UNIT - I

Introduction to Software Engineering: Definitions – Size Factors – Quality and Productivity Factors – Managerial Issues – Planning a software project: Defining the problem – Developing a Solution Strategy – Planning the Development Process – Planning an Organization structure – Other Planning Activities.

Introduction to Software Engineering:

Software engineering (sometimes abbreviated to SE) is a profession dedicated to designing, implementing, and modifying software so that it is of higher quality, more affordable, maintainable, and faster to build. The term software engineering first appeared in the 1968

NATO Software Engineering Conference, and was meant to provoke thought regarding the perceived "software crisis" at the time. Since the field is still relatively young compared to its sister fields of engineering, there is still much debate around what software engineering actually is, and if it conforms to the classical definition of engineering. Some people argue that development of computer software is more art than science, and that attempting to impose engineering disciplines over a type of art is an exercise in futility because what represents good practice in the creation of software is not even defined. Others, such as Steve McConnell, argue that engineering's blend of art and science to achieve practical ends provides a useful model for software development. The IEEE Computer Society's Software Engineering Body of Knowledge defines "software engineering" as the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches; that is, the application of engineering to software.

Software development, a much used and more generic term, does not necessarily subsume the engineering paradigm. Although it is questionable what impact it has had on actual software development over the last more than 40 years, the field's future looks bright according to Money Magazine and Salary.com, who rated "software engineering" as the best job in the United States in 2006.

When the first modern digital computers appeared in the early 1940s, the instructions to make them operate were wired into the machine. Practitioners quickly realized that this design was not flexible and came up with the "stored program architecture" or von Neumann architecture. Thus the first division between "hardware" and "software" began with abstraction being used to deal with the complexity of computing.

Definitions

Software engineering is the technological and managerial discipline concerned with systematic production of software products developed and modified on time and within cost estimates.

Software

(1) Instructions (computer programs) that when executed provide desired function and performance,

(2) Data structures that enable the programs to adequately manipulate information,

(3) Documents that describe the operation and use of the programs. There is no question that other, more complete definitions could be offered. But we need more than a formal definition.

Software Characteristics

To gain an understanding of software (and ultimately an understanding of software engineering), it is important to examine the characteristics of software that make it different from other things that human beings build. When hardware is built, the human creative process (analysis, design, construction, testing) is ultimately translated into a physical form. If we build a new computer, our initial sketches, formal design drawings, and bread boarded prototype evolve into a physical product (chips, circuit boards, power supplies, etc.).

Software is a logical rather than a physical system element. Therefore, software

Has characteristics that are considerably different than those of hardware:

1. Quality
2. Usefulness
3. Reliability
4. Easy to understand
5. Less memory and execution time
6. Cost effective

What are the attributes of good software?

The software should deliver the required functionality and performance to the user and

Should be maintainable, dependable and usable

l Maintainability

• Software must evolve to meet changing needs

l Dependability

• Software must be trustworthy

l Efficiency

• Software should not make wasteful use of system resources

l Usability

• Software must be usable by the users for which it was designed

Systems Engineering

Designing, implementing, deploying and operating systems which include hardware, software and people.

Size Factors

1. **Total effort devoted to software**
   Now a day, people expect high quality software. So human effort spent increases which results to more maintenance effort and cost. Software engineering helps to make software cheap and easy to maintain.

2. **Distribution of effort**
   Effort is distributed among different phases like analysis, design, implementation, testing, adaption, and enhancement. Even after delivery of a software product, effort is spent on maintaining it. Normally, a software life is five times the time spent of development. i.e. ., if 2 years is spent for developing software then its life is 10 years.

Maintenance also involves other activities like enhancement the product, adaptation to new changes and correction of bugs in current release.

For increased quality and productivity, stress is on

- Good documentation
- Standards tools and technology

Project size categories:

Projects are named after the size of project. They are

Trivial project:

It is done by one programmer for 1-4 weeks. Size is about 500 source lines. This product is used by him. Generally no planning is done. It can be improved by testing.

Small project:

Disadvantages:

Other programmers cannot use as there is no documentation done, no standard notations etc.It is done by one programmer for 1-6 months.

Size 1000-2000 source lines. It is generally a simple application. It can be implemented by conducting discussion among programmers or programmer and customer. It can be improved by using standard notations.

Medium Project:

It is done by 2-5 programmers for 1-2 years. Size of project is 10k-50k source lines. This project is done for a customer. It is implemented by having planning, documentation, and reviews between projects. It can be improved by implementing software principles.

Large Project:

It is done by5-20 programmers for 2-3 years. It is done in groups. Planning is really hard and is done both by programmers and customers. It is improved by implementing systematic process, standard documentation and conducting reviews.

Ex. Large compilers, real time control system, database package. Problem is in planning and communication between group and customers etc.

Very Large Project:

It is done by 100-1000 programmers for 4-5 years. Source code is 1 million lines. It involves several subsystems. Each subsystem does its own task also communicate with the other.

Extremely Large Project:

It is done by 2000 to 5000 programmers for 10 years. It has 1-10 million source lines. It involves interaction of very large subsystems. The project must be highly accurate

Type	Programmer	Time	Size	Planning	Improvement	Example
Trivial	1	1-4 weeks	500	-	By testing	Programmer Does for himself
Small	1	1-6 month	1k-2k	Discussion among co-programmers with customer	By using standard notion	Commercial applications
Medium	2-5	1-2 years	10k-50k	Planning reviews, documentation	Implementing software principles	Small MIS
Large	5-20	2-5 years	50k-100k	Between programmer and customer	Systematic process review	Large compilers
Very Large	100-1000	4-5 years	Million	By different subsystems	Effective comm among several subsystems	IBM OS/360
Extremely large	2k-5k	5-10 years	Up to 10 million	By different large subsystems	Effective comm. And strive for accuracy right from beginning	Air traffic control

Programmer’s time

Programmers do not spend 100% of their time in coding. They have different functions like Design, Planning, Documentation, Meeting etc.

Quality and Productivity Factors

Following are the factors that affect the quality and productivity

1) Individual ability
2) Team communication
3) Product complexity
4) Appropriate notations
5) Systematic approaches
6) Controlling changes
7) Level of technology
8) Required reliability
9) Available time
10) Problem understanding
11) Stability of requirements
12) Required skills
13) Facilities and resources
14) Adequacy of training
15) Management skills
16) Appropriate goals
17) Rising expectations

1) Individual ability

Production and maintenance of software is a labor intensive activity. So these factors are direct functions of individual ability and effort.

a) The general competence of the individual and

b) Familiarity with the particular application area.

2) Team communication

The new approach towards software development requires many individuals to be involved. Many recent innovations in software engineering, like design reviews, structured walkthroughs and code reading exercises improve communication between the programmers.

3) Product complexity

There are three generally acknowledged levels of product complexity

a) Application programs,

b) Utility programs and

c) System level programs.

Application programs have the highest productivity and Systems programs have the lowest productivity, measured in terms of LOC per programmer day.

4) Appropriate notations

The representation schemes are of fundamental importance. Good and standardized notations can clarify the relationships and interactions, of any specific interest.

5) Systematic approach

Following certain standard systematic approach in procedures and techniques makes the software development process very professional and contribute in improving the quality of the software product.

6) Controlling change

- The factors that contribute the factor of change in software are
  a) Design deficiencies in hardware,
  b) Change in requirements due to poor understanding of problem,

c) External economic and political factors etc.

To minimize the change the following suggestions can be utilized

a) Use of appropriate notations and standard techniques,

b) Scope of the mechanism to change ie flexibility provided to the software,

c) During planning process the strategies for controlled changes should be included, etc.

7) Level of technology

As the level of technology is changing rapidly hence modern programming languages should be used during development. Modern programming languages provide

a) Improved facilities for data definition & data usage,

b) User-defined exception handling,

c) Concurrent programming etc.

Also varied tools are provided such as assemblers and primitive debugging facilities integrated under a single fully integrated development environment.

8) Level of reliability

High reliability can be achieved by taking great care during –

a) Analysis,

b) Design,

c) System testing and

d) Maintenance.

Both human and machine resources are required to increase reliability. Too much reliability causes decreased productivity when measured in LOC produced Per Programmer Month.

9) Problem understanding

Failure to understand the problem can result in an unsuccessful project. There are some suggestions for avoiding these problems –

a) Careful planning,

b) Customer interviews,

c) Task observation,

d) Prototyping,

e) Precise product specifications etc.

10) Available time

Programmer productivity is sensitive to the, calendar time available for project completion. It is observed that the development time can not be compressed below 75 percent of the nominal development time. Extending a project beyond some nominal duration increases the total effort required.

11) Required Skills

The practice of SE required variety of skills. The requirement definition and design activity are conceptual in nature hence require good creative problem solving skills. Debugging requires deductive or detective kind of skills. Preparations of external documents require good writing and expressive skills. Software Engineers should have good social skills to interact properly with the managers, customers and other engineers.

12) Facilities and resources

Through various studies it had been found that work-related factors such as –

a) Good machine,

b) Quite place of work,

c) Plenty of access to the machine

d) Creative challenges,

e) Variety of tasks,

f) Opportunities for professional advancement etc, creates more satisfaction among the programmers rather than status factors.

13) Adequacy of training

It is seen that the fresher's from college do not have the following skills, which are very necessary for becoming a successful software engineer –

a) Express oneself clearly in English

b) Develop & validate software requirements and design specifications.

c) Work with in application area.

d) Perform software maintenance.

e) Perform economic analysis.

f) Work with project management techniques.

g) Work in groups.

Hence adequate of training is required for entry-level programmers.

14) Management Skills

As the concept of SE is new hence the managers who work in traditional styles are unable to properly manage the SE activities. Also it is seen that during giving promotions if the competency of the candidate is only measured in the technical point of view then there may arise the same kind of problem. So the management skills in a candidate should also include technical and managerial competency.

15) Appropriate goals

Setting of appropriate goals is a major contributing factor in successful execution of a project. Generally the most common goals include

a) Generality,

b) Efficiency and

c) Reliability.

High productivity and quality factors can be achieved by adhering to the goals and requirements established for the software during project planning.

16) Rising expectations

Progress is constantly being made in the development of tools and techniques to improve software quality and programmer productivity, equally diversity, size and complexity of software applications are growing at a fast rate but also the expectations and increasing demands. Hence it's necessary to adhere to the latest that is happening.

17) Stability of requirements

The requirements of a software project includes

a) Manpower,

b) Software and

c) Hardware.

The requirements definition is done during problem analysis phase where the amount and type of resources are predicted. In this point if the analysis and prediction is not based on facts, expert guidance and experience from previous projects, then the requirements would change throughout the development process and may extend to maintenance phase. Un-stability of requirements

increases the cost, degrades overall quality and reduces the productivity. Hence great care should be taken during the initial phase of problem identification & problem analysis.

Other factors

There are several other factors that influence the productivity –

a) Familiarity to the programming environment,

b) Amount and type of access to the computing system,

c) Stability of the computing system,

d) Memory & timing constrains,

e) Experience in that particular area of interest,

f) Data-base size etc.

Managerial Issues

Managers provide resources for technical activities. They are responsible for

1. Cost estimation of project
2. Induction of man power
3. Training employees
4. Completion of project with a time period

Ideal or expected activities of Managers in a firm –

1. Managers should control the resources and the environment in which technical activities occurs.
2. They also have ultimate responsibility for ensuring that software products are delivered on time and within cost estimates.
3. They have the responsibility of ensuring that the software products exhibit the functional and quality attributes desired by the customer.

4. The managers are responsible for project management which includes methods for organizing and monitoring the project progress.
5. Organizing and monitoring a project includes –
   a) Cost estimation,

   b) Resource allocation policies,

   d) Budgetary control,

   e) Setting project milestones,

   f) Making schedule adjustments,

   g) Establishing quality assurance procedures,

   h) Establishing effective communication between project members,

   i) Customer communications,

   j) Development of contractual agreements with customers taking in to view legal obligations and responsibilities.

Communication with customers and their people (programmer)

They face some problem while they manage the product

1. Planning is not given due importance

2. Accountability is poor

3. They cannot estimate the resources accurately

4. Project managers do not have inputs like techniques, progress readily available so that they can decide.

5. Standards and techniques is only on paper. It is very hard to implement so software product becomes unreliable and hard to maintain.

To avoid all these

1. Select managers on their ability to manage
2. They should be given training at regular intervals and teach them new techniques followed.
3. A firm must be very strict on standards, documentation etc.
4. Take previous project as input and analyse that project. These help managers to be accurate for next time.
5. Allow for contingencies so that the project can adjust with different situation like short of manpower, lack of resources etc.

Planning a software project:

Defining the problem

This is the first step in planning of the software project is prepare, in the customer's terminology, a concise statement of the problem to be solved and the constraints that exist for its solution. The definitive problem statement should include a description of the present situation and the goal to be achieved by the new system. Now we will go through the steps that are followed when we the planning of software project is done.

The definition of the problem must be in the form of payroll problem, inventory problem, etc and not as DMA channels and sorting algorithms. They must be in simple customer terminology. Through understanding of the problem domain and environment is a necessity. Planner must be highly skilled in order to understand the various complexities.

Defining the problem

1. Developing a statement that defines the problem a specifies the goals to be achieved.
2. Justify a computerized solution strategy for the problem.
3. Identification and gathering of baseline data.
4. Determining the system-level goals and requirements.
5. Establishing a acceptance criteria for the system.
6. Developing solution strategies.
7. Outlining solution strategies.
8. Checking for feasibility of each strategy
9. Recommend a solution strategy and reject the other with reasoning.
10. Developing a list of priorities.
11. Planning the development process.
12. Define a life –cycle model and an organizational structure for the project.
13. Plan the configuration management tools, quality assurance and validation activities.
14. Determine phase-dependent tools, techniques, and notations to be used.
15. Establish preliminary cost estimates

Developing a Solution Strategy

`It is not a detailed solution plan. Here, all possible solutions including external factors are explored. Planners conduct feasibility study and cost estimations on different possible solutions. Solutions strategy should be generated without regard for adapted feasibility. Planners choose one strategy and adapt it. Generally, solution strategy derived is a combination of different strategy. Solution strategy is based on solution constraints, available time, resources, technology, and trade off decisions.

Different techniques adapted to find a solution strategy are

Case Study:

Different case are discussed during development of solution strategy.

Worst case study

Worst cases are discussed during development to avoid new surprises during project.

A simulation is done in order to see pros and cons of solution

Prototype

A prototype is constructed for solution. It incorporated some-parts of original model. It helps to justify our strategy.

A solution strategy Simulation contains priority list of features (i.e)

Planning the Development Process

The linear sequential model is the oldest and the most widely used paradigm for software engineering. However, criticism of the paradigm has caused even active supporters to question its efficacy. Among the problems that are sometimes encountered when the linear sequential model is applied are:

1. Real projects rarely follow the sequential flow that the model proposes.

Although the linear model can accommodate iteration, it does so indirectly.

As a result, changes can cause confusion as the project team proceeds.

2. It is often difficult for the customer to state all requirements explicitly. The linear sequential model requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.

3. The customer must have patience. A working version of the program(s) will not be available until late in the project time-span. A major blunder, if undetected until the working program is reviewed, can be disastrous. In an interesting analysis of actual projects Bradac [BRA94], found that the linear nature of the classic life cycle leads to "blocking states" in which some project team members must wait for other members of the team to complete dependent tasks. In fact, the time spent waiting can exceed the time spent on productive work! The blocking state tends to be more prevalent at the beginning and end of a linear sequential process. Each of these problems is real. However, the classic life cycle paradigm has a definite and important place in software engineering work. It provides a template into which methods for analysis, design, coding, testing, and support can be placed. The classic life cycle remains a widely used procedural model for software engineering. While it does have weaknesses, it is significantly better than a haphazard approach to software development.

Often, a customer defines a set of general objectives for software but does not identify detailed input, processing, or output requirements. In other cases, the developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the form that human/machine interaction should take. In these, and many other situations, a prototyping paradigm may offer the best approach.

Waterfall Software Development Life Cycle Model

The simplest software development life cycle model is the waterfall model, which states that the phases are organized in a linear order. A project begins with feasibility analysis. On the successful demonstration of the feasibility analysis, the requirements analysis and project planning begins.

The design starts after the requirements analysis is done. And coding begins after the design is done. Once the programming is completed, the code is integrated and testing is done. On succeful completion of

testing, the system is installed. After this the regular operation and maintenance of the system takes place. The following figure demonstrates the steps involved in waterfall life cycle model.

Requirement Analysis Palnning

System Design and Specification

Coding & Verification

Testing & Integration

Maintenance

The Waterfall Software Life Cycle Model

With the waterfall model, the activities performed in a software development project are requirements analysis, project planning, system design, detailed design, coding and unit testing, system integration and testing. Linear ordering of activities has some important consequences. First, to clearly identify the end of a phase and beginning of the others. Some certification mechanism has to be employed at the end of each phase. This is usually done by some verification and validation. Validation means confirming the output of a phase is consistent with its input (which is the output of the previous phase) and that the output of the phase is consistent with overall requirements of the system. The consequences of the need of certification is that each phase must have some defined output that can be evaluated and certified. Therefore, when the activities of a phase are completed, there should be an output product of that phase and the goal of a phase is to produce this product. The outputs of the earlier phases are often called intermediate products or design document. For the coding phase, the output is the code. From this point of view, the output of a software project is to justify the final

program along with the use of documentation with the requirements document, design document, project plan, test plan and test results.

Another implication of the linear ordering of phases is that after each phase is completed and its outputs are certified, these outputs become the inputs to the next phase and should not be changed or modified. However, changing requirements cannot be avoided and must be faced. Since changes performed in the output of one phase affect the later phases, that might have been performed. These changes have to made in a controlled manner after evaluating the effect of each change on the project. This brings us to the need for configuration control or configuration management.

The certified output of a phase that is released for the best phase is called baseline. The configuration management ensures that any changes to a baseline are made after careful review, keeping in mind the interests of all parties that are affected by it. There are two basic assumptions for justifying the linear ordering of phase in the manner proposed by the waterfall model.

For a successful project resulting in a successful product, all phases listed in the waterfall model must be performed anyway.

Any different ordering of the phases will result in a less successful software product.

Project Output in a Waterfall Model

As we have seen, the output of a project employing the waterfall model is not just the final program along with documentation to use it. There are a number of intermediate outputs, which must be produced in order to produce a successful product.

The set of documents that forms the minimum that should be produced in each project are:

- Requirement document
- Project plan
- System design document
- Detailed design document
- Test plan and test report
- Final code
- Software manuals (user manual, installation manual etc.)
- Review reports

Except for the last one, these are all the outputs of the phases. In order to certify an output product of a phase before the next phase begins, reviews are often held. Reviews are necessary especially for the requirements and design phases, since other certification means are frequently not available. Reviews are formal meeting to uncover deficiencies in a product. The review reports are the outcome of these reviews.

Prototype Life Cycle Model

The prototyping paradigm (fig) begins with requirements gathering. Developer and customer meet and define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory. A "quick design" then occurs. The quick design focuses on a representation of those aspects of the software that will be visible to the customer/user (e.g., input approaches and output formats). The quick design leads to the construction of a prototype. The prototype is evaluated by the customer/user and used to refine requirements for the software to be developed. Iteration occurs as the prototype is tuned to satisfy the needs of the customer, while at the same time enabling the developer to better understand what needs to be done. Ideally, the prototype serves as a mechanism for identifying software requirements. If a working prototype is built, the developer attempts to use existing program fragments or applies tools (e.g., report generators, window managers) that enable working programs to be generated quickly. But what do we do with the prototype when it has served the purpose just described? Brooks provides an answer: In most projects, the first system built is barely usable. It may be too slow, too big, awkward in

use or all three. There is no alternative but to start again, smarting but smarter, and build a redesigned version in which these problems are solved . . . When a new system concept

or new technology is used, one has to build a system to throw away, for even the best planning

is not so omniscient as to get it right the first time. The management question, therefore,

is not whether to build a pilot system and throw it away. You will do that. The only

question is whether to plan in advance to build a throwaway, or to promise to deliver the

throwaway to customers . . . The prototype can serve as "the first system." The one that Brooks recommends we throw away. But this may be an idealized view. It is true that both customers and developers like the prototyping paradigm. Users get a feel for the actual system and

developers get to build something immediately. Yet, prototyping can also be problematic

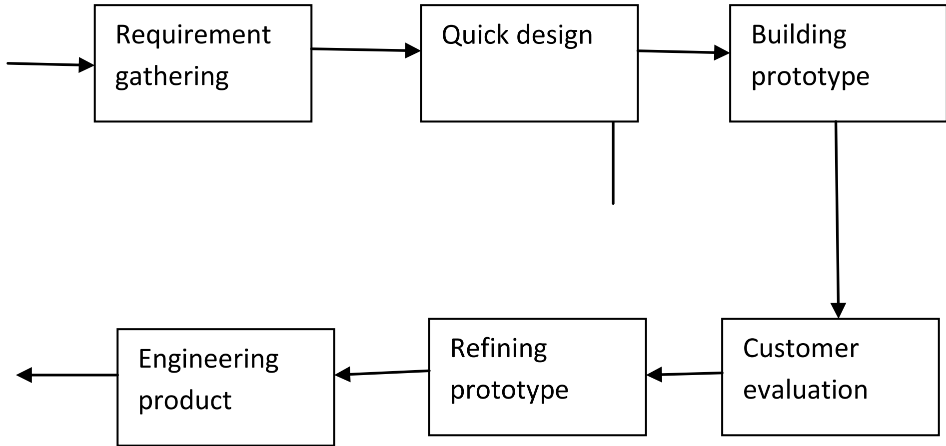
for the following reasons:

1. The customer sees what appears to be a working version of the software, unaware that the prototype is held together “with chewing gum and baling wire,” unaware that in the rush to get it working no one has considered overall software quality or long-term maintainability. When informed that the product must be rebuilt so that high levels of quality can be maintained, the customer cries foul and demands that "a few fixes" be applied to make the prototype a working product. Too often, software development management relents.

2. The developer often makes implementation compromises in order to get a prototype working quickly. An inappropriate operating system or programming language may be used simply because it is

available and known; an inefficient algorithm may be implemented simply to demonstrate capability. After a time, the developer may become familiar with these choices and forget all the reasons why they were inappropriate. The less-than-ideal choice

has now become an integral part of the system. Although problems can occur, prototyping can be an effective paradigm for software engineering. The key is to define the rules of the game at the beginning; that is, the customer and developer must both agree that the prototype is built to serve as a mechanism for defining requirements. It is then discarded (at least in part) and the actual software is engineered with an eye toward quality and maintainability.



Prototype life cycle model

Planning an Organization structure

A proper organization structure is most important as it helps in decision-making and the tackling of any problem in the best possible way. Now the planning of the organizational structure can be explained in the following ways.

Project structure:

It is classified into 3 types

1. Project format
2. Functional format
3. Matrix format

Project Format:

This is a format where a team of programmers is grouped and they look in to all the steps in the development of a project. This may range from 1 to 3 years. They will be shifted to new projects only on the completion of the project. Some of the team members stay back to look into the maintenance of the finished project.

.

Functional Format:

In the functional approach to organization, a different team of programmers performs each phase of the project, and the work products pass from one team to the other. A typical variation on the functional format involves three teams: an analysis team, a design and implementation team, and a testing and maintenance team. Team members are periodically transferred from function to avoid

overspecialization. This requires more communication between teams but personnel become specialists in particular roles and results in more attention to proper documentation.

Matrix Format:

In these organizations each of the function has its own management team and a group of specialist personnel who are concerned only with that function. In matrix organizations, everyone has at lease two bosses, and the need to resolve ambiguity and conflict is the price paid for project accountability.

Programming team structure:

It is classified into 3 types

1. Democratic teams
2. Chief programmer teams
3. Hierarchical team structure

Democratic teams:

Fig. 1. Egoless Team Structure. Authority is dispersed and communication linkages decentralized.

These are termed as “ego-less team” which has one team leader who assumes the position of the first among the equals. The advantages of this structure are each member contributes to decision-making; knowledge exchange between team members is smooth and increased job satisfaction. The disadvantages are communication overheads which lead to delayed decisions.

Chief programmer teams:

They are highly structured and efficient allocations done by the chief programmer. There is the presence of the librarian who maintains the documentation.

The advantages are centralized decision making and reduced communication paths. This can also result low morale for subordinate programmers.

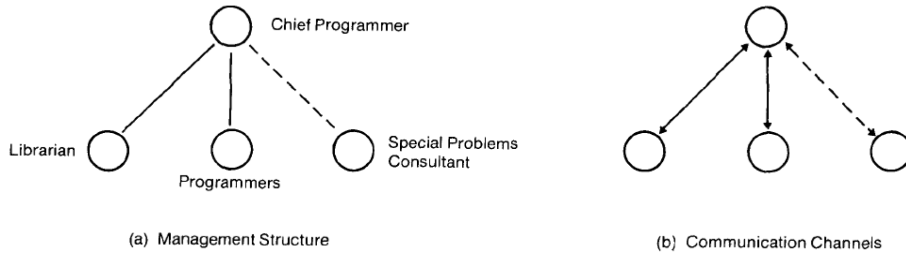


Fig. 2. Chief Programmer Team Structure. Authority is vested in the chief programmer and communication is centralized to this individual.

Hierarchical team structure

This is a middle organizational structure between the previous two. This limits the number of communication paths. This will be a very effective structure when a large project is being handled and many subsystems are working over it. The major disadvantage is that technically competent programmers may be promoted into management positions. This may have two different effects based on the personality traits of the respective person being promoted.

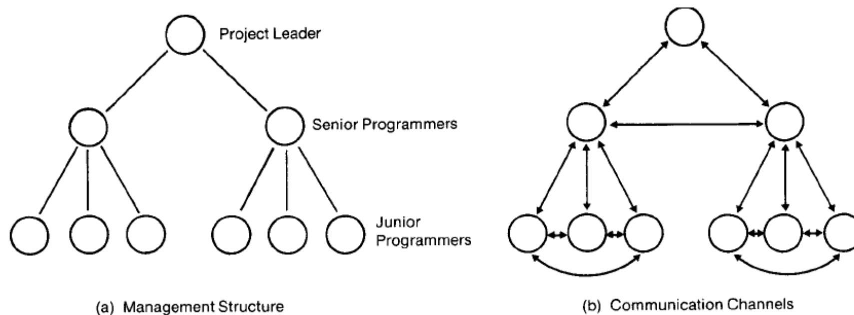


Fig. 3. Controlled Decentralized Team Structure. Authority is vested in the project leader and senior programmers, but communication at each level of the management hierarchy is decentralized.

Other Planning Activities

1. **Configuration management and quality assurance**

Configuration management is concerned with controlling changes in work productivity, status of project, project standards, audits acceptance testing. Planning in configuration management specifies the procedure to be used, specifies tools and budget of project. During design adherence to standards, tools, quality is done. During implementation design specifications, source code, configuration management is done.

Configuration management (CM) is the detailed recording and updating of information that describes an enterprise's hardware and software. Such information typically includes the versions and updates that have been applied to installed software packages and the locations and network addresses of hardware devices. Special configuration management software is available. When a system needs hardware or software upgrade, a computer technician can accesses the configuration management program and database to see what is currently installed. The technician can then make a more informed decision about the upgrade needed.

An advantage of a configuration management application is that the entire collection of systems can be reviewed to make sure any changes made to one system do not adversely affect any of the other systems. During testing phase acceptance testing and preparation of results are done.

2. **Quality Assurance**

Quality assurance, or QA for short, refers to a program for the systematic monitoring and evaluation of the various aspects of a project, service, or facility to ensure that standards of quality are being met.

It is important to realize also that quality is determined by the program sponsor. QA cannot absolutely guarantee the production of quality products, unfortunately, but makes this more likely.

Two key principles characterize QA: "fit for purpose" (the product should be suitable for the intended purpose) and "right first time" (mistakes should be eliminated). QA includes regulation of the quality of raw materials, assemblies, products and components; services related to production; and management, production and inspection processes.

It is important to realize also that quality is determined by the intended users, clients or customers, not by society in general: it is not the same as 'expensive' or 'high quality'. Even goods with low prices can be considered quality items if they meet a market need. QA is more than just testing the quality of aspects of a product, service or facility, it analyzes the quality to make sure it conforms to specific requirements and comply with established plans.

3. **Planning for independent verification and validation.**

Generally, a separate team is for verification and validation. Verification deals with whether the work is completed and it is consistent.

Validation deals with planning and, execution of test cases and it are executed by independent organization.

4. **Planning phase dependent tools** and technology Automated tools, notations, modern techniques, develop SRS (Software Requirement Specification), architectural and detailed design are planned well.

5. **Other factors:** Estimates like preliminary cost estimates, estimation of staffing levels are necessary for planning.

Cost estimation

UNIT - II

Software Cost Estimation: Software cost factors – Software Cost Estimation Techniques – Staffing-level Estimation – Estimating Software Maintenance Costs – The Software Requirements Specification – Formal Specification Techniques – Languages and Processors for Requirements Specification.

Software Cost Estimation:
In the early days of computing, software costs constituted a small percentage of the overall computer-based system cost. An order of magnitude error in estimates of software cost had relatively little impact. Today, software is the most expensive element of virtually all computer-based systems. For complex, custom systems, a large cost estimation error can make the difference between profit and loss. Cost overrun can be disastrous for the developer.

Software cost and effort estimation will never be an exact science. Too many variables human, technical, environmental, political—can affect the ultimate cost of software and effort applied to develop it. However, software project estimation can be transformed from a black art to a series of systematic steps that provide estimates with acceptable risk.
To achieve reliable cost and effort estimates, a number of options arise:

1. Delay estimation until late in the project (obviously, we can achieve 100% accurate estimates after the project are complete!).
2. Base estimates on similar projects that have already been completed.
3. Use relatively simple decomposition techniques to generate project cost and effort estimates.
4. Use one or more empirical models for software cost and effort estimation. Unfortunately, the first option, however attractive, is not practical. Cost estimates must be provided "up front." However, we should recognize that the longer we wait, the more we know, and the more we know, the less likely we are to make serious errors in our estimates.

The second option can work reasonably well, if the current project is quite similar to past efforts and other project influences (e.g., the customer, business conditions, the SEE, deadlines) are equivalent. Unfortunately, past experience has not always been a good indicator of future results.

The remaining options are viable approaches to software project estimation. Ideally, the techniques noted for each option should be applied in tandem; each used as a cross-check for the other. Decomposition techniques take a "divide and conquer" approach to software project estimation. By decomposing a project into major functions and related software engineering activities, cost and effort estimation can be performed in a stepwise fashion. Empirical estimation models can be used to complement decomposition techniques and offer a potentially valuable estimation approach in their own right. A model is based on experience (historical data) and takes the formed = f (vi)
Where d is one of a number of estimated values (e.g., effort, cost, project duration) and vi are selected independent parameters (e.g., estimated LOC or FP).

Automated estimation tools implement one or more decomposition techniques or empirical models. When combined with a graphical user interface, automated tools provide an attractive option for estimating. In such systems, the characteristics of the development organization (e.g., experience, environment) and the software to be developed are described. Cost and effort estimates are derived from these data. Each of the viable software cost estimation options is only as good as the historical data used to seed the estimate. If no historical data exist, costing rests on a very shaky foundation.

Background:

Software Cost Estimation is widely considered to be a weak link in software project management. It requires a significant amount of effort to perform it correctly. Errors in Software Cost Estimation can be attributed to a variety of factors.
The group of people responsible for creating a software cost estimate can vary with each organization. However the following is possible in most scenarios -

- People who are directly involved with the implementation are involved in the estimate.

- Project Manager is responsible for producing realistic cost estimates.
- Project Managers may perform this task on their own or consult with programmers responsible.
- Various studies indicate that if the programmers responsible for development are involved in the estimation it was more accurate. The programmers have more motivation to meet the targets if they were involved in the estimation process.

Impact of Under-estimating:

Under-Estimating a project can be vary damaging

- It leads to improper Project Planning - It can also result in under-staffing and may result in an over worked and burnt out team
- Above all the quality of deliverables may be directly affected due insufficient
..Testing and QA
- Missed Dead lines cause loss of Credibility and goodwill

The Estimation Process:

Generally the Software Cost estimation process comprises of 4 main steps:

1) Estimate the size of the development product. This comprises of various sub-steps or sub tasks. These tasks may have been done already during Requirement Analysis phase. If not then they should be done as a part of the estimation Process. Important thing is that they should be done to ensure the success of the Estimation Process and the Software Project as a whole.

a) Create a detailed Work Break Down Structure. This directly impacts the accuracy of the estimate. This is one of the most important steps. The Work Break down structure should include any and all tasks that are within the scope of the Project, which is being estimated. The most serious handicap is the inability to clearly visualize the steps involved in the Project. Executing a Software Project is not just coding.

b) The work Break down structure will include the size and complexity of each software module that can be expressed as number of Lines of Code, Function Points, or any other unit of measure

c) The Work Break down structure should include tasks other than coding such as Software Configuration Management, various levels and types of Testing, Documentation, Communication, User Interaction, Implementation, Knowledge Transition, Support tasks(if any) and so on

d) Clearly indicate or eliminate any gray areas (vague/unclear specifications etc.)

e) Also take into account the various Risk Factors and down times. There are many different Risk Factors involved – Technical aspects such as availability of the Environment, Server/Machine uptime, 3rd party Software Hardware failures or Human aspects – Employee Attrition, Sick time, etc. Some of them may seem to be 'overkill' but real world experience shows that these factors affect the time lines of a project. If ignored they may adversely impact the Project timelines and estimates.

2) Estimate the effort in person-hours.

The Result of various tasks involved in step 1 is an effort estimate in person hours. The effort of various Project tasks expressed in person-hours is also influenced by various factors such as:

a) Experience/Capability of the Team members
b) Technical resources
c) Familiarity with the Development Tools and Technology Platform

3) Estimate the schedule in calendar months

The Project Planners work closely with the Technical Leads, Project Manager and other stakeholders and create a Project schedule. Tight Schedules may impact the Cost needed to develop the Application.

4) Estimate the project cost in dollars (or other currency)

Based on the above information the project effort is expressed in dollars or any other currency.
Measuring the Size/Complexity of the Software Program:

This is one of the most elusive aspects in the Software Cost Estimation Process.
There are different methodologies for arriving at and expressing the size/complexity of the Software Program. Some of the popular ones are

1) Function Points
2) Lines of Code
3) Feature Points
4) Mk II function points
5) 3D Function Points
6) Benchmarking

We briefly explain each of the above methods in the next few pages

Function Points

The Function Point methodology was developed by Allan Albrecht at IBM. This methodology is based on the belief that the size of a software project can be estimated during the requirements analysis. It takes into account the inputs and outputs of the system. Five classes of items are counted:

1. External Inputs
2. External Outputs
3. Logical Internal Files
4. External Interface Files
5. External Inquiries

Lines of code:

Counting lines of code measures software from the developers' point of view The number of lines of code is the traditional way of measuring the application size. Many people consider this method as irrelevant now. There are technical problems with the lines of code measure. It is difficult to compare lines of code when a mix of technologies is used. There is no standard definition of what a line of code is. A Program may have blank lines, comments, data declarations, and multi-line statements.

Feature points Methodology:

It was developed by Software Productivity Research (SPR) in 1986. This technique takes into account the number of algorithms used in the application. It is compatible with the Function Points Methodology. The size calculated by the two methods for an ordinary transactional program would be the same. Feature Points Methodology is generally more useful for estimation in real-time process control, mathematical optimization and various embedded systems. The estimates are higher and considered more accurate in these cases.

Mk II function points Methodology:

This was developed Charles Symons in 1984 at Nolan, Norton & Co., part of KPMG Management Consulting. The Original Function Point approach suffers from the following weaknesses:

· It is often difficult to identify the components of an application. The original Function Point Methodology assigned weights to function point components based on "debate and trial." The original Function Point Methodology did not provide a means of accounting for internal complexity. 'Feature points' technique addresses these issues. When small systems are combined into larger applications. Function Points Methodology makes the total function point count less than the sum of the components.

MKII decomposes the application being counted into a collection of logical transactions. Each transaction consists of an input, a process and an output. For each transaction, Unadjusted Function Points (UFP) become a function of the number of input data element-types, entity-types referenced and output data element-types. The UFPs for the entire system are then summed. Mk II is widely used in the UK, India, Singapore, Hong Kong and Europe. Users include governmental organizations, finance, insurance, retail and manufacturing.

3D function points:

This methodology was developed by Boeing Company and published in 1992. The new technique was designed to address two classic

problems associated with the Albrecht approach (the original Functional Point Methodology)

a) The original Functional Point Methodology is not user friendly
b) It is inaccurate when measuring complex scientific and real-time systems.

The 3D function points takes into account the following Dimensions - data, function and control. The data dimension is similar to the original Function Point Methodology. The function dimension accounts for the transformations or algorithms. The control dimension accounts for transitions or changes in application state.

Benchmarking:

Over the years many Organizations with significant development experience and mature processes have collected metrics on the various software development projects. These include the time, effort required to develop applications on various platforms and in various Business Domains. Based on this data benchmarks are created.

Each new software module to be developed can be categorized using the

a) Number of inputs
b) Number of outputs
c) Number of transactions
d) Algorithms
e) Features of the module

Programmer's ability

This is measured on the parameter of programmer productivity which is achieved by relating the best and the worst performances for a sample of programmers. In large projects the differences in individual programmer ability will tend to average out, but on projects utilizing 5 or less programmers, the differences make significance.

Product complexity:

The categories of software products they are utility, application and system programs. These are measured by PM(Programmer – Months) and the time taken to develop, TDEV. Taking these into account parallel activity can be performed on system and utility programs.

Product Size:

A large software project is more costly than a small one, but where we have to concentrate is that as the size increases the effort increases exponentially and hence the cost.

Available time:

It indicates an extreme penalty for schedule compression and an extreme reward for expanding the project schedule.

System development times

	Analysis	Design	Coding	Testing	Documentation
Assembly code	3 weeks	5 weeks	8 weeks	10 weeks	2 weeks
High-level language	3 weeks	5 weeks	4 weeks	6 weeks	2 weeks

	Size	Effort	Productivity
Assembly code	5000 lines	28 weeks	714 lines/month
High-level language	1500 lines	20 weeks	300 lines/month

Required level of reliability:

Software reliability can be defined as the probability that a program will perform a required function under stated conditions for a stated period of time. Reliability can be expressed in terms of accuracy, robustness, completeness and consistency of the source code.

Level of technology:

The better level of technology the higher productivity so lower cost because the time taken to complete the project would be less and the lesser number of resources will be used.

Software cost components

- Hardware and software costs.
- Travel and training costs.
- Effort costs
    – The salaries of engineers involved in the project;
    – Social and insurance costs.
- Effort costs must take overheads into account
    – Costs of building, heating, lighting.
    – Costs of networking and communications.
    – Costs of support resources
    – Costs of shared facilities (e.g library, staff restaurant, etc.).

Costing and pricing

- There is not a simple relationship between the development cost and the price charged to the customer.
- Broader organisational, economic, political and business considerations influence the price charged.

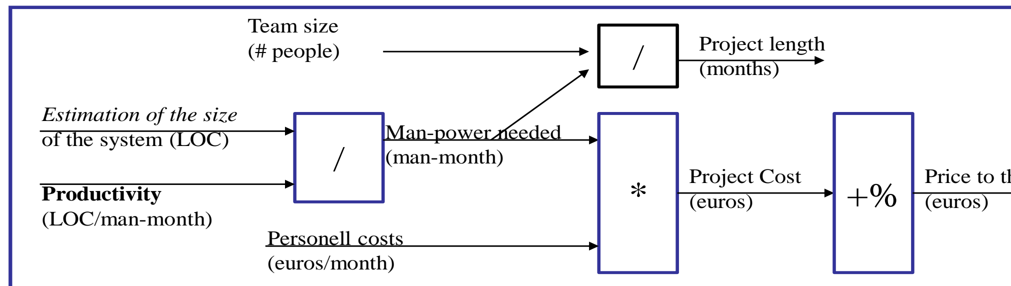
- Software Cost Estimation Techniques
  - Cost estimates can be made either **top-down or bottom-up.**
  - Top – down estimation first focuses on system level costs, such as the computing resources and personnel to develop the system, as well as the costs of configuration management, quality assurance, system integration, training, and publications. This is done by basically two approaches the top-down and the bottom-up approach.
  - The bottom-up cost estimation first estimates the cost to develop each module or subsystem. Those costs are combined to arrive at an overall estimate.
  - Top-down estimation has the advantage of focusing on system-level costs, but may overlook various technical factors in some modules to be developed.

Estimation Techniques

- **Pricing to Win:**
  - The software cost is estimated to be whatever the customer has available to spend on the project. The estimated effort depends on the customer's budget and not on the software functionality.
- **Parkinson's law**
  - Work expands to fill the time available. The cost is determined by available resources rather than by objective assessment. If the software has to be delivered in 12 months and 5 people are available, the effort required is estimated to be 60 person-months.
- **Analogy**
  - This technique is applicable when other projects in the same application domain have been completed. The cost of a new project is estimated by analogy with these completed projects. Myers (Myers 1989) gives a very clear description of this approach.

- **Expert Judgment**
  - Several experts on the proposed software development techniques and the application domain are consulted. They each estimate the project cost. These estimates are compared and discussed. The estimation process iterates until an agreed estimate is reached.
- **Algorithmic cost modelling**
  - A model based on historical cost information that relates some software metric (usually its size) to the project cost is used. An estimate is made of that metric and the model predicts the effort required.

A rough Model



Measurement problems

- Estimating the size of the measure (e.g. how many function points).
  Estimating the total number of programmer months that have elapsed.
- Estimating contractor productivity (e.g. documentation team) and incorporating this estimate in overall estimate.

- What's a line of code?
    - The measure was first proposed when programs were typed on cards with one line per card;
    - How does this correspond to statements as in Java which can span several lines or where there can be several statements on one line.
- What programs should be counted as part of the system?
- This model assumes that there is a linear relationship between system size and volume of documentation.
-

Delphi Cost Estimation:

This technique was developed at rand Corp. in 1948 to gain expert consensus without introducing the adverse side affects of group meetings.
This Delphi technique can be adapted to software cost estimation in the following manner.

1. A coordinator provides each estimate with the system definition document and a form for recording their estimate.
2. The estimators study and complete their estimation anonymously. They ask questions to the coordinator but do not discuss with one another.
3. The coordinator makes a summary and includes any unused rationales noted by the estimators.
4. Estimators complete another estimation, again anonymously, using the results of the previous estimates. The estimators whose estimates differ sharply from the group may be asked to justify their answer, anonymously.
5. The process is iterated for as many rounds as required. No group discussion is allowed during the entire process.

It is possible that after several rounds of estimates will not lead to a consensus estimate. In this case, the coordinator must discuss the issues involved with each estimator to determine the reasons for the differences.
The coordinator may have to gather additional information and present it to the estimators in order resolve the differences in viewpoint.

Summary

Delphi cost estimation approach tries to overcome some of the shortcomings of the expert judgment approach. Delphi estimation is carried out by a team composed of a group of experts and a coordinator. In this approach, the coordinator provides each estimator with a copy of the software requirements specification (SRS) document and a form for recording his cost estimate. Estimators complete their individual estimates anonymously and submit it to the coordinator. In their estimates, the estimators mention any unusual characteristic of the product which has influenced their estimation. The coordinator prepares and distributes

the summary of the responses of all the estimators, and includes any unusual rationale noted by any of the estimators. Based on this summary, the estimators re-estimate. This process is iterated for several rounds. However, no discussion among the estimators is allowed during the entire estimation process. The idea behind this is that if any discussion is allowed among the estimators, then many estimators may easily get influenced by the rationale of an estimator who may be more experienced or senior. After the completion of several iterations of estimations, the coordinator takes the responsibility of compiling the results and preparing the final estimate.

Expert Judgment:

The mostly used cost estimation technique is expert judgment, which is an inherently top-down estimation technique.
Expert judgment relies on the experience, background and business sense of one or more key people in the organization.
The biggest advantage of the judgment, namely, experience, can also

3.5.1 Expert Judgment Technique

Expert judgment is one of the most widely used estimation techniques. In this approach, an expert makes an educated guess of the problem size after analyzing the problem thoroughly. Usually, the expert estimates the cost of the different components (i.e. modules or subsystems) of the system and then combines them to arrive at the overall estimate. However, this technique is subject to human errors and individual bias. Also, it is possible that the expert may overlook some factors inadvertently. Further, an expert making an estimate may not have experience and knowledge of all aspects of a project. For example, he may be conversant with the database and user interface parts but may not be very knowledgeable about the computer communication part.

be a liability.

