7

The static structure: classes

Examining the software engineering background of our discussion, you have seen the reasons for demanding a better approach to modular design: reusability and extendibility. You have realized the limitations of traditional approaches: centralized architectures limiting flexibility. You have discovered the theory behind the object-oriented approach: abstract data types. You have heard enough about the problems. On to the solution!

This chapter and the others in part C introduce the fundamental techniques of object-oriented analysis, design and programming. As we go along, we will develop the necessary notation.

Our first task is to examine the basic building blocks: classes.

7.1 OBJECTS ARE NOT THE SUBJECT

What is the central concept of object technology?

Think twice before you answer "object". Objects are useful, but they are not new. Ever since Cobol has had structures; ever since Pascal has had records; ever since the first C programmer wrote the first C structure definition, humanity has had objects.

Objects are studied in detail in the next chapter.

Objects remain important to describe the execution of an O-O system. But the basic notion, from which everything in object technology derives, is **class**, previewed in the preceding chapter. Here again is the definition:

Definition: class

A class is an abstract data type equipped with a possibly partial implementation.

Abstract data types are a mathematical notion, suitable for the specification stage (also called analysis). Because it introduces implementations, partial or total, the notion of class establishes the necessary link with software construction — design and implementation. Remember that a class is said to be effective if the implementation is total, deferred otherwise.

Like an ADT, a class is a type: it describes a set of possible data structures, called the ***instances*** of the class. Abstract data types too have instances; the difference is that an instance of an ADT is a purely mathematical element (a member of some mathematical set), whereas an instance of a class is a data structure that may be represented in the memory of a computer and manipulated by a software system.

For example if we have defined a class *STACK* by taking the ADT specification of the previous chapter and adding adequate representation information, the instances of that class will be data structures representing individual stacks. Another example, developed in the rest of this chapter, is a class *POINT* modeling the notion of point in a two-dimensional space, under some appropriate representation; an instance of that class is a data structure representing a point. Under one of the representations studied below, the cartesian representation, each instance of *POINT* is a record with two fields representing the horizontal and vertical coordinates, *x* and *y*, of a point.

The definition of "class" yields as a byproduct a definition of "object". An object is simply an instance of some class. For example an instance of class *STACK* — a data structure representing a particular stack — is an object; so is an instance of class *POINT*, representing a particular point in two-dimensional space.

The software texts that serve to produce systems are classes. Objects are a run-time notion only: they are created and manipulated by the software during its execution.

The present chapter is devoted to the basic mechanisms for writing software elements and combining them into systems; as a consequence, its focus is on classes. In the next chapter, we will explore the run-time structures generated by an object-oriented system; this will require us to study some implementation issues and to take a closer look at the nature of objects.

7.2 AVOIDING THE STANDARD CONFUSION

A class is a model, and an object is an instance of such a model. This property is so obvious that it would normally deserve no comments beyond the preceding definitions; but is has been the victim of so much confusion in the more careless segment of the literature that we must take some time to clarify the obvious. (If you feel that you are immune to such a danger, and have avoided exposure to sloppy object-oriented teaching, you may wish to skip this section altogether as it essentially belabors the obvious.)

The next section, for readers who do not like the belaboring of the obvious, is "THE ROLE OF CLASSES", 7.3, page 169.

What would you think of this?

> *Among the countries in Europe we may identify the Italian. The Italian has a mountain chain running through him North-South and he likes good cooking, often using olive oil. His climate is of the Mediterranean type, and he speaks a beautifully musical language.*

See e.g. Oliver Sacks, ``The Man Who Mistook His Wife for a Hat and Other Clinical Tales'', Harper Perennials, 1991.

If someone in a sober state talked or wrote to you in this fashion, you might suspect a new neurological disease, the inability to distinguish between categories (such as the Italian nation) and individuals members of these categories (such as individual Italians), reason enough to give to the ambulance driver the address of Dr. Sacks's New York clinic.

Yet in the object-oriented software literature similar confusions are common. Consider the following extract from a popular book on O-O analysis, which uses the example of an interactive system to discuss how to identify abstractions:

[Coad 1990], *3.3.3, page 67.*

> *[W]e might identify a "User" Object in a problem space where the system does not need to keep any information about the user. In this case, the system does not need the usual identification number, name, access privilege, and the like. However, the system does need to monitor the user, responding to requests and providing timely information. And so, because of required Services on behalf of the real world thing (in this case, User), we need to add a corresponding Object to the model of the problem space.*

In the same breath this text uses the word *objects*, *user* and *thing* in two meanings belonging to entirely different levels of abstraction:

Exercise E7.1, page 216, asks you to clarify each use of "Object" in this text.

- A typical user of the interactive system under discussion.
- The *concept* of user in general.

Although this is probably a slip of terminology (a peccadillo which few people can claim never to have committed) rather than a true confusion on the authors' part, it is unfortunately representative of how some of the literature deals with the model-instance distinction. If you start the study of a new method with this kind of elementary mix-up, real or apparent, you are not likely to obtain a rational approach to software construction.

The mold and the instance

Take this book — the copy which you are currently reading. Consider it as an object in the common sense of the term. It has its own individual features: the copy may be brand new, or already thumbed by previous readers; perhaps you wrote your name on the first page; or it belongs to a library and has a local identification code impressed on its spine.

The basic properties of the book, however, such as its title, publisher, author and contents, are determined by a general description which applies to every individual copy: the book is entitled *Object-Oriented Software Construction*, it is published by Prentice Hall, it talks about the object-oriented method, and so on. This set of properties defines not an object but a class of objects (also called, in this case, the **type** of these objects; for the time being the notions of type and class may be considered synonymous).

Call the class *OOSC*. It defines a certain mold. Objects built from this mold, such as your copy of the book, are called ***instances*** of the class. Another example of mold would be the plaster cast that a sculptor makes to obtain an inverted version of the design for a set of identical statues; any statue derived from the cast is an instance of the mold.

In the quotation from *The Name of the Rose* which opens part C, the Master is explaining how he was able to determine, from traces of the snow, that Brownie, the Abbot's horse, earlier walked here. Brownie is an instance of the class of all horses. The sign on the snow, although imprinted by one particular instance, includes only enough information to determine the class (horse), not its identity (Brownie). Since the class, like the sign, identifies all horses rather than a particular horse, the extract calls it a sign too.

Page 163.

Exactly the same concepts apply to software objects. What you will write in your software systems is the description of classes, such as a class *LINKED_STACK* describing properties of stacks in a certain representation. Any particular execution of your system may use the classes to create objects (data structures); each such object is derived from a class, and is called an **instance** of that class. For example the execution may create a linked stack object, derived from the description given in class *LINKED_STACK*; such an object is an instance of class *LINKED_STACK*.

The class is a software text. It is static; in other words, it exists independently of any execution. In contrast, an object derived from that class is a dynamically created data structure, existing only in the memory of a computer during the execution of a system.

This, of course, is in line with the earlier discussion of abstract data types: when specifying *STACK* as an ADT, we did not describe any particular stack, but the general notion of stack, a mold from which one can derive individual instances ad libitum.

The statements "*x* is an instance of *T*" and "*x* is an object of type *T*" will be considered synonymous for this discussion.

With the introduction of inheritance we will need to distinguish between the *direct instances* of a class (built from the exact pattern defined by the class) and its *instances* in the more general sense (direct instances of the class or any of its specializations).

See "Instances", page 475.

Metaclasses

Why would so many books and articles confuse two so clearly different notions as class and object? One reason — although not an excuse — is the appeal of the word "object", a simple term from everyday language. But it is misleading. As we already saw in the discussion of seamlessness, although some of the objects (class instances) which O-O systems manipulate are the computer representations of objects in the usual sense of the term, such as documents, bank accounts or airplanes, many others have no existence outside of the software; they include in particular the objects introduced for design and implementation purposes — instances of classes such as *STATE* or *LINKED_LIST*.

Another possible source of confusion between objects and classes is that in some cases we may need to treat classes themselves as objects. This need arises only in special contexts, and is mainly relevant to developers of object-oriented development environments. For example a compiler or interpreter for an O-O language will manipulate data structures representing classes written in that language. The same would hold of other tools such as a browser (a tool used to locate classes and find out about their properties) or a configuration management system. If you produce such tools, you will create objects that represent classes.

Pursuing an analogy used earlier, we may compare this situation to that of a Prentice Hall employee who is in charge of preparing the catalog of software engineering titles. For the catalog writer, OOSC, the concept behind this book, is an object — an instance of a class "catalog entry". In contrast, for the reader of the book, that concept is a class, of which the reader's particular copy is an instance.

Some object-oriented languages, notably Smalltalk, have introduced a notion of **metaclass** to handle this kind of situation. A metaclass is a class whose instances are themselves classes — what the *Name of the Rose* extract called "signs of signs".

We will avoid metaclasses in this presentation, however, since they bring more problems than benefits. In particular, the addition of metaclasses makes it difficult to have static type checking, a required condition of the production of reliable software. The main applications of metaclasses are better obtained through other mechanisms anyway:

"Universal classes", page 580.

- You can use metaclasses to make a set of features available to many or all classes. We will achieve the same result by arranging the inheritance structure so that all classes are descendants of a general-purpose, customizable class *ANY*, containing the declarations of universal features.

See "The creation instruction", page 232.

- A few operations may be viewed as characterizing a class rather than its instances, justifying their inclusion as features of a metaclass. But these operations are few and known; the most obvious one is object creation — sufficiently important to deserve a special language construct, the creation instruction. (Other such operations, such as object duplication, will be covered by features of class *ANY*.)

- There remains the use of metaclasses to obtain information about a class, such as a browser may need: name of the class, list of features, list of parents, list of suppliers etc. But we do not need metaclasses for that. It will suffice to devise a library class, *E_CLASS*, so that each instance of *E_CLASS* represents a class and its properties. When we create such an instance, we pass to the creation instruction an argument representing a certain class *C*; then by applying the various features of *E_CLASS* to that instance, we can learn all about *C*.

In practice, then, we can do without a separate concept of metaclass. But even in a method, language or environment that would support this notion, the presence of metaclasses is no excuse for confusing molds and their instances — classes and objects.

7.3 THE ROLE OF CLASSES

Having taken the time to remove an absurd but common and damaging confusion, we may now come back to the central properties of classes, and in particular study why they are so important to object technology.

To understand the object-oriented approach, it is essential to realize that classes play two roles which pre-O-O approaches had always treated as separate: module and type.

Modules and types

Programming languages and other notations used in software development (design languages, specification languages, graphical notations for analysis) always include both some module facility and some type system.

A module is a unit of software decomposition. Various forms of module, such as routines and packages, were studied in an earlier chapter. Regardless of the exact choice of module structure, we may call the notion of module a **syntactic** concept, since the decomposition into modules only affects the form of software texts, not what the software can do; it is indeed possible in principle to write any Ada program as a single package, or any Pascal program as a single main program. Such an approach is not recommended, of course, and any competent software developer will use the module facilities of the language at hand to decompose his software into manageable pieces. But if we take an existing program, for example in Pascal, we can always merge all the modules into a single one, and still get a working system with equivalent semantics. (The presence of recursive routines makes the conversion process less trivial, but does not fundamentally affect this discussion.) So the practice of decomposing into modules is dictated by sound engineering and project management principles rather than intrinsic necessity.

See chapter 3.

Types, at first sight, are a quite different concept. A type is the static description of certain dynamic objects: the various data elements that will be processed during the execution of a software system. The set of types usually includes predefined types such as *INTEGER* and *CHARACTER* as well as developer-defined types: record types (also known as structure types), pointer types, set types (as in Pascal), array types and others. The notion of type is a **semantic** concept, since every type directly influences the execution of a software system by defining the form of the objects that the system will create and manipulate at run time.

The class as module and type

In non-O-O approaches, the module and type concepts remain distinct. The most remarkable property of the notion of class is that it subsumes these two concepts, merging them into a single linguistic construct. A class is a module, or unit of software decomposition; but it is also a type (or, in cases involving genericity, a type pattern).

Much of the power of the object-oriented method derives from this identification. Inheritance, in particular, can only be understood fully if we look at it as providing both module extension and type specialization.

What is not clear yet is *how* it is possible in practice to unify two concepts which appear at first so distant. The discussion and examples in the rest of this chapter will answer this question.

7.4 A UNIFORM TYPE SYSTEM

An important aspect of the O-O approach as we will develop it is the simplicity and uniformity of the type system, deriving from a fundamental property:

Object rule

Every object is an instance of some class.

