Procedural Abstraction

Procedural Abstraction

Procedures combine the methods of:

- **Abstraction by Parameterization**
- **Abstraction by Specification**

in a way that allows us to abstract a single action or task, such as computing the gcd of two integers or sorting an array.

Abstraction by Parameterization

- **abstracts from the identity of the data being used.**

The abstraction is defined in terms of formal parameters, the actual data are bound to these formals when the abstraction is used.

e.g.

```
int squares(int x) {
   return x * x ;
}
```

Use of abstraction: y = squares(2);

Abstraction by Parameterization

- **identity of the actual data : *irrelevant***
- **presence, number and type of parameters : *relevant***

e.g.

```
int squares(int x) {
   return x * x ;
}
```

Use of abstraction: y = squares(2);

Irrelevant: **the value of x**

Relevant: **one parameter of type int**

Benefits of Abstraction by Parameterization

- **Allows to describe a large number of computations relatively simply**

- **Generalizes modules so that they can be used in more situations.**

- **A virtue of such generalization is that they decrease the amount of code that needs to be written, modified and maintained.**

Abstraction by Specification

- **focuses on the behavior that the user can depend on and abstract from the details of implementing that behavior.**

- **behavior – "what is done" –** ***relevant***
- **method realizing the behavior – "how it is done" –** ***irrelevant***

Abstraction by Specification

e.g.

```
float sqrt ( float coeff ) {
  //REQUIRES: coeff > 0
  //EFFECTS: Returns an approximation to the square
  //                root of coeff
  ……
}
```

Use of abstraction: y = sqrt(2);

- y is an approximation to the square root of 2: *relevant*

- details of how the square root is determined: *irrelevant*

Benefits of Abstraction by Specification

Provides a method for achieving a program structure with *two* advantageous properties:

- ***Locality*: the implementation of an abstraction can be read or written without needing to examine the implementation of any other abstractions**
  - **to write a program that uses an abstraction, a programmer need to understand only its behavior, not the details of its implementation**
  - **different abstractions that make up a program can be implemented by people working independently.**

Benefits of Abstraction by Specification (contd.)

- ***Modifiability*: an abstraction can be reimplemented without requiring changes to any abstraction that use it.**
  - **if the implementation of an abstraction changes, but its specification does not, the rest of the program will not be affected by the change.**
  - **helps to bound the effects of program modification and maintenance**
  - **e.g. we could change the algorithm used to implement the `sqrt` procedure, and the programs using `sqrt` would continue to run correctly with this new implementation.**

Specifications (used in procedural abstraction)

- **Specifications are written in a *specification language* that can be either formal or informal.**

- **Formal specifications have a precise meaning and can be verified automatically**

- **Informal specifications (written in informal English) are easier to read and write than formal ones, but giving them a precise meaning is difficult.**

Specifications (contd.)

- **Specification language is not a programming language.**

- **We will use informal specifications. We will associate with a procedure a *comment* that is sufficiently informative to allow others to use that procedure without looking at its body.**
  - **A good way to write such comments is to use assertions**

Specification template for procedural abstractions

return_type procedureName(…..)

//REQUIRES: this clause states any constraints on use

//MODIFIES: this clause identifies all modified inputs

//EFFECTS: this clause defines the behavior

Example

```
float sqrt ( float coeff ) {
    //REQUIRES: coeff > 0
    //EFFECTS: Returns an approximation to the square
   //                root of coeff
    ……
}
```

REQUIRES Clause

- **states the constraints under which the abstraction is defined.**

- **It is needed if the procedure is *partial* (that is, if its behavior is not defined for some inputs).**

- **It can be omitted if the procedure is *total* (that is, if its behavior is defined for all type-correct inputs).**

MODIFIES Clause

- **lists the names of any inputs (including implicit inputs) that are modified by the procedure.**

- **If some inputs are modified, we say that the procedure has a side effect.**

- **It can be omitted if no inputs are modified.**

EFFECTS Clause

- describes the behavior of the procedure for all inputs not ruled out by the REQUIRES clause.

- It must define what outputs are produced and also what modifications are made to the inputs listed in the MODIFIES clause.

- It is written under the assumption that the REQUIRES clause is satisfied.

Specification template for class providing standalone procedures (i.e. a class containing only static methods)

```
visibility className {
   //OVERVIEW: this clause defines the purpose of the class as a whole
   visibility static procedure1 …
   visibility static procedure1 …
}
```

Example

```
public class Arrays {
//OVERVIEW: stand-alone procedures for manipulating
//arrays of ints

public static int search (int [ ] a, int x)
//EFFECTS: if x is in a, returns the index where x is
//         stored, otherwise returns -1
```

search and searchSorted do not modify their inputs – so no modifies clause

Example

```
public static int searchSorted (int [ ] a, int x)
//REQUIRES: a is sorted in ascending order
//EFFECTS: if x is in a, returns the index where x is
//         stored, otherwise returns –1


public static void sort (int [ ] a)
//MODIFIES: a
//EFFECTS: Rearranges the elements of a in ascending
//         order
}
```

sort and search are *total* since their specifications do not contain a REQUIRES clause.

searchSorted is *partial*, it only does its job if the argument is sorted

Implicit inputs

- **In addition to the formal parameters as inputs, a procedure can have implicit inputs. Implicit input modifications should be described in specifications**

- **e.g.**

```
public static void copyLine()
// REQUIRES: System.in contains a line of text
// MODIFIES: System.in and System.out
// EFFECTS: Reads a line of text from System.in, advances
//                  cursor in System.in to end of line, writes the line
//                  on System.out
```

