Yashwantrao Chavan
Maharashtra
Open University
[Established by Government of Maharashtra]

CMP 205

Software Engineering

Software Engineering

Writer : Shri. Shirish Deshpande

Yashwantrao Chavan Maharashtra Open University

Vice-Chancellor : Prof. E. Vayunandan

SCHOOL OF COMPUTER SCIENCE : SCHOOL COUNCIL

Writer

Shri. Shirish Deshpande
Shriguru Prasad
No. 4, Anandbhuvan Society
Dhankawadi
Pune

Editor

Prof. Shriram Zade
Dr. Moonje Institute of Management and Computer Studies
Gangapur Road,
Nashik

Coordinator

Mr. Madhav Palshikar
Sr. Lecturer
School of Computer Science
Y.C.M. Open University
Nashik

Production

Shri. Anand Yadav
Manager,
Print Production Centre
Y. C. M. Open University, Nashik - 422 222

(First edition developed under DEC development grant)

■ **First Publication :** June 2010 ■ **Reprint :** Sept. 2017

■ **Typesetting & Cover Design :** Om Computers, Nashik - 422 007

● **Printed by :** Shri. Narendra Shaligram, M/s. Replica Printers, 2, Chitco Centre, Wakilwadi, Nashik 422001

● **Published by :** Dr. Dinesh Bhonde, Registrar, Y. C. M. Open University, Nashik - 422 222

AG17-118 Software Engineering (CMP 205)

Lesson 1

Software Engineering and Models

Overview

Software Engineering has gained a phenomenal importance in the recent years in the System Development Life Cycle. Many learned people have worked on the topic and provided various techniques and methodologies for effective and efficient testing. Today, even though we have many books and articles on software test engineering, people are fallacious in understanding the underlying concepts of the subject.

Software Engineering Made Easy is aimed at bringing the technicalities of Software Engineering into one place and arriving at a common understanding. We wanted to bring out a base knowledge bank where software engineers can start to learn the science and art of Software Engineering, Software engineering is layered technology. It comprises of a quality focus, process, methods and tools. When you build a product or system it undergoes a series of steps/stages, a road map that helps you to create a timely, high quality product. This is achieved following the software process. Software Process comes under Umbrella activity of Software Quality Assurance (SQA) and Software Configuration Management (SCM). Software models are part of real world surrounding us, like banking or the airline booking. This world around us changes over time. So the software has to change too. It has to evolve together with the changing reality.

Learning Objectives

- What is Software Engineering , its scope , content and importance
- What are the software problems
- To know the phases in development. Of software
- Types of Software Applications.
- Software Engineering Code of Ethics
- To understand the Software Engineering Approach
- To understand Characteristics of Software Process
- To understand Definition of Software Process
- To understand Software Life Cycle Model
- To understand Software Development Process

Introduction

"Knowledge is of two kinds: we know a subject ourselves, or we know where we can find information upon it."
-Samuel Johnson

Software is playing an increasingly important and central role in all aspects of daily life—in government, banking and finance, teaching, transportation, entertainment, medication, farming, and law. The number, size, and application domains of programs being developed have grown dramatically; as a result, billions of dollars are being spent on software development, and the livelihood and lives of millions directly depend on the effectiveness of this development. Unfortunately, there are severe problems in the cost, timeliness, and quality of many software products; even more serious is the effect that quality problems can have on the safety-critical elements of software that directly affect the health and welfare of humans.

What is Software Engineering and Why is it Important?

In various texts on this topic, one encounters a definition of the term software engineering. Software Engineering defined in the first NATO conference as (Definition by Fritz Bauer "Software engineering is the establishment use of the sound engineering principles in order to obtain economically software i.e. reliable and works efficiently on real machines."

IEEE standard Definition of Software Engineering

Software engineering is application of systematic, disciplined, quantifiable approach to development, operation and maintenance of software i.e. application of engineering to software.

Importance of software engineering

Software engineering is an engineering approach for software development. We can alternatively view it as a systematic collection of past experience. The experience is arranged in the form of methodologies and guidelines. A small program can be written not including using software engineering principles. But if one wants to develop a huge software product, then software engineering principles are crucial to achieve a excellent quality software cost effectively. These definitions can be elaborated with the help of a building construction analogy.

Suppose you have a companion who asked you to build a small wall as shown in fig. 1.1. You would be able to do that using your common sense. You will get building materials like bricks; cement etc. and you will then build the wall.

Fig. 1.1: A Small Wall

But what would happen if the same friend asked you to build a large multi-storeyed building as shown in fig. 1.2

Fig. 1.2: A Multi-storeyed Building

You don't have a very good idea about building such a huge complex. It would be very difficult to extend your idea about a small wall construction into constructing a large building. Even if you tried to build a large building, it would collapse because you would not have the requisite knowledge about the power of resources, testing, planning, architectural design, etc. Building a small wall and building a large building are entirely different ball games. You can use your intuition and still be successful in building a small wall, but building a large building requires knowledge of civil, architectural and current engineering principles.
Without using software engineering principles it would be difficult to develop large programs. In industry it is usually needed to develop large programs to accommodate multiple functions. A problem with developing such huge business-related programs is that the difficulty and difficulty levels of the programs increase exponentially with their sizes. For example, a program of size 1,000 lines of code has some complexity. But a program with 10,000 LOC is not just 10 times more difficult to develop, but may as well turn out to be 100 times more difficult unless software engineering principles are used. In such situations software engineering techniques come to rescue. Software engineering helps to reduce the programming complexity. Software engineering principles use two important techniques to reduce problem complexity: abstraction and decomposition.

Evolution of Software Engineering

The software industry has evolved through 4 eras, 50's –60's, mid 60's –late 70's, mid 70's-mid 80's, and mid 80's-present. Each period has its personal distinctive characteristics, but over the years the software's have increased in dimension and complexity. Several problems are common to almost all of the eras and are discussed below.

Software Problem

Many industry observers have defined problems associated with software development as a "crisis". The impact of some of the spectacular software failures have occurred over the past decade. It is true that software people succeed more often than they fail. The problem is lasting long time or recurring often. The set of problems that are encountered in the software development is not limited to software that doesn't function properly

The Software Crisis dates back to the 1960's when the primary reasons for this situation were less than acceptable software engineering practices. In the early stages of software there was a lot of interest in computers, a lot of code written but no established standards. Then in early 70's a lot of computer programs started failing and people lost confidence and thus an industry crisis was declared. Various reasons leading to the disaster(crisis) included:

Hardware advances outpacing the capability to build software for this hardware.
The ability to build in pace with the demands.
Increasing dependency on software's
Struggle to build reliable and high quality software
Poor design and insufficient resources.

This crisis though identified in the early years, exists to date and we have examples of software failures around the world. Software is basically considered a failure if the project is terminated because of costs or overrun schedules, if the project has experienced overruns in excess of 50% of the original or if the software results in client lawsuits. Some examples of failures include failure of Aviation traffic control systems, failure of medical software, and failure in telecommunication software. The primary reason for these failures is due to bad software engineering practices accepted. Some of the worst software practices include:

No historical software-measurement data.
Rejection of accurate cost estimates.

Failure to use automated estimating and planning tools.

Excessive, irrational schedule pressure and creep in user requirements.

Failure to monitor progress and to perform risk management.

Failure to use design reviews and code inspections.

To avoid these failures and thus improve the record, what is needed is a better understanding of the process, better estimation techniques for cost, time and quality measures. But the question is, what is a process? Process transform inputs to outputs i.e. a product. A software process is a set of activities, methods and practices relating revolution that people use to develop and maintain software.

At present a large number of problems exist due to a chaotic software process and the occasional success depends on individual efforts. Therefore to be able to deliver successful software projects, a focus on the process is essential since a focus on the product alone is likely to miss the scalability (that can be expanded to meet future needs) issues, and improvements in the existing system. This focus would help in the predictability of outcomes, project trends, and project characteristics.

The process that has been defined and adopted needs to be managed well and thus process management comes into play. Process management is concerned with the knowledge and management of the software process, its technical views and also ensures that the processes are being followed as expected and improvements are shown.

From this we conclude that a set of defined processes can possibly save us from software project failures. But it is nonetheless important to note that the process alone cannot help us avoid all the problems, because with changeable conditions the need varies and the process has to be adaptive to these varying needs. Importance needs to be given to the human aspect of software development since that only can have a huge impact on the results, and effective price and time estimations may go totally scrap if the human resources are not planned and managed efficiently. Secondly, the reasons mentioned related to the software engineering principles may be resolved when the needs are correctly identified. Correct identification would then make it easier to identify the best practices that can be applied because one process that might be suitable for one organization may not be most suitable for another.

Therefore to make a successful product a combination of Process and Technicalities will be required under the umbrella of a well-defined process.

Software engineering concerns the construction of large program.

Traditional programming techniques and tools are primarily aimed at supporting programming in the small. It holds programming language, tools (flowcharts) and methods (structured programming).present day software development projects result in systems containing large number of (inter-related) programs.

The central theme is mastering complexity.

In general, the problems are such that they can not be surveyed in their entirety. One is forced to split the problem into parts such that each separate part can be grasped while the interaction between parts remains simple. The whole complication does not increase in this way, but it does become controllable.

Software evolves.

Most software models a part of reality, such as processing request in library or tracking money transfer in the bank. If software is not to become absolute it has to evolve (change) with the reality i.e. being modeled.

The efficiency with which software is developed is of crucial importance.

Total cost and development time of software project is high. This also holds the maintenance of software.

Regular co-operation between people is an integral part of programming in large.

Since the problems are large, many people have to work concurrently at solving those problems. There must be clear arrangements for distribution of work, methods of communication, responsibilities and so on.

The software has to support its users effectively.

Software is developed to support users at work. The functionality offered should fit user's task.

Software engineering is a field in which members of one culture create artifacts.

Not only software engineers lack factual knowledge of the domain for which they develop software, they lack knowledge of its culture as well.

The above list depicts that software engineering has slots of facets. Software engineering means not only the programming, though it is an important part of software engineering. The construction of software is something different from the construction of physical products. The cost of software construction is incurred during development and not during production. Software is logical means intangible in nature (hard to visualize) rather than physical. Physical products wear out over a period of time and therefore have to be maintained. Software does not wear out but requires maintenance because of errors detected late or user changes requirement. Software reliability is determined by the manifestation of errors already present and not by physical factors such as wear and tear.

Phases In The Software Development

When building a house the builder does not start with the pilling up bricks. Rather the requirements and possibilities of the client are analyzed first, taking into account such factors as family structure, hobbies, finances and the like, the architect takes these factors into consideration when designing a house. Only after the design is agreed upon is the actual started.

Similarly when building software first, the problem to be solved is analyzed and the requirements are described in a very exact way. Then design is made based on these requirements. Finally the construction process i.e. actual programming of the solution is started.

There are distinguishable numbers of phases in the development of software. (Fig 1.1).

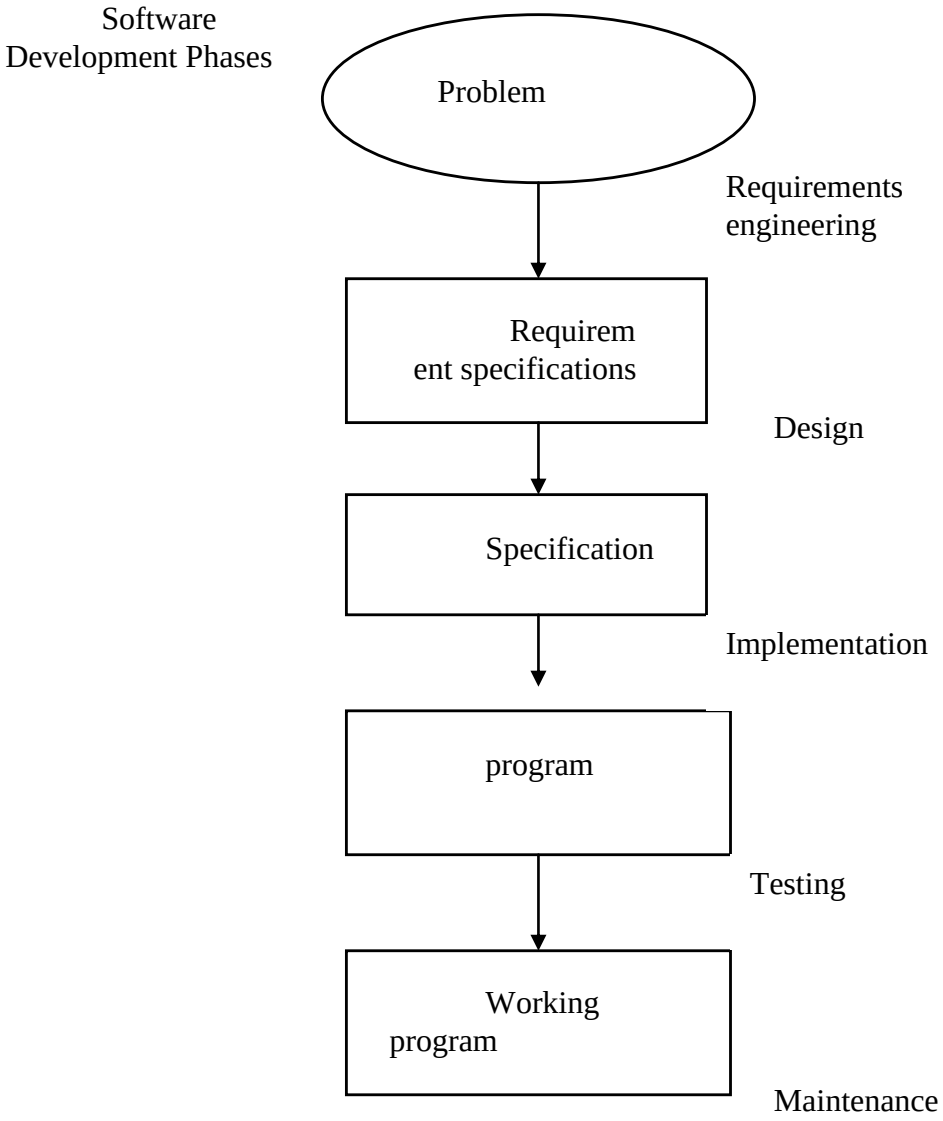


Figure 1.1 Software Development Phases

The Process model shown above is simple in nature. In real life things will be more complex. In a project all these activities (phases) are separate as shown above but they overlap each other most of the time. For example in implementation phase one part of system is designed and other parts (components) not yet designed. There is no strict linear progression between the phases (from requirement to design, from design to implementation). This happens because we have to backtrack previous phase when errors discovered or requirements changes.

Requirements Phase

This is where every software project starts. Requirements serve many purposes for a software development project. For starters, requirements define what the software is supposed to do. The software requirements serve as the basis for all the future design, coding, and testing that will be done on the project. Typically, requirements start out as high-level common statements about the software's functionality as perceived by the users of the software. Requirements are further defined through presentation, look and feel, and other criteria. Each top-level necessity is assigned to one or more subsystems within an application. Subsystem-level requirements are further refined and allocated to individual modules. As this process points out, requirements definition is not just a process that takes place at the start of a development project, but an ongoing process. This is especially true when a spiral development model is used. We must be more careful during requirement phase then more chances are there that system will meet customer/ client's expectations. The document in which the result of requirement phase is noted by technical writer is called as Software Requirement Specification (SRS).

Design Phase

In the design phase the architecture is established. This phase starts with the requirement document delivered by the requirement phase and maps the requirements into architecture. The architecture defines the components, their interfaces and behaviors. The deliverable design document is the architecture. The design document describes a plan to implement the requirements. This phase represents the ``how" phase. Details on computer programming languages and environments, machines, packages, application architecture, distributed architecture layering, memory size, platform, algorithms, data structures, global type definitions, interfaces, and many other engineering details are established. The design may include the usage of existing components. The document in which the result of design phase is noted by software engineer is called as Functional Specification Document (FSD).

Implementation Phase

In the implementation phase, the team builds the components either from scratch or by composition. Given the architecture document from the design phase and the requirement document from the analysis phase, the team should build precisely what has been requested, though there is still room for innovation and flexibility. For example, a component may be narrowly designed for this particular system, or the component may be made more general to satisfy a reusability guideline. The implementation phase deals with issues of quality, performance, baselines, libraries, and debugging. The end deliverable is the product itself. The result of implementation phase is called as executable program.

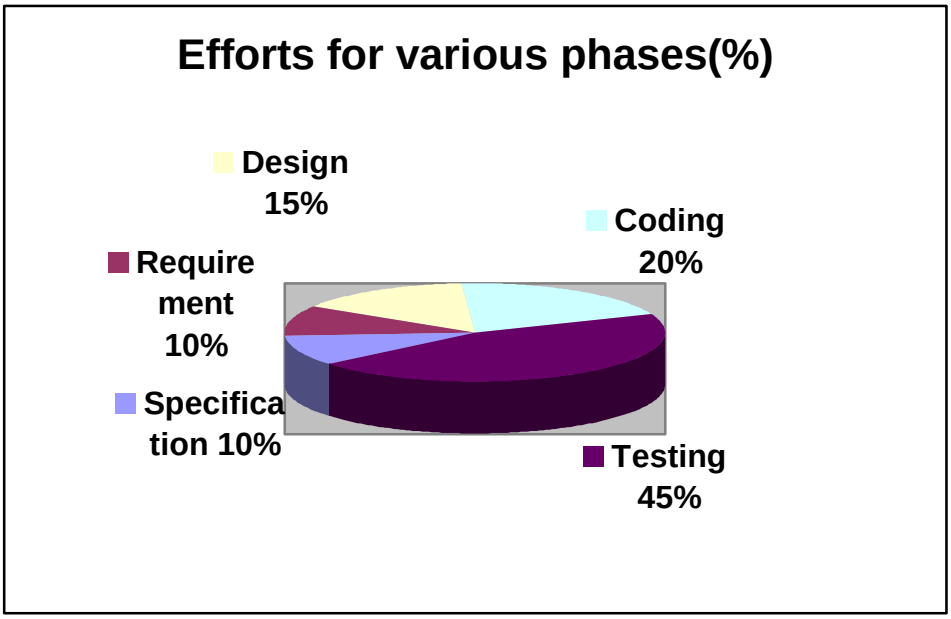
Testing Phase

The testing phase is not exactly occuring after implementation phase. That means testing phase occurs only after implementation phase is not correct. The testing phase actually starts from requirements phase. During the subsequent phases testing is continued and refined (that means testing is continuous process through the Software Development Life Cycle). The earlier the errors are detected; it is cheaper to correct them. Testing phase comprises of verification and validation. Verification means reading code(for example reviews and walkthrough). Validation means executing the code (for example unit testing, integration testing, system testing, user acceptance testing).

Maintenance Phase

After delivery of the software, there are often errors in the software that have gone undetected. These errors require to be corrected. In addition to this, the software requires changes and enhancements. All these type of changes are carried out in maintenance phase. Maintenance

phase is related to all these activities which are necessary to keep the software up and running after it has been delivered to the customer/user.



Types of Software Applications

System Software – Controls operating system behavior

It is a collection of programs to service other programs. Some System Software processes complex information structures, for example the compilers, editors etc. Other System Software processes large data, for example operating system components, drivers etc. The System Software are having heavy interaction with hardware; heavy usage by multiple users; concurrent operations which requires scheduling, resource sharing.

Real-time Software – Nuclear reactor temperature control

Software that monitors /analyzes/controls real world events as they occur is called real time. A data collecting components that collects the data, formats information from external environment, an analysis component that changes information as required by application, monitoring component that coordinates with other components so that real time software response can be maintained.

Business Software – All applications like Billing, Payroll, inventory

Business information processing is largest single software application. These software evolved as Management Information System (MIS) that access large databases containing business information. These application helps in business operation and management decision making.

Engineering and Scientific software – Simulations and statistical packages (uses input historical data) Astronomy

Computer aided design, system simulation other interactive applications have begun to take on real time and system software characteristics.

Embedded software – Cruise Controls in car , microwave oven

Intelligent products have become commonplace in every consumer and industrial market. Embedded software resides in read only memory and used to control products and systems for the consumer and industrial market.

Personal Computer software – MS Office

The personal computer software market increased in large scale in past two decades.

Web Based software- Internet based applications, E Commerce

The web pages retrieved by a browser are software that incorporates executable instructions (Perl , html, java) and data (hyper text) .

Artificial Intelligence software – Robot to develop photographic images

Artificial Intelligence software makes use of non numerical algorithms to solve complex problems that are not amenable to computation or straight forward analysis.

Software Engineering Code of Ethics

The development of complex software system involves many people such, as software developers, testers, managers, general managers, customers etc. Within this organization structure the relationship between individuals is unbalanced. For example a software developer has more knowledge about system being developed than his manager. When such type of unbalanced relationship is there then it requires trust among individuals. Such reliance provides opportunities for unethical behavior. To overcome this type of behavior IEEE and ACM has developed code of ethics.

1. PUBLIC - Software engineers shall act consistently with the public interest.
2. CLIENT AND EMPLOYER - Software engineers shall behave in a manner that is in the best interests of their client (customer) and employer consistent with the public interest.
3. PRODUCT - Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
4. JUDGMENT - Software engineers shall maintain integrity and independence in their expert judgment.
5. MANAGEMENT - Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
6. PROFESSION - Software engineers shall proceed the integrity and reputation of the profession consistent with the civic interest.
7. COLLEAGUES - Software engineers shall be reasonable to and supportive of their colleagues.
8. SELF - Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

The Software you developed affects the public. The health, safety and welfare of public is the primary concern of the code of ethics.

Software Engineering Approach

Software engineering is layered technology. It comprises of a quality focus, process, methods and tools.

A quality focus- Total quality management fosters a continuous process improvement culture and this culture leads to the development of more mature approaches to software engineering. The bedrock that supports software engineering is quality focus.

Process – It is foundation for software engineering. Software engineering method/process is the glue that holds the technology layers together and enables rational and timely development of computer software. Process defines a framework for a set of key process areas (KPAs) that must be established for effective delivery of software engineering technology. These KPAs forms the basis for management controls of software projects and establish the context in which technical methods are applied, work products (models, documents, data, reports, forms) are produced, milestones are established, quality is ensured, and change is properly managed.

Methods- Software engineering methods provide the technical how to for building software. Methods encompass a broad array of tasks that include requirement analysis, design, program construction, testing and support. Software engineering methods rely on a set of basic principles that govern each area of the technology and include modeling activities and other techniques.

Tools- Software engineering tools provide automated support for the process and methods. When tools are integrated, information created by one tool can be used by another.

Software Process

When you build a product or system it undergoes a series of steps/stages, a road map that helps you to create a timely, high quality product. This road map which you follow to achieve this product or system is called as a software process.

The process provides interaction between

- Users and designers
- Users and evolving tools
- Designers and evolving tools

Software Process gives stability and control to organization activity. Software process provides frame work for the development of high quality software.

Definition of Software Process

A software process can be defined as a set of activities, methods, practices, and transformations that people use to develop and maintain software and the associated products (for example - project plans, design documents, code, test cases, and user manuals).

Characteristics of Software Process

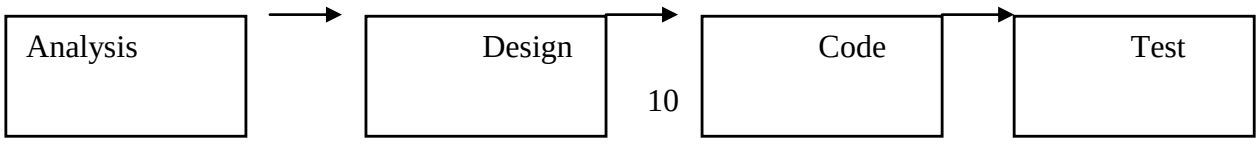
- Greater emphasis on systematic development
- Computer Aided for Software Engineering (CASE)
- A concentration on finding out the user's requirements
- Formal specification of the requirements of a system
- Demonstration of early version of a system (prototyping)
- Greater emphases on trying to ensure error free code

Software Process comes under Umbrella activity of Software Quality Assurance (SQA) and Software Configuration Management (SCM).

Software Life Cycle Model

The linear sequential model -

This is also known as Classic Life Cycle Model (or) Linear Sequential Model (or) Waterfall Method. This linear sequential model suggests a systematic, sequential approach to software development that begins at the system level and progresses through analysis, design, coding, testing and support.

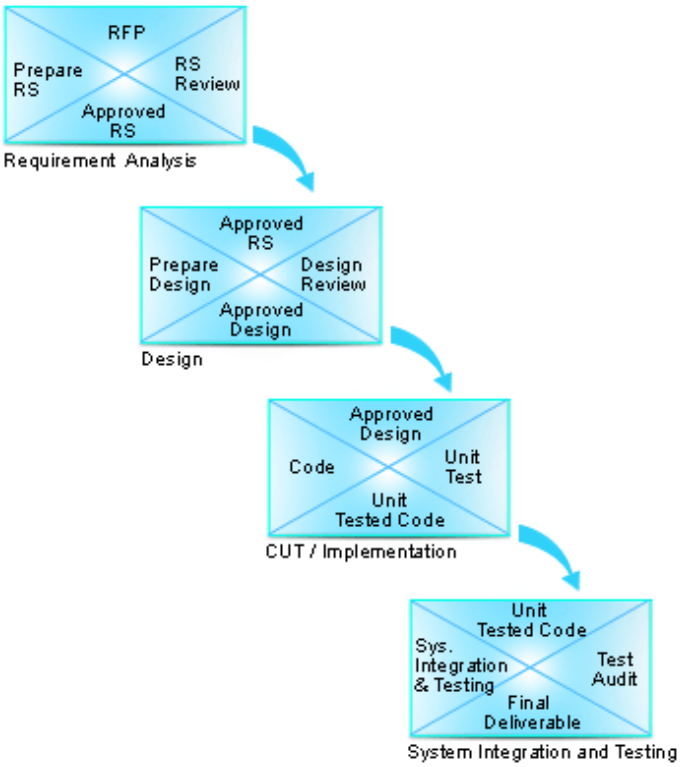


Waterfall Model

The waterfall model shows a process, where developers are to follow these steps in order:

1. Software Requirement Analysis (Requirements specification)
2. Design
3. Construction
4. Integration
5. Testing and debugging
6. Installation
7. Maintenance

System / information engineering and modeling:- The software is part of larger system (software, hardware, people and databases), works starts with requirements for all system elements and then allocating some subset of these requirements to software. System engineering and analysis encompass requirements gathering at system level with a small amount of top level design and analysis. Information engineering encompasses requirements gathering at the strategic business level and at the business area level.



Waterfall Model

After each step is finished, the process proceeds to the next step, just as builders don't revise the foundation of a house after the framing has been erected.

There is a misconception that the process has no provision for correcting errors in early steps (for example, in the requirements). In fact this is where the domain of requirements management comes in which includes change control.

