• Data that exists between various versions of a program
• Data that outlives the program
Persistence is theproperty of an object tbrougb which its existence transcends time (i.e. tbe object continues to exist after its creator ceases to exist) and/or space (i. e. the objects location moves from the address space in wbich it was created).

Applying the Object Model

Benefits of the Object Model

As we have shown, the object model is fundamentally different from the models embraced by
the more traditional methods of structured analysis, structured design, and structured programming. This does not mean that the object model abandons all of the sound principles and experiences of these older methods. Rather, it introduces several novel elements that build upon these earlier models. Thus, the object model offers a number of significant benefits that other models simply do not provide. Most importantly, the use of the object model leads us to construct systems that embody the five attributes of well-structured complex systems. In our experience, there are five other practical benefits to be derived from the application of the object model.

Applications of the Object Model

The object model has proven applicable to a wide variety of problem domains. many of the domains for which systems exist that may properly be called object-oriented. The Bibliography provides an extensive list of references to these and other applications. Object-oriented analysis and design may be the only method we have today that can be employed to attack the complexity inherent in very large systems. In all fairness, however, the use of object-oriented development may be ill-advised for some domains, not for any technical reasons, but for nontechnical ones, such as the absence of a suitably trained staff or a good development environment

Open Issues

To effectively apply the elements of the object model, we must next address several open issues:
• What exactly are classes and objects?
• How does one properly identify the classes and objects that are relevant to a particular application?
• What is a suitable notation for expressing the design of an object-oriented system?
• What process can lead us to a weil-structured object-oriented system?
• What are the management implications of using object-oriented design?

UNIT-II

CLASS AND OBJECTS

Nature of an Object

The ability to recognize physical objects is a skill that humans learn at a very early age. A brightly colored ball will attract an infant's attention, but typically, if you hide the ball, the child will not try to look for it; when the object leaves her field of vision, as far as she can determine, it ceases to exist. It is not until near the age of one that a child normally develops what is called the object concept, a skill that is of critical importance to future cognitive development. Show a ball to a one-year-old and then hide it, and she will usually search for it
even though it is not visible.
From the perspective of human cognition, an object is any of the following:
• A tangible and/or visible thing
• Something that may be apprehended intellectually
• Something toward which thought or action is directed
An object 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

State

Semantics Consider a vending machine that dispenses soft drinks. The usual behavior of such objects is that when one puts coins in a slot and pushes a button to make a selection, a drink emerges from the machine. What happens if a user first makes a selection and then puts money in the slot? Most vending machines just sit and do nothing, because the user has violated the basic assumptions of their operation. Stated another way, the vending machine was playing a role (of waiting for coins) that the user ignored (by making a selection first). Similarly, suppose that the user ignores the warning light that says "Correct change only," and puts in extra money. Most machines are use rhostile; they will happily swallow the excess coins.
The state of an object encompasses all of the (usually static) properties of to be object plus to be current (usually dynamic) values of each of these properties.

Behavior

The Meaning of Behavior No object exists in isolation. Rather, objects are acted upon, and
themselves act upon other objects. Thus, we may say that
Behavior is how an object acts and reacts, in terms of its state changes and message passing.
In other words, the behavior of an object represents its outwardly visible and testable activity

Identity

Semantics Khoshafian and Copeland offer the following definition:
"Identity is that property of an object which distinguishes it from all other objects "
They go on to note that "most programming and database languages use variable names to distinguish temporary objects, mixing addressability and identity. Most database systems use identifier keys to distinguish persistent objects, mixing data value and identity." The failure to recognize the difference between the name of an object and the object itself is the source of many kinds of errors in object-oriented programming.

Relationships Among Objects

Kinds of Relationships

An object by itself is intensely uninteresting. Objects contribute to the behavior of a system by collaborating with one another. As Ingalls suggests, "Instead of a bit-grinding processor raping and plundering data structures, we have a universe of well-behaved objects that courteously ask each other to carry out their various desires" . For example, consider the object structure of an airplane, which has been defined as "a collection of parts having an inherent tendency to fall to earth, and requiring constant effort and supervision to stave off that outcome" . Only the collaborative efforts of all the component objects of an airplane enable it to fly. The relationship between any two objects encompasses the assumptions that each makes about the other, including what operations can be performed and what behavior results. We have found that two kinds of object hierarchies are of particular interest in object-oriented analysis and design, namely:
• Links
• Aggregation

