Today: **Key Design Principles**

Broad history of architecture and design

- 1960's: Structured Programming
  - Response to spaghetti code
  - "GOTO Considered Harmful", Dijkstra
    - Why?
  - It took a while to convince people that coding without GOTOs was
    - Possible
    - Desirable
- 1970's: Structured Design
  - There are good ways and bad ways to divide programs in subroutines
- 1980's and 1990's: Object Orientation
  - Move away from structured languages and into object-oriented programming
  - Object-oriented analysis and design as a way to make sense of the world

Module Structure

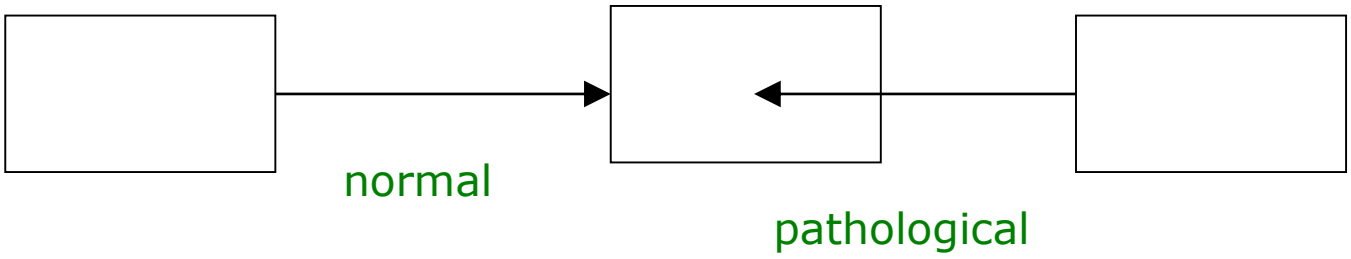
- Classic paper:
  - "On the Criteria to be Used in Decomposing Systems into Modules", by David Parnas
    - Discusses modularization (where "module" = collection of subroutines and data elements)
    - Critiques design by flowchart
    - Hints towards object-oriented approaches to design
    - Key lessons still overlooked by most of us!

Structured Design

- Early work of software design (from 1979) that presented concepts such as cohesion, coupling, and encapsulation.
  - "Structured Design:
    Fundamentals of a Discipline of Computer Program and Systems Design"
    - by Edward Yourdon and Larry Constantine (1979)

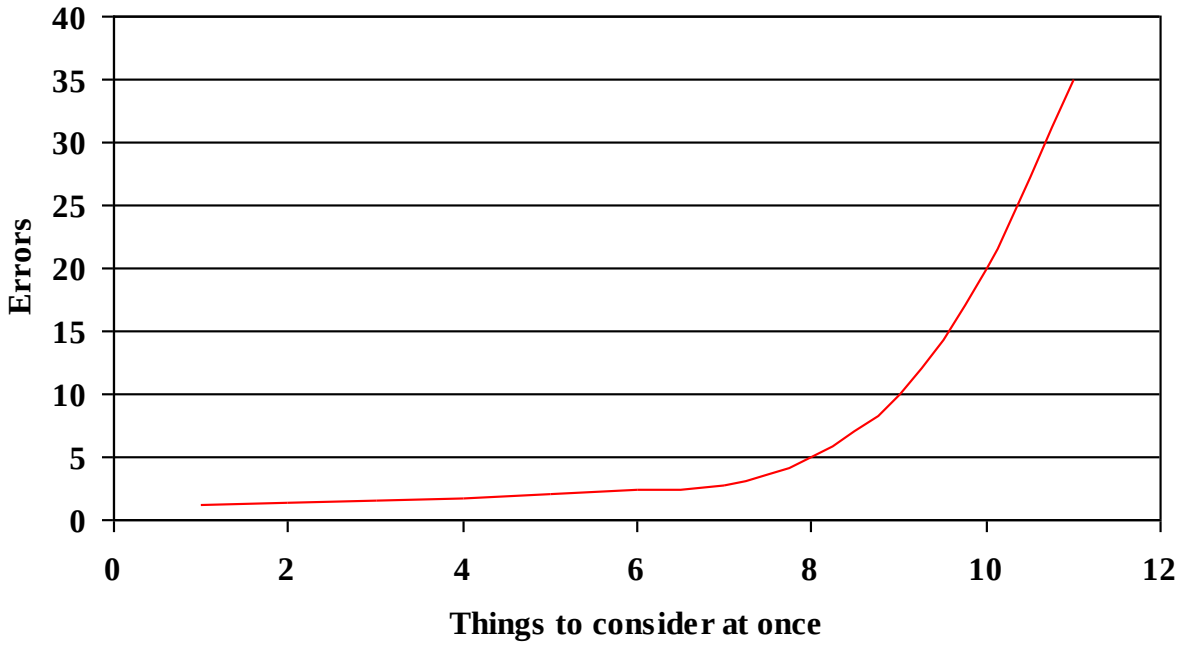
- Modules are not the same as for Parnas:
  - Module: A lexically contiguous sequence of program statements, bounded by boundary elements, having an aggregate identifier.
    - A function, a procedure, a method

