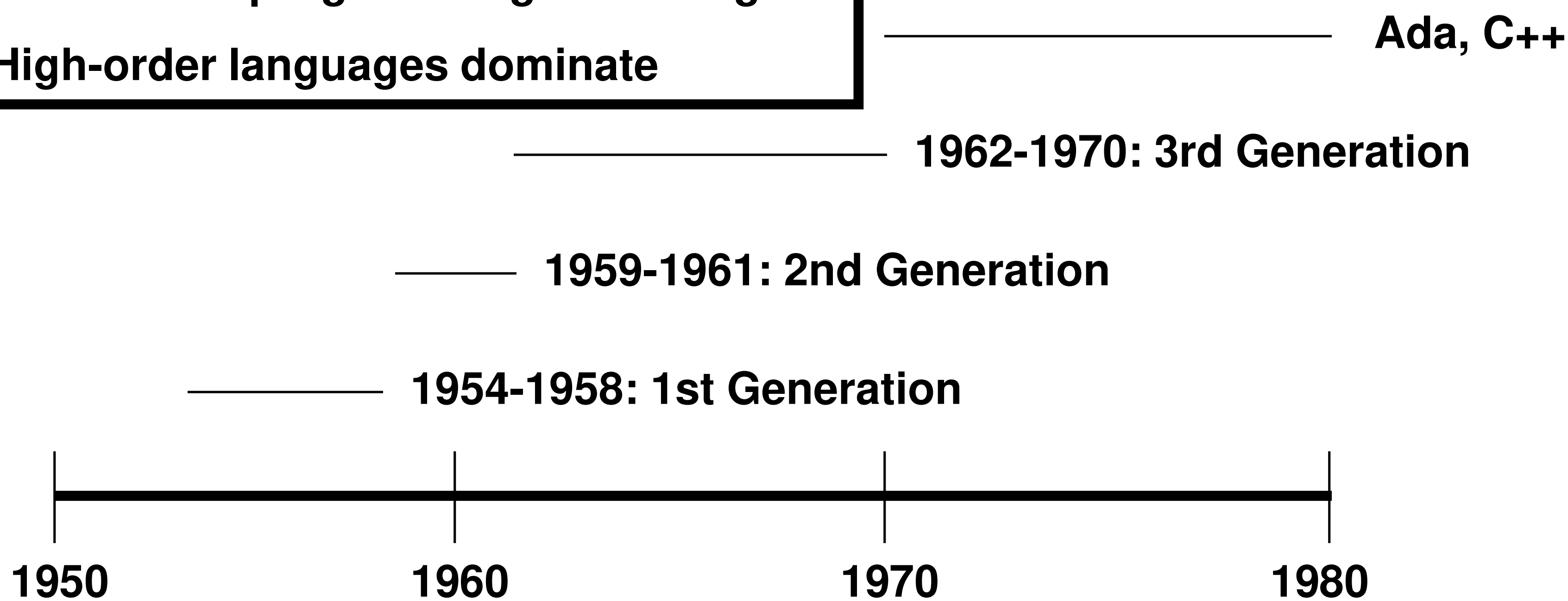
THE OBJECT MODEL

- Programming Language Generations
- What Is an Object (and What is Not an Object)?
- OOP, OOD, and OOA
- Programming Paradigms
- No Single Paradigm
- Elements of the Object Model
- Benefits and Applications of the Object Model

