6

Abstract data types

This opened my mind, I started to grasp what it means to use the tool known as algebra. I'll be damned if anyone had ever told me before: over and again Mr. Dupuy [the mathematics teacher] was making pompous sentences on the subject, but not once would he say this simple word: it is a ***division of labor****, which like any division of labor produces miracles, and allows the mind to concentrate all of its forces on just one side of objects, on just one of their qualities.*

What a difference it would have made for us if Mr. Dupuy had told us: This cheese is soft or it is hard; it is white, it is blue; it is old, it is young; it is yours, it is mine, it is light or it is heavy. Of so many qualities let us consider only the weight. Whatever that weight may be, let us call it A. Now, without thinking of the weight any more, let us apply to A everything that we know of quantities.

Such a simple thing; yet no one was saying it to us in that faraway province…

Stendhal, *The Life of Henry Brulard*, 1836.

For abstraction consists only in separating the perceptible qualities of bodies, either from other qualities, or from the bodies to which they apply. Errors arise when this separation is poorly done or wrongly applied: poorly done in philosophical questions, and wrongly applied in physical and mathematical questions. An almost sure way to err in philosophy is to fail to simplify enough the objects under study; and an infallible way to obtain defective results in physics and mathematics is to view the objects as less composite than they are.

Denis Diderot, *A Letter on the Blind for the Benefit of Those Who Can See*, 1749.

Letting objects play the lead role in our software architectures requires that we describe them adequately. This chapter shows how.

You are perhaps impatient to dive into the depths of object technology and explore the details of multiple inheritance, dynamic binding and other joys; then you may at first look at this chapter as an undue delay since it is mostly devoted to the study of some mathematical concepts (although all the mathematics involved is elementary).

But in the same way that even the most gifted musician will benefit from learning a little music theory, knowing about abstract data types will help you understand and enjoy the practice of object-oriented analysis, design and programming, however attractive the concepts might already appear without the help of the theory. Since abstract data types establish the theoretical basis for the entire method, the consequences of the ideas introduced in this chapter will be felt throughout the rest of this book.

There is more. As we will see at chapter end, these consequences actually extend beyond the study of software proper, yielding a few principles of intellectual investigation which one may perhaps apply to other disciplines.

6.1 CRITERIA

To obtain proper descriptions of objects, we need a method satisfying three conditions:

- The descriptions should be precise and unambiguous.
- They should be complete — or at least as complete as we want them in each case (we may decide to leave some details out).
- They should not be **overspecifying**.

The last point is what makes the answer non-trivial. It is after all easy to be precise, unambiguous and complete if we "spill the beans" by giving out all the details of the objects' representation. But this is usually *too much* information for the authors of software elements that need to access the objects.

This observation is close to the comments that led to the notion of information hiding. The concern there was that by providing a module's source code (or, more generally, implementation-related elements) as the primary source of information for the authors of software elements that rely on that module, we may drown them in a flood of details, prevent them from concentrating on their own job, and hamper prospects of smooth evolution. Here the danger is the same if we let modules use a certain data structure on the basis of information that pertains to the structure's representation rather than to its essential properties.

"Information Hiding", page 51.

6.2 IMPLEMENTATION VARIATIONS

To understand better why the need for abstract data descriptions is so crucial, let us explore further the potential consequences of using physical representation as the basis for describing objects.

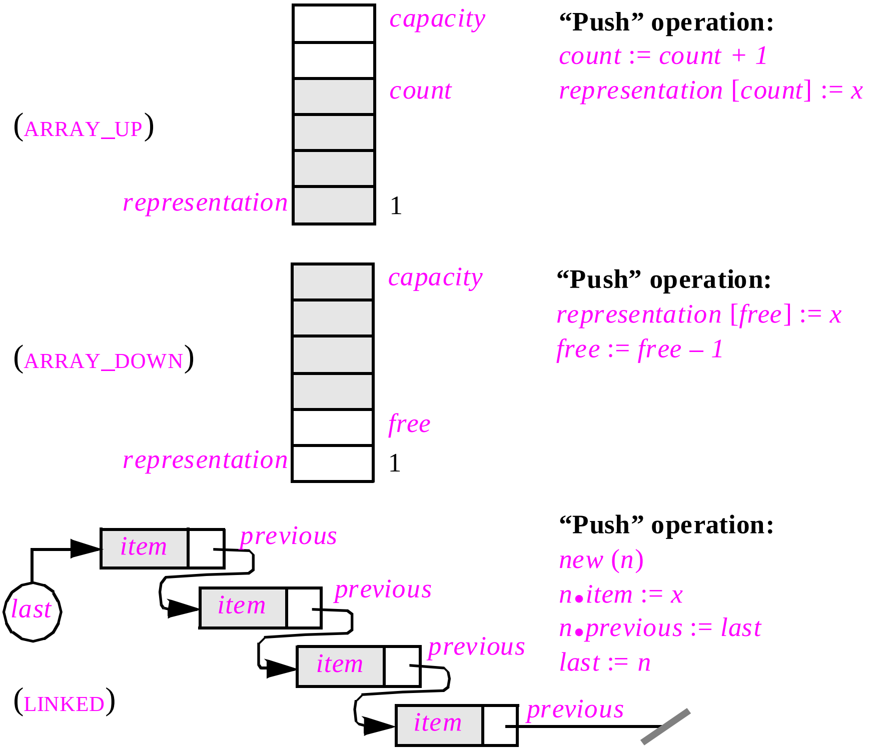
A well-known and convenient example is the description of stack objects. A stack object serves to pile up and retrieve other objects in a last-in, first-out ("LIFO") manner, the latest inserted element being the first one to be retrieved. The stack is a ubiquitous structure in computing science and in many software systems; the typical compiler or interpreter, for example, is peppered with stacks of many kinds.

Stacks, it must be said, are also ubiquitous in didactic presentations of abstract data types, so much so that Edsger Dijkstra is said to have once quipped that "abstract data types are a remarkable theory, whose purpose is to describe stacks". Fair enough. But the notion of abstract data type applies to so many more advanced cases in the rest of this book that I do not feel ashamed of starting with this staple example. It is the simplest I know which includes about every important idea about abstract data types.

Stack representations

Several possible physical representations exist for stacks:

Three possible representations for a stack



"Push" operation:

```
count := count + 1
representation [count] := x
```

"Push" operation:

```
representation [free] := x
free := free – 1
```

"Push" operation:

```
new (n)
n.item := x
n.previous := last
last := n
```

The figure illustrates three of the most common representations. Each has been given a name for ease of reference:

- ARRAY_UP: represent a stack through an array *representation* and an integer *count* whose value ranges from 0 (for an empty stack) to *capacity*, the size of the array *representation*; stack elements are stored in the array at indices 1 up to *count*.

- ARRAY_DOWN: like ARRAY_UP, but with elements stored from the end of the array rather than from the beginning. Here the integer is called *free* (it is the index of the highest free array position, or 0 if all positions are occupied) and ranges from *capacity* for an empty stack down to 0. The stack elements are stored in the array at indices *capacity* down to *free + 1*.

- LINKED: a linked representation which stores each stack element in a cell with two fields: *item* representing the element, and *previous* containing a pointer to the cell containing the previously pushed element. The representation also needs *last*, a pointer to the cell representing the top.

Next to each representation, the figure shows a program extract (in Pascal-like notation) giving the corresponding implementation for a basic stack operation: pushing an element *x* onto the top.

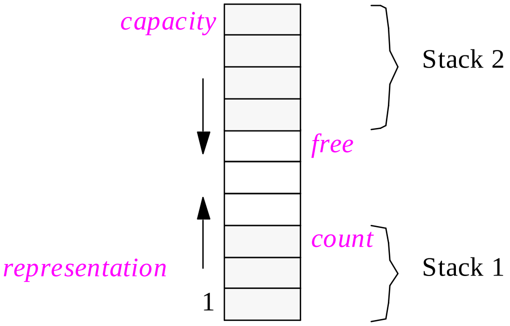
For the array representations, ARRAY_UP and ARRAY_DOWN, the instructions increase or decrease the top indicator (*count* or *free*) and assign *x* to the corresponding array element. Since these representations support stacks of at most *capacity* elements, robust implementations should include guards of the respective forms

```
if count < capacity then …
if free > 0 then …
```

which the figure omits for simplicity.

For LINKED, the linked representation, pushing an element requires four operations: create a new cell *n* (done here with Pascal's *new* procedure, which allocates space for a new object); assign *x* to the new cell's *item* field; chain the new cell to the earlier stack top by assigning to its *previous* field the current value of *last*; and update *last* so that it will now be attached to the newly created cell.

Although these are the most frequently used stack representations, many others exist. For example if you need **two** stacks of elements of the same type, and have only limited space available, you may rely on a single array with two integer top markers, *count* as in ARRAY_UP and *free* as in ARRAY_DOWN; one of the stacks will grow up and the other will grow down. The representation is full if and only if $count = free$.



Head-to-head representation for two stacks

The advantage, of course, is to lessen the risk of running out of space: with two arrays of capacity *n* representing stacks under ARRAY_UP or ARRAY_DOWN, you exhaust the available space whenever *either* stack reaches *n* elements; with a single array of size $2n$ holding two head-to-head stacks, you run out when the *combined* size reaches $2n$, a less likely occurrence if the two stacks grow independently. (For any variable values *p* and *q*, $max\ (p + q) \leq max\ (p) + max\ (q)$.)

Each of these and other possible representations is useful in some cases. Choosing one of them as "the" definition of stacks would be a typical case of overspecification. Why should we consider ARRAY_UP, for example, more representative than LINKED? The most visible properties of ARRAY_UP — the array, the integer *count*, the upper bound — are irrelevant to an understanding of the underlying structure.

The danger of overspecification

Why is it so bad to use a particular representation as specification?

"ABOUT SOFTWARE MAINTENANCE", 1.3, page 17.

The results of the Lientz and Swanson maintenance study, which you may recall, give a hint. More than 17% of software costs was found to come from the need to take into account changes of data formats. As was noted in the discussion, too many programs are closely tied to the physical structure of the data they manipulate. A method relying on the physical representation of data structures to guide analysis and design would not be likely to yield flexible software.

So if we are to use objects or object types as the basis of our system architectures, we should find a better description criterion than the physical representation.

How long is a middle initial?

Lest stacks make us forget that, beyond the examples favored by computer scientists, data structures are ultimately connected with real-life objects, here is an amusing example, taken from a posting on the Risks forum (*comp.risks* Usenet newsgroup) of the dangers of a view of data that is too closely dependent on concrete properties:

Risks forum, 10.74, 3 Jan. 1993. Posting by Darrell D.E. Long: ``Dehumanization by old Cobol programs". Abbreviated.

See exercise E6.5, page 161.

> *My dear mother blessed (or perhaps cursed) all of her children with two middle initials, in my case "D" and "E". This has caused me a good deal of trouble.*
>
> *It seems that TRW sells certain parts of your credit information, such as your name and a demographic profile. I recently got a new credit card from Gottchalks and found to my chagrin that my name had been truncated to "Darrell D. Long". I went to the credit manager and was assured that things would be fixed. Well, two things happened: I got a new credit card, this time as "Darrell E. Long", and TRW now has an annotation in my file to the effect "File variation: middle initial is E". Soon after this I start getting mail for "Darrell E. Long" (along with the usual "Darrell Long" and "Darrell D. Long" and the occasional "Darrell D. E. Long").*
>
> *I called up the credit bureau and it seems that the programmer who coded up the TRW database decided that all good Americans are entitled to only one middle initial. As the woman on the phone patiently told me "They only allocated enough megabytes (sic) in the system for one middle initial, and it would probably be awfully hard to change".*

Aside from the typical example of technobabble justification ("megabytes"), the lesson here is the need to avoid tying software to the exact physical properties of data. TRW's system seems similar to those programs, mentioned in an earlier discussion, which "knew" that postal codes consist of exactly five digits.

