Chapter 2: The Object-Oriented Design Process

In this chapter, we will learn the development of software based on object-oriented design methodology.

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

2.1 From Problem to Code

Requirements analysis, design, and implementation and testing are essential activities of a software development.

2.1.1 Requirements Analysis

- The goal of requirements analysis is to establish the functions, services, and constraints of the software to be developed. During this stage, the developers and the software users work together to produce a *requirements specification,* which will be the input to the design phase.
- There are two categories of requirements:
    - *functional* requirements which are concerned with functions and services to be carried out by the software.
    - *nonfunctional* requirements which are concerned with the constraints under which the software must operate, such as response time and memory consumption
- The requirements specification should address both functional and nonfunctional requirements.
- Functional requirements analysis
    - Goal: to transform a vague understanding of the problem into a precise description of task that the software system needs to carry out.

- A *use case* is a list of the user actions and system responses for a particular sub problem in the order that they are likely to occur. A set of use cases is an *analysis* technique to describe how a software system works.
- Typically, a functional specification is generated after building use cases. However, sometimes certain requirements are uncovered before the definition of the use case. Remember that software development is an iterative process.
- A functional specification, which is a part of requirements specification, has the following characteristics:
  Completely defines tasks to be solved
  Free from internal contradictions
  Readable both by domain experts and software developers
  Reviewable by diverse interesting parties
  Testable against reality

2.1.2 Object-Oriented Design Phase

- Goals
  - Identify classes
  - Identify the responsibilities of these classes
  - Identify relationships among classes

  These are goals, not sequential steps. The discovery process is iterative.
- Results
  - Textual description of classes and their important responsibilities
  - Diagrams of class relationships
  - Diagrams of important usage scenarios
  - State diagrams for objects with rich state

2.1.3 Implementation and Testing

- Implementation is the realization of the software design in a specific programming language. Each unit (a single class) is implemented separately.
- Unit testing is done to ensure that each unit functions properly with respect to its specification before the units are integrated.
- The individual units are integrated and tested as a whole to ensure that the entire software system works properly conforming to its specification.

Again, a software development in practice is not a simple linear progression through these activities - analysis, design and implementation. The common theme is to develop a software product in iterations.

2.2 Object and Class Concepts

Objects and classes are two fundamental concepts in object-oriented development. The representation of objects and classes dealt with in the object-oriented models is an approximation of the interpretation of objects and classes in the real world.

	Interpretation in the Real World	Representation in the Model
Object	An object represents anything in the real world that can be distinctly identified.	An object has an *identity*, a *state*, and a *behavior*.
Class	A class represents a set of objects with similar characteristics and behavior.	A class characterizes the structure of states and behaviors that are shared by all its instances.

Objects:

- The identity of an object distinguishes the object from other objects.
- The state of an object is composed of a set of field. Each field has a name, a type, and a value.
- The behavior of an object is defined by a set of methods that an object supports.

Note: Two objects are *equal* if their states are equal, that is, if the value of the corresponding fields of the two objects are equal. Two objects are *identical* if they have the same identify, that is, if the references are the same.

Class: A template for creating its instances (= objects). Specifically, a class defines (1) the names and types of all fields and (2) the names, types, and implementations of all methods.

2.3 Identifying Classes

At this level, you don't have to worry how the classes are implemented. Focus on concepts, not implementation.

(1) In the given functional description (Section 2.12), look for *nouns* for identifying an initial set of classes. Example: Mailbox, Message, Mail System, Telephone, and etc

(2) Turn your attention to other classes that are required to carry out necessary work. Example: MessageQueue - the storage of Messages with a FIFO policy.
(3) Check if there are any other required classes by looking at the categories of classes

Class Categories

Tangible things: Things easily identifiable in the problem domain (nouns)
Agents

- Sometimes it is useful to convert an operation of a class to an agent class.
- It has characteristics around the action it carries out.
- Often we use agents to decouple operations from a class.
- Example: The operation of parsing input is encapsulated in the Scanner agent.

Events and transactions

- Typically used to retain information from the past.
  - The last mouse position, the last set of coordinates of a plane, the last keystroke.

        - The MouseEvent class remembers when and where the mouse was moved or clicked.

User and role

- Used to establish different users with different roles and permissions of a system.
- Administrator, Reviewer

Systems

- Model a subsystem or the overall system being built.
- Used to initiate and terminate the system
- Example: MailSystem class.

System interfaces and devices

- These capture system resources and the interaction of the system.
- Display window, input reader, output file, etc.
- Example: the File class

Foundational Classes

- These are typically generic fundamental classes.
- At the beginning we should assume they exist.
- They encapsulate data types with well defined properties and actions.
- These classes are the highest focus for reuse.
- Example: Date, Rectangle, String

2.4 Identifying Responsibilities

- To discover responsibilities, look for *verbs* in problem description.
- A responsibility must belong to exactly one class.
    - Example: Add message to mailbox;
    - Who is responsible: Message or Mailbox?
- A responsibility of a class should stay at one abstraction level.
    - Highest level: Mail System
    - Mid-level: Mail box – shouldn't deal with
        - processing key strokes (lower level)
        - initialization of the system (higher level)
    - Lowest level: files, keyboard and mouse interfaces

2.5 Relationships between Classes

The following relationships can exist among classes:

- Generalization that includes inheritance and implementation; is-a relationship
- Association; general binary relationship without concerning specific implementation
- Aggregation and Composition; has-a relationship
- Dependency; uses relationship

2.5.1 Generalization

There are two specific forms of generalization: inheritance and implementation.

(1) Inheritance relationship is formed when a class (or an interface) extends another class (or interface.

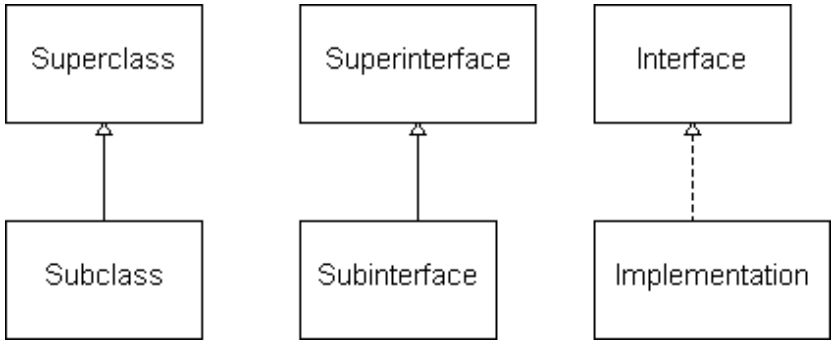
The *inheritance* relation between classes.
- When class C2 *extends* class C1, class C2 is a subclass of C1, and C1 is a superclass of class C2.
- A subclass inherits and reuses the fields and methods of a superclass.

The *inheritance* relation between two interfaces.
- When interface I2 *extends* interface I1, interface I2 is a sub interface of interface I1, and interface I1 is a super interface of interface I2.
- This relation represents the expansion of the service contract. If a class implements I2, the class should supply all the methods defined in I1 and I2 to be a concrete class.

(2) The *implementation* relation between a class and an interface.
When class C2 implements interface I1, class C2 is known as an implementation of interface I1, and interface I1 is known as an interface of class C2.
Class C2 should implement all the methods defined in I1 to be a concrete class.

