INTERNATIONAL ATOMIC ENERGY AGENCY
UNITED NATIONS EDUCATIONAL, SCIENTIFIC AND CULTURAL ORGANIZATION
INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS
I.C.T.P., P.O. BOX 586, 34100 TRIESTE, ITALY, CABLE: CENTRATOM TRIESTE

UNITED NATIONS INDUSTRIAL DEVELOPMENT ORGANIZATION
INTERNATIONAL CENTRE FOR SCIENCE AND HIGH TECHNOLOGY
c/o INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS 34100 TRIESTE (ITALY) VIA GRIGNANO, 9 (ADRIATICO PALACE) P.O. BOX 586 TELEPHONE 040-224572 TELEFAX 040-224575 TELEX 460449 APH I

SMR/474 - 10

COLLEGE ON "THE DESIGN OF REAL-TIME CONTROL SYSTEMS" 1 - 26 October

STRUCTURED DESIGN

P. BARTHOLDI
Observatory of Geneva
CH-12900 Sauverny
Switzerland

These are preliminary lecture notes, intended only for distribution to participants.

Structured Design

Paul Bartholdi
Observatory of Geneva
CH-1290 Sauverny
Switzerland

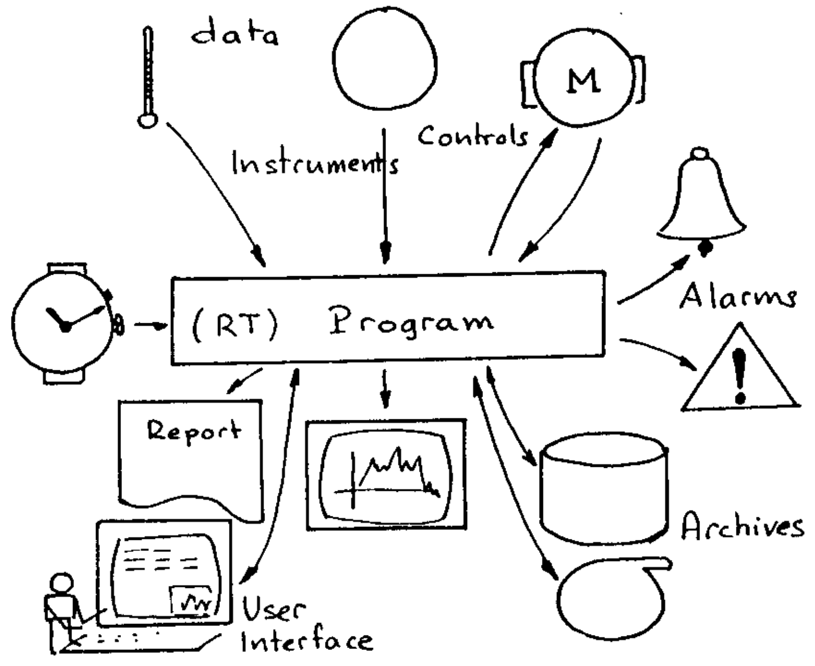
Preliminary notes - Trieste - october 1990

College on "Design of Real-Time Control Systems"

Contents

`bartho@obs.unige.ch` `bartho@cgeuge54.bitnet` `20579::ugobs::bartho`
tel +41 22 755 26 11 fax +41 22 755 39 83

1 Introduction



Structured Design encompass all aspects of the figure above, and not only the program box.

The model we will follow is **Top Down Design Analysis with Stepwise Refinement** for both **Data and Algorithms**.

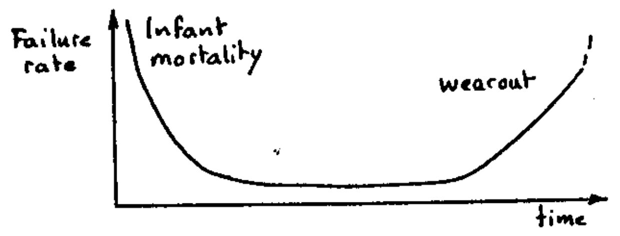
A good design should avoid that the final product look like a patchwork with all sorts of pieces loosely linked to each other, many holes and no possibility left to grow with time.