- **Normal** and **pathological** connections between modules:



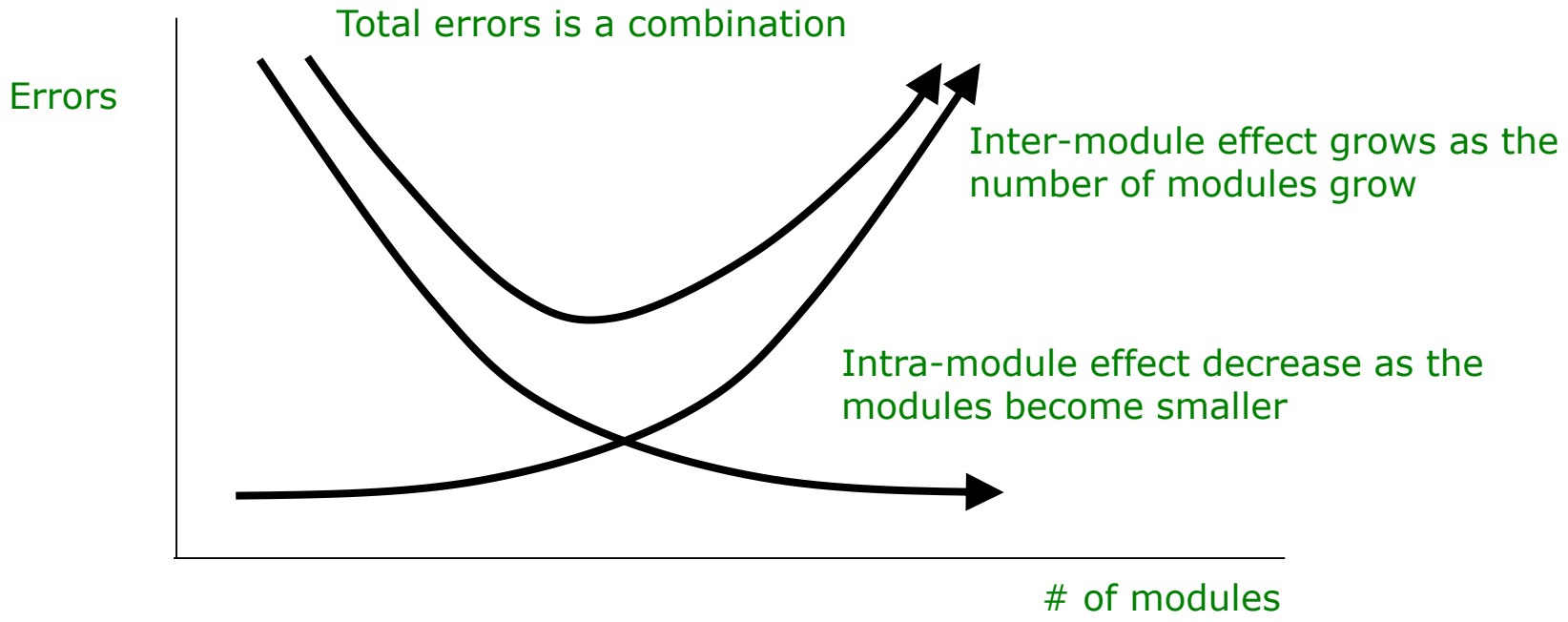
Human limitations on dealing with complexity

