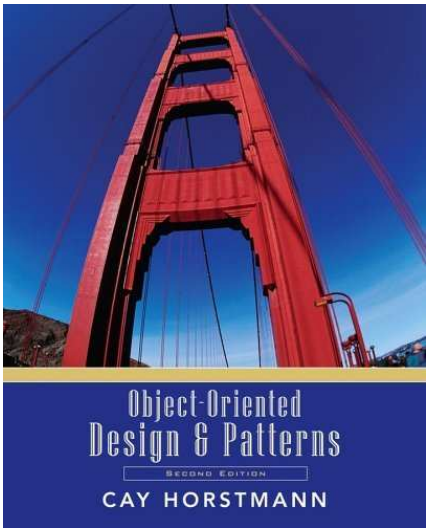
Object-Oriented Design & Patterns

Cay S. Horstmann

Chapter 2

The Object-Oriented Design Process



Chapter Topics

- From Problem to Code
- The Object and Class Concepts
- Identifying Classes
- Identifying Responsibilities
- Relationships Between Classes
- 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

Functional Specification

- 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

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

- 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 problem 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

- Dependency ("uses")
- Aggregation ("has")
- Inheritance ("is")

Dependency Relationship

- `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`

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:

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

- 1 : *n* relationship:

```
public class MessageQueue{   . . .   private ArrayList<Message> elements;}
```


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.

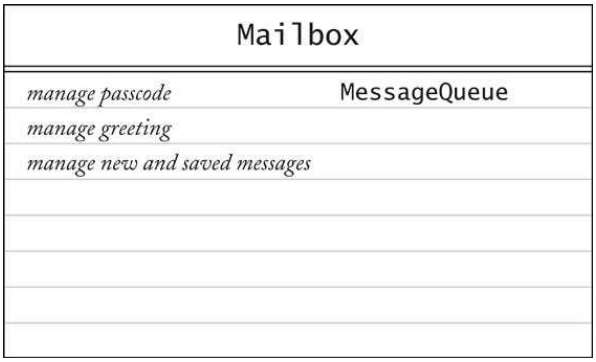
Variation #2

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



CRC Cards

- 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

MailSystem	
manage mailboxes	Mailbox

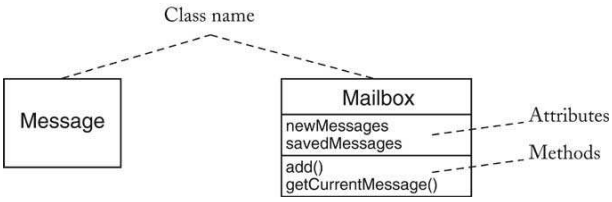
UML Diagrams

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

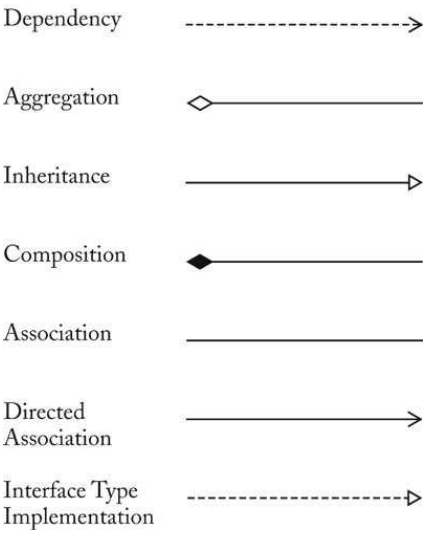
Class Diagrams

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

Class Diagrams

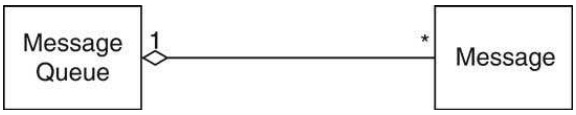


Class Relationships



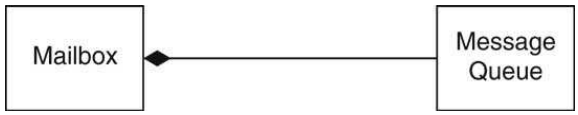
Multiplicities

- any number (0 or more): `*`
- one or more: `1..*`
- zero or one: `0..1`
- exactly one: `1`



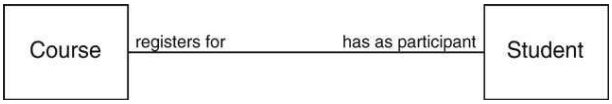
Composition

- Special form of aggregation
- Contained objects don't exist outside container
- Example: message queues permanently contained in mail box



