Software Engineering

(SoftEng2.doc)

by

David S Burris

Ph.D. in Computing Science

Certified Computer Programmer

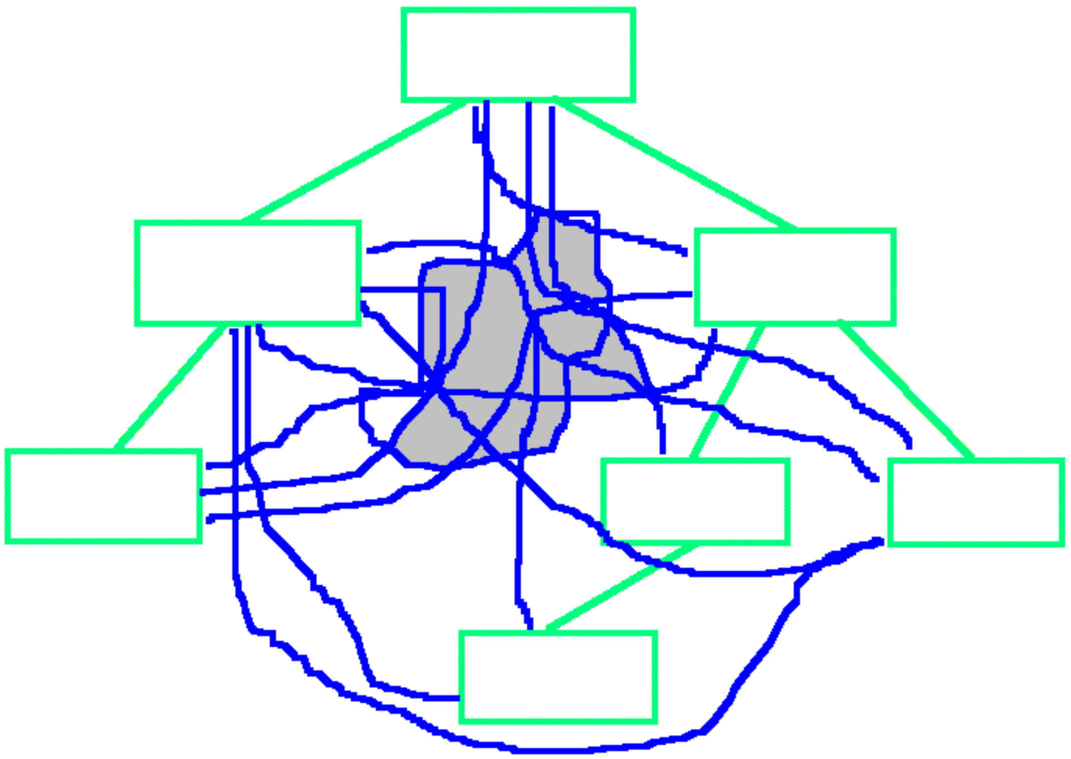
Certified Systems Professional

Certified Computing Professional

csc_dsb@shsu.edu

COUPLING

Coupling **is a measure of inter modular dependence. It may be because of shared content (ideas split across modules), shared data, or control.** ***As a design principle, it is desirable to minimize coupling.***



Example of Common-Environment (Data) Coupling

Minimized by design employing: single idea per module (separately implementable, maintainable, modifiable), black box concept.

Cohesion

Cohesion **is the cement that binds the processing elements of a module together. As a design principle, it is desirable to** ***maximize cohesion.***

1) **Coincidental:** **Little or no constructive relationship exists between module elements.**

2) **Logical:** **The processing elements can be thought of as falling into the same logical class of similar or related functions.**

3) **Temporal:** **The processing elements are related in that they must be executed within a limited window in time.**

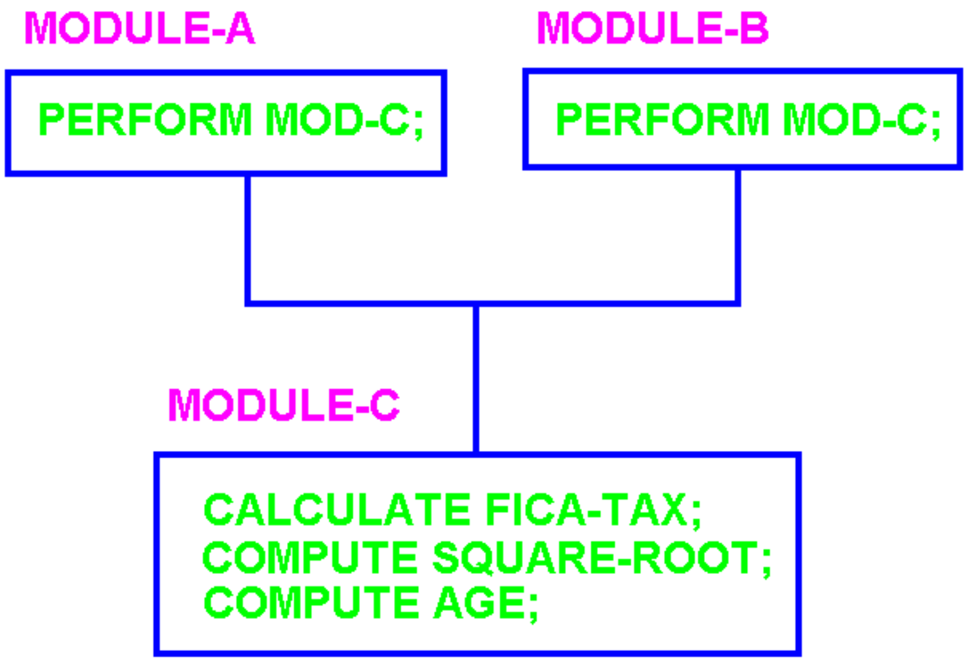
4) **Procedural:** **Associates processing elements on the basis of their procedural or algorithmic relationship. This frequently results from splitting a flow chart across algorithmic boundaries into modules. The modules often contain multiple ideas, parts of multiple ideas, or part of a single idea.**

5) **Communicational:** **All processing elements operate on the same input data set and / or produce the same output data.**

6) **Sequential:** **The outputs from one processing element serve as input data for the next processing element.**

7) **Functional:** **All elements of processing in the module are directly related to producing the result, and all elements necessary to produce the result are present in the module. No components not required exist in the module.**

1) **Coincidental:** **Little or no constructive relationship exists between module elements (frequently multiple partial ideas).**



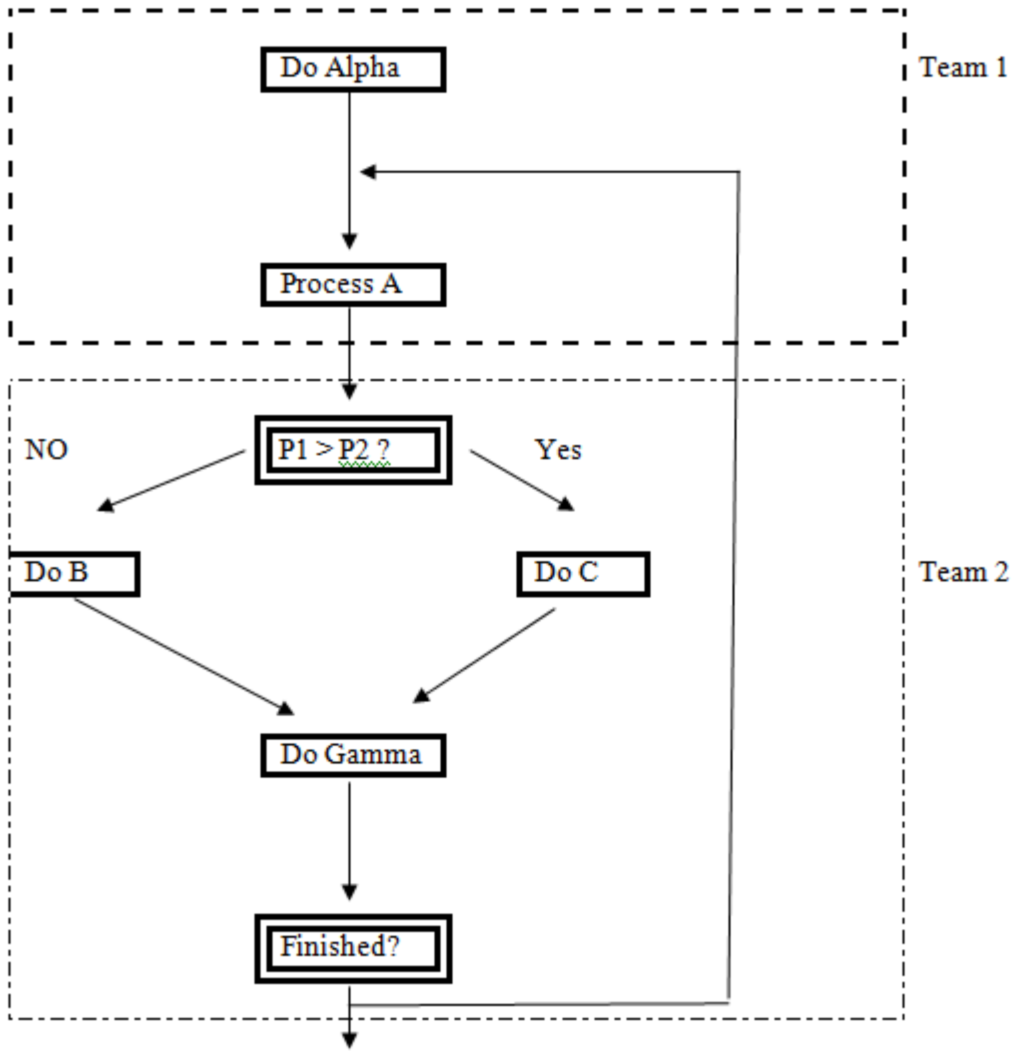
2) **Logical:** **The processing elements can be thought of as falling into the same logical class of similar or related functions.**

A) **INPUT-ALL MODULE;**
B) **EDIT-ALL-FIELDS;**

3) **Temporal:** **The processing elements are related in that they must be executed within a limited window in time.**

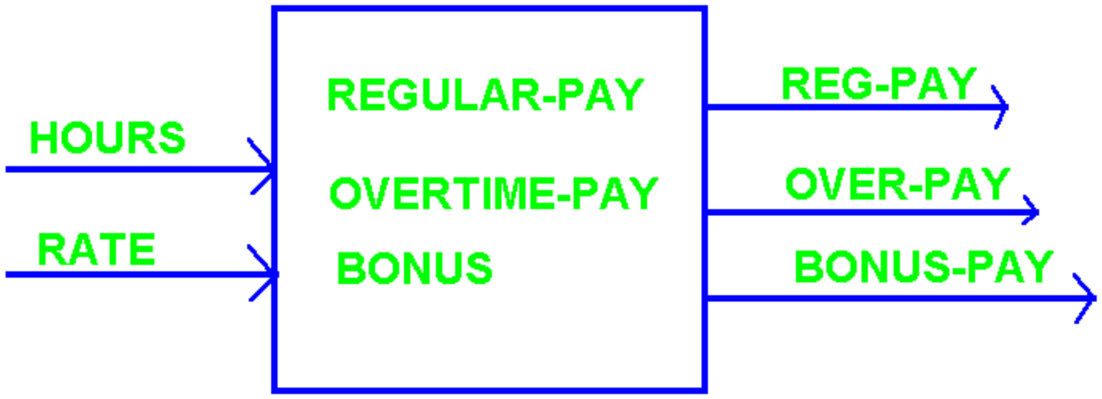
A) **Opening / Closing modules.**
B) **Time critical real-time module.**

4) **Procedural:** **Associates processing elements on the basis of their procedural or algorithmic relationship. This frequently results from splitting a flow chart across algorithmic boundaries into modules. The modules often contain multiple ideas, parts of multiple ideas, or part of a single idea.**

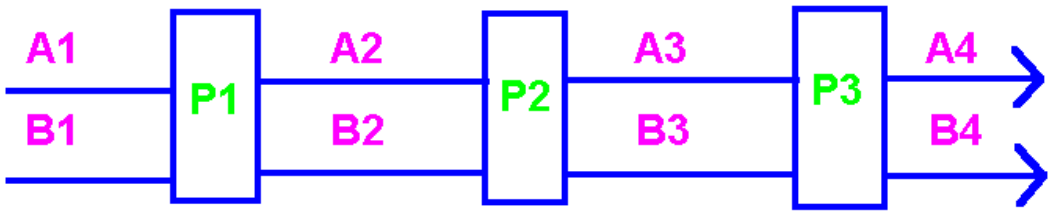


Typical goal: assign all teams an equivalent amount of work.

