Design and Software Architecture

Modularization Techniques

Miaoqing Huang
University of Arkansas
Spring 2010
Outline

1. Interface
2. Design Notations
3. Categories of Modules
4. Techniques for Design for Changes
Outline

1 Interface
2 Design Notations
3 Categories of Modules
4 Techniques for Design for Changes
The interface

- Should reveal as little information of the implementation as possible
- Should provide sufficient information for other modules to use the services
To understand the nature of USES, we need to know what a used module exports through its interface.

The client imports the resources that are exported by its servers.

- Modules implement the exported resources.
- Implementation is hidden to clients.

A clear distinction between interface and implementation is a key design principle.

Supports separation of concerns:
- Clients care about resources exported from servers.
- Servers care about implementation.

Interface acts as a contract between a module and its clients.
Information Hiding

- Basis for design (i.e., module decomposition)
- Implementation secrets are hidden to clients
  - They can be changed freely if the change does not affect the interface
- Try to encapsulate changeable design decisions as implementation secrets within module implementations
  - Protecting other parts of the system from extensive modification if the design decision is changed
- A common use of information hiding is to hide the physical storage layout for data
Imagine you ask someone from a different experiment what the momentum distribution of neutral particles because you want to compare their findings with what you have got. What can happen is

- Some nice guy already did that analysis, gives you a table, and you are done
- Most information is hidden, only the result is given
- Somebody gives you a number of convoluted tables, numbers, and instructions, and says “now you can easily calculate the distribution... oh, one more thing, don’t forget the correction for multiplicity...”
- Too much information is given, which is a headache
Information Hiding
Another Example

```cpp
class cylinder {
    protected:
    float radius;
    float height;

    public:
    float GetVolume();
    float GetSurface();

    cylinder (float, float);
};

float cylinder::GetVolume() {
    return ( PI*radius*radius * height );
}

float cylinder::GetSurface() {
    return ( 2*PI*radius*height + 2*PI*radius*radius );
}

cylinder::cylinder(float r, float h){
    radius = r;
    height = h;
}

#include <iostream.h>
main()
{
    float volume;
    cylinder c( 2.0, 1.0 );
    volume = c.GetVolume();
    cout << "volume is " << volume << endl;
}
```

Here we construct a new cylinder with radius 2 and height 1.

Then we ask the cylinder to please calculate its volume and return it to us.
We introduce a SYMBOL_TABLE module
  - provides operations to
    - CREATE an entry for a new variable
    - GET the value associated with a variable
    - PUT a new value for a given variable

The module hides the internal data structure of the symbol table
  - the data structure may freely change without affecting clients

Interface should not reveal what we expect may change later
  - Use an additional parameter, POS, to indicate the success of the function call
    - Valid pointer: the variable exists
    - Null pointer: the variable does not exist

<table>
<thead>
<tr>
<th>a</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>200</td>
</tr>
<tr>
<td>c</td>
<td>10</td>
</tr>
<tr>
<td>x</td>
<td>-5</td>
</tr>
<tr>
<td>y</td>
<td>13</td>
</tr>
</tbody>
</table>
Outline

1. Interface
2. Design Notations
3. Categories of Modules
4. Techniques for Design for Changes
Design Notations

- Notations allow designs to be described precisely
- They can be textual or graphic
  - TDN (Textual Design Notation)
  - GDN (Graphical Design Notation)
- TDN and GDN
  - Illustrate how a notation may help in documenting design
  - Illustrate what a generic notation may look like
  - Are representative of many proposed notations
module X
uses Y, Z
exports var A: integer;
  type B: array (1..10) of real;
procedure C (D: in out B; E: in integer; F: in real);
Here is an optional natural-language description of what A, B, and C actually are, along with possible constraints or properties that clients need to know; for example, we might specify that objects of type B sent to procedure C should be initialized by client and should never contain all zeros.

implementation
  If needed, here are general comments about the rationale of the modularization, hints on the implementations, etc.
  is composed of R, T
end X
TDN Example

module R
uses Y
exports var K: record ... end;
  type B: array (1..10) of real;
  procedure C (D: in out B; E: in integer; F: in real);
implementation
  .
  .
end R

module T
uses Y, Z, R
exports var A: integer;
implementation
  .
  .
end T
More on TDN

Comments in TDN
- May be used to specify the protocol to be followed by the clients so that exported services are correctly provided
  - e.g., a certain operation that does the initialization of the module should be called before any other operation
  - e.g., an insert operation cannot be called if the table is full

Benefits
- Notation helps describe a design precisely
- Design can be assessed for consistency
  - Having defined module X, modules R and T must be defined eventually
  - R, T replace X

One more example, compiler of MINI language, on the textbook
GDN description of module X

Module X

Module R
Module T

Module Y

Module Z

A
B
C
Decomposition of module X

Module X

Module R

Module T

Module Y

Module Z

K

B

C

A
Outline

1. Interface
2. Design Notations
3. Categories of Modules
4. Techniques for Design for Changes
Categories of modules

**Functional modules**
- Traditional form of modularization
- Provide a procedural abstraction
- Encapsulate an algorithm
  - e.g., sorting module, fast Fourier transform module, ...

