Modularity and Object-Oriented Programming — Modules

Steven Zeil

Oct. 27, 2003

Contents

Modularity

1. Program Structuring
2. Information Hiding
3. Modules
4. Classes
5. Polymorphic Modules

---

1 Program Structuring

1. Modules
2. User Defined Data Types
3. Abstract Data Types

1.1 Modules

"Modules" arose in response to increasing size of programs.

- Just as procedures group related statements
- Modules group related declarations
  - procedures
  - types
  - objects (variables and constants)

What is a Module?

- "Module" first used as a loose term for either
  - a single procedure, or
  - a group of related procedures with associated other declarations

---

- Design techniques began to highlight groupings into modules
- HLL's eventually followed suit, driven by idea of ADT

Why Modules?

- control of name space
- well-defined interfaces
- division of responsibilities
- Encourages hierarchical style, making application code more readable.

1.2 User Defined Data Types

- Data Type
- User Defined Data Type
- Abstraction

1.2.1 Data Type

We have previously defined a **data type** as a collection of values with an associated representation and a set of permitted operations.

- In early HLL's the set of operations on the type consists of
  - The op's permitted by the HLL on the representation, and
  - any user-written procedures that manipulate objects of that type
- Often a broader set than a designer would like.

1.2.2 User Defined Data Type

We may distinguish between those types that are

- **primitive**, supplied by the HLL
- **User Defined Data Types** (UDDT), defined by the programmer using type expressions to describe a representation

---

Example:

```
type MailingList is
   array[1..NumPeople] of Address;
```

1.2.3 Abstraction

An **abstraction** is a mental model in which certain details are ignored in order to get at the "essential" idea.

- **Procedural abstraction** is a mental model of what a procedure should do (ignoring *how* it does it).
- **Data abstraction** is a mental model of how a collection of data behaves.
  - UDDT's are often intended to capture some data abstraction.

1.3 Abstract Data Types

1. Definition
2. ADT as contract

1.3.1 Definition

An **abstract data type** (ADT) is a type name and a set of operations on that type.

ADT versus DT

- An ADT is not a data type, because it has no representation.
- We **implement** an ADT by supplying a representation for the data and algorithms for the operations.
  - An ADT implementation *is* a data type.

Another way of looking at it:

- An ADT is a design concept
- A DT is a programming language concept
- An ADT implementation is the combination of the two.

1.3.2 ADT as contract

An ADT represents a contract between the ADT developer and the users (application programmers).

- Users may alter/examine values of this type only via the operations provided.
- The developer promises to leave the ADT specification unchanged.
- The developer may change the implementation of the ADT at any time.

ADTs came to dominate much of the thinking about program design.

By adhering to the contract,

- Users can be designing and even implementing the application before the details of the ADT implementation have been worked out. This helps in
  - top-down design
  - development by teams
- The ADT implementors knows exactly what they must provide and what they are allowed to change.
- ADT's designed in this manner are often re-usable. By reusing code, we save time in
  - implementation
  - testing and debugging

- We gain the flexibility to try different engineering for the ADT, without needing to alter the application code.

A Sample Module: Modula 2

```
DEFINITION MODULE WordCounts;
  CONST LineWidth = 80;
        WordLength = 24;

  TYPE Table;

  PROCEDURE InitTable (VAR t: Table);
  PROCEDURE Insert (
    VAR t: Table;
    VAR word: ARRAY OF CHAR;
     count: INTEGER);
  PROCEDURE Find (
     t: Table;
    VAR word: ARRAY OF CHAR)
     : INTEGER;
  PROCEDURE Size(t: Table)
     : INTEGER;
END WordCounts.
```

```
IMPLEMENTATION MODULE WordCounts;

 FROM Storage IMPORT Allocate;

 CONST TableSize = 3000;

 TYPE Entry = RECORD
    word: ARRAY[0..WordLength] OF CHAR;
    count: INTEGER;
  END;
  TableBody = RECORD
    size: INTEGER;
    data: ARRAY[1..TableSize] OF ENTRY;
  END;
  Table=POINTER TO TableBody;

 PROCEDURE InitTable (VAR t: Table);
 BEGIN
  t := Allocate(t, SIZE(TableBody));
  t^.size := 0;
 END;
    ⋮
END WordCounts;
```

---

2 Information Hiding

1. Motivation
2. Encapsulation
3. User Defined Data Types
4. Abstract Data Types

2.1 Motivation

Every design can be viewed as a collection of "design decisions".

