CS1001

Lecture 18

Overview

- Object Oriented Design

Goals

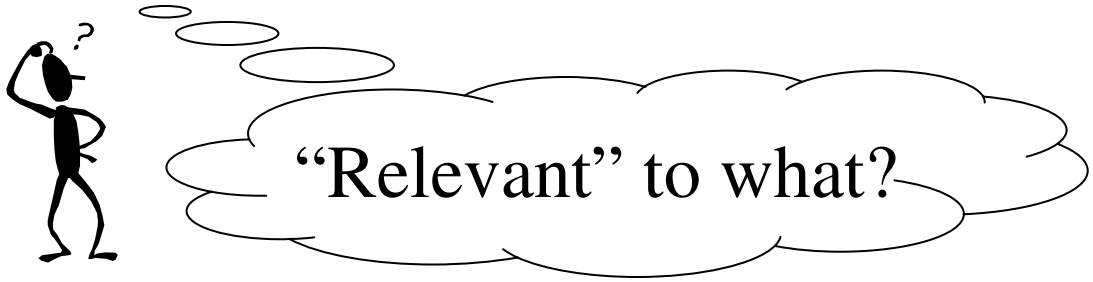
- Learn Object-Oriented Design Methodologies

Assignments

- Brookshear: Ch 5.5, Ch 6.3/6.4, Ch 7 (especially 7.7) (Read)
- Read linked documents on these slides (slides will be posted in courseworks)

Abstraction

- Abstraction means ignoring irrelevant features, properties, or functions and emphasizing the relevant ones...



- ... relevant to the given project (with an eye to future reuse in similar projects).

Abstraction (cont'd)

- Example from javax.swing:
  public abstract class AbstractButton

  Fields:

  protected ButtonModel model ←
  etc.

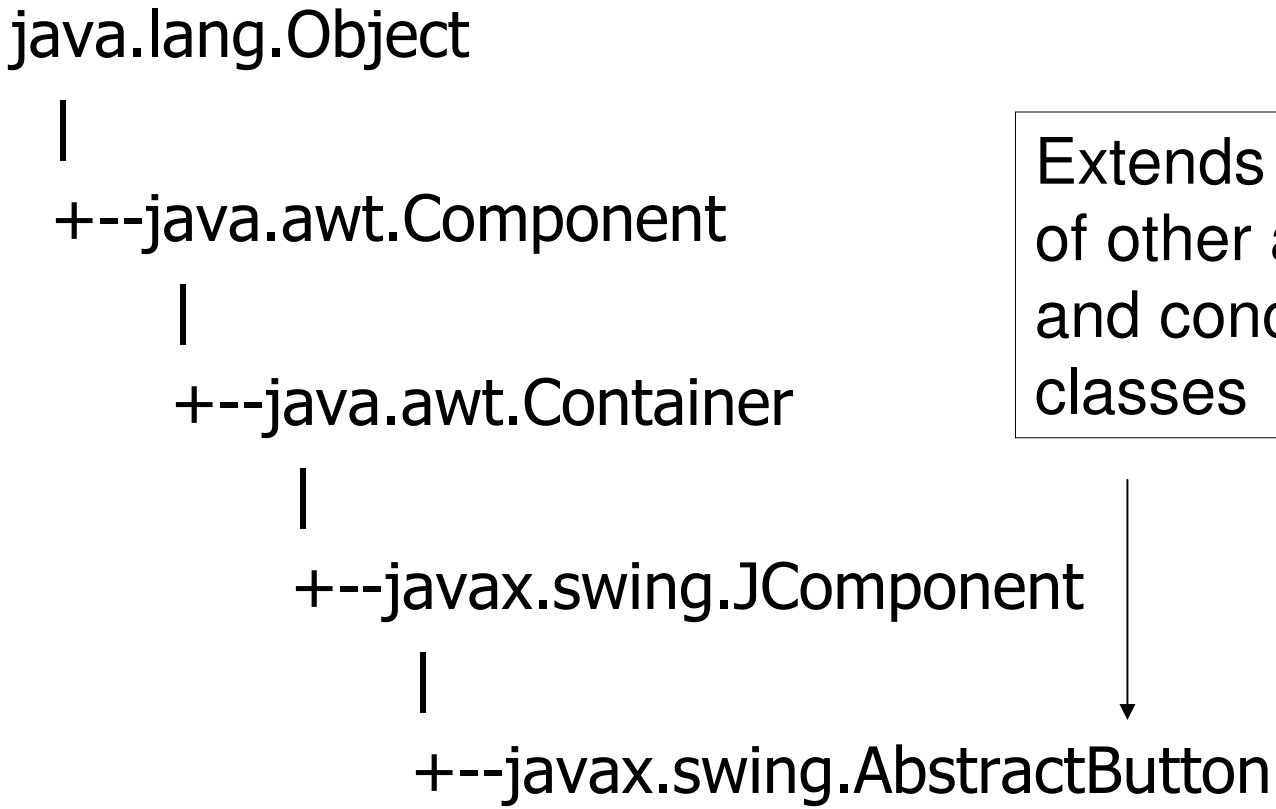
  The data model that determines the button's state

