Object-Oriented Design & Patterns

Adapted from: Cay S. Horstmann

By: Fernando A. Rojas Morales

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

Chapter Topics

- CRC Cards
- UML Class Diagrams
- Sequence Diagrams
- State Diagrams
- Using javadoc for Design Documentation
- Case Study: A Voice Mail System

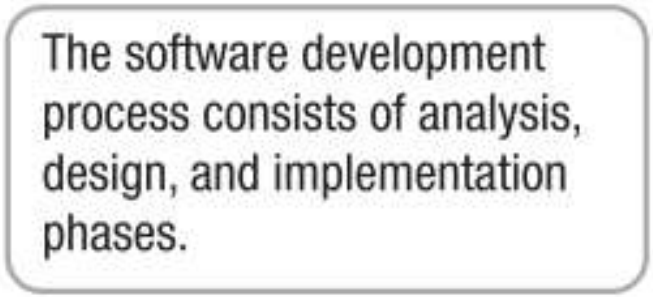
From Problem to Code

Programming tasks originate from the desire to solve a particular problem.
[or to take the best of a technology; say opportunity]
The task may be simple, such as writing a program that generates and formats a report, or complicated, such as writing a word processor.
The end product is a working program.

From Problem to Code

Three Phases:

- Analysis
- Design
- Implementation

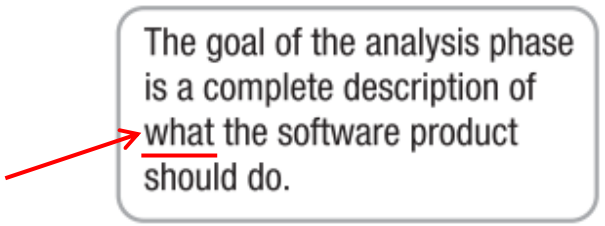


Case Study: Voice Mail System

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



Analysis Phase

Different ways to express functionality:

- Description
- Use/Cases; Scenarios
- Features (FDD)
- User-Stories (XP, SCRUM)

Design Phase

Goals

- Identify classes
- Identify behavior (responsibilities) of classes
- Identify relationships among classes

These are goals, not steps.

The goal of object-oriented design is the identification of classes, their responsibilities, and the relationships among them.

Design Phase

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

The goal of the implementation phase is the programming, testing, and deployment of the software product.

Implementation Phase

- Object-oriented design is particularly suited for prototyping; the OO promise.
- You should not rush the analysis and design phase to get to a working prototype quickly, nor should you hesitate to reopen the previous phases if a prototype yields new insight.
- For small to medium-sized products, a prototype can expand into a complete product. If you follow this evolutionary approach, be sure that the transition from prototype to final product is well managed and that enough time is allocated to fix mistakes and implement newly discovered improvements.

Object and Class Concepts

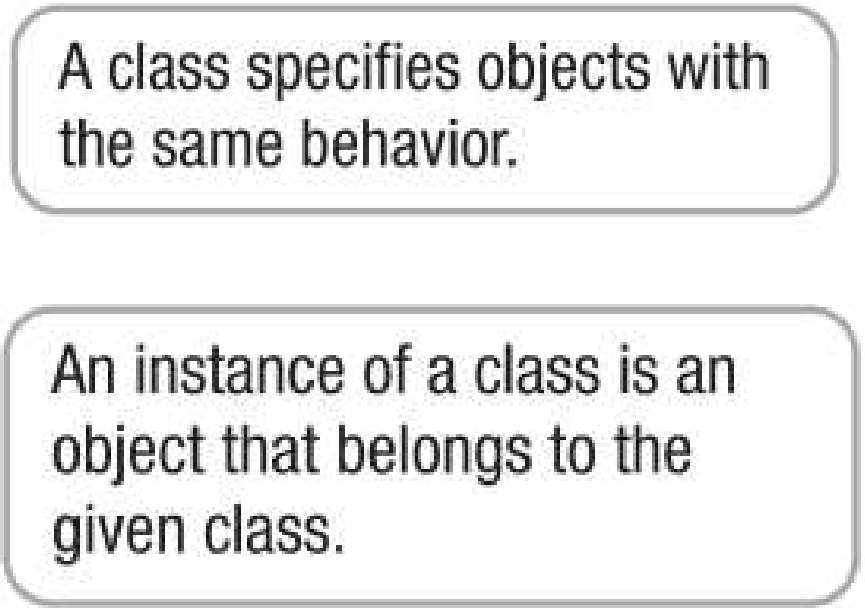
- Object: Three characteristic concepts
  - State
  - Behavior
  - Identity

An object is characterized by its state, behavior, and identity.

- The collection of all information held by an object is the object's state.
- The behavior of an object is defined by the operations that an object supports.
- Each object has its own identity.

Object and Class Concepts

- Class: *Blueprint* for objects with the same behavior and common set of possible states



Case Study: A Voice Mail System

To walk through the basic steps of the object-oriented design process, we will consider the task of writing a program that simulates a telephone voice mail system, similar to the message system that many companies use.

In a voice mail system, a person dials an extension number and, provided the other party does not pick up the telephone, leaves a message. The other party can later retrieve the messages, keep them, or delete them. Real-world systems have a multitude of fancy features: Messages can be forwarded to one or more mailboxes; distribution lists can be defined, retained, and edited; and authorized persons can send broadcast messages to all users.

We will design and implement a program that simulates a voice mail system, without creating a completely realistic working phone system. We will simply represent voice mail by text that is entered through the keyboard. We need to simulate the three distinct input events that occur in a real telephone system: speaking, pushing a button on the telephone touchpad, and hanging up the telephone. We use the following convention for input: An input line consisting of a single character `1 . . . 9` or `#` denotes a pressed button on the telephone touchpad. For example, to dial extension 13, you enter

```
1
3
#
```

An input line consisting of the single letter `H` denotes hanging up the telephone. Any other text denotes voice input.

The first formal step in the process that leads us toward the final product (the voice mail system) is the analysis phase. Its role is to crisply define the behavior of the system. In this example, we will define the behavior through a set of use cases. Note that the use cases by themselves are *not* a full specification of a system. The functional specification also needs to define system limitations, performance, and so on.

Identifying Classes

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

- Mailbox
- Message
- User
- Passcode
- Extension
- Menu

Class names should be nouns in the singular form.

Identifying Classes

Focus on *concepts*, not on 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

Categories of Classes

Sometimes it is helpful to change an operation into an agent class. For example, the "compute page breaks" operation on a document could be turned into a `Paginator` class, which operates on documents. Then the paginator can work on a part of a document while another part is edited on the screen. In this case, the agent class is invented to express parallel execution.

The `Scanner` class is another example. As described in Chapter 1, a `Scanner` is used to scan for numbers and strings in an input stream. Thus, the operation of parsing input is encapsulated in the `Scanner` agent.

Agent classes often end in "er" or "or".

Event and transaction classes are useful to model records of activities that describe what happened in the past or what needs to be done later. An example is a `MouseEvent` class, which remembers when and where the mouse was moved or clicked.

User and role classes are stand-ins for actual users of the program. An `Administrator` class is a representation of the human administrator of the system. A `Reviewer` class in an interactive authoring system models a user whose role is to add critical annotations and recommendations for change. User classes are common in systems that are used by more than one person or where one person needs to perform distinct tasks.

System classes model a subsystem or the overall system being built. Their roles are typically to perform initialization and shutdown and to start the flow of input into the system. For example, we might have a class `MailSystem` to represent the voice mail system in its entirety.

System interface classes model interfaces to the host operating system, the windowing system, a database, and so on. A typical example is the `File` class.

