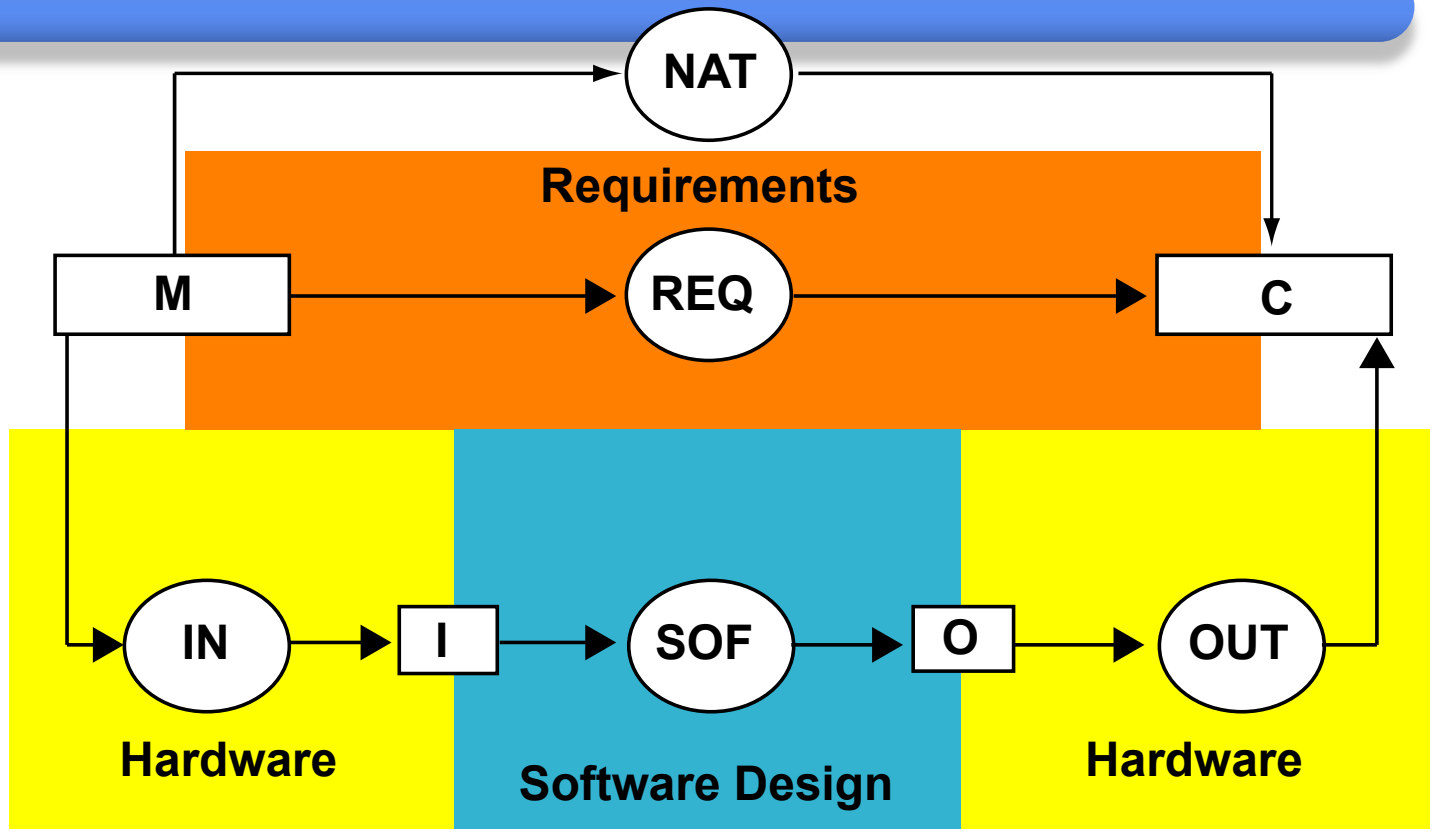
Comments on Design for Capstone

Alan Wassyng

4 Variable Model

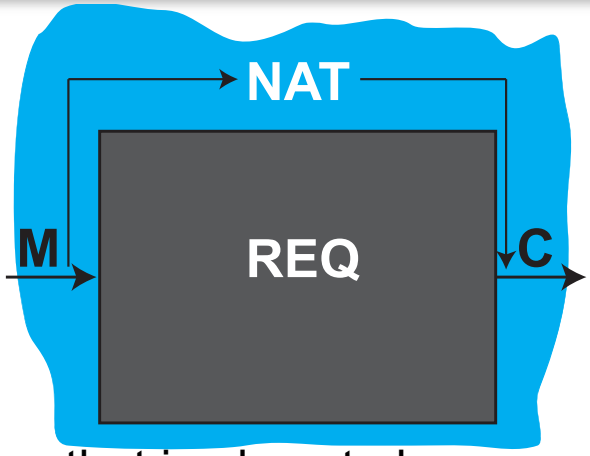
The 4 variable model relates the requirements to the design.

The requirements are **described** by REQ which is a relation between the *monitored variables*, M, and the *controlled variables*, C. NAT represents the constraints imposed by the environment.



The software design is represented by SOF that maps I, the *input variables*, to O, the *output variables*. I is generated by the hardware peripherals from M, and C is generated by the hardware from O. When we describe the system design, we need to include IN and OUT, the mappings from M to I and from C to O, for the chosen hardware.

