CSE 4513

Lec – 6

Software Design (Basics)

All contents are based on textbooks, research articles, online resources, and personal experience.

If you give me a program that **works** perfectly but is impossible to change, then it won’t work when the requirements change, and I won’t be able to make it work. Therefore the program will become useless.

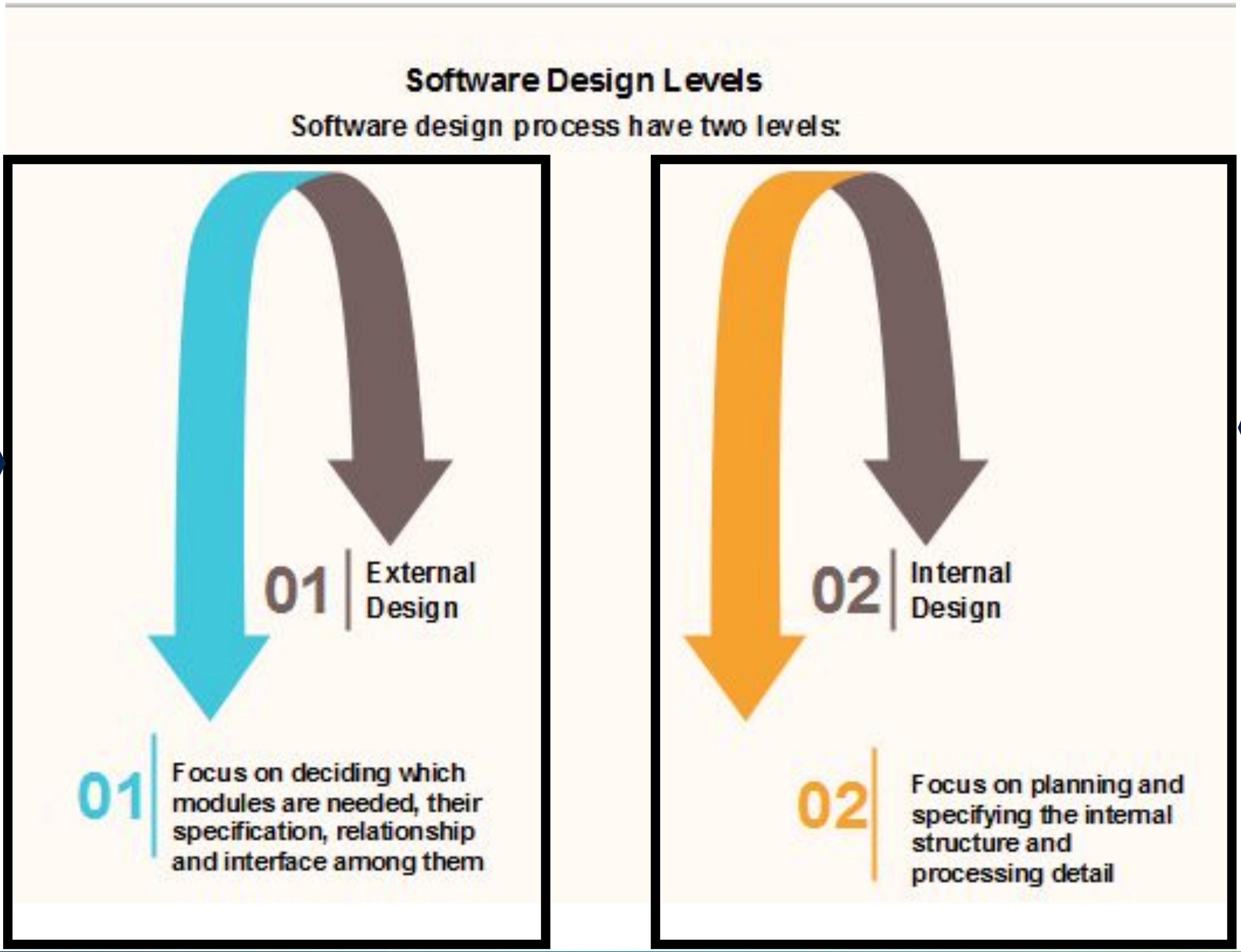
If you give me a program that **does not work** but is easy to change, then I can make it work, and keep it working as requirements change. Therefore the program will remain continually useful.

What is Software Design

is a mechanism to **transform user requirements** into some suitable form, which **helps the programmer in software coding and implementation**.

- ✔ It deals with representing the client's requirement, as described in SRS (Software Requirement Specification) document, into a form, i.e., easily implementable using programming language.
- ✔ moves the concentration from the problem domain to the solution domain.
- ✔ In software design, we consider the system to be a **set of components or modules** with clearly defined **behaviors & boundaries**.

