Data Abstraction

I felt like de Kooning, who was asked to comment on a certain abstract painting, and answered in the negative. He was then told it was the work of a celebrated monkey. 'That's different. For a monkey, it's terrific.' -- Igor Stravinsky

- Encapsulation 10.2
- Abstract data types 10.3
- Language examples 10.5
- Parameterized abstract data types 10.6

Data Types

We once defined a *data type* as a specification of the kinds of values a variable can store and a specification of the kinds of operations that can be performed on it.

- *data type*
  - a set of objects *and* a set of operations on the objects
  - implementation details of both object and operations hidden
  - integer
    - 2 bytes?
    - Ho-Lo?
    - 2's complement?
- *user-defined data type*
  - a programmer can *extend* the data abstraction of built-in types
  - no *systematic* way to specify the operations

Abstract Data Types (ADTs)

- *user-defined abstract data type*
  - a programmer can extend the data abstraction of built-in types
  - systematic way of defining the data type
  - systematic way of defining operations on the type *and* associating them with the type
  - mechanism for hiding the implementation details of the type and its operations from “clients” (programs that use the ADT)
  - mechanism for specifying the communication of information between the type and its clients
    - ADT must *export* some information to the client
    - client must *import* some information from the ADT

Encapsulation

Encapsulation provides *some* of the features required of abstract data types.

- Encapsulation
  - grouping subprograms and their data in a separate unit by themselves
  - provides organization of a large program into logically related groups of subprograms and data
  - allows parts of large programs to be compiled separately (and not recompiled unnecessarily)
  - allows subprograms to be compiled and distributed without source code
    - major abstraction!

Encapsulation in C

m2.c

```
#include <stdio.h>

int X;

void f1(char *caller)
{
 printf("I'm in f1 in m2 again (%d times)\n", X++);
 printf("My caller was %s\n", caller);
}
```

m1.c

```
extern int X;
extern void f1(char *caller);

void f2()
{
 char me[9] = "f2 in m1";

 f1(me);
}

void main()
{
 char me[11] = "main in m1";

 X = 1;
 f1(me);
 f2();
}
```

```
{tamale1}kbarker(42) gcc -c -o m2.o m2.c
{tamale1}kbarker(43) gcc -o main m1.c m2.o
{tamale1}kbarker(44) main
I'm in f1 in m2 again (1 times)
My caller was main in m1
I'm in f1 in m2 again (2 times)
My caller was f2 in m1
{tamale1}kbarker(45)
```


Encapsulation in Prolog

m1.pl

```
:- multifile ctr_set/2, ctr_inc/1.
:- ensure_loaded(m2).

count_em :-
   ctr_set(i, 0),
   f(_, _),
   ctr_inc(i),
   fail.
count_em :-
   i(N),
   write(N), nl.
```

m2.pl

```
:- multifile ctr_set/2, ctr_inc/1.

ctr_set(Ctr, N) :-
   C1 =.. [Ctr, _],
   retractall(C1),
   C =.. [Ctr, N],
   assert(C).

ctr_inc(Ctr) :-
   C =.. [Ctr, N],
   retract(C),
   N1 is N + 1,
   C1 =.. [Ctr, N1],
   assert(C1).
```

```
| ?- compile(m1).
% compiling file m1.pl
%  compiling file m2.pl
%  m2.pl compiled in module user, 0.010 sec 480 bytes
% m1.pl compiled in module user, 0.020 sec 1,404 bytes

yes
| ?- count_em.
7

yes
| ?-
```

Defining Abstract Data Types

An *abstract data type* is an encapsulation that contains a specific data type and its operations.

- *specification*
  - name of the data type
  - names of the operations
    - parameter types
    - return types
- *representation*
  - definition of the data type in terms of other types
    - built-in types
    - other ADTs
- *implementation*
  - process details of operations (bodies of subprograms)

Communicating with an ADT

One of the most important aspects of an ADT is information hiding: some information is hidden inside the ADT encapsulation, but some must be available to clients.

- *export*
  - which operations and data objects are to be visible to potential clients
  - usually *specification* only (not *representation* or *implementation*)
  - may be implicit
    - all of specification is exported
  - or may be explicit
    - certain operations and data objects named as exports
- *import*
  - which of the exports of an ADT does the client request to see

What's the point of hiding details?