The Object rule will apply not just to composite, developer-defined objects (such as data structures with several fields) but also to basic objects such as integers, real numbers, boolean values and characters, which will all be considered to be instances of predefined library classes (*INTEGER*, *REAL*, *DOUBLE*, *BOOLEAN*, *CHARACTER*).

This zeal to make every possible value, however simple, an instance of some class may at first appear exaggerated or even extravagant. After all, mathematicians and engineers have used integers and reals successfully for a long time, without knowing they were manipulating class instances. But insisting on uniformity pays off for several reasons:

- It is always desirable to have a simple and uniform framework rather than many special cases. Here the type system will be entirely based on the notion of class.

The mathematical axioms defining integers are known as Peano's axioms.

- Describing basic types as ADTs and hence as classes is simple and natural. It is not hard, for example, to see how to define the class *INTEGER* with features covering arithmetic operations such as "+", comparison operations such as "<=", and the associated properties, derived from the corresponding mathematical axioms.

- By defining the basic types as classes, we allow them to take part in all the O-O games, especially inheritance and genericity. If we did not treat the basic types as classes, we would have to introduce severe limitations and many special cases.

  As an example of inheritance, classes *INTEGER*, *REAL* and *DOUBLE* will be heirs to more general classes: *NUMERIC*, introducing the basic arithmetic operations such as "+", "–" and "*", and *COMPARABLE*, introducing comparison operations such as "<". As an example of genericity, we can define a generic class *MATRIX* whose generic parameter represents the type of matrix elements, so that instances of *MATRIX* [*INTEGER*] represent matrices of integers, instances of *MATRIX* [*REAL*] represent matrices of reals and so on. As an example of combining genericity with inheritance, the preceding definitions also allow us to use the type *MATRIX* [*NUMERIC*], whose instances represent matrices containing objects of type *INTEGER* as well as objects of type *REAL* and objects of any new type *T* defined by a software developer so as to inherit from *NUMERIC*.

With a good implementation, we do not need to fear any negative consequence from the decision to define all types from classes. Nothing prevents a compiler from having special knowledge about the basic classes; the code it generates for operations on values of types such as *INTEGER* and *BOOLEAN* can then be just as efficient as if these were built-in types in the language.

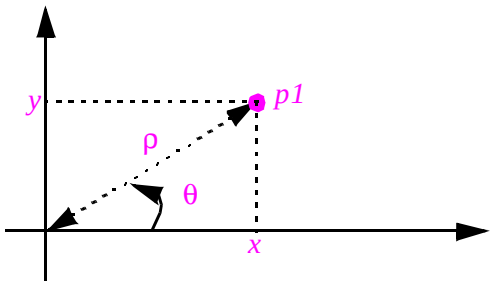
Reaching the goal of a fully consistent and uniform type system requires the combination of several important O-O techniques, to be seen only later: expanded classes, to ensure proper representation of simple values; infix and prefix operators, to enable usual arithmetic syntax (such as *a* < *b* or *–a* rather than the more cumbersome *a*.*less_than* (*b*) or *a*.*negated*); constrained genericity, needed to define classes which may be adapted to various types with specific operations, for example a class *MATRIX* that can represent matrices of integers as well as matrices of elements of other numeric types.

7.5 A SIMPLE CLASS

Let us now see what classes look like by studying a simple but typical example, which shows some of the fundamental properties applicable to almost all classes.

The features

The example is the notion of point, as it could appear in a two-dimensional graphics system.



A point and its coordinates

To characterize type *POINT* as an abstract data type, we would need the four query functions *x*, *y*, ρ, θ. (The names of the last two will be spelled out as *rho* and *theta* in software texts.) Function *x* gives the abscissa of a point (horizontal coordinate), *y* its ordinate (vertical coordinate), ρ its distance to the origin, θ the angle to the horizontal axis. The values of *x* and *y* for a point are called its cartesian coordinates, those of ρ and θ its polar coordinates. Another useful query function is *distance*, which will yield the distance between two points.

Then the ADT specification would list commands such as *translate* (to move a point by a given horizontal and vertical displacement), *rotate* (to rotate the point by a certain angle, around the origin) and *scale* (to bring the point closer to or further from the origin by a certain factor).

The name translate refers to the "translation" operation of geometry.

It is not difficult to write the full ADT specification including these functions and some of the associated axioms. For example, two of the function signatures will be

$$x: POINT \rightarrow REAL$$
$$translate: POINT \times REAL \times REAL \rightarrow POINT$$

and one of the axioms will be (for any point *p* and any reals *a*, *b*):

$$x\ (translate\ (p1, a, b)) = x\ (p1) + a$$

expressing that translating a point by <*a*, *b*> increases its abscissa by *a*.

Exercise E7.2, page 216.

You may wish to complete this ADT specification by yourself. The rest of this discussion will assume that you have understood the ADT, whether or not you have written it formally in full, so that we can focus on its implementation — the class.

Attributes and routines

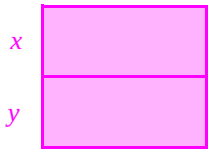
Any abstract data type such as *POINT* is characterized by a set of functions, describing the operations applicable to instances of the ADT. In classes (ADT implementations), functions will yield features — the operations applicable to instances of the class.

"Function categories", page 134.

We have seen that ADT functions are of three kinds: queries, commands and creators. For features, we need a complementary classification, based on how each feature is implemented: by space or by time.

The example of point coordinates shows the difference clearly. Two common representations are available for points: cartesian and polar. If we choose cartesian representation, each instance of the class will contain two fields representing the x and y of the corresponding point:

Representing a point in cartesian coordinates

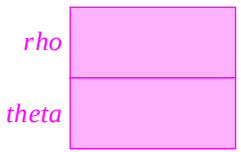


(*CARTESIAN_POINT*)

If *p1* is the point shown, getting its x or its y simply requires looking up the corresponding field in this structure. Getting ρ or θ, however, requires a computation: for ρ we must compute $\sqrt{x^2 + y^2}$, and for θ we must compute $arctg\ (y/x)$ with non-zero x.

If we use polar representation, the situation is reversed: ρ and θ are now accessible by simple field lookup, x and y require small computations (of $\rho\ cos\ \theta$ and $\rho\ sin\ \theta$).

Representing a point in polar coordinates



(*POLAR_POINT*)

This example shows the need for two kinds of feature:

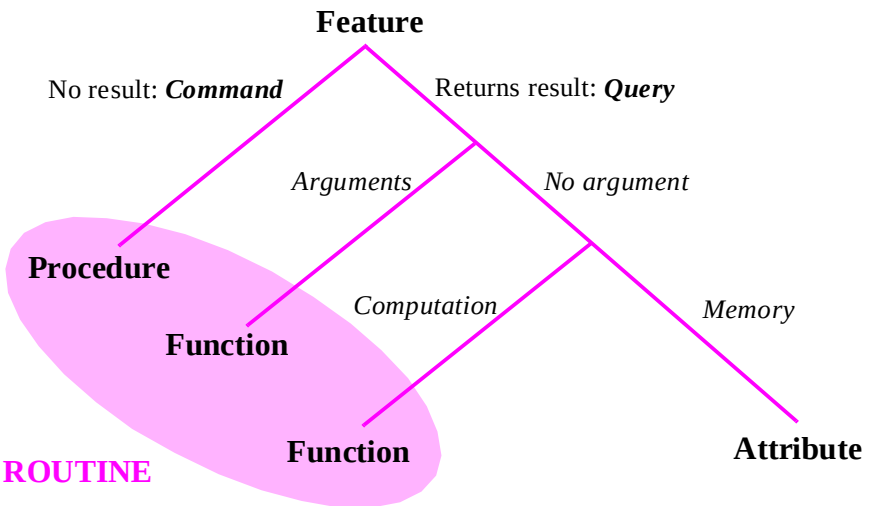
- Some features will be represented by space, that is to say by associating a certain piece of information with every instance of the class. They will be called **attributes**. For points, x and y are attributes in cartesian representation; *rho* and *theta* are attributes in polar representation.

- Some features will be represented by time, that is to say by defining a certain computation (an algorithm) applicable to all instances of the class. They will be called **routines**. For points, *rho* and *theta* are routines in cartesian representation; *x* and *y* are routines in polar representation.

A further distinction affects routines (the second of these categories). Some routines will return a result; they are called **functions**. Here *x* and *y* in polar representation, as well as *rho* and *theta* in cartesian representation, are functions since they return a result, of type *REAL*. Routines which do not return a result correspond to the commands of an ADT specification and are called **procedures**. For example the class *POINT* will include procedures *translate*, *rotate* and *scale*.

Be sure not to confuse the use of "function" to denote result-returning routines in classes with the earlier use of this word to denote the mathematical specifications of operations in abstract data types. This conflict is unfortunate, but follows from well-established usage of the word in both the mathematics and software fields.

The following tree helps visualize this classification of features:

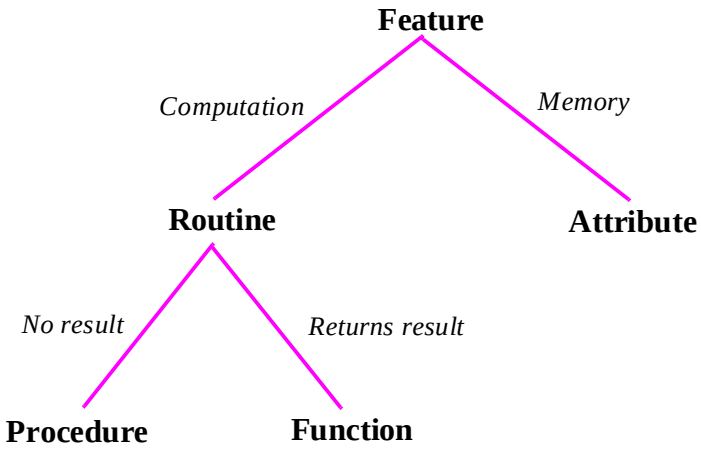


Feature classification, by role

This is an external classification, in which the principal question is how a feature will look to its clients (its users).

We can also take a more internal view, using as primary criterion how each feature is implemented in the class, and leading to a different classification:

Feature classification, by implementation



Uniform access

One aspect of the preceding classifications may at first appear disturbing and has perhaps caught your attention. In many cases, we should be able to manipulate objects, for example a point *p1*, without having to worry about whether the internal representation of *p1* is cartesian, polar or other. Is it appropriate, then, to distinguish explicitly between attributes and functions?

The answer depends on whose view we consider: the supplier's view (as seen by the author of the class itself, here *POINT*) or the client's view (as seen by the author of a class that uses *POINT*). For the supplier, the distinction between attributes and functions is meaningful and necessary, since in some cases you will want to implement a feature by storage and in others by computation, and the decision must be reflected somewhere. What would be wrong, however, would be to force the **clients** to be aware of the difference. If I am accessing *p1*, I want to be able to find out its *x* or its ρ without having to know how such queries are implemented.

See "Uniform Access", page 55.

The Uniform Access principle, introduced in the discussion of modularity, answers this concern. The principle states that a client should be able to access a property of an object using a single notation, whether the property is implemented by memory or by computation (space or time, attribute or routine). We shall follow this important principle in devising a notation for feature call below: the expression denoting the value of the *x* feature for *p1* will always be

p1.x

whether its effect is to access a field of an object or to execute a routine.

> As you will have noted, the uncertainty can only exist for queries without arguments, which may be implemented as functions or as attributes. A command must be a procedure; a query with arguments must be a function, since attributes cannot have arguments.

The Uniform Access principle is essential to guarantee the autonomy of the components of a system. It preserves the class designer's freedom to experiment with various implementation techniques without disturbing the clients.

Pascal, C and Ada violate the principle by providing a different notation for a function call and for an attribute access. For such non-object-oriented languages this is understandable (although we have seen that Algol W, a 1966 predecessor to Pascal, satisfied uniform access). More recent languages such as C++ and Java also do not enforce the principle. Departing from Uniform Access may cause any internal representation change (such as the switch from polar to cartesian or some other representation) to cause upheaval in many client classes. This is a primary source of instability in software development.

The Uniform Access principle also yields a requirement on documentation techniques. If we are to apply the principle consistently, we must ensure that it is not possible to determine, from the official documentation on a class, whether a query without arguments is a function or an attribute. This will be one of the properties of the standard mechanism for documenting a class, known as the short form.

"Using assertions for documentation: the short form of a class", page 389.