A more refined form of expert judgment is the estimation made by a group of experts. Estimation by a group of experts minimizes factors such as individual oversight, lack of familiarity with a particular aspect of a project, personal bias, and the desire to win contract through overly optimistic estimates. However, the estimate made by a group of experts may still exhibit bias on issues where the entire group of experts is biased due to reasons such as political considerations. Also, the decision made by the group may be dominated by overly assertive members.

Work Breakdown Structures:

This is bottom-up estimation tool. A work breakdown structure is a hierarchical chart that accounts for the individual parts of a system. A WEB chart can indicate either product hierarchy or process hierarchy. This product hierarchy helps in indentifying the manner in which the components are interconnected. A WEB chart indentifies the work activities and the relationship among those activities.
The advantages of WEB technique are in identifying and accounting for various process and product factors, in making explicit exactly which cost are included in the estimate.

3.9.1 Work Breakdown Structure

Work Breakdown Structure (WBS) is used to decompose a given task set recursively i
small activities. WBS provides a notation for representing the major tasks needed to
carried out in order to solve a problem. The root of the tree is labelled by the probl
name. Each node of the tree is broken down into smaller activities that are made
children of the node. Each activity is recursively decomposed into smaller sub-activities u
at the leaf level, the activities require approximately two weeks to develop. Figure
represents the WBS of an MIS (Management Information System) software.

While breaking down a large task into smaller subtasks, the manager has to make some hard decisions. If a task is broken down into a large number of very small activities, these can be distributed to a larger number of engineers. If the activity ordering permits, the solutions to these can be carried out independently. Thus, it becomes possible to develop the product faster (with the help of additional manpower of course!). Therefore, to be able to complete a project in the least amount of time, the manager needs to break large tasks into smaller ones, expecting to find more parallelism. However, it is not useful to subdivide tasks into units which take less than a week or two to execute. Very fine subdivision means that a disproportionate amount of time must be spent on preparing and revising various charts.

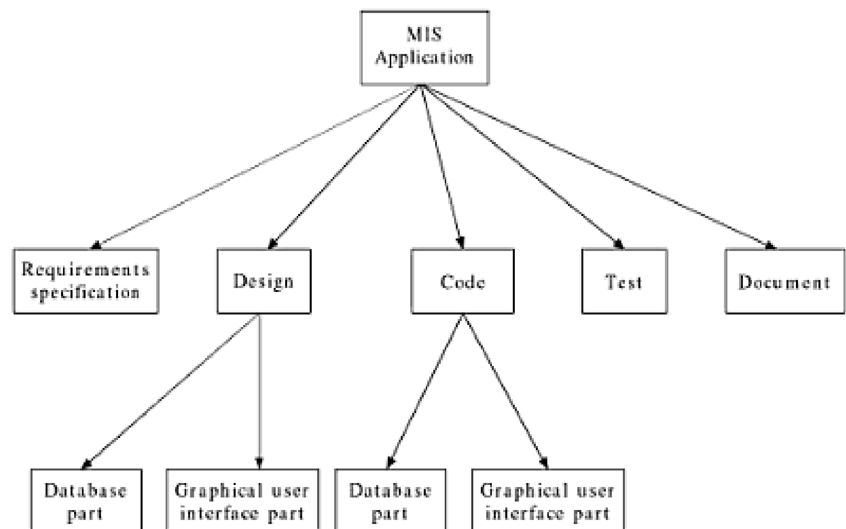


FIGURE 3.7 Work breakdown structure of an MIS problem.

Staffing-level Estimation

Once the effort required to develop a software has been determined, it is also necessar
determine the staffing requirement for the project. Putnam first studied the problem of
should be a proper staffing pattern for software projects. He extended the work of No
who had earlier investigated the staffing pattern of research and development (R&D) typ
projects. In order to appreciate the staffing pattern of software projects, we must
understand Norden's and Putnam's results.

3.8.1 Norden's Work

Norden studied the staffing patterns of several R&D projects. He found that the sta
pattern can be approximated by the Rayleigh distribution curve (see Figure 3.6). No
represented the Rayleigh curve by the following equation:

$$E = \frac{K}{t_d^2} * t * e^{-t^2/2t_d^2}$$

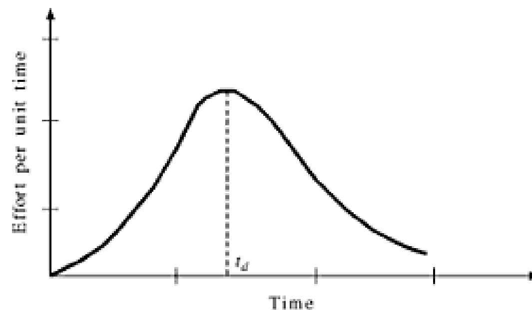


FIGURE 3.6 Rayleigh curve.

where E is the effort required at time t. E is an indication of the number of engineers (or the staffing level) at any particular time during the duration of the project, K is the area under the curve, and t_d is the time at which the curve attains its maximum value. It must be remembered that the results of Norden are applicable to general R&D projects and were not meant to model the staffing pattern of software development projects.

3.8.2 Putnam's Work

Putnam studied the problem of staffing of software projects and found that software development has characteristics very similar to any other R&D projects studied by Norden and that the Rayleigh–Norden curve can be used to relate the number of delivered lines of code to the effort and the time required to develop the product. By analyzing a large number of army projects, Putnam derived the following expression:

$$L = C_k K^{1/3} t_d^{4/3}$$

The various terms of this expression are as follows:

- K is the total effort expended (in PM) in the product development and L is the product size in KLOC.
- t_d corresponds to the time of system integration and testing. Therefore, t_d can be approximately considered as the time required to develop the software.
- C_k is the state of technology constant and reflects constraints that impede the progress of the programmer. The value of $C_k = 2$ corresponds to poor development environment (no methodology, poor documentation, and review, etc.), $C_k = 8$ corresponds to good software development environment (software engineering principles are adhered to), and $C_k = 11$ corresponds to an excellent environment (in addition to following software engineering principles, automated tools and techniques are used). The exact value of C_k for a specific project can be computed from the historical data of the organization developing it.

Putnam suggested that optimal staff build-up on a project should follow the Rayleigh curve. Only a small number of engineers are needed at the beginning of a project to carry out planning and specification tasks. As the project progresses and more detailed work is required, the number of engineers reaches a peak. After implementation and unit testing, the number of project members falls.

However, the staff build-up should not be carried out in large installments. The team size should be increased or decreased slowly whenever required to match the Rayleigh–Norden curve. Experience shows that a very rapid build up of project staff any time during the project development generally indicates schedule slippage.

It should be clear that a constant level of manpower throughout the project duration would lead to wastage of effort and increase the time and effort required to develop the product. If a constant number of engineers are used over all the phases of a project, some phases would be overstaffed and some understaffed leading to inefficient use of manpower, and consequent schedule slippage and increase in cost.

If we examine the Rayleigh curve, we can see that approximately 40% of the area under the Rayleigh curve is to the left of t_d and 60% to the right of t_d. This has been verified mathematically by integrating the expression provided by Putnam. This implies that the ratio of the effort required to develop the product to its maintenance effort is approximately 40 : 60. We have already seen in Chapter 2 that this is an acceptable pattern of distribution of effort between the development and maintenance of a product.

3.8.3 Effect of Schedule Change on Cost

By using the Putnam's proposed expression for L,

$$K = \frac{L^3}{(C_k)^3 (t_d)^4}$$

or

$$K = \frac{C}{(t_d)^4} \qquad (\because C = L^3/(C_k)^3 \text{ is a constant for the same pro}$$

Estimating Software Maintenance Costs

Software maintenance includes adding enhancements product, adapt product to new environment and correcting problems.
Concerns for maintenance are
Planning number of estimate number of maintenance programmers i.e., number of source instructions to be maintained.

Number of instructions that can be handled by the individual programmer.

Activity ratio I = SI added + SI modified

SI total

$$\text{Activity ratio } I = \frac{\text{SI added} + \text{SI modified}}{\text{SI total}}$$

SI = number of source instructions
PMm = ACT x MMdev
PMm = programmer months
PMm = ACT x EAF x MMdev
Where
EAF = Effective Adjustment Factor

The Software Requirements Specification

Software requirements analysis. The requirements gathering process is intensified and focused specifically on software. To

understand the nature of the program(s) to be built, the software engineer ("analyst") must understand the information domain for the software, as well as required function, behavior, performance, and interface. Requirements for both the system and the software are documented and reviewed with the customer.

It deals with the technical specification of requirements of the project. It is done in SRS (Software Requirements Specification). Its goal is to specify technical requirements for the software product in a precise and clear way. Software requirements specifications can presented using some automated tools.

The Software Requirements Specification is produced at the culmination of the analysis
task. The function and performance allocated to software as part of system engineering
are refined by establishing a complete information description, a detailed functional description, a representation of system behavior, an indication of performance requirements and design constraints, appropriate validation criteria, and other information pertinent to requirements.

A **Software Requirements Specification** (**SRS**) is a complete description of the behavior of the system to be developed. It includes a set of use cases that describe all the interactions the users will have with the software. Use cases are also known as functional requirements. In addition to use cases, the SRS also contains non-functional (or supplementary) requirements. Non-functional requirements are requirements which impose constraints on the design or implementation (such as performance engineering requirements, quality standards, or design constraints).

General Outline of a SRS

Software Requirements Specifications (SRS)

Cover Page

Revisions Page

Table of Contents

1 INTRODUCTION

1.1 Product Overview
1.2 Purpose
1.3 Scope
1.4 Reference
1.5 Definition And Abbreviation

2 OVERALL DESCRIPTION

2.1 Product Perspective
2.2 Product Functions
2.3 User Characteristics
2.4 General Constraints
2.5 Assumptions and Dependencies

3 SPECIFIC REQUIREMENTS

3.1 External Interface Requirements
3.1.1 User Interfaces
3.1.2 Hardware Interfaces
3.1.3 Software Interfaces
3.1.4 Communications Protocols
3.1.5 Memory Constraints
3.1.6 Operation
3.1.7 Product function
3.1.8 Assumption and Dependency
3.2 Software Product Features
3.3 Software System Attributes
3.3.1 Reliability
3.3.2 Availability
3.3.3 Security
3.3.4 Maintainability
3.3.5 Portability
3.3.6 Performance
3.4 Database Requirements
3.5 Other Requirements

4 ADDITIONAL MATERIALS

SRS for UMS (University Management System):

1. GENERAL DESCRIPTION – UMS is University Management System for managing the records of the alumni of the university as well as staff, faculty and higher authorities.

1.1 Purpose – The purpose for developing this type of software or introducing this UMS is to facilitate everyone who is concerned with the university.

1.2 Scope – The scope of UMS is global i.e. it should be able to be accessed from anywhere through internet i.e. registered users must be able to login to their accounts by directly accessing the university's website and then signing in with their username and password anytime and anywhere.

1.3 Abbreviation – UMS University Management System

1.4 Overview – As the ums is able to have a user interface. It should have a drop down boxes and if we drag mouse on any control at our welcome screen information regarding that the control should be displayed. Help menu should be there. As a teacher it should provide them to upload the various assignments and the attendance of the students. As a developer it should make a user interface which is user friendly. He should make the UMS as simple as he can. Backup at the main server should be made.

2. OVERALL DESCRIPTION

2.1 Product Perspective – product i.e. UMS should be able to provide a basic and easy interchange of information i.e. it should be able to remove the communication gaps between a teacher and the student. It should have chat facilities for all the users that are online. It should be compatible with all the operating systems.

2.2 Product Functions - The following are the product functions of the UMS:

- The UMS login box should on the official website of the university.
- The password field should be secured.
- After signing in all updates and new announcements for users should be displayed.
- By clicking on the dropdown box of the options the user should be able to view progress reports, assignments, notes, attendance, placement services and results.
- User should be able to change the passwords.
- Web pages should support pdf, ppt, doc and similar supported formats so that they can be easily downloadable and unloadable.

2.3 User Characteristics – A user can only have his/her registration number as username so if he joins the university then only he can login. This prevents misuse, unauthorized access and hacking of the product.

2.4 General Constraints – Server capacity is how many users can access or can be online at once. More is the number of users more will be the network traffic and hence the server comes in a down state. Personal firewall and updating is a tough task, it should be such that it should not block the network traffic, making the system slower. Firewall of the UMS should not collide with the firewall of the user system.

2.5 Assumptions and Dependencies – UMS should work even at when the network traffic is high. Server should have a power backup as well as a database backup. The UMS should be compatible with most of the operating systems i.e. previous and latest ones.

3 SPECIFIC REQUIREMENTS.

3.1 External Interface Required

3.1.1 User Interfaces – The external users are the students and the teachers of the university. The students can have an access to their accounts for their attendance, assignments etc. The teachers have also

an account to access their account for uploading of the students' attendance and the assignments to be submitted by them.

3.1.2 Hardware Interfaces – The external hardware interface used for accessing the UMS is the personal computers of the teachers and the students. The PCs may be laptops with wireless LAN as the internet connections provided will be wireless.

3.1.3 Software Interfaces – The Operating Systems can be any version of Windows, Linux, Unix or Mac which supports TCP/IP protocols.

3.1.4 Communication Interfaces – The communication interface is a local area network through wireless network routers.

3.2 Performance Requirements – The PCs used must be at least Pentium 4 machines so that they can give optimum performance of the product.

3.3 Design Constraints – The constraints at the designing time are that the needs of the university students and the teachers may keep on changing so the designers must keep this in view and design the product in this way that it is easily updatable.

3.4 Attributes – The following are the attributes of the product UMS:

- It should be equipped with current and archive database.
- All records can easily be updated.
- It should have its personal firewall.
- It should facilitate student with updating his/her account, downloading or uploading of assignments from anywhere.
- It should also do the same for teachers they can also have their pay checks online i.e. UMS should be capable of online transaction.

3.5 Other Requirements – The software is such that as the time goes by the need of the university management, students and teachers may keep on changing thus it is made to change from time to time.

4 ADDITIONAL MATERIALS

Data Flow Diagram for UMS:

A software requirements specification (SRS) is a comprehensive description of the intended purpose and environment for software under development. The SRS fully describes what the software will do and how it will be expected to perform.

An SRS minimizes the time and effort required by developers to achieve desired goals and also minimizes the development cost. A good SRS defines how an application will interact with system hardware, other programs and human users in a wide variety of real-world situations. Parameters such as operating speed, response time, availability, portability, maintainability, footprint, security and speed of recovery from adverse events are evaluated. Methods of defining an SRS are described by the IEEE (Institute of Electrical and Electronics Engineers) specification 830-1998.

THE COCOMO MODEL:

Boehm introduces a hierarchy of software estimation models bearing the genetic name COCOMO for **Constructive Cost Model**. The hierarchy of models takes the following form.

Model – 1. The basic COCOMO model is static single – valued model that computes software development effort (and cost) as a function of program size expressed in estimated lines of code (LOC).

Model – 2. The intermediate COCOMO model computes software development effort as a function of program size and a set of "cost drivers" that includes subjective assessments of the product, personnel, and project attributes.

Model – 3. The advanced COCOMO model incorporates all characteristics of the intermediate version with an assessment of the cost driver's impact on each step of the software engineering process.

The COCOMO model incorporates all characteristics of the intermediate version with an assessment of the cost driver's impact of each step of the engineering process.

The COCOMO models are defined for three classes of software projects. Using Boehm's terminology these are:

1. Organic mode – relatively small, simple software projects in which small team with good application experience work to set of less than rigid requirements.
2. Semi-detached mode – an intermediate (in size and complexity) software project in which teams with mixed experience levels must meet a blend of rigid and less than rigid requirements.

3. Embedded mode – a software project must be developed within a set of tight hardware, software and operational constraints.

The basic model is extended to consider a set “cost driver attributes” that can be grouped into 4 group categories:

1. Product attributes
    - Required software reliability
    - Size of application data
    - Complexity of the product
2. Hardware attributes
    - Run-time performance constraints
    - Memory constraints
    - Volatility of virtual machine environment
    - Required turn around time
3. Personnel attributes
    - Analyst capability
    - Software engineer capability
    - Application experience
    - Virtual machine experience
4. Project attributes
    - Use of software tools
    - Application of software engineering methods
    - Required development schedule

Each of the 15 attributes is rated on a six-point scale that ranges from “very low” to “extra high”. Based on this rating an effort multiplier is determined from the tables by Boehm and the product of effort multipliers is an effort adjustment factor (EAF).

The EAF values range from 0.9 to 1.4

The intermediate COCOMO model takes the form:

$$E = a_i \, (KLOC) \exp(b_i) * EAF$$

Where E is the effort applied in person-months and KLOC is the estimated number of delivered lines of code for the project. The coefficient a_i and the exponent b_i are taken from the table.

THE PUTNAM ESTIMATION MODEL:

Boehm introduces a hierarchy of software estimation models bearing the generic name COCOMO for **Constructive cost model.** The hierarchy of models takes the following form.

- Model – 1. The basic COCOMO model is static single-valued model that computes software development effort (and cost)

as a function of program size expressed in estimated lines of code (LOC).

- Model – 2. The intermediate COCOMO model computes software development effort as a function of program size and a set of "cost drivers" that includes subjective assessments of the product, hardware, personnel and project attributes.
- Model – 3. The advanced COCOMO model incorporates all characteristics of the intermediate version with an assessment of the driver's impact on each step of the software engineering process.

Formal Specification Techniques

Formal methods allow a software engineer to create a specification that is more complete, consistent, and unambiguous than those produced using conventional or object oriented methods. Set theory and logic notation are used to create a clear statement of facts (requirements). This mathematical specification can then be analyzed to prove correctness and consistency. Because the specification is created using mathematical notation, it is inherently less ambiguous than informal modes of representation.
Who does it? A specially trained software engineer creates a formal specification.

Why is it important? In safety-critical or mission critical systems, failure can have a high price. Lives may be lost or severe economic consequences can arise when computer software fails. In such situations, it is essential that errors are uncovered before software is put into operation. Formal methods reduce specification errors dramatically and, as a consequence, serve as the basis for software that has very few errors once the customer begins using it. What are the steps? The first step in the application of formal methods is to define the data invariant, state, and operations for a system function. The data invariant is a condition that is true throughout the execution of a function that contains a collection
of data, The state is the stored data that a function accesses and alters; and operations are actions that take place in a system as it reads or writes data to a state. An operation is associated with two conditions: a precondition and a post condition. The notation and heuristics of sets and constructive specification—set operators, logic operators, and sequences—form the basis of formal methods. What is the work

product? A specification represented in a formal language such as Z or VDM is produced when formal methods are applied. How do I ensure that I've done it right? Because formal methods use discrete mathematics as the specification mechanism, logic proofs can be applied to each system function to demonstrate that the specification is correct

Formal specification is part of a more general collection of techniques that are known as 'formal methods'. These are all based on mathematical representation and analysis of software. Formal methods include

- Formal specification;
- Specification analysis and proof;
- Transformational development;
- Program verification.

Acceptance of formal methods

- Formal methods have not become mainstream software development techniques as was once predicted quality. Hence the need for formal methods has been reduced; Market changes have made time-to-market rather than software with a low error count the key factor. Formal methods do not reduce time to market;
- The scope of formal methods is limited. They are not well-suited to specifying and analysing user

interfaces and user interaction; Formal methods are still hard to scale up to large systems.

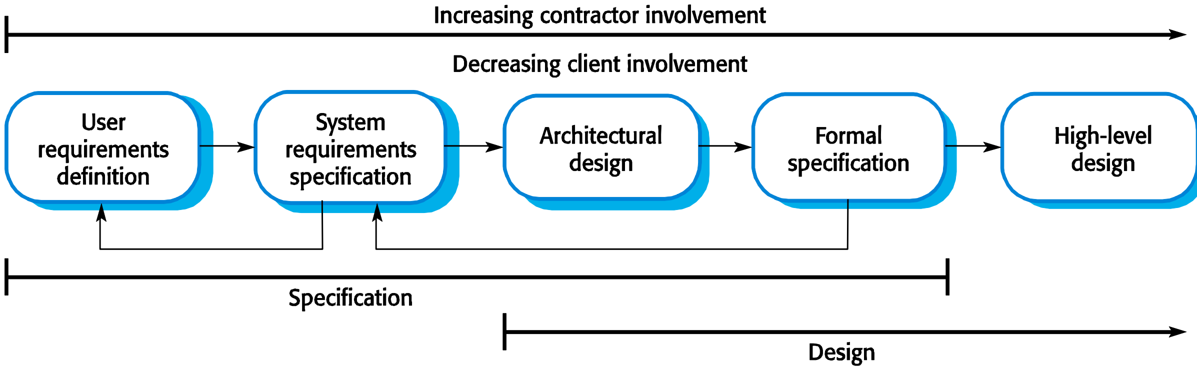
Use of formal methods

- The principal benefits of formal methods are in reducing the number of faults in systems.
- Consequently, their main area of applicability is in critical systems engineering. There have been several successful projects where formal methods have been used in this area.
- In this area, the use of formal methods is most likely to be cost-effective because high system failure costs must be avoided.

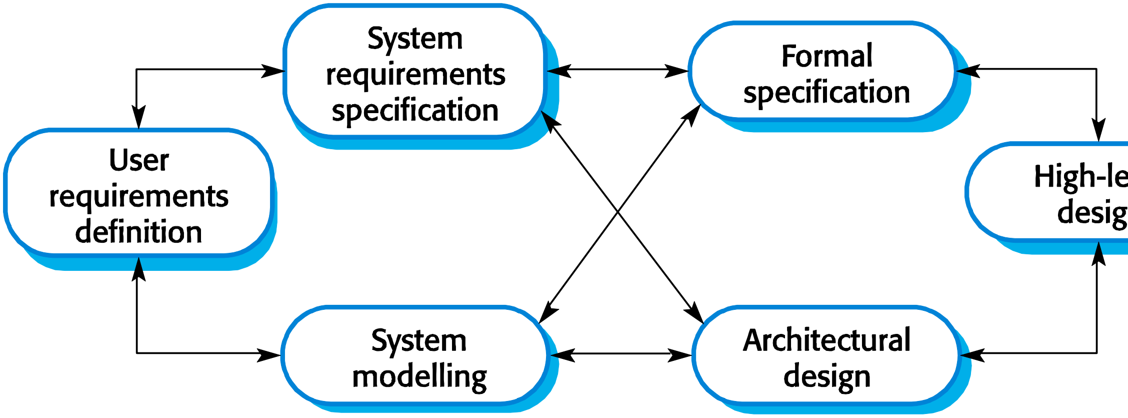
Specification in the software process

- Specification and design are inextricably intermingled.
- Architectural design is essential to structure a specification and the specification process.
- Formal specifications are expressed in a mathematical notation with precisely defined vocabulary, syntax and semantics.

Specification and design



Specification in the software process



Use of formal specification

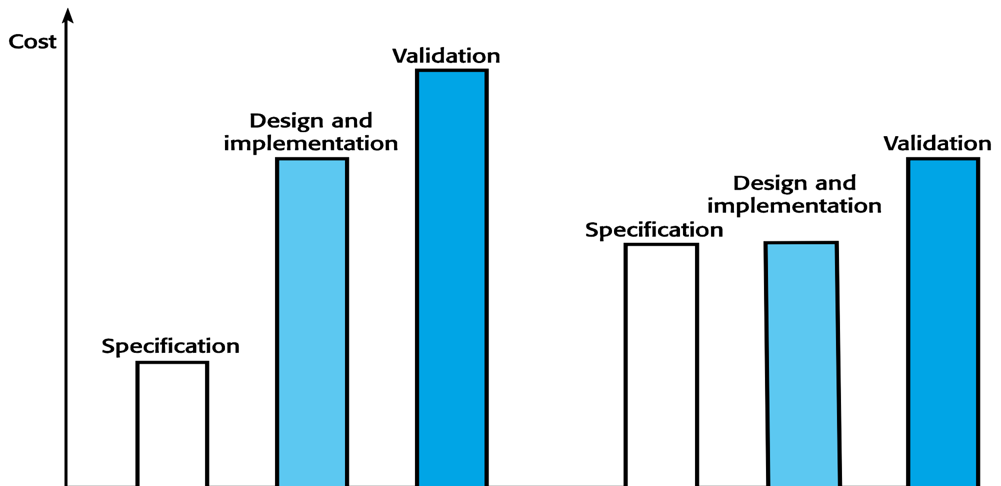
1. Formal specification involves investing more effort in the early phases of software development.
2. This reduces requirements errors as it forces a detailed analysis of the requirements.
3. Incompleteness and inconsistencies can be discovered and resolved.
4. Hence, savings as made as the amount of rework due to requirements problems is reduced.

Cost profile

1 The use of formal specification means that the cost profile of a project changes

- There are greater up front costs as more time and effort are spent developing the specification;
- However, implementation and validation costs should be reduced as the specification process reduces errors and ambiguities in the requirements.

Development costs with formal specification



Specification techniques

l Algebraic specification

- The system is specified in terms of its operations and their relationships.

l Model-based specification

- The system is specified in terms of a state model that is constructed using mathematical constructs such as sets and sequences. Operations are defined by modifications to the system's state.

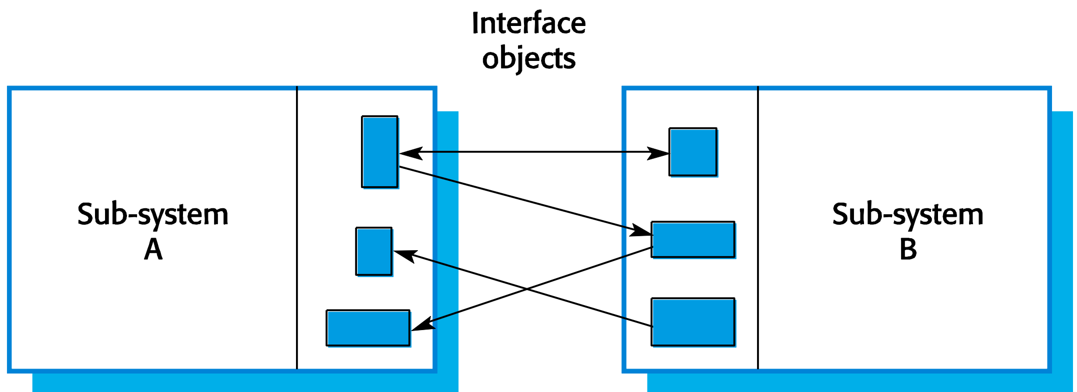
Formal specification languages

Interface specification

1. Large systems are decomposed into subsystems with well-defined interfaces between these subsystems.

2. Specification of subsystem interfaces allows independent development of the different subsystems.
3. Interfaces may be defined as abstract data types or object classes.
4. The algebraic approach to formal specification is particularly well-suited to interface specification as it is focused on the defined operations in an object.

Sub-system interfaces



Specification components

- Introduction
    - Defines the sort (the type name) and declares other specifications that are used.
- Description
    - Informally describes the operations on the type.
- Signature

    - Defines the syntax of the operations in the interface and their parameters.
- Axioms
    - Defines the operation semantics by defining axioms which characterise behaviour.

- Formal system specification complements informal specification techniques.
- Formal specifications are precise and unambiguous. They remove areas of doubt in a specification.
- Formal specification forces an analysis of the system requirements at an early stage. Correcting errors at this stage is cheaper than modifying a delivered system.
- Formal specification techniques are most applicable in the development of critical systems and standards.

Languages and Processors for Requirements Specification

In the context of computer-based systems (and software), the term specification means
different things to different people. A specification can be a written document, a graphical model, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these.
Some suggest that a "standard template" should be developed and used for a system specification, arguing that this leads to requirements that are presented in a consistent and therefore more understandable manner. However, it is sometimes necessary to remain flexible when a specification is to be developed. For large systems, a written document, combining natural language descriptions and graphical models may be the best approach. However, usage scenarios may be all that are required for smaller products or systems that reside within well-understood technical environments.
The System Specification is the final work product produced by the system and requirements engineer. It serves as the foundation for hardware engineering, software engineering, database engineering, and human engineering. It describes the function and performance of a computer-based system and the constraints that will govern its development. The specification bounds each allocated system element.

Software maintenance in software engineering is the modification of a software product after delivery to correct faults, to improve performance or other attributes..this section describes the six software maintenance processes as:

1. The implementation processes contains software preparation and transition activities, such as the conception and creation of the maintenance plan, the preparation for has handling problems identified during development, and the follow-up on product configuration management.
2. The problem and modification analysis process, which is executed once the application has become the responsibility of the maintenance group. The maintenance programmer must analyze each request, confirm it (by reproducing the situation) and check its validity, investigate it and propose a solution, document the request and the solution proposal, and, finally, obtain all the required authorizations to apply the modifications.
3. The process considering the implementation of the modification itself.
4. The process acceptance of the modification, by confirming the modified work with the individual who submitted the request in order to make sure the modification provided a solution.
5. The migration process (platform migration, for example) is exceptional, and is not part of daily maintenance tasks. If the software must be ported to another platform without any change in functionality, this process will be used and a maintenance project team is likely to be assigned to this task.
6. Finally, the last maintenance process, also an event which does not occur on a daily basis, is the retirement of a piece of software.

There are a number of processes, activities and practices that are unique to maintainers, for example:

- Transition: a controlled and coordinated sequence of activities during which a system is transferred progressively from the developer to the maintainer;
- Service Level Agreements (SLAs) and specialized (domain-specific) maintenance contracts negotiated by maintainers;

- Modification Request and Problem Report Help Desk: a problem-handling process used by maintainers to prioritize, documents and route the requests they receive;
- Modification Request acceptance/rejection: modification request work over a certain size/effort/complexity may be rejected by maintainers and rerouted to a developer.

Target Description Language

TDL is a descriptive language for the specification of hardware resources and assembly language of a given target processor. Each TDL specification is composed of several parts: a specification of the hardware resources, a description of the instruction set, a constraint section, and an assembly section.

nML

The nML language relates three different views of an instruction set: the behavior, the assembly syntax, and the object code image. It also allows the instruction set to be broken down into classes of instructions with shared code, syntax, and image. Addressing modes are also supported explicitly. nML is a form of attributed grammar. An attributed
grammar is a declarative description of a language with attributes added onto the "rules" of the grammar. Each view of the processor (action, syntax, and image) is an attribute attached to the rule for that instruction or addressing mode. The nML language has 3
important language features:
 Declarations
 Addressing Modes expressed as rules
 Instructions expressed as rules

LISA

LISA is a language designed for the formalized description of programmable architectures, their peripherals, and interfaces. It has C like syntax and it, furthermore, supports hierarchical modeling, allowing structured and easily maintainable code. The task of retargeting different tools like the compiler, the assembler, and the simulator of a given processor require different types of architecture related information. This is provided in different models[2]:
Memory model: list the register and memories and respective size.
 Resource model: describe available hardware resources and the resource requirement of operations.

 Behavioral model: abstraction of the activities of hardware structures changing the
state of the system.
 Instruction set model: collection of all instructions as combinations of hardware
operations that are permitted by the CPU controller.
 Timing model: specification of the activation sequence of hardware operations.
Of the above languages the only one that represents the scope of a cycle accurate processor description, is LISA. It is for this reason that there would be
Observable similarities between BPDL and LISA.

3. BPDL APPROACH

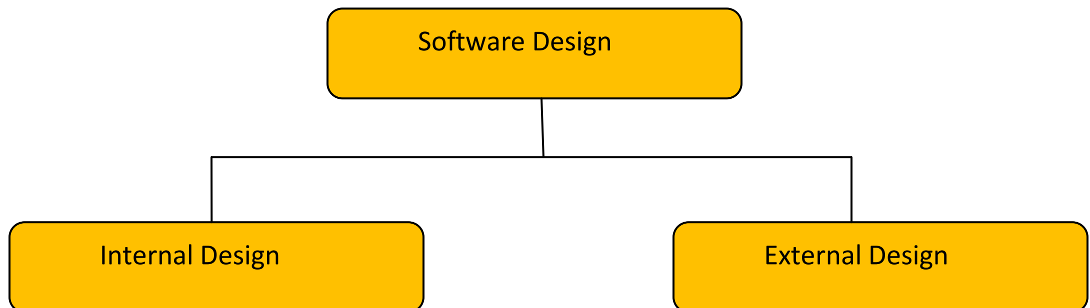
BPDL is a language designed for the formalized description of programmable architectures. The first version of BPDL contained many structural flaws, which were observable during the actual description of a processor. One of the most significant changes in the second version is the syntactical and conceptual similarity with C++. All the keywords specified in BPDL are in block letters to make them more visible. Assembly like specification of numerals is used, with numbers followed by ‘b’, ‘h’ and ‘d’ for binary, hexadecimal and decimal (default) respectively.

UNIT - III

Software design : Fundamental Design Concepts – Modules and Modularization Criteria – Design Notations – Design Techniques – Detailed Design Considerations – Real Time and Distributed System Design – Test Plans – Milestones, walkthroughs, and Inspections.

Software design:

Design involves conceiving and planning out in the mind and making a drawing, pattern or sketch of it. There are essentially 3 types of designs involved in software engineering:



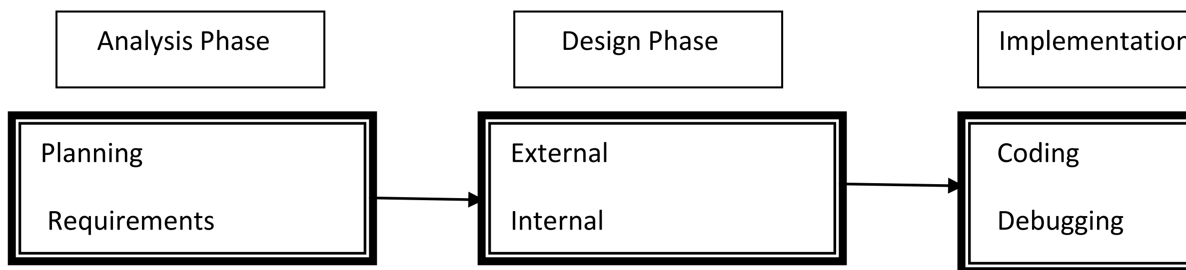
External Design

This involves conceiving, planning and specifying the external characteristics of a software product. A few notable external characteristics are user interface, report formats, data sources and data links. External design starts during the analysis phase and continues into the design phase. Once the requirements are obtained, they are refined and a high level architecture of the system is designed.

Internal design:

This involves conceiving, planning out and specifying the internal characteristics of a software product. Processing details are specified in this design phase. The architectural design is mapped to internal data structures and algorithms and a blue print is obtained.

Alternatives are compared and the trade-offs are understood which leads further into the implementation and testing stages.



Design is a meaningful engineering representation of something that is to be built. It can be traced to a customer's requirements and at the same time assessed for quality against a
Set of predefined criteria for "good" design. In the software engineering context, design focuses on four major areas of concern: data, architecture, interfaces, and components.

Why is it important? You wouldn't attempt to build a house without a blueprint, would you? You'd risk confusion, errors, a floor plan that didn't make sense, windows and doors in the wrong place . .. a mess. Computer software is considerably more complex than a house; hence, we need a blueprint— the design. What are the steps? Design begins with the requirements model. We work to transform this model into four levels of design detail: the data structure, the system architecture, the interface representation, and the component level detail. During each design activity, we apply basic concepts and principles that lead to high quality. What is the work product? Ultimately, a Design Specification is produced. The specification is composed of the design models that describe data, architecture, interfaces, and components. Each is a work product of the design process. How do I ensure that I've done it right? At each stage, software design work products are reviewed for clarity, correctness, completeness, and consistency with the requirements and with one another.

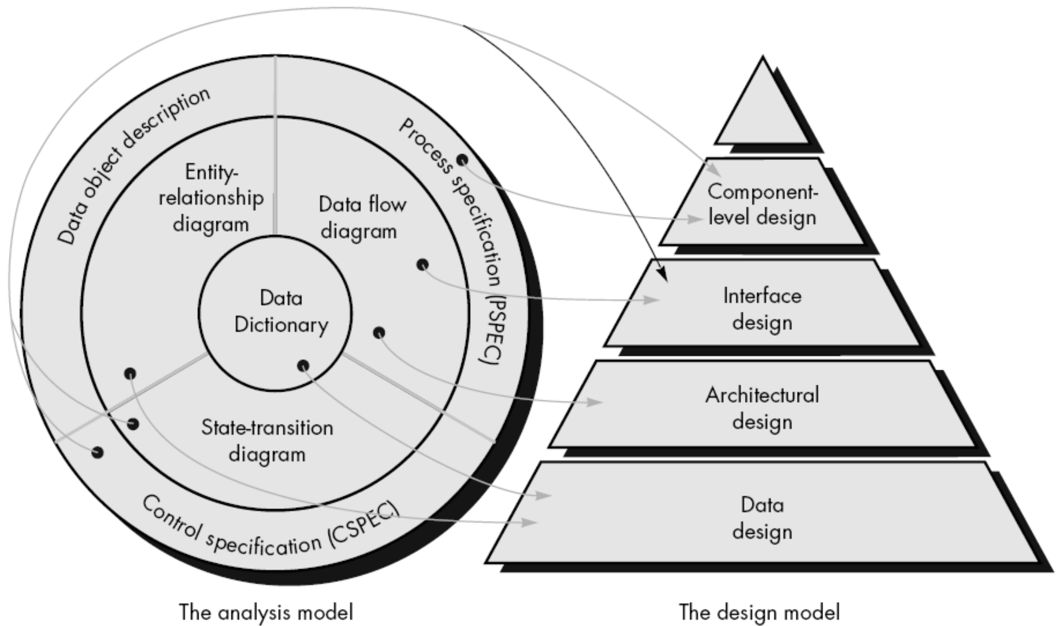


FIGURE 13.1 Translating the analysis model into a software design

ch of the elements of the analysis model provides information that is necessary to create the four design models required for a complete specification of design. The flow of information during software design is illustrated in Figure 13.1.Software requirements, manifested by the data, functional, and behavioral models, feed the design task. Using one of a number of design methods the design task produces a data design, an architectural design, an interface design, and a component design. The data design transforms the information domain model created during analysis into the data structures that will be required to implement the software. The data objects and relationships defined in the entity relationship diagram and the detailed data content depicted in the data dictionary provide the basis for the data design activity. Part of data design may occur in conjunction with the design of software architecture. More detailed data design occurs as each software component is designed. The architectural design defines the relationship between major structural elements of the software, the "design patterns" that can be used to achieve the requirements that have been defined for

the system, and the constraints that affect the way in which architectural design patterns can be applied [SHA96]. The architectural design representation—the framework of a computer-based system—can be derived from the system specification, the analysis model, and the interaction of subsystems defined within the analysis model. The interface design describes how the software communicates within itself, with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, data and control flow diagrams provide much of the information required for interface design. The component-level design transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the PSPEC, CSPEC, and STD serve as the basis for component design. During design we make decisions that will ultimately affect the success of software construction and, as important, the ease with which software can be maintained. But why is design so important? The importance of software design can be stated with a single word—quality. Design is the place where quality is fostered in software engineering. Design provides us with representations of software that can be assessed for quality. Design is the only way that we can accurately translate a customer's requirements into a finished software product or system. Software design serves as the foundation for all

Fundamental Design Concepts

The following are the fundamental concepts involved in software design:

Abstraction:

Abstraction is the theory that allows one to deal with concepts apart from the particular instances of those concepts. Abstraction is an important tool in software engineering in many aspects. For example one can easily specify the characteristics of the system during the requirements and design phase and hence separate it from its implementation details.
When we consider a modular solution to any problem, many levels of abstraction can

be posed. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more procedural
Orientation is taken. Problem-oriented terminology is coupled with implementation- oriented terminology in an effort to state a solution. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented.
[T]he psychological notion of "abstraction" permits one to concentrate on a problem at
some level of generalization without regard to irrelevant low level details; use of abstraction also permits one to work with concepts and terms that are familiar in the problem environment without having to transform them to an unfamiliar structure . . .
Each step in the software process is a refinement in the level of abstraction of the software solution. During system engineering, software is allocated as an element of
a computer-based system. During software requirements analysis, the software solution
is stated in terms "that are familiar in the problem environment." As we move through the design process, the level of abstraction is reduced. Finally, the lowest level of abstraction is reached when source code is generated. As we move through different levels of abstraction, we work to create procedural and data abstractions. A procedural abstraction is a named sequence of instructions that has a specific and limited function. An example of a procedural abstraction would be the word open for a door. Open implies a long sequence of procedural steps (e.g.,walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.). A data abstraction is a named collection of data that describes a data object In the context of the procedural abstraction open, we can define a data abstraction called **door**. Like any data object, the data abstraction for **door** would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions). It follows that the procedural abstraction open would make use of information contained in the attributes of the data abstraction **door**.
Many modern programming languages provide mechanisms for creating abstract
data types. For example, the Ada package is a programming language mechanism
that provides support for both data and procedural abstraction. The original abstract

data type is used as a template or generic data structure from which other data structures
can be instantiated.
"Abstraction is one of the fundamental ways that we as humans cope with complexity."
.
Abstraction and refinement are complementary concepts. Abstraction enables a
designer to specify procedure and data and yet suppress low-level details. Refinement
helps the designer to reveal low-level details as design progresses. Both concepts
aid the designer in creating a complete design model as the design evolves.

Information Hiding:

As seen earlier information hiding also an OO concept. Each module in a software system hides its processing details and activities and communicates with the other modules only through well-defined interfaces. Data abstraction is also an example of information hiding.

Concurrency:

Concurrent systems are those systems in which there are multiple independent processes which can be activated simultaneously. On a multi processor system such a task can be done by sharing them across the processor. On a single processor system, concurrency can be achieved by the process of interleaving.

Verification:

It is a basic concept of software design.
A design can be accepted by customer only if its verified and satisfies the customer requirements. The design is the kick-start for the project and so its verification to meet the client's needs is important.

Aesthetics:

In any art and technology aesthetic points are to be considerations. In software design the aesthetic consideration is an important area to be fine tuned by the designer to meet the end-user's taste. The design should be simple, elegant, neat and clear.

Structure

A structure is a fundamental concept of software design. As in modularity structure permits the breaking up or decomposition of a larger system into smaller units with well defined relationships between the different units. A network is a good example of a structure. A network has nodes and arcs and represented as a direct graph.

MODULES AND MODULARIZATION CRITERIA

Modularity

The concept of modularity in computer software has been espoused for almost five decades. Software architecture embodies modularity; that is, software is divided into separately named and addressable components, often called modules, that are integrated to satisfy problem requirements. It has been stated that "modularity is the single attribute of software that allows a program to be intellectually manageable" [MYE78]. Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a reader. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible. To illustrate this point, consider the following argument based on observations of human problem solving.

Modular decomposability:

If a design method provides a systematic mechanism for decomposing the problem into sub problems, it will reduce the complexity of the overall problem, thereby achieving an effective modular solution.

Modular composability:

If a design method enables existing (reusable) design components to be assembled into a new system, it will yield a modular solution that does not reinvent the wheel.

Modular understandability:
If a module can be understood as a standalone unit (without reference to other modules), it will be easier to build and easier to change.

Modular continuity:
If small changes to the system requirements result in changes to individual modules, rather than system wide changes, the impact of change-induced side effects will be minimized.

Modular protection:
If an aberrant condition occurs within a module and its effects are constrained within that module, the impact of error-induced side effects will be minimized.

Cohesion:
Cohesion is a natural extension of the information hiding concept. A cohesive module performs a single task within a software procedure, requiring little interaction with procedures being performed in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing. Cohesion may be represented as a "spectrum." We always strive for high cohesion, although the mid-range of the spectrum is often acceptable. The scale for cohesion is non linear. That is, low-end cohesiveness is much "worse" than middle range, which is nearly as "good" as high-end cohesion.

High cohesion

In computer programming, cohesion is a measure of how strongly-related or focused the responsibilities of a single module are. As applied to object-oriented programming, if the methods that serve the given class tend to be similar in many aspects, then the class is said to have high cohesion. In a highly-cohesive system, code readability and the likelihood of reuse is increased, while complexity is kept manageable.

Cohesion is decreased if:

- The functionalities embedded in a class, accessed through its methods, have little in common.