See page 18.

The author of the message reproduced above was mainly concerned about junk mail, an unpleasant but not life-threatening event; the archives of the Risks forum are full of computer-originated name confusions with more serious consequences. The "millenium problem", mentioned in the discussion of software maintenance, is another example of the dangers of accessing data based on physical representation, this one with hundreds of millions of dollars' worth of consequences.

6.3 TOWARDS AN ABSTRACT VIEW OF OBJECTS

How do we retain completeness, precision and non-ambiguity without paying the price of overspecification?

Using the operations

In the stack example, what unites the various representations in spite of all their differences is that they describe a "container" structure (a structure used to contain other objects), where certain operations are applicable and enjoy certain properties. By focusing not on a particular choice of representation but on these operations and properties, we may be able to obtain an abstract yet useful characterization of the notion of stack.

The operations typically available on a stack are the following:

- A command to push an element on top of a stack. Let us call that operation *put*.
- A command to remove the stack's top element, if the stack is not empty. Let us call it *remove*.
- A query to find out what the top element is, if the stack is not empty. Let us call it *item*.
- A query to determine whether the stack is empty. (This will enable clients to determine beforehand if they can use *remove* and *item*.)

In addition we may need a creator operation giving us a stack, initially empty. Let us call it *make*.

Two points may have caught your attention and will deserve more explanation later in this chapter. First, the operation names may seem surprising; for the moment, just think of *put* as meaning *push*, *remove* as meaning *pop*, and *item* as meaning *top*. Details shortly (on the facing page, actually). Second, the operations have been divided into three categories: creators, which yield objects; queries, which return information about objects; and commands, which can modify objects. This classification will also require some more comments.

In a traditional view of data structures, we would consider that the notion of stack is given by some data declaration corresponding to one of the above representations, for example (representation ARRAY_UP, Pascal-like syntax):

```
count: INTEGER
representation: array [1 .. capacity] of STACK_ELEMENT_TYPE
```

where *capacity*, a constant integer, is the maximum number of elements on the stack. Then *put*, *remove*, *item*, *empty* and *make* would be routines (subprograms) that work on the object structures defined by these declarations.

The key step towards data abstraction is to reverse the viewpoint: forget for the moment about the representation; take the operations themselves as defining the data structure. In other words, a stack **is** any structure to which clients may apply the operations listed above.

A laissez-faire policy for the society of modules

The method just outlined for describing data structures shows a rather selfish approach to the world of data structures: like an economist of the most passionate supply-side, invisible-hand, let-the-free-market-decide school, we are interested in individual agents not so much for what they *are* internally as for what they *have* to offer to each other. The world of objects (and hence of software architecture) will be a world of interacting agents, communicating on the basis of precisely defined protocols.

The economic analogy will indeed accompany us throughout this presentation; the agents — the software modules — are called *suppliers* and *clients*; the protocols will be called *contracts*, and much of object-oriented design is indeed *Design by Contract*, the title of a later chapter.

See "BEYOND SOFTWARE", 6.6, page 147.

As always with analogies, we should not get too carried away: this work is not a textbook on economics, and contains no hint of its author's views in that field. It will suffice for the moment to note the remarkable analogies of the abstract data type approach to some theories of how human agents should work together. Later in this chapter we will again explore what abstract data types can tell us beyond their original area of application.

Name consistency

For the moment, let us get back to more immediate concerns, and make sure you are comfortable with the above example specification in all its details. If you have encountered stacks before, the operation names chosen for the discussion of stacks may have surprised or even shocked you. Any self-respecting computer scientist will know stack operations under other names:

Common stack operation name	**Name used here**
push	*put*
pop	*remove*
top	*item*
new	*make*

Why use anything else than the traditional terminology? The reason is a desire to take a high-level view of data structures — especially "containers", those data structures used to keep objects.

Stacks are just one brand of container; more precisely, they belong to a category of containers which we may call **dispensers**. A dispenser provides its clients with a mechanism for storing (*put*), retrieving (*item*) and removing (*remove*) objects, but without giving them any control over the choice of object to be stored, retrieved or removed. For example, the LIFO policy of stacks implies that you may only retrieve or remove the element that was stored last. Another brand of dispenser is the queue, which has a first-in, first-out (FIFO) policy: you store at one end, retrieve and remove at the other; the element that you retrieve or remove is the oldest one stored but not yet removed. An example of a container which is **not** a dispenser is an array, where you choose, through integer indices, the positions where you store and retrieve objects.

Because the similarities between various kinds of container (dispensers, arrays and others) are more important than the differences between their individual storage, retrieval and removal properties, this book constantly adheres to a standardized terminology which downplays the differences between data structure variants and instead emphasizes the commonality. So the basic operation to retrieve an element will always be called *item*, the basic operation to remove an element will always be called *remove* and so on.

These naming issues may appear superficial at first — "cosmetic", as programmers sometimes say. But do not forget that one of our eventual aims is to provide the basis for powerful, professional libraries of reusable software components. Such libraries will contain tens of thousands of available operations. Without a systematic and clear nomenclature, both the developers and the users of these libraries would quickly be swamped in a flood of specific and incompatible names, providing a strong (and unjustifiable) obstacle to large-scale reuse.

Naming, then, is *not* cosmetic. Good reusable software is software that provides the right functionality and provides it under the right names.

The names used here for stack operations are part of a systematic set of naming conventions used throughout this book. A later chapter will introduce them in more detail.

Chapter 26, in particular "CHOOSING THE RIGHT NAMES", 26.2, page 879.

How not to handle abstractions

In software engineering as in other scientific and technical disciplines, a seminal idea may seem obvious once you have been exposed to it, even though it may have taken a long time to emerge. The bad ideas and the complicated ones (they are often the same) often appear first; it takes time for the simple and the elegant to take over.

This observation is true of abstract data types. Although good software developers have always (as a result of education or mere instinct) made good use of abstraction, many of the systems in existence today were designed without much consideration of this goal.

I once did a little involuntary experiment which provided a good illustration of this state of affairs. While setting up the project part of a course which I was teaching, I decided to provide students with a sort of anonymous marketplace, where they could place mock "for sale" announcements of software modules, without saying who was the source of the advertisement. (The idea, which may or may not have been a good one, was to favor a selection process based only on a precise specification of the modules' advertized facilities.) The mail facility of a famous operating system commonly favored by universities seemed to provide the right base mechanism (why write a new mail system just for a course project?); but naturally that mail facility shows the sender's name when it delivers a message to its recipients. I had access to the source of the corresponding code — a huge C program — and decided, perhaps foolishly, to take that code, remove all references to the sender's name in delivered messages, and recompile.

Aided by a teaching assistant, I thus embarked on a task which seemed obvious enough although not commonly taught in software engineering courses: systematic program *de*construction. Sure enough, we quickly found the first place where the program accessed the sender's name, and we removed the corresponding code. This, we naïvely thought, would have done the job, so we recompiled and sent a test mail message; but the sender's name was still there! Thus began a long and surreal process: time and again, believing we had finally found the last reference to the sender's name, we would remove it, recompile, and mail a test message, only to find the name duly recorded once again in its habitual field. Like the Hydra in its famous fight, the mailer kept growing a new head every time we thought we had cut the last neck.

Finally, repeating for the modern era the earlier feat of Hercules, we slew the beast for good; by then we had removed more than twenty code extracts which all accessed, in some way or other, information about the message sender.

Writing MAIL_MESSAGE is the topic of exercise E6.4, page 161.

Although the previous sections have only got us barely started on our road to abstract data types, it should be clear by now that any program written in accordance with even the most elementary concepts of data abstraction would treat *MAIL_MESSAGE* as a carefully defined abstract notion, supporting a query operation, perhaps called *sender*, which returns information about the message sender. Any portion of the mail program that needs this information would obtain it solely through the *sender* query. Had the mail program been designed according to this seemingly obvious principle, it would have been sufficient, for the purpose of my little exercise, to modify the code of the *sender* query. Most likely, the software would also then have provided an associated command operation *set_sender* to update sender information, making the job even easier.

What is the real moral of that little story (besides lowering the reader's guard in preparation for the surprise mathematical offensive of the next section)? After all, the mail program in question is successful, at least judging by its widespread use. But it typifies the current quality standard in the industry. Until we move significantly beyond that standard, the phrase "software engineering" will remain a case of wishful thinking.

Oh yes, one more note. Some time after my brief encounter with the mail program, I read that certain network hackers had intruded into the computer systems of highly guarded government laboratories, using a security hole of that very mail program — a hole which was familiar, so the press reported, to all those in the know. I was not in the know; but, when I learned the news, I was not surprised.

6.4 FORMALIZING THE SPECIFICATION

The glimpse of data abstraction presented so far is too informal to be of durable use. Consider again our staple example: a stack, as we now understand it, is defined in terms of the applicable operations; but then we need to define these operations!

Informal descriptions as above (*put* pushes an element "on top of" the stack, *remove* pops the element "last pushed" and so on) do not suffice. We need to know precisely how these operations can be used by clients, and what they will do for them.

An abstract data type specification will provide this information. It consists of four paragraphs, explained in the next sections:

- TYPES.
- FUNCTIONS.
- AXIOMS.
- PRECONDITIONS.

These paragraphs will rely on a simple mathematical notation for specifying the properties of an abstract data type (ADT for short).

> The notation — a mathematical formalism, not to be confused with the software notation of the rest of this book even though for consistency it uses a similar syntactic style — has no name and is not a programming language; it could serve as the starting point for a formal *specification* language, but we shall not pursue this avenue here, being content enough to use self-explanatory conventions for the unambiguous specification of abstract data types.

Specifying types

The TYPES paragraph indicates the types being specified. In general, it may be convenient to specify several ADTs together, although our example has only one, *STACK*.

By the way, what is a type? The answer to this question will combine all the ideas developed in the rest of this chapter; a type is a collection of objects characterized by functions, axioms and preconditions. If for the moment you just view a type as a set of objects, in the mathematical sense of the word "set" — type *STACK* as the set of all possible stacks, type *INTEGER* as the set of all possible integer values and so on — you are not guilty of any terrible misunderstanding. As you read this discussion you will be able to refine this view. In the meantime the discussion will not be too fussy about using "set" for "type" and conversely.

On one point, however, you should make sure to avoid any confusion: an abstract data type such as *STACK* is not an object (one particular stack) but a collection of objects (the set of all stacks). Remember what our real goal is: finding a good basis for the modules of our software systems. As was noted in the previous chapter, basing a module on one particular object — one stack, one airplane, one bank account — would not make sense. O-O design will enable us to build modules covering the properties of all stacks, all airplanes, all bank accounts — or at least of some stacks, airplanes or accounts.

An object belonging to the set of objects described by an ADT specification is called an **instance** of the ADT. For example, a specific stack which satisfies the properties of the *STACK* abstract data type will be an instance of *STACK*. The notion of instance will carry over to object-oriented design and programming, where it will play an important role in explaining the run-time behavior of programs.