(3) Conceptually, generalization models the *is-a* relationship in the real world; that is, if C2 is a subclass/subinterface/implementation of C1, then every instance of C2 *is a* C1.

(4) UML notation for generalization

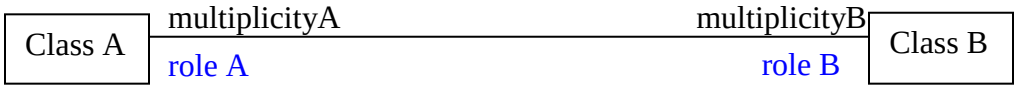


Inheritance of classes | Inheritance of interfaces | Implementation of interfaces

2.5.2 Association

- Association represents *general* binary relationships between classes; A class directly or indirectly references the other class.
- The association relationship can also be implemented in a variety of different ways.

UML notation for association relationship.



Multiplicity

- *l..u* specifies an inclusive range of integers from the lower bound *l* to the upper bound *u*. The lower bound may be any positive number or zero. The upper bound may also be any positive number or * (for unlimited).
- *i* specifies a singleton range that contains integer *i*.
- * specifies the entire nonnegative integer range: 0, 1, 2, 3, …

Example

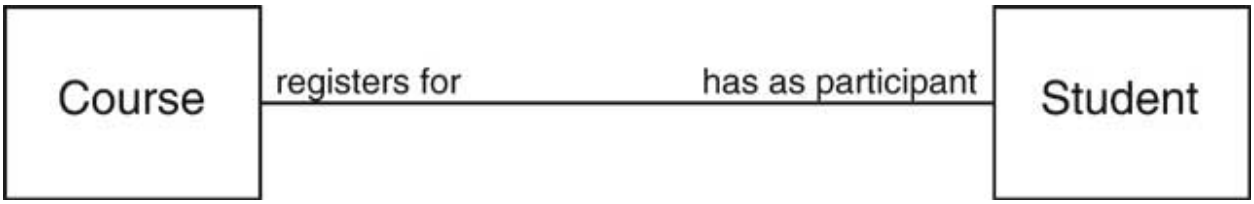
Indicator	Meaning
0..1	Zero or one
1	One only
0..*	Zero or more
2..5	Two to five
2,4	Two or four (Discontinuous multiplicities are not common and UML2 removed them.)

Association Link

The solid line indicates the *navigation of the association*. If there is a direct or indirect reference from class C1 to class C2, then it is *navigable* from C1 to C2.

- By default, an association is assumed to be navigable in both directions.

Example: Course has set of students and student registers for set of courses



- If an association is *only navigable in one direction*, it must be explicitly shown with a navigation arrow at the end of the association link pointing in the direction that is navigable. (The text book terminology is "directed association".)

  Example: Message doesn't know about message queue containing it

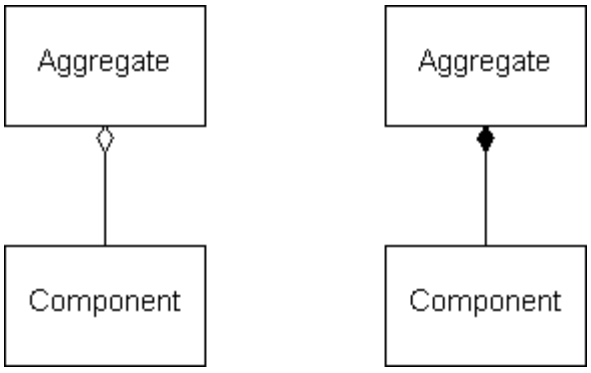
2.5.3 Aggregation and Composition

(1) Aggregation
- A special form of association (an association definitely represented by an instance field).
- Represents the *has-a* or *part-whole* relationship.
- Simply a structural relationship that distinguishes the whole, that is, the aggregate class, from the parts, that is, the component class.
- Does not imply any relationship in the lifetime of the aggregate and the components.

(2) Composition
- A stronger form of aggregation
- The aggregate class exclusively owns the component class.
- The lifetime of the components is entirely included in the lifetime of the aggregate.

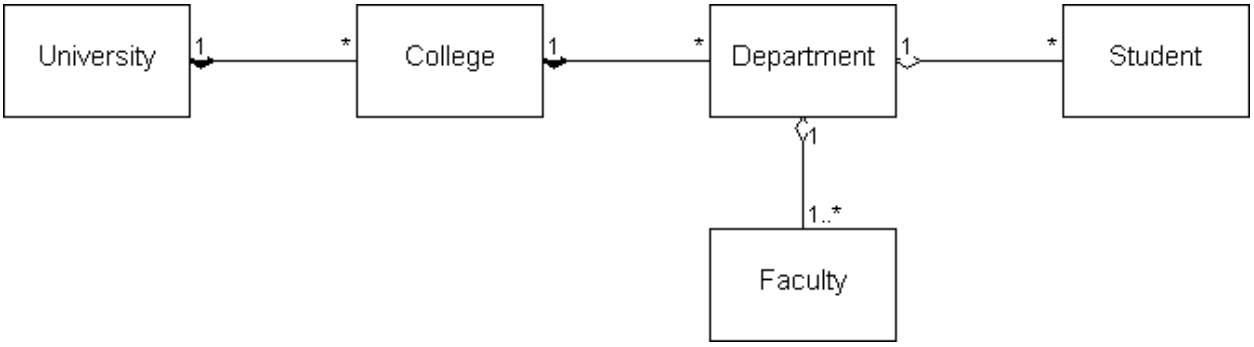
Note:
a. The distinction between aggregation and composition is entirely conceptual. We will not distinguish aggregation and composition, and textbook uses *only aggregation* notation.
b. *Not all* instance fields of a class correspond to aggregation. Attributes are also instance fields. If the type of an instance field is a foundational type such as a number or a string, the field is considered as an attribute, not aggregation.

(3) UML notations



Aggregation Composition

(4) Example



2.5.4 Dependency

(1) A common form of dependency is the *use* relation among classes. In other words, Class C1 depends on class C2 if C1 *uses* C2 in places such as the parameters, local variables, or return types of its methods.

(2) Minimize the number of dependencies between classes: reduce coupling. When classes depend on each other, changes in one of them can force changes in the others.

Example: Replace

```
public class Message
{  void print() { System.out.println(text); }
   …
}
```

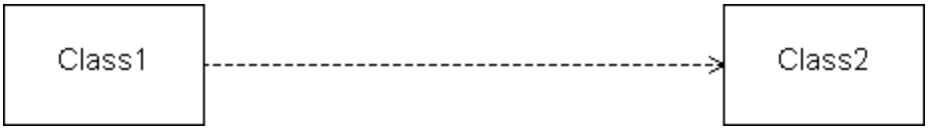
with

```
public class Message
{ String getText() { // can print anywhere }
```

```
    …
}
```