5) **Communicational:** **All processing elements operate on the same input data set and / or produce the same output data.**



6) **Sequential:** **The outputs from one processing element serve as input data for the next processing element.**



7) **Functional:** **All elements of processing in the module are directly related to producing the result, and all elements necessary to produce the result are present in the module. No components not required exist in the module.**

Relative strength of Cohesion

On a scale of one - to - ten:

0: Coincidental

1) Logical

3) Temporal

5) Procedural

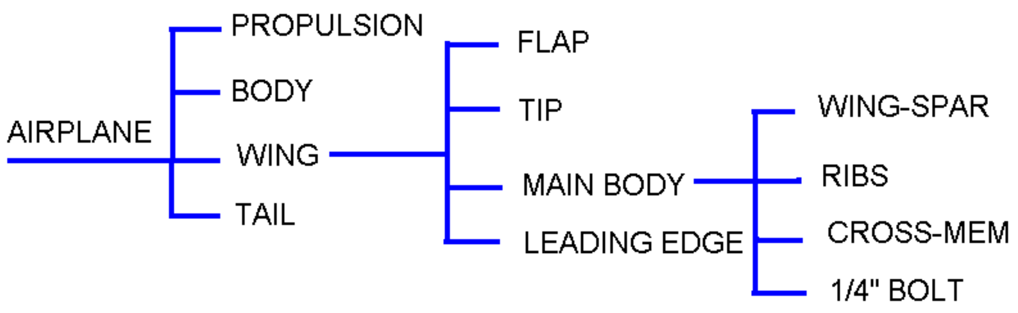
7) Communicational

9) Sequential

10) Functional

"*Structured Design, Fundamentals of a Discipline of Computer Program and System Design*," by Edward Yourdon / Larry L. Constantine, Prentice-Hall, 1979, ISBN 0-13-854471-9.

Top-Down Iterative Refinement



GUIDELINES:

1) **Begin by defining a single node to represent the system.**

2) **Identify the largest conceptual chunks of which the system is composed.**

3) **Define these to be the next level of decomposition.**

4) **Make sure that the level of detail present in the decomposition is consistent and correct.**

5) **Select a new node as a parent and proceed as in steps 2 through 4 until an acceptable level of detail is obtained. Ideally each chunk should contain a single concept and be separately implementable. Subdivision must cease when the resulting subdivisions can no longer be separately implementable.**

6) **Use each parent node as a statement of *what* is needed.**

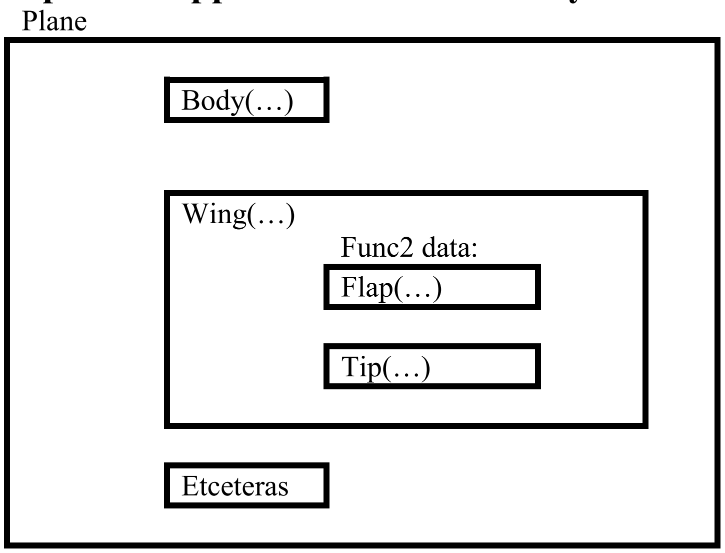
7) **Derive the child nodes by describing *how* the *what* will be accomplished or implemented.**

8) **Bear in mind there exists an implied sequence of operations from left to right. Label each node with a single verb noun combination.**

Direct support in Pascal (Wirth -1968) and Ada (1985). Support in "C/C++" libraries and code awkward. Supported awkwardly by object oriented languages.

Pascal specifically designed to provide support!

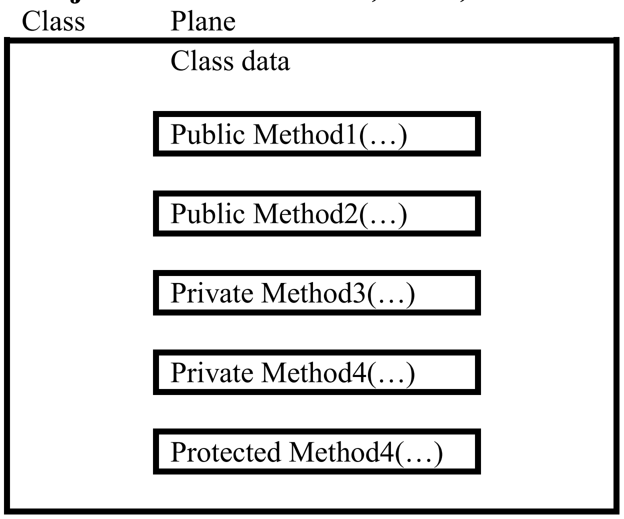
Specific support: Pascal/Modula by Nicolas Wirth. Ada great support.



Restricted access, limited scope. Function/procedure variables are protected. Function2 data available to Func3 and Func4 but not outside Func2.

Use of nested blocks.

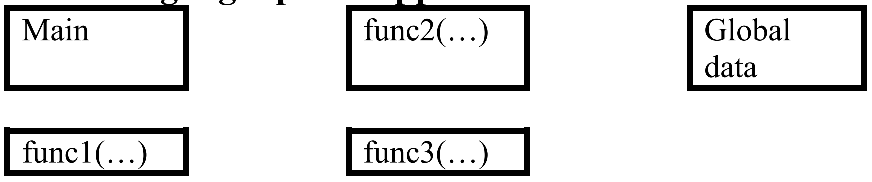
Object Oriented: Ada, C++, SmallTalk, limited support.



Class data global within class but not accessible outside class. Method variables are protected. Private/public limit access. Protected allows limited but dangerous access, especially during maintenance and modification.

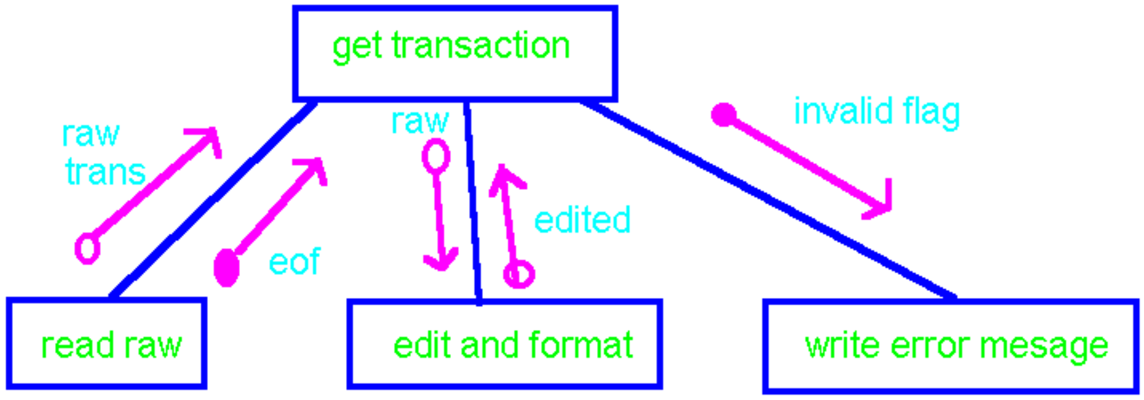
In general, no ability to nest blocks/methods.

"C" Language: poor support



All modules exhibit global access.

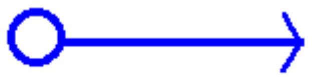
Functions stored separately or grouped into libraries. Uncontrolled global/external scope to functionality and data.



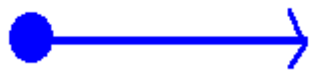
You are not finished with design until all elements of data and control have been specified.

Indicate:

1) data couple

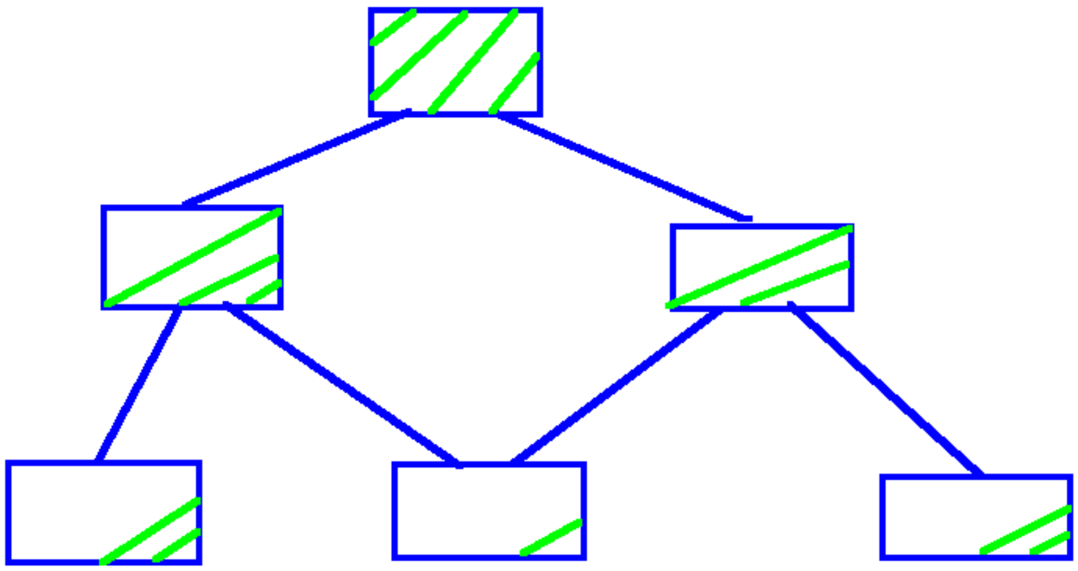


2) control couple



At a minimum, a design must break a problem up into its constituent parts. It must then show how the parts are related by control and data flow.

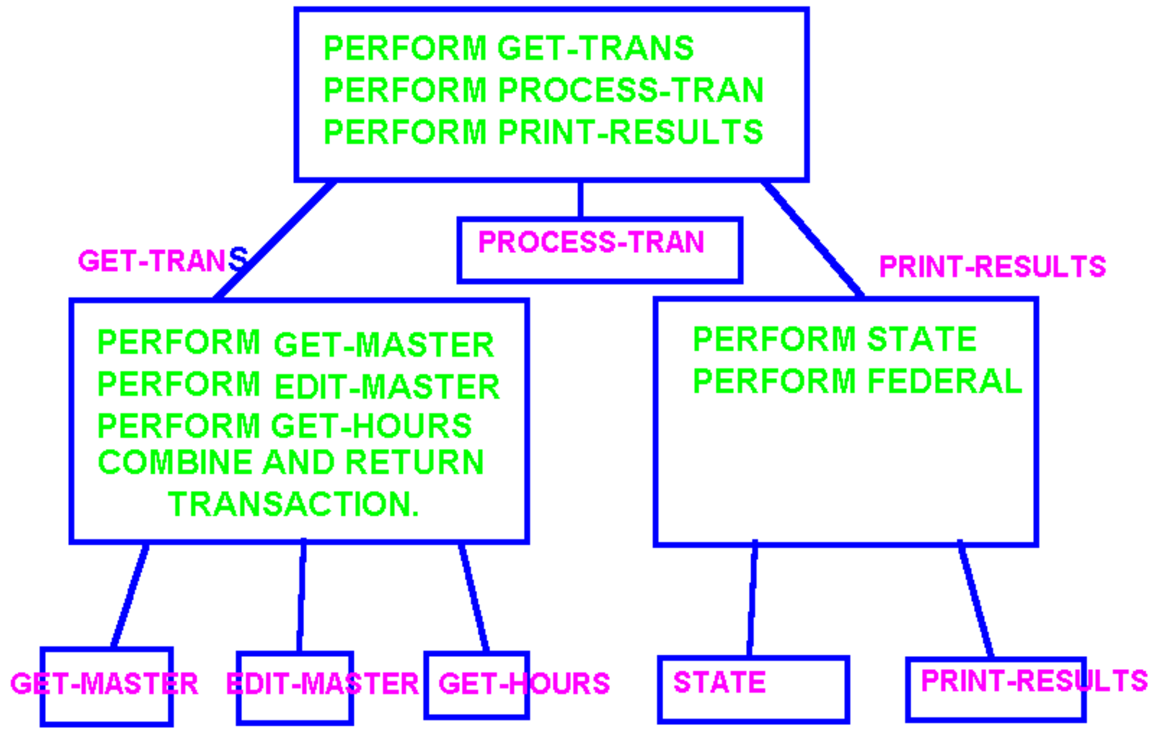
Factored Systems



A system in which the higher level modules are involved mostly in control and the lower level modules are mostly implementation details is said to be "HIGHLY FACTORED."

Software Psychology Examples: Political system, most work environments, many families, factored systems match our innate nature, etceteras.

