THE OBJECT MODEL

- Programming Language Generations
- What is an Object?
- Programming Paradigms
- Elements of the Object Model
- Relationships Among Objects
- Classification

Objectives of Module 3

- Present and discuss the concept of the Object Model and its evolution.
- Present and discuss the elements of the Object Model.
- Present and discuss the idea that proper classification is very important to an object-oriented design and how and why it is difficult to obtain a proper classification.

Readings on the Object Model

- The Object Model was first introduced by Jones (1979) and Williams (1986). See:

  Jones, A. "The Object Model: A Conceptual Tool for Structuring Software" in *Operating Systems*, ed. R. Bayer et al., New York, NY: Springer-Verlag, 1979

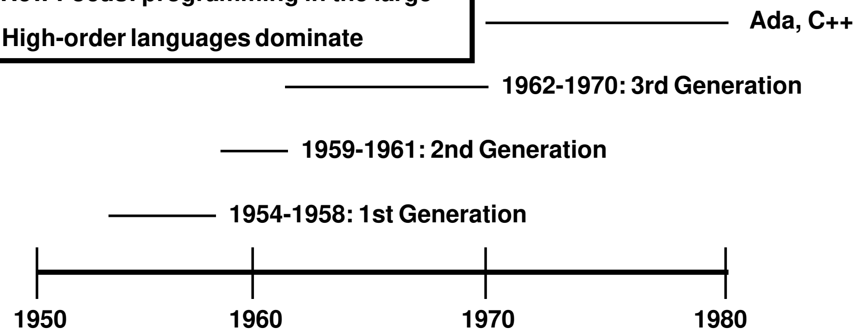
  Williams, L. *The Object Model in Software Engineering*, Boulder, CO: Software Engineering Research, 1986

- Alan Kay's Ph.D. thesis (1969) established the direction for much of the work in object-oriented programming that followed. See:

  Kay, A. *The Reactive Engine*, Salt Lake City, Utah: The University of Utah, Department of Computer Science, 1969

PROGRAMMING LANGUAGE GENERATIONS

- New Focus: programming in the large
- High-order languages dominate

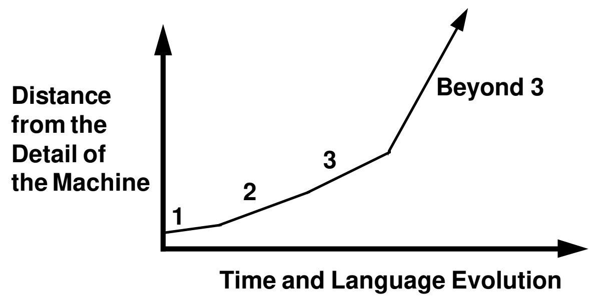


Gen	*Sample Languages*	*Language Features*
1	FORTRAN I	Mathematical expressions
2	FORTRAN II, COBOL	Subroutines, data handling
3	Pascal, Simula	Blocks, typing, classes

- **Programming Language Generations**
- *What Is an Object ?*
- *Programming Paradigms*
- *Elements of the Object Model*
- *Relationships Among Objects*
- *Classification*

Evolution of Abstraction

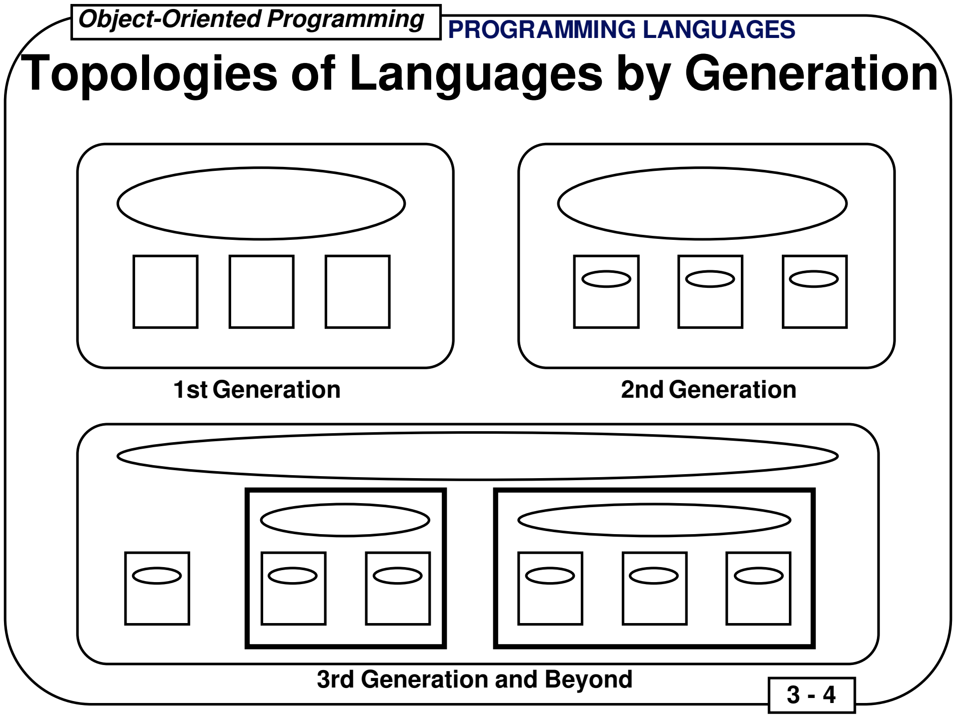
Gen	*Kind of Abstraction*
1	**Mathematics**
2	**Algorithm and procedures**
3	**Data and data models of real-world entities**
Beyond	**Objects and object models of real-world entities**



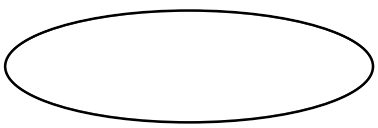
There is a growing need to increase the level of abstraction when designing software systems, especially as the complexity of these systems increases.

