

**SE 2AA4, CS 2ME3 (Introduction to Software
Development)**

Winter 2017

13 Module Decomposition (Ghezzi Ch. 4, H&S Ch. 7)

Dr. Spencer Smith

Faculty of Engineering, McMaster University

February 3, 2017



Module Decomposition

- Administrative details
- Module decomposition
- Software architecture
- Design for change
- Relationship between modules
- The USES relation
- Module decomposition by secrets
- The IS_COMPONENT_OF relation
- Techniques for design for change
- Module guide

Administrative Details

- Assignment 1
 - ▶ E-mail the instructor if you haven't received your partner's code
 - ▶ Lab report due by 11:59 pm February 2
- Assignment 2
 - ▶ Files due by 11:59 pm Feb 15
 - ▶ E-mail partner files by 11:59 pm Feb 16
 - ▶ Lab report due by 11:59 pm Feb 27
- Midterm exam
 - ▶ March 1, 7:00 pm, TSH/120
 - ▶ 90 minute duration
 - ▶ Multiple choice - 30–40 questions?
 - ▶ Open book (any paper)

Assumptions versus Exceptions

- The assumptions section lists assumptions the module developer is permitted to make about the programmer's behaviour
- Assumptions are expressed in prose
- Use assumptions to simplify the MIS and to reduce the complexity of the final implementation
- Interface design should provide the programmer with a means to check so that they can avoid exceptions
- When an exceptions occurs no state transitions should take place, any output is *don't care*

QueueADT Module Syntax (Abstract Object)

What is missing from this interface?

Exported Access Programs

Routine name	In	Out	Exceptions
q_init		queueT	
add	T		NOT_INIT, FULL
pop			NOT_INIT, EMPTY
front		T	NOT_INIT, EMPTY
size		integer	NOT_INIT
isempty		boolean	NOT_INIT
isfull		boolean	NOT_INIT

If MAX_SIZE is exported, what could you replace isempty and isfull by? (This new interface will move some work to the programmer.)

Quality Criteria

- Consistent
 - ▶ Name conventions
 - ▶ Ordering of parameters in argument lists
 - ▶ Exception handling, etc.
- Essential - omit unnecessary features (only one way to access each service)
- General - cannot always predict how the module will be used
- As implementation independent as possible
- Minimal - avoid access routines with two potentially independent services
- High cohesion - components are closely related
- Low coupling - not strongly dependent on other modules
- Opaque - information hiding

QueueADT Module Syntax (Abstract Object)

Is this interface minimal?

Exported Access Programs

Routine name	In	Out	Exceptions
q_init		queueT	
add	T		NOT_INIT, FULL
pop		T	NOT_INIT, EMPTY
size		integer	NOT_INIT
isinit		boolean	

- front has been merged with pop
- size replaces isempty and isfull
- isinit is added

Modular Decomposition

- Until now our focus has been on individual modules, but how do we decompose a large software system into modules?
- We need to decompose the system into modules, assign responsibilities to those modules and ensure that they fit together to achieve our global goals
- We need to produce a software architecture
- The architecture (modular decomposition) is summarized in a Software Design Document

Software Architecture

- Shows gross structure and organization of the system to be defined
- Its description includes the description of
 - ▶ Main components of the system
 - ▶ Relationship among those components
 - ▶ Rationale for decomposition into its components
 - ▶ Constraints that must be respected by any design of the components
- Guides the development of the design

Specific Techniques for Design for Change

What software tool would you use if you wanted to select at build time between two implementations of a module, each distinguished by a different decision for their shared secret?

Specific Techniques for Design for Change

- Anticipate definition of all family members
- Identify what is common to all family members, delay decisions that differentiate among different members
- Configuration constants
 - ▶ Factor constant values into symbolic constants
 - ▶ Compile time binding
 - ▶ `MAXSPEED = 5600`
- Conditional compilation
 - ▶ Compile time binding
 - ▶ Works well when there is a preprocessor, like for C
 - ▶ If performance is not a concern, can often “fake it” at run time
- Make
- Software generation
 - ▶ Compiler generator, like yacc
 - ▶ Domain Specific Language

Questions

- What relationships have we discussed between modules?
- Are there desirable properties for these relations?