Before starting any design, the problem should be analysed with both the originator and the end users, to define carefully and precisely :

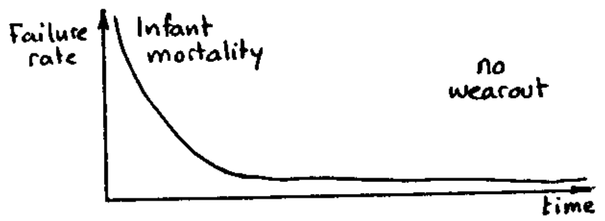
1. What is (are) the problem(s), which goal(s), keeping a global overview of the situation.
2. What are the data, both incoming and outgoing, how are they related, arranged, where are they.
3. What is available for testing, what are the foreseeable futur changes and enhancements, how the maintenance will be done

1.1 Comparision Software/Hardware

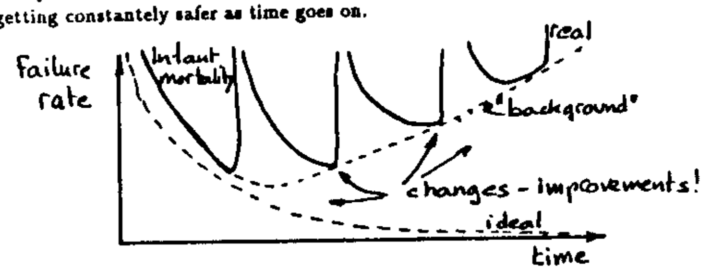
It is interesting to compare hardware and software getting older, that is how a single product ages.



A typical hardware has a large failure rate due to "infant mortality" when new, that gets rapidly smaller. Then the failure rate remains constant for quite long, up to the time some parts get broken again because of age. Parts can be replaced giving new youth, without much interaction to the rest of the system. Infant mortality is caused both by design and construction errors.



Software presents a very similar curve in the early phase, but we would expect the absence of wear out, getting constantely safer as time goes on.



What is observed is quite different. For various reasons, most software are constantely updated, and each change brings a new spike of errors that decays slowly. Even worst, the "background level" increases with time.

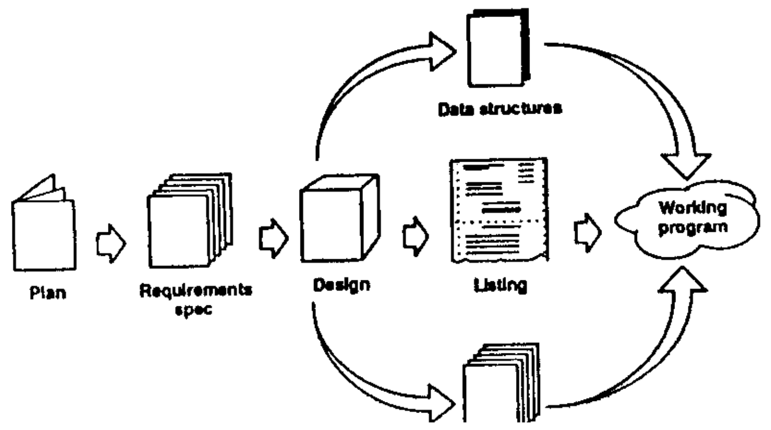
If the software by itself does not wear out, there are also no spare parts.

While most modern hardware are made of standard components and copied in a very large number of pieces, many software are still custom built from scratch.

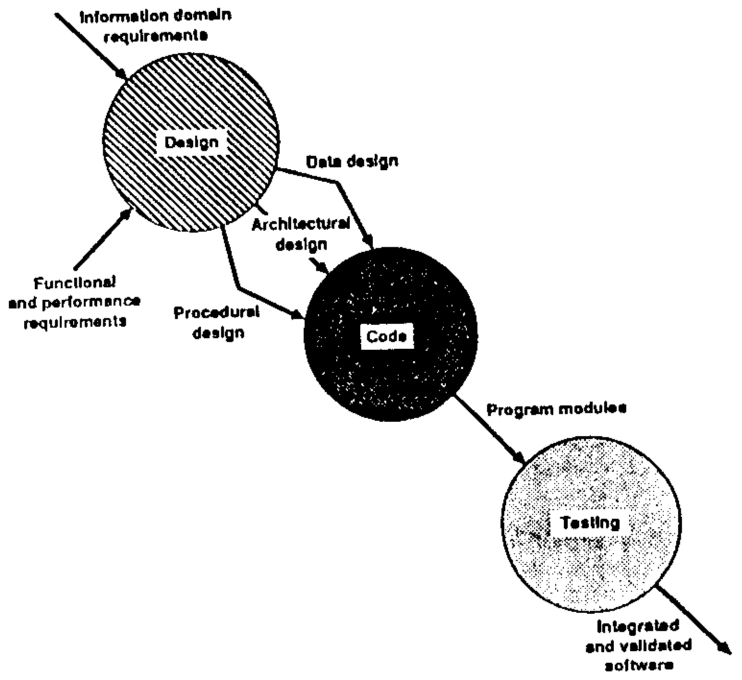
2 Some Precepts

This is a summary of the lectures on Programming Technics and Tools given during the previous Colleges on Microprocessors.