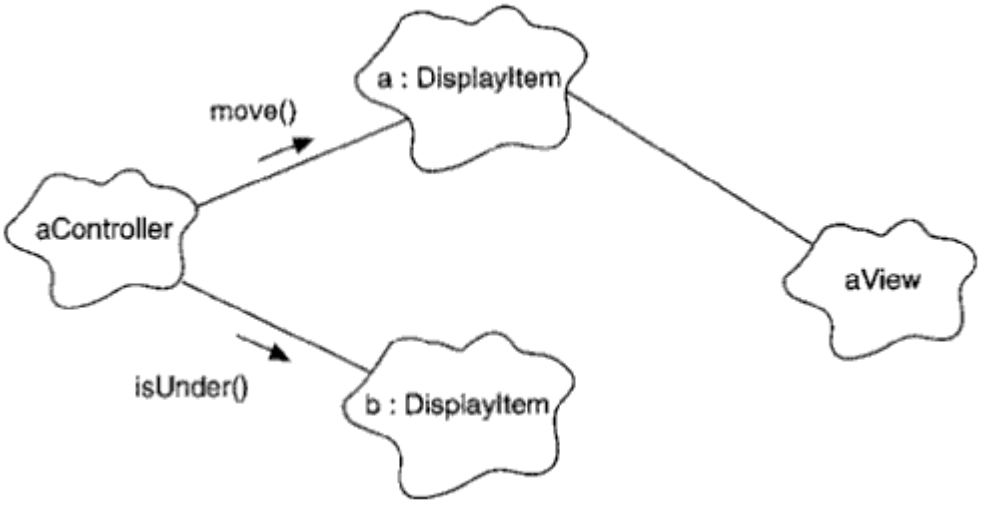


Figure 3-2
Links

Links

Semantics The term *link* derives from Rumbaugh, who defines it as a "physical or conceptual

connection between objects". An object collaborates with other objects through its links to these objects. Stated another way, a link denotes the specific association through which one
object (the client) applies the services of another object (the supplier), or through which oneobject may navigate to another. Figure 3-2 illustrates several different links. In this figure, a line between two object icons represents the existence of a link between the two and means that messages may pass along this path. Messages are shown as directed lines representing the direction of the message, with a label naming the message itself. For example, here we see that the object **aController** has links to two instances of **DisplayItem** (the objects **a** and **b**). Although both **a** and **b** probably have links to the view in which they are shown, we have chosen to highlight only once such link, from **a** to **aView**. Only across these links may one object send messages to another.

As a participant in a link, an object may play one of three roles:
• Actor An object that can operate upon other objects but is never operated upon by other objects; in some contexts, the terms *active object* and *actor* are interchangeable
• Server An object that never operates upon other objects; it is only operated upon by other objects
• 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

Aggregation

Semantics Whereas links denote peer-to-peer or client/supplier relationships, aggregation denotes a whole/part hierarchy, with the ability to navigate from the whole (also called the *aggregate*) to its parts (also known as its *attributes*). In this sense, aggregation is a specialized kind of association. For example, as shown in Figure 3-3, the object **rampController** has a link to the object **growingRamp** as well as an attribute **h** whose class is **Heater**. The object **rampController** is thus the whole, and **h** is one of its parts. In other words, **h** is a part of the state of the object **rampController**. Given the object **rampController**, it is possible to find its corresponding heater **h**. Given an object such as **h**, it is possible to navigate to its enclosing object (also called its *container*) if and only if this knowledge is a part of the state of **h**. Aggregation may or may not denote physical containment. For example, an airplane is composed of wings, engines, landing gear, and so on: this is a case of physical containment. On the other hand, the relationship between a shareholder and her shares is an aggregation relationship that does not require physical containment. The shareholder uniquely owns shares, but the shares are by no means a physical part of the shareholder. Rather, this whole/part relationship is more conceptual and therefore less direct than the physical aggregation of the parts that form an airplane.

