SFWRENG / MECHTRON 3K04 Software Development

This Week: High-Level SE

McMaster University | Engineering
Computing & Software

September 29th 2025

Dr. Angela Zavaleta Bernuy

Brighter World

High-Level Software Engineering Concepts

- How to structure the software so the requirements can be implemented safely, clearly, and in a way regulators will accept.
- We are going to cover four main topics:

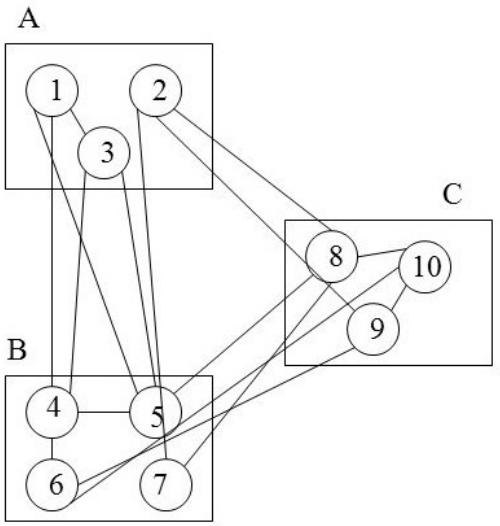
1. Relational Hierarchies
2. Cohesion and Coupling
3. Module Design
4. Module Documentation

1. Relational Hierarchies: The USES Relation

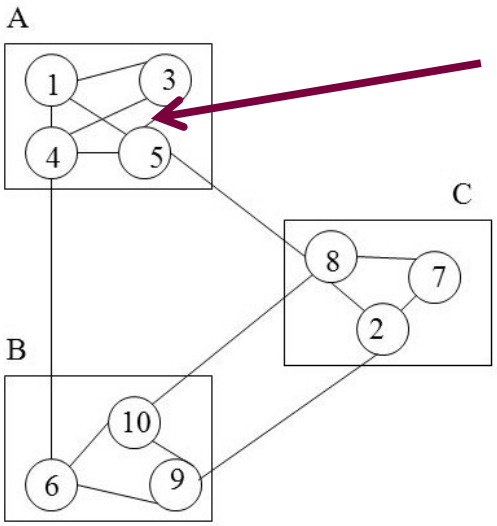
- USES should be a hierarchy
  - Clear direction of dependence
- Hierarchies make software easier to understand
  - We can proceed from leaf nodes (who do not use others) upwards
- They make software easier to build
  - You can implement the leaf modules first, since nothing depends on them
- They make software easier to test
  - Testing can also start bottom-up
- Not the only form of hierarchy in software…

2. Cohesion and Coupling

We can evaluate ***coupling*** from an external view ("uses"), but we need to see internal details to judge ***cohesion***.



Bad modularization:
low cohesion, high coupling



Usually an indication of reasonable cohesion

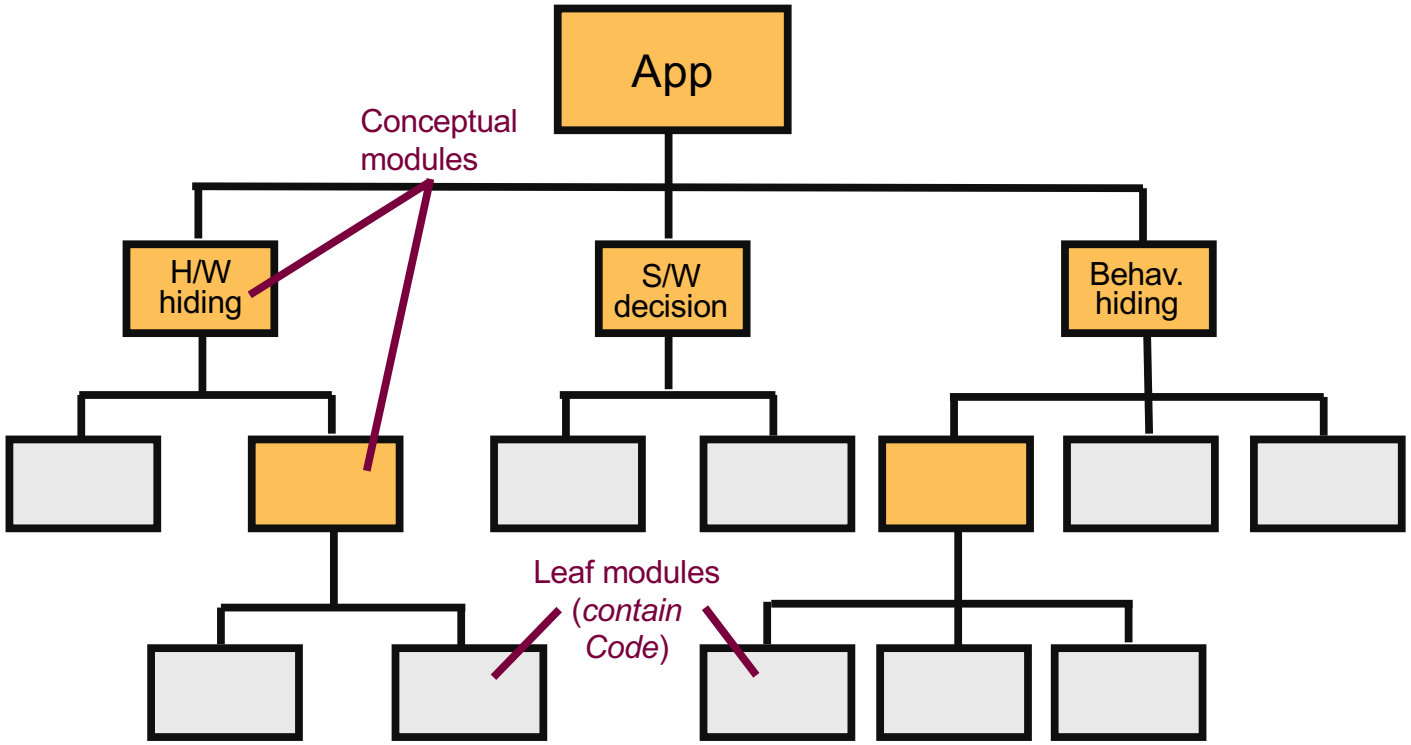
Good modularization:
high cohesion, low coupling