In Object-Oriented Programming, the level of abstraction is at the level of the object. The Object encapsulates data specific to an object or class of objects (the member data, which represents the attributes of the class of objects) and functions which operate on that data.

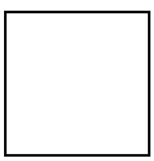
Different Object-Oriented Programming Languages (OOPL's) have different ways of defining classes, but the inclusion of member data and member functions in a class is a common feature of most OOPL's, particularly Ada and C++.



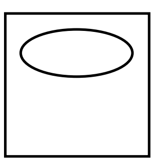
Key to Symbols



Data



Subprogram



Subprogram containing data and other subprograms

Package containing local data and subprograms, exporting only desired subprograms and data while hiding others

A Shift in Focus

Given that:

Verbs => Procedures and Functions

Nouns => Data

Then:

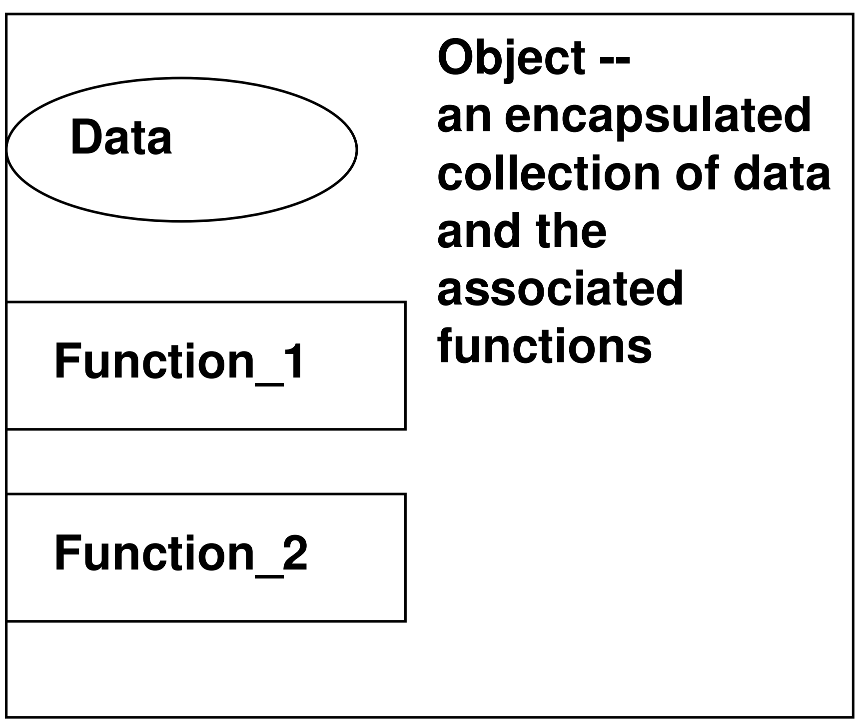
Function-oriented Program = Collection of Verbs Supported by Nouns

Object-oriented Program = Collection of Nouns Supported by Verbs

Which is a More Realistic Model of the World?

- **A collection of functions being performed?**
- **A collection of objects interacting with each other?**

Data becomes the basis of the modular breakdown of the code, as opposed to functions. Functions are grouped with the data they operate on.



WHAT IS AN OBJECT?

Informal, Intuitive Definition

An *object* is an integral entity which can:

- **change state**
- **behave in certain discernable ways**
- **be manipulated by various forms of stimuli**
- **stand in relation to other objects**

***Objects* :**

- **exist, occupy space, and assume a state**
- **possess attributes**
- **exhibit behaviors**

- *Programming Language Generations*
- **What Is an Object?**
- *Programming Paradigms*
- *Elements of the Object Model*
- *Relationships Among Objects*
- *Classification*

Formal Definition

Object Concept **- objects have a permanence and identity apart from any operation upon them**

Formal definition of an object from the perspective of OOD:

> ***Object*** **- an entity which has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class; the terms instance and object are interchangeable**
>
> -- Grady Booch, *Object-Oriented Design with Applications*, 1991, Page 77

Informal definition of an object from the perspective of human cognition:

Object - any of the following:
- a tangible and/or visible thing
- something that may be apprehended intellectually
- something toward which thought or action is directed

-- Grady Booch, *Object-Oriented Design with Applications*, 1991, Page 76

Three key aspects of an object:
- state (sometimes realized as attributes)
- identity
- behaviors

Examples of Non-Objects

- Attributes, such as time, beauty, or color
- Emotions, such as love or anger
- Entities which are normally objects but are, instead, thought of as attributes of objects when a particular problem space is considered

Temperature

```
Oven_Temp : TEMPERATURE
   := 350.0;  -- degrees F
```

Temperature as an object

Temperature Sensor

```
type SENSOR is record
   Temp : TEMPERATURE;
   Redundancy : MULTIPLEX;
   Location: MEMORY_ADDRESS;
end record;
Oven_Temp : SENSOR := (
   Temp => 350.0,  -- degrees F
   Redundancy => TRIPLEX,
   Location => 16#1a0#  );
```

