Information Hiding

Daniel M. Berry

Information Hiding -1

The concept of information hiding (IH) comes from the seminal paper,

"On the criteria to be used in decomposing systems into modules", *CACM*, Dec., 1972

by David L. Parnas.

The purpose of information hiding is to obtain a modularization of the code of a system that isolates changes into single modules.

Information Hiding -2

Information hiding is a way to use abstract data types and abstract objects, such as provided by

- **Ada packages**
- **Simula classes**
- **C++ classes**

I assume that you know at least one of these!

If you don't, then go learn!

ADT&O -1

It is necessary to teach exploitation of abstract data types and objects (ADT&O).

I have found many people writing FORTRAN code in Ada syntax or C code in C++ syntax.

On the other hand, it is possible to exploit ADT&O even in assembly language and FORTRAN!

ADT&O -2

What's a nice programming topic like you doing in a real rough requirements engineering course like this?

- **ADT&O is a good way to organize a domain model**
- **ADT&O is a good way to organize a software design**

ADT&O -3

Remember that you may have to do some design to discover all requirements.

With ADT&O, the domain model and design can be built without exposing implementation details!

ADT&O -4

In my experience, if I program with ADT&O using information hiding, my modules end up being the types and objects of the domain model.

So even if I am thinking implementation, I end up with a domain model!

ADT&O and IH -1

For those of you who are implementation-bound in your thinking, ADT&O with information hiding frees you from thinking implementation.

ADT&O with IH gives you a way of thinking at an abstract level without having to worry about whether you can implement, because you can!

ADT&O and IH -2

You will see an entirely different way of thinking about problems, in which you work with problem-level concepts rather than more traditional data- or control-flow concepts.

If you’re already thinking this way, bravo!

If not, then learn!

Parnas’s Sample Problem

Requirements:

The KWIC index system accepts an ordered set [sic] of lines, each line is an ordered set [sic] of words, and each word is an ordered set [sic] of characters.

Any line may be “circularly shifted” by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all the circular shifts of all lines in alphabetical order.

List of Six Lines

Make - A Program for Maintaining Computer Programs
Reference Manual for the Ada Programming Language
The NYU Ada/Ed System, An Overview
Program Design Language
Software Development Processor User Reference Manual
The Ina Jo Reference Manual

The next slide shows the KWIC (KeyWord In Context) index of the above lines.

Make	- A Program for Maintaining Computer Programs	[Fel78]
Make -	A Program for Maintaining Computer Programs	[Fel78]
Reference Manual for the	Ada Programming Language	[ADA81]
The NYU	Ada/Ed System, An Overview	[NYU81]
The NYU Ada/Ed System,	An Overview	[NYU81]
Make - A Program for Maintaining	Computer Programs	[Fel78]
Program	Design Language	[CFG75]
Software	Development Processor User Reference Manual	[Yav80]
Make - A Program	for Maintaining Computer Programs	[Fel78]
Reference Manual	for the Ada Programming Language	[ADA81]
The	Ina Jo Reference Manual	[LSSE80]
The Ina	Jo Reference Manual	[LSSE80]
Program Design	Language	[CFG75]
Reference Manual for the Ada Programming	Language	[ADA81]
Make - A Program for	Maintaining Computer Programs	[Fel78]
	Make - A Program for Maintaining Computer Programs	[Fel78]
Reference	Manual for the Ada Programming Language	[ADA81]
Software Development Processor User Reference	Manual	[Yav80]
The Ina Jo Reference	Manual	[LSSE80]
The	NYU Ada/Ed System, An Overview	[NYU81]
The NYU Ada/Ed System, An	Overview	[NYU81]
Software Development	Processor User Reference Manual	[Yav80]
	Program Design Language	[CFG75]
Make - A	Program for Maintaining Computer Programs	[Fel78]
Reference Manual for the Ada	Programming Language	[ADA81]
Make - A Program for Maintaining Computer	Programs	[Fel78]
	Reference Manual for the Ada Programming Language	[ADA81]
Software Development Processor User	Reference Manual	[Yav80]
The Ina Jo	Reference Manual	[LSSE80]
	Software Development Processor User Reference Manual	[Yav80]
The NYU Ada/Ed	System, An Overview	[NYU81]
Reference Manual for	the Ada Programming Language	[ADA81]
	The Ina Jo Reference Manual	[LSSE80]
	The NYU Ada/Ed System, An Overview	[NYU81]
Software Development Processor	User Reference Manual	[Yav80]

