UNIT-2

Object Model

Structured design methods evolved to guide developers who use procedural language and algorithms as their fundamental building blocks to build complex system.

Similarly, object-oriented design methods have evolved to help developers who exploit the expressive power of object-based and object-oriented programming languages, using the class and object as basic building blocks.

The object model is applicable not just to the programming languages, but also to the design of user interfaces, databases, and even computer architectures. The reason for this widespread application is simply that an object orientation helps us to cope with the complexity.

Unfortunately, most programmers today are formally and informally trained only in the principles of structured design. Many good engineers have developed useful software systems using these techniques. However, there are limits to the amount of complexity we can handle using only algorithmic decomposition; thus we must turn to object-oriented decomposition.

Furthermore, if we try to use languages such as C++ and Ada as if they were only traditional, algorithmically oriented languages, we not only miss the power available to us, but we usually end up worse off than if we had used an older language such as C or Pascal.

*****OOP, OOD, and OOA**

Because the object model derives from so many disparate sources, it includes lot of confusion in terminology. The same concept is referred by different terms by different languages.

To minimize the confusion, let's define what is object-oriented and what is not.

What we can agree upon is that the concept of an object is central to anything object-oriented. Informally, an object is a tangible entity that exhibits some well-defined behavior.

Objects are "entities that combine the properties of procedures and data since they perform computations and save local state". Objects serve (help) to unify the ideas of algorithmic and data abstraction.

In the object model, emphasis is placed on crisply characterizing the components of the physical or abstract system to be modeled by a programmed system.

Objects have a certain 'integrity' which should not (in fact, cannot) be violated.

An object can only change state, behave, be manipulated, or stand in relation to other objects in ways appropriate to that object. Stated differently, there exist invariant properties that characterize an object and its behavior.

An elevator, for example, is characterized by invariant properties including [that] it only travels up and down inside its shaft. Any elevator simulation must incorporate these invariants [, for they are integral to the notion of an elevator].

UNIT-II: Object Oriented Concepts

Object Oriented Programming – Object Oriented Design – Object Oriented Analysis – Elements of the Object Model - The Nature of an Object – Relationships among Objects

2.1 Object-Oriented Programming

Object-Oriented Programming (OOP) Definition

Object-oriented programming is 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.

There are three important parts to this definition: object-oriented programming (1) uses *objects*, not algorithms, as its fundamental logical building blocks (the “part of” hierarchy); (2) each object is an instance of some class; and (3) classes are related to one another via inheritance relationships (the "is a" hierarchy)

If any of these elements is missing, it is not an object-oriented program. Specifically, programming without inheritance is distinctly (definitely) not object-oriented; we call it *programming with abstract data types*. By this definition, some languages are object-oriented, and some are not.

Object-Oriented Language is one that has mechanisms which support the object-oriented style of programming well. That is, it should provide facilities that make it convenient to use OO style. No exceptional effort or skill to write such programs should be required.

From a theoretical perspective, one can do object oriented programming in non-object oriented programming languages like Pascal and even COBOL or assembly language, but it is badly awkward to do so.

A language is object-oriented if and only if it satisfies the following requirements:

- It supports objects that are data abstractions with an interface of named operations and a hidden local state.
- Objects have an associated type [class].
- Types [classes] may inherit attributes from super types [super classes]

For a language to support inheritance means that it is possible to express "is a" relationships among types, for example, a red rose is a kind of flower, and a flower is a kind of plant.

If a language does not provide direct support for inheritance, then it is not object-oriented. We distinguish such languages by calling them ***object-based*** rather than *object-oriented*.

Under this definition, Smalltalk, Object Pascal, C++, Eiffel, and CLOS are all object-oriented, and Ada is object-based.

However, since objects and classes are elements of both kinds of languages, it is both possible and highly desirable for us to use object-oriented design methods for both object-based and object-oriented programming languages.

Object-Oriented Design

The emphasis in programming methods is primarily on the proper and effective use of particular language mechanisms. By contrast, design methods emphasize the proper and effective structuring of a complex system.

Object-Oriented Design Definition

Object-oriented design is 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.

There are two important parts to this definition: object-oriented design (1) leads to an object oriented decomposition and (2) uses different notations to express different models of the logical (class and object structure) and physical (module and process architecture) design of a system, in addition to the static and dynamic aspects of the system.

The object-oriented design uses object-oriented decomposition (uses class and object abstractions to logically structure systems) whereas structured design uses algorithmic abstractions.

The term *object oriented design* to refer to any method that leads to an object-oriented decomposition.

Object-Oriented Analysis

The object model has influenced even earlier phases of the software development life cycle. Traditional structured analysis techniques focus upon the flow of data within a system.

Object-oriented analysis (OOA) emphasizes the building of real-world models, using an object-oriented view of the world.

Object-Oriented Analysis Definition

Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain.

How are OOA, OOD, and OOP related? Basically, the products of object oriented analysis serve as the models from which we may start an object-oriented design; the products of object-oriented design can then be used as blueprints for completely implementing a system using object-oriented programming methods.

That is, output of the OOA forms an input for the OOD, and output of the OOD forms an input for the OOP.

2.2 Elements of the Object Model

Most programmers work in one language and use only one programming style. They program in a pattern enforced by the language they use. Frequently, they have not been exposed to alternate ways of thinking about a problem, and hence have difficulty in seeing the advantage of choosing a style more appropriate to the problem at hand

A **programming style** is a way of organizing programs on the basis of some conceptual model of programming and an appropriate language to make programs written in the style clear.

There are five main kinds of programming styles. They are shown below along with the kinds of abstractions they employ:

Programming Style	**Abstractions**
Procedure-oriented	Algorithms
Object-oriented	Classes and objects

Logic-oriented	Goals, often expressed in a predicate calculus
Rule-oriented	If-then rules
Constraint-oriented	Invariant relationships

There is no single programming style that is best for all kinds of applications. For example, rule-oriented programming would be best for the design of a knowledge base, and procedure-oriented programming would be best suited for the design of computation-intense operations.