Early work in software design noted that

- widely separated procedures often were inconsistent because they assumed different decisions.

- ...

- if design decisions were later changed, finding all the affected code was a major difficulty.

Information Hiding & Design

David Parnas formulated the principle:

> Every module [procedure] should be designed so as to hide one design decision from the rest of the program.

He argued that such **information hiding** made future changes more economical.

Information hiding predates ADTs, but can be applied even to procedural design.
Example: a calculator program
Consider the design decisions:

- read from standard input
- write to standard output
- output expression is in postfix form
- will accept `+-*/`, but not $**$, `sqrt()`, ...
- data structures for various expression node kinds

Information hiding predates ADTs, but can be applied even to procedural design.
Example: a calculator program
Consider the design decisions:

- read from standard input `file openInput()`
- write to standard output `file openOutput()`
- output expression is in postfix form `printPostfix`
- will accept `+-*/`, but not $**$, `sqrt()`, ...`parse()`, `evaluate()`
- data structures for various expression node kinds — no one function can hide this

---

- Modules help provide a grouping when functions must cooperate to hide a design decision.
- Hiding data structures is particularly appropriate for ADTs
  - Consequently, modules are often designed around ADT.
  - The ADT "contract" is essentially concerned with info hiding.

2.2 Encapsulation

Although "modules" can be designed without language support, they rely on programmers' self-discipline for enforcement of information hiding.

Encapsulation is the enforcement of information hiding by programming language constructs.

Refer again to the Modula 2 table:

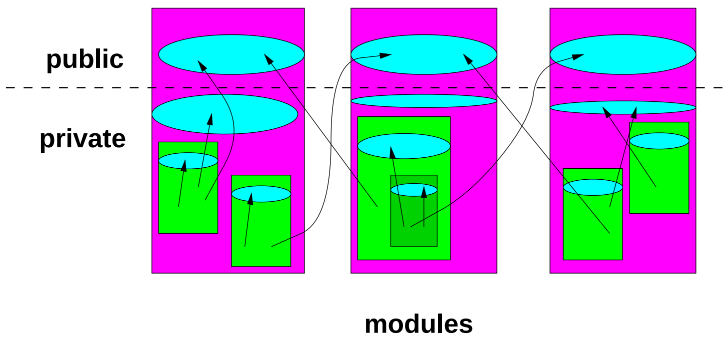
```
DEFINITION MODULE WordCounts;
  CONST LineWidth = 80;
        WordLength = 24;

  TYPE Table;

  PROCEDURE InitTable (VAR t: Table);
  PROCEDURE Insert (
     VAR t: Table;
     VAR word: ARRAY OF CHAR;
     count: INTEGER);
  PROCEDURE Find (
     t: Table;
     VAR word: ARRAY OF CHAR)
     : INTEGER;
  PROCEDURE Size(t: Table)
     : INTEGER;
END WordCounts.
```

Note that `Table`'s representation does not appear anywhere in the module definition.

- Application code imports module definitions.
- Module implementations are compiled separately.

Therefore everything in the module implementation is hidden from the rest of the program.

- A very pure form compared to other HLL's.



In Modula 2, any type declared in the module definition, but not given a representation until the module implementation, is called an **anonymous type**.

Anonymous types *must* be implemented as a `POINTER TO` something.

Why do you think this is the case?

