Programming Languages & Paradigms
PROP HT 2011

Lecture 4

Subprograms, abstractions, encapsulation, ADT

Beatrice Åkerblom
beatrice@dsv.su.se

Stockholm University

Course Council

Stockholm University

Meeting on friday!

Talk to them and tell them what you think is good and bad

Simon Ottervald
Daniel Malmqvist
????

Subprograms

Stockholm University

Fundamentals of Subprograms

Stockholm University

- General characteristics of subprograms:
  - A subprogram has a **single entry** point
  - The **caller is suspended** during execution of the called subprogram
  - **Control always returns** to the caller when the called subprogram's execution terminates

Subprograms - Definitions

Stockholm University

- A **subprogram definition** describes both the interface to the subprogram abstraction and its actions
- A **subprogram call** is an explicit request that the subprogram be executed
- A **subprogram header** is the first part (line) of the definition, including the name, the kind of subprogram, and the formal parameters
- The **parameter profile** of a subprogram is the number, order, (and types) of its parameters
- The **protocol** of a subprogram is its parameter profile plus, if it is a function, its return type

Subprograms - Definitions (cont'd)

Stockholm University

- A **subprogram declaration** provides the protocol, but not the body, of the subprogram (e.g. in C and C++ header files)
- A **formal parameter** is a variable listed in the subprogram header and bound to storage only during execution of the subprogram
- An **actual parameter** represents a value or address used in the subprogram call statement

```
>> def add(first, second)
>>    puts("Result is #{first + second}")
>> end

>> add(2, 3)
Result is 5

>> f, s = 2, 3
>> add(f,s)
Result is 5
```


Actual/Formal Parameter Correspondence

Stockholm University

- Positional

```
>> def order(first, second)
>>    puts("First is #{first} and second is #{second}")
>> end
>> add(3, 4)
First is 3 and second is 4
```

- Keyword

```
>>> def order(first, second):
...    print("1st: " + str(first) + " 2nd: " + str(second))
...
>>> order(second = 3, first = 7)
1st: 7 2nd: 3
```


Actual/Formal Parameter Correspondence (cont'd)

Stockholm University

- Default Values:

```
>>> def default(first, second = 2):
...    print("1st: " + str(first) + " 2nd: " + str(second))
...
>>> default(3)
1st: 3 2nd: 2
>>> default(second=3, first=7)
1st: 7 2nd: 3
```

- Number of actual vs. formal parameters usually required to be the same (unless default values are provided), exceptions are e.g. C, Perl, Javascript

```
js> function only_care_about_first(first) {
  > print(first);
  > }
js> only_care_about_first("one", "two", "three")
one
```

Stockholm University

Actual/Formal Parameter Correspondence (cont'd)

- Variable length parameter list

```
js> function care_about_all() {
  >   for (var i = 0; i < arguments.length; i++)
  >      print(arguments[i]);
  > }
js> care_about_all("one", "two", "three")
one
two
three
```


Stockholm University

Blocks of Code as Parameters

- Not only a construct for iterators in Ruby...

```
>> def map (list)
>>   for elem in list
>>      yield elem
>>   end
>> end
=> nil
>> sum = 0
=> 0
>> map([1,3,5]) {|number| sum += number}
=> [1, 3, 5]
>> puts sum
9
=> nil
```


Stockholm University

Blocks of Code as Subprograms

```
c := [:arg | Transcript show: arg; cr.].

c value: 'The argument'.

In Transcript:
The argument
```


Stockholm University

Local Variables - Stack-dynamic

- Bound to storage when subprograms starts executing
- Advantages:
  - Support for recursion
  - Storage for locals is shared among some subprograms
- Disadvantages:
  - Allocation/deallocation time
  - Indirect addressing
  - Subprograms cannot be history sensitive

Local Variables - Static

- Static locals are bound to storage throughout the program execution (preserved between subprogram calls)

```
#include<stdio.h>

int adder(int list[], int listlen){
  static int sum = 0; int count;
  for (count = 0; count < listlen; count++)
    sum += list[count];
  return sum;
}
int main () {
  int result; int list[] = {1,2,3,4,5};

  result = adder(list, 5);
  printf("%d\n", result);

  result = adder(list, 5);
  printf("%d\n", result);
}
```

```
beatrice@triton:~$ gcc testc.c -o test
beatrice@triton:~$ ./test
15
30
```