The class

Here is a version of the class text for *POINT*. (Any occurrence of consecutive dashes -- introduces a comment, which extends to the end of the line; comments are explanations intended for the reader of the class text and do not affect the semantics of the class.)

```
indexing
        description: "Two-dimensional points"
class POINT feature
        x, y: REAL
                -- Abscissa and ordinate

        rho: REAL is
                -- Distance to origin (0, 0)
            do
                    Result := sqrt (x ^ 2 + y ^ 2)
            end

        theta: REAL is
                -- Angle to horizontal axis
            do
                …Left to reader (exercise E7.3, page 216) °
            end
```

```
        distance (p: POINT): REAL is
                -- Distance to p
            do
                Result := sqrt ((x – p.x) ^ 2 + (y – p.y) ^ 2)
            end

        translate (a, b: REAL) is
                -- Move by a horizontally, b vertically.
            do
                x := x + a
                y := y + b
            end

        scale (factor: REAL) is
                -- Scale by factor.
            do
                x := factor * x
                y := factor * y
            end

        rotate (p: POINT; angle: REAL) is
                -- Rotate around p by angle.
            do
                …Left to reader (exercise E7.3, page 216) …
            end
end
```

The next few sections explain in detail the non-obvious aspects of this class text.

The class mainly consists of a clause listing the various features and introduced by the keyword **feature**. There is also an **indexing** clause giving general *description* information, useful to readers of the class but with no effect on its execution semantics. Later on we will learn three optional clauses: **inherit** for inheritance, **creation** for non-default creation and **invariant** for introducing class invariants; we will also see how to include two or more **feature** clauses in one class.

7.6 BASIC CONVENTIONS

Class *POINT* shows a number of techniques which will be used throughout later examples. Let us first look at the basic conventions.

Recognizing feature kinds

Features *x* and *y* are just declared as being of type *REAL*, with no associated algorithm; so they can only be attributes. All other features have a clause of the form

```
    is
        do
            … Instructions…
        end
```

which defines an algorithm; this indicates the feature is a routine. Routines *rho*, *theta* and *distance* are declared as returning a result, of type *REAL* in all cases, as indicated by declarations of the form

```
rho: REAL is …
```

This defines them as functions. The other two, *translate* and *scale*, do not return a result (since they do not have a result declaration of the form *:T* for some type *T*), and so they are procedures.

Since *x* and *y* are attributes, while *rho* and *theta* are functions, the representation chosen in this particular class for points is cartesian.

Routine bodies and header comments

The body of a routine (the **do** clause) is a sequence of instructions. You can use semicolons, in the Algol-Pascal tradition, to separate successive instructions and declarations, but the semicolons are optional. We will omit them for simplicity between elements on separate lines, but will always include them to delimit instructions or declarations appearing on the same line.

For details see "The War of the Semicolons", page 897.

All the instructions in the routines of class *POINT* are assignments; for assignment, the notation uses the := symbol (again borrowed from the Algol-Pascal conventions). This symbol should of course not be confused with the equality symbol =, used, as in mathematics, as a comparison operator.

Another convention of the notation is the use of header comments. As already noted, comments are introduced by two consecutive dashes --. They may appear at any place in a class text where the class author feels that readers will benefit from an explanation. A special role is played by the **header comment** which, as a general style rule, should appear at the beginning of every routine, after the keyword **is**, indented as shown by the examples in class *POINT*. Such a header comment should tersely express the purpose of the routine.

Attributes should also have a header comment immediately following their declaration, aligned with routine's header comments, as illustrated here with *x* and *y*.

The indexing clause

At the beginning of the class comes a clause starting with the keyword **indexing**. It contains a single entry, labeled *description*. The indexing clause has no effect on software execution, but serves to associate information with the class. In its general form it contains zero or more entries of the form

See "A note about component indexing", page 78.

```
index_word: index_value, index_value, …
```

where the *index_word* is an arbitrary identifier, and each *index_value* is an arbitrary language element (identifier, integer, string…).

The benefit is twofold:

- Readers of the class get a summary of its properties, without having to see the details.
- In a software development environment supporting reuse, query tools (often known as *browsers*) can use the indexing information to help potential users find out about available classes; the tools can let the users enter various search words and match them with the index words and values.

Chapter 36 describes a general O-O browsing mechanism.

The example has a single indexing entry, with *description* as index word and, as index value, a string describing the purpose of the class. All classes in this book, save for short examples, will include a *description* entry. You are strongly encouraged to follow this example and begin every class text with an **indexing** clause providing a concise overview of the class, in the same way that every routine begins with a header comment.

"Self-Documentation", page 54.

Both indexing clauses and header comments are faithful applications of the Self-Documentation principle: as much as possible of a module's documentation should appear in the text of the module itself.

Denoting a function's result

We need another convention to understand the texts of the functions in class *POINT*: *rho*, *theta* and *distance*.

An "entity" is a name denoting a value. Full definition on page 213.

Any language that supports functions (value-returning routines) must offer a notation allowing the body of a function to set the value which will be returned by any particular call. The convention used here is simple: it relies on a predefined entity name, *Result*, denoting the value that the call will return. For example, the body of *rho* contains an assignment to *Result*:

```
Result := sqrt (x ^ 2 + y ^ 2)
```

Result is a reserved word, and may only appear in functions. In a function declared as having a result of type *T*, *Result* is treated in the same way as other entities, and may be assigned values through assignment instructions such as the above.

Initialization rules will be given in "The creation instruction", page 232.

Any call to the function will return, as its result, the final value assigned to *Result* during the call's execution. That value always exists since language rules (to be seen in detail later) require every execution of the routine, when it starts, to initialize *Result* to a preset value. For a *REAL* the initialization value is zero; so a function of the form

```
non_negative_value (x: REAL): REAL is
        -- The value of x if positive; zero otherwise.
    do
        if x > 0.0 then
            Result := x
        end
    end
```

will always return a well-defined value (as described by the header comment) even though the conditional instruction has no **else** part.

The discussion section of this chapter examines the rationale behind the *Result* convention and compares it with other techniques such as return instructions. Although this convention addresses an issue that arises in all design and programming languages, it blends particularly well with the rest of the object-oriented approach.

See "Denoting the result of a function", page 210.

Style rules

The class texts in this book follow precise style conventions regarding indentation, fonts (for typeset output), choice of names for features and classes, use of lower and upper case.

The discussion will point out these conventions, under the heading "style rules", as we go along. They should not be dismissed as mere cosmetics: quality software requires consistency and attention to all details, of form as well as of content. The reusability goal makes these observations even more important, since it implies that software texts will have a long life, during which many people will need to understand and improve them.

You should apply the style rules right from the time you start writing a class. For example you should never write a routine without immediately including its header comment. This does not take long, and is not wasted time; in fact it is time saved for all future work on the class, whether by you or by others, whether after half an hour or after half a decade. Using regular indentation, proper spelling for comments and identifiers, adequate lexical conventions — a space before each opening parenthesis but not after, and so on — does not make your task any longer than ignoring these rules, but compounded over months of work and heaps of software produces a tremendous difference. Attention to such details, although not sufficient, is a necessary condition for quality software (and quality, the general theme of this book, is what defines software engineering).

The elementary style rules are clear from the preceding class example. Since our immediate goal is to explore the basic mechanisms of object technology, their precise description will only appear in a later chapter.

Chapter 26 is devoted to style rules.

Inheriting general-purpose facilities

Another aspect of class *POINT* which requires clarification is the presence of calls to the *sqrt* function (in *rho* and *distance*). This function should clearly return the square root of a real number, but where does it come from?

Since it does not seem appropriate to encumber a general-purpose language with specialized arithmetic operations, the best technique is to define such operations as features of some specialized class — say *ARITHMETIC* — and then simply require any class that needs these facilities to inherit from the specialized class. As will be seen in detail in a later chapter, it suffices then to write *POINT* as

```
class POINT inherit
        ARITHMETIC
feature
        … The rest as before …
end
```

See "FACILITY INHERITANCE", 24.9, page 847.

This technique of inheriting general-purpose facilities is somewhat controversial; one can argue that O-O principles suggest making a function such as *sqrt* a feature of the class representing the object to which it applies, for example *REAL*. But there are many operations on real numbers, not all of which can be included in the class. Square root may be sufficiently fundamental to justify making it a feature of class *REAL*; then we would write *a*.*sqrt* rather than *sqrt* (*x*). We will return, in the discussion of design principles, to the question of whether "facilities" classes such as *ARITHMETIC* are desirable.

7.7 THE OBJECT-ORIENTED STYLE OF COMPUTATION

Let us now move to the fundamental properties of class *POINT* by trying to understand a typical routine body and its instructions, then studying how the class and its features may be used by other classes — clients.

The current instance

Here again is the text of one of our example routines, procedure *translate*:

```
translate (a, b: REAL) is
        -- Move by a horizontally, b vertically
    do
        x := x + a
        y := y + b
    end
```

At first sight this text appears clear enough: to translate a point by *a* horizontally, *b* vertically, we add *a* to its *x* and *b* to its *y*. But if you look at it more carefully, it may not be so obvious anymore! Nowhere in the text have we stated what point we were talking about. To whose *x* and whose *y* are we adding *a* and *b*? In the answer to this question will lie one of the most distinctive aspects of the object-oriented development style. Before we are ready to discover that answer we must understand a few intermediate topics.

A class text describes the properties and behavior of objects of a certain type, points in this example. It does so by describing the properties and behavior of a typical instance of that type — what we could call the "point in the street" in the way newspapers report the opinion of the "man or woman in the street". We will settle for a more formal name: the **current instance** of the class.

Once in a while, we may need to refer to the current instance explicitly. The reserved word

```
Current
```

will serve that purpose. In a class text, *Current* denotes the current instance of the enclosing class. As an example of when *Current* is needed, assume we rewrite *distance* so that it checks first whether the argument *p* is the same point as the current instance, in which case the result is 0 with no need for further computation. Then *distance* will appear as

```
distance (p: POINT): REAL is
        -- Distance to p
do
        if p /= Current then
                Result := sqrt ((x — p.x) ^ 2 + (y — p.y) ^ 2)
        end
end
```

(/= is the inequality operator. Because of the initialization rule mentioned above, the conditional instruction does not need an **else** part: if *p* = *Current* the result is zero.)

In most circumstances, however, the current instance is implicit and we will not need to refer to *Current* by its name. For example, references to *x* in the body of *translate* and the other routines simply mean, if not further qualified: "the *x* of the current instance".

This only pushes back the mystery, of course: "who" really is *Current*? The answer will come with the study of routine calls below. As long as we only look at the routine text, it will suffice to know that all operations are relative, by default, to an implicitly defined object, the current instance.

Clients and suppliers

Ignoring for a few moments the enigma of *Current*'s identity, we know how to define simple classes. We must now study how to use their definitions. Such uses will be in other classes — since in a pure object-oriented approach every software element is part of some class text.

There are only two ways to use a class such as *POINT*. One is to inherit from it; this is studied in detail in later chapters. The other one is to become a **client** of *POINT*.

Chapters 14 to 16 study inheritance.

The simplest and most common way to become a client of a class is to declare an entity of the corresponding type:

Definition: client, supplier

Let *S* be a class. A class *C* which contains a declaration of the form *a*: *S* is said to be a client of *S*. *S* is then said to be a supplier of *C*.

In this definition, *a* may be an attribute or function of *C*, or a local entity or argument of a routine of *C*.

For example, the declarations of *x*, *y*, *rho*, *theta* and *distance* above make class *POINT* a client of *REAL*. Other classes may in turn become clients of *POINT*. Here is an example:

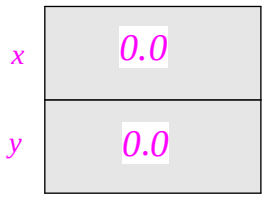
```
class GRAPHICS feature
        p1: POINT

        …

        some_routine is
                -- Perform some actions with p1.
            do
                … Create an instance of POINT and attach it to p1 …
                p1.translate (4.0, –1.5)                    --**
                …
            end
        …
end
```

Before the instruction marked --** gets executed, the attribute *p1* will have a value denoting a certain instance of class *POINT*. Assume that this instance represents the origin, of coordinates $x = 0$, $y = 0$:

The origin



Entity *p1* is said to be **attached** to this object. We do not worry at this point about how the object has been created (by the unexplained line that reads "…Create object…") and initialized; such topics will be discussed as part of the object model in the next chapter. Let us just assume that the object exists and that *p1* is attached to it.

Feature call

The starred instruction,

```
p1.translate (4.0, –1.5)
```

deserves careful examination since it is our first complete example of what may be called the **basic mechanism of object-oriented computation**: feature call. In the execution of an object-oriented software system, all computation is achieved by calling certain features on certain objects.

