Chapter – 1

Object Oriented Software Development Concepts

Objectives

At the end of this Chapter, the students will be able to:

- Differentiate the concept of procedural paradigm and object oriented paradigm.
- Analyze any scenario in OOAD perspective.
- Evaluate OOAD concepts.
- Analyze class and object relationship

1.0 Introduction

"A physician, a civil engineer, and a computer scientist were arguing about what was the oldest profession in the world. The physician remarked, "Well, in the Bible, it says that God created Eve from a rib taken out of Adam. This clearly required surgery, and so I can rightly claim that mine is the oldest profession in the world." The civil engineer interrupted, and said, "But even earlier in the book of Genesis, it states that God created the order of the heavens and the earth from out of the chaos.

This was the first and certainly the most spectacular application of civil engineering. Therefore, fair doctor, you are wrong: mine is the oldest profession in the world." The computer scientist leaned back in her chair, smiled, and then said confidently, "Ah, but who do you think created the chaos?" "The more complex the system, the more open it is to total breakdown". Rarely would a builder think about adding a new sub-basement to an existing 100-story building. Doing that would be very costly and would undoubtedly invite failure. Amazingly, users of software systems rarely think twice about asking for equivalent changes. Besides, they argue, it is only a simple matter of programming."

1.1 Examining Object Orientation

OO concepts affect the whole development process:

- Humans think in terms of nouns (objects) and verbs (behaviours of objects).
- With OOSD, both problem and solution domains are modelled using OO concepts.

• The *Unified Modeling Language* (UML) is a de facto standard for modeling OO Software.
• OO languages bring the implementation closer to the language of mental models. The UML is a good bridge between mental models and implementation.

1.1.1. Comparing the Procedural and OO Paradigms

	Procedural Paradigm	**OO Paradigm**
Organizational structure	Focuses on hierarchy of procedures and sub procedures Data is separate from procedures	Network of collaborating objects Methods (processes) are often bound together with the state (data) of the object
Protection against modification or access	Data is difficult to protect against inappropriate modifications or access when it is passed to or referenced by many different procedures.	The data and internal methods of objects can be protected against inappropriate modifications or access by using encapsulation
Ability to modify software	Can be expensive and difficult to make software that is easy to change, resulting in many "Brittle" systems	Robust software that is easy to change, if written using good OO principles and patterns
Reuse	Reuse of methods is often achieved by copy-and-paste or 1001 parameters.	Reuse of code by using generic components (one or more objects) with well-defined interfaces. This is achieved by extension of classes (or interfaces) or by composition of objects.
Configuration of special cases	Often requires if or switch statements. Modification is risky because it often requires altering existing code. So, modifications must be done with extreme care apart from requiring extensive regression testing. These factors make even minor changes costly to implement.	Polymorphic behavior can facilitate the possibility of modifications being primarily additive, subtractive, or substitution of whole components (one or more objects); thereby, reducing the associated risks and costs

There are four major elements in the Object Orientation model:
1. Abstraction
2. Encapsulation
3. Modularity
4. Hierarchy

 **By major, we mean that a model without any one of these elements is not object oriented.**

There are three minor elements of the object model:

1. Typing

   In OOP, a class is visualized as a type having properties distinct from any other types. Typing is the enforcement of the notion that an object is an instance of a single class or type. It also enforces that objects of different types may not be generally interchanged; and can be interchanged only in a very restricted manner if absolutely required to do so.

The two types of typing are:

- **Strong Typing**: Here, the operation on an object is checked at the time of compilation, as in the programming language Eiffel.

- **Weak Typing**: Here, messages may be sent to any class. The operation is checked only at the time of execution, as in the programming language Smalltalk.

2. Concurrency

   Concurrency in operating systems allows performing multiple tasks or processes simultaneously.

   Concurrency simply means simultaneous execution of more than one thread. True concurrency requires hardware support in the form of multiple execution units, (multi-processor systems or hyper threading for example). Otherwise concurrency is 'simulated' by context switching between multiple threads.

3. Persistence

   An object occupies a memory space and exists for a particular period of time. In traditional programming, the lifespan of an object was typically the lifespan of the execution of the program that created it. In files or databases, the object lifespan is longer than the duration of the process creating the object. This property by which an object continues to exist even after its creator ceases to exist is known as persistence.

    **By minor, we mean that each of these elements is a useful, but not essential, part of the object model.**

concentrate on the outside view of an object and leads us to what Meyer calls the *contract model* of programming. The outside view of each object defines a contract on which other objects may depend, and which in turn must be carried out by the inside view of the object itself.

