CSE201: Advanced Programming

Lecture 15: Unified Modeling Language

Vivek Kumar

Computer Science and Engineering

IIIT Delhi

vivekk@iiitd.ac.in

Last Lecture

- **JUnit unit testing**
- For a given class `Foo`, create another class `FooTest` to test it, containing various "test case" methods to run.
- Each method looks for particular results and passes / fails
- The idea: Put “assert” calls in your test methods to check things you expect to be true. If they aren't, the test will fail
- **Inner classes**
- Favors logical grouping, encapsulation, and readability of code

```
/* Junit test runner class */

import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
   public static void main(String[] args) {
      Result result= JUnitCore.runClasses(MyTest.class);
      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }
      System.out.println(result.wasSuccessful());
   }
}
```

```
/* The class method to be tested */
public class Sum {
    private int var1, var2;
    public Sum(int v1, int v2) {var1=v1; var2=v2;}
    public void incr () {
        var1++; var2++;
    }
    @Override
    public boolean equals(Object o) {
        if(o!=null && getClass()==o.getClass()) {
            Sum s = (Sum) o;
            return ((var1==s.var1)&&(var2==s.var2));
        }
        return false;
    }
    @Override
    public String toString() {
       return "("+Integer.toString(var1)+",”
                      +Integer.toString(var2)+")";
    }
}
```

```
public class SamsungGalaxy {

    private FixedBattery myBattery;
    public SamsungGalaxy() {
        myBattery = new FixedBattery();
    }
    private class FixedBattery {
        private boolean runDiagnosis() { ..... }
        ....
    }
    public static void main(String[] args) {
        SamsungGalaxy sg = new SamsungGalaxy();
        SamsungGalaxy.FixedBattery sgb
                   = sg.new FixedBattery();
        boolean test = sgb.runDiagnosis();
    }
}
```

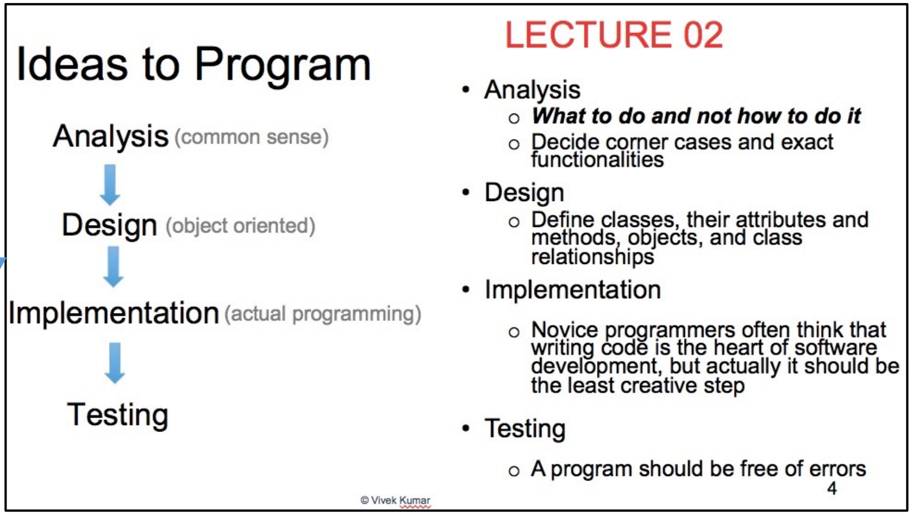
```
/* Junit test class */
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class MyTest {

    @Test
    public void testIncr() {
        Sum mySum = new Sum(1, 1);
        mySum.incr();
        Sum expected = new Sum(3, 3);
        assertEquals(expected, mySum); //should fail
    }
}
```

Today’s Lecture

- Introduction to UML
  - We already covered UML in bits and pieces in prior lectures
    - Sequence diagram (Lecture 2)
    - Representing class relationships (Lectures 3–6)
- Relationships in use case diagrams
- Goal of this lecture is to give you more familiarity with UML
  - You can model 80% of problems by using about 20% UML
  - We will only cover less than 20% here
    - Not possible to teach everything…

What is UML?

- UML stands for Unified Modeling Language
- It's a widely used modeling language in the field of software engineering
- It's used to analyze, design, and implement software-based systems
- Pretty pictures (diagrams)



LECTURE 02

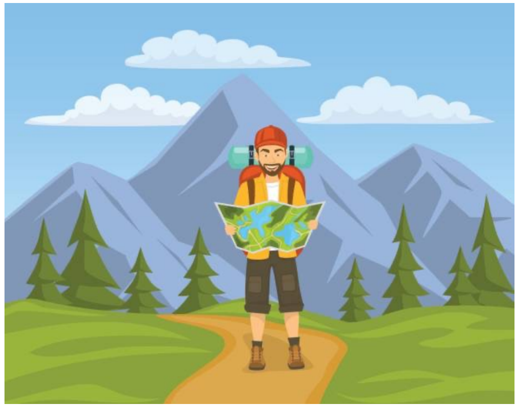
Ideas to Program

Analysis (common sense)
↓
Design (object oriented)
↓
Implementation (actual programming)
↓
Testing

- Analysis
  - ***What to do and not how to do it***
  - Decide corner cases and exact functionalities
- Design
  - Define classes, their attributes and methods, objects, and class relationships
- Implementation
  - Novice programmers often think that writing code is the heart of software development, but actually it should be the least creative step
- Testing
  - A program should be free of errors

© Vivek Kumar

4



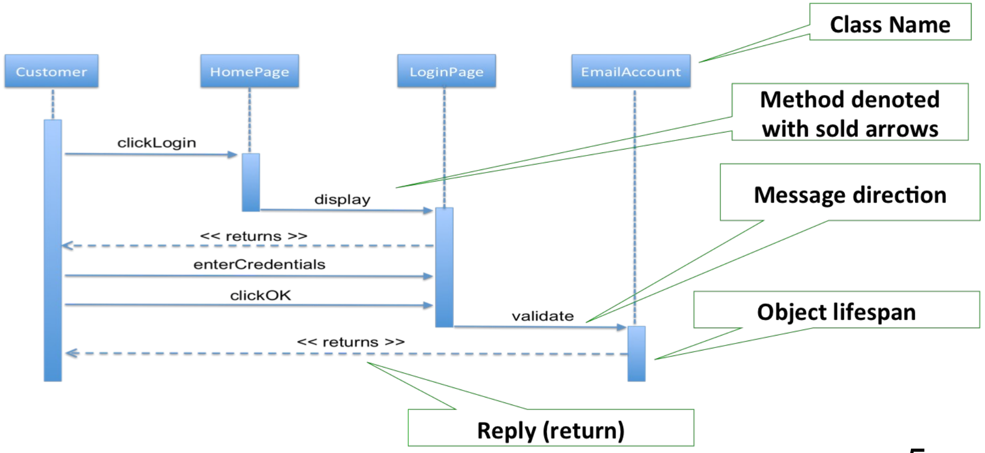
Motivations for UML

- We need a modeling language to:
    - help develop efficient, effective and correct designs, particularly Object Oriented designs
    - communicate clearly with project stakeholders (concerned parties: developers, customer, etc)
    - give us the “big picture” view of the project

