

## Chapter 1

---

---

# Introduction to Real-Time Systems and Objects

Real-time applications vary in size and scope from wristwatches and microwave ovens to factory automation and nuclear power plant control systems. Applying a general methodology to the development of real-time systems means that it must meet the tight performance and size constraints of small 4-bit and 8-bit controllers, yet scale up to networked arrays of powerful processors coordinating their activities to achieve a common purpose. Object-oriented methodologies are no silver bullet, but they offer significant improvements over traditional structured methodologies for the development of real-time systems.

Real-time systems are ones in which timeliness is essential to correctness. Object-oriented modeling is a natural fit for capturing the various characteristics and requirements of systems that have hard deadlines on performance.

### Notation and Concepts Discussed

What is special about real-time systems?	Advantages of objects
Dealing with time	Objects and the UML
Real-time operating systems	UML notation

---

## 1.1 What Is Special about Real-Time Systems?

If you read the popular computer press, you would come away with the impression that most computers sit on a desktop (or lap) and run Windows. In terms of the numbers of deployed systems, embedded real-time systems are orders of magnitude more common than their more visible desktop cousins. A tour of the average affluent American home might find one or even two standard desktop computers, but literally dozens of smart consumer devices, each containing one or more processors. From the washing machine and microwave oven to the telephone, stereo, television, and automobile, embedded computers are everywhere. They help us evenly toast our muffins and identify mothers-in-law calling on the phone. Embedded computers are even more prevalent in industry. Trains, switching systems, aircraft, chemical process control, and nuclear power plants all use computers to improve our productivity and quality of life safely and conveniently (not to mention that they also keep a significant number of us gainfully employed).

The software for these embedded computers is more difficult to construct than it is for the desktop. Real-time systems have all the problems of desktop applications plus many more. Non-real-time systems do not concern themselves with timelines, robustness, or safety—at least not nearly to the same extent as real-time systems. Real-time systems often do not have a conventional computer display or keyboard, but lie at the heart of some apparently noncomputerized device. The user of these devices may never be aware of the CPU embedded within making decisions about how and when the system should act. The user is not intimately involved with such a device as a computer *per se*, but rather as an electrical or mechanical appliance that provides services. Such systems must often operate for days or even years without stopping, in the most hostile environments. The services and controls pro-

vided must be autonomous and timely. Frequently, these devices have the potential to do great harm if they fail unsafely.

Real-time systems encompass all devices with performance constraints. *Hard deadlines* are performance requirements that absolutely must be met. A missed deadline constitutes an erroneous computation and a system failure. In these systems, *late* data is *bad* data. *Soft* real-time systems are constrained only by average time constraints—examples include on-line databases and flight reservation systems. In these systems, *late* data is still *good* data. The methods presented in this text may be applied to the development of all performance-constrained systems, hard and soft alike. When we use the term *real-time* alone, we are specifically referring to hard real-time systems.

An *embedded system* contains a computer as part of a larger system, and does not exist primarily to provide standard computing services to a user. A desktop PC is not an embedded system, unless it is within a tomographical imaging scanner or some other device. A computerized microwave oven or VCR is an embedded system because it does no “standard computing.” In both cases, the embedded computer is part of a larger system that provides some noncomputing feature to the user, such as popping corn or showing Schwarzenegger ripping telephone booths from the floor (*Commando*, a heart-warming tale if there ever was one).

Most real-time systems interact directly with electrical devices and indirectly with mechanical ones. Frequently, custom software, written specifically for the application, must control the device. This is why real-time programmers have the reputation of being “bare metal code pounders.” You cannot buy a standard device driver or Windows VxD to talk to custom hardware components. Programming these device drivers requires very low-level manipulation. This kind of programming requires intimate knowledge of the electrical properties and timing characteristics of the actual devices.

Virtually all real-time systems either monitor or control hardware, or both. Sensors provide information to the system about the state of its external environment. Medical monitoring devices, such as electrocardiography (ECG) machines use sensors to monitor patient and machine status. Air speed, engine thrust, attitude, and altitude sensors provide aircraft information for proper execution of flight control plans. Linear and angular position sensors sense a robot’s arm position and adjust it via DC or stepper motors.

Many real-time systems use actuators to control their external environment or guide some external processes. Flight control computers command engine thrust and wing and tail flap orientation to meet flight parameters. Chemical process control systems control when, what kind, and the amounts of different reagents added to mixing vats. Pacemakers make the heart beat at appropriate intervals with electrical leads attached to the inside walls of the heart.

Naturally, most systems containing actuators also contain sensors. Although there are some open loop control systems, the majority of control systems use environmental feedback to ensure that the control loop is acting properly.

Standard computing systems react almost entirely to the user and nothing else.<sup>1</sup> Real-time systems, on the other hand, may interact with the user, but have more concern for interactions with their sensors and actuators.

One of the problems that arises with environmental interaction is that the universe has an annoying habit of disregarding our opinions of how and when it ought to behave. External events are frequently not predictable. Systems must react to events when they occur rather than when it might be convenient. An ECG monitor must alarm quickly following the cessation of cardiac activity if it is to be of value. The system cannot delay alarm processing until later that evening when the processor load is less. Many hard real-time systems are *reactive* in nature, and their responses to external events must be tightly bounded in time. Control loops, as we shall see later, are very sensitive to time delays. Delayed actuations destabilize control loops.

Most real-time systems do one or a small set of high-level tasks. The actual execution of those high-level tasks requires many simultaneous lower level activities. This is called *concurrency*. Since single processor systems can do only a single thing at a time, they implement a *scheduling policy* that controls when tasks execute. In multiple processor systems, true concurrency is achievable since the processors execute asynchronously. Individual processors within such systems schedule many threads pseudoconcurrently as well.

---

<sup>1</sup> It is true that behind the scenes even desktop computers must interface with printers, mice, keyboards, and networks. The point is that they do this only to facilitate the user's whim.

Embedded systems are usually constructed with the least powerful computers that can meet the functional and performance requirements. Real-time systems ship the hardware along with the software as part of a complete system package. As many products are extremely cost sensitive, marketing and sales concerns push for using smaller processors and less memory. Providing smaller CPUs with less memory lowers the manufacturing cost. This per-shipped-item cost is called *recurring cost*, because it recurs as each device is manufactured. Software has no significant recurring cost—all the costs are bound up in development, maintenance, and support activities, making it appear to be free.<sup>2</sup> This means that most often, choices are made that decrease hardware costs while increasing software development costs.

Under UNIX, if a developer needs a big array, he might just allocate space for 1,000,000 floats with little thought of the consequences. If the program doesn't use all that space—who cares? The workstation has dozens of megabytes of RAM and gigabytes of virtual memory in the form of hard disk storage. The embedded systems developer cannot make these simplifying assumptions. He must do more with less, resulting in convoluted algorithms and extensive performance optimization. Naturally, this makes the real-time software more complex and expensive to develop and maintain.

Real-time developers often use tools hosted on PCs and workstations, but targeted to smaller, less capable computer platforms. This means that they must use cross-compiler tools, which are often more temperamental than the more widely used desktop tools. Additionally, the hardware facilities available on the target platform—such as timers, A/D converters, and sensors—cannot easily be simulated on a workstation. The discrepancy between the development and the target environments adds time and effort for the developer wanting to execute and test his code. The lack of sophisticated debugging tools on most small targets complicates testing as well. Small embedded targets often do not even have a display on which to view error and diagnostic messages.

Frequently, real-time developers must design and write software

---

<sup>2</sup> Unfortunately, many companies opt for decreasing hardware recurring costs without considering all of the development cost ramifications, but that's fodder for another book.

for hardware that does not yet exist. This creates very real challenges since they cannot validate their understanding of how the hardware functions. Integration and validation testing become more difficult and lengthy.

Embedded real-time systems must often run continuously for long periods of time. It would be awkward to have to reset your flight control computer because of a GPF<sup>3</sup> while in the air above Newark. The same applies to cardiac pacemakers, which last up to *10 years* after implantation. Unmanned space probes must function properly for years on nuclear or solar power supplies. This is different from desktop computers that are reset at least daily. It may be acceptable to reboot your desktop PC when you discover one of those hidden Excel “features,” but it is much less acceptable for a life support ventilator or the control avionics of a 767 passenger jet.

Embedded system environments are often adverse and computer-hostile. In surgical operating rooms, electrosurgical units create electrical arcs to cauterize incisions. These produce extremely high EMI (electromagnetic interference) and can physically damage unprotected computer electronics. Even if the damage is not permanent, it is possible to corrupt memory storage, degrading performance or inducing a systems failure.

Apart from increased reliability concerns, software is finding its way ever more frequently into safety systems. Medical devices are perhaps the most obvious safety-related computing devices, but computers control many kinds of vehicles such as aircraft, spacecraft, trains, and even automobiles. Software controls weapons systems and ensures the safety of nuclear power and chemical plants. There is compelling evidence that the scope of industrial and transportation accidents is increasing [1,2].<sup>4</sup> Clearly, greater care must be taken in the development of systems that have potentially catastrophic effects.

For all the reasons mentioned previously, **developing real-time software is generally much more difficult than developing non-real-time software.** The development environments have fewer tools, and the

---

<sup>3</sup> *General Protection Fault*, a term that was introduced to tens of millions of people with Microsoft’s release of Windows 3.1.

<sup>4</sup> It is not a question as to whether or not safety-critical software developers are paranoid. The real question is “Are they paranoid enough?”

ones that exist are often less capable than those for desktop environments or for “Big Iron” mainframes. Embedded targets are slower and have less memory, yet must still perform within tight deadlines. These additional concerns translate into more complexity for the developer, which means more time, more effort, and (unless we’re careful indeed) more defects than standard desktop software of the same size.

---

## 1.2 Dealing with *Time*

A critical aspect of real-time systems is how time itself is handled. The design of a real-time system must identify the timing requirements of the system and ensure that the system performance is both correct *and* timely.

