Programming Languages Seminar

Program Structure and readability

Lefel Yaniv
Hagay Pollak

1

Decomposition into modules

- On the criteria to be used in decomposing systems into modules – by D.L.Parnas.(1972)

2

Introduction

- The philosophy of modular programming (1970)
  - Segmentation of project effort.
  - System modularity.
  - Inputs and outputs are well defined.
  - Module integrity is tested independently.
  - System is maintained in modular fashion.
    - System errors and deficiencies can be traced to specific modules.
    - Limiting the scope of detailed error searching.

Modular programming advancement

- Major advancement in modular programming:
  - A module can be written with little knowledge of code in another module.
  - Modules can be replaced without reassembly of the whole system.

Benefits of modular programming

- Managerial – development time reduced.
- Product flexibility – one module can be changed independently of others.
- Comprehensibility – System is better designed because it is better understood.

What is modularization ?

- Modularization: partial system description, design decisions made prior to commence of work.
- Module – a responsibility assignment

- What are the criteria to be used in dividing the system into modules ?

7

Case study: KWIC index – description

- Input:
  - An order set of lines.
  - Each line is an ordered set of words,
  - Each word is an ordered set of characters.
- Output:
  - Listing of all circular shifts of all lines in alphabetical order.

8

KWIC index – Modularization 1

- Example:

Input	Output
AA CC BBB	AA CC BBB
CDCD ABA	BBB AA CC
	CC BBB AA
	ABA CDCD
	CDCD ABA

Case study: KWIC index – remarks

- “Such a system could be produced by a good programmer within a week or two”.
- We must go through the exercise of treating this problem as if it were a large project.

KWIC index – Modularization 1

- Module 1: Input
  - Read the input into a memory. Create a list of pointers which point to the beginning of each line.
- Module 2: Circular Shift.
  - Prepares an index of the first characters of each circular shift. Eventually creating a list of pairs (line number, starting address).

11

KWIC index – Modularization 1 Cont’

Example (module 2)

Original line: Line 2: BBB AA CC

Module 2 output for line 2:

BBB AA CC

AA CC BBB

CC BBB AA

Represented by

(2,+0),(2,+4),(2,+7).

12

KWIC index – Modularization 1 Cont'

- Module 3: Alphabetizing.
  - Takes the arrays produced by modules 1,2.
  - Arranges the circular shifts in alphabetically order.
  - New data is kept in same format as in module 2.

KWIC index – Modularization 1 Cont'

Example (module 3)

Original data set: (2,+0),(2,+4),(2,+7).

Alphabetically sorted set: (2,+4),(2,+0),(2,+7)

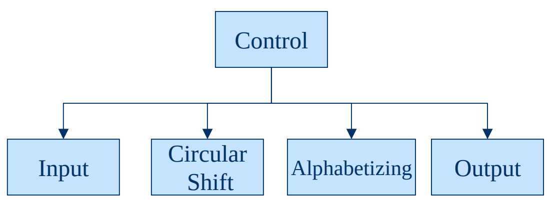
Before	After
(2,+0),(2,+4),(2,+7)	(2,+4),(2,+0),(2,+7)
BBB AA CC	AA CC BBB
AA CC BBB	BBB AA CC
CC BBB AA	CC BBB AA

KWIC index – Modularization 1 Cont’

- Module 4: Output.
  - Listing circular shifts.
- Module 5: Master Control
  - Control sequencing (modules 1,2,3,4).
  - Handle error messages, space allocation, etc.

15

KWIC index – Modularization 1 scheme



Modularization 1 scheme

16

KWIC index – Modularization 2

- Module 1: Line Storage
  - Handles all data storage of the lines and characters.
- Module 2: Input.
  - Reads lines from input.
  - Calls Line Storage module to store the lines in the memory.
- Module 3: Circular Shift.
  - Function CSSETUP() – must be called before the used of other functions in the module.
  - Function CSCHAR(I,w,c) – provides the value representing the cth character in the wth word of the Ith circular shift of all lines.
  - Other functions TBD.

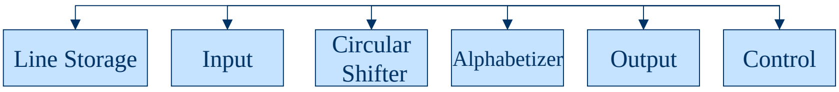
17

KWIC index – Modularization 2

- Module 4: Alphabetizer
  - Function ALPH – performs module setup.
  - Function ITH(i) – returns the index of the circular shift line which comes ith in the alphabetical ordering.
- Module 5: Output
  - Provides interface for printing the desired output.
- Module 6: Master Control.
  - Similar to modularization 1.

