The Object-Oriented Design Process

Horstmann Chapter 2

CS 4354
Summer II 2016

Jill Seaman

2.1 From Problem to Code

A Simplified Software Development Process, three phases:

- ✦Analysis
  - Completely defines tasks to be solved by the program
  - Result is a detailed textual description called a Functional Specification
- ✦Design
  - Structures the programming tasks into a set of interrelated classes
    - ❖ Identify classes (and attributes)
    - ❖ Identify responsibilities of the classes
    - ❖ Identify relationships among the classes
- ✦Implementation
  - Classes are coded and tested

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 Objects and Classes

- **Objects** have state and behavior:
  - ✦State: the information stored by the object
    - Values of the fields of a Java object
  - ✦Behavior: the operations an object supports
    - Methods a Java object can perform
- **Class** is a collection of objects with the same behavior and common set of possible states.

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
    - – Callers phone number (or extension)
    - – Text

Identifying Classes

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

2.4 Identifying Responsibilities

- Look for verbs in the functional specifications
  - ✦ Add message to mailbox
  - ✦ Remove message from mailbox
  - ✦ Set the text of a message
- Every operation is 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 **uses** instances of another)
  - ✦ Example: Mailbox uses a Telephone object to play a Message for a User.
- **Aggregation** relationships: (objects of one class **contains** 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 also has a major and a GPA, Faculty also has 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 print 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 instance fields.
- Multiplicity: how many objects of B can be related to an object of A?
  - **1 to 1** exactly one B is related to an instance of A 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 Actor is an external entity that interacts with the system.
  - different kinds of users (roles), other systems, etc.
- A Use case 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.
  - focused on one goal of an actor
  - 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 caller 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 caller 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 caller speaks the message.
6. The caller 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.`

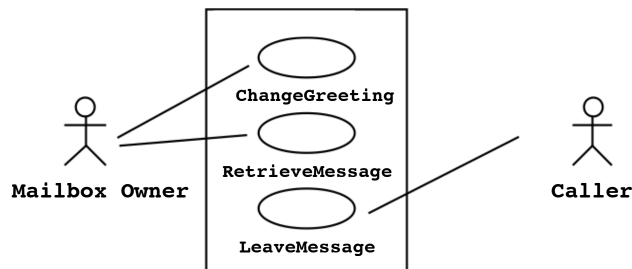
1.3. Continue with Step 2.

Variation #2

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

2.2. The voice mail system discards the empty message.

Use case diagram for a voice mail system



- Actors are stick figures
- Use cases are ovals, with name usually inside the oval
- The boundary of the system is the box
- Each oval corresponds to one of the use case descriptions
- Purpose of this diagram: completeness
  make sure no desired use cases are missing.

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

Mailbox	
manage passcode	MessageQueue
manage greeting	
manage new and saved messages	

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:.
  - ✦ Caller connects to voice mail system and dials extension number
  - ✦ "Someone" 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

MailSystem	
manage mailboxes	Mailbox

CRC Cards process

- How to decide which class has which responsibilities?
  - ✦ see guidelines in 2.7
  - ✦ GRASP patterns (a later topic) formalize these guidelines
- 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
  - ✦State Diagrams
  - ✦Sequence Diagrams
  - ✦Activity Diagrams

Tools for drawing UML diagrams

- 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** (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)

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
  - ✦Associations that can occur between objects in various classes

Class Diagrams: details

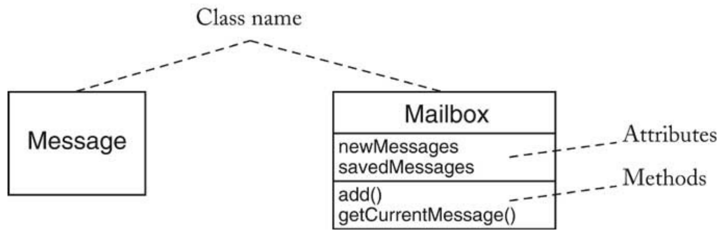
- Classes are boxes composed of three compartments:
  - ✦Top compartment: name
  - ✦Center compartment: attributes
  - ✦Bottom compartment: operations
- Lines between classes represent associations between classes
- Types for attributes and operation parameters and results are optional (note type goes after the attribute/parameter/function)

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

- Attribute and Operation compartments are sometimes omitted for simplicity

Sample classes

- The class names, attributes, and operations/methods are labelled (the labels and dashed lines are not part of the classes).
- Note the one on the left does not include attributes or operations.



Associations

- Associations are the relationships between classes
  - ✦Associations are noted with a line between the boxes.
- Associations can be symmetrical (bidirectional) or asymmetrical (unidirectional).
  - ✦Unidirectional association is indicated by using a line with an arrow
  - ✦The arrow indicated in which direction navigation is supported.
  - ✦If the line has no arrows, it's assumed to be bidirectional.

