Software Engineering

(SoftEng1.doc)

By

David S Burris

Ph.D. in Computing Science

Certified Computer Programmer

Certified Systems Professional

Certified Computing Professional

csc_dsb@shsu.edu

The Texas Board of Professional Engineers adopted software engineering as a distinct discipline under which an engineering license may be issued (PE).

“The Software Engineering 2004 Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering” published by the Association of Computing Machinery (ACM) and IEE Computer Society (IEEE-CS).

<u>As of 2012 twenty-one U.S. Universities received accreditation in software engineering</u> from the Accreditation Board of Engineering and Technology (ABET).

The projected sales of software reached approximately $180 billion in 2000!

Software Engineering 2014

Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering

A Volume of the Computing Curricula Series

23 February 2015

Joint Task Force on Computing Curricula
IEEE Computer Society
Association for Computing Machinery

Software Engineering Code of Ethics and Professional Practice

(short version)

Software engineers shall commit them-selves to making the analysis, specification, design, development, testing, and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety, and welfare of the public, software engineers shall adhere to the following eight principles:

1. Public. Software engineers shall act consistently with the public interest.

2. Client and employer. Software engineers shall act in a manner that is in the best interests of their client and employer, consistent with the public interest.

3. Product. Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.

4. Judgment. Software engineers shall maintain integrity and independence in their professional judgment.

5. Management. Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.

6. Profession. Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.

7. Colleagues. Software engineers shall be fair to and supportive of their colleagues.

8. Self. Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Alan Davis' 1994 Principles of Software Engineering (expanded to over 200)

Because the product of software engineering is not physical, physical laws do not form a suitable foundation. Instead, software engineering has had to evolve its principles based solely on observations of thousands of projects. The following are probably the 15 most important principles:

1) **Make quality number one priority;**
2) **High-quality software is possible but it may have a high price;**
3) **Give products to customers early;**
4) **Determine the problem before writing requirements;**
5) **Evaluate design alternatives;**
6) **Use an appropriate process model;**
7) **Use different languages for different phases (no optimal language);**
8) **Minimize intellectual distance (solution space matches real-world);**
9) **Put technique before tools (do not do the wrong thing faster);**
10) **Get it right before you make it faster (put off optimization);**
11) **Inspect code (inspection specification, design, code reduce errors);**
12) **Good management is more important than good technology (good managers can do extraordinary things with limited resources);**
13) **People are the key to success (software is labor intensive, need talented people with appropriate drive);**
14) **Follow hype with care (popular with others is not a good indicator of success for your project); and**
15) **Take responsibility (blaming the problem on others, the schedule or the process is irresponsible).**

Additional attributes should include maintainability, installability, usability, reusability, interoperability, and modifiability!

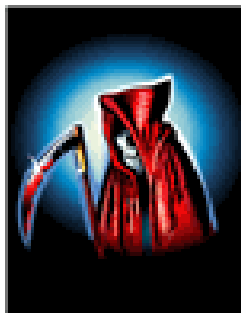
PROJECT IN TROUBLE?

Our development team will beat the deadline. Even now they are working after hours, applying every known technique to avoid the consequences of not meeting the deadline!

Pull a rabbit

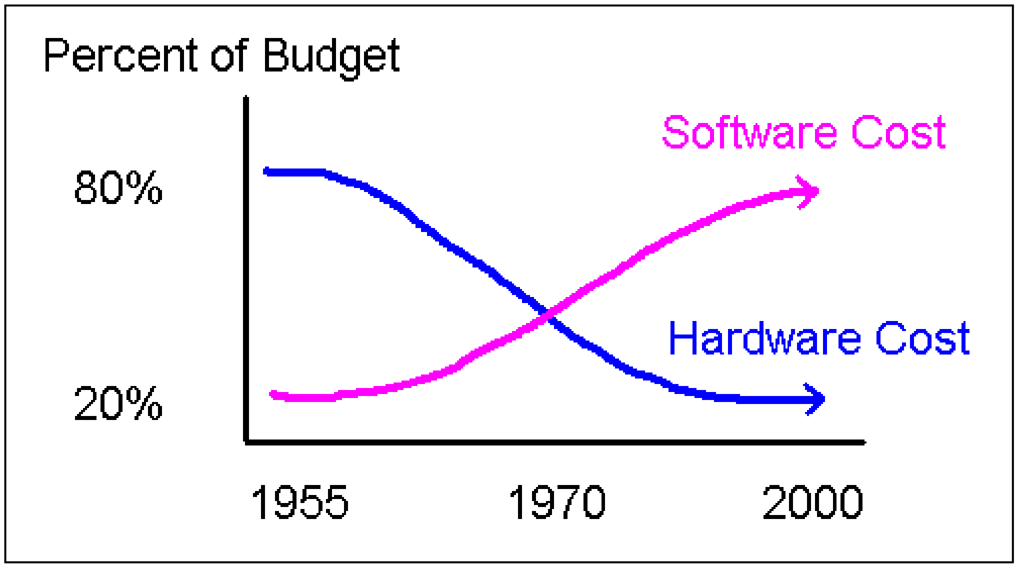
Bubble bubble, Toil and Trouble! **Alakazam!**

Majic, Vodoo, Witch Doctor



Is the Grim Reaper waiting for YOU?

Historical Spending Trends

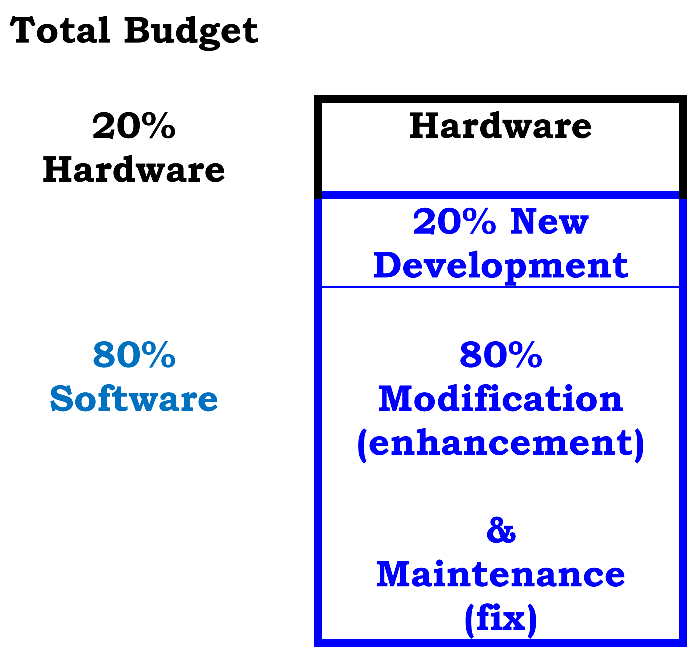


Spending Trends: The cost of hardware has dropped from 80% of budget to 20% of budget. Software cost has gone from 20% to 80% and are rising. This is a desirable trend.

15% of all projects fail, 25% of larger projects - Demarco and Lister Lister 1987: "Peopleware Productive Projects and Teams."

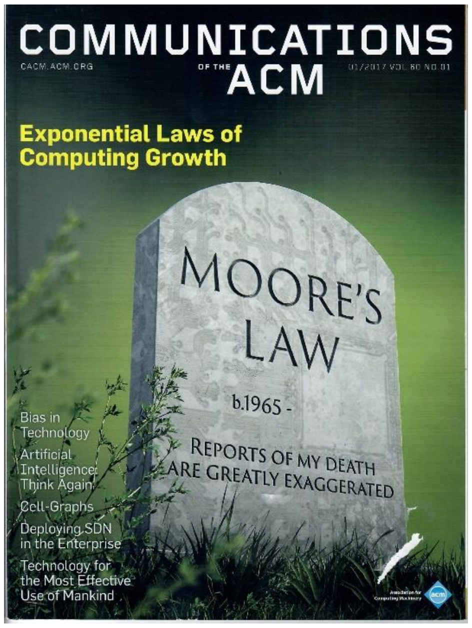
"The Mythical Man-Month" by Frederick P. Brooks, Jr. – father of the IBM System 360. The real problems are people and complexity!

Historical Spending Trends



The percentage spent in new program development as opposed to maintenance and modification (enhancement) is frequently out of proportion.

Excessively high enhancement cost tends to stifle new development. By the mid 80's, some larger companies were spending more than 95% of their software budget on M&M!


COMMUNICATIONS OF THE ACM
CACM.ACM.ORG
01/2017 VOL.60 NO.01
Exponential Laws of Computing Growth
MOORE'S LAW
b.1965 -
REPORTS OF MY DEATH ARE GREATLY EXAGGERATED
Bias in Technology
Artificial Intelligence: Think Again
Cell-Graphs
Deploying SDN in the Enterprise
Technology for the Most Effective Use of Mankind

In 1965 Gordon Earle Moore (co-founder of Intel) extraplolated from lab data collected from 1959 to 1965 the number of transistors, resistors and capacitors on chips would double every year year from 2^6 in 1965 to 2^{16} in 1975. Faster clock speeds are not currently practical to speed up computations due to heat and coordination problems (metastability). Dennard scaling showed how to reduce component dimension without increasing power density. Dennard scaling reached an inflection point in the 1990s due to heat dissipation limiting clock speed to approximately 3.5Ghz. Koomey's Law documents the exponential growth of computations per computer and and per unit of energy from 1946 to 2009 placing constraints on an energy-constrained industry.

