Document Management System

April 25, 201?

CS4125: System Analysis

Authors

Name: XXX	ID:1234567
Name: YYY	ID:1234567
Name: ZZZ	ID:1234567

Table of Content

- Communication diagram
- System architecture diagram with interface
- ER diagram
- State chart

Evaluation

- gap between initial sketches and implementation
- Problems with designing security/concurrency using UML
- Our view of the UML

1 Description

For this project we were assigned the task of designing and implementation of a document management system to be used in a college department. This system will allow users of different types login and use the services provided. The types of users are lectures and students. Students will be able to search the papers, upload papers, edit papers delete papers and comment on papers. A lecturer is able to do all of the above plus accept papers and review papers.

Our system will be web based making it easy for both lecturers and students to access the papers. It will let users who are registered with the colleges LDAP system and meet the right criteria use the site. Or goal is to make it as user friendly as possible with an easy to use front end.

2. Software life cycle

A software development life cycle is a methodology use in the creation of new software. Without a methodology for making software it is very easy for bugs and errors to creep into the code also it can cause problems with software being delivered late or worse still the software that is delivered is not what the customer wants. Also without a plan it can become very expensive to fix bugs later on in the life-cycle. Some of the stages of a typical life-cycle are shown below.

- Analysis
- Design
- implementation
- Verification
- Maintenance

There are many different methodology's each with its own emphasis on some of the stages shown above. The two we considered where the waterfall model and RUP.

The waterfall model

The waterfall model is one of the oldest and best know life-cycles. It has a strong emphasis on up front design. Its is called the waterfall model because each stage of the life-cycle needs to be completed before you can move onto the next stage. Revision of stages that have already been completed is frowned upon in this model. As such is often criticized for being too ridged. Its supporters claim that time spent in the analysis and design stages will produce better code and will save time on error testing and debugging later on in the life-cycle.

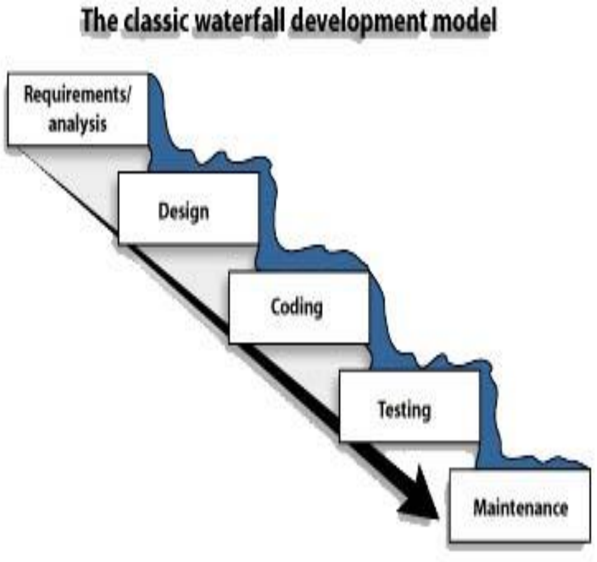


Figure 1.

In the analysis phase we have to look at what is required what the customer is looking for. This involves meeting with the customer discussions on what they want.

In the design stage we have to create use case diagrams. And consider how the system should be built. And discussions how the user will interact with the systems and talk about the non-functional requirements.

In the coding stage we implement the design from the last stage.

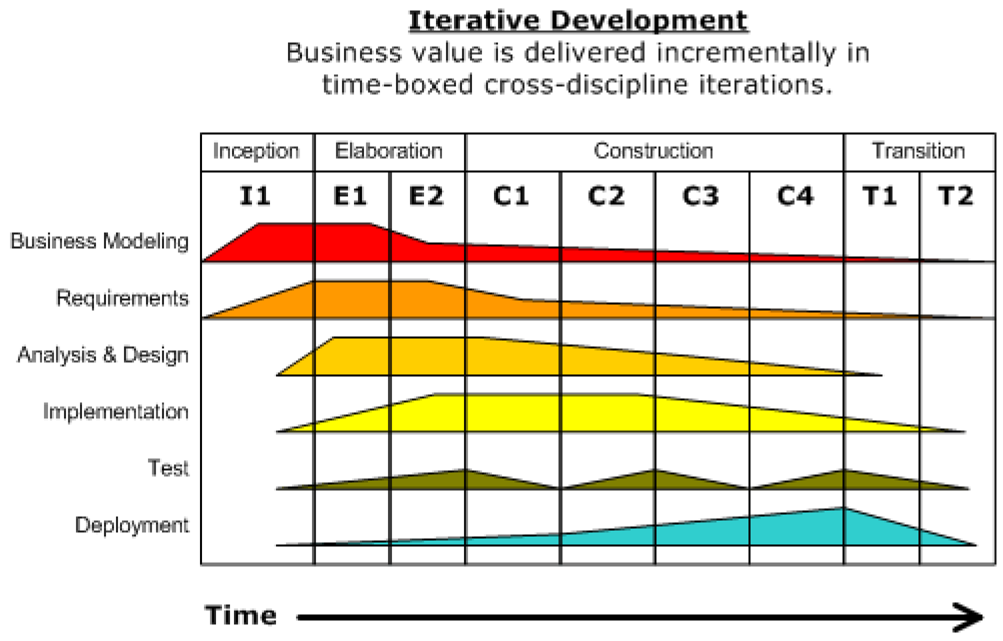
Then the system has to be tested.

And finally the system is ready for release and maintenance.

If the system was well designed and documented then this should save considerable amount of time and money in the maintenance. Maintenance can be one of the most expensive parts of software.

The RUP model

The Rational Unified Process (RUP) is an iterative software development framework created by the Rational Software Corporation, a division of IBM since 2003[wikipedia]. The RUP model does share things in common with the waterfall model how there is an overlap between the work flow in the different stages. So most of the analysis and design is done in the first few stages but it does not stop once development has begun. This means that the customer can be keep in house while the project is under construction and give feed back on how they feel about it and might suggest new features be added to the project after implementation has begun.



The RUP is broken down into four phases 1 inception, 2 elaboration, 3 construction and 4 transition .

Inception.

Establish the requirements in discussions with the stake holders. Make estimates of schedule and costs and lay out expected results.

Elaboration.

A lot of design work making use cases. Make sure that all the actors have been correctly identified. Have a prototype of the basic features.

Construction.

This is the phase where most of the design has been finished and the main focus is on coding. With some work being done on the testing of the system.

Transition.

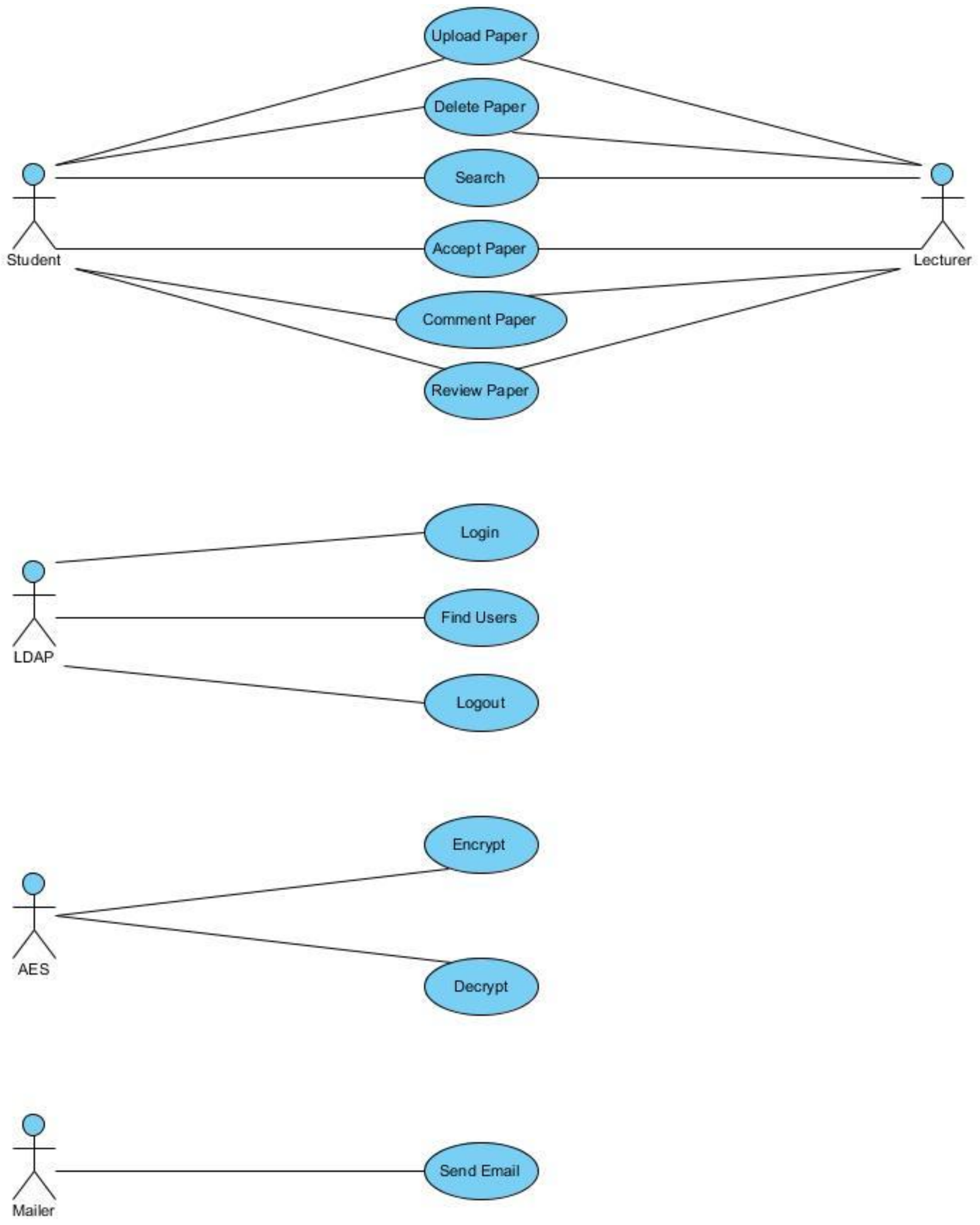
This is the final phase where the test is completed. Training and manuals produced for the end user and finally a release. Once the product has been released that is the end of the life-cycle.

