ABSTRACT DATA TYPES

An *abstract data type* is a specified set of objects and operations. A software module that implements an abstract data type hides information about how its objects are represented. The module hides information by forcing the type's objects to be manipulated indirectly through the operations. Since objects are only manipulated indirectly through these operations, the information that the operations "know" is encapsulated in the module and can thus be changed easily.

The ability to easily change software is important, because the cumulative cost of all changes to a program is usually higher than the cost of writing it in the first place (1). But why do programs need to change at all? Physical processes, such as friction, do not affect software. However, wear and tear is only one reason for fixing or replacing a machine; machines also need fixing if they have design errors, and they are sometimes improved or replaced by a machine that better fits their user's needs. Programs need to be maintained or enhanced for similar reasons. For example, once a program is released, its users typically see the need for correcting various design or coding errors, and often desire several improvements. Errors in programs that support critical business or safety functions cannot be ignored, and lives or the health of a business may depend on the speed with which such errors can be corrected. But even a user's desires may become necessary requirements in a competitive market, where the faster a company can respond to such desires for change, the more profitable it can be. Planning for such inevitable corrections and enhancements is good practice, and will save much time, frustration, and money.

After examining the problems that abstract data types solve, we will look at information hiding in general. Then we will see how to design abstract data types. We will conclude with a look at some more advanced topics.

ABSTRACT DATA TYPES ENCAPSULATE DATA STRUCTURES

Abstract data types, and encapsulation in general, help ease changes in a program's data structures. Such changes are very common. Furthermore, if one does not use abstract data types and encapsulation, such changes can be very costly.

Data Structures

A *data structure* is a computer representation of information. For, example, an array of 24 integers might be used to represent a series of hourly temperature readings.

Data structures may be either primitive or compound. *Primitive* (or *atomic*) data structures include the basic values found in a programming language, such as Characters and Integers. *Composite* data structures are built from such primitive values and other composite data structures. Arrays, records, lists, and sets are families of composite data structures. A particular member of such a family, such as an array of integers, is a composite data structure.

Details of Data Structures. To write code that uses a data structure, one must know two important details. These are its format and its abstraction mapping. The *format* of a data structure is how it is laid out in a computer's memory. For example, an Integer data structure might be formatted as a string of 32 bits. An array of 24 integers might be formatted as a consecutive sequence of 24 such 32-bit strings.

A data structure's *abstraction mapping* is a mapping from its format to the abstract values that it represents. For example, the abstraction mapping from an Integer data structure's format, a string of 32 bits, to an integer, in the abstract value space of mathematics, might be based on binary notation. For example, the bit string "00000000000000000000000000000101" represents the integer 5.

Abstraction mappings for composite data structures are usually built from the abstraction mappings of their elements. For example, consider how a sequence of 24 32-bit strings might be used to represent a series of temperature readings for a single day. It is easiest to describe this abstraction mapping in two stages: first use the abstraction mapping for the integers to transform the sequence of 24 32-bit strings into a sequence of 24 integers, then state how the 24 integers map to the sequence of temperature readings. For this reason, the format and abstraction mapping of a composite data structure are usually stated at a higher level of abstraction than those for a primitive data structure are. For example, we would state the format of the data structure for a day's temperature readings as a sequence of 24 integers, each of which represents the reading in degrees Celsius, with the first element of the sequence representing the reading for midnight, then next for 1AM, and so on. Similarly, one might regard the sequence of integers itself as a data structure, an array, and thus say that the format of the data structure was an array of 24 integers, call it "A," with each integer representing the reading in degrees Celsius, and with A[0] representing the reading for midnight, A[1] for 1AM, and so on up to A[23] representing the reading for 11PM.

Details and Changes. A program can extract information from a data structure if it "knows" both kinds of detail about the data structure. For example, to get the reading for 3AM from the array "A" described above one would write an expression like "A[3]" (depending on the exact programming language used). The programming language knows how to use the name of the array and its format (and the format of the integers) to extract the required information. However, note that the correspondence between 3AM and the "3" in "A[3]" is determined by the abstraction mapping for this particular data structure. If this abstraction mapping were changed, then even if the format (an array of 24 integers) were unchanged, the expression for extracting the reading for 3AM might have to change. For example, suppose that the abstraction mapping were changed such that the last reading of the day, the one taken at 11PM, was stored in A[0], the 10PM reading in A[1], and the midnight reading in A[23]. Then to extract the 3AM reading one would use the expression A[20]. Similarly, if the abstraction function maps an integer t to t degrees Celsius, then to extract the

reading in degrees Kelvin requires adding 273 to t. But if the abstraction function were changed to storing the temperature readings in degrees Kelvin, then extracting the reading in degrees Kelvin would not require this addition.

A desire for greater efficiency in the program's use of time or space is a common reason to change a program's data structures. For example, if requests for temperature readings in degrees Kelvin were more common than requests in degrees Celsius, then it would be more efficient to change the abstraction mapping and to store the readings in degrees Kelvin. As another example, the speed of a search through a list of names can be made faster if the data structure is changed from a list to a binary search tree or a hash table. (Searching for a name in a list takes, on the average, time proportional to half of the number of the names in the list. On the other hand, searching for a name in a binary search tree takes, on the average, time proportional to the logarithm of the number of names.)

Another reason for changing a data structure is to allow the program to store or manipulate additional information. For example, to record the humidity during each hour of a day as well, one would need to either change the data structure for the temperature readings, or add a new data structure to the program.

Why Changing Data Structures Can Be Costly. Making changes to a program's data structures can be costly because information *about* them tends to propagate throughout the program. Of course, the information contained *in* the data structures, the information being manipulated by the program itself, needs to flow throughout the program; that causes no maintenance problems. What causes problems is the use of a data structure's format and abstraction mapping throughout the program. When every part of the program uses these details, any change to these details could affect all parts of the program; hence any such change requires examining the entire program.