The exponential growth of computations however continues today using multi-tasking, multi-cores, and programmers learning to take advantage of the architectures.

Continued exponential growth depends on three conditions to sustain growth: 1) growth at the chip level, 2) demand at the system level, and 3) demand by the adopting community.

"Exponential Laws of Computing Growth," CACM Jan. 2017, Vol. 60, No. 1, p 54-65: Peter J. Denning and Ted G Lewis.

ERRORS INCURRED IN

DESIGN VERSUS IMPLEMENTATION

Considerable time and money are required during coding and testing to correct errors created during the requirements definition and design phase of a traditional software project.

Design errors outnumber other errors by a ratio of three to two and are usually more persistent.

Thayer, "Understanding Software Through Analysis of Empirical Data," Proceedings of 1975 National Computer Conference.

Boehm, etal., "Some Experience with Automated Aids to the Design of Large-Scale Reliable Software," IEEE Transactions on Software Engineering, March, 1975.

Frederick Brooks: "The Mythical Man Month" – 1000 known, uncorrectable errors in every release of OS360. Fixing them and the resulting errors lead back to the original error (1975 and 1995).

Data General: Unknown error 162. Knew error but inconsistent design made them impossible to fix. For example, set system date before or after leap-year-date to avoid losing file system!

If builders built buildings the way programmers wrote programs, the first woodpecker that came along would destroy civilization!

(Gerald Weinberg - 1971)

Later recanted – **specification and design are more problematic.**

Problems with Traditional Systems Analyst / Designer /Programmer / Coder Implemented Software Systems

1) **Projects are frequently late and exceed resource estimates.**

2) **The customer realizes too late it's not what they really wanted, takes responsibility and abandons the project.**

3) **It does what the customer wants but not in a manner that fits the current mode of operation. Again, the customer either takes responsibility or blames the developer!**

4) **Fifty percent or more of development time is spent in debugging.**

5) **Delivered programs contain an unacceptable number of errors.**

6) **The software cannot be reasonably maintained. Small changes in environment or businesss practice (hardware, software support environment [e.g., OS or file system], business rules) result in major modifications with respect to effort or cost.**

7) **Optimization for space or time is intractable.**

Partial Course Goals

1) What it means to be a ***Software Engineered Systems***.

2) ***Minimizing*** the cost of software versus minimizing time to completition:

   cost = development + maintenance +
   modification + other terms.

   development = analysis + architectural design +
   detail design + code + debug + installation.

3) The structure of a minimum cost system, how do you manage the "Life Cycle."

4) ***Measures of human complexity and design metrics:*** coupling, cohesion, fan-in, fan-out, afferent, efferent, factored, scope of effect, scope of control, Miller's Law, Graicunas' Law.

5) Formal Design Techniques: Control Flow, Top-Down-Iterative Refinement, Static Data Flow Analysis (e.g., Jackson's Method), Dynamic Data Flow Analysis (Yourdon and Constantine), Object Oriented Methods (OOA, OOD, OOP).

6) Systems Communications (the products of one system are the inputs of multiple other systems).

7) Implementation Strategies, e.g., top-down (classification) versus bottom up (composition).

8) Resource Estimating.

9) Project control and management: including Traditional Team, Weinberg (egoless), Group Review (Walk Througs), Chief Programmer, Peer Review, RAD, JAD, Tiger, Agile.

Misconception:

Software Engineering is not an exact discipline producing precise closed form (exact) solutions to problems.

All engineering depends as much on common practice and empirical knowledge as scientific fact.

Software Engineering is aimed at producing alternative solutions to problems that do not have a well defined solution and making successive decisions until the "best" solution is arrived at.

Problems with specific closed form solutions exist most often in universities.

Industry must frequently face problems that are open ended. We really do not know the final result or best way to arrive at it early in the project.

Typical Questions

In The

Arsenal Of The Analyst

1) What is the purpose of doing the work?
2) Who utilizes it?
3) Why is it necessary?
4) Do many people use it?
5) Could it be eliminated or combined with another operation?
6) Is ADP the only answer?
7) Is the work being done by the right people?
8) Is it being done in the right place?
9) Is the capacity of the system adequate now?
10) Are there any bottlenecks?
11) Is overtime required?
12) Will a computer expand capacity?
13) Are the reports now available, relevant to the chosen objectives?
14) Do top executives really need and want speedier reporting of data?
15) Is greater accuracy really needed?
16) Do managers need additional reports?
17) Could these reports be gathered without using a computer?
18) Does customer service need to be improved?
19) Is ADP required to make the improvement?
20) How much processing must be done to satisfy the real needs?
21) What will it cost to satisfy these needs by revisiting current methods?
22) Have any expected systems changes been made or contemplated with the view of later system integration?
23) Is a total systems philosophy desirable for the activity?

Typical Questions In The Arsenal Of The Analyst

What is the purpose of doing the work?

Who utilizes it?

Why is it necessary?

Do many people use it?

Could it be eliminated or combined with another operation?

Is ADP the only answer?

Is the work being done by the right people?

Is it being done in the right place?

Is the capacity of the system adequate now?

Are there any bottlenecks?

Is overtime required?

Will a computer expand capacity?

Are the reports now available, relevant to the chosen objectives?

Do top executives really need and want speedier reporting of data?

Is greater accuracy really needed?

Do managers need additional reports?

Could these reports be gathered without using a computer?

Does customer service need to be improved?

Is ADP required to make the improvement?

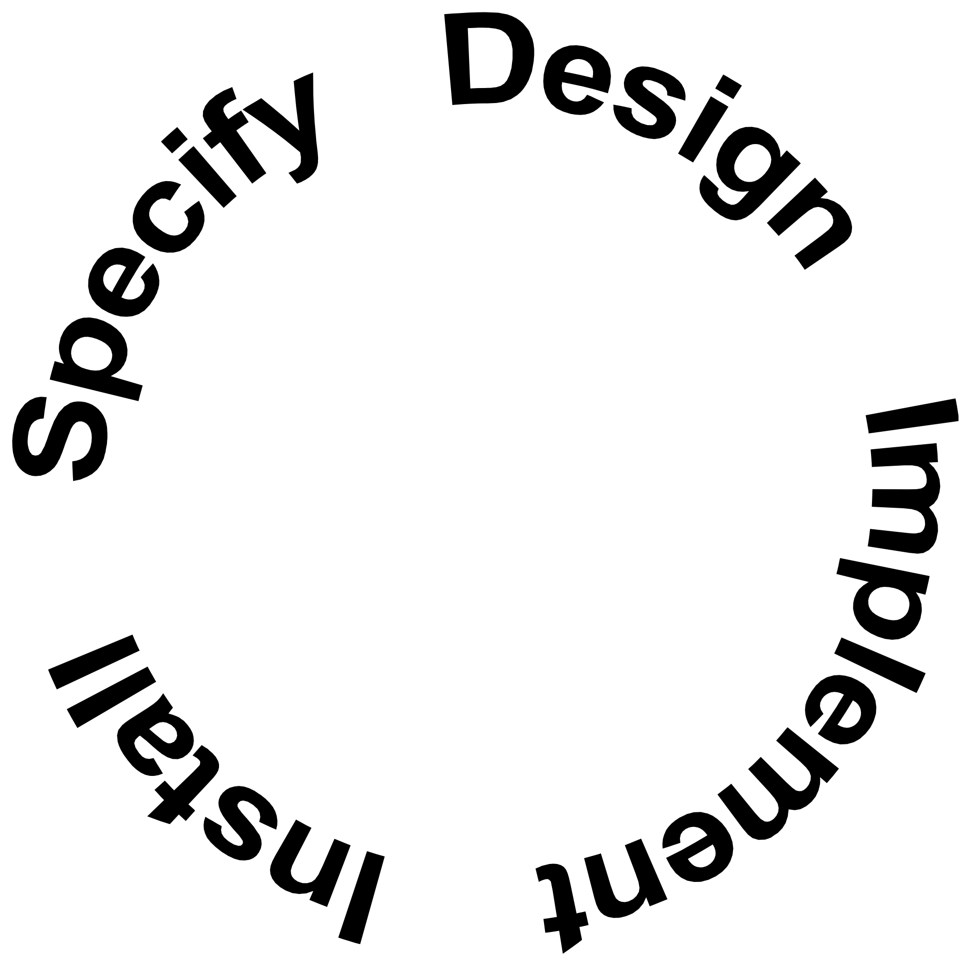
How much processing must be done to satisfy the real needs?

What will it cost to satisfy these needs by revisiting current methods?

Have any expected systems changes been made or contemplated with the view of later system integration?

Is a total systems philosophy desirable for the activity?

