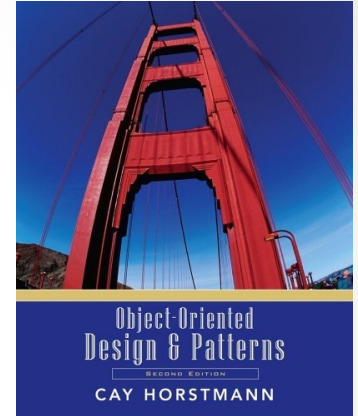


# Object-Oriented Design & Patterns

2<sup>nd</sup> edition

Cay S. Horstmann



## Chapter 2: The Object-Oriented Design Process

CPSC 2100

Software Design and Development

# Chapter Objective

In this chapter we introduce the main topic of the book; object-oriented design.

The chapter introduces a miniature version of a typical object-oriented design methodology that can guide you from the functional specifications of a program to its implementation.

You will see how to find and document classes and the relationships between them, using CRC cards and **UML Diagrams**.

# 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 the tasks to be performed.
  - Free from internal contradictions.
  - Readable both by domain experts and software developers.
  - Reviewable by diverse interested parties.
  - Testable against reality.
- Answers the questions:
  - *who will use the system?*
  - *What the system will do?*
  - *Where and when it will be used?*

# Analysis Phase

## Example:

- Writing a word-processing program.
  - Fonts, footnotes, multiple columns, document sections.
  - Interaction of those features.
- User manual.
  - Precisely worded to remove as much ambiguity as possible.
- **Use case**  $\Rightarrow$  **describe the behavior of the system.**  
is a description of a sequence of actions that yields a benefit for a user of a system.
- **What needs to be done, not how it should be done.**

# Analysis Phase

This phase has three steps:

1. Develop an analysis strategy
  - Model the current system (as-is system)
  - Formulate the new system (to-be system)
2. Gather the requirements (through interviews or questionnaires)
  - Develop a system concept
  - Create a business model to represent:
    - Business data
    - Business processes
3. Develop a system proposal

# Design Phase

- **Goals:**
  - Identify classes.
  - Identify behavior of classes.
  - Identify relationships among classes.

# Design Phase

- **Artifacts:**
  - Textual description of classes and most important responsibilities (key methods).
  - Diagrams of class relationships.
  - Diagrams of important usage scenarios.
  - State diagrams for objects whose behaviors is highly state-dependent.
- Typically, the design phase is more time-consuming than the actual programming.
- A good design greatly reduces the time required for implementation and testing.

# Design Phase

- **The design phase has four steps:**
  1. Develop a design strategy.
  2. Design architecture and interfaces.
  3. Develop databases and file specifications.
  4. Develop the program design to specify:
    - What programs to write
    - What each program will do

# Implementation Phase

**This phase has three steps:**

1. Construct the system
  - Build it (write the programming code)
  - Test it
2. Install system
  - Train the users
3. Support the system (maintenance)

# Object-Oriented Systems

## Analysis and Design

- **Use-case driven**
  - Use-cases define the behavior of a system
  - Each use-case focuses on one business process
- **Architecture centric:**
  - **Functional (external) view:** focuses on the user's perspective.
  - **Static (structural) view:** focuses on attributes, methods, classes & relationships.
  - **Dynamic (behavioral) view:** focuses on messages between classes and resulting behaviors.



# Object-Oriented Systems

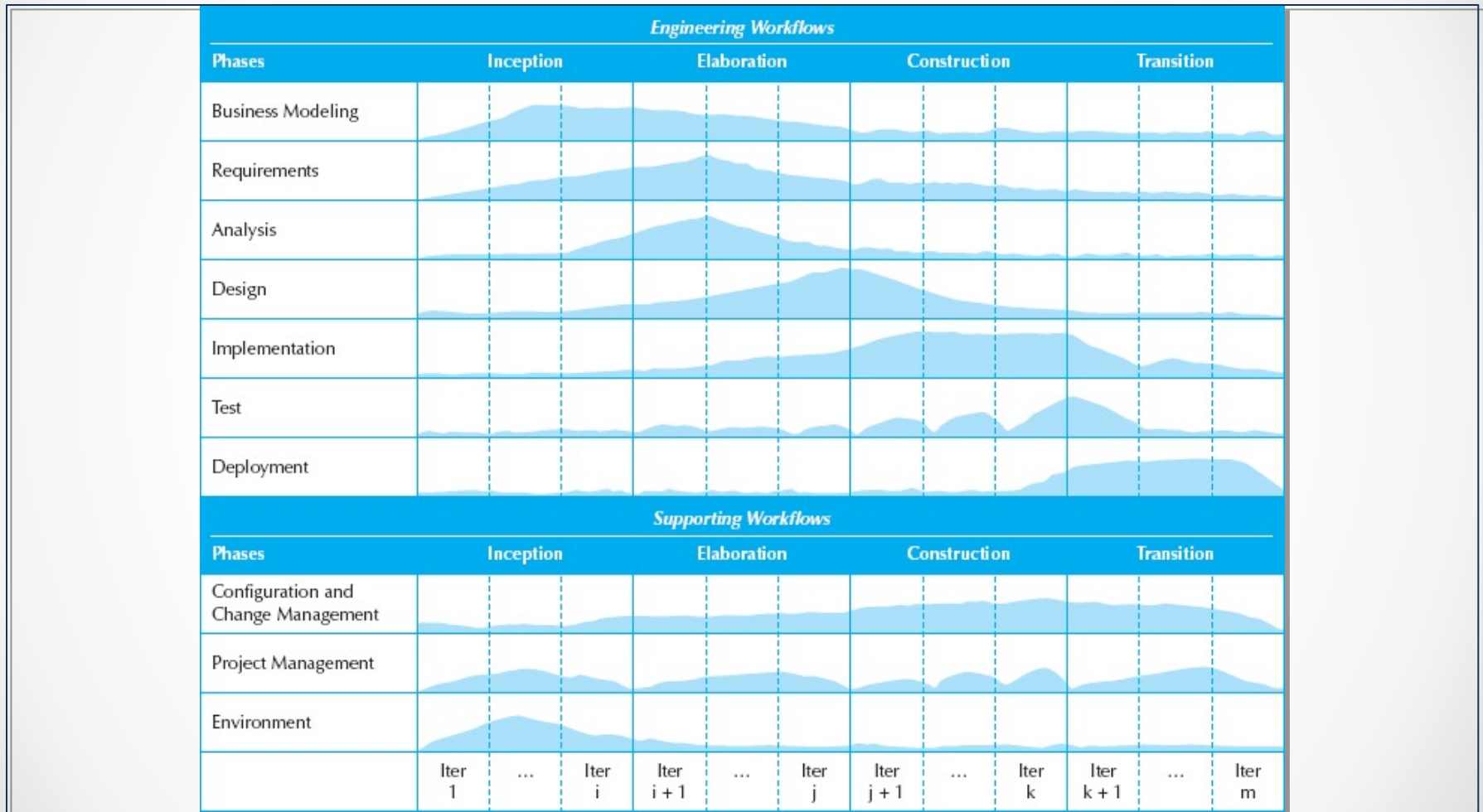
## Analysis and Design

- **Benefits of OOSAD**
  - Break a complex system into smaller, more manageable modules.
  - Work on modules individually.
  - See the system more realistically—as the users do.

# The Unified Process

- A specific methodology that maps out when and how to use the various UML techniques for object-oriented analysis and design.
- A two-dimensional process consisting of phases and workflows
  - Phases are time periods in development.
  - Workflows are the tasks that occur in each phase.
  - Activities in both phases & workflows will overlap.

# The Unified Process



# Unified Modeling Language (UML)

- Provides a common vocabulary of object-oriented terms and diagramming techniques rich enough to model any systems development project from analysis through implementation.
- Version 2.0 has 14 diagrams in 2 major groups:
  - Structure diagrams
  - Behavior diagrams

# UML Structure Diagrams

- Represent the data and static relationships in an information system
  - Class
  - Object
  - Package
  - Deployment
  - Component
  - Composite structure

# UML Behavior Diagrams

- Depict the dynamic relationships among the instances or objects that represent the business information system
  - Use-case diagrams
  - Activity
  - Sequence
  - Communication
  - Interaction overview
  - Timing
  - Behavior state machine
  - Protocol state machine,

# The Object and Class Concept

- **Object:** Three characteristic concepts:
  1. **State:** the condition of an object at a certain stage in its lifetime.
  2. **Behavior:** what an object does or capable of doing.
  3. **Identity:** what distinguishes it from other objects.

**Example:** mailbox in a voice mail system.

- **Class:** Collection of similar objects.
- **Methods & Messages**
  - **Methods:** the behavior of a class
  - **Messages:** information sent to an object to trigger a method (procedure call)

# Identifying Classes

- **Rule of thumb:** Look for *nouns* in problem description
  - Mailbox.
  - Message.
  - User.
  - Passcode.
  - Extension.
  - Menu.
- Focus on *concepts*, not implementation.
  - MessageQueue stores messages.
  - Don't worry yet how the queue is implemented.



# Categories of Classes

- **Tangible Things.**
  - Mailbox class, Message class, Document class, . . .
- **Agents.**
  - Scanner class.
- **Events and Transactions.**
  - MouseEvent class.
- **Users and Roles.**
  - Administrator class, Reviewer class.
- **Systems.**
  - MailSystem class.
- **System interfaces and devices.**
  - File class.
- **Foundational Classes.**
  - String class, Date class, . . .

# Identifying Responsibilities

- **Rule of thumb:** Look for *verbs* in problem description.

**Example:** 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?

# Relationships Between Classes

- 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

- One important design goal is to Minimize dependency (reduce coupling).

Example:

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

- Removes dependence on System, PrintStream.

```
public String getText() // can print anywhere
```

# Aggregation

- Object of a class contains objects of another class.
- **Example:** MessageQueue aggregates Messages.
  - MessageQueue has a Message.
- **Example:** Mailbox aggregates MessageQueue
  - Mailbox has a 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.
  - ForwardMessage is a Message
- **Example:** Greeting *does not* inherit from Message (Can't store greetings in mailbox).

# Exercise

A department store has a bridal registry. This registry keeps information about the customer, the products that the store carries, and the products each customer register for. Customers typically register for a large number of products and many customers register for the same products.

# Exercise

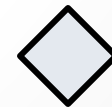
Whenever new patient are seen for the first time, they complete a patient information form that asks their name, address, phone number and insurance carrier, which are stored in the patient information file. Patients can be signed up with only one carrier, but they must be signed up to be seen by a doctor. Each time a patient visits the doctor, an insurance claim is sent to the carrier for payment. The claim must contain information about the visit, such as the date, purpose, and cost. It would be possible for a patient to submit two claims on the same day.