For example, consider a calendar management program. In this program, there will be a data structure used to represent dates. Suppose the format of a date is a record of three fields, each itself an array of two characters, with each character containing a digit. (This might be a reasonable choice if the most common operation performed on dates is to read or write them from or to some external device.) The abstraction mapping is as follows. The two digits in the "month" field's array represent the number of the month (with 1 for January), the two digits in the "day" field represent the day of the month, and the two digits in the "year" field represent the year within the current century. Without any further conventions or language support, these details will be used everywhere in the program.

Now suppose that, at some later time, we decide to change the program to deal with dates in more than one century. One way to make this change would be to change the data structure for dates, so that 4 characters were used to store the year (with the characters read left to right representing a year in the Common Era). However, details of the format and interpretation of dates might have been used anywhere in the program. For example, at one spot in the program, the year in a date, D, might be printed by printing the characters "2", "0", and then the values of the expressions "D.year[0]" and "D.year[1]." However, if the year 2101 is stored with D.year[0] = "2", D.year[1] = "1", D.year[2] = "0", and D.year[3] = "1", then this way of printing the year is incorrect, as it would print 2101 as "2021"! This kind of dependence on details might occur anywhere in the program, hence changing the representation of dates to use four characters for the year requires an expensive reexamination of the entire program. Exactly this kind of expensive reexamination of entire programs occurred at the end of the last century for exactly this reason.

When the code that uses a data structure is found in several programs, the problems caused by the propagation of the detailed information about the data structure are essentially unlimited. For example, if a data structure is present in an operating system or a commercial software component, details about it may be written into so many programs that it will be extraordinarily expensive to change.

Information Hiding and Encapsulation

To avoid such difficulties, it is best to hide the detailed information about each data structure in a small section of one's program (2). Such a small section of a program, which can typically be separately compiled and which typically hides the details of one data structure (or data type, see below), is called a *module*.

A module *hides* or *encapsulates* the details of a data structure if no other part of the program can directly depend on these details. That is, the parts of the program outside the module must manipulate the data structure indirectly. This technique is called *information hiding*, because it hides information about the data structure's details in a module. It is also called *encapsulation*, because the module is a "capsule" containing the hidden information in the sense that the details are protected from the other parts of the program.

Operations (Methods). Program modules achieve information hiding by restricting code outside the program module itself to the use of a specified set of *operations*. In the jargon of object-oriented programming these are called *methods*. These operations of a module "know" the details of the data structures that they manipulate, in the sense that they are coded using the details of the format of that data and its abstraction mapping. The other parts of the program manipulate the data structure indirectly, by calling the operations.

These operations can also be looked on as the set of services a module provides to the rest of a program. For example, one might have a module that hides information about the details of the data structure used for storing the appointments in a calendar manager program. This module might provide operations to add an appointment to the set of scheduled appointments, to cancel an appointment, and to find all the appointments for a given date.

Specifying a Module's Interface. The operations of a module are critical, since they form the interface between the data structures encapsulated by the module and the rest of the program. That is, instead of depending on the details of

the data structures, the rest of the program depends on the operations of the module. This means that the operations must be designed to be sufficiently high-level, or abstract, so that the details of the data structures hidden in the module did not leak through the interface. It also means that the exact behavior of the operations must be carefully specified.

A prime example of a set of operations that are too low-level to effectively hide information would be a set that included an operation that returned a pointer to the module's data structure to clients. With such an operation, clients would be able to directly manipulate the data structure. This would provide no information hiding at all. (In the jargon, returning a pointer to a module's internal data structure, or allowing a client to keep such a pointer to an data structure that is later made internal to the module, is called "exposing the representation" (3).

The *specification* of a module's operations describes how to call each operation and what happens when it is called. Such a specification is best thought of as a contract (3, 4). Like any contract, it gives benefits and obligations to each party involved. The two parties are the code that uses the operations (from outside of the module), called *client code*, and the module's own *implementation code*. The obligation of the implementation code is to make the operations behave as specified in the contract. The implementation code benefits by being able to use whatever algorithms or data structures are desired. (Usually the algorithms and data structures will be chosen to minimize costs.) The implementation also benefits by being able to change its algorithms and data structures easily, as described above. The obligation of the client code is to only use the operations provided by the module, and to use them in the ways specified in the contract; that is, the client code must not use any information about the details of the implementation's data structure. The benefit to the client code is that the client code can be written more succinctly, because it is written at a higher-level. In addition, since client code is written at a higher level, it is easier to understand than it would be if it were written in terms of the details of the data structures. When, as is often the case, the client code makes up the bulk of the program, the program, as a whole, may become clearer. The program as a whole is also more easily improved. This may include making it more efficient, because efficiency problems with the program's data structures or algorithms may be more easily isolated and fixed, as they are encapsulated in specific modules.

As a contract, the specification fulfills the usual roles familiar from contracts in the legal system. That is, as long as both sides fulfill their obligations, either party may change anything not specified in the contract. In particular, this includes the details of the implementation. In modern object-oriented programming languages one can use a *class* as the implementation module. If this is done, then the language will automatically prohibit direct access by clients to data structures in a class. This automatically ensures that the client code fulfills some of its obligations. But even if the contract is only enforced by convention, its use in hiding information about data structures is the key idea that allows them to be easily changed.

The Benefits of Information Hiding. If the details of a data structure's format and abstraction mapping are hidden in a module, then when the details need to be changed, one only has to examine and change that module. Because the module is a small part of a larger program, this makes the program easier to change. For example, suppose the calendar manager program has a module that encapsulates the data structure used for storing its user's appointments. Then the details of this data structure can be changed without examining the entire program; only the module in question needs to be examined.