Afferent and Efferent Data Flow



Afferent Input **Efferent Output**

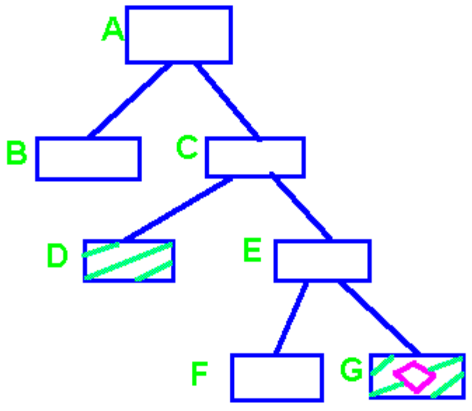
Data collected by subordinates and combined to form a complete transaction which is passed up to superordinates is referred to as "afferent" data flow. If data is taken from a superordinate module, filtered, and passed to subordinate modules, the process is termed efferent data flow. The terms afferent and efferent are taken from the field of biology by way of general systems theory. Afferent neurons carry sensor data from the body extremities inward and upward to the brain (central transform, processing organ). Efferent nerves carry the motor signals from the brain downward and outward to the body extremities.

SCOPE - -OF - EFFECT/CONTROL

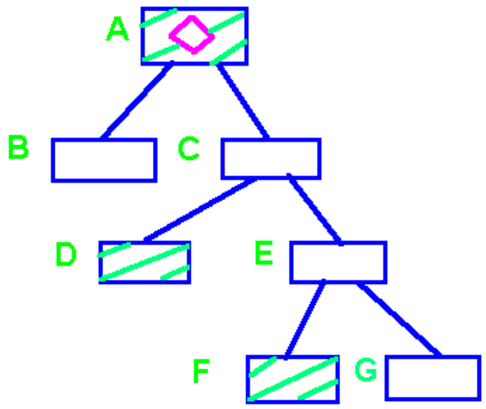
def1: The *scope of effect* of a decision is the collection of all modules containing any processing dependent on the decision.

def2: The *scope of control* of a decision is the module itself and "all" its subordinates.

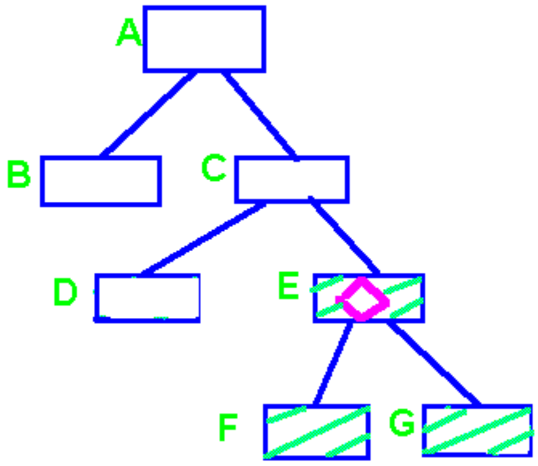
Design Goal: For any given decision, *the scope of effect should be a subset of the scope of control* in which the decision was made.



SCOPE OF CONTROL



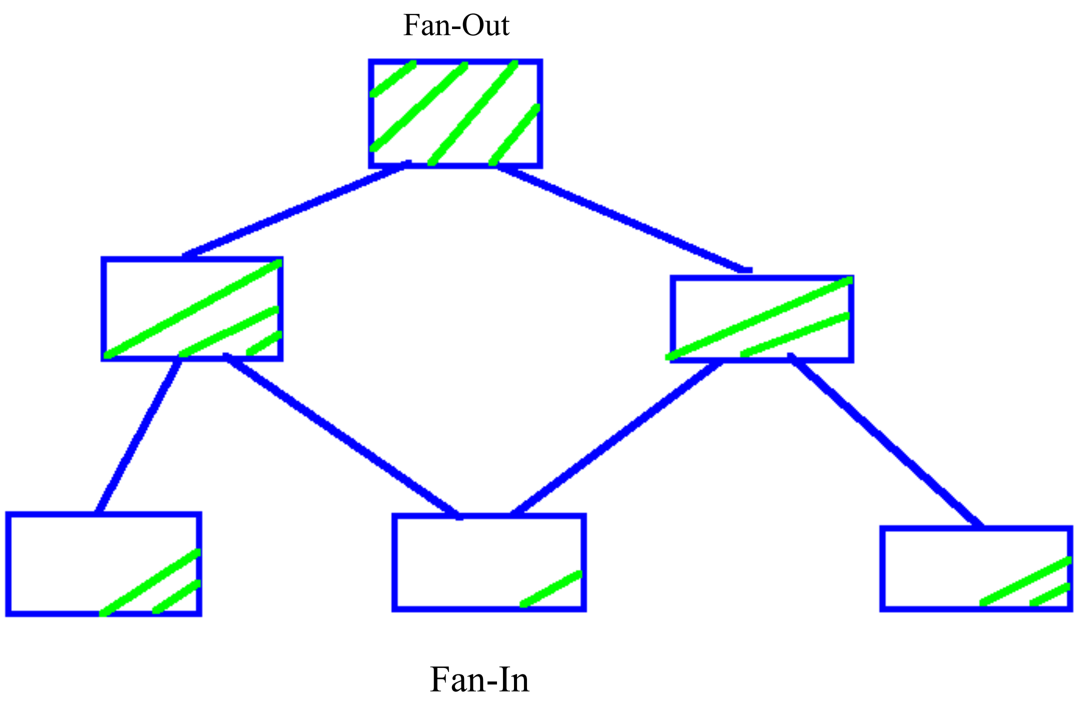
SCOPE OF EFFECT



IDEAL SCOPE

Preference by real developers for Scope over Listerine, Lavoris, etceteras

Clues to Complexity: Number of modules and depth of design



Good designs are typically quite deep compared to traditional design which stopped when it was felt the complexity had been reduced to a comprehendible level without showing all the details. The modules in a good design are typically smaller than traditional design efforts with more detail on dataflow, requirements, and restrictions/expectations.

The number of modules and depth however may not be good indicators of design quality. The more important questions on modularization tend to be centered on cohesion, coupling, and other metrics.

Average Fan-out:

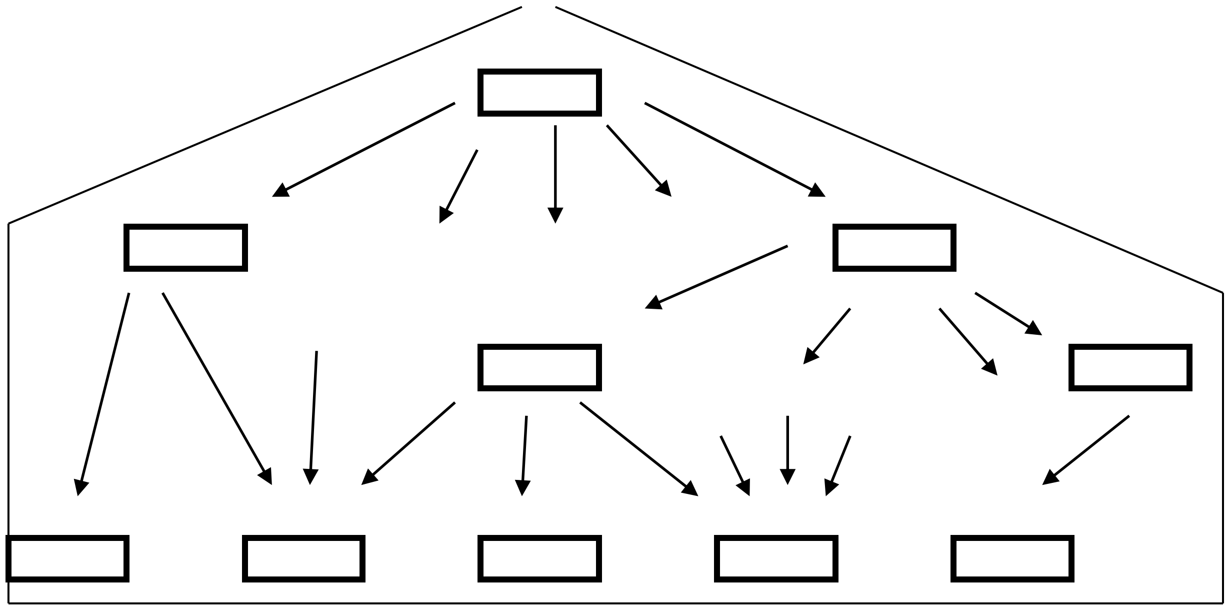
$$\text{Avg} = \frac{2 + 2 + 2}{3} = 2.$$ Rule of thumb is average of 3 to 4 – Miller's Law.

Fan-In:

High fan-in to black boxes implies cost reduction during: specification, design, implementation, maintenance, and modification.

Clues to Complexity: Number of modules and depth of design

Fan-In and Fan-Out



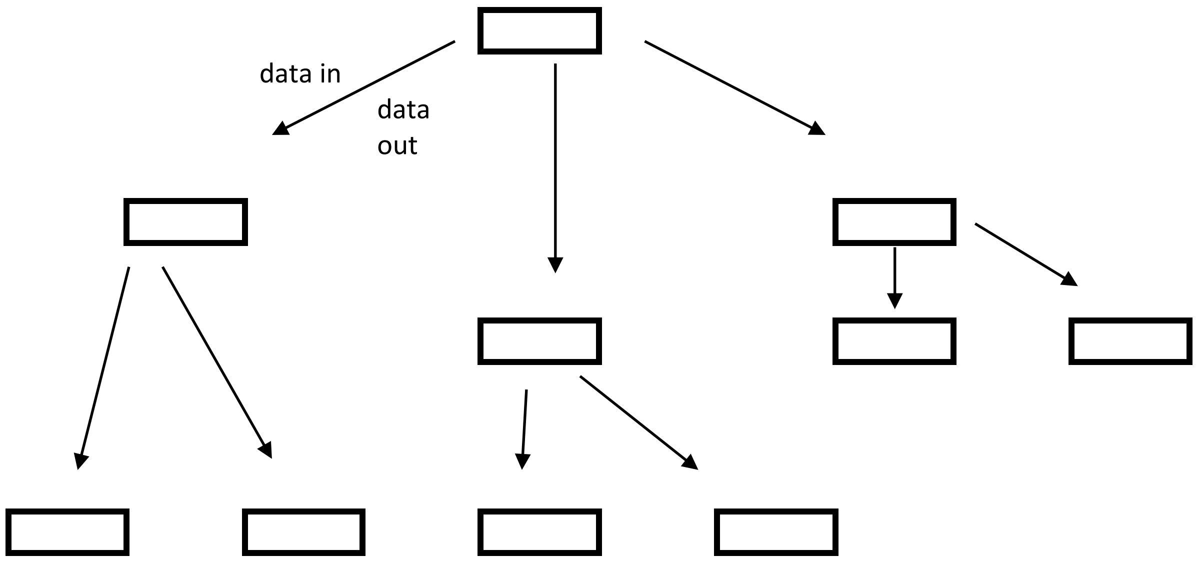
Good designs tend to be highly factored with upper modules mostly command and control with details in lower modules. Fan-out tends to be high initially with more fan-in deeper in the design. Ideally modules should contain single complete ideas – separately implementable.

Constantine: Cost figures from more than 100 medium sized systems developed for multiple organizations.

Some authors refer to this as a mosque structure.

Clues to Complexity: Number of modules and depth of design

Transform Centered Morphology



Afferent Branch Central Transform Efferent Branch

** Inputs and outputs should be indicated for each branch of control!

Afferent: Collect all raw data required for computation prior to starting computations.

Central Transform: Process collected data to produce results.

Efferent: Separate results and raw inputs routing them to appropriate destinations.

Given a choice between a Hierarchical, Mosaic/Network or OO or other design: the design reducing interconnections by obeying Graicunas Law with branching factors obeying Miller's Law while maximizing cohesion and minimizing coupling will almost always be the appropriate choice!

Data Flow Analysis

By

Edward Yourdon and Larry Constantine

1. **Restate the problem as a data flow graph**

2. **Identify the afferent and efferent data elements**

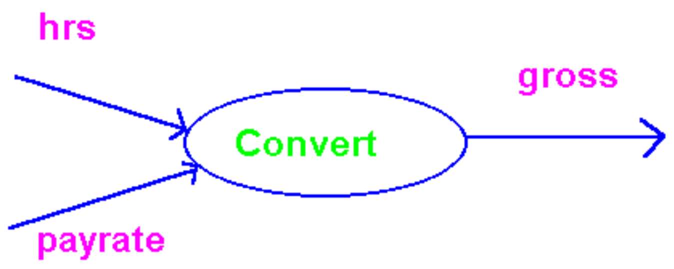
3. **First-level factoring**

4. **Factoring of afferent, efferent, and transform branches. Transform centered designs tend to be highly factored with sizable numbers of modules at intermediate levels in the hierarchy. Higher level and intermediate modules do very little "work" except to control and coordinate the work of subordinates.**

Based on studies by Larry Constantine of similar software projects, e.g., several accounts receivable systems. Some projects were relatively cost effective others relatively expensive. The cost effective projects exhibited similar characteristics. The hope was the similar characteristics exhibited by cost effective projects could be learned/taught and applied over a wide range of projects to reduce cost.

Maurice H. Halstead in *"Elements of Software Science:"*
"The structure of a program should reflect the structure of the information it processes!"