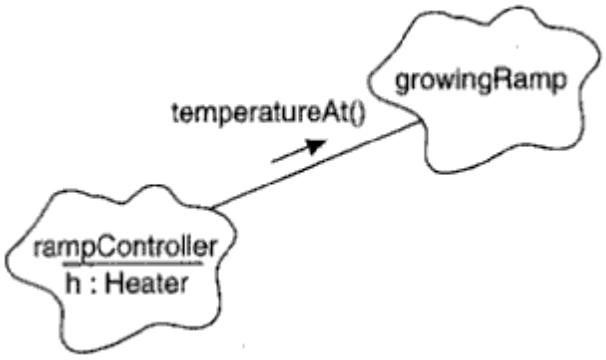


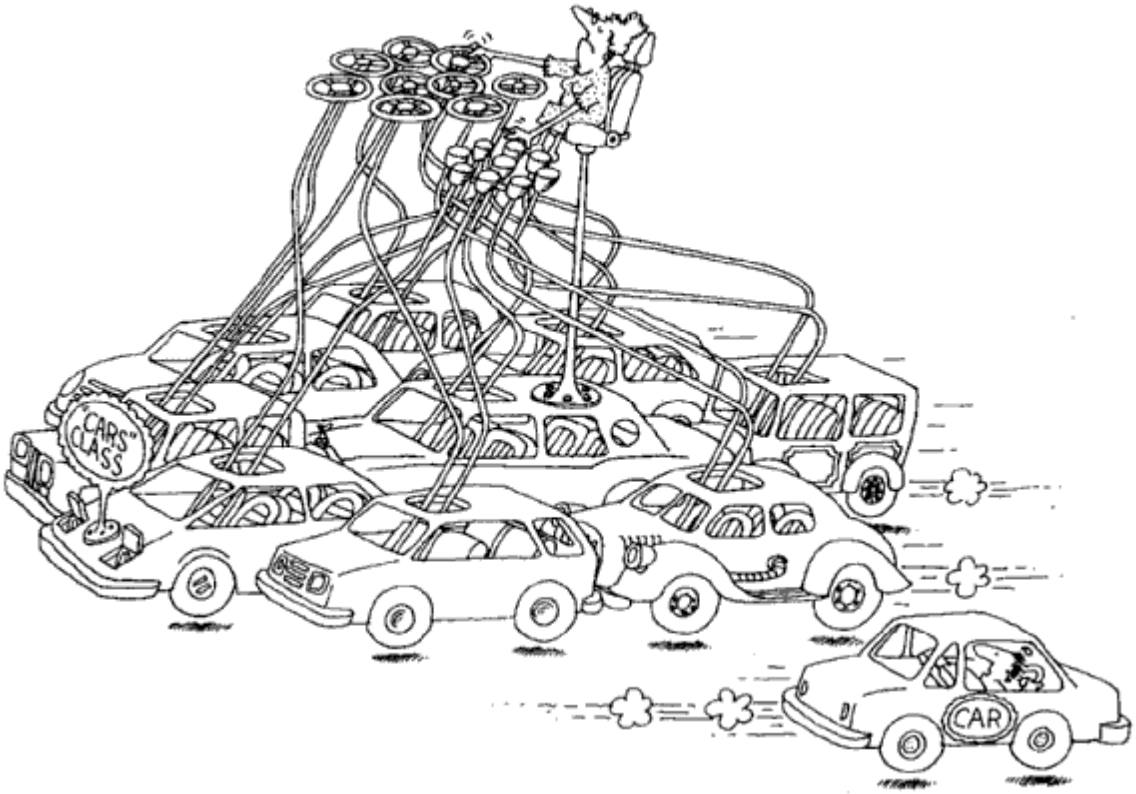
Figure 3-3
Aggregation

Nature of a Class

The concepts of a class and an object are tightly interwoven, for we cannot talk about an object without regard for its class. However, there are imiportant differences between these two terms. Whereas an object is a concrete entity that exists in time and space, a class represents only an abstraction, the “essence" of an object, as it were. Thus, we may speak of the class **Mammal**, which represents the characteristics common to all mammals. To identify a particular mammal in this class, we must speak of "this mammal” or "that mammal."

A class is a set of objects that share a common structure and a common behavior.

A single object is simply an instance of a class.



A class represents a set of objects that share a common structure and a common behavior.

Interface and Implementation

Meyer and Snyder have both suggested that programming is largely a matter of "contracting": the various functions of a larger problem are decomposed into smaller problems by subcontracting them to different elements of the design. Nowhere is this idea more evident than in the design of classes. Whereas an individual object is a concrete entity that performs some role in the overall system, the class captures the structure and behavior common to all related objects. Thus, a class serves as a sort of binding contract between an abstraction and all of its clients. By capturing these decisions in the interface of a class, a strongly typed programming language can detect violations of this contract during compilation.

We can further divide the interface of a class into three parts:

• Public

A declaration that is accessible to all clients

• Protected

A declaration that is accessible only to the class itself, its subclasses, and its friends

• Private

A declaration that is accessible only to the class itself and its friends

Class Life Cycle

We may come to understand the behavior of a simple class just by understanding the semantics of its distinct public operations in isolation. However, the behavior of more interesting classes (such as moving an instance of the class **DisplayItem,** or scheduling an instance of the class **TemperatureController**) involves the interaction of their various operations over the lifetime of each of their instances.

Relationships Among Classes

Consider for a moment the similarities and differences among the following classes of objects:

flowers, daisies, red roses, yellow roses, petals, and ladybugs. We can make the following observations:

• A daisy is a kind of flower.

• A rose is a (different) kind of flower.

• Red roses and yellow roses are both kinds of roses.

• A petal is a part of both kinds of flowers.

• Ladybugs eat certain pests such as aphids, which may be infesting certain kinds of flowers.

. Specifically, most object-oriented languages provide direct support for some combination of the following relationships:

• Association

• Inheritance
• Aggregation
• Using
• Instantiation
• Metaclass

Association

Example In an automated system for retail point of sale, two of our key abstractions include products and sales. As shown in Figure 3-4, we may show a simple association between these two classes: the class **Product** denotes the products sold as part of a sale, and the class **Sale** denotes the transaction through which several products were last sold. By implication, this association suggests bidirectional navigation: given an instance of **Product**, we should be able to locate the object denoting its sale, and given an instance of **Sale**, we should be able to locate
all the products sold during the transaction.

