SOFTWARE ENGINEERING
CO3001

CHAPTER 5 — INTRODUCTION TO OBJECT ORIENTED PROGRAMMING

Anh Nguyen-Duc

Tho Quan Thanh

Agenda

- History
- Key OOP Concepts
  - Object, Class
  - Instantiation, Constructors
  - Encapsulation
  - Inheritance and Subclasses
  - Abstraction
  - Reuse
  - Polymorphism, Dynamic Binding
- Object-Oriented Design and Modeling

Agenda

- There are different approaches to writing computer programs.
  - Procedural programming
  - Object oriented programming
- They all involve decomposing your programs into parts.

"And so, from Europe, we get things such ... object-oriented analysis and design (a clever way of breaking up software programming instructions and data into small, reusable objects, based on certain abstraction principles and design hierarchies.)"

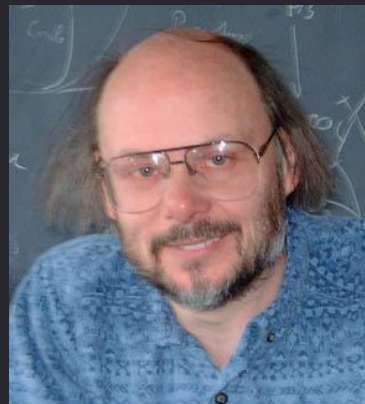-Michael A. Cusumano, The Business Of Software

OOP … since 1962

- Simula 1 (1962 - 1965) and Simula 67 (1967) Norwegian Computing Center, Oslo, Norway by Ole-Johan Dahl and Kristen Nygaard.

Dahl and Nygaard at the time of Simula's development
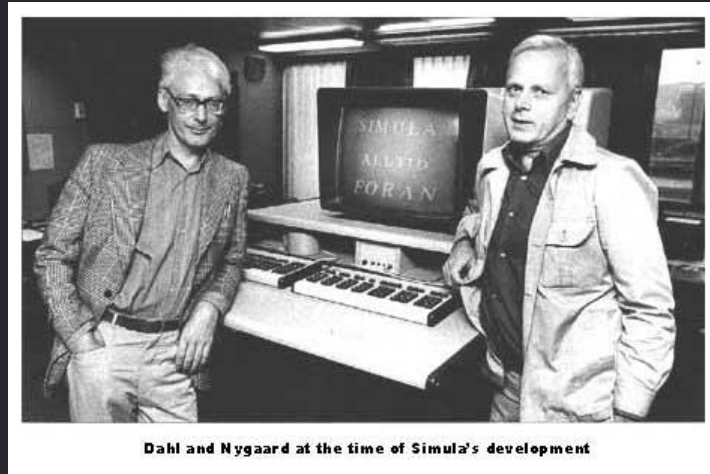
Turing Award Winners - 2001

OOP ... since 1962

- Smalltalk (1970s), Alan Kay's group at Xerox PARC

- C++ (early 1980s), Bjarne Stroustrup, Bell Labs

OOP Languages

- Modula – 3, Oberon, Eiffel, Java, C#, Python
  - many languages have some Object Oriented version or capability
- One of the dominant styles for implementing complex programs with large numbers of interacting components
  - … but not the only programming paradigm and there are variations on object oriented programming

Definition – OOP, Class

- Object-oriented programming is a method of programming based on a hierarchy of classes, and well-defined and cooperating objects
- A class is a structure that defines the data and the methods to work on that data. When you write programs in the Java language, all program data is wrapped in a class, whether it is a class you write or a class you use from the Java platform API libraries