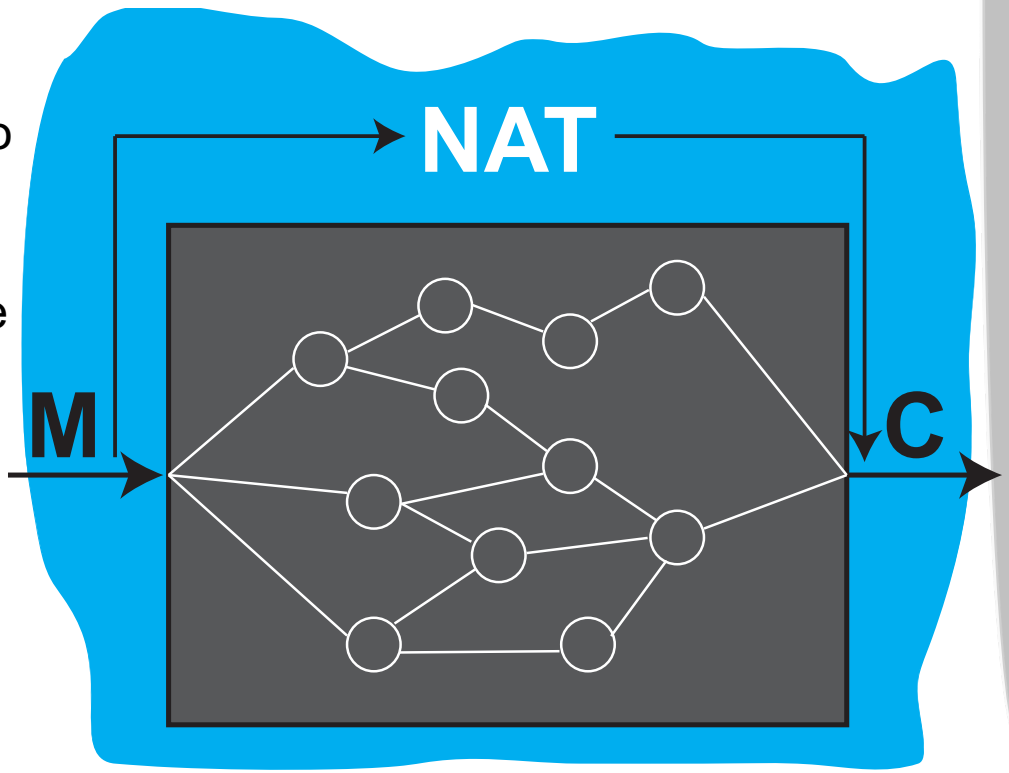
System Requirements - Overview



What we want to do is specify REQ in terms of C and M without any intermediate functions. This would give us a black-box description of the required behaviour.

$$C \in REQ(M)$$

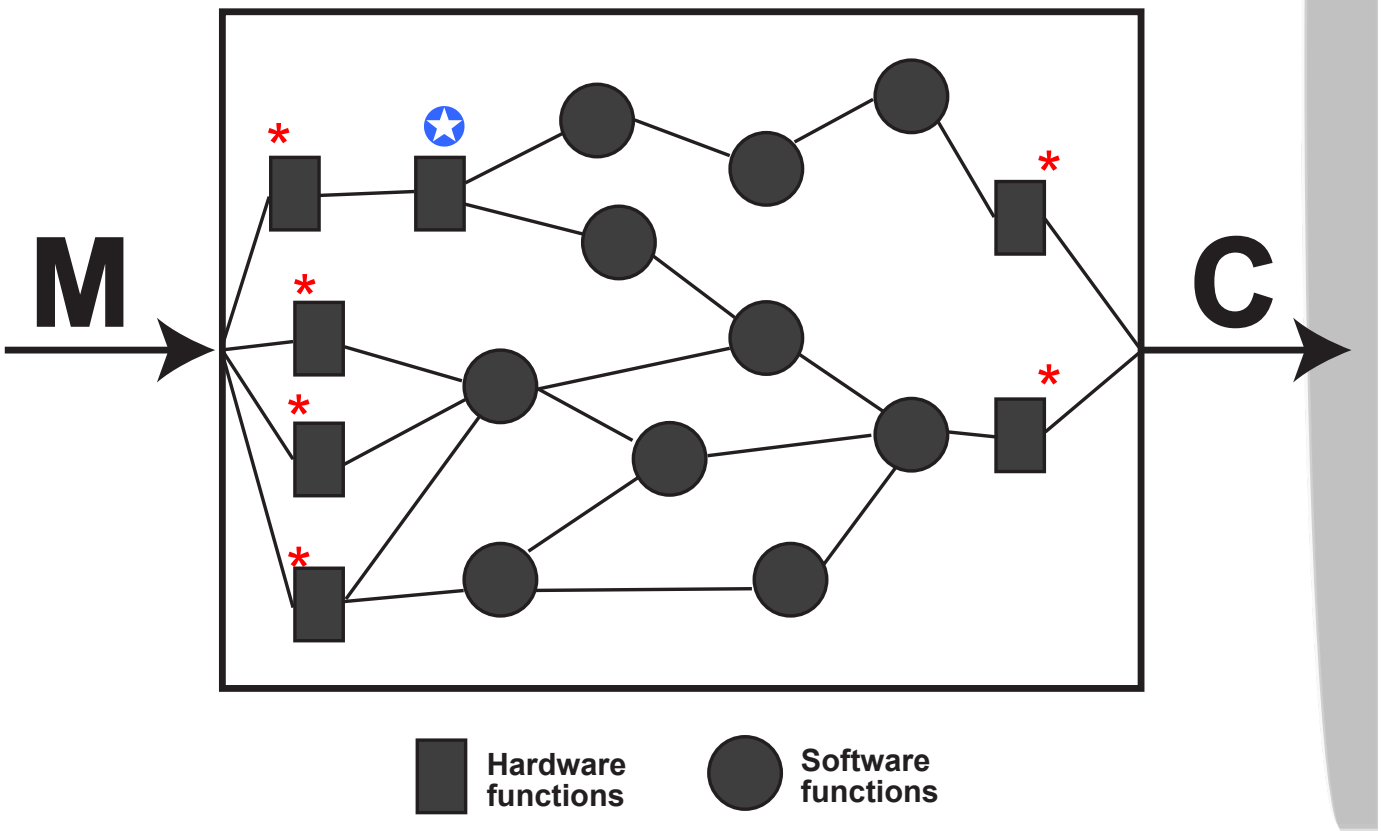
However, that is almost always not possible. The behaviour is too complex to describe it in a single function/relation. So, we use intermediate functions to describe behaviours that still make sense to the domain expert, and the overall behaviour is given by the composition of those functions. Note that in describing the behaviour of the functions, we can use a mathematical notation or natural language.