The object-oriented style is best suited to the broadest set of applications. It often serves as the architectural framework in which we employ other paradigms.

Each of these styles of programming is based upon its own conceptual framework. Each requires a different mindset, a different way of thinking about the problem.

For all things object-oriented, the conceptual framework is the **Object Model**.

There are four major elements of this model (**Object 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
2. Concurrency
3. Persistence

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

Without this conceptual framework, you may be programming in OO language such as C++, but your design is going to smell like a procedural language such as C application.

You will have missed out on or otherwise abused the expressive power of the object-oriented language. More importantly it indicates that you are not likely to have mastered the complexity of the problem at hand.

(1) Abstraction

The Meaning of Abstraction

Abstraction is one of the fundamental ways to cope with complexity.

The **Abstraction** arises from a recognition of similarities between certain objects, situations, or processes in the real world, and the decision to concentrate upon these similarities and to ignore for the time being the differences.

The **Abstraction** is a simplified description, or specification, of a system that emphasizes some of the system's details or properties while suppressing others. A good abstraction is one that emphasizes details that are significant to the user and suppresses details that are immaterial.

A concept qualifies as an **Abstraction** only if it can be described, understood, and analyzed independently of the mechanism that will eventually be used to realize it

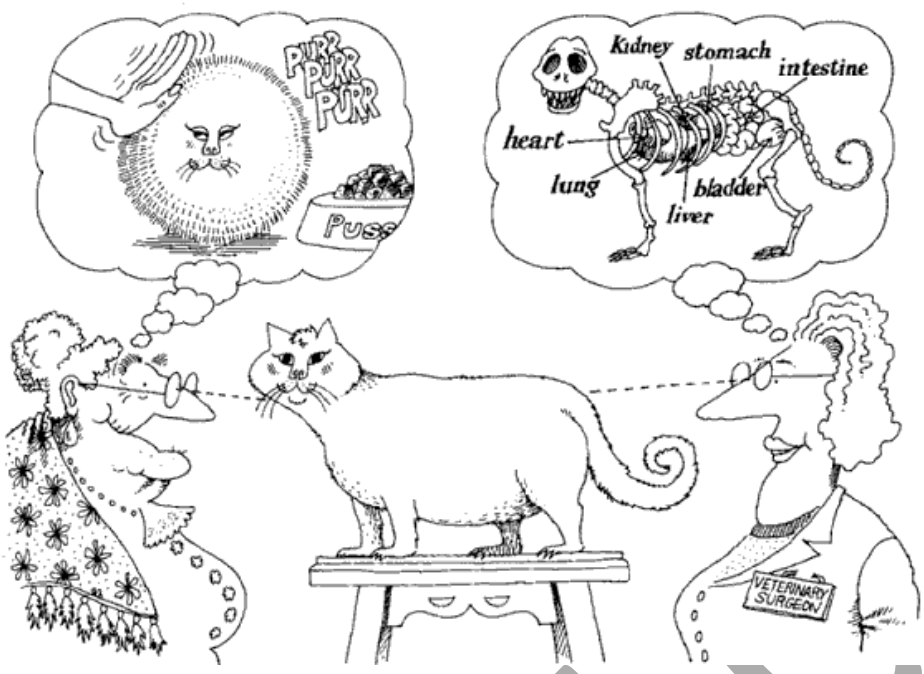
Combining these different viewpoints, an **abstraction** can be defined as follows:

An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.

An abstraction focuses on the outside view (behavior) of an object, and so serves to separate an object's essential behavior from its implementation (inside view).

This behavior/implementation division is called an ***abstraction barrier***. It is achieved by applying the **principle of least commitment**, through which the interface of an object provides its essential behavior, and nothing more.

We like to use an additional principle that we call the ***principle of least astonishment (surprise)***, through which an abstraction captures the entire behavior of some object, no more and no less, and offers no surprises or side effects that go beyond the scope of the abstraction.



Abstraction focuses upon the essential characteristics of some object, relative to the perspective of the viewer.

Deciding upon the right set of abstractions for a given domain is the central problem in object-oriented design.

There is a spectrum of abstraction, from objects which closely model problem domain entities to objects which really have no reason for existence

From the most to the least useful, these kinds of abstractions include the following:

Entity Abstraction	An object that represents a useful model of a problem domain or solution-domain entity It is Most Useful type of abstraction. We should strive to build Entity Abstractions, as it closely models the problem domain
Action Abstraction	An object that provides a generalized set of operations, all of which perform the same kind of function
Virtual Machine Abstraction	An object that groups together operations that are all used by some superior level of control, or operations that all use some junior-level set of operations
Coincidental	An object that packages a set of operations that have no relation

Abstraction	to each other It is Least useful (Not useful).

A *client* is any object that uses the resources of another object (known as the *server*). The Client Object uses resources of Server Object.

The behavior of an object is characterized by the services that it provides to other objects, as well as the operations that it may perform upon other objects.

***Contract Model* of programming**

The outside view of each object defines a contract upon which other objects may depend, and which in turn must be carried out by the inside view of the object itself (often in collaboration with other objects).

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.

Individually, each operation that contributes to this contract has a unique signature comprising all of its formal arguments and return type.

We call the entire set of operations that a client may perform upon an object, together with the legal orderings in which they may be invoked, its ***protocol***.

A protocol denotes the ways in which an object may act and react, and thus constitutes the entire static and dynamic outside view of the abstraction.

Central to the idea of an abstraction is the concept of invariance.

An ***invariant*** is some Boolean (true or false) condition whose truth must be preserved.

For each operation associated with an object, we may define ***preconditions*** (invariants assumed by the operation) as well as ***post conditions*** (invariants satisfied by the operation).

Violating an *invariant* breaks the *contract* associated with an abstraction.

If a *precondition* is violated, this means that a Client has not satisfied its part (of the bargain), and hence the server cannot proceed reliably.