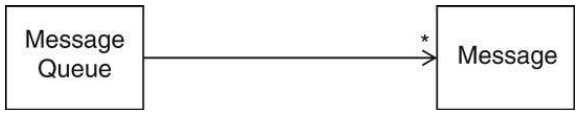
Association

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



Association

- Some associations are bidirectional
  Can navigate from either class to the other
- Example: Course has set of students, student has set of courses
- Some associations are directed
  Navigation is unidirectional
- Example: Message doesn't know about message queue containing it



Interface Types

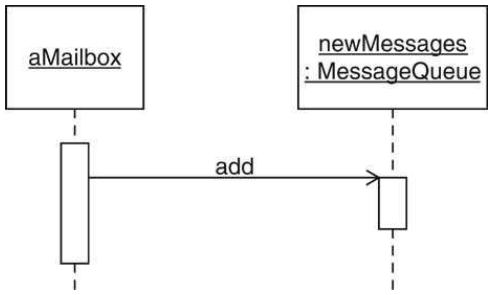
- Interface type describes a set of methods
- No implementation, no state
- Class implements interface if it implements its methods
- In UML, use stereotype «interface»

Tips

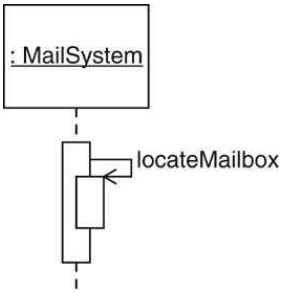
- Use UML to inform, not to impress
- Don’t draw a single monster diagram
- Each diagram must have a specific purpose
- Omit inessential details

Sequence Diagrams

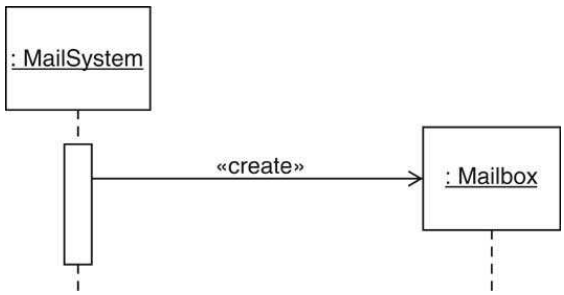
- Each diagram shows dynamics of scenario
- Object diagram: class name underlined



