

**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 1, 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
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

- How to define the structure of a modular system?
- What are the desirable properties of that structure?

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

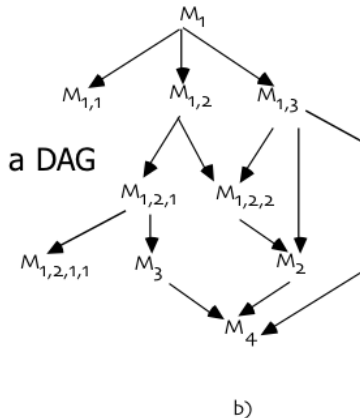
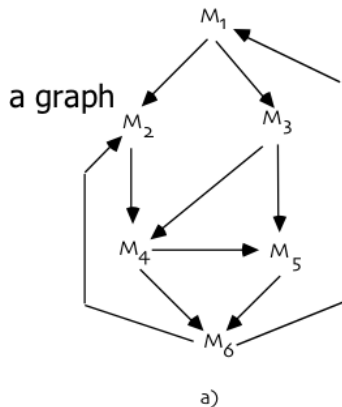
- Transitive closure r^+ of r

$M_i r^+ M_j$ 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)



DAG Versus Tree

Is a DAG a tree? Is a tree a DAG?

DAG Versus Tree

Would you prefer your uses relation is a tree?

The USES Relation

- A uses B
 - ▶ A requires the correct operation of B
 - ▶ A can access the services exported by B through its interface
 - ▶ This relation is “statically” defined
 - ▶ A depends on B to provide its services
 - ▶ For instance, A calls a routine exported by B
- A is a client of B; B is a server
- Inheritance, Association and Aggregation imply Uses

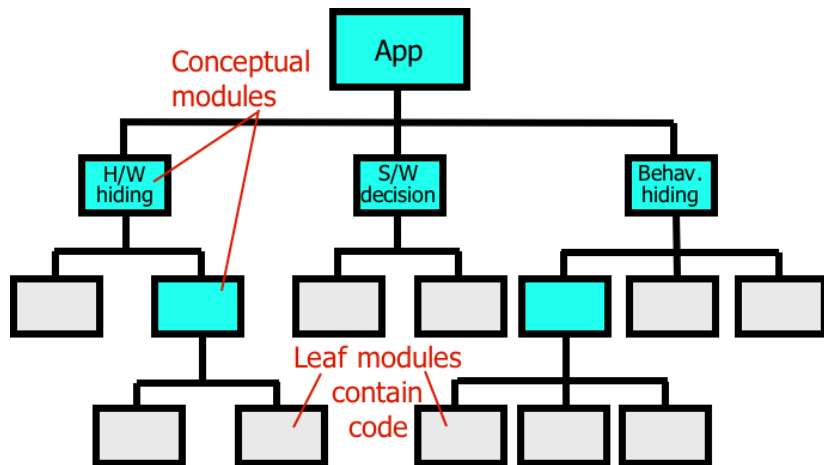
Desirable Properties

- USES should be a hierarchy
 - ▶ Hierarchy makes software easier to understand
 - ▶ We can proceed from the leaf nodes (nodes that do not use other nodes) upwards
 - ▶ They make software easier to build
 - ▶ They make software easier to test
- Low coupling
- Fan-in is considered better than Fan-out: WHY?

Hierarchy

- Organizes the modular structure through **levels of abstraction**
- Each level defines an **abstract (virtual) machine** for the next level
- Level can be defined precisely
 - ▶ M_i has level 0 if no M_j exists such that $M_i r M_j$
 - ▶ Let k be the maximum level of all nodes M_j such that $M_i r M_j$, then M_i has level $k + 1$

Module Decomposition (Parnas)



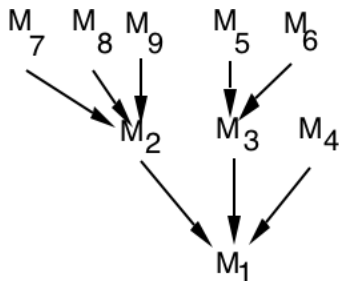
Module Decomposition (Parnas)

Does the module decomposition on the previous slide show a Uses relation? Is it a DAG? Is it a tree?

