Introduction to ADTs
6.170, Lecture 7
Fall 2005

This lecture is the first in a series on data abstraction. It explains the motivation for data abstraction, and the key ideas of characterization by operations, and representation independence. In subsequent lectures, we'll delve more deeply into the theory of abstract types.

7.1 User-Defined Types

In the early days of computing, a programming language came with built-in types (such as integers, booleans, strings, etc.) and built-in procedures, eg. for input and output. Users could define their own procedures: that's how large programs were built.

A major advance in software development was the idea of abstract types: that one could design a programming language to allow user-defined types too. This idea came out of the work of many researchers, notably Dahl (the inventor of the Simula language), Hoare (who developed many of the techniques we now use to reason about abstract types), Parnas (who coined the term 'information hiding' and first articulated the idea of organizing program modules around the secrets they encapsulated), and here at MIT, Barbara Liskov and John Guttag, who did seminal work in the specifica-tion of abstract types, and in programming language support for them (and developed 6170!).

The key idea of data abstraction is that a type is characterized by the operations you can perform on it. A number is something you can add and multiply; a string is something you can concatenate and take substrings of; a boolean is something you can negate, and so on. In a sense, users could already define their own types in early programming languages: you could create a record type date, for example, with integer fields for day, month and year. But what made abstract types new and different was the focus on operations: the user of the type would not need to worry about how its values were actually stored, in the same way that a programmer can ignore how the compiler actually stores integers. All that matters is the operations.

In Java, as in many modern programming languages, the separation between built-in types and user-defined types is a bit blurry. The classes in `java.lang`, such as `String` and `Integer` are regarded as built-in (especially `String` since it has its own special syntax for literals); whether you regard all the collections of `java.util` as built-in is less clear (and not very important anyway). Java complicates the issue by having primitive types that are not objects. The set of these types, such as `int` and `boolean`, cannot be extended by the user.

7.2 Classifying Types and Operations

Types, whether built-in or user-defined, can be classified as *mutable* or *immutable*. The objects of a mutable type can be changed: that is, they provide operations which when executed cause the results of other operations on the same object to give different results. So `Vector` is mutable, because you can call `addElement` and observe the change with the `size` operation. But `String` is immutable, because its operations create new string objects rather than changing existing ones. Sometimes a type will be provided in two forms, a mutable and an immutable form. `StringBuffer`, for example, is a mutable version of `String` (although the two are certainly not the same Java type, and are not interchangeable).

Immutable types are generally easier to reason about. Aliasing is not an issue, since sharing cannot be observed. And sometimes using immutable types is more efficient, because more sharing is possible. But many problems are more naturally expressed using mutable types, and when local changes are needed to large structures, they tend to be more efficient. Often, when you design an abstract type, it won't be immediately obvious whether the type should be mutable or immutable. You should generally err on the side of making it immutable; novices make much too much use of mutable types (and pay the price in complexity and poor performance).

The operations of an abstract type are classified as follows:

- *Creators* create new objects of the type. A creator may take an object as an argument, but not an object of the type being created.
- *Producers* create new objects from old objects. The `concat` method of `String`, for example, is a producer: it takes two strings and produces a new one representing their concatenation.
- *Mutators* change objects. The `addElement` method of `Vector`, for example, mutates a vector by adding an element to its high end.
- *Observers* take objects of the abstract type and return objects of a different type. The `size` method of `Vector`, for example, returns an integer.

We can summarize these distinctions schematically like this:

7.2.1 `creator: t -> T`

7.2.2 `producer: T, t -> T`

7.2.3 `mutator: T, t -> void`

7.2.4 `observer: T, t -> t`

These show informally the shape of the signatures of operations in the various classes. Each `T` is the abstract type itself; each `t` is some other type. An occurrence of a `t` on the left hand side is short for zero or more occurrences of other types (which may differ from one another); an occurrence of `T` on the left is short for one or more occurrences of the abstract type.

For example, string concatenation has the signature

7.2.5 `concat: String, String -> String`

matching the shape

7.2.6 `concat: T, T -> T`

7.2.7 and is therefore a producer. The size method has the signature

7.2.8 `size: String -> int`

matching the shape

7.2.9 `size: T -> t`

and is therefore an observer.

