COMP 303 - Lecture Notes for Week 2

- Slides edited from, Object-Oriented Design & Patterns, by Cay S. Horstmann
- Original slides available from: http://www.horstmann.com/design_and_patterns.html
- Modifications made by Laurie Hendren, McGill University
- Topics this week:
  - The Object-Oriented Design Process, Chapter 2
  - Using subversion for version control

---

The Object-Oriented Design Process - Overview

- Concepts
  - From Problem to Code
  - The Object and Class Concepts
  - Identifying Classes
  - Identifying Responsibilities
  - Relationships Between Classes
- Techniques and Tools
  - Use Cases
  - CRC Cards
  - UML Class Diagrams
  - Sequence Diagrams
  - State Diagrams
  - Using javadoc for Design Documentation
  - Case Study: A Voice Mail System

---

From Problem to Code

Three Phases:
- Analysis
- Design
- Implementation

Case Study: Voice Mail System

---

Analysis Phase: What needs to be done

Functional Specification: detailed textual description

- Completely defines tasks to be solved
- Free from internal contradictions
- Readable both by domain experts and software developers
- Reviewable by diverse interested parties
- Testable against reality
Design Phase: Structuring programming tasks into set of interrelated classes

Goals

- Identify classes
- Identify behavior of classes
- Identify relationships among classes

Artifacts

- Textual description of classes and key methods
- Diagrams of class relationships
- Diagrams of important usage scenarios
- State diagrams for objects with rich state

Implementation Phase: Programming

- Implement and test classes
- Combine classes into program
- Avoid "big bang" integration
- Prototypes can be very useful

Object and Class Concepts

- **Object**: Three characteristic concepts
  - State
  - Behavior
  - Identity
- **Class**: Collection of similar objects

Identifying Classes

Rule of thumb: Look for **nouns** in functional description

- Mailbox
- Message
- User
- Passcode
- Extension
- Menu
Identifying Classes
Focus on concepts, not implementation
- MessageQueue stores messages
- Don’t worry yet how the queue is implemented

Categories of Classes
- Tangible Things
- Agents
- Events and Transactions
- Users and Roles
- Systems
- System interfaces and devices
- Foundational Classes

Identifying Responsibilities
Rule of thumb: Look for verbs in problem description
Behavior of MessageQueue:
- Add message to tail
- Remove message from head
- Test whether queue is empty

Responsibilities
- OO Principle: Every operation is the responsibility of a single class
- Example: Add message to mailbox
- Who is responsible: Message or Mailbox?

Class Relationships
1. Dependency ("uses")
2. Aggregation ("has")
3. Inheritance ("is")

Class Relationships (1): Dependency Relationships
- C depends on D: Method of C manipulates objects of D
- Example: Mailbox depends on Message
- If C doesn’t use D, then C can be developed without knowing about D
**Coupling**
- Minimize dependency: reduce coupling
- Example: Replace `void print() // prints to System.out` with `String getText() // can print anywhere`
- Removes dependence on `System`, `PrintStream`

**Class Relationships (2): Aggregation**
- Object of a class contains objects of another class
- Example: `MessageQueue` aggregates `Messages`
- Example: `Mailbox` aggregates `MessageQueue`
- Implemented through instance fields

**Multiplicities**

1 : 1 or 1 : 0...1 relationship:

```java
public class Mailbox {
    private Greeting myGreeting;
}
```

1 : n relationship:

```java
public class MessageQueue {
    private ArrayList elements;
}
```

**Class Relationships (3): Inheritance**

- More general class = superclass
- More specialized class = subclass
- Subclass supports all method interfaces of superclass (but implementations may differ)
- Subclass may have added methods, added state
- Subclass inherits from superclass
- Example: `ForwardedMessage` inherits from `Message`
- Example: `Greeting does not inherit from Message` (Can’t store greetings in mailbox)
Use Cases

- Analysis technique
- Each use case focuses on a specific scenario
- Use case = sequence of actions
- Action = interaction between actor and computer system
- Each action yields a result
- Each result has a value to one of the actors
- Use variations for exceptional situations

Sample Use Case

**Leave a Message**

1. Caller dials main number of voice mail system
2. System speaks prompt: "Enter mailbox number followed by ".
3. User types extension number
4. System speaks: "You have reached mailbox xxxx. Please leave a message now."
5. Caller speaks message
6. Caller hangs up
7. System places message in mailbox

Sample Use Case -- Variations

**Variation #1**

1.1. In step 3, user enters invalid extension number
1.2. Voice mail system speaks: "You have typed an invalid mailbox number."
1.3. Continue with step 2.