Self call

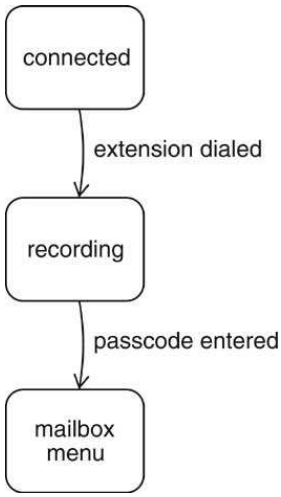


Object Construction



State Diagram

- Use for classes whose objects have interesting states



Design Documentation

- Recommendation: Use Javadoc comments
- Leave methods blank

```
/**   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. 5)

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 messagePress 2 to delete the current messagePress 3 to save the current messagePress 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

Use Case: Change the 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

Use Case: Change the 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

Mailbox	
keep new and saved messages	`MessageQueue`

Initial CRC Cards: MessageQueue

MessageQueue	
add and remove messages in FIFO order	

Initial CRC Cards: MailSystem

MailSystem	
manage mailboxes	`Mailbox`

Telephone

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

Telephone

Telephone
take user input from touchpad,
microphone, hangup
speak output

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

Connection

Connection	
get input from telephone	Telephone
carry out user commands	MailSystem
keep track of state	

Analyze Use Case: Leave a message

1. User dials extension. `Telephone` sends number to `Connection`
   (Add collaborator `Connection` to `Telephone`)
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 message. `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	
take user input from touchpad, microphone, hangup	Connection
speak output	

Result of Use Case Analysis

Connection	
get input from telephone	Telephone
carry out user commands	MailSystem
keep track of state	Mailbox
record voice input	Message

Result of Use Case Analysis

Mailbox	
keep new and saved messages	MessageQueue
manage greeting	

Result of Use Case Analysis

Message
manage message contents

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`).
   `Connection` asks `Telephone` to speak message
8. `Connection` asks `Telephone` to speak menu
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

Result of Use Case Analysis

Mailbox	
keep new and saved messages	`MessageQueue`
manage greeting	
manage passcode	
retrieve, save, delete messages	

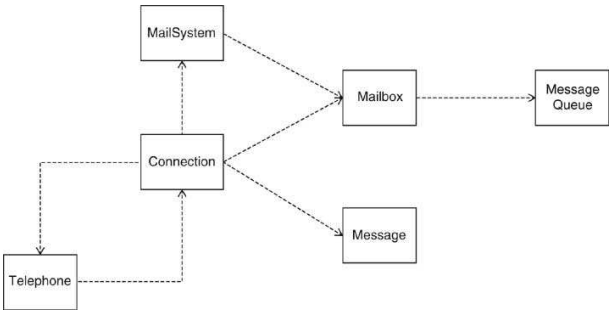
CRC Summary

- 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`
- `Message` doesn't depends on `Mailbox`
- `Connection` depends on `Telephone, MailSystem, Message, Mailbox`
- `Telephone` depends on `Connection`

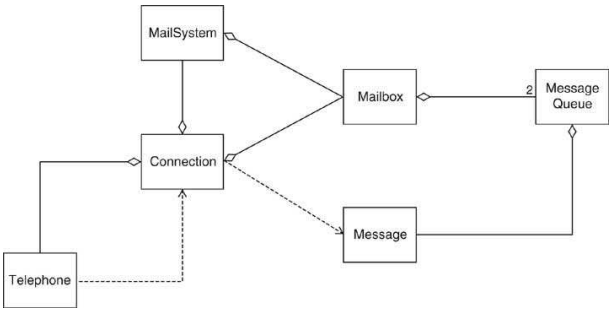
Dependency Relationships



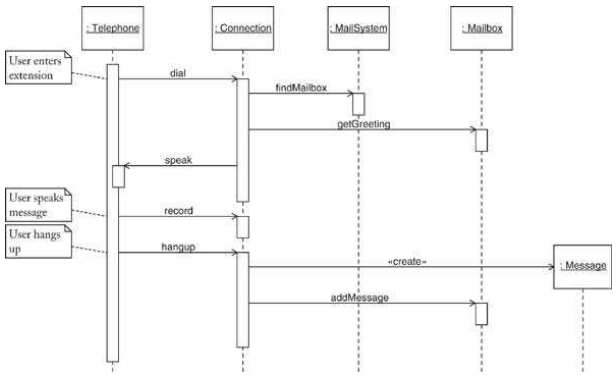
Aggregation Relationships

- 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 mailsystem and a telephone

UML Class Diagram for Voice Mail System



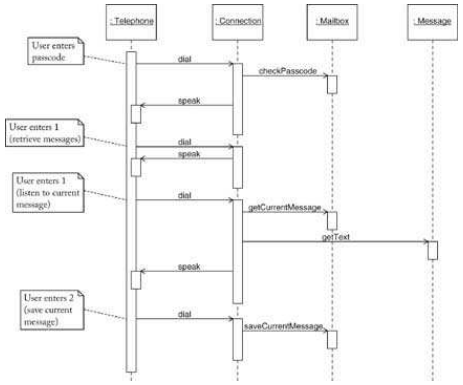
Sequence Diagram for Use Case: Leave a message



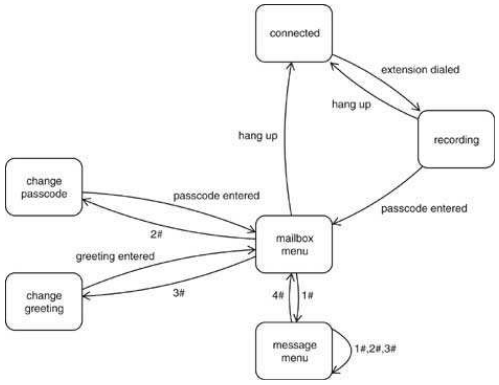
Interpreting a Sequence Diagram

- 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/MailSystemTester.java


```
01: /**
02:    A message left by the caller.
03: */
04: public class Message
05: {
06:    /**
07:       Construct a Message object.
08:       @param messageText the message text
09:    */
10:    public Message(String messageText)
11:    {
12:       text = messageText;
13:    }
14:
15:    /**
16:       Get the message text.
17:       @return message text
18:    */
19:    public String getText()
20:    {
21:       return text;
22:    }
23:
24:    private String text;
25: }
```