From an economic standpoint, information hiding can be thought of as purchasing an option (5). Recall that, in economics, buying an option on X gives one a guarantee of being able to buy X at a later date for a specified price. Why not just wait until X is needed to buy it? Because either it might not be available at that time or it might cost more than the price specified in the option (plus the cost of the option). In this sense the option purchased by hiding information about the details of a data structure D is the ability to change, at a later date, the details of D's representation without looking at every part of the program. The cost of this option is the cost of writing a module that hides the information about D plus the efficiency loss (if any) that results from manipulating the data structures indirectly through the operations. This cost is justified if the cost of changing the data structure at a later date (i.e., the cost of examining the entire program) is more than the cost of just changing the module (plus the cost of creating and using the module). In a sufficiently large program, the cost of examining the entire program easily outweighs the up-front costs of creating and using a module to hide the data structure's details. Sophisticated compilers can also eliminate much of the cost of manipulating the data structures indirectly.

Other Applications of Information Hiding. This idea of information hiding is not limited to hiding data structure details. For example, it is often used in operating systems to hide information about the detailed management of devices or other resources. In an operating system, a "device driver" provides an interface, with specified operations such as "get" and "put," through which users can manipulate input/output devices. For example, a device driver allows users to manipulate storage media (such as hard disks) from many different manufacturers, without having to rewrite their programs when different media are installed.

The manual that describes the instruction set of a computer can also be seen as a specified interface that allows many different implementations, without requiring new compilers to be written each time a processor's detailed implementation changes. In this case the operations in the interface are the computer's instructions. The assembly language for a computer is a human-readable abstraction of this interface.

Similarly, the reference manual of a programming language also provides a specified interface that insulates the programmer from changes in the computers that the program may run on, and thus provides a degree of independence from the details of these computers. The operations

of this interface are the statements, expressions, and declarations of the programming language, including the ways in which these can be combined.

A programming language may also provide information hiding for its built-in data structures, such as integers and floating point numbers. It does this by specifying the syntax and meaning of the operations that can be used to manipulate them. In this context, information hiding is called *representation independence*, because programs are not dependent on the formats for integers and floating point numbers found on different computers.

Abstract Data Types

Data Types. A family of similar data structures, each with the same format and abstraction mapping, is called a *data type*. The individual data structures in such a family are called *instances* or *objects* of that data type.

Often a program will use many instances of a data type; for example, the calendar manager program might have many instances of the data type Appointment, each one a data structure that represents a separate appointment. It is inconvenient to define a module for each such instance of a data type. Furthermore, if the objects of such a data type are created as the program runs, then it is not possible to define a program module for each object, because the modules are created before run-time.

Because of these problems, it is common to define a module for each data type in a program. In object-oriented programming, this is often taken to an extreme; that is, modules are not created for individual data structures but are only used for data types.

Definition. An *abstract data type* (*ADT*) is a specified interface for modules that encapsulate a data type. This definition can be seen from two points of view. From the implementation side, such a specification permits many different implementation modules. Thus, an ADT can be viewed as an abstraction of all the modules that correctly implement the specified interface. Each such module consists of code that implements the operations; each of these may use a different format and abstraction mapping for the data structures of the objects of the data type that it manipulates.

However, from the client's side, internal differences in the implementation modules do not matter. Since clients can only depend on the operations in the module's interface and their specified behavior, all modules look the same to the client. Hence, from the client's side, an ADT can be thought of as a set of objects and set of operations. This agrees with the definition given earlier.

What connects these two points of view is the ADT's specification. The specification describes the operations in terms of the abstract values of objects. For example, the abstract value of an Appointment object might be a tuple of a date, a time, and a string stating the purpose of the appointment. The operation that changes the length of an Appointment is specified in terms of how it affects the time part of this tuple. On the implementation side, the data structure's abstraction mapping connects the data format to the abstract values; one can check the correctness of the code that implements the operations with respect to the specification by comparing the abstract value at the end of the operation with that specified. On the client side, to produce a desired effect on the abstract value of an object, the client invokes the ADT's operations. An implementation can thus be seen as translating such abstract requests into operations performed at the lower level of its data structure's format.

CREATION AND DESIGN OF ADTS

This section discusses the creation and design of ADTs. It begins with general considerations, and then describes elaborations found in object-oriented programming.

What ADTs to Specify?

How does one decide what ADTs should be used in a program? A basic strategy that is adequate for smaller applications is to consider the nouns one would use to describe the workings of an application program (or a set of similar programs) to be candidates for ADTs. Similarly, the verbs used to describe what happens in such a program are candidates for the operations of such ADTs.

However, both the set of ADTs and the set of operations often need to be expanded with some "internal" types and operations. For example, it may be useful to use a Stack or binary search tree for algorithmic or efficiency purposes, even though these are not nouns used to describe the application. Similarly, it may be useful to have copy or iteration operations, even though these are not verbs used in describing the application.

Finding ADTs in the Calendar Manager Example. For example, consider again the calendar manager program. Its requirements document (or brief overview) might include statements such the following. "The program can record appointments for a person's schedule, both recurring appointments and one-time appointments. Recurring appointments may be scheduled weekly on any given weekday between any two dates, or monthly. Appointments can be scheduled to start at any time of the day, and may last for any length of time. The purpose of an appointment, and additional annotations about it (such as the place where it will occur) can also be noted. Appointments can be easily changed." From such a description, one may note nouns such as: appointment (both recurring and one-time), date, time of day, purpose, and annotation. This may suggest designing types for Appointment, Date, and Time. Perhaps a type Purpose would also be appropriate, but "annotations" might be left as Strings for the moment, as there is little in the way of activities connected with them.

Checking a Design by Assigning Responsibilities. One way to refine and check a planned set of ADTs is to look at how the system's "responsibilities" are partitioned among the various ADTs (6).

