6.170 Recitation 1 - Specifications and Testing

Declarative specs

Example: Consider a function that returns the index of the minimum value in an array:

```
static int findMinIndex(int[] a)
```

Write a declarative spec.

Possible solution:

Requires: a ≠ null. perhaps also that a.length > 0--it may be better to not require this and specify that an IllegalArgumentException will be thrown.

Modifies: nothing

Returns i s.t. 0 ≤ i < a.length and for all j | 0≤ j <a.length => a[i] ≤ a[j]

An operational spec would detail how this minimum is actually found (e.g. looping through the array and keeping track of the min seen so far, etc.).

(Note that the spec did not indicate the behavior of the method when the minimum value occurs in multiple locations in the array. This is an example of underdeterminedness. A stronger specification might indicate that the function will return the smallest possible i that satisfies the above condition.)

Strong Versus Weak Specifications

A **strong specification** is one that is tolerant on inputs and demanding on outputs whereas a **weak specification** is one that is demanding on inputs and weak on outputs. In 6.170, we generally want to produce strong specifications as they are more useful for the clients of our specifications. For example, consider the following three specifications of `double sqrt(double x)`:

A
```
@requires x >= 0
@return y such that |y^2 - x| <= 1
```

B
```
@return y such that |y^2 - x| <= 1
@throws IllegalArgumentException if x < 0
```

C
```
@requires x >= 0
@return y such that |y^2 - x| <= 0.1
```

We can make the following comparisons between specifications:

- **B** is a *stronger* specification than **A** because B requires less than **A**.
- **A** is a *weaker* specification than **C** because A promises less than **C**.
- **B** and **C** are *incomparable* because neither is strictly stronger nor weaker than the other.

These comparisons are important because a module with a stronger specification can always replace a module with a weaker specification. **However, one cannot generalize that strong or weak specifications are better**: strong specifications are bad where weak ones should be used; weak specifications are bad where strong ones should be used.

For example, consider `find(int[] a, int elt)` method that finds the index of an element in an array. This could be implemented efficiently with a binary search algorithm if we require that the array is sorted. So, even though its specification is weaker than that of an implementation that uses a general-purpose algorithm, the method may be more desirable in certain cases.

Fib example

Fibonacci.java

```
package rec1;

/**
 *
 */
public interface Fibonacci {
    public int fib(int n);
}
```

RecursiveFib.java

```
package rec1;

public class RecursiveFib implements Fibonacci {
    public int fib(int n) {
        if(n==1 || n ==2) return 1;
        return fib(n-2)+fib(n-1);
    }

}
```

LinearFib.java

```
package rec1;

public class LinearFib implements Fibonacci{
    public int fib(int n) {
        int current = 1, next = 1, nextNext;
        for (int i = 1; i < n; i++) {
            nextNext = next + current;
            current = next;
            next = nextNext;
        }
        return current;



    }
}
```

CachingFib.java

```
package rec1;

import java.util.HashMap;
import java.util.Map;

public class CachingFib extends LinearFib {
    private Map<Integer, Integer> cache = new HashMap<Integer,
Integer>();

    public int fib(int n) {
         if (cache.containsKey(n)) {
             return cache.get(n);
         } else {
             int v = super.fib(n);
             cache.put(n, n);  // This is an INTENTIONAL BUG!!!
             return v;
         }
    }
}
```

We have a Fibonacci interface, that is implemented by RecursiveFib, LinearFib, and CachingFib.

1. Given the Fibonacci interface, come up with a spec for the `fib` method. Points that we need to clarify in spec:
    - What does the input `n` mean? Is it 0-indexed or 1-indexed?
    - Should n>0 be a precondition? Our general philosophy is that you should have the least restrictive preconditions (unless it's overly costly to detect those abnormal cases), so here, we probably should not require n > 0 and instead specify that an IAE will be thrown if n <= 0.
2. Come up with black box tests:
    - e.g. fib(1), fib(2), fib(3), fib(4), fib (5), and maybe something larger like fib(30). and fib(0), fib(-1) if we don't have the precondition that n>0.
3. Simple implementations in RecursiveFib.java and LinearFib.java
4. Show CachingFib, which tries to cache values of previous calls in a HashMap, but does so incorrectly (put() call should have arguments n, v). The bug can be revealed by calling fib() with the same input twice. e.g. fib(30) should return 832040, but if you do consecutive calls, the second call returns 30.
5. CachingFib may be further optimized by using cached values for smaller n's when n cannot be found in the cache.
6. Caveat: if CachingFib extended RecursiveFib instead of LinearFib, the code would actually be doing something very tricky, because the recursive calls to fib(n-1) and fib(n-2) in RecursiveFib would dynamically dispatch to CachingFib.fib()!