```
01: import java.util.ArrayList;
02:
03: /**
04:    A first-in, first-out collection of messages. This
05:    implementation is not very efficient. We will consider
06:    a more efficient implementation in chapter 3.
07: */
08: public class MessageQueue
09: {
10:    /**
11:       Constructs an empty message queue.
12:    */
13:    public MessageQueue()
14:    {
15:       queue = new ArrayList<Message>();
16:    }
17:
18:    /**
19:       Remove message at head.
20:       @return message that has been removed from the queue
21:    */
22:    public Message remove()
23:    {
24:       return queue.remove(0);
25:    }
26:
27:    /**
28:       Append message at tail.
29:       @param newMessage the message to be appended
30:    */
31:    public void add(Message newMessage)
32:    {
33:       queue.add(newMessage);
34:    }
35:
36:    /**
37:       Get the total number of messages in the queue.
38:       @return the total number of messages in the queue
39:    */
40:    public int size()
41:    {
42:       return queue.size();
43:    }
44:
```

```
45:    /**
46:       Get message at head.
47:       @return message that is at the head of the queue, or null
48:       if the queue is empty
49:    */
50:    public Message peek()
51:    {
52:       if (queue.size() == 0) return null;
53:       else return queue.get(0);
54:    }
55:
56:    private ArrayList<Message> queue;
57: }
```

```
001: /**
002:    A mailbox contains messages that can be listed, kept or discarde
003: */
004: public class Mailbox
005: {
006:    /**
007:       Creates Mailbox object.
008:       @param aPasscode passcode number
009:       @param aGreeting greeting string
010:    */
011:    public Mailbox(String aPasscode, String aGreeting)
012:    {
013:       passcode = aPasscode;
014:       greeting = aGreeting;
015:       newMessages = new MessageQueue();
016:       keptMessages = new MessageQueue();
017:    }
018:
019:    /**
020:       Check if the passcode is correct.
021:       @param aPasscode a passcode to check
022:       @return true if the supplied passcode matches the mailbox pas
023:    */
024:    public boolean checkPasscode(String aPasscode)
025:    {
026:       return aPasscode.equals(passcode);
027:    }
028:
029:    /**
030:       Add a message to the mailbox.
031:       @param aMessage the message to be added
032:    */
033:    public void addMessage(Message aMessage)
034:    {
035:       newMessages.add(aMessage);
036:    }
037:
038:    /**
039:       Get the current message.
040:       @return the current message
041:    */
042:    public Message getCurrentMessage()
043:    {
044:       if (newMessages.size() > 0)
```

```
045:          return newMessages.peek();
046:       else if (keptMessages.size() > 0)
047:          return keptMessages.peek();
048:       else
049:          return null;
050:    }
051:
052:    /**
053:       Remove the current message from the mailbox.
054:       @return the message that has just been removed
055:    */
056:    public Message removeCurrentMessage()
057:    {
058:       if (newMessages.size() > 0)
059:          return newMessages.remove();
060:       else if (keptMessages.size() > 0)
061:          return keptMessages.remove();
062:       else
063:          return null;
064:    }
065:
066:    /**
067:       Save the current message
068:    */
069:    public void saveCurrentMessage()
070:    {
071:       Message m = removeCurrentMessage();
072:       if (m != null)
073:          keptMessages.add(m);
074:    }
075:
076:    /**
077:       Change mailbox's greeting.
078:       @param newGreeting the new greeting string
079:    */
080:    public void setGreeting(String newGreeting)
081:    {
082:       greeting = newGreeting;
083:    }
084:
085:    /**
086:       Change mailbox's passcode.
087:       @param newPasscode the new passcode
088:    */
```

```
089:    public void setPasscode(String newPasscode)
090:    {
091:       passcode = newPasscode;
092:    }
093:
094:    /**
095:       Get the mailbox's greeting.
096:       @return the greeting
097:    */
098:    public String getGreeting()
099:    {
100:       return greeting;
101:    }
102:
103:    private MessageQueue newMessages;
104:    private MessageQueue keptMessages;
105:    private String greeting;
106:    private String passcode;
107: }
```