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*.

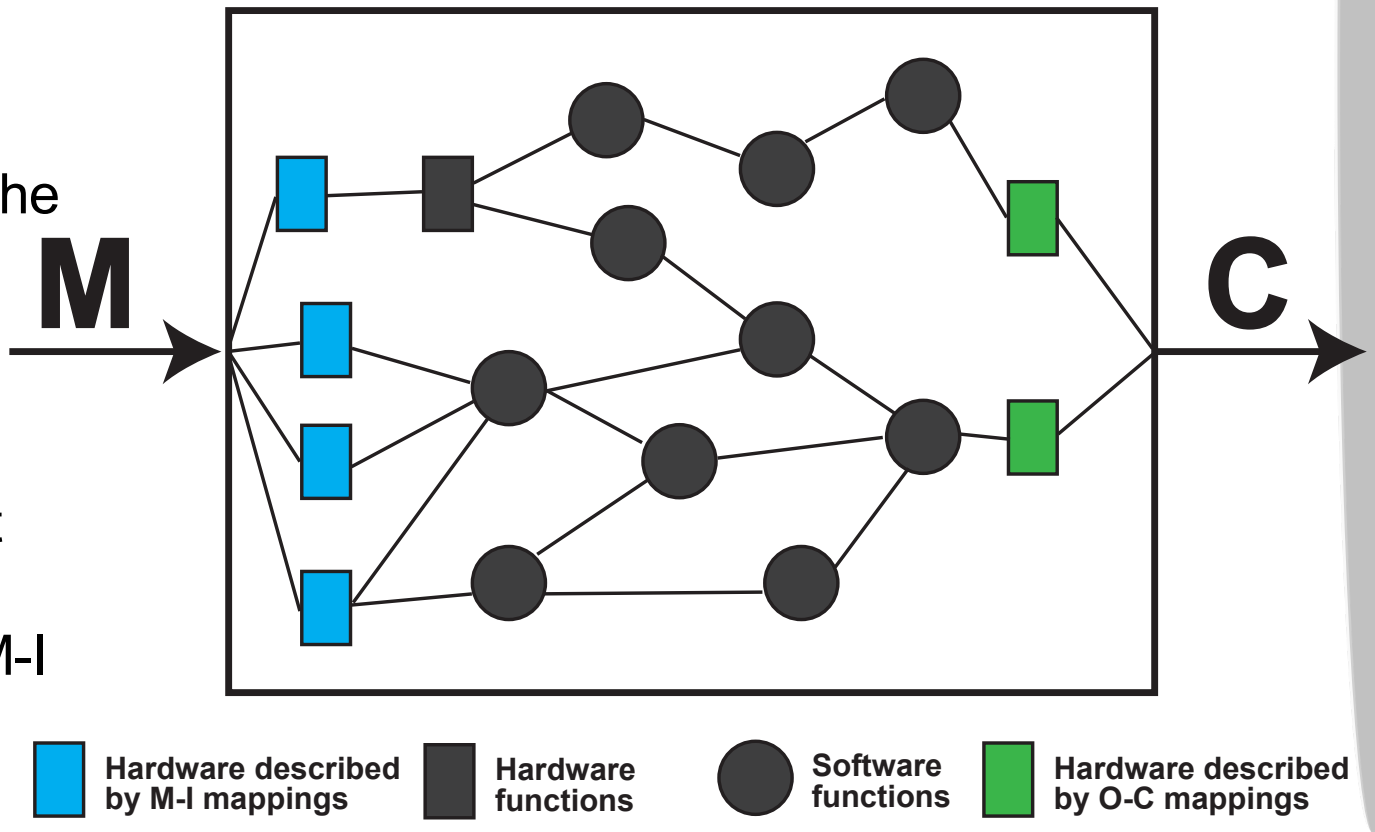
In a few cases, a hardware component may perform actual functional behaviour✪. The software functions may now be used to specify the software requirements.



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 may now be used to specify the software requirements. The hardware functions that represent IN and OUT must be described by M-I and C-O mappings.



System Design - Overview

- What makes a good design?
  - Correct

    The behaviour described in the design must comply with the behaviour described in the requirements, and any new behaviour must be justified
  - Unambiguous

    The descriptions of behaviour must be unambiguous
  - Complete

    Behaviour must be described for all possible input variable combinations, and an argument provided as to why all relevant behaviour has been described
  - Understandable

    The design must be understandable by all stakeholders
  - Maintainable – robust with respect to change

    Every project is subject to future change. *Information hiding* can produce a design that is robust with respect to anticipated changes

Design for Different Kinds of Projects

- We have projects that are almost entirely software applications, and other projects that involve considerable hardware components
- It turns out that the Design documentation is not completely different for the different kinds of projects
- The assumption is that every project requires hardware of some description – even web based apps depend on hardware provided by the platform

Hardware/Software Projects

- Show decomposition into hardware and software components and provide rationale for the decomposition
- Describe M-I and O-C mappings – this describes how the hardware translates between physical variables (monitored and controlled variables) and their software counterparts (input and output variables)
- Describe behavior of any hardware components that provide functionality
- Document software design

Software Applications

- If necessary describe M-I and O-C mappings
- If not necessary (M=I and O=C provided by platform) then document this
- Document software design

