(mail)^n

Elaboration Three

CS616
Fall 2004
Dr. Marchese
December 22, 2004

Prepared by:

Elias Rosero

Jwalant Dholakia
Joseph Aulisi

Vision

Revision History

Version	Date	Description	Author(s)
Final Inception Document	December 22, 04	Final Draft. No major modifications from the first draft submitted on October 27, 04 have been identified.	Joseph Aulisi Elias Rosero Jwalant Dholakia

Introduction

Our team sees (mail)^n as an email program designed to house all the email received by an organization and its users in a form that is easily searchable, expansive, and permanent.

Positioning

Business Opportunity

Organizations today have a wide variety of email systems from which to choose, though none of them guarantee against data loss, and many are prohibitively expensive to employ and maintain. Although many systems offer users the ability to view their email via a client program installed on the user's machine, or via a web-based client, email itself is stored in proprietary data files, or plain text files, which can grow quite large, thus becoming unwieldy, and can have an adverse effect on the speed at which mail is retrieved and read.

Problem Statement

Email systems today are not secure and can potentially lose the data they were designed to store. Many systems that store email messages in a proprietary format keep mail in a single file, which, over time, is at a high risk of corruption. When this corruption occurs, users can expect to lose all of their messages. Some systems, due to their omnipresence, are often exploited by malicious virus attacks, which can compromise a client machine and propagate around the world in a matter of minutes, leaving a path of infected computers with no chance of recovering data.

Product Position Statement

(mail)^n is designed for use by small to mid-sized organizations that are looking for an email system that provides its users with the ability to view their email securely from any web browser. Each user's email is stored in a database, which offers a virtually unlimited mailbox size for each user, and offers the ability to quickly locate past emails. Apart from being a rock-solid email program, (mail)^n offers a convenient contact management tool, and a task database, which allows corporate users the ability to schedule inter-office meetings.

Alternatives and Competition

Currently, there are many alternatives to (mail)^n, including Microsoft Exchange, Lotus Notes, and even outsourced email services. One of the problems with outsourcing email

is the mailbox size limit that vendors place on services. (mail)^n offers an unlimited mailbox size for all users on the system. Microsoft Exchange and Lotus Notes are pricey alternatives, requiring the purchase of hardware and software, but also the presence of someone with the skills to administer these services.

Stakeholder Descriptions

Stakeholder (Non-User) Summary
Major stakeholders will be business owners in a position to decide the best, most efficient and cost-effective way to manage mail. Those who decide on which mail system to roll out are the ones who stand to lose the most, should the system fail. (mail)^n offers a worry-free solution.

User Summary
Two types of users, administrators and employees, will be using (mail)^n. Employees are standard users with no administrative privileges. Administrators, on the other hand, have more privileges, and are responsible for adding employees to the system, among other tasks.

Key High-Level Goals and Problems of the Stakeholders

High-Level Goal	Priority	Problems and Concerns	Current Solution
World-wide accessibility	High	Will the system be available when users travel?	Most solutions offer a web access to email.
Unlimited storage	High	Will the system grow with the company?	Online services offer a service with a hard limit on disk quota.
Security	High	Will the system be resistant to viruses, worms, and prying eyes?	Bugs in current systems can allow for such attacks.
Ease of use	High	Will there be a need for training?	Most email systems are easy to use, but can become bloated with too many features.

User-Level Goals
–Employee: send, receive, and save emails, store contact information, set up tasks.
–Administrator: backup mail system, set up users easily.

User Environment
Employees and administrators will be able to access (mail)^n with any modern web browser, from any computer connected to the environment.

Product Overview

Product Perspective
(mail)^n will reside on a robust computer system housed within the walls of company, or within a co-location service. It will provide services to all the users of an organization,

and will need to collaborate with an external mail service, and a database, as shown in figure Vision-1.0.

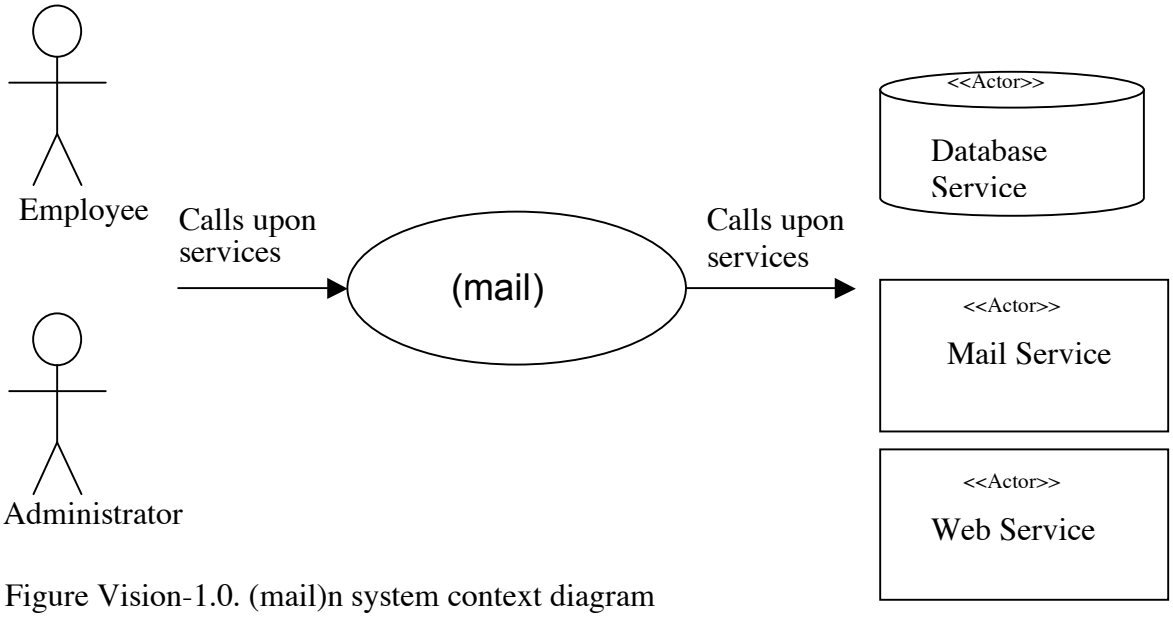


Figure Vision-1.0. (mail)n system context diagram

Summary of Benefits

Because $(mail)^n$ is web-based, the system is accessible from anywhere in the world. Users are able to login and access all the information they need. An employee will be able to save every correspondence they receive and send while working at an organization. Since the system is regularly backed up, there is no need to fear loss of emails. $(mail)^n$ is not prone to virus attacks, so the data remains safe. People using $(mail)^n$ will already be familiar with point navigating web sites, so there is no need to train them to use this system.