Foundation classes are classes such as `String`, `Date`, or `Rectangle`. They encapsulate basic data types with well-understood properties. At the design stage, you should simply assume that these classes are readily available, just as the fundamental types (integers and floating-point numbers) are.

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

To discover responsibilities, look for verbs in the problem description.

Responsibilities

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

A responsibility must belong to exactly one class.

Responsibilities

When discovering responsibilities, programmers commonly make wrong guesses and assign the responsibility to an inappropriate class. For that reason, it is helpful to have more than one person involved in the design phase. If one person assigns a responsibility to a particular class, another can ask the hard question, "How can an object of this class possibly carry out this responsibility?" The question is hard because we are not yet supposed to get to the nitty-gritty of implementation details. But it is appropriate to consider a "reasonable" implementation, or better, two different possibilities, to demonstrate that the responsibility can be carried out.

TIP When assigning responsibilities, respect the natural *layering of abstraction levels.* At the lowest levels of any system, we have files, keyboard and mouse interfaces, and other system services. At the highest levels there are classes that tie together the software system, such as `MailSystem`. The responsibilities of a class should stay at *one abstraction level.* A class `Mailbox` that represents a mid-level abstraction should not deal with processing keystrokes, a low-level responsibility, nor should it be concerned with the initialization of the system, a high-level responsibility.

Class Relationships

- Dependency ("uses")
- Aggregation ("has")
- Inheritance (“behaves-as") ; *not “is-a”*

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

Dependency is an asymmetric relationship

A class depends on another class if it manipulates objects of the other class.

Coupling

- Minimize dependency: reduce *coupling*
- Example: Replace
  void print() // prints to System.out
  with
  String getText() // can print anywhere
- Removes dependence on System, PrintStream

TIP Minimize the number of dependencies between classes. When classes depend on each other, changes in one of them can force changes in the others.

Aggregation

- Object of a class contains objects of another class
- Aggregation is a special case of dependency.
- Described as the "has-a" relationship.
- Example: *MessageQueue* aggregates *Messages*
- Example: *Mailbox* aggregates *MessageQueue*
- Implemented through instance fields

A class aggregates another if its objects contain objects of the other class.

Aggregation

- Not all instance fields of a class correspond to aggregation.
- If an object contains a field of a very simple type such as a number, string, or date, it is considered merely an attribute, not aggregation.

The distinction between aggregation and attributes depends on the context of your design. You'll need to make a judgment whether a particular class is "very simple", giving rise to attributes, or whether you should describe an aggregation relationship.

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 [LSP]

- 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

A class inherits from another if it incorporates the behavior of the other class.

Inheritance

- Subclass inherits from superclass
- Example: *ForwardedMessage* inherits from *Message*
- Example: *Greeting does not* inherit from *Message* (Can't store greetings in mailbox)

A class *inherits* from another if all objects of its class are special cases of objects of the other class, capable of exhibiting the same behavior but possibly with additional responsibilities and a richer state.

Use Cases

- Use cases are an analysis technique to describe in a formal way how a computer system should work.
- Each *use case* focuses on specific scenarioS
- 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

A use case lists a sequence of actions that yields a result that is of value to an actor.

Sample Use Case

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

Use Case -- Variations

- Most scenarios that potentially deliver a valuable outcome can also fail for one reason or another.
- A use case should include variations that describe these situations; called alternate paths of the use case.

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.

Sample Use Case -- Variations

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

A CRC card is an index card that describes a class, its high-level responsibilities, and its collaborators.

http://c2.com/doc/oopsla89/paper.html.

CRC Cards

Mailbox	
manage passcode	MessageQueue
manage greeting	
manage new and saved messages	

CRC Cards

- Responsibilities should be *high level* ; most include more than one method
- 1 - 3 responsibilities per card
- Collaborators are for the class, not for each responsibility
- Best filled by playing the CRC-Scenario-game

TIPs

- Beware of the *omnipotent* system class.
- Beware of classes with *magical* powers that have no connection with the real world or computer systems.
- Watch out for unrelated responsibilities.
- Resist the temptation to add responsibilities just because they *can* be done.
- A class with *no responsibilities* surely is not useful.

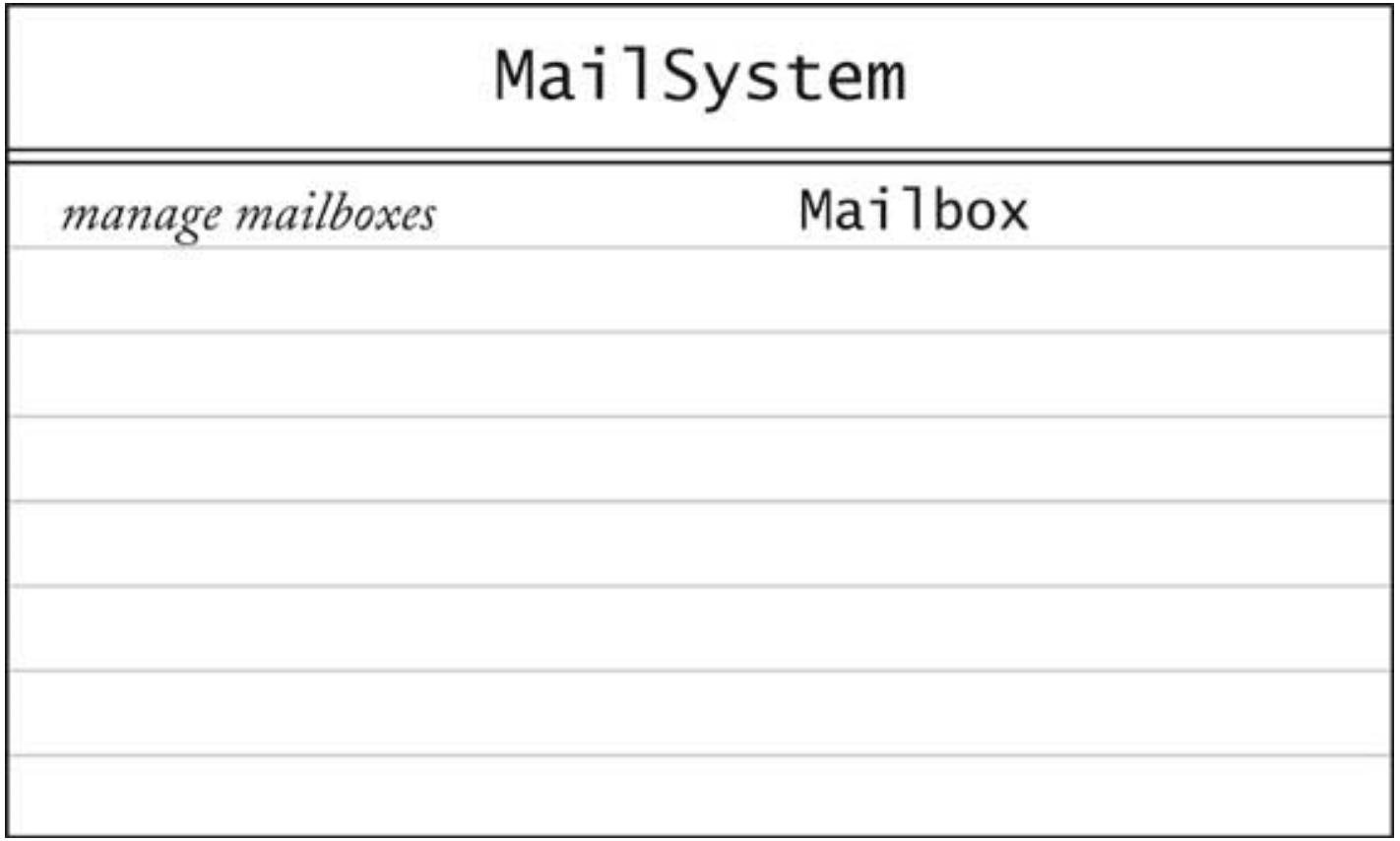
Walkthroughs

- CRC cards are quite intuitive for “walking through” use cases.
- The walkthroughs with CRC cards are particularly suited for group discussion.
- Uses cases or scenarios are walked through to find responsibilities.
- Play the CRC-Scenario Game.

