

Lecture 8 Summary

Abstract Types

Spring 2002: Devadas/Guttag

8.1. Introduction

In this lecture, we look at a particular kind of dependence, that of a client of an abstract type on the type's representation, and see how it can be avoided. We also discuss briefly the notion of specification fields for specifying abstract types, the classification of operations, and the tradeoff of representations.

8.2. 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. 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 `Integer` and `Boolean` are built-in; 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.

8.3. 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.

The operations of an abstract type are classified as follows:

- *Constructors* create new objects of the type. A constructor may take an object as an argument, but not an object of the type being constructed.
- *Producers* create new objects from old objects; the terms are synonymous. 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:

```
constructor: t -> T
producer: T, t -> T
mutator: T, t -> void
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. In general, when a type is shown on the left, it can occur more than once. For example, a producer may take two values of the abstract type; string *concat* takes two strings. The occurrences of *t* on the left may also be omitted; some observers take no non-abstract arguments (eg, *size*), and some take several.

This classification gives some useful terminology, but it's not perfect. In complex data types, there may be operations that are producers and mutators, for example. Some people use the term 'producer' to imply that no mutation occurs.

8.4. 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:

```
public class List {
    public List ();
    public void add (int i, Object e);
    public void set (int i, Object e);
    public void remove (int i);
    public int size ();
    public Object get (int i);
}
```

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*. 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,

```
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:

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

public void add (int i, Object e);
// modifies this
// effects
//   throws IndexOutOfBoundsException if i < 0 or i > length (this.elems)
//   else this.elems' = this.elems [0..i-1] ^ <e> ^ this.elems [i..]

public void set (int i, Object e);
// modifies this
// effects
//   throws IndexOutOfBoundsException if i < 0 or i >= length (this.elems)
//   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.

8.5. Designing an Abstract Type

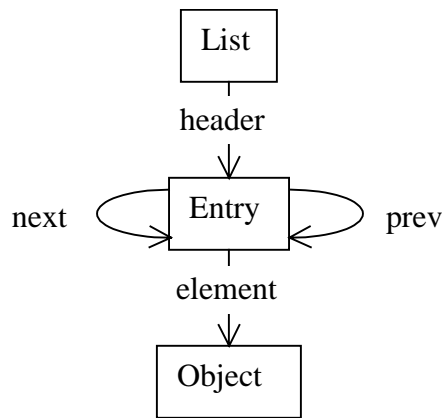
Designing an abstract type involves choosing good operations and determining how they should behave. A few rules of thumb:

- 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 behavior 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.

8.6. Choice of Representations

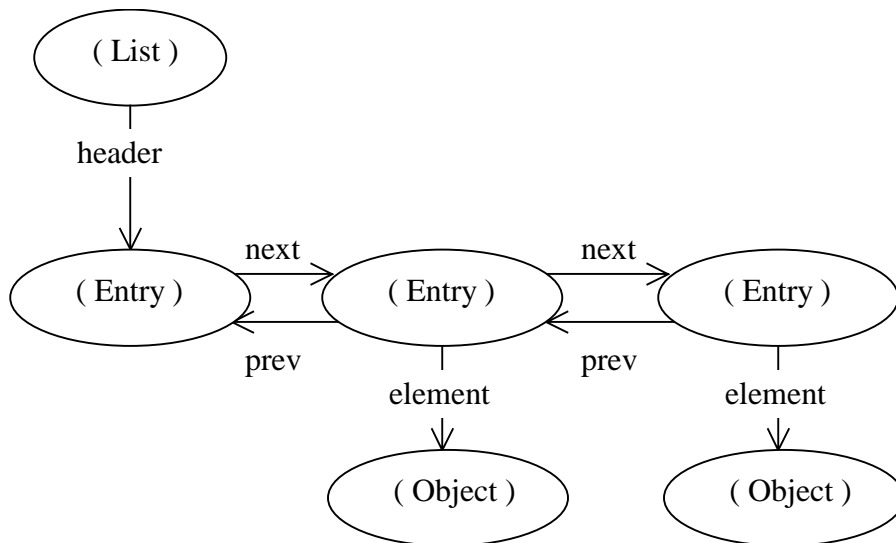
So far, we have focused on the characterization of abstract types by their operations. In the code, a class that implements an abstract type provides a *representation*: the actual data structure that supports the operations. The representation will be a collection of fields each of which has some other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

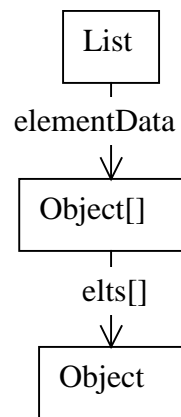


The list object has a field *header* that references an *Entry* object. An *Entry* object is a record with three fields: *next* and *prev* which may hold references to other *Entry* objects (or be null), and *element*, which holds a reference to an element object. The *next* and *prev* fields are links 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.

