L10: Abstract Data Types Part 2

Today

- Interfaces & multiple implementations of one ADT
- Abstraction function & representation invariant

Required Reading

- Interfaces
  http://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html
- Collection interfaces (focus on Set, List, and Map)
  http://docs.oracle.com/javase/tutorial/collections/interfaces/
- Collection implementations (again, Set, List, and Map, plus wrappers and convenience impl's)
  http://docs.oracle.com/javase/tutorial/collections/implementations/

Introduction

In this lecture, we push further in our study of abstract data types. Java’s collection classes provide a good example of the idea of separating interface and implementation, and we study a more formal idea of what it means for a class to implement an ADT, via the notions of abstraction functions and rep invariants.

Interfaces

One of the key principles in ADT design is representation independence: other code should be able to use an ADT implementation without knowledge of how it works internally. As a result, we can swap in different ADT implementations without disturbing a program. To allow that program to be written, it is important to have a very clear contract that any implementation of a particular ADT must follow. The Java concept of interfaces is a useful tool in formalizing such contracts.

To briefly review the required reading on interfaces given above: an interface is a static type that imposes requirements on objects that we might want to say belong to that type. The requirements come in the form of methods that must be present, where each method is given a signature providing its argument and return types. A class may make its objects compatible with the interface by declaring an explicit relationship via an implements clause.

Java’s static type checking allows the compiler to catch many mistakes that lead to incorrectly implementing an ADT’s contract. For instance, it is a compile-time error to omit one of the required methods, or to give it the wrong return type. Unfortunately, Java’s type checker is not strong enough to catch all serious errors in ADT implementation. The notion of specs that we have been applying all along provides the missing piece. The compiler doesn’t check for us that code adheres to specs, but, to use ADTs correctly, it is important to have specs in mind, and to reason about why any given ADT implementation follows the appropriate spec.

Let’s consider as an example one of the ADTs from the Java collections library, List, the ADT of mutable finite sequences. Here is a slightly tweaked copy of part of the List definition, with commentary:

These lecture notes have been collaboratively authored, with contributions from Saman Amarasinghe, Adam Chlipala, Srini Devadas, Michael Ernst, Max Goldman, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama. Copyright held by the authors, all rights reserved.
/** List represents a mutable list of elements. */
public interface List<E> {


Next we have two observer methods, which read a list’s private state without mutation. Notice how the specs are in terms of our abstract notion of a sequence; it would be malformed to mention the details of any particular implementation of lists with particular private fields. These specs should apply to any valid implementation of the list ADT. It is legal to specify one interface method in terms of another, as we do when we mention size in the spec for get.

/** Modifies this list by adding e to the end of the sequence. */
* @param e element to add; requires e != null. */
public void add(E e);

/** Modifies this list by removing the first occurrence of e, if found. */
* Does nothing if e is not found in the list.
* @param e element to remove; requires e != null.
* @return true if e was in the list; false if not */
public boolean remove(E e);

/** Replace an element at an index. */
* @param i index into the list, starting from 0
* @param v new value for that list position
* @throws IndexOutOfBoundsException if i not in [0,size) */
public void set(int i, E e) throws IndexOutOfBoundsException;

The story for these three mutator methods is basically the same as for the observers. We still write specs at the level of our abstract model of sequences.
/examples of producer methods

/**
 * Get the part of this list between from, inclusive, and to, exclusive.
 * The returned list is a "view" on this list, which means that
 * modifications to the sublist will change the larger list too.
 * Any structural change to either list invalidates the relationship,
 * and arbitrary behavior is then allowed. Structural changes are those
 * beside changing a sequence element with set().
 * @param from starting index
 * @param to ending index; requires 0 <= from <= to <= size()
 * @return sublist containing list positions in [from,to)
 *
 * @param from starting index
 * @param to ending index; requires 0 <= from <= to <= size()
 * @return sublist containing list positions in [from,to)
 */
public List<E> subList (int from, int to);

This last method has a complex specification, governing sharing of different views of the same mutable object. To be sure we implement such methods correctly, it will be helpful to develop a more formal notion of what it means to implement an ADT. But first, let’s review some of the reasons to use interfaces.

Why Interfaces?

Interfaces are used pervasively in real Java code. Not every class is associated with an interface, but there are a few good reasons to bring an interface into the picture.

- **Documentation for both the compiler and for humans.** Not only does an interface help the compiler catch ADT implementation bugs, but it is also much more useful for a human to read than the code for a concrete implementation. Such an implementation intersperses ADT-level types and specs with implementation details.