A responsibility is a task found in the system's requirements. For example, an Appointment object might take responsibility for remembering an appointment's date, time, length, purpose and other annotations. However, remem-

bering a collection of Appointment objects, and organizing them into a schedule might be the responsibility of a Schedule object.

To double check that the set of ADTs is adequate, one can see if each responsibility in the system's requirements document is assigned to some ADT. Each responsibility should be assigned to a single type of object. Doing so helps keep information about how that responsibility is managed hidden within a single ADT. For example, if changing an appointment's purpose was the responsibility of both a Schedule and an Appointment object, then information about the purpose of appointments could not be hidden entirely within an Appointment (or Schedule) object.

For example, in Figure 1, several ADTs and their responsibilities for the calendar manager program are described. One notable aspect of this list is that it highlighted the responsibility of remembering the place of an Appointment, which I had previously forgotten. It also becomes clear from this set of responsibilities that no ADT is responsible for communicating with the user. Additional ADTs should be designed to handle the user-interface.

Checking a Design by Use-case Scenarios. Another way to check the suitability of a design is by tracing various "use-case scenarios" (7) in the design. A *use-case scenario* is a particular way in which the system will be used. For example, for the calendar manager program, one use case is that a user will check their appointments for the day. Another use-case would be to create a new appointment for a date other than the current date.

The way to use a use-case scenario to check a design is to see how the ADTs in the design are used while playing out the scenario. For example, consider the use-case in which the user checks their appointments for the day. In this case the user-interface (which is so far missing in our design) gets the command to check appointments, it finds the current date (also missing from our design), and then asks the Schedule object for a list of appointments for that date. It might also sort the appointments in order by their starting time, which involves asking each appointment object for its starting time. To display the appointments, it will have to fetch the other information from each: the ending time, the place, and the purpose. This process helps to find missing ADTs, and ensure that all required responsibilities are covered. It also tends to give one a good sense for what operations will be useful for carrying out the required tasks.

Designing Individual ADTs

Once the responsibilities of each ADT are decided, one can think about the detailed design of each ADT. This involves deciding on what information the ADT instances are responsible for holding, and what operations it is able to perform.

Designing Object States. Objects are usually responsible for remembering some information. The client's view of this information is the object's *abstract state*. Another name for this is the object's *abstract value* (8). The abstract value of an object is a mathematical abstraction of its representation in the computer. Recall that the set of abstract values is the target of a data structure's abstraction mapping. In designing an object's state, it is best to focus on this abstract, client-centered point of view, and not plunge into details of the format of some particular data structure. Focusing on abstract values helps ensure that the objects can be described adequately to clients and that the implementation can be changed easily.

As an example, consider designing the state of Time objects. A reasonable set of abstract values might be pairs of integers, each representing the number of hours and minutes past midnight. One can also think of the parts of such a pair as an object's *abstract fields* or *specification variables*. An abstract field does not have to be implemented; such "fields" are only used for specification purposes. Since the set of abstract values is a mathematical concept, the format of an implementation data structure does not need to have two fields, despite the use of two abstract fields in the specification. For example, an implementation might use a single integer field, representing the number of minutes past midnight. The only requirement is that the implementation's data structure have an abstraction mapping that maps the chosen format to the abstract value set in a way that makes the operation implementations meet their specifications (8).

Why not use a single integer as the abstract value of a Time object? That can certainly be done. However, since abstract values are mathematical concepts, convenience and clarity are their most important attributes. One should not worry about "saving storage space" in the design of abstract values. Space is cost-free in mathematics!

Compound objects may have other objects as part of their abstract values. For example, consider Appointment objects. A reasonable choice for their abstract values might be a tuple of a Date object, a Time object, and two Strings (for the appointment's purpose and place). This is clearer (more high level than) specifying the abstract values as a tuple of 5 integers and two Strings, since the 5 integers have to be separately interpreted as representing the year, month, day, hour, and minutes.

To make such specifications of compound abstract values work, it is necessary to use the concept of object identity. We postulate that each object has a unique *identity*, which can be thought of as its address in a computer's memory. Two objects with the same abstract value do not necessarily have the same identity. For example, two appointments with the same date, time, and purpose, may have different identities; one way this can arise is if a user copies an appointment, perhaps to later change its date. Having two separate objects (i.e., with different identities) is important for making this scenario work. Thus, the abstract values of our Appointment objects would be a tuple containing a Date object's identity, a Time object's identity, and the identities of two String objects.

It is sometimes useful to distinguish between a collection of object values and a collection of object identities, as these have different kinds of abstract values. For example, it might be reasonable for the abstract value of a Schedule object to be a set of Appointment object identities. This would allow one to store a copy of an existing Appointment in a Schedule object. By contrast, if the abstract value of a

```
Class   Responsibilities

Schedule        remember all appointments
        add or delete an appointment
        return all appointments for a date
        return all appointments for a date and time
        schedule recurring appointments

Appointment     remember date, time, purpose, and place
        compare date and time to another appointment
        return an appointment like this, but with a new date

Time    remember hours and minutes past midnight
        compare to another time
        return number of hours and minutes to another time
        return the time a given number of hours and minutes from this time

Date    remember year, month, and day
        return the day of the week
        return the next date
        return the date a given number of days in the future
```

Figure 1. Classes and responsibilities for the calendar manager example.

Schedule object were a set of values of Appointment objects (i.e., tuples instead of object identities), it would be impossible to store a separate copy of an Appointment object.

Designing Operations. The operations of an ADT correspond to the actions that are required to carry out its responsibilities. We start by describing various kinds of operations, and then turn to the design of individual operations.