# Business Process Modeling

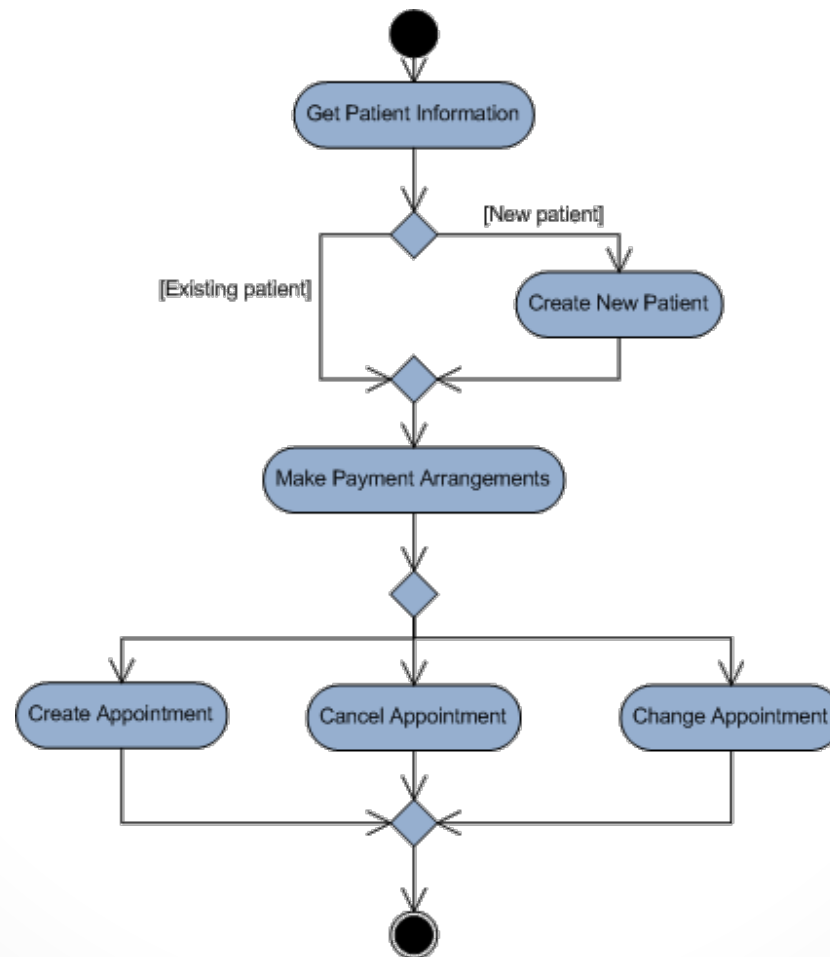
- Functional models: describe business processes and the interaction of an information system with its environments.
- Business process models describe the activities that collectively support a business process.
- A very powerful tool for communicating the analyst's current understanding of the requirements with the user.
- **Activity diagrams** are used to logically modeling the behavior in a business process and workflows.

# Activity Diagram Syntax

- Action or Activity
  - Represents action or set of actions
- Control Flow
  - Shows sequence of execution
- Initial Node
  - The beginning of a set of actions
- Final Node
  - Stops all flows in an activity
- Decision Node
  - Represents a test condition



# Sample Activity Diagram



# Guidelines for Activity Diagrams

1. Set the scope of the activity being modeled.
2. Identify the activities, control flows, and object flows that occur between the activities.
3. Identify any decisions that are part of the process being modeled.
4. Identify potential parallelism in the process.
5. Draw the activity diagram.

# Use Cases

- A use case illustrates the activities that are performed by users of a system.
- Describe basic functions of the system
  - What the user can do
  - How the system responds
- Use cases are building blocks for continued design activities.
- Use case name should be a verb-noun phrase (e.g., Make Appointment).



# Types of Use Cases

Purpose	Amount of information	
	Overview	Detail
	High-level <b>overview</b> of issues <b>essential</b> to understanding required functionality	<b>Detailed</b> description of issues <b>essential</b> to understanding required functionality
	High-level <b>overview</b> of a specific set of steps performed on the <b>real</b> system once implemented	<b>Detailed</b> description of a specific set of steps performed on the <b>real</b> system once implemented

# Use Case Elements: Relationships

- **Association**  
documents the communication between the use case and the actors that use the use case.
- **Extend**  
represents the extension of the functionality of the use case to incorporate optional behavior.
- **Include**  
shows the mandatory inclusion of another use case.
- **Generalization**  
allows use cases to support inheritance

# Use Case Elements: Flows

- Normal Flows  
include only those steps that normally are executed in a use case.
- Sub-Flows  
the normal flow of events decomposed to keep the normal flow of events as simple as possible.
- Alternate or Exceptional Flows  
flows that do happen but are not considered to be the norm.

- Sample use case description

Use-Case Name: Make appointment		ID: 2	Importance Level: High
Primary Actor: Patient		Use Case Type: Detail, essential	
Stakeholders and Interests: Patient - wants to make, change, or cancel an appointment Doctor - wants to ensure patient's needs are met in a timely manner			
Brief Description: This use case describes how we make an appointment as well as changing or canceling an appointment.			
Trigger: Patient calls and asks for a new appointment or asks to cancel or change an existing appointment.			
Type: External			
Relationships: Association: Patient Include: Make Payment Arrangements Extend: Create New Patient Generalization:			
Normal Flow of Events: 1. The Patient contacts the office regarding an appointment. 2. The Patient provides the Receptionist with his or her name and address. 3. The Receptionist validates that the Patient exists in the Patient database. 4. The Receptionist executes the Make Payment Arrangements use case. 5. The Receptionist asks Patient if he or she would like to make a new appointment, cancel an existing appointment, or change an existing appointment. If the patient wants to make a new appointment, the S-1: new appointment subflow is performed. If the patient wants to cancel an existing appointment, the S-2: cancel appointment subflow is performed. If the patient wants to change an existing appointment, the S-3: change appointment subflow is performed. 6. The Receptionist provides the results of the transaction to the Patient.			
Subflows: S-1: New Appointment 1. The Receptionist asks the Patient for possible appointment times. 2. The Receptionist matches the Patient's desired appointment times with available dates and times and schedules the new appointment. S-2: Cancel Appointment 1. The Receptionist asks the Patient for the old appointment time. 2. The Receptionist finds the current appointment in the appointment file and cancels it. S-3: Change Appointment 1. The Receptionist performs the S-2: cancel appointment subflow. 2. The Receptionist performs the S-1: new appointment subflow.			
Alternate/Exceptional Flows: 3a: The Receptionist executes the Create New Patient use case. S-1, 2a1: The Receptionist proposes some alternative appointment times based on what is available in the appointment schedule. S-1, 2a2: The Patient chooses one of the proposed times or decides not to make an appointment.			

# Use-Case Diagrams

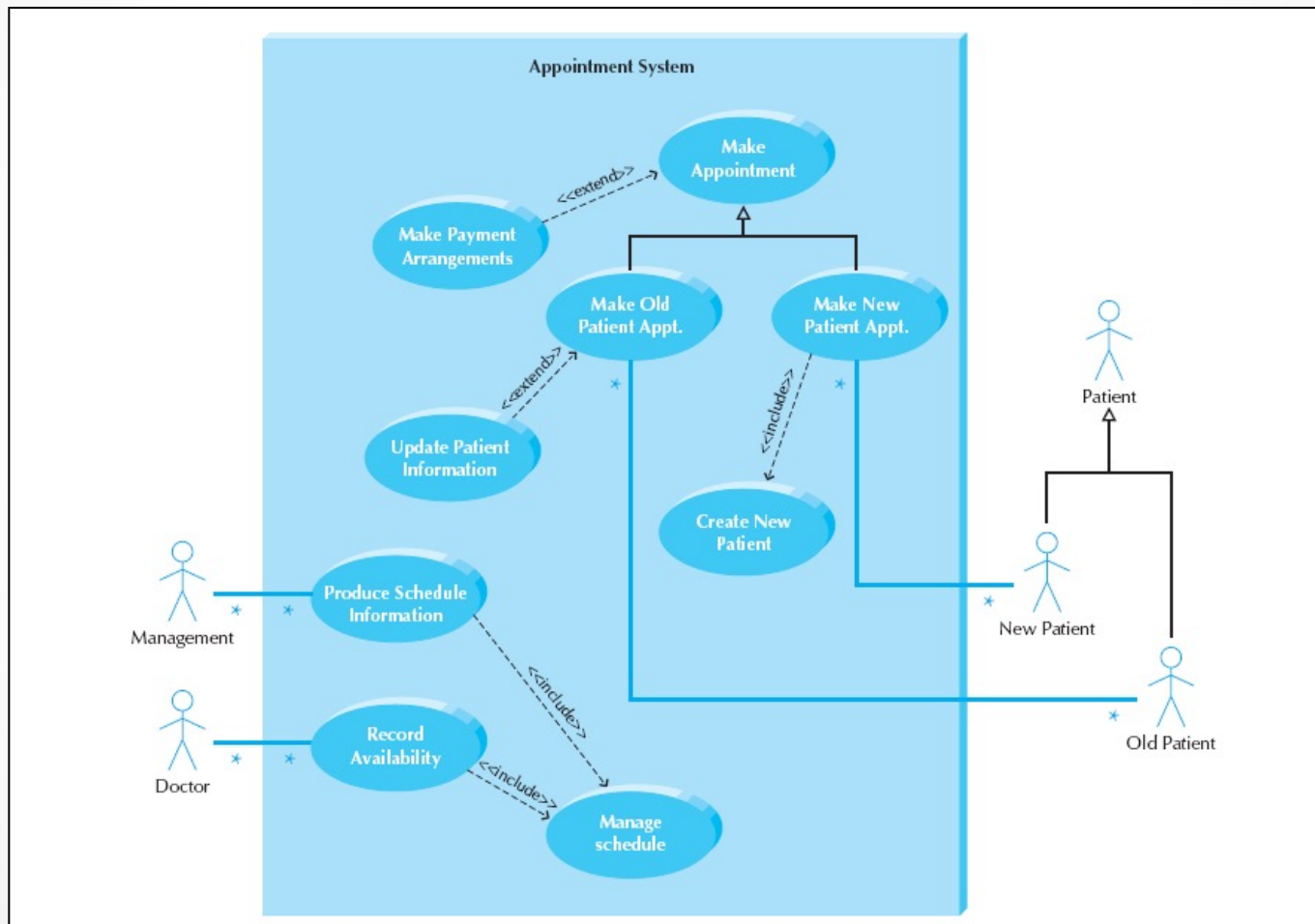
...