Software Life



Cycle

Software Life Cycle Models

Freeman	Metzger	Boehm
Needs Analysis	System Definition	System Requirements
Specification		Software Requirements
Architectural Design		Preliminary Design
Detail Design	Design	Detail Design
	Programming	Code &
Implementation	System Test	Debug
	Acceptance	Test &
	Installation &	Pre-Operation
Maintenance	Operation	Operation & Maintenance

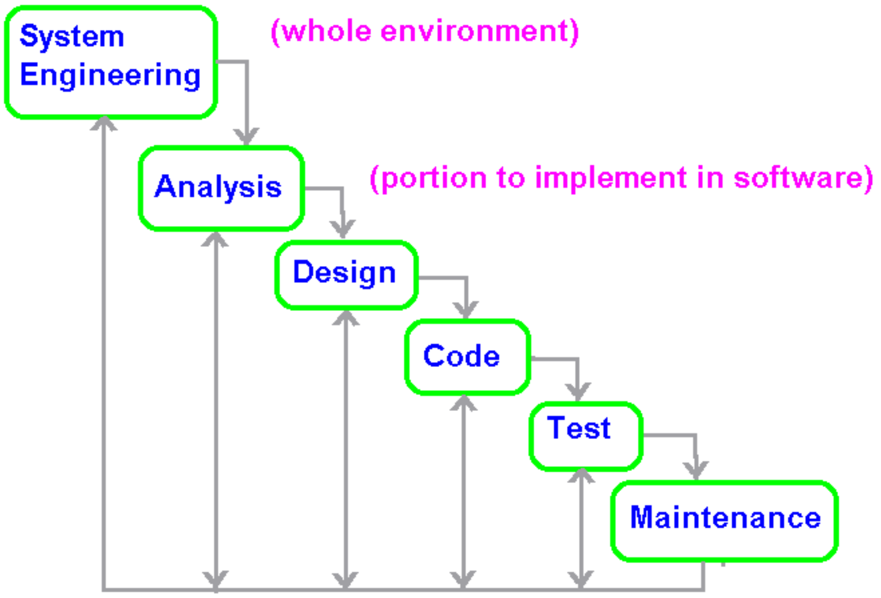
P. Freeman, "Tutorial on Software Design Techniques," New York: IEEE Computer Society, 1976.

P.W. Metzger, "Managing a Programming Project," Englewood Cliffs, N.J.: Prentice-Hall, 1973.

B.W. Boehm, "Software Engineering," IEEE Transactions on Computers, Vol C-25, No. 12, Dec 1976, pp1226-1241.

Classics in Software Engineering, ed. E.N. Yourdon, (New York: Yourdon Press, 1979) pp 325-361.

Classic Waterfall Life Cycle



The paradigm demands a systematic, sequential approach to development. Management measures progress by the developers current stage in the water fall. Hence, management does not encourage backtracking at any stage.

Problems:

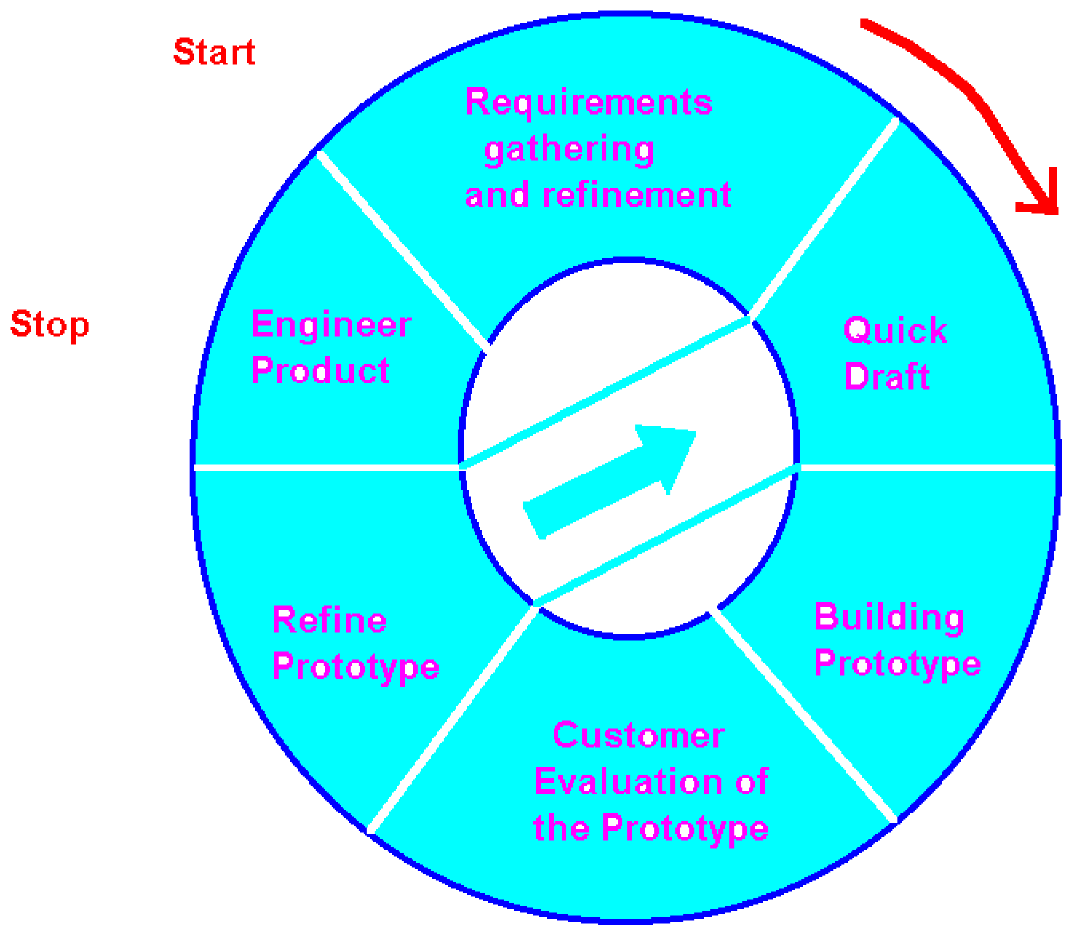
1) **"Large" real projects rarely follow the sequential flow proposed by the model. Iteration always occurs and creates problems in model application.**

2) **It is frequently difficult or impossible for the customer to state all requirements explicitly. The waterfall model has difficulty accommodating this natural uncertainty.**

3) **The customer must be patient. A working version of the software will not be available until late in the project. A major blunder not detected until the software is reviewed means disaster.**

If the customer is sure of steps 1 and 2, the waterfall model is frequently desireable.

Roger S. Pressman, "Software Engineering A Practitioner's Approach," 3rd ed., pp. 25-28, ISBN 0-07-050814-3.


Start
Stop
Requirements gathering and refinement
Quick Draft
Building Prototype
Customer Evaluation of the Prototype
Refine Prototype
Engineer Product

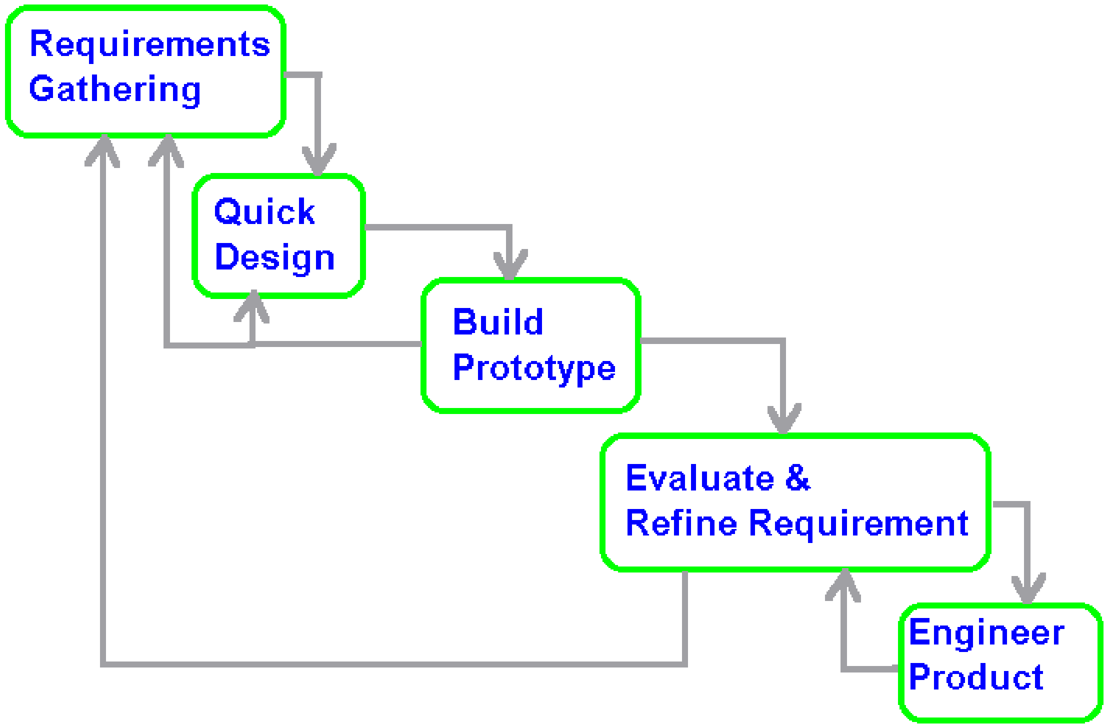
Prototyping Paradigm

Roger S. Pressman, "Software Engineering A Practitioner's," 3rd Edition, McGraw-Hill, 1982, pp 22 - 36, ISBN 0-07-050814-3.

Agile Development, classic life cycle models and approaches suggested by the Software Engineering Institute frequently represent different ends of the development spectrum!

Traditional projects insist all requirements be determined prior to implementation. All functionality must be in the final product. Agile tends to encourage developing functional requirements during development. If time or budget runs out, functionality may be sacrificed.

