15CSE202 Object Oriented Programming Lecture 9

OO Design with Class Diagram

Nalinadevi Kadiresan

CSE Dept.

Amrita School of Engg.

Beginnings of a Method

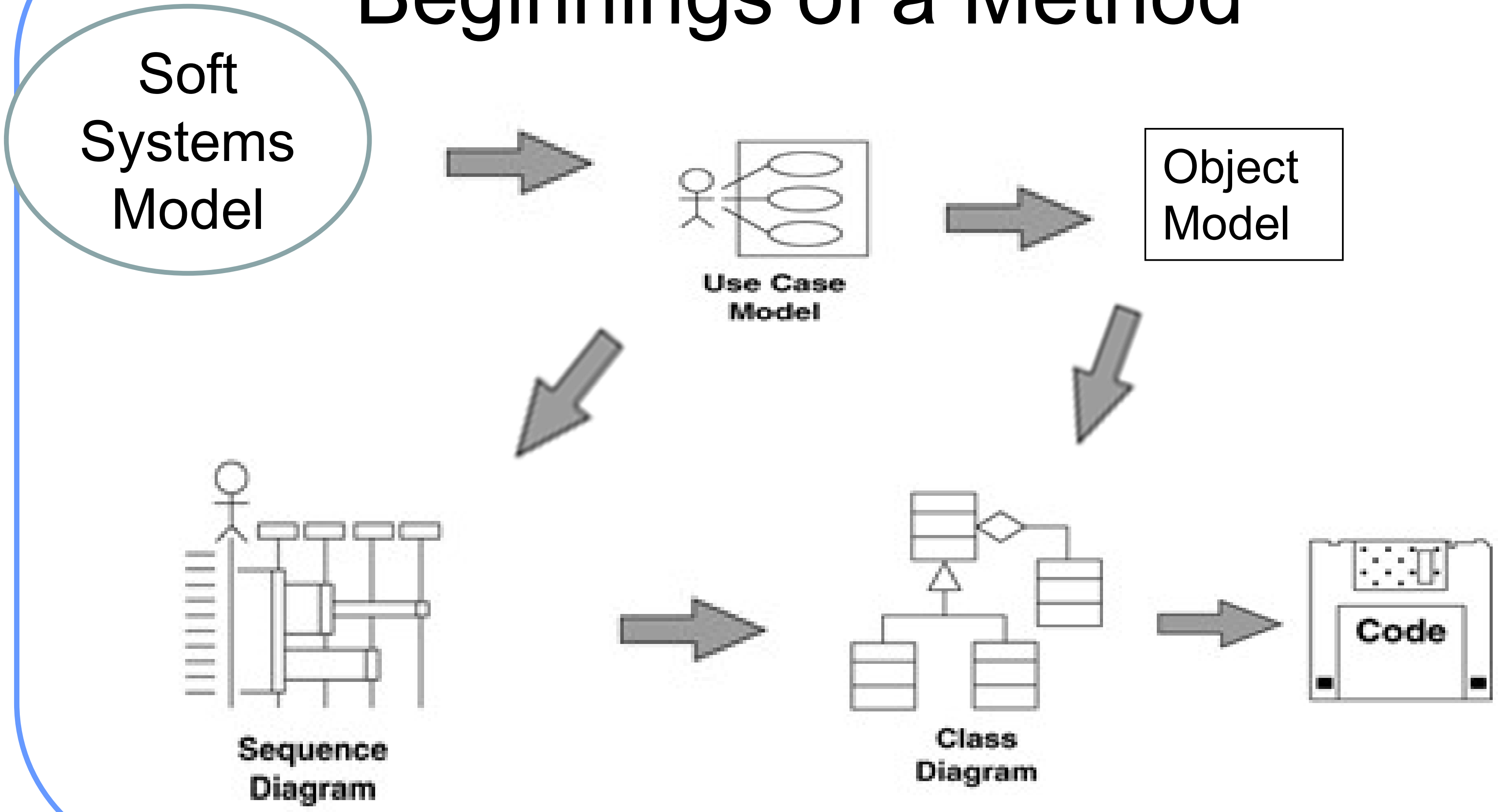
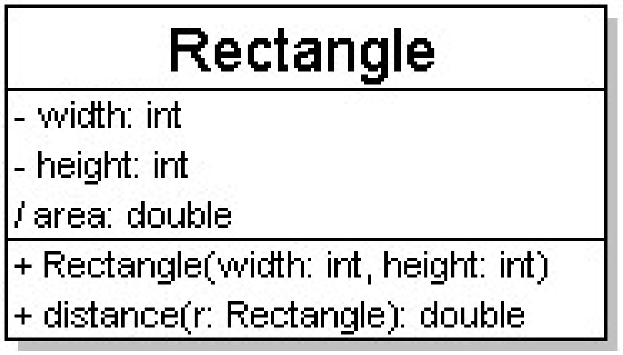
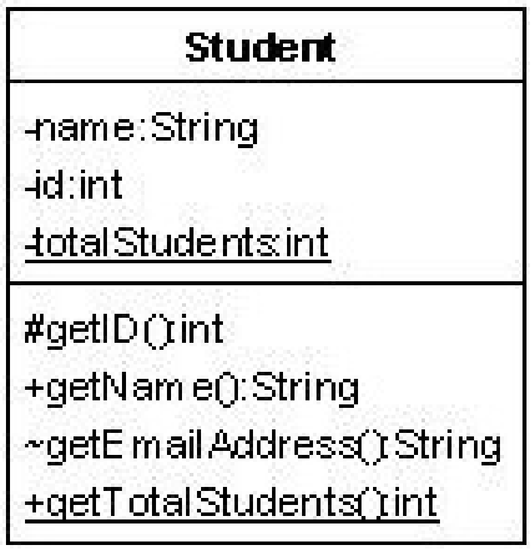


Diagram of one class

- class name in top of box
  - write <<interface>> on top of interfaces' names
  - use *italics* for an *abstract class* name
- attributes (optional)
  - should include all fields of the object
- operations / methods (optional)
  - may omit trivial (get/set) methods
    - but don't omit any methods from an interface!
  - should not include inherited methods


Rectangle
- width: int
- height: int
/ area: double
+ Rectangle(width: int, height: int)
+ distance(r: Rectangle): double

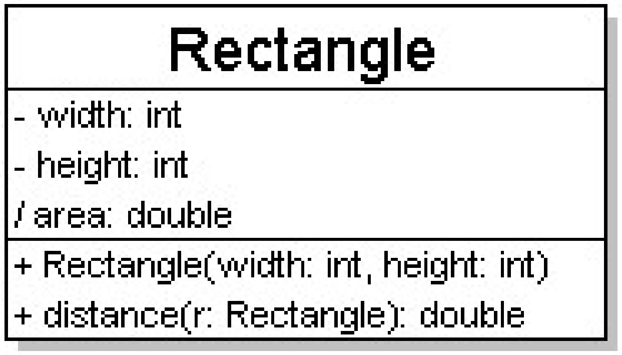


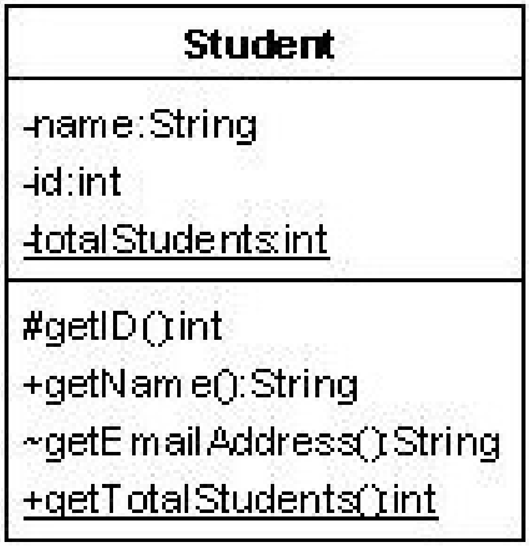
Class attributes

- attributes (fields, instance variables)
  - *visibility name* : *type* [*count*] = *default_value*
  - visibility:

+	public
#	protected
-	private
~	package (default)
/	derived

  - underline <u>static attributes</u>
  - **derived attribute**: not stored, but can be computed from other attribute values
  - attribute example:
    - balance : double = 0.00


Rectangle
- width: int
- height: int
/ area: double
+ Rectangle(width: int, height: int)
+ distance(r: Rectangle): double

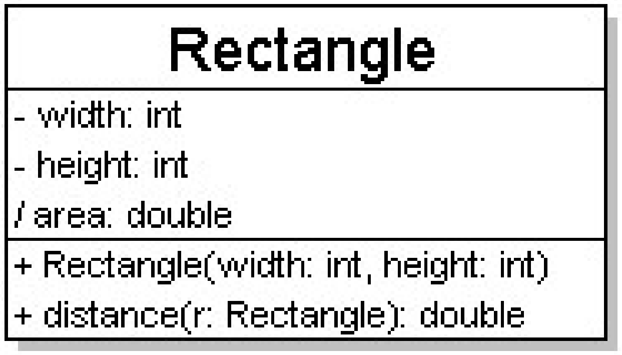


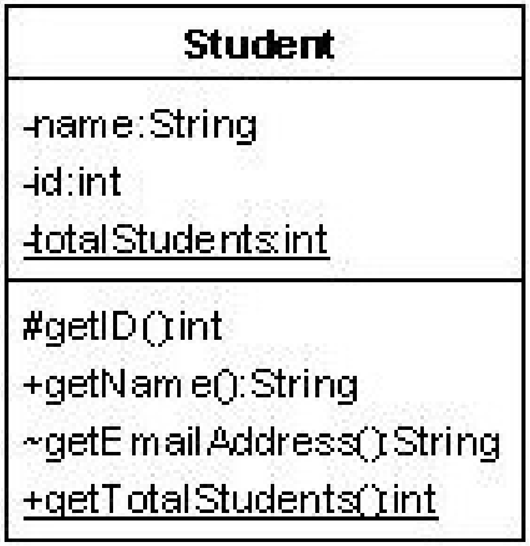
Class operations / methods

- operations / methods
  - *visibility name* (*parameters*) : *return_type*
  - visibility:

visibility:	+	public
	#	protected
	-	private
	~	package (default)