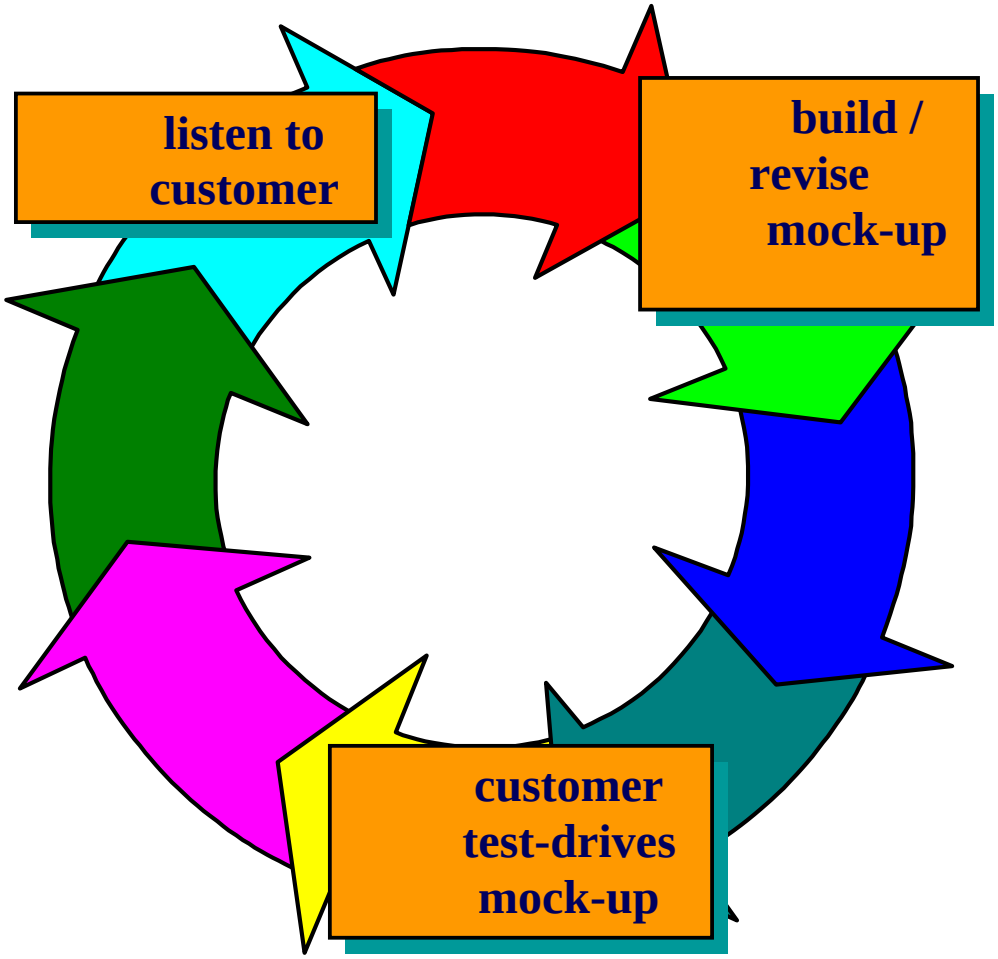
Advantages

- Simple and easy to use.
- Easy to manage due to the rigidity of the model – each phase has specific deliverables and a review process.
- Phases are processed and completed one at a time.
- Works well for smaller projects where requirements are very well understood.(For example Payroll, Accounting etc)

Disadvantages

- Adjusting scope during the life cycle can kill a project
- No working software is produced until late during the life cycle.
- High amounts of risk and uncertainty.
- Poor model for complex and object-oriented projects.
- Poor model for long and ongoing projects.
- Poor model where requirements are at a moderate to high risk of changing

Prototyping Model



Software prototyping, a possible activity during software development, is the creation of prototypes, i.e., incomplete versions of the software program being developed. A prototype typically simulates only a few aspects of the features of the eventual program, and may be completely different from the eventual implementation. The conventional purpose of a prototype is to allow users of the software to evaluate developers' proposals for the design of the eventual product by actually trying them out, rather than having to interpret and evaluate the design based on descriptions. Prototyping can also be used by end users to describe and prove requirements that developers have not considered, so "controlling the prototype" can be a key factor in the business relationship between solution providers and their clients/customer.

The process of prototyping involves the following steps

1. Identify basic requirements
   Determine basic requirements including the input and output information preferred. Details, such as security, can typically be overlooked
2. Develop Initial Prototype
   The initial prototype is developed that comprises of user interfaces.
3. Review
   The customers, including end-users, examine the prototype and provide feedback on additions or changes.
4. Revise and Enhancing the Prototype

Using the feedback both the specifications and the prototype can be improved. Negotiation about what is within the scope of the contract/product may be necessary. If changes are introduced then a repeat of steps 3 and step 4 may be required

Prototyping is especially good for designing good human-computer interfaces and on-line systems,

Advantages

Reduced time and costs: Prototyping can improve the quality of requirements and specifications provided to developers. Because changes cost exponentially more to implement as they are detected later in development, the early determination of *what the customer really wants* can result in faster and less expensive software.

Improved and increased user involvement: Prototyping requires customer involvement and allows them to see and interact with a prototype allowing them to provide better and more complete feedback and specifications. The presence of the prototype being examined by the user prevents many misunderstandings and wrong communication that occur when each side believe the other understands what they said. Since users know the problem domain better than anyone on the development team does, increased interaction can result in final product that has greater tangible and intangible quality. The final product is more likely to satisfy the users desire for look, feel and performance.

Disadvantage

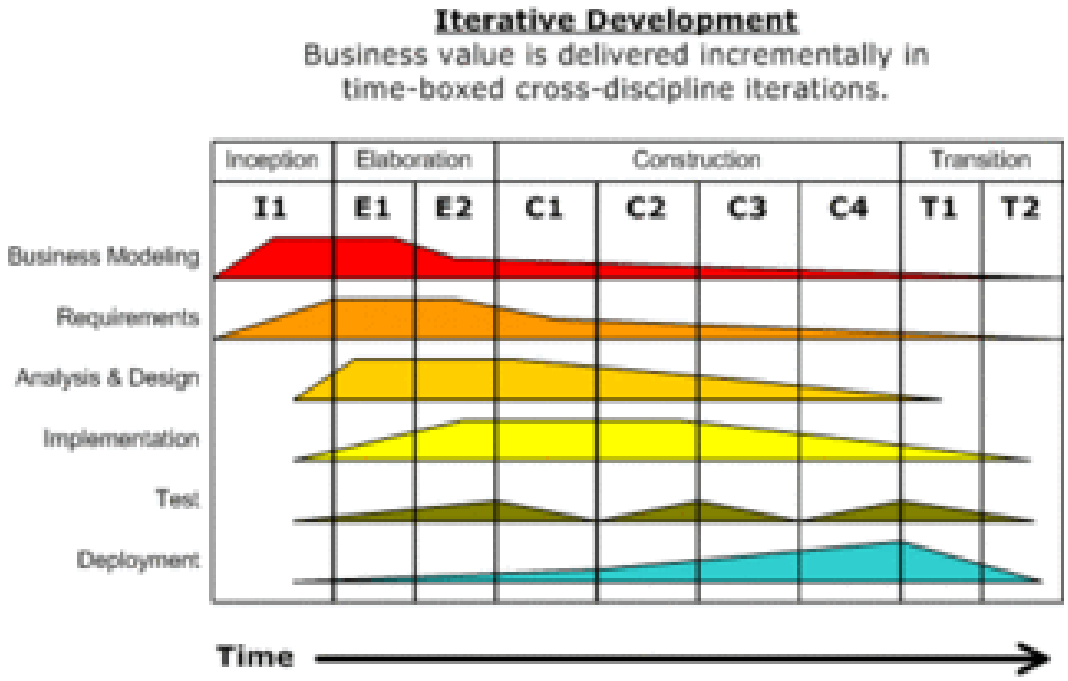
Insufficient analysis: The focus on a limited prototype can distract developers from properly analyzing the complete project. This can lead to overlooking better solutions, preparation of incomplete specifications or the conversion of limited prototypes into inadequately engineered final projects that are hard to maintain. Further, since a prototype is limited in functionality it may not scale well if the prototype is used as the basis of a final deliverable, which may not be noticed if developers are too focused on building a prototype as a model.

User confusion of prototype and finished system: Users can start to think that a prototype, intended to be thrown away, is actually a final system that merely needs to be finished or polished. (They are, for example, often unaware of the effort needed to add error-checking and security features which a prototype may not have.) This can lead them to expect the prototype to accurately model the performance of the final system when this is not the intent of the developers. Users can also become emotionally involved to features that were included in a prototype for consideration and then removed from the specification for a final system. If users are able to require all proposed features be included in the final system this can lead to conflict.

Developer misunderstanding of user objectives: Developers may assume that users share their objectives (e.g. to deliver core functionality on time and within budget), without understanding wider commercial issues. For example, user representatives attending Enterprise software (e.g. PeopleSoft) events may have seen demonstrations of "transaction auditing" (where changes are logged and displayed in a difference grid view) without being told that this feature demands additional coding and often requires more hardware to handle/leverage extra database accesses. Users might believe they can demand auditing on every field, whereas developers might think this is feature creep because they have made assumptions about the extent of user requirements. If the solution provider has committed delivery before the user requirements were reviewed/ finalized, developers are between a rock and a hard place, particularly if user management derives some advantage from their failure to implement requirements.

Developer attachment to prototype: Developers can also become emotionally involved to prototypes they have spent a great deal of effort producing; this can lead to problems like attempting to convert a limited prototype into a final system when it does not have an appropriate underlying architecture. (This may suggest that throwaway prototyping, rather than evolutionary prototyping, should be used.)

Excessive development time of the prototype: A key property to prototyping is the fact that it is supposed to be done quickly. If the developers lose sight of this fact, they very well may try to develop a prototype that is too complex. When the prototype is thrown away the precisely developed requirements that it provides may not yield a sufficient increase in productivity to make up for the time spent developing the prototype. Users can become stuck in debates over details of the prototype, holding up the development team and delaying the final product.



Iterative Development

Iterative development slices the deliverable business value (system functionality) into iterations. In each iteration a slice of functionality is delivered using cross-platform work, starting from the model/requirements through to the testing/deployment. The unified process groups iterations into phases: inception, elaboration, construction, and transition.

Inception identifies project scope, risks, and requirements (functional and non-functional) at a high level but in enough detail that work can be estimated.

Elaboration delivers a working architecture that mitigates the top risks and fulfills the non-functional requirements.

Construction incrementally fills-in the architecture with production-ready code created from analysis, design, implementation, and testing of the functional requirements.

Transition delivers the system into the production operating environment.

Each of the phases may be divided into 1 or more iterations, which are usually time-bound rather than feature-bound. Architects and analysts work one iteration ahead of developers and testers to keep their work-product backlog full.

Advantages

- Generates working software quickly and early during the software life cycle.

- More flexible – less costly to change scope and requirements.
- Easier to test and debug during a smaller iteration.
- Easier to deal with risk because risky parts are identified and handled during its iteration.
- Each iteration is an easily managed milestone.

Disadvantages

- Each phase of an iteration is rigid and do not overlap each other.
- Problems may arise pertaining to system architecture because not all requirements are collected up front for the entire software life cycle.

Waterfall vs. Iterative Development

Waterfall development completes the project-wide work-products of each discipline in a single step before moving on to the next discipline in the next step. Business value is delivered all at once, and only at the very end of the project.

Iterative Enhancement

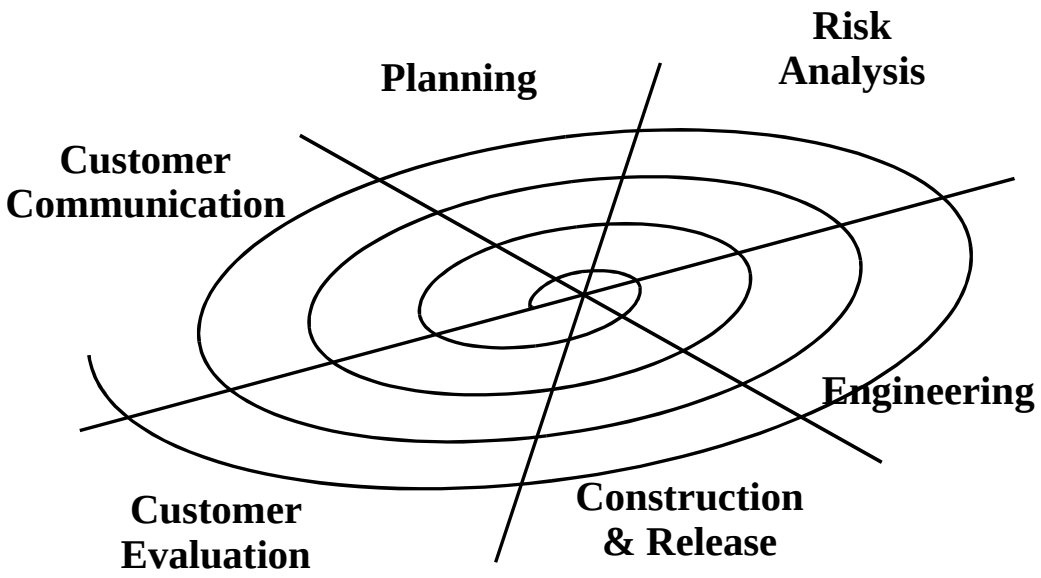
The basic idea behind iterative enhancement is to develop a software system incrementally, allowing the developer to take advantage of what was being learned during the development of earlier, incremental, deliverable versions of the system. Learning comes from both the development and use of the system, where possible. Key steps in the process were to start with a simple implementation of a subset of the software requirements and iteratively enhance the evolving sequence of versions until the full system is implemented. At each iteration, design modifications are made and new functional capabilities are added.

The Procedure itself consists of the Initialization step, the Iteration step, and the Project Control List. The initialization step creates a base version of the system. The goal for this initial implementation is to create a product to which the user can react. It should offer a sampling of the key aspects of the problem and provide a solution that is simple enough to understand and implement easily. To guide the iteration process, a project control list is created that contains a record of all tasks that need to be performed. It includes such items as new features to be implemented and areas of redesign of the existing solution. The control list is constantly being revised as a result of the analysis phase.

The iteration involves the redesign and implementation of a task from project control list, and the analysis of the current version of the system. The goal for the design and implementation of any iteration is to be simple, straightforward, and modular, supporting redesign at that stage or as a task added to the project control list. The level of design detail is not dictated by the interactive approach. In a light-weight iterative project the code may represent the major source of documentation of the system; however, in a mission-critical iterative project a prescribed Software Design Document may be used. The analysis of iteration is based upon user feedback, and the program analysis facilities available. It involves analysis of the structure, modularity, usability, reliability, efficiency, & achievement of goals. The project control list is modified in light of the analysis results.

Spiral Model

The **spiral model** is a software development process joining elements of both design and prototyping-in-stages, in an effort to combine advantages of top-down and bottom-up concepts. Also known as the spiral lifecycle model, it is SDM a systems development method used in information technology (IT). This model of development combines the features of the prototyping model and the waterfall model. The spiral model is intended for large, expensive and complicated projects.



The spiral model is divided into a number of framework activities, also known as task regions. Typically, there are six task regions. The figure depicts a spiral model that contains six task regions:

Customer communication - tasks required to set up effective communication between developer and customer.

Planning - tasks required to define resources, timelines, and additional project related information.

Risk analysis - responsibilities required to assess both technical and managing risks.

Engineering - tasks required to build one or more representations of the application.

Construction and Release - responsibilities required to construct, test , install and provide user support (e.g., documentation and training).

Customer Evaluation - responsibilities required to obtain customer feedback based on evaluation of the software representations created during the engineering stage and implemented during the installation stage.

Each of the regions is populated by a series of work tasks that are adapted to the characteristics of the project to be undertaken. For small projects, the number of work tasks and their formality is low. For larger, more critical projects, each task region contains more work tasks that are defined to achieve a higher level of formality. In all cases, the umbrella activities (e.g., software configuration management and software quality assurance) are applied.

The spiral model is used most often in large projects.

Advantages

High amount of risk analysis

Good for large and mission-critical projects.

Software is produced early in the software life cycle.

Disadvantages

A costly model

Risk analysis requires highly specific expertise.

Project’s success is highly dependent on the risk analysis phase.

Doesn’t work well for smaller projects.

Software Development Process

A software development process is a structure imposed on the development of a software product. Synonyms include software life cycle and software process. There are several models for such processes, each describing approaches to a variety of tasks or activities that take place during the process.

- **Software Requirements analysis**

The most important task in creating a software product is extracting the requirements or requirements analysis. Customers typically have an abstract idea of what they want as an end result, but not what software should do. Incomplete, ambiguous, or even contradictory requirements are recognized by skilled and experienced software engineers at this point. Frequently demonstrating live code may help reduce the risk that the requirements are incorrect. One specific method here is Software Elements Analysis.

Once the general requirements are gleaned (gathered) from the client, an analysis of the scope of the development should be determined and clearly stated. This is often called a scope document. Certain functionality may be out of scope of the project as a function of cost or as a result of unclear requirements at the start of development. If the development is done outwardly, this document can be considered a legal document so that if there are ever disputes, any ambiguity of what was promised to the client can be clarified.

Domain Analysis (subject area of the new software) is often the first step in attempting to design a new piece of software, whether it be an addition to an existing software, a new application, a new subsystem or a complete new system. Assuming that the developers and the analysts are not sufficiently knowledgeable in the subject area of the new software, the first task is to investigate the so-called "domain" of the software. The more knowledgeable they are about the domain already, the less work required. Another objective of this work is to make the analysts, who will afterward try to extract and collect the requirements from the area experts, speak with them in the domain's own terminology, facilitating a better understanding of what is being said by these experts. If the analyst does not use the proper terminology it is likely that they will not be taken seriously, thus this phase is an important prelude (introduction) to extracting and gathering the requirements. If an analyst hasn't done the appropriate work confusion may ensue: "I know you believe you understood what you think I said, but I am not sure you realize what you heard is not what I meant."

- **Specification**

Specification is the task of precisely describing the software to be written, possibly in a rigorous way. In practice, most successful specifications are written to understand and fine-tune applications that were already well-developed, although safety-critical software systems are often carefully specified prior to application development. Specifications are most important for external interfaces that must remain stable. A good way to determine whether the specifications are sufficiently precise is to have a third party review the documents making sure that the requirements and Use Cases are logically sound.

- **Architecture**

The architecture of a software system or software architecture refers to an abstract representation of that system. Architecture is concerned with making sure the software system will meet the requirements of the product, as well as ensuring that future requirements can be addressed. The architecture footstep also addresses interfaces between the software system and supplementary software products, as well as the underlying hardware or the host operating system.

- Design, **implementation** and testing

Implementation is the part of the process in that software engineers program the code for the project.

Software testing is an integral and important part of the software development process. This component of the process ensures that bugs are recognized as early as possible.

Documenting the internal design of software for the purpose of future maintenance and enhancement is done throughout development. This may also include the authoring of an API, be it external or internal.

- Deployment and maintenance

Deployment starts after the code is appropriately tested, is approved for release and sold or otherwise distributed into a production environment.

Software Training and Support is important because a large percentage of software projects fail because the developers fail to realize that it doesn't matter how much time and planning a development team puts into creating software if nobody in an organization ends up using it. People are often resistant to change and avoid venturing into an unfamiliar area, so as a part of the deployment phase, it is very important to have training classes for new clients of your software.

Maintenance and enhancing software to manage with newly found problems or new requirements can take extra time as compared with the initial development of the software. It may be necessary to insert code that does not fit the original design to correct an unforeseen problem or it may be that a customer is requesting more functionality and code can be added to accommodate their requests. It is during this phase that customer calls come in and you see whether your testing was extensive enough to uncover the problems before customers do.

Summary

Use of software is must in all sectors, as it helps not in improving the operations only but also in management decision making. Good software is that which meets customer requirement. Software Engineering is concerned with the problems that have to do with the construction of huge software. When developing huge programs it is done in phased manner. Requirements gathering and analysis then system design, program design, coding using fictional specification and then testing and last is maintenance (this is required when new evolution or requirements changes).We get the process model in which we iterate phases from time to time to refine them. We speak about the SDLC. We discussed different phases of SDLC. We explored the primary reason for Software failures. We speak about the code of ethics to be followed. Software Engineering methods enforce discipline in development process which result in best quality product get delivered.

Software engineering is layered technology. It comprises of a quality focus, process, methods and tools. When you build a product or system it undergoes a series of steps/stages, a road map that helps you to create a timely, high quality product. This is achieved following the software process. Software Process comes under Umbrella activity of Software Quality Assurance (SQA) and Software Configuration Management (SCM).. Development engineering has two approaches a) Waterfall Model b) Evolutionary (iterative) Model. Waterfall Model is used where the risk of software development is nil, when risk is high the Evolutionary (iterative) Model is used to control risk. After this the software engineer chooses the software development process model. The process models are linear, sequential, prototype, RAD, incremental development, Spiral model. The selection of process model is based on the nature of system and the proposed software solution. The core process in all models is requirement analysis, design, coding, testing.

Specialties of Different Models

Water fall Model

Advantages are Simple and easy to use, Easy to manage due to the rigidity of the model – each phase has specific deliverables and a review process., Phases are processed and completed one at a time., Works well for smaller projects where requirements are very well understood.(For example Payroll, Accounting etc)

Disadvantages are Adjusting scope during the life cycle can kill a project , No working software is produced until late during the life cycle, High amounts of risk and uncertainty, Poor model for complex and object-oriented projects, Poor model for long and ongoing projects, Poor model when requirements are changing

Prototyping Model
Advantages are Reduced time and costs, Improved and increased user involvement.
Disadvantages are insufficient analysis, User confusion of prototype and finished system, Developer misunderstanding of user objectives, Developer attachment to prototype, and Excessive development time of the prototype.

Iterative Development
Advantages are Generates working software quickly and early during the software life cycle, more flexible – less costly to change scope and requirements, Easier to test and debug during a smaller iteration, Easier to control risk because risky parts are identified and monitored during its iteration, Each iteration is an easily managed milestone.
Disadvantages are Each phase of an iteration is stiff and do not partly cover each other, Problems may arise pertaining to system architecture as not all requirements are collected up front for the entire software life cycle.

Spiral Model
Advantages are High amount of risk analysis, Good for large and mission-critical projects, Software is produced early in the software life cycle,
Disadvantages are A costly model, Risk analysis requires highly specific expertise, Project's success is highly dependent on the risk analysis phase, Doesn't work well for smaller projects.

Self Test

1. The document in which the result of ----------is noted by technical writer is called as Software Requirement Specification
    A. Requirement phase
    B. Design phase
    C. Testing phase
    D. Implementation phase
    E. Maintenance phase
2. The document in which the result of ---------- is noted by software engineer is called as Functional Specification Document
    A. Requirement phase
    B. Design phase
    C. Testing phase
    D. Implementation phase
    E. Maintenance phase
3. The result of ---------- is called as executable program.
    A. Requirement phase
    B. Design phase
    C. Testing phase
    D. Implementation phase

E. Maintenance phase

4. The testing phase starts from --------.
   A. Requirement phase
   B. Design phase
   C. Testing phase
   D. Implementation phase
   E. Maintenance phase
5. --------. is related to all these activities which are necessary to keep the software up and running after it has been delivered to the customer
   A. Requirement phase
   B. Design phase
   C. Testing phase
   D. Implementation phase
   E. Maintenance phase
6. The testing phase occurs only after implementation phase State whether this statement is true or false.
   a) True. b) False
7. Operating system components, drivers software comes under ----------
   A. System Software
   B. Real time software
   C. Business software
   D. Engineering and Scientific software

8. Billing, Payroll software comes under ----------
   A. System Software
   B. Real time software
   C. Business software
   D. Engineering and Scientific software
9. Cruise Controls in car software comes under ----------
   A. System Software
   B. Real time software
   C. Business software
   D. Embedded software

10. Verification means reading code State whether this statement is true or false.
    a) True. b) False
11. Software engineering is ----------technology
    A) Hardware
    B) Software
    C) Layered
    D) Spiral
12. Software engineering comprises of ----------
    A) Quality focus
    B) Process
    C) Methods , Tools.
    D) All of the above
13. Software engineering ----------provide the technical how to for building software
    A) Quality focus
    B) Process
    C) Methods
    D) Tools.

14. Software engineering -------- provide automated support for the process and methods.
    A) Quality focus
    B) Process
    C) Methods
    D) Tools.
15. Software engineering process is the glue that holds the technology layers together State whether this statement is true or false.
    a) True. b) False
16. Total quality management fosters a continuous process improvement culture State whether this statement is true or false.
    a) True. b) False
17. Key Process Areas (KPAs) forms the basis for management controls of software projects. State whether this statement is true or false.
    a) True. b) False
18. The software process provides interaction between
    A. Users and designers
    B. Users and evolving tools
    C. Designers and evolving tools
    D All of the above
19. Total Key Process Areas for 1 to 5 levels are –
    A) 10
    B) 6
    C) 7
    D) 18
20. The spiral model is divided into a ------ number of framework activities *or* task regions
    A) 10
    B) 6
    C) 7
    D) 4

Lesson 2

Requirement Analysis

Overview

Requirements are the agreed upon facts about what an application or system must accomplish for its users. They are largely independent of the methods or notations used to analyze, design, implement or test the target application. The process of managing requirements can determine project success and is especially important for larger systems, out-sourced projects or life critical software. Although primarily focused on software development, much of this discussion also applies to other activities like hardware development or systems engineering.

In the mid-1980s, prototyping was seen as the solution to the requirements analysis problem. Prototypes are mock-ups of an application. Mock-ups permit end-users to visualize an application which is not yet constructed. Prototypes help users get an idea of what the system will look like, and make it easier for users to decide design decisions without waiting for the system to be debeloped. Major improvements in communication between users and developers were often seen with the introduction of prototypes. Early views of applications led to fewer changes later and hence reduced overall costs considerably.

Learning Objectives

- To understand Software Requirements
- To understand Requirements Analysis and Definition
- To understand Need for SRS, Requirement Process ,Requirement gathering
- To understand Types of Requirements
- To understand Requirements analysis issues
- To understand Prototyping, Characteristics of SRS, Components of SRS
- To understand Requirements Validation, Importance of feasibility study
- To understand DFD, ERD, DD

Introduction

A typical project will have hundreds or even thousands of requirements. Identifying and specifying those requirements will take time and effort, but the project payback can be enormous since it impacts every aspect of a project as well as design, implementation, testing, end-user documentation and project management. A concrete foundation can significantly reduce project risk and increase the efficiency of the entire development..
Historical data shows that defects occurring in the requirement identification phase are less costly to correct. Requirement defects include missing, incomplete, non-essential, ambiguous, overlapping or non-implemented requirements. A methodical approach to dealing with requirements can greatly

reduce these deficiencies. Peers or customers can review printed requirements to find gaps in the understanding of what the project must accomplish.
When developers spend time on design and implementation of non-essential or defective requirements, they're not just wasting time that delays the project. The added project expand often increases future maintenance costs, slows the final product execution, complicates the user experience and makes the project more ridged and susceptible to errors during future enhancements. An accurate requirement list keeps the development effort clean and focused.

Requirements Analysis

This is where every software project starts. Requirements serve many purposes for a software development project. For starters, requirements define what the software is supposed to do. The software requirements serve as the basis for all the future design, coding, and testing that will be done on the project. Typically, requirements start as high-level general statements about the software's functionality as gathered/ collected from the users of the software. Requirements are then defined through performance, aesthetic view(look and feel), and extra criteria. Each high-level requirement is assigned to one or more subsystems within an application. Subsystem-level requirements are further refined and allocated to individual modules. As this process pointed out, requirements definition is not just a process that takes place at the beginning of a development project, but an ongoing process. This is especially true when a spiral development model is used. In engineering, a **requirement** is a singular documented need of what a particular product or service should be or do. It is most commonly used in a formal sense in systems engineering or software engineering. It is a statement that identifies a necessary attribute, capability, characteristic, or quality of a system in order for it to have value and utility to a user.

Definition
"Requirements engineering is the branch of software engineering concerned with the real world goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behavior, and to their evolution over time and across software families."

Requirements describe what the system is supposed to do using Natural language, Diagrams, Other notation.

In the classical engineering approach, sets of requirements are used as inputs into the design stages of product development. Requirements depicts what elements and functions are required for the particular project. Requirements analysis is a software engineering task that joins the gap between system level requirements engineering and software design. Requirements engineering activities result in the specification of software's operational characteristics like function, data, behavior, indicate software's interface with other system components and establish constraints that software must meet. Requirements analysis allows software engineer / analyst to refine the software allocation and build models. Requirements analysis provides the software designer with a representation of information, function and behavior that can be transformed to data, architectural, interface, component level designs. Software requirements analysis is divided into five areas of efforts

- Problem recognition
- Evaluation and synthesis
- Modeling
- Specification
- Review

. The requirements development phase may have been preceded by a feasibility study, or a conceptual analysis phase of the project. The requirements phase may be broken down into requirements elicitation (gathering the requirements from stakeholders), analysis (checking for consistency and completeness), specification (documenting the requirements) and verification (making sure the specified requirements are correct).

Need for SRS

The requirement specification is the end product of requirement engineering phase. Its purpose is to communicate results of requirement analysis to others. It serves as an anchor point against which subsequent steps can be justified. SRS is starting point for the design phase. It serves as foundation for database engineering ,hardware engineering, software engineering and human engineering. The specification joins each allocated system element. 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 of the interactions that the users will have with the software. Use cases are also known as functional requirements. In addition to use cases, the SRS also contains nonfunctional (or supplementary) requirements. Non-functional requirements are requirements which impose constraints on the design or implementation (such as performance requirements, quality standards, or design constraints).

Requirement Process

Managing requirements is a process, not an event. That process starts at the conception of a project and continues until the developed system has been discontinued and is no longer supported. Later we'll discuss the structure of a requirement entry, where one field is its Status.