SW Design Level



Software Design Levels

Software design process have two levels:

01 Focus on deciding which modules are needed, their specification, relationship and interface among them

02 Focus on planning and specifying the internal structure and processing detail

High Level Design

Detail Level Design

Objectives of SW Design

01 **Correctness**
should be correct as per requirement

02 **Completeness**
should have all components like data structures, modules, and external interfaces, etc.

03 **Efficiency**
Resources should be used efficiently by the program.

04 **Flexibility**
Able to modify on changing needs**.**

05 **Consistency**
There should not be any inconsistency in the design..

06 **Maintainability**
should be so simple so that it can be easily maintainable by other designers.

Software Design Principles

- ✔ Software design principles are concerned with providing means to handle the complexity of the design process effectively.
- ✔ Effectively managing the complexity will not only reduce the effort needed for design but can also reduce the scope of introducing errors during design.
- ✔ The key software design principles are:

SOLID

- Will be discussed in detail

DRY

- Don't Repeat Yourself
- each small pieces of knowledge (code) may only occur exactly once in the entire system.
- This helps us to write scalable, maintainable and reusable code.

YAGNI

- You aren't gonna need it
- always implement things when you actually need them
- never implements things before you need them.

KISS

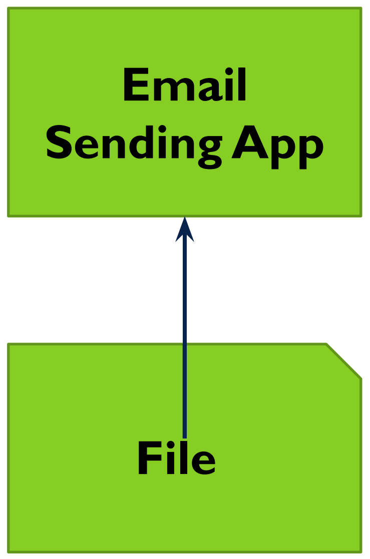
- Keep it simple, Stupid!
- keep each small piece of software simple
- unnecessary complexity should be avoided.

DRY encourages code reusability
KISS emphasizes simplicity in software design
YAGNI advocates for avoiding unnecessary features

