The Object-Oriented Design Process
Horstmann Chapter 2

CS 3354
Spring 2017

Jill Seaman

2.1 From Problem to Code

Software Development Process (Wikipedia): A splitting of of software development work into distinct phases (or stages) containing certain activities with the intent of better planning and management.

Goal of programming: solve a problem, such as write a word processor.

Three phases of a simplified object-oriented software development process:

✦Analysis

✦Design

✦Implementation

These phases are often interleaved.

2.1.1 The Analysis Phase

✦Goal of the analysis phase: a precise description of tasks the software product should do.

✦Result is a detailed textual description called a **functional specification**

✦Having these characteristics:

- Completely defines tasks to be solved by the program
- Free from internal contradictions
- Readable both by customers, users, and software developers
- Testable against reality

✦Could be described using **use cases:**
a use case is a description of a sequence of actions that yields a benefit for a user of a system.

2.1.2 The Design Phase

✦Goal of the design phase: Structure the programming tasks into a set of interrelated classes.

- Identify classes (and attributes)
- Identify responsibilities of the classes
- Identify relationships among the classes

✦Results in the following artifacts:

- Textual description of classes and key responsibilities
- Diagrams of relationships among the classes
- Diagrams of important usage scenarios
- State diagrams for objects whose behavior is state-dependent

2.1.3 The Implementation Phase

- ✦Goal of the implementation phase: the programming, testing, and deployment of the software product.
  - Classes and methods are coded, tested, and deployed
  - Using the design phase artifacts as a guide
- ✦These activities are often performed iteratively:
  - Start with a small working version of the program (a prototype)
  - Test and debug
  - Implement more features and repeat
- ✦Testing must demonstrate that the program conforms to the analysis phase documentation (functional specification and/or use cases).

Review of Object-Oriented Concepts

- **Encapsulation**: combining data and code into a single object.
- **Data hiding (or Information hiding)** is the ability to hide the details of data representation from the code outside of the object.
- **Interface:** the mechanism that code outside the object uses to interact with the object.
  - ✦ The object's (public) functions
  - ✦ Specifically, outside code needs to "know" only the function prototypes (not the function bodies).

2.2 Object and Class Concepts

- Programming language independent view of Objects.
- **Objects** have state and behavior:
  - ✦State: the information stored by the object
    - This changes over time, if the object is mutable.
    - Values of the fields of a Java object
  - ✦Behavior: the operations an object supports
    - Methods a Java object can perform
  - ✦Identity: two objects could have the same state and behavior but be considered different from each other.
- **Class** is a collection of objects with the same behavior and common set of possible states.
  - ✦instance of a class is an object that belongs to the class.

2.3 Identifying Classes

- Look for nouns in the functional specification.
- Focus on concepts, not implementation
- The attributes of the nouns become the fields of the class that represent the state of the objects in that class.
  - ✦ Message might be a noun from the specification (for a voice mail system)
  - ✦ Attributes of a message (also found in the specification) become the fields:
    - Timestamp
    - Caller's phone number (or extension)
    - Text

Identifying Classes

- Other nouns typical of those that can be found in the functional description of a voice mail system:
  - Mailbox
  - Message
  - User
  - Passcode
  - Extension
  - Menu
- Now consider the storage of messages in a mailbox. The mailbox owner wants to listen to the messages in the order in which they were added: FIFO (first in, first out). So we'll add:
  - MessageQueue

Identifying Classes

- Once all the nouns in the functional specification are addressed, consider necessary classes from these categories:
  - Tangible things (like Mailbox, Message, Document, etc).
  - Agents (they perform an operation, like Paginator or Scanner)
  - Events and transactions (like MouseEvent)
  - Users and roles (like Administrator or Reviewer)
  - Systems (or subsystems) (like Mail System)
  - System interfaces and devices (these model the OS, like File)
  - Foundational classes (probably from the library, like String, Date, Rectangle)

2.4 Identifying Responsibilities

- Operations the class "should" perform
- Look for verbs in the functional specifications
  - Add message to mailbox
  - Remove message from mailbox
  - Set the text of a message
- Every operation should be the responsibility of a single class
- Not always easy to decide which class is responsible:
  - Example: Add message to a mailbox
  - Who is responsible, the message or the mailbox?