3 Modules

1. Scope and Encapsulation
2. Varying Roles

3.1 Scope and Encapsulation

Modules provide encapsulation by modifying the scope rules for declarations occurring within them.

- Some declarations are **visible** to application code. Some are **invisible**.
- In many languages, modules can nest.
- ...

- Declarations have fully qualified names, e.g., `WordCounts.Table`, `java.awt.Graphics.paint()`

- In limited contexts, fully qualified names can often be shortened.
- C++ is unusual in using a different operator for name qualification than is used for record field selection: `std::string::iterator`, `std::cout.flush()`

Mechanisms for encapsulation in modules:

1. Separate Specification
2. Import and Export

3.1.1 Separate Specification

The collection of declarations that are visible to application code is the module's **specification**. The remainder of the module is the **implementation**.

One simple mechanism for encapsulation is to separate the spec. from the impl.

- Application code sees only the spec.
- Modula 2 relies on this separation.
- Similar separation is used in Ada and C++, but these use other mechanisms as well.

3.1.2 Import and Export

Pure separation not only hides info from applications, but also from the compiler.

- Leads to compromises, as in the Modula 2 anonymous types
- Forces function calls rather than inlining
  - problem since many ADTs have some trivial op's

Problems resolved via import/export controls.

Selective Export

An earlier version of Modula 2 required explicit EXPORT lists:

```
DEFINITION MODULE WordCounts;

  EXPORT Table, InitTable, Insert,
         Find, Size;

  CONST LineWidth = 80;
        WordLength = 24;

  TYPE Table;

  PROCEDURE InitTable (VAR t: Table);
```

```
  PROCEDURE Insert (
    VAR t: Table;
    VAR word: ARRAY OF CHAR;
    count: INTEGER);
  PROCEDURE Find (
    t: Table;
    VAR word: ARRAY OF CHAR)
    : INTEGER;
  PROCEDURE Size(t: Table)
    : INTEGER;
END WordCounts.
```

`LineWidth` and `WordLength` are invisible to applications.

- Modula 2 dropped this requirement because practice showed that programmers wanted to export everything in the spec.
  - Otherwise they'd have put it in the impl.

- Other languages adopted the idea in a less explicit form:
  - making regions of the spec. public or private, or
  - labeling individual declarations as public or private.

Export Control in Java

- Java does not separate module spec. and impl.
- It relies exclusively on export control for encapsulation.
- Declarations may be labeled private or public
  - default is "visible within same package"

```
class WordCounts {
  private int Size;
  private String[] keys;
  private int[] counts;

  public WordCounts () {
   Size = 0;
   keys = new String[TableSize];
   counts = new int[TableSize];
   }

  void insert (String word, int count) {
    ⋮ }

  int find (String word, int count) {
    ⋮ }

  int size() {return Size;}
}
```

Selective Import

All modular languages allow a programmer to control which module spec's are imported.

Some allow further control over which symbols from those specs to import.

```
MODULE Application;

 IMPORT WordCounts;
 FROM InOut IMPORT EOL, Read, Write;

 VAR t: Table; /* from WordCounts */
BEGIN
  InitTable (t);
  Write ("Hello ");
  InOut.WriteLn ("World");
END Application;
```

- `IMPORT` by itself imports all symbols from the module spec.
- `FROM...IMPORT` imports selected symbols. Others must be accessed by fully qualified names.

