3. Decomposition Into Modules

Overview of Chapter 3: Decomposition Into Modules

3.1 the *criteria* to be used in decomposing systems into modules
3.2 structuring complex software with the *module guide*
3.3 time and space decomposition of *complex structures*
3.4 designing software for ease of *extension and contraction*

3.1 The Criteria to be Used in Decomposing Systems into Modules

Text for Chapter 3.1

[Par72] Parnas, D. L. *On the criteria to be used in decomposing systems into modules*. Commun. ACM **15**(12), 1053–1058 (1972).

Seminal paper on information hiding and modularization. Still valid.

Additional Background for Chapter 3.1

[HoWe01] Hoffman, D. M. and Weiss, D. M., editors. *Software Fundamentals – Collected Papers by David L. Parnas*. Addison-Wesley (Mar. 2001).

A collection of important Parnas papers. With introductions on their history and current relevance. Includes [Par72].

What is a Module?

- historically: a unit of measure
  - e.g., 2,54 cm
- manufacturers learned to build parts that were one unit large
- word now: the parts themselves
- modules: relatively self-contained systems, combined to make a larger system
- design: often is assembly of many previously designed modules

The Constraints on Modules

- if modules are hardware:
  obvious how to put them together
  - well-known physical constraints
  - well-identified time for module assembly
- if modules are software:
  no obvious constraints
  - software modules can be arbitrarily large
  - their interfaces can be arbitrarily complex
- during software development:
  several different times at which parts are combined, several different ways of putting parts together

Modules of Software – When are Parts Put Together?

1. while *writing* software
   - parts: work assignments for programmer(s)
   - when: before compilation or execution
2. when *linking* object programs
   - parts: separately compiled (or assembled) programs
   - when: before execution
3. while *running* a program in limited memory
   - parts: executable programs or data
   - when: during run-time

- literature: uses "module" for all three!
- this ambiguity leads to confusion
- *this lecture: only the first meaning* ("while writing SW")

The Constraints on the Three Structures

what constrains our choice of "modularization"?

- for write-time "modules":
  - intellectual coherence for programmer
  - ability to understand, verify
  - ease of change
- for link-time "modules":
  - duplicate names
  - time needed to re-compile and link
- for run-time "modules":
  - memory size
  - frequency of references to items outside module
  - time needed to load into memory

- these three sets of constraints are independent
- only commonality: the word "module"
- three different design concepts

Old Example for a Confusion

- TSS/360
  - time sharing system by IBM, in the 60's
  - very slow
- a well-known IBM researcher:
  "reason is over-modularization"
  - memory thrashing
  - memory management interpretation
- previous popular wisdom:
  make modules as small as possible
  - work assignment interpretation
- two meanings were confused

Recent Example for a Confusion

- a recent book on "software architecture"
  - presents and compares different styles
    for organizing large software
  - text book
  - well-known authors
  - uses Parnas' KWIC example (see below)
- does not distinguish write-time / link-time modules
  - e.g., does run-time performance comparisons for write-time modules
- book not used for this lecture. . .

The Effect of Confusing the Meanings

- inefficiency, if
  - forcing write-time modules to be link-time modules:
    - ▷ overhead for frequently executed call sequences
  - forcing write-time modules to be run-time modules:
    - ▷ overhead for frequent memory loads
- high development/maintenance costs, if
  - forcing run-time modules to be write-time modules:
    - ▷ difficult to program and to maintain

- write-time modules need not be compiled separately
  one may use *macro substitution* or similar

Write-Time Modules

- we want the following properties:
  - can be designed and changed independently
  - can be sub-divided into further modules
- when to stop sub-dividing into modules?
  - when so small that it is easier
    to write a new one than to change it
  - when the cost of specifying the interface exeeds any future benefit
    from having smaller modules
- "module = work assignment" is only a definition,
  need guidelines for designing a module structure

Example: A KWIC Index Production System

- KWIC: "Key Words In Context"
- the KWIC index system accepts an ordered set of lines
- each line is an ordered set of words
- each word is an ordered set of characters
- any line may be "circularly shifted" by repeatedly removing
  the first word and appending it to the end of the line
- the KWIC index system outputs a listing of all circular
  shifts of all lines in alphabetical order

Example of a KWIC Index

input	output
THE COLOUR OF MAGIC	COLOUR OF MAGIC THE
THE LIGHT FANTASTIC	EQUAL RITES
EQUAL RITES	FANTASTIC THE LIGHT
MORT	LIGHT FANTASTIC THE
MOVING PICTURES	MAGIC THE COLOUR OF
	MORT
	MOVING PICTURES
	OF MAGIC THE COLOUR
	PICTURES MOVING
	RITES EQUAL
	THE COLOUR OF MAGIC
	THE LIGHT FANTASTIC

Output of The Unix ptx Utility

(ptx: "permuted index")

```
          THE   COLOUR OF MAGIC
                EQUAL RITES
    THE LIGHT   FANTASTIC
          THE   LIGHT FANTASTIC
THE COLOUR OF   MAGIC
                MORT
                MOVING PICTURES
   THE COLOUR   OF MAGIC
       MOVING   PICTURES
        EQUAL   RITES
                THE COLOUR OF MAGIC
                THE LIGHT FANTASTIC
```


Ideas for a Modularization

- pretend: programming task is so large that it must be performed by serveral persons
- how should we modularize the KWIC index software?
  - which modules?
  - which interfaces between modules?