UML Connectors for associations

Dependency

Aggregation

Inheritance

Composition

Association

Directed Association

Interface Type Implementation

- The first three types are discussed in section 2.4
- The others are in upcoming slides

- These are significant:
  - ✦Shape of the arrow head
  - ✦Direction of the arrow
  - ✦Dashed or solid lines

Multiplicity

- Multiplicity: a set of integers labeling one end of an association
- It indicates how many objects of class B can be related to an object of class A.
  - 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** association has a multiplicity 1 on each end.
  - ✦A **one-to-many** association has a multiplicity 1 on one end and 0..n or 1..n on the other.
  - ✦A **many-to-many** association has a multiplicity 0..n or 1..n on both ends.

Multiplicity example

- This drawing indicates:
  - Message Queue aggregates (contains) Messages, because the open diamond indicates the container class
  - Each Message Queue object contains any number (0 or more) Message objects.
  - Each Message object is contained by at most 1 Message Queue object (a Message object cannot belong to more than one Queue).

Aggregation and Composition

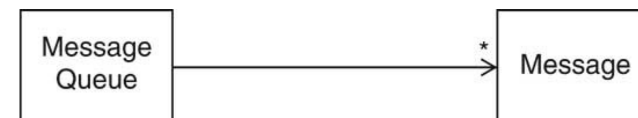
- Composition is a special case of aggregation where the composite (container) object has sole responsibility for the life cycle of the component parts.
  - ✦The component object cannot exist outside the container.
  - ✦An object may be part of only one composite.
  - ✦Specified with a closed diamond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

Association

- Some designers do not like aggregation and prefer to use a more general association relationship
  - ✦No diamonds
  - ✦Can specify roles at the ends of the association,
  - ✦Roles clarify the purpose of the association.

Directed Associations

- 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. Can get from Message Queue to the Message, but Message does not know about its containing Message Queue
  - ✦If the line has no arrows, it's assumed to be bidirectional. Student can find the Courses its registered for, and the Course can find the students registered for it.

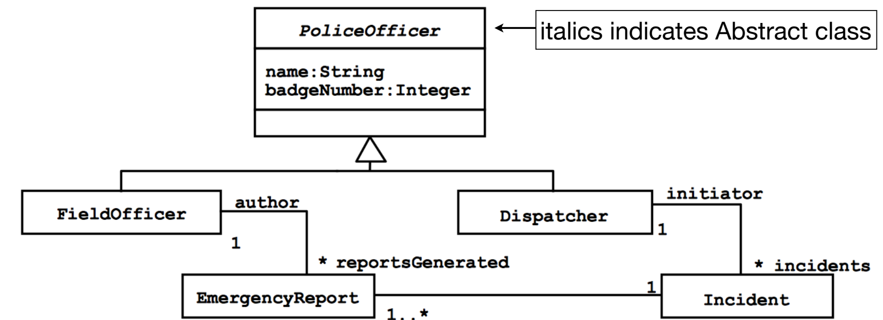


Interface Types

- An Interface type describes a set of methods
- It has no attributes
- A class implements an interface type if it implements its methods
- In UML, use stereotype «interface»
  - ✦Dashed line with open arrow pointing to the Interface type

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

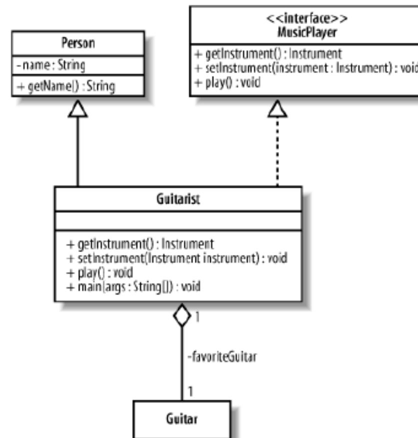
Which of the following are true according to this diagram?

a) Persons have a name

b) Guitarists have a name

c) Guitars have a name

d) MusicPlayers have a name



Class Diagrams: Good Practices

- Do not add an attribute to a class to represent the end of an association already in the diagram
  - ✦Among other problems, this is redundant
- Too many attributes (and/or operations) for one class: split the class into two.
- Comprehensive class diagrams with all classes, attributes, and associations are too difficult to read.
  - ✦Each diagram should clearly depict one aspect of the design
  - ✦Omit the unnecessary details (but be consistent)
  - ✦You will probably have several class diagrams depicting different aspects of the design

2.9 Sequence Diagrams

- Represent the dynamic behavior of the system
- They describe patterns of communication among a set of interacting objects.
- Objects communicate by calling operations (methods) on other objects. These are called messages.
- A sequence diagram usually represents the steps of a use case.