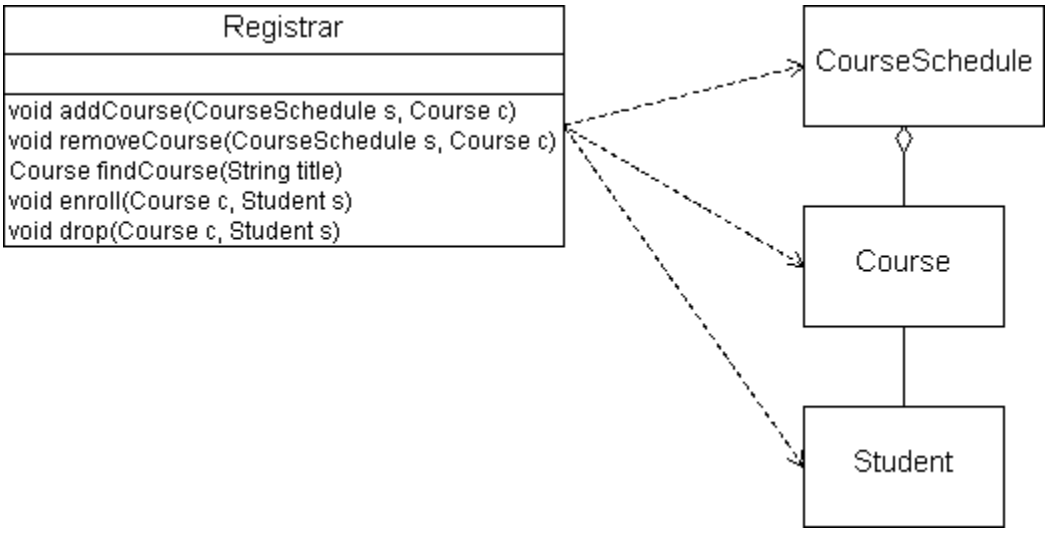
to remove dependence on the `System` and the `PrintStream` classes.

(3) UML notation for dependency



Class1 depends on Class2.

(4) Example



2.6 Use Cases

(1) Overview

- Uses cases are an *analysis* technique.
- A use case is a list of the user actions and system responses for a specific scenario in the order that they are likely to occur. A use case must yield an observable result that is of value to the user of the system.
    - Example
        - In the case of "Leave a Message", the value to the caller is the fact that message is left in the mail box.
        - Just "Dialing a phone number" would not be a valid use case.
- Use *variations* for exceptional situations

(2) Example of a use case

Use Case: Leave a Message

Step	User's Action	System's Response
1	The caller dials the main number of the voice mail system.	
2		The voice mail system speaks a prompt. `Enter mailbox number followed by #.`
3	The caller types in the extension number of the message recipient.	

4		System speaks
		`You have reached mailbox xxxx.`
		`Please leave a message now`

5	The caller speaks message	
6	The caller hangs up	
7		The voice mail system places message in mailbox

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

(4) The role of use cases:

- Use cases capture the functional requirements of the system.
- Use cases play a role to the extent that they enhance the developer's domain knowledge.
- The validation team validates that the system does indeed satisfy the requirements by performing a "thought experience" walking through the use cases.
- Use cases can be used to refine CRC cards

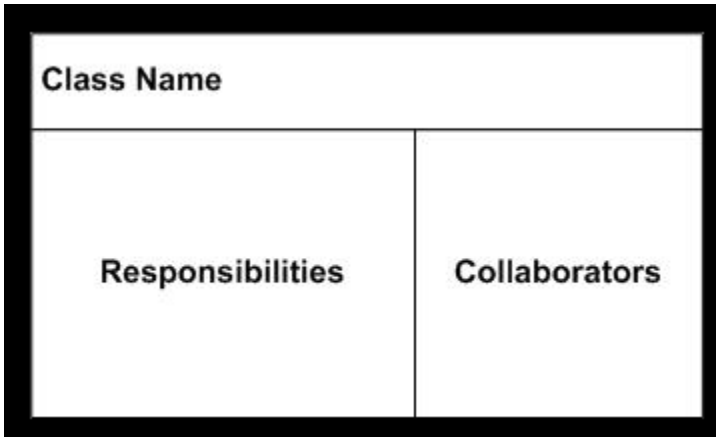
2.7 CRC (Classes, Responsibilities, and Collaborators) Cards

The CRC method is a very effective design tool in identifying classes, their responsibilities, and relationships to other classes.

(1) The CRC card

- Use one index card for each class. (Typically, an `3″ x 5″` index card)

- Class name on top of card
- Responsibilities on left
    - The responsibilities should be at a high level without concerning implementation details.
    - Between 1~3 responsibilities per card is ideal.
- Collaborators on right
    - The collaborators don't have to be on the same lines as the responsibilities.
    - List collaborators as you discover them, without regard for the ordering.



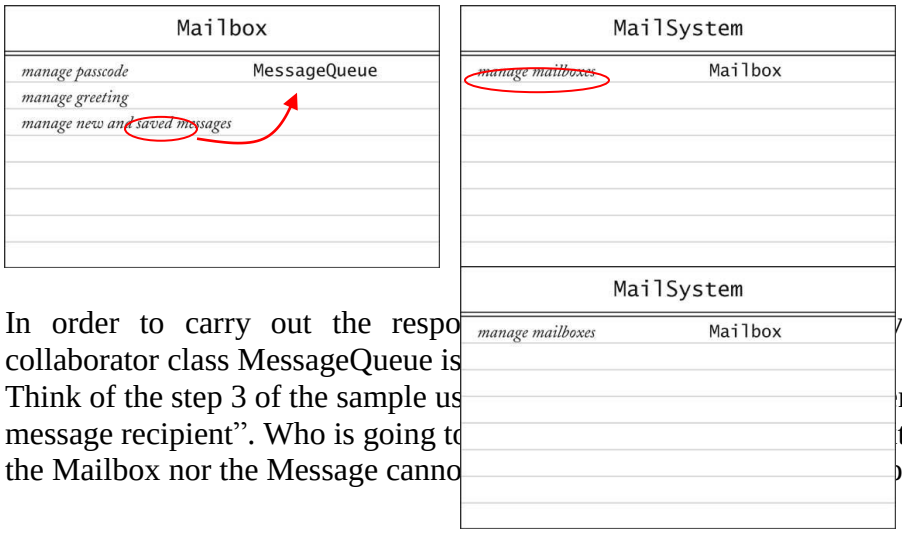
(2) How do you create CRC models ? A CRC model is created *iteratively* performing the following steps:

- Find classes
- Find responsibilities
- Define collaborators
- Move the cards around in a way that two cards that collaborate with one another should be placed close together. By doing so, it is easier to understand the relationships between classes
- Walk through use cases, naming cards and responsibilities

(2) Tips for using CRC cards

- Do not equate responsibilities with methods. A single responsibility may give rise to a number of methods. Between one and three responsibilities per card is ideal.
- Do not overburden a class with too many responsibilities. If it happens, split the class in two.
- Responsibilities in the same class should be related.
- Do not add responsibilities just they can be done.

(3) Example



- In order to carry out the respo ved messages", the collaborator class MessageQueue is
- Think of the step 3 of the sample us nsion number of the message recipient". Who is going t tension ? Neither the Mailbox nor the Message canno f its mailboxes !