KWIC Index of Above Lines

First 5 lines of Index:

Make	**- A Program for Maintaining Computer Programs**	**[Fel78]**
Make -	***A* Program for Maintaining Computer Programs**	**[Fel78]**
Reference Manual for the	***A*da Programming language**	**[Ada81]**
The NYU	***A*da/ed System, An Overview**	**[NYU81]**
The NYU Ada/ed System,	***A*n Overview**	**[NYU81]**

Two Modularizations

Parnas shows first a conventional modularization and then one that does a better job of being modifiable.

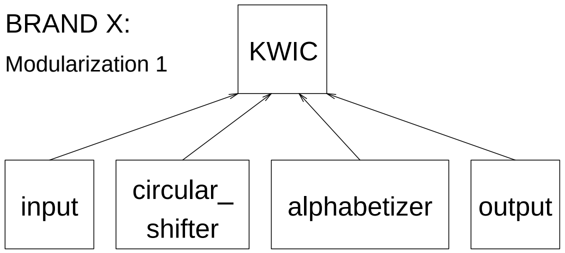
Modularization 1: Brand X

Modularization 2: Brand DLP

Guess which one is supposed to be better!

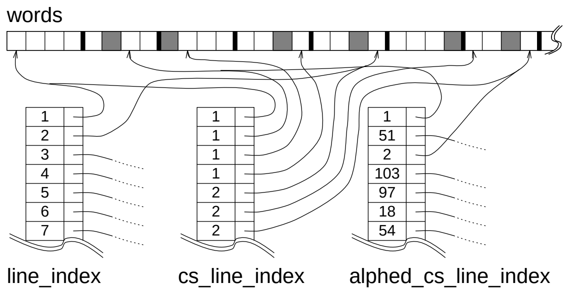
The textual module descriptions are from Parnas himself.

Modularization 1 -1



Modularization 1 -2

DATA STRUCTURE DIAGRAM



Modularization 1 -3

Module 1: Input. ***This module reads the data lines from the input medium and stores them in core for processing by the remaining modules. The characters are packed four to a word, and an otherwise unused character is used to indicate the end of a word. An index is kept to show the start of each line.***

Modularization 1 -4

Module 2: Circular Shift. ***This module is called after the input module has completed its work. It prepares an index which gives the address of the first character of each circular shift, and the original index of the line in the array made up by module 1. It leaves its output in core with words in pairs (original line number, starting address).***

Modularization 1 -5

Module 3: Alphabetizing. ***This module takes as input the arrays produced by modules 1 and 2. It produces an array in the same format as that produced by module 2. In this case, however, the circular shifts are listed in another order (alphabetically).***

Modularization 1 -6

Module 4: Output. ***Using the arrays produced by module 3 and module 1, this module produces a nicely formatted output listing of all of the circular shifts. In a sophisticated system the actual start of each line will be marked, pointers to further information may be inserted, and the start of the circular shift may actually not be the first word in the line, etc.***

Modularization 1 -7

Module 5: Master Control. ***This module does little more than control the sequencing among the other four modules. It may also handle error messages, space allocation, etc.***

Modularization 1 -8

Parnas says:

It should be clear that the above does not constitute a definitive document. Much more information would have to be supplied before work could start. The defining documents would include a number of pictures [as above] showing core formats, pointer conventions, calling conventions, etc. All of the interfaces between the four modules must be specified before work could begin.