Project Plan

Description	Person	Delivered
Narrative	XXX	week 8
Software life-cycle	XXX	week 8
Project Plan	All 3	week 7
Requirements		
Use case diagrams	XXX	week 8
Descriptions	ZZZ	week 8
Non-functional requirements	YYY	week 8
Screen shots	YYY	week 12
Initial sketches		
Class diagrams	All 3	
Communication Diagram		
System architecture diagram	ZZZ	week 11
Research		
Description	YYY	week 12
Implementation		
Object-Orientated	All 3	week 8 – 12

Appropriate use of patterns	All 3	week 10
Security/ Concurrency/Session Mgt.	YYY	week 9
Evidence of test	All 3	week 10 – 12
Abstraction of the Implementation		
Class Diagram	All 3	week 11 - 12
Communication Diagram	All 3	week 11 - 12
System architecture	All 3	week 11 - 12
ER-Diagram	All 3	week 11 - 12
State Diagram		week 9
Evaluation		week 12
References		

Requirements



Use case diagram

Non functional requirements

With non functional requirements we looked at things such as:

- Cryptography
- UI
- Portknocking

When saving user information, all information should be consider sensitive even if it’s simply a username and a date of birth, you do not want this information leaking out. One way to stop this from happening is to encrypt any information you save. This way if someone does get the information, it will make no sense to them.

An encryption system we looked into was AES. This is a trusted and tested encryption/decryption system. When we wished to save something to a database or pass it to the LDAP system, we would first encrypt it, save the information and then to read it we would encrypt it using the same key that was use you encrypt it under the same encryption system.

For the UI we kept a minimalistic approach to the layout. For reasons such as simply is direct and easy to navigate and there was no need to complicate things. In this project we focused on using tables in the pages. We had one CSS file that would then define the interface layout for the tables and then went with the same system across all pages. For displaying the document we just left the text file in raw txt format. There was no need or time to style them and since we dealt with plain text and not rich text, it would add problems for very little details.

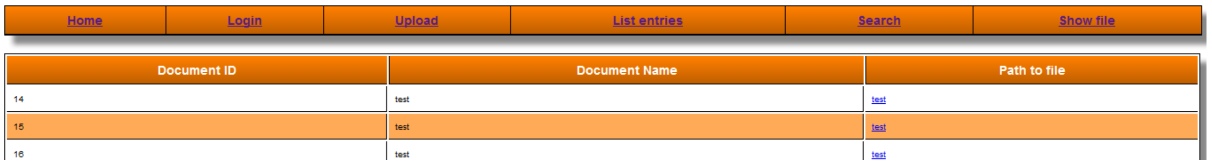
Port knocking was used as a secure was of passing files from our local server to our backup server. This system works by sending a sequence of messages to a server and responding to what we get back. This would only allow people who knew who to respond properly to gain access to the server. Preventing hackers from getting access and also safely backing up our files.

structured use case description

USE CASE 1	Upload	
Goal in Context **Scope & Level** **Preconditions** **Success End Conditions** **Failed End Conditions** **Primary, Secondary, Actors** **Trigger**	Allow user to upload a file, save a file to a server to display later Side wide User logged in, file already created File uploaded File not uploaded User, Upload Function, Server Upload Function	
DESCRIPTION	**Step**	**Action**
	1	User logges in and has file local
	2	User navigates to upload page, inserts file name, description and summary
	3	User attaches file and uploads
	4	File is written to local server disc and backeu up to cloud
EXTENSIONS	**Step** 4a	**Branching Action** File is not uploaded and user is notyfied
VARIATIONS	4	File is already there and user is notyfied of updated version
Related Information	1 Upload	
Priority	Top	
Performance	1-10 seconds	
Frequency	200/day	
Channel to actors	Upload function	
OPEN ISSUES	When taking file from upload page, summary and title parameters are lost.	
Due Date	Unresolved	
…any other management information…		
Superordinates	Filewriter	
Subordinates	Backup to cloud PortKnocker	

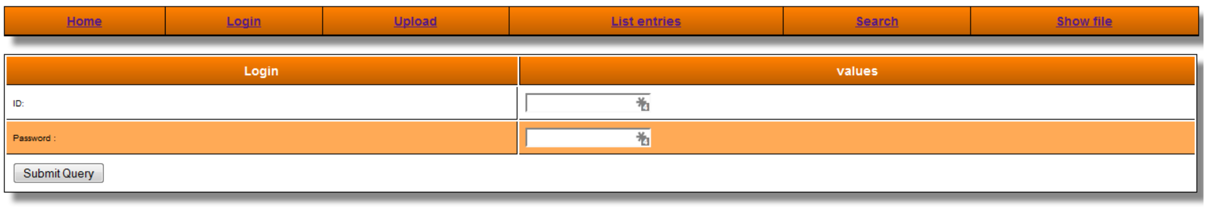
Screen shots

listing documents



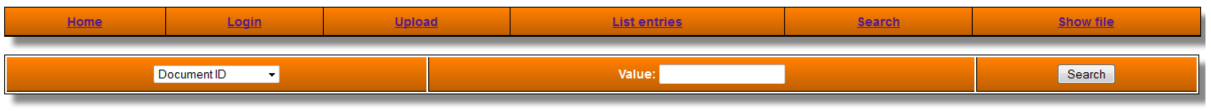
Document ID	Document Name	Path to file
14	test	test
15	test	test
16	test	test

Login

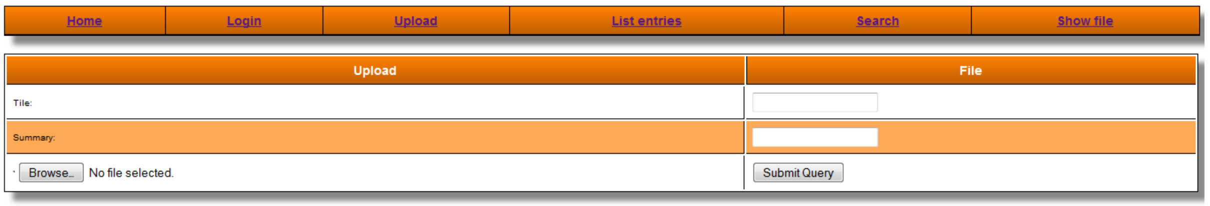


Login	values
ID:	
Password :	

