



Empirical Study of Component Based Software Engineering with Aspect Oriented Programming

A Master's Thesis

by

Axel Anders Kvale

Part 2: Code From The Implementations

Supervisor: Reidar Conradi
Thesis advisor: Jingyue Li

Trondheim, 1. November 2004

Department of Computer and Information Science
Norwegian University of Science and Technology (NTNU)

1 Table of contents

1	Table of contents	1
2	The AOP implementation of the JES server	2
2.1	LogController	2
2.2	Shutdown.....	19
2.3	ExceptionPrinter.....	21
2.4	Mail	22
2.5	ShutdownService.....	24
2.6	ConfigurationManager	25
2.7	ConfigurationParameterContants	37
2.8	PasswordManager	39
2.9	ConfigurationProcessor	40
2.10	Exceptions	42
2.11	EmailAddress	43
2.12	Message.....	45
2.13	User	46
2.14	ConnectionProcessor.....	49
2.15	DeliveryService	50
2.16	DnsService	53
2.17	ServiceListener.....	54
2.18	Pop3Processor	56
2.19	SMTPMessage	65
2.20	SMTPProcessor.....	69
2.21	SMTPRemoteSender.....	75
2.22	SMTPSender	79
3	Code from replacing the logging-component.....	84
3.1	AOP.....	84
3.2	OOP.....	98
3.2.1	Mail	98
3.2.2	Pop3Processor	101
4	Code from implementation of SpamAssassin	111
4.1	AOP.....	111
4.2	OOP.....	112
5	Code from implementation of SpamProbe.....	114
5.1	AOP.....	114
5.2	OOP.....	114

2 The AOP implementation of the JES server

2.1 LogController

```
package com.ericdaugherty.mail.server.aspects;

//Log4j imports
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

//DNS imports
import org.xbill.DNS.Lookup;
import org.xbill.DNS.Record;

//Java imports
import java.io.*;
import java.net.*;
import java.lang.reflect.Method;
import java.lang.reflect.InvocationTargetException;
import java.util.*;

//JES imports
import com.ericdaugherty.mail.server.*;
import com.ericdaugherty.mail.server.errors.*;
import com.ericdaugherty.mail.server.configuration.*;
import com.ericdaugherty.mail.server.configuration.console.*;
import com.ericdaugherty.mail.server.services.general.*;
import com.ericdaugherty.mail.server.services.pop3.*;
import com.ericdaugherty.mail.server.services.smtp.*;
import com.ericdaugherty.mail.server.info.*;

import com.ericdaugherty.mail.server.aspects.Shutdown;

public privileged aspect LogController{

    //local functions for the aspect
    public Log LogInit(Class c){
        Log log = LogFactory.getLog( c.getName() );
        log.info("Logging started in " + c.getName());
        System.out.println("Logging started in " + c.getName());
        return log;
    }

    //The log interface for objects with non-static log
    public interface LogInterface {};
    public Log LogInterface.logTest;

    //The log interface for objects with static logs
    public interface StaticLogInterface{};

    /*****
    * This part places the logging objects in the appropriate classes
    *****/

    //the mail-class. This is treated separately for now...
    private static Log com.ericdaugherty.mail.server.Mail.log;

    //static (a regexp way to intertype into classes would be nice
    private static Log com.ericdaugherty.mail.server.services.general.ServiceListener.logTest;
    private static Log com.ericdaugherty.mail.server.services.pop3.Pop3Processor.logTest;
    private static Log
com.ericdaugherty.mail.server.configuration.console.ConfigurationProcessor.logTest;
    private static Log com.ericdaugherty.mail.server.info.Message.logTest;
    private static Log com.ericdaugherty.mail.server.services.smtp.SMTPMessage.logTest;
    private static Log com.ericdaugherty.mail.server.services.smtp.SMTPProcessor.logTest;
    private static Log com.ericdaugherty.mail.server.services.smtp.SMTPRemoteSender.logTest;
    private static Log com.ericdaugherty.mail.server.services.smtp.SMTPSender.logTest;

    //If we change the tree-structure of the package, this can be done in one line
    declare parents : com.ericdaugherty.mail.server.services.general.ServiceListener
implements StaticLogInterface;
    declare parents : com.ericdaugherty.mail.server.services.pop3.Pop3Processor implements
StaticLogInterface;
```

```

    declare parents :
com.ericdaugherty.mail.server.configuration.console.ConfigurationProcessor implements
StaticLogInterface;
    declare parents : com.ericdaugherty.mail.server.info.Message implements
StaticLogInterface;
    declare parents : com.ericdaugherty.mail.server.services.smtp.* implements
StaticLogInterface;

//non-static
declare parents : com.ericdaugherty.mail.server.ShutdownService implements LogInterface;
declare parents : com.ericdaugherty.mail.server.configuration.ConfigurationManager
implements LogInterface;
declare parents : com.ericdaugherty.mail.server.services.general.DeliveryService
implements LogInterface;
declare parents : com.ericdaugherty.mail.server.info.User implements LogInterface;

/*****
* This part places the logging-init functions in the appropriate classes
*****/

/**
 * This method is responsible for initializing the logging mechanism used
 * by this application. All required information is loaded from the
 * configuration file.
 */
private static void Mail.initializeLogging( String configurationDirectory ) {

    File logConfigurationFile = new File( configurationDirectory, "log.conf" );

    if( !logConfigurationFile.exists() ) {
        initializeDefaultLogging( logConfigurationFile );
    }
    else
    {
        boolean enableLog4j = false;
        try
        {
            // Get a reference to the
org.apache.log4j.PropertyConfigurator.configureAndWatch( String filename )
            // method.
            Class propertyConfigurator =
Class.forName("org.apache.log4j.PropertyConfigurator");
            Method configureMethod = propertyConfigurator.getMethod( "configureAndWatch",
new Class[] { String.class } );

            // Invoke the method using the config file we verified.
configureMethod.invoke( null, new Object[] {
logConfigurationFile.getAbsolutePath() } );

            log = LogFactory.getLog( Mail.class );

            log.info( "Logger using log4j. Class: Mail.class" );

            // log4j is successfully configured.
            enableLog4j = true;
        }
        catch( ClassNotFoundException classNotFoundException )
        {
            // log4j is not available.
        }
        catch( NoSuchMethodException noSuchMethodException ) {
            // log4j is not available.
            System.err.println( "Found log4j Class but method configureAndWatch(String) is
not available." );
        }
        catch( IllegalAccessException illegalAccessException ) {
            // log4j is not available.
            System.err.println( "Found log4j Class but method configureAndWatch(String)
caused an IllegalAccessException." );
        }
        catch( InvocationTargetException invocationTargetException ) {
            Throwable targetException = invocationTargetException.getTargetException();
            System.err.println( "Error occurred while configuring log4j: " +
targetException );
        }
    }
}

```

```

        // If log4j was not configured, initialize the default logger.
        if( !enableLog4j ) {
            initializeDefaultLogging( logConfigurationFile );
        }
    }
}

/**
 * Initializes the logger with the default configuration if no log file is found.
 *
 * @param logConfigurationFile the file that may contain properties
 * to use to configure the default logger.
 *
 * NOTE:
 * Moved to this aspect by Axel
 *
 */
private static void Mail.initializeDefaultLogging( File logConfigurationFile ) {

    String DEFAULT_THRESHOLD = "info";
    String threshold = DEFAULT_THRESHOLD;

    // See if the default threshold is defined in the log.conf file.
    if( logConfigurationFile.exists() ) {

        try {
            Properties logConfigurationProperties = new Properties();
            logConfigurationProperties.load( new FileInputStream( logConfigurationFile )
        );

            threshold = logConfigurationProperties.getProperty(
ConfigurationParameterContants.LOGGING_DEFAULT_THRESHOLD, DEFAULT_THRESHOLD );
            threshold = threshold.trim();
            if( !threshold.equals( "debug" ) && !threshold.equals( "info" ) &&
!threshold.equals( "warn" ) &&
!threshold.equals( "error" ) && !threshold.equals( "fatal" ) ) {
                System.err.println( "Invalid value for property " +
ConfigurationParameterContants.LOGGING_DEFAULT_THRESHOLD + ": " + threshold );
                threshold = DEFAULT_THRESHOLD;
            }
        }
        catch (IOException ioException) {
            System.err.println( "Error loading properties from: " +
logConfigurationFile.getAbsolutePath() + ". " + ioException );
            threshold = DEFAULT_THRESHOLD;
        }
    }

    // commons-logging will default to a logging configuration.
    // see
http://jakarta.apache.org/commons/logging/api/org/apache/commons/logging/package-summary.html
    System.setProperty( "org.apache.commons.logging.simplelog.defaultlog", threshold );
    log = LogFactory.getLog( Mail.class );
    log.warn( "log.conf file not found. Using default log configuration." );
}

/*****
 * The pointcut definitions
 *****/

//The main(..) of Mail
private pointcut main(String[] args) :
    within(Mail) && execution(* main(..) ) && args(args);

//static-log constructors
private pointcut StaticLogIFInit(StaticLogInterface sli):
    this(sli) && initialization(StaticLogInterface.new());
//non-static log constructors
private pointcut LogIFInit(LogInterface li) :
    this(li) && initialization(LogInterface.new(..));

//defining all the pointcuts where we want to log

/*****
 * The advices
 *****/

```

```

*****/

    before(String[] args):  main(args){
        //kjør initLog
        System.out.print("\r\n\r\nStarting LogInit in class: " +
thisJoinPointStaticPart.getSignature().getDeclaringType().getName() + "\r\n\r\n");

        //For some funny reason 'this' will not point to the executing object. Have to
use the joinpoint to get it
        Mail m = (Mail)(thisJoinPoint.getThis());

        // Get the 'root' directory for the mail server.
        String directory = m.getConfigurationDirectory( args );

        // Initialize the logging mechanism.  We want to do this before we do
// anything else.
        m.initializeLogging( directory );

    }

//static log init, must have a case for each class with static log
before(StaticLogInterface sli): StaticLogIFInit(sli){
//this must be repeated for each class with a static log-object. A dynamically
type-casting would be nice...
    if(sli instanceof ServiceListener){
        if(((ServiceListener)sli).logTest == null)
            ((ServiceListener)sli).logTest = LogInit(ServiceListener.class);
    }
    else if(sli instanceof Pop3Processor){
        if(((Pop3Processor)sli).logTest == null)
            ((Pop3Processor)sli).logTest = LogInit(Pop3Processor.class);
    }
    else if(sli instanceof ConfigurationProcessor){
        if(((ConfigurationProcessor)sli).logTest == null)
            ((ConfigurationProcessor)sli).logTest =
LogInit(ConfigurationProcessor.class);
    }
    else if(sli instanceof Message){
        if(((Message)sli).logTest == null)
            ((Message)sli).logTest = LogInit(Message.class);
    }
    else if(sli instanceof SMTPMessage){
        if(((SMTPMessage)sli).logTest == null)
            ((SMTPMessage)sli).logTest = LogInit(SMTPMessage.class);
    }
    else if(sli instanceof SMTPProcessor){
        if(((SMTPProcessor)sli).logTest == null)
            ((SMTPProcessor)sli).logTest = LogInit(SMTPProcessor.class);
    }
    else if(sli instanceof SMTPRemoteSender){
        if(((SMTPRemoteSender)sli).logTest == null)
            ((SMTPRemoteSender)sli).logTest =
LogInit(SMTPRemoteSender.class);
    }
    else if(sli instanceof SMTPSender){
        if(((SMTPSender)sli).logTest == null)
            ((SMTPSender)sli).logTest = LogInit(SMTPSender.class);
    }
    else
        System.out.println("Something wrong. Have forgotten a class");
}

//non-static log init
before(LogInterface li) : LogIFInit(li){
    li.logTest = LogInit(li.getClass());
}

//Mail class logging
private pointcut Mail_NTshutdown() :
    within(Mail) && execution(* shutdown(String[]));

before() : Mail_NTshutdown() {
    Mail m = (Mail)(thisJoinPoint.getThis());
    m.log.debug( "NT Service requested application shutdown." );
}

private pointcut Mail_shutdown() :

```

```

        within(Mail) && execution(public static void shutdown());

before() : Mail_shutdown() {
    Mail m = (Mail)(thisJoinPoint.getThis());
    m.log.warn( "Shutting down Mail Server.  Server will shut down in 60 seconds."
);
}

private pointcut Mail_sl(int port, Class c, int threads) :
    within(Mail) && call(ServiceListener.new( int, Class, int)) && args(port, c,
threads);

before(int port, Class c, int threads) : Mail_sl(port, c, threads){
    Mail m = (Mail)(thisJoinPoint.getThis());
    if( m.log.isDebugEnabled() ){
        if(c == Pop3Processor.class)
            m.log.debug( "Starting POP3 Service on port: " + port );
        else if(c == SMTPProcessor.class)
            m.log.debug( "Starting SMTP Service on port: " + port );
    }
}

//ConigurationManager class logging

private pointcut CFM_loaduser(EmailAddress ema, ConfigurationManager cf) :
    target(cf) && call(User ConfigurationManager.getUser(EmailAddress)) &&
args(ema);

after(EmailAddress ema, ConfigurationManager cf) returning (User u) :
CFM_loaduser(ema,cf){
    if( cf.logTest.isInfoEnabled() && u == null ) cf.logTest.info( "Tried to load
non-existent user: " + ema.getAddress() );
}

private pointcut CF_loadgp (ConfigurationManager cf) :
    this(cf) && cflow( call(void loadGeneralProperties())) && set(String[]
ConfigurationManager.localDomains);

after(ConfigurationManager cf) : CF_loadgp(cf){
    cf.logTest.info( "Loaded " + cf.localDomains.length + " local domains." );
}

private pointcut CF_loadup(ConfigurationManager cf, String un) :
    this(cf) && cflow( call(void loadUserProperties())) && call(* Map.put(String,
User)) && args(un);

after (ConfigurationManager cf, String un) throwing: CF_loadup(cf,un) {
    cf.logTest.warn( "Skipping user: " + un + ".  Address is invalid." );
}

private pointcut CF_loadusers(ConfigurationManager cf) :
    this(cf) && set(Map ConfigurationManager.users);

after (ConfigurationManager cf) : CF_loadusers(cf) {
    if( cf.logTest.isInfoEnabled() ) cf.logTest.info( "Loaded " + cf.users.size() +
" users from user.conf" );
}

private pointcut CF_changeuc(ConfigurationManager cf) :
    this(cf) && cflow(call(void loadUserProperties())) && call(*
Properties.store(...));

after (ConfigurationManager cf) returning : CF_changeuc(cf) {
    cf.logTest.info( "Changes to user.conf persisted to disk." );
}

private pointcut CF_erroruc(ConfigurationManager cf) :
    this(cf) && cflow(call(void loadUserProperties())) && call(*
Properties.store(...));

after (ConfigurationManager cf) throwing : CF_erroruc(cf) {
    cf.logTest.error( "Unable to store changes to user.conf!  Plain text passwords
were not hashed!" );
}

private pointcut CF_errorpps(ConfigurationManager cf, String sValue, int iValue) :

```

```

        this(cf) && ( cflow(call(int parsePort(String, int))) && args(sValue, iValue) )
&& call(* Integer.parseInt(..));

        after (ConfigurationManager cf, String sValue, int iValue) throwing : CF_errorpps(cf,
sValue, iValue) {
            cf.logTest.warn( "Error parsing port string: " + sValue + " using default
value: " + iValue );
        }

        private pointcut CF_errorecpg(ConfigurationManager cf, String address, String
password):
            this(cf) && (cflow( call(User loadUser(String, Properties))) && args(address,
..)) && (call(String PasswordManager.encryptPassword(String)) && args(password));

        after (ConfigurationManager cf, String address, String password) returning :
CF_errorecpg(cf, address, password) {
            if(password == null)
                cf.logTest.error( "Error encrypting plaintext password from user.conf
for user " + address );
        }

        private pointcut CF_errorfwai(ConfigurationManager cf, String fullAddress, String
fwAddress) :
            this(cf) &&
            (cflow(call(User loadUser(..))) && args(fullAddress,..)) &&
            (call(EmailAddress.new(String)) && args(fwAddress)));

        after (ConfigurationManager cf, String fullAddress, String fwAddress) throwing :
CF_errorfwai( cf, fullAddress, fwAddress ){
            cf.logTest.warn("Forward address: " + fwAddress + " for " + fullAddress + " is
invalid and will be ignored." );
        }

        private pointcut CF_loadfwa(ConfigurationManager cf, EmailAddress[] emailAddresses,
String fullAddress) :
            this(cf) &&
            (cflow(call(User ConfigurationManager.loadUser(..))) && args(fullAddress, ..))
&&
            (call(* User.setForwardAddresses(EmailAddress[])) && args(emailAddresses));

        after(ConfigurationManager cf, EmailAddress[] emailAddresses, String fullAddress) :
CF_loadfwa(cf, emailAddresses, fullAddress){
            if( cf.logTest.isDebugEnabled() ) cf.logTest.debug( emailAddresses.length + "
forward addresses load for user: " + fullAddress );
        }

        private pointcut CF_changecf() :
            this(ConfigurationManager.ConfigurationFileWatcher) &&
            call(* loadGeneralProperties());

        before() : CF_changecf() {
            ((LogInterface)thisJoinPoint.getThis()).logTest.info( "General Configuration
File Changed, reloading..." );
        }

        private pointcut CF_changeucf() :
            this(ConfigurationManager.ConfigurationFileWatcher) &&
            call(* loadUserProperties());

        before() : CF_changeucf() {
            ((LogInterface)thisJoinPoint.getThis()).logTest.info( "User Configuration File
Changed, reloading..." );
        }

        private pointcut CF_errorcfw(Throwable throwable) :
            this(ConfigurationManager.ConfigurationFileWatcher) && handler( Throwable ) &&
args(throwable);

        before(Throwable throwable) : CF_errorcfw(throwable) {
            ((LogInterface)thisJoinPoint.getThis()).logTest.error( "Error in
ConfigurationWatcher thread. Thread will continue to execute. " + throwable, throwable );
        }

        //ConfigurationProcessor class logging
        private pointcut CP_errorsotu(ConfigurationProcessor cp) :
            this(cp) && call(* ServerSocket.setSoTimeout(..));

```

```

after(ConfigurationProcessor cp) throwing : CP_errorsotu(cp) {
    cp.logTest.fatal( "Error initializing Socket Timeout in SMTPProcessor" );
}

private pointcut CP_errordc(ConfigurationProcessor cp, Throwable e) :
    this(cp) && handler(Throwable) && args(e);

before(ConfigurationProcessor cp, Throwable e) : CP_errordc(cp, e){
    cp.logTest.debug( "Disconnecting Exception:", e );
    cp.logTest.info( "Disconnecting" );
}

private pointcut CP_errordce(ConfigurationProcessor cp) :
    this(cp) && cflow(handler(Throwable)) && call(* Socket.close());

after (ConfigurationProcessor cp) throwing( IOException ioe ) : CP_errordce(cp) {
    cp.logTest.debug( "Error disconnecting.", ioe );
}

//DeliveryService class logging
private pointcut DS_addauthip(DeliveryService ds, String clientIp) :
    this(ds) && execution(* ipAuthenticated(String)) && args(clientIp);

before (DeliveryService ds, String clientIp) : DS_addauthip(ds, clientIp) {
    if( ds.logTest.isDebugEnabled() ) ds.logTest.debug( "Adding authenticated IP
address: " + clientIp );
}

private pointcut DS_lockmbx(DeliveryService ds, EmailAddress address):
    this(ds) && execution(* isMailboxLocked(EmailAddress)) && args(address);

before (DeliveryService ds, EmailAddress address) : DS_lockmbx(ds, address){
    if( ds.logTest.isDebugEnabled() ) ds.logTest.debug( "Locking Mailbox: " +
address.getAddress() );
}

private pointcut DS_unlockmbx(DeliveryService ds, EmailAddress address) :
    this(ds) && execution(* unlockMailbox(EmailAddress)) && args(address);

before (DeliveryService ds, EmailAddress address) : DS_unlockmbx(ds, address){
    if( ds.logTest.isDebugEnabled() ) ds.logTest.debug( "Unlocking Mailbox: " +
address.getAddress() );
}

private pointcut DS_erroraad(DeliveryService ds) :
    this(ds) && cflow(execution(* isRelayApproved(..))) && call(*
StringTokenizer.nextToken() );

before (DeliveryService ds) throwing: DS_erroraad(ds) {
    ds.logTest.warn( "Invalid ApprovedAddress found. Skipping." );
}

//Message class logging
private pointcut MSG_delset(Message m, boolean deleted) :
    this(m) && execution(* setDeleted(boolean)) && args(deleted);

before (Message m, boolean deleted) : MSG_delset(m, deleted) {
    m.logTest.debug( "Setting is deleted to: " + deleted );
}

//Pop3Processor class logging
private pointcut P3_errorinitsotu(Pop3Processor pc) :
    this(pc) && call(* ServerSocket.setSoTimeout(..));

after (Pop3Processor pc) throwing : P3_errorinitsotu(pc) {
    pc.logTest.fatal( "Error initializing Socket Timeout in Pop3Processor" );
}

private pointcut P3_hostconnect(Pop3Processor pc, InetAddress remoteAddress) :
    this(pc) && target(remoteAddress) && call(String InetAddress.getHostAddress());

after (Pop3Processor pc, InetAddress remoteAddress) returning (String clientIp) :
P3_hostconnect(pc, remoteAddress) {
    if( pc.logTest.isInfoEnabled() ) pc.logTest.info( remoteAddress.getHostAddress() +
 "(" + clientIp + ") socket connected via POP3." );
}

```

```

private pointcut P3_errordc(Pop3Processor pc, Throwable e) :
    this(pc) && cflow(execution(* run())) && (handler(Throwable) && args(e));

before (Pop3Processor pc, Throwable e) : P3_errordc(pc, e) {
    pc.logTest.debug( "Disconnecting Exception:", e );
    pc.logTest.info( "Disconnecting" );
}

private pointcut P3_errordcsm(Pop3Processor pc) :
    this(pc) && cflow(execution(* run())) && cflow(handler(Throwable)) && call(*
write(..));

after (Pop3Processor pc) throwing(Exception e): P3_errordcsm(pc) {
    pc.logTest.debug( "Error sending disconnect message.", e );
}

private pointcut P3_errordce(Pop3Processor pc) :
    this(pc) && cflow(execution(* run())) && cflow(handler(Throwable)) && call(*
Socket.close());

after (Pop3Processor pc) throwing(IOException ioe): P3_errordce(pc) {
    pc.logTest.debug( "Error disconnecting.", ioe );
}

private pointcut P3_shutdown(Pop3Processor pc) :
    this(pc) && execution(* run());

after (Pop3Processor pc) : P3_shutdown(pc) {
    pc.logTest.warn( "Pop3Processor (1) shut down gracefully" );
}

private pointcut P3_shuttingdown(Pop3Processor pc) :
    this(pc) && execution(* shutdown());

before (Pop3Processor pc) : P3_shuttingdown(pc) {
    pc.logTest.warn( "Shutting down Pop3Processor." );
}

private pointcut P3_userquit(Pop3Processor pc, String s) :
    this(pc) && execution(* checkQuit(String)) && args(s);

before (Pop3Processor pc, String s) : P3_userquit(pc, s) {
    if(s.equals(pc.COMMAND_QUIT)) pc.logTest.debug( "User has QUIT the session." );
}

private pointcut P3_userlogin(Pop3Processor pc, User user):
    this(pc) && target(user) && cflow(execution(* authenticate())) && call(*
User.isPasswordValid( String) );

after (Pop3Processor pc, User user) returning (boolean bool): P3_userlogin(pc, user) {
    if(bool) { if( pc.logTest.isInfoEnabled() ) pc.logTest.info( "User: " +
user.getFullUsername() + " logged in successfully."); }
    else pc.logTest.info( "Login failed for user: " + user.getFullUsername());
}

private pointcut P3_loginfailed(Pop3Processor pc, EmailAddress address) :
    this(pc) && cflow(execution(* authenticate())) && (call(*
ConfigurationManager.getUser( EmailAddress ) ) && args(address));

after (Pop3Processor pc, EmailAddress address) returning(User user): P3_loginfailed(pc,
address){
    if(user == null) pc.logTest.info( "Login failed for user: " +
address.getAddress());
}

private pointcut P3_isdeletemsg(Pop3Processor pc, String argument, User user) :
    this(pc) && target(user) && cflow( (execution(* handleRetr(String)) ||
execution(* handleTop(String))) && args(argument)) && call(Long User.getNumberOfMessage());

after (Pop3Processor pc, String argument, User user) returning(long numMessages) :
P3_isdeletemsg(pc, argument, user){
    if( pc.logTest.isDebugEnabled() ) {
        int messageNumber = Integer.parseInt(argument);
        pc.logTest.debug( "Is Msg Deleted: " + user.getMessage( messageNumber
).isDeleted() );
        pc.logTest.debug( "Message: " + messageNumber + " of " + numMessages );
    }
}

```

```

}

private pointcut P3_msgnf(Pop3Processor pc) :
    this(pc) && call(FileReader.new(...));

after (Pop3Processor pc) throwing (FileNotFoundException fnfe) : P3_msgnf(pc) {
    pc.logTest.error( "Requested message could not be found on disk.", fnfe );
}

private pointcut P3_errorretmsg(Pop3Processor pc) :
    this(pc) && call(* BufferedReader.readLine()) && (cflow(execution(*
handleRetr(...)) || cflow(execution(* handleTop(...))));

after (Pop3Processor pc) throwing (IOException ioe) : P3_errorretmsg(pc) {
    pc.logTest.error( "Error retrieving message.", ioe );
}

private pointcut P3_handletop(Pop3Processor pc) :
    this(pc) && execution(* handleTop(...));

before (Pop3Processor pc) : P3_handletop(pc) {
    pc.logTest.debug( "In Top" );
}

private pointcut P3_notpass(Pop3Processor pc) :
    this(pc) && cflow(execution(String read())) && call(String String.trim());

after (Pop3Processor pc) returning (String inputLine) : P3_notpass(pc) {
    if( pc.logTest.isDebugEnabled() && !inputLine.startsWith( "PASS" ) )
pc.logTest.debug( "Read Input: " + inputLine );
}

private pointcut P3_errorrfs(Pop3Processor pc, IOException ioe) :
    this(pc) && cflow(execution(String read())) && ( handler(IOException) &&
args(ioe));

before (Pop3Processor pc, IOException ioe) : P3_errorrfs(pc, ioe) {
    pc.logTest.error( "Error reading from socket.", ioe );
}

private pointcut P3_writeop(Pop3Processor pc, String message) :
    this(pc) && execution(* write(String)) && args(message);

before (Pop3Processor pc, String message) : P3_writeop(pc, message){
    if( pc.logTest.isDebugEnabled() ) pc.logTest.debug( "Writing Output: " +
message );
}

//ServiceListener class logging
private pointcut SL_startsl(ServiceListener sl) :
    this(sl) && execution(void run());

before (ServiceListener sl) : SL_startsl(sl) {
    if( sl.logTest.isDebugEnabled() ) sl.logTest.debug( "Starting ServiceListener
on port: " + sl.port );
}

private pointcut SL_errorslcreate(ServiceListener sl, IOException e) :
    this(sl) && cflow(execution(void run())) && handler(IOException) && args(e);

before (ServiceListener sl, IOException e) : SL_errorslcreate(sl, e){
    InetAddress listenAddress =
ConfigurationManager.getInstance().getListenAddress();
    String address = "localhost";
    if( listenAddress != null ) {
        address = listenAddress.getHostAddress();
    }
    sl.logTest.error("Could not create ServiceListener on address: " + address + "
port: " + sl.port + ". No connections will be accepted on this port!" );
}

private pointcut SL_acceptcon(ServiceListener sl) :
    this(sl) && cflow(execution(void run())) && call(ServerSocket.new(...));

after (ServiceListener sl) returning (ServerSocket ss) : SL_acceptcon(sl) {
    sl.logTest.info( "Accepting Connections on port: " + ss.getLocalPort() );
}

```

```

    private pointcut SL_errorslc(ServiceListener sl, Exception e) :
        this(sl) && cflow(execution(void run())) && handler(Exception) && args(e);

    before (ServiceListener sl, Exception e) : SL_errorslc(sl, e) {
        sl.logTest.error("ServiceListener Connection failed on port: " + sl.port + ".
Error: " + e );
    }

    private pointcut SL_errorshutdown(ServiceListener sl) :
        this(sl) && cflow(execution(void shutdown())) && call(* Thread.join());

    after (ServiceListener sl) throwing : SL_errorshutdown(sl) {
        sl.logTest.error("Was interrupted while waiting for thread to die");
    }

    private pointcut SL_shutdown(ServiceListener sl) :
        this(sl) && cflow(execution(void shutdown())) && call(* Thread.join());

    after (ServiceListener sl) returning : SL_shutdown(sl) {
        sl.logTest.info("Thread gracefully terminated");
    }

    private pointcut SL_errorcs(ServiceListener sl) :
        this(sl) && cflow(execution(void shutdown())) && call(* ServerSocket.close());

    after (ServiceListener sl) throwing (Exception e): SL_errorcs(sl) {
        sl.logTest.error( "Failed to close server socket", e );
    }

    private pointcut SL_closesocket(ServiceListener sl) :
        this(sl) && cflow(execution(void shutdown())) && call(* ServerSocket.close());

    after (ServiceListener sl) returning : SL_closesocket(sl) {
        sl.logTest.error( "Server socket successfully closed" );
    }

    //ShutdownService logger, inside Shutdown aspect
    private pointcut SSR_shutdown() :
        this(Shutdown) && call(void shutdown());

    before () : SSR_shutdown(){
        Mail.log.warn( "Server is shutting down." );
    }

    after () throwing (Exception e) : SSR_shutdown() {
        Mail.log.error("Failed to terminate properly", e);
    }

    after () returning : SSR_shutdown() {
        Mail.log.warn( "Server shutdown complete." );
    }

    //SMTPMessage class logger
    private pointcut SM_errorgetdir(SMTPMessage sm, File failedDir) :
        this(sm) && target(failedDir) && cflow(execution(void moveToFailedFolder())) &&
call(boolean File.exists());

    after (SMTPMessage sm, File failedDir) returning(boolean bool): SM_errorgetdir(sm,
failedDir) {
        if( !bool ) sm.logTest.info( "failed directory does not exist. Creating: " +
failedDir.getAbsolutePath() );
    }

    private pointcut SM_errorcreatefaileddir(SMTPMessage sm, File failedDir) :
        this(sm) && target(failedDir) && cflow(execution(void moveToFailedFolder())) &&
call(boolean File.mkdirs());

    after (SMTPMessage sm, File failedDir) returning(boolean bool):
SM_errorcreatefaileddir(sm, failedDir){
        if( !bool ) sm.logTest.error( "Error creating failed directory. No incoming
mail will be accepted!" );
    }

    private pointcut SM_errormovetodir(SMTPMessage sm, File messageLocation) :
        this(sm) && target(messageLocation) && cflow(execution(void
moveToFailedFolder())) && call(boolean File.renameTo(...));