  Methods:

  void addActionListener (ActionListener l);
  String getActionCommand();
  String getText()
  etc.

  Apply to any button: "regular" button, a checkbox, a toggle button, etc.

Abstraction (cont'd)



Encapsulation

- Encapsulation means that all data members (*fields*) of a class are declared private. Some methods may be private, too.
- The class interacts with other classes (called the *clients* of this class) only through the class's constructors and public methods.
- Constructors and public methods of a class serve as the *interface* to class's clients.

Encapsulation (cont'd)

- Ensures that structural changes remain local:
  - Usually, the structure of a class (as defined by its fields) changes more often than the class's constructors and methods.
  - Encapsulation ensures that when fields change, no changes are needed in other classes (a principle known as "locality").

Quiz

- True or False? Abstraction and encapsulation are helpful for the following:

  - ☐ Team development _____
  - ☐ Reusable software _____
  - ☐ GUI programming _____
  - ☐ Easier program maintenance _____

Answer

- True or False? Abstraction and encapsulation are helpful for the following:

  - ☑ Team development _____
  - ☑ Reusable software _____
  - ☐ GUI programming _____
  - ☑ Easier program maintenance _____

UML

- "Unified Modeling Language"
- Not so much a language, but more a process for designing software
- Provides a rigorous way of describing the high-level architecture and design of a software system

Elevator Problem

A product is to be installed to control elevators in a building with m floors. The problem concerns the logic required to move elevators between floors according to the following constraints:

- Each elevator has a set of m buttons, one for each floor. These illuminate when pressed and cause the elevator to visit the corresponding floor. The illumination is canceled when the elevator visits the corresponding floor.

- Each floor, except the first floor and top floor has two buttons, one to request and up-elevator and one to request a down-elevator. These buttons illuminate when pressed. The illumination is canceled when an elevator visits the floor and then moves in the desired direction.

- When an elevator has no requests, it remains at its current floor with its doors closed.

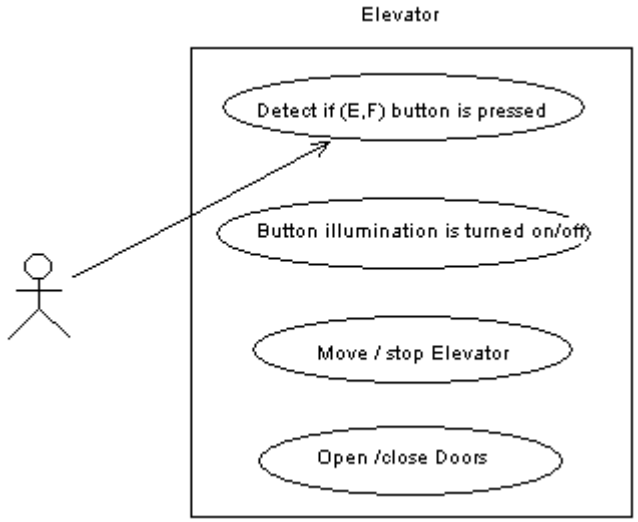
UML Components

UML is a modeling language that only specifies *semantics* and notation

- Use Case Diagram
- Class Diagram
- Sequence Diagram
- Collaboration Diagram
- State Diagram

