$(mail)^n$

Elaboration 1

CS616
Fall 2004
Dr. Marchese
November 10, 2004

Elias Rosero
Jwalant Dholakia
Joseph Aulisi

Domain Model

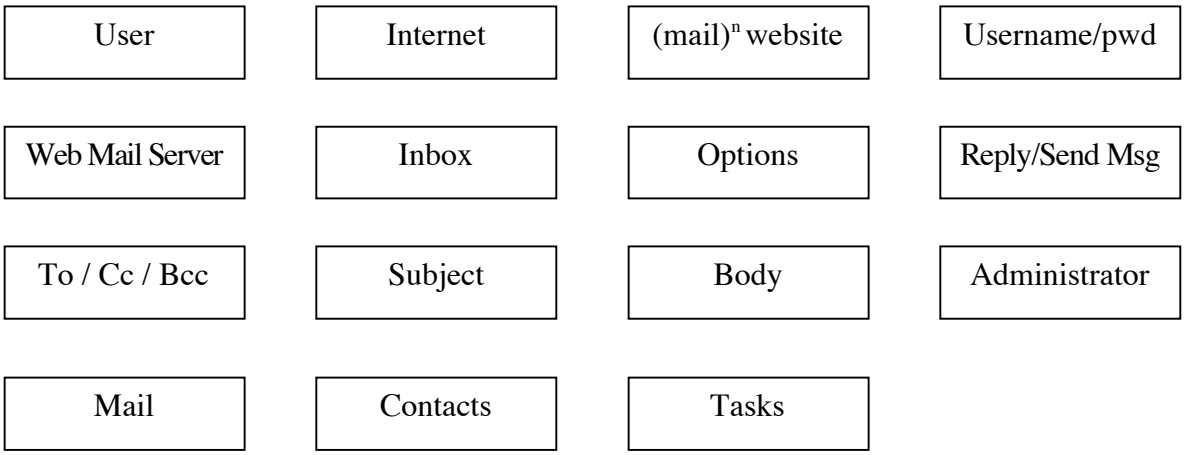
Conceptual Class Category	Examples
physical or tangible objects	*Physical Network, Internet Access Terminals, and Server*
specifications, designs, or descriptions of things	*System Documentation, Prototypes, and Interface Design*
Places	*Company Offices, and Physical Server Location*
roles of people	*Administrator, and User*
containers of other things	*Server Storage Drives*
things in a container	*Data (e-mails, contact information etc)*
other computer or electro-mechanical systems external to the system	*Internet, Web Servers, and Browsers*
abstract noun concepts	*Workaholic, and Technical*
organizations	*End User Company*
events	*Mail Sent, Mail Received, Mail Deleted etc*
processes (often not represented as a concept, but may be)	*New user account Created*
rules and policies	*Terms and Conditions of E-mail Service use, Certificate of Liability*
records of finance, work, contracts, legal matters	*Contract signed with Customer Company*
Catalogs	*Special Features of (mail)n listed for access to potential buyer companies*
manuals, documents, reference papers, books	*User Guide/Manual, Documentation, Troubleshooting Manual, FAQ's*

Finding Conceptual Classes using Main Success Scenario

The following Conceptual Classes have been identified from Main Success Scenario (Basic Flow) and other alternative scenarios for $(mail)^n$ specified in the Inception Document

- User
- Internet
- $(mail)^n$ website
- Username / password
- Database system
- Inbox
- Options
- Reply, CC
- Subject
- Body
- Send
- Logout

The following is a graphical representation of (mail)^n Domain model specified without attributes and associations:



The following is final version of the Domain Model for (mail)n based on the above mentioned conceptual classes. We have not specified every possible association between conceptual classes for purposes of simplicity. Also, class relationships have been identified and relevant attributes mentioned.

