10

Concepts in Object-Oriented Languages

Over the past 30 years, object-oriented programming has become a prominent software design and implementation strategy. The topics covered in this chapter are object-oriented design, four key concepts in object-oriented languages, and the way these language concepts support object-oriented design and implementation.

An *object* consists of a set of operations on some hidden data. An important characteristic of objects is that they provide a uniform way of encapsulating almost any combination of data and functionality. An object can be as small as a single integer or as large as a file system or database. Regardless of its size, all interactions with an object occur by means of simple operations that are called *messages* or *member-function calls*.

If you look in magazines or research journals, you will find the adjective *object-oriented* applied to a variety of languages. As object orientation has become more popular and gained wider commercial acceptance, advocates of specific languages have decided that their favorite language is now object oriented. This has created some amount of confusion about the meaning of object oriented. In this book, we are interested in making meaningful distinctions between different language features and understanding how specific features support different kinds of programming. Therefore, the term *object-oriented language* is used to refer to programming languages that have objects and the four features highlighted in this chapter: dynamic lookup, abstraction, subtyping, and inheritance.

10.1 OBJECT-ORIENTED DESIGN

Object-oriented design involves identifying important concepts and using objects to structure the way that these concepts are embodied in a software system. The following list of steps is taken from one overview of object-oriented design, written by object-oriented design proponent Grady Booch (Object-Oriented Design with Applications, Benjamin/Cummings, 1991):

- Identify the objects at a given level of abstraction.
- Identify the semantics (intended behavior) of these objects.

https://doi.org/10.1017/CBO9780511804175.011 Published online by Cambridge University Press

- Identify the relationships among the objects.
- Implement the objects.

Object-oriented design is an iterative process based on associating objects with components or concepts in a system. The process is iterative because typically we implement an object by using a number of subobjects, just as we typically implement a procedure by calling a number of finer-grained procedures. Therefore, after the important objects in a system are identified and implemented at one level of abstraction, the next iteration will involve identifying additional objects and implementing them. The "relationships among objects" mentioned here might be relationships between their interfaces or relationships between their implementations. Modern object-oriented languages provide mechanisms for using relationships between interfaces and relationships between implementations in the design and implementation process.

The data structures used in the early examples of top-down programming (see Section 9.1) were very simple and remained invariant under successive refinements of the program. When refinement involves replacing a procedure with more detailed procedures, older forms of structured programming languages such as Algol, Pascal, and C are adequate. For more complex tasks, however, both the procedures and the data structures of a program need to be refined together. Because objects are a combination of functions and data, object-oriented languages support the joint refinement of functions and data more effectively than do procedure-oriented languages.

10.2 FOUR BASIC CONCEPTS IN OBJECT-ORIENTED LANGUAGES

All object-oriented languages have some form of object. As mentioned in the preceding section, an object consists of functions and data, accessible only through a specific interface. In common object-oriented languages, including Smalltalk, Modula-3, C++, and Java, the implementation of an object is determined by its *class*. In these languages, we create objects by creating an *instance* of their classes.

The function parts of an object are called *methods* or *member functions*, and the data parts of an object are called *instance variables*, fields, or data members.

Programming languages with objects and classes typically provide dynamic lookup, abstraction, subtyping, and inheritance. These are the four main language concepts for object-oriented programming. They may be summarized in the following manner:

- *Dynamic lookup* means that when a message is sent to an object, the function code (or *method*) to be executed is determined by the way that the object is implemented, not some static property of the pointer or variable used to name the object. In other words, the object "chooses" how to respond to a message, and different objects may respond to the same message in different ways.
- *Abstraction* means that implementation details are hidden inside a program unit with a specific interface. For objects, the interface usually consists of a set of public functions (or *public methods*) that manipulate hidden data.
- *Subtyping* means that if some object `a` has all of the functionality of another object `b`, then we may use `a` in any context expecting `b`.
- *Inheritance* is the ability to reuse the definition of one kind of object to define another kind of object.

https://doi.org/10.1017/CBO9780511804175.011 Published online by Cambridge University Press

These terms are defined and these features are explored in more detail in the following subsections.

There are several forms of object-oriented languages that are not covered directly in this book. One form is the *delegation-based language*. Two delegation-based languages are Dylan, originally designed to program Apple Newton personal digital assistants, and Self, a general-purpose language evolving out of research on implementation of object-oriented languages. In delegation-based languages, objects are defined directly from other objects when new methods are added by means of *method addition* and old methods are replaced by means of *method override*. Although delegation-based languages do not have classes, they do have the four essential characteristics required for object-oriented languages.

10.2.1 Dynamic Lookup

In any object-oriented language, there is some way to invoke the operations associated with an object. A general syntax for invoking an operation on an object, possibly with additional arguments, is

```
object ⟶ operation (arguments)
```

In Smalltalk, this is called "sending a message to an object," whereas in C++ it is called "calling a member function of an object." To avoid switching back and forth between different choices of terminology, we will use the Smalltalk terminology for the remainder of this section. In Smalltalk terminology, a *message* consists of an operation name and set of additional arguments. When a message is sent to an object, the object responds to the message by executing a function called a *method*.

Dynamic lookup means that a method is selected dynamically, at run time, according to the implementation of the object that receives a message. The important property of dynamic lookup is that different objects may implement the same operation differently. For example, the statement

```
x ⟶ add(y)
```

sends the message `add(y)` to the object `x`. If `x` is an integer, then the method (code implementing this operation) may add integer `y` to `x`. If `x` is a set, then the `add` method may insert `y` into the set `x`. These operations have different effects and are implemented differently. However, a single line of code `x ⟶ add(y)` inside a loop could cause integer addition the first time it is executed and set insertion the second time if the value of the variable `x` changes from an integer to a set between one pass through the loop and another.