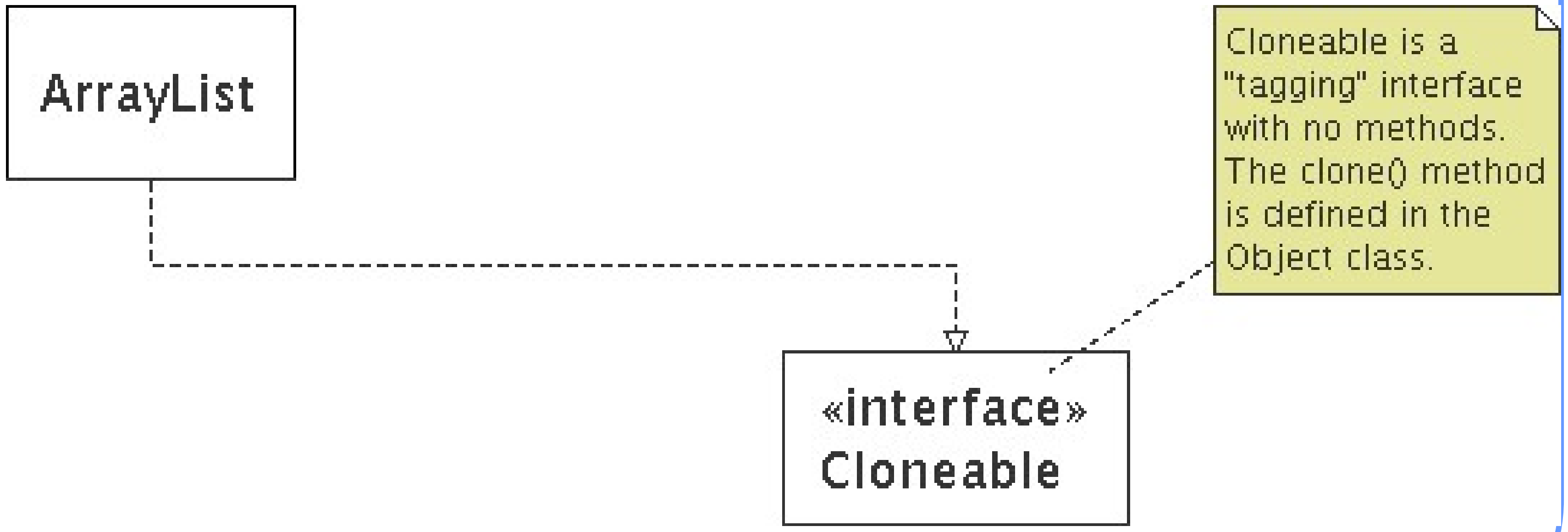
  - underline <u>static methods</u>
  - parameter types listed as (name: type)
  - omit *return_type* on constructors and when return type is void
  - method example:
    + distance(p1: Point, p2: Point): double


Rectangle
- width: int
- height: int
/ area: double
+ Rectangle(width: int, height: int)
+ distance(r: Rectangle): double

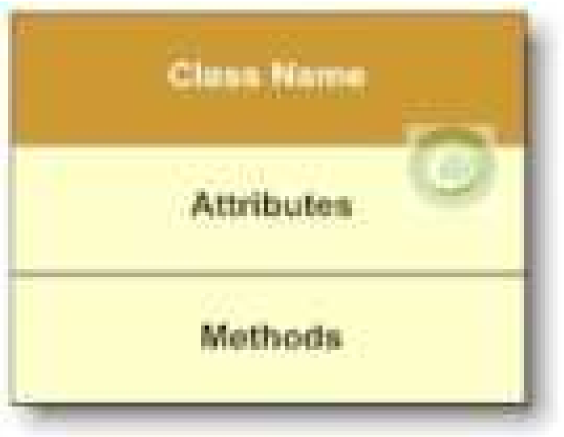


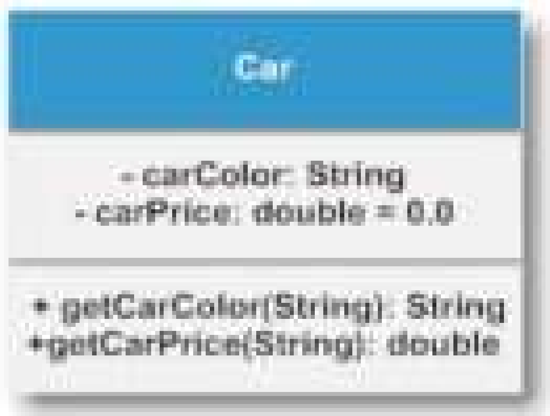
Comments

- represented as a folded note, attached to the appropriate class/method/etc by a dashed line


ArrayList
«interface»
Cloneable
Cloneable is a "tagging" interface with no methods. The clone() method is defined in the Object class.

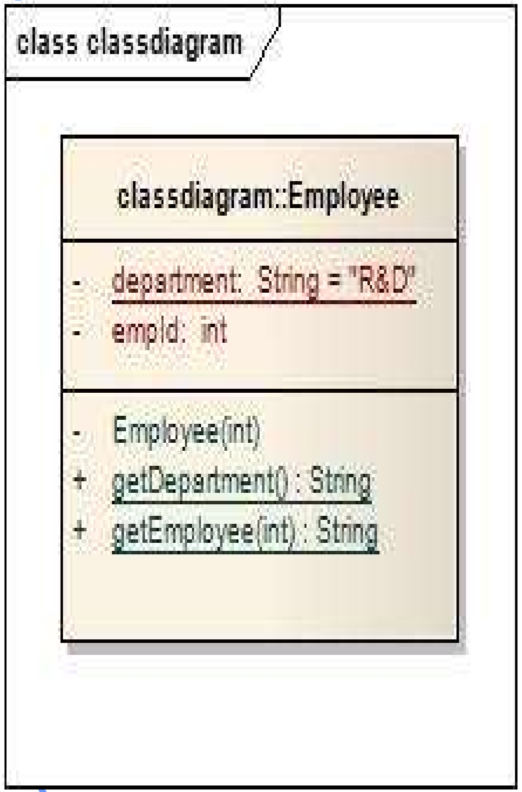
Concrete class and its Java code





```
public class Car {
    private String carColor;
    private double carPrice = 0.0;
    public String getCarColor(String model) {  return carColor;}
    public double getCarPrice(String model) { return carPrice;}
}
```

Concrete Class and its Java code


class classdiagram
classdiagram::Employee
- department: String = "R&D"
- empId: int
- Employee(int)
+ getDepartment() : String
+ getEmployee(int) : String

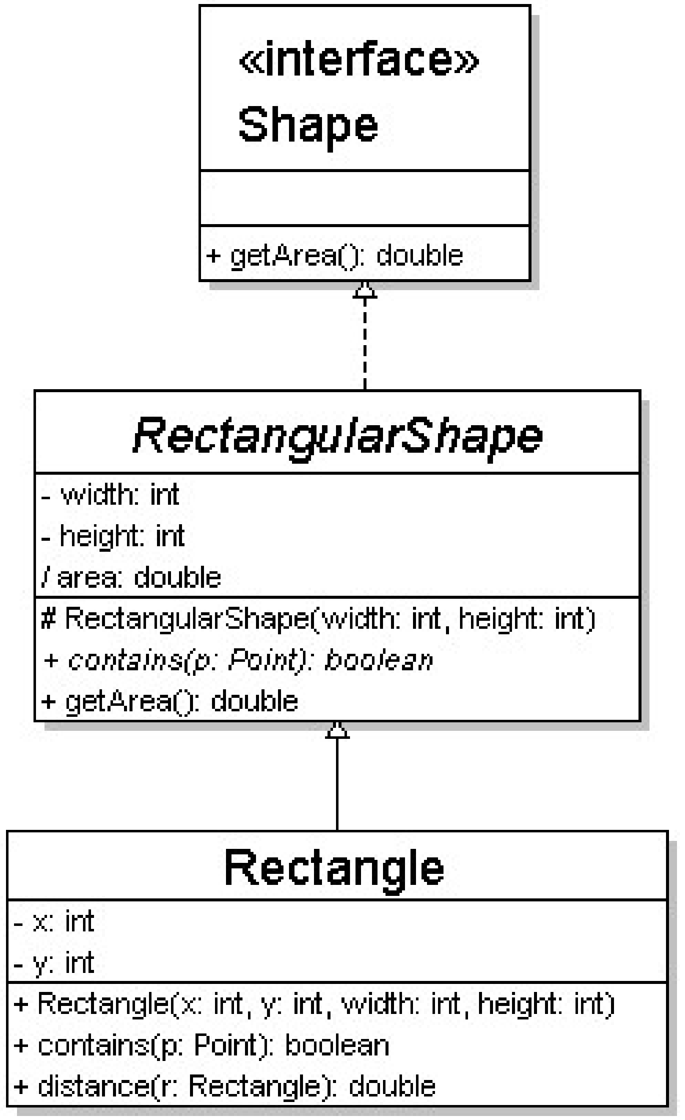
```
public class Employee {
        private static String department = "R&D";
        private int empId;

        private Employee(int employeeId) { this.empId =
            employeeId; }
        public static String getEmployee(int emplId) {
                if (emplId == 1) {
                return "idiotechie";
                } else {
                return "Employee not found";
                }
        }
        public static String getDepartment() {  return
            department;   }
}
```

Relationships btwn. classes

- **generalization**: an inheritance relationship
  - inheritance between classes
  - interface implementation
- **association**: a usage relationship
  - dependency
  - aggregation
  - composition

