```java
package server;

import java.io.*;
import java.util.*;
import ocsf.server.*;
import common.*;

public class EchoServer implements Observer {

  final public static int DEFAULT_PORT = 5555;

  static final String PASSWORDFILE = "passwords.txt";

  static final int LINEBREAK = 10; // Added in Phase 3
  static final int RETURN = 13; // Added in Phase 3
  static final int SPACE = 32; // Added in Phase 3

  ObservableOriginatorServer server;

  String serverChannel = null;

  Vector blockedUsers = new Vector();

  private ChatIF serverUI;

  private boolean closing = false;

  public EchoServer(ObservableOriginatorServer server, ChatIF serverUI) throws
IOException {
    this.server= server;
    this.serverUI = serverUI;
    server.addObserver(this);
    server.listen();
  }

  public void sendToAllClients(Object msg) {
    Thread[] clients = server.getClientConnections();

    for (int i = 0; i < clients.length; i++) {
      ConnectionToClient c = (ConnectionToClient)(clients[i]);

      try {
        //If the client is logged on, send the message
        if (((Boolean)(c.getInfo("passwordVerified"))).booleanValue())
          c.sendToClient(msg);
      } catch (IOException e) {
        serverUI.display("WARNING - Cannot send message to a client.");
      }
    }
  }

  public void update(Observable obs, Object msg) {
    // block added in phase 4 to handle Originator Messages.
    if (! (msg instanceof OriginatorMessage))
      return;

    OriginatorMessage message= (OriginatorMessage)msg;

    if (! (message.getMessage() instanceof String))
```

```java
      return;

    String command = (String)message.getMessage();
    ConnectionToClient client= (ConnectionToClient)message.getOriginator();

    if (command.startsWith(ObservableServer.CLIENT_CONNECTED)) {
      clientConnected(client);
      return;
    } else if (command.startsWith(ObservableServer.CLIENT_DISCONNECTED)) {
      clientDisconnected(client);
      return;
    } else if (command.startsWith(ObservableServer.CLIENT_EXCEPTION)) {
      int ie= command.indexOf('.');
      clientException(client, new Exception(command.substring(ie)));
      return;
    } else if (command.startsWith(ObservableServer.LISTENING_EXCEPTION)) {
      int ie= command.indexOf('.');
      listeningException(new Exception(command.substring(ie)));
      return;
    } else if (command.startsWith(ObservableServer.SERVER_STARTED)) {
      serverStarted();
      return;
    } else if (command.startsWith(ObservableServer.SERVER_STOPPED)) {
      serverStopped();
      return;
    } else if (command.startsWith(ObservableServer.SERVER_CLOSED)) {
      serverClosed();
      return;
    }

    // In general, display the command on the server's UI
    // Don't display it if the user is blocked
    if (!blockedUsers.contains(((String)(client.getInfo("loginID"))))) {
      // Only display it if the server is on the same channel as
      // the client or is in the 'superchannel'.
      // The server is in the superchannel by default, and this is
      // indicated by serverChannel being null.
      if (serverChannel == null || serverChannel.equals(client.getInfo("channel"))) {
        serverUI.display("Message: \"" + command + "\" from " + client.getInfo("loginID"));
      }
    }

    // If the user has logged in, process the command or send the message
    if (((Boolean)(client.getInfo("passwordVerified"))).booleanValue()) {
      // If the command was to list the users.  Added in phase 3.
      if (command.startsWith("#whoison"))
        sendListOfClients(client);

      // If the command was to retrieve the channel. Added phase 3
      if (command.startsWith("#getchannel")) {
        try {
          client.sendToClient("Currently on channel: " + client.getInfo("channel"));
        } catch(IOException e) {
          serverUI.display("Warning: Error sending message.");
        }
      }

      // If the command was to send a private message. Added phase 3.
      if (command.startsWith("#private"))
        handleCmdPrivate(command, client);

      // If the command was to change channels.  Added phase 3.
      if (command.startsWith("#channel"))
        handleCmdChannel(command, client);

      // If the command was to return to the main channel. Added phase 3.
      if (command.startsWith("#nochannel"))
        handleCmdChannel("#channel main", client);

      // If the command was to broadcast a public message. Added phase 3.
      if (command.startsWith("#pub"))
        handleCmdPub(command, client);

      // If the command was to forward messages. Added phase 3.
      if (command.startsWith("#fwd"))
        handleCmdFwd(command, client);

      // If the command was to stop forwarding messages. Added phase 3.
      if (command.startsWith("#unfwd")) {
        client.setInfo("fwdClient", "");

        try {
          client.sendToClient("Messages will no longer be forwarded");
        } catch(IOException e) {
          serverUI.display("Warning: Error sending message.");
        }
      }

      // If the command was to block a user.  Added phase 3.
      if (command.startsWith("#block"))
        handleCmdBlock(command, client);

      // If the command was to unblock users.  Added phase 3.
      if (command.startsWith("#unblock"))
        handleCmdUnblock(command, client);

      // If the command was to verify the users a client blocks. Added phase 3.
      if (command.startsWith("#whoiblock"))
        handleCmdWhoiblock(client);

      // If the command was to verify the users who are blocking
      // the client requesting the check. Added phase 3.
      if (command.startsWith("#whoblocksme"))
        checkForBlocks((String)(client.getInfo("loginID")), client);

      // If no command is recognized, send a message to the client's current
channel.
      if (!command.startsWith("#")) {
        sendChannelMessage(client.getInfo("loginID") + "> " + command,
           (String)client.getInfo("channel"),
           (String)(client.getInfo("loginID")));
      }
    }
    //If the user is not logged in, log him in.
    else {
      clientLoggingIn(command, client);
    }
  }

  /**
   * This method is called to handle data entered from the Server's console.