Dynamic lookup is sometimes confused with overloading, which is a mechanism based on static types of operands. However, the two are very different, as we will see.

https://doi.org/10.1017/CBO9780511804175.011 Published online by Cambridge University Press

Dynamic lookup is a very useful language feature and an important part of object-oriented programming. Consider, for example, a simple graphics program that manipulates pictures containing shapes such as squares, circles, and triangles. Each square object may contain a draw method with code to draw a square, each circle a draw method that contains code to draw a circle, and so on. When the program wants to display a given picture, sending a draw message to each shape in the picture can do this. The part of the program that sends the draw message does not have to know which kind of shape will receive the message. Instead, each shape receiving a draw message will know how to draw that shape. This makes sense because the implementer of a specific shape is in the best position to figure out how to draw that kind of shape.

We can understand some aspects of dynamic lookup and scoping by using a brief comparison with abstract data types. Using an abstract data-type mechanism, we might define matrices as follows:

```
abstype matrix = . . .
  with
      create(. . . ) = . . .
      update(m, i, j, x) = . . . set m(i, j) = x. . .
      add(m1, m2) = . . .
      . . .
  end;
```

A characteristic of this implementation of matrices is that the add function takes two matrices as arguments, with call of the form

```
add(x, y)
```

The declaration of type matrix and associated operations has a specific scope. Within this scope, add refers specifically to the function declared for matrices. Therefore, in an expression add(x,y), both x and y must be matrices. If add were defined for complex numbers in some outer scope, then either the inner declaration hides the outer one or the language must provide some static overloading mechanism.

With objects in a class-based language, we might instead declare matrices as follows:

```
class matrix
   . . .       (representation)
   update(i, j, x) = . . . set (i, j) of *this* matrix . . .
   add(m) = . . . add m to *this* matrix . . .
end
```

The `add` method of a matrix requires one matrix as an argument. The method might be invoked by an expression such as

```
x ⟶ add(y)
```

In this expression, the operation `add` appears to have only one argument, the matrix that is to be added to the matrix `x` receiving the message `add(y)`.

There are several ways that dynamic lookup may be implemented. In one implementation, each object contains a pointer to a method lookup table that associates a method body with each message defined for that object. When a message is sent to an object at run time, the corresponding method is retrieved from that object's method table. Because different objects may have different method lookup tables, sending the same message to different objects may result in the execution of different code.

It is also possible to think of dynamic lookup as a run-time form of overloading. More specifically, we can think of each method name as the name of an overloaded function. When a message `m` is sent to an object named by variable `x`, then `x` is treated as the first argument of an overloaded function named `m`. Unlike traditional overloading, though, the code to execute must be chosen according to the run-time value of `x`. In contrast, traditional overloading uses the static type of a variable `x` to decide which code to use.

Dynamic lookup is an important part of Smalltalk, C++, and Java. In Smalltalk and Java, method lookup is done dynamically by default. In C++, only *virtual* member functions are selected dynamically.

There is a family of object-oriented languages that is based on the "run-time overloading" view of dynamic lookup. The most prominent design of this form is the common Lisp object system, sometimes referred to by the acronym CLOS. In CLOS, an expression corresponding to

```
x ⟶ f(y,z)
```

is treated as a call `f(x, y, z)` to an overloaded function with three arguments. Although ordinary dynamic lookup would select a function body for `f` based on the implementation of `x` alone, CLOS method lookup uses all three arguments. This feature is sometimes called *multiple dispatch* to distinguish it from more conventional single-dispatch languages in which only one of the arguments of a function (the object receiving the message) determines the function body that is called at run time.

Multiple dispatch is useful for implementing operations such as equality, in which the appropriate comparisons to use depend on the dynamic type of both the receiver object and the argument object. Although multiple dispatch is in some ways more general than the single dispatch found in Smalltalk, C++, and Java, there is also some loss of encapsulation. Specifically, to define a function on different kinds of arguments, that function must have access to the internal data of each function argument.

https://doi.org/10.1017/CBO9780511804175.011 Published online by Cambridge University Press

Because single-dispatch languages are the object-oriented mainstream, we focus on single-dispatch languages in this book.

10.2.2 Abstraction

As discussed in Chapter 9, abstraction involves restricting access to a program component according to its specified interface. In most modern object-oriented languages, access to an object is restricted to a set of public operations that are chosen by the designer and implementer of the object. For example, in a program that manipulates geometric shapes, each shape could be represented by an object. We could implement an object representing a circle by storing the center and radius of the circle. The designer of circle objects could choose to make a function that changes the center of the circle part of the interface or choose not to put such a function in the interface. If there is no public function for changing the center of a circle, then no client code could change the center of a circle, as client code can manipulate objects only through their interface.

Abstraction based on objects is similar in many ways to abstraction based on abstract data types: Objects and abstract data types both combine functions and data, and abstraction in both cases involves distinguishing between a public interface and private implementation. However, other features of object-oriented languages make abstraction in object-oriented languages more flexible than abstraction in which abstract data types are used. One way of understanding the flexibility of object-oriented languages is by looking at the way that relationships between similar abstractions can be used to advantage.

Consider the following two abstract data types, written in ML syntax. The first is an abstract data type of queues, the second an abstract data type of priority queues. For simplicity, both queues and priority queues are defined for only integer data:

```
exception Empty;
abstype queue = Q of int list
   with
        fun mk_Queue() = Q(nil)
        and is_empty(Q(l)) = l=nil
        and add(x,Q(l)) = Q(l @ [x])
        and first (Q(nil)) = raise Empty | first (Q(x::l)) = x
        and rest (Q(nil)) = raise Empty  | rest (Q(x::l)) = Q(l)
        and length (Q(nil)) = 0          | length (Q(x::l))= 1 + length (Q(l))
    end;
```

In this abstract data type, a queue is represented by a list. The add operation uses the ML append operator @ to add a new element to the end of a list. The first and the rest operations read and remove an element from the front of a list. Because client code cannot manipulate the representation of a queue directly, the implementation maintains an invariant: List elements appear in first-in/first-out order, regardless of how queues are used in client programs.