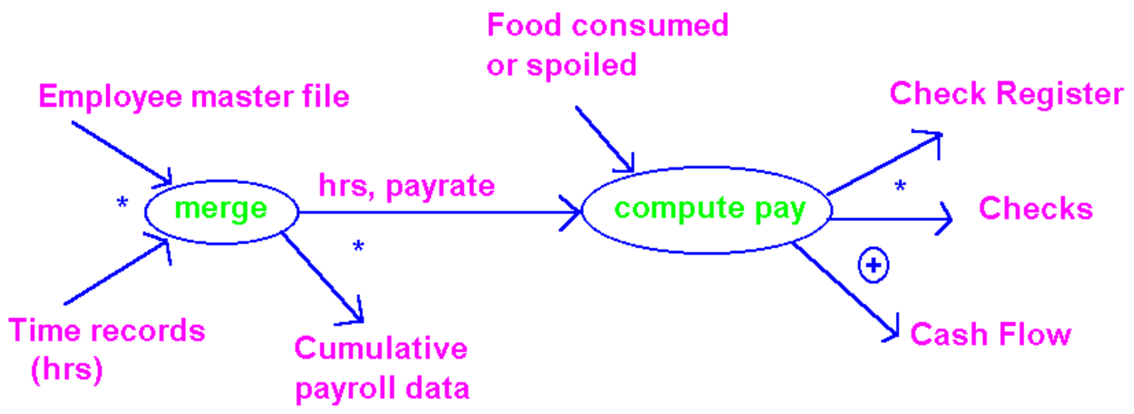
Data Flow Graphs



Combine hours worked and pay rate into gross pay.

During design concentrate on identifying what is required, the necessary inputs and desired outputs. Do not worry about how to eventually implement the required task.

Assume you own a short order food store and need to either produce the payroll or a cash flow report. You charge employees for food consumed on the job or destroyed through carelessness or accident.



⊕ implies "OR"

* implies "AND"

It is better to show too much detail than not enough.

STEPS TO DRAWING DATA FLOW GRAPHS:

1. **It is essential not to become entangled in aspects of procedure or decision-making. The designer should not worry about, and the graph should not show such things as:**

   a) **loops,**
   b) **initialization routines,**
   c) **termination routines,**
   d) **recovery procedures,**
   e) **error detection should be thought of as a filter which separates good data from bad data, not a decision branch.**

2. **Label the data elements carefully with descriptive names as data elements enter and leave a transform bubble. If a designer has difficulty finding a succulent name for a function, it may be a clue that the module should not exist or lacks cohesion.**

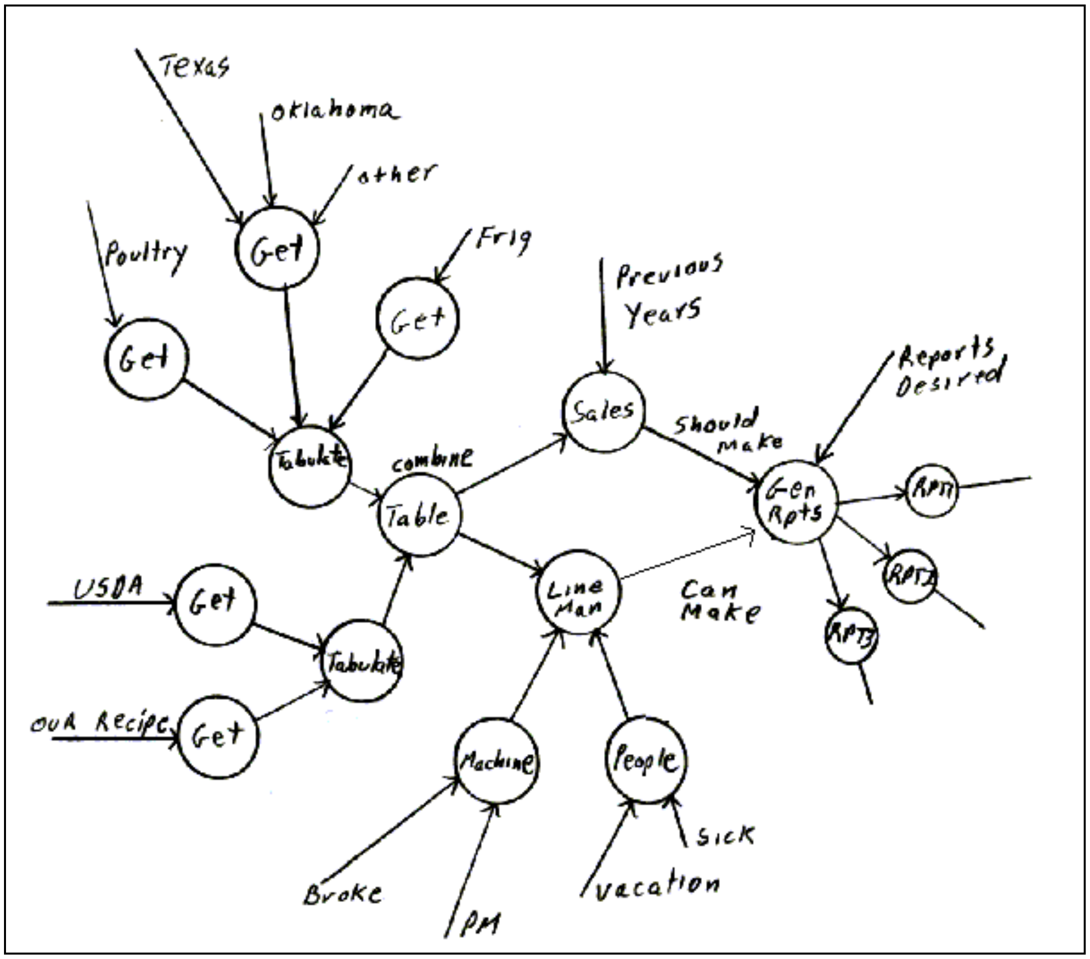
3. **Use "and" and "or" operators where appropriate in diagrams.**

4. **Make sure the data flow is correct for the level of detail being modeled. If in Doubt, show too much detail rather than too little.**

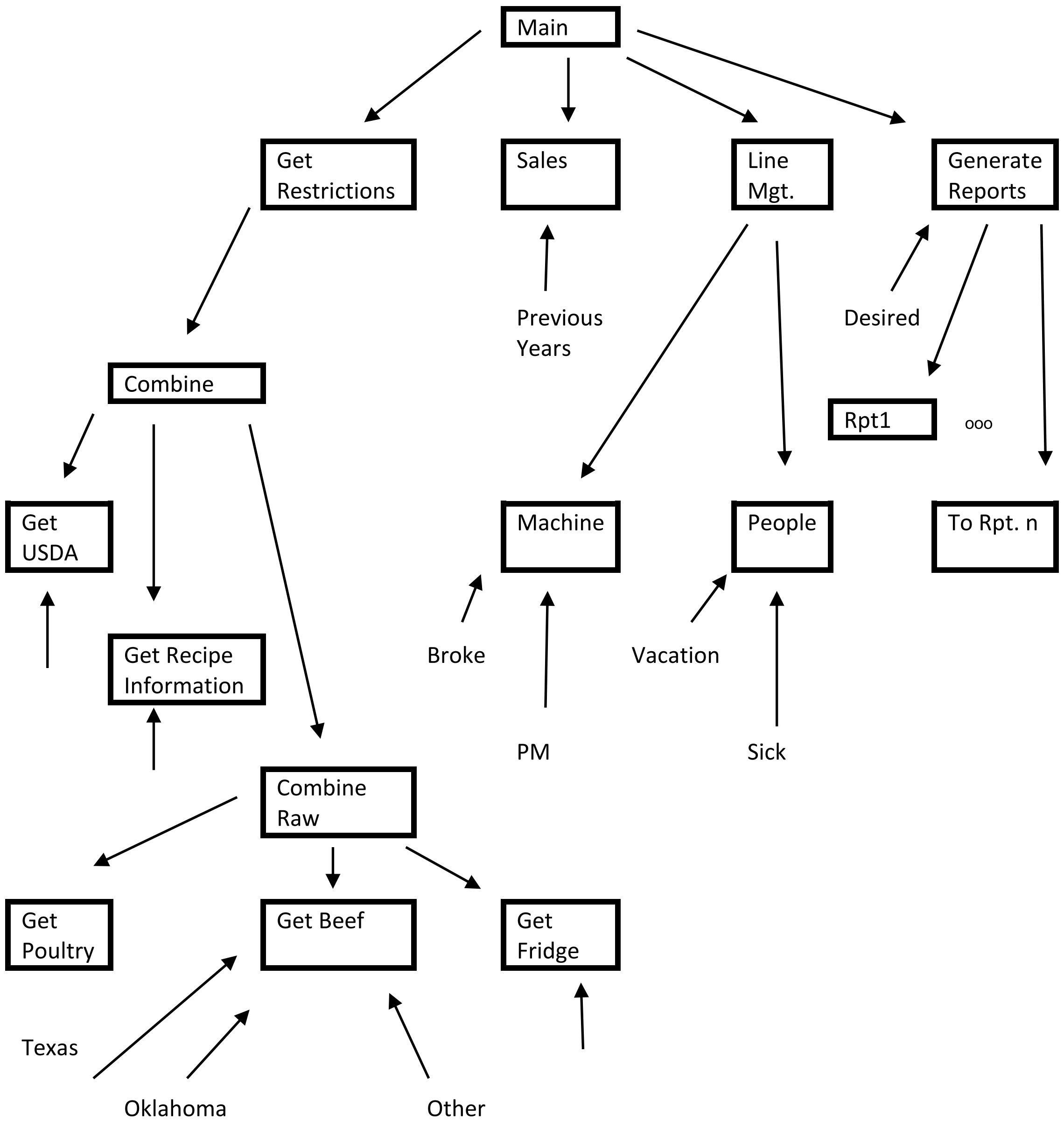
Def: "afferent" data elements are those highest-level elements of data that are furthest removed from physical input, yet still constitute real inputs to the system.

Def: "efferent" data elements are the furthest removed from the physical outputs which may still be regarded as outgoing.

Ask Sales to project what process meats we should manufacture based on last year's sales, USDA Guidelines, company recipes, raw materials available at the stock yards, and left-over meat from last month in the refrigerator. Remember we must consider the available personnel and machines for use in production.



Factored System



PACKAGING

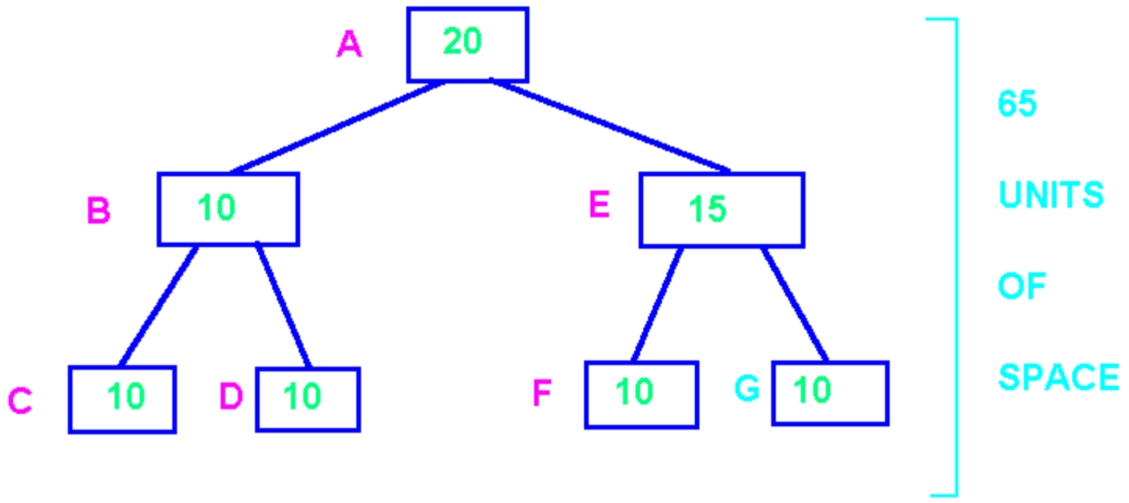
Ultimately we must make a system fit into the available physical memory. Normally we wish to simultaneously minimize execution time.

In general it is not possible to simultaneously reduce both memory utilization and run time. These are frequently mutually exclusive goals.

Traditionally, memory constraints have guided the design from inception to finish. *Today it is recognized that deferring packaging until the end of the design process improves our ability to maximize memory utilization*.

Minimizing the use of main memory *requires modules with high cohesion and low coupling*. Initially, the design should emphasize high cohesion and low coupling without consideration of memory limitations.

OVERLAY SCHEME



OPTIMUM SPACE = 85 UNITS.

DESIRED SPACE = 65 units.

MINIMUM SPACE = 45 units.

BEST COMPROMISE = 55 units.