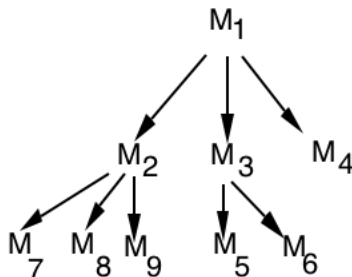
IS_COMPONENT_OF

- The Parnas decomposition by secrets gives an IS_COMPONENT_OF relationship
- Used to describe a higher level module as constituted by a number of lower level modules
- A IS_COMPONENT_OF B means B consists of several modules of which one is A
- B COMPRISES A
- $M_{S,i} = \{M_k | M_k \in S \wedge M_k \text{ IS_COMPONENT_OF } M_i\}$ we say that $M_{S,i}$ IMPLEMENTS M_i
- How is IS_COMPONENT_OF represented in UML?

A Graphical View



(IS_COMPONENT_OF)



(COMPRISES)

They are a hierarchy

Product Families

- Careful recording of (hierarchical) USES relation and IS_COMPONENT_OF supports design of program families
- Attempt to recognize modules that will differ in implementation between family members
- New program family member should start at the documentation of the design, not with the code

Remember - Information Hiding

- Basis for design (i.e. module decomposition)
- Implementation secrets are hidden to clients
- They can be changed freely if the change does not affect the interface
- Try to encapsulate changeable requirements and design decisions as implementation secrets within module implementations
- Decomposition by secrets, not by sequence of steps

Prototyping

- Once an interface is defined, implementation can be done
 - ▶ First quickly but inefficiently
 - ▶ Then progressively turned into the final version
- Initial version acts as a prototype that evolves into the final product

Module Guide

- Part of Parnas' Rational Design Process (RDP)
- When decomposing the system into modules, we need to document the module decomposition so that developers and other readers can understand and verify the decomposition
- Parnas proposed a Module Guide (MG) based on the decomposition module tree shown earlier

RDP - MG

- The MG consists of a table that documents each module's service and secret
- Conceptual modules will have broader responsibilities and secrets
- Following a particular branch, the secrets at lower levels "sum up" to the secret at higher levels
- The leaf modules that represent code will contain much more precise services and secrets
- Only the leaf modules are actually implemented
- The MG should list the likely and unlikely changes on which the design is based

Example

3. Module Hierarchy

Level 1	Level 2	Level 3	Level 4
Hardware-Hiding module	Input Device module	Mouse Motion module	
	Output Device module	Keyboard Input module	
Function Drivers module		File Reading module	
	Screen Display module		
	File Writing module		
	Master Control module		
	Frame Display module		
	User Command Detect module		
Behavior-Hiding module	Geometry Specification module	Physical Attributes Specification module	Boundary Specification module
			Subdivision Specification module
			Material Property Specification module
			Boundary Condition Specification module
	Save_Load module	Save_Load Input File module	
		Write Output File module	
Shared Services module	Frame Geometry module		
	Error Handle module		
	Mesh Drawing module		
	Drawing Tools module		
Software Decision module	Input Data module		Combinatorial Grid module
	Mesh Data module		Geometric Grid module
			Physical Attributes module
	Generic Tools module		Grid Function Vector module
			Edge Iterator module
			Boundary Iterator module
			Cell Neighbor Search module
	Mesh Generating Algorithm module		Geometric Mesh Generation module
			Physical Attributes Assignment module

Module Details

- For each module
- Module name
- Secret (informal description)
- Service or responsibility (informal description)
- For “leaf” modules add
 - ▶ Associated requirement
 - ▶ Anticipated change
 - ▶ Module prefix

RDP - MIS

- For each leaf module we need to document its interface and its implementation
- In RDP, the interfaces are documented in the Module Interface Specification (MIS)
- We have already seen MIS examples specified as Module State Machines

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.