The TYPES paragraph simply lists the types introduced in the specification. Here:

TYPES

- *STACK* [*G*]

Our specification is about a single abstract data type *STACK*, describing stacks of objects of an arbitrary type *G*.

Genericity

See “Genericity”, page 96.

In *STACK* [*G*], *G* denotes an arbitrary, unspecified type. *G* is called a **formal generic parameter** of the abstract data type *STACK*, and *STACK* itself is said to be a generic ADT. The mechanism permitting such parameterized specifications is known as genericity; we already encountered a similar concept in our review of package constructs.

It is possible to write ADT specifications without genericity, but at the price of unjustified repetition. Why have separate specifications for the types “stack of bank accounts”, “stack of integers” and so on? These specifications would be identical except where they explicitly refer to the type of the stack elements — bank accounts or integers. Writing them, and then performing the type substitutions manually, would be tedious. Reusability is desirable for specifications too — not just programs! Thanks to genericity, we can make the type parameterization explicit by choosing some arbitrary name, here *G*, to represent the variable type of stack elements.

As a result, an ADT such as *STACK* is not quite a type, but rather a type pattern; to obtain a directly usable stack type, you must obtain some element type, for example *ACCOUNT*, and provide it as **actual generic parameter** corresponding to the formal parameter *G*. So although *STACK* is by itself just a type pattern, the notation

STACK [*ACCOUNT*]

is a fully defined type. Such a type, obtained by providing actual generic parameters to a generic type, is said to be **generically derived**.

The notions just seen are applicable recursively: every type should, at least in principle, have an ADT specification, so you may view *ACCOUNT* as being itself an abstract data type; also, a type that you use as actual generic parameter to *STACK* (to produce a generically derived type) may itself be generically derived, so it is perfectly all right to use

STACK [*STACK* [*ACCOUNT*]]

specifying a certain abstract data type: the instances of that type are stacks, whose elements are themselves stacks; the elements of these latter stacks are bank accounts.

As this example shows, the preceding definition of “instance” needs some qualification. Strictly speaking, a particular stack is an instance not of *STACK* (which, as noted, is a type pattern rather than a type) but of some type generically derived from *STACK*, for example *STACK* [*ACCOUNT*]. It is convenient, however, to continue talking about instances of *STACK* and similar type patterns, with the understanding that this actually means instances of their generic derivations.

Similarly, it is not quite accurate to talk about *STACK* being an ADT: the correct term is "ADT pattern". For simplicity, this discussion will continue omitting the word "pattern" when there is no risk of confusion.

> The distinction will carry over to object-oriented design and programming, but there we will need to keep two separate terms:
>
> - The basic notion will be the **class**; a class may have generic parameters.
> - Describing actual data requires **types**. A non-generic class is also a type, but a generic class is only a type pattern. To obtain an actual type from a generic class, we will need to provide actual generic parameters, exactly as we derive the ADT *STACK* [*ACCOUNT*] from the ADT pattern *STACK*.
>
> Later chapters will explore the notion of genericity as applied to classes, and how to combine it with the inheritance mechanism.

Chapter 10 and appendix B.

Listing the functions

After the TYPES paragraph comes the FUNCTIONS paragraph, which lists the operations applicable to instances of the ADT. As announced, these operations will be the prime component of the type definition — describing its instances not by what they are but by what they have to offer.

Below is the FUNCTIONS paragraph for the *STACK* abstract data type. If you are a software developer, you will find the style familiar: the lines of such a paragraph evoke the **declarations** found in typed programming languages such as Pascal or Ada. The line for *new* resembles a variable declaration; the others resemble routine headers.

FUNCTIONS

- $put: STACK\ [G] \times G \rightarrow STACK\ [G]$
- $remove: STACK\ [G] \nrightarrow STACK\ [G]$
- $item: STACK\ [G] \nrightarrow G$
- $empty: STACK\ [G] \rightarrow BOOLEAN$
- $new: STACK\ [G]$

Each line introduces a mathematical function modeling one of the operations on stacks. For example function *put* represents the operation that pushes an element onto the top of a stack.

Why functions? Most software people will not naturally think of an operation such as *put* as a function. When the execution of a software system applies a *put* operation to a stack, it will usually modify that stack by adding an element to it. As a result, in the above informal classification of commands, *put* was a "command" — an operation which may modify objects. (The other two categories of operations were creators and queries).

See also "The imperative and the applicative", page 351.

An ADT specification, however, is a mathematical model, and must rely on well-understood mathematical techniques. In mathematics the notion of command, or more generally of changing something, does not exist as such; computing the square root of the number 2 does not modify the value of that number. A mathematical expression simply defines certain mathematical objects in terms of certain other mathematical objects: unlike the execution of software on a computer, it never changes any mathematical object.

Yet we need a mathematical concept to model computer operations, and here the notion of function yields the closest approximation. A function is a mechanism for obtaining a certain result, belonging to a certain target set, from any possible input belonging to a certain source set. For example, if **R** denotes the set of real numbers, the function definition

$$square\_plus\_one: \mathbf{R} \rightarrow \mathbf{R}$$
$$square\_plus\_one\ (x) = x^2 + 1 \qquad \text{(for any } x \text{ in } \mathbf{R}\text{)}$$

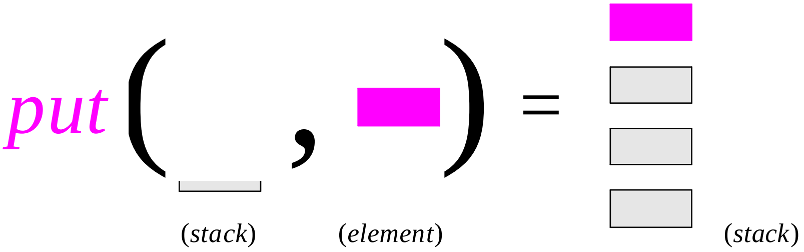
introduces a function *square_plus_one* having **R** as both source and target sets, and yielding as result, for any input, the square of the input plus one.

The specification of abstract data types uses exactly the same notion. Operation *put*, for example, is specified as

$$put: STACK\ [G] \times G \rightarrow STACK\ [G]$$

which means that *put* will take two arguments, a *STACK* of instances of *G* and an instance of *G*, and yield as a result a new *STACK* [*G*]. (More formally, the source set of function *put* is the set *STACK* [*G*] × *G*, known as the **cartesian product** of *STACK* [*G*] and *G*; this is the set of pairs <*s*, *x*> whose first element *s* is in *STACK* [*G*] and whose second element *x* is in *G*.) Here is an informal illustration:

Applying the put function



With abstract data types, we only have functions in the mathematical sense of the term; they will produce neither side effects nor in fact changes of any kind. This is the condition that we must observe to enjoy the benefits of mathematical reasoning.

When we leave the ethereal realm of specification for the rough-and-tumble of software design and implementation, we will need to reintroduce the notion of change; because of the performance overhead, few people would accept a software execution environment where every "push" operation on a stack begins by duplicating the stack. Later we will examine the details of the transition from the change-free world of ADTs to the change-full world of software development. For the moment, since we are studying how best to specify types, the mathematical view is the appropriate one.

The role of the operations modeled by each of the functions in the specification of *STACK* is clear from the previous discussion:

- Function *put* yields a new stack with one extra element pushed on top. The figure on the preceding page illustrates $put\ (s, x)$ for a stack s and an element x.
- Function *remove* yields a new stack with the top element, if any, popped; like *put*, this function should yield a command (an object-changing operation, typically implemented as a procedure) at design and implementation time. We will see below how to take into account the case of an empty stack, which has no top to be popped.
- Function *item* yields the top element, if any.
- Function *empty* indicates whether a stack is empty; its result is a boolean value (true or false); the ADT *BOOLEAN* is assumed to have been defined separately.
- Function *new* yields an empty stack.

The FUNCTIONS paragraph does not fully define these functions; it only introduces their **signatures** — the list of their argument and result types. The signature of *put* is

$$STACK\ [G] \times G \rightarrow STACK\ [G]$$

indicating that *put* accepts as arguments pairs of the form $<s, x>$ where s is an instance of $STACK\ [G]$ and x is an instance of G, and yields as a result an instance of $STACK\ [G]$. In principle the target set of a function (the type that appears to the right of the arrow in signature, here $STACK\ [G]$) may itself be a cartesian product; this can be used to describe operations that return two or more results. For simplicity, however, this book will only use single-result functions.

The signature of functions *remove* and *item* includes a crossed arrow $\nrightarrow$ instead of the standard arrow used by *put* and *empty*. This notation expresses that the functions are not applicable to all members of the source set; it will be explained in detail below.

The declaration for function *new* appears as just

$$new\colon STACK$$

with no arrow in the signature. This is in fact an abbreviation for

$$new\colon \rightarrow STACK$$

introducing a function with no arguments. There is no need for arguments since *new* must always return the same result, an empty stack. So we just remove the arrow for simplicity. The result of applying the function (that is to say, the empty stack) will also be written *new*, an abbreviation for $new\ ()$, meaning the result of applying *new* to an empty argument list.

Function categories

The operations on a type were classified informally at the beginning of this chapter into creators, queries and commands. With an ADT specification for a new type T, such as $STACK\ [G]$ in the example, we can define the corresponding classification in a more rigorous way. The classification simply examines where *T* appears, relative to the arrow, in the signature of each function:

- A function such as *new* for which *T* appears only to the right of the arrow is a **creator function**. It models an operation which produces instances of *T* from instances of other types — or, as in the case of a constant creator function such as *new*, from no argument at all. (Remember that the signature of *new* is considered to contain an implicit arrow.)
- A function such as *item* and *empty* for which *T* appears only on the left of the arrow is a **query function**. It models an operation which yields properties of instances of *T*, expressed in terms of instances of other types (*BOOLEAN* and the generic parameter *G* in the examples).
- A function such as *put* or *remove* for which *T* appears on both sides of the arrow is a **command function**. It models an operation which yields new instances of *T* from existing instances of *T* (and possibly instances of other types).

An alternative terminology calls the three categories "constructor", "accessor" and "modifier". The terms retained here are more directly related to the interpretation of ADT functions as models of operations on software objects, and will carry over to class features, the software counterparts of our mathematical functions.

The AXIOMS paragraph

We have seen how to describe a data type such as *STACK* through the list of functions applicable to its instances. The functions are known only through their signatures.

To indicate that we have a stack, and not some other data structure, the ADT specification as given so far is not enough. Any "dispenser" structure, such as a first-in-first-out queue, will also satisfy it. The choice of names for the operations makes this particularly clear: we do not even have stack-specific names such as *push*, *pop* or *top* to fool ourselves into believing that we have defined stacks and only stacks.

This is not surprising, of course, since the FUNCTIONS paragraph declared the functions (in the same way that a program unit may declare a variable) but did not fully define them. In a mathematical definition such as the earlier example

$$square\_plus\_one: \mathbf{R} \rightarrow \mathbf{R}$$

$$square\_plus\_one\ (x) = x^2 + 1 \qquad \text{(for any } x \text{ in } \mathbf{R}\text{)}$$

the first line plays the role of the signature declaration, but there is also a second line which defines the function's value. How do we achieve the same for the functions of an ADT?

Here we should not use an explicit definition in the style of the second line of *square_plus_one*'s definition, because it would force us to choose a representation — and this whole discussion is intended to protect us from representation choices.