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



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

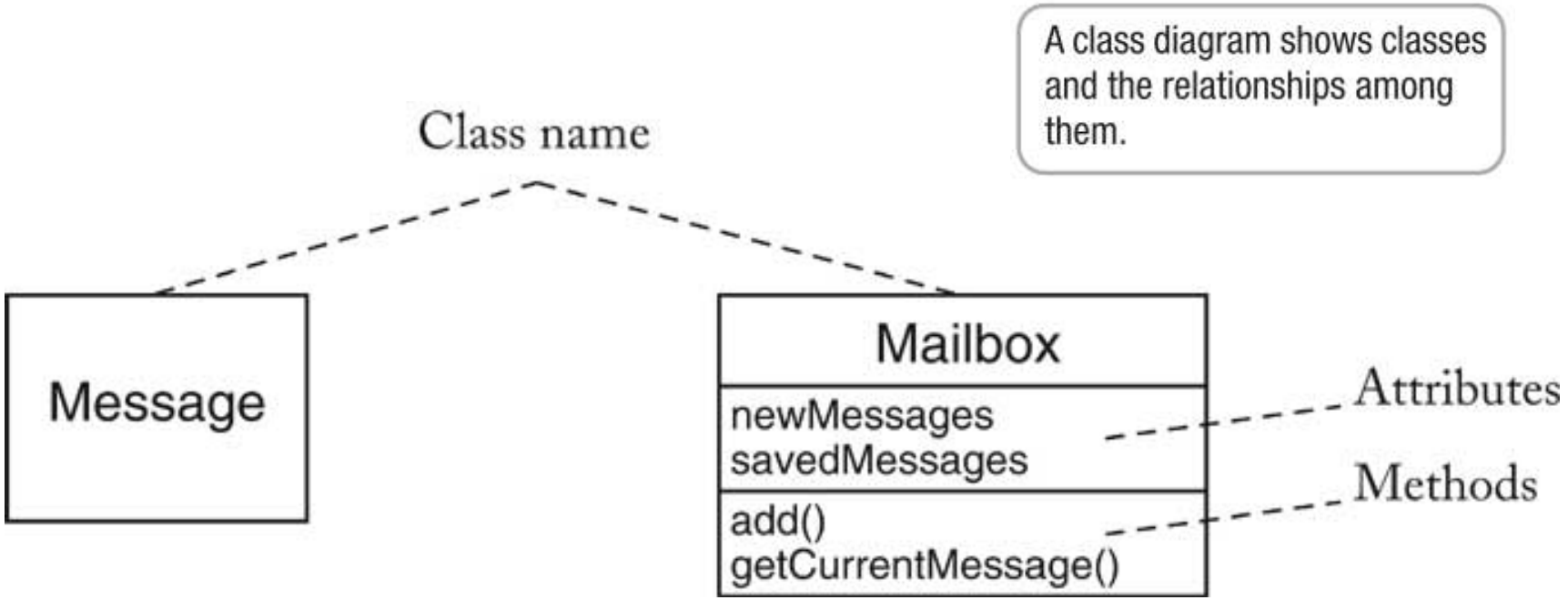
A UML diagram illustrates an aspect of an object-oriented design, using a standardized notation.

Class Diagrams

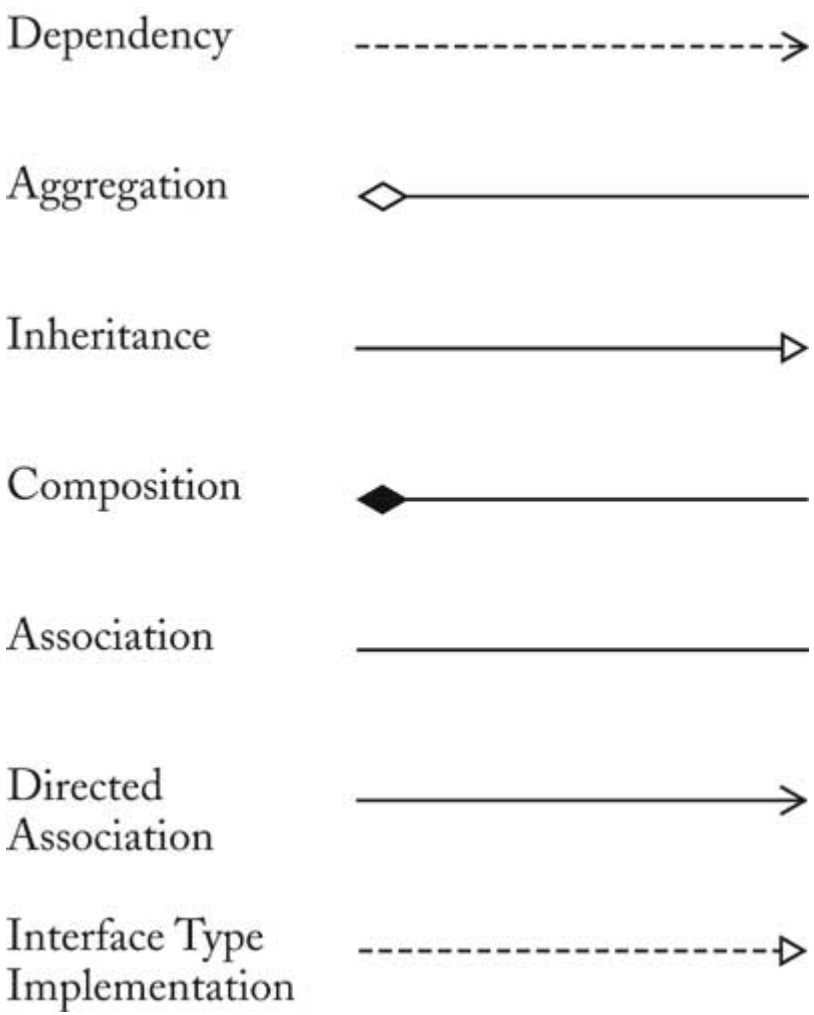
- Rectangle with class name
- Optional compartments
  - Attributes
  - Methods
- Include only key attributes and methods

TIP If you have lots of attributes, check whether you can group some of them into classes.

Class Diagrams



Class Relationships

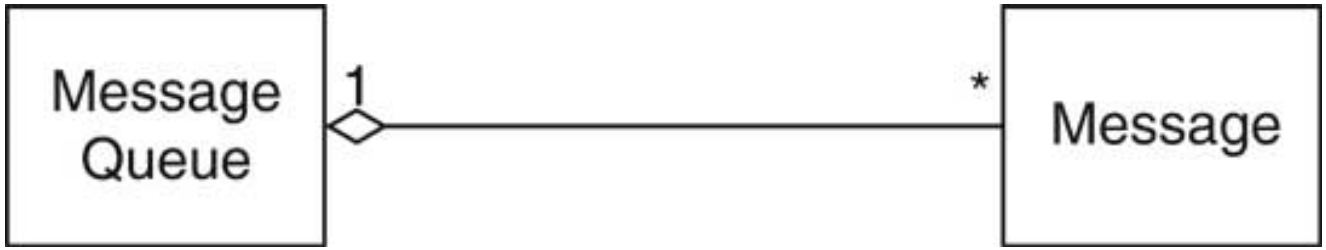


Multiplicities

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

Aggregation

- Aggregation “wins” over dependency
- If a class aggregates another, it uses it



Dependency

- The most important relationship to control is the dependency or “uses” relationship.
- Too many dependencies make it difficult to evolve a design over time.