This particular feature call means: apply to *p1* the feature *translate* of class *POINT*, with arguments *4.0* and *–1.5*, corresponding to *a* and *b* in the declaration of *translate* as it appears in the class. More generally, a feature call appears in its basic form as one of

```
x.f
x.f (u, v, …)
```

In such a call, *x*, called the **target** of the call, is an entity or expression (which at run time will be attached to a certain object). As any other entity or expression, *x* has a certain type, given by a class *C*; then *f* must be one of the features of *C*. More precisely, in the first form, *f* must be an attribute or a routine without arguments; in the second form, *f* must be a routine with arguments, and *u*, *v*, …, called the **actual arguments** for the call, must be expressions matching in type and number the formal arguments declared for *f* in *C*.

In addition, *f* must be available (exported) to the client containing this call. This is the default; a later section will show how to restrict export rights. For the moment, all features are available to all clients.

"SELECTIVE EXPORTS AND INFORMATION HIDING", 7.8, page 191.

The effect of the above call when executed at run time is defined as follows:

Effect of calling a feature *f* on a target *x*

Apply feature *f* to the object attached to *x*, after having initialized each formal argument of *f* (if any) to the value of the corresponding actual argument.

The Single Target principle

What is so special about feature call? After all, every software developer knows how to write a procedure *translate* which moves a point by a certain displacement, and is called in the traditional form (available, with minor variants, in all programming languages):

translate (*p1*, *4.0*, *–1.5*)

Unlike the object-oriented style of feature call, however, this call treats all arguments on an equal basis. The O-O form has no such symmetry: we choose a certain object (here the point *p1*) as target, relegating the other arguments, here the real numbers *4.0* and *–1.5*, to the role of supporting cast. This way of making every call relative to a single target object is a central part of the object-oriented style of computing:

Single Target principle

Every operation of object-oriented computation is relative to a certain object, the current instance at the time of the operation's execution.

To novices, this is often the most disconcerting aspect of the method. In object-oriented software construction, we never really ask: "Apply this operation to these objects". Instead we say: "Apply this operation to **this** object here." And perhaps (in the second form): "Oh, by the way, I almost forgot, you will need those values there as arguments".

What we have seen so far does not really suffice to justify this convention; in fact its negative consequences will, for a while, overshadow its advantages. An example of counter-intuitive effect appears with the function *distance* of class *POINT*, declared above as *distance* (*p*: *POINT*): *REAL*, implying that a typical call will be written

p1.*distance* (*p2*)

which runs against the perception of *distance* as a symmetric operation on two arguments. Only with the introduction of inheritance will the Single Target principle be fully vindicated.

The module-type identification

"The class as module and type", page 170.

The Single Target principle is a direct consequence of the module-type merge, presented earlier as the starting point of object-oriented decomposition: if every module is a type, then every operation in the module is relative to a certain instance of that type (the current instance). Up to now, however, the details of that merge remained a little mysterious. A class, it was said above, is both a module and a type; but how can we reconcile the syntactic notion of module (a grouping of related facilities, forming a part of a software system) with the semantic notion of type (the static description of certain possible run-time objects)? The example of *POINT* makes the answer clear:

How the module-type merge works

The facilities provided by class *POINT*, viewed as a module, are precisely the operations available on instances of class *POINT*, viewed as a type.

This identification between the operations on instances of a type and the services provided by a module lies at the heart of the structuring discipline enforced by the object-oriented method.

The role of *Current*

With the help of the same example, we are now also in a position to clear the remaining mystery: what does the current instance really represent?

The form of calls indicates why the text of a routine (such as *translate* in *POINT*) does not need to specify "who" *Current* is: since every call to the routine will be relative to a certain target, specified explicitly in the call, the execution will treat every feature name appearing in the text of the routine (for example *x* in the text of *translate*) as applying to that particular target. So for the execution of the call

p1.*translate* (*4.0*, *–1.5*)

every occurrence of *x* in the body of *translate*, such as those in the instruction

x := *x* + *a*

means: "the *x* of *p1*".

The exact meaning of *Current* follows from these observations. *Current* means: "the target of the current call". For example, for the duration of the above call, *Current* will denote the object attached to *p1*. In a subsequent call, *Current* will denote the target of that new call. That this all makes sense follows from the extreme simplicity of the object-oriented computation model, based on feature calls and on the Single Target principle:

Feature Call principle

F1 • No software element ever gets executed except as part of a routine call.

F2 • Every call has a target.

Qualified and unqualified calls

It was said above that all object-oriented computation relies on feature calls. A consequence of this rule is that software texts actually contain more calls than meet the eye at first. The calls seen so far were of one of the two forms introduced above:

x.*f*
x.*f* (*u*, *v*, …)

Such calls use so-called dot notation (with the "." symbol) and are said to be **qualified** because the target of the call is explicitly identified: it is the entity or expression (*x* in both cases above) that appears before the dot.

Other calls, however, will be unqualified because their targets are implicit. As an example, assume that we want to add to class *POINT* a procedure *transform* that will both translate and scale a point. The procedure's text may rely on *translate* and *scale*:

```
transform (a, b, factor: REAL) is
        -- Move by a horizontally, b vertically, then scale by factor.
    do
        translate (a, b)
        scale (factor)
    end
```

The routine body contains calls to *translate* and *scale*. Unlike the earlier examples, these calls do not show an explicit target, and do not use dot notation. Such calls are said to be **unqualified**.

Unqualified calls do not violate the property called F2 in the Feature Call principle: like qualified calls, they have a target. As you have certainly guessed, the target in this case is the current instance. When procedure *transform* is called on a certain target, its body calls *translate* and *scale* on the same target. It could in fact have been written

```
do
    Current.translate (a, b)
    Current.scale (factor)
```

More generally, you may rewrite any unqualified call as a qualified call with *Current* as its target. The unqualified form is of course simpler and just as clear.

Strictly speaking, the equivalence only applies if the feature is exported.

The unqualified calls that we have just examined were calls to routines. The same discussion applies to attributes, although the presence of calls is perhaps less obvious in this case. It was noted above that, in the body of *translate*, the occurrence of *x* in the expression *x* + *a* denotes the *x* field of the current instance. Another way of expressing this

property is that *x* is actually a feature call, so that the expression as a whole could have been written as *Current*.*x* + *a*.

More generally, any instruction or expression of one of the forms

f
f (*u*, *v*, …)

is in fact an unqualified call, and you may also write it in qualified form as (respectively)

Current.*f*
Current.*f* (*u*, *v*, …)

although the unqualified forms are more convenient. If you use such a notation as an instruction, *f* must be a procedure (with no argument in the first form, and with the appropriate number and types of arguments in the second). If it is an expression, *f* may be an attribute (in the first form only, since attributes have no arguments) or a function.

Be sure to note that this syntactical equivalence only applies to a feature used as an instruction or an expression. So in the following assignment from procedure *translate*

x := *x* + *a*

only the occurrence of *x* on the right-hand side is an unqualified call: *a* is a formal argument, not a feature; and the occurrence of *x* on the left is not an expression (one cannot assign a value to an expression), so it would be meaningless to replace it by *Current*.*x*.

Operator features

A further look at the expression *x* + *a* leads to a useful notion: operator features. This notion (and the present section) may be viewed as pure "cosmetics", that is to say, covering only a syntactical facility without bringing anything really new to the object-oriented method. But such syntactical properties can be important to make developers' life easier if they are present — or miserable if they are absent. Operator features also provide a good example of how successful the object-oriented paradigm can be at integrating earlier approaches smoothly.

Here is the idea. Although you may not have guessed it, the expression *x* + *a* contains not just one call — the call to *x*, as just seen — but two. In non-O-O computation, we would consider + as an operator, applied here to two values *x* and *a*, both declared of type *REAL*. In a pure O-O model, as noted, the only computational mechanism is feature call; so you may consider the addition itself, at least in theory, to be a call to an addition feature.

The Object rule was given on page 171.

To understand this better, consider how we could define the type *REAL*. The Object rule stated earlier implied that every type is based on some class. This applies to predefined types such as *REAL* as well as developer-defined types such as *POINT*. Assume you are requested to write *REAL* as a class. It is not hard to identify the relevant features: arithmetic operations (addition, subtraction, negation…), comparison operations (less than, greater than…). So a first sketch could appear as:

```
indexing
        description: "Real numbers (not final version!)"
class REAL feature
        plus (other: REAL): REAL is
                        -- Sum of current value and other
                do
                        …
                end
        minus (other: REAL) REAL is
                        -- Difference of current value and other
                do
                        …
                end
        negated: REAL is
                        -- Current value but with opposite sign
                do
                        …
                end
        less_than (other: REAL): BOOLEAN is
                        -- Is current value strictly less than other?
                do
                        …
                end
        … Other features …
end
```

With such a form of the class, you could not write an arithmetic expression such as *x + a* any more; instead, you would use a call of the form

x.plus (a)

Similarly, you would have to write *x.negated* instead of the usual *–x*.

One might try to justify such a departure from usual mathematical notation on the grounds of consistency with the object-oriented model, and invoke the example of Lisp to suggest that it is sometimes possible to convince a subset of the software development community to renounce standard notation. But this argument contains it owns limitations: usage of Lisp has always remained marginal. It is rather dangerous to go against notations which have been in existence for centuries, and which people have been using since elementary school, especially when there is nothing wrong with these notations.

A simple syntactical device reconciles the desire for consistency (requiring here a single computational mechanism based on feature call) and the need for compatibility with traditional notations. It suffices to consider that an expression of the form

x + a

is in fact a call to the addition feature of class *REAL*; the only difference with the *plus* feature suggested above is that we must rewrite the declaration of the corresponding feature to specify that calls will use operator notation rather than dot notation.

Here is the form of a class that achieves this goal:

The next chapter will show how to declare this class as "expanded". See "The role of expanded types", page 256.

```
indexing
        description: "Real numbers"
class REAL feature
        infix "+" (other: REAL): REAL is
                        -- Sum of current value and other
                do
                        …
                end
        infix "–" (other: REAL) REAL is
                        -- Difference of current value and other
                do
                        …
                end
        prefix "–": REAL is
                        -- Current value but with opposite sign
                do
                        …
                end
        infix "<" (other: REAL): BOOLEAN is
                        -- Is current value strictly less than other?
                do
                        …
                end
        … Other features …
end
```

Two new keywords have been introduced: **infix** and **prefix**. The only syntactical extension is that from now on we may choose feature names which, instead of identifiers (such as *distance* or *plus*), are of one of the two forms

```
infix "§"
prefix "§"
```

where § stands for an operator symbol chosen from a list which includes +, –, ✳, <, <= and a few other possibilities listed below. A feature may have a name of the **infix** form only if it is a function with one argument, such as the functions called *plus*, *minus* and *less_than* in the original version of class *REAL*; it may have a name of the **prefix** form only if it is a function with no argument, or an attribute.

Infix and prefix features, collectively called **operator features**, are treated exactly like other features (called **identifier features**) with the exception of the two syntactical properties already seen:

- The name of an operator feature as it appears in the feature's declaration is of the form **infix** "§" or **prefix** "§", rather than an identifier.
- Calls to operator features are of the form *u* § *v* (in the infix case) or § *u* (in the prefix case) rather than using dot notation.

As a consequence of the second property, operator features only support qualified calls. If a routine of class *REAL* contained, in the first version given earlier, an unqualified call of the form *plus* (*y*), yielding the sum of the current number and *y*, the corresponding call will have to be written *Current* + *y* in the second version. With an identifier feature, the corresponding notation, *Current*.*plus* (*y*), is possible but we would not normally use it in practice since it is uselessly wordy. With an operator feature we do not have a choice.

Other than the two syntactical differences noted, operator features are fully equivalent to identifier features; for example they are inherited in the same way. Any class, not just the basic classes such as *REAL*, can use operator features; for example, it may be convenient in a class *VECTOR* to have a vector addition function called **infix** "+".

The following rule will apply to the operators used in operator features. An operator is a sequence of one or more printable characters, containing no space or newline, and beginning with one of

+ – ✻ / < > = \ ^ @ # | &

In addition, the following keywords, used for compatibility with usual boolean notation, are permitted as operators:

not and or xor and then or else implies

In the non-keyword case, the reason for restricting the first character is to preserve the clarity of software texts by ensuring that any use of an infix or prefix operator is immediately recognizable as such from its first character.

Basic classes (*INTEGER* etc.) use the following, known as standard operators:

- Prefix: + – **not**.
- Infix: + – ✻ / < > <= >= // \\ ^ **and or xor and then or else implies**.

  The semantics is the usual one. // is used for integer division, \\ for integer remainder, ^ as the power operation, **xor** as exclusive or. In class *BOOLEAN*, **and then** and **or else** are variants of **and** and **or**, the difference being explained in a later chapter, and **implies** is the implication operator, such that *a* **implies** *b* is the same as (**not** *a*) **or else** *b*.