Summary of System Features

- all data is stored in a database, which allows for fast searches, and eliminates the possibility of email file corruption
- there is no need to delete emails
- users can store all contact information in the system
- users can set up meetings by sending an email to fellow employees. If fellow employees agree to meet at the specified date and time, the tasks for the agreeing employees are updated to reflect this appointment
- the contact module easily interfaces with mail module, which makes addressing emails easy

Other Requirements and Constraints

For an in-depth look at the design constraints, functionality, usability, reliability and performance, see the Supplementary Specification and Use Case models.

Use-Case Model

The use-case model used in this document to describe $(mail)^n$ is accomplished using the 'fully dressed' format. Given the nature of the system, it is empirical to enumerate as many details as possible about the system for completeness.

Before presenting the use-case scenarios, we must illustrate the relations between actors and their goals. Namely:

Actors	**Goals**
Web-server	Inter-median between the database and the mail server and the user. It also provides an interface for the user.
Mail-server	This is where the emails are stored at first. They are held there until they are read and processed into the DB
Database	Stores mail messages, contact information, and task information. Also returns information when queried.
User	Must be able to send, receive, read, and store emails. Must also be able to manage contact groups and tasks.
Administrator	To create, or delete an account within the system.

Having identified the set of 'actors', and their respective goals, we continue to map out the first use case scenario, in which the primary actor is a user.

Use Case UC1: Processing Messages

Primary Actor: User
Stakeholders and Interests:
- User: Wants to navigate through the email system easily and be able to send, receive and store emails. In addition, the user also wants the ability to manage contact groups, and edit tasks.
- Administrator: Wants to be able to create, modify or delete accounts.
- Company: (given that $(mail)^n$ is used within a corporation, although it could be used by anyone.) Wants to ensure that all of the employees using the system can easily do so without the need of someone to assist the employees. Also, the system has to be virus resistant, because employees will rely on having the ability to store information. The information must be guarded for safe retrieval.

Preconditions: User is connected online, and is using a modern web browser.
Success Guarantee (Post conditions): Emails are sent, received and stored. Tasks are correctly managed, and contact information is correctly updated. Administrators have full control over the user accounts.

Main Success Scenario (or Basic Flow):

1. User connects to the Internet.
2. User goes to the (mail)^{n} web site.
3. User enters his/her user name and password.
4. The system verifies that both user name and password match with the contents in the database.
5. The user is now logged into the system.
6. The first screen that the user sees is the 'inbox' screen.
7. From that point, the user may click on a particular unread message to view its content, or choose to perform any other available option.
8. Assuming that the user reads an unread message, and wishes to reply, either to the sender, or to the sender and the CC's, the user clicks on the "Reply" button.
9. A new screen appears in which the user may enter the recipient information, a subject line, and the body of the email.
10. After writing the email, and filling out the "To:" field, the message is ready to be sent.
11. The user clicks on the "Send" button.
12. The user is then presented with a "Message successfully sent" message upon completion of the sent action.
13. The user is redirected to the inbox screen, where he/she may continue to browse emails.
14. The user chooses to terminate his/her session.
15. The user successfully logs out.

Extensions (or Alternative Flows):

At any time, System fails:

To support recovery and correct storing of emails, ensure that all messages are stored into the database once the messages are read by the user. Emails will not be lost due to a local machine failure.

1. User restarts machine, connects to the internet, logs in, and emails are still present.
2. System keeps messages in external machines, not available to the user, so the messages are kept safe.
3. Invalid identifier:
    1. System does not match the string of username, and/or password with existing usernames and/or passwords in the database.
    2. System rejects entry, and redirects user to the initial screen where the user may attempt to log in once again.
4. Internet connection is lost:
    1. When reconnected, the user must enter his/her username and password again.
    2. Loosing Internet connection is the equivalent to logging out.
5. System detects failure to connect to mail server:
    1. An error message will display, stating that the mail server is temporarily out of service.

6a. The user does not fill out required fields of a message:

    1. At least one valid recipient's address must be provided to send a message.

6b. The user enters an incorrect email address in the recipient's box:
    1. The system will email the user shortly after the message has been sent, notifying the user that the email address does not exist.
7. The user wants to send an email attachment larger than 10MB:
    1. The system will display an error message stating that the attachment exceeds 10MB (the limit size of attachment per email).
8. The administrator wants to enter a new user's information:
    1. The new username must not exceed 12 alphanumeric characters.
    2. The new password must not exceed 8 alphanumeric characters.
9. The user wants to add a new task:
    1. The user may not add a new task that takes place earlier than the present time at which the user wants to add the new task.
    2. An error message is presented stating that the task must be set to a future date.

Special Requirements:
- The front end of $(mail)^n$ requires a modern web browser that understands Dynamic HTML and Cascading Style Sheets.
- Users are required to have an Internet connection.
- User-friendly front end.

Frequency of Occurrence: The system could be used as frequently as needed.

Open Issues:
- What can be customized for using the system within corporations versus using it publicly?

This particular fully dressed Use Case pinpoints the various 'paths' that a given user may take when using $(mail)^n$. It illustrates the 'happy path' as well as all of the areas in which the system may display an error message due to an improper use. It is explicit in pointing out the system's limitations, as well as the user's limitations. This information is essential that is needed to have a formal set of system constraints.

Supplementary Specification

Revision History

Version	Date	Description	Author(s)
Final Inception Document	December 22, 04	Final Draft. No major modifications from the first draft submitted on October 27, 04 have been identified.	Joseph Aulisi Elias Rosero Jwalant Dholakia

Introduction

This section documents various (mail)^n requirements and specifics not captured in the use cases.

Functionality

Logging and Error Handling
Make a log of all identified errors for future review.

Pluggable Business Rules
Depending upon the requirement of client Company some minor modifications in the look and feel of the system (like changing menu system, color, adding company logo etc) can be added.

Security
(mail)^n is a web-based e-mail system. So, in order to ensure that only company employees who have the right to use the system are using it, and no one else, user authentication is of prime importance.