This contract thus establishes all the assumptions a client object may make about the behavior of a server object. In other words, this contract encompasses the responsibilities of an object, namely, the behavior for which it is held accountable

1.1.3. The Meaning of Encapsulation

Abstraction and encapsulation are complementary concepts: Abstraction focuses on the observable behavior of an object, whereas encapsulation focuses on the implementation that gives rise to this behavior. Encapsulation is most often achieved through information hiding (not just data hiding), which is the process of hiding all the secrets of an object that do not contribute to its essential characteristics; typically, the structure of an object is hidden, as well as the implementation of its methods. "No part of a complex system should depend on the internal details of any other part"**.** Whereas abstraction "helps people to think about what they are doing," encapsulation "allows program changes to be reliably made with limited effort".

Hiding is a relative concept: What is hidden at one level of abstraction may represent the outside view at another level of abstraction. The underlying representation of an object can be revealed, but in most cases only if the creator of the abstraction explicitly exposes the implementation, and then only if the client is willing to accept the resulting additional complexity. Thus, encapsulation cannot stop a developer from doing stupid things**.**

Of course, no programming language prevents a human from literally seeing the implementation of a class, although an operating system might deny access to a particular file that contains the implementation of a class.

1.1.4. The Meaning of Modularity

"The act of partitioning a program into individual components can reduce its complexity to some degree. Although partitioning a program is helpful for this reason, a more powerful justification for partitioning a program is that it creates a number of well defined, documented boundaries within the program. These boundaries, or interfaces, are invaluable in the comprehension of the program"**.**

Most languages that support the module as a separate concept also distinguish between the interface of a module and its implementation. Thus, it is fair to say that modularity and encapsulation go hand in hand.

Deciding on the right set of modules for a given problem is almost as hard a problem as deciding on the right set of abstractions.

Modules serve as the physical containers in which we declare the classes and objects of our logical design.

Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.

Thus, the principles of abstraction, encapsulation, and modularity are synergistic. An object provides a crisp boundary around a single abstraction, and both encapsulation and modularity provide barriers around this abstraction.

Abstraction is a good thing, but in all except the most trivial applications, we may find many more different abstractions than we can comprehend at one time.

Encapsulation helps manage this complexity by hiding the inside view of our abstractions. Modularity helps also, by giving us a way to cluster logically related abstractions. Still, this is not enough. A set of abstractions often forms a hierarchy, and by identifying these hierarchies in our design, we greatly simplify our understanding of the problem.

1.1.5 The Meaning of Hierarchy

Hierarchy is a ranking or ordering of abstractions. The two most important hierarchies in a complex system are its class structure (the "is a" hierarchy) and its object structure (the "part of" hierarchy).

1.1.5.1. Examples of Hierarchy: Single Inheritance

Inheritance is the most important "is a" hierarchy, and as we noted earlier, it is an essential element of object-oriented systems. Basically, inheritance defines a relationship among classes, wherein one class shares the structure or behavior defined in one or more classes (denoting single inheritance and multiple inheritances, (respectively). Inheritance thus represents a hierarchy of abstractions, in which a subclass inherits from one or more super classes. Typically, a subclass augments or redefines the existing structure and behavior of its super classes.

There is a healthy tension among the principles of abstraction, encapsulation, and hierarchy. "Data abstraction attempts to provide an opaque barrier behind which methods and state are hidden; inheritance requires opening this interface to some extent and may allow state as well as methods to be accessed without abstraction"

1.1.5.2. Examples of Hierarchy: Aggregation

Whereas these "is a" hierarchies denote generalization/specialization relationships, "part of" hierarchies describe aggregation relationships. For example, consider the abstraction of a garden. We can contend that a garden consists of a collection of plants together with a growing plan. In other words, plants are "part of" the garden, and the growing plan is "part of" the garden. This "part of" relationship is known as **aggregation**.

Aggregation is not a concept unique to object-oriented development or object oriented programming languages. Indeed, any language that supports record-like structures supports aggregation. However, the combination of inheritance with aggregation is powerful: Aggregation permits the physical grouping of logically related structures, and inheritance allows these common groups to be easily reused among different abstractions.

1.1.6. The Meaning of Typing

The concept of a type derives primarily from the theories of abstract data types. "A type is a precise characterization of structural or behavioral properties which a collection of entities all share". For our purposes, we will use the terms *type* and *class* interchangeably.

