

www.IntukMaterials.com Classes and Objects

The Nature of an Object

What Is and What Isn't 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. Through the object concept, a child comes to realize that objects have a permanence and identity apart from any operations upon them.

In the previous chapter, we informally defined an object as a tangible entity that exhibits some well-defined behavior. 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

We add to our informal definition the idea that an object models some part of reality and is therefore something that exists in time and space. In software, the term *object* was first

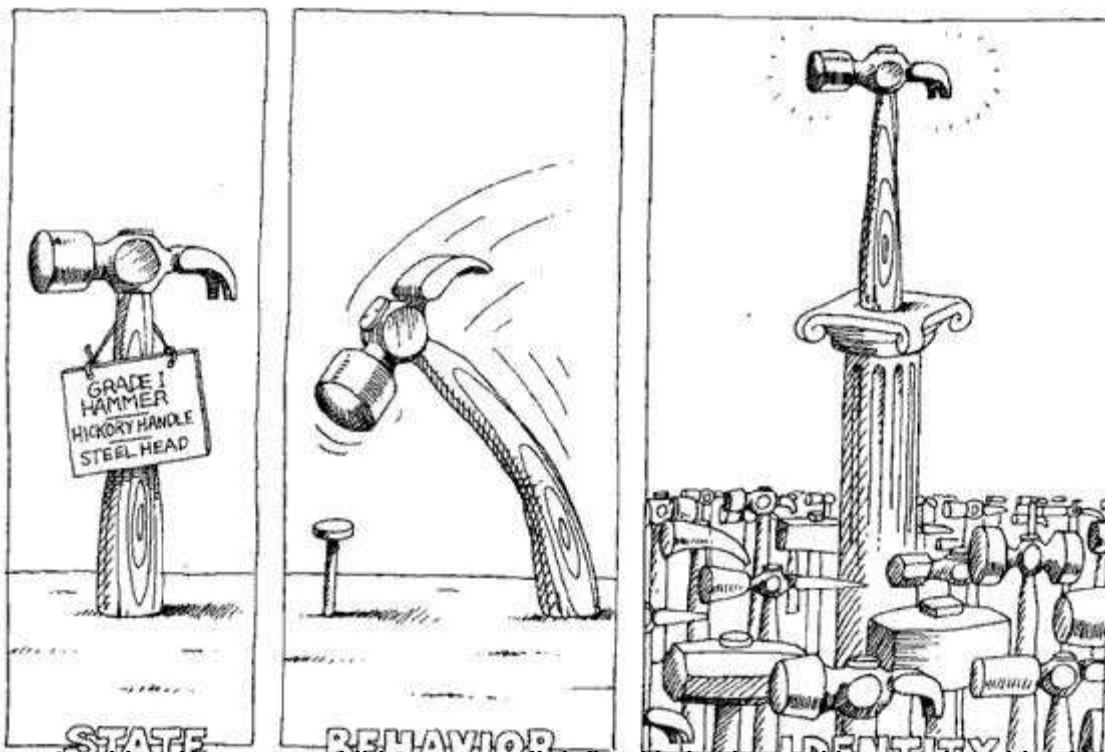
formally applied in the Simula language; objects typically existed in Simula programs to simulate some aspect of reality .

Real-world objects are not the only kinds of objects that are of interest to us during software development. Other important kinds of objects are inventions of the design process whose collaborations with other such objects serve as the mechanisms that provide some higher-level behavior . This leads us to the more refined definition of Smith and Tockey, who suggest that "an object represents an individual, identifiable item, unit, or entity, either real or abstract, with a well-defined role in the problem domain" . In even more general terms, we define an object as anything with a crisply defined boundary .

Consider for a moment a manufacturing plant that processes composite materials for making such diverse items as bicycle frames and airplane wings. Manufacturing plants are often divided into separate shops: mechanical,

chemical, electrical, and so forth. Shops are further divided into cells, and in each cell we have some collection of machines, such as die stamps, presses, and lathes. Along a manufacturing line, we might find vats containing raw materials, which are used in a chemical process to produce blocks of composite materials, and which in turn are formed and shaped to produce end items such as bicycle frames and airplane wings. Each of the tangible things we have mentioned thus far is an object. A lathe has a crisply defined boundary that separates it from the block of composite material it operates upon; a bicycle frame has a crisply defined boundary that distinguishes it from the cell of machines that produced the frame itself.

Some objects may have crisp conceptual boundaries, yet represent intangible events or processes. For example, a chemical process in a manufacturing plant may be treated as an object, because it has a crisp conceptual boundary, interacts with certain other objects through a wellordered collection of operations that unfolds over time, and exhibits a welldefined behavior. Similarly, consider a CAD/CAM system for modeling solids. Where two solids such as a sphere and a cube intersect, they may form an irregular line of intersection. Although it does not exist apart from the sphere or cube, this line is still an object with crisply defined conceptual boundaries.



An object has state, exhibits some well-defined behavior, and has a unique identity.

Some objects may be tangible, yet have fuzzy physical boundaries. Objects such as rivers, fog, and crowds of people fit this definition¹³. Just as the person holding a hammer tends to see everything in the world as a nail, so the developer with an object-oriented mindset begins to think that everything in the world is an object. This perspective is a little naive, because there are some things that are distinctly not objects. For example, attributes such as time, beauty, or color are not objects, nor are emotions such as love and anger. On the other hand, these things are all potentially properties of other objects. For example, we might say that a man (an object) loves his wife (another object), or that a particular cat (yet another object) is gray.

Thus, it is useful to say that an object is something that has crisply defined boundaries, but this is not enough to guide us in distinguishing one object from another, nor does it allow us to judge the quality of our abstractions. Our experience therefore suggests the following definition:

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 userhostile; they will happily swallow the excess coins.

In each of these circumstances, we see how the behavior of an object is influenced by its history: the order in which one operates upon the object is important. The reason for this event- and time-dependent behavior is the existence of state within the object. For example, one essential state associated with the vending machine is the amount of money currently entered by a user but not yet applied to a selection. Other important properties include the amount of change available and the quantity of soft drinks on hand.

From this example, we may form the following low-level definition:

The 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.

Another property of a vending machine is that it can accept coins. This is a static (that is, fixed) property, meaning that it is an essential characteristic of a vending machine. In contrast, the actual quantity of coins accepted at any given moment

represents the dynamic value of this property, and is affected by the order of operations upon the machine. This quantity increases as a user inserts coins, and then decreases when a salesperson services the machine. We say that values are "usually dynamic" because in some cases values are static. For example, the serial number of a vending machine is a static property and value.

A property is an inherent or distinctive characteristic, trait, quality, or feature that contributes to making an object uniquely that object. For example, one essential property of an elevator is that it is constrained to travel up and down and not horizontally. Properties are usually static, because attributes such as these are unchanging and fundamental to the nature of the object. We say "usually" static, because in some circumstances the properties of an object may change. For example, consider an autonomous robot that can learn about its environment. It may first recognize an object that appears to be a fixed barrier, only to learn later that this object is in fact a door that can be opened. In this case, the object created by the robot as it builds its conceptual model of the world gains new properties as new knowledge is acquired.

All properties have some value. This value might be a simple quantity, or it might denote another object. For example, part of the state of an elevator might have the value 3, denoting the current floor on which the elevator is located. In the case of the vending machine, the state of the vending machine encompasses many other objects, such as a collection of soft drinks. The individual drinks are in fact distinct objects; their properties are different from those of the machine (they can be consumed, whereas a vending machine cannot), and they can be operated upon in distinctly different ways. Thus, we distinguish between objects and simple values: simple quantities such as the number 3 are "a temporal, unchangeable, and non-instantiated," whereas objects "exist in time, are changeable, have state, are instantiated, and can be created, destroyed, and shared" .

The fact that every object has state implies that every object takes up some amount of space, be it in the physical world or in computer memory.

Example Consider the structure of a personnel record. In C++ we might write:

```
struct PersonnelRecord
{
    char   name[100];
    int    socialSecurit Number;
    char   department[10];
    float  salary;
}
```

Each part of this structure denotes a particular property of our abstraction of a personnel record. This declaration denotes a class, not an object, because it does not represent a specific instance¹⁴. To declare objects of this class, we write

```
PersonnelRecord deb, dave, karen, jim, tom, denise, kaitlyn, krista, elyse;
```

Here, we have nine distinct objects, each of which takes up some amount of space in memory. None of these objects shares its space with any other object,

although each of them has the same properties; thus their states have a common representation.

It is good engineering practice to encapsulate the state of an object rather than expose it as in the preceding declaration. For example, we might rewrite that class declaration as follows:

```
class PersonnelRecord {
public:
    char* employeeName() const;
    int   employeeSocialSecurityNumber() const;
    char* employeeDepartment() const;
protected:
    char  name[100];
    int   socialSecurityNumber;
    char  department[10];
    float salary;
};
```

This declaration is slightly more complicated than the previous one, but it is superior for a number of reasons¹⁵. Specifically, we have written this class so that its representation is hidden from all other outside clients. If we change its representation, we will have to recompile some code, but semantically, no outside client will be affected by this change (in other words, existing code will not break). Also, we have captured certain decisions about the problem space by explicitly stating some of the operations that clients may perform upon objects of this class. In particular, we grant all clients the right to retrieve the name, social security number, and department of an employee. Only special clients (namely, subclasses of this class) have permission to modify the values of these properties. Furthermore, only these special clients may modify or retrieve the salary of an employee, whereas outside clients may not. Another reason why this declaration is better than the previous one has to do with reuse. As we will see in a later section, inheritance makes it possible for us to reuse this abstraction, and then refine it or specialize it in a variety of ways.

We may say that all objects within a system encapsulate some state, and that all of the state within a system is encapsulated by objects. However, encapsulating the state of an object is a start, but is not enough to allow us to capture the full intent of the abstractions we discover and invent during development. For this reason, we must also consider how objects behave.

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.

An operation is some action that one object performs upon another in order to elicit a reaction. For example, a client might invoke the operations `append` and `pop` to grow and shrink a queue object, respectively. A client might also invoke the operation `length`, which returns a value denoting the size of the queue object but does not alter the state of the queue itself. In pure object-oriented languages such as Smalltalk, we speak of one object passing a message to another. In languages such as C++, which derive from more procedural ancestors, we speak of one object invoking the member function of another. Generally, a message is simply an

operation that one object performs upon another, although the underlying dispatch mechanisms are different. For our purposes, the terms *operation* and *message* are interchangeable.

In most object-oriented programming languages, operations that clients may perform upon an object are typically declared as *methods*, which are part of the class's declaration. C++ uses the term *member function* to denote the same concept; we will use these terms interchangeably.

Message passing is one part of the equation that defines the behavior of an object; our definition for behavior also notes that the state of an object affects its behavior as well. Consider again the vending machine example. We may invoke some operation to make a selection, but the vending machine will behave differently depending upon its state. If we do not deposit change sufficient for our selection, then the machine will probably do nothing. If we provide sufficient change, the machine will take our change and then give us our selection (thereby altering its state). Thus, we may say that the behavior of an object is a function of its state as well as the operation performed upon it, with certain operations having the side effect of altering the object's state. This concept of side effect thus leads us to refine our definition of state:

The state of an object represents the cumulative results of its behavior.

Most interesting objects do not have state that is static; rather, their state has properties whose values are modified and retrieved as the object is acted upon.

Example Consider the following declaration of a queue class in C++:

```
class Queue {
public:
    Queue();
    Queue(const Queue&);
    virtual ~Queue();

    virtual Queue& operator=(const Queue&);
    virtual int operator=(const Queue&) const;
    int operator!=(const Queue&) const;
    virtual void clear();
    virtual void append(const void*);
    virtual void pop();
```

```
virtual void remove(int at),  
virtual int length() const;  
virtual int isEmpty() const;  
virtual const void* front() const;  
virtual int location(const void*);
```

protected:

...


```
};
```

This class uses the common C idiom of setting and getting items via `void*`, which provides the abstraction of a heterogeneous queue, meaning that clients can append objects of any class to a queue object. This approach is not particularly type-safe, because the client must remember the class of the objects placed in the queue. Also, the use of `void*` prevents the `Queue` object from “owning” its items, meaning that we cannot rely upon the action of the queue's destructor (`~Queue()`) to destroy the elements in the queue. In a later section we will study parameterized types, which mitigate these problems.

Since the declaration `Queue` represents a class, not an object, we, must declare instances that clients can manipulate:

```
Queue a, b, c, d;
```

Continuing, we may operate upon these objects as in the following code:

```
a.append(&deb);
a.append(&karen);
a.append(kdenise);
b = a;
a.pop();
```

After executing these statements, the queue denoted by `a` contains two items (with a pointer to the `karen` record at its front), and the queue denoted by `b` contain three items (with the `deb` record at its front). In this manner, each of these queue objects embodies some distinct state, and this state affects the future behavior of each object. For example, we may safely pop `b` three more times, but `a` may be safely popped only two more times.

Operations An operation denotes a service that a class offers to its clients. In practice, we have found that a client typically performs five kinds of operations upon an object¹⁶. The three most common kinds of operations are the following:

- **Modifier** An operation that alters the state of an object
- **Selector** An operation that accesses the state of an object, but does not alter the state
- **Iterator** An operation that permits all parts of an object to be accessed in some well-defined order

Because these operations are so logically dissimilar, we have found it useful to apply a coding style that highlights their differences. For example, in our declaration of the class `Queue`, we first declare all modifiers as non-const member functions (the operations `clear`, `append`, `pop`, and `remove`), followed by all selectors as const functions (the operations `length`, `isEmpty`, `front`, and

location).

Two other kinds of operations are common; they represent the infrastructure necessary to create and destroy instances of a class:

- Constructor An operation that creates an object and/or initializes its state
- Destructor An operation that frees the state of an object and/or destroys the object itself

In C++, constructors and destructors are declared as part of the definition of a class (the members `Queue` and `~Queue`), whereas in Smalltalk and CLOS, such operations are typically part of the protocol of a metaclass (that is, the class of a class).

In pure object-oriented programming languages such as Smalltalk, operations may only be declared as methods, since the language does not allow us to declare procedures or functions separate from any class. In contrast, languages such as Object Pascal, C++, CLOS, and Ada allow the developer to write operations as free subprograms; in C++, these are called nonmember functions. *Free subprograms* are procedures or functions that serve as nonprimitive operations upon an object or objects of the same or different classes. Free subprograms are typically grouped according to the classes upon which they are built; therefore, we call such collections of free subprograms *class utilities*. For example, given the preceding declaration of the package `Queue`, we might write the following nonmember function:

```
void copyUntilFound(Queue& from, Queue& to, void* item)
{
    while ((!from.isEmpty()) && (from.front() != item)) {
        to.append(from.front());
        from.pop();
    }
}
```

The purpose of this operation is to repeatedly copy and then pop the contents of one queue until the given item is found at the front of the queue. This operation is not primitive; it can be built from lower-level operations that are already a part of the `Queue` class.

It is common style in C++ (and Smalltalk) to collect all logically related free subprograms and declare them as part of a class that has no state. In particular, in C++, these become static.

Thus, we may say that all methods are operations, but not all operations are methods: some operations may be expressed as free subprograms. In practice, we are inclined to declare most operations as methods, although as we discuss in a later section, there are sometimes compelling reasons to do otherwise, such as when a particular operation affects two or more objects of different classes, and there is no particular benefit in declaring that operation in one class over the other.

Roles and Responsibilities Collectively, all of the methods and free subprograms associated with a particular object comprise its *protocol*. The protocol of an object thus defines the envelope of an object's allowable behavior, and so comprises the entire static and dynamic view of the object. For most nontrivial abstractions, it is useful to divide this larger protocol into logical groupings of behavior. These collections, which thus partition the behavior space of an object, denote the *roles* that an object can play. As Adams suggests, a role is a mask that an object wears [8], and so defines a contract between an abstraction and its clients.

Unifying our definitions of state and behavior, Wirfs-Brock defines the *responsibilities* of an object to "include two key items: the knowledge an object maintains and the actions an object can perform. Responsibilities are meant to convey a sense of the purpose of an object and its place in the system. The responsibilities of an object are all the services it provides for all of the contracts it supports" . In other words, we may say that the state and behavior of an object collectively define the roles that an object may play in the world, which in turn fulfill the abstraction's responsibilities.

Indeed, most interesting objects play many different roles during their lifetime; for example :

A bank account may be in good or bad standing, and which role it is in affects the semantics of a withdrawal transaction.

To a trader, a share of stock represents an entity with value that may be bought or sold; to a lawyer, the same share denotes a legal instrument encompassing certain rights.

In the course of one day, the same person may play the role of mother, doctor, gardener, and movie critic.

In the case of the bank account, the roles that this object can play are dynamic yet mutually exclusive: a bank account can be either in good or bad standing, but not both. In the case of the share of stock, its roles overlap slightly, but each role is static relative to the client that interacts with the share. In the case of the person, her roles are quite dynamic, and may change from moment to moment.

As we will discuss further in Chapters 4 and 6, we often start our analysis of a problem by examining the various roles that an object plays. During design, we refine these roles by inventing the particular operations that carry out each role's responsibilities.

Objects as Machines The existence of state within an object means that the order in which operations are invoked is important. This gives rise to the idea that each object is like a tiny, independent machine [11]. Indeed, for some objects, this event- and time-ordering of operations is so pervasive that we can best formally characterize the behavior of such objects in terms of an equivalent finite state machine. In Chapter 5, we will show a particular notation for hierarchical finite state machines that we may use for expressing these semantics.

Continuing the machine metaphor, we may classify objects as either active or passive. An *active object* is one that encompasses its own thread of control, whereas a *passive object* does not. Active objects are generally autonomous, meaning that they can exhibit some behavior without being operated upon by another object. Passive objects, on the other hand, can only undergo a state change when explicitly acted upon. In this manner, the active objects in our system serve as the roots of control. If our system involves multiple threads of control, then we will usually have multiple active objects. Sequential systems, on the other hand, usually have exactly one active object, such as a main window object responsible for managing an event loop that dispatches messages. In such architectures, all other objects are passive, and their behavior is ultimately triggered by messages from the one active object. In other kinds of sequential system architectures (such as transaction processing systems), there is no obvious central active object, and so control tends to be distributed throughout the system's passive objects.

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.

Example Consider the following declarations in C++. First, we start with a simple structure that denotes a point in space:

```
struct Point {  
    int x;  
    int y;  
    Point() : x(0), y(0) {}  
    Point(int xValue, int yValue) : x(xValue), y(yValue) {}  
};
```

Here, we have chosen to declare `Point` as a structure, not as a full-blown class. The rule of thumb we apply to make this distinction is simple. If our abstraction represents a simple record of other objects and has no really interesting behavior that applies to the object as a whole, make it a structure. However, if our abstraction requires behavior more intense than just simple puts and gets of largely independent record items, then make it a class. In the case of our `Point` abstraction, we define a point as (x, y) representing an (x, y) coordinate in

space. For convenience, we provide one constructor that provides a default $(0, 0)$ value, and another

constructor that initializes a point with an explicit

value.

Next, we provide, a class that denotes a display item. A display item is a common abstraction in all GUI-centric systems: it represents the base class of all objects that have a visual representation on some window, and so captures the structure and behavior common to all such objects. Here we have an abstraction that is more than just a simple record of data. Clients expect to be able to draw, select, and move display items, as well as query their selection state and location. We may capture our abstraction in the following C++ declaration:

```
class DisplayItem {
public:
    DisplayItem();
    DisplayItem(const Point& location);
    virtual ~DisplayItem();

    virtual void draw();
    virtual void erase();
    virtual void select();
    virtual void unselect();
    virtual void move(const Point& location);

    int isSelected() const;
    Point location() const;
    int isUnder(const Point& location) const;

protected:
    ...
};
```

This declaration is incomplete: we have intentionally omitted all of the constructors and operators needed to handle copying, assignment, and tests for equality. We will consider these aspects of our abstraction in the next section.