3.2 Varying Roles

Modules are often organized around

- groups of procedures and other declarations that perform related functions
- a data object
- an ADT
- groups of related ADTs

3.2.1 Module Samples: Ada

1. A Math Library
2. A Queue Object
3. A Queue ADT

A Math Library

```
package MathLib is
  Pi: constant := 3.14159_26535_89793;
  function Sqrt (X: real) return real;
  function Log (X: real) return real;
  function Log (X, Base: real) return real;
  function Exp (X: real) return real;
  function "**" (X: real) return real;
end MathLib;
```

Elsewhere, a package body would be supplied to provide the function implementations:

```
package body MathLib is

  function Sqrt (X: real) return real is
  begin
    ...
  end Sqrt;
  ⋮
end MathLib;
```

Code to use this library might look like

```
with MathLib;
function quadroot(x: real) return real is
begin
  return MathLib.sqrt(MathLib.sqrt(x));
end quadroot;
```

or

```
with MathLib; use MathLib;
function quadroot(x: real) return real is
begin
  return sqrt(sqrt(x));
end quadroot;
```

A Queue Object

```
package One_Integer_Queue is

  Queue_Is_Empty: exception;
  Queue_Is_Full: exception;

 procedure Add_To_End
     (An_Element: in integer);
 function Front return integer;
 function Is_Empty return boolean;
 procedure Remove_Front;

end One_Integer_Queue;
```

```
package body One_Integer_Queue is
    Q_Size: constant integer := 1000;

    Q: array(0..Q_Size-1) of integer;
    Front, Back: positive;

 procedure Add_To_End
   (An_Element: in integer) is
 begin
   if Front /= Back then
     Q(Back) := An_Element;
     Back := (Back + 1) mod Q_Size;
   else
     raise Queue_Is_Full;
   end if;
 end Add_To_End;
 ⋮
end One_Integer_Queue;
```

A Queue ADT

```
package Integer_Queues is

type Integer_Queue is private;

Queue_Is_Empty: exception;

procedure Add_To_End
   (An_Element: in integer;
    Of_The_Queue: in out Queue);

function Create return Queue;
function Front (Of_The_Queue: Queue)
   return integer;
function Is_Empty (The_Queue: Queue)
   return boolean;
procedure Remove_Front
   (Of_The_Queue: in out Queue);

private -- HIDDEN ENTITIES

    type Queue_Node;
    type a_Queue is access Queue_Node;

    type Queue is record
        Front: a_Queue;
```

```
        Back: a_Queue;
    end record;

end Integer_Queues;
```

```
with Unchecked_Deallocation;
package body Integer_Queues is

type Queue_Node is record
   Value: integer;
   Link: a_Queue;
end record;

procedure Add_To_End
    (An_Element: in integer;
     Of_The_Queue: in out Queue) is

  New_Node: a_Queue :=
   new Queue_Node'(Value => An_Element,
                   Link => null);
begin
 if Of_The_Queue.Front = null then
    Of_The_Queue.Front := New_Node;
 else
    Of_The_Queue.Back.Link := New_Node;
 end if;
 Of_The_Queue.Back := New_Node;
end Add_To_End;
  ⋮
```

---

4 Classes

1. **??**
2. Classes in C++
3. Namespaces in C++

4.1 Classes in C++

The C++ class combines encapsulation with type declaration by adding to ordinary `struct`s:

- convenient syntax for function members
  - automatically initialized
  - implicit `this` pointer as first argument
  - implicit `this->` when referring to own members
- export control via `public` and `private`
  - `protected` also possible, used with inheritance
  - `private` control can be relaxed by naming other functions and classes as “friends”

4.1.1 Other C++ class features

- initialization via **constructors**
- finalization via **destructors**
- special syntax: `const` applied to a member function for class `C` indicates that `this` has type `const C*` rather than `C*`
  - Implication is that a const member function cannot alter the object it is used with.
- Members declared as `static` are global to the class.
  - A static data member is, in effect, shared by all objects of that class.
  - Static function members do not get a `this`, because they belong to the class, not to any one object.
  - Other code accessed public static members by qualified name.

