SE 2A04 Fall 2001

Software Modules

Instructor: W. M. Farmer

Revised: 24 October 2001

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
   - The primitive components of the language are usually procedures called **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 functions 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 (interface functions)
- 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

Example: Stack as Array 1

```
MODULE Stack1;

IMPORT Out;

(* INTERFACE *)

(*
PROCEDURE Reset();
PROCEDURE MaxHeight(): INTEGER;
PROCEDURE Height(): INTEGER;
PROCEDURE Empty(): BOOLEAN;
PROCEDURE Full(): BOOLEAN;
PROCEDURE Push(i: INTEGER);
PROCEDURE Pop();
PROCEDURE Top(): INTEGER;
*)
```


Example: Stack as Array 2

```
(* IMPLEMENTATION *)

(* 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("Stack1.EmptyStackException: The stack is empty.");
    HALT(33);  (* Abort program *)
  END EmptyStackException;

PROCEDURE FullStackException();
  BEGIN
    Out.String("Stack1.FullStackException: The stack is full.");
    HALT(33);  (* Abort program *)
  END FullStackException;
```


Example: Stack as Array 3

```
(* Interface functions *)

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
 h := 0;
END Stack1.
```


Example: Stack as Linked List 1

```
MODULE Stack2;

IMPORT Out;

(* INTERFACE *)

(*
PROCEDURE Reset();
PROCEDURE MaxHeight(): INTEGER;
PROCEDURE Height(): INTEGER;
PROCEDURE Empty(): BOOLEAN;
PROCEDURE Full(): BOOLEAN;
PROCEDURE Push(i: INTEGER);
PROCEDURE Pop();
PROCEDURE Top(): INTEGER;
*)
```


Example: Stack as Linked List 2

```
(* IMPLEMENTATION *)

(* 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("Stack2.EmptyStackException: The stack is empty.");
    HALT(33);  (* Abort program *)
  END EmptyStackException;

PROCEDURE FullStackException();
  BEGIN
    Out.String("Stack2.FullStackException: The stack is full.");
    HALT(33);  (* Abort program *)
  END FullStackException;

(* Interface functions *)

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 Stack2.
```

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 set of interface functions 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 functions

Module Design Documents

- **Module Guide**
- For each module:
  - **Module Interface Specification (MIS)**
  - **Module Internal Design (MID)**

Module Guide

- The Module Guide lists all the modules of the software product
- The following information is given for each module:
  1. Module name
  2. Module nickname (2 or 3 letters)
  3. Service: Short informal description of what services the module provides
  4. Secret: Short informal description of what secret the module hides

Components of an Input/Output MIS

1. **Imported modules**
2. **Interface**
   - Types
   - Constant names and types
   - Procedure names and types
3. **Exceptions**
4. **Input/output specification**

Example: MIS for Stacks ADT 1

- Imported modules: none
- Interface:

```
TYPE Stack;
CONST Bottom: Stack;
PROCEDURE Push(i: INTEGER; s: Stack): Stack;
PROCEDURE Top(s: Stack): INTEGER;
PROCEDURE Pop(s: Stack): Stack;
```

- Exceptions:
  - EmptyStack : Stack $\rightarrow *$

Example: MIS for Stacks ADT 2

- Input/output specification (axioms):
  1. **Bottom is not a Push stack.**
     $\forall i : \texttt{INTEGER}, s : \texttt{Stack} \,.\, \texttt{Bottom} \neq \texttt{Push}(i, s)$
  2. **Push is one-to-one.**
     $\forall i_1, i_2 : \texttt{INTEGER}, s_1, s_2 : \texttt{Stack} \,.$
     $\texttt{Push}(i_1, s_1) = \texttt{Push}(i_2, s_2) \supset (i_1 = i_2 \wedge s_1 = s_2)$
  3. **Induction axiom for stacks.**
     $\forall P : \texttt{Stack} \rightarrow \texttt{BOOLEAN} \,.$
     $[P(\texttt{Bottom}) \wedge$
     $\forall s : \texttt{Stack} \,.\, P(s) \supset \forall i : \texttt{INTEGER} \,.\, P(\texttt{Push}(i, s))]$
     $\supset \forall s : \texttt{Stack} \,.\, P(s)$

Example: MIS for Stacks ADT 3

4. **Top applied to a Push stack.**
   $\forall i : \texttt{INTEGER}, s : \texttt{Stack} \,.\, \texttt{Top}(\texttt{Push}(i, s)) = i$

5. **Pop applied to a Push stack.**
   $\forall i : \texttt{INTEGER}, s : \texttt{Stack} \,.\, \texttt{Pop}(\texttt{Push}(i, s)) = s$

6. **Bottom has no top: EmptyStack exception.**
   $\texttt{Top}(\texttt{Bottom}) \uparrow$

7. **Bottom has no pop: EmptyStack exception.**
   $\texttt{Pop}(\texttt{Bottom}) \uparrow$

Note: This MIS has the form of an **axiomatic theory** (L, Γ) where

- L is the **language** defined by the interface of the MID
- Γ is the set of axioms of the MID

Example: Stacks ADT Module 1

```
MODULE Stacks;

IMPORT Out;


(* INTERFACE *)

(*
TYPE Stack;
CONST Bottom: Stack;
PROCEDURE Push(i: INTEGER, s: Stack): Stack;
PROCEDURE Top(s: Stack): INTEGER;
PROCEDURE Pop(s: Stack): Stack;
*)


(* IMPLEMENTATION *)


(* Types *)

TYPE

  Stack* = POINTER TO StackRec;
```


Example: Stacks ADT Module 2

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


(* Constants *)