Although the concepts of a type and a class are similar, we include typing as a separate element of the object model because the concept of a type places a very different emphasis on the meaning of abstraction. Specifically, we state the following:

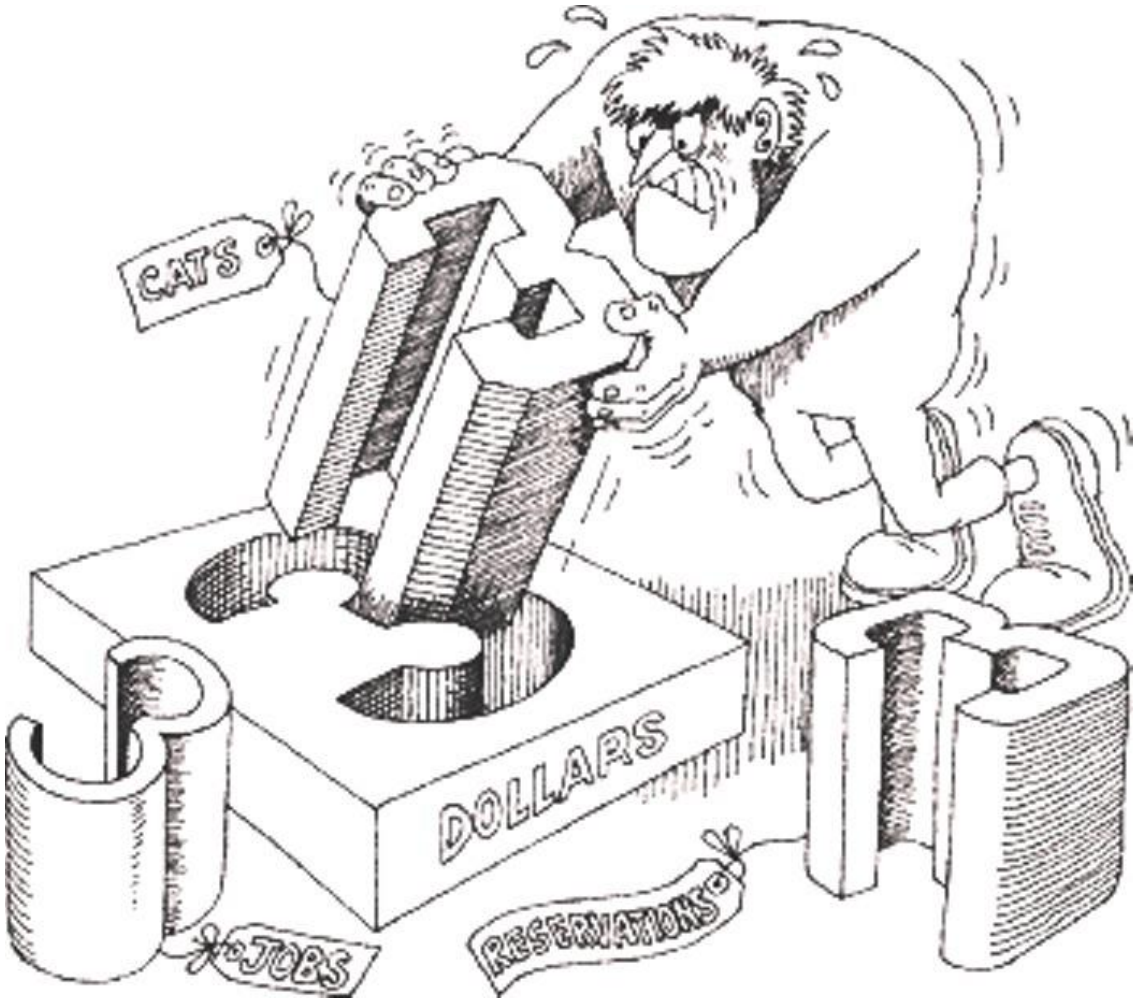

Figure 1.2
Strong typing prevents mixing of abstractions

Typing is the enforcement of the class of an object, such that objects of different types may not be interchanged, or at the most, they may be interchanged only in very restricted ways. Typing lets us express our abstractions so that the programming language in which we implement them can be made to enforce design decisions.

 **When we use object-oriented methods to analyze or design a complex software system, our basic building blocks are classes and objects.**

1.2 The Nature of an Object

 **The ability to recognize physical objects is a skill that humans learn at a very early age.**

Example 1.1

A brightly colored ball will attract an infant's attention, but typically, if you hide the ball, the child will not try to look for it; when the object leaves her field of vision, as far as she can determine, it ceases to exist. It is not until near the age of one that a child normally develops what is called the object concept, a skill that is of critical importance to future cognitive development. Show a ball to a one-year-old and then hide it, and she will usually search for it even though it is not visible. Through the object concept, a child comes to realize that objects have a permanence and identity apart from any operations on them

1.2.1. What Is and What Isn't an Object

An object is a tangible entity that exhibits some well-defined behavior. From the perspective of human cognition, an object is any of the following:

- A tangible and/or visible thing.
- Something that may be comprehended intellectually.
- Something toward which thought or action is directed.