The three types of time constraints on computation are:

- Hard** The correctness of response includes a description of timeliness. A late answer is incorrect and constitutes a system failure. A cardiac pacemaker must avoid pacing during specific periods of time following a contraction, or fibrillation (uncoordinated contraction of random myocardial cells—this is a *bad* thing) can occur. The time of pacing is an example of a hard real-time requirement.
- Soft** Soft timeliness requirements are specified using an average response time. If a single computation is late, it is not usually significant, although consistently late computation can result in system failures. If an airline reservation system takes a few extra seconds, the data remains valid.
- Firm** Firm deadlines are a combination of both hard and soft timeliness requirements. The computation has a shorter soft requirement, and a longer hard requirement. A patient ventilator must mechanically ventilate the patient a certain amount in the long run. A breath could come a few seconds late without affecting patient safety. However, a several minute delay in the initiation of a breath is unacceptable. Many requirements specified as soft are truly firm in nature.

The basic concepts of timeliness in real-time systems are straightforward. Most time requirements come from bounds on the performance of reactive systems. The system must react in a timely way to external events. The reaction may be a simple digital actuation, such as turning on a light, or a complicated control loop controlling dozens of actuators simultaneously. Typically, many subroutines or tasks must execute between the causative event and the resulting system action. External requirements bound the overall performance of the control path. Each of the processing activities in the control path is assigned a portion of the overall time budget. The sum of the time budgets for any path must be less than or equal to the overall performance constraint.

### 1.2.1 Real-Time Operating Systems

Most moderate to complex real-time systems use a Real-Time Operating System (RTOS). The functions of an RTOS are much the same as those for a normal operating system:

- Managing the interface to the underlying computer hardware
- Scheduling and preempting tasks
- Managing memory
- Providing common services including I/O to standard devices such as keyboards, video, LCD displays, pointing devices, and printers

RTOSs differ from normal operating systems in a variety of ways. The most important of these are:

- Scalability
- Scheduling policies
- Support for embedded, diskless target environments

First of all, the RTOS is usually *scalable*. That means the RTOS is structured like the growth rings on a redwood tree. The inner-most ring, called the kernel, provides the most essential features of the RTOS. Other features are added as necessary. Scalability makes an RTOS widely applicable to both small single-processor applications, and large distributed ones. RTOS vendors call this a *microkernel architecture*, emphasizing the small size of the minimalist kernel.

Fairness doctrines determine task scheduling in many operating systems. This ensures that all tasks have equal access to the CPU. Non-preemptive scheduling relies heavily on the proper execution of the application threads. Such schedulers cannot schedule other tasks until the currently executing thread explicitly releases control. One misbehaved task can starve all other threads by not releasing control to the OS in a timely fashion. RTOSs most commonly provide priority-based pre-emption<sup>5</sup> for control of scheduling. In this kind of scheduling, the higher priority task always preempts lower priority tasks when the former becomes ready to run. In real-time systems, average performance is a secondary concern. The primary concern is that the system meets all computational deadlines even in the absolute worst case.

RTOSs are tailored for embedded systems. They typically provide the ability to boot from ROM—a real advantage in systems without disk storage. Many can even operate out of ROM. This decreases the time necessary for system boot. Furthermore, EMI is less likely to corrupt ROM, so executing out of ROM increases the reliability of most systems.

---

## 1.3 Advantages of Objects

The previous section was all about the hard luck story that is our profession, developing real-time embedded systems. The good news is that although these issues will never go away, advances in the technology of representing and developing complex systems make them more tractable. Just as Fred Brooks said, there is no “silver bullet” that magically will make software development easy, but we can make incremental improvements in the way we think about systems and the way we develop them. This new technology is called object-oriented development, and it has been around for almost two decades now.

The primary advantages of object-oriented development are:

---

<sup>5</sup> The *priority* of a task is a measure of its timeliness requirement, not its criticality. The tighter the deadline, the higher the priority. This concern is orthogonal with the criticality or importance of the task.

- Consistency of model views
- Improved problem domain abstraction
- Improved stability in the presence of changes
- Improved model facilities for reuse
- Improved scalability
- Better support for reliability and safety concerns
- Inherent support for concurrency

The net result is that the object way is *better*. It's not dramatically better in the sense that software development will suddenly become easy, but better in that it enables us to build more complex systems in less time with fewer defects. Let's discuss each of these benefits in turn.

### 1.3.1 Consistency of Model Views

One of the problems with structured methods is the difficulty in mapping analysis views to design views and vice versa. Even though both representations are views of the very same system, it is nontrivial to show the exact correspondence between the analysis views (data flow and entity relationship diagrams) and the design views (structure charts). The fact that an infinite set of designs can fulfill the same analysis model doesn't help either. Once you're down in the code, it is difficult to show which data flow or process the code implements. The concepts used in data flow modeling and code writing are fundamentally disjoint. This makes it hard to show that the code in fact implements the analysis model.

In object-oriented systems, the same set of modeling views is used in all phases of development. Objects and classes identified in the analysis model have direct representations in the code so it is almost trivial to show the relationship between the definition of the problem (analysis) and its solution (the code).

Object-oriented systems are developed using one of two approaches. Either the analysis model is elaborated by adding design concepts (the *elaborative* development model) or a translator is built that embodies the design decisions directly (the *translative* development model). In either case, the analysis model maps directly to the implementation.

### 1.3.2 Improved Problem Domain Abstraction

Structured methods have some limited facilities for abstraction and encapsulation. However, they enforce an artificial separation of structure and behavior that greatly weakens their effectiveness. A *sensor* must be modeled on one hand as some data values and on the other as a set of operations, but the inherent link between them that exists in the real world is not maintained.

Object-oriented modeling maintains the strong cohesion among data items and the operations that manipulate them. Because this is how the real world exists, object-oriented abstractions are more intuitive and powerful. Even the vocabulary for naming the objects comes from the problem domain. Users and marketers can understand the implications of their requirements much more clearly because it is constructed using their own concepts. The object perspective is at a higher level, closer to the problem domain and further away from the computer science implementation domain. This results in a system that has loose coupling between independent aspects of your system while maintaining good cohesion of aspects that are inherently tightly coupled.

### 1.3.3 Improved Stability in the Presence of Changes

Every developer has had the experience that a small change in requirements has a catastrophic effect on the software structure. This is because the foundation of structured systems is *fundamentally unstable* and subject to radical changes. The structured development world is rather like a Dali painting in which *a priori* truths are subject to *a posteriori* modification. It works fine for art, but that's no way to live.

Because object-oriented system abstractions are based on the real world, they tend to be much more stable. The fundamental structure of the real universe doesn't undergo daily fluctuations.<sup>6</sup> Changing requirements is usually a matter of adding or removing aspects of the model rather than a total restructuring of the system.

---

<sup>6</sup> I understand that this means that politics must be excluded from the "real universe," but I'm OK with that. Ever since the Tennessee State Senate tried to legislate the value of pi to be 3.0 I think they've been banished to an alternative universe anyway.

### 1.3.4 Improved Model Facilities for Reuse

Structured systems have had limited success with reuse. If the component does exactly what you need, great—you can reuse it (provided that it links properly to your compiler on your operating system, even though it was developed with an older revision of a now-unavailable compiler). Reuse in structured systems is generally a matter of modifying the source code of the component to meet your new requirements or integrate with your new environment. The net result has been a truly abysmal record of reuse.

Object-oriented modeling includes two tactical means for improving reuse—generalization and refinement. Generalization (a.k.a. *inheritance*) supports reuse by adding and extending existing components with no changes to their source code. This is the powerful notion of “programming by difference” and allows the developer to code only the things that are different.

Refinement is similar to generalization but allows the incomplete specification of objects, which are then refined by adding the missing pieces. The same basic structure is reused by using different missing parts. It is possible to write a sort routine once and refine it for different data types, like integers, floats, accounts, EEG measurements, and target coordinates. The code that actually sorts the collection had to be written only one time but it can be applied to many different circumstances.<sup>7</sup>

### 1.3.5 Improved Scalability

The whole point of any kind of method for developing software is to manage complexity. Small systems are less complex than larger ones (Duh!), and if the system is small enough no method is required at all. At the other end of the spectrum, where we *really* need development methods is when the systems are large and complicated. Structured methods work well for small to medium scale systems, but they fail when confronted with large scale problems.

The lack of scalability of structured systems is due to a number of weaknesses within the structured way of thinking and modeling.

---

<sup>7</sup> The reuse facilities in object-oriented methods are an enabling technology permitting reuse to occur. They do not automatically guarantee that reuse *will* occur—that is a sociological issue beyond the scope of this book.

Structured systems have weaker abstraction and encapsulation facilities, meaning that structured systems tend to have some level of pathological coupling that becomes more severe as the scale of the problem grows. The use of different modeling notations and concepts in different phases means that “getting to there from here” is harder and more error-prone. Lastly, too many system aspects aren’t directly modeled, meaning that *ad hoc* approaches must be applied. As the system grows, these *ad hoc* approaches become less tenable.

Objects do it better. Improved abstraction and encapsulation maintains looser coupling among components, decreasing pathological coupling. The use of the same notation throughout the development process means that there is no “can’t get there from here” syndrome when moving from analysis to design to code. The notation itself is obvious and simple and not full of *ad hoc* artifacts necessary to circumvent the deficiencies in the method.

### 1.3.6 Better Support for Reliability and Safety Concerns

Because of better abstraction and encapsulation, the interaction of different object-oriented components can be limited to a few well-defined interfaces. This improves reliability because it is possible to control how the components interact. Additionally, it is possible to more clearly and cleanly enforce pre- and post-conditions required to make your system run properly. For example, C standard arrays require the user of the component to make sure that array bounds are not exceeded; object language allows you to build the reliability checking into the array itself. Also, object systems offer exception handling for ensuring that exceptional and fault conditions are handled correctly. Finally, because of the improved support for reuse, better-tested components can be reused so that less of each new system must be developed from scratch.

### 1.3.7 Inherent Support for Concurrency

Concurrency is a fact of life—and a very important fact for real-time embedded systems developers. Structured methods have no notion of concurrency, task management, or task synchronization. These important aspects of your system can’t even be represented using standard structured methods.

Object-oriented systems are inherently concurrent, and the details of tasking and task synchronization can be represented explicitly using orthogonal components in statecharts, active objects, and object messaging. These are powerful tools in the struggle to build correct systems that meet tight performance requirements.