2.8 UML (Unified Modeling Language) Diagrams

UML defines various types of diagrams. We will learn three of them: *Class* Diagrams, *Sequence* Diagrams, and *State* Diagrams

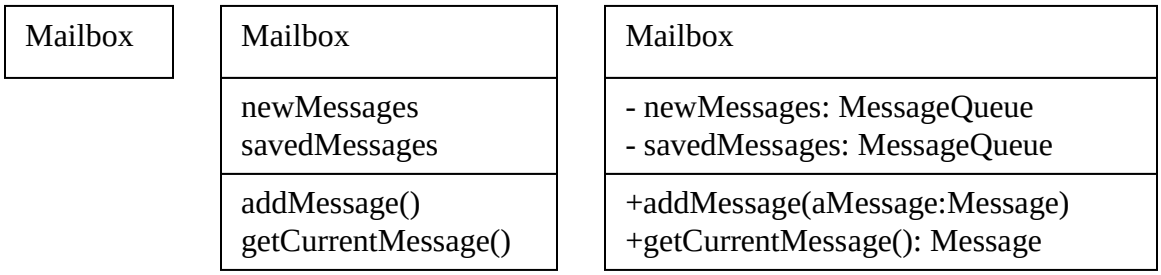
2.8.1 Class Diagrams

(1) Notation for class

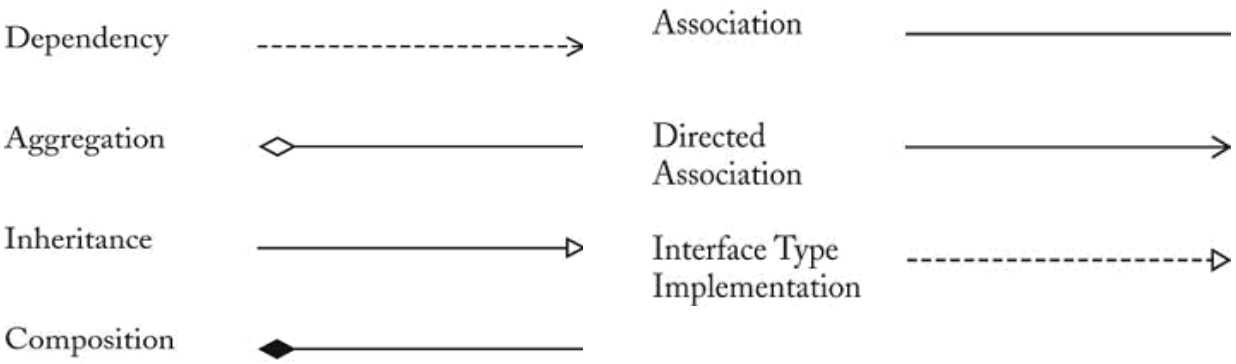
- Rectangle with class name
- Compartments (Optional)
  - Attributes
    - You can also specify the type of attributes.
    - Syntax: *attribute: Type*, for example, text: String
  - Methods
    - You can also specify the parameter and return types of a method
    - Syntax: *method_name (pname1: type1, pname2:type2, ...): return_type*, for example, getMessage(index : int) : Message
  - Include only key attributes and methods
- Visibility of features (Optional)

Visibility	Java Syntax	UML Syntax
public	public	+
protected	protected	#
package		~
private	private	-

- Example: Various UML representations for the Mailbox class



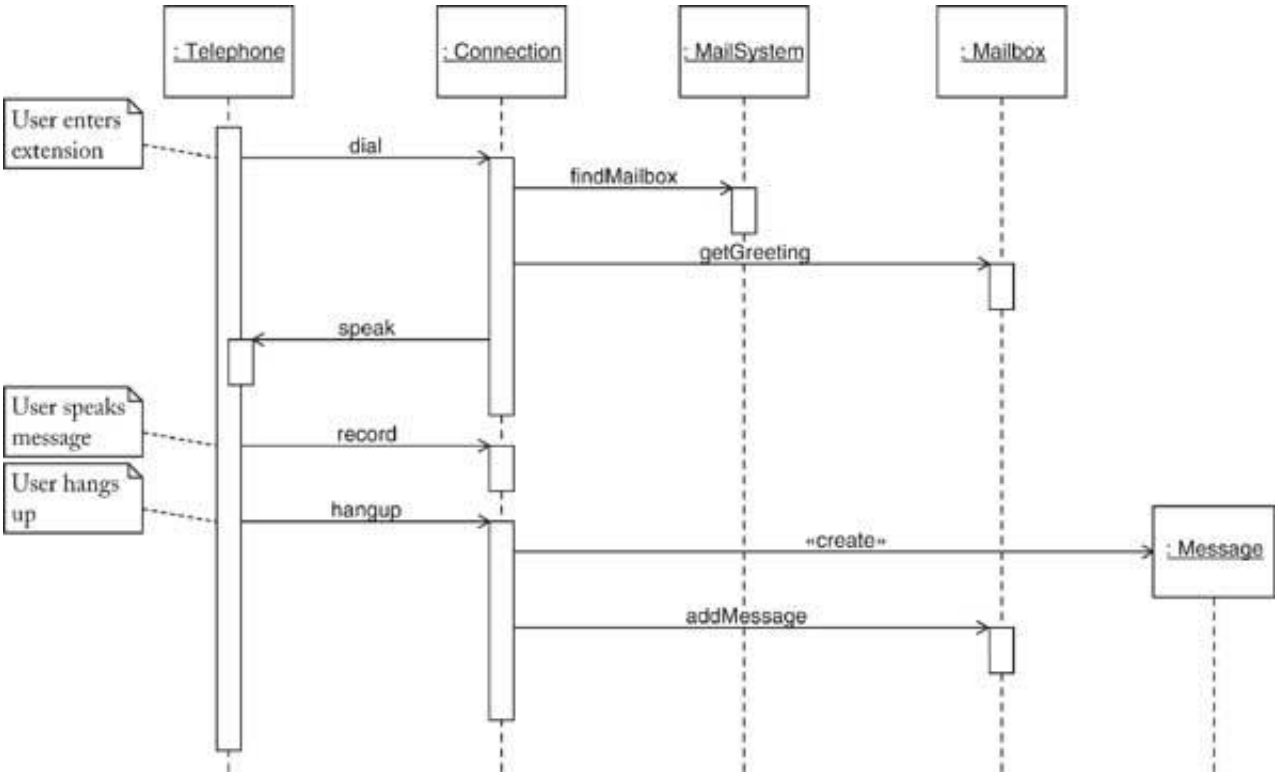
(2) Notations for class relationships



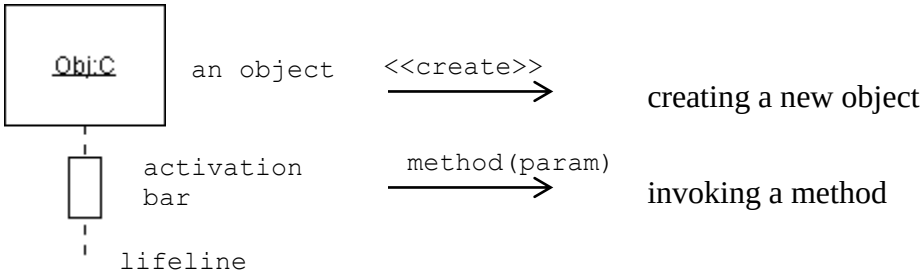
2.8.2 Sequence Diagrams

(1) Overview

- Sequence diagrams are used to show the flow of information, that is interactions between objects, through the program.
- Sequence diagrams are generally developed on a use case basis and how the message sequence (that is, whose method is called by whom) associated with a particular use case.
- A sequence diagram is a two-dimensional diagram with time running down the vertical axis and objects listed across the horizontal axis.
- Example: Sequence diagram for “Leaving a Message”



