

# CAS 741, CES 741 (Development of Scientific Computing Software)

Fall 2018

## 13 Modular Design

Dr. Spencer Smith

Faculty of Engineering, McMaster University

October 23, 2018



# Modular Design

- Administrative details
- Questions?
- Overview of design
- Modular decomposition: advantages, guidelines etc.
- Module guide
- Module guide example
- Integration Testing

# Administrative Details

- VnV GitHub issues for colleagues
  - ▶ Assigned 1 colleague (see Repos.xlsx in repo)
  - ▶ Provide at least 5 issues on their VnV Plan
  - ▶ Grading as before
  - ▶ Due by Friday, Oct 26, 11:59 pm
  - ▶ If you have an extension past today, please let your partner know
  - ▶ If your partner has an extension, you have three days after their deadline
- Grading scheme for VnV now available on Avenue
- Template for MG updated in repo

# Administrative Details: Deadlines

MG Present	Week 08	Week of Oct 29
MG	Week 09	Nov 5
MIS Present	Week 10	Week of Nov 12
MIS	Week 11	Nov 19
Unit VnV or Impl. Present	Week 12	Week of Nov 26
Unit VnV Plan	Week 13	Dec 3
Final Doc	Week 14	Dec 10

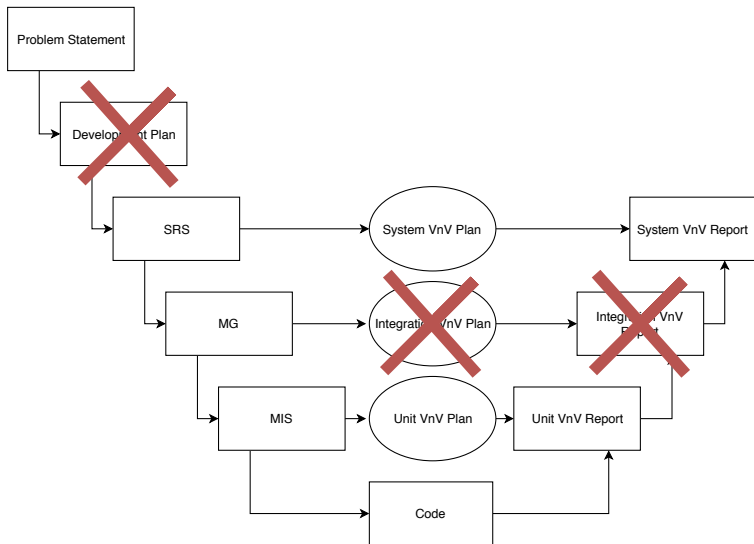
# Administrative Details: Presentation Schedule

- MG Present
  - ▶ **Wednesday: Karol, Malavika, Robert, Hanane**
  - ▶ **Friday: Brooks, Vajiheh, Olu, Jennifer**
- MIS Present
  - ▶ Wednesday: Malavika, Robert
  - ▶ Friday: Hanane, Jennifer
- Unit VnV Plan or Impl. Present
  - ▶ Wednesday: Brooks, Vajiheh
  - ▶ Friday: Olu, Karol

# Questions?

- Questions about Verification and Validation plan?

# Review of our “Faked” Rational Design Process



# SWHS MG Example

<https://github.com/smiths/swhs/tree/master/docs/Design/MG>



# What is Design?

- Your requirements document identifies “What,” now we begin to look at “How”
- Your system should meet both your functional and nonfunctional requirements
- There is no unique “optimal” design
  - ▶ Different goals will lead to different designs
  - ▶ There is a mix of art and science in design
  - ▶ Even with fully formal requirements specification there does not yet exist a systematic way to obtain a design
  - ▶ Favour art in some areas and favour science in others

# What is Design Continued?

- Provides structure to any artifact
- Decomposes system into parts, assigns responsibilities, ensures that parts fit together to achieve a global goal
- Design refers to
  - ▶ Activity
    - ▶ Bridge between requirements and implementation
    - ▶ Structure to an artifact
  - ▶ Result of the activity
    - ▶ System decomposition into modules (module guide)
    - ▶ Module interface specification (MIS)

# Why Decompose Into Modules?

- Separation of concerns
- Cannot understand all of the details
- All engineering fields use decomposition
- Modules will act as “work assignments”
- Decomposition needs to follow a systematic procedure (as for SRS)
- Need to ensure that modules when fit together achieve our global goals
- Document in a Software Design Document (Module Guide)