- Don't lose sight of the forest for its tree
- Program in haste and debug forever
- Starting afresh is usually easier than patching an old program
- Consider together both data and operations on them
- Each subprogram should do only one task, but do it well
- Each subprogram should hide something
- Use "drivers" to simulate (lower level) subprograms
- Keep input and output as separate modules
- Make program reading easy to do
- Keep documentation concise but descriptive
- Always name variables and subprograms with greatest care
  - self explanable, in particular for subprograms and global variables
  - short for locals, do-loops, indexes
  - use common/systematic pre/suffix
  - avoid meaningless words and misspellings
  - avoid error prone names (l ("El"), 1 (one), O ("O"), 0(zero), ...)
- The quality of test data is more important than quantity
- Do not ignore the illegal, unprobable, impossible values, check them always
- If your program is too slow, find the 10% where you spend 90% of the time



3 Design fundamentals


Information domain requirements
Design
Data design
Architectural design
Functional and performance requirements
Procedural design
Code
Program modules
Testing
Integrated and validated software

The design should:

1. exhibit a hierarchical organization that makes intelligent use of control among the elements.
2. be modular, logicaly partitioned into elements that perform specific functions and subfunctions, or are related data
3. contain distinct and separable representation of data and procedures
4. lead to modules that exhibit independent functional characteristics
5. be designed by informations obtained during the requirement analysis

3.1 Development cycle

Any project, any system, should be considered in a wider circle than just program design. It starts with requirement gathering, and end many years later, after many updates, improvements etc, when fully replaced by a new product. The following figures show how all phases may have feedback into previous ones. Even the requirements gathering never ends, nor does designing, coding, refining, testing and maintening.

The last figure is taken from a book by Galileo Galilei (1638) concerning a test bed for measuring the bending of a beam under weight.

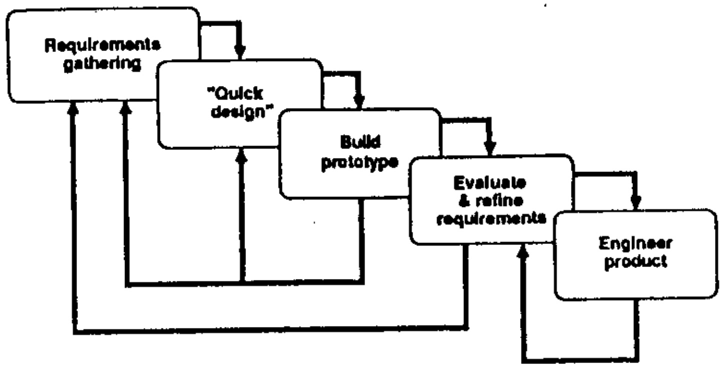


Figure 1: Global product cycle

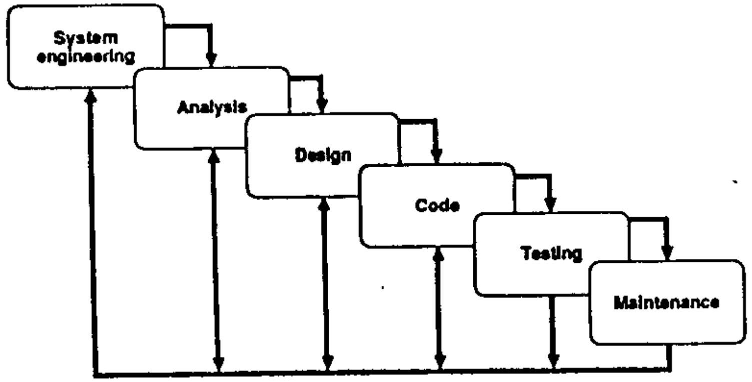


Figure 2: Software cycle

The last figure is taken from a book by Galileo Galilei (1638) concerning a test bed for measuring the bending of a beam under weight.