(2) notations used in a sequence diagram

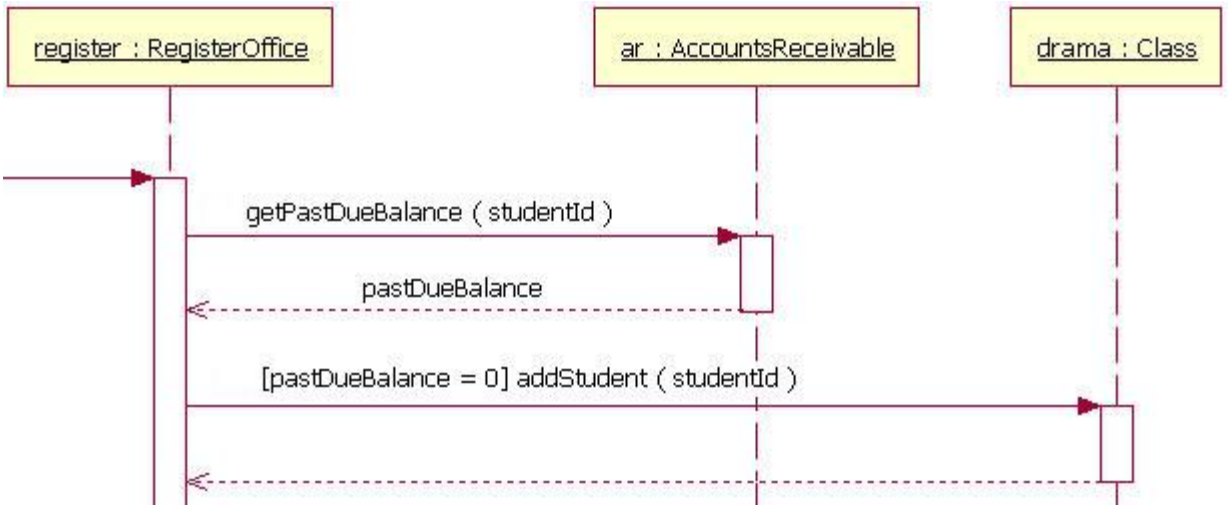


- Objects
  - A rectangle with the name of the object underlined.
  - Objects are listed across the top of the sequence diagram unless they are created during the time period represented by the sequence diagram.
  - If an object is created, it is shown lower in the diagram. Example: Message
- Life Lines
  - A dashed line that begins when the object is created and ends when the object is destroyed.
- Activation Bars
  - The thin long rectangles along the life line. When a method is invoked, the activation bar starts, and when the method returns, it ends.
- Messages (Method Calls)

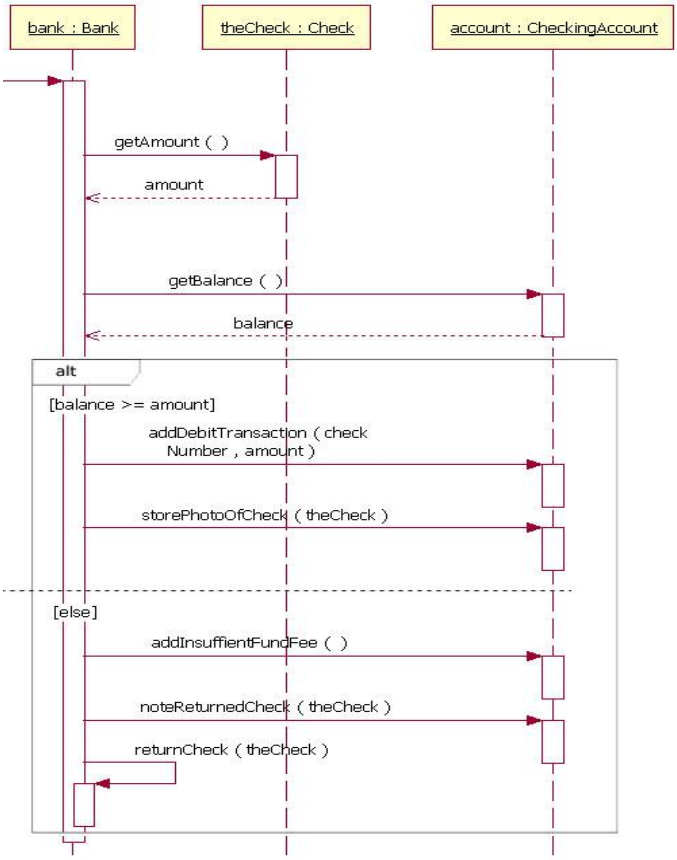
- Messages are indicated by a horizontal arrow from the sending object to the receiving object. The method name is shown above the arrow. The parameters are optional.

- Use of Notes
  - Free form of text enclosed in a rectangle with the upper right corner folded down.

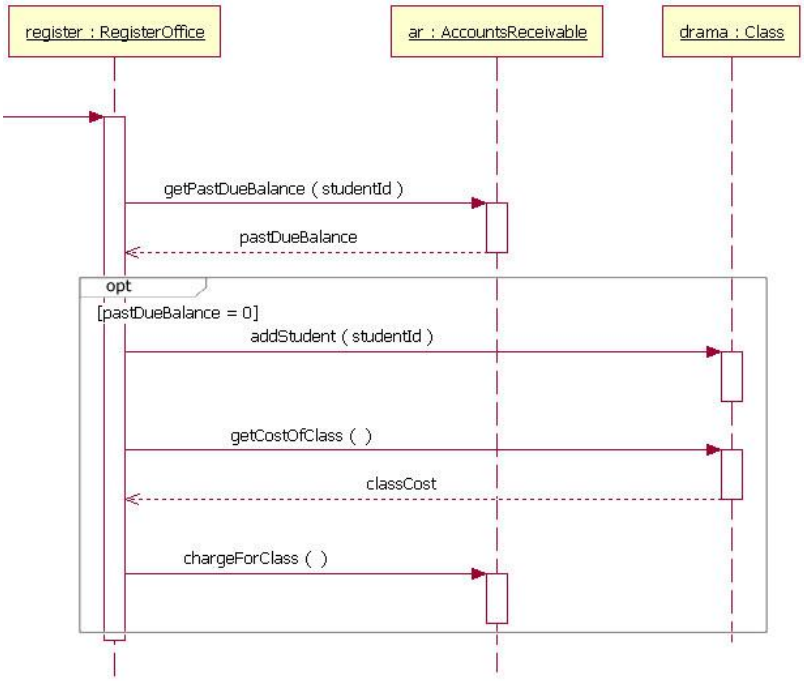
- Guards
  - A condition must be met for a message to be sent to. The following example shows a guard [pastDueBalance = 0] on the message addStudent method.