Similarly, if a *post-condition* is violated, this means that a server has not carried out its part (of the contract), and so its clients can no longer trust the behavior of the server.

An **exception** is an indication that some *invariant* has not been or cannot be satisfied.

Certain languages permit objects to throw exceptions so as to abandon processing and alert some other object to (about) the problem, who in turn may catch the exception and handle the problem.

The terms ***operation, method*, and *member function*** evolved from three different programming language (Ada, Smalltalk, and C++, respectively). They all mean virtually the same thing, and can be used interchangeably.

All abstractions have static as well as dynamic properties. For example, a file object has properties such as “amount of space”, “Name”, and “Content”.

These are all static properties. The value of each of these properties is dynamic, relative to the lifetime of the object: a file object may grow or shrink in size, its name may change, or its contents may change.

Following example shows static (whose value remains same throughout life span of object) and dynamic (whose value changes throughout life span of object) properties of Car object and Person object.

- Car
  - Static Property: Registration No., Company Name
  - Dynamic Property: Current Speed, Owner Name
- Person
  - Name, Date of Birth, SSN (Static Property)
  - Cell Phone, Address, Hobby, Salary (Dynamic Property)

[In a procedure-oriented style of programming, things happen when subprograms are called and statements are executed.

In a rule-oriented style of programming, things happen when new events cause rules to fire, which in turn may trigger other rules, and so on.]

In an object-oriented style of programming, things happen whenever we operate upon an object (i.e., when we *send a message* to an object).

Thus, invoking an operation upon an object elicits (causes) some reaction from the object.

What operations we can meaningfully perform upon an object and how that object reacts constitute the entire **behavior of the object**.

Examples of Abstraction

Maintaining the proper greenhouse environment depends upon the kind of plant being grown and its age. One must control diverse factors such as temperature, humidity, light, pH, and nutrient concentrations.

On a large farm, it is not unusual to have an automated system that constantly monitors and adjusts these elements.

Simply stated, the purpose of an automated gardener is to efficiently carry out, with minimal human intervention, growing plans for the healthy production of multiple crops.

One of the key abstractions in this problem is that of a sensor.

Actually, there are several different kinds of sensors. Anything that affects production must be measured, and so we must have sensors for air and water temperature, humidity, light, pH, and nutrient concentrations, among other things.

Viewed from the outside, a **temperature sensor** is simply an object that knows how to measure the **temperature** at some specific **location**.

What is a temperature? It is some numeric value, within a limited range of values and with a certain precision that represents degrees in the scale of Fahrenheit, Centigrade, or Kelvin.

What then is a location? It is some identifiable place on the farm at which we desire to measure the temperature.

What is important for a temperature sensor is not so much where it is located, but the fact that it has a unique location and identity from all other temperature sensors.

What are the responsibilities of a temperature sensor? A sensor is responsible for knowing the temperature at a given location, and reporting that temperature when asked.

More concretely, what operations can a client perform upon a temperature sensor? A client can calibrate (adjust) it, as well as ask what the current temperature is.

We can represent the abstraction of Temperature Sensor object as under.

Object Name	Temperature Sensor
Object Properties	Temperature Location
Object Responsibilities	Calibrate() OR setTemp(Temperature t) CurrentTemperature() OR getCurrentTemp()

In C++, this abstraction can be represented as below.

```
//Temperature in degrees Fahrenheit.
typedef float Temperature;

// Number uniquely denoting the location of a sensor
typedef unsigned int Location;

class TemperatureSensor
{
      Public:
            TemperatureSensor(Location);
            ~TemperatureSensor();

            void calibrate(Temperature actualTemperature);
            Temperature currentTemperature() const;
      private:
            …
            //the representation (implementation) is hidden in the private part of the class
};
```

The two typedefs, **Temperature** and **Location**, provide convenient aliases for more primitive types, thus letting us express our abstractions in the vocabulary of the problem domain.

TemperatureSensor is defined as a class. We must first create an *instance (object) [*so that we have something upon which to operate].

```
Temperature t;

TemperatureSensor TS_One(10);    // Creates one object
TemperatureSensor TS_Two(20);    // Creates another object

t = TS_One.currentTemperature(); // requests to get current temperature
                                 // of location number 10
```

Consider the invariants associated with the operation **currentTemperature()**.

Its preconditions include the assumption that the sensor is associate with a valid location, and its post-conditions include the assumption that the value returned is in degrees Fahrenheit.

Example of Active Abstraction

The abstraction we have described thus far is passive (Passive Abstraction).

Some client object must operate upon temperature sensor object to determine its current temperature.

There is another legitimate abstraction. Rather than the temperature sensor being passive, we might make it active. So that it is not acted upon but rather acts upon other objects whenever the temperature at its location changes a certain number of degrees from a given set point.

The responsibilities have changed slightly: a sensor is now responsible for reporting the current temperature when some condition occurs (not just when asked).

A common programming style used in such circumstances is the **callback**, in which a client provides a function to the server (the callback function), and the server calls the client's function whenever the appropriate conditions are met.

Thus, we might write the following:

```
class ActiveTemperatureSensor
{
      public:
            ActiveTemperatureSensor(Location, void (*f)(Location,
            Temperature));
            ~ActiveTemperatureSensor();

            void calibrate(Temperature actualTemperature);
            void establishSetpoint(Temperature setpoint, Temperature
            delta);

            Temperature currentTemperature() const;
      private:
      …
};
```

Whenever we create a sensor object, we must provide its location and we must also provide a callback function.

Additionally, a client of this abstraction may invoke the operation **establishSetpoint** to establish a critical range of temperatures.

It is then the responsibility of the **ActiveTemperatureSensor** object to invoke the given callback function whenever the temperature at its location drops below or rises above the given setpoint.

When the callback is invoked, the sensor provides its location and the current temperature, so that the client has sufficient information to respond to the condition.

Notice that a client can still inquire as to the current temperature of a sensor at any time.