The First Example

The first example is always a *stack* ADT. We'll use a hypothetical notation (not a particular language):

```
adt stack(item);
operations
  newstack()        →   stack;
  push(stack, item) →   stack;
  pop(stack)        →   stack;
  top(stack)        →   item;
  is_empty(stack)   →   boolean;
var  s: stack;  i: item;
conditions
  pop(newstack()) = newstack();
  pop(push(s, i)) = s;
  top(push(s, i)) = i;
  is_empty(newstack()) = true;
  is_empty(push(s, i)) = false;
errors
  top(newstack())
end stack;
```

ADT name + component

operation "signatures"

local names

abstraction of representation and implementation

Discussing the First Example

- operation signatures
  - the names of operations
  - the types of their parameters
  - the types of their return values
- conditions and errors
  - an abstraction of the representation and implementation sections of an ADT
  - a feature of our hypothetical notation only (not of a real language)
- adt name
  - the name of the ADT (stack)
  - the name of its component (a stack of what?)
  - the type of the component will have to be defined

Wouldn't it be nice to leave the component type unspecified?

Visualizing Stacks

The notion of stack in our *stack* ADT is a bit unusual. A stack is described as the recursive pushing of its items:

- stack visualization

```
42
 5
13
```

- ADT stack

```
push(push(push(newstack(), 13), 5), 42)
```

All stacks can be described in terms of two operations: newstack and push. These are the *primitive operations* for the stack ADT.

Using the Stack ADT

Here are some possible expressions involving our *stack* ADT

- how a client in our hypothetical, highly abstract language might use the ADT

- `newstack()`
- `push(newstack(), 13)`
- `push(push(newstack(), 13), 5)`
- `top(push(push(newstack(), 13), 5)) = 5`
- `pop(push(push(newstack(), 13), 5)) = push(newstack(), 13)`
- `is_empty(push(push(newstack(), 13), 5)) = false`

The Second Example

```
adt queue(item);
operations
  newq()              →  queue;
  addq(queue, item)   →  queue;
  delq(queue)         →  queue;
  frontq(queue)       →  item;
  is_emptyq(queue)    →  boolean;
var  q: queue;  i: item;
conditions
  frontq(addq(q, i))     = if   is_emptyq(q) then i
                           else frontq(q);
  delq(newq())           = newq();
  delq(addq(q, i))       = if   is_emptyq(q) then newq()
                           else addq(delq(q), i);
  is_emptyq(newq())      = true;
  is_emptyq(addq(q, i))  = false;
errors
  frontq(newq())
end queue;
```


Visualizing Queues

Like a stack, a queue is described as the recursive adding of its items:

- queue visualization `13 5 42`
- ADT queue

  `addq(addq(addq(newq(), 13), 5), 42)`

What are the primitive operations for our queue *ADT?*

Using the Queue ADT

Here are some possible expressions involving our *queue* ADT

- `newq()`
- `addq(newq(), 13)`
- `addq(addq(newq(), 13), 5)`
- ```
  frontq(addq(addq(newq(), 13), 5))
    = frontq(addq(newq(), 13))
    = 13
  ```
- ```
  delq(addq(addq(newq(), 13), 5))
    = addq(delq(addq(newq(), 13)), 5)
    = addq(newq(), 5)
  ```
- `is_emptyq(addq(addq(newq(), 13), 5)) = false`

The Third Example

```
adt bst(item);
operations
  newt()               →  bst;
  make(bst, item, bst) →  bst;
  left(bst)            →  bst;
  val(bst)             →  item;
  right(bst)           →  bst;
  insert(item, bst)    →  bst;
  isnewtree(bst)       →  boolean;
  intree(item, bst)    →  boolean;
var  L: bst;  R: bst;
     i: item; j: item;
conditions
  left(make(L, i, R))       = L;
  right(make(L, i, R))      = R;
  val(make(L, i, R))        = i;
  insert(j, newt())         = make(newt(), j, newt());
  insert(j, make(L, i, R)) = if j = i then make(L, i, R);
                             if j < i then make(insert(j, L), i, R);
                             if j > i then make(L, i, insert(j, R));
```

etc.