Nested Subprograms

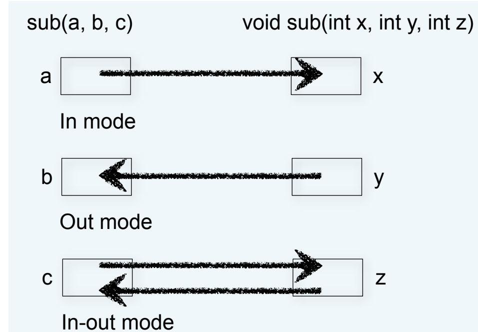
- Possibility to create hierarchy of both logic and scopes

```
>> def foo
>>   def bar
>>     puts "bar"
>>   end
>>   bar
>>   puts "foo"
>> end
=> nil
>> foo
bar
foo
```


Python

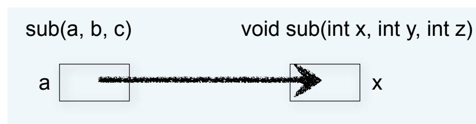
Parameter Passing Methods

- Semantic Models
  - in mode
  - out mode
  - in-out mode
- Conceptual Models of Transfer:
  - Physically move a value
  - Move an access path



Pass-by-value (in mode)

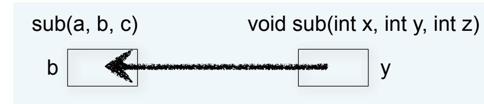
- Either by physical move or access path
- Disadvantages of access path method:
  - Must write-protect in the called subprogram
  - Accesses cost more (indirect addressing)
- Disadvantages of physical move:
  - Requires more storage (duplicated space)
  - Cost of the moves (if the parameter is large)



Pass-by-result (out mode)

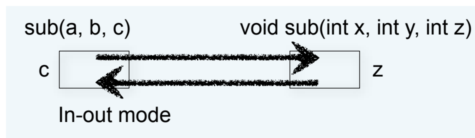
- Local's value is passed back to the caller
- Physical move is usually used
- Disadvantages:
  - If value is passed, time and space
- In both cases, order dependence may be a problem e.g. procedure sub1(y: int, z: int); ... sub1(x, x);
- Value of x in the caller depends on order of assignments at the return

Out mode



Pass-by-value-result (in-out mode)

- Physical move, both ways
- Also called pass-by-copy
- Disadvantages:
  - Those of pass-by-result
  - Those of pass-by-value



Pass-by-reference (in-out mode)

- Pass an access path
- Advantage:
  - Passing process is efficient (no copying and no duplicated storage)
- Disadvantages:
  - Slower accesses
  - Allows aliasing:
    - Actual parameter collisions, e.g. procedure

```
sub1(a: int, b: int); ... sub1(x, x);
```

    - Array element collisions:

```
sub1(a[i], a[j]); /* if i = j */
sub2(a, a[i]);
```

    - Collision between formals and globals
- Root cause of all of these is: The called subprogram is provided wider access to nonlocals than is necessary

Pass-by-name (multiple mode)

Stockholm University

- By textual substitution
- Formals are bound to an access method at the time of the call
- Actual binding to a value or address takes place at the time of a reference or assignment (evaluated when and only when the parameter is actually used)
- Advantage: flexibility of late binding
- Disadvantage: hard to read and understand

- scalar variable, pass-by-reference
- constant expression, pass-by-value
- an array element (or expression referencing a variable), like nothing else

```
procedure sub1(x: int; y: int);
begin
   x := 1; y := 2; x := 2; y := 3;
end;
sub1(i, a[i]);
```


Stockholm University

Implementing Parameter Passing

- ALGOL 60 and most of its descendants use the run-time stack

- **Pass-by-value** - copy value to the stack locations used as storage for formal parameters
- **Pass-by-result** - stack locations used as storage for formal parameters and on return copied into actual parameter storage
- **Pass-by-reference** - regardless of form, put the address in the stack

Stockholm University

Design Considerations for Parameter Passing

- Efficiency
- One-way or two-way

← These two are in conflict with one another!