**Variation #2**

2.1. After step 4, caller hangs up instead of speaking message
2.3. Voice mail system discards empty message

CRC Cards

- CRC = Classes, Responsibilities, Collaborators
- Developed by Beck and Cunningham
- Use an index card for each class
- Class name on top of card
- Responsibilities on left
- Collaborators on right
CRC Cards

<table>
<thead>
<tr>
<th>Mailbox</th>
</tr>
</thead>
<tbody>
<tr>
<td>manage passcode</td>
</tr>
<tr>
<td>manage greeting</td>
</tr>
<tr>
<td>manage new and saved messages</td>
</tr>
</tbody>
</table>

- Responsibilities should be high level
- 1 - 3 responsibilities per card
- Collaborators are for the class, not for each responsibility

Walkthroughs

- Use case: "Leave a message"
- Caller connects to voice mail system
- Caller dials extension number
- "Someone" must locate mailbox
- Neither Mailbox nor Message can do this
- New class: MailSystem
- Responsibility: manage mailboxes

Walkthroughs

<table>
<thead>
<tr>
<th>MailSystem</th>
</tr>
</thead>
<tbody>
<tr>
<td>manage mailboxes</td>
</tr>
</tbody>
</table>

UML Diagrams (a more formal tool)

- UML = Unified Modeling Language
- Unifies notations developed by the "3 Amigos" Booch, Rumbaugh, Jacobson
- Many diagram types
- We’ll use three types:
  1. Class Diagrams
  2. Sequence Diagrams
  3. State Diagrams
UML (1): Class Diagrams

- Rectangle with class name
- Optional compartments
  - Attributes
  - Methods
- Include only key attributes and methods

UML Connectors - Class Relationships

- Dependency
- Aggregation
- Inheritance
- Composition
- Association
- Directed Association
- Interface Type Implementation

Multiplicities

- any number (0 or more): *
- one or more: 1..*
- zero or one: 0..1
- exactly one: 1
**Composition**
- Special form of aggregation
- Contained objects don’t exist outside container
- Example: message queues permanently contained in mail box

**Association**
- Some designers don’t like aggregation
- More general association relationship
- Association can have roles

**Interface Types**
- Interface type describes a set of methods
- No implementation, no state
- Class implements interface if it implements its methods
- In UML, use stereotype "interface"
Tips
- Use UML to inform, not to impress
- Don’t draw a single monster diagram
- Each diagram must have a specific purpose
- Omit inessential details

UML (2): Sequence Diagrams
- Each diagram shows dynamics of scenario
- Object diagram: class name underlined

Sequence Diagrams: Self call

Sequence Diagrams: Object Construction
**UML (3): State Diagrams**

- Use for classes whose objects have interesting states

![State Diagram](#)

**Design Documentation**

- Recommendation: Use Javadoc comments
- Leave methods blank

```java
/**
 * Adds a message to the end of the new messages.
 * @param aMessage a message
 */
public void addMessage(Message aMessage) {
}
```

- Don't compile file, just run Javadoc
- Makes a good starting point for code later

**Case Study: Voice Mail System**

- Use text for voice, phone keys, hangup
- 1 2 ... 0 # on a single line means key
- H on a single line means "hang up"
- All other inputs mean voice
- In GUI program, will use buttons for keys (see ch. 4)

**Use Case: Reach an Extension**

1. User dials main number of system
2. System speaks prompt: "Enter mailbox number followed by #"
3. User types extension number
4. System speaks: "You have reached mailbox xxxx. Please leave a message now"
Use Case: Leave a Message

1. Caller carries out Reach an Extension
2. Caller speaks message
3. Caller hangs up
4. System places message in mailbox

Use Case: Log in

1. Mailbox owner carries out Reach an Extension
2. Mailbox owner types password and #
   (Default password = mailbox number. To change, see Change the Passcode)
3. System plays mailbox menu:
   "Enter 1 to retrieve your messages."
   "Enter 2 to change your passcode."
   "Enter 3 to change your greeting."

