Modularity

Guidelines for design
in any programming language

Modular Software

- Software constructed as assemblies of small pieces
  - » **Each piece encompasses the data and operations necessary to do one task well**
- Modular software ==> maintainable software
  - » **Uses divide and conquer principle**
- Meyer:
  - » **To achieve extendibility, reusability, compatibility, need modular software and methods to produce modular software**
- In OO design — non-OO design
  - » **Module ≡ Class** — **Module ≡ File**

Issues in Modular Design

- Information hiding
- Independence
  - **Each module implements a separable part of the whole**
  - **modules have small, simple interfaces**
  - **high interaction between modules is usually symptomatic of a bad modular design**
- Key ideas: **coupling** and **cohesion**
  - **Cohesion – how "self contained" a module is**
  - **Coupling – how dependent modules are on each other**

Want high cohesion and low coupling

Criteria for Modularity

- Want a modular design method satisfying
  - **Decomposability**
  - **Composability**
  - **Understandability**
  - **Continuity**
  - **Protection**

- Without these, we cannot produce modular software

Decomposability

- Decomposition
  - **Break a problem into sub-problems connected by simple structures**
    - **Minimize communication between sub-problems**
    - **Permit further work to proceed separately on each sub-problem**
  - **Example**
    - **See slides on top down design**

Composability

- Composition
  - » Produce software from reusable plug and play modules
  - » Composed software is itself a reusable module
  - » Reusable modules work in environments different from the ones in which they were developed
  - » Examples
    - > Using pipe in the Unix shell to combine Unix commands
    - > See slides on abstract data types and bottom-up design

Decomposability and Composability

- Composability and decomposability are independent and often at odds
  - » **Top down design favours generating modules that fulfill specific requirements, hence, are unsuitable for composition**
  - » **Bottom up design favours general modules that are too general, hence when combined generate inefficient systems – in size and speed**

- Both top down – decomposition – and bottom up – composition are required, however
  - » **Trick is to know when and how to best use both methods**

Understandability and Continuity

- Understandable
  - » **Minimize need to understand module context**
    - > **Know or examine as few other modules as possible**
    - > **Very important for maintenance**
- Continuity
  - » **The smaller the change in specification, the fewer the number of modules that must be changed (edited) and if possible compiled**
    - > **Example: use of symbolic constants – need to change value in one place but requires recompilation of every module using the constant**

Understandability and Continuity – 2

- Related to coupling and cohesion

A module should do one thing well

Modular Protection

- Confine abnormal run time errors to one or a very few modules

- Avoid propagation of error conditions to neighbouring modules
  - » **Example**
    - > **Validate input before propagating it to other modules**

- Exceptions in languages like C++ and Java can be used in an undisciplined manner leading to violations of protection
  - » **Exceptions raised in one part of the system should not be handled by a remote part of the system**

Design Rules to Ensure Modularity

- We have seen criteria for modular software development
- From them we can deduce the following rules that can help establish the properties we want in our designs
  - » **Direct Mapping rule**
  - » **Few interfaces rule**
  - » **Small interfaces rule**
  - » **Explicit interfaces rule**
  - » **Information Hiding rule**

Direct Mapping Rule

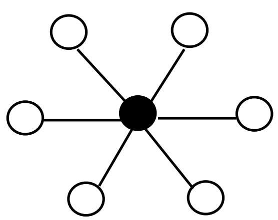
Correspondence
The structure used in implementing a software system should remain compatible with the structure used in modeling the system

- Software design involves addressing needs in a problem domain
- Have to understand the problem AND its domain, then formulate a solution
- Model our solution in some notation (we use BON)
- Need a clear mapping from the proposed solution (in BON) to program source text
- Arises from **continuity** and **decomposability**