   * @param message The message typed by the user.
   */
  public synchronized void handleMessageFromServerUI(String message) {
    //If the command is #quit.  Added in phase 2
    if (message.startsWith("#quit"))
      quit();

    //If the command is #stop. Added in phase 2
    if (message.startsWith("#stop")) {
      if(server.isListening()) {
        server.stopListening();
      } else {
        serverUI.display("Cannot stop the server before it is restarted.");
      }

      return;
    }

    //If the command is #start.  Added in phase 2
    if (message.startsWith("#start")) {
      closing = false;
      if (!server.isListening()) {
        try {
          server.listen();
          serverChannel = null;
        } catch(IOException e) {
          serverUI.display("Cannot listen.  Terminating server.");
          quit();
        }
      } else {
        serverUI.display("Server is already running.");
      }
      return;
    }

    //If the command is #close. Added in phase 2
    if (message.startsWith("#close")) {
      closing = true;  // Indicates server is closing down
      sendToAllClients("Server is shutting down.");
      sendToAllClients("You will be disconnected.");

      try {
        server.close();
      } catch(IOException e) {
        serverUI.display("Cannot close normally. Terminating server.");
        quit();
      }
      return;
    }

    //If the command is #getport.  Added in phase 2
    if (message.startsWith("#getport")) {
      serverUI.display("Current port: " + server.getPort());
      return;
    }

    //If the command is #setport.  Added in phase 2
    if (message.startsWith("#setport")) {
      if ((server.getNumberOfClients() != 0) || (server.isListening())) {
        serverUI.display("Cannot change port while clients are "
                    + "connected or while server is listening.");
      } else {
        try {
          int port = 0;
          port = Integer.parseInt(message.substring(9));

          //If the port number is invalid
          if ((port < 1024) || (port > 65535)) {
            server.setPort(5555);
            serverUI.display("Invalid port number.  Port unchanged.");
          } else {
            server.setPort(port);
            serverUI.display("Port set to " + port);
          }
        } catch(Exception e) {
          serverUI.display("Invalid use of the #setport command.");
          serverUI.display("Port unchanged.");
        }
      }
      return;
    }

    //If command is #whoison (List users)  Added in phase 3.
    if (message.startsWith("#whoison")) {
      sendListOfClients(null);
      return;
    }

    //If the command was a punt command (boot user) Added in phase 3.
    if (message.startsWith("#punt")) {
      handleServerCmdPunt(message);
      return;
    }

    //If message is a #warn command. Added in phase 3.
    if (message.startsWith("#warn")) {
      handleServerCmdWarn(message);
      return;
    }

    //If command is #channel.  Added in phase 3
    if (message.startsWith("#channel")) {
      String oldChannel = serverChannel;
      if (!(oldChannel == null)) {
        sendChannelMessage("The server has left this channel.", serverChannel, "");
      }

      try {
        serverChannel = message.substring(9);
      } catch (StringIndexOutOfBoundsException e) {
        serverChannel = null;
        serverUI.display("Server will now receive all messages.");
      }

      if (serverChannel != null) {
        sendChannelMessage("The server has joined this channel.", serverChannel,
"");
```

```java
      }

      serverUI.display("Now on channel: " + serverChannel);
      return;
    }

    //If command is #nochannel.  Added in phase 3.
    if (message.startsWith("#nochannel")) {
      if (serverChannel != null) {
        sendChannelMessage("The server has left this channel.", serverChannel, "");
      }

      serverChannel = null;
      serverUI.display("Server will now receive all messages.");
      return;
    }

    //If command is #pub.  Added in phase 3.
    if (message.startsWith("#pub")) {
      handleCmdPub(message, null);
      return;
    }

    //If command is #getchannel
    if (message.startsWith("#getchannel")) {
      if (server.isListening() || server.getNumberOfClients() > 0) {
        serverUI.display("Currently on channel: " + serverChannel);
      } else {
        serverUI.display("Server has no active channels.");
      }
      return;
    }

    //If the command is to block a user.  Added in phase 3.
    if (message.startsWith("#block")) {
      handleServerCmdBlock(message);
      return;
    }

    //If the command was to unblock.  Added in phase 3.
    if (message.startsWith("#unblock")) {
      handleCmdUnblock(message, null);
      return;
    }

    //If the command is to check which users are blocked.  Added in phase 3.
    if (message.startsWith("#whoiblock")) {
      handleCmdWhoiblock(null);
      return;
    }

    //If command to send a private message.  Added in phase 3.
    if (message.startsWith("#private")) {
      handleCmdPrivate(message, null);
      return;
    }

    //If command is to check users who are blocking the server.  Added phase 3.
    if (message.startsWith("#whoblocksme")) {
      checkForBlocks("server", null);
      return;
    }

    //If command is a help command.
    if (message.startsWith("#?") || message.startsWith("#help")) {
      serverUI.display("\nServer-side command list:"
      + "\n#block <loginID> -- Blocks all messages from the specified client."