https://devopedia.org/cohesion-vs-coupling

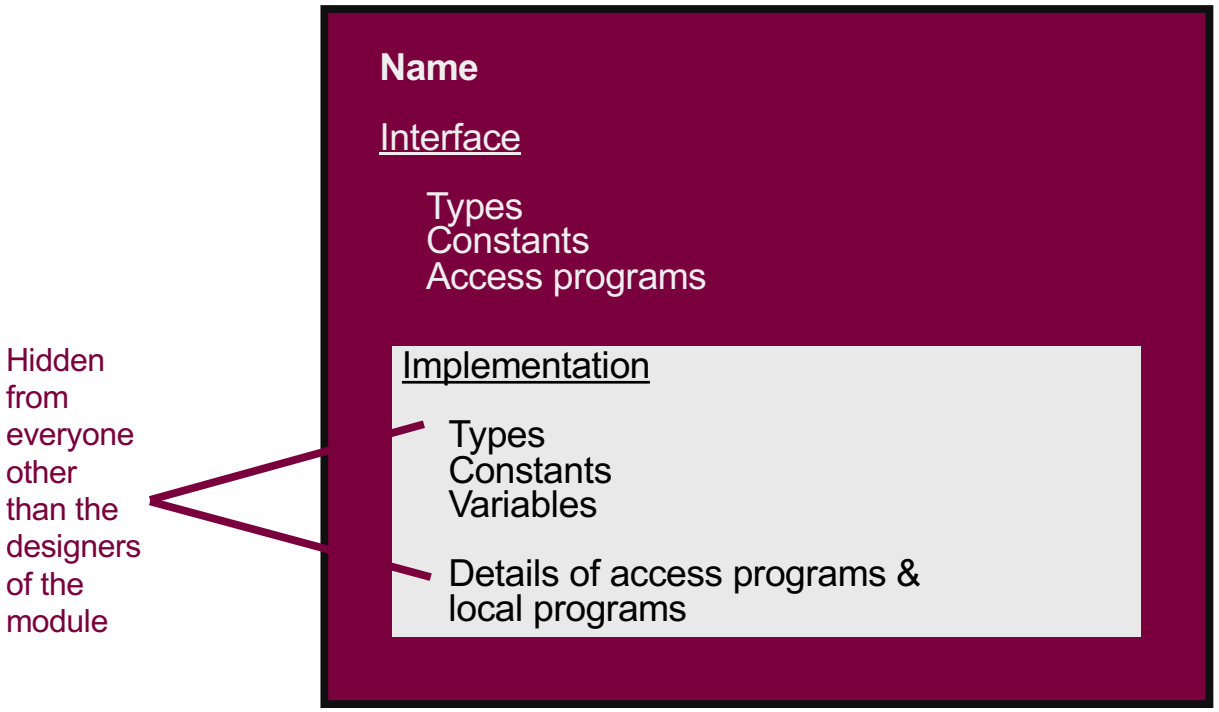
3. Module Design

- The process of defining clear boundaries, responsibilities, and interfaces for software modules
  - **Goal**: Make the system easier to develop, test, maintain, and to be reviewed by a regulator!
- Why does it matter?
  - Makes the code easier to change without breaking the whole system
  - Critical in safety-critical systems, where regulators require clear design documentation
- Key principles:
  - **Information hiding:** modules expose only what others need (interface) and keep details private (implementation)
  - **Separation of concerns:** each module is responsible for one well-defined function
  - **Independence:** modules should minimize unnecessary interdependencies
- How?