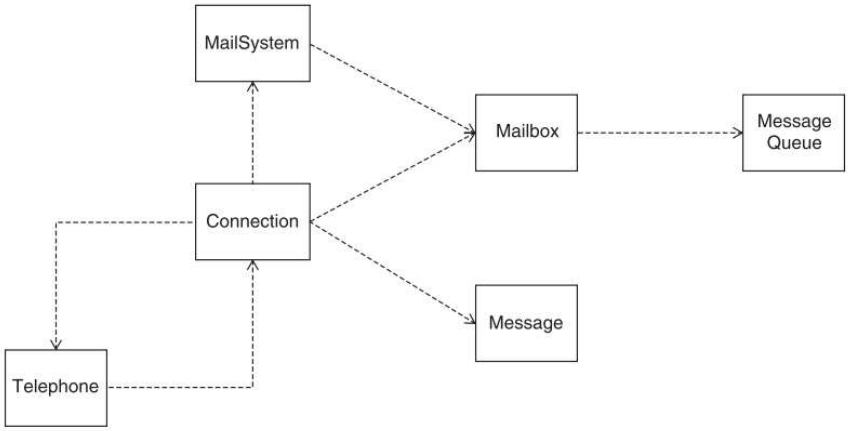
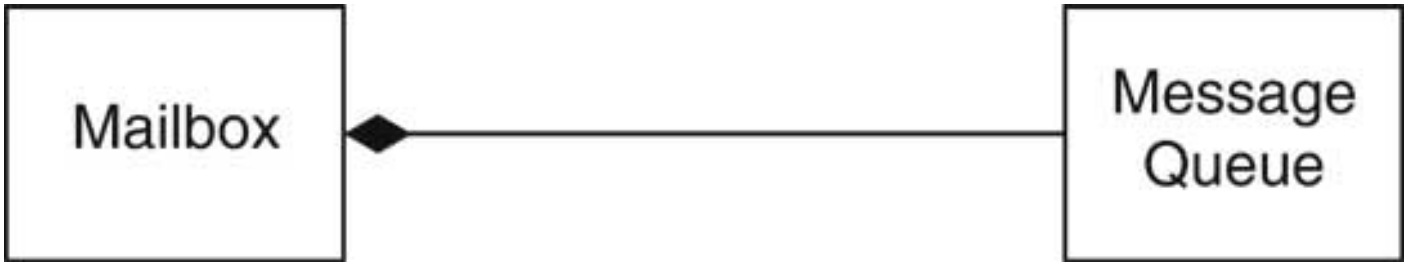


Figure 15

The Voice Mail System Dependencies from the CRC Cards

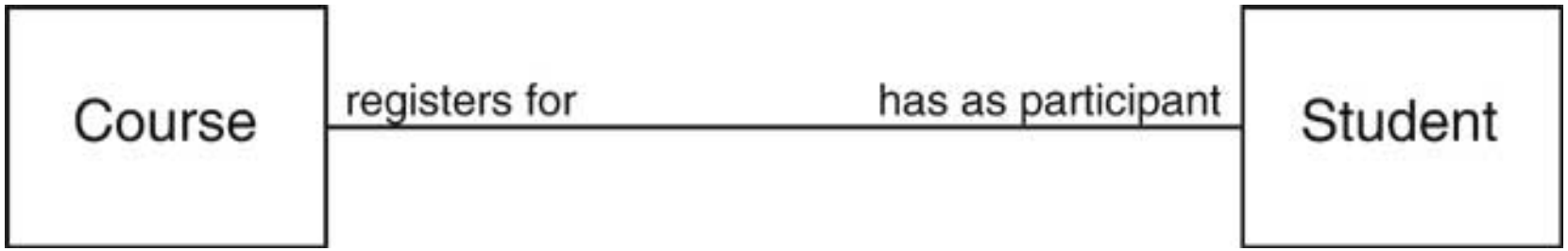
Composition

- 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



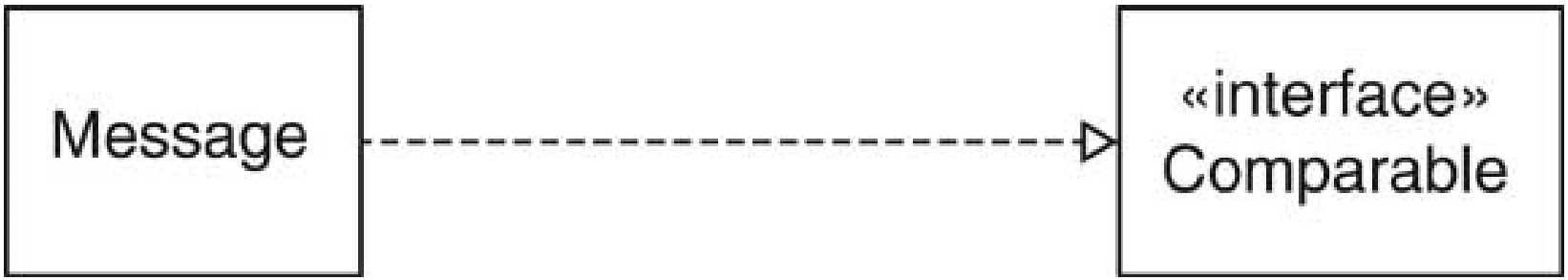
Association

- 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

Association

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

- Sequence diagrams describe object interactions.
- Shows the time ordering of a sequence of method calls.
- Are valuable for documenting complex interactions between objects.
- Assure oneself at design time that there will be no surprises during the implementation.

A sequence diagram shows the time ordering of a sequence of method calls.

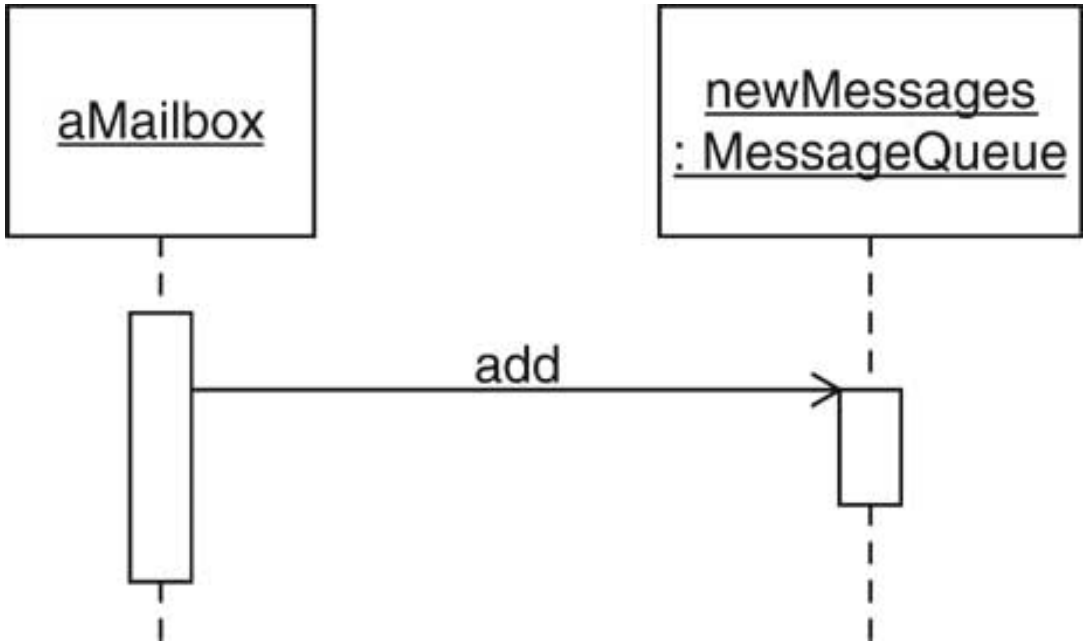
Sequence Diagrams

- When drawing a sequence diagram, you omit a large amount of detail; you do not indicate branches or loops.
- The principal purpose of a sequence diagram is to show the objects that are involved in carrying out a particular scenario and the order of the method calls that are executed.
- The question of "how to acquire collaborators" have to be answered.