Use Case: Retrieve Messages

1. Mailbox owner carries out Log in
2. Mailbox owner selects "retrieve messages" menu option
3. System plays message menu:
   "Press 1 to listen to the current message"
   "Press 2 to delete the current message"
   "Press 3 to save the current message"
   "Press 4 to return to the mailbox menu"
4. Mailbox owner selects "listen to current message"
5. System plays current new message, or, if no more new messages, current old message.
   Note: Message is played, not removed from queue
6. System plays message menu
7. User selects "delete current message". Message is removed.
8. Continue with step 3.

Use Case: Retrieve Messages

Variation #1

1.1. Start at Step 6
1.2. User selects "save current message".
   Message is removed from new queue and appended to old queue
1.3. Continue with step 3.
Use Case: Change the Greeting

1. Mailbox owner carries out Log in
2. Mailbox owner selects “change greeting” menu option
3. Mailbox owner speaks new greeting
4. Mailbox owner presses #
5. System sets new greeting

Variation #1: Hang up before confirmation

1.1. Start at step 3.
1.2. Mailbox owner hangs up.
1.3. System keeps old greeting.

Use Case: Change the Passcode

1. Mailbox owner carries out Log in
2. Mailbox owner selects “change passcode” menu option
3. Mailbox owner dials new passcode
4. Mailbox owner presses #
5. System sets new passcode

Variation #1: Hang up before confirmation

1.1. Start at step 3.
1.2. Mailbox owner hangs up.
1.3. System keeps old passcode.
CRC Cards for Voice Mail System

Some obvious classes
- Mailbox
- Message
- MailSystem

Initial CRC Cards: Mailbox

<table>
<thead>
<tr>
<th>Mailbox</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>keep new and saved messages</em></td>
</tr>
<tr>
<td>MessageQueue</td>
</tr>
</tbody>
</table>

Initial CRC Cards: MessageQueue

<table>
<thead>
<tr>
<th>MessageQueue</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>add and remove messages in FIFO order</em></td>
</tr>
</tbody>
</table>

Initial CRC Cards: MailSystem

<table>
<thead>
<tr>
<th>MailSystem</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>manage mailboxes</em></td>
</tr>
<tr>
<td>Mailbox</td>
</tr>
</tbody>
</table>
Telephone

- Who interacts with user?
- Telephone takes button presses, voice input
- Telephone speaks output to user

Connection

- With whom does Telephone communicate?
- With MailSystem?
- What if there are multiple telephones?
- Each connection can be in different state
  (dialing, recording, retrieving messages,...)
- Should mail system keep track of all connection states?
- Better to give this responsibility to a new class.
Analyze Use Case: Leave a message

1. User dials extension. Telephone sends number to Connection
   (Add collaborator Telephone to Connection)
2. Connection asks MailSystem to find matching Mailbox
3. Connection asks Mailbox for greeting
   (Add responsibility "manage greeting" to Mailbox,
   add collaborator Mailbox to Connection)
4. Connection asks Telephone to play greeting
5. User speaks greeting. Telephone asks Connection to record it.
   (Add responsibility "record voice input" to Connection)
6. User hangs up. Telephone notifies Connection.
7. Connection constructs Message
   (Add card for Message class, add collaborator Message to Connection)
8. Connection adds Message to Mailbox

Result of Use Case Analysis (Telephone)

<table>
<thead>
<tr>
<th>Telephone</th>
</tr>
</thead>
<tbody>
<tr>
<td>take user input from touchpad,</td>
</tr>
<tr>
<td>microphone, hangup</td>
</tr>
</tbody>
</table>

Result of Use Case Analysis (Connection)

<table>
<thead>
<tr>
<th>Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>get input from telephone</td>
</tr>
<tr>
<td>carry out user commands</td>
</tr>
<tr>
<td>keep track of state</td>
</tr>
<tr>
<td>record voice input</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Result of Use Case Analysis (Mailbox)

<table>
<thead>
<tr>
<th>Mailbox</th>
</tr>
</thead>
<tbody>
<tr>
<td>MessageQueue</td>
</tr>
<tr>
<td>keep new and saved messages</td>
</tr>
<tr>
<td>manage greeting</td>
</tr>
</tbody>
</table>
Result of Use Case Analysis (Message)

<table>
<thead>
<tr>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>manage message contents</td>
</tr>
</tbody>
</table>

Result of Use Case Analysis (Mailbox)