Temperature as an attribute of an object

Kinds of Software Objects

- **Real-world, tangible objects with boundaries that may or may not be clearly defined**
- **Inventions of the design process which collaborate with other objects to provide some higher-level behavior**
- **Intangible events or processes with well-defined conceptual boundaries**

Tangible	**Clearly-defined boundaries**
	No clearly-defined boundaries
Intangible	**Clearly-defined boundaries**
	No clearly-defined boundaries
	Events or processes
	Inventions of the design process

Objects in a software system come from two key sources:

- **discovery**, where the software objects map to real-world objects discovered during the analysis of the problem
- **invention**, where the software objects have been invented by the designers

Dissecting an Object

Formal definition of an object from the perspective of OOD:

> ***Object*** **- an entity which has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class; the terms instance and object are interchangeable**
>
> -- Grady Booch, *Object-Oriented Design with Applications*, 1991, Page 77

This definition of an object refers to three key features:

- **State**
- **Behavior**
- **Identity**

These key features will be discussed in detail.

Consider a simple variable in a program, such as

I : Integer range 1..20 := 12;

State --

The value of the variable (in this case, 12). Note that one of the attributes of this variable is that it can only take on values from 1 to 20.

Behavior --

A variable like this is passive, meaning that it cannot take on an activity of its own volition, but there is a set of operations (+, -, /, *, etc.) that may operate upon it.

Identity --

I is the name, or identity, of this variable. **I** is a member of the class **Integer**, although its value range is restricted.

State

State of an object **- encompasses all of the (usually static) properties of the object plus the current (usually dynamic) values of each of these properties**

-- Grady Booch, *Object-Oriented Design with Applications*, 1991, Page 78

Property or attribute of an object **- a part of the state of the object which is an inherent or distinctive characteristic, trait, quality, or feature that contributes to making an object uniquely that object**

-- Grady Booch, *Object-Oriented Design with Applications*, 1991, Page 78

All properties have some value:

- a scalar quantity
- a vector quantity or an object

Because every object has state, every object takes up some amount of space, be it physical space or computer memory.

In OOPL's, member data, which define the attributes of a class of objects, are defined in a part of the specification of the object's class. For example,

```
class Complex {
protected:
  float real_part;
  float imag_part;
public:
  Complex (float rp=0.0, float ip=0.0);  // constructor
  Complex &operator+ (Complex &arg);  // a+b
  Complex &operator= (Complex &arg);  // a=b
  float real(void);  // f = a.real();
  float imag(void);  // f = a.imag();
};
```

State of an Object - Example

Temperature Sensor

```
type TEMPERATURE_SENSOR is record
   Temp : TEMPERATURE;  -- degrees F
   Redundancy : MULTIPLEX;
   Location: MEMORY_ADDRESS;
end record;
Oven_Temp : TEMPERATURE_SENSOR := (
   Temp => 350.0,  -- degrees F
   Redundancy => TRIPLEX,
   Location => 16#1a0#  );
```

Objects of class TEMPERATURE_SENSOR, such as Oven_Temp, have three attributes:

- ***Temp*, a dynamic attribute which changes with time**
- ***Redundancy*, a static attribute (the number of sensed points) which is fixed when the object is created**
- ***Location*, a static attribute which is fixed when the object is created**

Behavior

Behavior of an object **- how an object acts and reacts, in terms of its state changes and message passing**

-- Grady Booch, *Object-Oriented Design with Applications*, 1991, Page 80

Operation **-- some action that one object performs upon another in order to elicit a reaction**

-- Grady Booch, *Object-Oriented Design with Applications*, 1991, Page 80

The terms *operation* and *message* are interchangeable.

Method **-- operation that a client may perform upon an object**

-- Grady Booch, *Object-Oriented Design with Applications*, 1991, Page 80

The terms *method* and *member function* are interchangeable.

In OOPL's, member functions, which define those operations that may be invoked on an object by its clients, are defined in a part of the specification of the object's class. For example,

```
class Complex {
protected:
  float real_part;
  float imag_part;
public:
  Complex (float rp=0.0, float ip=0.0);  // constructor
  Complex &operator+ (Complex &arg);  // a+b
  Complex &operator= (Complex &arg);  // a=b
  float real(void);  // f = a.real();
  float imag(void);  // f = a.imag();
};
```

Behavior of an Object - Example

```
package Temperature_Sensor is

  type STATUS is (NOT_OK, OK);
  type TEMPERATURE is FLOAT range -400.0 .. 3_000.0; -- deg F
  type MULTIPLEX is (SIMPLEX, DUPLEX, TRIPLEX);
  type MEMORY_ADDRESS is INTEGER range 0 .. 1_024;

  type OBJECT is record
    Temp       : TEMPERATURE;
    Redundancy : MULTIPLEX;
    Location   : MEMORY_ADDRESS;
  end record;

  function Current_Temperature (Item : in OBJECT)
    return TEMPERATURE;

  function Reliability (Item : in OBJECT)
    return STATUS;

end Temperature_Sensor;
```