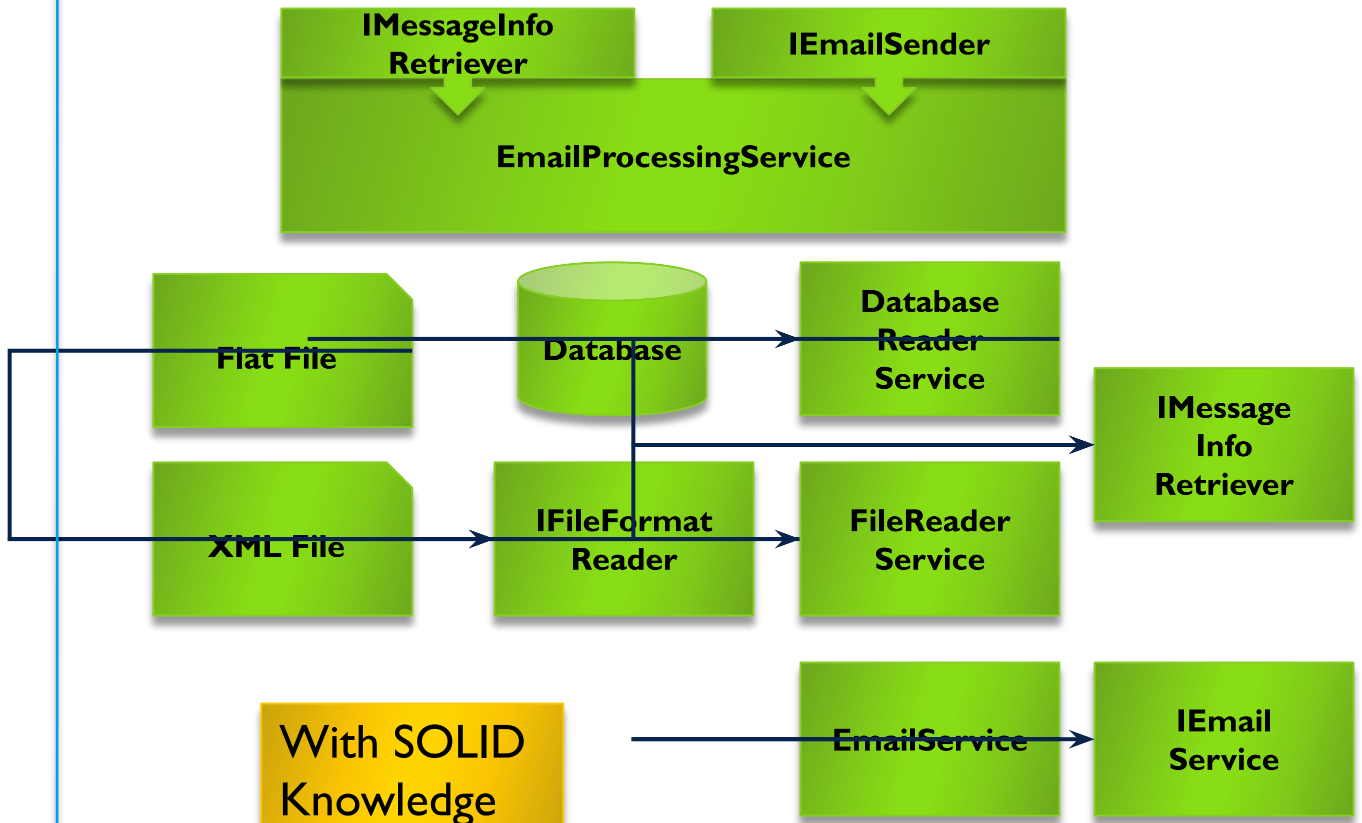
Software Design Principles - SOLID

- ✔ In Object Oriented Programming (OOP), SOLID is an acronym, introduced by Michael Feathers, for five design principles used to make software design more understandable, flexible, and maintainable.
- ✔ There are five SOLID principles:
  - Single Responsibility Principle (SRP)
  - Open Closed Principle (OCP)
  - Liskov Substitution Principle (LSP)
  - Interface Segregation Principle (ISP)
  - Dependency Inversion Principle (DIP)

WHY SOLID



Without SOLID Knowledge



With SOLID Knowledge

Recap OO Concept

Classes and Objects

- ✔ **A Class** is a blueprint or template from which objects are created
- ✔ A **class** defines object properties including a valid range of values, and a default value.
- ✔ A **class** also describes object behavior.
- ✔ An **object** is a member or an "instance" of a class.
- ✔ Objects are generated by the classes and they actually contain values.
- ✔ We design an application at the class level. But the code in OOP is organized around object

Class: Human **Object:** Man, Woman
Class: Fruit **Object:** Apple, Banana, Mango, Guava etc.

Let's take an example

We want to develop device management system, for example Mobile phone.
how you design such software?
Let’s start with few mobiles

Figure out few differences

These also can be listed as their common characteristics

Figure out few common action that can be performed by these devices

Let's take an example

Common Characteristics

- ✓Manufacturer
- ✓Dimension
- ✓No. of Camera
- ✓No. of Sim supported
- ✓Price

Common Action

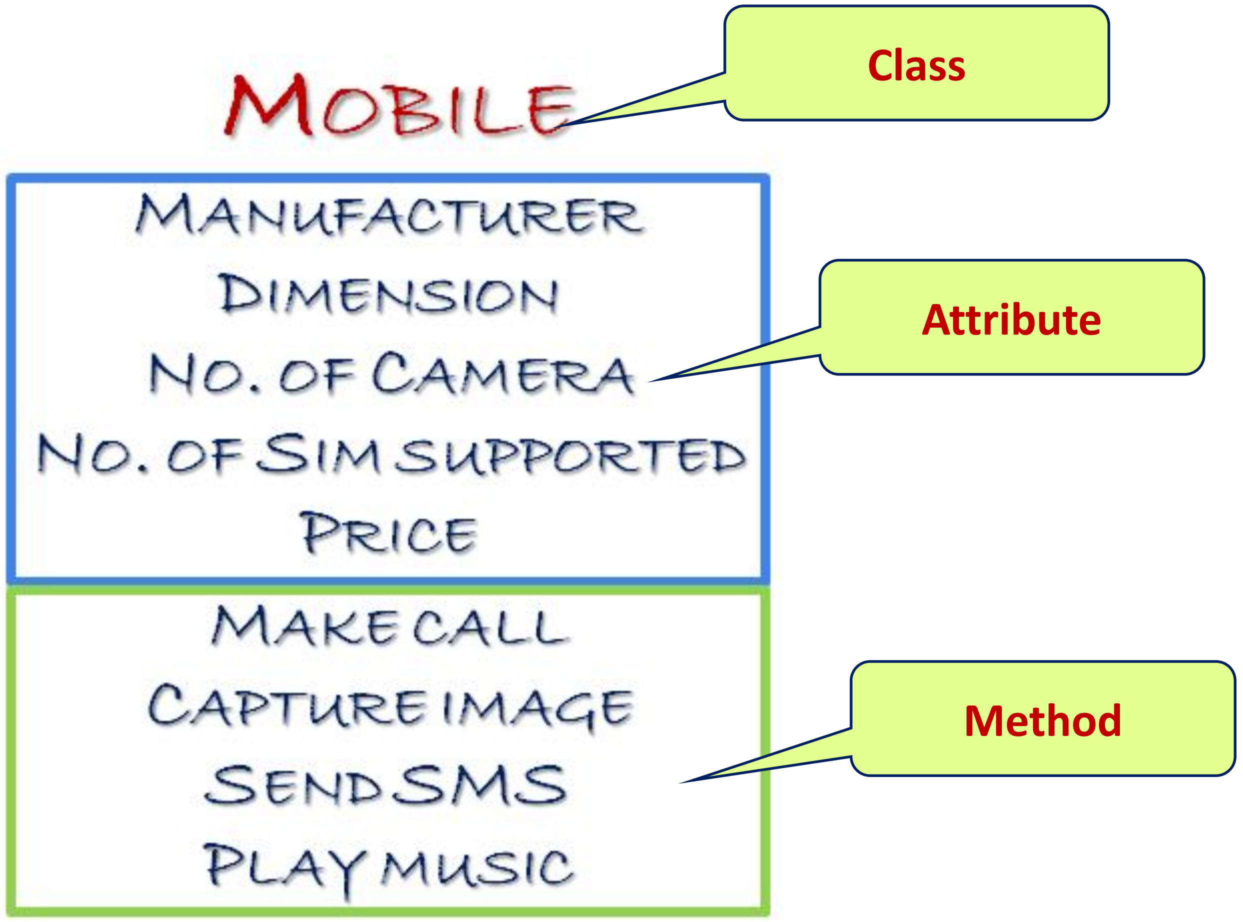
- ✓Make call
- ✓Capture image
- ✓Send SMS
- ✓Play music