The Status field indicates the state of progress of a requirement entry through its lifecycle. The Proposed state identifies a newly suggested requirement. Approved state indicates that a requirement will be implemented in the present project. Implemented state shows that the requirement has been implemented and is ready for testing. Validated state shows that the requirement is completed. Postponed state indicates that the requirement won't be addressed by the current project.

An automated tool can track the progress of each requirement through each state of its lifecycle. It should also allow you to add new states if your process requires it. For example, a big system that includes system requirements and product requirements you might require many validation states like Unit Validation, Product Validation and System Validation. If you're outsourcing design and implementation, you might want an Out Source Validation and In House Validation state.

To manage expectations and achieve your project schedule, you'll want to get most requirements nailed down very earlier in a development project. To decrease project risk, focus early prototyping efforts and detailed analysis on those aspects of a project with the biggest unknowns and most significant impact on a project's success.

Requirements are identified and refined using numerous methods including customer surveys, brainstorming sessions, marketing analysis and engineering analysis activities. Data flow diagrams and entity-relation diagrams are commonly used tools during this phase of the project. These diagrams provide a communication medium between the analyst, customer and management. They increase understanding of the problem domain and help to surface the essential requirements for the project. Various methods and notations have been developed for analyzing systems and designing software. Structured analysis and design is popular in real-time, embedded systems. Desktop applications often use object-oriented analysis and design with notations like UML. Data modeling has been used for decades to design database systems. Some diagramming techniques like DFDs and Use Cases diagrams are used during the process of discovering requirements while others like structure charts, class diagrams and object diagrams are used during design to satisfy specified

requirements. Later we'll discuss how traceability ties requirement entries to analysis diagrams, design diagrams, code files and test designs or procedures.

Requirement gathering (Elicitation)/ Fact Finding

In requirements engineering, requirements elicitation is the practice of obtaining the requirements of a system from users, customers and other stakeholders. The practice is also sometimes referred to as requirements gathering. Requirements elicitation is non-trivial because you can never be sure you get all requirements from the user and customer by just asking them what the system should do.

Requirement gathering technique comprises of

- **Interviews**
  - Structured, planned, close-ended conversations
  - Unstructured, encouraged to speak and brainstorm

- **System/documentation inspection**
  - Look at other similar systems to see if there is functionality that might want/need to incorporate
- **Paper prototypes**
  - Hand sketches of interface storyboard to determine scenarios
- **Scenarios**
  - A scenario is a way a user might use a target product to accomplish some objective
  - An easy way for the client to define functionality in an accessible way
  - Only major system functions should be explored/ explained with scenarios.
  - Example of scenarios

Buy a Product Scenario:

The customer browses the catalog and adds desired items to the shopping basket. When the customer wishes to pay the customer describes the shipping and credit card information and confirms the sale. The system checks the authorization on the credit card and confirms the sale both immediately and with a follow-up email.

Types of Requirements

1. Functional requirements
   i) Relate to the functionality of the system
2. Non-functional requirements -- non-functional properties of the system:
   ii) Quantitative constraints
      (a) Response time
      (b) Accuracy
   iii) Language/OS
   iv) Documentation/process requirements
   v) Hardware consideration
      (a) Compatibility
      (b) Interaction
   vi) Error handling
      (a) System exception
      (b) Safety considerations
      (c) Worst case scenarios
   vii) Quality issues
      (a) Reliability
         (i) Availability

(ii) Robustness

viii) Resource issues

(i) Constraints on the resources consumed by the system

1. Disk space
2. Network traffic

Problem analysis

Problem evaluation and solution is the major area of effort for requirement analysis. The analyst must define

- all data objects, evaluate the flow
- content of information,
- define and elaborate all software functions
- understand software behavior in the context of events that affect the system
- establish system interface characteristics and find additional design constraints

Each of these task helps in describing the problem so that an overall solution may be synthesized.

For example, a purchase order system is required a vendor for spare parts. The analyst finds that problem with current system include 1) unable to obtain the status of a component quickly. 2) two days required to update bin card. 3) Consolidation of orders is not possible as a result multiple orders are placed to same vendor. 4) Comparative statement (different vendors) not feasible for best price.

Once problem have been identified, the analyst determines what information is to be produced by the new system and what sort of data will be provided to the system. For example the customer desires status report of spare parts on daily basis (how many parts are consumed and remaining balance). On evaluating current problems and expected information (input and output) the analyst begins to synthesize appropriate solution. To start with the data objects, processing functions, and system behavior are defined in detail. Once this information has been framed, basic architectures for implementation are considered. A client server system would be appropriate or not. A database management system would be required or not. The process of evaluation continues until both analyst and customer feel confident that software can be adequately specified for subsequent development process. During this process the analyst creates models of the system to better understand data and control flow, functional processing, operational behavior and information content. The customer is not aware of precisely what is required. The developer may be unsure that a specific solution will fulfill function and performance. For these reasons, an alternative approach to requirement analysis, called prototyping is conducted.

Requirements analysis issues

Stakeholder issues

Details a number of ways users can inhibit (slow down) requirements gathering:

- Users do not understand what they want or users don't have a clear idea of their requirements
- Users will not commit to a set of written requirements
- Users insist on new requirements after the cost and schedule have been fixed
- Communication with users is slow
- Users often do not participate in reviews or are incapable of doing so
- Users are technically unsophisticated
- Users do not understand the development process
- Users do not understand present technology

This may lead to the situation where user requirements keep changing even when system or product development has been started.

Engineer/Developer problems/issues

Possible problems caused by engineers and developers during requirements analysis are:

- Technical personnel and end users may have different vocabularies. Consequently, they may wrongly believe they are in perfect agreement until the finished product is supplied.
- Engineers and developers may try to make the requirements fit an existing system or model, rather than develop a system specific to the needs of the client.
- Analysis may often be carried out by engineers or programmers, as compared to personnel with the people skills and the domain knowledge to understand a client's needs properly.

Structural Analysis

A large number of analysis methods have been developed. All these analysis methods are related by a set of operational principles.

- The information domain of a problem must be represented and understood.
- The functions that the software is to perform must be defined.
- The behavior of the software must be represented.
- The models that shows information function and behavior must be partitioned in a manner that uncovers detail in a layered fashion.
- The analysis process should move from essential information toward implementation detail.

By applying these principles the analyst approaches the problem systematically. The information domain is examined so that function may be understood completely. Models are used so that the characteristics of function and behavior can be communicated in a compact fashion. Partitioning is applied to reduce complexity.

There are set of guiding principles for requirement engineering.

- Understand the problem before you begin to create the analysis model. There is tendency to rush to a solution even before the problem is understood. That means developed software solves wrong problem (which is not aimed).
- Develop prototypes that enable a user to realize how human/ machine interaction will occur. As perception of the quality of software is often based on user friendliness of the interface.
- Record the origin of and the reason for every requirement. This is important for establishing traceability back to the customer.
- Use multiple views of requirements. Building data, functional and behavioral models provide the software engineer with three different views.
- Rank requirements. Tight deadlines may prevent the implementation of every software requirement. Requirement traceability matrix is created.
- Work to eliminate ambiguity. As most of the requirements are described in a natural language, there is always a chance for ambiguity the formal technical reviews can uncover and eliminate ambiguity.

A software engineer who follows these principles is more likely to develop a software application that will provide an excellent foundation for design.

Prototyping

The prototyping paradigm (model) can be close-ended or open-ended. The close-ended approach is often known as throwaway prototyping. In this approach, a prototype serves solely as a rough model of requirements. It is then discarded and the software is engineered using different model. An open-ended approach known as evolutionary prototyping, it uses the prototype as the first part of an analysis activity that will be continued into design and construction. Before determining close-ended or open-ended it is important to determine whether the system to be built is agreeable to prototyping. In general application that creates dynamic visual displays, interacts heavily with a user or demands algorithms that must be developed in an evolutionary fashion is a prototyping candidate.

Prototyping methods and tools

For effective software prototyping a prototype must be developed rapidly so that the customer may assess results and recommend changes. Rapid prototyping, three generic classes of methods and tools are available:-

- Fourth generation techniques- It encompass a broad array of database query and reporting languages, program and application generators. 4GT enable the software engineer to generate executable code quickly.
- Reusable software components- Another approach to rapid prototyping is to assemble, rather than build, the prototype by using a set of existing software components. This method will work only is a library system is developed so that components that do exist can be cataloged and then retrieved. It should be noted that an existing software product can be used as a prototype for a new, improved product.
- Formal specification and prototyping environment- A number of formal specification languages and tools have been developed as a replacement for natural language specification techniques. Now a days developers are in the process of developing interactive environments that
    - Enable an analyst to interactively develop language based specifications of a system.
    - Invoke automated tools that translate the language based specifications into executable code.
    - Enable the customer to use the prototype executable code to refine formal requirements.

However, over the next decade, though it is a useful technique, prototyping never solve the requirements problem:

- Managers, once they see a prototype, may have a hard time understanding that the finished design will not be produced for some time.
- Designers always feel compelled to use patched prototype code in the real system, because they are worried to 'waste time' starting again.
- Prototypes principally help with design decisions and user interface design. However, they can not tell you what the requirements originally were.
- Designers and end users can focus too much on user interface design and too little on producing a system that serves the business process.
- Prototypes work well for user interfaces; screen layout and screen flow but are not so useful for batch or asynchronous processes which may involve complex database updates and/or calculations.

Prototypes can be flat diagrams (referred to as 'wireframes') or working applications havingsynthesized functionality. Wireframes are made in a variety of graphic design documents, and often remove all color from the software design (i.e. use a grayscale color palette) in instances where the final software is expected to have graphic design applied to it. This helps to prevent confusion over the final visual look and feel of the application.

We must be more careful during requirement phase then more chances are there that system will meet customer/ client's expectations. The document in which the result of requirement phase is noted by technical writer is called as Software Requirement Specification (SRS).

Requirement Specification

Specification principles-
Specification may be viewed as a representation process. Requirements are represented in a manner that ultimately leads to successful implementation. The Specification principles are as under:-

- Separate functionality from implementation.
- Develop a model of the desired behavior of a system that encompasses data and the functional responses of a system to various stimuli (motivations) from the environment.
- Establish the context in which software operates by specifying the manner in which other system components interact with software.
- Define the environment in which the system operates and indicate how a highly knotted collection of agents react to stimuli in the environment (changes to object) produced by those agents.
- Create a cognitive model (relating to the process of acquiring knowledge by the use of reasoning) rather than a design or implementation model.
- A specification is always a model an abstraction of some real situation that is normally quite complex.
- The content and structure of Specification should be open to change.

These are the basic principles provides a basis for representing software requirements. There are set of guidelines to create software requirements specification.

- Representation format and content should be relevant to the problem.
- Information contained within the specification should be nested.
- Diagrams and other notational forms should be restricted in number and consistent in use.
- Representations should be revisable (should be available for update).

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

Characteristics of SRS
Software Requirements Specification (SRS) is written at the end of requirement gathering (elicitation) and analysis modeling. The SRS is written in text is narrative, supported by graphical models, system charts, and high level pseudo programs in natural English language. The SRS

should be written for all requirements small, large, simple or complex, including the obvious ones. The SRS is based on approved Requirement Definition and Document (RDD).

A good SRS document has certain characteristics that must be present to qualify it as a good. The characteristics are appended below

- Clarity
- Completeness
- Traceability
- Feasibility
- Testability
- Correctness
- Consistency
- Modifiability

Clarity: - SRS is clear when it has a single interpretation for the author, the user, the end user, the stake holder, the developer, the tester and the customer. This is possible if the language of SRS is unambiguous. Clarity can be ascertained after viewing the SRS by third party. It can be enhanced if SRS includes diagrams, models and charts.

Completeness:- SRS is complete when it is documented after

- The involvement of all types of concerned personnel.
- Focusing on all problems, goals, and objectives, and not only on functions and features.
- Correct definition of scope and boundaries of the software and system.

Structuring - labeling, maintaining minimum redundancy are the essential features to increase the modifiability of the SRS.

Traceability: - Traceability from specification to solution and from solution to specification is very important. The characteristic facilitates an understanding to effect on various functions and features, if modification is the requirement. Secondly, if an error occurs in testing, it's a cause analysis is possible through traceability. With traceability, navigating through test plan, test case, test results, design, and architecture and source code is easily possible.

Feasibility: - Requirement Definition and Document and SRS needs to be confirmed on technical and operational feasibility .SRS many times assume the use of technology is capable enough to deliver what is expected in SRS. The operational feasibility must be checked through environment checking. It is assumed that sources of data, user capability, system culture, work culture, and such other aspects satisfy the expectations of the developer. These must be confirmed before development launch.

Testability: - SRS should be written in such a way that it is possible to create a test plan to confirm whether specifications can be met and requirements can be delivered.

- Considering functional and non-functional requirements
- Determining features and facilities required for each requirements
- Ensuring that users and stake holders freeze the requirement

Correctness:- the requirements are termed as correct if they have a link to goals and objectives . the requirement and its specification through the software system should achieve the business goals and solve managerial and operational problems. A requirement and its functions and features are considered correct when it is most efficient and effective, and when it is considered valid by all concerned.

Consistency: - consistency in SRS is essential to achieve correct results across the system. This is achieved by achieved by use of standard terms and definitions, application of business rules in all functionality, use of data dictionary. If consistency is not there then wrong SRS will created and end result will be failure in delivery to customer.

Modifiability:- The need to modify occurs through the SDLC process. Requirement Definition and Document and SRS are created in a structured manner means requirement traceability matrix is

created so that it will be easy to modify whenever there is change in requirement. This helps in tracking any detail and changes made at all places.

Components of SRS (Template)

Table of Contents
Revision History
1. Introduction
1.1 Purpose
1.2 Document Conventions
1.3 Intended Audience and Reading Suggestions
1.4 Project Scope
1.5 References
2. Overall Description
2.1 Product Perspective
2.2 Product Features
2.3 User Classes and Characteristics
2.4 Operating Environment
2.5 Design and Implementation Constraints
2.6 User Documentation
2.7 Assumptions and Dependencies
3. System Features
3.1 System Feature 1
3.2 System Feature 2 (and so on)
4. External Interface Requirements
4.1 User Interfaces
4.2 Hardware Interfaces
4.3 Software Interfaces
4.4 Communications Interfaces
5. Other Nonfunctional Requirements
5.1 Performance Requirements
5.2 Safety Requirements
5.3 Security Requirements
5.4 Software Quality Attributes
6. Other Requirements
Appendix A: Glossary
Appendix B: Analysis Models
Appendix C: Issues List

Specific language-
A Software Requirements Specification helps two groups of people.

- For the user, the SRS is a clear and precise description of the functionality that the system has to offer.
- For the designer, the SRS is a starting point for the design.

However, it is not easy to serve both the groups with one and the same document. The user is in general best served by a document which speaks his language. The designer on the other hand is best served with a language in which concepts from his world are used. In practice following description techniques are often used for the Software Requirements Specification:-
1) Natural language 2) Pictures 3) Formal language
An advantage of natural language is that the specification is very readable and understandable to the user and other non professional involved. Pictures may be put to advantage in bringing across the functional architecture of the system. A formal language allows us to use tools in analyzing the requirements.

Requirements Validation (Review) - The requirements specification should reflect the mutual understanding of the problem to be solved by the prospective users and the development organization has everything been described, and has it been described properly. Validating the requirements thus means checking them for properties like correctness, completeness, ambiguity, and internal and external consistency. This involves user participation in validation process. They are the owners of the problem and they are the only ones to decide whether the requirements specification adequately describes their problem. The developer presents the requirements that have been allocated to the module. Requirements reviews are an activity started after SRS has been created. These reviews examine system requirements to ensure they are feasible and that they meet the stated needs of the user. They also verify software relationships; for example, the structural limits of how much load (e.g., transactions or number of concurrent users) a system can handle. Output from this review is a statement of requirements ready to be translated into system design.

The work products (software application) produced as consequence of requirements engineering are assessed for quality during validation process. The requirements validation process checks the SRS to ensure that total system requirements have been stated unambiguously and consistency is followed. This ensures that the work products conform to the standards established for the work process work product and the project. The review team participants are system engineers, customers, end users and other stake holders who examine the SRS checking for errors in content or interpretation, missing information, inconsistency, unachievable requirements. Mainly it is verified that completeness, correctness, traceability, testability, consistency with the help of requirements review checklist. To get stable SRS requirements review checklist should be reviewed continuously. It should be reviewed by independent /third party. Any defects/ findings should be documented and tracked for resolution and accordingly SRS should be modified. Major features are changed then it should be reviewed for its technical and operational feasibility.

Importance of feasibility study

There is a technical committee, which studies the SRS to check that it is economical, technical, and operational. If a project is seen to be feasible from the results of the study, the next logical step is to proceed with it. The research and information uncovered in the **feasibility study** will support the detailed planning and reduce the research time.

There are five common factors (TELOS) of feasibility study which are discussed below:-

Technology and system feasibility
The assessment is based on an outline design of system requirements in terms of Input, Processes, Output, Fields, Programs, and Procedures. This can be quantified in terms of volumes of data, trends, frequency of updating, etc. in order to estimate whether the new system will perform adequately or not.

Economic feasibility
Economic analysis is the most frequently used method for evaluating the effectiveness of a new system. More commonly known as cost/benefit analysis, the procedure is to determine the benefits and savings that are expected from a candidate system and compare them with costs. If benefits outweigh costs, then the decision is made to design and implement the system.

Legal feasibility
Determines that the proposed system comply with legal requirements. e.g. a Information Processing system must comply with the Data Protection Acts.

Operational feasibility
Is a measure of how well a proposed system solves the problems, and takes advantages of the opportunities identified during scope definition and how it satisfies the requirements identified in the requirements analysis phase of system development.

Schedule feasibility
A project will fail if it takes too long to be completed before it is useful. That means estimating how much time the system will take to develop, and if it can be completed in a given time frame using some methods like payback period. Schedule feasibility is a measure of how reasonable the project timetable is. Given our technical expertise, are the project deadlines reasonable? Some projects are initiated with specific deadlines. You need to determine whether the deadlines are mandatory or desirable.

The major focus is on feasibility

- it is economical to reduce business risk
- it is technical to reduce software solution risk
- it is operational to achieve customer satisfaction

In the requirement review formal check list is used to examine each requirement. A subset of checklist questions appended below:

Clarity questions
- Are the requirements written in non-technical understandable language?
- Are there any requirements that could have more than one interpretation?

Completeness questions
- Is there a table of contents?
- Are all figures, tables, and diagrams labeled?

Consistency questions
- Are there any requirements describing the same object that conflict with other requirements with respect to terminology?
- Are there any requirements describing the same object that conflict with respect to characteristics?

Traceability questions
- Are all requirements traceable back to a specific user need?
- Are all requirements traceable back to a specific source document or person?

Modifiability questions
- Is the requirements document clearly and logically organized?
- Does the organization adhere to an accepted standard?
- Is there any redundancy in the requirements?

This review checklist ensures that the validation team (review team) has done everything possible to conduct a thorough review of each requirement. That means any error/defect is not migrated in software solution.

Data Flow Diagrams (DFDs)

Data flow design is divided into two stages. First stage a logical design is derived in the form of a set of data flow diagrams. In this stage structured analysis is done. In second stage the logical

design is changed into a program structure charts. The second stage is known as structural design. The combination is referred to as structured analysis/ structural design.

Structured analysis can be viewed as a proper requirements analysis method in so far it addresses the modeling of some universe of discourse. It should be noted that, as data flow diagrams are refined, the analyst performs an implicit functional decomposition of the system as well. At the same time, the diagram refinements result in corresponding data refinements. The analysis process thus has design aspects as well.

Structured design, being a strategy to map the information flow contained in data flow diagrams into a program structure, is a genuine component of the design phase. The main result of structured analysis is a series of data flow diagrams. Four types of data entity are distinguished in these diagrams

External entities are the source or destination of a transaction. These entities are located outside the domain considered in a data flow diagram. External entities are indicated as squares.

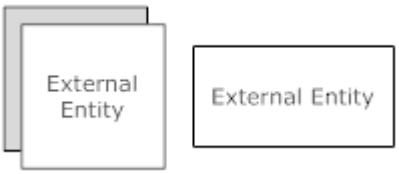
Processes transform data. Processes are denoted by circles.

Data flows between processes, external entities and data stores. A data flow is indicated by an arrow. Data flows are paths along which data structures travel.

Data stores lie between two processes. This is indicated by the name of data store between two parallel lines. Data stores are places where data structures are stored until needed.

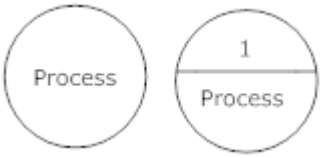
Data Flow Diagram Notations
You can use Data Flow Diagram notations on your data flow diagrams as per *Yourdon & Coad.*

External Entity Notations



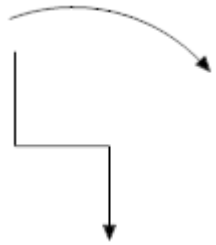
External Entity
External entities are objects outside the system, with which the system communicates. External entities are sources and destinations of the system's inputs and outputs.

Process Notations



Process
A process transforms incoming data flow into outgoing data flow.

Dataflow Notations



Dataflow
Data flows are pipelines through which packets of information flow. Label the arrows with the name of the data that moves through it.

Data store Notations

Data Store
Data stores are repositories of data in the system. They are sometimes also referred to as files.

Developing a Data Flow Diagrams (DFDs)

- The system designer make "a context level DFD" or Level 0, which shows the "communication" (data flows) among "the system" (represented by one process) and "the system environment" (represented by terminators).
- The system is "subdivided in lower level (Level 1)" DFD into a set of "processes, data stores, and the data flows along with these processes and data stores".
- Each process is then decomposed/ divided further into an "even lower level diagram containing its sub processes".
- This approach "then continues on the subsequent sub processes", until a required and sufficient level of detail is reached which is called the primitive process.

We will illustrate the various process steps of SA/SD (structured analysis/ structural design) by analyzing and designing a simple library automation system. The system allows the library clients to borrow and return the books. It also reports library management about how the library is used by the clients.

At the highest level we draw a context diagram. A context diagram is a data flow diagram with one process, denoting the system. Its main purpose is to depict the interaction of the system with the environment (the collection of external entities). For our simple library system this is done in Figure A. (Context diagram for library system) This diagram has yet to be supplemented by a description of the structure of both the input and output to the central process.

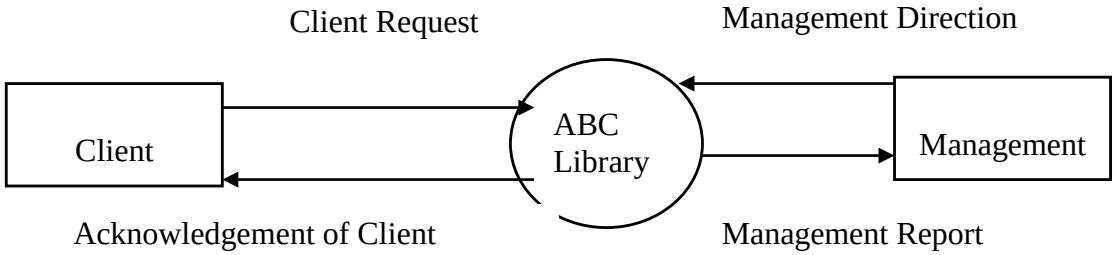


Figure A Context level DFD for Library automation (Level 0 DFD)

Next this, top level diagram is further decomposed. This example, could guide to the data flow diagram; of Figure B. in this diagram we have expanded the central process node of the context diagram. A client request is first analyzed in a process labeled 'preliminary processing'. As a result, one of the 'borrow title' or 'return title' is activated. Both these processes update a data store labeled 'catalog administration'. Client requests are logged in a data store 'log file'. This data store is used to produce management reports.

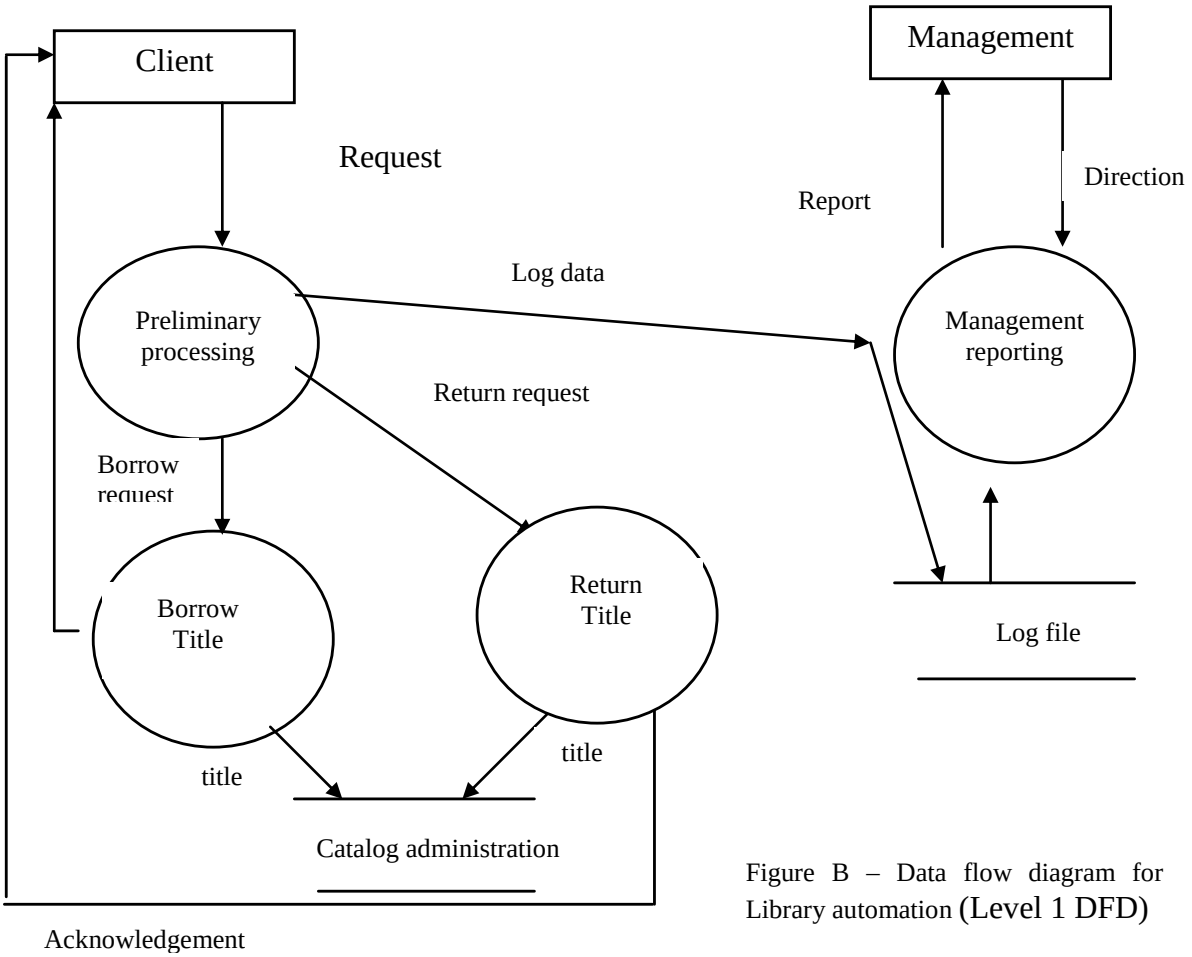


Figure B – Data flow diagram for Library automation (Level 1 DFD)

For more complicated applications, various diagrams could be drawn; one for each sub system can be identified. These subsystems in turn are further decomposed into diagrams at lower levels. We thus get a hierarchy of diagrams. As an example a possible refinement of the 'preliminary processing' node is shown in Figure C. In the lower level diagrams also; the external entities are usually omitted.