PROGRAMMING LANGUAGE GENERATIONS

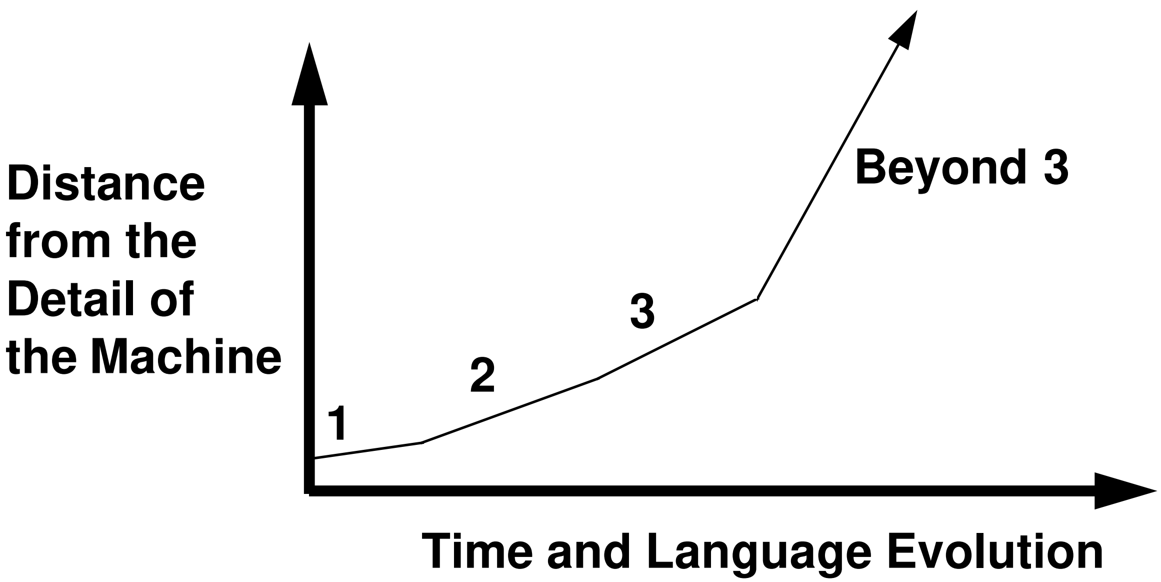
- **New Focus: programming in the large**
- **High-order languages dominate**



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

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



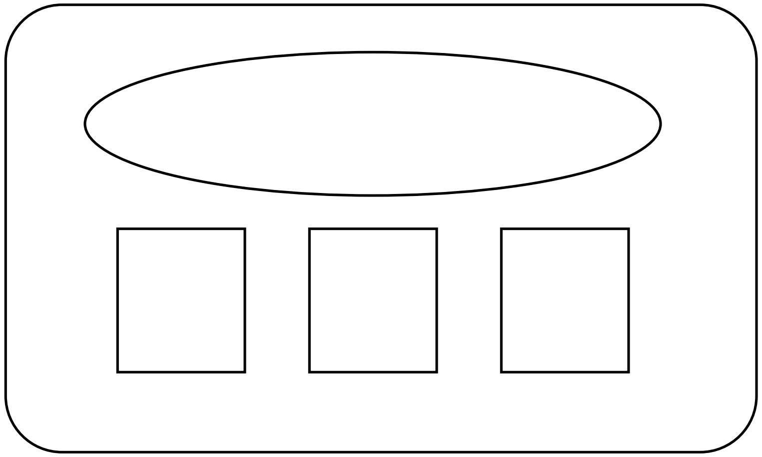
Heritage of Some Key Languages

Lang	*Parents*
Ada	ALGOL 68, Pascal, Simula, Alphard, CLU, about 20 others
CLOS	Lisp, LOOPS, Flavors
C++	C, Simula

