Object Model

OOAD

Foundations of the Object Model

- Structured Design Methods evolved
  - For Developer (who build complex systems)
  - Using Procedural PL
  - Using algorithms as their fundamental building blocks (Algorithmic Abstraction)

- Object-Oriented Design Methods evolved
  - For Developers (who build complex systems)
  - Using object-based and object-oriented PL
  - using the class and object as basic building blocks.

Object Model

- 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 that
  - an object orientation helps to cope with the complexity (inherent in many different kinds of systems).

Object Model

- Unfortunately, most programmers are trained only in the principles of Structured Design (SD)
  - Many good engineers have developed useful software using SD
  - However, there are limits to the amount of complexity we can handle using only algorithmic decomposition
  - Thus we must turn to object-oriented decomposition.

Object Model

- if we try to use object based 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

- As Object Model is derived from so many disparate sources, it includes lot of confusion in terminology

- Some unified concepts
  - the concept of an object is central to anything object-oriented
    - Where an object is a tangible entity that exhibits some well-defined behavior

OOP, OOD, and OOA

- Some unified concepts …
  - Objects are "entities that combine the properties of procedures and data since they perform computations and save local state".
  - Objects serve 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.

OOP, OOD, and OOA

- 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

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.*

OOP

- Three important parts of the 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)

OOP

- 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*

OOP

- OO PL
  - Well support for the object-oriented style of programming is must for OOPL
  - That is, it should be convenient to use OO style in writing program in OOPL
  - No exceptional effort or skill should be required

OOP

- Theoretically OOP can be done using non-object oriented PL like Pascal and COBOL

- 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]

OOP

- We say that a PL supports inheritance
  - If 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, for example
  - Smalltalk, Object Pascal, and C++ are object-oriented, and
  - Ada is object-based.
-
- However, since objects and classes are elements of both kinds PL
  - object-oriented design methods can be used with both

Object-Oriented Design (OOD)

- 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.

- What then is object-oriented design?

OOD
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.*

OOD

- Two important parts of the 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.

OOD

- OOD
  - support for object-oriented decomposition
  - uses class and object abstractions to logically structure systems

- Structured Design
  - uses algorithmic abstractions

Object-Oriented Analysis (OOA)

- Object Model has influenced even earlier phases of SE

- Structured Analysis
  - focuses upon the flow of data within a system.

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

OOA
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.*

OOA, OOD, OOP

- How are OOA, OOD, and OOP related?

- OOD uses outcome of OOA
- OOP uses outcome of OOD

- That is, OOA → OOD → OOP

Elements of the Object Model
Kinds of Programming Pattern

- Most Programmers work in one PL and use only one programming style
  - Unable to think Alternate ways (more appropriate style) to solve problem

- 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

Kinds of Programming Pattern

- Five main types of Programming Style and respective Abstraction used

– Procedure-oriented	Algorithms
– Object-oriented	Classes and objects
– Logic-oriented	Goals (Predicate Calculus)
– Rule-oriented	If-then rules
– Constraint-oriented	Invariant relationships

Kinds of Programming Pattern

- 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
  - indeed, this programming paradigm 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**.

Object Model (OM)

- There are four major elements of OM
  - Abstraction
  - Encapsulation
  - Modularity
  - Hierarchy

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

Object Model

- There are three minor elements of the OM
  - Typing
  - Concurrency
  - Persistence

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

Object Model

- Without this conceptual framework, you may be programming in a OOPL (such as C++, Java, or C#).
- But, your design is going to smell like a procedural PL (such as C, FORTRAN, or Pascal)
- You will miss expressive power of the object-oriented language
- More importantly, it is likely that you have not mastered the complexity of the problem at hand.