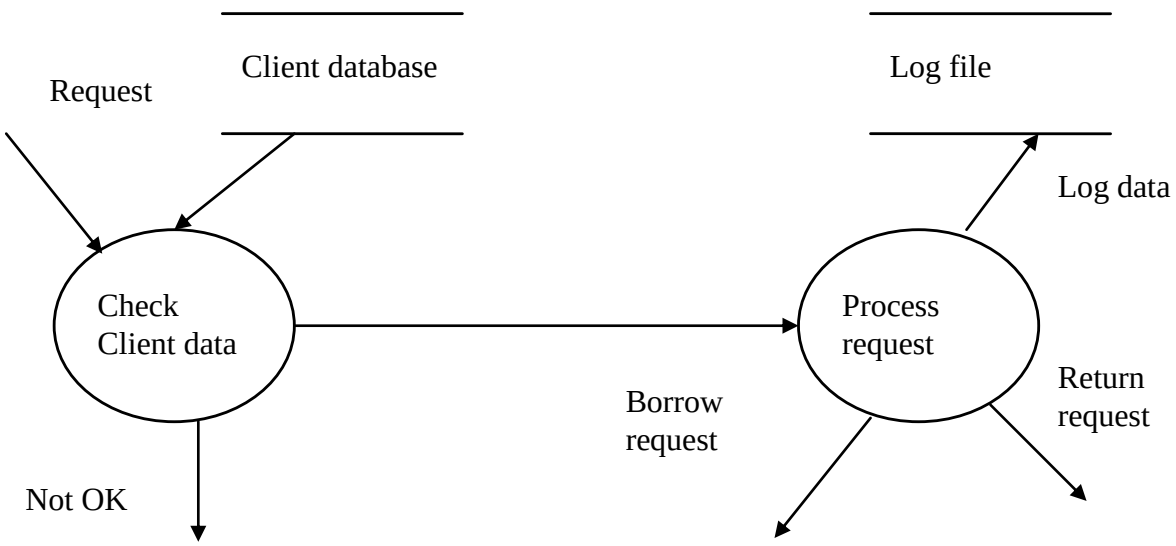


Figure C DFD for 'preliminary processing' (Level 2 DFD)

The top down decomposition stops when a process becomes sufficiently straight forward and does not warrant further expansion. These primitive processes are described in mini-specs. A mini-spec serves to communicate the algorithm of the process to relevant parties. It may use notations like structured natural language, pseudo code or decision tables. Example screen layouts can be added to illustrate how the input and output will look. An example of mini-spec for the process named 'process request' is given below:-

Identification: Process request
Description:

1. Enter type of request
    a. If invalid issue a warning and repeat step 1
    b. If step 1 has been repeated five times terminate the transaction
2. Enter book identification
    a. If invalid issue a warning and repeat step 2
    b. If step 2 has been repeated five times terminate the transaction
3. Log the client identification request type and book identification

The contents of the data flows in DFD are recorded in a data dictionary. Though this name suggests something grand, it is nothing more than a precise description of the structure of the data. This is often done in the form of regular expressions. For example data dictionary entries are as appended below:-

Borrow-requests = client id + book id
Return request = client id + book id
Log data = client id + [borrow| return] + book id

Book id = author-name + title + (ISBN) + [proc |series |other]

Conventions: [] means include one of the enclosed options; + means addition and; () means enclosed items are optional; options are separated by |

Nowadays, the static aspects of the data tend to be modeled in ER diagrams.

The result of structured analysis is a logical model of the system. It consists of a set of DFDs, amplified by descriptions of its constituents in the form of mini-specs, formats of data stores and so on. In the subsequent structured design step, data flow diagrams are changed into a collection of modules that identify another module and pass data. The result of the structured design step is expressed in a hierarchical set of structure charts. There are no strict rules for this step. Text books on the data flow technique give guidelines, and sometimes even well defined strategies, for how to get from a set of data flow diagrams to a hierarchical model for the implementation. These guidelines are strongly inspired by the various notations, most notably cohesion and coupling.

The major heuristic involves the choice of the top level structure chart. Many data processing systems are essentially transform-centered. Input is read and possibly edited, a major transformation is done, and the result is output. One way to decide upon the central transformation is to trace the input through the data flow diagram until it can no longer be considered input. The same is done, in the other direction, for the output. The bubble in between acts as the central transform. If we view the bubbles in the DFD as beads, and the data flows as threads, we obtain the corresponding structure chart by picking the bead that corresponds to the central transformation and shaking the DFD. The processes in the data flow diagram become the modules of the corresponding structure chart denote module calls, whereas the arrows in a data flow diagram denote the flows of the data from A to B is often realized through a call of B to A. sometimes it is difficult to select one central transformation. In that case, a dummy root element is added and the resulting input-process-output scheme is of the form depicted in figure.

Because of the transformation orientation of the structure chart, the relations between modules in the graph have a producer-consumer character. One module produces a stream of data which is then consumed by another module. The control flow is one whereby modules call subordinate modules so as to realize the required transformation. There is potentially complex stream of information between producer and consumer. The major contribution of structured design is found in the guidelines that aim to reduce complexity of the interaction between modules. These guidelines concern the cohesion and coupling criteria.

Entity Relationship Diagrams (ERD)

> "Description or analogy used to visualize something that cannot be directly observed"
> *Webster's Dictionary*

Data Model

Relatively simple representation of complex real-world data structures

At the time of a relational database designing, very first an entity-relationship diagram is drawn and developed to understand the requirements of the database and its processing. Drawing an entity-relationship diagram helps in understanding of an organization's data needs and can serve as a *schema* diagram for the required system's database. A schema diagram is diagram that show the structure of the data in a database. All systems analysis and design methodologies enclose entity-relationship diagramming as an important part of the methodology and it is drawn using CASE (Computer Aided Software Engineering) tools. An entity-relationship diagram could provide

as the basis for the design of the files in a usual file-based system as well as for a schema diagram in a database system.

The details of how to draw the diagrams differ from one method to another, but they all have the same basic fundamentals: entity types, attributes and relationships These three categories are required to model the essentially static data-based parts of any organization's information processing requirements.

Entity Types

An *entity type* is type of object that we wish to store data information. Which entity types you decide to include on your diagram depends on your application. In an accounting application for a business you would accumulate data about customers, suppliers, products, invoices and payments and if the business manufacturing then the products, you would need to store data about materials and production steps. Each of these would be classified as an entity type because you would want to store data information about each one. In an entity-relationship diagram an entity type is shown as a box. In Fig. 2.1, an entity type is Customer . Each entity type is shown once. There may be many entity types in an ER diagram. The *name* of an entity type is singular , it represents a type.

An entity type is comprises of a *set* of objects. For this cause people use the alternative term entity set. An entity is simply one member or element or instance of the type or set. So an entity is one individual within an entity type. For example, within the entity type CUSTOMER, D Sameer might be one entity. He is an individual entity within the type of the type 'customer'.

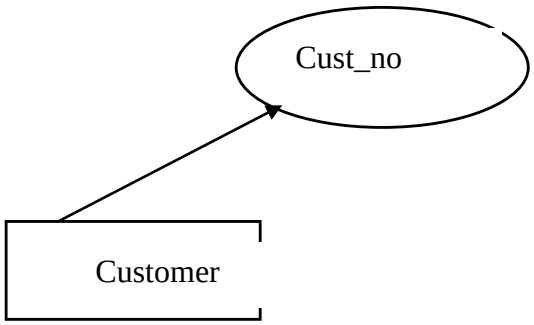


Fig. 2.1 An entity type CUSTOMER and one attribute Cust_no

Attributes

The data information we want to maintain about each entity inside an entity type is contained in attributes. An attribute is some feature about the entities that we are interested in and want to hold on the record. In fact we store the value of the attributes on the record. Each entity within the entity type will have the same set of attributes, but in general different *attribute values*. For example the value of the attribute ADDRESS for a customer D Sameer in a CUSTOMER entity type might be '10 Sea face rd " whereas the value of the attribute 'address' for another customer D Milind might be '2 Club resorts'.

There will be the same number of attributes for each entity within an entity type. That is one of the characteristics of entity-relationship modeling and relational database management. We store the same type of attributes about each entity within the entity type.

Primary Key

Attributes can be shown on the entity-relationship diagram in an oval shape. In Fig. 2.1, one of the attributes of the entity type CUSTOMER is shown. It is up to you which attributes you want to show on the diagram. In many cases an entity type may have more attributes. There is often not space on the diagram to show all of the attributes, but you might choose to show an attribute that is used to identify each entity from all the others in the entity type. This attribute is known as the **primary key.** Sometimes you might need more than one attribute in the primary key to identify the entities.

Refer Fig. 2.1 In this ,the attribute Cust_no.is shown. Assuming the organization storing the data information confirms that each customer is allocated a different cust_no, that attribute could act as the primary key, since it identifies each customer; it distinguishes each customer from all the rest. No two customers have the same value for the attribute cust_no. Some people would say that an attribute is a candidate for being a primary key because it is `unique'. They mean that no two entities within that entity type can have the same value of that attribute. In practice it is best not to use that word because it has other connotations.

As earlier briefed, you may need to have a group of attributes to construct a primary key. For example if the organization is using the CUSTOMER entity type they are not allocating a unique customer number to their customers, then it is necessary to use a composite key,(combination of attributes) for example one consisting of the attributes SURNAME and INITIALS together, to differentiate between customers with common surnames such as Joshi.

An Object Oriented Analysis, it is considered as an substitute to entity-relationship modeling, focuses on this difference between object and type, making it clear that it is possible for an item to be both an object (instance, entity) and a type (class, entity type) at the same time. There is generally no problem in coping with this in entity-relationship modeling provided the modeler makes clear what he or she means. In above explained example we have seen that by placing simple well-selected attribute on the entity-relationship diagram helps to clarify uncertainty. It is an important skill set of the systems analyst and database designer to be able to recognize and control such ambiguities where they arise. Cautious naming of entity types is another device to enhance clarity and reduce uncertainty.

Fig. 2.3(a) uses the card file and individual cards within it as being similar to an entity type and an entity respectively. In Fig. 2.3(b) the set - element model is used to show the same thing, and in Fig.2.3(c) the entity-relationship model for the same situation is shown. These are three different models of the same incident. Notice that the entity-relationship model version does not explicitly show individual entities. You are meant to know that 'within' the entity type CUSTOMER there are many customer entities.

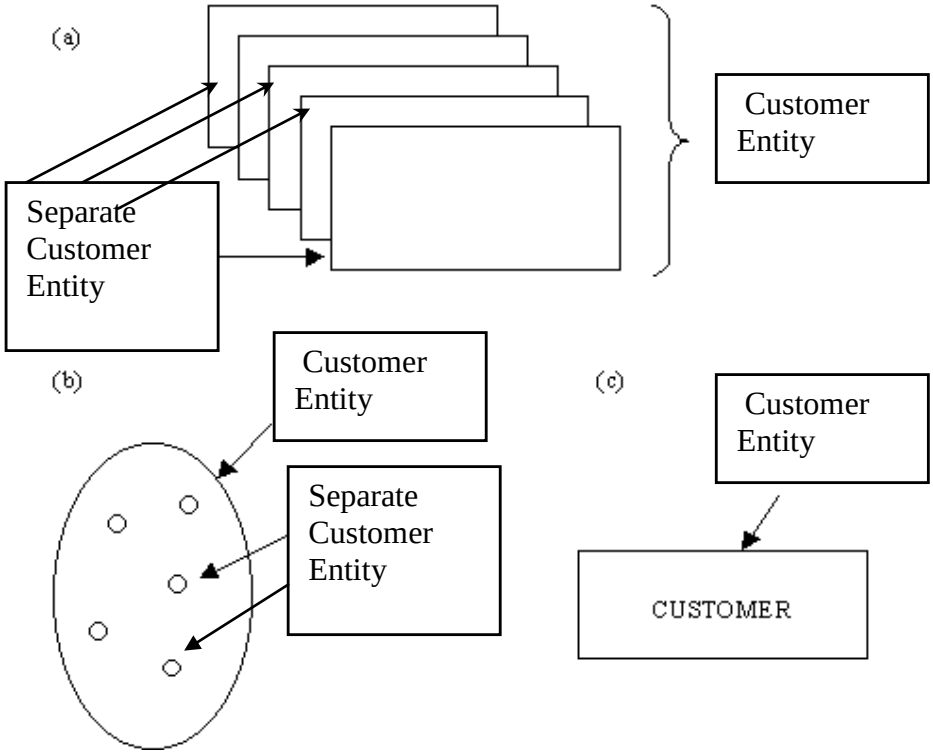


Fig. 2.3 Three types of thinking of an entity type.

The primary key helps as an identifier and serves as the method of representing relationships between entities. The primary key becomes a foreign key in all those entity types to which it is related in a one to one or one to many relationship type.

Relationship Types

The first two main elements of entity-relationship diagrams are attributes and entity types. The final element is the relationship type. Sometimes, the word 'types' is not used and relationship types are referred as 'relationships'. As there is a difference between the terms, it is recommended that to use the term relationship type.

The entities have relationships between them, and relationships between entities on the entity-relationship diagram are shown where appropriate. An entity-relationship diagram consists of a network of entity types and connecting relationship types. A relationship type is a named association between entities. Every entity has individual relationships of the type between them. An individual person (entity) occupies (relationship) an individual house (entity). In an entity-relationship diagram, this is generalized into entity types and relationship types. The entity type PERSON is related to the entity type HOUSE by the relationship type OCCUPIES. There are lots of individual persons, lots of individual houses, and lots of individual relationships linking them.

There can be more than one type of relationship between entities. For an example of three different relationship types between two entity types see Fig. 2.1. shows a single relationship type 'Received' and its inverse relationship type sent between the two entity types CUSTOMER and INVOICE. It is very important to name all relationship types. The reader of the diagram must know what the relationship type means and it is up to you the designer to make the meaning clear from the relationship type name. The direction of both the relationship type and its inverse should be shown to help clarity and immediate readability of the diagram.

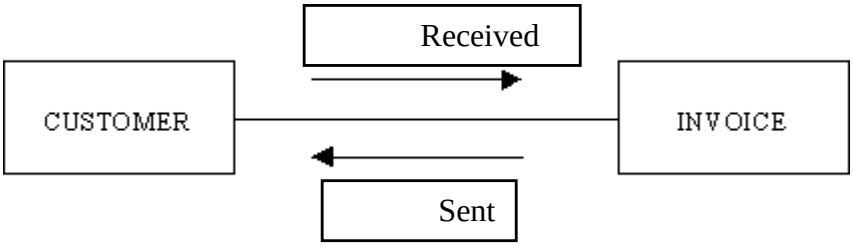


Fig. 2.4 A relationship on an entity-relationship diagram.

In the development of a database system, many people will be reading the entity-relationship diagram and so it should be immediately readable and totally clear-cut When the database is implemented, application programmers and query writers, will start using the entity-relationship diagram. Misunderstanding of the model may result in many lost man-hours going wrong tracks. There is less harm in adding redundant information into your entity-relationship model. Get user to explain entity-relationship model to you ,then you will know how clear it is.

In Fig. 2.4 what is being 'said' is that customers *received* invoices and invoices sent to customers. How many invoices a customer might have received (the max number and the min number) and how many customers an invoice might have been sent to, is shown by the *degree* of the relationship type. The 'degree' of relationship types is defined below.

In Fig. 2.5 three different ways of illustrating the existence of a relationship type are shown. In (a), in which the CUSTOMER and INVOICE entity types are represented by index cards, it can be seen that there is a `received' relationship type between customer number 4 and invoice numbers 1 and 2. Customer number 4 has `received' these two invoices. These two invoices sent to customer number 4. In (b) the same information is shown by means of notation with the relationship type `received' and inverse relationship type sent to linking customer entities and invoice entities. Fig. 2.5(c) is the entity-relationship diagram version and information about individual entities and which entity is linked to which is lost. The reason for this is simply that in a real database there would be hundreds of customer and invoice entities and it would be impossible to show each one on the entity-relationship diagram.

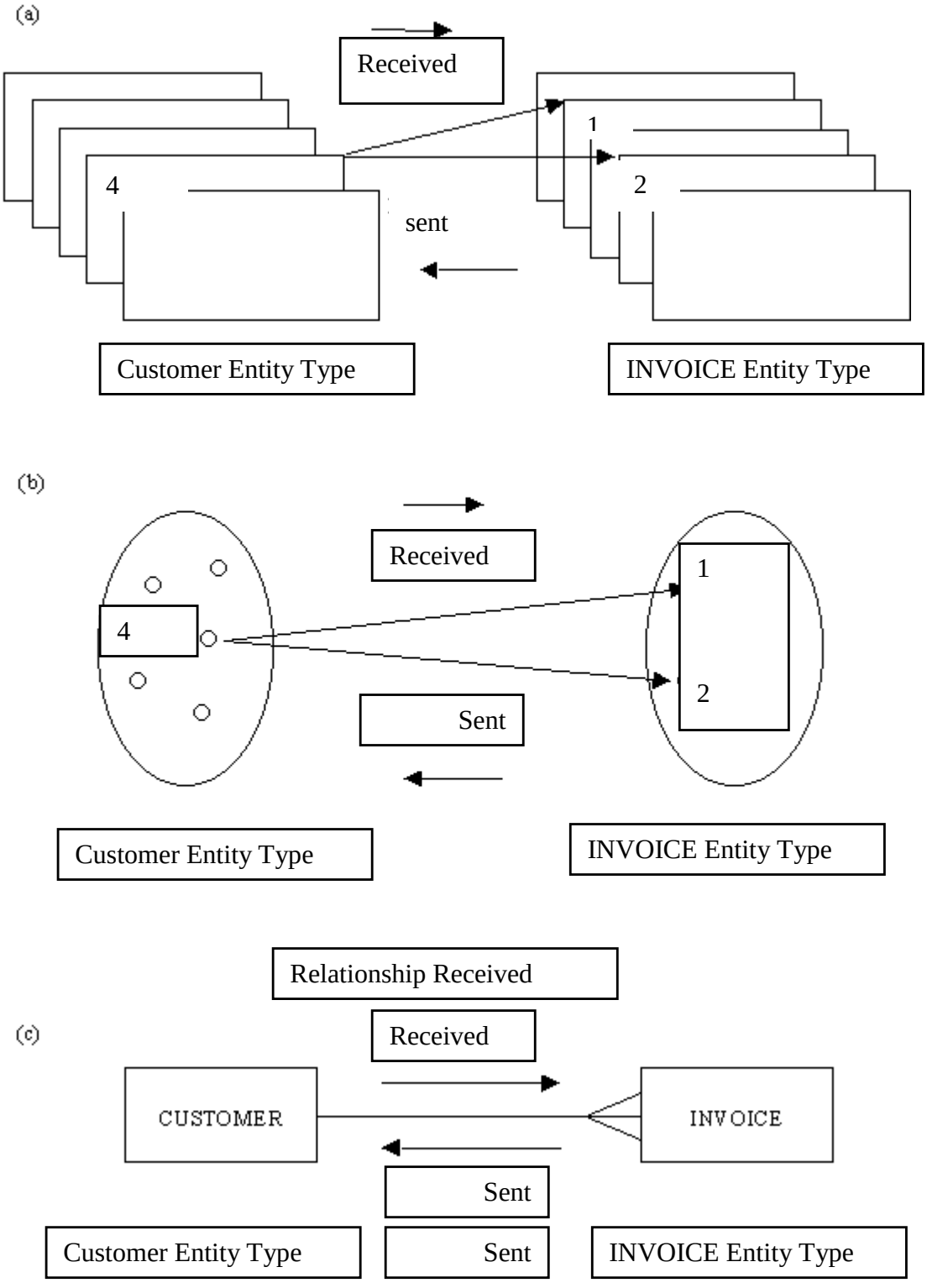


Fig. 2.5 A relationship, three ways of thinking

It was mentioned earlier that there is in fact a distinction between relationships and relationship types. In Fig. 2.5(a) and (b) there are in fact two relationships shown: one between customer 4 and invoice 1 and one between customer 4 and invoice 2, so strictly speaking `received' is a relationship type consisting of a number of relationships between entity types.

To conclude note that relationships between entity types are represented in a relational database using foreign keys. The value of the primary key of one entity is placed in every entity of the second type to which it is related.

Classification of Relationships Types

A relationship type can be classified by the number of entity types involved, and with the degree of the relationship type, as is shown in Fig. 2.6. These methods of classifying relationship types are complementary. To describe a relationship type effectively, you need to say what the name of the relationship type and its inverse are and their meaning, if not clear from their names and you also need to declare the entity type or types involved and the degree of the relationship type that links the entities.

The purpose of discussing the number of entity relationship types is to initiate the expressions such as unary, binary, and ternary, with examples. The number of entity types in the relationship type affects the ultimate form of the relational database.

The reason of discussing the degree of relationship types is to define the relevant terms, to give examples, and to show the impact that the degree of a relationship type has on the form of the ultimate implemented relational database.

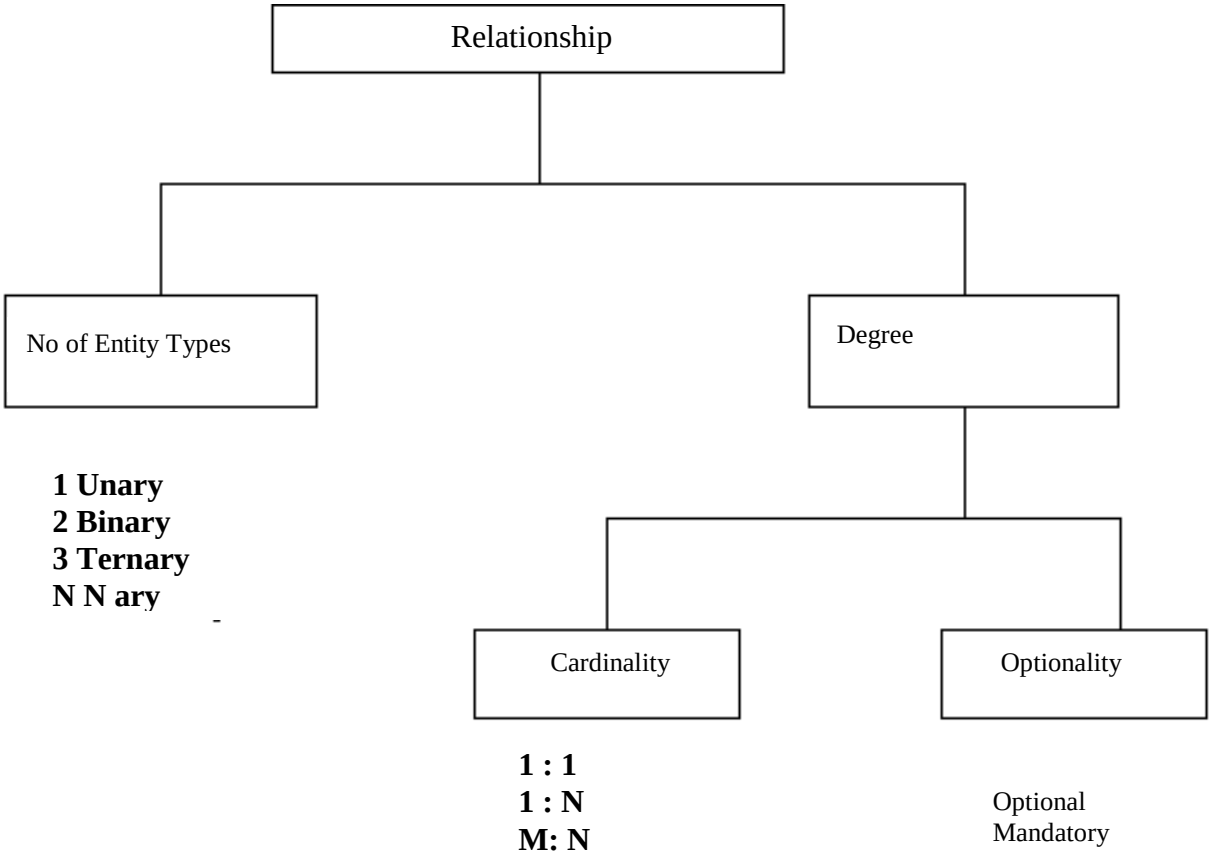
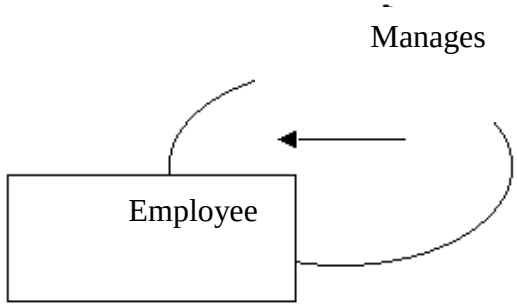


Fig. 2.6 The classification of relationships

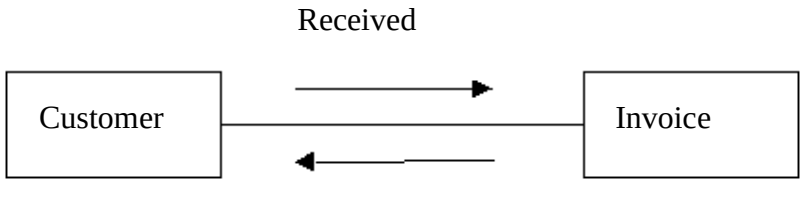
Entity Types

If a relationship type is between entities in a single entity type then it is called a unary relationship type. For example - the relationship `friendship' among entities within the entity type PERSON. If a relationship type is among entities in one entity type and entities in another entity type then it is called a binary relationship type because two entity types are involved in the relationship type. An example is the relationship 'Received' in Fig. 2.4 and Fig. 2.5 between customers and invoices. Another example of a binary relationship type is 'Purchased' among entity types CUSTOMER and PRODUCT. Two entity types are involved so the relationship is binary.

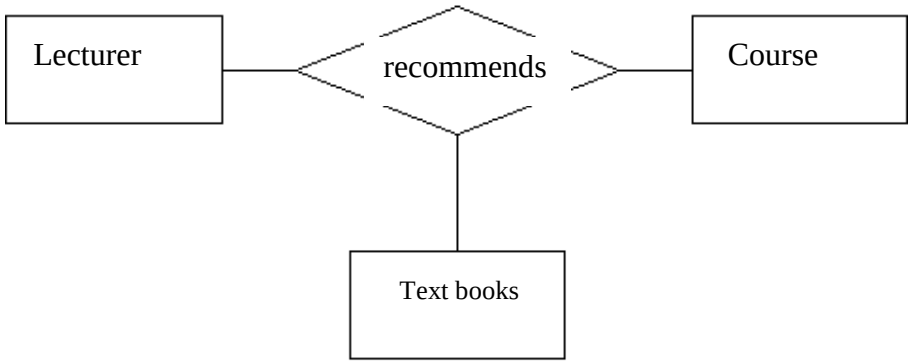
It is possible to model relationship types involving more than two entity types. For example a LECTURER 'recommends' a certain TEXT on a certain COURSE. Here the relationship type is 'recommends'. This relationship type is said to be a ternary relationship type as three entity types are involved. Examples of unary, binary and ternary relationship types are shown in Fig. 2.7.



(a) Unary



(b) Binary



(c) Ternary

Fig. 2.7 one, two, three entity types

Cardinality and Optionality

The maximum degree is called cardinality and the minimum degree is called optionality. The 'degree' is the term used to indicate the number of attributes in a relation while `cardinality' is the number of tuples in a relation. Here, we are not talking about database tables relations but relationship types, the associations between database tables and the real world entity types they form.

To show degree three symbols are used. A circle means zero, a line means *one* and a crows foot means many. The cardinality is shown next to the entity type and the optionality (if required) is shown behind it. Refer to Fig. 2.8(a). In Fig. 2.8(b) the relationship type R-relation has cardinality one-to-many because one A-alpha is related by R-relation to many B-betas and one B-beta is related (by R-relation's inverse) to one A-alpha. Generally, the degree of a relationship type is described by its cardinality. R-relation would be called a 'one-to-many' relationship type. To fully describe the degree of a relationship type however we should also specify its optionality.