2.5 Identifying Relationships

- **Dependency** relationships: (objects of one class **use** instances of another)
  - Example: Mailbox uses a Telephone object to play a Message for a User.
- **Aggregation** relationships: (objects of one class **contain** instances of another)
  - Example: Mailbox contains a MessageQueue, MessageQueue contains Messages
- **Inheritance** relationships: (objects of one class **are special cases of** instances of another)
  - Example: University Person has an ID number, Student is a U.P. with a major and a GPA, Faculty is a U.P. with an office number.

Dependency Relationships

- Class A **depends** on Class B if it uses or refers to B in ANY way.
- If A can be developed without any knowledge of B, then it does NOT depend on B.
- **Coupling** is a measure of how strongly one class is connected to, has knowledge of, or relies on other classes.
- Goal: reduce coupling/dependency so changes to one class do not affect the others. For example:
- Replace:

```
void print() {System.out.println(text);}// prints to System.out
```

with `String getText() { return text; }//can be printed anywhere`

- Removes dependence on System, PrintStream

Aggregation Relationships

- Class A **aggregates** Class B if it objects of class A contain objects of class B over a period of time.
- It's a special case of dependency, implemented using **fields**.
- Multiplicity: how many objects of B can be related to an object of A?
  - **1 to 1** exactly one A is related to an instance of B and vice versa.
  - **1 to many** one A is related to any number of B's (a collection of B's), but each B has one A.
  - **many to many** an A has any number of B's, and vice versa.

Inheritance Relationships

- Class A **inherits** from Class B if it objects of class A are special cases of objects of class B, capable of exhibiting the same behavior but possibly with additional responsibilities and a richer state.
  - B is the superclass (it's more general)
  - A is the subclass (it's more specific)
  - subclass has added operations and attibutes
  - subclass has all operations and attributes of superclass (but may implement the operations differently).
- Inheritance is much less common than the dependency and aggregation relationships.

2.6 Use Cases

- Use Cases are an **analysis** technique to describe the functionality of the system from an external point of view.
- An <u>Actor</u> is an external entity that interacts with the system.
  - different kinds of users (roles), other systems, etc.
- A <u>Use case</u> is a textual description of (some of) the behavior of the system from an actor's point of view.
  - describes a single user/system interaction
  - described as a sequence of events: 1. user does X 2. system does Y …
  - focused on one goal of an actor (one kind of user)
  - yields a visible/observable result of value to the actor
  - should include variations that describe exceptional situations (failures)

Sample Use Case

Leave a Message

1. The **user** dials the main number of the voice mail system.
2. The **voice mail system** speaks the following prompt:

```
Enter mailbox number followed by #.
```

3. The **user** types in the extension number of the message recipient.
4. The **voice mail system** speaks the following message:

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

5. The **user** speaks the message.
6. The **user** hangs up.
7. The **voice mail system** places the recorded message in the recipient's mailbox.

Sample Use Case - Variations

Variation #1

1.1. In Step 3, the **user** enters an invalid extension number.

1.2. The **voice mail system** speaks:

```
You have typed an invalid mailbox number.
```

1.3. Continue with Step 2.

Variation #2

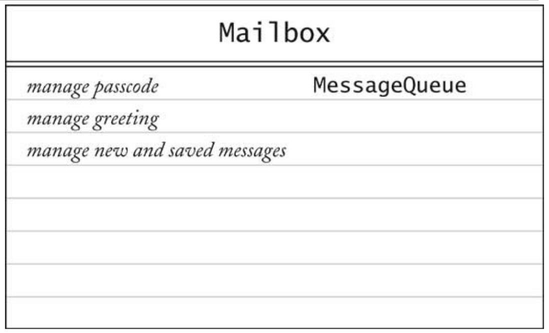
2.1. After Step 4, the **user** hangs up instead of speaking a message.

2.2. The **voice mail system** discards the empty message.

2.7 CRC Cards

- The CRC card method is an effective **design** technique for discovering classes, responsibilities, and relationships.
  - ✦ CRC = Classes, Responsibilities, Collaborators
- A CRC card is an **index card** that describes one class and lists its responsibilities and collaborators (dependent classes).
  - ✦ one card per class
  - ✦ class name on top
  - ✦ responsibilities on left (should be high level, ideally 1-3 per card)
  - ✦ collaborators on right (these are for the class, not for each responsibility).