What are the primitive operations for the bst *ADT?*

Using the BST ADT

- create an empty tree
  - `newt()`
- insert 13
  - `insert(13, newt())`
  - `= make(newt(), 13, newt())`
- insert 5
  - `insert(5, make(newt(), 13, newt()))`
  - `= make(insert(5, newt()), 13, newt())`
  - `= make(make(newt(), 5, newt()), 13, newt())`
- insert 42
  - `insert(42, make(make(newt(), 5, newt()), 13, newt()))`
  - `= make(make(newt(), 5, newt()), 13, insert(42, newt()))`
  - `= make(make(newt(), 5, newt()), 13, make(newt(), 42, newt()))`

Language Example: Modula-2

An ADT in Modula-2 is written in two parts (called *modules*):

- *Definition module*
  - corresponds to *specification*
    - ADT name
    - subprogram signatures
  - *all* names in the definition module are exported
- *Implementation module*
  - corresponds to *representation* and *implementation*
    - type definition(s)
    - subprogram bodies

Backdoor polymorphism

- export types whose definitions appear only in the implementation module are called *opaque types*
  - all opaque types must be *pointers* or *synonyms* of other opaques

A Modula-2 Queue...

```
defintion module integer_q_module;
  type queue;
  procedure newq: queue;
  procedure addq(q: queue; i: integer): queue;
  procedure delq(q: queue): queue;
  procedure frontq(q: queue): integer;
  procedure is_emptyq(q: queue): boolean;
end integer_q_module;

implementation module integer_q_module;
  type q_ptr  = pointer to q_node;
       q_node = record
                  elem: integer;
                  next: q_ptr
                end;
       q_rec  = record
                  fr, tl: q_ptr
                end;
       queue  = pointer to q_rec;
```

279

A Modula-2 Queue Continued...

```
(* still inside implementation module integer_q_module *)
  procedure newq: queue;
    var q: queue; p: q_ptr;
  begin
    new(p);
    p^.next := nil;
    q^.fr := p;
    q^.tl := p;
    return q;
  end;

  procedure addq(q: queue; i: integer): queue;
    var p: q_ptr;
  begin
    q^.tl^.elem := i;
    new(q^.tl^.next);
    q^.tl := q^.tl^.next;
    q^.tl^.next := nil;
    return q;
  end;
```

280

A Modula-2 Queue Continued

```
(* still inside implementation module integer_q_module *)
  procedure delq(q: queue): queue;
  begin
    if q^.fr <> q.tl then
      q^.fr := q^.fr^.next;
    return q;             (* should raise exception on else *)
  end;

  procedure frontq(q: queue): integer;
  begin
    if q^.fr <> q^.tl then
      return q^.fr^elem;
  end;                    (* should raise exception on else *)

  procedure is_emptyq(q: queue): boolean;
  begin
    return q^.fr = q^.tl;
  end;

end integer_q_module;
```

281

Using A Modula-2 Queue

```
module main;
  from integer_q_module
    import addq, delq, newq, frontq, is_emptyq, queue;
  from InOut
    import Read, ReadLn, EOL, ReadInt, WriteLn, WriteInt;
  var my_queue: queue;
      elem: integer;
begin
  my_queue := newq;
  my_queue := addq(my_queue, 13);
  my_queue := addq(my_queue, 5);
  my_queue := addq(my_queue, 42);
  if ~ is_emptyq(my_queue) then begin
    elem := frontq(my_queue);
    my_queue := delq(my_queue);
  end;
  ...
end main;
```

282

Language Example: Java

Abstract data types are supported directly in Java as *classes*

- A Java class is a type
  - programs declare variables as class types
  - a variable is an *instance* of the class type
  - class instances are called *objects*
- A Java class contains
  - *instance variables* (data defined inside the class)
    - each instance gets its own set of instance variables
  - *methods* (subprograms defined inside the class)
    - all instances share a single set of methods
- Information hiding
  - variables and methods in a class can be *private* or *public*
    - *private* variables and methods have *class scope* (hidden from clients)
    - *public* variables and methods are exported

A Java Stack

```
class stack {
  private int[] s;
  private int stacksize, topofstack;
  public stack() {
    stacksize = 100;
    s = new int[stacksize];         // each instance gets its own variables
    topofstack = -1;
  }
  public void push(int i) {
    if(topofstack < stacksize - 1)
      s[++topofstack] = i;
  }                                 // should raise exception on else
  public void pop() {
    if(topofstack >= 0)
      topofstack--;
  }                                 // should raise exception on else
  public int top() {
    if(topofstack >= 0)
      return(s[topofstack])
  }                                 // should raise exception on else
  public boolean is_empty() {
    return(topofstack == -1);
  }
}
```