Elements of the Object Model
1. Abstraction

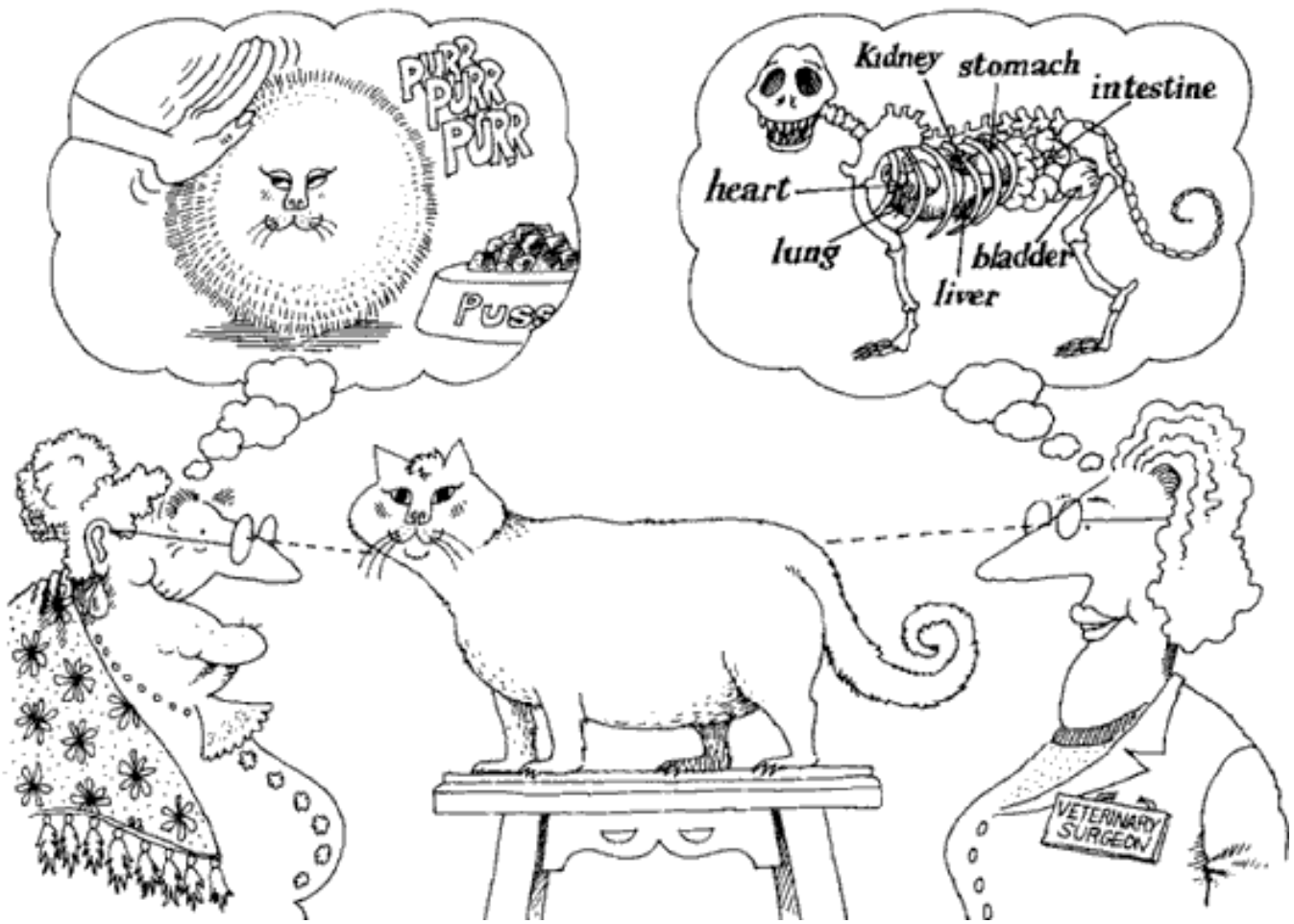
- Abstraction is one of the fundamental ways to cope with complexity
- **Abstraction** arises from a recognition of similarities between certain objects, situations, or processes in the real world, and the decision to concentrate upon these similarities and to ignore for the time being the differences
- **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 reader or user and suppresses details that are, at least for the moment, immaterial or diversionary
- 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

Elements of the Object Model
1. Abstraction

- Combining these different viewpoints, we define an abstraction 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****.*

perspective of the viewer



Elements of the Object Model
1. Abstraction

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

- This behavior/implementation division is called ***abstraction barrier*** 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***, through which an abstraction captures the entire behavior of some object, no more and no less, and offers no surprises or side effects that beyond the scope of the abstraction.