**Libraries**
- A group of related procedural abstractions
  - e.g., mathematical libraries

**Common pools of data**
- Data shared by different modules
  - e.g., configuration constants
Categories of modules

Abstract Objects

- Objects manipulated via interface functions
- Data structure hidden to clients
- Modules with state
Categories of modules
Abstract Objects – Example

Evaluate arithmetic expression in Polish postfix form

```
module STACK
exports
  procedure PUSH (VAL: in integer);
  procedure POP_2(VAL1, VAL2: out integer);
end STACK
```

Expression: a b c + *
Categories of modules
Abstract Objects – Example

Evaluate arithmetic expression in Polish postfix form

module STACK
exports
    procedure PUSH (VAL: in integer);
    procedure POP_2(VAL1, VAL2: out integer);
end STACK
Categories of modules
Abstract Objects – Example

Evaluate arithmetic expression in Polish postfix form

module STACK
exports
procedure PUSH (VAL: in integer);
procedure POP_2(VAL1, VAL2: out integer);
end STACK

Evaluate arithmetic expression in Polish postfix form

module STACK
exports
procedure PUSH (VAL: in integer);
procedure POP_2(VAL1, VAL2: out integer);
end STACK
Categories of modules

Abstract Objects – Example

Evaluate arithmetic expression in Polish postfix form

```pascal
module STACK
exports
  procedure PUSH (VAL: in integer);
  procedure POP_2(VA1, VA2: out integer);
end STACK
```
Categories of modules
Abstract Objects – Example

Evaluate arithmetic expression in Polish postfix form

module STACK
exports
  procedure PUSH (VAL: in integer);
  procedure POP_2 (VAL1, VAL2: out integer);
end STACK
Categories of modules

Abstract Objects – Example

Evaluate arithmetic expression in Polish postfix form

```
module STACK
exports
    procedure PUSH (VAL: in integer);
    procedure POP_2(VAL1, VAL2: out integer);
end STACK
```
Evaluate arithmetic expression in Polish postfix form

module STACK
events
    procedure PUSH (VAL: in integer);
    procedure POP_2(VAL1, VAL2: out integer);
end STACK
Evaluate arithmetic expression in Polish postfix form

module STACK
exports
  procedure PUSH (VAL: in integer);
  procedure POP_2 (VAL1, VAL2: out integer);
end STACK
Evaluate arithmetic expression in Polish postfix form

```pascal
module STACK
exports
  procedure PUSH (VAL: in integer);
  procedure POP_2 (VAL1, VAL2: out integer);
end STACK
```

```
b + c
```

```
a
```
Categories of modules
Abstract Objects – Example

Evaluate arithmetic expression in Polish postfix form

```
module STACK
exports
  procedure PUSH (VAL: in integer);
  procedure POP_2 (VAL1, VAL2: out integer);
end STACK
```
Evaluate arithmetic expression in Polish postfix form

```plaintext
module STACK
exports
    procedure PUSH (VAL: in integer);
    procedure POP_2 (VAL1, VAL2: out integer);
end STACK
```

Expression: `a*(b+c)`
Evaluate arithmetic expression in Polish postfix form

```
module STACK
exports
    procedure PUSH (VAL: in integer);
    procedure POP_2 (VAL1, VAL2: out integer);
end STACK
```

Expression: \[ a \times (b + c) \]
Categories of modules

Abstract data types

- An abstract data-type module is a module that exports a type, along with the operations needed to access and manipulate objects of that type
  - class in Java and C++
  - Instances of an abstract data type are abstract objects

```plaintext
module STACK_HANDLER
exports
type STACK = ?;
This is an abstract data-type module; the data structure is a secret hidden in the implementation part.
procedure PUSH (S: in out STACK; VAL: in integer);
procedure POP (S: in out STACK; VAL: out integer);
function EMPTY (S: in STACK) : BOOLEAN;
.
.
end STACK_HANDLER
```
Specific techniques for design for change

- Use of configuration constants
  - Using symbolic constants is a common implementation practice
    - e.g., `#define` in C language

```c
int a[10];
if ((k>=0) && (k<=9)) {
    perform indexing;
}
else {
    do other action;
}
```
Specific techniques for design for change

- Use of configuration constants
  - Using symbolic constants is a common implementation practice
  - e.g., \#define in C language

```c
#define SIZE 10
int a[SIZE];
if ((k>=0) && (k<=SIZE-1)) {
    perform indexing;
}
else {
    do other action;
}
```
Specific techniques for design for change

- Conditional compilation
  - ...source fragment common to all versions...

```c
//use C preprocessor
#ifdef hardware-1
  ...source fragment for hardware 1 ...
#endif

#ifdef hardware-2
  ...source fragment for hardware 2 ...
#endif
```
Summary

- Interface
  - Reveal as little information of implementation as possible; however, enough to use the services provided to other modules
  - Information hiding

- Design notations
  - Formally and precisely describe a design
  - TDN and GDN

- Categories of modules
  - Five different types

- Particular techniques for design for change
  - Configuration constants
  - Conditional compilation