Usability

Human Factors

- (mail)^n is going to be used in an office environment by employees and company administrators. The employees might end up spending a good deal of time using (mail)^n in any given day. So, it is extremely important that (mail)^n is user friendly.
- The text size should be large enough so that it does not get difficult and cumbersome to read messages after an entire day in front of the computer screen.
- The employee should be given an audible signal when a new mail arrives because he might not be constantly monitoring the (mail)^n browser to check for new mails (Proposed).
- It is also important that the new incoming messages do not take too much time to open up as this might increase the frustration and stress on part of the employees using (mail)^n.

Reliability

Recoverability
If the server or internet connection is down, an employee might not be able to access his mails on $(mail)^n$. This might be a bottleneck for the system but it does not seem to be a critical issue. No further decisions are made about it as of now. However, if the system goes down because of some internal error (like database server crash, invalid query etc) the system should be set up in such a way that it recovers as much unsaved data as possible and restarts itself.

Performance

A consistent and efficient performance of $(mail)^n$ is important. But after all, $(mail)^n$ is a web-based application involving a Wide Area Network. And sometimes this network might go down and bring the application to a halt. Factors like these create a bottleneck on the performance of $(mail)^n$. Further performance issues will be identified once the system is up and running and in production use.

Supportability

Adaptability
$(mail)^n$ can be set up to be used by an administrator or an employee. Depending upon the user type certain privileges (like adding new users, deleting users etc) are allowed.

Configurability
$(mail)^n$ has the functionality to classify incoming messages into various sub-folders depending upon desire of the system user. $(mail)^n$ is definitely configurable in making e-mails organized and systematically classified.

Implementation Constraints

$(mail)^n$ is developed using Java, and PostgreSQL as well as other components. The main reason for this was ease of use and ability to modify code efficiently in the future if needed.

Purchased Components

We were able to set up the server on a test basis using existing hardware. No purchased components are identified as of now.

Free Open Source Components

(mail)^n makes heavy use of open source components. PostgreSQL is an open source database. The pages of (mail)^n are served with Tomcat, an open source Java environment from the Apache Software Foundation. The heart of the send and receive functions of the system are built with JavaMail, which is free to use.

Interfaces

Noteworthy Hardware and Interfaces

- Server
- Internet Connection Hardware

Software Interfaces

- Web Browser

Domain Rules

ID	Rule	Changeability	Source
RULE 1	User Authentication required before showing (mail)^n mails page	Low	(mail)^n policy
RULE 2	Idle connections for more than 15 minutes automatically logged out of (mail)^n	Depending upon business rules of a company for which (mail)^n is set up, modification can be made into the system to disable this feature	Company policy
RULE 3	Once browser window is closed, the user is automatically logged out of the system. Re-authentication required to login again	To enhance security of (mail)^n we keep changeability of this rule to Low	(mail)^n policy

Legal issues

Making sure all terms and conditions specified in the open source software user-agreement are satisfied.
Strictly enforce (mail)^n user agreement upon client companies.

Information in Domains of Interest

Network connectivity

If the Local Area Network on the client company site goes down, all local machines on the company network might not be able to access mail using $(mail)^n$ even if the Internet Network and $(mail)^n$ server are up and running. In this case, the company might be willing to invest in a stand-alone computer that is directly connected to an outside network to access important messages.

Glossary

Revision History

Version	Date	Description	Author(s)
Final Inception Document	December 22, 04	Final Draft. No major modifications from the first draft submitted on October 27, 04 have been identified.	Joseph Aulisi Elias Rosero Jwalant Dholakia

Term	Definition and Information	Aliases
(mail)^{n}	Email system where all message information is saved to an object relational database, allowing unlimited mailbox size, and a secure, easy to use interface	
Message	An individual email sent or received by a user of the system	Mail, email
Inbox	A virtual repository of new, unsorted messages	
Sent folder	A virtual repository recording messages that have already been emailed from the user's account	
Deleted folder	A virtual repository of messages a user has removed from the inbox, but is not ready to permanently remove from the system	
Folder	A user-defined repository of filed messages	
Log in	The action of supplying a id and password in order to gain access to (mail)^{n}	
Log out	The action of leaving (mail)^{n} securely.	
Session	The time starting when the user logs in until the user logs out	
Module	A distinct feature of (mail)^{n}, including the mail module, the contacts module, and the tasks module	
Contact	Any entry into the contacts database that includes information about a person, such as name, street address, business, email address, and telephone number	
Task	Any entry into the tasks database that describes an event, or appointment with the attributes of date and time	Event, appointment, meeting
Search	An action of iterating through the database to find a particular piece of information	
Username	A unique identifier for each user of (mail)^{n}	Id, employee id, uid, e-id, eid
Employee	Any identifiable user of (mail)^{n}	
Administrator	An employee of an organization that has extended privileges on the system, such as user creation, password assignment, and user deletion	admin
Database	A central repository of all email messages, contacts, and tasks	

Term	Definition and Information	Aliases
To	The addressee of a new mail message	recipient
From	The original sender of a mail message	
CC	Additional recipients of a mail message	
BCC	Blind carbon copy, which conceals recipients of a message from other recipients	
Flag	An attribute of a mail message	
Attachment	A separate file that is part of a mail message	
Forward	The action of sending a received message to another party	
Reply	The action of answering a received message with a response the to the sender	
Reply All	The action of answering a received message with a response to the sender and all parties to which the message was addressed	
Flagged	An attribute of a message that has been distinctly marked, as to alert the user that this message needs additional attention	
Body	The part of an email that carries the actual semantic message	
Subject	The part of an email that contains a summary of the body of the message	
Timestamp	The date and time a message was sent or received	
Read	(pronounced reed) The action of opening an email message and reading its contents	
Read	(pronounced red) The state of a message after it has been opened and scrutinized	
Unread	The state of a message before it has been opened	
Mail-id	A unique identifier for each mail message	mailid
Group	A user-defined list of intended recipients of messages	
Invitation	A message sent to a group of employees asking them to attend a meeting	Envite, evite, invite, nvite
Public	A contact that is visible to all users of (mail)^{n}	
Private	A contact that is visible to only the user that created the contact	
Category	An attribute of the Contacts table that records if a contact is public or private	

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
- User Groups
- (mail)^n Database