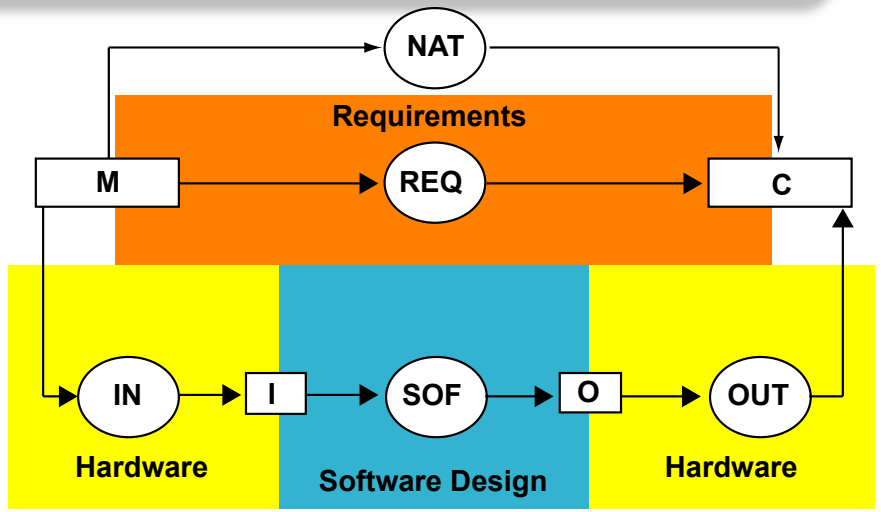
Software Design

- Modified 4 variable model helps deal with mismatch between requirements and design variables
- Software decomposition
- Ways of evaluating how good the decomposition is
- Documenting the design in sufficient detail to be useful and to help debugging later

Preliminary Concepts

Recall the 4 variable model

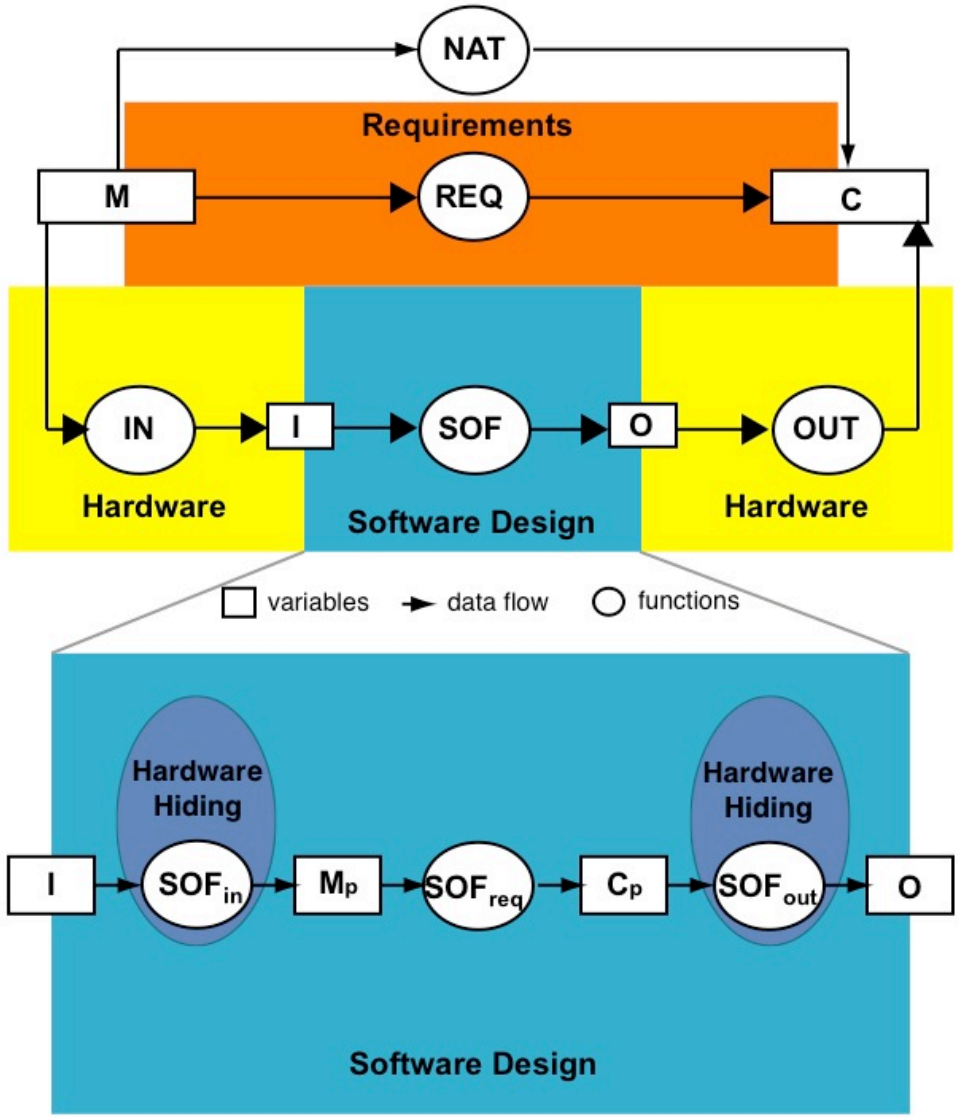
- REQ is the function describing the required behaviour in the requirements, subject to constraints in NAT
- SOF is the function we want to describe in the SDD, and we want the behaviour in SOF to comply with the required behaviour specified in REQ
- The problem is that, in general, I ≠ M and O ≠ C
- It would be much simpler if we could construct SOF in a way that simply mimics REQ, and this is not possible if their inputs are different



