3.4 Designing Software for Ease of Extension and Contraction

Text for Chapter 3.4

[Par79] Parnas, D. L. *Designing software for ease of extension and contraction*. IEEE Trans. Softw. Eng. **SE-5**(2), 128–138 (Mar. 1979).

Additional Background for Chapter 3.4

[Par76] Parnas, D. L. *On the design and development of program families*. IEEE Trans. Softw. Eng. **2**(1), 1–9 (Mar. 1976).

Stepwise refinement vs. information hiding; families of programs.

Motivation

some common complaints about software systems:

- deliver early release with subset of functionality?
  → the subset won't work until everything works
- add simple capability?
  → rewrite most of the current code
- remove unneeded capability?
  → rewrite much of the current code

A Family of Programs

- usually you don't write a single program,
  but a *family* of programs
- families of systems: Chapter 5
- here special case:
  families of programs where
  - some members are subsets of other members, or
  - several members share a large common subset

Alternatives for the Software Producer

- a "super" system
  - generality costs
    - ▷ memory, speed: still important for embedded systems
    - ▷ difference to mathematics
- a system for the "average" user
  - doesn't really fit for anybody
- a set of independently developed systems
  - with subtle differences → maintenance nightmare
- a subsettable "super" system
  - each family member offers a subset of services of the largest member

A Subsettable System

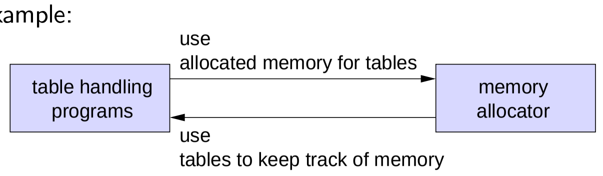
- individual installations only pay for what they need
  - computer resources
  - marketing
- incremental implementation possible
- allows for fail-soft subsets
- ability to contract by deleting whole programs,
  not by modifying programs
- ability to extend by adding programs,
  without changing programs

The Uses Hierarchy, Again

- is the key to subsets!
- kind of structure:
  - parts: programs (not modules)
  - relation: uses (i.e., requires-the-presence-of)
- time: design time
- definition of "uses":
  Given a program A with specification S and a program B,
  A uses B iff
  A cannot satisfy S unless B is present and functioning correctly

Design Error: Loops in the Uses Relation

example:



- neither works until both work
- if either is removed, the other no longer works
- should memory allocator build own tables?
  - code duplication

example (from Multics):

- virtual memory uses file system
- file system uses virtual memory

Basic Steps in the Design of a Subsettable System

1. identify the subsets
2. make list of programs belonging to each module
3. decide on uses matrix for the programs
4. construct the uses hierarchy from the matrix

Identify the Subsets

- during requirements definition
- search for minimal useful subset
- search for minimal useful increments
  - even if it appears trivial now
- each increment later becomes a write-time module in the design

Make List of Programs Belonging to Each Module

- access programs
- internal programs
  - cannot be used directly by outside programs
  - can use other programs
- main programs
  - cannot be used (are top-level)
  - can use other programs

Basic Steps in the Design of a Subsettable System

1. identify the subsets
2. make list of programs belonging to each module
3. decide on uses matrix for the programs
4. construct the uses hierarchy from the matrix

Decide on Uses Matrix for the Programs

- three possibilities for each pair (A, B)
  - A may use B
  - B may use A
  - neither may use the other

Conditions for Allowing Program A to Use Program B

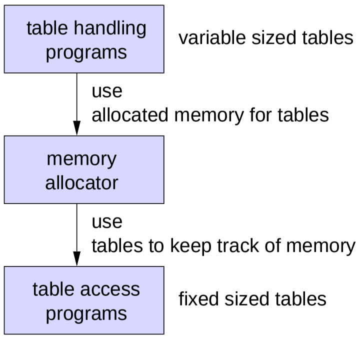
- A is simpler because it uses B
- B is not more complex because it is not allowed to use A
- there is a useful subset containing B and not A
- there are no useful subsets containing A and not B

- all conditions must be satisfied

Construct the Uses Hierarchy from the Matrix

- could be done by a tool
  - see Ada's "with" clause to make the uses relation explicit
- make list of programs at level 0
  - they don't use other programs
- work up from there
  - level 1 programs use only level 0 programs
  - level 2 programs . . .
- the uses matrix and hierarchy must be maintained, of course

Conflict Removal: Sandwiching



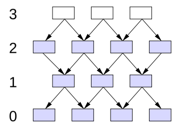
- message:

  a level (in the uses hierarchy)
  is not a module (in the write-time hierarchy)