Modularization 1 -9

I add the additional defining documents using Ada notation:

```
procedure KWIC is
    type PAIR is record
        line_no:INTEGER;
        character_no:INTEGER;
    end record;
    type INDEX_TABLE is array
        (INTEGER range <>) of PAIR;
    large_no: constant INTEGER:=MAX_INT;
```


Modularization 1 -10

```
    words:STRING(1..large_no);
    line_index:INDEX_TABLE(1..large_no);
    cs_line_index:INDEX_TABLE(1..large_no);
    alphed_cs_line_index:INDEX_TABLE(1..large_no);

    procedure input is separate;
    procedure make_circular_shifts is separate;
    procedure alphabetize is separate;
    procedure output is separate;
```

Modularization 1 -11

```
begin
    input;
    make_circular_shifts;
    alphabetize;
    output;
end KWIC;
```


Modularization 1 -12

```
separate(KWIC)
procedure input is
begin
    null;
end;

separate(KWIC)
procedure make_circular_shifts is
begin
    null;
end;
```


Modularization 1 -13

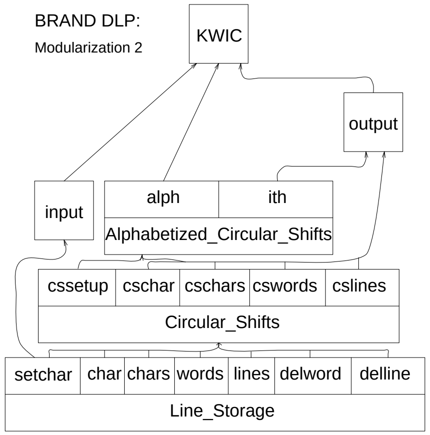
```
separate(KWIC)
procedure alphabetize is
begin
    null;
end;

separate(KWIC)
procedure output is
begin
    null;
end;
```


Modularization 1 -14

More from Parnas:

This is a modularization in the sense meant by all proponents of modular programming [circa 1972]. The system is divided into a number of modules with well-defined interfaces; each one is small enough and simple enough to be thoroughly understood and well programmed. Experiments on a small scale indicate that this is approximately the decomposition which would be proposed by most programmers for the task specified.

Modularization 2 -1



Modularization 2 -2

Module 1: Line Storage. This module consists of a number of functions or subroutines which provide the means by which the user of the module may call on it. The function call **CHAR(r,w,c)** ***will have as value an integer representing the*** **c** ***th character in the*** **r** ***th line,*** **w** ***th word. A call such as*** **SETCHAR(r,w,c,d)** ***will cause the*** **c** ***th character in the*** **w** ***th word of the*** **r** ***th line to be the character represented by*** **d** ***(i.e.,*** **CHAR(r,w,c) = d*****).***

Modularization 2 -3

WORDS(r) ***returns as value the number of words in line*** **r*****. There are certain restrictions in the way that these routines may be called; if these restrictions are violated the routines "trap" to an error-handling subroutine which is to be provided by the users of the routine. Additional routines are available which reveal to the caller the number of words in any line, the number of lines currently stored, and the number of characters in any word....***

Modularization 2 -4

Module 2: Input. This module reads the original lines from the input media and calls the line storage module to have them stored internally.

Modularization 2 -5

***Module 3: Circular Shifter.** The principal functions provided by this module are analogs of functions provided in module 1. The module creates the impression that we have created a line holder containing not all of the lines but all of the circular shifts of the lines. Thus the function call* **CSCHAR(l,w,c)** *provides the value representing the* **c** *th character in the* **w** *th word of the* **l** *th circular shift.*

Modularization 2 -6

It is specified that (1) if **i < j** *then the shifts of line* **i** *precede the shifts of line* **j**, *and (2) for each line the first shift is the original line, the second shift is obtained by making a one-word rotation to the first shift, etc. A function* **CSSETUP** *is provided which must be called before the other functions have their specified value....*

Modularization 2 -7

***Module 4: Alphabetizer.** This module consists principally of two functions. One,* **ALPH**, *must be called before the other will have a defined value. The second,* **ITH**, *will serve as an index.* **ITH(i)** *will give the index of the circular shift that comes* **i** *th in the alphabetical ordering.*

Modularization 2 -8

***Module 5: Output.** This module will give the desired printing of set [sic] of lines or [sic] circular shifts.*

***Module 6: Master Control.** Similar in function to the modularization above.*

Modularization 2 -9

References 3 and 8 of Parnas's paper give formal definitions of Modules 1, 3, and 4 (the interesting ones!!).