A priority queue is similar to a queue, except that elements are removed according to some preference ordering. More specifically, some priority is given to elements, and the first and the remove operations read and remove the queue elements that have highest priority:

```
abstype pqueue = Q of int list
  with
     fun mk_PQueue() = Q(nil)
     and is_empty(Q(l)) = l=nil
     and add(x,Q(l)) =
         let fun insert(x,nil) = [x:int]
         |  insert(x,y::l) = if x<y then x::y::l else y::insert(x,l)
         in Q(insert(x,l)) end
     and first (Q(nil)) = raise Empty | first (Q(x::l)) = x
     and rest (Q(nil)) = raise Empty  | rest (Q(x::l)) = Q(l)
     and length (Q(nil)) = 0          | length (Q(x::l))= 1 + length (Q(l))
  end;
```

The interface of an abstract data type is the list of public functions and their types. Queues and priority queues have the same interface: Both have the same number of operations, the operations have the same names, and each operation has the same type in both cases, except for the difference between the type names pqueue and queue. The point of this example is that, although the interfaces of queues and priority queues are identical, this correspondence is not used in traditional languages with abstract data types. In contrast, if we implement queues and priority queues in an object-oriented language, then we can take advantage of the similarity between the interfaces of these data structures.

A drawback to the kind of abstract data types used in ML and similar languages becomes apparent when we consider a program that uses both queues and priority queues. For example, suppose that we build a system with several wait queues, such as a hospital. In a hospital billing department, customers are served on a first-come, first-serve basis. However, in a hospital emergency room, patients are treated in order of the severity of their injuries or ailments. In a hospital program, we might like to treat priority queues and ordinary queues uniformly. For example, we might wish to count the total number of people waiting in any line in the hospital. To write this code, we would like to have a list of all the queues (both priority and ordinary) in the hospital and sum the lengths of all the queues in the list. However, if the length operation is different for queues and priority queues, we have to decide whether to call q_length or pq_length, even though the correct operation is uniquely determined by the data. This shortcoming of ordinary abstract data types is eliminated in object-oriented programming languages by a combination of subtyping (see Subsection 10.2.3) and dynamic lookup.

The implementation of priority queues shows us another drawback of traditional abstract data types. Although the priority queue add function is different from the

https://doi.org/10.1017/CBO9780511804175.011 Published online by Cambridge University Press

queue add function, the other five functions have identical implementations. In an object-oriented language, we may use inheritance (see Subsection 10.2.4) to define priority_queue from queue (or vice versa), giving only the new add function and reusing the other functions.

10.2.3 Subtyping

Subtyping is a relation on types that allows values of one type to be used in place of values of another. Although it is simplest to describe subtyping in the context of statically typed programming languages, there is also an implicit subtyping relation in untyped languages. We will discuss subtyping assuming that we are in a typed language, in most of this section, turning to untyped languages in the final paragraphs.

In most typed languages, the application of a function f to an argument x requires some relation between the type of f and the type of x. The most common case is that f must be a function from type A to type B, for some A and B, and x must be a variable or expression of type A. We can think of this type comparison as a comparison for equality: The type checker finds a type A → B for function f and a type C for x, and checks that A=C.

In languages with subtyping, there is a subtype relation on types. The basic principle associated with subtyping is *substitutivity*: If A is a subtype of B, then any expression of type A may be used without type error in any context that requires an expression of type B. We write A <: B to indicate that A is a subtype of B.

With subtyping, the subtype relation is used in place of equality in type checking. Specifically, to type the application of f to argument x, the type checker finds a type A → B for function f and a type C for x, and checks that C is a subtype of A.

The primary advantage of subtyping is that it permits uniform operations over various types of data. For example, subtyping makes it possible to have heterogeneous data structures that contain objects that belong to different subtypes of some common type. Consider as an example a queue containing various bank accounts to be balanced. These accounts could be savings accounts, checking accounts, investment accounts, and so on. However, if each type of account is a subtype of bank_account, then a queue of elements of type bank_account can contain all types of accounts.

Subtyping in an object-oriented language also allows functionality to be added without modifying general parts of a system. If objects of a type B lack some desired behavior, then we may wish to replace objects of type B with objects of another type A that have the desired behavior. In many cases, the type A will be a subtype of B. By designing the language so that substitutivity is allowed, one may add functionality in this way without any other modification to the original program.

This use of subtyping helps in building a series of prototypes of an airport scheduling system. In an early prototype, one would define a class airplane with methods such as position, orientation, and acceleration that would allow a control-tower object to affect the approach of an airplane. In a later prototype, it is likely that different types of airplanes would be modeled. If we add classes for Boeing 757s and Beechcrafts, these would be subtypes of airplane, containing extra methods and fields reflecting features specific to these aircraft. By virtue of the subtyping relation, Beechcrafts and Boeings are subtypes of airplane, and the general control algorithms that apply to all airplanes can be used for Beechcrafts and Boeings without modification.

https://doi.org/10.1017/CBO9780511804175.011 Published online by Cambridge University Press

10.2.4 Inheritance

Inheritance is a language feature that allows new objects to be defined from existing ones. We discuss the form of inheritance that appears in most class-based object-oriented languages by using a neutral notation. The following class A defines objects with private data v and public methods f and g. We define the class B by inheriting the declarations of A, redefining g, and adding a private variable w:

```
class A =
    private
        val v = ...
    public
        fun f(x) = ... g(...) ...
        fun g(y) = ... original definition ...
    end;
class B = extend A with
    private
        val w = ...
    public
        fun g(y) = ... new definition ...
    end;
```

In principle, inheritance can be implemented by code duplication. For every object or class of objects defined by inheritance, there is a corresponding definition that does not use inheritance; it is obtained by expansion of the definition so that inherited code is duplicated. The importance of inheritance is that it saves the effort of duplicating (or reading duplicated) code and that, when one class is implemented by inheriting from another, changes to one affect the other. This has a significant impact on code maintenance and modification.

