SOFTWARE ENGINEERING

The economic production of real computer software systems.

Properties of a real computer software system:

- large (many lines of code, performs several seperate but related functions)
- developed by a team
- long development time
- reasonable life span.
- the system developers are not the final users.
- maintenance is required.

1. SOFTWARE LIFE CYCLE

Each step must be checked before progressing. The late an error is found, the more costly it is.

1.1 SPECIFICATION

Establish and specify:

- the requirements of the system
- the specification of the system
- operational constraints

1.2 DESIGN

Must be derived from an analysis of the specification

- individual components must be specified
- file and data structures must be specified
- interfaces between system component must be specified.
- backup & security components " " "
- hardware " " "

1.3 IMPLEMENTATION

The design must be turned into code, if possible using a high level language which runs on the target machine. The code must be to company standards.

1.4 TESTING

Individual modules tested by programmer
Module integration " " analyst
System " " user

1.5 OPERATION AND MAINTENANCE

The system must be installed, any errors found must be corrected, and changes must be made when required.

2 RELEVANT IMPORTANCE OF LIFE CYCLE STAGES

IMPLEMENTATION - 33%

- REQUIREMENTS 10%
- SPECIFICATION 10%
- DESIGN 15%
- CODING 20%
- MODULE TESTING 25%
- SYSTEM INTEGRATION TESTING 20%

MAINTENANCE - 67%

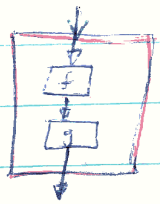
The maintenance requirements should be minimised

3. STRUCTURED PROGRAMMING

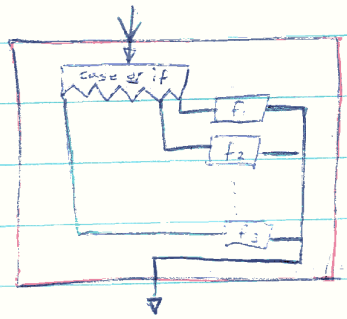
3.1) MATHEMATICAL THEORY

Any program that can be represented as a flowchart can be implemented using only 3 types of control structures - sequential, selection and iteration.

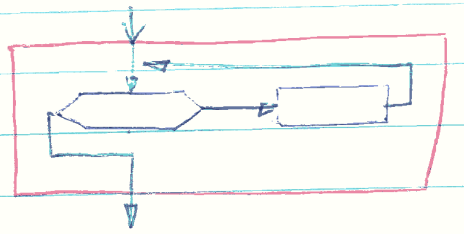
SEQUENTIAL:



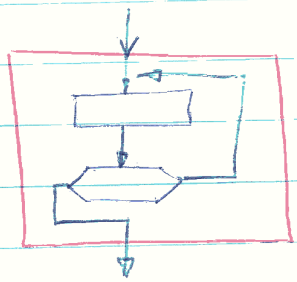
SELECTION



ITERATION



or



Note the absence of GOTO and that each block has only one entry and one exit point.

We can then say: A program is structured if program blocks are connected by concatenation, selection and iteration only, and every block has exactly one entry and exit point. Keep GOTO's to a minimum - preferably only use them when an error is detected and always in a forward direction. They should preferably only be used on detection of some abnormal condition to jump to the end of that routine.

3.2) GENERAL FEATURES OF STRUCTURED CODE

Our aim is to write easily maintainable programs. Therefore they must be quickly comprehendible and safely modifiable. As each block of a structured program has only 1 entry and 1 exit point, they can be safely modified without directly affecting any other block. Thus structured code should be easily modifiable, and easy to write, test, read and comprehend.

3.3) VARIABLES

Global variables should be kept to a minimum. Store that are only used locally should be defined locally. Input parameters should be called by value, output parameters called by name, and redundant values should not be passed through the parameter list. Variable names should be meaningful and easily distinguished.

3.4) READABILITY

3.4.1) COMMENTS

Sufficient appropriate comments should be used. Comments should be put in when the program is written, not afterwards, and should be updated immediately when this is necessary. Prologue comments should be used at the beginning of a routine to explain what the routine does, what it is called by and what it calls, files and parameters used, etc. Meaningful explanatory comments should be used within the routine, and should not

simply be paraphrases of the code but explanations of it

3.4.2) GENERAL READABILITY

Indentation should be used to display structure, and blank comment lines to split sections of code. Only one statement per line should be used.