# Use Case Diagram Syntax

- Actor
  - person or system that derives benefit from and is external to the subject
- Use Case
  - Represents a major piece of system functionality
- Association Relationship
- Include Relationship
- Extend Relationship
- Generalization Relationship



# Sample Use Case



# Creating Use-Case Descriptions and Use-Case Diagrams

...



# Identify the Major Use Cases

1. Review the activity diagram.
2. Find the subject's boundaries.
3. Identify the primary actors and their goals.
4. Identify and write the overviews of the major use cases for the above.
5. Carefully review the current use cases. Revise as needed.

# Extend the Major Use Cases

6. Choose one of the use cases to expand.
7. Start filling in the details of the chosen use case.
8. Write the normal flow of events of the use case.
9. If the normal flow of events is too complex or long, decompose into sub flows.
10. List the possible alternate or exceptional flows.
11. For each alternate or exceptional flow, list how the actor and/or system should react.

# Confirm the Major Use Cases

12. Carefully review the current set of use cases. Revise as needed.
13. Start at the top again.

# Create the Use Case Diagram

1. Draw the subject boundary.
2. Place the use cases on the diagram.
3. Place the actors on the diagram.
4. Draw the associations.

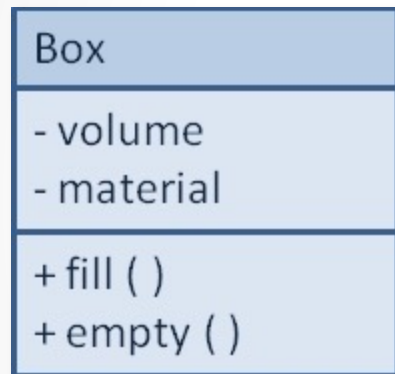
# Exercise

- A. Create an **activity diagram** and a **set of detail use case descriptions** for the process of buying glasses from the viewpoint of the patient, but do not bother to identify the flow of events within each use case. The first step is to see an eye doctor who will give you a prescription. Once you have a prescription, you go to a glasses store, where you select your frames and place the order for your glasses. Once the glasses have been made, you return to the store for a fitting and pay for the glasses.
- B. Draw a use case diagram for the process of buying glasses.

# Structural Model

- A formal way of representing the objects that are used and created by a business system
  - People
  - Places
  - Things
- **Main goal:** to discover the key data contained in the problem domain and to build a structural model of the objects.
- Typical structural models:
  - Class Responsibility Collaboration (CRC) cards
  - Class (and Object) diagrams

# Classes, Attributes, and Operations



- **Classes**

Templates for instances of people, places, or things

- **Attributes**

Properties that describe the state of an instance of a class (an object)

- **Operations**

Actions or functions that a class can perform

# Relationships

- Describe how classes relate to one another
- Three basic types in UML
  1. **Generalization**  
Enables inheritance of attributes and operations
  2. **Aggregation**  
Relates parts to wholes
  3. **Association**  
Miscellaneous relationships between classes



# Responsibilities and Collaborations

- Responsibilities
  - Knowing: things that an instance of a class must be capable of knowing.
  - Doing: things that an instance of a class must be capable of doing.
- Collaboration
  - Objects working together to service a request.

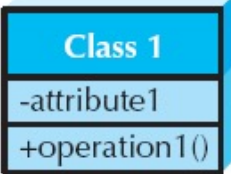
# Front-Side of a CRC Card

<b>Class Name:</b> Patient	<b>ID:</b> 3	<b>Type:</b> Concrete, Domain
<b>Description:</b> An individual that needs to receive or has received medical attention		<b>Associated Use Cases:</b> 2
<b>Responsibilities</b> Make appointment _____ Calculate last visit _____ Change status _____ Provide medical history _____ _____ _____ _____		<b>Collaborators</b> Appointment _____ _____ _____ Medical history _____ _____ _____ _____

# Back-Side of a CRC Card

<b>Attributes:</b>	
Amount (double)	
Insurance carrier (text)	
<b>Relationships:</b>	
Generalization (a-kind-of):	Person
Aggregation (has-parts):	Medical History
Other Associations:	Appointment

# Elements of a Class Diagram

<p><b>A class:</b></p> <ul style="list-style-type: none"><li>• Represents a kind of person, place, or thing about which the system will need to capture and store information.</li><li>• Has a name typed in bold and centered in its top compartment.</li><li>• Has a list of attributes in its middle compartment.</li><li>• Has a list of operations in its bottom compartment.</li><li>• Does not explicitly show operations that are available to all classes.</li></ul>	 <p>A UML class diagram for 'Class 1'. It is a rectangle divided into three horizontal compartments. The top compartment is blue and contains the text 'Class 1' in white. The middle compartment is light blue and contains the text '-attribute1'. The bottom compartment is light blue and contains the text '+operation1()'.</p>
<p><b>An attribute:</b></p> <ul style="list-style-type: none"><li>• Represents properties that describe the state of an object.</li><li>• Can be derived from other attributes, shown by placing a slash before the attribute's name.</li></ul>	<p>attribute name /derived attribute name</p>
<p><b>An operation:</b></p> <ul style="list-style-type: none"><li>• Represents the actions or functions that a class can perform.</li><li>• Can be classified as a constructor, query, or update operation.</li><li>• Includes parentheses that may contain parameters or information needed to perform the operation.</li></ul>	<p>operation name ()</p>

# Attribute Visibility

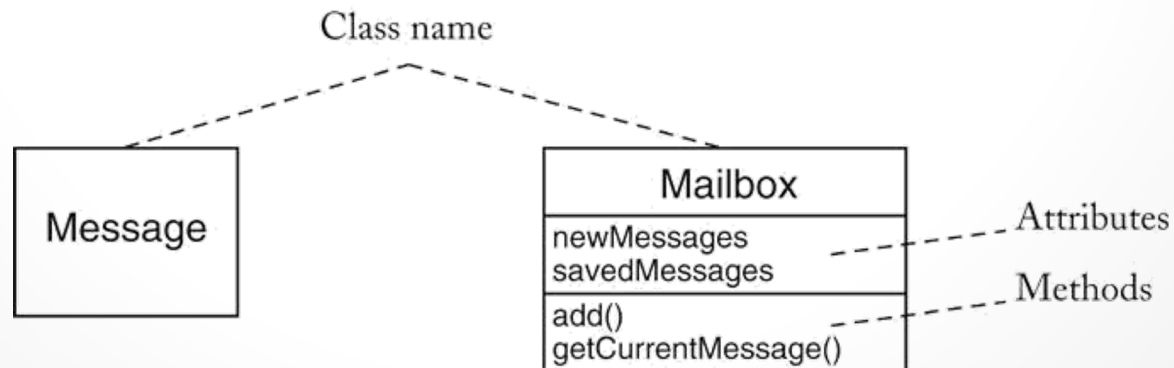
- Attribute visibility can be specified in the class diagram
  - Public attributes (+) are visible to all classes
  - Private attributes (-) are visible only to an instance of the class in which they are defined
  - Protected attributes (#) are like private attributes, but are also visible to descendant classes
- Visibility helps restrict access to the attributes and thus ensure consistency and integrity

# Operations


- Constructor
  - Creates object
- Query
  - Makes information about state available
- Update
  - Changes values of some or all attributes


# Class Diagrams

- Rectangle with class name.
- **Optional compartments**
  - Attributes
    - text
    - text : String
  - Methods
    - getMessage()
    - getMessage(index : int) : Message



# Class Relationships

Dependency 

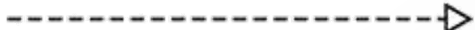
Aggregation 

Inheritance 

Composition 

Association 

Directed Association 

Interface Type Implementation 



# Multiplicities

“has” relationship (Aggregation)

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



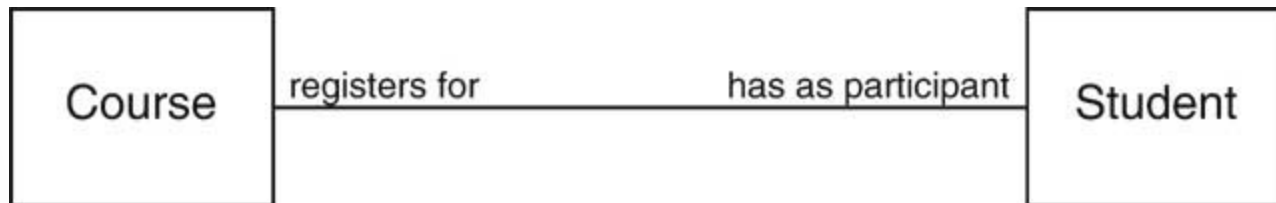
# Composition

- 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 – implementation specific.
- More general association relationship.
- Association can have **roles** .

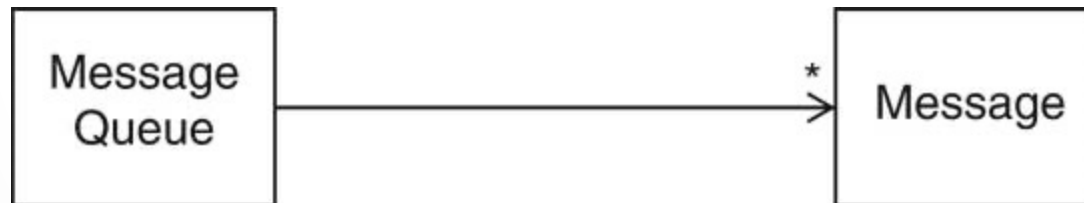


- Some associations are **bidirectional**. Can navigate from either class to the other.

# More on Association

- Some associations are directed.