Because we expect clients to declare subclasses of this class, we have declared its destructor and all of its modifiers as virtual. In particular, we expect concrete subclasses to redefine `draw` to reflect the behavior of drawing domain specific items in a window. We have not declared any of its selectors as virtual, because we do not expect subclasses to refine this behavior. Note also that the one selector `isUnder` involves more than just retrieving a simple state value. Here, the semantics of this operation require the object to calculate if the given point falls anywhere within the frame of the display item.

To declare instances of this class, we might write the following:

```
DisplayItem item1;
DisplayItem* item2 = new DisplayItem(Point(75, 75));
DisplayItem* item3 = new DisplayItem(Point(100, 100));
DisplayItem* item4 = 0;
```

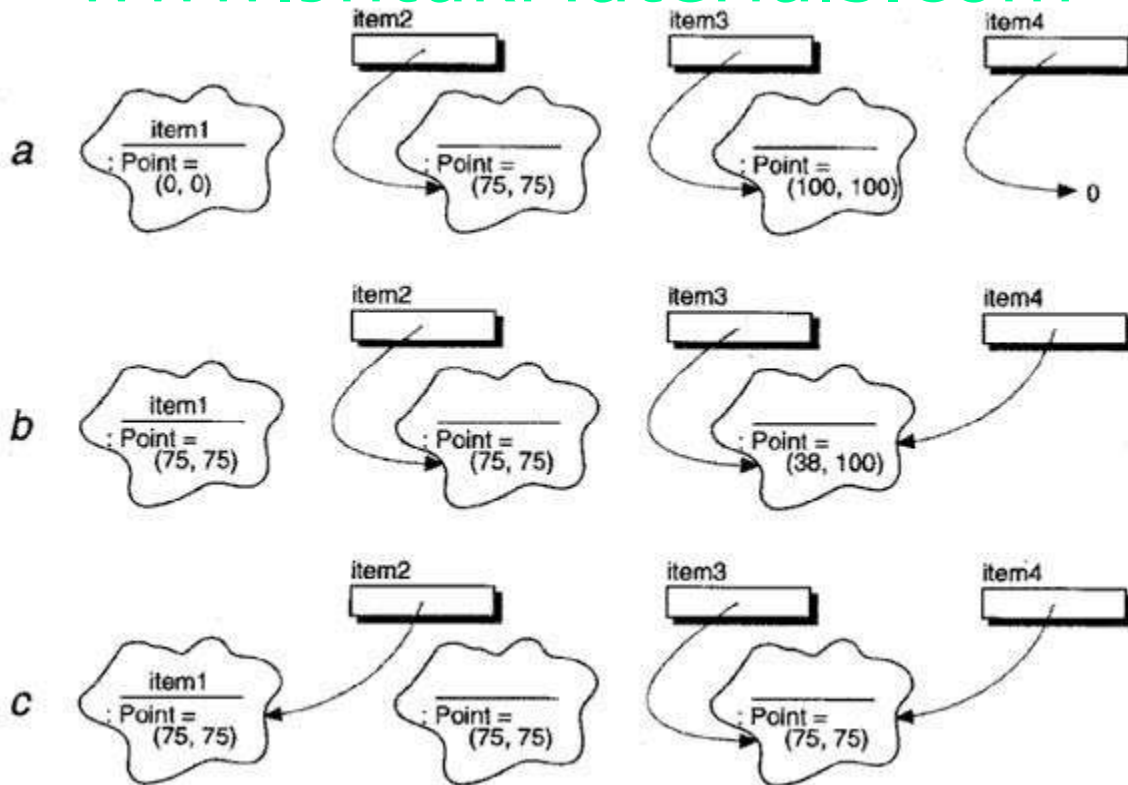


Figure 3-1
Object Identity

As Figure 3-1a shows, the elaboration of these declarations creates four names and three distinct objects. Specifically, elaboration sets aside four locations in memory whose names are `item1`, `item2`, `item3`, and `item4`, respectively. Also, `item1` is the name of a distinct `DisplayItem` object, but the other three names each denote a *pointer* to a `DisplayItem` object. Only `item2` and `item3` actually point to distinct `DisplayItem` objects (because only their declarations allocate a new `DisplayItem` object); `item4` designates no such object. Furthermore, the names of the objects pointed to by `item2` and `item3` are anonymous: we can only refer to these distinct objects indirectly, by dereferencing their pointer value. Thus, we may properly say that `item2` points to a distinct `DisplayItem` object, whose name we may refer to indirectly as **item2*. The unique identity (but not necessarily the name) of each object is preserved over the lifetime of the object, even when its state is changed. This is like the Zen question about a river: is a river the same river from one day to the next, even though the same water never flows through it? For example, consider the results of executing the following statements:

```
item1.move(item2->location());
item4 = item3;
item4->move(Point(38, 100));
```

Figure 3-1b illustrates these results. Here we see that `item1` and the object designated by `item2` both have the same location state, and that `item4` now also designates the same object as does `item3`. Notice that we use the phrase "the object designated by `item2`" rather than saying "the object `item2`." The first phrase is more precise, although we will sometimes use these phrases interchangeably.

Although `item1` and the object designated by `item2` have the same state, they represent distinct objects. Also, note that we have changed the state of the object designated by `item3` by operating upon it through its new indirect name, `item4`. This is a situation we call *structural sharing*, meaning that a given object can be named in more than one way; in other words, there are aliases to the object. Structural sharing is the source of many problems in object-oriented programming. Failure to recognize the side effects of operating upon an object through aliases often leads to memory leaks, memory-access violations, and even worse, unexpected state changes. For example, if we destroyed the object designated by `item3` using the expression `delete item3`, then `item4`'s pointer value would be meaningless: this is a situation we call a *dangling reference*.

Consider also Figure 3-1c, which illustrates the results of executing the following statements:

```
item2 = &item1;  
item4->move(item2->location());
```

The first statement introduces an alias, for now `item2` designates the same object as `item1`; the second statement accesses the state of `item1` through the new alias. Unfortunately, we have introduced a memory leak: the object originally designated by `item2` can no longer be named, either directly or indirectly, and so its identity is lost. In languages such as Smalltalk and CLOS, such objects will be garbage-collected and their storage reclaimed automatically, but in languages such as C++, their storage will not be reclaimed until the program that created them finishes. Especially for long-running programs, memory leaks such as this are either bothersome or disastrous¹⁷.

Copying, Assignment, and Equality Structural sharing takes place when the identity of an object is aliased to a second name. In most interesting object-oriented applications, using aliases simply cannot be avoided. For example, consider the following two function declarations in C++:

```
void highlight(DisplayItem& i);  
void drag(DisplayItem i);           // Dangerous
```

Invoking the first function with the argument `item1` creates an alias: the formal parameter `i` denotes a reference to the object designated by the actual parameter, and hence `item1` and `i` will name the same object at execution time. On the other hand, invoking the second function with the argument `item1` makes a copy of the actual parameter, and so there is no alias: `i` denotes a completely different object (but with the same state) as does `item1`. In languages such as C++ where there is a distinction between passing arguments by reference versus by

value, care must be taken to avoid operating upon a copy of an object, when the intent was to operate upon the original object itself¹⁸. Indeed, as we will discuss in a later section, passing objects by reference in C++ is essential to eliciting polymorphic behavior. In general, passing objects by reference is the most desirable practice for nonprimitive objects, for its semantics only involve copying references, not state, and hence is far more efficient for passing anything larger than simple values.

In some circumstances, however, copying is the intended semantics, and in languages such as C++, it is possible to control the semantics of copying. In particular, we may introduce a copy constructor to a class's declaration, as in the following code fragment, which we would declare as part of the declaration for `DisplayItem`:

```
DisplayItem(const DisplayItem&);
```

In C++, a copy constructor may be invoked either explicitly (as part of the declaration of an object) or implicitly (as when passing an object by value). Omitting this special constructor invokes the default copy constructor, whose semantics are defined as a memberwise copy. However, for objects whose state itself involves pointers or references to other objects, default memberwise copying is usually dangerous, for copying then implicitly introduces lower-level aliases. The rule of thumb we apply, therefore, is that we omit an explicit copy constructor only for those abstractions whose state consists of simple, primitive values; in all other cases, we usually provide an explicit copy constructor.

This practice distinguishes what some languages call *shallow* versus *deep* copying. Smalltalk, for example, provides the methods `shallowCopy` (which copies the object, but shares its state) and `deepCopy` (which copies the object as well as its state, and recursively so). Redefining these operations for aggregate classes permits a mixture of semantics: copying a higher-level object might copy most of its state, but introduce aliases for certain other lower-level elements.

Assignment is also generally a copying operation, and in languages such as C++, its semantics can be controlled as well. For example, we might add the following declaration to our declaration of `DisplayItem`:

```
virtual DisplayItem& operator=(const DisplayItem&);
```

We declare this operator as virtual, because we expect a subclass to redefine its behavior. As with the copy constructor, we may implement this operation to provide either shallow or deep copy semantics. Omitting this explicit declaration invokes the default assignment operator, whose semantics are defined as a memberwise copy.

Closely related to the issue of assignment is that of equality. Although it seems like a simple concept, equality can mean one of two things. First, equality can mean that two names designate the same object. Second, equality can mean that two names designate distinct objects whose states are equal. For example, in Figure 3-1c, both kinds of equality evaluate to true between `item1` and `item2`. However, only the second kind of equality evaluates to true between `item1` and `item3`.

In C++, there is no default equality operator, thus we must establish our own semantics by introducing the explicit operators for equality and inequality as part of the declaration for

`DisplayItem`:

```
virtual int operator==(const DisplayItem&)
const; int operator!=(const DisplayItem&) const;
```

Our style is to declare the equality operator as virtual (because we expect subclasses to redefine its behavior) and to declare the inequality operator as nonvirtual (we always want inequality to mean the logical negation of equality: subclasses should not override this behavior).

In a similar manner, we may explicitly define the meaning of ordering operators, such as tests for less-than or greater-than orderings between two objects.

Object Life Span The lifetime of an object extends from the time it is first created (and thus first consumes space) until that space is reclaimed. To explicitly create an object, we must either declare it or allocate it.

Declaring an object (such as `item1` in our earlier example) creates a new instance on the stack. Allocating an object (such as `item3`) creates a new instance on the heap. In C++, in either case, whenever an object is created, its constructor is automatically invoked, whose purpose is to allocate space for the object and establish an initial stable state. In languages such as Smalltalk, such constructor operations are actually a part of the object's metaclass, not the object's class - we will examine metaclass semantics later in this chapter.

Often, objects are created implicitly. For example, in C++ passing an object by value creates a new object on the stack that is a copy of the actual parameter. Furthermore, object creation is transitive: creating an aggregate object also creates any objects that are physically a part of the whole. Overriding the semantics of the copy constructor and assignment operator in C++ permits explicit control over when such parts are created and destroyed. Also, in C++ it is possible to redefine the semantics of the `new` operator (which allocates instances on the heap), so that each class can provide its own memory management policy.

In languages such as Smalltalk, an object is destroyed automatically as part of garbage collection when all references to it have been lost. In languages without garbage collection, such as C++, an object continues to exist and consume space even if all references to it are lost. Objects created on the stack are implicitly destroyed whenever control passes beyond the

block in which the object was declared. Objects created on the heap with the `new` operator must be explicitly destroyed with the `delete` operator. Failure to do so leads to memory leaks, as we discussed earlier. Deallocating an object twice (usually because of an alias) is equally bad, and may manifest itself in memory corruption or a complete crash of the system.

In C++, whenever an object is destroyed either implicitly or explicitly, its destructor is automatically invoked, whose purpose is to deallocate space assigned to the object and its part, and to otherwise clean up after the object (such as, for example, closing files and releasing resources)¹⁹.

Persistent objects have slightly different semantics regarding destruction. As we discussed in the previous chapter, certain objects may be persistent, meaning that their lifetime transcends the lifetime of the program that created them. Persistent objects are usually elements of some larger, object-oriented database framework, and so the semantics of destruction (and also creation) are largely a function of the policies of the particular database system. In such systems, the most common approach to providing persistence is the use of a persistent mixin class. All objects for which we desire persistence semantics thus have this mixin class as a superclass somewhere in their class's inheritance lattice.

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

Seidewitz and Stark call these *seniority* and *parent/child* relationships, respectively [15].

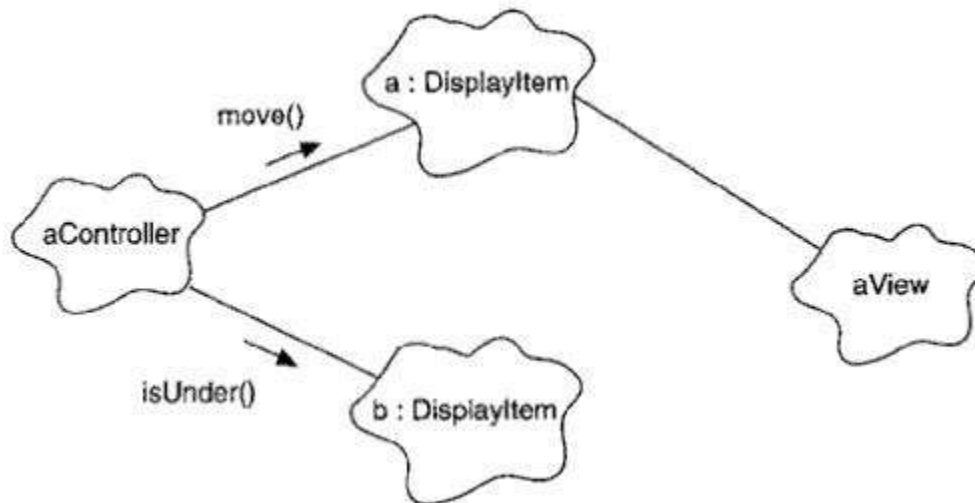


Figure 3-2
Links

Links

Semantics The term *link* derives from Rumbaugh, who defines it as a "physical or conceptual connection between objects" [16]. 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 one-object 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.

Message passing between two objects is typically unidirectional, although it may occasionally be bidirectional. In our example, the object `aController` only invokes operations upon the two display objects (to move them and query their location), but the display objects do not themselves operate upon the controller object. This separation of concerns is quite common in well-structured object-oriented systems, as we discuss in Chapter 5²⁰. Notice also that

although message passing is initiated by the client (such as `aController`) and is directed toward the supplier (such as object `a`), data may flow in either direction across a link. For example, when `aController` invokes the operation `move` upon `a`, data flows from the client to the supplier. However, when `aController` invokes the operation `isUnder` upon object `b`, the result passes from the supplier to the client.

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

Restricted to the context of Figure 3-2, `aController` represents an actor object, `aView` represents a server object, and `a` represents an agent that carries out the controller's request to draw the item in the view.

Example In many different kinds of industrial processes, certain reactions require a temperature ramp, wherein we raise the temperature of some substance, hold it at that temperature for a fixed period, and then let it cool to ambient temperature. Different processes require different profiles: some objects (such as telescope mirrors) must be cooled slowly, whereas other materials (such as steel) must be cooled rapidly. This abstraction of a temperature ramp has a sufficiently well-defined behavior that it warrants the creation of a class, such as the following. First, we introduce a typedef whose values represent elapsed time in minutes:

Number denoting elapsed minutes

```
typedef unsigned int Minute;
```

This typedef is similar to that for `Day` and `Hour`, which we introduced in Chapter 2. Next, we provide the class `TemperatureRamp`, which is conceptually a time/temperature mapping:

```
class TemperatureRamp {
public:
    TemperatureRamp();
    virtual ~TemperatureRamp();

    virtual void clear();
```



```
Temperature temperatureAt(Minute);
```

```
protected:
```

```
    ...  
}
```

In keeping with our style, we have declared a number of operations as virtual, because we expect there to be subclasses of this class.

Actually, the behavior of this abstraction is more than just a literal time/temperature mapping. For example, we might set a temperature ramp that requires the temperature to be 250° F at time 60 (one hour into the temperature ramp) and 150° F at time 180 (three hours into the process), but then we would like to know what the temperature should be at time

This requires linear interpolation, which is therefore another behavior we expect of this abstraction.

One behavior we explicitly do not require of this abstraction is the control of a heater to carry out a particular temperature ramp. Rather, we prefer a greater separation of concerns, wherein this behavior is achieved through the collaboration of three objects: a temperature ramp instance, a heater, and a temperature controller. For example, we might introduce the following class: .

```
class TemperatureController {  
public:  
    TemperatureController(Location);  
    ~TemperatureController();  
  
    void process(const TemperatureRamp&);  
  
    Minute schedule(const TemperatureRamp&) const;  
  
private:  
    ...  
}
```

This class uses the typedef `Location` introduced in Chapter 2. Notice that we do not expect there to be any subclasses of this class, and so have not made any of its operations virtual.

The operation `process` provides the central behavior of this abstraction; its purpose is to carry out the given temperature ramp for the heater at the given location. For example, given the following declarations:

```
TemperatureRamp growingRamp;  
TemperatureController rampController(7);
```

We might then establish a particular temperature ramp, then tell the controller to carry out this profile:


```
growingRamp.bind(250, 60);  
growingRamp.bind(150, 180);  
  
rampController.process(growingRamp);
```

Consider the relationship between the objects `growingRamp` and `rampController`: the object `rampController` is an agent responsible for carrying out a temperature ramp, and so uses the object `growingRamp` as a server. This link manifests itself in the fact that the object `rampController` uses the object `growingRamp` as an argument to one of its operations.

A comment regarding our style: at first glance, it may appear that we have devised an abstraction whose sole purpose is to wrap a functional decomposition inside a class to make it appear noble and object-oriented. The operation `schedule` suggests that this is not the case. Objects of the class `TemperatureController` have sufficient knowledge to determine when a particular profile should be scheduled, and so we expose this operation as an additional behavior of our abstraction. In some high-energy industrial processes (such as steel making), heating a substance is a costly event, and it is important to take into account any lingering heat from a previous process, as well as the normal cool-down of an unattended heater. The operation `schedule` exists so that clients can query a `TemperatureController` object to determine the next optimal time to process a particular temperature ramp.

Visibility Consider two objects, `A` and `B`, with a link between the two. In order for `A` to send a message to `B`, `B` must be visible to `A` in some manner. During our analysis of a problem, we can largely ignore issues of visibility, but once we begin to devise concrete implementations, we must consider the visibility across links, because our decisions here dictate the scope and access of the objects on each side of a link.

In the previous example, the object `rampController` has visibility to the object `growingRamp`, because both objects are declared within the same scope, and `growingRamp` is presented as an argument to an operation upon the object `rampController`. Actually, this is just one of the four different ways that one object may have visibility to another:

- The supplier object is global to the client.

- The supplier object is a parameter to some operation of the client.

- The supplier object is a part of the client object.

- The supplier object is a locally declared object in some operation of the client. How one object is made visible to another is a tactical design issue.