UML Diagrams

Three types of UML diagrams that we will cover:

1. **Class diagrams:** Represents static structure
2. **Use case diagrams:** Sequence of actions a system performs to yield an observable result to an actor
3. **Sequence diagrams:** Shows how groups of objects interact in some behavior
   - Already covered in Lecture 02



UML Diagrams: Class Diagrams

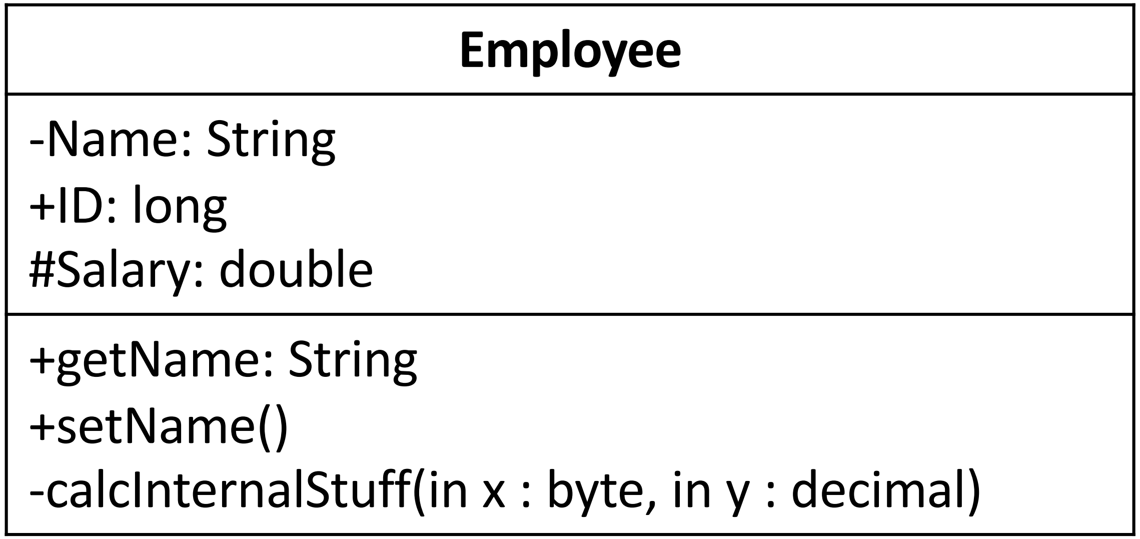
- Better name: “Static structure diagram”
  - Doesn‘t describe temporal aspects
  - Doesn‘t describe individual objects: Only the overall structure of the system
- There are “object diagrams” where the boxes represent instances
  - Rarely used and not covered in this course

UML Class Notation

- A class is a rectangle divided into three parts
  - Class name
  - Class attributes (i.e. data members, variables)
  - Class operations (i.e. methods)

- Modifiers
  - Private: -
  - Public: +
  - Protected: #
  - Static: Underlined

- Abstract class/methods
  - Name in italics

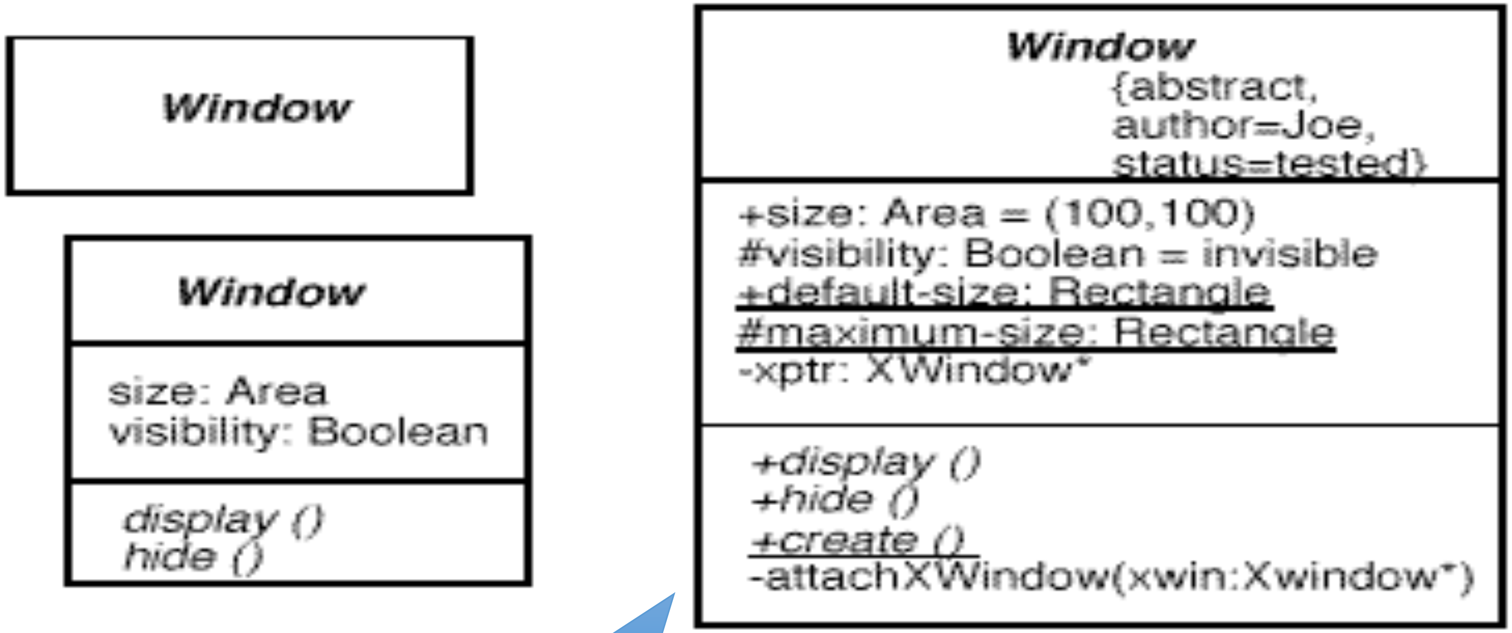


Employee

-Name: String
+ID: long
#Salary: double

+getName: String
+setName()
-calcInternalStuff(in x : byte, in y : decimal)

