Systems Analysis. (Structured Systems Design)

OBJECTIVE: To explain the main methodologies for designing structured systems

We will analyse a system by studying & determining the flow of data through the system, using Gane & Sarson's method. Note: we define a client as a person / organisation that wants a program or system (collection of programs) developed.

SYSTEMS ANALYSIS is the process whereby clients needs are extracted, analysed & refined. These needs must be presented in a format which is understandable to both the client & the designer.

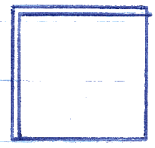
SYSTEMS DESIGN is the process whereby the document produced by the systems analyst is converted into specifications to be used by the programmer.

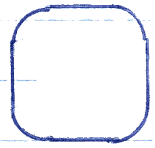
CODING is what the programmer does.

Sometimes a BUSINESS ANALYST acts as the interface between the client and the systems analyst.

1. DRAW A DATA FLOW DIAGRAM

This must reflect the present system. It is a logical model of the flow of data, and has little to do with physical implementation which comes later.


Source or destination of data


Process which transforms flow of data


flow of data


Store of data.

To avoid crossing lines draw the symbol more than once, and add lines for identification

The original DFD should then be expanded after consultation with management and staff – it is thus essential that management and staff can understand the DFD.

For simplicity the creation and maintenance of files should not be shown on the top-level DFD. However, each process in this top level can be exploded further (and so on), and file management can be shown on these lower levels.

ERROR CONDITIONS should be left till later, as these add too much detail – they should be included in the 2nd & 3rd level diagrams. This simplifying rule is essential.

Once the DFD is complete, the automation boundaries must be determined, as well as what type of automation (batch / online / bureau / dedicated) to use.

The DFD can thus be used to illustrate possible system options – management (or users) should decide which system they want. The DFD should thus reflect what the business actually does, not what the analyst thinks that it does.

2. Design a Data Dictionary.

The data flowing through the DFD should be specified and broken down in the data dictionary. This should omit physical detail, but should be precise enough for the analyst and user to review errors and omissions

3 Define the Logic of the Processes.

Each of the processes should be described in greater detail (possible methods - decision trees, tables; structured English)

4 Define the Data Stores.

The exact contents of each data store should be specified. These should be as simple as possible. They will later go into the data dictionary.

The value of different types of accesses should be determined, and all immediate accesses required must be defined.

5 Define the Physical Records.

Once we know how each part of the system is to be realised i.e. - manual parts
- automated parts (batch, on-line, etc)

We must specify for each file:

- the file name
- organisation (sequential, indexed, etc)
- the storage medium
- the records — no of fields
  - field name
  - contents
  - format
  - key
  - ascending / descending
  - comments

6 DETERMINE THE I/O SPECIFICATIONS.

Input forms and screens must be specified, with respect at least to components, if not to detailed layout. Printed output must be specified, where possible in detail, and certainly as to estimated length.

7. PERFORM SIZING

- Determine the volume and frequency of input
- Determine frequency of each printed output
- Determine the number & size of records that pass between CPU and mass storage device
- For each file
  - determine the number of records
  - and thus the file size.

8. Determine the Hardware Requirements.

Ideally, the hardware can now be specified. In practice, however, the company may already have the hardware, or the hardware decision may have had to be made much earlier because of long delivery dates.

We have accumulated all the information on:

- input volumes
- output volumes
- mass storage volumes (tapes, disks)
- type of file access, and buffer size implications
- no. & size of programs.

A required response time for the system (or subsystems) must be obtained from the end-user, as well as those response times which are critical.

The user should be presented with various possible systems balancing response time and cost, and should be involved in the decision-making.

With the volume and response time information; the hardware needs should be determined:

- input devices and speeds
- output devices and speeds
- number, type and speed of mass storage devices.
- size of main memory
- speed of CPU.
- impact of operating system (paging, etc)
- type and configuration of OS (I/O channels etc).

