SE 2A04 Fall 2002

02 Software Modules

Instructor: W. M. Farmer

Revised: 03 October 2002

What is a Software Module?

- Modules are relatively self-contained systems that can be combined to make large systems (Parnas)

- Design is often the assembly of many previously designed modules (Parnas)
  - Modules are interconnectable and interchangeable parts
  - Modules can be designed, implemented, tested, and changed independently

- A **software module** is a cohesive collection of data and procedures that provides a set of **services** to other modules
  - Programs and procedures are usually not modules
  - Modules usually have state

Components of a Module

A software module has two components:

1. An **interface** that allows other modules to use the services the module provides
   - The interface is a **language** for requesting the services
   - Most of the primitive components of the language are procedures called **interface procedures**, **interface functions**, or **access functions**

2. An **implementation** of the interface that provides the services offered by the module
   - The implementation is hidden from other modules
   - The interface procedures are implemented together and may share data structures
   - The implementation may utilize the services offered by other modules

Examples of Modules

- An **object**
  - Consists of data (**fields**) and procedures (**methods**)
  - Has **state** and **behavior**

- An **abstract data structure**

- An **abstract data type (ADT)**

Structure of an Oberon Module

- Interface:
  - Exported type declarations
  - Exported constant declarations
  - Exported variable declarations (not recommended)
  - Exported procedure declarations

- Implementation:
  - Exported and local types
  - Exported and local constants
  - Exported and local variables
  - Exported and local procedures
  - Exported types, constants, variables, and procedures of the imported modules

An Example Interface

An Oberon interface for a stack module:

```
INTERFACE Stack;
  PROCEDURE Reset();
  PROCEDURE MaxHeight(): INTEGER;
  PROCEDURE Height(): INTEGER;
  PROCEDURE Empty(): BOOLEAN;
  PROCEDURE Full(): BOOLEAN;
  PROCEDURE Push(i: INTEGER);
  PROCEDURE Pop();
  PROCEDURE Top(): INTEGER;
END Stack.
```

Example: Stack as Array (1)

```
(*

Title: Stack as Array

Interface:

INTERFACE Stack;
  PROCEDURE Reset();
  PROCEDURE MaxHeight(): INTEGER;
  PROCEDURE Height(): INTEGER;
  PROCEDURE Empty(): BOOLEAN;
  PROCEDURE Full(): BOOLEAN;
  PROCEDURE Push(i: INTEGER);
  PROCEDURE Pop();
  PROCEDURE Top(): INTEGER;
END Stack.

*)

MODULE Stack;

IMPORT Out;
```

Example: Stack as Array (2)

```
(* Constants and variables *)

CONST max = 1000;                 (* maximum height *)

VAR h : INTEGER;                  (* height of stack *)
    s : ARRAY max OF INTEGER;     (* stack contents *)

(* Exceptions: *)

PROCEDURE EmptyStackException();
BEGIN
  Out.String("Stack.EmptyStackException: The stack is empty.");
  HALT(1)  (* Abort program *)
END EmptyStackException;

PROCEDURE FullStackException();
BEGIN
  Out.String("Stack.FullStackException: The stack is full.");
  HALT(1)  (* Abort program *)
END FullStackException;
```

Example: Stack as Array (3)

```
(* Interface procedures *)

PROCEDURE Reset*();
BEGIN
  h := 0
END Reset;

PROCEDURE MaxHeight*(): INTEGER;
BEGIN
  RETURN max
END MaxHeight;

PROCEDURE Height*(): INTEGER;
BEGIN
  RETURN h
END Height;

PROCEDURE Empty*(): BOOLEAN;
BEGIN
  RETURN Height() = 0
END Empty;
```

Example: Stack as Array (4)

```
PROCEDURE Full*(): BOOLEAN;
BEGIN
  RETURN Height() = MaxHeight()
END Full;

PROCEDURE Push*(i: INTEGER);
BEGIN
  IF ~Full() THEN
    s[h] := i;
    h := h + 1
  ELSE
    FullStackException()
  END
END Push;

PROCEDURE Pop*();
BEGIN
  IF ~Empty() THEN
    h := h - 1
  ELSE
    EmptyStackException()
  END
END Pop;
```