Using the TEMPERATURE_SENSOR Package

```
with Temperature_Sensor;
with Console;
procedure Show_Oven_Temperature is

  Oven_Temp : Temperature_Sensor.OBJECT :=
    (Temp       => 0.0,  -- initial dummy condition
     Redundancy => Temperature_Sensor.TRIPLEX,
     Location   => 16#1a0#);

begin -- Show_Oven_Temperature

  -- Display the current temperature
  Console.Put("Current oven temperature is ");
  Console.Put (FLOAT(Temperature_Sensor.Current_Temperature
    (Oven_Temp)), 4, 1, 0);
  Console.New_Line;

end Show_Oven_Temperature;
```

Behavior - Kinds of Operations

- *Modifier* - an operation that alters the state of an object, such as a get_with_update or put operation
- *Selector* - an operation that accesses the state of an object, but does not alter the state, such as a get operation
- *Iterator* - an operation that permits all parts of an object to be accessed in some well-defined order, such as movement through a linked list
- *Constructor* - an operation that creates and object and/or initializes its state
- *Destructor* - an operation that frees the state of an object and/or destroys the object itself

Behavior - The Protocol of an Object

Protocol **- all of the methods and free subprograms [procedures or functions that serve as nonprimitive operations upon an object or objects of the same or different classes] associated with a particular object**

-- Grady Booch, *Object-Oriented Design with Applications*, 1991, Pp 82-83

The protocol of an object defines the envelope of that object's allowable behavior, comprising the entire external view of the object (both static and dynamic).

In OOPL's, the *protocol* of an object is evident in its class definition.

```
class Complex {
protected:
  float real_part;
  float imag_part;
public:
  Complex (float rp=0.0, float ip=0.0);  // constructor
  Complex &operator+ (Complex &arg);  // a+b
  Complex &operator= (Complex &arg);  // a=b
  float real(void);  // f = a.real();
  float imag(void);  // f = a.imag();
};
```

Behavior - Objects as Machines

The *Finite State Machine* provides a good model for some objects.

Objects may be either active or passive:

- ***Active Object*** **- an object that encompasses its own thread of control**
- ***Passive Object*** **- an object that does not encompass its thread of control**

Since an object has state, the order in which operations are invoked is important. This gives rise to the view of an object as an independent *machine*. For some objects, time ordering of their operations is so important that the object's behavior can be formally characterized in terms of a *finite state machine*.

Active objects are autonomous, exhibiting a behavior without being operated upon by another object.

Passive objects can only undergo a state change when explicitly acted upon.

Some OOPL's, like Ada, have constructs to support the definition of active objects. In Ada, these are called tasks, and they begin execution as soon as their declarations are encountered.

Identity

Identity **- that property of an object which distinguishes it from all other objects**

-- Khoshafian and Copeland, "Object Identity," *SIGPLAN Notices*, Volume 21, Issues 11, November 1986, Page 406

I : Integer range 1..20 := 12;

identity of the object

The failure to distinguish between the name of an object and the object itself is the source of many errors in object-oriented programming.

Lifetime of an Object - the time span extending from the time an object is first created (and consumes space) until that space is reclaimed

Note that an object can continue to exist even if all references to it are lost.

Identity - Object Assignment

Object Assignment **differs from copying in that in object assignment, the identity of an object is duplicated by assignment to a second name. Two names then refer to the same object.**

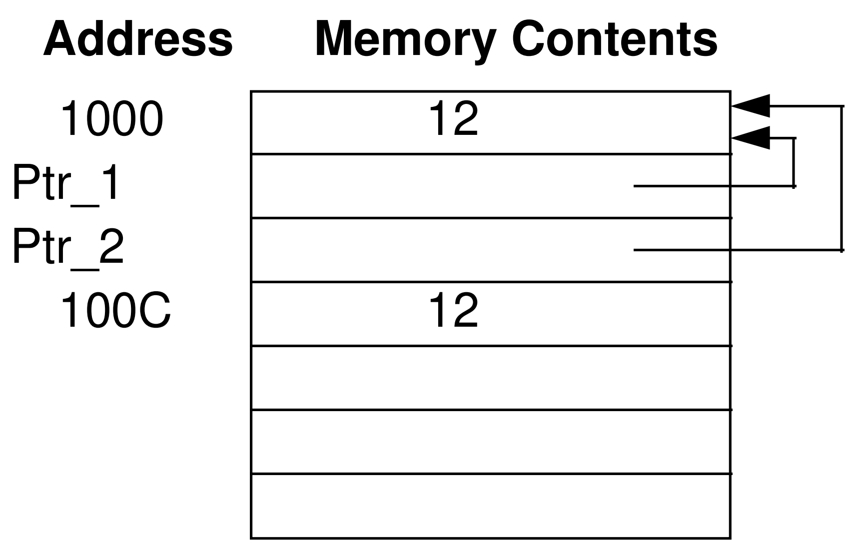
Conventional Assignment **refers to the act of copying the state information of one object into another object. The state of two objects is now the same, but the state of one object may be changed without affecting the other.**

Identity - Equality

Like assignment, *Equality* can have two meanings:

- **two names are equal if they designate the same object**
- **two names are equal if they designate different objects but their state is identical**

Equality can be confusing. One meaning refers to two entities addressing the same space, the other refers to the contents of the space addressed.



Ptr_1 = Object(1000) because they address the same space

Object(1000) = Object(100C) because they contain the same value

What is a Class?

Class **- a set of objects that share a common structure and a common behavior**

-- Grady Booch, *Object-Oriented Design with Applications*, 1991, Page 93

A class represents only an *abstraction*, whereas an object, *an instance of a class*, is a concrete entity that exists in time and space.