3.5) INPUT AND OUTPUT

Input values should be echoed and checked for validity. If they are not valid, the reason for this should be reported and as much recovery as reasonably possible should be allowed.

Output layout should be clear and should contain the required information

3.6) PROGRAM DEVELOPMENT

3.6.1) TOP DOWN PROGRAMMING AND PROGRAM MODULES.

Break the problem into logical components, repeating this process until each component is easily codable. Each component, irrespective of level, is called a module. PASCAL PROCEDURE and FUNCTION can be used to create modules, although they are unable to control the visibility of variable names and do not allow the retention of local variables from one call to the next (making global variables often necessary)

The ADA PACKAGE does provide all the features required by a module. These are:

- abstract data types whose representation is concealed from the user, but which may be manipulated by the user.
- related operations may be grouped together to share variables in a controlled way.
- independent compilation of a module is possible because it is a self contained unit making no reference outside of its environment.

3.6.2) MODULE STRENGTH & MODULE COUPLING

These are two measures of the effectiveness of modules. Module strength is a measure of the 'goodness' of a module, and is used to maximise the relationships between elements in a module. Module coupling is a measure of the interconnections and relationships between modules, and is used to minimise relationships between modules.

3.6.2.1) MODULE STRENGTH

This may be separated into seven types, from worst to best. These are

3.6.2.1.1) COINCIDENTAL STRENGTH

The function of the module cannot be defined except by examining the code, as the module performs multiple

unrelated functions. This occurs when arbitrary or illogical modularization is used.

3.6.2.1.2) LOGICAL STRENGTH

The module can perform a number of related functions, one of which is explicitly selected by the calling module. This can lead to long parameter lists. The user must also be aware of all the arguments of all the functions to perform one function.

3.6.2.1.3) CLASSICAL STRENGTH

The module performs multiple sequential functions that are weakly related.

3.6.2.1.4) PROCEDURAL STRENGTH

The module performs multiple sequential functions that are not logically related but are related by the problem being solved

3.6.2.1.5) COMMUNICATIONAL STRENGTH

A module that has procedural strength with the feature that all the functions operate on the same data.

3.6.2.1.6) INFORMATIONAL STRENGTH

The module contains multiple entry points where each entry point performs a single specific function and all of

the functions are related by a concept, data structure or resource which is hidden within the module. ADA PACKAGES can have informational strength.

3.6.2.1.7) FUNCTIONAL STRENGTH

The module performs a single specific task. A functional strength module may be decomposed into lower level modules that can accurately be described by single primitive functions.

3.6.2.2) MODULE COUPLING

Again there are seven types, from worst to best. Module coupling is a measure of the relationships between modules. These relationships should be minimized.

3.6.2.2.1) CONTENT COUPLING

Two modules are content coupled if one module directly references or changes the inside of the other module (mainly occurs in assembly language programming)

3.6.2.2.2) COMMON COUPLING

The module references a global data structure. This makes readability worse, often exposes the module to more data than necessary, can make the modules logic obscure, and decreases the modules generality.

3.6.2.2.3) EXTERNAL COUPLING

Modules reference some externally declared variable (as in common coupling) except that only 1 data item is involved (Note that if only one such module exists, the variable can be made local to the module, so in general two or more modules are externally coupled)

3.6.2.2.4) CONTROL COUPLING

One module explicitly passes an element of control to another module. This implies that the higher level module knows about the inner workings of the lower level module.

3.6.2.2.5) STAMP COUPLING

The modules are coupled by a data structure which is passed in the parameter list.

3.6.2.2.6) DATA COUPLING

Only the data actually required by the module is passed to it through the parameter list, ie, no extraneous data is passed.

3.6.2.2.7) NO DIRECT COUPLING

The module is self contained and has no data passed to it.

3.6.3 MODULE SIZE

If module strength and module size were the only criteria, this would lead to large monolithic programs, which although well constructed, would be unreadable. Thus we have to compromise to achieve high strength and low coupling with reasonable module size. This can be done by correct analysis and problem decomposition.