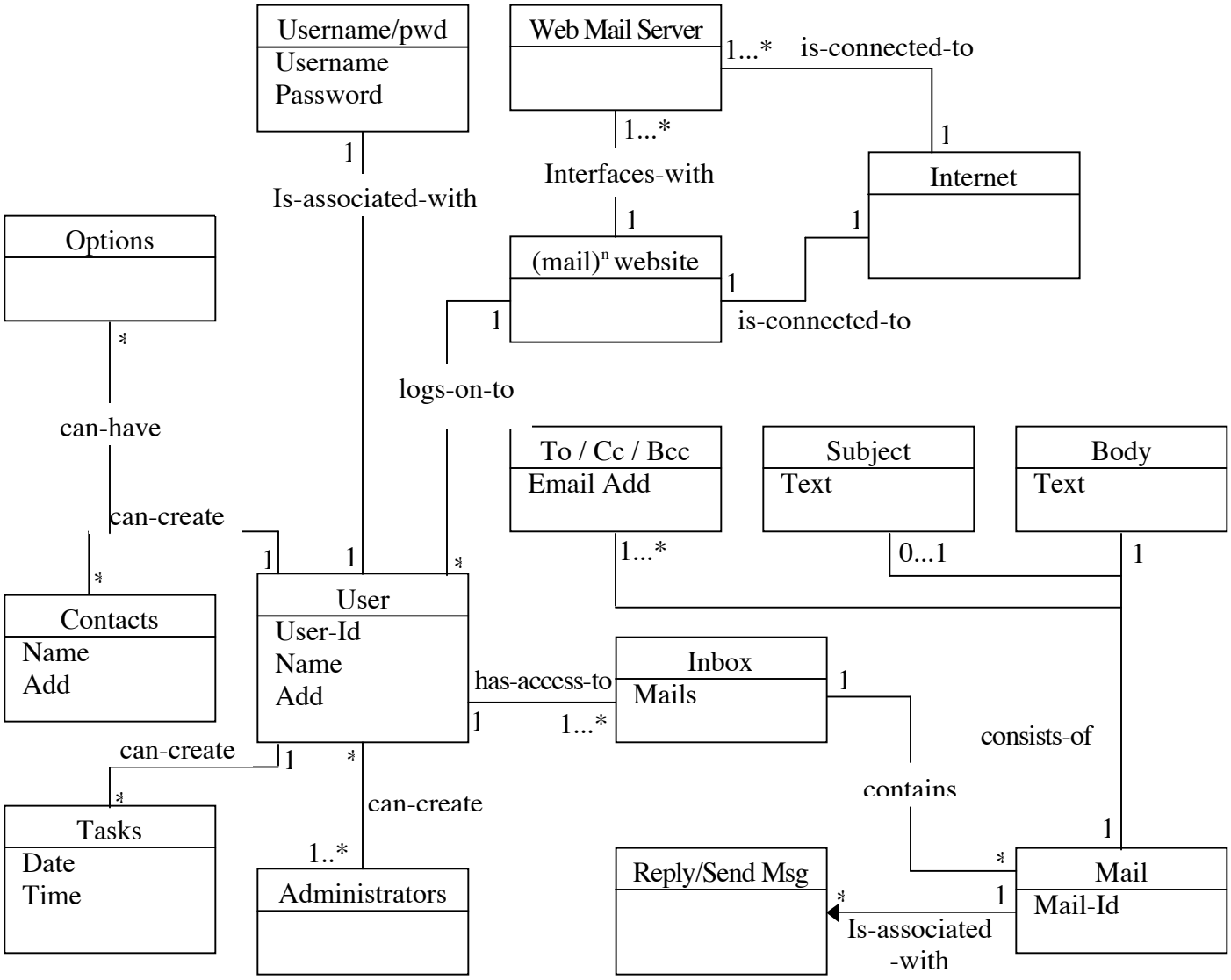


Fig: Domain Model

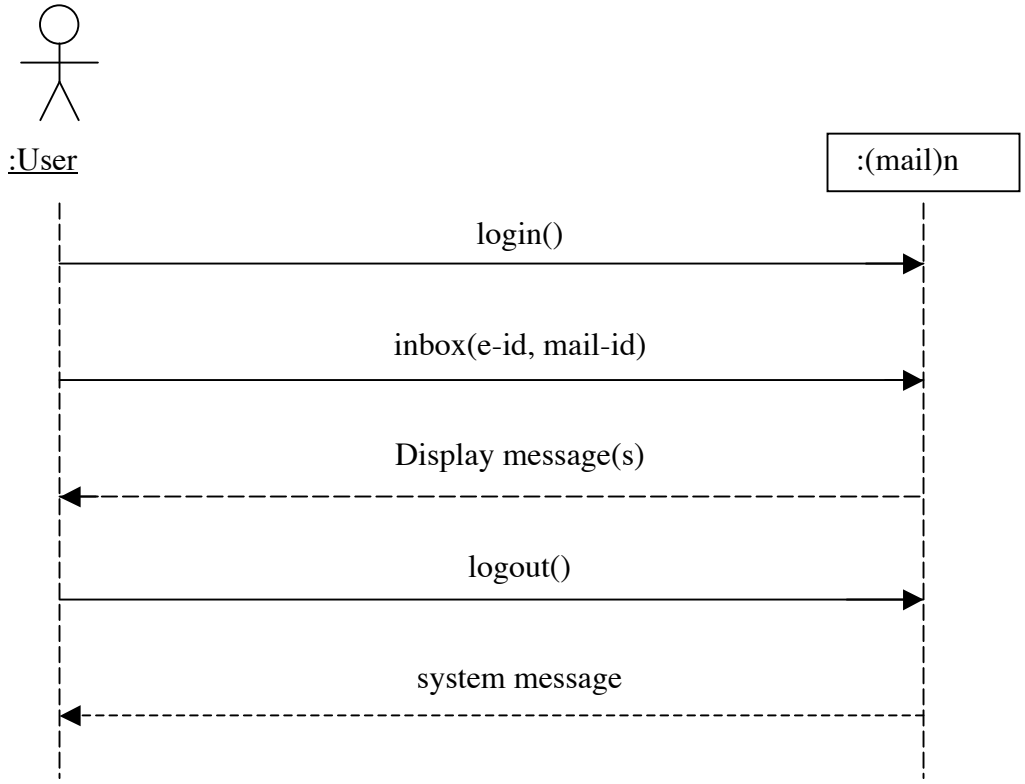
Use-Case Model

System Sequence Diagram

The system sequence diagrams presented in this section are graphical representations of the use-case scenarios presented in earlier documentation. The system is presented as a “black box”, and the diagram has a chronological order where time increases downward.