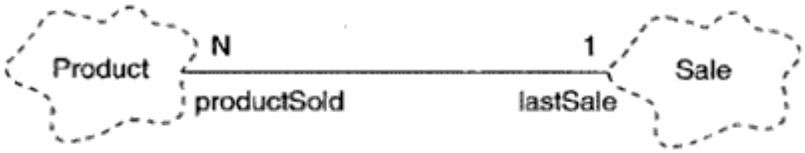


Figure 3-4 **Association**

Inheritance

Examples After space probes are launched, they report back to ground stations with information regarding the status of important subsystems (such as electrical power and propulsion systems) and different sensors (such as radiation sensors, mass spectrometers, cameras, micro meteorite collision detectors, and so on). Collectively, this relayed information is called *telemetry data*. Telemetry data is commonly transmitted as a bit stream consisting of a header, which includes a time stamp and some keys identifying the kind of information that follows, plus several frames of processed data from the various subsystems and sensors. Because this appears to be a straightforward aggregation of different kinds of data, we might be tempted to define a record type for each kind of telemetry data. For example, in C++, we might write

A subdass may inherit the structure and behavior of its superdass.

Single Inheritance Simply stated, inheritance is a relationship among classes wherein one class shares the structure and/or behavior defined in one (*single inheritance*) or more (*multiple inheritance*) other classes. We call the class from which another class inherits its superclass. In our example, **TelemetryData** is a superclass of **ElectricalData.** Similarly, we call a class that inherits from one or more classes a subclass; **ElectricalData** is a subclass of **TelemetryData.** Inheritance therefore defines an "is a" hierarchy among classes, in which a subclass inherits from one or more superclasses. This is in fact the litmus test for inheritance given classes A and B, if A "is not a" kind of B, then A should not be a subclass of B. In this sense, **ElectricalData** is a specialized kind of the more generalized class **TelemetryData.** The ability of a language to support this kind of inheritance distinguishes object-oriented from object-based programming languages.

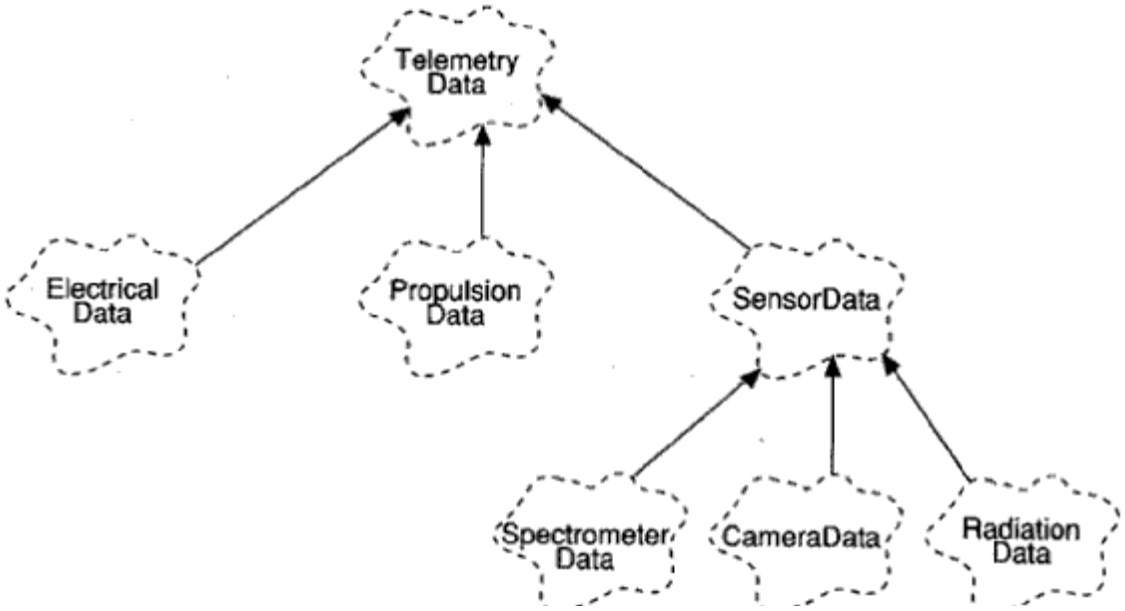


Figure 3-5 Single Inheritance

Multiple Inheritance With single inheritance, each subclass has exactly one superclass. However, as Vlissides and Linton point out, although single inheritance is very useful, "it often forces the programmer to derive from one of two equally attractive classes. This limits the applicability of predefined classes, often making it necessary to duplicate code. For example, there is no way to derive a graphic that is both a circle and a picture; one must derive from one or the other and reimplement the functionality of the class that was excluded" [40]. Multiple inheritance is supported directly by languages such as C++ and CLOS and, to a limited degree, by Smalltalk. The need for multiple inheritance in objectoriented programming languages is still a topic of great debate. In our experience, we find multiple inheritance to be like a parachute: you don't aIways need it, but when you do, you're really happy to have it on hand.