**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»

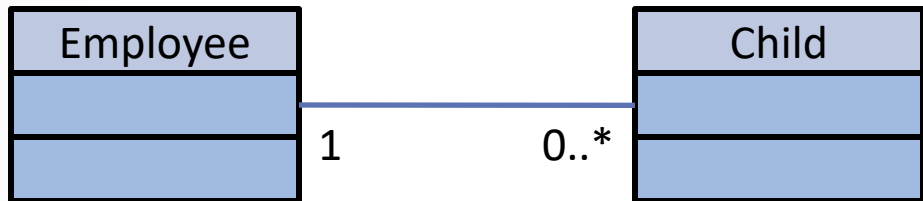


# Multiplicities



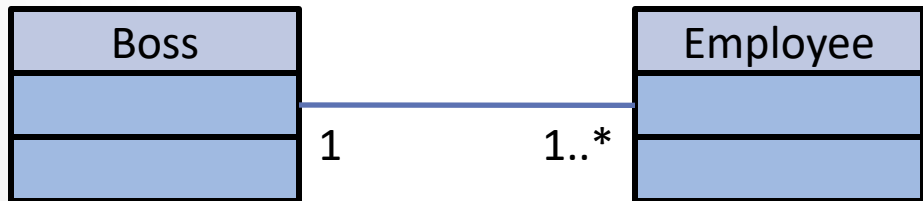
## Exactly one:

A department has one and only one boss



## Zero or more:

An employee has zero to many children



## One or more:

A boss is responsible for one or more employees



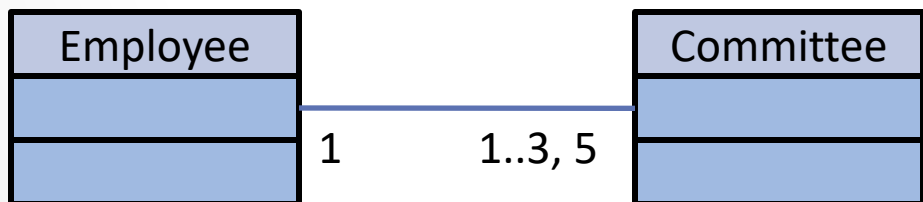
**Zero or one:**

An employee can be married to 0 or 1 spouse



**Specified range:**

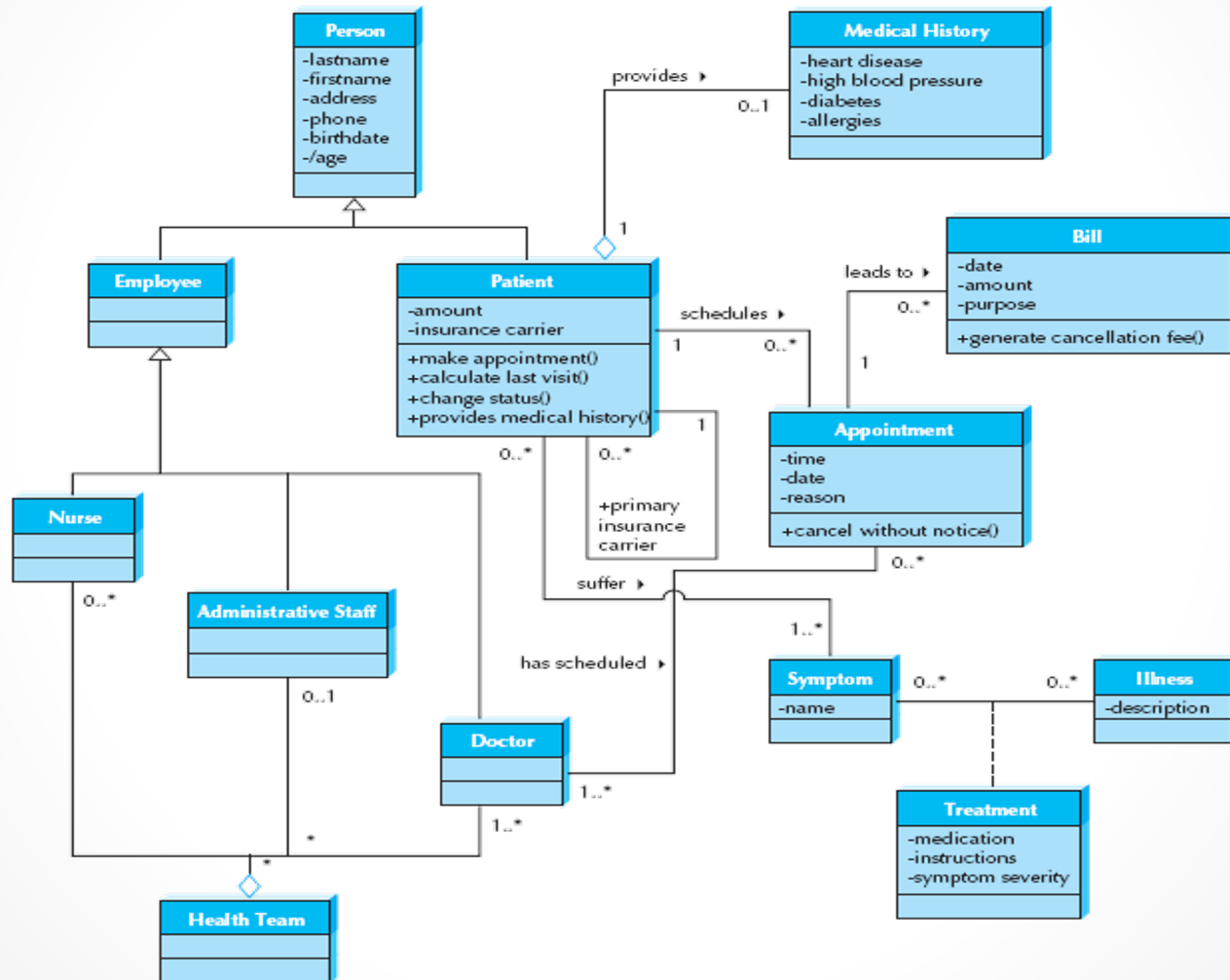
An employee can take 2 to 4 vacations each year



**Multiple disjoint ranges:**

An employee can be in 1 to 3 or 5 committees

# Sample Class Diagram

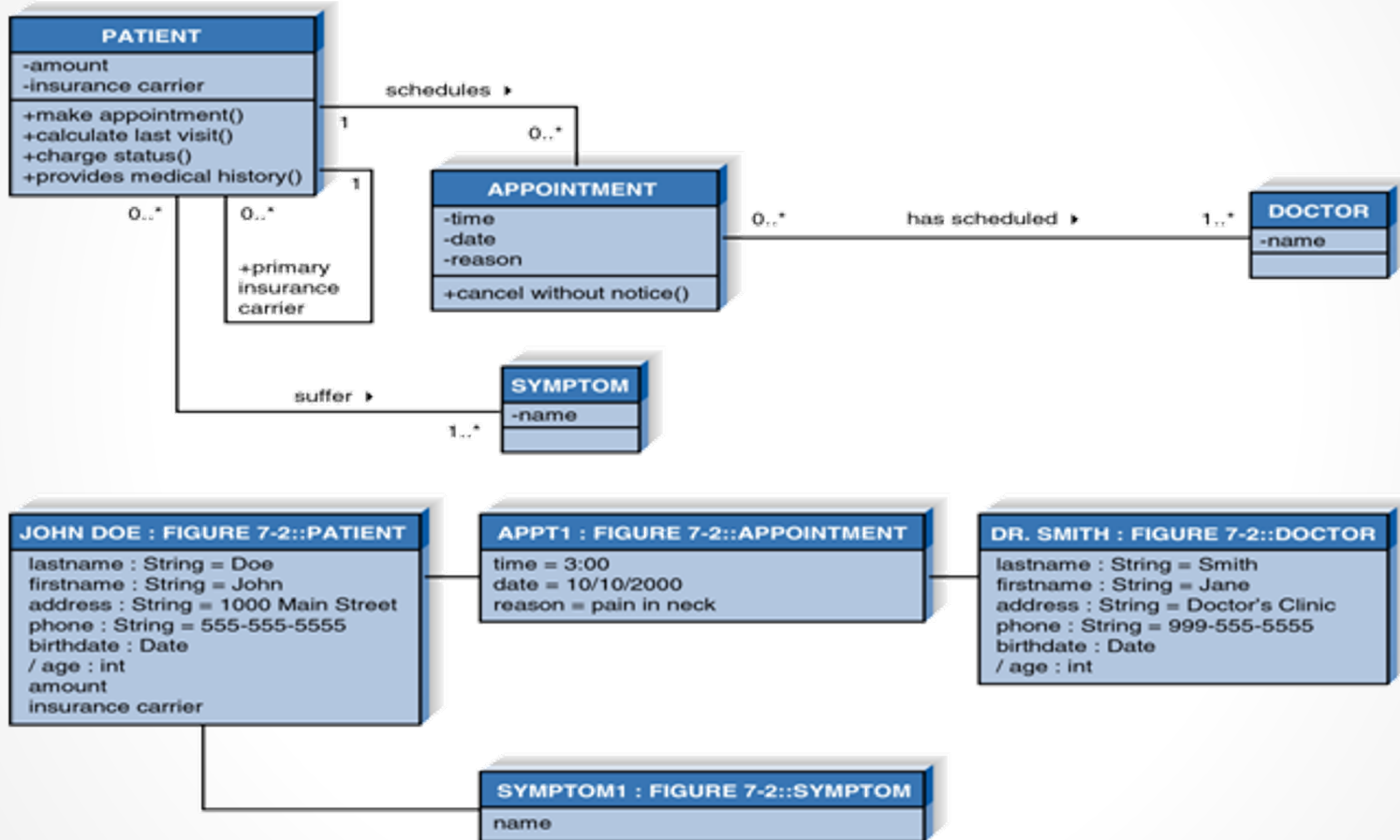




# Tips

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

# Object Diagram



# Exercise

- Draw a class diagram for the following classes. Consider that the entities represent a system for a patient billing system. Include only the attributes that would be appropriate for this context.
- Patient:
  - (age, name, address, insurance carrier)
- Insurance carrier:
  - (name, number of patients on plan, address, phone)
- Doctor:
  - (specialty, provider identification number, phone, name).
- Create an Object diagram based on the class diagram you drew before.

# Exercise

- A university is interested in designing a system that will track its researchers. Information of interest includes: researcher name, title, position; university name, location, enrollment; and research interests. Researchers are associated with one institution, and each researcher has several research interests.
- Draw a class diagram.
- Create an object diagram based on the class diagram.
- Create a CRC card for the Researcher class.

# Behavioral Modeling

- Systems have static and dynamic characteristics
  - Structural models describe the static aspects of the system.
  - Behavioral models describe the dynamics and interactions of the system and its components.
- Behavioral models describe how the classes described in the structural models interact in support of the *use cases*.


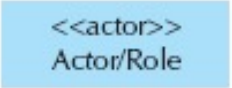
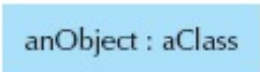


# Interaction Diagram Components

- **Objects**  
an instantiation of a class.
- **Operations**  
the behaviors of an instance of a class.
- **Messages**  
information sent to objects to tell them to execute one of their behaviors.

# Sequence Diagrams

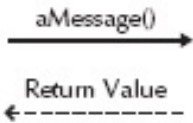



- Illustrate the objects that participate in a use-case.
- Show the messages that pass between objects for a particular use-case.

