
not even supported by the Independent JPEG Group and its only popular use is in digital medical images.

At the suggestion of the project’s advisor several libraries were evaluated to see if one could be used for help satisfy these requirements. No libraries native to .NET were found which meant that any library that was chosen would need to be wrapped in .NET. One stipulation of this project was that all code must be owned by Epic. This stipulation meant that any chosen library must have a compatible license such as the MIT or BSD license.

During research several libraries were found that claimed to support the required compression types: GDCM, DCMTK, SourceForge JPEG, and Pegasus. After contacting the communities of these groups and testing their utilities it was decided to work with GDCM. GDCM was chosen since at the time of this iteration it had just started to support C# wrapping. The C# support for GDCM was in the very early beta phases. GDCM had an active for community for help which would prove to be very useful of the course of this iteration. The GDCM library was also tested to see if it would support the sample set of compressed DICOM files provided by Epic and after testing it was found to decompress all sample compressed DICOM files that were provided by Epic.

5.2 Image Decompression GUI Requirements

As the image decompression happens completely on the backend, no GUI requirements were necessary for this iteration.

5.3 Image Decompression Prototyping

The prototyping in this iteration had a large portion dedicated to getting GDCM to work properly with C# and this project. As this first portion of the prototyping phase progressed updates and patches that were developed for GDCM
were add to the GDCM source code to help GDCM support C# better. There were many problems that were overcome during the GDCM development phase such as getting GDCM to properly compile, getting GDCM to properly work with a WPF browser application, making the codecs in GDCM available to C# code, and updating the SWIG wrapping to bridge the gap between C++ and C#. After this phase of the prototype was complete GDCM was fully useable in the project with the prototyping being able to fully decompress JPEG lossless data.

After the GDCM library was updated and patched there was still one problem with the prototype, it was unable to support compression types other than JPEG Lossless, even though GDCM is able to. After much discussion with the GDCM community about this it was discovered that GDCM needed the pixel data to be given to it in its original fragmented form. In DICOM files when pixel data is compressed it is fragmented into several sections. This project originally combined all these fragments into one array. This worked for JPEG Lossless as JPEG Lossless data only had one fragment, but other compression types may have many fragments. This meant the design would need to change how the pixel data is currently read and stored.

After this change in the prototype was completed there were still some JPEG Extended DICOM files that were not working. After research into how GDCM decompresses the pixel data it was discovered that the first fragment, the basic table offset, is skipped in GDCM and not read. This meant the first fragment needed to be thrown out and not sent to GDCM. After this change the prototype worked for all sample DICOM files provided by Epic.

5.4 Image Decompression Design

During the design process of the image decompress iteration several goals had to be kept in mind: currently supported uncompressed files must not be
affected, it must be easily extensible so that future compression types can be
painlessly added, and it should impact the current design as little as possible.

Figure 10. Image Decompression Class Diagram

As can be seen in Figure 10, an abstract class called “Decompressor” was
added that had two public methods, one would decompress data for 8-bit data and
one would decompress data for 16-bit data. For this project there would be two
crude classes that implement this: “StandardDecompressor” and
“NonDecompressor”. StandardDecompressor would act as a bridging point to
GDCM, internally providing all necessary data conversion to tell GDCM what to
decompress and then return the uncompressed data. NonDecompressor would be an empty class, the reason for having this class is to keep all decompression, or in this case the lack of, logic inside the decompression classes. This allows the other DICOM Image classes to be unaware of decompression and instead just pass its image data through a Decompressor class always.

The type of Decompressor would be dependent on the transfer syntax. In the DICOM standard the different values and their meanings are explained. From the transfer syntax the correct Decompressor could be created. To do this it was decided to use a variation of the Factory pattern common throughout .NET. This variation has a static Create method in the abstract class, in this case the Decompressor class, which would return an object of type Decompressor. This provides the user an intuitive way to get a Decompressor object without having to know the details. For the design of this method it was decided to not use a factory class and instead just have the method do the work. This is because the decision is a simple comparison of the transfer syntax and does not include any real state or extra logic.

A few changes to the current design would have to happen, too. The way the DICOM Image classes currently read pixel data is to store the pixel data in one large array, regardless of fragments. Because GDCM needs the fragments to be kept separate the DICOM Image classes would need to store them in their fragmented state. Another change that would be needed was how pixel transformations were preformed. An example of a pixel transformation can be in a MONOCHROME1 image. The pixel values in a MONOCHROME1 image are transformed by inverting their values to normalize them. Currently pixel transformations are applied as the pixels are read for efficiency. This will have to be changed to be done after the Decompressor has decompressed the pixel data.