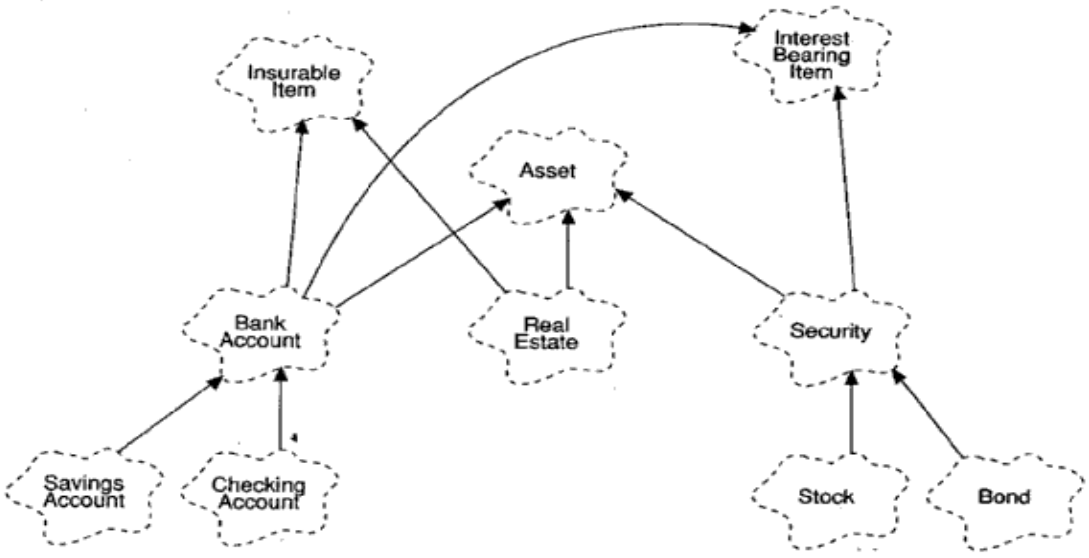


Figure 3-7 Multiple Inheritance

Aggregation

Example Aggregation relationships among classes have a direct parallel to aggregation relationships among the objects corresponding to these classes.

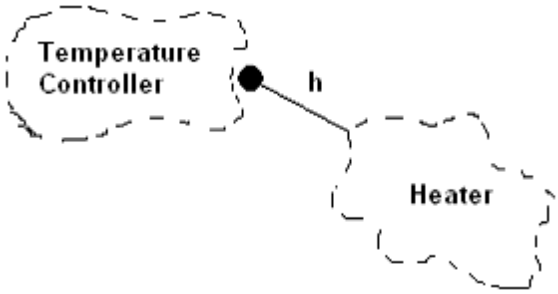


Figure 3-8 **Aggregation**

Using

Example Our carlier example of the **rampController** and **growingRamp** objects illustrated a link between the two objects, which we represented via a "using" relationship between their corresponding classes, **TemperatureController** and **TemperatureRamp**:

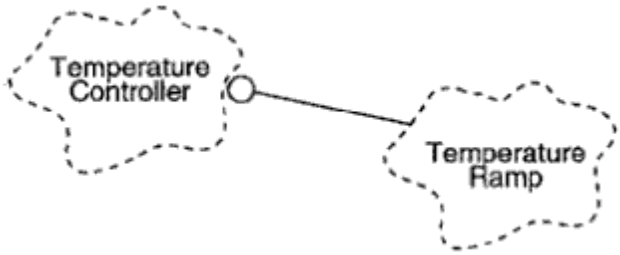


Figure 3-9 **The "Using" Relationship**

Instantiation

Examples Our earlier declaration of the class **Queue** was not very satisfying because its abstraction was not type-safe. We can vastly improve our abstraction by using languages such as Ada, C++, and Eiffel that support genericity.

For example, we might rewrite our earlier class declaration using a parameterized class in C++:

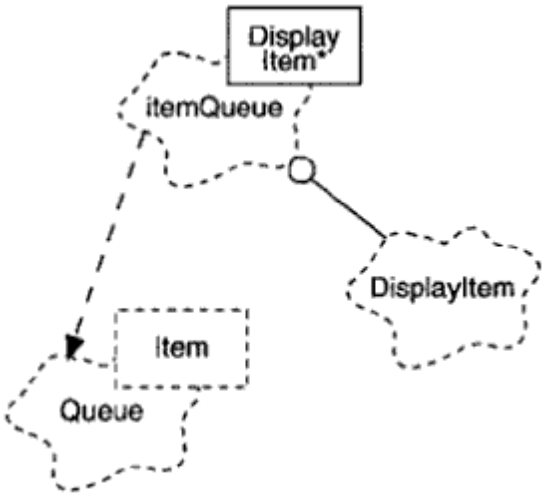


Figure 3-10 **Instantiation**

Metaclass

We have said that every object is an instance of some class. What if we treat a class itself as an object that can be manipulated? To do so, we must ask, What is the class of a class? The answer is simply, a metaclass. To state it another way, a *metaclass* is a class whose instances are themselves classes. Languages such as Smalltalk and CLOS support the concept of a metaclass directly; C++ does not. Indeed, the idea of a metaclass takes the idea of the object model to its natural completion in pure object-oriented programming languages.