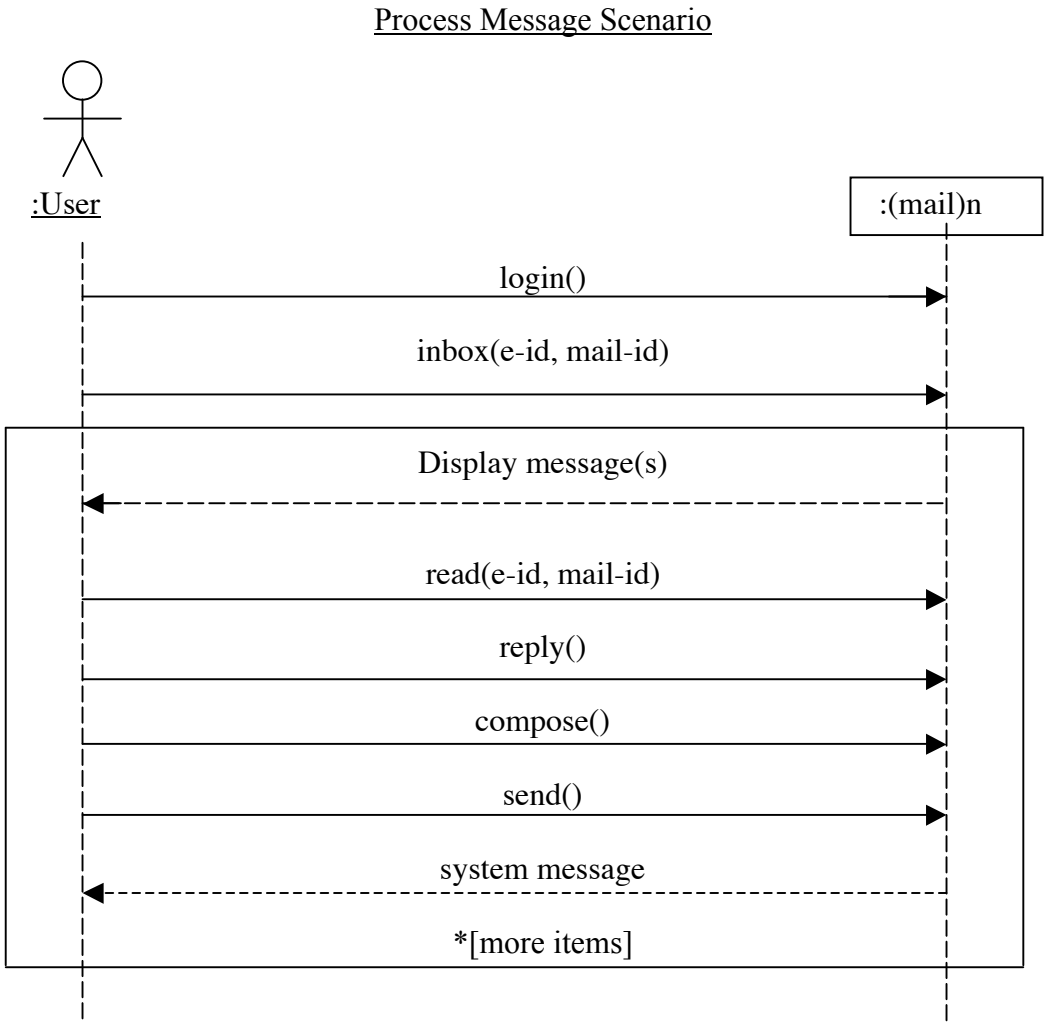
The following is a system sequence diagram for the “happy path” used in the Inception document presented earlier.

Viewing Inbox Scenario



As stated in the use-case scenario, the above sequence diagram represents a simple path of a user entering the system, and simply viewing the contents of the inbox. After viewing the email messages, the user logs out.

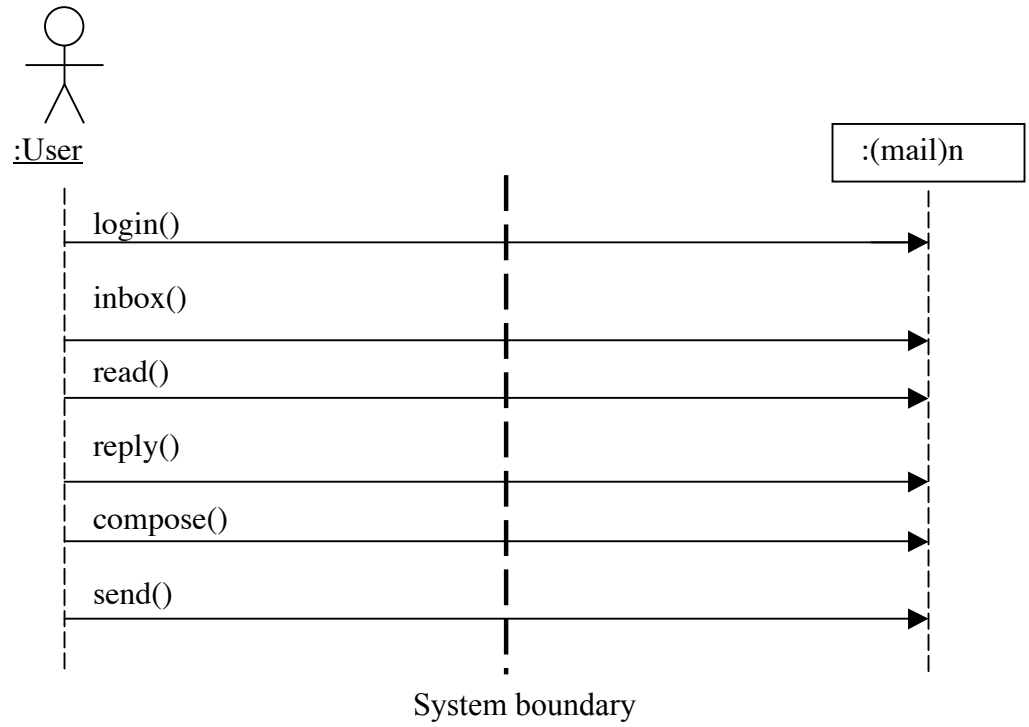
The following SSD will follow a slighter more complicated path, which we consider to be the most common one when using an email system.



The box encapsulating the inbox(), read (), reply(), compose(), and 0 send(), represents an iteration that may occur when using the system. In other words, the user may do this as many times as needed. The notation "*[more items] suggests this.

Identifying System Events

The system boundaries are defined by grouping the system events. In the SSD, the system events are those actions that the user invokes for the system to do. For simplicity sake, the system events presented here are only some examples of the entire design model.