```

```

        after (SMTPMessage sm, File messageLocation) returning( boolean bool):
SM_errormovetodir(sm, messageLocation){
            if( !bool ) sm.logTest.error( "moveToFailedFolder failed. Message was
not renamed." );
        }

        private pointcut SM_errordirdne(SMTPMessage sm, File smtpDirectory) :
            this(sm) && target(smtpDirectory) && cflow(execution(void save())) &&
call(boolean File.exists());

        after (SMTPMessage sm, File smtpDirectory) returning(boolean bool): SM_errordirdne(sm,
smtpDirectory){
            if( !bool ) sm.logTest.info( "SMTP Mail directory does not exist.
Creating: " + smtpDirectory.getAbsolutePath() );
        }

        private pointcut SM_errorcreatedir(SMTPMessage sm, File smtpDirectory) :
            this(sm) && target(smtpDirectory) && cflow(execution(void save())) &&
call(boolean File.mkdirs());

        after (SMTPMessage sm, File smtpDirectory) returning(boolean bool):
SM_errorcreatedir(sm, smtpDirectory){
            if( !bool ) sm.logTest.error( "Error creating SMTP Mail directory. No
incoming mail will be accepted!" );
        }

        private pointcut SM_errorclosespoolfile(SMTPMessage sm) :
            this(sm) && cflow(execution(void save())) && call(* FileWriter.close());

        after (SMTPMessage sm) throwing : SM_errorclosespoolfile(sm) {
            sm.logTest.warn( "Unable to close pool file for SMTPMessage " +
sm.messageLocation.getAbsolutePath() );
        }

        private pointcut SM_loadsmtpmsg(SMTPMessage sm, String filename) :
            this(sm) && cflow(execution(* load(String)) && args(filename)) && call(String
BufferedReader.readLine());

        after (SMTPMessage sm, String filename) returning (String version) : SM_loadsmtpmsg(sm,
filename){
            if( sm.logTest.isDebugEnabled() )
                sm.logTest.debug( "Loading SMTP Message " + filename + " version " +
version );
            if( !sm.FILE_VERSION.equals( version ) )
                sm.logTest.error( "Error loading SMTP Message. Can not handle file version: "
+ version );
        }

        private pointcut SM_errorparseaddress(SMTPMessage sm, String addresses) :
            this(sm) && cflow(execution(* inflateAddresses(String)) && args(addresses)) &&
call(EmailAddress.new(...));

        after (SMTPMessage sm, String addresses) throwing( InvalidAddressException
invalidAddressException) : SM_errorparseaddress(sm, addresses){
            sm.logTest.error( "Unable to parse to address read from database. Full String
is: " + addresses, invalidAddressException );
        }

        //SMTPProcessor class logging
        private pointcut SP_errorsocketto(SMTPProcessor sp) :
            this(sp) && cflow(execution(void run())) && call(*
ServerSocket.setSoTimeout(..));

        after (SMTPProcessor sp) throwing(SocketException se) : SP_errorsocketto(sp){
            sp.logTest.fatal( "Error initializing Socket Timeout in SMTPProcessor" );
        }

        private pointcut SP_socketcon(SMTPProcessor sp, InetAddress remoteAddress) :
            this(sp) && target(remoteAddress) && cflow(execution(void run())) &&
call(String InetAddress.getHostAddress());

        after (SMTPProcessor sp, InetAddress remoteAddress) returning(String clientIp) :
SP_socketcon(sp, remoteAddress){
            if( sp.logTest.isInfoEnabled() ) sp.logTest.info( remoteAddress.getHostName() +
"(" + clientIp + ") socket connected via SMTP." );
        }

```

```

private pointcut SP_errordce(SMTPProcessor sp, Throwable e) :
    this(sp) && cflow(execution(void run())) && handler(Throwable) && args(e);

before (SMTPProcessor sp, Throwable e) : SP_errordce(sp, e){
    sp.logTest.debug( "Disconnecting Exception:", e );
    sp.logTest.info( "Disconnecting" );
}

private pointcut SP_errordc(SMTPProcessor sp) :
    this(sp) && cflow(execution(void run())) && call(* Socket.close());

before (SMTPProcessor sp) throwing(IOException ioe) : SP_errordc(sp){
    sp.logTest.debug( "Error disconnecting.", ioe );
}

private pointcut SP_shutdown(SMTPProcessor sp) :
    this(sp) && execution(void run());

after (SMTPProcessor sp) : SP_shutdown(sp){
    sp.logTest.warn( "SMTPProcessor shut down gracefully" );
}

before (SMTPProcessor sp) : SP_shutdown(sp){
    sp.logTest.warn( "Shutting down SMTPProcessor." );
}

private pointcut SP_erroruq(SMTPProcessor sp) :
    this(sp) && execution(void checkQuit());

before (SMTPProcessor sp) throwing : SP_erroruq(sp) {
    sp.logTest.debug( "User has QUIT the session." );
}

private pointcut SP_errormfe(SMTPProcessor sp) :
    this(sp) && cflow(execution(* handleMailFrom(..))) && call(EmailAddress.new());

after (SMTPProcessor sp) returning: SP_errormfe(sp){
    sp.logTest.debug( "MAIL FROM is empty" );
}

private pointcut SP_mailfrom(SMTPProcessor sp, String fromAddress) :
    this(sp) && cflow(execution(* handleMailFrom(..))) &&
call(EmailAddress.new(String)) && args(fromAddress);

after (SMTPProcessor sp, String fromAddress) returning: SP_mailfrom(sp, fromAddress){
    if( sp.logTest.isDebugEnabled() ) sp.logTest.debug( "MAIL FROM: " +
fromAddress );
}

private pointcut SP_errorparsefa(SMTPProcessor sp, String fromAddress) :
    this(sp) && cflow(execution(* handleMailFrom(..))) &&
call(EmailAddress.new(String)) && args(fromAddress);

after (SMTPProcessor sp, String fromAddress) throwing: SP_errorparsefa(sp,
fromAddress){
    sp.logTest.debug( "Unable to parse From Address: " + fromAddress );
}

private pointcut SP_rcptto(SMTPProcessor sp, EmailAddress address, String clientIp) :
    this(sp) && cflow(execution(void handleRcptTo(..))) && call(boolean
DeliveryService.acceptAddress( EmailAddress, String )) && args(address,clientIp);

after (SMTPProcessor sp, EmailAddress address, String clientIp) returning(boolean bool)
: SP_rcptto(sp, address, clientIP){
    if(bool) {
        if( sp.logTest.isDebugEnabled() ) sp.logTest.debug( "RCTP TO: " +
address.getAddress() + " accepted." );
    }
    else{
        if( sp.logTest.isInfoEnabled() ) sp.logTest.info( "Invalid delivery
address for incoming mail: " + address.getAddress() + " from client: " + clientIp );
    }
}

private pointcut SP_errorrcptto(SMTPProcessor sp, EmailAddress address) :

```

```

        this(sp) && cflow(execution(* handleRcptTo(..))) &&
call(EmailAddress.new(String)) && args(address);

        after (SMTPProcessor sp, EmailAddress address) throwing( InvalidAddressException iae )
: SP_errorrcptto(sp, address){
            sp.logTest.debug( "RCTP TO: " + address + " rejected." );
        }

        private pointcut SP_readdata(SMTPProcessor sp, String inputString) :
            this(sp) && target(inputString) && cflow(execution(* handleData())) &&
call(boolean String.equals(..));

        after (SMTPProcessor sp, String inputString) returning (boolean bool) : SP_readdata(sp,
inputString){
            if(bool) { if( sp.logTest.isDebugEnabled() ) sp.logTest.debug( "Read Data: " +
inputString ); }
            else sp.logTest.debug( "Data Input Complete." );
        }

        private pointcut SP_errorretdata(SMTPProcessor sp) :
            this(sp) && cflow(execution(* handleData())) && call(*
BufferedReader.readLine());

        after (SMTPProcessor sp) throwing(IOException ioe) : SP_errorretdata(sp){
            sp.logTest.error( "An error occured while retrieving the message data.", ioe );
        }

        private pointcut SP_handledata(SMTPProcessor sp) :
            this(sp) && execution(* handleData());

        after (SMTPProcessor sp) : SP_handledata(sp){
            if( sp.logTest.isInfoEnabled() ) sp.logTest.info( "Message " +
sp.message.getMessageLocation().getName() + " accepted for delivery." );
        }

        private pointcut SP_readdata(SMTPProcessor sp) :
            this(sp) && cflow(execution(* read())) && call(String String.trim());

        after (SMTPProcessor sp) returning(String inputLine) : SP_readdata(sp) {
            if( sp.logTest.isDebugEnabled() ) { sp.logTest.debug( "Read Input: " +
inputLine ); }
        }

        private pointcut SP_errorrfs(SMTPProcessor sp) :
            this(sp) && cflow(execution(* read())) && call(* BufferedReader.readLine());

        after (SMTPProcessor sp) throwing(IOException ioe) : SP_errorrfs(sp) {
            sp.logTest.error( "Error reading from socket.", ioe );
        }

        private pointcut SP_writemsg(SMTPProcessor sp, String message) :
            this(sp) && execution(void write(String)) && args(message);

        before(SMTPProcessor sp, String message) : SP_writemsg(sp, message){
            if( sp.logTest.isDebugEnabled() ) { sp.logTest.debug( "Writing: " + message );
}
        }

        //SMTPRemoteSender class logging
        private pointcut SRS_errorclosesocket(SMTPRemoteSender srs) :
            this(srs) && call(* Socket.close());

        after (SMTPRemoteSender srs) throwing(IOException ioe) : SRS_errorclosesocket(srs){
            srs.logTest.error( "Error closing socket: " + ioe );
        }

        private pointcut SRS_dnslookup(SMTPRemoteSender srs, EmailAddress address) :
            this(srs) && cflow(execution(Socket connect(EmailAddress)) && args(address)) &&
call(* Lookup.run());

        after (SMTPRemoteSender srs, EmailAddress address) returning(Record[] records):
SRS_dnslookup(srs,address){
            if(records == null) srs.logTest.warn( "DNS Lookup for domain: " +
address.getDomain() + " failed." );
        }

        private pointcut SRS_errorconsmtp(SMTPRemoteSender srs, String host) :

```

```

        this(srs) && call(Socket.new(String, int)) && args(host,..);

    after (SMTPRemoteSender srs, String host) throwing ( Exception e ) :
SRS_errorconsmtplib(srs,host){
    srs.logTest.debug( "Connection to SMTP Server: " + host + " failed with
exception: " + e ) ;
}

    private pointcut SRS_errorrrfs(SMTPRemoteSender srs) :
        this(srs) && call(* BufferedReader.readLine());

    after (SMTPRemoteSender srs) throwing(IOException ioe) : SRS_errorrrfs(srs){
        srs.logTest.error( "Error reading from socket.", ioe );
    }

    private pointcut SRS_readline(SMTPRemoteSender srs) :
        this(srs) && call(* BufferedReader.readLine());

    after (SMTPRemoteSender srs) returning (String s) : SRS_readline(srs){
        if( srs.logTest.isDebugEnabled() ) { srs.logTest.debug( "Read Input: " + s ); }
    }

    //SMTPSender class logging
    private pointcut SS_findmsgtodeliver(SMTPSender ss) :
        this(ss) && get(boolean SMTPSender.running);

    after (SMTPSender ss) returning (boolean running) : SS_findmsgtodeliver(ss){
        if(running) ss.logTest.debug( "Checking for SMTP messages to deliver" );
    }

    private pointcut SS_errordelivermsg(SMTPSender ss) :
        this(ss) && call(* SMTPMessage.load(..));

    after (SMTPSender ss) throwing( Throwable throwable ) : SS_errordelivermsg(ss){
        ss.logTest.error( "An error occurred attempting to deliver an SMTP Message: " +
throwable, throwable );
    }

    private pointcut SS_errorsleep(SMTPSender ss) :
        this(ss) && call(* Thread.sleep(..));

    after (SMTPSender ss) throwing (InterruptedException ie) : SS_errorsleep(ss){
        ss.logTest.error( "Sleeping Thread was interrupted." );
    }

    private pointcut SS_run(SMTPSender ss) :
        this(ss) && execution(void run());

    after (SMTPSender ss) : SS_run(ss){
        ss.logTest.warn( "SMTPSender shut down gracefully." );
    }

    private pointcut SS_shutdown(SMTPSender ss) :
        this(ss) && execution(* shutdown());

    before (SMTPSender ss) : SS_shutdown(ss) {
        ss.logTest.warn( "Attempting to shut down SMTPSender." );
    }

    private pointcut SS_skipdeliversmtplibmsg(SMTPSender ss, SMTPMessage message) :
        this(ss) && execution(* deliver(SMTPMessage)) && args(message);

    before (SMTPSender ss, SMTPMessage message) : SS_skipdeliversmtplibmsg(ss, message){
        if( message.getScheduledDelivery().getTime() > System.currentTimeMillis() ){
            if( ss.logTest.isDebugEnabled() ) ss.logTest.debug( "Skipping delivery
of message " + message.getMessageLocation().getName() + " because the scheduled delivery time
is still in the future: " + message.getScheduledDelivery() );
        }
    }

    private pointcut SS_deliversmtplibmsg(SMTPSender ss, EmailAddress address, SMTPMessage
message) :
        this(ss) && ( call(* deliverLocalMessage(..)) || call(*
deliverRemoteMessage(..)) ) && args(address, message);

    before(SMTPSender ss, EmailAddress address, SMTPMessage message) :
SS_deliversmtplibmsg(ss, address, message){

```

```

        if( ss.logTest.isDebugEnabled() ) { ss.logTest.debug( "Attempting to deliver
message from: " + message.getFromAddress().getAddress() + " to: " + address ); }
    }

    after (SMTPSender ss, EmailAddress address, SMTPMessage message) throwing
(NotFoundException e): SS_deliversmtppmsg(ss, address, message){
        ss.logTest.info( "Delivery attempted to unknown user: " +
address.getAddress() );
    }

    after (SMTPSender ss, EmailAddress address, SMTPMessage message) returning:
SS_deliversmtppmsg(ss, address, message){
        if( ss.logTest.isInfoEnabled() ) { ss.logTest.info( "Delivery
complete for message " + message.getMessageLocation().getName() + " to: " + address ); }
    }

    private pointcut SS_removesmtppmsg(SMTPSender ss, File f):
        this(ss) && target(f) && call(boolean File.delete());

    after(SMTPSender ss, File f) returning(boolean bool) : SS_removesmtppmsg(ss, f){
        if(!bool)
            ss.logTest.error( "Error removed SMTP message after delivery! This
message may be redelivered. " + f.getName() );
    }

    private pointcut SS_movetofailedfolder(SMTPSender ss) :
        this(ss) && call(* Message.moveToFailedFolder());

    before (SMTPSender ss) : SS_movetofailedfolder(ss){
        ss.logTest.info( "Delivery of message from MAILER_DAEMON failed, moving to
failed folder." );
    }

    after (SMTPSender ss) throwing : SS_movetofailedfolder(ss){
        ss.logTest.error( "Unable to move failed message to 'failed' folder." );
    }

    private pointcut SS_updatespooledmsg(SMTPSender ss):
        this(ss) && cflow(execution(* deliver(..)) && call(* SMTPMessage.save()));

    after (SMTPSender ss) throwing(Exception exception): SS_updatespooledmsg(ss) {
        ss.logTest.error( "Error updating spooled message for next delivery. Message
may be re-delivered.", exception );
    }

    private pointcut SS_deliverlocal(SMTPSender ss, EmailAddress address) :
        this(ss) && execution(* deliverLocalMessage(EmailAddress, ..)) &&
args(address,..);

    before (SMTPSender ss, EmailAddress address) : SS_deliverlocal(ss, address){
        if( ss.logTest.isDebugEnabled() ) { ss.logTest.debug( "Delivering Message to
local user: " + address.getAddress() ); }
    }

    private pointcut SS_getuser(SMTPSender ss) :
        this(ss) && call(User ConfigurationManager.getUser(EmailAddress));

    after (SMTPSender ss) returning(User user) : SS_getuser(ss){
        if(user == null)
            ss.logTest.debug( "User not found" );
    }

    private pointcut SS_delivertodefault(SMTPSender ss, EmailAddress address):
        this(ss) && cflow(call(* ConfigurationManager.isDefaultUserEnabled())) &&
call(User ConfigurationManager.getUser(EmailAddress)) && args(address);

    after (SMTPSender ss, EmailAddress address) returning(User user) :
SS_delivertodefault(ss, address){
        if(user == null) {ss.logTest.debug( "User not found in default delivery
options" );}
        else if( ss.logTest.isDebugEnabled() ) { ss.logTest.info( "Delivering
message to default user: " + address ); }
    }

    private pointcut SS_deliveringto(SMTPSender ss):
        this(ss) && call(File File.createTempFile(..));

```

```

        after (SMTPSender ss) returning (File f) : SS_deliveringto(ss){
            if( ss.logTest.isDebugEnabled() ) { ss.logTest.debug( "Delivering to: " +
f.getAbsolutePath() ); }
        }

        private pointcut SS_errorlocaldelivery(SMTPSender ss, IOException ioe) :
            this(ss) && cflow(execution(* deliverLocalMessage(..)) && handler(IOException)
&& args(ioe));

        before (SMTPSender ss, IOException ioe) : SS_errorlocaldelivery(ss, ioe){
            ss.logTest.error( "Error performing local delivery.", ioe );
        }

        private pointcut SS_errorclosestream(SMTPSender ss) :
            this(ss) && call(* BufferedWriter.close());

        after (SMTPSender ss) throwing(IOException ioe) : SS_errorclosestream(ss){
            ss.logTest.error( "Error closing output Stream.", ioe );
        }

        private pointcut SS_remotedelivery(SMTPSender ss, EmailAddress address) :
            this(ss) && execution(* deliverRemoteMessage( EmailAddress , SMTPMessage)) &&
args(address, ..);

        before (SMTPSender ss, EmailAddress address) : SS_remotedelivery(ss, address){
            if( ss.logTest.isDebugEnabled() ) { ss.logTest.debug( "Delivering Message to
remote user: " + address ); }
        }

        private pointcut SS_bouncemsg(SMTPSender ss, EmailAddress address, SMTPMessage message
) :
            this(ss) && execution(* bounceMessage( EmailAddress, SMTPMessage ) ) &&
args(address, message);

        before (SMTPSender ss, EmailAddress address, SMTPMessage message ) : SS_bouncemsg(ss,
address, message){
            if( ss.logTest.isInfoEnabled() ) { ss.logTest.info( "Bouncing Messsage
from " + message.getFromAddress().getAddress() + " to " + address.getAddress() ); }
        }

        private pointcut SS_errorstorebounce(SMTPSender ss) :
            this(ss) && cflow(call(* bounceMessage(..)) && call(* SMTPMessage.save()));

        after (SMTPSender ss) throwing : SS_errorstorebounce(ss){
            ss.logTest.error( "Error storing outgoing 'bounce' email message");
        }

        //User class logging
        private pointcut U_authenticate(User u) :
            this(u) && execution(* isPasswordValid(..));

        before (User u) : U_authenticate(u) {
            if( u.logTest.isDebugEnabled() ) u.logTest.debug( "Authenticating User: " +
u.getFullUsername() );
        }

        after (User u) returning(boolean result) : U_authenticate(u){
            if( u.logTest.isDebugEnabled() && !result ) u.logTest.debug( "Authentication
Failed for User: " + u.getFullUsername() );
        }

        private pointcut U_getuserdir(User u) :
            this(u) && call( boolean File.exists() ) && cflow(execution(*
getUserDirectory()));

        after (User u) returning(boolean bool) : U_getuserdir(u){
            if(!bool) { if( u.logTest.isInfoEnabled() ) u.logTest.info( "Directory for
user: " + u.getFullUsername() + "does not exist, creating..." ); }
        }

        private pointcut U_errorgetuserdir(User u, File directory) :
            this(u) && target(directory) && call( boolean File.isDirectory() ) &&
cflow(execution(* getUserDirectory()));

        after (User u, File directory) returning(boolean bool) : U_errorgetuserdir(u,
directory){

```

```
        if(!bool) { u.logTest.error( "User Directory: " + directory.getAbsolutePath() +  
" for user: " + u.getFullUsername() + " does not exist." ); }  
    }  
}
```

2.2 Shutdown

```
package com.ericdaugherty.mail.server.aspects;

import com.ericdaugherty.mail.server.Mail;
import com.ericdaugherty.mail.server.ShutdownService;
import com.ericdaugherty.mail.server.services.pop3.Pop3Processor;
import com.ericdaugherty.mail.server.services.smtp.*;
import com.ericdaugherty.mail.server.services.general.*;

privileged aspect Shutdown{

    public void ConnectionProcessor.shutdown(){};
    /**
     * Notifies thread to stop processing and exit.
     */
    public void Pop3Processor.shutdown() {
        running = false;
    }
    public void SMTPProcessor.shutdown() {
        running = false;
    }
}

public void SMTPSender.shutdown() {
    running = false;
}

/** The ShutdownService Thread. Started when the JVM is shutdown. */
private static Thread Mail.shutdownServiceThread;
private static ShutdownService Mail.shutdownService;

/**
 * Provides a 'safe' way for the application to shut down. This
 * method is provided to enable compatability with the JavaService
 * NT Service wrapper class. It defers the call to the shutdown method.
 *
 * @param args
 */
public static void Mail.shutdown( String[] args ) {
    shutdown();
}

/**
 * Provides a 'safe' way for the application to shut down. It will
 * attempt to stop the running threads. If the threads to not stop
 * within 60 seconds, the application will force the threads to
 * stop by calling System.exit();
 */
public static void Mail.shutdown() {

    popListener.shutdown();
    smtpListener.shutdown();
    smtpSender.shutdown();

    try{
        smtpSenderThread.join(10000);
    }
    catch (InterruptedException ie){}
    smtpSenderThread = null;
}

/**
 * Stops all processors.
 */
public void ServiceListener.shutdown() {
    for( int index = 0; index < processors.length; index++ ) {

        processors[index].shutdown();

        try{
            threadPool[index].join(10000);
        }
        catch (InterruptedException ie){}
        threadPool[index] = null;
    }

    try{
```

```

        serverSocket.close();
    }
    catch(Exception e){}
serverSocket = null;
}

private pointcut MailConstructor() :
    within(Mail) && execution(* main(String[]));

private pointcut ShutdownServiceRun() :
    this(ShutdownService) && execution(public void run());

after () : MailConstructor(){
    //Initialize ShutdownService
    Mail.shutdownService = new ShutdownService();
    Mail.shutdownServiceThread = new Thread(Mail.shutdownService);
    Runtime.getRuntime().addShutdownHook( Mail.shutdownServiceThread );
}

after () : ShutdownServiceRun(){
    try {
        Mail.shutdown();
    }
    catch (Exception e) {
}
}
}

```

2.3 ExceptionPrinter

```
package com.ericdaugherty.mail.server.aspects;

import com.ericdaugherty.mail.*;
import java.io.*;
import java.lang.*;
import java.net.*;

public aspect ExceptionPrinter{

    private pointcut allExceptions( ) :
        handler(Exception+);

    private pointcut notSupported( ) :
        handler(SocketTimeoutException) ||
        handler(NullPointerException) ||
        handler(ArrayIndexOutOfBoundsException) ||
        !within(com.ericdaugherty.mail...);

    private pointcut exceptions( Exception e ) :
        ( allExceptions() && !notSupported() ) && args(e);

    before ( Exception e ): exceptions( e ) {
        System.out.print("\r\n*** Exception ***" +
            "\r\nClass: " +
            thisJoinPointStaticPart.getSignature().getDeclaringType().getName() +
            "\r\nMethod: " +
            thisEnclosingJoinPointStaticPart.getSignature().getName() +
            "\r\nException: " +
            e +
            "\r\n***\r\n");
    }
}
```

2.4 Mail

```
/*
 * $Workfile: Mail.java $
 * $Revision: 1.4 $
 * $Author: edaugherty $
 * $Date: 2003/10/15 19:06:23 $
 *
 * $Edited by: Axel Anders Kvale $
 */
*****/

package com.ericdaugherty.mail.server;

//Java imports
import java.io.*;
import java.lang.reflect.Method;
import java.lang.reflect.InvocationTargetException;
import java.util.Properties;

//Local imports
import com.ericdaugherty.mail.server.configuration.ConfigurationManager;
import com.ericdaugherty.mail.server.configuration.ConfigurationParameterContants;
import com.ericdaugherty.mail.server.services.general.ServiceListener;
import com.ericdaugherty.mail.server.services.smtp.SMTPSender;
import com.ericdaugherty.mail.server.services.smtp.SMTPProcessor;
import com.ericdaugherty.mail.server.services.pop3.Pop3Processor;

/**
 * This class is the entrypoint for the Mail Server application. It
 * creates threads to listen for SMTP and POP3 connections. It also
 * handles the configuration information and initialization of the
 * User subsystem.
 */
public class Mail {

    private static ServiceListener popListener;
    private static ServiceListener smtpListener;
    private static SMTPSender smtpSender;

    /** The SMTP sender thread */
    private static Thread smtpSenderThread;

    /**
     * This method is the entrypoint to the system and is responsible
     * for the initial configuration of the application and the creation
     * of all 'service' threads.
     */
    public static void main( String[] args ) {

        // Perform the basic application startup. If anything goes
        // wrong here, we need to abort the application.
        try {
            // Get the 'root' directory for the mail server.
            String directory = getConfigurationDirectory( args );

            // Initialize the Configuration Manager.
            ConfigurationManager configurationManager =
                ConfigurationManager.initialize( directory );

            //Start the threads.
            int port;
            int executeThreads =
                configurationManager.getExecuteThreadCount();

            //Start the Pop3 Thread.
            port = configurationManager.getPop3Port();
            popListener =
                new ServiceListener( port, Pop3Processor.class,
                    executeThreads );
            new Thread( popListener, "POP3" ).start();

            //Start SMTP Threads.
            port = configurationManager.getSmtpPort();
            smtpListener =
                new ServiceListener(
```

```

        port, SMTPProcessor.class, executeThreads );
    new Thread( smtpListener, "SMTP" ).start();

    //Start the SMTPSender thread (This thread actually
    // delivers the mail recieved by the SMTP threads.
    smtpSender = new SMTPSender();
    smtpSenderThread = new Thread( smtpSender, "SMTPSender" );
    smtpSenderThread.start();

}
catch( RuntimeException runtimeException ) {
    System.err.println(
        "The application failed to initialize." );
    System.err.println( runtimeException.getMessage() );
    runtimeException.printStackTrace();
    System.exit( 0 );
}
}

/*****
// Private Interface
*****/

/**
 * Parses the input parameter for the configuration directory,
 * or defaults to the local directory.
 *
 * @param args the commandline arguments.
 * @return the directory to use as the 'root'.
 */

private static String getConfigurationDirectory( String[] args ) {

    String directory = ".";
    File directoryFile;

    // First, check to see if the location was passed as a
    // paramter.
    if (args.length == 1) {
        directory = args[0];
    }
    // Otherwise, use the default, which is 'mail.conf' in the
    // current directory.
    else if( (directoryFile = new File( directory )).exists() ) {
        System.out.println(
            "Configuration Directory not specified. Using \"
            + directoryFile.getAbsolutePath() + "\" );
    }
    // If no file was specified and the default does not exist,
    // printing out a usage line.
    else {
        System.out.println(
            "Usage: java com.ericdaugherty.mail.server.Mail"
            + "<configuration directory>");
        throw new RuntimeException(
            "Unable to load the configuration file." );
    }

    return directory;
}
}
}

```

2.5 ShutdownService

```
/*
 * $Workfile: Mail.java $
 * $Revision: 1.3 $
 * $Author: edaugherty $
 * $Date: 2003/10/15 19:01:01 $
 *
 * $Edited by: Axel Anders Kvale $
 *
 */
package com.ericdaugherty.mail.server;

/**
 * This thread runs when the JVM is shutdown.
 *
 * @author Eric Daugherty
 */
public class ShutdownService implements Runnable {

    /** Logger */
    //private Log log = LogFactory.getLog( this.getClass() );

    /**
     * Runs when the JVM is shutdown. Stops all threads gracefully.
     */
    public void run() {

    }
}
```