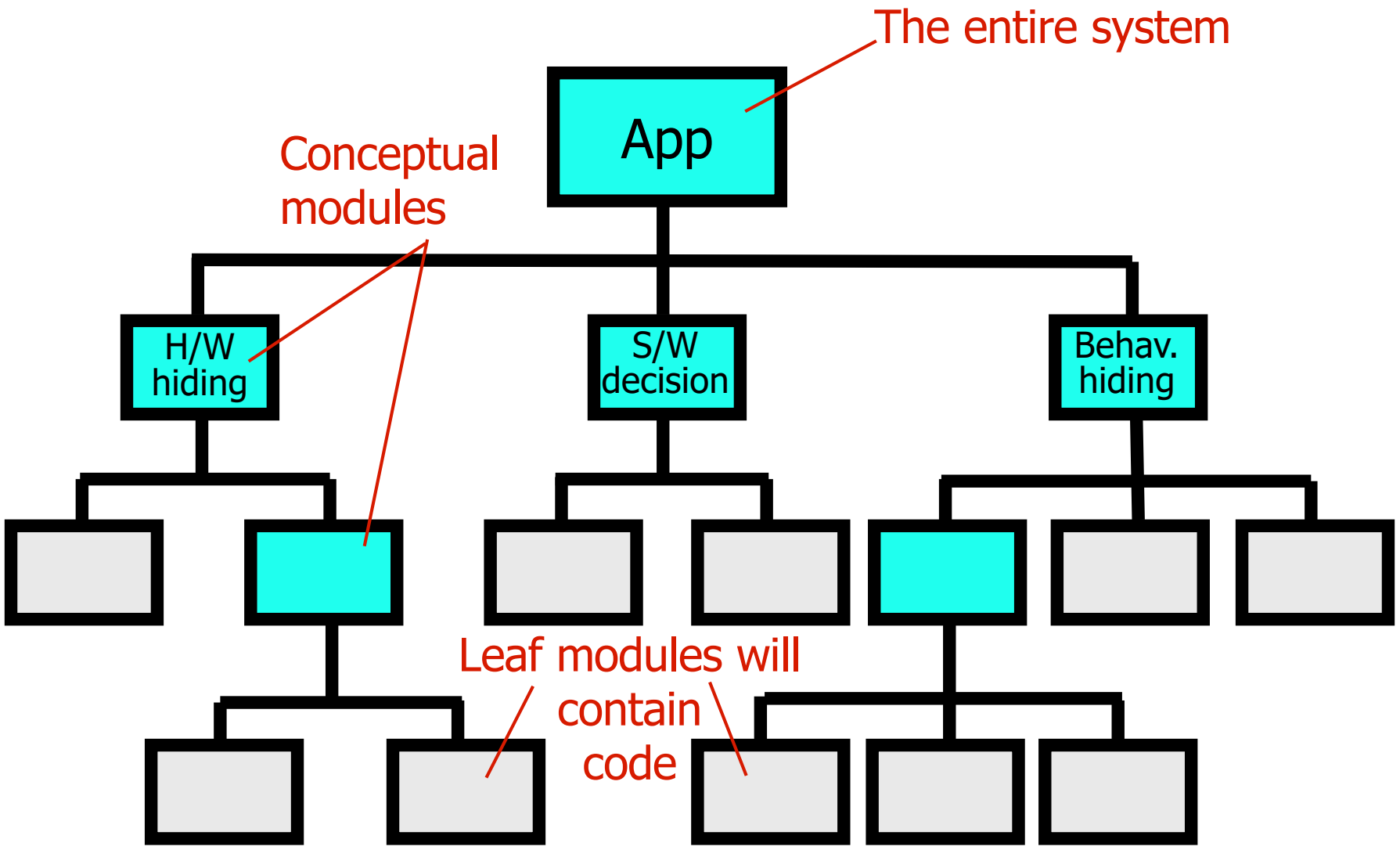
Preliminary Concepts

This problem is dealt with very easily

- In the modified 4 variable model (we have called it the 6 variable model), we show that we can create hardware hiding functions (SOF_{in} and SOF_{out}) that enable us to use REQ directly as a base for our software design SOF_{req}
- If $M_p = SOF_{in}(I)$ is close enough to M, then if $SOF_{req} = REQ$, we should have C_p close enough to C so that $O = SOF_{out}(C_p)$ results in $C = OUT(O)$



Module Decomposition (Parnas)

