Chapter: Object Oriented Concepts and Modeling

By

Er. Raj Kiran Chhatkuli

Course Outline

1. *Object oriented concepts*
2. *Object oriented system development*
3. *Identifying the elements of an object model*

Object oriented concepts

Class

- It is a user-defined data type, which has data members and member functions.
- Data members are the *data variables* and member functions are the *functions* or operations used to manipulate these variables and together these data members and member functions define the properties and behaviour of the objects in a Class.
- For example, Consider the Class **'Car'**. There may be many cars with different names and brand but all of them will share some common **properties** like all of them will have 4 wheels, Speed Limit, Mileage range etc. So here, Car is the class and wheels, speed limits, mileage are their properties.
- In the example of class **Car**, the **data member (properties)** will be speed limit, mileage etc and **member functions (operations)** can be apply brakes, increase speed etc.

Object

- An object is an instance of an class.
- **Object** is a bundle of data and its behaviour (often known as methods).
- An Objects have two characteristics: **states** and **behaviors**.
- *For e.g. an object **'House'** has:*
  - ❑ **State**: Address, Color, Area
  - ❑ **Behavior**: Open door, close door
- So if we had to write a class based on states and behaviours of House. We can do it like this: States can be represented as instance variables and behaviours as methods of the class.

Class and Object

```
class House
{
      char Address[20];
      char Color;  char Area[20];
   public:
      void Open door(){ }
      Void Close door(){ }
};
int main()
{
   House h1; // h1 is a object (instance of class House)
}
```

- Let’s take another example of an object **‘Car’:**

❑ **State**: Color, Brand, Weight, Model

❑ **Behavior(Method)**: Break, Accelerate, Slow Down, Gear Change

- As we have seen above, the states and behaviors of an object, can be rep by variables and methods in the class respectively.

- **Characteristics of Objects**
  - Abstraction
  - Encapsulation
  - Message passing

Abstraction

- Abstraction in OOP is like using a TV remote. You don't need to know how the remote communicates with the TV or how the internal electronics work. You just press a button, and it performs a specific function. In the same way, abstraction allows you to focus on what an object does without worrying about the nitty-gritty details of how it achieves those actions.
- One of the most fundamental concept of OOPs is Abstraction. Abstraction is a process where you show only “relevant” data and “hide” unnecessary details of an object from the user.
- ***For example:*** when we login to our **Facebook account**, we enter our user_id (email/PhNumber) & password and press login, what happens when we press login, how the input data sent to facebook server, how it gets verified is all abstracted away from the us.

```
abstract class Animal {
public abstract void makeSound();
}
class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog: Woof Woof!");
    }
}
class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Cat: Meow Meow!");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
Dog myDog = new Dog();
        Cat myCat = new Cat();
        myDog.makeSound();         // Output: Dog: Woof Woof!
        myCat.makeSound();         // Output: Cat: Meow Meow!
    }
}
```

- The Main class demonstrates how you can use the common Animal interface to make different animals produce their respective sounds without worrying about the specific details of each animal.

Encapsulation

Example:

- Consider a smartphone. The inner workings of the phone, such as the processor, battery, and other components, are encapsulated within the device. The user interacts with the phone through a well-defined interface (the screen, buttons, etc.), and the internal details are hidden.

Encapsulation

- Encapsulation is like putting your code in a box and giving it a set of rules. The box, or class, contains data (attributes) and actions (methods), and it decides who can access or modify what's inside. This helps in keeping things organized and preventing unwanted interference.
- Encapsulation is the process of binding the data with the code that manipulates it.
- It keeps the data and the code safe from external interference.

```
public class Book {
    // Private attributes (encapsulation)
    private String title;
    private String author;
    // Public methods to access and modify attributes
    public String getTitle() {
        return title;
    }
    public void setTitle(String newTitle) {
title = newTitle;
    }
    public String getAuthor() {
        return author;
    }
    public void setAuthor(String newAuthor) {
author = newAuthor;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
Book myBook = new Book();
myBook.setTitle("The Adventures of Java");
        myBook.setAuthor("Code Writer");
System.out.println("Title: " + myBook.getTitle()); // Output: Title: The Adventures of Java
System.out.println("Author: " + myBook.getAuthor());  // Output: Author: Code Writer
    }    }
```

In this example, encapsulation ensures that the title and author attributes are not directly accessible from outside the Book class. Instead, you interact with the book's details through the public methods, which allows for validation and controlled access.