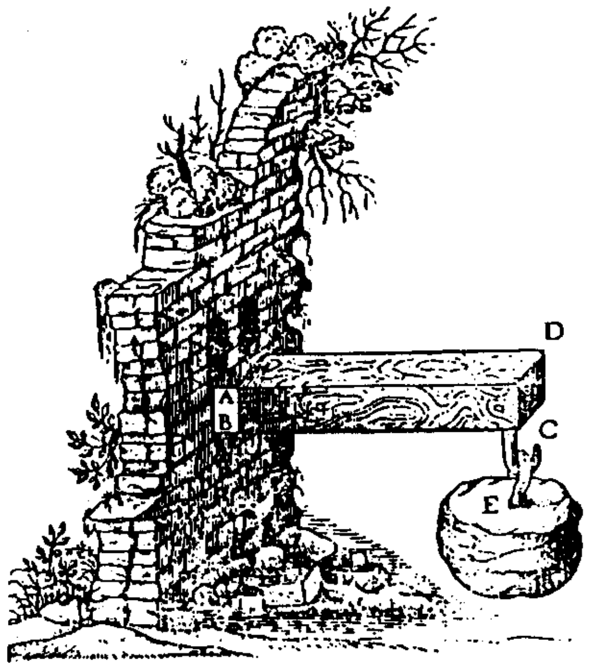


Figure 3: From *Discorsi e Demonstrazionioni matematiche intorno a due nuove scienze*, Leiden 1638

3.2 Methods

The following elements are various methods developed to help organize a well structured desgin. They should usualy be combined,using more intelligence than mechanical obedience.

3.3 Program structure

Build a static representation of the link between all modules, that is which modules call which module, which are called by which module.

This is necessary for testing, debugging, updating ...

It will look like a tree, with the main program at the top (the root), and the modules that stand alone at the bottom as leaves.

It could also be presented as a cross-reference table.

3.4 Data structure

It represents the logical relationship among individual elements of data, possibly on many levels.

It will dictate the organization of data, the access methods, their degree of associativity, and possibly processing alternatives.

If the data are numerous, then it is worth building a **data dictionary** with definitions, relations among them, constraints on elements, operations on them etc.

If applicable, build a data structure library with definitions (declarations) and operations for complex structures like lists, FIFO, stacks, tables, tree etc.

3.5 Stepwise refinement

This is probably the key that should be applied to all aspects of the program developpement.

The program is developed by successively refining levels of procedural (and data) detail, by decomposing a macroscopic statement of a function in a stepwise fashion until programming statements are reached. As tasks are refined, so the data may have to be refined, decomposed or structured. Both procedure and data specification should be refined in parallel.