Synchronization Whenever one object passes a message to another across a link, the two objects are said to be *synchronized*. For objects in a completely sequential application, this synchronization is usually accomplished by simple method invocation, as described in the

sidebar. However, in the presence of multiple threads of control, objects require more sophisticated message passing in order to deal with the problems of mutual exclusion that can occur in concurrent systems. As we described earlier, active objects embody their own thread of control, and so we expect their semantics to be guaranteed in the presence of other active objects. However, when one active object has a link to a passive one, we must choose one of three approaches to synchronization:

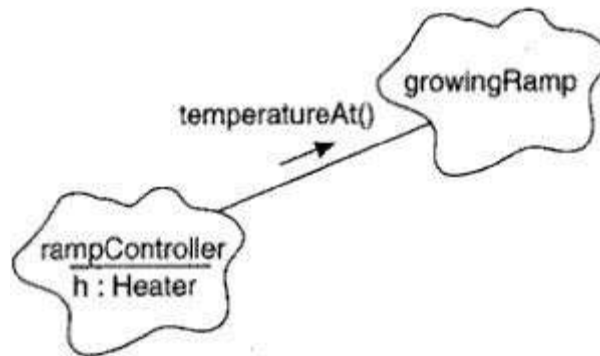


Figure 3-3
Aggregation

- Sequential The semantics of the passive object are guaranteed only in the presence of a single active object at a time.
- Guarded The semantics of the passive object are guaranteed in the presence of multiple threads of control, but the active clients must collaborate to achieve mutual exclusion.
- Synchronous The semantics of the passive object are guaranteed in the presence of multiple threads of control, and the supplier guarantees mutual exclusion.

All the objects described thus far in this chapter are sequential. In Chapter 9, we will illustrate each of these other forms in greater detail.

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.

There are clear trade-offs between links and aggregation. Aggregation is sometimes better because it encapsulates parts as secrets of the whole. Links are sometimes better because they permit looser coupling among objects. Intelligent engineering decisions require careful weighing of these two factors.

By implication, an object that is an attribute of another has a link to its aggregate. Across this link, the aggregate may send messages to its parts.

Example To continue our declaration of the class `TemperatureController`, we might complete its private part as follows:

```
Heater h;
```

This declares `h` as a part of each instance of `TemperatureController`. According to our declaration of the class `Heater` in the previous chapter, we must properly create this attribute, because its class does not provide a default constructor. Thus, we might write the constructor for the

`TemperatureController` as follows:

```
TemperatureController::TemperatureController(location l)
: h(l) {}
```

The Nature of a Class

What Is and What Isn't a Class

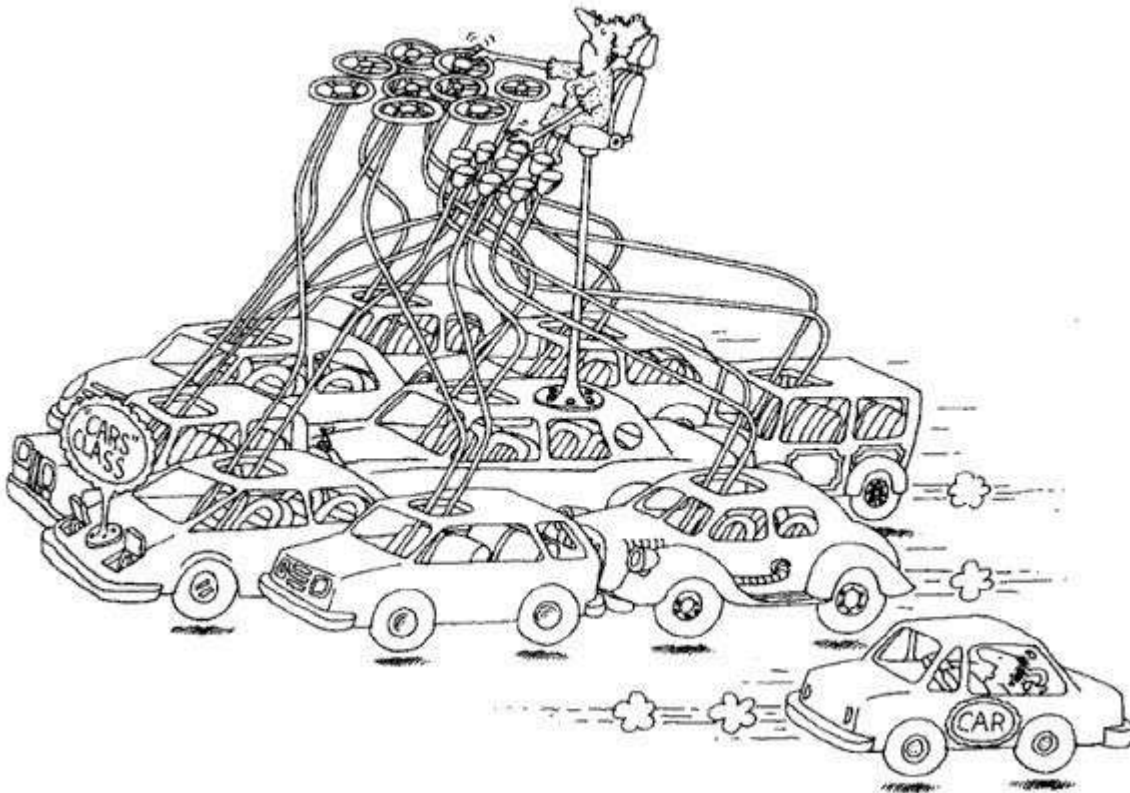
The concepts of a class and an object are tightly interwoven, for we cannot talk about an object without regard for its class. However, there are important differences between these two terms. Whereas an object is a concrete entity that exists in time and space, a class represents only an abstraction, 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."

In everyday terms, we may define a class as "a group, set, or kind marked by common attributes or a common attribute; a group division, distinction, or rating based on quality,

degree of competence, or condition" [17]²¹. In the context of object-oriented analysis and design, we define a class as follows:

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.

What isn't a class? An object is not a class, although, curiously, as we will describe later, a class may be an object. 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.

It is important to note that the class - as defined by most programming languages - is a necessary but insufficient vehicle for decomposition. Sometimes abstractions are so complex that they cannot be conveniently expressed in terms of a single class declaration. For example, at a sufficiently high level of abstraction, a GUI framework, a database, and an entire inventory system are all conceptually individual objects, none of which can be expressed as a

single class²². Instead, it is far better for us to capture these abstractions as a cluster of classes whose instances collaborate to provide the desired structure and behavior. Stroustrup calls such a cluster a *component* [18]. For reasons that we will explain in Chapter 5, we call each such cluster a *class category*.

Interface and Implementation

Meyer [19] and Snyder [20] 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.

This view of programming as contracting leads us to distinguish between the outside view and the inside view of a class. The *interface* of a class provides its outside view and therefore emphasizes the abstraction while hiding its structure and the secrets of its behavior. This interface primarily consists of the declarations of all the operations applicable to instances of this class, but it may also include the declaration of other classes, constants, variables, and exceptions as needed to complete the abstraction. By contrast, the *implementation* of a class is its inside view, which encompasses the secrets of its behavior. The implementation of a class primarily consists of the implementation of all of the operations defined in the interface of the class.

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

Different programming languages provide different mixtures of public, protected, and private parts, which developers can choose among to establish specific access rights for each part of a class's interface and thereby exercise control over what clients can see and what they can't see.

In particular, C++ allows a developer to make explicit distinctions among all three of these different parts²³. The C++ friendship mechanism permits a class to distinguish certain privileged classes that are given the rights to see the class's protected and private parts. Friendships break a class's encapsulation, and so, as in life, must be chosen carefully. By contrast, Ada permits declarations to be public or private, but not protected. In Smalltalk, all instance variables are private, and all methods are public. In Object Pascal, both fields and operations are public and hence unencapsulated. In CLOS, generic functions are public, and slots may be made private, although their access can be broken via the function `slot-value`.

The state of an object must have some representation in its corresponding class, and so is typically expressed as constant and variable declarations placed in the protected or private part of a class's interface. In this manner, the representation common to all instances of a class is encapsulated, and changes to this representation do not functionally affect any outside clients.

The careful reader may wonder why the representation of an object is part of the interface of a class (albeit a nonpublic part), not of its implementation. The reason is one of practicality; to do otherwise requires either object-oriented hardware or very sophisticated compiler technology. Specifically, when a compiler processes an object declaration such as the following in C++:

```
DisplayItem item1;
```

it must know how much memory to allocate to the object `item1`. If we defined the representation of an object in the implementation of a class, we would have to complete the class's implementation before we could use any clients, thus defeating the very purpose of separating the class's outside and inside views.

The constants and variables that form the representation of a class are known by various terms, depending upon the particular language we use. For example, Smalltalk uses the term *instance variable*, Object Pascal uses the term *field*, C++ uses the term *member object*, and CLOS uses the term *slot*. We will use these terms interchangeably to denote the parts of a class that serve as the representation of its instance's state.

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. As described earlier in this chapter, the instances

of such classes act as little machines, and since all such instances embody the same behavior, we can use the class to capture these common event- and time-ordered semantics. As we discuss in Chapter 5, we may describe such dynamic behavior for certain interesting classes by using finite state machines.

Relationships Among Classes

Kinds of Relationships

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.

From this simple example we conclude that classes, like objects, do not exist in isolation. Rather, for a particular problem domain, the key abstractions are usually related in a variety of interesting ways, forming the class structure of our design [21].

We establish relationships between two classes for one of two reasons. First, a class relationship might indicate some sort of sharing. For example, daisies and roses are both kinds of flowers, meaning that both have brightly colored petals, both emit a fragrance, and so on. Second, a class relationship might indicate some kind of semantic connection. Thus, we say that red roses and yellow roses are more alike than are daisies and roses, and daisies and roses are more closely related than are petals and flowers. Similarly, there is a symbiotic connection between ladybugs and flowers: ladybugs protect flowers from certain pests, which in turn serve as a food source for the ladybug.

In all, there are three basic kinds of class relationships [22]. The first of these is generalization/specialization, denoting an "is a" relationship. For instance, a rose is a kind of flower, meaning that a rose is a specialized subclass of the more general class, flower. The second is whole/part, which denotes a "part of" relationship. Thus, a petal is not a kind of a flower; it is a part of a flower. The third is association, which denotes some semantic dependency among otherwise unrelated classes, such as between ladybugs and flowers. As another example, roses and candles are largely independent classes, but they both represent things that we might use to decorate a dinner table.

Several common approaches have evolved in programming languages to capture generalization/specialization, whole/part, and association relationships. Specifically, most object-oriented languages provide direct support for some combination of the following relationships:

- Association
- Inheritance
- Aggregation
- Using
- Instantiation
- Metaclass

An alternate approach to inheritance involves a language mechanism called *delegation*, in which objects are viewed as prototypes (also called *exemplars*) that delegate their behavior to related objects, thus eliminating the need for classes [23].

Of these six different kinds of class relationships, associations are the most general but also the most semantically weak. As we will discuss further in Chapter 6, the identification of associations among classes is often an activity of analysis and early design, at which time we begin to discover the general dependencies among our abstractions. As we continue our design and implementation, we will often refine these weak associations by turning them into one of the other more concrete class relationships.

Inheritance is perhaps the most semantically interesting of these concrete relationships, and exists to express generalization/specialization relationships. In our experience, however, inheritance is an insufficient means of expressing all of the rich relationships that may exist among the key abstractions in a given problem domain. We also need aggregation relationships, which provide the whole/part relationships manifested in the class's instances. Additionally, we need using relationships, which establish the links among the class's instances. For languages such as Ada, C++, and Eiffel, we also need instantiation relationships, which, like inheritance, support a kind of generalization, although in an entirely different way. Metaclass relationships are quite different and are only explicitly supported by languages such as Smalltalk and CLOS. Basically, a metaclass is the class of a class, a concept that allows us to treat classes as objects.

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.

We may capture these semantics in C++ by using what Rumbaugh calls *buried pointers* [24].

For example, consider the highly elided declaration of these two classes:

```
class Product;  
class Sale;  
  
class Product {  
public:  
    ...  
protected:  
    Sale* lastSale;  
};
```

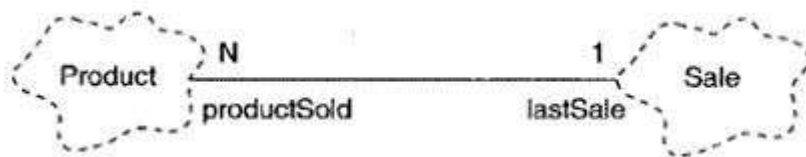


Figure 3-4
Association

```
class Sale {  
public:  
    ...  
protected:  
    Product** productSold;  
}
```

Here we show a one-to-many association: each instance of `Product` may have a pointer to its last sale, and each instance of `Sale` may have a collection of pointers denoting the products sold.

Semantic Dependencies As this example suggests, an association only denotes a semantic dependency and does not state the direction of this dependency (unless otherwise stated, an association implies bidirectional navigation, as in our example), nor does it state the exact way in which one class relates to another (we can only imply these semantics by naming the role each class plays in relationship with the other). However, these semantics are sufficient during the analysis of a problem, at which time we need only to identify such dependencies. Through the creation of associations, we come to capture the participants in a semantic relationship, their roles, and, as we will discuss, their cardinality.

Cardinality Our example introduced a one-to-many association, meaning that for each instance of the class `Sale`, there are zero or more instances of the class `Product`, and for each product, there is exactly one sale. This multiplicity denotes the *cardinality* of the association. In practice, there are three common kinds of cardinality across an association:

One-to-one

One-to-many
Many-to-many

A one-to-one relationship denotes a very narrow association. For example, in retail telemarketing operations, we would find a one-to-one relationship between the class `sale` and the class `CreditCardTransaction`: each sale has exactly one corresponding credit card transaction, and each such transaction corresponds to one sale. Many-to-many relationships are also common. For example, each instance of the class `customer` might initiate a transaction with several instances of the class `SalesPerson`, and each such salesperson might interact with many different customers. As we will discuss further in Chapter 5, there are variations upon these three basic forms of cardinality.

Inheritance

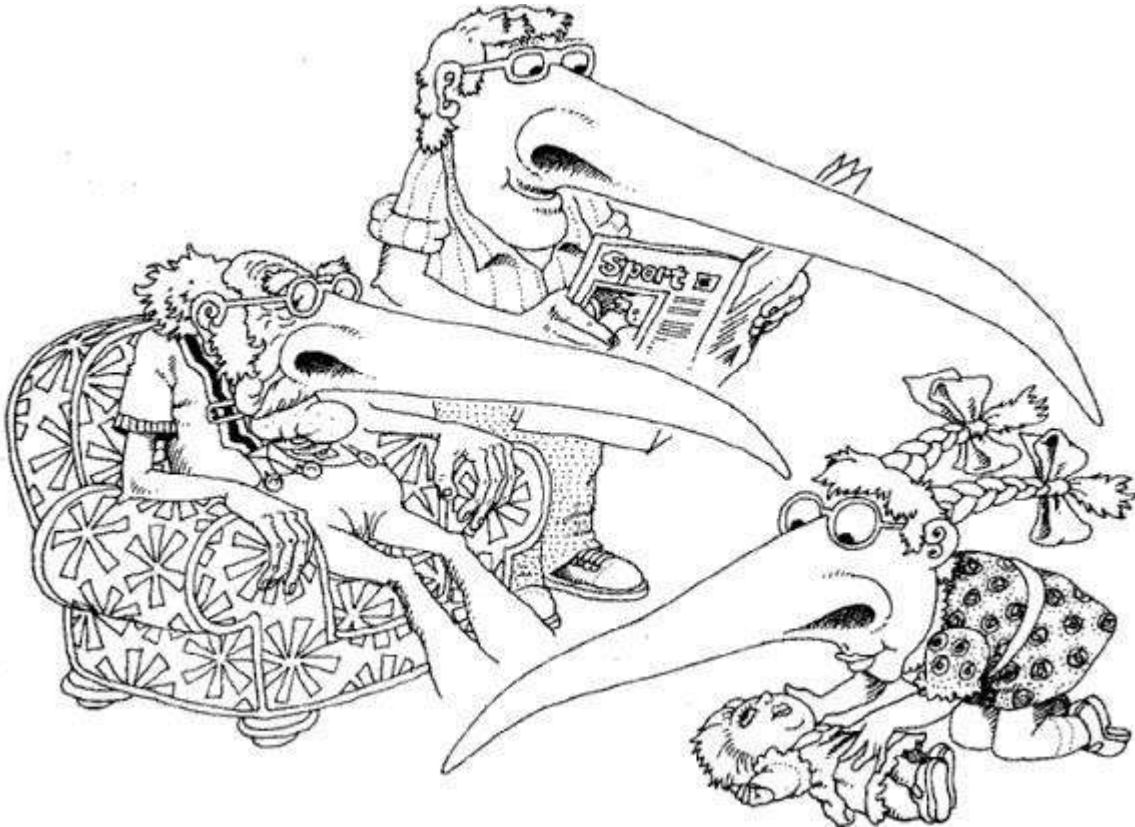
Examples After space probes are launched, they report back to ground stations with information regarding the status of important subsystems (such as electrical power and propulsion systems) and different sensors (such as radiation sensors, mass spectrometers, cameras, micro meteorite collision detectors, 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

```
class Time ...

struct ElectricalData {
    Time timeStamp;
    int id;
    float fuelCell1Voltage, fuelCell2Voltage;
    float fuelCell1Amperes, fuelCell2Amperes;
    float currentPower;
};
```

There are a number of problems with this declaration. First, the representation of `ElectricalData` is completely unencapsulated. Thus, there is nothing to prevent a client from changing the value of important data such as the `timeStamp` OR `currentPower` (which is a derived attribute, directly proportional to the current voltage and amperes drawn from both fuel cells). Furthermore, the representation of this structure is exposed, so if we were to change the representation (for example, by adding new elements or changing the bit alignment of existing ones), every client would be affected. At the very least, we would certainly have to recompile every reference to this structure. More importantly, such changes might violate the assumptions that clients had made about this exposed representation and cause the logic in our program to break. Also, this structure is largely devoid of meaning: a number of

operations are applicable to instances of this structure as a whole (such as transmitting the data, or calculating a check sum to detect errors during transmission), but there is no way to directly associate these operations with this structure. Lastly, suppose our analysis of the system's requirements reveals the need for several hundred different kinds of telemetry data, including other electrical data that encompassed the preceding information and also included voltage readings from various test points throughout the system. We would find that declaring these additional structures would create a considerable amount of redundancy, both in terms of replicated structures and common functions.



A subclass may inherit the structure and behavior of its superclass.

A slightly better way to capture our decisions would be to declare one class for each kind of telemetry data. In this manner, we could hide the representation of each class and associate its behavior with its data. Still, this approach does not address the problem of redundancy.

A far better solution, therefore, is to capture our decisions by building a hierarchy of classes, in which specialized classes inherit the structure and behavior defined by more generalized classes. For example:

```
class TelemetryData {  
public:  
    TelemetryData();  
    virtual ~TelemetryData();
```

```
virtual void transmit();

Time currentTime() const;

protected:
    int id;
    Time timeStamp;
};
```

This declares a class with a constructor and a virtual destructor (meaning that we expect to have subclasses), as well as the functions `transmit` and `currentTime`, which are both visible to all clients. The protected member objects `id` and `timeStamp` are slightly more encapsulated, and so are accessible only to the class itself and its subclasses. Note that we have declared the function `currentTime` as a public selector, which makes it possible for a client to access the `timeStamp`, but not change it.

Next, let's rewrite our declaration of the class `ElectricalData`:

```
class ElectricalData : public TelemetryData {
public:
    ElectricalData(float v1, float v2, float a1, float a2);
    virtual ~ElectricalData();
    virtual void transmit();

    float currentPower() const;

protected:
    float fuelCell1Voltage, fuelCell2Voltage;
    float fuelCell1Amperes, fuelCell2Amperes;
};
```

This class inherits the structure and behavior of the class `TelemetryData`, but adds to its structure (the four new protected member objects), redefines its behavior (the function `transmit`), and adds to its behavior (the function `currentPower`).

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.