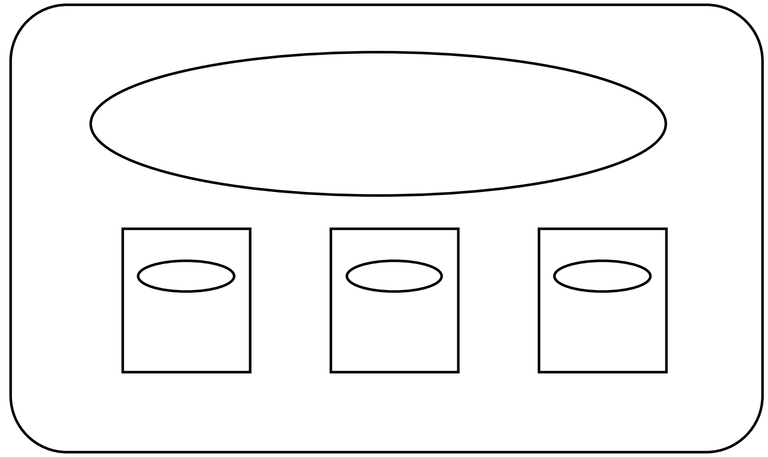
Perspective of Some Key Languages

Lang	*Perspective*
Ada	Object-based
CLOS	Object-oriented
C++	Object-oriented

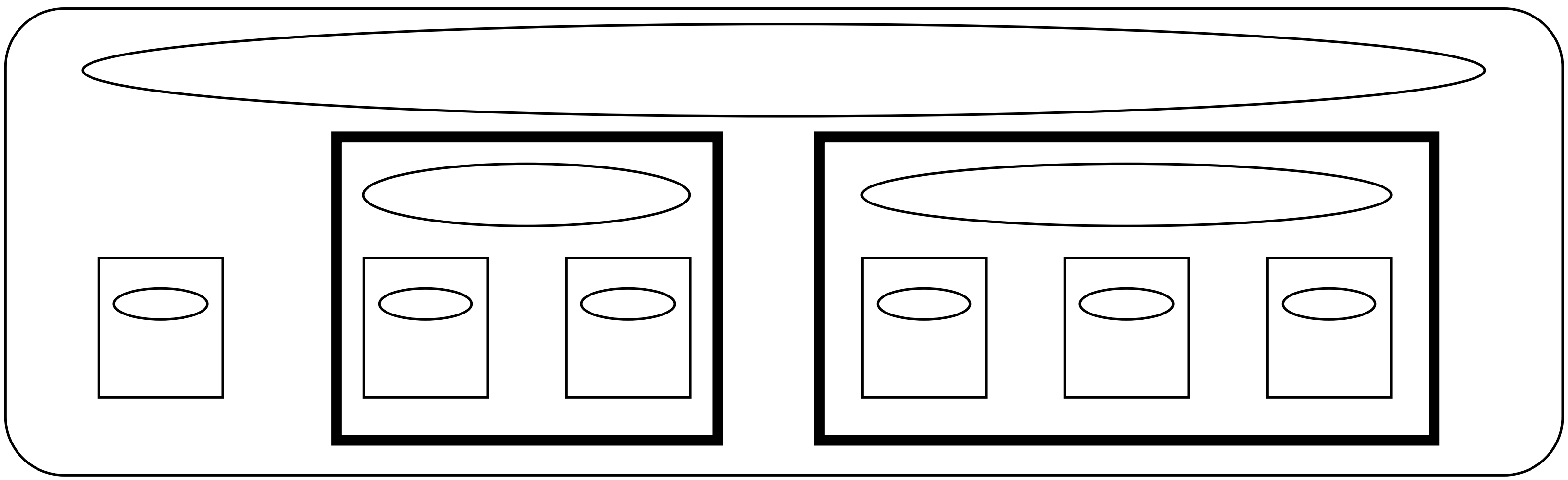
Topologies of Languages by Generation



1st Generation



2nd Generation



3rd Generation and Beyond

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?**

WHAT IS AN OBJECT (AND WHAT IS NOT AN OBJECT)?

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**

OOP, OOD, and OOA
Definitions

- ***Object-Oriented Programming*** **(OOP) - a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships**
- ***Object-Oriented Design*** **(OOD) - a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design**
- ***Object-Oriented Analysis*** **(OOA or OORA) - a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain**

-- Grady Booch, *Object-Oriented Design with Applications*, 1991, Pp 36-37

Object-Oriented Programming

Object-Oriented Programming **(OOP) - 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**

Key parts of this definition:

- **OOP uses objects, not algorithms, as the fundamental building blocks.**
- **Each object is a member of some class.**
- **Classes are related to one another via inheritance relationships.**

Note: A language is *object-oriented* if it supports all the key parts of the definition of OOP. A language is *object-based* if it supports all the key parts except inheritance.