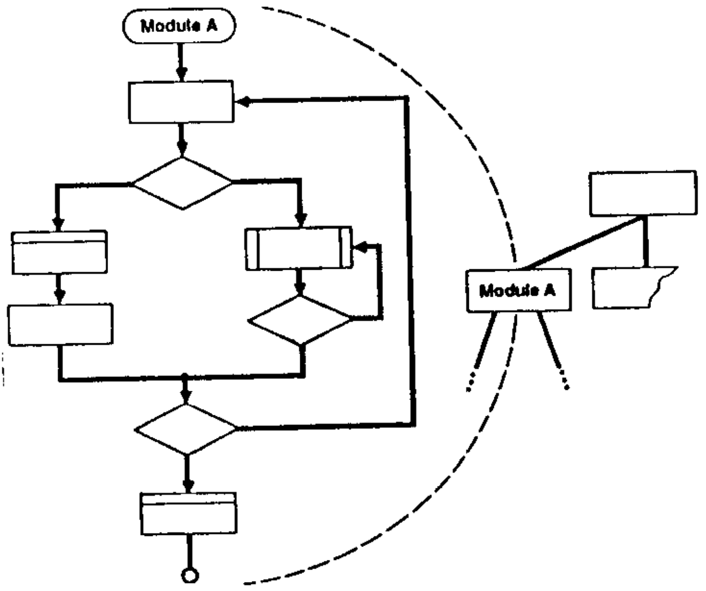


Figure 4: Module refined into substeps

3.6 Modularity

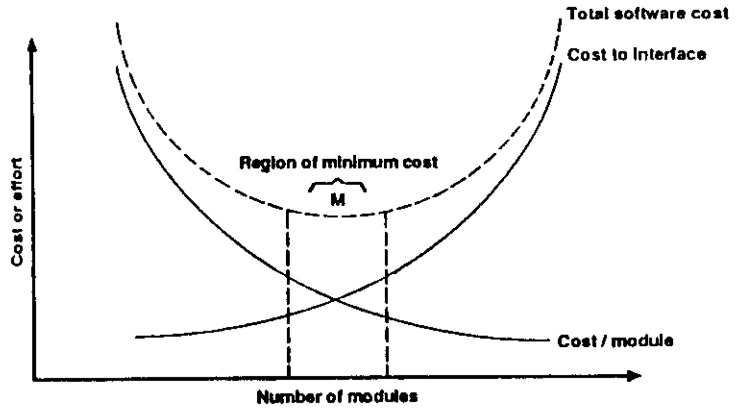
A module should

- be intellectually manageable

- do a single task, with single entry and outlet
- be complete by itself
- hide all unnecessary information (both procedures and data) to higher levels
- be easily testable

Use Stepwise refinement to reduce all large modules to more manageable ones.

Note: Their is a tradeoff between the size and the total number of modules. If the total number of modules becomes too large, the relations between the modules become unmanageable.



3.7 Abstraction

Do not specify the detailed structure of both data and procedures while it is not necessary. Keep them for the lowest level possible.

3.8 Information hiding

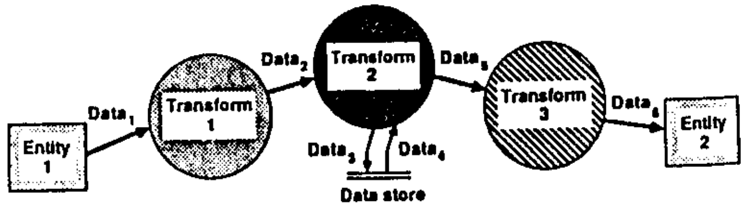
Higher levels should see only **WHAT** not **HOW** the procedures and data structures are.

4 Data Flow Design

This emphasises the dynamical aspect of the data processing.

It is best for sequential processing of not too complex data structures, including real-time etc.

The main idea is to follow the successive transformations, spread and agglutination of data, from the external sources to the external sinks.



□	External entity:	A source of system inputs, or sink of system outputs
○	Process:	Performs some transformation of its input data to yield its output data
⌒	Data flow:	Used to connect processes to each other, to sources or sinks; the arrowhead indicates direction of data transfer
⩶	Data store:	A repository of data, the arrowheads indicate net inputs and outputs to the store

Use stepwise refinement to go from the simplistic overview ($\xrightarrow{in}\text{(OP)}\xrightarrow{out}$) to fully detailled model.

When we move from the data flow to the program design, we will have to do three levels of factorization:

1. identify all input controllers, all output controllers and the transform center as the three main modules.
2. map each individual node of the data flow into corresponding modules. Begining at the transform center, and moving outward along incoming, then outgoing paths, map transform into subordinate levels of procedures.
3. Apply common sense and good design strategy to clean up the final design, mostly regrouping or refining some modules.

Note: Transforms are usually of two kinds

1. Pure transform: each individual set of data is successively transformed into its final form

2. Transactional: a single element trigger information gathering and spreading that is not directely related to the original data.

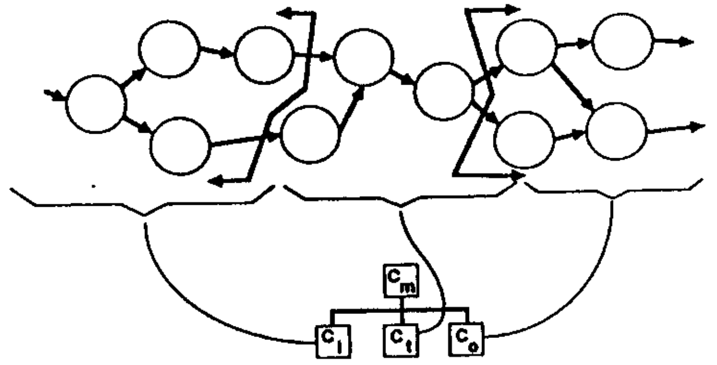


Figure 5: From dataflow to modules

A distributed, microprocessor based *patient monitoring system* is to be developed for a hospital.

