Lecture 6: Procedural Abstractions

- **Defining procedural abstractions**
  - the parts of a procedural abstraction
  - total vs. partial procedures
  - side effects
- **Implementing procedural abstractions**
  - defensive programming
  - optimization
  - some comments on program style

Note: procedural abstraction applies to any language, no matter what the units are called:

procedures (e.g. Ada, Modula,…)
functions (e.g. C, ML,…)
methods (e.g. java,…)

Procedural Abstractions

- **A procedure maps from input to output parameters**
  - it may modify its parameters
  - it may have side effects
  - it may return a result
- **aim for "*Referential Transparency*"**
  - procedure does the same thing, no matter where it is used
  - basis of the Cleanroom approach
- **A procedural abstraction ("specification"):**
  - describes what a procedure does, ignores how it does it
  - different implementations of the abstraction can differ over details
  - one implementation can be substituted for another
- **Advantages**
  - Locality: programmers don't need to know implementation details
  - Modifiability: replacing an implementation does not affect the rest of the system
  - Language Independence: implementation could be any programming language

Defining abstractions

- **Abstractions need to be precisely defined**
  - formally (mathematically): very precise; can be automatically checked
  - informally (e.g. natural language description): less precise, easier to read and write
- **Need to define five things:**
  - The way in which the procedure communicates (input/output parameters)
  - The conditions under which the procedure will work
  - What the procedure achieves
  - Any side effects (changes to global variables or system state)
  - Any exceptions raised

```
procedure sort(a:array of int, len:int) returns array of int
requires: a is an array that is at least len integers long
effects: returns a copy of the array a with its elements sorted
  into ascending order
modifies: reduces available heap space by n * sizeof(int)
raises: arraybounderror if a is not a valid pointer to an array of
  length len; memerror if there is insufficent heap space for a new
  array of length len
```


Total vs. Partial Procedures

- **A total procedure**
  - works for any input
    - (within the type checking restrictions of the language)
  - hence has no requires clause
  - e.g.

```
procedure length(a: stack of int) returns b:int
effects: b is the number of elements in a
```

- **A partial procedure**
  - works for some of the possible inputs
  - e.g.

```
procedure sqrt (a:int) returns b: real
requires: a ≥ 0
effects: b is an approximation of the square root of
  a to within ±10^-4
```

  - (square root only works for non-negative integers)
  - The requires clause places restrictions on the operation of the procedure
  - The procedure is only guaranteed to work if the requires clause is met

Specifying Side Effects

→ Side effects

- If a procedure modifies its environment in any way, this is a side effect
  - e.g. modifying global variables
  - e.g. allocating or de-allocating memory
  - e.g. printing text on the screen (actually: writing to the output stream)
  - e.g. reading characters from the keyboard (actually: consuming the input stream)
- A pure function has no side effects
  - all communication is through its parameters and return result
- All programming languages allow procedures/functions to have side effects
  - input/output is impossible otherwise(!)
  - but side effects make a program harder to understand and more prone to error

→ Use of 'modifies'

```
procedure initialize_counter()
  returns old:int
modifies: the global variable
  count is set to zero
effects: old is set to the value
  of count before initialization
```

```
procedure readlines (n:int)
  returns s:list of strings
requires n>=0
modifies: advances the input stream by
  up to n lines
effects: s is a list of up to n strings,
  containing characters on the next n
  lines of input. Newline characters are
  not included in the strings
```


Different Implementations

```
procedure search (a: list of int, x:int) returns i:int
requires: a is sorted in ascending order
effects: If x is in a, i is the index of an occurrence
  of x in a, so that a[i]=x otherwise i is -1
```

→ Many possible implementations:

- linear search - slow but easy to implement
- binary search - fast for large lists
- …

→ These satisfy the abstraction, but:

- what if x occurs more than once?
- what if a is not sorted?
- If we care about any of these details, they should be described in the abstraction.