Just to make sure we understand what an explicit definition would look like, let us write one for the stack representation ARRAY_UP as sketched above. In mathematical terms, choosing ARRAY_UP means that we consider any instance of *STACK* as a pair

<count, representation>, where *representation* is the array and *count* is the number of pushed elements. Then an explicit definition of *put* is (for any instance *x* of *G*):

$$put\ (<count,\ representation>,\ x)\ =\ <count\ +\ 1,\ representation\ [count+1:\ x]>$$

where the notation *a* [*n*: *v*] denotes the array obtained from *a* by changing the value of the element at index *n* so that it is now *v*, and keeping all other elements, if any, as they are.

This definition of function *put* is just a mathematical version of the implementation of the *put* operation sketched in Pascal notation, next to representation ARRAY_UP, in the picture of possible stack representations at the beginning of this chapter.

Figure page 123.

But this is not what we want; "Free us from the yoke of representations!", the motto of the Object Liberation Front and its military branch (the ADT brigade), is also ours.

The political branch specializes in class-action suits.

Because any explicit definition would force us to select a representation, we must turn to **implicit** definitions. We will refrain from giving the values of the functions of an ADT specification; instead we will state properties of these values — all the properties that matter, but those properties only.

The AXIOMS paragraph states these properties. For *STACK* it will be:

AXIOMS

For any *x*: *G*, *s*: *STACK* [*G*],

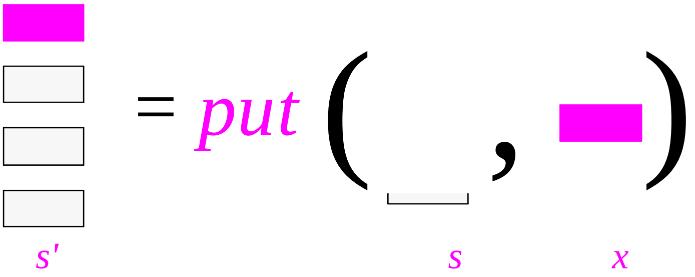
A1 • *item* (*put* (*s*, *x*)) = *x*

A2 • *remove* (*put* (*s*, *x*)) = *s*

A3 • *empty* (*new*)

A4 • **not** *empty* (*put* (*s*, *x*))

The first two axioms express the basic LIFO (last-in, first-out) property of stacks. To understand them, assume we have a stack *s* and an instance *x*, and define *s'* to be *put* (*s*, *x*), that is to say the result of pushing *x* onto *s*. Adapting an earlier figure:



Applying the** put **function

Here axiom A1 tells us that the top of *s'* is *x*, the last element that we pushed; and axiom A2 tells us that if we remove the top element from *s'*, we get back the stack *s* that we had before pushing *x*. These two axioms provide a concise description of the fundamental property of stacks in pure mathematical terms, without any recourse to imperative reasoning or representation properties.

Axioms A3 and A4 tell us when a stack is empty and when it is not: a stack resulting from the creator function *new* is empty; any stack resulting from pushing an element on an existing stack (empty or not) is non-empty.

These axioms, like the others, are predicates (in the sense of logic), expressing that a certain property is always true for every possible value of *s* and *x*. Some people prefer to read A3 and A4 in the equivalent form

For any *x*: *G*, *s*: *STACK* [*G*]

A3' • *empty* (*new*) = **true**

A4' • *empty* (*put* (*s*, *x*)) = **false**

under which you may also view them, informally at least, as defining function *empty* by induction on the size of stacks.

Two or three things we know about stacks

ADT specifications are **implicit**. We have encountered two forms of implicitness:

- The ADT method defines a set of objects implicitly, through the applicable functions. This was described above as defining objects by what they have, not what they are. More precisely, the definition never implies that the operations listed are the only ones; when it comes to a representation, you will often add other operations.
- The functions themselves are also defined implicitly: instead of explicit definitions (such as was used for *square_plus_one*, and for the early attempt to define *put* by reference to a mathematical representation), we use axioms describing the functions' properties. Here too there is no claim of exhaustiveness: when you eventually implement the functions, they will certainly acquire more properties.

This implicitness is a key aspect of abstract data types and, by implication, of their future counterparts in object-oriented software construction — classes. When we define an abstract data type or a class, we always talk *about* the type or class: we simply list the properties we know, and take these as the definition. Never do we imply that these are the only applicable properties.

Implicitness implies openness: it should always be possible to add new properties to an ADT or a class. The basic mechanism for performing such extensions without damaging existing uses of the original form is inheritance.

"More on implicitness", page 149.

The consequences of this implicit approach are far-reaching. The "supplementary topics" section at the end of this chapter will include more comments about implicitness.

Partial functions

The specification of any realistic example, even one as basic as stacks, is bound to encounter the problems of undefined operations: some operations are not applicable to every possible element of their source sets. Here this is the case with *remove* and *item*: you cannot pop an element from an empty stack; and an empty stack has no top.

The solution used in the preceding specification is to describe these functions as partial. A function from a source set X to a target set Y is partial if it is not defined for all members of X. A function which is not partial is **total**. A simple example of partial function in standard mathematics is *inv*, the inverse function on real numbers, whose value for any appropriate real number x is

$$inv\,(x) = \frac{1}{x}$$

Because *inv* is not defined for $x = 0$, we may specify it as a partial function on **R**, the set of all real numbers:

$$inv\colon \mathbf{R} \nrightarrow \mathbf{R}$$

To indicate that a function may be partial, the notation uses the crossed arrow $\nrightarrow$; the normal arrow $\rightarrow$ will be reserved for functions which are guaranteed to be total.

The **domain** of a partial function in $X \nrightarrow Y$ is the subset of X containing those elements for which the function yields a value. Here the domain of *inv* is $\mathbf{R} - \{0\}$, the set of real numbers other than zero.

The specification of the *STACK* ADT applied these ideas to stacks by declaring *put* and *item* as partial functions in the FUNCTIONS paragraph, as indicated by the crossed arrow in their signatures. This raises a new problem, discussed in the next section: how to specify the domains of these functions.

In some cases it may be desirable to describe *put* as a partial function too; this is necessary to model implementations such as ARRAY_UP and ARRAY_DOWN, which only support a finite number of consecutive *put* operations on any given stack. It is indeed a good exercise to adapt the specification of *STACK* so that it will describe bounded stacks with a finite capacity, whereas the above form does not include any such capacity restriction. This is a new use for partial functions: to reflect implementation constraints. In contrast, the need to declare *item* and *remove* as partial functions reflected an abstract property of the underlying operations, applicable to all representations.

Exercise E6.9, page 162.

Preconditions

Partial functions are an inescapable fact of software development life, merely reflecting the observation that not every operation is applicable to every object. But they are also a potential source of errors: if f is a partial function from X to Y, we are not sure any more that the expression $f\,(e)$ makes sense even if the value of e is in X: we must be able to guarantee that the value belongs to the domain of f.

For this to be possible, any ADT specification which includes partial functions must specify the domain of each of them. This is the role of the PRECONDITIONS paragraph.

For *STACK*, the paragraph will appear as:

PRECONDITIONS

- *remove* (*s*: *STACK* [*G*]) **require not** *empty* (*s*)
- *item* (*s*: *STACK* [*G*]) **require not** *empty* (*s*)

where, for each function, the **require** clause indicates what conditions the function's arguments must satisfy to belong to the function's domain.

The boolean expression which defines the domain is called the **precondition** of the corresponding partial function. Here the precondition of both *remove* and *item* expresses that the stack argument must be non-empty. Before the **require** clause comes the name of the function with dummy names for arguments (*s* for the stack argument in the example), so that the precondition can refer to them.

> Mathematically, the precondition of a function *f* is the **characteristic function** of the domain of *f*. The characteristic function of a subset *A* of a set *X* is the total function *ch*: *X* $\rightarrow$ *BOOLEAN* such that *ch* (*x*) is true if *x* belongs to *A*, false otherwise.

The complete specification

The PRECONDITIONS paragraph concludes this simple specification of the *STACK* abstract data type. For ease of reference it is useful to piece together the various components of the specification, seen separately above. Here is the full specification:

ADT specification of stacks

TYPES

- *STACK* [*G*]

FUNCTIONS

- *put*: *STACK* [*G*] $\times$ *G* $\rightarrow$ *STACK* [*G*]
- *remove*: *STACK* [*G*] $\nrightarrow$ *STACK* [*G*]
- *item*: *STACK* [*G*] $\nrightarrow$ *G*
- *empty*: *STACK* [*G*] $\rightarrow$ *BOOLEAN*
- *new*: *STACK* [*G*]

AXIOMS

For any *x*: *G*, *s*: *STACK* [*G*]

A1 • *item* (*put* (*s*, *x*)) = *x*

A2 • *remove* (*put* (*s*, *x*)) = *s*

A3 • *empty* (*new*)

A4 • **not** *empty* (*put* (*s*, *x*))

PRECONDITIONS

- *remove* (*s*: *STACK* [*G*]) **require not** *empty* (*s*)
- *item* (*s*: *STACK* [*G*]) **require not** *empty* (*s*)

Nothing but the truth

The power of abstract data type specifications comes from their ability to capture the essential properties of data structures without overspecifying. The stack specification collected on the preceding page expresses all there is to know about the notion of stack in general, excluding anything that only applies to some particular representations of stacks. All the truth about stacks; yet nothing but the truth.

This provides a general model of computation with data structures. We may describe complex sequences of operations by mathematical expressions enjoying the usual properties of algebra; and we may view the process of carrying out the computation (executing the program) as a case of algebraic simplification.

In elementary mathematics we have been taught to take an expression such as

$$cos^2\,(a - b) + sin^2\,(a + b - 2 \times b)$$

and apply the rules of algebra and trigonometry to simplify it. A rule of algebra tells us that we may simplify $a + b - 2 \times b$ into $a - b$ for any a and b; and a rule of trigonometry tells us that we can simplify $cos^2\,(x) + sin^2\,(x)$ into 1 for any x. Such rules may be combined; for example the combination of the two preceding rules allow us to simplify the above expression into just 1.

In a similar way, the functions defined in an abstract data type specification allow us to construct possibly complex expressions; and the axioms of the ADT allow us to simplify such expressions to yield a simpler result. A complex stack expression is the mathematical equivalent of a program; the simplification process is the mathematical equivalent of a computation, that is to say, of executing such a program.