- Good programming = limited access to variables, which means one-way whenever possible
- Efficiency = pass by reference is fastest way to pass structures of significant size

- Also, functions should not allow reference parameters -> side- effect

Stockholm University

Parameters that are Subprogram Names

- Are parameter types checked?
  - Early Pascal and FORTRAN 77 do not
  - Later versions of Pascal and FORTRAN 90 do
  - Ada does not allow subprogram parameters
  - Java does not allow method names to be passed as parameters
  - C and C++ - pass pointers to functions; parameters can be type checked

Stockholm University

Parameters that are Subprogram Names (contd)

- What is the correct referencing environment for a subprogram that was sent as a parameter?

- **Shallow binding:** the environment where it is called
- **Deep binding:** the environment where it was declared
- **Ad hoc binding:** the environment where it was passed as an actual parameter

- For static-scoped languages, deep binding is most natural
- For dynamic-scoped languages, shallow binding is most natural

Stockholm University

```
js> function sub1(){
  >   var x;
  >   function sub2(){
  >      print(x);
  >   };

  >   function sub3(){
  >      var x;
  >      x = 3;
  >      sub4(sub2);
  >   };

  >   function sub4(subx){
  >      var x;
  >      x = 4;
  >      subx();
  >   };

  >   x = 1;
  >   sub3();
  > };
```

- What is the referencing environment of `sub2` when it is called in `sub4`?
  - Deep binding => sub2, sub1
  - Shallow binding => sub2, sub4, sub3, sub1

Stockholm University

Generic Subprograms

- A **polymorphic** subprogram is one that takes parameters of different types on different activations
- Overloaded subprograms provide ad hoc polymorphism
- **Parametric polymorphism** is provided by subprograms that
  - takes a generic parameter
  - use that generic parameter in type expressions that describes the type of the parameters of the subprogram
  - This means that different instantiations of the subprogram can take (and check) parameters of different types

Stockholm University

Coroutines

- A coroutine is a subprogram that has multiple entries and controls them itself
- Also called symmetric control
- A coroutine call is named a resume
- The first resume of a coroutine is to its beginning, but subsequent calls enter at the point just after the last executed statement in the coroutine

Stockholm University

Abstraction

Stockholm University

Abstraction – fundamental in Computer Science

- Abstraction in Computer Science often implies simplification:
  - replacing a complex and detailed real-world situation by an understandable model
  - we only want to consider parts of reality important for the things we want our system to handle
  - we want to avoid everything else since it would only make the more important things harder to see
- Abstract doesn't mean imprecise

Roy Lichtenstein (American, 1923-1997), the six prints in the "Bull Profile Series,"

Stockholm University

Bull I

Bull II

Bull III

Bull IV

Bull V

Bull VI

"The essence of abstraction is to extract essential properties while omitting inessential details."
[Ross et al, 1975]

Stockholm University

Encapsulation - Original Meaning & Motivation

- Large programs have two special needs:
  - Some means of **organization**, other than simply division into subprograms
  - Some means of **partial compilation** (compilation units that are smaller than the whole program)
- Obvious solution: a grouping of subprograms that are logically related into a unit that can be separately compiled

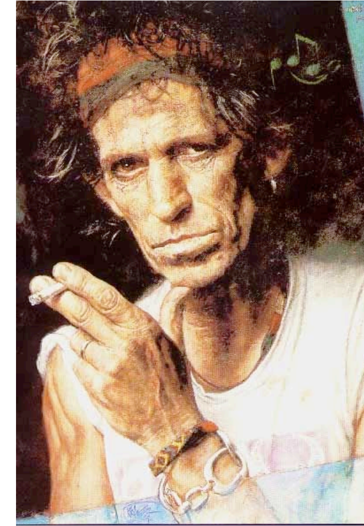
- Examples of encapsulation mechanisms:
  - Nested subprograms in some (e.g., Pascal, JavaScript, etc.)
  - Separate compilation of files containind one or more subprograms (e.g. FORTRAN 77 and C)

Stockholm University

Encapsulation vs. Information Hiding

- As so many other things the meaning of encapsulation is a subject of discussion.
- In OO often considered to be interchangeable with information hiding. Authors seldom distinguish between the two and often directly claim they are the same.