Different Levels of Specifying Classes



Window

Window

size: Area
visibility: Boolean

display ()
hide ()

Window
{abstract,
author=Joe,
status=tested}

+size: Area = (100,100)
#visibility: Boolean = invisible
+default-size: Rectangle
#maximum-size: Rectangle
-xptr: XWindow*

+display ()
+hide ()
+create ()
-attachXWindow(xwin:Xwindow*)

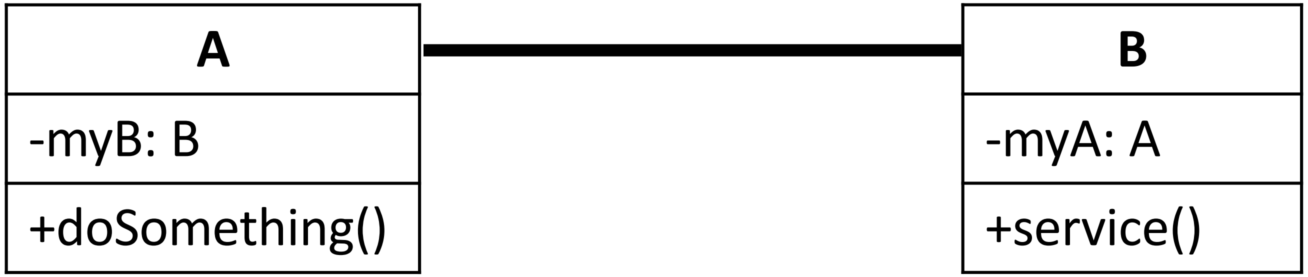
Use this for your project

Class Relationships

- UML diagrams for these class relationships are already covered before (Lectures 04, 05 and 08)
  - Association
  - Composition
  - Dependency
  - Inheritance
- We will only cover binary association relationship here

Class Relationship: Binary Association

Both entities “Knows About” each other (two-way association)



A

-myB: B

+doSomething()

B

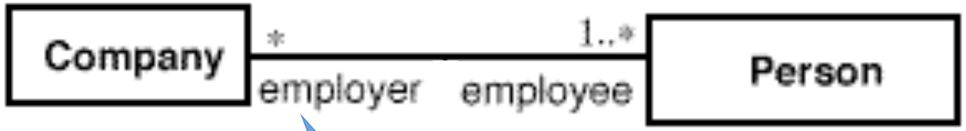
-myA: A

+service()

UML Multiplicities

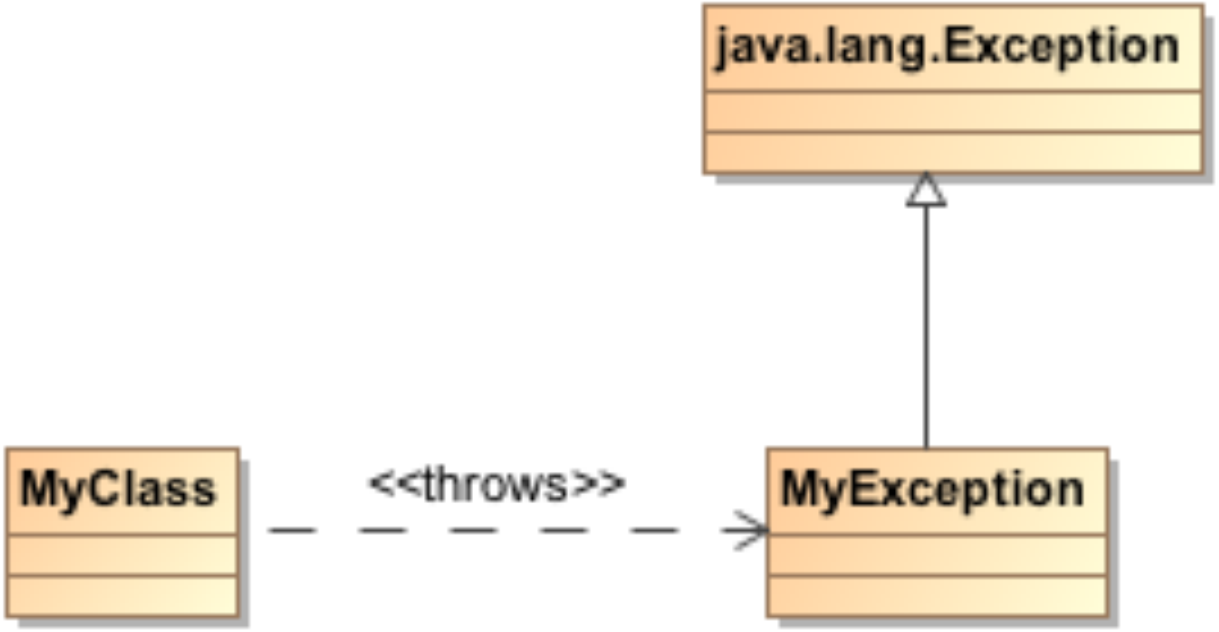
Links on associations to specify more details about the relationship

Multiplicities	Meaning
0..1	zero or one instance. The notation "*n . . M*" indicates *n* to *m* instances.
0..* *or* *	no limit on the number of instances (including none).
1	exactly one instance
1..*	at least one instance