We give Ada module structure to which these definitions can be attached.

Modularization 2 -10

```
package LINE_STORAGE is
    function char (l,w,c:INTEGER)
        return CHARACTER;
    procedure setchar (l,w,c:INTEGER;
        d:CHARACTER);
    function chars (l,w:INTEGER)
        return INTEGER;
    function words (l:INTEGER)
        return INTEGER;
    function lines return INTEGER;
```


Modularization 2 -11

```
    ...
    -- error handling exceptions
end LINE_STORAGE;
```


Modularization 2 -12

```
with LINE_STORAGE;
use LINE_STORAGE;
package CIRCULAR_SHIFTS is
    procedure cssetup;
    function cschar (l,w,c:INTEGER)
        return CHARACTER;
    function cschars (l,w:INTEGER)
        return INTEGER;
    function cswords (l:INTEGER)
        return INTEGER;
    function cslines return INTEGER;
```

Modularization 2 -13

```
     -- error handling exceptions
end CIRCULAR_SHIFTS;
```


Modularization 2 -14

```
with CIRCULAR_SHIFTS;
use CIRCULAR_SHIFTS;
package ALPHABETIZED_CIRCULAR_SHIFTS is
     procedure alph;
     function ith(i:INTEGER)
          return INTEGER;
     -- error handling exceptions
end ALPHABETIZED_CIRCULAR_SHIFTS;
```


Modularization 2 -15

```
with ALPHABETIZED_CIRCULAR_SHIFTS;
use ALPHABETIZED_CIRCULAR_SHIFTS;
procedure KWIC is
     procedure input is separate;
     procedure output is separate;
begin
     input;
     alph;
     output;
end KWIC;
```


Modularization 2 -16

```
with LINE_STORAGE; use LINE_STORAGE;
separate(KWIC);
procedure input is
     l,w,c:INTEGER;
     d:CHARACTER;
begin
     -- read in characters one by one into d, breaking into
     -- words and lines, setting the line, word, and
     -- character indices, l, w, and c, and doing for each,
     setchar(l,w,c,d);
end input;
```

Modularization 2 -17

```
with ALPHABETIZED_CIRCULAR_SHIFTS,
    CIRCULAR_SHIFTS,TEXT_IO;
use ALPHABETIZED_CIRCULAR_SHIFTS,
    CIRCULAR_SHIFTS,TEXT_IO;
separate(KWIC);
procedure output is
    l:INTEGER;
begin
    for i in 1..cslines() loop
        l:=ith(i);
```


Modularization 2 -18

```
            for w in 1..cswords(l) loop
                for c in 1..cschars(l,w) loop
                    -- in the proper
                    -- place for the
                    -- fancy output do
                    put(cschar(l,w,c));
                end loop;
            end loop;
        end loop;
end output;
```


Parnas's Comparison -1

Parnas says about the two modularizations:

General.* *Both schemes will work. The first is quite conventional, and the second has been used successfully in a class project [* each student programmed a different module and complete programs were built in all possible combinations*]. Both will reduce the programming to the relatively independent programming of a number of small, manageable, programs.

Parnas's Comparison -2

Note first that the two decompositions may share all data representations and access methods. Our discussion is about two different ways of cutting up what* may *be the same object. A system built according to decomposition 1 could conceivably be identical* after assembly *to one built according to decomposition 2.

Parnas's Comparison -3

We now use Ada facilities to cause the code of the two modularizations to be (almost) identical

I now provide package bodies for the packages of the Ada rendition of Modularization 2 using code from the Ada rendition of Modularization 1!