Implementing procedures

The implementation of a procedure should produce the behavior defined by its specifications

- **It should modify only those inputs that appear in the MODIFIES clause**
- **If all inputs satisfies the REQUIRES clause, then it should produce the result in accordance with the EFFECTS clause.**

- **Specifications are written first, implementations are done later**

Implementing procedures – Example 1

```
public class Arrays {
   //OVERVIEW: …
   public static int searchSorted (int [ ] a, int x) {
     // REQUIRES: a is sorted in ascending order
     // EFFECTS: if x is in a, returns the index where x
     //          is stored, otherwise returns -1
     // uses linear search
     if (a == null) return -1;
     for (int i = 0; i < a.length; i++)
        if (a[i] == x) return i;
        else if (a[i] > x) return -1;
     return -1;
   }
}
```

If a is unsorted => returns wrong result

If a is null => returns –1

Behavior is consistent with what is described in specifications

Implementing procedures – Example 2

```
public class Arrays {
    //OVERVIEW: …

    public static int sort (int [ ] a) {
      // MODIFIES: a
      // EFFECTS: Rearranges the elements of a in ascending order
      // e.g. if a = {3,1,6,1} , a_post={1,1,3,6}
      if (a == null) return;
      quicksort (a, 0, a.length-1);
   }
```

Implementing procedures – Example 2 (contd.)

```
  private static void quickSort(int [ ] a, int low, int
high) {
    // REQUIRES: a is not null and 0 <= low and high < a.length
    // MODIFIES: a
    // EFFECTS: Sorts a[low],a[low+1],...,a[high] in ascending order
    if (low >= high) return;
        int mid = partition(a, low, high);
        quickSort(a,low,mid);
        quickSort(a,mid,high);
   }
```

Implementing procedures – Example 2 (contd.)

```
private static int partition(int [] a, int i, int j) {
   // REQUIRES: a is not null and 0 <= i < j < a.length
   // MODIFIES: a
   // EFFECTS: Reorders the elements in two contiguous
   // groups, a[i],...,a[res] and a[res+1],...,a[j]
   // such that each element in
  // the second group is at least as large as each element in
   // the first group
  // returns res
   ........
  }
}
```

Designing Procedural Abstractions

Procedures are introduced during program design to shorten the calling code and clarify its structure.

- **it is possible to introduce too many procedures. e.g. introducing `partition` procedure is worth because it has a well-defined purpose. Further decomposition is counter-productive.**
- **if the purpose of a procedure is difficult to state, then probably it should not be introduced.**
- **if identical code is repeated multiple times, then probably it will be useful to define a procedure.**

Properties of Procedures and their implementations

- **minimality**
- **underdetermined behavior**
- **deterministic implementation**
- **generality**
- **simplicity**

Designing Procedural Abstractions (contd.)

• **Procedures should be designed to be *minimally constraining*: a specification should constrain the details of the procedure only to the extent necessary. In this way, more freedom is given to the implementer who may be able to provide an efficient implementation. e.g. the algorithm is undefined.**

Designing Procedural Abstractions (contd.)

• sometimes, this leads to a procedure being *underdetermined*. This means, for certain inputs, instead of a single correct output, there is a set of acceptable outputs. e.g. `search` and `searchSorted` procedures are underdetermined because we did not state exactly what index should be returned if x occurs in the array more than once.

Designing Procedural Abstractions (contd.)

- **Example**

```
public class Arrays {
    // OVERVIEW: ...
    public static int searchSorted (int[ ] a, int x) {
        // REQUIRES: a is sorted in ascending order.
        // EFFECTS: If x is in a, returns an index where x is stored,
        //                  otherwise, returns -1.
        ……  // linear search
    }
}
```

Designing Procedural Abstractions (contd.)

Uses binary search. Two procedures may return different indices if x appears more than once.

- **Example (contd.)**

```
public class Arrays {
    // OVERVIEW: ...
     public static int searchSorted (int[ ] a, int x) {
          // REQUIRES: a is sorted in ascending order.
          // EFFECTS: If x is in a, returns an index where x is stored,
          //                otherwise, returns -1.
          ……  // binary search
     }
}
```

Designing Procedural Abstractions (contd.)

- **an underdetermined abstraction usually has a *deterministic implementation*, I.e. if called twice with identical inputs, behaves identically on the two calls. e.g. both implementations of searchSorted are *deterministic*.**

Designing Procedural Abstractions (contd.)

• ***Generality*: a specification is more general if it can handle a larger class of inputs, e.g if a procedure works on any size array is more general than one that works only on arrays of some fixed size. It is worth generalizing a procedure if doing so increases its usefulness.**

• ***Simplicity*: procedures should have a well-defined and easily explained purpose that is independent of its context of use. Good check: give the procedure a name that describes its purpose. If difficult to think a name, there may be a problem.**

Designing Procedural Abstractions (contd.)

• *Partial procedures* (are always specified with a REQUIRES clause) are less safe than *total ones.* They should only be used when:

- **the context of use is limited (e.g. private helper methods like quickSort)**
- **they enable a substantial benefit such as better performance, e.g. searchSorted do not need to check that the array is sorted or not**

Procedural Abstraction - Example

```
private static int partition(int [] a, int i, int j) {
    // REQUIRES: a is not null and 0 <= i < j < a.length
    // MODIFIES: a
    // EFFECTS: ……..
    int x = a[i];
    while (true) {
    while (a[j] > x) j--;
    while (a[i] < x) i++;
        if (i < j) { // need to swap
            int temp = a[i]; a[i]=a[j]; a[j] = temp;
            j--; i++;
        } else return j;
    }
}
```