Search



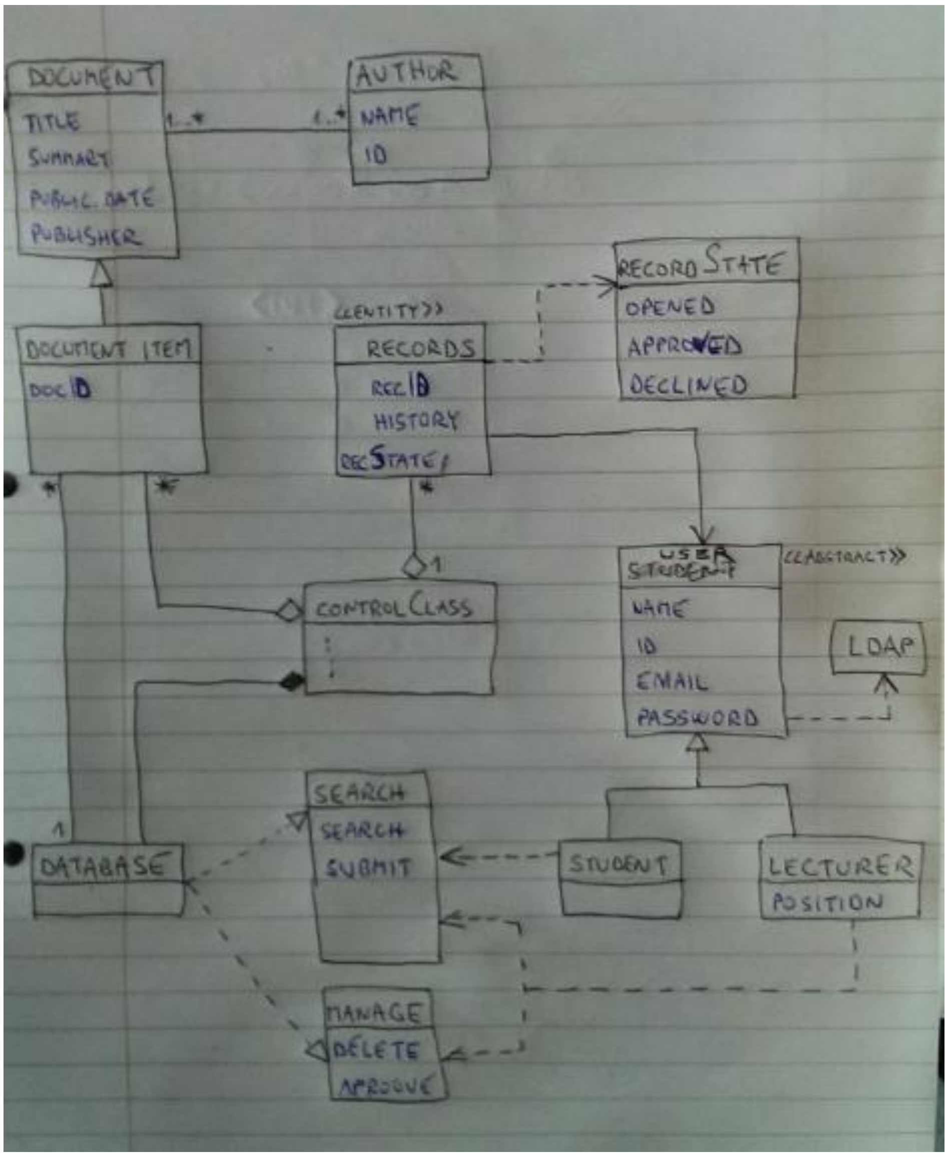
Upload



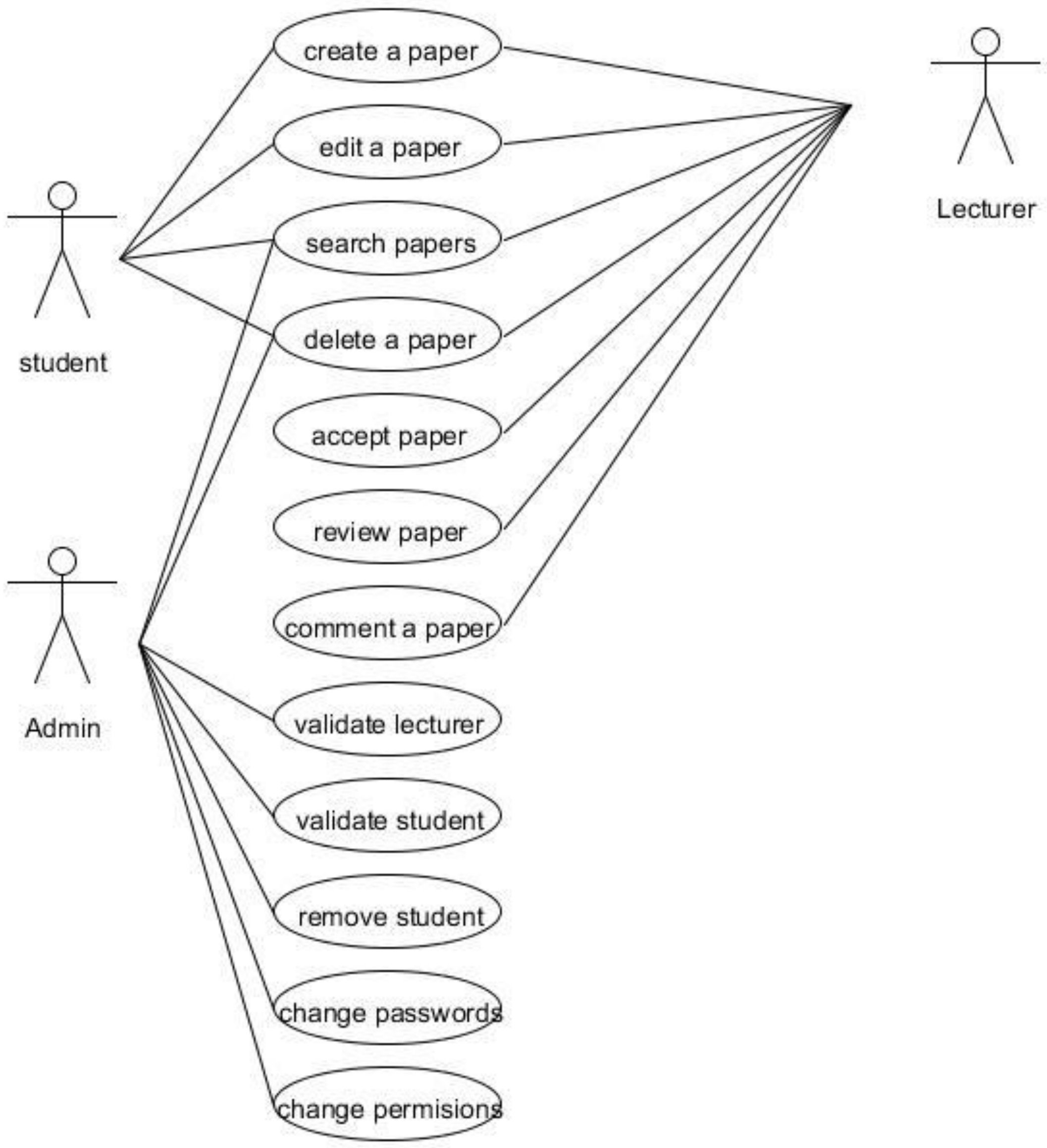
Upload	File
Tile:	
Summary:	

Initial sketches

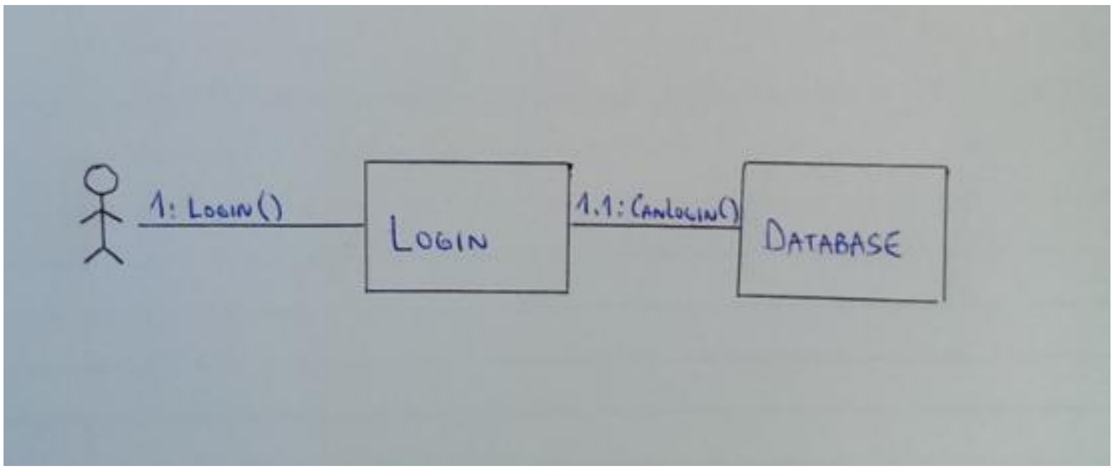
Class diagram



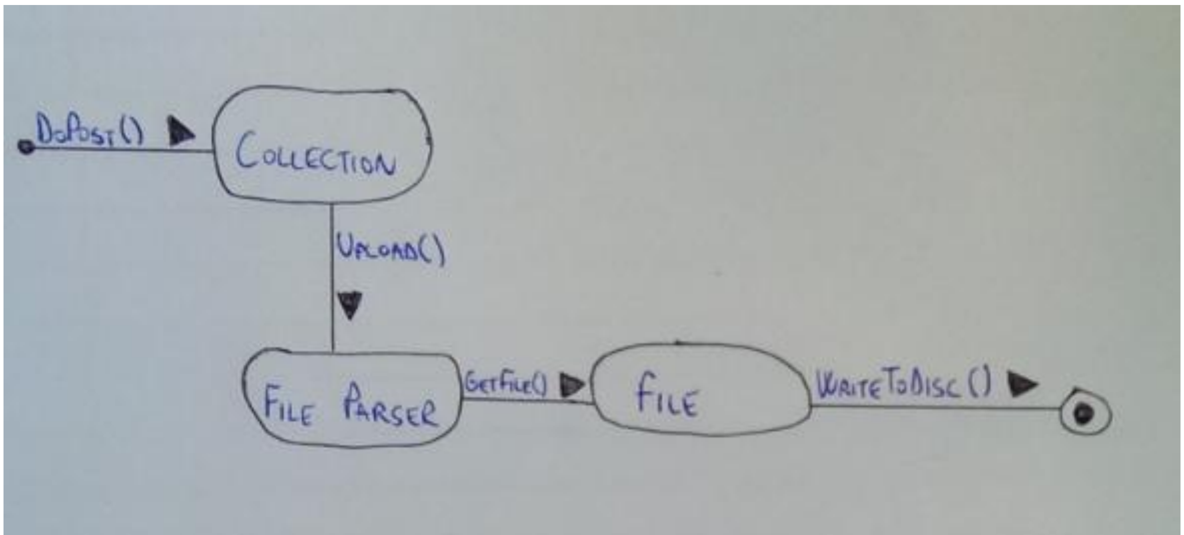
Use case initial diagram



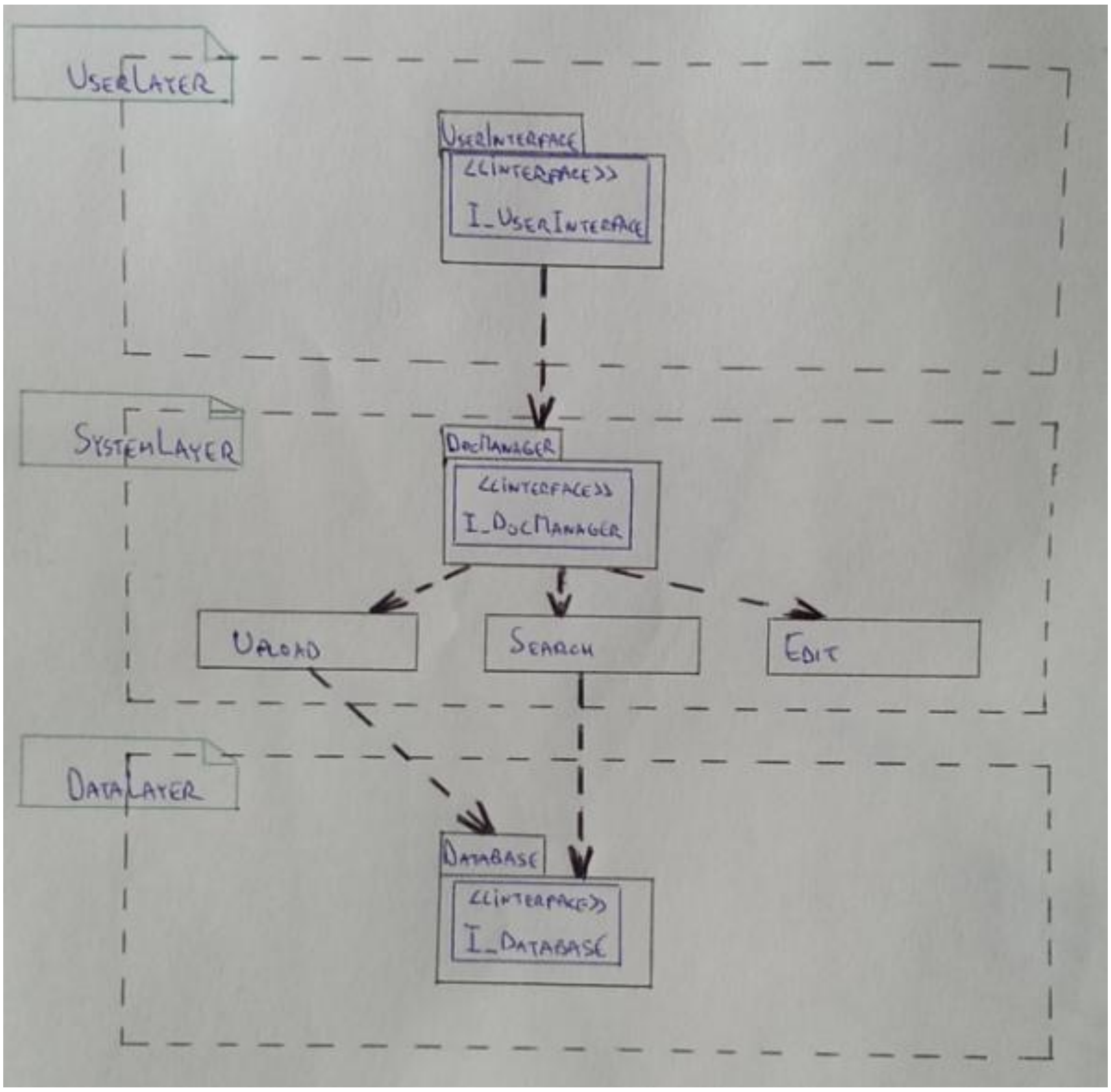
Communication diagram



Initial State chart



System architecture diagram with interfaces


USERLAYER
USERINTERFACE
<<INTERFACE>>
I_USERINTERFACE
SYSTEMLAYER
DOCMANAGER
<<INTERFACE>>
I_DOCMANAGER
UPLOAD
SEARCH
EDIT
DATALAYER
DATABASE
<<INTERFACE>>
I_DATABASE

Research

Observer pattern

In a system where something needs to be notified with something is changed, you're going to need the observer pattern. This is pattern is where we have a "Subject" and an "Observer". When a user wishes to find out about updates to a particular document, they can become attached to the document (subject) as an observer. They can then be notified about changes make to the document.

