Structured Design

- Structured Design
  - Fundamentals of a Discipline of Computer Program and Systems Design
    - Edward Yourdon / Larry L. Constantine
  - Prentice-Hall, 1979
- Purpose
  - Make methodical the process of designing software systems
    - Mainly business systems
- Approach
  - Defines properties of a good procedural design
  - Defines a step-by-step method for transforming a data flow graph into a procedural design
    - N.B. calls “procedures” (possibly with associated static data) “modules”, which differs from Parnas’ use of the term as a grouping of multiple procedures and related data

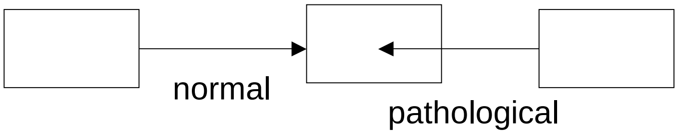
Structured Design Significance

- Very popular in business circles.
- Never caught on in academic circles
  - Ideas are somewhat half-baked
    - “theorems” with silly or no proofs
    - Ill-described concepts (no firm definitions)
  - At a time when predicate logic to describe programming semantics was in vogue
- Nonetheless, full of useful concepts.
- Somewhat dated, as it applies best to data-flow oriented software (not interactive, real-time, or database oriented)
  - E.g., read update records from tape, merge them into a master file, and print a report.

Modules and Connections

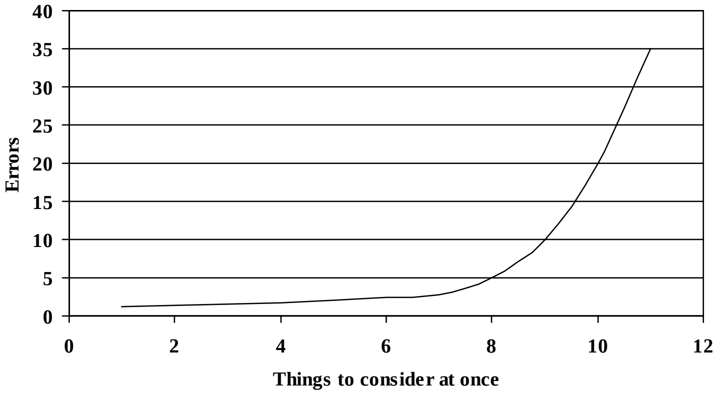
- Module
  - A lexically contiguous sequence of program statements, bounded by boundary elements, having an aggregate identifier.
    - i.e., a “function” or “procedure” or ? “method” ?

- Connections



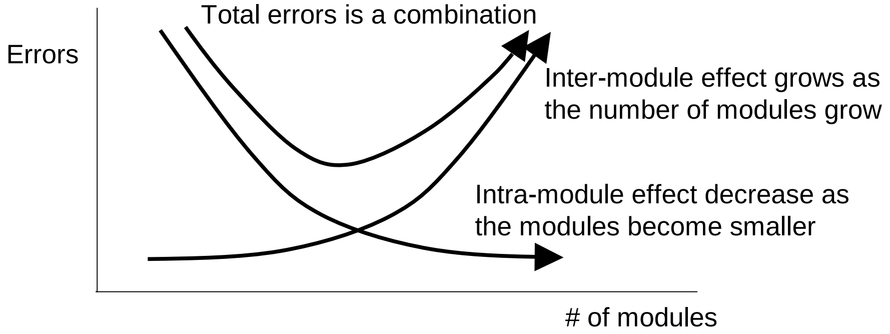
Limitations on Dealing with Complexity

- Errors: 7 ± 2 rule
  - Based on work of psychologist George Miller
    - Now questioned in the HCI community



Total Errors in a System

- Two opposing forces:
  - Intra-module complexity: Complexity within one module
  - Inter-module complexity: Complexity of modules interacting with one another