Object-Oriented Design

Object-Oriented Design **(OOD) - 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**

Key parts of this definition:

- **OOD leads to an object-oriented decomposition.**
- **OOD uses different notations to express different models of the logical (class and object structure) and physical (module and process architecture) design of a system.**

Note: OOD refers to any method that leads to an object-oriented decomposition. Object-oriented decomposition is what makes OOD different from structured design.

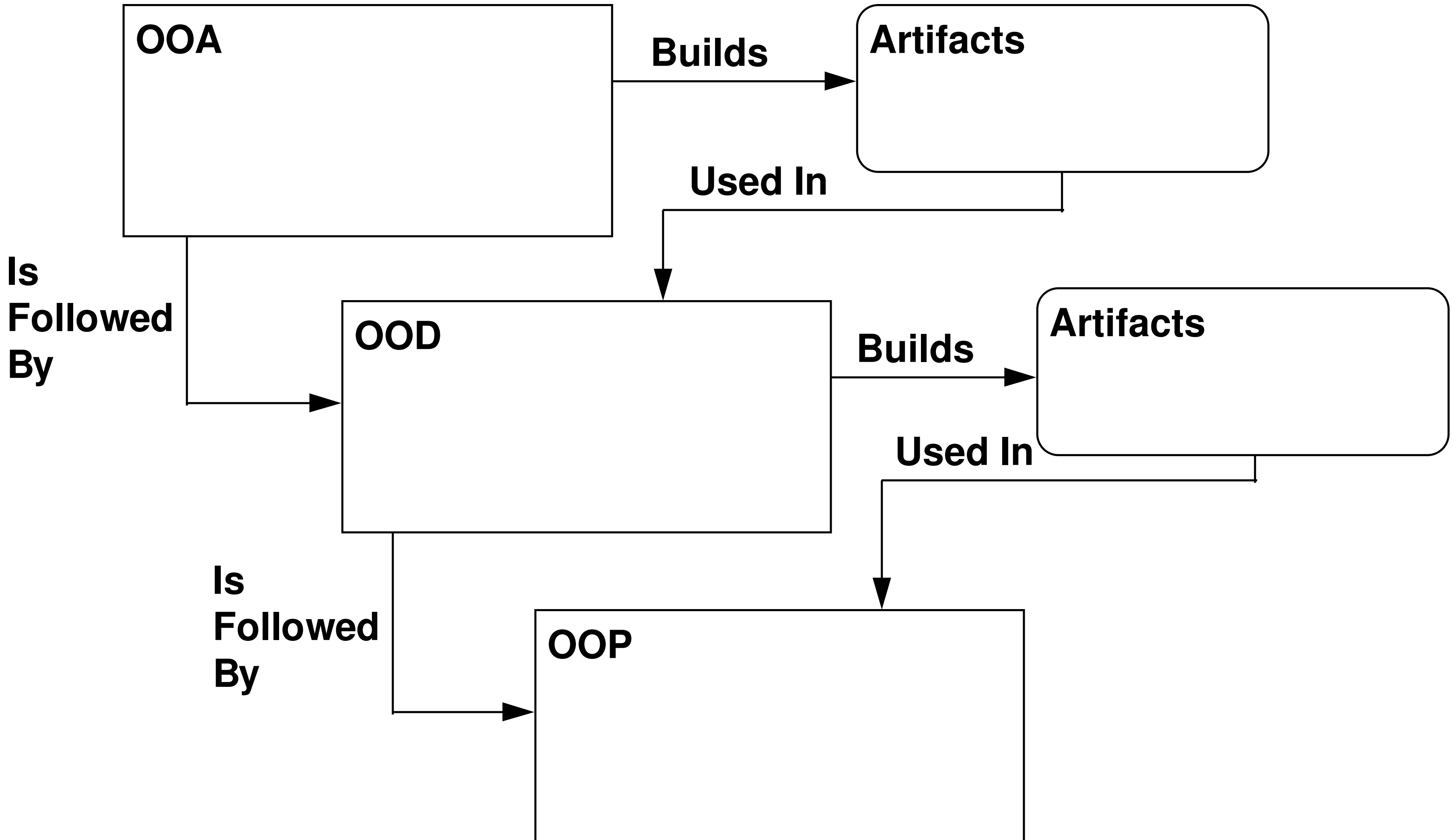
Object-Oriented Analysis

Object-Oriented Analysis **(OOA or OORA) - a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain**