Relationships Between Modules

- Let S be a set of modules

$$S = \{M_1, M_2, \dots, M_n\}$$

- A binary relation r on S is a subset of $S \times S$
- If M_i and M_j are in S , $\langle M_i, M_j \rangle \in r$ can be written as $M_i r M_j$

Relations

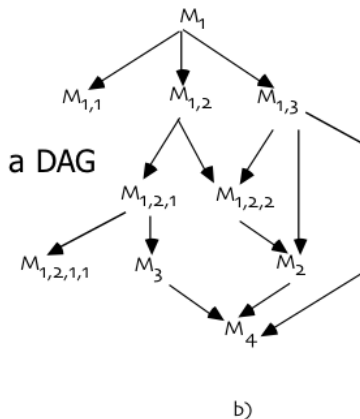
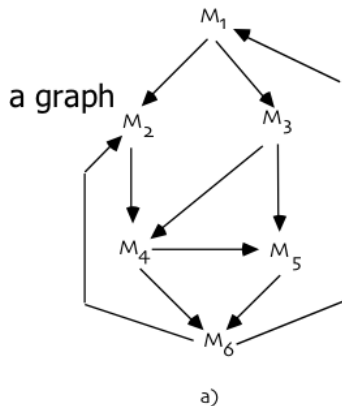
- Transitive closure r^+ of r

$M_i r^+ M_j$ iff $M_i r M_j$ or $\exists M_k$ in S such that $M_i r M_k$ and $M_k r^+ M_j$

- r is a hierarchy iff there are no two elements M_i, M_j such that $M_i r^+ M_j \wedge M_j r^+ M_i$

Relations Continued

- Relations can be represented as graphs
- A hierarchy is a DAG (directed acyclic graph)



Why do we prefer the uses relation to be a DAG?

References

- Parnas, David L, Software Fundamentals: collected papers by David L. Parnas, edited by Daniel M. Hoffmann and David M. Weiss, Lucent Technologies and Daniel M. Hoffmann, 2001, ISBN 0-201-70369-6
- Parnas, D. L., “On a ‘Buzzword’: Hierarchical Structure”, IFIP Congress 74, North Holland Publishing Company, 1974, pp. 336–339
- Parnas, D. L., “On the Criteria to be Used in Decomposing Systems into Modules”, Communications of the ACM, 15, 12, December 1972, pp. 1053–1058.

References Continued

- Parnas, D. L., “Designing Software for Ease of Extension and Contraction”, Copyright 1979, IEEE Transaction on Software Engineering, March 1979, pp. 128–138,
- Parnas, D. L., Clements, P. C., Weiss, D. M., “The Modular Structure of Complex Systems”, IEEE Transaction on Software Engineering, March 1985, Vol SE-11, No. 3, pp. 259-266 (special issue on the 7th International Conference on Software Engineering)

References Continued

- Parnas, D. L., Clements, P. C., “A Rational Design Process: How and Why to Fake it”, IEEE Transactions on Software Engineering, Vol. SE-12, No. 2, February 1986, pp. 251-257.
- Parnas, On the design and development of program families, IEEE Transactions on Software Engineering, SE-2(1), March 1976.
- Hoffmann, Daniel, M., and Paul A. Strooper, Software Design, Automated Testing, and Maintenance A Practical Approach, International Thomson Computer Press, 1995, <http://citeseer.ist.psu.edu/428727.html>

References Continued

- Dahl, Dijkstra and Hoare, Structured Programming, Academic Press, 1972 (modular decomposition)
- ElSheikh, Ahmed, W. Spencer Smith, and Samir E. Chidiac. (2004) Semi-formal design of reliable mesh generation systems. Advances in Engineering Software, Vol 35, Issue 12, pp 827-841.
- Carlo Ghezzi, Mehdi Jazayeri and Dino Mandrioli, Fundamentals of Software Engineering, 2nd Ed., Prentice Hall, 2003

References Continued

- Dijkstra, The structure of THE multiprogramming system. Communications of the ACM, 11(5): 341-346, May 1968.
- Linger, Mills and Witt. Structured Programming: Theory and Practice, Addison-Wesley, 1979 (step-wise refinement)
- Wirth, Program development by stepwise refinement, Communications of the ACM, 14(4):221-227, April 1971.