Total errors is a combination

Inter-module effect grows as the number of modules grow

Intra-module effect decrease as the modules become smaller

Overall Cost

- The cost of developing most systems
  - ≈ cost of ***debugging*** them (+ cost of ***changing*** them nowadays)
- These costs are directly related to the overall complexity
  - Complexity injects more errors and makes them harder to fix
  - Complexity requires more changes and makes them harder to effect.
- Complexity can be decreased by breaking the problem into smaller pieces
  - So long as those pieces are relatively independent of one another
- Eventually, the process of breaking pieces into smaller pieces creates more complexity than it eliminates.
  - 1970s: Happens later than most designers would like to believe.
  - 2000s: Happens sooner than most designers would like to believe.

Design Approach

- Therefore, there is some optimal level of sub-division that minimizes complexity
  - Use your judgment
- Once you know the right level, then must choose ***how*** to sub-divide:
  - Minimize ***coupling*** between modules
    - Reduces the complexities 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/dependencies
- Loosely coupled modules are joined by few interconnections/dependencies

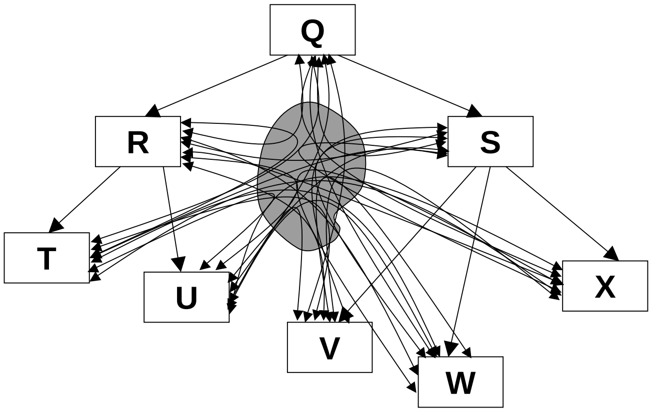
- Wish to minimize coupling between modules in a system
  - Coupling = probability that in coding/modifying/debugging module A we will have to take into account something about module B

Influences on Coupling

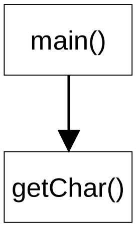
- Type of connection
  - Minimally connected: parameters to a subroutine
  - Pathologically connected: non-parameter data references
- Interface complexity
  - Number of parameters/returns
  - Difficulty of usage. e.g., A = sqrt(x,y,z)
- Information flow
  - Data flow
    - Passing a of data to be acted upon in a uniform fashion
  - Control flow
    - Passing of flags that govern how other data is processed
- Binding time
  - More static = more complex
    - e.g., literal ’80’ versus pervasive constant N_STUDENTS, versus execution-time parameter.

Common-Environment Coupling

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

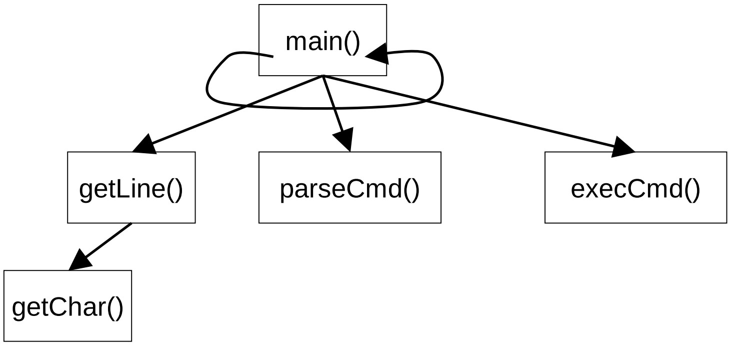


Coupling Example 1



- Alternate interfaces:
  - `void getChar(bool& eof, char& c)`
  - `char getChar(bool& eof);`
  - `char getChar();`
- Either way:
  - 1 data coupling
  - 1 control coupling

Coupling Example 2



Interfaces

- `void getChar(char& c, bool& eof);`
- `void getLine(char* &line, bool& eof);`
- `void parseCmd(char* line, Command& cmd);`
- `void execCmd(Command cmd);`

- 3 data couplings, 4 command couplings

Example

- Go to your tutorial!
- I will give you a similar question on the exam.