What is NOT a Class?

An *object* is not a class, but a class may be an object (to be discussed in the OOD course).

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

The Class as a Contractual Binding

The class captures the structure and behavior common to all related objects, serving as a binding contract between an abstraction and all of its clients.

Strongly typed programming languages can detect violations of the contract that is a class during compilation.

Two views of a class:

- ***Interface*** **- the outside view of a class, emphasizing the abstraction while hiding the structure and details of how its behavior works**
- ***Implementation*** **- the inside view of a class, which details the internal structure of a class and the details of how its behavior works**

In Ada, we have the specification and the body. The specification defines the interface and the body defines the implementation. All four Ada program units (the subprogram, the package, the generic, and the task) have separate forms for their specifications and bodies.

In C++, we have the declaration and the definition. The declaration defines the interface and the definition defines the implementation. Definition information may be mixed in with the declaration, however.

The Interface to a Class

The interface to a class consists of:

- **primarily, the declarations of all operations applicable to instances of the class; these operations may be invoked by clients of the class objects**
- **the declaration of other classes**
- **constants**
- **variables**
- **exceptions**

The last four are included if they are needed to complete the abstraction.

The Interface to a Class, Continued

The interface to a class can be divided into three parts:

- ***Public*** **- a declaration that is visible to all clients of the objects of a class**
- ***Protected*** **- a declaration that is not visible to any other classes except the subclasses of the class**
- ***Private*** **- a declaration that is not visible to any other classes**

C++ does the best job in allowing a developer to make explicit distinctions among these different parts of a class interface. Ada permits declarations to be public or private, but not protected.

An example of a C++ class with protected, private, and public members:

```
class Intermixed {
protected:
  float x;
private:
  float y;
public:
  void set_x(float);  // set value of X
  void set_y(float);
  void print(void);   // print out X and Y
};
```

The State of an Object

- Defined in private part of class declaration
- Constant and variable declarations

Why is the State of an Object NOT in the Implementation?

- Needed by the compiler
- Technology not sufficiently advanced

The State of an Object

The state of an object is usually represented as constant and variable declarations placed in the private part of a class interface. This encapsulates the representation common to the objects of a class, and changes to this representation do not have a functional affect on the clients.

Why is the State of an Object NOT in the Implementation?

Placing state information in the implementation of a class would completely hide it from the clients, but, with today's technology, placing state information in the implementation rather than the private interface of a class would require either object-oriented hardware or very sophisticated compiler technology. Compiler technology can solve this problem, but the compiler must be able to discern information about the size of the object of the class.

PROGRAMMING PARADIGMS

Most programmers work in one language and use only one programming style. They program in a paradigm 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.

-- Jenkins and Glasgow, "Programming Styles in Nail," *IEEE Software*, Volume 3, Number 1, Page 48 (Jan 1986)

Main Kinds of Programming Paradigms

Paradigm	*Kinds of Abstractions Employed*
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

- *Programming Language Generations*
- *What Is an Object ?*
- **Programming Paradigms**
- *Elements of the Object Model*
- *Relationships Among Objects*
- *Classification*

No Single Paradigm is Best for All Kinds of Applications!

Each style is based on its own conceptual framework.

Examples:

- Rule-oriented programming is best for the design of a knowledge base.
- Procedure-oriented programming is best for the solution of sets of simultaneous equations.
- Object-oriented programming is best for industrial-strength software in which complexity is the dominant issue.

ELEMENTS OF THE OBJECT MODEL

The *Object Model* is the conceptual framework for all things object-oriented.

Major Elements

- ✓ Abstraction
- ✓ Encapsulation
- ✓ Modularity
- ✓ Hierarchy

Minor Elements

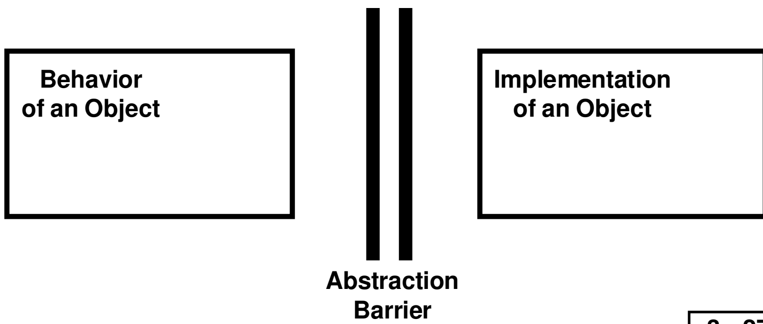
- ✓ Typing
- ✓ Concurrency
- ✓ Persistence

- *Programming Language Generations*
- *What Is an Object ?*
- *Programming Paradigms*
- **Elements of the Object Model**
- *Relationships Among Objects*
- *Classification*

The Object Model is the conceptual framework for all things object-oriented. Without this conceptual framework, you may program in a language like C++ or Ada, but your design will "smell" like FORTRAN, Pascal, or C. Many of the benefits of the language and its potential will be lost.

Abstraction

Abstraction - what distinguishes one kind of object from another kind of object



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.

-- Grady Booch, Object-Oriented Design with Applications, 1991, Page 39

Abstraction and the Problem Domain

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

"Determining the correct set of abstractions" is covered in detail in the next Module.