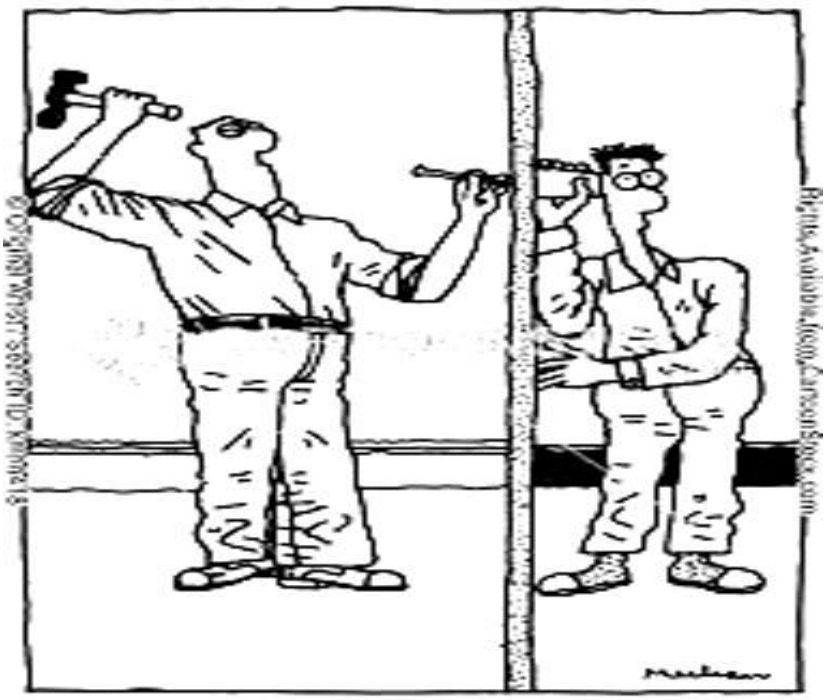

Figure 1.3

Example 1.2

Just as the person holding a hammer tends to see everything in the world as a nail, so the developer with an object-oriented mindset begins to think that everything in the world is an object. This perspective is a little naive because some things are distinctly not objects.

For example, attributes such as beauty or color are not objects, nor are emotions such as love and anger. On the other hand, these things are all potentially properties of other objects. For example, we might say that a man (an object) loves his son(another object), or that a particular cat (yet another object) is gray.