Key parts of this definition:

- **OOA uses the vocabulary of the problem domain, thereby forming real-world models of the problem.**

How are OOP, OOD, and OOA Related?



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**

NO SINGLE PARADIGM

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. 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.

Major Elements

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

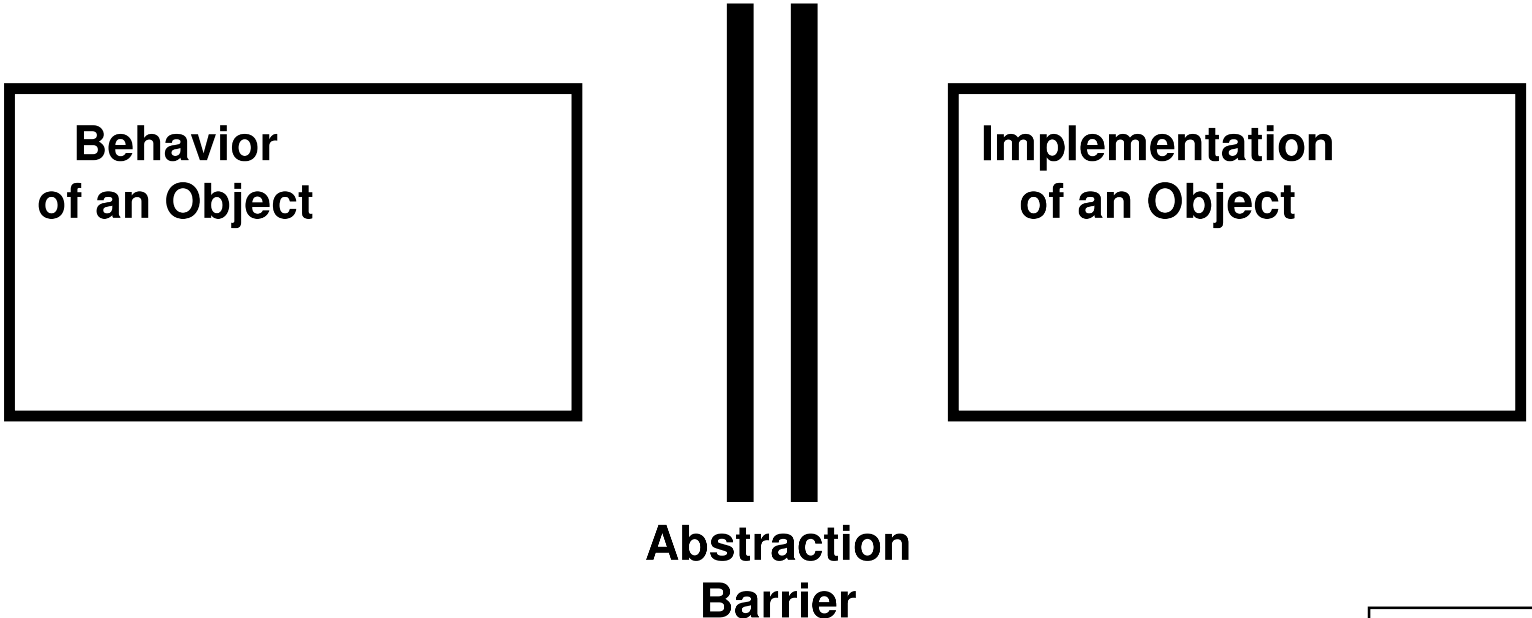
Minor Elements

- ✓ **Typing**
- ✓ **Concurrency**
- ✓ **Persistence**

Abstraction

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 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