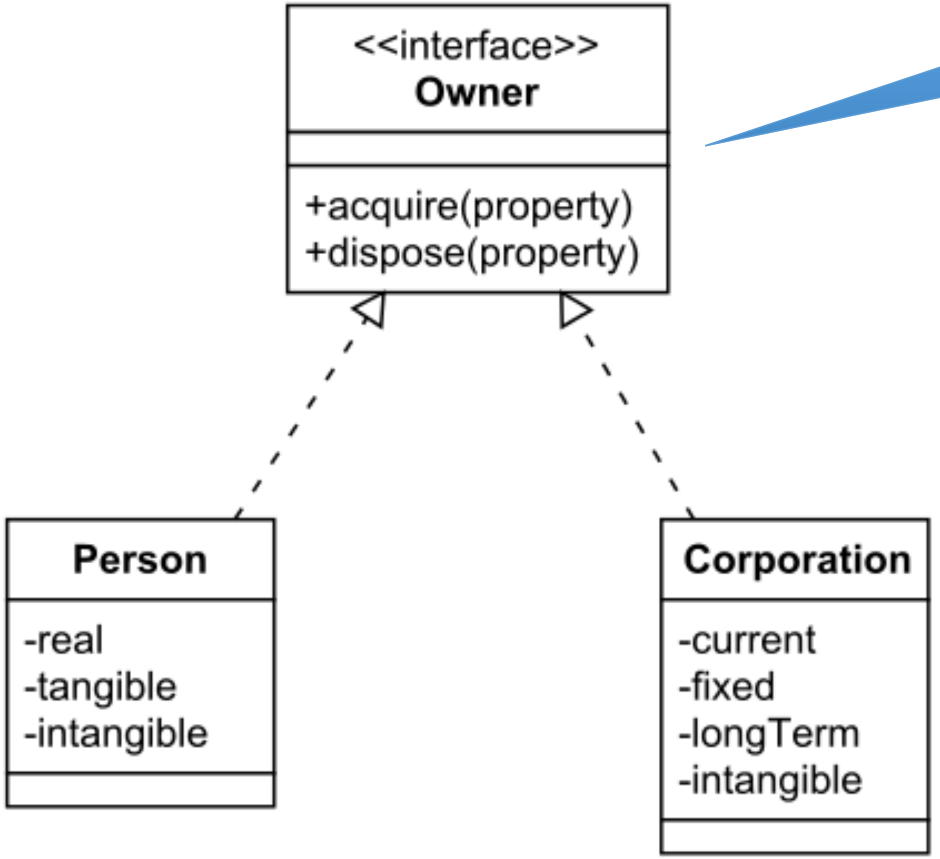


How you will implement?