3. Module Design: Module Decomposition (Parnas)



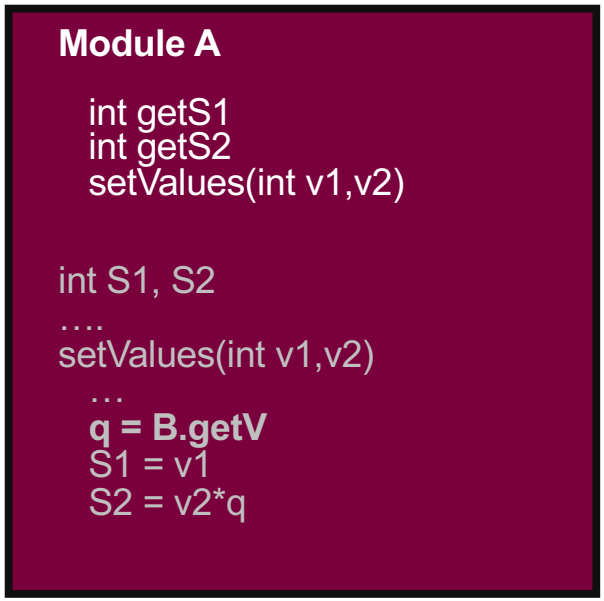
3. Module Design: Module Fundamentals

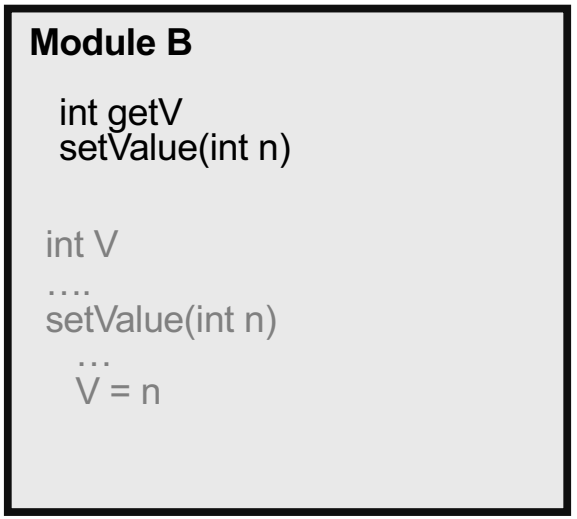


3. Module Design: Interface vs. Implementation

- To understand the nature of the USES relationship, we need to know what a used module *exports* through its *interface*
- The client *imports* the resources that are exported by its servers
- Modules *implement* the exported resources
- Implementation is *hidden* to clients
- Key design principle: clear distinction between interface and implementation:
  - Supports separation of concerns:
    - Clients care about resources exported from servers and servers care about implementation
  - The interface acts as a contract between a module and its clients

3. Module Design: Interface vs. Implementation





Module A uses Module B via de USES relation

A depends only on B's interface, not its implementation.

3. Module Design: Module Fundamentals

The **interface** is like the tip of an iceberg

The **implementation** is like the bulk of an iceberg

Need to **document the *effects* of interface functions**, not just their inputs/outputs, but also what they change

3. Module Design: Interface vs. Implementation

- To describe the interface, we need to describe the relationship between:
  - outputs (responses) and
  - inputs (stimuli)
- There are many different ways of doing this, ranging from natural language to formal notation
  - We will discuss this later!
  - Some of which we have discussed already…
  - Tables, flowcharts, Gherkin's, etc.

3. Module Design: Information Hiding

- Core design principle that is the basis for back-end design (i.e. module decomposition)
- *Encapsulate changeable (requirements) and design decisions as implementation secrets within module implementations*
- That is:
  - Implementation details are hidden to clients
  - These details can be changed freely if the change does not affect the interface (or even the entire system)
    - Hardware: Sensors, input/output devices, actuators, etc.
    - Behaviour: Number of items, timing, etc.
    - Design decisions: Data structures, algorithm, platforms, etc.
- Changes may occur in the requirements or/and in the design decisions stage

3. Module Design: Information Hiding

- How can we apply the information hiding principle?
  - Make a list of anticipated changes while preparing requirements
  - Make a list of changes that are not likely while preparing requirements
  - For each likely change, decide on the “secret” you want to hide and document it
  - As you finalize requirements and start the design, update the above lists
  - *Apply information hiding principle in developing the design*
  - At the end of design, make sure list is accurate
  - At the end of the design, “test” how well you have followed the information hiding
    - Design review test!