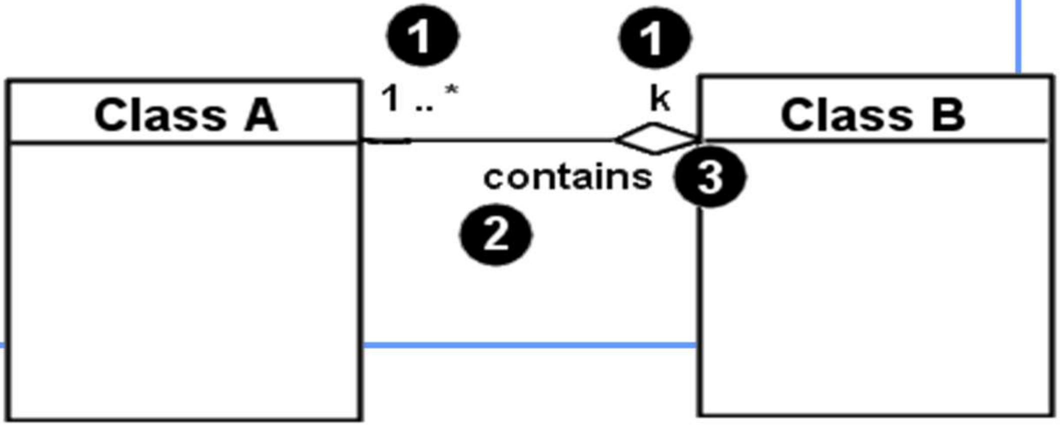
Generalization relationships

- generalization (inheritance) relationships
  - hierarchies drawn top-down with arrows pointing upward to parent
  - line/arrow styles differ, based on whether parent is a(n):
    - class: solid line, black arrow
    - abstract class: solid line, white arrow
    - interface: dashed line, white arrow
  - we often don't draw trivial / obvious generalization relationships, such as drawing the Object class as a parent


«interface»
Shape
+ getArea(): double
RectangularShape
- width: int
- height: int
/ area: double
# RectangularShape(width: int, height: int)
+ contains(p: Point): boolean
+ getArea(): double
Rectangle
- x: int
- y: int
+ Rectangle(x: int, y: int, width: int, height: int)
+ contains(p: Point): boolean
+ distance(r: Rectangle): double

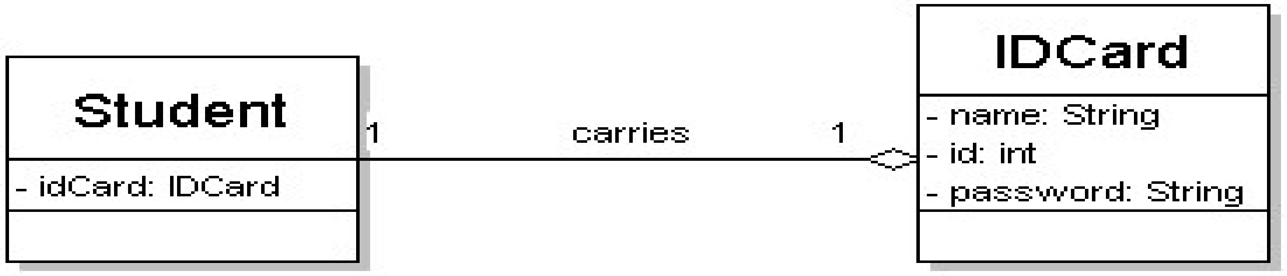
Associational relationships

- associational (usage) relationships
  1. multiplicity (how many are used)
     - * ⇒ 0, 1, or more
     - 1 ⇒ 1 exactly
     - 2..4 ⇒ between 2 and 4, inclusive
     - 3..* ⇒ 3 or more
  2. name (what relationship the objects have)
  3. navigability (direction)

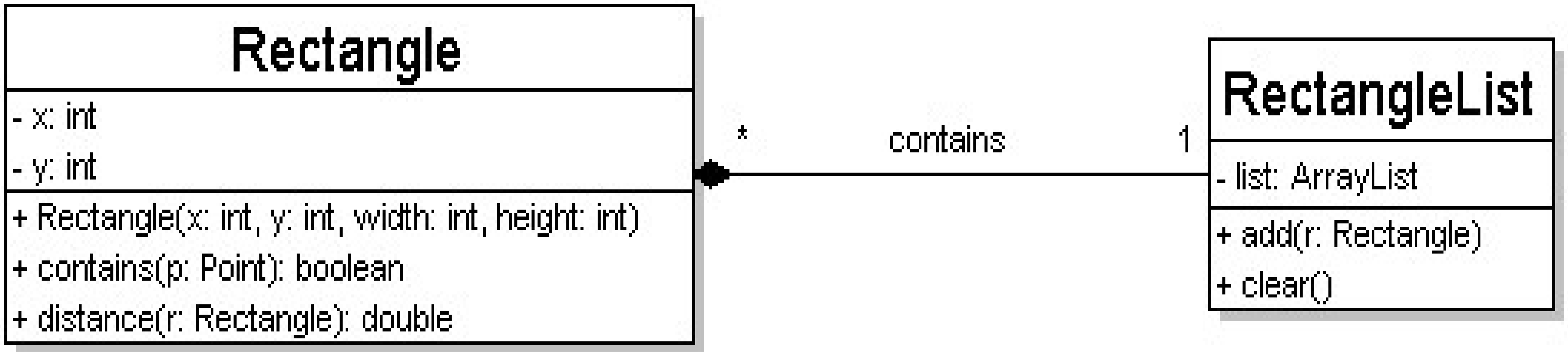


Multiplicity of associations

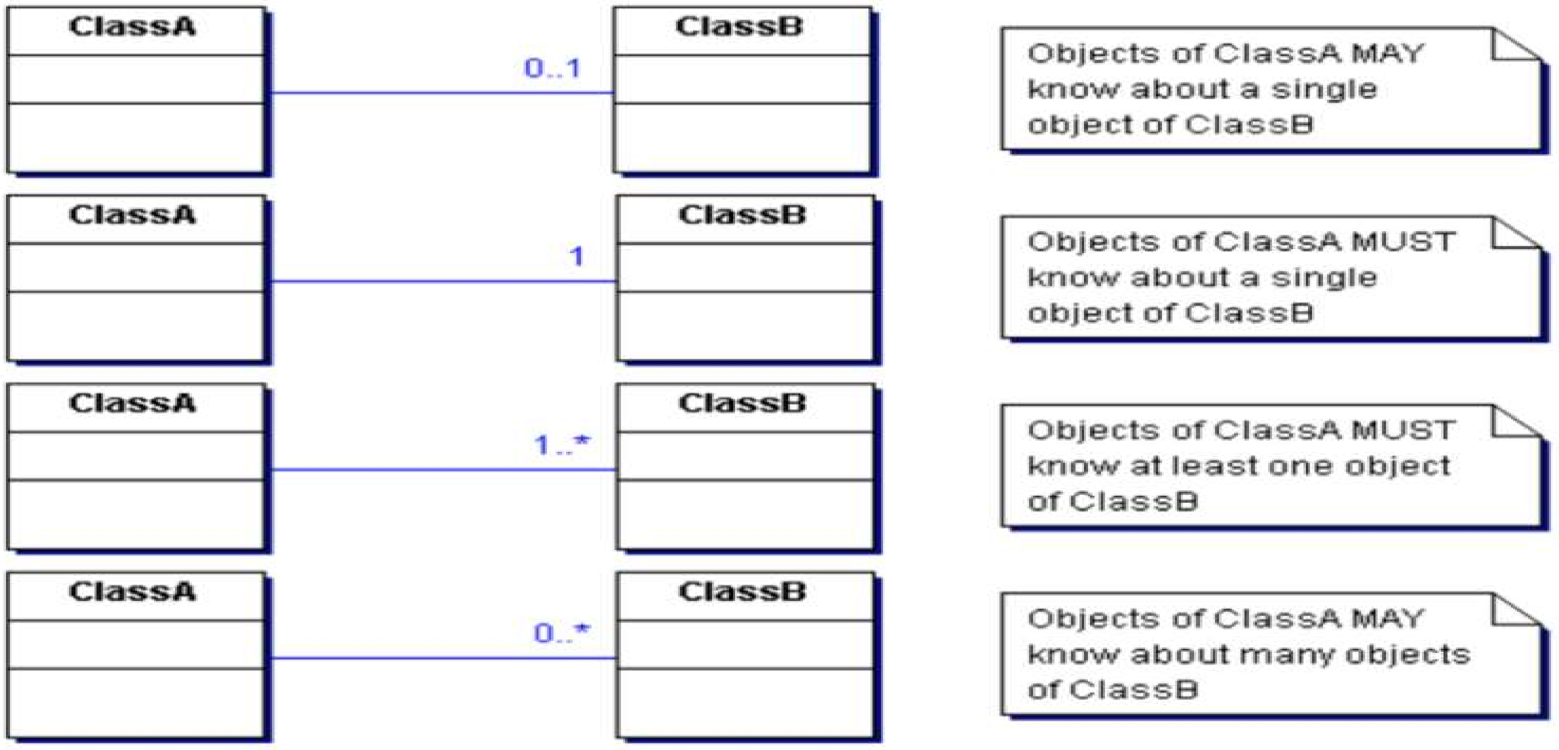
- one-to-one
  - each student must carry exactly one ID card



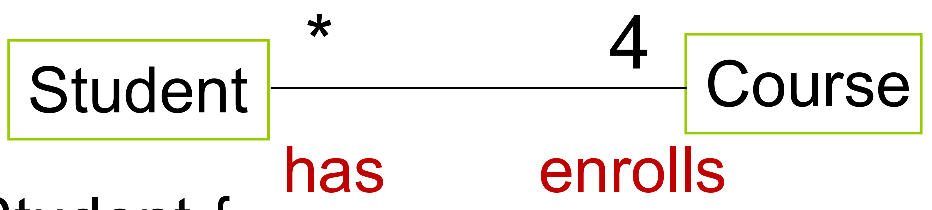
- one-to-many
  - one rectangle list can contain many rectangles


Rectangle
- x: int
- y: int
+ Rectangle(x: int, y: int, width: int, height: int)
+ contains(p: Point): boolean
+ distance(r: Rectangle): double
*
contains
1
RectangleList
- list: ArrayList
+ add(r: Rectangle)
+ clear()