Here is an example. With the specification of the *STACK* abstract data type as given above, we can write the expression

$$\begin{array}{l} item\ (remove\ (put\ (remove\ (put\ (put\ (\\ \qquad remove\ (put\ (put\ (put\ (new, x1), x2), x3)), \\ \qquad item\ (remove\ (put\ (put\ (new, x4), x5)))), x6)), x7))) \end{array}$$

Let us call this expression *stackexp* for future reference. It is perhaps easier to understand *stackexp* if we define it in terms of a sequence of auxiliary expressions:

$$\begin{array}{l} s1 = new \\ s2 = put\ (put\ (put\ (s1, x1), x2), x3) \\ s3 = remove\ (s2) \\ s4 = new \\ s5 = put\ (put\ (s4, x4), x5) \\ s6 = remove\ (s5) \\ y1 = item\ (s6) \\ s7 = put\ (s3, y1) \\ s8 = put\ (s7, x6) \\ s9 = remove\ (s8) \end{array}$$

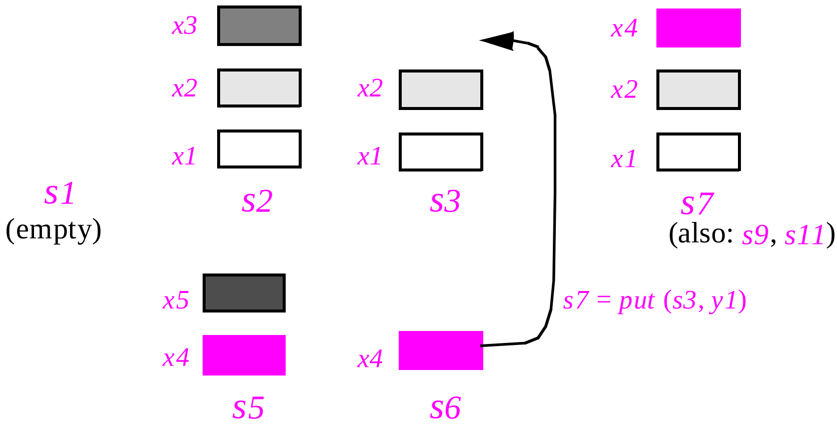
$s10 = put\ (s9, x7)$

$s11 = remove\ (s10)$

$stackexp = item\ (s11)$

Whichever variant of the definition you choose, it is not hard to follow the computation of which $stackexp$ is a mathematical model: create a new stack; push elements $x1$, $x2$, $x3$, in this order, on top of it; remove the last pushed element ($x3$), calling $s3$ the resulting stack; create another empty stack; and so on. Or you can think of it graphically:

Stack manipulations



You can easily find the value of such an ADT expression by drawing figures such as the above. (Here you would find $x4$.) But the theory enables you to obtain this result formally, without any need for pictures: just apply the axioms repeatedly to simplify the expression until you cannot simplify any further. For example:

- Applying A2 to simplify $s3$, that is to say $remove\ (put\ (put\ (put\ (s1, x1), x2), x3))$, yields $put\ (put\ (s1, x1), x2))$. (With A2, any consecutive $remove$-put pair cancels out.)
- The same axiom indicates that $s6$ is $put\ (s4, x4)$; then we can use axiom A1 to deduce that $y1$, that is to say $item\ (put\ (s4, x4))$, is in fact $x4$, showing that (as illustrated by the arrow on the above figure) $s7$ is obtained by pushing $x4$ on top of $s3$.

And so on. A sequence of such simplifications, carried out as simply and mechanically as the simplifications of elementary arithmetic, yields the value of the expression $stackexp$, which (as you are invited to check for yourself by performing the simplification process rigorously) is indeed $x4$.

This example gives a glimpse of one of the main theoretical roles of abstract data types: providing a formal model for the notion of program and program execution. This model is purely mathematical: it has none of the imperative notions of program state, variables whose values may change in time, or execution sequencing. It relies on the standard expression evaluation techniques of ordinary mathematics.

6.5 FROM ABSTRACT DATA TYPES TO CLASSES

We have the starting point of an elegant mathematical theory for modeling data structures and in fact, as we just saw, programs in general. But our subject is software architecture, not mathematics or even theoretical computing science! Have we strayed from our path?

Not by much. In the search for a good modular structure based on object types, abstract data types provide a high-level description mechanism, free of implementation concerns. They will lead us to the fundamental structures of object technology.

Classes

ADTs will serve as the direct basis for the modules that we need in the search begun in chapter *3*. More precisely, an object-oriented system will be built (at the level of analysis, design or implementation) as a collection of interacting ADTs, partially or totally implemented. The basic notion here is **class**:

Definition: class

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

So to obtain a class we must provide an ADT and decide on an implementation. The ADT is a mathematical concept; the implementation is its computer-oriented version. The definition, however, states that the implementation may be partial; the following terminology separates this case from that of a fully implemented class:

Definition: deferred, effective class

A class which is fully implemented is said to be **effective**. A class which is implemented only partially, or not at all, is said to be **deferred**. Any class is either deferred or effective.

To obtain an effective class, you must provide all the implementation details. For a deferred class, you may choose a certain style of implementation but leave some aspects of the implementation open. In the most extreme case of “partial” implementation you may refrain from making any implementation decision at all; the resulting class will be fully deferred, and equivalent to an ADT.

How to produce an effective class

Consider first the case of effective classes. What does it take to implement an ADT? Three kinds of element will make up the resulting effective class:

E1 • An ADT specification (a set of functions with the associated axioms and preconditions, describing the functions' properties).

E2 • A choice of representation.

E3 • A mapping from the functions (*E1*) to the representation (*E2*) in the form of a set of mechanisms, or **features**, each implementing one of the functions in terms of the representation, so as to satisfy the axioms and preconditions. Many of these features will be routines (subprograms) in the usual sense, although some may also appear as data fields, or "attributes", as explained in the next chapters.

For example, if the ADT is *STACK*, we may choose as representation (step *E2*) the solution called ARRAY_UP above, which implements any stack by a pair

<representation, count>

where *representation* is an array and *count* an integer. For the function implementations (*E3*) we will have features corresponding to *put*, *remove*, *item*, *empty* and *new*, which achieve the corresponding effects; for example we may implement *put* by a routine of the form

```
put (x: G) is
            -- Push x onto stack.
            -- (No check for possible stack overflow.)
      do
            count := count + 1
            representation [count] := x
      end
```

The combination of elements obtained under *E1*, *E2* and *E3* will yield a class, the modular structure of object technology.

The role of deferred classes

For an effective class, all of the implementation information (*E2*, *E3* above) must be present. If any of it is missing, the class is deferred.

The more deferred a class, the closer it is to an ADT, gussied up in the kind of syntactic dress that will help seduce software developers rather than mathematicians. Deferred classes are particularly useful for analysis and for design:

- In object-oriented analysis, no implementation details are needed or desired: the method uses classes only for their descriptive power.
- In object-oriented design, many aspects of the implementation will be left out; instead, a design should concentrate on high-level architectural properties of the system — what functionalities each module provides, not how it provides them.
- As you move your design gradually closer to a full implementation, you will add more and more implementation properties until you get effective classes.

But the role of deferred classes does not stop there, and even in a fully implemented system you will often find many of them. Some of that role follows from their previous applications: if you started from deferred classes to obtain effective ones, you may be well inspired to keep the former as ancestors (in the sense of inheritance) to the latter, to serve as a living memory of the analysis and design process.

Too often, in software produced with non-object-oriented approaches, the final form of a system contains no record of the considerable effort that led to it. For someone who is asked to perform maintenance — extensions, ports, debugging — on the system, trying to understand it without that record is as difficult as it would be, for a geologist, to understand a landscape without having access to the sedimentary layers. Keeping the deferred classes in the final system is one of the best ways to maintain the needed record.

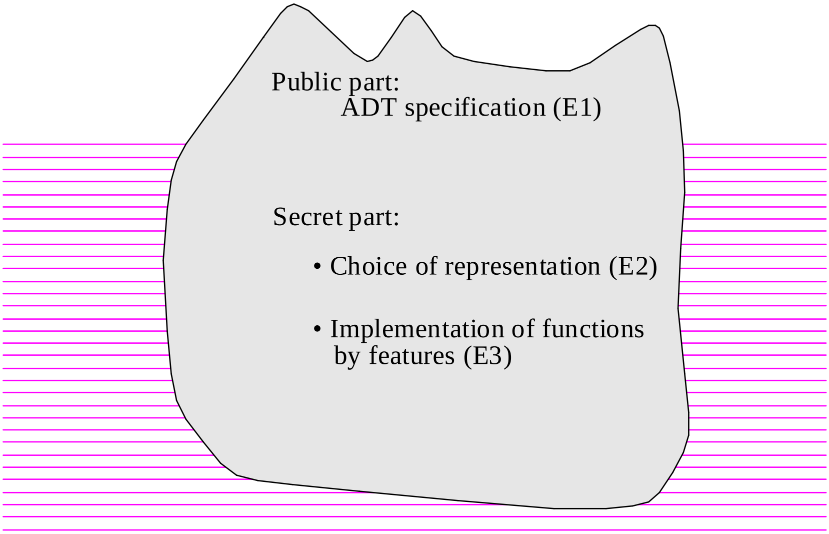
Deferred classes also have purely implementation-related uses. They serve to classify groups of related types of objects, provide some of the most important high-level reusable modules, capture common behaviors among a set of variants, and play a key role (in connection with polymorphism and dynamic binding) in guaranteeing that the software architecture remains decentralized and extendible.

The next few chapters, which introduce the basic object-oriented techniques, will at first concentrate on effective classes. But keep in mind the notion of deferred class, whose importance will grow as we penetrate the full power of the object-oriented method.

Abstract data types and information hiding

A particularly interesting consequence of the object-oriented policy of basing all modules on ADT implementations (classes) is that it provides a clear answer to a question that was left pending in the discussion of information hiding: how do we select the public and private features of a module — the visible and invisible parts of the iceberg?

See the mention of vagueness in the middle of page 52.


Public part:
ADT specification (E1)
Secret part:
• Choice of representation (E2)
• Implementation of functions by features (E3)

The ADT view of a module under information hiding

If the module is a class coming from an ADT as outlined above, the answer is clear: of the three parts involved in the transition, *E1*, the ADT specification, is public; *E2* and *E3*, the choice of representation and the implementation of the ADT functions in terms of this representation, should be secret. (As we start building classes we will encounter a fourth part, also secret: auxiliary features needed only for the internal purposes of these routines.)

So the use of abstract data types as the source of our modules gives us a practical, unambiguous guideline for applying information hiding in our designs.

Introducing a more imperative view

The transition from abstract data types to classes involves an important stylistic difference: the introduction of change and imperative reasoning.

As you will remember, the specification of abstract data types is change-free, or, to use a term from theoretical computing science, *applicative*. All features of an ADT are modeled as mathematical functions; this applies to creators, queries and commands. For example the push operation on stacks is modeled by the command function

$$put\colon STACK\ [G] \times G \rightarrow STACK\ [G]$$

specifying an operation that returns a new stack, rather than changing an existing stack.

Classes, which are closer to the world of design and implementation, abandon this applicative-only view and reintroduce commands as operations that may change objects.

For example, *put* will appear as a routine which takes an argument of type *G* (the formal generic parameter), and modifies a stack by pushing a new element on top — instead of producing a new stack.

This change of style reflects the imperative style that prevails in software construction. (The word “operational” is also used as synonym for “imperative”.) It will require the corresponding change in the axioms of ADTs. Axioms A1 and A4 of stacks, which appeared above as

A1 • *item* (*put* (*s*, *x*)) = *x*

A4 • **not** *empty* (*put* (*s*, *x*))

will yield, in the imperative form, a clause known as a **routine postcondition**, introduced by the keyword **ensure** in

```
put (x: G) is
        -- Push x on top of stack
    require
        … The precondition, if any …
    do
        … The appropriate implementation, if known …
    ensure
        item = x
        not empty
    end
```

Here the postcondition expresses that on return from a call to routine *put*, the value of *item* will be *x* (the element pushed) and the value of *empty* will be false.

Other axioms of the ADT specification will yield a clause known as the **class invariant**. Postconditions, class invariants and other non-applicative avatars of an ADT's preconditions and axioms will be studied as part of the discussion of assertions and Design by Contract.