The SSDs are used to clarify the major operations that the system is designed to handle. Therefore, SSDs are not necessary for every path through the system. We have chosen the most common paths that (mail)n is going to be used for. The system events listed above will be used in the following section of this elaboration document to provide Operation Contracts.

Use-Case Model: Adding Detail with Operation Contracts

Operation contracts are used to describe the system behavior in terms of state changes to objects in the Domain Model, after a system operation has been executed. In other words, we use the system events in the previous section in order to identify a system operation, and from there we elaborate on the contract. The contract specifies what the system is required to do, and ties the relationships between the system and the domain model (the real world).
Operation Contracts will be used only on those system operations that are deemed to be of the most importance in (mail)n. Our domain model and design model are very similar to each other, because an email system takes place in a cyber space where email messages are sent electronically. Thus, the operation contracts will reflect this similarity

and use factors such as the mail server and web server to be our "real world" domain model.

Contract C1: send

Operation:	send(message-id, e-id, recipients)
Cross References:	Use Case UC1: Processing Messages
Pre-Conditions:	at least the "To" field is filled with a valid email address.
Post-Conditions:	- a unique message id, was created - the message id was associated with the current e-id - a connection with SMPT was made - the database was updated to reflect the message sent action

Contract C2: fetch

Operation:	fetch(message-id, e-id, recipients, authenticator)
Cross References:	Use Case UC1: Processing Messages
Pre-Conditions:	there is an email(s) underway
Post-Conditions:	- the message was stored in the inbox folder - a connection with IMAP was made - an authentication was made associated with creating a session

The operation contracts are useful in that a clear association between the domain model and the system operation is made, without the details of how it's done. The domain model attributes are changed, formed or broken from the system operations. And the "post-conditions" are used to narrate those changes using plain English.

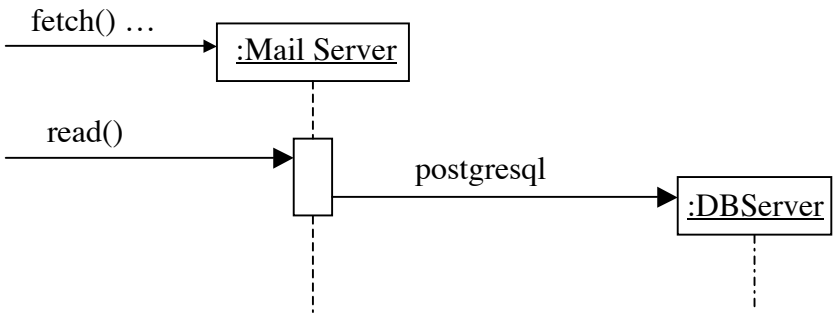
Design Model: Use-Case Realizations

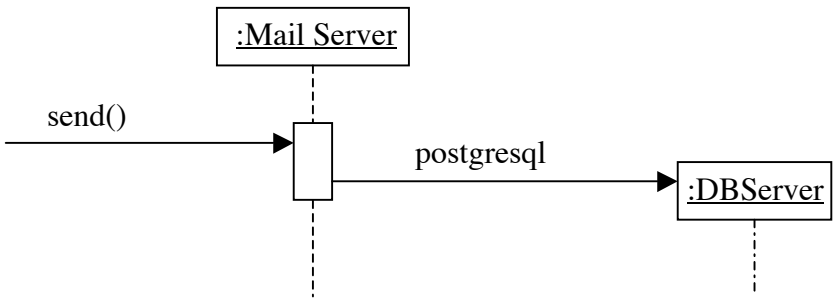
So far, we have used SSDs to illustrate the system boundary and identify the system events. Then we used those system events to elaborate the operation contracts, which shows the system operations associated with the domain model.
Now, we are going to present Use- Case Realizations. The system events are going to be used once again. But within interaction diagrams, which illustrate how objects interact to complete the tasks that the system must be able to perform.

As described in the SSD, the most important system events are:

- *Process Message: read(), send()*

For each system even, we are going to create a sequence diagram, as follows:





Software Architecture Document

Architectural Representation

The architecture of $(mail)^n$ will be described using technical memos and architectural views.

Architectural Factors and Decisions

In the Supplementary Specification, we touched upon the factors which must be complete when $(mail)^n$ is finally released. It outlines the FURPS considered during the Inception phase. A series of technical memos follow, further detailing our original intentions.

Technical Memo
Issue: Functionality

Solution Summary: Email is saved in a database to provide permanence and accessibility.

Factors
• (mail)n must store all email messages for individuals of a company, but must also be able to send messages, and to save contact and task information.

Solution
Email messages are downloaded from a mail server, chopped into component parts, then each message part is saved to the mail database. Messages are sent via an SMTP agent, which checks the sending credentials of a user with SMTP Authorization. Each user of $(mail)^n$ can store contact information and tasks to the database by issuing insert statements.

Motivation
Email users want to be able to save the messages they have received. As a user receives more messages, it becomes difficult to manage all this received mail. The database provides an easy way to retrieve these messages.

Technical Memo
Issue: Security

Solution Summary: Security is provided by a mandatory login, and via SSL.

Factors
• Users of $(mail)^n$ must be shielded from those who would try to gain unauthorized access to the system.

• Users must be reassured that their system passwords cannot be discovered by potential intruders.

Solution
Security is provided through a call to the database, to check the provided username and password against the database. If a match is found, a session variable is created to hold these values. If no match is found, the intruder is locked out of the system, and shown only the initial login page. Each page of the system must check the session variable for proper credentials, to avoid any back-door entry into the system.

The system must be accessed via SSL to ensure that passwords are protected, and sent over the internet as plain text.

Motivation
Corporate email is sensitive information. If $(mail)^n$ were easily cracked, it would never be considered as a viable email option.