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

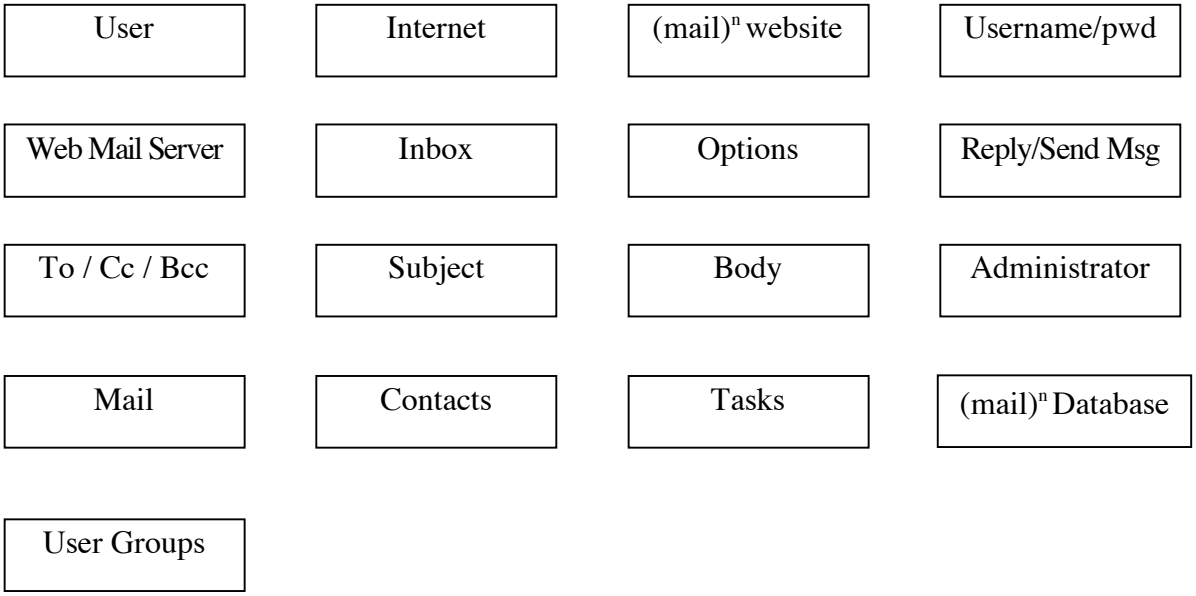


Figure 1.0: Conceptual Classes

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.