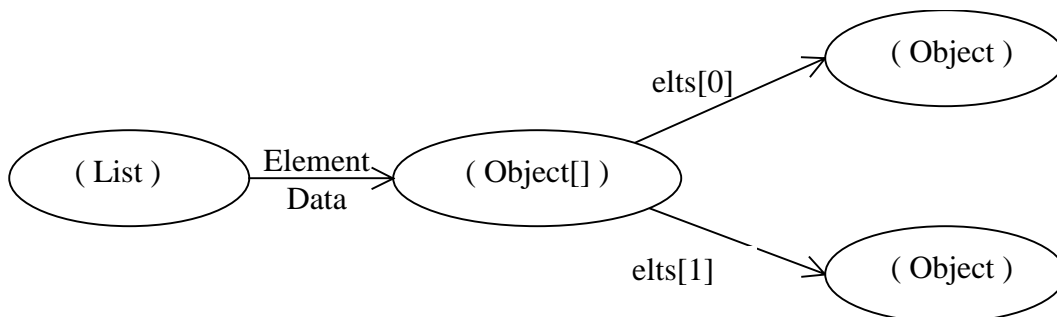
The following instance diagram shows a list containing two elements:



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 its 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 representation is better, since it provides random access, in constant time, while the linked list has to perform a sequential search.

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.

8.7. Representation Independence

Representation independence means ensuring 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

constructor that does the conversion. We discover that `Vector` has a method `copyInto` that copies the elements of the vector into an array. Here's what we now write:

```
List l = new List ();  
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. Luckily (or actually 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 he 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 (and that's why it's what the Java `LinkedList` implementation does).

Now our clever hack is likely to produce some buggy behaviors whose cause is hard to track down. 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). `Get` and `set` operations will appear to work, but the first call to `size` will fail mysteriously.

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.

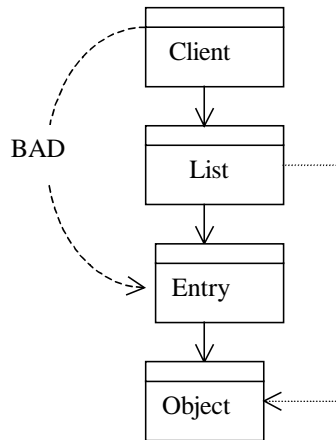
```
public Entry getEntry (int i)
```

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

```
l.set (i, x); ... ; l.set (i, y)
```

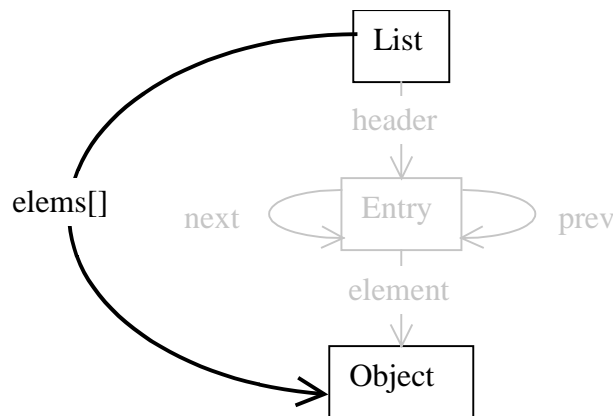
we can now write

```
Entry e = l.getEntry (i);  
e.element = x;  
...  
e.element = y;
```



This also violates representation independence, because when we switch to the array representation, there will no longer be *Entry* objects. We can illustrate the problem with a module dependency diagram shown below.

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:



In the *Entry* example we have exposed the representation. A more plausible exposure, which is quite common, arises from implementing a method that returns a collection. When the representation already contains a collection object of the appropriate type, it is tempting to return it directly. For example, suppose that *List* has a method *toArray* that returns an array of elements

corresponding to the elements of the list. If we had implemented the list itself as an array, we might just return the array itself. If the *size* field was based on the index at which a null reference first appears) a modification to this array may break the computation of size.

```
a = l.toArray ();           // exposes the rep
a[i] = null;                // ouch!!

...
l.get (i);                  // now behaves unpredictably
```

Once *size* is computed wrongly, all hell breaks loose: subsequent operations may behave in arbitrary ways.

8.8. Language Mechanisms

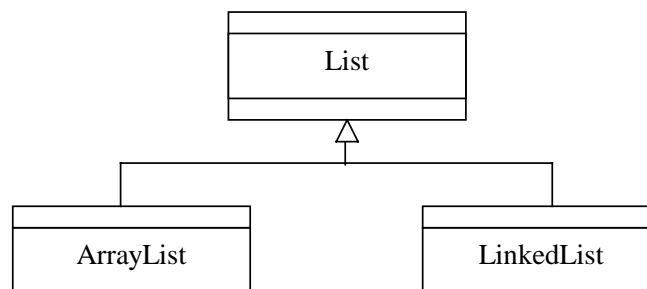
To prevent access to the representation, we can make the fields private. This eliminates the array hack; the statement

```
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. The interface breaks the dependence between the client and



another class, in this case the representation class:

This approach is nice because an interface cannot have (non-static) fields, so the issue of accessing the representation never arises. But because interfaces in Java cannot have constructors, it can be awkward to use in practice, since information about the signatures of the constructors that are shared across

implementation classes cannot be expressed in the interface. Moreover, since the client code must at some point construct objects, there will be dependencies on the concrete classes (which we will obviously try to localize). The *Factory* pattern, which we will discuss later in the course, addresses this particular problem.