# Sequence Diagram Syntax

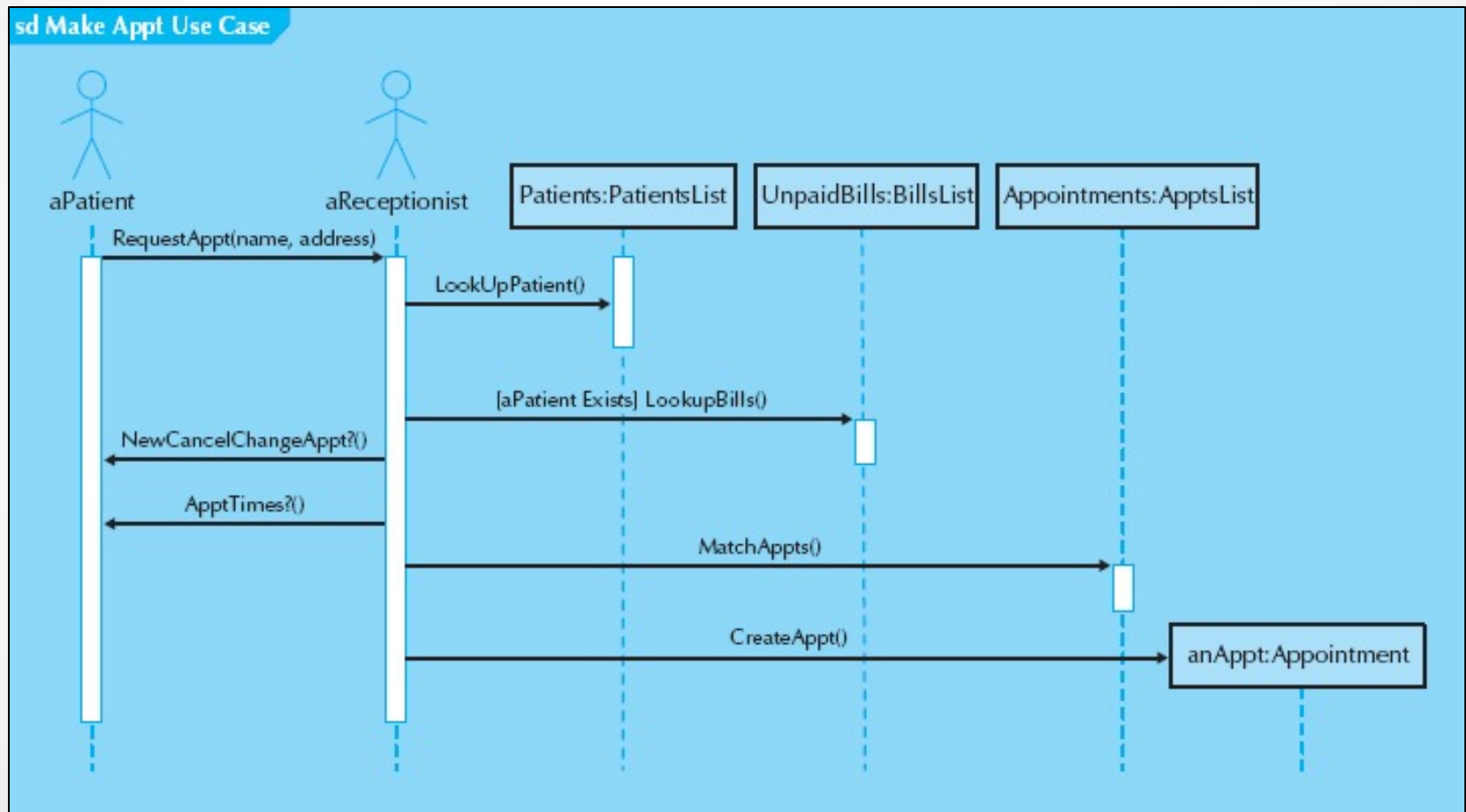
<p><b>An actor:</b></p> <ul style="list-style-type: none"> <li>■ Is a person or system that derives benefit from and is external to the system.</li> <li>■ Participates in a sequence by sending and/or receiving messages.</li> <li>■ Is placed across the top of the diagram.</li> <li>■ Is depicted either as a stick figure (default) or, if a nonhuman actor is involved, as a rectangle with &lt;&lt;actor&gt;&gt; in it (alternative).</li> </ul>	 <p>anActor</p> 
<p><b>An object:</b></p> <ul style="list-style-type: none"> <li>■ Participates in a sequence by sending and/or receiving messages.</li> <li>■ Is placed across the top of the diagram.</li> </ul>	
<p><b>A lifeline:</b></p> <ul style="list-style-type: none"> <li>■ Denotes the life of an object during a sequence.</li> <li>■ Contains an X at the point at which the class no longer interacts.</li> </ul>	
<p><b>An execution occurrence:</b></p> <ul style="list-style-type: none"> <li>■ Is a long narrow rectangle placed atop a lifeline.</li> <li>■ Denotes when an object is sending or receiving messages.</li> </ul>	



# More Sequence Diagram Syntax

<p><b>A message:</b></p> <ul style="list-style-type: none"> <li>■ Conveys information from one object to another one.</li> <li>■ A operation call is labeled with the message being sent and a solid arrow, whereas a return is labeled with the value being returned and shown as a dashed arrow.</li> </ul>	 <p>The diagram shows a solid arrow pointing right labeled 'aMessage()' and a dashed arrow pointing left labeled 'Return Value'.</p>
<p><b>A guard condition:</b></p> <ul style="list-style-type: none"> <li>■ Represents a test that must be met for the message to be sent.</li> </ul>	 <p>The diagram shows a solid arrow pointing right labeled '[aGuardCondition]: aMessage()'.</p>
<p><b>For object destruction:</b></p> <ul style="list-style-type: none"> <li>■ An X is placed at the end of an object's lifeline to show that it is going out of existence.</li> </ul>	 <p>The diagram shows a single 'X' character.</p>
<p><b>A frame:</b></p> <ul style="list-style-type: none"> <li>■ Indicates the context of the sequence diagram.</li> </ul>	 <p>The diagram shows a light blue rectangle with a darker blue tab on the top left labeled 'Context'.</p>

# Sample Sequence Diagram









# Steps to build Sequence Diagrams

1. Set the context.
2. Identify which objects will participate.
3. Set the lifeline for each object.
4. Lay out the messages from top to bottom of the diagram based on the order in which they are sent.
5. Add execution occurrence to each object's lifeline.
6. Validate the sequence diagram.

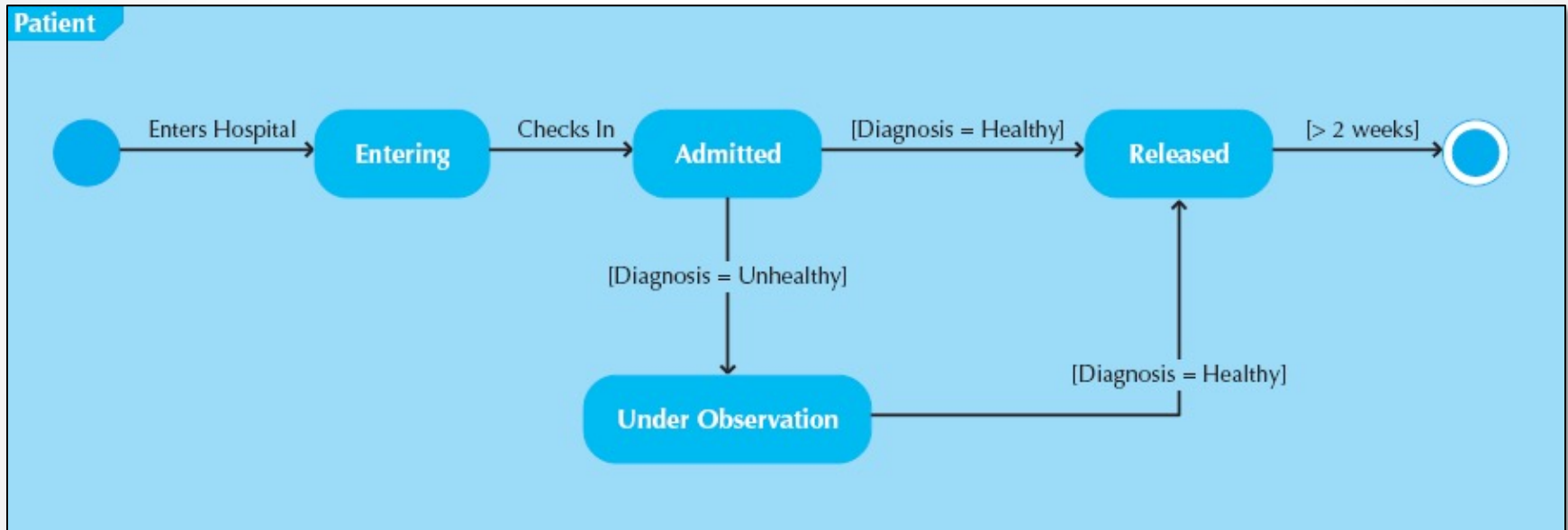
# Components of State Machines

- States  
values of an object's attributes at a point in time
- Events  
change the values of the object's attributes
- Transitions  
movement of an object from one state to another
- Actions  
atomic, non-decomposable processes
- Activities  
non-atomic, decomposable processes

# State Machine Syntax

<b>A state:</b> <ul style="list-style-type: none"> <li>■ Is shown as a rectangle with rounded corners.</li> <li>■ Has a name that represents the state of an object.</li> </ul>	
<b>An initial state:</b> <ul style="list-style-type: none"> <li>■ Is shown as a small, filled-in circle.</li> <li>■ Represents the point at which an object begins to exist.</li> </ul>	
<b>A final state:</b> <ul style="list-style-type: none"> <li>■ Is shown as a circle surrounding a small, filled-in circle (bull's-eye).</li> <li>■ Represents the completion of activity.</li> </ul>	
<b>An event:</b> <ul style="list-style-type: none"> <li>■ Is a noteworthy occurrence that triggers a change in state.</li> <li>■ Can be a designated condition becoming true, the receipt of an explicit signal from one object to another, or the passage of a designated period of time.</li> <li>■ Is used to label a transition.</li> </ul>	
<b>A transition:</b> <ul style="list-style-type: none"> <li>■ Indicates that an object in the first state will enter the second state.</li> <li>■ Is triggered by the occurrence of the event labeling the transition.</li> <li>■ Is shown as a solid arrow from one state to another, labeled by the event name.</li> </ul>	
<b>A frame:</b> <ul style="list-style-type: none"> <li>■ Indicates the context of the behavioral state machine.</li> </ul>	

# State Machine Example



# Steps to Build a State Machine

1. Set the context.
2. Identify the initial, final, and stable states of the object.
3. Determine the order in which the object will pass through the stable states.
4. Identify the events, actions, and guard conditions associated with the transitions.
5. Validate the behavioral state machine.