2.6 ConfigurationManager

```
/*
 * $Source: /mail/server/configuration/ConfigurationManager.java $
 * $Revision: 1.10 $
 * $Author: edaugherty $
 * $Date: 2003/12/24 02:26:34 $
 *
 * $ Edited by: Axel Anders Kvale $
 *
 */
package com.ericdaugherty.mail.server.configuration;

import java.util.Properties;
import java.util.StringTokenizer;
import java.util.Vector;
import java.util.Enumeration;
import java.util.Map;
import java.util.HashMap;
import java.util.ArrayList;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.File;
import java.io.FileOutputStream;
import java.net.InetAddress;
import java.net.UnknownHostException;

import com.ericdaugherty.mail.server.info.User;
import com.ericdaugherty.mail.server.info.EmailAddress;
import com.ericdaugherty.mail.server.errors.InvalidAddressException;

/**
 * Provides a centralized repository for all configuration
 * information.
 * <p>
 * All configuration information should be retrieved here for
 * every use. The ConfigurationManager will reload
 * configuration changes dynamically.
 * <p>
 * Classes may cache the reference to the ConfigurationManager
 * instance, as only one will ever be created.
 *
 * @author Eric Daugherty
 */
public class ConfigurationManager
    implements ConfigurationParameterContants {

    private static ConfigurationManager instance;

    /** The file reference to the mail.conf configuration file */
    private File generalConfigurationFile;

    /** The timestamp for the mail.conf file when it was last loaded */
    private long generalConfigurationFileTimestamp;

    /** The file reference to the user.conf configuration file */
    private File userConfigurationFile;

    /** The timestamp for the user.conf file when it was last loaded */
    private long userConfigurationFileTimestamp;

    /** Tracks whether the user configuration properties were changed
     * during loading */
    private boolean userConfModified = false;

    /** The root directory used to store the incoming and outgoing
     * messages. */
    private String mailDirectory;

    /** Array of domains that the SMTP server should accept mail for
     * local delivery */
    private String[] localDomains;

    /** The number of threads to use for each listener */
    private int executeThreadCount;
}
```

```

/** The local IP address to listen on. Null for all addresses */
private InetAddress listenAddress;

/** The port the SMTP server listens on. */
private int smtpPort;

/** The port the POP3 server listens on */
private int pop3Port;

/** The timeout length for authenticated ip addresses */
private long authenticationTimeoutMilliseconds;

/** True if POP Before SMTP is enabled */
private boolean enablePOPBeforeSMTP;

/** IP Addresses that are allowed to relay mail. */
private String[] relayApprovedIpAddresses;

/** True if all outgoing mail should go though the default
 * server */
private boolean defaultSmtpServerEnabled;

/** The servers to send all outgoing mail through */
private String[] defaultSmtpServers;

/**
 * True if email to the local domain for a non-existent
 * user should be delivered to the default user.
 */
private boolean defaultUserEnabled;

/** The user to delivery default email to */
private EmailAddress defaultUser;

/** The number of seconds to wait between delivery attempts */
private long deliveryIntervalSeconds;

/**
 * The max number of delivery attempts before message is
 * considered 'undeliverable' and moved to 'failed' folder
 */
private int deliveryAttemptThreshold;

/** A Map of Users keyed by their full username */
private Map users;

/**
 * Initialize the file path. Enforces the Singleton pattern.
 *
 * @param generalConfigurationFile the file to load the general
 * configuration from.
 * @param userConfigurationFile the file to load the user
 * configuration from.
 */
private ConfigurationManager(
    File generalConfigurationFile, File userConfigurationFile )
{
    this.generalConfigurationFile = generalConfigurationFile;
    this.userConfigurationFile = userConfigurationFile;
}

/**
 * Initializes the ConfigurationManager to use the specified
 * directory. This method should only be called once during
 * startup, and then never again. The file path can not
 * be re-initialized!
 *
 * @param configurationDirectory the directory that contains
 * mail.conf and user.conf
 * @return returns the singleton instance of the
 * ConfigurationManager.
 * @throws RuntimeException thrown if called more than once, the
 * file does not exist, or there is an error loading the file.
 */
public static synchronized ConfigurationManager initialize(
    String configurationDirectory ) throws RuntimeException

```

```

{
    String generalConfigFilename = "mail.conf";
    String userConfigFilename = "user.conf";

    // Make sure we are not already configured.
    if( instance != null )
    {
        throw new RuntimeException(
            "ConfigurationManager:initialize() " +
            "called more than once!" );
    }

    // Verify the General config file exists.
    File generalConfigFile = new File(
        configurationDirectory, generalConfigFilename );
    if( !generalConfigFile.exists() ||
        !generalConfigFile.isFile() )
    {
        throw new RuntimeException(
            "Invalid mail.conf ConfigurationFile! " +
            generalConfigFile.getAbsolutePath() );
    }

    // Verify the User config file exists.
    File userConfigFile = new File(
        configurationDirectory, userConfigFilename );
    if( !userConfigFile.exists() || !userConfigFile.isFile() )
    {
        throw new RuntimeException(
            "Invalid user.conf ConfigurationFile! " +
            userConfigFile.getAbsolutePath() );
    }

    // Go ahead and create the singleton instance.
    instance = new ConfigurationManager(
        generalConfigFile, userConfigFile );

    instance.setMailDirectory( configurationDirectory );

    // Load the properties from disk.
    instance.loadProperties();

    // Start the Watchdog Thread
    instance.new ConfigurationFileWatcher().start();

    return instance;
}

/**
 * Provides access to the singleton instance.
 *
 * @return the singleton instance.
 */
public static synchronized ConfigurationManager getInstance()
{
    if( instance == null )
    {
        throw new RuntimeException(
            "ConfigurationManager can not be accessed " +
            "before it is initialized!" );
    }
    return instance;
}

public void loadProperties()
{
    loadGeneralProperties();
    loadUserProperties();
}

/**
 * The root directory used to store the incoming and outgoing
 * messages.
 *
 * @return String
 */
public String getMailDirectory() {

```

```

        return mailDirectory;
    }

    /**
     * Get the max number of delivery attempts before message is
     * considered
     * 'undeliverable' and moved to 'failed' folder
     * @return int
     */
    public int getDeliveryAttemptThreshold() {
        return deliveryAttemptThreshold;
    }

    /**
     * The root directory used to store the incoming and outgoing
     * messages.
     *
     * @param mailDirectory String
     */
    public void setMailDirectory(String mailDirectory) {
        this.mailDirectory = mailDirectory;
    }

    /**
     * Array of domains that the SMTP server should accept mail for
     * local delivery
     *
     * @return String array
     */
    public String[] getLocalDomains() {
        return localDomains;
    }

    /**
     * Checks the local domains to see if the specified
     * parameter matches.
     *
     * @param domain a domain to check.
     * @return true if and only if it matches exactly an existing
     * domain.
     */
    public boolean isLocalDomain( String domain )
    {
        domain = domain.toLowerCase();
        if( localDomains != null && localDomains.length > 0 )
        {
            int numDomains = localDomains.length;
            for( int index = 0; index < numDomains; index++ )
            {
                if( localDomains[index].equals( domain ) )
                {
                    return true;
                }
            }
        }
        return false;
    }

    /**
     * Array of domains that the SMTP server should accept mail for
     * local delivery
     *
     * @param localDomains String array
     */
    public void setLocalDomains(String[] localDomains) {
        this.localDomains = localDomains;
    }

    /**
     * The number of threads to use for each listener.
     *
     * @return int
     */
    public int getExecuteThreadCount() {
        return executeThreadCount;
    }

```

```

/**
 * The number of threads to use for each listener.
 *
 * @param executeThreadCount int
 */
public void setExecuteThreadCount(int executeThreadCount) {
    this.executeThreadCount = executeThreadCount;
}

/**
 * The local IP address to listen on. Null for all addresses
 *
 * @return null for all addresses.
 */
public InetAddress getListenAddress() {
    return listenAddress;
}

/**
 * The port the SMTP server listens on.
 *
 * @return port number
 */
public int getSmtpPort() {
    return smtpPort;
}

/**
 * The port the SMTP server listens on.
 *
 * @param smtpPort port number
 */
public void setSmtpPort(int smtpPort) {
    this.smtpPort = smtpPort;
}

/**
 * The port the POP3 server listens on.
 *
 * @return port number
 */
public int getPop3Port() {
    return pop3Port;
}

/**
 * The port the POP3 server listens on.
 *
 * @param pop3Port port number
 */
public void setPop3Port(int pop3Port) {
    this.pop3Port = pop3Port;
}

/**
 * Returns the specified user, or null if the user
 * does not exist.
 *
 * @param address the user's full email address.
 * @return null if the user does not exist.
 */
public User getUser(EmailAddress address)
{
    User user = (User) users.get( address.getAddress() );
    return user;
}

/** The timeout length for authenticated ip addresses */
public long getAuthenticationTimeoutMilliseconds() {
    return authenticationTimeoutMilliseconds;
}

/** The timeout length for authenticated ip addresses */
public void setAuthenticationTimeoutMinutes(long minutes) {
    this.authenticationTimeoutMilliseconds = minutes * 60 * 1000;
}

```

```

/** True if POP Before SMTP is a valid relay option */
public boolean isEnabledPOPBeforeSMTP() {
    return enablePOPBeforeSMTP;
}

/** True if POP Before SMTP is a valid relay option */
public void setEnablePOPBeforeSMTP(boolean enablePOPBeforeSMTP) {
    this.enablePOPBeforeSMTP = enablePOPBeforeSMTP;
}

/** IP Addresses that are allowed to relay mail. */
public String[] getRelayApprovedIpAddresses() {
    return relayApprovedIpAddresses;
}

/** IP Addresses that are allowed to relay mail. */
public void setRelayApprovedIpAddresses(
    String[] relayApprovedIpAddresses) {
    this.relayApprovedIpAddresses = relayApprovedIpAddresses;
}

/** True if all outgoing mail should go through the default
 * server */
public boolean isDefaultSmtpServerEnabled() {
    return defaultSmtpServerEnabled;
}

/** True if all outgoing mail should go through the default s
 * erver */
public void setDefaultSmtpServerEnabled(
    boolean defaultSmtpServerEnabled) {
    this.defaultSmtpServerEnabled = defaultSmtpServerEnabled;
}

/** The servers to send all outgoing mail through */
public String[] getDefaultSmtpServers() {
    return defaultSmtpServers;
}

/** The server to send all outgoing mail through */
public void setDefaultSmtpServers(String[] defaultSmtpServers) {
    this.defaultSmtpServers = defaultSmtpServers;
}

public boolean isDefaultUserEnabled() {
    return defaultUserEnabled;
}

public void setDefaultUserEnabled(boolean defaultUserEnabled) {
    this.defaultUserEnabled = defaultUserEnabled;
}

public EmailAddress getDefaultUser() {
    return defaultUser;
}

public void setDefaultUser(EmailAddress defaultUser) {
    this.defaultUser = defaultUser;
}

/** The number of seconds to wait between delivery attempts */
public long getDeliveryIntervalSeconds() {
    return deliveryIntervalSeconds;
}

/** The number of milliseconds to wait between delivery
 * attempts */
public long getDeliveryIntervealMilliseconds()
{
    return deliveryIntervalSeconds * 1000;
}

/** The number of seconds to wait between delivery attempts */
public void setDeliveryIntervalSeconds(
    long deliveryIntervalSeconds) {
    this.deliveryIntervalSeconds = deliveryIntervalSeconds;
}

```

```

/**
 * Loads the properties file into the local variables for quick
 * access.
 */
private void loadGeneralProperties()
{
    Properties properties = new Properties();

    try {
        FileInputStream inputStream = new FileInputStream(
            generalConfigurationFile );
        properties.load( inputStream );
    }
    catch (IOException e) {
        // All checks should be done before we get here, so
        // there better
        // not be any errors. If so, throw a RuntimeException.
        throw new RuntimeException(
            "Error Loading Properties File! " +
            "Unable to continue Operation." );
    }

    //
    // Load the local domains
    //

    String domains = properties.getProperty( DOMAINS, "" );
    localDomains = tokenize( domains.trim().toLowerCase() );
    if( domains.length() == 0 )
    {
        throw new RuntimeException( "No Local Domains defined!" +
            "Can not run without local domains defined." );
    }

    //
    // Load the number of Execute Threads for each listener
    //

    //Load the number of execute threads to use for each
    //ServiceListener.
    String threadsString = properties.getProperty(
        EXECUTE_THREADS, "5" );
    try {
        executeThreadCount = Integer.parseInt( threadsString );
    }
    catch( NumberFormatException nfe ) {
        executeThreadCount = 5;
    }

    //
    // Load the address port numbers
    //

    String listenAddressString = properties.getProperty(
        LISTEN_ADDRESS, "" );
    listenAddressString.trim();
    // If not address is specified, default to null.
    // ServiceListener can handle
    // a null listenAddress.
    if( listenAddressString.length() > 0 )
    {
        try {
            listenAddress = InetAddress.getByName(
                listenAddressString );
        }
        catch ( UnknownHostException unknownHostException ) {
            throw new RuntimeException(
                "Invalid value for property: " +
                LISTEN_ADDRESS +
                ". Server will listen on all addresses. " +
                unknownHostException );
        }
    }
    else
    {
        listenAddress = null;
    }
}

```

```

}

String smtpPortString = properties.getProperty( SMTPPORT );
String pop3PortString = properties.getProperty( POP3PORT );
smtpPort = parsePort( smtpPortString, 25 );
pop3Port = parsePort( pop3PortString, 110 );

//
// Load the SMTP Delivery Parameters

enablePOPBeforeSMTP = Boolean.valueOf( properties.getProperty(
    RELAY_POP_BEFORE_SMTP, "false" ) ).booleanValue();
// Initialize the timeout Minutes parameter
String timeoutString = properties.getProperty(
    RELAY_POP_BEFORE_SMTP_TIMEOUT, "10" );
try {
    setAuthenticationTimeoutMinutes(
        Long.parseLong( timeoutString ) );
}
catch( NumberFormatException nfe ) {
    //Set the default to 10 minutes.
    setAuthenticationTimeoutMinutes( 10 );
}

// Relay approved IP Addresses
String ipAddresses = properties.getProperty(
    RELAY_ADDRESSLIST, "" );
setRelayApprovedIpAddresses( tokenize( ipAddresses ) );

// Load default Server info.

String smtpServers = properties.getProperty(
    DEFAULT_SMTP_SERVERS, "" ).trim();
if( smtpServers.length() > 0 ) {
    defaultSmtpServerEnabled = true;
    defaultSmtpServers = tokenize( smtpServers );
}
else {
    defaultSmtpServerEnabled = false;
    defaultSmtpServers = new String[0];
}

// Load default user info
String defaultUserString = properties.getProperty(
    DEFAULT_USER, "" ).trim();
if( defaultUserString.length() > 0 )
{
    try {
        defaultUser = new EmailAddress( defaultUserString );
        defaultUserEnabled = true;
    }
    catch (InvalidAddressException e) {
        throw new RuntimeException(
            "Invalid address for default user: " +
                defaultUserString );
    }
}
else
{
    defaultUser = null;
    defaultUserEnabled = false;
}

String deliveryIntervalString =
    properties.getProperty( SMTP_DELIVERY_INTERVAL, "10" );
try {
    //Convert to number and then convert to ms.
    setDeliveryIntervalSeconds(
        Long.parseLong( deliveryIntervalString ) );
}
catch( NumberFormatException nfe ) {
    setDeliveryIntervalSeconds( 10 );
}

// Set the Delivery Attempt Threshold.

```

```

try
{
    deliveryAttemptThreshold =
        Integer.parseInt(
            properties.getProperty(
                SMTP_DELIVERY_THRESHOLD, "10" ) );
}
catch( NumberFormatException numberFormatException )
{
    deliveryAttemptThreshold = 10;
}

// Update the 'last loaded' timestamp.
generalConfigurationFileTimestamp =
    generalConfigurationFile.lastModified();
}

private void loadUserProperties() {

    // Load the properties
    Properties properties = new Properties();

    try {
        FileInputStream inputStream =
            new FileInputStream( userConfigurationFile );
        properties.load( inputStream );
    }
    catch (IOException e) {
        // All checks should be done before we get here, so there
        // better not be any errors.  If so, throw a
        // RuntimeException.
        throw new RuntimeException(
            "Error Loading Properties File! " +
            "Unable to continue Operation." );
    }

    // Clear the modified flag so we know if we have to save the
    // file after loading.
    userConfModified = false;

    //
    // Load the users
    //

    Map users = new HashMap();
    Enumeration propertyKeys = properties.keys();
    String key;
    String fullUsername;
    String correctedUsername;
    while( propertyKeys.hasMoreElements() )
    {
        key = (String) propertyKeys.nextElement();
        if( key.startsWith( USER_DEF_PREFIX ) )
        {
            fullUsername = key.substring(
                USER_DEF_PREFIX.length() );
            correctedUsername = fullUsername.toLowerCase();
            try {
                users.put(
                    correctedUsername, loadUser(
                        fullUsername, properties ) );
            }
            catch (InvalidAddressException e) {
            }
        }
    }
    this.users = users;

    // Save the user configuration if they changed.
    if( userConfModified ) {
        try {
            FileOutputStream out = new FileOutputStream(
                userConfigurationFile );
            properties.store( out, USER_PROPERTIES_HEADER );
        }
        catch (IOException e) {
        }
    }
}

```

```

    }

    // Update the 'last loaded' timestamp.
    userConfigurationFileTimestamp =
        userConfigurationFile.lastModified();
}

/**
 * Loads the values of the specified key from the configuration
 * file.
 * This method parses the value into a String array
 * using the comma (,) as a delimiter. This method returns an
 * array of size 0 if the the value string was null or empty.
 *
 * @param value the string to tokenize into an array.
 * @return a String[] of the values, or an empty array if the key
 * could not be found.
 */
public static String[] tokenize( String value ) {

    if( value == null || value.trim().equals( "" ) ) {
        return new String[0];
    }
    else {
        StringTokenizer stringTokenizer = new StringTokenizer(
            value, "," );
        Vector tokenVector = new Vector();
        while( stringTokenizer.hasMoreTokens() ) {
            tokenVector.addElement(
                stringTokenizer.nextToken().trim() );
        }

        String[] values = new String[ tokenVector.size() ];
        return (String[]) tokenVector.toArray( values );
    }
}

/**
 * Converts the string into a valid port number.
 *
 * @param stringValue the string value to parse
 * @param defaultValue the default value to return if parsing
 * fails.
 * @return a valid int.
 */
private int parsePort( String stringValue, int defaultValue )
{
    int value = defaultValue;
    if( stringValue != null && stringValue.length() > 0 )
    {
        try {
            value = Integer.parseInt( stringValue );
        }
        catch (NumberFormatException e) {
        }
    }
    return value;
}

/**
 * Creates a new User instance for the specified username
 * using the specified properties.
 *
 * @param fullAddress full username (me@mydomain.com)
 * @param properties the properties that contain the user
 * parameters.
 * @return a new User instance.
 */
private User loadUser( String fullAddress, Properties properties )
throws InvalidAddressException
{
    EmailAddress address = new EmailAddress( fullAddress );
    User user = new User( address );

    // Load the password
    String password = properties.getProperty(
        USER_DEF_PREFIX + fullAddress );
}

```

```

// If the password is not hashed, hash it now.
if( password.length() != 60 ) {
    password = PasswordManager.encryptPassword( password );
    properties.setProperty(
        USER_DEF_PREFIX + fullAddress, password );
    if( password == null ) {
        throw new RuntimeException(
            "Error encrypting password for user: " +
            fullAddress );
    }
    userConfModified = true;
}
user.setPassword( password );

// Load the 'forward' addresses.
String forwardAddressesString =
    properties.getProperty(
        USER_PROPERTY_PREFIX +
        fullAddress +
        USER_FILE_FORWARDS );
String[] forwardAddresses = new String[0];
if( forwardAddressesString != null &&
    forwardAddressesString.trim().length() >= 0 )
{
    forwardAddresses = tokenize( forwardAddressesString );
}
ArrayList addressList =
    new ArrayList( forwardAddresses.length );
for( int index = 0; index < forwardAddresses.length; index++ ){
    try {
        addressList.add(
            new EmailAddress( forwardAddresses[index] ) );
    }
    catch (InvalidAddressException e) {
    }
}

EmailAddress[] emailAddresses =
    new EmailAddress[ addressList.size() ];
emailAddresses =
    (EmailAddress[]) addressList.toArray( emailAddresses );

user.setForwardAddresses( emailAddresses );

return user;
}

/*****
// Watchdog Inner Class
/*****

/**
 * Checks the user configuration file and reloads it if it is new.
 */
class ConfigurationFileWatcher extends Thread {

    /**
     * Initialize the thread.
     */
    public ConfigurationFileWatcher() {
        super( "User Config Watchdog" );
        setDaemon( true );
    }

    /**
     * Check the timestamp on the file to see
     * if it has been updated.
     */
    public void run() {
        long sleepTime = 10 * 1000;
        while( true )
        {
            try {
                Thread.sleep( sleepTime );
                if( generalConfigurationFile.lastModified() >
                    generalConfigurationFileTimestamp ) {
                    loadGeneralProperties();
                }
            }
            catch (Exception e) {
            }
        }
    }
}

```

```

        }
        if( userConfigurationFile.lastModified() >
            userConfigurationFileTimestamp ) {
            loadUserProperties();
        }
    }
    catch( Throwable throwable ) {
    }
}
}

private static final String LF = "\r\n";

private static final String USER_PROPERTIES_HEADER =
    "# Java Email Server (JES) User Configuration" + LF +
    "#" + LF +
    "# All users are defined in this file. To add a user, follow" + LF +
    "# the following pattern:" + LF +
    "# user.<username@domain>=<plain text password>" + LF +
    "#" + LF +
    "# The plain text password will be converted to a hash when the file" + LF +
    "# is first loaded by the server." + LF +
    "#" + LF +
    "# Additional configuration such as forward addresses can be specified as:" + LF +
    "# userprop.<username@domain>.forwardAddresses=<Comma list of forward addresses>" + LF
+
    "#" + LF +
LF +
    "# When a message is received for a local user, the user's address will be replaced" +
LF +
    "# with the addresses in the forwardAddresses property. If you also wish to have" +
    "# a copy delivered to the local user, you may add the user's local address to" + LF +
    "# the forwardAddresses property" + LF +
    "#";
}

```

2.7 ConfigurationParameterContants

```
/* *****
 * $Workfile: ConfigurationParameterContants.java $
 * $Revision: 1.3 $
 * $Author: edaugherty $
 * $Date: 2003/12/24 02:26:34 $
 *
 * *****/

package com.ericdaugherty.mail.server.configuration;

/**
 * This interface defines the names for all configuration properties
 * loaded from the configuration file. This interface also serves as
 * documentation for the possible paramters and their expected values.
 * <p>
 * Descriptions of the paratmers are also available in the ConfigurationTool
 * java application included with this distribution.
 *
 * @author Eric Daugherty
 */
public interface ConfigurationParameterContants {

    // *****
    // General Paramters
    // *****

    /**
     * The local IP address that the server will listen on. If this
     * property is not set, or is invalid, the server will listen on all
     * addresses.
     */
    public static final String LISTEN_ADDRESS = "listen.address";

    /**
     * The pop3port parameter defines the port to listen to incoming
     * Pop3 connection on. By default, this value should be 110.
     */
    public static final String POP3PORT = "pop3port";

    /**
     * The smtpport parameter defines the port to listen to incoming
     * SMTP connection on. By default, this value should be 25.
     */
    public static final String SMTPPORT = "smtpport";

    /**
     * The domains parameter defines the domain names that this server
     * will accept mail for. All domains not listed here will will either
     * be relayed or ignored. To configure mulitiple domains, just separate
     * each domain with a comma.
     */
    public static final String DOMAINS = "domains";

    /**
     * The number of threads that will be allocated to each connection listener
     * for each port.
     */
    public static final String EXECUTE_THREADS = "threads";

    // *****
    // Mail Delivery Paramters
    // *****

    /**
     * The default user to deliver mail addressed to local unknown users.
     */
    public static final String DEFAULT_USER = "defaultuser";

    /**
     * The default server to deliver mail addressed to remote users.
     */
    public static final String DEFAULT_SMTP_SERVERS = "defaultsmtpservers";

    /**
```

```

    * Enables the POP3 login as a valid address for SMTP Relaying.
    */
    public static final String RELAY_POP_BEFORE_SMTP = "relay.popbeforesmtp";

    /**
     * This setting only applies when RELAY_POP_BEFORE_SMTP is set to true.
     * This setting defines the timeout period (in minutes) between when a
     * user (ip address) last authenticated with the POP3 server and when they will
     * no longer be able to send SMTP mail to remote domains. This option defaults
     * to 10 minutes.
     */
    public static final String RELAY_POP_BEFORE_SMTP_TIMEOUT = "relay.popbeforesmtp.timeout";

    /**
     * This is the label for the UI
     */
    public static final String RELAY_ADDRESSLIST = "relay.ipaddresses";

    /**
     * The server stores incoming SMTP messages on disk before attempting to deliver them.
    This
     * setting determines how often (in seconds) the server checks the disk for new messages
    to deliver. The
     * smaller the number, the faster message will be processed. However, a smaller number
    will cause
     * the server to use more of your system's resources.
     */
    public static final String SMTP_DELIVERY_INTERVAL = "smtpdelivery.interval";

    /**
     * The server picks the messages from the disk in order to deliver them. If some message
     * cannot be delivered to remote SMTP server at that moment, because of some error, then
    the message
     * will be kept on the disk for later delivery attempt. However server can't retry
    delivery
     * indefinitely, therefore following config entry will set maximum number of retries
    before the server
     * gives up on the message and moves it from smtp spool directory to failed directory.
     */
    public static final String SMTP_DELIVERY_THRESHOLD = "smtpdelivery.threshold";

    /**
     * User Parameters
     */
    /**
     * Defines the prefix to usernames stored in the properties file.
     * A username should be stored as:
     * USER_DEF_PREFIX<user@domain.com>=<password>
     */
    public static final String USER_DEF_PREFIX = "user.";

    /**
     * Defines the prefix for user properties usernames stored in the
     * properties file.
     * A user property should be stored as:
     * USER_PROPERTY_PREFIX<user@domain.com>.<property name>=<value>
     */
    public static final String USER_PROPERTY_PREFIX = "userprop.";

    /**
     * The USER_PROPERTY_PREFIX<user@domain.com>.<forwardAddresses>=<value>
     * property defines a comma separated
     * list of addresses that mail to this user will be forwarded to.
     */
    public static final String USER_FILE_FORWARDS = ".forwardAddresses";

    /**
     * Logging Paramters
     */
    /**
     * Defines the default log threshold to use if log4j is not enabled.
     */
    public static final String LOGGING_DEFAULT_THRESHOLD = "defaultthreshold";
}

```

2.8 PasswordManager

```
/* *****
 * $Workfile: PasswordManager.java $
 * $Revision: 1.1 $
 * $Author: edaugherty $
 * $Date: 2003/10/01 19:30:25 $
 *
 * *****/

package com.ericdaugherty.mail.server.configuration;

//Java imports
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

/**
 * Creates encrypted passwords and validating passwords.
 *
 * @author Eric Daugherty
 */
public class PasswordManager {

    /**
     * *****
     * // Public Interface
     * *****
     */

    /**
     * Creates a one-way has of the specified password. This allows passwords to be
     * safely stored in the database without any way to retrieve the original value.
     *
     * @param password the string to encrypt.
     *
     * @return the encrypted password, or null if encryption failed.
     */
    public static String encryptPassword( String password ) {

        try {
            MessageDigest md = MessageDigest.getInstance("SHA");

            //Create the encrypted Byte[]
            md.update( password.getBytes() );
            byte[] hash = md.digest();

            //Convert the byte array into a String

            StringBuffer hashStringBuf = new StringBuffer();
            String byteString;
            int byteLength;

            for( int index = 0; index < hash.length; index++ ) {

                byteString = String.valueOf( hash[index ] + 128 );

                //Pad string to 3. Otherwise hash may not be unique.
                byteLength = byteString.length();
                switch( byteLength ) {
                    case 1:
                        byteString = "00" + byteString;
                        break;
                    case 2:
                        byteString = "0" + byteString;
                        break;
                }
                hashStringBuf.append( byteString );
            }

            return hashStringBuf.toString();
        }
        catch( NoSuchAlgorithmException nsae ) {
            System.out.println( "Error getting password hash - " + nsae.getMessage() );
            return null;
        }
    }
}
```

2.9 ConfigurationProcessor

```
/*
 * $Workfile: ConfigurationProcessor.java $
 * $Revision: 1.6 $
 * $Author: edaugherty $
 * $Date: 2003/10/01 19:30:25 $
 *
 * $Edited by: Axel Anders Kvale$
 */
*****/

package com.ericdaugherty.mail.server.configuration.console;

import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;
import java.io.PrintWriter;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.InterruptedIOException;
import java.io.IOException;

import com.ericdaugherty.mail.server.services.general.ConnectionProcessor;
import com.ericdaugherty.mail.server.configuration.ConfigurationManager;

/**
 * This class provides a console based interface to the configuration
 * information.
 *
 * @author Eric Daugherty
 */
public class ConfigurationProcessor implements ConnectionProcessor {

    /** The ConfigurationManager */
    private static ConfigurationManager configurationManager =
        ConfigurationManager.getInstance();

    /** Indicates if this thread should continue to run or shut down */
    private boolean running = true;

    /** The server socket used to listen for incoming connections */
    private ServerSocket serverSocket;

    /** Socket connection to the client */
    private Socket socket;

    /** Writer to sent data to the client */
    private PrintWriter out;

    /** Reader to read data from the client */
    private BufferedReader in;

    /**
     * Sets the socket used to communicate with the client.
     */
    public void setSocket(ServerSocket serverSocket) {
        this.serverSocket = serverSocket;
    }

    /**
     * Entry point for this thread to handle a connection.
     */
    public void run() {
        try {
            //Set the socket to timeout every second so it does not
            //just block forever.
            serverSocket.setSoTimeout( 1000 );
        }
        catch( SocketException se ) {}

        while( running ) {
            try {
                socket = serverSocket.accept();
            }
        }
    }
}
```

```

        //Prepare the input and output streams.
        out = new PrintWriter(socket.getOutputStream(), true);
        in = new BufferedReader(
            new InputStreamReader(
                socket.getInputStream() ));

        handleCommands();
    }
    catch( InterruptedException ioe ) {
        //This is fine, it should time out every second if
        //a connection is not made.
    }
    //If any exception gets to here uncaught, it means we
    // should just disconnect.
    catch( Throwable e ) {
        try {
            if( socket != null ) {
                socket.close();
            }
        }
        catch( IOException ioe ) {}
    }
}

/**
 * Notifies the service to stop processing connections.
 */
public void shutdown() {
    running = false;
}

private void handleCommands() {
    out.write( "Welcome to the JES Configuration Console." );
}
}

```

2.10 Exceptions

```
/*
 * $Workfile: AuthenticationException.java $
 * $Revision: 1.2 $
 * $Author: edaugherty $
 * $Date: 2003/10/01 19:30:24 $
 */
package com.ericdaugherty.mail.server.errors;

/**
 * Defines an exception to be used when a login attempt fails.
 *
 * @author Eric Daugherty
 */
public class AuthenticationException extends Exception {

    public AuthenticationException() {
        super();
    }
}

/*
 * $Workfile: InvalidAddressException.java $
 * $Revision: 1.2 $
 * $Author: edaugherty $
 * $Date: 2003/10/01 19:30:24 $
 */
package com.ericdaugherty.mail.server.errors;

/**
 * Defines an exception to be used when an address can not be parsed.
 *
 * @author Eric Daugherty
 */
public class InvalidAddressException extends Exception {

    public InvalidAddressException() {
        super();
    }
}
//EOF

/*
 * $Workfile: InvalidAddressException.java $
 * $Revision: 1.1 $
 * $Author: edaugherty $
 * $Date: 2003/10/01 19:30:24 $
 */
package com.ericdaugherty.mail.server.errors;

/**
 * Defines an exception when something required was not found.
 *
 * @author Eric Daugherty
 */
public class NotFoundException extends Exception {

    public NotFoundException()
    {
        super();
    }

    public NotFoundException( String message )
    {
        super( message );
    }
}
}
```