3. Module Design: Information Hiding

- Design review test:
  - Did you successfully **encapsulate** the parts of the system that are likely to change (e.g., hardware details, algorithms, data formats)?
  - If one of those changes happened tomorrow, would it **only require changes inside one module**, without ripple effects across the system?
  - Did you keep the interfaces clean, exposing **only what clients actually need**, and **hiding implementation** details?
  - Are the secrets you chose (the design decisions likely to change) **really hidden**, or did some of them leak out into other parts of system?
- To make this work, we need a clear and careful interface.

3. Module Design: Interface Design

- The interface should not reveal what we expect may change later
- It should act as a firewall preventing access to hidden parts
- The interface should not expose internal data structures
- The interface is what connects modules together
    - And it is where mistakes in design show up first!

3. Module Design: Prototyping

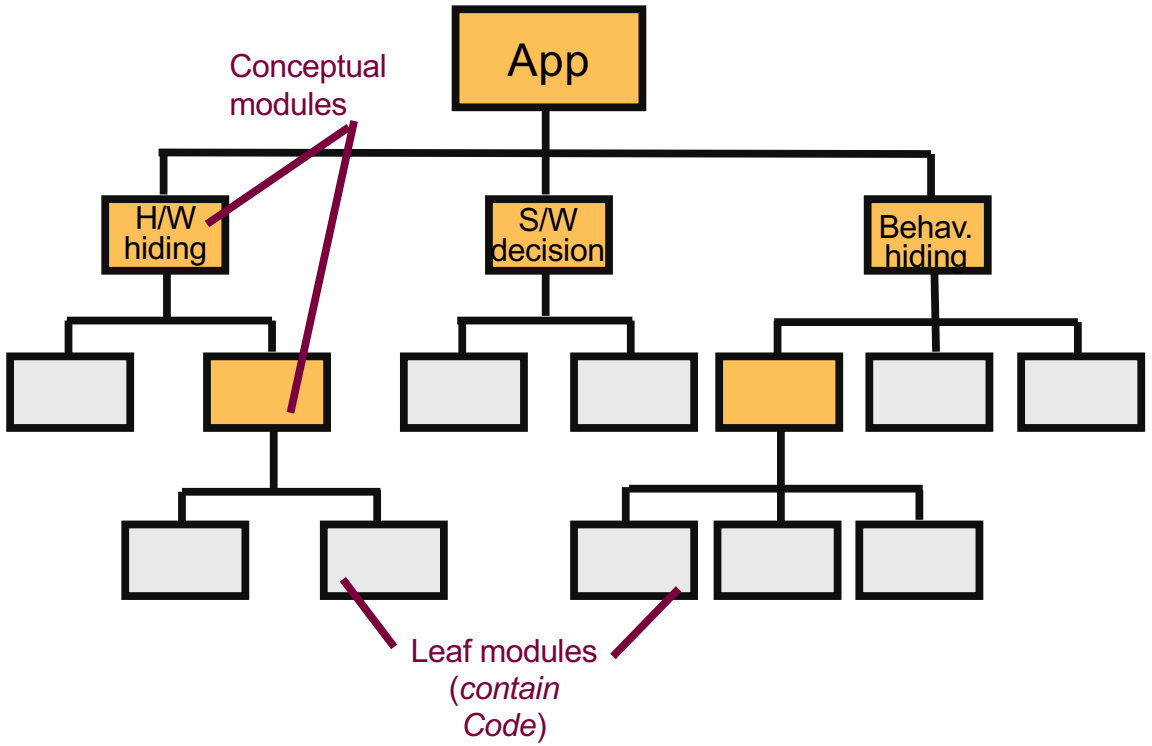
- Once an interface is defined, implementation can start
  - The first version will be done quickly, and inefficiently
  - Then, progressively turned into the final version
- The initial version acts as a prototype that evolves into the final product
- To make sure our designs are precise and understandable by others, we rely on design frameworks and notations!