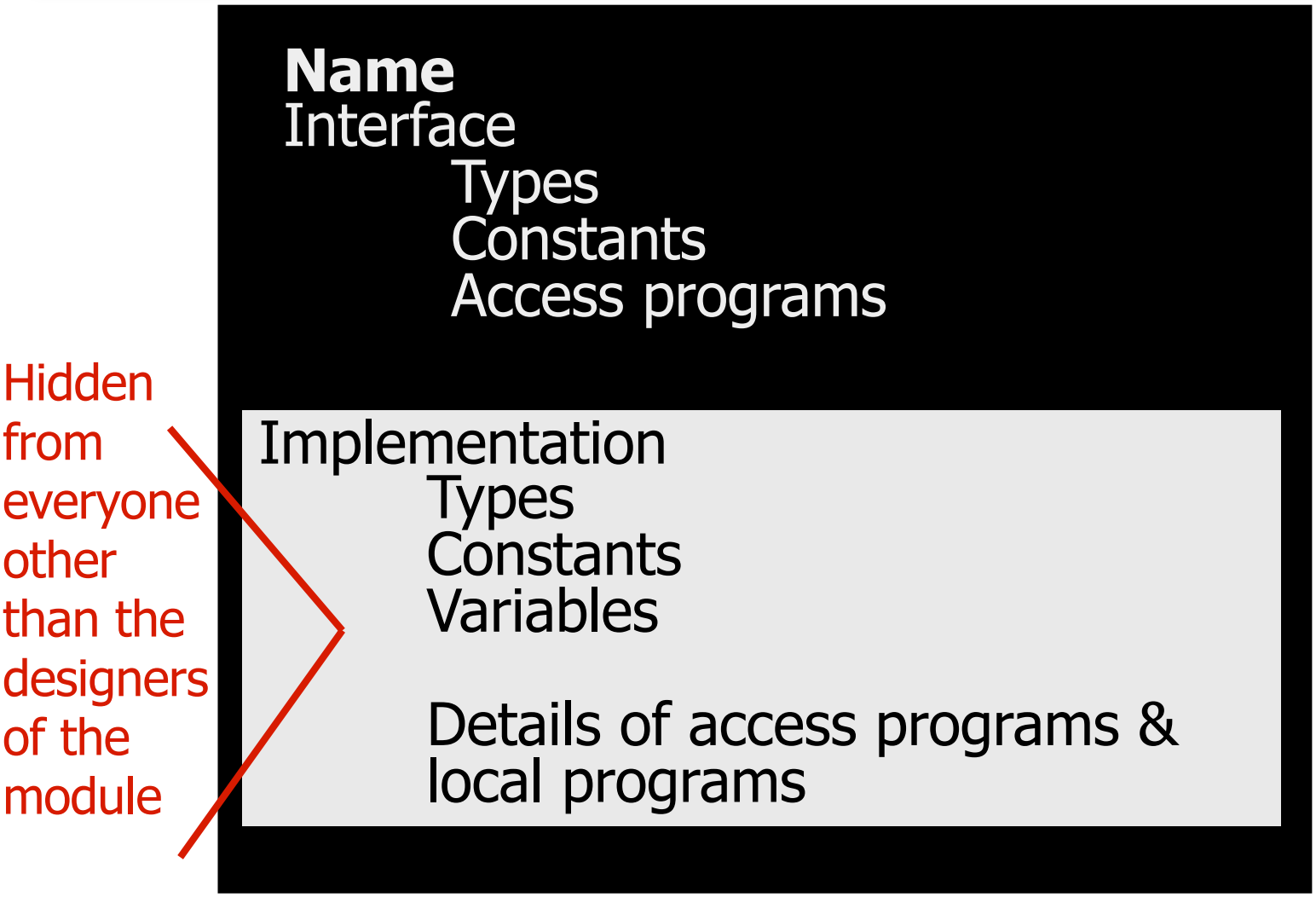


ID	NAME	RESPONSIBILITY	SECRET
1	App	Entire system	All secrets
1.1	H/W hiding	All hardware related behaviour	List of h/w secrets
1.1.1	Analog input class 1	Behaviour for all analog I/O	Secret for class 1
1.1.2	Digital I/O	Behaviour for all digital I/O	List of secrets
1.1.2.1	Digital inputs class 2	Behaviour 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

Module Fundamentals

Name
Interface
- Types
- Constants
- Access programs

Module Fundamentals



Module Fundamentals

Module A
int getS1
int getS2
setValues(int v1,v2)

int S1, S2
....
setValues(int v1,v2)
...
q = B.getV
S1 = v1
S2 = v2*q

Module B
int getV
setValue(int n)

int V
....
setValue(int n)
...
V = n

Module Fundamentals

Module A
Uses B
int getS1
int getS2
setValues(int v1,v2)

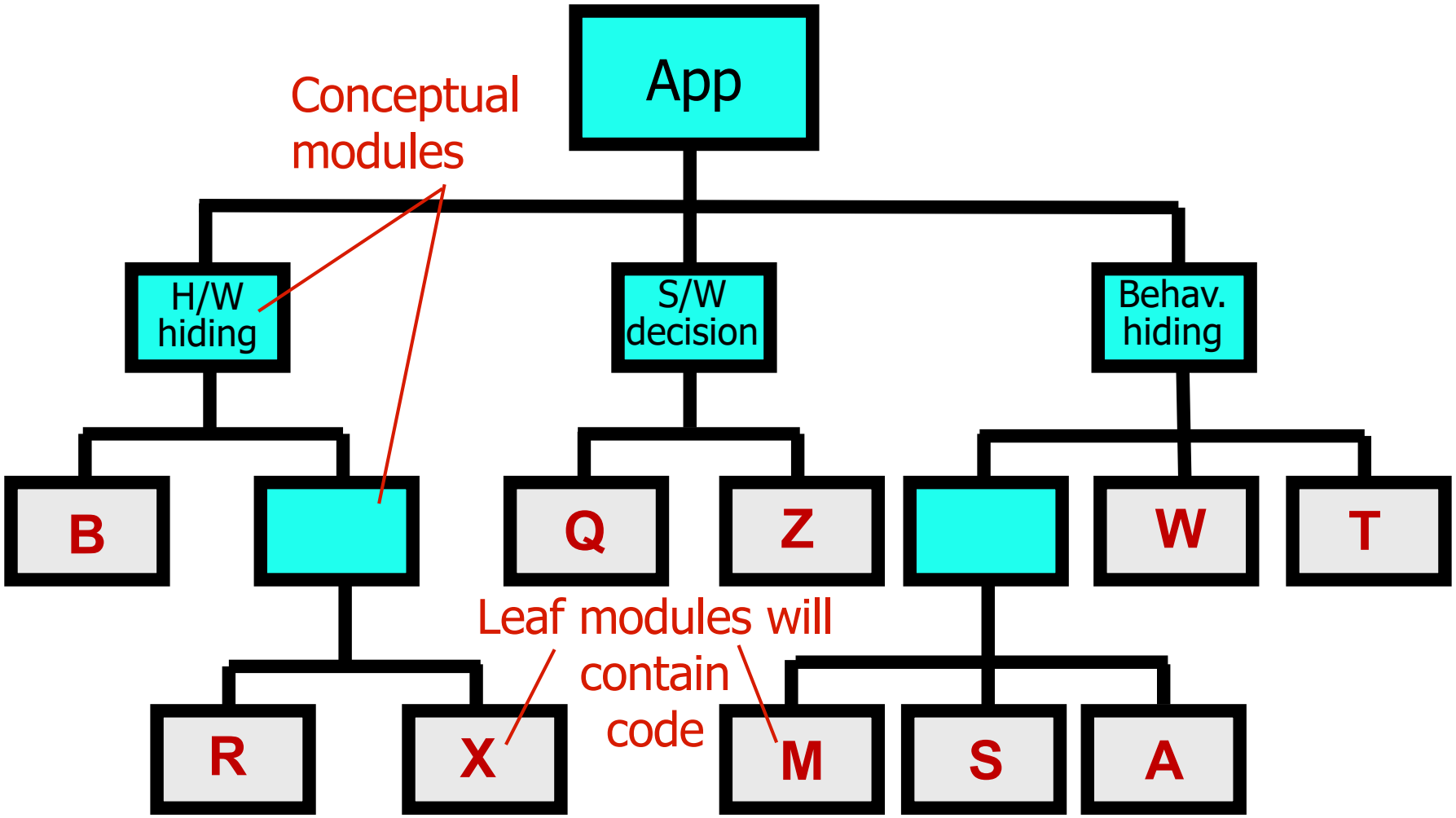
```
int S1, S2
....
setValues(int v1,v2)
   ...
   q = B.getV
   S1 = v1
   S2 = v2*q
```

Module B
int getV
setValue(int n)

```
int V
....
setValue(int n)
   ...
   V = n
```