Exceptions

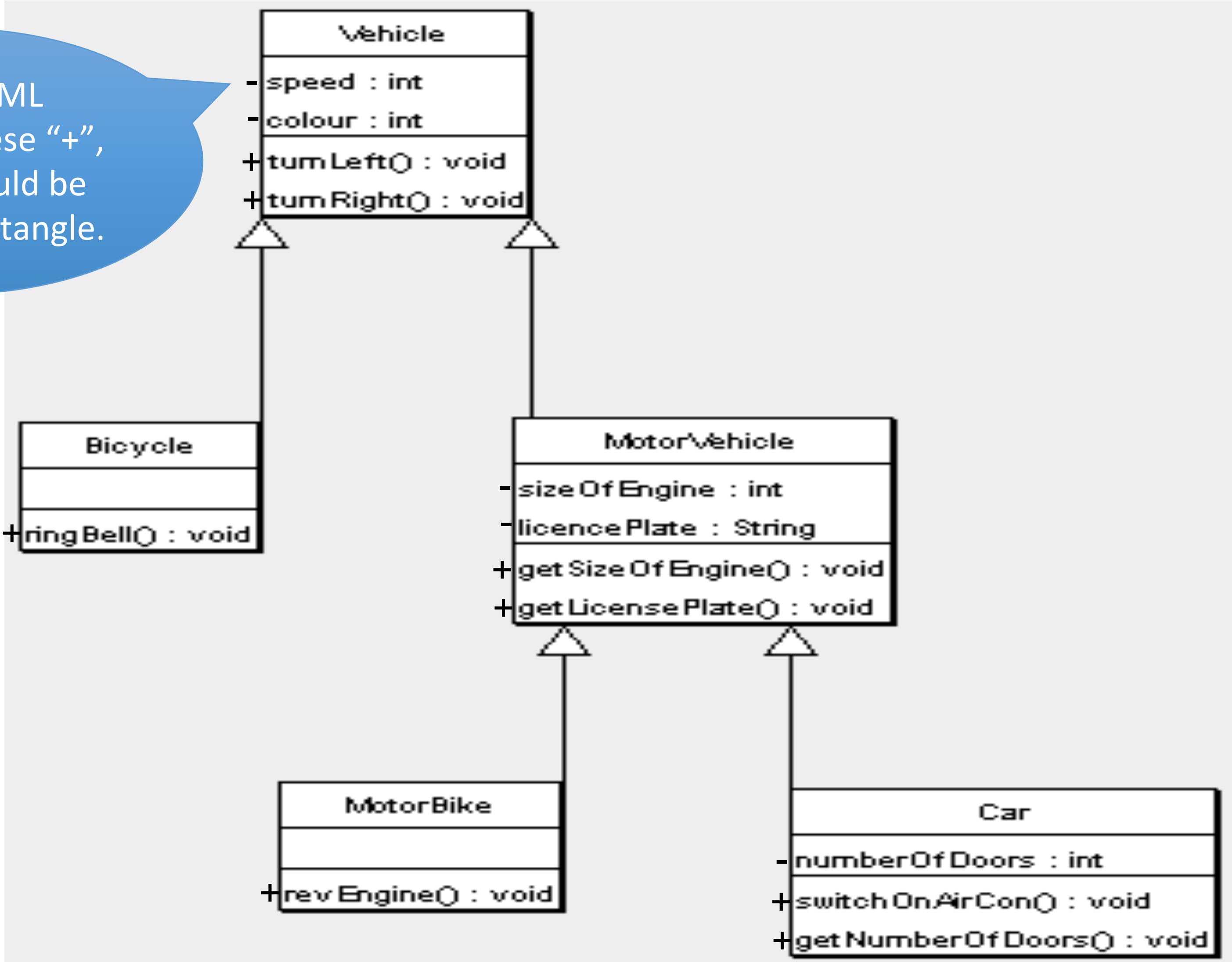


Interfaces



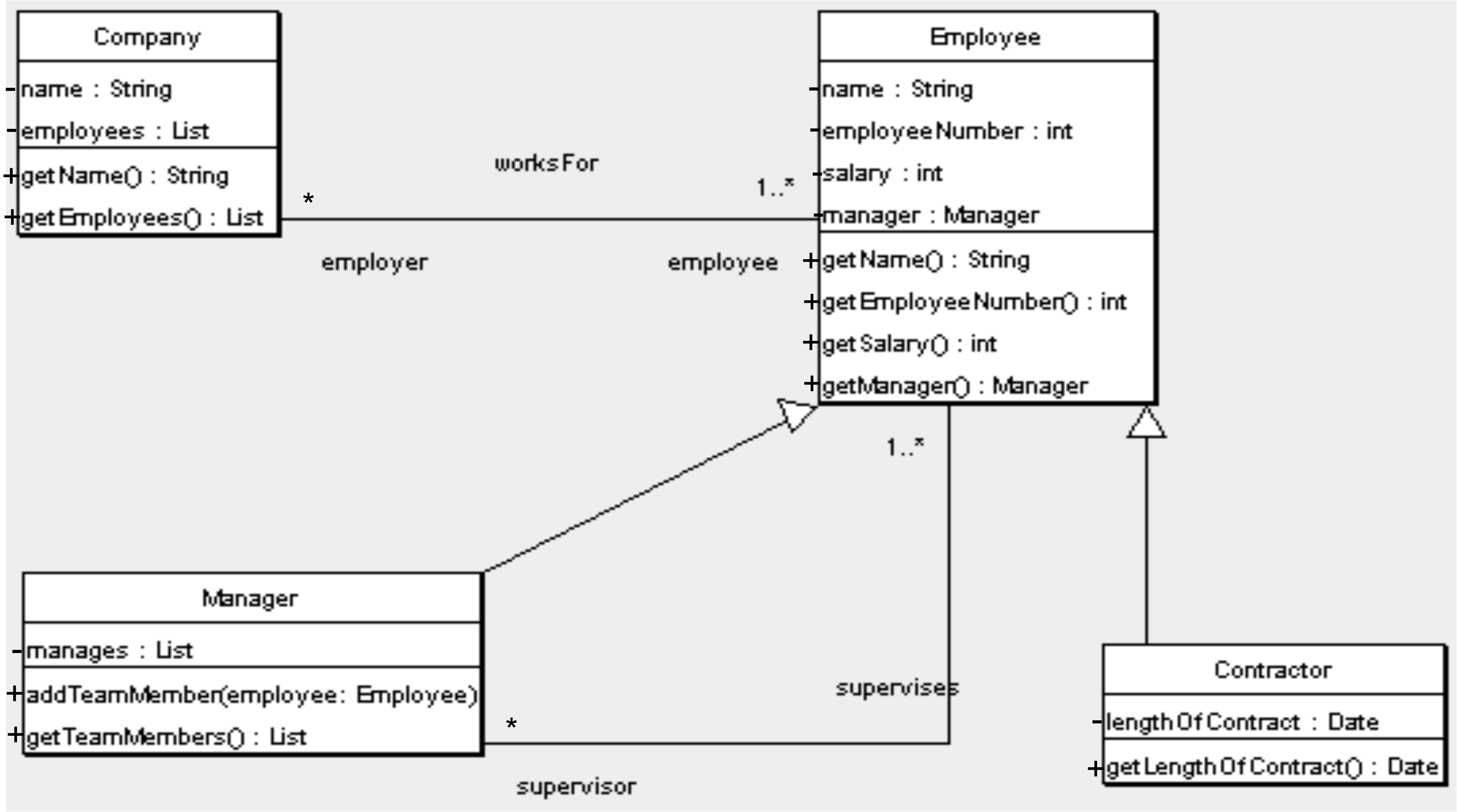
How is this diagram different from that of a class ?

Sample Class Diagram (1/2)



In your UML diagrams, these "+", "-", etc, should be inside the rectangle.

Sample Class Diagram (2/2)

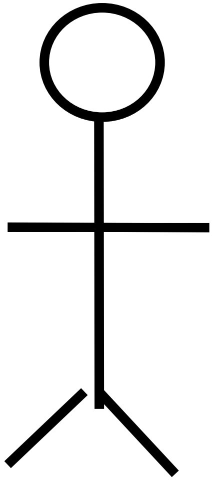


UML Diagrams: Use Cases

- Means of capturing requirements
  - Used at a very early phase of software development for requirement gathering (analysis phase)
  - Provides a high level overview of the system
  - Class diagrams are created after generating use case diagrams
- Document interactions between user(s) and the system
  - User (actor) is not part of the system itself
  - But an actor can be *another* system
- A scenario based technique in UML
- **Use case diagrams** describe what a system does from the standpoint of an external observer. The emphasis is on *what* a system does rather than *how*

Actors in Use Case

- What is an Actor?
  - A user or outside system that interacts with the system being designed in order to obtain some value from that interaction
  - It can be a:
    - Human
    - Peripheral device (hardware)
    - External system or subsystem
    - Time or time-based event
  - Labelled using a descriptive noun or phrase
  - Represented by stick figure



Use Case Analysis (1/4)

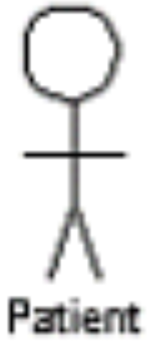
- Sample scenario
  - *“A patient calls the clinic to make an appointment for a yearly checkup. The receptionist finds the nearest empty time slot in the appointment book and schedules the appointment for that time slot"*
- We want to write a use case for this scenario

Source: http://www.cs.fsu.edu/~baker/swe1/restricted/notes/ppt/UseCaseDiagrams.ppt

Use Case Analysis (2/4)

- Sample scenario
  - *"A patient calls the clinic to make an appointment for a yearly checkup. The receptionist finds the nearest empty time slot in the appointment book and schedules the appointment for that time slot"*

- Who is the actor?
  - The actor is a "Patient" here



Source: http://www.cs.fsu.edu/~baker/swe1/restricted/notes/ppt/UseCaseDiagrams.ppt

Use Case Analysis (3/4)

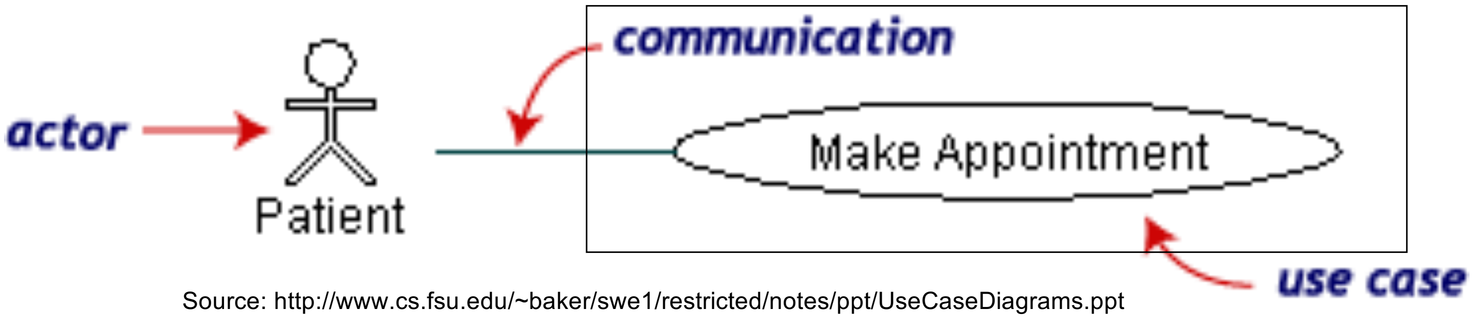
- Sample scenario
  - “A *patient calls the clinic to make an appointment for a yearly checkup. The receptionist finds the nearest empty time slot in the appointment book and schedules the appointment for that time slot”*