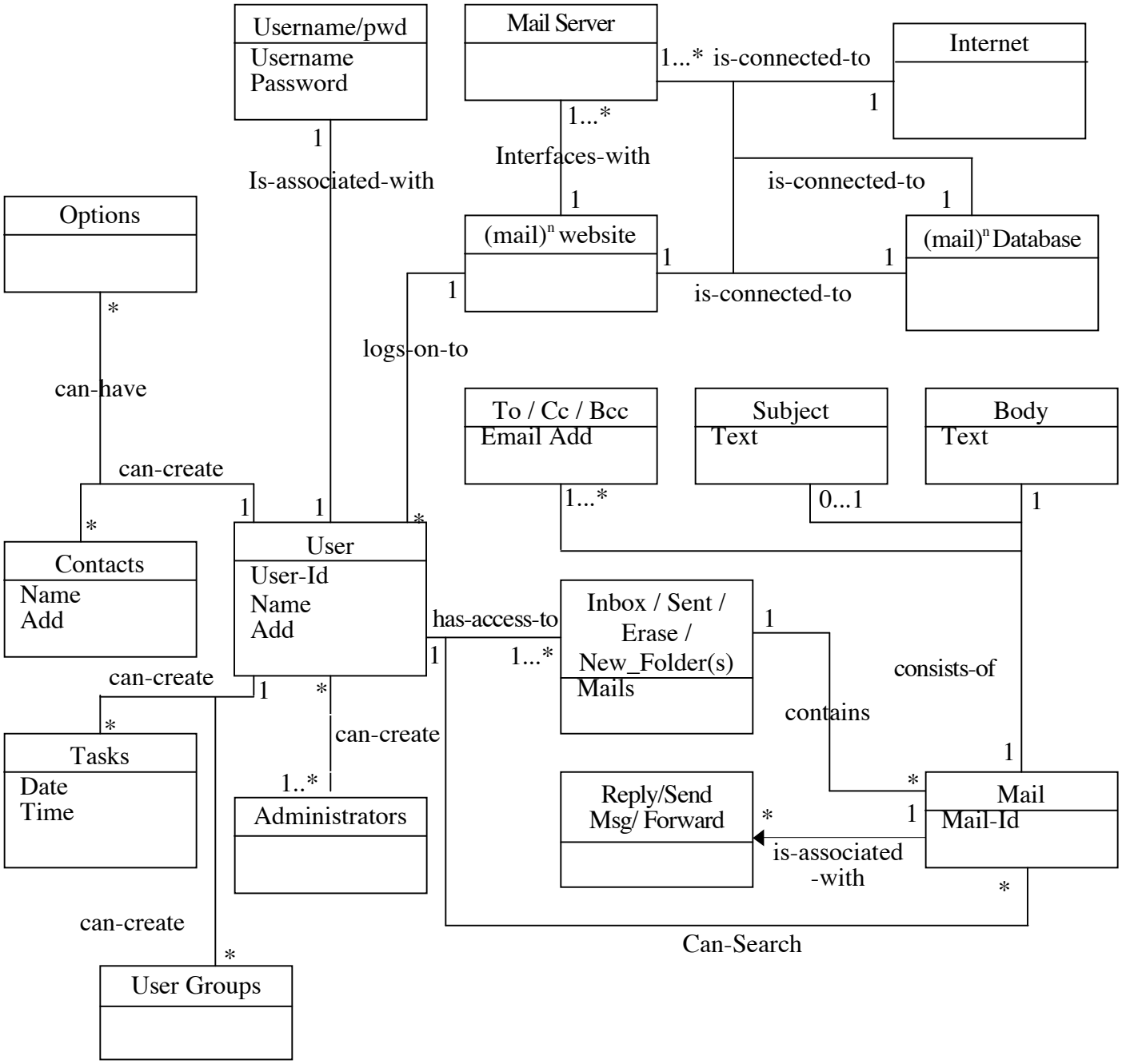


Figure 2.0 Refined Domain Model

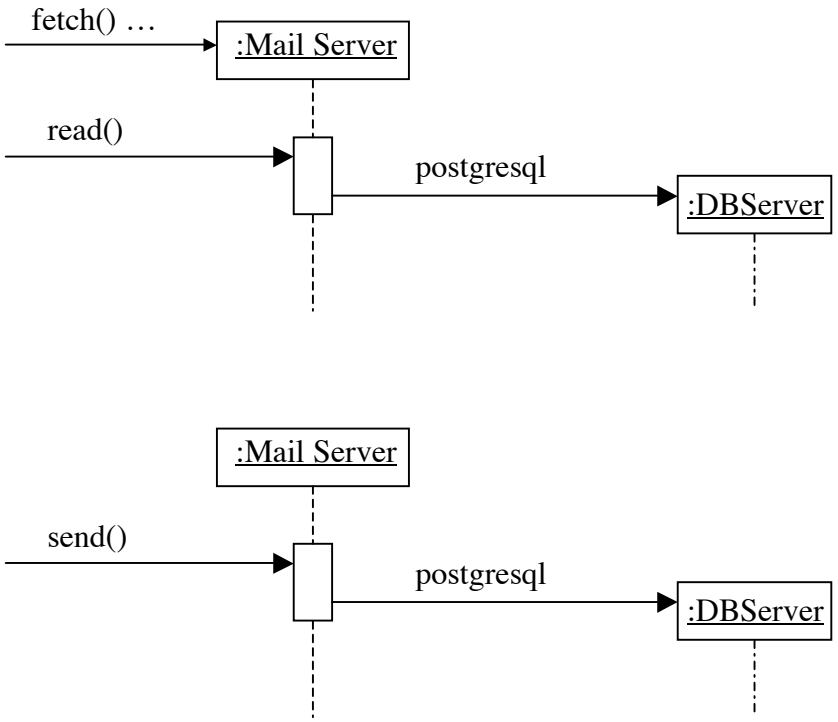
Design Model: Use-Case Realizations

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. Originally, Fetch was designed to iterate through all ids in the login table, using the id and password, make calls to the mail server and download all the mail for each user. While testing, this proved to be an untenable method, so Fetch evolved into GrabMail, which is a new servlet class that is run every two minutes once a user logs into $(mail)^n$. 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

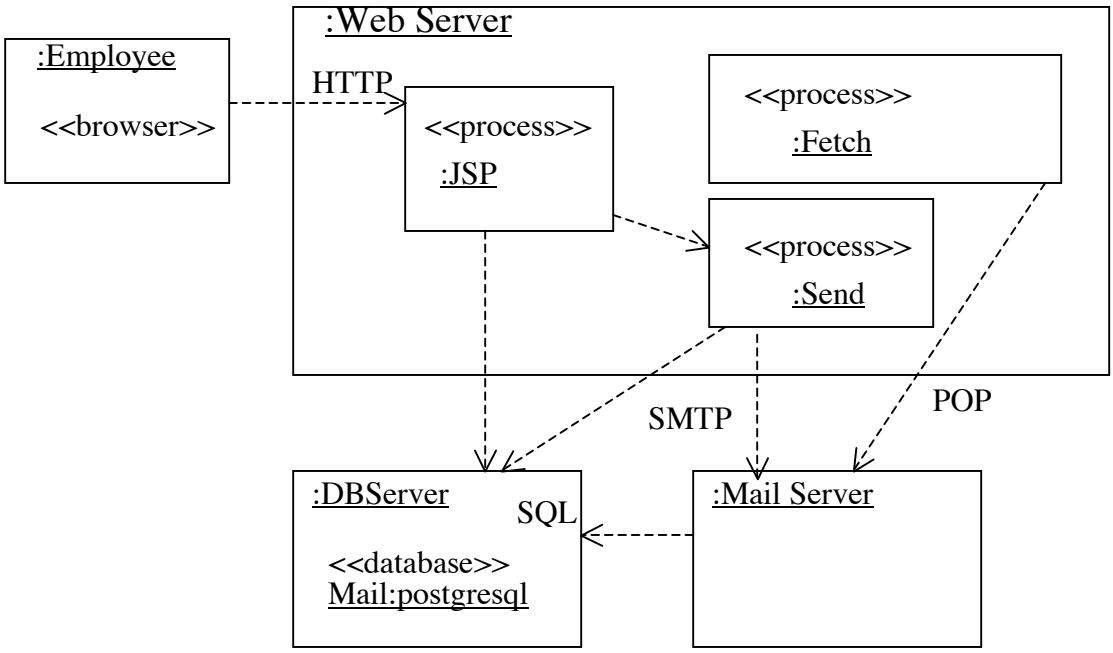
When a new message appears on the mail server, JavaMail takes the steps necessary to communicate with the remote mail server, supplying it with the proper user name and password. Once the connection is made, each part of the message is easily retrieved by JavaMail methods, finally resulting in an insert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

The class responsible for downloading mail, GrabMail, must be run often, to ensure that email reaches the users in a timely manner. To achieve this, it will be necessary to refresh the page that calls the GrabMail servlet every two minutes. 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 GrabMail 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 GrabMail 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.