Kinds of Operations. When beginning to specify the operations of an ADT, it is helpful to think of specifying operations of several standard kinds. The first kind is one that creates or initializes objects. Such an operation is commonly called a *constructor*. An operation that creates an object of a type T without using any other objects of type T is called a *primitive constructor*. Some programming languages, such as Smalltalk, allow primitive constructors to create the new objects and return them to the caller. But many languages, such as C++ and Java, take the responsibility upon themselves to create objects, and only allow a primitive constructor to initialize them once they are created. In any case, unless one is specifying an abstract type that is not supposed to have objects created for it, then one wants some primitive constructor operations. (A data type that is not supposed to have objects created for it is sometimes is useful in object-oriented programming, where it can be used as the supertype of some other type of objects. Classes that implement such types are called *abstract classes*.)

Another kind of operation that is common in ADTs is an *observer*. Observer operations are used to extract information from objects. For example, an operation that would extract the date of an Appointment object would be an observer.

The opposite of an observer operation is an operation to change an ADT's objects. Such an operation is called a *mutator*. A mutator changes an object's abstract value. For example, an operation to change the date of an Appointment would be a mutator.

It is also possible to have operations of mixed kinds. Mixed constructors and observers are called *non-primitive constructors*. For example, an operation that takes a Time object and returns a new one that is for a time one hour later would be a non-primitive constructor.

Mixes of observers and mutators are sometimes appropriate, but because observers are used in expressions, and side effects in expressions make reasoning about programs more difficult, such mixes should be approached cautiously. However, if one is designing an ADT for a programming language that has expressions as its primary syntactic unit, like Smalltalk, then such operations may be needed. For example, in Smalltalk, every operation returns a value; hence mutators are also, in a sense, observers. By convention, in Smalltalk every mutator returns the implicit argument of the operation (named "self"). As a more interesting example, a mixed operation might change the ending time of an Appointment object and return its length.

Operations that do a mutation and then return some sort of "status code" may often be more appropriately designed as mutators that may throw exceptions.

Errors and Exceptions. Another consideration in designing the operations of an ADT, is how to handle errors or exceptional cases. *Errors* arise from misuse of an operation, for example, changing the ending time of an appointment to be before its starting time. *Exceptions* are unusual but not completely unexpected events, for example, reading past the end of a file. There are two general strategies for dealing with errors and exceptions:

1. Have the clients check for them, or
2. Have the implementation check for them.

However, for a given ADT, a firm choice between these alternatives should always be made and recorded in the ADT's specification. If a firm choice is not made then both the implementation and client code will, for defensive purposes, always check for such conditions. Such duplicate checks can be a source of inefficiency; hence it is always a good

idea to decide on one of these two strategies.

If clients must check for errors and exceptional situations, then the specification should use preconditions. A *precondition* is a logical predicate that that says what clients must make true when an operation is called. For example, the operation that changes the ending time of an Appointment object might have a precondition that requires the ending time to be later than its starting time. If this is done, then the operation's implementation can assume that the ending time given is later than the starting time; hence it need not check for this error. (If the operation is called with inputs that do not satisfy the precondition, then the implementation is not obligated by the specification to do anything useful; it might even go into an infinite loop or abort the program.) Mathematically, one can view an operation as a relation between its inputs and its outputs. The precondition describes the domain of this relation; that is, the precondition describes what inputs are permitted.

If the clients of an ADT cannot be trusted to do the checking themselves, then it is best to specify that the implementation must check for them. Such *defensive* specifications are useful for general-purpose libraries of ADTs, whom the clients are unknown. A defensive specification mandates that the operation that changes the ending time of an Appointment checks that the ending time is later than the starting time and that the operation must throw an exception if it is not. Client code that needs to validate its own input could catch this exception.

If a type is to have both untrusted and trusted clients, then it may be useful to specify a both kinds of ADT for the same concept. Untrusted clients can use the ADT with the defensive specification. Trusted clients can use the ADT specified with preconditions. Furthermore, the defensive specification can be implemented by simply performing the necessary checks and calling an implementation of the ADT specified with preconditions.

Immutable Objects. For some types, it is reasonable to not have any mutator operations. Objects of such a type are called *immutable*. Since there are no mutators in such a type, an immutable object's abstract value does not change over time. As such, immutable objects often represent pure values. For example, both Time and Date objects are immutable, which matches one's intuition that a specific time or date is an unchanging measure. Such objects typically have many non-primitive constructors. Compound objects, however, are typically *mutable*.

Evaluating ADT Designs. The specification that forms the contract between an ADT's client code and its implementations is a key decision in design that affects future changes and costs. This section considers various criteria for evaluating specifications.

It is most important that the specification hide information by being sufficiently abstract. A specification could fail to be sufficiently abstract by being too close to some particular implementation data structure. One specification, A, is more abstract than another, C, if A has more correct implementations than C does. An implementation is correct if its data structures and algorithms meet each operation's specified contract. Implementations will be different if they have different data structures and algorithms. If A is more abstract than C, then A is also said to be a *higher-level* specification than C. In this case, C is said to be a *refinement* of A. Higher-level contracts, since they allow more implementations, allow the data structures and algorithms used in a program to be more easily changed, because any change from one correct implementation of a given contract to another does not affect client code. A contract that only allowed one implementation would thus not allow any changes to the implementation data structures. Such a low-level specification would be an extreme case of *implementation bias* (9).

Intelligence. In designing the operations of an ADT, it is important to try to make them "intelligent" (10). The intelligence of the operations of an ADT can be estimated by how easy it is for clients to make changes to objects that seem common or likely. That is, the operations should not just fetch and set the information stored in the objects, but ideally should perform more complex services for the client code. One can identify some of these by considering various use-case scenarios. By having the operations do more than simply fetch and set the information, part of the application logic can be handled by the ADT, and the client code will be simplified. However, equally to be avoided is putting all of the application logic into an ADT. Instead, a middle ground is ideal.