Multiplicity


ClassA
ClassB
0..1
Objects of ClassA MAY know about a single object of ClassB
ClassA
ClassB
1
Objects of ClassA MUST know about a single object of ClassB
ClassA
ClassB
1..*
Objects of ClassA MUST know at least one object of ClassB
ClassA
ClassB
0..*
Objects of ClassA MAY know about many objects of ClassB

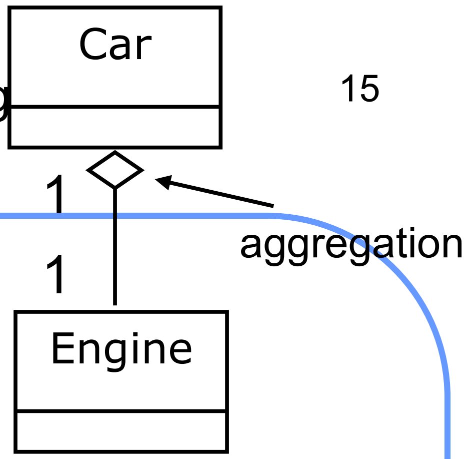
Association: Model to Implementation

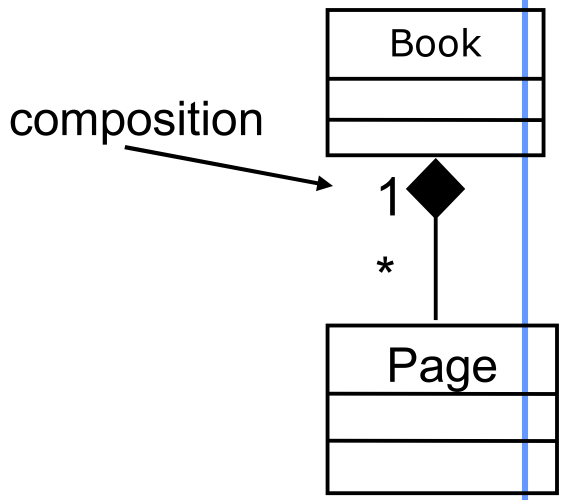


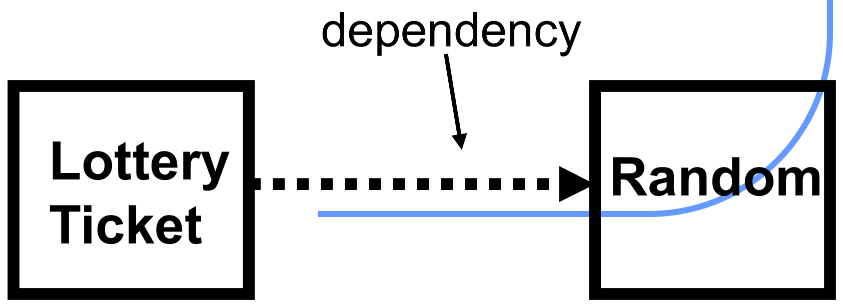
```
Class Student {
    Course enrolls[4];
}

Class Course {
    Student have[];
}
```

Association types

- **aggregation**: “has-a"
  - symbolized by a clear white diamond
- **composition**:”part-off” / "is entirely made of"
  - stronger version of aggregation
  - the parts live and die with the whole
  - symbolized by a black diamond
- **dependency**: "uses temporarily"
  - symbolized by dotted line
  - often is an implementation detail, not an intrinsic part of that object's state

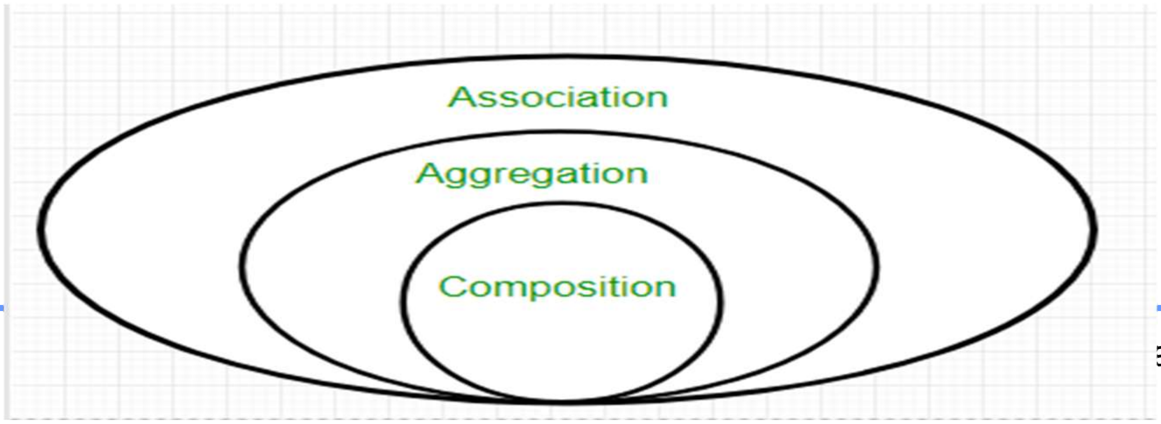




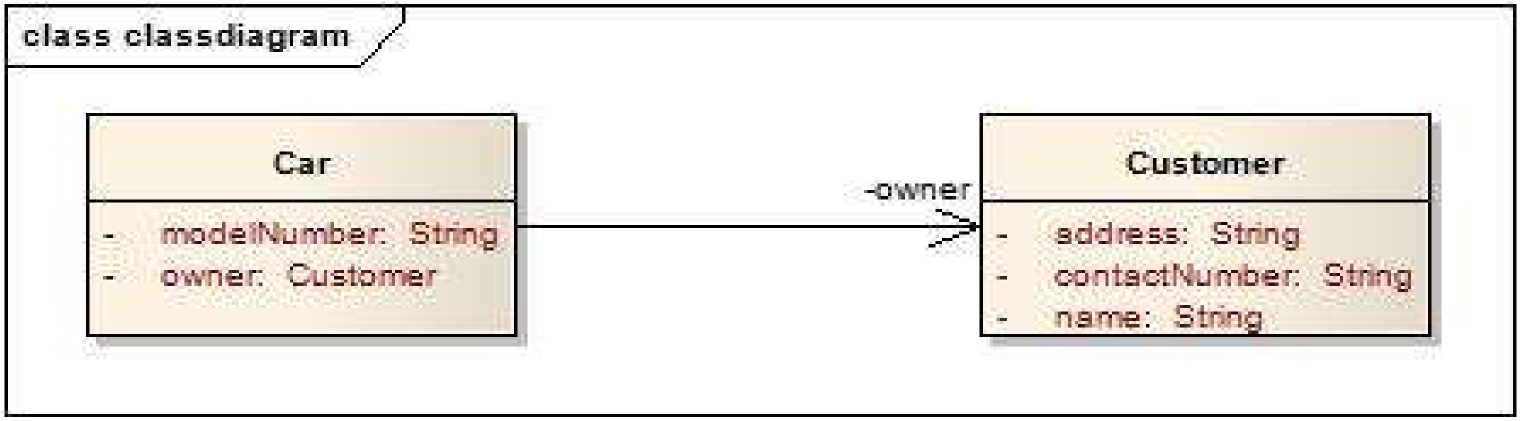


Association Types

- Aggregation implies a relationship where the child **can exist independently** of the parent.
- Composition implies a relationship where the child **cannot exist independent** of the parent.
- Composition is a **strong** Association whereas Aggregation is a **weak** Association