<table>
<thead>
<tr>
<th>Mailbox</th>
</tr>
</thead>
<tbody>
<tr>
<td>keep new and saved messages</td>
</tr>
<tr>
<td>manage greeting</td>
</tr>
<tr>
<td>manage passcode</td>
</tr>
<tr>
<td>retrieve, save, delete messages</td>
</tr>
</tbody>
</table>

Analyse Use Case: Retrieve messages

1. User types in passcode. Telephone notifies Connection.
2. Connection asks Mailbox to check passcode. (Add responsibility "manage passcode" to Mailbox)
3. Connection sets current mailbox and asks Telephone to speak menu.
4. User selects "retrieve messages". Telephone passes key to Connection.
5. Connection asks Telephone to speak menu.
6. User selects "listen to current message". Telephone passes key to Connection.
7. Connection gets first message from current mailbox. (Add "retrieve messages" to responsibility of Mailbox).
8. Connection asks Telephone to speak message.
9. User selects "save current message". Telephone passes key to Connection.
10. Connection tells Mailbox to save message. (Modify responsibility of Mailbox to "retrieve, save, delete messages")
11. Connection asks Telephone to speak menu.

CRC Summary

- One card per class
- Responsibilities at high level
- Use scenario walkthroughs to fill in cards
- Usually, the first design isn’t perfect.
  (You just saw the author’s third design of the mail system)
UML Class Diagram for Mail System

- CRC collaborators yield dependencies
- Mailbox depends on MessageQueue
- Mailbox and Connection depend on Mailbox
- Connection depends on Telephone, MailSystem, Message, Mailbox
- Telephone depends on Connection

Dependency Relationships

Aggregation Relationships

- A mail system has mailboxes
- A mailbox has two message queues
- A message queue has some number of messages
- A connection has a current mailbox
- A connection has references to a mail system and a telephone

UML Class Diagram for Voice Mail System
Sequence Diagram for Use Case: Leave a message

Interpreting a Sequence Diagram
- Each key press results in separate call to dial, but only one is shown
- Connection wants to get greeting to play
- Each mailbox knows its greeting
- Connection must find mailbox object:
  - Call findMailbox on MailSystem object
- Parameters are not displayed (e.g. mailbox number)
- Return values are not displayed (e.g. found mailbox)
- Note that connection holds on to that mailbox over multiple calls