As an example, consider the Appointment type. A low-level design might treat an Appointment as a record, with operations to get and set the appointment's date, time, length, place, and purpose. A better design has operations to fetch this information, but would also include operations to: change the length of an existing appointment by some specified amount of time, create hour-long or half-hour long appointments, create a similar appointment for the same time next week, next month, or next year, and compare a given date and time to the appointment's date and time.

Observability and Controllability. One can also think about the design from the perspective of whether it allows access to and control of the state of the objects of that type (or other hidden resources). Good designs are both observable and controllable (11, 12). An *observable* ADT allows its clients to extract the intended information from each object. For example, if the date or time of an appointment object could not be obtained by using the operations of the Appointment type, then the Appointment ADT would not be observable. A *controllable* ADT allows its clients to make the type's objects change into any desired state; that is, the object should be able to be put into a state where its observable information has an arbitrary legal value. For example, if the Appointment data type does not provide a way to make an appointment on February 29 of a leap year, then the type would not be controllable.

The notions of observability and controllability can be made more formal by thinking about the abstract values of a type's objects. For example, the Time ADT will be observable if the number of hours and minutes past midnight can be computed from its objects. It will be controllable if objects can be made with any number of hours between 0

and 23 and minutes between 0 and 59. A type with mutable objects is controllable if each object can be mutated to taking on any abstract state.

Cohesion and Coupling. Each ADT should have a well-defined set of responsibilities. An ADT design is *cohesive* if its responsibilities "hang together" because they are closely related. For example, the Appointment ADT would not be cohesive if, besides its responsibilities for manipulating the date, time, place, and purpose of an appointment, it was also responsible for the low-level details of playing a sound file through a computer's speakers. Although such a responsibility might easily evolve from the ability of an appointment to play an alarm, it has little to do with the other responsibilities of the Appointment ADT. If it were included, it would make the ADT less cohesive.

One can check cohesion more carefully when the abstract values of objects are specified using abstract fields. To do this, one checks whether each operation of an ADT reads or writes either just one or every abstract field. There should also be at least one operation that uses all the abstract fields. Operations that use just one abstract field are okay, as are operations that use all the fields. However, if an object has three abstract fields and some operation only deals with two of them, then the abstraction fails the cohesion check. The solution is to either remove the offending operation, or to split the abstraction into different ADTs.

Coupling is a measure of how much one module is dependent on another module. Strong coupling between two modules means that when one is changed, then the other should also be changed, or at least checked to see if it needs to be changed. Hence, it is best to avoid strong coupling. Strong coupling may occur when two modules use the same global variables. For example, if the Schedule ADT and the Appointment ADT both used a global variable that holds the current date, then this would provide unnecessarily strong coupling between them. To avoid strong coupling, it is best to pass such information to the relevant operations as an argument, or to have both call some other operation, instead of using global variables.

Parameterized (Generic) ADTs

Many ADTs can be generalized to be more reusable by parameterization. For example, consider an ADT for a sequence of 24 hourly temperature readings, HourlyReadings. The operations of this type would include: getting a reading for a given hour ("fetch"), setting a given hour's reading to a given temperature ("store"), and perhaps finding the average, minimum, and maximum temperature in that period. Thus far we have been treating the temperature readings as integers. But suppose that for various applications, we need different amounts of precision. One application might need to keep readings to the nearest integer degree, but others might need floating point numbers. Clearly, the idea of the HourlyReadings ADT, which has objects with abstract values that are a sequence of 24 hourly temperature readings, is applicable to both cases. Making two separate ADTs for such closely related concepts would cause duplication of implementation code, which would cause maintenance problems.

Instead of duplicating code in such cases, it is better to abstract from the family of related ADTs by making a *generic* or *parameterized* ADT that, when instantiated, can generate each member of the family. Such an ADT can be thought of a function that takes a type and produces an ADT, and so it is an ADT *generator* in the sense that when the parameters are supplied it generates an ADT. In the specification, the type parameters are thought of as a fixed but arbitrary types. The usual notation in specifications is talk about a generic instance of the generator, such as HourlyReadings[T], where "T" is the name of the abstraction's formal type parameter. Clients instantiate such a type generator, making a type, by passing it an actual type parameter. For example, HourlyReadings[int] would be an ADT which keeps hourly temperature readings as integers, while HourlyReadings[float] would keep them as floating point numbers.

Often some operations are required of the types of objects used as actual type parameters. For example, HourlyReadings[T] might require that the type T have the usual arithmetic operations of addition and division (by an integer), which would allow the average of the readings to be computed. The signatures and behavior of these operations should be stated in the specification of the ADT generator (13).

In programming languages that support parameterized ADTs, such as C++ and Ada, the use of explicit type parameters extends to operations and subroutines. For example, a parameterized function, such as sort[T], can be thought of as a function generator. This style of programming leads to *parametric polymorphism*, in which one piece of code can operate on many different kinds of data. Object-oriented programs exhibit a different kind of polymorphism, which comes from message passing and subtyping.

Refinements for Object-Oriented Programs

Message Passing (Dynamic Dispatch). To explain the message passing mechanism of object-oriented (*OO*) languages, we first present the problem it solves.

In a non-OO language, one must know the exact type of an object in order to apply operations to its objects. For example, in Ada 83, one would write "Appointment'get_start(myAppt)" to get the starting time from the appointment named "myAppt". This uses the operation get_start, found in the program's implementation of Appointment.