- ***Abstraction*** involves simplifying complex systems by modeling classes based on essential properties and behaviors. **It focuses on what an object does rather than how it achieves its functionality.** For example, when dealing with a "Vehicle" class, you abstract away the specific details of cars, bicycles, or trucks and focus on common characteristics like speed or fuel level.

- ***Encapsulation,*** on the other hand, is about bundling the data (attributes) and methods (functions) that operate on the data into a single unit, usually a class. **This unit restricts direct access to some of its components, providing a protective barrier.** Using the "Vehicle" class example, encapsulation ensures that the internal details of how speed is calculated or how fuel is consumed are hidden from the outside world. The object exposes only the necessary interfaces for interaction.

Inheritance

Example:

- Think of vehicles. We can have a base class like Vehicle, and then derive specific classes like Car, Motorcycle, and Truck. The common functionalities such as start, stop, and accelerate can be inherited from the base class.

Inheritance

- Inheritance is like a parent passing down traits to their child. The child shares some characteristics with the parent but might have its own unique features. Similarly, in programming, a subclass can inherit attributes and behaviors from a superclass.
- Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a new class (subclass or derived class) to inherit attributes and behaviors from an existing class (superclass or base class).
- This concept enables code reuse, promotes modularity, and establishes a relationship between classes based on a hierarchy.

```
// Base class (superclass)
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }    }
// Derived class (subclass)
class Dog extends Animal {
    void sound() {
        System.out.println("Dog barks");
    }   }
public class InheritanceExample {
    public static void main(String[] args) {
        // Creating an instance of the derived class (Dog)
        Dog myDog = new Dog();
        // Calling the overridden method in the derived class
        myDog.sound();
    }   }
```

Polymorphism

Example:

- In Java or Python, for instance, the + operator can be used to add numbers (int), concatenate strings (String), or even merge lists. The same operator behaves differently based on the types of operands.

Polymorphism

- Polymorphism means to process objects differently based on their data type.
- In other words it means, ***one method with multiple implementation***, for a certain class of action. And which implementation to be used is *decided at runtime depending upon the situation* (i.e., data type of the object)
- This can be implemented by designing a generic interface, which provides generic methods for a certain class of action and there can be multiple classes, which provides the implementation of these generic methods.

```
class Shape {
    void draw() {
        System.out.println("Drawing a shape");
    }   }
class Circle extends Shape {
    void draw() {
        System.out.println("Drawing a circle");
    }    }
class Square extends Shape {
    void draw() {
        System.out.println("Drawing a square");
    }  }
public class PolymorphismExample {
    public static void main(String[] args) {
        // Creating instances of different shapes
        Shape shape1 = new Circle();
        Shape shape2 = new Square();
        // Calling the draw method on different shapes
        shape1.draw();  // Calls draw method in Circle class
        shape2.draw();  // Calls draw method in Square class
    }   }
```

Object oriented System Development

Function and data methods

- The existing methods for system development can basically be divided into function/data methods and object-oriented methods.
- Function/ data methods *treat functions and/or data as being more or less separate* where as object-oriented methods view functions and data as highly integrated.
- Function/ data methods thus distinguish between functions and data, where *functions, in principle, are active and have behavior*, and *data is a passive* holder of information which is affected by functions.
- The system is typically broken down into functions, whereas data is sent between those functions. The functions are broken down further and eventually converted into source code.

Function and data methods…..

- A system developed using a function/ data method often becomes difficult to maintain. *A major problem with function/ data methods is that, in principle, all functions must know how the data is stored, that is, its data structure.*
- It is often the case that different types of data have slightly different data formats, which means that we often need condition clauses to verify the data type. The programs, therefore, often need to have either IF-THEN or CASE structures, which really do not have anything to do with the functionality, but relate only to the different data formats.
- The programs thus become difficult to read, as we are often not interested in data format, but only in the functionality. Furthermore, to change a data structure, we must modify all the functions relating to the structure.
- Systems developed using such methods often become quite unstable; a slight modification will generate major consequences.

Function and data methods…..