2.11 EmailAddress

```
/* *****  
 * $Workfile: EmailAddress.java $  
 * $Revision: 1.2 $  
 * $Author: edaugherty $  
 * $Date: 2003/10/01 19:30:24 $  
 *  
 * $Edited by:Axel Anders Kvale$  
 *  
***** */  
  
package com.ericdaugherty.mail.server.info;  
  
//Java imports  
import java.io.Serializable;  
  
//Local imports  
import com.ericdaugherty.mail.server.errors.InvalidAddressException;  
  
/**  
 * Represents a full email address, including username and domain. This  
 * class performs conversions between a full email address, and a username  
 * and domain.  
 */  
public class EmailAddress implements Serializable {  
  
    // *****  
    // Public Interface  
    // *****  
  
    // *****  
    // Constructor(s)  
  
    /**  
     * Creates an empty email address. This is possible form  
     * SMTP messages that have no MAIL FROM address.  
     */  
    public EmailAddress() {  
  
        _isEmpty = true;  
    }  
  
    /**  
     * Creates a new instance of this class using a single string  
     * that contains the full email address (joe@mydomain.com).  
     */  
    public EmailAddress( String fullAddress ) throws InvalidAddressException {  
  
        setFullAddress( fullAddress );  
    }  
  
    /**  
     * Creates a new instance of this class using a username string  
     * and an address string.  
     */  
    public EmailAddress( String username, String domain ) {  
        setUsername( username );  
        setDomain( domain );  
    }  
  
    /**  
     * Override toString to return the full address  
     */  
    public String toString() {  
        return getAddress();  
    }  
  
    // *****  
    //JavaBean Methods  
  
    public String getUsername(){  
        if( _isEmpty ) {  
            return "";  
        }  
        else {  

```

```

        return _username;
    }
}

public void setUsername(String username) {
    _isEmpty = false;
    _username = username.trim().toLowerCase();
}

public String getDomain(){
    if( _isEmpty ) {
        return "";
    }
    else {
        return _domain;
    }
}

public void setDomain(String domain){
    _isEmpty = false;
    _domain = domain.trim().toLowerCase();
}

public String getAddress() {
    return getFullAddress( getUsername(), getDomain() );
}

public void setAddress( String fullAddress ) throws InvalidAddressException {
    setFullAddress( fullAddress );
}

/*****
// Private Interface
*****/

/**
 * Combines a username and domain into a single email address.
 */
private String getFullAddress( String username, String domain ) {

    if( _isEmpty ) {
        return "";
    }
    else {
        StringBuffer fullAddress = new StringBuffer( username );
        fullAddress.append( "@" );
        fullAddress.append( domain );

        return fullAddress.toString();
    }
}

/**
 * Parses a full address into a username and password for storage.
 */
private void setFullAddress( String fullAddress ) throws InvalidAddressException {

    //Parse toAddress into username and domain.
    int index = fullAddress.indexOf( "@" );
    if( index == -1 ) {
        throw new InvalidAddressException();
    }

    setUsername( fullAddress.substring( 0, index ) );
    setDomain( fullAddress.substring( index + 1 ) );

    _isEmpty = false;
}

/*****
// Variables
*****/

private String _username = "";
private String _domain = "";
private boolean _isEmpty = false;
}

```

2.12 Message

```

/*****
 * $Workfile: Message.java $
 * $Revision: 1.2 $
 * $Author: edaugherty $
 * $Date: 2003/10/01 19:30:24 $
 *
 * $Edited by:Axel Anders Kvale$
 *
 *****/

package com.ericdaugherty.mail.server.info;

import java.io.File;

/**
 * Simple bean class that represents a POP3 Message used in the User class
 * and POP3 Service.
 *
 * @author Eric Daugherty
 */
public class Message {

    private File messageLocation;
    private boolean deleted = false;

    public File getMessageLocation(){ return messageLocation; }

    public void setMessageLocation(File messageLocation){ this.messageLocation =
messageLocation; }

    public long getMessageSize(){ return messageLocation.length(); }

    public boolean isDeleted(){ return deleted; }

    public void setDeleted(boolean deleted){
        this.deleted = deleted;
    }

    public String getUniqueId() {
        String location = messageLocation.getAbsolutePath();

        int begin = location.lastIndexOf( "pop" ) + 3;
        int end = location.lastIndexOf( ".jmsg" );

        return location.substring( begin, end );
    }
}

```

2.13 User

```

/*****
 * $Workfile: User.java $
 * $Revision: 1.5 $
 * $Author: edaugherty $
 * $Date: 2003/11/24 21:09:26 $
 *
 * $Edited by:Axel Anders Kvale$
 *
 *****/

package com.ericdaugherty.mail.server.info;

import java.io.File;

import com.ericdaugherty.mail.server.configuration.ConfigurationManager;
import com.ericdaugherty.mail.server.configuration.PasswordManager;

/**
 * Represents a user object. This class is responsible for providing
 * all information about a specific user and their mailbox.
 *
 * @author Eric Daugherty
 */
public class User {

    private String username;
    private String domain;
    private String password;
    private EmailAddress[] forwardAddresses;

    private Message[] messages = null;

    private ConfigurationManager configurationManager = null;

    /**
     * Creates a new user with the full username (user and domain).
     *
     * @param address User's full email address
     * @param configurationManager Reference to the
     * ConfigurationManager
     */
    public User( EmailAddress address,
                ConfigurationManager configurationManager )
    {
        username = address.getUsername().trim().toLowerCase();
        domain = address.getDomain().trim().toLowerCase();

        this.configurationManager = configurationManager;
    }

    /**
     * Creates a new user with the full username (user and domain).
     *
     * @param address User's full email address
     */
    public User( EmailAddress address )
    {
        this( address, ConfigurationManager.getInstance() );
    }

    /**
     * Returns true if and only if the specified plain text
     * password's hash
     * value matches the hashed password for this user.
     *
     * @param plainTextPassword the password to validate.
     * @return true if it matches.
     */
    public boolean isPasswordValid( String plainTextPassword )
    {
        boolean result =
            getPassword().equals(
                PasswordManager.encryptPassword(
                    plainTextPassword ) );
    }
}

```

```

        return result;
    }

    public String getUsername(){ return username; }

    public void setUsername(String username){
        this.username = username; }

    public String getFullUsername() {
        return getFullUsername( username, domain ); }

    public String getDomain(){ return domain; }

    public void setDomain(String domain){ this.domain = domain; }

    public String getPassword(){ return password; }

    public void setPassword(String password){
        this.password = password; }

    public EmailAddress[] getForwardAddresses(){
        return forwardAddresses; }

    public void setForwardAddresses(EmailAddress[] forwardAddresses){
        this.forwardAddresses = forwardAddresses; }

    /**
     * Returns an array of Strings that represent email addresses to
     * deliver email to this user to.  If the forwardAddresses is not
     * null or empty, this will return the forwardAddresses array.
     * Otherwise, this will return the user's email address.
     *
     * @return array of strings that represent email addresses.
     */
    public EmailAddress[] getDeliveryAddresses() {

        if( forwardAddresses != null && forwardAddresses.length > 0 ) {
            return forwardAddresses;
        }
        else {
            return new EmailAddress[] {
                new EmailAddress( getUsername(), getDomain() ) };
        }
    }

    /**
     * Returns an array of Message objects that represents all messages
     * stored for this user.
     */
    public Message[] getMessages() {

        if( messages == null ) {

            File directory = getUserDirectory();

            String[] fileNames = directory.list();

            int numMessage = fileNames.length;

            messages = new Message[numMessage];
            Message currentMessage;

            for( int index = 0; index < numMessage; index++ ) {
                currentMessage = new Message();
                currentMessage.setMessageLocation(
                    new File( directory, fileNames[index] ) );
                messages[index ] = currentMessage;
            }
        }
        return messages;
    }

    /**
     * Gets the specified message.  Message numbers are 1 based.
     * This method counts on the calling method to verify that the
     * messageNumber actually exists.
     */

```

```

public Message getMessage( int messageNumber ) {
    return getMessages()[messageNumber - 1];
}

/**
 * Gets the total number of messages currently stored for this
 * user.
 */
public long getNumberOfMessage() {
    return getMessages().length;
}

/**
 * Gets the total size of the messages currently stored for this
 * user.
 */
public long getSizeOfAllMessage() {
    Message[] message = getMessages();

    long totalSize = 0;

    for ( int index = 0; index < message.length; index++) {
        totalSize += message[index].getMessageLocation().length();
    }

    return totalSize;
}

/**
 * Gets the user's directory as a file. This method also verifies
 * that that directory exists.
 */
public File getUserDirectory() {

    String mailDirectory = configurationManager.getMailDirectory();
    File directory =
        new File(
            mailDirectory + File.separator +
                "users" + File.separator + getFullUsername() );

    if ( !directory.exists() ) {
        directory.mkdirs();
    }

    if( !directory.isDirectory() ) {
        throw new RuntimeException( "User's Directory path: " +
            directory.getAbsolutePath() + " is not a directory!" );
    }

    return directory;
}

/**
 * This method removes any cached message information this user may
 * have stored
 */
public void reset() {
    messages = null;
}

/**
 * Converts a username and domain to the combined username.
 */
private static String getFullUsername( String username,
    String domain ) {

    return username + "@" + domain;
}
}

```

2.14 ConnectionProcessor

```
/*
 * $Workfile: ConnectionProcessor.java $
 * $Revision: 1.1 $
 * $Author: edaugherty $
 * $Date: 2003/01/23 22:41:28 $
 *
 * $Edited by: Axel Anders Kvale
 *
 */
package com.ericdaugherty.mail.server.services.general;

//Java imports
import java.net.ServerSocket;

/**
 * Defines the interface for all classes that will handle a connection.
 * This interface is used by ServiceListener to interact with the
 * Connection Processors.
 *
 * @author Eric Daugherty
 */
public interface ConnectionProcessor extends Runnable {

    /**
     * Sets the socket used to communicate with the client.
     */
    public void setSocket( ServerSocket serverSocket );
}

```

2.15 DeliveryService

```
*****
* $Workfile: DeliveryService.java $
* $Revision: 1.3 $
* $Author: edaugherty $
* $Date: 2003/10/15 19:01:54 $
*
* $Edited by: Axel Anders Kvale
*
*****/

package com.ericdaugherty.mail.server.services.general;

//Java imports
import java.util.*;

//Local imports
import com.ericdaugherty.mail.server.info.EmailAddress;
import com.ericdaugherty.mail.server.configuration.ConfigurationParameterContants;
import com.ericdaugherty.mail.server.configuration.ConfigurationManager;

/**
 * Handles the evaluation of general mail delivery rules, including SMTP
 * Relaying.
 */
public class DeliveryService implements ConfigurationParameterContants {

    /** Singleton Instance */
    private static DeliveryService instance = null;

    private ConfigurationManager configurationManager;

    /** The IP Addresses that have logged into the POP3 server recently */
    private Hashtable authenticatedIps;

    /** The mailboxes that are currently locked */
    private Hashtable lockedMailboxes;

    /**
     * Load the paramters from the Mail configuration.
     */
    protected DeliveryService() {

        configurationManager = ConfigurationManager.getInstance();

        //Initialize the Hashtable for tracking authenticated ip addresses.
        authenticatedIps = new Hashtable();
        //Initialize the Hashtable for tracking locked mailboxes
        lockedMailboxes = new Hashtable();
    }

    /**
     * Accessor for the singleton instance for this class.
     */
    public static synchronized DeliveryService getDeliveryService(){
        if ( instance == null ) {
            instance = new DeliveryService();
        }
        return instance;
    }

    /**
     * Determines if the domain for the specified email address is
     * hosted
     * locally in this mail server.
     */
    public boolean isLocalAddress( EmailAddress address ) {

        return configurationManager.isLocalDomain(
            address.getDomain() );
    }

    /**

```

```

* Checks an address to see if we should accept it for delivery.
*/
public boolean acceptAddress( EmailAddress address,
                             String clientId ) {

    // Check to see if the email should be address should be accepted
    // for delivery.
    boolean isValid = false;

    // Set isValid to true if one of the rules matches.
    isValid = isLocalAddress( address ) || // Accept all local email.
        ( configurationManager.isEnabledPOPBeforeSMTP() &&
          isAuthenticated( clientId ) )
        ||
        ( isRelayApproved( clientId,
          configurationManager.getRelayApprovedIpAddresses() ) );
    return isValid;
}

/**
 * This method should be called whenever a client authenticates themselves.
 */
public void ipAuthenticated( String clientId ) {
    authenticatedIps.put( clientId, new Date() );
}

/**
 * This method locks a mailbox so that two clients can not access the same mailbox
 * at the same time.
 */
public void lockMailbox( EmailAddress address ) {
    lockedMailboxes.put( address.getAddress(), " " );
}

/**
 * Checks to see if a user currently has the specified mailbox locked.
 */
public boolean isMailboxLocked( EmailAddress address ) {
    return lockedMailboxes.containsKey( address.getAddress() );
}

/**
 * Unlocks an mailbox.
 */
public void unlockMailbox( EmailAddress address ) {
    lockedMailboxes.remove( address.getAddress() );
}

/**
 * Checks the current state to determine if a user from this
 * IP address has authenticated with the POP3 server with the
 * timeout length.
 */
private boolean isAuthenticated( String clientId ) {

    boolean retval = false;

    //Do a quick check to see if this ip is even registered
    //in the HashTable.
    if( authenticatedIps.containsKey( clientId ) ) {
        Date authenticationDate =
            (Date) authenticatedIps.get( clientId );

        //Calculate the current time and the time that the login
        //will timeout.
        long currentTime = System.currentTimeMillis();
        long timeoutTime =
            authenticationDate.getTime() +
            configurationManager.getAuthenticationTimeoutMilliseconds();

        //If the timeout time is in the future, the ip is still
        //authenticated.
        if( timeoutTime > currentTime ) {
            retval = true;
        }
        else {
            //If the IP address has timed out, remove it from the

```

```

        //hashtable.
        authenticatedIps.remove( clientIp );
    }
}
return retval;
}

/**
 * Returns true if the client IP address matches an IP address in the
 * approvedAddresses array.
 *
 * @param clientIp The IP address to test.
 * @param approvedAddresses The approved list.
 * @return true if the address is approved.
 */
private boolean isRelayApproved( String clientIp,
                                String[] approvedAddresses ) {

    String approvedAddress;
    for( int index = 0; index < approvedAddresses.length; index++ ) {
        approvedAddress = approvedAddresses[index];
        // Check for an exact match.
        if( clientIp.equals( approvedAddress ) ) {
            return true;
        }
        // Check for a partial match
        else {
            int wildcardIndex = approvedAddress.indexOf( "*" );
            if( wildcardIndex != -1 ) {
                boolean isMatch = true;
                StringTokenizer clientIpTokenizer =
                    new StringTokenizer( clientIp, "." );
                StringTokenizer approvedAddressTokenizer =
                    new StringTokenizer( approvedAddress, "." );
                String clientIpToken;
                String approvedAddressToken;
                while( clientIpTokenizer.hasMoreTokens() )
                {
                    try {
                        clientIpToken =
                            clientIpTokenizer.nextToken().trim();
                        approvedAddressToken =
                            approvedAddressTokenizer.nextToken().trim();
                        if( !clientIpToken.equals(
                            approvedAddressToken) &&
!approvedAddressToken.equals( "*" ) ) {
                            isMatch = false;
                            break;
                        }
                    }
                    catch (NoSuchElementException noSuchElementException) {
                        isMatch = false;
                        break;
                    }
                }
                // Return true if you had a match.
                if (isMatch) return true;
            }
        }
    }
    return false;
}
}

```

2.16 DnsService

```
/*
 * $Workfile: DnsService.java $
 * $Revision: 1.2 $
 * $Author: edaugherty $
 * $Date: 2003/10/01 19:30:24 $
 *
 * $Edited by: Axel Anders Kvale
 */
*****

package com.ericdaugherty.mail.server.services.general;

//DNSLib imports
import org.xbill.DNS.*;

/**
 * This class provides an interface to DNS Lookup implementations.
 * <p>
 * The current version of this class depends on the org.xbill.DNS libraries
 * to perform DNS Lookups.
 *
 * @author Eric Daugherty
 */
public class DnsService {

    /**
     * This method returns an array of addresses that represent MX Entries
     * for the specified domain. If no MX entries were found for the
     * specified domain, and empty array is returned.
     *
     * @param domain the domain perform the MX Lookup on.
     */
    public static String[] getMXEntries( String domain ) {

        Record[] records = dns.getRecords( domain, Type.MX );

        //Just return an empty array if no domains were found.
        if( records == null ) {
            return new String[0];
        }

        int numEntries = records.length;

        String[] addresses = new String[numEntries];

        for( int index = 0; index < numEntries; index++ ) {

            addresses[index] = ( MXRecord)records[index] ).getTarget().toString();
        }

        return addresses;
    }
}

```

2.17 ServiceListener

```
*****
* $Workfile: ServiceListener.java $
* $Revision: 1.3 $
* $Author: edaugherty $
* $Date: 2003/11/08 01:13:42 $
*
* $Edited by: Axel Anders Kvale
*
*****/

package com.ericdaugherty.mail.server.services.general;

import java.net.*;
import java.io.*;

import com.ericdaugherty.mail.server.configuration.ConfigurationManager;

/**
 * This class listens for incoming connections on the specified port
 * and starts a new thread to process the request. This class
 * abstracts common functionality required to start any type of service
 * (POP3 or SMTP), reducing the requirement to duplicate this code
 * in each package.
 *
 * @author Eric Daugherty
 */
public class ServiceListener implements Runnable {

    /** Array of processors */
    private ConnectionProcessor[] processors;

    /** The port to listen on for incoming connections. */
    private int port;

    /** The type of class to use to handle requests. */
    private Class connectionProcessorClass;

    /** The number of threads to create to listen on this port */
    private int threads;

    /** Thread pool */
    private Thread[] threadPool = null;

    /** server socket */
    private ServerSocket serverSocket;

    /**
     * Creates a new instance and stores the initial paramters.
     */
    public ServiceListener( int port,
                           Class connectionProcessorClass,
                           int threads ) {

        this.port = port;
        this.connectionProcessorClass = connectionProcessorClass;
        this.threads = threads;
    }

    /**
     * Entry point for the thread. Listens for incoming connections and
     * start a new handler thread for each.
     */
    public void run() {

        InetAddress listenAddress =
            ConfigurationManager.getInstance().getListenAddress();
        try {
            if( listenAddress == null ) {
                // listen to the given port
                serverSocket = new ServerSocket( port );
            }
            else {
                // 50 is the default backlog size.
                serverSocket = new ServerSocket( port, 50, listenAddress );
            }
        }
    }
}
```

```

    }
    catch (IOException e) {return;}

    ConnectionProcessor processor;
    long threadCount = 0;
    String threadNameBase = Thread.currentThread().getName();

    //Initialize threadpools.
    try {

        processors = new ConnectionProcessor[ threads ];
        threadPool = new Thread[ threads ];

        for( int index = 0; index < threads; index++ ) {
            //Create the handler now to speed up connection time.
            processor =
                (ConnectionProcessor) connectionProcessorClass.newInstance();
            processors[index] = processor;

            processor.setSocket( serverSocket );

            //Create, name, and start a new thread to handle this request.
            threadPool[index] =
                new Thread(processor, threadNameBase + ":" + ++threadCount );
            threadPool[index].start();
        }
    }
    catch (Exception e){}
}
}

```

2.18 Pop3Processor

```
/* *****
 * $Workfile: Pop3Processor.java $
 * $Revision: 1.4 $
 * $Author: edaugherty $
 * $Date: 2003/11/15 21:40:30 $
 *
 * $Edited by: Axel Anders Kvale
 *
 * *****/

package com.ericdaugherty.mail.server.services.pop3;

import java.net.*;
import java.io.*;

import com.ericdaugherty.mail.server.info.*;
import com.ericdaugherty.mail.server.services.general.DeliveryService;
import com.ericdaugherty.mail.server.services.general.ConnectionProcessor;
import com.ericdaugherty.mail.server.configuration.ConfigurationManager;

/**
 * Handles an incoming Pop3 connection. See rfc 1939 for details.
 *
 * @author Eric Daugherty
 */
public class Pop3Processor extends Thread implements ConnectionProcessor {
    /** The ConfigurationManager */
    private static ConfigurationManager configurationManager =
        ConfigurationManager.getInstance();

    /** Indicates if this thread should continue to run or shut down */
    private boolean running = true;

    /** The server socket used to listen for incoming connections */
    private ServerSocket serverSocket;

    /** Socket connection to the client */
    private Socket socket;

    /** The IP address of the client */
    private String clientIp;

    /** The user currently logged in */
    private User user = null;

    /** Writer to sent data to the client */
    private PrintWriter out;
    /** Reader to read data from the client */
    private BufferedReader in;

    /**
     * Sets the socket used to communicate with the client.
     */
    public void setSocket( ServerSocket serverSocket ) {

        this.serverSocket = serverSocket;
    }

    /**
     * Entrypoint for the Thread, this method handles the interaction with
     * the client socket.
     */
    public void run() {

        try {
            //Set the socket to timeout every 10 seconds so it does not
            //just block forever.
            serverSocket.setSoTimeout( 1000 );
        }
        catch( SocketException se ) {}

        while( running ) {
            try {
                socket = serverSocket.accept();
            }
        }
    }
}
```

```

//Prepare the input and output streams.
out = new PrintWriter(socket.getOutputStream(), true);
in = new BufferedReader(new InputStreamReader( socket.getInputStream() ));

InetAddress remoteAddress = socket.getInetAddress();
clientIp = remoteAddress.getHostAddress();

//Output the welcome message.
write( WELCOME_MESSAGE );

//Forces the client to property authenticate.
user = authenticate();
user.reset();

//Parses the input for commands and delegates to the appropriate methods.
handleCommands();
}
catch( SocketTimeoutException ste ) {
    //This is fine, it should time out every 10 seconds if
    //a connection is not made.
}
//If any exception gets to here uncaught, it means we should just disconnect.
catch( Throwable e ) {

    //Unlock the user's mailbox
    if( user != null ) {
        EmailAddress userAddress = new EmailAddress( user.getUsername(),
user.getDomain() );
        DeliveryService.getDeliveryService().unlockMailbox( userAddress );
    }

    try {
        write( MESSAGE_DISCONNECT );
    }
    catch( Exception e1 ) { //Nothing to do.}
    try {
        if( socket != null ) {
            socket.close();
        }
    }
    catch( IOException ioe ) { //Nothing to do.}
}
}
}

/**
 * Checks to verify that the command is not a quit command. If it is,
 * the current state is finalized (all messages marked as deleted are
 * actually deleted) and closes the connection.
 */
private void checkQuit( String command ) {

    if( command.equals( COMMAND_QUIT ) ) {

        //Delete the messages marked as deleted from disk
        if( user != null ) {
            Message[] messages = user.getMessages();
            int numMessage = messages.length;
            Message currentMessage = null;

            for( int index = 0; index < numMessage; index++ ) {
                currentMessage = messages[index];
                if( currentMessage.isDeleted() ) {
                    messages[index].getMessageLocation().delete();
                }
            }
        }

        // TODO Find a better way to handle user logoffs.
        throw new RuntimeException();
    }
}

/**
 * The user must authenticate before moving on to enter
 * more commands. This method will listen to incoming

```

```

* commands until the user either successfully authenticates
* or quits.
*/
private User authenticate() {

    //Reusable Variables.
    String inputString;
    String command;
    String argument;

    String username = "";
    String domain = "";
    String password = "";
    EmailAddress address = null;
    DeliveryService deliveryService = DeliveryService.getDeliveryService();

    //Get the username from the client.
    boolean userAccepted = false;

    while( !userAccepted ) {
        inputString = read();

        command = parseCommand( inputString );
        argument = parseArgument( inputString );

        //Check to see if they sent the user command.
        if( command.equals( COMMAND_USER ) ) {

            //Make sure they sent a username
            if( argument.equals( "" ) ) {
                write( MESSAGE_TOO_FEW_ARGUMENTS );
            }
            else {
                int atIndex = argument.indexOf( "@" );

                //Verify that the username contains the domain.
                if( atIndex == -1 ) {
                    write( MESSAGE_NEED_USER_DOMAIN );
                }
                else {
                    //Accept the user, and proceed to get the password.
                    username = argument.substring( 0, atIndex );
                    domain = argument.substring( atIndex + 1 );

                    address = new EmailAddress( username, domain );

                    //Check to see if the user's mailbox is locked
                    if( deliveryService.isMailboxLocked( address ) ) {
                        write( MESSAGE_USER_MAILBOX_LOCKED );
                    }
                    else {
                        write( MESSAGE_USER_ACCEPTED + argument );
                        userAccepted = true;
                    }
                }
            }
        }
        else {
            write( MESSAGE_INVALID_COMMAND + command );
        }
    }

    //The user has been accepted, now get the password.
    boolean passwordAccepted = false;

    while( !passwordAccepted ) {
        inputString = read();

        command = parseCommand( inputString );
        argument = parseArgument( inputString );

        //Check to see if they sent the user command.
        if( command.equals( COMMAND_PASS ) ) {

            //Make sure they sent a password
            if( argument.equals( "" ) ) {
                write( MESSAGE_TOO_FEW_ARGUMENTS );
            }
        }
    }
}

```

```

        }
        else {
            password = argument;
            passwordAccepted = true;
        }
    }
    else {
        write( MESSAGE_INVALID_COMMAND + command );
    }
}

User user = configurationManager.getUser( address );
if( user != null && user.isPasswordValid( password ) )
{
    deliveryService.ipAuthenticated( clientIp );
    deliveryService.lockMailbox( address );
    write( MESSAGE_LOGIN_SUCCESSFUL );
    return user;
}
else
{
    //The login failed, display a message to the user and disconnect.
    write( MESSAGE_INVALID_LOGIN + username );
    throw new RuntimeException();
}
}

/**
 * Handles all the commands related the the retrieval of mail.
 */
private void handleCommands() {

    //Reusable Variables.
    String inputString;
    String command;
    String argument;

    //This just runs until a SystemException is thrown, which
    //signals us to disconnect.
    while( true ) {

        inputString = read();

        command = parseCommand( inputString );
        argument = parseArgument( inputString );

        //Identify the command and call the appropriate helper method.
        if( command.equals( COMMAND_STAT ) ) {
            handleStat();
        }
        else if( command.equals( COMMAND_LIST ) ) {
            handleList( argument );
        }
        else if( command.equals( COMMAND_RETR ) ) {
            handleRetr( argument );
        }
        else if( command.equals( COMMAND_DELE ) ) {
            handleDele( argument );
        }
        else if( command.equals( COMMAND_NOOP ) ) {
            write( "+OK" );
        }
        else if( command.equals( COMMAND_RSET ) ) {
            handleRset();
        }
        else if( command.equals( COMMAND_TOP ) ) {
            handleTop( argument );
        }
        else if( command.equals( COMMAND_UIDL ) ) {
            handleUidl( argument );
        }
        else {
            write( MESSAGE_INVALID_COMMAND + command );
        }
    }
}
}

```

```

/**
 * Handles the 'stat' command, which returns the total number of message
 * and the total size of those message.
 */
private void handleStat() {

    write( "+OK " + user.getNumberOfMessage() + " " + user.getSizeOfAllMessage() );
}

/**
 * Handles the 'list' command, which returns the total number of messages and
 * size along with a list of the individual message sizes.
 */
private void handleList( String argument ) {

    if( argument.equals( "" ) ) {
        long numMessages = user.getNumberOfMessage();
        long sizeMessage = user.getSizeOfAllMessage();

        write( "+OK " + numMessages + " messages ( " + sizeMessage + " octets)" );

        for( int index = 0; index < numMessages; index++ ) {
            write( (index + 1) + " " + user.getMessage( index + 1
).getMessageLocation().length() );
        }
        write( "." );
    }
    else {
        int messageNumber = 0;

        try {
            messageNumber = Integer.parseInt( argument );
        }
        catch( NumberFormatException nfe ) {
            write( MESSAGE_NOT_A_NUMBER );
            return;
        }

        long numMessages = user.getNumberOfMessage();

        if( messageNumber > numMessages || user.getMessage( messageNumber ).isDeleted() )
        {
            write( MESSAGE_NO_SUCH_MESSAGE );
            return;
        }

        write( "+OK " + messageNumber + " " + user.getMessage( messageNumber
).getMessageLocation().length() );
    }
}

/**
 * Sends the specified email message to the client.
 */
private void handleRetr( String argument ) {

    int messageNumber = 0;

    try {
        messageNumber = Integer.parseInt( argument );
    }
    catch( NumberFormatException nfe ) {
        write( MESSAGE_NOT_A_NUMBER );
        return;
    }

    long numMessages = user.getNumberOfMessage();

    if( messageNumber > numMessages || user.getMessage( messageNumber ).isDeleted() ) {
        write( MESSAGE_NO_SUCH_MESSAGE );
        return;
    }

    write( MESSAGE_OK );

    BufferedReader fileIn = null;
    try {

```

```

        //Open an reader to read the file.
        fileIn = new BufferedReader( new FileReader( user.getMessage( messageNumber
).getMessageLocation() ) );

        //Write the file to the client.
        String currentLine = fileIn.readLine();
        while (currentLine != null) {
            write( currentLine );
            currentLine = fileIn.readLine();
        }
        write( "." );
    }
    catch( FileNotFoundException fnfe ) {}
    catch( IOException ioe ) {write( "-ERR Error retrieving message" );}
    finally {
        //Make sure the input stream gets closed.
        try {
            if( fileIn != null ) {
                fileIn.close();
            }
        }
        catch( IOException ioe ) {//Nothing to do...}
    }
}

/**
 * Marks the specified message for deletion. The message will only
 * be deleted if the user later enters the QUIT command, as per the
 * spec.
 */
private void handleDele( String argument ) {

    int messageNumber = 0;

    try {
        messageNumber = Integer.parseInt( argument );
    }
    catch( NumberFormatException nfe ) {
        write( MESSAGE_NOT_A_NUMBER );
        return;
    }

    long numMessages = user.getNumberOfMessage();

    if( messageNumber > numMessages ) {
        write( MESSAGE_NO_SUCH_MESSAGE );
    }
    else if( user.getMessage( messageNumber ).isDeleted() ) {
        write( MESSAGE_ALREADY_DELETED );
    }
    else {
        user.getMessage( messageNumber ).setDeleted( true );
        write( MESSAGE_OK );
    }
}

/**
 * Unmarks all deleted messages.
 */
private void handleRset() {

    Message[] messages = user.getMessages();
    int numMessage = messages.length;

    for( int index = 0; index < numMessage; index++ ) {
        messages[index].setDeleted( false );
    }

    write( MESSAGE_OK );
}

/**
 * Returns the header and first x lines for the
 * specified message.
 */
private void handleTop( String argument ) {

    int messageNumber = 0;