We can fine how the "Subject" behaves with a simple interface as such:

- attach(UserManagement.User observer); This will add a user to the list of users being notified about changes

- detach(UserManagement.User observer); This will remove a user if they wish to get no more updates

- notifyObserver(Document doc); This will send a message to the user that things have changed and how to view the changes

We create a new instance of the DocumentImpl class for each document. So if we wanted to have these functions from the interface used in the class, we could just implement the interface. But we went the route whereby we instantiated the DocumntSubject interface in a class of it's own called DocumntSubjectImpl, defined the methods for the three above used extended DocumntSubjectImpl when a document was changed to notify users. This is how the class looked:

```
  public ArrayList<UserManagement.User> allTheObservers = new
ArrayList<User>();


   @Override
   public void notifyObserver(Document.Document doc) {
      Mailer mailer = new MailerImpl();
      for(User user : allTheObservers)
      {
         user.notifyUsers(doc);
         mailer.sendSimpleMail(user.getEmail(), "The most amazing site in the
world",
                "The document " + doc.getTitle() + " has been updated","The doc has
been updated");
      }
   }


   @Override
   public void attach(User observer) {
      allTheObservers.add(observer);
   }


   @Override
   public void detach(User observer) {
      allTheObservers.remove(observer);
   }
```

Next part to be concerned about was how the user got notified about the changes after the message was sent. We created an interface called "UserObserver" and created the abstract methods:

- notifyUsers(Document.Document doc); This will handle what happens when the user is to be notified.

From here, we would check if the user is in the current sessions that are logged in and if they are, give them a pop up with the information that the document has been updated and a link to the new file

Database

On first look of the database system, we set out with a single class to take care of the interaction with the backend. The intent of the database throughout the entire project was to have the following methods to allow for interaction with the database:

- *readDataBase()*; This would allow us to get the database read into an array list and returned to use as needed
- *addToDocs(Document doc);* This would allow us to pass in a document and extract the required parts to add to the database, thus keeping a record of uploaded documents
- *search(String con, String value);* This would all us to search the database and only get entries for wills based on certain search criteria such as "document title" (con) and with certain values (vale)

This was the very minimum we needed to talk with the database server. Upon further use and planning, we then made an interface for the database. This was for reasons such as:

- Connections variables would always be the same, no matter who wanted to manage the database interactions
- Certain queries would always be the same, such as "insert" and "select all"

It was then decided that we would like to apply the factory method to the class. This was used as only one connection to the database needed to be open at a time and we only ever needed the one instance of the database class to perform this. Thus, we created the static function *getInstance()* as used it as follows:

```
public static DataBaseImpl getInstance()
  {
     if(dataBase == null)
     {
        try
        {
           Class.forName("com.mysql.jdbc.Driver");

           connect = DriverManager.getConnection(address, username, password);

           statement = connect.createStatement();

        }


        catch (SQLException | ClassNotFoundException ex)

        {

           Logger.getLogger(DatabaseFactory.class.getName()).log(Level.SEVERE, null,
ex);

        }

        return new DataBaseImpl();

     }

     return dataBase;

  }
```

Since we also used external libraries (the JDBC to connect to the database), we needed to include them for the connection to work. This posed a problem and a **bad code smell** as in the interface, where we first wanted to include the reference to the external library, we needed a try/catch. This meant that our interface could not hold it as everything had to be final and static and declared outside of methods and would not allow for our try/catch.

We were then posed with two schools of thought on how to use the factory method.

1. Create instances of the instance: Database database = Database.getInstance();
2. Or call statically : Database.getInstance.readDatabase();

We went with the static way of calling things since instances where not important due to the factory method and we could get the instance and use it using the static *getInstance()*

Sending emails

We decided to include a system to send emails to uses that subscribe to documents. If a user has tagged that they wish to be updated about a document but are not logged in, they will be emailed about the update. If the user is logged in and tagged that they wish to be updated, they will both receive an email and a notification onscreen of the update. Here, we're just going to talk about the sending of emails.

First off we needed to plan our interface for the class. This one was quite simple as there would only be one method required for the class to work as we wanted it:

- sendSimpleMail(String mailTo, String mailFrom, String subject, String body)

As you might see from the method, this just needed information such as:

- mailTo; who the mail would be sent to. This would be got in the form of getEmail() off the list of uses subscribed to the document
- mailFrom; Depending on the department or how we wanted this to be taken care of, this would be either the site name, a users name or the name of a department. This will say who the mail is from
- subject: Simply the subject of the email. In most cases it would be "The document <<doc.getTitle()>> has been updated."
- body: The body of the message. It may have some revision notes as set out by the author but would also contain the link to the version of the file

So now that we know how we want this class to behave, as per the interface, now we just need to implement it. Since we're using external libraries here, we had extra functionality we wouldn't normally have. Now we could use SMTP and this short code to send what we wanted

```
    Properties properties = System.getProperties();

        // Get the default Session object.
        Session session = Session.getDefaultInstance(properties);

        MimeMessage message = new MimeMessage(session);

        try {

            //Set the from field
            message.setFrom(new InternetAddress(mailFrom));

            //Set the to field
            message.addRecipient(Message.RecipientType.TO, new
InternetAddress(mailTo));

    //Set the subject field
            message.setSubject(subject);

    //Set the body of the message
            message.setText(body);

            //Attempt to send our email
            Transport.send(message);
    } catch (MessagingException messageExp) {
            messageExp.printStackTrace
    }
```

Lightweight Directory Access Protocol (LDAP)

The Lightweight Directory Access Protocol is a directory service protocol that provides a mechanism used to connect, search, and modify Internet directories.

The LDAP directory service is based on a client-server model. The function of LDAP is to enable access to an existing directory.

In a network, a directory tells you where in the network something is located. On TCP/IP networks (including the Internet), the domain name system (DNS) is the directory system used to relate the domain name to a specific network address (a unique location on the network). However, you may not know the domain name. LDAP allows you to search for an individual without knowing where they're located.

There are many different ways to provide a directory service. Different methods allow different kinds of information to be stored in the directory, place different requirements on how that information can be referenced, queried and updated, how it is protected from unauthorized access, etc. Some directory services are local, providing service to a restricted context. Other services are global, providing service to a much broader context.

In our implementation we did not have the time or scope to implement a LDAP so instead we simulated it with a LDAP Java class shown below.

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

package Login;

import Login.Student;
import Login.Lecturer;
import Services.User;
import java.util.ArrayList;
import java.util.List;

/**
 *
 * @author Team supercool
 */
public class LDAP {

    private static ArrayList<User> allThePeople;

    /**
     * Set up the LDAP system
     */
    public static void init()
    {
        allThePeople = new ArrayList<>();
        allThePeople.add(new Student("YYY", "zxxx", "asd@ul.ie", 1));
        allThePeople.add(new Student("ZZZ", "dasjb", "adsavs@ul.ie", 2));
        allThePeople.add(new Student("XXX", "advon", "ajnve@ul.ie", 3));
        allThePeople.add(new Lecturer("JJ", "asobf", "JJ@ul.ie", 4, "Boss"));

    }

    /**
     * See if the username and password match a record
     * @param ID int ID of the user trying to log in
     * @param password String Password of the user trying to log in
     * @return boolean True if record matched
     */
    public static boolean canLogIn(int ID, String password)
    {
        for(User user : allThePeople )
        {
           if(user.getID() == ID && user.getPassword().equals(password))
           {
              return true;
           }
        }
        return false;
    }

    /**
     * Find a user based on a id
     * @param id ID of the user
     * @return The user, null if not found
     */
    public User findUserByID(int id)
    {
        for(User user : allThePeople)
        {
           if(user.getID() == id)
```

```
            {
                return user;
            }
        }
        return null;
    }