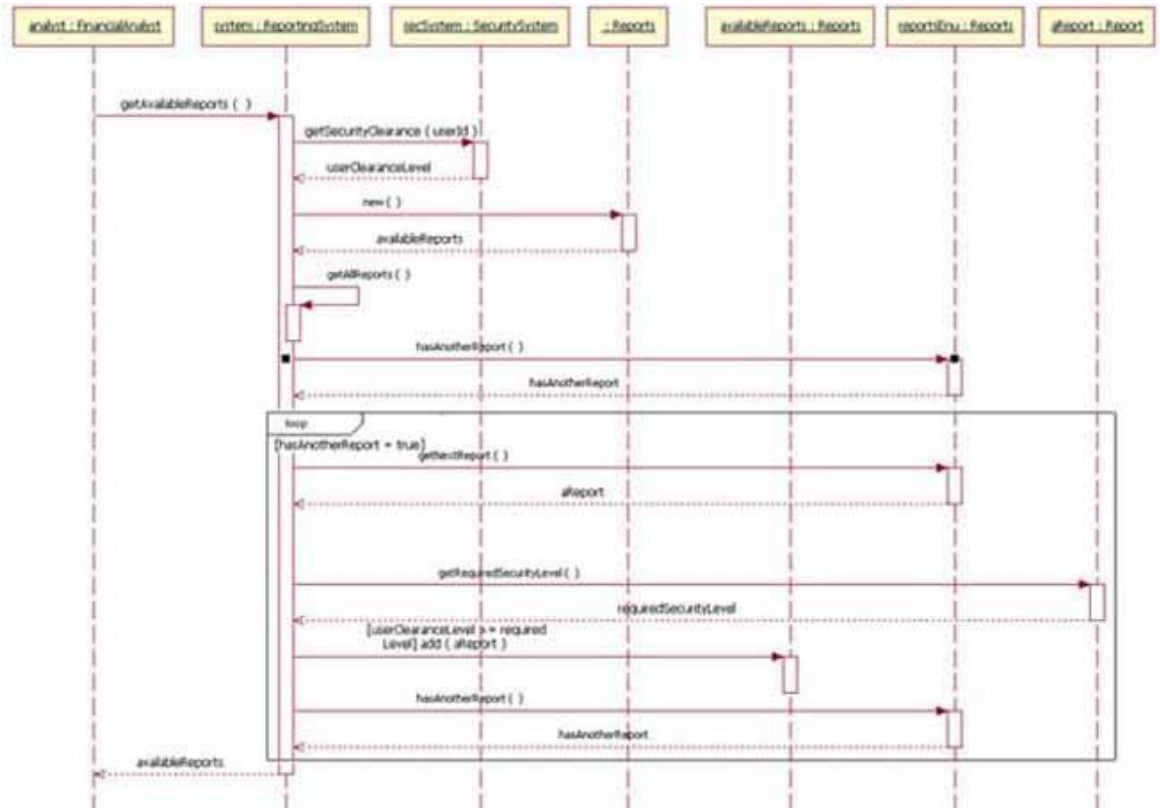
- Alternatives
  - To designate a mutually exclusive choice between two or more message sequences. Alternatives allow the modeling of the classic "if then else" logic.



- Option
  - To model a sequence that, given a certain condition, will occur; otherwise, the sequence does not occur. An option is used to model a simple "if then" statement



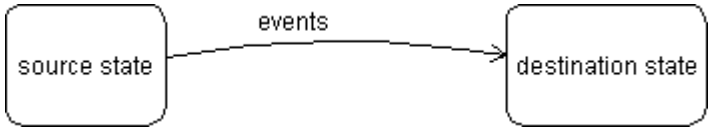
- Loop
  - To model a repetitive sequence.



2.8.3 State Diagram

State diagrams depict the flow of control using the concepts of states and transitions.

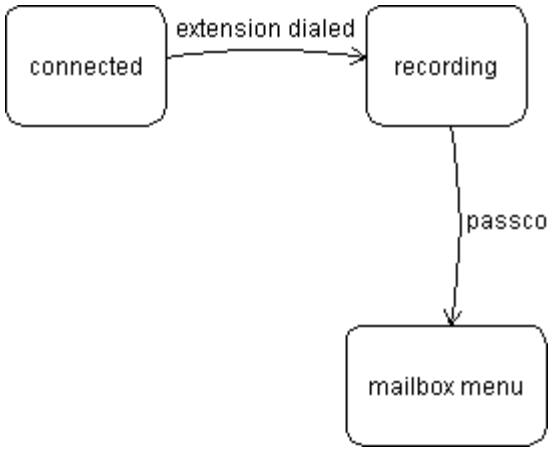
- A *state* is a condition in the life of an object.
- A *transition* is a relationship between two states indicating that an object in the source state will perform certain actions and enter the destination state.
- An *event* is an occurrence of a stimulus that can trigger a state transition.



States are particularly common with objects that interact with the program user. Suppose there is an object, as a part of the voice mail system, that serves the user to

Enter the extension number of a mail box
Enter the pass code
Enter a menu to start playing message

- The corresponding state diagram for such an object is



2.9 Using javadoc for Design Documentation

- 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 to extract the HTML documentation.
  - To share the HTML documentation with your team members.
  - Makes a good starting point for code later

2.12 Case Study: A simple Voice Mail System

Initial Set of Requirements

A person dials an extension number and leaves a message. The three distinct input events are simulated: speaking, pushing a button, and hang up

Pushing a button: An input line consisting of 1 2 ... 0 or #.

Hang up: An input line consisting of a single line H.

Speaking: Any other text means voice input. (Represent voice mail by text that is entered through keyboard.)

Analysis using Use Cases

Use Case: Reach an Extension

Steps	User's Action	System's Response
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

Steps	User's Action	System's Response
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

Steps	User's Action	System's Response
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

Steps	User's Action	System's Response
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. Note: Message is played, not removed from queue

Variation #1

1.1. Start at Step 3
1.2. User selects "delete current message".
Message is permanently removed
1.3. Continue with step 3

Variation #2

1.1. Start at Step 3
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

Steps	User's Action	System's Response
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

Steps	User's Action	System's Response
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.

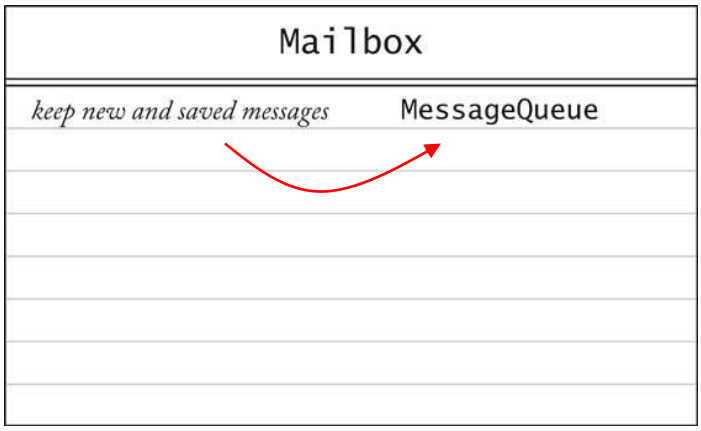
You can define the specific system behavior through a set of use cases. Note that the set of use cases by itself is not the functional specification of the system. You need to produce a precise functional specification as a result of analysis.