- A **use case** is a summary of scenarios for a single task or goal
  - So, what is the use case here?
  - The use case is “Make Appointment”

Source: http://www.cs.fsu.edu/~baker/swe1/restricted/notes/ppt/UseCaseDiagrams.ppt

Use Case Analysis (4/4)

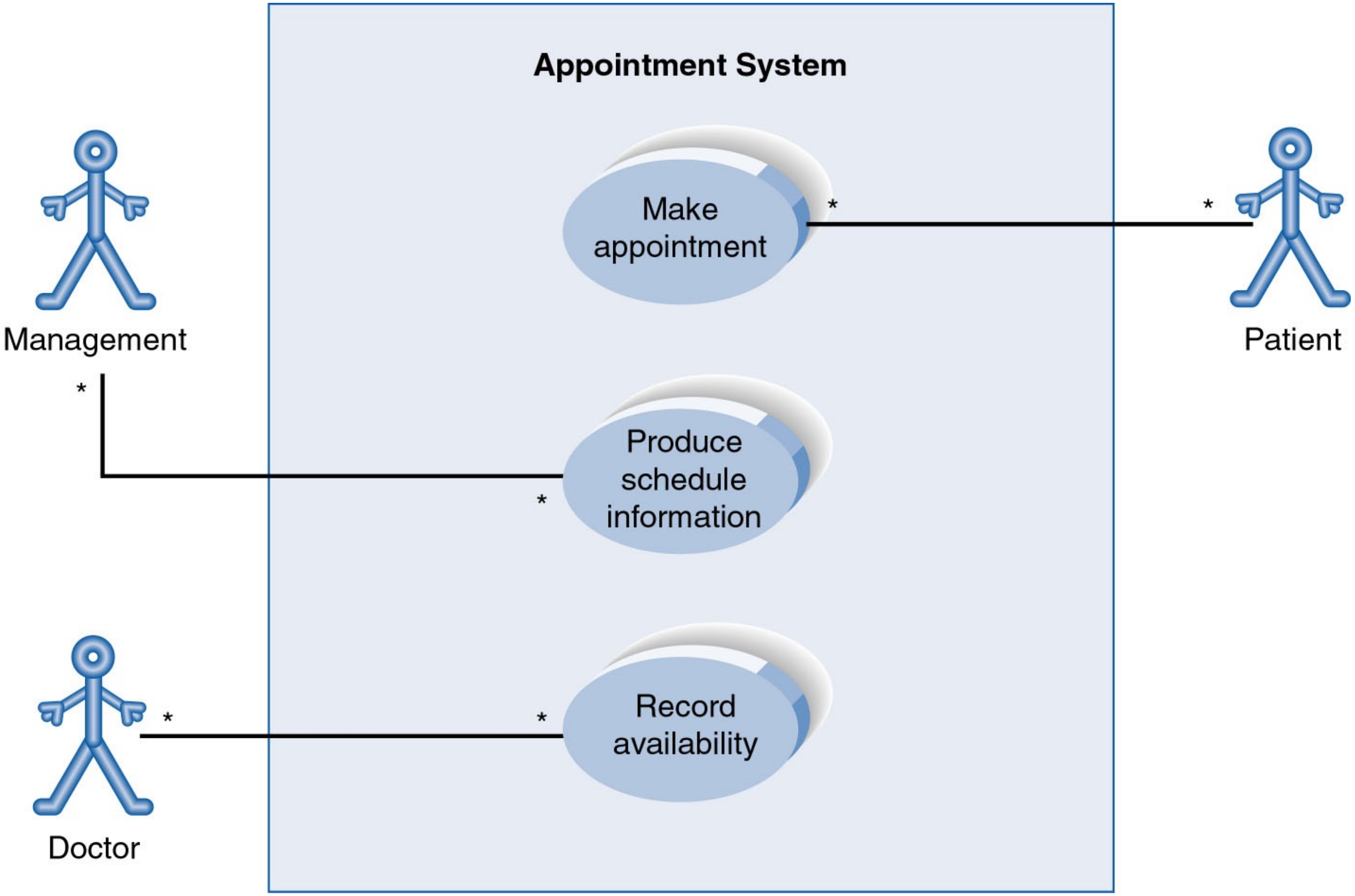
- The picture below is a **Make Appointment** use case for the medical clinic.
- The actor is a **Patient**. The connection between actor and use case is a **communication**
- Actors are stick figures
- Use cases are ovals
  - Labelled using a descriptive verb-noun phrase
- Communications are lines that link actors to use cases
- Boundary rectangle is placed around the perimeter of the system to show how the actors communicate with the system



Source: http://www.cs.fsu.edu/~baker/swe1/restricted/notes/ppt/UseCaseDiagrams.ppt

Use Case Diagram

- A use case diagram is a collection of actors, use cases, and their communications



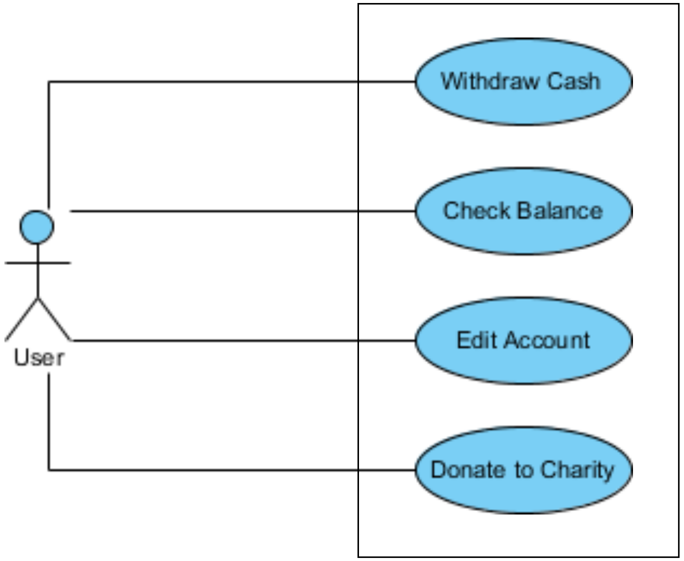
Source: http://www.cs.fsu.edu/~baker/swe1/restricted/notes/ppt/UseCaseDiagrams.ppt