- Class: a collection of data (fields/ variables) and methods that operate on that data
  - define the contents/capabilities of the instances (objects) of the class
  - a class can be viewed as a factory for objects
  - a class defines a recipe for its objects
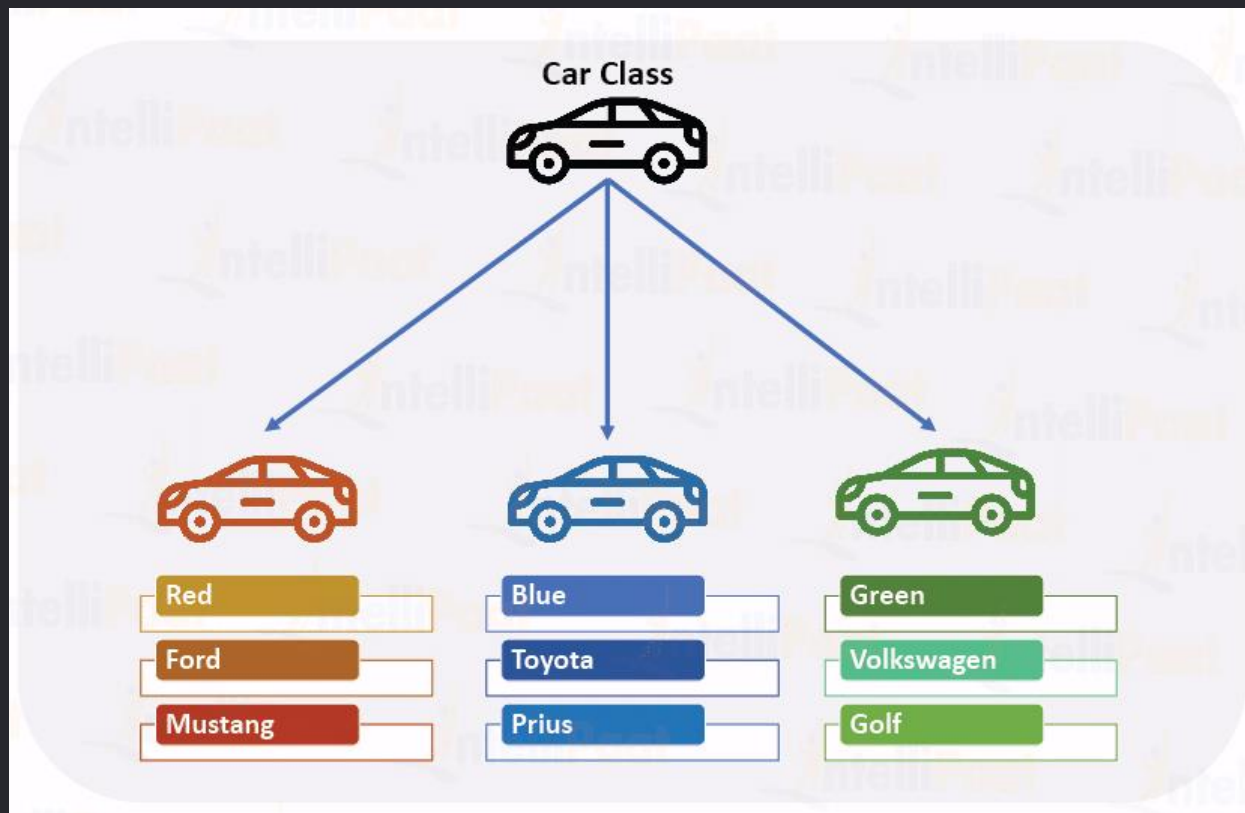
Example of a class (Java)

```java
class Customer {
    // Fields/ variables/ Data
    private String name;    //Can get but not change
    private double salary;  // Cannot get or set
    // Constructor
    Customer(String n, double s) {
        name = n; order = s;
    }
    // Methods
    void pay () {
        System.out.println("Pay to the order of " +
                           name + " $" + order);
    }
    public String getName() { return name; } // getter
}
```

```java
Customer a = new Customer(«Anh», 500);
a.pay();
Customer b = new Customer(«Tho», 600);
String anh_name = a.getName();
b.pay();
```
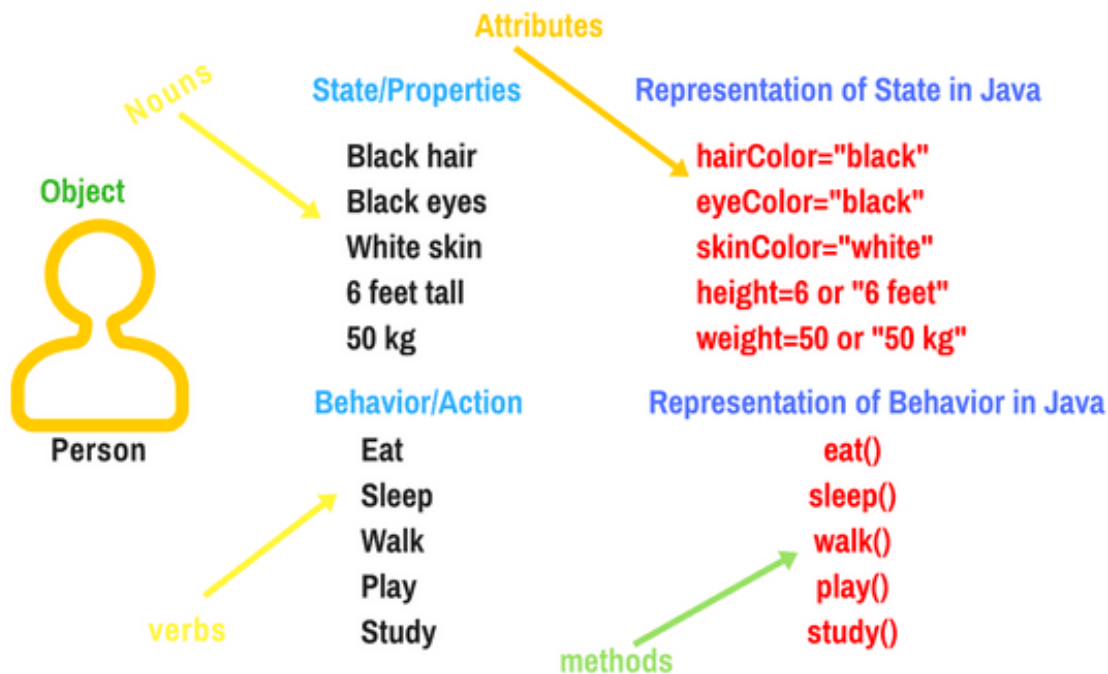
Definition – Class, Object