```

```

        int numLines = 0;

        int spaceIndex = argument.indexOf( " " );
        if( spaceIndex == -1 ) {
            write( MESSAGE_TOO_FEW_ARGUMENTS );
            return;
        }
        String arg1 = argument.substring( 0, spaceIndex ).trim();
        String arg2 = argument.substring( spaceIndex + 1 ).trim();

    try {
        messageNumber = Integer.parseInt( arg1 );
        numLines = Integer.parseInt( arg2 );
    }
    catch( NumberFormatException nfe ) {
        write( MESSAGE_NOT_A_NUMBER );
        return;
    }

    long numMessages = user.getNumberOfMessage();

    if( messageNumber > numMessages || user.getMessage( messageNumber ).isDeleted() ) {
        write( MESSAGE_NO_SUCH_MESSAGE );
        return;
    }

    write( MESSAGE_OK );

    BufferedReader fileIn = null;
    try {
        //Open an reader to read the file.
        fileIn = new BufferedReader( new FileReader( user.getMessage( messageNumber
).getMessageLocation() ) );

        //Write the Message Header.
        String currentLine = fileIn.readLine();
        while ( currentLine != null && !currentLine.equals( "" ) ) {
            write( currentLine );
            currentLine = fileIn.readLine();
        }

        //Write an empty line to separate header from body.
        write( currentLine );
        currentLine = fileIn.readLine();

        //Write the requested number of lines from the body of the
        //message, or until the entire message has been written.
        int index = 0;
        while( index < numLines && currentLine != null ) {
            write( currentLine );
            currentLine = fileIn.readLine();
            index++;
        }

        write( "." );
    }
    catch( FileNotFoundException fnfe ) {}
    catch( IOException ioe ) {write( "-ERR Error retrieving message" );}
    finally {
        //Make sure the input stream gets closed.
        try {
            if( fileIn != null ) {
                fileIn.close();
            }
        }
        catch( IOException ioe ) {}
    }
}

/**
 * Returns the unique id of the specified message, or all the unique
 * ids of the non-deleted messages.
 */
private void handleUidl( String argument ) {

    //Return all messages unique ids
    if( argument == null || argument.length() == 0 ) {

```

```

        long numMessages = user.getNumberOfMessage();
        Message message;

        write( MESSAGE_OK );

        //Write out each non-deleted message id.
        for( int index = 0; index < numMessages; index++ ) {
            message = user.getMessage( index + 1 );
            if( !message.isDeleted() ) {
                write( (index + 1) + " " + message.getUniqueId() );
            }
        }

        write( "." );
    }
    //Output a single messages unique id.
    else {

        int messageNumber = 0;

        try {
            messageNumber = Integer.parseInt( argument );
        }
        catch( NumberFormatException nfe ) {
            write( MESSAGE_NOT_A_NUMBER );
            return;
        }

        long numMessages = user.getNumberOfMessage();

        if( messageNumber > numMessages || user.getMessage( messageNumber
).isDeleted() ) {
            write( MESSAGE_NO_SUCH_MESSAGE );
            return;
        }

        write( MESSAGE_OK + " " + messageNumber + " " + user.getMessage(
messageNumber ).getUniqueId() );
    }
}

/**
 * Reads a line from the input stream and returns it.
 */
private String read() {
    try {
        String inputLine = in.readLine().trim();
        return inputLine;
    }
    catch( IOException ioe ) {throw new RuntimeException();}
}

/**
 * Writes the specified output message to the client.
 */
private void write( String message ) {
    out.print( message + "\r\n" );
    out.flush();
}

/**
 * Parses the input stream for the command. The command is the
 * begining of the input stream to the first space. If there is
 * space found, the entire input string is returned.
 * <p>
 * This method converts the returned command to uppercase to allow
 * for easier comparison.
 * <p>
 * Additionally, this method checks to verify that the quit command
 * was not issued. If it was, a SystemException is thrown to terminate
 * the connection.
 */
private String parseCommand( String inputString ) {

    int index = inputString.indexOf( " " );

```

```

        if( index == -1 ) {
            String command = inputString.toUpperCase();
            checkQuit( command );
            return command;
        }
        else {
            String command = inputString.substring( 0, index ).toUpperCase();
            checkQuit( command );
            return command;
        }
    }

/**
 * Parses the input stream for the argument. The argument is the
 * text starting after the first space until the end of the inputstring.
 * If there is no space found, an empty string is returned.
 * <p>
 * This method does not convert the case of the argument.
 */
private String parseArgument( String inputString ) {

    int index = inputString.indexOf( " " );

    if( index == -1 ) {
        return "";
    }
    else {
        return inputString.substring( index + 1 ).trim();
    }
}

/*****
// Constants
/*****

//Message Constants
//General Message
private static final String WELCOME_MESSAGE = "+OK EricDaugherty's Java Pop Server Ready";
private static final String MESSAGE_DISCONNECT = "+OK Pop server signing off.";
private static final String MESSAGE_OK = "+OK";
private static final String MESSAGE_INVALID_COMMAND = "-ERR Unknown command: ";
private static final String MESSAGE_TOO_FEW_ARGUMENTS = "-ERR Too few arguments for this
command.";

//Authentication Messages
private static final String MESSAGE_NEED_USER_DOMAIN = "-ERR User names must contain the
username and domain. ex: \"root@mydomain.com\"";
private static final String MESSAGE_USER_ACCEPTED = "+OK Password required for ";
private static final String MESSAGE_LOGIN_SUCCESSFUL = "+OK Login successful";
private static final String MESSAGE_USER_MAILBOX_LOCKED = "-ERR User's Mailbox is locked";
private static final String MESSAGE_INVALID_LOGIN = "-ERR Password supplied is incorrect
for user: ";

//Other Messages
private static final String MESSAGE_NOT_A_NUMBER = "-ERR Command requires a valid number
as an argument.";
private static final String MESSAGE_NO_SUCH_MESSAGE = "-ERR No such message.";
private static final String MESSAGE_ALREADY_DELETED = "-ERR Message already deleted.";

//Command Constants
private static final String COMMAND_QUIT = "QUIT";
private static final String COMMAND_USER = "USER";
private static final String COMMAND_PASS = "PASS";
private static final String COMMAND_STAT = "STAT";
private static final String COMMAND_LIST = "LIST";
private static final String COMMAND_RETR = "RETR";
private static final String COMMAND_DELE = "DELE";
private static final String COMMAND_NOOP = "NOOP";
private static final String COMMAND_RSET = "REST";
private static final String COMMAND_TOP = "TOP";
private static final String COMMAND_UIDL = "UIDL";
}

```

2.19 SMTPMessage

```
*****
* $Workfile: SMTPMessage.java $
* $Revision: 1.4 $
* $Author: edaugherty $
* $Date: 2003/12/24 02:26:35 $
*
* $Edited by: Axel Anders Kvale
*
*****/

package com.ericdaugherty.mail.server.services.smtp;

//Java imports
import java.io.*;
import java.util.Date;
import java.util.List;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
import java.util.StringTokenizer;

//Local imports
import com.ericdaugherty.mail.server.configuration.ConfigurationManager;
import com.ericdaugherty.mail.server.info.EmailAddress;
import com.ericdaugherty.mail.server.errors.InvalidAddressException;

/**
 * Bean class used to store incoming SMTP message on disk (via Java Serialization)
 * for delivery by the SMTPSender thread.
 *
 * @author Eric Daugherty
 */
public class SMTPMessage implements Serializable {

    private static final String DELIMITER = "\r\n";

    private static final String FILE_VERSION = "1.0";

    /** The ConfigurationManager */
    private static ConfigurationManager configurationManager =
ConfigurationManager.getInstance();

    private Date timeReceived;
    private Date scheduledDelivery;
    private int deliveryAttempts;
    private EmailAddress fromAddress;
    private List toAddresses = new ArrayList();
    private List dataLines = new ArrayList();
    private File messageLocation = null;

    /**
     * Initializes a new message with the current time.
     */
    public SMTPMessage() {
        Date now = new Date();
        timeReceived = now;
        scheduledDelivery = now;
        deliveryAttempts = 0;
    }

    public Date getTimeReceived() {
        return timeReceived;
    }

    public void setTimeReceived(Date timeReceived) {
        this.timeReceived = timeReceived;
    }

    public Date getScheduledDelivery() {
        return scheduledDelivery;
    }

    public void setScheduledDelivery(Date scheduledDelivery) {
```

```

        this.scheduledDelivery = scheduledDelivery;
    }

    public int getDeliveryAttempts() {
        return deliveryAttempts;
    }

    public void setDeliveryAttempts(int deliveryAttempts) {
        this.deliveryAttempts = deliveryAttempts;
    }

    public EmailAddress getFromAddress(){ return fromAddress; }

    public void setFromAddress(EmailAddress fromAddress){ this.fromAddress = fromAddress; }

    public List getToAddresses() { return toAddresses; }

    public void setToAddresses( List toAddresses ) { this.toAddresses = toAddresses; }

    public void addToAddress( EmailAddress toAddress ) { toAddresses.add( toAddress ); }

    public List getDataLines() { return dataLines; }

    public void addDataLine( String line ) { dataLines.add( line ); }

    public File getMessageLocation(){ return messageLocation; }

    public void setMessageLocation(File messageLocation){ this.messageLocation =
messageLocation; }

    /**
     * Moves the message to the 'failed' Directory.
     */
    public void moveToFailedFolder() throws Exception {
        File failedDir = new File( configurationManager.getMailDirectory() + File.separator +
"failed" );

        // If the directory does not exist, create it.
        if( !failedDir.exists() ) {
            if( !failedDir.mkdirs() )
            {
                throw new Exception( "Unable to create failed Directory." );
            }
        }

        File messageLocation = getMessageLocation();
        String newLocation= configurationManager.getMailDirectory() + File.separator + "failed"
+ File.separator + messageLocation.getName();
        if( !messageLocation.renameTo( new File(newLocation) ) )
        {
            throw new Exception( "moveToFailedFolder failed. Message was not renamed." );
        }
    }

    /**
     * Saves the message to the Mail Spool Directory.
     */
    public void save() throws Exception {

        File smtpDirectory = new File( configurationManager.getMailDirectory() +
File.separator + "smtp" );

        // If the directory does not exist, create it.
        if( !smtpDirectory.exists() ) {
            if( !smtpDirectory.mkdirs() )
            {
                throw new Exception( "Unable to create SMTP Mail Directory." );
            }
        }

        File messageFile = getMessageLocation();

        if( messageFile == null ) {
            messageFile = File.createTempFile( "smtp", ".ser", smtpDirectory );
            setMessageLocation( messageFile );
        }
    }

```

```

FileWriter writer = new FileWriter( messageFile );
try
{
    writer.write( FILE_VERSION );
    writer.write( DELIMITER );
    writer.write( getFromAddress().toString() );
    writer.write( DELIMITER );
    writer.write( flattenAddresses( getToAddresses() ) );
    writer.write( DELIMITER );
    writer.write( String.valueOf( getTimeReceived().getTime() ) );
    writer.write( DELIMITER );
    writer.write( String.valueOf( getScheduledDelivery().getTime() ) );
    writer.write( DELIMITER );
    writer.write( String.valueOf( getDeliveryAttempts() ) );
    writer.write( DELIMITER );
    List dataLines = getDataLines();
    for( int index = 0; index < dataLines.size(); index++ )
    {
        writer.write( (String) dataLines.get( index ) );
        writer.write( DELIMITER );
    }
}
finally
{
    try
    {
        if( writer != null )
        {
            writer.close();
        }
    }
    catch( IOException e ){}
}
}

/**
 * Loads an individual message from disk.
 *
 * @param filename the filename of the message.
 * @throws IOException thrown if there is any IO error while reading the message.
 */
public static SMTPMessage load( String filename ) throws Exception {

    File messageFile = new File( filename );
    FileReader fileReader = new FileReader( messageFile );
    BufferedReader reader = new BufferedReader( fileReader );

    try
    {
        String version = reader.readLine();
        if( !FILE_VERSION.equals( version ) )
        {
            throw new IOException( "Invalid file version: " + version );
        }
        // Initialize a new message with the right file location
        SMTPMessage message = new SMTPMessage();
        message.setMessageLocation( messageFile );

        // Load each variable
        message.setFromAddress( new EmailAddress( reader.readLine() ) );
        message.setToAddresses( inflateAddresses( reader.readLine() ) );
        message.setTimeReceived( new Date( Long.parseLong( reader.readLine() ) ) );
        message.setScheduledDelivery( new Date( Long.parseLong( reader.readLine() ) ) );
        message.setDeliveryAttempts( Integer.parseInt( reader.readLine() ) );

        String inputLine = reader.readLine();
        while( inputLine != null )
        {
            message.addDataLine( inputLine );
            inputLine = reader.readLine();
        }

        return message;
    }
    catch( InvalidAddressException invalidAddressException )
    {
        throw new IOException( "Unable to parse the address from the stored file." );
    }
}

```

```

    }
    catch( NumberFormatException numberFormatException )
    {
        throw new IOException( "Unable to parse the data from the stored file into a
number. " + numberFormatException.toString() );
    }
    finally
    {
        if( reader != null )
        {
            reader.close();
        }
    }
}

/**
 * Converts a <code>List</code> of <code>EmailAddress</code>
 * instances into a comma delimited string.
 *
 * @param addresses Collection of Address instances.
 * @return Comma delimited String of the addresses.
 */
private static String flattenAddresses( Collection addresses )
{
    StringBuffer toAddresses = new StringBuffer();
    EmailAddress address;
    Iterator addressIterator = addresses.iterator();
    while( addressIterator.hasNext() )
    {
        address = (EmailAddress) addressIterator.next();
        toAddresses.append( address.toString() );
        toAddresses.append( "," );
    }

    // Remove the last comma.
    toAddresses.deleteCharAt( toAddresses.length() - 1 );

    return toAddresses.toString();
}

/**
 * Converts a comma delimited string of addresses into a
 * <code>List</code> of <code>EmailAddress</code> instances.
 *
 * @param addresses Comma delimited String of addresses.
 * @return List of Address instances.
 */
private static List inflateAddresses( String addresses )
{
    StringTokenizer addressTokenizer = new StringTokenizer( addresses, "," );
    List addressList = new ArrayList();
    EmailAddress address;

    try
    {
        while( addressTokenizer.hasMoreTokens() )
        {
            address = new EmailAddress( addressTokenizer.nextToken() );
            addressList.add( address );
        }

        return addressList;
    }
    catch( InvalidAddressException invalidAddressException )
    {
        throw new RuntimeException( "Error parsing address. Message Delivery Failed." );
    }
}
}

```

2.20 SMTPProcessor

```

/*****
 * $Workfile: SMTPProcessor.java $
 * $Revision: 1.11 $
 * $Author: edaugherty $
 * $Date: 2003/11/08 01:14:15 $
 *
 * $Edited by: Axel Anders Kvale
 *
 *****/

package com.ericdaugherty.mail.server.services.smtp;

//Java imports
import java.net.*;
import java.io.*;
import java.util.*;

//Local imports
import com.ericdaugherty.mail.server.info.EmailAddress;
import com.ericdaugherty.mail.server.info.User;
import com.ericdaugherty.mail.server.errors.InvalidAddressException;
import com.ericdaugherty.mail.server.services.general.DeliveryService;
import com.ericdaugherty.mail.server.services.general.ConnectionProcessor;
import com.ericdaugherty.mail.server.configuration.ConfigurationManager;

/**
 * Handles an incoming SMTP connection. See rfc821 for details.
 *
 * @author Eric Daugherty
 */
public class SMTPProcessor implements ConnectionProcessor {

    /** The ConfigurationManager */
    private static ConfigurationManager configurationManager =
ConfigurationManager.getInstance();

    /** Indicates if this thread should continue to run or shut down */
    private boolean running = true;

    /** The server socket used to listen for incoming connections */
    private ServerSocket serverSocket;

    /** Socket connection to the client */
    private Socket socket;

    /** The IP address of the client */
    private String clientIp;

    /** The incoming SMTP Message */
    private SMTPMessage message;

    /** Writer to sent data to the client */
    private PrintWriter out;
    /** Reader to read data from the client */
    private BufferedReader in;

    /**
     * Sets the socket used to communicate with the client.
     */
    public void setSocket( ServerSocket serverSocket ) {

        this.serverSocket = serverSocket;
    }

    /**
     * Entrypoint for the Thread, this method handles the interaction with
     * the client socket.
     */
    public void run() {

        try {

```

```

        //Set the socket to timeout every 10 seconds so it does not
        //just block forever.
        serverSocket.setSoTimeout( 1000 );
    }
    catch( SocketException se ) {}

    while( running ) {
        try {
            socket = serverSocket.accept();

            //Prepare the input and output streams.
            out = new PrintWriter(socket.getOutputStream(), true);
            in = new BufferedReader(new InputStreamReader( socket.getInputStream() ));

            InetAddress remoteAddress = socket.getInetAddress();
            clientIp = remoteAddress.getHostAddress();

            write( WELCOME_MESSAGE );

            //Initialize the input message.
            message = new SMTPMessage();

            //Parses the input for commands and delegates to the appropriate methods.
            handleCommands();

        }
        catch( SocketTimeoutException ste ) {
            //This is fine, it should time out every 10 seconds if
            //a connection is not made.
        }
        //If any exception gets to here uncaught, it means we should just disconnect.
        catch( Throwable e ) {
            try {
                write( MESSAGE_DISCONNECT );
            }
            catch( Exception e1 ) {}
            try {
                if( socket != null ) {
                    socket.close();
                }
            }
            catch( IOException ioe ) {}
        }
    }
}

/**
 * Checks to make sure that the incoming command is not a quit.  If so,
 * the connection is terminated.
 */
private void checkQuit( String command ) {

    if( command.equals( COMMAND_QUIT ) ) {
        throw new RuntimeException();
    }
}

/**
 * Handles all the commands related the the sending of mail.
 */
private void handleCommands() {

    //Reusable Variables.
    String inputString;
    String command;
    String argument;

    int lastCommand = NONE;

    //This just runs until a SystemException is thrown, which
    //signals us to disconnect.
    while( true ) {

        inputString = read();

        command = parseCommand( inputString );
        argument = parseArgument( inputString );
    }
}

```

```

if( command.equals( COMMAND_HELO ) ) {
    write( "250 Hello " + argument );
    lastCommand = HELO;
}

//NOOP - Do Nothing.
else if( command.equals( COMMAND_NOOP ) ) {
    write( MESSAGE_OK );
}

//Resets the state of the server back to the initial
//state.
else if( command.equals( COMMAND_RSET ) ) {
    message = new SMTPMessage();
    write( MESSAGE_OK );
    lastCommand = RSET;
}

//Not only check the command, but the full string, since the prepare command
//method only returns the text before the first string, and this is a two
//word command.
else if( command.equals( COMMAND_MAIL_FROM ) &&
inputString.toUpperCase().startsWith( "MAIL FROM:" ) ) {

    if( lastCommand == HELO || lastCommand == NONE || lastCommand == RSET ) {
        if( handleMailFrom( inputString ) ) {
            lastCommand = MAIL_FROM;
        }
    }
    else {
        write( MESSAGE_COMMAND_ORDER_INVALID );
    }
}

//Not only check the command, but the full string, since the prepare command
//method only returns the text before the first string, and this is a two
//word command.
else if( command.equals( COMMAND_RCPT_TO ) &&
inputString.toUpperCase().startsWith( "RCPT TO:" ) ) {

    if( lastCommand == MAIL_FROM || lastCommand == RCPT_TO ) {
        handleRcptTo( inputString );
        lastCommand = RCPT_TO;
    }
    else {
        write( MESSAGE_COMMAND_ORDER_INVALID );
    }
}

else if( command.equals( COMMAND_DATA ) ) {

    if( lastCommand == RCPT_TO && message.getToAddresses().size() > 0 ) {
        handleData();
        // Reset for another message
        message = new SMTPMessage();
        lastCommand = RSET;
    }
    else {
        write( MESSAGE_COMMAND_ORDER_INVALID );
    }
}

else {
    write( MESSAGE_INVALID_COMMAND + command );
}
}
}

/**
 * Handle the "MAIL FROM:" command, which defines the sending address for
 * this message.
 */
private boolean handleMailFrom( String inputString ) {

    String fromAddress = parseAddress( inputString.substring( 10 ) );

    try {
        //It is legal for the MAIL FROM address to be empty.
        if( fromAddress == null || fromAddress.trim().equals( "" ) ) {
            message.setFromAddress( new EmailAddress() );
        }
        //Although this is the normal case...

```

```

        else {
            EmailAddress address = new EmailAddress( fromAddress );
            message.setFromAddress( address );
        }
        write( MESSAGE_OK );
        return true;
    }
    catch( InvalidAddressException iae ) {
        write( MESSAGE_USER_INVALID );
        return false;
    }
}

/**
 * Handle the "RCPT TO:" command, which defines one of the receiving addresses.
 */
private void handleRcptTo( String inputString ) {

    String toAddress = parseAddress( inputString.substring( 8 ) );

    try {
        EmailAddress address = new EmailAddress( toAddress );
        //Check the address to see if we can deliver it.
        DeliveryService deliveryService = DeliveryService.getDeliveryService();
        if( deliveryService.acceptAddress( address, clientIp ) ) {
            // Check to see if it is a local user. If so, ask to
            // user object for the delivery addresses.
            User localUser = configurationManager.getUser( address );
            if( localUser != null ) {
                EmailAddress[] addresses = localUser.getDeliveryAddresses();
                for( int index = 0; index < addresses.length; index++ ) {
                    message.addToAddress( addresses[index] );
                }
            }
            // Otherwise, just add the address.
            else {
                message.addToAddress( address );
            }
            write( MESSAGE_OK );
        }
        else {
            throw new InvalidAddressException();
        }
    }
    catch( InvalidAddressException iae ) {
        write( MESSAGE_USER_NOT_LOCAL );
        return;
    }
}

/**
 * Accepts the data being written to the socket.
 */
private void handleData() {

    write( MESSAGE_SEND_DATA );

    //Add a timestamp to the message to track when the message arrived.
    message.addDataLine( "X-ReceivedDate: " + new Date() );
    //Add a line to the message to track that the message when through this server.
    message.addDataLine( "Received: by EricDaugherty's JES SMTP " +
configurationManager.getLocalDomains()[0] + " from client: " + clientIp );

    try {
        String inputString = in.readLine();

        while( !inputString.equals( "." ) ) {
            message.addDataLine( inputString );
            inputString = in.readLine();
        }
    }
    catch( IOException ioe ) {
        throw new RuntimeException();
    }

    //Write the message to disk.

```

```

    try {
        message.save();
        write( MESSAGE_OK );
    }
    catch ( Exception se ) {
        write( MESSAGE_SAVE_MESSAGE_ERROR );
        throw new RuntimeException( se.getMessage() );
    }
}

/**
 * Reads a line from the input stream and returns it.
 */
private String read() {
    try {
        String inputLine = in.readLine().trim();
        return inputLine;
    }
    catch( IOException ioe ) {
        throw new RuntimeException();
    }
}

/**
 * Writes the specified output message to the client.
 */
private void write( String message ) {
    out.print( message + "\r\n" );
    out.flush();
}

/**
 * Parses the input stream for the command. The command is the
 * beginning of the input stream to the first space. If there is
 * space found, the entire input string is returned.
 * <p>
 * This method converts the returned command to uppercase to allow
 * for easier comparison.
 * <p>
 * Additionally, this method checks to verify that the quit command
 * was not issued. If it was, a SystemException is thrown to terminate
 * the connection.
 */
private String parseCommand( String inputString ) {

    int index = inputString.indexOf( " " );

    if( index == -1 ) {
        String command = inputString.toUpperCase();
        checkQuit( command );
        return command;
    }
    else {
        String command = inputString.substring( 0, index ).toUpperCase();
        checkQuit( command );
        return command;
    }
}

/**
 * Parses the input stream for the argument. The argument is the
 * text starting after the first space until the end of the inputstring.
 * If there is no space found, an empty string is returned.
 * <p>
 * This method does not convert the case of the argument.
 */
private String parseArgument( String inputString ) {

    int index = inputString.indexOf( " " );

    if( index == -1 ) {
        return "";
    }
    else {
        return inputString.substring( index + 1 );
    }
}

```

```

}

/**
 * Parses an address argument into a real email address. This
 * method strips off any &gt; or &lt; symbols.
 */
private String parseAddress( String address ) {

    int index = address.indexOf( "<" );
    if( index != -1 ) {
        address = address.substring( index + 1 );
    }
    index = address.indexOf( ">" );
    if( index != -1 ) {
        address = address.substring( 0, index );
    }
    return address;
}

/*****
// Constants
/*****

//Message Constants
//General Message
private static final String WELCOME_MESSAGE = "220 Welcome to EricDaugherty's Java SMTP
Server.";
private static final String MESSAGE_DISCONNECT = "221 SMTP server signing off.";
private static final String MESSAGE_OK = "250 OK";
private static final String MESSAGE_COMMAND_ORDER_INVALID = "503 Command not allowed
here.";
private static final String MESSAGE_USER_NOT_LOCAL = "550 User does not exist.";
private static final String MESSAGE_USER_INVALID = "451 Address is invalid.";
private static final String MESSAGE_SEND_DATA = "354 Start mail input; end with
<CRLF>.<CRLF>";
private static final String MESSAGE_SAVE_MESSAGE_ERROR = "500 Error handling message.";
private static final String MESSAGE_INVALID_COMMAND = "500 Command Unrecognized: ";

//Commands
private static final String COMMAND_HELO = "HELO";
private static final String COMMAND_RSET = "RSET";
private static final String COMMAND_NOOP = "NOOP";
private static final String COMMAND_QUIT = "QUIT";
private static final String COMMAND_MAIL_FROM = "MAIL";
private static final String COMMAND_RCPT_TO = "RCPT";
private static final String COMMAND_DATA = "DATA";

//SMTP Commands
public int NONE = 0;
public int HELO = 1;
public int QUIT = 2;
public int MAIL_FROM = 3;
public int RCPT_TO = 4;
public int DATA = 5;
public int DATA_FINISHED = 6;
public int RSET = 7;
}

```

2.21 SMTPRemoteSender

```
/* *****
 * $Workfile: SMTPRemoteSender.java $
 * $Revision: 1.8 $
 * $Author: edaugherty $
 * $Date: 2004/02/10 20:21:03 $
 *
 * $Edited by: Axel Anders Kvale
 *
 * *****/

package com.ericdaugherty.mail.server.services.smtp;

//Java Imports
import java.net.*;
import java.io.*;
import java.util.*;

//DNS imports
import org.xbill.DNS.*;

//Local Imports
import com.ericdaugherty.mail.server.info.EmailAddress;
import com.ericdaugherty.mail.server.configuration.ConfigurationManager;
import com.ericdaugherty.mail.server.errors.NotFoundException;

/**
 * This class handles sending messages to external SMTP servers for delivery.
 *
 * @author Eric Daugherty
 */
public class SMTPRemoteSender {

    /** ConfigurationManager */
    private static ConfigurationManager configurationManager =
    ConfigurationManager.getInstance();

    /** Writer to sent data to the client */
    private PrintWriter out;
    /** Reader to read data from the client */
    private BufferedReader in;

    public SMTPRemoteSender() {

    }

    /** *****
    // Methods

    /**
    * Handles delivery of messages to addresses not handled by this server.
    */
    public void sendMessage( EmailAddress address, SMTPMessage message ) throws
    NotFoundException, RuntimeException {

        //Open the connection to the server.
        Socket socket = connect( address );

        // Set the timeout so reads do not hang forever.
        try
        {
            socket.setSoTimeout( 60 * 1000 );
        }
        catch( SocketException e )
        {
            throw new RuntimeException( "Unable to set the Socket SO Timeout: "+
            e.getMessage() );
        }

        try {
            try {
                //Get the input and output streams.
                out = new PrintWriter( socket.getOutputStream(), true);
                in = new BufferedReader( new InputStreamReader( socket.getInputStream() ));
            }
        }
    }
}
```

```

        //Perform initial commands
        sendIntro( address, message );

        //Send message data
        sendData( message );

        //Close the connection.
        sendClose();
    }
    catch( IOException ioe ) {
        throw new RuntimeException( "IOException occured while talking to remote
domain: " + address.getDomain() );
    }
}
finally {
    if( socket != null ) {
        try {
            socket.close();
        }
        catch( IOException ioe ) {}
    }
}
}

/**
 * Determines the MX entries for this domain and attempts to open
 * a socket. If no connections can be opened, a SystemException is thrown.
 */
private Socket connect( EmailAddress address ) {

    Socket socket = null;

    String[] mxEntries = null;

    String domain = address.getDomain();

    //Check to see if a default smtp server is configured before performing
    //the DNS lookup.
    if( configurationManager.isDefaultSmtpServerEnabled() )
    {
        mxEntries = configurationManager.getDefaultSmtpServers();
    }
    else {
        try
        {
            // Lookup the MX Entries
            Record [] records = new Lookup(domain, Type.MX).run();
            if( records == null )
            {
                records = new Record[0];
            }

            // Convert the MX Entries to strings and sort them in order
            // of priority.
            mxEntries = new String[records.length];
            short priority = 0;
            short nextPriority = Short.MAX_VALUE;
            int mxIndex = 0;
            while( mxIndex < mxEntries.length )
            {
                for (int i = 0; i < records.length; i++) {
                    MXRecord mx = (MXRecord) records[i];
                    if( mx.getPriority() == priority )
                    {
                        mxEntries[mxIndex++] = mx.getTarget().toString();
                        if(mxIndex >= mxEntries.length) break;
                    }
                    else if( mx.getPriority() < nextPriority && mx.getPriority() >
priority )
                    {
                        nextPriority = (short)mx.getPriority();
                    }
                }
                priority = nextPriority;
                nextPriority = Short.MAX_VALUE;
            }
        }
    }
}

