Software Management Techniques

Janet Lind

There is currently much information available to document the fact that the cost of hardware is decreasing dramatically, but the cost of software continues to climb. When questioning the source of this problem, it is necessary to consider that many hardware functions are now being implemented in software or firmware. It is also true that computers are constantly being used in new applications, and computer users have increasingly sophisticated needs.

Today's software systems suffer from a variety of problems. Often they are delivered later than originally scheduled. The systems may cost more than the original projections. The software may not meet the user's requirements, or may be unreliable. When the need arises to correct or upgrade the system, the cost involved may be in excess of the cost of the original system.[18]

One of the most pressing problems in software project management is the lack of a well-developed structure for guiding the individual programmer. Instead of directing the programmer's activities, the manager can often only manage an idea until all parts of the project are completed. This problem arises from the fact that the only clearly defined point in the programmer's work is completion. More definition of the process is needed.[2]

There is no reason why software development should be exempt from the formats found in other engineering fields. Lab notebooks, design reviews, and failure and reliability analysis have proved their value.

The lack of a disciplined approach to software development may produce programs which are difficult to understand or maintain, affecting overall cost. Therefore it is important to develop a more rigorous framework to delineate the several steps in the programming process. Knowing the proper steps to follow will allow a programming team to develop more common objectives about the problem solution. This will improve the product and the group motivation by allowing the programmers to focus on more immediate goals.

Even though the approach being taken is to define a series of programming steps, it is always important to allow feedback to improve the product. A sequential description of program development steps will be defined here, but a problem found may cause a redefinition in preceeding steps to provide a more correct solution.[8]

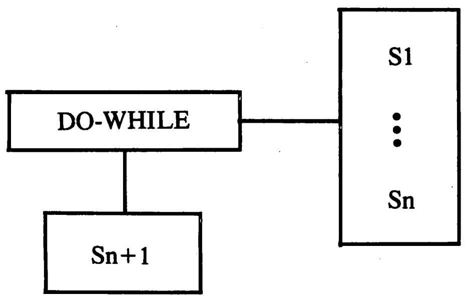
When first approaching a software project, it is necessary to perform a problem analysis. Here the inputs and outputs must be specified and the relationships between them must be described. A programming notebook should be kept to indicate how decisions were reached.[4]

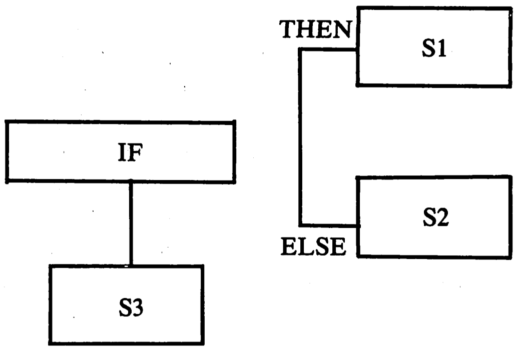
Part of the problem analysis includes decisions about the resources available. This includes both people and computer power. When considering the hardware used, it is no longer strictly correct to consider implementing everything possible in software. To increase productivity and to simplify code requirements, it may be worthwhile to purchase or develop hardware to meet the problem.

Another choice would be the use of multiple processors, which gives greater flexibility than implementing a function in hardware. This could also decrease the complexity of a given program, for it will no longer be responsible for as many portions of the function. Programs could also run concurrently, reducing timing constraints on a single system. This would make programming in higher level languages more attractive because the added processor capabilities offset the less efficient code produced. After completion of problem analysis, a walkthrough should be performed.