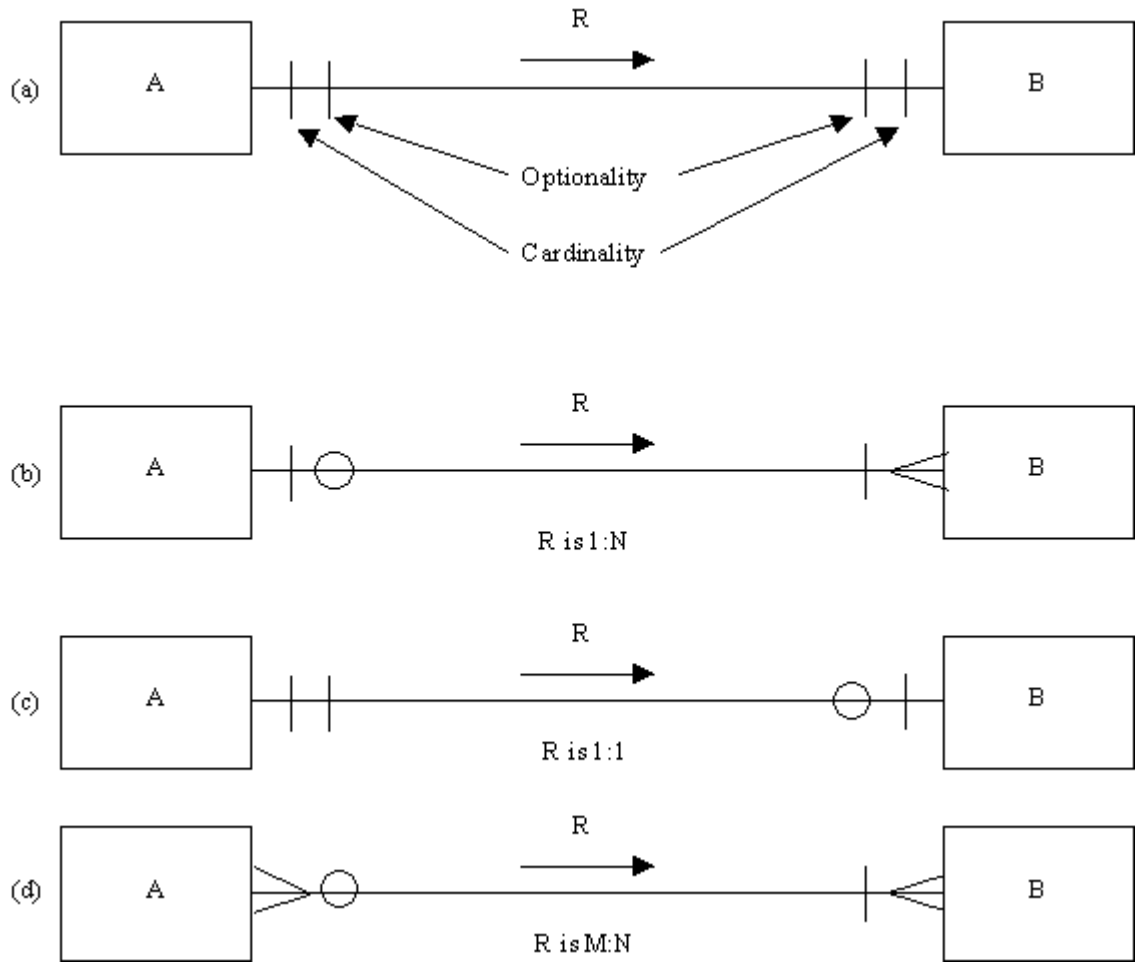


Fig. 2.8 Degree of Relationship

The optionality of relationship type R-relation in Fig. 2.8(b) is one as shown by the line. This means that the minimum number of B-beta that an A-alpha is related to is one. A-alpha must be related to at least one B-beta. Considering the optionality and cardinality of relationship type R-relation together, we can say that one A-alpha entity is related by R-relation to one or more B-beta

entities. Another way of describing the optionality of one, is to say that R-relation is a mandatory relationship type. An A-alpha must be related to a B-beta. R-relation's optionality is mandatory. With optionality, the opposite of 'mandatory' is optional..

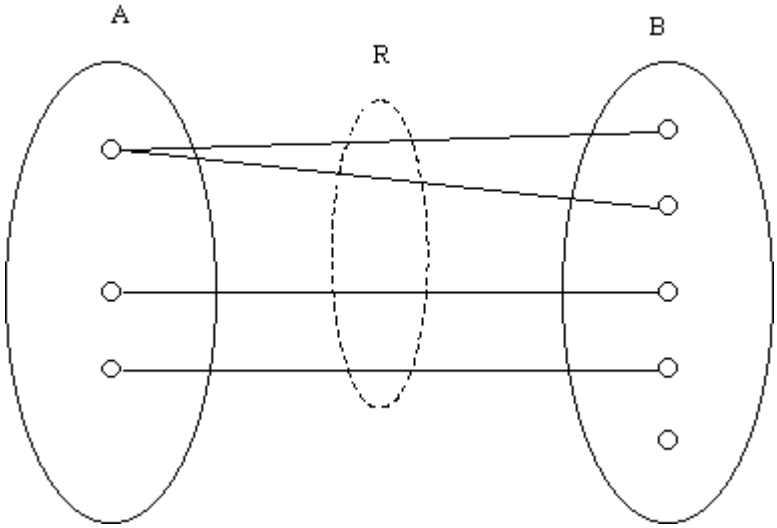


Fig. 2.9 A set diagram representation of Fig. 2.8(b).

The case of Fig. 2.8(b) is shown in the form of a set diagram in Fig 2.9. The two entity types A and B are shown as sets (the oval shapes). The entities are shown as small elements in the sets. The relationship type R-relation links A entities to B entities. It shows which A entities are related to which B entities. Notice that it is possible for an A entity to be related to one or more B entities. The maximum number of Bs for a given A is 'many' (for example the first A entity is related to two Bs) and the maximum number of As for a given B is one. This establishes the one-many cardinality of R-relation. The minimum number of Bs for a given A is 1. (There are no A entities without a B entity). This establishes mandatory optionality of R-relation. There can exist a B that is not related to any A; for example the last B entity.

A One-Many relationship type

In Fig. 2.10 the procedure for deriving the degree of a relationship type and putting it on the entity relationship diagram is shown. The example relates part of a sales accounting system. Customers may receive nil (zero) or more invoices from sales system. The relationship type is called as `received' and is from CUSTOMER to INVOICE. The the customer has received invoices zero and accordingly the `received' relationship type is optional. This is shown by the zero on the line. The maximum number of invoices the customer may have received is `many'. This is shown by the crowsfoot. This is briefed in Fig. 2.10(a). The definition of the relationship type the next step is to name the inverse relationship type. Clearly if a customer received an invoice, the invoice sent to the customer and this is an appropriate name for this inverse relationship type. Now consider the degree of the inverse relationship type. The minimum number of customers you would send an invoice to is one; you wouldn't send it to no-one. The optionality is thus one. The inverse relationship type is mandatory. The maximum number of customers you would send an invoice to is also one so the cardinality is also one. This is summarized in Fig. 2.10(b). Fig. 2.10(b) shows the completed relationship.

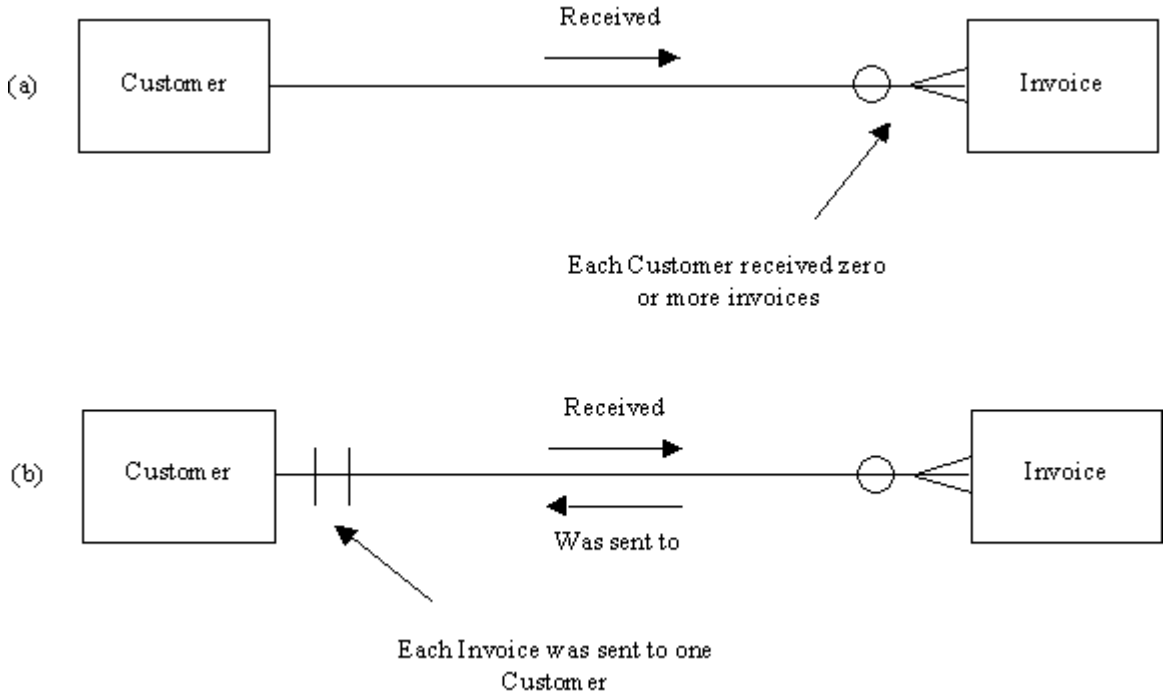


Fig. 2.10 A 1:N (one-:many) relationship.

A word of warning is useful here. In order to obtain the correct degree for a relationship type (one-one or one-many or many-many) you *must* ask *two* questions. Both questions *must* begin with the word `one'. In the present case the two questions you would ask when drawing in the relationship line and deciding on its degree would be:

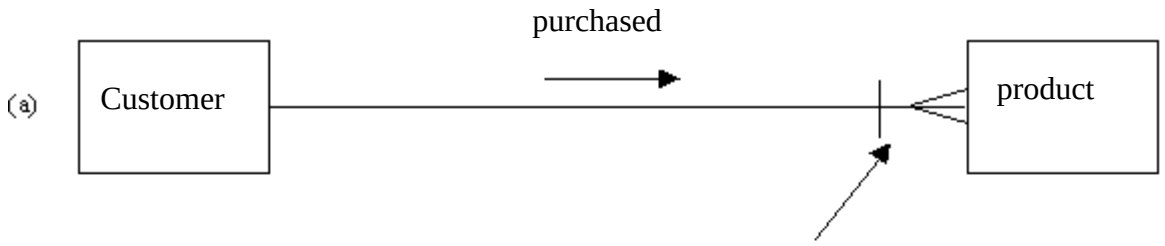
Question 1: One customer received how many invoices?
Answer: Zero or more.

Question 2: One invoice was sent to how many customers?
Answer: One.

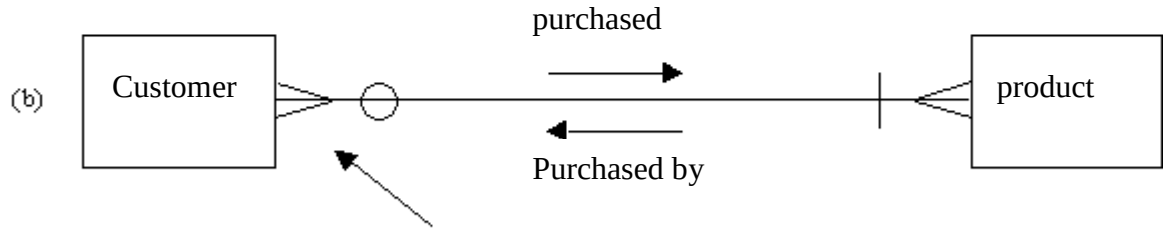
This warning is based on observations of many student database designers getting the degree of relationship types wrong. The usual cause of error is only asking one question and not starting with the word 'one'. For example a student might say (incorrectly): 'Many customers receive many invoices' (which is true) and wrongly conclude that the relationship type is many-many. The second most common source of error is either to fail to name the relationship type and say something like 'Customer to Invoice is one-to-many' (which is meaningless) or give the relationship type an inappropriate name.

A Many-Many relationship type

Fig. 2.11 an example of a many-many relationship type



Customer purchased one / more than one product



Product may be Purchased/may not be purchased by Customer

Fig. 2.11 A M:N (many-many) relationship.

To correctly derive the degree of this relationship (and the answers) there are two questions:

Question 1: One customer purchased how many product types?
Answer: One or more.

Question 2: One product type was purchased by how many customers?
Answer: Zero or more.

Note that the entity type has been called PRODUCT TYPE rather than PRODUCT which might mean an individual piece that the customer has purchased. In that case the cardinality of 'purchased_by' would be one and not many because an individual piece can of course only go to one customer.

We have assumption that every customer on the database has purchased at least one product; hence the mandatory optionality of `purchased'. If this were not true in the situation under study then a zero would appear instead. The zero optionality of purchased_by is due to our assumption that a product type might as yet have no purchases at all.

A One-One relationship type

Fig. 2.13 gives an example of a one-one relationship type being derived. It concerns a person and his or her birth certificate. We assume that everyone has one and that a certificate registers the birth of one person only.

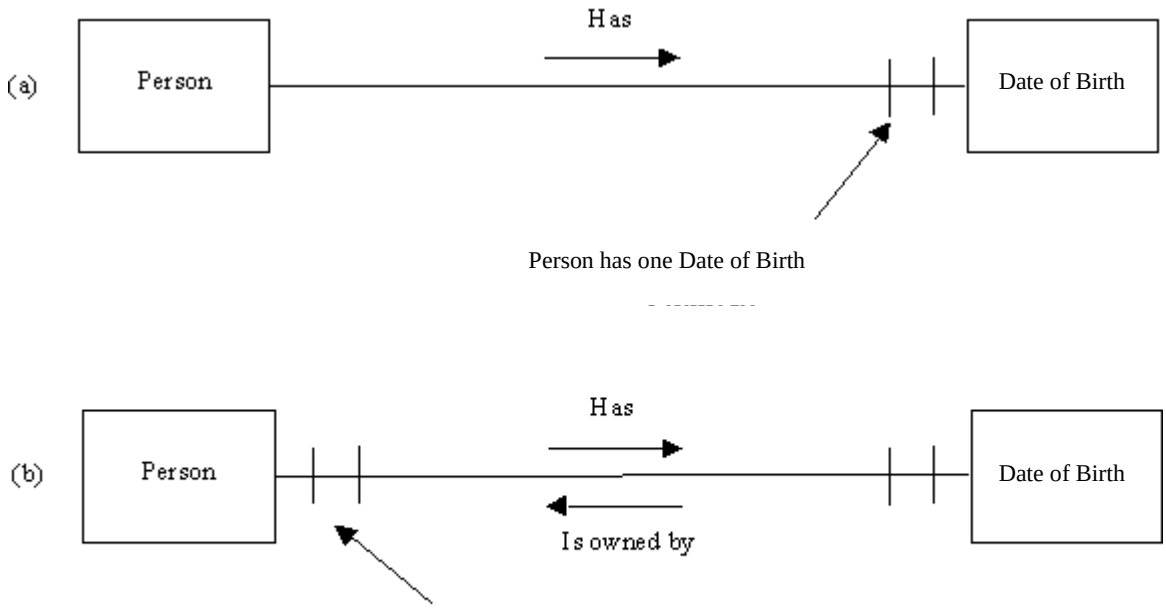


Fig. 2.13 Deriving a 1:1 (one to one) relationship.

Question 1: How many birth certificates has a person?
Answer: One.

Question 2: How many persons is a birth certificate owned by?
Answer: One.

.

Data Dictionary

In Database Management System, a file that defines the basic organization of a database. A data dictionary contains a list of all files in the database, the number of records in each file, and the names and each fields type. The majority database management systems keep the data dictionary hidden from users to prevent them from by mistake deleting its contents.

Data dictionaries do not hold any real data from the database, only accounting information for managing it. Without a data dictionary, however, a database management system cannot have right to use data from the database.

The analysis model encompasses representations of data objects, function, and control. in each representation data objects and control items play a role. Therefore it is necessary to provide an organized approach for representing the characteristics of each data object and control item. This is accomplished with the data dictionary.

The data dictionary has been proposed as a quasi formal grammar for describing the content of objects defined during structural analysis. This important modeling notation has been defined in the following manner:

The data dictionary is an organized listing of all data elements that are pertinent to the system, with precise, rigorous definitions so that both user and system analyst will have a common understanding of inputs, outputs, components of stores and intermediate calculations.
Today, the data dictionary is always implemented as part of a CASE 'structure analysis and design tool' although the format of dictionaries varies from tool to tool, most contain the following information:

- Name – the primary name of the data or control item, the data store or an external entity.
- Alias- other names used for the first entry.
- Where used/how used – a listing of the processes that use the data or control item and how it is used(e.g. input to the process, output from the process, as a store, as an external entity)
- Content description – a notation for representing content.
- Supplementary information – other information about data types, preset values (if known), restrictions or limitations, and so forth.

Once a data object or control item name and its aliases are entered into the data dictionary, consistency in naming can be enforced. That is, if an analysis team member decides to name a newly derived data item XYZ, but XYZ is already in the dictionary, the CASE tool supporting the dictionary reports a message about duplicate names. This improves the consistency of the analysis model and helps to reduce errors.

"where-used/how- used" information is recorded automatically from the flow models. When a dictionary entry is created, the CASE tool scans DFD's and CFD's to determine which processes use the data or control information and how it is utilized. This may appear trivial, it is actually one of the most important benefits of the dictionary. During analysis there is an almost continuous stream of changes. For large projects, it is often quite difficult to determine the impact of a change. A software engineer has asked where is this data object used? What else will have to change if we modify it? Because data dictionary can be treated as a database, the analyst can ask where used/how used? And get answers to these queries.
The notation used to develop a content description is noted below:

Data Construct	Notation	Meaning
	=	Is composed of
Sequence	+	And
Selection	[|]	Either or
Repetition	$()^n$	N repetitions of
	()	Optional data
	---	Delimits comments

The notation enables a software engineer to represent composite data in one of the three fundamental ways that it can be constructed.

1. As a sequence data items
2. As a selection from among a set of data items
3. As a repeated data items are grouped. Each data item entry that is represented as part of a sequence, selection or repetition may itself be another composite data item that needs further refinement within the dictionary.

For example – the data item telephone number. It could be 8 digit local number, a 5 digit + 8 digit (STD number). The data dictionary provides us with a precise definition of telephone number for the data flow diagram. In addition it indicates where and how this data item is used and any additional information that is relevant to it. A sample data dictionary entry is shown below:-

Name	Telephone number
Aliases	None
Where used/How used	Set –up generates list (Output) Dial number (input)
Description	Telephone number =[local number |STD] local number =8 digit STD=5 digit + 8 digit

The content description is expanded until all composite data items have been represented as elementary items or until all composite items are represented in terms that would be well known and unambiguous to all readers. It is also important to note that a specification of elementary data often restricts a system. For large systems, the data dictionary grows rapidly in size and complexity. It is extremely difficult to maintain a dictionary manually. For this purpose CASE tools should be used.

Summary

Requirements are an integral part of the development process connected through traceability with marketing specifications, analysis diagrams, design models, code files, test procedures and almost every other project deliverable. The structure of product requirements usually lends itself well to organizing the design, implementation and test activities that follow. The mapping between system requirements and product requirements and between product requirements and design, implementation and test procedures will help ensure complete coverage and eliminate overlap.
Each member of the development team uses a different subset of the requirement's information to do his or her job.
Given the leveraging aspect of the requirement's foundation, a good set of requirements is your best weapon in controlling project quality, cost and schedule.
Analysis must focus on the information, functional and behavioral domains of a problem. To better understand what is required models are created the problem is partitioned and representations that show the essence of requirements.
In Requirement phase the deliverables are as mention below:-

- Requirement Traceability Matrix
- Requirement Checklist

It is not possible to completely specify a problem at an early stage. Prototyping offers an alternative approach that results in an executable model of the software from which requirements can be refined. To properly conduct prototyping special tools and technique are required. The SRS is developed as a result of requirement analysis. An important requirement of requirements is that they are understood by the customer/client for concept exploration/feedback/validation. Requirement document should not have of too much technical jargon terms. An advantage of natural language is that the specification is very readable and understandable to the user and other non professional involved. Pictures may be put to advantage in bringing across the functional architecture of the system. A formal language allows us to use tools in analyzing the requirements. Review is necessary to ensure that the developer and customer/client have the same perception of the system.
Just like designing a house, designing software is an activity which demands creativity and a fair dose of craftsmanship. The quality of designer is of paramount importance in this process. The

essence of design process is that the system is decomposed into parts that each has less complexity than the whole. We discussed design method Data Flow Diagrams (DFDs) in this chapter.

A Data flow diagram helps in identifying the transaction data in the data model. Data flow diagrams were invented by Larry Constantine.

It is common practice to draw a context-level Data flow diagram. It shows the interaction between the system and outside entities. The DFD is designed to show how a system is divided into smaller portions and to highlight the flow of data between those parts. This context-level Data flow diagram is then "exploded" to show more system details.

Data flow diagrams (DFDs) are one of the three essential perspectives of Structured Systems Analysis and Design Method (SSADM). The sponsor of a project and the end users will need to be briefed and consulted throughout all stages of a system's evolution. With a dataflow diagram, users are able to visualize how the system will operate, what the system will accomplish, and how the system will be implemented. The previous system's dataflow diagrams can be drawn up and compared with the new system's dataflow diagrams to draw comparisons to implement a more professional system. Dataflow diagrams can be used to provide the end user with a physical idea of where the data they input ultimately has an effect upon the structure of the whole system from order to dispatch to report. How any system is developed can be determined through a dataflow diagram.

Self Test

21. Requirements are the agreed upon facts about what an application or system must accomplish for its users. State whether this statement is true or false.
    a) True. b) False
22. Software requirements analysis is divided into ------areas of efforts.
    a) Two. b) Five c) Six
23. State Software requirements analysis phases.
24. Requirement defects include----------, incomplete, non-essential, --------, overlapping or non-implemented requirements
    a) missing
    b) extra
    c) ambiguous
    d) incorrect
25. The defects occurring in the requirement identification phase are -----costly to correct.
    a) less
    b) more

26. Requirements analysis is a software engineering task that bridges the gap between system level requirements engineering and software design. State whether this statement is true or false.
    a) True. b) False
27. Requirement process starts at the ---------- of a project.
    a) middle
    b) end
    c) conception

28. Use cases are also known as----------.
    a) functional requirements. b) basic requirements
29. The SRS is based on approved Requirement Definition and Document (RDD). State whether this statement is true or false.
    a) True. b) False
30. What kind of specific language is required for SRS?
31. An advantage of natural language is that the specification is ---------- and ---------- to the user and other non professional involved.
    a) very readable b) understandable c) less readable d) minimum understandable
32. Explain requirement validation.
33. Requirements reviews are an activity started after SRS has been created. State whether this statement is true or false.
    a) True. b) False
34. The review team participants are ----------.
    a) system engineers
    b) customers,end users
    c) other stake holders
    d) All of the above
35. The bubble in between acts as the central transform. State whether this statement is true or false.
    a) True. b) False
36. The processes in the data flow diagram become the ----------.
    a) modules of the corresponding structure chart b) data module
37. One module produces a stream of data which is then consumed by another module. State whether this statement is true or false.
    a) True. b) False
38. The result of structured analysis is a ---------- of the system.
    a) logical model b) demo model c) prototype model
39. ------- between processes, external entities and data stores.
    a) data flows b) data changes c) data processed d) data exchanges
40. Data stores lie between ----------.
    a) One processes b) three processes c) two processes. d) four processes

Lesson 3

Software Design

Overview

In preliminary design we are going to discuss creation of algorithm, defining the modes and states of the system, creation of data flow and control flow at the middle level, data structures definition, sizing and allocation of memory, user interface designing, time line for the software project, review process for checking and approval of preliminary design.

In design principles we are going to discuss basic design principles, module level concepts.
In design notations we are going to discuss specification, notations and comparison of design notations.

Structured design being a strategy to map the information flow contained in data flow diagrams into a program structure is a genuine component of the design phase. Structured design methodology consists of transform flow and transaction flow.

Transform mapping is a set of steps that allows a DFD with transform flow characteristics to be mapped into specific architectural style. In many software applications, a sole data item triggers one or a number of information flows that affect a function implied by the triggering data item.

The blueprint for a house is not complete without a representation of doors, windows and utility connections for water, electricity and telephone. The doors, windows and utility connections for computer software make up the interface design of a system. Interface design focuses on three areas of concern: a) the design of interfaces between software components b) the design of interfaces between the software and other nonhuman products and consumers of information and c) the design of the interface between a human and the computer.

Learning Objectives

- To understand Preliminary Design, Design Principles Design Notations
- To understand Structured Design Methodology
- To understand Transform mappings and Transaction mapping
- To understand Verification

A **legacy system** is an old method, technology, computer system or application program that continues to be used, typically because it still functions for the users' wants, even though newer technology or more efficient methods of performing a task are now available. A legacy system may include actions or terminology which are no longer relevant in the current context, and may hinder or confuse understanding of the methods or technologies used.

The term "Legacy" may have little to do with the size or age of the system — mainframes run 64-bit Linux and Java alongside 1960s vintage code.

Although the term is most commonly used to describe computers and software, it may also be used to describe human behaviors, methods, and tools.

NASA example

NASA's Space Shuttle program still uses a large amount of 1970s-era technology. Replacement is cost-prohibitive because of the expensive requirement for flight certification; the inheritance hardware currently being used has completed the expensive integration and certified for flight, however any new equipment would have to go through that entire process – requiring extensive tests of the new components in their new configurations – before a single unit could be used in the Space Shuttle program. This would make any new system that started the certification process a de facto legacy system by the time of completion.

Additionally, because the entire Space Shuttle system, including ground and launch vehicle assets, was designed to work together as a closed system, and the specifications have not changed, all of the certified systems and components still serve well in the roles for which they were designed. It was advantageous for NASA – even before the Shuttle was scheduled to be retired in 2010 – to keep using many pieces of 1970s technology rather than to upgrade those systems.

Importance of Design

Design is a meaningful engineering representation of something that is to be built. It can be traced to a customer's requirements. Design focuses on four areas which are data, architecture, interfaces and components. As we know that without blueprint we cannot construct a house. The design in software engineering plays important role without that if we start developing product then it will be confusion making, error prone, components at wrong place. In nut shell computer software is complex in nature than a house, so we need a design as a blueprint.

Preliminary Design

In a broader manner following activities are carried out in this phase:-

- Creation of proper algorithm
- Defining the modes and states of the system
- Creation of data flow and control flow at the middle level
- Defining data structures and related things
- Creation of sizing and allocation of memory
- User interface designing
- Creation of time line for the software project
- A review process for checking and approval of preliminary design

In this phase first of all requirements are identified, decomposed from system specifications and transformed into functional specifications. The requirements are transformed into middle level design. The creation and managing documents at this phase is difficult because it is desirable to separate software and hardware functionality. The hardware requirement may be general or specific depending upon this hardware functional specifications are created. The creation of functional specifications for each middle level component is repetitive in nature. Using CASE tools this can be achieved. The preliminary design concern about mapping requirements to middle level components and how these components behave and interact is checked. At this middle level physical and logical interfaces are defined and data flow through the system is decided. The sizing, timeline, throughput (response time per unit time) estimations are documented for that part of the system and performance requirements. These requirements are derived from the system specification to

describe the software and hardware. On bigger system plans are created during this phase to help in the control and management of the products. The test plan and database definition documents are created. The test plan is a master document; it describes test environment, entry criteria, exit criteria, features to be tested, features not to be tested, roles and responsibilities. Test plans are always mapped with/ traced with the requirements for checking requirement coverage. The aim of preliminary design is to complete the logical design of all the components.

Activities of the Development Team

- **Prepare preliminary design diagrams.** Using functional decomposition or object-oriented techniques expand the preliminary software architecture that was proposed in earlier phases. Idealize the expanded architecture to minimize features with the aim of, it could make the software difficult to implement, test, maintain, or reuse. Evaluate design options. Weigh choices according to system priorities (e.g., optimized performance, ease of use, reuse considerations, reliability, or maintainability). By using prototyping and performance modeling to test alternatives, especially in risk areas. Study the software components that are candidates for reuse. If system response requirements are tough model the performance of the reusable components. Update the reuse plan to reflect the results of these reuse verification activities. Generate high-level diagrams of the selected system design and walk the analysts of the requirements definition team through them. Use the high-level design diagrams to clarify the system process flow from the analyst's perspective. Spotlight on system and subsystem interfaces. Explain refinements to the operations scenarios arising from analysis activities and include preliminary versions of user screen and report formats in the walkthrough materials.