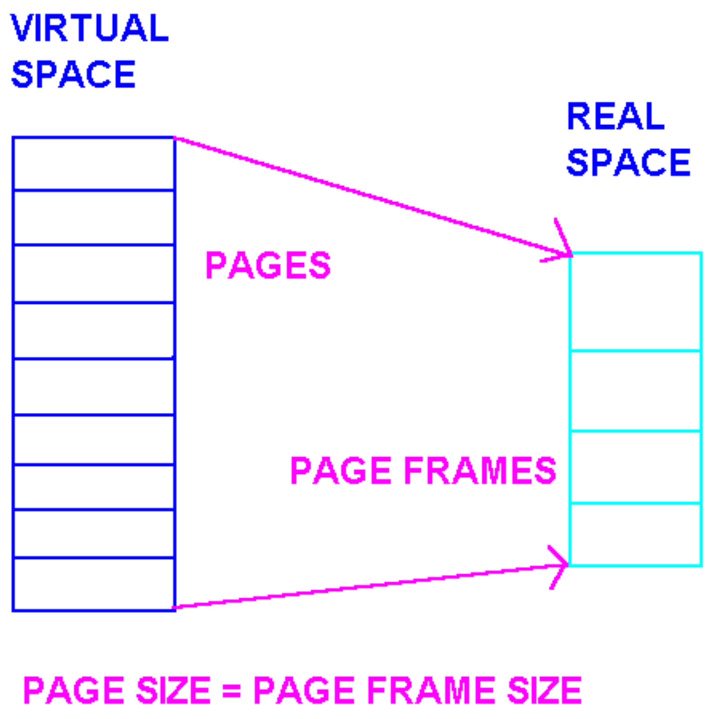
Note an overlay executive must exist at run time and be invoked on each module reference to determine if the overlay is in main memory or auxiliary storage. Upon a fault, the module in memory, at least any data areas, will potentially have to be swapped to auxiliary storage before the overlay is swapped in.

Changes to the overlay due to expansion/compression during original development, maintenance (errors), modification (enhancement or changes in environment).

//OVERLAY B,E; C,D,F,G;

VIRTUAL SYSTEMS

Paged / Segmented / Paged-Segmented



VIRTUAL SPACE >> REAL SPACE

Dynamic address translation of every instruction works because:

1) Programs and data tend to be sequential.

2) Programmers group related code in the same modules (functions, procedures, methods, classes).

3) Loops.

Faults: a) when real memory has free space; b) real memory is full (Least Recently Used {LRU}, FIFO, LIFO, RANDOM, working set, etceteras)

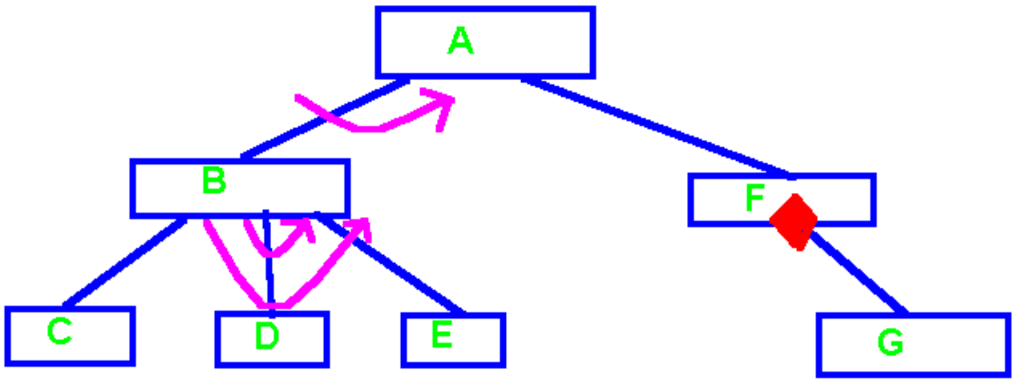
Note 50% or more of large software application are not utilized normally required by the user – word processors, spread sheets, operating systems.

BASIC GUIDELINES

Assume a design exhibiting high cohesion and low coupling.

After the design has been completed, we have the maximum information about procedural and communicational structure. This information may be use to determine which modules should be placed in the same overlay / page. <u>This placement may be accomplished by physical arrangement of the text (code) prior to compilation, or by the appropriate use of a linkage editor</u>.

As a basic guideline, include in the same load unit modules that have a high frequency of reference.



GROUPING CRITERIA:

1) **Iteration producing high frequency of reference. Give priority to inner nesting.**

2) **Volume of reference is more important when available. Example: Process A to F for all 200,000 records but A to B only 50 times.**

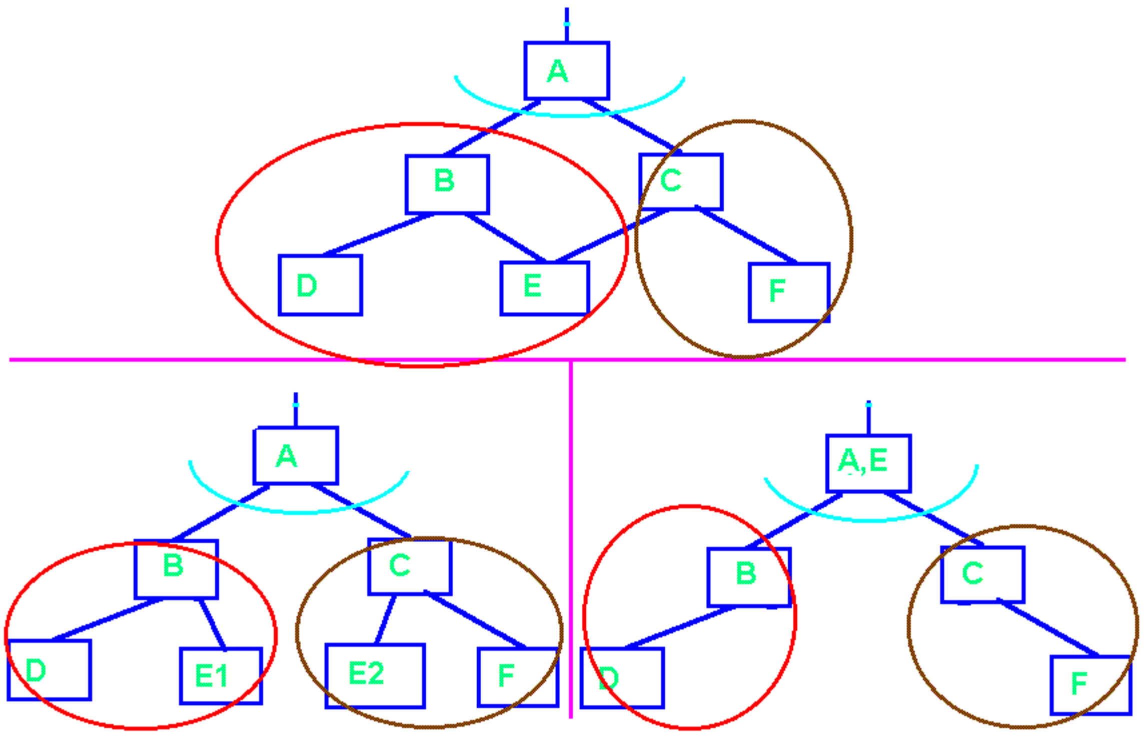
3) **Less useful is the time interval between references, e.g., assume 15 milliseconds is spent in each of F and G versus a design where 20 minutes is spent in each of F and G. Assume the overhead for a fault is 30 milliseconds.**

4) **Do not include in the same load unit modules that are not normally needed (infrequently used error routines or modules whose use is rare) with high use modules.**

5) **Do not package one shot routines such as *initialization* or *closing* with routines that remain in memory for most of the program execution time.**

6) **Sorts with large input and / or output times offer a natural break point.**

7) Main memory space may be minimized and execution time reduced by duplicating code frequently used as a target for fan-in.



Duplicated **Factored**

Special Considerations for Virtual Memory Environments:

1) Avoid branch statements over large distances. Group logically related modules close physically.

2) Be careful of using a ***binary search*** on tables that cross many page boundaries. Since each probe is at the half way point of what is left, each probe may incur a fault.

3) Beware looping routines that improperly cross page boundaries.

   Assume that there is only enough memory for three data pages at a time after the program is loaded.

   A,B,C: ARRAY(1..100, 1..100) OF INTEGER;
   -- Assume 100 integers per page, and each row
   -- occupies one page (lexicographic/row major storage order).

```
for i in 1.. 100 loop          //traverse across row
    for j in 1..100 loop
        A(i,j) = B(i,j) + C(i,j);
    end loop;
end loop;
```

-- num faults = 3 per row * 100 rows = 300 faults.

```
for j in 1.. 100 loop          //traverse down col
    for i in 1..100 loop
        A(i,j) = B(i,j) + C(i,j);
    end loop;
end loop;
```

-- num faults = 3 per element in a col * 100 rows * 100 cols = 30,000 faults.

OPTIMIZATION FOR TIME

Optimization for time should be considered *only after the system has been completely designed and should never influence the design itself*.

In general it is easier and cheaper to make working software run faster than it is to make fast software work.

When designers/programmers optimize as they work, they do local optimization. Humans have a limited capacity for the amount of information they can retain at one time. As a result, local optimizations may actually slow the overall system down. Waiting until the design has been completed allows for global optimization. Global optimization normally results in greater efficiency gains than local optimization and at less cost.

In general, *optimization for time requires* that the design of the system to consist of highly cohesive modules with low coupling to the rest of the system.

"Structured Design, Fundamentals of a Discipline of Computer Program and Systems Design," by Edward Yourdon andLarry L. Constantine, pp. 290-304.

"A Emperical Study of FORTRAN Programs," by Donald Knuth, *Software Practice and Experience*, Vol. 1, No. 2 (April-June 1971), pp. 105-133.

Points of interest:

1) **Execution time is not uniformly distributed over the code.** Knuth's studies showed that only about 5% of the code in a typical system accounts for 50% of the CPU time. Unfortunately these bottle necks can only be detected normally while executing "real" data.

2) **Optimization is irrelevant if the program does not work.** Optimization frequently introduces extra flags, complexity, and increased bugs. Properly designed systems actually aid optimization for time.

3) **Over emphasis on optimization tends to detract from other goals such as code clarity and ease of use.** The increased complexity frequently makes it very difficult to debug the original software and makes maintenance and modification untenable.

4) **The simple way is frequently the most efficient way.** Intricate , sophisticated tricks lead to:

    a) complicated rats nest,
    b) "dead" code, computations irrelevant to the task at hand, and
    c) multiple computations of the same thing.

5) **Systems exhibiting high cohesion and low coupling can be easily optimized.** Changes can be made in independently implemented modules without a snowballing affect throughout the rest of the system.

6) **The efficiency of a system depends on the competence of the designer.** It is better to have a small competent design team than a large group of average practitioners.

Organized Approach to Optimization for Time

1) **Determine the execution time for each module.** This may be done using a hardware monitor or software instrumented code. Refer to the execution time for module "*i*" as T(i).

2) **Examine each module and estimate its potential for improvement**, P(i). For example a sequential search of a table containing N items has time proportional to (N+1)/2. A binary search of the same table would have a search time proportional to log base 2 of n. Sorting algorithms are data sensitive (random versus ordered).

3) **Estimate the cost of the improvement**, C(i). For example, to switch from a sequential to a binary search of the above table might require the table to be sorted. The search code might be available in a standardized re-usable code library. If not it would have to be coded, see a text on data structures such as "Fundamental Algorithms," by Donald Knuth, ISBN 0-201-03809-9.

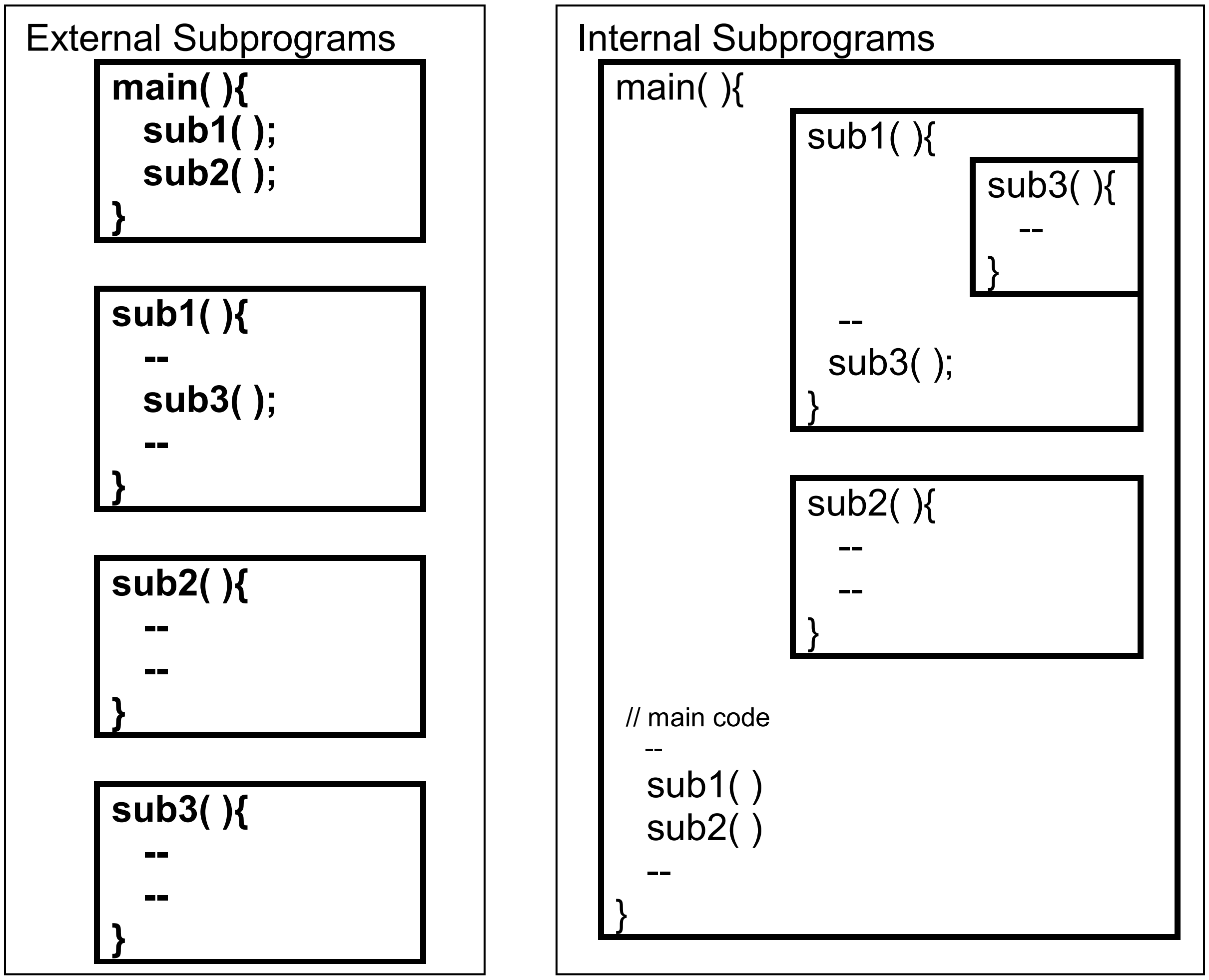
4) **Establish priorities for the optimization process**, PR(i). The highest priority module should give the greatest improvement in performance for the least cost. Calculate PR(i) = A*P(i)*T(i) - B*C(i), where A and B are appropriate weighting functions.