Sample CRC Card



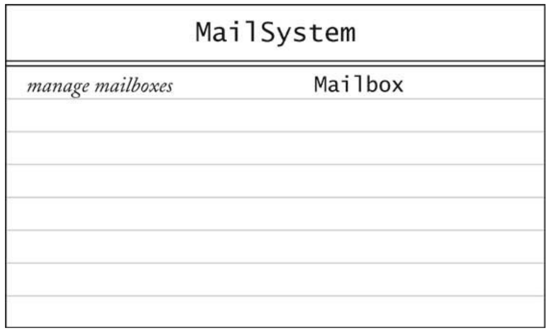
- Class=Mailbox
- Responsibilities:
  - ✦ manage passcode
  - ✦ manage greeting
  - ✦ manage new and saved messages
- Collaborators:
  - ✦MessageQueue

CRC Card Process: Walkthrough

- Take a use case and assign each task to one of the classes.
  - ✦ add new classes as necessary
  - ✦ look for collaborating classes to delegate responsibility.
- Example: Leave a message:.
  - ✦ User connects to voice mail system and dials extension number
  - ✦ "Someone" (some class) must locate mailbox:
    - Neither Mailbox nor Message can do this (not enough info)
    - Need to create a new class: MailSystem
    - Responsibility: manage mailboxes

New CRC Card

- Class=MailSystem
- Responsibilities:
  - ✦ manage mailboxes
- Collaborators:
  - ✦Mailbox



CRC Cards Process

- How to decide which class has which responsibilities? See guidelines in section 2.7:
  - ✦ Beware the "God" class: don't give the "system" class too many responsibilities.
  - ✦Avoid "mission creep". If a class acquires too many responsibilities, then consider splitting it in two.
  - ✦Watch out for unrelated responsibilities: storing messages and parsing input are not related (separate classes).
  - ✦Don't add responsibilities just because they can be done: "sort messages"
- Note: CRC cards are good for discovering classes and operations, but not for documenting them.

What is UML?

- Unified Modeling Language
- UML is a graphical notation to articulate (and communicate) complex ideas in Object-Oriented software development
- UML resulted from a unification of many existing notations.
- The goal of UML is to provide a standard notation that can be used by all object-oriented development methods.
- UML includes:
  - ✦Use Case Diagrams
  - ✦**Class Diagrams**
  - ✦**Sequence Diagrams**
  - ✦**State Diagrams**
  - ✦Activity Diagrams

Tools for Drawing UML Diagrams

The textbook mentions these:

- Rational Rose (http://www.ibm.com/software/rational/) ($$$)
- Together (Formerly of Borland) ($$$)
- ArgoUML (http://argouml.tigris.org/) and its commercial cousin Poseidon UML Community Edition (http://www.gentleware.com/)
- Dia (http://www.gnome.org/projects/dia)
- For simple UML diagrams, you can use the Violet tool that you can download from http://horstmann.com/violet.

UMLet

- **Umlet** (http://umlet.com) Free UML Tool for Fast UML Diagrams. I recommend this one!! Download, then double click on umlet.jar. Or try the web app: http:umletino.com (UMLetino)
- Each diagram has a graphical AND a textual representation, and both are displayed in the tool.
- The tool allows you to create and modify your diagrams using the graphical tool (drag and drop, resize, etc) as well as the editor tool (modify the textual representation).
- The diagram is stored in a whatever.uxf file, which actually stores the textual representation in XML format.
- See UmletSamples.zip on the class website (or TRACS) for uxf files of some samples used in the slides.
- Documentation: http://www.umlet.com/faq.htm

2.8 Class Diagrams

- Used to describe the internal structure of the system.
- They are static: they display the (unchanging) relationships among the classes that exist throughout the lifetime of the system.
- They describe the system in terms of
  - ✦**Classes**, an abstract representation of a set of objects
  - ✦**Attributes**, properties of the objects in a class
  - ✦**Operations** that can be performed on objects in a class (responsibilities)
  - ✦**Relationships** that can occur between objects in various classes

