CAS 741 (Development of Scientific Computing Software)

Winter 2024

Modular Design

Dr. Spencer Smith

Faculty of Engineering, McMaster University

February 27, 2024

Modular Design

- Administrative details
- Feedback on SRS documents
- Questions?
- Overview of design
- Modular decomposition: advantages, guidelines etc.
- Module guide
- Module guide example

Administrative Details

- For the design documentation you do not have to use the MG and MIS templates

Administrative Details: Report Deadlines

MG + MIS	Week 09	Mar 15
Final Documentation	Week 13	Apr 12

- The written deliverables will be graded based on the repo contents as of 11:59 pm of the due date
- If you need an extension for a **written** doc, please ask
- When ready, assign issues to your primary and secondary reviewers
- GitHub issues due two days after assignment deadlines
- From Drasil Code onward, Drasil projects no longer need to maintain traditional SRS

Administrative Details: Presentations

POC Demo	Week 07	Week of Feb 26
MG + MIS	Week 09	Week of Mar 11
MG + MIS	Week 09	Week of Mar 11
Unit VnV/Implement	Week 12	Week of Apr 3

- Specific schedule depends on final class registration
- Informal presentations with the goal of improving everyone's written deliverables
- Domain experts and secondary reviewers (and others) will ask questions

Presentation Schedule

- Syst V&V Plan Present (L11, L12) (20 min)
- Proof of Concept Demonstrations (L14) (20 min)
  - **Mar 1: Cynthia, Valerie, Waqar, Yi-Leng**
- MG+MIS Present (L17, L18) (20 minutes)
  - Mar 12: Nada, Morteza, Kim Ying, Atiyeh
  - Mar 15: Fatemeh, Yiding, Tanya, Volunteer?

Presentation Sched Cont'd

- Implementation Present (L22–L25) (15 min each)
  - Mar 29: Fatemeh, Waqar, Al, Tanya, Atiyeh, Yi-Leng
  - Apr 2: Nada, Phil, Xinyu, Fasil, Seyed Ali, Kim Ying
  - Apr 5: Gaofeng, Morteza, Valerie, Hunter, Cynthia, Adrian
  - Apr 9: Yiding

Presentation Schedule

- 3 presentations each
  - SRS everyone
  - VnV and POC subset of class
  - Design subset of class
  - Implementation everyone
- If you will miss a presentation, please trade with someone
- Implementation presentation could be used to run a code review, or code walkthrough

Feedback on SRS

- SRS checklist
  - Follow the template
  - Filename for SRS from template (new)
  - All symbols are in the table of contents
  - Project is given a name
  - Characteristics with courses and level (clarified)
  - Goal statements are functional (new)
  - Each assumption is used at least once
  - All chunks are used at least once (new)
  - IMs are referenced by requirements

Feedback on SRS

- Writing checklist
  - Opening and closing “quotes”
  - Periods that do not end sentences are followed by only one space
  - Long names in math mode use either mathit or text
  - Document is spell checked
  - Symbols formatted the same way in every context
- Types information very helpful

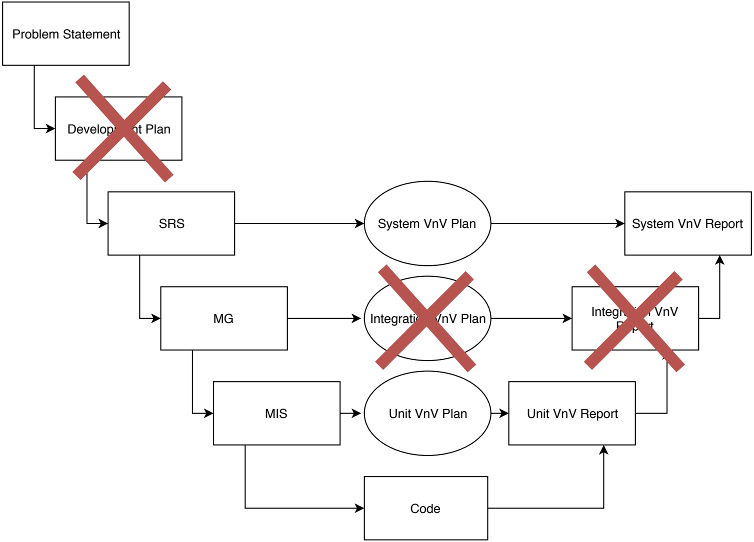
SRS + VnV Issues — (Mostly) Doing a great job!

- Be sure to do your GitHub reviews!
- Close issues when they are resolved
  - Explain why closing the issue
    - Maybe you will just address the issue in a comment
    - Maybe the issue will lead to repo changes
  - Include the commit hash (you just need the number)
  - Small, well-defined, commits
  - Link to other issues using hash symbol
- Take and give feedback in the collegial spirit, use emojis as appropriate

Questions?