- Object creation: memory is allocated for the object's fields as defined in the class
- Initialization is specified through a <u>constructor</u>
- A special method invoked when objects are created

- Different objects have the same attributes but the values of those attributes can vary
  - Reminder: The class definition specifies the attributes and methods for *all objects*

- The current value of an object's attribute's determines it's state.

Concept: Classes describe objects

Concept: Classes describe objects

Attributes

State/Properties — **Representation of State in Java**

Black hair — hairColor="black"
Black eyes — eyeColor="black"
White skin — skinColor="white"
6 feet tall — height=6 or "6 feet"
50 kg — weight=50 or "50 kg"

Object — Person

Nouns

Behavior/Action — **Representation of Behavior in Java**

Eat — eat()
Sleep — sleep()
Walk — walk()
Play — play()
Study — study()

verbs

methods

www.scientecheasy.com

Class Person {
 private String hairColor;
….
}

Notation: How to declare and create objects

```
Employee secretary;  // declares secretary
secretary = new Employee (); // allocates space
Employee secretary = new Employee(); // does both
```
But the secretary is still "blank" (**null**)
```
secretary.name = "Adele";  // dot notation
secretary.birthday ();  // sends a message
```

Notation: How to reference a field or method

Inside a class, no dots are necessary
　　class Person { ... age = age + 1; ...}

Outside a class, you need to say which object you are talking to
　　if (john.age < 75) john.birthday ();

If you don't have an object, you cannot use its fields or methods!

Inheritance

- Inheritance:
  - programming language feature that allows for the implicit definition of variables/methods for a class through an existing class
- An object *also* inherits:
  - the fields described in the class's superclasses
  - the methods described in the class's superclasses
- A class is *not* a complete description of its objects!