  (discussion)

What are the Criteria for a Modularization?

- well, what are they?
- does our modularization meet them?

"Conventional" Modularization

1. Input Module

- reads data lines from input medium
- stores them in memory, packed four to a word
- end of word marker: an otherwise unused character
- makes index to show start address of each line

input interface: input format, marker conventions
output interface: memory format

2. Circular Shift Module

- called after input module
- makes index with addresses of first char. of shifts
- output is array of pairs of words (start of line, start of shift)

input interface: memory format
output interface: memory format, perhaps the same

3. Alphabetizing Module

- takes the arrays of modules 1 and 2
- produces an array in format of module 2
- the result is ordered alphabetically

input interface: memory format
output interface: memory format

4. Output Module

- takes the arrays of module 3 and 1
- produces formatted output listing
- maybe: mark start of line, . . .

input interface: memory format
output interface: paper format, conventions, . . .

5. Master Control Module

- controls the sequencing of the other modules
- handles error messages, memory allocation, . . .

interface: names of the program to be invoked

Some Likely Changes

1. input format
   (a) line break characters (`\n` / `\r\n` / `\r`)
   (b) word break characters
   (c) size of a character (7 bit / 8 bit / Unicode)
2. memory formats
   (a) keep all lines in memory?
   (b) pack characters four to a word?
   (c) store shifts explicitly / as index+offset
3. decision to sort all output before starting to print

4. decision to produce all shifts
   (a) eliminate shifts starting with noise words
   (b) eliminate shifts not starting with only-words
5. different alphabetizations
   (a) ignore case
   (b) locale
6. output format
   (a) different visual output layouts
   (b) truncate overlong lines in output
   (c) generate output for different formatting tools

Parnas' Modularization

1. Line Holder Module

- special purpose memory to hold lines of KWIC index

interface programs:

- GET_CHAR(lineno, wordno, charno)
- SET_CHAR(lineno, wordno, charno, char)
- CHARS(lineno, wordno)
- WORDS(lineno)
- LINES
- DELETE_LINE(lineno)
- DELETE_WORD(lineno, wordno)

2. Input Module

- reads from input medium
- calls line-holder programs to store in memory

interface program:

- INPUT

3. Circular Shift Module

- creates "virtual" list of circular shifts
- uses line holder programs to get data from memory
- may or may not create an actual table

interface programs:

- CS_SETUP
- CS_CHAR(lineno, wordno, charno)
- . . . (analogs to the other programs of the input module)

4. Alphabetizer Module

- does actual sorting of the shifts
- may or may not produce a new list
- if it doesn't, it makes a directory

interface programs:

- ALPH
- ITH(lineno)
- . . . (some more supporting programs)

5. Output Module

- does the actual printing
- calls ITH and circular shift programs

interface program:

- OUTPUT

6. Master Control Module

- links all modules together to do the job
- is the main program, but very simple
- calls INPUT, CS_SETUP, ALPH, and OUTPUT

Comparison of the Two Modularizations

- both:
  - small, manageable programs,
    to be programmed independently
  - may use same data representations
  - may use same algorithms
  - may result in identical code after compilation
- different:
  - way of cutting up the system
  - interfaces

- changeability:
  - 2nd modularization better changeable (compare list on slide 187)
- independent development:
  - 1st: cooperation of all teams until best data representation is found
  - 2nd: teams can start independently early
- comprehensibility:
  - 1st: output module can be understood only by understanding some constraints of the alphabetizer, shifter, and input module

The Criteria

criteria for designing *information-hiding* modules:

- identify the design decisions that are likely to change
  - requires experience and judgement
  - is additional work up-front
- have a module for each that is very likely to change

The *Secret* of a Module

- the design decision that might change
  - only the implementor needs to know what decision was made

Examples for Module Secrets

- line holder module
  - how lines are represented in memory
- input module
  - input format
- circular shift module
  - how shifts are represented
- alphabetizer module
  - sorting algorithm
  - time when alphabetization is done
- output module
  - output format

Some Specific Criteria

the following should be hidden in a single module:

- a data structure, its access and modyfying procedures
- a routine and its assembly call sequence
- control block formats (into a control block module)
- character codes, alphabetic orderings, . . .
- sequence of processing

Interface Between Modules

- the *assumptions* that they make about each other

Module Structure

system structure:
- a system's parts and their connections
  - connections: the modules' interfaces (i.e., assumptions)
  - parts: work assignments (modules)

Efficiency and Implementation

- frequency of switching between modules at run-time:
  - steps-in-processing approach: low frequency
  - information-hiding approach: high frequency
- module access programs need not be subroutines
  - the usual space-time tradeoffs apply
  - supporting language constructs:
    - ▷ macros (in C, C++, not in Java)
    - ▷ `inline` functions/methods (in C++, not in Java)
    - ▷ templates (in C++, not in Java)
  - automatically optimizing compilers
    - ▷ they know size of code, but not frequency of calls

Information Hiding and Abstract Data Types

- data abstraction is a special case of information hiding
  - algorithms can be hidden as well
- data types allow many copies of the hidden structure
  - each variable has one copy

Information Hiding and Object-Orientation

- both: group data and programs together
- information hiding: no inheritance
- OO: often no distinction of write-time/link-time modules

Information Hiding and Program Families

- designing not a single program, but a program family
- early: decisions shared by all members
- postpone: decisions likely to change

- see Chapters 3.4 and 5