A subclass typically augments or restricts the existing structure and behavior of its superclasses. A subclass that augments its superclasses is said to use inheritance for extension. For example, the subclass `GuardedQueue` might extend the behavior of its superclass `Queue` by providing extra operations that make instances of this class safe in the presence of multiple threads of control. In contrast, a subclass that constrains the behavior of its superclasses is said to use inheritance for restriction. For example, the subclass `UnselectableDisplayItem` might constrain the behavior of its superclass, `DisplayItem`, by prohibiting clients from selecting its instances in a view. In practice, it is not always so clear whether or not a subclass augments or restricts its superclass; in fact, it is common for a subclass to do both.

Figure 3-5 illustrates the single inheritance relationships deriving from the superclass `TelemetryData`. Each directed line denotes an "is a" relationship. For example, `CameraData` "is a" kind of `SensorData`, which in turn "is a" kind of `TelemetryData`. This is identical to the hierarchy one finds in a semantic net, a tool often used by researchers in cognitive science and artificial intelligence to organize knowledge about the world [25]. Indeed, as we discuss further in Chapter 4, designing a suitable inheritance hierarchy among abstractions is largely a matter of intelligent classification.

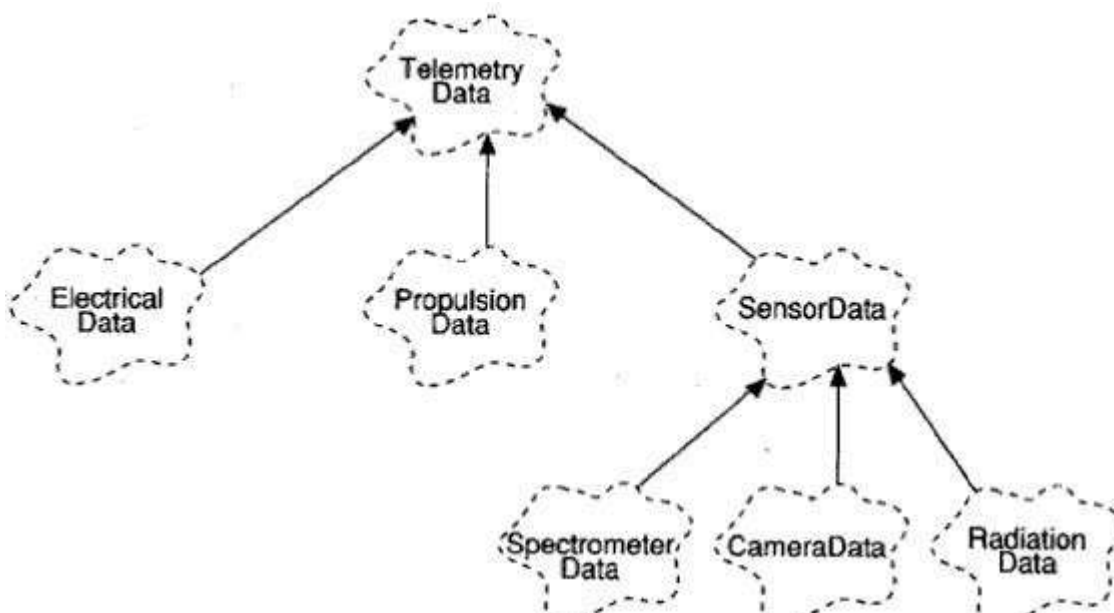


Figure 3-5
Single Inheritance

We expect that some of the classes in Figure 3-5 will have instances and some will not. For example, we expect to have instances of each of the most specialized classes (also known as leaf classes or concrete classes), such as `ElectricalData` and `SpectrometerData`. However, we are not likely to have any instances of the intermediate, more generalized classes, such as `SensorData` or even `TelemetryData`. Classes with no instances are called *abstract classes*. An abstract class is written with the expectation that its subclasses will add to its structure and behavior, usually by completing the implementation of its (typically) incomplete methods. In fact, in Smalltalk,

a developer may force a subclass to redefine the method introduced in an abstract class by using the method `subclassResponsibility` to implement a body for the abstract class's method. If the subclass fails to redefine it, then invoking the method results in an execution error. C++ similarly allows the developer to assert that an abstract class's method cannot be invoked directly by initializing its declaration to zero. Such a method is called a *pure virtual function*, and the language prohibits the creation of instances whose class exports such functions.

The most generalized class in a class structure is called the *base class*. Most applications have many such base classes, which represent the most generalized categories of abstractions within the given domain. In fact, especially in C++, well-structured object-oriented architectures generally have forests of inheritance trees, rather than one deeply rooted inheritance lattice. However, some languages require a topmost base class, which serves as the ultimate superclass of all classes. In Smalltalk, this class is called `Object`.

A given class typically has two kinds of clients :

- Instances
- Subclasses

It is often useful to define different interfaces for these two kinds of clients . In particular, we wish to expose only outwardly visible behaviors to instance clients, but we need to expose helping functions and representations only to subclass clients. This is precisely the motivation for the public, protected, and private parts of a class definition in C++: a designer can choose what members are accessible to instances, to subclasses, or to both clients. As we mentioned earlier, in Smalltalk the developer has less control over access: instance variables are visible to subclasses but not to instances, and all methods are visible to both instances and subclasses (one can mark a method as private, but this hiding is not enforced by the language).

There is a very real tension between inheritance and encapsulation. To a large degree, the use of inheritance exposes some of the secrets of an inherited class. Practically, this means that to understand the meaning of a particular class, you must often study all of its superclasses, sometimes including their inside views.

Inheritance means that subclasses inherit the structure of their superclass. Thus, in our earlier example, the instances of the class `ElectricalData` include the member objects of the superclass (such as `id` and `timeStamp`), as well as those of the more specialized classes (such as

`fuelCell1Voltage`, `fuelCell2Voltage`, `fuelCell1Amperes`, and `fuelCell2Amperes`)²⁴.

Subclasses also inherit the behavior of their superclasses. Thus, instances of the class `ElectricalData` may be acted upon with the operations `currentTime` (inherited from its superclass), `currentPower` (defined in the class itself) `transmit` (redefined in the subclass). Most object-oriented programming languages permit methods from a superclass to be redefined and new methods

to be added. In Smalltalk, for example, any superclass method may be redefined in a subclass. In C++, the developer has a bit more control. Member functions that are declared as *virtual* (such as the function `transmit`) may be redefined in a subclass; members declared otherwise (the default) may not be redefined (such as the function `currentTime`).

Single Polymorphism For the class `TelemetryData`, we might implement the member function `transmit` as follows:

```
void TelemetryData::transmit()
{
    //transmit the id
    //transmit the timestamp
}
```

We might implement the same member function for the class `ElectricalData` as follows:

```
void ElectricalData::transmit()
{
    TelemetryData::transmit();
    //transmit the voltages
    //transmit the amperes
}
```

In this implementation, we first invoke the corresponding superclass function (using the fully qualified name `TelemetryData::transmit`), which transmits the data's `id` and `timeStamp`, and then we transmit the data particular to the `ElectricalData` subclass.

Suppose that we have an instance of each of these two classes:

```
TelemetryData telemetry;
ElectricalData electrical(5.0, -5.0, 3.0, 7.0);
```

Now, given the following nonmember function,

```
void transmitFreshData(TelemetryData& d, const Time& t)
{
    if (d.currentTime() >= t)
        d.transmit();
}
```

what happens when we invoke the following two statements?

```
transmitFreshData(telemetry, Time(60));
transmitFreshData(electrical, Time(120));
```

In the first statement, we transmit a bit stream consisting of only an `id` and a `timeStamp`. In the second statement, we transmit a bit stream consisting of an `id`, a `timeStamp`, and four other floating-point values. How is this so? Ultimately, the implementation of the function

`transmitFreshData` simply executes the statement `d.transmit()`, which does not explicitly distinguish the class of `d`.

The answer is that this behavior is due to polymorphism. Basically, *polymorphism* is a concept in type theory wherein a name (such as the parameter `d`) may denote instances of many different classes as long as they are related by some common superclass. Any object denoted by this name is thus able to respond to some common set of operations in different ways.

As Cardelli and Wegner note, "Conventional typed languages, such as Pascal, are based on the idea that functions and procedures, and hence operands, have a unique type. Such languages are said to be monomorphic, in the sense that every value and variable can be interpreted to be of one and only one type. Monomorphic programming languages may be contrasted with polymorphic languages in which some values and variables may have more than one type" [28]. The concept of polymorphism was first described by Strachey [29], who spoke of *ad hoc* polymorphism, by which symbols such as "+" could be defined to mean different things. Today, in modern programming languages, we call this concept *overloading*. For example, in C++, one may declare functions having the same names, as long as their invocations can be distinguished by their signatures, consisting of the number and types of their arguments (in C++, unlike Ada, the type of a function's returned value is not considered in overload resolution). Strachey also spoke of *parametric polymorphism*, which today we simply call *polymorphism*.

Without polymorphism, the developer ends up writing code consisting of large case or switch statements²⁵. For example, in a non-object-oriented programming language such as Pascal, we cannot create a hierarchy of classes for the various kinds of telemetry data; rather, we have to define a single, monolithic variant record encompassing the properties associated with all the kinds of data. To distinguish one variant from another, we have to examine the tag associated with the record. Thus an equivalent procedure to `transmitFreshData` might be written in Pascal as follows:

```
const
    Electrical = 1;
    Propulsion = 2;
    Spectrometer = 3;
...
procedure Transmit_Fresh_Data(The_Data : Data; The_Time :
Time); begin
    if (The_Data.Current_Time >= The_Time) then
        case The_Data.Kind of
            Electrical: Transmit_Electrical_Data(The_Data);
            Propulsion: Transmit_Propulsion_Data(The_Data);
            ...
        end
    end;
end;
```

To add another kind of telemetry data, we would have to modify the variant record and add it to every case statement that operated upon instances of is record. This is particularly error-prone, and, furthermore, adds instability to the design.

In the presence of inheritance, there is no need for a monolithic type, since we may separate different kinds of abstractions. As Kaplan and Johnson note, "Polymorphism is most useful when there are many classes with the same protocols" [30]. With polymorphism, large case statements are unnecessary, because each object implicitly knows its own type.

Inheritance without polymorphism is possible, but it is certainly not very useful. This is the situation in Ada, in which one can declare derived types, but because the language is monomorphic, the actual operation being called is always known at the time of compilation.

Polymorphism and late binding go hand in hand. In the presence of polymorphism, the binding of a method to a name is not determined until execution. In C++, the developer may control whether a member function uses early or late binding. Specifically, if the method is declared as virtual, then late binding is employed, and the function is considered to be polymorphic. If this virtual declaration is omitted, then the method uses early binding and thus can be resolved at the time of compilation. How an implementation selects a particular method for execution is described in the sidebar.

Inheritance and Typing Consider again the redefinition of the member `transmit`:

```
void ElectricalData::transmit()
{
    TelemetryData::transmit();
    transmit the voltages
    transmit the amperes
}
```

Most object-oriented programming languages permit the implementation of a subclass's method to directly invoke a method defined by some superclass. As this example shows, it is also quite common for the implementation of a redefined method to invoke the method of the same name defined by a parent class. In Smalltalk, one may invoke a method starting from the immediate ancestor class by using the keyword `super`; one may also refer to the object for which a method was invoked via the special variable `self`. In C++, one can invoke the method of any accessible ancestor by prefixing the method name with the name of the class, thus forming a *qualified name*, and one may refer to the object for which a method was invoked via the implicitly declared pointer named `this`.

In practice, a redefined method usually invokes a superclass method either before or after doing some other action. In this manner, subclass methods play the role of augmenting the behavior defined in the superclass²⁶.

In Figure 3-5, all of the subclasses are also subtypes of their parent class. For example, instances of `ElectricalData` are considered to be subtypes as well as subclasses of `TelemetryData`. The fact that typing parallels inheritance relationships is common to most strongly typed object-oriented programming languages, including C++. Because Smalltalk is largely typeless, or at most weakly typed, this issue is less of a concern.

The parallel between typing and inheritance is to be expected when we view the generalization/specialization hierarchies created through inheritance as the means of capturing the semantic connection among abstractions. Again, consider the declarations in C++:

```
TelemetryData telemetry;  
ElectricalData electrical(5.0, -5.0, 3.0, 7.0);
```

Invoking a Method

In traditional programming languages, invoking a subprogram is a completely static activity. In Pascal for example, for a statement that calls the subprogram `p`, a compiler will typically generate code that creates a new stack frame, places the proper arguments on the stack, and then changes the flow of control to begin executing the code associated with `p`. However, in languages that support some form of polymorphism, such as Smalltalk, and C++, invoking an operation may require a dynamic activity, because the class of the object being operated upon may not be known until runtime. Matters are even more interesting when we add inheritance to the situation. The semantics of invoking an operation in the presence of inheritance without polymorphism is largely the same as for a simple static subprogram call, but in the presence of polymorphism, we must use a much more sophisticated technique.

Consider the class hierarchy in Figure 3-6, which shows the base class `DisplayItem` along with three subclasses named `Circle`, `Triangle`, and `Rectangle`. `Rectangle` also has one subclass, named `SolidRectangle`. In the class `DisplayItem`, suppose that we define the instance variable `theCenter`

In CLOS, these different method roles are made explicit by declaring a method with the qualifiers `:before` and `:after`, as well as `:around`. A method without a qualifier is considered a *primary* method and does the central work of the desired behavior. *Before* methods and *after* methods augment the behavior of a primary method; they are called before and after the primary method, respectively. *Around* methods form a wrapper around a primary method, which may be invoked at some place inside the method by the `call-next-method` function.

(denoting the coordinates for the center of the displayed item), along with the following operations as in our earlier example:

- `draw` Draw the item.
- `move` Move the item.
- `location` Return the location of the item.

The operation `location` is common to all subclasses, and therefore need not be redefined, but we expect the operations `draw` and `move` to be redefined since only the subclasses know how to draw and move themselves.

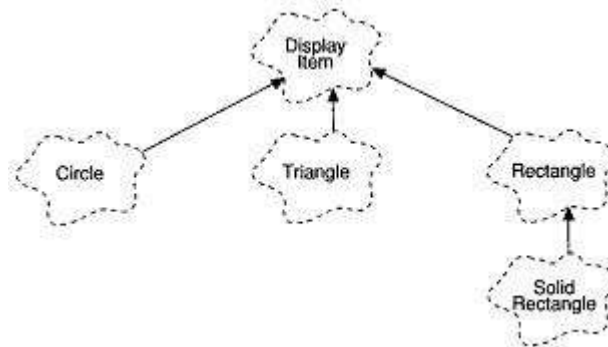


Figure 3-6
Displayitem Class Diagram

The class `Circle` must include the instance variable `theRadius` and appropriate operations to `set` and retrieve its value. For this subclass, the redefined operation `draw` draws a circle of the given radius, centered on `theCenter`. Similarly, the class `Rectangle` must include the instance variables `theHeight` and `theWidth`, along with appropriate operations to `set` and retrieve their values. For this subclass, the operation `draw` draws a rectangle with the given height and width, again centered on `theCenter`. The subclass `SolidRectangle` inherits all characteristics of the class `Rectangle`, but again redefines the behavior of the operation `draw`. Specifically, the implementation of `draw` for the class `SolidRectangle` first calls `draw` as defined in its superclass `Rectangle` (to draw the outline of the rectangle) and then fills in the shape.

Consider now the following code fragment:

```

DisplayItem* items[10];
...
for (unsigned index = 0; index < 10; index++)
    items[index]->draw();
    
```

The invocation of `draw` demands polymorphic behavior. Here, we find a heterogeneous array of items, meaning that the collection may contain pointers to objects of any of the `DisplayItem` subclasses. Suppose now that we have some client object that wishes to draw all of the items

found in this collection, as in the code fragment. Our approach is to iterate through the array and invoke the operation `draw` upon each object we encounter. In this situation, the compiler cannot statically generate code to invoke the proper `draw` operation, because the class of the object being operated upon is not known until runtime. Let's consider how various object-oriented programming languages deal with this situation.

Because Smalltalk is a typeless language, method dispatch is completely dynamic. When the client sends the message `draw` to an item found in the list, here is what happens:

The item object looks up the message in its class's message dictionary. If the message is found, the code for that locally defined method is invoked. If the message is not found, the search for the method continues in the superclass.

This process continues up the superclass hierarchy until the message is found, or until we reach the topmost base class, `object`, without finding the message. In the latter case, Smalltalk ultimately passes the message `doesNotUnderstand`, to signal an error.

The key to this algorithm is the message dictionary, which is part of each class's representation and is therefore hidden from the client. This dictionary is created when the class is created, and contains all the methods to which instances of this class may respond. Searching for the message is time-consuming; method lookup in Smalltalk takes about 1.5 times as long as a simple subprogram call. All production-quality Smalltalk implementations optimize method dispatch by supplying a cached message dictionary, so that commonly passed messages may be invoked quickly. Caching typically improves performance by 20%-30% [31].

The operation `draw` defined in the subclass `SolidRectangle` poses a special case. We said that its implementation of `draw` first calls `draw` as defined in the superclass `Rectangle`. In Smalltalk, we specify a superclass method by using the keyword `super`. Then, when we pass the message `draw` to `super`, Smalltalk uses the same method-dispatch algorithm as above, except that the search begins in the superclass of the object instead of its class.

Studies by Deutsch suggest that polymorphism is not needed about 85% of the time, so message passing can often be reduced to simple procedure calls [32]. Dulf notes that in such cases, the developer often makes implicit assumptions that permit an early binding of the object's class [33]. Unfortunately, typeless languages such as Smalltalk have no convenient means for communicating these implicit assumptions to the compiler.

More strongly typed languages such as C++ do let the developer assert such information. Because we want to avoid method dispatch wherever possible but must still allow for the occurrence of polymorphic dispatch, invoking a method in these languages proceeds a little differently than in Smalltalk.

In C++, the developer can decide if a particular operation is to be bound late by declaring it to be `virtual`; all other methods are considered to be bound early, and thus the compiler can

statically resolve the method call to a simple subprogram call. In our example, we declared `draw` as a virtual member function, and the method `location` as nonvirtual, since it need not be redefined by any subclass. The developer can also declare nonvirtual methods as inline, which avoids the subprogram call, and so trades off space for time.

To handle virtual member functions, most C++ implementations use the concept of a *vtable*, which is defined for each object requiring polymorphic dispatch, when the object is created (and thus when the class of the object is fixed). This table typically consists of a list of pointers to virtual functions. For example, if we create an object of the class `Rectangle`, then the vtable will have an entry for the virtual function `draw`, pointing to the closest implementation of `draw`. If, for example, the class `DisplayItem` included the virtual function `Rotate`, which was not redefined in the class `Rectangle`, then the vtable entry for `Rotate` would point to the implementation of `Rotate` in the class `DisplayItem`. In this manner, runtime searching is eliminated: referring to a virtual member function of an object is just an indirect reference through the appropriate pointer, which immediately invokes the correct code without searching [34],

The implementation of `draw` for the class `SolidRectangle` introduces a special case in C++ as well. To make the implementation of this method refer to the method `draw` in the superclass, C++ requires the use of the scope operator. Thus, one must write:

```
Rectangle::draw() ;
```

Studies by Stroustrup suggest that a virtual function call is just about as efficient as a normal function call [35]. In the presence of single inheritance, a virtual function call requires only about three or four more memory references than a normal function call; multiple inheritance adds only about five or six memory references.

Method dispatch in CLOS is complicated because of the presence of `:before`, `:after`, and `:around` methods. The existence of multiple polymorphism also complicates matters.

Method dispatch in CLOS normally uses the following algorithm:

- Determine the types of the arguments.
- Calculate the set of applicable methods.
- Sort the methods from most specific to most general, according to the object's class precedence list.
- Call all `:before` methods.
- Call the most specific primary method.
- Call all `:after` methods.
- Return the value of the primary method [36].

