Appendices M

Document Management

1.1.1. Uploading a document to the Server

```
2665    private byte[] uploadDocument(String fileName) throws SQLException, IOException {
2666        File file = new File(fileName);
2667        byte documentData[] = new byte[(int) file.length()];
2668        try (BufferedInputStream input = new BufferedInputStream(new FileInputStream(file.getName()))) {
2669            input.read(documentData, 0, documentData.length);
2670        }
2671        return documentData;
2672    }
```

Fig x – Extract from ClientImpl – uploadDocument()

As you can see from Fig x I implemented a method that constructs an array of bytes from a given file. To do this I used a BufferedInputStream, and supply the constructor of the BufferedInputStream with a FileInputStream for the given file that is to be uploaded. I then invoke the read() method on the BufferedInputStream, supplying the empty array as the destination to read to, and zero and the document length as the boundaries for the read. I then return the array of bytes back to the invoking method.

The byte of arrays is then able to be passed to the client as a parameter along with the document name and some other information associated with the document, for the Server to then store the file.

```
518    public int createPersonDocument(int pRef, String fileName, String comment) throws RemoteException, SQLException {
519        if (server.isAlive()) {
520            byte[] documentData = null;
521            try {
522                documentData = this.uploadDocument(fileName);
523            } catch (IOException ex) {
524                Logger.getLogger(ClientImpl.class.getName()).log(Level.SEVERE, null, ex);
525            }
526            if (documentData != null && documentData.length >= 1) {
527                return server.createPersonDocument(pRef, fileName, documentData, comment, user.getUsername());
528            }
529        }
530        return 0;
531    }
```

Fig x – Extract from ClientImpl class – createPersonDocument()

As you can see from Fig x, once the array of bytes has been created, I then check to see if the array of bytes (document data) has elements within the array, if so I then invoke a method on the Server which creates the file on the Server with the associated information.

```
786     public int createPersonDocument(int pRef, String fileName, byte[] buffer, String comment, String createdBy)
787         if (this.database.personExists(pRef) && !this.database.getPerson(pRef).hasDocument(fileName)) {
788             fileName = documentsLocation + "Person" + pRef + " - " + fileName + "v1";
789             DocumentImpl document = this.uploadDocument(fileName, buffer, comment, createdBy);
790             Person person = (Person) this.database.getPerson(pRef);
791             person.createDocument(document, new ModifiedBy("Created Person Document", new Date(), createdBy));
792             this.database.updatePerson(pRef);
793             this.database.createPersonDoc(pRef, document);
794             return 1;
795         }
796         return 0;
    }
```

Fig x – Extract from ServerImpl class – createPersonDocument()

As you can see from Fig x, I firstly check to ensure the information supplied is valid, if so I then amend the filename to include the location the document will be stored at, along with some uniquely identifying information for the object the document belongs to, such as Person information, and then finally add the version number. I then invoke a uploadDocument() method which reconstructs the document as shown below and then return a Document object, which holds the filename and some other document information. I then update the database, and save the document file path to the database. The server now has a local copy of the file, which can then be accessed by any client of 'MSc Properties' system, by downloading a version of the document.

```
3876    private DocumentImpl uploadDocument(String fileName, byte[] buffer, String comment, String createdBy) {
3877        try {
3878            File file = new File(fileName);
3879            try (BufferedOutputStream output = new BufferedOutputStream(new FileOutputStream(fileName))) {
3880                output.write(buffer, 0, buffer.length);
3881                output.flush();
3882                Note note = this.createNote(comment, createdBy);
3883                DocumentImpl document = new DocumentImpl(documentRef++, file, note, createdBy, new Date());
3884                return document;
3885            }
3886        } catch (Exception e) {
3887            System.out.println("ServerImpl: " + e.getMessage());
3888        }
3889        return null;
3890    }
```

Fig x – Extract from ServerImpl class – uploadDocument()

As you can see from Fig x, I do the reverse to when I am converting the file into an array of bytes, and use a BufferedOutputStream, with a FileOutputStream for the file, as a parameter for the constructor of the BufferedOutputStream. I then write from the array of bytes called buffer, supplied as a parameter to the uploadDocument method, using the BufferedOutputStream.write method with the array of bytes as the source for the data, and 0 and the length of the array, as boundaries for the write method. This write method then writes the data to the file specified by the file name. I can then create a Document object, which will store the file name, along with the file path and some other bits of information associated with the document.

1.1.2. Downloading a document from the Server

```
540     public int downloadPersonDocument(int pRef, int dRef) throws RemoteException, SQLException {
541         if (server.isAlive()) {
542             try {
543                 this.openDocument(server.downloadPersonDocument(pRef, dRef, user.getUsername()));
544                 return 1;
545             } catch (IOException ex) {
546                 Logger.getLogger(ClientImpl.class.getName()).log(Level.SEVERE, null, ex);
547             }
548         }
549         return 0;
550     }
```

Fig x – Extract from ClientImpl class – downloadPersonDocument()

As you can see from Fig x, when a client invokes the downloadPersonDocument method it invokes the method openDocument() and supply it with the return value from the method Server.downloadPersonDocument() which would be the array of bytes for the document to be downloaded.

```
824     @Override
825     public byte[] downloadPersonDocument(int pRef, int dRef, String downloadedBy) throws RemoteException {
826         if (this.database.personExists(pRef) && this.database.getPerson(pRef).hasDocument(dRef)) {
827             return this.downloadDocument(dRef);
828         }
829         return null;
830     }
```