3. Module Design: Design Frameworks/Notations

- Notations allow designs to be described precisely
- They can be be textual or graphical:
  - E.g., Parnas' Rational Design Process (RDP), Unified Modeling Language (UML), Z Notation, etc.
- We will expand one of these: Parnas' Rational Design Process (RDP)
  - It has 3 major components:
    - Module Guide (MG)
    - Module Interface Specification (MIS), and
    - Module Internal Design (MID)

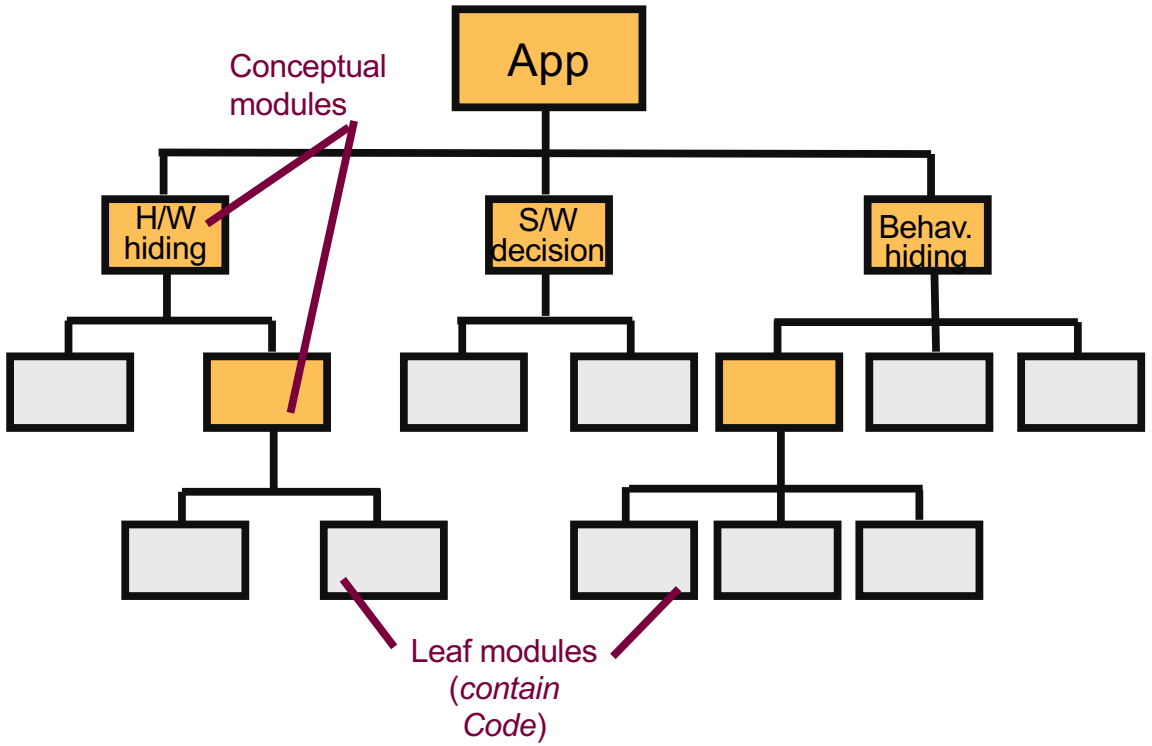
3. Module Design: RDP – MG

- When decomposing the system into modules (as per earlier discussions), we need to document the module decomposition so that the developers and other readers can understand the decomposition
- Parnas proposed a Module Guide (MG) based on the decomposition module tree shown earlier



3. Module Design: RDP – MG

- The MG consists of a table that documents each module's responsibility and *secret details*
- Conceptual modules will have broader responsibilities and *secret details*
- The leaf modules that represent code will contain much more precise responsibilities and *secret details*



3. Module Design: RDP – MG (Example)

#	Name	Responsibility	Secret Details
…	…	…	…
1.3	A/Is	Interface with the A/I hardware to produce software variables that represent the field data	Hardware representation of data & how to get data from hardware
1.3.1	FlowSensors	Software representation of flow measurements	Relationship between physical flow and software representation
…	…	…	…

Which one of these modules is conceptual and which one is a leaf?

3. Module Design: RDP – MG (Example)

ID	NAME	RESPONSIBILITY	SECRET
1	App	Entire system	All secrets
1.1	H/W hiding	All hardware related behaviour	List of h/w secrets
1.1.1	Analog input class 1	Behaviour for all analog I/O	Secret for class 1
1.1.2	Digital I/O	Behaviour for all digital I/O	List of secrets
1.1.2.1	Digital inputs class 2	Behaviour for all digital inputs	Secret for class 2
1.1.2.2	Digital outputs class 3	Behaviour for all digital outputs	Secret for class 3
1.2	S/W decisions	All behaviour for s/w decisions	List of secrets
1.2.1	S/W decision class 4	Behaviour for class 4	Secret for class 4
1.2.2	S/W decision class 5	Behaviour for class 5	Secret for class 5
1.3	Behaviour hiding	Most of the normal behaviour	List of secrets
1.3.1	…	…	…
1.3.2	Behaviour class 6	Behaviour for that class	Secret for class 6
1.3.3	Behaviour class 7	Behaviour for that class	Secret for class 7
1.3.1.1	Behaviour class 8	Behaviour for that class	Secret for class 8
1.3.1.2	Behaviour class 9	Behaviour for that class	Secret for class 9
1.3.1.3	Behaviour class 10	Behaviour for that class	Secret for class 10