Cohesion

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

Elements of Processing

- A module is composed of ***processing elements.***
  - ill-defined
  - roughly corresponds to flowchart steps
- Cohesion is a measure of how well the processing elements hang together as a module

- Cohesion of a module is
  - approximately the highest level of cohesion which is applicable to all elements of processing in the module

Levels of Lack of Cohesion

- Coincidental
  - No rhyme or reason for doing 2 things in the same sub-routine
    - void computeAndRead(double x, double& sqrtX, char& c);
- Logical
  - Similar class of things
    - char input(bool fromFile, bool fromStdin);
- Temporal
  - Things that happen one after the other
    - void initSimulationAndPrepareFirst()

Levels of Lack of Cohesion (cont'd)

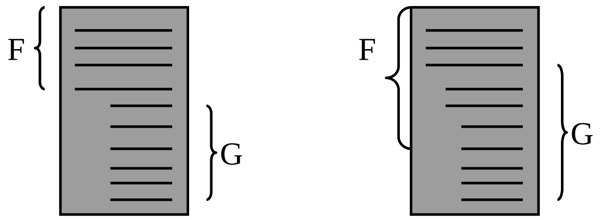
- Procedural
  - Operation are together because they are in the same loop or decision process (but no higher cohesion exists)
  - typeDecide(m)
    - Decide type of plant being simulated and perform simulation part 1.
- Communicational
  - All operations are on the same set of input data, or produce the same set of output data
    - void printReport(data x, data y, data z)
- Sequential
  - A sequence of steps that take the output of the previous step and process it into input for the next step.
    - string compile(String program) {
      parse, semantic analysis, code generation }

Cohesion (cont'd)

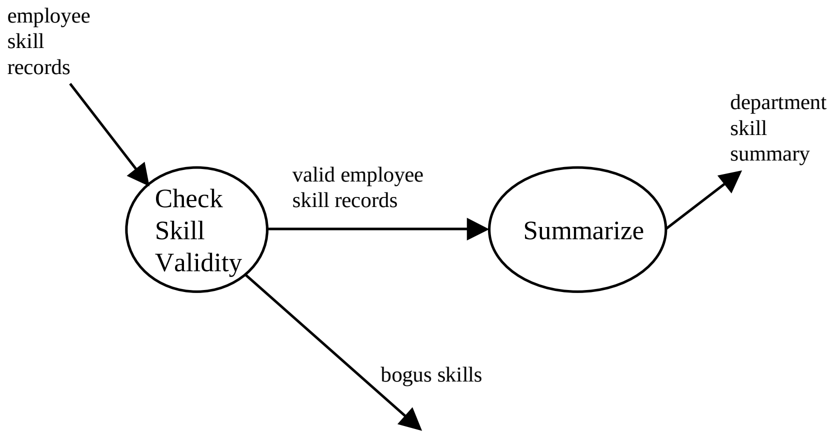
- Functional
  - That which is none of the above
    - double sqrt(double x);
  - Does one and only one conceptual thing.
  - Equivalent to Information Hiding

Implementation and Cohesion

- Consider module FG that does two things: F and G
- When doing these things in the same module, chances are there is some common code than can be shared.
- If F and G have high cohesion, that's ok.
- Otherwise it becomes difficult to work with



Data Flow Diagrams (DFDs)



employee
skill
records

Check
Skill
Validity

valid employee
skill records

Summarize

department
skill
summary

bogus skills

Structured Design Methodology

- Transform Analysis
  - – Restate the problem as a data flow graph
  - – Identify Afferent and Efferent data elements
    - afferent: high-level input data, furthest removed from the physical input, which are still considered inputs
    - efferent: high-level output data, furthest removed from the physical output, which are still considered outputs
  - – Factor Afferent, Efferent, and Transform branches