3.7) PROGRAM TESTING AND VERIFICATION

Testing is the process of executing a program in a controlled environment. It is the most viable method of error detection. Testing can be used to show the presence of bugs, never their absence.

3.7.1) CODE INSPECTION

This is done by making up an inspection team with a chairman, the program designer, the program implementor, and the person who tested the program. The code is distributed before the meeting. At the meeting the programmer and implementor may give overviews of their task. An attempt to find (but not to correct) errors is made. The advantages this has are that many errors may be found in a single session as more individuals are checking the system, and that the errors

can be fixed data at one session.

3.7.2) ALL ENCOMPASSING TESTING STRATEGIES.

3.7.2.1) EXHAUSTIVE INPUT TESTING (DATA DRIVEN OR BLACK BOX TESTING)

Every possible input condition is tested. This is impossible as there usually an infinite number of input conditions.

3.7.2.2) LOGIC DRIVEN TESTING (CODE DRIVEN OR WHITE BOX TESTING)

The logic of the program is used to devise tests to test every possible path through the program. This is impossible as there are usually too many possible paths.

3.7.3) PRACTICAL TESTING

We need to choose a subset of all possible test cases that has a high probability of detecting most errors. There are two main methods - testing to specification (limited black box testing) and testing to code (limited white box testing). Usually black box testing is done and is supplemented with white box testing.

3.7.3.1) TEST TO SPECIFICATION. (MORE APPLICABLE TO UPPER LEVEL MODULES)

A subset of the input data is chosen using the following criteria:

3.7.3.1.1) EQUIVALENCE PARTITIONING

A well selected test case should have 2 properties:

- it should reduce, by more than 1, the number of other test cases that must be developed to achieve some predefined goal of 'reasonable' testing
- it should cover a large set of other test cases.

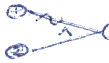
Equivalence partitioning is a technique for determining which classes of input data have common properties so that, if the program does not display an erroneous output for one member of a class, it should not do so for any member of that class. These are called equivalence classes. Testing should be done with both valid and invalid test cases.

3.7.3.1.2) BOUNDARY VALUE ANALYSIS.

Boundary value analysis is similar to equivalence partitioning, but differs in that only elements on or near the boundary values are tested, and both input and output equivalence classes are considered. BVA usually has a high payoff.

3.7.3.1.3) CAUSE - EFFECT GRAPHING

This is a technique that aids the systematic selection of a set of high yield test cases from the input and output specification of the program. The graphs are digital logic diagrams describing the connection between the causes (inputs) and effects (outputs). It reduces the number of situations that have to be tested. Causes are assigned numbers, as are effects, and then related by drawing lines between the nodes of cause and effect. Conventional logic symbols are also used

eg  etc.

3.7.3.1.4) ERROR GUESSING

Educated error guessing can be very effective.

3.7.3.2) TEST TO CODE (MORE APPLICABLE TO LOWER LEVEL MODULES)

A subset of all the possible paths is chosen using the following criteria:

3.7.3.2.1) STATEMENT COVERAGE

This criteria requires every statement in the program to be executed at least once. This is a very weak criterion

3.7.3.2.2) DECISION COVERAGE

3.7.3.2.3) CONDITION COVERAGE

3.7.3.2.4) DECISION / CONDITION COVERAGE

3.7.3.2.5) MULTIPLE CONDITIONS COVERAGE

All possible combinations of condition outcomes in each decision should be invoked at least once.

3.7.4) TESTING STRATEGIES

Here we will cover more general testing, using drivers and stubs. A driver is a module which provides input to some lower level module which is being tested, and a stub is a lower level dummy module that is used so that the upper level module being tested will run normally. A stub will usually return some trivial message and possibly some precoded data which is required by the module under test.

3.7.4.1) NON-INCREMENTAL TESTING (BIG BANG TESTING)

All modules are tested thoroughly in isolation, then they are put together and tested simultaneously. This means that every module is thoroughly tested, but both drivers and stubs have to be written, there is no interface error isolation and all interface problems are found late. This is thus a poor method of testing

3.7.4.2) INCREMENTAL TESTING

3.7.4.2.1) TOP-DOWN TESTING

Proceed top-down, level by level, coding and testing each module in turn (any lower level modules are coded as stubs). This has the advantage that major flaws are detected early, as are interface errors

and error isolation is easy. The disadvantages are that stubs, which are often more complex than the first appear, have to be written, and it is difficult to test lower level modules thoroughly.