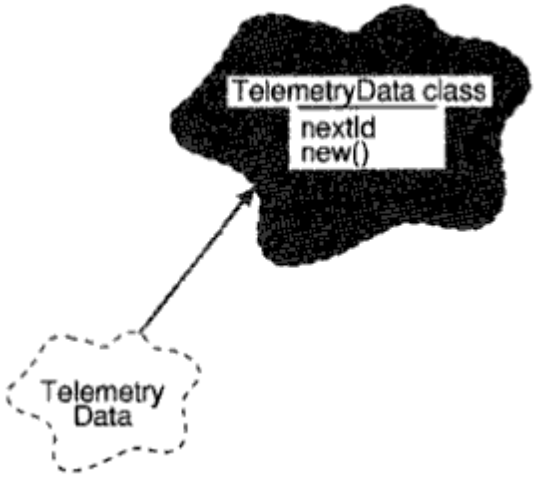


Figure 3-11 **Metaciasses**

Interplay of Classes and Objects

Relationships Between Classes and Obiects

Clases and object are separate yet intimately related concepts. Specifically, every object is the

instance of some class, and every class has zero or more instances. For practically all applications, classes are static; therefore, their existence, semantics, and relationships are fixed prior to the execution of a program. Similarly, the class of most objects is static, meaning that once an object is created, its class is fixed. In sharp contrast, however, objects are typically created and destroyed at a furious rate during the li time of an application. For example, consider the classes and objects in the implementation of an traffic control system. Some of the more important abstractions include planes, flight plans, runways, and air spaces. By their very definition, the meanings of these classes of objects are relatively static. They must be static, for otherwise one could not build an application that embodied knowIedge of such commonsense facts as that planes can take off, fly, and then land, and that two planes should not occupy the same space at the same time. Conversely, the instances of these classes are dynamic. At a fairly slow rate, new runways are built, and old ones are deactivated. Faster yet, new flight plans are filed, and old ones are filed away. With great frequency, new planes enter a particular air space, and old ones leave.

Role of Classes and Objects in Analysis and Design

During analysis and the early stages of design, the developer has two primary tasks:

• Identify the classes and objects that form the vocabulary of the problem domain.

• Invent the structures whereby sets of objects work together to provide the behaviors that satisfy the requirements of the problem.

Collectively, we call such classes and objects the *key abstractions* of the problem, and we call these cooperative structures the *mechanisms* of the implementation. During these phases of development, the focus of the developer must be upon the outside view of these key abstractions and mechanisms. This view represents the logical framework of the system, and therefore encompasses the class structure and object structure of the system. In the later stages of design and then moving into implementation, the task of the developer changes: the focus is on the inside view of these key abstractions and mechanisms, involving their physical representation. We may express these design decisions as part of the system's module architecture and process architecture.

The Importance of Proper Classification

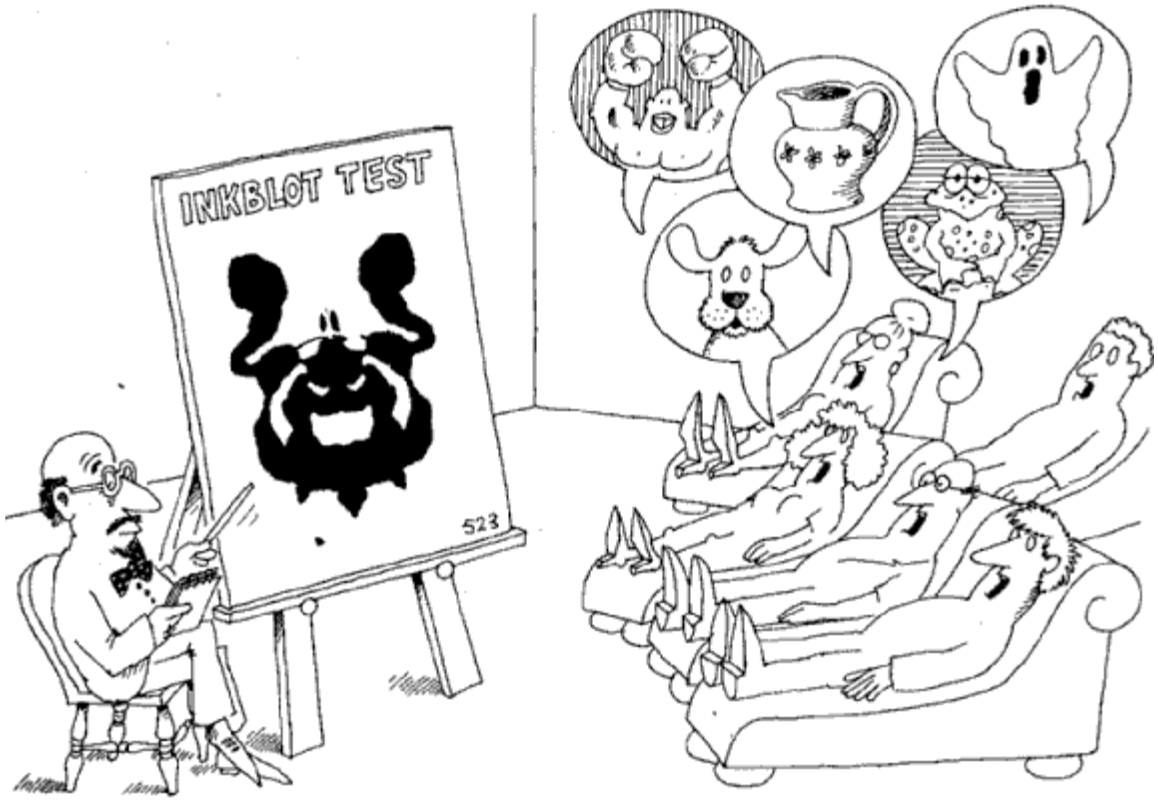
Classification and Object-Oriented Development