Remember when examining a method to include the receiver argument; a method like `concat` appears in Java with a signature with *one* argument and one result because the receiver is implicit.

This classification gives some useful terminology, but it's not perfect. In many of the Java collection classes (in `java.util`), for example, mutators often return a reference to the abstract object or to a previous value held inside it. and may thus also be classified as producers or observers. Some people use the term 'producer' to imply that no mutation occurs.

Note that our terms don't line up neatly with Java notions. Constructors and creators are not the same, for example: a copy constructor is not a creator (because it takes a value of the abstract type), and a factory method is a creator but not a constructor. This can be a bit confusing, but it's useful to make more general distinctions than Java, and not to be tied to one programming language.

There's a bit of a terminological mess surrounding iteration. Some languages, such as CLU, provided a special kind of procedure for iteration that returned elements one at a time, and retained its program counter between calls; it was called as *iterator*. In Java, an *iterator* is an object that provides methods for iterating over a collection. The term *generator* used to be a synonym of *producer*, but our course text uses it to refer to a method that returns an iterator!

7.3 Example: List

Let's look at an example of an abstract type: the list. A list, in Java, is like an array. It provides methods to extract the element at a particular index, and to replace the element at a particular index. But unlike an array, it also has methods to insert or remove an element at a particular index. In Java, `List` is an *interface* with many methods, but for now, let's imagine it's a simple class with the following methods:

```
7.3.1   public class List {
7.3.2      public List ();
7.3.3      public void add (int i, Object e);
7.3.4      public void set (int i, Object e);
7.3.5      public void remove (int i);
7.3.6      public int size ();
7.3.7      public Object get (int i);
7.3.8      }
```

The add, set and remove methods are mutators; the size and get methods are observers. It's common for a mutable type to have no producers (and an immutable type certainly cannot have mutators).

To specify these methods, we'll need some way to talk about what a list looks like. We do this with the notion of *specification fields* or *abstract fields*. You can think of an object of the type as if it had these fields, but remember that they don't actually need to be fields in the implementation, and there is no requirement that a specification field's value be obtainable by some method. In this case, we'll describe lists with a single specification field,

```
7.3.9   seq [Object] elems;
```

where for a list l, the expression l.elems will denote the sequence of objects stored in the list, indexed from zero. Now we can specify some methods:

```
7.3.10  public void get (int i);
7.3.11  // throws
7.3.12  //    IndexOutOfBoundsException if i < 0 or i > length (this. elems)
7.3.13  // returns
7.3.14  //    this.elems [i]

7.3.15  public void add (int i, Object e);
7.3.16  // modifies this
7.3.17  // effects
7.3.18  //    throws
7.3.19  //IndexOutOfBoundsException if i < 0 or i > length (this. elems)
7.3.20  //    NullPointerException if e is null
7.3.21  //    else this.elems = this_pre.elems [0..i-1] ^ <e> ^ this_pre.elems [i..]

7.3.22  public void set (int i, Object e);
7.3.23  // modifies this
7.3.24  // effects
7.3.25  //    throws IndexOutOfBoundsException if i < 0 or i >= length (this. elems)
7.3.26  //    NullPointerException if e is null
7.3.27  //    else this.elems [i] = e and this.elems unchanged elsewhere
```

In the postcondition of add, I've used s[i..j] to mean the subsequence of s from indices i to j, and s[i..] to mean the suffix from i onwards. The caret means sequence concatenation. So the postcondition says that, when the index is in bounds or one above, the new element is 'spliced in' at the given index.

7.4 Designing an Abstract Type

Designing an abstract type has two aspects: specification (ie, choosing operations and determining how they should behave) and implementation (selecting a representation). The specification aspect is almost always the more challenging and important, because it's less easily changed later.