- Methods carry out many varied activities, often using coarsely-grained or unrelated sets of data.

Disadvantages of low cohesion (or "weak cohesion") are:

- Increased difficulty in understanding modules.
- Increased difficulty in maintaining a system, because logical changes in the domain affect multiple modules, and because changes in one module require changes in related modules.
- Increased difficulty in reusing a module because most applications won’t need the random set of operations provided by a module.

Types of cohesion

Cohesion is a qualitative measure meaning that the source code text to be measured is examined using a rubric to determine a cohesion classification. The types of cohesion, in order of the worst to the best type, are as follows:

1. **Coincidental cohesion (weak)**
    a. Parts of a component are simply bundled together.
2. **Logical cohesion (weak)**
    a. Components which perform similar functions are grouped.
3. **Temporal cohesion (weak)**
    a. Components which are activated at the same time are grouped.
4. **Communicational cohesion (medium)**
    a. All the elements of a component operate on the same input or produce the same output.
5. **Sequential cohesion (medium)**
    a. The output for one part of a component is the input to another part.
6. **Functional cohesion (strong)**
    a. Each part of a component is necessary for the execution of a single function.

7. **Object cohesion (strong)**

a. Each operation provides functionality which allows object attributes to be modified or inspected.

Coupling

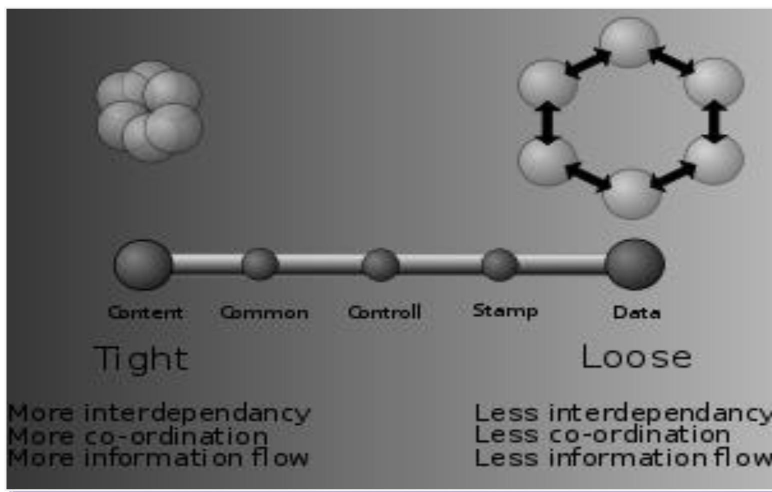
Coupling is a measure of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.

TYPES:

Content coupling (high)

Content coupling is when one module modifies or relies on the internal workings of another module (e.g., accessing local data of another module).

Therefore changing the way the second module produces data (location, type, timing) will lead to changing the dependent module.



Common coupling

Common coupling is when two modules share the same global data (e.g., a global variable).

Changing the shared resource implies changing all the modules using it.

External coupling

External coupling occurs when two modules share an externally imposed data format, communication protocol, or device interface.

Control coupling

Control coupling is one module controlling the flow of another, by passing it information on what to do (e.g., passing a what-to-do flag).

Stamp coupling (Data-structured coupling)

Stamp coupling is when modules share a composite data structure and use only a part of it, possibly a different part (e.g., passing a whole record to a function that only needs one field of it).

This may lead to changing the way a module reads a record because a field, which the module doesn't need, has been modified.

Data coupling

Data coupling is when modules share data through, for example, parameters. Each datum is an elementary piece, and these are the only data shared (e.g., passing an integer to a function that computes a square root).

Message coupling (low)

This is the loosest type of coupling. Modules are not dependent on each other, instead they use a public interface to exchange parameter-less messages (or events, see Message passing).

No coupling

Modules do not communicate at all with one another.

Design Notations

Graphical Design Notation

"A picture is worth a thousand words," but it's rather important to know which picture and which 1000 words. There is no question that graphical tools, such as the flowchart or box diagram, provide useful pictorial patterns that readily depict procedural detail. However, if graphical tools are misused, the wrong picture may lead to the wrong software.

A flowchart is quite simple pictorially. A box is used to indicate a processing step. A diamond represents a logical condition, and arrows show the flow of control.

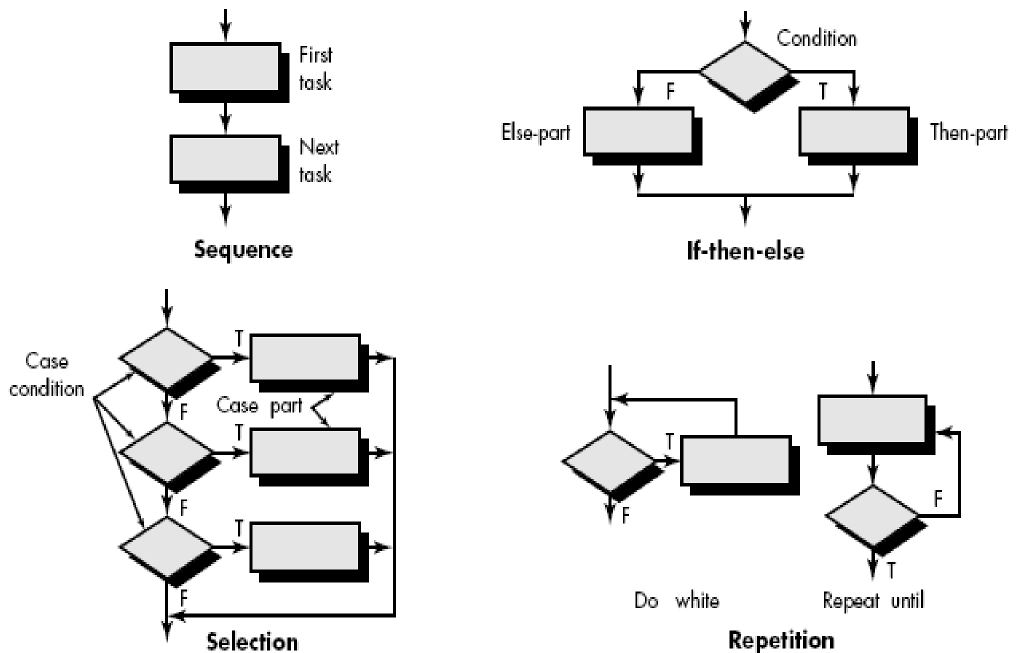


FIGURE 16.1
Flowchart constructs

Figure 16.1 illustrates three structured constructs. The sequence is represented as two processing boxes connected by an line (arrow) of control. Condition, also called ifthen- else, is depicted as a decision diamond that if true, causes then-part processing to occur, and if false, invokes else-part processing. Repetition is represented using two slightly different forms. The do while tests a condition and executes a loop task repetitively as long as the condition holds true. A repeat until executes the loop task first, then tests a condition and repeats the task until the condition fails. The selection (or select-case) construct shown in the figure is actually an extension of the if-then-else. A parameter is tested by successive decisions until a true condition occurs and a case part processing path is executed. The structured constructs may be nested within one another as shown in

UML is popular for its diagrammatic notations. We all know that UML is for visualizing, specifying, constructing and documenting the

components of software and non software systems. Here the Visualization is the most important part which needs to be understood and remembered by heart.

UML notations are the most important elements in modeling. Efficient and appropriate use of notations is very important for making a complete and meaningful model. The model is useless unless its purpose is depicted properly.

So learning notations should be emphasized from the very beginning. Different notations are available for things and relationships. And the UML diagrams are made using the notations of things and relationships. Extensibility is another important feature which makes UML more powerful and flexible.

Structural Things:

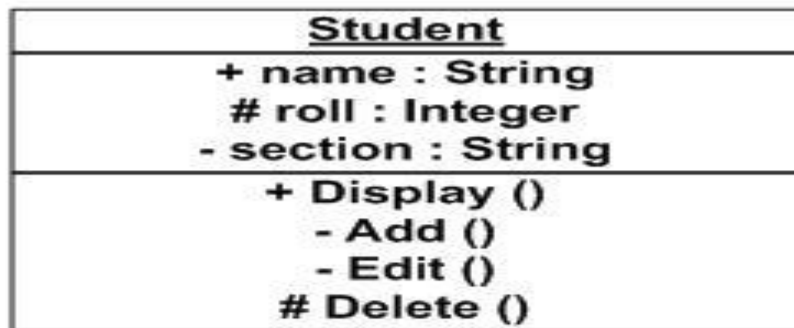
Graphical notations used in structural things are the most widely used in UML. These are considered as the nouns of UML models. Following are the list of structural things.

- Classes
- Interface
- Collaboration
- Use case
- Active classes
- Components
- Nodes

Class Notation:

UML class is represented by the diagram shown below. The diagram is divided into four parts.

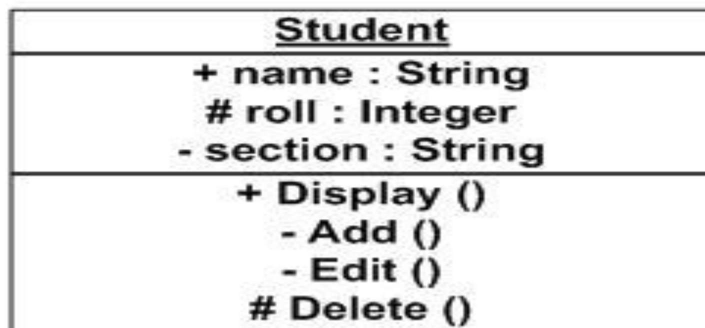
The top section is used to name the class.
The second one is used to show the attributes of the class.
The third section is used to describe the operations performed by the class.
The fourth section is optional to show any additional components.



Classes are used to represent objects. Objects can be anything having properties and responsibility.

Object Notation:

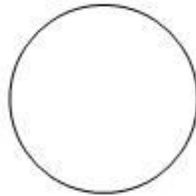
The object is represented in the same way as the class.



As object is the actual implementation of a class which is known as the instance of a class. So it has the same usage as the class.

Interface Notation:

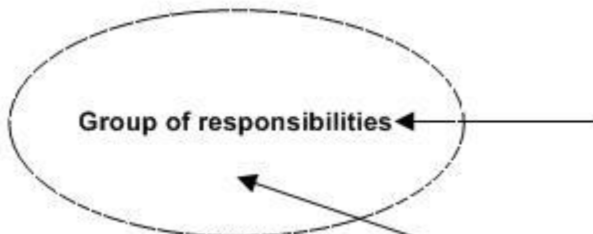
Interface is represented by a circle as shown below. It has a name which is generally written below the circle.



Interface is used to describe functionality without implementation. Interface is the just like a template where you define different functions not the implementation. When a class implements the interface it also implements the functionality as per the requirement.

Collaboration Notation:

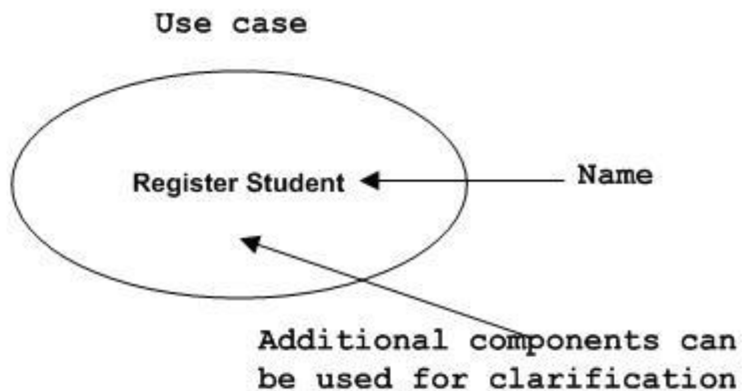
Collaboration is represented by a dotted eclipse as shown below. It has a name written inside the eclipse.



Collaboration represents responsibilities. Generally responsibilities are in a group.

Use case Notation:

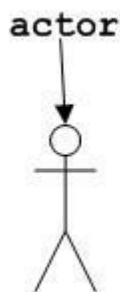
Use case is represented as an eclipse with a name inside it. It may contain additional responsibilities.



Use case is used to capture high level functionalities of a system.

Actor Notation:

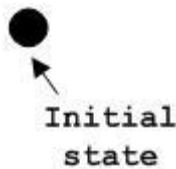
An actor can be defined as some internal or external entity that interacts with the system.



Actor is used in a use case diagram to describe the internal or external entities.

Initial State Notation:

Initial state is defined show the start of a process. This notation is used in almost all diagrams.



The usage of Initial State Notation is to show the starting point of a process.

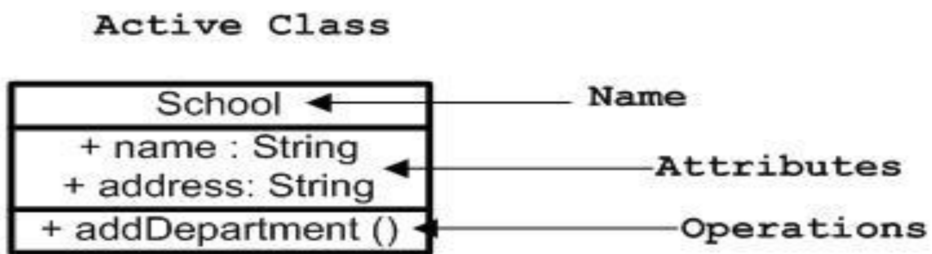
Final State Notation:

Final state is used to show the end of a process. This notation is also used in almost all diagrams to describe the end.

The usage of Final State Notation is to show the termination point of a process.

Active class Notation:

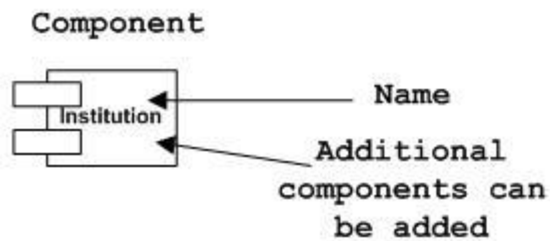
Active class looks similar to a class with a solid border. Active class is generally used to describe concurrent behavior of a system.



Active class is used to represent concurrency in a system.

Component Notation:

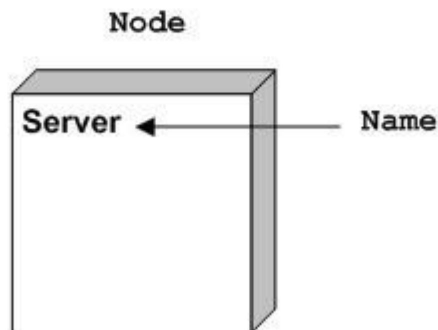
A component in UML is shown as below with a name inside. Additional elements can be added wherever required.



Component is used to represent any part of a system for which UML diagrams are made.

Node Notation:

A node in UML is represented by a square box as shown below with a name. A node represents a physical component of the system.



Node is used to represent physical part of a system like server, network etc.

Relationships

A model is not complete unless the relationships between elements are described properly. The Relationship gives a proper meaning to an UML model. Following are the different types of relationships available in UML.

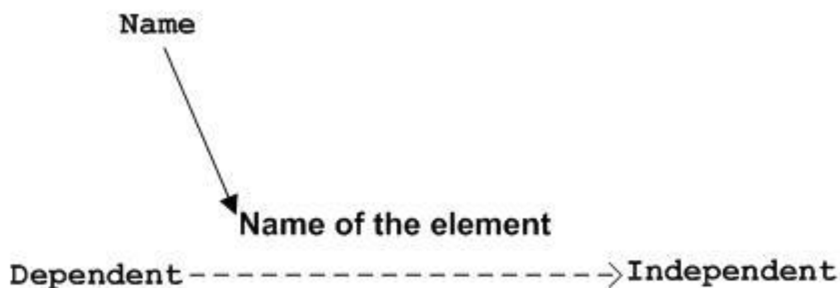
- Dependency

- Association
- Generalization
- Extensibility

Dependency Notation:

Dependency is an important aspect in UML elements. It describes the dependent elements and the direction of dependency.

Dependency is represented by a dotted arrow as shown below. The arrow head represents the independent element and the other end the dependent element.

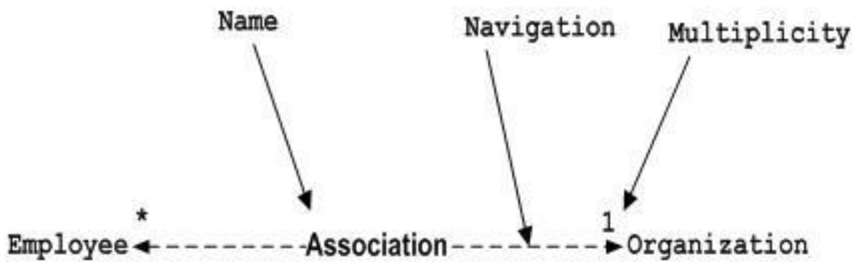


Dependency is used to represent dependency between two elements of a system.

Association Notation:

Association describes how the elements in an UML diagram are associated. In simple word it describes how many elements are taking part in an interaction.

Association is represented by a dotted line with (without) arrows on both sides. The two ends represent two associated elements as shown below. The multiplicity is also mentioned at the ends (1, * etc) to show how many objects are associated.



Association is used to represent the relationship between two elements of a system.

Generalization Notation:

Generalization describes the inheritance relationship of the object oriented world. It is parent and child relationship.

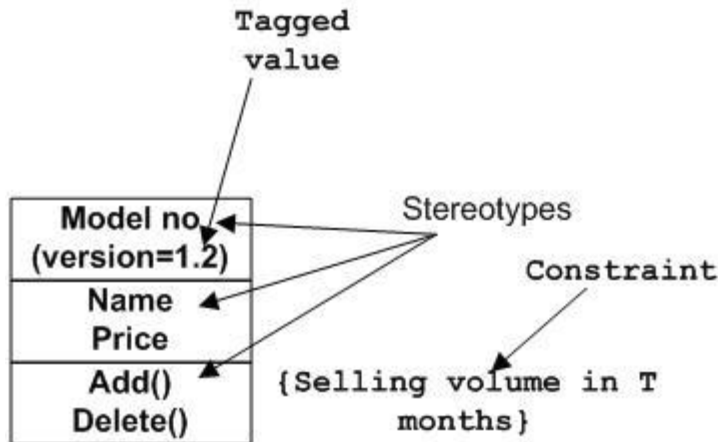
Generalization is represented by an arrow with hollow arrow head as shown below. One end represents the parent element and the other end child element.

Generalization is used to describe parent-child relationship of two elements of a system.

Extensibility Notation:

All the languages (programming or modeling) have some mechanism to extend its capabilities like syntax, semantics etc. UML is also having the following mechanisms to provide extensibility features.

Stereotypes (Represents new elements)
Tagged values (Represents new attributes)
Constraints (Represents the boundaries)



Extensibility notations are used to enhance the power of the language. It is basically additional elements used to represent some extra behavior of the system. These extra behaviors are not covered by the standard available notations.

Design Techniques

Design in software engineering involves
*Developing a conceptual view of the system
*Establishing a system structure
*Establishing data structure
*Identifying data stores and data sinks
*Decomposing high level functions into sub-functions
*Relating modules and determining the interconnections between the modules.
*Establishing perfect algorithms
There are several techniques developed for system design. These are called design methodologies or viewpoints or guidelines for the design process.

Refinement

Stepwise refinement is a top-down design strategy originally proposed by Niklaus Wirth. A program is developed by successively refining levels of procedural detail. A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached. An overview of the concept is provided by Wirth:

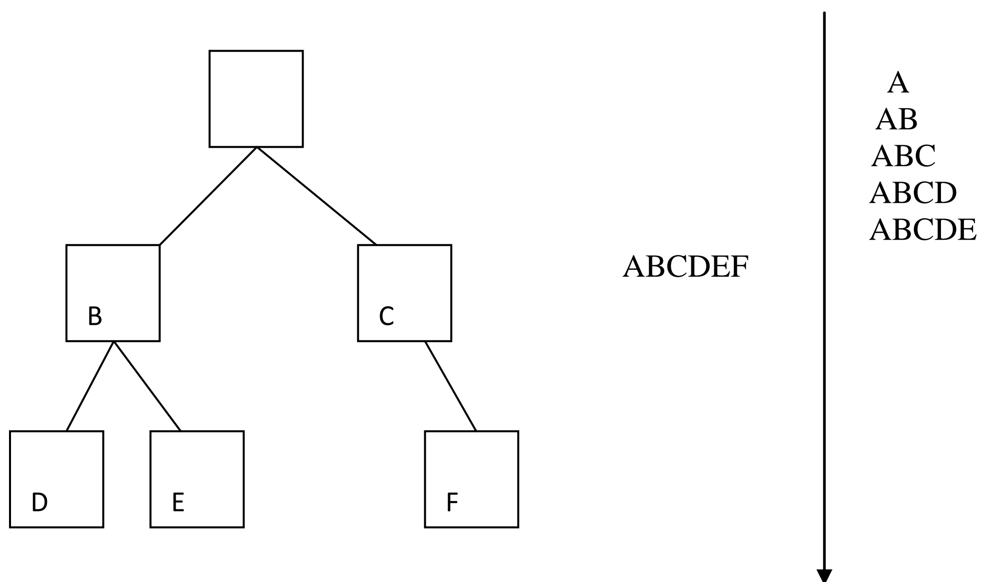
In each step (of the refinement), one or several instructions of the given program are decomposed into more detailed instructions. This successive decomposition or refinement of specifications terminates when all instructions are expressed in terms of any underlying computer or programming language . . . As tasks are refined, so the data may have to be refined, decomposed, or structured, and it is natural to refine the program and the data specifications in parallel.

Every refinement step implies some design decisions. It is important that . . . the programmer be aware of the underlying criteria (for design decisions) and of the existence of alternative solutions . . .

The process of program refinement proposed by Wirth is analogous to the process of refinement and partitioning that is used during requirements analysis. The difference is in the level of implementation detail that is considered, not the approach.

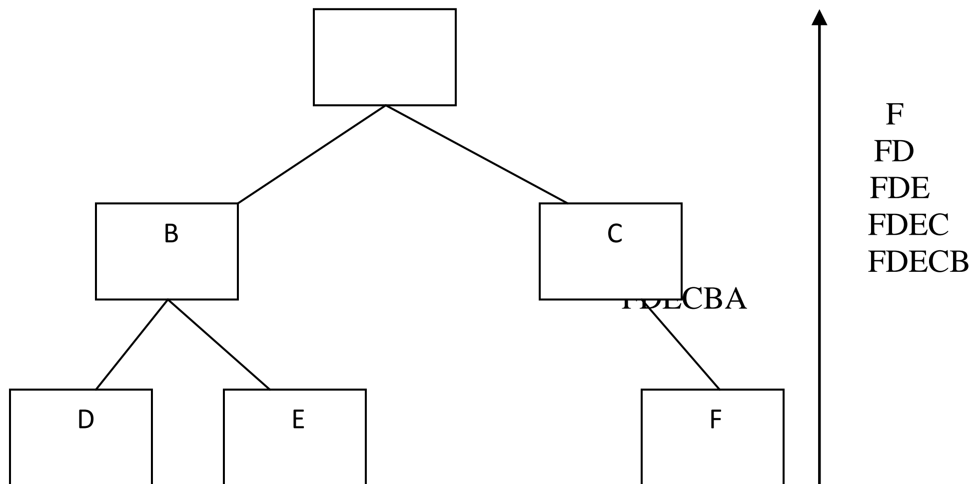
Refinement is actually a process of elaboration. We begin with a statement of function (or description of information) that is defined at a high level of abstraction. That is, the statement describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the information. Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

Abstraction and refinement are complementary concepts. Abstraction enables a designer to specify procedure and data and yet suppress low-level details. Refinement helps the designer to reveal low-level details as design progresses. Both concepts aid the designer in creating a complete design model as the design evolves.
Integrated Top down approach
Integrated top down approach integrates all the 3 basic activities namely design, development and testing.
Design is done in a top-down methodology from higher level routines which co-ordinate with the lower level functions and control them.



Bottom-up Approach illustration

This is the reversal of the top-down programming methodology. The program units from the bottom are added first and goes on up the tree toward the top level system.



Detailed Design Considerations

Detailed design is concerned with

- Details of algorithms
- Data representation
- Connection between function and data structure
- Packaging of software

During this phase semantics is given more importance than syntax. After architectural design detailed design is done. It involves in design of algorithms and data representations at higher level.

It helps in

*Easy implementation of product

*Inspections, walkthrough, reviews like CDR (Critical Design Review)

Product packaging

This concept deals several issues like how global data are shared among various module, structure of compilation, how parameters are passed, how programs and functions are made as single unit. Detailed design is done in such a way that I line in design give rise to maximum of 10 lines of source code.

Real – Time and Distributed System Design

Data architecture was generally limited to data structures at the program level and databases at the application level. But today, businesses large and small are awash in data. It is not unusual for even a moderately sized business to have dozens of databases serving many applications encompassing hundreds of gigabytes of data. The challenge for a business has been to extract useful information from this data environment, particularly when the information desired is cross functional. (e.g., information that can be obtained only if specific marketing data are cross-correlated with product engineering data).

Architectural styles

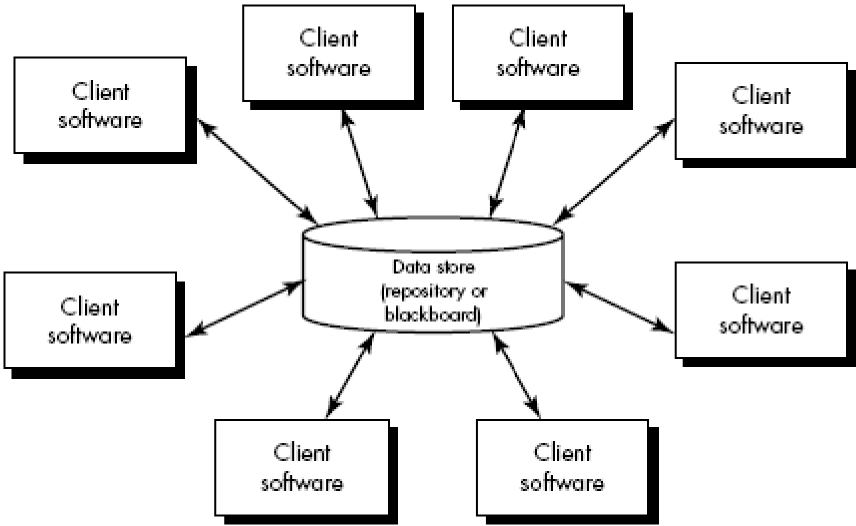
The software that is built for computer-based systems also exhibits one of many architectural styles.
1 Each style describes a system category that encompasses
(1) a set of components (e.g., a database, computational modules) that perform a function
required by a system;
(2) a set of connectors that enable "communication, coordinations and cooperation" among components;
(3) constraints that define how components can be integrated to form the system; and
(4) semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts [BAS98].
In the section that follows, we consider commonly used architectural patterns for software.

Data-centered architectures.

A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. Figure 14.1 illustrates a typical data-centered style. Client software accesses a central repository. In some cases the data repository is passive. That is, client software accesses the data independent of any changes to the data or the actions of other client software. A variation on this approach transforms the repository into a "blackboard" that sends

notifications to client software when data of interest to the client change.

Fig .14.1



Data-centered architectures promote integrability.

That is, existing components can be changed and new client components can be added to the architecture without concern about other clients (because the client components operate independently). In addition, data can be passed among clients using the blackboard mechanism (i.e., the blackboard component serves to coordinate the transfer of information between clients). Client components independently execute processes.

Data-flow architectures.

This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A pipe and filter pattern (Figure 14.2a) has a set of components, called filters, connected by pipes that transmit data from one component to the next. Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form. However, the filter does not require knowledge of the working of its neighboring filters. If the data flow degenerates into a single line of transforms, it is termed batch sequential. This pattern (Figure 14.2b) accepts a batch of data and then applies a series of sequential components (filters) to transform it.

FIGURE 14.2
Data flow architectures

(a) Pipes and filters

(b) Batch sequential

Call and return architectures.

This architectural style enables a software designer (system architect) to achieve a program structure that is relatively easy to modify and scale. A number of sub styles [BAS98] exist within this category:

• Main program/subprogram architectures. This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components, which in turn may invoke still other components. Figure 13.3 illustrates an architecture of this type.]

• Remote procedure call architectures. The components of a main program/ subprogram architecture are distributed across multiple computers on a network

Object-oriented architectures.

The components of a system encapsulate data and the operations that must be applied to manipulate the data. Communication and coordination between components is accomplished via message passing.

Layered architectures.

The basic structure of a layered architecture is illustrated in Figure 14.3. A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set. At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions. These architectural styles are only a small subset of those available to the software designer.2 Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural pattern (style) or combination of patterns (styles) that best fits those characteristics and constraints can be chosen. In many cases, more than one pattern might be appropriate and alternative architectural styles might be designed and evaluated.

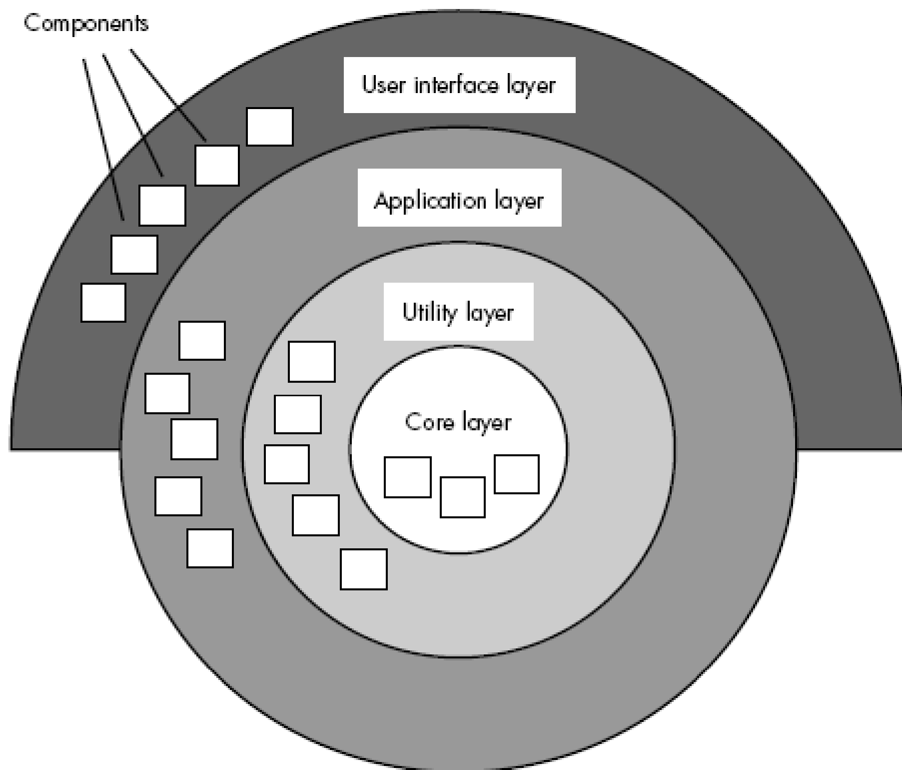


Fig 14.3

Test Plans

A test plan describes the various activities that are to be conducted to demonstrate that a software product does meet the requirements specified.
The chief objective of testing is to obtain an error – free pristine software product.
The test plan contains information such as the objective of testing, test completion criteria, test cases, special issues and so on. The various tests that will be performed are:

Functional Tests:

Are conducted with typical working conditions- with typical input values and the appropriate expected output values. The initial values assumed, defaults system values, ordering of data, data boundaries are a few that are included in the functional tests.

Performance tests:

As the name implies test the system performance. These test the response time, program execution time, I/O access times, memory utilization. Also they test the load on the communication system if the software is a network based product. Bottlenecks are identified where data stagnates and appropriate results are not arrived at the right time.

Stress Tests:

Stress tests are used to determine the system performance under OVERLOAD. All exceptional cases are handled in this test to determine how good a system behaves under catastrophic conditions. A fine example of such a test is to disconnect a communication link when the system is processing some request and then determine how the system tries to gracefully handle the exception.

Other ways to stress test a system are to “login” from more than a terminal and determining the system load. Stress test provide valuable information regarding the system’s weaknesses and strengths. Design flaws can be found in such test and the system may require fine tuning.

Structural Test:

Structural test are those tests which consider the actual execution logic of the software product. All loops, paths and routines are tested with all possible combinations of input values so that each and every logical path in the routine is traversed and errors are detected. This makes the testing complete in its aspect. Structural rests are the real test in terms of finding the flaw in the actual algorithm used to accomplish a particular task. These tests also determine unused paths or those paths which will never execute in a program and hence aid in the removal of unwanted or unreachable code.

Milestones, walkthroughs, and Inspections.

MILESTONES

A Software product or a software project on its entire growth from the preliminary requirements stage to the implementation maintenance and the user manual preparation stage undergoes lots of changes, development, corrections and so on.

Milestones are somewhat similar to the bookmarks in the way they serve the intellectual community. Milestones are established to keep the project on the right track, increase communication between the developers and the clients, following the project schedule and adhering to standards and the requirements.
There are two milestones that are conducted during the course of a software project.

Walkthroughs and Inspections

Walkthroughs are a simple, practical, well- accepted technique for drastically improving the quality of software. A walkthrough is peer group review of any product.

“People never seem to have time to do the job right the first time, but they always have the time to do it twice.”A walkthrough group reviews the material or the work product of a review. The person who created the product has to assist his peers who review his product by providing appropriate documents and materials. The emphasis here is to detect bugs and not fixing bugs. Once the design or code flaws are determined after having . “ Walked-through” , a secretary records all the bugs or the errors and keeps track of them and also prepares a checklist.
It is the job of the person who developed the work product or the material to fix the bugs and report to the review team usually consists of a group of 4-5 people.

Walkthrough, or **walk-through**, is a term describing the consideration of a process at an abstract level.

The term is often employed in the software industry (see software walkthrough) to describe the process of inspecting algorithms and

source code by following paths through the algorithms or code as determined by input conditions and choices made along the way. The purpose of such code walkthroughs is generally to provide assurance of the fitness for purpose of the algorithm or code; and occasionally to assess the competence or output of an individual or team.

The term is employed in the theatrical and entertainment industry to describe a rehearsal where the major issues of choreography and interaction are practiced and resolved, prior to more formal "dress rehearsals".

The term is often used in the world of learning where a tutor/trainer will walk through the process for the first time. It is regarded as a literal walk through of the learning at the groups pace ensuring that everyone takes in the new knowledge and skills.

Benefits:

1. Find errors, quickly and economically.
2. Increase programmer productivity
3. Reduce maintenance effort.
4. Spot gross design or implementation inefficiencies.
5. Establish standards for analysis, design, coding, testing, and documentation.
6. Training
7. Insurance that work can be salvaged if someone leaves.
8. Determine employee efficiency and hence productivity

Types of Walkthroughs:

Formal

1. For the big-wigs
2. Often superficial
3. Slow feedback
4. Critiques whose quality is highly variable
5. Review by supervisors
6. Long preparation time
7. Relatively complete products

Informal

1. Little preparation from rough notes
2. Quick feedback review by peers
3. Reviewers have “ownership”

Semi-formal

1. Review by peers entirely on the project
2. Has agenda, time limit, and a set of procedures.

Guidelines to be observed during walkthroughs:

- Ensure all work products are reviewed. This includes the developer to the project leader. Ensuring this will create a congenial atmosphere for project development.
- Major issues must be addressed.
- Walkthroughs should be of a strict duration without dragging them for longer times.
- Avoid correction of errors during walkthroughs and ensure detection of errors.
- Review the product, not the producer.
- Set an agenda and maintain it.
- Limit debate and rebuttal.
- Highlight problem areas, but don’t attempt to solve every problem noted.
- Take written notes.
- Limit the number of participants and insist upon advance preparation.
  (3-4)
- Develop a checklist for each work product that is likely to be reviewed.
- Allocate resources and time schedule for quick completion.
- Conduct meaningful training for all reviewers.

Inspections:

The IEEE defines an inspection as “ a formal verification techniquole assigned to them. One might test in which software requirements, design or code is examined in detail by aperson or a group other than

the author to detect faults, violations of development standards and other problems”.

- Inspections make use of checklists of items to be examined on the work products.

- Unlike walkthroughs, here each “inspector” inspects the product based on the role assigned to them. One might test the user interface, one might inspect the functionality and so on.
- Items to be inspected in a code are subprogram interfaces, decisions made, data referencing, I/O statements, comments, data flow and memory usage.

Some examples of checklist items include

- Agreement between number of formal and actual parameters.
- Are constants being passed to modifiable arguments?
- Consistent usage of global variables among various modules.
- Maximum number of lines in a module.
- Agreement in naming conventions to be followed.
- Are comments being used appropriately and self documenting?

UNIT - IV

Implementations issues: Structured Coding Techniques – Coding Style – Standards and Guidelines – documentation guidelines – Type Checking – Scoping Rules – Concurrency Mechanisms.

Implementations issues:

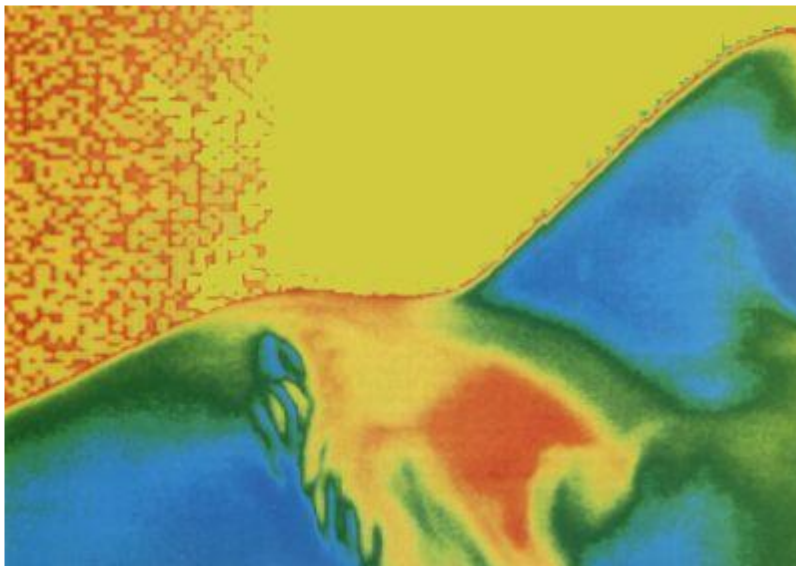
Structured Coding Techniques:

Over the past several months, we've talked a lot about concepts that distinguish the discipline of software engineering from the art of computer programming. The underlying theme has been that a disciplined, structured approach to software development will result in a final product of higher quality and greater durability than will a more casual approach. This durability becomes particularly important when your software system enters the maintenance phase, which is

everything that you do to the system after you thought you were done with it. Another of our goals is to design and write program modules so that they can be reused in future projects, thereby saving us the pain of continually reinventing all sorts of wheels.

We've talked a lot about how to decompose even a very complex software system into a collection of well-defined, bite-sized processes. If we do this properly, we can go off and write program modules (subroutines, functions, procedures) in our favorite language with confidence that we'll be able to plug the pieces together successfully at the end. Last time, we looked at some ideas for documenting our final product, in the form of both internal comments describing each specific program module and external documentation showing how the system as a whole fits together and operates.

The one aspect of software development we really haven't addressed is the actual writing of the computer program itself. I've been making the assumption that most of you are seasoned programmers, with some experience in modern structured languages such as C, Pascal or GFA BASIC. But perhaps the time is ripe to re-hash a topic that helped begin the software engineering revolution: structured programming.



STRUCTURED • PROGRAMMING

While programming, continually ask yourself,

"How would I like to be the one who has to read, understand and change this program in the future?"

What is structured programming?

It's difficult to provide a concise answer to this question. In some ways, it fits into the "I know it when I see it" category, although we will discuss some specific structured programming principles shortly. But on the other hand, structured programming is consistent with our general philosophy of software engineering, which emphasizes enhanced communication between people. Computer hardware has become powerful enough that programmers should no longer emphasize compactness of code and cleverness of algorithm on their road to the nirvana of "efficiency." Contemporary software development stresses clarity and structure of code, not cuteness. While programming, continually ask yourself, "How would I like to be the one who has to read, understand and change this program in the future?" Structured programming can be a powerful aid to effective software maintenance. And in some ways, structured programming is an attitude, as well as a behavior. The emphasis is on person-to-person efficiency, not computer efficiency.

The golden rule

In my opinion, the single most important idea of structured programming is that the code you write should represent a clear, simple and straightforward solution to the problem at hand. Keep this guiding principle in mind as you read the rest of this article.

Start at the top

One of the basic precepts of structured programming has been with us clear through our software engineering sojourn: a structured program consists of a hierarchical collection of individual modules, which appear more abstract at the top levels and more detailed at the lower levels. This fits with our overall strategy of hierarchical decomposition, which we've followed from structured system specification through structured design and now down to structured programming. The process of building a program in this fashion is

called "top-down programming," or step-wise refinement Another point I stressed during our discussion of program design is that each of these modules (or processes, in design terms) communicates with others through well-defined data interfaces. These data interfaces typically are subroutine parameter lists or function argument lists. Each part of the program appears to other parts as a black box that simply performs its assigned function in some unknown way. You've encountered this idea every time you called a built-in function in some programming language. Think about 8-bit Atari BASIC. Do you recall the STICK function? It told you something about the position of the joystick. Do you know how to communicate with the STICK function? Yes; all you had to do was pass it the number of the joystick you were interested in, like this: STICK(1), and it returned a numeric answer. Do you know how it worked? No; could be magic, for all you know. Do you care? No. This is the beauty of a "black box" approach to software development. There's no reason why the modules you write should be any different in this regard than the modules supplied by the guys who wrote the language you're using.

In practice, you apply the notion of step-wise refinement by writing your initial description of each fundamental process in a very high-level "language" that we called pseudo code. This is a first attempt at a picture of how each process will accomplish its assigned task. As you continue down the path from design toward code, you add detail to this description until eventually you reach something that conforms to the exact syntax of the language you're using: source code. I suppose we could consider that one additional step takes place even after this, which is the compiling of your source code into something the computer can deal with: object code. Fortunately, we humans can halt our step-wise refinement at the source code stage and let the machine take over from there. (Someday, we'll be able to stop at the pseudocode stage.)

The dreaded GOTO

The first thing most people learn about structured programming is that you shouldn't use GOTO statements. This notion stems from an article published in 1968 by E. W. Dijkstra, which helped the structured programming push get underway in earnest. The succinct title said it all: "Go To Statement Considered Harmful" (Communications of the ACM, 11 (3), 147-148).

We don't always have a choice. In older forms of BASIC, GOTO statements were needed everywhere, because there just wasn't the richness of commands that we need to avoid GOTO. The careless use of GOTO inevitably leads to the notorious "spaghetti code" that makes a program nearly impossible to comprehend and debug. The worst case is a GOTO that branches back to a previous statement in the program listing.Modern programming languages provide logic and control commands that allow us to almost completely avoid using GOTO statements (we'll discuss these shortly). However, there are still a few situations in which a GOTO actually can result in cleaner, more understandable code. Error-handling situations sometimes benefit from GOTOs. Premature exits from loops, or breaking out of deeply nested IF structures, may be more easily handled with a GOTO than by some other method. Nonetheless, the general guideline that GOTOs should not be used for routine transfer of control within a program is still valid, so try to break any lingering bad habits from your earlier experiences with BASIC interpreters.

A question of style

The programming style you use can greatly influence the readability of your code. While not strictly part of structured programming, there are some matters of style to keep in mind. A pretty good book on this topic is The Elements of Programming Style, Second Edition, by Kernighan and Plauger (McGraw-Hill, 1978). It's somewhat dated, and the code examples are all in FORTRAN or PL/I, but the principles remain the same. The following represent some of the highlights of programming style in my mind; some don't apply to all programming languages. Use indentation to visually block logically related sections of your code, such as the sections of IF/ELSE IF/ELSE/END IF and SELECT/CASE constructs. Use blank lines in the source code to further delineate sections of the program. Use comments judiciously in the source code for clarification (we talked about this last time), and make sure the comments are accurate. Don't bother to document bad code—rewrite it, instead. Never put more than one statement on the same source line. We always did this in Atari BASIC because it saved six bytes per statement, but you don't need such tricks when you have a megabyte at your disposal.