Let's take an example

- So far we have defined following things,
- **Class** - Mobile

✔**Attributes –** Manufacturer, Dimension, No. of Camera, No. of SIM supported, Price

✔**Methods**- Make call ,Capture image, Send SMS, Play music



Let's take an example

Now, for different values of attribute (manufacturer, dimension, no. of camera..etc.) in your class, you will get different mobile…

Mobile

Manufacturer Dimension No. of Camera No. of Sim supported Price
Make call Capture image Send SMS Play music

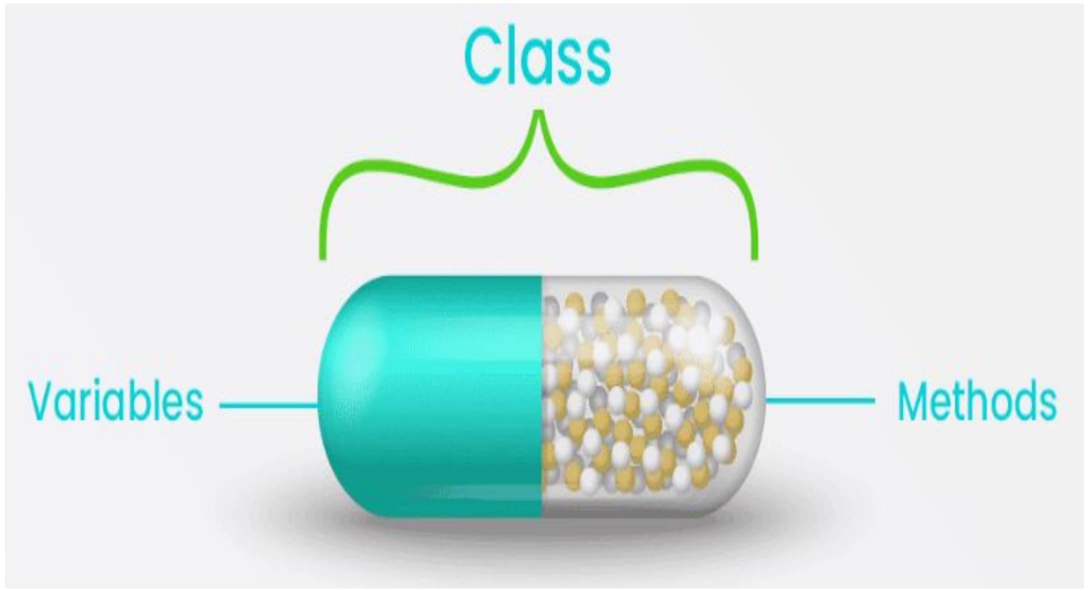
Manufacturer = Apple
Dimension = 164.3 x 74.6 x 8.4 mm
No. of Camera = 3
No. of Sim supported =1
Price = $990