}
```

We use this to simulate a student or lecturer login to the systems.

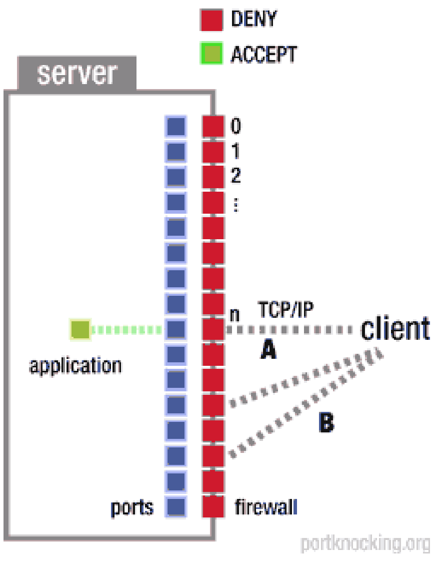
Port Knocking

Broadly, port knocking is a form of host-to-host communication in which information flows across closed ports. There are various variants of the port knocking method - information may be encoded into a port sequence or a packet-payload. In general, data are transmitted to closed ports and received by a monitoring daemon which intercepts the information without sending a receipt to the sender.

Recently a physical knock detecting device that does to the door what port knock does to your server has been reported. This knock detector is mounted on the inside of a door and listens to ... you guessed it, secret knocks. Once a knock is detected, the device unlocks the door.

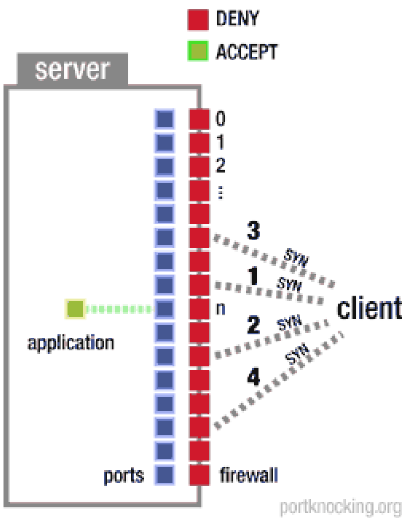
In one instance, port knocking refers to a method of communication between two computers (arbitrarily named here *client* and *server*) in which information is encoded, and possibly encrypted, into a sequence of port numbers. This sequence is termed the *knock*. Initially, the *server* presents no open ports to the public and is monitoring all connection attempts. The *client* initiates connection attempts to the *server* by sending SYN packets to the ports specified in the *knock*. This process of *knocking* is what gives *port knocking* its name. The *server* offers no response to the *client* during the *knocking* phase, as it "silently" processes the port sequence. When the *server* decodes a valid knock it triggers a server-side process.

<u>Step 1:</u> (A) client cannot connect to application listening on port n; (B) client cannot establish connection to any port



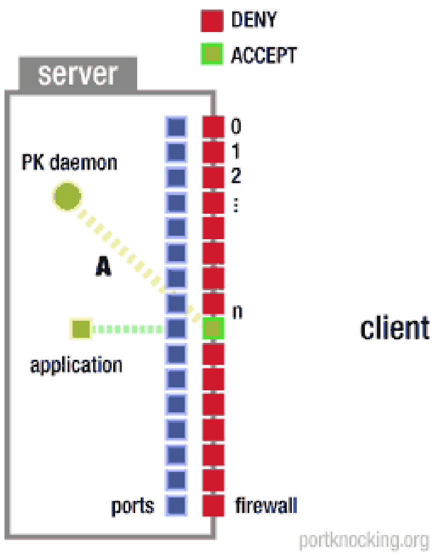
<u>Step 2:</u>

| (1,2,3,4) client connects to a well-defined set of ports in a sequence that contains an encrypted message by sending SYN packets; client has a priority knowledge of the port knocking daemon and its configuration, but receives no acknowledgement during this phase because firewall rules preclude any response



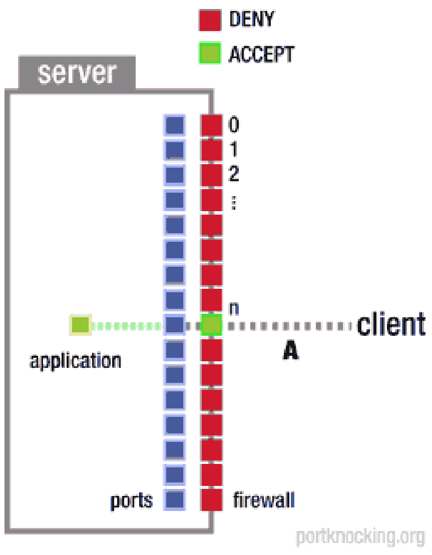
<u>Step 3:</u>

| (A) server process (a port knocking daemon) intercepts connection attempts and interprets (decrypts and decodes) them as comprising an authentic "port knock"; server carries out specific task based on content of port knock, such as opening port n to client



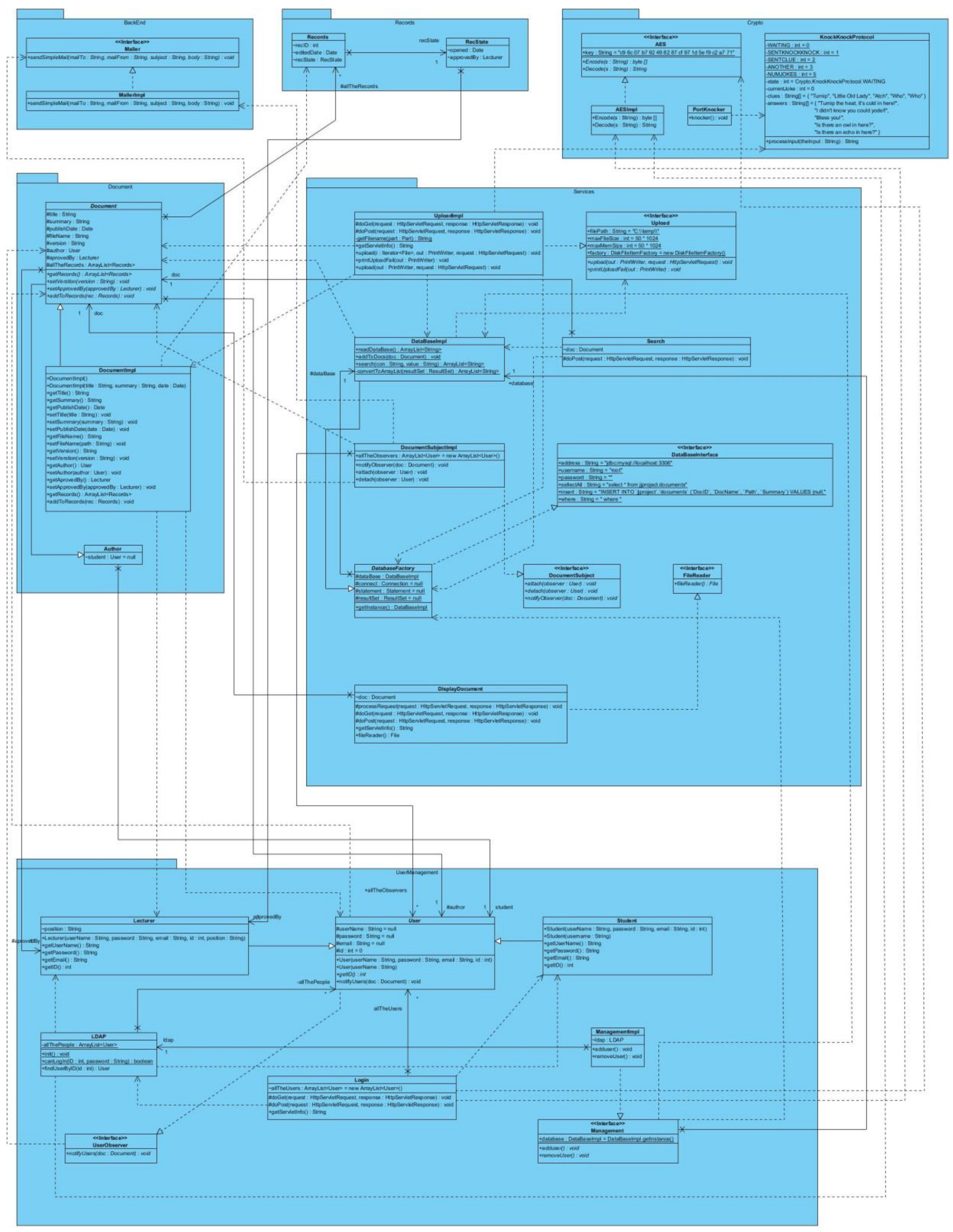
Step 4:

| (A) client connects to port n and authenticates using application’s regular mechanism

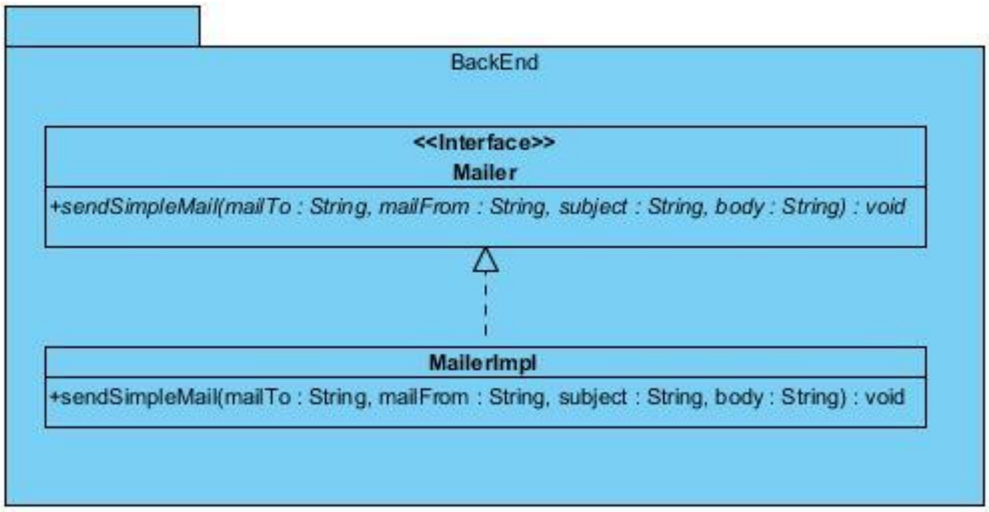


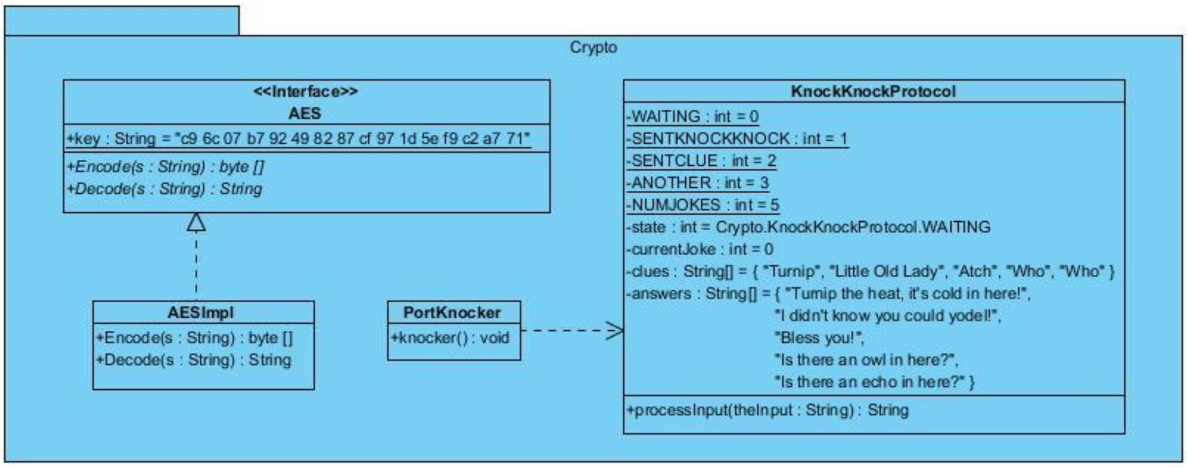
Abstraction of the implementation

