CMPSCI 187: Programming With Data Structures

Lecture 7: Array-Based StringLogs
David Mix Barrington
19 September 2012

An Array-Based StringLog Class

- The StringLog ADT and Interface
- Choosing Data Fields
- Coding the Constructors
- Coding the Transformers
- Coding the Observers
- Stepwise Refinement
- Testing: An Interactive Driver

The StringLog ADT and Interface

- Last time we defined an **abstract data type (ADT)** for the StringLog data structure. A StringLog is a collection of strings to which we may insert new strings and for which we can test membership of a given string and get a single string describing the whole collection.

- The following Java interface lists the methods that a StringLog class must implement. Today we'll see the first of two classes that do so.

```
public interface StringLogInterface {
   void insert(String element);
   boolean isFull( );
   int size( );
   boolean contains(String element);
   void clear( );
   String getName( );
   String toString( );}
```

Choosing Data Fields

- We know what data a StringLog object must keep -- its name and the collection of strings. The name is easy, as we declare "`protected String name`". The "`protected`" gives access to this class and any subclasses.

- Today we will look at keeping the collection in an **array** -- in Lecture #8 we will keep it in a linked list. But how big should the array be?

- The array will have a size, and the collection will be **full** if we cannot insert any more strings. We'll set the size when we create the object.

- That means that there are two "sizes" around, the size of the array and the number of locations used. We'll keep the used locations as a **consecutive prefix** of the array, the indices from 0 through one less than the size. We need an int variable to keep track of the last index used:

```
protected String[ ] log;
protected int lastIndex = -1;
```

Coding the Constructors

- Because an ArrayStringLog object will have a maximum capacity, we want to give the user a choice of setting this capacity or using a default one. This means two different constructors, **overloaded**, with different signatures.

- DJW give the argument to the constructor the same name as the data field it modifies, so they use the `this` reference to distinguish them.

- Unlike them, I've used the `this` constructor call below to avoid repeating the code of the first constructor when I do the same job with the second.

```
public ArrayStringLog(String name, int maxSize) {
   log = new String [maxSize];
   this.name = name;}

public ArrayStringLog(String name) {
   this (name, 100);}
```

Coding the Transformers

- To insert a new string, we make a new location active by changing `lastIndex`, then fill that location with the given string. If the array was already full we get an exception, but the *user* should have known better.

- To clear the StringLog (leaving the name and capacity the same), we can just move the `lastIndex` to -1 again and the array, though it might have strings still in it, would act just as if it didn't. But DJW are being thorough in wiping out those strings, and this could matter if someone later assumed that those unused locations were all `null`. They are also releasing that memory.

```
public void insert (String element){
// Precondition: This StringLog is not full
   lastIndex++;
   log[lastIndex] = element;}

public void clear ( ) {
   for (int i = 0; i <= lastIndex; i++)
      log[i] = null;
   lastIndex = -1;}
```

Coding the Observers

- The easy code first (three of the five observers):

```
public boolean isFull( ){
   return (lastIndex = log.length - 1));}

public int size( ) {
   return lastIndex + 1;}

public String getName( ){
   return name;}
```

- To test whether a given string is in the log, we really have to check each possibility as it might be anywhere in the active part of the array. And to assemble a single string with the contents of the whole collection, we have to work through each string of the array again. So while the above methods take O(1) time each, `contains` and `toString` are each O(n) time.

Code for `contains` and `toString`

- For toString we get a title line, two blank lines, then each string on its own line with a number. I used "+=" to help it fit on the page.

- We have a `while` where we could have used a `for` loop. Note that we take advantage of the canned Java method to ignore case in comparing strings.

```
public String toString( ){
   String logString = "Log: " + name + "\n\n";
   for (int i = 0; i <= lastIndex; i++)
      logString += ((i+1) + ". " + log[i] + "\n");
   return logString;}

public boolean contains (String element){
   int location = 0;
   while (location <= lastIndex) {
      if (element.equalsIgnoreCase(log[location]))
         return true;
      else location++;}
   return false;}
```

Stepwise Refinement

- These methods are simple enough that we can code them all at once, but DJW use `contains` as an example of an important general programming technique, called **stepwise refinement**, on pages 88-91.

- Using **pseudocode**, we can move from the original specification of the method, say "return true if `element` is there, false if it is not", to the actual Java code that does the job.

- We break up a pseudocode instruction into pieces, such as:

```
set variables
while (we still need to search)
   check the next value
return (whether we found the element)
```

- When the small pseudocode steps are simple enough, we replace them with real code. A block of code coming from a single pseudocode statement may be headed by a comment indicating what that statement was.

Testing in General

- Once we have written our class, is it correct? Does each method have the correct behavior in every legitimate situation? We can increase our confidence that this is true in two ways: **testing** and **validation**.

- Validation involves arguments (i.e., “proofs”) of assertions about the code, like “if the precondition is true and the code terminates, the postcondition is then true”. We use the rules of logic and the definition of the language to do this.

- Testing can *never* check all the possible legitimate situations, but thorough testing can give you some confidence. You need to generate a set of **test cases** that is representative, in some way, of the entire set of situations.

- Project 1 gives us an extra wrinkle -- how do we test the `remove( )` method when it has several valid behaviors and you are supposed to choose from them at random?

An Interactive Driver

- On pages 95-101 DJW build an **interactive test driver** for the ArrayStringLog class, which allows a user to run a variety of tests on the class, with a variety of parameters.

- The user picks a constructor to make an ArrayStringLog object, then can run any of the class' methods on that object and see the results.

- To be representative, you would want to test `isFull` on both full and non-full arrays, test `contains` on strings that are and aren't there, and when present are in a variety of positions in the array, test for partial matches and case-sensitive matches, and so on.

- DJW's driver doesn't test that you get an exception when you insert into a full collection -- to test this you would want a `try...catch` block so that the expected exception doesn't crash your driver.