```

```

    }
    catch( ParseException e )
    {
        throw new RuntimeException( "ParseException while looking up domain MX
Entry: " + e.getMessage() );
    }
}

for( int index = 0; index < mxEntries.length; index++ ) {

    try {
        socket = new Socket( mxEntries[index], 25 );
        return socket;
    }
    catch( Exception e ) { }
}
throw new RuntimeException( "Could not connect to any SMTP server for domain: " +
domain );
}

/**
 * This method sends all the commands necessary to prepare the remote server
 * to receive the data command.
 */
private void sendIntro( EmailAddress address, SMTPMessage message ) {

    //Check to make sure remote server introduced itself with appropriate message.
    if( !read().startsWith( "220" ) ) {
        throw new RuntimeException( "Error talking to remote Server" );
    }

    //Send HELO command to remote server.
    write( "HELO " + configurationManager.getLocalDomains()[0] );
    if( !read().startsWith( "250" ) ) {
        throw new RuntimeException( "Error talking to remote Server" );
    }

    //Send MAIL FROM: command
    write( "MAIL FROM:<" + message.getFromAddress().getAddress() + ">" );
    if( !read().startsWith( "250" ) ) {
        throw new RuntimeException( "Error talking to remote Server" );
    }

    //Send RCTP TO: command
    write( "RCPT TO:<" + address.getAddress() + ">" );
    if( !read().startsWith( "250" ) ) {
        throw new RuntimeException( "Error talking to remote Server" );
    }
}

/**
 * This method sends the data command and all the message data to the
 * remote server.
 */
private void sendData( SMTPMessage message ) {

    //Send Data command
    write( "DATA" );
    if( !read().startsWith( "354" ) ) {
        throw new RuntimeException( "Error talking to remote Server" );
    }

    //Get the data to write.
    List dataLines = message.getDataLines();
    int numDataLines = dataLines.size();

    //Write the data.
    for( int index = 0; index < numDataLines; index++ ) {
        write( (String) dataLines.get( index ) );
    }

    //Send the command end data transmission.
    write( "." );

    if( !read().startsWith( "250" ) ) {
        throw new RuntimeException( "Error talking to remote Server" );
    }
}

```

```

    }
}

private void sendClose() {

    write( "QUIT" );
    if( !read().startsWith( "221" ) ) {
        throw new RuntimeException( "Error talking to remote Server" );
    }
}

/**
 * Returns the response code generated by the server.
 * This method will handle multi-line responses, but will
 * only log the responses, and discard the text, returning
 * only the 3 digit response code.
 *
 * @return 3 digit response string.
 */
private String read() {
    try {
        String responseCode;

        //Read in the first line. This is the only line
        //we really care about, since the response code
        //must be the same on all lines.
        String inputText = in.readLine();
        if( inputText == null )
        {
            inputText = "";
        }
        else
        {
            inputText = inputText.trim();
        }

        if( inputText.length() < 3 ) {
            throw new RuntimeException( "SMTP Response too short. Aborting
Send. Response: " + inputText );
        }

        //Strip of the response code.
        responseCode = inputText.substring( 0, 3 );

        //Handle Multi-Line Responses.
        while( ( inputText.length() >= 4 ) && inputText.substring( 3, 4
).equals( "-" ) ) {
            inputText = in.readLine().trim();
        }

        return responseCode;
    }
    catch( IOException ioe ) {
        throw new RuntimeException();
    }
}

/**
 * Writes the specified output message to the client.
 */
private void write( String message ) {

    out.print( message + "\r\n" );
    out.flush();
}
}

```

2.22 SMTPSender

```

/*****
 * $Workfile: SMTPSender.java $
 * $Revision: 1.12 $
 * $Author: edaugherty $
 * $Date: 2003/12/24 02:26:35 $
 *
 * $Edited by: Axel Anders Kvale
 *
 *****/

package com.ericdaugherty.mail.server.services.smtp;

//Java imports
import java.io.*;
import java.util.Date;
import java.util.Vector;
import java.util.List;

//Local imports
import com.ericdaugherty.mail.server.info.User;
import com.ericdaugherty.mail.server.info.EmailAddress;
import com.ericdaugherty.mail.server.errors.NotFoundException;
import com.ericdaugherty.mail.server.services.general.DeliveryService;
import com.ericdaugherty.mail.server.configuration.ConfigurationManager;

/**
 * This class (thread) is responsible for checking the disk for unsent message and
 * delivering them to the proper local address or remote smtp server.
 * <p>
 * There should be only one instance of this thread running in the system at
 * any one time.
 */
public class SMTPSender implements Runnable {

    /** The ConfigurationManager */
    private static ConfigurationManager configurationManager =
ConfigurationManager.getInstance();

    private boolean running = true;

    /**
     * The entypoint for this thread. This method handles the lifecycle
     * of this thread.
     */
    public void run() {

        while( running ) {

            try {

                File smtpDirectory = new File( configurationManager.getMailDirectory() +
File.separator + "smtp" );

                if( smtpDirectory.exists() && smtpDirectory.isDirectory() ) {

                    File[] files = smtpDirectory.listFiles();
                    int numFiles = files.length;

                    for( int index = 0; index < numFiles; index++ ) {
                        try {
                            deliver( SMTPMessage.load( files[index].getAbsolutePath() ) );
                        }
                        catch( Throwable throwable ) {}
                    }
                }

                //Rest the specified sleep time. If it is greater than 10 seconds
                //Wake up every 10 seconds to check to see if the thread is shutting
                //down.
                long sleepTime = configurationManager.getDeliveryIntervealMilliseconds();
                if( configurationManager.getDeliveryIntervealMilliseconds() < 10000 ) {
                    Thread.sleep( sleepTime );
                }
            }
            else {

```

```

        long totalSleepTime = sleepTime;
        while( totalSleepTime > 0 && running ) {
            if( totalSleepTime > 10000 ) {
                totalSleepTime -= 10000;
                Thread.sleep( 10000);
            }
            else {
                Thread.sleep( totalSleepTime );
                totalSleepTime = 0;
            }
        }
    }
}

catch( InterruptedException ie ) {
    //log.error( "Sleeping Thread was interrupted." );
}
catch( Throwable throwable )
{ }
}

}

/**
 * This method takes a SMTPMessage and attempts to deliver it. This
 * method assumes that all the addresses have been validated before,
 * and does not perform any delivery rules.
 */
private void deliver( SMTPMessage message ) {

    List toAddresses = message.getToAddresses();
    int numAddress = toAddresses.size();
    Vector failedAddress = new Vector();
    EmailAddress address = null;

    // If the next scheduled delivery attempt is still in the future, skip.
    if( message.getScheduledDelivery().getTime() > System.currentTimeMillis() )
    {
        return;
    }

    for( int index = 0; index < numAddress; index++ ) {
        try {
            address = (EmailAddress) toAddresses.get( index );

            DeliveryService deliveryService = DeliveryService.getDeliveryService();

            try {
                if( deliveryService.isLocalAddress( address ) ) {
                    deliverLocalMessage( address, message );
                }
                else {
                    deliverRemoteMessage( address, message );
                }
            }
            catch (NotFoundException e) {
                //The addressee does not exist. Notify the sender of the error.
                bounceMessage( address, message );
            }
        }
        catch( Throwable throwable ) {
            failedAddress.addElement( toAddresses.get( index ) );
        }
    }

    // If all addresses were successful, remove the message from the pool
    if( failedAddress.size() == 0 ) {
        // Log an error if the delete fails. This will cause the message to get
        // delivered again, but it is too late to roll back the delivery.
        boolean deleted = message.getMessageLocation().delete();
    }
    // Update the message with any changes.
    else {
        message.setToAddresses( failedAddress );
        int deliveryAttempts = message.getDeliveryAttempts();
    }
}

```

```

directory. // If the message is a bounced email, just give up and move it to the failed
// If the message is a bounced email, just give up and move it to the failed
if(message.getFromAddress().getUsername().equalsIgnoreCase("MAILER_DAEMON"))
{
    try {
        message.moveToFailedFolder();
    }
    catch (Exception e) {}
}
// If we have not passed the maximum delivery count, calculate the
// next delivery time and save the message.
else if( deliveryAttempts < configurationManager.getDeliveryAttemptThreshold() )
{
    message.setDeliveryAttempts( deliveryAttempts + 1 );

    // Reschedule later, 1 min, 2 min, 4 min, 8 min, ... 2^n
    // Cap delivery interval at 2^10 minutes. (about 17 hours)
    if( deliveryAttempts > 10 )
    {
        deliveryAttempts = 10;
    }
    long offset = (long)Math.pow( 2, deliveryAttempts);
    Date schedTime = new Date(System.currentTimeMillis() + offset*60*1000);
    message.setScheduledDelivery( schedTime );

    try {
        message.save();
    }
    catch( Exception exception ) {}
}
// All delivery attempts failed, bounce message.
else
{
    // Send a bounce message to all failed addresses.
    for( int index = 0; index < failedAddress.size(); index++ ) {
        try {
            EmailAddress bounce_address = (EmailAddress)(
failedAddress.elementAt(index) );
            bounceMessage(bounce_address, message);
        }
        catch(Exception e) {}
    }

    // Remove the original message.
    message.getMessageLocation().delete();
}
}
}

/**
 * This method takes a local SMTPMessage and attempts to deliver it.
 */
private void deliverLocalMessage( EmailAddress address, SMTPMessage message )
throws NotFoundException {

    User user = null;
    //Load the user. If the user doesn't exist, a not found exception will
    //be thrown and the deliver() message will deal with the notification.
    user = configurationManager.getUser( address );
    if( user == null )
    {
        //Check to see if a default delivery mailbox exists, and if so, deliver it.
        //Otherwise, just throw the NotFoundException to bounce the email.
        if( configurationManager.isDefaultUserEnabled() ) {
            EmailAddress defaultAddress = configurationManager.getDefaultUser();
            //If this throws a NotFoundException, go ahead and let it bounce.
            user = configurationManager.getUser( defaultAddress );
            if( user == null ) throw new NotFoundException();
        }
        else {
            throw new NotFoundException( "User does not exist and no default delivery
options found." );
        }
    }

    //The file to write to.
    File messageFile = null;

```

```

//The output stream to write the message to.
BufferedWriter out = null;

try {

    //Get the directory and create a new file.
    File userDirectory = user.getUserDirectory();
    messageFile = userDirectory.createTempFile("pop", ".jmsg", userDirectory );

    //Open the output stream.
    out = new BufferedWriter( new FileWriter( messageFile ) );

    //Get the data to write.
    List dataLines = message.getDataLines();
    int numDataLines = dataLines.size();

    //Write the X-DeliveredTo: header
    out.write( "X-DeliveredTo: " + address.getAddress() );
    out.write( "\r\n" );

    //Write the data.
    for( int index = 0; index < numDataLines; index++ ) {
        out.write( (String) dataLines.get( index ) );
        out.write( "\r\n" );
    }
}
catch( IOException ioe ) {
    if( messageFile != null ) {
        //The message was not fully written, so delete it.
        messageFile.delete();
    }
}
finally {
    if( out != null ) {
        try {
            //Make sure we close up the output stream.
            out.close();
        }
        catch( IOException ioe ) {}
    }
}
}

/**
 * Handles delivery of messages to addresses not handled by this server.
 */
private void deliverRemoteMessage( EmailAddress address, SMTPMessage message ) throws
NotFoundException {

    //Delegate this request to the SMTPRemoteSender class.
    new SMTPRemoteSender().sendMessage( address, message );
}

private void bounceMessage( EmailAddress address, SMTPMessage message ) {

    SMTPMessage bounceMessage = new SMTPMessage();

    //Set the from address as mailserver@ the first (default) local domain.
    EmailAddress fromAddress = new EmailAddress( "MAILER_DAEMON",
configurationManager.getLocalDomains()[0] );

    bounceMessage.setFromAddress( fromAddress );
    bounceMessage.addToAddress( message.getFromAddress() );
    bounceMessage.addDataLine( "From: Mail Delivery Subsystem <MAILER-DAEMON@" +
configurationManager.getLocalDomains()[0] + ">" );
    bounceMessage.addDataLine( "To: " + message.getFromAddress().getAddress() );
    bounceMessage.addDataLine( "Subject: Message Delivery Error." );
    bounceMessage.addDataLine( "Date: " + new Date().toString() ); //TODO: Improve date
handling.
    bounceMessage.addDataLine( "" );
    bounceMessage.addDataLine( "Error delivering message to: " + address.getAddress() );
    bounceMessage.addDataLine( "This message will not be delivered." );
    bounceMessage.addDataLine( "" );
    bounceMessage.addDataLine( "-----" );

    List dataLines = message.getDataLines();

```

```
int numLines = dataLines.size();

for( int index = 0; index < numLines; index++ ) {
    bounceMessage.addDataLine( (String) dataLines.get( index ) );
}
bounceMessage.addDataLine( " " );

//Save this message so it will be delivered.
try {
    bounceMessage.save();
}
catch (Exception e) {
    throw new RuntimeException();
}
}
```

3 Code from replacing the logging-component

3.1 AOP

```
package com.ericdaugherty.mail.server.aspects;

//Log imports
import java.util.logging.*;

//DNS imports
import org.xbill.DNS.Lookup;
import org.xbill.DNS.Record;

//Java imports
import java.io.*;
import java.net.*;
import java.lang.reflect.Method;
import java.lang.reflect.InvocationTargetException;
import java.util.*;

//JES imports
import com.ericdaugherty.mail.server.*;
import com.ericdaugherty.mail.server.errors.*;
import com.ericdaugherty.mail.server.configuration.*;
import com.ericdaugherty.mail.server.configuration.console.*;
import com.ericdaugherty.mail.server.services.general.*;
import com.ericdaugherty.mail.server.services.pop3.*;
import com.ericdaugherty.mail.server.services.smtp.*;
import com.ericdaugherty.mail.server.info.*;

public privileged aspect LogController{

    //local functions for the aspect
    public Logger LogInit(Class c){
        Logger log = Logger.getLogger( c.getPackage().getName() );
        //log.info("Logging started in " + c.getName());
        System.out.println("Logging started in " + c.getName());
        return log;
    }

    //The log interface for objects with non-static log
    public interface LogInterface {};
    public Logger LogInterface.logTest;

    /*****
    * This part places the logging objects in the appropriate classes
    *****/

    //the mail-class. This is treated separately for now...
    private static Logger com.ericdaugherty.mail.server.Mail.log
        = Logger.getLogger( Mail.class.getPackage().getName() );

    //static (a regexp way to intertype into classes would be nice
    private static Logger
com.ericdaugherty.mail.server.services.general.ServiceListener.logTest
    = Logger.getLogger( ServiceListener.class.getPackage().getName() );
    private static Logger com.ericdaugherty.mail.server.services.pop3.Pop3Processor.logTest
    = Logger.getLogger( Pop3Processor.class.getPackage().getName() );
    private static Logger
com.ericdaugherty.mail.server.configuration.console.ConfigurationProcessor.logTest
    = Logger.getLogger( ConfigurationProcessor.class.getPackage().getName() );
    private static Logger com.ericdaugherty.mail.server.info.Message.logTest
    = Logger.getLogger( Message.class.getPackage().getName() );
    private static Logger com.ericdaugherty.mail.server.services.smtp.SMTPMessage.logTest
    = Logger.getLogger( SMTPMessage.class.getPackage().getName() );
    private static Logger com.ericdaugherty.mail.server.services.smtp.SMTPProcessor.logTest
    = Logger.getLogger( SMTPProcessor.class.getPackage().getName() );
    private static Logger com.ericdaugherty.mail.server.services.smtp.SMTPRemoteSender.logTest
    = Logger.getLogger( SMTPRemoteSender.class.getPackage().getName() );
    private static Logger com.ericdaugherty.mail.server.services.smtp.SMTPSender.logTest
    = Logger.getLogger( SMTPSender.class.getPackage().getName() );
```

```

//non-static
declare parents : com.ericdaugherty.mail.server.ShutdownService implements LogInterface;
declare parents : com.ericdaugherty.mail.server.configuration.ConfigurationManager
implements LogInterface;
declare parents : com.ericdaugherty.mail.server.services.general.DeliveryService
implements LogInterface;
declare parents : com.ericdaugherty.mail.server.info.User implements LogInterface;

//The main(..) of Mail
private pointcut Mail_main(String[] args) :
    within(Mail) && execution(* main(..)) && args(args);

//non-static log constructors
private pointcut LogIFInit(LogInterface li) :
    this(li) && initialization(LogInterface.new(..));

//defining all the pointcuts where we want to log

before(String[] args): Mail_main(args){
    System.out.print("\r\n\r\nStarting LogInit in class: " +
thisJoinPointStaticPart.getSignature().getDeclaringType().getName() + "\r\n\r\n");
}

//non-static log init
before(LogInterface li) : LogIFInit(li){
    li.logTest = LogInit(li.getClass());
}

//Mail class logging
private pointcut Mail_NTshutdown() :
    within(Mail) && execution(* shutdown(String[]));

before() : Mail_NTshutdown() {
    Mail m = (Mail)(thisJoinPoint.getThis());
    m.log.log(Level.CONFIG, "NT Service requested application shutdown." );
}

private pointcut Mail_shutdown() :
    within(Mail) && execution(public static void shutdown());

before() : Mail_shutdown() {
    Mail m = (Mail)(thisJoinPoint.getThis());
    m.log.warn( "Shutting down Mail Server. Server will shut down in 60 seconds."
);
}

private pointcut Mail_sl(int port, Class c, int threads) :
    within(Mail) && call(ServiceListener.new( int, Class, int)) && args(port, c,
threads);

before(int port, Class c, int threads) : Mail_sl(port, c, threads){
    Mail m = (Mail)(thisJoinPoint.getThis());
    if(c == Pop3Processor.class)
        m.log.log(Level.CONFIG, "Starting POP3 Service on port: " +
port );
    else if(c == SMTPProcessor.class)
        m.log.log(Level.CONFIG, "Starting SMTP Service on port: " +
port );
}

//ConigurationManagng
private pointcut CFM_loaduser(EmailAddress ema, ConfigurationManager cf) :
    target(cf) && call(User ConfigurationManager.getUser(EmailAddress)) &&
args(ema);

after(EmailAddress ema, ConfigurationManager cf) returning (User u) :
CFM_loaduser(ema,cf){
    if( u == null ) cf.logTest.info( "Tried to load non-existent user: " +
ema.getAddress() );
}

private pointcut CF_loadgp (ConfigurationManager cf) :
    this(cf) && cflow( call(void loadGeneralProperties())) && set(String[]
ConfigurationManager.localDomains);

```

```

after(ConfigurationManager cf) : CF_loadgp(cf){
    cf.logTest.info( "Loaded " + cf.localDomains.length + " local domains." );
}

private pointcut CF_loadup(ConfigurationManager cf, String un) :
    this(cf) && cflow( call(void loadUserProperties()) ) && call(* Map.put(String,
User)) && args(un);

after (ConfigurationManager cf, String un) throwing: CF_loadup(cf,un) {
    cf.logTest.warning( "Skipping user: " + un + ". Address is invalid." );
}

private pointcut CF_loadusers(ConfigurationManager cf) :
    this(cf) && set(Map ConfigurationManager.users);

after (ConfigurationManager cf) : CF_loadusers(cf) {
    cf.logTest.info( "Loaded " + cf.users.size() + " users from user.conf" );
}

private pointcut CF_changeuc(ConfigurationManager cf) :
    this(cf) && cflow(call(void loadUserProperties()) ) && call(*
Properties.store(..));

after (ConfigurationManager cf) returning : CF_changeuc(cf) {
    cf.logTest.info( "Changes to user.conf persisted to disk." );
}

private pointcut CF_erroruc(ConfigurationManager cf) :
    this(cf) && cflow(call(void loadUserProperties()) ) && call(*
Properties.store(..));

after (ConfigurationManager cf) throwing : CF_erroruc(cf) {
    cf.logTest.severe( "Unable to store changes to user.conf! Plain text passwords
were not hashed!" );
}

private pointcut CF_errorpps(ConfigurationManager cf, String sValue, int iValue) :
    this(cf) && ( cflow(call(int parsePort(String, int)) ) && args(sValue, iValue) )
&& call(* Integer.parseInt(..));

after (ConfigurationManager cf, String sValue, int iValue) throwing : CF_errorpps(cf,
sValue, iValue) {
    cf.logTest.warning( "Error parsing port string: " + sValue + " using default
value: " + iValue );
}

private pointcut CF_errorecpp(ConfigurationManager cf, String address, String
password):
    this(cf) && (cflow( call(User loadUser(String, Properties)) ) && args(address,
..)) && (call(String PasswordManager.encryptPassword(String)) && args(password));

after (ConfigurationManager cf, String address, String password) returning :
CF_errorecpp(cf, address, password) {
    if(password == null)
        cf.logTest.severe( "Error encrypting plaintext password from user.conf
for user " + address );
}

private pointcut CF_errorfwai(ConfigurationManager cf, String fullAddress, String
fwAddress) :
    this(cf) &&
(cflow(call(User loadUser(..)) ) && args(fullAddress,..)) &&
(call(EmailAddress.new(String)) && args(fwAddress)));

after (ConfigurationManager cf, String fullAddress, String fwAddress) throwing :
CF_errorfwai( cf, fullAddress, fwAddress ){
    cf.logTest.warning("Forward address: " + fwAddress + " for " + fullAddress + "
is invalid and will be ignored." );
}

private pointcut CF_loadfwa(ConfigurationManager cf, EmailAddress[] emailAddresses,
String fullAddress) :
    this(cf) &&
(cflow(call(User ConfigurationManager.loadUser(..)) ) && args(fullAddress, ..))
&&
(call(* User.setForwardAddresses(EmailAddress[]) ) && args(emailAddresses));

```

```

        after(ConfigurationManager cf, EmailAddress[] emailAddresses, String fullAddress) :
CF_loadfwa(cf, emailAddresses, fullAddress){
    cf.logTest.log(Level.CONFIG, emailAddresses.length + " forward addresses load
for user: " + fullAddress );
}

    private pointcut CF_change cf() :
        this(ConfigurationManager.ConfigurationFileWatcher) &&
        call(* loadGeneralProperties());

    before() : CF_change cf() {
        ((LogInterface)thisJoinPoint.getThis()).logTest.info( "General Configuration
File Changed, reloading..." );
    }

    private pointcut CF_changeucf() :
        this(ConfigurationManager.ConfigurationFileWatcher) &&
        call(* loadUserProperties());

    before() : CF_changeucf() {
        ((LogInterface)thisJoinPoint.getThis()).logTest.info( "User Configuration File
Changed, reloading..." );
    }

    private pointcut CF_errorcfw(Throwable throwable) :
        this(ConfigurationManager.ConfigurationFileWatcher) && handler( Throwable ) &&
args(throwable);

    before(Throwable throwable) : CF_errorcfw(throwable) {
        ((LogInterface)thisJoinPoint.getThis()).logTest.log( Level.SEVERE, "Error in
ConfigurationWatcher thread. Thread will continue to execute. " + throwable, throwable );
    }

    //ConfigurationProcessor class logging
    private pointcut CP_errorsotu(ConfigurationProcessor cp) :
        this(cp) && call(* ServerSocket.setSoTimeout(..));

    after(ConfigurationProcessor cp) throwing : CP_errorsotu(cp) {
        cp.logTest.log(Level.SEVERE, "Error initializing Socket Timeout in
SMTPProcessor" );
    }

    private pointcut CP_errordc(ConfigurationProcessor cp, Throwable e) :
        this(cp) && handler(Throwable) && args(e);

    before(ConfigurationProcessor cp, Throwable e ) : CP_errordc(cp, e){
        cp.logTest.log(Level.CONFIG, "Disconnecting Exception:", e );
        cp.logTest.info( "Disconnecting" );
    }

    private pointcut CP_errordce(ConfigurationProcessor cp) :
        this(cp) && cflow(handler(Throwable)) && call(* Socket.close());

    after (ConfigurationProcessor cp) throwing( IOException ioe ) : CP_errordce(cp) {
        cp.logTest.log(Level.CONFIG, "Error disconnecting.", ioe );
    }

    //DeliveryService class logging
    private pointcut DS_addauthip(DeliveryService ds, String clientIp) :
        this(ds) && execution(* ipAuthenticated(String)) && args(clientIp);

    before (DeliveryService ds, String clientIp) : DS_addauthip(ds, clientIp) {
        ds.logTest.log(Level.CONFIG, "Adding authenticated IP address: " + clientIp );
    }

    private pointcut DS_lockmbx(DeliveryService ds, EmailAddress address):
        this(ds) && execution(* isMailboxLocked(EmailAddress)) && args(address);

    before (DeliveryService ds, EmailAddress address) : DS_lockmbx(ds, address){
        ds.logTest.log(Level.CONFIG, "Locking Mailbox: " + address.getAddress() );
    }

    private pointcut DS_unlockmbx(DeliveryService ds, EmailAddress address) :
        this(ds) && execution(* unlockMailbox(EmailAddress)) && args(address);

    before (DeliveryService ds, EmailAddress address) : DS_unlockmbx(ds, address){

```

```

        ds.logTest.log(Level.CONFIG, "Unlocking Mailbox: " + address.getAddress());
    }

    private pointcut DS_erroraad(DeliveryService ds) :
        this(ds) && cflow(execution(* isRelayApproved(..))) && call(*
StringTokenizer.nextToken());

    before (DeliveryService ds) throwing: DS_erroraad(ds) {
        ds.logTest.warning( "Invalid ApprovedAddress found. Skipping." );
    }

    //Message class logging
    private pointcut MSG_delset(Message m, boolean deleted) :
        this(m) && execution(* setDeleted(boolean)) && args(deleted);

    before (Message m, boolean deleted) : MSG_delset(m, deleted) {
        m.logTest.log(Level.CONFIG, "Setting is deleted to: " + deleted);
    }

    //Pop3Processor class logging
    private pointcut P3_errorinitsotu(Pop3Processor pc) :
        this(pc) && call(* ServerSocket.setSoTimeout(..));

    after (Pop3Processor pc) throwing : P3_errorinitsotu(pc) {
        pc.logTest.log(Level.SEVERE, "Error initializing Socket Timeout in
Pop3Processor" );
    }

    private pointcut P3_hostconnect(Pop3Processor pc, InetAddress remoteAddress) :
        this(pc) && target(remoteAddress) && call(String InetAddress.getHostAddress());

    after (Pop3Processor pc, InetAddress remoteAddress) returning (String clientIp) :
P3_hostconnect(pc, remoteAddress) {
        pc.logTest.info( remoteAddress.getHostAddress() + "(" + clientIp + ") socket
connected via POP3." );
    }

    private pointcut P3_errordc(Pop3Processor pc, Throwable e) :
        this(pc) && cflow(execution(* run())) && (handler(Throwable) && args(e));

    before (Pop3Processor pc, Throwable e) : P3_errordc(pc, e) {
        pc.logTest.log(Level.CONFIG, "Disconnecting Exception:", e);
        pc.logTest.info( "Disconnecting" );
    }

    private pointcut P3_errordcsm(Pop3Processor pc) :
        this(pc) && cflow(execution(* run())) && cflow(handler(Throwable)) && call(*
write(..));

    after (Pop3Processor pc) throwing(Exception e): P3_errordcsm(pc) {
        pc.logTest.log(Level.CONFIG, "Error sending disconnect message.", e);
    }

    private pointcut P3_errordce(Pop3Processor pc) :
        this(pc) && cflow(execution(* run())) && cflow(handler(Throwable)) && call(*
Socket.close());

    after (Pop3Processor pc) throwing(IOException ioe): P3_errordce(pc) {
        pc.logTest.log(Level.CONFIG, "Error disconnecting.", ioe);
    }

    private pointcut P3_shutdown(Pop3Processor pc) :
        this(pc) && execution(* run());

    after (Pop3Processor pc) : P3_shutdown(pc) {
        pc.logTest.warning( "Pop3Processor (1) shut down gracefully" );
    }

    private pointcut P3_shuttingdown(Pop3Processor pc) :
        this(pc) && execution(* shutdown());

    before (Pop3Processor pc) : P3_shuttingdown(pc) {
        pc.logTest.warning( "Shutting down Pop3Processor." );
    }

    private pointcut P3_userquit(Pop3Processor pc, String s) :
        this(pc) && execution(* checkQuit(String)) && args(s);

```

```

before (Pop3Processor pc, String s) : P3_userquit(pc, s) {
    if(s.equals(pc.COMMAND_QUIT)) pc.logTest.log(Level.CONFIG, "User has QUIT the
session." );
}

private pointcut P3_userlogin(Pop3Processor pc, User user):
    this(pc) && target(user) && cflow(execution(* authenticate())) && call(*
User.isPasswordValid( String) );

after (Pop3Processor pc, User user) returning (boolean bool): P3_userlogin(pc, user) {
    if(bool) { pc.logTest.info( "User: " + user.getFullUsername() + " logged in
successfully."); }
    else pc.logTest.info( "Login failed for user: " + user.getFullUsername());
}

private pointcut P3_loginfailed(Pop3Processor pc, EmailAddress address) :
    this(pc) && cflow(execution(* authenticate())) && (call(*
ConfigurationManager.getUser( EmailAddress )) && args(address));

after (Pop3Processor pc, EmailAddress address) returning(User user): P3_loginfailed(pc,
address){
    if(user == null) pc.logTest.info( "Login failed for user: " +
address.getAddress());
}

private pointcut P3_isdeletemsg(Pop3Processor pc, String argument, User user) :
    this(pc) && target(user) && cflow( (execution(* handleRetr(String)) ||
execution(* handleTop(String))) && args(argument)) && call(long User.getNumberOfMessage());

after (Pop3Processor pc, String argument, User user) returning(long numMessages) :
P3_isdeletemsg(pc, argument, user){
    int messageNumber = Integer.parseInt(argument);
    pc.logTest.log(Level.CONFIG, "Is Msg Deleted: " + user.getMessage(
messageNumber ).isDeleted() );
    pc.logTest.log(Level.CONFIG, "Message: " + messageNumber + " of " +
numMessages );
}

private pointcut P3_msgnf(Pop3Processor pc) :
    this(pc) && call(FileReader.new(...));

after (Pop3Processor pc) throwing (FileNotFoundException fnfe) : P3_msgnf(pc) {
    pc.logTest.log( Level.SEVERE, "Requested message could not be found on disk.",
fnfe );
}

private pointcut P3_errorretmsg(Pop3Processor pc) :
    this(pc) && call(* BufferedReader.readLine()) && (cflow(execution(*
handleRetr(...)) || cflow(execution(* handleTop(...))));

after (Pop3Processor pc) throwing (IOException ioe) : P3_errorretmsg(pc) {
    pc.logTest.log( Level.SEVERE, "Error retrieving message.", ioe );
}

private pointcut P3_handletop(Pop3Processor pc) :
    this(pc) && execution(* handleTop(...));

before (Pop3Processor pc) : P3_handletop(pc) {
    pc.logTest.log(Level.CONFIG, "In Top" );
}

private pointcut P3_notpass(Pop3Processor pc) :
    this(pc) && cflow(execution(String read())) && call(String String.trim());

after (Pop3Processor pc) returning (String inputLine) : P3_notpass(pc) {
    if( !inputLine.startsWith( "PASS" ) ) pc.logTest.log(Level.CONFIG, "Read
Input: " + inputLine );
}

private pointcut P3_errorrfs(Pop3Processor pc, IOException ioe) :
    this(pc) && cflow(execution(String read())) && ( handler(IOException) &&
args(ioe));

before (Pop3Processor pc, IOException ioe) : P3_errorrfs(pc, ioe) {
    pc.logTest.log( Level.SEVERE, "Error reading from socket.", ioe );
}