Kinds of Abstraction

Entity abstraction **- an object that represents a useful model of an entity in 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 abstraction **- an object that packages a set of operations that have no relation to each other**

Entity --

```
class Sensor {
private:
  int data;
public:
  int read(void);
};
```

Entity Abstractions

Client **- an object that uses the resources of another object**

Behavior of an object **- the operations that a client may perform upon the object (the** ***protocol*** **of the object) and the operations that the object may perform upon other objects**

All entity abstractions may have two kinds of properties:

- ***Static*** **- fixed for the life of the object; example: a file's name or identity**
- ***Dynamic*** **- can vary during the life of the object; example: a file's size**

Encapsulation

Encapsulation, or Information Hiding **- the process of hiding all the details of an object that do not contribute to its essential characteristics**

-- Grady Booch, *Object-Oriented Design with Applications*, 1991, Page 46

Abstraction and Encapsulation are complementary concepts:

- Abstraction hides the implementation of an object from most clients, focusing on the outside view of an object
- Encapsulation prevents clients from seeing the inside view of an object, where the behavior of the object is implemented and the state information on the object is retained (in many cases)

Modularity

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

-- Grady Booch, *Object-Oriented Design with Applications*, 1991, Page 52

Classes and objects are implemented in modules to produce the architecture of a system.

There are two aspects to a module:

- The interface to a module, called a specification in Ada
- The implementation of a module, called a body in Ada

Issues Concerning Modularity

Technical --

- Class and object selection - modules are the containers of the classes and objects
- Logically-related classes and objects grouping
- Visibility of modules to other modules
- Isolation of system dependencies
- Reuse of modules across applications
- Limits placed on the size of object code segments, particularly when a compiler places one and only one module into one and only one object code segment

Non-Technical --

- Work assignments may be given on a module basis
- Modules usually serve as configuration items
- Some modules may require more security

Modules and Classes/Objects

Two entirely independent design decisions:

- **Finding the right classes and objects**
- **Organizing the classes and objects into separate modules**

The selection of classes and objects is a part of the logical design.

The identification of modules is a part of the physical design.

Logical and physical design decisions must take place iteratively; one cannot be completed before the other.

Hierarchy

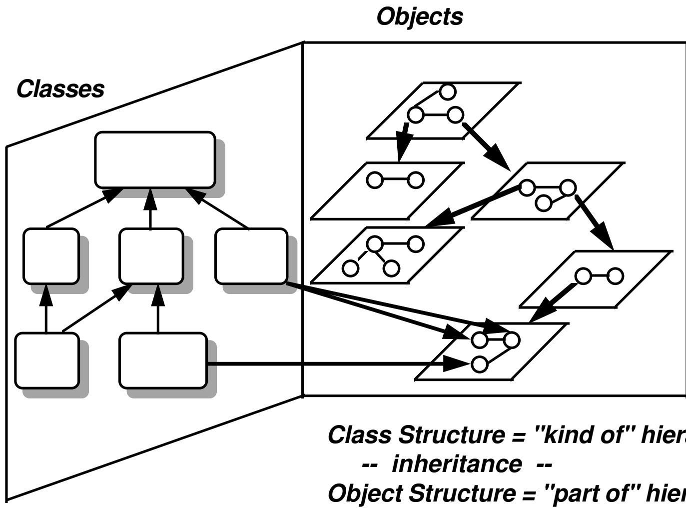
Hierarchy **- the ranking or ordering of abstractions**

-- Grady Booch, *Object-Oriented Design with Applications*, 1991, Page 54

The two most important hierarchies in a complex system:

- **the *class structure* (the "kind of" hierarchy)**
- **the *object structure* (the "part of" hierarchy)**

Two Key Hierarchies



Class Structure = "kind of" hierarchy
-- inheritance --
Object Structure = "part of" hierarchy
-- aggregation --

This picture, also known as a canonical representation of a class-based system, shows the objects in a system and their relationships as containers of subordinate objects. The classes in the system and their relationships to other classes (inheriting relationships) are also shown. Finally, a pairing of objects with their classes is shown.

Each object belongs to one and only one class at a given time, although subclasses may exist (and an object may change classes from time to time).

Each class is realized by zero or more objects.

Typing

Typing **- 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**

-- Grady Booch, *Object-Oriented Design with Applications*, 1991, Page 59

A *type* is very similar to a *class*. Typing allows abstractions to be expressed in such a way that the programming language used to implement the design can be used to enforce the design decisions.

Languages may be strongly typed, weakly typed, or untyped. All three kinds of languages may be object-oriented or object-based.

In a strongly typed language, all expressions are guaranteed to be type-consistent.

Some Benefits of Strong Typing

- **Runtime crashes of programs reduced**
- **Early error detection**
- **Type declarations help to document programs:**

```
type VELOCITY is new FLOAT range 0.0 .. 1_000.0; -- MPH
X : VELOCITY;
Y : FLOAT;
```

- **Code efficiency may be improved**

- With strong type checking, many problems which could cause runtime crashes of programs will be caught at compile time. For example, calling a subroutine with two integer parameters when it required three integer parameters or calling a subroutine with an integer and a string when it required an integer and a character can be caught at compile time.
- Early error detection afforded by strong type checking can reduce the development time, cost, and effort. The earlier an error is caught, the better.
- Type declarations help to document programs. The declaration of X below is much better than the declaration of Y:

  X : VELOCITY;

  Y : FLOAT;
- Many compilers can generate more efficient object code if types are declared. In the following example, a byte may be used instead of a full integer:

  type CHAR_COUNTER is range 0 .. 128;

Static Typing and Dynamic Binding

Static Typing, Static Binding, or Early Binding **- the types of variables are fixed at compile time**

Dynamic Binding or Late Binding **- the types of variables are not known until runtime**

Combinations of strong and weak typing with static and dynamic binding may be supported in various languages in various ways:

- Ada supports strong typing and static binding
- C++ supports strong typing and static or dynamic binding
- Smalltalk has no typing but supports dynamic binding

Polymorphism and Typing

Polymorphism **- the 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**

Monomorphism **is the opposite of polymorphism, so a monomorphic object may only respond to the set of operations associated with its own class.**

A polymorphic object may respond to the set of operations associated with the superclass and also the set of operations associated with its own class.