These disadvantages can be somewhat lessened in adding I/O and critical (error prone or complex) modules early.

3.7.4.2.2) BOTTOM-UP TESTING

Begin with the lowest level modules, and move upwards, coding & testing each module in turn. This has the advantage that each module is thoroughly tested and error isolation is good. However, drivers have to be written, and major flaws and errors may only be found late.

3.7.4.2.3) SANDWICH TESTING

This is a compromise between top-down and bottom-up testing. The top and bottom level are thoroughly tested and major faults are found early. Disadvantages are that both drivers and stubs must be written and the middle levels of modules may not be thoroughly tested.

3.9.5) TESTING GUIDELINES & PRINCIPLES

- a programming organization should not test its own programs
- thoroughly inspect the results of each test
- a necessary part of a test case is the definition of the expected output or result
- a programming should avoid testing his/her own program
- test cases should be written for invalid as well as valid and expected input conditions.
- examine the program for unwanted side effects
- avoid throw-away test cases
- do not plan a testing effort on the assumption that no errors will be found but on the contrary
- the probability of the existence of more errors in a section of code is a to the number of errors already found there
- testing is the process of executing a program with the intention of finding errors
- a good test case is one that has a high probability of detecting an as yet undiscovered error
- a successful test case is one which detects an as yet undiscovered error

4) PROGRAMMING TOOLS

4.1) COMPILERS

There should provide error diagnostic facilities and efficient machine code. The programming environment should provide 2 compatible compilers: a development compiler that provide good error diagnostic, and an optimising compiler that produces efficient code.

The development compile should provide a program listing with the associated line number. Errors should be reported in their position and possible reason given. The programmer should have control directives to specify the way printout should be produced (suppression of parts, pagination, etc). For block structured languages the lexical level should be indicated at the beginning and end of each block (in RHS of printout). Source text and compile messages should be easily distinguishable. The start and end of each procedure should be identified in RHS of printout.

4.2) CROSS REFERENCER

This should be incorporated into the compiling system to provide names, types and line nos of objects in the system for each procedure, the procedure parameters and local and global variables.

4.3) EDITORS

A suitable editor (line screen or context) should be provided.

4.4) CODE CONTROL SYSTEM

A library system allowing for easy filing, referencing and discriminating between various version of the system.

5) TESTING TOOLS

5.1) TEST DATA GENERATORS

These can automatically generate a large number of test cases. They are useful when the peformance of a system is to be tested, and when the syntax of the output from the system can be formally specified as it can be checked.

5.2) EXECUTION FLOW SUMMARISERS

These report how many times each statement has been performed.

5.3) FILE COMPARATORS

When large volumes of test output is involved it is very easy to make some error. Data comparators can be used to compare two files and report the difference between them. By preparing one file with the expected output, and sending the actual output to another, the differences can be found easily.

6) DEBUGGING TOOLS & METHODS

Debugging is the identification and correction of errors.

6.1) OUTPUT DEBUG PRINT STATEMENTS

The programmer inserts these at various points in the program. They should be easily identifiable for later editing out. If a conditional compile is available, they can be excluded from the production program by a compile option. Certain specially designated output statements are conditionally executed depending on the setting of some global switch by the user.

6.2) CORE DUMPS

These are very crude.

6.3) SYMBOLIC DUMP

This is a list of all variable names and values either at the end of the program or on execution of

a "peep" command.

6.4) PROGRAM TRACE PACKAGE

A variety of traces are possible:

- trace change of all variable values
- trace change of certain specified values
- trace out entire program or over specified regions with tron & troff commands
- trace entry / exit of procedures
- give information on trend selected.

Trace packages can be very expensive in terms of extra memory space required, longer execution time of program, and the potentially unmanageable quantity of output produced.

6.5) STATIC PROGRAM ANALYSERS

In certain older languages an analyser performs a range of checks not performed by the compiler such as reporting on variable when defined and matching parameter types, etc.