Select meaningful variable and procedure names. Don't use different names to represent the same piece of information (refer to your data dictionary). Explicitly declare the type of each variable used, if your

language permits this. Use parentheses to resolve any ambiguities in mathematical expressions, even if they aren't required for the operation to be executed correctly. Make sure conditional tests (IF some conditions THEN do something) read clearly. Generally speaking, the first condition tested for should be the desired condition, with an error condition handled second.

Program defensively: Try to anticipate all possible errors in input data or mathematical operations, and write code to handle such situations. This includes validating input data before trying to use it and testing for such mathematical problems as division by 0 or taking the logarithm of a negative number. Make sure that input data does not exceed the bounds of what the routine can handle. Think of the user when designing your programs; make input easy to prepare correctly, and make output self-explanatory.

Initialize variables before using them. Who knows what was in those bytes before they were reserved for a variable's use? Avoid multiple entry points, and exits from loops and subroutines (more about this later).

The first priority is to get the program running correctly. You can worry about optimization later. And when you do, make sure the program still runs correctly. Don't try to optimize every little step; the compiler will do a lot of this for you. Usually, a program spends most of its time in a small section of the code, so concentrate your optimization efforts here (if you can find it).

Use the best algorithms you can find for calculations, but remember that both the algorithm and the structure of your data will influence how the algorithm will be implemented in code. Insert "instrumentation" checkpoints in your programs to write out intermediate results someplace; so that you can verify accuracy, track down errors and assess efficiency. These outputs can be sent to a trace file on the disk, which you can then examine at your leisure.

Building blocks

Another basic premise of structured programming states that any program logic, no matter how complex, can be expressed in terms of just three kinds of logical operations: sequence, selection and

iteration. A program then is made up of a series of blocks of code to perform these operations. Let's define these three kinds of operations.

Sequence—a series of program statements are executed one after the other, in the order in which they appear in the source code. Obviously, this rules out statements like GOTO and IF, restricting us just to statements that perform some specific action. (A CALL or GOSUB to another procedure would qualify as an action in this sense.) Hence, a block of statements that are executed sequentially is called an "action block."

Selection—one set of statements, from a choice of two or more, is selected for sequential execution, based on some criterion. One way to accomplish this is to use an IF/THEN/ELSE construct. Some of the languages will also permit a SELECT/CASE/OTHERWISE/END - type structure, perhaps with different but analogous keywords. The set of statements that gets executed in each case is itself an action block. Sometimes selection constructs are called "branch blocks."

Iteration—a series of statements is executed repeatedly until some termination condition is met. These are also called "loop blocks." Virtually all languages contain simple FOR/NEXT or DO/END-type loops. More modern languages include variations such as DO UNTIL/END and DO WHILE/END.

These three kinds of "control blocks" have some features in common. First, the code in each is executed from top to bottom, which is the same way that it appears in the source file. This makes the program much easier to read and understand than does the convoluted branching you find in so many BASIC programs. Of course, in a selection block, not every statement is executed, and in a loop block they may be executed more than once, but they still are always executed from top to bottom.

In addition, each control block has just one logical entry point: the first statement. And if they're well structured, they have just one logical exit point: the last statement. A complete program is written by assembling and nesting blocks of these three kinds to perform the required processing.

Believe it or not, it's possible to write more or less structured programs in BASIC by following these rules. Some simulation of

certain missing language features is required and some GOTOs inevitably creep in. But by keeping the notion of just three flavors of control blocks in mind, a surprisingly good job of structured programming can be done. If you still use BASIC, I encourage you to read a series of articles on structured programming in BASIC by Arthur Luehrmann in the May, June and July 1984 issues of Creative Computing. Unfortunately, Creative Computing is no longer published, but perhaps you or a friend has these back issues in the dusty magazine archives.

More iteration

I imagine you're pretty comfortable with the ideas behind action blocks and selection blocks, but let's take a close look at the iteration, or repetition, constructs. I'm sure you're familiar with the simplest type of loop, which looks like this in BASIC:

```
FOR I = 1 TO 10
  Calculate something
NEXT I
```

In other languages, such a loop is commonly called a DO loop and is terminated by an END or END DO statement:

```
DO I = 1 TO 10
Calculate something

END
```

In either case, some test is used to determine the number of times the loop is executed. In these simple examples, a variable called I (the index variable) is incremented after each iteration and compared to the value 10. If I is less than or equal to 10, the statements in the loop are executed again; otherwise, execution of the loop terminates. Here we're assuming that the value of I will go up by one on each iteration. Of course, you can set a different step interval with a statement like: DO I = 1 TO 10 BY 0.5.

Here's the key question: Is the comparison done before the loop is executed or after? There's a big difference. Suppose I has a value of 20 at the time this loop is encountered in the course of executing the

program. Will the loop be executed (since the value of the index variable is already greater than 10) or not?

In Atari BASIC, the comparison is done after the contents of the loop are executed, so the loop is always executed at least once. I think this is generally true of simple FOR/NEXT and DO/END loops. Many modern languages resolve any ambiguity by providing two explicit statement choices: DO WHILE and DO UNTIL. In a DO UNTIL loop, the termination condition is tested at the end of the loop, so the loop is always executed at least once. The simple FOR/NEXT loop in BASIC is thus a DO UNTIL type loop. In a DO WHILE construct, the termination condition is tested at the beginning of the loop. If the termination condition is already true, the contents of the loop aren't executed at all.

One important point is that DO WHILE and DO UNTIL loops need not rely on a changing index variable in the termination test. Any logical expression can be used, such as: DO UNTIL STATUS = ‘DONE’. It also may be possible to have complex combinations of termination conditions, either of which could cause loop execution to cease. Consider this example, which will terminate either when I is greater than 100 or when J becomes less than or equal to 30:

D0 I = 1 To 100 BY 10 WHILE J > 30

Calculate something

END

Maybe it will help to see a visual representation of these two looping structures. I'm sure you're familiar with flowcharts, in which action statements are represented with rectangles and decisions with diamonds. Figure la uses a fragment of a flowchart to illustrate that, in a DO WHILE loop, the conditional test is done before the action statements are executed. In Figure 1b, you see that the conditional test for a DO UNTIL is performed after the action statements are executed. I'm always getting these two confused, so maybe these simple diagrams will help.

FIGURE 1A

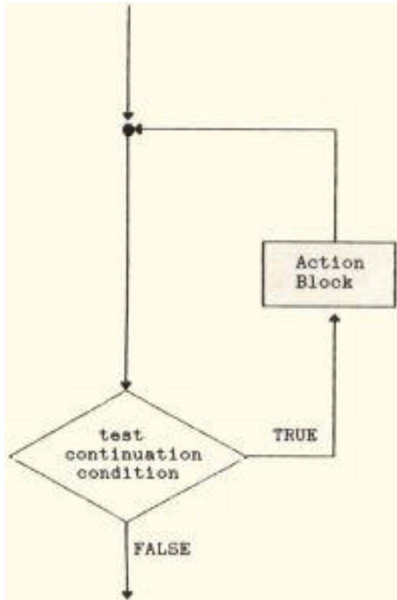


FIGURE 1B

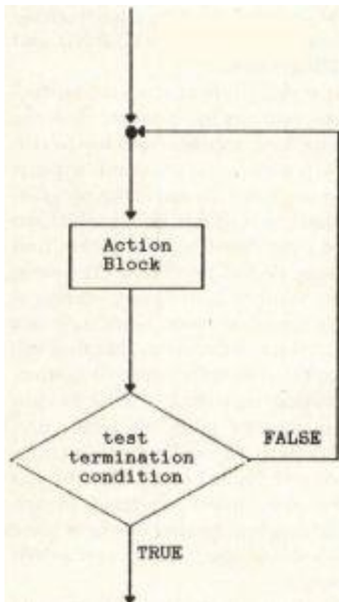
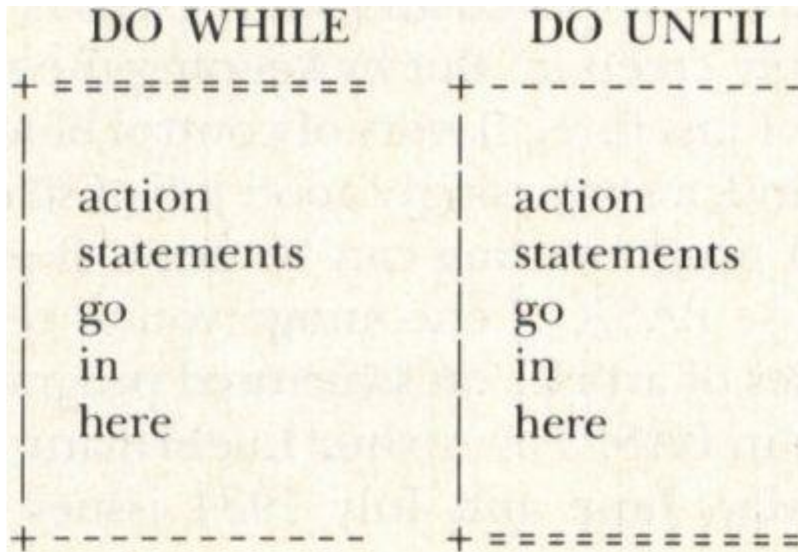


FIGURE 2



Once upon a time we talked about using another diagramming technique to depict the structure of a program module: action diagrams. You may recall that, in an action diagram, a loop is denoted by a double horizontal line, which encloses the action statements in the loop within brackets. You can distinguish between DO WHILE and DO UNTIL logic by placing the double horizontal line at the top or the bottom of the loop bracket, respectively.

Figure 2 shows what you get.

There are times when you want execution of a loop to end prematurely, for a variety of reasons. Bad input data may have been encountered, some error condition may have cropped up or whatever. Remember that we want to write our code so that each control block has only one logical exit point, so let's resist the impulse to GOTO out of the loop. A DO WHILE is a good approach, because you can put the anticipated error or exit conditions into the WHILE clause:

DO I = 1 TO 10 WHILE EOF = ‘FALSE’ AND INPUT = ‘OK’

But things can get tricky if you don't have a DO WHILE available; you may just have to violate the one-exit rule. Some languages provide a command like LEAVE for premature loop exits. In BASIC, you can set the index variable to its termination condition, so that the loop isn't executed anymore; FORTRAN won't let you do this. A

more graceful solution is to use an IF block that tests for the abnormal termination condition right inside your DO block:

```
DO I = 1 TO 10
  IF EOF = 'FALSE' AND INPUT = 'OK' THEN
    DO
      Calculate something
    END
  END IF
END
```

This is basically a way to simulate the DO WHILE command with your own explicit test. Note the innermost DO/END block without any loop criteria on the DO statement. This is really defining an action block to be executed within the selection block (IF/END IF), not a loop block. If you prefer, you can think of this as a loop block that gets executed exactly one time.

Let me out!

The one-entry, one-exit concept applies to individual modules just as it does to loops. Suppose you're calling a subroutine from a main program. You should always enter the subroutine at its top and exit at its bottom. This seems straightforward enough, but there are several ways things can go awry.

Some languages permit multiple entry points to a called procedure, but this is a bad practice. For example, in BASIC you may have seen the use of GOSUB in this form:

```
10 IF A = 1 THEN GOSUB 1000
20 IF A = 2 THEN GOSUB 2000
30 END
1000 Calculate something
2000 calculate something else
2010 RETURN
```

Essentially what we have here is one subroutine (Lines 1000-2010) with two entry points (Line 1000 or Line 2000). If A = 1, then control is passed to Statement 1000, it drops through to Statement 2000, then returns at Statement 2010. But if A = 2, only the Lines 2000 and

2010 are executed. Concise, but very confusing; don't do it. If it's necessary to selectively execute certain statements in the subroutine under only certain conditions, put a test for those conditions at the top of the subroutine, or (even better) write two separate subroutines.

Similarly, each subroutine should have only one exit point or RETURN statement. But now something of a dilemma crops up. Suppose that we're being good little software engineers and are programming very defensively, writing lots of code at the top of the subroutine to validate ("audit") the input data. If errors in the data are encountered, we want to abort the operations in the subroutine and return to the calling program with some indication of what went wrong. How can we handle this if we only want to have one exit point in the subroutine? I can think of three alternatives.

1. Forget about the one-exit rule and have multiple RETURN statements in the subroutine, one for each failed audit plus the proper one at the very end.
2. Forget about the no GOTO rule and have a GOTO after each failed audit, which transfers control to the single RETURN statement at the end of the subroutine.
3. Nest our audit tests (which are probably IF statements) as deep as necessary so that only conditions which pass all the tests finally work their way to the inner sanctum where the actual subroutine code resides.

It seems that we have a real quandary here. All of these approaches seem to violate one of the structured programming rules we're trying to abide by. Actually, the third scenario is okay if we have only a couple of audit conditions, because nesting IF statements two or three levels deep is perfectly fine. More levels of nesting than that becomes a problem both in terms of readability (do you want to match up six or seven END IF statements with the corresponding IFs on previous screens?) and because, if you're indenting nicely at each IF level, you may wind up with awfully short lines available for the actual subroutine code in the middle of the IFs. Not good.

My preference in handling the validation question is to think of the blocks of audit code as a "filter" through which the data must pass in order to wend its way to the heart of the module. I tend to put each audit block at the same level of indentation and have a RETURN

statement in each block. The main processing section of the procedure has only a single RETURN statement.

Using the filter analogy, any time a piece of bad data is encountered; the RETURN statement acts as a block that prevents the process from continuing. I find that this approach keeps the code easy to read and modify, although I am violating the one exit rule. The next best solution would be to GOTO the single RETURN statement each time an audit fails, but I find this approach to be a little harder to read.

It might be useful to let the calling program know why processing was halted in a subroutine. The calling program will take different actions depending on the cause of the error, be it bad input data, mathematical impossibilities in the computations, file I/O problems or whatever. One way to do this is to pass a return code back to the calling program. The numeric value of the return code might represent the sequence number for an error message to display to the user. The interpretation of the error numbers could rely on a table or external file of messages or actions to match the return codes.

Alternatively, the return code might actually be the name of a procedure to call for handling the error or otherwise transferring execution control. By convention, a return code of 0 usually indicates that no error was encountered.

Wrapping up

The structured programming concepts we've talked about here are all geared toward making the programs you write more understandable to human beings. The computer doesn't care if your program makes sense, so long as it compiles properly. But anyone who must understand how your program works needs all the help he or she can get. You can provide that help not only by using good structured programming practices, but also by using common sense. Keeping your code clear and simple, rather than cute, condensed or clever, will go a long way toward writing programs that are easy to read, comprehend and alter.

Programming style is a set of rules or guidelines used when writing the source code for a computer program. It is often claimed that following a particular programming style will help programmers to read and understand source code conforming to the style, and help to avoid introducing errors.

ning style used in a particular program may be derived from the **coding standards** or **code conventions** of a company or other computing organization, as well as the preferences of the author of the code. Programming styles are often designed for a specific programming language (or language family): style considered good in C source code may not be appropriate for BASIC source code, and so on. However, some rules are commonly applied to many languages.

Elements of good style

Good style is a subjective matter, and is difficult to define. However, there are several elements common to a large number of programming styles. The issues usually considered as part of programming style include the layout of the source code, including indentation; the use of white space around operators and keywords; the capitalization or otherwise of keywords and variable names; the style and spelling of user-defined identifiers, such as function, procedure and variable names; the use and style of comments; and the use or avoidance of particular programming constructs (such as GOTO statements).

Code appearance

Programming styles commonly deal with the visual appearance of source code, with the goal of requiring less human cognitive effort to extract information about the program. Software has long been available that formats source code automatically, leaving coders to concentrate on naming, logic, and higher techniques. As a practical point, using a computer to format source code saves time, and it is possible to then enforce company-wide standards without debates.

Indentation

Indent styles assist in identifying control flow and blocks of code. In some programming languages indentation is used to delimit logical blocks of code, correct indentation in these cases is more than a matter of style. In other languages indentation and whitespace does not affect function, although logical and consistent indentation makes code more readable. Compare:

```
if (hours < 24 && minutes < 60 && seconds < 60)
{
   return true;
}
else
{
   return false;
}
```

or

```
if (hours < 24 && minutes < 60 && seconds < 60) {
   return true;
} else {
   return false;
}
```

with something like

```
if  (    hours<
24  && minutes<
60  && seconds<
60  )
{return    true
;}         else
{return   false
;}
```

The first two examples are probably much easier to read because they are indented in an established way (a "hanging paragraph" style). This indentation style is especially useful when dealing with multiple nested constructs.

Python uses indentation to indicate control structures, so correct indentation is required. By doing this, the need for bracketing with curly braces ({ and }) is eliminated. On the other hand copying and pasting Python code can lead to problems, because the indentation level of the pasted code may not be the same as the indentation level of the current line. Such reformatting is tedious to do by hand, but some text editors and IDEs have features to do it automatically. There are also problems when Python code could be rendered unusable when posted on a forum or web page that removes whitespace, though this problem can be avoided where it is possible to enclose code in whitespace-preserving tags such as "<pre> ... </pre>" (for HTML), "[code]" ... "[/code]" (for bbcode), etc.

Haskell similarly has the off-side rule which lets indentation define blocks; however, unlike in Python, indentation is not compulsory in Haskell — curly braces and semicolons can be (and occasionally are) used instead.

Vertical alignment

It is often helpful to align similar elements vertically, to make typo-generated bugs more obvious. Compare:

```
$search = array('a', 'b', 'c', 'd', 'e');
$replacement = array('foo', 'bar', 'baz', 'quux');

// Another example:

$value = 0;
$anothervalue = 1;
$yetanothervalue = 2;
```

with:

```
$search      = array('a',   'b',   'c',   'd',   'e');
$replacement = array('foo', 'bar', 'baz', 'quux');

// Another example:

         $value = 0;
  $anothervalue = 1;
$yetanothervalue = 2;
```

The latter example makes two things intuitively clear that were not clear in the former:

- the search and replace terms are related and match up: they are not discrete variables;
- there is one more search term than there are replacement terms. If this is a bug, it is now more likely to be spotted.

Arguments against vertical alignment are

- **Inter-line false dependencies**; tabular formatting creates dependencies across lines. For example, if an identifier with a long name is added to a tabular layout, the column width may have to be increased to accommodate it. This forces a bigger change to the source code than necessary, and the essential change may be lost in the noise. This is detrimental to source code control where inspecting differences between versions is essential.

- **Brittleness**; if a programmer does not neatly format the table when making a change; maybe legitimately with the previous point in mind; the result becomes a mess that deteriorates with further such changes.

- **Resistance to change**; tabular formatting requires more effort to maintain. This may put off a programmer from making a beneficial change, such as adding, correcting or improving the name of an identifier, because it will mess up the formatting.

- **Reliance on mono-spaced font**; tabular formatting assumes that the editor uses a fixed-width font. Most modern code editors support proportional fonts, and the programmer may prefer to use a proportional font for readability.
- **Tool dependence**; some of the effort of maintaining alignment can be alleviated by tools (e.g. a source code editor that supports elastic tabstops), although that creates a reliance on tools.

Coding Style

'Coding style' refers to the way source code is formatted. For C, this involves things like brace placement, indentation, and the way

parentheses are used. GNOME has a mix of coding styles, and we do not enforce any one of them. The most important thing is for the code to be consistent within a program or library - code with sloppy formatting is not acceptable, since it is hard to read.

When writing a new program or library, please follow a consistent style of brace placement and indentation. If you do not have any personal preference for a style, we recommend the Linux kernel coding style, or the GNU coding style.

Read the (Standards)Writing C info node in the GNU documentation. Then, get the Linux kernel sources and read the file linux/Documentation/CodingStyle, and ignore Linus's jokes. These two documents will give you a good idea of what we recommend for GNOME code.

Indentation Style

For core GNOME code we prefer the Linux kernel indentation style. Use 8-space tabs for indentation.

Using 8-space tabs for indentation provides a number of benefits. It makes the code easier to read, since the indentation is clearly marked. It also helps you keep your code honest by forcing you to split functions into more modular and well-defined chunks - if your indentation goes too far to the right, then it means your function is designed badly and you should split it to make it more modular or re-think it.

8-space tabs for indentation also helps you to design functions that fit nicely in a single screen, which means that people can understand the code without having to scroll back and forth in order to understand it.

If you use Emacs, then you can select the Linux kernel indentation style by including this in your .emacs file:

```
(add-hook 'c-mode-common-hook
          (lambda ()
            (c-set-style "k&r")
            (setq c-basic-offset 8)))
```

On newer Emacsen or with a newer cc-mode, you may be able to simply do this instead:

```
(add-hook 'c-mode-common-hook
        (lambda ()
          (c-set-style "linux")))
```

If you use vim, then you can select the GNOME kernel indentation style by including this fragment in your ~/.vimrc file:

```
set ts=8
if !exists("autocommands_loaded")
  let autocommands_loaded = 1
  augroup C
      autocmd BufRead *.c set cindent
  augroup END
endif
```

The GNU indentation style is the default for Emacs, so you do not need to put anything in your .emacs to enable it. If you wish to select it explicitly, substitute "gnu" for "linux" in the example above.

If you know how to customize indentation styles in other popular editors, please tell us about it so that we can expand this document.

Naming Conventions

It is important to follow a good naming convention for the symbols in your programs. This is especially important for libraries, since they should not pollute the global namespace - it is very annoying when a library has sloppily-named symbols that clash with names you may want to use in your programs.

Function names should be of the form module_submodule_operation, for example, gnome_canvas_set_scroll_region or gnome_mime_get_keys. This naming convention eliminates inter-module clashes of symbol names. This is very important for libraries.

Symbols should have descriptive names. As Linus says, do not use cntusr(), use count_active_users() instead. This makes code very easy to read and almost self-documenting.

Try to use the same naming conventions as in GTK+ and the GNOME libraries:

- Function names are lowercase, with underscores to separate words, like this: gnome_canvas_set_scroll_region(), gnome_mime_get_keys().
- Macros and enumerations are uppercase, with underscores to separate words, like this: GNOMEUIINFO_SUBTREE() for a macro, and GNOME_INTERACT_NONE for an enumeration value.
- Types and structure names are mixed upper and lowercase, like this: GnomeCanvasItem, GnomeIconList.

Using underscores to separate words makes the code less cramped and easier to edit, since you can use your editor's word commands to navigate quickly.

If you are writing a library, then you may need to have exported symbols that are to be used only within the library. For example, two of the object files that compose the library libfoo.so may need to access symbols from each other, but this symbols are not meant to be used from user programs. In that case, put an underscore before the function name and make the first words follow the standard module/submodule convention. For example, you could have a function called _foo_internal_frobnicate().

Consistency in Naming

It is important that your variables be consistently named. For example, a module that does a list manipulation may choose to name the variables that hold a list pointer "l", for terseness and simplicity. However, it is important that a module that manipulates widgets and sizes does not use variables called "w" for both widgets and widths (as in width/height values); this would make the code inconsistent and harder to read.

Of course, these very short and terse names should only be used for the local variables of functions. Never call a global variable "x"; use a longer name that tells what it does.

Standards and Guidelines

Documentation guidelines

In addition to a working program and its source code, you must also author the documents discussed below to gain full credit for the programming project. The fundamental structure of these documents is entirely independent of project, programming language, and operating system. You will find a number of advantages when you pursue a rigid documentation approach to programming. First of all, you will have a firm understanding of the task at hand before you start coding. A good understand of the problem leads to a clean design that tends to have fewer bugs. Always make your goal to program it right the first time! The next advantage is that others will be able to use your documentation to test the program, fix bugs, and make enhancements. In the corporate world, these duties are normally performed by different people and often by different groups within a single company. Therefore, the more detailed, organized, and
easy-to-read your documentation is, the more you help other people do their jobs. As you learn to write solid documentation, you will also come to appreciate reading solid documentation, and will eventually detest reading technical crap (the world is full of poorly written technical books and manuals). In other words, write simply and clearly. The way you write is just as important as the details you present. Always strive to spell correctly and use proper grammar. The campus Writing Center can aid you in this respect.

- User Requirements Document (URD)
- Requirements Analysis Document (RAD)
- User Interface Specification (UIS)
- Prototype
- Object Oriented Analysis (OOA) or High Level Design (HLD)
- Object Oriented Design (OOD) or Low Level Design (LLD)
- Code Documentation (CD)
- Testing Documentation (TD)

- User's Guide (UG)

User Requirements Document (URD)

This document describes the problem from the user's point of view. It briefly describes the problem domain, e.g.. a psychology experiment or a small business accounting package. Then the document delivers a simple and exact description of the problem. After the problem description, the user states exactly what he/she would like the software system to do. While this may seem to indicate a user interface, it is better to focus on the tasks to be solved rather than the interface required to solve them. However the user may require a specific interface, e.g.. a GUI rather than a command line interface. The crux of this document is to identify what the user requires of the program, and not what the user requires of the programmer. This document furnishes the programmer with a formal description of the problem. Ideally, this document would be written by someone in marketing who has talked with a customer; not by a programmer. The most important thing to remember is that this document describes the functionality required of the program.

The URD has:

- User’s view of the problem
- Brief description of the problem domain
- Complete description of the problem
- What is expected from a software solution
- What is not expected from a software solution

The URD does not have:

- Programmer’s point of view
- Programming jargon or technical details
- Description of programming languages or environments unless it is a specific user requirement
-
- Description of the solution. This is not a design document. We only want requirements here.

Requirements Analysis Document (RAD)

This document takes the URD as a starting point and looks at the problem from a designer's point of view. However, instead of diving directly to implementation details, the analysis focuses on the system and software requirements needed to implement the user requirements. This document gets detailed, but does not delve into programming details. Instead, take the user's requirements and clearly identify all of the details and mitigating factors that will affect the solution that the user wants. An analysis may indicate a preference for a particular programming language that best suits the problem domain rather than an algorithm to satisfy a particular requirement. The RAD looks at the URD as defining an entire system, and then breaks the URD down into bite-size chunks (divide and conquer). These chunks identify the subsystems of the overall solution, and the relationships between them. But the RAD also goes further and identifies the actual details of the problem that the user may not be aware of.
The RAD also maps the domain of software systems onto the user requirements. For example, the RAD may indicate that a database is needed for a particular subsystem, or that an expert system can satisfy certain other requirements. The RAD is written from the designer's perspective. An astute software designer is one who is aware of available software systems and paradigms. He/she should know what types of systems and solutions work best in different environments. The RAD, then, identifies the software systems and paradigms that will best fit the user requirements. The RAD doesn't design a solution; it merely identifies the most beneficial means for an implementation.

The RAD has:

- designer's interpretation of the user's requirements: identify the "real" problem(s)
- breakdown the problem into high level constituent parts
- deep analysis of these parts and identification of all relevant details
- identify existing solutions
- identify alternative technical solutions

- link these solutions to the problem(s), especially with respect to details
- suggest the best solution and break it into parts
- devise ways to test the solution

The RAD does not have:

- user's point of view
- implementation details
- algorithms
- user interface specification

User Interface Specification (UIS)

This document describes exactly what the user interface is going to do, what it looks like, and how the user interacts with the program. The UIS does not describe how the interface is implemented. Nor does it describe what the program does behind the interface. Rather, the UIS focuses in detail specifically on the user interface itself. For a GUI, the UIS would define the components and all options on a MenuBar, all of the MenuBar headings, all of the submenus, and all of the options of those submenus. The UIS may describe the functionality of each of the mouse buttons, if appropriate. The UIS also describes the presentation of data, be it graphics, text, or a combination. The UIS should be understandable to the user. The UIS should contain drawings or screen captures
of prototype interaces. For non-GUI applications, the user interface may be either a Command Line Interface (CLI) for which the UIS could be similar to a UNIX man page, or could be an Application Programmatic Interface (API). An API is really nothing more than a collection of library routines that allow you to link and/or layer software components. For an API, the UIS consists of a definition of the calling interfaces, names of routines or object methods, parameters, and descriptions of what they will do.
The UIS has:
Description of the complete user interface, CLI, or API
what the user interface looks like to the user
how the user interface behaves
how the user interacts with the system
if GUI, names for all of the interactive components, from the mouse to buttons to menus and scrollbars, and pictures of what the interface should look like
if CLI, flags and arguments, inputs and outputs as in a UNIX man page

if API, complete description of the link/call interface
The UIS does not have:
implementation details coverage of the mechanics beneath the user interface

Prototype

Once you know what the user interface (GUI, CLI, or API) is going to look like, go ahead and try to build a "shell" of the user interface as quickly as possible. This is called a prototype. You don't have to implement the core functionality - you just code the interface to see 1) if it's possible and 2) how it appears to the user. Ideally, you should end up throwing this prototype away. The prototype should be shown to the user to see if it's what the user had in mind. That's why you really shouldn't put too much effort into it because chances are the user is going to say, "No, I wanted it to look like this" or
Software Documentation Guidelines "I like this screen, but that one sucks", etc. You want to ensure that the user sees what they want to see - and this applies to GUI, CLI, and API. It's much easier to change a prototype than it is to do all the work on the internals first and then have to change things. The prototype helps you design things in such a way that the interface is independent of the underlying implementation of the actual problem. The interface is not the solution to the problem. But the interface is the user's window to that functionality. That functionality, or solution to the problem, begins with the next step.

Object Oriented Analysis (OOA) or High Level Design

The OOA applies an object-oriented view to the problem. The easiest way to do this is to pick out all of the nouns in the RAD. Each noun is usually an object. Through out nouns that aren't substantive objects. The next step is to write a detailed description of each object, no matter how trivial it may seem or how much you take this object for granted. Each object must be completely and succinctly documented. This is called a data dictionary. Next, look for overlap between objects and remove objects that are not important to the problem domain. For example, take out user interface components. You will deal with those separately. Once you have removed unnecessary objects, identify their attributes and methods. More often than not, you will find that some of the objects you have are merely attributes of other objects. Next, establish relationships between objects. For example, an employee works for a company. Here, both

company and individual are objects. An individual works for a company. A company employs an individual. A company has many workers. An individual usually works for a single company. Here, we have both defining and numerical
(cardinality) relationships. Find such relationships between your objects, and define the cardinality, eg. one-to-one, one-to-many, many-to-many.
The next step of the OOA is to do the same thing with your UIS. The user interface and underlying application subsystems should be completely independent of one another. In fact, you should be able to design and develop your interface and your underlying application independently of one another. You should make your user interface classes as generic as possible, and subclass off of them to get application-specific behavior. For example, to make a list of colors you would probably use a generic List object. A List has a series of text string labels for each choice, lets the user make a choice, activates some function when such a choice is made, allows the user to add an item to the list, delete one, and so on. So far, you could use this List object in almost any application that requires a List
interface object. However, for a color selector, you may want to show the actual color in the rectangular slot where you usually show the textual name of the color. In this case, design the object hierarchy such that most of the functionality is encapsulated in the List object, but then subclass a Color List class to present the colors rather than the textual labels. Design interactive objects in a similar way: allow an application to register actions with an object rather than defining that an object performs specific tasks. Most of the appearance and functionality is then abstracted away from the end user in a particular program. This way you can share an object between applications, and assign application-specific behavior with registration rather than explicit coding. If you are not using an object oriented programming language like C++ or Java, you can still use object oriented design techniques. However, if you prefer, you can write a High Level Design document in place of the OOA. The HLD accomplishes many of the same goals, but from a non-OO approach.
The OOA has:
data dictionary defining exactly what each object represents (in English) class diagrams showing the name, attributes, and methods of each class relationships between the classes, depicting graphically and textually a set of class diagrams and data dictionary for the application domain a set of class diagrams and data dictionary for the interface domain end-user readability

The HLD has:
detailed breakdown of technical solution in subsystems descriptions of data structures and operations required for each subsystem detailed interaction between subsystems, including interface subsystem(s)
The OOA and HLD do not have:
algorithms
implementation details

Object Oriented Design (OOD) or Low Level Design (LLD)

The OOD is as close to coding as you can get without actually coding. If you do this document correctly, the code will just fall out naturally. The OOD takes the classes in the OOA a level deeper into the realm of pseudo-code. The OOD defines the datatypes for the attributes. The OOD also defines the algorithms and implementation details of the class methods. However, you are not writing code yet. The OOD should be language-independent. You are merely specifying more exactly what the attributes and methods consist of. Like the OOA, the OOD addresses both the interface and application class hierarchies. Also like the OOA, it keeps the two domains separate and independent of one another.
The OOD goes another step beyond the OOA by identifying the objects, rather than the classes, required to implement the software system. While the user may be able to comprehend the OOA, the OOD goes very deep into software design. The OOD is much more detailed than the OOA and establishes instantiations from the class hierarchy and their relationships to one another, especially associations and cardinality. The OOD also provides the algorithms for all class/object methods pseudo-code. For a graphical
environment, the OOD would specify how the event loop dispatches events to specific objects. For example, a mouse-down in a drawing canvas would activate the pen object to start drawing, while a mouse-up would terminate the line. The OOD often makes use of state and event diagrams that define exactly what happens when the user interacts with the graphical components. It maps the user interactions with graphical components to underlying application subsystems. For the drawing example, the OOD maps the mouse down to the pen object that draws the line. The mouse doesn't draw the line, the pen object does, using the mouse coordinates as a guide. For non-OO design, the LLD contains essentially the same contents as the OOD, namely, explicit detail of all data types and functions. You should provide psuedo-code for all algorithms and flesh out all aspects of the

programming effort without yet resorting to actual code. If you were to compare this documentation process to writing a research paper, you could regard the HLD as the chapter and section titles, the LLD as the bullet items for each section, and the code as the text of the paper. Each document just gets more and more detailed.
The OOD and LLD have: inner details of class attributes (data types) and methods (functions) detailed object (as opposed to class) diagrams for OOD

- State diagrams
- Event diagrams
- Pseudo-code
- Algorithmic descriptions

The OOD and LLD do not have: Code

Code Documentation (CD)

You are expected to fully document your code. Every class and class method should have a name, a brief one-line description, and a detailed description of the algorithm. All methods also require descriptions of all inputs and outputs. If applicable, you should also note any caveats - things that could go wrong or things that the code doesn't address. Put assumptions in the caveats section. If you are coding in Java, you should use the documentation tags that work with the javadoc utility. This utility automatically generates web pages for your documentation. To make things consistent, simply cut and paste the textual descriptions of your classes, objects, and methods from your OOD directly into the code. Then let javadoc do the dirty work. If you are not coding in Java, you can still use the same tags and see if javadoc operates on your source files. Otherwise, you could write such a utility yourself!

Testing Documentation (TD)

The TD describes how you tested your program to prove that it works successfully. You should include testbeds for both the user interface and application aspects of your program. You should also provide test datasets where applicable. For example, if you program does some form of text processing, you should provide example file inputs that test specific features of the program requirements. You should pay special attention to borderline values and bogus user input. The

TD should also include evaluation criteria, so you know what you are actually testing for.

User's Guide (UG)

This document tells a user how to use your system. The format is up to you. Pick your favorite User's Guide and copy its format. You may even want to use the UNIX man page system. Assume that the user knows nothing about your project. Provide an overview and then details of each subsystem. Make sure you explain how the whole system works before delving into the details. This document is for the end-user or another programmer, so you have to explain the obvious, e.g..Basically, you have to tell someone how to use your system.

Type Checking

Scoping Rules
What is a "normal" scoping rule? In many languages (AWK, Snobol, Perl, etc)
a variable is global unless declared local. Most other languages (C, C++,
Pascal and its heirs, Haskell, Lisp, etc) do need global and local

declarations, but we usually only care for the locals when reading a program. In C, for instance, when you read inside a function something like

```
  varname = 1;
```

you do not go all over the file (and includes) to see whether "varname" is
declared as global. You just check if varname is declared local; if it's not, then it is a global variable. A compiler does the same, and only *after* seeing that the variable is not local it will need a global declaration.

Very few languages do create local variables without some kind of explicit
declarations. This is because local variables are local to a given scope,
and so we must specify this scope. If we get the rule that "a variable is
local unless defined globally", we will have some weird behaviours; for
instance, in Lua (and also in C and Java) the scope of a variable is not a
whole function, but the inest block where the variable is declared. So, if
you write something like

```
  if a then i=1 else ... end
```

the scope of the variable 'i' (now local by default) would be only the "then" part of the if!! (the inner block...)

In Lua you do not need global declarations. You can complain about this
(and many people do); but to assume that a variable without this global
declaration should be automatically considered local does not look "normal"
to most languages, and certainly is not the right way to solve the problem
with undeclared global variables.

If the real problem is the use of global variables without declaration,
there is an easy and efficient way to solve that, using tag methods. The
idea is to set the tag methods 'get global' and 'set global' over the tag nil
(which is the value of any "undeclared" variable), and to keep a table with
the name of all declared global variables. When you read or write a nil
global variable, the tag method goes to the table to check whether the
variable has been declared. When you read or write a non-nil global, the
most frequent case, there is no cost at all.

```
=================================================================
_Globals = {}   -- a set to keep the names of global variables

function declare (globalname)
  _Globals[globalname] = 1
end

function undeclare (globalname)   -- optional
  rawsetglobal(globalname, nil)
  _Globals[globalname] = nil
end


settagmethod(tag(nil), 'setglobal', function (name, oldvalue, newvalue)
  if not _Globals[name] then
    error("cannot write undeclared variable `"..name.."'")
  end
  return rawsetglobal(name, newvalue)
end)

settagmethod(tag(nil), 'getglobal', function (name, value)
  if not _Globals[name] then
    error("cannot read undeclared variable `"..name.."'")
  end
  return value   -- or nil (value must be nil...)
end)
```

==

Example:

```
> dofile'temp'   -- the above code
> print(i)
lua: cannot read undeclared variable `i'
Active Stack:
       function error [in file (C)]
       `getglobal' tag method [in file temp]
       main of (dostring) >> "print(i)"
> declare'i'
> print(i)
nil
> i=10
> print(i)
10
> a = 10
lua: cannot write undeclared variable `a'
Active Stack:
       function error [in file (C)]
       `setglobal' tag method [in file temp]
       main of (dostring) >> "a = 10"
> undeclare'i'
> i='hello'
lua: cannot write undeclared variable `i'
Active Stack:
       function error [in file (C)]
       `setglobal' tag method [in file temp]
       main of (dostring) >> "i='hello'"
```

The scoping rules of a Programing Language dictate how FreeVariables - symbol names which are found in the body of a scope (a function, macro, class, whatever) but not defined there - are resolved.

Several different strategies exist:

- Disallowing them altogether. Very rare; though some languages may disallow free references to all but a pre-defined set of symbols (often called keywords or special forms) which are provided by the language.

- Only two scoping levels - global and local. C Language and many assemblers use this rule. A Free Variable defined in a C function must refer to a "global" symbol (meaning defined either at file or global scope; as opposed to within a function body). This simplifies the implementation of C greatly (and many C programmers don't miss more advanced scope rules at all). Cee Plus Plus sticks to the C tradition in many ways, though the presence of features such as classes and namespaces cause C++ to relax the scoping rules quite a bit. (Still, no C++ function can bind to the local variables of another C++ function). Java, with InnerClasses, relaxes the scoping rules further.
- LexicalScoping. Used by CommonLisp, SchemeLanguage, AlgolLanguage, PascalLanguage, and many others. If a variable isn't found in a given scope, the enclosing scope is searched; repeating until the outermost scope is reached. Two important variants of this are DeepBinding and ShallowBinding. With DeepBinding, variables are bound to the environment of where the function is defined; with shallow binding variables are bound to the environment of where the function is called. Most languages which support LexicalScoping support DeepBinding for functions; most macro systems (excluding Scheme's DefineSyntax) use shallow binding. It is possible to divide DeepBinding further into two separate forms (I am not aware of any generally accepted terminology to describe these forms). In one form, a FreeVariable essentially is an alias for the actual variable in the enclosing scope; if that variable changes than the value of the variable in the function being considered also changes. (If the enclosing scope has exited, possible if FirstClass functions are mixed in with LexicalScoping, then the value the variable had at exit changes). Most languages which support DeepBinding support this form. In the other form; the FreeVariable takes the value that it has at the point when the function in question is first defined; and does not change. (In other words, the function using the FreeVariable makes a copy of the value provided by the enclosing scope). I'm not aware of any languages which do this for FreeVariables?; though objects used as closures have this behavior. Java InnerClasses (when defined within a function) sidestep this issue by only allowing references to variables in the enclosing function which are declared to be final - in other words, those whose value does not change.
- DynamicScoping (early dialects of Lisp, CommonLisp special variables, exported environment variables in UnixOs): The caller is checked for a binding for the variable; if one is found, it is used. Otherwise, the caller's caller is checked, and so on. If no definition is

found, it is either an error or a default value is used, depending on the semantics of the language.

The following C-like program illustrates the different scoping rules. Apply a different rule, that's that what would print. Of course, this is not legal C in real life, as CeeLanguage doesn't allow nested functions.

```
int main (void)
{
    const char *scope = "Lexical, deep, by copy ";

    void print_scope (void)
    {
        printf ("%s\n", scope);
    }

    scope = "Lexical, deep, aliasing";

    void (*)(void) helper_func (void) /* Returns ptr to function;
pretend its a closure */
    {
        const char *scope = "Lexical, shallow scoping";

        void do_print_scope (void)
        {
            print_scope();
        }
        return do_print_scope;
    }

    void do_it (void)
    {
        const char *scope = "Dynamic Scoping";
        helper_func()(); /* Call the function returned by helper_func();
*/
    }
    do_it();  /* Print what scoping we are using */
}
```

Another example, in pseudo PascalLanguage: (from 'Compilers: Principles, Techniques, and Tools')

```
program scoping;
```

```
var r : string;
procedure show;
  begin
  writeln(r);
  end;
procedure scope;
  var r : string;
  begin
  r:='Dynamic';
  show;
  end;
begin
r:='Scope';
show;
r:='Lexical';
scope;
end.
```

Note: C does use LexicalScoping but does not allow function definitions to be nested (Standard C doesn't, but GNU C does). C++ also uses LexicalScoping. Namespaces and classes are just lexical scopes, like any other, as are Java InnerClasses.

For example:

```
int func() {
    int outer_local = 1;

    { // this block introduces a new lexical scope
        int inner_local = 2;

        printf( "%i %i\n", inner_local, outer_local );
    }

    // inner_local is no longer in scope.
}
```

Will print:

```
2 1
```

The important point about C/C++ is that it doesn't allow FreeVariables? in one function to refer to anything defined in another function; this eliminates the need for a StaticChain and/or closures.

So this perfectly valid ISO 9899:1999 does not use LexicalScoping?

```
int main() {

  for (int i = 0; i < 10; i++) {
    dosomething(i);
    }
  // i is no longer accessible.
}
```

This is not a GNU extension, this is valid C99. AFAIK the standard does say that blocks group a set of declarations and statements into a syntactic unit, and a compound statement is a block.

Actually, C++ does allow nested functions; you just have to spell the internal function differently. The following is not legal in C++:

```
void outer (int x)
{
    void inner (int y);
    {
        cout << y+1;
    }

    inner(x);
}
```

The following, however, is.

```
void outer (int x)
{
     class { void operator () (int y)
     {
        cout << y+1;
     } inner;

     inner(x);
}
```

In other words, functors are a great way of faking nesting functions. Of course, the functor (still) has no access to the variables defined in outer.

Concurrency Mechanisms.

Software engineering is a team effort. Collaboration refers to the ability of team members to work together to reach a common goal.

Collaboration involves communication and sharing of ideas, plans, and progress on an individual's activities. The exact nature of this communication and sharing can vary greatly. For example, working out a design on a shared whiteboard is one form of collaboration. Another form involves one individual taking primary responsibility for an activity and requesting help from others on particular parts of the activity, such as when writing a paper.

In a software engineering environment, the artifacts that the developers wish to collaborate on can be represented as objects. Collaboration can be supported by controlling access to these objects in a way that allows the collaborators to see and modify the objects but prohibits others from doing so. In this paper, we describe a mechanism that supports a range of collaborative concurrency control capabilities, providing what we believe is needed in software engineering environments.

Current approaches to supporting software engineering teams provide better support for prohibiting access to objects than for allowing access by selected individuals. Many software engineering tools use a check in/checkout mechanism to control access. In these systems, there is a public area in which objects are readable by everyone in the project. An individual checks out an object into a private workspace, where only that individual can see and modify it. When the changes are complete, the individual checks the object back into the public area. To collaborate in this type of environment typically involves working outside of the system. For example, developers might use the operating system's access control mechanisms to create groups of collaborators and change accessibility on individual workspaces so that they become group workspaces. Another approach is to allow multiple developers to checkout the same item and require them to merge their changes when they check the objects back in.