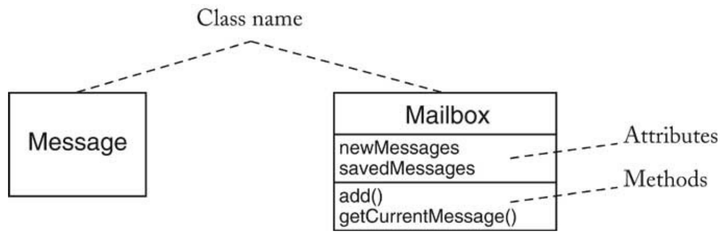
Class Diagrams: details

- Classes are boxes composed of three compartments:
  - ✦Top compartment: name
  - ✦Center compartment: attributes
  - ✦Bottom compartment: operations/methods
- Include only key attributes and methods (not getters/setters, etc.)
- Attribute and Operations: are often omitted for simplicity
- Data types for attributes and operation parameters and results are also optional (note type goes after the attribute/parameter/function)

```
text : String
getMessage(index : int) : Message
```

Sample Class Boxes

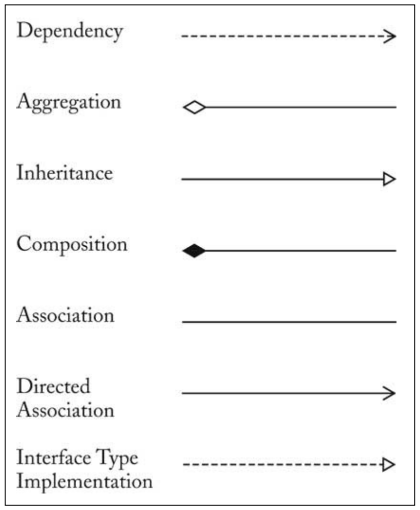
- Message and Mailbox are Class names
- newMessages and savedMessages are Attributes
- add() and getCurrentMessages() are Operations (or Methods)
- Note the one on the left does not include attributes or operations.



UML Connectors for Relationships

These are the types of relationships that are commonly denoted in UML class diagrams:

- Dependency (dashed line)
- Aggregation (line with open diamond)
- Inheritance (line with open triangle)
- Composition (line with closed diamond)
- Association (line)
- Directed Association (line with arrow)
- Interface Type Implementation (dashed line with open triangle)



Special Cases of Dependency

- **Directed relationships**: Arrow from A to B indicates the direction of navigation. No arrow indicates a bidirectional relationship.

These describe directed relationships, represented using an arrow from A to B:

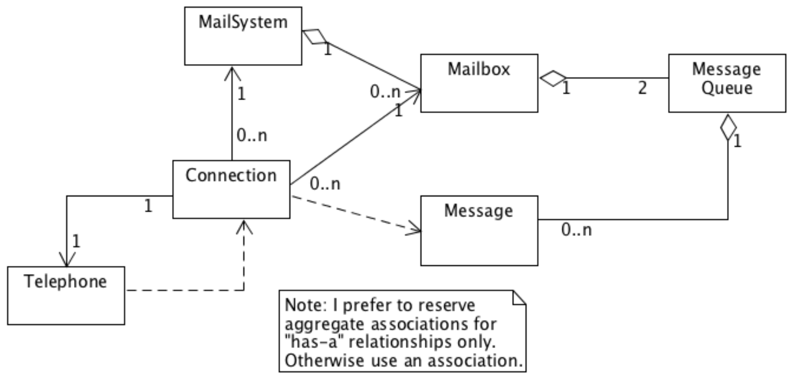
- Dependency: objects of class A use objects of class B in ANY way.
- Association: special kind of dependency where objects of class A contain a class-level reference to objects of class B
- Aggregation: nearly equivalent to an Association: represents “has-a” relationship. Open diamond is on the owner. Usually bidirectional (no arrows, B knows about its owner).
- Composition: special kind of Aggregation/Association where A owns the instances of B (it creates them and destroys them).

Multiplicity

- Multiplicity: a range of integers labeling one end of an **association.**
- It indicates how many objects of that class can be related to one object of the other.
  - any number (0 or more): 0..n, 0..* or *
  - one or more: 1..n or 1..*
  - zero or one: 0..1
  - exactly one: 1
- Most associations belong to one of these three types:
  - ✦A **one-to-one, one-to-many,** or **many-to-many** (see slide 14)
- Multiplicity applies to these connectors **only**:
  - ✦Association, Aggregation, Composition.