5.5 Implementing and Testing Image Decompression
During implementation testing was very important to make sure images that could currently be read correctly would still be read correctly after the changes. To do this an automated testing program was made that would try to read every sample DICOM file provided by Epic and export it as a png to a test directory. In this testing program there were two directories that the images could be saved to. The testing program would use a different directory each time it exported the DICOM files as pngs. This allowed the previous batch of images to be compared to the newly created images and make sure everything was working identically on the images that could already be correctly read.
6. Communication

The following sections provide an overview of the analysis, design, implementation, and testing of the communication iteration. It was decided by Epic that the communication must work with a RIS Emulator application provided by them. The functionality for this iteration consists of the bi-directional communication of viewing information and the sending of measurement information. This iteration was broken into two smaller iterations: the first iteration created the groundwork for the bi-directional communication of viewing information with the RIS Emulator, the second iteration added the measurement sending capabilities. This was done as the first iteration would be very self-contained; it added new communication classes into the design without changing many other classes. The second iteration would require more modifications to previously established classes and the addition of a measurement backend.

6.1 Viewing Information Communication Functional Requirements

The following functional requirements were gathered for the viewing information communication iteration. These functional requirements were gathered through a series of meetings between the project developer and Davin Sannes performing his role as the project’s sponsor.

<table>
<thead>
<tr>
<th>Requirement #</th>
<th>Requirement Name</th>
<th>Requirement Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comm.1</td>
<td>Send Open</td>
<td>To alert an outside RIS application the</td>
</tr>
<tr>
<td></td>
<td>Patient</td>
<td>viewer has opened a new patient.</td>
</tr>
<tr>
<td>---</td>
<td>-----------------</td>
<td>------------------------------------------------------</td>
</tr>
<tr>
<td>Comm.2</td>
<td>Send Switch Patient</td>
<td>To alert an outside RIS application the viewer has switch the patient it is viewing.</td>
</tr>
<tr>
<td>Comm.3</td>
<td>Send Closed Patient</td>
<td>To alert an outside RIS application the viewer has closed the patient it was viewing.</td>
</tr>
<tr>
<td>Comm.4</td>
<td>Send Open Study</td>
<td>To alert an outside RIS application the viewer has opened a new study.</td>
</tr>
<tr>
<td>Comm.5</td>
<td>Send Switch Study</td>
<td>To alert an outside RIS application the viewer has switched the study it is viewing.</td>
</tr>
<tr>
<td>Comm.6</td>
<td>Send Closed Study</td>
<td>To alert an outside RIS application the viewer has closed the study it was viewing.</td>
</tr>
<tr>
<td>Comm.7</td>
<td>Receive Open Patient</td>
<td>To receive a message from a RIS to open a patient in the viewer.</td>
</tr>
<tr>
<td>Comm.8</td>
<td>Receive Switch Patient</td>
<td>To receive a message from a RIS to switch to a patient in the viewer,</td>
</tr>
<tr>
<td>Comm.9</td>
<td>Receive Close Patient</td>
<td>To receive a message from a RIS to close a patient in the viewer.</td>
</tr>
<tr>
<td>Comm.10</td>
<td>Receive Open Study</td>
<td>To receive a message from a RIS to open a study in the viewer.</td>
</tr>
<tr>
<td>Comm.11</td>
<td>Receive Switch Study</td>
<td>To receive a message from a RIS to switch to a study in the viewer.</td>
</tr>
<tr>
<td>Comm.12</td>
<td>Receive Close Study</td>
<td>To receive a message from a RIS to close a study in the viewer.</td>
</tr>
<tr>
<td>Comm.13</td>
<td>Send Destroy</td>
<td>To alert an outside RIS application that the viewer is closing.</td>
</tr>
<tr>
<td>Comm.14</td>
<td>Receive Destroy</td>
<td>To receive a message from a RIS to close the viewer.</td>
</tr>
</tbody>
</table>

Table 4. Viewing Information Communication Requirements
In addition to the requirements, described in Table 4, there were several nonfunctional requirements that they must satisfy. The viewer had to be able to talk to the RIS Emulator locally, over a network, and must also be to run without any communication. This meant there had to be three modes of communication: local, remote, and none. The requirements Comm.13 and Comm.14 should only be invoked when either the viewer or the RIS Emulator are closed.