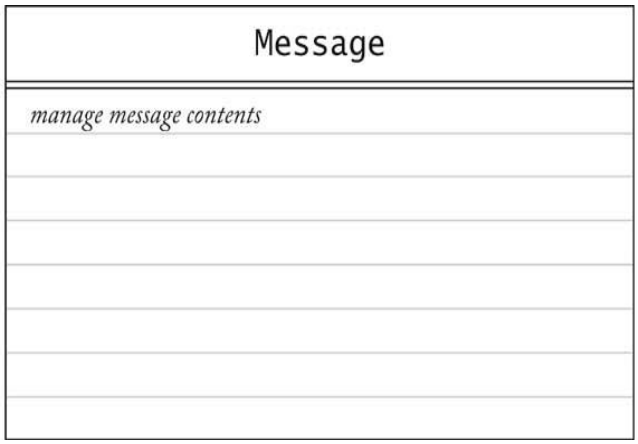
Design using CRC Cards for Voice Mail System

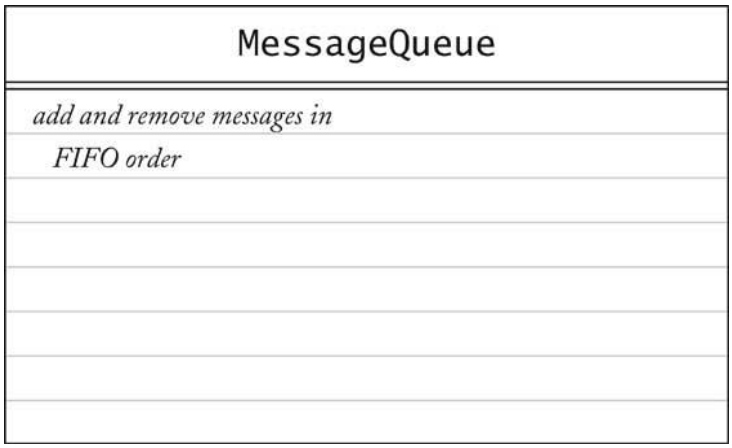
Through noun/verb analysis of the functional specification, you can identify obvious classes and their responsibilities.

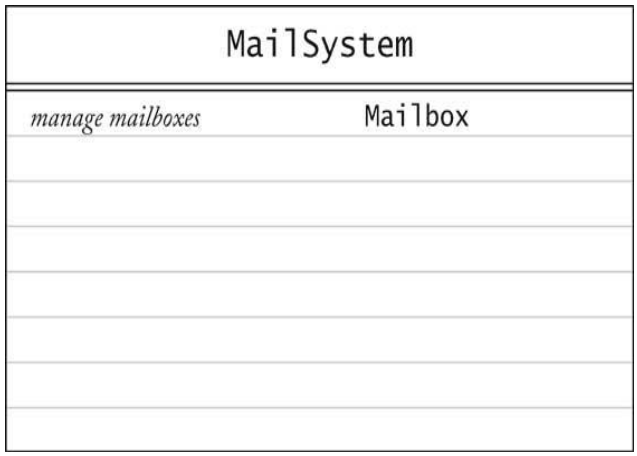
- Mailbox
- Message
- MailSystem

Initial CRC Cards:









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

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

[Q] What if there are multiple telephones ?
Who is in charge of tracking all connection states ?

- 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	
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
   (Add collaborator MailSystem to Connection)
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 the 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 collaborator Message to Connection)
8. Connection adds Message to Mailbox

Result of Use Case Analysis

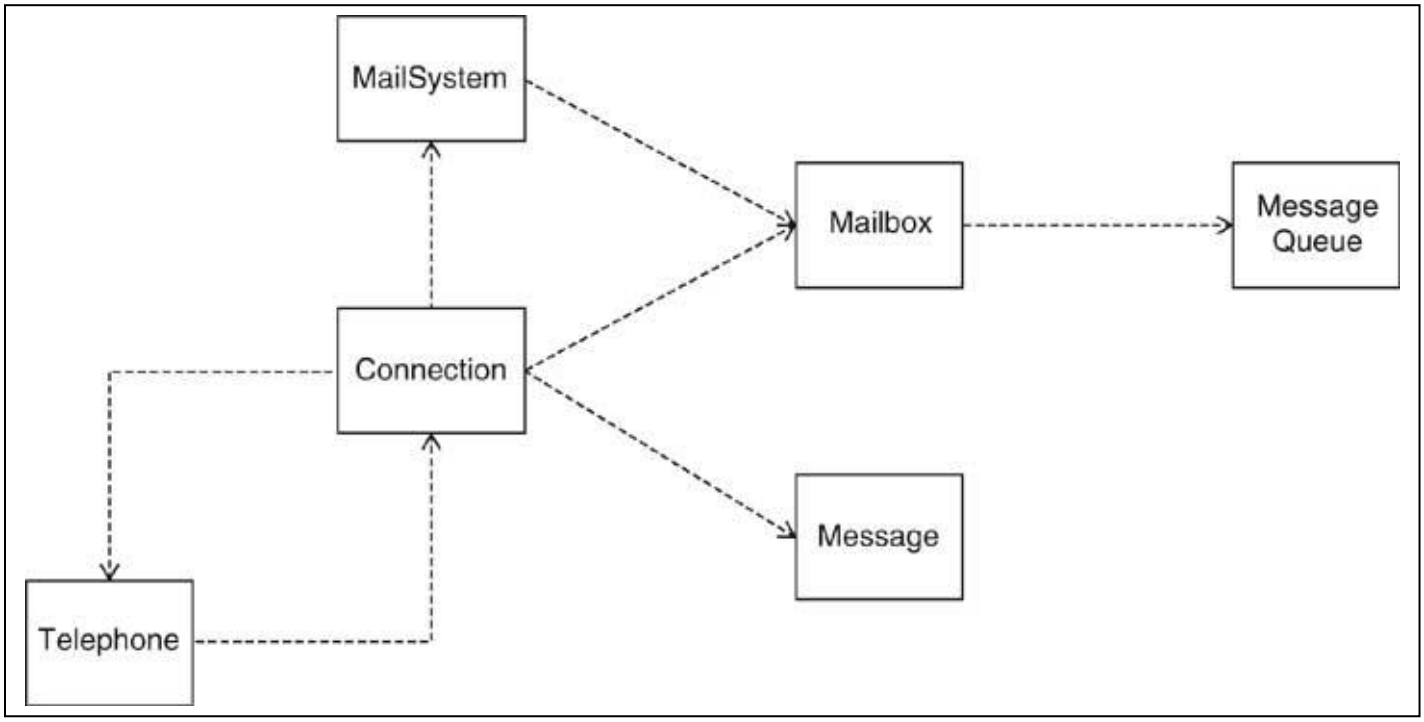
Telephone	
take user input from touchpad, microphone, hangup	Connection
speak output	

