Procedure Specifications
José M. Vidal
Tue Jan 18 10:44:04 EST 2005

These slides are based on
- MIT 6.170: Open Courseware, Procedure Specifications
- MIT 6.170: Open Courseware, Abstract Types

1 Introduction

- Specifications are required for team work.
- Many nasty bugs arise out of a misunderstanding about what something does.
- Specifications are easier to read than code, so the client is happier.
- Specifications allow the implementor to change her code without having to tell anyone.

2 Behavioral Equivalence

static int findA (int [] a, int val) {
  for (int i = 0; i < a.length; i++) {
    if (a[i] == val)
      return i;
  }
  return a.length;
}

static int findB (int [] a, int val) {
  for (int i = a.length -1 ; i >= 0; i--) {
    if (a[i] == val) return i;
  }
  return -1;
}

- Are these behaviorally equivalent?
- Not really They differ when val is either missing or appears more than once.
- Still, it depends on the specification. They would be equivalent if the specification was:

  requires: val occurs in a
  effects: returns result such that a[result] = val
3 Specification Structure

- A precondition, indicated by the keyword requires
- A postcondition, indicated by the keyword effects
- A frame condition, indicated by the keyword modifies
- The precondition is an obligation of the client.
- The postcondition is an obligation of the implementor of the method.
- The frame condition identifies which objects may be modified.

4 Declarative Specifications

- **Operational** specifications give a series of steps that the method performs.
- **Declarative** specifications give properties of final outcome.
- Prefer declarative.

```java
public StringBuffer reverse()
    // modifies: this
    // effects: Let n be the length of the old character sequence, the
    // one contained in the string buffer just prior to execution of the
    // reverse method. Then the character at index k in the new
    // character sequence is equal to the character at index n-k-1 in
    // the old character sequence.
    // Or, more formally
    // effects: length(this.seq) = length(this.seq )
    // for all k: 0..length(this.seq)-1 — this.seq [k] =
    // this.seq[length(this.seq)-k-1]

public boolean startsWith(String prefix)
    // Tests if this string starts with the specified prefix.
    // effects:
    // if (prefix = null) throws NullPointerException
    // else returns true iff exists a sequence s such that (prefix.seq ^
    // s = this.seq)

public String substring(int i)
    // effects:
    // if i < 0 or i > length (this.seq) throws IndexOutOfBoundsException
    // else returns r such that
    // some sequence s — length(s) = i & s ^ r.seq = this.seq
```
4.1 More Declarative Spec. Examples

/*This one does not determine the outcome but it is
still useful to clients. */
public boolean maybePrime ()
   // effects: if this BigInteger is composite, returns false

/*Same here, we don’t know in which order they will be
notified */

public void addObserver(Observer o)
   // modifies: this
   // effects: this.observers = this.observers + {o}

public void notifyObservers()
   // modifies the objects in this.observers
   // effects: calls o.notify on each observer o in this.observers

5 Exceptions and Preconditions

• The most common use of preconditions is to demand a property because it would be hard or
  expensive to check it.

• It is an engineering judgment. What is the cost of the check? what is the scope of the method?

• For example, binary search methods require that the array be sorted, otherwise they could not
deliver $O(\log n)$. If they had to sort, it would be $O(n \cdot \log n)$.

• Even when using a precondition, you might want to check it if it can be done speedily and
easily.

6 Specification Ordering

• Specification A is at least as strong as specification B if
  1. A’s precondition is no stronger than B’s
  2. A’s postcondition is no weaker than B’s, for the states that satisfy B’s precondition.

• That is, you can always weaken a precondition or strengthen a postcondition (make more
  promises).

7 Judging Specifications

• It should be coherent. It should not have lots of different cases.

• The results should be informative. The effects should tell the client something about what it’s
  doing.

• The specification should be strong enough. The modifies should be minimal: do as little as
  possible.

• The specification should be weak enough so that it can do what it promises to do most of the
time (instead of failing).
8 User-Defined Types

- All languages have built-in types. OO-languages allow one to define new types.
- The key idea of a data abstraction is that a type is characterized by the operations you can perform on it.
- User-defined types are useful for achieving de-coupling.

9 Types of Types

- A mutable type is one whose objects can be changed. An immutable type cannot be changed.
- In java Vector is mutable but String is immutable.
- Immutable types are generally easier to reason about.
- Mutable types might be faster to change, if large and change is only to a small part.

9.1 Types of Operations

- Constructors create new objects of the type.
- Producers create new objects from old objects.
- Mutators change objects.
- Observers take objects of the abstract type and return objects of a different type.

9.2 Example: List

```java
public class List {
    public List ();
    public void add (int i, Object e);
    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
    public void remove (int i);
    public int size ();
    public Object get (int i);
    // throws
    // IndexOutOfBoundsException if i < 0 or i > length (this.elems)
    // returns
    // this.elems [i] public void add (int i, Object e);
}
```

10 Designing and Abstract Type

- Its better to have a few simple operations that can be combined in powerful ways.
- Each operation should have a well-defined purpose.
• The set of operations should be adequate to satisfy all.
• The type may be either generic or domain-specific, but should not mix the two.

11 Representation Independence

• **Representation Independence** means ensuring that the use of an abstract type is independent of its representation.

• Changes in the representation should have no effect on the code outside the abstract type.

Vector v = new Vector();
List l = new List();
....
v.copyInto (l.elementData);

• This works as long as the List representation does not change.

class List {
    public Entry getEntry (int i);
}

//we can now write

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

• This is also bad because if the representation changes there might no longer be Entry objects.

• The List class should only use Objects.

12 Language Mechanisms

• Use **private** to prevent access to data members.

• Use **interfaces** to achieve representation independence.
  
  – Interfaces don’t have data members.
  
  – They also allow multiple implementations.

• Since interfaces don’t have constructors, it is wise to use the Factory pattern.

---

**Notes**


This talk is available at [http://jmvidal.cse.sc.edu/talks/specifications](http://jmvidal.cse.sc.edu/talks/specifications)

Copyright © 2004 Jose M Vidal. All rights reserved.