7.4.1 Specification Design

A few rules of thumb:

- It's better to have a few, simple operations that can be combined in powerful ways than lots of complex operations.
- Each operation should have a well-defined purpose, and should have a coherent behaviour rather than a panoply of special cases.
- The set of operations should be *adequate*; there must be enough to do the kinds of computations clients are likely to want to do. A good test is to check that every property of an object of the type can be extracted. For example, if there were no get operation, we would not be able to find out what the elements of the list are. Basic information should not be inordinately difficult to obtain. The size method is not strictly necessary, because we could apply get on increasing indices, but this is inefficient and inconvenient.
- The type may be *generic*: a list or a set, or a graph, for example. Or it may be *domain-specific*: a street map, an employee database, a phone book, etc. But it should not mix generic and domain-specific features.
- Although functionality is your first concern, efficiency is important too. You need to ensure that the operations allow a client of the type to work efficiently. This impacts the choice of operations in two ways: you have to include the right operations so the client doesn't have to do something convoluted, and you must design the operations so they can be implemented efficiently.

7.4.2 Implementation: Choice of Representation

So far, we have focused on the characterization of abstract types by their operations. In the code, a class that implements an abstract type provides a representation: the actual data structure that supports the operations. The representation will be a collection of

fields each of which has some other Java type; in a recursive implementation, a field may have the abstract type but this is rarely done in Java.

Linked lists are a common representation of lists, for example. The following object model shows a linked list implementation similar (but not identical to) the LinkedList class in the standard Java library:

The list object has a field header that references an Entry object. An Entry object is a record with three fields: next and prev which may hold references to other Entry objects (or be null), and element, which holds a reference to an element object. The next and prev fields are links that point forwards and backwards along the list. In the middle of the list, following next and then prev will bring you back to the object you started with. Let's assume that the linked list does not store null references as elements. There is always a dummy Entry at the beginning of the list whose element field is null, but this is not interpreted as an element.

The following object diagram shows a list containing two elements::

Note how the last entry points to the header entry: this is why the next and prev fields are marked as one-to-one in the object model.

Another, different representation of lists uses an array. The following object model shows how lists are represented in the class ArrayList in the standard Java library:

Here's a list with two elements in this representation:

These representations have different merits. The linked list representation will be more efficient when there are many insertions at the front of the list, since it can splice an element in and just change a couple of pointers. The array representation has to bubble all the elements above the inserted element to the top, and if the array is too small, it may need to allocate a fresh, larger array and copy all the references over. If there are many get and set operations, however, the array list rep-resentation is better, since it provides random access, in constant time, while the linked list has to perform a sequential search.

Another important consideration is ease of implementation. The array representation is probably easier to implement (and certainly easier to get right). It uses fewer objects and has fewer invariants. We'll look at rep invariants in detail in a later lecture.

We may not know when we write code that uses lists which operations are going to predominate. The crucial question, then, is how we can ensure that it's easy to change representations later.

7.5 Representation Independence

Representation independence means that the use of an abstract type is independent of its representation, so that changes in representation have no effect on code outside the abstract type itself. Let's examine what goes wrong if there is no independence, and then look at some language mechanisms for helping ensure it.

Suppose we know that our list is implemented as an array of elements. We're trying to make use of some code that creates a sequence of objects, but unfortunately, it creates a `Vector` and not a `List`. Our `List` type doesn't offer a producer that does the conversion from `Vector`. But we discover that `Vector` has a method `copyInto` that copies the elements of the vector into an array. Relying on our knowledge that `List` is represented as an array with the field `elementData`, we now write:

7.5.1
```
List l = new List ();
```
7.5.1
```
v. copyInto (l. elementData);
```

What a clever hack! Like many hacks it works for a little while. Suppose the implementor of the `List` class now changes the representation from the array version to the linked list version. Now the list `l` won't have a field `elementData` at all, and the compiler will reject the program. This is a failure of representation independence: we'll have to change all the places in the code where we did this.

Having the compilation fail is not such a disaster. It's much worse if it succeeds and the change has still broken the program. Here's how this might happen.

In general, the size of the array will have to be greater than the number of elements in the list, since otherwise it would be necessary to create a fresh array every time an element is added or removed. So there must be some way of marking the end of the segment of the array containing the elements. Suppose the implementor of the list has designed it with the convention that the segment runs to the first null reference, or to the end of the array, whichever is first. Unluckily, our hack works under these circumstances.

Now our implementor realizes that this was a bad decision, since determining the size of the list requires a linear search to find the first null reference. So she adds a `size` field and updates it when any operation is performed that changes the list. This is much better, because finding the size now takes constant time. It also naturally handles null references as list elements.