Objects, lifelines, and activation boxes

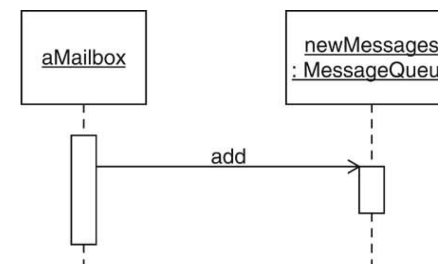
- **Objects** are represented with a box containing a name and the Class the object is an instance of.
  - ✦ *name : Class*
  - ✦ The underline indicates this is an object, not a class.
  - ✦ Only one part (the name or the Class) is required to be specified.
- The dotted line below the object is the object's **lifeline**
  - ✦ Vertical rectangle: an **activation box** representing the duration of an operation.
  - ✦ There must be a message pointing to the top of the box indicating the operation the box corresponds to.

Messages and return arrows

- **Messages** are represented with a solid horizontal arrow from one object's lifeline to another and represent method calls.
  - ✦ The call originates in the object at the source of the arrow
  - ✦ It is received by the object at the end of the arrow
  - ✦ The order in which the messages occur is top to bottom on the page.
  - ✦ The message must be labeled with the name, but arguments are optional.
- **Return arrows** are dashed arrows from the bottom of the activation box back to the lifeline of the object that sent the initial message.
  - ✦ These are optional!
  - ✦ They might be labeled with the return value

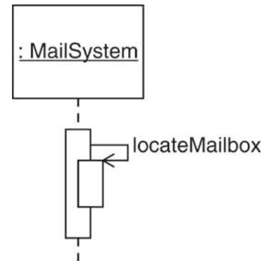
Sequence diagram with one operation

- This illustrates the add method of the MessageQueue class
- A Message object is added to the Message Queue that holds the new messages
  - ✦The diagram corresponds to the java statement: `newMessages.add(...)` but the parameters are not indicated in this diagram

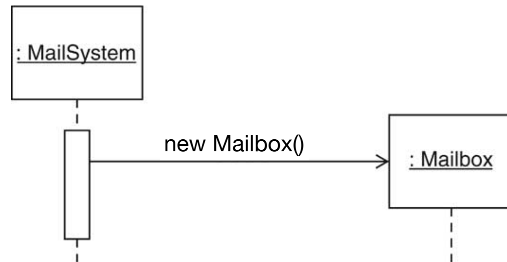


Self-call and Create new

- Self-call (object calls one of its own methods)
  - ✦Message arrow back to original activation:



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



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 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.
- We will see examples in the case study at the end of this chapter.

When and how to use Sequence Diagrams

- 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 new classes
- During Object-Oriented Design, sequence diagrams and the class diagram are often developed in tandem.

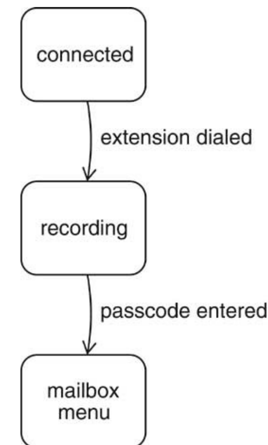
2.10 State diagrams

- 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.
- 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

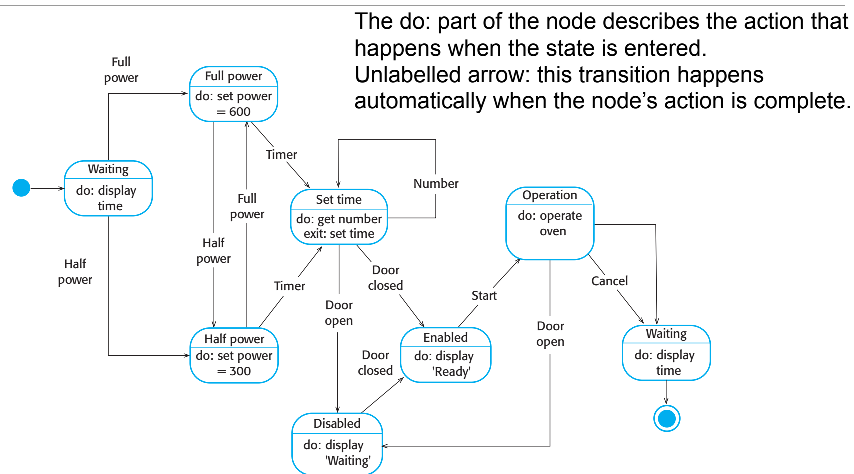
State diagram for voice mail system (incomplete)



- 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 diagram of a microwave oven

The do: part of the node describes the action that happens when the state is entered.
Unlabelled arrow: this transition happens automatically when the node's action is complete.



This diagram is missing (at least) one transition

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