- Another problem with function/ data methods is that people do not naturally think in the terms they structure in.
- The *requirement specification* is normally formulated in normal human language. It often describes in *'What-terms'* what the system will do, what functionality the system will support and what items should exist in the system.
- This is often reformulated into 'How-terms' for the functional breakdown when the focus changes. In this way we create a large semantic gap between the external and internal views of the system.
- Note that nothing in the methods explicitly tells you to change focus to the implementation. We shall see that, with an object-oriented method, the system will even be internally structured from the model taken from the requirement specification.

Object oriented Analysis

- The purpose of object-oriented analysis, as with all other analysis, is to obtain an understanding of the application: *an understanding depending only on the system's functional requirements*.
- The difference between object-oriented analysis and function/data analysis is, as previously mentioned, *the means of expression*. While function/ data analysis methods commence by considering the system's behavior and/or data separately, object oriented analysis combines them and regards them as **integrated objects**. Object-oriented analysis can be characterized as an iteration between analyzing the behavior and information of the system. Moreover, object-oriented analysis uses the object-oriented techniques introduced in the previous chapter.
- Object-oriented analysis contains, in some order, the following activities:
    - ✓ Finding the objects
    - ✓ Organizing the objects
    - ✓ Describing how the objects interact
    - ✓ Defining the operations of the objects
    - ✓ Defining the objects internally

1. Finding the objects:

- Identifying the main components or entities in your software system.

Example: In a library management system, objects could include books, borrowers, and librarians.

2. Organizing the objects:

- Structuring and grouping these identified objects based on their relationships and roles in the system.

Example: Grouping books, borrowers, and librarians into distinct classes or modules.

3. Describing how the objects interact:

- Defining how these objects communicate or collaborate with each other to achieve specific tasks.

Example: Books can be borrowed by borrowers, and librarians manage the borrowing process.

4. Defining the operations of the objects:

- Specifying the actions or methods that each object can perform within the system.

Example: A book object may have methods like "checkOut" and "return," while a librarian object might have methods like "issueFine" or "addNewBook."

5. Defining the objects internally:

- Describing the internal attributes (properties or data) of each object and how it manages its own state.

Example: A book object could have internal attributes like "title," "author," and "availability."

Finding the objects

- The objects can be found as naturally occurring entities in the application domain. An object becomes typically a noun which exists in the domain and, because of this, it is often a good start to learn the terminology for the problem domain. By means of learning what is relevant in the application domain, the objects will be found. It is often the case that there is no problem in finding objects; the difficulty is usually in selecting those objects relevant to the system.
- The aim is to find the essential objects, which are to remain essential throughout the system's life cycle. As they are essential, they will probably always exist and, in this way, we hope to obtain a stable system. Stability also depends on the fact that modifications often begin from some of these items and therefore are local.
- *For example*, in an application for controlling a water tank, typical objects would include Contained Water, Regulator, Valve and Tank; for a banking application typical objects would include Customer, Account, Bank and Clerk.

Finding the objects

- The majority of object-oriented methods today have only one type of object. In this way, one *obtains a simple and general model*. Yet there are reasons for having several different object types: with only one object type, it can be quite difficult to see the difference between different objects. By having different object types, one can more quickly obtain an overview of the system. One can also obtain more support in improving the system's structure by having different rules for different objects. An example of this is that a passive object containing persistent information should not be dependent on objects which deal with the interface, as modifications in the interface are very common.
- Different object types can be organized according to different criteria. Some examples are to *group them by characteristics*, such as *active/ passive, physical/ conceptual, temporary /permanent/ persistent, part/whole, generic/specific, private/public, shared/ non-shared.*

Organizing the objects

- There are a number of criteria to use for the *classification and organization of objects and classes of objects*. One classification starts by considering *how similar classes of objects are to each other*.
- This is normally the basis of inheritance hierarchy; a class can inherit another class. Another classification can be made by considering which objects work with which other objects, or *how an object is a part of anothe*r; for example a house can be built of doors and windows.
- A similar classification is to see which *objects are in some way dependent on another* and thus have modification as a basis of grouping into subsystems.

Object interaction

- In order to obtain a picture of *how the object fits into the system*, we can describe *different scenarios or use cases* in which the object takes part and communicates with other objects.
- In this way, we can fully describe the object's surroundings and what the other objects expect from our object.
- The object's interface can be decided from these scenarios.
- We then also consider how certain objects are part of other objects.