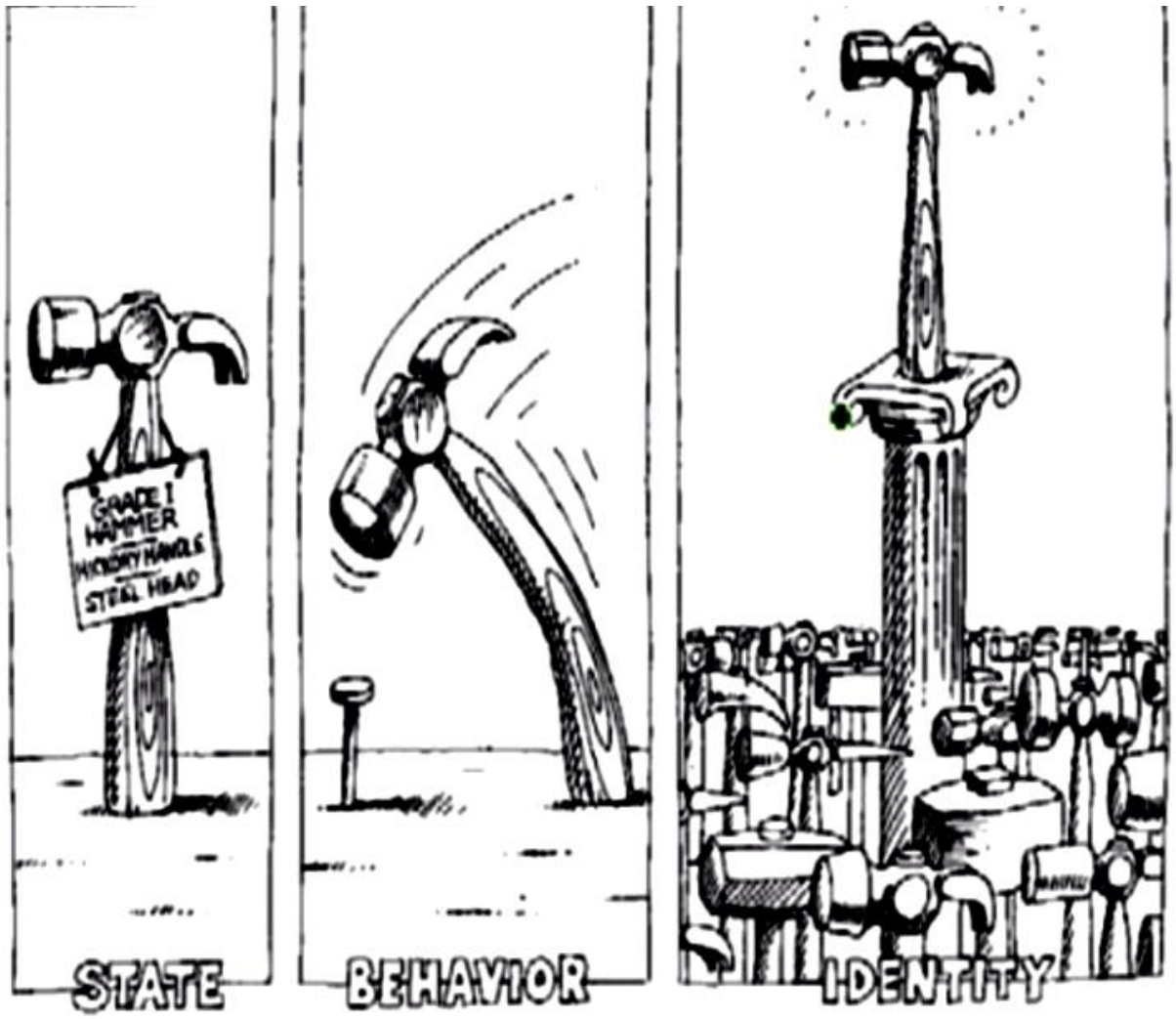

Figure 1.4
An object has state, exhibits some well-defined behavior, and has a unique identity

An object is an entity that has state, behavior, and identity. The structure and behavior of similar objects are defined in their common class. The terms *instance* and *object* are interchangeable.

1.2.1.1. State

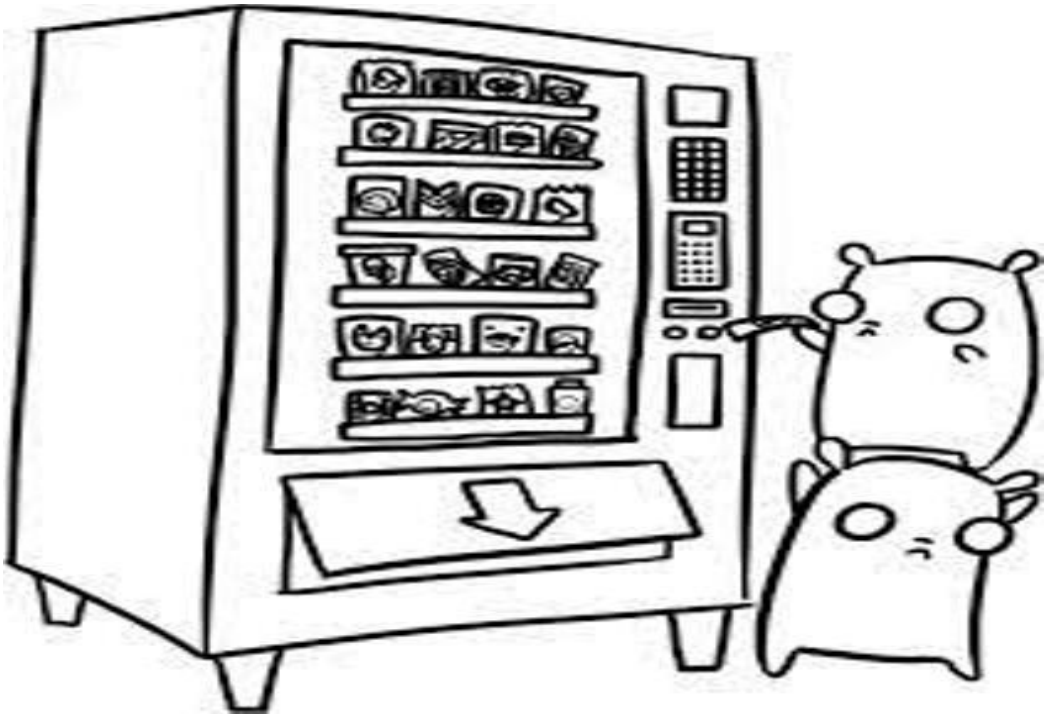

Figure 1.5

Example 1.4
Consider a vending machine that dispenses soft drinks. The usual behavior of such objects is that when someone puts money in a slot and pushes a button to make a selection, a drink emerges from the machine. What happens if a user first makes a selection and then puts money in the slot? Most vending machines just sit and do nothing because the user has violated the basic assumptions of their operation.