Prototyping Life Cycle



<u>Prototyping appears to best if:</u>

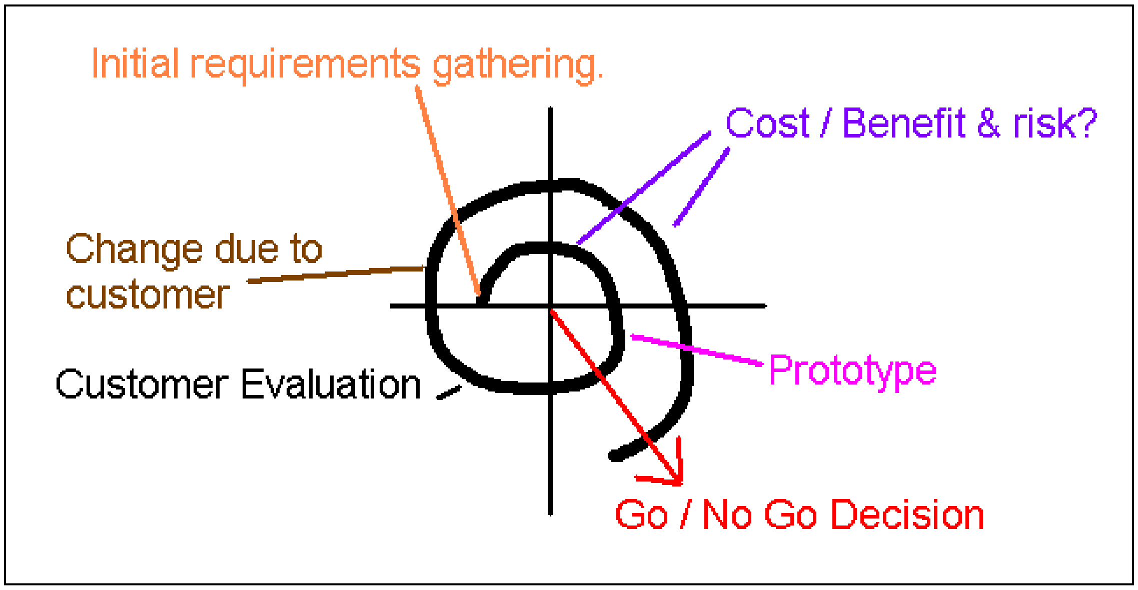
1) The customer has a set of general objectives for software, but cannot identify all the detailed inputs and / or outputs, or processing requirements.
2) May not be sure of the efficiency of an algorithm or the best approach to solving the problem.
3) Not sure of the operating environment or its adaptability.
4) Unsure of the form that the man-machine interaction should take.

Prototypes may be:

- a fully or partially functional model,
- a paper model (picture show) displaying user screens and interactions.

Can the prototype be expanded to complete the final system or should it be limited to requirements definition?

Spiral Model



The project alternately gathers information followed by an evaluation of the risks and benefits. If the benefits exceed the risk, a prototype is built. The prototype is evaluated by the customer'.

The customer's evaluation is used for another information gather stage followed by risk analysis, building a new prototype, and another customer evaluation.

This process is repeated until a final product is determined or the cost becomes prohibitive and the project is abandoned.

Three cycles are sufficient to evaluate many projects.

This open-ended approach to evaluation and implementation is hard to sell in many environments even though it may be the best and most cost effective method for the current problem.

General Approach

If you ask the average developer if they use an organized approach to the specification and development of software they are almost sure to answer in the affirmative.

What they really do might look more like:

A) **Manager Says:** **Quick, quick, we have just been given a new project that must be finished next month!**

B) **Mary, you run upstairs and find out the details of what they really want the system to accomplish.**

C) **Meanwhile, the rest of you start coding or we will never finish on time!**

Traditional software development is an art characterized by:

A) **Large doses of folklore,**

B) **black magic, and**

C) **Occasional flashes of intuition.**

Classic Roles

User/Client: **Recognizes a need.**

Systems Analysist: **Helps the user to specify the need, recognize the actual problem as opposed to symptoms caused by the problem (silo effect). Usually required to specify tentative algorithms, data base, hardware, and approach to solution. Evaluate initial risk and cost / benefit analysis. The systems analysist must know what the user does in great detail and how what the user does integrates with the rest of the company. Systems analysist tend to be more user oriented than technically oriented. Only tentative decisions should made during specification as minimal detail is actually known. Ideally a specification should be a statement of what is needed, not how it should be accomplished!**

Systems Designer: **Frequently knows a lot about the user domain but is more technically oriented. Responsible for the formal overall system architecture. The systems designer determines packaging, makes all major software, hardware, and design decisions. The systems designer determines the approach used to solve the problem. They take the amorphous blob which constitutes the specification, break it into portions and indicate how the portions will be communicate with respect to control data.**

***Design is 99% perspiration and 1% inspiration*. The purpose of a design method is to keep the perspiration from drowning the inspiration! (Mac Alford paraphrase of Thomas Edison on Genius)**

Programmer: **Determines algorithms at the detail level. Is a sequential search adequate or should a binary search or hashing algorithm be utilized?**

Coder: **Knows how to apply a programming language to implement code once the solution approach and algorithm has been determined.**

Objectives of Design

1) Optimize the use of the computing resource.

		Proj1		Proj2
a)	Space.	3 meg.		2 gig
b)	Time.	Exec	1.8 hrs.	2.2 hrs.
c)	Human?	Modify	4.7 days	4 hrs. *best!

2) Optimize Reliability.
   - a) mean-time-between-failure, MTBF.
   - b) mean-time-to-repair, MTTR.

   Quality measure for large number of users:
   MTBF = {[N+(N-1)+(N-2)+...+(N-I)]*C} / M

   Where
   - N = number of copies in use,
   - M = number of bugs encountered in a given year,
   - C = estimate of number of hours the product is used per year,
   - I = number of years the product has been used.

   "Measuring Software Reliability," by Hansen, *Mini-Micro Systems*, August 1977, pp54-57.

   - c) System availability.
     SYS-AVAIL = MTBF / (MTBF + MTTR).

3) Modifiability – ability to adapt to changing environment

4) Flexibility – for inventory in grocery store to add new product lines (related applications)

5) Generality – range => grocery store, drug store, school (not directly related)

COST: Less expensive to design and implement general purpose software than tailored software for each application. It may not however be as efficient of the hardware and the initial implementation generally requires more effort and higher cost.

The Ultimate Goal For Most Software Projects Is Economic:

1) Professional: The lowest possible net discounted value of all future resource consumption.
2) Professional: The maximum net present value of future profits.
3) Backwoods: Most "Bang per Buck!"

The Ultimate Goal For Design Is To Produce Minimum Cost Systems

Minimum cost to:

develop,
operate,
maintain, and
modify.

Least cost to develop may be the current goal but is seldom the best long term goal!

A Specification Consists of

a set of functional requirements (otherwise know by such titles as "systems specification" or "functional specification") that describe in precise terms the desired *inputs* when possible, the *outputs* required by the user, *algorithms* essential for the computations, and any *constraints* such as the number of transactions that must be processed per minute or raw data/results retention requirements.

A specification should not specify job steps, memory size, the design tool or programming language to be used, the number or kind of intermediate files, database product, record layouts, or other hardware / software required by the implementation team.

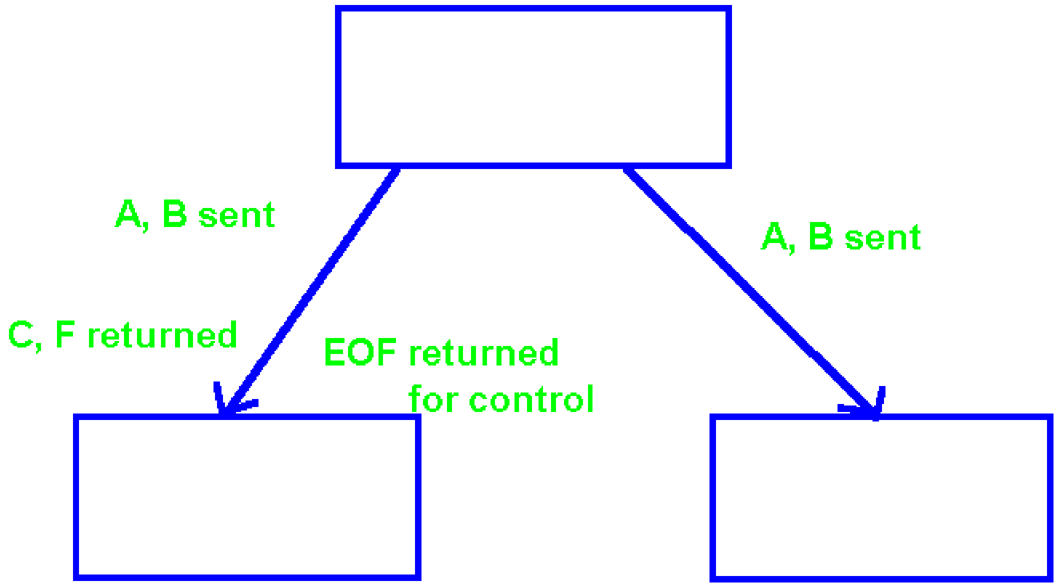
We do not wish the systems analysts to design. **This is somewhat unrealistic of course since the systems analysts are required to discuss "feasibility" and "cost-effectiveness" in terms of the size, and power of the computing / software resources require. Remember that any commitments made at this stage must be done with minimal knowledge of the final system. Such commitments frequently severely hamper the design team if they are constrained to live with the less than optimal decision.**

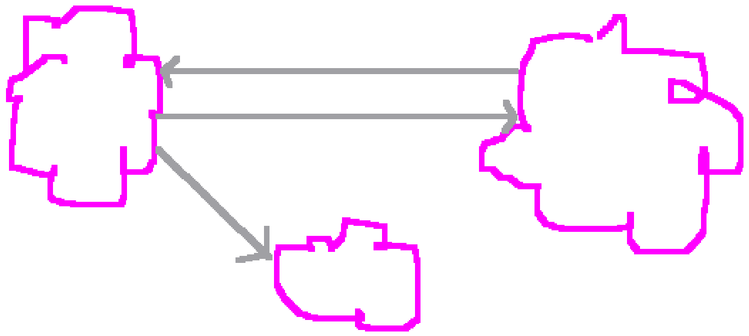
What Constitutes a Design

A *design* must take the original problem and *break in up into its constituent parts.* In addition it must *show the visibility of each part with respect to the other parts* of the design both *with respect to control flow* and *data flow*.