A straightforward implementation of inheritance that avoids code duplication is to build linked data structures of method lookup tables. More specifically, for each class, a lookup table can contain the list of operations associated with the class. When one class inherits from another, then the second class can contain a pointer to the lookup table of the first. If a method is inherited, it can be found in the method lookup table of the class in which it was originally defined. A scheme of this form is used in the implementation of Smalltalk, as we will see in Subsection 11.5.3.

A significant optimization may be made in statically typed languages such as C++, in which the set of possible messages to each object can be determined at compile time. If lookup tables can be constructed so that all subclasses of a given class store repeated pointers in the same relative positions, then the offset of a method within a lookup table can be computed at compile time. This reduces the cost of method lookup to a simple indirection without a search, followed by an ordinary function call. We will look at this implementation of inheritance in more detail in Subsection 12.3.3.

Inheritance and Abstraction

In ordinary modules or abstract data types, there are two views of an abstraction: the client view and the implementation view. With inheritance, there are three views

https://doi.org/10.1017/CBO9780511804175.011 Published online by Cambridge University Press

of classes: the implementation view, the client view, and the inheritance view. The inheritance view is the view of classes that inherit from a class. Because object definitions have two external clients, there are two interfaces to the outside: The *public* interface lists what the general client may see, whereas the *protected* interface lists what inheritors may see. (This terminology comes from C++.) In most languages, the public interface is a subset of the protected one. In Smalltalk, these interfaces are generated automatically: The public interface includes all the methods of an object, whereas the protected interface is all methods and all instance variables (data). In C++, the programmer explicitly declares which components of an object are public, which are protected, and which are *private* and visible only in the class definition itself. We will discuss this in more detail in Section 12.3.

10.2.5 Closures as Objects

The first characteristic of objects, dynamic lookup,is also provided by records (in Pascal or ML terminology) or structs (in C terminology). In a language with closures, we can simulate objects by using records that have function components. If an object has private data (or functions), then we can hide them by using static scoping. This leads us to look at record closures as a simple first model of objects. It turns out to be instructive to see how useful this notion of object is and where it falls short.

To see the similarities, consider the following ML code:

```
exception Empty;
fun newStack(x) =
  let val store = ref [x]
  in
     {push = fn(y) => store := y ::(!store),
     pop = fn() => case !store of
                    nil = raise Empty
                 |  (y::ys) = (store := ys; y)
     }
     end;
val myStack = newStack(0);
#push(myStack)(1);
#pop(myStack)( );
```

The notation #field_name(record_value) is ML notation for field selection. In Pascal-like syntax, this expression would be written as record_value.field_name.

The function newStack returns a record with two function components, the first called push, the second called pop. Because the fields of this record contain functions, they are represented at run time as closures. The environment pointers for these closures point to the activation record for the newStack function, which stores the local data store. The initial value of store is a list containing only the initial element passed as an argument to newStack. If you draw out the activation records and closures, you will obtain a diagram that is very similar to the ones we will be drawing to represent

https://doi.org/10.1017/CBO9780511804175.011 Published online by Cambridge University Press

objects. However, most object-oriented languages optimize the representation in one or more ways.

Because closures and objects have essentially the same functionality, it is reasonable to wonder why we talk about "object-oriented" programming, instead of "closure-oriented" programming. In other words, what do object-oriented programming languages have that languages like ML lack? The answer is subtyping and inheritance. If you try to translate an object-oriented program into a non-object-oriented language, you will appreciate the language support for subtyping and inheritance.

10.2.6 Inheritance Is Not Subtyping

Perhaps the most common confusion surrounding object-oriented languages is the difference between subtyping and inheritance. The simplest distinction between subtyping and inheritance is this: *Subtyping is a relation on interfaces, inheritance is a relation on implementations.*

One reason subtyping and inheritance are often confused is that some class mechanisms combine the two. A typical example is C++, in which A will be recognized by the compiler as a subtype of B only if B is a public base class of A. Combining subtyping and inheritance is an elective design decision, however; C++ could have been designed differently without linking subtyping and public base classes in this way.

We may see that, in principle, subtyping and inheritance do not always go hand-in-hand by considering an example suggested by object-oriented researcher, Alan Snyder. Suppose we are interested in writing a program that requires dequeues, stacks, and queues. These are three similar kinds of data structures, with the following basic characteristics:

- *Queues*: Data structures with insert and delete operations, such that the first element inserted is the first one removed (first-in, first-out),
- *Stacks*: Data structures with insert and delete operations, such that the first element inserted is the last one removed (last-in, first-out),
- *Dequeues*: Data structures with two insert and two delete operations. A dequeue, or doubly ended queue, is essentially a list that allows insertion and deletion from each end. If an element is inserted at one end, then it will be the first one returned by a series of removes from that end and the last one returned by a series of removes from the opposite end.

An important part of the relationship among stacks, queues, and dequeues is that a dequeue can serve as both a stack and a queue. Specifically, suppose a dequeue d has insert operations insert_front and insert_rear and delete operations delete_front and delete_rear. If we use only insert_front and delete_rear, we have a queue. However, if we use insert_front and delete_front, we have a stack.

One way to implement these three classes is first to implement dequeue and then implement stack and queue by appropriately restricting (and perhaps renaming) the operations of dequeue. For example, we may obtain stack from dequeue by limiting access to those operations that add and remove elements from one end of a dequeue. Similarly, we may obtain queue from dequeue by restricting access to those operations that add elements at one end and remove them from the other. This method of defining stack and queue by inheriting from dequeue is possible in C++ through the use of

https://doi.org/10.1017/CBO9780511804175.011 Published online by Cambridge University Press

private inheritance. (This is not a recommended style of implementation; this example is used simply to illustrate the differences between subtyping and inheritance.)