The need to know the exact type of an object makes it more difficult to change programs. For example, suppose that, after writing the first version of the calendar manager program, one adds a second type of appointment, RecurringAppt. In Ada 83 one would have to write "RecurringAppt'get_start(myRecAppt)" to extract the starting time from an object, myRecAppt, of this type. Since the program needs to manipulate both types of appointments at once, and since it cannot know which to expect, it must use a variant record data structure, which can hold either type. A variant record object has an abstract state that consists of two abstract fields: a type tag and an object whose type depends on the type tag. For example, we might have a variant record type, Appt, with two possible

type tags, "Appt_tag" and "Recurring_tag." When the type tag is "Appt_tag", the object stored is an Appointment object; when it is "Recurring_tag", the object is a RecurringAppt object.

Using a variant record, "appt", one might write something like the code in Figure 2 to extract the start time of appt. The code tests the type tag of appt, then extracts the start time from the object by dispatching to the appropriate type's get_start operation. Not only is this tedious, but it makes the program difficult to change. Imagine what happens if another type of appointment is added to the program, then all such dispatching case statements must be found and updated.

The message passing mechanism in OO languages is designed to automate this kind of code. Thus it also makes adding new types that are similar to existing types easier. In essence, all objects in an OO language are like the variant record objects in non-OO languages. That is, objects in an OO language allow one to find their exact type at run-time. Each object contains a pointer to some language-specific data structure that represents the class at run-time; typically this includes the name of the class, and the code for the instance methods of that class. An *instance method* or *instance operation* is a method that takes an existing object as an argument. The primitive constructors, and other such operations that are still called directly, instead of being dispatched to indirectly, are called *class methods* or *class operations*. (Class methods need not be stored in the class.)

Calls to instance methods are dispatched to the appropriate code based on the run-time type of the object involved. Abstractly, the view presented to the programmer is that objects contain the operations that work on them. In a singly-dispatched language, like Smalltalk, C++, and Java, instance methods are dispatched on what would otherwise be the first argument of an operation. This argument is sometimes called the *receiver*, or the *implicit* or *default argument* to an instance operation. For example, instead of writing "Appointment'get_start(myAppt)", one writes "myAppt.get_start()", and the object myAppt is the implicit argument. Within the code for an instance method, the default argument is named "self" (in Smalltalk) or "this" (in C++ and Java). The syntax "myAppt.get_start()" embodies the idea that, to invoke a method, one first extracts the method from the object, and then calls it. This mechanism is called *dynamic dispatch* or *message passing*. When the term "message passing" is used, the name of the method invoked ("get_start") and its arguments are thought of as a *message*, which is sent to the object, to ask it to do something. An invocation such as "myAppt.get_start()" is also thought of as sending the message "get_start()" to the object myAppt. (Note, however, that no concurrency or distribution is necessarily involved.)

Subtype Polymorphism. Message passing allows client code to be written that is independent of the exact types of objects. That is, client code that sends messages to objects is polymorphic, since it can work on objects of different types. This kind of polymorphism is called *subtype polymorphism*. Like parametric polymorphism, it helps make code more general and resistant to change. It is related to parametric polymorphism in that message passing code uses operations of types that are passed to it. However, in subtype polymorphism the types are passed to the code at run-time, in the objects being manipulated, instead of being passed at compile-time, separately from the objects, as in parametric polymorphism.

Message passing and subtype polymorphism focus attention on the instance methods of objects and downplay the role of class methods. Consider the set of all messages that an object can be sent. This set forms the object's *instance protocol* (14), and is, in essence, a collection of signature information. It corresponds to a Java interface.

Suppose that objects of type S have an instance protocol that includes all the messages in the instance protocol of type T. Then S objects can be manipulated as T objects without encountering any type errors. This means that the type S is a *subtype* of T. In OO languages, if S is a subtype of T, then objects of type S can be assigned to variables of type T, passed as parameters where T objects are expected, and returned from functions and methods that are declared to return T objects. This is safe because any message that is sent to an object that is supposed to have type T is in the protocol of S objects. For example, RecurringAppt is a subtype of Appointment if any message that can be sent to an Appointment object can be sent to a RecurringAppt object without encountering a type error.

Behavioral Subtyping. Client code does not just depend on the absence of type errors; it also depends on the behavior of objects that it manipulates. An ADT S is a *behavioral subtype* of T if each S object behaves like some T object when manipulated according to the specification of T's instance protocol (18). In essence, the objects of a behavioral subtype have to obey the specifications of all the instance methods of their supertypes (4–17).

With behavioral subtyping, message passing becomes truly useful, because one can reason about, and test, the correctness of client code in a modular fashion (19). For example, if one can show that an operation that manipulates an Appointment object must accomplish a certain task, based on the specification of the type Appointment, then this conclusion will be valid for all behavioral subtypes of Appointment. Hence this client code will not only be able to manipulate objects of behavioral subtypes—it can do so predictably. It is this property that allows OO programs to be developed in an evolutionary manner.

Inheritance (Subclassing). Strictly speaking, the inheritance mechanism of OO languages has little to do with abstract data types. Recall that a class is a program module that can be used to implement an ADT in an OO language. *Inheritance* is a mechanism that allows one to implement a class by stating how it differs from some other class. A class defined by inheritance is called a *subclass* or *derived class*. A subclass may have (in some languages, like C++) more than one *superclass* or *base class* from which it is derived. A subclass will inherit fields (data structures) and method code from its superclasses. However, code for class methods such as primitive constructors is not inherited by subclasses. A subclass can also add new fields and methods. It is also possible to add some behavior to a superclass's

```
case appt.type_tag of
  Appt_tag => return Appointment'get_start(appt.appointment);
  Recurring_tag => return RecurringAppt'get_start(appt.recurring);
end
```

Figure 2. Ada 83-like pseudo code for dispatching based on the type code in a variant record object.

method without rewriting it completely.

Because of these properties, it is often convenient to implement a behavioral subtype by using inheritance to derive a subclass of a class that implements the supertype. For example, one way to implement RecurringAppt as a behavioral subtype of Appointment, is to use a subclass of a class that implements Appointment.