18

KWIC index – Modularization 2 Static Model

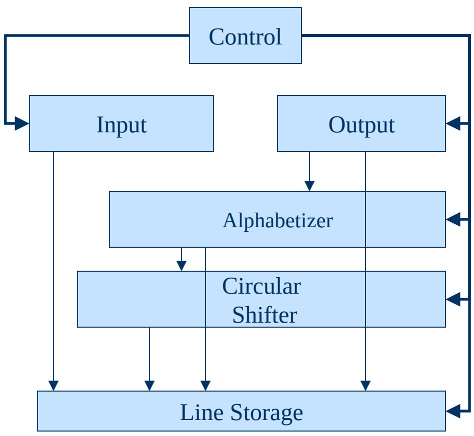


Modularization 2 scheme

* Additional internal connections might exist.

19

KWIC index – Modularization 2 Dynamic Model



20

Modularization comparison – Commonalities

- Both decompositions will work.
- Both provide a well defined segmentation of the system.
- The decompositions are identical in the runnable representation (same output).

Modularization comparison – Dissimilarities

- The two decompositions have different **representations** for
  - Changing
  - Documenting
  - Understanding
  - Etc.
- Other **representations** are part of the system, and not only the running part.

Modularization comparison – Changeability

- Design decisions which are likely to change (partial list):
  - Input format
  - Data structure
  - Function (method) implementation
  - Sequencing of events (setup and build a list or trace data according to demand)

Modularization comparison – KWIC Changeability examples

Change \ Modular	I	II
Input format	Input module	Input module
Data packing	All modules	Line Storage module
Circular shift format	Alphabetizer, output, circular shifting modules	Circular shifter module
Alphabetizing	Output, alphabetizer	Alphabetizer

Modularization comparison – Independent Development

- Modularization I
  - Modules share the same physical data structure.
  - The development of data formats will be a joint effort by the development groups.
- Modularization II
  - Abstract interfaces, consist mainly in function names and parameters.
  - Relatively simple decisions required, thus development of modules begins earlier (than in modularization I).

Modularization comparison – Comprehensibility

- Modularization I
  - The system will only be comprehensible as a whole.
- Modularization II
  - Each module is independent. To understand a module, one needs only to understand the module, and the interfaces to other modules.

Decompositions - Criteria used

- Modularization I
  - Each module is a major step in the processing.
  - Modules derive from a flowchart.
- Modularization II
  - “Information hiding” used both for data and for implementation.

Decompositions - Criteria offered (encapsulating into modules)

- Data structure and accessing functions.
- Similar data should be hidden in a module.
- Sequence of instructions.
- Sequence of processing of data items.

Decompositions - Efficiency

- The separation between modules is required for the readability representation, not in the running representation of the program.
- Inter module function calls raise procedure call overhead.
- Can be improved by compiler - time optimization.

Hierarchical structure

- Hierarchical structure is used if a module "uses" or "depends on" another module.
- Hierarchical structure and decomposition are two independent properties of system structure.
- Benefits of partial ordering:
  - Higher levels of the system are simplified since they use the services of lower levels.
  - Ability to replace system components in the hierarchical structure.

Conclusion

- It is incorrect to decompose the system into modules on the basis of a flowchart.
- Modules will not necessarily correspond to steps in the processing.
- Build a list of design decision likely to change.
- Each module is designed to hide such decisions from the others.
- Assemble programs as a collection of code from various modules.

31

Structured programming

- By Edsger W. Dijkstra (1969)

32

Introduction

- Is it possible to increase programming ability?
- What techniques should then be applied in the process of program composition?

Program size

- This discussion is concerned with programs that are large due to complexity of their task.
- Consider a program including N modules.
- Let us assume the probability of correctness of each module is p.
- Therefore the probability of correctness of the program is $Q = p^N$.

Program size

- For a large N, p must be close to 1, if Q is to differ significantly from 0.
- Combining subsets into large components does not improve the correctness of the program. $p^{(N/2)} * p^{(N/2)} = p^N = Q$.