- Questions?

Review of our "Faked" Rational Design Process



Our Goals for Design Overview

- Handle complexity and size by decomposing a complex system into parts (modules, units, objects, ...)
- Decomposition should be based on a principle – we'll emphasize information hiding
- Uses relation between parts should be a hierarchy
- Document the interface for all modules
  - Access programs (methods)
  - Inputs and outputs and their type
  - State variables (instance variables)
  - Syntax and semantics
  - Environment variables
- Use abstraction to make the specification more general, and easier
- Formal specification where possible
- Input-Calculate-Output pattern

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) [4, 5]
  - 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) [3, 6]
  - 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

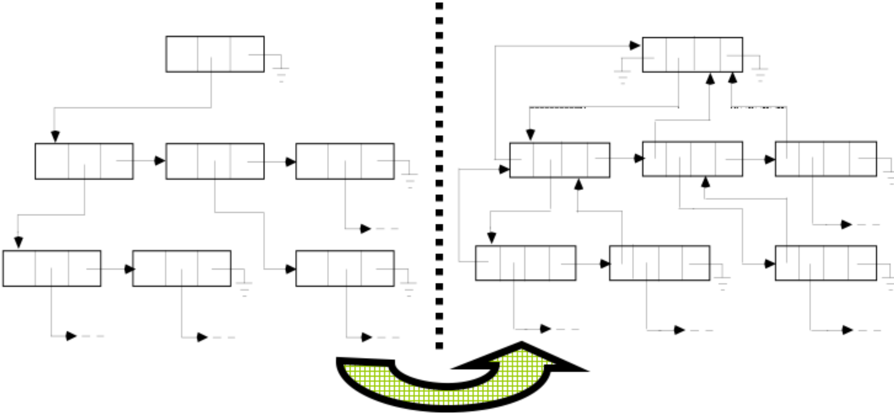
Sample Likely Changes

What are some examples of likely changes for software?

Sample Likely Changes [1]

- 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 modules 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 [4]
- 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

Questions

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

Relationships Between Modules [1]

- Let S be a set of modules

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

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

Relations

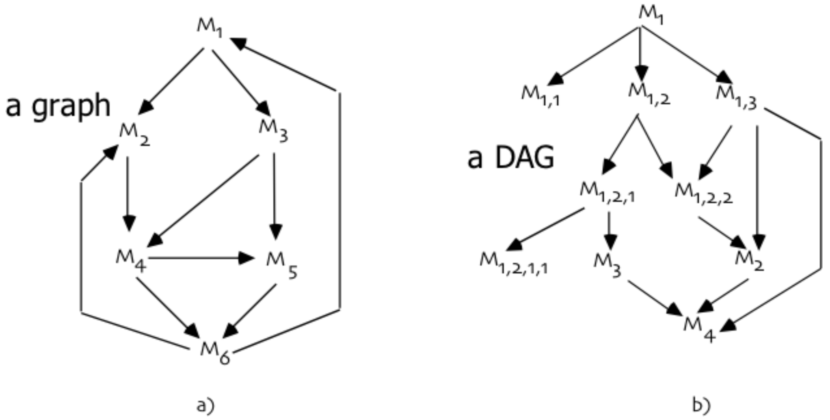
- Transitive closure r^+ of r

  $M_i r^+ M_j$ 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 [5]
  - 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

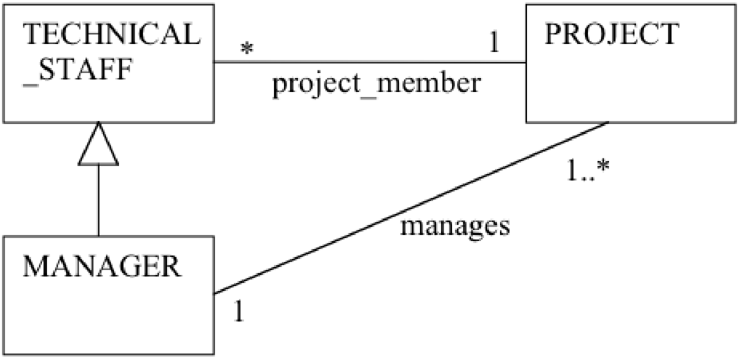
Your program has code like:

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

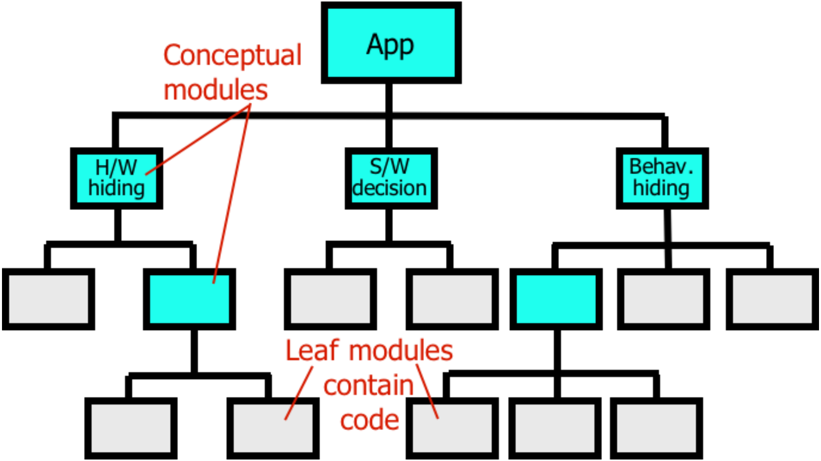
This is the only place where each module is used. Does this mean the uses relation depends on the dynamic execution of the program?

Question about Association and DAG

Is the uses relation here a DAG?



Module Decomposition (Parnas)



Module Decomposition (Parnas)

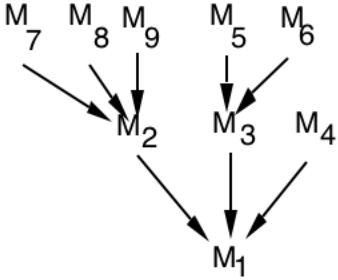
For the module decomposition on the previous slide:

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

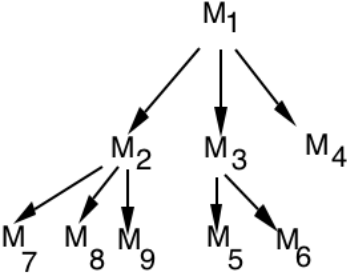
IS_COMPONENT_OF

- 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

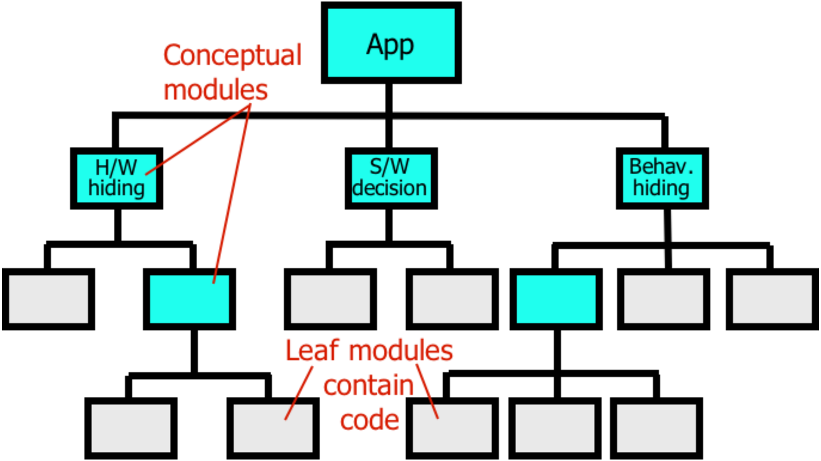
Module Guide [7]

- 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 [2]

- 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 [2]

- 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

Req.	**Modules**
R1	M1, M2, M3, M7
R2	M2, M3
...	...

AC	**Modules**
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

Examples of Modules [1]

- 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.

Getting Started

1. Find a similar example to your problem and use that as a starting point
2. Draft module names and secrets
3. For each module sketch out:
    - Classify module type (record, library, abstract object, abstract data type, generic ADT)
    - Access program syntax
    - State variables (if applicable)
4. Iterate on design

Appendix: Information Hiding Examples

- Hoffman and Strooper (1995) textbook on software development: The running example is of a symbol table. A very complete example. There is a complete chapter on the module guide in the text. It is well explained there.
- Parnas Et Al (1984) "The Module Structure of Complex Systems" : This example is right back to the source. The example focuses on the A7E military fighter jet.
- Parnas (1979) "Designing Software For Ease of Extension and Contraction"
- von Mohrenschildt (2005) "The Maze Tracing Robot A Sample Specification": This is a small and complete example.

Appendix Cont'd: Information Hiding Examples

- Jegatheesan and MacLachlan (2018), Module Guide for Solar Water Heating Systems Incorporating Phase Change Material
- Liu (2020) Module Guide for Radio Signal Strength Calculator
- Key points
  - One module, one secret
  - Secrets are often nouns (data structure, algorithm, hardware, etc.)
  - Secrets are sometimes phrased with "How to ..."
  - Secrets ideally will have a one to one mapping with the anticipated changes for the software

References I

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.

David Parnas.
On the design and development of program families.
IEEE Transactions on Software Engineering, SE-2(1):1–9, 1976.

References II

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.

David L. Parnas.
Designing software for ease of extension and contraction.
IEEE Transactions on Software Engineering, SE-5(2):128–138, March 1979.

References III

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.