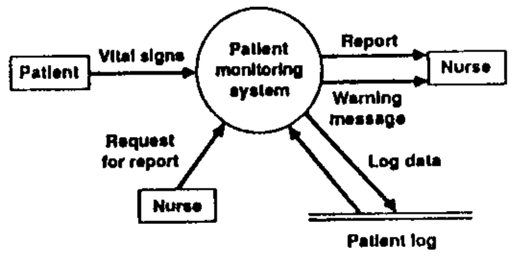


Figure 6: Example of successive refinements of the data flow for a patient monitoring system, global view

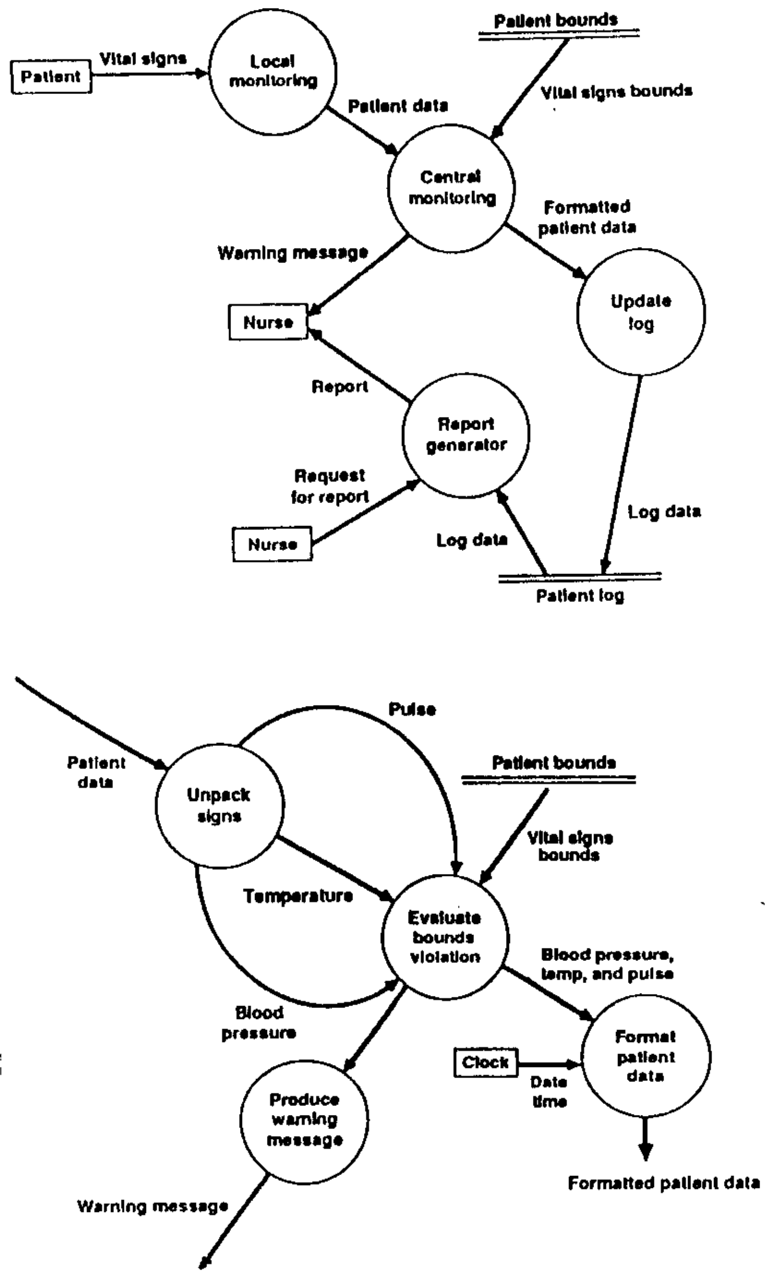


Figure 7: Two successive refinements

5 Data Structured Design

Data Structured Design emphasizes the (complex) static relations between the elementary data.

It is best applied when the data have a well defined, hierarchical structure.

Use a structure that maps the real world.

Separate the logical structure from the software implementation that is language dependant.

Postpone any unnecessary detail to the lower levels, hide them to the higher ones.

Use stepwise refinement to go from global to atomic data.

At each level, associate the possible operations like

file	open, close, rewind
record	read, write, modify, lock
element	compare, zero ...

6 Object Oriented Design

6.1 Overview

Object Oriented stuff is one of the latest invention in the trend of better software developments.

An object is a component of the real world mapped into the software domain with both data and operations.

An object is made of three components:

1. Data structure
2. process on the data (operation, manipulation)
3. invocation – message passed to the object to get something done

- The data structure and the algorithms used for the operations are both completely hidden to the user.

  The user knows only the name of the operations, and can only send "messages" about what should be done and not how or any further detail.
- All objects belong to a class of similar objects and inherit the data structure and operations defined for that class. An individual object is an instance of the larger class, for which we need only specify how the new object differs from the class.