Ideally a plan for *staffing* of personnel required, *software products* to be utilized, *hardware resources* required, and a *time-line for completion* should be included.

In other words a plan for implementing the software must be established and the management *"process"* identified.





DESIGN

A ***specification*** is a set of functional requirements for a system or program. It should normally state what is required or desired, not how the solution is to be accomplished.

Design is an activity that begins after the systems analysts has produced a set of functional requirements for a program or system and ends when the designer has specified the components of the system and their interrelationship (including data and control).

It is insufficient, in most cases, for the designer to consider a solution a design. Rather several solutions should be generated and evaluated on a technical basis to choose the "best" design. "Best" in the sense of maximizing such technical objectives as efficiency, reliability, time to completinion, and maintainability while satisfying such design constraints as memory size and response time.

Achieving Minimum Cost Systems

Successful design is based on the principle of divide and conquer. The cost polynomial = specification + design + coding + maintenance + modification + …!

I) The cost of implementing a computer system is minimized when the parts of the design meet the following criteria:

1) **the parts are separately solvable, ideally with one idea per module, and**

2) **the parts of the solution correspond to identifiable parts of the problem. The majority of time fixing errors (75+%) is normally locating them!**

II) The cost of maintenance is minimized when the parts of the design are:

1) **easily mapped from the logical system as seen by the user to the implementation, and**

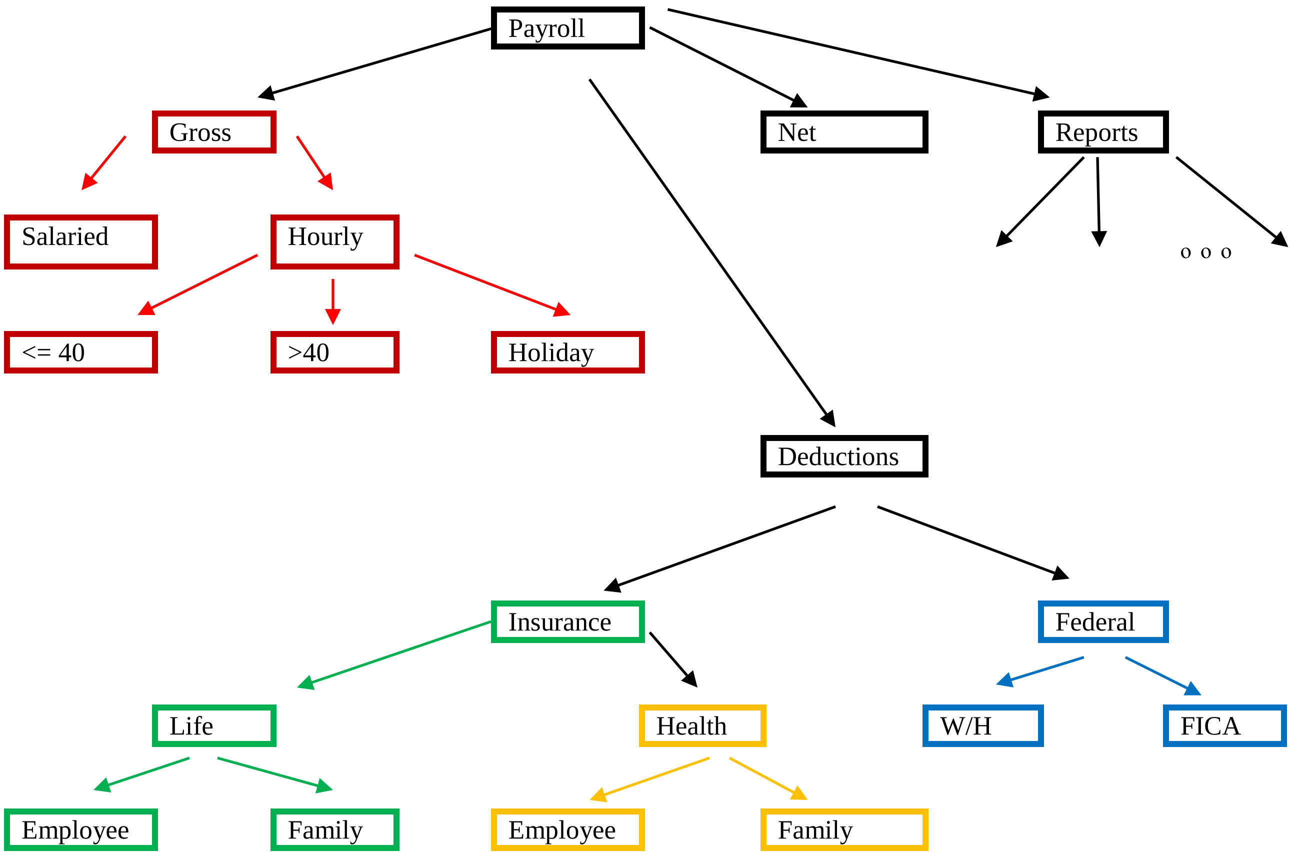
2) **may be corrected separately.**

III) The cost of modification is minimized when the parts of the design are:

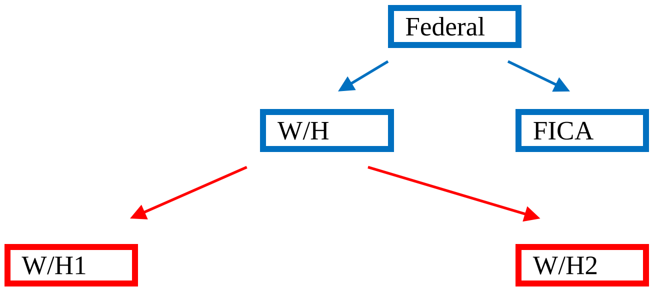
1) **easily mapped from the logical system as seen by the user to the implementation, and**

2) **may be corrected separately.**

Design modules containing single ideas. The modules are separately implementable after specifying inputs, outputs, and possible a specific algorithm. The design is highly factored and meets other design criteria such as Miller's Law while maximizing cohesion and minimizing coupling.



Poor design. The single idea in module W/H is split into two modules to allow two personnel to work on the implementation simultaneously. Unfortunately since each module contains part of the idea in another module. They are not separately implementable. The implementers must cooperate and be in constant communications.



Design Summary

The effort to implement, maintain, and modify a system is generally minimized when:

1) each piece of the design corresponds to exactly one small piece of the problem, and

2) the relationships between a system's pieces corresponds to relationships between the pieces of the original problem.

Good design is an exercise in partitioning and organizing the parts of a problem into smaller sub problems such that each sub problem eventually corresponds to a piece of the original system. A good design preserves the relationships that exist between the parts of the problem in the computer solution. It should not introduce new relationships.

Important Design Questions:

1) **How should we subdivide the problem?**

   Designs should organize the parts of a system in a manner that preserves the natural interconnections of a system while not introducing interrelationships that do not exist in the original problem.

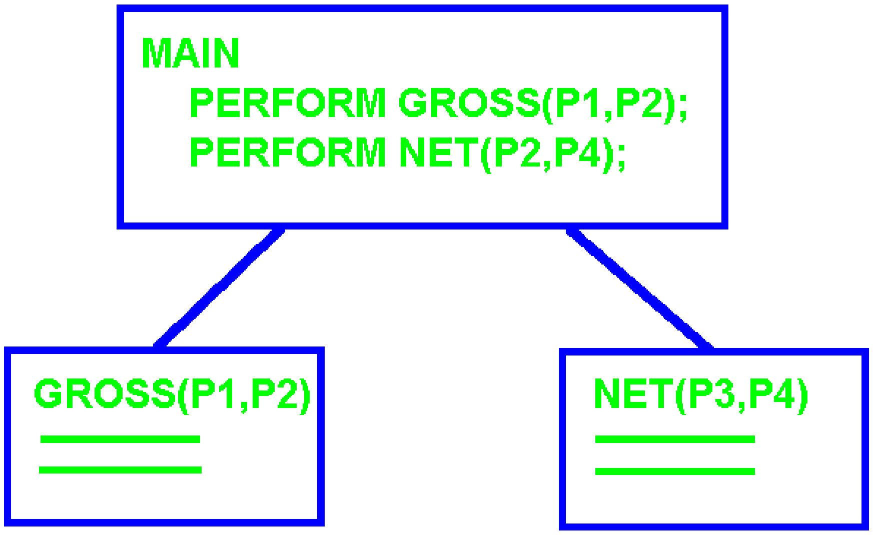
2) **Which aspects of a problem belong in the same part of the system?**

   Highly interrelated part of the problem should be in the same module of the system, i.e., thing that belong together should go together.