6.2 Viewing Information Communication Requirements

The following GUI Requirements were gathered during the viewing information communication iteration.

<table>
<thead>
<tr>
<th>GUI Control</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Patient Combo Box</td>
<td>Allows the user to open, close, or switch the current patient.</td>
</tr>
</tbody>
</table>

Figure 11. Main Screen with Viewing Information GUI Requirements
Study Combo Box | Allows the user to open, close, or switch the current patient.
---|---
Reconnect Button | Allows the user to reconnect to the RIS when the Viewer is not connected

Table 5. Viewing Information Main Screen GUI Requirements

As seen in Figure 11 and described in Table 5, the additions to the Main Screen for the viewing information GUI requirements were very minimal. When deciding the GUI changes the main effort was to work the necessary new functionality into existing GUI controls. For instance, the Study Combo Box was modified to have its behavior decide whether a study was close, open, or switched. The Patient Combo Box was added to the GUI as previously the viewer did not allow the selected patient to be changed. The selected patient was instead passed in to the viewer upon startup and never changed. The inclusion of the Patient Combo Box allowed the patients to be opened, closed, or switched by the user.

The Reconnect Button was also added to the Main Screen. This was done after a discussion with Epic about how they would like to alert the user of problems with communications. Epic said they would like the viewer to alert the user, allow them to continue working, and give them to option to try to reestablish a connection. To satisfy this request the Reconnect Button only appears after a connection has been dropped or a communication problem is unresolved. After the viewer finds a problem with the connection it displays a message box to the user that the RIS cannot be reached and the Reconnect button will appear. After the connection is reestablished the Reconnect Button becomes invisible.
Figure 12. Study Closed Reason Dialog

<table>
<thead>
<tr>
<th>GUI Control</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason Combo Box</td>
<td>Allows the user to select the reason a study was closed.</td>
</tr>
<tr>
<td>OK Button</td>
<td>Accepts the selected reason as the reason a study was closed.</td>
</tr>
</tbody>
</table>

Table 6. Study Closed Reason Dialog GUI Requirements

The Study Closed Reason Dialog, as seen in Figure 12 and described in Table 6, was added to the GUI because when a study is closed the requirement Comm.6 required that a reason be given as an input parameter. This dialog was added to allow the user to select their reason for closing a study.

6.3 Viewing Information Communication Prototyping

The prototyping for this iteration needed to accomplish two things: create a working model of communication with the RIS Emulator locally and model communication with the RIS Emulator over a network. The first prototype was to communicate with the RIS Emulator locally. The sole purpose of this prototype
was to demonstrate how to send the RIS Emulator communications and to prove the prototype can receive communications from the RIS Emulator. The RIS Emulator was a full application that was made with COM, not .NET, which provided a set of challenges, as this meant it could not be directly referenced as easily as if it could if it were made in .NET.

To add in communication with the RIS Epic provided the basic materials they use for integration with commercial PACSs. These materials included a document describing how the RIS Emulator communicates and two .NET compatible interfaces, IEpicPACS and IEpicRIS. IEpicRIS was an interface for the RIS Emulator while, IEpicPACS is an interface the RIS Emulator expects from the PACS, or in this case the prototype. After researching how to obtain handles to outside programs and communications with Epic, a solution was found that would allow the prototype to obtain a handle to the RIS Emulator, using this solution it was possible to send communications to the RIS Emulator. To receive communications from the RIS Emulator a few steps needed to be taken. The RIS Emulator would need a handle to the prototype and from this handle the emulator would be able to directly send messages to the prototype. This allowed communications to be received but also brought to light another problem. When the prototype would receive a message it could not update its GUI as it would cause a threading error. A solution was found using WPF’s dispatcher to add an event onto the GUI’s event queue. This solved the threading problem.

The next problem to solve in the prototype type was how to communicate with the RIS Emulator remote over a network. Because the RIS Emulator could not be modified it was decided that a possible solution could be a server that wrapped around the RIS Emulator. The server would act as a remote bridging application to handle communication between the prototype and the RIS Emulator.
To solve the problem of remote bi-directional communication .NET provides a variety of technologies that can be used to solve it. After researching some of the different technologies and methods the following were considered to be potential solutions: remoting, sockets, WCF, and peer to peer. In deciding which technology would best suited the following things needed to be defined: what kind of network will be used, is the server location known by the client, and is the connection multiplicity one to one, one to many, or many to many.