The remaining stages are:

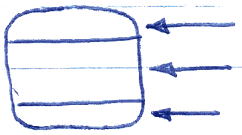
- design the programs (CS3)
- code the programs (CS2)
- test the programs
  - module testing (CS2)
  - integration testing (CS2)
  - acceptance testing (CS3)
    (actually test the system with the people who are going to use the system in a simulated environment)

9. More on Data Flow Diagrams.

9.1 Data Flow:

- each data flow is a pipe through which data flows
- each data flow should have its contents written alongside
- in early versions write DF descriptions in lower case.
- in later versions of the DFD, when data has been defined in the data dictionary, write the description in capitals.
- often different parcels of data go along the same path. Sometimes it is difficult to characterise the contents of a data path. If many different types of data flow between two entities, it is unwieldy to use seperate paths for each type - rather use a routing process, or process each type seperately.

9.2 Processes



9.3 Explosion Conventions.

- Use process # . id # (eg number processes inside process 4 as 4.1, 4.2, etc).
- Draw lower level processes within the boundary of the upper level process
- Show all data flows
- Extra data flows will be needed for :- greater detail
  - error conditions, etc.
- Data stores should be shown completely inside the boundary if they are created and used by this process and no other.

9.4 Error Handling & Exception Handling.

- do this at the second level
- assess its importance though ; if it is very important incorporate into first level.

9.5 Overview of Drawing DFD's.

① Identify external entities involved
② Identify scheduled inputs and output
  - discuss logical groupings of inputs & outputs
  - mark I/O solely related to error & exception conditions

③ Identify enquiries and on-demand information requests.
specify 1 data flow for input & 1 for output

④ Use a large sheet of paper and draw a first draft by hand
- don't consider timings
- draw a system that never starts / stops (ie self-contained)

⑥ Accept that you will need a few drafts to get the picture untangled.

⑦ Check the first draft to ensure that all inputs and outputs are included (error & exception conditions excluded)

⑧ Produce a second draft, using a template and minimising cross-overs

⑨ Conduct a walk-through with users).

⑩ Produce lower-level explosions - handle errors, exceptions, etc.

⑪ Produce final version

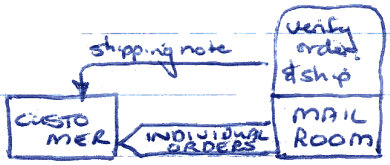
⑫ If investment is worthwhile, produce artists' copy.

10 MATERIALS FLOW.

The materials flow should be done seperately from the data flow. However, we need to describe the logical functions which transform the material at each point and indicate the associated data flow (although the processes may be different from those on the DFD, and some materials flow can occur without data flow).

Target:

example:



11 DATA FLOW ANALYSIS.

Each DFD process will become a program (or portion of a program) ie, a module. We want modules of high strength and low coupling. We use data flow analysis to achieve this.

At this stage we have complete precise information as to the input and output of each process (ie we know what we expect / require.

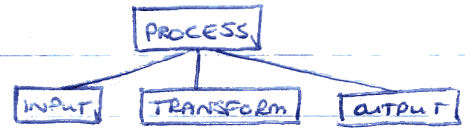
Schematically we can represent the process as a path:

We must start at the input and output points.

and work inwards finding the points of highest abstraction (for input this is the point at which it loses the quality of being input and becomes just data being operated on by the process, for output it is the point at which the data may first be considered output). The middle section of this decomposition is called the transform module. We thus have three modules:

- input module
- transform module
- output module.

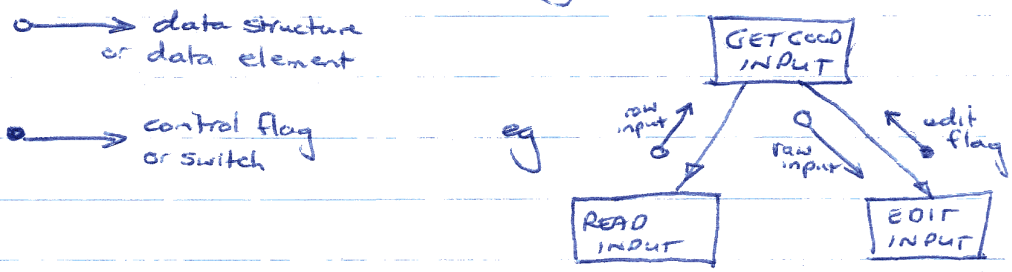
Using top-down design:



If necessary, repeat this process with each of the three modules to decompose them further. Continue this process of stepwise refinement until each module has a single function (ie high strength).