Elements of the Object Model
1. Abstraction

- Spectrum/Kinds of Abstractions (most to least useful)
  - Entity Abstraction
  - Action Abstraction
  - Virtual Machine Abstraction
  - Coincidental Abstraction

Elements of the Object Model
1. Abstraction

- Entity Abstraction
  - An object that represents a useful model of a problem domain or solution-domain entity
  - Most Useful
  - We should strive to build Entity Abstractions, as it closely models the problem domain

Elements of the Object Model
1. Abstraction

- Action Abstraction
  - An object that provides a generalized set of operations, all of which perform the same kind of function

Elements of the Object Model
1. Abstraction

- 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

Elements of the Object Model
1. Abstraction

- Coincidental Abstraction
  - An object that packages a set of operations that have no relation to each other
  - Least useful (Not useful)

Elements of the Object Model
1. Abstraction

- **Client Object** uses resources of **Server Object**

- 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 encompasses (include) the ***responsibilities*** *of an object, namely, the behavior for which it is* held accountable

Elements of the Object Model
1. Abstraction

- 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***.

Elements of the Object Model
1. Abstraction

- 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).

Elements of the Object Model
1. Abstraction

- *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, 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

Elements of the Object Model
1. Abstraction

- 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 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 PL* (Ada, Smalltalk, and C++, respectively). They all mean virtually the same thing **can be used interchangeably**.

Elements of the Object Model
1. Abstraction

- All abstractions have static as well as dynamic properties.

- For example, a file object
  - amount of space
  - Name
  - 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
  - Its contents may change.

Elements of the Object Model
1. Abstraction

- All abstractions have static as well as dynamic properties.
  - Car
    - Static Property: Registration No., Company Name
    - Dynamic Property: Current Speed, Owner Name
  - Person
    - Name, Date of Birth, SSN
    - Cell Phone, Address, Hobby, Salary

Elements of the Object Model
1. Abstraction

- 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 (reveals) 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**.

Elements of the Object Model
1. Abstraction

- **Examples of Abstraction (Assume)**
- Sensors (Air, Water, Humidity, Light, pH, etc.)

- Temperature Sensor
  - Along with Location Info.

- Responsibility?
  - Reporting the current Temp. in F

Elements of the Object Model
1. Abstraction

```
typedef float Temperature;          //Temperature in degrees Fahrenheit
typedef unsigned int Location;      // Number uniquely denoting the location of a sensor
class TemperatureSensor
{
    Public:
        TemperatureSensor(Location);
        ~TemperatureSensor() ;

        void calibrate(Temperature actualTemperature);

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

Elements of the Object Model
1. Abstraction

- Private Parts
  - Its representation (Implementation) is Hidden in Private parts of object

- Instance (Object that can be operated upon)

```
Temperature temperature;
//Creates two different objects (instances) of the class “TemperatureSensor)
TemperatureSensor greenhouselSensor(l);
TemperatureSensor greenhouse2Sensor(2);
temperature = greenhouselSensor.currentTemperature();
```

Elements of the Object Model
1. Abstraction

- Invariants (associated with the operation currentTemperature)
  - Pre-condition
    - Object will be created with valid location
  - Post-condition
    - Returns Temperature Value in Fahrenheit

Elements of the Object Model
1. Abstraction

- Passive VS Active Object
  - Passive abstraction is one in which some client object must operate on abstracted entity to get service (e.g., to get current Temperature)
  - In Active abstraction, abstracted entity acts upon (rather than being acted upon by) other objects whenever some condition arise (e.g., I need SMS, when temperature is not in a given range)

- Responsibility Changes in Active abstraction
  - Now, Sensor is responsible to report current temperature
  - Earlier, client was responsible to demand temp.

- This scenario can be implemented with the "callback" feature of PL

- When client creates an Object
  - An additional information, particularly, CallBack function also needs to be passed. This callback function will be called on detecting predetermine condition
- Server Object calls “CallBack” function when some condition arises

Elements of the Object Model
1. Abstraction

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

void calibrate(Temperature actualTemperature);

void establishSetpoint(Temperature setpoint, Temperature delta);

Temperature currentTemperature() const;     //Client can still inquire

private:
…
};
```