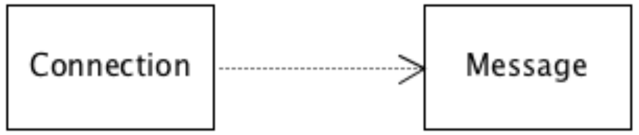
Sample Diagram:

- ClassDiagram.uxf from UmletSamples.zip (my version of the voice mail system, classes and relationships only).


MailSystem
Mailbox
Message Queue
Connection
Message
Telephone
1
0..n
2
Note: I prefer to reserve aggregate associations for "has-a" relationships only. Otherwise use an association.

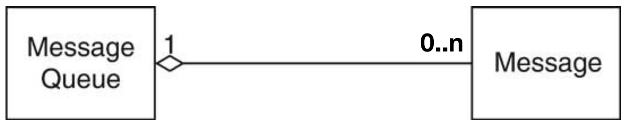
Dependency Example

- Class A **depends** on Class B if it uses or refers to B in ANY way.
- This drawing indicates:
  - Connection uses Message in some way (other than aggregation, inheritance, etc.)
    - ✦ Connection objects create and/or obtain Message objects, and pass them as arguments to methods on other objects.
  - Message does not use Connection in any way.



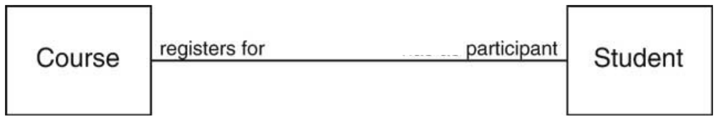
Aggregation with Multiplicity Example

- This drawing indicates:
  - MessageQueue aggregates (contains) Messages, because the open diamond next to MessageQueue indicates the container class.
  - Each Message Queue object contains any number (0 or more) Message objects, because of the 0..n next to Message.
  - Each Message object is contained by at exactly 1 Message Queue object because of the 1 next to MessageQueue.



Association Example

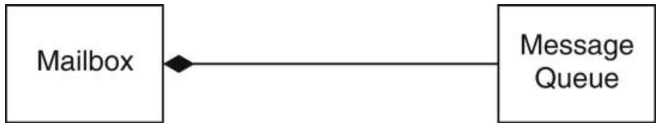
- Some designers do not like aggregation and prefer to use a more general association relationship
  - ✦No diamonds
  - ✦Can (optionally) specify roles at the ends of the association,
  - ✦Roles clarify the purpose of the association.
- For example: the student to course relationship.



- ✦You may use either associations or aggregations in your diagram, as long as you indicate the multiplicity.

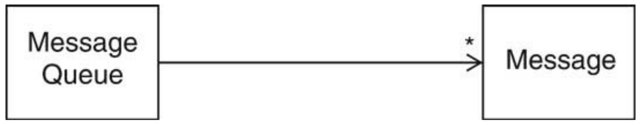
Composition Example

- Composition is a special case of aggregation where the composite (container) object has sole responsibility for the life cycle of the component parts.
  - ✦The component object cannot exist outside the container.
  - ✦An object may be part of only one composite.
  - ✦Specified with a closed diamond on the composite (whole) side.
  - ✦Not used by all designers.
  - ✦Could be used to represent the Mailbox to MessageQueue association.



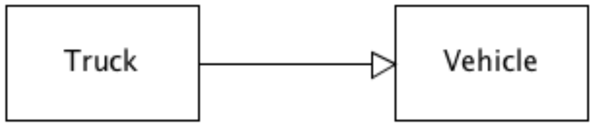
Directed Association Example

- Associations can be Bidirectional or Unidirectional.
  - ✦Unidirectional association is indicated by using a line with an arrow
  - ✦The arrow indicated in which direction navigation is supported.
  - ✦The diagram below is a version of the MessageQueue to Message relationship indicating we can get from Message Queue to the Message, but Message does not know about its containing Message Queue.