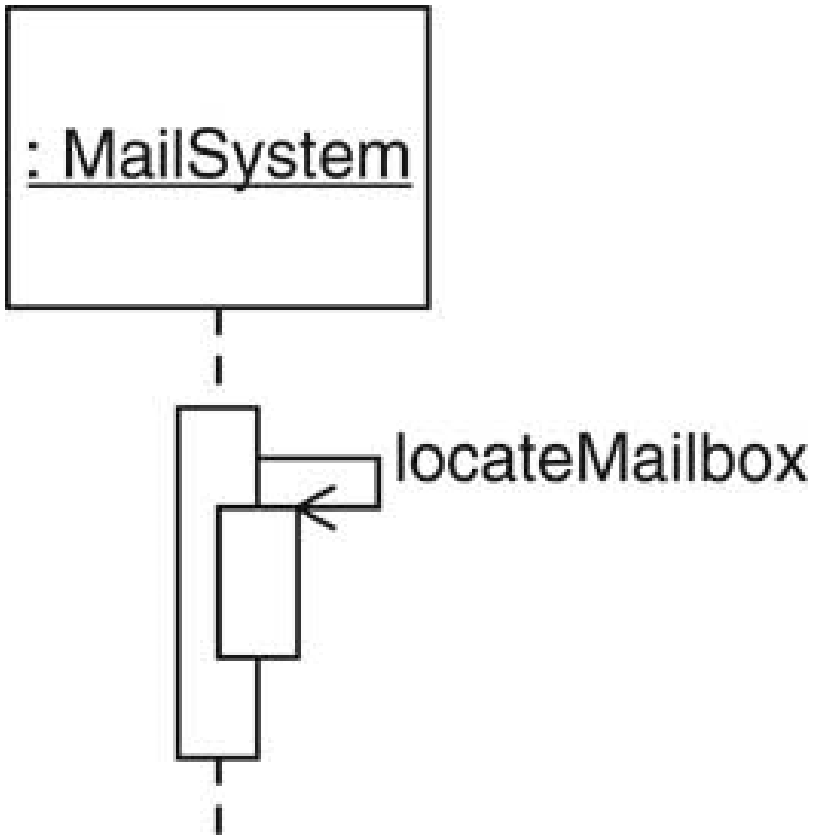
TIP If you played through a use case when using CRC cards, then it is probably a good idea to use a sequence diagram to document that scenario. On the other hand, there is no requirement to use sequence diagrams to document every method call.

Sequence Diagrams

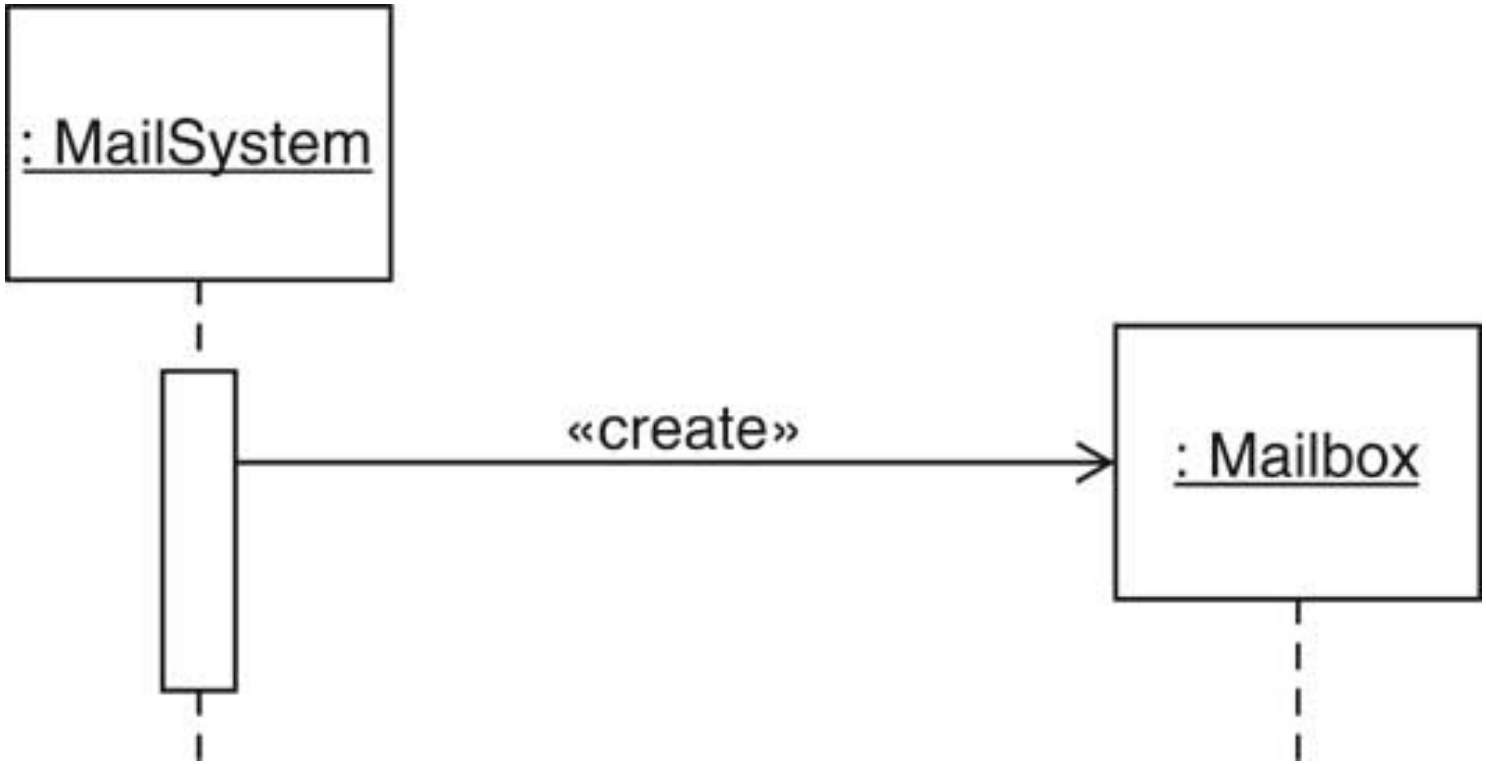
- Each diagram shows dynamics of scenario
- Object diagram: class name underlined