Manufacturer = Samsung
Dimension = 164.3 x 74.6 x 8.4 mm
No. of Camera = 3
No. of Sim supported =2
Price = $440

Manufacturer = Xiaomi
Dimension = 164.3 x 74.6 x 8.4 mm
No. of Camera = 3
No. of Sim supported =2
Price = $440

ENCAPSULATION

- The ability to protect some components of the object from external entities ("private").
- Encapsulation is achieved when each object keeps its state **private**, inside a class.
- Other objects don't have direct access to this state. Instead, they can only call a list of public functions — called methods.
- This is also known as ***hiding.***
- An object **A** can learn about the values of attributes of another object **B**, only by invoking the corresponding method (message) associated to the object **B**.



Example:

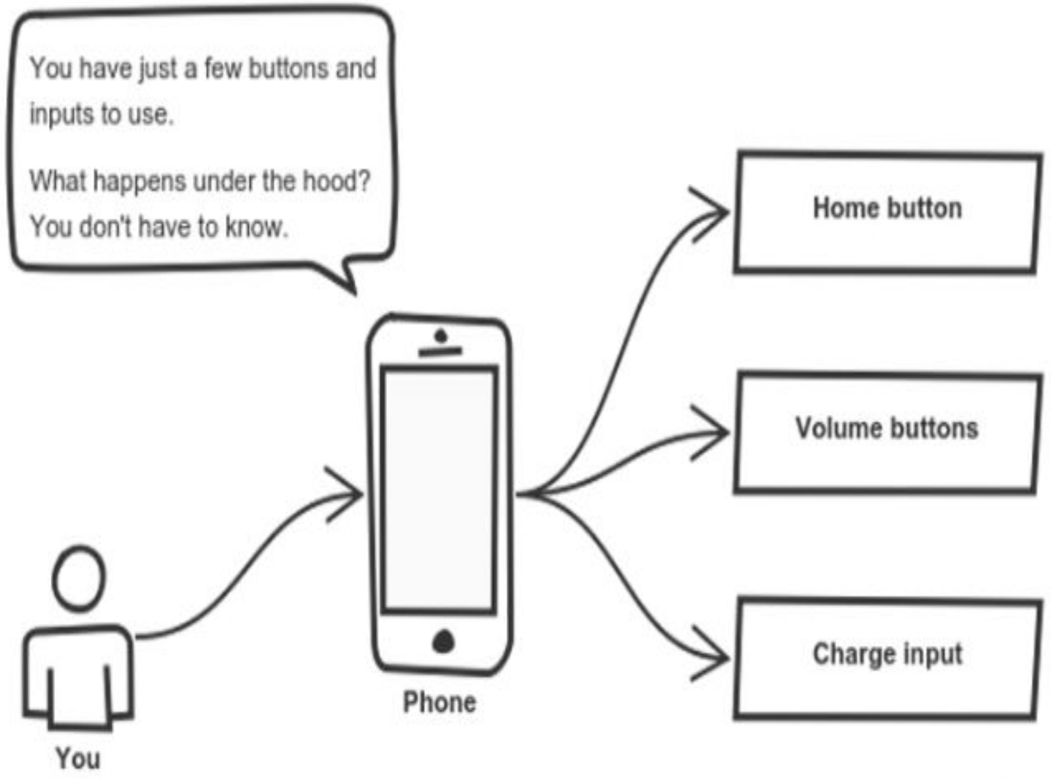
•Consider a BankAccount class where __balance is private.

•Users cannot directly access __balance, but can modify it through public methods like deposit() and withdraw().

ABSTRACTION

- ✔ Abstraction can be thought of as a natural extension of encapsulation.
- ✔ Its main **goal** is to **handle complexity by hiding unnecessary details** from the user.
- ✔ Applying abstraction means that each object should **only** expose a high-level mechanism for using it.
- ✔ This mechanism should **hide** internal implementation details. It should only reveal operations relevant for the other objects.

Think about your mobile again


You have just a few buttons and inputs to use.
What happens under the hood? You don't have to know.
Home button
Volume buttons
Charge input
Phone
You

In an **email system**, users interact with high-level operations like *sending* or *receiving* an email without knowing the inner workings (SMTP, servers, encryption).

Public, Private & Protected

- Public: Accessible from anywhere—inside the class, outside the class, or from any subclass

- Private: Only accessible within the class where they are defined. Neither subclasses nor external classes can access private methods or attributes.

- Protected: (special case) Accessible within the class and by subclasses, but not by external code.