Elements of the Object Model
1. Abstraction

- Object may collaborate with other objects to achieve some behavior
  - Plan Condition & Growing Plan

```
// Structure denoting relevant plan conditions
struct Condition                          //Struct Vs. Class
{
    Temperature temperature;
    Lights lighting;
    pH acidity;
    Concentration concentration;
};
```

Elements of the Object Model
1. Abstraction

```
class GrowingPlan {
public:
  GrowingPlan(char* name);
  virtual ~GrowingPlan();

  void clear();
  virtual void establish(Day, Hour, const Condition&);
  const char* name() const;
  const Condition& desiredConditions(Day, Hour) const;
protected:
…
};
```

Elements of the Object Model
1. Abstraction

- Constructor and Destructor (Creates/Destroys)
  - GrowingPlan(), ~GrowingPlan()
  - Called Meta-Operations
- Modifiers – Update state
  - Clear(), Establish()
- Selectors (Returns/Selects state value)
  - name(), desiredConditions()

Elements of the Object Model Example

- Example in C++
  - Abstraction
  - Encapsulation

Elements of the Object Model
2. Encapsulation

- 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.

- No part of a complex System should depend on the internal details of any other part

Elements of the Object Model
2. Encapsulation

- **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 hiding***

Elements of the Object Model
2. Encapsulation

- ***Information Hiding***
  - 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)

- **Encapsulation** provides explicit barriers among different abstractions of system
  - objects at one level of abstraction are shielded from implementation details at lower levels of abstraction

Elements of the Object Model
2. Encapsulation

- 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.

Elements of the Object Model
2. Encapsulation

- The interface of a class is the one place where we assert/state 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.

Elements of the Object Model
2. Encapsulation

- **Definition**

- *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.*

Elements of the Object Model
2. Encapsulation

- Example

- Consider 3 abstractions
  - Temperature Sensor
  - Heater (Turn ON, Turn OFF, IsON)
  - Temperature Control
    - Higher-level behavior build on a top of Sensor & Heater
    - Adds new semantic, prevent frequent ON, OFF of Heater; when temperature is near boundary conditions (hysteresis)