Although stack and queue may be implemented from dequeue, they are not subtypes of dequeue. Consider a function f that takes a dequeue d as an argument and then adds an element to both ends of d. If stack or queue were a subtype of dequeue, then function f should work equally well when given a stack s or a queue q. However, adding elements to both ends of either a stack or a queue is not legal; hence, neither stack nor queue is a subtype of dequeue. In fact, the reverse is true. Dequeue is a subtype of both stack and queue, as any operation valid for either a stack or a queue would be a legal operation on a dequeue. Thus, inheritance and subtyping are different relations in principle: it makes perfect sense to define stack and queue by inheriting from dequeue, but dequeue is a subtype of stack and queue, not the other way around.

A more detailed comparison of the two mechanisms appears in Section 11.7, in which the inheritance and subtyping relationships among Smalltalk collection classes are analyzed.

10.3 PROGRAM STRUCTURE

There are some systematic differences between the structure of function-oriented (or procedure-oriented) programs and object-oriented programs. One of the main differences is in the organization of functions and data. In a function-oriented program, data structures and functions are declared separately. If a function will be applied to many types of data, then it is common to use some form of case or switch statement within the function body. In an object-oriented program, functions are associated with the data they are designed to manipulate. Using dynamic lookup, the programming language implementation will select the correct function for each kind of data. This basic difference between function-oriented and object-oriented programs is illustrated by a comparison of Example 10.1 and Example 10.2. A longer example illustrating this point, written in C and C++, appears in Appendix B.1.

In both Examples 10.1 and 10.2, we consider a hospital simulation. The data in these examples represent doctors, nurses, and orderlies. The functions that will be applied to these data include a function to display information about a hospital employee and a function to set or determine the pay of an employee.

Example 10.1 Conventional Function-Oriented Organization

In a conventional function-oriented program, operations are grouped into function. If we want a single function to display information about all types of hospital employees, then we may use run-time tests to determine how to apply each operation to the given data. In outline, codes for display and pay functions might look like this:

```
display(x) =
  case type(x) of
        Doctor  : [“display Doctor” ]
        Nurse   : [“display Nurse” ]
        Orderly : [“display Orderly” ]
  end;
```

https://doi.org/10.1017/CBO9780511804175.011 Published online by Cambridge University Press

```
end;
pay(x) =
    case type(x) of
        Doctor  : [“pay Doctor a lot” ]
        Nurse   : [“pay Nurse less” ]
        Orderly : [“pay Orderly less than that” ]
    end;
end;
```

Example 10.2 Object-Oriented Organization

In an object-oriented program, functions are grouped with the data they are designed to manipulate. For the hospital example, the doctor, nurse, and orderly classes will contain the code for the two functions. In outline, this produces the following program organization:

```
class Doctor =
     display = “Display Doctor”;
     pay = “pay Doctor a lot ”;
end;
class Nurse =
     display = “display Nurse”;
     pay = “pay Nurse less”;
end;
class Orderly =
     display = “display Orderly”;
     pay = “pay Orderly less than that”;
end;
```

Comparison of Examples 10.1 and 10.2

The data and operations used in Examples 10.1 and 10.2 may be arranged into the following matrix. In the conventional function-oriented organization, the code is arranged by row into functions that work for all kinds of data. In the object-oriented organization, the code is arranged by column, grouping each function case with the data it is designed for.

Operation	Doctor	Nurse	Orderly
Display	Display Doctor	Display Nurse	Display Orderly
Pay	Pay Doctor	Pay Nurse	Pay Orderly

In the function-oriented organization, it is relatively easy to add a new operation, such as PayBonus or Promote, but difficult to add a new kind of data, such as Administrator or Intern. In the object-oriented organization, it is easy to add new data, such as Administrator or Intern, but more cumbersome to add new operations such as PayBonus or Promote because this involves changes to every class.

https://doi.org/10.1017/CBO9780511804175.011 Published online by Cambridge University Press

10.4 DESIGN PATTERNS

The design pattern method is a popular approach to software design that has developed along with the rise in popularity of object-oriented programming. In basic terms, a *design pattern* is a general solution that has come from the repeated addressing of similar problems. Design patterns are not solutions developed from first principles or generic code that can simply be instantiated for a variety of purposes. Instead, a design pattern is a guideline or approach to solving a kind of problem that occurs in a number of specific forms. Solutions based on a design pattern can be similar; applying a design pattern to a specific situation can require some thought.

The concept of a design pattern can be used in any design discipline, such as mechanical design or architecture. The work of architect Christopher Alexander is often cited as an inspiration for software design patterns. Here is an architectural example, excerpted from one of Alexander's books (*A Pattern Language: Towns, Buildings, Construction, Oxford Univ. Press, 1977*). This passage includes both a description of the problem context and a solution developed as a result of experience:

> *Sitting Circle*
> *... A group of chairs, a sofa and a chair, a pile of cushions – these are the most obvious things in everybody's life – and yet to make them work, so people become animated and alive in them, is a very subtle business. Most seating arrangements are sterile, people avoid them, nothing ever happens there. Others seem somehow to gather life around them, to concentrate and liberate energy. What is the difference between the two?*
> *... Therefore, place each sitting space in a position which is protected, not cut by paths or movements, roughly circular, made so that the room itself helps suggest the circle – not too strongly – with paths and activities around it, so that people naturally gravitate toward the chairs when they get into the mood to sit. Place the chairs and cushions loosely in the circle, and have a few too many.*

When programmers find that they have solved the same kind of problem over and over again in slightly different ways but using essentially the same design ideas, they may try to identify the general design pattern of their solutions. The popularity of this process has led to the identification of a large number of software design patterns. To quote pattern advocate Jim Coplien, a *good* pattern does the following:

- *It solves a problem*: Patterns capture solutions, not just abstract principles or strategies.
- *It is a proven concept*: Patterns capture solutions with a track record, not theories or speculation.
- *The solution isn't obvious*: Many problem-solving techniques (such as software design paradigms or methods) try to derive solutions from first principles. The best patterns *generate* a solution to a problem indirectly – a necessary approach for the most difficult problems of design.
- *It describes a relationship*: Patterns don't just describe modules, but describe deeper system structures and mechanisms.

https://doi.org/10.1017/CBO9780511804175.011 Published online by Cambridge University Press

- *The pattern has a significant human component (minimize human intervention).* All software serves human comfort or quality of life; the best patterns explicitly appeal to aesthetics and utility.