Fig x – Extract from ServerImpl class – downloadPersonDocument()

```
3859    private byte[] downloadDocument(int dRef) {
3860        if (this.database.documentExists(dRef)) {
3861            try {
3862                Document document = this.database.getDocument(dRef);
3863                File file = new File(document.getFilePath());
3864                byte buffer[] = new byte[(int) file.length()];
3865                try (BufferedInputStream input = new BufferedInputStream(new FileInputStream(document.getDocumentPath()))) {
3866                    input.read(buffer, 0, buffer.length);
3867                }
3868                return (buffer);
3869            } catch (Exception e) {
3870                System.out.println("FileImpl: " + e.getMessage());
3871            }
3872        }
3873        return (null);
3874    }
```

Fig x – Extract from ServerImpl class downloadDocument()

As you can see from Fig x and Fig x, The downloadPersonDocument() method just checks to see if the parameters supplied are valid, and then invokes a local method downloadDocument(), and the downloadDocument() method converts the file stored locally to the server, into an array of bytes in a similar fashion to how the Client does when the client is uploading a file to the server, using BufferedInputStream and FileInputStreams, and read the data from the file into the array of bytes. The array of bytes is then returned to the invoking method.

```
2674    private File openDocument(byte[] documentData) throws SQLException, IOException {
2675        File file = File.createTempFile("msctmp", ".pdf", new File("D:/"));
2676        try (BufferedOutputStream output = new BufferedOutputStream(new FileOutputStream(file.getName()))) {
2677            output.write(documentData, 0, documentData.length);
2678            output.flush();
2679        }
2680        Desktop desktop = Desktop.getDesktop();
2681        desktop.open(file);
2682
2683        return file;
2684    }
```

Fig x – Extract from ClientImpl class – openDocument()

Now the server has supplied the openDocument() method with the array of bytes for the file, as you can see from Fig x, I then have to reconstruct the file, in the same way the Server had to reconstruct the file when a client uploads a document to the file. To do this I again use a BufferedOutputStream, with a FileOutputStream, and then invoke the BufferedOutputStream.write method to write the data to a file.

As the client does not need to save a copy locally to them, I decided to use the File.createTempFile() method and supply it with a temporary name, this then allows for the file to be deleted once the client has finished with the file, however it also allows for the client to manually save a copy of the file if they do require.

Once the data from the array of bytes has been written to the temporary file specified by the file name, I then create a Desktop object, which is initialised with the Desktop.getDesktop() method, which returns the clients desktop. I then invoke the Desktop.open() method on the desktop object, and supply the temporary file I have just created. This then opens the temporary file, with the default application for the type of file that is being opened. This leaves the client actually viewing the file they selected to download.

1.1.3. Version Control

To implement version control functionality, I decided to create a list of File objects within the document class, which enables me to add an updated file to the document object, and with each updated file that is added to the document object, the filename version increments by 1, to keep a track of which version the document is currently at.

```
798     @Override
        public int updatePersonDocument(int pRef, int dRef, byte[] buffer, String modifiedBy) throws RemoteException {
800         if (this.database.personExists(pRef) && this.database.documentExists(dRef) && this.database.getPerson(pRef).hasDocument(dRef)) {
801             this.uploadNewVersionDocument(dRef, buffer, modifiedBy);
802             return 1;
803         }
804         return 0;
805     }
```

Fig x – Extract from ServerImpl class, updatePersonDocument()

```
3892    private DocumentImpl uploadNewVersionDocument(int docRef, byte[] buffer, String modifiedBy) {
3893        if (this.database.documentExists(docRef)) {
3894            try {
3895                DocumentImpl document = (DocumentImpl) this.database.getDocument(docRef);
3896                String fileName = document.getDocumentPath() + "\\" + Utils.getFileNameWithoutVersion(document.getDocumentName()) +
3897                        (document.getPreviousVersions().size() + 2);
3898                File file = new File(fileName);
3899                try (BufferedOutputStream output = new BufferedOutputStream(new FileOutputStream(fileName))) {
3900                    output.write(buffer, 0, buffer.length);
3901                    output.flush();
3902                    document.createNewVersion(file, new ModifiedBy("Updated Document", new Date(), modifiedBy));
3903                    return document;
3904                }
3905            } catch (Exception e) {
3906                System.out.println("ServerImpl: " + e.getMessage());
3907            }
3908        }
3909        return null;
3910    }
```

Fig x – Extract from ServerImpl

As you can see from Fig x and Fig x, when the client wants to update a document (create a new version of a file), the client invokes updateXDocument(), with X being a specific object such as Person, and again the client just supplies an array of bytes for the new version of the file, this is then constructed back into the file in the same way as previously shown, but instead of using the same file name as was previously used when the document was first created I increment the version number by 1.

This is done by invoking a static method from my own Utils class which extracts the filename without the version number or the extension and uses the list of files size from the document object to get the new version number and then reconstruct the filename with the new version number.

A client of ‘MSc Properties’ is then able to view the most current version of the document, but can also see a list of previous version by invoking the Document.getPreviousVerions() method.