Parnas's Comparison -4

```
package body LINE_STORAGE is
    type PAIR is record
        line_no:INTEGER;
        character_no:INTEGER;
    end record;
    type INDEX_TABLE is array
        (INTEGER range <>) of PAIR;
    large_no: constant INTEGER:=MAX_INT;

    the_words:STRING(1..large_no);
    line_index:INDEX_TABLE(1..large_no);
```


Parnas's Comparison -5

```
    function char (l,w,c:INTEGER)
        return CHARACTER is ...
    procedure setchar (l,w,c:INTEGER;
        d:CHARACTER) is ...
    function chars (l,w:INTEGER)
        return INTEGER is ...
    function words (l:INTEGER)
        return INTEGER is ...
    function lines return INTEGER is ...
    ...
```


Parnas's Comparison -6

```
☛   pragma IN_LINE(char,setchar);
    -- error handling exceptions
end LINE_STORAGE;
```

Parnas's Comparison -7

```
package body CIRCULAR_SHIFTS is
    type PAIR is record
        line_no:INTEGER;
        character_no:INTEGER;
    end record;
    type INDEX_TABLE is array
        (INTEGER range <>) of PAIR;
    large_no: constant INTEGER:=MAX_INT;

    cs_line_index:INDEX_TABLE(1..large_no);
```


Parnas's Comparison -8

```
procedure cssetup is ...
function cschar (l,w,c:INTEGER)
    return CHARACTER is ...
function cschars (l,w:INTEGER)
    return INTEGER is ...
function cswords (l:INTEGER)
    return INTEGER is ...
function cslines return INTEGER is ...
```


Parnas's Comparison -9

```
☞   pragma IN_LINE(cssetup,cschar);
    -- error handling exceptions
end CIRCULAR_SHIFTS;
```


Parnas's Comparison -10

```
package body ALPHABETIZED_CIRCULAR_SHIFTS is
    type PAIR is record
        line_no:INTEGER;
        character_no:INTEGER;
    end record;
    type INDEX_TABLE is array
        (INTEGER range <>) of PAIR;
    large_no: constant INTEGER:=MAX_INT;

    alphed_cs_line_index:INDEX_TABLE(1..large_no);
```

Parnas's Comparison -11

```
procedure alph is ...
function ith(i:INTEGER)
    return INTEGER is ...
pragma IN_LINE(ith);
-- error handling exceptions
```


Parnas's Comparison -12

The procedures that were left out-of-line are those that were deemed likely to be out-of-line in Modularization 1.

In the resulting object code, all the data structures end up in the same block, just as in Modularization 1.

***However*, each data structure is visible only to the procedures of the module from which it came!**

Parnas's Comparison -13

Parnas says:

The differences between the two alternatives are in the way they are divided into work assignments, and the interfaces between modules. The algorithms used in both cases might be identical. The systems are substantially different even if identical in the runnable representation.

Parnas's Comparison -14

This is possible because the runnable representation need only be used for running; other representations are used for changing, documenting, understanding, etc. The two systems will not be identical in those other representations.

Parnas's Comparison -15

***changeability.** There are a number of design* **[i.e., implementation]** *decisions which are questionable and likely to change under many circumstances. This is a partial list*

1. *Input format.*
2. *The decision to have all lines stored in core. For large jobs it may prove inconvenient or impractical to keep all of the lines in core at any one time.*

Parnas's Comparison -16

3. *The decision to pack the characters four to a word. In cases where we are working with small amounts of data it may prove undesirable to pack the characters; time will be saved by a character per word layout. In other cases, we may pack, but in different formats.*

Parnas's Comparison -17

4. *The decision to make an index for the circular shifts rather that [sic] store them as such. Again, for a small index or a large core, writing them out may be the preferable approach. Alternatively, we may choose to prepare nothing during* CSSETUP. *All computations could be done during the calls on the other functions such as* CSCHAR.

Parnas's Comparison -18

I add to this:

***CSSETUP* is provided specifically to allow many implementations.**

- **if the implementation needs some set up, then *CSSETUP* is non-empty**
- **If the implementation needs no set up, then *CSSETUP* is empty; if in addition, *CSSETUP* is inline, then it costs nothing!**

Parnas's Comparison -19

Getting back to Parnas's comments:

5. ***The decision to alphabetize the list once, rather than either (a) search for each item when needed, or (b) partially alphabetize as is done in Hoare's FIND.... In a number of circumstances it would be advantageous to distribute the computation involved in alphabetization over the time required to produce the index.***

Parnas's Comparison -20

Let us now examine each change and see which modules in each modularization need to be modified to effect the change.