# Behavioral State Machines

- A dynamic model that shows the different states through which a single object passes during its life in response to events, along with its responses and actions
- Typically not used for all objects
  - Just for complex ones



# Exercise

- Create a **sequence diagram** for the following scenario description:

When members join the health club, they pay a fee for a certain length of time. The club wants to mail out remainder letters to members asking them to renew their memberships one month before their memberships expire. About half of the members do not renew their memberships. These members are sent follow-up surveys to complete asking why they decided not to renew so that the club can learn how to increase retention. If the member did not renew because of cost, a special discount is offered to that customer. Typically, 25 percent of accounts are reactivated because of this offer.

# Unified Modeling Language (UML)

- UML is a *modeling language* for object oriented system analysis, design, and deployment. UML is not a product, nor is a process or methodology.
- UML support multiple views of same system, with varying degrees of detail or generalization as needed:
  1. **Owner's view:** what the owner wants, or the conceptual view of the system.
  2. **Architect's view:** how the architect conceives the solution, of the logical view of the system.
  3. **Builder's view:** the blueprints for building the product, or the physical view of the system.

# Use Cases

- **Analysis technique.**
- Each *use case* focuses on a **specific scenario**.
- Use case = sequence of ***actions***.
- **Action** = interaction between ***actor*** and computer system.
  - Leave a message.
  - Retrieve a message.
- Each action yields a ***result***.
- Each result has a ***value*** to one of the actors.
- Use ***variations*** for exceptional situations.
  - Message queue is full.
  - Wrong password.

# Sample Use Case

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 recipient's 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 cards describe Classes, Responsibilities, Collaborators (dependent classes).
- 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 Card Example

Mailbox	
<i>manage passcode</i>	MessageQueue
<i>manage greeting</i>	
<i>manage new and saved messages</i>	

- 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.



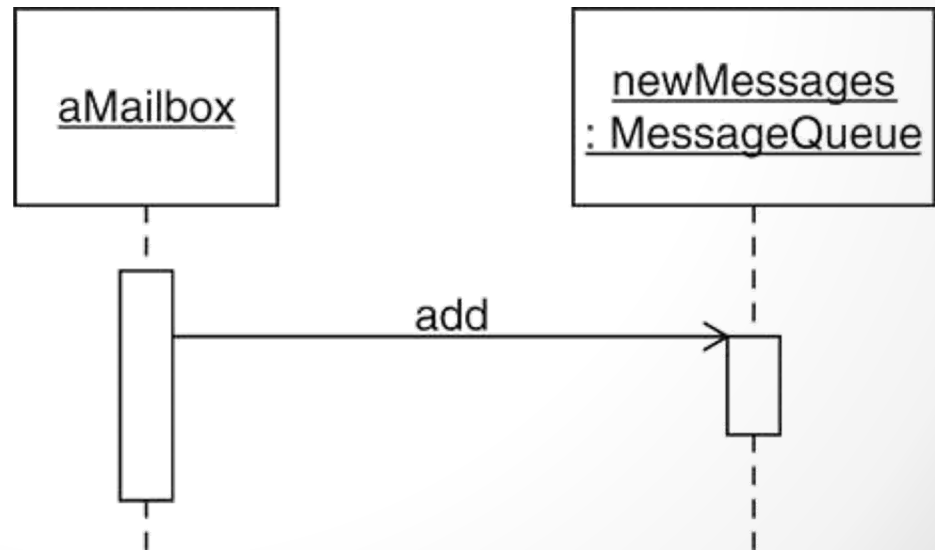
# Walkthrough Example

MailSystem	
<i>manage mailboxes</i>	Mailbox

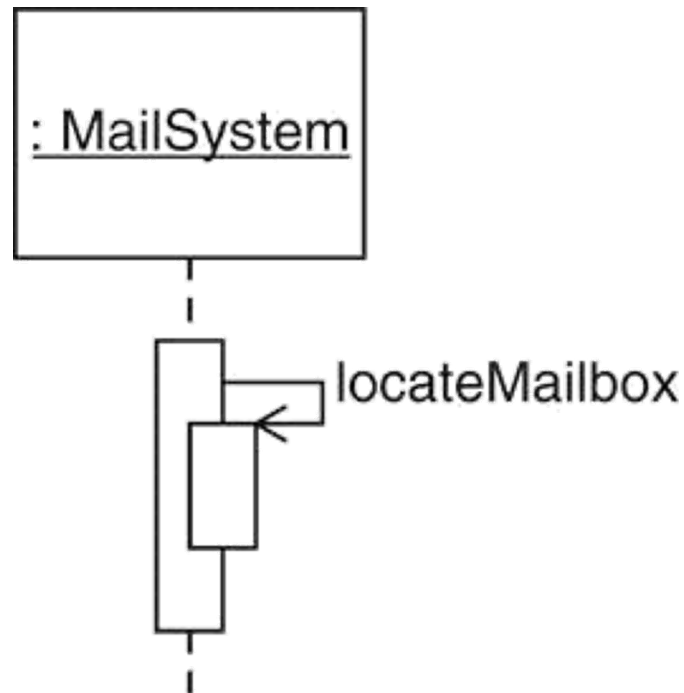
# Sequence Diagrams

## Object Diagrams

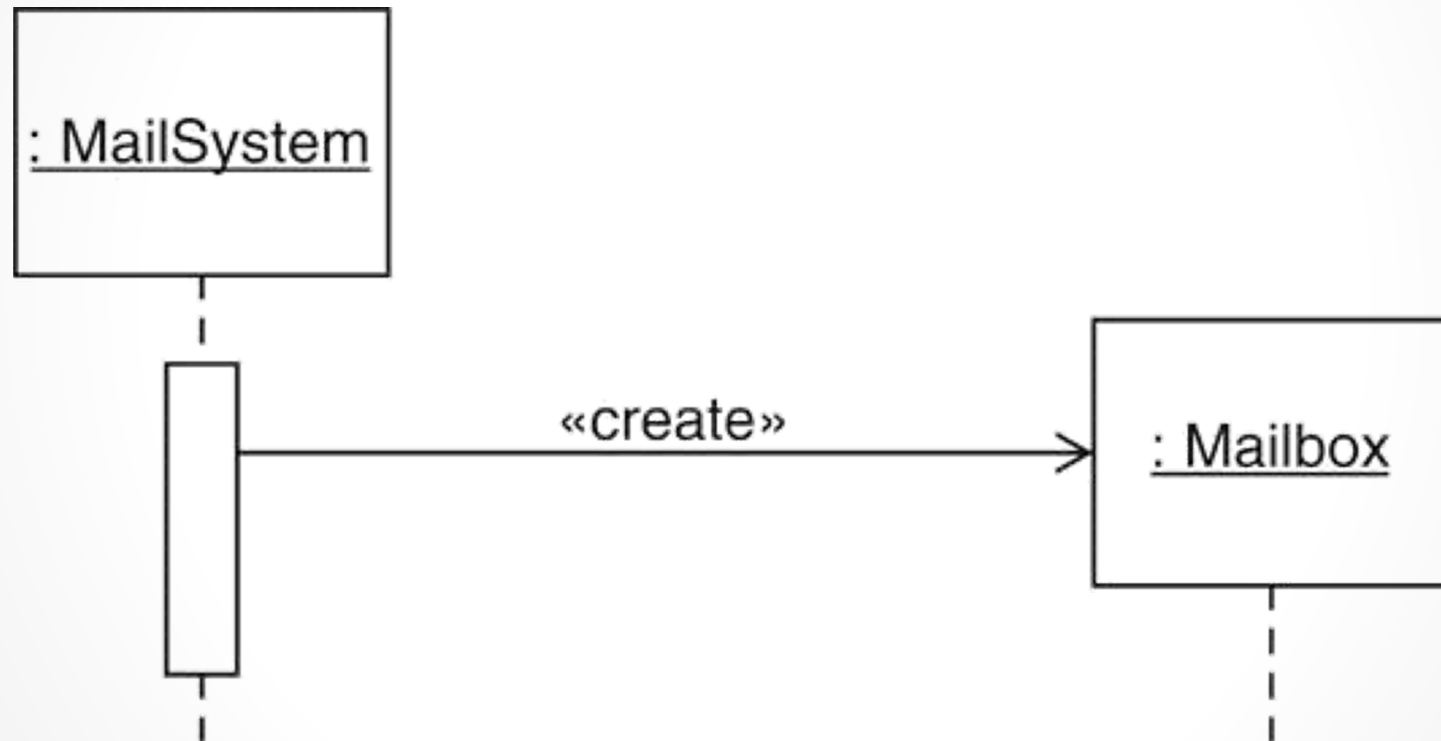
- Each diagram shows dynamics of scenario.
- Interactions between objects.
- Object diagram: class name underlined
  - objectName : ClassName (full description).
  - objectName
  - : ClassName
- Lifelines.
- Activation bars.



# Sequence Diagram - Self call

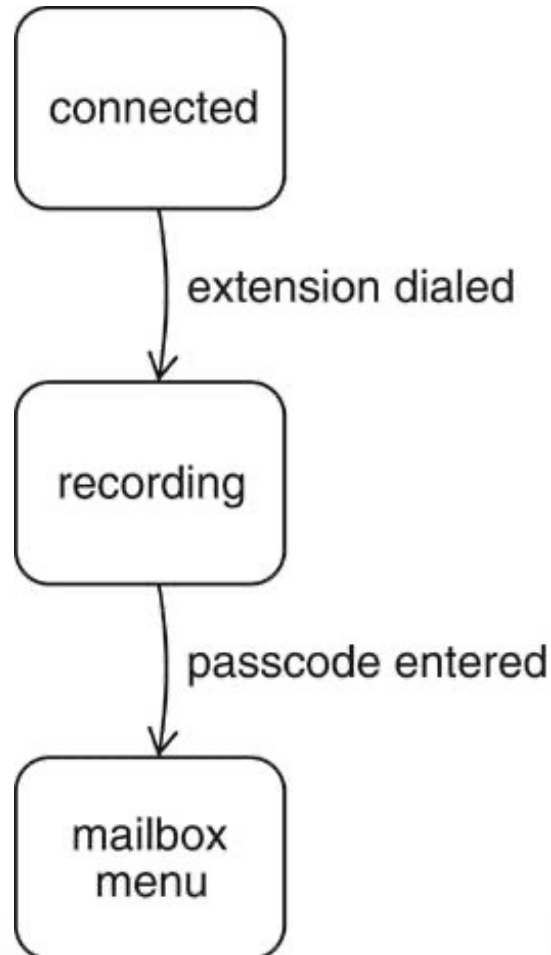


# Object Construction



# State Diagrams

- Use for classes whose objects have interesting states



# Using javadoc for 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

## A Voice Mail System

- Represent voice as text entered from the keyboard.
- 1 2 ... 0 # on a single line means key.
- H on a single line means "hang up".
- All other inputs mean voice.