- **Allowing performance trade-offs.** Different implementations of the ADT can provide methods with very different performance characteristics. Different applications may work better with different choices, but we would like to code these applications in a way that is representation-independent. From a correctness standpoint, it should be possible to drop in any new implementation of a key ADT with simple, localized code changes.

- **Flexibility in providing invariants.** Different implementations of an ADT can provide different invariants.
  - **Optional methods.** List from the Java standard library marks all mutator methods as optional. By building an implementation that does not support these methods, we can provide immutable lists. Some operations are hard to implement with good enough performance on immutable lists, so we want mutable implementations, too. Code that doesn’t call mutators can be written to work automatically with either kind of list.
  - **Methods with intentionally loose specifications.** An ADT for finite sets could leave unspecified the element order one gets when converting to a list. Some implementations might use slower method implementations that manage to keep the set representation in some sorted order, allowing quick conversion to a sorted list; while other implementations might make many methods faster by not bothering to support conversion to sorted lists.

- **Multiple views of one class.** A Java class may implement multiple methods. For instance, a user interface widget displaying a drop-down list is natural to view as both a widget and a list. The class for this widget could implement both interfaces. In other words, we don’t implement an ADT multiple times just because we are choosing different data structures; we may make multiple implementations because many different sorts of objects may also be seen as special cases of the ADT, among other useful perspectives.
• **More and less trustworthy implementations.** Another reason to implement an interface multiple times might be that it is easy to build a simple implementation that you believe is correct, while you can work harder to build a fancier version that is more likely to contain bugs. You can choose implementations for applications based on how bad it would be to get bitten by a bug.

**Rep Invariant and Abstraction Function**

We now take a deeper look at the theory underlying abstract data types. This theory is not only elegant and interesting in its own right; it also has immediate practical application to the design and implementation of abstract types. If you understand the theory deeply, you’ll be able to build better abstract types, and will be less likely to fall into subtle traps.

In thinking about an abstract type, it helps to consider the relationship between two spaces of values. The space of *rep* or *representation* values consists of the values of the actual implementation entities. In simple cases, an abstract type will be implemented as a single object, but more commonly a small network of objects is needed, so this value is actually often something rather complicated. For now, though, it will suffice to view it simply as a mathematical value.

The space of *abstract* values consists of the values that the type is designed to support. These are a figment of our imaginations. They’re platonic entities that don’t exist as described, but they are the way we want to view the elements of the abstract type, as clients of the type. For example, an abstract type for unbounded integers might have the mathematical integers as its abstract value space; the fact that it might be implemented as an array of primitive (bounded) integers, say, is not relevant to the user of the type.

Now of course the implementor of the abstract type must be interested in the representation values, since it is the implementor’s job to achieve the illusion of the abstract value space using the rep value space.

Suppose, for example, that we choose to use a string to represent a set of characters. Then these form our two value spaces. We can show the two value spaces graphically, with an arc from a rep value to the abstract value it represents:

There are several things to note about this graph:

• Every abstract value is mapped to. The purpose of implementing the abstract type is to support operations on abstract values. Presumably, then, we will need to be able to create and manipulate all possible abstract values, and they must therefore be representable.

• Some abstract values are mapped to by more than one rep value. This happens because the representation isn’t a tight encoding. There’s more than one way to represent an unordered set of characters as a string.
Not all rep values are mapped. In this case, the string "abbc" is not mapped. If the type of the rep is nontrivial, it will not make sense to give an interpretation for all rep values. A doubly linked list representation, for example, can be twisted into all kinds of pretzel configurations that won’t correspond to simple sequences, and for which we won’t want to write special cases in the code. Or sometimes we will want to impose certain properties on the rep to make the code of the operations more efficient or easier to write. In this case, we have decided that the array should not contain duplicates. This will allow us to terminate the remove method when we hit the first instance of a particular character, since we know there can be at most one.

In practice, we can only illustrate a few elements of the two spaces and their relationships; the graph as a whole is infinite. So we describe it by giving two things:

1. An abstraction function that maps rep values to the abstract values they represent:
\[ AF : R \rightarrow A \]

The arcs in the diagram show the abstraction function. In the terminology of functions, the properties we discussed above can be expressed by saying that the function is onto, not necessarily one-to-one, and often partial.

2. A rep invariant that maps rep values to booleans:
\[ RI : R \rightarrow \text{boolean} \]

For a rep value \( r \), \( RI r \) is true if and only if \( r \) is mapped by \( AF \). In other words, \( RI \) tells us whether a given rep value is well-formed. Alternatively, you can think of \( RI \) as a set: it’s the subset of rep values on which \( AF \) is defined.