So we can build a catalog of "software IC", objects created by reusing more general classes and adding refined data structures and operations.

6.2 Object description

1. **Protocol description** (for the user) establish the interface by defining each message the object can receive, and the related operation the object should perform when it receives the message. This is made of the message name (= procedure name), with the formal list of parameters, and carefully documenting comments that describe informally the message.
2. **Implementation description** (for the implementer=supplier) show the implementation detrails for each operation, including data structure and procedures.

This produce so called **encapsulated data structure.**

6.3 Methodology

Note: This applies to any level of the design, including the whole system.

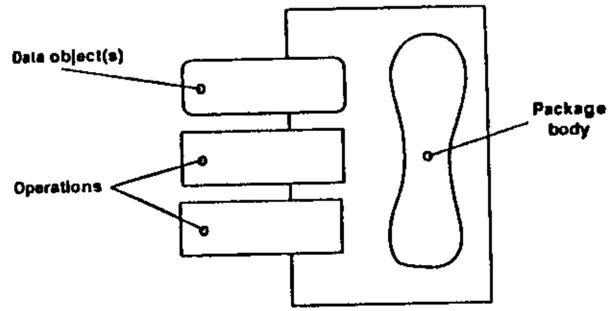
1. Define the problem precisely, but informaly
2. Develop an informal strategy for the software realization of the world problem
3. Formalize the strategy:
   (a) identify objects and their attributes (data, procedures etc)
   (b) identify operations that can be applied

   (c) establish interfaces by showing the relationships between objects and operations
   (d) decide on detailled design that will provide an implementation description for the objects
4. Reapply steps 2,3 and 4 recursively until a complete design is created (– inheritance)

Note:

- use the same word for similar messages, example: READ for disk, tape, keyboard, switch, ADC etc.
- common nouns in informal strategy correspond generaly to classes, proper nouns to instance of a class.
- verbs, predicates, correspond to procedures (messages)
- objects can be made of data but also of processes. Example of message to process: START, STOP, WAIT, DIE etc.
- if two or more objects are required for an operation, then determine which object's underlying implementation (private part) must be known to the operation (example: ADD a record to a file, *add* belongs to the file, not to the record).
- if an operation requires the knowledge of more than one type of object, then the operation is not functionally cohesive and should be rejected ...

6.4 Graphical representation for OOD: Booch diagram



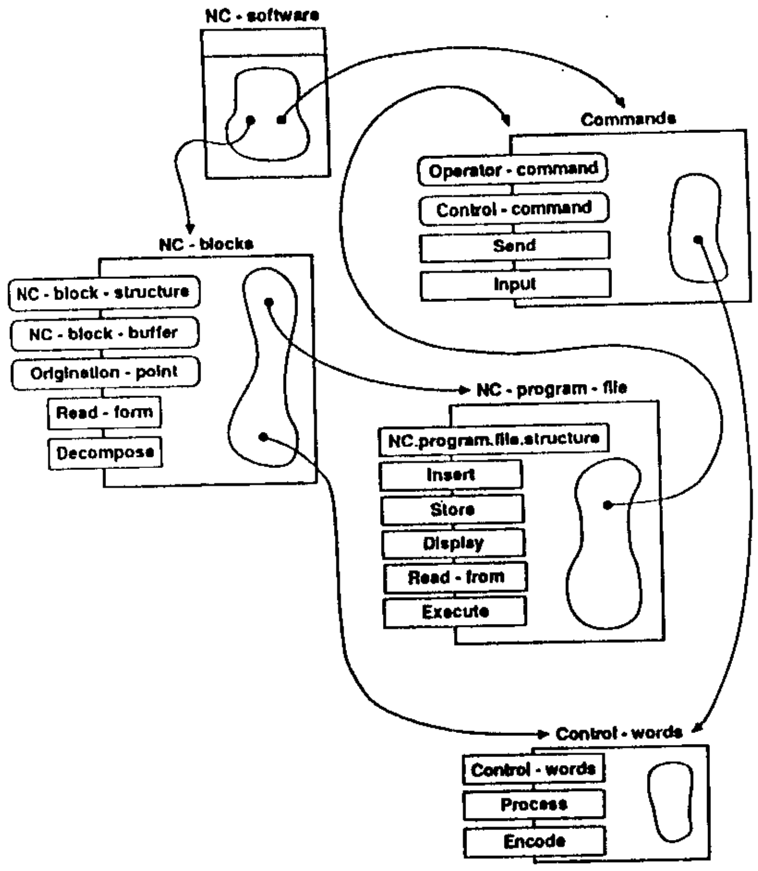
6.5 Package

A Package is a coherent set of data objects and operations. This concept was introduced in modern languages like modula2 and Ada. Then the only thing available to the user are the lists of messages to create instances of objects and manipulate them.