# Use Case

## Reach an Extension

1. User 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.



# 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 #
3. System plays mailbox menu:
  - Enter 1 to retrieve your messages.
  - Enter 2 to change your passcode.
  - Enter 3 to change your greeting.

# Use Case

## Retrieve Messages

1. Mailbox owner carries out **Log in**.
2. Mailbox owner selects "retrieve messages" menu option.
3. System plays message menu:
  - Press 1 to listen to the current message
  - Press 2 to delete the current message
  - Press 3 to save the current message
  - Press 4 to return to the mailbox menu
4. Mailbox owner selects "listen to current message".
5. System plays current new message, or, if no more new messages, current old message.
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	
<i>keep new and saved messages</i>	MessageQueue

# Initial CRC Cards: MessageQueue

MessageQueue
<i>add and remove messages in</i>
<i>FIFO order</i>

# Initial CRC Cards: MailSystem

MailSystem	
<i>manage mailboxes</i>	Mailbox

# Telephone

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

# Telephone

Telephone
<i>take user input from touchpad,</i>
<i>microphone, hangup</i>
<i>speak output</i>

# Connection

- With whom does Telephone communicate, 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	
<i>get input from telephone</i>	Telephone
<i>carry out user commands</i>	MailSystem
<i>keep track of state</i>	



# Use Case

## Leave 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.

# Use Case

## Leave 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	
<i>take user input from touchpad,</i>	Connection
<i>microphone, hangup</i>	
<i>speak output</i>	

# Result of Use Case Analysis

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

# Result of Use Case Analysis

Mailbox	
<i>keep new and saved messages</i>	MessageQueue
<i>manage greeting</i>	

# Result of Use Case Analysis

Message
<i>manage message contents</i>

# 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.

# Use Case

## Retrieve messages

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



# Use Case

## Retrieve messages

- 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	
<i>keep new and saved messages</i>	MessageQueue
<i>manage greeting</i>	
<i>manage passcode</i>	
<i>retrieve, save, delete messages</i>	

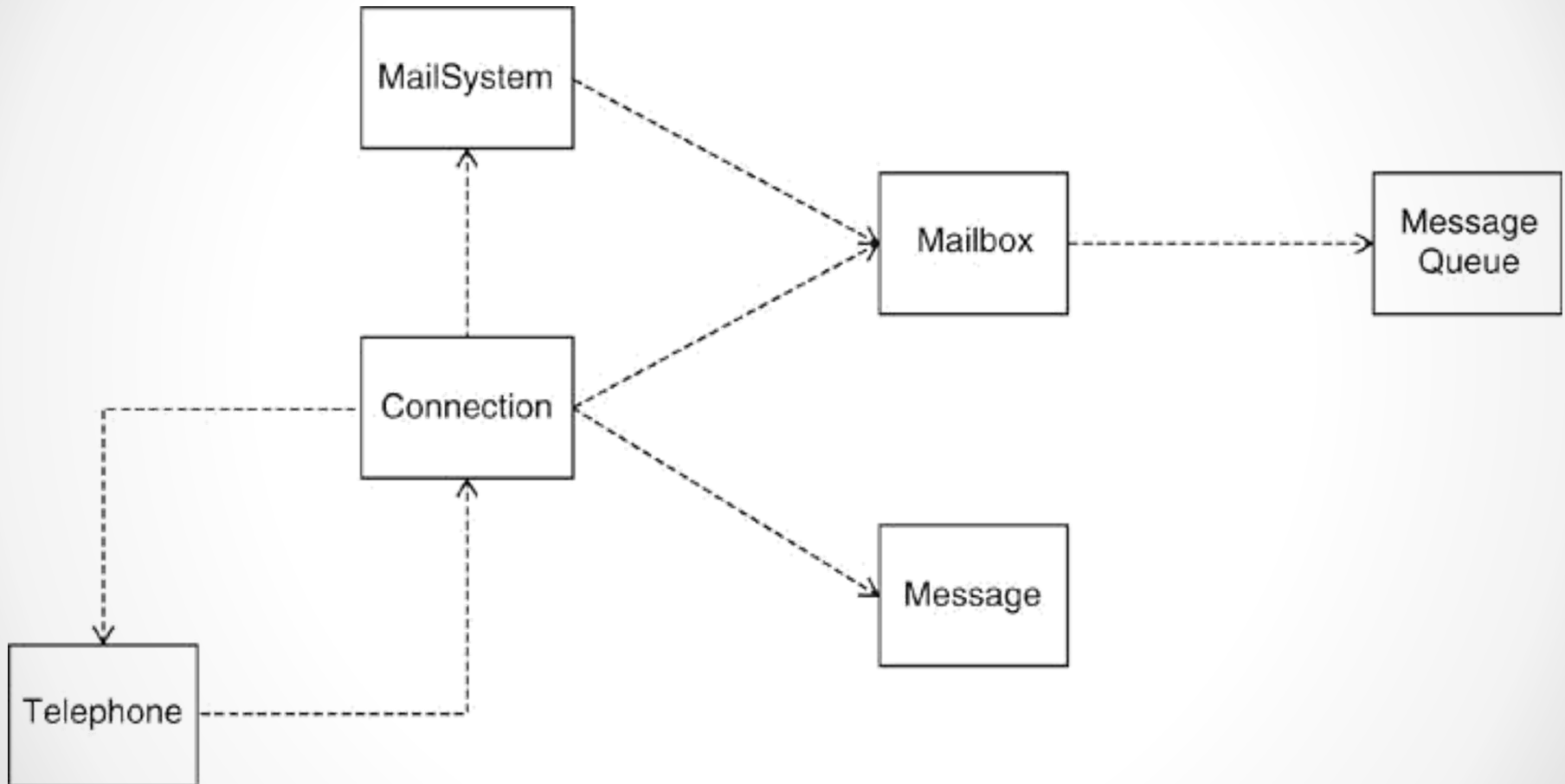
# CRC Summary

- One card per class.
- Responsibilities at high level.
- Use scenario walkthroughs to fill in cards.
- Usually, the first design isn't perfect.