See "Non-strict boolean operators", page 454.

Operators not in the "standard" list are called free operators. Here are two examples of possible operator features using free operators:

- When we later introduce an *ARRAY* class, we will use the operator feature **infix** "@" for the function that returns an array element given by its index, so that the *i*-th element of an array *a* may be written simply as *a* @ *i*.
- In class *POINT*, we could have used the name **infix** "|–|" instead of *distance*, so that the distance between *p1* and *p2* is written *p1* |–| *p2* instead of *p1*.*p2*.

The precedence of all operators is fixed; standard operators have their usual precedence, and all free operators bind tighter than standard operators.

The use of operator features is a convenient way to maintain compatibility with well-accepted expression notation while ensuring the goal of a fully uniform type system (as stated by the Object Rule) and of a single fundamental mechanism for computation. In the same way that treating *INTEGER* and other basic types as classes does not need to cause any performance problem, treating arithmetic and boolean operations as features does not

need to affect efficiency. Conceptually, *a* + *x* is a feature call; but any good compiler will know about the basic types and their features, and will be able to handle such a call so as to generate code at least as good as the code generated for *a* + *x* in C, Pascal, Ada or any other language in which + is a special hard-wired language construct.

When using operators such as +, < and others in expressions, we may forget, most of the time, that they actually stand for feature calls; the effect of these operators is the one we would expect in traditional approaches. But it is pleasant to know that, thanks to the theoretical context of their definition, they do not cause any departure from object-oriented principles, and fit in perfectly with the rest of the method.

7.8 SELECTIVE EXPORTS AND INFORMATION HIDING

See "Information Hiding", page 51.

In the examples seen so far all the features of a class were exported to all possible clients. This is of course not always acceptable; we know from earlier discussion how important information hiding is to the design of coherent and flexible architectures.

"SELECTIVE EXPORTS", 23.5, page 796.

Let us take a look at how we can indeed restrict features to no clients, or to some clients only. This section only introduces the notation; the chapter on the design of class interfaces will discuss its proper use.

Full disclosure

By default, as noted, features declared without any particular precaution are available to all clients. In a class of the form

```
class S1 feature
        f …
        g …
        …
end
```

features *f*, *g*, … are available to all clients of *S1*. This means that in a class *C*, for an entity *x* declared of type *S1*, a call

```
x.f …
```

is valid, provided the call satisfies the other validity conditions on calls to *f*, regarding the number and types of arguments if any. (For simplicity the discussion will use identifier features as examples, but it applies in exactly the same way to operator features, for which the clients will use calls in infix or prefix form rather than dot notation.)

Restricting client access

To restrict the set of clients that can call a certain feature *h*, we will use the possibility for a class to have two or more **feature** clauses. The class will then be of the form

```
class S2 feature
        f …
        g …
feature {A, B}
        h …
        …
end
```

Features *f* and *g* have the same status as before: available to all clients. Feature *h* is available only to *A* and *B*, and to their descendants (the classes that inherit directly or indirectly from *A* or *B*). This means that with *x* declared of type *S2* a call of the form

```
x.h …
```

is invalid unless it appears in the text of *A*, *B*, or one of their descendants.

As a special case, if you want to hide a feature *i* from all clients, you may declare it as exported to an empty list of clients:

```
class S3 feature { }
        i …
end
```

This is not the recommended style; see S5 below.

In this case a call of the form *x.i* (…) is always invalid. The only permitted calls to *i* are unqualified calls of the form

```
i (…)
```

appearing in the text of a routine of *S3* itself, or one of its descendants. This mechanism ensures full information hiding.

The possibility of hiding a feature from all clients, as illustrated by *i*, is present in many O-O languages. But most do not offer the selective mechanism illustrated by *h*: exporting a feature to certain designated clients and their proper descendants. This is regrettable since many applications will need this degree of fine control.

The discussion section of the present chapter explains why selective exports are a critical part of the architectural mechanisms of the object-oriented approach, avoiding the need for "super-modules" that would hamper the simplicity of the method.

"The architectural role of selective exports", page 209.

We will encounter various examples of selective exports in subsequent chapters, and will study their methodological role in the design of good modular interfaces.

"SELECTIVE EXPORTS", 23.5, page 796.

Style for declaring secret features

A small point of style. A feature declared in the form used above for *i* is secret, but perhaps this property does not stand out strongly enough from the syntax. In particular, the difference with a public feature may not be visible enough, as in

Not the recommended style; see S5 next.

```
class S4 feature
        exported…
feature { }
        secret …
end
```

where feature *exported* is available to all clients whereas *secret* is available to no client. The difference between **feature** { }, with an empty list in braces, and **feature**, with no braces, is a little weak. For that reason, the recommended notation uses not an empty list but a list consisting of the single class *NONE*, as in

The recommended style.

```
class S5 feature
        … Exported …
feature {NONE}
        … Secret …
end
```

"The bottom of the pit", page 582.

Class *NONE*, which will be studied in a later chapter in connection with inheritance, is a Base library class which is so defined as to have no instances and no descendants. So exporting a feature to *NONE* only is, for all practical purposes, the same as keeping it secret. As a result there is no meaningful difference between the forms illustrated by *S4* and *S5*; for reasons of clarity and readability, however, the second form is preferred, and will be employed in the rest of this book whenever we need to introduce a secret feature.

Exporting to yourself

A consequence of the rules seen so far is that a class may have to export a secret feature. Assume the declaration

```
indexing
        note: "Invalid as it stands (see explanations below)"
class S6 feature
        x: S6
        my_routine is do … print (x.secret) … end
feature {NONE}
        secret: INTEGER
end -- class S6
```

By declaring *x* of type *S6* and making the call *x.secret*, the class becomes its own client. But this call is invalid, since *secret* is exported to no class! That the unauthorized client is *S6* itself does not make any difference: the {*NONE*} export status of *secret* makes any call *x.secret* invalid. Permitting exceptions would damage the simplicity of the rule.

The solution is simple: instead of **feature** {*NONE*} the header of the second **feature** clause should read **feature** {*S6*}, exporting the feature to the class itself and its descendants.

Be sure to note that this is only needed if you want to use the feature in a qualified call such as appears in *print* (*x.secret*). If you are simply using *secret* by itself, as in the

instruction *print* (*secret*), you of course do not need to export it at all. Features declared in a class must be usable by the routines of the class and its descendants; otherwise we could never do anything with a secret feature! Only if you use the feature indirectly in a qualified call do you need to export it to yourself.

7.9 PUTTING EVERYTHING TOGETHER

The previous discussions have introduced the basic mechanisms of object-oriented computation, but we are still missing the big picture: how does anything ever get executed?

Answering this question will help us piece everything together and understand how to build executable systems from individual classes.

General relativity

What is a little mind-boggling is that every description given so far of what happens at run time has been relative. The effect of a routine such as *translate* is relative to the current instance; within the class text, as noted, the current instance is not known. So we can only try to understand the effect of a call with respect to a specific target, such as *p1* in

p1.*translate* (*u*, *v*)

But this brings the next question: what does *p1* actually denote? Here again the answer is relative. The above call must appear in the text of some class such as *GRAPHICS*. Assume that *p1* is an attribute of class *GRAPHICS*. Then the occurrence of *p1* in the call, as noted above, may be viewed as a call: *p1* stands for *Current*.*p1*. So we have only pushed the problem further, as we must know what object *Current* stood for at the time of the above call! In other words, we must look at the client that called the routine of class *GRAPHICS* containing that call.

So this attempt at understanding a feature call starts off a chain of reasoning, which we will not be able to follow to the end unless we know where execution started.

The Big Bang

To understand what is going on let us generalize the above example to an arbitrary call. If we do understand that arbitrary call, we will indeed understand all of O-O computation, thanks to the Feature Call principle which stated that

F1 • No software element ever gets executed except as part of a routine call.

F2 • Every call has a target.

See page 186.

Any call will be of one of the following two forms (the argument list may be absent in either case):

- Unqualified: *f* (*a*, *b*, …)
- Qualified: *x*.*g* (*u*, *v*, …)

The call appears in the body of a routine *r*. It can only get executed as part of a call to *r*. Assume we know the target of that call, some object OBJ. Then the target *t* is easy to determine in each case:

T1 • For the unqualified form, *t* is simply OBJ. Cases T2, T3 and T4 will apply to the qualified form.

T2 • If *x* is an attribute, the *x* field of OBJ has a value which must be attached to some object; *t* is that object.

T3 • If *x* is a function, we must first execute the (unqualified) call to *x*; the result gives us *t*.

T4 • If *x* is a local entity of *r*, earlier instructions will have given *x* a value, which at the time of the call must be attached to a certain object; *t* is that object.

The only problem with these answers is of course that they are relative: they only help us if we know the current instance OBJ. What is OBJ? Why, the target of the current call, of course! As in the traditional song (the kid was eaten by the cat, the cat was bitten by the dog, the dog was beaten by the stick…), we do not see the end of the chain.

To transform these relative answers into absolute ones, then, we must know what happened when everything started — at Big Bang time. Here is the rule:

Definition: system execution

Execution of an object-oriented software system consists of the following two steps:

- Create a certain object, called the **root object** for the execution.
- Apply a certain procedure, called a **creation procedure**, to that object.

At Big Bang time, an object gets created, and a creation procedure gets started. The root object is an instance of a certain class, the system's **root class**; the creation procedure is one of the procedures of the root class. In all but trivial systems, the creation procedure will itself create new objects and call routines on them, triggering more object creations and more routine calls. System execution as a whole is the successive deployment of all the pieces in a giant and complex firework, all resulting directly or indirectly from the initial lighting of a minuscule spark.

Once we know where everything starts, it is not difficult to trace the fate of *Current* throughout this chain reaction. The first current object, at the start of everything (Big Bang time, when the root's creation procedure is called), is the root object. Then at any stage during system execution let *r* be the latest routine to have been called; if OBJ was the current object at the time of the call to *r*, here is what becomes of *Current* during the execution of *r*:

C1 • If *r* executes an instruction which does not call a routine (for example an assignment), we keep the same object as current object.

C2 • Starting an unqualified call also keeps the same object as current object.

C3 • Starting a qualified call *x.f …* causes the target object of that call, which is the object attached to *x* (determined from OBJ through the rules called T1 to T4 at the top of the previous page), to become the new current object. When the call terminates, OBJ resumes its role as current object.

In cases C2 and C3 the call may be to a routine that itself includes further calls, qualified or not; so this rule must be understood recursively.

There is nothing mysterious or confusing, then, in the rule for determining the target of any call, even though that rule is relative and in fact recursive. What is mind-boggling is the power of computers, the power we use to play sorcerer's apprentice by writing a deceptively small software text and then executing it to create objects and perform computations on them in numbers so large — number of objects, number of computations — as to appear almost infinite when measured on the scale of human understanding.

Systems

The emphasis in this chapter is on classes: the individual components of object-oriented software construction. To obtain executable code, we must assemble classes into systems.

The definition of a system follows from the previous discussion. To make up a system we need three things:

- A set *CS* of classes, called the system's **class set**.
- The indication of which class in *CS* is the **root class**.
- The indication of which procedure of the root class is the **root creation procedure**.

To yield a meaningful system, these elements must satisfy a consistency condition, **system closure**: any class needed directly or indirectly by the root class must be part of *CS*.

Let us be a little more precise:

- A class *C* **needs directly** a class *D* if the text of *C* refers to *D*. There are two basic ways in which *C* may need directly *D*: *C* may be a client of *D*, as defined earlier in this chapter, and *C* may inherit from *D*, according to the inheritance relation which we will study later.
- A class *C* **needs** a class *E*, with no further qualification, if *C* is *E* or *C* needs directly a class *D* which (recursively) needs *E*.

With these definitions we may state the closure requirement as follows:

Definition: system closure

A system is closed if and only if its class set contains all classes needed by the root class.

If the system is closed, a language-processing tool, such as a compiler, will be able to process all its classes, starting with the root class, and recursively handling needed

classes as it encounters their names. If the tool is a compiler, it will then produce the executable code corresponding to the entire system.

This act of tying together the set of classes of a system, to generate an executable result, is called **assembly** and is the last step in the software construction process.

Not a main program

The discussions in the previous chapters repeatedly emphasized that systems developed with the object-oriented method have no notion of main program. By introducing the notion of root class, and requiring the system specification to indicate a particular creation procedure, have we not brought main programs back through a side door?

Not quite. What is wrong with the traditional notion of main program is that it merges two unrelated concepts:

- The place where execution begins.
- The top, or fundamental component of the system's architecture.