```
001: /**
002:    Connects a phone to the mail system. The purpose of this
003:    class is to keep track of the state of a connection, since
004:    the phone itself is just a source of individual key presses.
005: */
006: public class Connection
007: {
008:    /**
009:       Construct a Connection object.
010:       @param s a MailSystem object
011:       @param p a Telephone object
012:    */
013:    public Connection(MailSystem s, Telephone p)
014:    {
015:       system = s;
016:       phone = p;
017:       resetConnection();
018:    }
019:
020:    /**
021:       Respond to the user's pressing a key on the phone touchpad
022:       @param key the phone key pressed by the user
023:    */
024:    public void dial(String key)
025:    {
026:       if (state == CONNECTED)
027:          connect(key);
028:       else if (state == RECORDING)
029:          login(key);
030:       else if (state == CHANGE_PASSCODE)
031:          changePasscode(key);
032:       else if (state == CHANGE_GREETING)
033:          changeGreeting(key);
034:       else if (state == MAILBOX_MENU)
035:          mailboxMenu(key);
036:       else if (state == MESSAGE_MENU)
037:          messageMenu(key);
038:    }
039:
040:    /**
041:       Record voice.
042:       @param voice voice spoken by the user
043:    */
044:    public void record(String voice)
```

```
045:    {
046:       if (state == RECORDING || state == CHANGE_GREETING)
047:          currentRecording += voice;
048:    }
049:
050:    /**
051:       The user hangs up the phone.
052:     */
053:    public void hangup()
054:    {
055:       if (state == RECORDING)
056:          currentMailbox.addMessage(new Message(currentRecording));
057:       resetConnection();
058:    }
059:
060:    /**
061:       Reset the connection to the initial state and prompt
062:       for mailbox number
063:     */
064:    private void resetConnection()
065:    {
066:       currentRecording = "";
067:       accumulatedKeys = "";
068:       state = CONNECTED;
069:       phone.speak(INITIAL_PROMPT);
070:    }
071:
072:    /**
073:       Try to connect the user with the specified mailbox.
074:       @param key the phone key pressed by the user
075:     */
076:    private void connect(String key)
077:    {
078:       if (key.equals("#"))
079:       {
080:          currentMailbox = system.findMailbox(accumulatedKeys);
081:          if (currentMailbox != null)
082:          {
083:             state = RECORDING;
084:             phone.speak(currentMailbox.getGreeting());
085:          }
086:          else
087:             phone.speak("Incorrect mailbox number. Try again!");
088:          accumulatedKeys = "";
```

```
089:       }
090:       else
091:          accumulatedKeys += key;
092:    }
093:
094:    /**
095:       Try to log in the user.
096:       @param key the phone key pressed by the user
097:     */
098:    private void login(String key)
099:    {
100:       if (key.equals("#"))
101:       {
102:          if (currentMailbox.checkPasscode(accumulatedKeys))
103:          {
104:             state = MAILBOX_MENU;
105:             phone.speak(MAILBOX_MENU_TEXT);
106:          }
107:          else
108:             phone.speak("Incorrect passcode. Try again!");
109:          accumulatedKeys = "";
110:       }
111:       else
112:          accumulatedKeys += key;
113:    }
114:
115:    /**
116:       Change passcode.
117:       @param key the phone key pressed by the user
118:     */
119:    private void changePasscode(String key)
120:    {
121:       if (key.equals("#"))
122:       {
123:          currentMailbox.setPasscode(accumulatedKeys);
124:          state = MAILBOX_MENU;
125:          phone.speak(MAILBOX_MENU_TEXT);
126:          accumulatedKeys = "";
127:       }
128:       else
129:          accumulatedKeys += key;
130:    }
131:
132:    /**
```

```
133:       Change greeting.
134:       @param key the phone key pressed by the user
135:     */
136:    private void changeGreeting(String key)
137:    {
138:       if (key.equals("#"))
139:       {
140:          currentMailbox.setGreeting(currentRecording);
141:          currentRecording = "";
142:          state = MAILBOX_MENU;
143:          phone.speak(MAILBOX_MENU_TEXT);
144:       }
145:    }
146:
147:    /**
148:       Respond to the user's selection from mailbox menu.
149:       @param key the phone key pressed by the user
150:     */
151:    private void mailboxMenu(String key)
152:    {
153:       if (key.equals("1"))
154:       {
155:          state = MESSAGE_MENU;
156:          phone.speak(MESSAGE_MENU_TEXT);
157:       }
158:       else if (key.equals("2"))
159:       {
160:          state = CHANGE_PASSCODE;
161:          phone.speak("Enter new passcode followed by the # key");
162:       }
163:       else if (key.equals("3"))
164:       {
165:          state = CHANGE_GREETING;
166:          phone.speak("Record your greeting, then press the # key");
167:       }
168:    }
169:
170:    /**
171:       Respond to the user's selection from message menu.
172:       @param key the phone key pressed by the user
173:     */
174:    private void messageMenu(String key)
175:    {
176:       if (key.equals("1"))
```