      + "\n#channel <channel> -- Connects to the specified channel."
      + "\n#close -- Stops the server and disconnects all users."
      + "\n#getchannel -- Gets the channel the server is currently connected to."
      + "\n#getport -- Gets the port the server is listening on."
      + "\n#help OR #? -- Lists all commands and their use."
      + "\n#nochannel -- Returns the server to the super-channel."
      + "\n#private <loginID> <msg> -- Sends a private message to the specified client."
      + "\n#pub -- Sends a public message."
      + "\n#punt <loginID> -- Kicks client out of the chatroom."
      + "\n#quit -- Terminates the server and disconnects all clients."
      + "\n#setport <newport> -- Specify the port the server will listen on."
      + "\n#start -- Makes the server restart accepting connections."
      + "\n#stop -- Makes the server stop accepting new connections."
      + "\n#unblock -- Unblock messages from all blocked clients."
      + "\n#unblock <loginID> -- Unblock messages from the specified client."
      + "\n#warn <loginID> -- Sends a warning message to the specified client."
      + "\n#whoblockme -- List clients who are blocking messages from the server."
      + "\n#whoiblock -- List all clients that the server is blocking messages from."
      + "\n#whoison -- Gets a list of all users and channel they are connected to.");
      return;
    }

    //If not a server-side command or is a message is to be displayed
    if (!(message.startsWith("#"))) {
      serverUI.display("SERVER MESSAGE> " + message);
      sendChannelMessage("SERVER MESSAGE> " + message, (serverChannel == null ? "main" : serverChannel), "server");
    } else {
      serverUI.display("Invalid command.");
    }
  }

  /**
   * This method gracefully kills the server.
   */
  public void quit() {
    try {
      closing = true;
      sendToAllClients("Server is quitting.");
      sendToAllClients("You will be disconnected.");
      server.close();
    } catch(IOException e) {}
    System.exit(0);
  }

  /**
   * This method overrides the one in the superclass.  Called
   * when the server starts listening for connections.
   */
  protected void serverStarted() {
    if (server.getNumberOfClients() != 0)
      sendToAllClients("Server has restarted accepting connections.");

    serverUI.display("Server listening for connections on port " +
server.getPort());
  }

  /**
   * This method overrides the one in the superclass.  Called
   * when the server stops listening for connections.
   */
  protected void serverStopped() {
    serverUI.display("Server has stopped listening for connections.");

    // If server is closing, the clients have already been notified.
    if (!closing)
      sendToAllClients("WARNING - Server has stopped accepting clients.");
  }

  /**
   * This method overrides the one in the superclass.  Called
   * when the server closes down.
   */
  protected void serverClosed() {
    serverUI.display("Server is closed.");
  }

  /**
   * This method overrides the one in the superclass.  Called
   * when the server stops listening for connections.
   */
  protected void listeningException(Throwable exception) {
    serverUI.display("An error has occured while listening.");
  }

  /**
   * This method is called when a client connects to the server.
   * Added in phase 2.
   * @param client The connection to the client who just connected.
   */
  protected void clientConnected(ConnectionToClient client) {
    serverUI.display("A new client is attempting to connect to the server.");
    client.setInfo("loginID", "");
    client.setInfo("channel", "");
    client.setInfo("passwordVerified", new Boolean(false));
    client.setInfo("creatingNewAccount", new Boolean(false));
    client.setInfo("fwdClient", "");
    client.setInfo("blockedUsers", new Vector());

    try {
      client.sendToClient("Enter your login ID:");
    } catch(IOException e) {
      try {
        client.close();
      } catch (IOException ex) {}
    }
  }

  /**
   * This method is called when a client disconnects from the server.
   * Added in phase 2.
   *
   * @param client The connection to the client who disconnected.
   */
  protected synchronized void clientDisconnected(ConnectionToClient client) {
    handleDisconnect(client);
  }

  /**
   * This method is called when an exception is detected in
   * ConnectionToClient.
   *
   * @param client The client who caused the exception
   * @param exception The exception thrown.
   */
  synchronized protected void clientException(ConnectionToClient client, Throwable exception) {
    handleDisconnect(client);
  }

  // Private methods ------------------------------------------------

  private void handleCmdWhoiblock(ConnectionToClient client) {
    Vector blocked;

    // If the client is not the server
    if (client != null) {
      blocked = new Vector((Vector)(client.getInfo("blockedUsers")));
    } else {
      blocked = new Vector(blockedUsers);
    }

    Iterator blockedIterator = blocked.iterator();

    // If some clients are blocked
    if (blockedIterator.hasNext()) {
      sendToClientOrServer(client,"BLOCKED USERS:");

      // Send the list of blocked users to the client
      while (blockedIterator.hasNext()) {
        String blockedUser = (String)blockedIterator.next();
        sendToClientOrServer(client, "Messages from " + blockedUser + " are blocked.");
      }
    } else {
      // No clients are blocked
      sendToClientOrServer(client, "No blocking is in effect.");
    }
  }

  private void handleCmdUnblock(String command, ConnectionToClient client) {
    Vector blocked = null;
    boolean removedUser = false;
    String userToUnblock = null;

    //If the client is not the server
    if (client != null) {
      blocked = (Vector)(client.getInfo("blockedUsers"));
    } else {
      blocked = blockedUsers;
    }

    // Check if any users were blocked.