The first of these is obviously necessary: every system will begin its execution somewhere, so we must have a way to let developers specify the starting point; here they will do so by specifying a root class and a creation procedure. (In the case of concurrent rather than sequential computation we may have to specify several starting points, one per independent thread of computation.)

On the concept of top, enough abuse has been heaped in earlier chapters to make further comments unnecessary.

But regardless of the intrinsic merit of each of the two notions, there is no reason to merge them: no reason to assume that the starting point of a computation will play a particularly important role in the architecture of the corresponding system. Initialization is just one of many aspects of a system. To take a typical example, the initialization of an operating system is its booting procedure, usually a small and relatively marginal component of the OS; using it as the top of the system's design would not lead to an elegant or useful architecture. The notion of system, and object technology in general, rely in fact on the reverse assumption: that the most important property of a system is the set of classes that it contains, the individual capabilities of these classes, and their relationships. In this view the choice of a root class is a secondary property, and should be easy to change as the system evolves.

For a critique of function-based decomposition see "FUNCTIONAL DECOMPOSITION", 5.2, page 103

As discussed extensively in an earlier chapter, the quest for extendibility and reusability requires that we shed the practice of asking "what is the main function?" at an early stage of the system's design and of organizing the architecture around the answer. Instead, the approach promotes the development of reusable software components, built as abstract data type implementations — classes. Systems are then built as reconfigurable assemblies of such components.

In fact, you will not always build systems in the practice of O-O software development. An important application of the method is to develop **libraries** of reusable

components — classes. A library is not a system, and has no root class. When developing a library, you may of course need to produce, compile and execute one or more systems along the way, but such systems are a means, not an end: they help test the components, and will usually not be part of the library as finally delivered. The actual delivered product is the set of classes making up the library, which other developers will then use to produce their own systems — or again their own libraries.

Assembling a system

The process of putting together a number of classes (one of which is designated as root) to produce an executable system was called "assembly" above. How in practice do we assemble a system?

Let us assume an operating system of the usual form, where we will keep our class texts stored in files. The language processing tool in charge of this task (compiler, interpreter) will need the following information:

A1 • The name of the root class.

A2 • A **universe**, or set of files which may contain the text of classes needed by the root (in the above precise sense of "needed").

This information should not be included in the class texts themselves. Identifying a class as root in its own text (A1) would violate the "no main program" principle. Letting a class text include information about the files where the needed classes reside would tie the class to a particular location in the file system of a given installation; this would prevent use of the class by another installation and is clearly inappropriate.

These observations suggest that the system assembly process will need to rely on some information stored outside of the text of the classes themselves. To provide this information we will rely on a little control language called Lace. Let us observe the process, but not until we have noted that the details of Lace are not essential to the method; Lace is just an example of a control language, allowing us to keep the O-O components (the classes) autonomous and reusable, and to rely on a separate mechanism for their actual assembly into systems.

A typical Lace document, known as an **Ace file**, might appear as follows:

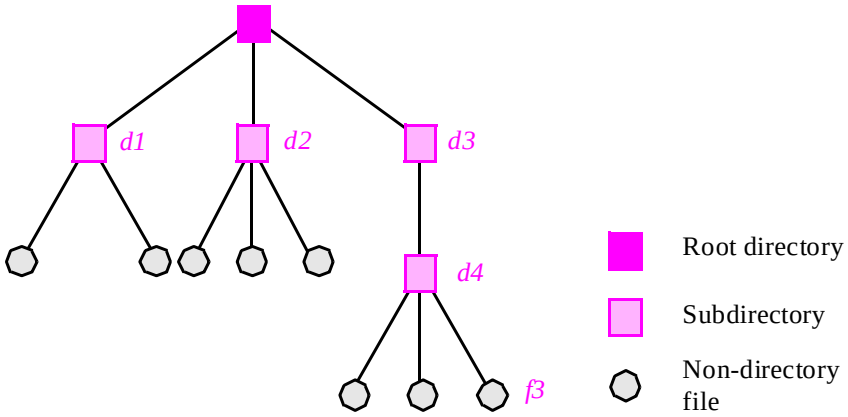
```
system painting root
        GRAPHICS ("painting_application")
cluster
        base_library: "\library\base";
        graphical_library: "\library\graphics";
        painting_application: "\user\application"
end -- system painting
```

The **cluster** clause defines the universe (the set of files containing class texts). It is organized as a list of clusters; a cluster is a group of related classes, representing a subsystem or a library.

Chapter 28 discusses the cluster model.

In practice, an operating system such as Windows, VMS or Unix provides a convenient mechanism to support the notion of cluster: directories. Its file system is structured as a tree, where only the terminal nodes (leaves), called “plain files”, contain directly usable information; the internal nodes, called directories, are sets of files (plain files or again directories).

A directory structure



We may associate each cluster with a directory. This convention is used in Lace as illustrated above: every cluster, with a Lace name such as *base_library*, has an associated directory, whose name is given as a string in double quotes, such as "*\library\base*". This file name assumes Windows conventions (names of the form *\dir1\dir2\...*), but this is just for the sake of the example. You can obtain the corresponding Unix names by replacing the backslash characters \ by slashes /.

> Although by default you may use the hierarchical structure of directories to represent cluster nesting, Lace has a notion of subcluster through which you can define the logical structure of the cluster hierarchy, regardless of the clusters’ physical locations in the file system.

The directories listed in the **cluster** clause may contain files of all kinds. To determine the universe, the system assembly process will need to know which ones of these files may contain class texts. A simple convention is to require the text of any class of name *NAME* to be stored in a file of name *name*.**e** (lower case). Let us assume this convention (which can easily be extended for more flexibility) for the rest of this discussion. Then the universe is the set of files having names of the form *name*.**e** in the list of directories appearing in the **cluster** clause.

The **root** clause of Lace serves to designate the root class of the system. Here the root class is *GRAPHICS* and, as indicated in parentheses, it appears in the *painting_application* cluster. If there is only one class called *GRAPHICS* in the universe, it is not necessary to specify the cluster.

Assume that you start a language processing tool, for example a compiler, to process the system described by the above Ace. Assume further that none of the classes in the system has been compiled yet. The compiler finds the text of the root class, *GRAPHICS*, in the file *graphics*.**e** of the cluster *painting_application*; that file appears in the directory

\user\application. By analyzing the text of class *GRAPHICS*, the compiler will find the names of the classes needed by *GRAPHICS* and will look for files with the corresponding **.e** names in the three cluster directories. It will then apply the same search to the classes needed by these new classes, repeating the process until it has located all the classes needed directly or indirectly by the root.

An important property of this process is that it should be **automatic**. As a software developer, you should not have to write lists of dependencies between modules (known as "Make files"), or to indicate in each file the names of the files that will be needed for its compilation (through what is known in C and C++ as "Include directives"). Not only is it tedious to have to create and maintain such dependency information manually; this process also raises the possibility of errors when the software evolves. All that the Ace requires you to provide is the information that no tool can find by itself: the name of the root class, and the list of locations in the file system where needed classes — what earlier was called the *class set* of the system — may appear.

To simplify the work of developers further, a good compiler will, when called in a directory where no Ace is present, construct a template Ace whose **cluster** clause includes the basic libraries (kernel, fundamental data structures and algorithms, graphics etc.) and the current directory, so that you will only have to fill in the name of the system and of its root class, avoiding the need to remember the syntax of Lace.

The end result of the compilation process is an executable file, whose name is the one given after **system** in the Ace — *painting* in the example.

The Lace language includes a few other simple constructs, used to control the actions of language processing tools, in particular compiler options and assertion monitoring levels. We will encounter some of them as we explore further O-O techniques. Lace, as noted, also supports the notion of logical subcluster, so that you can use it to describe complex system structures, including the notions of subsystem and multi-level libraries.

Using a system description language such as Lace, separate from the development language, allows classes to remain independent from the system or systems in which they intervene. Classes are software components, similar to chips in electronic design; a system is one particular assembly of classes, similar to a board or a computer made by assembling a certain set of chips.

Printing your name

Reusable software components are great, but sometimes all you want to do is just a simple task, such as printing a string. You may have been wondering how to write a "program" that will do it. Having introduced the notion of system, we can answer this burning question. (Some people tend to be nervous about the whole approach until they see how to do this, hence this little digression.)

The following little class has a procedure which will print a string:

```
class SIMPLE creation
        make
feature
        make is
                        -- Print an example string.
                do
                        print_line ("Hello Sarah!")
                end
end
```

On GENERAL see "Universal classes", page 580.

The procedure *print_line* can take an argument of any type; it prints a default representation of the corresponding object, here a string, on a line. Also available is *print* which does not go to a new line after printing. Both procedures are available to all classes, coming from a universal ancestor, *GENERAL*, as explained in a later chapter.

To obtain a system that will print the given string, do the following:

E1 • Put the above class text in a file called *simple*.**e** in some directory.

E2 • Start the compiler.

E3 • If you have not provided an Ace, you will be prompted to edit a new one, automatically generated from a template; just fill in the name of the root class, *SIMPLE*, the name of the system — say *my_first* — and the cluster directory.

E4 • Exit from the editor; the compiler will assemble the system and produce an executable file called *my_first*.

E5 • Execute the result. On platforms such as Unix with a notion of command-line execution a command will have been generated, of name *my_first*; simply type that name. On graphical platforms such as Windows and OS/2, a new icon will have appeared, labeled *my_first*; just double-click on that icon.

The result of the last step will be, as desired, to print on your console the message

```
Hello Sarah!
```

Structure and order: the software developer as arsonist

We now have an overall picture of the software construction process in the object-oriented method — assembling classes into systems. We also know how to reconstruct the chain of events that will lead to the execution of a particular operation. Assume this operation is

[A]

```
x.g (u, v, ...)
```

appearing in the text of a routine *r* of a class *C*, of which we assume *x* to be an attribute. How does it ever get executed? Let us recapitulate. You must have included *C* in a system, and assembled that system with the help of an appropriate Ace. Then you must have started an execution of that system by creating an instance of its root class. The root's

creation procedure must have executed one or more operations which, directly or indirectly, caused the creation of an instance C_OBJ of *C*, and the execution of a call of the form

[B]

```
a.r (...)
```

where *a* was at the time attached to C_OBJ. Then the call shown as [A] will execute *g*, with the arguments given, using as target the object attached to the *x* field of C_OBJ.

So by now we know (as well we should) how to find out the exact sequence of events that will occur during the execution of a system. But this assumes we look at the entire system. In general we will not be able, just by examining the text of a given class, to determine the order in which clients will call its various routines. The only ordering property that is immediately visible is the order in which a given routine executes the instructions of its body.

Even at the system level, the structure is so decentralized that the task of predicting the precise order of operations, although possible in principle, is often difficult. More importantly, it is usually not very interesting. Remember that we treat the root class as a somewhat superficial property of the system — a particular choice, made late in the development process, of how we are going to combine a set of individual components and schedule their available operations.

This downplaying of ordering constraints is part of object technology's constant push for decentralization in system architectures. The emphasis is not on "the" execution of "the" program (as in Pascal or C programming and many design methods) but on the services provided by a set of classes through their features. The *order* in which the services will be exercised, during the execution of a particular system built from these classes, is a secondary property.

The method goes in fact further by prescribing that *even if you know* the order of execution you should not base any serious system design decision on it. The reason for this rule was explored in earlier chapters: it is a consequence of the concern for extendibility and reusability. It is much easier to add or change services in a decentralized structure than to change the order of operations if that order was one of the properties used to build the architecture. This reluctance of the object-oriented method to consider the order of operations as a fundamental property of software systems — what an earlier discussion called the shopping list approach — is one of its major differences with most of the other popular software design methods.

See "Premature ordering", page 110.

These observations once again evoke the picture of the software developer as firework expert or perhaps arsonist. He prepares a giant conflagration, making sure that all the needed components are ready for assembly and all the needed connections present. He then lights up a match and watches the blaze. But if the structure has been properly set up and every component is properly attached to its neighbors, there is no need to follow or even try to predict the exact sequence of lightings; it suffices to know that every part that must burn will burn, and will not do so before its time has come.

7.10 DISCUSSION

As a conclusion to this chapter, let us consider the rationale behind some of the decisions made in the design of the method and notation, exploring along the way a few alternative paths. Similar discussion sections will appear at the end of most chapters introducing new constructs; their aim is to spur the reader's own thinking by presenting a candid, uncensored view of a few delicate issues.