Our approach is to provide an object management infrastructure that supports a very general model of cooperative concurrency control. The goal is to allow individuals to decide which objects they want to work on privately and which objects and with whom they want to collaborate. In developing this infrastructure we focused on four common scenarios in software engineering:

- Co-authoring - Multiple developers modify the same object. This can be a real-time co-authorship as in a shared

whiteboard environment or an off-line co-authorship as in coding the same class.

- Observation - One developer modifies an object while others can see the progress being made, but cannot modify the object. For example, a tester might be able to observe a design under development in order to develop test cases.
- Loaning - One developer gives an object to another to work on for a period of time, but expects to get the object back when the second developer is done. For example, a designer might loan a class design to some reviewers expecting to get the design back with review comments added to it.
- Transferring - One developer gives an object to another but does not expect to get that object back. For example, a developer may get reassigned to a different project and therefore need to transfer responsibility for a collection of objects to another individual.

Individuals manipulate objects through executing programs, referred to as clients. Individuals collaborate with each other by allowing the respective clients that they execute to collaborate.

In our approach, collaboration is object-based. Support for cooperation is presented through a capability-based cooperative concurrency control mechanism. A client has to request a capability for an operation on an object before performing the operation. A client will not be granted the capability if the operation being requested conflicts with the operations on the same object for which capabilities are already held by other clients. To determine if there are conflicts, the cooperative concurrency control mechanism maintains two data structures that are associated with each object.

One data structure is a conflicting operation matrix. Two operations on an object are defined to be conflicting if the order in which they are executed affects the state of the object or the return value of either operation. A conflicting operation matrix specifies whether one operation on an object belonging to a class conflicts with another operation on the same object. This matrix can be used to support a simple read/write locking scheme or a more complex semantics-based locking scheme.

A client, A, cooperates with another client, B, by allowing the cooperating client to concurrently perform conflicting operations on a

shared object. The second data structure represents these allowed conflicting operations on an object and is specified by clients through the use of cooperation declarations. A cooperation declaration made by client A, for a specific object, identifies:

- The client(s) with whom client A wishes to cooperate
- The set of operations on the object that client A allows the cooperating clients to perform even if they conflict with client A's operations.
- Client A's role in the collaboration, which is one of collaborator, loaner, or transferred. As a collaborator, client A can continue to perform all operations that it has capabilities for. As a loaner or transferred, client A can only perform non-conflicting operations. A loaner expects to get the objects back when the borrowing clients are done, whereas a transferred does not.

Client B can now cooperate with client A by requesting a capability for the operations that client A specified in its cooperation declaration. A capability is granted to client B if:

- The request is for an operation on an object that does not conflict with any of that object's operations for which capabilities are held by other clients. Conflicts are determined using the conflicting operation matrix for the class to which the object belongs.
- There are cooperation declarations from the owners of the conflicting capabilities allowing client B to perform the operation.

In the absence of collaboration, this mechanism reduces to a lock-based concurrency control mechanism that would prevent two conflicting operations from being performed concurrently. If the conflicting operation matrix defines conflicts for only read and write operation, it reduces to standard read-write locking.

If client A is a collaborator, client B acquires separate capabilities for the operations specified in the cooperation declaration. In this case, both client A and client B hold capabilities for those operations. If client A is a loaner or a transferred, the capabilities held by client A for those operations are transferred to client B. If client A is a loaner, the capabilities are transferred back to client A when client B

completes operations. If client A is a transferred, the capabilities are released when client B completes it operations.

Let's see how these primitives could be used to support the loan example from earlier. This example is described in terms of the people to make it easy to understand. This would, however, be implemented by making the respective client programs perform the operations attributed to people. Suppose Robin is designing class X and Pat is expected to review it. For simplicity, assume the class is represented as a single object and that standard reader/writer locking provides the default definition for concurrency control. Also, assume that there are currently no capabilities held on the object representing class X. In this case, Robin can acquire a capability to create/edit class X's design since there are no conflicting capabilities. While Robin is doing this, there can be no concurrent readers or writers of the design since they would conflict with the existing capability. To give the design to Pat, Robin would make a cooperation declaration to loan the design to Pat for writing so that Pat could add review comments. Now, Pat can get a write capability even though it conflicts with Robin's write capability. Pat does the review and then returns the object which restores Robin's write capability. If Robin wanted to continue modifying the design while Pat was reviewing it, Robin would have used a co-author declaration instead of a loan declaration.

Most concurrency control mechanisms prevent concurrent access involving conflicting operations. Our approach provides a range of alternatives from one extreme where there is true concurrency/collaboration to more restricted forms of collaboration such as various forms of delegation. We would expect that one might build a layer on top of our mechanisms that hid these primitives and provided a more appropriate mechanism for a specific software engineering process. For example, the concurrency control aspects of an advanced transaction mechanism could be built using these primitives, or a process could be defined in which a manager specified who was responsible for various activities with the capabilities being automatically created and delegated as the process executes. What we have defined here are the primitive mechanisms that allow a wide range of cooperative concurrency control policies to be established.

I need a mechanism to implement the following scenario:

1. two or more threads need to load a given set of values at the same time
2. only one request must be done per value, so if two threads need to load the same subsets, one must wait for the other
3. I don't want to have a lock (or mutex, or another primitive) on each value since they can be potentially too high.

The scenario could be (suppose thread B enters a little bit earlier)

```
        thread A         thread B
values    5, 8, 9, 12       7, 8, 9, 13, 14
request   5,      12       7, 8, 9, 13, 14
waits for   8, 9
                      >>data loaded<<
retrieves   8, 9


        >> data loaded <<
returns   5, 8, 9, 12
```

Which concurrent mechanism should I use for this?

Remember a producer/consumer won't work since thread A and B are not exactly consumers (they are only interested on certain data).

Structured programming (sometimes known as modular programming) is a subset of procedural programming that enforces a logical structure on the program being written to make it more efficient and easier to understand and modify. Certain languages such as Ada, Pascal, and dBASE are designed with features that encourage or enforce a logical program structure.

Structured programming frequently employs a top-down design model, in which developers map out the overall program structure

into separate subsections. A defined function or set of similar functions is coded in a separate module or sub module, which means that code can be loaded into memory more efficiently and that modules can be reused in other programs. After a module has been tested individually, it is then integrated with other modules into the overall program structure.

Program flow follows a simple hierarchical model that employs looping constructs such as "for," "repeat," and "while." Use of the "Go To" statement is discouraged.

Structured programming was first suggested by Corrado Bohm and Guiseppe Jacopini. The two mathematicians demonstrated that any computer program can be written with just three structures: decisions, sequences, and loops. Edsger Dijkstra's subsequent article, Go To Statement Considered Harmful was instrumental in the trend towards structured programming. The most common methodology employed was developed by Dijkstra. In this model (which is often considered to be synonymous with structured programming, although other models exist) the developer separates programs into subsections that each has only one point of access and one point of exit. Almost any language can use structured programming techniques to avoid common pitfalls of unstructured languages. Unstructured programming must rely upon the discipline of the developer to avoid structural problems, and as a consequence may result in poorly organized programs. Most modern procedural languages include features that encourage structured programming. Object-oriented programming (OOP) can be thought of as a type of structured programming, uses structured programming techniques for program flow, and adds more structure for data to the model.

UNIT - IV

Implementations issues: Structured Coding Techniques – Coding Style – Standards and Guidelines – documentation guidelines – Type Checking – Scoping Rules – Concurrency Mechanisms.

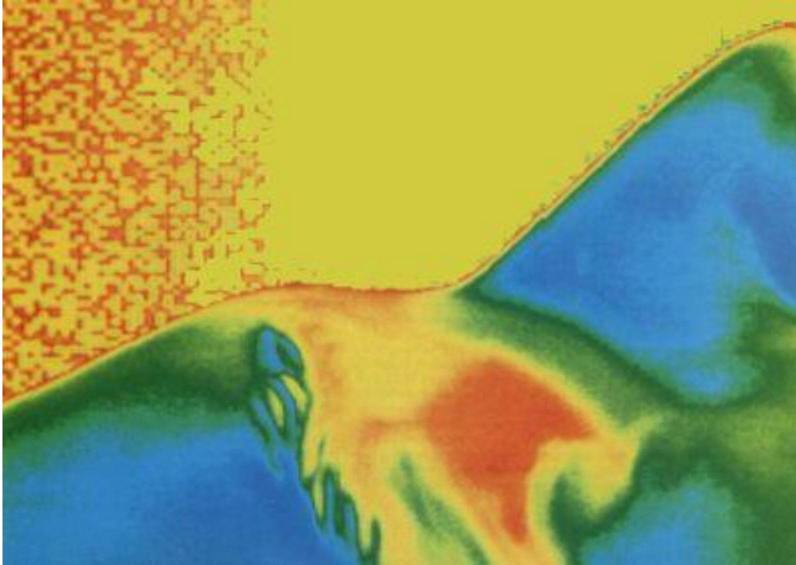
Implementations issues:

Structured Coding Techniques:

Over the past several months, we've talked a lot about concepts that distinguish the discipline of software engineering from the art of computer programming. The underlying theme has been that a disciplined, structured approach to software development will result in a final product of higher quality and greater durability than will a more casual approach. This durability becomes particularly important when your software system enters the maintenance phase, which is everything that you do to the system after you thought you were done with it. Another of our goals is to design and write program modules so that they can be reused in future projects, thereby saving us the pain of continually reinventing all sorts of wheels.

We've talked a lot about how to decompose even a very complex software system into a collection of well-defined, bite-sized processes. If we do this properly, we can go off and write program modules (subroutines, functions, procedures) in our favorite language with confidence that we'll be able to plug the pieces together successfully at the end. Last time, we looked at some ideas for documenting our final product, in the form of both internal comments describing each specific program module and external documentation showing how the system as a whole fits together and operates.

The one aspect of software development we really haven't addressed is the actual writing of the computer program itself. I've been making the assumption that most of you are seasoned programmers, with some experience in modern structured languages such as C, Pascal or GFA BASIC. But perhaps the time is ripe to re-hash a topic that helped begin the software engineering revolution: structured programming.



STRUCTURED • PROGRAMMING

While programming, continually ask yourself,

"How would I like to be the one who has to read, understand and change this program in the future?"

What is structured programming?

It's difficult to provide a concise answer to this question. In some ways, it fits into the "I know it when I see it" category, although we will discuss some specific structured programming principles shortly. But on the other hand, structured programming is consistent with our general philosophy of software engineering, which emphasizes enhanced communication between people. Computer hardware has become powerful enough that programmers should no longer emphasize compactness of code and cleverness of algorithm on their road to the nirvana of "efficiency." Contemporary software development stresses clarity and structure of code, not cuteness. While programming, continually ask yourself, "How would I like to be the one who has to read, understand and change this program in the future?" Structured programming can be a powerful aid to effective software maintenance. And in some ways, structured programming is an attitude, as well as a behavior. The emphasis is on person-to-person efficiency, not computer efficiency.

The golden rule

In my opinion, the single most important idea of structured programming is that the code you write should represent a clear, simple and straightforward solution to the problem at hand. Keep this guiding principle in mind as you read the rest of this article.

Start at the top

One of the basic precepts of structured programming has been with us clear through our software engineering sojourn: a structured program consists of a hierarchical collection of individual modules, which appear more abstract at the top levels and more detailed at the lower levels. This fits with our overall strategy of hierarchical decomposition, which we've followed from structured system specification through structured design and now down to structured programming. The process of building a program in this fashion is called "top-down programming," or step-wise refinement Another point I stressed during our discussion of program design is that each of these modules (or processes, in design terms) communicates with others through well-defined data interfaces. These data interfaces typically are subroutine parameter lists or function argument lists. Each part of the program appears to other parts as a black box that simply performs its assigned function in some unknown way. You’ve encountered this idea every time you called a built-in function in some programming language. Think about 8-bit Atari BASIC. Do you recall the STICK function? It told you something about the position of the joystick. Do you know how to communicate with the STICK function? Yes; all you had to do was pass it the number of the joystick you were interested in, like this: STICK(1), and it returned a numeric answer. Do you know how it worked? No; could be magic, for all you know. Do you care? No. This is the beauty of a "black box" approach to software development. There's no reason why the modules you write should be any different in this regard than the modules supplied by the guys who wrote the language you're using.

In practice, you apply the notion of step-wise refinement by writing your initial description of each fundamental process in a very high-level "language" that we called pseudo code. This is a first attempt at a picture of how each process will accomplish its assigned task. As you continue down the path from design toward code, you add detail

to this description until eventually you reach something that conforms to the exact syntax of the language you're using: source code. I suppose we could consider that one additional step takes place even after this, which is the compiling of your source code into something the computer can deal with: object code. Fortunately, we humans can halt our step-wise refinement at the source code stage and let the machine take over from there. (Someday, we'll be able to stop at the pseudocode stage.)

The dreaded GOTO

The first thing most people learn about structured programming is that you shouldn't use GOTO statements. This notion stems from an article published in 1968 by E. W. Dijkstra, which helped the structured programming push get underway in earnest. The succinct title said it all: "Go To Statement Considered Harmful" (Communications of the ACM, 11 (3), 147-148).

We don't always have a choice. In older forms of BASIC, GOTO statements were needed everywhere, because there just wasn't the richness of commands that we need to avoid GOTO. The careless use of GOTO inevitably leads to the notorious "spaghetti code" that makes a program nearly impossible to comprehend and debug. The worst case is a GOTO that branches back to a previous statement in the program listing.Modern programming languages provide logic and control commands that allow us to almost completely avoid using GOTO statements (we'll discuss these shortly). However, there are still a few situations in which a GOTO actually can result in cleaner, more understandable code. Error-handling situations sometimes benefit from GOTOs. Premature exits from loops, or breaking out of deeply nested IF structures, may be more easily handled with a GOTO than by some other method. Nonetheless, the general guideline that GOTOs should not be used for routine transfer of control within a program is still valid, so try to break any lingering bad habits from your earlier experiences with BASIC interpreters.

A question of style

The programming style you use can greatly influence the readability of your code. While not strictly part of structured programming, there are some matters of style to keep in mind. A pretty good book on this

topic is The Elements of Programming Style, Second Edition, by Kernighan and Plauger (McGraw-Hill, 1978). It's somewhat dated, and the code examples are all in FORTRAN or PL/I, but the principles remain the same. The following represent some of the highlights of programming style in my mind; some don't apply to all programming languages. Use indentation to visually block logically related sections of your code, such as the sections of IF/ELSE IF/ELSE/END IF and SELECT/CASE constructs. Use blank lines in the source code to further delineate sections of the program. Use comments judiciously in the source code for clarification (we talked about this last time), and make sure the comments are accurate. Don't bother to document bad code—rewrite it, instead. Never put more than one statement on the same source line. We always did this in Atari BASIC because it saved six bytes per statement, but you don't need such tricks when you have a megabyte at your disposal.

Select meaningful variable and procedure names. Don't use different names to represent the same piece of information (refer to your data dictionary). Explicitly declare the type of each variable used, if your language permits this. Use parentheses to resolve any ambiguities in mathematical expressions, even if they aren't required for the operation to be executed correctly. Make sure conditional tests (IF some conditions THEN do something) read clearly. Generally speaking, the first condition tested for should be the desired condition, with an error condition handled second.

Program defensively: Try to anticipate all possible errors in input data or mathematical operations, and write code to handle such situations. This includes validating input data before trying to use it and testing for such mathematical problems as division by 0 or taking the logarithm of a negative number. Make sure that input data does not exceed the bounds of what the routine can handle. Think of the user when designing your programs; make input easy to prepare correctly, and make output self-explanatory.

Initialize variables before using them. Who knows what was in those bytes before they were reserved for a variable's use? Avoid multiple entry points, and exits from loops and subroutines (more about this later).

The first priority is to get the program running correctly. You can worry about optimization later. And when you do, make sure the

program still runs correctly. Don't try to optimize every little step; the compiler will do a lot of this for you. Usually, a program spends most of its time in a small section of the code, so concentrate your optimization efforts here (if you can find it).

Use the best algorithms you can find for calculations, but remember that both the algorithm and the structure of your data will influence how the algorithm will be implemented in code. Insert "instrumentation" checkpoints in your programs to write out intermediate results someplace; so that you can verify accuracy, track down errors and assess efficiency. These outputs can be sent to a trace file on the disk, which you can then examine at your leisure.

Building blocks

Another basic premise of structured programming states that any program logic, no matter how complex, can be expressed in terms of just three kinds of logical operations: sequence, selection and iteration. A program then is made up of a series of blocks of code to perform these operations. Let's define these three kinds of operations.

Sequence—a series of program statements are executed one after the other, in the order in which they appear in the source code. Obviously, this rules out statements like GOTO and IF, restricting us just to statements that perform some specific action. (A CALL or GOSUB to another procedure would qualify as an action in this sense.) Hence, a block of statements that are executed sequentially is called an "action block."

Selection—one set of statements, from a choice of two or more, is selected for sequential execution, based on some criterion. One way to accomplish this is to use an IF/THEN/ELSE construct. Some of the languages will also permit a SELECT/CASE/OTHERWISE/END - type structure, perhaps with different but analogous keywords. The set of statements that gets executed in each case is itself an action block. Sometimes selection constructs are called "branch blocks."

Iteration—a series of statements is executed repeatedly until some termination condition is met. These are also called "loop blocks." Virtually all languages contain simple FOR/NEXT or DO/END-type loops. More modern languages include variations such as DO UNTIL/END and DO WHILE/END.

These three kinds of "control blocks" have some features in common. First, the code in each is executed from top to bottom, which is the same way that it appears in the source file. This makes the program much easier to read and understand than does the convoluted branching you find in so many BASIC programs. Of course, in a selection block, not every statement is executed, and in a loop block they may be executed more than once, but they still are always executed from top to bottom.

In addition, each control block has just one logical entry point: the first statement. And if they're well structured, they have just one logical exit point: the last statement. A complete program is written by assembling and nesting blocks of these three kinds to perform the required processing.

Believe it or not, it's possible to write more or less structured programs in BASIC by following these rules. Some simulation of certain missing language features is required and some GOTOs inevitably creep in. But by keeping the notion of just three flavors of control blocks in mind, a surprisingly good job of structured programming can be done. If you still use BASIC, I encourage you to read a series of articles on structured programming in BASIC by Arthur Luehrmann in the May, June and July 1984 issues of Creative Computing. Unfortunately, Creative Computing is no longer published, but perhaps you or a friend has these back issues in the dusty magazine archives.

More iteration

I imagine you're pretty comfortable with the ideas behind action blocks and selection blocks, but let's take a close look at the iteration, or repetition, constructs. I'm sure you're familiar with the simplest type of loop, which looks like this in BASIC:

```
FOR I = 1 TO 10
  Calculate something
NEXT I
```

In other languages, such a loop is commonly called a DO loop and is terminated by an END or END DO statement:

```
DO I = 1 TO 10
```

Calculate something

END

In either case, some test is used to determine the number of times the loop is executed. In these simple examples, a variable called I (the index variable) is incremented after each iteration and compared to the value 10. If I is less than or equal to 10, the statements in the loop are executed again; otherwise, execution of the loop terminates. Here we're assuming that the value of I will go up by one on each iteration. Of course, you can set a different step interval with a statement like: DO I = 1 TO 10 BY 0.5.

Here's the key question: Is the comparison done before the loop is executed or after? There's a big difference. Suppose I has a value of 20 at the time this loop is encountered in the course of executing the program. Will the loop be executed (since the value of the index variable is already greater than 10) or not?

In Atari BASIC, the comparison is done after the contents of the loop are executed, so the loop is always executed at least once. I think this is generally true of simple FOR/NEXT and DO/END loops. Many modern languages resolve any ambiguity by providing two explicit statement choices: DO WHILE and DO UNTIL. In a DO UNTIL loop, the termination condition is tested at the end of the loop, so the loop is always executed at least once. The simple FOR/NEXT loop in BASIC is thus a DO UNTIL type loop. In a DO WHILE construct, the termination condition is tested at the beginning of the loop. If the termination condition is already true, the contents of the loop aren't executed at all.

One important point is that DO WHILE and DO UNTIL loops need not rely on a changing index variable in the termination test. Any logical expression can be used, such as: DO UNTIL STATUS = ‘DONE’. It also may be possible to have complex combinations of termination conditions, either of which could cause loop execution to cease. Consider this example, which will terminate either when I is greater than 100 or when J becomes less than or equal to 30:

D0 I = 1 To 100 BY 10 WHILE J > 30

Calculate something

END

Maybe it will help to see a visual representation of these two looping structures. I'm sure you're familiar with flowcharts, in which action statements are represented with rectangles and decisions with diamonds. Figure la uses a fragment of a flowchart to illustrate that, in a DO WHILE loop, the conditional test is done before the action statements are executed. In Figure 1b, you see that the conditional test for a DO UNTIL is performed after the action statements are executed. I'm always getting these two confused, so maybe these simple diagrams will help.

FIGURE 1A

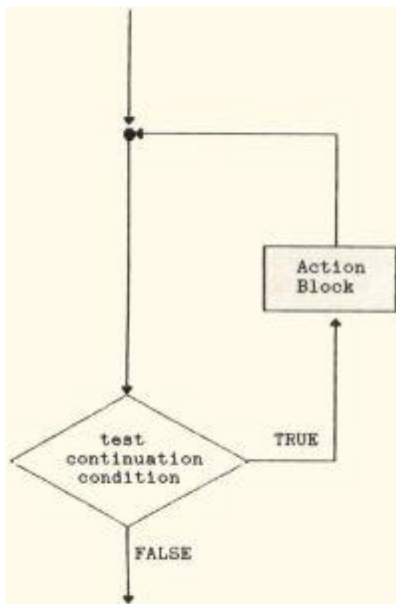


FIGURE 1B

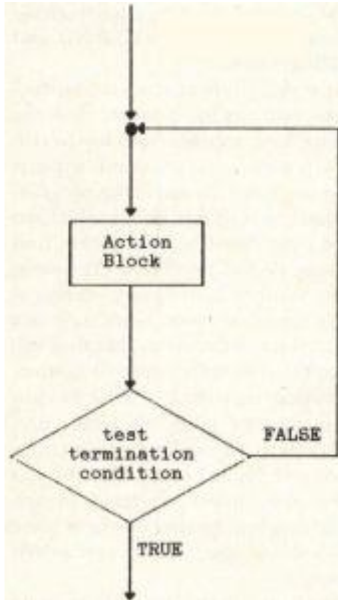
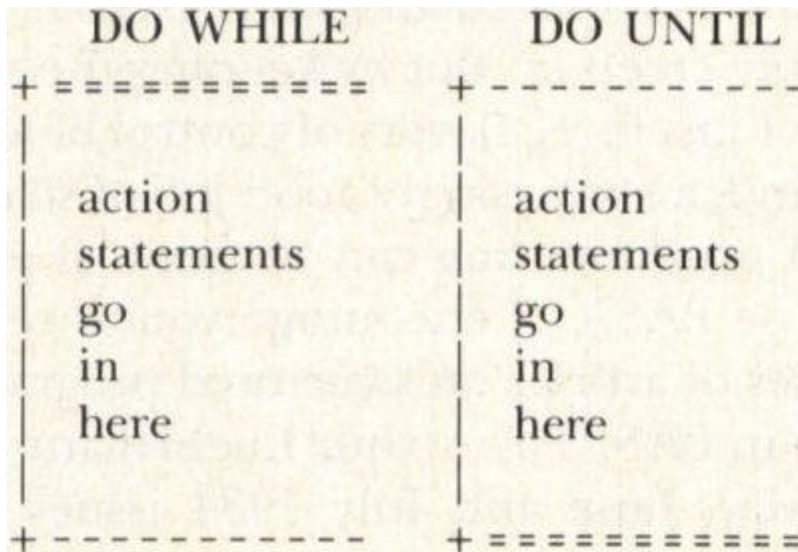


FIGURE 2



Once upon a time we talked about using another diagramming technique to depict the structure of a program module: action diagrams. You may recall that, in an action diagram, a loop is denoted by a double horizontal line, which encloses the action statements in the loop within brackets. You can distinguish between DO WHILE and DO UNTIL logic by placing the double horizontal line at the top or the bottom of the loop bracket, respectively.

Figure 2 shows what you get.

There are times when you want execution of a loop to end prematurely, for a variety of reasons. Bad input data may have been encountered, some error condition may have cropped up or whatever. Remember that we want to write our code so that each control block has only one logical exit point, so let's resist the impulse to GOTO out of the loop. A DO WHILE is a good approach, because you can put the anticipated error or exit conditions into the WHILE clause:

DO I = 1 TO 10 WHILE EOF = ‘FALSE’ AND INPUT = ‘OK’

But things can get tricky if you don't have a DO WHILE available; you may just have to violate the one-exit rule. Some languages provide a command like LEAVE for premature loop exits. In BASIC, you can set the index variable to its termination condition, so that the loop isn't executed anymore; FORTRAN won't let you do this. A more graceful solution is to use an IF block that tests for the abnormal termination condition right inside your DO block:

```
DO I = 1 TO 10
  IF EOF = ‘FALSE’ AND INPUT = ‘OK’ THEN
    DO
      Calculate something
    END
  END IF
END
```

This is basically a way to simulate the DO WHILE command with your own explicit test. Note the innermost DO/END block without any loop criteria on the DO statement. This is really defining an action block to be executed within the selection block (IF/END IF), not a loop block. If you prefer, you can think of this as a loop block that gets executed exactly one time.

Let me out!

The one-entry, one-exit concept applies to individual modules just as it does to loops. Suppose you're calling a subroutine from a main program. You should always enter the subroutine at its top and exit at its bottom. This seems straightforward enough, but there are several ways things can go awry.

Some languages permit multiple entry points to a called procedure, but this is a bad practice. For example, in BASIC you may have seen the use of GOSUB in this form:

10 IF A = 1 THEN GOSUB 1000
20 IF A = 2 THEN GOSUB 2000
30 END
1000 Calculate something
2000 calculate something else
2010 RETURN

Essentially what we have here is one subroutine (Lines 1000-2010) with two entry points (Line 1000 or Line 2000). If A = 1, then control is passed to Statement 1000, it drops through to Statement 2000, then returns at Statement 2010. But if A = 2, only the Lines 2000 and 2010 are executed. Concise, but very confusing; don't do it. If it's necessary to selectively execute certain statements in the subroutine under only certain conditions, put a test for those conditions at the top of the subroutine, or (even better) write two separate subroutines.

Similarly, each subroutine should have only one exit point or RETURN statement. But now something of a dilemma crops up. Suppose that we're being good little software engineers and are programming very defensively, writing lots of code at the top of the subroutine to validate ("audit") the input data. If errors in the data are encountered, we want to abort the operations in the subroutine and return to the calling program with some indication of what went wrong. How can we handle this if we only want to have one exit point in the subroutine? I can think of three alternatives.

4. Forget about the one-exit rule and have multiple RETURN statements in the subroutine, one for each failed audit plus the proper one at the very end.
5. Forget about the no GOTO rule and have a GOTO after each failed audit, which transfers control to the single RETURN statement at the end of the subroutine.
6. Nest our audit tests (which are probably IF statements) as deep as necessary so that only conditions which pass all the tests finally work their way to the inner sanctum where the actual subroutine code resides.

It seems that we have a real quandary here. All of these approaches seem to violate one of the structured programming rules we're trying to abide by. Actually, the third scenario is okay if we have only a couple of audit conditions, because nesting IF statements two or three levels deep is perfectly fine. More levels of nesting than that becomes a problem both in terms of readability (do you want to match up six or seven END IF statements with the corresponding IFs on previous screens?) and because, if you're indenting nicely at each IF level, you may wind up with awfully short lines available for the actual subroutine code in the middle of the IFs. Not good.

My preference in handling the validation question is to think of the blocks of audit code as a "filter" through which the data must pass in order to wend its way to the heart of the module. I tend to put each audit block at the same level of indentation and have a RETURN statement in each block. The main processing section of the procedure has only a single RETURN statement.

Using the filter analogy, any time a piece of bad data is encountered; the RETURN statement acts as a block that prevents the process from continuing. I find that this approach keeps the code easy to read and modify, although I am violating the one exit rule. The next best solution would be to GOTO the single RETURN statement each time an audit fails, but I find this approach to be a little harder to read.

It might be useful to let the calling program know why processing was halted in a subroutine. The calling program will take different actions depending on the cause of the error, be it bad input data, mathematical impossibilities in the computations, file I/O problems or whatever. One way to do this is to pass a return code back to the calling program. The numeric value of the return code might represent the sequence number for an error message to display to the user. The interpretation of the error numbers could rely on a table or external file of messages or actions to match the return codes.

Alternatively, the return code might actually be the name of a procedure to call for handling the error or otherwise transferring execution control. By convention, a return code of 0 usually indicates that no error was encountered.

Wrapping up

The structured programming concepts we've talked about here are all geared toward making the programs you write more understandable to human beings. The computer doesn't care if your program makes sense, so long as it compiles properly. But anyone who must understand how your program works needs all the help he or she can get. You can provide that help not only by using good structured programming practices, but also by using common sense. Keeping your code clear and simple, rather than cute, condensed or clever, will go a long way toward writing programs that are easy to read, comprehend and alter.

Programming style is a set of rules or guidelines used when writing the source code for a computer program. It is often claimed that following a particular programming style will help programmers to read and understand source code conforming to the style, and help to avoid introducing errors.

ning style used in a particular program may be derived from the **coding standards** or **code conventions** of a company or other computing organization, as well as the preferences of the author of the code. Programming styles are often designed for a specific programming language (or language family): style considered good in C source code may not be appropriate for BASIC source code, and so on. However, some rules are commonly applied to many languages.

Elements of good style

Good style is a subjective matter, and is difficult to define. However, there are several elements common to a large number of programming styles. The issues usually considered as part of programming style include the layout of the source code, including indentation; the use of white space around operators and keywords; the capitalization or otherwise of keywords and variable names; the style and spelling of user-defined identifiers, such as function, procedure and variable names; the use and style of comments; and the use or avoidance of particular programming constructs (such as GOTO statements).

Code appearance

Programming styles commonly deal with the visual appearance of source code, with the goal of requiring less human cognitive effort to extract information about the program. Software has long been available that formats source code automatically, leaving coders to concentrate on naming, logic, and higher techniques. As a practical point, using a computer to format source code saves time, and it is possible to then enforce company-wide standards without debates.

Indentation

Indent styles assist in identifying control flow and blocks of code. In some programming languages indentation is used to delimit logical blocks of code, correct indentation in these cases is more than a matter of style. In other languages indentation and whitespace does not affect function, although logical and consistent indentation makes code more readable. Compare:

```
if (hours < 24 && minutes < 60 && seconds < 60)
{
   return true;
}
else
{
   return false;
}
```

or

```
if (hours < 24 && minutes < 60 && seconds < 60) {
   return true;
} else {
   return false;
}
```

with something like

```
if (    hours<
24  && minutes<
60  && seconds<
```

```
60 )
{return  true
;}      else
{return  false
;}
```

The first two examples are probably much easier to read because they are indented in an established way (a "hanging paragraph" style). This indentation style is especially useful when dealing with multiple nested constructs.

Python uses indentation to indicate control structures, so correct indentation is required. By doing this, the need for bracketing with curly braces ({ and }) is eliminated. On the other hand copying and pasting Python code can lead to problems, because the indentation level of the pasted code may not be the same as the indentation level of the current line. Such reformatting is tedious to do by hand, but some text editors and IDEs have features to do it automatically. There are also problems when Python code could be rendered unusable when posted on a forum or web page that removes whitespace, though this problem can be avoided where it is possible to enclose code in whitespace-preserving tags such as "<pre> ... </pre>" (for HTML), "[code]" ... "[/code]" (for bbcode), etc.

Haskell similarly has the off-side rule which lets indentation define blocks; however, unlike in Python, indentation is not compulsory in Haskell — curly braces and semicolons can be (and occasionally are) used instead.

Vertical alignment

It is often helpful to align similar elements vertically, to make typo-generated bugs more obvious. Compare:

```
$search = array('a', 'b', 'c', 'd', 'e');
$replacement = array('foo', 'bar', 'baz', 'quux');

// Another example:

$value = 0;
$anothervalue = 1;
$yetanothervalue = 2;
```

with:

```
$search      = array('a',   'b',   'c',   'd',   'e');
$replacement = array('foo', 'bar', 'baz', 'quux');

// Another example:

         $value = 0;
    $anothervalue = 1;
$yetanothervalue = 2;
```

The latter example makes two things intuitively clear that were not clear in the former:

- the search and replace terms are related and match up: they are not discrete variables;
- there is one more search term than there are replacement terms. If this is a bug, it is now more likely to be spotted.

Arguments against vertical alignment are

- **Inter-line false dependencies**; tabular formatting creates dependencies across lines. For example, if an identifier with a long name is added to a tabular layout, the column width may have to be increased to accommodate it. This forces a bigger change to the source code than necessary, and the essential change may be lost in the noise. This is detrimental to source code control where inspecting differences between versions is essential.

- **Brittleness**; if a programmer does not neatly format the table when making a change; maybe legitimately with the previous point in mind; the result becomes a mess that deteriorates with further such changes.

- **Resistance to change**; tabular formatting requires more effort to maintain. This may put off a programmer from making a beneficial change, such as adding, correcting or improving the name of an identifier, because it will mess up the formatting.

- **Reliance on mono-spaced font**; tabular formatting assumes that the editor uses a fixed-width font. Most modern code

editors support proportional fonts, and the programmer may prefer to use a proportional font for readability.

- **Tool dependence**; some of the effort of maintaining alignment can be alleviated by tools (e.g. a source code editor that supports elastic tabstops), although that creates a reliance on tools.

Coding Style

'Coding style' refers to the way source code is formatted. For C, this involves things like brace placement, indentation, and the way parentheses are used. GNOME has a mix of coding styles, and we do not enforce any one of them. The most important thing is for the code to be consistent within a program or library - code with sloppy formatting is not acceptable, since it is hard to read.

When writing a new program or library, please follow a consistent style of brace placement and indentation. If you do not have any personal preference for a style, we recommend the Linux kernel coding style, or the GNU coding style.

Read the `(Standards)Writing C` info node in the GNU documentation. Then, get the Linux kernel sources and read the file `linux/Documentation/CodingStyle`, and ignore Linus's jokes. These two documents will give you a good idea of what we recommend for GNOME code.

Indentation Style

For core GNOME code we prefer the Linux kernel indentation style. Use 8-space tabs for indentation.

Using 8-space tabs for indentation provides a number of benefits. It makes the code easier to read, since the indentation is clearly marked. It also helps you keep your code honest by forcing you to split functions into more modular and well-defined chunks - if your indentation goes too far to the right, then it means your function is designed badly and you should split it to make it more modular or re-think it.

8-space tabs for indentation also helps you to design functions that fit nicely in a single screen, which means that people can understand the code without having to scroll back and forth in order to understand it.

If you use Emacs, then you can select the Linux kernel indentation style by including this in your `.emacs` file:

```
(add-hook 'c-mode-common-hook
        (lambda ()
          (c-set-style "k&r")
          (setq c-basic-offset 8)))
```

On newer Emacsen or with a newer cc-mode, you may be able to simply do this instead:

```
(add-hook 'c-mode-common-hook
        (lambda ()
          (c-set-style "linux")))
```

If you use vim, then you can select the GNOME kernel indentation style by including this fragment in your `~/.vimrc` file:

```
set ts=8
if !exists("autocommands_loaded")
  let autocommands_loaded = 1
  augroup C
      autocmd BufRead *.c set cindent
  augroup END
endif
```

The GNU indentation style is the default for Emacs, so you do not need to put anything in your `.emacs` to enable it. If you wish to select it explicitly, substitute "gnu" for "linux" in the example above.

If you know how to customize indentation styles in other popular editors, please tell us about it so that we can expand this document.

Naming Conventions

It is important to follow a good naming convention for the symbols in your programs. This is especially important for libraries, since they should not pollute the global namespace - it is very annoying when a library has sloppily-named symbols that clash with names you may want to use in your programs.

Function names should be of the form `module_submodule_operation`, for example, `gnome_canvas_set_scroll_region` or `gnome_mime_get_keys`. This naming convention eliminates inter-module clashes of symbol names. This is very important for libraries.

Symbols should have descriptive names. As Linus says, do not use `cntusr()`, use `count_active_users()` instead. This makes code very easy to read and almost self-documenting.

Try to use the same naming conventions as in GTK+ and the GNOME libraries:

- Function names are lowercase, with underscores to separate words, like this: `gnome_canvas_set_scroll_region()`, `gnome_mime_get_keys()`.
- Macros and enumerations are uppercase, with underscores to separate words, like this: GNOMEUIINFO_SUBTREE() for a macro, and GNOME_INTERACT_NONE for an enumeration value.
- Types and structure names are mixed upper and lowercase, like this: GnomeCanvasItem, GnomeIconList.

Using underscores to separate words makes the code less cramped and easier to edit, since you can use your editor's word commands to navigate quickly.

If you are writing a library, then you may need to have exported symbols that are to be used only within the library. For example, two of the object files that compose the library `libfoo.so` may need to access symbols from each other, but this symbols are not meant to be used from user programs. In that case, put an underscore before the function name and make the first words follow the standard

module/submodule convention. For example, you could have a function called `_foo_internal_frobnicate().`

Consistency in Naming

It is important that your variables be consistently named. For example, a module that does a list manipulation may choose to name the variables that hold a list pointer "l", for terseness and simplicity. However, it is important that a module that manipulates widgets and sizes does not use variables called "w" for both widgets and widths (as in width/height values); this would make the code inconsistent and harder to read.

Of course, these very short and terse names should only be used for the local variables of functions. Never call a global variable "x"; use a longer name that tells what it does.

Standards and Guidelines

Documentation guidelines

In addition to a working program and its source code, you must also author the documents discussed below to gain full credit for the programming project. The fundamental structure of these documents is entirely independent of project, programming language, and operating system. You will find a number of advantages when you pursue a rigid documentation approach to programming. First of all, you will have a firm understanding of the task at hand before you start coding. A good understand of the problem leads to a clean design that tends to have fewer bugs. Always make your goal to program it right the first time! The next advantage is that others will be able to use your documentation to test the program, fix bugs, and make enhancements. In the corporate world, these duties are normally performed by different people and often by different groups within a single company. Therefore, the more detailed, organized, and
easy-to-read your documentation is, the more you help other people do their jobs. As you learn to write solid documentation, you will also come to appreciate reading solid documentation, and will eventually detest reading technical crap (the world is full of poorly written technical books and manuals). In other words, write simply

and clearly. The way you write is just as important as the details you present. Always strive to spell correctly and use proper grammar. The campus Writing Center can aid you in this respect.

- User Requirements Document (URD)
- Requirements Analysis Document (RAD)
- User Interface Specification (UIS)
- Prototype
- Object Oriented Analysis (OOA) or High Level Design (HLD)
- Object Oriented Design (OOD) or Low Level Design (LLD)
- Code Documentation (CD)
- Testing Documentation (TD)
- User's Guide (UG)

User Requirements Document (URD)

This document describes the problem from the user's point of view. It briefly describes the problem domain, e.g.. a psychology experiment or a small business accounting package. Then the document delivers a simple and exact description of the problem. After the problem description, the user states exactly what he/she would like the software system to do. While this may seem to indicate a user interface, it is better to focus on the tasks to be solved rather than the interface required to solve them. However the user may require a specific interface, e.g.. a GUI rather than a command line interface. The crux of this document is to identify what the user requires of the program, and not what the user requires of the programmer. This document furnishes the programmer with a formal description of the problem. Ideally, this document would be written by someone in marketing who has talked with a customer; not by a programmer. The most important thing to remember is that this document describes the functionality required of the program.

The URD has:

- User’s view of the problem
- Brief description of the problem domain
- Complete description of the problem
- What is expected from a software solution
- What is not expected from a software solution

The URD does not have:

- Programmer's point of view
- Programming jargon or technical details
- Description of programming languages or environments unless it is a specific user requirement
-
- Description of the solution. This is not a design document. We only want requirements here.

Requirements Analysis Document (RAD)

This document takes the URD as a starting point and looks at the problem from a designer's point of view. However, instead of diving directly to implementation details, the analysis focuses on the system and software requirements needed to implement the user requirements. This document gets detailed, but does not delve into programming details. Instead, take the user's requirements and clearly identify all of the details and mitigating factors that will affect the solution that the user wants. An analysis may indicate a preference for a particular programming language that best suits the problem domain rather than an algorithm to satisfy a particular requirement. The RAD looks at the URD as defining an entire system, and then breaks the URD down into bite-size chunks (divide and conquer). These chunks identify the subsystems of the overall solution, and the relationships between them. But the RAD also goes further and identifies the actual details of the problem that the user may not be aware of.

The RAD also maps the domain of software systems onto the user requirements. For example, the RAD may indicate that a database is needed for a particular subsystem, or that an expert system can satisfy certain other requirements. The RAD is written from the designer's perspective. An astute software designer is one who is aware of available software systems and paradigms. He/she should know what types of systems and solutions work best in different environments. The RAD, then, identifies the software systems and paradigms that will best fit the user requirements. The RAD doesn't design a solution; it merely identifies the most beneficial means for an implementation.

The RAD has:

- designer's interpretation of the user's requirements: identify the "real" problem(s)
- breakdown the problem into high level constituent parts
- deep analysis of these parts and identification of all relevant details
- identify existing solutions
- identify alternative technical solutions
- link these solutions to the problem(s), especially with respect to details
- suggest the best solution and break it into parts
- devise ways to test the solution

The RAD does not have:

- user's point of view
- implementation details
- algorithms
- user interface specification

User Interface Specification (UIS)

This document describes exactly what the user interface is going to do, what it looks like, and how the user interacts with the program. The UIS does not describe how the interface is implemented. Nor does it describe what the program does behind the interface. Rather, the UIS focuses in detail specifically on the user interface itself. For a GUI, the UIS would define the components and all options on a MenuBar, all of the MenuBar headings, all of the submenus, and all of the options of those submenus. The UIS may describe the functionality of each of the mouse buttons, if appropriate. The UIS also describes the presentation of data, be it graphics, text, or a combination. The UIS should be understandable to the user. The UIS should contain drawings or screen captures
of prototype interaces. For non-GUI applications, the user interface may be either a Command Line Interface (CLI) for which the UIS could be similar to a UNIX man page, or could be an Application Programmatic Interface (API). An API is really nothing more than a collection of library routines that allow you to link and/or layer software components. For an API, the UIS consists of a definition of

the calling interfaces, names of routines or object methods, parameters, and descriptions of what they will do.
The UIS has:
Description of the complete user interface, CLI, or API
what the user interface looks like to the user
how the user interface behaves
how the user interacts with the system
if GUI, names for all of the interactive components, from the mouse to buttons to menus and scrollbars, and pictures of what the interface should look like
if CLI, flags and arguments, inputs and outputs as in a UNIX man page
if API, complete description of the link/call interface
The UIS does not have:
implementation details coverage of the mechanics beneath the user interface

Prototype

Once you know what the user interface (GUI, CLI, or API) is going to look like, go ahead and try to build a "shell" of the user interface as quickly as possible. This is called a prototype. You don't have to implement the core functionality - you just code the interface to see 1) if it's possible and 2) how it appears to the user. Ideally, you should end up throwing this prototype away. The prototype should be shown to the user to see if it's what the user had in mind. That's why you really shouldn't put too much effort into it because chances are the user is going to say, "No, I wanted it to look like this" or
Software Documentation Guidelines "I like this screen, but that one sucks", etc. You want to ensure that the user sees what they want to see - and this applies to GUI, CLI, and API. It's much easier to change a prototype than it is to do all the work on the internals first and then have to change things. The prototype helps you design things in such a way that the interface is independent of the underlying implementation of the actual problem. The interface is not the solution to the problem. But the interface is the user's window to that functionality. That functionality, or solution to the problem, begins with the next step.

Object Oriented Analysis (OOA) or High Level Design

The OOA applies an object-oriented view to the problem. The easiest way to do this is to pick out all of the nouns in the RAD. Each noun

is usually an object. Through out nouns that aren't substantive objects. The next step is to write a detailed description of each object, no matter how trivial it may seem or how much you take this object for granted. Each object must be completely and succinctly documented. This is called a data dictionary. Next, look for overlap between objects and remove objects that are not important to the problem domain. For example, take out user interface components. You will deal with those separately. Once you have removed unnecessary objects, identify their attributes and methods. More often than not, you will find that some of the objects you have are merely attributes of other objects. Next, establish relationships between objects. For example, an employee works for a company. Here, both company and individual are objects. An individual works for a company. A company employs an individual. A company has many workers. An individual usually works for a single company. Here, we have both defining and numerical
(cardinality) relationships. Find such relationships between your objects, and define the cardinality, eg. one-to-one, one-to-many, many-to-many.