---

## 1.4 Object Orientation with UML

The Unified Modeling Language (UML) is a language for expressing the constructs and relationships of complex systems. It began as a response to the Object Modeling Group's (OMG) request for a proposal for a standard object-oriented methodology. Spearheaded by Rational's Grady Booch, Jim Rumbaugh, and Ivar Jacobson, the UML is being submitted to the OMG by some of the major software companies in the world, including i-Logix, Digital, HP, ICON Computing, Microsoft, MCI Systemhouse, Oracle, Texas Instruments, and Unysis. Contributions have been made by many of the top object modelers, such as David Harel, Peter Coad, Jim Odell, and others.

UML is more complete than other methods in its support for modeling complex systems and is particularly suited for including real-time embedded systems. Its major features include:

- Object model
- Use cases and scenarios
- Behavioral modeling with statecharts
- Packaging of various kinds of entities
- Representation of tasking and task synchronization
- Models of physical topology
- Models of source code organization
- Support for object-oriented patterns

Throughout the course of this book, these features will be described in more detail, and their use shown by examples. For now, let's explore the fundamental aspects of the UML object model.

### 1.4.1 Objects

Structured methods look at a system as a collection of functions decomposed into more primitive functions. Data is secondary in structured view and concurrency isn't dealt with at all. The object perspective is different in that the fundamental decompositional unit is the *object*. So, what is an object?

**The short form:** An object is a cohesive entity that has attributes, behavior, and (optionally) state.

**The long form:** Objects represent things that have both data and behavior. Objects may represent real world things, like dogs, airfoil control surfaces, sensors, or engines. They may represent purely conceptual entities like bank accounts, trademarks, marriages, or lists. They can be visual things, like fonts, letters, ideographs, histograms, polygons, lines, or circles. All these things have various aspects, such as:

- Attributes (data)
- Behavior (operations or methods)
- State (memory)
- Identity
- Responsibilities

Let's take an example from each of these object categories. A real-world thing might be a sensor that can detect and report both a linear value and its rate of change (Table 1-1).

**Table 1-1:** *Sensor Object*

Attributes	Behavior	State	Identity	Responsibility
Linear value Rate of Change (RoC)	Acquire Report Reset Zero Enable Disable	Last value Last RoC	Instance for robot arm joint	Provides information for the precise location of the end of the robot arm in absolute space coordinates.

The sensor object contains two attributes; the monitored sensor value and its computed Rate of Change (RoC). The behaviors support data acquisition and reporting, and permit configuration of the sensor. The object state consists of the last acquired / computed values. The identity specifies exactly which object instance is under discussion. The responsibility of the sensor is defined to be how it contributes to the overall system functionality. Its attributes and behaviors must collaborate together to help the object achieve its responsibilities.

A bank account is a conceptual entity, but is nonetheless an important object (Table 1-2).

The account object also has enabling attributes and behaviors. The attributes include the balance, interest rate, and the amount of money accrued due to interest. The behaviors allow you to deposit and withdraw money, create and destroy the account, and get information about it.

A font is a visual object—see Table 1-3. Certain characteristics of an object may be more important for some objects than for others. One could envision a sensor class that had no state—whenever you asked it for information, it sampled the data and returned it, rather than storing it internally. An array of numbers is an object that doesn't have any really interesting behaviors.

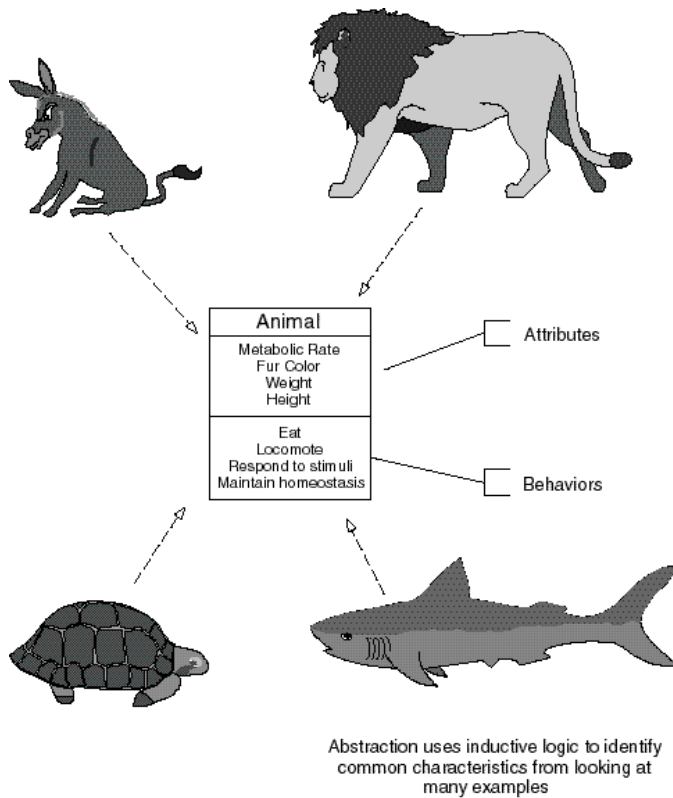
The key idea of objects is that they combine these properties into a single cohesive entity. The structured approach to software design deals with data and functions as totally separate entities. Data flow

**Table 1.2:** *Bank Account Object*

Attributes	Behavior	State	Identity	Responsibilities
Balance Interest Rate Accrued Interest	Debit Credit Report Open Close	Current balance	Sam's checking account	Stores money for Sam providing access via checks.  Provides interest on balance as long as it remains above a specific value.

**Table 1-3:** *Font Object*

Attributes	Behavior	State	Identity	Responsibility
Point size Serif Color	Draw Char Erase Char Load Unload Set color	Current character set	Times New Roman 16 pt. normal san serif (first load)	Provides a visually attractive typeface in a particular size for the display of readable English text messages.

**Figure 1-1:** *Object Abstraction*

diagrams show both data flow and data processes. Data can be decomposed if necessary. Independently, structure charts show the static call tree to decompose functions (somewhat loosely related to the data processes). Objects fuse related data and functions together. The *object* is the fundamental unit of decomposition in object-oriented programming (see Figure 1-1).

Abstraction is the process of identifying the key aspects of the entity and ignoring the rest. A chair is an abstraction defined as “a piece of furniture with at least one leg, a back, and a flat surface for sitting.” That some chairs are made of wood while others may be plastic or metal is inessential to the abstraction of “chair.” When we abstract objects we select only those aspects that are important, relative to our point of view. For example, as a runner, my abstraction of dogs is as “high-speed teeth delivery systems.” The fact that they may have a pancreas or a tail is immaterial to my modeling domain.

The object metaphor is powerful for a couple of reasons. First and foremost, it aligns well with common daily experience. In the real world, we deal with objects all the time, and each one has all the properties we’ve assigned to the preceding objects. Rocks may not have interesting behavior—but they do have attributes, like color, weight, and size. They certainly have responsibilities, such as intimidating hungry pit bulls. Most objects have behavior as well. Engines turn on or off, deliver torque, guzzle gas, and require maintenance. Object-oriented decomposition allows us to use our hard-won intuition that we’ve gained by simply living in the world and interacting with it. This is not true of functional decomposition.

ALGOL-based languages introduced the concept of an Abstract Data Type (ADT). Rather than defining only the underlying data bit patterns, ADTs include the operations that make sense in terms of their use and purpose. An enumerated type is an example of an ADT. Pascal provides three operators for enumerated types: `ord()`, `pred()`, and `succ()`, or ordinal value, predecessor, and successor, respectively. Ada provides the same operators as attributes `A’POS`, `A’PRED`, and `A’SUCC`, in addition to a few more. C has much weaker abstraction facilities. It short-circuits the abstraction by making visible the internal unsigned integer structure of enumerations. It is important to consider the ADTs and the operations defined on them together.

In their simplest expression, objects are nothing more than ADTs

**Table 1-4:** *Common ADTs*

Data Structure	Operations
Stack	Push Pop Full Empty
Queue	Insert Remove Full Empty
Linked List	Insert Remove Next Previous
Tree	Insert Remove Next Previous

bound together with related operators. This is a low-level perspective and doesn't capture all of the richness available in the object paradigm. Software developers use such ADTs and operators as low level mechanisms all the time—stacks, queues, trees, and all the other basic data structures are nothing more than objects with specific operations defined. Consider the common ADTs in Table 1-4.

At a low-level of abstraction, these are merely objects that provide these operations intrinsically rather than ADTs with separate functions to provide the services. In Pascal, to insert an item in a stack, you might have code that looks like this:

```
type
  OKType = {NoCanDo, CanDo}
  stackFrame = array [1..100] of float;
  record stack
    st: stackFrame; { holds stack values }
    top: integer; { holds top of stack }
  end;
```

```

function insert(var s: stack; f: float) : OKType;
begin
    if s.top > 100 then
        insert := NoCanDo
    else
        begin
            s.st[s.top] := f;
            s.top:= s.top + 1;
            insert := CanDo;
        end;
    end; { insert }

var
    s: stack;
    result: OKType;
begin
    s.top := 1; { start at the beginning }
    result:= insert(s, 3.14159265);
end.