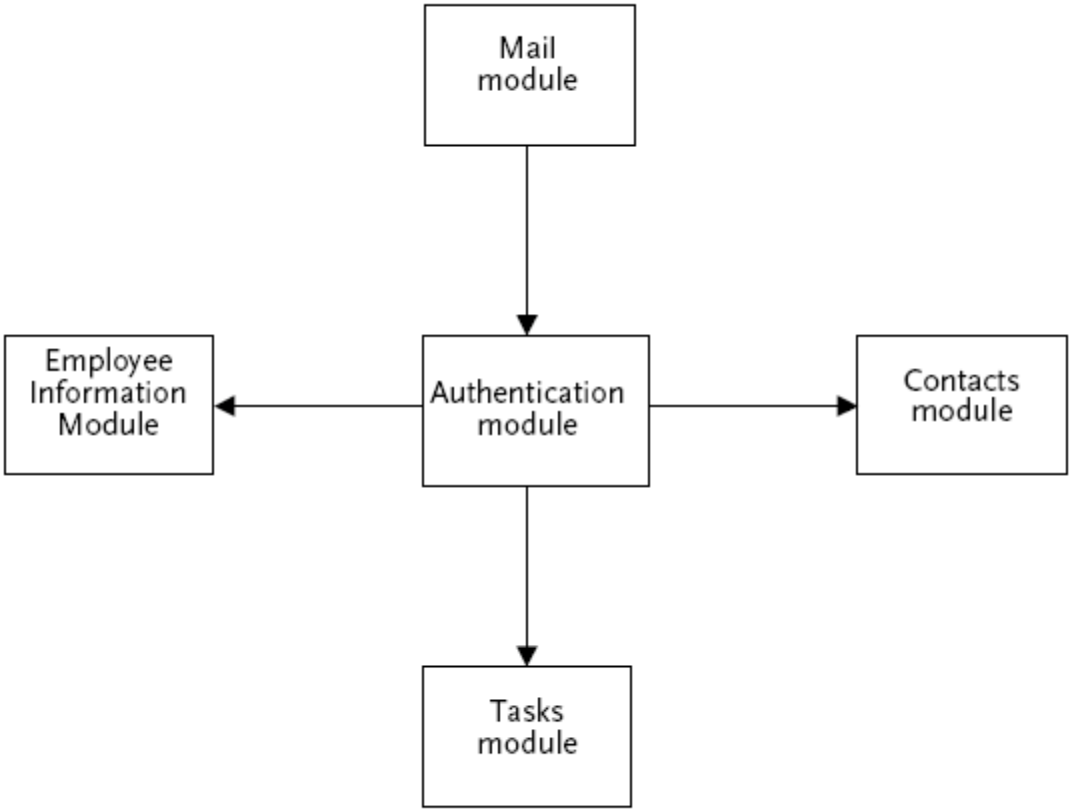
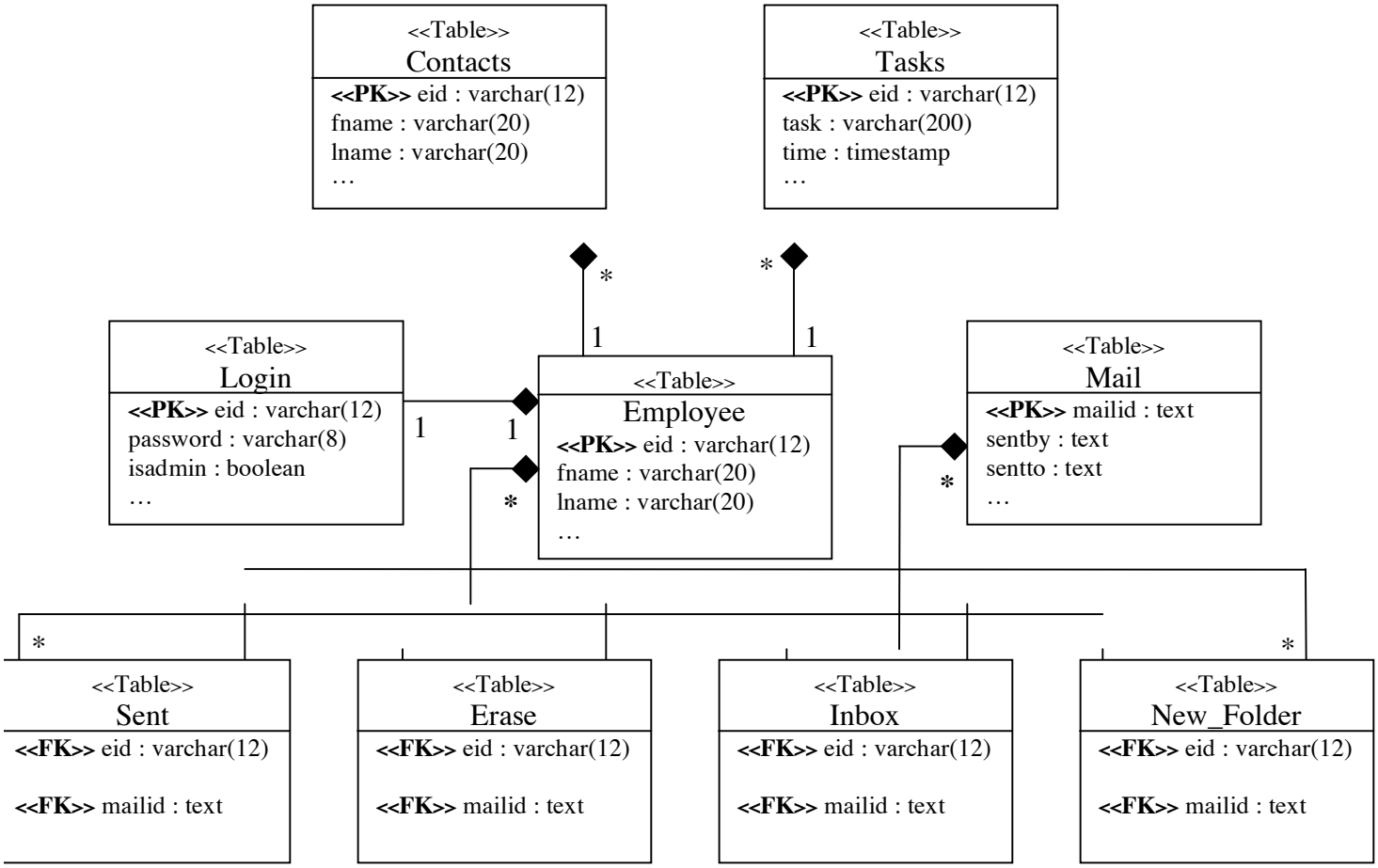