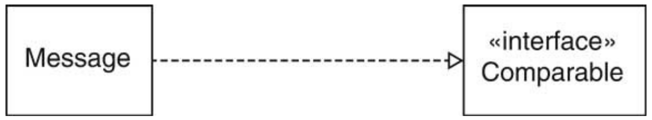
Inheritance Example

- Objects of class A are special cases of objects of class B
- Class A objects inherit attributes and operations from class B.
- In UML:
  - ✦Solid line with open arrow pointing to the Superclass
- A Truck is a special case of a Vehicle:



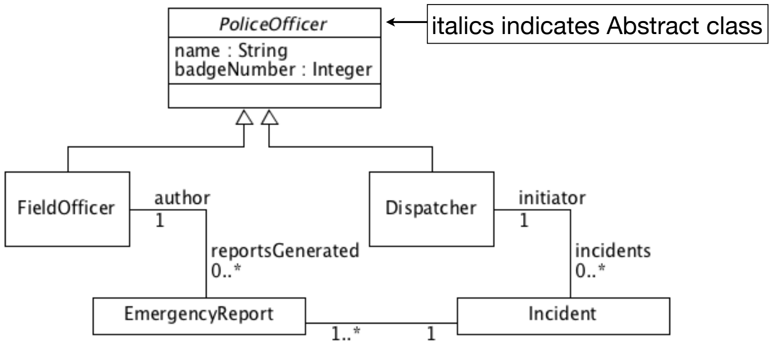
Interface Types Example

- Objects of class A implement the methods of class B (the interface)
- An Interface type describes a set of methods, but has no attributes.
- A class implements an interface type if it implements its methods
- In UML, use stereotype «interface» for the type
  - ✦Dashed line with open arrow pointing to the Interface type
- A Message implements the methods of Comparable:



Sample Class Diagram with Inheritance

- Note that FieldOfficer and Dispatcher inherit from PoliceOfficer
  - ✦ the subclasses may have attributes and operations of its own, as well as the attributes and operations of the base class (it inherits them).



Class Diagram with an Interface

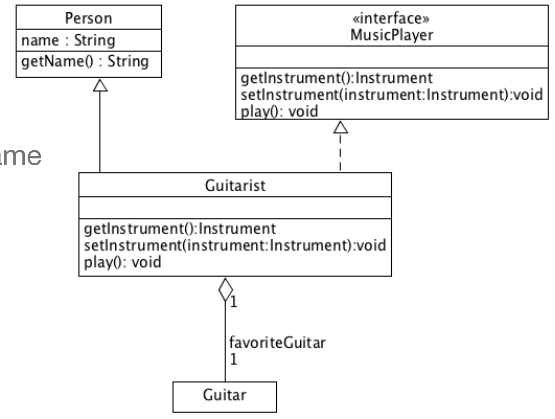
Which of the following are true according to this diagram?

a) Persons have a name

b) Guitarists have a name

c) Guitars have a name

d) MusicPlayers have a name



Person
name : String
getName() : String
«interface»
MusicPlayer
getInstrument():Instrument
setInstrument(instrument:Instrument):void
play(): void
Guitarist
getInstrument():Instrument
setInstrument(instrument:Instrument):void
play(): void
1
favoriteGuitar
1
Guitar

Class Diagrams: Good Practices

- Do not use an attribute in a class to represent a relationship already in the diagram
  - ✦Among other problems, this is redundant

- Comprehensive class diagrams with all classes, attributes, and associations are too difficult to read.
  - ✦ Use UML to inform, not to impress
  - ✦ Don't draw a single monster diagram
  - ✦ Each diagram must have a specific purpose
  - ✦ Omit inessential details

2.9 Sequence Diagrams

- Represent the dynamic behavior of the system
- A sequence diagram shows the time ordering of a sequence of method calls, including nested method calls.
- They describe patterns of communication among a set of interacting objects.
  - ✦ Objects communicate by calling methods on other objects.
- A sequence diagram could be used to represents the steps of a use case.

Objects

- Sequence diagrams describe interactions between Objects (NOT Classes).
- **Objects** are represented with a box containing the object's name and the Class the object is an instance of.
    - ✦ It is underlined to indicates this is an **object**, not a class.
- The text inside the box has one of these formats:
    - ✦ objectName : ClassName
    - ✦ objectName (class not specified)
    - ✦ : ClassName (object not specified)
- Objects must be in a row (in any order) at the top of the diagram.