The solution design is driven by the I/O and their relationships defined in the problem analysis. Several different documentation techniques and evaluation criteria can be used in the structured design. Data flow diagrams can be used at the high level abstraction to model the flow of data through the system.[5]

Higher order software notation, or HOS, which was developed as part of the Apollo Program at Draper labs, defines a very useful flowcharting technique. Each control structure has a horizontal block showing the program flow in that structure. This type of flowchart does not show extra arrows, and allows easy identification of each possible branch. This notation also uses the same identation as should appear in the actual code.[3]





HOS Example

Some of the evaluation criteria used in structured design include decisions about the possible program development tools available. Certain programming languages may provide better support for the data structures to be used. They may also affect the amount of coupling required between modules. It is important to consider the capabilities of the computer system on which the program will be run, including memory management techniques and I/O capabilities.

When doing structured design, the design team is often tempted to perform just the top level abstraction as a team, designing the lower levels individually. There are some important reasons for doing a single integrated design of the entire application. First, subdivision of the design may result in excessive coupling of the major systems. The resulting packaging into programs from a subdivided design may be suboptimal. A complete overall structural design could produce more efficient and convenient packaging. Subdividing the design work will very often result in duplicate programming. It is particularly unfortunate when minor changes occur in a few structures, yielding a new system which could have shared entire subsystems and many levels of modules.[9]

Even though there are reasons for completing the entire structural design as a single unit, this is not always possible. In that case it would be best to produce a high level abstraction of the program flow and identify the more independent subsections. Those with few, uncomplicated interconnections could be treated independently. To avoid duplication of code, frequent mutual design walkthroughs and cross-checks should be performed.

Either while the structured design is being developed, or after its completion, the testing must be planned. It is necessary to design the test cases before the coding is begun. This allows peer review to verify that the designed code can be tested.

If the HOS flowchart notation is used, each program branch can be easily identified, and therefore tests can be designed to exercise each branch. If each program branch is numbered, a test matrix can be developed to indicate which tests execute which branches. The input and output to each test must also be specified.[4]

Both the structured code design and the test design should be carefully reviewed via structured walkthrough techniques. When considering walkthroughs, it is necessary to determine if it is more economical for an error to be found by the programmer, or by a group of 3 to 5 people. Part of the cost-benefit calculation is the turnaround time for repairing errors. Recent studies indicate that it is roughly ten times more expensive to fix a design error after it has been coded than to repair an error detected in design phase. It is also quite possible that when looking for errors, the programmer can repeat a logic error and never find the bug. Walkthroughs can help avoid this.[10]

Test Case	Input		Branch					Output		
	V1	V2	1	2	3	4	5	V1	V3	V4
1	0	0	X					0	0	0
2	0	1		X	X			0	0	2
	1	0		X		X		0	2	1
3	1	1	X		X		X	1	1	1

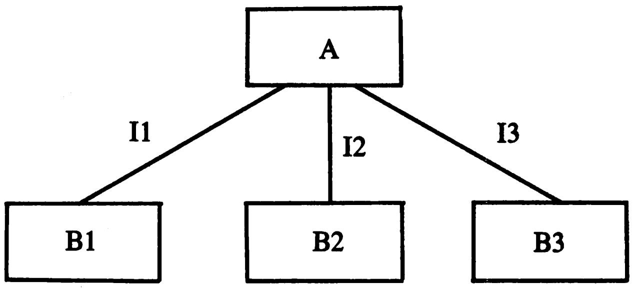
Test Matrix Example

There are other walkthrough benefits which must be weighed against the cost. The product quality is improved. The walkthrough participants are better trained in the product and are able to exchange important information. This exchanged information increases the probability that the product can be salvaged if a programmer leaves before completion. A walkthrough is also a good environment for feedback into other areas.

After the designs have been accepted, coding and debugging can begin. Here structured programming techniques should be both understood and applied. Using the HOS flowchart technique makes program flow and structuring obvious at coding time.

It is too simple to believe that code without "GO-TO" commands is always good. The language being used should be well comprehended by the programmers to ensure that the proper constructs are used. The code within each module must be structured. Concurrent documentation should also be kept.

