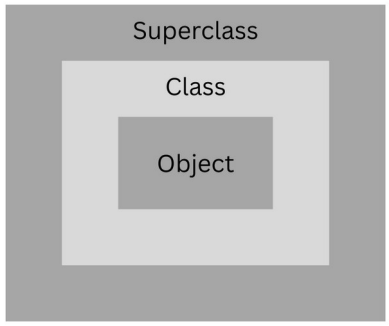
Object-Oriented Programming What then, is object-oriented programming (or OOP, as it is sometimes written)? We define it as follows:

> *Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.*

There are three important parts to this definition: object-oriented programming (1) uses *objects*, not algorithms, as its fundamental logical building blocks (the "part of" hierarchy we introduced in Chapter l); (2) each object is an instance of some class; and (3) classes are related to one another via inheritance relationships (the "is a" hierarchy we spoke of in Chapter l). A program may appear to be object-oriented, but if any of these elements is missing, it is not an object-oriented program. Specifically, programming without inheritance 'is distinctly not object-oriented; we call it *programming with abstract data types*.

By this definition, some languages are object-oriented, and some are not. Stroustrup suggests that: "if the term 'object-oriented language' means anything, it must mean a language that has mechanisms that support the object-oriented style of programming well.... A language supports a programming style well if it provides facilities that make it convenient to use that style. A language does not support a technique if it takes exceptional effort or skill to write such programs; in that case, the language merely enables programmers to use the techniques" [33]. From a theoretical perspective, one can fake object oriented programming in non-object-oriented programming languages like Pascal and even COBOL or assembly language, but it is horribly ungainly to do so. Cardelli and Wegner thus say "that a language is object-oriented if and only if it satisfies the following requirements:

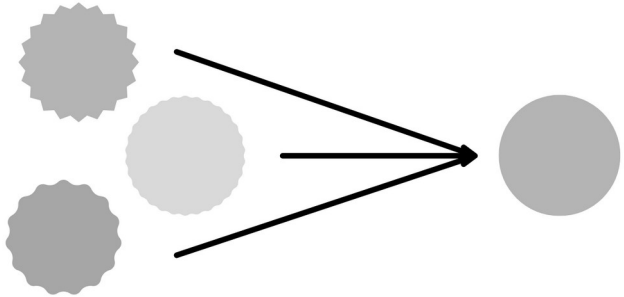
- It supports objects that are data abstractions with an interface of named operations and a hidden local state.
- Objects have an associated type [class].
- Types [classes] may inherit attributes from supertypes [superclasses]" [34].



The Meaning of Abstraction Abstraction is one of the fundamental ways that we as humans cope with complexity. Hoare suggests that "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" [42]. Shaw defines an abstraction as "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" [43]. Berzins, Gray, and Naumann recommend that ~'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", [44]. 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.*

An abstraction focuses on the outside view of an object, and so serves to separate an object's essential behavior from its implementation. Abelson and Sussman call this behavior/implementation division an *abstraction barrier* [45] achieved by applying the principle of least commitment, through which the interface of an object provides its essential behavior, and nothing more [46]. "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.



Excerpts from Booch, G. (1994). *Object-Oriented Analysis and Design*. Santa Clara, CA. Addison Wesley.

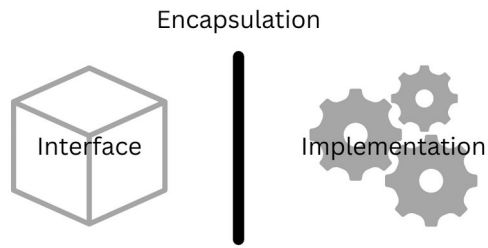
Abstraction and encapsulation are complementary concepts: abstraction focuses upon the observable behavior of an object, whereas *encapsulation* focuses upon the implementation that gives rise to this behavior. Encapsulation is most often achieved through *information biding*, which is the process of hiding all the secrets of an object that do not contribute to its essential characteristics; typically, the structure of an object is hidden, as well as the ,implementation of its methods.

Encapsulation provides explicit barriers among different abstractions and thus leads to a clear separation of concerns. For example, consider again the structure of a plant. To understand how photosynthesis works at a high level of abstraction, we can ignore details such as the responsibilities of plant roots or the chemistry of cell walls. Similarly, in designing a database application, it is standard practice to write programs so that they don't care about the physical representation of data, but depend only upon a schema that denotes the data's logical view [52]. In both of these cases, objects at one level of abstraction are shielded from implementation details at lower levels of abstraction.

Liskov goes as far as to suggest that "for abstraction to work, implementations must be encapsulated" [53]. In practice, this means that each class must have two parts: an interface and an implementation. The *interface* of a class captures only its outside view, encompassing our abstraction of the behavior common to all instances of the class. The *implementation* of a class comprises the representation of the abstraction as well as the mechanisms that achieve the desired behavior. The interface of a class is the one place where we assert all of the assumptions that a client may make about any instances of the class; the implementation encapsulates details about which no client may make assumptions.

To summarize, we define *encapsulation* as follows:

> *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.*



Excerpts from Booch, G. (1994). *Object-Oriented Analysis and Design*. Santa Clara, CA. Addison Wesley.

The Meaning of Modularity As Myers observes, "The act of partitioning a program into individual components can reduce its complexity to some degree. . . . Although partitioning a program is helpful for this reason, a more powerful justification for partitioning a program is that it creates a number of well defined, documented boundaries within the program. These boundaries, or interfaces, are invaluable in the comprehension of the program" [57]. In some languages, such as Smalltalk, there is no concept of a module, and so the class forms the only physical unit of decomposition. In many others, including Object Pascal, 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. Especially for larger applications, in which we may have many hundreds of classes, the use of modules is essential to help manage complexity.

Deciding upon the right set of modules for a given problem is almost as hard a problem as deciding upon the right set of abstractions. Zelkowitz is absolutely right when he states that "because the solution may not be known *hen the design stage starts, decomposition into smaller modules may be quite difficult. For older applications (such as compiler writing), this process may become standard, but for new ones (such as defense systems or spacecraft control), it may be quite difficult" [59].

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

The Meaning of Hierarchy Abstraction is a good thing, but in all except the most trivial applications, we may find many more different abstractions than we can comprehend at one time. Encapsulation helps manage this complexity by hiding the inside view of our abstractions. Modularity helps also, by giving us a way to cluster logically related abstractions. Still, this is not enough. A set of abstractions often forms a hierarchy, and by identifying these hierarchies in our ,design, we greatly simplify our understanding of the problem.

We define hierarchy as follows:

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

The most important hierarchies in a complex system are its class structure e "is a" hierarchy) and its object structure (the "part of' hierarchy).