Parnas's Comparison -21

Change	Modularization 1	Modularization 2
Input	Input	Input
Lines in core	*all*	Lines
Packing	*all*	Lines
Store CSs	CS Alph Output	CS
Alph on the fly	Alph Output	Alph

Parnas's Comparison -22

In Modularization 1, most changes affected all modules!

In Modularization 2, each change affected only *one* module!

Fantastic!!!!!

Kinds of Changes -1

Note though, that we have been talking about implementation changes *only*.

Functionality changes are an entirely different kind of animal.

Kinds of Changes -2

For functionality changes, we must expect to have new modules and new procedures added to existing modules.

However, my experience is that a good modularization, that hides implementation changes, shields very well against many functionality changes.

Such changes go much easier with a good decomposition than without.

How to Decompose Well -1

We consider several methods to decompose systems into modules whose internals are easily modified independently.

All of these methods yield nearly the same decomposition.

How to Decompose Well -2

The methods are

- **Parnas's method**
- **Nouns as module names**
- **Myers's criteria**
- **Britton & Parnas's ideas**
- **Booch's ideas**
- **other ideas**

Parnas's Method

Come up with a first cut decomposition.

Get a list of as many changes as you can think of; ... blue sky!

If the effect of each change is isolated to one module, you have a good decomposition.

If not, go back to the drawing board.

Nouns as Module Names

In my experience, the general nouns of the problem statement make good ADTs and these ADTs make good modules.

In fact, Abbott, Booch, and Berzins & Luqi suggest starting with an informal strategy and its identified nouns.

This method was suggested in the earlier lecture titled "Survey of Methods and Notations" as a way to get a domain model!

Myers's Criteria -1

Myers talks about cohesion and coupling.

Cohesion is the degree to which a module is doing one and only one thing.

Coupling is the degree to which two modules communicate.

Myers's Criteria -2

We want

high cohesion

and

low coupling.

Myers's Criteria -3

Cohesion:

The procedures

print page footer
goto next page
print page header

are each more cohesive than the procedure

print page footer and the next page header

which does the work of all three in that order.

Myers's Criteria -4

A single procedure or function is more cohesive as a module than an abstract data type.

Myers's Criteria -5

Coupling:

- **no communication at all is less coupling than parameter passing**
- **parameter passing is less coupling than common usage of global variables**
- **communicating with individual items is less coupling than communicating with data structures**

Myers's Criteria -6

So why are ADTs good modules despite the fact that an ADT is less cohesive than its individual operations?

An ADT groups together into one module all routines that must share access to a collection of data structures, so that nothing outside of the module needs to see these data structures.

And ...

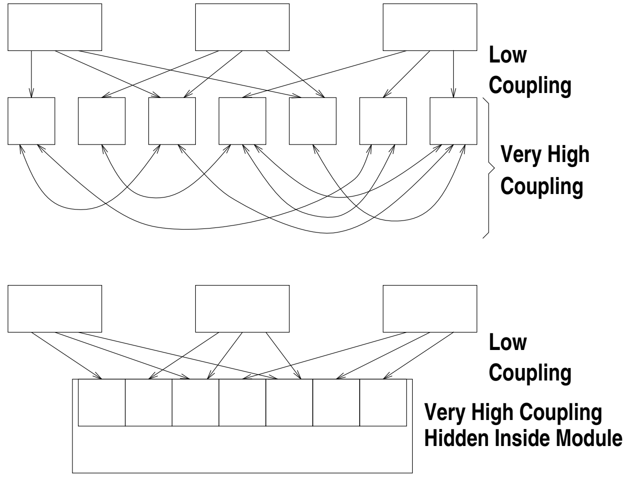
Myers's Criteria -7

An ADT causes outsiders to see these data structures as single indivisible items rather than as their implementing data structures.

An ADT forces outsiders to pass these data items as parameters to the operations instead of being able to directly access the components of these items as shared global data structures.

Myers's Criteria -8

That is, we suffer a slight reduction in cohesiveness to obtain big reduction in coupling.