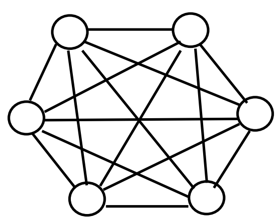
Few Interfaces Rule

Every module should communicate with as few others as possible

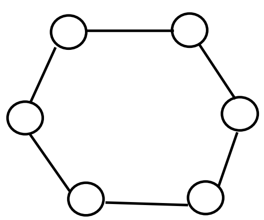
- Restrict the number of communication channels between modules
- Arises from **protection**, **continuity**, **composability**, **decomposability** and **understandability**



Hub



Composite



Ring

Small Interfaces Rule

If two modules communicate, they should exchange as little information as possible

- Also known as **weak coupling**
- Relates to the size of connections rather than their number

Small Interfaces Rule – 2

- Historical bad idea: Fortran COMMON block
  - » **COMMON block1 A[75], B[25]**
  - » **COMMON block1 C[50], D[50]**
    - > **View memory in two different ways!**

- Local variables via Algol-60 block structure

var i

Access all variables in outer block

i := i + 5

Explicit Interfaces Rule

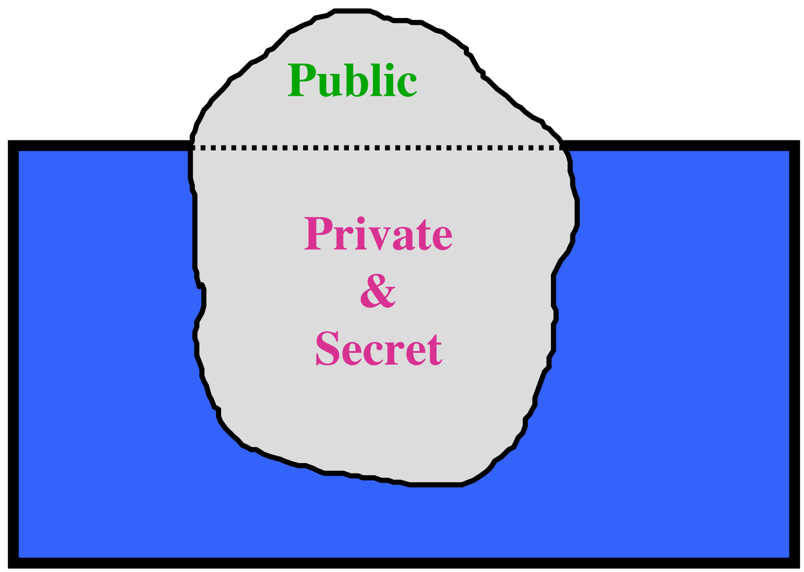
Whenever two modules A and B communicate, this must be obvious from the text of A or B or both

- Conversation is limited to a few participants and only a few words
- Conversations are **loud** and **public**
- Really important with respect to **understandability**
- Worry about procedure **parameters** as well as **shared data**

Information Hiding Rule (Parnas 72)

The designer of every module must select a subset of properties as the official information about the module, to be made available to client modules

- Only **some**, but **not all** of the module's properties are public; the rest are secret
- Public ≡ **interface**



Software Construction Principles

- **Linguistic Modular Units Principle**
  - » **Modules must correspond to syntactic language units**

- **Self-Documenting Principle**
  - » **Module designers should make all information about the module part of the module itself**

Software Construction Principles – 2

- **Uniform Access Principle**
  - » All module services should be available through a uniform notation, which does not betray whether they are implemented through storage or computation
  - » Allow implementer to make space-time tradeoffs
- **Single Choice Principle**
  - » Whenever a system must support a set of alternatives, one and only one module in the system should know their exhaustive list

Software Construction Principles – 3

In real projects
A module needs to be both open and closed!

- Open-Closed Principle
  - Open – Available for extension – add new features
  - Closed – Available for client use – stable in spite of extensions
  - When are we done?
  - We must make modules available to others!
- Classical approach
  - Close when stability is reached, reopen when necessary
  - But need to reopen all the clients too!
  - Inheritance offers a solution to this problem