# Benefits of Modularity

- Shorter development time
- Improved verification
- Reduced maintenance costs
- Easier to understand
  - ▶ Small modules
  - ▶ An abstract interface
- Modules can be developed independently
- Modules can be tested independently
- Modules can be reused
- Software is easy to change, extend, maintain
- This requires identifying the anticipated changes in the design and in the requirements

# Two Important Goals for Decomposition

- Design for change (Parnas) [5, 6]
  - ▶ Designers tend to concentrate on current needs
  - ▶ Special effort needed to anticipate likely changes
  - ▶ Changes can be in the design or in the requirements
  - ▶ Too expensive to design for all changes, but should design for likely changes
- Product families (Parnas) [4, 7]
  - ▶ Think of the current system under design as a member of a program family
  - ▶ Analogous to product lines in other engineering disciplines
  - ▶ Example product families include automobiles, cell phones, etc.
  - ▶ Design the whole family as one system, not each individual family member separately

Use Design Principle of Information Hiding

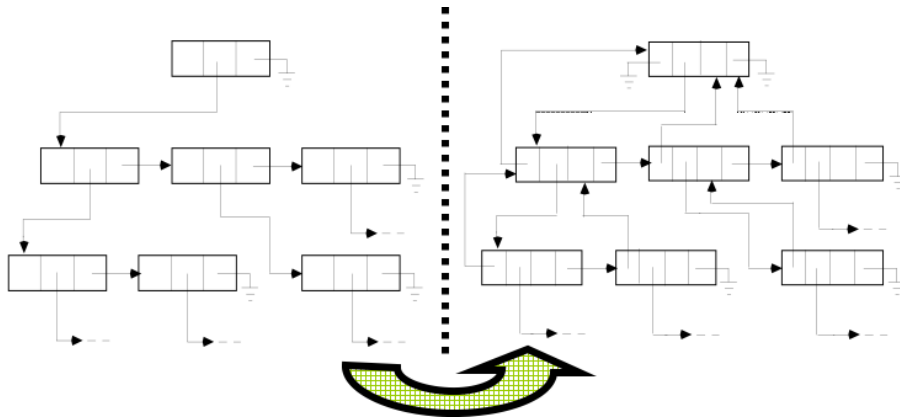
# Sample Likely Changes

What are some examples of likely changes for software?

# Sample Likely Changes [2]

- Algorithms – like replacing inefficient sorting algorithm with a more efficient one
- Change of data representation
  - ▶ From binary tree to threaded tree
  - ▶ Array implementation to a pointer implementation
  - ▶ Approx. 17% of maintenance costs attributed to data representation changes (Lientz and Swanson, 1980)
- Change of underlying abstract machine
  - ▶ New release of operating system
  - ▶ New optimizing compiler
  - ▶ New version of DBMS
  - ▶ etc.
- Change of peripheral devices

# Binary Tree to Threaded Tree





# Sample Likely Changes

- Change of “social” environment
  - ▶ Corresponds to requirements changes
  - ▶ New tax regime
  - ▶ EURO versus national currency in EU
  - ▶ New language for user interface
  - ▶ y2k
- Change due to development process (prototype transformed into product)

# Components of a Module

- A software module has two components
  1. An **interface** that enables the module's clients to use the service the module provides
  2. An **implementation** of the interface that provides the services offered by the module

# The Module Interface

- A module's interface can be viewed in various ways
  - ▶ As a **set of services**
  - ▶ As a **contract** between the module and its clients
  - ▶ As a **language** for using the module's services
- The interface is **exported** by the module and **imported** by the module's clients
- An interface describes the **data** and **procedures** that provide access to the services of the module

# The Module Implementation

- A module's implementation is an implementation of the module's interface
- The implementation is **hidden** from other modules
- The interface data and procedures are implemented together and may share data structures
- The implementation may utilize the services offered by other modules

# Information Hiding

- Made explicit by Parnas [5]
- Basis for design (that is modular decomposition (Module Guide))
- Implementation secrets are hidden to clients
- Secret can be changed freely if the change does not affect the interface
- Try to encapsulate changeable design decisions as implementation secrets within module implementations