Unresolved Issues
Purchase and installation of an SSL Certificate

Technical Memo
Issue: Usability

Solution Summary: Ease of use and a comfortable web-based interface are provided.

Factors
• Users must be comfortable working with the system.

Solution
Since $(mail)^n$ is web-based, it uses an interface that most people already know how to use. The interface is friendly, and uses a color palette that is easy on the eyes.

Motivation
Email must be available to corporate users from anywhere in the world. By making $(mail)^n$ a web-based application, users can easily access new messages, but also have access to all archived messages.

Technical Memo
Issue: Reliability–Recovery from Failure

Solution Summary: Email is held offsite in case of network failure, and backed up regularly.

Factors

- (mail)n must always be accessible.
- If system fails, recovery must be timely.

Solution
$(mail)^n$ should be employed on a network that is always connected to the internet, and on trusted hardware. In case the internet network experiences outages, email will be held on the mail server until it can be downloaded into the database. The mail database is backed up regularly, in case the hardware itself should fail.

Motivation
Messages arrive moment by moment, and need to be stored on a system that requires little human intervention.

Unresolved Issues
What is the best way to back up the database?

Technical Memo
Issue: Implementation Constraints

Solution Summary: $(mail)^n$ uses open-source and free components.

Factors
- A relational database is needed.
- JavaMail must be used to retrieve and send mail.

Solution
The database of choice is PostgreSQL. It is an open-source, object-relational database that is robust and SQL standards compliant. In order to communicate with the mail server, we will be using JavaMail, as it is free to use, and already has all the methods we need to retrieve email, and also to send it using SMTP Authorization.

Motivation
If open-source components were not used, we could not offer the system free of charge.

Logical View

The major subsystems of $(mail)^n$ include the files hosted by the web server, the classes that communicate with the mail server, and the database server. The web server, running Apache Jakarta Tomcat, serves JSP pages. Most of the JSP pages make a call to the database to retrieve data, based on the login id of the user. JSP is most important to the user interface, to quickly display the contents of the database. In order to make this work, it is necessary to have the proper JDBC class written expressly for PostgreSQL. Luckily, this file is supplied with the PostgreSQL database. The only caveat is to make sure this file is placed in the correct classpath.

The most important classes in the system are Fetch and Send. Fetch iterates through all ids in the login table, using the id and password, makes a call to the mail server and downloads all the mail for each user. Messages are parsed on the spot, and written to the database. Send works a bit differently, as it takes data from an HTML form, writes each field to the mail database, creates a singular mail message, which is then sent via JavaMail, using SMTP Authentication.

Process View

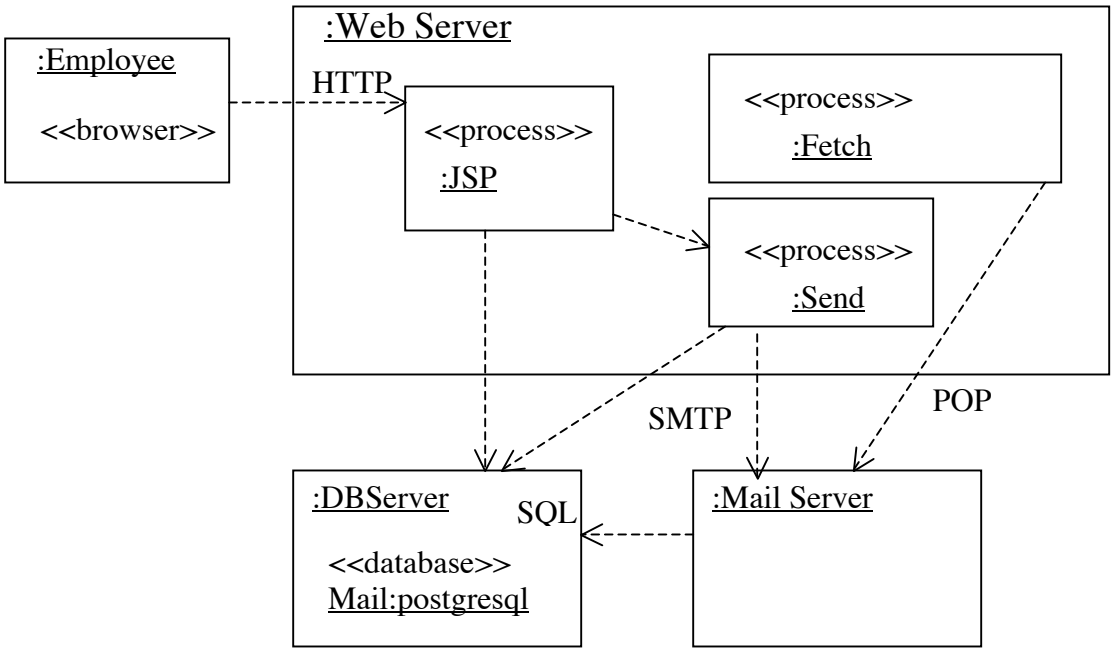
When a new message appears on the mail server, JavaMail takes the steps necessary to communicate with the remote mail server, supplying it with the proper user name and password. Once the connection is made, each part of the message is easily retrieved by JavaMail methods, finally resulting in an insert into the Mail table.

The class responsible for downloading mail, Fetch, must be run often, to ensure that email reaches the users in a timely manner. To achieve this, it will be necessary to create a cron job that runs the Fetch class once per minute. The Send class, on the other hand, only has to be run when it is invoked by a user sending mail.

Use-Case View

The most architecturally significant use case is that of UC-1 Processing Messages. This use case illustrates how an employee logs into $(mail)^n$, reads the messages that have been saved to the database, then sends a new message. This use case covers all the architecture, as it shows the employee logging into the system via the web server. The Fetch class, which works in the background, retrieves messages from the mail server and saves them to the database. The web server in turn displays these messages to the employee.