For this connection the server location could be known by the client when the client is started as it could be passed as a URL parameter. The network for this communication would be a LAN, and the connection multiplicity would be one to one. For these criteria it was deciding that remoting would be best. Remoting was chosen because it is far simpler to use than sockets, WCF is very complex and aimed at web services, and Peer to Peer would be needlessly complex as the server can be known and the connection multiplicity is only one to one.

In researching a solution that leveraged .NET remoting there were several choices that had to be made such as the protocol to use and how the server would communication back to the client. It was decided to use a TCP channel for the communication as reliability of the communications was more important than speed. From this a working solution for talking from the client to the server was creating. To talk from the server back to the client there were two main methods that could work: server events and passing a handle to the server. A prototype was made for each method of server to client communication and both prototypes worked correctly allowing the server to communicate back to the client. However, it was decided that passing a handle to the server would be the best solution. The decision was made because events were more complex to implement and more importantly events cannot have a return type. This would be a large limitation for any future additions to the communication functionality.
Last in the prototyping phase, the remote prototype needed to prove that the server could act as a bridge to the RIS Emulator. To do this a client would connect to the server, which was running on a computer with the RIS Emulator, and send a communication such as patient switched. The server would then pass the communication on to the RIS Emulator. To do this the code used in the previous prototype was implemented in the Server. This created a working solution for communication remotely to the RIS Emulator.

6.4 Viewing Information Communication Design

![Diagram of Viewing Information Communication Class Diagram](image)

Figure 13. Viewing Information Communication Class Diagram
While designing the communication iteration the most important ideas were that while the viewer needed to have three communicate modes: none, local, and remote; and there needed to be a common interface for all these modes. As can be seen in Figure 13, to do this an abstract class called Communicator was created with three concrete child classes: NonCommunicator, LocalEpicCommunicator, and TcpRemoteCommunicator. NonCommunicator would be an empty classed used for when a communication mode of none is chosen. The NonCommunicator class would allow all the communication logic, or lack thereof in this case, to be contained within the communication classes so no outside classes would have to have any logic or knowledge of communications. LocalEpicCommunicator would be used when the communication mode is local and as such would communicate to a local RIS Emulator. The TcpRemoteCommunicator would be responsible to communicating over a network to a remote server acting a bridge to the RIS Emulator.

It was decided during the design that all communication would happen asynchronously. This decision was made because during the prototyping phase when the remote prototype’s GUI would freeze during communication problems. The prototype’s GUI would freeze because it was performing communication synchronously. This would not provide a good user experience for the project. To perform communication asynchronously the implementation of communication from the client to the server would make use of WPF’s BackgroundWorker class. The BackgroundWorker class provides a way to easily have methods performed asynchronously.

To tell the Viewer which mode of communication to use a new URL parameter would be added to the Viewer called “communication”, the value for this URL parameter would tell the Viewer what kind of communication to use: “LocalEpic” if it is local, the TCP connection string if it is remote, and all other values are considered to be none. To create the actual Communicator object a
variation of the Factory pattern that is common in .NET was used. The abstract class Communicator has a static method with a return type of Communicator. In this method it checks what the passed communication URL parameter is and decides which child class to create. Since this is just a simple comparison operation the logic is contained in the method, if however, in future versions of the project it becomes more complex an actual factory class should be created and used here.

It was decided the Communicator class would use events to communicate back anything it needs to tell the viewer, such as incoming messages. This was done to keep the communication classes completely separate from the rest of the project. Using events allows the Communicator classes to alert other classes in the viewer without having to know anything about them. The event system is used instead of exceptions for the cases when a connection is lost or gained. This is because exceptions are very costly operations and this solution provided an easy and more efficient way of handling connection problems, which in certain scenarios, can be very common.

6.5 Viewing Information Communication Implementation and Testing

The implementation of the viewing information communication and implementation went very smooth due to extensive prototyping and designing. To test the communication it was important to test a variety of scenarios, such as over a network when the network goes down or if the server computer abruptly crashes. To test the communication functionality two computers were connected with a wireless router. During these tests one computer acted as the server and the other as the client. To test the functionality the tester would manually perform functions on both the client and the server that would invoke a communication between them, such as switching the patient. The other computer would then be
checked to make sure it properly received and acted on the message. During the
testing the server, client, and wireless network were closed improperly for several
of the tests to make sure both the server and client could properly handle these
network problems.