- **Prepare prologs and PDL for the high-level functions/objects**. For FORTRAN systems, prepare prologs and PDL (program design language) to one level below the subsystem drivers. For Ada /C systems, prepare and compile specifications for the principal objects in the system and construct sufficient PDL to show the dependencies among packages and subprograms. Provide completed design figures, unit prologs/package specifications, and PDL to other members of the development team for independent inspection and certification.
- **Document the selected design in the preliminary design report.** Include the reuse plan, alternative design decisions, and external interfaces.
- **Conduct the PDR and resolve RIDs**. (review item disposition) Record design changes for use in the detailed design phase, but do not update the preliminary design report.

Activities of the Management Team

During preliminary design, the manager's focus changes from planning to control. The following are the major management activities of this phase:

- **Reassess schedules, staffing, training, and other resources.** Begin the software development history by recording lessons learned and project statistics from the requirements analysis phase. consist of percentages of project effort and schedule consumed growth of system size estimates, and team composition. Ensure that the development team contains a mix of software engineers and personnel experienced in the application area. If the project is large, partition the development team into groups, usually by subsystem, and hire group leaders. Adjust staffing levels to compensate for changes in requirements and staff attrition, and ensure the team obtains the training it needs to meet specific project demands.
- **Plan, coordinate, and control requirements changes**. Interface with analysts and the customer to facilitate resolution of requirements issues and TBDs. Monitor the number and severity of requirements questions submitted to analysts and the timeliness of responses. Ensure that analysts and the customer know the dates by which TBDs need to be resolved and understand the impact of changing or undetermined requirements items. Assess the technical

impact and cost of each specification modification. Constrain modifications that necessitate extensive rework and non-critical enhancements.

- **Control the quality of the preliminary design process and its products** During, day-to-day management activities. Ensure that design walkthroughs, inspections, and reviews are scheduled and conducted. Attend walkthroughs and, optionally, inspections and oversee the reporting, tracking, and resolution of the design issues that arise. Make certain that all requisite software documentation is generated and review the preliminary design document. Ensure that the team adheres to project standards and configuration management procedures.
- **Plan an orderly transition to the detailed design phase**. Consider the impacts of TBDs,(to be determined) specification modifications, and schedule or team adjustments. Revise project estimates of effort, duration, and size and update corresponding sections of the SDMP (software development/management plan). Develop the project build strategy and prepare a initial/ preliminary build plan reflecting prototyping results, project risks, and residual TBDs.

Increase team size if necessary to begin detailed design and address the training needs of additional personnel. Oversee the establishment of online libraries to store unit prologs, PDL, and reused code. While project and group leaders prepare for PDR, start the rest of the team on detailed design activities. Control the PDR and confirm that they have met all exit criteria before saying the phase is complete.

Activities of the Requirements Definition Team

During the preliminary design phase, the requirements definition team provides support to software developers through the following activities:

- **Continue to resolve requirements issues and TBDs.** Clarify ambiguous, conflicting, or incomplete requirements. Provide prompt, written replies to developers' requirements questions and talk about these responses with developers. Respond to changes in high-level system requirements, evaluate the impact of each change, and prepare specification modifications accordingly.
- **Participate in design walkthroughs and the PDR.** Thoroughly analyze the proposed design. Work with developers to refine the operational scenarios and preliminary user interface. Follow up with developers to address issues raised during the walkthroughs. Review the preliminary design report and all supporting hardcopy resources before PDR. Pose questions and provide critiques of the initial, high-level system design during the review meeting, and use RIDs to document serious discrepancies

Design Principles

Software design is both a process and model. The design process is a sequence of steps that enable the designer to describe all aspects of the software to be built. It is important to note however, that the design process is not simply a cookbook. Creative skill, past experience, a sense of what makes good software and an overall commitment to quality are critical success factors for a competent design.

The design model is equivalent of an architect’s plans for a house. It begins by representing the totally one thing to be built and slowly refines the thing to provide guidance for constructing each detail. Similarly, the design model that is created for software provides a variety of different views of the computer software.

Basic design principles enable the software engineer to navigate the design process, a set of principles of software design , which have been adapted and extended in following list:-

- The design process should not suffer from tunnel vision. A good designer should consider alternative approaches, judging each based on requirements of the problem, the resources available to do the job, and the design concepts.

- The design should be traceable to the analysis model. Because a single element of design model often traces to multiple requirements of the problem, it is necessary to have a means for tracking how requirements have been satisfied by the design model.
- The design should not reinvent the wheel. Systems are constructed using a set of design patterns, many of which have likely been encountered before. These patterns should always be chosen as an alternative to reinvention. Time is short and resources are limited. Design time should be in representing truly new ideas and integrating those patterns that already exist.
- The design should exhibit uniformity and integration. A design is uniform if it appears that single person developed the entire thing. Rules of style and format should be defined for a design team previous to design work begins. A design is integrated if care is taken in defining interfaces among design components.
- The design should be structured to accommodate change. The design concepts discussed in the next section enable a design to achieve its principle.
- The design should be structured to degrade tenderly, even when aberrant data , events or operating conditions are arises. Properly designed software should never explode. It should be designed to house unusual conditions, and if it must terminate processing, do so in a polished manner.
- Coding means not design. Design means not coding. When in depth procedural designs are created for program components, the level of abstraction of the design model is higher as compare to source code. The only design decisions made at the coding level address the small implementations details that enable the procedural design to be coded.
- The design should be assessed for quality as it is being created not after fact. A variety of design concepts and design measures are available to assist the designer in assessing quality.
- The design should be reviewed to minimize conceptual (semantic) errors. There is sometimes a tendency to focus on finer points when the design is reviewed, missing the forest for trees. A design team has to confirm that main conceptual fundamentals of the design have been addressed. Afterwards thought about of the design model syntax.

These design principles are correctly applied, then the software engineer creates a design that depicts both internal and external quality factors. External quality factors are speed, reliability, correctness usability. Internal quality factors are of importance to software engineers.

Module Level Concepts

Modular design has become accepted approach in all engineering disciplines. A modular design reduces difficulty, facilitates change and results in easier implementation. It also helps in parallel development of different parts of system. The concept of modularity has been espoused (adopted) for almost four decades. Software is divided into separately named and addressable components, called modules. Meyer defines five criteria that enable us to evaluate a design method with respect to its ability to define an effective modular system:

- Modular decomposability:

A design method provides a systematic mechanism for decomposing the problem into sub-problems --> reduce the complexity and achieve the modularity

- Modular compos ability:

A design method allows existing (reusable) design components to be assembled into a new system.

- Modular understandability:

A module can be understood as a standalone unit it will be easier to build and easier to change.

- Modular continuity:

Small changes to the system requirements result in changes to individual modules, rather than system-wide changes.

- Modular protection:

An abnormal condition occurs within a module and its effects are constrained within the module.

Effective Modular Design

Functional independence:

The concept of functional independence is a direct outcome of modularity and the concepts of abstraction and information hiding. Functional independence is achieved by developing modules with sole minded function and an aversion (dislike) to excessive interaction with other modules.
That means we want to design software so that each module addresses a specific sub function of requirement and has a simple interface when viewed from other parts of the program structure.

Why Independence is important?
Software with effective modularity that is independent module is easier to develop because function may be compartmentalized and interfaces are simplified. Independent modules are easier to maintain as secondary effect caused by design or code modification are limited. That means error propagation is reduced and reusable modules are possible. In nutshell functional independence is a key to good design and design is the key to software quality.
Independence is measured using two qualitative criteria a) Cohesion and b) Coupling.

Cohesion: Cohesion is a measure of the relative functional strength of a module. Cohesion is a natural extension of the information hiding concept, means module should be designed so that information contained within a module is inaccessible to other module which have no need for such information. High cohesion is good and low cohesion is worse.

A cohesive module performs a single task in a procedure with fewer communications with others.

At the low end of the cohesion we come across a module which performs a certain set of tasks that relates to each other loosely. Such modules are termed **coincidently cohesive.** A performs tasks so as to are related logically is **logically cohesive.** When a module contains tasks so as to are related by the fact that all must be executed with the same span of time, the module exhibits **temporal cohesion**.
Moderate levels of cohesion are relatively close to one another in the degree of module independence. When processing elements of a module are related and must be executed in a specific order, **procedural cohesion** exists. When all processing elements concentrate on one area of data structure, **communicational cohesion** is present.

Coupling: Coupling is a measure of the relative interdependence 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. In software design we strive for lowest achievable coupling. Simple connectivity among modules results in software that is easier to understand and less prone to a ripple effect caused when errors occur at one location and propagate through a system. Different types of module coupling are shown in Figure 10.1. below:-

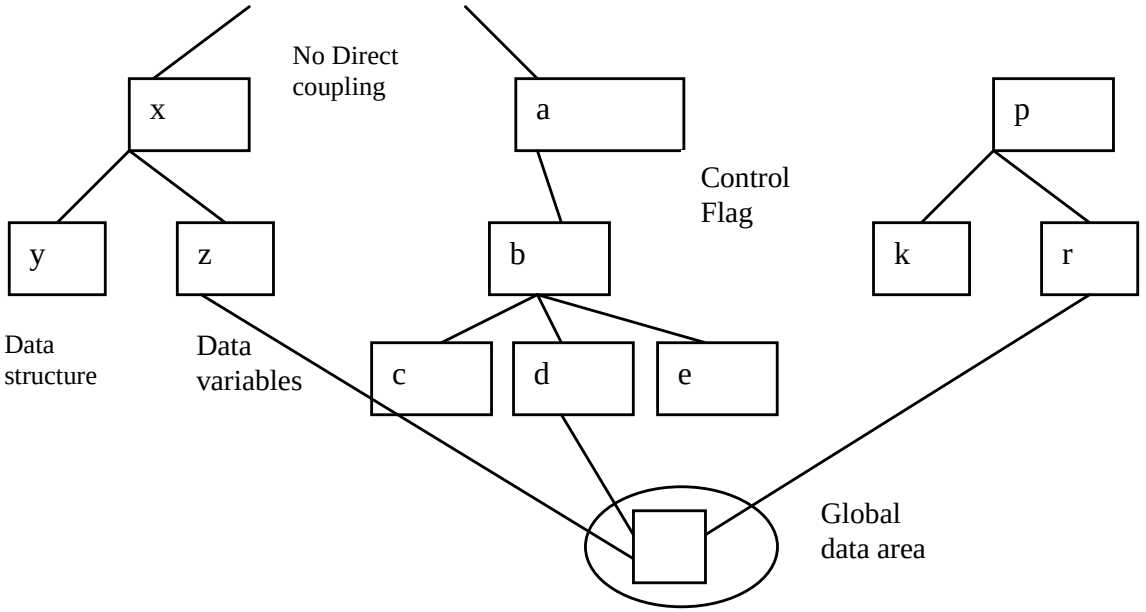


Figure 3.1

Module x and a are subordinate to different modules. Each is unrelated and therefore no direct coupling occurs. Module z is subordinate to module x and is accessed via a conventional argument list through which data are passed. Simple data are passed; one to one correspondence of items exists. This type of coupling is called as low coupling. When a portion of data structure is passed via a module interface, this type of coupling is called as stamp coupling. Module y and x shows stamp coupling. At moderate levels coupling is characterized by passage of control among modules. Control coupling is very common in software design; it is shown in Figure 3.1 where control flag is passed between module a and b. High levels of coupling occur when modules are tied to an environment external to the software. For example input/output couples a module to specific devices and communication protocols. External coupling is essential but should be limited to a small number of modules with structure. High coupling also occur when a number of modules reference a global data area. Module z and d are showing common coupling, each module is accessing data item in a global data area. (For example a disk file or a globally accessible memory area) Module z initializes the item and module d recomputed and updates the item. Let's assume that an error occurs and module d updates the item incorrectly. Later on in processing module r reads the item, attempts to process it and fails causing software to abort. The apparent cause of abort is module r, but the actual cause is module d. In nutshell to overcome common coupling problem software designer must be aware of potential consequences of common coupling and take special care to guard against them.

Content coupling, which is highest degree of coupling, occurs when one module makes use of data or control information maintained within the boundary of another module.

Content coupling also occurs when branches are made in to the middle of module. This type of Content coupling should be avoided.

Design Notations and Specifications

It is often said that a picture is worth a thousand words. In all sorts of designs, be they for houses, radios or other electronic appliances, extensive use is made of various drawing techniques. In many cases, the resulting drawings are major outcome of design process. It seems obvious to do the same when designing software.

There are number of reasons why this is not as simple as it may first appear:

- Software is far more subject to change. Incorporating those changes in the design takes a lot of effort and, if things have not been automated, reality and its pictorial counterpart will soon diverge.
- One of the major issues in software design is the hierarchical decomposition of the system. This seems to play a lesser role in other types of design, possibly because most radios and houses have less complex structure than the average 5000 line program. Many drawing techniques do not require very strict rules for this hierarchical decomposition.
- The outcome of the design is the major source of information for the programmer. The programmer is best served by a very precise description. Existing drawing techniques tend to be fairly simple and pictures thus obtained have to be extensively decorated through natural language annotations. The text added often allows multiple interpretations.

However, pictures may be a powerful tool in design of software. This holds in particular for global stages of design. In those stages there tends to be frequent interaction with the user. The user is often more at ease with pictures than with some formal description. Example of this type technique is data flow diagrams.

We may predict pictures to be generated automatically from a formal description. If it is done, these pictures at least have sound semantics (study of meaning in language). As of now, quite number tools are available for applying this kind of drawing technique. Many of the shortcomings of these techniques can be overcome by such tools.

Various types of symbology can be used to describe a design. Narrative text and formal languages represent extreme orientations along this dimension. A major advantage of formal languages is that they allow formal checks. Narrative text tends to be unwieldy (bulky) and ambiguous. The low level design is often documented in a constrained language, of pseudo code. A pseudo code generally supports the control structures offered by nearly all high level languages, such as if-then-else constructs, while or repeat constructs and block structures like begin end. A pseudo code often provide help for a number of useful data structures such as stacks, sequences, lists and sets along with their usual operations. Apart from that they impose hardly any strict rules as regards syntax or semantics and you may freely add new names and notations. Software design languages based on an existing high level programming language allow for a smooth transition from design to code. A pseudo code can not be compiled. It should have following characteristics:-

- A fixed syntax of keywords that provide for all structured constructs, data declaration and modularity characteristics.
- A free syntax of natural language that describes processing features.
- Data declaration facilities that should include both simple and complex data structures.
- Subprogram definition and calling techniques that support various modes of interface description.

Graphical design notation comprises of a flow chart and the box diagram. A flow chart is simple pictorially. A box is used to indicate a processing step. A diamond represents a logical condition and arrows show the flow of control.

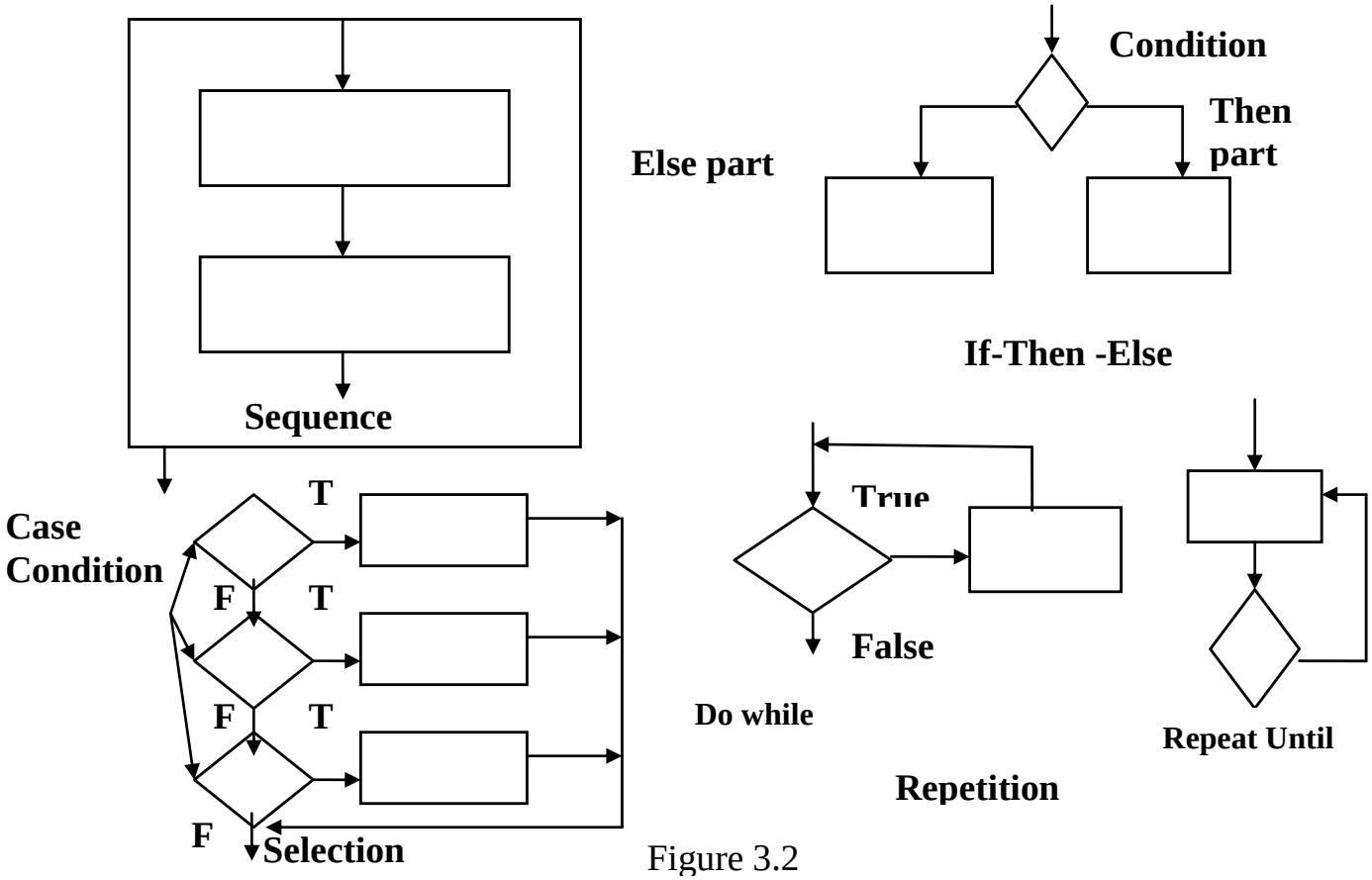


Figure 3.2

Figure 3.2 above shows three constructs. The sequence is represented as two processing boxes connected by line of control. Condition also called if-then-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. A 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 the condition and repeats the task until the condition fails. The selection (or select case) construct shown in the Figure 3.1 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 box diagram evolved from a desire to develop a procedural design represented that would not allow violation of the structured constructs. It has following characteristics a) functional domain (the scope of repetition or if-then-else) is well defined and clearly visible as a pictorial representation b) arbitrary transfer of control is impossible c) the scope of local and global data can be easily determined d) recursion is easy to represent. The graphical representation of structured constructs using the box diagram is shown in Figure 3.3. The sequence is represented using two boxes connected bottom to top. To represent if-then-else a condition box is followed by a then part and else part box. Repetition is shown with bounding pattern that encloses the process to be repeated. Selection is represented using the graphical form shown in Figure 3.3.

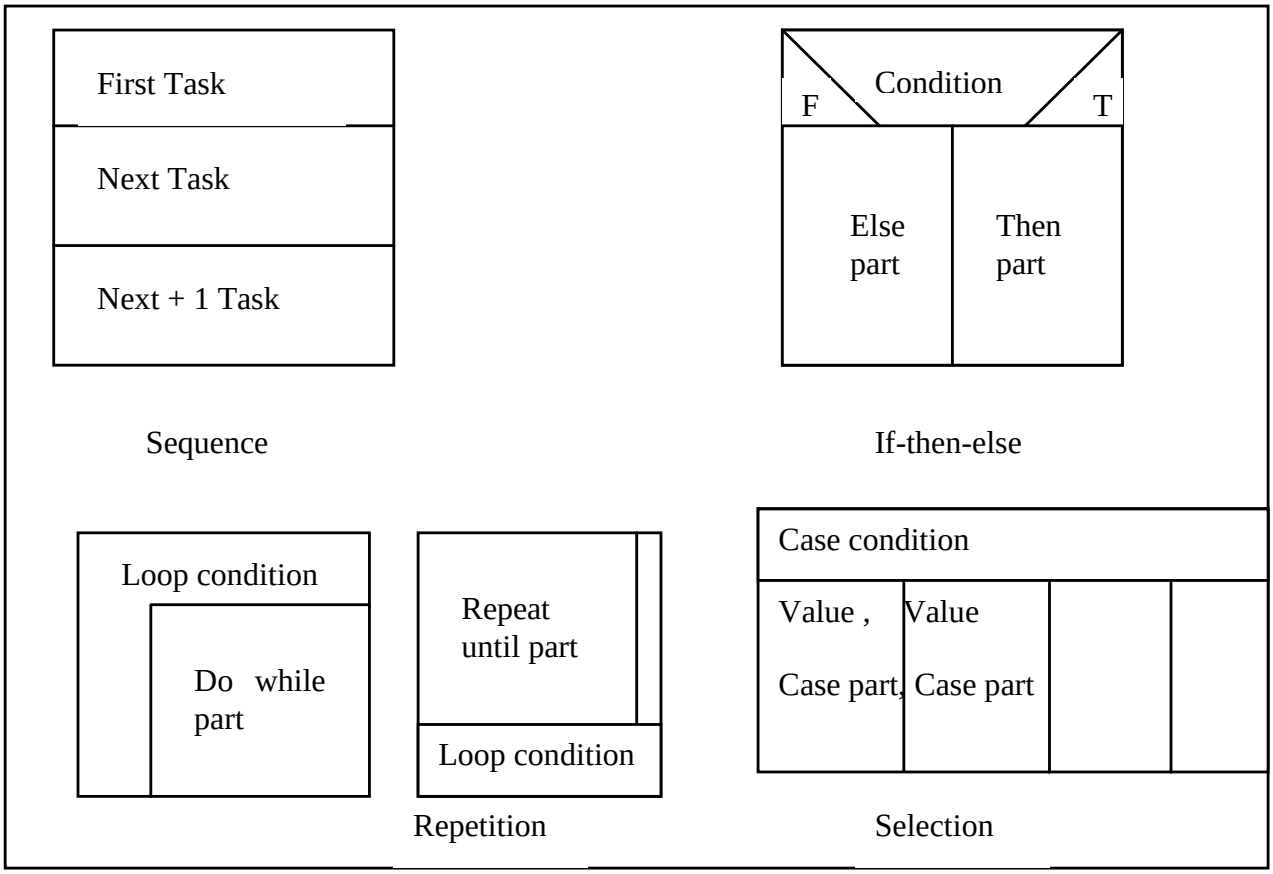


Figure 3.3

Comparison of Design Notation

In preceding section, we presented a number of different techniques for representing a procedural design. A comparison must be predicated on the premise that any notation for component level design, if used correctly, can be invaluable aid in the design process; conversely, even the best notation, if poorly applied, adds little to understanding. With this thought in the mind, we examine criteria that may be applied to compare design notation.

Design notation should lead to procedural representation that is very easy to understand and review. In addition, the notation should enhance "code to" ability so that code does, in fact, become a natural by product of design. Finally the design representation must be easily maintainable so that design always represents the program correctly.

The following attributes of design notation have been established in context of the general characteristics described previously:

- Modularity: design notation should support the development of modular software and provide a means of interface specifications.
- Overall simplicity: design notation should be relatively simple to learn, relatively easy to use and generally easy to read.
- Ease of editing: the procedural design may need modification as the software process proceeds. The ease with which a design represented can be edited can help facilitate each software engineering task.
- Machine readability: notation that can be input directly into a computer based development method offers significant benefits.

- Maintainability: software maintenance is most costly phase of software lifecycle. Maintenance of software configuration nearly always means maintenance of the procedural design representation can be edited can help facilitate each software engineering task.
- Structural enforcement: the benefits of a design approach that apply structured programming concepts have already have been discussed. Design notation that enforces the use of only the structured constructs promotes good design practice.
- Automatic processing: a procedural design has information that can be processed to give the designer new or better insights into the correctness and quality of design. Such insight can be enhanced with reports provided via software design tools.
- Data representation: the ability to represent local and global data is an essential element of component level design. Ideally, design notation should represent such data directly.
- Logic verification: automatic verification of design logic is a goal that is paramount during software testing. Notation that enhances the ability to verify logic greatly improves testing adequacy.
- "Code to" ability: the software engineering task that follows component level design is code generation. Notation that may be converted easily to source code reduces effort and error.

A natural question that arises in any discussion of design notation is "what notation is really the best, given the attributes noted above?" any answer to this question is admittedly subjective and open to debate. However, it appears that program design language (pseudo code) offers best combination of characteristics. A pseudo code may be embedded directly into source listing, improving documentation and making maintenance less difficult. Editing can be accomplished in the company of any text editor or word processing system. The pictorial nature of flowcharts and box diagrams provide a perspective on control flow that most designers prefer. In the final analysis the choice of a design tool may be more closely related to human factors than to technical attributes.