Form of declarations

To hone our critical skills on something that is not too life-threatening, let us start with a syntactical property. One point worth noting is the notation for feature declarations. For routines, there are none of the keywords **procedure** or **function** such as they appear in many languages; the form of a feature determines whether it is an attribute, a procedure or a function. The beginning of a feature declaration is just the feature name, say

```
f …
```

When you have read this, you must still keep all possibilities open. If a list of arguments comes next, as in

```
g (a1: A; b1: B; …) …
```

then you know *g* is a routine; it could still be either a function or a procedure. Next a type may come:

```
f: T …
g (a1: A; b1: B; …): T …
```

In the first example, *f* can still be either an attribute or a function without arguments; in the second, however, the suspense stops, as *g* can only be a function. Coming back to *f*, the ambiguity will be resolved by what appears after *T*: if nothing, *f* is an attribute, as in

```
my_file: FILE
```

But if an **is** is present, followed by a routine body (**do** or the variants **once** and **external** to be seen later), as in

```
f: T is
            -- …
        do … end
```

f is a function. Yet another variant is:

```
f: T is some_value
```

Chapter 18 discusses constant attributes.

which defines *f* as a **constant attribute** of value *some_value*.

The syntax is designed to allow easy recognition of the various kinds of feature, while emphasizing the fundamental similarities. The very notion of feature, covering routines as well as attributes, is in line with the Uniform Access principle — the goal of providing clients with abstract facilities and downplaying their representation differences. The similarity between feature declarations follows from the same ideas.

Attributes vs. functions

Let us explore further the consequences of the Uniform Access principle and of grouping attributes and routines under a common heading — features.

"Uniform Access", page 55; see also, in the present chapter, "Uniform access", page 175.

The principle stated that clients of a module should be able to use any service provided by the module in a uniform way, regardless of how the service is implemented — through storage or through computation. Here the services are the features of the class; what is meaningful for clients is the availability of certain features and their properties. Whether a given feature is implemented by storing appropriate data or by computing the result on demand is, for most purposes, irrelevant.

Assume for example a class *PERSON* containing a feature *age* of type *INTEGER*, with no arguments. If the author of a client class writes the expression

Isabelle.*age*

the only important information is that *age* will return an integer, the age field of an instance of *PERSON* attached, at run-time, to the entity *Isabelle*. Internally, *age* may be either an attribute, stored with each object, or a function, computed by subtracting the value of a *birth_date* attribute from the current year. But the author of the client class does not need to know which one of these solutions was chosen by the author of *PERSON*.

The notation for *accessing* an attribute, then, is the same as for calling a routine; and the notations for *declaring* these two kinds of feature are as similar as conceptually possible. Then if the author of a supplier class reverses an implementation decision (implementing as a function a feature that was initially an attribute, or conversely), clients will not be affected; they will require neither change, possibly not even recompilation.

The figures appeared on pages 174 and 175.

The contrast between the supplier's and client's view of the features of a module was apparent in the two figures which helped introduce the notion of feature earlier in this chapter. The first used as its primary criterion the distinction between routines and attributes, reflecting the internal (implementation) view, which is also the supplier's view. In the second figure, the primary distinction was between commands and queries, the latter further subdivided into queries with and without arguments. This is the external view — the client's view.

The decision to treat attributes and functions without arguments as equivalent for clients has two important consequences, which later chapters will develop:

- The first consequence affects software documentation. The standard client documentation for a class, known as the **short form** of the class, will be devised so as not to reveal whether a given feature is an attribute or a function (in cases for which it could be either).

"Using assertions for documentation: the short form of a class", page 389.

- The second consequence affects inheritance, the major technique for adapting software components to new circumstances without disrupting existing software. If a certain class introduces a feature as a function without arguments, descendant classes will be permitted to **redefine** the feature as an attribute, substituting memory for computation.

"Redeclaring a function into an attribute", page 491.

Exporting attributes

The class text was on page 176.

A consequence of the preceding observations is that classes may export attributes. For example, class *POINT*, in the cartesian implementation introduced earlier, has attributes *x* and *y*, and exports them to clients in exactly the same way as the functions *rho* and *theta*. To obtain the value of an attribute for a certain object, you simply use feature call notation, as in *my_point*.*x* or *my_point*.*theta*.

This ability to export attributes differs from the conventions that exist in many O-O languages. Typical of these is Smalltalk, where only routines (called "methods") may be exported by a class; attributes ("instance variables") are not directly accessible to clients.

A consequence of the Smalltalk approach is that if you want to obtain the effect of exporting an attribute you have to write a small exported function whose only purpose is to return the attribute's value. So in the *POINT* example we could call the attributes *internal_x* and *internal_y*, and write the class as follows (using the notation of this book rather than the exact Smalltalk syntax, and calling the functions *abscissa* and *ordinate* rather than *x* and *y* to avoid any confusion):

```
class POINT feature -- Public features:
        abscissa: REAL is
                        -- Horizontal coordinate
                do Result := internal_x end
        ordinate: REAL is
                        -- Vertical coordinate
                do Result := internal_y end
        … Other features as in the earlier version …
feature {NONE} -- Features inaccessible to clients:
        internal_x, internal_y: REAL
end
```

This approach has two drawbacks:

- It forces authors of supplier classes to write many small functions such as *abscissa* and *ordinate*. Although in practice such functions will be short (since the syntax of Smalltalk is terse, and makes it possible to give the same name to an attribute and a function, avoiding the need to devise special attribute names such as *internal_x* and *internal_y*), writing them is still a waste of effort on the part of the class author, and reading them is a useless distraction for the class reader.
- The method entails a significant performance penalty: every access to a field of an object now requires a routine call. No wonder object technology has developed a reputation for inefficiency in some circles. (It is possible to develop an optimizing compiler which will expand calls to *abscissa*-style functions in-line, but then what is the role of these functions?)

The technique discussed in this chapter seems preferable. It avoids the need for cluttering class texts with numerous little extra functions, and instead lets the class designers export attributes as needed. Contrary to what a superficial examination might

suggest, this policy does not violate information hiding; it is in fact a direct implementation of this principle and of the associated principle of Uniform Access. To satisfy these requirements it suffices to make sure that attributes, as seen by clients, are indistinguishable from functions without arguments, and that they have the same properties for inheritance and class documentation.

This technique reconciles the goals of Uniform Access (essential for the clients), ease of writing class texts (essential for the suppliers), and efficiency (essential for everyone).

The client's privileges on an attribute

Exporting an attribute, using the techniques just discussed, allows clients to access the value of an attribute for a certain object, as in *my_point.x* It does not allow clients to modify that value. You may not assign to an attribute; the assignment

Warning: illegal construct — for illustration only.

```
my_point.x := 3.7
```

is syntactically illegal. The syntax rule is simple: *a.attrib*, if *attrib* is an attribute (or for that matter a function) is an expression, not an entity, so you cannot assign to it, any more than you can assign to the expression *a + b*.

To make *attrib* accessible in modification mode, you must write and export an appropriate procedure, of the form:

```
set_attrib (v: G) is
        -- Set to v the value of attrib.
    do
        attrib := v
    end
```

Instead of this convention, one could imagine a syntax for specifying access rights, such as

Warning: not a retained notation. For discussion only.

```
class C feature [AM]
    …
feature [A] {D, E}
    …
```

where *A* would mean access and *M* modification. (Specifying *A* could be optional: if you export something you must at least allow clients to access it in read mode). This would avoid the frequent need for writing procedures similar to *set_attrib*.

Besides not justifying the extra language complication, this solution is not flexible enough. In many cases, you will want to export *specific* ways of modifying an attribute. For example, the following class exports a counter, and the right to modify it not arbitrarily but only by increments of +1 or –1:

```
class COUNTING feature
        counter: INTEGER
        increment is
                -- Increment counter
            do
                count := count + 1
            end
        decrement is
                -- Decrement counter
            do
                count := count — 1
            end
end
```

Similarly, class *POINT* as developed in this chapter does not let its clients set the *x* and *y* of a point directly; clients can change the values of these attributes, but only by going through the specific mechanisms that have been exported for that purpose, procedures *translate* and *scale*.

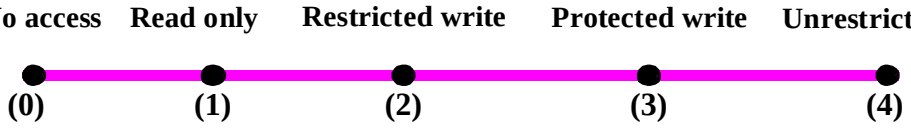
When we study assertions we will see another fundamental reason why it is inappropriate to let clients perform direct assignments of the *a.attrib := some_value* form: not all *some_value* are acceptable. You may define a procedure such as

```
set_polygon_size (new_size: INTEGER) is
        -- Set the number of polygon vertices to new_size.
    require
        new_size >= 3
    do
        size := new_size
    end
```

requiring any actual argument to be 3 or more. Direct assignments would make it impossible to enforce this constraint; a call could then produce an incorrect object.

These considerations show that a class writer must have at his disposal, for each attribute, *five* possible levels for granting access privileges to clients:

Possible client privileges on an attribute



Level 0 is total protection: clients have no way of accessing the attribute. At level 1 and above, you make the attribute available for access, but at level 1 you do not grant any modification right. At level 2, you let clients modify the attribute through specific algorithms. At level 3, you let them set the value, but only if it satisfies certain constraints, as in the polygon size example. Level 4 removes the constraints.

The solution described in this chapter is a consequence of this analysis. Exporting an attribute only gives clients access permission (level 1); permission to modify is specified by writing and exporting appropriate procedures, which give clients restricted rights as in the counter and point examples (level 2), direct modification rights under some constraints (3) or unrestricted rights (4).

This solution is an improvement over the ones commonly found in O-O languages:

- In Smalltalk, as noted, you have to write special encapsulation functions, such as the earlier *abscissa* and *ordinate*, just to let clients access an attribute at level 1; this may mean both extra work for the developer and a performance overhead. Here there is no need to write routines for attribute access; only for attribute modifications (levels 2 and above) do we require writing a routine, since it is conceptually necessary for the reasons just seen.

- C++ and Java are the other extreme: if you export an attribute then it is up for grabs at level 4: clients can set it through direct assignments in the *my_point.x := 3.7* style as well as access its value. The only way to achieve level 2 (not 3 in the absence of an O-O assertion mechanism in these languages) is to hide the attribute altogether, and then write exported routines, both procedures for modification (levels 2 or 4) and functions for access (level 1). But then you get the same behavior as with the Smalltalk approach.

This discussion of a fairly specific language trait illustrates two of the general principles of language design: do not needlessly bother the programmer; know when to stop introducing new language constructs at the point of diminishing returns.

Optimizing calls

At levels 2 and 3 of the preceding discussion, the use of explicit procedure calls such as *my_polygon.set_size (5)* to change an attribute value is inevitable. At level 4, one could fear the effect on performance of using the *set_attrib*-style. The compiler, however, can generate the same code for *my_point.set_x (3.7)* as it would for *my_point.x := 3.7* had this last phrasing been legal.

> ISE's compiler achieves this through a general in-line expansion mechanism, which eliminates certain routine calls by inserting the routine body directly, with appropriate argument substitutions, into the caller's code.

In-line expansion is indeed one of the transformations that we may expect from an optimizing compiler for an object-oriented language. The modular style of development fostered by object technology produces many small routines. It would be unacceptable for developers to have to worry about the effect of the corresponding calls on performance. They should just use the clearest and most robust architecture they can devise, according to the modularity principles studied in this book, and expect the compiler to get rid of any calls which may be relevant to the design but not necessary for the execution.

In some programming languages, notably Ada and C++, developers specify what routines they want expanded in-line. I find it preferable to treat this task as an automatic optimization, for several reasons:

- It is not always correct to expand a call in-line; since the compiler must, for correctness, check that the optimization applies, it may just as well spare developers the trouble of requesting it in the first place.
- With changes in the software, in particular through inheritance, a routine which was inlinable may become non-inlinable. A software tool is better than a human at detecting such cases.
- On a large system, compilers will always be more effective. They are better equipped to apply the proper heuristics — based on routine size and number of calls — to decide what routines should be inlined. This is again especially critical as the software changes; we cannot expect a human to track the evolution of every piece.
- Software developers have better things to do with their time.

"Garbage collector requirements", page 305, and *"The C++ approach to binding", page 513*.

The modern software engineering view is that such tedious, automatable and delicate optimizations should be handled by software tools, not people. The policy of leaving them to the responsibility of developers is one of the principal criticisms that have been leveled at C++ and Ada. We will encounter this debate again in studying two other key mechanisms of object technology: memory management, and dynamic binding.

The architectural role of selective exports

The selective export facility is not just a convenience; it is essential to object-oriented architecture. It enables a set of conceptually related classes to make some of their features accessible to each other without releasing them to the rest of the world, that is to say, without violating the rule of Information Hiding. It also helps us understand a frequently debated issue: whether we need modules above the level of classes.