```

There are some open issues. Who ensures that `s.top` is initially set to 1? Here it is decoupled from the declaration of the stack variables and appears after the main BEGIN. Does the `insert()` function apply to other stack-type objects that might store integers or strings instead of floats? These questions arise because in Pascal there is no way to bind the ADT to the operations defined for it. Compare the preceding code with a C++ implementation:

```

class stack {
    int size;
    int top;
    float *st;
public:
    // constructor sets up top ok
    stack(int s=100) : size(s), top(0) {
        // st = new float[size]; };
    void insert(float f) {
        if (top > size)
            throw StackOverflow;
        else
            st[top++] = f;
    };
    ~stack() { delete st[]; };
};

```

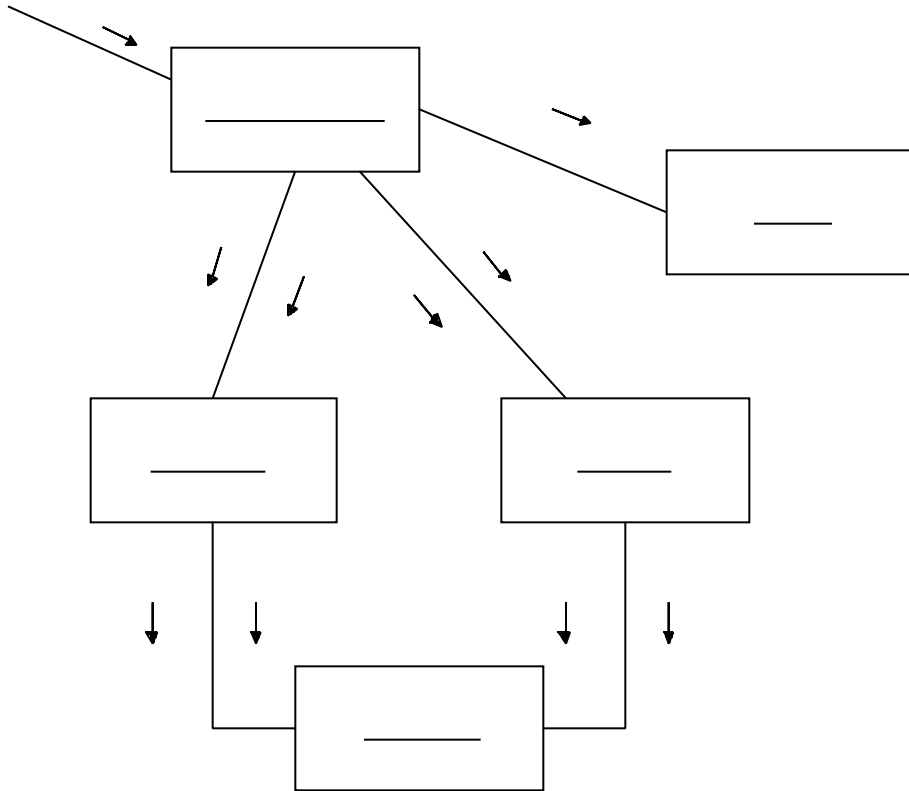
```
void main() {
    stack s, t(400);
    s.insert(3.14159265);
    t.insert(2.7182818284);
};
```

Although in this case it may not look like much of an improvement, the stack object binds the data and its associated operations together. Because the concept of a stack is meaningless without both the operations and the data, it makes the most sense to bind these things tightly together—they are different aspects of a single concept. This is called *strong cohesion*—the appropriate binding together of inherently tightly coupled properties. The result of the cohesion is that we can easily create a different sized stack (the default gives us our 100-element stack). The stack object itself handles the details of creation and deletion of variables of type stack. The insert operation is also a part of the stack object.

In a more general sense, objects may be thought of as autonomous machines. Our bodies are constructed of diverse sets of cells that have different attributes, roles, behavior, state, and responsibilities. They come together in higher level collaborations called *organs* to achieve some higher level systemic function, like digestion, locomotion, or thermoregulation. The cells themselves are autonomous and take care of their internal details, just like software objects.

Since objects are autonomous machines, it is easier to ensure that they are loosely coupled with the objects around them. In fact, the execution of behaviors is far more general under this notion than in the standard functional model. In the functional model, it is assumed that the caller calls a function and waits until it is complete and returns, whereupon the caller resumes. This is only one of several available models of interobject communication. Objects may implement this synchronous (direct function) call, but may also implement asynchronous calls as well, as when the called object runs within a different thread of execution or even a different processor. Different mechanisms for handling guards and blocking are also available. The model of object-as-machine is a very general one.

Rather than depict one object calling a service of another, the general model is that one object sends a message to the other requesting a service or operation (see Figure 1-2). Messages may be implemented in



**Figure 1-2:** *Objects Collaborate to Achieve System Functionality*

many different ways to achieve different effects. At the modeling stage, message implementation is not an essential detail, and as such, it should not be visible.

### 1.4.2 Attributes

Attributes, in OO-speak, refer to the data encapsulated within an object. It might be the balance of a bank account, the current picture number in an electronic camera, the color of a font, or the owner of a trademark. Some objects may have just one or a small number of sim-

ple attributes. Others may be quite rich. In some object-oriented languages, all instances of data types are objects, from the smallest integer type to the most complex aggregate. In C++, in deference to minimizing the difference between C and C++, variables of the elementary data types, such as *int* and *float*, are not really objects. Programmers may treat them as if they are objects (well, almost anyway—but that’s a programming, rather than a modeling, issue) but C++ does not require it.

### 1.4.3 Behavior

Interesting objects do interesting things. Passive objects supply behaviors for other objects; that is, they provide services that other objects may request. ADTs are typically passive objects. In the previous C++ example, the stack objects provide storage for simple data values, the means for inserting and removing them from storage, and some simple error checking to ensure their integrity. Active objects form the roots of threads and invoke the services (behaviors) of the passive objects.

Logically, behavior can be modeled as three distinct types: simple, automaton, and continuous. All three are important, although the second has a particular importance in real-time systems.

The first kind of behavior is called *simple*. The object performs services on request and keeps no memory of previous services. Each action is atomic and complete, at least from an external perspective. A simple object may maintain a collection of primitive data types and operations defined on them. A binary tree object, for instance, shows simple behavior. Another example is a  $\cos(x)$  function.  $\cos(\pi/2)$  always returns the same value, regardless of what value it was invoked with before. It retains no memory of previous invocations. This kind of object is also called *primitive*.

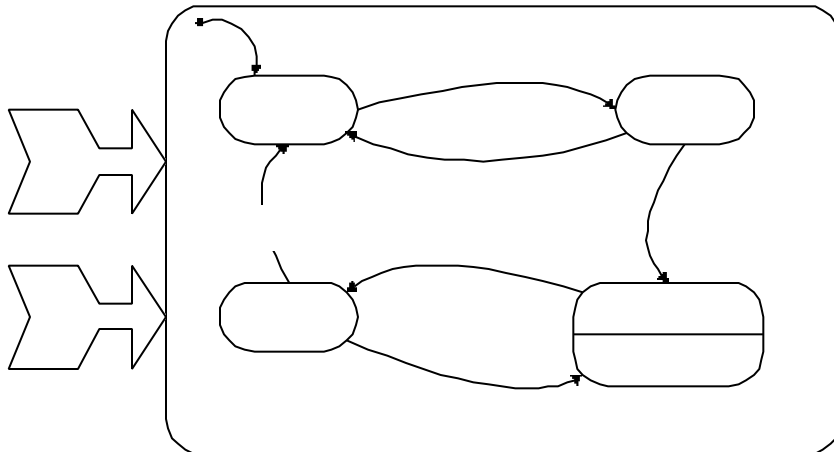
The second type of object behavior treats the object as a particular type of machine, called an *automaton* or Finite State Machine (FSM). This kind of object possesses a bounded (*finite*) set of conditions of existence (*states*). It must be in one and only one state at a time. An automaton exhibits modal behavior—each mode constituting a state. A state is an independent condition of existence defined by the set of events it processes and the actions it performs. Because objects with state machines react to events in well-defined ways, they are also called *reactive objects*.

Incoming events can induce transitions between object states in

some predefined manner. Some object-oriented methods claim that all objects exhibit state behavior. A sample-and-hold A/D converter is such an object, as shown in Figure 1-3. It shows the states of

- Enabled
- Sampling
- Holding
- Disabled

The third kind of object behavior is called *continuous*. An object with continuous behavior is one with an infinite, or at least unbounded, set of existence conditions. One example is an *algorithmic object*. This is an object that executes some algorithm on a possibly infinite data stream. A moving average algorithm performs a smoothing function over an incoming data stream. Objects with continuous behavior are objects whose current behavior is dependent on past behavior and inputs, but the dependency is of a continuous, rather than discrete nature. Fuzzy systems and PID control loops are examples of continuous systems, as are pseudo-random number generators and digital filters. Their current behavior depends on past history, but in a quantitative not qualitative way.



**Figure 1-3:** State Machine for a Sample-and-Hold A/D Converter

### 1.4.4 Messaging

The logical interface between objects is done with the passing of *messages*. A message is an abstraction of data and/or control information passed from one object to another. Many different implementations are possible. For example:

- A function call
- Mail via a Real-Time Operating System (RTOS)
- An event via an RTOS
- An interrupt
- A semaphore-protected shared resource
- An Ada rendezvous
- A Remote Procedure Call (RPC) in a distributed system

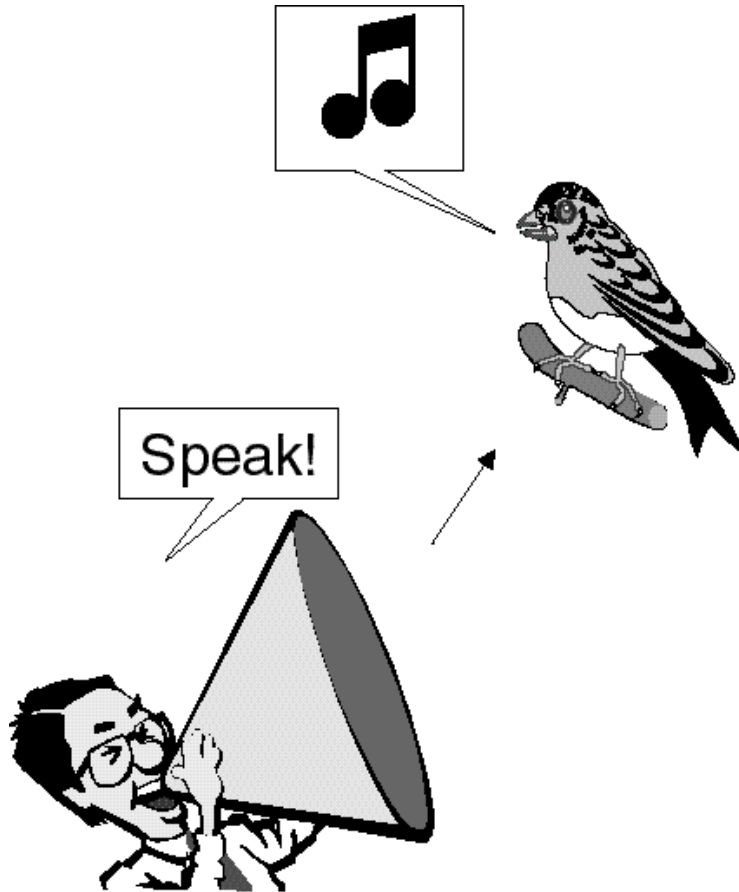
Early analysis identifies the key messages. Later design elaborates an implementation strategy that defines the synchronization and timing requirements for each message. Internally, the object translates the messages into acceptor operations, state transitions, commands, or data to munch on, as appropriate. Messages occur only between objects that have an association (see Figure 1-4).