The next step of the OOA is to do the same thing with your UIS. The user interface and underlying application subsystems should be completely independent of one another. In fact, you should be able to design and develop your interface and your underlying application independently of one another. You should make your user interface classes as generic as possible, and subclass off of them to get application-specific behavior. For example, to make a list of colors you would probably use a generic List object. A List has a series of text string labels for each choice, lets the user make a choice, activates some function when such a choice is made, allows the user to add an item to the list, delete one, and so on. So far, you could use this List object in almost any application that requires a List
interface object. However, for a color selector, you may want to show the actual color in the rectangular slot where you usually show the textual name of the color. In this case, design the object hierarchy such that most of the functionality is encapsulated in the List object, but then subclass a Color List class to present the colors rather than the textual labels. Design interactive objects in a similar way: allow an application to register actions with an object rather than defining that an object performs specific tasks. Most of the appearance and functionality is then abstracted away from the end user in a particular program. This way you can share an object between applications, and assign application-specific behavior with registration rather than

explicit coding. If you are not using an object oriented programming language like C++ or Java, you can still use object oriented design techniques. However, if you prefer, you can write a High Level Design document in place of the OOA. The HLD accomplishes many of the same goals, but from a non-OO approach.
The OOA has:
data dictionary defining exactly what each object represents (in English) class diagrams showing the name, attributes, and methods of each class relationships between the classes, depicting graphically and textually a set of class diagrams and data dictionary for the application domain a set of class diagrams and data dictionary for the interface domain end-user readability
The HLD has:
detailed breakdown of technical solution in subsystems descriptions of data structures and operations required for each subsystem detailed interaction between subsystems, including interface subsystem(s)
The OOA and HLD do not have:
algorithms
implementation details

Object Oriented Design (OOD) or Low Level Design (LLD)

The OOD is as close to coding as you can get without actually coding. If you do this document correctly, the code will just fall out naturally. The OOD takes the classes in the OOA a level deeper into the realm of pseudo-code. The OOD defines the datatypes for the attributes. The OOD also defines the algorithms and implementation details of the class methods. However, you are not writing code yet. The OOD should be language-independent. You are merely specifying more exactly what the attributes and methods consist of. Like the OOA, the OOD addresses both the interface and application class hierarchies. Also like the OOA, it keeps the two domains separate and independent of one another.
The OOD goes another step beyond the OOA by identifying the objects, rather than the classes, required to implement the software system. While the user may be able to comprehend the OOA, the OOD goes very deep into software design. The OOD is much more detailed than the OOA and establishes instantiations from the class hierarchy and their relationships to one another, especially associations and cardinality. The OOD also provides the algorithms for all class/object methods pseudo-code. For a graphical

environment, the OOD would specify how the event loop dispatches events to specific objects. For example, a mouse-down in a drawing canvas would activate the pen object to start drawing, while a mouse-up would terminate the line. The OOD often makes use of state and event diagrams that define exactly what happens when the user interacts with the graphical components. It maps the user interactions with graphical components to underlying application subsystems. For the drawing example, the OOD maps the mouse down to the pen object that draws the line. The mouse doesn't draw the line, the pen object does, using the mouse coordinates as a guide. For non-OO design, the LLD contains essentially the same contents as the OOD, namely, explicit detail of all data types and functions. You should provide psuedo-code for all algorithms and flesh out all aspects of the programming effort without yet resorting to actual code. If you were to compare this documentation process to writing a research paper, you could regard the HLD as the chapter and section titles, the LLD as the bullet items for each section, and the code as the text of the paper. Each document just gets more and more detailed.
The OOD and LLD have: inner details of class attributes (data types) and methods (functions) detailed object (as opposed to class) diagrams for OOD

- State diagrams
- Event diagrams
- Pseudo-code
- Algorithmic descriptions

The OOD and LLD do not have: Code

Code Documentation (CD)

You are expected to fully document your code. Every class and class method should have a name, a brief one-line description, and a detailed description of the algorithm. All methods also require descriptions of all inputs and outputs. If applicable, you should also note any caveats - things that could go wrong or things that the code doesn't address. Put assumptions in the caveats section. If you are coding in Java, you should use the documentation tags that work with the javadoc utility. This utility automatically generates web pages for your documentation. To make things consistent, simply cut and paste the textual descriptions of your classes, objects, and methods from your OOD directly into the code. Then let javadoc do the dirty work. If you are not coding in Java, you can still use the same tags and see

if javadoc operates on your source files. Otherwise, you could write such a utility yourself!

Testing Documentation (TD)

The TD describes how you tested your program to prove that it works successfully. You should include testbeds for both the user interface and application aspects of your program. You should also provide test datasets where applicable. For example, if you program does some form of text processing, you should provide example file inputs that test specific features of the program requirements. You should pay special attention to borderline values and bogus user input. The TD should also include evaluation criteria, so you know what you are actually testing for.

User's Guide (UG)

This document tells a user how to use your system. The format is up to you. Pick your favorite User's Guide and copy its format. You may even want to use the UNIX man page system. Assume that the user knows nothing about your project. Provide an overview and then details of each subsystem. Make sure you explain how the whole system works before delving into the details. This document is for the end-user or another programmer, so you have to explain the obvious, e.g..Basically, you have to tell someone how to use your system.

Type Checking

Scoping Rules
What is a "normal" scoping rule? In many languages (AWK, Snobol, Perl, etc)
a variable is global unless declared local. Most other languages (C, C++,
Pascal and its heirs, Haskell, Lisp, etc) do need global and local
declarations, but we usually only care for the locals when reading a
program. In C, for instance, when you read inside a function something like

```
varname = 1;
```

you do not go all over the file (and includes) to see whether "varname" is
declared as global. You just check if varname is declared local; if it's
not, then it is a global variable. A compiler does the same, and only
after seeing that the variable is not local it will need a global
declaration.

Very few languages do create local variables without some kind of explicit
declarations. This is because local variables are local to a given scope,
and so we must specify this scope. If we get the rule that "a variable is
local unless defined globally", we will have some weird behaviours; for
instance, in Lua (and also in C and Java) the scope of a variable is not a
whole function, but the inest block where the variable is declared. So, if
you write something like

```
if a then i=1 else ... end
```

the scope of the variable 'i' (now local by default) would be only the
"then" part of the if!! (the inner block...)

In Lua you do not need global declarations. You can complain about this
(and many people do); but to assume that a variable without this global
declaration should be automatically considered local does not look "normal"
to most languages, and certainly is not the right way to solve the problem
with undeclared global variables.

If the real problem is the use of global variables without declaration,
there is an easy and efficient way to solve that, using tag methods. The
idea is to set the tag methods 'get global' and 'set global' over the tag nil
(which is the value of any "undeclared" variable), and to keep a table with
the name of all declared global variables. When you read or write a nil
global variable, the tag method goes to the table to check whether the
variable has been declared. When you read or write a non-nil global, the
most frequent case, there is no cost at all.

```
=================================================================
_Globals = { }   -- a set to keep the names of global variables

function declare (globalname)
  _Globals[globalname] = 1
end

function undeclare (globalname)   -- optional
  rawsetglobal(globalname, nil)
  _Globals[globalname] = nil
end
```

```
settagmethod(tag(nil), 'setglobal', function (name, oldvalue, newvalue)
  if not _Globals[name] then
    error("cannot write undeclared variable `"..name.."'")
  end
  return rawsetglobal(name, newvalue)
end)

settagmethod(tag(nil), 'getglobal', function (name, value)
  if not _Globals[name] then
    error("cannot read undeclared variable `"..name.."'")
  end
  return value   -- or nil (value must be nil...)
end)
```

===

Example:

```
> dofile'temp'   -- the above code
> print(i)
lua: cannot read undeclared variable `i'
Active Stack:
        function error [in file (C)]
        `getglobal' tag method [in file temp]
        main of (dostring) >> "print(i)"
> declare'i'
> print(i)
nil
> i=10
> print(i)
10
> a = 10
lua: cannot write undeclared variable `a'
Active Stack:
        function error [in file (C)]
        `setglobal' tag method [in file temp]
        main of (dostring) >> "a = 10"
> undeclare'i'
> i='hello'
lua: cannot write undeclared variable `i'
Active Stack:
```

```
function error [in file (C)]
`setglobal' tag method [in file temp]
main of (dostring) >> "i='hello'"
```

The scoping rules of a Programing Language dictate how FreeVariables - symbol names which are found in the body of a scope (a function, macro, class, whatever) but not defined there - are resolved.
Several different strategies exist:

- Disallowing them altogether. Very rare; though some languages may disallow free references to all but a pre-defined set of symbols (often called keywords or special forms) which are provided by the language.
- Only two scoping levels - global and local. C Language and many assemblers use this rule. A Free Variable defined in a C function must refer to a "global" symbol (meaning defined either at file or global scope; as opposed to within a function body). This simplifies the implementation of C greatly (and many C programmers don't miss more advanced scope rules at all). Cee Plus Plus sticks to the C tradition in many ways, though the presence of features such as classes and namespaces cause C++ to relax the scoping rules quite a bit. (Still, no C++ function can bind to the local variables of another C++ function). Java, with InnerClasses, relaxes the scoping rules further.
- LexicalScoping. Used by CommonLisp, SchemeLanguage, AlgolLanguage, PascalLanguage, and many others. If a variable isn't found in a given scope, the enclosing scope is searched; repeating until the outermost scope is reached. Two important variants of this are DeepBinding and ShallowBinding. With DeepBinding, variables are bound to the environment of where the function is defined; with shallow binding variables are bound to the environment of where the function is called. Most languages which support LexicalScoping support DeepBinding for functions; most macro systems (excluding Scheme's DefineSyntax) use shallow binding. It is possible to divide DeepBinding further into two separate forms (I am not aware of any generally accepted terminology to describe these forms). In one form, a FreeVariable essentially is an alias for the actual variable in the enclosing scope; if that variable changes than the value of the variable in the function being considered also changes. (If the enclosing scope has exited, possible if FirstClass functions are mixed in with LexicalScoping, then the value the variable had at exit

changes). Most languages which support DeepBinding support this form. In the other form; the FreeVariable takes the value that it has at the point when the function in question is first defined; and does not change. (In other words, the function using the FreeVariable makes a copy of the value provided by the enclosing scope). I'm not aware of any languages which do this for FreeVariables?; though objects used as closures have this behavior. Java InnerClasses (when defined within a function) sidestep this issue by only allowing references to variables in the enclosing function which are declared to be final - in other words, those whose value does not change.

- DynamicScoping (early dialects of Lisp, CommonLisp special variables, exported environment variables in UnixOs): The caller is checked for a binding for the variable; if one is found, it is used. Otherwise, the caller's caller is checked, and so on. If no definition is found, it is either an error or a default value is used, depending on the semantics of the language.

The following C-like program illustrates the different scoping rules. Apply a different rule, that's that what would print. Of course, this is not legal C in real life, as CeeLanguage doesn't allow nested functions.

```
int main (void)
{
    const char *scope = "Lexical, deep, by copy ";

    void print_scope (void)
    {
        printf ("%s\n", scope);
    }

    scope = "Lexical, deep, aliasing";

    void (*)(void) helper_func (void) /* Returns ptr to function;
pretend its a closure */
    {
        const char *scope = "Lexical, shallow scoping";

        void do_print_scope (void)
        {
            print_scope();
        }
        return do_print_scope;
```

```
    }

    void do_it (void)
    {
       const char *scope = "Dynamic Scoping";
       helper_func()(); /* Call the function returned by helper_func();
*/
    }
    do_it();  /* Print what scoping we are using */
 }
```

Another example, in pseudo PascalLanguage: (from 'Compilers: Principles, Techniques, and Tools')

```
 program scoping;
 var r : string;
 procedure show;
   begin
   writeln(r);
   end;
 procedure scope;
   var r : string;
   begin
   r:='Dynamic';
   show;
   end;
 begin
 r:='Scope';
 show;
 r:='Lexical';
 scope;
 end.
```

Note: C does use LexicalScoping but does not allow function definitions to be nested (Standard C doesn't, but GNU C does). C++ also uses LexicalScoping. Namespaces and classes are just lexical scopes, like any other, as are Java InnerClasses.

For example:

```
 int func() {
    int outer_local = 1;

    { // this block introduces a new lexical scope
       int inner_local = 2;
```

```
     printf( "%i %i\n", inner_local, outer_local );
   }

   // inner_local is no longer in scope.
 }
```

Will print:

```
 2 1
```

The important point about C/C++ is that it doesn't allow FreeVariables? in one function to refer to anything defined in another function; this eliminates the need for a StaticChain and/or closures.

So this perfectly valid ISO 9899:1999 does not use LexicalScoping?

```
 int main() {

   for (int i = 0; i < 10; i++) {
     dosomething(i);
     }
   // i is no longer accessible.
 }
```

This is not a GNU extension, this is valid C99. AFAIK the standard does say that blocks group a set of declarations and statements into a syntactic unit, and a compound statement is a block.

Actually, C++ does allow nested functions; you just have to spell the internal function differently. The following is not legal in C++:

```
 void outer (int x)
 {
     void inner (int y);
     {
         cout << y+1;
     }

     inner(x);
 }
```

The following, however, is.

```
 void outer (int x)
 {
      class { void operator () (int y)
```

```
    {
      cout << y+1;
    } inner;

    inner(x);
}
```

In other words, functors are a great way of faking nesting functions. Of course, the functor (still) has no access to the variables defined in outer.

Concurrency Mechanisms.

Software engineering is a team effort. Collaboration refers to the ability of team members to work together to reach a common goal. Collaboration involves communication and sharing of ideas, plans, and progress on an individual's activities. The exact nature of this communication and sharing can vary greatly. For example, working out a design on a shared whiteboard is one form of collaboration. Another form involves one individual taking primary responsibility for an activity and requesting help from others on particular parts of the activity, such as when writing a paper.

In a software engineering environment, the artifacts that the developers wish to collaborate on can be represented as objects. Collaboration can be supported by controlling access to these objects in a way that allows the collaborators to see and modify the objects but prohibits others from doing so. In this paper, we describe a mechanism that supports a range of collaborative concurrency control capabilities, providing what we believe is needed in software engineering environments.

Current approaches to supporting software engineering teams provide better support for prohibiting access to objects than for allowing access by selected individuals. Many software engineering tools use a check in/checkout mechanism to control access. In these systems, there is a public area in which objects are readable by everyone in the project. An individual checks out an object into a private workspace, where only that individual can see and modify it. When the changes are complete, the individual checks the object back into the public area. To collaborate in this type of environment typically involves working outside of the system. For example, developers might use the operating system's access control mechanisms to create groups of

collaborators and change accessibility on individual workspaces so that they become group workspaces. Another approach is to allow multiple developers to checkout the same item and require them to merge their changes when they check the objects back in.

Our approach is to provide an object management infrastructure that supports a very general model of cooperative concurrency control. The goal is to allow individuals to decide which objects they want to work on privately and which objects and with whom they want to collaborate. In developing this infrastructure we focused on four common scenarios in software engineering:

- Co-authoring - Multiple developers modify the same object. This can be a real-time co-authorship as in a shared whiteboard environment or an off-line co-authorship as in coding the same class.
- Observation - One developer modifies an object while others can see the progress being made, but cannot modify the object. For example, a tester might be able to observe a design under development in order to develop test cases.
- Loaning - One developer gives an object to another to work on for a period of time, but expects to get the object back when the second developer is done. For example, a designer might loan a class design to some reviewers expecting to get the design back with review comments added to it.
- Transferring - One developer gives an object to another but does not expect to get that object back. For example, a developer may get reassigned to a different project and therefore need to transfer responsibility for a collection of objects to another individual.

Individuals manipulate objects through executing programs, referred to as clients. Individuals collaborate with each other by allowing the respective clients that they execute to collaborate.

In our approach, collaboration is object-based. Support for cooperation is presented through a capability-based cooperative concurrency control mechanism. A client has to request a capability for an operation on an object before performing the operation. A client will not be granted the capability if the operation being requested conflicts with the operations on the same object for which capabilities are already held by other clients. To determine if there

are conflicts, the cooperative concurrency control mechanism maintains two data structures that are associated with each object.

One data structure is a conflicting operation matrix. Two operations on an object are defined to be conflicting if the order in which they are executed affects the state of the object or the return value of either operation. A conflicting operation matrix specifies whether one operation on an object belonging to a class conflicts with another operation on the same object. This matrix can be used to support a simple read/write locking scheme or a more complex semantics-based locking scheme.

A client, A, cooperates with another client, B, by allowing the cooperating client to concurrently perform conflicting operations on a shared object. The second data structure represents these allowed conflicting operations on an object and is specified by clients through the use of cooperation declarations. A cooperation declaration made by client A, for a specific object, identifies:

- The client(s) with whom client A wishes to cooperate
- The set of operations on the object that client A allows the cooperating clients to perform even if they conflict with client A's operations.
- Client A's role in the collaboration, which is one of collaborator, loaner, or transferred. As a collaborator, client A can continue to perform all operations that it has capabilities for. As a loaner or transferred, client A can only perform non-conflicting operations. A loaner expects to get the objects back when the borrowing clients are done, whereas a transferred does not.

Client B can now cooperate with client A by requesting a capability for the operations that client A specified in its cooperation declaration. A capability is granted to client B if:

- The request is for an operation on an object that does not conflict with any of that object's operations for which capabilities are held by other clients. Conflicts are determined using the conflicting operation matrix for the class to which the object belongs.

- There are cooperation declarations from the owners of the conflicting capabilities allowing client B to perform the operation.

In the absence of collaboration, this mechanism reduces to a lock-based concurrency control mechanism that would prevent two conflicting operations from being performed concurrently. If the conflicting operation matrix defines conflicts for only read and write operation, it reduces to standard read-write locking.

If client A is a collaborator, client B acquires separate capabilities for the operations specified in the cooperation declaration. In this case, both client A and client B hold capabilities for those operations. If client A is a loaner or a transferred, the capabilities held by client A for those operations are transferred to client B. If client A is a loaner, the capabilities are transferred back to client A when client B completes operations. If client A is a transferred, the capabilities are released when client B completes it operations.

Let's see how these primitives could be used to support the loan example from earlier. This example is described in terms of the people to make it easy to understand. This would, however, be implemented by making the respective client programs perform the operations attributed to people. Suppose Robin is designing class X and Pat is expected to review it. For simplicity, assume the class is represented as a single object and that standard reader/writer locking provides the default definition for concurrency control. Also, assume that there are currently no capabilities held on the object representing class X. In this case, Robin can acquire a capability to create/edit class X's design since there are no conflicting capabilities. While Robin is doing this, there can be no concurrent readers or writers of the design since they would conflict with the existing capability. To give the design to Pat, Robin would make a cooperation declaration to loan the design to Pat for writing so that Pat could add review comments. Now, Pat can get a write capability even though it conflicts with Robin's write capability. Pat does the review and then returns the object which restores Robin's write capability. If Robin wanted to continue modifying the design while Pat was reviewing it, Robin would have used a co-author declaration instead of a loan declaration.

Most concurrency control mechanisms prevent concurrent access involving conflicting operations. Our approach provides a range of alternatives from one extreme where there is true concurrency/collaboration to more restricted forms of collaboration such as various forms of delegation. We would expect that one might build a layer on top of our mechanisms that hid these primitives and provided a more appropriate mechanism for a specific software engineering process. For example, the concurrency control aspects of an advanced transaction mechanism could be built using these primitives, or a process could be defined in which a manager specified who was responsible for various activities with the capabilities being automatically created and delegated as the process executes. What we have defined here are the primitive mechanisms that allow a wide range of cooperative concurrency control policies to be established.

I need a mechanism to implement the following scenario:

4. two or more threads need to load a given set of values at the same time
5. only one request must be done per value, so if two threads need to load the same subsets, one must wait for the other
6. I don't want to have a lock (or mutex, or another primitive) on each value since they can be potentially too high.

The scenario could be (suppose thread B enters a little bit earlier)

```
        thread A        thread B

values    5, 8, 9, 12       7, 8, 9, 13, 14

request   5,      12      7, 8, 9, 13, 14

waits for    8, 9

                      >>data loaded<<

retrieves    8, 9
```

```
        >> data loaded <<

returns   5, 8, 9, 12
```

Which concurrent mechanism should I use for this?

Remember a producer/consumer won't work since thread A and B are not exactly consumers (they are only interested on certain data).

Structured programming (sometimes known as modular programming) is a subset of procedural programming that enforces a logical structure on the program being written to make it more efficient and easier to understand and modify. Certain languages such as Ada, Pascal, and dBASE are designed with features that encourage or enforce a logical program structure.

Structured programming frequently employs a top-down design model, in which developers map out the overall program structure into separate subsections. A defined function or set of similar functions is coded in a separate module or sub module, which means that code can be loaded into memory more efficiently and that modules can be reused in other programs. After a module has been tested individually, it is then integrated with other modules into the overall program structure.

Program flow follows a simple hierarchical model that employs looping constructs such as "for," "repeat," and "while." Use of the "Go To" statement is discouraged.

Structured programming was first suggested by Corrado Bohm and Guiseppe Jacopini. The two mathematicians demonstrated that any computer program can be written with just three structures: decisions, sequences, and loops. Edsger Dijkstra's subsequent article, Go To Statement Considered Harmful was instrumental in the trend towards structured programming. The most common methodology employed was developed by Dijkstra. In this model (which is often considered to be synonymous with structured programming, although other models exist) the developer separates programs into subsections that each has only one point of access and one point of exit. Almost any language can use structured programming techniques to avoid common pitfalls of unstructured languages. Unstructured programming must rely upon the discipline of the developer to avoid

structural problems, and as a consequence may result in poorly organized programs. Most modern procedural languages include features that encourage structured programming. Object-oriented programming (OOP) can be thought of as a type of structured programming, uses structured programming techniques for program flow, and adds more structure for data to the model.

UNIT -V

Quality Assurance – Walkthroughs and Inspections – Static Analysis – Symbolic Execution – Unit Testing and Debugging – System Testing – Formal Verification: Enhancing Maintainability during Development – Managerial Aspects of Software Maintenance – Source Code Metrics – Other Maintenance Tools and Techniques.

Quality Assurance

Quality assurance consists of the auditing and reporting functions of management. The goal of quality assurance is to provide management with the data necessary to be informed about product quality, thereby gaining insight and confidence that product quality is meeting its goals. Of course, if the data provided through quality assurance identify problems, it is management's responsibility to address the problems and apply the necessary resources to resolve quality issues.

Cost of Quality

The cost of quality includes all costs incurred in the pursuit of quality or in performing quality-related activities. Cost of quality studies are conducted to provide a base- line for the current cost of quality, identify opportunities for reducing the cost of quality, and provide a normalized basis of comparison. The basis of normalization is almost always dollars. Once we have normalized quality costs on a dollar basis, we have the necessary data to evaluate where the opportunities lie to improve our processes. Furthermore, we can evaluate the effect of changes in dollar-based terms.

Quality costs may be divided into costs associated with prevention, appraisal, and failure. Prevention costs include

• quality planning

• formal technical reviews
• test equipment
• training

Appraisal costs include activities to gain insight into product condition the “first time through” each process. Examples of appraisal costs include

• in-process and inter process inspection
• equipment calibration and maintenance
• testing

Failure costs are those that would disappear if no defects appeared before shipping a product to customers. Failure costs may be subdivided into internal failure costs and external failure costs. Internal failure costs are incurred when we detect a defect in our product prior to shipment. Internal failure costs include

• rework
• Repair
• failure mode analysis

External failure costs are associated with defects found after the product has been
shipped to the customer. Examples of external failure costs are

• complaint resolution
• product return and replacement
• help line support
• warranty work

As expected, the relative costs to find and repair a defect increase dramatically as we go from prevention to detection to internal failure to external failure costs. Figure 8.1, based on data collected by Boehm [BOE81] and others, illustrates this phenomenon. Anecdotal data reported by Kaplan, Clark, and Tang [KAP95] reinforces earlier cost statistics and is based on work at IBM’s Rochester development facility:

Software quality assurance

Even the most jaded software developers will agree that high-quality software is an important goal. But how do we define quality? A wag once said, "Every program does something right, it just may not be the thing that we want it to do."

Many definitions of software quality have been proposed in the literature. For our purposes, software quality is defined as Conformance to explicitly stated functional and performance

requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.
There is little question that this definition could be modified or extended. In fact, a definitive definition of software quality could be debated endlessly. For the purposes of this book, the definition serves to emphasize three important points:

1. Software requirements are the foundation from which quality is measured. Lack of conformance to requirements is lack of quality.
2. Specified standards define a set of development criteria that guide the manner in which software is engineered. If the criteria are not followed, lack of quality will almost surely result.
3. A set of implicit requirements often goes unmentioned (e.g., the desire for ease of use and good maintainability). If software conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspect.

Background Issues

Quality assurance is an essential activity for any business that produces products to be used by others. Prior to the twentieth century, quality assurance was the sole responsibility of the craftsperson who built a product. The first formal quality assurance and control function was introduced at Bell Labs in 1916 and spread rapidly throughout the manufacturing world. During the 1940s, more formal approaches to quality control were suggested. These relied on measurement and continuous process improvement as key elements of quality management.

Today, every company has mechanisms to ensure quality in its products. In fact, explicit statements of a company's concern for quality have become a marketing ploy during the past few decades. The history of quality assurance in software development parallels the history of quality in hardware manufacturing. During the early days of computing (1950s and 1960s), quality was the sole responsibility of the programmer. Standards for quality assurance for software were introduced in military contract software development during the 1970s and have spread rapidly into software development in the commercial world Extending the definition presented earlier, software quality assurance is a "planned and systematic pattern of

actions" that are required to ensure high quality in software. The scope of quality assurance responsibility might best be characterized by paraphrasing a once-popular automobile commercial: "Quality Is Job #1." The implication for software is that many different constituencies have software quality assurance responsibility—software engineers, project managers, customers, salespeople, and the individuals who serve within an SQA group. The SQA group serves as the customer's in-house representative. That is, the people who perform SQA must look at the software from the customer's point of view.

SQA Activities

Software quality assurance is composed of a variety of tasks associated with two different constituencies—the software engineers who do technical work and an SQA group that has responsibility for quality assurance planning, oversight, record keeping, analysis, and reporting. Software engineers address quality (and perform quality assurance and quality control activities) by applying solid technical methods and measures, conducting formal technical reviews, and performing well-planned software testing. Only reviews are discussed in this chapter. Technology topics are discussed in Parts Three through Five of this book. The charter of the SQA group is to assist the software team in achieving a high quality end product. The Software Engineering Institute recommends a set of SQA activities that address quality assurance planning, oversight, record keeping, analysis, and reporting. These activities are performed (or facilitated) by an independent SQA group that:

Prepares an SQA plan for a project.

The plan is developed during project planning and is reviewed by all interested parties. Quality assurance activities performed by the software engineering team and the SQA group are governed by the plan. The plan identifies

- Evaluations to be performed
- Audits and reviews to be performed
- Standards that are applicable to the project
- Procedures for error reporting and tracking
- Documents to be produced by the SQA group
- Amount of feedback provided to the software project team

Participates in the development of the project's software process description.
The software team selects a process for the work to be performed. The SQA group reviews the process description for compliance with organizational policy, internal software standards, externally imposed standards (e.g., ISO-9001), and other parts of the software project plan.
Reviews software engineering activities to verify compliance with the defined software process.
The SQA group identifies, documents, and tracks deviations from the process and verifies that corrections have been made.

Audits designated software work products to verify compliance with those defined as part of the software process.
The SQA group reviews selected work products; identifies, documents, and tracks deviations; verifies that corrections have been made; and periodically reports the results of its work to the project manager.

Ensures that deviations in software work and work products are documented and handled according to a documented procedure.
Deviations may be encountered in the project plan, process description, applicable standards, or technical work products.

Records any noncompliance and reports to senior management.
Noncompliance items are tracked until they are resolved. In addition to these activities, the SQA group coordinates the control and management of change (Chapter 9) and helps to collect and analyze software metrics.

Walkthroughs and Inspections

Walkthroughs are a simple, practical, well- accepted technique for drastically improving the quality of software. A walkthrough is peer group review of any product.

"People never seem to have time to do the job right the first time, but they always have the time to do it twice."A walkthrough group reviews the material or the work product of a reviewee. The person who created the product has to assist his peers who review his product by providing appropriate documents and materials. The emphasis here is to detect bugs and not fixing bugs. Once the

design or code flaws are determined after having. “Walked-through” , a secretary records all the bugs or the errors and keeps track of them and also prepares a checklist.
It is the job of the person who developed the work product or the material to fix the bugs and report to the review team usually consists of a group of 4-5 people.

Benefits:

9. Find errors, quickly and economically.
10. Increase programmer productivity
11. Reduce maintenance effort.
12. Spot gross design or implementation inefficiencies.
13. Establish standards for analysis, design, coding, testing, and documentation.
14. Training
15. Insurance that work can be salvaged if someone leaves.
16. Determine employee efficiency and hence productivity

Types of Walkthroughs:

Formal

8. For the big-wigs
9. Often superficial
10. Slow feedback
11. Critiques whose quality is highly variable
12. Review by supervisors
13. Long preparation time
14. Relatively complete products

Informal

4. Little preparation from rough notes
5. Quick feedback review by peers
6. Reviewers have “ownership”

Semi-formal

3. Review by peers entirely on the project
4. Has agenda, time limit, and a set of procedures.

Guidelines to be observed during walkthroughs:

- Ensure all work products are reviewed. This includes the developer to the project leader. Ensuring this will create a congenial atmosphere for project development.
- Major issues must be addressed.
- Walkthroughs should be of a strict duration without dragging them for longer times.
- Avoid correction of errors during walkthroughs and ensure detection of errors.
- Review the product, not the producer.
- Set an agenda and maintain it.
- Limit debate and rebuttal.
- Highlight problem areas, but don't attempt to solve every problem noted.
- Take written notes.
- Limit the number of participants and insist upon advance preparation.
- Develop a checklist for each work product that is likely to be reviewed.
- Allocate resources and time schedule for quick completion.
- Conduct meaningful training for all reviewers.

Inspections:

The IEEE defines an inspection as " a formal verification techniquole assigned to them. One might test in which software requirements, design or code is examined in detail by a person or a group other than the author to detect faults, violations of development standards and other problems".

- Inspections make use of checklists of items to be examined on the work products.

- Unlike walkthroughs, here each "inspector" inspects the product based on the role assigned to them. One might test the user interface; one might inspect the functionality and so on.
- Items to be inspected in a code are subprogram interfaces, decisions made, data referencing, I/O statements, comments, data flow and memory usage.

Some examples of checklist items include

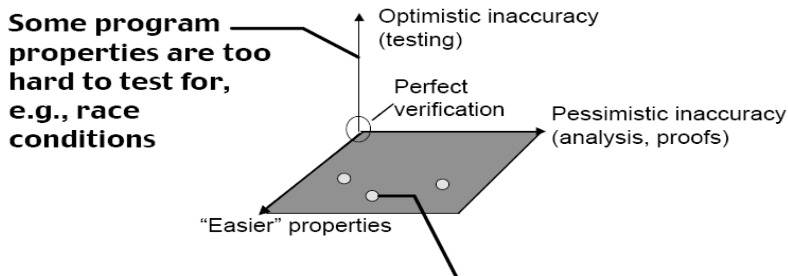
- Agreement between number of formal and actual parameters.
- Are constants being passed to modifiable arguments?
- Consistent usage of global variables among various modules.
- Maximum number of lines in a module.
- Agreement in naming conventions to be followed.
- Are comments being used appropriately and self documenting?

Static Analysis

Static analysis, **static projection**, and **static scoring** are terms for simplified analysis wherein the effect of an immediate change to a system is calculated without respect to the longer term response of the system to that change. Such analysis typically produces poor correlation to empirical results. It's opposite, dynamic analysis or dynamic scoring, is an attempt to take into account how the system is likely to respond to the change. One common use of these terms is budget policy in the United States, although it also occurs in many other statistical disputes.

• Static = without program execution – An over-broad term, includes almost everything except conventional testing
• Why?
– Because dynamic testing requires running code;
Analysis can be applied earlier in development
– Because some kinds of defects are hard to find by testing (e.g., timing-dependent errors)
– Because testing and analysis are complementary; each is best at finding different faults

Analysis where testing fails ...



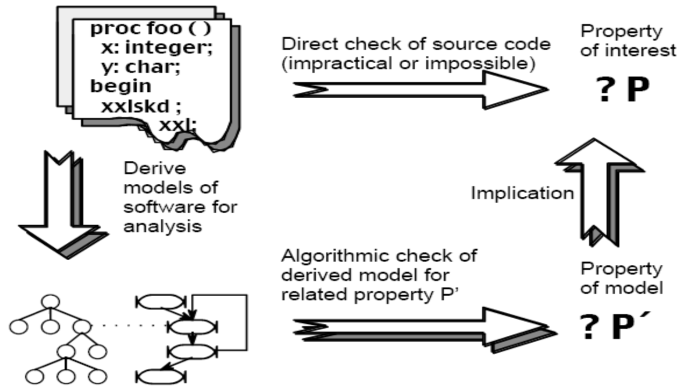
We can’t prove program correctness, but we can prove (simple) properties of (simplified) models

Static Analysis: Low and High Tech

- **Low–tech static analysis:**
  - Software inspection
  - Simple syntactic standards and manual checks
- **High–tech static analysis**
  - Enforced syntactic checks
    - Well–formedness checks in specifications, designs, and code (e.g., matching connectors in design diagrams)
  - Automated program analyses
    - Often based on data flow analysis
  - Finite–state verification and other “high–power” analyses of models

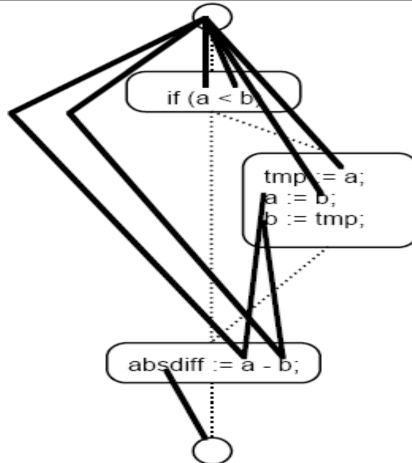
Example:
Uninitialized pointer
• **Dynamic test:** easy to test on a particular execution, but impossible to know if we’ve
covered every possible case
• Static (data flow) check: three possible outcomes, one is “maybe” (depending on
execution path) – exact static check is not possible in general
• Easier (stronger) property: enforce initialization idiom, ban explicit deallocation (as in Java)

Analysis of Models



Data flow analysis

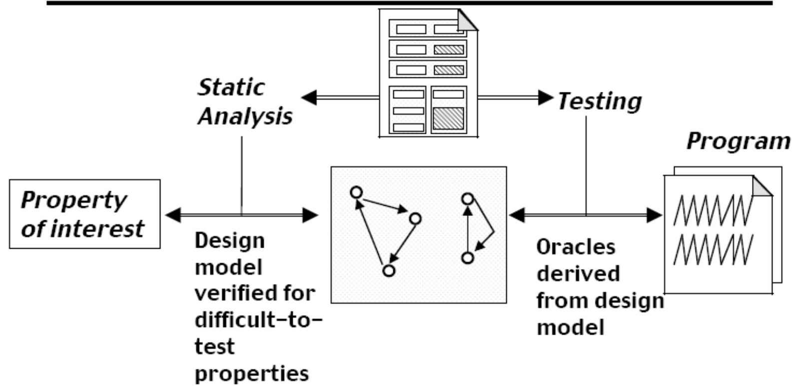
```
function absdiff (a, b: integer)
   return integer is
         tmp: integer;
begin
         if (a < b) then
                  tmp := a;
                  a := b;
                  b := tmp;
         end if;
         return ( a - b );
end absdiff;
```



What is static analysis good for?

- Not a replacement for testing – focused, (mostly) automated analysis for limited classes of faults.
- More thorough than testing (within scope) – Conservative analyses are tantamount to formal verification
- Also augments testing, e.g., dependence analysis for data flow testing

Combining Analysis and Test



Testing for conformance to a verified design model can be more effective than directly testing for a property of interest.

Static analysis

The term was used in 1977 in an international academic journal, in a discussion of tax policy. In recent years, it has become very common in academic, business and political discussions of US government economic policy.

Examples

A famous example of static analysis comes from overpopulation theory. Starting with Thomas Malthus at the end of the 18th century, various commentators have projected some short-term population growth trend for years into the future, resulting in the prediction that there would be disastrous overpopulation within a generation or two. Malthus himself essentially claimed that British society would collapse under the weight of overpopulation by 1850, while during the 1960s the book The Population Bomb made similar dire predictions for the US by the 1980s.

Similarly, some scientists used a short-term trend of temperature declines during the 1930s to theorize that the world would be in an ice age by 1978. As with the overpopulation theories, the projection was less accurate than a roll of the dice because it didn't take into

account how factors interact, nor how a short-term trend was being treated like a long-term trend.

For economic policy discussions, predictions that assume no significant change of behavior in response to change in incentives are often termed static projection (and in the US Congressional Budget Office, "static scoring"). By contrast, dynamic scoring refers to projections based on historical response of population to the effect of economic policy changes such as tax increases or cuts.

Typically, static analysis works for very simple systems: for example, how fast snow is accumulating in what is thought to be the mid-point of a blizzard. But even then it must be tempered with rationality—guess how much longer the storm might actually last, rather than assuming that snow will fall continually for the next sixty years, and project the average of the storm so far, rather than plotting the curve of its growth as if that will continue to increase for the second half.

However, when applied to dynamically responsive systems, static analysis tends to produce results that are not only incorrect but opposite in direction to what was predicted, as shown in the following applications.

Applications

Economic Policy

Presidents from John F Kennedy to Ronald Reagan were explicit in their recognition of the apparent paradox that while static scoring would suggest a change in taxes will produce revenue change commensurate with the tax, in reality revenue tends to change in the opposite direction, due to the response of those affected.

The empirical evidence for this is available but not widely discussed. Several references, both from economic and political sources, draw on data demonstrating the difference between static scoring of a tax policy change, and the actual result.

Technological singularity

Some have criticized the notion of a technological singularity as an instance of static analysis: accelerating change in some factor of information growth, such as Moore's law or computer intelligence, is projected into the future, resulting in exponential growth or hyperbolic growth (to a singularity), that suggest that everything will be known by a relatively early date.

Satire: safety razors

A satire on this idea has been presented using the development of safety razors: After their invention, all safety razors were single-bladed for 70 years. Then the first double-bladed razor was introduced. It only required 15 years for a third blade to be added, and then one year for the fourth and fifth. Fitting these five data points to a hyperbolic curve produced the prediction that within nine years of the calculation—by the year 2015—safety razors would have an infinite number of blades.

Definition -

Static analysis, also called static code analysis, is a method of computer program debugging that is done by examining the code without executing the program. The process provides an understanding of the code structure, and can help to ensure that the code adheres to industry standards. Automated tools can assist programmers and developers in carrying out static analysis. The process of scrutinizing code by visual inspection alone (by looking at a printout, for example), without the assistance of automated tools, is sometimes called program understanding or program comprehension.

The principal advantage of static analysis is the fact that it can reveal errors that do not manifest themselves until a disaster occurs weeks, months or years after release. Nevertheless, static analysis is only a first step in a comprehensive software quality-control regime. After static analysis has been done, dynamic analysis is often performed in

an effort to uncover subtle defects or vulnerabilities. In computer terminology, static means fixed, while dynamic means capable of action and/or change. Dynamic analysis involves the testing and evaluation of a program based on execution. Static and dynamic analysis, considered together, are sometimes referred to as glass-box testing.

In computer science, **symbolic execution** (also **symbolic evaluation**) refers to the analysis of programs by tracking symbolic rather than actual values, a case of abstract interpretation. The field of symbolic simulation applies the same concept to hardware. Symbolic computation applies the concept to the analysis of mathematical expressions.

Symbolic execution is used to reason about all the inputs that take the same path through a program.

Example

Consider the program below, which reads in a value and fails if the input is 6. If this program is symbolically executed, a special symbolic variable (as distinct from the program's variables) is associated with the values returned from the read function. These symbolic variables, and expressions of them are tracked in a special symbolic state. The symbolic variable, which we call s, is assigned to y in the symbolic state, later when y is multiplied by two, y is updated to contain the expression 2 * s.

At any control transfer instructions, such as the y ==12, a Path Constraint is updated to track which branch was taken. In this example assuming the condition is true, the Path Constraint is updated, from being empty, to contain: 2 * s ==12.

```
y = read()
y = 2 * y
if (y == 12)
  fails()
print("OK")
```

By negating some of the conditions in the Path Constraint, and by using a constraint solver to obtain satisfying assignments to the modified Path Constraint it is possible to generate inputs that explore new parts of the program.

Testing

Symbolic execution is useful for software testing because it can analyse if and when errors in the code may occur. It can be used to predict what code statements do to specified inputs and outputs. It is also important for considering path traversal.

Limitations

Symbolic execution is used to reason about a program path-by-path. This may be superior to reasoning about a program, like Dynamic program analysis does, input-by-input. But if few inputs take the same path through the program, there is no saving over testing each of the inputs separately.

Addressing the path explosion of symbolic execution is a research problem .

History

The concept of symbolic execution was introduced academically with descriptions of: the Select system, the EFFIGY system, the DISSECT system, and Clarke's system. See a bibliography of more technical papers published on symbolic execution.

Symbolic Execution

In computer science, **symbolic execution** (also **symbolic evaluation**) refers to the analysis of programs by tracking symbolic rather than actual values, a case of abstract interpretation. The field of symbolic simulation applies the same concept to hardware. Symbolic computation applies the concept to the analysis of mathematical expressions.

- Symbolic execution is used to reason about all the inputs that take the same path through a program.

Example

Consider the program below, which reads in a value and fails if the input is 6. If this program is symbolically executed, a special symbolic variable (as distinct from the program's variables) is associated with the values returned from the read function. These symbolic variables, and expressions of them are tracked in a special symbolic state. The symbolic variable, which we call s, is assigned to y in the symbolic state, later when y is multiplied by two, y is updated to contain the expression 2 * s.

At any control transfer instructions, such as the y ==12, a Path Constraint is updated to track which branch was taken. In this example assuming the condition is true, the Path Constraint is updated, from being empty, to contain: 2 * s ==12.

```
y = read()
y = 2 * y
if (y == 12)
  fails()
print("OK")
```

By negating some of the conditions in the Path Constraint, and by using a constraint solver to obtain satisfying assignments to the modified Path Constraint it is possible to generate inputs that explore new parts of the program.

Testing

Symbolic execution is useful for software testing because it can analyse if and when errors in the code may occur. It can be used to predict what code statements do to specified inputs and outputs. It is also important for considering path traversal.

Limitations

Symbolic execution is used to reason about a program path-by-path. This may be superior to reasoning about a program, like Dynamic program analysis does, input-by-input. But if few inputs take the same path through the program, there is no saving over testing each of the inputs separately.

Addressing the path explosion of symbolic execution is a research problem.

History

The concept of symbolic execution was introduced academically with descriptions of: the Select system, the EFFIGY system, the DISSECT system, and Clarke's system. See a bibliography of more technical papers published on symbolic execution.

Unit Testing and Debugging

In computer programming, **unit testing** is a method by which individual units of source code are tested to determine if they are fit for use. A unit is the smallest testable part of an application. In procedural programming a unit may be an individual function or procedure. Unit tests are created by programmers or occasionally by white box testers.

- Ideally, each test case is independent from the others: substitutes like method stubs, mock objects,[1] fakes and test harnesses can be used to assist testing a module in isolation. Unit tests are typically written and run by software developers to ensure that code meets its design and behaves as intended. Its implementation can vary from being very manual (pencil and paper) to being formalized as part of build automation.

Benefits

The goal of unit testing is to isolate each part of the program and show that the individual parts are correct.[2] A unit test provides a strict, written contract that the piece of code must satisfy. As a result, it affords several benefits. Unit tests find problems early in the development cycle.