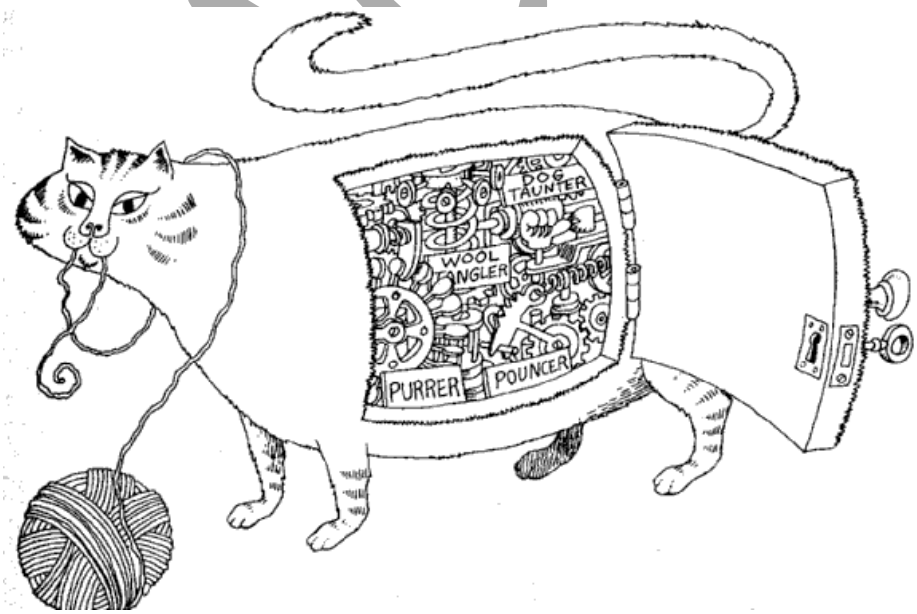
What if a client never establishes a setpoint? Our abstraction must make some reasonable assumption: one design decision might be to initially assume an infinite range of critical temperatures, and so the callback would never be invoked until some client finally established a setpoint.

How the **ActiveTemperatureSensor** class carries out its responsibilities is a function of its inside view, and is of no concern to outside clients. It is the secrets of the class, which are implemented by the private parts of the class.

An Object may collaborate with other objects to achieve some behavior. How the objects cooperate with one another (is a design decision that) define the boundaries of each abstraction and thus the responsibilities and protocols of each object.

In declaration of class, public part can include constructor & destructor, member functions (used to create and destroy object), modifiers (used to change the state of the object by altering value of some property) and selectors (used to reads the value of some property).

(2) Encapsulation



Encapsulation hides the details of the implementation of an object.

The Meaning of Encapsulation

Whichever the representation (method of implementation of class in its private part) is chosen is immaterial to the client's contract with this class, as long as that representation upholds the contract.

Simply stated, the abstraction of an object should precede the decisions about its implementation.

Once an implementation is selected, it should be treated as a secret of the abstraction and hidden from most clients.

It is suggested that “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”.

Abstraction and encapsulation are complementary concepts: abstraction focuses upon the observable behavior of an object, whereas *encapsulation* focuses upon the implementation that gives rise to this behavior.

Encapsulation is most often achieved through ***information biding***, 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. The interfaces of some of the methods (private member functions) which are used internally by class also may be hidden.

Encapsulation provides explicit barriers among different abstractions of the system and thus leads to a clear separation of concerns.

[For example, consider designing a database application, it is standard practice to write programs so that they don't care about the physical representation of data, but depend only upon a schema that denotes the data's logical view.]

The objects at one level of abstraction are shielded from implementation details at lower levels of abstraction.

For abstraction to work, implementations must be encapsulated. This means that each class must have two parts: an interface and an implementation.

The ***interface*** of a class captures only its outside view, encompassing our abstraction of the behavior common to all instances of the class. The ***implementation*** of a class comprises the representation of the abstraction as well as the mechanisms that achieve the desired behavior.

The interface of a class is the one place where we assert all of the assumptions that a client may make about any instances of the class; the implementation encapsulates details about which no client may make assumptions.

Definition of *encapsulation*

Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation.

These encapsulated elements are sometimes called the "secrets" of an abstraction.

Examples of Encapsulation

To illustrate the principle of encapsulation, consider hydroponics gardening system. One of the key abstraction in this problem domain is that of a heater. A heater is at a fairly low level of abstraction, and thus we might decide that there are only three meaningful operations that we can perform upon this object: turn it on, turn it off, and find out if it is running.

We do not make it a responsibility of this abstraction to maintain a fixed temperature. Instead, we choose to give this responsibility to *another object*, which must collaborate with a *temperature sensor* and a *heater* to achieve this higher-level behavior. We call this behavior higher-level because it builds upon the primitive semantics of temperature sensors and heaters and adds some new semantics, namely, hysteresis, which prevents the heater from being turned on and off too rapidly when the temperature is near boundary conditions.

By deciding upon this separation of responsibilities, we make each individual abstraction more cohesive.

We begin with another **typedef**:

```
// Boolean type
enum Boolean {FALSE, TRUE};
```

[Metaoperations, constructor and destructor operations, initializes and destroys instances of the class, respectively]

We might write the definition of the class *Heater* in C++ as follows:

```
Class Heater
{
        public:
                Heater(location);           //Constructor
                ~Heater();                  //Destructor

                void turnOn();
                void turnoff();

                Boolean isOn() const;
        private:
                ...
};
```

This interface represents all that a client needs to know about the class **Heater**.

Turning to the inside view of this class, we have an entirely different perspective. Suppose that our system engineers have decided to locate the computers that control each greenhouse away from the building, and to connect each computer to its sensors and actuators via serial lines. The implementation of this class may be highly complex and may involve/use the services of other objects/classes.

Intelligent encapsulation localizes (focuses/concentrates) design decisions that are likely to change. As a system evolves, its developers might discover that in (during) actual use, certain operations take longer than acceptable or that some objects consume more space than is available. In such situations, the representation of an object is often changed to apply more efficient algorithms, or to optimize for space.