A common confusion students have about abstraction functions and rep invariants is that they imagine that they are determined by the choice of rep and abstract value spaces, or even by the abstract value space alone. If this were the case, they would be of little use, since they would be saying something redundant that’s already available elsewhere.

It’s easy to see why the abstract value space alone doesn’t determine \( AF \) or \( RI \): there can be several representations for the same abstract type. A set of characters could equally be represented as a string, as above, or as a bit vector, with one bit for each possible character. Clearly we need two separate functions to map these two different rep value spaces.

It’s less obvious why the choice of both spaces doesn’t determine \( AF \) and \( RI \). The key point is that defining a type for the rep, and thus choosing the values for the space of rep values, does not determine which of the rep values will be deemed to be legal, and of those that are legal, how they will be interpreted. Rather than deciding, as we did above, that the strings have no duplicates, we could instead allow duplicates, but at the same time require that the characters be sorted, appearing in nondecreasing order. This would allow us to perform a binary search on the string and thus check membership in logarithmic rather than linear time. Same rep value space — different rep invariant.

Even with the same type for the rep value space and the same rep invariant \( RI \), we might still have different interpretations \( AF \). Suppose \( RI \) admits any string of characters. Then we could define \( AF \), as above, to interpret the array’s elements as the elements of the set. But there’s no a priori reason to let the rep decide the interpretation. Perhaps we’ll interpret consecutive pairs of characters as subranges, so that the string "acgg" represents the set \{a,b,c,g\}.

The essential point is that designing an abstract type means not only choosing the two spaces — the abstract value space for the specification and the rep value space for the implementation — but also deciding what rep values to use and how to interpret them.
Example: Rational Numbers

Here's an example of an abstract data type for rational numbers. Look closely at its rep invariant and abstraction function.

```java
public class RatNum {
    private final int numer;
    private final int denom;

    // Rep invariant:
    // denom > 0
    // numer/denom is in reduced form

    // Abstraction Function:
    // represents the rational number numer / denom

    /** Make a new RatNum == n. */
    public RatNum(int n) {
        numer = n;
        denom = 1;
        checkRep();
    }

    /**
     * Make a new RatNum == (n / d).
     * @param n numerator
     * @param d denominator
     * @throws ArithmeticException if d == 0
     */
    public RatNum(int n, int d) throws ArithmeticException {
        // reduce ratio to lowest terms
        int g = gcd(n, d);
        n = n / g;
        d = d / g;

        // make denominator positive
        if (d < 0) {
            numer = -n;
            denom = -d;
        } else {
            numer = n;
            denom = d;
        }
        checkRep();
    }
}
```

Here is a picture of the abstraction function and rep invariant for this code. Or, rather, it’s almost a correct picture. Can you spot what is off, if we take the comments above to define the AF and RI?
The RI requires that numerator/denominator pairs be in reduced form (i.e., lowest terms), so pairs like (2,4) and (18,12) above should be drawn as outside the RI.

It would be completely reasonable to design another implementation of this same ADT with a more permissive RI. With such a change, some operations might become more expensive to perform, and others cheaper.

**Checking Rep Invariants & Implementing Abstraction Functions**

The rep invariant isn’t just a neat mathematical idea. If your implementation asserts the rep invariant at run time, then you can catch bugs early.

```java
// Check that the rep invariant is true
// *** Warning: this does nothing unless you turn on assertion checking
// by passing -enableassertions to Java
private void checkRep() {
    assert denom > 0;
    assert gcd(numer, denom) == 1;
}
```

toString() is a useful place to implement the abstraction function:

```java
/**
 * return a string representation of this rational number */
/**
 * This effectively implements the abstraction function
 */
public String toString() {
    return (denom > 1) ? (numer + "/" + denom) : (numer + "");
}
```

**Example: Lists**

Now we have the machinery to convince ourselves that an implementation of the List ADT is correct. The sublist method adds some complications, so let’s first consider how to implement ArrayList and LinkedList without supporting that sort of fancy snapshotting.
The `ArrayList` case is simple: we use an array in the natural way. With such a direct correspondence, we have a trivial, almost all-inclusive rep invariant.

```java
private E[] elts;

// Rep invariant:
// elts != null
// Abstraction function:
// represents the sequence with exactly the
// elements of elts
```