# Examples of Modules [2]

- Record
  - ▶ Consists of only data
  - ▶ Has state but no behaviour
- Collection of related procedures (library)
  - ▶ Has behaviour but no state
  - ▶ Procedural abstractions
- Abstract object
  - ▶ Consists of data (**fields**) and procedures (**methods**)
  - ▶ Consists of a collection of **constructors**, **selectors**, and **mutators**
  - ▶ Has state and behaviour

# Examples of Modules Continued

- Abstract data type (ADT)
  - ▶ Consists of a collection of abstract objects and a collection of procedures that can be applied to them
  - ▶ Defines the set of possible values for the type and the associated procedures that manipulate instances of the type
  - ▶ Encapsulates the details of the implementation of the type
- Generic Modules
  - ▶ A single abstract description for a family of abstract objects or ADTs
  - ▶ Parameterized by type
  - ▶ Eliminates the need for writing similar specifications for modules that only differ in their type information
  - ▶ A generic module facilitates specification of a stack of integers, stack of strings, stack of stacks etc.

# Questions

- What relationships are there between modules?
- Are there desirable properties for these relations?



# Relationships Between Modules [2]

- 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

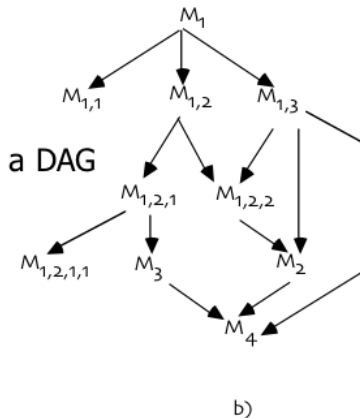
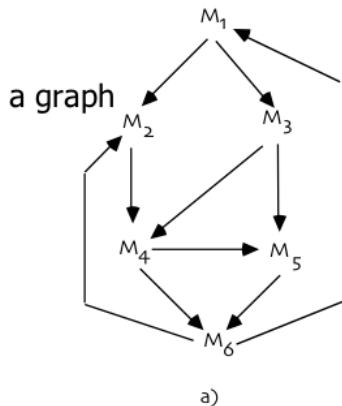
- Transitive closure  $r^+$  of  $r$

$M_i r^+ M_j$  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?

# Desirable Properties

- USES should be a hierarchy [6]
  - ▶ 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?

# 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?

# 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  $k + 1$

# Static Definition of Uses Relation

Your program has code like:

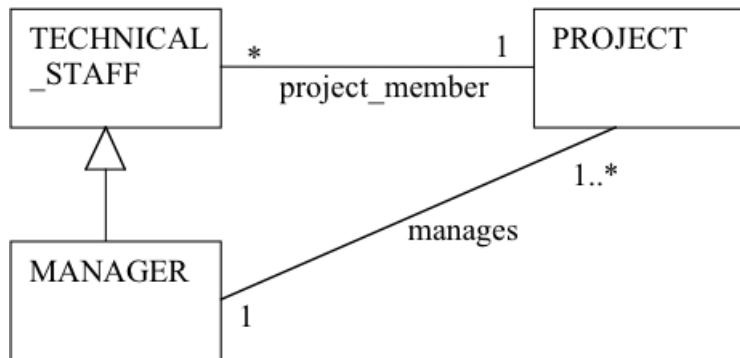
```
if cond then ServiceFromMod1 else ServiceFromMod2
```

This is the only place where each module is used. Does this mean the uses relation depends on the dynamic execution of the program?

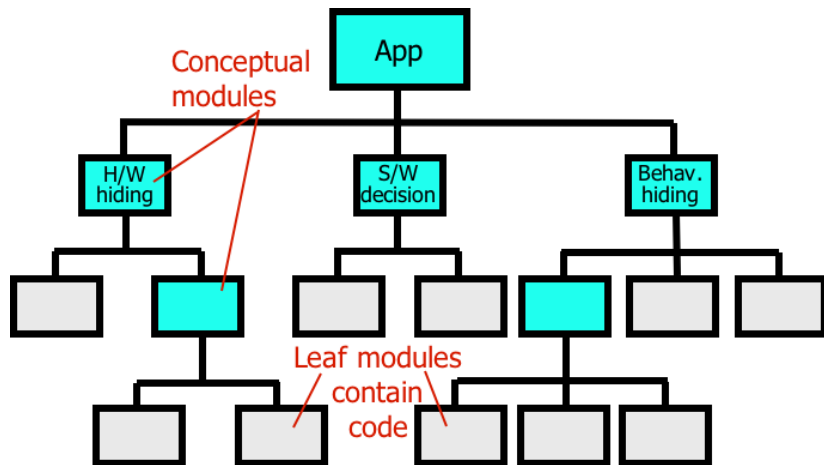


# Question about Association and DAG

Is the uses relation here a DAG?