Sequence Diagram for Use Case: Retrieve messages

Connection State Diagram
Java Implementation

- Ch2/mail/Message.java
- Ch2/mail/MessageQueue.java
- Ch2/mail/Mailbox.java
- Ch2/mail/Connection.java
- Ch2/mail/MailSystem.java
- Ch2/mail/Telephone.java
- Ch2/mail/MailSystemTest.java

```java
import java.util.ArrayList;

public class MessageQueue
{
    private ArrayList queue;

    /** Constructs an empty message queue. */
    public MessageQueue()
    {
        queue = new ArrayList();
    }

    /** Remove message at head. */
    public Message removeFirst()
    {
        return (Message) queue.get(0);
    }

    /** Append message at tail. */
    public void add(Message newMessage)
    {
        queue.add(newMessage);
    }

    /** Get message at head. */
    public Message getFirst()
    {
        return (Message) queue.get(0);
    }

    /** Get the total number of messages in the queue. */
    public int size()
    {
        return queue.size();
    }
}
```

```java
public class Message
{
    private String text;

    /** Construct a Message object. */
    public Message(String messageText)
    {
        text = messageText;
    }

    /** Get the message text. */
    public String getText()
    {
        return text;
    }
}
```

```java
import java.util.ArrayList;

public class MessageQueue
{
    private ArrayList queue;

    /** Constructs an empty message queue. */
    public MessageQueue()
    {
        queue = new ArrayList();
    }

    /** Remove message at head. */
    public Message removeFirst()
    {
        return (Message) queue.remove(0);
    }

    /** Append message at tail. */
    public void add(Message newMessage)
    {
        queue.add(newMessage);
    }

    /** Get message at head. */
    public Message getFirst()
    {
        return (Message) queue.get(0);
    }

    /** Get the total number of messages in the queue. */
    public int size()
    {
        return queue.size();
    }
}
```

```java
import java.util.ArrayList;

public class MessageQueue
{
    private ArrayList queue;

    /** Constructs an empty message queue. */
    public MessageQueue()
    {
        queue = new ArrayList();
    }

    /** Remove message at head. */
    public Message removeFirst()
    {
        return (Message) queue.remove(0);
    }

    /** Append message at tail. */
    public void add(Message newMessage)
    {
        queue.add(newMessage);
    }

    /** Get message at head. */
    public Message getFirst()
    {
        return (Message) queue.get(0);
    }

    /** Get the total number of messages in the queue. */
    public int size()
    {
        return queue.size();
    }
}
```
** A mailbox contains messages that can be listed, kept or discarded.

```java
/**
 * A mailbox contains messages that can be listed, kept or discarded.
 */
public class Mailbox
{

    private MessageQueue newMessages;
    private String greeting;
    private String passcode;

    /**
     * Creates Mailbox object.
     * @param aPasscode passcode number
     * @param aGreeting greeting string
     */
    public Mailbox(String aPasscode, String aGreeting)
    {
        passcode = aPasscode;
        greeting = aGreeting;
        newMessages = new MessageQueue();
        keptMessages = new MessageQueue();
    }

    /**
     * Get the current message.
     * @return the current message
     */
    public Message getCurrentMessage()
    {
        if (newMessages.size() > 0)
            return newMessages.getFirst();
        else if (keptMessages.size() > 0)
            return keptMessages.getFirst();
        else
            return null;
    }

    /**
     * Change mailbox’s passcode.
     * @param newPasscode the new passcode
     */
    public void setPasscode(String newPasscode)
    {
        passcode = newPasscode;
    }

    /**
     * Add a message to the mailbox.
     * @param aMessage the message to be added
     */
    public void addMessage(Message aMessage)
    {
        newMessages.add(aMessage);
    }

    /**
     * Check if the passcode is correct.
     * @param aPasscode a passcode to check
     * @return true if the supplied passcode matches the mailbox passcode
     */
    public boolean checkPasscode(String aPasscode)
    {
        return aPasscode.equals(passcode);
    }

    /**
     * Record voice.
     */
    public void recordVoice()
    {
        // Code to record voice
    }

    /**
     * Get the mailbox’s greeting.
     * @return the greeting
     */
    public String getGreeting()
    {
        return greeting;
    }

    /**
     * Change mailbox’s greeting.
     * @param newGreeting the new greeting string
     */
    public void setGreeting(String newGreeting)
    {
        greeting = newGreeting;
    }

    /**
     * Connects a phone to the mail system. The purpose of this
     * class is to keep track of the state of a connection, since
     * the phone itself is just a source of individual key presses.
     */
    public class Connection
    {

        /**
         * Construct a Connection object.
         * @param s a MailSystem object
         * @param p a Telephone object
         */
        public Connection(MailSystem s, Telephone p)
        {
            system = s;
            phone = p;
            resetConnection();
        }

        /**
         * Respond to the user’s pressing a key on the phone touchpad
         * @param key the key pressed by the user
         */
        public void dial(String key)
        {
            if (state == CONNECTED)
                connect(key);