The identification of classes and objects is the hardest part of object-oriented analysis and design. Our experience shows that identification involve both discovery and invention. Through discovery, we come to recognize the key abstractions and mechanisms that form the vocabulary of our problem domain. Through invention, we devise generalized abstractions as well as new mechanisms that specify how objects collaborate. Ultimately, discovery and invention are both problems of classification, and classification is fundamentally a problem of finding sameness. When we classify, we seek to group things that have a common structure or exhibit a common behavior. Intelligent classification is actually a part of all good science. As Michalski and Stepp observe, "An omnipresent problem in science is to construct meaningful classifications of observed objects or situations. Such classifications facilitate human comprehension of the observations and the subsequent development of a scientific theory". The same philosophy applies to engineering. In the domain of building architecture and city planning, Alexander notes that, for the architect, "his act of design, whether humble, or gigantically complex, is governed entirely by the patterns he has in his mind at that moment, and his ability to combine these patterns to form a new design" [3]. Not surprisingly, then, classification is relevant to every aspect of object-oriented design. Classification helps us to identify generalization, specialization, and aggregation hierarchies among classes. By recognizing the common pattems of interaction among objects, we come to invent the mechanisms that serve as the soul of our implementation. Classification also guides us in making decisions about modularization. We may choose to place certain classes and objects together in the same module or in different modules, depending upon the sameness we find among these declarations; coupling and cohesion are simply measures of this sameness. Classification also plays a role in allocating processes to processors. We place certain processes together in the same processor or different processors, depending upon packaging, performance, or reliability concerns.

The Difficulty of Classification

Examples of Classification In the previous chapter, we defined an object as something that has a crisply defined boundary. However, the boundaries that distinguish one object from another are often quite fuzzy. For example, look at your leg. Where does your knee begin, and where does it end? In recognizing human speech, how do we know that certain sounds connect to form a word, and are not instead a part of any surrounding words? Consider also the design of a word processing system. Do characters constitute a class, or are whole words a better choice? How do we treat arbitrary, noncontiguous selections of text? Also, what about sentences, paragraphs, or even whole documents: are these classes of objects relevant to our problem? The fact that intelligent classification is difficult is hardly new information. Since there are parallels to the same problems in object-oriented design, consider for a moment the problems of classification in two other scientific disciplines: biology and chemistry. Until the eighteenth century, the prevailing scientific thought was that all living organisms could be arranged from the most simple to the most complex, with the measure of complexity being highly subjective (not surprisingly, humans were usually placed at the top of this list). In the mid-1700s, however, the Swedish botanist Carolus Lirmaeus suggested a more detailed taxonomy for categorizing organisms, according to what he called *genus* and *species*. A century later, Darwin proposed the theory that natural selection was the mechanism of evolution, whereby present-day species evolved from older ones. Darwin's theory depended upon an intelligent classification of species. As Darwin himself states, naturalists "try to arrange the species, genera, and families in each class, on what is called the natural system. But what is meant by this system? Some authors look at it merely as a scheme for arranging together those living objects which are most alike, and for separating those which are most unlike" [4]. In contemporary biology, classification denotes "the establishment of a hierarchical system of categories on the basis of presumed natural relationships among organisms" [5]. The most general category in a biological taxonomy is the kingdom, followed in order of increasing specialization, by phylum, subphylum, class, order, family, genus, and, finally, species. Historically, a particular organism is placed in a specific category according to its body structure, internal structural characteristics, and evolutionary relationships.

The Incremental and lterative Nature of Classification We have not said all this to defend lengthy software development schedules, although to the manager or end user, it does sometimes seem that software engineers need centuries to complete their work. Rather, we have told these stories to point out that intelligent classification is intellectually hard work, and that it best comes about through an incremental and iterative process. This incremental and iterative nature is evident in the development of such diverse software technologies as graphical user interfaces, database standards, and even fourth-generation languages. As Shaw has observed in software engineering, "The development of individual abstractions often follows a common pattern. First, problems are solved *ad hoc*. As experience accumulates, some solutions turn out to work better than others, and a sort of folklore is passed informally from person to person. Eventually, the useful solutions are understood more systematically, and they are codified and analyzed. This enables the development of models that support automatic implementation and theories that allow the generalization of the solution.



Different observers will classify the same object in different ways.