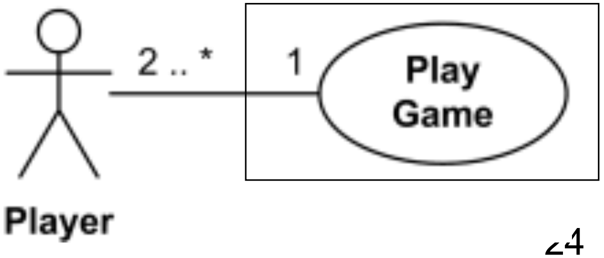
Relationships for Use Cases

- Association
- Generalization
- Extend
- Include

Association Relationship

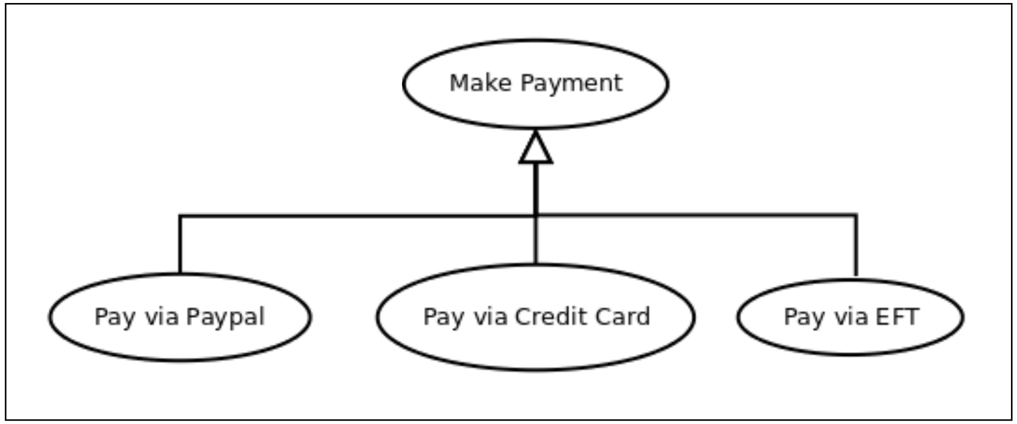
- Exists only between an actor and a use case
  - Indicates that an actor can use certain functionality of the system
- Represented by a sold line without arrowhead
  - Most commonly used representation
  - Uncommon to show one-way association
- The association between an actor and a use case can also show multiplicity at each end

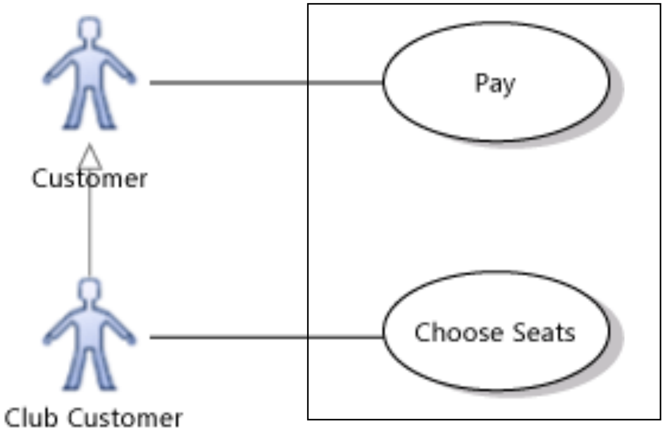




Generalization Relationship

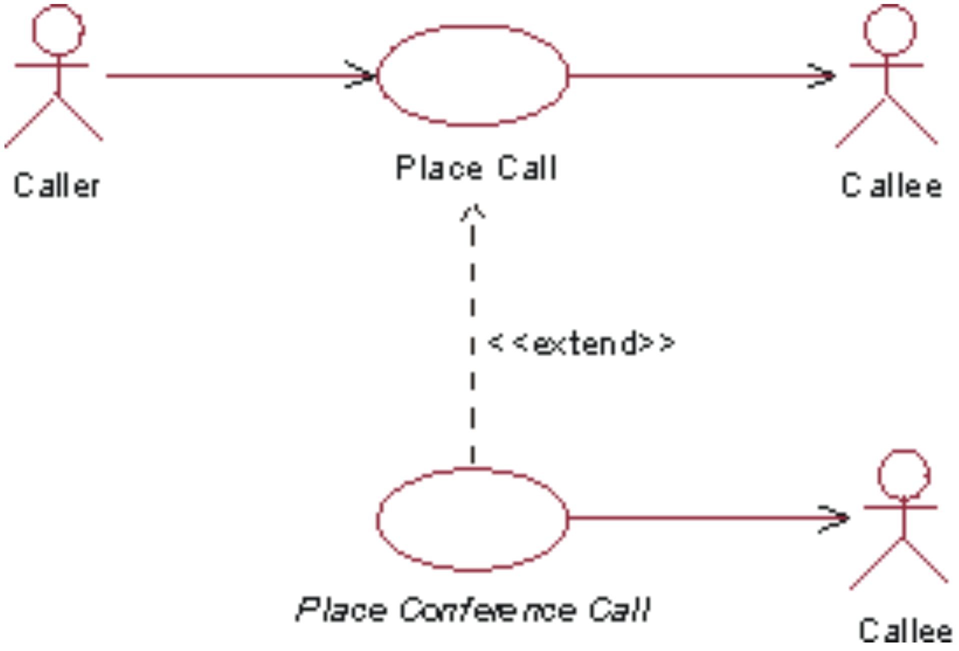
- Could exit between two actors or between two use cases
  - Indicates parent/child relationship
- Represented by a solid line with a triangular and hollow arrowhead
  - From child to parent