Fortran



Stockholm University

Stockholm University

Encapsulation

- The idea of encapsulation comes from the need to cleanly distinguish between the specification and the implementation of an operation and the need for modularity

- Encapsulation means separating the interface of an abstraction from its implementation
- The use of encapsulation makes it easier to create abstractions.

Stockholm University

Information Hiding

- A design principle
  - Hide data, structure and any differences between exposed data and internal representation

- What abstractions we use controls what information should be hidden

Stockholm University

Data Abstraction

- An **abstract data type** is a user-defined data type that satisfies the following:
  - The **representation** of and **operations on** objects of the type are defined in a single syntactic unit
    - other units can create objects of the type.
  - The **representation** of objects of the type is kept separate from the program units that use these objects, so the only operations possible are those provided in the type's definition.

Possible with abstraction, not necessary...

Stockholm University

Data Abstraction (Continued)

- Advantage of Restriction 1:
  - Same as those for encapsulation: program organization, modifiability (everything associated with a data structure is together), and separate compilation
- Advantage of Restriction 2:
  - Reliability - by hiding the data representations, user code cannot directly access objects of the type. User code cannot depend on the representation, allowing the representation to be changed without affecting user code.

Well, let's remember this and see how it works...

Stockholm University

Language Example C++

- Based on C struct type and Simula 67 classes where the class is the encapsulation device
- All of the class instances of a class share a single copy of the member functions
- Each instance of a class has its own copy of the class data members
- Instances can be static, stack dynamic, or heap dynamic

Language Examples C++ (cont'd)

Stockholm University

- Information Hiding:
  - Private clause for hidden entities
  - Public clause for interface entities
  - Protected clause - for inheritance
- Constructors:
  - Functions to initialize the data members of instances (they DO NOT create the objects)
  - May also allocate storage if part of the object is heap-dynamic
  - Can include parameters to provide parameterization of the objects
  - Implicitly called when an instance is created
  - Can be explicitly called
  - Name is the same as the class name

Stockholm University

Language Example C++ (cont'd)

- Destructors
  - Functions to cleanup after an instance is destroyed; usually just to reclaim heap storage
  - Implicitly called when the object's lifetime ends
  - Can be explicitly called
  - Name is the class name, preceded by a tilde (~)

Stockholm University

Language Example Java

- "Similar" to C++, except:
  - All user-defined types are classes
  - All objects are allocated from the heap and accessed through reference variables
  - Individual entities in classes have access control modifiers (private or public), rather than clauses

Stockholm University

Python

- Only name mangling
  - Not really reliable
  - Problem with renaming methods

```
>>> class Example:
      def __method(self):
        print "Deeo"

>>> ex = Example()
>>> dir(ex)
['_Example__method', '__doc__',
                              '__module__']
>>> ex._Example__method()
Deeo
```


Stockholm University

Ruby

- Name-based information hiding
- Involves expensive dynamic checking
- Information hiding can be circumvented
  - Removed by subclass
  - Ignored by reflection

```
> class Example
>   private
>   def method; print "Deeo"; end
> end

> ex = Example.new
> ex.method
NoMethodError: private method `method' \
           called for #<Example:0x2223d0>
```

Stockholm University

Not-so-very-private Ruby

```
> ex.send("method")
Deeo

> def ex.back_door; method; end
> ex.back_door
Deeo

> class Sub < Example
>   def method; super; end
> end
```


Stockholm University

Smalltalk

- All methods are public
- All member variables are private

Stockholm University

Io

- Private could be simulated by explicitly checking sender in every private method
- Expensive
- Not visible from the outside

Stockholm University

Conclusion?

- Encapsulation with information hiding is perhaps
  - not compatible with being highly dynamic
  - too expensive in a dynamic setting
  - not (so) important in the domains where dynamic languages are used?

The End

References

- Sebesta, R - “Concepts of Programming Languages”
- Louden, K - “Programming Languages: Principle and Practice”
- Scott, M - “Programming Language Pragmatics”
- Parnas D.L. - "On the Criteria To Be Used in Decomposing Systems into Modules"

Thursday, November 10, 11