Fig: Architecture Model


<<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

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

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>");
   }
```

Use-Case Storyboards, UI Prototypes

Updated SSD diagrams for *Processing Message* use case of (mail)^n have been described here. Also, updated versions of SSDs the describe the rest of the functionalities of (mail)^n including employee options and administrator options are described.

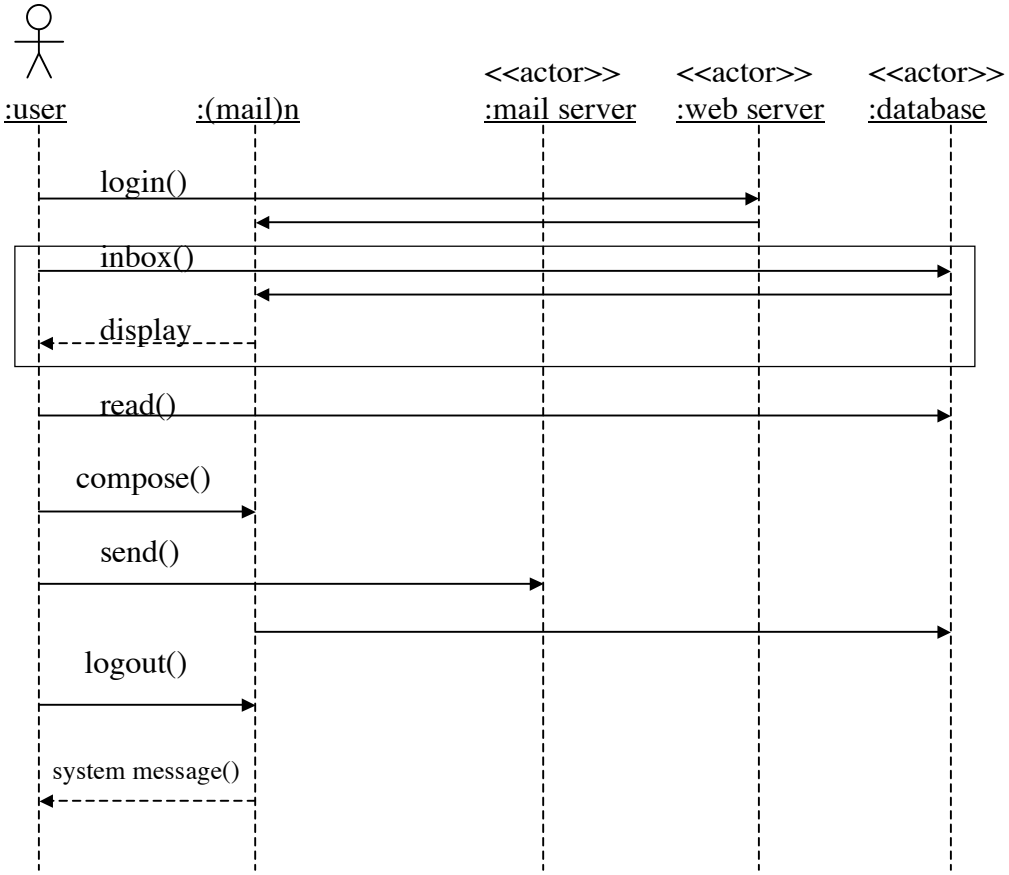


Figure 1.0 SSD of Processing Messages

In figure 1.0 we observe an SSD that graphically represents what the use-case scenario presented as text in the earlier stages of development described. The graph contains the user, the system (mail)^n, and the three actors: web server, mail server, and database. The actions that stem from the user are represented with vectors that expand to the appropriate 'instance'. For example, when the user wants to send an email message, upon a click of the mouse or a keyboard entry, first the action is processed through the web server, and then the system saves a copy of the message in the database. The first action is shown with a vector from the user to the web server, illustrating that the user has requested to send a message. Directly below, there is another vector originating from the system (mail)^n pointing to the database to illustrate that the system saves a copy.

It is necessary to recall that SSDs are drawn with a chronological order downward. Which means that the actions are sequential. It is also important to clarify that the actions that are contained within the box are those actions that are iterative.

The second SSD is a graphical representation of the use-case "Employee Options". Although the SSDs follow a chronological order, in this case the actions taken by the user/employee are specified in a random order. The options listed are accessible to the user at any point within the system. The actions are therefore listed without an order.

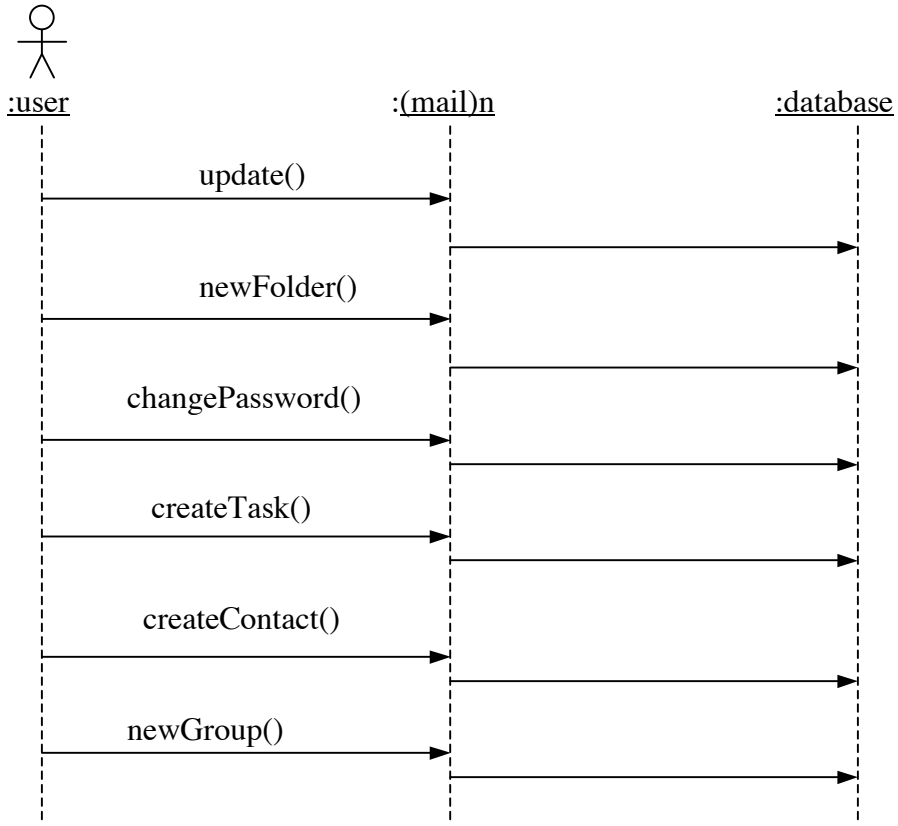


Figure 1.1 SSD of Employee Options

In figure 1.1 we have and SSD that shows the various options that an employee has within the system. There are no responses made to the user that are necessary to point out, because the updates are made to the database. For example, if a user wants to create a new folder to organize his contacts into groups, then the user clicks on the link for creating new folders and the system automatically prompts for the name of the new folder. Once the user clicks on OK, the request is performed in the database.

The following SSD represents the use-case designed for the Administrator's Options. As stated in early documentation, the administrator has privileges that a normal does not have. For example, an administrator has the power to create, deleted or modify accounts. The initial screen is different for the administrators.

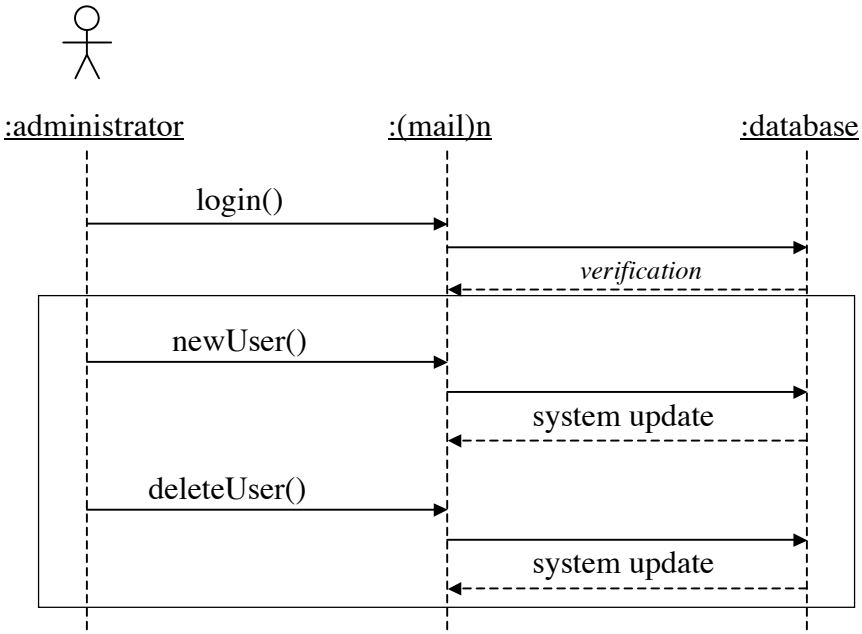


Figure 1.2 SSD of Administrator Options

Upon verifying that an administrator has logged on to the system, the administrator may add or delete users. In figure 1.2 the administrator's options are represented in the diagram. An administrator, when adding a user to the system, is basically updating the *employee table* as well as the *login table*.

As per any of the SSDs in this section, they may be further clarified by reading the *Use case system contracts*. Within the contracts, the tables that are updated upon an action are explicitly mentioned.

Use Case System Operations Contracts

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 the mail transfer agent was made - the message was saved to the mail database, and an entry was made in the sent table

Contract C2: fetch

Operation:	fetch(message-id, e-id, recipients, authenticator)
Cross References:	Use Case UC1: Processing Messages
Pre-Conditions:	Messages exist on the mail server.
Post-Conditions:	- an authentication was made associated with creating a session - an connection with IMAP was made - the message was stored in the mail table - an entry was made in the inbox table - an associated entry was made in the flag table - the read flag in the flag table was marked as false - the session was closed

Contract C3: read

Operation:	read()
Cross References:	Use Case UC1: Processing Messages
Pre-Conditions:	There is a message to read.
Post-Conditions:	- the read field of the flag table was marked as true

Contract C4: reply

Operation:	reply(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. There is a message that has been received.