It may be necessary to make a few minor modifications to the final decomposition to achieve the lowest possible coupling. Coding can now be performed.

Note: convention used by Yourdon & Constantine.

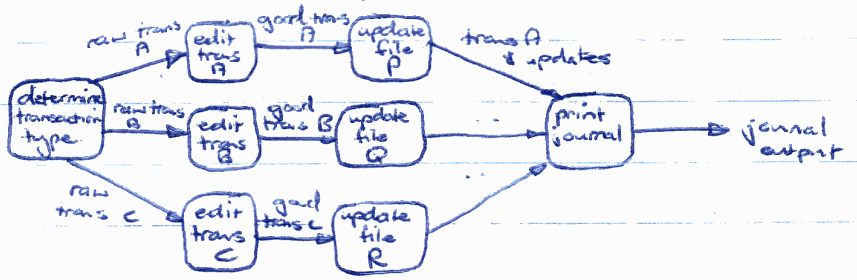


NB: As soon as we send data to a sub-module and get some answer back, we have in essence some form of computing

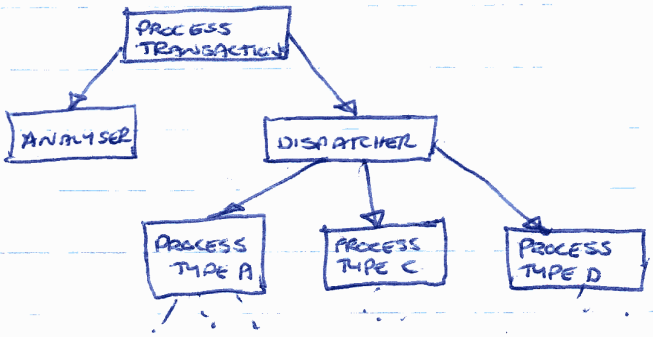
12 Transaction Centered Systems.

Not all systems are transform centered where all the transactions follow the same path. Often you have different types of transactions each of which performs a different function which is similar in outline but different in detail.

eg



This type of system is usually broken down into analyser and dispatcher modules. with the dispatcher performing the necessary routing.



13 More Complex Systems.

Sometimes there are a number of input and output streams. The easiest method is to find the points of highest abstraction for each input and each output stream.

Note that dispatcher modules should be kept simple and separate, and duplicate decision making must be avoided.

14 Jackson Structured Programming

This method of programming, developed by M. Jackson, is not structured in the conventional sense. It is a methodology a program structure from a study of the structure of the input and output data.

There are three components: notation, design methodology, & means of handling intractible cases

Notation.

The same notation is used for both programs and data. Flowcharts are used for programs, and structured flow diagrams for systems

Design Methodology

This results in a program whose structure is based on the structure of the data it uses. Unique as a design tool.

Exception Cases

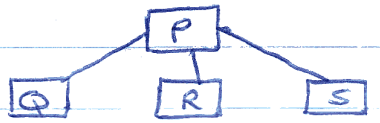
Some cases can't be handled by the design methodology, as the structure of the data is different to that of the program. Jackson provides mechanisms (inversion, backtracking) for dealing with structure clashes.

It is unusual that Jackson admits his method may not always work, and that he provides methods to deal with these cases.

14.1 NOTATION.

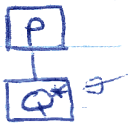
Structured programming permits 3 control structures. sequence, iteration & selection.

(a) SEQUENCE



PROGRAM STRUCTURE	DATA STRUCTURE
MODULE P.	
PERFORM Q	01 P.
PERFORM R	02 Q type
PERFORM S	02 R type
END P.	02 S type.