Without selective exports, the only solution (other than renouncing Information Hiding altogether) would be to introduce a new modular structure to group classes. Such super-modules, similar to Ada's or Java's packages, would have their own rules for hiding and exporting. By adding a completely new and partly incompatible module level to the elegant framework defined by classes, they would yield a bigger, hard-to-learn language.

Rather than using a separate package construct, the super-modules could themselves be classes; this is the approach of Simula, which permits class nesting. It too brings its share of extra complexity, for no clear benefit.

We have seen that the simplicity of object technology relies for a good part on the use of a single modular concept, the class; its support for reusability relies on our ability to extract a class from its context, keeping only its logical dependencies. With a super-module concept we run the risk of losing these advantages. In particular, if a class belongs to a package or an enclosing class we will not be able to reuse it by itself; if we want to include it in another super-module we will need either to import the entire original super-module, or to make a copy of the class — not an attractive form of reuse.

The need will remain to group classes in structured collections. This will be addressed in a later chapter through the notion of *cluster*. But the cluster is a management and organizational notion; making it a language construct would jeopardize the simplicity of the object-oriented approach and its support for modularity.

Chapter 28.

When we want to let a group of classes grant each other special privileges, we do not need a super-module; selective exports, a modest extension to basic information hiding, provide a straightforward solution, allowing classes to retain their status of free-standing software components. This is, in my opinion, a typical case of how a simple, low-tech idea can outperform the heavy artillery of a "powerful" mechanism.

Listing imports

Each class lists, in the headers of its **feature** clauses, the features that it makes available to others. Why not, one might ask, also list features obtained from other classes? The encapsulation language Modula-2 indeed provides an **import** clause.

In a typed approach to O-O software construction, however, such a clause would not serve any purpose other than documentation. To use a feature *f* from another class *C*, you must be a client or (through inheritance) a descendant of that class. In the first case, the only one seen so far, this means that every use of *f* is of the form

a.*f*

where, since our notation is typed, *a* must have been declared:

a: *C*

showing without any ambiguity that *f* came from the *C*. In the descendant case the information will be available from the official class documentation, its "flat-short form".

"The flat-short form", page 543.

So there is no need to bother developers with import clauses.

There is a need, however, to *help* developers with import documentation. A good graphical development environment should include mechanisms that enable you, by clicking a button, to see the suppliers and ancestors of a class, and follow the import chain further by exploring their own suppliers and ancestors.

See chapter 36.

Denoting the result of a function

An interesting language issue broached earlier in this chapter is how to denote function results. It is worth exploring further although it applies to non-O-O languages as well.

Consider a function — a value-returning routine. Since the purpose of any call to the function is to compute a certain result and return it to the caller, the question arises of how to denote that result in the text of the function itself, in particular in the instructions which initialize and update the result.

The convention introduced in this chapter uses a special entity, *Result*, treated as a local entity and initialized to the appropriate default value; the result returned by a call is

the final value of *Result*. Because of the initialization rules, that value is always defined even if the routine body contains no assignment to *Result*. For example, the function

```
f: INTEGER is
        do
                if some_condition then Result := 10 end
        end
```

will return the value 10 if *some_condition* is satisfied at the time of the call, and 0 (the default initialization value for *INTEGER*) otherwise.

The technique using *Result* originated, as far as I know, with the notation developed in this book. (Since the first edition it has found its way into at least one other language, Borland's Delphi.) Note that it would not work in a language allowing functions to be declared within functions, as the name *Result* would then be ambiguous. Among the techniques used in earlier languages, the most common are:

A • Explicit return instructions (C, C++/Java, Ada, Modula-2).

B • Treating the function name as a variable (Fortran, Algol 60, Simula, Algol 68, Pascal).

Convention A relies on an instruction of the form **return** *e* whose execution terminates the current execution of the enclosing function, returning *e* as the result. This technique has the benefit of clarity, since it makes the returned value stand out clearly from the function text. But it suffers from several drawbacks:

A1 • Often, the result must in practice be obtained through some computation: an initialization and a few subsequent updates. This means you must introduce and declare an extraneous variable (an entity in the terminology of this chapter) just for the purpose of holding the intermediate results of the computation.

A2 • The technique tends to promote multiple-exit modules, which are contrary to the principles of good program structuring.

A3 • The language definition must specify what will happen if the last instruction executed by a call to the function is not a **return**. The Ada result in this case is to raise … a run-time exception! (This may be viewed as the ultimate in buck-passing, the language designers having transferred the responsibility for language design issues not just to software developers, but finally to the *end-users* of the programs developed in the language!)

Note that it is possible to solve the last two problems by treating **return** not as an instruction, but as a syntactic clause which would be a required part of any function text:

```
function name (arguments): TYPE is
        do
                …
        return
                expression
        end
```

This solution remains compatible in spirit with the idea of a **return** instruction while addressing its most serious deficiencies. No common language, however, uses it, and of course it still leaves problem A1 open.

The second common technique, B, treats a function's name as a variable within the text of the function. The value returned by a call is the final value of that variable. (This avoids introducing a special variable as mentioned under A1.)

The above three problems do not arise in this approach. But it raises other difficulties because the same name now ambiguously denotes both a function and a variable. This is particularly confusing in a language allowing recursion, where a function body may use the function's name to denote a recursive call. Because an occurrence of the function's name now has two possible meanings, the language must define precise conventions as to when it denotes the variable, and when it denotes a function call. Usually, in the body of a function *f*, an occurrence of the name *f* as the target of an assignment (or other contexts implying a value to be modified) denotes the variable, as in

f := *x*

and an occurrence of *f* in an expression (or other contexts implying a value to be accessed) denotes a recursive function call, as in

x := *f*

which is valid only if *f* has no arguments. But then an assignment of the form

f := *f* + *1*

will be either rejected by the compiler (if *f* has arguments) or, worse, understood as containing a recursive call whose result gets assigned to *f* (the variable). The latter interpretation is almost certainly not what the developer had in mind: if *f* had been a normal variable, the instruction would simply have increased its value by one. Here the assignment will usually cause a non-terminating computation. To obtain the desired effect, the developer will have to introduce an extra variable; this takes us back to problem A1 above and defeats the whole purpose of using technique B.

The convention introduced in this chapter, relying on the predefined entity *Result*, avoids the drawbacks of both A and B. An extra advantage, in a language providing for default initialization of all entities including *Result*, is that it simplifies the writing of functions: if, as often happens, you want the result to be the default value except in specific cases, you can use the scheme

```
do
    if some_condition then Result := "Some specific value" end
end
```

without worrying about an **else** clause. The language definition must, of course, specify all default values in an unambiguous and platform-independent way; the next chapter will introduce such conventions for our notation.

Page 233.

A final benefit of the *Result* convention will become clear when we study Design by Contract: we can use *Result* to express an abstract property of a function's result,

Chapter 11.

independent of its implementation, in the routine's postcondition. None of the other conventions would allow us to write

```
infix "|_" (x: REAL): INTEGER is
        -- Integer part of x
    do
        … Implementation omitted …
    ensure
        no_greater: Result <= x
        smallest_possible: Result + 1 > x
    end
```

The postcondition is the **ensure** clause, stating two properties of the result: that it is no greater than the argument; and that adding 1 to it yields a result greater than the argument.

Complement: a precise definition of entities

It will be useful, while we are considering notational problems, to clarify a notion that has repeatedly been used above, but not yet defined precisely: entities. Rather than a critical concept of object technology, this is simply a technical notion, generalizing the traditional notion of variable; we need a precise definition.

Entities as used in this book cover names that denote run-time values, themselves attached to possible objects. We have now seen all three possible cases:

Definition: entity

An entity is one of the following:

E1 • An attribute of a class.

E2 • A routine's local entity, including the predefined entity *Result* for a function.

E3 • A formal argument of a routine.

Case E2 indicates that the entity *Result* is treated, for all purposes, as a local entity; other local entities are introduced in the **local** clause. *Result* and other local entities of a routine are initialized anew each time the routine is called.

All entities except formal arguments (E3) are writable, that is to say may appear as the target *x* of an assignment *x* := *some_value*.

7.11 KEY CONCEPTS INTRODUCED IN THIS CHAPTER

- The fundamental concept of object technology is the notion of class. A class is an abstract data type, partially or fully implemented.
- A class may have instances, called objects.

- Do not confuse objects (dynamic items) with classes (the static description of the properties common to a set of run-time objects).
- In a consistent approach to object technology, every object is an instance of a class.
- The class serves as both a module and a type. The originality and power of the O-O model come in part from the fusion of these two notions.
- A class is characterized by features, including attributes (representing fields of the instances of the class) and routines (representing computations on these instances). A routine may be a function, which returns a result, or a procedure, which does not.
- The basic mechanism of object-oriented computation is feature call. A feature call applies a feature of a class to an instance of that class, possibly with arguments.
- Feature call uses either dot notation (for identifier features) or operator notation, prefix or infix (for operator features).
- Every operation is relative to a "current instance" of a class.
- For clients of a class (other classes which use its features), an attribute is indistinguishable from a function without arguments, in accordance with the Uniform Access principle.
- An executable assembly of classes is called a system. A system contains a root class and all the classes which the root needs directly or indirectly (through the client and inheritance relations). To execute the system is to create an instance of the root class and to call a creation procedure on that instance.
- Systems should have a decentralized architecture. Ordering relations between the operations are inessential to the design.
- A small system description language, Lace, makes it possible to specify how a system should be assembled. A Lace specification, or Ace, indicates the root class and the set of directories where the system's clusters reside.
- The system assembly process should be automatic, with no need for Make files or Include directives.
- The Information Hiding mechanism needs flexibility: besides being hidden or generally available, a feature may need to be exported to some clients only; and an attribute may need to be exported for access only, access and restricted modification, or full modification.
- Exporting an attribute gives clients the right to access it. Modifying it requires calling the appropriate exported procedure.
- Selective exports are necessary to enable groups of closely related classes to gain special access to each other's features.
- There is no need for a super-module construct above classes. Classes should remain independent software components.
- The modular style promoted by object-oriented development leads to many small routines. Inlining, a compiler optimization, removes any potential efficiency

consequence. Detecting inlinable calls should be the responsibility of the compiler, not software developers.

7.12 BIBLIOGRAPHICAL NOTES

Chapter 35, bibliography on page 1138.

The notion of class comes from the Simula 67 language; see the bibliographical references of the corresponding chapter. A Simula class is both a module and a type, although this property was not emphasized in the Simula literature, and was dropped by some successors of Simula.

The Single Target principle may be viewed as a software equivalent of a technique that is well known in mathematical logic and theoretical computing science: **currying**. To curry a two-argument function *f* is to replace it by a one-argument function *g* yielding a one-argument function as a result, such that for any applicable *x* and *y*:

$$(g\ (x))\ (y) = f\ (x, y)$$

To curry a function, in other words, is to specialize it on its first argument. This is similar to the transformation described in this chapter to replace a traditional two-argument routine *rotate*, called under the form

rotate (*some_point, some_angle*)

by a one-argument function with a target, called under the form

some_point.*rotate* (*some_angle*)

Chapter 32 (discussion on the CD).

[M 1990]describes currying and some of its applications to computing science, in particular the formal study of programming language syntax and semantics. We will encounter currying again in the discussion of graphical user interfaces.

A few language designs have used the concept of object as a software construct rather than just a run-time notion as described in this chapter. In such approaches, meant for exploratory programming, there may be no need for a notion of class. The most notable representative of this school of thought is the Self language [Chambers 1991], which uses "prototypes" rather than classes.

The detail of the conventions for infix and prefix operators, in particular the precedence table, is given in [M 1992].

James McKim brought to my attention the final argument for the *Result* convention (its use for postconditions).

EXERCISES

E7.1 Clarifying the terminology

[This exercise requires two well-sharpened pencils, one *blue* and the other *red*.]

Study the textbook extract used earlier in this chapter to illustrate the confusion between objects and classes; for each use of the word "object", "thing" or "user" in that extract, underline the word in *blue* if you think that the authors really meant object; underline the word in *red* if you think that they really meant class.

See "What would you think of this?", page 166.

E7.2 *POINT* as an abstract data type

Write an abstract data type specification for the notion of two-dimensional point, as suggested in the informal introduction of that notion.

E7.3 Completing *POINT*

Complete the text of class *POINT* by filling in the missing details and adding a procedure *rotate* (to rotate a point around the origin) as well as any other feature that you feel is necessary.

Page 176.

E7.4 Polar coordinates

Write the text of class *POINT* so as to use a polar, rather than cartesian, representation.