Use of message passing enforces loose coupling. In analysis, one does not specify interface details such as synchronicity, function call format, rendezvous, time outs, etc. These are design and implementation details that can be decided later, once the overall problem is better understood.

An object's interface is the facade that it presents to the world, and is defined by the set of protocols within which the object participates. An interface protocol consists of three things:

- Preconditions
- Signature
- Postconditions

The preconditions are the conditions guaranteed to be true before the message is sent or received. Preconditions are normally the responsibility of the object sending the message. Postconditions are the things guaranteed to be true by the time the message is processed and are the



**Figure 1-4:** *Sending a Message to an Object*

responsibility of the receiver of the message. The message signature is the exact mechanism used for message transfer. This can be a function call with the parameters and return type or RTOS message post/pend pair, or bus message protocol.

The interface should reflect the essential characteristics of the object that require visibility to other objects. Objects should hide inessential details. Objects enforce strong encapsulation. In C++, for example,

common practice is to hide data members but publish the operations that manipulate the data.

### 1.4.5 Responsibility

The responsibilities of an object are the roles that it serves within the system. The interface and the behaviors provide the means by which responsibilities are met, but do not define them. Consider a front-end loader tractor as an object, as described in Table 1-5.

The responsibilities are the roles the tractor will play for the road construction firm that uses it. The behaviors must be sufficiently rich to enable the responsibilities to be fulfilled, but do not of themselves define those responsibilities.

### 1.4.6 Concurrency

Unlike subroutines in structure charts, objects are inherently concurrent unless otherwise specified. It is theoretically possible for each object to run on its own processor. It is the physical structure of modern computers that drives sequential threads.

**Table 1-5:** *Tractor Characteristics*

Attributes	Carrying capacity Wheel size Maximum engine output Clearance height Weight
Behaviors	Lift Drop Move Set direction Change gear
Responsibilities	Move dirt from road bed to truck Dig holes for bridge supports Fill holes mistakenly dug

*A thread is a set of operations executed in sequence.* Concurrent threads can run on separate processors, meaning that the relative speeds with which they progress are uncoupled. On the same processor, we must rely on pseudo-concurrency provided by the underlying operating system, or write our own executive. These concurrency mechanisms allow the threads to progress more or less independently.

### 1.4.7 Objects as Autonomous Machines

Taken together, the characteristics of objects, such as attributes, behaviors, interfaces, encapsulation, and concurrency, allow each object to act as a separate entity—an autonomous machine. This machine does not have to be very smart, but must own its responsibilities and collaborate with other machines to achieve some higher order goals. At a minimum, objects must ensure:

- Data integrity
- Interface protocols are followed
- Their own behavior

This is true of small simple objects as well as more complex elaborate ones. *Simple objects are akin to biological cells.* Cells manage their own metabolism, absorb nutrients, maintain intracellular homeostasis, and fulfill whatever small function they provide to the system as a whole. Large groups of cells collaborate to form organs that excrete hormones, locomote, maintain systemic homeostasis, and play video games. Even larger collaborations form individual people, and collaborations of these (in pathological cases) form ANSI standards committees! Within the context of their responsibilities, these objects protect their own interests (especially in the standards committees).

This leads to a fundamental rule of object systems—*distributed intelligence*. *Each object, however lowly and simple, has enough brains to manage its own resources and perform its own behaviors.* This is different from functional decomposition in which it is common to have elaborately complex master subroutines that know everything about everybody. The truth of the matter is that it is far easier to construct dozens of semismart objects than a single really smart object that knows everything.

### 1.4.8 Classes

In the object-oriented world the term *class* is used in precisely the same way as in philosophy. A class is an abstraction of the common properties from a set containing many similar objects.

A class can be thought of as the type of an object.<sup>8</sup> The values 0, -3, and 7879 are all instances (objects) of the class integer. Further, all instances of a class have all the properties defined by the class. A mammal may have fur, bears its young live, and is homeothermic. This is true of all members of the class mammals—cats, mice, bears, and even rock stars. This does not mean that instances of the class are all the same—cats are certainly different from rock stars—but they share at least some set of common properties (most notably: fur, indifference to the needs of others, and a universal inability to sing). The values of these properties may be different among instances, but all properties must be present. For example, an account class may define a balance attribute—that is, all accounts have balances. Some accounts may have positive values, while others hover around zero or even dip into negative numbers.

Object-oriented designers uncover classes much the same way as philosophers—by observing a number of objects and abstracting the common properties. Table 1-6 contains some example classes.

In each of these cases, it is possible to imagine specific object instances of these classes. Just as a *struct* in C defines data structure, a class defines the type of objects created in its likeness (see Figure 1-5).

A class defines the attributes and behaviors of the objects it instantiates,<sup>9</sup> but not their responsibilities. All properties of objects of a class are all the same in *type*, but not in *value*. That is, if a class has an attribute, such as color or charm, then instances of the class have the characteristic although their particular value of the attribute may differ. Responsibilities, however, are context-specific, and are determined by the use of the object within that context. A simple container object may

---

<sup>8</sup> Strictly speaking, the *type* refers to the interface of the object—objects with the same interface are of the same type, regardless of their class. The class of an object defines its internal implementation. This is not normally a useful distinction unless you are using languages that make the difference visible, such as Java.

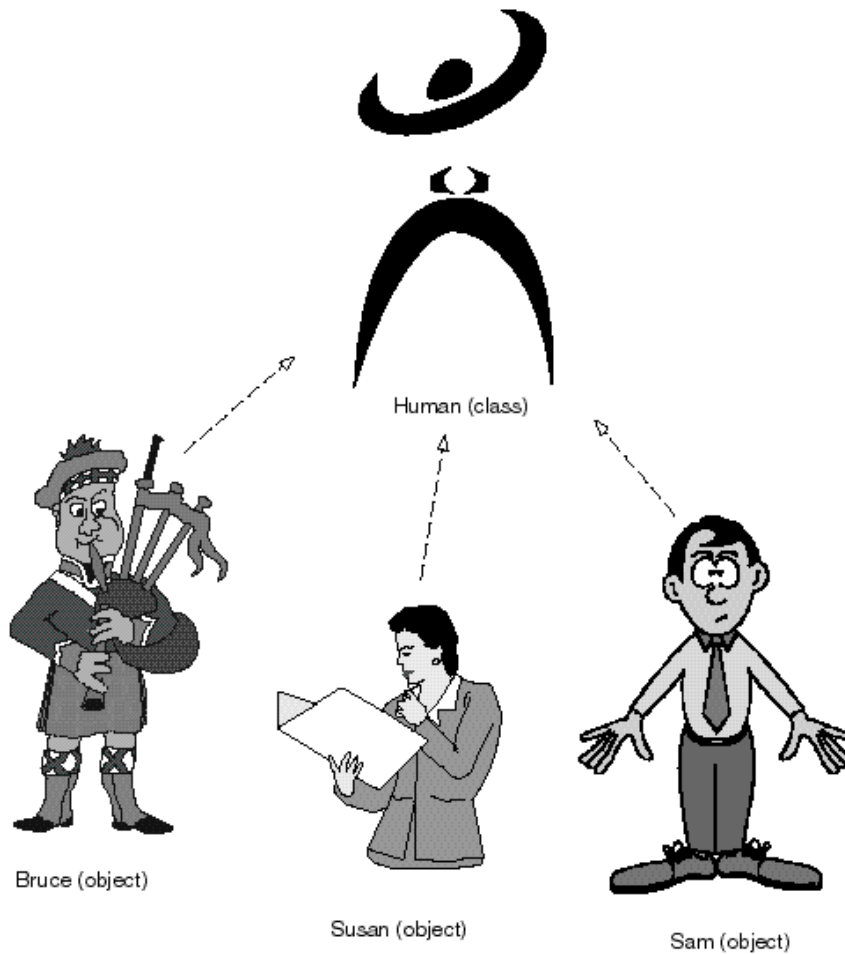
<sup>9</sup> The term *instantiation* comes from the term *instance*, as in *making an instance* (i.e., object) of a class.

**Table 1-6:** *Class Examples*

Class	Attributes	Behaviors	Example Objects	Responsibilities
Bank account	Account type	Open	Sam's checking account	Maintain updated balance to account for credits, debits, and interest. Also maintain an account history.
	Account number	Close	Julie's savings account	
	Balance	Credit Debit		
Elevator	Capacity	Go to floor	Elevator 1 Blg 6	Carry passengers to their desired floor.
	Current floor	Stop	Elevator 3 Blg 1	
	Current direction	Open door Close door		
Marriage	Wedding Date	Wed	Cindy's first marriage	Maintain stable social group.
	Number of Children	Divorce		
	Children	Create Children		
Airline flight	Flight number	Takeoff	My flight to Jamaica	Carry passengers and luggage to destination safely.
	Date	Land		
	Point of origin	Lose Luggage		
	Destination			
	Departure time			
ECG signal	Heart rate	Get heart rate	George's ECG signal	Monitor and report status of patient's cardiac function to the attending physician; alarm for possibly dangerous conditions.
	PVC count	Set alarm limits		
	ST segment height	Display waveform		
	Display rate			
	Scalefactor			

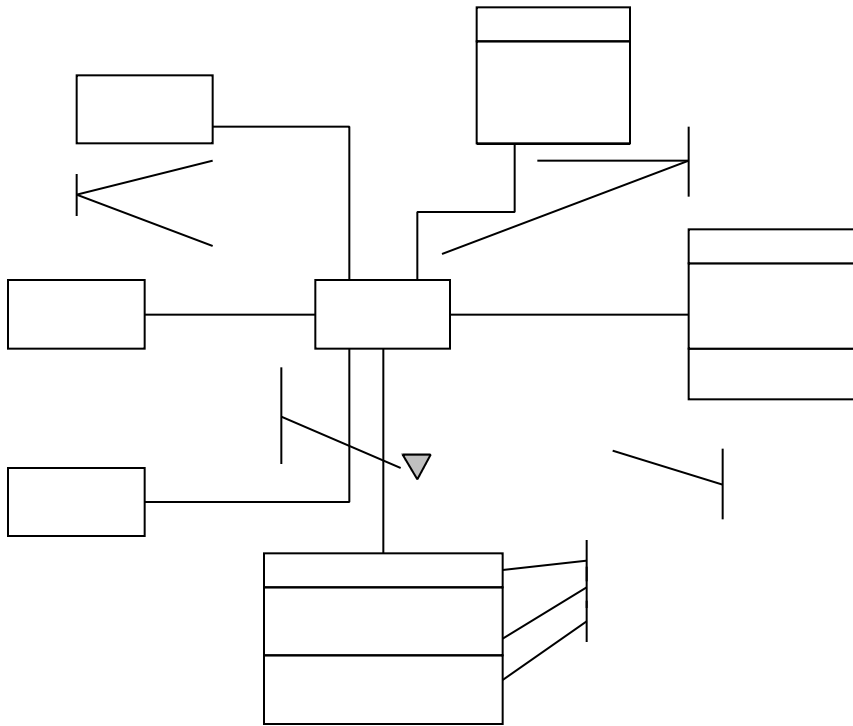
hold checking accounts and be responsible for coordinating access to these accounts. A container object of the same class may hold inventory records and coordinate access for an inventory control system. The responsibilities are similar in type but differ in the specifics. An object's class defines the responsibilities of an object only when all such objects are all used in the same context and in the same way.