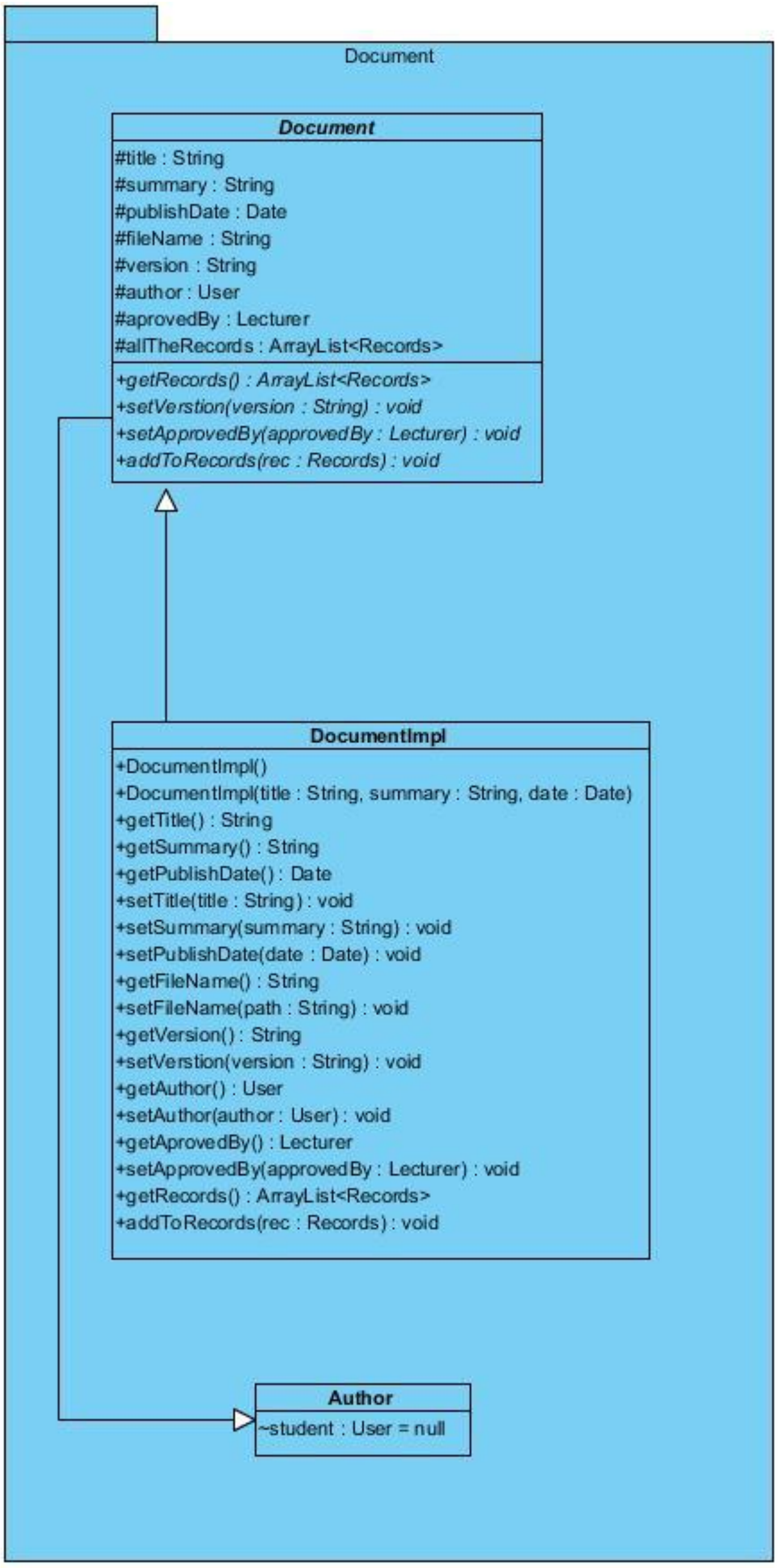
class diagram

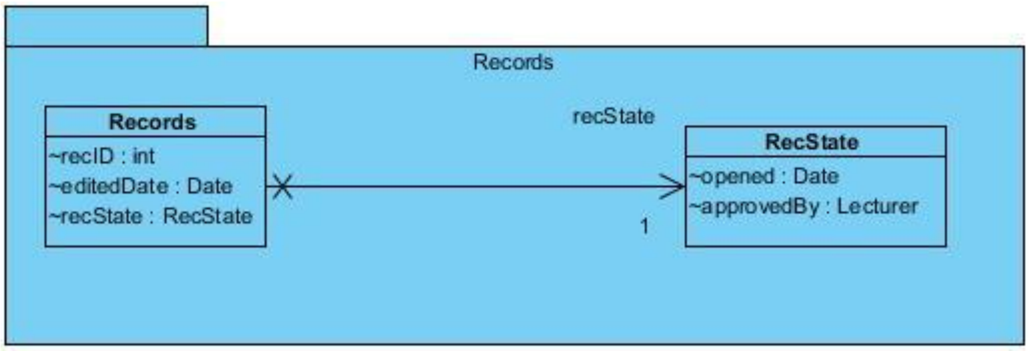


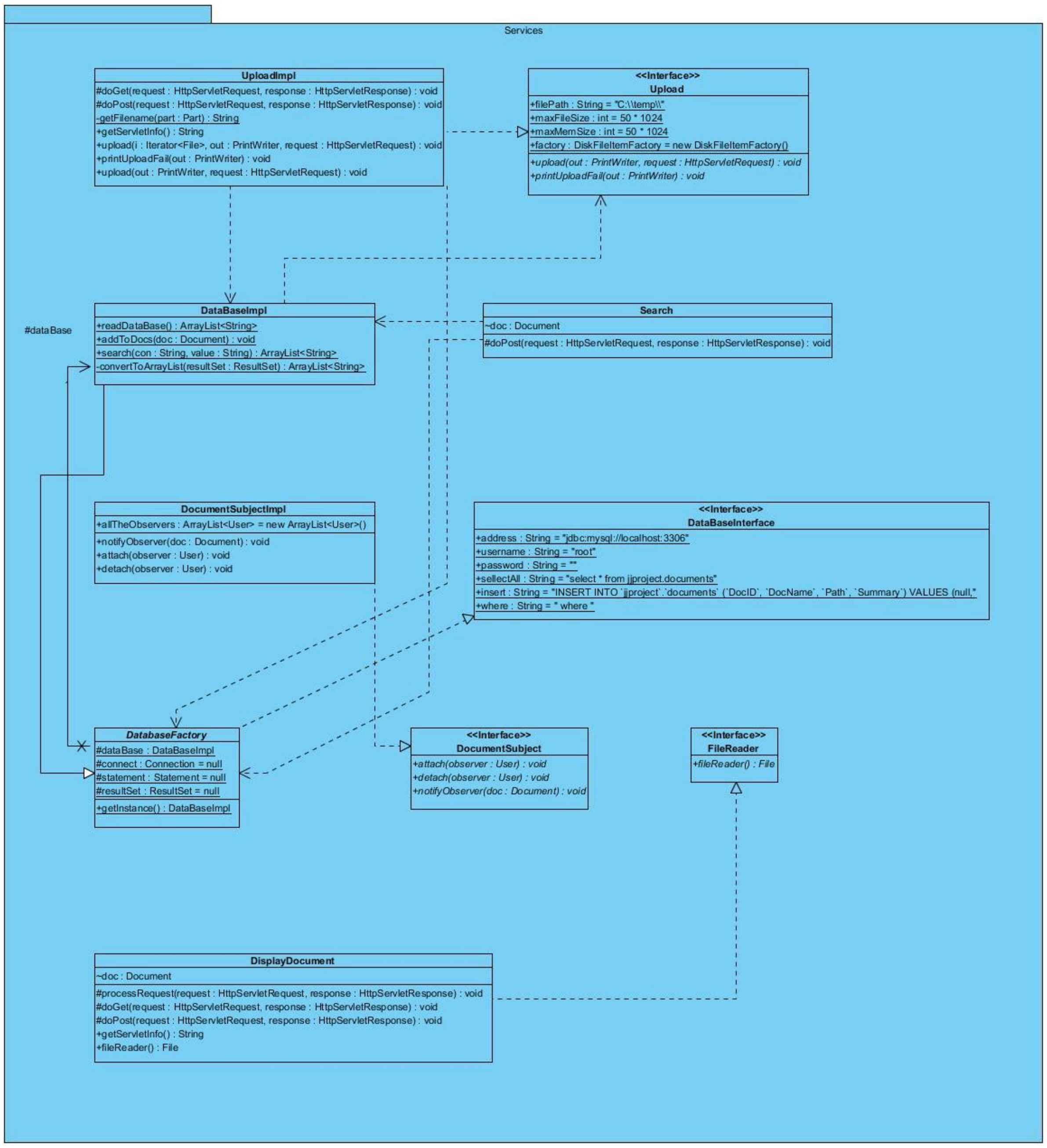
package diagrams


BackEnd
<<Interface>>
Mailer
+sendSimpleMail(mailTo : String, mailFrom : String, subject : String, body : String) : void
MailerImpl
+sendSimpleMail(mailTo : String, mailFrom : String, subject : String, body : String) : void

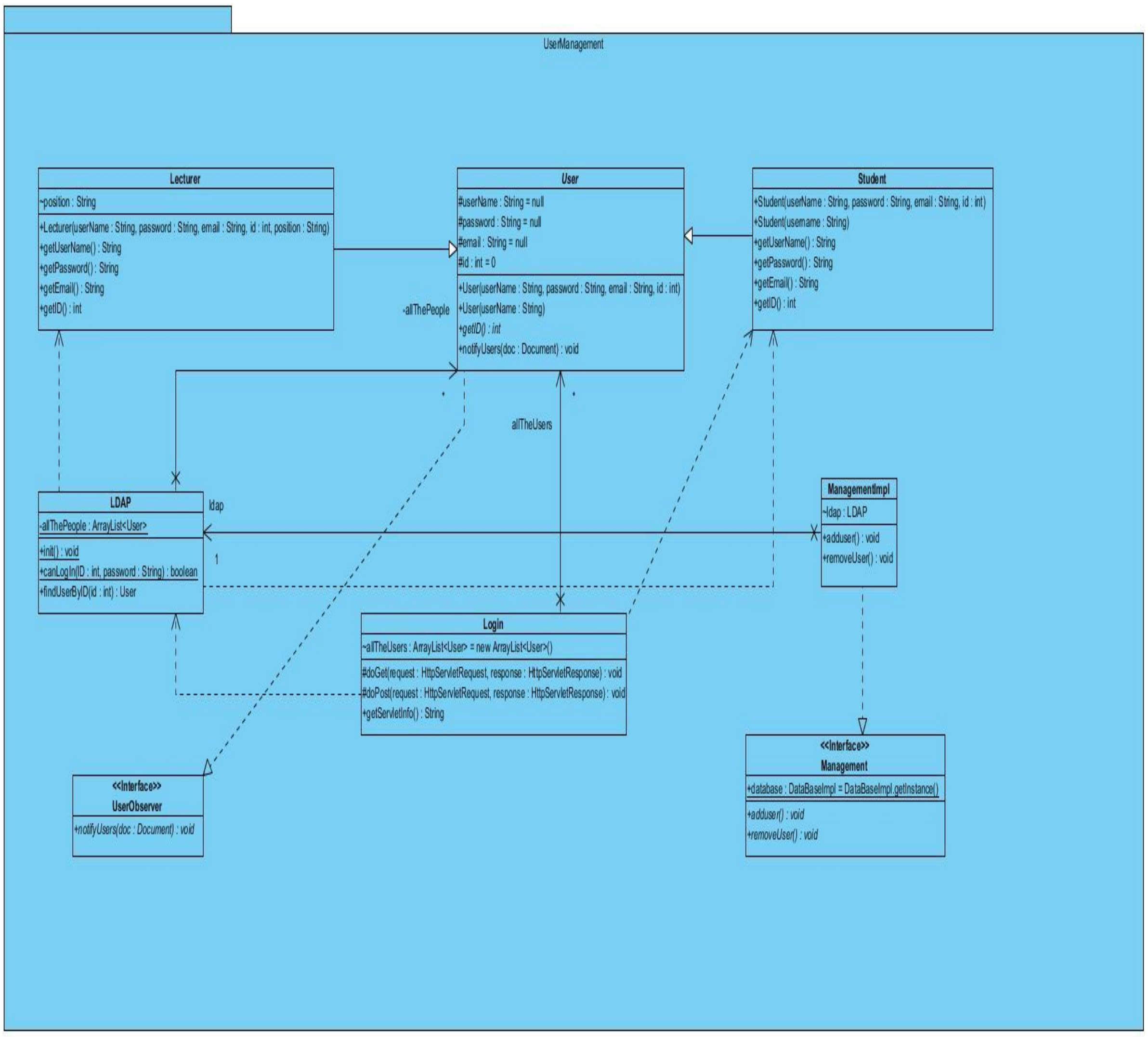

Crypto
<<Interface>>
AES
+key : String = "c9 6c 07 b7 92 49 82 87 cf 97 1d 5e f9 c2 a7 71"
+Encode(s : String) : byte []
+Decode(s : String) : String
AESImpl
+Encode(s : String) : byte []
+Decode(s : String) : String
PortKnocker
+knocker() : void
KnockKnockProtocol
-WAITING : int = 0
-SENTKNOCKKNOCK : int = 1
-SENTCLUE : int = 2
-ANOTHER : int = 3
-NUMJOKES : int = 5
-state : int = Crypto.KnockKnockProtocol.WAITING
-currentJoke : int = 0
-clues : String[] = { "Turnip", "Little Old Lady", "Atch", "Who", "Who" }
-answers : String[] = { "Tumip the heat, it's cold in here!",
"I didn't know you could yodel!",
"Bless you!",
"Is there an owl in here?",
"Is there an echo in here?" }
+processInput(theInput : String) : String


Document
Document
#title : String
#summary : String
#publishDate : Date
#fileName : String
#version : String
#author : User
#aprovedBy : Lecturer
#allTheRecords : ArrayList<Records>
+getRecords() : ArrayList<Records>
+setVerstion(version : String) : void
+setApprovedBy(approvedBy : Lecturer) : void
+addToRecords(rec : Records) : void
DocumentImpl
+DocumentImpl()
+DocumentImpl(title : String, summary : String, date : Date)
+getTitle() : String
+getSummary() : String
+getPublishDate() : Date
+setTitle(title : String) : void
+setSummary(summary : String) : void
+setPublishDate(date : Date) : void
+getFileName() : String
+setFileName(path : String) : void
+getVersion() : String
+setVerstion(version : String) : void
+getAuthor() : User
+setAuthor(author : User) : void
+getAprovedBy() : Lecturer
+setApprovedBy(approvedBy : Lecturer) : void
+getRecords() : ArrayList<Records>
+addToRecords(rec : Records) : void
Author
~student : User = null

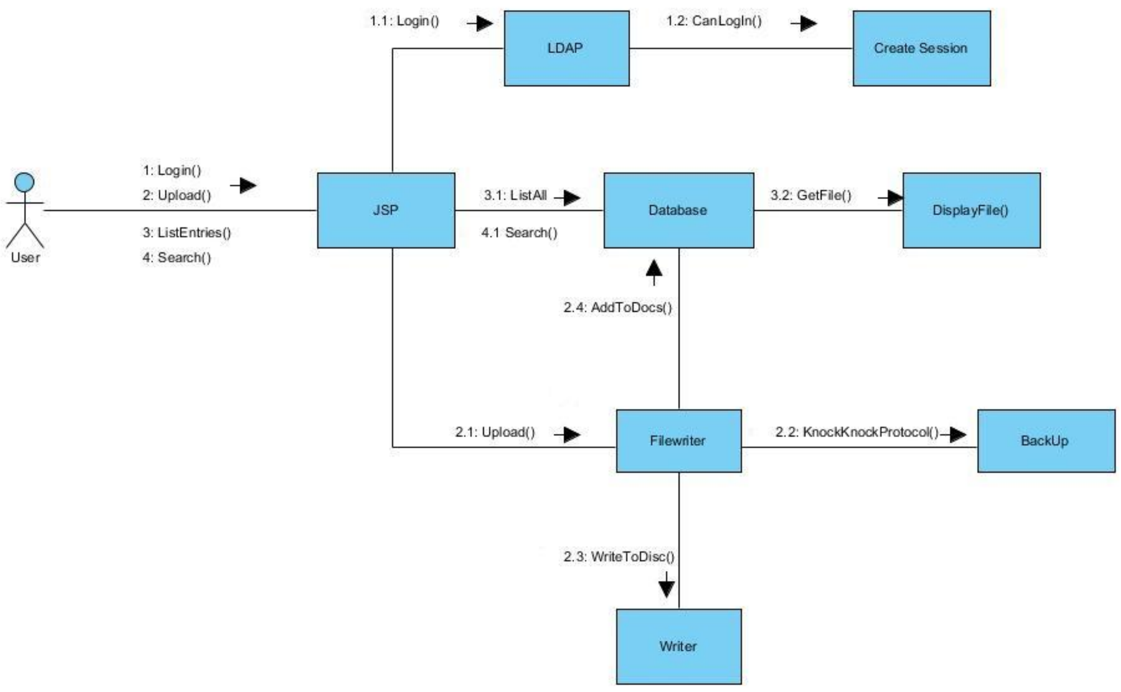

Records
Records
~recID : int
~editedDate : Date
~recState : RecState
recState
1
RecState
~opened : Date
~approvedBy : Lecturer



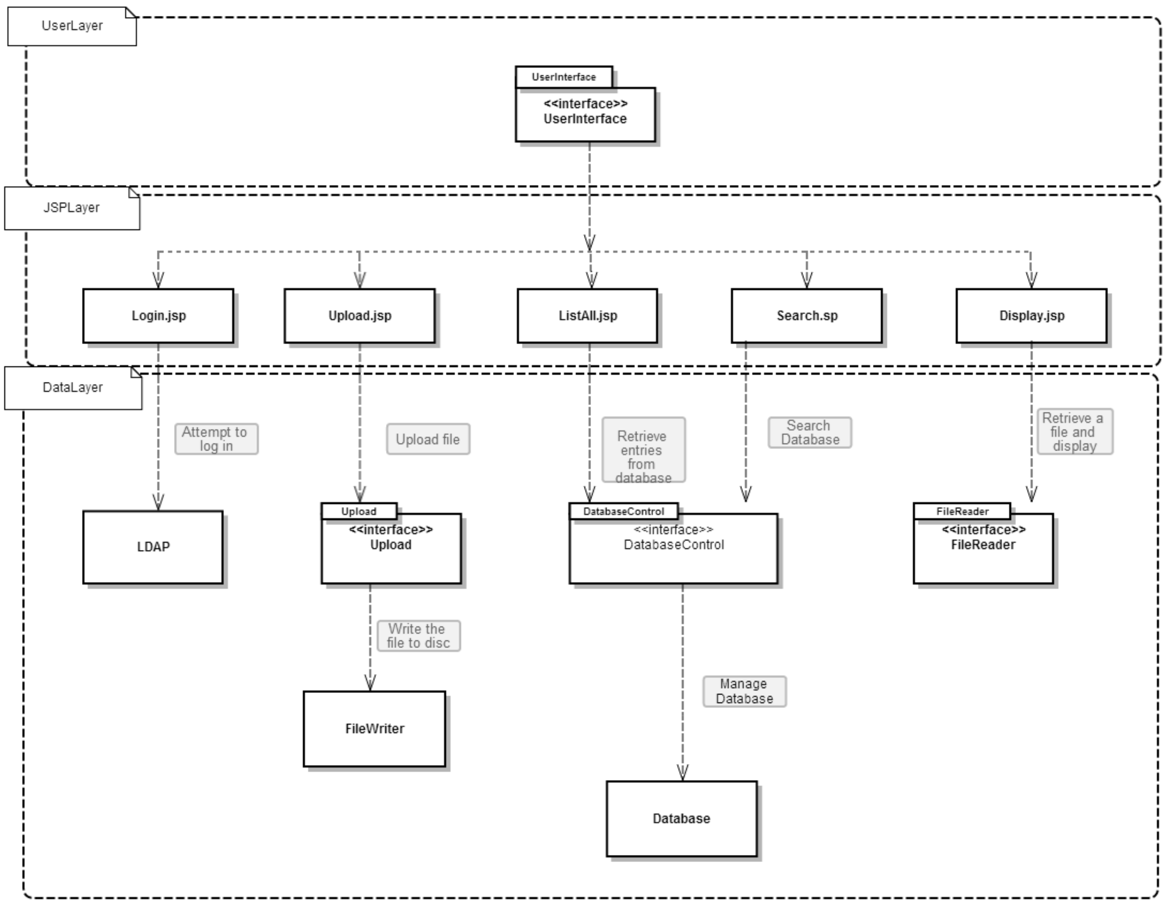
Services
UploadImpl
#doGet(request : HttpServletRequest, response : HttpServletResponse) : void
#doPost(request : HttpServletRequest, response : HttpServletResponse) : void
-getFilename(part : Part) : String
+getServletInfo() : String
+upload(i : Iterator<File>, out : PrintWriter, request : HttpServletRequest) : void
+printUploadFail(out : PrintWriter) : void