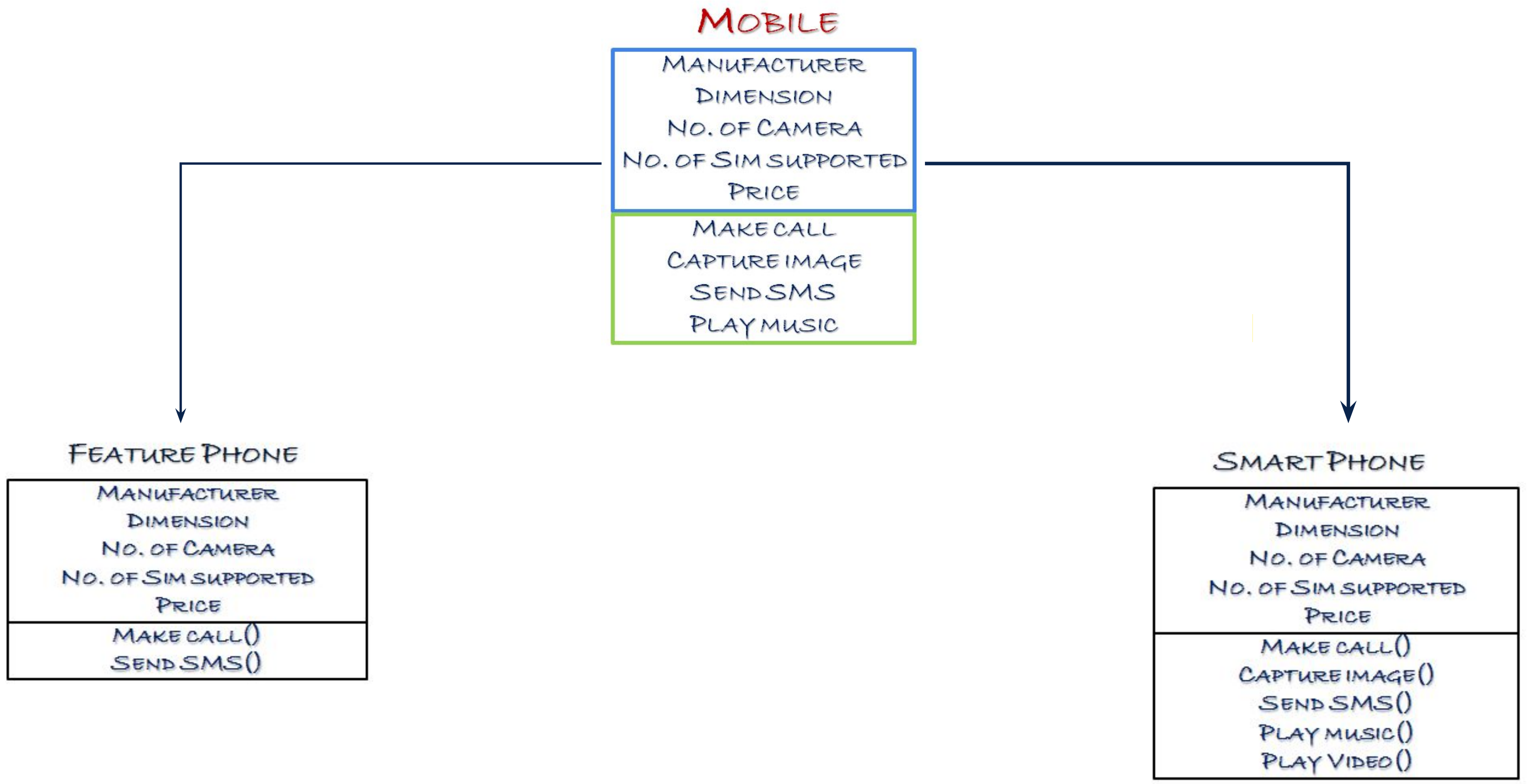
Class Hierarchies & Inheritance

Class Hierarchies & Inheritance

- ✔ A mechanism by which one class (child/subclass) acquires the properties and behaviors (methods) of another class (parent/superclass).
- ✔ Objects are often very similar. They share common logic. But not **entirely** the same
- ✔ A class may have several ancestors, up to Object
- ✔ If you don't specify a superclass, Object is assumed
- ✔ Every class may have one or more subclasses
- ✔ Inheritance is important since it leads to the reusability of code.