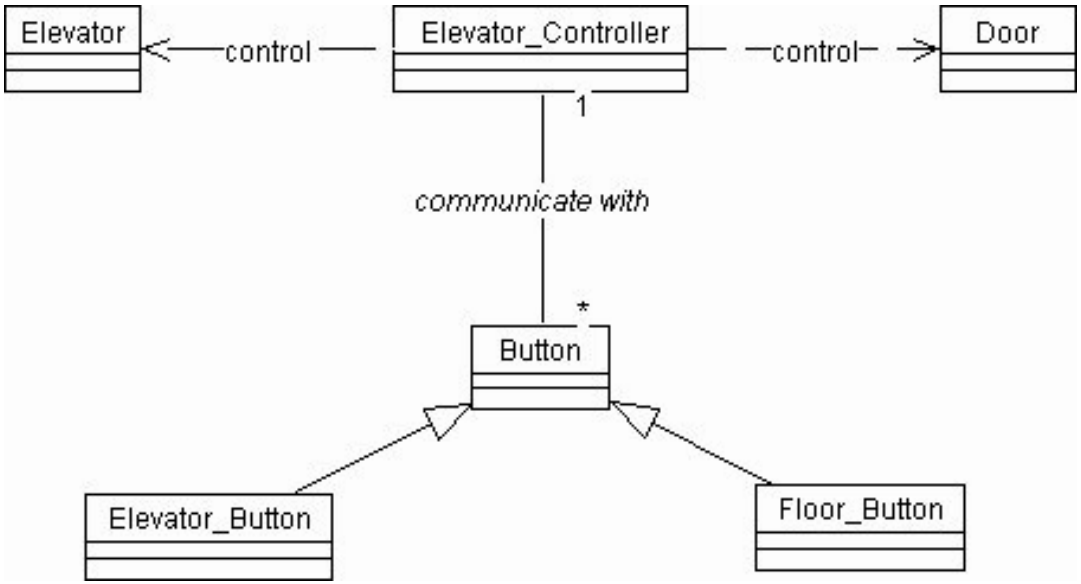
Use Case

- A generalized description of how a system will be used.
- Provides an overview of the intended functionality of the system.
- Understandable by laymen as well as professionals.



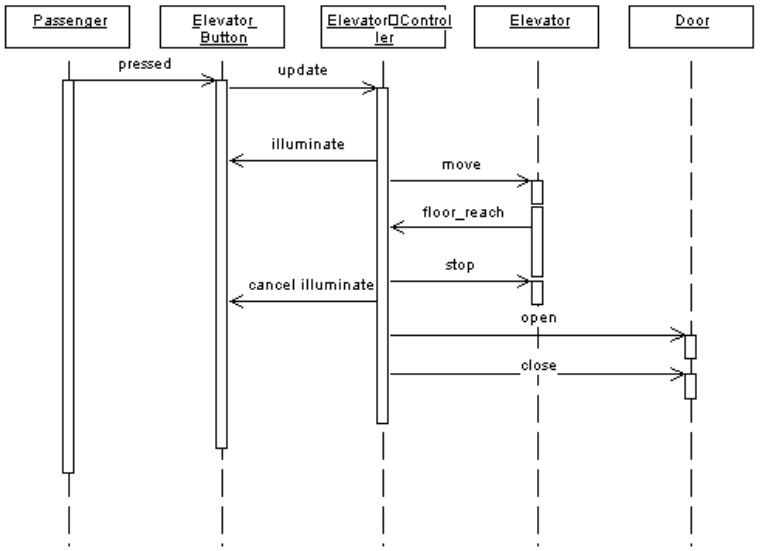
Class Diagram

- Class diagrams show the static structure of the object, their internal structure, and their relationships.