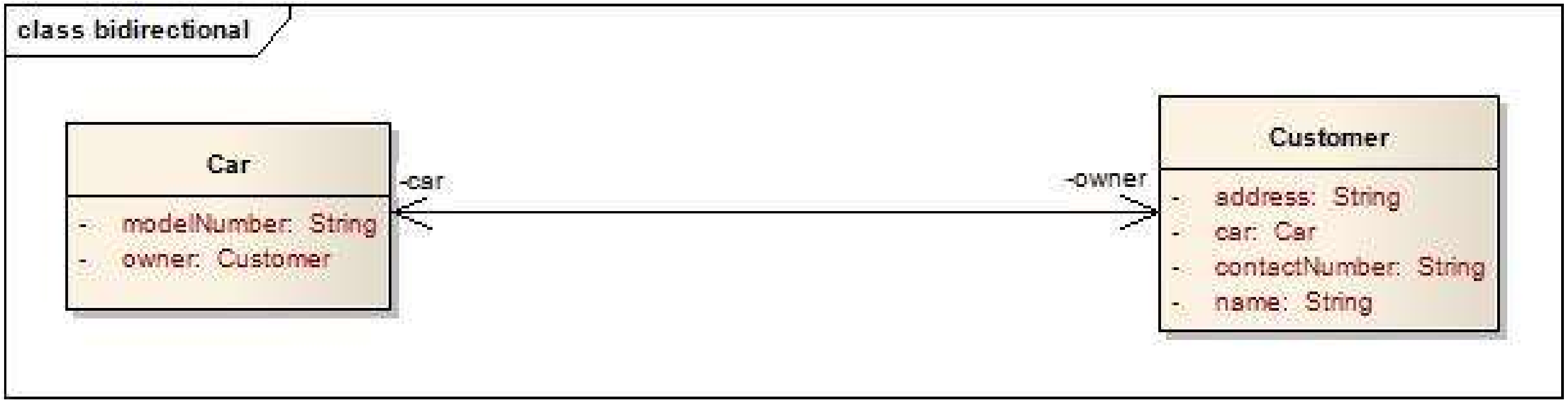


Association



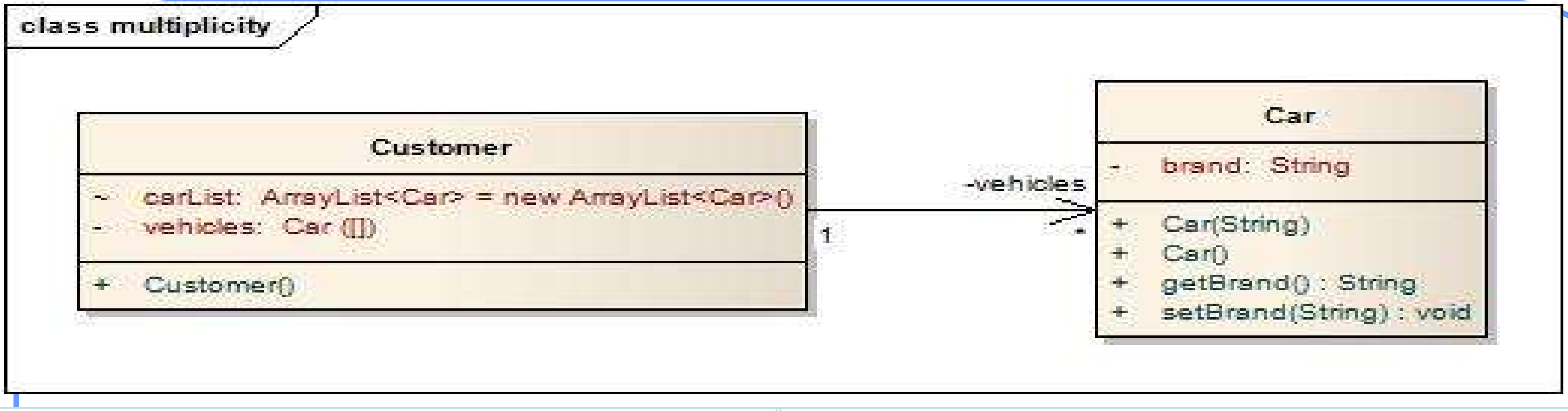
```
public class Customer {
        private String name;
        private String address;
        private String contactNumber;
}
public class Car {
        private String modelNumber;
        private Customer owner;
}
```

Association



```
public class Customer {
        private String name;
        private String address;
        private String contactNumber;
        private Car car;
}
public class Car {
        private String modelNumber;
        private Customer owner;
}
```

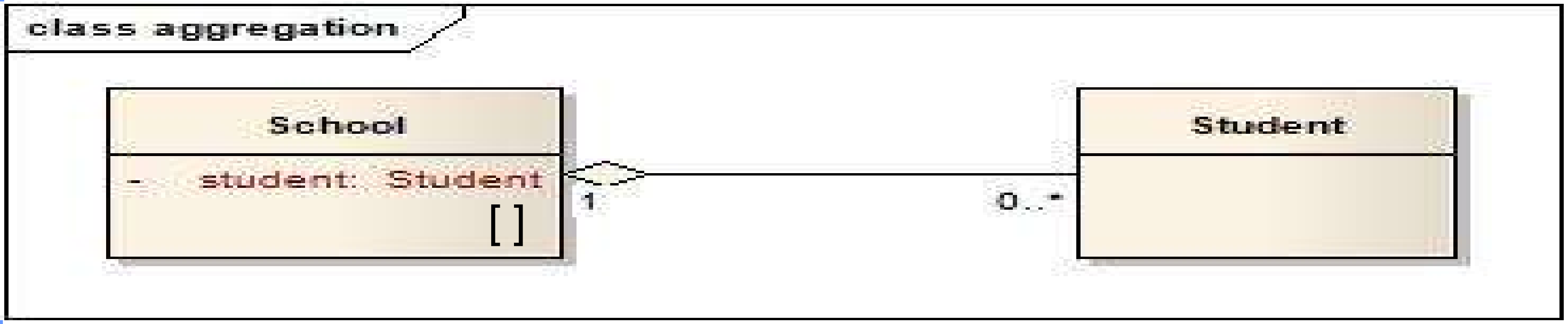
Allows bidirectional association



```
public class Car {
        private String brand;
        public Car(String brands){
            this.brand = brands;  }

        public Car() {   }

        public String getBrand() { return
            brand;}

        public void setBrand(String brand) {
            this.brand = brand;}
}
```

```
public class Customer {
        private Car[] vehicles;
        ArrayList<Car> carList = new
                                ArrayList<Car>();
public Customer(){
vehicles = new Car[2];
vehicles[0] = new Car("Audi");
vehicles[1] = new Car("Mercedes");
carList.add(new Car("BMW"));
carList.add(new Car("Chevy"));
}

}
```

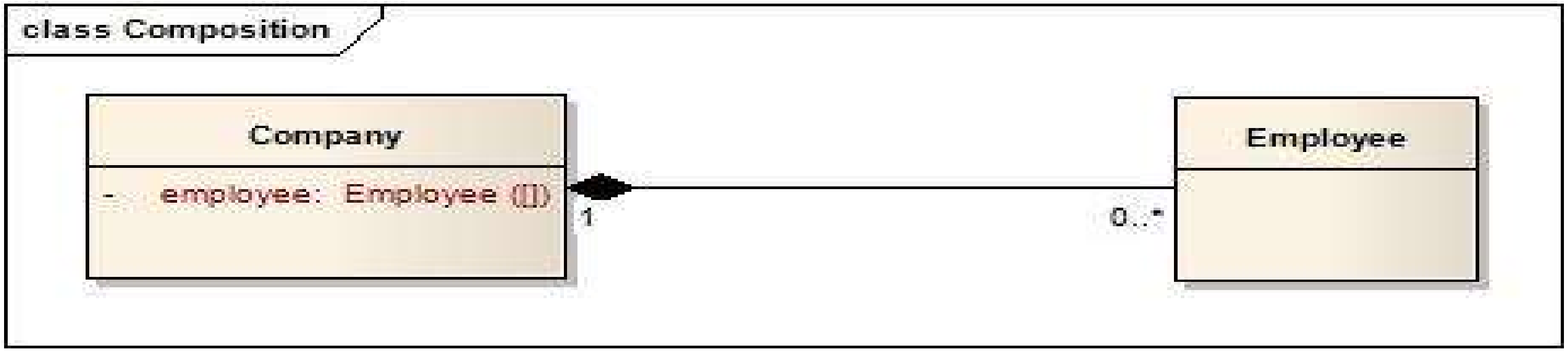
Aggregation



```
public class Student {
}

public class School {
    private Student[] student;
}
```

Does not allow bidirectional association

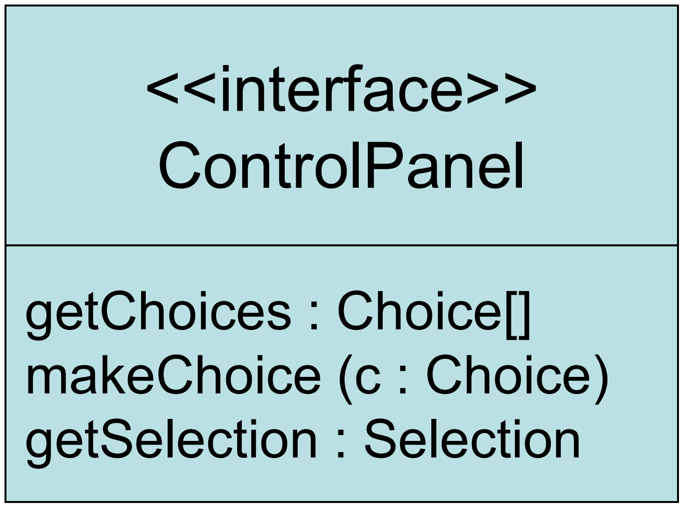
Composition



```
public class Employee {
}

public class Company {
      private Employee[] employee;
}
```

Interface Services

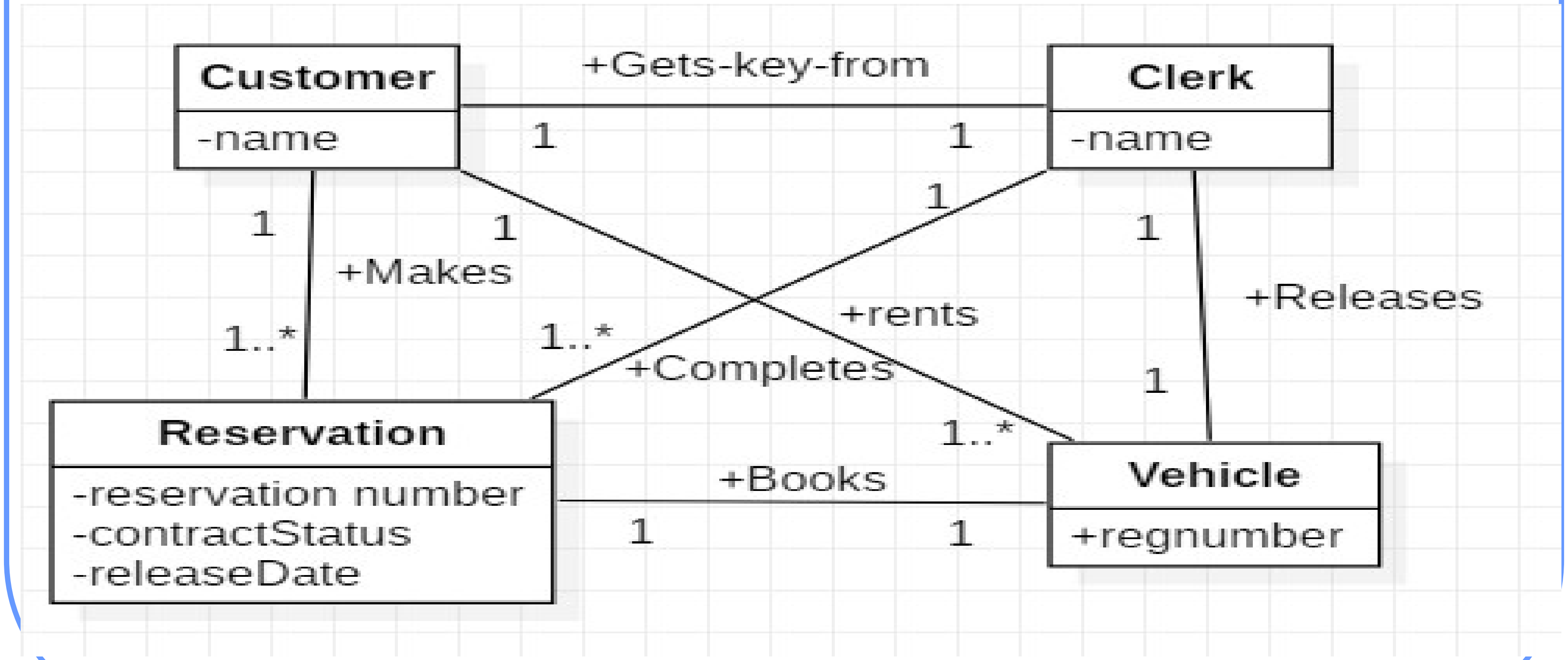
- Interfaces do not get instantiated. They have no attributes or state. Rather, they specify the services offered by a related class