3. Module Design: 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 Specifications (MIS)
- We would aim to use some form of mathematical approach to specify the MIS behaviour, as black-box as possible
- It describes what the interface to your class is, both syntax and semantics
  - Syntax: list of items with parameters
  - Semantics: description of behaviour of each method
    - For each method, describe the value of the outputs as they depend on the inputs

3. Module Design: RDP – Views

- As well as the MG, the modular decomposition should be displayed using a variety of views.
- An obvious one is a “Uses Hierarchy”, which can be formed once the MIS for all modules is complete.
- It is not always obvious how to achieve this:
  - For complex systems, graphical representations are not successful without good tool support

3. Module Design: RDP – MID

- Finally, we can document the Module Internal Design (MID) for each module
- This provides the implementation of the module, i.e. how we deliver on what we promised in the MIS
- The more complete this is, the better.
- But, it represents a requirement for the coder, so it is better represented at a higher level of abstraction that the code.

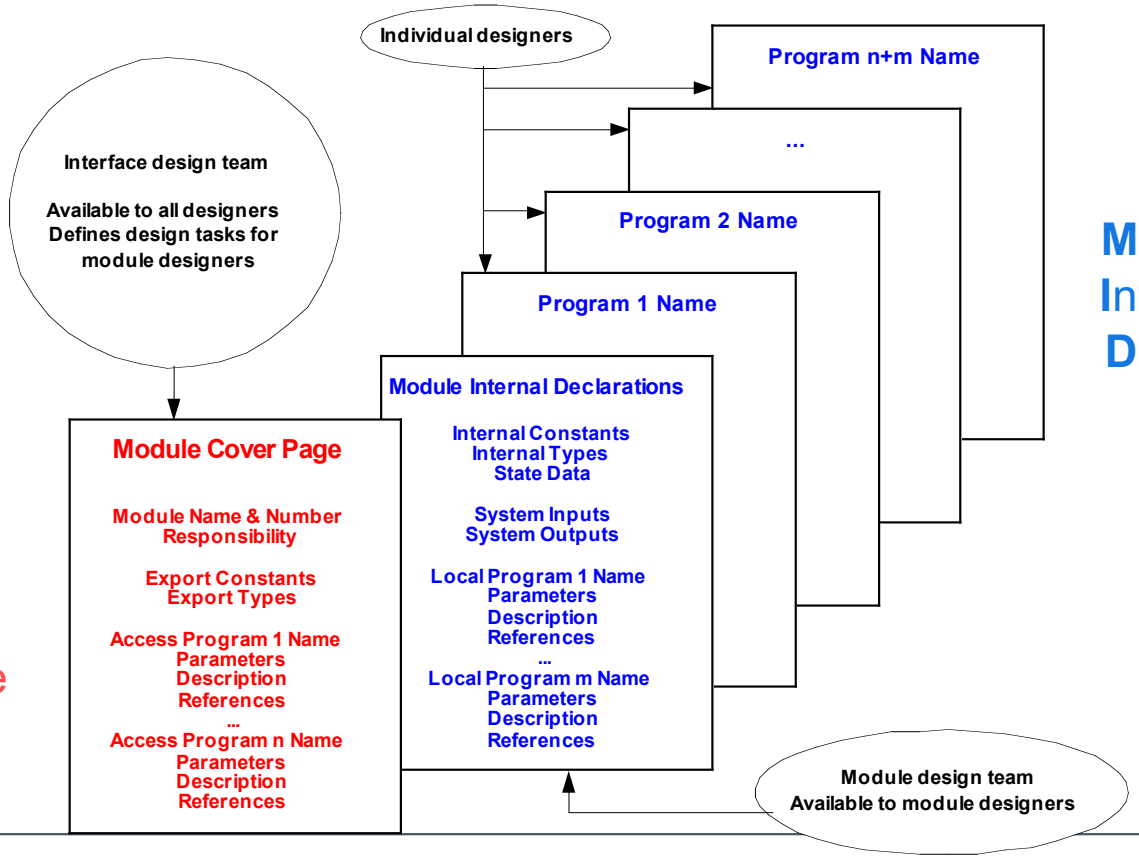
3. Module Design: RDP – Module Documents