For example, a package for vectors and matrices will provide tools to define, read, write matrices, add, substract, multiply, invert, transpose them, calculate eigenvalues and vectors, perform singular value decomposition etc.

An other package for strings will provide similar tools to define, read and write them, concanate, truncate, trim, search ...

A third package could help with histogrammes, clear them, add new values, print them, send them to an other graphic package ...



6.6 Remarks

The first computer languages (algol, fortran, cobol) put all their effort on algorithms and program structures. The next generation (pascal, etc) introduced the notion of data structure into the language, but would not provide the tools to bind together data and operation on them. The latest ones (algol68, modula2, Ada etc) have added the notion of package as seen above, whithout specifying the entity of objects. C++ and ObjectC are preprocessor implementation of objects for C leaving available all the power (and pitfall) of the underlying language. NEON (a forth extension dialect with objects) could be a nice experimental system to work with the notion of objects.

The MACH operating system from the Carnegie Mellon University, a unix derivative used for the NEXT workstation, is entirely based on objects.

In an other area, Objects with their data and operations are now recognised as basic elements in relational data bases (Ingress).

A lot of discussion is still taking place concerning inheritance of procedures. Should they take place

at compile time (early binding) or at execution time (late binding)?

While Object Oriented Languages may not be available, Object Oriented Design methodology can be used manually for all the implementation on any language.

6.7 Summary

An object is a nice way to do data abstraction and procedure hiding.

An object lets you see only WHAT (what is the object, what can be done with it) and not HOW (how it is implemented, how the operations are done).

At the same time, it clearly separates data and operations on them, and explicitly defines the interface between the various objects.

The main advantage of OOD, if it is done in a clean way, is that both the data structures with their attributes, and operations can be changed, updated, adapted to new extended conditions without impairing the rest of the program. This is certainly a key advantage when the external conditions (real world, hardware) are changing much more rapidly than the programs themselves.

7 Design for Real Time Systems

Real Time Systems are characterized by:

- parts of the operations are driven by incoming events (data, interrupts, alarms ...), that must be served in time,
- the whole system is made of many tasks that run concurrently, synchronizing themselves with others, starting new tasks, killing others etc.
- data must be exchanged rapidly between tasks, with locks when necessary.

All the methods described earlier can be used for Real Time, providing one adds timing informations to data flows, data structures that can be shared or exchanged between tasks, and synchronization operations.

7.1 Synchronization and Communication

Synchronization and communication use generally variants of the following tools: semaphores, mailboxes, messages.

Semaphores have been introduced very early by Dijkstra for *THE* operating system. The idea is to have a set of global variables, and two "atomic" (that is when started, the operation will always be terminated before any other operation is started) operations on them: `procure` and `liberate`.

`procure(ev)` looks at variable `ev`. If it is zero, it is set to one and the operation is terminated. If is is not zero, the calling task is suspended up to the time it becomes zero again.

`liberate(ev)` reset `ev` to zero.

Suppose a task T1 wants to write into a buffer X, an other T2 to work with X when it is ready, without being disturbed by T1 or any other.

Then we create a global variable eX, initialized to one. T2 will try to get X with `procure(eX)`, but will be suspended. T1 will write on X, then `liberate(eX)` and T2 can work on X. T1 may want now X again, using `procure(eX)`, but it is locked, and it will have to wait for T2 to execute `liberate(eX)` etc.

A mail box is just what we have built above, the association of a semaphore and a buffer in-memory.

Note that both semaphores and mailboxes can be done in software or in hardware, in particular when many processors need to be synchronised.

In the third case, one process sends a message to another one. The later is automatically activated by the operating system to process the message. The first may wait for an answer or continue in parallel with the others.

"Messages" can be real buffers, or just pointers.

Finally, all the previous tools are very effective to handle task synchronisation dynamically, in an asynchronous way. This may not be the most efficient if the system is predominantly cyclical, with few or no asynchronous events. Then a careful analysis of the flow, with good timing informations, can produce a very efficient static system without any synchronization. Such a system is of course very rigid and very sensitive to jitter or glitches in the events.