    // If none were, notify the client
    if (blocked.size() == 0) {
      sendToClientOrServer(client, "No blocking is in effect.");
      return;
    }

    // Obtain the user to unblock. If no user is specified, then
    // an exception will be thrown and all users will be removed.
    try {
      userToUnblock = command.substring(9);
    } catch(StringIndexOutOfBoundsException e) {
      // We will unblock all users
      userToUnblock = "";
    }

    // If we want to unblock the server.
    if (userToUnblock.toLowerCase().equals("server"))
      userToUnblock = "server";

    // Get rid of the blocked user or all blocked users
    Iterator blockedIterator = blocked.iterator();
    while (blockedIterator.hasNext()) {
      String blockedUser = (String)blockedIterator.next();

      if(blockedUser.equals(userToUnblock) || userToUnblock.equals("")) {
        blockedIterator.remove();
        removedUser = true;
        sendToClientOrServer(client, "Messages from " + blockedUser + " will now be
displayed.");
      }
    }

    // Display error if user not found
    if(!removedUser) {
      sendToClientOrServer(client, "Messages from " + userToUnblock + " were not
blocked.");
    }
  }

  private void handleCmdBlock(String command, ConnectionToClient client) {
    Vector addBlock = null;

    // This next line will verify a client was specified.  If not,
    // return an error message.
    try {
      // If there is no specified user to block we will go
      // to the catch block
      String userToBlock = command.substring(7);

      //If the user wants to block the server
      if (userToBlock.toLowerCase().equals("server")) {
        userToBlock = "server";
      }

      // If the user tries to block himself
      if (userToBlock.equals(client.getInfo("loginID"))) {
        try {
```

```java
          client.sendToClient("Cannot block the sending of messages to yourself.");
        } catch(IOException ex) {
         serverUI.display("Warning: Error sending message.");
        }
        return;
      } else {
        // Blocking another user
        // Verify if the login to block is valid
        if (isLoginUsed(userToBlock) || userToBlock.equals("server")) {
          // If the user we want to block is online
          if (isLoginBeingUsed(userToBlock, false) && !userToBlock.equals("server")) {
            ConnectionToClient toBlock = getClient(userToBlock);

            // If that user is forwarding to the client requesting
            // the block, end the forwarding and notify them both.
            if (((String)(toBlock.getInfo("fwdClient"))).equals(((String)(client.getInfo("loginID"))))) {
              toBlock.setInfo("fwdClient", "");
              try {
                toBlock.sendToClient("Forwarding to " + client.getInfo("loginID")
                     + " has been cancelled because " + client.getInfo("loginID")
                     + " is now blocking messages from you.");

                client.sendToClient("Forwarding from " + toBlock.getInfo("loginID")
                     + " to you has been terminated.");
              } catch(IOException ioe) {
                serverUI.display("Warning: Error sending message.");
              }
            }
          }

          //Add the blocked user to the user's blocked users vector
          addBlock = (Vector)(client.getInfo("blockedUsers"));
          addBlock.addElement(userToBlock);
        }
        //If the user is trying to block a non-existing user.
        else {
          try {
            client.sendToClient("User " + userToBlock + " does not exist.");
          } catch(IOException ioe) {
            serverUI.display("Warning: Error sending message.");
          }
          return;
        }

        //Send confirmation to the client that the user's messages will now be blocked.
        try {
          client.sendToClient("Messages from " + userToBlock + " will be blocked.");
        } catch(IOException ex) {
          serverUI.display("Warning: Error sending message.");
        }
      }
    } catch(StringIndexOutOfBoundsException e) {
      try {
        client.sendToClient("ERROR - usage #block <loginID>");
      } catch(IOException ex) {
        serverUI.display("Warning: Error sending message.");
      }
    }
  }

  private void handleCmdFwd(String command, ConnectionToClient client) {
    try {
      String destineeName = command.substring(5);

      try {
        // If the client is trying to forward to himself.
        if (destineeName.equals(client.getInfo("loginID"))) {
          client.sendToClient("ERROR - Can't forward to self");
          return;
        } else {
          // If the client is trying to forward to the server
          if (destineeName.toLowerCase().equals("server")) {
            client.sendToClient("ERROR - Can't forward to SERVER");
            return;
          } else {
            // If the client specified a non-existing client.
            if (getClient(destineeName) == null) {
              client.sendToClient("ERROR - Client does not exist");
              return;
            }
          }
        }
      } catch(IOException e) {
        serverUI.display("Warning: Error sending message.");
      }

      // Find out if we are already forwarding. This will be used
      // later when we check for a forwarding loop
      String tempFwdClient = (String)(client.getInfo("fwdClient"));

      // Get the connection to the intended destinee.