Structured Design Methodology

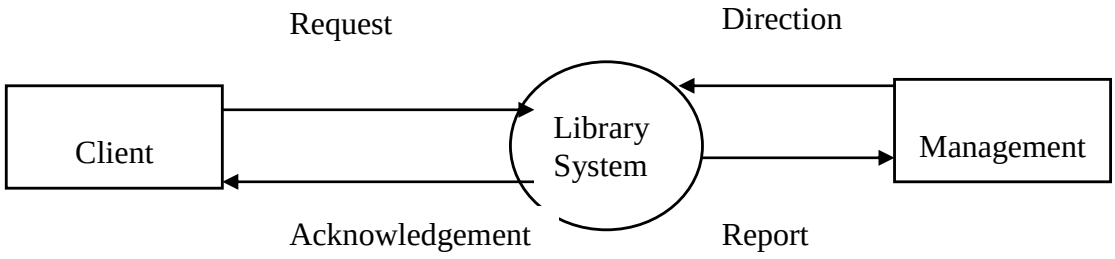


Figure A Context level DFD for Library automation (Level 0 DFD)

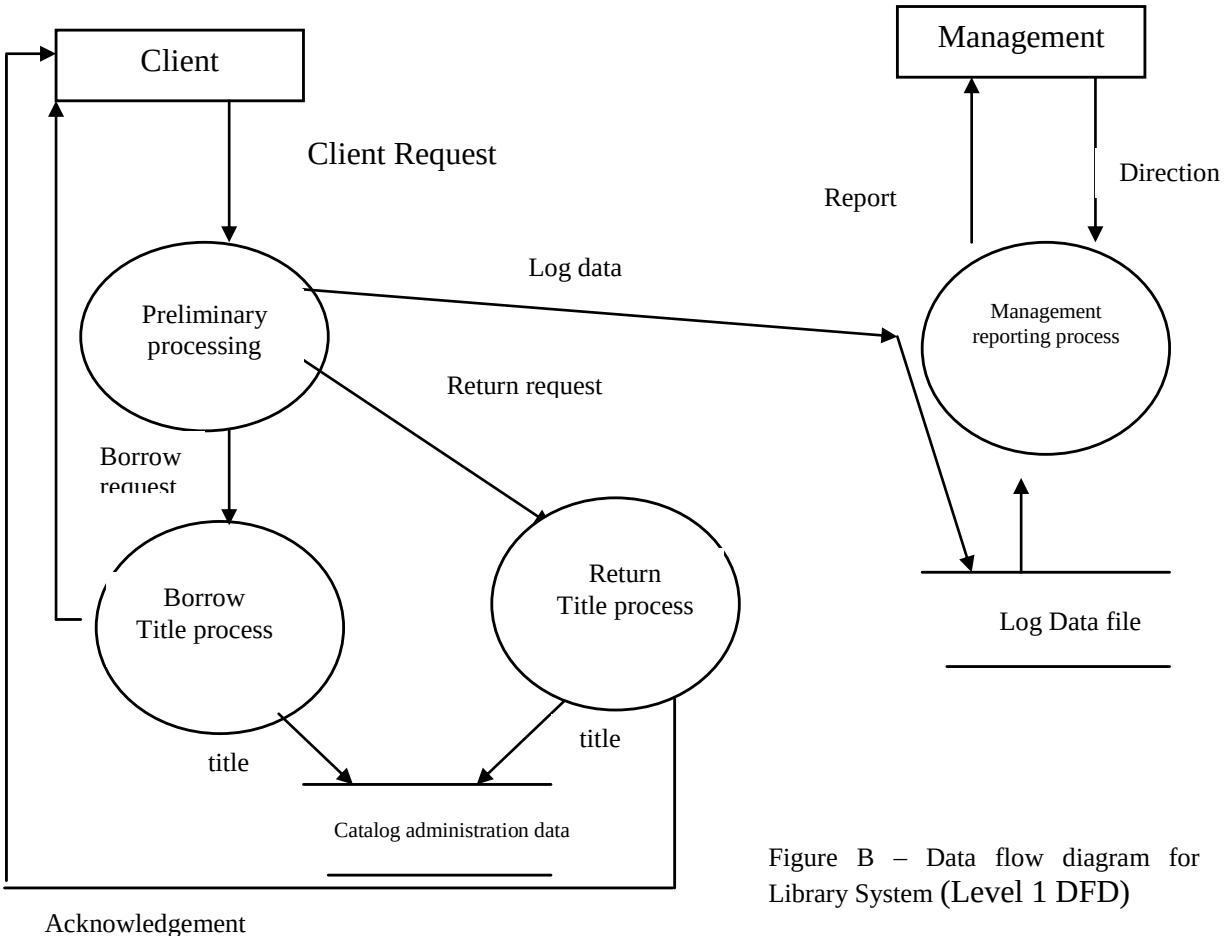


Figure B – Data flow diagram for Library System (Level 1 DFD)

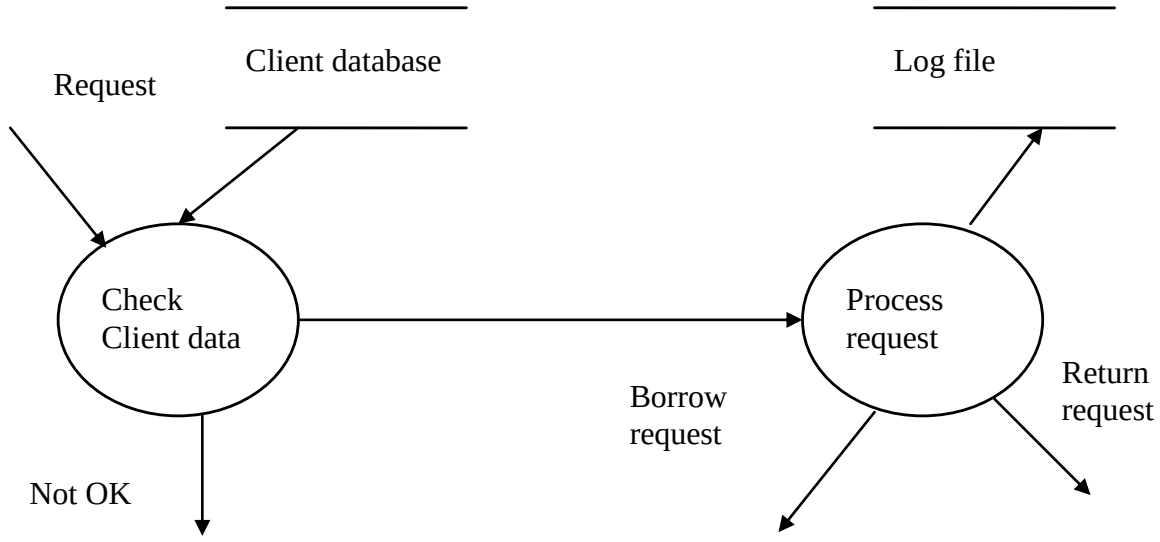


Figure C DFD for 'preliminary processing' (Level 2 DFD)

Software requirements can be mapped into various representations of the design model. To illustrate one approach to architectural mapping, we consider **the call and return** architecture. This is an extremely common structure for many types of systems. The mapping technique to be presented enables a designer to derive reasonably complex call and return architecture from data flow diagrams within the requirements model. The technique sometimes called as structured design. Structured design is often characterized as a data flow design method because it provides a convenient transition from a data flow diagram to software architecture. The transition from information flow (DFDs) to program structure is accomplished as part of a six step process:-

1) The type of information flow established 2) flow boundaries are indicated 3) the DFD is mapped into program structure 4) organize hierarchy is defined 5) resultant structure is refined using design measures and heuristics 6) the architectural description is refined and elaborated.

The type of information flow is the driver for the mapping in step 3. There are two flow types A) Transform Flow B) Transaction Flow.

Transform flow:- information must enter and exit software in an external world form. For example data typed on a key board, tones on a telephone line, and video images in a multimedia application are all forms of external world information. Such external data must be converted into an internal form for processing. Information enters the system along paths that convert external data into an internal form. These paths are identified as **incoming flow**. At the kernel (core) of the software a transition occurs. Incoming data are passed through a transform center and begin to move along routes that now lead out of the software. Data moving alongside these paths are called **outgoing flow**. The overall flow of data occurs in a sequential manner and follows one or only a few, straight line paths. When a segment of a data flow diagram exhibits these characteristics **transform flow** is present.

Transaction flow:- the fundamental system model implies transform flow, therefore it is possible to characterize all data flow in this category. However information flow is often characterized by a single data item called **transaction**, which triggers other data flow along one of many paths. When a DFD takes the form shown in Figure D, **transaction** flow is present.

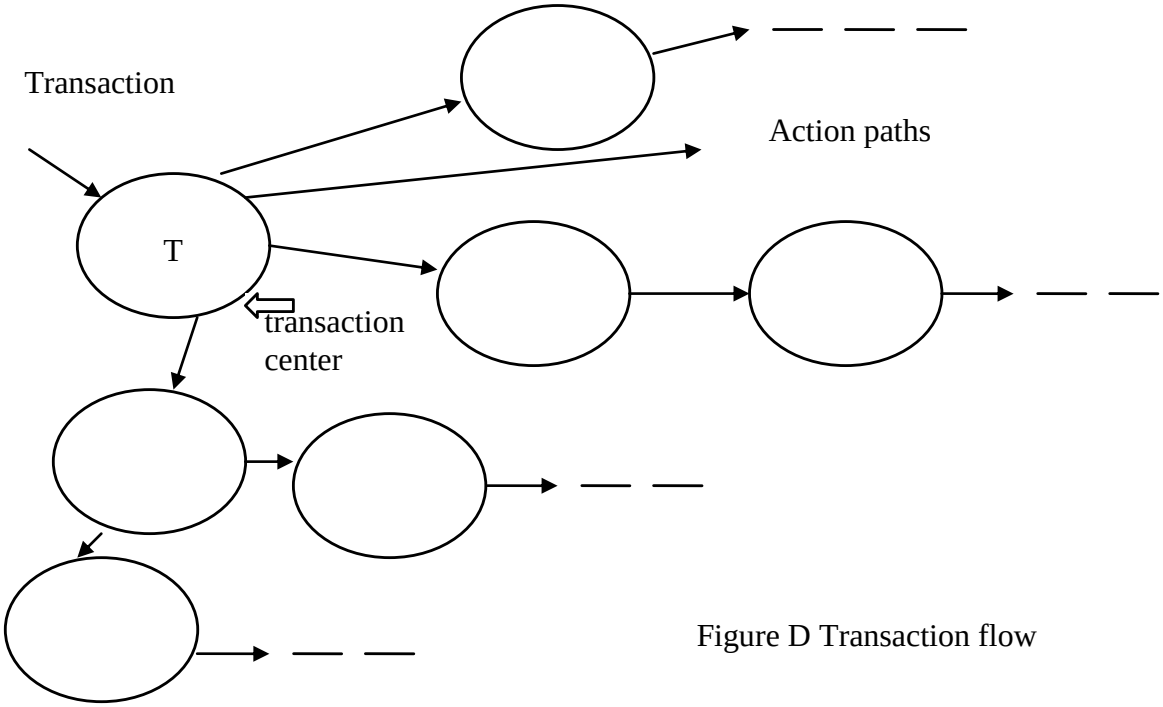


Figure D Transaction flow

Transaction flow is characterized by data moving along an incoming path that converts external world information into a transaction. The transaction is evaluated and based on its value flow alongside one of many action paths is initiated. The hub of information flow from which many action paths originate is called a transaction center. It should be noted that within a DFD for a large system both transform and transaction flow may be present. For example in a transaction oriented flow, information flow along an action path may have transform flow characteristics.

Transform mapping

Transform mapping is a set of steps that allows a DFD with transform flow characteristics to be mapped into specific architectural style.

Design Steps:-

An example discussed in Figure A, B, C will be used to illustrate each step in transform mapping. The step begins with an evaluation of work done during necessities analysis and then move to the design of the software architecture.

Step 1 Review the fundamental system model.

The fundamental system model encompasses the level 0 DFD and supporting information. In reality, the design step begins with an evaluation of both system specification and the software requirements specification. Both documents describe information flow and structure at the software interface. Figure A (Level 0 DFD), B (Level 1 DFD) shows data flow for library automation.

Step 2 Review and refine data flow diagrams for the software.

Information obtained from analysis models contained in the software requirements specification is refined to produce greater detail. For example, Figure C, the level 2 DFD for library automation is shown.

Step 3 Determine whether the DFD has transform or transaction flow characteristics.

In general information flow within a system can be always be represented as transform. However when an obvious transaction characteristic is encountered, a different design mapping is recommended. (Figure D) In this step the designer selects global flow characteristic based on the prevailing nature of the DFD. In addition, local regions of transform or transaction flow are

isolated. These sub flows can be used to refine program architecture derived from a global characteristic described previously.

Step 4 Isolate the transform center by specifying incoming and outgoing flow boundaries.

In the preceding section incoming flow was described as a path in which information is converted from external to internal form. Incoming and outgoing flow boundaries are open to interpretation. Care should be taken when boundaries are selected, a variance of one bubble along a flow path will generally have little impact on the final program structure.

Step 5 Perform first level factoring.

Program structure represents a top down distribution of control. Factoring results in a program structure in which top level modules perform decision making and low level modules carry out most input, computation and output work. Middle level modules perform some control and do moderate amounts of work. When transform flow is encountered, a DFD is mapped to a specific structure (a call and return architecture) that provides control for incoming, transform and outgoing information process.

Step 6 Perform second level factoring.

Second level factoring is accomplished by mapping individuals transforms (bubbles) of a DFD into appropriate modules within the architecture. Beginning at the transform center boundary and moving outward along incoming, and outgoing paths, transforms are mapped into subordinate levels of the software structure. One-to-one mapping between DFD transforms and software modules is required. Two or even three bubbles can be combined and represented as one module or a single bubble may be expanded to two or more modules. Practical considerations and measures of design quality dictate the outcome of second level factoring. Review and refinement may lead to changes in this structure, but it can serve as a 1st iteration design.

Second level factoring for incoming flow follows in the same manner. Factoring is accomplished by moving outward from the transform center boundary on the incoming flow side. Each of the data conversion or calculation transforms of the transform portion of the DFD is mapped into module subordinate to the transform controller.

Step 7 Improve the 1st iteration architecture using design heuristics for improved software quality.

1st iteration architecture can always be refined by applying concepts of independence. Module are exploded or imploded to produce prudent factoring, good cohesion, minimal coupling and most important a structure that can be implemented with no difficulty, tested without confusion and maintained without grief. Refinements are dictated by the analysis and assessments methods like collect scenarios, elicit requirements, constraints and environment description.

The objective of above discussed seven steps is to develop an architectural representation of software. That is once structure is defined; we can evaluate and refine software architecture by viewing it as a whole. Modifications made at this time require little additional work yet can have a profound impact on software quality.

Transaction mapping

In many software applications, a single data item triggers one or a number of information flows that affect a function implied by the triggering data item. The data item called transaction and its corresponding flow characteristics are discussed along with figure D.

Design steps:-

The design steps for transaction mapping are similar and in some cases identical to steps for transform mapping. A major difference lies in the mapping of DFDs to software structure.

Step 1 Review the elementary system model.

Step 2 Review and refine data flow diagrams for the software.

Step 3 Determine whether the DFD has transform or transaction flow characteristics.

Steps 1, 2, 3 are identical to corresponding steps in transform mapping.

Step 4 Identify the transaction center and the flow characteristics along each of the action paths.

The location of the transaction center can be immediately discerned (distinguish) from the DFD. The transaction center lies at the origin of a number of actions paths that flow radially from it. The incoming path and all action paths must also be isolated.

Step 5 Map the DFD in a program structure amenable to transaction processing.

Transaction flow is mapped into an architecture that contains an incoming branch and a dispatch branch. The structure of the incoming branch is developed in much the same way as transform mapping. Starting at the transaction center bubbles along the incoming paths are mapped into modules. The structure of the dispatch branch contains a dispatcher module which controls all subordinate action modules. Each action flow of the DFD is mapped to a structure that corresponds to its specific flow characteristics.

Step 6 Factor and refine the transaction structure and the structure of.

Each action path of the data flow diagram has its own information flow characteristics. We have already noted that transform or transaction flow may be encountered. The action path related structure is developed using the design steps discussed above.

Step 7 Refine the 1st iteration architecture.

This step for transaction mapping is identical to the corresponding step for transform mapping. In both design approaches, criteria such as module independence, practicality and maintainability must be carefully considered as structural modifications are proposed.

Verification

The software architecture captures early design decisions. Since these early decisions have a large impact it is important to start testing even at this early stage. Manul techniques such as reviews and inspections can obviously be applied at this stage. At this stage software architecture with respect to quality attributes such as maintainability, flexibility can be verified. This is achieved by developing a set of scenarios and evaluating the extent to which the architecture meets the quality requirements for each of these scenarios. The software architecture can be used to develop a skeletal version of the system. This skeletal version contains all of the architecture's components in a basic form. The skeletal system can be used as an environment for the incremental implementation of the system. It can be used as an environment for testing (verification) testing the system.

Software Detail Design

1. Software Design Fundamentals

The concepts, notions, and terminology introduced here form an underlying basis for understanding the role and scope of software design.

1.1. General Design Concepts

The designer's goal is to produce a model or representation of an entity that will later be built.

Software is not the only field where design is involved. In the common sense, we can view design as a form of problem solving. For example, the concept of a wicked problem–a problem with no definitive solution–is interesting in terms of understanding the limits of design. A number of other notions and concepts are also of interest in accepting design in its general sense: goals, alternatives , constraints, representations, and solutions.

1.2. Context of Software Design

To know the role of software design, it is important to understand the context in which it fits, in the life cycle of software engineering. It is important to understand the major characteristics of software requirements analysis vs. software design vs. software construction vs. software testing.

1.3. Software Design Process

Software design is an repetitive process during which requirements are translated into a blueprint for constructing the software. Initially the blue print depicts a holistic view of software. That is the design is represented at a high level of abstraction, a level that can be directly traced to the specific system objective and more detailed data, functional and behavioral requirements. Software design is generally considered a two-step process

1.3.1. Architectural design

Architectural design depicts how software is decomposed and organized into components (the software architecture). Software architecture mention indirectly to the overall structure of the software. The behavior in which that structure gives conceptual integrity for a system. In its simplest form, architecture is the hierarchical structure of program components (modules), the manner in which these components interact and the structure of data that are used by the components. Software architecture is the construction of work product that gives the maximum return on investment with respect to quality, schedule and cost. Software architecture comprises of structural, extra functional properties.

1.3.2. Detailed design

Detailed design describes the specific behavior of these components.
The output of this process is a set of models and artifacts that record the major decisions that have been taken.

1.4. Enabling Techniques

According to the Oxford English Dictionary, a principle is "a basic truth or a general law … that is used as a basis of reasoning or a guide to action." Software design principles, also called enabling techniques, are key notions considered fundamental to many different software design approaches and concepts.

1.4.1. Abstraction

Abstraction is "the process of forgetting information so that things that are different can be treated as if they were the same." Abstraction is one of the fundamental principles humans use to cope with complexity. Abstraction is defined as "The essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply distinct conceptual boundaries relative to the perspective of the viewer." In the context of software design, two type abstraction mechanisms are parameterization and specification. Abstraction by specification leads to three major kinds of abstraction: procedural concept, data abstraction, and control (iteration) abstraction. Example- The procedural abstraction –word open for a door. Open implies a long sequence of procedural steps-walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door etc. The data concept for door would be a set of attributes that describe the door means door type, swing direction, opening mechanism, weight, dimensions etc. The control abstraction-it implies a program control mechanism with no specifying internal details.

1.4.2. Coupling and Cohesion

Coupling is defined as the strength of the relationships among modules, whereas cohesion is defined by how the elements making up a module are related. Coupling focuses on inter-module aspects, while cohesion emphasizes intra-module characteristics.

Coupling measures the strength of association between one module and another. Strong coupling complicates a system, since a module is harder to understand, change, or to correct if it is extremely interrelated with other modules. Complexity can be reduced by designing systems with weak coupling between modules.
There are several forms of coupling:

- Low coupling
- Controlled coupling
- Common coupling
- High coupling

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. In nutshell cohesive module should do just one thing.
Cohesion measures the degree of connectivity between the functions and elements of a single module. There are several forms of cohesion:

- functional
- logical
- procedural
- communicational
- sequential
- informal

1.4.3. Decomposition and modularization

Decomposing and modularizing large software into a number of smaller independent ones, usually with the goal of placing different functionalities or responsibilities in different components. Modularization is concerned with the meaningful decomposition of a software system and with its grouping into subsystems and components. The major task is to decide how to physically package the entities that form the logical structure of an application. Modules serve as physical containers for functionalities or responsibilities of a system. Modularization is closely related to encapsulation.

1.4.4. Encapsulation/information hiding

The concepts of modularity lead every software designer to a fundamental question: “how do we decompose software to obtain the best set of modules?” Encapsulation/information hiding means grouping and packaging the fundamentals and internal details of an abstraction and making those details inaccessible. In other words modules should be specified and designed so that information contained within a module is in assessable to other modules that have no need for such information. Encapsulation deals with grouping the fundamentals of an abstraction that comprise its structure and behavior, and with separating different abstractions from each other. Encapsulation provides explicit barriers between abstractions, fostering non-functional properties like changeability and reusability. Information hiding involves concealing the details of a component's implementation from its clients, to better handle system complexity and to minimize the coupling between components. Any details of a component that clients do not need to know in order to use it properly should be hidden by the component.

1.4.5. Separation of interface and implementation

Separating interface and implementation involves defining a component by specifying a public interface, known to the clients, separate from the details of how the component is realized. Different or unrelated responsibilities should be separated from each other within a software system. Collaborating components that contribute to the solution of a specific task should be separated from components that are involved in the computation of other tasks.

1.4.6. Sufficiency, completeness and primitiveness

Achieving sufficiency, completeness and primitiveness means ensuring that a software component captures all the important characteristics of an abstraction, and nothing more.

Sufficiency -a element should capture those characteristics of an abstraction that are necessary to permit a meaningful and efficient interaction with that component.

Completeness -a component should capture all appropriate characteristics of its abstraction.

Primitiveness -all operations of a component can be implemented easily without dependencies and/or complexities.

Module Specification

1. The specification must provide to the intended user all the information that he will need to use the program correctly and nothing more.
2. The specification must provide to the implementer, all the information about the intended use that he needs to complete the program, and no additional information; in particular, no information about the structure of the calling program should be conveyed.
3. The specification must be sufficiently formal that it can conceivably be machine tested for consistency, completeness (in the sense of defining the outcome of all possible uses) and other desirable properties of a specification.

Note that we do not insist that machine testing be done, only that it could conceivably be done. By this requirement we intend to rule out all natural language specifications.'

4. The specification should discuss the program in the terms normally used by user and implementer alike rather than some other area of discourse. By this we intend to exclude the specification of programs in terms of the mappings they provide between large input domains and large output domains or their specification in terms of mappings onto little automata, etc. The basis of the technique is a view of a program module as a device with a set of switch inputs and readout indicators. The technique specifies the possible positions of the input switches and the effect of moving the switches on the values of the readout indicators. We insist that the values of the readout indicators be completely determined by the previous values of those indicators and the positions of the input switches. [Aside: The notation allows for some of the pushbuttons to be combined with indicator lights or readouts (with the result that we must press on a button in order to read), but we have not yet found occasion to use that facility. A simple extension of the notation allows the specification of mechanisms in which the values of the readout indicators are not determined by the above factors, but can be predicted only by knowing the values of certain "hidden" readout indicators which cannot actually be read by the user of the device. We have considerable doubts about the advisability of building devices which must be specified using this feature, but the ability to specify such devices is inexpensively gained.]

In software language we consider each module as providing a number of subroutines or functions which can cause changes in state, and other functions or procedures which can give to a user program the values of the variables making up that state. We refer to these all as functions. We distinguish two classes of readout functions: the most important class provides information which cannot be determined without calling that function unless the user maintains duplicate information in his own program's data structures. A second class, termed mapping functions, provides redundant information, in that the value of these functions is completely predictable from the current values of other readout functions. The mapping functions are provided as a notational convenience to keep the specifications and the user programs smaller. For each function we specify:

I. The set of possible values: (integers, real, truth values, etc.).

2. Initial values: (either "undefined" or a member of the set specified in item 1). "Undefined" is considered a special value, rather than an unpredictable value.

3. Parameters: each parameter is specified as belonging to one of the sets named in item 1.

4. Effect: with the exception of mapping functions, almost all the information in the specification is contained. Under "effect": we place two distinct types of items which require a more detailed discussion. First, we state that if the "effect" section is empty, then there is absolutely no way to detect that the function has been called. One may call it arbitrarily often and monitor no effect except the passage of time. The modules that we have specified have "traps" built in. There is a sequence of statements in the "effect" section which specifies the conditions under which certain "error" handling routines will be called. These conditions are treated as incorrect usage of the module and response is considered to be the responsibility of the calling program. For that reason it is assumed that the "error" handling routine's body will not be considered part of the module specified, but will be written by the users of the module. If such a condition occurs, there is to be no observable result of the call of the routine except the transfer of control. When there is a sequence of error statements, the first one in the list which applies is the only one which is invoked. In some cases, the calling program will correct its error and return to have the function try again; in others, it will not. If it does return, the function is to behave as if this were the first call. There is no memory of the erroneous call. This approach to error handling is provoked by two considerations which are peripheral (minor) to this paper. First, we desire to make it possible to write the code for the "normal" cases without checking for the occurrence of unusual or erroneous situations. The "trap" approach helps this. Second, we wish to promote the proper handling of errors in many-leveled software. In our opinion this implies that each routine receives all messages from the routines that it uses and either (1) hides the trap from its user or (2) passes to its user an error indication which is meaningful to a program which knows only the specification of the routine that it called and does not know of the existence of routines called by that routine. The reader will find that our insistence that (1) response to errors is the responsibility of any routine which called another routine in an "incorrect" way and (2) that when such an error call is made; there is no record of the previous call, places quite a demand on the implementers of each module. They must not make irreversible changes unless they are certain that they can complete the changes to be made without calling any "error" routines**.** The reader will note that we generally specify a separate routine for each case which might be handled separately. The user may make several routines have identical bodies, if the distinction between those cases is not important to him. The remaining statements are sequence independent. They can be "shuffled" without changing their meaning. These statements are equations describing the values (after the function call) of the other functions in the module. It is specified that no changes in any functions (other than mapping functions) occur unless they are implied by the effect section. The effect section can refer only to values of the function parameters and values of readout functions. The value changes of the mapping functions are not mentioned; those changes can be derived from the changes in the functions used in the definitions of the mapping functions. All this will have much clarity as we discuss the subsequent examples. In some cases we may specify the effect of a sequence to be null. Using this we imply that sequence may be inserted in any other sequence without changing the effect of the other sequence.

Using the Specifications

The specifications will be of maximum helpfulness only if we accept methods that make full use of them. Our aim has been to produce specifications which are in a real logic just as testable as programs. We will gain the most in our system building abilities if we have a technique for usage of the specifications which involves testing the specifications long before the programs specified are produced. The statements being made at this level are precise enough that we should not have to wait for a lower level representation in order to find the errors. Such specifications are at least as demanding of precision as are programs; they may well be as complex as some programs. Thus they are as likely to be in error. Because specifications cannot be "run (execute)," we may be tempted to postpone their testing until we have programs and can run them. For many reasons such an approach is wrong. We are able to test such specifications because they provide us with a set of

saying for a formal deductive scheme. As a result, we may be able to prove certain "theorems" about our specifications.
By asking the proper set of such questions, the "correctness" of a set of specifications may be verified. The meaning of "correctness," is dependent on the nature of the object being specified. Using the same approach of taking the specifications as saying and attempting to prove theorems, one may ask questions about possible changes in system structure. For example, one may ask which modules will have to be altered, if certain restrictions assumed before were removed. It would be obviously useful if there were a support system which would input the specifications and provide question answering ability above the specifications. What is necessary is that system builders develop the habit of verifying the specifications whether by machine or by manually before constructing and debugging the programs. Incidentally, the theorem proving approach might also be considered as a basis for a program which searches automatically for implementations of a specified module.

Interface Design-

Screen Design/ User Interface Design
The screen design should be visually appealing. It should be simple and neat. Web pages use dazzling colors and lots of images to make them attractive, this type of design takes a longer time to load the pages.
There are three basic rules 1) Place the user in control 2) Reduce the user memory load 3) Make the interfaces consistent.
Place the user in control:-
He/ She wanted that a system that reacted to his/ her needs and helped his/ her get things done. He/ She wanted to control the computer, not the computer control him/her.
There are certain principles to allow the user to maintain control:-

a) Provide flexible interaction
b) Allow user interaction to be interruptible and undoable
c) Streamline interaction as skill levels advance and make it customizable
d) Hide technical internals from the casual user

Reduce the user memory load
The more the user has to remember, the more error prone will be the interaction with the system. Whenever possible system should remember information and assist the user with an interaction scenario that assist recall.
There are certain principles to enable an interface to reduce the user memory load:-

a) Reduce demand on short term memory
b) Establish meaningful defaults
c) Define shortcuts that are intuitive
d) The visual layout of the interface should be based on a real world image
e) Disclose information in a progressive fashion

Make the interface consistent
The interface should present and acquire information in a consistent fashion. That means all visual information is organized according a design standard that maintained throughout all screen displays, input mechanisms are constrained to a limited set that used consistently throughout the application and mechanisms for navigations from task to task are consistently defined and implemented.
There are certain principles to help make the interface consistent:-

a) Allow the user to put the current task into a meaningful context
b) Maintain consistency across a family of application
c) If prototype/demo interactive models have created user expectations do not make changes unless there is a strong reason to do so. For example **Ctrl+S** shortcut used for **Save** then **Ctrl+S** shortcut should not be used for **Send** in new version.

All these interface design principles discussed above provide basic guidance for a software engineer.

Therefore, while designing pages, you should keep the following guidelines in mind:
The controls that need user input should have the correct tab order and should be grouped and arranged in an order that makes sense while entering data. The controls should be properly aligned.

UI Design Recommendations
Some of the common recommendations about interfaces include:

- Visibility: Every available operation/feature should be perceptible, either by being currently displayed or was recently shown so that it has not faded from user's short-term memory. Visible system features are called affordances;
- Transparency: Expose the system state at every moment. For example, when using different tools in manipulation, it is common to change the cursor shape to make the user aware of the current manipulation mode (rotation, scaling, or such);
- Consistency: Whenever possible, comparable operations should be activated in the same way (although, see [Error! Reference source not found.] for some cautionary remarks). Or, stated equivalently, any interface objects that look the same are the same;

- Reversibility: Include mechanisms to recover a prior state in case of user or system errors (for example, undo/redo mechanism). Related to this, user action should be interruptible, in case the user at any point changes their mind, and allowed to be redone;
- Intuitiveness: Design system behavior to minimize the amount of surprise experienced by the target user (here is where metaphors (decomposition) and analogies in the user interface can help);
- Guidance: Provide meaningful feedback when errors occur so the user will know what follow-up action to perform; also provide context-sensitive user help facilities. Obviously, the software engineer should not adopt certain design just because it is convenient to implement. The human aspect of the interface must be foremost. Experience has shown that, although users tend at first to express preference for good looks, they
- Eventually come around to value experience. The interface designer should, therefore, be more interested in how the interface feels, than in its aesthetics. What something looks like should come after a very detailed conversation about what it will do.