4.1.2 Module Samples: C++

1. A Queue ADT
2. A Math Library

A Queue ADT

```
class Integer_Queues {
public:
  Integer_Queues();
  ~Integer_Queues();
  void add_To_End (int an_Element);
  int front () const;
  bool is_Empty () const;
  void remove_Front ();
private:
  ...
```

```
private:
  struct Queue_Node {
    int: value;
    Queue_Node* next;
  };

  Queue_Node* front;
  Queue_Node* back;
}
```

```
#include "integer_queues.h"

void Integer_Queues::add_To_End
    (int an_Element)
{
  Queue_Node* newNode = new Queue_Node;
  newNode->value = an_Element;
  newNode->next = 0;

  if (front == null) {
     newNode->next = front
     front = newNode;
     }
  else
    back->next = newNode;
  back = newNode;
}
  ⋮
```

A Math Library

```
class MathLib {
public:
  static const double Pi = 3.14159_26535_89793;
  static double sqrt (double);
  static double log (double);
  static double log (double base,
                     double x);
  static double exp (double);
  static double pow (double);
}
```

Not very satisfying, because we would need to use fully qualified names in application code

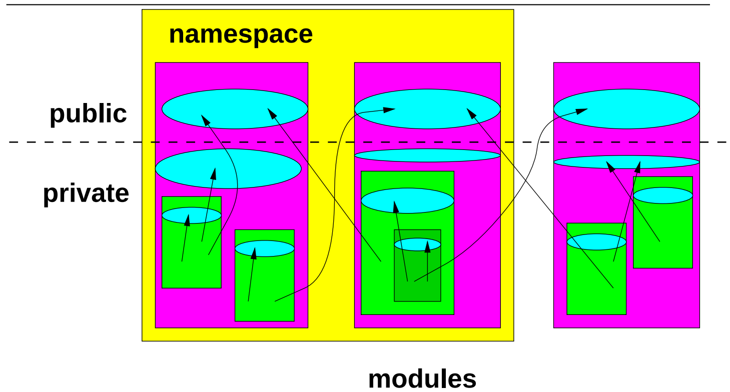
```
z = MathLib::sqrt(y);
```

4.2 Namespaces in C++

This points up a limitation to the class-as-module approach: it works well for modules whose role is to provide ADT's, less well for others.

- A new construct, the C++ **namespace** was added
  - namespaces also provide better overall control of name visibility

---



A namespace is a "pure" module containing types, classes, functions, ...

```
namespace MathLib {
  const double Pi;
  double sqrt (double);
  double log (double);
  double log (double base, double x);
  double exp (double);
  double pow (double);
}
```

As with classes, implementations must identify themselves by fully qualified name:

```
#include "mathlib.h"

const double MathLib::Pi = 3.14159_26535_89793;

const double MathLib::sqrt(double x)
{
  ⋮
}
```

Application code can import names from name spaces

```
#include "mathlib.h"
```

```
using namespace MathLib;
  // imports entire namespace

double quadroot(double x)
{
  return sqrt(sqrt(x));
}

double halfPi()
{
  return Pi/2.0;
}
```

Selective import is also possible:

```
#include "mathlib.h"

using MathLib::log;
  // import only the 2 log functions

double quadroot(double x)
{
  using MathLib::sqrt;
     // import local to { }
  return sqrt(sqrt(x));
}

double halfPi()
{
  return MathLib::Pi/2.0;
}
```

Can't Say Hello?

This code is *not* standard C++:

```
#include <iostream.h>

int main()
{
  cout << "Hello, world!" << endl;
  return 0;
}
```

The C++ language standard created a new namespace, `std`, within which most "predefined" names are placed.

```
#include <iostream.h>   // Error: no such header

int main()
{
  cout << "Hello, world!" << endl; // Error cout & endl
                                   // are undeclared
  return 0;
}
```

We can write

```
#include <iostream>

int main()
{
  std::cout << "Hello, world!" << std::endl;
  return 0;
}
```

---

or we can write

```
#include <iostream>

using namespace std;

int main()
{
  cout << "Hello, world!" << endl;
  return 0;
}
```

---

or we can write

```
#include <iostream>

using std::cout, std::endl;

int main()
{
  cout << "Hello, world!" << endl;
  return 0;
}
```

Language standards generally don't reuqire compilers to reject illegal code.