Deployment View



This deployment diagram shows the three distinct networking components, namely, the web server, mail server and database server. The diagram shows how the Fetch process independently POPs mail off of the mail server, and inserts it into the database. The processes named JSP and Send are connected to the Employee. These processes are only executed when an employee is actively using the system. When a message is sent, the employee fills in a JSP form. The form calls the Send class as an action. The Send class logs into the mail server using SMTP Authorization, and sends the message. The Send process will also record this transaction to the database.

Data Model

Data description
Major data objects
The following data objects will be presented and managed by the system:

1. Authentication module

The functionality of this module is achieved by using the login database at the backend. The purpose of this module can be described as doing the task of authenticating the user into the system on the basis of the user id and the password provided and also providing facility to mail the forgotten password on the secondary email address of the user if such a request is made. Updating of the records is also permitted in that if a user wants to change his password, he is allowed to do so.

2. Employee information module

This module stores the personal information of the employee in the database. Once the employee logs into the system for the first time, the employee is requested to enter his personal information, like name, address, telephone number, etc. into the system. The employee can also later update/modify his personal information as required.

3. Contact information module

The employee using (mail)n has the facility to store his public/private contacts which can be used while sending mail. The employee can also modify and/or delete contacts.

4. Tasks module

Day-to-day tasks for a particular employee can be stored in this calendar-like task module. Updating and deleting tasks is allowed.

5. Mail module

The mail module is the crux of the system. The employee has the ability to send, receive and delete messages and manage his mail by putting them in various user-defined folders.

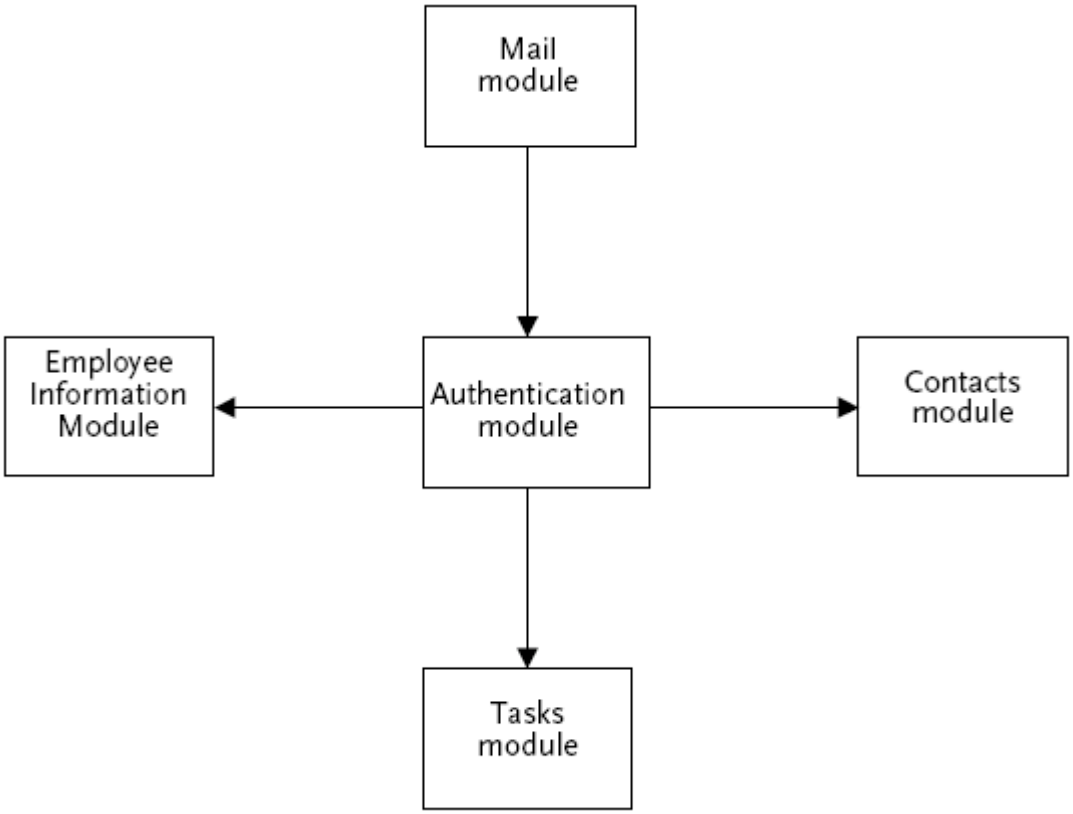
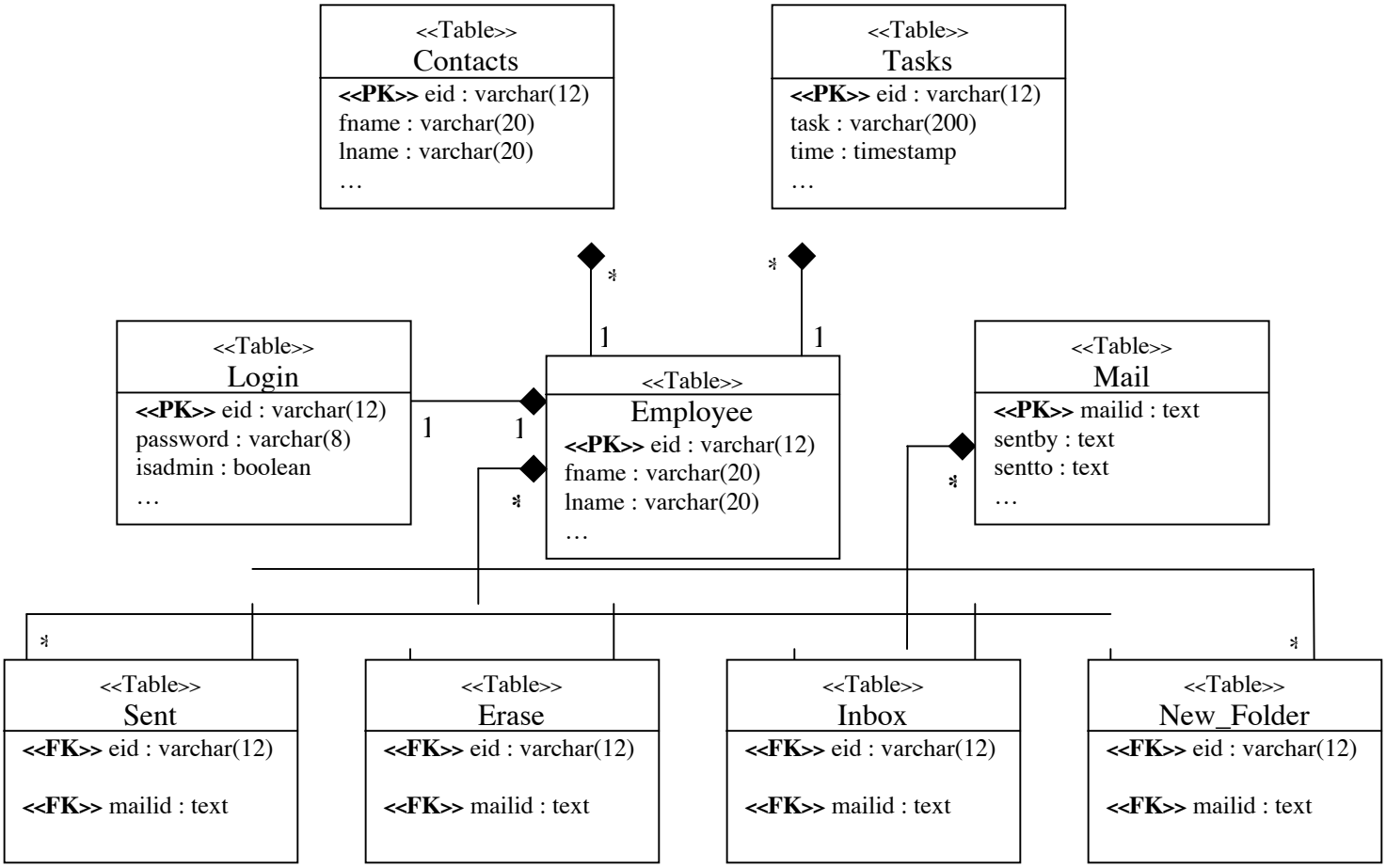