During the course of testing one scenario was discovered that was not
covered in the prototyping and design. This scenario was when the server was
itself closed and then brought back up before the client is alerted of the server
going down. In this circumstance the client would still be able to communicate
with the server without a problem, but the server would have lost its handle to the
client and be unable to communicate back to the client. To solve this problem a
method was added to the client interface called ServerClosedEarly. When the
server would be closed it would check to see it is a normal close caused by the
RIS Emulator or the Viewer, if not it would call this method to tell the Viewer it
is going down.

6.6 Measurement Communication Functional Requirements

The following functional requirements were gathered for the measurement
communication iteration. These requirements were gathered over a series of
meetings between the developer and Davin Sannes acting as the project’s sponsor.

<table>
<thead>
<tr>
<th>Requirement #</th>
<th>Requirement Name</th>
<th>Requirement Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comm.15</td>
<td>Send Measurement</td>
<td>To alert an outside RIS that a measurement has been taken.</td>
</tr>
<tr>
<td>Comm.16</td>
<td>Delete Measurement</td>
<td>To alert an outside RIS that a measurement has been deleted.</td>
</tr>
</tbody>
</table>

Table 7. Measurement Communication Requirements
The scope of requirements, described in Table 7, is only for the viewer’s currently available measurement which is a text measurement. It was found that the RIS Emulator does have functionality for communicating any measurement information. It was decided by Epic to have the measurements saved in an XML data file which must be saved on the computer running the RIS Emulator. Epic left it up to the developer to decide the format of the XML. The information that needs to be sent with the measurements is the Patient, Study, and DICOM file the measurement corresponds to.

6.7 Measurement Communication GUI Requirements

![Image](image_url)

Figure 14. Main Screen with Add Text Measurement

<table>
<thead>
<tr>
<th>GUI Control</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Add Text Context Menu Item | Allows the user to add a new text measurement to a DICOM image.

Table 8. Add Text Measurement GUI Requirements

The Add Text Content Menu Item, as seen in Figure 14 and described in Table 8, will invoke the functional requirement Comm.15.

Figure 15. Main Screen with Delete Text Measurement

<table>
<thead>
<tr>
<th>GUI Control</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edit Text Context Menu Item</td>
<td>Allows the user to edit a text measurement in a DICOM image.</td>
</tr>
<tr>
<td>Delete Text Context Menu Item</td>
<td>Allows the user to delete a text measurement in a DICOM image.</td>
</tr>
</tbody>
</table>

Table 9. Delete Text Measurement GUI Requirements
The Delete Text Content Menu Item, as seen in Figure 15 and described in Table 9, will invoke functional requirement Comm.16. The Edit Text Context Menu item will actually delete the text measurement and then create a new one in its place, invoking Comm.16 and Comm.15.

6.8 Measurement Communication Prototyping

No prototyping phase was necessary for this iteration as the requirements for this iteration was well understood.

6.9 Measurement Communication Design
Figure 16-1. Top of Final Class Diagram
Figure 16-2. Middle of Final Class Diagram
The design process of this iteration would finish the class diagram for the project, which can be seen in Figures 16-1, 16-2, and 16-3. The design of this iteration required small modifications to a large portion of the design. The largest change was that the original project did not have any class for the measurements; it was simply a proof of concept as to how measurements could work. This required a measurement backend to be designed first. To do this it was decided to make an abstract Measurement class that would have the information that is common to all measurements such as: the date and time the measurement was taken and the collection of image frames the measurements corresponds to.
To handle the creation and management of measurements a Measurement Controller was be created. This class would be in charge of creating, deleting, and handling the measurements of a DICOM file. This class helped to decouple the measurement classes and the various GUI classes. This class uses events to communicate any changes to the measurements to outside classes, allowing the MeasurementController class to remain decoupled from other classes in the design.

The general format for the XML to be used for measurement storing can be seen in Figure 17.

```xml
<Action type="""" dateTime="""" frames="""" image="""" study="""" patient="""" />
```

Figure 17: General Measurement XML Format

In Figure 17 Action is either “Added” or “Deleted”, dateTime is the date and time the measurement was taken, frames is the listing of common separated frames, or “All” if it corresponds to all frames, image is the DICOM file the measurement was taken from, study is the study the measurement was taken from, and patient is the patient the measurement was taken from.