            else if (state == RECORDING)
                login(key);
            else if (state == CHANGE_PASSCODE)
                changePasscode(key);
            else if (state == CHANGE_GREETING)
                changeGreeting(key);
            else if (state == MAILBOX_MENU)
                mailboxMenu(key);
            else if (state == MESSAGE_MENU)
                messageMenu(key);
        }

        /**
         * A Connection object.
         */
        public class Connection
        {

            /**
             * Construct a Connection object.
             * @param s a MailSystem object
             * @param p a Telephone object
             */
            public Connection(MailSystem s, Telephone p)
            {
                system = s;
                phone = p;
                resetConnection();
            }

            /**
             * Respond to the user’s pressing a key on the phone touchpad
             * @param key the key pressed by the user
             */
            public void dial(String key)
            {
                if (state == CONNECTED)
                    connect(key);
                else if (state == RECORDING)
                    login(key);
                else if (state == CHANGE_PASSCODE)
                    changePasscode(key);
                else if (state == CHANGE_GREETING)
                    changeGreeting(key);
                else if (state == MAILBOX_MENU)
                    mailboxMenu(key);
                else if (state == MESSAGE_MENU)
                    messageMenu(key);
            }
        }
    }
}
```
042:     @param voice voice spoken by the user
043:  */
044:  public void record(String voice)
045:  {
046:      //
047:      if (state == RECORDING | state == CHANGE_GREETING)
048:      {
049:          currentRecording += voice;
050:          accumulatedKeys += key;
051:          phone.speak(MESSAGE_MENU_TEXT);
052:          if (currentMailbox != null)
053:          {
054:              currentMailbox.setPasscode(accumulatedKeys);
055:              currentMailbox.addMessage(new Message(currentRecording));
056:              currentMailbox.addMessage(new Message(MESSAGE_MENU_TEXT));
057:              phone.speak(MESSAGE_MENU_TEXT);
058:          }
059:      }
060:  }
061:  }  // Reset the connection to the initial state and prompt
062: */
063: private void resetConnection()
064: {
065:      currentRecording = "";
066:      accumulatedKeys = "";
067:      state = CONNECTED;
068:      phone.speak(INITIAL_PROMPT);
069:  }
070: */
071: private void hangup()
072: */
073: */ Try to connect the user with the specified mailbox.
074: 
075: @param key the phone key pressed by the user
076: */
077: private void connect(String key)
078: {
079:     if (key.equals("1"))
080:     {
081:         currentMailbox = system.findMailbox(accumulatedKeys);
082:         if (currentMailbox != null)
083:         {
084:             phone.speak("Record your greeting, then press the # key");
085:             currentMailbox = system.findMailbox(accumulatedKeys);
086:             currentRecording = "";
087:             accumulatedKeys = "";
088:             state = MAILBOX_MENU;
089:             phone.speak(MESSAGE_MENU_TEXT);
090:             if (key.equals("2"))
091:             {
092:                 accumulatedKeys += key;
093:                 /** Try to log in the user.
094:                  @param key the phone key pressed by the user
095:                  */
096:                  if (key.equals("1"))
097:                  {
098:                      if (currentMailbox.checkPasscode(accumulatedKeys))
099:                      {
100:                          state = MAILBOX_MENU;
101:                          phone.speak(MESSAGE_MENU_TEXT);
102:                      }
103:                  }
104:                  else
105:                  {
106:                      phone.speak("Incorrect passcode. Try again!");
107:                      accumulatedKeys = "";
108:                  }
109:              }
110:          }
111:      }
112:  }
113:  }  // Change passcode.
114: */
115: private void changePasscode(String key)
116: {
117:     if (key.equals("1"))
118:     {
119:         currentMailbox.setPasscode(accumulatedKeys);
120:         phone.speak("Record your greeting, then press the # key");
121:         currentMailbox = system.findMailbox(accumulatedKeys);
122:         if (currentMailbox != null)
123:         {
124:             state = MAILBOX_MENU;
125:             phone.speak(MESSAGE_MENU_TEXT);
126:             accumulatedKeys = "";
127:             if (key.equals("1"))
128:             {
129:                 accumulatedKeys += key;
130:                 phone.speak(MESSAGE_MENU_TEXT);
131:             }
132:         }  // Change greeting.
133:         if (key.equals("1"))
134:         {
135:             state = MAILBOX_MENU;
136:             phone.speak("Record your greeting, then press the # key");
137:             if (key.equals("1"))
138:             {
139:                 currentMailbox.setGreeting(currentRecording);
140:                 currentRecording = "";
141:                 state = MAILBOX_MENU;
142:                 phone.speak(MESSAGE_MENU_TEXT);
143:             }
144:         }
145:     }
146:  }  // Respond to the user's selection from mailbox menu.
147:  }  // Respond to the user's selection from mailbox menu.
148:  }  // Respond to the user's selection from mailbox menu.
149:  }  // Respond to the user's selection from mailbox menu.
150:  }  // Respond to the user's selection from mailbox menu.