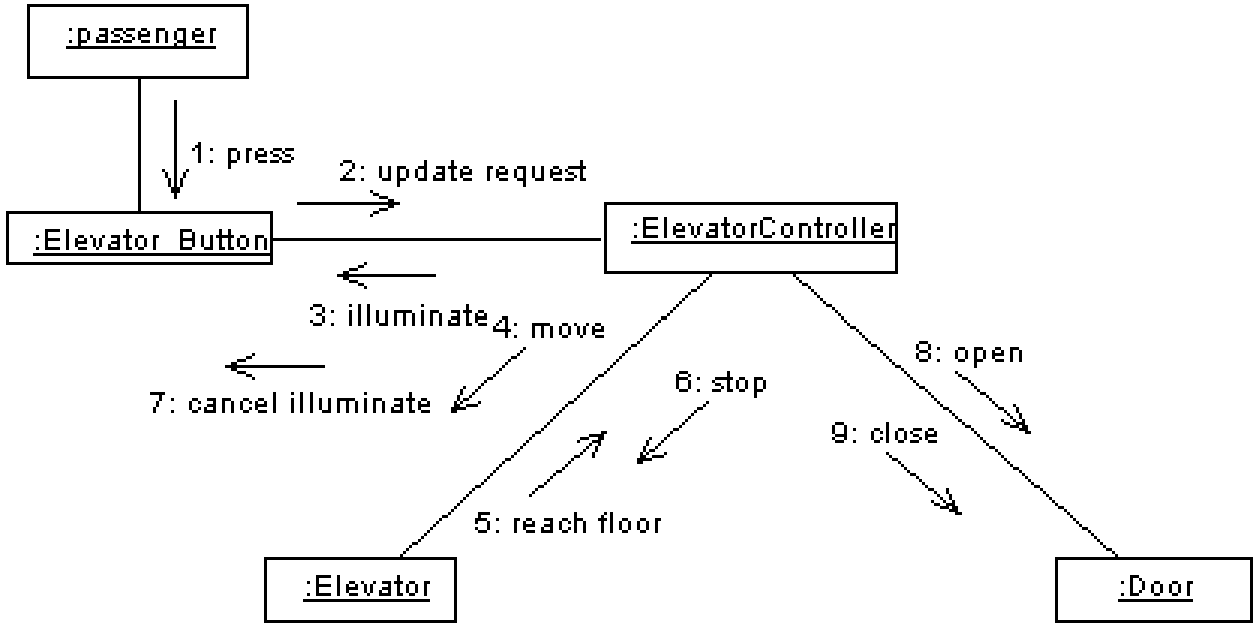
Sequence Diagram

- A sequence diagram and collaboration diagram conveys similar information but expressed in different ways. A Sequence diagram shows the explicit sequence of messages suitable for modeling a real-time system, whereas a collaboration diagram shows the relationships between objects.



Collaboration Diagram

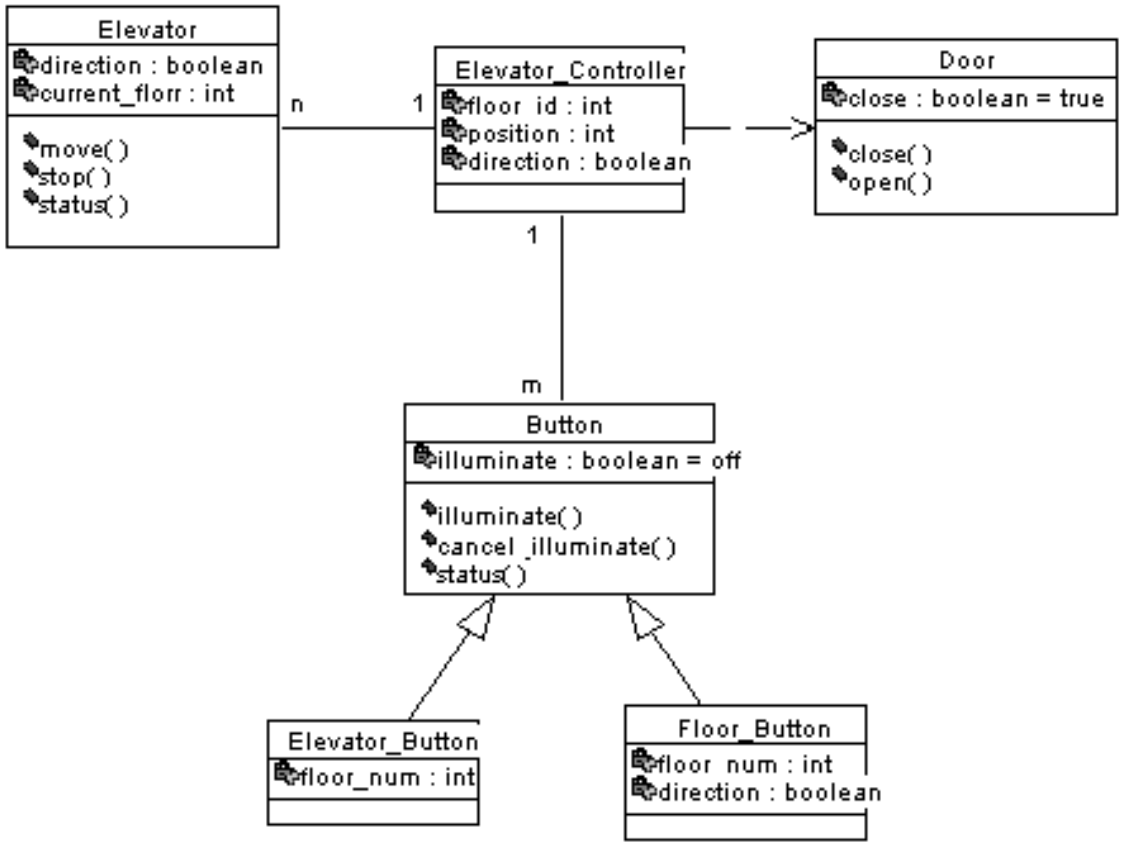
- Describes the set of interactions between classes or types
- Shows the relationships among objects