Operations on objects

- The object's operations come naturally when we consider an object's interface.
- The operations can also be identified directly from the application, when we consider what can be done with the items we model.
- They can be primitive (for example, create, add, delete) or more complex such as putting together some report of information from several objects.
- If one obtains very complex operations, new objects can be identified from them. Generally, it is better to avoid objects that are too complex.

Object implementation

- Finally, the object should be defined internally, which includes defining the information that each object must hold.
- Even the number of instances that can be created of each object is interesting; one may wish to have alternative ways of storing information. Some of the attributes can be inherited.

Object oriented construction

- Object-oriented construction means that the analysis model is designed and implemented in source code.
- This source code is executed in the target environment, which often means that the ideal model produced by the analysis model must be molded to fit into the implementation environment.

- For details see book (I. Jacobson) pg. 79

Object Oriented Design

- Object–Oriented Design (OOD) involves implementation of the conceptual model produced during object-oriented analysis.
- In OOD, concepts in the analysis model, which are technology−independent, are mapped onto *implementing classes, constraints are identified and interfaces are designed, resulting in a model for the solution domain*, i.e., a detailed description of how the system is to be built on concrete technologies.
- Grady Booch has defined object-oriented design as
  - “A method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design”.

Object Oriented Design

- The implementation details generally include −

❑ Restructuring the class data (if necessary),

❑ Implementation of methods, i.e., internal data structures and algorithms,

❑ Implementation of control, and

❑ Implementation of associations.

Object Oriented Programming

- Object-oriented programming (OOP) is a programming paradigm based upon objects (having both data and methods) that aims to incorporate the advantages of modularity and reusability. Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs.

- Grady Booch has defined object–oriented programming as
  - “A method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships”.

Object Oriented Programming

- The important features of object–oriented programming are −
  - ❑ Bottom–up approach in program design
  - ❑ Programs organized around objects, grouped in classes
  - ❑ Focus on data with methods to operate upon object’s data
  - ❑ Interaction between objects through functions
  - ❑ Reusability of design through creation of new classes by adding features to existing classes

- Some examples of object-oriented programming languages are C++, Java, Smalltalk, Delphi, C#, Perl, Python, Ruby, and PHP.

  - details on book (I Jacobson) pg.84

Identifying Elements of Object Model

Identifying Elements of Object Model

- Identifying Classes and Objects
- Specifying Attributes
- Defining Operations
- Finalizing the Object Definition

1. Identifying Classes and Objects:

Classes: Think of classes as blueprints or templates for objects. They define the properties and behaviors that objects created from them will have.

Objects: Objects are instances of classes. They are real-world entities that have state (attributes) and behavior (operations/methods).

Example:

Class: Car

Objects: Toyota Camry, Ford Mustang

2. Specifying Attributes:

Attributes: These are the characteristics or properties that define the state of an object.

Example:

Attributes for Car class: color, model, year, fuelType

3. Defining Operations:

Operations/Methods: These are the actions or behaviors that objects can perform. They define what an object can do.

Example:

Operations for Car class: startEngine(), accelerate(speed), brake()

4. Finalizing the Object Definition:

Once you've identified classes, specified attributes, and defined operations, you finalize the object definition by *creating instances of the class*.

```
class Car:
    # Attributes
    def __init__(self, color, model, year, fuel_type):
        self.color = color
        self.model = model
        self.year = year
        self.fuel_type = fuel_type
    # Operations
    def start_engine(self):
        print("Engine started.")
    def accelerate(self, speed):
        print(f"Accelerating to {speed} mph.")
    def brake(self):
        print("Braking.")
```

```
# Creating objects
car1 = Car("Blue", "Toyota Camry", 2022, "Gasoline")
car2 = Car("Red", "Ford Mustang", 2023, "Diesel")
# Using object methods
car1.start_engine()
car1.accelerate(60)
car2.brake()
```

- In this example, we have a Car class with attributes (color, model, year, fuel_type) and operations (start_engine(), accelerate(speed), brake()). We create two car objects (car1 and car2) and use their methods.

See More examples

Identifying Classes and Objects..