Identifying Classes and Objects

Classical and Modern Approaches

The problem of classification has been the concern of countless philosophers, linguists, cognitive scientists, and mathematicians, even since before the time of Plato. It is reasonable to study their experiences and apply what we learn to object-oriented design. Historically, there have only been three general approaches to classification:

- Classical categorization
- Conceptual clustering
- Prototype theory

Classical Categorization In the classical approach to categorization, "All the entities that have a given property or collection of properties in common form a category. Such properties

are necessary and sufficient to define the category" . For example, married people constitute a category: one is either married or not, and the value of this property is sufficient to decide to which group a particular person belongs. On the other hand, tall people do not form a category, unless we can agree to some absolute criteria for what distinguishes the property of tall from short. *Classical categorization* comes to us first from Plato, and then from Aristotle through his classification of plants and animals, in which he uses a technique much akin to the contemporary children's game of Twenty Questions (Is it an animal, mineral, or vegetable? Does it have fur or feathers? Can it fly? Does is smell?) . Later philosophers, most notably Aquinas, Descartes, and Locke, adopted this approach. As Aquinas stated, "We can name a thing according to the knowledge we have of its nature from its properties and effects"

Conceptual Clustering *Conceptual clustering* is a more modern variation of the classical approach, and largely derives from attempts to explain how knowledge is represented. As Stepp and Michalski state, "In this approach, classes (clusters of entities) are generated by first formulating conceptual descriptions of these classes and then classifying the entities according to the descriptions". For example, we may state a concept such as "a love song." This is a concept more than a property, for the "love songness" of any song is not something that may be measured empirically. However, if we decide that a certain song is more of a love song than not, we place it in this category. Thus, conceptual clustering represents more of a probabilistic clustering of objects.

Prototype Theory Classical categorization and conceptual clustering are sufficiently expressive to account for most of the classifications we ever need in the design of complex software systems. However, there are still some situations in which these approaches are inadequate. This leads us to the more recent approach to classification, called *prototype theory*, which derives primarily from the work of Rosch and her colleagues in the field of cognitive psychology

Object-Oriented Analysis

The boundaries between analysis and design are fuzzy, although the focus of each is quite distinct. In analysis, we seek to model the world by *discovering* the classes and objects that form the vocabulary of the problem domain, and in design, we *invent* the abstractions and mechanisms that provide the behavior that this model requires

Key Abstractions and Mechanisms

Identifying Key Abstractions

Finding Key Abstractions A *key abstraction* is a class or object that forms part of the vocabulary of the problem domain. The primary value of identifying such abstractions is that

they give boundaries to our problem; they highlight the things that are in the system and therefore relevant to our design, and suppress the things that are outside the system and therefore superfluous. The identification of key abstractions is highly domain-specific. As Goldberg states, the "appropriate choice of objects depends, of course, on the purposes to which the application will be put and the granularity of information to be manipulated". As we mentioned earlier, the identification of key abstractions involves two processes: discovery and invention. Through discovery, we come to recognize the abstractions used by domain experts; if the domain expert talks about it, then the abstraction is usually important. Through invention, we create new classes and objects that are not necessarily part of the problem domain, but are useful artifacts in the design or implementation. For example, a customer using an automated teller speaks in terms of accounts, deposits, and withdrawals; these words are part of the vocabulary of the problem domain. A developer of such a system uses these same abstractions, but must also introduce new ones, such as databases, screen managers, lists, queues, and so on. These key abstractions are artifacts of the particular design, not of the problem domain.

Refining Key Abstractions Once we identify a certain key abstraction as a candidate, we must evaluate it according to the metrics described in the previous chapter. As Stroustrup suggests, "Often this means that the programmer must focus on the questions: how are objects of this class created? can objects of this class be copied and/or destroyed? What operations can be done on such objects? If there are no good answers to such questions, the concept probably wasn't 'clean' in the first place, and it might be a good idea to think a bit more about the problem and the proposed solution instead of immediately starting to 'code around' the problems"

Identifying Mechanisms

Finding Mechanisms In the previous chapter, we used the term *mechanism* to describe any structure whereby objects collaborate to provide some behavior that satisfies a requirement of the problem. Whereas the design of a class embodies the knowledge of how individual objects behave, a mechanism is a design decision about how collections of objects cooperate. Mechanisms thus represent patterns of behavior. For example, consider a system requirement for an automobile: pushing the accelerator should cause the engine to run faster, and releasing the accelerator should cause the engine to run slower. How this actually comes about is absolutely immaterial to the driver. Any mechanism may be employed as long as it delivers the required behavior, and thus which mechanism is selected is largely a matter of design choice. More specifically, any of the following designs might be considered:

• A mechanical linkage from the accelerator to the carburetor (the most common mechanism).

• An electronic linkage from a pressure sensor below the accelerator to a computer that controls the carburetor (a drive-by-wire mechanism).

• No linkage exists; the gas tank is placed on the roof of the car, and gravity causes fuel to flow to the engine. Its rate of flow is regulated by a clip around the fuel line; pushing on the accelerator pedal eases tension on the clip, causing the fuel to flow faster (a low-cost mechanism).