Facilitates change

Unit testing allows the programmer to refactor code at a later date, and make sure the module still works correctly (i.e., regression testing). The procedure is to write test cases for all functions and

methods so that whenever a change causes a fault, it can be quickly identified and fixed.

Readily-available unit tests make it easy for the programmer to check whether a piece of code is still working properly.

In continuous unit testing environments, through the inherent practice of sustained maintenance, unit tests will continue to accurately reflect the intended use of the executable and code in the face of any change. Depending upon established development practices and unit test coverage, up-to-the-second accuracy can be maintained.

Simplifies integration

Unit testing may reduce uncertainty in the units themselves and can be used in a bottom-up testing style approach. By testing the parts of a program first and then testing the sum of its parts, integration testing becomes much easier.

An elaborate hierarchy of unit tests does not equal integration testing. Integration with peripheral units should be included in integration tests, but not in unit tests.[citation needed] Integration testing cannot be fully automated and thus still relies heavily on human testers.[citation needed]

Documentation

Unit testing provides a sort of living documentation of the system. Developers looking to learn what functionality is provided by a unit and how to use it can look at the unit tests to gain a basic understanding of the unit's API.

Unit test cases embody characteristics that are critical to the success of the unit. These characteristics can indicate appropriate/inappropriate use of a unit as well as negative behaviors that are to be trapped by the unit. A unit test case, in and of itself, documents these critical characteristics, although many software development environments do not rely solely upon code to document the product in development.

By contrast, ordinary narrative documentation is more susceptible to drifting from the implementation of the program and will thus become outdated (e.g., design changes, feature creep, relaxed practices in keeping documents up-to-date).

Design

When software is developed using a test-driven approach, the unit test may take the place of formal design. Each unit test can be seen as a design element specifying classes, methods, and observable behaviour. The following Java example will help illustrate this point.

Here is a test class that specifies a number of elements of the implementation. First, that there must be an interface called Adder, and an implementing class with a zero-argument constructor called AdderImpl. It goes on to assert that the Adder interface should have a method called add, with two integer parameters, which returns another integer. It also specifies the behaviour of this method for a small range of values.

```
public class TestAdder {
   public void testSum() {
      Adder adder = new AdderImpl();
      assert(adder.add(1, 1) == 2);
      assert(adder.add(1, 2) == 3);
      assert(adder.add(2, 2) == 4);
      assert(adder.add(0, 0) == 0);
      assert(adder.add(-1, -2) == -3);
      assert(adder.add(-1, 1) == 0);
      assert(adder.add(1234, 988) == 2222);
   }
}
```

In this case the unit test, having been written first, acts as a design document specifying the form and behaviour of a desired solution, but not the implementation details, which are left for the programmer. Following the "do the simplest thing that could possibly work" practice, the easiest solution that will make the test pass is shown below.

```
interface Adder {
   int add(int a, int b);
}
class AdderImpl implements Adder {
   int add(int a, int b) {
      return a + b;
   }
}
```

Unlike other diagram-based design methods, using a unit-test as a design has one significant advantage. The design document (the unit-test itself) can be used to verify that the implementation adheres to the design. With the unit-test design method, the tests will never pass if the developer does not implement the solution according to the design.It is true that unit testing lacks some of the accessibility of a diagram, but UML diagrams are now easily generated for most modern languages by free tools (usually available as extensions to IDEs). Free tools, like those based on the xUnit framework, outsource to another system the graphical rendering of a view for human consumption.

Separation of interface from implementation

Because some classes may have references to other classes, testing a class can frequently spill over into testing another class. A common example of this is classes that depend on a database: in order to test the class, the tester often writes code that interacts with the database. This is a mistake, because a unit test should usually not go outside of its own class boundary, and especially should not cross such process/network boundaries because this can introduce unacceptable performance problems to the unit test-suite. Crossing such unit boundaries turns unit tests into integration tests, and when test cases fail, makes it less clear which component is causing the failure. See also Fakes, mocks and integration testsInstead, the software developer should create an abstract interface around the database queries, and then implement that interface with their own mock object. By abstracting this necessary attachment from the code (temporarily reducing the net effective coupling), the independent unit can be more thoroughly tested than may have been previously achieved. This results in a higher quality unit that is also more maintainable.

Unit testing limitations

Testing cannot be expected to catch every error in the program: it is impossible to evaluate every execution path in all but the most trivial programs. The same is true for unit testing. Additionally, unit testing by definition only tests the functionality of the units themselves. Therefore, it will not catch integration errors or broader system-level errors (such as functions performed across multiple units, or non-functional test areas such as performance). Unit testing should be done in conjunction with other software testing activities. Like all forms of software testing, unit tests can only show the presence of errors; they cannot show the absence of errors.

Software testing is a combinatorial problem. For example, every boolean decision statement requires at least two tests: one with an outcome of "true" and one with an outcome of "false". As a result, for every line of code written, programmers often need 3 to 5 lines of test code.[3] This obviously takes time and its investment may not be worth the effort. There are also many problems that cannot easily be tested at all – for example those that are nondeterministic or involve multiple threads. In addition, writing code for a unit test is as likely to be at least as buggy as the code it is testing. Fred Brooks in The Mythical Man-Month quotes: never take two chronometers to sea. Always take one or three. Meaning, if two chronometers contradict, how do you know which one is correct?

To obtain the intended benefits from unit testing, rigorous discipline is needed throughout the software development process. It is essential to keep careful records not only of the tests that have been performed, but also of all changes that have been made to the source code of this or any other unit in the software. Use of a version control system is essential. If a later version of the unit fails a particular test that it had previously passed, the version-control software can provide a list of the source code changes (if any) that have been applied to the unit since that time.

It is also essential to implement a sustainable process for ensuring that test case failures are reviewed daily and addressed immediately.[4] If such a process is not implemented and ingrained into the team's workflow, the application will evolve out of sync with the unit test suite, increasing false positives and reducing the effectiveness of the test suite.

Applications

Extreme Programming

Unit testing is the cornerstone of Extreme Programming, which relies on an automated unit testing framework. This automated unit testing framework can be either third party, e.g., x Unit, or created within the development group.

Extreme Programming uses the creation of unit tests for test-driven development. The developer writes a unit test that exposes either a software requirement or a defect. This test will fail because either the requirement isn't implemented yet, or because it intentionally exposes a defect in the existing code. Then, the developer writes the simplest code to make the test, along with other tests, pass.

Most code in a system is unit tested, but not necessarily all paths through the code. Extreme Programming mandates a "test everything that can possibly break" strategy, over the traditional "test every execution path" method. This leads developers to develop fewer tests than classical methods, but this isn't really a problem, more a restatement of fact, as classical methods have rarely ever been followed methodically enough for all execution paths to have been thoroughly tested.[citation needed] Extreme Programming simply recognizes that testing is rarely exhaustive (because it is often too expensive and time-consuming to be economically viable) and provides guidance on how to effectively focus limited resources.

Crucially, the test code is considered a first class project artifact in that it is maintained at the same quality as the implementation code, with all duplication removed. Developers release unit testing code to the code repository in conjunction with the code it tests. Extreme Programming's thorough unit testing allows the benefits mentioned above, such as simpler and more confident code development and refactoring, simplified code integration, accurate documentation, and more modular designs. These unit tests are also constantly run as a form of regression test.

Techniques

Unit testing is commonly automated, but may still be performed manually. The IEEE does not favor one over the other. A manual approach to unit testing may employ a step-by-step instructional document. Nevertheless, the objective in unit testing is to isolate a unit and validate its correctness. Automation is efficient for achieving this, and enables the many benefits listed in this article. Conversely, if not planned carefully, a careless manual unit test case may execute as an integration test case that involves many software components, and thus preclude the achievement of most if not all of the goals established for unit testing.

To fully realize the effect of isolation while using an automated approach, the unit or code body under test is executed within a framework outside of its natural environment. In other words, it is executed outside of the product or calling context for which it was originally created. Testing in such an isolated manner reveals unnecessary dependencies between the code being tested and other units or data spaces in the product. These dependencies can then be eliminated.

Using an automation framework, the developer codes criteria into the test to verify the unit's correctness. During test case execution, the framework logs tests that fail any criterion. Many frameworks will also automatically flag these failed test cases and report them in a summary. Depending upon the severity of a failure, the framework may halt subsequent testing.

As a consequence, unit testing is traditionally a motivator for programmers to create decoupled and cohesive code bodies. This practice promotes healthy habits in software development. Design patterns, unit testing, and refactoring often work together so that the best solution may emerge.

DEBUGGING

When a test case uncovers an error, debugging is the process that results in the removal of the error. Debugging is not testing but it

always occurs as a consequence of testing. Some of the debugging approaches:

- Brute force
- Back tracking and
- Cause Elimination

Brute force involves memory drumps, run-time traces and ‘print’ statement to test the variables status.

Back tracking suggests to trace backward through the source code from the site of error until finding the error statement.
In cause elimination, data Related to the error occurrence are organized to isolate potential causes. In addition, the most valuable during debugging is the counsel of other software engineers.

System Testing

Definition

The process of performing a variety of tests on a system to explore functionality or to identify problems. System testing is usually required before and after a system is put in place. A series of systematic procedures are referred to while testing is being performed. These procedures tell the tester how the system should perform and where common mistakes may be found. Testers usually try to "break the system" by entering data that may cause the system to malfunction or return incorrect information. For example, a tester may put in a city in a search engine designed to only accept states, to see how the system will respond to the incorrect input.

System testing

software or hardware is testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. System testing falls within the scope of black box testing, and as such, should require no knowledge of the inner design of the code or logic.

As a rule, system testing takes, as its input, all of the "integrated" software components that have successfully passed integration testing and also the software system itself integrated with any applicable hardware system(s). The purpose of integration testing is to detect any inconsistencies between the software units that are integrated together (called assemblages) or between any of the assemblages and the hardware. System testing is a more limiting type of testing; it seeks to detect defects both within the "inter-assemblages" and also within the system as a whole.

Testing the whole system

System testing is performed on the entire system in the context of a Functional Requirement Specification(s) (FRS) and/or a System Requirement Specification (SRS). System testing is an investigatory testing phase, where the focus is to have almost a destructive attitude and tests not only the design, but also the behaviour and even the believed expectations of the customer. It is also intended to test up to and beyond the bounds defined in the software/hardware requirements specification(s).

Types of tests to include in system testing

The following examples are different types of testing that should be considered during System testing:

- GUI software testing
- Usability testing
- Performance testing
- Compatibility testing
- Error handling testing
- Load testing
- Volume testing
- Stress testing
- Security testing
-
- Scalability testing
- Sanity testing
- Smoke testing
- Exploratory testing
- Ad hoc testing

- Regression testing
- Reliability testing
- Installation testing
- Maintenance testing
- Recovery testing and failover testing.
- Accessibility testing, including compliance with:
    - Americans with Disabilities Act of 1990
    - Section 508 Amendment to the Rehabilitation Act of 1973 Web Accessibility Initiative (WAI) of the World Wide Web Consortium (W3C)

Although different testing organizations may prescribe different tests as part of System testing, this list serves as a general framework or foundation to begin with.

Ad hoc testing

is a commonly used term for software testing performed without planning and documentation (but can be applied to early scientific experimental studies).

The tests are intended to be run only once, unless a defect is discovered. Ad hoc testing is a part of exploratory testing, being the least formal of test methods. In this view, ad hoc testing has been criticized because it isn't structured, but this can also be strength: important things can be found quickly. It is performed with improvisation; the tester seeks to find bugs with any means that seem appropriate. It contrasts to regression testing that looks for a specific issue with detailed reproduction steps, and a clear expected result. Ad hoc testing is most often used as a complement to other types of testing.

A **sanity test** or **sanity check**

is a basic test to quickly evaluate whether a claim or the result of a calculation can possibly be true. The point of a sanity test is to rule out certain classes of obviously false results, not to catch every possible error. In arithmetic, for example, when multiplying by 9, using the divisibility rule for 9 to verify that the sum of digits of the result is divisible by 9 is a sanity test - it will not catch every

multiplication error, however it's a quick and simple method to discover many possible errors.

In computer science, a sanity test is a very brief run-through of the functionality of a computer program, system, calculation, or other analysis, to assure that the system or methodology works as expected, often prior to a more exhaustive round of testing.

GUI software testing

In computer science, **GUI software testing** is the process of testing a product that uses a graphical user interface, to ensure it meets its written specifications. This is normally done through the use of a variety of test cases.

Compatibility testing,

part of software non-functional tests, is testing conducted on the application to evaluate the application's compatibility with the computing environment. Computing environment may contain some or all of the below mentioned elements:

- Computing capacity of Hardware Platform (IBM 360, HP 9000, etc.)..
- Bandwidth handling capacity of networking hardware
- Compatibility of peripherals (Printer, DVD drive, etc.)
- Operating systems (MVS, UNIX, Windows, etc.)
- Database (Oracle, Sybase, DB2, etc.)
- Other System Software (Web server, networking/ messaging tool, etc.)
- Browser compatibility (Firefox, Netscape, Internet Explorer, Safari, etc.)

Browser compatibility testing, can be more appropriately referred to as user experience testing. This requires that the web applications are tested on different web browsers, to ensure the following:

- Users have the same visual experience irrespective of the browsers through which they view the web application.

- In terms of functionality, the application must behave and respond the same way across different browsers.
- **Smoke testing** is a term used in plumbing, woodwind repair, electronics, computer software development, infectious disease control, and the entertainment industry. It refers to the first test made after repairs or first assembly to provide some assurance that the system under test will not catastrophically fail. After a smoke test proves that "the pipes will not leak, the keys seal properly, the circuit will not burn, or the software will not crash outright," the assembly is ready for more stressful testing.
- **Regression testing** is any type of software testing that seeks to uncover software errors after changes to the program (e.g. bugfixes or new functionality) have been made, by retesting the program. The intent of regression testing is to assure that a change, such as a bugfix, did not introduce new bugs.[1] Regression testing can be used to test the system efficiently by systematically selecting the appropriate minimum suite of tests needed to adequately cover the affected change. Common methods of regression testing include rerunning previously run tests and checking whether program behavior has changed and whether previously fixed faults have re-emerged. "One of the main reasons for regression testing is that it's often extremely difficult for a programmer to figure out how a change in one part of the software will echo in other parts of the software."[2] This is done by comparing results of previous tests to results of the current tests being run.
- **Exploratory testing** is an approach to software testing that is concisely described as simultaneous learning, test design and test execution. Cem Kaner, who coined the term in 1983,[1] now defines exploratory testing as "a style of software testing that emphasizes the personal freedom and responsibility of the individual tester to continually optimize the quality of his/her work by treating test-related learning, test design, test execution, and test result interpretation as mutually supportive activities that run in parallel throughout the project."[2]

Scalability Testing

part of the battery of non-functional tests, is the testing of a software application for measuring its capability to scale up or scale out [1]- in

terms of any of its non-functional capability - be it the user load supported, the number of transactions, the data volume etc

Volume Testing

Belongs to the group of non-functional tests, which are often misunderstood and/or used interchangeably. Volume testing refers to testing a software application with a certain amount of data. This amount can, in generic terms, be the database size or it could also be the size of an interface file that is the subject of volume testing. For example, if you want to volume test your application with a specific database size, you will expand your database to that size and then test the application's performance on it. Another example could be when there is a requirement for your application to interact with an interface file (could be any file such as .dat, .xml); this interaction could be reading and/or writing on to/from the file. You will create a sample file of the size you want and then test the application's functionality with that file in order to test the performance.

Testing for reliability is about exercising an application so that failures are discovered and removed before the system is deployed. Because the different combinations of alternate pathways through an application are high, it is unlikely that you can find all potential failures in a complex application. However, you can test the most likely scenarios under normal usage conditions and validate that the application provides the expected service. As time permits, you can apply more complicated tests to reveal subtler defects.

Usability testing

is a technique used to evaluate a product by testing it on users. This can be seen as an irreplaceable usability practice, since it gives direct input on how real users use the system.[1] This is in contrast with usability inspection methods where experts use different methods to evaluate a user interface without involving users.

Usability testing focuses on measuring a human-made product's capacity to meet its intended purpose. Examples of products that commonly benefit from usability testing are foods, consumer

products, web sites or web applications, computer interfaces, documents, and devices. Usability testing measures the usability, or ease of use, of a specific object or set of objects, whereas general human-computer interaction studies attempt to formulate universal principles.

System Testing: Why? What? & How?

'**Unit testing**' focuses on testing each unit of the code.

'**Integration testing**' focuses on testing the integration of "units of code" or components.
Each level of testing builds on the previous level.

'System Testing' is the next level of testing. It focuses on testing the system as a whole.

This article attempts to take a close look at the System Testing Process and analyze:
Why System Testing is done? What are the necessary steps to perform System Testing? How to make it successful?

How does System Testing fit into the Software Development Life Cycle?

In a typical Enterprise, 'unit testing' is done by the programmers. This ensures that the individual components are working OK. The 'Integration testing' focuses on successful integration of all the individual pieces of software (components or units of code).

Once the components are integrated, the system as a whole needs to be rigorously tested to ensure that it meets the Quality Standards.

Thus the System testing builds on the previous levels of testing namely unit testing and Integration Testing.

Usually a dedicated testing team is responsible for doing 'System Testing'.

Why System Testing is important?

System Testing is a crucial step in Quality Management Process.

- In the Software Development Life cycle System Testing is the first level where
  the System is tested as a whole
- The System is tested to verify if it meets the functional and technical
  requirements.
- The application/System is tested in an environment that closely resembles the
  production environment where the application will be finally deployed

- The System Testing enables us to test, verify and validate both the Business
  requirements as well as the Application Architecture

Prerequisites for System Testing:

The prerequisites for System Testing are:

- All the components should have been successfully Unit Tested.
- All the components should have been successfully integrated and Integration
  Testing should be completed.
- An Environment closely resembling the production environment should be
  created.

When necessary, several iterations of System Testing are done in multiple environments.

Steps needed to do System Testing:

The following steps are important to perform System Testing:

Step 1: Create a System Test Plan
Step 2: Create Test Cases
Step 3: Carefully Build Data used as Input for System Testing

Step 4: If applicable create scripts to
- Build environment and
- to automate Execution of test cases

Step 5: Execute the test cases
Step 6: Fix the bugs if any and re test the code
Step 7: Repeat the test cycle as necessary

What is a 'System Test Plan'?

As you may have read in the other articles in the testing series, this document typically describes the following:

- The Testing Goals
- The key areas to be focused on while testing
- The Testing Deliverables
- How the tests will be carried out
- The list of things to be Tested
- Roles and Responsibilities
- Prerequisites to begin Testing
- Test Environment
- assumption
- What to do after a test is successfully carried out
- What to do if test fails

Formal Verification:

Formal verification is the use of mathematical techniques to ensure that a design conforms to some precisely expressed notion of functional correctness. Concretely, assume that you have (1) a model of a design, (2) some description of the environment that the design is supposed to operate in, and (3) some properties that the design is intended to fulfill. Given this information, you may want to search for some input patterns that the environment could generate that will violate the properties. Classical approaches to demonstrating that such input stimuli exist are random simulation, or directed test. Formal verification is an alternative approach that can be used both to search for inputs sequences that violate the properties, and prove that the properties always hold in the case when no such stimuli exist.Formal verification can be applied to designs described at many different levels of abstraction, ranging from the gate level, to RTL implementations, and in some cases

even to transaction level models described in standardized programming languages.There are several different use models for formal verification. Some design teams use it to boost the coverage of random simulation and hunt for bugs, whereas others make full blown use of it to conclusively prove that certain properties hold. State-of-the-art tools integrate formal verification engines with simulation in a seamless way. A particular formal verification problem of great interest in EDA is equivalence checking. Here, the property that is checked is whether two different designs always have the same input/output behavior with respect to some notion of equivalence. Special algorithms are used when it is known that the two designs differ only in restricted ways, for example when combinational optimization only has been used to derive one from the other. Compute resources required by formal verification algorithms generally grow very quickly as design sizes increase. Current formal verification research aims to increase the capacity of the design analysis algorithms, and to make formal verification usable earlier in the design cycle. For equivalence checking, hot topics include better capacity for data path verification and support for verification of sequential transformations.

Enhancing Maintainability during Development

Managerial Aspects of Software Maintenance

SOFTWARE MAINTENANCE

often considered to be an unpleasant, time consuming, expensive an d unrewarding occupation - something that is carried out at the end of development only when a bsolutely necessary.Modification of a **software** product after delivery , to correct faults, to improve performance or other attributes, or to a dapt the product to a modified environment

Modifying a program after it has been put into use
Maintenance management is concerned with planning and predictin g the process of change
Configuration management is the management of products undergoing change.

Enhancing Maintainability

Many activities during **software** development enhance the maintain ability of **software** product.

- Analysis activities
- Standards and guidelines
- Design activities
- Implementation activities
- Supporting documents

From **maintenance** view point, the most important activities tha t occur during analysis are establishing standards and guidelines for t he project and the work products to ensure uniformity of the product s, setting of milestones to ensure that the work products are produce d on schedule, specifying quality assurance etc.

Software maintenance may be performed by the developing or ganization, by the customer, or by a third party on behalf of the custo mer. In any case the customer must be given an estimate of the reso urces required and likely costs to be incurred in maintaining the syst em.

Standards and guidelines: various types of standards and guidelines can be developed to enhance the maintainability of **software**. Standa rd formats for requirements documents and design specifications, str uctured coding conventions and standardized formats for the support ing documents like users manual etc will contribute to the understan dability and hence maintainability of the **software**. Standards can be specified by the **software** quality group.

Design activities: Architectural design is concerned with developin g the functional components, conceptual data structures and intercon nections in **software** system. Detailed design is concerned with spec ifying algorithmic details, concrete data representations and details o f the interfaces among routines and data structures.

Implementation activities: Implementation, like design, should have the primary goal of producing **software** that is easy to understand an d easy to modify.

Supporting documents: **Maintenance** guide and test suite descriptio n are the two important supporting documents that should be prepare d during the **software** development cycle in order to ease **maintena nce** activities.

Managerial Aspects of Software Maintenance

Successful **software maintenance**, like all **software** engineering ac tivities, requires a combination of **managerial** skills and technical ex pertise. One of the most important **aspects** of **software maintenanc e** involves tracking and control of **maintenance** activities. **Maintena nce** activity for a **software** product usually occurs in response to a c hange request filed by a user of the product.

Change requests are usually initiated by users. A change request ma y entail enhancement, adaptation or error correction. A change requ est is first reviewed by an analyst, either closes the change request or submits to the control board the change request, the proposed fix, a nd an estimate of the resources required to satisfy the request.

Change control board: The control board reviews and approves all change requests.The board may deny, recommend a modified versio n of change, or approve the change as submitted. The analyst provid es liaison between the change control and the request initiator. Appr oved changes are forwarded to the **maintenance** programmers for ac tion in accordance with the priority and constrains established by the change control board. The **software** is modified, revalidated and sub mitted to the change control board for approval. If the change contro l board approves, the master tapes and external documents are updat ed to reflect the changes, and the modified **software** is distributed to user sites as specified by the control board.

Change Request Summaries: The status of the change requests and **software maintenance** activities should be summarized on a weekly or monthly basis. The summary should report emergency problems and temporary fix in effect since the last report; new change request s received and their probable dispositional ole open requests, along with the status of progress and probable closing date for each; and c hange requests that have been closed since the last summary report, including a description of each closed request and its disposition. In addition, a **maintenance** trends summary should be included in each change request summary; a trends summary graph showing the num

ber of new requests and the total number of open requests as a functi on of time. Quality Assurance Activities: The quality assurance gro up should conduct audits and spots checks to determine that external documents are properly updated to reflect modifications. Quality ass urance group monitors change requests, prepares change request sum maries, performs regression testing of **software** modifications, provi des configuration management, and retains and protects the physical media for **software** products. The group should be represented on the change control board and should have sign-
off authority for new releases of modified **software** products. Chang e control is administered by quality assurance personnel.

Organizing **maintenance** programmers: **Software maintenance** can be performed by the development team or my members of separate organization. There are advantages and disadvantages to both appro aches.

Members of the development team will be intimately familiar with t he product; they will understand the design philosophy of the system and why it functions as it does. Also they will take great care to des ign and implement the system to enhance maintainability. On the ot her hand they will probably be less careful in preparing the supportin g documentation. Also they may be assigned to new project while re taining the responsibility for **maintenance** of the released product.

Maintenance by a separate group forces more attention to standards and high quality documentation. It is also has the advantage of releas ing the development team to pursue other activities. They can becom e highly expert on various details of the product because they devote t heir full attention to the product. However, a morale problem associa ted with **maintenance** programming, and rightly or wrongly a stigm a is often associated with being a "**maintenance** programmer".

A desirable method of organizing **maintenance** programming is to p eriodically rotate programmers between development and **maintena nce**.

Introduction

In software engineering, software maintenance is the process of enhancing and optimizing deployed software (software release), as well as remedying defects. Software maintenance is one of the phases in the software development process, and follows deployment of the software into the field. The software maintenance phase involves changes to the software in order to correct defects and deficiencies found during field usage as well as the addition of new functionality to improve the software's usability and applicability.

Software maintenance involves a number of specific techniques. One technique is static slicing, which is used to identify all the program code that can modify some variable. It is generally useful in refactoring program code and was specifically useful in assuring Y2K compliance.

The software maintenance phase is an explicit part of the waterfall model of the software development process which was developed during the structured programming movement of computer programming. The other major model, the spiral model developed during the object oriented movement of software engineering makes no explicit mention of a maintenance phase. Nevertheless, this activity is notable, considering the fact that two-thirds of a software system's lifetime cost involves maintenance.

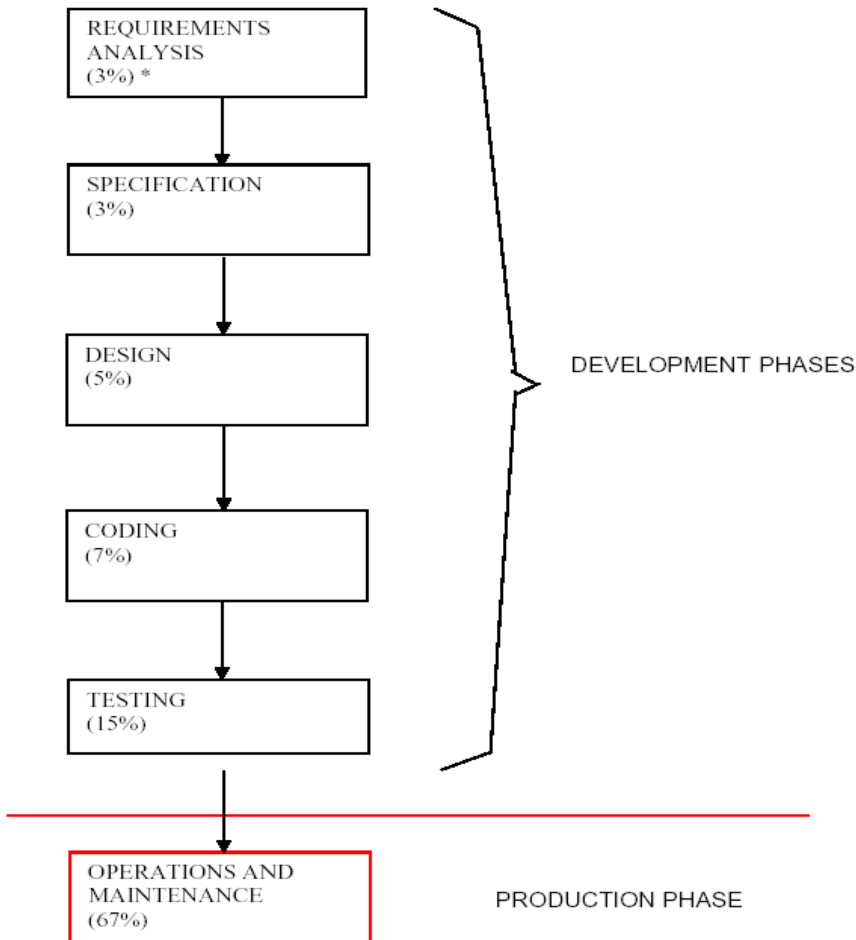


Figure 1: The software maintenance phases

In a formal software development environment, the developing organization or team will have some mechanisms to document and track defects and deficiencies. Software just like most other products, is typically released with a known set of defects and deficiencies. The software is released with the issues because the development organization decides the utility and value of the software at a particular level of quality outweighs the impact of the known defects and deficiencies.

The known issues are normally documented in a letter of operational considerations or release notes so that the users of the software will be able to work around the known issues and will know when the use of the software would be inappropriate for particular tasks.

With the release of the software, other, undocumented defects and deficiencies will be discovered by the users of the software. As these

issues are reported into the development organization, they will be entered into the defect tracking system.
The people involved in the software maintenance phase are expected to work on these known issues, address them, and prepare for a new release of the software, known as a maintenance release, which will address the documented issues.

Need for Maintenance

Maintenance is needed to ensure that the software continues to satisfy user requirements. Maintenance is applicable to software developed using any software life cycle model (for example, spiral). The system changes due to corrective and non-corrective software actions. Maintenance must be performed in order to:

- Correct faults
- Improve the design
- Implement enhancements
- Interface with other systems
- Adapt programs so that different hardware, software, system features, and telecommunications facilities can be used
- Migrate legacy software
- Retire software

The maintainer's activities comprise four key characteristics, according to Pfleeger:

- Maintaining control over the software's day-to-day functions
- Maintaining control over software modification
- Perfecting existing functions
- Preventing software performance from degrading to unacceptable levels

Categories of Maintenance

Maintenance consists of four parts:

- Corrective maintenance: Reactive modification of a software product performed after delivery to correct discovered problems. It deals with fixing bugs in the code.
- Adaptive maintenance: Modification of a software product performed after delivery to keep a software product usable in

a changed or changing environment. It deals with adapting the software to new environments.

- Perfective maintenance: Modification of a software product after delivery to improve performance or maintainability. It deals with updating the software according to changes in user requirements.
- Preventive maintenance: Modification of a software product after delivery to detect and correct latent faults in the software product before they become effective faults. It deals with updating documentation and making the software more maintainable.

All changes to the system can be characterized by these four types of maintenance. Corrective maintenance is 'traditional maintenance' while the other types are considered as 'software evolution'.

Key Issues in Software Maintenance

A number of key issues must be dealt with to ensure the effective maintenance of software. It is important to understand that software maintenance provides unique technical and management challenges for software engineers. Trying to find a fault in software containing 500K lines of code that the software engineer did not develop is a good example. Similarly, competing with software developers for resources is a constant battle. Planning for a future release, while coding the next release and sending out emergency patches for the current release, also creates a challenge. The following section presents some of the technical and management issues related to software maintenance. They have been grouped under the following topic headings:

- Technical issues
- Management issues
- Cost estimation and Measures

Technical Issues

Limited understanding

Limited understanding refers to how quickly a software engineer can understand where to make a change or a correction in software which this individual did not develop. Research indicates that some 40% to

60% of the maintenance effort is devoted to understanding the software to be modified. Thus, the topic of software comprehension is of great interest to software engineers. Comprehension is more difficult in text-oriented representation, in source code, for example, where it is often difficult to trace the evolution of software through its releases/versions if changes are not documented and when the developers are not available to explain it, which is often the case. Thus, software engineers may initially have a limited understanding of the software, and much has to be done to remedy this.

Testing

The cost of repeating full testing on a major piece of software can be significant in terms of time and money. Regression testing, the selective retesting of a software or component to verify that the modifications have not caused unintended effects, is important to maintenance. As well, finding time to test is often difficult. There is also the challenge of coordinating tests when different members of the maintenance team are working on different problems at the same time. When software performs critical functions, it may be impossible to bring it offline to test.

Impact analysis

Impact analysis describes how to conduct, cost effectively, a complete analysis of the impact of a change in existing software. Maintainers must possess an intimate knowledge of the software’s structure and content. They use that knowledge to perform impact analysis, which identifies all systems and software products affected by a software change request and develops an estimate of the resources needed to accomplish the change. Additionally, the risk of making the change is determined. The change request, sometimes called a modification request (MR) and often called a problem report (PR), must first be analyzed and translated into software terms. It is performed after a change request enters the software configuration management process. Arthur states that the objectives of impact analysis are:

- Determination of the scope of a change in order to plan and implement work
- Development of accurate estimates of resources needed to perform the work
- Analysis of the cost/benefits of the requested change

- Communication to others of the complexity of a given change

The severity of a problem is often used to decide how and when a problem will be fixed. The software engineer then identifies the affected components. Several potential solutions are provided and then a recommendation is made as to the best course of action.
Software designed with maintainability in mind greatly facilitates impact analysis.

Maintainability

How does one promote and follow up on maintainability issues during development? The IEEE [IEEE610.12-90] defines maintainability as the ease with which software can be maintained, enhanced, adapted, or corrected to satisfy specified requirements. ISO/IEC defines maintainability as one of the quality characteristics (ISO9126-01).
Maintainability sub-characteristics must be specified, reviewed, and controlled during the software development activities in order to reduce maintenance costs. If this is done successfully, the maintainability of the software will improve. This is often difficult to achieve because the maintainability sub-characteristics are not an important focus during the software development process. The developers are preoccupied with many other things and often disregard the maintainer's requirements. This in turn can, and often does, result in a lack of system documentation, which is a leading cause of difficulties in program comprehension and impact analysis. It has also been observed that the presence of systematic and mature processes, techniques, and tools helps to enhance the maintainability of a system.

Management Issues

Alignment with organizational objectives

Organizational objectives describe how to demonstrate the return on investment of software maintenance activities. Bennett states that "initial software development is usually project-based, with a defined time scale and budget. The main emphasis is to deliver on time and within budget to meet user needs. In contrast, software maintenance often has the objective of extending the life of software for as long as possible. In addition, it may be driven by the need to meet user demand for software updates and enhancements. In both cases, the

return on investment is much less clear, so that the view at senior management level is often of a major activity consuming significant resources with no clear quantifiable benefit for the organization."

Staffing

Staffing refers to how to attract and keep software maintenance staff. Maintenance is often not viewed as glamorous work. Deklava provides a list of staffing-related problems based on survey data. As a result, software maintenance personnel are frequently viewed as "second-class citizens" and morale therefore suffers.

Process

Software process is a set of activities, methods, practices, and transformations which people use to develop and maintain software and the associated products. At the process level, software maintenance activities share much in common with software development (for example, software configuration management is a crucial activity in both). Maintenance also requires several activities which are not found in software development. These activities present challenges to management.

Organizational aspects of maintenance

Organizational aspects describe how to identify which organization and/or function will be responsible for the maintenance of software. The team that develops the software is not necessarily assigned to maintain the software once it is operational.

In deciding where the software maintenance function will be located, software engineering organizations may, for example, stay with the original developer or go to a separate team (or maintainer). Often, the maintainer option is chosen to ensure that the software runs properly and evolves to satisfy changing user needs. Since there are many pros and cons to each of these options, the decision should be made on a case-by-case basis. What is important is the delegation or assignment of the maintenance responsibility to a single group or person, regardless of the organization's structure.

Outsourcing

Outsourcing of maintenance is becoming a major industry. Large corporations are outsourcing entire portfolios of software systems, including software maintenance. More often, the outsourcing option is selected for less mission-critical software, as companies are

unwilling to lose control of the software used in their core business. Carey reports that some will outsource only if they can find ways of maintaining strategic control. However, control measures are hard to find. One of the major challenges for the outsourcers is to determine the scope of the maintenance services required and the contractual details. McCracken states that 50% of outsourcers provide services without any clear service-level agreement. Outsourcing companies typically spend a number of months assessing the software before they will enter into a contractual relationship. Another challenge identified is the transition of the software to the outsourcer.

Maintenance Cost Estimation

Software engineers must understand the different categories of software maintenance, discussed above, in order to address the question of estimating the cost of software maintenance. For planning purposes, estimating costs is an important aspect of software maintenance.

Cost estimation

It was mentioned in “Impact Analysis”, that impact analysis identifies all systems and software products affected by a software change request and develops an estimate of the resources needed to accomplish that change.
Maintenance cost estimates are affected by many technical and non-technical factors. ISO/IEC14764 states that “the two most popular approaches to estimating resources for software maintenance are the use of parametric models and the use of experience”. Most often, a combination of these is used.

Parametric models

Some work has been undertaken in applying parametric cost modeling to software maintenance. Significance is that data from past projects are needed in order to use the models. Jones discusses all aspects of estimating costs, including function points (IEEE14143.1-00), and provides a detailed chapter on maintenance estimation.

Experience

Experience, in the form of expert judgment (using the Delphi technique, for example), analogies, and a work breakdown structure, are several approaches which should be used to augment data from parametric models. Clearly the best approach to maintenance

estimation is to combine empirical data and experience. These data should be provided as a result of a measurement program.

Software Maintenance Measurement

Grady and Caswell discuss establishing a corporate-wide software measurement program, in which software maintenance measurement forms and data collection are described. The Practical Software and Systems Measurement (PSM) project describes an issue-driven measurement process that is used by many organizations and is quite practical.

There are software measures that are common to all endeavors, the following categories of which the Software Engineering Institute (SEI) has identified: size; effort; schedule; and quality. These measures constitute a good starting point for the maintainer.

Specific Measures

Abran presents internal benchmarking techniques to compare different internal maintenance organizations. The maintainer must determine which measures are appropriate for the organization in question. IEEE1219-98 suggests measures which are more specific to software maintenance measurement programs. That list includes a number of measures for each of the four sub-characteristics of maintainability:

- Analyzability: Measures of the maintainer's effort or resources expended in trying to diagnose deficiencies or causes of failure, or in identifying parts to be modified
- Changeability: Measures of the maintainer's effort associated with implementing a specified modification
- Stability: Measures of the unexpected behavior of software, including that encountered during testing
- Testability: Measures of the maintainer's and users' effort in trying to test the modified software

Certain measures of the maintainability of software can be obtained using available commercial tools.

Maintenance Process

The Maintenance Process subarea provides references and standards used to implement the software maintenance process. The Maintenance Activities topic differentiates maintenance from

development and shows its relationship to other software engineering activities.

Maintenance Processes

Maintenance processes provide needed activities and detailed inputs/outputs to those activities, and are described in software maintenance standards IEEE 1219 and ISO/IEC 14764.

The maintenance process model described in the Standard for Software Maintenance (IEEE1219) starts with the software maintenance effort during the post-delivery stage and discusses items such as planning for maintenance.

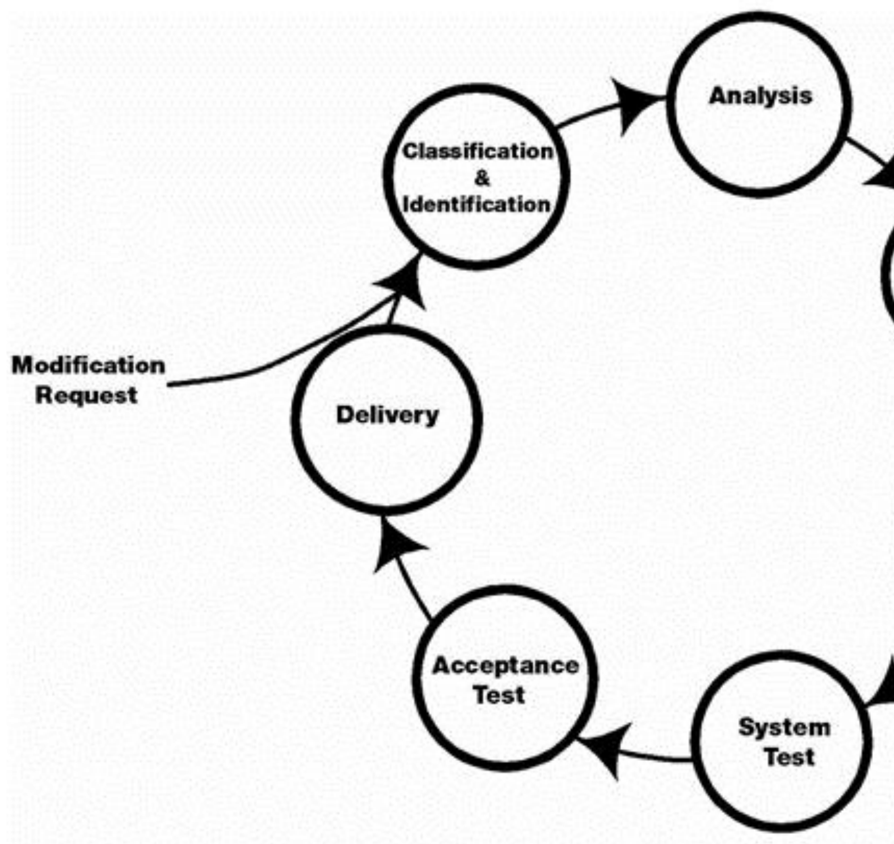


Figure 2: Software maintenance process activities (IEEE1219-98)

ISO/IEC 14764 is an elaboration of the IEEE/EIA 12207.0-96 maintenance process. The activities of the ISO/IEC maintenance process are similar to those of the IEEE, except that they are aggregated a little differently.

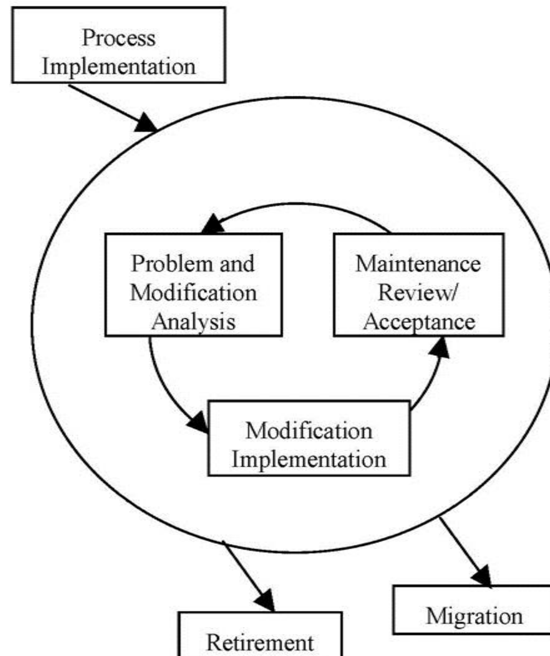


Figure 3: Software maintenance process
(ISO/IEC 14764-00)

Each of the ISO/IEC 14764 primary software maintenance activities is further broken down into tasks, as follows.

- Process Implementation
- Problem and Modification Analysis
- Modification Implementation
- Maintenance Review/Acceptance
- Migration
- Software Retirement

Maintenance Activities

As already noted, many maintenance activities are similar to those of software development. Maintainers perform analysis, design, coding, testing, and documentation. They must track requirements in their activities just as is done in development, and update documentation as baselines change. ISO/IEC14764 recommends that, when a maintainer refers to a similar development process, he must adapt it to meet his specific need. However, for software maintenance, some activities involve processes unique to software maintenance.

Unique activities

There are a number of processes, activities, and practices that are unique to software maintenance, for example:

- Transition: a controlled and coordinated sequence of activities during which software is transferred progressively from the developer to the maintainer.
- Modification Request Acceptance/Rejection: modification request work over a certain size/effort/complexity may be rejected by maintainers and rerouted to a developer.
- Modification Request and Problem Report Help Desk: an end-user support function that triggers the assessment, prioritization, and costing of modification requests.
- Impact Analysis.
- Software Support: help and advice to users concerning a request for information (for example, business rules, validation, data meaning and ad-hoc requests/reports).
- Service Level Agreements (SLAs) and specialized (domain-specific) maintenance contracts which are the responsibility of the maintainers.

Supporting activities

Maintainers may also perform supporting activities, such as software maintenance planning, software configuration management, verification and validation, software quality assurance, reviews, audits, and user training.