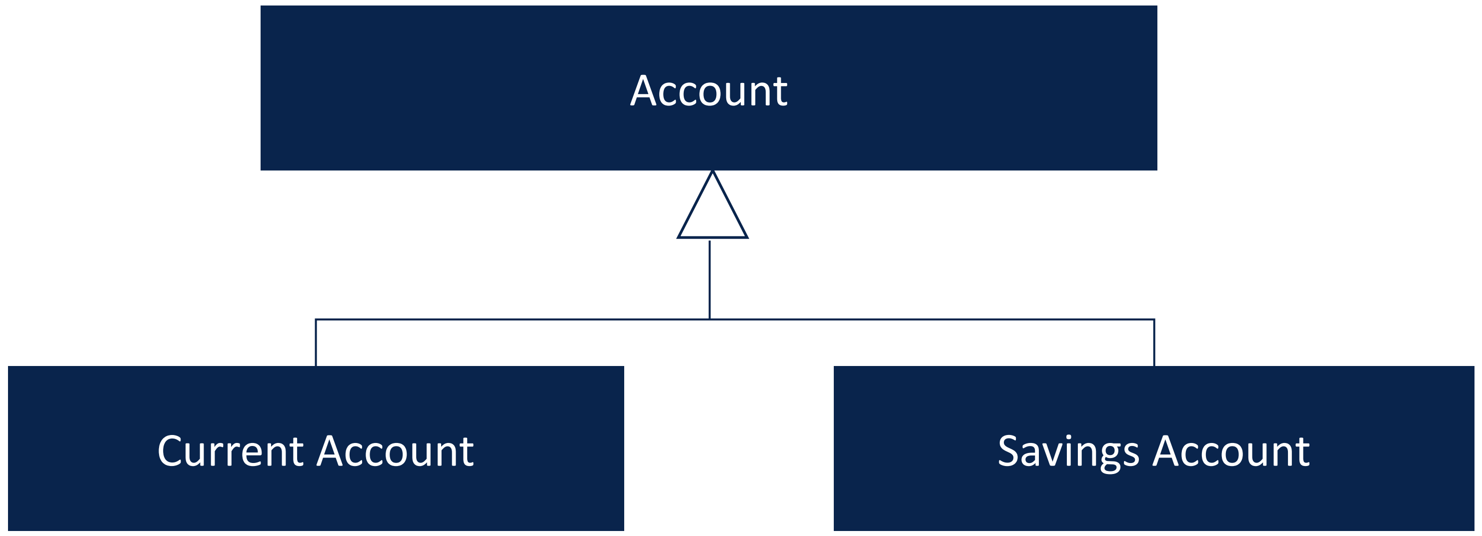
In an **email system**, a Message class could define general methods like send(). A SecureMessage class can inherit from Message and add encryption features specific to secure communication.