Beyond reading about general principles of design patterns, the best way to learn about patterns is to study some examples and use patterns in your programming. Here are a couple of examples. You can find many more in the books and web pages devoted to design patterns. A widely used book is *Design Patterns: Elements of Reusable Object-Oriented Software* by E. Gamma, R. Helm, R. Johnson, and J. Vlissides (Addison-Wesley, 1994).

Example 10.3 Singleton Design Pattern

The singleton design pattern is a *creational* design pattern, meaning that it is a pattern that is used for creating objects in a certain way. Here is a brief overview of the singleton pattern, that uses the kind of subject headings that are commonly used in books and other presentations of design patterns.

Motivation

The singleton pattern is useful in situations in which there should be a single instance (object) of a class. This pattern gives a class direct control over how many instances can be created. This is better than making the programmer responsible for creating only one instance, as the restriction is built into the program.

Implementation

Only one class needs to be written to implement the singleton pattern. The class uses encapsulation to keep the class constructor (the function that returns new objects of the class) hidden from client code. The class has a public method that calls the constructor only if an object of the class has not already been created. If an object has been created, then the public function returns a pointer to this object and does not create a new object.

Sample Code

Here is how a generic singleton might be written in C++. Readers who are not familiar with C++ may wish to scan the explanation and return to this example after reading Chapter 12. The interface to class Singleton provides a public method that lets client code ask for an instance of the class:

```
class Singleton {
    public:
        static Singleton* instance(); // function that returns an instance
    protected:
        Singleton();                  // constructor is not made public
    private:
        static Singleton* _instance;  // private pointer to single object
};
```

Here is the implementation. Initially, the private pointer _instance is set to 0. In the

https://doi.org/10.1017/CBO9780511804175.011 Published online by Cambridge University Press

implementation of public method instance(), a new object is created and assigned to _instance only if a previous call has not already created an object of this class:

```
Singleton* Singleton::_singleton = 0
Singleton* Singleton::instance() {
    if (_instance = = 0){ _instance = new Singleton; }
    return _instance;
}
```

Example 10.4 Façade

Façade is a *structural object* pattern, which means it is a pattern related to composing objects into larger structures containing many objects.

Motivation

The *façade* pattern provides a single object for accessing a set of objects that have been combined to form a structure. In effect, the façade provides a higher-level interface to a collection of objects, making the collection easier to use.

Implementation

There is a façade class, defined for a set of classes that are used to make up a structure “behind” the facade. In a typical use, a façade object has relatively little actual code, passing most calls to objects in the structure behind the façade.

Example of Façade Pattern

Façade is a very common pattern when a task is accomplished by a combination of the results of a number of subtasks. For example, a compiler might be constructed by implementation of a lexical scanner, parser, semantic analyzer, and other phases indicated in the figure in Subsection 4.1.1. If each phase is implemented as an object with methods that perform its main functions, then the compiler itself will be a façade object that takes a program as input and uses the separate objects that are implementing each phase to compile the program. A user of the compiler may see the interface presented by the compiler object. This is a more useful interface than the more detailed interfaces to the constituent objects that are hidden behind the façade.

10.5 CHAPTER SUMMARY

This chapter contains a short overview of object-oriented design and summarizes the four basic concepts associated with object-oriented languages: dynamic lookup, abstraction, subtyping, and inheritance.

- *Dynamic lookup* means that when a message is sent to an object, the function code (or *method*) that is executed is determined by the way that the object is implemented. Different objects may respond to the same message in different ways.

- *Abstraction* means that implementation details are hidden inside a program unit with a specific interface. The interface of an object is usually a set of public functions (or *public methods*) that manipulate hidden data.
- *Subtyping* means that if some object a has all of the functionality of another object b, then we may use a in any context expecting b.
- *Inheritance* is the ability to reuse the definition of one kind of object to define another kind of object.

In conventional languages that implement closures and allow records to contain functions, records provide a form dynamic lookup and abstraction. Subtyping and inheritance, in the form needed to support object-oriented programming, are generally not found in conventional languages.

Many people confuse subtyping and inheritance. As the term is used in this book, subtyping is a relation on types that allows values of one type to be used in place of values of another. (In Section 11.7, Smalltalk is used to discuss subtyping in a language that does not have a static type system.) As the term is used in this book, inheritance allows new objects to be defined from existing ones. In class-based languages, inheritance allows the implementation of one class to be reused as part of the implementation of another. The simplest way to keep subtyping and inheritance straight is to remember this: *Subtyping is a relation on interfaces* and *inheritance is a relation on implementations.*

In Section 10.3, the difference between the organizational structure of object-oriented programs and the organizational structure of conventional programs is summarized. In conventional languages, functions are designed to operate on many types of data. In object-oriented programs, functions can be written to operate on a single type of data, with dynamic lookup finding the right function at run time.

In Section 10.4, we looked at the basic idea behind design patterns and saw two examples, singleton and façade. A design pattern is a general solution that has come from the repeated addressing of similar problems. The design pattern method is a popular approach to software design that has evolved along with object-oriented programming.