This ability to change the representation of an abstraction without disturbing any of its clients is the essential benefit of encapsulation.

Ideally, attempts to access the underlying representation of an object should be detected as early as possible.

C++ offers flexible control over the Visibility of member objects and member functions. Specifically, members may be placed in the public, private, or protected parts of a class.

Members declared in the public parts are visible to all clients; members declared in the private parts are fully encapsulated (not visible outside); and members declared in the protected parts are visible only to the class itself and its subclasses.

C++ also supports the notion of *friends*: cooperative classes that are permitted to see each other's private parts.

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.

Thus, encapsulation cannot stop a developer from doing stupid things: "Hiding is for the prevention of accidents, not the prevention of fraud"

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.

In practice, there are times when one must study the implementation of a class to really understand its meaning, especially if the external documentation is lacking.

(3) Modularity

The Meaning of Modularity

It is observed that “the act of partitioning a program into individual components can reduce its complexity to some degree”.

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 (understanding) of the program.

In some languages, such as Smalltalk, there is no concept of a module, and so the class forms the only physical unit of decomposition. In many others, including Object Pascal, and C++, the module is a separate language construct, and therefore warrants a separate set of design decisions.

In these languages, classes and objects form the logical structure of a system; we place these abstractions in *modules* to produce the system's physical architecture. Especially for larger applications, in which we may have many hundreds of classes, the use of modules is essential to help manage complexity.

Modularization consists of dividing a program into modules which can be compiled separately, but which have connections with other modules. The connections between modules are the assumptions which the modules make about each other.

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.

As with encapsulation, particular languages support modularity in diverse ways. A module is also known as Unit, Package, or Assembly, DLL, etc. by different languages.

[For example, modules in C++ are nothing more than separately compiled files. Object Pascal is a little more formal about the matter. In this language, the syntax for *Units* (modules) distinguishes between module interface and implementation. Dependencies among units may be asserted only in a module's interface.

Ada goes one step further. A Package (module) has two parts: the package specification and the package body. Unlike Object Pascal, Ada allows connections among modules to be asserted separately in the specification and body of a package. Thus, it is possible for a package body to depend upon modules that are otherwise not visible to the package's specification.]

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

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

For tiny problems, the developer might decide to declare every class and object in the same package. A better solution is to group logically related classes and objects in the same module, and expose only those elements that other modules absolutely must see.

Arbitrary modularization is sometimes worse than no modularization at all.

In traditional structured design, modularization is primarily concerned with the meaningful grouping of subprograms, using the criteria of *coupling* and *cohesion*. In object-oriented design, the problem is subtly different: the task is to decide where to physically package the classes and objects from the design's logical structure.

There are several useful technical as well as nontechnical guidelines to achieve an intelligent modularization of classes and objects.

- The overall goal of the decomposition into modules is the reduction of software cost by allowing modules to be designed and revised independently
- Each module's structure should be simple enough that it can be understood fully (without going through the details of other modules)
- it should be possible to change the implementation of the modules without knowledge of the implementation of other modules and without affecting the behavior of other modules

As we have seen module has two parts: Specification (Interface/header) and Implementation (body). In practice, the cost of recompiling the body of a module is relatively small: only that unit need be recompiled and the application relinked. However, the cost of recompiling the interface of a module is relatively high. Especially with strongly typed languages, one must recompile the module interface, its body, and all other modules that depend upon this interface, the modules that depend upon these modules, and so on.

Thus, for very large programs, a change in a single module interface might result in many minutes if not hours of recompilation. For this reason, a module's interface should be as narrow as possible.

Hide as much as we can in the implementation of a module. Incrementally shifting declarations from a modules implementation to its interface is far less painful than ripping (splitting/breaking) out extraneous interface code.

The developer must therefore balance two competing technical concerns: the desire to encapsulate abstractions, and the need to make certain abstractions visible to other modules.

Following is the guidance

- System details that are likely to change independently should be the secrets of separate modules
- The only assumptions that should appear between modules are those that are considered unlikely to change.
- Every data structure is private to one module; it may be directly accessed by one or more programs within the module but not by programs outside the module. Any other program that requires information stored in a module's data Structures must obtain it by calling module programs"

In other words, strive to build modules that are highly cohesive (by grouping logically related abstractions) and loosely coupled (by minimizing the dependencies among modules).

From this perspective, we may define modularity as follows:

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 synergetic.

An object provides a crisp boundary around a single abstraction, and both encapsulation and modularity provide barriers around this abstraction.

Two additional technical issues can affect modularization decisions.

First, since modules usually serve as the elementary and indivisible units of software at can be reused across applications, a developer might choose to package classes and objects into modules in a way that makes their **reuse** convenient.

Second, many compilers generate object code in segments, one for each module. Therefore, there may be practical limits on the size of individual modules. With regard to the dynamics of subprogram calls, the placement of declarations within modules can greatly affect the **locality of reference** and thus, the paging behavior of a virtual memory system. Poor locality happens when subprogram calls occur across segments and lead to cache misses and page thrashing that ultimately slow down the whole system.

Several competing nontechnical needs may also affect modularization decisions. Typically, work assignments in a development team are given on a module-by-module basis, and so the boundaries of modules may be established to minimize the interfaces among different parts of the development organization.

On a larger scale, the same situation applies with subcontractor relationships. Abstractions may be packaged so as to quickly stabilize the module interfaces agreed upon among the various companies.

Module design decisions may be driven by documentation requirements/efforts sometimes.

Security may also be an issue: most code may be considered unclassified, but other code that might be classified secret or higher is best placed in separate modules.

Most important point is: Finding the right classes and objects and then organizing them into separate modules are largely independent design decisions.

The identification of classes and objects is part of the logical design of the system, but the identification of modules is part of the system's physical design.

One cannot make all the logical design decisions before making all the physical ones, or vice versa; rather, these design decisions happen iteratively.