5) **Optimize the modules as required starting with the highest priority module.** Only optimize as many modules as are required to meet the specifications.

   **** If you cannot meet the specifications by optimizing the modules determined in step one to be the bottle necks, STOP, the system should probably be abandoned and another approach to solving the problem sought. No matter how much you want to make the system work, it is probably not possible. Some modules should have a negative optimization priority (frequently modified / complex)!

Other considerations.

1) External versus Internal Subprograms.



```
main( ){
  sub1( );
  sub2( );
}

sub1( ){
  --
  sub3( );
  --
}

sub2( ){
  --
  --
}

sub3( ){
  --
  --
}
```

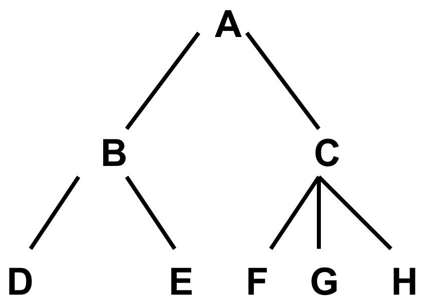
```
main( ){
            sub1( ){
                        sub3( ){
                          --
                        }
              --
              sub3( );
            }

            sub2( ){
              --
              --
            }

 // main code
  --
  sub1( )
  sub2( )
  --
}
```

External routines are compiled at separate times. In general, the translator must prepare to pass all parameters, save all registers when the subprogram is invoked, perform the work, restore the registers, and return to the point of invocation.

With internal routines, all code is available. If not all registers are in use, it may not be necessary to save any registers.

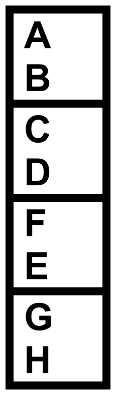
External routines may be shared with other programs. Internal routines may not be shared. This sharing may be an advantage or a disadvantage. Internal routines should be in the same page.

2) **The manner in which external routines are grouped in memory has a significant effect on performance.**
Grouping can be done by sequencing all code in the same file for the language translator or by using a linkage editor.

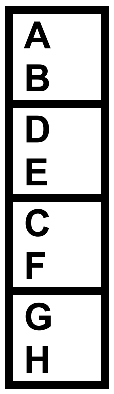


Assume that all modules are the same size. Assume a paged virtual computer with sufficient space in a page to hold to program modules. Now assume the following groupings of program modules to pages.

Grouping 1



Grouping 2

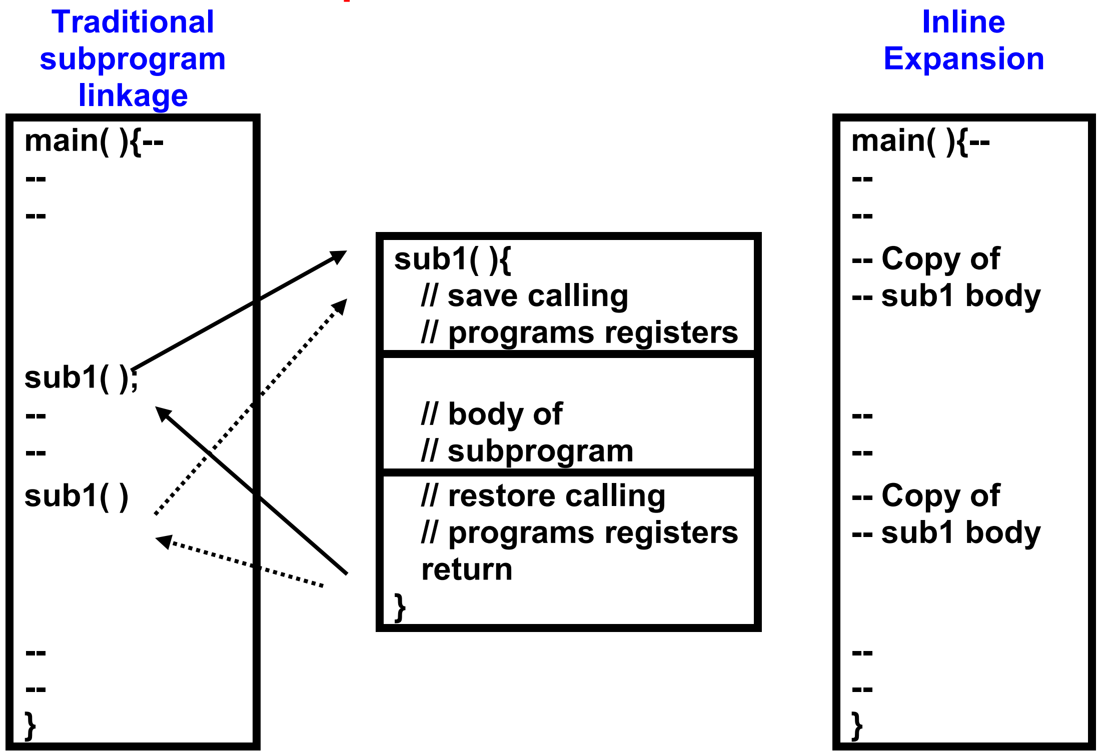


For grouping 1, the tree branch ABDE requires three pages of real memory for efficient execution provided control moves between all modules in the hierarchy. The tree branch ACFGH requires four pages of memory.

The branch ABDE only requires two pages of real memory using grouping 2. The branch ACFGH only requires three pages of memory.

The same arguments work for overlay schemes.

3) Inline code expansion.



In traditional subprogram linkage the subprogram must:

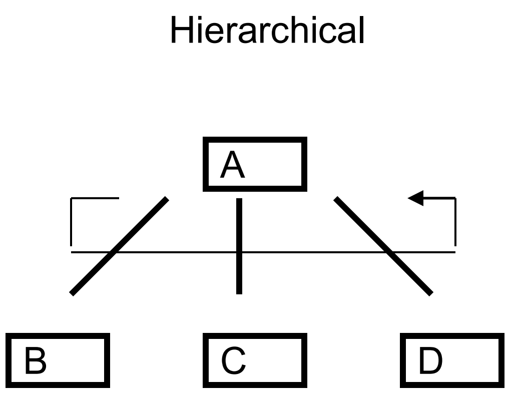
1) Prepare parameter list and return address prior to branch.
2) First save the calling programs registers (usually in a stack) as they must share the registers.
3) Second, perform the work of the subprogram.
4) Restore the calling program's registers so that any intermediate results or addresses in the registers are available upon return.
5) Return to the calling program.

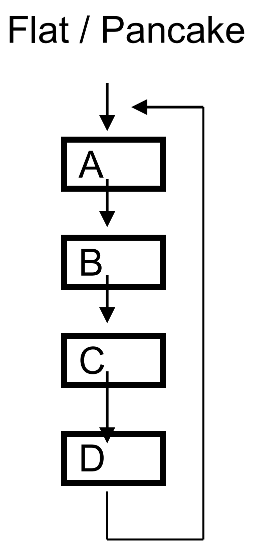
This can represent substantial overhead but is necessary as the main and subprogram are compiled at different times and must share a limited pool of registers.

In inline expansion, the subprogram linkage is replaced by copying the subprogram body at each location it is referenced. Since the translator has complete knowledge of all current register assignments, it may be possible to utilize free registers avoiding the save restore process. It is not necessary to pass a return address to the subprogram or to affect the return. All code is sequential.

Classic space / time trade off, the inline code runs faster but you now have multiple copies. Note copies may be in auxillary storage as opposed to main memory allowing efficient swapping when associated code is loaded.

4) Pancake Structure to Reduce Subprogram Linkage Overhead.





5) Global (Pathological) versus Normal Parameter Passing

(High versus low data coupling)

Pathological

```
int p1, p2, p3;

main( ){
   int a, b = 3, c = 5;
   sub1(a, b, c);
}

void sub1(int x, int y, int z ){
   x = y + z / p1 – p2 * p3;
}
```

Normal

```
main( ){
   int p1, p2, p3;
   int a, b = 3, c = 5;
   sub1(a, b, c, p1, p2, p3);
}

void sub1(int x, int y, int z,
          int v1, int v2, int v3){
   x = y + z / v1 – v2 * v3;
}
```

Most translators can actually generate slightly faster executing code using pathological (global) variables as they do not have to pass the address through a parameter list and reference the variable via the parameter list. However this introduces all the traditional problems associated with high data coupling.

6) Compile Time Binding Versus Run Time Binding (Polymorphism).

Also call-by-value, call-by-reference, and call-by-name.

```
Class Pet{
   // Data structures

   Speak( ){ cout << “Cough, cough!”; }
   // other methods
}

Class Dog inherits Pet{
   // Data structures specific to a dog

    Speak( ){ cout << “Bow Wow!”; }
}

Class Cat inherits Pet{
    //Data structures specific to a cat

    Speak( ){ cout << “Meow, purr!”; }
}


void main( ){
    Dog aDog;
    Cat aCat;
    Pet Menagerie[4];
    Menagerie[0] = new Pet;
    Menagerie[1] = new Dog;
    Menagerie[2] = new Cat
    Menagerie[3] = new Dog;

    aDog.speak( );  // Compile time bind, “Bow Wow!”
    aCat.speak( );   // Compile time bind, “Meow, purr!”

    for(int k = 0; k < 4; k++){
        Menagerie[k]→ Speak( );  // Run time bind! – language dependent
                                 // Ada and Java versus C++ .  Compile
    }            // time bind always “Cough, cough!”  A C++ programmer must
}               // specify run time binding over default compile time!
```

7) Data Type Conversion

```
A, B: Integer;
C, D: Float;

A := A + C * B – C;
```

This is equivalent to:

```
A := Integer( Float( A ) + C * Float( B ) – C );
```

Time lost in conversions can more than double program run time.
Use the same data type (and fastest) for all variables when possible.
“pic 99” implies 2 digits of accuracy.

COBOL:

```
01    A    pic    99.
01    B    pic    99.
01    C    pic    99.

Compute A = B + C.
```

The default storage for COBOL is printer (character) format. The reason is a value is typically calculated once then printed multiple times in different reports. Hence A and B must be converted to numeric prior to the addition then back to character format when assigned to A. Solution:

```
01    A    pic    99 usage is computational.
01    B    pic    99 usage is computational.
01    C    pic    99 usage is computational.
```

8) Poorly Directed Junior Programmer

We asked a newly assigned junior programmer to fix an error in a module just to keep them out of the way while meeting a deadline. He spent a week when we expected a day. He announced that not only have they fixed the error but have optimized the module so that it runs 60% faster. What are you going to say? You did not check on him, you didn't have the time. You cannot admit that his effort was worthless as the module only consumed about 1 / 34,896,987 of a second on a system that runs for 5 hours. I sure did not want to tell my boss what had happened!

9) Optimizing Compiler / Translator

Use an optimizing translator to look for loop invariants, subscript optimization, and other possible optimizations automatically.