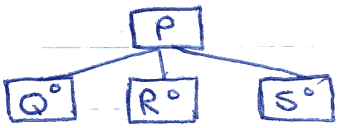
(b) ITERATION



Kleene star (0 or more)

PROGRAM STRUCTURE	DATA STRUCTURE
MODULE P	01 P.
PERFORM Q n TIMES	02 Q type OCCURS n TIMES

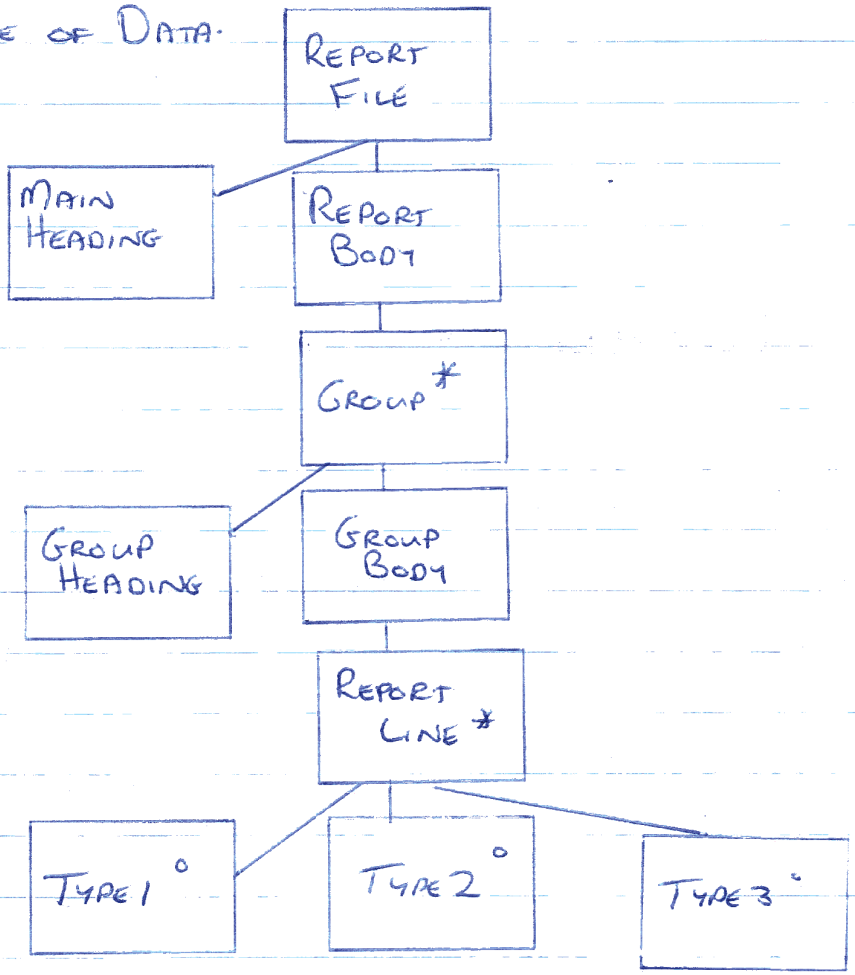
(c) SELECTION



PROGRAM STRUCTURE	DATA STRUCTURE
IF cond = c1 PERFORM Q	01 P.
ELSE IF cond = c2 PERFORM R	02 Q type
ELSE PERFORM S	02 R redefines Q type
	02 S redefines Q type

Example: STRUCTURE OF A REPORT OUTPUT FILE

STRUCTURE OF DATA:



J.S.P. is in essence the following: the structure of the program which produces the report must also have the above structure, each box being a procedure, paragraph or module.

14.2 DESIGN METHODOLOGY

4 STEPS:

1. Record your understanding of the problem environment by defining the structure of the input and output data.
2. Find a 1-1 relationship between components of these

data structures. If this can't be done, certain intermediate data structures are defined.

③ Form a program structure based on these relationships.

④ Allocate the elementary operations (prog statements) to the appropriate components of the program structure.