Self call

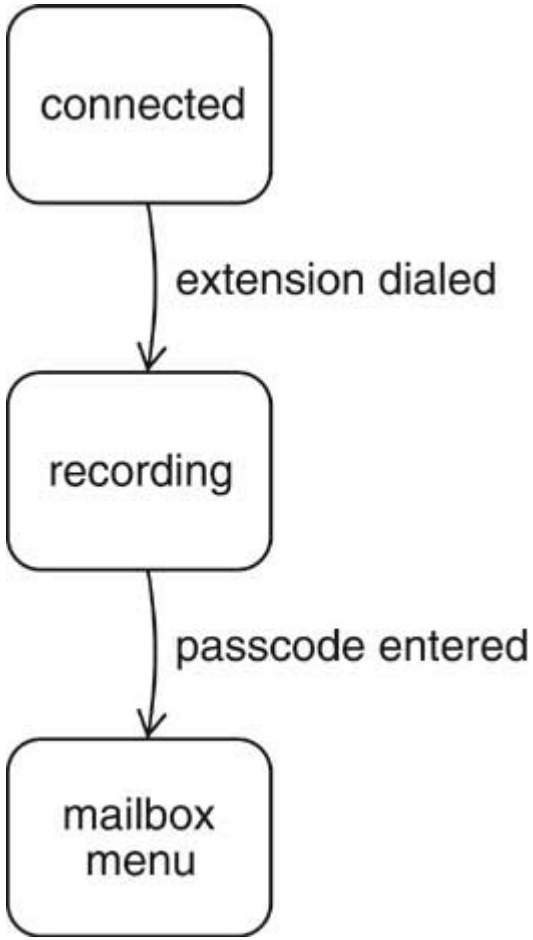


Object Construction



State Diagram

- A state diagram shows the states of an object and the transitions between states.
- Use for classes whose objects have interesting states