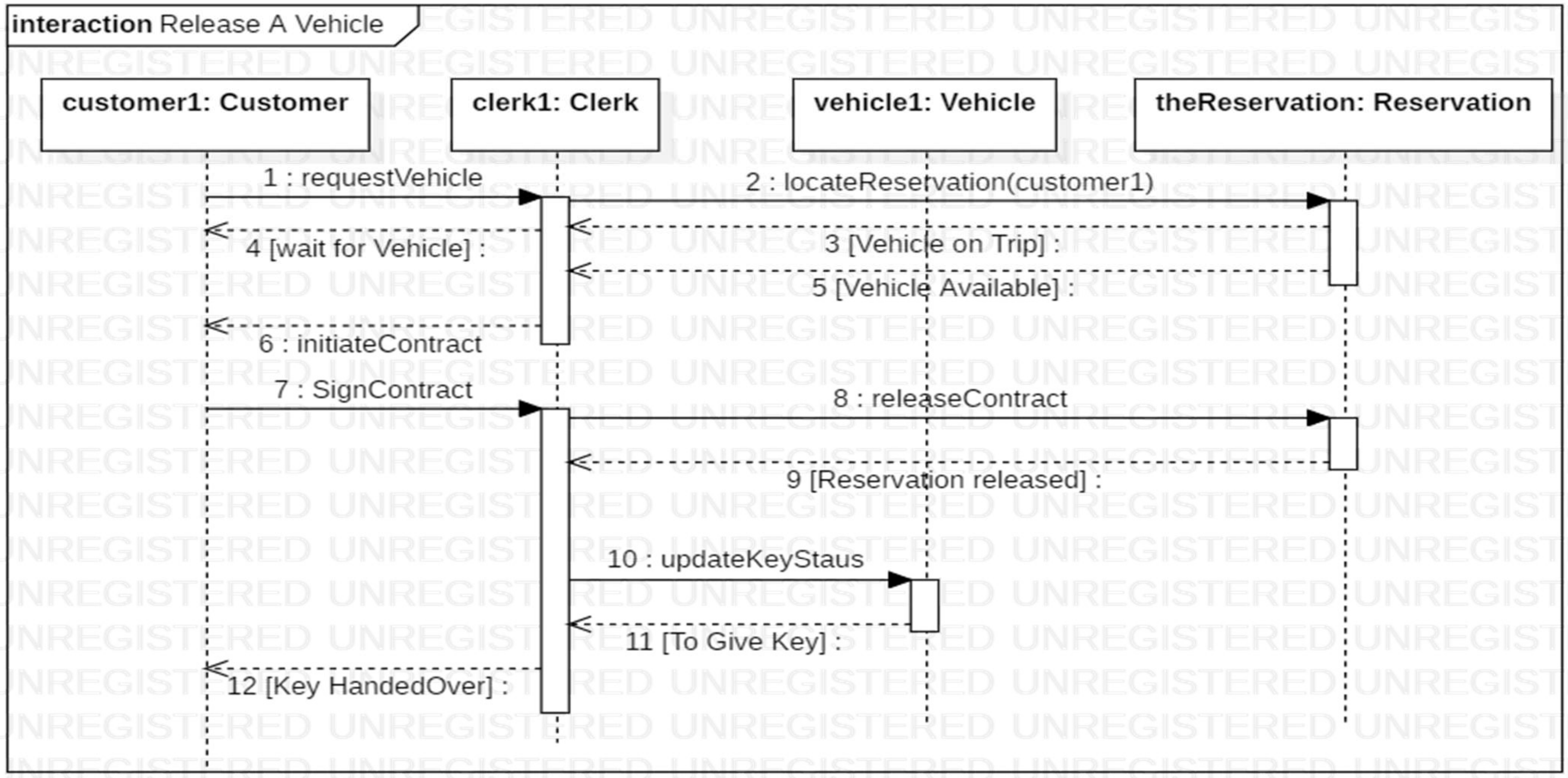
Types of Classes

- Ones found during analysis (From Object Model):
  - people, places, events, and things about which the system will capture information
  - ones found in application domain
- Ones found during design
  - specific objects like windows and forms that are used to build the system

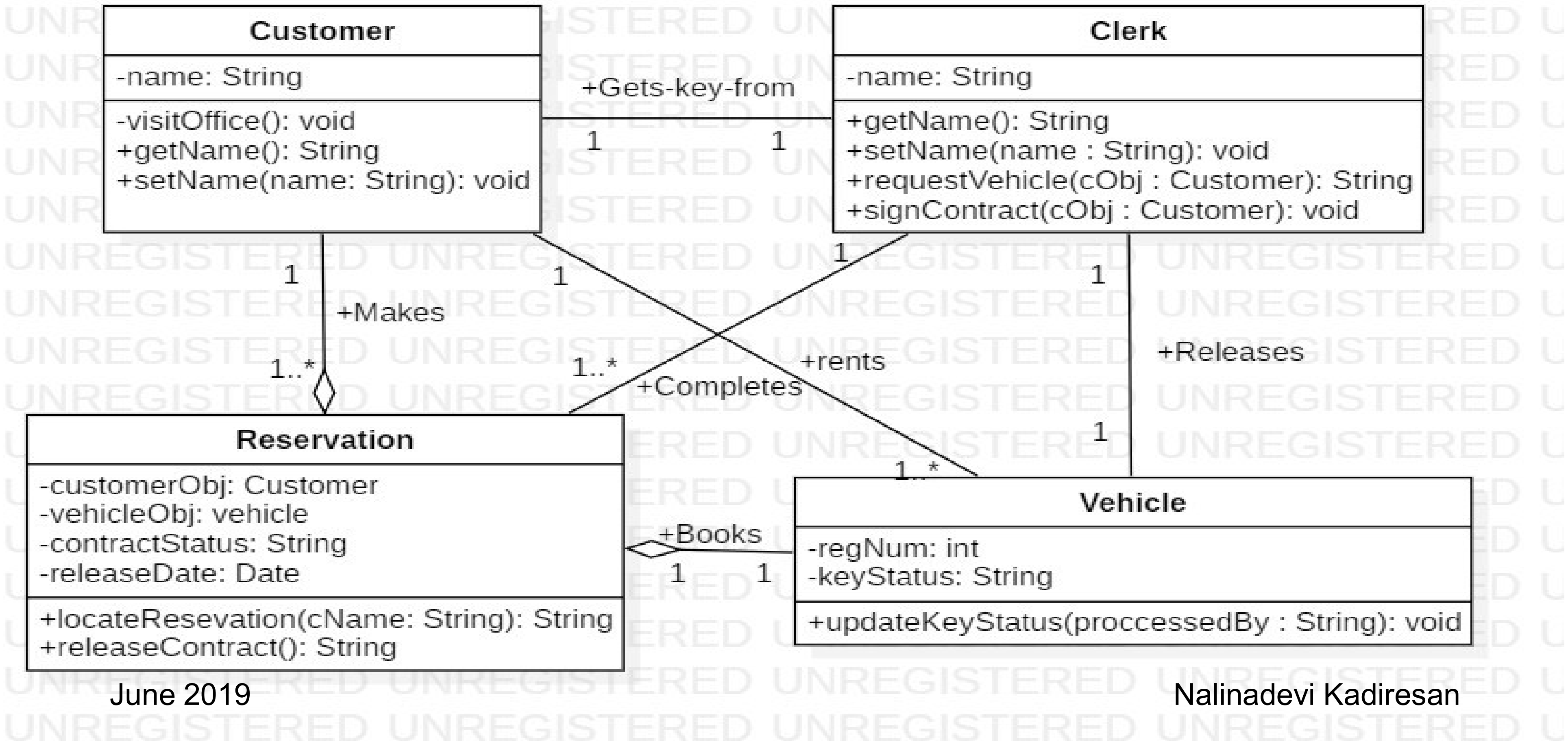
Example-1 Car Rental Application Object Model