CLOS also introduces a metaobject protocol, whereby one may redefine the very algorithm used for generic dispatch (although in practice, one typically uses the predefined process). As Winston and Horn wisely point out, "The CLOS algorithm is complicated, however, and even wizard-level CLOS programmers try to get by without thinking about it, just as physicists try to get by with Newtonian mechanics rather than dealing with quantum mechanics" [37].

The following assignment statement is legal:

```
telemetry = electrical; // electrical is a subtype of telemetry
```

Although legal, this statement is also dangerous: any additional state defined for an instance of the subclass is sliced upon assignment to an instance of the superclass. In this example, the four member objects, `fuelCell1Voltage`, `fuelCell2Voltage`, `fuelCell1Amperes`, and `fuelCell2Amperes`, would not be copied, because the object denoted by the variable `telemetry` is an instance of the class `TelemetryData`, which does not have these members as part of its state.

The following statement is not legal:

```
electrical = telemetry; // Illegal: telemetry is not a subtype of  
                        electrical
```

To summarize, the assignment of object `x` to object `y` is possible if the type of `x` is the same as the type or a subtype of `y`.

Most strongly typed languages permit conversion of the value of an object from one type to another, but usually only if there is some superclass/subclass relationship between the two. For example, in C++ one can explicitly write conversion operators for a class using what are called *type casts*. Typically, as in our example, one uses implicit type conversion to convert an instance of a more specific class for assignment to a more general class. Such conversions are said to be *type-safe*, meaning that they are checked for semantic correctness at compilation time. We sometimes need to convert a variable of a more general class to one of a more specific class, and so must write an explicit type cast. However, such operations are not type-safe, because they can fail during execution time if the object being coerced is incompatible with the new type²⁷. Such conversions are actually not rare (although they should be avoided unless there is compelling reason), since the developer often knows the real types of certain objects. For example, in the absence of parameterized types, it is common practice to build classes such as sets and bags that represent collections of objects, and because we want to permit collections of instances of arbitrary classes, we typically define these collection classes to operate upon instances of some base class (a style much safer than the `void*` idiom used

earlier for the class `Queue`). Then, iteration operations defined for such a class would only know how to return objects of this base class. However, within a particular application, a developer might only place objects of some specific subclass of this base class in the collection. To invoke a class-specific operation upon objects visited during iteration, the developer would have to explicitly coerce each object visited to the expected type. Again, this operation would fail at execution time if an object of some unexpected type appeared in the collection.

Most strongly typed languages permit an implementation to better optimize method *dispatch* (lookup), often reducing the message to a simple subprogram call. Such optimizations are straightforward if the language's type hierarchy parallels its class hierarchy (as in C++). However, there is a dark side to unifying these hierarchies. Specifically, changing the structure or behavior of some superclass can affect the correctness of its subclasses. As Micallef states, "If subtyping rules are based on inheritance, then reimplementing a class such that its position in the inheritance graph is changed can make clients of that class type-incorrect, even if the external interface of the class remains the same" [38].

These issues lead us to the very foundations of inheritance semantics. As we noted earlier in this chapter, inheritance may be used to indicate sharing or to suggest some semantic connection. As stated another way by Snyder, "One can view inheritance as a private decision of the designer to 'reuse' code because it is useful to do so; it should be possible to easily change such a decision. Alternatively, one can view inheritance as making a public declaration that objects of the child class obey the semantics of the parent class, so that the child class is merely specializing or refining the parent class" [39]. In languages such as Smalltalk, and CLOS, these two views are indistinguishable. However, in C++ the developer has greater control over the implications of inheritance. Specifically, if we assert that the superclass of a given subclass is public (as in our example of the class `ElectricalData`), then we mean that the subclass is also a subtype of the superclass, since both share the same interface (and therefore the same structure and behavior). Alternately, in the declaration of a class, one may assert that a superclass is `private`, meaning that the structure and behavior of the superclass are shared but the subclass is not a subtype of the superclass²⁸. This means that for private superclasses, the public and protected members of the superclass become private members of the subclass, and hence inaccessible to lower subclasses. Furthermore, no subtype relationship between the subclass and its private superclass is formed, because the two classes no longer present the same interface to other clients.

Consider the following class declaration:

```
class InternalElectricalData : private ElectricalData
{ public:
    InternalElectricalData(float v1, float v2, float a1, float a2);

    virtual ~InternalElectricalData();

    ElectricalData::currentPower;
};
```

In this declaration, methods such as `transmit` are not visible to any clients of this class, because `ElectricalData` is declared to be private superclass. Because `InternalElectricalData` is not a subtype of `ElectricalData`, this also means that we cannot assign instances of `InternalElectricalData` to objects of the superclass, as we can for classes using public superclasses. Lastly, note that we have made the member function `currentPower` visible by explicitly naming the function. Without this explicit naming, it would be treated as private. As you would expect, the rules of C++ prohibit one from making a member in a subclass more visible than it is in its superclass. Thus, the member object `timeStamp`, declared as a protected member in the class `TelemetryData`, could not be made public by explicit naming as done for `currentPower`.

In languages such as Ada, the equivalent of this distinction can be achieved by using derived types versus subtypes. Specifically, a subtype of a type defines no new type, but only a constrained subtype, while a derived type defines a new, incompatible type, which shares the same representation as its parent type.

As we discuss in a later section, there is great tension between inheritance for reuse and aggregation.

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 object-oriented programming languages is still a topic of great debate. In our experience, we find multiple inheritance to be like a parachute: you don't always need it, but when you do, you're really happy to have it on hand.

Consider for a moment how one might organize various assets such as savings accounts, real estate, stocks, and bonds. Savings accounts and checking accounts are both kinds of assets typically managed by a bank, so we might classify both of them as kinds of bank accounts, which in turn are kinds of assets. Stocks and bonds are managed quite differently than bank accounts, so we might classify stocks, bonds, mutual funds, and the like as kinds of securities which in turn are also kinds of assets.

However, there are many other equally satisfactory ways to classify savings accounts, real estate, stocks, and bonds. For example, in some contexts, it may be useful to distinguish insurable items such as real estate and certain bank accounts (which, in the United States, are

insured up to certain limits by the Federal Depositors Insurance Corporation). It may also be useful to identify assets that return a dividend or interest, such as savings accounts, checking accounts, and certain stocks and bonds.

Unfortunately, single inheritance is not expressive enough to capture this lattice of relationships, so we must turn to multiple inheritance²⁹. Figure 3-7 illustrates such a class structure. Here we see that the class `Security` is a kind of `Asset` as well as a kind of `InterestBearingItem`. Similarly, the class `BankAccount` is a kind of `Asset`, as well as a kind of

`InsurableItem` and `InterestBearingItem`.

To capture these design decisions in C++, we might write the following (highly elided)

declarations. First, we start with the base classes:

```
class Asset ...  
class InsurableItem ...  
class InterestBearingItem ...
```

Next we have various intermediate classes, each of which has multiple superclasses:

```
class BankAccount : public Asset,  
                    public InsurableItem,  
                    public InterestBearingItem ...  
class RealEstate : public Asset,  
                  public InsurableItem ...  
class Security : public Asset,  
                public InterestBearingItem ...
```

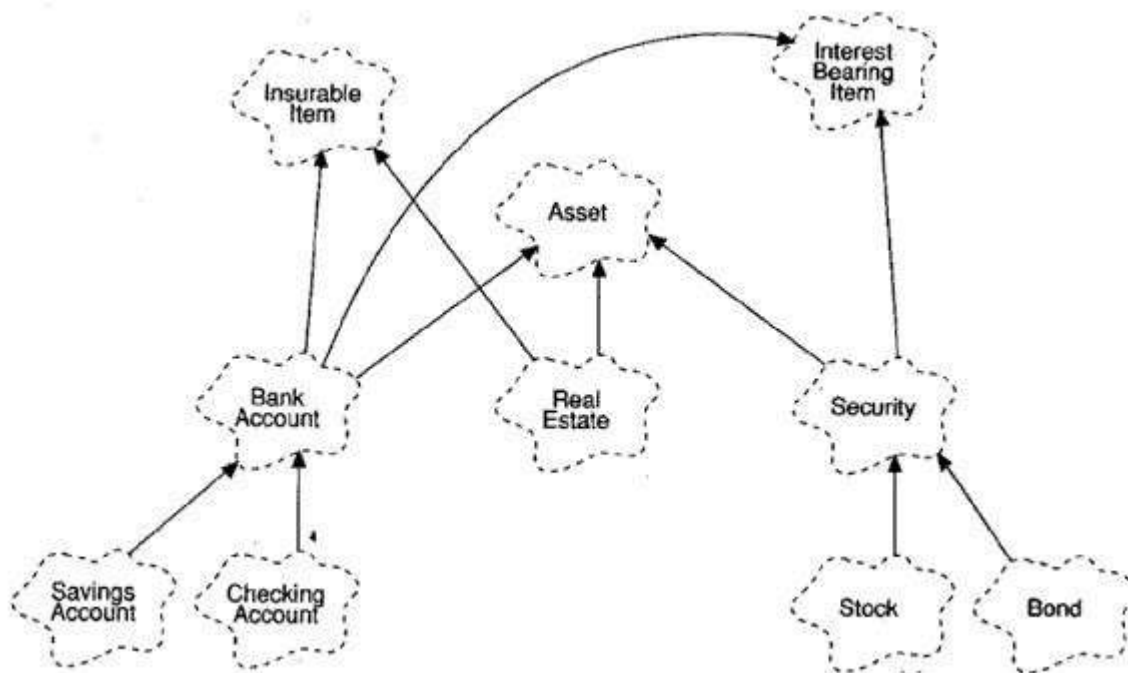


Figure 3-7
Multiple Inheritance

And finally, we have the remaining leaf classes:

```
class SavingsAccount : public BankAccount ...  
class CheckingAccount : public BankAccount ...  
  
class Stock : public Security ...  
class Bond : public Security ...
```

Designing a suitable class structure involving inheritance, and especially involving multiple inheritance, is a difficult task. As we explain in Chapter 4, this is often an incremental and iterative process. Two problems present themselves when we have multiple inheritance: How do we deal with name collisions from different superclasses, and how do we handle repeated inheritance?

Name collisions are possible when two or more different superclasses use the same name for some element of their interfaces, such as instance variables and methods. For example, suppose that the classes `InsurableItem` and `Asset` both have attributes named `presentValue`, denoting the present value of the item. Since the class `RealEstate` inherits from both of these classes, what does it mean to inherit two operations with the same name? This in fact is the key difficulty with multiple inheritance: clashes may introduce ambiguity in the behavior of the multiply inherited subclass.

There are three basic approaches to resolving this kind of clash. First, the language semantics might regard such a clash as illegal, and reject the compilation of the class. This is the approach taken by languages such as Smalltalk and Eiffel. In Eiffel, however, it is possible to

rename items so that there is no ambiguity. Second, the language semantics might regard the same name introduced by different classes as referring to the same attribute, which is the approach taken by CLOS. Third, the language semantics might permit the clash, but require that all references to the name fully qualify the source of its declaration. This is the approach taken by C++³⁰.

The second problem is repeated inheritance, which Meyer describes as follows: "One of the delicate problems raised by the presence of multiple inheritance is what happens when a class is an ancestor of another in more than one way. If you allow multiple inheritance into a language, then sooner or later someone is going to write a class *D* with two parents *B* and *C*, each of which has a class *A* as a parent - or some other situation in which *D* inherits twice (or more) from *A*. This situation is called repeated inheritance and must be dealt with properly"

[41]. As an example, suppose that we define the following (ill-conceived) class:

```
class MutualFind : public Stock,  
                  public Bond ...
```

This class introduces repeated inheritance of the class *Security*, which is a superclass of both

Stock and *Bond*.

There are three approaches to dealing with the problem of repeated inheritance. First, we can treat occurrences of repeated inheritance as illegal. This is the approach taken by Smalltalk and Eiffel (with Eiffel again permitting renaming to disambiguate the duplicate references). Second, we can permit duplication of superclasses, but require the use of fully qualified names to refer to members of a specific copy. This is one of the approaches taken by C++. Third, we can treat multiple references to the same class as denoting the same class. This is the approach taken by C++ when the repeated superclass is introduced as a virtual base class. A virtual base class exists when a subclass names another class as its superclass and marks that superclass as *virtual*, to indicate that it is a shared class. Similarly, in CLOS repeated classes are shared, using a mechanism called the *class precedence list*. This list, calculated whenever a new class is introduced, includes the class itself and all of its superclasses, without duplication, and is based upon the following rules:

A class always has precedence over its superclass.

Each class sets the precedence order of its direct superclasses [42].

In this approach, the inheritance graph is flattened, duplicates are removed, and the resulting hierarchy is resolved using single inheritance [43]. This is akin to the computation of a topological sorting of classes. If a total ordering of classes can be calculated, then the class that introduces the repeated inheritance is accepted. Note that this total ordering may be unique, or there may be several possible orderings (and a deterministic algorithm will always select

one such ordering). If no ordering can be found (for example, when there are cycles in the class dependencies), the class is rejected.

The existence of multiple inheritance gives rise to a style of classes called *mixins*. Mixins derive from the programming culture surrounding the language Flavors: one would combine ("mix in") little classes to build classes with more sophisticated behavior. As Hendler observes, "A mixin is syntactically identical to a regular class, but its intent is different. The purpose of such a class is solely to ... [add] functions to other flavors [classes] - one never creates an instance of a mixin" [44]. In Figure 3-7, the classes `InsurableItem` and `InterestBearingItem` are mixins. Neither of these classes can stand alone; rather, they are used to augment the meaning of some other class³¹. Thus, we may define a mixin as a class that embodies a single, focused behavior and is used to augment the behavior of some other class via inheritance. The behavior of a mixin is usually completely orthogonal to the behavior of the classes with which it is combined. A class that is constructed primarily by inheriting from mixins and does not add its own structure or behavior is called an *aggregate class*.

Multiple Polymorphism Consider again the following member function declared for the class `DisplayItem`:

```
virtual void draw();
```

The purpose of this operation is to draw the given object in some context. This operation is declared as `virtual` and is therefore polymorphic, meaning that whenever we invoke this operation for a particular object, the proper subclass's implementation of this operation will be called, using an algorithm for method dispatch as described in the sidebar. This is an example of single polymorphism, meaning that the method is specialized (is polymorphic) on exactly one parameter, namely, the object for which the operation is invoked.

Suppose now that we need a slightly different behavior, depending upon the exact display device we use. In one case, we would want the method `draw` to display a high-resolution graphical representation; in another, we would want it to print a representation quickly, and so would draw only a very coarse image. We could declare two distinct although very similar operations, such as `drawGraphic` and `drawText`. This is not entirely satisfying, however, because this solution does not scale very well: introducing yet another drawing context requires us to add a new operation to every class in the `DisplayItem` hierarchy.

In languages such as CLOS, we can write operations called *multimethods* that are polymorphic on more than one parameter (such as the display item and the display device). In languages that support only single polymorphism (such as C++), we can fake this multiple polymorphic behavior by using an idiom called *double dispatching*.

First, we might define a hierarchy of display devices, rooted in the class `DisplayDevice`. Next, we would rewrite the `DisplayItem` operation as follows:

³¹

```
virtual void draw(DisplayDevice&);
```

In the implementation of this method, we would invoke drawing operations that are polymorphic on the given actual `DisplayDevice` parameter - thus the name double dispatch: `draw` first exhibits polymorphic behavior based upon the object's exactly subclass of `DisplayItem`, and then next exhibits polymorphic behavior based upon the argument's exact subclass of `DisplayDevice`.

This idiom can be extended to any degree of polymorphic dispatch.

Aggregation

Example Aggregation relationships among classes have a direct parallel to aggregation relationships among the objects corresponding to these classes. For example, consider again the declaration of the class `TemperatureController`:

```
class TemperatureController {
public:
    TemperatureController(Location);
    ~TemperatureController();

    void process(const TemperatureRamp&);

    Minute schedule(const TemperatureRamp&) const;

private:
    Heater h;
};
```

As we show in Figure 3-8, the class `TemperatureController` denotes the whole, and an instance of the class `Heater` is one of its parts. This corresponds exactly to the aggregation relationship among the instances of these classes illustrated in Figure 3-3.

Physical Containment In the case of the class `TemperatureController`, we have aggregation as containment *by value*, a kind of physical containment meaning that the `Heater` object does not exist independently of its enclosing `TemperatureController` instance. Rather the lifetimes of these two objects are intimately connected: when we create an instance of `TemperatureController`, we also create an instance of the class `Heater`. When we destroy our `TemperatureController` object, by implication we also destroy the corresponding `Heater` object.

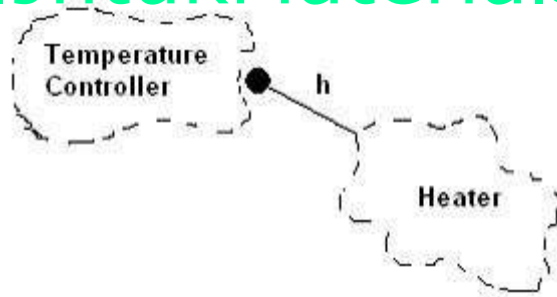


Figure 3-8
Aggregation

A less direct kind of aggregation is also possible, called containment *by reference*. For example, we might replace the private part of the class `TemperatureController` with the following declaration³²:

```
Heater* h;
```

In this case, the class `TemperatureController` still denotes the whole, and an instance of the class `Heater` is still one of its parts, although that part must now be accessed indirectly. Hence, the lifetimes of these two objects are not so tightly coupled as before: we may create and destroy instances of each class independently. Furthermore, because it is possible for the part to be structurally shared, we must decide upon some policy whereby its storage is properly created and reclaimed by only one agent that shares references to that part.

Aggregation asserts a direction to the whole/part relationship. For example, the `Heater` object is a part of the `TemperatureController` object, and not vice versa. Containment by value may not be cyclic (that is, both objects may not physically be parts of one another), although containment by reference may be (each object may hold a pointer to the other)³³.

Of course, as we described in an earlier example, aggregation need not require physical containment, as implied through containment by value or by reference. For example, although shareholders own stocks, a shareholder does not physically contain the owned stocks. Rather, the lifetimes of these objects may be completely independent, although there is still conceptually a whole/part relationship (each share is always a part of the shareholder's assets), and thus our representation of this aggregation can be very indirect. For example, we might declare the class `Shareholder`, whose state includes a key to a database table that we may use to look up the shares owned by a particular shareholder. This is still aggregation, although not physical containment. Ultimately, the litmus test for aggregation is this: If and only if there exists a whole/part relationship between two objects, we must have an aggregation relationship between their corresponding classes.

Multiple inheritance is often confused with aggregation. In fact, in C++ protected or private inheritance can easily be replaced with `protected` or `private` aggregation of an instance of the superclass, with no loss in semantics. When considering inheritance versus aggregation, remember to apply the litmus test for each. If you cannot honestly affirm that there is an "is a" relationship between two classes,

then aggregation or some other relationship should be used instead of inheritance.

Using

Example Our earlier 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`:

```
class TemperatureController {
public:
    TemperatureController(Location);
    ~TemperatureController();

    void process(const TemperatureRamp&);

    Minute schedule(const TemperatureRamp&) const;

private:
    Heater h;
};
```

The class `TemperatureRamp` appears as part of the signature in certain member functions, and thus we can say that `TemperatureController` uses the services of the class `TemperatureRamp`.

Clients and Suppliers "Using" relationships among classes parallel the peer-to-peer links among the corresponding instances of these classes. Whereas an association denotes a bidirectional semantic connection, a "using" relationship is one possible refinement of an association, whereby we assert which abstraction is the client and which is the supplier of certain services. We illustrate such a client/supplier "using" relationship in Figure 3-9³⁴.