```

```

private pointcut P3_writeop(Pop3Processor pc, String message) :
    this(pc) && execution(* write(String) && args(message));

before (Pop3Processor pc, String message) : P3_writeop(pc, message){
    pc.logTest.log(Level.CONFIG, "Writing Output: " + message );
}

//ServiceListener class logging
private pointcut SL_startsl(ServiceListener sl) :
    this(sl) && execution(void run());

before (ServiceListener sl) : SL_startsl(sl) {
    sl.logTest.log(Level.CONFIG, "Starting ServiceListener on port: " + sl.port );
}

private pointcut SL_errorslcreate(ServiceListener sl, IOException e) :
    this(sl) && cflow(execution(void run())) && handler(IOException) && args(e);

before (ServiceListener sl, IOException e) : SL_errorslcreate(sl, e){
    InetAddress listenAddress =
ConfigurationManager.getInstance().getListenAddress();
    String address = "localhost";
    if( listenAddress != null ) {
        address = listenAddress.getHostAddress();
    }
    sl.logTest.log(Level.SEVERE, "Could not create ServiceListener on address: " +
address + " port: " + sl.port + ". No connections will be accepted on this port!" );
}

private pointcut SL_acceptcon(ServiceListener sl) :
    this(sl) && cflow(execution(void run())) && call(ServerSocket.new(..));

after (ServiceListener sl) returning (ServerSocket ss) : SL_acceptcon(sl) {
    sl.logTest.info( "Accepting Connections on port: " + ss.getLocalPort() );
}

private pointcut SL_errorslc(ServiceListener sl, Exception e) :
    this(sl) && cflow(execution(void run())) && handler(Exception) && args(e);

before (ServiceListener sl, Exception e) : SL_errorslc(sl, e) {
    sl.logTest.log(Level.SEVERE, "ServiceListener Connection failed on port: " +
sl.port + ". Error: " + e );
}

private pointcut SL_errorshutdown(ServiceListener sl) :
    this(sl) && cflow(execution(void shutdown())) && call(* Thread.join(..));

after (ServiceListener sl) throwing : SL_errorshutdown(sl) {
    sl.logTest.log(Level.SEVERE, "Was interrupted while waiting for thread to
die");
}

private pointcut SL_shutdown(ServiceListener sl) :
    this(sl) && cflow(execution(void shutdown())) && call(* Thread.join(..));

after (ServiceListener sl) returning : SL_shutdown(sl) {
    sl.logTest.info("Thread gracefully terminated");
}

private pointcut SL_errorcs(ServiceListener sl) :
    this(sl) && cflow(execution(void shutdown())) && call(* ServerSocket.close());

after (ServiceListener sl) throwing (Exception e): SL_errorcs(sl) {
    sl.logTest.log(Level.SEVERE, "Failed to close server socket", e );
}

private pointcut SL_closesocket(ServiceListener sl) :
    this(sl) && cflow(execution(void shutdown())) && call(* ServerSocket.close());

after (ServiceListener sl) returning : SL_closesocket(sl) {
    sl.logTest.log(Level.SEVERE, "Server socket successfully closed" );
}

//ShutdownService in Shutdown aspect
private pointcut SSR_shutdown() :
    this(Shutdown) && call(void shutdown());

```

```

before () : SSR_shutdown(){
    Mail.log.warning( "Server is shutting down." );
}

after () throwing (Exception e) : SSR_shutdown() {
    Mail.log.log(Level.SEVERE, "Failed to terminate properly", e);
}
after () returning : SSR_shutdown(){
    Mail.log.warning( "Server shutdown complete." );
}

//SMTPMessage class logger
private pointcut SM_errorgetdir(SMTPMessage sm, File failedDir) :
    this(sm) && target(failedDir) && cflow(execution(void moveToFailedFolder())) &&
call(boolean File.exists());

    after (SMTPMessage sm, File failedDir) returning(boolean bool): SM_errorgetdir(sm,
failedDir) {
        if( !bool ) sm.logTest.info( "failed directory does not exist. Creating: " +
failedDir.getAbsolutePath() );
    }

    private pointcut SM_errorcreatefaileddir(SMTPMessage sm, File failedDir) :
        this(sm) && target(failedDir) && cflow(execution(void moveToFailedFolder())) &&
call(boolean File.mkdirs());

    after (SMTPMessage sm, File failedDir) returning(boolean bool):
SM_errorcreatefaileddir(sm, failedDir){
        if( !bool ) sm.logTest.log(Level.SEVERE, "Error creating failed directory. No
incoming mail will be accepted!" );
    }

    private pointcut SM_errormovetodir(SMTPMessage sm, File messageLocation) :
        this(sm) && target(messageLocation) && cflow(execution(void
moveToFailedFolder())) && call(boolean File.renameTo(..));

    after (SMTPMessage sm, File messageLocation) returning(boolean bool):
SM_errormovetodir(sm, messageLocation){
        if( !bool ) sm.logTest.log(Level.SEVERE, "moveToFailedFolder failed.
Message was not renamed." );
    }

    private pointcut SM_errordirdne(SMTPMessage sm, File smtpDirectory) :
        this(sm) && target(smtpDirectory) && cflow(execution(void save())) &&
call(boolean File.exists());

    after (SMTPMessage sm, File smtpDirectory) returning(boolean bool): SM_errordirdne(sm,
smtpDirectory){
        if( !bool ) sm.logTest.info( "SMTP Mail directory does not exist.
Creating: " + smtpDirectory.getAbsolutePath() );
    }

    private pointcut SM_errorcreatedir(SMTPMessage sm, File smtpDirectory) :
        this(sm) && target(smtpDirectory) && cflow(execution(void save())) &&
call(boolean File.mkdirs());

    after (SMTPMessage sm, File smtpDirectory) returning(boolean bool):
SM_errorcreatedir(sm, smtpDirectory){
        if( !bool ) sm.logTest.log(Level.SEVERE, "Error creating SMTP Mail
directory. No incoming mail will be accepted!" );
    }

    private pointcut SM_errorclosespoolfile(SMTPMessage sm) :
        this(sm) && cflow(execution(void save())) && call(* FileWriter.close());

    after (SMTPMessage sm) throwing : SM_errorclosespoolfile(sm) {
        sm.logTest.warning( "Unable to close spool file for SMTPMessage " +
sm.messageLocation.getAbsolutePath() );
    }

    private pointcut SM_loadsmtpmsg(SMTPMessage sm, String filename) :
        this(sm) && cflow(execution(* load(String)) && args(filename)) && call(String
BufferedReader.readLine());

    after (SMTPMessage sm, String filename) returning (String version) : SM_loadsmtpmsg(sm,
filename){

```

```

        sm.logTest.log(Level.CONFIG, "Loading SMTP Message " + filename + " version "
+ version );
        if( !sm.FILE_VERSION.equals( version ) )
            sm.logTest.log(Level.SEVERE, "Error loading SMTP Message. Can not handle
file version: " + version );
    }

    private pointcut SM_errorparseaddress(SMTPMessage sm, String addresses) :
        this(sm) && cflow(execution(* inflateAddresses(String)) && args(addresses)) &&
call(EmailAddress.new(..));

    after (SMTPMessage sm, String addresses) throwing( InvalidAddressException
invalidAddressException) : SM_errorparseaddress(sm, addresses){
        sm.logTest.log(Level.SEVERE, "Unable to parse to address read from database.
Full String is: " + addresses, invalidAddressException );
    }

    //SMTPProcessor class logging
    private pointcut SP_errorsocketto(SMTPProcessor sp) :
        this(sp) && cflow(execution(void run())) && call(*
ServerSocket.setSoTimeout(..));

    after (SMTPProcessor sp) throwing(SocketException se) : SP_errorsocketto(sp){
        sp.logTest.log(Level.SEVERE, "Error initializing Socket Timeout in
SMTPProcessor" );
    }

    private pointcut SP_socketcon(SMTPProcessor sp, InetAddress remoteAddress) :
        this(sp) && target(remoteAddress) && cflow(execution(void run())) &&
call(String InetAddress.getHostAddress());

    after (SMTPProcessor sp, InetAddress remoteAddress) returning(String clientIp) :
SP_socketcon(sp, remoteAddress){
        sp.logTest.info( remoteAddress.getHostName() + "(" + clientIp + ") socket
connected via SMTP." );
    }

    private pointcut SP_errordce(SMTPProcessor sp, Throwable e) :
        this(sp) && cflow(execution(void run())) && handler(Throwable) && args(e);

    before (SMTPProcessor sp, Throwable e) : SP_errordce(sp, e){
        sp.logTest.log(Level.CONFIG, "Disconnecting Exception:", e );
        sp.logTest.info( "Disconnecting" );
    }

    private pointcut SP_errordc(SMTPProcessor sp) :
        this(sp) && cflow(execution(void run())) && call(* Socket.close());

    before (SMTPProcessor sp) throwing(IOException ioe) : SP_errordc(sp){
        sp.logTest.log(Level.CONFIG, "Error disconnecting.", ioe );
    }

    private pointcut SP_shutdown(SMTPProcessor sp) :
        this(sp) && execution(void run());

    after (SMTPProcessor sp) : SP_shutdown(sp){
        sp.logTest.warning( "SMTPProcessor shut down gracefully" );
    }

    before (SMTPProcessor sp) : SP_shutdown(sp){
        sp.logTest.warning( "Shutting down SMTPProcessor." );
    }

    private pointcut SP_erroruq(SMTPProcessor sp) :
        this(sp) && execution(void checkQuit(..));

    before (SMTPProcessor sp) throwing : SP_erroruq(sp) {
        sp.logTest.log(Level.CONFIG, "User has QUIT the session." );
    }

    private pointcut SP_errormfe(SMTPProcessor sp) :
        this(sp) && cflow(execution(* handleMailFrom(..)) && call(EmailAddress.new()));

    after (SMTPProcessor sp) returning: SP_errormfe(sp){
        sp.logTest.log(Level.CONFIG, "MAIL FROM is empty" );
    }

```

```

    private pointcut SP_mailfrom(SMTPProcessor sp, String fromAddress) :
        this(sp) && cflow(execution(* handleMailFrom(..))) &&
call(EmailAddress.new(String)) && args(fromAddress);

    after (SMTPProcessor sp, String fromAddress) returning: SP_mailfrom(sp, fromAddress){
        sp.logTest.log(Level.CONFIG, "MAIL FROM: " + fromAddress );
    }

    private pointcut SP_errorparsefa(SMTPProcessor sp, String fromAddress) :
        this(sp) && cflow(execution(* handleMailFrom(..))) &&
call(EmailAddress.new(String)) && args(fromAddress);

    after (SMTPProcessor sp, String fromAddress) throwing: SP_errorparsefa(sp,
fromAddress){
        sp.logTest.log(Level.CONFIG, "Unable to parse From Address: " +
fromAddress );
    }

    private pointcut SP_rcptto(SMTPProcessor sp, EmailAddress address, String clientIp) :
        this(sp) && cflow(execution(void handleRcptTo(..))) && call(boolean
DeliveryService.acceptAddress( EmailAddress, String )) && args(address,clientIp);

    after (SMTPProcessor sp, EmailAddress address, String clientIp) returning(boolean bool)
: SP_rcptto(sp, address, clientIP){
        if(bool) {
            sp.logTest.log(Level.CONFIG, "RCTP TO: " + address.getAddress() + "
accepted." );
        }
        else{
            sp.logTest.info( "Invalid delivery address for incoming mail: " +
address.getAddress() + " from client: " + clientIp );
        }
    }

    private pointcut SP_errorrctpto(SMTPProcessor sp, EmailAddress address) :
        this(sp) && cflow(execution(* handleRcptTo(..))) &&
call(EmailAddress.new(String)) && args(address);

    after (SMTPProcessor sp, EmailAddress address) throwing( InvalidAddressException iae )
: SP_errorrctpto(sp, address){
        sp.logTest.log(Level.CONFIG, "RCTP TO: " + address + " rejected." );
    }

    private pointcut SP_readdata(SMTPProcessor sp, String inputString) :
        this(sp) && target(inputString) && cflow(execution(* handleData())) &&
call(boolean String.equals(..));

    after (SMTPProcessor sp, String inputString) returning (boolean bool) : SP_readdata(sp,
inputString){
        if(bool) { sp.logTest.log(Level.CONFIG, "Read Data: " + inputString ); }
        else sp.logTest.log(Level.CONFIG, "Data Input Complete." );
    }

    private pointcut SP_errorretdata(SMTPProcessor sp) :
        this(sp) && cflow(execution(* handleData())) && call(*
BufferedReader.readLine());

    after (SMTPProcessor sp) throwing(IOException ioe) : SP_errorretdata(sp){
        sp.logTest.log(Level.SEVERE, "An error occured while retrieving the message
data.", ioe );
    }

    private pointcut SP_handledata(SMTPProcessor sp) :
        this(sp) && execution(* handleData());

    after (SMTPProcessor sp) : SP_handledata(sp){
        sp.logTest.info( "Message " + sp.message.getMessageLocation().getName() + "
accepted for delivery." );
    }

    private pointcut SP_readdata(SMTPProcessor sp) :
        this(sp) && cflow(execution(* read())) && call(String String.trim());

    after (SMTPProcessor sp) returning(String inputLine) : SP_readdata(sp) {
        sp.logTest.log(Level.CONFIG, "Read Input: " + inputLine );
    }

```

```

private pointcut SP_errorrfs(SMTPProcessor sp) :
    this(sp) && cflow(execution(* read())) && call(* BufferedReader.readLine());

after (SMTPProcessor sp) throwing(IOException ioe) : SP_errorrfs(sp) {
    sp.logTest.log(Level.SEVERE, "Error reading from socket.", ioe );
}

private pointcut SP_writemsg(SMTPProcessor sp, String message) :
    this(sp) && execution(void write(String)) && args(message);

before(SMTPProcessor sp, String message) : SP_writemsg(sp, message){
    sp.logTest.log(Level.CONFIG, "Writing: " + message );
}

//SMTPRemoteSender class logging
private pointcut SRS_errorclosesocket(SMTPRemoteSender srs) :
    this(srs) && call(* Socket.close());

after (SMTPRemoteSender srs) throwing(IOException ioe) : SRS_errorclosesocket(srs){
    srs.logTest.log(Level.SEVERE, "Error closing socket: " + ioe );
}

private pointcut SRS_dnslookup(SMTPRemoteSender srs, EmailAddress address) :
    this(srs) && cflow(execution(Socket connect(EmailAddress)) && args(address)) &&
call(* Lookup.run());

    after (SMTPRemoteSender srs, EmailAddress address) returning(Record[] records):
SRS_dnslookup(srs,address){
    if(records == null) srs.logTest.warning( "DNS Lookup for domain: " +
address.getDomain() + " failed." );
}

private pointcut SRS_errorconsmtp(SMTPRemoteSender srs, String host) :
    this(srs) && call(Socket.new(String, int)) && args(host,..);

after (SMTPRemoteSender srs, String host) throwing ( Exception e ) :
SRS_errorconsmtp(srs,host){
    srs.logTest.log(Level.CONFIG, "Connection to SMTP Server: " + host + " failed
with exception: " + e ) ;
}

private pointcut SRS_errorrfs(SMTPRemoteSender srs) :
    this(srs) && call(* BufferedReader.readLine());

after (SMTPRemoteSender srs) throwing(IOException ioe) : SRS_errorrfs(srs){
    srs.logTest.log(Level.SEVERE, "Error reading from socket.", ioe );
}

private pointcut SRS_readline(SMTPRemoteSender srs) :
    this(srs) && call(* BufferedReader.readLine());

after (SMTPRemoteSender srs) returning (String s) : SRS_readline(srs){
    srs.logTest.log(Level.CONFIG, "Read Input: " + s );
}

//SMTPSender class logging
private pointcut SS_findmsgtodeliver(SMTPSender ss) :
    this(ss) && get(boolean SMTPSender.running);

after (SMTPSender ss) returning (boolean running) : SS_findmsgtodeliver(ss){
deliver" );
}

private pointcut SS_errordelivermsg(SMTPSender ss) :
    this(ss) && call(* SMTPMessage.load(..));

after (SMTPSender ss) throwing( Throwable throwable ) : SS_errordelivermsg(ss){
Message: " + throwable, throwable );
}

private pointcut SS_errorsleep(SMTPSender ss) :
    this(ss) && call(* Thread.sleep(..));

after (SMTPSender ss) throwing (InterruptedException ie) : SS_errorsleep(ss){

```

```

        ss.logTest.log(Level.SEVERE, "Sleeping Thread was interrupted." );
    }

    private pointcut SS_run(SMTPSender ss) :
        this(ss) && execution(void run());

    after (SMTPSender ss) : SS_run(ss){
        ss.logTest.warning( "SMTPSender shut down gracefully.");
    }

    private pointcut SS_shutdown(SMTPSender ss) :
        this(ss) && execution(* shutdown());

    before (SMTPSender ss) : SS_shutdown(ss) {
        ss.logTest.warning( "Attempting to shut down SMTPSender." );
    }

    private pointcut SS_skipdeliversmtppmsg(SMTPSender ss, SMTPMessage message) :
        this(ss) && execution(* deliver(SMTPMessage)) && args(message);

    before (SMTPSender ss, SMTPMessage message) : SS_skipdeliversmtppmsg(ss, message){
        if( message.getScheduledDelivery().getTime() > System.currentTimeMillis() ){
            ss.logTest.log(Level.CONFIG, "Skipping delivery of message " +
message.getMessageLocation().getName() + " because the scheduled delivery time is still in the
future: " + message.getScheduledDelivery() );
        }
    }

    private pointcut SS_deliversmtppmsg(SMTPSender ss, EmailAddress address, SMTPMessage
message) :
        this(ss) && (call(* deliverLocalMessage(..)) || call(*
deliverRemoteMessage(..))) && args(address, message);

    before(SMTPSender ss, EmailAddress address, SMTPMessage message) :
SS_deliversmtppmsg(ss, address, message){
        ss.logTest.log(Level.CONFIG, "Attempting to deliver message from: " +
message.getFromAddress().getAddress() + " to: " + address );
    }

    after (SMTPSender ss, EmailAddress address, SMTPMessage message) throwing
(NotFoundException e): SS_deliversmtppmsg(ss, address, message){
        ss.logTest.info( "Delivery attempted to unknown user: " +
address.getAddress() );
    }

    after (SMTPSender ss, EmailAddress address, SMTPMessage message) returning:
SS_deliversmtppmsg(ss, address, message){
        ss.logTest.info( "Delivery complete for message " +
message.getMessageLocation().getName() + " to: " + address );
    }

    private pointcut SS_remoovesmtppmsg(SMTPSender ss, File f):
        this(ss) && target(f) && call(boolean File.delete());

    after(SMTPSender ss, File f) returning(boolean bool) : SS_remoovesmtppmsg(ss, f){
        if(!bool)
            ss.logTest.log(Level.SEVERE, "Error removed SMTP message after
delivery! This message may be redelivered. " + f.getName() );
    }

    private pointcut SS_movetofailedfolder(SMTPSender ss) :
        this(ss) && call(* Message.moveToFailedFolder());

    before (SMTPSender ss) : SS_movetofailedfolder(ss){
        ss.logTest.info( "Delivery of message from MAILER_DAEMON failed, moving to
failed folder." );
    }

    after (SMTPSender ss) throwing : SS_movetofailedfolder(ss){
        ss.logTest.log(Level.SEVERE, "Unable to move failed message to 'failed'
folder." );
    }

    private pointcut SS_updatespooledmsg(SMTPSender ss):
        this(ss) && cflow(execution(* deliver(..)) && call(* SMTPMessage.save());

    after (SMTPSender ss) throwing(Exception exception): SS_updatespooledmsg(ss) {

```

```

        ss.logTest.log(Level.SEVERE, "Error updating spooled message for next
delivery. Message may be re-delivered.", exception );
    }

    private pointcut SS_deliverlocal(SMTPSender ss, EmailAddress address) :
        this(ss) && execution(* deliverLocalMessage(EmailAddress, ..)) &&
args(address,..);

    before (SMTPSender ss, EmailAddress address) : SS_deliverlocal(ss, address){
        ss.logTest.log(Level.CONFIG, "Delivering Message to local user: " +
address.getAddress() );
    }

    private pointcut SS_getuser(SMTPSender ss) :
        this(ss) && call(User ConfigurationManager.getUser(EmailAddress));

    after (SMTPSender ss) returning(User user) : SS_getuser(ss){
        if(user == null)
            ss.logTest.log(Level.CONFIG, "User not found" );
    }

    private pointcut SS_delivertodefault(SMTPSender ss, EmailAddress address):
        this(ss) && cflow(call(* ConfigurationManager.isDefaultUserEnabled())) &&
call(User ConfigurationManager.getUser(EmailAddress)) && args(address);

    after (SMTPSender ss, EmailAddress address) returning(User user) :
SS_delivertodefault(ss, address){
        if(user == null) {ss.logTest.log(Level.CONFIG, "User not found in
default delivery options" );}
        ss.logTest.info( "Delivering message to default user: " + address );
    }

    private pointcut SS_deliveringto(SMTPSender ss):
        this(ss) && call(File File.createTempFile(..));

    after (SMTPSender ss) returning (File f) : SS_deliveringto(ss){
        ss.logTest.log(Level.CONFIG, "Delivering to: " + f.getAbsolutePath() );
    }

    private pointcut SS_errorlocaldelivery(SMTPSender ss, IOException ioe) :

        this(ss) && cflow(execution(* deliverLocalMessage(..)) && handler(IOException)
&& args(ioe);

    before (SMTPSender ss, IOException ioe) : SS_errorlocaldelivery(ss, ioe){
        ss.logTest.log(Level.SEVERE, "Error performing local delivery.", ioe );
    }

    private pointcut SS_errorclosestream(SMTPSender ss) :
        this(ss) && call(* BufferedWriter.close());

    after (SMTPSender ss) throwing(IOException ioe) : SS_errorclosestream(ss){
        ss.logTest.log(Level.SEVERE, "Error closing output Stream.", ioe );
    }

    private pointcut SS_remotedelivery(SMTPSender ss, EmailAddress address) :
        this(ss) && execution(* deliverRemoteMessage( EmailAddress , SMTPMessage)) &&
args(address, ..);

    before (SMTPSender ss, EmailAddress address) : SS_remotedelivery(ss, address){
        ss.logTest.log(Level.CONFIG, "Delivering Message to remote user: " + address
);
    }

    private pointcut SS_bouncemsg(SMTPSender ss, EmailAddress address, SMTPMessage message
) :
        this(ss) && execution(* bounceMessage( EmailAddress, SMTPMessage ) ) &&
args(address, message);

    before (SMTPSender ss, EmailAddress address, SMTPMessage message ) : SS_bouncemsg(ss,
address, message){
        ss.logTest.info( "Bouncing Messsage from " +
message.getFromAddress().getAddress() + " to " + address.getAddress() );
    }

    private pointcut SS_errorstorebounce(SMTPSender ss) :

```

```

        this(ss) && cflow(call(* bounceMessage(..)) && call(* SMTPMessage.save()));
    after (SMTPSender ss) throwing : SS_errorstorebounce(ss){
        ss.logTest.log(Level.SEVERE, "Error storing outgoing 'bounce' email message");
    }

    //User class logging

    private pointcut U_authenticate(User u) :
        this(u) && execution(* isPasswordValid(..));

    before (User u) : U_authenticate(u) {
        u.logTest.log(Level.CONFIG, "Authenticating User: " + u.getFullUsername() );
    }

    after (User u) returning(boolean result) : U_authenticate(u){
        if( !result ) u.logTest.log(Level.CONFIG, "Authentication Failed for User: " +
u.getFullUsername() );
    }

    private pointcut U_getuserdir(User u) :
        this(u) && call (boolean File.exists()) && cflow(execution(*
getUserDirectory()));

    after (User u) returning(boolean bool) : U_getuserdir(u){
        if(!bool) { u.logTest.info( "Directory for user: " + u.getFullUsername() +
"does not exist, creating..." ); }
    }

    private pointcut U_errorgetuserdir(User u, File directory) :
        this(u) && target(directory) && call (boolean File.isDirectory()) &&
cflow(execution(* getUserDirectory()));

    after (User u, File directory) returning(boolean bool) : U_errorgetuserdir(u,
directory){
        if(!bool) { u.logTest.log(Level.SEVERE, "User Directory: " +
directory.getAbsolutePath() + " for user: " + u.getFullUsername() + " does not exist." ); }
    }
}

```

3.2 OOP

Only a few classes are shown. The changes to the rest of the classes are identical.

3.2.1 Mail

```
/* *****  
 * $Workfile: Mail.java $  
 * $Revision: 1.4 $  
 * $Author: edaugherty $  
 * $Date: 2003/10/15 19:06:23 $  
 *  
 * $Modified by: Axel Anders Kvale$  
 *  
 ***** */  
package com.ericdaugherty.mail.server;  
  
//Java imports  
import java.io.*;  
import java.lang.reflect.Method;  
import java.lang.reflect.InvocationTargetException;  
import java.util.Properties;  
  
import java.util.logging.*;  
  
//Local imports  
import com.ericdaugherty.mail.server.configuration.ConfigurationManager;  
import com.ericdaugherty.mail.server.configuration.ConfigurationParameterContants;  
import com.ericdaugherty.mail.server.services.general.ServiceListener;  
import com.ericdaugherty.mail.server.services.smtp.SMTPSender;  
import com.ericdaugherty.mail.server.services.smtp.SMTPProcessor;  
import com.ericdaugherty.mail.server.services.pop3.Pop3Processor;  
  
/**  
 * This class is the entrypoint for the Mail Server application. It creates  
 * threads to listen for SMTP and POP3 connections. It also handles the  
 * configuration information and initialization of the User subsystem.  
 *  
 * @author Eric Daugherty  
 */  
public class Mail {  
  
    /* *****  
    // Variables  
    /* *****  
  
    //Threads  
  
    private static ServiceListener popListener;  
    private static ServiceListener smtpListener;  
    private static SMTPSender smtpSender;  
    private static ShutdownService shutdownService;  
  
    /** The SMTP sender thread */  
    private static Thread smtpSenderThread;  
  
    /** The ShutdownService Thread. Started when the JVM is shutdown. */  
    private static Thread shutdownServiceThread;  
  
    /** Logger Category for this class. This variable is initialized once  
     * the main logging system has been initialized */  
    private static Logger log = null;  
  
    /* *****  
    // Public Interface  
    /* *****  
  
    /**  
     * Provides a 'safe' way for the application to shut down. This  
     * method is provided to enable compatability with the JavaService  
     * NT Service wrapper class. It defers the call to the shutdown method.  
     *  
     * @param args  
     */  
    public static void shutdown( String[] args ) {
```

```

        log.log(Level.CONFIG, "NT Service requested application shutdown." );
        shutdown();
    }

    /**
     * Provides a 'safe' way for the application to shut down. It will attempt
     * to stop the running threads. If the threads do not stop within 60 seconds,
     * the application will force the threads to stop by calling System.exit();
     */
    public static void shutdown() {

        log.warning( "Shutting down Mail Server. Server will shut down in 60 seconds." );

        popListener.shutdown();
        smtpListener.shutdown();
        smtpSender.shutdown();

        try{
            smtpSenderThread.join(10000);
        }
        catch (InterruptedException ie)
        {
            log.log(Level.SEVERE,"Was interrupted while waiting for thread to die");
        }

        log.info("Thread gracefully terminated");
        smtpSenderThread = null;
    }

    //*****
    // Main Method

    /**
     * This method is the entrypoint to the system and is responsible
     * for the initial configuration of the application and the creation
     * of all 'service' threads.
     */
    public static void main( String[] args ) {

        // Perform the basic application startup. If anything goes wrong here,
        // we need to abort the application.
        try {

            // Get the 'root' directory for the mail server.
            String directory = getConfigurationDirectory( args );

            // Initialize the Configuration Manager.
            ConfigurationManager configurationManager = ConfigurationManager.initialize(
directory );

            //Start the threads.
            int port;
            int executeThreads = configurationManager.getExecuteThreadCount();

            //Start the Pop3 Thread.
            port = configurationManager.getPop3Port();
            log.log(Level.CONFIG, "Starting POP3 Service on port: " + port );
            popListener = new ServiceListener( port, Pop3Processor.class, executeThreads );
            new Thread( popListener, "POP3" ).start();

            //Start SMTP Threads.
            port = configurationManager.getSmtpPort();
            log.log(Level.CONFIG, "Starting SMTP Service on port: " + port );
            smtpListener = new ServiceListener( port, SMTPProcessor.class, executeThreads );
            new Thread( smtpListener, "SMTP" ).start();

            //Start the SMTPSender thread (This thread actually delivers the mail recieved
            //by the SMTP threads.
            smtpSender = new SMTPSender();
            smtpSenderThread = new Thread( smtpSender, "SMTPSender" );
            smtpSenderThread.start();

            //Initialize ShutdownService
            shutdownService = new ShutdownService();
            shutdownServiceThread = new Thread(shutdownService);
            Runtime.getRuntime().addShutdownHook( shutdownServiceThread );

```

```

    }
    catch( RuntimeException runtimeException ) {
        System.err.println( "The application failed to initialize." );
        System.err.println( runtimeException.getMessage() );
        runtimeException.printStackTrace();
        System.exit( 0 );
    }
}

/*****
// Private Interface
*****/

/**
 * Parses the input parameter for the configuration directory, or defaults
 * to the local directory.
 *
 * @param args the commandline arguments.
 * @return the directory to use as the 'root'.
 */
private static String getConfigurationDirectory( String[] args ) {

    String directory = ".";
    File directoryFile;

    // First, check to see if the location was passed as a paramter.
    if (args.length == 1) {
        directory = args[0];
    }
    // Otherwise, use the default, which is 'mail.conf' in the current directory.
    else if( (directoryFile = new File( directory )).exists() ) {
        System.out.println( "Configuration Directory not specified. Using \" +
directoryFile.getAbsolutePath() + "\" );
    }
    // If no file was specified and the default does not exist, printing out a usage line.
    else {
        System.out.println("Usage: java com.ericdaugherty.mail.server.Mail <configuration
directory>");
        throw new RuntimeException( "Unable to load the configuration file." );
    }

    return directory;
}
}