      ConnectionToClient destinee = getClient(destineeName);

      // If the destinee is not blocking messages from the client
      // requesting the forwarding.
      if (!(((Vector)(destinee.getInfo("blockedUsers"))).contains((String)
(client.getInfo("loginID"))))) {
        client.setInfo("fwdClient", destineeName);
      } else {
        try {
          client.sendToClient("Cannot forward to " + destineeName
            + " because " + destineeName + " is blocking messages from you.");
        } catch(IOException e) {
          serverUI.display("Warning: Error sending message.");
        }
        return;
      }

      try {
        // If the client can be forwarded to without causing a loop
        if (isValidFwdClient(client)) {
          client.sendToClient("Messages will be forwarded to: " +
client.getInfo("fwdClient"));
        } else {
          // Reset forwarding to original value
          client.setInfo("fwdClient", tempFwdClient);
          client.sendToClient("ERROR - Can't forward because a loop would result");
```

```java
        }
      } catch(IOException e) {
        serverUI.display("Warning: Error sending message.");
      }
    } catch (StringIndexOutOfBoundsException e) {
      try {
        client.sendToClient("ERROR - usage: #fwd <loginID>");
      } catch(IOException ex) {
        serverUI.display("Warning: Error sending message.");
      }
    }
  }

  private void handleCmdPub(String command, ConnectionToClient client) {
    String sender = "";
    try {
      sender = (String)(client.getInfo("loginID"));
    } catch(NullPointerException e) {
      sender = "server";
    }

    try {
      Thread[] clients = server.getClientConnections();

      for (int i = 0; i < clients.length; i++) {
        ConnectionToClient c = (ConnectionToClient)(clients[i]);

        // If the client selected by the iterator is not blocking messages from the
sender.
        if (!(((Vector)(c.getInfo("blockedUsers"))).contains(sender))
          && ((Boolean)(c.getInfo("passwordVerified"))).booleanValue()) {
          c.sendToClient("PUBLIC MESSAGE from " + sender
            + "> " + command.substring(5));
        }
      }

      // If the server is not blocking messages from the sender.
      if (!blockedUsers.contains(sender)) {
        serverUI.display("PUBLIC MESSAGE from " + sender
          + "> " + command.substring(5));
      }
    } catch(IOException e) {
      serverUI.display("Warning: Error sending message.");
    }
  }

  private void handleCmdChannel(String command, ConnectionToClient client) {
    String oldChannel = (String)(client.getInfo("channel"));

    // Default new channel is the original channel that users start in
    String newChannel = "main";

    if(command.length() > 9)
      newChannel = command.substring(9);

    client.setInfo("channel", newChannel);
    if (!oldChannel.equals("main")) {
      sendChannelMessage(client.getInfo("loginID")
         + " has left channel: " + oldChannel, oldChannel, "");
    }

    if (!newChannel.equals("main")) {
      sendChannelMessage(client.getInfo("loginID")
         + " has joined channel: " + newChannel, newChannel, "");
    }

    // If the server receives all messages or is in the same channel
    // as the client requesting the change, it will display a message
    // indicating the change.
    if (serverChannel == null || serverChannel.equals(client.getInfo("channel"))) {
      serverUI.display(client.getInfo("loginID") + " has joined channel: " + newChannel);
    }
  }

  private void handleCmdPrivate(String command, ConnectionToClient client) {
    try {
      // Indicates where the spaces are in the command
      int firstSpace = command.indexOf(" ");
      int secondSpace = command.indexOf(" ", firstSpace + 1);

      // Separate the different parts of the command
      // These can throw the StringIndexOutOfBoundsException
      String sender = "";
      String loginID = command.substring(firstSpace + 1, secondSpace);
      String message = command.substring(secondSpace + 1);

      try {
        sender = (String)(client.getInfo("loginID"));
      } catch (NullPointerException e) {
        sender = "server";
      }

      // If the message is for the server, display it and return
      if (loginID.toLowerCase().equals("server")) {
        //If the server is not blocking messages from the sender
        if (!blockedUsers.contains(sender)) {
          serverUI.display("PRIVATE MESSAGE from " + sender + "> " + message);
        }
        //If the server is blocking messages from the sender.
        else {
          try {
            client.sendToClient("Cannot send message because " + loginID + " is blocking messages from you.");
          } catch(IOException e) {
            serverUI.display("Warning: Error sending message.");
          }
        }
      }
      // If the message is not for the server
      else {
        try {
          Thread[] clients = server.getClientConnections();

          //Iterate through all the clients to find the destinee
          for (int i = 0; i < clients.length; i++) {
            ConnectionToClient c = (ConnectionToClient)(clients[i]);

            if (c.getInfo("loginID").equals(loginID)) {
```

```
       // Once found, check if the user is not blocking messages from the sender.
       if (!(((Vector)(c.getInfo("blockedUsers"))).contains(sender))) {

        // If he is not, check for a client to forward messages to.
        if (!c.getInfo("fwdClient").equals("")) {
          getFwdClient(c, sender).sendToClient("Forwarded> PRIVATE MESSAGE from "
           + sender + " to " + c.getInfo("loginID") + "> " + message);
        } else {
          c.sendToClient("PRIVATE MESSAGE from " + sender + "> " + message);
        }
        serverUI.display("Private message: \""
         + message + "\" from " + sender + " to " + c.getInfo("loginID"));
       }
       //If the user is blocking messages from the sender.