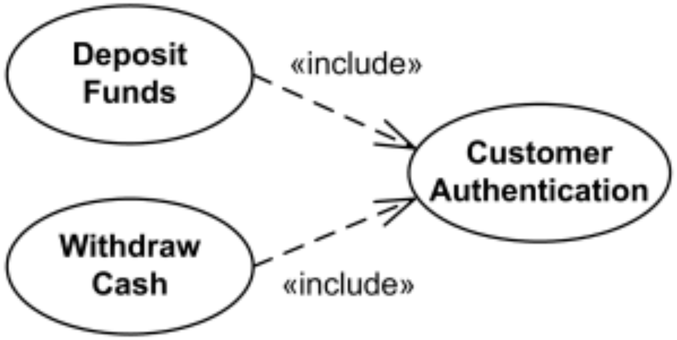
Extend Relationship “<<extend>”

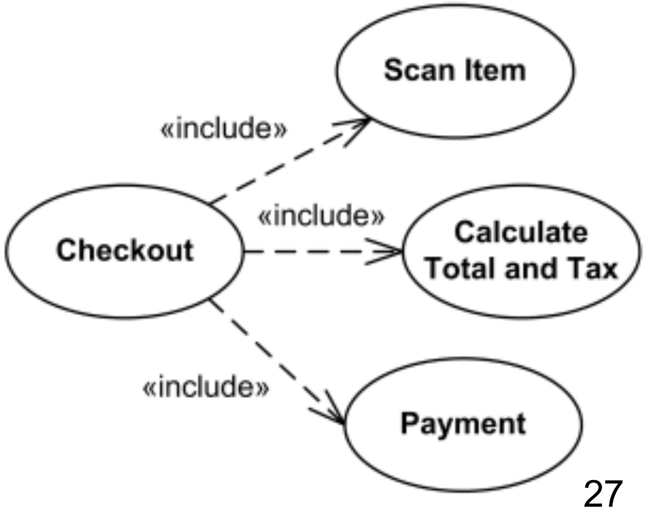
- Exists only between use cases
  - This relationships represent optional or seldom invoked cases
  - Indicates that although one use case is a variation of another but it is invoked rarely
    - Lot of shared code between these use cases **(not to be confused with inheritance)**
- Represented using a dashed arrow with an arrowhead. The notation “<< extend >>” is also mentioned above the arrow
  - The direction of the arrow is toward the extended use cases



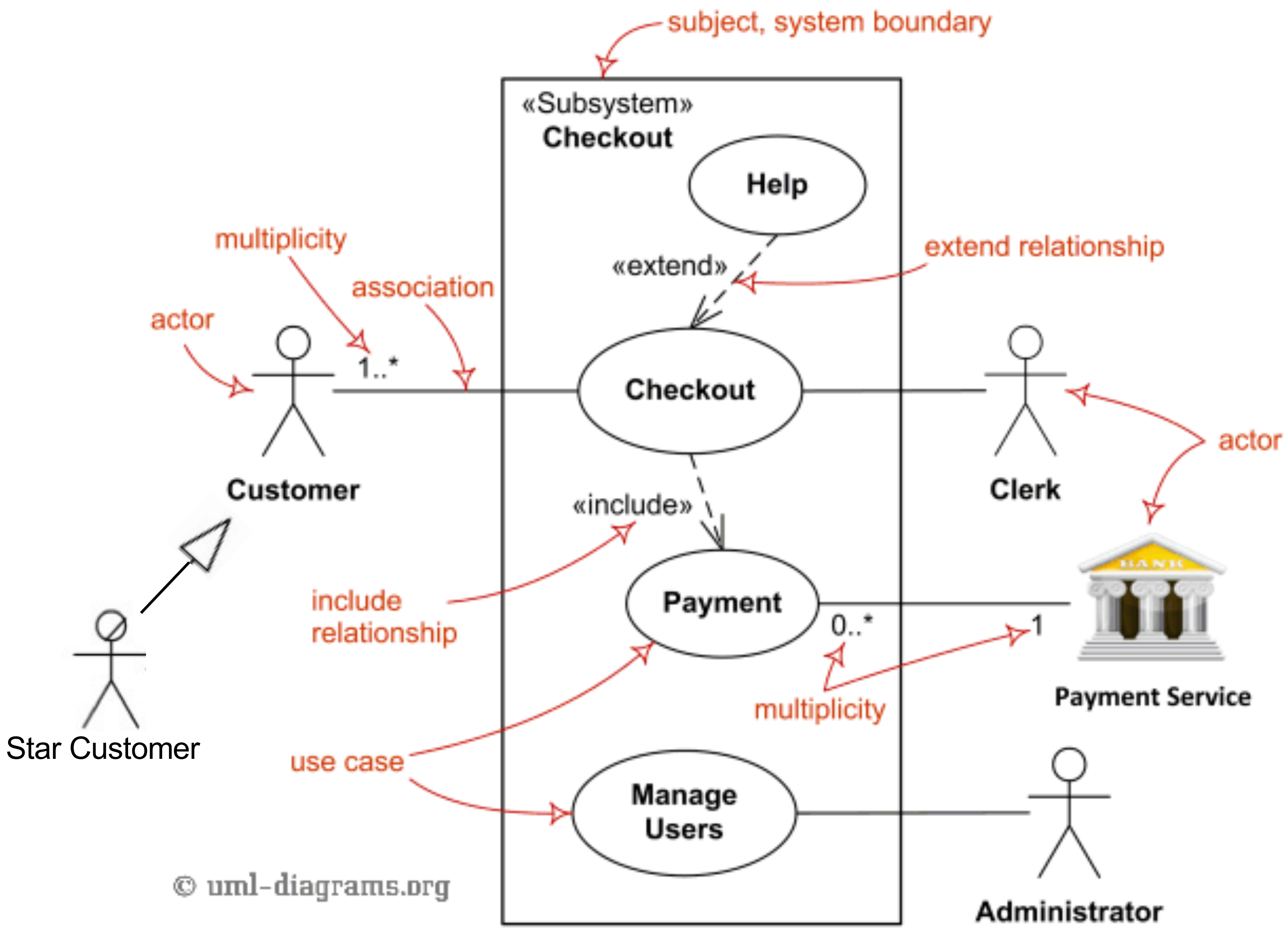
Include Relationship “<<include>”

- Exists only between use cases
  - Represents behavior that is factored out of the use case
  - Doesn’t mean that the factored out use case is an optional or seldom invoked cases
- Represented using a dashed arrow with an arrowhead. The notation “<< include>>” is also mentioned above the arrow
  - The direction of the arrow is toward the included use case





Sample Use Case



Next Lecture

- Event driven programming using JavaFX