UML classes are shown using rectangles with the name of the class



**Figure 1-5:** *Classes and Objects*

inside the rectangle. A variation uses a three-segment box; the top segment has the name of the class, the middle segment contains a list of attributes, and the bottom segment contains a list of operations. Not all of the attributes or operations need to be listed. Figure 1-6 shows a simple autopilot system consisting of an autopilot class and various sensor and actuator classes it uses.



**Figure 1-6:** *A Simple Class Diagram*

### 1.4.9 Associations among Classes and Objects

For one object to send messages to another, they must associate with each other in some way. You can imagine that some association exists between the following pairs of objects:

Object	Association	Object
Engine	?	Piston
Flight computer	?	Engine
Linear position sensor	?	Sensor
Ship	?	Planks
Elevator	?	Call button
Bank customer	?	Bank account

As we will see later, the process of analysis identifies the key objects in the system as well as how they relate to each other. All associations can be (optionally) named when they clarify the association. The preceding relationships can be refined:

Object	Association	Object
Engine	<i>has a</i>	Piston
Flight computer	<i>controls an</i>	Engine
Linear position sensor	<i>is a type of</i>	Sensor
Ship	<i>contains some</i>	Planks
Elevator	<i>is called by</i>	Call button
Bank customer	<i>stores money in a</i>	Bank account

In UML, relationships exist between classes. Five elementary types of object relationships exist: *association, aggregation, composition, generalization, and refinement.*

*Associations* are relationships that manifest themselves at run-time to permit the exchange of messages among objects. Associations are shown using simple lines connecting two objects. Unless otherwise specified, UML associations are bidirectional and support messaging in either direction. When it is clear that messages go in only one direction, an open arrowhead points to the receiving object.

*Aggregation associations* are shown with diamonds at the owner end of the relationship. *Aggregation* is used when one object logically or

physically contains another. *Composition* is a strong form of aggregation in which the owner is explicitly responsible for the creation and destruction of the part objects. The directed lines with closed arrowheads indicate a *generalization* or *is-a-kind-of* relationship. *Refinement* is shown with a closed arrowhead, like generalization, but uses dashed lines. Refinement relationships support generic or template elaborations of incomplete class specifications.

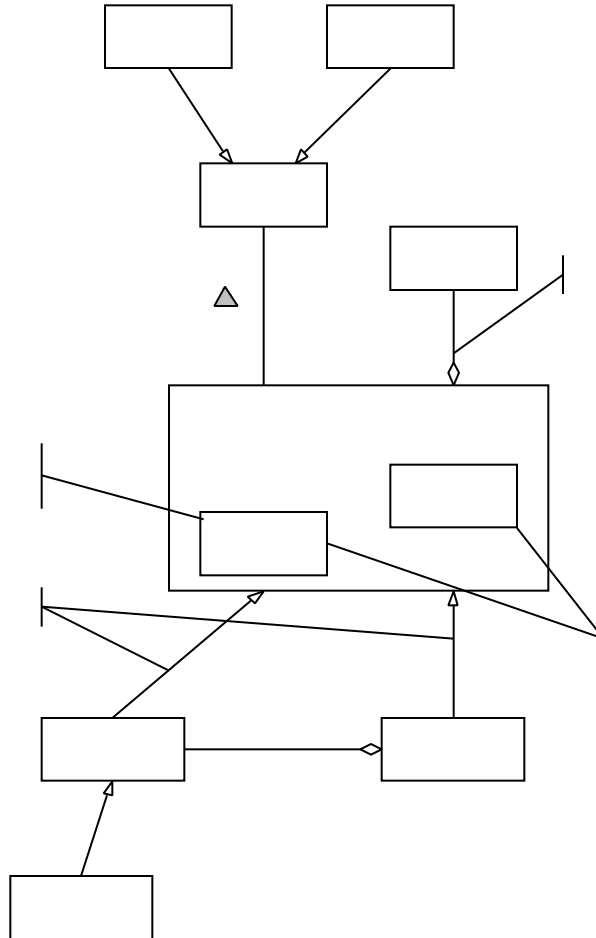
The numbers at each end of the relationship line denote the number of objects that participate in the relationship at each end—this is called the *multiplicity* of the role. We see in Figure 1-7 that one Window object can have 0, 1, or 2 scroll bars. Because the Window can have no scroll bars, this is an *optional* relationship. The scroll bar, for its side, works with only a single window, so the number at the Window side of the relationship is one. If multiple windows shared a scroll bar, then the cardinality would be "\*" (an indicator for "unspecified but greater than or equal to 0") or the fixed number, if known. Figure 1-8 shows a more real-time example, a sensor.

### *Association*

When one object uses the services of another, but does not own it, the objects have an association. *Associations* are appropriate when any of the following is true:

- One object uses the services of another but is not an aggregate of it
- The lifecycle of the used class is not the responsibility of the user class; that is, it is not responsible for both the used object's creation and destruction
- The association between objects is looser than one of aggregation
- The association can be characterized as client-server
- The used object is shared and used equally by many others

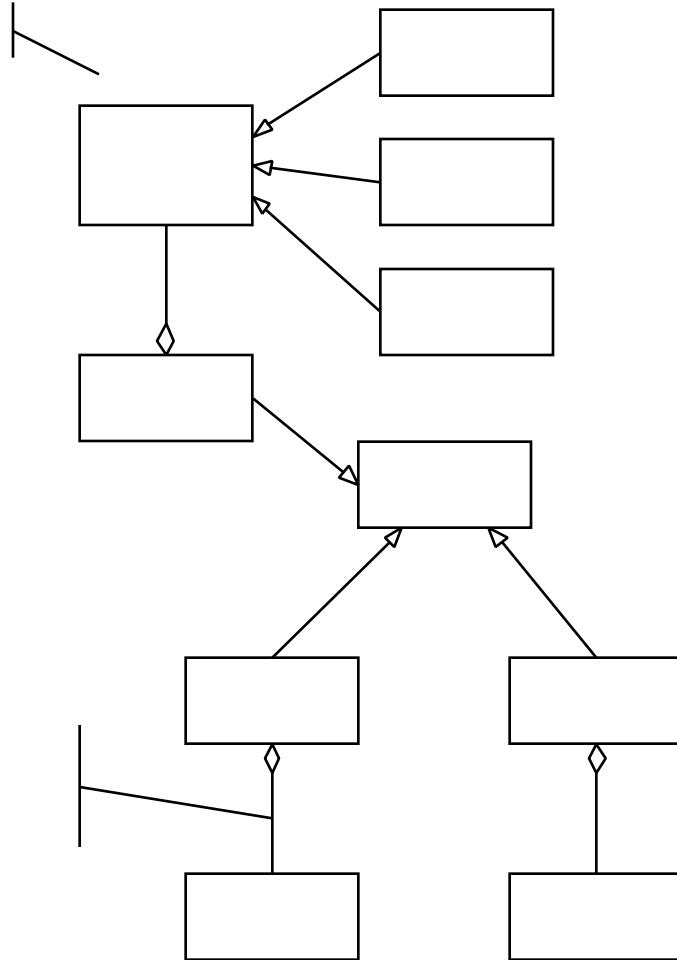
The class diagrams show that a Window class *uses* (specifically, gets user input from) various input devices. The two types shown are a mouse and keyboard, but other devices are conceivable, including tablets, microphones, and even a modem for a remote session.



**Figure 1-7:** *Window Class Relationships*

### *Aggregation*

An **aggregation** relationship applies when one object physically or conceptually contains another. The larger class is referred to as the *owner* or *whole*, and contains the diamond end of the aggregation. The smaller class is the *owned*, *part*, or *component* class. The owner is typically responsible for the creation and destruction of the owned class. UML al-



**Figure 1-8:** *Sensor Class Relationships*

allows components of aggregates to be shared among owners. When an object has shared ownership, some ad hoc rule must specify who has the responsibility for its creation and destruction.

In Windows programming, a *window* is a kind of object that contains a *client area*. The client area cannot stand on its own without being contained by a window. The client area comes into existence during the creation of the window and is destroyed when the window is

destroyed. In the Sensor class shown, a Sensor class has an A/D converter.

### *Composition*

**Composition** is a strong form of aggregation. Components normally are shown by actual inclusion of the component class within the composite. Alternatively, an aggregation association can be used, but shown with a filled diamond. Components of composites cannot be shared (that is, they can have only one owner) and the composite is required to create and destroy its components. A common use of composites is as active objects—that is, objects that are the roots of threads. These active objects create the threads in which to operate, and their components execute within this thread. The composite receives messages and events from the RTOS and other threads, and dispatches them to the appropriate components within its own thread.

### *Generalization*

When one class is a specialization of another, the relationship is called **generalization** or **inheritance**. It means that the child or descendent class has all the characteristics defined by the parent, although it might specialize them. The child may also extend its parent class by adding additional attributes and behaviors. **Fundamental to generalization is that it is an “is-a-kind-of” relationship between classes.** A mammal is-a-kind-of animal and an infrared sensor is-a-kind-of sensor.