       else {
        sendToClientOrServer(client, "Cannot send message because "
         + loginID + " is blocking messages from you.");
       }
      }
     }
    } catch(IOException e) {
     serverUI.display("Warning: Error sending message.");
    }
   }
  } catch (StringIndexOutOfBoundsException e) {
   sendToClientOrServer(client, "ERROR - usage: #private <loginID> <msg>");
  }
 }

 private void checkForBlocks(String login, ConnectionToClient client) {
  String results = "User block check:";

  if (!login.equals("server")) {
   if (blockedUsers.contains(login))
    results += "\nThe server is blocking messages from you.";
  }

  Thread[] clients = server.getClientConnections();

  for (int i = 0; i < clients.length; i++) {
   ConnectionToClient c = (ConnectionToClient)(clients[i]);

   Vector blocked = (Vector)(c.getInfo("blockedUsers"));
   if (blocked.contains(login)) {
    results += "\nUser " + c.getInfo("loginID") + " is blocking your messages.";
   }
  }
  if (results.equals("User block check:")) {
   results += "\nNo user is blocking messages from you.";
  }

  sendToClientOrServer(client, results);
 }

 private boolean isValidFwdClient(ConnectionToClient client) {
  boolean clientFound = false;
  ConnectionToClient testClient = client;
```

```java

    // This block will make sure the client exists
    Thread[] clients = server.getClientConnections();
    for (int i = 0; i < clients.length; i++) {
      ConnectionToClient tempc = (ConnectionToClient)(clients[i]);
      if (tempc.getInfo("loginID").equals(testClient.getInfo("fwdClient"))) {
        clientFound = true;
      }
    }

    if (!clientFound)
      return false;

    // This block will check for endless loops
    String theClients[] = new String[server.getNumberOfClients() + 1];
    int i = 0;

    // Loops until it finds a client that doesn't forward
    while (testClient != null && testClient.getInfo("fwdClient")!="") {
      // The name is added to the array
      theClients[i] = (String)(testClient.getInfo("loginID"));

      // If the name is in the array, return false as there is an endless loop
      for(int j = 0; j < i; j++) {
        if (theClients[j].equals(theClients[i]))
          return false;
      }
      i++;

      // Set "testClient" to the forwarded ConnectionToClient instance
      testClient = getClient((String)testClient.getInfo("fwdClient"));
    }
    return true;
  }

  private ConnectionToClient getClient(String loginID) {
    Thread[] clients = server.getClientConnections();

    for (int i = 0; i < clients.length; i++) {
      ConnectionToClient c = (ConnectionToClient)(clients[i]);
      if (c.getInfo("loginID").equals(loginID))
        return c;
    }
    return null; // If client wasn't found, return null
  }

  private void clientLoggingIn(String message, ConnectionToClient client) {
    // Ignore blanks, if the user just hits 'enter'
    if (message.equals(""))
      return;

    // If the client has not logged in yet and has entered
    // guest as his login, create a new account
    if ((client.getInfo("loginID").equals("")) && (message.equals("guest"))) {
      // Save a flag so that when the next message arrives we
      // know that it is the login ID for the new account
      client.setInfo("creatingNewAccount", new Boolean(true));

      try {
        client.sendToClient("\n*** CREATING NEW ACCOUNT ***\nEnter new LoginID :");
      } catch(IOException e) {
        try {
          client.close();
        } catch (IOException ex) {}
      }
    } else {
      // If creating a new account, and the user has just submitted his new login, process it
      if ((client.getInfo("loginID").equals(""))
         && (((Boolean)(client.getInfo("creatingNewAccount"))).booleanValue())) {
        client.setInfo("loginID", message);
        try {
          client.sendToClient("Enter new password :");
        } catch(IOException e) {
          try {
            client.close();
          } catch (IOException ex) {}
        }
      } else {
        // If the client is creating a new account and has just
        // entered the password, then process it
        if ((!client.getInfo("loginID").equals(""))
           && (((Boolean)(client.getInfo("creatingNewAccount"))).booleanValue())) {
          // If the login is not in the password file, accept the new account
          if (!isLoginUsed((String)(client.getInfo("loginID")))) {
            client.setInfo("passwordVerified", new Boolean(true));
            client.setInfo("creatingNewAccount", new Boolean(false));
            client.setInfo("channel", "main");

            addClientToRegistry((String)(client.getInfo("loginID")), message);
            serverUI.display(client.getInfo("loginID") + " has logged on.");
            sendToAllClients(client.getInfo("loginID") + " has logged on.");
          } else {
            // If creating a new account, but the login is already used then keep prompting for a login
            client.setInfo("loginID", "");
            client.setInfo("creatingNewAccount", new Boolean(false));
            try {
              client.sendToClient("Login already in use.  Enter login ID:");
            } catch(IOException e) {
              try {
                client.close();
              } catch (IOException ex) {}
            }
          }
        }
        // If the client is not creating a new account and has entered a login
        else {
          if (client.getInfo("loginID").equals("")) {
            client.setInfo("loginID", message);
            try {
              client.sendToClient("Enter password:");
            } catch(IOException e) {
              try {
                client.close();
              } catch (IOException ex) {}
            }
          } else {
            // If the client is not creating a new account and has entered a password
```

```java
      // Verify the client's login.