A state diagram shows the states of an object and the transitions between 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)
{
}
```

Design Documentation

- Don't compile file, just run Javadoc
- Makes a good starting point for code later
- Doc-1st approach; write documentation using *javadoc* and generate HTML document without writing code yet.

You can use `javadoc` to generate design information by applying comments to classes and methods that are not yet implemented.

Case Study: Voice Mail System

- To walk through the basic steps of the object-oriented design process, we will consider the task of writing a program that simulates a telephone voice mail system.
- We will define the behavior through a set of use cases.
- Use cases are not a full specification of a system.
- The functional specification also needs to define system limitations, performance, and so on.

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**)

Use Case: Log in

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

Use Case: Retrieve Messages

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.

TIP Consider reasonable generalizations when designing a system. What features might the next update contain? What features do competing products implement already? Check that these features can be accommodated without radical changes in your design.

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	

Scenario Walkthrough

- Now that we have some idea of the components of the system, it is time for a simple scenario walkthrough.
- Walkthrouhg scenarios start with a simple one.

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

Analyze Use Case: Leave a message

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,	Connection
microphone, hangup	
speak output	

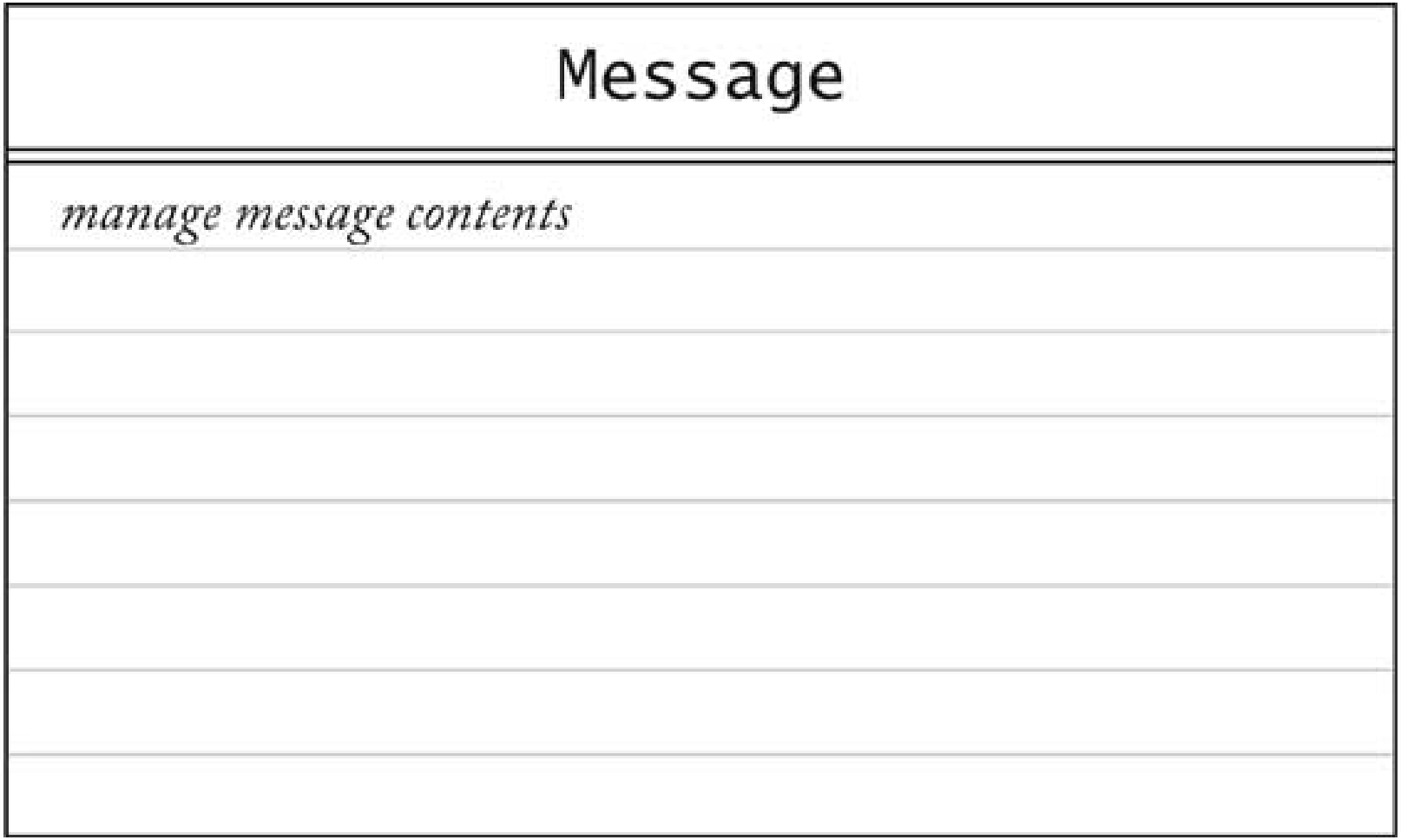
Result of Use Case Analysis

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



Analyze 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

Analyze Use Case: Retrieve messages

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

Analyze Use Case: Retrieve messages