Actually, one class may use another in a variety of ways. In our example, the `TemperatureController` uses the `TemperatureRamp` in the signature of its interface. The `TemperatureController` might also use another class such as `Predictor` in its implementation of the member function `schedule`. This is not an assertion of a whole/part relationship: an instance of the `Predictor` class is only used by and is not a part of the `TemperatureController` instance. Typically,

such a "using" relationship manifests itself by the implementation of some operation declaring a local object of the used class.

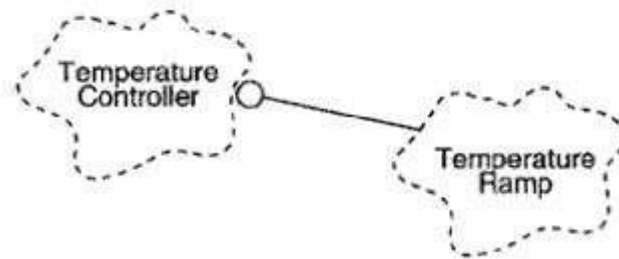


Figure 3-9
The "Using" Relationship

Strict "using" relationships are occasionally too confining because they allow the client access only to the public interface of the supplier. Sometimes, for tactical reasons, we must break our encapsulation of these abstractions, which is the very purpose of the `friend` concept in C++.

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++:

```
template<class Item>
class Queue {
public:
    Queue();
    Queue(const Queue<Item>&);
    virtual ~Queue();

    virtual Queue<Item>& operator=(const Queue<Item>&);
    virtual int operator==(const Queue<Item>&) const; int
    operator!=(const Queue<Item>&) const;
    virtual void clear();
    virtual void append(const Item&);
    virtual void pop();
    virtual void remove(int at);

    virtual int length() const;
    virtual int isEmpty() const;
```

```
virtual const Item& front() const;
virtual int location(const void*);

protected:
    ...
}
```

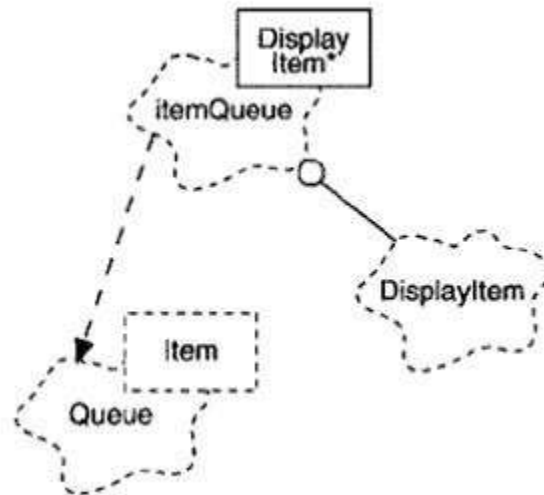


Figure 3-10
Instantiation

Note that in this declaration we no longer append and retrieve objects via `void*` (which is not type-safe); we do so via the class `Item` declared as a template argument.

A parameterized class cannot have instances unless we first instantiate it. For example, we might declare two concrete queue objects, a queue of integers, and a queue of display items:

```
Queue<int> intQueue;
Queue<DisplayItem*> itemQueue;
```

The objects `intQueue` and `itemQueue` are instances of distantly different classes, and are not even united by any common superclass, although they both derive from the same parameterized class. For reasons that we describe further in Chapter 9, we use a pointer to the class `DisplayItem` in the second instantiation, so that objects of a `DisplayItem` subclass placed in the queue will not be sliced, but will preserve their polymorphic behavior.

These instantiations are type-safe. C++'s typing rules will reject any statements that attempt to append or retrieve anything other than integers from `intQueue` and anything but instances of `DisplayItem` or its subclasses from `itemQueue`.

Figure 3-10 illustrates the relationships among the parameterized class `Queue`, its instantiation for `DisplayItem`, and its corresponding instance `itemQueue`.

Genericity There are four basic ways to build classes such as the parameterized class `Queue`. First, we can use macros. This is the style one had to use in earlier versions of C++, but as

Stroustrup observes, this "approach does not work well except on a small scale" [45] because maintaining macros is clumsy and outside the semantics of the language; furthermore, each instantiation results in a new copy of the code. Second, we can take the approach used by Smalltalk and rely upon inheritance and late binding [46]. With this approach, we may build only heterogeneous container classes, because there is no way to assert the specific class of the container's elements; every item is treated as if it were an instance of some distant base class. Third, we may take an approach commonly used in languages such as Object Pascal, which are strongly typed, support inheritance, but do not support any form of parameterized classes. Here, we build generalized container classes, as in Smalltalk, but then use explicit typechecking code to enforce the convention that the contents are all of the same class, which is asserted when the container object is created. This approach has significant runtime overhead. Fourth, we may take the approach first introduced by CLU and provide a direct mechanism for parameterizing classes, as in our example. A *parameterized class* (also known as a *generic class*) is one that serves as a template for other classes - a template that may be parameterized by other classes, objects, and/or operations. A parameterized class must be instantiated (that is, its parameters must be filled in) before objects can be created. C++ and Eiffel both support generic class mechanisms.

In Figure 3-10, note that to instantiate the class `Queue`, we must also use the class `DisplayItem`. Indeed, instantiation relationships almost always require some "using" relationships, which make visible the actual classes used to fill in the template.

Meyer has pointed out that inheritance is a more powerful mechanism than genericity and that much of the benefit of genericity can be achieved through inheritance, but not vice versa

In practice, we find it helpful to use a language that supports both inheritance and parameterized classes.

Parameterized classes may be used for much more than building container classes. As Stroustrup points out, "Type parameterization will allow arithmetic functions to be parameterized over their basic number type so that programmers can (finally) get a uniform way of dealing with integers, single-precision floating-point numbers, double-precision floating point-numbers, etc." .

From a design perspective, parameterized classes are also useful in capturing certain design decisions about the protocol of a class. Whereas a class definition exports the operations that one may perform upon instances of that class, the arguments of a template serve to import classes (and values) that provide a specific protocol. In C++, this conformance checking is done at compilation time, when expanding the instantiation. For example, we might declare an ordered queue class that represents collections of objects that are sorted according to some criteria. This parameterized class must rely upon some class `Item`, as before, but also expects `Item` to provide some ordering operation. By parameterizing the class in this manner, we make it more loosely coupled: we can match the formal argument `Item` with any class that provides this ordering function. In this sense, we may define a parameterized class as one that denotes a family of classes whose

structure and behavior are defined independently of their formal class parameters.

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.

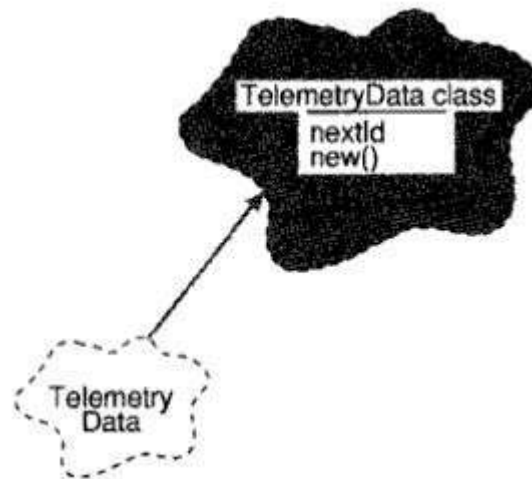


Figure 3-11
Metaciasses

Robson motivates the need for metaclasses by noting that "in a system under development, a class provides an interface for the programmer to interface with the definition of objects. For this use of classes, it is extremely useful for them to be objects, so that they can be manipulated in the same way as all other descriptions" .

In languages such as Smalltalk, the primary purpose of a metaclass is to provide class variables (which are shared by all instances of the class) and operations for initializing class variables and for creating the metaclass's single instance . By convention, a Smalltalk metaclass typically contains examples that show the use of the metaclass's class. For example, as shown in Figure 3-11, we might in Smalltalk define a class variable `nextID` for the metaclass of `TelemetryData`, whose purpose is to assist in generating distinct id's upon the creation of each instance of `TelemetryData`. Similarly, we might define an operation for creating new instances of the class, which perhaps generates them from some preallocated pool of storage.

Although C++ does not explicitly support metaclasses, its constructor and destructor semantics serve the purpose of metaclass creation operations. Also,

C++ has provisions for class variables and metaclass operations. Specifically, in C++ one may declare a member

object or a member function as *static*, meaning that the member is shared by all instances of the class. Static member objects in C++ are equivalent to Smalltalk's class variables, and static member functions are equivalent to metaclass operations in Smalltalk.

As we have mentioned, support for metaclasses in CLOS is even more powerful than in Smalltalk. Through the use of metaclasses, one may redefine the very semantics of elements such as class precedence, generic functions, and methods. The primary benefit of this facility is that it permits experimentation alternate object-oriented programming paradigms and facilitates the construction of software development tools, such as browsers.

In CLOS, the predefined class named `standard-class` is the metaclass of all defined via `defclass`. This metaclass defines the method `make-instance`, which implements the semantics of how instances are created. `standard-class` also defines the algorithm for computing the class precedence list. CLOS allows the behavior of both of these methods to be redefined.

Methods and generic functions may also be treated as objects in CLOS. Because they are somewhat different than the usual kinds of objects, class objects, method objects, and generic function objects are collectively called *metaobjects*. Each method is an instance of the predefined class `standard-method`, and each generic function is treated as an instance of the class `standard-generic-function`. Because the behavior of these predefined classes may be redefined, it is possible to change the meanings of methods and generic functions.

The Interplay of Classes and Objects

Relationships Between Classes and Objects

Classes 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 lifetime 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 knowledge 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.

On Building Quality Classes and Objects

Measuring the Quality of an Abstraction

Ingalls suggests that "a system should be built with a minimum set of unchangeable parts; those parts should be as general as possible; and all parts of the system should be held in a uniform framework" [51]. With object-oriented development, these parts are the classes and objects that make up the key abstractions of the system, and the framework is provided by its mechanisms.

In our experience, the design of classes and objects is an incremental, iterative process. Frankly, except for the most trivial abstractions, we have never been able to define a class exactly right the first time. As Chapters 4 and 7 explain, it takes time to smooth the conceptual jagged edges of our initial abstractions. Of course, there is a cost to refining these abstractions, in terms of recompilation, understandability, and the integrity of the fabric of our system design. Therefore, we want to come as close as we can to being right the first time.

How can one know if a given class or object is well designed? We suggest five meaningful metrics:

- Coupling
- Cohesion
- Sufficiency
- Completeness

Coupling is a notion borrowed from structured design, but with a liberal interpretation it also applies to object-oriented design. Stevens, Myers, and Constantine define coupling as "the measure of the strength of association established by a connection from one module to another. Strong coupling complicates a system since a module is harder to understand, change, or correct by itself if it is highly interrelated with other modules. Complexity can be reduced by designing systems with the weakest possible coupling between modules" [52]. A counterexample to good coupling is given by Page-Jones, in his description of a modular stereo system in which the power supply is located in one of the speaker cabinets .

Coupling with regard to modules is still applicable to object-oriented analysis and design, but coupling with regard to classes and objects is equally important. However, there is tension between the concepts of coupling and inheritance, because inheritance introduces significant coupling. On the one hand, weakly coupled classes are desirable; on the other hand, inheritance- which tightly couples superclasses and their subclasses- helps us to exploit the commonality among abstractions.

The idea of cohesion also comes from structured design. Simply stated, cohesion measures the degree of connectivity among the elements of a single module (and for object-oriented design, a single class or object). The least desirable form of cohesion is coincidental cohesion, in which entirely unrelated abstractions are thrown into the same class or module. For example, consider a class comprising the abstractions of dogs and spacecraft, whose behaviors are quite unrelated. The most desirable form of cohesion is functional cohesion, in which the elements of a class or module all work together to provide some well-bounded behavior. Thus, the class `Dog` is functionally cohesive if its semantics embrace the behavior of a dog, the whole dog, and nothing but the dog.

Closely related to the ideas of coupling and cohesion are the criteria that a class or module should be sufficient, complete, and primitive. By *sufficient*, we mean that the class or module captures enough characteristics of the abstraction to permit meaningful and efficient interaction. To do otherwise renders the component useless. For example, if we are designing the class `set`, it is wise to include an operation that removes an item from the set, but our wisdom is futile if we neglect an operation that adds an item. In practice, violations of this characteristic are detected very early; such shortcomings rise up almost every time we build a client that must use this abstraction. By *complete*, we mean that the interface of the class or module captures all of the meaningful characteristics of the abstraction. Whereas sufficiency implies a minimal interface, a complete interface is one that covers all aspects of the abstraction. A complete class or module is thus one whose interface is general enough to be commonly usable to any client. Completeness is a subjective matter, and it can be overdone. Providing all meaningful operations for a particular abstraction overwhelms the user and is generally unnecessary, since many high-level operations can be composed from low-level ones. For this reason, we also suggest that classes and modules be primitive. *Primitive* operations are those that can be efficiently implemented only if given access to the underlying representation of the abstraction. Thus, adding an item to a set is primitive, because to

implement this operation `Add`, the underlying representation must be visible. On the other hand, an operation that adds four items to a set is not primitive, because it can be implemented just as efficiently upon the more primitive `Add` operation, without having access to the underlying representation. Of course, efficiency is also a subjective measure. An operation is indisputably primitive if we can implement it only through access to the underlying representation. An operation that could be implemented on top of existing primitive operations, but at the cost of significantly more computational resources, is also a candidate for inclusion as a primitive operation.

Choosing Operations

Functional Semantics Crafting the interface of a class or module is plain hard work. Typically, we make a first attempt at the design of a class, and then, as we and others create clients, we find it necessary to augment, modify, and further refine this interface. Eventually, we may discover patterns of operations or patterns of abstractions that lead us to invent new classes or to reorganize the relationships among existing ones.

Within a given class, it is our style to keep all operations primitive, so that each exhibits a small, well-defined behavior. We call such methods *fine-grained*. We also tend to separate methods that do not communicate with one another. In this manner, it is far easier to construct subclasses that can meaningfully redefine the behavior of their superclasses. The decision to contract out a behavior to one versus many methods may be made for two competing reasons: lumping a particular behavior in one method leads to a simpler interface but larger, more complicated methods; spreading a behavior across methods leads to a more complicated interface, but simpler methods. As Meyer observes, "A good designer knows how to find the appropriate balance between too much contracting, which produces fragmentation, and too little, which yields unmanageably large modules" [54].

It is common in object-oriented development to design the methods of a class as a whole, because all these methods cooperate to form the entire protocol of the abstraction. Thus, given some desired behavior, we must decide in which class to place it. Halbert and O'Brien offer the following criteria to be considered when making such a decision:

- | | |
|----------------------------|---|
| • Reusability | Would this behavior be more useful in more than one context? |
| • Complexity | How difficult is it to implement the behavior? |
| • Applicability | How relevant is the behavior to the type in which it might be placed? |
| • Implementation knowledge | Does the behavior's implementation depend upon the internal details of a type |

We usually choose to declare the meaningful operations that we may perform upon an object as methods in the definition of that object's class (or superclass). In languages such as C++ and CLOS, however, we may also declare such operations as free subprograms, which we then group in class utilities. In C++ terminology, a *free subprogram* is a nonmember function.

Because free subprograms cannot be redefined as methods can, they are less general. However, utilities are helpful in keeping a class primitive and in reducing the coupling among classes, especially if these higher-level operations involve objects of many different classes.

Time and Space Semantics Once we have established the existence of a particular operation and defined its functional semantics, we must decide upon its time and space semantics. This means that we must specify our decisions about the amount of time it takes to complete an operation and the amount of storage it needs. Such decisions are often expressed in terms of best, average, and worst cases, with the worst case specifying an upper limit on what is acceptable.

Earlier, we also mentioned that whenever one object passes a message to another across a link, the two objects must be synchronized in some manner. In the presence of multiple threads of control, this means that message passing is much more than a subprogram-like dispatch. In most of the languages we use, synchronization among objects is simply not an issue, because our programs contain exactly one thread of control, meaning that all objects are sequential. We speak of message passing in such situations as simple, because its semantics are most akin to simple subprogram calls. However, in languages that support concurrency³⁵ we must concern ourselves with more sophisticated forms of message passing, so as to avoid the problems created if two threads of control act upon the same object in unrestrained ways. As we described earlier, objects whose semantics are preserved in the presence of multiple threads of control are either guarded or synchronized objects.

We have found it useful in some circumstances to express concurrency semantics for each individual operation as well as for the object as a whole, since different operations may require different kinds of synchronization. Message passing may thus take one of the following forms:

- Synchronous An operation commences only when the sender has initiated the action and the receiver is ready to accept the message; the sender and receiver will wait indefinitely until both parties are ready to proceed.
- Balking The same as synchronous, except that the sender will abandon the operation if the receiver is not immediately ready.
- Timeout The same as synchronous, except that the sender will only wait for a specified amount of time for the receiver to be ready.
- Asynchronous A sender may initiate an action regardless of whether the receiver is expecting the message.

The form can be selected on an operation-by-operation basis, but only after the functional semantics of the operation have been decided upon.

Choosing Relationships

Collaborations Choosing the relationships among classes and among objects is linked to the selection of operations. If we decide that object *x* sends message *m* to object *y*, then either directly or indirectly, *y* must be accessible to *x*; otherwise, we could not name the operation *m* in the implementation of *x*. By *accessible*, we mean the ability of one abstraction to see another and reference resources in its outside view. Abstractions are accessible to one another only where their scopes overlap and only where access rights are granted (for example, private parts of a class are accessible only to the class itself and its friends). Coupling is thus a measure of the degree of accessibility.

One useful guideline in choosing the relationships among objects is called the Law of Demeter, which states that "the methods of a class should not depend in any way on the structure of any class, except the immediate (top-level) structure of their own class. Further, each method should send messages to objects belonging to a very limited set of classes only"

The basic effect of applying this law is the creation of loosely coupled classes, whose implementation secrets are encapsulated. Such classes are fairly unencumbered, meaning that to understand the meaning of one class, you need not understand the details of many other classes.

In looking at the class structure of an entire system, we may find that its inheritance hierarchy is either wide and shallow, narrow and deep, or balanced. Class structures that are wide and shallow usually represent forests of free-standing classes that can be mixed and matched [57]. Class structures that are narrow and deep represent trees of classes that are related by a common ancestor. There are advantages and disadvantages to each approach. Forests of classes are more loosely coupled, but they may not exploit all the commonality that exists. Trees of classes exploit this commonality, so that individual classes are smaller than in forests. However, to understand a particular class, it is usually necessary to understand the meaning of all the classes it inherits from or uses. The proper shape of a class structure is highly problem-dependent.

We must make similar trade-offs among inheritance, aggregation, and using relationships. For example, should the class *car* inherit, contain, or use the classes named *Engine* and *Wheel*? In this case, we suggest that an aggregation relationship is more appropriate than an inheritance relationship. Meyer states that between the classes *A* and *B*, "inheritance is appropriate if every instance of *B* may also be viewed as an instance of *A*. The client relationship is appropriate when every instance of *B* simply possesses one or more attributes of *A*". From another perspective, if the behavior of an object is more than the sum of its individual parts, then creating an aggregation relationship rather than an inheritance relationship between the appropriate classes is probably superior.

Mechanisms and Visibility Deciding upon the relationship among objects is mainly a matter of designing the mechanisms whereby these objects interact. The question the developer must ask is simply, Where does certain knowledge go? For example, in a manufacturing plant, materials (called *lots*) enter manufacturing cells to be processed. As they enter certain cells, we must notify the room's manager to take appropriate action. We now have a design choice: is the entry of a lot into a room an operation upon the room, an operation upon the lot, or an operation upon both? If we decide that it is an operation upon the room, then the room must be visible to the lot. If we decide that it is an operation upon the lot, then the lot must be visible to the room, because the lot must know what room it is in. Lastly, if we consider this to be an operation upon both the room and the lot, then we must arrange for mutual visibility. We must also decide on some visibility relationship between the room and the manager (and not the lot and the manager); either the manager must know the room it manages, or the room must know of its manager.