10.6 LOOKING FORWARD: SIMULA, SMALLTALK, C++, JAVA

In the next three chapters, we will look at four object-oriented languages:

- *Simula*, the first object-oriented language. The object model in Simula was based on procedure activation records, with objects originally described as procedures that return a pointer to their own activation record. There was no abstraction in Simula 67, but a later version incorporated abstraction into the object system. Simula was an important inspiration for C++.
- *Smalltalk*, a dynamically typed object-oriented language. Many object-oriented ideas originated or were popularized by the Smalltalk group, which built on Alan Kay's then-futuristic idea of the Dynabook. The Dynabook, which was never built by this group, was intended to be a small portable computer capable of running a user-friendly programming language. We will look at the Smalltalk implementation of method lookup and later compare this with C++.
- C++, a widely used statically typed object-oriented language. This language is

https://doi.org/10.1017/CBO9780511804175.011 Published online by Cambridge University Press

designed for efficiency around the principle that programs that do not use a certain feature should run as efficiently as programs written in a language without that feature. A significant design constraint was backward compatibility with C.

- *Java*, a modern language design in which security and portability are valued as much as efficiency. Some interesting features are interfaces, which provide explicit support for abstract base classes, and run-time class loading, intended for use in a distributed environment.

Because there is not enough time to study all aspects of each language, we will concentrate on a few important or distinctive features of each one. One general theme in our investigation of these languages is the trade-off between language features and implementation complexity.

Simula is primarily important as a historical language and for the way it illustrates the relationship between objects and activation records. Of the remaining three languages, Smalltalk represents one extreme and C++ the other. Smalltalk is extremely flexible and based on the notion that everything is an object. C++, on the other hand, is defined to favor efficiency over conceptual simplicity. Although C++ provides objects, many features of C++ are inherited from C and are not based on objects. Java is a compromise between Smalltalk and C++ in the sense that the flexibility of the implementation and organization around objects are closer to Smalltalk than to C++. Java also contains features not found in either of the other languages, such as dynamic class loading and a typed intermediate language.

EXERCISES

10.1 Expression Objects

We can represent expressions given by the grammar

$$e \quad ::= \quad num \mid e + e$$

by using objects from a class called expression. We begin with an "abstract class" called expression. Although this class has no instances, it lists the operations common to all kinds of expressions. These are a predicate telling whether there are subexpressions, the left and right subexpressions (if the expression is not atomic), and a method computing the value of the expression:

```
class expression() =
    private fields:
        (* none appear in the _interface_ *)
    public methods:
        atomic?()    (* returns true if no subexpressions *)
        lsub()       (* returns "left" subexpression if not atomic *)
        rsub()       (* returns "right" subexpression if not atomic *)
        value()      (* compute value of expression *)
end
```

Because the grammar gives two cases, we have two subclasses of expression, one for numbers and one for sums:

```
class number(n) = extend expression() with
    private fields:
```

https://doi.org/10.1017/CBO9780511804175.011 Published online by Cambridge University Press

```
            val num = n
        public methods:
            atomic?() = true
            lsub    () = none (* not allowed to call this, *)
            rsub    () = none (* because atomic?() returns true *)
            value   () = num
    end
    class sum(e1, e2) = extend expression() with
        private fields:
            val left = e1
            val right = e2
        public methods:
            atomic?() = false
            lsub    () = left
            rsub    () = right
            value   () = ( left.value() ) + ( right.value() )
    end
```

(a) *Product Class:* Extend this class hierarchy by writing a prod class to represent product expressions of the form

$$e \quad ::= \quad \ldots \mid e * e.$$

(b) *Method Calls:* Suppose we construct a compound expression by

```
val a = number(3);
val b = number(5);
val c = number(7);
val d = sum(a,b);
val e = prod(d,c);
```

and send the message value to e. Explain the sequence of calls that are used to compute the value of this expression: e.value(). What value is returned?

(c) *Unary Expressions:* Extend this class hierarchy by writing a square class to represent squaring expressions of the form

$$e \quad ::= \quad \ldots \mid e^2.$$

What changes will be required in the expression interface? What changes will be required in subclasses of expression? What changes will be required in functions that use expressions?* What changes will be required in functions that do not use expressions? (Try to make as few changes as possible to the program.)

(d) *Ternary Expressions:* Extend this class hierarchy by writing a cond class to represent conditionals† of the form

$$e \quad ::= \quad \ldots \mid e?e : e$$

What changes will be required if we wish to add this ternary operator? [As in part (c), try to make as few changes as possible to the program.]

[*] Keep in mind that not all functions simply want to evaluate entire expressions. They may call the other methods as well.

[†] In C, conditional expressions a?b:c evaluate a and then return the value of b if a is nonzero or return the value of c if a is zero.

https://doi.org/10.1017/CBO9780511804175.011 Published online by Cambridge University Press

(e) *N-Ary Expressions:* Explain what kind of interface to expressions we would need if we would like to support atomic, unary, binary, ternary and *n*–ary operators without making further changes to the interface. In this part of the problem, we are not concerned with minimizing the changes to the program; instead, we are interested in minimizing the changes that may be needed in the future.