State Diagram

- A state diagram shows the sequences of states an object goes through during it's life cycle in response to stimuli, together with its responses and actions.

Detail


Elevator
direction : boolean
current_florr : int
move( )
stop( )
status( )
n
1
Elevator_Controller
floor_id : int
position : int
direction : boolean
Door
close : boolean = true
close( )
open( )
1
m
Button
illuminate : boolean = off
illuminate( )
cancel_illuminate( )
status( )
Elevator_Button
floor_num : int
Floor_Button
floor_num : int
direction : boolean

Polymorphism

- We often want to refer to an object by its primary, most specific, data type.
- This is necessary when we call methods specific to this particular type of object:

```
ComputerPlayer player1 = new ComputerPlayer();
HumanPlayer player2 = new HumanPlayer("Nancy", 8);
...
if ( player2.getAge () < 10 )
    player1.setStrategy (new Level1Strategy ());
```

Polymorphism (cont'd)

- But sometimes we want to refer to an object by its inherited, more generic type:

```
Player players[ ] = new Player[2];
players[0] = new ComputerPlayer();
players[1] = new HumanPlayer("Nancy", 8);

game.addPlayer(players[0]);
game.addPlayer(players[1]);
```

Both ComputerPlayer and HumanPlayer implement Player

Polymorphism (cont'd)

- Why disguise an object as a more generic type?
  - To mix different related types in the same collection
  - To pass it to a method that expects a parameter of a more generic type
  - To declare a more generic field (especially in an abstract class) which will be initialized and "specialized" later.

Polymorphism (cont'd)

- Polymorphism ensures that the appropriate method is called for an object of a specific type when the object is disguised as a more generic type:

```
while ( game.notDone() )
{
   players[k].makeMove();
   k = (k + 1) % numPlayers;
}
```

The appropriate makeMove method is called for all players (e.g., for a HumanPlayer and a ComputerPlayer).

Polymorphism (cont'd)

- Good news: polymorphism is already supported in Java — all you have to do is use it properly.
- Polymorphism is implemented using a technique called ***late*** (or ***dynamic***) ***method binding***: which exact method to call is determined at run time.

OO Software Design

- Designing a good OOP application is a daunting task.
- It is largely an art: there are no precise rules for identifying classes, objects, and methods.
- Many considerations determine which classes should be defined and their responsibilities.
- A bad design can nullify all the potential OOP benefits.

OO Design (cont'd)

- A few considerations that determine which classes are defined and their responsibilities:
  - Manageable size
  - Clear limited functionality
  - Potential reuse
  - Support for multiple objects
  - The need to derive from a library class
  - The need to make a listener or to implement a particular interface
  - The need to collect a few data elements in one entity

Review:

- Name the main software development concerns that are believed to be addressed by OOP.
- Explain the dual role of inheritance.
- Can an interface extend another interface? If so, what does it mean?
- Can an interface extend a class? If so, what does it mean?
- Why do you think Java does not allow a class to extend several classes?

Review (cont'd):

- What is abstraction?
- Explain how encapsulation helps in software maintenance.
- Why sometimes objects end up disguised as objects of more generic types?
- What is polymorphism?