- How do we capture and communicate those design decisions?
- Module Guide: the high-level map of the system; shows module decomposition
- Uses hierarchy (produced after all Module Interface Specification)
  - Helps us understand dependencies
- Module Design for each module: Captured in two detailed documents
  - Module Interface Specification: Describes what the module exports (interface)
  - Module Internal Design: Describes the hidden implementation details

4. Module Documentation: RDP

Note: in OOP "modules" are "classes" and "access programs" are "methods"

We will continue to compare modules and classes, but this picture would not change for classes

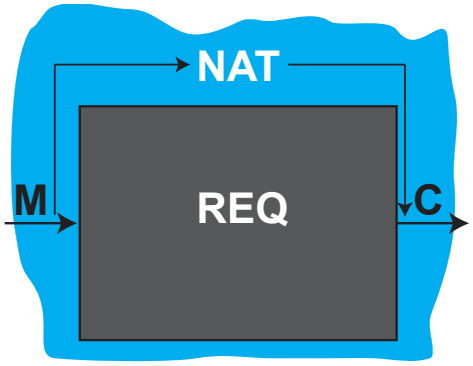


How can we evaluate our Software Design?

How to Evaluate Our Software Design

- **Completion**
  - Every software requirement is implemented in the design
  - Functions are complete on their input domains
- **Correctness**
  - Every software requirement is *correctly* implemented (within tolerance)
- **Structure**
  - USES relationship is a hierarchy
  - Modules (classes) exhibit low coupling and high cohesion
  - Interfaces do not include state variables
  - Interfaces do not disclose secrets
  - MIS and MID are adequate requirements for the implementation
  - The MID of a USED module is not required to design the internal of the module that USES it
- **Maintenance**
  - A list of likely requirements and design changes is provided
  - Secrets are documented in the MG
  - Each secret is encapsulated in a module

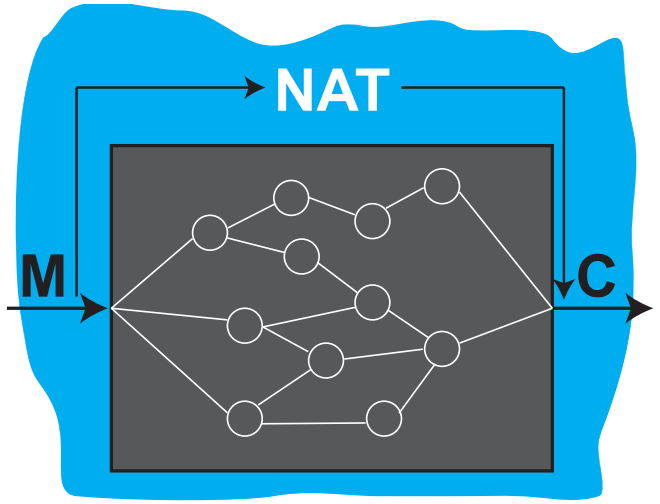
System Requirements: Review



We need to specify REQ in terms of M and C without any intermediate functions.
This would give us a black-box description of the required behaviour.

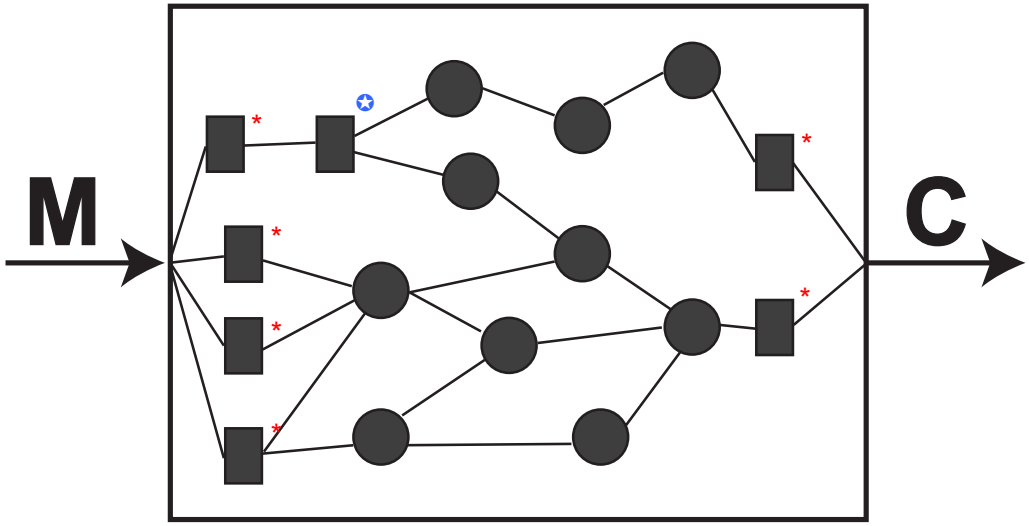
$$C \in REQ(M)$$

This is not always possible.
The behaviour is too complex to describe it in a single function/relation.
We use intermediate functions to describe behaviours that still make sense to the domain expert.
The overall behaviour is given by the composition of the functions.