Inheritance Is Not Behavioral Subtyping. However, it is important to realize that inheritance does not necessarily produce classes that implement behavioral subtypes. For example, if the RecurringAppt class redefines the get_start() method to go into an infinite loop, or to always return midnight, then RecurringAppt would not correctly implement a behavioral subtype of Appointment. Indeed, in C++ one can use "private inheritance" to make a subclass that does not produce a subtype. When using C++ or a similar language, one should either make subtypes that are behavioral subtypes, or use private inheritance. Doing so ensures that the type system's checking enforces not only subtyping but also behavioral subtyping. (Of course, the type system will not prove that one type is a behavioral subtype of another, but it will track the declarations given to it.)

RELATED TOPICS

Specifications can be written either formally, using some mathematically well-defined notation, or informally, in English. Informal specifications suffer from ambiguity, although they are useful for giving overviews and motivation and in situations where the cost of ambiguity and misunderstanding is not high. However, even if you do not use formal specifications, studying them will help you in being more precise in your use of informal techniques. See Specification Languages.

Formal specifications can be used to formally verify the correctness of an implementation of an ADT (8). See ***Programming theory***.

Some form of specification is needed for testing (validating) code. *Black-box* testing of implementations of an ADT is entirely based on the ADT's specification; the specification is critical for telling what the results of a test should be.

ADTs that are used in concurrent or distributed programs have to deal with issues such as locking, to prevent race conditions such as two clients extracting the same job from a queue. One way to do this is to specify that the ADT makes some client operations wait for some condition to be made true by other clients (20, 21). A sampling of formal specification techniques for concurrent and real-time systems is found in the book *Formal Methods for Industrial Applications* (22). See also ***Real-time systems***.

BIBLIOGRAPHY

1. B. W. Boehm, *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ, 1981.
2. D. L. Parnas, On the criteria to be used in decomposing systems into modules. *Commun. of the ACM*, **15**(12): 1053–1058, 1972.
3. B. Liskov and J. Guttag, *Abstraction and Specification in Program Development*. Cambridge, MA: MIT Press, 1986.
4. B. Meyer,Applying " Design by Contract". *Computer*, **25**(10): 40–51, 1992.
5. K. J. Sullivan, P. Chalasani, S. Jha and V. Sazawal," Software Design as an Investment Activity: A Real Options Perspective." InL. Trigeorgis (ed.), *Real Options and Business Strategy:Applications to Decision Making*, pp. 215–261. Risk Books, London, England, 1999.
6. R. Wirfs-Brock, B. Wilkerson, and L. Weiner, *Designing Object-Oriented Software*. Englewood Cliffs, NJ: Prentice-Hall, 1990.
7. I. Jacobson, *Object-Oriented Software Engineering:A Use Case Driven Approach*. Reading, MA: Addison-Wesley, 1994.
8. C. A. R. Hoare, Proof of correctness of data representations. *Acta Informatica*, **1**(4): 271–281, 1972.
9. C. B. Jones, *Systematic Software Development Using VDM*. Second ed., Englewood Cliffs, NJ: Prentice-Hall 1990.
10. A. J. Riel, *Object-Oriented Design Heuristics*. Reading MA: Addison-Wesley, 1996.
11. D. L. Parnas and D. P. Siewiorek, Transparency in the Design of Hierarchically Structured Systems. *Communications of the ACM*, **18**(7): 401–408, 1975.
12. W. F. Ogden, M. Sitaraman, B. W. Weide, and S. H. Zweben, Part I: The RESOLVE Framework and Discipline — A Research Synopsis. *ACM SIGSOFT Software Engineering Notes*, **19**(4): 23–28, 1994.
13. G. W. Ernst, R. J. Hookway, J. A. Menegay, and W. F. Ogden, Modular Verification of Ada Generics. *Computer Languages*, **16**(3/4): 259–280, 1991.
14. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.
15. B. Meyer, *Object-Oriented Software Construction.* Second ed., Englewood Cliffs, NJ: Prentice Hall, 1997.
16. A. Wills, Refinement in Fresco. InK. Lano andH. Houghton (eds.), *Object-Oriented Specification Case Studies*, pp. 184-201, Englewood Cliffs, NJ: Prentice Hall, 1994.
17. K. K. Dhara and G. T. Leavens,Forcing behavioral subtyping through specification Inheritance. In*Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pp. 258–267, IEEE Computer Society Press, 1996.
18. B. Liskov and J. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, **16**(6): 1811–1841, 1994.
19. G. T. Leavens and W. E. Weihl, Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, **32**(8): 705–778, 1995.
20. B. Liskov and W. Weihl, Specifications of distributed programs. *Distributed Computing*, **1**: 102–118, 1986.

21. E. Rodríguez, M. B. Dwyer, C. Flanagan, J. Hatcliff, G. T. Leavens, and Robby, Extending JML for Modular Specification and Verification of Multi-Threaded Programs. InAndrew P. Black (ed.), *Proceedings ECOOP 2005 — Object-Oriented Programming 19th European Conference, Glasgow, UK*, pp. 551–576. Volume 3586 of *Lecture Notes in Computer Science*, Springer-Verlag, 2005.
22. J.-R. Abrial, E. Börger, andH. Langmaack (eds.). *Formal Methods for Industrial Applications: Specifying and programming the Steam Boiler Controller*. Berlin: Springer-Verlag, 1996.

Reading List

Meyer's book, *Object-Oriented Software Construction (15) is a comprehensive discussion of object-oriented techniques with an extensive bibliography. This book also discusses more formal specification techniques for abstract data types.*

A quick introduction to object-oriented design can be found in the book Designing Object-Oriented Software (6). Programmers should read the widely acclaimed Design Patterns book (14).

GARY T. LEAVENS
Department of Computer Science, Iowa State University, Ames, IA, 50011-1041