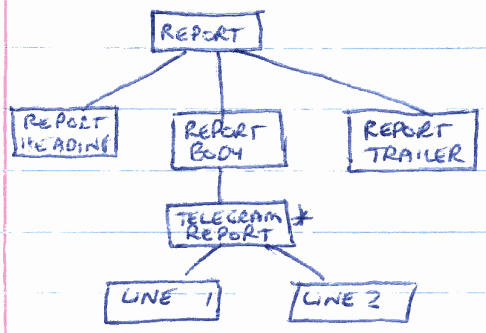
143 NON-STANDARD CASES

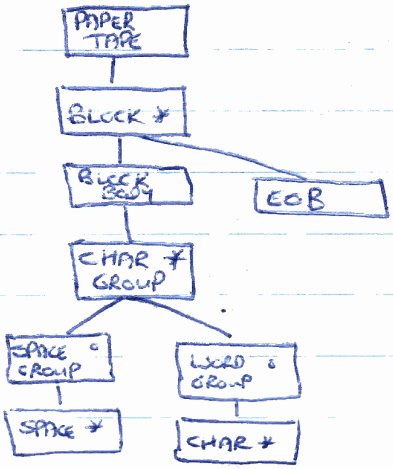
What happens when the input & output data structures dont match?

Example: Telegram problem.

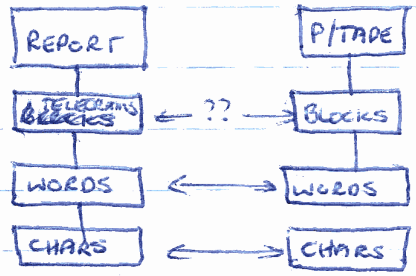
- input consists of a number of telegrams
- input device consists of blocks of 100 chars.
- each block contains a number of words seperated by blanks
- each telegram terminated by ZZZZZ, input device by EOF.
- telegram can begin/end anywhere, and span a number of blocks.
- we want to output: - heading
  - no of words per telegram
  - no of words > 12 chars per telegram
  - trailer message.

```
TELEGRAMS ANALYSIS
TELEGRAM 1
        15 WORDS OF WHICH 2 ARE OVERSIZE
    :
END ANALYSIS
```





These two data structures may be simplified as below:



There is a structure clash between blocks & telegrams.

The classical solution would be to use an intermediate file - find the highest level common denominator between the input & output structures (words in this case), and use this as the record structure of an intermediate file.

Jackson's solution is not to use an intermediate file, but instead to consider P2 as a special type of subroutine (called a co-routine) of P1. In essence, P1 produces a word and calls P2W passing that word.

Conceptually, coroutines must retain values of local variables on exit, and allow reentry to the coroutine at the statement immediately after the one from which it last exited. Coroutines do not exist in COBOL, but can

be simulated by using a state vector.

14.4 Some Comments on JSP.

1. The Resulting Programs are Unstructured.

Programs developed using JSP tend to have many GOTO's in. In general, each part of the program also reads ahead to determine where to go to next. The state vector solution to inversion problems is also very unstructured.

Thus JSP & structured programming are incompatible.

2. Stepwise Refinement is Not Used.

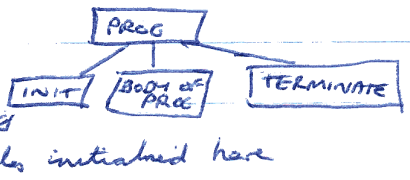
With JSP the procedures appear as a consequence of the structure of the data, with the relevant operations slotted into the diagram representing the structure of the data.

With stepwise refinement, the various procedures are a "natural development" of the refinement technique & may be developed further in turn.

3 Operations on a Specific Data Structure are Spread over the Entire Program, so JSP modules have Low Strength.

4 JSP Programs are Unnatural.

Typical JSP structure



In many JSP progs. variables are initialised twice – at the start in the INIT module, and also just before using for the first time.

5) DIFFICULT TO CHANGE THE DATA STRUCTURES.

With JSP, a major change in the data structure must necessitate a major change in the program structure. With stepwise refinement or composite structured design, the program structure should be largely unaffected by a change in the data – ie unless you are very sure the structure of the data will not change, don't use JSP.

6) JSP IS NOT NECESSARILY IN COMPETITION WITH COMPOSITE STRUCTURED DESIGN.

If you must use JSP consider:

- use data flow analysis to break the problem up into modules and obtain their input & output structure.
- use JSP to code the modules with the structure of the I/O determined above.

JSP seems to work well for small, simple examples.