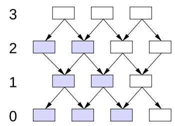
  - uses relationship is between programs, not modules
  - there are no "layers of abstraction"
  - in a subsetted system, there may be subsets of the programs in the modules
    - ▷ the designer of *each* module must identify the useful subsets

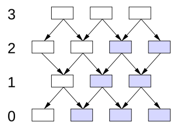
Deriving Subsets from the Uses Relation

- any level is a subset



- can also omit parts of levels





Levels and Virtual Machines

- def. virtual machine: a set of variables and operations, implemented in software
- each level is a virtual machine
  - applications programs are simpler: they use virtual machine programs
- upper level machines are *less powerful*
  - resources used to implement a VM must not be available to a program that uses the VM
  - upper level machines more specialized
- upper level machines are more convenient and safer

Evaluation Criteria for a Uses Hierarchy

1. all desirable subsets?
2. no duplicated or almost alike programs?
3. is it simple?

Getting All Desirable Subsets

- principle of minimal steps
  - start with minimal useful subset
  - minimal useful increments
- examples of violation:
  - RC4000 operating system combined synchronization and message passing
  - Hydra operating system combined parameter passing and run-time type checking

The One, Fixed, Variable Pattern

- a common, useful pattern for designing a uses hierarchy
- three levels of operations:
  - operations on *one* item
  - operations on a *fixed* number of similar items
  - operations on a *variable* number of similar items
- you might want to have three subsets
- language/library support for "fixed", "variable" supersets of "one" data element
  - C++, Java, . . .

Example: an Address Processing System

- read, store, and write out lists of addresses
- example taken from [Par79]

Information in an Address

- last name
- given names
- organization
- internal identifier
- street address or P.O. box
- city or mail unit identifier
- state
- Zip code
- title
- branch of service if military
- GS grade if civil service

- each field may be empty

Basic Assumptions

- the items on previous slide will be processed by all application programs
- the input formats are subject to change
- the output formats are subject to change
- choice of input/output format for different systems:
  - fixed format
  - run-time choice from a fixed set
  - user-specified format definition language

  (one/fixed/variable)
- representation of addresses in main memory will vary

- most systems: only a subset of addresses in main memory at any one time
  - number needed may vary
  - some systems: number needed may vary at run-time

Proposed Design Decisions

- input and output programs will be table driven
  - table specifies format
  - secret of input and output modules:
    content and organization of format tables
- secret of address storage module (ASM):
  representation of addresses in main memory
  - changing a part of an address is cheaper than growing or shrinking the address table

- address file module (AFM):
  used if more addresses than main memory
  - interface compatible to ASM
  - provides additional operations for efficient sequential iteration
- implementation of AFM has ASM, BFM as submodule
  - block file module (BFM):
    stores data blocks (size of at least an address),
    does not look at content
  - the ASM within the AFM has two interfaces:
    - ▷ "normal" interface: addresses and their fields
    - ▷ interface for blocks of contiguous storage, input/output
  - BFM might be part of operating system

Access Programs of "Normal" Interface of ASM

addTit:	asm × integer × string → asm	fetTit:	asm × integer → string
addGN:	asm × integer × string → asm	fetGN:	asm × integer → string
addLN:	asm × integer × string → asm	fetLN:	asm × integer → string
addServ:	asm × integer × string → asm	fetServ:	asm × integer → string
addBOrC:	asm × integer × string → asm	fetBOrC:	asm × integer → string
addCOrA:	asm × integer × string → asm	fetCOrA:	asm × integer → string
addSOrP:	asm × integer × string → asm	fetSOrP:	asm × integer → string
addCity:	asm × integer × string → asm	fetCity:	asm × integer → string
addState:	asm × integer × string → asm	fetState:	asm × integer → string
addZip:	asm × integer × string → asm	fetZip:	asm × integer → string
addGsL:	asm × integer × string → asm	fetGsL:	asm × integer → string
setNum:	asm × integer → asm	fetNum:	asm × integer

Component Programs of Address Input Module

InAd:	Reads in an address in the currently selected format and calls ASM or AFM programs to store it.
InFSel:	Selects a format from an existing set of format tables for InAd. There is always a format selected.
InFCr:	Adds a new format to the tables used by InFSel. The format is specified in a "format language". Selection is not changed.
InTabExt:	Adds a blank table to the set of input format tables.
InTabChg:	Rewrites a table in the input format tables. Selection is not changed.
InFDel:	Deletes a table from the set of format tables. The selected format cannot be deleted.
InAdSel:	Reads in an address using one of a set of formats. Choice is specified by an integer parameter.
InAdFo:	Reads in an address in a format specified as one of its parameters (a string in the format definition language). The format is selected and added to the tables and subsequent addresses could be read in using InAd.