Myers's Criteria -9

There is another way to explain this; put all data structures and procedures that have to be modified together, and *no more*, into the same module.

Thus, the data structures that implement a pushdown stack and the operation bodies that operate on these structures should be in the same module.

Myers's Criteria -10

However, ...

For sure, do not put into this module the functions that implement a stack-oriented, postfix-polish operator pocket calculator.

Myers's Criteria -11

Possibly, do not put into this module procedures such as

```
procedure ptop(s:in out INTSTACKVAL;
      t: out INTEGER) is
begin
      t:=top(s);
      pop(s);
end;
```

which can be defined using exported operations.

Myers's Criteria -12

Why only possibly?

It may be useful to insist that *ptop* be implemented by direct access to the data structure.

Myers's Criteria -13

In fact, this is why we trade a slight reduction in cohesion for a big reduction in coupling.

Yes, the individual operations of the stack package are more cohesive than the package, but if you have to change one of those operation bodies you probably have to change all of them. So why not lump them all together?

Britton & Parnas's Ideas -1

Britton and Parnas say:

The overall goal of the decomposition into modules is the reduction of software cost by allowing modules to be designed, [**implemented*****], and revised independently....***

Britton & Parnas's Ideas -2

Each module's structure should be simple enough that it can be understood fully; it should be possible to change the implementation of [**a module*****] without knowledge of the implementation of other modules and without affecting the behavior of other modules; [*****and*****] the ease of making a change in the design should bear a reasonable relationship to the likelihood of the change being needed.***

Booch's Ideas -1

Booch adds to this discussion:

There is a pragmatic edge to these guidelines. In practice, the cost of recompiling the body of a module is relatively small; only that unit need be recompiled and application relinked. However the cost of recompiling the **interface** ***of a module is relatively high.***

Booch's Ideas -2

Especially with strongly typed languages, one must recompile the module interface, its body, **all other modules that depend upon this interface, the modules that depend on these modules, and so on*****.***

[emphasis is mine]

Booch's Ideas -3

Thus, it is important to get the abstraction defined right.

The closer the abstraction's interface is to real life, the less likely it is to change.

Other Ideas -1

Still another criterion:

Build modules to make reuse convenient.

This causes ADTs to be designed to be logically complete. For example, if in one application using a stack, you never need to clear the stack to empty or ask if the stack is empty, you put these operations in anyway.

Other Ideas -2

Doing so

- **increases chances of reuse in other applications of stacks**
- **reduces chances that interface will have to be changed later; after all, the interface is exactly the abstraction; if the abstraction is logically complete, so is the interface.**

Other Ideas -3

How can you tell if the abstraction is complete?

One thing for sure, if you need an operation *o* to define another, then *o* is part of the abstraction. This accounts for the *is_empty* function in a *Stack* module.

For the others, good ol' experience is the only way to tell.

Why this Topic? -1

One point came up in discussion with one of you after hours:

We are admonished to avoid making design decisions while writing requirements documents.

We are admonished against doing design during requirements analysis.

Why this Topic? -2

Parnas's method for decomposing modules is clearly intended as a method for decomposing design.

However, it is a method for doing design while delaying implementation decisions.

Anything that is hidden can be changed and thus the issue for which it is a decision is effectively delayed!

Why this Topic? -3

But given that this is a design technique, why do I claim that Parnas's decomposition is a suitable basis for building requirement domain models?

In giving a domain model, we are trying to avoid pinning down the implementation, to avoid constraining the implementor beyond what is required to meet requirements.

Why this Topic? -4

However, the purpose of Parnas's method is precisely to allow any implementation.

We have proved for the KWIC example that the modularization allows implementations whose structure bears no resemblence to that of the modules.

Why this Topic? -5

I claim that any Parnas decomposition does not constrain implementations beyond what is necessary to meet requirements.

Moreover it turns out (and we can usually force it to be!) that if the elements of the decomposition correspond to elements of the problem description, then the decomposition makes a good domain model.

Why this Topic? -6

That's why I teach Parnas's method in a requirements engineering course!