Concept: Classes form a hierarchy

- Classes are arranged in a treelike structure called a hierarchy

- The class at the root is named Object

- Every class, except Object, has a superclass

- When you define a class, you specify its superclass
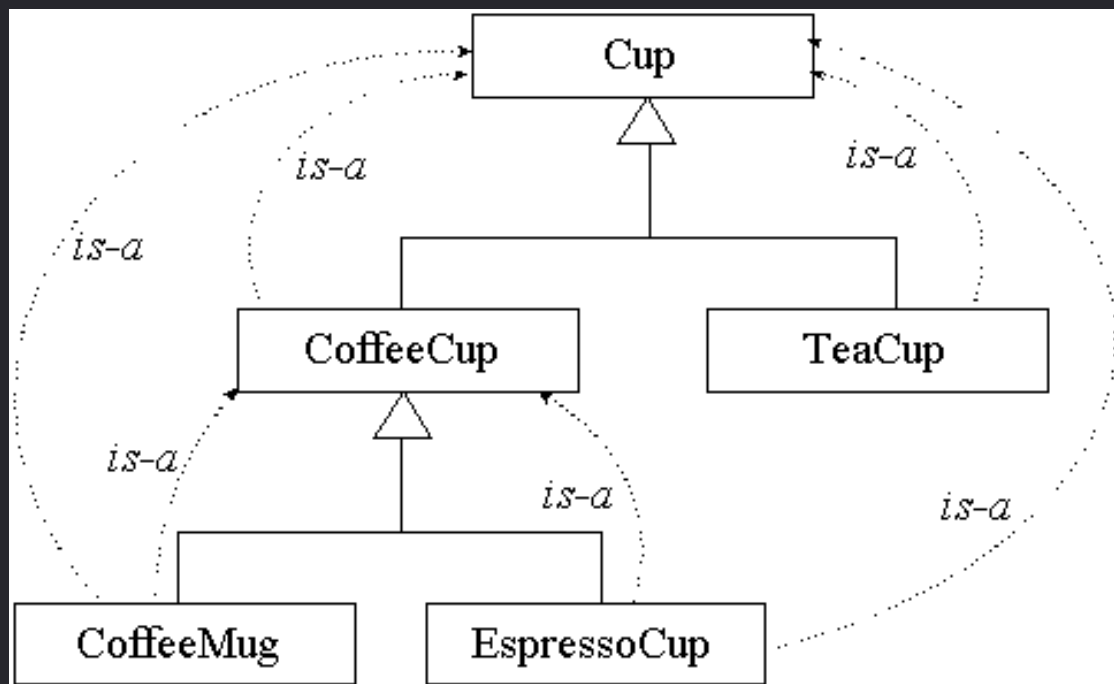 - If you don't specify a superclass, Object is assumed
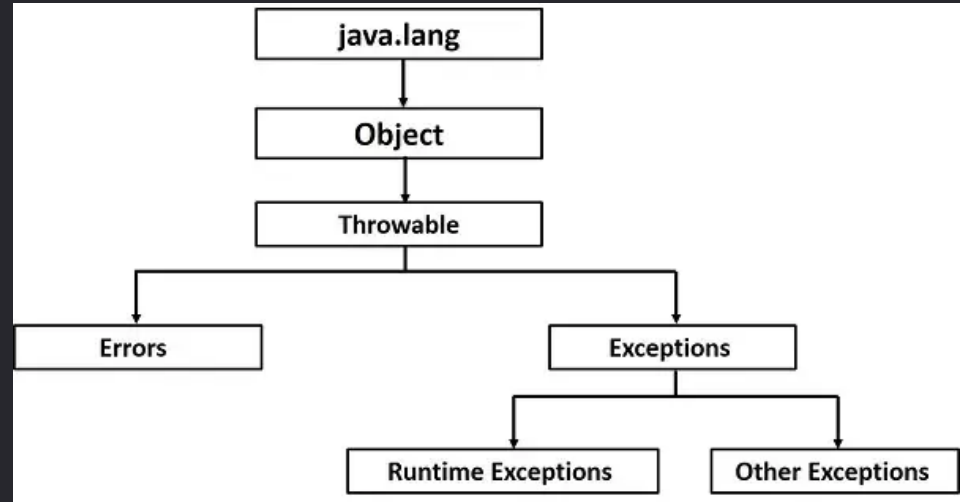
Concept: Classes form a hierarchy

- Subclass relationship
  - B is a subclass of A
  - B inherits all definitions (variables/methods) in A
- A class may have several ancestors, up to **Object**
- Every class may have one or more subclasses