Type hierarchies are created from classes and their inheritance relationships. Such hierarchies form the basis of some kinds of frameworks, such as MFC (Microsoft Foundation Classes) and Borland’s OWL (Object Windows Library).

In the Window class diagram in Figure 1-7, the Window parent class has two direct descendants, SDI (Single Document Interface) and MDI (Multiple Document Interface) classes. The SDI class is further subtyped into a Dialog Box class. A Dialog Box is an SDI window that does not have a menu<sup>10</sup> and has visual controls placed in its client area. Mul-

---

<sup>10</sup> Note that all subtypes of the class Windows *can* have a menu (the multiplicity of “0,1” makes it optional), but dialog boxes commonly do not.

tiplicity makes no sense for inheritance relationships, and so is not depicted. Since the parent Window class has a client area, all of its descendants do as well.

Inheritance is an extraordinarily powerful facility, despite its seeming simplicity. It allows objects to be *specified by difference* rather than from scratch each time. In standard structured methods, extending or specializing a function requires modification of the source code to produce a new routine that meets your needs. In object-oriented systems, you may subclass the parent to create a child class and merely add the additional attributes and behaviors needed. If a behavior needs to be implemented differently for a subclass, that's no problem either. You redefine the behavior in the subclass. The object-oriented paradigm ensures the correct version of the behavior will be called based on the object type you have.

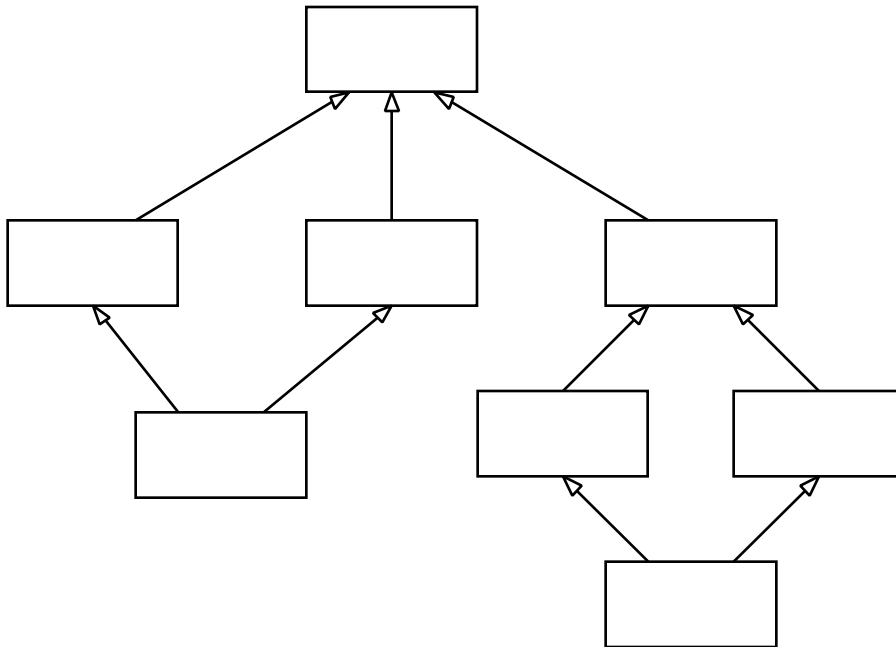
What we have discussed so far is called *single inheritance*—a class has one parent. Some languages, such as Ada and Smalltalk, support only single inheritance. Other languages, such as C++, allow you to inherit from more than one parent class—this is called *multiple inheritance*.

The primary reason for multiple inheritance is that you may want to use or create objects that share characteristics from multiple class libraries. One of the goals of object-orientation is to reuse software; multiple inheritance can facilitate reuse by allowing mixing of different class hierarchies. Remember that the inheritance relationship is just another way to say “is-a-kind-of.” It is entirely possible for this relationship to hold within more than one class hierarchy. See Figure 1-9.

As Ellis [3] points out, the characteristics of a helicopter and a fixed-wing propeller airplane are significantly different. So how do you classify the Osprey V-22? It is a helicopter that has two propellers that can be oriented vertically as in a helicopter, or rotated in-flight to the orientation of propellers on a fixed wing airplane. Both “Rotary Wing Vehicle” and “Fixed Wing Vehicle” applies, so which do you choose?

Figure 1-10 shows another instance of multiple inheritance, adapted from [4]. Here are two important hierarchies for a university—employees and students. Employees have attributes such as:

- Rate of pay
- Tax deductions
- Medical plan options
- Social Security payments made this year



**Figure 1-9:** *Multiple Inheritance*

Students have an entirely orthogonal set of attributes such as:

- Current class load
- Grade point average
- Advisor
- Major
- Minor
- Year
- Unpaid tuition amount due University

These are both rich types containing important information. Which hierarchy do you select as most important for a graduate teaching assistant?

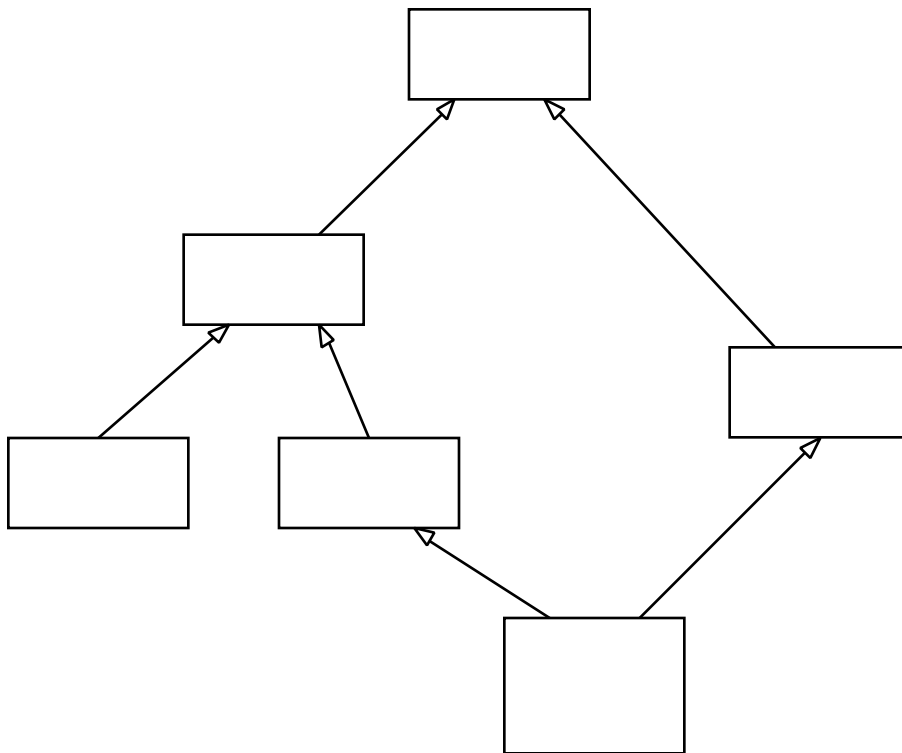
One solution to both these problems is to inherit from both hierarchies. This solution is problematic if the two hierarchies have a com-

mon point anywhere above the direct parents of the class. Remember that an object inherits all characteristics of its parent. If the School Member has the characteristics shown in Table 1-7, then the definition of Student and Faculty might be that in Tables 1-8 and 1-9, respectively.

So, the Graduate Teaching Assistant would then have the class definition described in Table 1-10.

Oops! Since the class Graduate Teaching Assistant inherits from two hierarchies (that just so happen to be joined at the hip), it has two of all characteristics from the School Member class. So if a class updates an *address*, which one is modified? Both?

C++ addresses this by providing the facility to mark a base class as *virtual*. A virtual base class will only be represented once in a multiply-



**Figure 1-10:** *Multiple Inheritance of School Membership*

**Table 1-7:** *School Member*

Attributes	Name Address
Behaviors	Accept school mailer

**Table 1-8:** *Student*

Attributes	Name (inherited from School Member) Address (inherited from School Member) Student number GPA Current class load Tuition owed
Behaviors	Accept school mailer (inherited from School Member) Send tuition bill Compute GPA

**Table 1-9:** *Faculty*

Attributes	Name (inherited from School Member) Address (inherited from School Member) Employee number Pay rate Pay period Deductions Social Security Number Medical benefits plan
Behaviors	Accept school mailer (inherited from School Member) Send paycheck Change deductions

inherited child class. If you forget to do this, though, you have two copies of all base class attributes. References to the base class members are now ambiguous unless they are fully qualified using the scope resolution operator ::, as in `student::Name` and `Faculty::Name`.

Inheritance is a mechanism to represent generalization-specialization.

**Table 1-10:** *Graduate Teaching Assistant*

Attributes	Name (inherited from School Member) Address (inherited from School Member) Student number GPA Current class load Tuition owed Name (inherited from School Member) Address (inherited from School Member) Employee number Pay rate Pay period Deductions Social Security Number Medical benefits plan
Behaviors	Accept school mailer (inherited from School Member) Send tuition bill Compute GPA Accept school mailer (inherited from School Member) Send paycheck Change deductions Accept school mailer (inherited from School Member) Send tuition bill Compute GPA

That is, the parent class is a generalized version of the child class. One of the important principles of generalization is the Liskov Substitution Principle (LSP). LSP states that a subclass must be freely substitutable for its superclass. This means that a subclass must continue to act as though it also is an instance of its superclass. Adog may be a specialized, extended form of mammal, but an instance of dog is still a mammal and has all the properties of mammals. LSP requires that subclasses do not constrain superclass behavior, such as by blocking or selectively inheriting some properties.

Almost always, a single inheritance hierarchy is specialized along a single characteristic or a small set of closely related characteristics. Multiple inheritance usually makes sense only if the sets of inheritance

hierarchies are specialized along *orthogonal characteristics* (properties with nothing in common). Otherwise, another solution probably will be more appropriate.