- If we look around a room, there is a set of physical objects that can be easily identified, classified, and defined (in terms of attributes and operations). But when we "look around" the problem space of a software application, the objects may be more difficult to comprehend.

- We can begin to identify objects by examining the problem statement or performing a "grammatical parse" on the processing narrative for the system to be built.

- Objects are determined by underlining each noun or noun clause and entering it in a simple table.

Identifying Classes and Objects..

An objects can be:

- **External entities** (e.g., other systems, devices, people) that produce or consume
  - information to be used by a computer-based system.
- **Things** (e.g., reports, displays, letters, signals) that are part of the information domain for the problem.
- **Occurrences or events** (e.g., a property transfer or the completion of a series of robot movements) that occur within the context of system operation.
- **Roles** (e.g., manager, engineer, salesperson) played by people who interact with the system.
- **Organizational units**(e.g., division, group, team) that are relevant to an application.
- **Places** (e.g., manufacturing floor or loading dock) that establish the context of the problem and the overall function of the system.
- **Structures** (e.g., sensors, four-wheeled vehicles, or computers) that define a
  - class of objects or in the extreme, related classes of objects.

Identifying Classes and Objects..

To illustrate how objects might be defined during the early stages of analysis, lets take an example of SafeHome security system. The processing narrative for SafeHome is as follow:

- SafeHome software enables the homeowner to configure the security system when it is installed, monitors all sensors connected to the security system, and interacts with the homeowner through a keypad and function keys contained in the SafeHome control panel.
- During installation, the SafeHome control panel is used to "program" and configure the system. Each sensor is assigned a number and type, a master password is programmed for arming and disarming the system, and telephone number(s) are input for dialing when a sensor event occurs.
- When a sensor event is sensed by the software, it rings an audible alarm attached to the system. After a delay time that is specified by the homeowner during system configuration activities, the software dials a telephone number of a monitoring service, provides information about the location, reporting and the nature of the event that has been detected.
- The number will be redialed every 20 seconds until telephone connection is obtained. All interaction with SafeHome is managed by a user-interaction subsystem that reads input provided through the keypad and function keys, displays prompting messages on the LCD display, displays system status information on the LCD display. Keyboard interaction takes the following form . . .

Identifying Classes and Objects..

Extracting the nouns, we can propose a number of potential objects:

Potential Object/Class	General Classification
homeowner	role or external entity
sensor	external entity
control panel	external entity
installation	occurrence
system (alias security system)	thing
number, type	not objects, attributes of sensor
master password	thing
telephone number	thing
sensor event	occurrence
audible alarm	external entity
monitoring service	organizational unit or external entity

Note:

The list would be continued until all nouns in the processing narrative have been considered. Note that we call each entry in the list a potential object. We must consider each further before a final decision is made

Identifying Classes and Objects..

Coad and Yourdon suggest six selection characteristics that should be used to considers each potential object for inclusion in the analysis model:

1.Retained information

The potential object will be useful during analysis only if information about it must be remembered.

2.Needed services

The potential object must have a set of identifiable operations that can change the value of its attributes in some way.

3.Multiple attributes

During requirement analysis, the focus should be on "major" information; an object with a single attribute may, in fact, be useful during design, but is probably better represented as an attribute of another object during the analysis activity.

Identifying Classes and Objects..

4. Common attributes

A set of attributes can be defined for the potential object and these attributes apply to all occurrences of the object.

5.Common operations

A set of operations can be defined for the potential object and these operations apply to all occurrences of the object.

6.Essential requirements

External entities that appear in the problem space and produce or consume information essential to the operation of any solution for the system will almost always be defined as objects in the requirements model.

To be considered a legitimate object for inclusion in the requirements model, a potential object should satisfy all (or almost all) of these characteristics. With this in mind, we apply the selection characteristics to the list of potential SafeHome objects:

Potential Object/Class	Characteristic Number That Applies
homeowner	rejected: 2 fails
sensor	accepted
control panel	accepted
installation	rejected
system (alias security system)	accepted
number, type	rejected: 3 fails, attributes of sensor
master password	rejected: 3 fails
telephone number	rejected: 3 fails
sensor event	accepted
audible alarm	accepted
monitoring service	rejected: 2 fails

Note:

1. *The preceding list is not all-inclusive, additional objects would have to be added to complete the model.*
2. *Some of the rejected potential objects will become attributes for those objects that were accepted (e.g., number and type are attributes of sensor, and master password and telephone number may become attributes of system)*
3. *Different statements of the problem might cause different "accept or reject" decisions to be made (e.g., if each homeowner had an individual password or was identified by voice print, the homeowner object would satisfy characteristics 2 and would have been accepted).*

Specifying Attributes

- Attributes describe an object that has been selected for inclusion in the analysis model. In essence, it is the attributes that define the object—that clarify what is meant by the object in the context of the problem space.
- For example, if we were to build a system that tracks baseball statistics for professional baseball players, the attributes of the object **player** would be quite different than the attributes of the same object when it is used in the context of the professional baseball pension system. In the former, attributes such as **name, position, batting average, fielding percentage, years played, and games played** might be relevant. For the latter, some of these attributes would be meaningful, but others would be replaced (or augmented) by attributes like **average salary, credit toward full vesting, pension plan options chosen, mailing address** etc.

Specifying Attributes

- To develop a meaningful set of attributes for an object, the analyst can again study the processing narrative (or statement of scope) for the problem and select those things that reasonably "belong" to the object.
- To illustrate, we consider the system object defined for SafeHome. Homeowner can configure the security system to reflect sensor information, alarm response information, activation/deactivation information, identification information, and so forth. Using the content description notation defined for the data dictionary , we can represent these composite data items in the following manner:

  - ❑ **sensor information**: sensor type, sensor number, alarm threshold
  - ❑ **alarm response information** = delay time , telephone number, alarm type
  - ❑ **activation/deactivation information** = master password, number of allowable tries , temporary password
  - ❑ **identification information** = system ID, verification phone number, system status

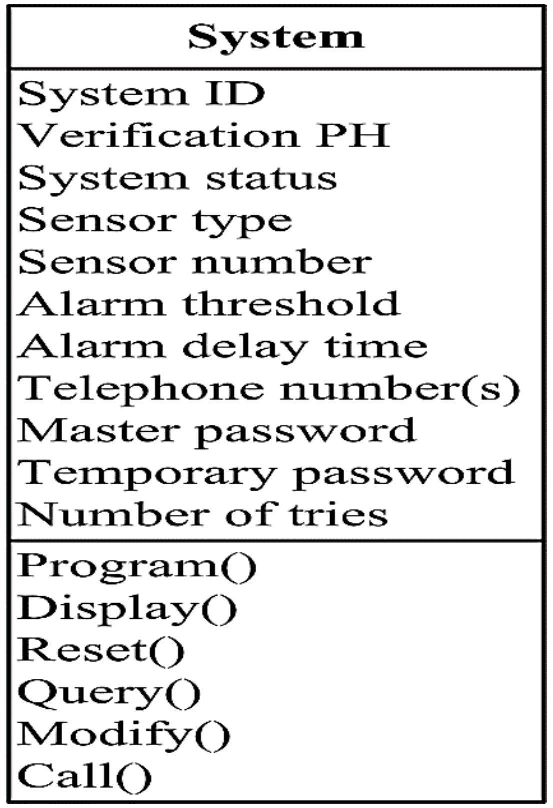
Defining Operations

- Operations define the behavior of an object and change the object’s attributes in some way. More specifically, an operation changes one or more attribute values that are contained within the object.
- Although many different types of operations exist, they can generally be divided into three broad categories:
  - operations that manipulate data in some way (e.g., adding, deleting, reformatting, selecting)
  - operations that perform a computation, and
  - operations that monitor an object for the occurrence of a controlling event.

Finalizing the Object\ Class Definition

- The definition of operations is the last step in completing the specification of an object.
- For example, sensor event will send a message to system to display the event location and number; control panel will send system a reset message to update system status; the audible alarm will send a query message; the control panel will send a modify message to change one or more attributes without reconfiguring the entire system object; sensor event will also send a message to call the phone number(s) contained in the object. Other messages can be considered and operations derived. So the resulting object definition for object **system** is shown in figure **(next slide)**

Finalizing the Object\ Class Definition


System
System ID
Verification PH
System status
Sensor type
Sensor number
Alarm threshold
Alarm delay time
Telephone number(s)
Master password
Temporary password
Number of tries
Program()
Display()
Reset()
Query()
Modify()
Call()