Example: Stack as Array (5)

```
PROCEDURE Top*(): INTEGER;
BEGIN
  IF ~Empty() THEN
    RETURN s[h - 1]
  ELSE
    EmptyStackException()
  END
END Top;


(* Initialization *)

BEGIN
  Reset()
END Stack.
```

Example: Stack as Linked List (1)

```
(*

Title: Stack as Linked List

Interface:

INTERFACE Stack;
  PROCEDURE Reset();
  PROCEDURE MaxHeight(): INTEGER;
  PROCEDURE Height(): INTEGER;
  PROCEDURE Empty(): BOOLEAN;
  PROCEDURE Full(): BOOLEAN;
  PROCEDURE Push(i: INTEGER);
  PROCEDURE Pop();
  PROCEDURE Top(): INTEGER;
END Stack.

*)

MODULE Stack;

IMPORT Out;
```

Example: Stack as Linked List (2)

```
(* Types *)

TYPE

  Stack = POINTER TO StackRec;

  StackRec =
    RECORD
      item: INTEGER;
      rest: Stack
    END;

(* Constants and variables *)

CONST max = 1000;                (* maximum height of stack *)

VAR h: INTEGER;                  (* height of stack *)
    s: Stack;                    (* start of stack list *)
```

Example: Stack as Linked List (3)

```
(* Exceptions: *)

PROCEDURE EmptyStackException();
BEGIN
  Out.String("Stack.EmptyStackException: The stack is empty.");
  HALT(1)  (* Abort program *)
END EmptyStackException;

PROCEDURE FullStackException();
BEGIN
  Out.String("Stack.FullStackException: The stack is full.");
  HALT(1)  (* Abort program *)
END FullStackException;

(* Interface procedures *)

PROCEDURE Reset*();
BEGIN
  s := NIL;
  h := 0
END Reset;
```

Example: Stack as Linked List (4)

```
PROCEDURE MaxHeight*(): INTEGER;
BEGIN
  RETURN max
END MaxHeight;

PROCEDURE Height*(): INTEGER;
BEGIN
  RETURN h
END Height;

PROCEDURE Empty*(): BOOLEAN;
BEGIN
  RETURN Height() = 0
END Empty;

PROCEDURE Full*(): BOOLEAN;
BEGIN
  RETURN Height() = MaxHeight()
END Full;
```

Example: Stack as Linked List (5)

```
PROCEDURE Push*(i: INTEGER);
VAR t: Stack;
BEGIN
  IF ~Full() THEN
    NEW(t);
    t^.item := i;
    t^.rest := s;
    s := t;
    h := h + 1
  ELSE
    FullStackException()
  END
END Push;

PROCEDURE Pop*();
BEGIN
  IF ~Empty() THEN
    s := s^.rest;
    h := h - 1
  ELSE
    EmptyStackException()
  END
END Pop;
```

Example: Stack as Linked List (6)

```
PROCEDURE Top*(): INTEGER;
BEGIN
  IF ~Empty() THEN
    RETURN s^.item
  ELSE
    EmptyStackException()
  END
END Top;


(* Initialization *)

BEGIN
  Reset()
END Stack.
```

The Principles of Modular Design (1)

1. **Separation of Concerns**
   - Different parts of the problem are handled by different modules (**horizontal decomposition**)
   - **What** (i.e., interface) is separated from **how** (i.e., implementation) (**vertical decomposition**)

2. **Abstraction**
   - Key ideas unlikely to change are expressed in the interface
   - Implementation details likely to change are left out of the interface

The Principles of Modular Design (2)

3. **Information Hiding**
   - Design decisions likely to change are hidden from other modules (**design for change**)
   - Each module's implementation is a "**secret**" (Parnas)

4. **Little Languages Method**
   - The interface is designed as a **language** that can solve a family of problems instead of just a single problem
   - More abstract languages are defined in terms of more concrete languages

Hallmarks of a Good Module

- The module is as independent from other modules as possible
- The interface is small and orthogonal
- The interface language is highly expressive
- Implementation details are hidden from other modules
- The data structures of the implementation are accessible only via the interface procedures

Definitional Extensions

- A module M' is an **definitional extension** of a module M if:

  1. M' imports only M and possibly some other modules that provide basic services like input and output
  2. M' does not have a state
  3. The interface components of M' are defined in terms of the interface components of M

- The interface language of M' is intended to be an enrichment of the interface language of M

- Unlike other modules, the interface of a good definitional extension can be large and nonorthogonal