# Module Decomposition (Parnas)



# Module Decomposition (Parnas)

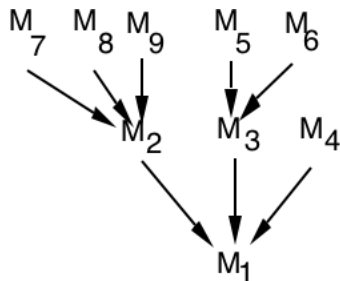
For the module decomposition on the previous slide:

- Does it show a Uses relation?
- Is it a DAG?
- Is it a tree?

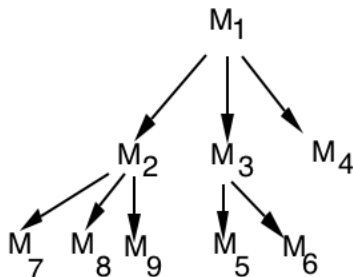
# IS\_COMPONENT\_OF

- 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  $M_i$

# A Graphical View



(IS\_COMPONENT\_OF)



(COMPRISES)

*They are a hierarchy*

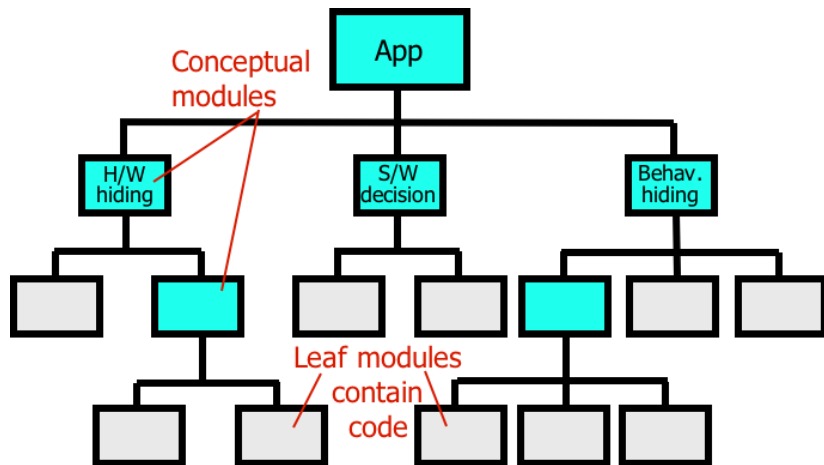
# Module Guide [8]

- 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
- Helps future maintainers find appropriate module
- Parnas proposed a Module Guide (MG) based on the decomposition module tree shown earlier
- Decomposition is usually three to five levels deep

# Three Top Conceptual Modules in an RDP MG

What are the three groups of modules in a typical information-hiding decomposition?

# Module Decomposition (Parnas)





# 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

# 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 (optional)

# RDP - MG

- Criteria for a good secret
  - ▶ One module one secret, especially for leaf modules (watch for “and”)
  - ▶ Secrets should often be nouns (data structure, algorithm, hardware, ...)
  - ▶ Secrets are often phrased as “How to ... ”

# Good Secret?

Is the following a good module secret: “The file format for the map and the rules for validating that the map satisfies the environmental constraints.”

# Typical Modules [3]

- What are the typical secrets for an input variable?
  - ▶ You have an input in the environment, how to get it into your system?
  - ▶ What format is the input data?
- What are the secrets for an output variable?
  - ▶ How to get an output from inside the system to the external environment?
  - ▶ How will the output be determined?
  - ▶ What format will the output have?
- What are the secrets for a state variable?
  - ▶ What rules are there governing the state transitions?
  - ▶ What data structures or algorithms are needed?