Post-Conditions:	- a unique message id was created - the message id was associated with the current e-id - a connection with the mail transfer agent was made - the message was saved to the mail database, and an entry was made in the sent table - the associated entry in the flag table for the original message was updated with the reply attribute set to true

Contract C5: forward

Operation:	forward(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. There is a message that has been received.
Post-Conditions:	- a unique message id was created - the message id was associated with the current e-id - a connection with the mail transfer agent was made - the message was saved to the mail database, and an entry was made in the sent table - the associated entry in the flag table for the original message was updated with the forward attribute set to true

Contract C6: delete

Operation:	delete(message-id, e-id)
Cross References:	Use Case UC1: Processing Messages
Pre-Conditions:	There is a message to be deleted.
Post-Conditions:	- the message-id and e-id were stored in the erase table - the associated entry was removed from the inbox table

Contract C7: empty

Operation:	empty(message-id, e-id)
Cross References:	Use Case UC1: Processing Messages
Pre-Conditions:	There is a message to be deleted.
Post-Conditions:	- the message-id and e-id were first removed from the erase table due to table constraints - the associated entry was removed from the flag table, due to table constraints - the entry was then removed from the mail table

Contract C8: move

Operation:	move(message-id, e-id)
Cross References:	Use Case UC1: Processing Messages
Pre-Conditions:	There is a message to be moved.
Post-Conditions:	- the message-id and e-id were stored in the new_folder table - the associated entry was removed from the inbox table

Contract C9: update

Operation:	update(form_data)
Cross References:	Use Case UC2: Employee Options
Pre-Conditions:	An individual has a valid account.
Post-Conditions:	- the employee table was updated with the correct data - the updated information was associated with the current e-id

Contract C10: newFolder

Operation:	newFolder(folder_name)
Cross References:	Use Case UC2: Employee Options
Pre-Conditions:	An individual has a valid account.
Post-Conditions:	- the employee table was updated with the correct data - the updated information was associated with the current e-id

Contract C11: changePassword

Operation:	changePassword(password)
Cross References:	Use Case UC2: Employee Options
Pre-Conditions:	An individual has a valid account.
Post-Conditions:	- the login table was updated with the correct data - the updated information was associated with the current e-id

Contract C12: createTask

Operation:	createTask(date, time)
Cross References:	Use Case UC2: Employee Options
Pre-Conditions:	An individual has a valid account.
Post-Conditions:	- an entry was made in the tasks table with the date and time parameters - the entered information was associated with the current e-id

Contract C13: createContact

Operation:	createContact(form_data)
Cross References:	Use Case UC2: Employee Options
Pre-Conditions:	An individual has a valid account.
Post-Conditions:	- a new entry was made in the contacts table - the entered information was associated with the current e-id - a field was set denoting the entered contact as public or private

Contract C14: newGroup

Operation:	newGroup(group_name, group_members)
Cross References:	Use Case UC2: Employee Options
Pre-Conditions:	An individual has a valid account.
Post-Conditions:	- an entry was made in the groups table - the entry was associated with the e-id of the person creating the group

Contract C15: newUser

Operation:	newUser(e-id, password)
Cross References:	Use Case UC3: Admin Options
Pre-Conditions:	An individual has a valid administrative account.
Post-Conditions:	- a new entry was made in the login table - this entry was associated with a new entry in the employee table, which was also entered at this time

Contract C16: deleteUser

Operation:	deleteUser(e-id)
Cross References:	Use Case UC3: Admin Options
Pre-Conditions:	An individual has a valid administrative account.
Post-Conditions:	- the entry was removed from the login table - the associated record was removed from the employee table

Contract C17: staffMeeting

Operation:	staffMeeting(date, time)
Cross References:	Use Case UC3: Admin Options
Pre-Conditions:	An individual has a valid administrative account.
Post-Conditions:	- an entry was made in the tasks table for each employee

Contract C18: publicGroup

Operation:	publicGroup(group_name, group_members)
Cross References:	Use Case UC3: Admin Options
Pre-Conditions:	An individual has a valid administrative account.
Post-Conditions:	- an entry was made in the groups table - the entry was associated with all employees