- George Miller: *The Magical Number Seven, Plus or Minus Two Some Limits on Our Capacity for Processing Information" (1956)*
  - 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

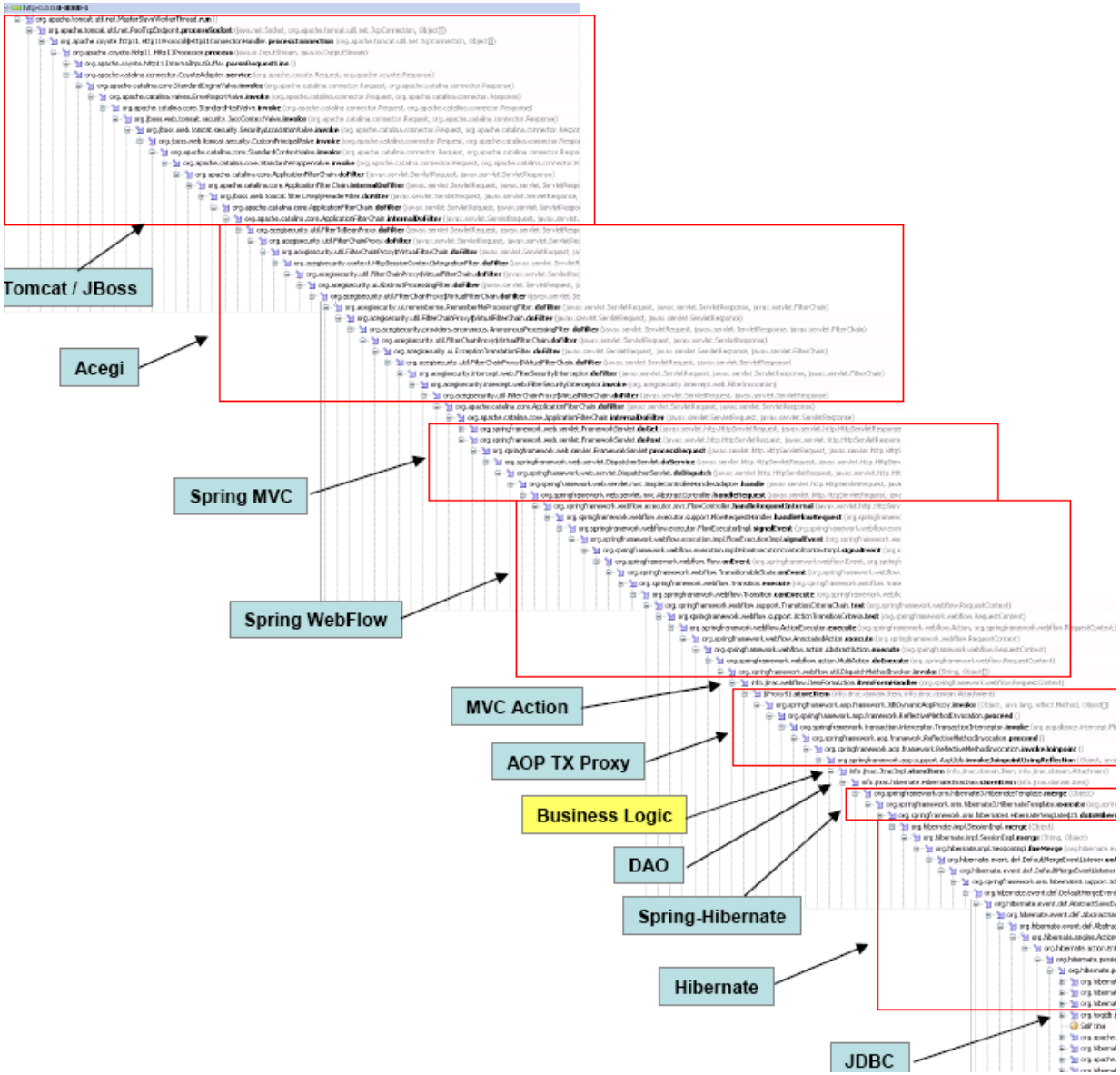
- Intra-module complexity
  - Complexity within one module
- Inter-module complexity
  - Complexity of modules interacting with one another



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.