Lifelines and Activation Boxes

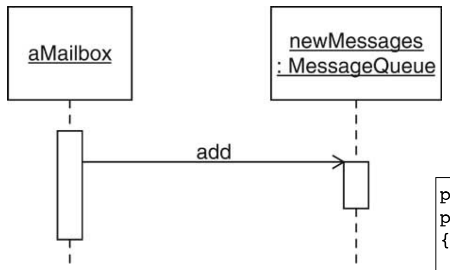
- The dotted line below the object is the object's **lifeline**
    - ✦ Vertical rectangle along the lifeline: an **activation bar** representing a method that is currently being executed.
    - ✦ The activation bar ends when the method returns.
    - ✦ The activation bar of a called method must be smaller than that of the calling method.

Method Calls

- **Method calls** are represented with a solid horizontal arrow from one object's lifeline to another.
    - ✦ The arrow is labelled with the method name (arguments are optional).
    - ✦ It must end at the top of an activation bar on the lifeline of the object on which the method is called.
    - ✦ The direction of the arrows (left to right or right to left) and the position of the arrows on the page indicate the sequence of method calls during the operation of the program.
        - An arrow closer to the top of the page occurs before one that is lower on the page.

Sequence Diagram with One Method Call

- What does the sequence diagram below illustrate?
    - ✦aMailbox is an object, probably of type Mailbox, and an unspecified method has been called on it.
    - ✦During the execution of that method, an add method has been called on the newMessages object (of type MessageQueue).

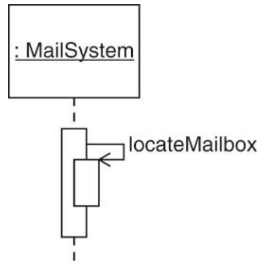


Here is the corresponding code excerpted from the Mailbox Class:

```
private MessageQueue newMessages;
public void addMessage(Message aMessage)
{
   newMessages.add(aMessage);
}
```

Self-call

- a Self-call is when an object calls one of its own methods
  - ✦ use a method call arrow back to original activation bar
  - ✦embed a new activation bar in the currently executing one:



Hypothetical code from MailSystem

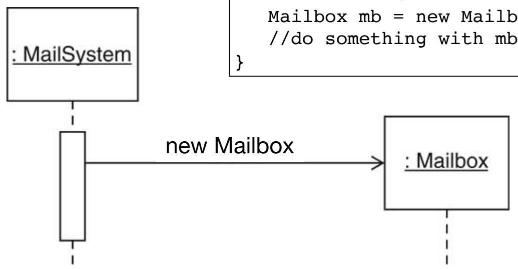
```
private Mailbox locateMailbox(String ext){
…
}
public void anotherMethod() {
   String test = “123”
   Mailbox mb = locateMailbox(test);
   //do something with mb here
}
```

Create a New Object Instance

- Creating new instances:
  - ✦“new Class()” message points directly to (lowered) object’s box:

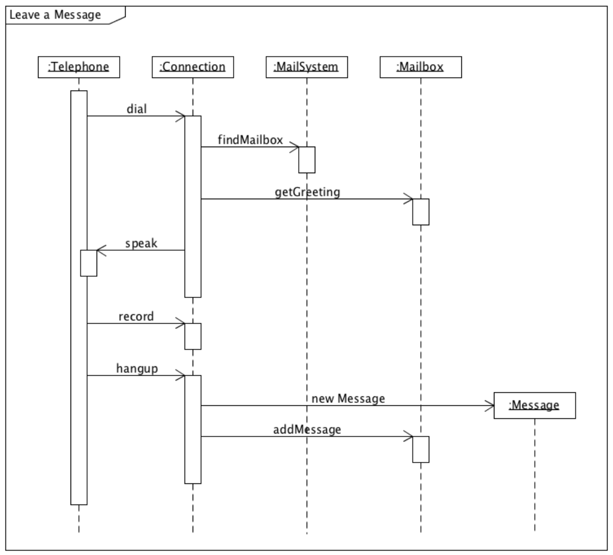
Hypothetical code from MailSystem

```
public void someMethod(int i) {
   String passcode = "" + (i + 1);
   String greeting = "You have reached mailbox " + (i + 1)
         + ". \nPlease leave a message now.”;
   Mailbox mb = new Mailbox(passcode, greeting)
   //do something with mb
}
```



Sample Diagram

- SequenceDiagram.uxf from UmletSamples.zip



Sample Diagram (some of the code):