Now our clever hack is likely to produce some strange behaviours. The list we created has a bad size field: it will hold zero however many elements there are in the list (since we updated the array alone, and not the `size` field). Get and set operations will probably fail (assuming that they reject an index greater than the size), and a call to `size` will certainly return the wrong value.

Here's another example. Suppose we have the linked list implementation, and we include an operation that returns the Entry object corresponding to a particular index.

```
7.5.3   public Entry getEntry (int i)
```

Our rationale is that if there are many calls to set on the same index, this will save the linear search of repeatedly obtaining the element. Instead of

```
7.5.4   l. set (i, x);
7.5.5   ...
7.5.6   l. set (i, y);
```

we can now write

```
7.5.7   Entry e = l. getEntry (i);
7.5.8   e. element = x;
7.5.9   ...
7.5.10  e. element = y;
```

This also violates representation independence, because when we switch to the array representation, there will no longer be Entry objects. This is the obvious problem, but there are more subtle problems. Suppose we assign null to the element of an entry. The specification of add (see 6.3.15 above) rejects attempts to insert a null value as an element. The implementor may rely on this, and find the header entry on a traversal around the ring by checking to see whether the element field is null. This assignment causes an entry that is not the header to have a null element field, so the code is now likely to break.

We can illustrate the problem with a 'module dependency diagram':

There should only be a dependence of the client type Client on the List class (and on the class of the element type, in this case Object, of course). The dependence of Client on Entry is the cause of our problems. Returning to our object model for this representation, we want to view the Entry class and its associations as internal to List. We can indicate this informally by colouring the parts that should be inaccessible to a client red (if you're reading a black and white printout, that's Entry and all its incoming and outgoing arcs), and by adding a specification field elems that hides the representation:

7.6 Language Mechanisms

To prevent access to the representation, we can make the fields private. This eliminates the array hack; the statement

```
7.6.1   v. copyInto (l. elementData);
```

would be rejected by the compiler because the expression l.elementData would illegally reference a private field from outside its class.

The Entry problem is not so easily solved. There is no direct access to the representation. Instead, the List class returns an Entry object that belongs to the representation. This is called *representation exposure*, and it cannot be prevented by language mechanisms alone. We need to check that references to mutable components of the representation are not passed out to clients, and that the representation is not built from mutable objects that are passed in. In the array representation for example, we can't allow a constructor that takes an array and assigns it to the internal field.

Interfaces provide another method for achieving representation independence. In the Java standard library, the two representations of lists that we discussed are actually distinct classes, ArrayList and LinkedList. Both are declared to extend the List interface. List declares only the operations and doesn't give representations or code:

```
7.6.2  public interface List {
7.6.3      void add (int i, Object e);
7.6.4      void set (int i, Object e);
7.6.5      void remove (int i);
7.6.6      int size ();
7.6.7      Object get (int i);
7.6.8      }
```

and the two classes are declared as implementations of List:

```
7.6.9  public class LinkedList implements List {
7.6.10     private Entry header;
7.6.11     ...
7.6.12     }

7.6.13 public class ArrayList implements List {
7.6.14     private Object[] eltData;
7.6.15     ...
7.6.16     }
```

Whenever possible, clients refer only to the List interface, so the classes containing the representations are not accessible. Here's a standard idiom for creating and manipulating objects:

```
7.6.17 List l = new LinkedList ();
7.6.18 ...;
7.6.19 l.add (i, e);
```

Note that the interface can't be used to *construct* the object; an interface has no constructors, and it is at the point of creation that we need to specify the implementation. But we have carefully declared the result of the constructor call as a List and not a LinkedList. A subsequent reference to l.header would now be illegal, even if the field were declared public.

The depedendences on the concrete classes due to constructor calls are localized as much as possible, but sometimes we would like to mitigate them further. The Factory pattern, which we will discuss later in the course, addresses this particular problem.

7.7 Summary

Abstract types are characterized by their operations. Representation independence makes it possible to change the representation of a type without its clients being changed. In Java, access control mechanisms and interfaces can help ensure independence. Representation exposure is trickier though, and needs to be handled by careful programmer discipline.