Object-oriented design

More UML
Interfaces

• An interface is a language construct specific to Java
• Java does not support multiple inheritance
• Interfaces provide some of the same functionality
  – A Java class can only inherit from one class
  – But can implement many interfaces
ATM Example

This hierarchy chart represents the ATM device classes as envisioned by the preliminary design. Each of the concrete subclasses represents an actual physical device; the abstract classes are used to abstract their common behavior. The design calls for Bank Card Reader and Receipt Printer to inherit from multiple superclasses, but this can’t be done in Java.
Using Interfaces

• Instead of redoing the design, can make the abstract classes into interfaces

• An interface specifies, but does not implement, specific methods; the implementation is left up to the subclasses that “inherit” the interface
Inheritance vs. Interfaces

• We can think of “real” inheritance as implementation inheritance
  – Subclasses inherit and may extend functionality of superclass
  – Superclasses pass on structure and behavior
Inheritance vs. Interfaces

• Interfaces provide only *definition* inheritance
  – Methods may be specified, but there is no superclass implementation
  – Instead, the subclasses define what is specified in the interface
  – Interfaces pass on structure, but not behavior
Interfaces & UML

• Interface relationships look like inheritance relationships, except a dashed line is used for connecting subclasses to superclasses

• Each interface’s name is enclosed in a double set of angle brackets: << >>
ATM Example

<<Input Device>>
Deposit Slot
Keypad
Bank Card Reader

<<Output Device>>
Cash Dispenser
Receipt Printer

<<Display Device>>
Display Screen
Dependency

• Dashed lines with closed arrowheads, as shown in the previous diagram, indicate definition inheritance via interfaces

• A dashed line with an open arrowhead simply indicates a dependency - that is, one class depending on another to fulfill a responsibility
Dependency example

In a previous lecture, we looked at the fact that Java doesn’t allow for operator overloading, which can complicate I/O tasks with dependencies.

The example below illustrates such a dependency; the Obj class contains a method that prints data to the screen, forcing reliance on two classes from the Java API, System and PrintStream.
Sequence Diagrams

- Sequence diagrams describe the interaction of classes as their methods are called.
- Where class diagrams provide a static picture of class relationships (basically just telling us that a relationship exists), sequence diagrams provide a dynamic picture of a system in action.
- Time is represented by the vertical dimension; the further down in the diagram, the later a method is invoked.
Elements of a sequence diagram

• Rectangles representing classes and objects
  – object names are underlined, and may appear in several forms:
    • full description: objectname: Classname
    • no class specified: objectname
    • class specified without object name; read as “some instance of this class”: : Classname
  – class names are not underlined

• Method names and other text
Elements of a sequence diagram

- Lifelines: dashed vertical lines representing the period of time during which an object exists (can be left indeterminate)
- Activation bars: rectangles laid over lifelines, representing a method’s execution in process
- Call arrows: horizontal arrows pointing from the calling object or class to the object or class that owns the called method
Elements of a sequence diagram

• Data tokens: short lines with an arrowhead on one end and a circle on the other, representing data items flowing between objects
  – Tokens pointing to the right represent arguments passed to methods
  – Tokens pointing the left represent return values
Sequence diagram: example 1

aMailbox is the calling object - calls MessageQueue.add(message)
This diagram represents a self call: the unnamed MailSystem object is calling its own locateMailbox method.
In this diagram, the MailSystem object calls the constructor for a Mailbox object. The <<create>> stereotype indicates the constructor call; note that the Mailbox object is slightly below the MailSystem object, because its lifeline doesn’t start until MailSystem creates it.
Sequence diagram: example 4

Each key press results in separate call to dial, but only one is shown
Connection must find mailbox object: Call findMailbox on MailSystem object
Connection wants to get greeting to play: Each mailbox knows its greeting
Parameters and return values are not displayed in this diagram
Note that connection holds on to that mailbox over multiple calls
State diagrams

- Some objects have a discrete set of states that affect their behavior; this is particularly true of objects that interact with the user.
- A state diagram shows the various states of a system while a particular process is carried out.
- State diagrams can be used to illustrate variations in a use case.
State diagram example - ATM

- Wait state - display greeting
  - Card is entered
    - Read card
      - Verify PIN
        - Incorrect
          - Display error message, wait for response
          - Try again
        - Correct
          - Display main menu
State diagram example - voice mail system