So we need to add that A "uses" B

Difference between Decomposition & Uses



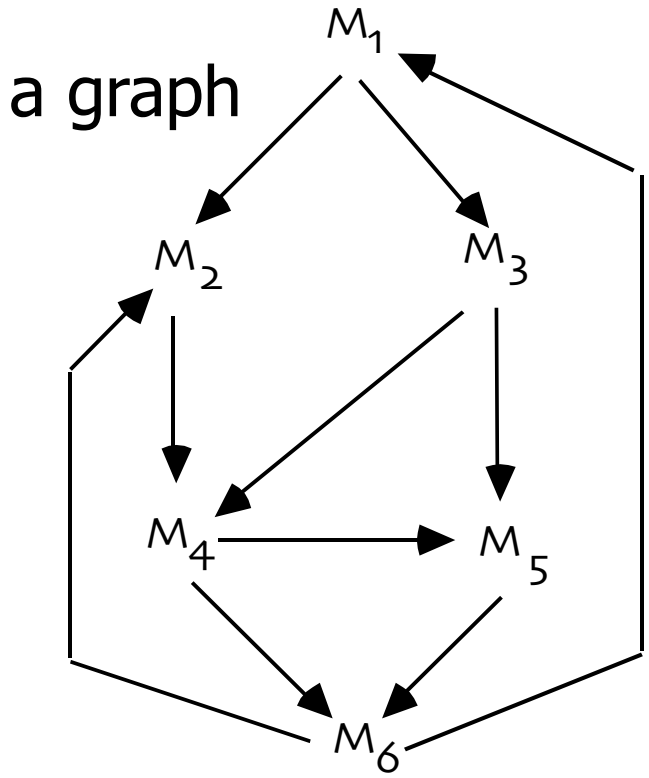
Uses relationship solely describes what leaf module (class) depends on another leaf module (class) eg A uses B, or S uses Z

Evaluating your Design

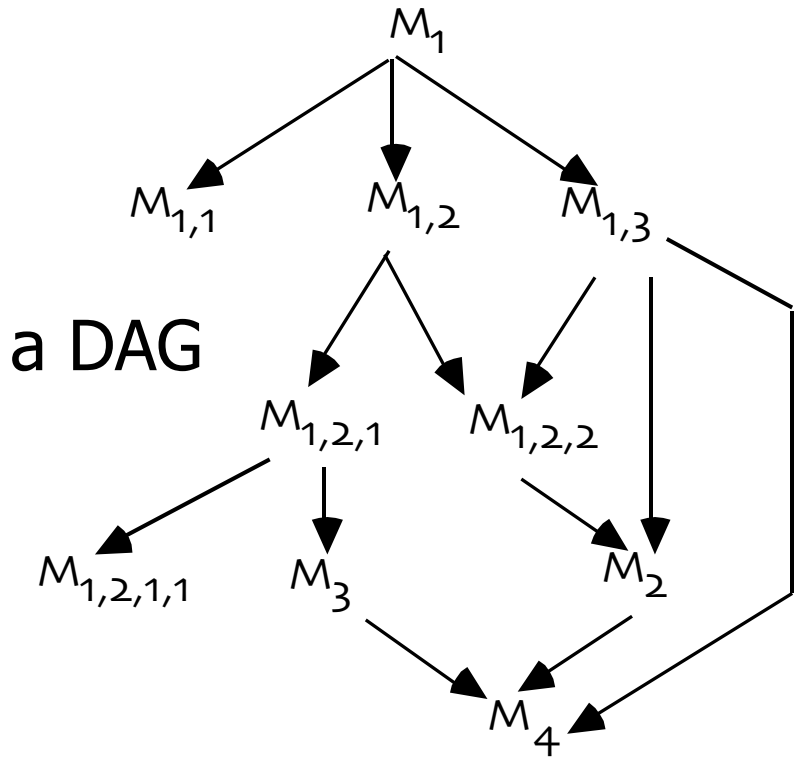
- Correctness – comply with requirements
- Completeness – includes all requirements
- Uses relationship – see discussion
- Cohesion – see discussion
- Coupling – see discussion
- Information hiding – “test” if each secret is hidden in a single class (no variables on interface!)

Relations

- A hierarchy is a DAG (directed acyclic graph)



a)



b)

Uses Relationship

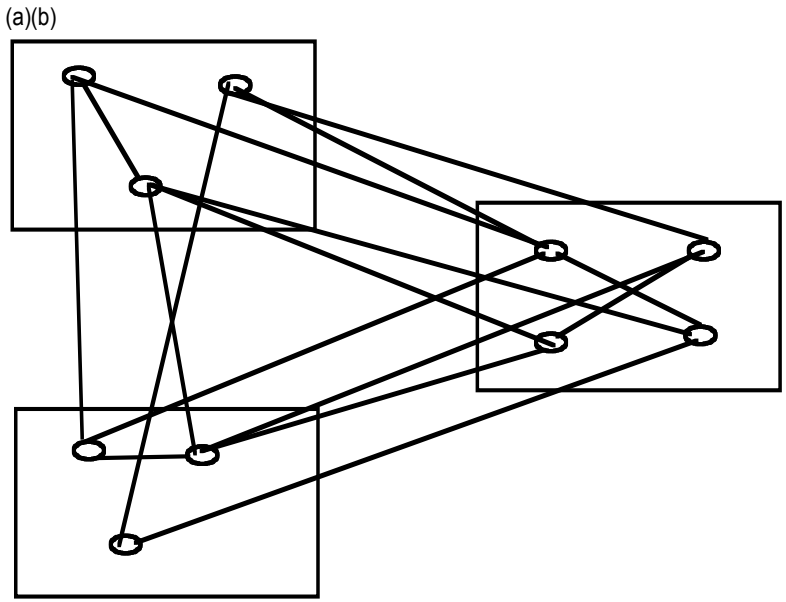
- A uses B
  - A requires the correct operation of B
  - A can access the services exported by B through its interface
  - it is “statically” defined
  - A depends on B to provide its services
    - example: A calls a routine exported by B
- USES should be a hierarchy
- Hierarchy makes software easier to understand
  - we can proceed from leaf nodes (who do not use others) upwards
- Hierarchy makes software easier to build
- Hierarchy makes software easier to test