Examples of Modularity

Let's look at modularity in the College Automation System. One of our key abstractions here is that of a Library Book Management, and we might therefore create a module whose purpose is to collect all of the classes associated with Library activities. The College Automation System may then import (use) this Library Module and may use it services.

Our design will probably include many other modules, each of which imports the interface of lower level units. Ultimately, we must define some main program from which we can invoke this application from the operating system.

In object-oriented design, defining this main program is often the least important decision, whereas in traditional structured design, the main program serves as the root, the keystone that holds everything else together.

The object-oriented view is more natural. It is observed that “Practical software systems are more appropriately described as offering a number of services. Defining these systems by single functions is usually possible, but yields rather artificial answers….Real systems have no top".

(4) Hierarchy

Encapsulation helps in managing complexity by hiding the inside-view of abstraction

- Module helps in clustering logically related abstractions
- This is not enough. A set of abstractions often forms a **hierarchy**
- Identifying these hierarchies greatly simplify our understanding of the problem.

Definition

- ***Hierarchy*** *is a ranking or ordering of abstractions.*

The most important hierarchies in a complex system are:

- its class structure (the "is a" hierarchy) and
- its object structure (the "part of' hierarchy).

- **Inheritance** is the most important "is a” hierarchy
- Inheritance defines a relationship among classes

- one class shares the structure or behavior defined in one or more classes (denoting ***single inheritance*** *and* ***multiple inheritance****,* respectively).
- Inheritance thus represents a hierarchy of abstractions, in which a sub-class inherits from one or more super-classes.
- Typically, a subclass augments or redefines the existing structure and behavior of its super-classes.

- Semantically, inheritance denotes an "is-a" relationship. For **example**
- Single Inheritance
  - House is a Property
  - Car is a Vehicle
  - Teaching Staff is a Staff
  - Multiple Inheritance
- Teaching, Non-Teaching, Trainee, Permanent
  - Trainee Teaching Staff is a Teaching Staff
  - Trainee Teaching Staff is a Trainee Staff

- Inheritance thus implies a generalization/ specialization hierarchy
  - wherein a subclass specializes the more general structure or behavior of its super-classes.
- Litmus test for inheritance
  - **if B "is not a" kind of A, then B should not inherit from A.**

```
class FruitGrowingPlan : public GrowingPlan {
        public:
                FruitGrowinjgPlan(char* name);
                virtual ~FruitGrowingPlan();
                virtual void establish(Day, Hour, Condition&);
                                                //Overridding (modified)
                void scheduleHarvest(Day, Hour);        //New Methods
                Boolean isHarvested() const;            //New Methods
                unsigned daysUntilHarvest() const;      //New Methods
                Yield estimatedYield() const;           //New Methods
        protected:
                Boolean repHarvested;                   //New Data Member
                Yield repYield;                         //New Data Member
};
```

- **Inheritance**
  - Supports Re-Use
  - Otherwise each class designer starts from scratch
  - Not only it is duplication of effort, it generates inconsistency in code also.

Inheritance VS. Abstraction

- **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

- For a given class, there are usually two kinds of clients:
  - objects that invoke operations upon instances of the class, and
  - subclasses that inherit from the class

- with inheritance, encapsulation can be violated in one of three ways:
  - The subclass might access an instance variable of its super-class,
  - Call a private operation of its super-class, or
  - Refer directly to super-classes of its super-class
- PL may allow to control Inheritance
  - E.g., in C++, one can use modifiers such as public, private, and protected

Example of Multiple Inheritance

```
class Plant {
        public:
                Plant (char* name, char* species);
                virtual ~Plant();
                void setDatePlanted(Day);
                virtual establishGrowingConditions(const Condition&);
                const char* name() const;
                const char* species() const;
                Day datePlanted() Const;

        protected:
                char* repName;
                char* repSpecies;
        Day repPlanted;

        private:
                …
};

class FlowerMixin {
        public:
                FlowerMixin(Day timeToFlower, Day timeToSeed);
                virtual ~FlowerMixin();
                Day timeToFlower() const;
                Day timeToSced() const;
        protected:
                …
```

```
};

class FruitVegetableMixin {
        public:
                FruitVegetablieMixin(Day timeToHarvest);
                virtual ~FruitVegetableMixin();
                Day timeToHarvesto const;
        protected:
                …
};
```

- class Rose : public Plant, public FlowerMixin…
- class Carrot : public Plant, public FruitVegetableMixin {};
- in both cases, we form the subclass by inheriting from two super-classes.
- Instances of the subclass Rose thus include the structure and behavior from the class **Plant** together with the structure and behavior from the class **FlowerMixin**

- Multiple inheritance introduces complexities for PL
- PL must address two issues
  - Clashes among names from different super-classes, and
  - repeated inheritance.
- Clashes will occur when two or more super-classes provide a field or operation with the same name or signature
  - In C++, such clashes must be resolved with explicit qualification
- Repeated inheritance occurs when two or more super-classes share a common super-class

- Example of Hierarchy (Aggregation)
- Whereas these "is a" hierarchies denote generalization/ specialization relationships, "part of" hierarchies describe aggregation relationships.
- For example, consider the following class:

```
class Garden {
        public:
                Garden();
                virtual ~Garden();
        protected:
                Plant* repPlants[100];
                GrowingPlan repPlan;
};
```

- In terms of its "is a" hierarchy, a high-level abstraction is generalized, and a low-level abstraction is specialized.

  - E.g., **Flower class** is at a higher level of abstraction then a **Plant class.**
- In terms of its "part of' hierarchy, a class is at a higher level of abstraction than any of the classes that make up its implementation
  - E.g., **Garden** is at a higher level of abstraction than the type **plant**

- **Aggregation** permits the physical grouping of logically related structures, and
- **Inheritance** allows these common groups to be easily reused among different abstractions.