      if ((isValidPwd((String)(client.getInfo("loginID")), message, true))
        && (!isLoginBeingUsed((String)(client.getInfo("loginID")), true))) {
       client.setInfo("passwordVerified", new Boolean(true));
       client.setInfo("channel", "main");

       // notify all users that a new client has logged on
       serverUI.display(client.getInfo("loginID") + " has logged on.");
       sendToAllClients(client.getInfo("loginID") + " has logged on.");
      } else {
       // If the login id or the password is invalid
       try {
        if (isLoginBeingUsed((String)(client.getInfo("loginID")), true)) {
         client.setInfo("loginID", "");
         client.sendToClient("Login ID is already logged on.\nEnter LoginID:");
        } else {
         client.setInfo("loginID", "");
         client.sendToClient("\nIncorrect login or password\nEnter LoginID:");
        }
       } catch(IOException e) {
        try {
         client.close();
        } catch (IOException ex) {}
       }
      }
     }
    }
   }
  }
 }

 private void addClientToRegistry(String clientLoginID, String clientPassword) {
  try {
   // Part 1 : Transfer the data from the password file to a character buffer
   FileInputStream inputFile = new FileInputStream(PASSWORDFILE);
   byte buff[] = new byte[inputFile.available()];

   for (int i = 0; i < buff.length; i++) {
    int character = inputFile.read();
    buff[i] = (byte)character;
   }
   inputFile.close(); // Close the input stream

   // Part 2 : Delete the password file since it will be created again
   File fileToBeDeleted = new File(PASSWORDFILE);
   fileToBeDeleted.delete();

   // Part 3 : Transfer the buffer and the client data to a new password file with the same name as the first
   FileOutputStream outputFile = new FileOutputStream(PASSWORDFILE);
   for(int i = 0; i < buff.length; i++) // Write the buffer
     outputFile.write(buff[i]);

   for(int i = 0; i < clientLoginID.length(); i++)
    outputFile.write(clientLoginID.charAt(i));

   outputFile.write(SPACE); // Write a space character

```

```java
      for (int i = 0; i < clientPassword.length(); i++)
        outputFile.write(clientPassword.charAt(i));

      outputFile.write(RETURN); // Write a carriage return
      outputFile.write(LINEBREAK); // Write a line break

      outputFile.close(); // Close the output stream
    } catch (IOException e) {
      serverUI.display("ERROR - Password File Not Found");
    }
  }

  private boolean isLoginUsed(String loginID) {
    //See if the loginID is in the password file. The "false"
    //indicates not to verify the password
    return isValidPwd(loginID, "", false);
  }

  private boolean isValidPwd(String loginID, String password, boolean
verifyPassword) {
    try {
      FileInputStream inputFile = new FileInputStream(PASSWORDFILE);
      boolean eoln = false; // Flag indicating the End Of Line
      boolean eof = false;  // Flag indicating the End Of File

      while (!eof) {
        eoln = false;
        String str = "";
        while (!eoln) {
          int character = inputFile.read();

          if(character == -1) {
            eof = true;
            break;
          } else {
            if (character == LINEBREAK) {
              eoln = true;

              // Verifies if the loginID is identical to the loginID
              // in the file and, if necessary, verifies if the
              // password is also identical to the password in the
              // file
              if ((str.substring(0, str.indexOf(" ")).equals(loginID))
                && ((str.substring(str.indexOf(" ") + 1).equals(password)) ||
(!verifyPassword))) {
                return true;
              }

              // This condition checks if the char is anything other
              // than a carriage return. The carriage return is
              // ignored therefore there is no need to handle it
            } else {
              if (character != RETURN) {
                str = str + (char)character;
              }
            }
          }
        }
      }
      inputFile.close(); // Close the input stream
    } catch (IOException e) {
      serverUI.display("ERROR - Password File Not Found");
    }
    return false;
  }

  private boolean isLoginBeingUsed(String loginID, boolean checkForDup) {
    boolean used = !checkForDup;

    if (loginID.toLowerCase().equals("server"))
      return true;

    // Creates an Iterator containing all the clients
    Thread[] clients = server.getClientConnections();

    for (int i = 0; i < clients.length; i++) {
      ConnectionToClient tempc = (ConnectionToClient)(clients[i]);
      if (tempc.getInfo("loginID").equals(loginID)) {
        if (used)
          return true;

        used = true;
      }
    }
    return false; // The name was not found
  }

  private void sendChannelMessage(String message, String channel, String login) {
    Thread[] clients = server.getClientConnections();

    for (int i = 0; i < clients.length; i++) {
      ConnectionToClient c = (ConnectionToClient)(clients[i]);

      if (c.getInfo("channel").equals(channel)
         && !(((Vector)(c.getInfo("blockedUsers"))).contains(login))) {
        try {
          if (!(c.getInfo("fwdClient").equals(""))) {
            getFwdClient(c, login).sendToClient("Forwarded> " + message);
          } else {
            c.sendToClient(message);
          }
        } catch(IOException e) {
          serverUI.display("Warning: Error sending message.");
        }
      }
    }
  }

  private ConnectionToClient getFwdClient(ConnectionToClient c, String sender) {
    Vector pastRecipients = new Vector();

    //Add the first recipient to the vector
    pastRecipients.addElement((String)(c.getInfo("loginID")));

    // Loops until it finds a client that doesn't forward messages
    while (!c.getInfo("fwdClient").equals("")) {
      Thread[] clients = server.getClientConnections();

      for (int i = 0; i < clients.length; i++) {
        ConnectionToClient tempc = (ConnectionToClient)(clients[i]);
```

```java
        if (tempc.getInfo("loginID").equals(c.getInfo("fwdClient"))) {
          // We have found the client being forwarded to by c
          // Now check that c is not blocking the original sender
          if (!(((Vector)(tempc.getInfo("blockedUsers"))).contains(sender))) {
            //Look in the previous recipients to see if any of them are blocked.