Procedure Design

→ Procedural abstractions:

- …have users and an implementor
- the abstraction defines the service offered to the users
- the implementor is free to provide the service in whatever way seems best (As long as it meets the specification)

→ The abstraction should:

- constrain things that matter to the user
  - e.g. whether sort creates a new list or modifies the old one…
- not constrain things that don't matter to the user
  - e.g. speed, efficiency, algorithm used…

→ Under-determination

- "some aspects of behavior are not defined"
  - e.g. search was underdetermined as we didn't say what to do if the element occurs more than once in the list.
- an under-determined specification may have implementations that behave differently

Desirable properties of procedures

→ Minimally specified

- only constrained to the extent required by the users

→ General

- able to work on a range of inputs (or input types)
  - e.g. search could be generalized to work on any array types
  - ...we might need to pass it a comparison function
- BUT: generalizing a procedure is only worthwhile if it becomes more useful
  - c.f. moving a method up the class hierarchy

→ Simple

- a well-defined and easily explained purpose
  - tip: if you can't think of a simple name for your procedure, it's probably overly complex (= not cohesive)

→ Non-trivial

- should achieve something significant
- don't decompose a program down into too many tiny pieces

Defensive Programming

➜ Murphy's law:

- anything that can go wrong will go wrong
- e.g. if you rely on precedence order for expressions, you'll make a mistake, so put brackets everywhere
  - x * y + a * b
  - (x * y) + (a * b)
- e.g. people will call your procedure with the wrong inputs, will forget to initialise data, etc, so always check!

➜ Partial Procedures are Problematic

- sooner or later someone will violate the 'requires' clause
- either: try to make them total
- or: add code at the beginning that checks the requires clause is met

Further advantages of abstraction

➜ Encapsulation

- all the important information about the procedure is stated explicitly in one place
- the detail is hidden

➜ Testing

- without an abstraction defined, how will you know if your procedure is correct?
- the abstraction will suggest unusual ("off nominal") test cases

➜ Optimization

- It is often hard to predict where bottlenecks will occur
- use abstractions to implement the whole program, then just optimize those procedures that need optimizing

➜ Error tracing

- abstractions help you build firewalls that stop errors propagating

Elements of Program Style

Source: *Adapted from Blum, 1992, p278-9*

Program code is an expression of a design that *will* change:

- write clearly, avoid cleverness
- use library functions
- avoid temporary variables
- clarity is more important than efficiency
- parenthesize to avoid ambiguity
- avoid confusing variable names
- don't patch bad code - rewrite it
- don't over-comment
- don't comment bad code - rewrite it
- format the code for readability

As a design, program code should convey intellectual clarity

- clarity is better than small gains in efficiency
- make it right before you make it faster
- make it robust before you make it faster
- make it clear before you make it faster
- choose a data representation that makes the program simple

Program code represents the result of problem solving

- write first in a simple pseudo-code then refine
- modularize
- write and test a big program in small pieces
- instrument your programs
- measure for bottlenecks before you optimize
- watch for "off-by-one" errors
- test the "boundary conditions"
- checks some answers by hand

Assumptions are dangerous

- test inputs for validity and plausibility
- identify bad input, recover if possible
- use self-identifying input
- make input easy to prepare
- make output self-explanatory
- make sure the code "does nothing" gracefully

Summary

➜ Procedural abstractions are useful

- they express the contract between user and implementor
- they are helpful for testing
- they facilitate modification

➜ Procedural abstractions must be defined precisely

- "abstract" does not mean the same as "vague"!
- strive for referential transparency

➜ This process works at all levels

- The principles shown here for procedures apply to all design levels:
  - specify the abstraction precisely
  - the specification should tell you everything you need to know to use the component
  - the specification should not include unnecessary design information
- Try it for:
  - systems, CSCIs, modules, packages, procedures, loops, ...

References

van Vliet, H. "Software Engineering: Principles and Practice (2nd Edition)" Wiley, 1999.

- deals with procedural abstraction briefly in section 11.1. But you'll also need to refer to:

Liskov, B. and Guttag, J., "Program Development in Java: Abstraction, Specification and Object-Oriented Design", 2000, Addison-Wesley.

- Chapter 3. I draw on Liskov's ideas extensively for advice on program design in this course. The commenting style I use ("requires", "effects", etc) is Liskov's. If you plan to do any extensive programming in Java, you should buy this book. If you don't buy it, borrow it and read the first few chapters.

Blum, B. "Software Engineering: A Holistic View". Oxford University Press, 1992

- Blum does an nice treatment on program design and abstractions (see especially section 4.2)

Prowell, S. J, Trammell, C. J, Linger, R., and Poore, J. H. "Cleanroom Software Engineering", 1999, Addison-Wesley

- The cleanroom approach relies heavily on encapsulation and referential transparency. It demonstrates how abstraction and specification can be used in the same way at each level of design.