Stated another way, the vending machine was in a state (of waiting for money) that the user ignored (by making a selection first). Similarly, suppose that the user ignores the warning light that says, "Correct change only," and puts in extra money. Most machines are user-hostile; they will happily swallow the excess money.

In each of these circumstances, we see how the behavior of an object is influenced by its history: The order in which one operates on the object is important. The reason for this event- and time-dependent behavior is the existence of state within the object.

From Above example we conclude the definition of an Object as follows:

The state of an object encompasses all of the (usually static) properties of the object plus the current (usually dynamic) values of each of these properties.

1.2.1.2. Behavior

No object exists in isolation. Rather, objects are acted on and they act on other objects. Thus, we may say the following:

Behavior is how an object acts and reacts, in terms of its state changes and message passing. In other words, the behavior of an object represents its outwardly visible activity. An operation is some action that one object performs on another in order to elicit a reaction. For example, a client might invoke the operations append and pop to grow and shrink a queue object, respectively. A client might also invoke the operation length, which returns a value denoting the size of the queue object but does not alter the state of the queue itself.

Message passing is one part of the equation that defines the behavior of an object; our definition for behavior also notes that the state of an object affects its behavior as well. Consider again the vending machine example. We may invoke some operation to make a selection, but the vending machine will behave differently depending on its state. If we do not deposit change sufficient for our selection, the machine will probably do nothing.

If we provide sufficient change, the machine will take our change and then give us our selection (thereby altering its state).

Most interesting objects do not have state that is static; rather, their state has properties whose values are modified and retrieved as the object is acted on. The behavior of an object is embodied in the sum of its operations.

1.2.1.3 Identity

Identity is that property of an object which distinguishes it from all other objects.

An object retains its identity even if some or all of the values of variables or definitions of methods change over time.

Several forms of identity:

- **value**: A data value is used for identity (e.g., the primary key of a tuple in a relational database).
- **name**: A user-supplied name is used for identity (e.g., file name in a file system).
- **built-in**: A notion of identity is built-into the data model or programming languages, and no user-supplied identifier is required (e.g., in OO systems).

Example 1.5

Identity is the property of a thing which distinguishes it from all other objects. Humans have id numbers, fingerprints, DNA profiles. All these are representations of the fact that

we are each unique and identifiable. This element of the object model can be confused with ***state***. State is the set of values that an object encapsulates. Two objects may have identical state, and yet are still separate, distinct, identifiable objects.
Objects in an OO system have distinct identity. It is part and parcel of what makes them an object.

1.2.1.4. Operations

An operation denotes a service that a class offers to its clients. In practice, we have found that a client typically performs five kinds of operations on an object.

The three most common kinds of operations are the following:

1. Modifier: an operation that alters the state of an object
2. Selector: an operation that accesses the state of an object but does not alter the state
3. Iterator: an operation that permits all parts of an object to be accessed in some well defined order.

Two other kinds of operations are common; they represent the infrastructure necessary to create and destroy instances of a class.

1. Constructor: an operation that creates an object and/or initializes its state.
2. Destructor: an operation that frees the state of an object and/or destroys the object itself.

1.2.1.5. Roles and Responsibilities

Collectively, all of the methods associated with a particular object comprise its protocol.
The protocol of an object thus defines the envelope of an object's allowable behavior and so comprises the entire static and dynamic view of the object.

For most nontrivial abstractions, it is useful to divide this larger protocol into logical groupings of behavior. These collections, which thus partition the behavior space of an object, denote the roles that an object can play. A role is a mask that an object wears and so defines a contract between an abstraction and its clients.

Figure 1.6

Objects can play many different roles as man plays

Responsibilities are meant to convey a sense of the purpose of an object and its place in the system. The responsibilities of an object are all the services it provides for all of the contracts it supports". other words, we may say that the state and behavior of an object collectively define the roles that an object may play in the world, which in turn fulfill the abstraction's responsibilities.

Indeed, most interesting objects play many different roles during their lifetime. Consider the following examples

Example 1.6

A bank account may have the role of a monetary asset to which the account owner may deposit or withdraw money. However, to a taxing authority, the account may play the role of an entity whose dividends must be reported on annually.