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

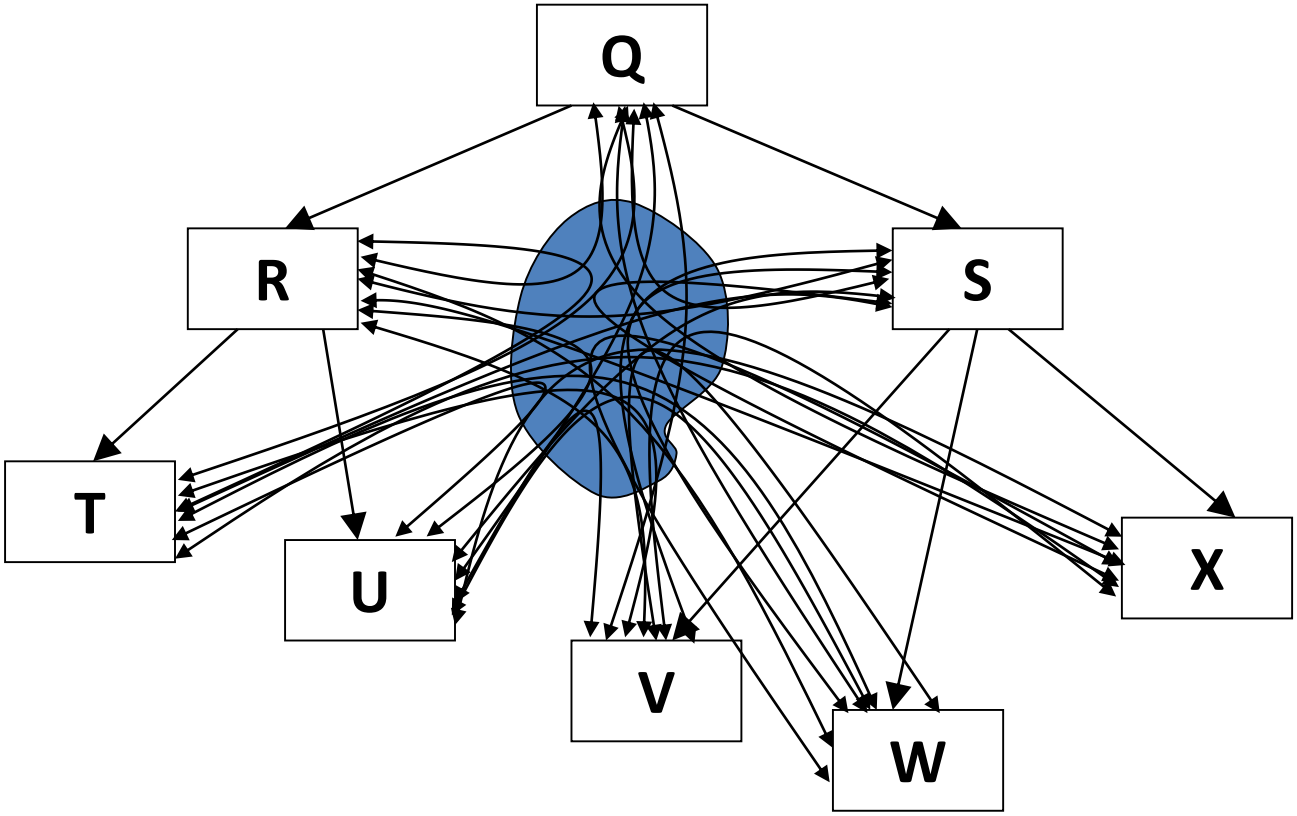
- Two modules are **independent** if each can function completely without the presence of the other
  - They are decoupled, or uncoupled
- Highly coupled modules are joined by many interconnections and dependencies
  - And loosely coupled modules have a few interconnections and dependencies
- Goal: Minimize 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)