6.6) INTERACTIVE DEBUGGING ENVIRONMENTS

The user program is run and a history file created of all program state changes. The user interacts with this history file to wind control

flow or data flow in the program as each statement executes. Such systems are potentially very useful but require considerable resources and are not widely available.

7) ORGANISATION OF PROGRAMMING TEAMS

7.1) WEINBERG'S APPROACH (1971)

Weinberg's hypothesis is that programming is a social activity, not an individual activity (if not it should be). He distinguished three types of personality traits:

- compliant: 'likes to work with people & be helpful'
- aggressive: 'likes to earn more money & prestige'
- detached: 'wants to be left alone to be creative'

He claimed that everyone is a combination of the 3 traits, but that programmers are often highly detached, while being highly attached to their programs, making it difficult to debug efficiently.

Weinberg's solution was egoless programming in democratic teams. The social environment and programmers values must be restructured, and programmers must be encouraged to:

- gain a group identity
- encourage fellow team members to find errors
- errors must not be considered a personal insult
- asking advice from another programmer must be seen as a compliment to the advice given, not an insult to the seeker

The advantages of Weinberg system are:

- positive attitude to finding bugs
- the more errors found the happier you should be
- the positive reinforcement leads to rapid error detection
- programs that are easy to debug are often easy to modify later on.

The disadvantages are:

- management has difficulty - who does it reward / fire
- programmers have difficulty adapting
- institutions teaching programming discourage a team approach.

Thus egoless programming is difficult to introduce, but once it occurs it should be encouraged as it does work.

7.2) CHIEF PROGRAMMER TEAM

This helps to reduce the number of communication channels in a project. The team consists of

- chief programmer
- back-up programmer
- programming secretary
- 1 - 3 programmers
- other specialists as required

The chief programmer is a combined manager and highly skilled programmer who

- designs the program
- allocates coding among the team members
- codes central and critical portions
- handles all interface issues
- reviews the work of all the other team members

The chief programmer is personally responsible for every line of code.

The back up programmer is necessary in case the chief programmer cannot complete the project. The B.U.P. must be totally competent to take over the project in every respect, and will also often do independent tasks such as test planning in order to free the C.P. to concentrate on the main design.

The programming secretary maintains the P.P.L. (program production library). This is a chronological record of programs produced, consisting of source code listings, J.C.L., the documentation and test data. The programmers hand the source code to the programming secretary, who is responsible for implementation, compiling, editing and testing. The P.S. must thus be highly skilled.

The programmers work under the direction of the C.P. or B.U.P. and only write code. They do not handle interface issues, the C.P. does.

8) DATA PROTECTION.

Data must be protected as far as possible against errors and tampering. There are several standard protection methods:

- validity checks on data
- check digits on data
- batching and batch control totals
- journal and audit trail
- restricting access to sensitive data

8.1) INTERNAL VALIDITY CHECKS

Type check: check that the data is of the correct type
Range check: check that the data is within some valid range
Other: Credit check - check customer credit
Reasonableness check - depends on situation

8.2) CHECK DIGITS

These are digits that are added to a number for the sole purpose of providing a check that the number is correct. The simplest is the binary parity bit. This allows detection of all single bit errors

The types of error which can occur are:

- transposition: two digits transposed (single - adjacent digit, double - non adjacent digit)

- transcription errors : replacement of digit by faulty digit
- random errors

Possible checking methods are :

1) Summing the digits modulo nine
   eg 7286 gives check digit = $(7+2+8+6) \bmod 9 = 5$
2) Make the number divisible by 11

   eg 7286 gives check digit = 4
   as $72864 = 6624 \times 11$

3) Weighting the digits according to position, sum the products and make divisible by 11

   eg 7286 gives check digit = 9

   as $(7\times5)+(2\times4)+(8\times3)+(6\times2) = 99 = 9 \times 11$

Unfortunately, any number that generates a check digit of 10 cannot be used (i.e. 9.1% of the integers)

The errors identified by these methods are :

Method	Single transposition	Single transcription	Random
1	No	All	A few
2	All	All	A few
3	All	All	Most

(see Fig 12.6)