Fig: Architecture Model


<<Table>>
Contacts
<<PK>> eid : varchar(12)
fname : varchar(20)
lname : varchar(20)
…
<<Table>>
Tasks
<<PK>> eid : varchar(12)
task : varchar(200)
time : timestamp
…
<<Table>>
Login
<<PK>> eid : varchar(12)
password : varchar(8)
isadmin : boolean
…
<<Table>>
Employee
<<PK>> eid : varchar(12)
fname : varchar(20)
lname : varchar(20)
…
<<Table>>
Mail
<<PK>> mailid : text
sentby : text
sentto : text
…
<<Table>>
Sent
<<FK>> eid : varchar(12)
<<FK>> mailid : text
<<Table>>
Erase
<<FK>> eid : varchar(12)
<<FK>> mailid : text
<<Table>>
Inbox
<<FK>> eid : varchar(12)
<<FK>> mailid : text
<<Table>>
New_Folder
<<FK>> eid : varchar(12)
<<FK>> mailid : text
1
1
1
1
*
*
*
*
*
*

Implementation Model

(mail)^n is essentially a series of JSP pages that live in a directory of an installation of Tomcat. The site itself is a frameset with top, left, and right frames. Each of these frames displays its own JSP page. The Top page, for instance, shows the (mail)n logo, it shows the date, and offers links to compose a new message, show contacts and tasks, perform a search, set options, and to logout. The left frame displays the employee mail folders. The right frame is the target frame for all links. All the action will take place in the right frame.

Since (mail)^n is a database-driven website, most pages will require a call to the database. There must be a JDBC driver on the system that can allow the JSP pages to communicate with the database. Luckily, PostgreSQL offers the JDBC driver. We connect to the database within the JSP as follows:

```
<%
try
   {
      Class.forName("org.postgresql.Driver");
      Connection conn = DriverManager.getConnection(
            "jdbc:postgresql:mailn",
            "postgres",
            ""
      );
      String query = "select * from login where eid = '"+id+"'
and password = '"+pwd+"'";
            Statement stmt = conn.createStatement();
            ResultSet rset = stmt.executeQuery(query);

}
   catch(SQLException e)
   {
      out.println("SQLException: " + e.getMessage() + "<BR>");
      while((e = e.getNextException()) != null)
         out.println(e.getMessage() + "<BR>");
   }
   catch(ClassNotFoundException e)
   {
 out.println("ClassNotFoundException: " + e.getMessage() +
"<BR>");
   }
%>
```

Depending upon what action the JSP is meant to perform, the call to the database changes from executeQuery to executeUpdate. Retrieving data requires a method of executeQuery, with a string parameter of the SQL query. executeUpdate, on the other hand, also takes a string parameter, but this string is most likely an update or delete statement. Notice all JSP code is enclosed in <% %> tags.