1) Single

2) Multilevel

3) Hierarchical
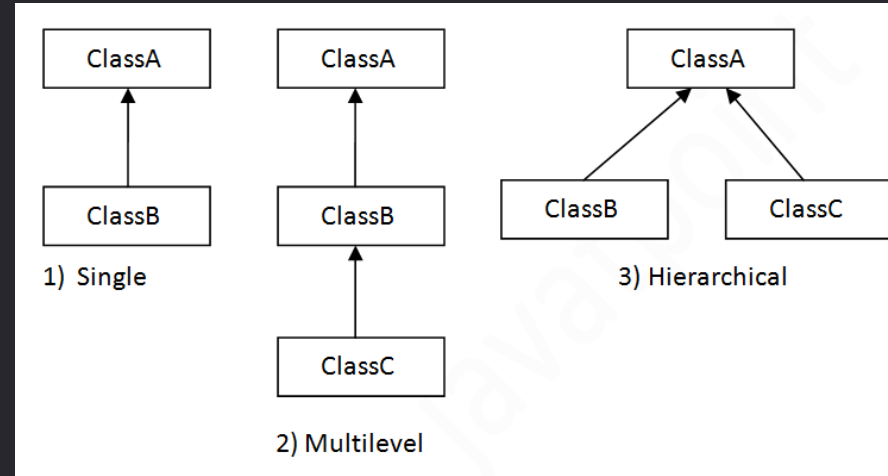
Example of (part of) a hierarchy

Example of inheritance

```
class Person {
    private String name;
    private int age;
    public void birthday () {
        age = age + 1;
    }
}
```

```
class Employee
        extends Person {
    private double salary;
    public void pay () { …}
}
…
Employee ee = new Employee();
ee.birthday()
```

Every Employee has name and age fields and birthday method *as well as* a salary field and a pay method.

Example: Assignment of subclasses

```
class Dog { … }
class Poodle extends Dog { … }
Dog myDog;
Dog rover = new Dog ();
Poodle yourPoodle;
Poodle fifi = new Poodle ();

myDog = rover;                    // ok
yourPoodle = fifi;               // ok
myDog = fifi;                    //ok
yourPoodle = rover;              // illegal
yourPoodle = (Poodle) rover;     //runtime check
```

Encapsulation

- Also know as separation of concerns and information hiding

- When creating new data types (classes) the details of the actual data and the way operations work is hidden from the other programmers who will use those new data types
    - So they don't have to worry about them
    - So they can be changed without any ill effects (loose coupling)

- Encapsulation makes it easier to be able to use something
    - microwave, radio, ipod, the Java String class

Encapsulation (A capsule)

Fig: Encapsulation
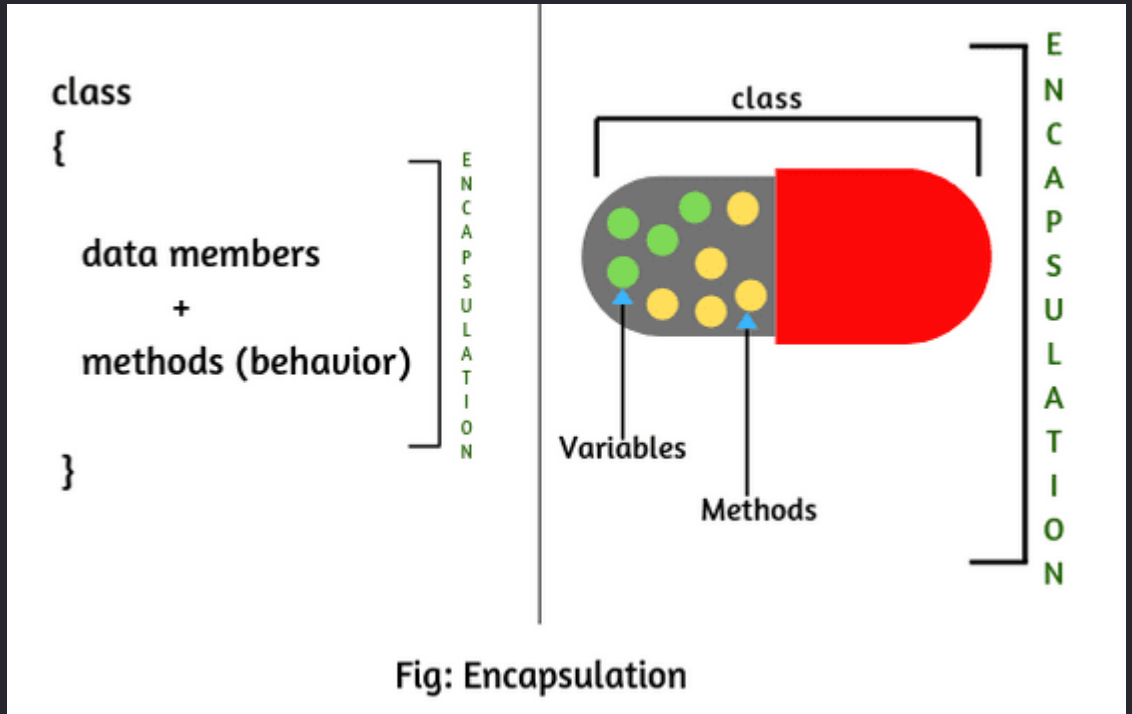
Kinds of access in Java

- Java provides four levels of access:
    - ▫ **public**: available everywhere
    - ▫ **protected**: available within the package (in the same subdirectory) and to all subclasses
    - ▫ [default]: available within the package
    - ▫ **private**: only available within the class itself
- The default is called package visibility
- In small programs this isn't important…right?