3) **Which aspects belong in different parts of the system?**

   Unrelated parts of the problem should reside in unrelated modules of the system. That is, thing which are not related to each other don't belong together.

Black Box Concept



RULE OF BLACK BOXES:

Whenever a function or capability is required during the design of a system, define is as a black box and make use of it without concern for its structural or methodological realization.

Black Box Characteristics:

1) **A true black box can be fully exploited (utilized) without knowledge of how it is constructed.**

2) **It exhibits *high integrity*; it does exactly the same function for every module that invokes it. It does not use special preambles or post ambles to manage differences!**

No "Gray" boxes where the module does a slightly different job (similar but different ideas) for different purposes as indicated by the input parameters!

DIFFICULTY VERSUS COST

Assume a function "D(X)" that measures the difficulty of a problem X and a function "C(X)" that measures the cost of its solution. Given two independent problems P and Q:

1) IF $D(P) > D(Q)$ THEN $C(P) > C(Q)$.

* Cost is a monotonically increasing function of problem size

In general a monolithic solution is more difficult than solving two independent problems separately and combining the results.

2) $D(P+Q) > D(P) + D(Q)$

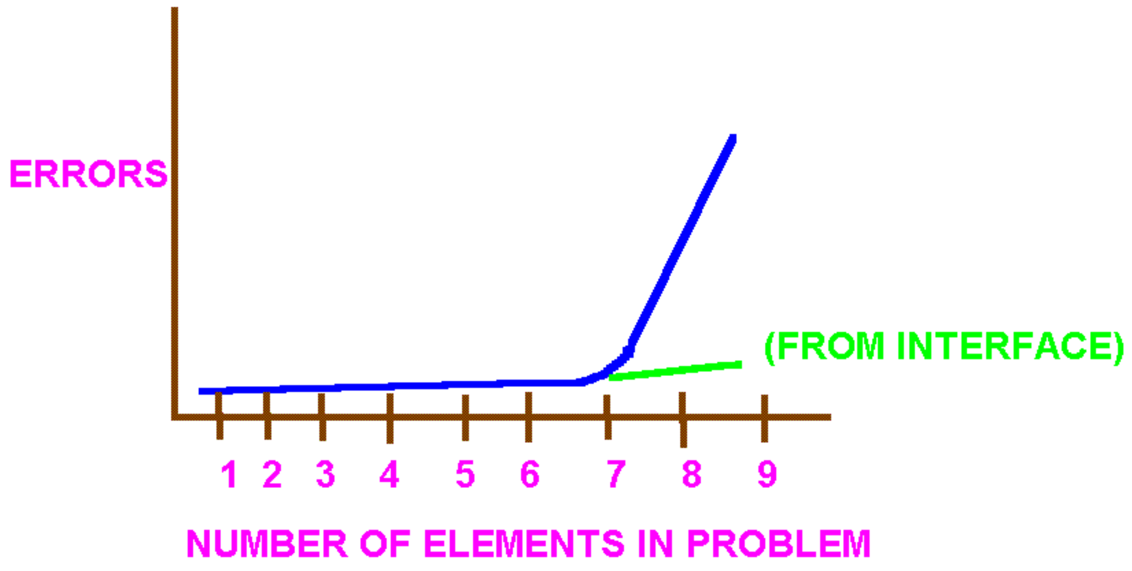
hence

3) $C(P+Q) > C(P) + C(Q)$,

where P and Q are relatively independent.

SPAN OF ABSOLUTE JUDGMENT OR UPPER LIMITS ON PERCEPTION

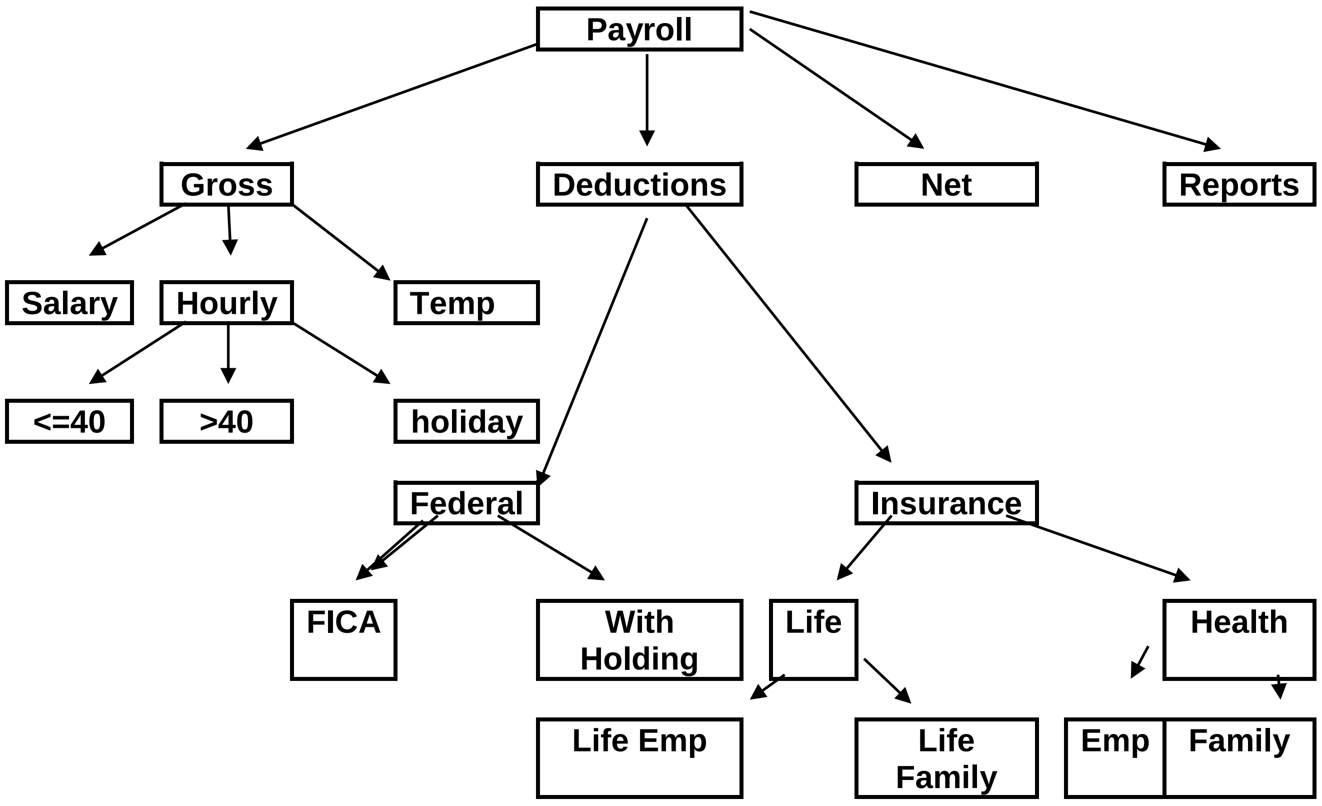
George Miller: Psychologist / Mathematician (1956)



It would appear that people can mentally juggle, deal with, or keep track of about 7 objects, entities, or concepts at a time if they are related.

MENTAL CAPACITY OF 7 +/- 2 !

Partial Design Payroll System



Partial design exhibiting control flow. Data flow must be added.

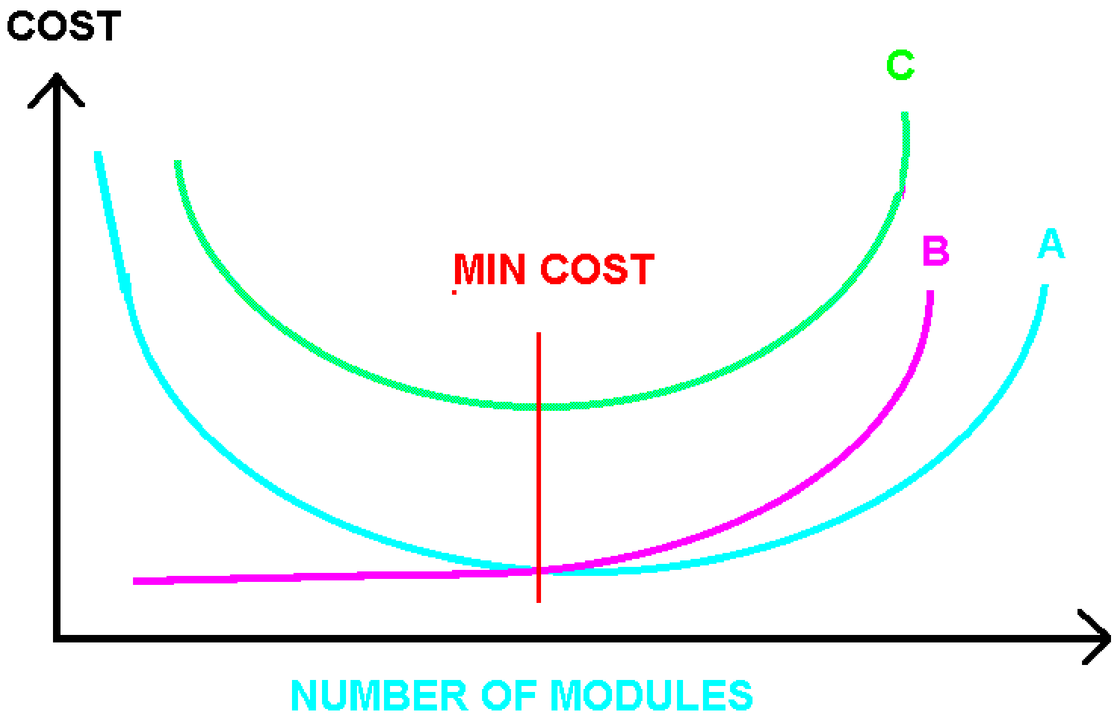
Example of data flow at user interface:

Hourly:
IN: Hours, pay rate
Out: Gross pay

Modularizations allows:

1) **Reduction in complexity**
2) **Results in lower implementation cost**
3) **Lower modification cost**
4) **Lower maintenance cost (fixing bugs)**
5) **Multiple people to work on the project with minimum conflict**

MINIMUM COST

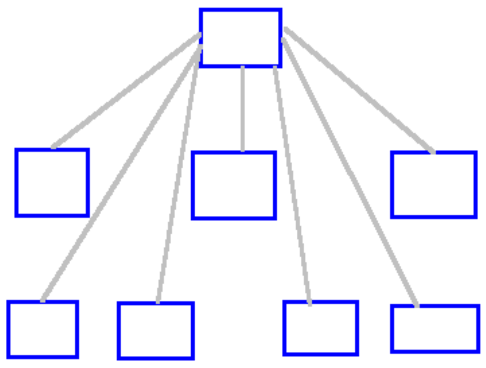


A: The cost of a system is generally reduced by design techniques that modularize a system into parts where each part consists of single complete ideas. This reduces the number of intra-module errors.

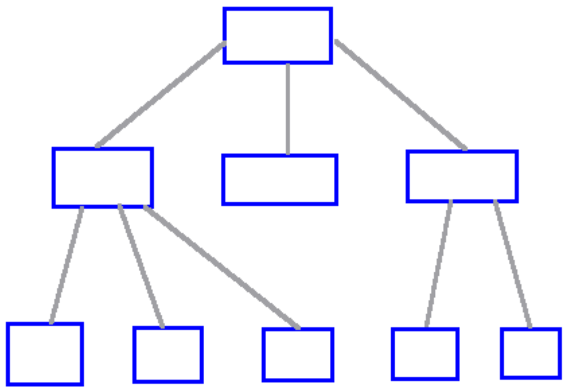
B: Increasing the number of modules increases the number of inter-module errors at the interfaces.

C: Total Cost = A + B.

Minimum occurs when there is one idea per module!



The main control module is subject to excessive detail. Decisions will be error prone and the director is a candidate for a coronary.



The detail that must be absorbed by the main module has been reduced to an acceptable level through "delegation" of authority. Decisions will be more effective and based on a more accurate knowledge base.

One of the first things a consulting firm looks at when attempting to understand the *information bottlenecks* and *control problems* in a company is its management structure chart. The management structure must provide for *sufficient and accurate information to stimulate the decision process* while *avoiding excessive and useless details* that would stifle the decision process.

In the structure on the left the CEO has everyone reporting directly to them. Both the number reporting and the details that must be absorbed, all at the same time, are subject to *Miller's Law*.

In the structure on the right, these effects have been *minimized* by summarization of data and delegation of control authority to lower level executives.

Graicunas Law

Consider a manager with M subordinates. The number of interactions (relationships) that may influence the managers decisions may be calculated as follows considering:

1) manager - to - employee
2) employee - to - employee
3) employee - to - group of employees

$$R = M(2^{M-1} + M - 1)$$

Ex: Given 5 subordinates – (using trunckated polynomial)

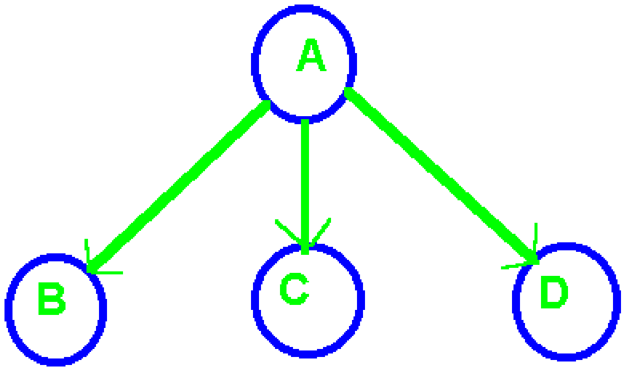
$R(5) = 5(2^{4} + 4) = 100.$

R(6) = 222, R(7) = 490, R(8) = 1080, R(9) = 2376.

The recommended span of control for managers is four to eight!

Example: Sales personnel selling products in another's territory. District Sales Manager solves problem by creating fixed boundaries to prevent competitive interaction between personnel (eliminates employee to employee). Consider the constant 1000 errors in every release of OS360.

RELATIONSHIPS



M = 3 employees:

1 - 1	**pairs**	**cliques**	**cross attributes**
A-B	A-B,C	A-B,C-D	B X C
A-C	A-C,B	A-C,B-D	C X B
A-D	A-B,D	A-D,B-C	B X D
	A-D,B		D X B
	A-C,D		C X D
	A-D,C		D X C

$$R = M(2^{M-1} + M - 1)$$

Employees	Relationships
2	6
3	18
4	44
8	1,080
12	24,708
20	10,486,140

Application to design: **allow any module to call any other module directly or indirectly!**

Reducing Complexity

To reduce complexity requires that the number of interactions between modules be reduced. Complexity can be greatly reduced by limiting the path for control and data from one element to another to a single path. This limits the number of interactions to

$$I = \frac{N(N-1)}{2}$$

where N is the number of modules in the system.

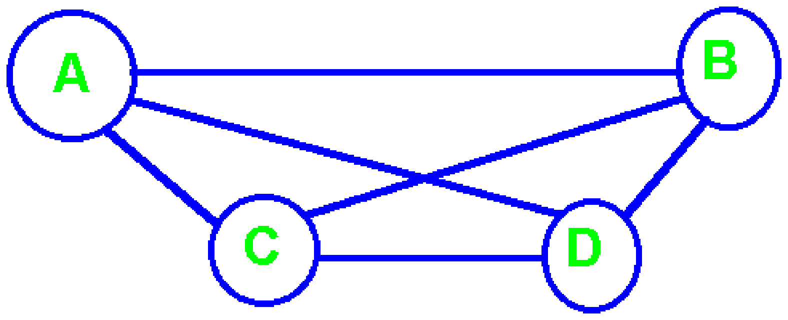
$$I \Rightarrow N^2$$

Applies to both CONTROL and DATA!

(eliminates use of global variables)

Allowing a Single Path for Interaction

Control and Data



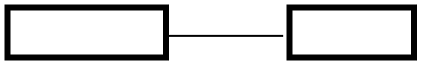
$$I = \frac{N(N-1)}{2} = \frac{4(4-1)}{2} = 6$$

Relationships		Interactions	
2	6	2	1
3	18	3	3
4	44	4	6
8	1,080	8	28
12	24,708	12	66
20	10,486,140	20	190
		200	19,900

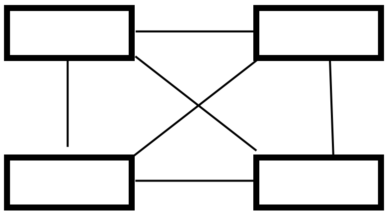
The above table indicates the number of tests for all modifications during maintenance and modification of software.

A simpler approach to the complexity of communications.

Assume only direct communications between personnel (no relaying of messages). The number of communications paths for N people is summation of the integers from 1 to N-1 or N * (N-1) / 2. For example the number of communications paths for 2 people is 1, for 4 people = 1 + 2 + 3 = 6, and for 6 people = 1 + 2 + 3 + 4 + 5 = 15. Even small increase in the size of a team greatly increase the number paths for communication of information and possibilities for introducing misunderstandings with resulting errors.



For 2 people, 1 path.



For 4 people, 6 paths or 1 + 2 + 3

The above examples assume only direct communications, no passing of information through an intermediary. Now consider the problem of miscommunications when passing information from one person to another is done correctly (both parties have the same idea) with a 2/3 probability. The probability of any two people correctly communicating is only 2/3. The probability of a second person communicating the information to a 3 person is only 2/3 * 2/3 = 4/9. The general case for "n" people is $(2/3)^{n-1}$. Hence the probability of a communication from the first person on a team reaching the 4th person accurately is $(2/3)^3$ = 0.296296296296, less than 1/3. Team organizational structures must reduce complexity and communications paths to improve the probability of accurate communications.