Component Programs of Address Output Module

OutAd: Prints out an address in the currently selected format. The information is in an ASM and identfied by its position there.
OutFSel: Selects a format from an existing set of format tables for OutAd. There is always a format selected.
OutFCr: Adds a new format to the tables used by OutFSel. The format is specified in a "format language". Selection is not changed.
OutTabExt: Adds a blank table to the set of output format tables.
OutTabChg: Rewrites a table in the output format tables. Selection is not changed.
OutFDel: Deletes a table from the set of format tables. The selected format cannot be deleted.
OutAdSel: Prints out an address using one of a set of formats. Choice is specified by an integer parameter.
OutAdFo: Prints out an address in a format specified as one of its parameters (a string in the format definition language). The format is selected and added to the tables and subsequent addresses could be printed using OutAd.

Component Programs of Address Storage Module

Fet<CompName>: Read information from an address store. (See Slide 333.)
Add<CompName>: Write information in an address store. (See Slide 333.)
GetBlock: Takes an integer parameter, returns a storage block.
SetBlock: Takes a storage block and an integer. Changes the contents of an address store – reflected by the Fet<CN> programs.
AsmExt: Extends an address store by appending a new address with empty components at the end of the address store.
AsmShr: "Shrinks" the address store.
AsmCr: Creates a new address store. The parameter specifies the number of components. All components are initially empty.
AsmDel: Deletes an existing address store.

Component Programs of Block File Module

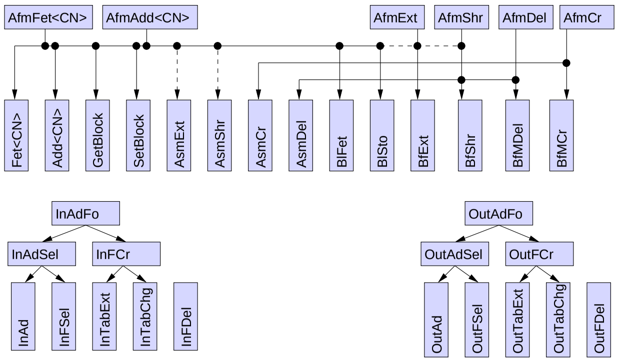
BlFet: Takes an integer and returns a "block".
BlSto: Takes a block and an integer and stores the block.
BfExt: Extends BFM by adding additional blocks to its capacity.
BfShr: Reduces the size of the BFM by removing some blocks.
BfMCr: Creates a file of blocks.
BfMDel: Deletes an existing file of blocks.

Component Programs of Address File Module

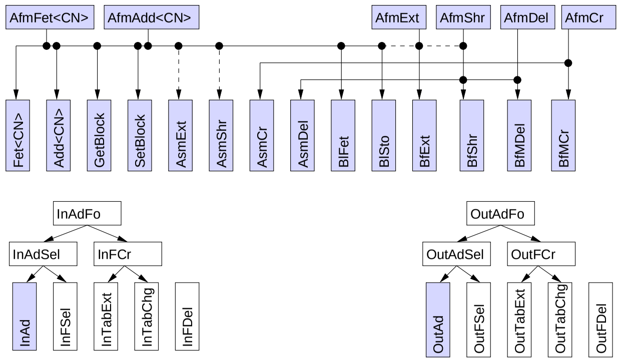
- provides all ASM programs except GetBlock and SetBlock.
- the programs are renamed as follows:

AfmFet<CompName>: As in ASM.
AfmAdd<CompName>: As in ASM.
AfmExt: As in BFM.
AfmShr: As in BFM.
AfmCr: As in BFM.
AfmDel: As in BFM.

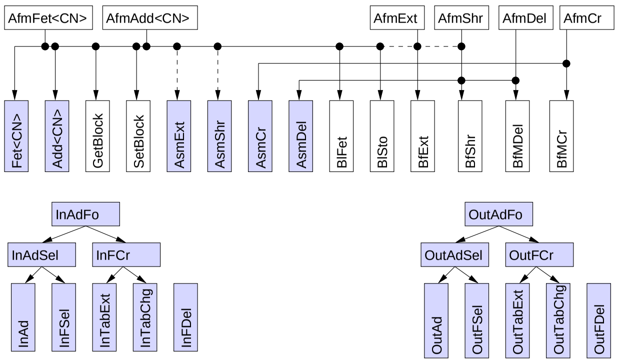
Uses Relation of the System



Subset: Addresses in a Single Format



Subset: Small Set of Addresses



Subset: Query-Only System