35

Program size – family of programs

- A large program is always a series of versions of the program.
- Different versions perform the same and/or similar tasks.
- We consider a program to be a member of a family, sharing components, correctness and substructure.

36

Program correctness

- Program testing can be used to show the presence of bugs, but never to show their absence.
- Program correctness should be proved on account of the program text.

37

Program correctness

- Program correctness can be proved.
- The effort required to prove program correctness may grow exponentially with program growth.

38

The relation between program and computation

- When programs are expressed as linear sequence of statements, sequencing should not be controlled by statements transferring control to labeled points (e.g. goto statements).

The relation between program and computation

- Let us consider a program, P1, of the form: $S_1, S_2, S_3, \ldots, S_N$; where S_i, is an individual statement.
- N steps of reasoning are needed to establish the correctness of P1.
- For the statement: <u>if</u> B <u>then</u> S_1 <u>else</u> S_2 , 2 steps of reasoning are needed.

The relation between program and computation

- Let P2 be the program:

<u>if</u> B_1 <u>then</u> S_{11} <u>else</u> S_{12}
<u>if</u> B_2 <u>then</u> S_{21} <u>else</u> S_{22}
…
<u>if</u> B_N <u>then</u> S_{N1} <u>else</u> S_{N2}

- To reduce P2 to the form of P1 it takes 2N steps.
- And then another N steps to understand the form of P1. Altogether 3N steps.

41

The relation between program and computation

- Trying to understand the algorithm as S_{ij} would lead to N*2^N steps of reasoning.
- Explanation: for each N statements consider 2^N options of executions.
- Conclusion: programs of the form P1 are preferable for step-wise abstraction.

42

Abstract data structures

- Abstract data structures and abstract statements (e.g. routines) represent design decisions.
- They are the natural unit of interchange for program modification.
- Let us call such a unit - a “pearl”.

Programs as necklaces strung from pearls

- A program is an ordered set of pearls – a necklace.
- The top pearl describes the program in its most abstract form.
- Lower pearls define and refine the upper pearls.
- Pearl seems to be a natural program module.

Programs as necklaces strung from pearls

- Specific design decision is actually an aspect of original problem statement.
- A pearl embodies specific design decision.
- Lower half of a necklace is the implementation of the upper half.
- Thus, the correctness of the upper half of the necklace can be established regardless of the choice of the bottom half.

45

Programs as necklaces strung from pearls

- The family of programs is the set of selections from a collection of pearls that can be strung into a necklace.

46

Conclusion

- Testing of a program is not a proof of correctness.
- The proof process requires abstraction of the statements.
- Design using the pearl model provides the abstraction required for the proof of correctness.

47

Goto statement

- "Goto statement considered harmful" –

  By Edsger W. Dijkstra (1968)

48

Motivation

- A programmer's activity seems to end when he constructed a correct program.
- A process is the dynamic behavior of a program.
- The main issue of the program is its process.

Motivation – cont'

- A program is the static description of a process.
- Our powers to visualize dynamic behavior are poorly developed.
- Our objective is to shorten the conceptual gap between the static program and the dynamic process.

Process progress

- Suppose a process stopped during execution, how can we redo the process to the same point?

51

Process progress

- Consider a program which includes assignments and conditional clauses.
- It is sufficient to point to the relevant text.
- Such a pointer will be called: “textual pointer”.

52

Process progress – cont’

- As we include procedures in the program, we also have to give an index to the procedure call.
- We characterize the progress of the process by a sequence of textual indices.

53

Process progress – cont’

- As we include repetition clauses in the program, we use “dynamic indexing”.
- Each entry into a repetition clause changes the index.
- The “dynamic index” enables counting of repetitions.
- The progress of the process is uniquely described by a (mixed) sequence of textual and/or dynamic indices.

54

Process progress – cont'

- We have a defined a coordinate system, describing the progress of the process.
- Now we can evaluate every variable in the program.

55

Goto statements

- Goto statements make it hard to find a meaningful coordinate system describing the progress of the process.
- The difficulty is that such a system, although unique is still unhelpful.
- We can't maintain a list of goto calls, as we did in the function calls, since the return locations are hard to trace.

56

Conclusion

- Goto statements make it hard to understand, read the code, and analyze the progress of the process.
- Since one of the roles of the program is the text representation, use of goto statements misses the program goals and should not be used.

57

Discussion

58