CONST Bottom* = NIL;              (* represents the empty stack *)


(* Exceptions: *)

PROCEDURE EmptyStackException();
  BEGIN
    Out.String("Stacks.EmptyStackException: The stack is empty.");
    HALT(33);  (* Abort program *)
  END EmptyStackException;
```


Example: Stacks ADT Module 3

```
(* Local functions *)

PROCEDURE Empty(s: Stack): BOOLEAN;
  BEGIN
    RETURN s = Bottom;
  END Empty;


(* Interface functions *)

PROCEDURE Push*(i: INTEGER; s: Stack): Stack;
  VAR t: Stack;
  BEGIN
    NEW(t);
    t^.item := i;
    t^.rest := s;
    RETURN t;
  END Push;
```

Example: Stacks ADT Module 4

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

PROCEDURE Pop*(s: Stack): Stack;
  BEGIN
    IF ~Empty(s) THEN
      RETURN s^.rest;
    ELSE
      EmptyStackException();
    END;
  END Pop;


END Stacks.
```


Example: MIS for Lists ADT 1

- Imported modules: none
- Interface:

```
TYPE List;
CONST Nil: List;
PROCEDURE Cons(i: INTEGER; k: List): List;
PROCEDURE Member(i: INTEGER, k: List): INTEGER;
PROCEDURE Take(i: INTEGER, k: List): List;
PROCEDURE Drop(i: INTEGER, k: List): List;
```

- Exceptions:
  - BadIndex : INTEGER $\times$ List $\rightarrow *$

Example: MIS for Lists ADT 2

- Input/output specification (axioms):

1. **Nil is not a Cons list.**
   $\forall i$: INTEGER, k : List . $\texttt{Nil} \neq \texttt{Cons}(i, k)$

2. **Cons is one-to-one.**
   $\forall i_1, i_2$: INTEGER, k_1, k_2 : List .
   $\texttt{Cons}(i_1, k_1) = \texttt{Cons}(i_2, k_2) \supset (i_1 = i_2 \wedge k_1 = k_2)$

3. **Induction axiom for lists.**
   $\forall P$: List $\rightarrow$ BOOLEAN .
   $[P(\texttt{Nil}) \wedge$
   $\forall k$: List . $P(k) \supset \forall i$: INTEGER . $P(\texttt{Cons}(i, k))]$
   $\supset \forall k$: List . $P(k)$

4. **Membership with respect to Nil.**
   $\forall i$: INTEGER . $\texttt{Member}(i, \texttt{Nil})\uparrow$

Example: MIS for Lists ADT 3

5. **Membership with respect to Cons.**
   $\forall i, j$: INTEGER, k : List .
   $\texttt{Member}(i, \texttt{Cons}(j, k)) \simeq$
   $\mathsf{if}(i < 0, \bot, \mathsf{if}(i = 0, j, \texttt{Member}(i - 1, k)))$

6. **Membership with respect to Take.**
   $\forall i, j$: INTEGER, k : List .
   $\texttt{Member}(i, \texttt{Take}(j, k)) \simeq$
   $\mathsf{if}(i < 0, \bot, \mathsf{if}(i < j, \texttt{Member}(i, k), \bot))$

7. **Membership with respect to Drop.**
   $\forall i, j$: INTEGER, k : List .
   $\texttt{Member}(i, \texttt{Drop}(j, k)) \simeq$
   $\mathsf{if}(i < 0, \bot, \texttt{Member}(j + i, k))$

Components of a Before/After MIS

1. **Imported modules**
2. **Interface**
   - Types
   - Constant names and types
   - Procedure names and types
3. **Exceptions**
4. **State constants** (with value conditions)
5. **State variables** (with initial values)
6. **Behavior rules**
   - Output rules
   - State transition rules
   - Exception rules

Example: Stack MIS 1

- Imported modules: none
- Interface:

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

- State constants:

  max : `INTEGER` $(0 \leq \text{max})$

- State variables:

  s : lists[`INTEGER`] (initially $s = \text{nil}$)

Example: Stack MIS 2

- Exceptions:

  EmptyStack : `BOOLEAN`

  FullStack : `BOOLEAN`

- Behavior rules:

`Reset`

Input	Output	Transition	Exception
		$s' = \text{nil}$	

`MaxHeight`

Input	Output	Transition	Exception
	max		

Example: Stack MIS 3

`Height`

Input	Output	Transition	Exception
	$\|s\|$		

`Empty`

Input	Output	State	Exception
	$\|s\| = 0$		

`Full`

Input	Output	State	Exception
	$\|s\| = \text{max}$		

Example: Stack MIS 4

`Push`

Input	Output	Transition	Exception
i : `INTEGER`		$s' = \mathsf{cons}(i, s)$	`Full()` $\supset$ FullStack

`Pop`

Input	Output	Transition	Exception
		$s' = \mathsf{tl}(s)$	`Empty()` $\supset$ EmptyStack

`Top`

Input	Output	Transition	Exception
	$\mathsf{hd}(s)$		`Empty()` $\supset$ EmptyStack

Module Structure

- Simple structure:
  - All module interfaces are accessible to all module implementations
  - All modules are indivisible units
- Access structure: Module interfaces are only available to certain module implementations
- Submodule structure: Modules may be decomposed of into submodules
  - Example: Modules may contain local modules

References

1. D. Parnas, "On the criteria to be used in decomposing systems into modules", in: D. Hoffman and D. Weiss, *Software Fundamentals*, Addison Wesley, 2001.

2. D. Parnas, P. Clements, and D. Weiss, "The modular structure of complex systems", in: D. Hoffman and D. Weiss, *Software Fundamentals*, Addison Wesley, 2001.