Cohesion and coupling

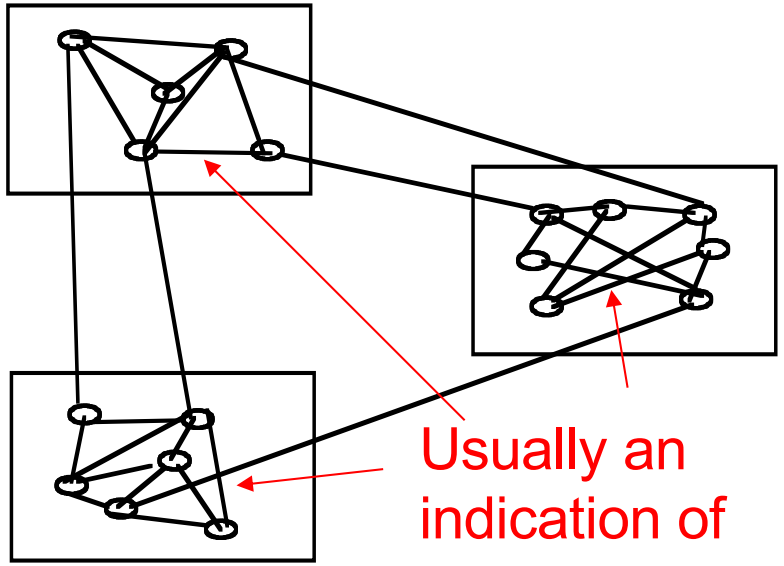
- Each module should be *highly cohesive*
  - module understandable as a meaningful unit
  - Components of a module are closely related to one another
- Modules should exhibit *low coupling*
  - modules have low interactions with others
  - understandable separately

A visual representation

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



high coupling

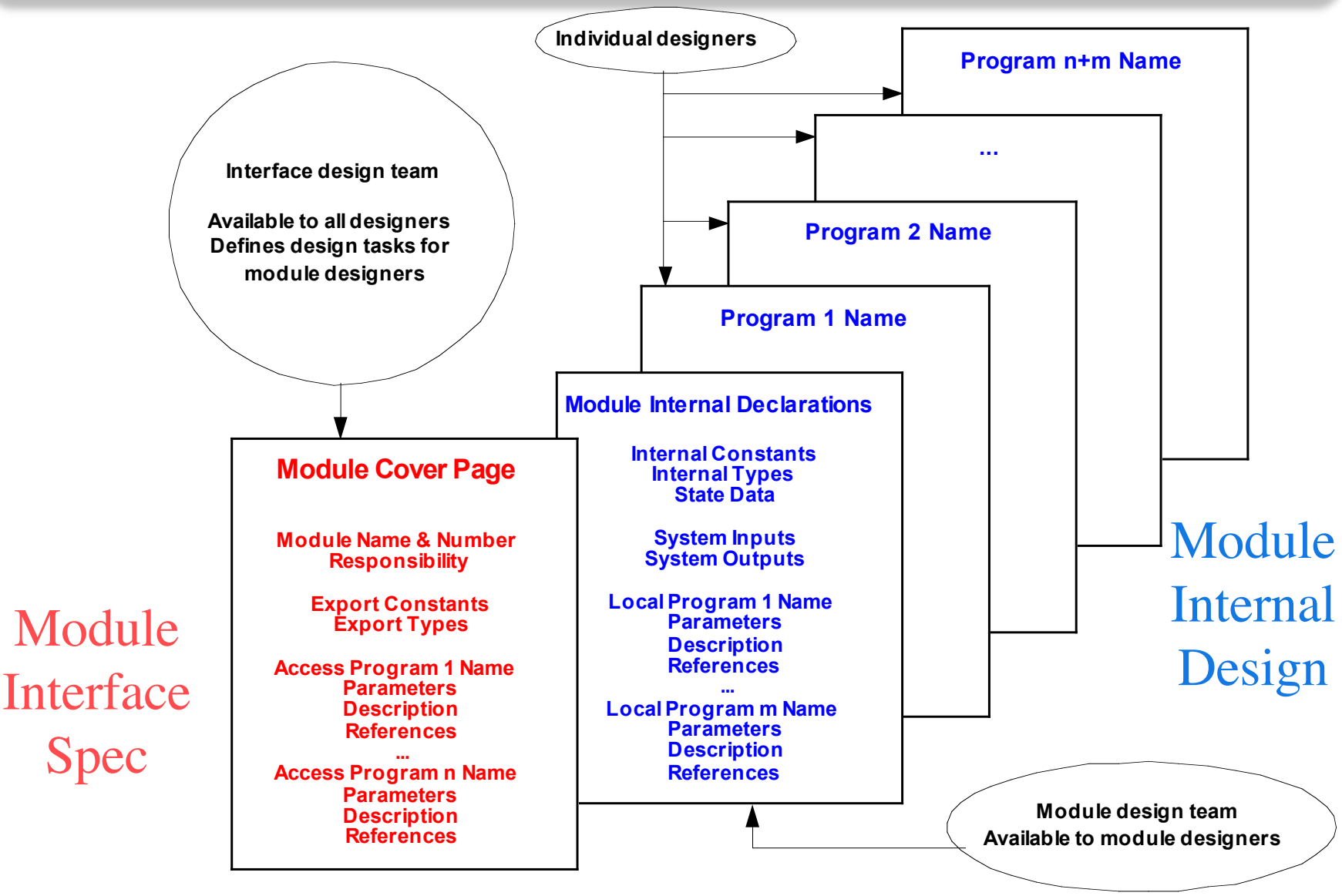


low coupling

Documenting Design

- Look at suggested contents & rubric
- Primary means for documenting a lot of the design are the
  - Module Interface Specification - MIS
  - Module Internal Design - MID

Documentation of Modules



MIS

- My opinion – MIS is the most important entity in your software design
- Describes what the interface to your class is – both syntax and semantics
- List of items with parameters etc is the syntax
- Description of behaviour of each method is the semantics
  - For each method describe the value of the outputs as they depend on the inputs