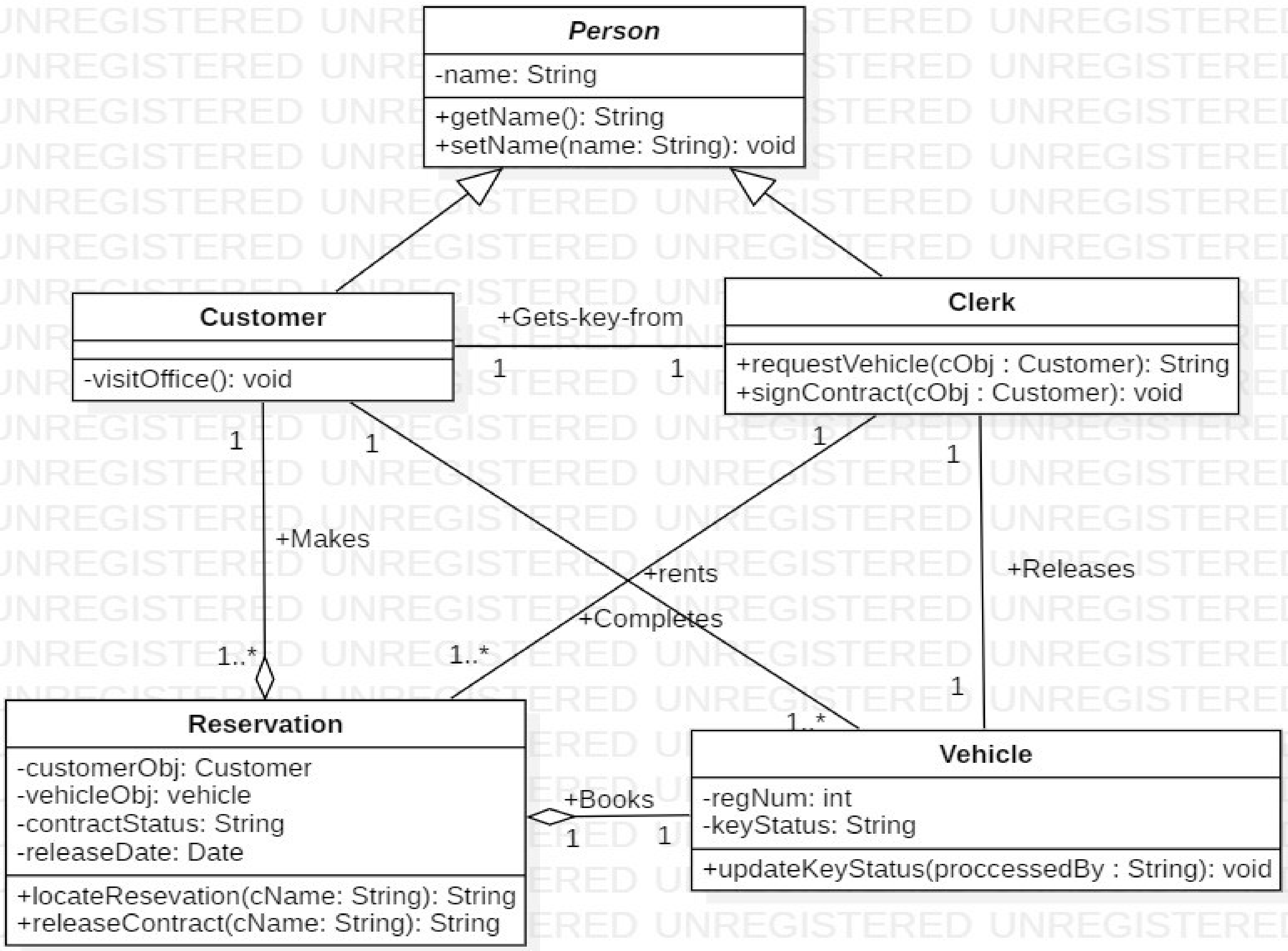


Example-1 Car Rental Application Sequence Diagram

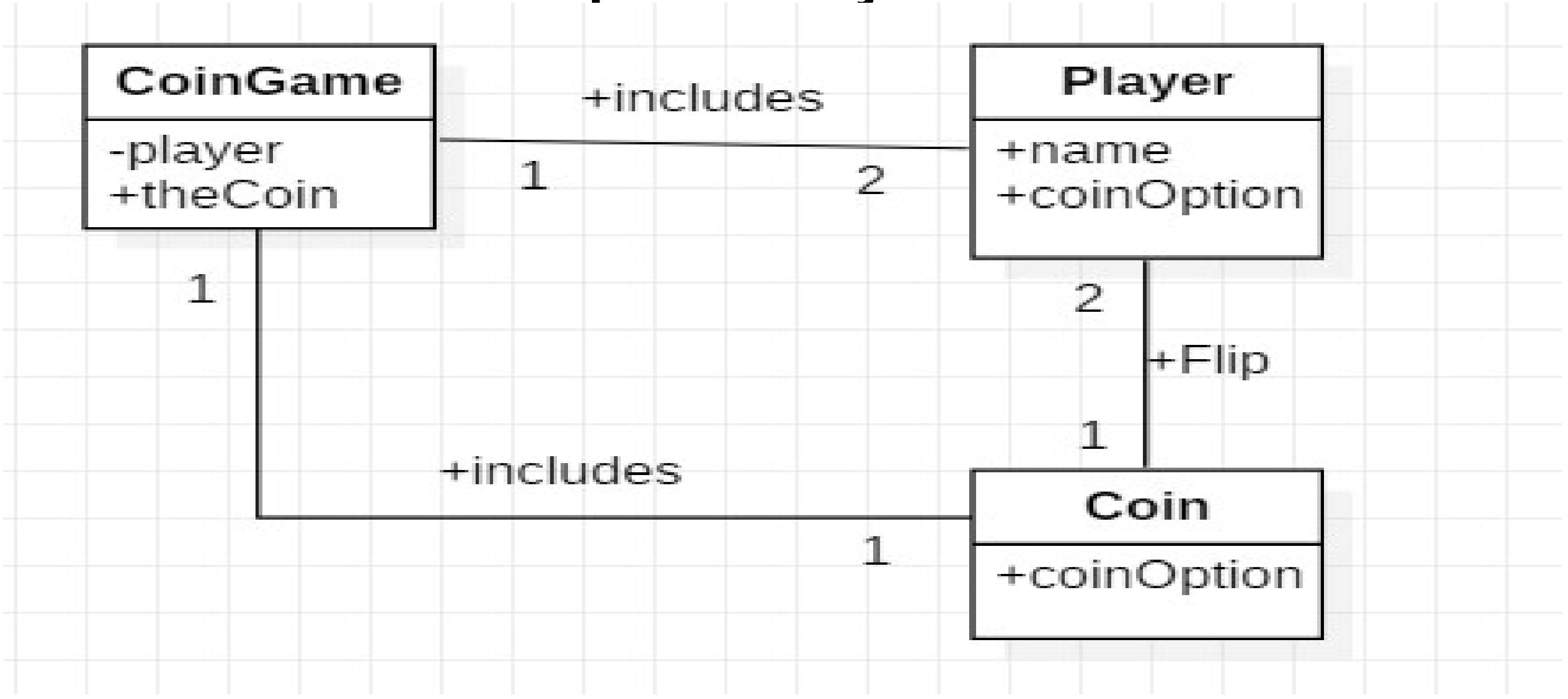


Example-1 Car Rental Application Class Diagram



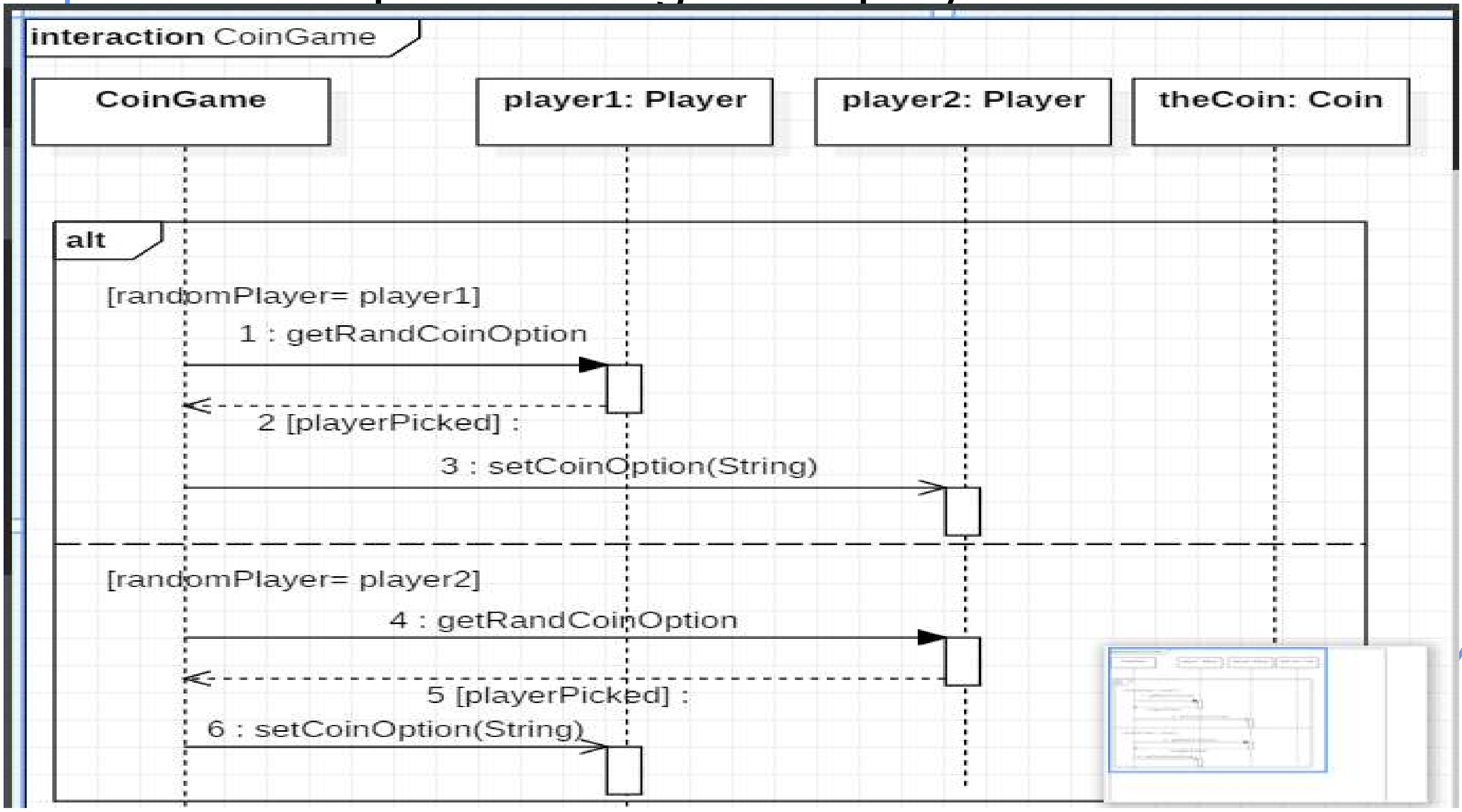

Person
-name: String
+getName(): String
+setName(name: String): void
Customer
-visitOffice(): void
+Gets-key-from
1
1
Clerk
+requestVehicle(cObj : Customer): String
+signContract(cObj : Customer): void
1
1
1
1
+Makes
+rents
+Releases
+Completes
1..*
1..*
1
1..*
Reservation
-customerObj: Customer
-vehicleObj: vehicle
-contractStatus: String
-releaseDate: Date
+locateResevation(cName: String): String
+releaseContract(cName: String): String
+Books
1
1
Vehicle
-regNum: int
-keyStatus: String
+updateKeyStatus(proccessedBy : String): void