During the design process, it is occasionally useful to state explicitly how one object is visible to another. There are four fundamental ways that object *x* may be made visible to object *y*:

- The supplier object is global to the client.

- The supplier object is a parameter to some operation of the client.

- The supplier object is a part of the client object.

- The supplier object is a locally declared object in the scope of the object diagram.

A variation upon each of these is the idea of shared visibility. For example, *y* might be a part of *x*, but *y* might also be visible to other objects in different ways. In Smalltalk, this kind of visibility usually represents a dependency between two objects. Shared visibility involves structural sharing, meaning that one object does not have exclusive access to another: the shared object's state may be altered via more than one path.

Choosing implementations

Only after we stabilize the outside view of a given class or object do we turn to its inside view. This perspective involves two different decisions: a choice of representation for a class or object and the placement of the class or object in a module.

Representation The representation of a class or object should almost always be one of the encapsulated secrets of the abstraction. This makes it possible to change the representation (for example, to alter the time and space semantics) without violating any of the functional assumptions that clients may have made. As Wirth wisely states, "The choice of representation is often a fairly difficult one, and it is not uniquely determined by the facilities available. It must always be taken in light of the operations that are to be performed upon the data". For example, given a class whose objects denote a set of flight-plan information, do we optimize the representation for fast searching or for fast insertion and

deletion? We cannot optimize for both, so our choice must be based upon the expected use of these objects. Sometimes it is not easy to choose, and we end up with families of classes whose interfaces

are virtually identical but whose implementations are radically different, in order to provide different time and space behavior.

One of the more difficult trade-offs when selecting the implementation of a class is between computing the value of an object's state versus storing it as a field. For example, suppose we have the class `Cone`, which includes the method `Volume`. Invoking this method returns the volume of the object. As part of the representation of this class, we are likely to use fields for the height of the cone and the radius of its base. Should we have an additional field in which we store the volume of the object, or should the method `Volume` just calculate it every time [60]? If we want this method to be fast, we should store the volume as a field. If space efficiency is more important to us, we should calculate the value. Which representation is better depends entirely upon the particular problem. In any case, we should be able to choose an implementation independently of the class's outside view; indeed, we should even be able to change this representation without its clients caring.

Packaging Similar issues apply to the declaration of classes and objects within modules. For Smalltalk, this is not an issue, because there is no concept of a module within the language. It is a different matter for languages such as Object Pascal, C++, CLOS, and Ada, which support the notion of the module as a separate language construct. The competing requirements of visibility and information hiding usually guide our design decisions about where to declare classes and objects. Generally, we seek to build functionally cohesive, loosely coupled modules. Many nontechnical factors influence these decisions, such as matters of reuse, security, and documentation. Like the design of classes and objects, module design is not to be taken lightly. As Parnas, Clements, and Weiss note with regard to information hiding, "Applying this principle is not always easy. It attempts to minimize the expected cost of software over its period of use and requires that the designer estimate the likelihood of changes. Such estimates are based on past experience and usually require knowledge of the application area as well as an understanding of hardware and software technology" .

www.JntukMaterials.com

Classification

Classification is the means whereby we order knowledge. In object-oriented design, recognizing the sameness among things allows us to expose the commonality within key abstractions and mechanisms, and eventually leads us to smaller and simpler architectures. Unfortunately, there is no golden path to classification. To the reader accustomed to finding cookbook answers, we unequivocally state that there are no simple recipes for identifying classes and objects. There is no such thing as the "perfect" class structure, nor the "right" set of objects. As in any engineering discipline, our design choices are a compromise shaped by many competing factors.

At a conference on software engineering, several developers were asked what rules they applied to identify classes and objects. Stroustrup, the designer of C++, responded "It's a Holy Grail. There is no panacea." Gabriel, one of the designers of CLOS, stated, "That's a fundamental question for which there is no easy answer. I try things". Fortunately, there does exist a vast legacy of experience with classification in other disciplines. From more classical approaches, techniques of object-oriented analysis have emerged that offer several useful recommended practices and rules of thumb for identifying the classes and objects relevant to a particular problem. These heuristics are the focus of this chapter.

The Importance of Proper Classification

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". 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 patterns 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". In contemporary biology, classification denotes "the establishment of a hierarchical system of categories on the basis of presumed natural relationships among organisms". 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. More

recently, classification has been approached by grouping organisms that share a common generic heritage: organisms that have similar DNA are grouped together. Classification by DNA is useful in distinguishing organisms that are structurally similar, but genetically very different. For example, contemporary research suggests that the lungfish and the cow are more closely related than the lungfish and the trout

To a computer scientist, biology may seem to be a stodgily mature discipline, with well-defined criteria for classifying organisms. This is simply not the case. As the biologist May reports, "At the purely factual level, we do not know to within an order of magnitude how many species of plants and animals we share the globe with: fewer than 2 million are currently classified, and estimates of the total number range from under 5 million to more than 50 million" . Furthermore, different criteria for classifying the same organisms yield different results. Martin suggests that "it all depends on what you want classification to do. If you want it to reflect precisely the genetic relatedness among species, that will give you, one answer. But if you want it instead to say something about levels of adaptation, then you, will get another" . The moral here is that even in scientifically rigorous disciplines, classification is highly dependent upon the reason for the classification.

Similar lessons may be learned from chemistry [9]. In, ancient times, all substances were thought to be some combination of earth, air, fire, and water. By today's standards (unless you are an alchemist), these do not represent very good classifications. In the mid-1600s, the chemist Robert Boyle proposed that elements were the primitive abstractions of chemistry, from which more complex compounds could be made. It wasn't until over a century later, in 1789, that the chemist Lavoisier published the first list of elements, containing some twenty-three items, some of which were later discovered not to be elements at all. The discovery of new elements continued and the list grew, but finally, in 1869, the chemist Mendeleev proposed the periodic law that gave a precise criteria for organizing all known elements, and could predict the properties of those yet undiscovered. The periodic law was not the final story in the classification of the elements. In the early 1900s, elements with similar chemical properties but different atomic weights were discovered, leading to the idea of isotopes of elements.

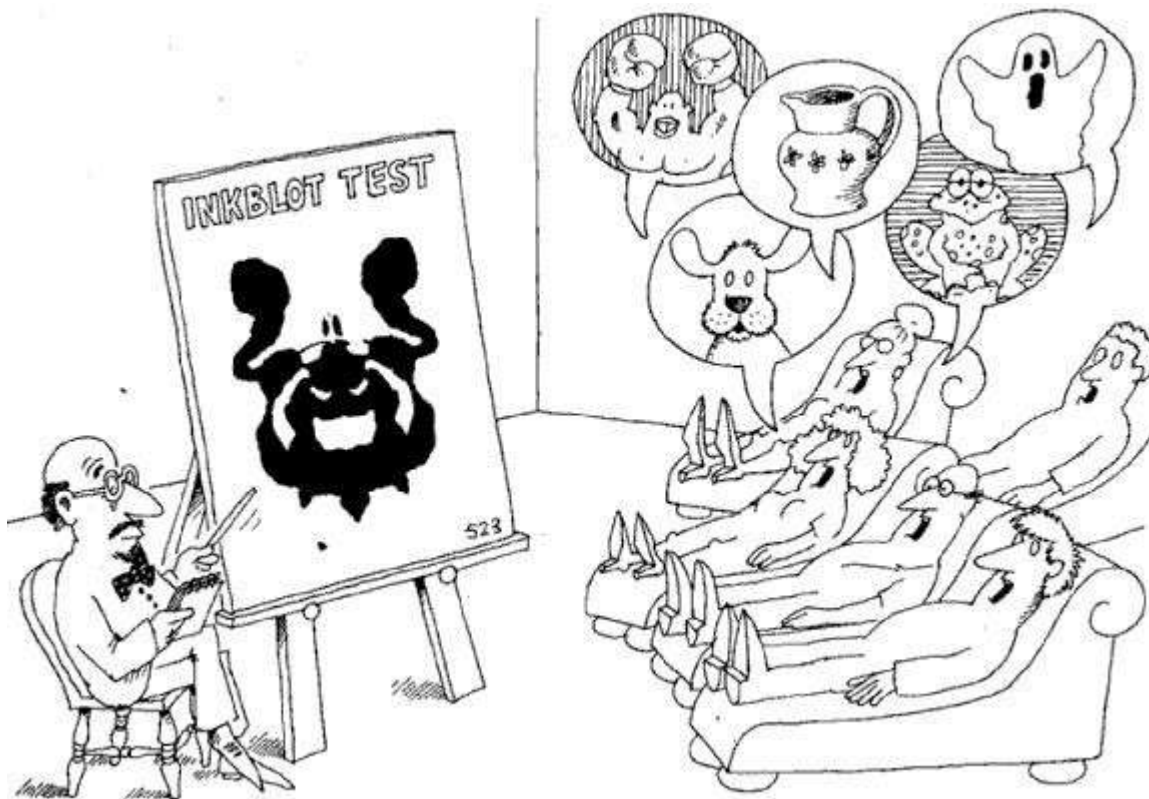
The lesson here is simple: as Descartes states, "The discovery of an order is no easy task.... yet

once the order has been discovered there is no difficulty at all in knowing it" . The best software designs look simple, but as experience shows, it takes a lot of hard work to design a simple architecture.

The Incremental and Iterative 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. This in turn enables a more sophisticated level of practice and allows us to tackle harder problems - which we often approach *ad hoc*, starting the cycle over again" .



Different observers will classify the same object in different ways.

The incremental and iterative nature of classification directly impacts the construction of class and object hierarchies in the design of a complex software system. In practice, it is common to assert a certain class structure early in a design and then revise this structure over time. Only at later stages in the design, once clients have been built that use this structure, can we meaningfully evaluate the quality of our classification. On the basis of this experience, we may decide to create new subclasses from existing ones (derivation). We may split a large class into several smaller ones (factorization), or create one larger class by uniting smaller ones (composition). Occasionally, we may even discover previously unrecognized commonality, and proceed to devise a new class (abstraction) .

Why then, is classification so hard? We suggest that there are two important reasons. First, there is no such thing as a "perfect" classification, although certainly some classifications are better than others. As Coombs, Raffia, and Thrall state, "There are potentially at least as many ways of dividing up the world into object systems as there are scientists to undertake the

task" . Any classification is relative to the perspective of the observer doing the classification. Flood and Carson give the example that the United Kingdom "could be seen as an economy by economists, a society by sociologists, a threatened chunk of nature by conservationists, a tourist attraction by some Americans, a military threat by rulers of the Soviet Union, and the green, green grass of home to the more romantic of us Britons" [14]. Second, intelligent classification requires a tremendous amount of creative insight. Birtwistle, Dahl, Myhrhaug, and Nygard observe that "sometimes the answer is evident, sometimes it is a matter of taste, and at other times, the selection of suitable components is a crucial point in the analysis" . This fact recalls the riddle, "Why is a laser beam like a goldfish? ... because neither one can whistle" . Only a creative mind can find sameness among such otherwise unrelated things.

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.

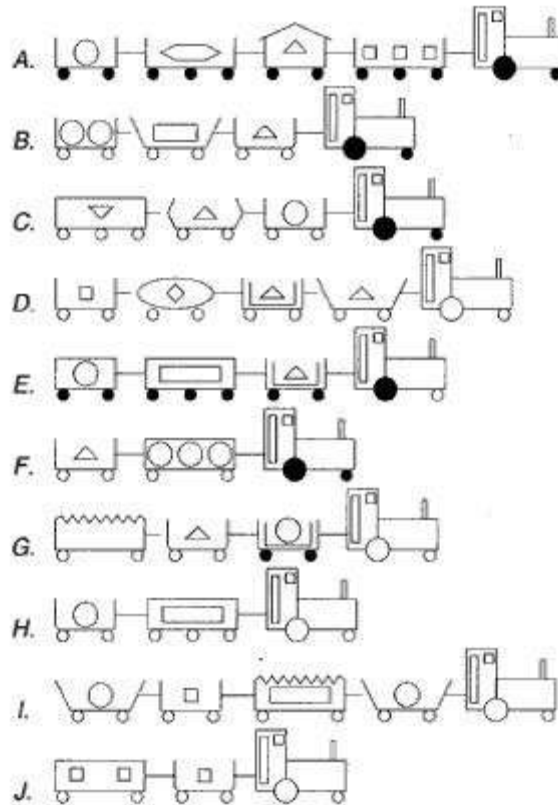


Figure 4-1
A Problem of Classification

The classical approach to categorization is also reflected in modern theories of child development. Piaget observed that around the age of one, a child typically develops the concept of object permanence; shortly thereafter, the child acquires skills in classifying these objects, first using basic categories such as dogs, cats, and toys . Later, the child discovers more general categories (such as animals) and more specific ones (such as beagles) .

To summarize, the classical approach uses related properties as the criteria for sameness among objects. Specifically, one can divide objects into disjoint sets depending upon the presence or absence of a particular property. Minsky suggests 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. Because these attributes scarcely interact at all with one another, you can put them together in any combination whatsoever to make an object that is either large or small, red or green, wooden or glass, and having the shape of a sphere or a cube" [24]. In a general sense, properties may denote more than just measurable characteristics; they may also encompass observable behaviors. For example, the fact that a bird

can fly but a fish cannot is one property that distinguishes an eagle from a salmon.

The particular properties that should be considered in a given situation are highly domain-specific. For instance, the color of a car may be important for the purposes of inventory control in an automobile manufacturing plant, but it is not at all relevant to the software that controls the traffic lights within a metropolitan area. This is in fact why we say that there are

no absolute measures of classification, although a given class structure may be better suited to one application than another. As James suggests, "No one scheme of classification, more than any other, represents the real structure or order of nature. Nature indifferently submits to any and all divisions which we wish to make among existing things. Some classifications may be more significant than others, but only by reference to our interests, not because they represent reality more accurately or adequately" .

Classical categorization permeates much of contemporary Western thought, but, as our earlier example of classifying tall and short people suggests, this approach is not always satisfactory. Kosok observes that "natural categories tend to be messy: Most birds fly, but some do not. Chairs can consist of wood, plastic, or metal and can have almost any number of legs, depending on the whim of the designer. It seems practically impossible to come up with a property list for any natural category that excludes all examples that are not in the category and includes all examples that are in the category" . These are indeed fundamental problems for classical categorization, which conceptual clustering and prototyping theory attempt to resolve.

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.

Conceptual clustering is closely related to fuzzy (multivalued) set theory, in which objects may belong to one or more groups, in varying degrees of fitness. Conceptual clustering makes absolute judgments of classification by focusing upon the "best fit."

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 [28].

There are some abstractions that have neither clearly bounded properties nor concepts. As Lakoff explains the problem, "Wittgenstein pointed out that a category like game does not fit the classical mold, since there are no common properties shared by all games. . . . Though there is no single collection of properties that all games share, the category of games is united by what Wingenstein calls family resemblances. . . . Wingenstein also observed that there was no fixed boundary to the category game. The category could be extended and new kinds of games introduced, provided that they resembled previous games in appropriate ways" .

This is why the approach is called *prototype theory*: 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 significant ways.

Lake off and Johnson apply prototype theory to the earlier problem of classifying chairs. They observe that "we understand beanbag chairs, barber chairs, and contour chairs as being chairs, not because they share some fixed set of defining properties with the prototype, but rather because they bear a sufficient family resemblance to the prototype. . . . There need be no fixed core of properties of prototypical chairs that are shared by both beanbag and barber chairs, yet they are both chairs because each, in its different way, is sufficiently close to the prototype. Interactional properties are prominent among the kinds of properties that count in determining sufficient family resemblance" .

This notion of interactional properties is central to the idea of prototype theory. In conceptual clustering, we group things according to distinct concepts. In prototype theory, we group things according to the degree of their relationship to concrete prototypes.

Applying Classical and Modern Theories To the developer in the trenches fighting changing requirements amidst limited resources and tight schedules, our discussion may seem to be far removed from the battlefields of reality. Actually, these three approaches to classification have direct application to object-oriented design.

In our experience, we identify classes and objects first according to the properties relevant to our particular domain. Here, we focus upon identifying the structures and behavior that are part of the vocabulary of our problem space. Many such abstractions are usually available for the picking [31]. If this approach fails to yield a satisfactory class structure, then we next consider clustering objects by concepts. Here, we focus our attention upon the behavior of collaborating objects. If either of these two approaches fails to capture our understanding of the problem domain, then we consider classification by association, through which clusters of objects are defined according to how closely each resembles some prototypical object.

More directly, these three approaches to classification provide the theoretical foundation of object-oriented analysis, which offers a number of pragmatic

practices and rules of thumb that we may apply to identify classes and objects in the design of a complex software system.

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.³⁶

In the following sections, we examine a number of proven approaches for analysis that are relevant to object-oriented systems.

Classical Approaches A number of methodologists have proposed various sources of classes and objects, derived from the requirements of the problem domain. We call these approaches *classical* because they derive primarily from the principles of classical categorization.

For example, Shlaer and Mellor suggest that candidate classes and objects usually come from one of the following sources :

- | | |
|-------------------|--|
| • Tangible things | Cars, telemetry data, pressure sensors |
| • Roles | Mother, teacher, politician |
| • Events | Landing, interrupt, request |
| • Interactions | Loan, meeting, intersection |

From the perspective of database modeling, Ross offers a similar list

- | | |
|-----------------|---|
| • People | Humans who carry out some function |
| • Places | Areas set aside for people or things |
| • Things | Physical objects, or groups of objects, that are tangible |
| • Organizations | Formally organized collections of people, resources, facilities, and capabilities having a defined mission, whose existence is largely independent of individuals |
| • Concepts | Principles or ideas not tangible <i>per se</i> ; used to organize or keep track of business activities and/or communications |
| • Events | Things that happen, usually to something elsen at a given date and time, or as steps in an ordered sequence |

Coad and Yourdon suggest yet another set of sources of potential objects [34]:

- | | |
|-------------|------------------------------------|
| • Structure | "Is a" and "part of" relationships |
|-------------|------------------------------------|

- Other systems External systems with which the application interacts
- Devices Devices with which the application interacts
- Events remembered An 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

At a higher level of abstraction, Coad introduces the idea of subject areas, which are basically logical groups of classes that relate to some higher-level system function.

Behavior Analysis Whereas these classical approaches focus upon tangible things in the problem domain, another school of thought in object-oriented analysis focuses upon dynamic behavior as the primary source of classes and objects.³⁷ These approaches are more akin to conceptual clustering: we form classes based upon groups of objects that exhibit similar behavior.

Wirfs-Brock, for example, emphasizes responsibilities, which denote "the knowledge an object maintains and the actions an object can perform. Responsibilities are meant to convey a sense of the purpose of an object and its place in the system. The responsibilities of an object are all the services it provides for all of the contracts it supports" . In this manner, we group things that have common responsibilities, and form hierarchies of classes involving superclasses that embody general responsibilities and subclasses that specialize their behavior.

Rubin and Goldberg offer an approach to identifying classes and objects derived from system functions. As they suggest, "the approach we use emphasizes first understanding what takes place in the system. These are the system behaviors. We next assign these behaviors to parts of the system, and try to understand who initiates and who participates in these behaviors. . .

Initiators and participants that play significant roles are recognized as objects, and are assigned the behavioral responsibilities for these roles" .

Rubin's concept of system behavior is closely related to the idea of function points, first suggested in 1979 by Albrech. A function point is "defined as one end-user business function"

A business function represents some kind of output, inquiry, input, file, or interface. Although the information-system roots of this definition show through, the idea of a function point generalizes to all kinds of automated systems: A function point is any relevant outwardly-visible and testable behavior of the system.