# UML Class Diagram - 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.

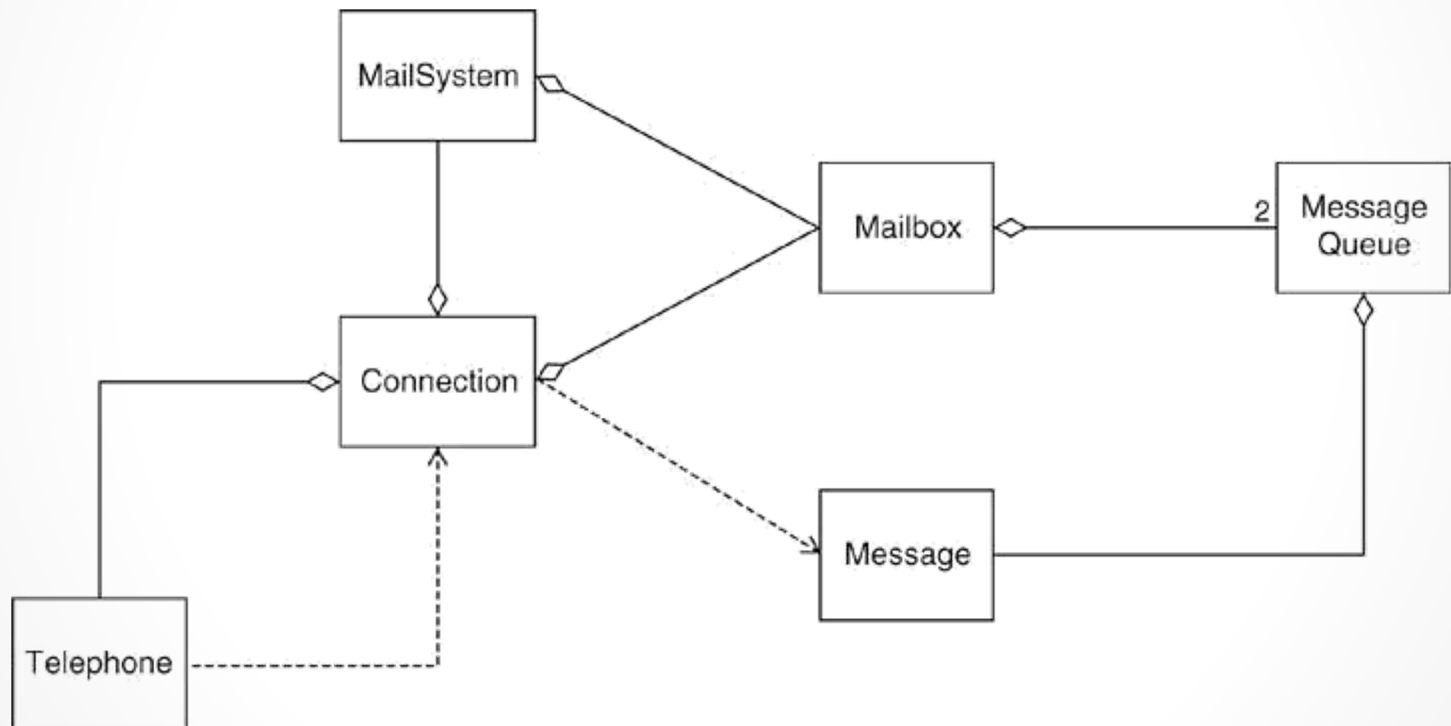
# Dependency Relationships



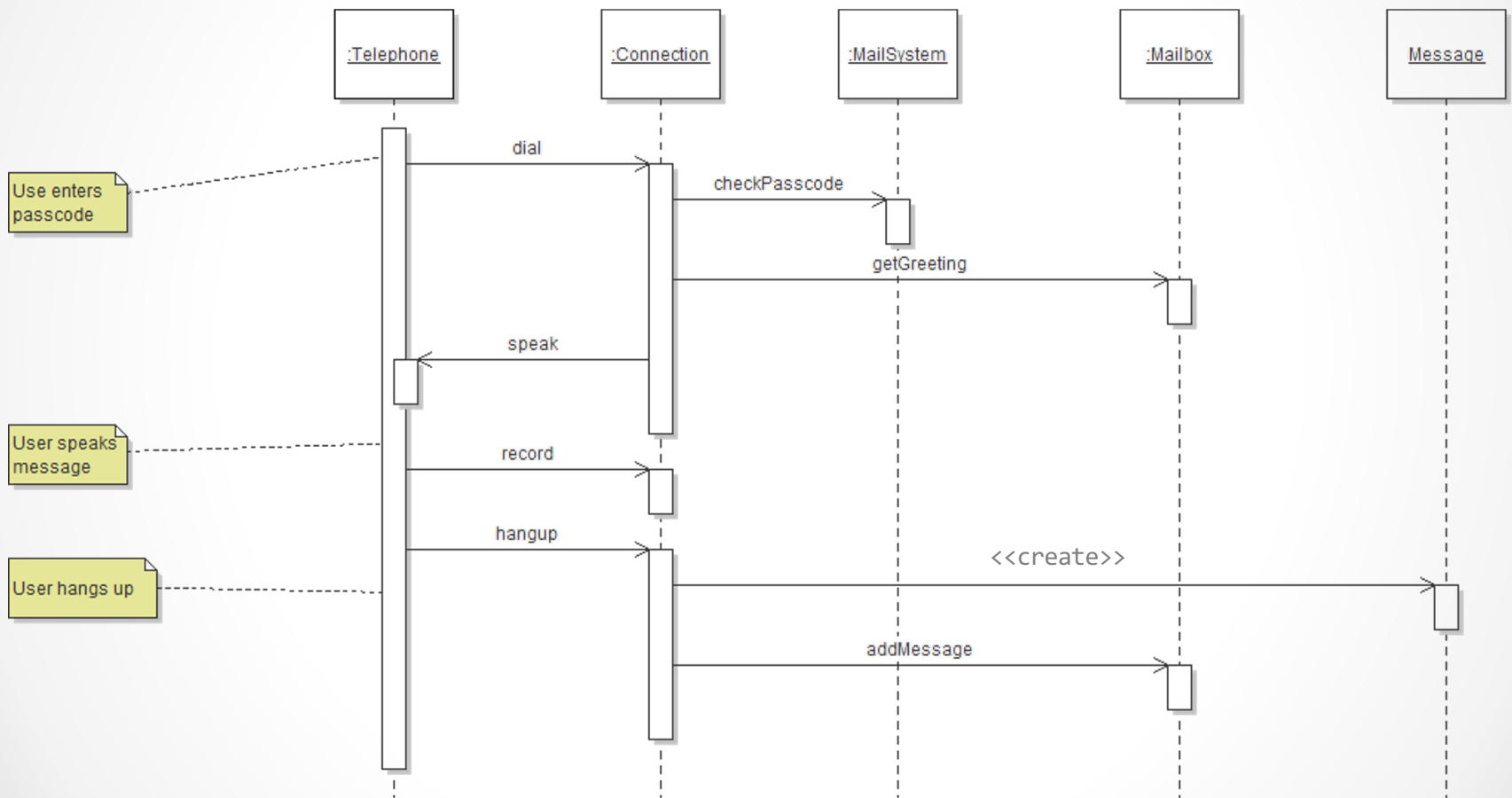
# Aggregation Relationships

- 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 - Voice Mail System



# Sequence Diagram - Use Case: Leave a message

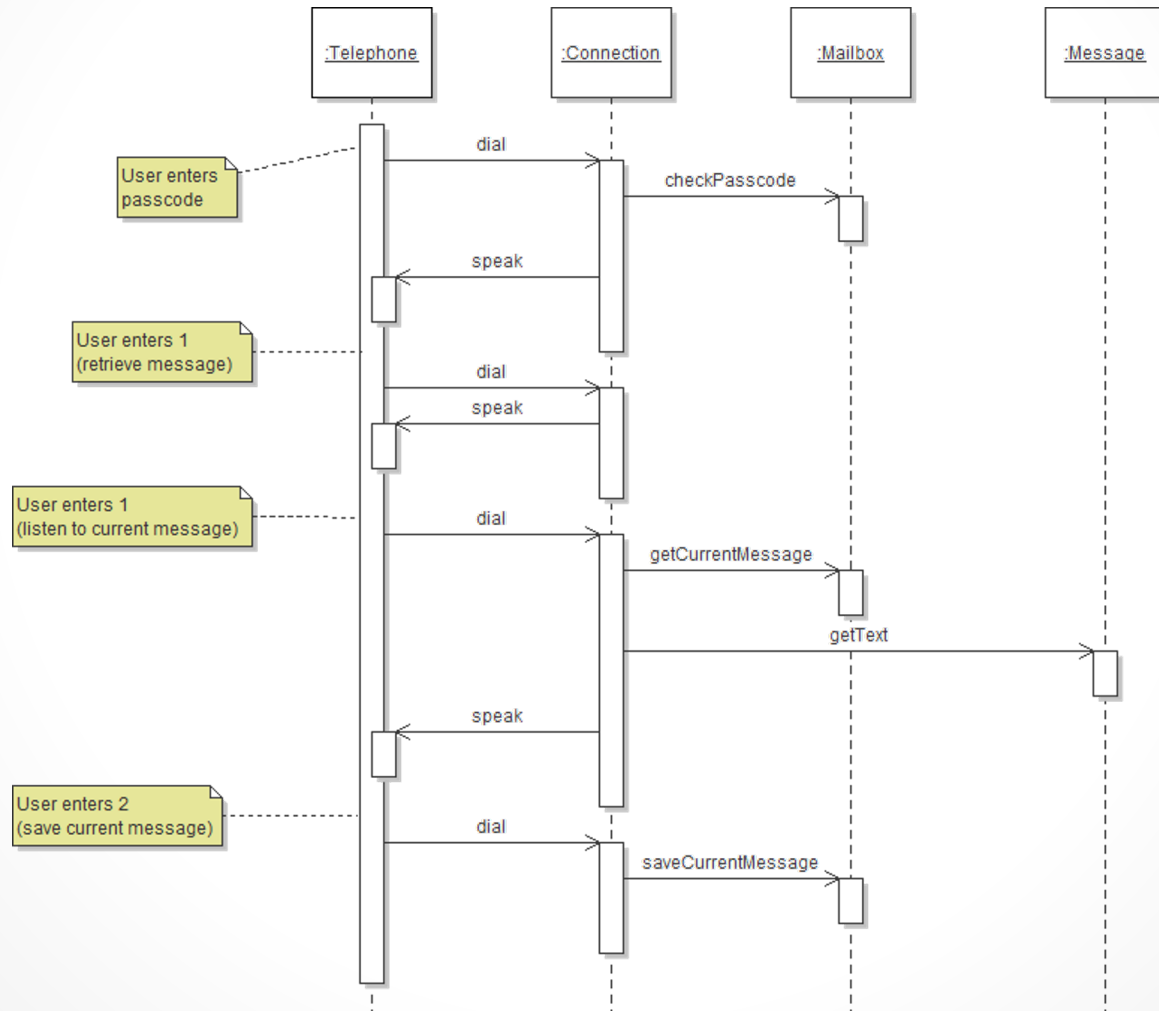




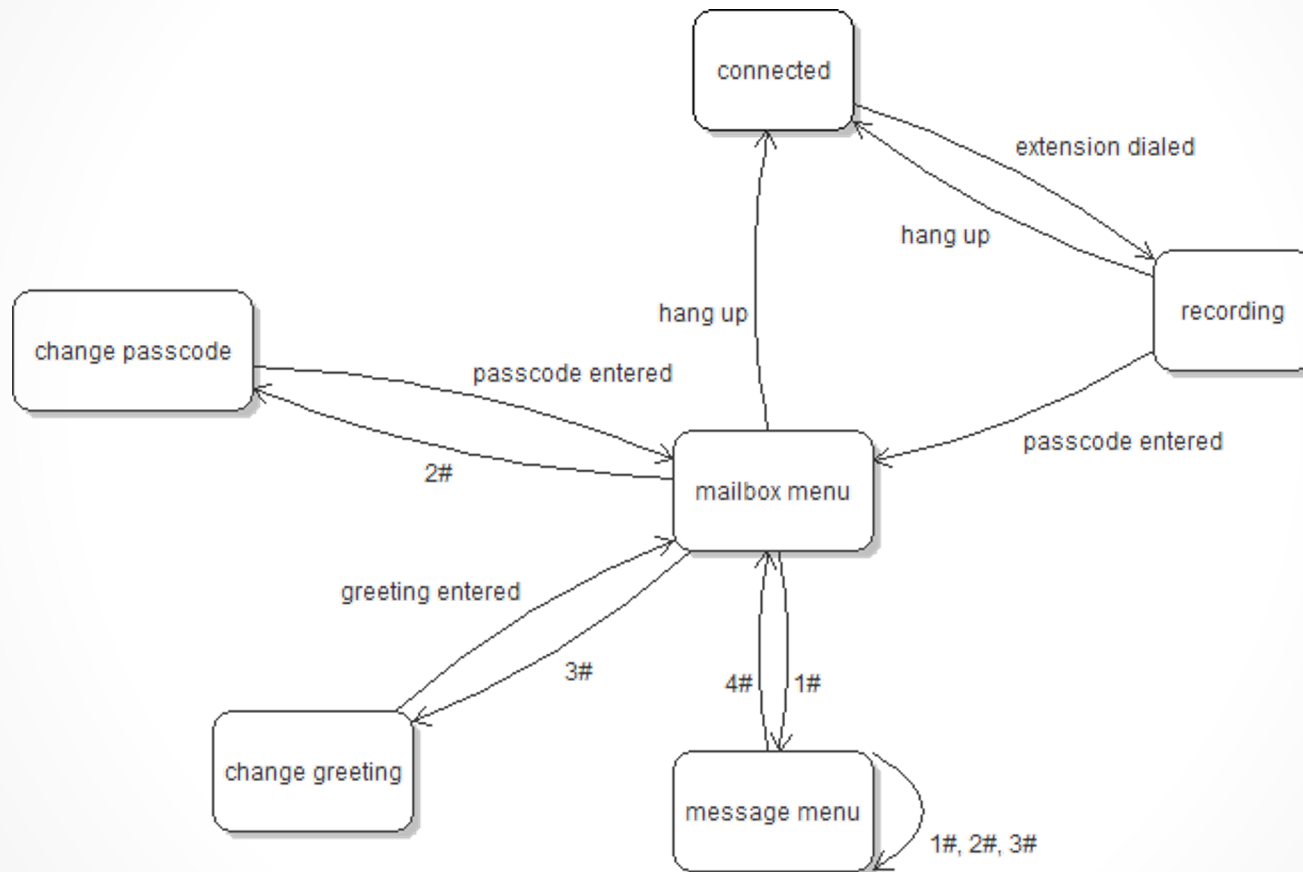
# Interpreting a Sequence Diagram

- 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 - Use Case: Retrieve messages



# Connection State Diagram



# Java Implementation

- See Java program files

## 2.12.5 Java Implementation (Pages: 74 – 85).

[Connection.java](#)

[Mailbox.java](#)

[MailSystem.java](#)

[Message.java](#)

[MessageQueue.java](#)

[Telephone.java](#)

[MailSystemTester.java](#)

# End of Chapter 2