```

3.2.2 Pop3Processor

```
/* *****
 * $Workfile: Pop3Processor.java $
 * $Revision: 1.4 $
 * $Author: edaugherty $
 * $Date: 2003/11/15 21:40:30 $
 *
 * $Modified by: Axel Anders Kvale$
 *
 * *****
 */

package com.ericdaugherty.mail.server.services.pop3;

//Java imports
import java.net.*;
import java.io.*;

//Log imports
import java.util.logging.*;

//Local imports
import com.ericdaugherty.mail.server.info.*;
import com.ericdaugherty.mail.server.services.general.DeliveryService;
import com.ericdaugherty.mail.server.services.general.ConnectionProcessor;
import com.ericdaugherty.mail.server.configuration.ConfigurationManager;

/**
 * Handles an incoming Pop3 connection. See rfc 1939 for details.
 *
 * @author Eric Daugherty
 */
public class Pop3Processor extends Thread implements ConnectionProcessor {

    /* *****
    // Variables
    /* *****

    /** Logger Category for this class. */
    private static Logger log = Logger.getLogger( Pop3Processor.class.getPackage().getName()
);

    /** The ConfigurationManager */
    private static ConfigurationManager configurationManager =
ConfigurationManager.getInstance();

    /** Indicates if this thread should continue to run or shut down */
    private boolean running = true;

    /** The server socket used to listen for incoming connections */
    private ServerSocket serverSocket;

    /** Socket connection to the client */
    private Socket socket;

    /** The IP address of the client */
    private String clientIp;

    /** The user currently logged in */
    private User user = null;

    /** Writer to sent data to the client */
    private PrintWriter out;
    /** Reader to read data from the client */
    private BufferedReader in;

    /* *****
    // Public Interface
    /* *****

    /**
     * Sets the socket used to communicate with the client.
     */
    public void setSocket( ServerSocket serverSocket ) {

        this.serverSocket = serverSocket;
    }
}
```

```

}

/**
 * Entrypoint for the Thread, this method handles the interaction with
 * the client socket.
 */
public void run() {

    try {
        //Set the socket to timeout every 10 seconds so it does not
        //just block forever.
        serverSocket.setSoTimeout( 1000 );
    }
    catch( SocketException se ) {
        log.log(Level.SEVERE, "Error initializing Socket Timeout in Pop3Processor" );
    }

    while( running ) {
        try {
            socket = serverSocket.accept();

            //Prepare the input and output streams.
            out = new PrintWriter(socket.getOutputStream(), true);
            in = new BufferedReader(new InputStreamReader( socket.getInputStream() ));

            InetAddress remoteAddress = socket.getInetAddress();
            clientIp = remoteAddress.getHostAddress();

            log.info( remoteAddress.getHostName() + "(" + clientIp + ") socket connected
via POP3." );

            //Output the welcome message.
            write( WELCOME_MESSAGE );

            //Forces the client to properly authenticate.
            user = authenticate();
            user.reset();

            //Parses the input for commands and delegates to the appropriate methods.
            handleCommands();
        }
        catch( InterruptedException iioe ) {
            //This is fine, it should time out every 10 seconds if
            //a connection is not made.
        }
        //If any exception gets to here uncaught, it means we should just disconnect.
        catch( Throwable e ) {
            log.log(Level.CONFIG, "Disconnecting Exception:", e );
            log.info( "Disconnecting" );

            //Unlock the user's mailbox
            if( user != null ) {
                EmailAddress userAddress = new EmailAddress( user.getUsername(),
user.getDomain() );
                DeliveryService.getDeliveryService().unlockMailbox( userAddress );
            }

            try {
                write( MESSAGE_DISCONNECT );
            }
            catch( Exception e1 ) {
                log.log(Level.CONFIG, "Error sending disconnect message.", e1 );
                //Nothing to do.
            }
            try {
                if( socket != null ) {
                    socket.close();
                }
            }
            catch( IOException ioe ) {
                log.log(Level.CONFIG, "Error disconnecting.", ioe );
                //Nothing to do.
            }
        }
    }
    log.warning( "Pop3Processor shut down gracefully" );
}

```

```

/**
 * Notifies this thread to stop processing and exit.
 */
public void shutdown() {
    log.warning( "Shutting down Pop3Processor." );
    running = false;
}

/*****
// Private Interface
/*****

/**
 * Checks to verify that the command is not a quit command. If it is,
 * the current state is finalized (all messages marked as deleted are
 * actually deleted) and closes the connection.
 */
private void checkQuit( String command ) {

    if( command.equals( COMMAND_QUIT ) ) {
        log.log(Level.CONFIG, "User has QUIT the session." );

        //Delete the messages marked as deleted from disk
        if( user != null ) {
            Message[] messages = user.getMessages();
            int numMessage = messages.length;
            Message currentMessage = null;

            for( int index = 0; index < numMessage; index++ ) {
                currentMessage = messages[index];
                if( currentMessage.isDeleted() ) {
                    messages[index].getMessageLocation().delete();
                }
            }
        }

        // TODO Find a better way to handle user logoffs.
        throw new RuntimeException();
    }
}

/**
 * The user must authenticate before moving on to enter
 * more commands. This method will listen to incoming
 * commands until the user either successfully authenticates
 * or quits.
 */
private User authenticate() {

    //Reusable Variables.
    String inputString;
    String command;
    String argument;

    String username = "";
    String domain = "";
    String password = "";
    EmailAddress address = null;
    DeliveryService deliveryService = DeliveryService.getDeliveryService();

    //Get the username from the client.
    boolean userAccepted = false;

    while( !userAccepted ) {
        inputString = read();

        command = parseCommand( inputString );
        argument = parseArgument( inputString );

        //Check to see if they sent the user command.
        if( command.equals( COMMAND_USER ) ) {

            //Make sure they sent a username
            if( argument.equals( "" ) ) {
                write( MESSAGE_TOO_FEW_ARGUMENTS );
            }
        }
    }
}

```

```

    }
    else {
        int atIndex = argument.indexOf( "@" );

        //Verify that the username contains the domain.
        if( atIndex == -1 ) {
            write( MESSAGE_NEED_USER_DOMAIN );
        }
        else {
            //Accept the user, and proceed to get the password.
            username = argument.substring( 0, atIndex );
            domain = argument.substring( atIndex + 1 );

            address = new EmailAddress( username, domain );

            //Check to see if the user's mailbox is locked
            if( deliveryService.isMailboxLocked( address ) ) {
                write( MESSAGE_USER_MAILBOX_LOCKED );
            }
            else {
                write( MESSAGE_USER_ACCEPTED + argument );
                userAccepted = true;
            }
        }
    }
}
else {
    write( MESSAGE_INVALID_COMMAND + command );
}
}

//The user has been accepted, now get the password.
boolean passwordAccepted = false;

while( !passwordAccepted ) {
    inputString = read();

    command = parseCommand( inputString );
    argument = parseArgument( inputString );

    //Check to see if they sent the user command.
    if( command.equals( COMMAND_PASS ) ) {

        //Make sure they sent a password
        if( argument.equals( "" ) ) {
            write( MESSAGE_TOO_FEW_ARGUMENTS );
        }
        else {
            password = argument;
            passwordAccepted = true;
        }
    }
    else {
        write( MESSAGE_INVALID_COMMAND + command );
    }
}

User user = configurationManager.getUser( address );
if( user != null && user.isPasswordValid( password ) )
{
    deliveryService.ipAuthenticated( clientIp );
    deliveryService.lockMailbox( address );
    write( MESSAGE_LOGIN_SUCCESSFUL );
    log.info( "User: " + address.getAddress() + " logged in successfully." );
    return user;
}
else
{
    //The login failed, display a message to the user and disconnect.
    write( MESSAGE_INVALID_LOGIN + username );
    log.info( "Login failed for user: " + username + "@" + domain );
    throw new RuntimeException();
}
}

/**
 * Handles all the commands related the the retrieval of mail.

```

```

*/
private void handleCommands() {

    //Reusable Variables.
    String inputString;
    String command;
    String argument;

    //This just runs until a SystemException is thrown, which
    //signals us to disconnect.
    while( true ) {

        inputString = read();

        command = parseCommand( inputString );
        argument = parseArgument( inputString );

        //Identify the command and call the appropriate helper method.
        if( command.equals( COMMAND_STAT ) ) {
            handleStat();
        }
        else if( command.equals( COMMAND_LIST ) ) {
            handleList( argument );
        }
        else if( command.equals( COMMAND_RETR ) ) {
            handleRetr( argument );
        }
        else if( command.equals( COMMAND_DELE ) ) {
            handleDele( argument );
        }
        else if( command.equals( COMMAND_NOOP ) ) {
            write( "+OK" );
        }
        else if( command.equals( COMMAND_RSET ) ) {
            handleRset();
        }
        else if( command.equals( COMMAND_TOP ) ) {
            handleTop( argument );
        }
        else if( command.equals( COMMAND_UIDL ) ) {
            handleUidl( argument );
        }
        else {
            write( MESSAGE_INVALID_COMMAND + command );
        }
    }
}

/**
 * Handles the 'stat' command, which returns the total number of message
 * and the total size of those message.
 */
private void handleStat() {

    write( "+OK " + user.getNumberOfMessage() + " " + user.getSizeOfAllMessage() );
}

/**
 * Handles the 'list' command, which returns the total number of messages and
 * size along with a list of the individual message sizes.
 */
private void handleList( String argument ) {

    if( argument.equals( "" ) ) {
        long numMessages = user.getNumberOfMessage();
        long sizeMessage = user.getSizeOfAllMessage();

        write( "+OK " + numMessages + " messages (" + sizeMessage + " octets)" );

        for( int index = 0; index < numMessages; index++ ) {
            write( (index + 1) + " " + user.getMessage( index + 1
).getMessageLocation().length() );
        }
        write( "." );
    }
    else {
        int messageNumber = 0;
    }
}

```

```

        try {
            messageNumber = Integer.parseInt( argument );
        }
        catch( NumberFormatException nfe ) {
            write( MESSAGE_NOT_A_NUMBER );
            return;
        }

        long numMessages = user.getNumberOfMessage();

        if( messageNumber > numMessages || user.getMessage( messageNumber ).isDeleted() )
        {
            write( MESSAGE_NO_SUCH_MESSAGE );
            return;
        }

        write( "+OK " + messageNumber + " " + user.getMessage( messageNumber
).getLocation().length() );
    }
}

/**
 * Sends the specified email message to the client.
 */
private void handleRetr( String argument ) {

    int messageNumber = 0;

    try {
        messageNumber = Integer.parseInt( argument );
    }
    catch( NumberFormatException nfe ) {
        write( MESSAGE_NOT_A_NUMBER );
        return;
    }

    long numMessages = user.getNumberOfMessage();

    log.log(Level.CONFIG, "Is Msg Deleted: " + user.getMessage( messageNumber
).isDeleted() );
    log.log(Level.CONFIG, "Message: " + messageNumber + " of " + numMessages );

    if( messageNumber > numMessages || user.getMessage( messageNumber ).isDeleted() ) {
        write( MESSAGE_NO_SUCH_MESSAGE );
        return;
    }

    write( MESSAGE_OK );

    BufferedReader fileIn = null;
    try {
        //Open an reader to read the file.
        fileIn = new BufferedReader( new FileReader( user.getMessage( messageNumber
).getLocation() ) );

        //Write the file to the client.
        String currentLine = fileIn.readLine();
        while (currentLine != null) {
            write( currentLine );
            currentLine = fileIn.readLine();
        }
        write( "." );
    }
    catch( FileNotFoundException fnfe ) {
        log.log(Level.SEVERE, "Requested message for user " + user.getFullUsername() + "
could not be found on disk.", fnfe );
    }
    catch( IOException ioe ) {
        log.log(Level.SEVERE, "Error retrieving message.", ioe );
        write( "-ERR Error retrieving message" );
    }
    finally {
        //Make sure the input stream gets closed.
        try {
            if( fileIn != null ) {
                fileIn.close();
            }
        }
    }
}

```

```

        }
    }
    catch( IOException ioe ) {
        //Nothing to do...
    }
}

/**
 * Marks the specified message for deletion. The message will only
 * be deleted if the user later enters the QUIT command, as per the
 * spec.
 */
private void handleDele( String argument ) {

    int messageNumber = 0;

    try {
        messageNumber = Integer.parseInt( argument );
    }
    catch( NumberFormatException nfe ) {
        write( MESSAGE_NOT_A_NUMBER );
        return;
    }

    long numMessages = user.getNumberOfMessage();

    if( messageNumber > numMessages ) {
        write( MESSAGE_NO_SUCH_MESSAGE );
    }
    else if( user.getMessage( messageNumber ).isDeleted() ) {
        write( MESSAGE_ALREADY_DELETED );
    }
    else {
        user.getMessage( messageNumber ).setDeleted( true );
        write( MESSAGE_OK );
    }
}

/**
 * Unmarks all deleted messages.
 */
private void handleRset() {

    Message[] messages = user.getMessages();
    int numMessage = messages.length;

    for( int index = 0; index < numMessage; index++ ) {
        messages[index].setDeleted( false );
    }

    write( MESSAGE_OK );
}

/**
 * Returns the header and first x lines for the
 * specified message.
 */
private void handleTop( String argument ) {

    log.log(Level.CONFIG, "In Top" );

    int messageNumber = 0;
    int numLines = 0;

    int spaceIndex = argument.indexOf( " " );
    if( spaceIndex == -1 ) {
        write( MESSAGE_TOO_FEW_ARGUMENTS );
        return;
    }
    String arg1 = argument.substring( 0, spaceIndex ).trim();
    String arg2 = argument.substring( spaceIndex + 1 ).trim();

    try {
        messageNumber = Integer.parseInt( arg1 );
        numLines = Integer.parseInt( arg2 );
    }
    catch( NumberFormatException nfe ) {

```

```

        write( MESSAGE_NOT_A_NUMBER );
        return;
    }

    long numMessages = user.getNumberOfMessage();

    log.log(Level.CONFIG, "Is Msg Deleted: " + user.getMessage( messageNumber
).isDeleted() );
    log.log(Level.CONFIG, "Message: " + messageNumber + " of " + numMessages );

    if( messageNumber > numMessages || user.getMessage( messageNumber ).isDeleted() ) {
        write( MESSAGE_NO_SUCH_MESSAGE );
        return;
    }

    write( MESSAGE_OK );

    BufferedReader fileIn = null;
    try {
        //Open an reader to read the file.
        fileIn = new BufferedReader( new FileReader( user.getMessage( messageNumber
).getMessageLocation() ) );

        //Write the Message Header.
        String currentLine = fileIn.readLine();
        while (currentLine != null && !currentLine.equals( "" ) ) {
            write( currentLine );
            currentLine = fileIn.readLine();
        }

        //Write an empty line to separate header from body.
        write( currentLine );
        currentLine = fileIn.readLine();

        //Write the requested number of lines from the body of the
        //message, or until the entire message has been written.
        int index = 0;
        while( index < numLines && currentLine != null ) {
            write( currentLine );
            currentLine = fileIn.readLine();
            index++;
        }

        write( "." );
    }
    catch( FileNotFoundException fnfe ) {
        log.log(Level.SEVERE, "Requested message for user " + user.getFullUsername() + "
could not be found on disk.", fnfe );
    }
    catch( IOException ioe ) {
        log.log(Level.SEVERE, "Error retrieving message.", ioe );
        write( "-ERR Error retrieving message" );
    }
    finally {
        //Make sure the input stream gets closed.
        try {
            if( fileIn != null ) {
                fileIn.close();
            }
        }
        catch( IOException ioe ) {
            //Nothing to do...
        }
    }
}

/**
 * Returns the unique id of the specified message, or all the unique
 * ids of the non-deleted messages.
 */
private void handleUidl( String argument ) {

    //Return all messages unique ids
    if( argument == null || argument.length() == 0 ) {

        long numMessages = user.getNumberOfMessage();

```

```

        Message message;

        write( MESSAGE_OK );

        //Write out each non-deleted message id.
        for( int index = 0; index < numMessages; index++ ) {
            message = user.getMessage( index + 1 );
            if( !message.isDeleted() ) {
                write( (index + 1) + " " + message.getUniqueId() );
            }
        }
        write( "." );
    }
    //Output a single messages unique id.
    else {

        int messageNumber = 0;

        try {
            messageNumber = Integer.parseInt( argument );
        }
        catch( NumberFormatException nfe ) {
            write( MESSAGE_NOT_A_NUMBER );
            return;
        }

        long numMessages = user.getNumberOfMessage();

        if( messageNumber > numMessages || user.getMessage( messageNumber
.isDeleted() ) {
            write( MESSAGE_NO_SUCH_MESSAGE );
            return;
        }

        write( MESSAGE_OK + " " + messageNumber + " " + user.getMessage(
messageNumber ).getUniqueId() );
    }
}
/**
 * Reads a line from the input stream and returns it.
 */
private String read() {
    try {
        String inputLine = in.readLine().trim();
        //Log the input, unless it is a password.
        if( !inputLine.startsWith( "PASS" ) ) {
            log.log(Level.CONFIG, "Read Input: " + inputLine );
        }
        return inputLine;
    }
    catch( IOException ioe ) {
        log.log(Level.SEVERE, "Error reading from socket.", ioe );
        throw new RuntimeException();
    }
}

/**
 * Writes the specified output message to the client.
 */
private void write( String message ) {
    log.log(Level.CONFIG, "Writing Output: " + message );
    out.print( message + "\r\n" );
    out.flush();
}

/**
 * Parses the input stream for the command. The command is the
 * begining of the input stream to the first space. If there is
 * space found, the entire input string is returned.
 * <p>
 * This method converts the returned command to uppercase to allow
 * for easier comparison.
 * <p>
 * Additinally, this method checks to verify that the quit command
 * was not issued. If it was, a SystemException is thrown to terminate
 * the connection.

```

```

*/
private String parseCommand( String inputString ) {

    int index = inputString.indexOf( " " );

    if( index == -1 ) {
        String command = inputString.toUpperCase();
        checkQuit( command );
        return command;
    }
    else {
        String command = inputString.substring( 0, index ).toUpperCase();
        checkQuit( command );
        return command;
    }
}

/**
 * Parses the input stream for the argument.  The argument is the
 * text starting afer the first space until the end of the inputstring.
 * If there is no space found, an empty string is returned.
 * <p>
 * This method does not convert the case of the argument.
 */
private String parseArgument( String inputString ) {

    int index = inputString.indexOf( " " );

    if( index == -1 ) {
        return "";
    }
    else {
        return inputString.substring( index + 1 ).trim();
    }
}

//*****
// Constants
//*****
//Message Constants
//General Message
private static final String WELCOME_MESSAGE = "+OK EricDaugherty's Java Pop Server Ready";
private static final String MESSAGE_DISCONNECT = "+OK Pop server signing off.";
private static final String MESSAGE_OK = "+OK";
private static final String MESSAGE_INVALID_COMMAND = "-ERR Unknown command: ";
private static final String MESSAGE_TOO_FEW_ARGUMENTS = "-ERR Too few arguments for this
command.";

//Authentication Messages
private static final String MESSAGE_NEED_USER_DOMAIN = "-ERR User names must contain the
username and domain.  ex: \"root@mydomain.com\"";
private static final String MESSAGE_USER_ACCEPTED = "+OK Password required for ";
private static final String MESSAGE_LOGIN_SUCCESSFUL = "+OK Login successful";
private static final String MESSAGE_USER_MAILBOX_LOCKED = "-ERR User's Mailbox is locked";
private static final String MESSAGE_INVALID_LOGIN = "-ERR Password supplied is incorrect
for user: ";

//Other Messages
private static final String MESSAGE_NOT_A_NUMBER = "-ERR Command requires a valid number
as an argument.";
private static final String MESSAGE_NO_SUCH_MESSAGE = "-ERR No such message.";
private static final String MESSAGE_ALREADY_DELETED = "-ERR Message already deleted.";

//Command Constants
private static final String COMMAND_QUIT = "QUIT";
private static final String COMMAND_USER = "USER";
private static final String COMMAND_PASS = "PASS";
private static final String COMMAND_STAT = "STAT";
private static final String COMMAND_LIST = "LIST";
private static final String COMMAND_RETR = "RETR";
private static final String COMMAND_DELE = "DELE";
private static final String COMMAND_NOOP = "NOOP";
private static final String COMMAND_RSET = "RSET";
    private static final String COMMAND_TOP = "TOP";
    private static final String COMMAND_UIDL = "UIDL";
}

```

4 Code from implementation of SpamAssassin

4.1 AOP

```
import com.ericdaugherty.mail.server.services.smtp.*;
import java.util.ArrayList;
import java.util.List;
import java.util.Collection;
import java.io.*;

public privileged aspect SpamCheck{
    private static String SMTPMessage.mapFlattenAddresses( Collection addresses ){
        return flattenAddresses(addresses);
    }
    private static final String DELIMITER = "\r\n";
    private pointcut SaveMessage(SMTPMessage sm):
        target(sm) && call(* SMTPMessage.save(..));
    private pointcut SMTPProcessor() :
        this(SMTPProcessor);

    before (SMTPMessage sm) : SaveMessage(sm) && SMTPProcessor(){
        try{
            List dataLines = new ArrayList();
            File messageFile = File.createTempFile( "spam", ".txt");
            FileWriter writer = new FileWriter( messageFile );
            try
            {
                writer.write( "From: " + sm.getFromAddress().toString() );
                writer.write( DELIMITER );
                writer.write( "To: " + sm.mapFlattenAddresses(
sm.getToAddresses() ) );
                writer.write( DELIMITER );
                for( int index = 0; index < sm.dataLines.size(); index++ )
                {
                    writer.write( (String) sm.dataLines.get( index ) );
                    writer.write( DELIMITER );
                }
                writer.close();
                String path = messageFile.getPath();
                String path_new = path.substring(0,path.length()-3)+"new";
                String command = new String("cmd /C
C:\\Perl\\bin\\spamassassin.bat < ");
                command += path;
                command += " > ";
                command += path_new;
                Process p;
                p = Runtime.getRuntime ().exec (command);
                p.waitFor();
                File new_message = new File(path_new);
                FileReader fileReader = new FileReader( new_message );
                BufferedReader reader = new BufferedReader(fileReader);
                String inputLine = reader.readLine();
                boolean writeTo = false, writeFrom = false;
                while( inputLine != null )
                {
                    if(inputLine.startsWith("To:")){
                        if(writeTo)
                            dataLines.add(inputLine);
                        else
                            writeTo=true;
                    }
                    else if(inputLine.startsWith("From:")){
                        if(writeFrom)
                            dataLines.add( inputLine );
                        else
                            writeFrom = true;
                    }
                    else
                        dataLines.add( inputLine );
                    inputLine = reader.readLine();
                }
                sm.dataLines = dataLines;
                reader.close();
            }
        }
    }
}
```

```

        catch(Exception e){};
    }
    catch(Exception e){};
}
}

```

4.2 OOP

```

/*****
 * $Workfile: SMTPMessage.java $
 * $Revision: 1.4 $
 * $Author: edaugherty $
 * $Date: 2003/12/24 02:26:35 $
 *
 * $Modified by: Axel Anders Kvale$
 *
 *****/

package com.ericdaugherty.mail.server.services.smtp;

//Java imports
import java.io.*;
import java.util.Date;
import java.util.List;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
import java.util.StringTokenizer;

//Log imports
import java.util.logging.*;

//Local imports
import com.ericdaugherty.mail.server.configuration.ConfigurationManager;
import com.ericdaugherty.mail.server.info.EmailAddress;
import com.ericdaugherty.mail.server.errors.InvalidAddressException;

/**
 * Bean class used to store incoming SMTP message on disk (via Java Serialization)
 * for delivery by the SMTPSender thread.
 *
 * Note: All methods are removed except the SpamCheck() method. This method is called from
 * SMTPSender
 *
 * @author Eric Daugherty
 */
public class SMTPMessage implements Serializable {

    public void spamCheck(){
        try{
            List dataLinesTmp = new ArrayList();
            File messageFile = File.createTempFile( "spam", ".txt");
            FileWriter writer = new FileWriter( messageFile );
            try
            {
                writer.write( "From: " + getFromAddress().toString() );
                writer.write( DELIMITER );
                writer.write( "To: " + flattenAddresses( getToAddresses() ) );
                writer.write( DELIMITER );
                for( int index = 0; index < dataLines.size(); index++ )
                {
                    writer.write( (String) dataLines.get( index ) );
                    writer.write( DELIMITER );
                }
                writer.close();
                String path = messageFile.getPath();
                String path_new = path.substring(0,path.length()-3)+"new";
                String command = new String("cmd /C
C:\\Perl\\bin\\spamassassin.bat < ");
                command += path;
                command += " > ";
                command += path_new;
                Process p;
                p = Runtime.getRuntime ().exec (command);
                //This should be done in a separate thread, or an error will
            }
            catch (Exception e) {}
        }
        catch (Exception e) {}
    }
}
halt the program forever.

```

```

//Since this is not for production, it will be left like this
for now...

p.waitFor();
File new_message = new File(path_new);
FileReader fileReader = new FileReader( new_message );
BufferedReader reader = new BufferedReader(fileReader);
String inputLine = reader.readLine();
boolean writeTo = false, writeFrom = false;
while( inputLine != null )
{
    if(inputLine.startsWith("To:")){
        if(writeTo)
            dataLinesTmp.add(inputLine);
        else
            writeTo=true;
    }
    else if(inputLine.startsWith("From:")){
        if(writeFrom)
            dataLinesTmp.add( inputLine );
        else
            writeFrom = true;
    }
    else
        dataLinesTmp.add( inputLine );
    inputLine = reader.readLine();
}
dataLines = dataLinesTmp;
reader.close();
}
catch(Exception e){};
}
catch(Exception e){};
}
}
//EOF

```

5 Code from implementation of SpamProbe

5.1 AOP

```
import com.ericdaugherty.mail.server.services.smtp.*;
import java.util.ArrayList;
import java.util.List;
import java.util.Collection;
import java.io.*;

public privileged aspect SpamCheck{
    private static String SMTPMessage.mapFlattenAddresses( Collection addresses ){
        return flattenAddresses(addresses);
    }
    private static final String DELIMITER = "\r\n";
    private pointcut SaveMessage(SMTPMessage sm):
        target(sm) && call(* SMTPMessage.save(..));
    private pointcut SMTPProcessor() :
        this(SMTPProcessor);

    before (SMTPMessage sm) : SaveMessage(sm) && SMTPProcessor(){
        try{
            File messageFile = File.createTempFile( "spam", ".txt");
            FileWriter writer = new FileWriter( messageFile );
            try
            {
                writer.write( "From: " + sm.getFromAddress().toString() );
                writer.write( DELIMITER );
                writer.write( "To: " + sm.mapFlattenAddresses(
sm.getToAddresses() ) );
                writer.write( DELIMITER );
                for( int index = 0; index < sm.dataLines.size(); index++ )
                {
                    writer.write( (String) sm.dataLines.get( index ) );
                    writer.write( DELIMITER );
                }
                writer.close();
                String path = messageFile.getPath();
                String path_new = path.substring(0,path.length()-3)+"new";
                String command = new String("cmd /C C:\\\\Perl\\bin\\spamprobe.bat
< ");
                command += path;
                command += " > ";
                command += path_new;
                Process p;
                p = Runtime.getRuntime ().exec (command);
                p.waitFor();
                File new_message = new File(path_new);
                FileReader fileReader = new FileReader( new_message );
                BufferedReader reader = new BufferedReader(fileReader);
                String inputLine = reader.readLine();
                boolean writeTo = false, writeFrom = false;
                int i = 0;
                while( inputLine != null )
                {
                    sm.dataLines.add( i, inputLine );
                    inputLine = reader.readLine();
                    i++;
                }
                reader.close();
            }
            catch(Exception e){};
        }
        catch(Exception e){};
    }
}
```

5.2 OOP

```
/*
 * $Workfile: SMTPMessage.java $
 * $Revision: 1.4 $
 * $Author: edaugherty $
 * $Date: 2003/12/24 02:26:35 $
 */
```

```

*
* $Modified by: Axel Anders Kvale$
*
*****/

package com.ericdaugherty.mail.server.services.smtp;

//Java imports
import java.io.*;
import java.util.Date;
import java.util.List;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
import java.util.StringTokenizer;

//Log imports
import java.util.logging.*;

//Local imports
import com.ericdaugherty.mail.server.configuration.ConfigurationManager;
import com.ericdaugherty.mail.server.info.EmailAddress;
import com.ericdaugherty.mail.server.errors.InvalidAddressException;

/**
 * Bean class used to store incoming SMTP message on disk (via Java Serialization)
 * for delivery by the SMTPSender thread.
 *
 * Note: All methods have been removed except the SpamCheck method. This method is called from
 * the SMTPSender class
 *
 * @author Eric Daugherty
 */
public class SMTPMessage implements Serializable {

    public void spamCheck(){
        try{
            File messageFile = File.createTempFile( "spam", ".txt");
            FileWriter writer = new FileWriter( messageFile );
            try
            {
                writer.write( "From: " + getFromAddress().toString() );
                writer.write( DELIMITER );
                writer.write( "To: " + flattenAddresses( getToAddresses() ) );
                writer.write( DELIMITER );
                for( int index = 0; index < dataLines.size(); index++ )
                {
                    writer.write( (String) dataLines.get( index ) );
                    writer.write( DELIMITER );
                }
                writer.close();
                String path = messageFile.getPath();
                String path_new = path.substring(0,path.length()-3)+"new";
                String command = new String("cmd /C C:\\\\Perl\\bin\\spamprobe.bat

< ");

                command += path;
                command += " > ";
                command += path_new;
                Process p;
                p = Runtime.getRuntime ().exec (command);
                //This should be done in a separate thread, or an error will
                halt the program forever.
                //Since this is not for production, it will be left like this
                for now...

                p.waitFor();
                File new_message = new File(path_new);
                FileReader fileReader = new FileReader( new_message );
                BufferedReader reader = new BufferedReader(fileReader);
                String inputLine = reader.readLine();
                boolean writeTo = false, writeFrom = false;
                int i = 0;
                while( inputLine != null )
                {
                    dataLines.add( i, inputLine );
                    inputLine = reader.readLine();
                    i++;
                }
            }
        }
    }
}

```

```
        reader.close();
    }
    catch(Exception e){};
}
catch(Exception e){};
}
}
//EOF
```