### *Refinement (Templates)*

The refinement relationship takes an incompletely specified entity and adds the previously unspecified aspects. The UML notational guide identifies the following types of refinement:

- Relation between a type and a class that realizes it (realization)
- Relation between an analysis class and a design class (design trace)
- Relation between a high-level construct at a coarse granularity and a lower-level construct at a finer granularity
- Relation between a construct and its implementation at a lower virtual layer, such as the implementation of a type as a collaboration of lower-level objects (implementation)
- Relation between a straightforward implementation of a construct and a more efficient but obscure implementation that accomplishes the same effect (optimization)

Normally, the only use of refinement you'll see is taking a generic, but incomplete, class specification and creating from it an instantiable class. A generic (*template* in C++-speak) is not, strictly speaking, a class. It is a template from which classes may be constructed by adding the missing details. You cannot construct an object from a generic directly—a class must be instantiated first.

Refinement is similar to generalization in that it also creates more specialized classes. At first glance, the distinction seems subtle indeed—in either case you are specializing some entity to get a class. Inheritance is used when you want to specialize how some behavior is performed or how some class is constructed. Generic instantiation is used when you want exactly the same behavior or structure, but applied on a novel component type.

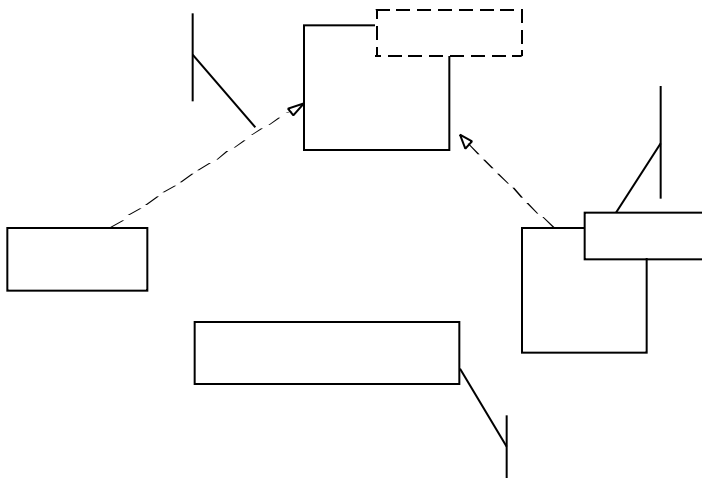
Collection or container classes are a common application of the refinement relationship. A collection class is one that aggregates many component objects. These component objects are usually homogenous (of the same class), or at least from a single inheritance hierarchy. Ex-

actly the same behavior applies regardless of the class being collected. Things you want to be able to do with a collection of classes might be:

- Get the first object in the collection
- Get the next object in the collection
- Add an object to the collection
- Remove an object from the collection

These behaviors are exactly the same regardless of whether you are dealing with a group of bank accounts, photodiode sensors, or automobiles. The behaviors of the objects themselves are vastly different, but the collection itself should behave in the same way nevertheless.

This is difficult to implement using inheritance, but straightforward using generic instantiation. Figure 1-11 shows how generics are represented. They use a dashed line with a closed arrowhead, similar to a generalization. The refining parameters are shown either on the association with a «bind» stereotype<sup>11</sup> or with the parameters shown in the class box between angled brackets.



**Figure 1-11:** *Refinement Relationship*

<sup>11</sup> A *stereotype* is the class of an entity in the UMLmetamodel. Stereotypes are discussed in more detail in the next section.

the refinement look the same except that the base type is defined, and the base type appears inside a box with a solid line. This example declares a generic stack, and then instantiates three different stack classes—one for integers, one for strings, and one for transaction objects.

Note that in Figure 1-11, one of the refined classes has no refinement relationship shown with the generic. It is implied by the use of the generic's name and the inclusion of the replacing parameters enclosed by angled brackets.

---

## 1.5 UML Diagrams and Notation

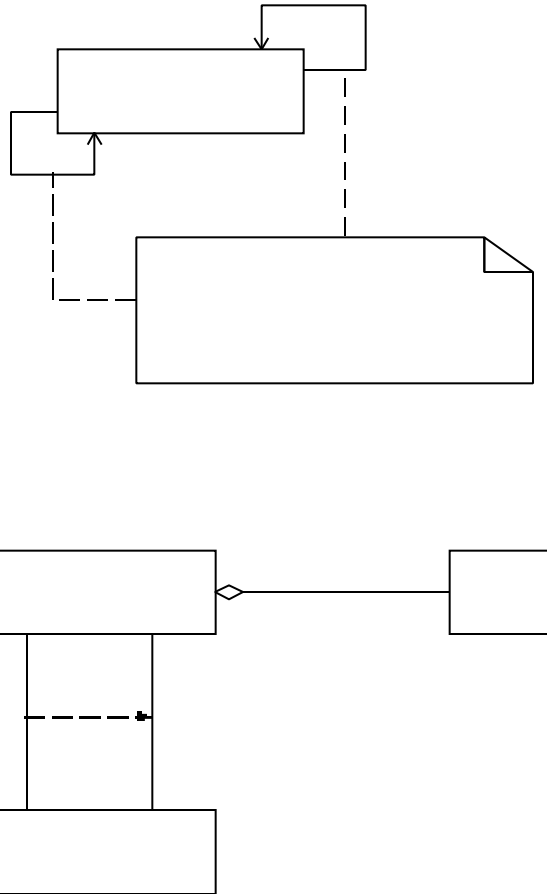
UML has a rich set of notations and semantics. This richness makes it applicable to a wide set of modeling applications and domains. In this chapter we have only scratched the surface. In the coming chapters, new notations within UML will be presented when the context requires them. A concise overview of the notation is provided in the Appendix. Some notational elements we wish to present here include the text note, the constraint, and the stereotype because they will be used in a variety of places throughout the book.

A text note is a diagrammatic element with no semantic impact. It is visually represented as a rectangle with the upper right-hand corner folded down. Text notes are used to provide textual annotations to diagrams in order to improve understanding.

A constraint is some additional condition applied against a modeling element. Timing constraints can be shown on sequence diagrams (discussed in some detail in the next chapter) specifying the time between messages, for example. Constraints are always shown inside curly braces, and may appear inside of text notes.

Figure 1-12 shows text notes and constraints used together. At the top, the class model for a doubly-linked list is shown as a single class with constrained associations. At the bottom, the associations between classes Worker and Team Member are constrained, in that the Manager\_of association is a subset of the Member\_of association.

A *stereotype* is the class of an entity in the UML metamodel. The UML metamodel is the model of UML itself, expressed in UML. Stereotypes provide an important extension mechanism to UML, allowing



**Figure 1-12:** *Constraints in Action*

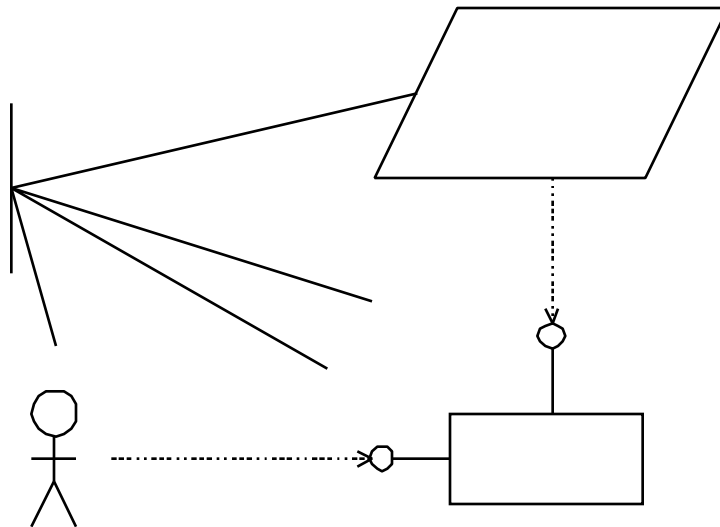
users to extend the modeling language to better address their needs. Each modeling element in UML is represented as a metaclass. A stereotyped metaclass is ultimately derived from an existing UML metaclass. For example, it is possible to create a new (meta)class of the UML *class* construct, which is just like the normal UML class but is extended or specialized. A class that represents the type (interface) of a class is just such a stereotyped class.

The usual notation for stereotypes is to enclose the stereotype in

guillemets<sup>12</sup> preceding the name of the entity; for example “«type» stack,” which is the name of a class providing a stack interface for another implementation. Special icons can be used instead of guillemets for common stereotypes. UML defines a number of common stereotype icons; feel free to add your own application domain specific icons (at the risk of introducing some nonportability, of course).

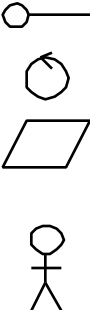
Figure 1-13 illustrates a few common UML stereotype icons. The “lollipop” is the icon for an «type» stereotype, which is a class that provides an interface for another. The stick figure is the icon for an «actor» stereotype, which refers to an external object that interacts with the system. The parallelogram icon is the stereotype for an «active» class, whose instances are the roots of threads.

All modeling entities in the UML can be stereotyped. For example, messages can be stereotyped. Some common stereotypes are shown in Figure 1-14. The coming chapters will introduce more stereotypes as needed.



**Figure 1-13:** *UML Class Stereotypes*

<sup>12</sup> If guillemets are unavailable, then double angled brackets are an acceptable alternative. For example, «type» may be substituted for «type».



**Figure 1-14:** *Some Common UML Stereotypes*

---

## 1.6 A Look Ahead

So far, we have only touched on the defining characteristics of real-time systems and the fundamental aspects of the object-oriented perspective. In the subsequent chapters of this book, we'll apply these ideas to the process of creating real-time embedded applications. The process is broken into the overall process steps of analysis and design. Analysis is subdivided into specification of external requirements and the identification of inherent classes and objects. Design is divided into three parts—architectural, mechanistic, and detailed levels of abstraction. Architectural design specifies the strategic decisions for the overall organization of the system, such as the design of the processor and concurrency models. Mechanistic design is concerned with the medium level of organization—the collaboration of objects to achieve common goals. Detailed design defines the internal algorithms and primitive data structures within classes. All the process steps are required to create efficient, correct designs that meet the system requirements.

---

## 1.7 References

- [1] Leveson, Nancy G., *Safeware: System Safety and Computers*. Reading, MA: Addison Wesley Longman, 1995.
- [2] Neumann, Peter G., *Computer Related Risks*. Reading, MA: Addison Wesley Longman, 1995.
- [3] Ellis, John R., *Objectifying Real-Time Systems*. New York: SIGS Books, 1994.
- [4] Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen, *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice Hall, 1991.