151:  }  // Respond to the user's selection from mailbox menu.
152:  
153:  }  // Respond to the user's selection from mailbox menu.
154:  }  // Respond to the user's selection from mailbox menu.
155:  }  // Respond to the user's selection from mailbox menu.
156:  }  // Respond to the user's selection from mailbox menu.
157:  }  // Respond to the user's selection from mailbox menu.
158:  
159:  }  // Respond to the user's selection from mailbox menu.
160:  }  // Respond to the user's selection from mailbox menu.
161:  }  // Respond to the user's selection from mailbox menu.
162:  
163:  }  // Respond to the user's selection from mailbox menu.
164:  
165:  }  // Respond to the user's selection from mailbox menu.
166:  }  // Respond to the user's selection from mailbox menu.
167:  }  // Respond to the user's selection from mailbox menu.
168:  
169:  }  // Respond to the user's selection from mailbox menu.
170:  }  // Respond to the user's selection from mailbox menu.
171:  }  // Respond to the user's selection from mailbox menu.
172:  
173:  }  // Respond to the user's selection from mailbox menu.
174:  }  // Respond to the user's selection from mailbox menu.
175:  }  // Respond to the user's selection from mailbox menu.
176:  
177:  }  // Respond to the user's selection from mailbox menu.
178:  
179:  }  // Respond to the user's selection from mailbox menu.
180:  
181:  }  // Respond to the user's selection from mailbox menu.
182:  
183:  }  // Respond to the user's selection from mailbox menu.
184:  
185:  }  // Respond to the user's selection from mailbox menu.
186:  
187:  }  // Respond to the user's selection from mailbox menu.
188:  
189:  }  // Respond to the user's selection from mailbox menu.
190:  
191:  }  // Respond to the user's selection from mailbox menu.
192:  
193:  }  // Respond to the user's selection from mailbox menu.
194:  
195:  }  // Respond to the user's selection from mailbox menu.
196:  
197:  }  // Respond to the user's selection from mailbox menu.
198:  
199:  }  // Respond to the user's selection from mailbox menu.
200:  
201:  }  // Respond to the user's selection from mailbox menu.
202:  
203:  }  // Respond to the user's selection from mailbox menu.
204:  
205:  }  // Respond to the user's selection from mailbox menu.
public class MailSystem {
    private int state;
    private static final int DISCONNECTED = 0;
    private static final int CONNECTED = 1;
    private static final int RECORDING = 2;
    private static final int MAILBOX_MENU = 3;
    private static final int MESSAGE_MENU = 4;
    private static final int CHANGE_PASSCODE = 5;
    private static final int CHANGE_GREETING = 6;
    private final String INITIAL_PROMPT = 
        "Enter 1 to listen to your messages\n" + 
        "Enter 2 to change your passcode\n" + 
        "Enter 3 to save the current message\n" + 
        "Enter 3 to delete the current message\n" + 
        "Enter 4 to return to the main menu\n";
    private BufferedReader reader;
    private ArrayList mailboxes;
    public MailSystem(int mailboxCount) {
        mailboxes = new ArrayList<>();
        // Initialize mail boxes.
        for (int i = 0; i < mailboxCount; i++) {
            String passcode = "" + (i + 1);
            String greeting = "You have reached mailbox " + (i + 1) + ". \nPlease leave a message now.;
            mailboxes.add(new MailBox(passcode, greeting));
        }
    }
    public MailBox findMailBox(String ext) {
        int i = Integer.parseInt(ext);
        if (1 <= i && i <= mailboxes.size())
            return MailBox(mailboxes.get(i - 1));
        else return null;
    }
    public MailBox getMailBox(String input) {
        boolean more = true;
        while (more) {
            String input = reader.readLine();
            if (input.equalsIgnoreCase("R"))
                c.hangup();
            else if (input.equalsIgnoreCase("Q"))
                more = false;
            else if (input.length() == 1)
                if ("1234567890#".indexOf(input) >= 0) 
                    c.dial(input);
            else if (input.length() > 2)
                c.record(input);
        }
        return null;
    }
    public void speak(String output) {
        System.out.println(output);
    }
    public Telephone(BufferedReader aReader) {
        this.reader = aReader;
    }
    public void run(Connection c) throws IOException {
        boolean more = true;
        while (more) {
            String input = reader.readLine();
            if (input.equalsIgnoreCase("R"))
                c.hangup();
            else if (input.equalsIgnoreCase("Q"))
                more = false;
            else if (input.length() == 1)
                if ("1234567890#".indexOf(input) >= 0) 
                    c.dial(input);
            else if (input.length() > 2)
                c.record(input);
        }
    }
    private Telephone phone;
    private int state;
    private static final int ...
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

/**
 * This program tests the mail system. A single phone communicates with the program through System.in/System.out.
 */
public class MailSystemTest {
    public static void main(String[] args) throws IOException {
        MailSystem system = new MailSystem(MAILBOX_COUNT);
        BufferedReader console = new BufferedReader(new InputStreamReader(System.in));
        Telephone p = new Telephone(console);
        Connection c = new Connection(system, p);
        p.run(c);
    }

    private static final int MAILBOX_COUNT = 20;
}