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

- It is not easy to reason about objects and scenarios at a high level.
- It can be extremely difficult to distinguish between operations (easy to implement) and those that sound easy but actually pose significant implementation challenges.
- The only solution to this problem is lots of practice.
- Try your best with the CRC cards, and when you run into trouble with the implementation, try again.
- There is no shame in redesigning the classes until a system actually works.

CRC Summary

- Generally, when using CRC cards, there are quite a few false starts and detours.
- One purpose of CRC cards is to fail early, to fail often, and to fail inexpensively.
- It is a lot cheaper to tear up a bunch of cards than to reorganize a large amount of source code.

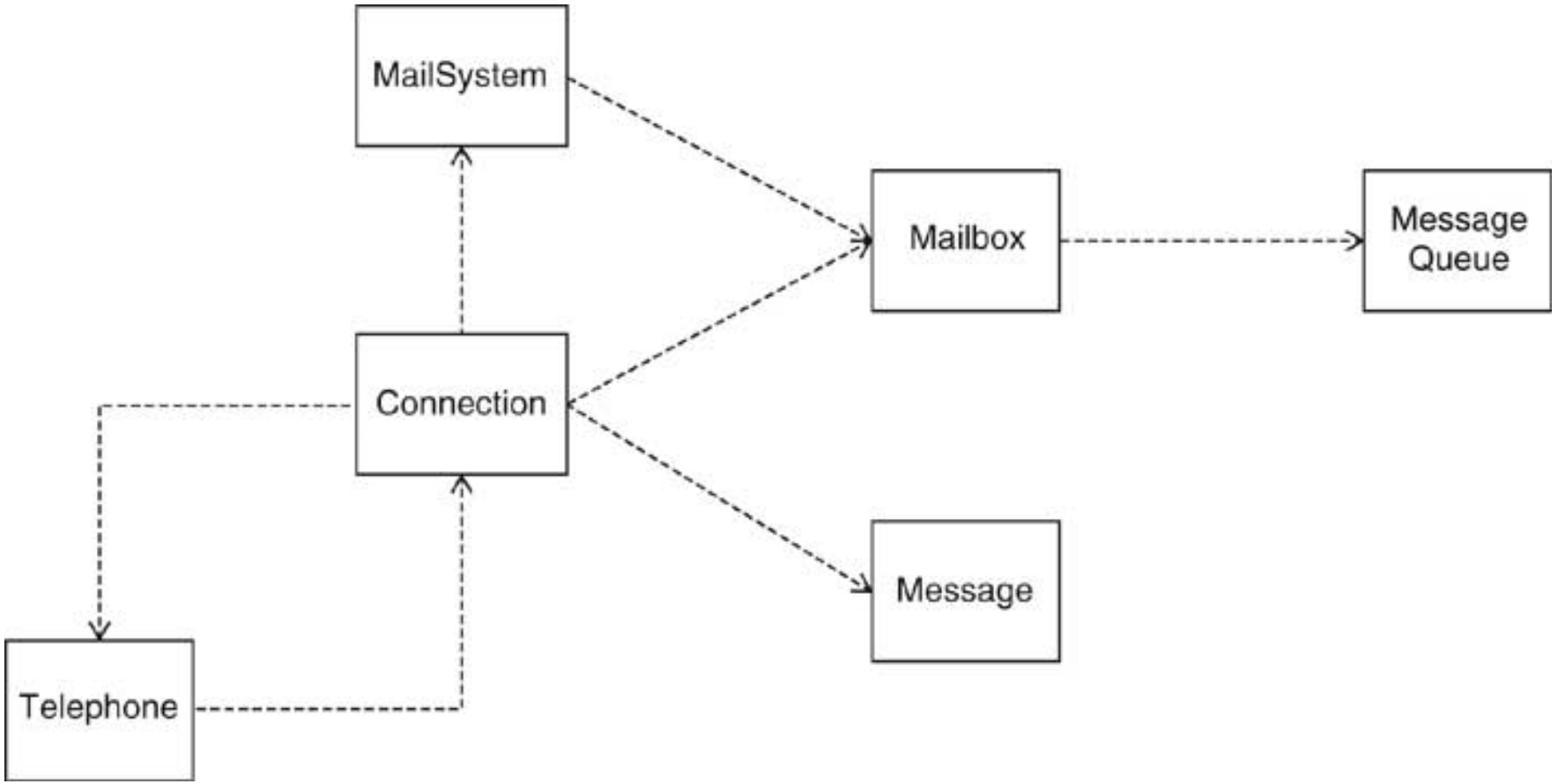
CRC Summary

- 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



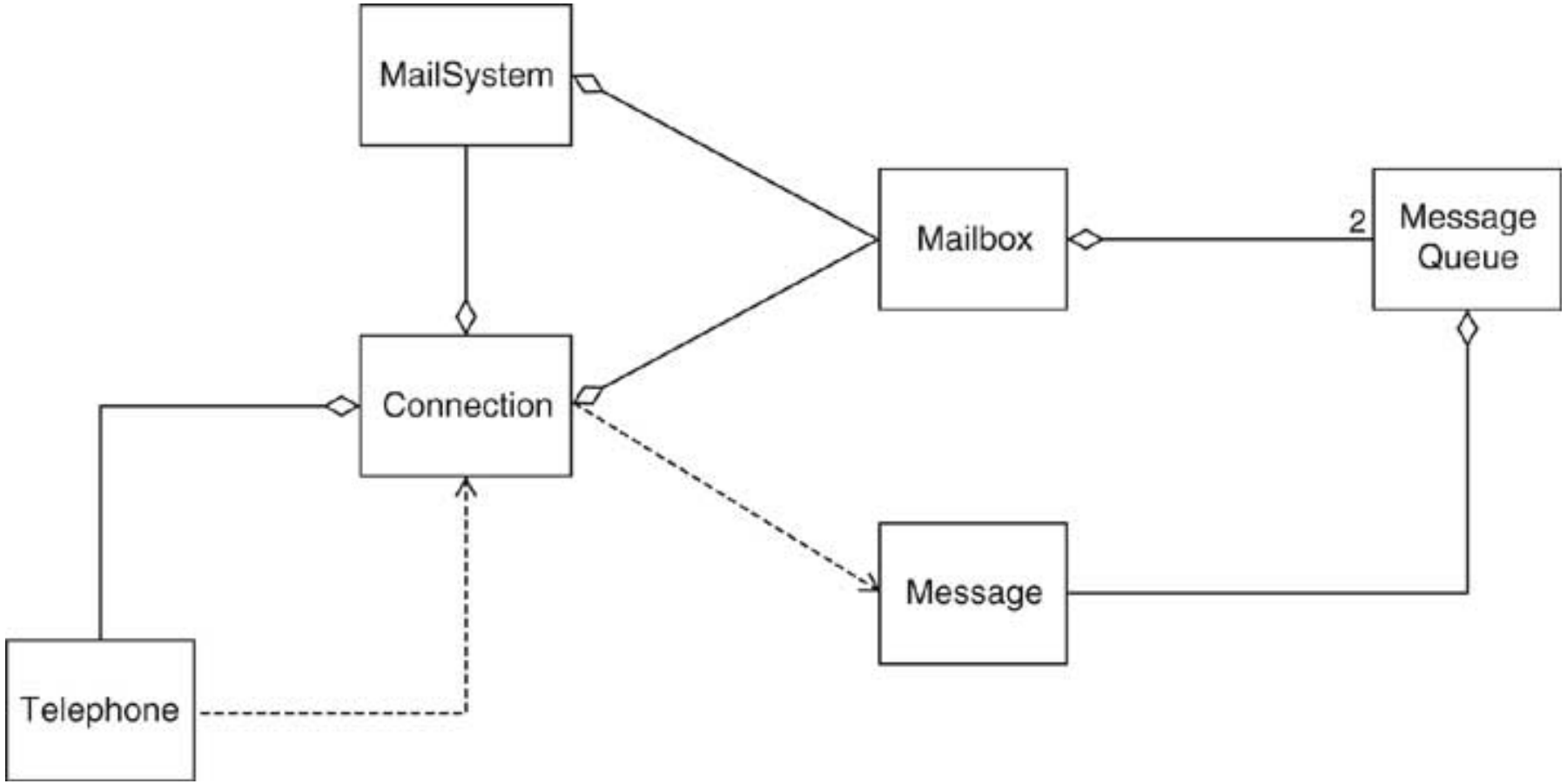
Aggregation Relationships

- Note that an aggregation relationship "wins" over a dependency relationship.
- If a class aggregates another, it clearly uses it, and you don't need to record the latter.

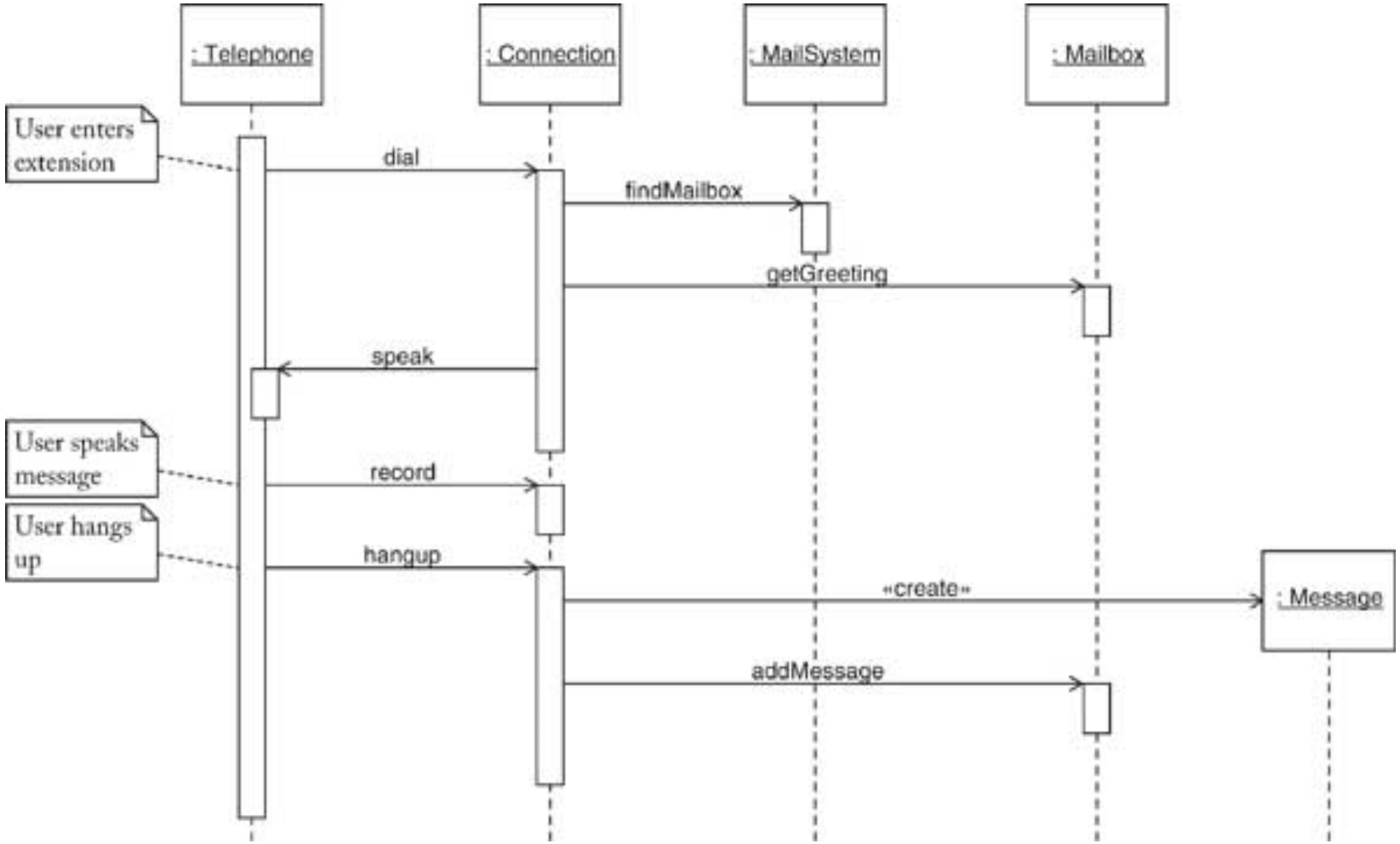
Aggregation Relationships

- 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

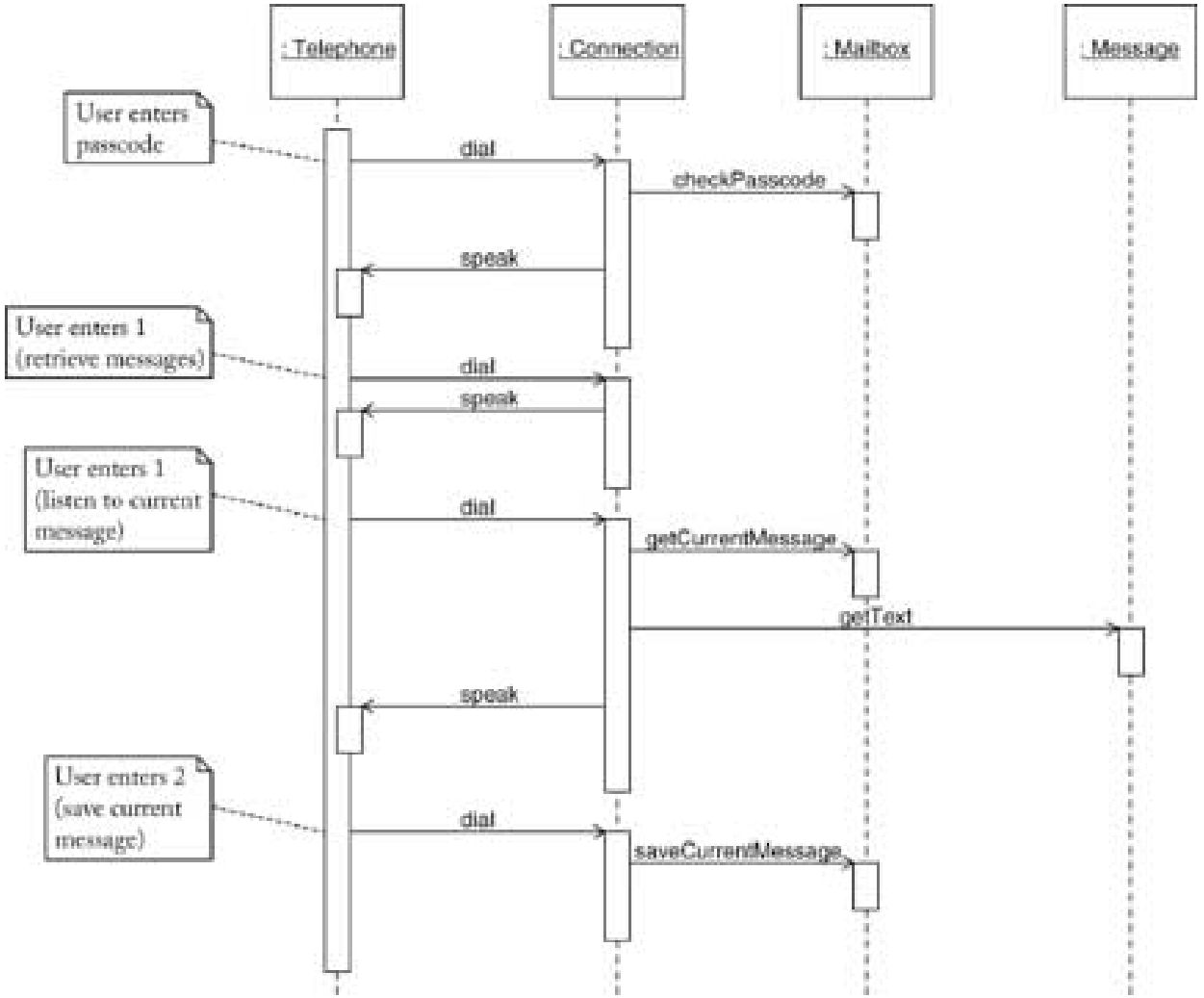
Interpreting a Sequence Diagram

- 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

UML Sequence and State Diagrams

- The purpose of a sequence diagram is to understand a complex control flow that involves multiple objects, and to assure oneself at design time that there will be no surprises during the implementation.
- Ask yourself exactly *where the objects of the diagram come from* and how the calling methods have access to them.

Sequence Diagram for Use Case: Retrieve messages



Java Implementations

- Run the program.
- Have a look at the code of the classes.
- Read the documentation comments and compare them with the CRC cards and the UML class diagrams.
- Look again at the UML sequence diagrams and trace the method calls in the actual code.
- Find the state transitions of a class.