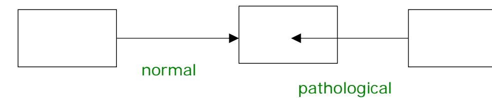
ECE450 – Software Engineering II

Today: **Key Principles of Software Architecture and Design (II)**

adapted from Dave Penny's CSC407 material

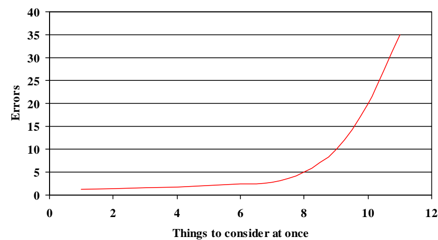
Structured Design

- Early work of software design (from 1979) that presented concepts such as cohesion, coupling, and encapsulation.
  - "Fundamentals of a Discipline of Computer Program and Systems Design"
    - by Edward Yourdon and Larry Constantine
- Modules are not the same as for Parnas:
  - Module: A lexically contiguous sequence of program statements, bounded by boundary elements, having an aggregate identifier.
    - A function, a procedure, a method
- **Normal** and **pathological** connections between modules:



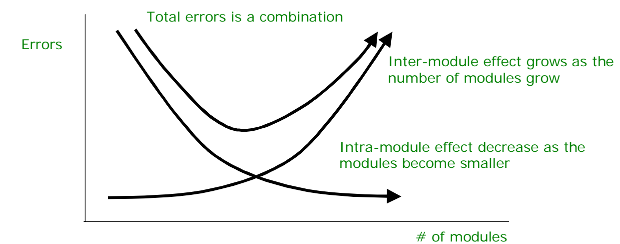
Human limitations on dealing with complexity

- George Miller: *The Magical Number Seven, Plus or Minus Two*
  - Can't keep track of too many things at the same time
  - Yourdon: Maximum number of subroutines called by a routine should be 5-9.



Two kinds of complexity

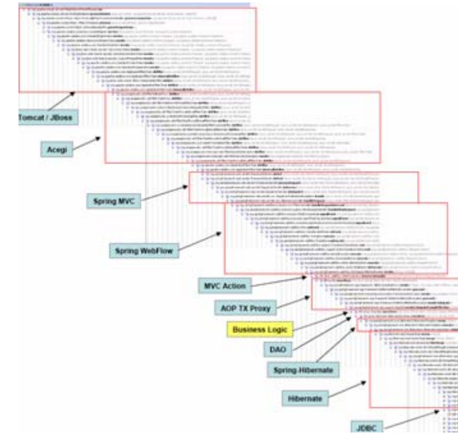
- Intra-module complexity
  - Complexity within one module
- Inter-module complexity
  - Complexity of modules interacting with one another



Overall cost

- The overall cost of a system depends on both:
  - The cost of production (and debugging)
  - And the cost of maintenance
    - Both are approximately equal for a typical system
- These costs are directly related to the complexity of the code
  - Complexity injects more errors and makes them harder to fix
  - Complexity requires more changes and makes them harder to effect
- Complexity can be reduced by breaking the problem into smaller pieces
  - (So long as the pieces are relatively independent of one another)
- But eventually the process of breaking pieces into smaller pieces creates more complexity than it eliminates.
  - 1970's: Happens later than most designers would like to believe
  - 2000's: Happens sooner than most designers would like to believe

In case you don't believe it…



Design approach

- Therefore, there is some optimal level of sub-division that minimizes complexity
  - But to reach it you need your judgment
- Once you know the right level, the key decision is to choose **how** to divide:
  - Minimize coupling between modules
    - Reduces complexity of interaction
  - Maximize cohesion within modules
    - Keeps changes from propagating
  - Duals of one another

Coupling

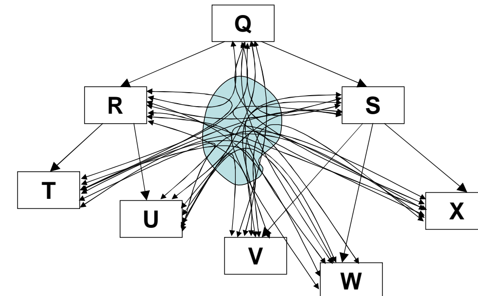
- Two modules are **independent** if each can function completely without the presence of the other
  - They are decoupled, or uncoupled
- Highly coupled modules are joined by many interconnections and dependencies
  - And loosely coupled modules have a few interconnections and dependencies
- Goal: Minimize coupling between modules in a system
  - Coupling translates into "the probability that in coding/modifying/debugging module A we will have to take into account something from module B"
- Note that a system that has only one module (function) is absolutely uncoupled
  - But that's not what we want!
  - (We'll analyze *cohesion*, coupling's complement, later)

Influences on coupling

- Type of connection
  - Minimally connected: parameters to a subroutine
  - Pathologically connected: non-parameter data references
- Interface complexity
  - Number of parameters/returns
  - Difficulty of usage
- Information flow
  - Data flow: Passing data is handled uniformly
  - Control flow: Passing of flags governs how data is processed
- Binding time
  - More static = more complex
    - E.g., literal "30" vs. pervasive constant N_STUDENTS, vs. execution-time parameter

Common-environment coupling

- A module writes into global data
- A different module reads from it (data or, worse, control)



Cohesion

- While minimizing coupling, we must also maximize cohesion
  - How well a particular module "holds together"
    - The cement that holds a module together
  - Answer the questions:
    - Does this make sense as a distinct module?
    - Do these things belong together?
- Best cohesion is when it comes from the problem space, not the solution space
  - Echoed years later in OOA/OOD

Levels of lack of cohesion (roughly from worst to best)

- Coincidental
  - No reason for doing two things in the same routine
    - double computeAndRead(double x, char c);
- Logical
  - Similar class of things that still should be separated
    - char input(bool fromFile, bool fromStdIn);
- Temporal
  - The fact that things happen one after the other is no excuse to put them in the same routine
    - void initSimulationAndPrepareFirst();
- Procedural
  - Operations are together because they are in the same loop or decision process, but no higher cohesion exists
    - typeDecide(m); // Decide type of plant being simulated and perform simulation part 1

Levels of lack of cohesion (roughly from worst to best) (cont)

- Communicational
  - Procedures that access the same data are kept together
    - void printReports(data x); // Outputs day report and monthly summary
- Sequential
  - A sequence of steps that take the output from the previous step as input for the next step
    - string compile(String program) {parse, semantic analysis, code generation}
- Functional
  - That which is none of the above
  - Does one and only one conceptual thing
  - Equivalent to information hiding
    - double sqrt(double x);

Implementation and cohesion

- No need to be dogmatic
- Consider module FG that does two things, F and G
  - Chances are there is some code that can be shared between them
  - If F and G have high cohesion, that's OK
  - Otherwise it will become difficult to work with