The Fetch class is responsible for downloading all mail into the database. It is built around Javamail.

```
import java.io.*;
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;

public class Fetch {
  public static void main (String args[])
      throws Exception {
    String host = "mailhost";

    // Get system properties
    Properties props = System.getProperties();
    props.put("mail.imap.host", host);

    // Setup authentication, get session
    Authenticator auth = new ourOwnAuthenticator("username", "password");
    Session session =
      Session.getDefaultInstance(props, auth);

    // Get the store
    Store store = session.getStore("imap");
    store.connect();

    // Get folder
    Folder folder = store.getFolder("INBOX");
    folder.open(Folder.READ_ONLY);

    // Get directory
    Message message[] = folder.getMessages();
    for (int i=0, n=message.length; i<n; i++) {
      System.out.println(i + ": "
        + message[i].getFrom()[0]
        + "\t" + message[i].getSubject());
      String content = message[i].getContent().toString();
      if (content.length() > 200) {
        content = content.substring(0, 200);
      }
      System.out.print(content)
    }

    // Close connection
    folder.close(false);
    store.close();
```

```
    System.exit(0);
  }
}
```

The Send class differs greatly from Fetch. Send is invoked as a servlet from compose.jsp. Compose.jsp holds a form with the fields of an email message. Send takes these fields and builds a message, which gets sent through JavaMail.

```
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class send extends HttpServlet {

   public void doPost(HttpServletRequest request,
                      HttpServletResponse response)
      throws IOException, ServletException
   {
      PrintWriter writer = response.getWriter();
      response.setContentType("text/html");
      writer.println("<body bgcolor=\"white\">");
      writer.println("<h1>Mail Example</h1>");

      ///////// set this variable to be your SMTP host

          String SMTP_HOST_NAME = "mail.ourhost.com";
          String username = "username";
          String password = "password";
          //get the fields from the html form
     String to = request.getParameter("to"); //modified Jd
          String cc = request.getParameter("cc");
          String bcc = request.getParameter("bcc");
    String subject = request.getParameter("subject");
    String body = request.getParameter("body");
      try {
          //Get system properties
```

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

        //Specify the desired SMTP server
           props.put("mail.smtp.auth", "true");

   Session session = Session.getDefaultInstance(props, null);

// create a new MimeMessage object (using the Session created above)
        Message message = new MimeMessage(session);
        message.setFrom(new InternetAddress(from));
message.setRecipients(Message.RecipientType.TO, new InternetAddress[] { new
InternetAddress(to) });
message.setRecipients(Message.RecipientType.CC, new InternetAddress[] { new
InternetAddress(cc) });
message.setRecipients(Message.RecipientType.BCC, new InternetAddress[] { new
InternetAddress(bcc) });
message.setSubject(subject);
message.setContent(body, "text/plain");
     Transport tr = session.getTransport("smtp");
     tr.connect(SMTP_HOST_NAME, username, password);
     message.saveChanges(); // don't forget this
     tr.sendMessage(message, message.getAllRecipients());
        // it worked!
        writer.println("<b>Thank you.  Your message to " + to + cc  + " Test "+ " was
successfully sent.</b>");
      } catch (Throwable t) {
        writer.println("<b>Unable to send message: <br><pre>");
        t.printStackTrace(writer);
        writer.println("</pre></b>");
      }

      writer.println("</body>");
      writer.println("</html>");
   }
```

Test Model

As the logic and routines are fleshed out in the coding stage of (mail)^n, unit testing will be conducted, using the white-box testing method. Since the functions that make up the whole of the system are separate components, we want to test at the component level, making sure there are no serious design flaws. Errors found at this stage are easily corrected.

To test each component function, data submitted through the HTML interface will be validated to ensure improper data, or malicious code does not make its way to the main processing functions. When data is then passed to the functions, code will manipulate the data, then it will be passed to the database, or out to the internet, depending on the function handling the data. If the data should leave a particular function and be passed to a separate, undeveloped function, stubs will be in place to stand in for the undeveloped functions.

At this stage, it is essential for multiple test cases to be passed to the functions. It is impossible to test every possible case, but we will employee a broad range of test cases, making sure that most bases are covered. The object is to uncover logic flaws, typographical errors, and misinterpretation of the function specifications.

(mail)^n is a web-based email system. As such, it exists on the internet, and can be subjected to attacks by unwanted individuals. For this reason, it is necessary to conduct Security Testing designed to keep these intruders at bay. Types of security testing will already be in use as the system is unit tested, but these tests will continue as the software is beta tested. In particular, members of the test team will be given the task of trying to crack into the system using brute-force tactics.

When a user sets a password, the entered password will be checked against a list of the current most common passwords. If the entered password matches on of these, it will be rejected. Security checks such as this will be in place as the system is integrated.

The coding of individual pages of (mail)^n is designed to allow entrance to the system only through the front door, which is the login page. Testers will try to gain entrance by creating a bookmark of their inbox, but if the ‘one way in, one way out’ design holds, they should not be able to view anything without having first logged in.

Components to be tested

The major components to be tested that are not part of the user interface include the mail server, the web server, and the database server.

Mail server testing

Test case: message is sent to the mail server
Expected response: message is saved to the database to the correct recipient

Test case: message is sent to the mail server, CC'ed to multiple $(mail)^n$ users
Expected response: a copy of the message is received in each of the addressed users

Test case: user sends message
Expected response: mail server sends the message over the internet

Test case: user sends message to multiple recipients
Expected response: mail server sends a copy to each of the recipients

Web server testing

Test case: web page is requested
Expected response: web page is served

Test case: communication with database server
Expected response: response is received from database

Database server testing

Test case: select statement
Expected response: requested records are returned

Test case: insert statement
Expected response: new records are created

Test case: update statement
Expected response: records added or edited are updated in the database

Test case: delete statement
Expected response: database deletes records

Test case: power failure or crash
Expected response: on restart, database is restored