10.2 Objects vs. Type Case

With object-oriented programming, classes and objects can be used to avoid "type-case" statements. Here is a program in which a form of case statement is used that inspects a user-defined type tag to distinguish between different classes of shape objects. This program would not statically type check in most typed languages because the correspondence between the tag field of an object and the class of the object is not statically guaranteed and visible to the type checker. However, in an untyped language such as Smalltalk, a program like this could behave in a computationally reasonable way:

```
enum shape_tag {s_point, s_circle, s_rectangle };

class point {
  shape_tag tag;
  int x;
  int y;

  point (int xval, int yval)
    { x = xval; y = yval; tag = s_point; }
  int x_coord () { return x; }
  int y_coord () { return y; }
  void move (int dx, int dy) { x += dy; y += dy; }
};
class circle {
  shape_tag tag;
  point c;
  int r;

  circle (point center, int radius)
    { c = center; r = radius; tag = s_circle }
  point center () { return c; }
  int radius () { return radius; }
  void move (int dx, int dy) { c.move (dx, dy); }
  void stretch (int dr) { r += dr; }
};

class rectangle {
  shape_tag tag;
  point tl;
  point br;

  rectangle (point topleft, point botright)
    { tl = topleft; br = botright; tag = s_rectangle; }
  point top_left () { return tl; }
  point bot_right () { return br; }
  void move (int dx, int dy) { tl.move (dx, dy); br.move (dx, dy); }
  void stretch (int dx, int dy) { br.move (dx, dy); }
};
/* Rotate shape 90 degrees. */
void rotate (void *shape) {
  switch ((shape_tag *) shape) {
    case s_point:
    case s_circle:
      break;
    case s_rectangle:
      {
        rectangle *rect = (rectangle *) shape;
        int d = ((rect->bot_right ().x_coord ()
                  - rect->top_left ().x_coord ()) -
                 (rect->top_left ().y_coord ()
                  - rect->bot_right ().y_coord ()));
        rect->move (d, d);
        rect->stretch (-2.0 * d, -2.0 * d);
      }
    }
}
```

(a) Rewrite this so that, instead of rotate being a function, each class has a rotate method and the classes do not have a tag.

(b) Discuss, from the point of view of someone maintaining and modifying code, the differences between adding a triangle class to the first version (as previously written) and adding a triangle class to the second [produced in part (a) of this question].

(c) Discuss the differences between changing the definition of rotate (say, from 90° to the left to 90° to the right) in the first and the second versions. Assume you have added a triangle class so that there is more than one class with a nontrivial rotate method.

10.3 Visitor Design Pattern

The extension and maintenance of an object hierarchy can be greatly simplified (or greatly complicated) by design decisions made early in the life of the hierarchy. This question explores various design possibilities for an object hierarchy representing arithmetic expressions.

The designers of the hierarchy have already decided to structure it as subsequently shown, with a base class Expression and derived classes IntegerExp, AddExp, MultExp, and so on. They are now contemplating how to implement various operations on Expressions, such as printing the expression in parenthesized form or evaluating the expression. They are asking you, a freshly minted language expert, to help.

The obvious way of implementing such operations is by adding a method to each class for each operation. The Expression hierarchy would then look like:

```
class Expression
{
    virtual void parenPrint();
    virtual void evaluate();
    //...
```

https://doi.org/10.1017/CBO9780511804175.011 Published online by Cambridge University Press

```
}
class IntegerExp : public Expression
{
    virtual void parenPrint();
    virtual void evaluate();
    //. . .
}
class AddExp : public Expression
{
    virtual void parenPrint();
    virtual void evaluate();
    //. . .
}
```

Suppose there are *n* subclasses of Expression altogether, each similar to IntegerExp and AddExp shown here. How many classes would have to be added or changed to add each of the following things?

(a) A new class to represent product expressions.

(b) A new operation to graphically draw the expression parse tree.

Another way of implementing expression classes and operations uses a pattern called the visitor design pattern. In this pattern, each operation is represented by a visitor class. Each visitor class has a visitCLS method for each expression class CLS in the hierarchy. The expression class CLS is set up to call the visitCLS method to perform the operation for that particular class. Each class in the expression hierarchy has an accept method that accepts a visitor as an argument and "allows the visitor to visit the class and perform its operation." The expression class does not need to know what operation the visitor is performing.

If you write a visitor class ParenPrintVisitor to print an expression tree, it would be used as follows:

```
Expression *expTree = . . .some code that builds the expression tree. . .;
Visitor *printer = new ParenPrintVisitor();
expTree->accept(printer);
```

The first line defines an expression, the second defines an instance of your ParenPrintVisitor class, and the third passes your visitor object to the accept method of the expression object.

The expression class hierarchy that uses the visitor design pattern has this form, with an accept method in each class and possibly other methods:

```
class Expression
{
    virtual void accept(Visitor *vis) = 0; //Abstract class
    //...
}
class IntegerExp : public Expression
{
    virtual void accept(Visitor *vis) {vis->visitIntExp(this);};
    //...
}
```

https://doi.org/10.1017/CBO9780511804175.011 Published online by Cambridge University Press

```
class AddExp : public Expression
{
    virtual void accept(Visitor *vis)
    { lhs->accept(vis); vis->visitAddExp(this); rhs->accept(vis); }
    //...
}
```

The associated Visitor abstract class, naming the methods that must be included in each visitor and some example subclasses, have this form:

```
class Visitor
{
    virtual void visitIntExp(IntegerExp *exp) = 0;
    virtual void visitAddExp(AddExp *exp) = 0;    // Abstract class
}
class ParenPrintVisitor : public Visitor
{
    virtual void visitIntExp(IntegerExp *exp) {// IntExp print code};
    virtual void visitAddExp(AddExp *exp) {// AddExp print code};
}
class EvaluateVisitor : public Visitor
{
    virtual void visitIntExp(IntegerExp *exp) {// IntExp eval code};
    virtual void visitAddExp(IntegerExp *exp) {// AddExp eval code};
}
```

Suppose there are *n* subclasses of Expression and *m* subclasses of Visitor. How many classes would have to be added or changed to add each of the following things by use of the visitor design pattern?

(c) A new class to represent product expressions.

(d) A new operation to graphically draw the expression parse tree.

The designers want your advice.

(e) Under what circumstances would you recommend using the standard design?

(f) Under what circumstances would you recommend using the visitor design pattern?

https://doi.org/10.1017/CBO9780511804175.011 Published online by Cambridge University Press