+upload(out : PrintWriter, request : HttpServletRequest) : void
<<Interface>>
Upload
+filePath : String = "C:\\temp\\"
+maxFileSize : int = 50 * 1024
+maxMemSize : int = 50 * 1024
+factory : DiskFileItemFactory = new DiskFileItemFactory()
+upload(out : PrintWriter, request : HttpServletRequest) : void
+printUploadFail(out : PrintWriter) : void
DataBaseImpl
+readDataBase() : ArrayList<String>
+addToDocs(doc : Document) : void
+search(con : String, value : String) : ArrayList<String>
-convertToArrayList(resultSet : ResultSet) : ArrayList<String>
#dataBase
Search
~doc : Document
#doPost(request : HttpServletRequest, response : HttpServletResponse) : void
DocumentSubjectImpl
+allTheObservers : ArrayList<User> = new ArrayList<User>()
+notifyObserver(doc : Document) : void
+attach(observer : User) : void
+detach(observer : User) : void
<<Interface>>
DataBaseInterface
+address : String = "jdbc:mysql://localhost:3306"
+username : String = "root"
+password : String = ""
+sellectAll : String = "select * from jjproject.documents"
+insert : String = "INSERT INTO `jjproject`.`documents` (`DocID`, `DocName`, `Path`, `Summary`) VALUES (null,"
+where : String = " where "
DatabaseFactory
#dataBase : DataBaseImpl
#connect : Connection = null
#statement : Statement = null
#resultSet : ResultSet = null
+getInstance() : DataBaseImpl
<<Interface>>
DocumentSubject
+attach(observer : User) : void
+detach(observer : User) : void
+notifyObserver(doc : Document) : void
<<Interface>>
FileReader
+fileReader() : File
DisplayDocument
~doc : Document
#processRequest(request : HttpServletRequest, response : HttpServletResponse) : void
#doGet(request : HttpServletRequest, response : HttpServletResponse) : void
#doPost(request : HttpServletRequest, response : HttpServletResponse) : void
+getServletInfo() : String
+fileReader() : File


UserManagement
Lecturer
~position : String
+Lecturer(userName : String, password : String, email : String, id : int, position : String)
+getUserName() : String
+getPassword() : String
+getEmail() : String
+getID() : int
User
#userName : String = null
#password : String = null
#email : String = null
#id : int = 0
+User(userName : String, password : String, email : String, id : int)
+User(userName : String)
+getID() : int
+notifyUsers(doc : Document) : void
Student
+Student(userName : String, password : String, email : String, id : int)
+Student(usemame : String)
+getUserName() : String
+getPassword() : String
+getEmail() : String
+getID() : int
-allThePeople
*
*
allTheUsers
LDAP
ldap
-allThePeople : ArrayList<User>
+init() : void
+canLogIn(ID : int, password : String) : boolean
+findUserByID(id : int) : User
1
ManagementImpl
~ldap : LDAP
+adduser() : void