"THE ADT CONNECTION", 11.10, page 373.

Back to square one?

If you followed carefully, starting with the chapter on modularity, the line of reasoning that led to abstract data types and then classes, you may be a little puzzled here. We started with the goal of obtaining the best possible modular structures; various arguments led to the suggestion that objects, or more precisely object types, would provide a better basis than their traditional competitors — functions. This raised the next question: how to describe these object types. But when the answer came, in the form of abstract data types (and their practical substitutes, classes), it meant that we must base the description of data on… the applicable functions! Have we then come full circle?

No. Object types, as represented by ADTs and classes, remain the undisputed basis for modularization.

It is not surprising that both the object and function aspects should appear in the final system architecture: as noted in the previous chapter, no description of software issues can be complete if it misses one of these two components. What fundamentally distinguishes object-oriented methods from older approaches is the distribution of roles: object types are the undisputed winners when it comes to selecting the criteria for building modules. Functions remain their servants.

In object-oriented decomposition, no function ever exists just by itself: every function is attached to some object type. This carries over to the design and implementation levels: no feature ever exists just by itself; every feature is attached to some class.

Object-oriented software construction

The study of abstract data types has given us the answer to the question asked at the beginning of this chapter: how to describe the object types that will serve as the backbone of our software architecture.

See page 116 for the original definition.

We already had a definition of object-oriented software construction: remaining at a high level of generality, it presented the method as "basing the architecture of any software system on modules deduced from the types of objects it manipulates". Keeping that first definition as the framework, we can now complement it with a more technical one:

> **Object-oriented software construction (definition 2)**
>
> Object-oriented software construction is the building of software systems as structured collections of possibly partial abstract data type implementations.

This will be our working definition. Its various components are all important:

- The basis is the notion of *abstract data type*.
- For our software we need not the ADTs themselves, a mathematical notion, but ADT *implementations*, a software notion.
- These implementations, however, need not be complete; the "*possibly partial*" qualification covers deferred classes — including the extreme case of a fully deferred class, where none of the features is implemented.
- A system is a *collection* of classes, with no one particularly in charge — no top or main program.
- The collection is *structured* thanks to two inter-class relations: client and inheritance.

6.6 BEYOND SOFTWARE

As we are completing our study of abstract data types it is worth taking a moment to reflect on the significance of this notion outside of its immediate intended application area.

What the ADT approach tells us is that a successful intellectual investigation should renounce as futile any attempt at knowing things from the inside, and concentrate instead on their usable properties. Do not tell me what you are; tell me what you have — what I can get out of you. If we need a name for this epistemological discipline, we should call it the *principle of selfishness*.

If I am thirsty, an orange is something I can squeeze; if I am a painter, it is color which might inspire my palette; if I am a farmer, it is produce that I can sell at the market; if I am an architect, it is slices that tell me how to design my new opera house, overlooking the harbor; but if I am none of these, and have no other use for the orange, then I should not talk about it, as the concept of orange does not for me even exist.

The principle of selfishness — you are but what you have — is an extreme form of an idea that has played a central role in the development of science: abstraction, or the importance of separating concerns. The two quotations at the beginning of this chapter, each in its own remarkable way, express the importance of this idea. Their authors, Diderot and Stendhal, were writers rather than scientists, although both obviously had a good understanding of the scientific method (Diderot was the living fire behind the Great Encyclopedia, and Stendhal prepared for admission into the École Polytechnique, although in the end he decided that he could do better things with his life). It is striking to see how both quotations are applicable to the use of abstraction in software development.

Yet there is more than abstraction to the principle of selfishness: the idea, almost shocking at first, that a property is not worth talking about unless it is useful in some direct way to the talker.

This suggests a more general observation as to the intellectual value of our field.

Over the years many articles and talks have claimed to examine how software engineers could benefit from studying philosophy, general systems theory, "cognitive science", psychology. But to a practicing software developer the results are disappointing. If we exclude from the discussion the generally applicable laws of rational investigation, which enlightened minds have known for centuries (at least since Descartes) and which of course apply to software science as to anything else, it sometimes seems that experts in the disciplines mentioned may have more to learn from experts in software than the reverse.

Software builders have tackled — with various degrees of success — some of the most challenging intellectual endeavors ever undertaken. Few engineering projects, for example, match in complexity the multi-million line software projects commonly being launched nowadays. Through its more ambitious efforts the software community has gained precious insights on such issues and concepts as size, complexity, structure, abstraction, taxonomy, concurrency, recursive reasoning, the difference between description and prescription, language, change and invariants. All this is so recent and so tentative that the profession itself has not fully realized the epistemological implications of its own work.

Eventually someone will come and explain what lessons the experience of software construction holds for the intellectual world at large. No doubt abstract data types will figure prominently in the list.

6.7 SUPPLEMENTARY TOPICS

The view of abstract data types presented so far will suffice for the uses of ADTs in the rest of this book. (To complement it, doing the exercises will help you sharpen your understanding of the concept.)

If, as I hope, you have been conquered by the elegance, simplicity and power of ADTs, you may want to explore a few more of their properties, even though the discussion of object-oriented methods will not use them directly. These supplementary topics, which may be skipped on first reading, are presented in the next few pages:

- Implicitness and its relationship to the software construction process.
- The difference between specification and design.
- The differences between classes and records.
- Potential alternatives to the use of partial functions.
- Deciding whether a specification is complete or not.

The bibliographical references to this chapter point to more advanced literature on abstract data types.

More on implicitness

The implicit nature of abstract data types and classes, discussed above, reflects an important problem of software construction.

One may legitimately ask what difference there is between a simplified ADT specification, using the function declarations

x: *POINT* → *REAL*
y: *POINT* → *REAL*

and the record type declaration which we may express in a traditional programming language such as Pascal under the form

```
type
    POINT =
        record
            x, y: real
        end
```

At first sight, the two definitions appear equivalent: both state that any instance of type *POINT* has two associated values x and y, of type *REAL*. But there is a crucial if subtle difference:

- The Pascal form is closed and explicit: it indicates that a *POINT* object is made of the two given fields, and no other.
- The ADT function declarations carry no such connotation. They indicate that one may query a point about its x and its y, but do not preclude other queries — such as a point's mass and velocity in a kinematics application.

From a simplified mathematical perspective, you may consider that the above Pascal declaration is a definition of the mathematical set *POINT* as a cartesian product:

$$POINT \triangleq REAL \times REAL$$

where $\triangleq$ means "is defined as": this defines *POINT* fully. In contrast, the ADT specification does not explicitly define *POINT* through a mathematical model such as the cartesian product; it just characterizes *POINT* implicitly by listing two of the queries applicable to objects of this type.

If at some stage you think you are done with the specification of a certain notion, you may want to move it from the implicit world to the explicit world by identifying it with the cartesian product of the applicable simple queries; for example you will identify points with <*x*, *y*> pairs. We may view this identification process as the very definition of the transition from analysis and specification to design and implementation.

Specification versus design

The last observation helps clarify a central issue in the study of software: the difference between the initial activities of software development — specification, also called analysis — and later stages such as design and implementation.

The software engineering literature usually defines this as the difference between "defining the problem" and "building a solution". Although correct in principle, this definition is not always directly useful in practice, and it is sometimes hard to determine where specification stops and design begins. Even in the research community, people routinely criticize each other on the theme "you advertize notation *x* as a specification language, but what it really expresses is designs". The supreme insult is to accuse the notation of catering to *implementation*; more on this in a later chapter.

See "The clouds and the precipice", page 905.

The above definition yields a more precise criterion: to cross the Rubicon between specification and design is to move from the implicit to the explicit; in other words:

Definition: transition from analysis (specification) to design

To go from specification to design is to identify each abstraction with the cartesian product of its simple queries.

The subsequent transition — from design to implementation — is simply the move from one explicit form to another: the design form is more abstract and closer to mathematical concepts, the implementation form is more concrete and computer-oriented, but they are both explicit. This transition is less dramatic than the preceding one; indeed, it will become increasingly clear in the pages that follow that object technology all but removes the distinction between design and implementation. With good object-oriented notations, what our computers directly execute (with the help of our compilers) is what to the non-O-O world would often appear as designs.

Classes versus records

Another remarkable property of object technology, also a result of the focus on implicit definition, is that you can keep your descriptions implicit for a much longer period than with any other approach. The following chapters will introduce a notation enabling us to define a class under the form

```
class POINT feature
    x, y: REAL
end
```

This looks suspiciously close to the above Pascal record type definition. But in spite of appearances the class definition is different: it is implicit! The implicitness comes from inheritance; the author of the class or (even more interestingly) someone else may at any time define a new class such as

```
class MOVING_POINT inherit
        POINT
feature
        mass: REAL
        velocity: VECTOR [REAL]
end
```

which extends the original class in ways totally unplanned for by the initial design. Then a variable (or entity, to use the terminology introduced later) of type *POINT*, declared as

```
p1: POINT
```

may become attached to objects which are not just of type *POINT* but also of any descendant type such as *MOVING_POINT*. This occurs in particular through "polymorphic assignments" of the form

```
p1 := mp1
```

where *mp1* is of type *MOVING_POINT*.

These possibilities illustrate the implicitness and openness of the class definition: the corresponding entities represent not just points in the narrow sense of direct instances of class *POINT* as initially defined, but, more generally, instances of any eventual class that describes a concept derived from the original.

See "The Open-Closed principle", page 57.

The ability to define software elements (classes) that are directly usable while remaining implicit (through inheritance) is one of the major innovations of object technology, directly answering the Open-Closed requirement. Its full implications will unfold progressively in the following chapters.

Not surprisingly for such a revolutionary concept, the realm of new possibilities that it opens still scares many people, and in fact many object-oriented languages restrict the openness in some way. Later chapters will mention examples.

Alternatives to partial functions

Among the techniques of this chapter that may have caused you to raise your eyebrows is its use of partial functions. The problem that it addresses is inescapable: any specification needs to deal with operations that are not always defined; for example, it is impossible to pop an empty stack. But is the use of partial functions the best solution?

It is certainly not the only possible one. Another technique that comes to mind, and is indeed used by some of the ADT literature, is to make the function total but introduce special error values to denote the results of operations applied to impossible cases.

For every type T, this method introduces a special "error" value; let us write it ω_T. Then for any function f of signature

$$f\text{: … Input types … } \rightarrow T$$

it specifies that any application of f to an object for which the corresponding computer operation may not be executed will produce the value ω_T.

Although usable, this method leads to mathematical and practical unpleasantness. The problem is that the special values are rather bizarre animals, which may unduly disturb the lives of innocent mathematical creatures.

Assume for example that we consider stacks of integers — instances of the generic derivation *STACK* [*INTEGER*], where *INTEGER* is the ADT whose instances are integers. Although we do not need to write the specification of *INTEGER* completely for this discussion, it is clear that the functions defining this ADT should model the basic operations (addition, subtraction, "less than" and the like) defined on the mathematical set of integers. The axioms of the ADT should be consistent with ordinary properties of integers; typical among these properties is that, for any integer n:

[Z1]

$$n + 1 \neq n$$

Now let n be the result of requesting the top of an empty stack, that is to say, the value of *item* (*new*), where *new* is an empty stack of integers. With the "special error element" approach, n must be the special value $\omega_{INTEGER}$. What then is the value of the expression $n + 1$? If the only values at our disposal are normal integers and $\omega_{INTEGER}$, then we ought to choose $\omega_{INTEGER}$ as the answer:

$$\omega_{INTEGER} + 1 = \omega_{INTEGER}$$

This is the only acceptable choice: any other value for $\omega_{INTEGER} + 1$, that is to say, any "normal" integer q, would mean in practical terms that after we attempt to access the top of an empty stack, and get an error value as a result, we can miraculously remove any trace of the error, simply by adding one to the result! This might have passed when all it took to erase the memory of a crime was a pilgrimage to Santiago de Compostela and the purchase of a few indulgences; modern mores and computers are not so lenient.