The `LinkedList` case is slightly more involved, but in the end we more or less wind up saying “represents the sequence read out of this linked list.”

```java
private Node first;
private class Node {
    E elt;
    Node next;
}

// Rep invariant:
// null is reachable from first (no cycles)
// Abstraction function:
// represents the sequence first.elt,
// first.next.elt, ..., first.next*.elt,
// up to but not including null
```

To support the shared-snapshot behavior of `subList`, we need to use more complex state and invariants for both cases. Here are the new skeletons of the two classes, with RIs and AFs:

```java
public class ArrayList<E> implements List<E> {
    private E[] elts;
    private int first, last;

    // Rep invariant:
    // elts != null
    // 0 <= first <= last <= elts.length
    // Abstraction function:
    // represents the length-n sequence
    // elts[first],...,elts[last-1]
}

public class LinkedList<E> implements List<E> {
    private Node first, last;
    private class Node {
        E elt;
        Node next;
    }

    // Rep invariant:
    // last is null or is reachable from first by
    // following next links
    // Abstraction function:
    // represents the sequence
    // first.elt, first.next.elt, ..., first.next*.elt,
    // up to but not including last
```
**Proving Correctness of ADT Implementations**

An AF/RI pair defines an invariant for a class. We can prove that our two examples from the last section are correct by using the *structural induction* technique from the last lecture. The invariant to be proved is exactly the rep invariant, which explains why we gave it that name!

Another correctness obligation comes from the representation-independent (interface-level) spec of each method. We must interpret each spec according to the chosen abstraction function and verify that the method code really follows that spec.

Here's a quick example for `LinkedList.set`. We start by reviewing the interface spec for this method.

```java
/** Replace an element at an index.  
 * @param i index into the list, starting from 0  
 * @param v new value for that list position  
 * @throws IndexOutOfBoundsException i  
 *         if i not in [0,size) */
public void set(int i, E e) throws IndexOutOfBoundsException;
```

And here is the implementation:

```java
public void set(int i, E e) {
    Node n;
    for (n = first; n != last && i > 0; n = n.next)  
        --i;
    if (i == 0 && n != last) {
        n.elt = e;
        return;
    }
    throw new IndexOutOfBoundsException();
}
```

First, we prove that the *rep invariant* is preserved. That is, we assume that the value `last` is eventually reached by repeatedly following `next` links starting from `first`. We need to prove that the invariant is still true after the method returns; this is trivial, because the only field update in the code is to `elt`, the field storing the data value of a node, so clearly the `next`-reachability of the list before and after are the same.

Now the harder part: prove that the abstract spec is implemented correctly, through the lens of our abstraction function. The abstract spec says our job is to transform the abstract sequence by replacing its \(i^{th}\) element with \(e\). As we go through the loop, we progress into simpler versions of the same problem: we consider shorter and shorter suffixes of the full sequence, decrementing \(i\) as necessary to ensure that it continues to point to the original target element within the current suffix. If the final if succeeds, then we know that we have reduced the problem to “replace the first element of this suffix sequence,” which is easy to do with a direct modification. The `n != last` test makes sure we have not already moved past the end of the list corresponding to the abstract sequence.

\[\square\]
Reasoning About ADTs from Client Code

The power of the AF/RI approach comes in the ways it allows us to reason about ADT implementations from code written independently of representation details. Here is a simple example, using comments to reason about an unknown List implementation in terms of the shared abstract model. The first comment gives a precondition, or assumption, for the span of code. The remaining comments give facts that we can deduce from the ADT spec. We write \(\sim\) to indicate that a concrete value represents an abstract value, according to whatever AF the concrete value uses internally.

```java
// l \sim\ empty
l.add(1);
// l \sim\ 1
l.add(2);
// l \sim\ 1, 2
l.remove(1);
// l \sim\ 2
int n = l.get(0);
// n = 2
```

Each of our deductions follows from the abstract specs only; we merely mentally execute standard mathematical operations on sequences. The Java code associated with these lecture notes runs this same code sequence on both ArrayList and LinkedList, yielding the same result. Though we don’t do so in this class, it is possible to develop rigorous mathematical machinery justifying why any correct implementation of the ADT will also validate the deductions we wrote in comments above.

Summary

Abstract data types are characterized by their operations, and Java interfaces help us formalize the idea of a set of operations that must be supported. We must go further in defining an effective ADT: methods must have representation-independent specs. The concepts of abstraction function and rep invariant can be used to explain why a class implements an ADT correctly. It is then possible to reason about an ADT abstractly, to show that client code behaves properly, without depending on details of a particular ADT implementation.