+removeUser() : void
Login
~allTheUsers : ArrayList<User> = new ArrayList<User>()
#doGet(request : HttpServletRequest, response : HttpServletResponse) : void
#doPost(request : HttpServletRequest, response : HttpServletResponse) : void
+getServletInfo() : String
<<Interface>>
UserObserver
+notifyUsers(doc : Document) : void
<<Interface>>
Management
+database : DataBaseImpl = DataBaseImpl.getInstance()
+adduser() : void
+removeUser() : void

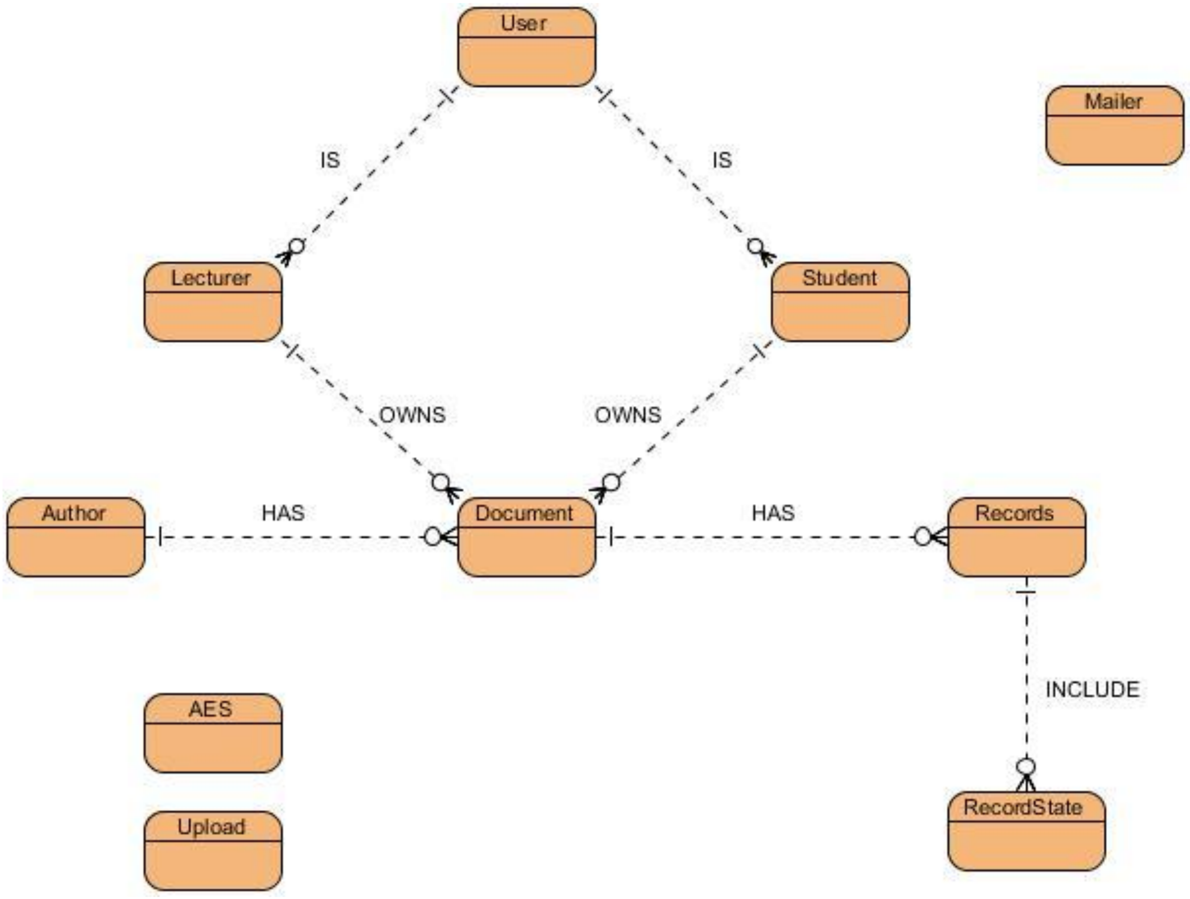
Communication diagram



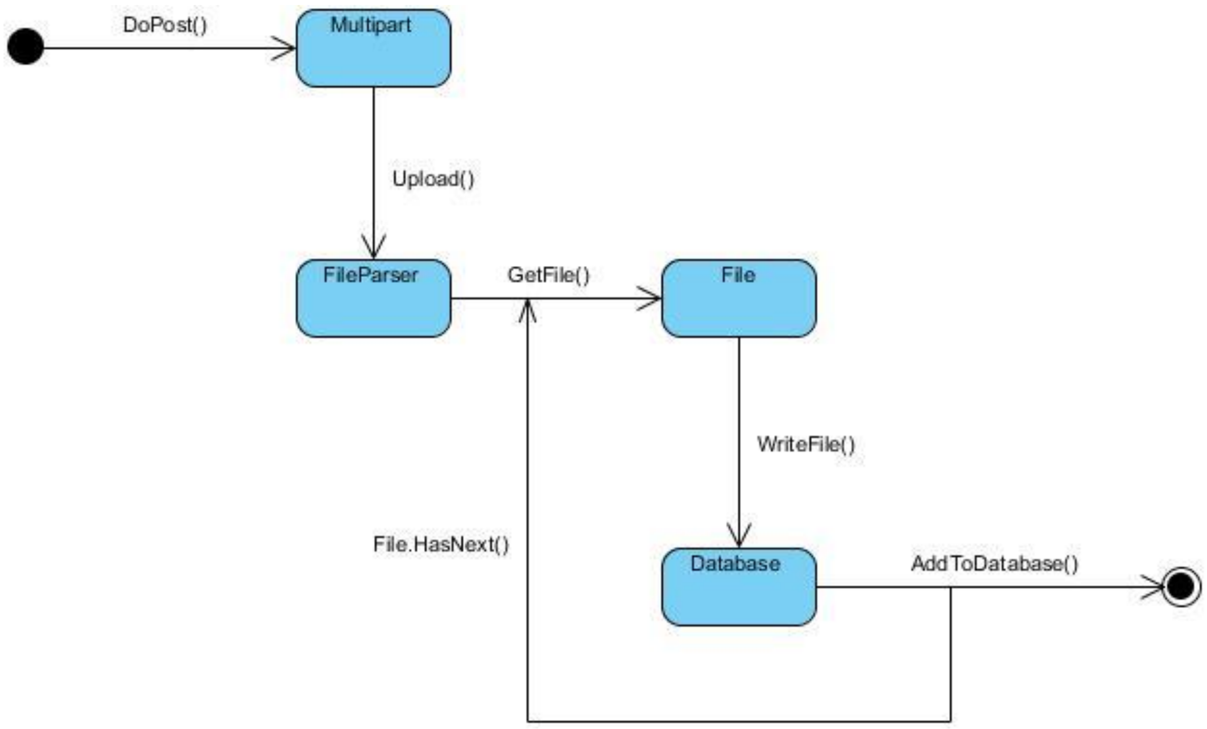
System architecture diagram with interfaces


UserLayer
UserInterface
<<interface>>
UserInterface
JSPLayer
Login.jsp
Upload.jsp
ListAll.jsp
Search.sp
Display.jsp
DataLayer
Attempt to log in
Upload file
Retrieve entries from database
Search Database
Retrieve a file and display
LDAP
Upload
<<interface>>
Upload
DatabaseControl
<<interface>>
DatabaseControl
FileReader
<<interface>>
FileReader
Write the file to disc
FileWriter
Manage Database
Database

ER diagram


User
Mailer
IS
IS
Lecturer
Student
OWNS
OWNS
Author
HAS
Document
HAS
Records
INCLUDE
AES
Upload
RecordState

State chart


DoPost()
Multipart
Upload()
FileParser
GetFile()
File
WriteFile()
File.HasNext()
Database
AddToDatabase()

Evaluation

After creating the initial plan for the project, we found that we had not taken everything into account and were left with issues we didn't foresee. This meant we had to go back to the drawing board (pun intended) and redesign the project from the ground up. We found that despite thinking we knew how to plan the project, the more we coded and learned in class, the more the code shaped with experience.

We would have a considerably better understanding of how to and why you would design to interfaces and not implementation. How interfaces can shape classes and their behaviour. The ideas behind many design patterns and how they can make the coding process more efficient and structured.

UML

As a view for the UML, without an experience of how to use it, a planner or software engineer may not foresee all the problems and could plan a whole team into a corner. We would agree it's a good system but would consider that a lot of time is needed to get a deep down understanding of software design before you should go planning with UML

From a point of view of reading it, it explains a lot about the code and allows the coder to visualise the structure of the program without needing to code it all. It's easy to read and can be a big help to implement a design.

REFERENCES:

http://msdn.microsoft.com/en-us/library/aa367008(v=vs.85).aspx

http://compnetworking.about.com/library/glossary/bldef-ldap.htm

http://www.portknocking.org/