Example 1.7

To a trader, a share of stock represents an entity with value that may be bought or sold; to a lawyer, the same share denotes a legal instrument to which are attached certain rights.

Example 1.8

In the course of one day, the same person may play the role of mother, doctor, gardener, and movie critic.

The roles played by objects are dynamic yet can be mutually exclusive. In the case of the share of stock, its roles overlap slightly, but each role is static relative to the client that interacts with the share. In the case of the person, her roles are quite dynamic and may change from moment to moment.

1.3 Relationships among Objects

An object by itself is intensely uninteresting. Objects contribute to the behavior of a system by collaborating with one another. "Instead of a bit-grinding processor raping and plundering data structures, we have a universe of well-behaved objects that courteously ask each other to carry out their various desires".

For example, consider the object structure of an airplane, which has been defined as "a collection of parts having an inherent tendency to fall to earth, and requiring constant effort and supervision to stave off that outcome". Only the collaborative efforts of all the component objects of an airplane enable it to fly.

The relationship between any two objects encompasses the assumptions that each makes about the other, including what operations can be performed and what behavior results.

We have found that two kinds of object relationships are of particular interest in object-oriented analysis and design, namely:

1. Links
2. Aggregation

1.3.1. Links

Link is defined as a physical or conceptual connection between objects. An object collaborates with other objects through its links to these objects. Stated another way, a link denotes the specific association through which one object (the client) applies the services of another object (the supplier), or through which one object may navigate to another.

As a participant in a link, an object may play one of three roles.

1. Controller: This object can operate on other objects but is not operated on by other objects. In some contexts, the terms active object and controller are interchangeable.

2. Server: This object doesn't operate on other objects; it is only operated on by other objects.

3. Proxy: This object can both operate on other objects and be operated on by other objects. A proxy is usually created to represent a real-world object in the domain of the application.

1.3.1.1. Synchronization

Whenever one object passes a message to another across a link, the two objects are said to be synchronized. For objects in a completely sequential application this synchronization is usually accomplished by simple method invocation. However, in the presence of multiple threads of control, objects require more sophisticated message passing in order to deal with the problems of mutual exclusion that can occur in concurrent systems. As we described earlier, active objects embody their own thread of control, so we expect their semantics to be guaranteed in the presence of other active objects. However, when one active object has a link to a passive one, we must choose one of three approaches to synchronization.

1.3.2. Aggregation

Whereas links denote peer-to-peer or client/supplier relationships, aggregation denotes a whole/part hierarchy, with the ability to navigate from the whole (also called the aggregate) to its parts.

Aggregation may or may not denote physical containment.

For example, an airplane is composed of wings, engines, landing gear, and so on: This is a case of physical containment.

On the other hand, the relationship between a shareholder and his or her shares is an aggregation relationship that does not require physical containment. The shareholder uniquely owns shares, but the shares are by no means a physical part of the shareholder.

Rather, this whole/part relationship is more conceptual and therefore less direct than the physical aggregation of the parts that form an airplane.

There are clear trade-offs between links and aggregation. Aggregation is sometimes better because it encapsulates parts as secrets of the whole. Links are sometimes better because they permit looser coupling among objects. Intelligent engineering decisions require careful weighing of these two factors.

1.4. The Nature of a Class

The concepts of a class and an object are tightly interwoven, for we cannot talk about an object without regard for its class. However, there are important differences between these two terms.

1.4.1. What Is and What Isn't a Class

Whereas an object is a concrete entity that exists in time and space, a class represents only an abstraction, the "essence" of an object, as it were. Thus, we may speak of the class Mammal, which represents the characteristics common to all mammals. To identify a particular mammal in this class, we must speak of "this mammal" or "that mammal."

In everyday terms, *Webster's Third New International Dictionary* defines a class as "a group, set, or kind marked by common attributes or a common attribute; a group division, distinction, or rating based on quality, degree of competence, or condition.

In the context of object-oriented analysis and design, we define a class as follows:

A class is a set of objects that share a common structure, common behavior, and common semantics. A single object is simply an instance of a class.

Figure 1.8
A class represents a set of objects that share a common structure and a common behavior

 **A class represents a set of objects that share a common structure and a common behavior.**

What isn't a class?
An object is not a class.

 **Objects that share no common structure and behavior cannot be grouped in a class because, by definition, they are unrelated except by their general nature as objects.**

1.5. Relationships among Classes

Consider for a moment the similarities and differences among the following classes of objects: flowers, daisies, red roses, yellow roses, petals, and ladybugs.

We can make the following observations.