They only require the compiler to accept legal code.

So many compiler vendors provide the following `iostream.h`:

```
#ifndef IOSTREAM_H
#define IOSTREAM_H
#include <iostream>
using std::iostream;
using std::istream;
using std::ostream;
using std::cin;
using std::cout;
```

```
   ⋮
#endif
```

. . . making following code:

```
#include <iostream.h>    // Compiler-specific

int main()
{
   cout << "Hello, world!" << endl;
   return 0;
}
```

acceptable *on that compiler*.

4.3 Classes in Smalltalk

Smalltalk provides classes without encapsulation (in the literal or technical sense).

Smalltalk was designed to be programmed via a graphical "browser"

- generally lacks syntactic enclosures like

Graphics *Collections* Numerics System	Array Bag *FIFOQueue* Set	*insertion* removal testing printing	*add:* addAll:

```
add: x
  "Adds x to the end of the queue"
  size := size + 1.
  dataArray at: size put: x.
```

5 Polymorphic Modules

These are "patterns" for modules in which one or more names are initially left undefined.

- Application code can **instantiate** a template by supplying bindings for those names.
- In effect, the compiler generates a "real" module by "filling in the blanks" of the pattern using the supplied bindings.

1. Ada Generics
2. C++ Templates

5.1 Ada Generics

```
generic
    type Element is private;
    with function copy (in Element)
       returns Element;
package Queues is

 type Queue is private;

 Queue_Is_Empty: exception;

 procedure Add_To_End
   (An_Element: in Element;
    Of_The_Queue: in out Queue);

 function Create return Queue;
 function Front (Of_The_Queue: Queue)
   return Element;
 function Is_Empty (The_Queue: Queue)
   return boolean;
 procedure Remove_Front
   (Of_The_Queue: in out Queue);

private

    type Queue_Node;
    type a_Queue is access Queue_Node;

    type Queue is record
        Front: a_Queue;
        Back: a_Queue;
    end record;
end Queues;
```

Application code explicitly instantiates a generic:

```
package IntQueues is
   new Queues (Graphs.Vertex,
               copyVertex);

function Dijkstra (g: in out Graph) is
  Q: IntQueues.Queue;
begin
```

5.2 C++ Templates

A similar mechanism in C++ is the **template**:

```
template <class Element>
class Queue {
public:
   Queue();
   ~Queue();
```

```
  void add_To_End
     (const Element& an_Element);
  Element front () const;
  bool is_Empty () const;
  void remove_Front ();
private:
  ⋮
```

```
private:
  struct Queue_Node {
    Element: value;
    Queue_Node* next;
  };

  Queue_Node* front;
  Queue_Node* back;
}
```

Application code explicitly instantiates class templates:

```
void dijkstra (Graph& g)
{
  Queue<Graphs::Vertex> Q;
  Q.add_to_End(g.start());
  ⋮
```

One difference between Ada generics and C++ templates is that

- in Ada, the generic must explicitly list as parameters any functions other than `:=` that it will use on the generic parameter types.
  - Application code can rename when it instantiates
- in C++, the compiler, as it instantiates the code, takes silent note of any functions used and assumes that the data type will supply a function of that name.

Both Ada and C++ also allow generic/template functions.

```
generic
  type T is private;
  with function "<" (x, y: in T)
     return boolean is <>;
function max (a, b: in T) returns T is
begin
  if (a < b) then
    return b;
  else
    return a;
end max;
```

```
template <class T>
T max (const T& x, const T& y)
{
  return (x < y) ? y : x;
}
```

In Ada, function generics are explicitly instantiated:

```
function maxInt is new Max(integer);
  ⋮
i := maxInt(k, 0);
```

In C++, function generics are implicitly instantiated:

```
int i, k;
string s, t, u;
  ⋮
i = max(k, 0); // max<int>
s = max(t, u); // max<string>
```