Encapsulation

```java
public class Coat {
    private double price;
    private String customer;

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public String getCustomer() {
        return customer;
    }

    public void setCustomer(String customer) {
        this.customer = customer;
    }
}
```

Abstraction

- OOP is about *abstraction*

- Abstraction is a method of hiding the implementation detail and only show the functionalities

- Encapsulation and Inheritance are examples of abstraction

Polymorphism

- Polymorphism means many (poly) shapes (morph)

- In Java, polymorphism refers to the fact that you can have multiple methods with the same name in the same class

- There are two kinds of polymorphism:
  - Overloading
    - Two or more methods with different signatures
  - Overriding
    - Replacing an inherited method with another having the same signature

Polymorphism

- two methods have to differ in their *names* or in the *number* or *types* of their parameters
  - foo(int i) and foo(int i, int j) are different
  - foo(int i) and foo(int k) are the same
  - foo(int i, double d) and foo(double d, int i) are different

Overloading

```java
class Test {
    public static void main(String args[]) {
        myPrint(5);
        myPrint(5.0);
    }

    static void myPrint(int i) {
        System.out.println("int i = " + i);
    }

    static void myPrint(double d) { // same name, different parameters
        System.out.println("double d = " + d);
    }
}

    int i = 5
    double d = 5.0
```

Overriding

```
class Animal {
    public static void main(String args[]) {
        Animal animal = new Animal();
        Dog dog = new Dog();
        animal.print();
        dog.print();
    }
    void print() {
        System.out.println("Superclass Animal");
    }
}

public class Dog extends Animal {
    void print() {
        System.out.println("Subclass Dog");
    }
}

        Superclass Animal
        Subclass Dog
```

- This is called overriding a method

- Method print in Dog overrides method print in Animal

- A subclass variable can *shadow* a superclass variable, but a subclass method can *override* a superclass method

Another examples

Overriding

```java
class Dog{
    public void bark(){
        System.out.println("woof ");
    }
}
class Hound extends Dog{
    public void sniff(){
        System.out.println("sniff ");
    }

    public void bark(){
        System.out.println("bowl");
    }
}
```

Same Method Name,
Same parameter

Overloading

```java
class Dog{
    public void bark(){
        System.out.println("woof ");
    }

    //overloading method
    public void bark(int num){
        for(int i=0; i<num; i++)
            System.out.println("woof ");
    }
}
```

Same Method Name,
Different Parameter

When to do?

- You should *overload* a method when you want to do essentially the same thing, but with different parameters
- You should *override* an inherited method if you want to do something slightly different than in the superclass
  - It's almost always a good idea to override public void toString() -- it's handy for debugging, and for many other reasons
  - To test your own objects for equality, override public void equals(Object o)
  - There are special methods (in java.util.Arrays) that you can use for testing array equality

Reuse

- Inheritance encourages software reuse

- Existing code need not be rewritten

- Successful reuse occurs only through careful planning and design
  - when defining classes, anticipate future modifications and extensions

Building Complex Systems

- From Software Engineering:
  complex systems are difficult to manage

- Proper use of OOP aids in managing this complexity

- The analysis and design of OO systems require corresponding modeling techniques

Object-Oriented Modeling

- UML: Unified Modeling Language
  - OO Modeling Standard
  - Booch, Jacobson, Rumbaugh
- What is depicted?
  - Class details and static relationships
  - System functionality
  - Object interaction
  - State transition within an object

Some UML Modeling Techniques

- Class Diagrams

- Use Cases/Use Case Diagrams

- Interaction Diagrams

- State Diagrams

Object-Oriented Design Models

- Static Model
  - Class Diagrams
- Dynamic Model
  - Use Cases, Interaction Diagrams, State Diagrams, others