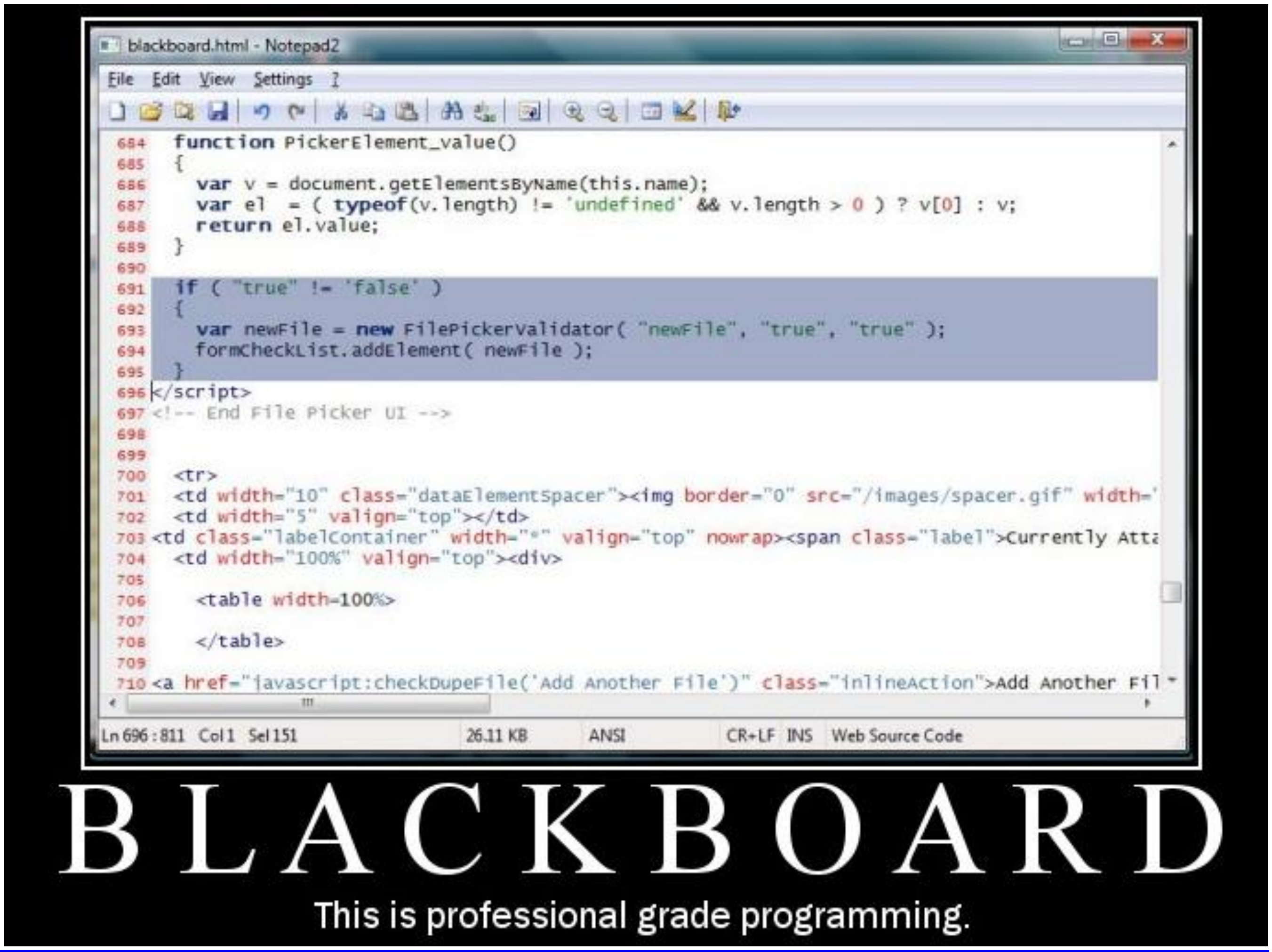
```
for( int k = 1; k < 896; k++) {
        J = 6;
        A[k] = A[k] + J * 12 + 32;
}
```

Changed by translator too:

```
J = 6;
Temp = J * 12 + 32;
for( int k = 1; k <= 1000; k++) {
        A[k] = A[k] + Temp;
}
```

Assume that "+" requires 1 machine cycle and "*" requires 30 machine cycles. Then the overhead for the first loop for **A[k] +J * 12 + 32** is (1 + 30 + 1) * 1000 = 32000. The overhead in the loop for the improved code is 1 * 1000 = 1000. This is a tremendous reduction in run time while potentially preserving the readability of the first section of code. Catch: optimizing compilers frequently obscure assembly language equivalents of code used by debuggers and sometimes generate undesirable side effects, the code does not execute as expected with strange results.

Found while debugging a problem in Blackboard:


blackboard.html - Notepad2
File Edit View Settings ?
Ln 696 : 811 Col 1 Sel 151 26.11 KB ANSI CR+LF INS Web Source Code
BLACKBOARD
This is professional grade programming.

```
function PickerElement_value()
{
  var v = document.getElementsByName(this.name);
  var el  = ( typeof(v.length) != 'undefined' && v.length > 0 ) ? v[0] : v;
  return el.value;
}

if ( "true" != 'false' )
{
  var newFile = new FilePickerValidator( "newFile", "true", "true" );
  formCheckList.addElement( newFile );
}
</script>
<!-- End File Picker UI -->


  <tr>
  <td width="10" class="dataElementSpacer"><img border="0" src="/images/spacer.gif" width='
  <td width="5" valign="top"></td>
<td class="labelContainer" width="*" valign="top" nowrap><span class="label">Currently Atta
  <td width="100%" valign="top"><div>

    <table width=100%>

    </table>

<a href="javascript:checkDupeFile('Add Another File')" class="inlineAction">Add Another Fil
```

If (“true” != “false”{
    // body
}

Example 2: Texaco program (found by manager Keith Wall) consisting of about 30 pages of green-bar code in a single nested if starting on about the 5 lines of the program.

Example 3: The errors in this code actually combine to make it work (contributed by Stephen Rugh after seeing it at http://cvmountain.com/2011/09/whats-wrong-with-this-code-really/). It removes all the tab pages from a collection.

Here are five lines of code I found during a review not too long ago. This code had been tested and was ready for release.

```
for ( int i=0 ; i < this.MyControl.TabPages.Count ; i++ )
{
  this.MyControl.TabPages.Remove ( this.MyControl.TabPages[i] );
  i--;
}
```

Feel free to stare at this for a while and absorb the quality.

Inefficient programmers tend to experiment randomly until they find a combination that seems to work.

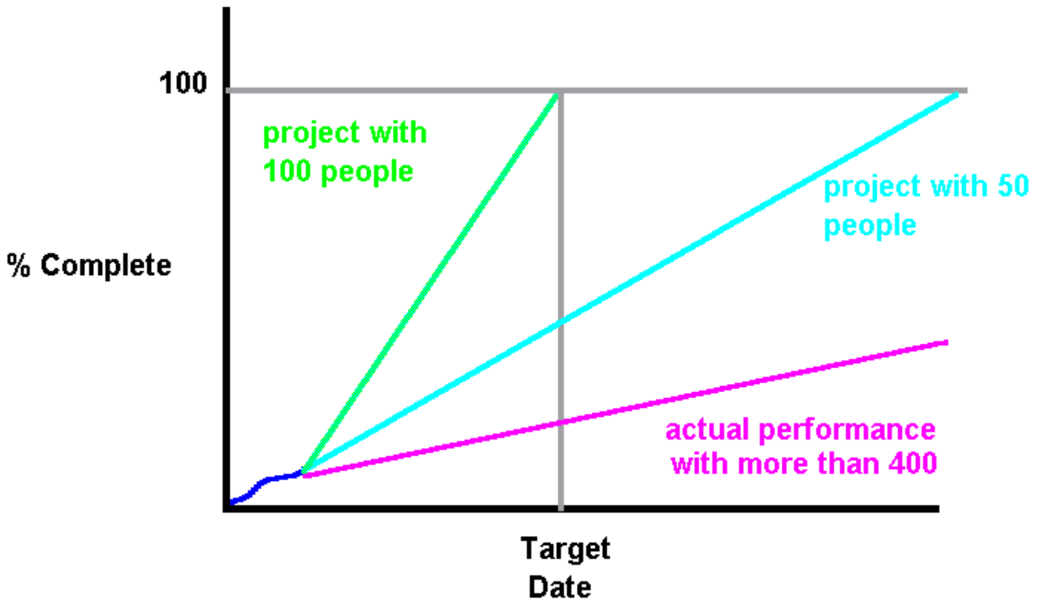
— Steve McConnell

Management Milieu

1) **In the "Mythical Man-Month" Frederick Brooks pointed out that managers frequently believe that *people and time can be freely interchanged!***

   That may be true in picking cotton or people on an assembly line, but in professional software task it is equivalent to suggesting that if one woman can produce a baby in nine months, two should be able to do it in four and a half months.

2)



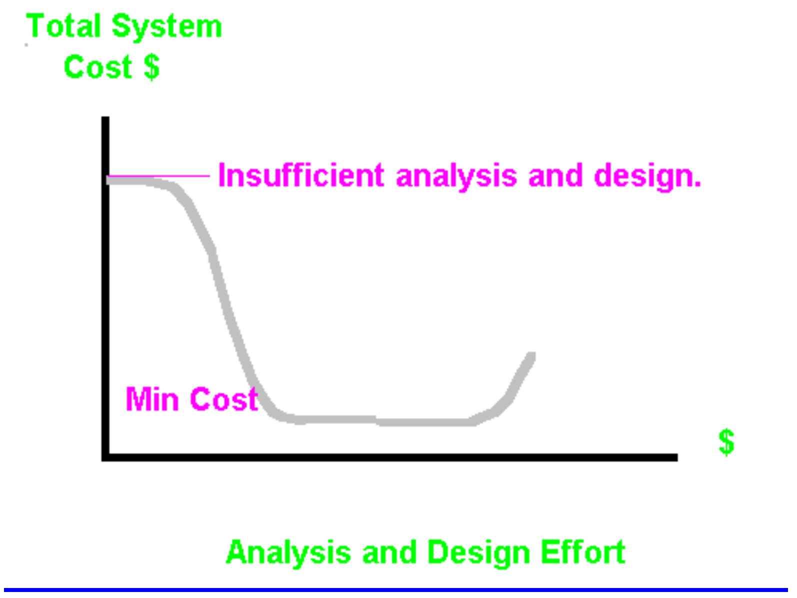
Mealy's Law

"There is an incremental person who, when added to a project consumes more energy (resources) than he or she makes available. Thus beyond a certain point, adding resources (people) slows progress in addition to increasing cost."

Brooks in Retrospect: **A manager adding excessive personal late in a project is guilty of a capital crime!**

3) **You simply cannot put multiple people on a project to speed it up when it is only a one person project and cannot be rushed. *Deadlines should reflect what is practical, not what is desired*. Must have complete design (blue print) to determine independent modules to estimate time to competition!**

4)



Under budgeting or premature termination of analysis or design leads to exceeding project budget and time estimates.

Don't be sold the bill of goods "*We cannot afford the luxury of a complete analysis and design, if implementation is not started immediately, we will never finish on time!*"

Features considered during analysis and design will never be so cheap again.

5) **The Thousand Module Effect:** **Turn a thousand implementers lose on a project without a complete prior design and there will be a thousand modules in the final system even if it is only 150 module system.**

Conway's Law:

"***Organizations which design systems are constrained to produce systems which are copies of the communications structures of these organizations.*** **If the system design is to be free to change, the organization must be prepared to change."**

Paraphrase:

The structure of any system designed by an organization is isomorphic to the structure of the organization.

Projects should be staffed around the design of the system. Systems should not be forced into the structure of an organization.

Frederick P. Brooks, Jr., "The Mythical Man-Month," Addison-Wesley, 1975, ISBN 0-201-00650-2.

M. E. Conway, "How do committees invent?" Datamation, 14, April 4, 1968, pp. 28-31.

Edward Yourdon and Larry L. Constantine, "Structured Design, Fundamentals of a Discipline of Computer Program and System Design," Prentice-Hall, 1979, ISBN 0-13-854471-9, pp.395-401.

How Should Large-Scale Systems Be Designed?

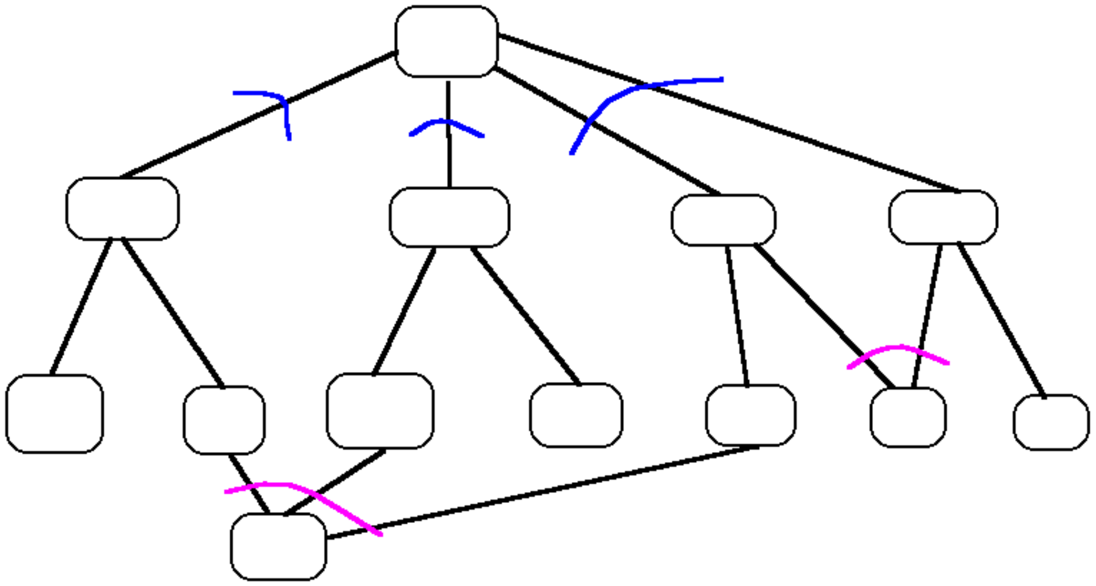
A single integrated design of the structure of an entire application area or larger system should be pursued as opposed to subdividing the design effort.