Ada supports only monomorphism while C++ supports both monomorphism and polymorphism. Polymorphism exists when the features of inheritance and dynamic binding interact with each other. Languages which are both strongly typed and statically bound, such as Ada, cannot support polymorphism.

Concurrency

Concurrency **- the property that distinguishes an active object from one that is not active**

-- Grady Booch, *Object-Oriented Design with Applications*, 1991, Page 66

An object is an excellent candidate for a concurrent entity because:

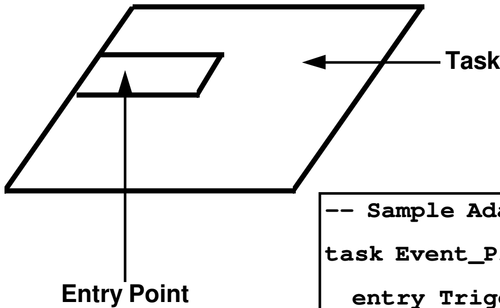
- **it implicitly defines a unit of distribution and activity**
- **it explicitly defines a communication interface**

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 concurrent system may have many threads of control, some transitory and some lasting the lifetime of the system.

Ada supports the declaration of concurrent objects, using its task program unit. C++ does not support concurrent objects directly, but it can by using the UNIX fork system call.

Tasks as Concurrent Objects

In Ada, the Ada runtime system implements the tasking model. This model can be implemented on one or many CPUs.




```
-- Sample Ada Task Specification
task Event_Process is
  entry Trigger (Input : in KIND);
end Event_Process;
-- Creating two Event_Process tasks
Processor1, Processor2 : Event_Process;
```

Persistence

Persistence **- 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 object's location moves from the address space in which it was created)**

-- Grady Booch, *Object-Oriented Design with Applications*, 1991, Page 70

An object in software takes up some amount of space and exists for a particular amount of time. Both its state and class must persist.

The spectrum of object persistence includes:

- Intermediate results in expression evaluation
- Local variables created during the execution of subprograms
- Global variables
- Heap items that exist outside the scope of their creation
- Data that exists between executions of a program
- Data that outlives the program

RELATIONSHIPS AMONG OBJECTS

An object of and by itself is usually uninteresting. However, a system of objects, wherein the objects collaborate with one another to define the behavior of the system, is intensely interesting.

Two kinds of object hierarchies are extensively employed in OOD:

- Using relationships, where one object employs the resources of another
- Containing relationships, where one object contains one or more other objects

- *Programming Language Generations*
- *What Is an Object ?*
- *Programming Paradigms*
- *Elements of the Object Model*
- **Relationships Among Objects**
- *Classification*

Using Relationships

Three roles:

- ***Actor* - operates upon other objects; an *active object***
- ***Server* - is only operated upon by other objects; a *passive object***
- ***Agent* - can do both**

Given a collection of objects involved in using relationships, each object may play one of three roles:

- Actor - an object that can operate upon other objects but that is never operated upon by other objects; an active object
- Server - an object that never operates upon other objects but is only operated upon by other objects; a passive object
- Agent - an object that can both operate upon other objects and be operated upon by other objects; an agent is usually created to do some work on behalf of an actor or another agent

Whenever one object passes a message to another with which it has a using relationship, the two objects must be synchronized. In a single thread of control, a subprogram call is adequate for synchronization. With multiple threads of control, a more complex method of synchronization must be devised in order to deal with the problems of mutual exclusion.

Using Relationships, Continued

The need for synchronization in an environment involving multiple threads of control leads to another way to classify kinds of objects:

- *Sequential object* - a passive object whose semantics are guaranteed only in the presence of a single thread of control
- *Blocking object* - a passive object whose semantics are guaranteed in the presence of multiple threads of control
- *Concurrent object* - an active object whose semantics are guaranteed in the presence of multiple threads of control

Containing Relationships

In a *containing relationship*, an object may encapsulate one or more other objects.

Advantages:

- **Reduce the number of objects**

Disadvantages:

- **Sometimes leads to undesirable tighter coupling**

In a containing relationship, an object may encapsulate one or more other objects. Some real-world object relationships are clearly containing relationships, such as the automobile engine which contains pistons, spark plugs, etc.

Containing an object rather than using an object is sometimes better because containing reduces the number of objects that must be visible at the level of the enclosing object.

Using an object is sometimes better than containing an object because containing an object leads to undesirable tighter coupling among objects in some cases.

Intelligent engineering decisions require careful weighing of these two factors.

CLASSIFICATION

- What is Classification?
- Classification and OOD
- Why is Classification So Hard?
- Approaches to Classification
- Domain Analysis

- *Programming Language Generations*
- *What Is an Object ?*
- *Programming Paradigms*
- *Elements of the Object Model*
- *Relationships Among Objects*
- **Classification**

What is Classification?