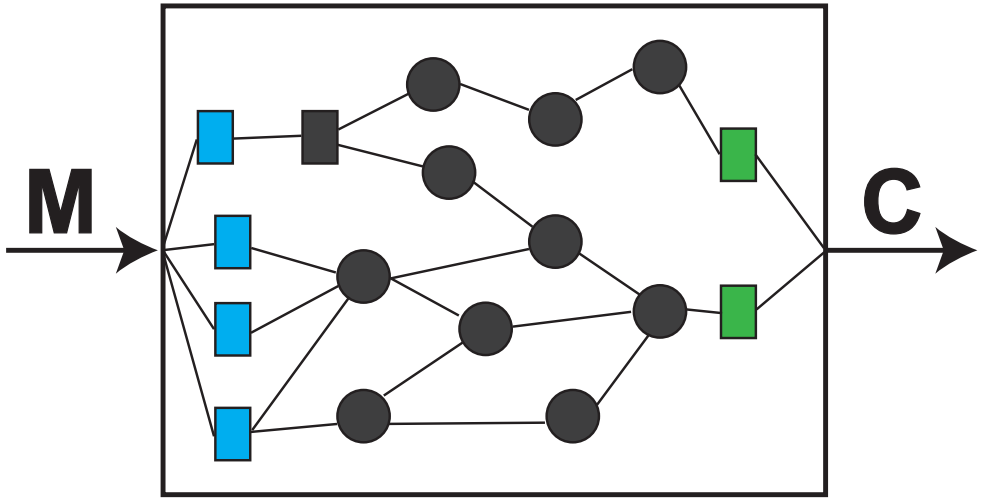
System Design: Overview

- Once we have the behaviour specified in the system requirements, we can partition the behaviour into hardware and software functions.
- In most cases, the hardware components will represent sensors and actuators and associated hardware represented by IN and OUT in the 4 variable-model*.
- In a few cases, a hardware component may perform actual functional behaviour ✪.
- The software functions in the system design can be used to specify the software requirements, if they are described adequately in the system design.



System Design: Overview

- Once we have the behaviour specified in the system requirements, we can partition the behaviour into hardware and software functions.
- In most cases, the hardware components will represent sensors and actuators and associated hardware represented by IN and OUT in the 4 variable-model.
- The hardware functions that represent IN and OUT must be described by M-I and O-C mappings.



Hardware described by M-I mappings | Hardware functions | Software functions | Hardware described by O-C mappings

Traditional Modules vs. Classes

- Older design paradigms and coding languages used coding units which were *procedural abstractions*
  - They encapsulated a series of program steps so that they could be used over-and-over.
- Sometimes those steps returned values calculated in that unit much like a mathematical function, and sometimes they did not.
  - That is, it may be a manipulation of global variables.
- In older languages like FORTRAN they were called *functions* (returned a value associated with the name of the function) or *procedures* (did not return a value, or returned values associated with variables in the parameter list)

Traditional Modules vs. Classes: Example

- You may be familiar with the C language
- Here are some examples in C:
  - `int simple(int n)` returns an integer value associated with the name simple, given an integer value for n
  - `void simplest()` returns nothing, just performs steps inside simplest
  - `int add1(int a, int b)` returns an integer using input integers a and b
  - `void add2(int a, int b, int *c)` returns an integer value called c using integer inputs a and b

Traditional Modules vs. Classes

- Moving on from procedural abstractions we arrived at *modules*
- This was a huge move forward to providing structures that we can use to put together procedural abstractions together with data that they operate on, all in one package
  - These are the modules we described earlier!
- These modules are excellent for implementing information hiding.
- However, they have one (sometimes serious) limitation: you can only instantiate a module once
  - So, if we make a module that implements an integer stack, called INTSTACK, we can only have one version of it
  - If our program needs 2 stacks, we cannot “create another one”, we would need to duplicate the module’s code and rename it (INSTACK2)
- *Object Oriented Design* and *Object Oriented Programming (OOP)* paradigms solve this problem

Abstract Data Types (ADTs)

- ADTs allow us to define a new type of data, along with operations that can be performed on it, while still hiding the details of how that data is actually represented.
- Example: A Stack ADT