With developing and testing code, it is also necessary to choose between a top-down or bottom-up approach to the overall structure. If hardware is being developed concurrent with software development, the lower level modules may be needed first to verify the hardware. In most other cases, a top-down approach can provide a more obvious visual presentation. This technique also allows modules to be tested together sooner. The interface between a node and its predecessor can be tested as soon as the lower level node is developed, allowing design or implementation errors to be detected and corrected earlier.[9]



Example

TOP-DOWN	BOTTOM-UP
1. Code and debug A	Code and debug B1
2. Code and debug B1	Code and debug B2
3. Test I1	Code and debug B3
4. Code and debug B2	Code and debug A
5. Test I2	Test I1
6. Code and debug B3	Test I2
7. Test I3	Test I3

A librarian function is helpful during coding and testing. The librarian can be an appropriately trained person, or an automated system. The librarian should maintain source programs and listings, as well as organizing all other technical information.

An automated system would avoid mixing media, which could be helpful in keeping a very accurate record of what changes are made. A record kept during edit phase could record what lines were modified and which variables were affected. A time stamp on this information could help other programmers know which version of code they were using. The knowledge that this system is being used will encourage a programmer to carefully analyze each change.

When the code can be tested, the test case matrix should be used to direct the tests applied. It may be useful to have the test run by the librarian. The test results should match those predicted, and a run log should be kept to document the test results. The purpose of the run should be stated, followed by an analysis of the run in terms of that purpose. This allows feedback for code correction and avoids haphazard modification. Any corrective actions which must be taken by the programmer should also be recorded.[5]

It may also be valuable to keep a time log to summarize the time needed for each step. This forces the programmer to review the actual effort expended in a task, and helps for making more realistic future estimates.

Throughout all activities, an independent auditing function can be performed. This will help detect errors unnoticed by the development team, and provides feedback.

The system described here is relatively involved and may be difficult to implement all at once. A pilot project could be chosen to use structured coding, structured design, and informal walkthroughs. As the process is implemented, it may be valuable to measure certain aspects such as the number of debugged lines of code produced per day and the number of bugs found after release. This can aid in future estimates. The amount of time spent in each walkthrough and the number of bugs found there should also be measured to help improve the techniques used.[6]

BIBLIOGRAPHY

[1]F. T. Baker, "Chief Programmer Team Management of Productin Programming," *IBM SYST. J.*, vol. 11, No. 1, 1972.

[2]F. T. Baker, "Structured Programming in a Production Environment," *IEEE Trans. Software Eng.*, pp. 241-252, June 1975.

[3]M. Hamilton and S. Zeldin, "Higher Order Software — A Methodology for Defining Software," *IEEE Trans. Software Eng.*, vol. se-2, pp. 9-32, Mar. 1976.

[4]P. Hsia and F. Petry, "A Framework for Discipline in Programming," *IEEE Trans. Software Eng.*, vol. se-6, no. 2, pp. 226-232, Mar. 1980.

[5]P. Hsia and F. Petry, "A Systematic Approach to Interactive Programming," *Computer*, pp. 27-34, June 1980.

[6]M. Page-Jones, *The Practical Guide to Structured Systems Design*, Yourdon Press, New York, N.Y., pp. 267-284, 1980.

[7]C. H. Reynolds, "What's Wrong with Computer Programming Management?," *On the Management of Computer Programming*, G. F. Weinwurm, Ed., Auerbach, Philadelphia, Pa., pp. 35-36, 1971.

[8]M. Walker, *Managing Software Reliability – the Paradigmatic Approach*, A. Salisbury, Ed., North Holland, New York, N.Y., pp. 32-41, 1981.

[9]E. Yourdon, *Managing the Structured Techniques*, Prentice-Hall, Inc., Englewood Cliffs, N.J., pp. 10-88, 1979.

[10]E. Yourdon, *Structured Walkthroughs*, Prentice-Hall, Inc., Englewood Cliffs, N.J., pp. 87-100, 1979.