- Aggregation raises the issue of ownership.
- Assume that the lifetime of a garden and its plants are independent
  - Implemented by including **pointers to Plant objects** rather than values.
- In contrast, we have decided that a GrowingPlan object is inherently associated with a Garden object, and does not exist independently of the garden.
  - For this reason, we use a **value of GrowingPlan.**

Summary

- Inheritance (IS A)
  - Single Inheritance
  - Multiple Inheritance
  - Object inherit other object(s) but both have independent existence
- Aggregation (Part of)
  - Existence of object depends on existence of parent object

(5) Typing

- A type is a precise characterization of structural or behavioral properties which a collection of entities all share

- we will use the terms type and class interchangeably
  - However, a type and a class are not exactly the same thing. It is sufficient to say that a class implements a type

- *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.

- Such enforcement is essential for programming-in-the-large

- The idea of conformance is central to the notion of typing

- Example of strong typing, wherein the rules of our domain prescribe and enforce certain legal combinations of abstractions
  - E.g., When we divide distance by time, we expect some value denoting speed, not weight

- A PL may be
  - strongly- typed
  - weakly typed
  - Untyped

- A PL may be
  - strongly- typed (e.g., Eiffel)
    - type conformance is strictly- enforced
    - E.g., operations cannot be called upon an object unless the exact signature of that operation is defined in the object's class or super-classes
    - violation of type conformance can be detected at the time of compilation

- A PL may be
  - Untyped (e.g., Smalltalk)
    - a client can send any message to any class (although a class may not know how respond to the message)
    - Violations of type conformance may not be known until execution, and usually manifest themselves as execution errors

- Languages such as C++ are hybrid: they have tendencies toward strong typing, but it is possible to ignore or suppress the typing rules.

- Consider the abstraction of the various kinds of storage tanks that might exist in a greenhouse.

- We are likely to have storage tanks for water as well as various nutrients; although one holds a liquid and the other a solid

- // Number denoting level from 0 to 100 percent
- typedef float Level;

```
class StorageTank {
public:

  StorageTank();
  virtual ~StorageTank();

  virtual void fill();
  virtual void startDraining();
  virtual void stopOraining();

  Boolean isEmpty() const;
  Level level() const;

protected:
…
};

 class WaterTank : public  StorageTank {
 public:

   WaterTank();
   ~WaterTank();

   virtual void fill();
   virtual void startDraining();
   virtual Void stopDraining();
   void startHeating();
   void stopHeating();

   Temperature currentTemperature() const;
 Protected:
 …
 };
```

```
class NutrientTank : public StorageTank {
public:

   NutrientTank();
   virtual ~NutrientTank();

   virtual void startDraining();
    virtual void stopDraining();

Protected:
…
};
```

- Suppose that we have the following declarations:
  - StorageTank s1, s2;
  - WaterTank w;
  - NutrientTank n;

- With regard to type checking among classes, C++ is more strongly typed, meaning that expressions that invoke operations are checked for type correctness at the time of compilation.

- For example, the following statements are legal:
  - Level 1 = sl.level();
  - w.startDraining();
  - n.stopDraining();

- However, the following statements are not legal and would be rejected at compilation time:
  - sl.startHeatinI(); Illegal
  - n.stopHeating(); Illegal
- Neither of these two statements is legal because the methods **startHeating** and **stopHeating** are not defined for the class of the corresponding variable, nor for any superclasses of its class.
- On the other hand, the following statement is legal:
  - n.fill();
- Though fill is not defined in the class **NutrientTank** it is defined in the superclass **StorageTank**, from which the class **NutrientTank** inherits its structure and behavior.

- However, there is a dark side to strong typing. Practically, strong typing introduces semantic dependencies such that even small changes in the interface of a base class require recompilation of all subclasses.

- A strongly typed language is one in which all expressions are guaranteed to be type-consistent.

- The following assignment statements are legal:
  - s1 = s2;
  - s1 = w;

- The first statement is legal because the class of the variable on the left side of the statement (**StorageTank**) is the same as the class of the expression on the right side.

- The second statement is also legal because the class of the variable on the left side (**StorageTank**) is a superclass of the variable on the right side (**WaterTank**).

- However, this assignment results in a loss of information (known in C++ as *slicing*).

- The subclass WaterTank introduces structure and behavior beyond that defined in the base class, and this information cannot be copied to an instance of the base class.

- Consider the following illegal statements:
  - w = s1; // Illegal
  - w = n; // Illegal

- The first statement is not legal because the class of the variable on the left side of the assignment statement (**WaterTank**) is a subclass of the class of the variable on the right side (**StorageTank**).

- The second statement is illegal because the classes of the two variables are peers, and are not along the same line of inheritance (although they have a common superclass).

- In some situations, it is necessary to convert a value from one type to another

- In general, type conversion is to be avoided, because it often represents a violation of abstraction

- important benefits to be derived from using strongly typed languages:
  - "Without type checking, a program in most languages can 'crash' in mysterious ways at runtime.
  - In most systems, the edit-compile-debug cycle is so tedious that early error detection is indispensable.

  - Type declarations help to document programs.
  - Most compilers can generate more efficient object code if types are declared"

- Untyped languages offer greater flexibility, but even with untyped languages it is observed that
  - "In almost all cases, the programmer in fact knows what sorts of objects are expected as the arguments of a message, and what sort of object will be returned"

- In practice, the safety offered by strongly typed languages usually more then compensates for the flexibility lost by not using an un-typed language, especially for programming-in-the large.

- **Examples of Typing: Static and Dynamic Binding** The concepts of strong typing and static typing are entirely different.

- Strong typing refers to type consistency, whereas static typing - also known as *static binding* or *early binding* - refers to the time when names are bound to types. Static binding means that the types all variables and expressions are fixed at the time of compilation; *dynamic binding* (also called late *binding*) means that the types of all variables and expressions are not known until runtime

- Because strong typing and binding independent concepts, a language may be both strongly and statically typed strongly typed yet support dynamic binding (Object Pascal and C++), or untyped yet support dynamic binding (Smalltalk)

- if (s1.level() > s2.level())
  - s2.fill();
- What are the semantics of invoking the selector **level**? This operation is declared only in the base **StorageTank**, and therefore, no matter what specific class or subclass instance we provide for the formal argument **s1**, the base class operation will be invoked. Here, the call to level is statically bound: at the time of compilation, we know exactly what operation will be invoked.