But choosing $\omega_{INTEGER}$ as the value of $n + 1$ when n is $\omega_{INTEGER}$ violates the above Z1 property. More generally, $\omega_{INTEGER} + p$ will be $\omega_{INTEGER}$ for any p. This means we must develop a new axiom system for the updated abstract data type (*INTEGER* enriched with an error element), to specify that every integer operation yields $\omega_{INTEGER}$ whenever any one of its arguments is $\omega_{INTEGER}$. Similar changes will be needed for every type.

The resulting complication seems unjustifiable. We cannot change the specification of integers just for the purpose of modeling a specific data structure such as the stack.

With partial functions, the situation is simpler. You must of course verify, for every expression involving partial functions, that the arguments satisfy the corresponding preconditions. This amounts to performing a sanity check — reassuring yourself that the result of the computation will be meaningful. Having completed this check, you may apply the axioms without further ado. You need not change any existing axiom systems.

Is my specification complete?

Another question may have crossed your mind as you were reading the above example of abstract data type specification: is there is any way to be sure that such a specification describes all the relevant properties of the objects it is intended to cover? Students who are asked to write their first specifications (for example when doing the exercises at the end of this chapter) often come back with the same question: when do I know that I have specified enough and that I can stop?

In more general terms: does a method exist to find out whether an ADT specification is complete?

If the question is asked in this simple form, the answer is a plain no. This is true of formal specifications in general: to say that a specification is complete is to claim that it covers all the needed properties; but this is only meaningful with respect to some document listing these properties and used as a reference. Then we face one of two equally disappointing situations:

- If the reference document is informal (a natural-language "requirements document" for a project, or perhaps just the text of an exercise), this lack of formality precludes any attempt to check systematically that the specification meets all the requirements described in that document.

- If the reference document is itself formal, and we are able to check the completeness of our specification against it, this merely pushes the problem further: how do we ascertain the completeness of the reference document itself?

In its trivial form, then, the completeness question is uninteresting. But there is a more useful notion of completeness, derived from the meaning of this word in mathematical logic. For a mathematician, a theory is complete if its axioms and rules of inference are powerful enough to prove the truth or falsity of any formula that can be expressed in the language of the theory. This meaning of completeness, although more limited, is intellectually satisfying, since it indicates that whenever the theory lets us express a property it also enables us to determine whether the property holds.

How do we transpose this idea to an ADT specification? Here the "language of the theory" is the set of all the **well-formed expressions**, those expressions which we may build using the ADT's functions, applied to arguments of the appropriate types. For example, using the specification of *STACK* and assuming a valid expression *x* of type *G*, the following expressions are well-formed:

```
new
put (new, x)
item (new)         -- If this seems strange, see comments on the next page.
empty (put (new, x))
stackexp           -- The complex expression defined on page 140.
```

The expressions *put* (*x*) and *put* (*x*, *new*), however, are not well-formed, since they do not abide by the rules: *put* always requires two arguments, the first of type *STACK* [*G*] and the second of type *G*; so *put* (*x*) is missing an argument, and *put* (*x*, *new*) has the wrong argument types.

The third example in the preceding box, *item* (*new*), does not describe a meaningful computation since *new* does not satisfy the precondition of *item*. Such an expression, although well-formed, is not **correct**. Here is the precise definition of this notion:

Definition: correct ADT expression

Let $f(x_1, \ldots, x_n)$ be a well-formed expression involving one or more functions on a certain ADT. This expression is correct if and only if all the x_i are (recursively) correct, and their values satisfy the precondition of f, if any.

Do not confuse "correct" with "well-formed". Well-formedness is a structural property, indicating whether all the functions in an expression have the right number and types of arguments; correctness, which is only defined for a well-formed expression, indicates whether the expression defines a meaningful computation. As we have seen, the expression *put* (*x*) is not well-formed (and so it is pointless to ask whether it is correct), whereas the expression *item* (*new*) is well-formed but not correct.

An expression well-formed but not correct, such as *item* (*new*), is similar to a program that compiles (because it is built according to the proper syntax and satisfies all typing constraints of the programming language) but will crash at run time by performing an impossible operation such as division by zero or popping an empty stack.

Of particular interest for completeness, among well-formed expressions, are **query expressions**, those whose outermost function is a query. Examples are:

empty (*put* (*put* (*new*, *x1*), *x2*))
item (*put* (*put* (*new*, *x1*), *x2*))
stackexp -- See page 140

The "queries" in our example, returning a result of type other than STACK, are item and empty. See "Function categories", page 134.

A query expression denotes a value which (if defined) belongs not to the ADT under definition, but to another, previously defined type. So the first query expression above has a value of type *BOOLEAN*; the second and third have values of type *G*, the formal generic parameter — for example *INTEGER* if we use the generic derivation *STACK* [*INTEGER*].

Query expressions represent external observations that we may make about the results of a certain computation involving instances of the new ADT. If the ADT specification is useful, it should always enable us to find out whether such results are defined and, if so, what they are. The stack specification appears to satisfy this property, at least for the three example expressions above, since it enables us to determine that the three expressions are defined and, by applying the axioms, to determine their values:

empty (*put* (*put* (*new*, *x1*), *x2*)) = *False*
item (*put* (*put* (*new*, *x1*), *x2*)) = *x2*
stackexp = *x4*

Transposed to the case of arbitrary ADT specifications, these observations suggest a pragmatic notion of completeness, known as *sufficient* completeness, which expresses that the specification contains axioms powerful enough to enable us to find the result of any query expression, in the form of a simple value.

Here is the precise definition of sufficient completeness. (Non-mathematically inclined readers should skip the rest of this section.)

Definition: sufficient completeness

An ADT specification for a type *T* is sufficiently complete if and only if the axioms of the theory make it possible to solve the following problems for any well-formed expression *e*:

S1 • Determine whether *e* is correct.

S2 • If *e* is a query expression and has been shown to be correct under *S1*, express *e*'s value under a form not involving any value of type *T*.

In *S2*, expression *e* is of the form $f\,(x_1, \ldots, x_n)$ where *f* is a query function, such as *empty* and *item* for stacks. *S1* tells us that *e* has a value, but this is not enough; in this case we also want to know what the value is, expressed only in terms of values of other types (in the *STACK* example, values of types *BOOLEAN* and *G*). If the axioms are strong enough to answer this question in all possible cases, then the specification is sufficiently complete.

Sufficient completeness is a useful practical guideline to check that no important property has been left out of a specification, answering the question raised above: when do I know I can stop looking for new properties to describe in the specification? It is good practice to apply this check, at least informally, to any ADT specification that you write — starting with your answers to the exercises of this chapter. Often, a formal proof of sufficient correctness is possible; the proof given below for the *STACK* specification defines a model which can be followed in many cases.

As you may have noted, *S2* is optimistic in talking about "the" value of *e*: what if the axioms yield two or more? This would make the specification useless. To avoid such a situation we need a further condition, known from mathematical logic as consistency:

Definition: ADT consistency

An ADT specification is consistent if and only if, for any well-formed query expression *e*, the axioms make it possible to infer at most one value for *e*.

The two properties are complementary. For any query expression we want to be able to deduce exactly one value: at least one (sufficient completeness), but no more than one (consistency).

Proving sufficient completeness

(This section and the rest of this chapter are supplementary material and its results are not needed in the rest of the book.)

Non-mathematically inclined readers may skip to "KEY CONCEPTS INTRODUCED IN THIS CHAPTER", 6.8, page 159

The sufficient completeness of an abstract data type specification is, in general, an undecidable problem. In other words, no general proof method exists which, given an arbitrary ADT specification, would tell us in finite time whether or not the specification is sufficiently complete. Consistency, too, is undecidable in the general case.

It is often possible, however, to prove the sufficient completeness and the consistency of a particular specification. To satisfy the curiosity of mathematically inclined readers, it is interesting to prove, as a conclusion to this chapter, that the specification of *STACK* is indeed sufficiently complete. The proof of consistency will be left as an exercise.

Proving the sufficient completeness of the stack specification means devising a valid rule addressing problems *S1* and *S2* above; in other words the rule must enable us, for an arbitrary stack expression *e*:

S1 • To determine whether *e* is correct.

S2 • If *e* is correct under *S1* and its outermost function is *item* or *empty* (one of the two query functions), to express its value in terms of *BOOLEAN* and *G* values only, without any reference to values of type *STACK* [*G*] or to the functions of *STACK*'s specification.

It is convenient for a start to consider only well-formed expressions which do not involve any of the two query functions *item* and *empty* — so that we only have to deal with expressions built out of the functions *new*, *put* and *remove*. This means that only problem *S1* (determining whether an expression is defined) is relevant at this stage. Query functions and *S2* will be brought in later.

The following property, which we must prove, yields a rule addressing *S1*:

Weight Consistency rule

A well-formed stack expression *e*, involving neither *item* nor *empty*, is correct if and only if its weight is non-negative, and any subexpression of *e* is (recursively) correct.

Here the "weight" of an expression represents the number of elements in the corresponding stack; it is also the difference between the number of nested occurrences of *put* and *remove*. Here is the precise definition of this notion:

Definition: weight

The weight of a well-formed stack expression not involving *item* or *empty* is defined inductively as follows:

W1 • The weight of the expression *new* is 0.

W2 • The weight of the expression *put* (*s*, *x*) is *ws* + *1*, where *ws* is the weight of *s*.

W3 • The weight of the expression *remove* (*s*) is *ws* — *1*, where *ws* is the weight of *s*.

Informally, the Weight Consistency rule tells us that a stack expression is correct if and only if the expression and every one of its subexpressions, direct or indirect, has at least as many *put* operations (pushing an element on top) as it has *remove* operations (removing the top element); if we view the expression as representing a stack computation, this means that we never try to pop more than we have pushed. Remember that at this stage we are only concentrating on *put* and *remove*, ignoring the queries *item* and *empty*.

This intuitively seems right but of course we must prove that the Weight Consistency rule indeed holds. It will be convenient to introduce a companion rule and prove the two rules simultaneously:

Zero Weight rule

Let *e* be a well-formed and correct stack expression not involving *item* or *empty*. Then *empty* (*e*) is true if and only if *e* has weight 0.

The proof uses induction on the nesting level (maximum number of nested parentheses pairs) of the expression. Here again, for ease of reference, are the earlier axioms applying to function *empty*:

STACK AXIOMS

For any *x*: *G*, *s*: *STACK* [*G*]

A3 • *empty* (*new*)

A4 • **not** *empty* (*put* (*s*, *x*))

An expression *e* with nesting level 0 (no parentheses) may only be of the form *new*; so its weight is 0, and it is correct since *new* has no precondition. Axiom A3 indicates that *empty* (*e*) is true. This takes care of the base step for both the Weight Consistency rule and the Zero Weight rule.