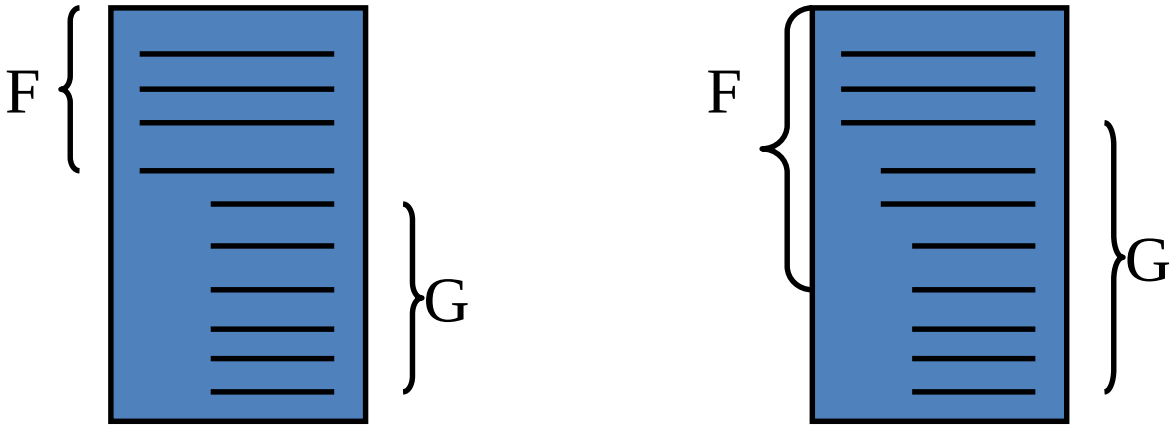
- Coincidental (worst)
  - No reason for doing two things in the same routine
    - double computeAndRead(double x, char c);
- Logical
  - Similar class of things that still should be separated
    - char input(bool fromFile, bool fromStdIn);
- Temporal
  - The fact that things happen one after the other 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, try to be flexible.
- 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



Object-Oriented Design

- The object-oriented transformation of the 1980's and 1990's was particularly profound, but it wasn't easy
  - Object-oriented development salesmen took advantage of the wave of enthusiasm
    - Objects were supposed to improve your performance tenfold
    - Promise of reuse: Plug in your classes anywhere you need them
  - Many people struggled to "get it"
    - ...and wrote object-oriented programs just like they used to write structure-oriented programs
    - I.e., programs -> classes; functions -> methods; or...
    - I.e., copy all of your program and put it in the main() method of your class.

- Object modeling in the 1990's
  - Resulting in high cohesion
    - Tends to isolate changes
    - Tends to make the program easier to understand

Object-Oriented Principles

- Encapsulation
  - Which embodies one of our now-familiar principles (information hiding)
  - Modern languages allow us to *enforce* encapsulation through access declaration (example: public vs. protected vs private attributes)
- Inheritance
  - Declare new classes by extending old ones
  - We inherit all of the old attributes and methods, but are free to modify/override any of them, and to add new ones
- Polymorphism
  - Substitute one type for another without the caller needing to know
    - We can make a *Student.getGrade()* call without worrying if we're dealing with an *UndergraduateStudent,* a *GraduateStudent,* or a generic *Student*.

Object Modeling Method

- How do we even come up with the classes we will use in our system?

- Step 1: Object-Oriented Analysis
  - Analyze the problem domain
    - Identify problem domain classes and relationships between classes
    - Identify attributes and methods
    - Identify states and transitions
    - Sample object structures and interactions
  - At this level we are not programming! We are abstracting the real world

- Step 2: Object-Oriented Design
  - Use the analysis as the core of a solution to:
    - User interface design
    - Database design
    - Program design

Can we model everything?

- I'd like to see you try...
  - But the world is too complex for us to model it completely
- A full model of you should include:
  - Your basic information (name, gender, etc.)
  - Your background
  - Your family background and a trace to your ancestors
  - Your medical history
  - Your record of marks
  - Your fingerprints and other bio-prints
  - Your financial information
  - A list of your friends, crushes, enemies, and acquaintances
  - Your DNA
  - ...
- We only model that which is relevant to the problem domain that we face

Summary of design principles

- Decomposition
  - Divide and conquer. Deal with less complicated problems.
- Information hiding
  - Each module/class hides its own secrets (data representations, algorithms, formats, lower-level interfaces)
- Minimize coupling
  - If we talk/share information, it's because the problem demands it
- Maximize cohesion
  - Each module/class does (close to) one thing only
- Extensibility
  - Open for extension, closed for modification
- *How does this compare to* ***SOLID*** *(the five basic principle of OOD-OOP)?*