            Iterator pastIterator = pastRecipients.iterator();

            while (pastIterator.hasNext()) {
              String pastRecipient = (String)pastIterator.next();
              if (((Vector)(tempc.getInfo("blockedUsers"))).contains(pastRecipient))
{
                //This means one of the past recipients is blocked
                //by the client supposed to be forwarded to.
                try {
                  c.sendToClient("Cannot forward message.  A past recipient of this
message is blocked by "
                    + (String)(tempc.getInfo("loginID")));
                } catch(IOException e) {
                  serverUI.display("Warning: Error sending message.");
                }
                return c;
              }
            }

            // Now continue looking for further forwarding  if necessary
            if(!tempc.getInfo("fwdClient").equals("")) {
              c = tempc;
              pastRecipients.addElement((String)(c.getInfo("loginID")));
            } else {
              return tempc;
            }
          } else {
            try {
              c.sendToClient("Cannot forward message.  Original sender is blocked by
"
                           + ((String)(c.getInfo("fwdClient"))));
            } catch(IOException e) {
              serverUI.display("Warning: Error sending message.");
            }
            return c;
          }
        }
      }
    }
    return c;
  }

  private void sendListOfClients(ConnectionToClient c) {
    Vector clientInfo = new Vector();

    Thread[] clients = server.getClientConnections();

    for (int i = 0; i < clients.length; i++) {
      ConnectionToClient tempc = (ConnectionToClient)(clients[i]);
      clientInfo.addElement((String)(tempc.getInfo("loginID"))
        + " --- on channel: " + (String)(tempc.getInfo("channel")));
    }

    //Sort the vector containing the information.
    Collections.sort(clientInfo);

    if (server.isListening() || server.getNumberOfClients() != 0) {
      sendToClientOrServer(c, "SERVER --- on channel: "
        + (serverChannel == null ? "main" : serverChannel));
    } else {
      serverUI.display("SERVER --- no active channels");
    }

    Iterator toReturn = clientInfo.iterator();

    while (toReturn.hasNext()) {
      sendToClientOrServer(c, (String)toReturn.next());
    }
  }

  private void handleServerCmdBlock(String message) {
    try {
      String userToBlock = message.substring(7);

      if (userToBlock.toLowerCase().equals("server")) {
        serverUI.display("Cannot block the sending of messages to yourself.");
        return;
      } else {
        if (isLoginUsed(userToBlock)) {
          blockedUsers.addElement(userToBlock);
        } else {
          serverUI.display("User " + userToBlock + " does not exist.");
          return;
        }
      }

      serverUI.display("Messages from " + userToBlock + " will be blocked.");
    } catch(StringIndexOutOfBoundsException e) {
      serverUI.display("ERROR - usage #block <loginID>");
    }
  }

  private void handleServerCmdPunt(String message) {
    Thread[] clients = server.getClientConnections();

    try {
      //Iterate to get the connection to the client we want to expell
      for (int i = 0; i < clients.length; i++) {
        ConnectionToClient c = (ConnectionToClient)(clients[i]);
        if (c.getInfo("loginID").equals(message.substring(6))) {
          //Ignore the exception that might occur as we only want
          //to get rid of this user.
          try {
            c.sendToClient("You have been expelled from this server.");
          } catch(IOException e) {}
          finally {
            try {
              c.close();
            }
            catch (IOException ex) {}
          }
        }
      }
    } catch(StringIndexOutOfBoundsException ex) {
      serverUI.display("Invalid use of the #punt command.");
    }
  }

  private void handleServerCmdWarn(String message) {
    Thread[] clients = server.getClientConnections();

    try {
      for (int i = 0; i < clients.length; i++) {
        ConnectionToClient c = (ConnectionToClient)(clients[i]);
        if (c.getInfo("loginID").equals(message.substring(6))) {
          //If an exception occurs, boot the user being warned.
          //He is causing more trouble than he's worth!
          try {
            c.sendToClient("Continue and you WILL be expelled.");
          } catch(IOException e) {
            try {
              c.close();
            } catch (IOException ex) {}
          }
        }
      }
    } catch(StringIndexOutOfBoundsException ex) {
      serverUI.display("Invalid use of the #warn command.");
    }
  }

  private void sendToClientOrServer(ConnectionToClient client, String message) {
    try {
      client.sendToClient(message);
    } catch(NullPointerException npe) {
      serverUI.display(message);
    } catch(IOException ex) {
      serverUI.display("Warning: Error sending message.");
    }
  }

  private void handleDisconnect(ConnectionToClient client) {
    if (!client.getInfo("loginID").equals("")) {
      try {
        Thread[] clients = server.getClientConnections();

        // Remove any forwarding to this client by others.
        for (int i = 0; i < clients.length; i++) {
          ConnectionToClient c = (ConnectionToClient)(clients[i]);
          if (client.getInfo("loginID").equals(c.getInfo("fwdClient"))) {
            c.setInfo("fwdClient", "");
            c.sendToClient("Forwarding to " + client.getInfo("loginID")+ " has been cancelled.");
          }
        }
        sendToAllClients(((client.getInfo("loginID") == null) ?
            "" : client.getInfo("loginID")) + " has disconnected.");
      } catch(IOException e) {
        serverUI.display("Warning: Error sending message.");
      }
      serverUI.display(client.getInfo("loginID") + " has disconnected.");
    }
  }
}
```