For the induction step, assume that the two rules are applicable to all expressions of nesting level *n* or smaller. We must prove that they apply to an arbitrary expression *e* of nesting level *n* + *1*. Since for the time being we have excluded the query functions from our expressions, one of the following two forms must apply to *e*:

E1 • *e = put (s, x)*

E2 • *e = remove (s)*

where *x* is of type *G*, and *s* has nesting level *n*. Let *ws* be the weight of *s*.

In case *E1*, since *put* is a total function, *e* is correct if and only if *s* is correct, that is to say (by the induction hypothesis) if and only if *s* and all its subexpressions have non-negative weights. This is the same as saying that *e* and all its subexpressions have non-negative weights, and so proves that the Weight Consistency rule holds in this case. In addition, *e* has the positive weight *ws + 1*, and (by axiom A4) is not empty, proving that the Zero Weight rule also holds.

In case *E2*, expression *e* is correct if and only if both of the following conditions hold:

EB1 • *s* and all its subexpressions are correct.

EB2 • **not** *empty (s)* (this is the precondition of *remove*).

Because of the induction hypothesis, condition *EB2* means that *ws*, the weight of *s*, is positive, or, equivalently, that *ws – 1*, the weight of *e*, is non-negative. So *e* satisfies the Weight Consistency rule. To prove that it also satisfies the Zero Weight rule, we must prove that *e* is empty if and only if its weight is zero. Since the weight of *s* is positive, *s* must contain at least one occurrence of *put*, which also appears in *e*. Consider the outermost occurrence of *put* in *e*; this occurrence is enclosed in a *remove* (since *e* has a *remove* at the outermost level). This means that a subexpression of *e*, or *e* itself, is of the form

remove (put (stack_expression, g_expression))

which axiom A2 indicates may be reduced to just *stack_expression*. Performing this replacement reduces the weight of *e* by 2; the resulting expression, which has the same value as *e*, satisfies the Zero Weight rule by the induction hypothesis. This proves the induction hypothesis for case *E2*.

The proof has shown in passing that in any well-formed and correct expression which does not involve the query functions *item* and *empty* we may "remove every *remove*", that is to say, obtain a canonical form that involves only *put* and *new*, by applying axiom A2 wherever possible. For example, the expression

put (remove (remove (put (put (remove (put (put (new, x1), x2)), x3), x4))), x5)

has the same value as the canonical form

put (put (new, x1), x5)

For the record, let us give this mechanism a name and a definition:

Canonical Reduction rule

Any well-formed and correct stack expression involving neither *item* nor *empty* has an equivalent "canonical" form that does not involve *remove* (that is to say, may fsonly involve *new* and *put*). The canonical form is obtained by applying the stack axiom A2 as many times as possible.

This takes care of the proof of sufficient completeness but only for expressions that do not involve any of the query functions, and consequently for property *S1* only (checking the correctness of an expression). To finish the proof, we must now take into account expressions that involve the query functions, and deal with problem *S2* (finding the values of these query expressions). This means we need a rule to determine the correctness and value of any well-formed expression of the form *f* (*s*), where *s* is a well-formed expression and *f* is either *empty* or *item*.

The rule and the proof of its validity use induction on the nesting level, as defined above. Let *n* be the nesting level of *s*. If *n* is 0, *s* can only be *new* since all the other functions require arguments, and so would have at least one parenthesis pair. Then the situation is clear for both of the query functions:

- *empty* (*new*) is correct and has value true (axiom A3).
- *item* (*new*) is incorrect since the precondition of *item* is **not** *empty* (*s*).

For the induction step, assume that *s* has a nesting depth *n* of one or more. If any subexpression *u* of *s* has *item* or *empty* as its outermost function, then *u* has a depth of at most *n* – *1*, so the induction hypothesis indicates that we can determine whether *u* is correct and, if it is, obtain the value of *u* by applying the axioms. By performing all such possible subexpression replacements, we obtain for *s* a form which involves no stack function other than *put*, *remove* and *new*.

Next we may apply the idea of canonical form introduced above to get rid of all occurrences of *remove*, so that the resulting form of *s* may only involve *put* and *new*. The case in which *s* is just *new* has already been dealt with; it remains the case for which *s* is of the form *put* (*s'*, *x*). Then for the two expressions under consideration:

- *empty* (*s*) is correct, and axiom A3 indicates that the value of this expression is **false**.
- *item* (*s*) is correct, since the precondition of *item* is precisely **not** *empty* (*s*); axiom A1 indicates that the value of this expression is *x*.

This concludes the proof of sufficient completeness since we have now proved the validity of a set of rules — the Weight Consistency rule and the Canonical Reduction rule — enabling us to ascertain whether an arbitrary stack expression is correct and, for a correct query expression, to determine its value in terms of *BOOLEAN* and *G* values only.

6.8 KEY CONCEPTS INTRODUCED IN THIS CHAPTER

- The theory of abstract data types (ADT) reconciles the need for precision and completeness in specifications with the desire to avoid overspecification.
- An abstract data type specification is a formal, mathematical description rather than a software text. It is *applicative*, that is to say change-free.
- An abstract data type may be generic and is defined by functions, axioms and preconditions. The axioms and preconditions express the semantics of a type and are essential to a full, unambiguous description.
- To describe operations which are not always defined, partial functions provide a convenient mathematical model. Every partial function has a precondition, stating the condition under which it will yield a result for any particular candidate argument.

- An object-oriented system is a collection of classes. Every class is based on an abstract data type and provides a partial or full implementation for that ADT.
- A class is effective if it is fully implemented, deferred otherwise.
- Classes should be designed to be as general and reusable as possible; the process of combining them into systems is often bottom-up.
- Abstract data types are implicit rather than explicit descriptions. This implicitness, which also means openness, carries over to the entire object-oriented method.
- No formal definition exists for the intuitively clear concept of an abstract data type specification being "complete". A rigorously defined notion, *sufficient* completeness, usually provides the answer. Although no method is possible to ascertain the sufficient completeness of an arbitrary specification, proofs are often possible for specific cases; the proof given in this chapter for the stack specification may serve as a guide for other examples.

6.9 BIBLIOGRAPHICAL NOTES

A few articles published in the early nineteen-seventies made the discovery of abstract data types possible. Notable among these are Hoare's paper on the "proof of correctness of data representations" [Hoare 1972a], which introduced the concept of abstraction function, and Parnas's work on information hiding mentioned in the bibliographical notes to chapter *3*.

Abstract data types, of course, go beyond information hiding, although many elementary presentations of the concept stop there. ADTs proper were introduced by Liskov and Zilles [Liskov 1974]; more algebraic presentations were given in [M 1976] and [Guttag 1977]. The so-called ADJ group (Goguen, Thatcher, Wagner) explored the algebraic basis of abstract data types, using category theory. See in particular their influential article [Goguen 1978], published as a chapter in a collective book.

Several specification languages have been based on abstract data types. Two resulting from the work of the ADJ group are CLEAR [Burstall 1977] [Burstall 1981] and OBJ-2 [Futatsugi 1985]. See also Larch by Guttag, Horning and Wing [Guttag 1985]. ADT ideas have influenced formal specification languages such as Z in its successive incarnations [Abrial 1980] [Abrial 1980a] [Spivey 1988] [Spivey 1992] and VDM [Jones 1986]. The notion of abstraction function plays a central role in VDM. Recent extensions to Z have established a closer link to object-oriented ideas; see in particular Object Z [Duke 1991] and further references in chapter *11*.

The phrase "separation of concerns" is central in the work of Dijkstra; see in particular his "Discipline of Programming" [Dijkstra 1976].

The notion of sufficient completeness was first published by Guttag and Horning (based on Guttag's 1975 thesis) in [Guttag 1978].

The idea that going from specification to design means switching from the implicit to the explicit by identifying an ADT with the cartesian product of its simple queries was suggested in [M 1982] as part of a theory for describing data structures at three separate levels (physical, structural, implicit).

EXERCISES

E6.1 Points

Write a specification describing the abstract data type *POINT*, modeling points in plane geometry. The specification should cover the following aspects: cartesian and polar coordinates; rotation; translation; distance of a point to the center; distance to another point.

E6.2 Boxers

Members of the Association Dijonnaise des Tapeventres, a boxing league, regularly compete in games to ascertain their comparative strength. A game involves two boxers; it either results in a winner and a loser or is declared a tie. If not a tie, the outcome of a game is used to update the ranking of players in the league: the winner is declared better than the loser and than any boxer *b* such that the loser was previously better than *b*. Other comparative rankings are left unchanged.

Specify this problem as a set of abstract data types: *ADT_LEAGUE*, *BOXER*, *GAME*. (**Hint**: do not introduce the notion of "ranking" explicitly, but model it by a function *better* expressing whether a player is better than another in the league.)

E6.3 Bank accounts

Write an ADT specification for a "bank account" type with operations such as "deposit", "withdraw", "current balance", "holder", "change holder".

How would you add functions representing the opening and closing of an account? (**Hint**: these are actually functions on another ADT.)

E6.4 Messages

Consider an electronic mail system with which you are familiar. In light of this chapter's discussion, define *MAIL_MESSAGE* as an abstract data type. Be sure to include not just query functions but also commands and creators.

E6.5 Names

See "How long is a middle initial?", page 125.

Devise a *NAME* abstract data type taking into account the different components of a person's name.

E6.6 Text

Consider the notion of text, as handled by a text editor. Specify this notion as an abstract data type. (This statement of the exercise leaves much freedom to the specifier; make sure to include an informal description of the properties of text that you have chosen to model in the ADT.)

E6.7 Buying a house

Write an abstract data type specification for the problem of buying a house, sketched in the preceding chapter. Pay particular attention to the definition of logical constraints, expressed as preconditions and axioms in the ADT specification.

"Ordering and O-O development", page 111.

E6.8 More stack operations

Modify the ADT specification of stacks to account for operations *count* (returning the number of elements on a stack), *change_top* (replacing the top of the stack by a given element) and *wipe_out* (remove all elements). Make sure to include new axioms and preconditions as needed.

E6.9 Bounded stacks

Adapt the specification of the stack ADT presented in this chapter so that it will describe stacks of bounded capacity. (Hint: introduce the capacity as an explicit query function; make *put* partial.)

E6.10 Queues

Describe queues (first-in, first-out) as an abstract data type, in the style used for *STACK*. Examine closely the similarities and differences. (**Hint**: the axioms for *item* and *remove* must distinguish, to deal with *put* (s, x), the cases in which s is empty and non-empty.)

E6.11 Dispensers

(This exercise assumes that you have answered the previous one.)

Specify a general ADT *DISPENSER* covering both stack and queue structures.

Discuss a mechanism for expressing more specialized ADT specifications such as those of stacks and queues by reference to more general specifications, such as the specification of dispensers. (**Hint**: look at the inheritance mechanism studied in later chapters.)

E6.12 Booleans

Define *BOOLEAN* as an abstract data type in a way that supports its use in the ADT definitions of this chapter. You may assume that equality and inequality operations ($=$ and $\neq$) are automatically defined on every ADT.

E6.13 Sufficient completeness

(This exercise assumes that you have answered one or more of the preceding ones.) Examine an ADT specification written in response to one of the preceding exercises, and try to prove that it is sufficiently complete. If it is not sufficiently complete, explain why, and show how to correct or extend the specification to satisfy sufficient completeness.

E6.14 Consistency

Prove that the specification of stacks given in this chapter is consistent.