Classification **- the means whereby we order knowledge**

Recognizing the "sameness" among things

No single approach

In OOD, recognizing the sameness among things allows us to expose the commonality within key abstractions and mechanisms and eventually leads to simpler designs.

However, there is no simple approach to the problem of identifying classes and objects. The selection of classes and objects for an OOD is a compromise shaped by many competing factors.

This module focuses on heuristics useful for identifying the classes and objects relevant to a particular problem.

Classification and OOD

The identification of classes and objects is the hardest part of OOD.

The identification of classes and objects involves:

- **discovery, through which we recognize the key abstractions and mechanisms that form the vocabulary of our problem domain**
- **invention, through which we devise generalized abstractions and new mechanisms that regulate how objects should collaborate**

During classification, we group entities that have a common structure or exhibit a common behavior.

Classification is highly dependent upon the reason for the classification, and different observers naturally tend to classify the same thing differently.

The best classifications result when an incremental and iterative process is applied. The quality of a classification can only be meaningfully evaluated at later stages in the design, once clients have been built which use the abstractions.

Why is Classification so Hard?

- There is no such thing as a "perfect" classification, although some classifications are better than others.
- Any classification is relative to the perspective of the observer doing the classification.
- Intelligent classification requires a tremendous amount of creative insight.

Why is a LASER beam like a goldfish?
Because neither one can whistle.

Creative insight or idiocy?

How is a speck of dust like a thought?
Both can be conceived of.

Approaches to Classification

- Classical categorization
- Conceptual clustering
- Prototype theory

Classical Categorization

Classical Categorization **- all entities that have a given property or set of properties in common form a category; such properties are necessary and sufficient to define the category**

-- Lakoff, G. *Women, Fire, and Dangerous Things: What Categories Reveal About the Mind*, 1987, The University of Chicago Press, Page 161

Related properties are therefore the criteria for sameness among objects in the Classical Categorization approach. One can divide objects into disjoint sets depending on the presence or absence of a particular property. Properties to be considered are domain-specific.

Marvin Minsky has suggested that "the most useful sets of properties are those whose members do not interact too much. This explains the universal popularity of that particular combination of properties: size, color, shape, and substance."

-- Minsky, M. **The Society of Mind**, 1986, Simon and Schuster, New York, Page 199

Conceptual Clustering

Conceptual Clustering **- a modern variation on the classical approach in which classes (clusters of entities) are generated by formulating conceptual descriptions of these classes and then classifying the entities according to the descriptions**

Conceptual clustering is a probabilistic clustering of objects.

Prototype Theory

Prototype Theory **- a class of objects is represented by a prototypical object**

Prototype Theory - based on work in the field of cognitive psychology, a class of objects is represented by a prototypical object, and an object is considered to be a member of this class if and only if it resembles this prototype in some significant ways

Prototype Theory is often applied when classical categorization and conceptual clustering fail. For instance, try to identify entities which fall into a class called "game" by classical categorization or conceptual clustering.

Classification and OOD Revisited

An approach proposed by Grady Booch:

1. **Identify classes and objects according to the properties relevant to the application domain.**
2. **If this fails, cluster objects by concepts.**
3. **If either (1) or (2) fail, classify by association, through which clusters of objects are defined according to how closely each resembles some prototypical object.**

These three approaches to classification provide the theoretical foundation of object-oriented analysis, domain analysis, and other methods applied to identify classes and objects in an object-oriented design.

Sources of Classes and Objects

Proposed by Shlaer and Mellor:

- Tangible things, such as cars, telemetry data, and sensors
- Roles, such as mother, teacher, and politician
- Events, such as landing, interrupt, and request
- Interactions, such as loan, meeting, and intersection

Proposed by Ross (from the perspective of data modeling):

- People - humans who carry out some function
- Places - areas set aside for people or things
- Things - tangible physical objects or groups of objects
- Organizations - collections of people, resources, facilities, and capabilities that have a defined mission
- Concepts - principles or ideas not tangible used to track activities and/or communications
- Events - things that happen

Sources, Continued

Proposed by Coad and Yourdon:

- **Structure - "kind of" and "part of" relationships**
- **Other systems - external systems with which the application interacts**
- **Devices - devices with which the application interacts**
- **Events remembered - a historical event that must be recorded**
- **Roles played - the different roles users play in interacting with the application**
- **Locations - physical locations, offices, and sites important to the application**
- **Organizational units - groups to which users belong**

Domain Analysis

Domain Analysis **- the process of identifying the classes and objects that are common to all applications within a given domain**

Contrast Domain Analysis to Object-Oriented Analysis, which focuses on one problem at a time.

Domain Analysis is useful for pointing you to the key abstractions that have proven useful in other related systems, giving the designer ideas for the abstractions pertinent in the system under design. Domain Analysis works well because there are very few truly unique kinds of software systems.

Suggested Steps in Domain Analysis

- **Construct a generic model -- consult with domain experts**
- **Examine existing systems**
- **Identify similarities and differences between the systems**
- **Refine the model**

- Construct a generic model of the domain by consulting with domain experts [a domain expert is simply a user or a person intimately familiar with the elements of a particular problem].
- Examine existing systems within the domain and represent this understanding in a common format.
- Identify similarities and differences between the systems by consulting with domain experts.
- Refine the generic model to accommodate existing systems.