- A daisy is a kind of flower.
- A rose is a (different) kind of flower.
- Red roses and yellow roses are both kinds of roses.
- A petal is a part of both kinds of flowers.
- Ladybugs eat certain pests such as aphids, which may be infesting certain kinds of flowers.

From this simple example we conclude that classes, like objects, do not exist in isolation. Rather, for a particular problem domain, the key abstractions are usually related in a variety of interesting ways, forming the class structure of our design.

1.5.1. Association

Of these different kinds of class relationships, associations are the most general but also the most semantically weak. The identification of associations among classes is often an activity of analysis and early design, at which time we begin to discover the general dependencies among our abstractions. As we continue our design and implementation, we will often refine these weak associations by turning them into one of the other more concrete class relationships

1.5.1.1. Semantic Dependencies

As Example 1.9 suggests, an association only denotes a semantic dependency and does not state the direction of this dependency (unless otherwise stated, an association implies bidirectional navigation, as in our example), nor does it state the exact way in which one class relates to another (we can only imply these semantics by naming the role each class plays in relationship with the other).

However, these semantics are sufficient during the analysis of a problem, at which time we need only to identify such dependencies. Through the creation of associations, we come to capture the participants in a semantic relationship, their roles, and their cardinality.

Example 1.9
For a vehicle, two of our key abstractions include the vehicle and wheels. As shown in
Figure 1.8, we may show a simple association between these two classes: the class Wheel and the class Vehicle. (Arguably, an aggregation would be better.) By implication, this association suggests bidirectional navigation.

Given an instance of Wheel, we should be able to locate the object denoting its Vehicle, and given an instance of Vehicle, we should be able to locate all the wheels.

Figure 1.8
Association

1.5.1.2. Multiplicity

Our example introduced a one-to-many association, meaning that for each instance of the class Vehicle, there are zero (a boat, which is a vehicle, has no wheels) or more instances of the class Wheel, and for each Wheel, there is exactly one Vehicle. This denotes the multiplicity of the association.

In practice, there are three common kinds of multiplicity across an association:
1. One-to-one
2. One-to-many
3. Many-to-many

A one-to-one relationship denotes a very narrow association. For example, in retail telemarketing operations, we would find a one-to-one relationship between the class Sale and the class
Credit-Card-Transaction: Each sale has exactly one corresponding credit card transaction, and each such transaction corresponds to one sale. Many-to-many relationships are also common. For example, each instance of the class Customer might initiate a transaction with several instances of the class Sales-Person, and each such salesperson might interact with many different customers.

1.5.2. Inheritance
Inheritance, perhaps the most semantically interesting of these concrete relationships, exists to express generalization/specialization relationships. In our experience, however, inheritance is an insufficient means of expressing all of the rich relationships that may exist among the key

abstractions in a given problem domain. An alternate approach to inheritance involves a language mechanism called *delegation*, in which objects delegate their behavior to related objects.

A subclass may inherit the structure and behavior of its super class
A slightly better way to capture our decisions would be to declare one class for each kind of telemetry data. In this manner, we could hide the representation of each class and associate its behavior with its data. Still, this approach does not address the problem of redundancy.
A far better solution, therefore, is to capture our decisions by building a hierarchy of classes, in which specialized classes inherit the structure and behavior defined by more generalized classes

Figure 1.9
A subclass may inherit the structure and behavior of its super class

Summary

- The maturation of software engineering has led to the development of object-oriented analysis, design.

- There are several different paradigms: data-oriented, procedure-oriented, and object-oriented.
- Comparison of different object oriented paradigms to procedure in software engineering
- Understanding different elements of Object Oriented concepts.
- Understanding the history or evolution of Object Oriented Technology.

Assignments:

1. Explain the need of Object Oriented?
2. Identify how object oriented approach is useful for developing new generation software?

References:

[1]:http://uml-tutorials.trireme.com/

[2]: http://dl.acm.org/citation.cfm?id=226068

[3]: http://www.jvoegele.com/software/langcomp.html

[4]: http://dl.dropboxusercontent.com/u/31779972/The%20New%20Software%20Engineering.pdf

[5]: http://ce.sharif.edu/courses/85-86/2/ce924/resources/root/Presentations/2.%20OOAD.pdf

[6]: Dennis A, Wixom B, Tegarden D. "Systems Analysis and Design with UML" , 2nd Edition, John Wiley & Sons

[7]: Fowler M, "UML Distilled", 2nd Edition, PHI/Pearson Education, 2002.

[8]: Schach Stephen R., "Introduction to Object Oriented Analysis and Design" 2nd Edition, Tata McGraw-Hill, 2003.