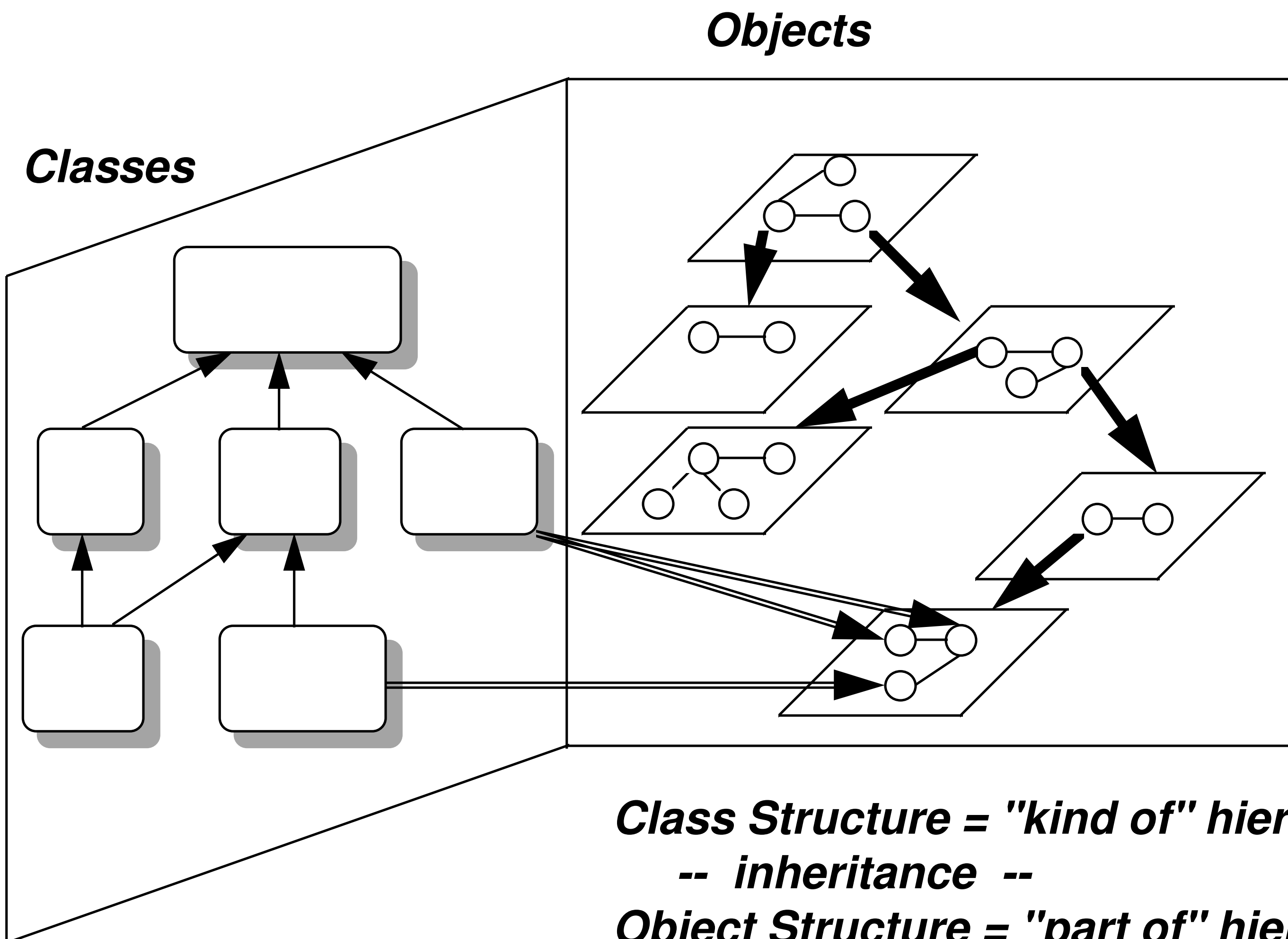
Hierarchy **- the ranking or ordering of abstractions**

-- 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 --

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

- 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

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.