Project Failures

In 1995 Chaos Report published by the Standish Group using a sample size of 365 respondents including large, medium, and small companies across most major industries found that only about 16% of software projects were completed on time and budget with all initially specified features and functions. They further reported the four most significant factors leading to successful projects were (52.4% of respondents):

1) **User involvement.**
2) **Executive management support.**
3) **Clear requirements statements.**
4) **Proper planning.**

The top 3 reasons for failure for software projects were listed as (37% of survey participant's classified projects as challenging):

1) **Lack of user input.**
2) **Incomplete requirements and specifications.**
3) **Changing requirements and specifications.**

The most often cited reasons for projects that were ultimately cancelled (36% of responses):

1) **Incomplete requirements.**
2) **Lack of user involvement.**
3) **Lack of resources.**

Software Product Failures

Software projects fail for many reasons including not only cost and schedule over-runs but also product failures. In 1996 Casper Jones in a famous paper/book reported the following reason for product failures:

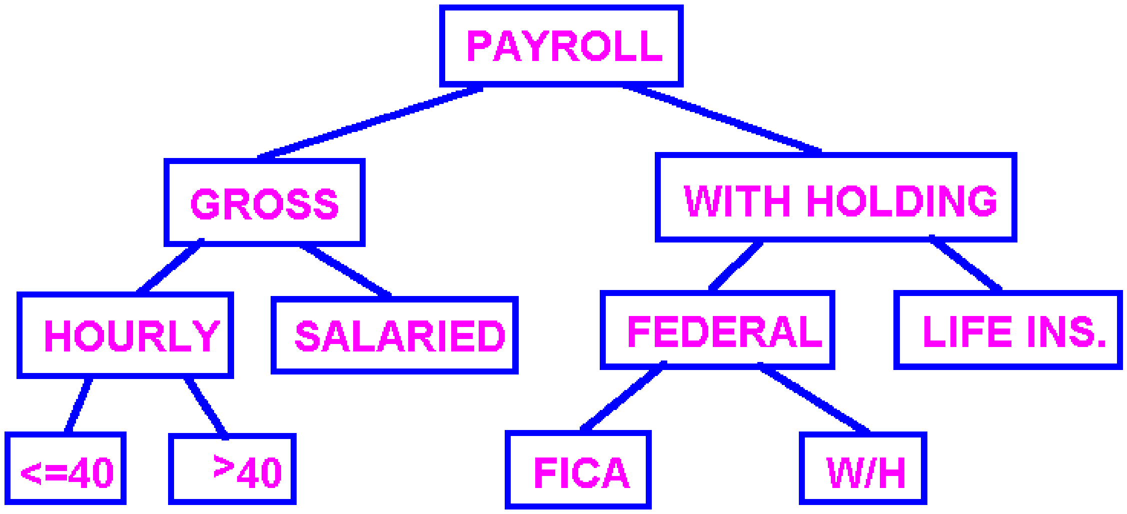
1) **Requirements errors: 12.5%**
2) **Design errors: 24.17%**
3) **Code errors: 38.33%**
4) **Documentation errors: 13.33%**
5) **Bad-fix errors: 11/67%**

Jones noted the numbers by themselves are misleading. Fixing a requirement error not detected prior to design and coding is far more expensive than fixing a coding error. **Formal reviews of specification and design documents appear to be the most productive means to prevent errors migrating into the coding phase of a project.**

The U.S. General Accounting Office report to the Committee on Armed Services of the U.S. Senate three basic management strategies as key to ensuring the delivery of high-quality software on time and within budget include:

1) Focused attention on software development environment(s)
2) Disciplined development processes
3) Methodical usage of metrics to gauge cost, schedule, and performaqnce targets

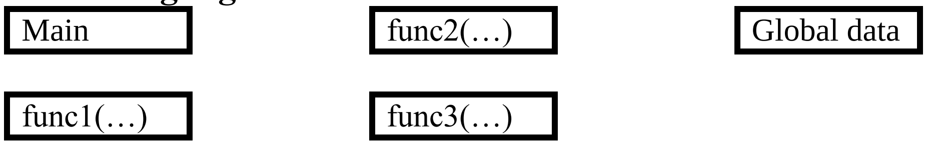
To Reduce Complexity, Traditional Logic has Emphasized the Use of Hierarchical Designs



Hierarchical Designs minimize problems with cross-communications better than mosaic (network) designs.

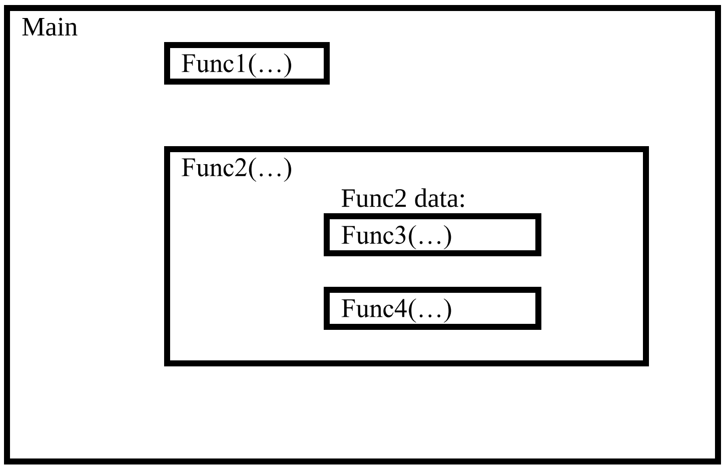
Miller/Graicunas Influence on Language Design

"C" Language

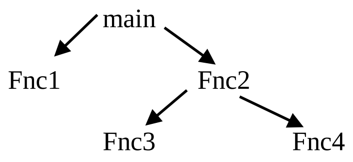


Functions stored separately or grouped into libraries. Uncontrolled global/external scope to functionality and data.

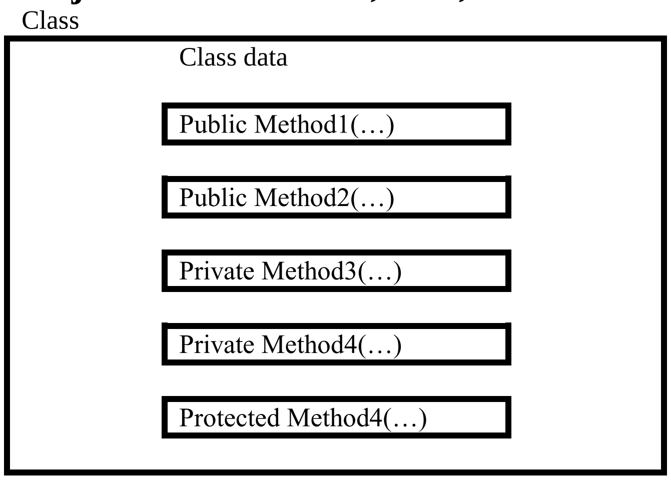
Pascal/Ada/Modula: (Nicholes Wirth – Pascal & Modula)



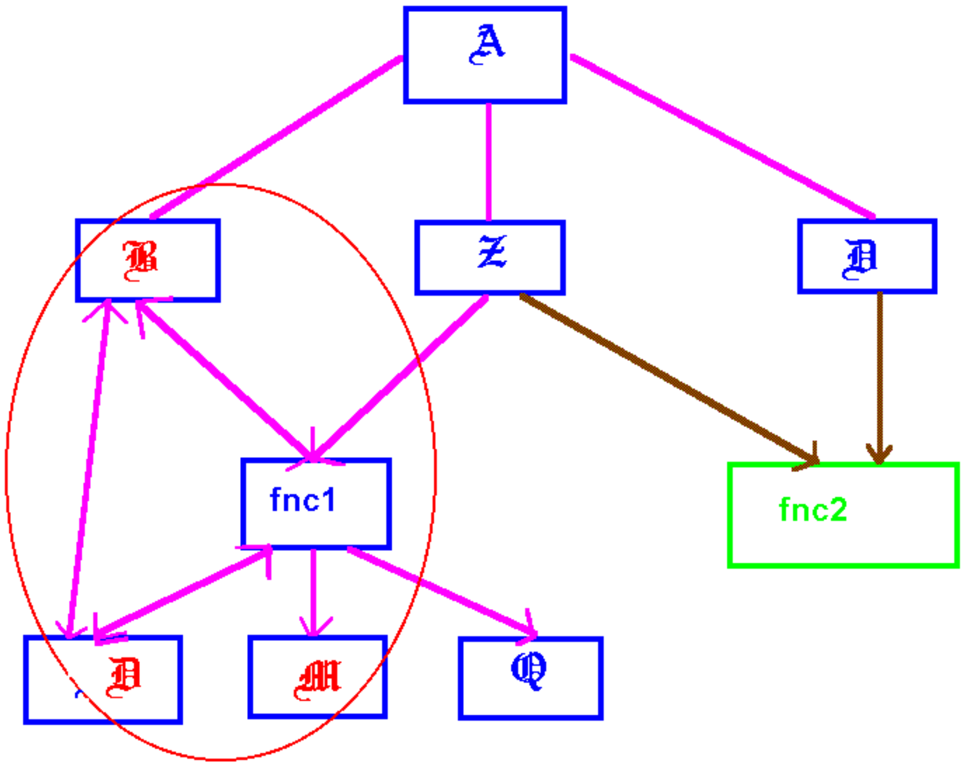
Restricted access, limited scope. Function/procedure variables are protected. Function2 data available to Func3 and Func4 but not outside Func2.



Object Oriented: Ada, C++, SmallTalk



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



It is highly desirable that a module which is the ***target of fan-in*** be a terminal module as is the case for "fnc2."

Fan-in is a primary goal of design. Every instance of fan-in means that duplicate specification, design, code, and debugging has been avoided. These advantages carry over into maintenance and modification.

A module that is the subject of fan-in should be a true black box, doing ***exactly the same function for all modules it services*** without tricky preambles or post ambles to produce slightly different results for different customers.

Modules which are both the ***target of fan-in and fan-out*** (fnc1) are far less desirable as they become subject to the effects of ***Graicunas Law*** and the complexity increases exponentially. Not only is original implementation, maintenance, and modification difficulty increased, but also packaging (optimization for space) as well as optimization for time becomes more complex.

This is made worse in the case of "fnc1" as it also participates in multiple module loops.