Why:

1) Subdivision of the design effort by management fiat without pursuing a complete integrated design establishes the interface between major subsystems with minimal knowledge. The best subdivisions are not always obvious. The frequent result is excessively coupled systems with poor cohesion.

2) Subdivision of the structural design ultimately determines the final system packaging into programs or suites of programs. Increased knowledge leads to more convenient, more efficient packaging. The ability to perform optimization for space and time frequently depends on packaging that reflects high levels of cohesion within the package and minimal coupling.

3) Possibly the most important reason for pursuing a complete design is that early subdivision tends to lead to duplicated effort. Different groups specify, design, code, and debug the same functionality. This separately implemented functionality must also be separately maintained during maintenance and system enhancement.



Design Oversight

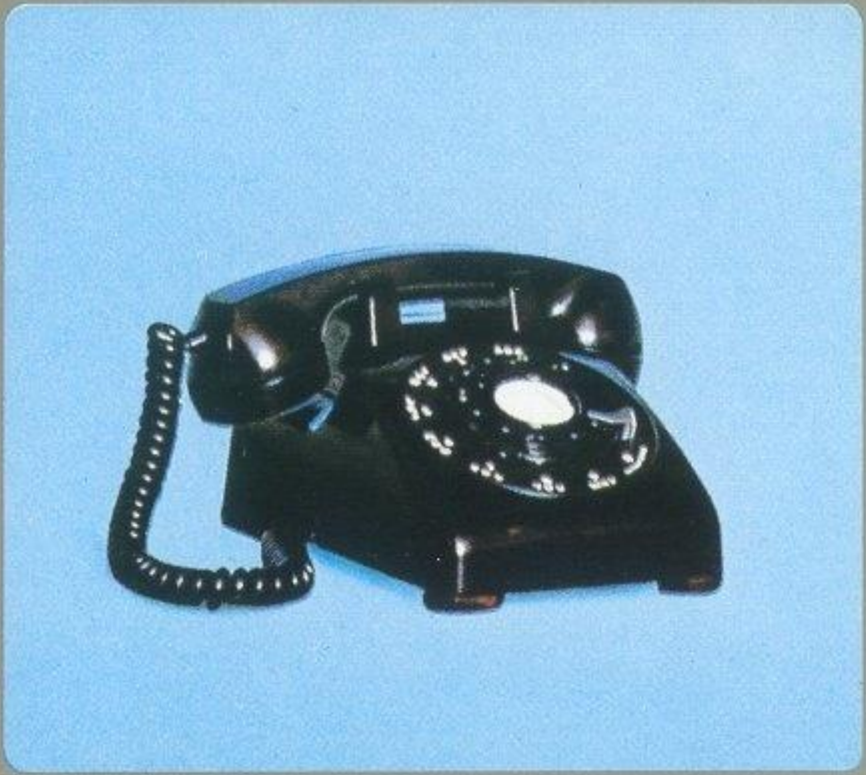
Interactive systems are frequently

- hastily proposed
- evaluated informally
- alternatives are never built or tested
- pilot projects are seldom built

Real Design Henry Dreyfuss (1955)

In the design of the 500-type telephone:

- Over 2000 human faces measured ear-to-mouth to determine spacing.
- Over 2500 design sketches made.
- Numerous variations of the hand grip tested prior to adopting the rounded-off triangular cross section.
- Variation of the face plate tested prior to selecting the 4-1/4 inch diameter.
- The angle of the dial was design to reduce glare.
- Placement of letters and numbers adjusted to reduce human fatigue.
- Clay and plaster models built to compare the leading designs.
- Extensive field testing of models: Field tests should involve actual users for sufficient time periods to get past initial learning problems and novelty. Competing designs should be compared in carefully controlled experimental conditions.



1949 **"500" TYPE DESK SET**

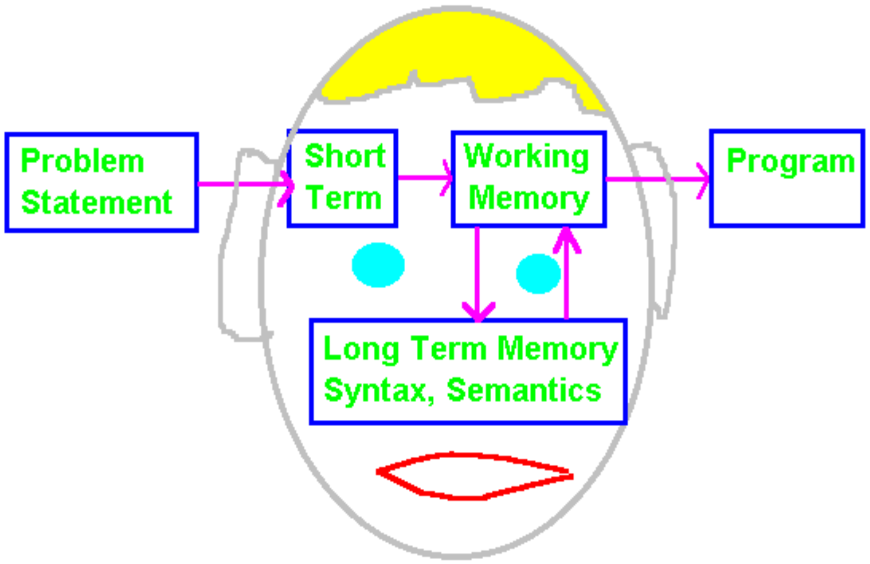
First in the new "500" series, which later would include a variety of colors. Rugged and functional, the "500" is the most commonly used telephone in the United States today. Standard with all the sets in this series is an adjustable volume control for the bell located in the base of the telephone.

DEF 3
ABC 2
1
GHI 4
JKL 5
MNO 6
PRS 7
TUV 8
WXY 9
0 OPERATOR
BELL SYSTEM
Western Electric 500 C/D
Rotary Dial Telephone
Produced December 1958

Design Oversight

No automatic system can be intelligently run by automatons or by people who dare not assert human intuition, human autonomy, human purpose. The judicious use of the computer depends quite literally upon the human employers ability to keep their own heads and reserve ultimate decision.

Paraphrase: Lewis Mumford “The Myth of the Machine” (circa 1970)



<u>Short term</u> memory holds sensory information for a few tenths of a second before passing it on and replacing it with new sensations. Short term or sensory memory acts as a filter for the rest of the brain.

<u>Working</u> memory processes the sensations and holds interpreted units of information for up to 30 seconds. This period can be extended by rehearsal or repetition. ***The size of working memory*** appears to obey George Millers limitation of the ***Magic Number Seven Plus or Minus Two***. The nature of the chunks are a function of experience and training. This is one reason professionals require 15 to 20 minutes preparation for technical task before becoming productive.

<u>Long-term</u> memory appears to be permanent, although some kinds of information may become more difficult to retrieve as time goes by. Its as if the catalogues to retrieve the information are discarded but the information is placed in storage closets.

Transferring chunks from *long-term* to *working* memory requires time and effort. People appear to need a break every 1-1/2 to 2 hours. Any technique that breaks a problem into chunks of this size facilitates problem completion and reduces the errors incurred. This is termed ***Closure*** - completion of a task leading to relief.

***** Anxiety or fear of failure greatly reduce the memory capacity of working memory and inhibit performance.

(George A. Miller, "The Magic Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information," Psychological Review, 63, (1956), 81-97.

User Engineering Principles:

First Principle: Know the user.

Minimize memorization, show sensitivity to human short-term memory limitations.

- Selection not entry.
- Names not numbers.
- Predictable behavior.
- Access to system information.

Optimize Operations.

- Rapid execution of common operations.
- Display inertia, as little of the screen should change as possible.
- Muscle memory, same operations done same way across all applications.
- Reorganize command parameters.

Engineer for errors.

- Good error messages, supportive not condemning.
- Engineer out common errors, the system should help prevent.
- Reversible actions.
- Redundancy.
- Data structure integrity.

W.J. Hansen, "User engineering principles for interactive systems," Proceedings of the Fall Joint Computer Conference, 39, AFIPS Press, Montvale, New Jersey, (1971), 523-532.

Provide a program action for every possible type of user input with feedback.

Minimize the need of the user to learn about the system, present a virtual system, restricting access to those who have acquired a certain knowledge level may be a good idea.

Provide a large number of explicit diagnostics, along with extensive on-line user assistance.

Provide program short-cuts for knowledgeable users; naive uses should be presented a simpler system, they will be more productive. Allow experienced users to express the same message in more that one way.

T. Wasserman, "The design of idiot-proof interactive systems," Proceedings of the National Computer Conference, 42, AFIPS Press Montvale, New Jersey, (1973).

Know the user population.

Respond clearly and consistently.

Carry forward a representation of the user's knowledge base.

Adapt wordiness to user needs, messages should be at the user's level of syntactic and semantic knowledge.

Provide the user with every opportunity to correct their errors.

Promote the personal worth of the individual user.

R.W. Pew and A.M. Rollins, "Dialog Specification Procedure," Bolt Beranek and Newman, Report No. 3129, Revised Edition, Cambridge, Massachusetts, 02138, (1975).

- Introduce through experience.
- Immediate feedback.
- Use the user's model. Emphasize that the user is in control of the terminal, the pace of interaction, the tutorial aids, and direction of progress. The user should always be able to establish what they have done and where they are in the process.
- Consistency and uniformity.
- Avoid acausality. The problem is the systems limitations not the users. Ask them to help the software understand, do not imply the problem is due to user's short comings. Do not dead end the user. Do not treat the user as a cheap data entry device.
- Query-in-depth (tutorial aids).
- Sequential - parallel tradeoff (allow choice of entry patterns).
- Observe ability and controllability.

Brian R. Gains and Peter V. Facey, "Some experience in interactive systems development and application," Proceedings of the IEEE, 63, 6, (June 1975), 891-911.

Forgiveness - ease in repairing errors.

Segmentation - layered approach.

Variety - choice of style.

Escape - break out of danger.

Guidance - direction and learning.

Leverage - flexible, powerful features.

M.W. Turoff, J. Whitescarver, and S.R. Hiltz, "The human machine interface in a computerized conferencing environment," Proceedings of the IEEE Conference on Interactive Systems, Man and Cybernetics, (1978), 145-157.

Many feel that the most detailed guide for the design of interactive display systems was developed by :

Stephen E. Engle and Richard E. Granda, "Guidelines for Man/Display Interfaces," IBM Poughkeepsie Laboratory Technical Report TR 00.2720, (December 19, 1975).

Summerized in:
"*Software Psychology Human Factors in Computer and Information Systems*" by Ben Shneiderman, Winthrop, 1980, pp224-232, pp247-266, ISBN 0-87626-816-5.

General:

The power of a computerized system must be better in every way than the manual system it replaces.

The quality of a new system is usually measured by users using the old system as a yardstick. It must have all the functions of the system being replaced even if they are not particularly important.

User satisfaction and effectiveness are not the same thing. A system can be very effective at what it does but unpleasant to use.

Error message must be supportive and constructive, never condemning and confrontive.

If critical design decisions are left to the technical personnel alone, those who oppose the new system will ensure its failure.

The first goal is to understand current practice, even if the new system will require different procedures. KNOW THE USER! Find out what they want and what they do not want. Find out what procedures need improvement from the standpoint of the user. Give them a chance to vent their displeasure with the design team. Listen carefully to user needs and desires.

Users resent being a pawn in a designer's technological chess game. Users want to improve their environment, but they want to do it themselves in their own way.

Acceptance of a new system is positively related to involvement in the implementation and negatively related to the perception of the system as threatening. (From a study by Roy H. Igersheim of 225 middle-level managers, Managerial response to an information system," Proceedings of the National Computer Conference, 45, AFIPS Press, Montvale, New Jersey, (1976), 877-882.

”Users fear losing their files more than anything else.”

When a command cannot be completed quickly, the use should receive progress reports periodically.

Users should always be able to determine their current status, what they have done recently, and their current options. Command stacks are helpful.

Nurture the user community:

- **create an active user community.**
- **on-site help.**
- **telephone consultants who really answer. If they do not immediately know the answer, the consultant should be expected to touch base with the user on a timely basis.**
- **an on-line suggestion box is helpful.**
- **for large communities promote:**
  - **newsletters**
  - **meetings**

Remember that you cannot satisfy everyone all the time. *View complaints as stemming from a desire to participate rather than an attack* (in spite of the user).

Be sensitive to the needs of minorities. Avoid creating dissatisfied groups. Avoid treating everyone in a vanilla fashion.