# Typical Modules [3]

- Input variables
  - ▶ Machine-hiding from hardware or OS service
  - ▶ Behaviour-hiding input format
- Output variables
  - ▶ Machine-hiding
  - ▶ Behaviour-hiding output format
  - ▶ Behaviour-hiding (calculation)
- State variables
  - ▶ Software decision hiding for data structure/algorithm
  - ▶ Behaviour-hiding state-drive
- Judgement is critical
- Often combine variables into the same module
- For non-embedded systems, machine hiding for input-output is often combined

# RDP - Views

- As well as the MG, the modular decomposition should be displayed using a variety of views
- An obvious one is the **Uses Hierarchy**
- The Uses Hierarchy is updated once the MIS for all modules is complete
- The Uses Hierarchy can be represented
  - ▶ Graphically (if it isn't too large and complex)
  - ▶ Using a binary matrix – **What would the binary matrix look like?**

# MG Template

- Table of contents
- Introduction
- Anticipated and unlikely changes
- Module hierarchy
- Connection between requirements and design
- Module decomposition
  - ▶ Hardware hiding modules
  - ▶ Behaviour hiding modules
  - ▶ Software decision hiding modules
- Traceability matrices
- Uses hierarchy between modules



# Traceability Matrices

- Traceability matrix help inspect the design
- Check for completeness, look at from a different viewpoint

<b>Req.</b>	<b>Modules</b>
R1	M1, M2, M3, M7
R2	M2, M3
...	...

<b>AC</b>	<b>Modules</b>
AC1	M1
AC2	M2
...	...

# Verification

- Well formed (consistent format/structure)
  - ▶ Follows template
  - ▶ Follows rules (one secret per module, nouns etc.)
- Feasible (implementable at reasonable cost)
  - ▶ Difficult to assess
  - ▶ Try sketches of MIS
- Flexible
  - ▶ Again try sketches of MIS
  - ▶ Thought experiment as if likely change has occurred
  - ▶ Low coupling
  - ▶ Encapsulate repetitive tasks
- May sometimes have to sacrifice information hiding

# Object Oriented Design Versus Modular Design

- OO-design and OO-languages are different
- OO-design
  - ▶ Classes and methods
  - ▶ Classes are like modules (state variables and access functions (methods))
  - ▶ An object is an instance of a class
  - ▶ Polymorphism
  - ▶ Inheritance - use carefully
- Implementation of modules using an OO-lang is natural

# References I



Ahmed H. ElSheikh, W. Spencer Smith, and Samir E. Chidiac.

Semi-formal design of reliable mesh generation systems.  
*Advances in Engineering Software*, 35(12):827–841, 2004.



Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli.  
*Fundamentals of Software Engineering*.

Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition,  
2003.



Daniel M. Hoffman and Paul A. Strooper.

*Software Design, Automated Testing, and Maintenance: A Practical Approach*.

International Thomson Computer Press, New York, NY,  
USA, 1995.

# References II



David Parnas.

On the design and development of program families.

*IEEE Transactions on Software Engineering*, SE-2(1):1–9, 1976.



David L. Parnas.

On the criteria to be used in decomposing systems into modules.

*Comm. ACM*, 15(2):1053–1058, December 1972.



David L. Parnas.

On a 'buzzword': Hierarchical structure.

In *IFIP Congress 74*, pages 336–339. North Holland Publishing Company, 1974.

# References III



David L. Parnas.

Designing software for ease of extension and contraction.

*IEEE Transactions on Software Engineering*, pages  
128–138, March 1979.



D.L. Parnas, P.C. Clement, and D. M. Weiss.

The modular structure of complex systems.

In *International Conference on Software Engineering*,  
pages 408–419, 1984.

# References IV



W. Spencer Smith and Lei Lai.

A new requirements template for scientific computing.

In J. Ralyté, P. Ågerfalk, and N. Kraiem, editors,  
*Proceedings of the First International Workshop on  
Situational Requirements Engineering Processes –  
Methods, Techniques and Tools to Support  
Situation-Specific Requirements Engineering Processes,  
SREP'05*, pages 107–121, Paris, France, 2005. In  
conjunction with 13th IEEE International Requirements  
Engineering Conference.

# References V



W. Spencer Smith, Lei Lai, and Ridha Khedri.

Requirements analysis for engineering computation: A systematic approach for improving software reliability.

*Reliable Computing, Special Issue on Reliable Engineering Computation*, 13(1):83–107, February 2007.