Maintenance planning activity

An important activity for software maintenance is planning, and maintainers must address the issues associated with a number of planning perspectives:

- Business planning (organizational level)
- Maintenance planning (transition level)
- Release/version planning (software level)
- Individual software change request planning (request level)

At the individual request level, planning is carried out during the impact analysis. The release/version planning activity requires that the maintainer:

- Collect the dates of availability of individual requests
- Agree with users on the content of subsequent releases/versions
- Identify potential conflicts and develop alternatives
- Assess the risk of a given release and develop a back-out plan in case problems should arise
- Inform all the stakeholders

Whereas software development projects can typically last from some months to a few of years, the maintenance phase usually lasts for many years. Making estimates of resources is a key element of maintenance planning. Those resources should be included in the developers' project planning budgets. Software maintenance planning should begin with the decision to develop a new system and should consider quality objectives (IEEE1061-98). A concept document should be developed, followed by a maintenance plan.
The concept document for maintenance should address:

- The scope of the software maintenance
- Adaptation of the software maintenance process
- Identification of the software maintenance organization
- An estimate of software maintenance costs

The next step is to develop a corresponding software maintenance plan. This plan should be prepared during software development, and should specify how users will request software modifications or report problems. Software maintenance planning is addressed in IEEE 1219 and ISO/IEC 14764. ISO/IEC14764 provides guidelines for a maintenance plan.
Finally, at the highest level, the maintenance organization will have to conduct business planning activities (budgetary, financial, and human resources) just like all the other divisions of the organization.

Software configuration management

The IEEE Standard for Software Maintenance, IEEE 1219, describes software configuration management as a critical element of the maintenance process. Software configuration management procedures should provide for the verification, validation, and audit of each step required to identify, authorize, implement, and release the software product.
It is not sufficient to simply track Modification Requests or Problem Reports. The software product and any changes made to it must be

controlled. This control is established by implementing and enforcing an approved software configuration management (SCM) process. SCM for software maintenance is different from SCM for software development in the number of small changes that must be controlled on operational software. The SCM process is implemented by developing and following a configuration management plan and operating procedures. Maintainers participate in Configuration Control Boards to determine the content of the next release/version.

Software quality

It is not sufficient, either, to simply hope that increased quality will result from the maintenance of software. It must be planned and processes implemented to support the maintenance process. The activities and techniques for Software Quality Assurance (SQA), V&V, reviews, and audits must be selected in concert with all the other processes to achieve the desired level of quality. It is also recommended that the maintainer adapt the software development processes, techniques and deliverables, for instance testing documentation, and test results.

Techniques for Maintenance

This subarea introduces some of the generally accepted techniques used in software maintenance.

Tools

Introduction

A software maintenance tool is an artifact that supports a software maintainer in performing a task. The use of tools for software maintenance simplifies the tasks and increases efficiency and productivity.

There are several criteria for selecting the right tool for the task. These criteria are capability, features, cost/benefit, platform, programming language, ease of use, openness

of architecture, stability of vendor, and organizational culture.

Capability decides whether the tool is capable of fulfilling the task. Once it has been decided that a method can benefit from being automated, then the features of the tool need to be considered for the job.

The tool must be analyzed for the benefits it brings against its cost. The benefit indicators of a tool are quality, productivity,

responsiveness, and cost reduction. The environment that the tool runs on is called the platform. The language of the source code is called the programming language. It's important to select a tool that supports a language that is an industry standard.

The tool should have a similar feel to the ones that the users are already familiar with. The tool should have the ability to be integrated with different vendors' tools. This will help when a tool will need to run with other tools. The openness of the architecture plays an important role when the maintenance problem is complex. Therefore, it is not always sufficient to use only one tool. There may need to be multiple tools running together.

It is also important to consider the vendor's credibility. The vendor should be capable of supporting the tool in the future. If the vendor is not stable, the vendor could run out of business and not be able to support the tool. Another important factor is the culture of the organization. Every culture has its own work pattern. Therefore, it is important to take into consideration whether the tool is going to be accepted by the target users.

The chosen tools must support program understanding and reverse engineering, testing, configuration management, and documentation.

Selecting a tool that promotes understanding is very important in the implementation of change since a large amount of time is used to study and understand programs.

Tools for reverse engineering also accomplish the same goal. The tools mainly consist of visualization tools, which assist the programmer in drawing a model of the system.

Examples of program understanding and reverse engineering tools include the program slicer static analyzer, dynamic analyzer, cross-reference and dependency analyzer.

Slicing is the mechanical process of marking all the sections of a program text that may influence the value of a variable at a given point in the program. Program slicing helps the programmers select and view only the parts of the program that are affected by the changes. Static analyzer is used in analyzing the different parts if the program such as modules, procedures, variables, data elements, objects and classes. A static analyzer allows general viewing of the program text and generates summaries of contents and usage of selected elements in the program text, such as variables or objects.

A dynamic analyzer could be used to analyze the program while it is executing. A data flow analyzer is a static analysis tool that allows the maintainer to track all possible data flow and control flow paths in the program. It allows analysis of the program to better outline the

underlying logic of the program. It also helps display the relationship between components of the system. A cross-reference produces information on the usage of a program. This tool helps the user focus on the parts that are affected by the change.

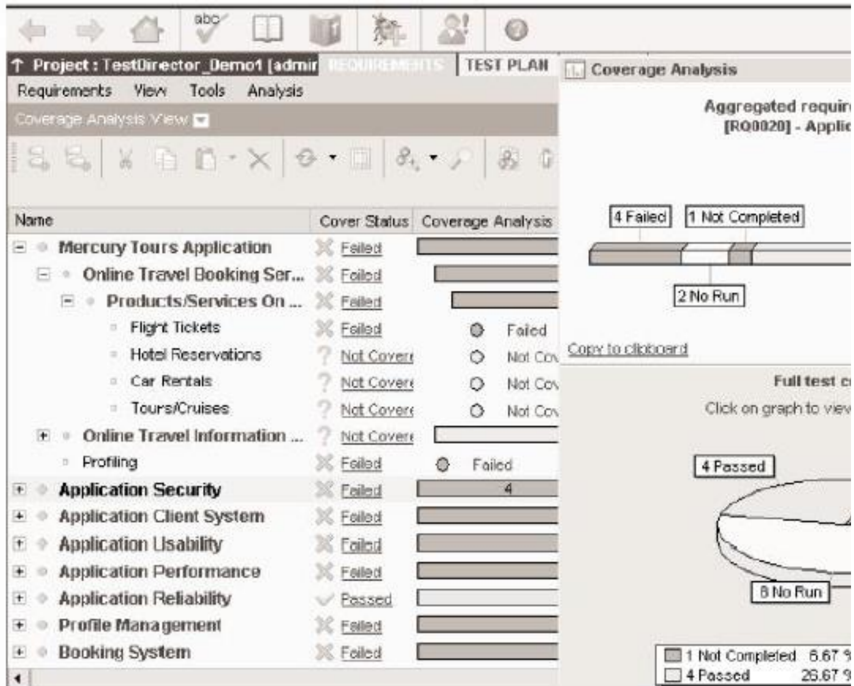
A dependency analyzer assists the maintainer to analyze and understand the interrelationships between entities in a program. Such a tool provides capabilities to set up and query the database of the dependencies in a program. It also provides graphical representations of the dependencies. Testing is the most time consuming and demanding task in software maintenance.

Therefore, it could benefit the most from tools. A test simulator tool helps the maintainer
to test the effects of the change in a controlled environment before implementing the change on the actual system. A test case generator produces test data that is used to test the functionality of the modified system, while a test path generator helps the maintainer to find all the data flow and control flow paths affected by the changes.

Configuration management benefits from automated tools. Configuration management and version control tools help store the objects that form the software system. A source control system is used to keep a history of the files so that versions can be tracked and the programmer can keep track of the file changes.

Commercially available products

There are numerous products on the market available for software maintenance. One type
of product is bug tracking tools, which play an important role in maintenance. Bugzilla by the Mozilla Foundation is an example of such a tool. Other bug tracking products are Test Director by Mercury Interactive, Silk Radar by Segue Software, SQA Manager by Rational software, and QA director by Compuware.



TestDirector's Requirements Manager links test cases to testing requirem traceability.

Figure 4: Test Director by Mercury Interactive

ProTeus III Expert CMMS by Eagle Technology, Inc. is a maintenance software package that lets users schedule preventative maintenance, generate automatic work orders, document equipment maintenance history, track assets and inventory, track personnel, create purchase orders, and generate reports. Microsoft Visual Source Safe is a source control system tool that is used by configuration management.

Products that are specific to programming languages are CCFinder and JAAT which

is specifically designed for JAVA programs. CCFinder identifies code clones in JAVA program. JAAT executes alias analysis for JAVA programs. For C++ programs, there is a tool called OCL query-based debugger which is a tool to debug C++ programs using queries formulated in the object constraint language.

Every one knows are dangerous words, when everyone knows, no-one thinks.

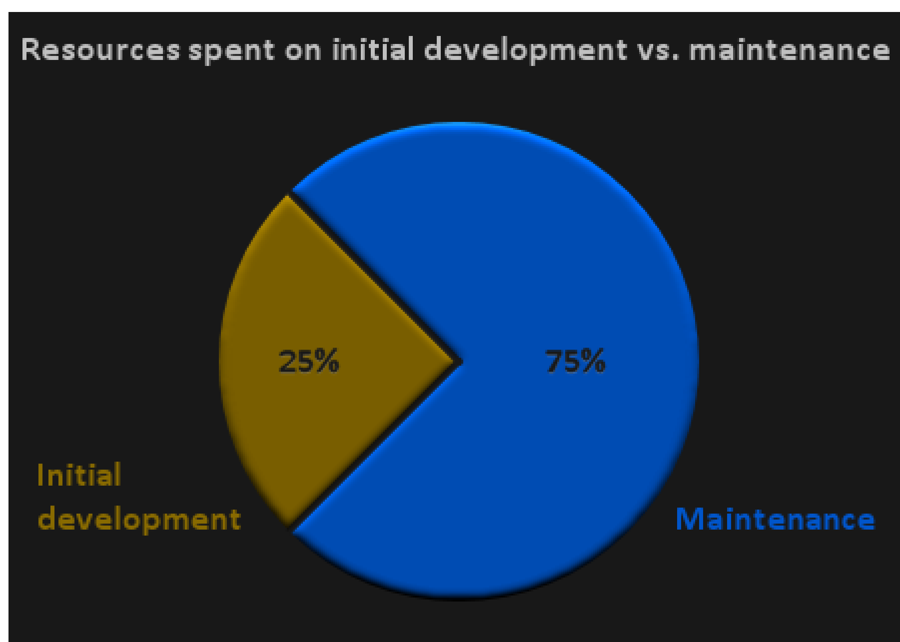
Software maintenance

"Everyone knows" are dangerous words. When everyone knows, no-one thinks.For example, have you ever met a professional software developer who said maintainability didn't matter? Of course not. Everyone knows maintainability is important!

But what does that really mean? What is maintainability, and how can we improve it?

Why does it matter?

In short, because this is where the effort on your next software project might go:



See, for example, the work of Barry Boehm on cost estimation.

Maintenance typically claims 40–80% of all project costs, usually toward the higher end.

So, if we know how maintenance works and what affects our ability to do it, then we can improve those things to make better software, faster

Reasons for performing maintenance

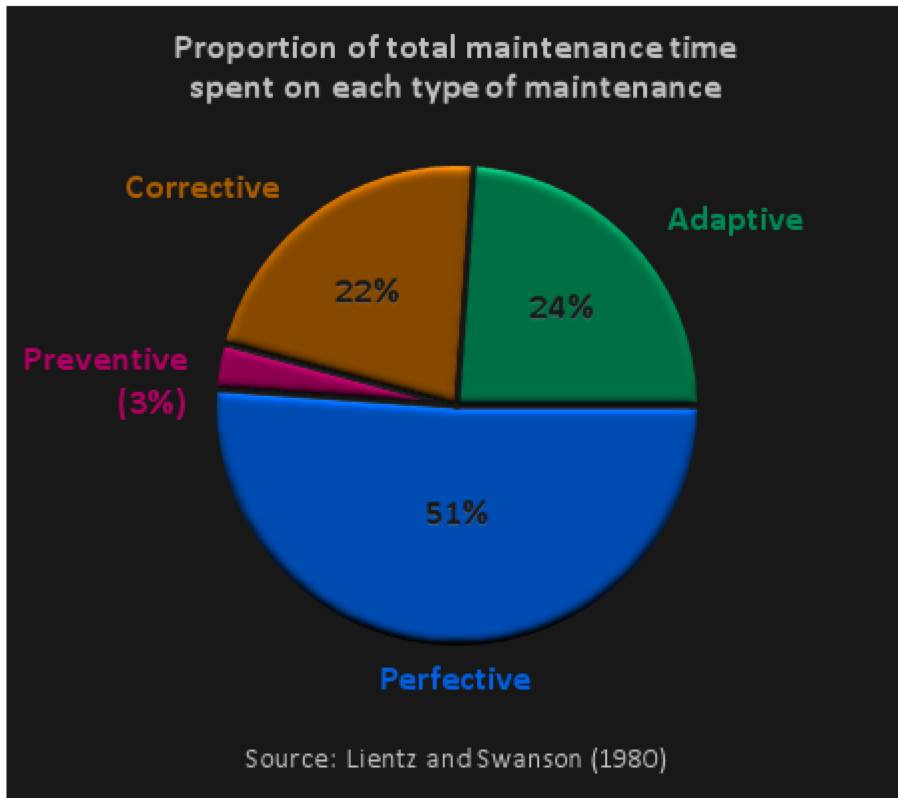
Lientz and Swanson surveyed nearly 500 data processing groups, and identified four distinct types of maintenance and the proportion of time typically spent on each.

Corrective maintenance is work to fix any kind of defect.

Adaptive maintenance responds to environmental changes, such as porting to new hardware or a different OS, but without affecting functionality.

Perfective maintenance includes any functionality changes to meet new requirements, as well as performance improvements.

Preventive maintenance is work to improve maintainability itself, such as refactoring an awkward design or adding comments.



Lientz, B.P. and E.B. Swanson, Software Maintenance Management, Addison-Wesley Longman, 1980.Surprisingly, perhaps, maintenance in the narrow sense of "fixing bugs" represents less than one-quarter of the total time spent working on existing code. Keeping up with changing requirements and operating environments takes a much greater proportion of the maintenance effort over the lifetime of a typical software system.

Despite the importance of maintainability, just 3% of maintenance time is spent improving it. Moreover, only 20% of respondents to a recent survey of software maintenance practices said they carried out specific checking for maintainability as part of their QA process.[3] No wonder code so often degenerates into spaghetti...

Lientz and Swanson’s paper may be three decades old, but it remains widely influential. IEEE standard 1219 defines **software maintenance** to be:

The modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.

The corresponding ISO standard, 14764, classifies maintenance using the same four types, as do many modern papers, though sometimes with slight variations in the definitions.

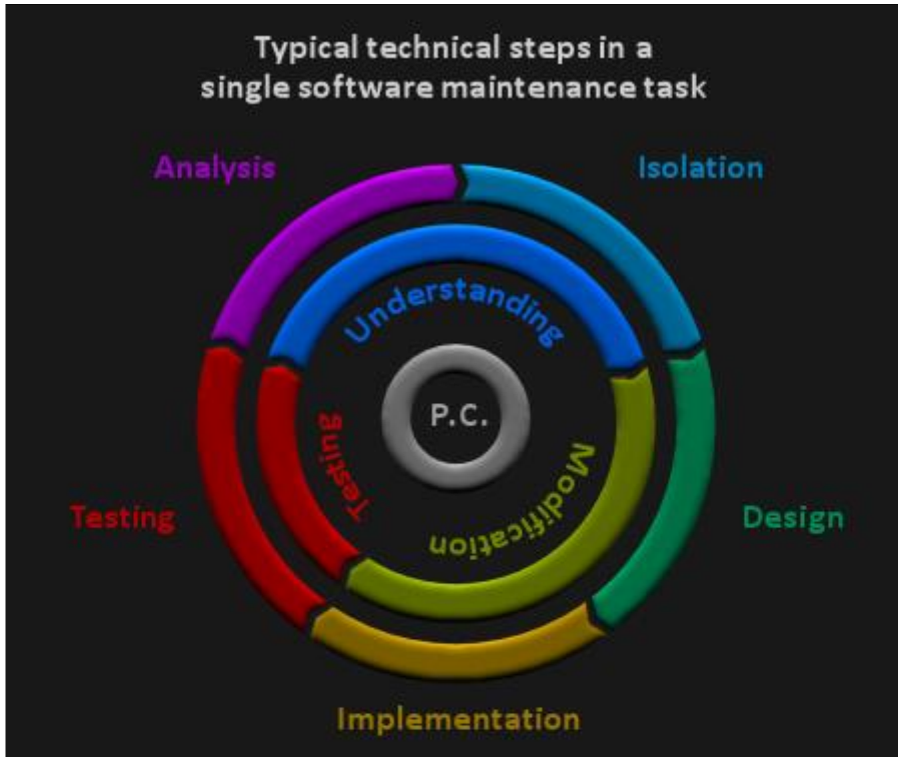
So, we have some data about why we need maintain our code. However, Lientz and Swanson’s approach is not very useful as a predictive tool to help us make decisions or work more effectively. For that, we need to look at how maintenance is performed.

The small-scale maintenance process

Let’s consider a single maintenance task in isolation, say implementing a particular feature or fixing a set of closely related bugs.

The beginning and end of the development process probably differ from initial development. For example, corrective maintenance tends to be driven by some sort of bug triage process, and the changes from any kind of maintenance may need to be deployed to existing installations in the field. Most concerns that arise at these times are non-technical.

What about the technical steps that form the core of the process?



We start with **understanding** what we want to achieve. As with initial development, this involves **analysis** to establish the specific requirements implied by the change. During maintenance, we also have an **isolation** step (sometimes called **impact analysis**), when we identify the area(s) of existing code affected.

Then comes **modification**, consisting of **design** and **implementation** steps. These, too, are like initial development, but with a change in emphasis because any new work must integrate with the established design.

Finally, we have various forms of **testing**. Because we are modifying existing code during maintenance, we add regression testing.

Like other software development processes, maintenance is often iterative, and these steps may form part of a cycle that runs several times before a particular task is complete.

Supporting all of these steps is **program comprehension**, the work of understanding and extracting useful information from our code. This is a very important underlying activity in software maintenance, and the comprehension element may use over 50% of the total time.[4]

[4] See, for example, the research of Anneliese von Mayrhauser, who has studied program comprehension extensively. Van Vliet suggests that overall time spent on the code-related steps is split roughly equally between isolation, modification and testing.[5] However, the distribution of time across the different steps varies greatly with the type of maintenance. For example, during adaptive maintenance, isolation may be a relatively minor part of the work, but for corrective maintenance, the isolation step includes tracking down the source of the defect, and may consume a much greater share of the time.

Information sources during maintenance

Did you ever get the feeling that you were writing a lot of documentation that no-one was ever going to need later? Well, it turns out that you were probably right. Industrial studies suggest a consistent order of importance for information sources during software maintenance, and documentation is way down the list:

Code is the main resource. Singer found that 7 in 10 industrial development groups looked to the code as their primary source.[6] However, this emphasis does come with a caveat: it applies only when the maintenance developer is already somewhat familiar with that code.

Coworkers are the other major source of information. If the developer doing the maintenance work is unfamiliar with the code, or if they can't readily see how to proceed, they will often consult the original programmer or another coworker who knows that part of the code better. As well as Singer, another study by Ko et al observed coworkers providing diverse kinds of information.

Database tools such as bug trackers can be valuable secondary resources. They tend to accumulate wisdom gained by the development team as the project evolves, in relatively concrete and searchable forms.

Documentation does matter, but its value depends very much on what is being described. Typical low-level documentation is widely distrusted by developers, who expect it to be out of date or to present conflicting information. However, higher-level documentation, giving an overview of a feature or describing its original intent, was cited as valuable in Singer. Documentation was also the most common source of information in Ko to answer questions of the form "How do I use this data structure or function?".

A larger scale: the laws of software evolution

Now let's consider a longer-term perspective. Lehman has also studied software evolution, and over the years has proposed several laws to describe it.[8] Here is the first:

Law of Continuing Change: A program that is used and that as an implementation of its specification reflects some other reality, undergoes continual change or becomes progressively less useful. The change or decay process

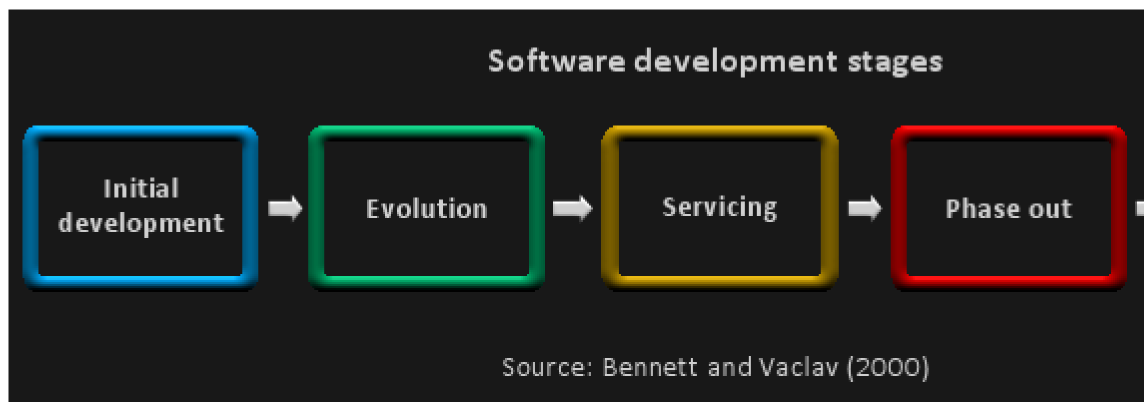
continues until it is judged more cost-effective to replace the system with a recreated version.

And here is the second:

Law of Increasing Complexity: As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it.

Stages in a software project's life

Bennett and Vaclav build on these ideas, suggesting that in reality software projects pass through up to five distinct stages over their lifetime, each with unique maintenance characteristics:[9]



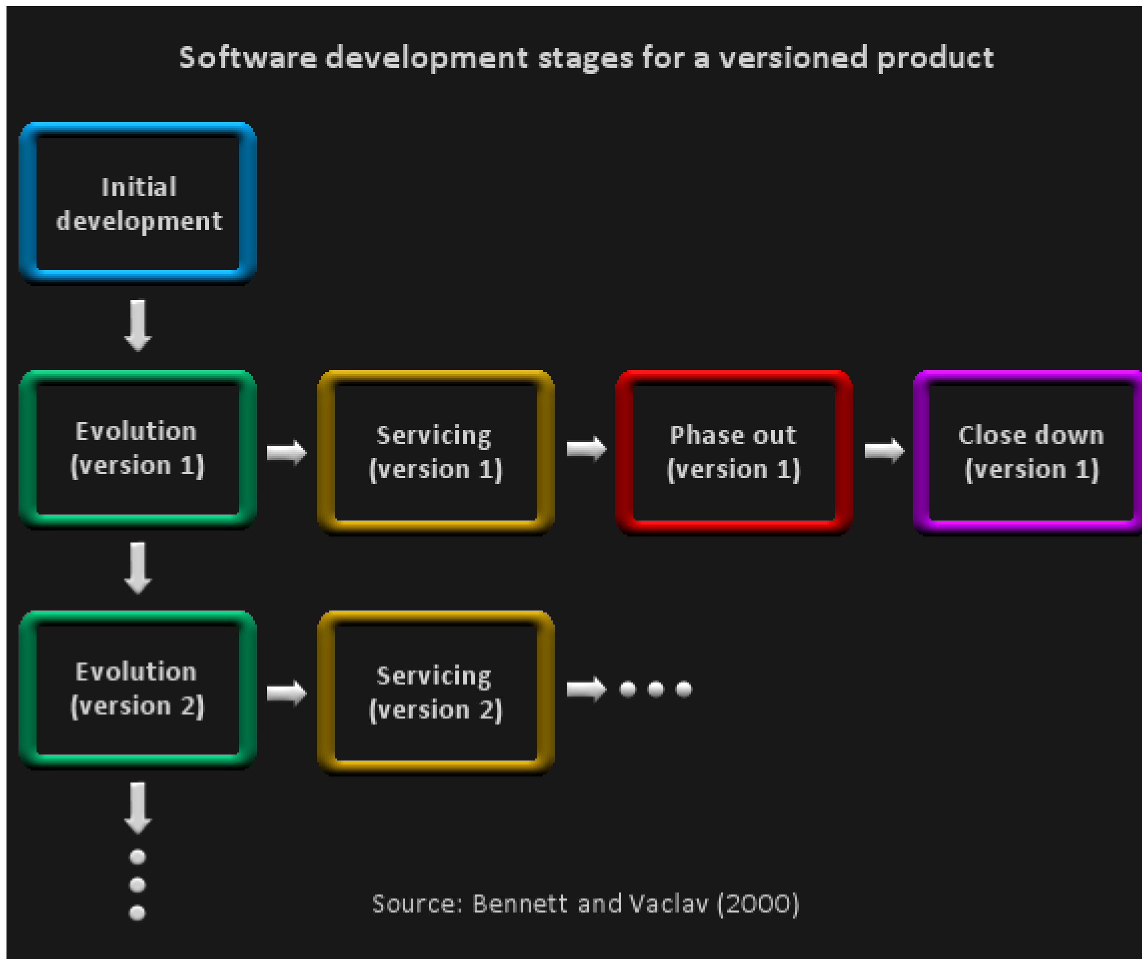
Initial development produces the first version of the product. During this stage, the architecture is determined. Also during this stage, the development team acquire a great deal of knowledge, from user requirements and understanding the application domain to the pros and cons of the architecture and familiarity with the data structures and algorithms used.

Following successful initial development, the project enters the **evolution** stage. During this period, the full range of maintenance tasks is available, to keep up with changing requirements and operating environments, and to fix defects. This depends on a combination of good software architecture and knowledge within the development team.

As changes are made to the code and the development team, the architecture and knowledge base may start to decay. If either degrades far enough, substantial changes to the software are no longer practical. The project then enters a **servicing** stage, where only small, tactical maintenance such as patching to fix a bug is viable. Bennett and Vaclav argue that this is normally a one-way transition: once a project has degenerated and entered servicing, it is no longer possible to return to effective evolution.

Eventually, a project may reach a point where it is no longer actively maintained at all, but is still in operation (**phase out**). At this stage, users must work around known bugs and limitations for themselves. Finally, some projects may be retired completely (**close down**).

The same staging principle holds for projects that release a string of major versions over an extended period, such as most off-the-shelf software: the main development path continues to evolve and produce new versions, but branches for older versions eventually diverge to the point where they enter their own servicing stage, and ultimately die out.



These staged models have powerful implications for how we develop software: good maintainability is not just about making changes easier and developers more productive, it is about whether making substantial changes is even possible. To borrow Ward Cunningham's brilliant metaphor, if we accumulate too much technical debt, we can become technically bankrupt. Then the only way back is to start over, which incurs heavy penalties.

To avoid this fate, so that a project can continue to evolve for as long as we need it to, we must preserve both the

integrity of the software architecture and the knowledge base available to the development team.

Principal factors affecting maintainability

Now that we've looked at both the short-term and long-term views of software maintenance, we can identity some important contributory factors for maintainability:

Readability

We have seen that understanding relies primarily on program comprehension, drawing information from the source code itself. It is therefore important that the code present useful information as clearly as possible.

Preservation of knowledge

Understanding is reinforced by the knowledge base built up by a development team over the course of a project and particularly during the initial development. Teams tend to lose knowledge over time, unless steps are taken to preserve it, through effective communication, good tools, and good documentation.

Testability

Testing takes many forms: unit testing, integration testing, high-level functional tests, code inspections… To these we add regression testing once we begin maintenance. Again, any steps we can take to make testing faster or more reliable will enhance maintainability.

Modifiability

It almost goes without saying that any steps we can take to make designs more adaptable and ease the mechanical implementation of changes will improve maintainability.

This is not intended to be an exhaustive list. In particular, it focuses on technical factors affecting the code, and to some extent on the documentation and tools that directly support development. Organisational and project management issues are for another day.

Summary

1. Why does it matter?

- Maintenance commonly uses the majority of a software project's development resources. A 3:1 ratio of maintenance to initial development work is not unusual.

2. Reasons for performing maintenance

- Maintenance can be classified as perfective, adaptive, corrective or preventive.
- Around 75% of maintenance work is responding to changing requirements and operating environment. Most of the rest is bug-fixing.
- Only 1 in 5 development groups surveyed made a point of checking maintainability during QA, and just 3% of maintenance time is spent improving maintainability.

3. The small-scale maintenance process

- There are mostly non-technical differences between maintenance and initial development at the start and the end of the process.
- The technical steps are similar: understanding, modification and testing.
- Understanding includes an extra “isolation” step during maintenance, to identify where in existing code changes need to be made.
- Testing methods will now include regression testing.
- Program comprehension is a vital underlying activity.

4. Information sources during maintenance

- The code itself is the primary source of information for developers.
- Consulting other developers is also common.
- Database tools such as bug trackers are valuable secondary sources.
- Developers distrust low-level documentation, but some other types can be useful.

5. A larger scale: the laws of software evolution

- Programs undergo continuous change or become progressively less useful.
- Program complexity tends to increase over time unless actively controlled.

6. Stages in a software project's life

- Projects pass through up to five stages:
    1. initial development
    2. evolution
    3. servicing
    4. phase out
    5. close down.
- During evolution, a full range of maintenance work can be performed.
- During servicing, maintenance is restricted to local, tactical changes.
- Once a project enters its servicing stage, further evolution is not possible.
- The key factors in extending the evolution stage are integrity of the software architecture and preservation of the developers' knowledge base.

7. Principal factors affecting maintainability

- Important technical factors affecting maintainability include:
    - readability
    - preservation of knowledge
    - modifiability
    - testability.
- To improve productivity and quality during maintenance, and to avoid moving to servicing prematurely, it is important to support all four of these areas.

FORMAL VERIFICATION

In the context of hardware and software systems, **formal verification** is the act of proving or disproving the correctness of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics[citation needed].

Contents

- 1 Usage
- 2 Approaches to formal verification
- 3 Validation and Verification
- 4 Industry usage
- 5 See also
- 6 References

Usage

Formal verification can be helpful in proving the correctness of systems such as: cryptographic protocols, combinational circuits, digital circuits with internal memory, and software expressed as source code.

The verification of these systems is done by providing a formal proof on an abstract mathematical model of the system, the correspondence between the mathematical model and the nature of the system being otherwise known by construction. Examples of mathematical objects often used to model systems are: finite state machines, labelled transition systems, Petri nets, timed automata, hybrid automata, process algebra, formal semantics of programming languages such as operational semantics, denotational semantics, axiomatic semantics and Hoare logic.[citation needed]

[edit] Approaches to formal verification

There are roughly two approaches to formal verification.[citation needed]

The first approach and formation is model checking, which consists of a systematically exhaustive exploration of the mathematical model (this is possible for finite models, but also for some infinite models where infinite sets of states can be effectively represented). Usually this consists of exploring all states and transitions in the model, by using smart and domain-specific abstraction techniques to consider whole groups of states in a single operation and reduce computing time. Implementation techniques include state space enumeration, symbolic state space enumeration, abstract interpretation, symbolic simulation, abstraction refinement. The properties to be verified are often described in temporal logics, such as linear temporal logic (LTL) or computational tree logic (CTL).

The second approach is logical inference. It consists of using a formal version of mathematical reasoning about the system, usually using theorem proving software such as a HOL theorem prover, the ACL2, Isabelle, or Coq theorem provers. This is usually only partially automated and is driven by the user's understanding of the system to validate. Recent tools such as Perfect Developer and ArC attempt to automate the proof process fully.

Validation and Verification

It has been suggested that this article or section be mergedVerification and Validation

Verification is one aspect of testing a product's fitness for purpose. Validation is the complementary aspect. Often one refers to the overall checking process as V & V.

- **Validation**: "Are we trying to make the right thing?", i.e., is the product specified to the user's actual needs?
- **Verification**: "Have we made what we were trying to make?", i.e., does the product conform to the specifications?

The verification process consists of static/structural and dynamic/behavioral aspects. E.g., for a software product one can inspect the source code (static) and run against specific test cases

(dynamic). Validation usually can be done only dynamically, i.e., the product is tested by putting it through typical and atypical usages ("Does it satisfactorily meet all use cases?"). Verification and Validation.

Software maintenance in software engineering is the modification of a software product after delivery to correct faults, to improve performance or other attributes..

this section describes the six software maintenance processes as:

1. The implementation processes contains software preparation and transition activities, such as the conception and creation of the maintenance plan, the preparation for handling problems identified during development, and the follow-up on product configuration management.
2. The problem and modification analysis process, which is executed once the application has become the responsibility of the maintenance group. The maintenance programmer must analyze each request, confirm it (by reproducing the situation) and check its validity, investigate it and propose a solution, document the request and the solution proposal, and, finally, obtain all the required authorizations to apply the modifications.
3. The process considering the implementation of the modification itself.
4. The process acceptance of the modification, by confirming the modified work with the individual who submitted the request in order to make sure the modification provided a solution.
5. The migration process (platform migration, for example) is exceptional, and is not part of daily maintenance tasks. If the software must be ported to another platform without any change in functionality, this process will be used and a maintenance project team is likely to be assigned to this task.
6. Finally, the last maintenance process, also an event which does not occur on a daily basis, is the retirement of a piece of software.

There are a number of processes, activities and practices that are unique to maintainers, for example:

- Transition: a controlled and coordinated sequence of activities during which a system is transferred progressively from the developer to the maintainer;
- Service Level Agreements (SLAs) and specialized (domain-specific) maintenance contracts negotiated by maintainers;
- Modification Request and Problem Report Help Desk: a problem-handling process used by maintainers to prioritize, documents and route the requests they receive;
- Modification Request acceptance/rejection: modification request work over a certain size/effort/complexity may be rejected by maintainers and rerouted to a developer.

A common perception of maintenance is that it is merely fixing bugs. However, studies and surveys over the years have indicated that the majority, over 80%, of the maintenance effort is used for non-corrective actions (Pigosky 1997). This perception is perpetuated by users submitting problem reports that in reality are functionality enhancements to the system.

Software maintenance and evolution of systems was first addressed by Meir M. Lehman in 1969. Over a period of twenty years, his research led to the formulation of eight Laws of Evolution (Lehman 1997). Key findings of his research include that maintenance is really evolutionary developments and that maintenance decisions are aided by understanding what happens to systems (and software) over time. Lehman demonstrated that systems continue to evolve over time. As they evolve, they grow more complex unless some action such as code refactoring is taken to reduce the complexity.

The key software maintenance issues are both managerial and technical. Key management issues are: alignment with customer priorities, staffing, which organization does maintenance, estimating costs. Key technical issues are: limited understanding, impact analysis, testing, maintainability measurement

As with every branch of science and engineering, software engineering requires both technical skills as well as managerial skills. Project management not one of the toughest tasks to accomplish. Maintenance especially needs a lot of management skills in order to enhance product equality, easy maintainability and hence the customer satisfaction. It is always a trusted principle of expecting

'repeat-business' from established clients rather that looking for newer customers.

Change Request Form"

Software maintenance requires tracking and control of activities. Problem reports have to be tract, recorded and solved accordingly.

This is accomplished using a systematic approach towards maintenance. When a change is necessary in the product, the activity occurs in response to a form filled by the end user of the product. This form is termed Change Requires Form.

A change request maybe feature enhancement, UI changes or an error fixation.

When a change request is made it is processed in the following systematic fashion:

Initiate software change request

Analyze request

Check foe valid requests

If (not a valid request) **then**

Close the request

Else

Submit the request to the change control board for analyze and acceptance

If (request approved by change control board) **then**

Make changes in code based on requests with constraints specified by the change control board. Perform tests foe the new changes and hence the system integrity Submit the new changes to the change control board

If (change control board approves the new changes) **then**

Update the master storage of the software with the new changes.

Update external documentations and user manuals

Make the changes known to everyone and the end users.

Else make changes as per the board's instructions and re-submit

Else Close the request **Endif.**

The change request form is submitted by the user. It is reviewed by the analyst. If the analyst finds the problem is not an error or irrelevant to the software product then he discusses with the user and then close the request. If the problem is potentially huge then the form if analyzed, fixes proposed and resources estimated and the problem is brought to the notice of the change control board.

Sometimes the problem might be a small error or a simple add-on to the UI screens. In such case the analyst provides immediate temporary fixes or makes fresh changes to the code with the aid of maintenance programmers.

Change Control Board: (CCB)

The Change Control Board is a group of technical personnel who review the change requests from the users and propose fixes. Any change to the software product has to be first approved by the board before it is implemented.

Usually change control board refers to the software configuration control. However there are several other separate boards in larger projects such as program CCB, Subsystem CCB, Software CCB and so on.

Once a change request is approved by the CCB then the problem is fixed by the maintenance programmers with the assistance from the analysts.

The composition of CCB and the hierarchy depends upon the kinds of project in the progress. For a large project the board may comprise several managers both from technical side as well as the management side and also senior business analysts who posses in-depth knowledge about the system.

The general members of the CCB may be:

- Quality Assurance Manager
- Configuration Manager
- Senior System Engineer
- Test Engineers

The major advantage of solving user requests by the above method through the change request board is that the maintenance programmers are shielded from the end users and only the analysts interact with them.

Change Request Summaries:

As change requests are submitted and fixes provided, they are recorded and summaries or logs are generated.

The maintenance activities are summarized on weekly or monthly basis.

These reports provide details on the various change requests, fixes performed for this requests, emergency problems if any and so on.

The reports are done a periodic basis which enable the management to keep in track of the versatility of the software product. The summaries also provide details on the number of open requests and the closed requests for each period during maintenance phase.

The summaries can also be used to generate TREND PLOTS to obtain a graphical representation of the maintenance activities.

Such a plot can be done to determine the number of open and closed requests and the frequency of bugs in the software product.

Source Code Metrics

SOFTWARE PROCESS
AND
PROJECT METRICS

Software process and project metrics are quantitative measures that enable **software** engineers to gain insight into the efficiency of t he **software** process and the projects conducted using the process fr amework.

In **software** project management, we are primarily concerned wi th productivity and quality metrics. The four reasons for measuring **s oftware** processes, products, and resources (to characterize, to evalu ate, to predict, and to improve).

Software metrics refers to a broad range of measurements for co mputer **software**. Measurement can be applied to the **software** proce ss with the intention of improving it on a continuous basis. Measure ment can be used throughout the **software** project to assist in estima tion, quality control, etc

Within the context of **software** project management, we are con cerned primarily with productivity and quality metrics.

Measures, Metrics, and Indicators

Measure: provides a quantitative indication of the size of some product or process attribute

Measurement: is the act of obtaining a measure

Metric: is a quantitative measure of the degree to which a syste m, component, or process possesses a given attribute.

Indication: is a metric or combination of metrics that provide in sight into **software** process, a **software** project or the product itself.

Metrics in the process and project domains

Measurement is common place in the engineering world. Metric s should be collected so that process and product indicators can be as certained.

Process indicators enables a **software** engineering organization t o gain insight into the efficacy of an existing process. ie Process indicators enable **software** project managers to: assess project status, track potential risks, detect problem area early, adjust workflow or tasks, and evaluate team ability to control product quality

Process Metrics

Private process metrics (e.g. defect rates by individual or modul e) are known only to the individual or team concerned.

Public process metrics enable organizations to make strategic ch anges to improve the **software** process.

Metrics should not be used to evaluate the performance of indivi duals.

Statistical **software** process improvement helps and organization to discover where they are strong and where are week.

Process Metrics

- Number of requests for corrective **maintenance**
- Average time required for impact analysis
- Average time taken to implement a change request
- Number of outstanding change requests
- If any or all of these is increasing, this may indicate a decline in maintainability

Project Metrics

- **Software** process metrics are used for strategic purpose, project measures are tactical. Project metrics, and the indicators derived from them are used by a PM and a **software** team to adapt project work flow and technical activities

- **Software** project metrics are used by the **software** team to adapt project workflow and technical activities.
- Project metrics are used to avoid development schedule delays, to mitigate potential risks, and to assess product quality on an on-going basis.
- Every project should measure its inputs (resources), outputs (deliverables), and results (effectiveness of deliverables).

Software Measurement

- Measurement in the physical world can be categorized into two: Direct and Indirect. **Software** metrics can be categorized similarly.
- Direct measures of the **software** engineering process include cost and effort applied. Direct measures of the product includes lines of code produced, execution speed, memory size and defects reported over some set period of time.
- The cost and effort required to build the **software**, the number of code produced and other direct measures are relatively easy to collect as long as specific conventions for measurement are established in advance.

Software Measurement

- Indirect measures of the product include functionality, quality, complexity, efficiency, reliability, maintainability, and many other "…abilities".
- Indirect measures are more difficult to assess

Size Oriented Metrics

- Derived by normalizing (dividing) any direct measure (e.g. defects or human effort) associated with the product or project by LOC.
- Size oriented metrics are widely used but their validity and applicability is widely debated.

Function Oriented Metrics

- Function oriented metrics use a measure of the functionality delivered by the application as a normalization value. Since functionality cannot be measured directly, it must be derived indirectly using other direct measures
- Function points are computed from direct measures of the information domain of a business **software** application and assessment of its complexity.
- Once computed function points are used like LOC to normalize measures for **software** productivity, quality, and other attributes.

Extended Function Point Metrics

- Function point measure was originally designed to be applied to business information system application
- Feature points and 3D function points provide a means of extending the function point concept to allow its use with real-time and other engineering applications.
- The relationship of LOC and function points depends on the language used to implement the **software**.

Other Maintenance Tools and Techniques.

Software maintenance as we have seen earlier is an important part of software engineering. Hence the role of a software engineering. Hence the role of a software maintainer is vital in the course of the product development and maintenance. A few tasks that are done by a maintainer are:

1. Negotiation with users
2. Analysis of change requests
3. Recommend solutions for the change control board
4. Involve in redesign, modification of software
5. Updating documents
6. Train users on the software product
7. Perform configuration control and provide temporary fixes.

In many organizations, the software maintainer gets necessary inputs and assistance from business analysts as well as technical heads in performing the above mentioned tasks. Technical writers

as handy in supporting software maintainer by updating the required documents and document and manuals.

Apart from this, there are a few automated tools that assist the maintainer in performance his job more effectively and efficiently. There are two types of support tools
Technical support tools and managerial support tools
A few examples of tools assisting technical support for the maintainer are:

- Text editors
- Debuggers
- Cross-reference generators
- Linkage editors
- Comparators
- Complexity metrics calculators
- Version control systems

Text Editors:

Text editors provide easy creation of programs, data and documents, edit them, and save them for future purposes. There are several general-purpose editors used for creating documents and programs. Integrated Development Environment [IDE] is a feature which many popular languages support with options ranging from general text editing to rich text editing, search options, program execution and debugging, comment and indent code for much better readability and so on. Examples of text editors are VI, WORD, PAGEMAKER.

Debuggers provide methods to set breakpoints in a program and hence enable the user to test the part of code that is suspected to be improper or ‘troublesome’. Debugging aids provide traps, memory dumps, traces and history files to aid the developers in detecting errors in code. Certain IDE programs provide in-built debuggers with ‘Watch Windows’ and ‘Immediate Windows’ to enable the user to find the value of a variable under consideration or test a condition. Just-In-Time debuggers are another useful debugging tools.

Cross-Reference generators:

These provide cross-reference listings for procedures, functions, and data references. The details provided by these cross-reference generators are vital in many cases to detect, fix bugs and hence maintain the software. They typically provide information on local variables, statement usages, who calls whom, no of times a routine is being called, function names etc.

Linkage editors:

When there are several modules of compiled object code, a linker links them together and hence generates an executable code. This also forms an essential tool for the maintenance programmers in selectively compiling and then linking those modules alone into the software system. Certain routine in the software may be modified during the maintenance period. Such modules alone have to be selectively compiled and linked into the system. Here the linkage editor comes handy to the maintainer.

Comparator:

This tool is used to compare two files and the generate a listing of the differences between them. Comparators are used to compare different versions of a program or test case or documents. This provides details regarding what change in the code produced the desired result and also the adverse effects of a particular piece of code. An example of such a comparator is WINDIFF.

Complexity metric calculators:

Complexity metrics can be calculated by various methods such as McCabe's cyclometric equation. Automated tools use such equations in determining the complexity of a code and hence also determine the change in complexity due to a modification in the source code.

Version control systems:

Version control, releases and configuration management systems are used to track the activities during the lifecycle of a software product. They keep track of versions of modules, releases of modules made on the respective dates, data and author of code.

They also provide backups of master source code at regular intervals. The usage of version control systems in important in a large software project comprising of several developers who work on the code simultaneously the central database provides an easy of means of controlling versions, preventing loss of code, keep history of code for easy testing as well as further enhancements of code and so on. An example of such a system is the Visual Source Server.