- On the other hand, consider the semantics of invoking the modifier **fill**, which is dynamically bound.
- This operation is declared in the base class and then redefined only in the subclass **WaterTank**.
- If the actual argument to **s1** is a **WaterTank** instance, then **WaterTank::fill** will be invoked;

- if the actual argument to **s1** is a **NutrientTank** instance, then **StorageTank::fill** will be invoked

- This feature is called *polymorphism*; it represents a concept in type theory in which a single name (such as a variable declaration) may denote objects of many different classes that are related by some common superclass. Any object denoted by this name is therefore able to respond to some common set of operations

- Polymorphism exists when the features of inheritance and dynamic binding interact.

- It is perhaps the most powerful feature of object-oriented programming languages next to their support for abstraction, and it is what distinguishes object-oriented programming from more traditional programming with abstract data types.

- polymorphism is also a central concept in object-oriented design.

(6) Concurrency

- For certain kinds of problems, an automated system may have to handle many different events simultaneously. Other problems may involve so much computation that they exceed the capacity of any single processor.
- In each of these cases, it is natural to consider using a distributed set of computers for the target implementation or to use processors capable of multitasking.
- A single process - also known as a ***thread of control*** is the root from which independent dynamic action occurs within a system.
- Every program has at least one thread of control, but a system involving concurrency may have many such threads:
    - some that are transitory, and others that last the entire lifetime of the system's execution.
- Systems executing across multiple CPUs allow for truly concurrent threads of control, whereas systems running on a single CPU can only achieve the illusion of concurrent threads of control, usually by means of some time-slicing algorithm.
- **heavyweight and lightweight concurrency**.
- A *heavyweight process* is one that is typically independently managed by the target operating system, and so encompasses its own address space.

- A *lightweight process* usually lives within a single operating system process along with other lightweight processes, which share the same address space.
- Communication among heavyweight processes is generally expensive, involving some form of inter-process communication (IPC)
- Communication among lightweight processes is less expensive, and often involves shared data.
- Designing system that encompasses multiple threads of control is harder because of several problems related to concurrent programming (such as Deadlock and Starvation).
- At the highest levels of abstraction, OOP can alleviate (lighten/improve) the concurrency problem for the majority of programmers by hiding the concurrency inside reusable abstractions
- an object model is appropriate for a distributed system because it implicitly defines
    - (1) the units of distribution and movement and
    - (2) the entities that communicate
- Whereas object-oriented programming focuses upon **data abstraction**, encapsulation, and inheritance, concurrency focuses upon **process abstraction** and synchronization
- The object is a concept that unifies these two different viewpoints: each object (drawn from an abstraction of the real world) may represent a separate thread of control (a process abstraction).
- Such objects are called *active*. In a system based on an object-oriented design, we can conceptualize the world as consisting of a set of cooperative objects, some of which are active and thus serve as centers of independent activity.
- ***Concurrency***
    - *Concurrency is the properly that distinguishes an active object from one that is not active*
- Concurrency may be implemented with the support of one or more of the following
    - OS
    - PL
    - Third Party Library

- One must consider how active objects synchronize their activities with one another as well as with objects that are purely sequential.
- For example, if two active objects try to send messages to a third object, we must be certain to use some means of mutual exclusion, so that the state of the object being acted upon is not corrupted when both active objects try to update its state simultaneously.
- In the presence of concurrency, it is not enough simply to define the methods of an object; we must also make certain that the semantics of these methods are preserved in the presence of multiple threads of control.

(7) Persistence

- An object in software takes up some amount of space and exists for a particular amount of time
- The spectrum of object persistence encompasses the following:
    - Transient results in expression evaluation
    - Local variables in procedure activations
    - Own variables [as in ALGOL 60], global variables, and heap items whose extent is different from their scope
    - Data that exists between executions of a program
    - Data that exists between various versions of a program
    - Data that outlives the program
- Traditional programming languages usually address only the first three kinds of object persistence
- persistence of the last three kinds is typically the domain of database technology.
- Unifying the concepts of concurrency and objects gives rise to concurrent object-oriented programming languages.
- In a similar fashion, introducing the concept of persistence to the object model gives rise to object-oriented databases (OODBs)
- OODBs offer to the programmer the abstraction of an object-oriented interface, through which database queries and other operations are completed in terms of objects
- This simplifies the development
- In particular, it allows us to apply the same design methods to the database and non-database segments of an application

- Very few object-oriented programming languages provide direct support for persistence;
- Smalltalk is one notable exception, wherein there are protocols for streaming objects to and from disk
- However, streaming objects to flat files is a naive solution to persistence that does not scale well.
- More commonly, persistence is achieved through a modest number of commercially available object-oriented databases or object-relation databases.
- Some data elements stored on disk may be manipulated (created and updated) by different PL classes.
- Maintaining the integrity is challenging
    - Each class must also transcend (exceed) across individual programs
- In most systems, an object, once created, consumes the same physical memory until it ceases to exist.
- However, for systems that execute upon a distributed set of processors, we must sometimes be concerned with persistence across space.
- In such systems, it is useful to think of objects that can move from machine to machine, and that may even have different representations on different machines
- ***Summary Definition***
- *Persistence is the property of an object through which its existence transcends time (i.e. the object continues to exist after its creator ceases to exist) and/or space (i. e. the objects location moves from the address space in which it was created).*

Benefits of Object Model

- helps us to exploit the expressive power of object-based and object-oriented PL
- encourages the reuse not only of software but of entire designs (OO systems are often smaller)
- more resilient to change (durable)
- reduces the risks inherent in developing
- complex systems