Monomorphism is the opposite of polymorphism, so a monomorphic object may only respond to 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

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.

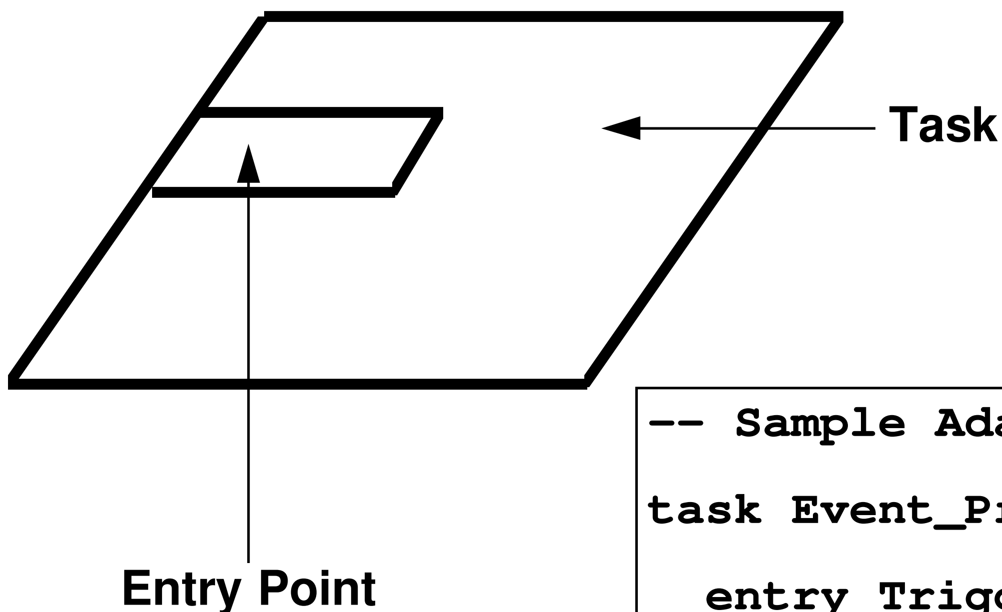
An object is an excellent candidate for a concurrent entity because:

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

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**

BENEFITS AND APPLICATIONS OF THE OBJECT MODEL

Benefits --

- **The Object Model leads us to construct systems that embody the five attributes of well-structured complex systems.**
- **The Object Model helps us exploit the expressive power of all object-based and object-oriented programming languages.**
- **The use of the Object Model encourages reuse of both code and designs.**
- **The use of the Object Model produces systems that are built upon stable intermediate forms, thereby being more resilient to change. Such systems can be allowed to evolve over time.**
- **The Object Model reduces the risk of developing complex systems.**
- **The Object Model appeals to the workings of human cognition.**

BENEFITS AND APPLICATIONS

Selected Applications --

- Air traffic control
- Animation
- Avionics
- Banking and insurance software
- Chemical process control
- Command and control systems
- Computer-aided design
- Computer-aided education
- Computer integrated manufacturing
- Databases
- Expert systems
- Film and stage storyboarding
- Hypermedia
- Image recognition
- Investment strategies
- Mathematical analysis
- Medical electronics
- Music composition
- Office automation
- Operating systems
- Reusable software components
- Robotics
- Software Development Environments
- Space station software
- Spacecraft and aircraft simulation
- Telecommunications and telemetry
- User interface design
- VLSI design