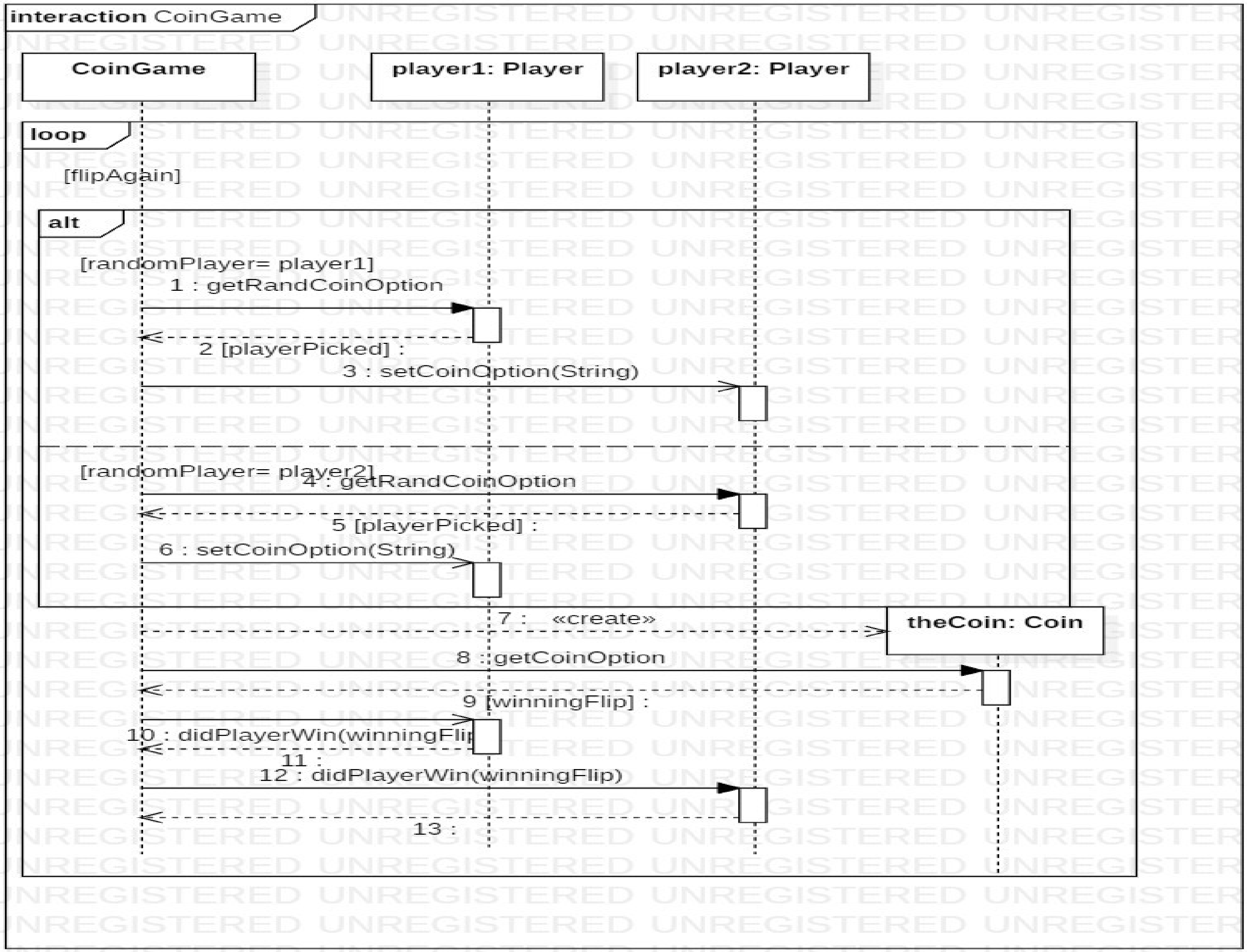
Example-2
Coin Flip -- Object Model



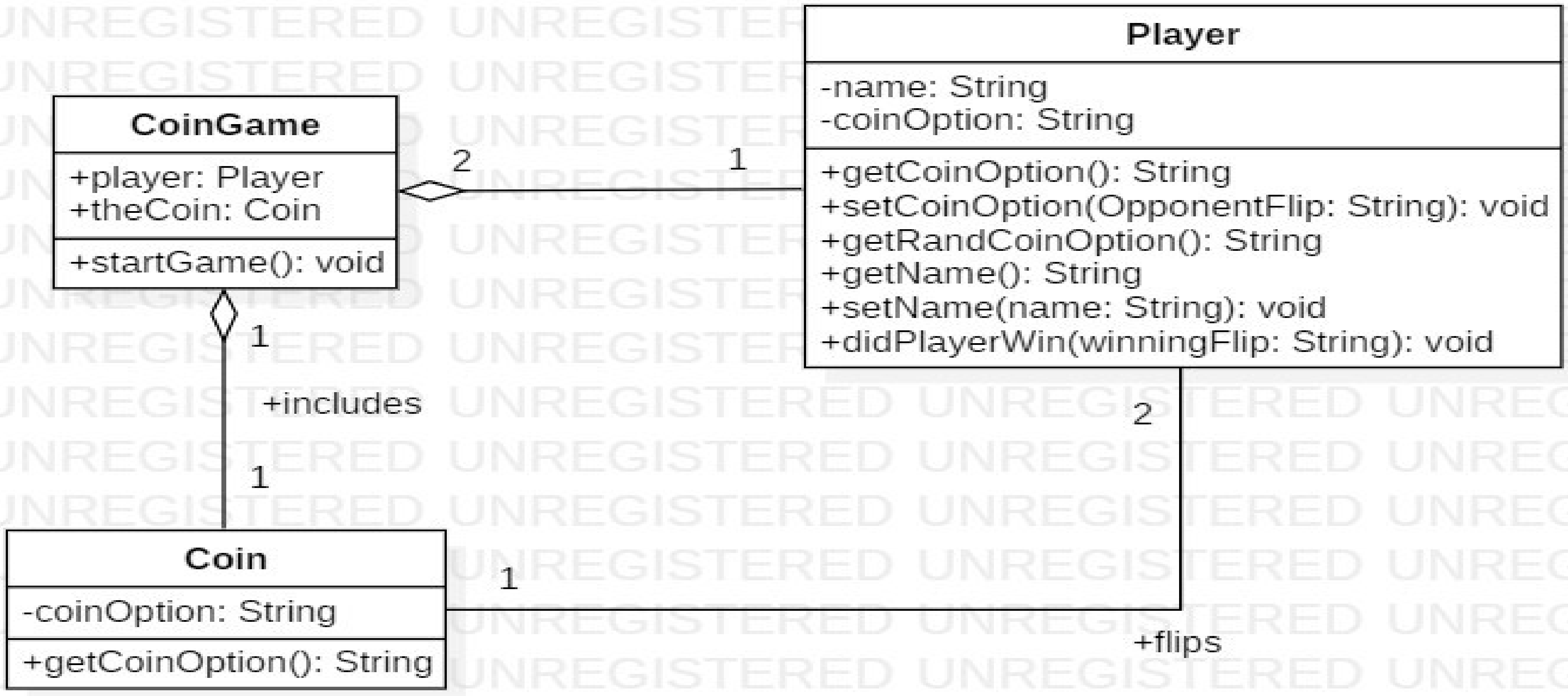
Example-2 Coin Flip

Sequence Diagram – player selection




interaction CoinGame
CoinGame
player1: Player
player2: Player
loop
[flipAgain]
alt
[randomPlayer= player1]
1 : getRandCoinOption
2 [playerPicked] :
3 : setCoinOption(String)
[randomPlayer= player2]
4 : getRandCoinOption
5 [playerPicked] :
6 : setCoinOption(String)
7 : «create»
theCoin: Coin
8 : getCoinOption
9 [winningFlip] :
10 : didPlayerWin(winningFlip
11 :
12 : didPlayerWin(winningFlip)
13 :

Example-2 Coin Flip Class Diagram



Class diagram exercise 1

- Let's do a class diagram for the following casual use case, *Start New Poker Round* :

  The scenario begins when the player chooses to start a new round in the UI. The UI asks whether any new players want to join the round; if so, the new players are added using the UI.

  All players' hands are emptied into the deck, which is then shuffled. The player left of the dealer supplies an ante bet of the proper amount. Next each player is dealt a hand of two cards from the deck in a round-robin fashion; one card to each player, then the second card.

  If the player left of the dealer doesn't have enough money to ante, he/she is removed from the game, and the next player supplies the ante. If that player also cannot afford the ante, this cycle continues until such a player is found or all players are removed.

Class diagram exercise 2

- Let's do a class diagram for the following casual use case, *Add Calendar Appointment* :

  The scenario begins when the user chooses to add a new appointment in the UI. The UI notices which part of the calendar is active and pops up an Add Appointment window for that date and time.

  The user enters the necessary information about the appointment's name, location, start and end times. The UI will prevent the user from entering an appointment that has invalid information, such as an empty name or negative duration. The calendar records the new appointment in the user's list of appointments. Any reminder selected by the user is added to the list of reminders.

  If the user already has an appointment at that time, the user is shown a warning message and asked to choose an available time or replace the previous appointment. If the user enters an appointment with the same name and duration as an existing group meeting, the calendar asks the user whether he/she intended to join that group meeting instead. If so, the user is added to that group meeting's list of participants.

Next Session will be

Object Oriented Analysis with Use Case Diagram