Code from Telephone:

```
public void run(Connection c) {
   boolean more = true;
   while (more)  {
      String input = scanner.nextLine();
      if (input == null) return;
      if (input.equalsIgnoreCase("H"))
         c.hangup();
      else if (input.equalsIgnoreCase("Q"))
         more = false;
      else if (input.length() == 1
         && "1234567890#".indexOf(input) >= 0)
         c.dial(input);
      else
         c.record(input);
   }
}
```

Code from Connection:

```
private Mailbox currentMailbox;
public void hangup() {
   if (state == RECORDING)
      currentMailbox.addMessage(new Message(currentRecording));
   resetConnection();
}
```

Strategy: Creating Sequence Diagrams From Code

- Make a table of method calls in the order they occur.
- For each method call, list the caller and the callee (the first one doesn't have a caller):

```
Method    caller class     callee class
------    ------------     ------------
scrub       (none)           A
foam       A                 B
…
```

See file Sequence.java on TRACS

- Now make boxes and lifelines for each of the classes.
- Add a labeled arrow for each row in the table, from the caller class's lifeline to the callee class's lifeline (label with method name).
- Then add activation records to capture the duration of each method.

Sequence Diagrams: Good Practices

- Do not indicate branches or loops in Sequence Diagrams.
  - ✦The UML defines a notation for that purpose, but it is a bit cumbersome and rarely used.
- The sequence diagram should be consistent with a given class diagram (or code) that specifies the classes and their operations
  - ✦ messages to an object's lifeline must correspond to valid operations for that object's class
- Activation boxes should not overlap horizontally unless one box's message has called the other.
- Don't have a sequence diagram for each use case, only for the scenarios that involve several interacting classes.

When and How to Use Sequence Diagrams

- During either design (forwards engineering) or implementation (reverse engineering)
- When you want to look at the behavior of several objects within a single use case.
- When the order of the method calls in the code seems confusing.
- When you are trying to determine which class should contain a given method.
  - ✦to uncover the responsibilities of the classes in the class diagrams
  - ✦to discover even more new classes
- During Object-Oriented Design, sequence diagrams and the class diagram are often developed in tandem.

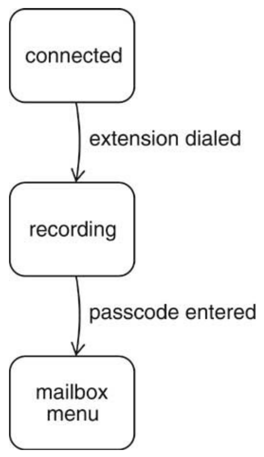
2.10 State Diagrams

- Describe dynamic behavior of an individual object (or subsystem) that can be in various states at different points in time.
  - ✦Not all objects have different states. Most do not.
- A state diagram describes the sequence of states an object goes through in response to external events
  - ✦A graph: states are nodes, transitions are directed edges
- Transitions from one state to another occur as a result of external events

States and Transitions

- A **state** is (often) represented as a value of an attribute of an object that is changed by an external event.
  - ✦A voice mail system can exist in three states: Connected, Recording, or Mailbox Menu
- A state is a node in the graph
- The state usually has noticeable impact on the behavior of the object
- A **transition** represents a change of state triggered by events, conditions, or time.
  - ✦Transitions are directed edges in the graph
  - ✦Edges are usually labelled by the event causing the transition

State diagram for Connection class (Incomplete)



- When the caller dials the voice mail system, it enters the connected state.
- After it dials a valid extension, the system enters the recording state, where it records whatever the caller speaks
- When the caller enters a passcode, the system is in the mailbox menu state.

State affects the behavior of the object

- In the “mailbox menu” state, spoken words are ignored.
- Voice input is recorded only when the system is in the “recording” state.
- The telephone touchpad has no concept of the states. A key press might be part of the mailbox number, passcode, or menu command.

When and how to use State Diagrams

- When designing a class that has an attribute that responds to external events (and determining which state the object is in is not trivial)
  - ✦Use the state diagram to document the transitioning behavior
- During testing
  - ✦If you have a state diagram, you can develop tests that perform a sequence of events and then verify that the object is in the correct state with respect to the diagram
- If your object (or system) does not have an attribute that responds to external events, do not use state diagrams.
- User Interface objects often have behavior that is useful to depict with a state diagram