Domain Analysis The principles we have dis'cussed thus far are typically applied to the development of single, specific applications. Domain analysis, on the other hand, seeks to identify the classes and objects that are common to all applications within a given domain, such as patient record tracking, bond trading, compilers, or missile avionics systems. If you are in the midst of a design and stuck for ideas as to the key abstractions that exist, a narrow domain analysis can help by pointing you to the key abstractions that have proven useful in other related systems. Domain analysis works well because, except for special situations, there are very few truly unique kinds of software systems.

The idea of domain analysis was first suggested by Neighbors. We define domain analysis as "an attempt to identify the objects, operations, and relationships [that] domain experts

perceive to be important about the domain". Moore and Bailin suggest the following steps in domain analysis:

- "Construct a strawman generic model of the domain by consulting with domain experts.

- 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" .

Domain analysis may be applied across similar applications (vertical domain analysis), as well as to related parts of the same application (horizontal domain analysis). For example, when starting to design a new patient-monitoring system, it is reasonable to survey the architecture of existing systems to understand what key abstractions and mechanisms were previously employed and to evaluate which were useful and which were not. Similarly, an accounting system must provide many different kinds of reports. By considering these reports within the same application as a single domain, a domain analysis can lead the developer to an understanding of the key abstractions and mechanisms that serve all the different kinds of reports. The resulting classes and objects reflect a set of key abstractions and mechanisms generalized to the immediate report-generation problem; therefore, the resulting design is likely to be simpler than if each report had been analyzed and designed separately.

Who exactly is a domain expert? Often, a domain expert is simply a user, such as a train engineer or dispatcher in a railway system, or a nurse or doctor in a hospital. A domain expert need not be a software engineer; more commonly, he or she is simply a person who is intimately familiar with all the elements of a particular problem. A domain expert speaks the vocabulary of the problem domain.

Some managers may be concerned with the idea of direct communication between developers and end users (for some, even more frightening is the prospect of letting an end user see a developer!). For highly complex systems, domain analysis may involve a formal process, using the resources of multiple domain experts and developers over a period of many months. In practice, such a formal analysis is rarely necessary. Often, all it takes to clear up a design problem is a brief meeting between a domain expert and a developer. It is truly amazing to see what a little bit of domain knowledge can do to assist a developer in making intelligent design decisions. Indeed, we find it highly useful to have many such meetings throughout the design of a system. Domain analysis is rarely a monolithic activity; it is better focused if we consciously choose to analyze a little, then design a little.

Use-Case Analysis In isolation, the practices of classical analysis, behavior analysis, and domain analysis all depend upon a large measure of personal experience on the part of the analyst. For the majority of development projects, this is unacceptable, because such a process is neither deterministic nor predictably successful.

However, there is one practice that can be coupled with all three of these earlier approaches, to drive the process of analysis in a meaningful way. That practice is use-case analysis, first formalized by Jacobson. Jacobson defines a use case as "a particular form or pattern or exemplar of usage, a scenario that begins with some user of the system initiating some transaction or sequence of interrelated events"

Briefly, we can apply use-case analysis as early as requirements analysis, at which time end users, other domain experts, and the development team enumerate the scenarios that are fundamental to the system's operation (we need not elaborate upon these scenarios at first, we can simply enumerate them). These scenarios collectively describe the system functions of the application. Analysis then proceeds by a study of each scenario, using storyboarding techniques similar to practices in the television and movie industry . As the team walks through each scenario, they must identify the objects that participate in the scenario, the responsibilities of each object, and how those objects collaborate with other objects, in terms of the operations each invokes upon the other. In this manner, the team is forced to craft a clear separation of concerns among all abstractions. As the development process continues, these initial scenarios are expanded to consider exceptional conditions as well as secondary system behaviors (what Goldstein and Alger speak of as peripheral topics). The results from these secondary scenarios either introduce new abstractions or add, modify, or reassign the responsibilities of existing abstractions. As we will discuss further in Chapter 6, scenarios also serve as the basis of system tests.

CRC Cards CRC cards have emerged as a simple yet marvelously effective way to analyze scenarios.³⁸ First proposed by Beck and Cunningham as a tool for teaching object-oriented programming [44], CRC cards have proven to be a useful development tool that facilitates brainstorming and enhances communication among developers. A CRC card is nothing more than a 3x5 index card,³⁹ upon which the analyst writes - in pencil - the name of a class (at the top of the card), its responsibilities (on one half of the card) and its collaborators (on the other half of the card). One card is created for each class identified as relevant to the scenario. As the team walks through the scenario, they may assign new responsibilities to an existing class, group certain responsibilities to form a new class, or (most commonly) divide the responsibilities of one class into more fine-grained ones, and perhaps distribute these responsibilities to a different class.

CRC cards can be spatially arranged to represent patterns of collaboration. As viewed from the dynamic semantics of the scenario, the cards are arranged to show the flow of messages among prototypical instances of each class; as viewed from the static semantics of the scenario, the cards are arranged to represent generalization/specialization or aggregation hierarchies among the classes.

Informal English Description A radical alternative to classical object-oriented analysis was first proposed by Abbott, who suggests writing an English description of the problem (or a part of a problem) and then underlining the nouns and verbs . The nouns represent candidate objects, and the verbs represent candidate operations upon them. This technique lends itself to automation, and such a system has been built at the Tokyo Institute of Technology and at Fujitsu .

Abbott's approach is useful because it is simple and because it forces the developer to work in the vocabulary of the problem space. However, it is by no means a rigorous approach, and it definitely does not scale well to anything beyond fairly trivial problems. Human language is a terribly imprecise vehicle of expression, so the quality of the resulting list of objects and operations depends upon the writing skill of its author. Furthermore, any noun can be verbed, and any verb can be nouned; therefore, it is easy to skew the candidate list to emphasize either objects or operations.

Structured Analysis A second alternative to classical object-oriented analysis uses the products of structured analysis as a front end to object-oriented design. This technique is appealing only because a large number of analysts are skilled in structured analysis, and many CASE tools exist that support the automation of these methods. Personally, we discourage the use of structured analysis as a front end to object-oriented design, but for some organizations, it is the only pragmatic alternative.

In this approach, we start with an essential model of the system, as described by data flow diagrams and the other products of structured analysis. These diagrams provide us with a reasonably formal model of the problem. From this model, we may proceed to identify the meaningful classes and objects in our problem domain in three different ways.

McMenamin and Palmer suggest starting with an analysis of the data dictionary and proceeding to analyze the model's context diagram. As they state, "With your list of essential data elements, think about what they tell you or what they describe. If they were adjectives in a sentence, for instance, what nouns would they modify? The answers to this question make up the list of candidate objects" . These candidate objects typically derive from the surrounding environment, from the essential inputs and outputs, and from the products, services, and other resources managed by the system.

The next two techniques involve analyzing individual data flow diagrams. Given a particular data flow diagram (using the terminology of Ward/Mellor), candidate objects may be derived from the following:

- External entities
- Data stores
- Control stores
- Control transformations

Candidate classes derive from two sources:

Data flows
Control flows

This leaves us with data transformations, which we assign either as operations upon existing objects or as the behavior of an object we invent to serve as the agent responsible for this transformation.

Seidewitz and Stark suggest another technique, which they call *abstraction analysis*. Abstraction analysis focuses upon the identification of central entities, which are similar in nature to central transforms in structured design. As they state, "In structured analysis, input and output data are examined and followed inwards until they reach the highest level of abstraction. The processes between the inputs and the outputs form the central transform. In abstraction analysis a designer does the same, but also examines the central transform to determine which processes and states represent the best abstract model of what the system does". After identifying the central entity in a particular data flow diagram, abstraction analysis proceeds to identify all the supporting entities by following the afferent and efferent data flows from the central entity, and grouping the processes and states encountered along the way. In practice, Seidewitz and Stark have found abstraction analysis a difficult technique to apply successfully, and as an alternative recommend object-oriented analysis methods [50].

We must emphasize that structured design, as normally coupled with structured analysis, is entirely orthogonal to the principles of object-oriented design. Our experience indicates that using structured analysis as a front end to object-oriented design often fails when the developer is unable to resist the urge of falling back into the abyss of the structured design mindset. Another very real danger is the fact that many analysts tend to write data flow diagrams that reflect a design rather than an essential model of the problem. It is tremendously difficult to build an object-oriented system from a model that is so obviously biased towards algorithmic decomposition. This is why we prefer object-oriented analysis as the front end to object-oriented design: there is simply less danger of polluting the design with preconceived algorithmic notions.

If you must use structured analysis as a front end, for whatever honorable reasons,⁴⁰ we suggest that you stop writing data flow diagrams as soon as they start to smell of a design instead of an essential model. Also, it is a healthy practice to walk away from the products of structured analysis once the design is fully underway. Remember that the products of development, including data flow diagrams, are not ends in themselves; they should be viewed simply as tools along the way that aid the developer's intellectual comprehension of the problem and its implementation. One typically writes a data flow diagram and then invents the mechanisms that implement the desired behavior. Practically speaking, the very act of design changes the developer's understanding of the problem, making the original model somewhat obsolete. Keeping the original model up to date with the design is terribly labor intensive, is not amenable to automation, and, frankly, doesn't add a lot of value. Thus,

only the products of structured analysis that are at a sufficiently high level of abstraction should be retained. They capture an essential model of the problem, and so lend themselves to any number of different designs.

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" [51].

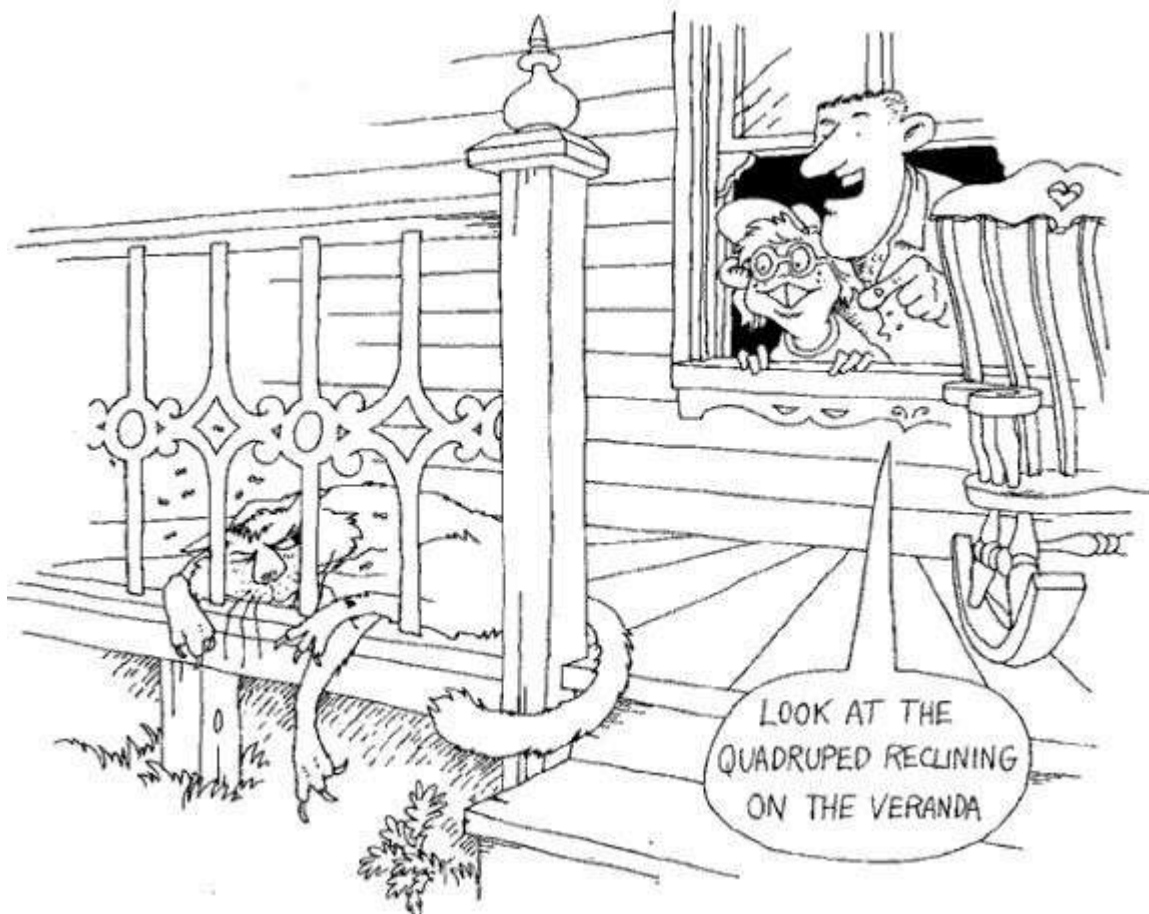
As we mentioned earlier, the identification of key abstractions involves two processes: discovery and invention. Through discovery, we come to recognize the abstractions used by domain experts; if the domain expert talks about it, then the abstraction is usually important.

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.

Perhaps the most powerful way to identify key abstractions is to look at the problem or design and see if there are any abstractions that are similar to the classes and objects that already exist. As we will discuss further in Chapter 6, in the absence of such reusable abstractions, we recommend the use of scenarios to drive the process of identifying classes and objects.

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" .

Given a new abstraction, we must place it in the context of the existing class and object hierarchies we have designed. Practically speaking, this is



Classes and objects should be at the right level of abstraction: neither too high nor too low.

neither a top-down nor a bottom-up activity. As Halbert and O'Brien observe, "You do not always design types in a type hierarchy by starting with a supertype and then creating the subtypes. Frequently, you create several seemingly disparate types, realize they are related, and then factor out their common characteristics into one or more supertypes . . . several passes up and down are usually required to produce a complete and correct program design"

This is not a license to hack, but an observation, based upon experience, that object-oriented design is both incremental and iterative. Stroustrup makes a similar observation when he notes that "the most common reorganizations of a class hierarchy are factoring the common part of two classes into a new class and splitting a class into two new ones" .

Placing classes and objects at the right levels of abstraction is difficult. Sometimes we may find a general subclass, and so may choose to move it up in the class structure, thus increasing the degree of sharing. This is called *class promotion* . Similarly, we may find a class to be too general, thus making inheritance by a subclass difficult because of the large semantic gap. This is called a *grainsize conflict* . In either case, we strive to identify cohesive and loosely coupled abstractions, so as to mitigate these two situations.

Naming things properly - so that they reflect their semantics - is often treated lightly by most developers, yet is important in capturing the essence of the abstractions we are describing. Software should be written as carefully as English prose, with consideration given to the reader as well as to the computer. Consider for a moment all the names we may need just to identify a single object: we have the name of the object itself, the name of its class, and the name of the module in which that class is declared. Multiply this by thousands of objects and possibly hundreds of classes, and you have a very real problem.

We offer the following suggestions:

- Objects should be named with proper noun phrases, such as `theSensor` or just simply

- `shape`.

- Classes should be named with common noun phrases, such as `Sensors` or `Shapes`.

- Modifier operations should be named with active verb phrases, such as `draw` or `moveLeft`.

- Selector operations should imply a query or be named with verbs the form "to be,"

- such as `extentOf` or `isOpen`.

- The use of underscores and styles of capitalization are largely matters of personal taste. No matter which cosmetic style you use, at least have your programs be self-consistent.

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

www.IntukMaterials.com
around the fuel line; pushing on the accelerator pedal eases tension on the clip, causing the fuel to flow faster (a low-cost mechanism).

Mechanisms are the means whereby objects collaborate to provide some higher-level behavior.

Which mechanism a developer chooses from a set of alternatives is most often a result of other factors, such as cost, reliability, manufacturability, and safety.

Just as it is rude for a client to violate the interface of another object, so it is socially unacceptable for objects to step outside the boundaries of the rules of behavior dictated by a particular mechanism. Indeed, it would be surprising for a driver if stepping -on-an accelerator turned on the car's lights instead of causing the engine to run faster.

Whereas key abstractions-reflect the vocabulary of the problem domain, mechanisms are the soul of the design. During the design process, the developer must consider not only the design of individual classes, but also how instances of these classes work together. Again, the use of scenarios drives this analysis process. Once a developer decides upon a particular pattern of collaboration, the work is distributed among many objects by defining suitable methods in the appropriate classes. Ultimately, the protocol of an individual class encompasses all the operations required to implement all the behavior and all the mechanisms associated with each of its instances.

Mechanisms thus represent strategic design decisions, as does the design of a class structure. In contrast, however, the interface of an individual class is more of a tactical design decision. These strategic decisions must be made explicitly; otherwise we will end up with a mob of relatively uncooperative objects, all pushing and shoving to do their work with little regard for other objects. The most elegant, lean, and fast programs embody carefully engineered mechanisms.

Mechanisms are actually one in a spectrum of patterns we find in well-structured software systems. At the low end of the food chain, we have idioms. An *idiom* is an expression peculiar to a certain programming language or application culture, representing a generally accepted convention for use of the language⁴¹. For example, in CLOS, no programmer would use underscores in function or variable names, although this is common practice in Ada . Part of the effort in learning a programming language is learning its idioms, which are usually passed down as folklore from programmer to programmer. However, as Coplien points out, idioms play an important role in codifying low-level patterns. He notes that, "many common programming tasks [are] idiomatic and therefore identifying such idioms allows "using C++ constructs to express functionality outside the language proper, while giving the illusion of being part of the language" .

At the high end of the food chain, we have frameworks. A *framework* is collection of classes that provide a set of services for a particular domain; a frame---work thus exports a number

of individual classes and mechanisms, which clients can use or adapt. As we will discuss in Chapter 9, frameworks represent reuse in the large.

Whereas idioms are part of a programming culture, frameworks are often the product of commercial ventures. For example, Apple's MacApp (and its successor, Bedrock) are both application frameworks, written in C++, for building applications that conform to Macintosh user interface standards. Similarly, the Microsoft Foundation Library and Borland's ObjectWindows library are frameworks for building applications that conform to the Windows user interface standards.

Examples of Mechanisms Consider the drawing mechanism commonly used in graphical user interfaces. Several objects must collaborate to present an image to a user: a window, a view, the model being viewed, and some client that knows when (but not how) to display this model. The client first tells the window to draw itself. Since it may encompass several subviews, the window next tells each of its subviews to draw themselves. Each subview in turn tells its model to draw itself, ultimately resulting in an image shown to the user. In this mechanism, the model is entirely decoupled from the window and view in which it is presented; views can send messages to models, but models cannot send messages to views. Smalltalk uses a variation of this mechanism, and calls it the *model-view-controller* (MVC) paradigm. A similar mechanism is employed in almost every object-oriented graphical user interface framework.

Mechanisms thus represent a level of reuse that is higher than the reuse of individual classes. For example, the MVC paradigm is used extensively in the Smalltalk user interface. The MVC paradigm in turn builds on another mechanism, the dependency mechanism, which is embodied in the behavior of the Smalltalk base class `Model`, and thus pervades much of the Smalltalk class library.

Examples of mechanisms may be found in virtually every domain. For example, the structure of an operating system may be described at the highest level of abstraction according to the mechanism used to dispatch programs. A particular design might be monolithic (such as MS-DOS), or it may employ a kernel (such as UNIX) or a process hierarchy (as in the THE operating system). In artificial intelligence, a variety of mechanisms have been explored for the design of reasoning systems. One of the most widely used paradigms is the blackboard mechanism, in which individual knowledge sources independently update a blackboard. There is no central control in such a mechanism, but any change to the blackboard may trigger an agent to explore some new problem-solving path [63]. Coad has similarly identified a number of common mechanisms in object-oriented systems, including patterns of time association, event logging, and broadcasting [64]. In

each case, these mechanisms manifest themselves not as individual classes, but as the structure of collaborating classes.