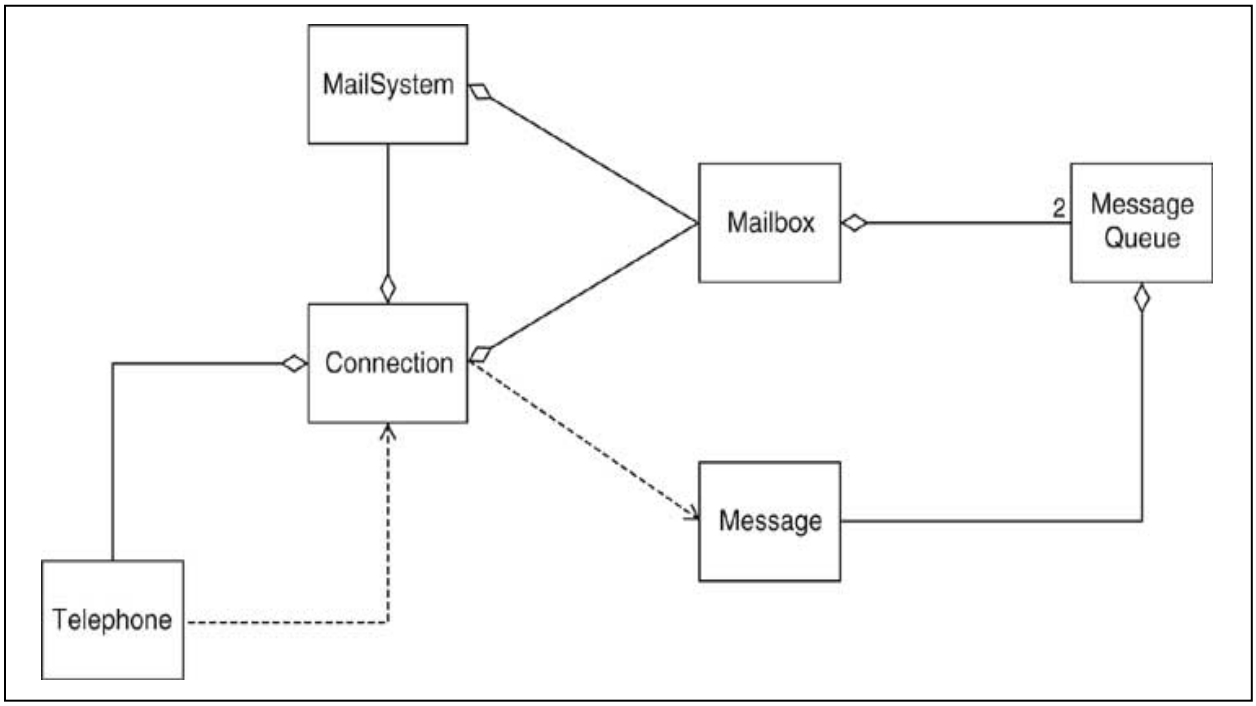
Connection	
get input from telephone	Telephone
carry out user commands	MailSystem
keep track of state	Mailbox
record voice input	Message

Mailbox	
keep new and saved messages	MessageQueue
manage greeting	

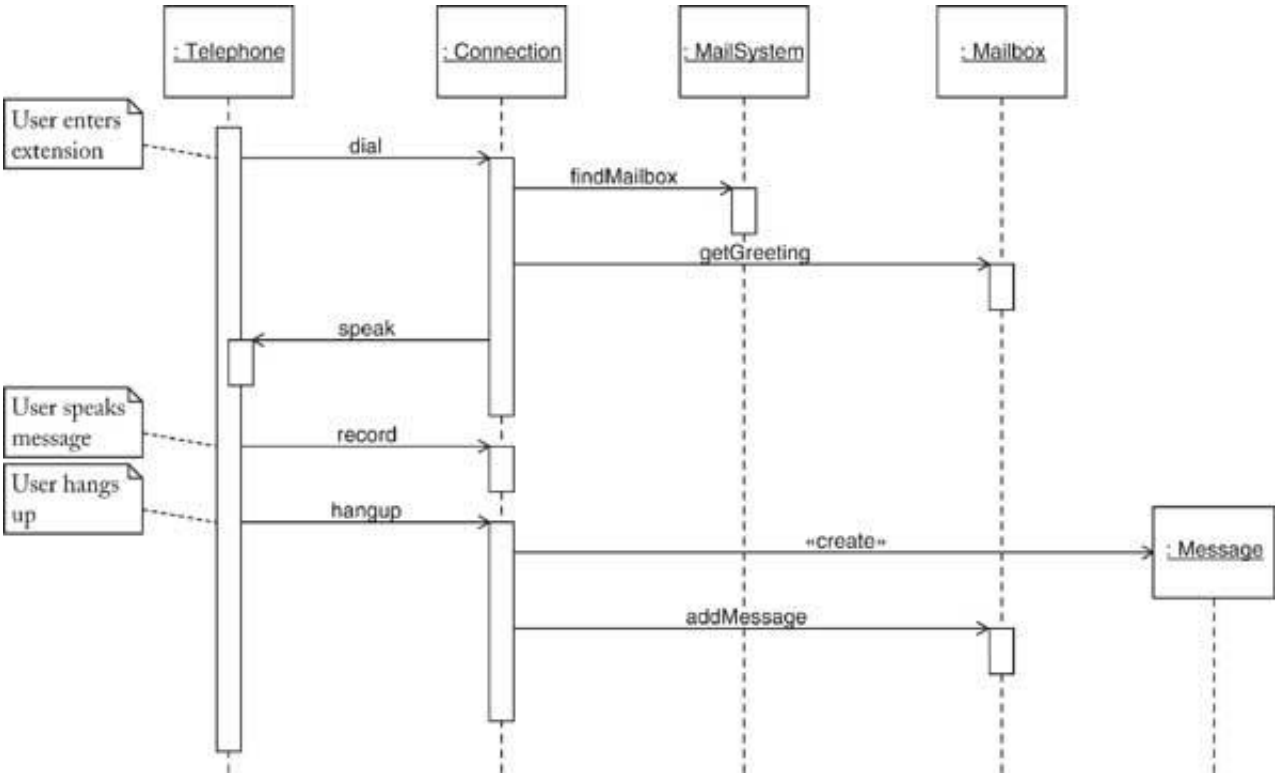
UML Class Diagram for Mail System

(1) Dependency Relationships

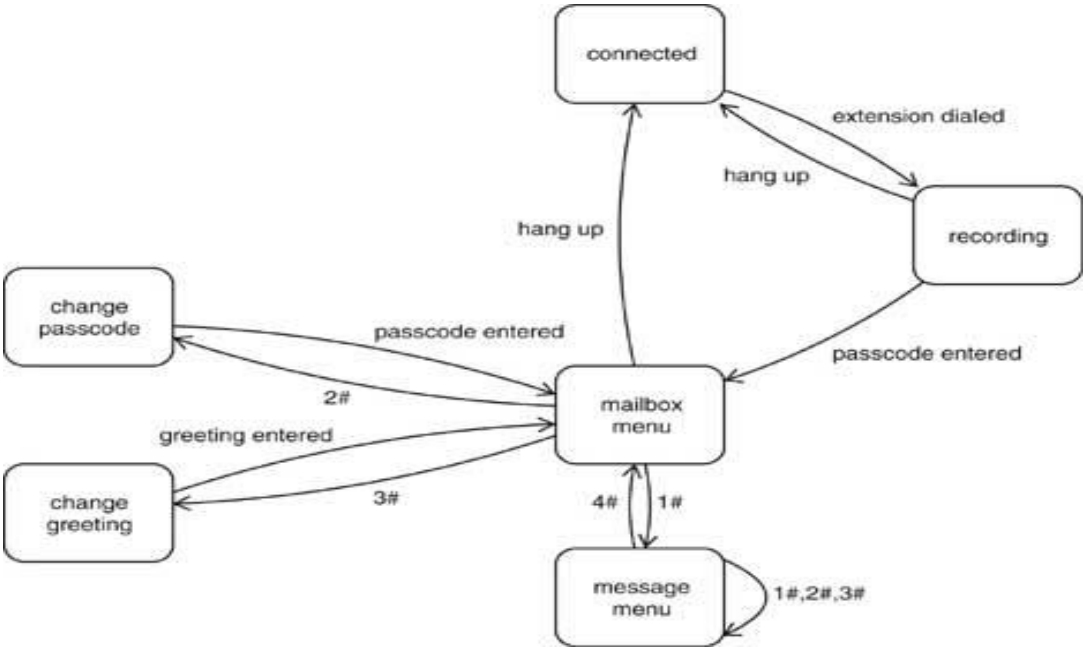




Sequence Diagram for Use Case: Leave a message



Connection State Diagram for the Connection object



Java Implementation: source codes are available from the text book.