Using A Java Stack

```
import java.io.*;

public class use_it {
  public static void main() {
    int elem;
    stack mystack = new stack();

    mystack.push(13);
    mystack.push(5);
    mystack.push(42);

    System.out.println("The top was: " + mystack.top());
    mystack.pop();

    elem = mystack.top();
    System.out.println("Then the top was: " + elem);
    mystack.pop();
  }
}
```


Parameterized ADTs

We already saw how Ada and C++ allow parameterization of subprogram types (♬ 245). What better place to use the ideas of parametric polymorphism than with abstract data types!

- using *Ada generics* or *C++ templates* allows us to define...
  - an *abstract stack* data type that stacks any element type
  - an *abstract stack* data type that stacks any number of elements
  - an *abstract queue* data type that queues any element type
  - an *abstract queue* data type that queues any number of elements
  - an *abstract binary search tree* type that can have any type as node
  - etc.

A C++ Parametric Stack

```
template <class Type>
class stack {
  private:
    Type *s;
    int stacksize, topofstack;
  public:
    stack(int stacksize) {
      s = new Type[stacksize];
      topofstack = -1;   }

    ~stack() {
      delete s;   }

    void push(Type i) {
      if(topofstack < stacksize - 1)
        s[++topofstack] = i;   }            // should raise exception on else

    void pop() {
      if(topofstack >= 0)
        topofstack--;   }                   // should raise exception on else

    int top() {
      if(topofstack >= 0)
        return(s[topofstack]);   }          // should raise exception on else

    int is_empty() {
      return(topofstack == -1);   }
}
```


Using A C++ Parametric Stack

```
#include <iostream.h>

void main() {
  stack(99) mystack1;
  stack(200) mystack2;

  mystack1.push(13);           // compiler generates code for int
  mystack1.push(5);            // version of the member functions
  mystack1.push(42);

  cout << "The top was: " << mystack1.top() << endl;
  mystack1.pop();

  mystack2.push('s');          // compiler generates code for char
  mystack2.push('e');          // version of the member functions
  mystack2.push('n');

  cout << "The top was: " << mystack2.top() << endl;
  mystack2.pop();
}
```

An Ada Parametric BST...

```
generic
  type item is private;
  with function "<"(X, Y: item) return boolean;

package bst_package is
  type bst is limited private;
  function newt return bst;
  function make(L: bst; i: item; R: bst) return bst;
  function left(T: bst) return bst;
  function val(T: bst) return item;
  function right(T: bst) return bst;
  function insert(i: item; T: bst) return bst;
  function isnewtree(T: bst) return boolean;
  function intree(i: item; T: bst) return boolean;

private
  type node is record
    left: bst; info: item; right: bst;
  end record;
  type bst is access node;

end bst_package;
```


An Ada Parametric BST Continued

```
package body bst_package is
  function newt return bst is begin
    return null;
  end newt;

  function make(L: bst; i: item; R: bst) return bst is begin
    return new bst'(L, i, R);
  end make;

  function left(T: bst) return bst is begin
    return T.left;
  end left;

  function val(T: bst) return item is begin
    return T.info;
  end val;

  function right(T: bst) return bst is begin
    return T.right;
  end right;

  function insert(i: item; T: bst) return bst is begin
    if T = null then return make(newt, i, newt); end if;
    if i < T.info then T.left := insert(i, T.left); end if;
    if T.info < i then T.right := insert(i, T.right); end if;
    return T;
  end insert;
end bst_package;
```

Using an Ada Parametric BST

```
with bst_package;
use bst_package;

package int_bst is new bst_package(integer, "<");
package string_bst is new bst_package(string(1..20), "<");
use int_bst, string_bst;

procedure main is
  i: integer;
  s: string(1..20);
  t1: int_bst.bst;
  t2: string_bst.bst;
begin
  i := 13;
  t1 := int_bst.newt;
  t1 := int_bst.insert(i, t1);
  t1 := int_bst.insert(5, t1);
  t1 := int_bst.insert(42, t1);
  s := "Ron";
  t2 := string_bst.newt;
  t2 := string_bst.insert(s, t2);
  t2 := string_bst.insert("Sami", t2);
  t2 := string_bst.insert("Marian", t2);
end main;
```