Elements of the Object Model
2. Encapsulation

- Example

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


Class Heater {
        public:
                Heater(location);
                ~Heater();
                void turnOn();
                void turnOff();
                Boolean isOn() const;
        private:
        …
};                      // This interface represent all that a client needs to know
                        // In-side implementation may highly complex using
                        // Serial Ports [Class] to send signals to distant Heaters
```

Elements of the Object Model
2. Encapsulation

- In future, developer may choose to use Memory-mapped I/O instead of serial communication lines

- This means Implementation is modified. But, it does not require to change Interface

- As long as Interface (functional behavior of abstraction) is not changed
  - Client code that uses this class need not have to be modified
  - Unless a particular client depends upon the time or space semantics of original implementation (highly undesirable)

Elements of the Object Model
2. Encapsulation

- The representation of an object is often changed by developer (based on system's performance) so that more efficient algorithm can be applied

- This ability to change the representation of an abstraction without disturbing any of its clients is essential **benefits of encapsulation**

Elements of the Object Model
2. Encapsulation

- The Encapsulation is supported by different PL in various ways and extents

- For example, C++ offers flexible control over the member data elements and member functions. Member may be placed in
  - Public (visible to all)
  - Private or (fully encapsulated)
  - Protected part (visible to class and its subclasses)

Elements of the Object Model
2. Encapsulation

- C++ also supports the notion of friends:
  - Cooperative classes that are permitted to see/use each other's parts.

- Hiding is relative concept.
  - The underlying representation of an object can be revealed, if developer wish to do so.
  - Encapsulation cannot stop a developer from doing stupid things
  - Further, no PL prevents a human from literally seeing the implementation of a class (although, an OS can prevent)
  - There are times when one must study the implementation of a class to really understand its meaning

Elements of the Object Model
3. Modularity

- The act of partitioning a program into individual components can reduce its complexity to some degree

- partitioning a program creates a number of well defined, documented boundaries within the program.

Elements of the Object Model
3. Modularity

- In some languages, such as Smalltalk, there is no concept of a module

- In many others, including Object Pascal, C++, CLOS, and Ada, 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.*

Elements of the Object Model
3. Modularity

- 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

Elements of the Object Model
3. Modularity

- Module and encapsulation go hand in hand
  - PL supports notion of
    - Interface of module and
    - its Implementation

- Again, different PL supports module in diverse way

- Module in also known as Unit, Package, Assembly, DLL, etc.

- Deciding the right set of modules for a given system is hard problem

Elements of the Object Model
3. Modularity

- 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

Elements of the Object Model
3. Modularity

- 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;
  - it should be possible to change the implementation of module without knowledge of the implementation of other modules and without affecting the behavior of other modules

Elements of the Object Model
3. Modularity

- The cost of changing (and re-compilation) an interface is very high compare to changing an implementation
  - Requires re-compilation of all module that uses services of the module directly or indirectly.
  - For this reason, For this reason, a module's interface should be as narrow as possible
  - hide as much as you can in the implementation of a module
  - Incremental addition to interface is better than handling extra interface

Elements of the Object Model
3. Modularity

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

- 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
  - cohesive (by grouping logically related abstractions) and
  - loosely coupled (by minimizing the dependencies among modules).

Elements of the Object Model
3. Modularity

- *Definition*

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

Elements of the Object Model
3. Modularity

- Other issues that can affect modularization decisions:
  - Package classes such that Reuse is convenient
  - Size of compiled unit
  - Work assignment to different teams
  - To save documentation & related effort
  - Security (the code that need security)

Elements of the Object Model
3. Modularity

- Logical Design
  - Identification of Classes & Objects
- Physical Design
  - Identification of Modules
- Both the designs are done iteratively
  - That is, No one follows the other completely

Elements of the Object Model
3. Modularity

- Example

- Module-X //depends on 3 modules

```
#include<string.h>
#include<math.h>
#include<moduleY.h>
…
Void main()
{ …}
```

Elements of the Object Model
3. Modularity

- Other way of division of Software
  - 2-tier Client/Server Architecture
  - 3-tier or N-tier Architecture
  - MVC Architecture
  - SOA

Elements of the Object Model
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.

Elements of the Object Model
4. Hierarchy

- 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).

Elements of the Object Model
4. 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.

Elements of the Object Model
4. Hierarchy

- 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

Elements of the Object Model
4. Hierarchy

- 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.**

Elements of the Object Model
4. Hierarchy

- 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
- };

Elements of the Object Model
4. Hierarchy

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

Elements of the Object Model
4. Hierarchy

- 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

Elements of the Object Model
4. Hierarchy

- 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

Elements of the Object Model
4. Hierarchy

- 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:
…
};
```

Elements of the Object Model
4. Hierarchy

- 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**

Elements of the Object Model
4. Hierarchy

- 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

Elements of the Object Model
4. Hierarchy

- 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;
};
```

Elements of the Object Model
4. Hierarchy

- 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**

Elements of the Object Model
4. Hierarchy

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

Elements of the Object Model
4. Hierarchy

- 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.**

Elements of the Object Model
5. Typing

Elements of the Object Model
5. Typing

Elements of the Object Model
5. Typing