Recall the mobile Phone Example



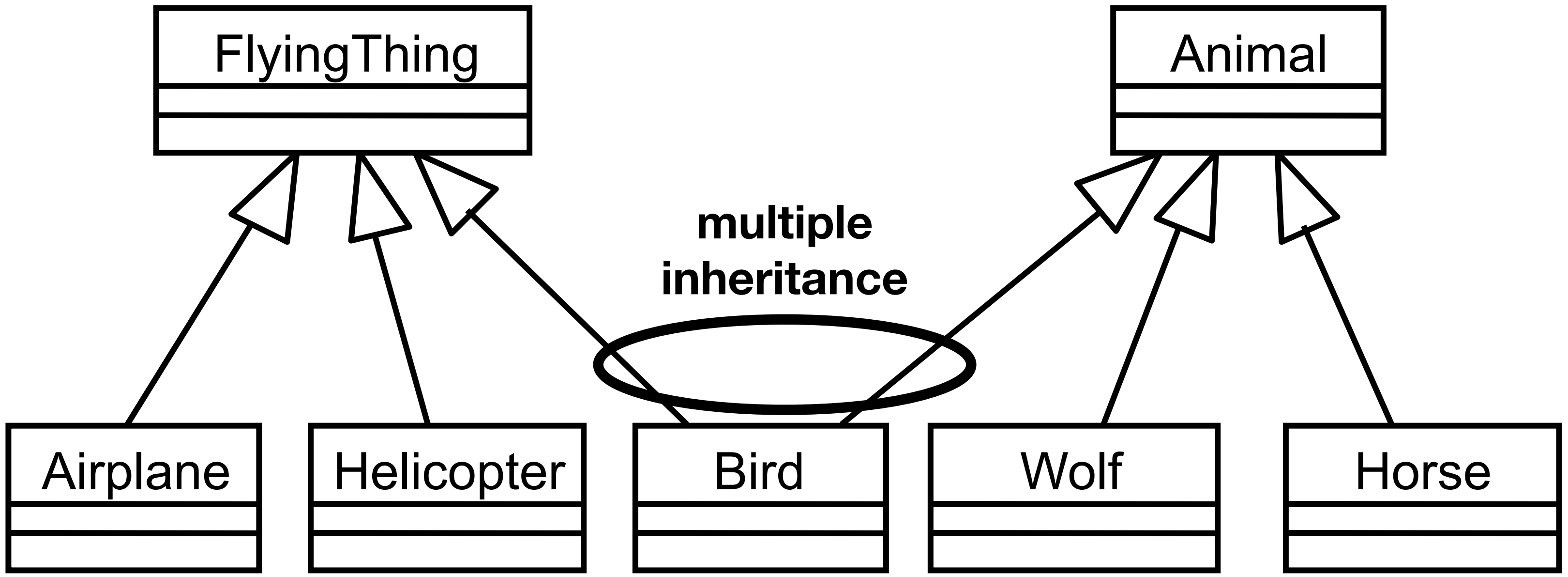
Generalization & Specialization

Generalization
Specialization



MULTIPLE INHERITANCE

A class can inherit from several other classes

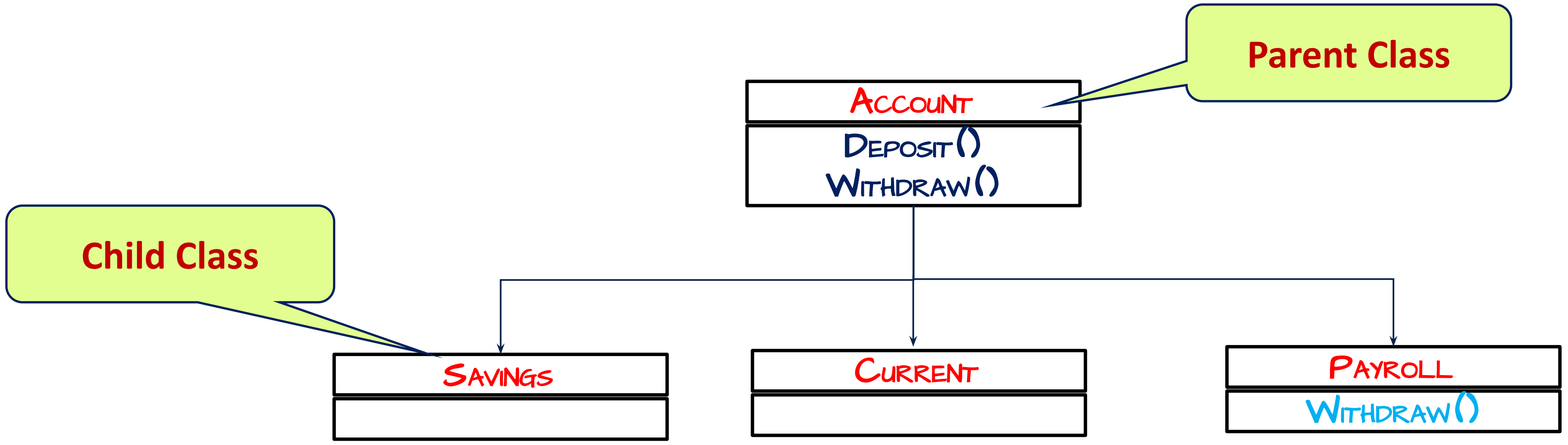


Use multiple inheritance only when needed, and always with caution!!!

POLYMORPHISM

- ✔ Inheritance lets users inherit attributes and methods, and polymorphism uses these methods to perform different tasks.
- ✔ Means that the same method will behave differently when it is applied to the objects of different classes

Let's take an example



The operation of deposit and withdraw is same for Savings and Current accounts. So the inherited methods from Account class will work. However the withdraw method need to be modified for payroll class.

when the "withdraw" method for saving account is called a method from parent account class is executed.

But ,when the "Withdraw" method for the payroll account is called withdraw method defined in the privileged class is executed. This is **Polymorphism in OOPs.**

Method Overriding

- Method Overriding is redefining a super class method in a sub class.
- **Rules for Method Overriding**
- The method signature i.e. method name, parameter list and return type have to match exactly.
- The overridden method can widen the accessibility but not narrow it, i.e. if it is private in the base class, the child class can make it public but not vice versa.

```
class Animal {
    void makeSound() {
        System.out.println("Animal makes a 
sound");
    }
}
class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Dog barks");
    }
}
```

Method Overloading

- Occurs when a class defines multiple methods with the same name but different parameters (number, type, or order of parameters).
- Overloaded methods enable you to perform different actions based on the arguments passed to them, providing flexibility in method usage.
- Method overloading is about defining multiple methods in a class with the same name but different parameters to perform different actions based on the arguments passed.

```
class Calculator {
    int add(int a, int b) {
        return a + b;
    }

    double add(double a, double b) {
        return a + b;
    }
}
```