Beyond this simple format each type of measurement would have different attributes and values depending on what information they store. The XML element used to represent text measurements can be seen in Figure 18.

```xml
<Action type="TextMeasurement" frames="""
       image="""" study="""" patient="""" location="""">
   TEXT_VALUE
</Action>
```

Figure 18: Text Measurement XML Format

In Figure 18 the attribute location is the exact X, Y location of the text measurement and TEXT_VALUE is the text value of the text measurement.
To convert measurements to XML a MeasurementToXMLConverter class was created. This class has two public methods called GetXMLFromMeasurementAdded and GetXMLFromMeasurementDeleted. From these two methods the MeasurementToXMLConverter class converts the measurements to the appropriate XML. This class will handle all the logic of how to convert measurements to XML consolidating the conversion logic to one class.

Finally the Communicator class and its child classes will need to be modified to support sending measurement information, send measurement added and send measurement deleted methods were added. How to save the XML data depended on the communication class. The NonCommunicator like its other methods did nothing. The LocalCommunicator uses the MeasurementToXMLConverter to convert the measurement into an xml string and then saves it to a file in the user’s document directory. The TcpRemoteCommunicator has to convert the measurement before sending to the server as only serializable data can be sent through .NET remoting. The RISServer class then had to modified able accept this xml string and save the string into a file in the user’s document directory.

**6.10 Measurement Communication Implementation and Testing**

During the implementation one problem arose that the original design did not address. In the current project there is no way to get the file name of the DICOM image. This was fixed by adding a property for the file path in the DICOM class. The DICOM class was already taking in a file path in the constructor, all that needed to be done is have the file path be stored and available as a project.

Testing this iteration was done by creating, editing, and deleting different text measurements in different patient, studies, and images in all three
communication modes. From these test the generated xml file was inspected by the tester to verify that the expected XML was generated for each action.
7. Continuing Work

There are many ways to improve upon the viewer in future versions. This is due to the fact that medical image viewers are extremely large and complex applications with an extremely large set of possible functionalities. Some of the areas which work can be furthered are increasing the support of the DICOM standard, expanding the communication system, adding more image filters, and expanding the measurement system.

Currently the DICOM parser only reads a minimal set of tags from DICOM files. This can be expanded to read more of the tags such as patient and study information which could be used to improve the viewer. For example, patient and study information could be displayed as an overlay on the displayed DICOM images. The parser can also be expanded to support even more compression types, which the design in this project was specifically aimed at making as easy as possible.

The current communications functionality could be expanded to send and receive messages such as zooming, panning, receiving measurement data, or even which images in a study should be minimized and maximized. The addition of more communications will allow the viewer to better integrate with current RIS’s in industry. The communication can also be expanded to support more types of communication such as to another program instead of the RIS Emulator; once again this is something the current design specifically aimed at making it easy to do.
The image filter system can be expanded to provide a wider range of available filters to help the user to analyze and understand the DICOM files they are viewing. Some of filters these new filters could be color filters, even smart filters with image recognitions that could highlight possible problems such as cancer tissue. This is one area that unfortunately will be a bit harder to work with, at least until WPF is updated to support multiple WPF Effects on one control.

One shortcoming currently in the viewer is the measurement system. In the project the viewer only supports text measurements. This is an area with great potential for expansion in the future. The addition of length, angle, area, speed measurements and could add great functionality to help users read and analyze DICOM files.
8. Conclusion

This document describes the software development of a medical image viewer for Epic Systems. This project added significant functionality to the previous version of the project and will serve Epic well as an internal testing and demonstration tool. Additionally, the knowledge and experience gained from this project will help Epic better serve their clients who are PACS.

While all the requirements and goals for this project were met a medical image viewer needs to be much larger and include more functionality than the current viewer. Much work will need to be done to this project to be considered usable in real world scenarios. This is mostly due to functionality that was outside the scope of this project, most notably measurements.

Epic will be able to use this software internally immediately as all versions of the project are stored on their servers with instructions detailing how to use the software. It is also worth noting that even if Epic does not continue the application as a single project they can still develop its parts individually. The parts of the project such as the DICOM parser, image filters, and communications can be used in other applications easily as the project was developed as several different DLL’s to allow the individual functionality to be easily imported into other projects.
9. Bibliography