```
177:       {
178:          String output = "";
179:          Message m = currentMailbox.getCurrentMessage();
180:          if (m == null) output += "No messages." + "\n";
181:          else output += m.getText() + "\n";
182:          output += MESSAGE_MENU_TEXT;
183:          phone.speak(output);
184:       }
185:       else if (key.equals("2"))
186:       {
187:          currentMailbox.saveCurrentMessage();
188:          phone.speak(MESSAGE_MENU_TEXT);
189:       }
190:       else if (key.equals("3"))
191:       {
192:          currentMailbox.removeCurrentMessage();
193:          phone.speak(MESSAGE_MENU_TEXT);
194:       }
195:       else if (key.equals("4"))
196:       {
197:          state = MAILBOX_MENU;
198:          phone.speak(MAILBOX_MENU_TEXT);
199:       }
200:    }
201:
202:    private MailSystem system;
203:    private Mailbox currentMailbox;
204:    private String currentRecording;
205:    private String accumulatedKeys;
206:    private Telephone phone;
207:    private int state;
208:
209:    private static final int DISCONNECTED = 0;
210:    private static final int CONNECTED = 1;
211:    private static final int RECORDING = 2;
212:    private static final int MAILBOX_MENU = 3;
213:    private static final int MESSAGE_MENU = 4;
214:    private static final int CHANGE_PASSCODE = 5;
215:    private static final int CHANGE_GREETING = 6;
216:
217:    private static final String INITIAL_PROMPT =
218:          "Enter mailbox number followed by #";
219:    private static final String MAILBOX_MENU_TEXT =
220:          "Enter 1 to listen to your messages\n"
```

```
        + "Enter 2 to change your passcode\n"
        + "Enter 3 to change your greeting";
   private static final String MESSAGE_MENU_TEXT =
         "Enter 1 to listen to the current message\n"
        + "Enter 2 to save the current message\n"
        + "Enter 3 to delete the current message\n"
        + "Enter 4 to return to the main menu";
}
```

```
import java.util.ArrayList;

/**
   A system of voice mail boxes.
*/
public class MailSystem
{
   /**
      Constructs a mail system with a given number of mailboxes
      @param mailboxCount the number of mailboxes
    */
   public MailSystem(int mailboxCount)
   {
      mailboxes = new ArrayList<Mailbox>();

      // 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));
      }
   }

   /**
      Locate a mailbox.
      @param ext the extension number
      @return the mailbox or null if not found
    */
   public Mailbox findMailbox(String ext)
   {
      int i = Integer.parseInt(ext);
      if (1 <= i && i <= mailboxes.size())
         return  mailboxes.get(i - 1);
      else return null;
   }

   private ArrayList<Mailbox> mailboxes;
}
```

```
import java.util.Scanner;

/**
   A telephone that takes simulated keystrokes and voice input
   from the user and simulates spoken text.
*/
public class Telephone
{
   /**
      Construct phone object.
      @param aScanner that reads text from a character-input stream
    */
   public Telephone(Scanner aScanner)
   {
      scanner = aScanner;
   }

   /**
      Speak a message to System.out.
      @param output the text that will be "spoken"
    */
   public void speak(String output)
   {
      System.out.println(output);
   }

   /**
      Loops reading user input and passes the input to the
      Connection object's methods dial, record or hangup.
      @param c the connection that connects this phone to the
      voice mail system
    */
   public void run(Connection c)
   {
      boolean more = true;
      while (more)
      {
         String input = scanner.nextLine();
         if (input == null) return;
         if (input.equalsIgnoreCase("H"))
            c.hangup();
         else if (input.equalsIgnoreCase("Q"))
            more = false;
         else if (input.length() == 1
```

```
            && "1234567890#".indexOf(input) >= 0)
            c.dial(input);
         else
            c.record(input);
      }
   }

   private Scanner scanner;
}
```

```
01: import java.util.Scanner;
02:
03: /**
04:    This program tests the mail system. A single phone
05:    communicates with the program through System.in/System.out.
06: */
07: public class MailSystemTester
08: {
09:    public static void main(String[] args)
10:    {
11:       MailSystem system = new MailSystem(MAILBOX_COUNT);
12:       Scanner console = new Scanner(System.in);
13:       Telephone p = new Telephone(console);
14:       Connection c = new Connection(system, p);
15:       p.run(c);
16:    }
17:
18:    private static final int MAILBOX_COUNT = 20;
19: }
```