Variables and objects
While naming variables and objects, keep the following guidelines in mind:

- Use a proper naming notation, such as Hungarian or camel-casing notation, to name variables and objects. Hungarian notation enables you to identify the datatype of the variable from the name of the variable. So, a variable storing the first name of an employee will be declared as sFirstName. In camel-casing notation, the variable names take the screen sfirstName, with the second part of the variable, which is a noun, capitalized.
- Name the variables and objects meaningfully. Meaningful names combined with Hungarian notation make the purpose and type of the variables clear. This results in a self-documented code, which is easy to understand and maintain.
- Declare the variables and objects in the beginning of a procedure. Declaration in the beginning makes the code execution more efficient, besides making it easy to understand by someone looking at the code text.
- Always initialize variables to assured default values before using them, to avoid any type conversion issues.
- Always rely on explicit conversion functions to eliminate confusion.

Currently the following types of user interface are the most common:

- **Graphical user interfaces** (GUI) accept input via devices such as computer keyboard and mouse and provide articulated graphical output on the computer monitor. In GUI design two different principles widely used: Object-oriented user interfaces (OOUIs) and application oriented interfaces.
- **Web-based user interfaces** or **web user interfaces** (WUI) accept input and provide output by generating web pages which are transmitted via the Internet and viewed by the user using a web browser program. Newer implementations utilize Java, AJAX, Adobe Flex, Microsoft .NET, or similar technologies to provide real-time control in a separate program, eliminating the need to refresh a traditional HTML based web browser. Administrative web interfaces for web-servers, servers and networked computers are often termed Control panels.

User interfaces that are common in various fields outside desktop computing:

- **Command line interfaces**, where the user provides the input by typing a command string with the computer keyboard and the system provides output by printing text on the computer monitor. Used by programmers and system administrators, in engineering and scientific environments, and by technically advanced personal computer users.
- **Tactile interfaces** supplement or replace other forms of output with haptic (sense of touch) feedback methods. Used in computerized simulators etc.
- **Touch user interface** are graphical user interfaces using a touch screen display as a combined input and output device. Used in many types of point of sale, industrial processes and machines, self-service machines etc.

Summary

A set of principles for software design:

- The design process should not suffer from “tunnel vision”.
- The design should be traceable to the analysis model.
- The design should not reinvent the wheel.
- The design should “minimize the intellectual distance” between the software and the problem in the real world.
- The design should exhibit uniformity and integration.
- The design should be structured to accommodate change.
- The design should be structured to degrade gently.
- Design is not coding.
- The design should be assessed for quality.
- The design should review to minimize conceptual errors.

External quality factors: observed by users. Internal quality factors: important to engineers

Information hiding:

Modules should be specified and designed so that the internal details of modules should be invisible or inaccessible to other modules.

Major benefits: reduce the change impacts in testing and maintenance

Functional independence:

Design modules based on independent functional features

Major benefits: effective modularity

Cohesion: a natural extension of the information hiding concept a module may perform a number of tasks.

A cohesive module performs a single task in a procedure with little interactions with others.

Goal: to achieve high cohesion for modules in a system.

Different types of cohesion*:*

- Coincidentally cohesive: a set of tasks related to each other loosely
- Logical connection among processing elements ---> logically cohesive:
- Data sharing among processing elements --> communication cohesion
-Order among processing elements --> procedural cohesion

Modularity (in program and data) and the concept of abstraction enable the designer to simplify and reuse software workings. Refinement provides a mechanism for representing successive layers of functional detail. Program and data structure contribute to an overall view of software architecture and procedure provides the detail necessary for algorithm implementation.

The software engineer must represent data structures, interfaces and algorithms insufficient detail to guide in the generation of programming language source code, at component level. To accomplish this, the designer uses one of a number of design notations that represent component level detail either in graphical, tabular, or text based formats.

Software architecture provides a holistic view of the system to be built. It depicts (shows) the structure and organization of software components, their properties and the connection between them. Software components include program modules and the various data representations that are manipulated by the program. Data design is an integral part of the derivation of the software architecture.

The architectural design method presented in this lesson uses data flow characteristics described in the analysis model to derive a commonly used architectural style. A data flow diagram is mapped into program structure using one of two mapping approaches transform mapping or transaction mapping. Transform mapping is applied to an information flow that exhibits distinct boundaries between incoming and outgoing data. The DFD is mapped into a structure that allocates control to input, processing and output along three separately factored module hierarchies. Transaction mapping is applied when a single information item causes flow to branch along one of many paths. The DFD is mapped into a structure that allocates control to a substructure that acquires and evaluates a transaction. Another substructure controls all potential processing actions based on transaction.

The software architecture is an extension of software design which explains how the software system design components are organized and actually implemented using links for integration and interfaces for communication and operations with the goal of achieving the highest design effectiveness. The objective of software architecture is that the software design is better understood from the point of view of the development. It enables to deliver the user requirements, stakeholders, and benefits and meets non functional requirements of the users and that of the system. Software architecture is the expansion of work product that gives the highest return on investment with respect to quality, schedule and cost. Software architecture comprises of structural , extra functional

properties and families of related systems. The designing is an intelligent activity and architecting the software is a creative process which ensures delivery of non -functional and functional software system requirements. The goal of architecture is to describe the system design to the developer, user and customer to create a common understanding of the proposed system architecture. Coupling is a measure of interconnection among modules in a software structure. The coupling modes occur because of design decisions made when structure was developed. Cohesion measures the degree of connectivity between the functions and elements of a single module. It is important to strive for high cohesion and recognize low cohesion so that software design can be modified to achieve greater functional independence.

The user interface is arguably the most important element of a computer based system. If the interface is poorly designed the user's ability to tap the computational power of an application may be severely hindered (delayed). In fact a weak interface may cause an otherwise well designed and solidly implemented application to fail. Three basic rules guide the design of effective user interfaces 1) Place the user in control 2) Reduce the user memory load 3) Make the interfaces consistent. User interface design begins with the identification of user, task, and environmental requirements. Task analysis is a design activity that defines user tasks and actions using either elaborative or object oriented approach.

The user interface is the window into the software. The interface molds a user's perception of quality of the system. If the window is wavy or broken, the user may reject an otherwise powerful computer based system.

Self Test

41. Structured design is often characterized as a data flow design method because it provides convenient transition from ----------.
    a) a data flow diagram to software architecture b) a data flow diagram c) a software architecture

42. The information flow can have ---------- and ---------- types.
    a) transform flow b) transaction flow c) intermittent flow
43. In Transform mapping, factoring results in a program structure in which top level modules perform ---------- and low level modules perform ---------- and output work.
    a) decision making b) most input, computation c) moderate amounts of work d) a and b
44. In Transform mapping, middle level modules perform some control and do ----------.
    a) moderate amounts of work b)decision making c) most input, computation
45. The design step begins with an evaluation of work done during requirements analysis. State whether this statement is true or false.
    a) True. b) False
46. The information must enter and exit software in an external world form. State whether this statement is true or false.
    a) True. b) False
47. ---------- flow is characterized by data moving along an incoming path that converts external world information into a transaction.
48. The ---------- is often characterized by a single data item called ----------, which triggers other data flow along one of many paths.
    a) Transaction flow b) Transform c) Transaction d) Transform flow
49. Design is a problem solving ---------and as such very much matter of -----and error.
    a) activity

b) trial
c) test
d) action

50. The out come of design process (the blue print) will be termed as ------and its notation, termed as the (technical) ----------.
    a) design
    b) trial
    c) specification
    d) action
51. Software design is an ---------process through which requirements are translated into a blueprint for constructing the software.
    a) iterative
    b) repetitive
    c) one time
    d) none of the above

52. Software design is generally considered a ----------step process.
    a) two
    b) three
    c) one
    d) none of the above

53. Software design comprises of ---------- and ----------.
    a) detail design
    b) architecture design
    c) coupling
    d) cohesion
54. Architectural design describes how software is ----------and organized into components.
    a) decomposed
    b) composed
    c) breakdown
    d) none of the above

55. Detailed design describes the ----------of these components.
    a) specific behavior
    b) behavior
    c) relation
    d) none of the above

56. Abstraction is one of the fundamental principles humans use to cope with----------.
    a) complexity
    b) behavior
    c) relation
    d) none of the above

57. Abstraction by specification leads to ---------- major kinds of abstraction.
    a) two
    b) three
    c) one
    d) none of the above

58. Coupling is defined as the ----------of the relationships between modules.
    a) strength
    b) weakness
    c) strong
    d) none of the above

59. Cohesion is defined by how the ----------making up a module are related.
    a) éléments
    b) parts
    c) components
    d) none of the above

60. The specification --------provide to the intended user all the information that he will need to use the program correctly and nothing more.
    a) must
    b) should
    c) can
    d) none of the above

Lesson 4

Coding and Programming Practices

Overview

This chapter discusses the portion of the software development process where the design is elaborated and the individual data elements and operations are designed in detail. Much of the chapter discussion is focused on notations that can be used to represent low level procedural designs. Students should be encouraged to experiment with different design notations and try to find one that suits their programming style and aesthetic sense. Object-oriented methods for component level design appear later in the text.

The purpose of the implementation phase is to build a total, high-quality software system from the "blueprint" provided in the detailed design document. The implementation phase begins after Critical Design Review (CDR) and proceeds according to the build plan prepared during the detailed design phase. For each build, character programmers code and test the units identified as belonging to the build, integrate the units into modules, and test module interfaces. At the same time, the application specialists on the development team prepare plans designed to test the functional capabilities of the build. Build regression tests — a selection of tests already conducted in previous builds — are included in each build test plan to ensure that newly added capabilities have not affected functions implemented previously.

Learning Objectives

- To understand Logic/ Algorithm
- To understand Coding
- To understand Programming practice
- To understand Structured Programming

Introduction - Algorithm / Logic Design

Algorithm design is a specific method to create a mathematical process in solving troubles. Applied algorithm design is algorithm engineering**.** Algorithm design is identified and incorporated into many solution theories of operation research, such as dynamic programming and divide-and-conquer. Techniques for designing and implementing algorithm designs are algorithm design patterns, such as template method patterns and decorator patterns, and uses of data structures, and name and sort lists. Some current day uses of algorithm design can be found in internet retrieval processes of web crawling packet routing and caching.

Classification

• Divide and conquer. A divide and conquer algorithm repeatedly reduces an instance of a problem to one or smaller instances of the same problem until the instances are small enough to solve easily. One such example of divide and conquer is merge sorting. Sorting can be done on each segment of

data after dividing data into segments and sorting of entire data can be obtained in the conquer phase by merging the segments. A simpler variant of divide and conquer is called a decrease and conquer algorithm, that solves an identical subproblem and uses the solution of this subproblem to solve the bigger problem. Divide and conquer divides the problem into multiple subproblems and so conquer stage will be more complex than decrease and conquer algorithms. An example of decrease and conquer algorithm is the binary search algorithm.

- Dynamic programming. When a problem shows optimal substructure, meaning the optimal solution to a problem can be constructed from optimal solutions to subproblems, and overlapping subproblems, meaning the same subproblems are used to solve many unlike problem instances, a quicker approach called dynamic programming avoids recomputing solutions that have already been computed. For example, the shortest path to a goal from a vertex in a weighted graph can be found by using the shortest path to the goal from all adjoining vertices. Dynamic programming and optimization go together. The main difference among dynamic programming and divide and conquer is that subproblems are more or less independent in divide and conquer, whereas subproblems overlap in dynamic programming. The difference between dynamic programming and straightforward recursion is in caching or optimization of recursive calls. When subproblems are independent and there is no repetition, optimization does not help; hence dynamic programming is not a solution for all complex problems. By using optimization or maintaining a table of subproblems already solved, dynamic programming reduces the exponential nature of many problems to polynomial complexity.

Logic is the study of the principles of valid demonstration and inference. Logic is a branch of philosophy. Logic is also commonly used today in argumentation theory.

For example – the process password transform performs all password validation for the computer application system. Process password receives a four-digit password from the interact with user function. The password is first compared to the master password stored within the system. If the master password matches “valid id message = true” is passed to the message and display function. If the master password do not matches, the four digits are compared to a table of secondary passwords (when temporary passwords issued) If the password matches an entry within the table “valid id message = true” is passed to the message and display function. If there is no match “valid id message = false” is passed to the message and status display function.

Importance of Implementation Phase

In implementation phase software product start to build a absolute, high-quality software system from the "blueprint" provided in the detailed design document. The implementation phase starts after Critical Design Review and proceeds according to the build plan prepared during the detailed design phase.

The system is realized through implementation producing the sources (source-code files, header files, make files, and so on) that will result in an executable system. The sources are described in an implementation model that consists of modules structured into implementation packages. The design model is the basis for implementation. In this stage the software design is realized as a set of working programs or program units. The unit-testing means verifying each unit that meets its specifications and it is verified that the defined input produces the desired results.

Implementation includes testing the separate classes and/or packages, but not testing that the packages/classes work together.

Coding

Construction is an activity in which the software has to come to terms with arbitrary and chaotic real-world constraints, and to do so exactly. Due to its proximity to real-world constraints, construction is more driven by practical considerations and software engineering is perhaps most craft-like in the construction area.

Construction Design

Some projects allocate more design activity to construction; others to a phase explicitly focused on design. In spite of the correct allocation, some in depth design work will occur at the construction level, and that design work tends to be dictated by immovable constraints imposed by the real-world problem that is being addressed by the software. Just as construction workers building a physical structure must make small-scale changes to account for unanticipated gaps in the builder's plans, software construction workers must make modifications on a smaller or larger scale to flesh out details of the software design during construction.

Construction Languages

Construction languages include all forms of communication by which a human can specify an executable problem solution to a computer. The simplest type of construction language is a configuration language, in which software engineers choose from a limited set of predefined options to create new or custom software installations. The text-based configuration documents used in both the Windows and Unix operating systems are examples of this, and the menu style selection lists of some program generators comprises another. To build applications out of toolkits, toolkit languages are used (integrated sets of application-specific reusable parts), and are more difficult than configuration languages. Toolkit languages may be explicitly defined as application programming languages (for example, scripts), or may simply be implied by the set of interfaces of a toolkit. Programming languages are the most flexible type of construction languages. They also contain the least amount of information about specific application areas and development processes, and so require the most training and skill to use effectively.

There are three general kinds of notation used for programming languages, namely:

- Linguistic
- Formal
- Visual

Linguistic notations are distinguished in particular by the use of word-like strings of text to represent complex software constructions, and the combination of such word like strings into patterns that have a sentence-like syntax. Properly used, each such string should have a strong semantic connotation providing an immediate intuitive understanding of what will happen when the underlying software construction is executed.

Formal notations depend less on intuitive, everyday meanings of words and text strings and more on definitions backed up by precise, unambiguous, and formal (or algebraic) definitions. Formal construction notations and formal methods are at the heart of most forms of system programming, where time behavior, accuracy and testability are more important than ease of mapping into natural language. Formal constructions also use precisely defined ways of combining symbols that avoid the ambiguity of many natural language constructions.

Visual notations rely much less on the text-oriented notations of both linguistic and formal construction, and instead rely on direct visual interpretation and placement of visual entities that represent the underlying software. Visual construction tends to be somewhat limited by the difficulty of making "complex"statements using only movement of visual entities on a display. However, it can also be a powerful tool in cases where the primary programming task is simply to build and "adjust" a visual interface to a program, the detailed behavior of which has been defined earlier.

This Coding area is concerned with knowledge about the construction of the software components that are identified and described in the design documents. This Coding area includes knowledge about translation of a design into an implementation language, program coding styles, and the development and use of program documentation.

Code Implementation

This Code Implementation unit is concerned with knowledge regarding how to convert a software design into an implementation programming language. This unit includes knowledge about modular and incremental programming, structured programming, and knowledge of various programming paradigms (assembly, procedural, object-oriented, functional, and logic). It also has knowledge about how to apply source code development tools and programming language translation tools.

Code Reuse

This Code Reuse unit is worried with knowledge about developing code by reuse of existing components and about developing reusable code. This unit also has knowledge about reusable libraries, the inheritance mechanism, module referencing, and software portability issues and techniques

Coding Standards and Documentation

This unit is concerned with knowledge about the use of standards for style and documentation in the construction of software. This unit has knowledge about how to develop internal and external program documentation.

Inspecting for execution to coding standards is certainly the most well-known of all features. The first action to be taken is to define or accept a coding standard. Usually a coding standard consists of a set of programming rules (e.g. 'Always check boundaries on an array when copying to that array'), naming conventions (e.g. 'Classes should start with capital C) and layout specifications (e.g. 'Indent 4 spaces'). It is recommended that existing standards are followed. The main advantage of this is that it saves a lot of effort. An extra reason for adopting this approach is that if you take a well-known coding standard there will probably be inspecting tools available that support this standard. It can even be put the other way around: purchase a static code analyzer and declare (a selection of) the rules in it as your coding standard. Lacking such tools, the enforcement of a coding standard in an organization is probable to fail. There are three main causes for this: the number of rules in a coding standard is usually so huge that nobody can remember them all; some context-sensitive rules that insist reviews of several files are very hard to check by human beings; and if people spend time inspecting coding standards in reviews, that will divert them from other defects they might otherwise find, making the review process less effective.

Activities of the Development Team:

- **Code new units** from the detailed design specifications and revise existing units that require modification. Code units so that each PDL statement can be easily matched with a set of coding statements. Use structured coding principles and local coding conventions. Prepare the command language procedures needed to execute the units.
- **Read new and revised units.** Ensure that each unit is read by a minimum of two members of the development team who are not the unit's authors. Correct any errors that are found, re-inspecting the unit as necessary. Certify all unit code.
- **Test each unit and module.** Prepare unit test procedures and data, and conduct unit tests. If the acceptance test team has provided an analytical test plan for the unit, complete the test cases specified in the plan and verify that the computational results are as expected. Have an experienced member of the development team review and certify all unit test procedures and results. Integrate logically related units into modules and integrate the modules into the growing build. Define and run enough tests to verify the I/O generated by the module and the interfaces

among units within the module. Ensure that the results of module testing are reviewed and certified.

- **Plan and conduct build tests.** Prepare the test plan for the build, and finish the preparation of command procedures and data needed for build testing.

The following considerations apply to the software construction coding activity:

- Techniques for creating reasonable source code, including naming and source code layout
- Use of enumerated types, classes, variables, named constants, and other similar entities
- Use of control structures
- Handling of error conditions—both planned errors and exceptions (input of bad data, for example)
- Prevention of code-level safety breaches (for example buffer overruns or array index overflows,)
- Resource usage via use of exclusion mechanisms and regulation in accessing serially reusable resources (including database locks or threads)

- Source code organization (into statements, routines, classes, packages, or other structures)
- Code documentation
- Code tuning

Code Reading

The first step in the unit verification process is code reading, a systematic procedure for examining and understanding the operation of a program. The Software Engineering Laboratory (SEL) has found code reading to be more cost effective in uncovering defects in software than either functional or structural testing and has formalized the code reading process as a key implementation technique.

Code reading is designed to verify the logic of the unit, the flow of control within the unit, and boundary conditions. It is conducted before unit testing, not afterwards. Only code that has compiled cleanly should be presented for code reading. Every new or modified unit is read by two or more than two team members. Each reader individually examines and annotates the code, reading it line by line to uncover faults in the unit's interfaces, control flow, logic, conformance to program design language (PDL), and adherence to coding standards. A checklist that is used by code readers on SEL-monitored projects is shown in Figure 16.1; its use fosters consistency in code reading by ensuring that the reader has a list of typical errors to look for and specific points to verify. The readers and the unit's developer then meet as a team to review the results of the reading and to identify problems that must be resolved.

They also inspect the test plan for the unit. If errors have been discovered in the unit, the reader who is leading the meeting (the moderator) returns the unit to the implementer for correction. The unit is then rechecked. When all errors have been determined, the moderator certifies that the code is satisfactory and signs the checklist. The implementer files the certified checklist in the software engineering notebook (SEN) for the unit.

Coding- Although coding is a mechanistic product of procedural design, errors can be introduced as the design is converted into a programming language. This is particularly accurate if the programming language not supporting data directly and control structures represented in the design. A code walkthrough can be an effective means for uncovering these translation errors. The checklist that follows assumes that a design walkthrough has been conducted and that algorithm rightness has been established as part of the design formal technical review.

Figure 16.1 Sample Checklists for Code Inspection

UNIT CODE INSPECTION CHECKLIST

Unit Name_____System_____Build/Release_____
Task Number_____Initial Inspection Date_____
Inspection Moderator_____

KEY INSPECTION QUESTIONS	Yes	No	Corrected
1. Is any input argument unused? Is any output argument not produced?	[ ]	[ ]	[ ]
2. Is any data type incorrect or inconsistent?	[ ]	[ ]	[ ]
3. Is any coded algorithm inconsistent with an algorithm explicitly stipulated in PDL or in requirements/specifications?	[ ]	[ ]	[ ]
4. Is any local variable used before it is initialized?	[ ]	[ ]	[ ]
5. Is any external interface incorrectly coded? That is, is any call statement or file/database access incorrectly coded? Also, for an Ada unit, is any external interface not explicitly referenced/with'd-in?	[ ]	[ ]	[ ]
6. Is any logic path incorrect?	[ ]	[ ]	[ ]
7. Does the unit have multiple entry points or multiple, normal (non-error) exits?	[ ]	[ ]	[ ]
ADDITIONAL INSPECTION QUESTIONS			
8. Is any part of the code inconsistent with the unit design specified in the prolog and PDL?	[ ]	[ ]	[ ]
9. Does the code or test plan contain any unauthorized deviations from project standards?	[ ]	[ ]	[ ]
10. Does the code contain any error messages that might be unclear to the user?	[ ]	[ ]	[ ]
11. If the unit was designed to be reusable, has any hindrance to reuse been introduced in the code?	[ ]	[ ]	[ ]

ACTION ITEMS AND COMMENTS
(List on a separate sheet. Refer to questions above by number.)

INSPECTION RESULTS

1. If all answers to 1-11 were "No, " the unit's code passes. Check here and sign below. [ ]
2. If there are serious deficiencies in the code (e.g., if more than one key question was answered "Yes") the author must correct the unit design and the moderator must schedule a reinspection.
   Scheduled date for reinspection:_____
3. If there are minor deficiencies in the code, the author must correct the unit design and hold a followup meeting with the moderator.
   Scheduled date for followup meeting:_____

Moderator's signature certifies that this unit meets all applicable standards and satisfies its requirements, and that any identified deficiencies have been resolved (applicable at initial inspection, followup meeting, or reinspection).

Moderator Signature:_____ Date:_____

Design walk through

The definition of testing outlines objectives that relate to evaluation, revealing defects and quality. As indicated in the definition two approaches can be used to achieve these objectives, **dynamic testing** and **static testing.**

Using dynamic testing methods, software is tested/executed using a set of input values and its output is then examined and compared to what is anticipated. During static testing, software work products are examined manually, or with a set of tools, but not executed.

Dynamic testing and static testing are complementary methods, as they tend to find different types of defects effectively and efficiently. Types of defects that are easier to find during static testing are: deviations from missing requirements, standards, design defects, non-maintainable code and incompatible interface specifications. Note that in comparision to dynamic testing, static testing finds defects rather than failures.

In addition to finding defects, the objectives of reviews are also informational, educational and communicational whereby participants learn about the content of software work products to help them understand the role of their own work and to plan for future stages of development. Reviews often represent project milestones, and support the establishment of a baseline for a software product. The type and quantity of defects found during reviews can also help testers focus their testing and select effective classes of tests. In some cases customers/users attend the review meeting and provide feedback to the development team, so reviews are also a means of customer/user communication.

Studies have shown that as a result of reviews, a significant increase in productivity and product quality can be achieved. Reducing the number of defects early in the product life cycle also means that less time has to be spent on testing and maintenance.

To summarize, the use of static testing, e.g. reviews, on software work products has various advantages:

- Since static testing can start early in the life cycle, early feedback on quality issues can be established, e.g. an early validation of user requirements and not just late in the life cycle during acceptance testing.
- By detecting defects at an early stage, rework costs are most often relatively low and thus a relatively cheap improvement of the quality of software products can be achieved.
- Since rework effort is considerably minimized, development productivity figures are expected to increase.
- The evaluation by a team has the additional benefit that there is an exchange of information among the participants.
- Static tests add to an increased awareness of quality issues.

Produce high-level diagrams of the chosen system design and walk the analysts of the requirements definition team through them. Use the high-level design diagrams to brief the system process flow from the analyst's perspective. Focus on system and subsystem interfaces. Explain refinements to the operations scenarios arising from analysis activities and include preliminary versions of user screen and report formats in the walkthrough materials.

Walkthroughs are primarily conducted as an aid to understanding, so participants are encouraged to analyze and question the material under discussion. Review materials are distributed to participants before the meeting. During the meeting, the walk-through leader gives a brief, tutorial overview of the product, then walks the reviewers through the materials step-by-step. An informal atmosphere and a free interchange of questions and answers with participants encourage the learning process.

Design Walkthroughs

Walk through each major function or object with other developers and the requirements definition team. During the detailed design phase, twice design walkthroughs are held for each subsystem, one in the early stages of detailed design and one later in the phase. Participants comprises of members of the development team, the requirements definition team, representatives of systems. At these walkthroughs, members of the development team step through the subsystem's design, explaining its algorithms, interfaces, and operational aspects. Participants examine and question the design at a detailed level to uncover such issues as incompatible interfaces, contradictions among algorithms, or prospective performance problems. The design of the user interface is specifically addressed in one or more additional walkthroughs. These sessions are attended by the future users of the system and concentrate on the users' point of view. The users learn how the system will appear to them under representative operational scenarios, give feedback to developers, and provide a "reality check" on the restructured requirements and specifications. Problems encountered with the specifications are documented on question-and-answer sheets and submitted to the requirements definition team for further action.

Developers conduct design walkthroughs to ensure that both the requirements definition and development teams understand the system design as it takes shape. Developers distribute materials prior to the walk-through to participants, who include other developers, analysts, user representatives, systems going to use will interact with the software under development, and managers. During the meeting, developers cautiously brief how functional/operational aspects of the system (processing sequences and interfaces, screen formats, and user inputs) are reflected in the emerging design. Participants comment on how completely and accurately developers have interpreted the system requirements. A recording assistance records discrepancies, errors, and inconsistencies and records action items. If significant issues remain at the end of the walk-through, a follow-up session may be scheduled. The first walk-through presents the overall, system level consideration and is later followed by walkthroughs of subordinate systems and their main parts. When a project is huge and the development team has been divided into groups, it is important that the group head of all subordinate system should attend the session.

Critical Design Review

At the critical design review (CDR), in nutshell, the detailed design is examined to determine whether levels of detail and completeness are sufficient for coding to begin. The detailed design phase culminates in the CDR. This review is attended by the development team and its managers, the requirements definition team and its quality assurance representatives, managers, user representatives, the CCB (Change Control Board), and other stake holders of the system. Participants evaluate the detailed design of the system to determine whether the design is sufficiently correct and complete for implementation to begin. They review the build plan to confirm that the implementation schedule and the capabilities allocated to the builds are feasible.

The emphasis at CDR is on modifications —to high-level designs, requirements, system operations, and development plans — made since the PDR (preliminary design review). Speakers should highlight these changes both on the slides and during their presentations, so that they become the focus of the review. The CDR gives an opportunity for the development team to raise issues that are of concern to management, the project office, quality assurance staff, and the CCB.

At the CDR, developers present statistics showing the number and percentage of components to be reused, and which of these are drawn from the RSL (reusable software library). They also present key points of the detailed reuse strategy, identify any changes to the reuse proposal that have been made since PDR, and describe new/revised reuse tradeoff (exchange) analyses.

CDR format is given below in Figure 15.1. An outline and suggested contents of the CDR Hardcopy material are presented in Figure 15.2. For this concise format to be effective, participants

must be familiar with the project background, requirements, and design. They should have attended the PDR and studied the detailed design document before the meeting.

Reviewers should address the following questions:

- Does the design satisfy all requirements and specifications?
- Are the operational scenarios acceptable?
- Is the design correct? Will the transformations specified produce the correct output from the input?
- Is the design robust? Is user input examined for potential errors before processing continues?
- Have all design guidelines and standards been followed? How well have data usage and access been localized? Has coupling between units (i.e., inter unit dependency) been minimized?
- Is each unit internally cohesive (i.e., does it serve a single purpose)?
- Is the design testable?
- Is the build schedule structured to provide early testing of end to end system capabilities? Is the schedule reasonable and feasible for implementing the design?

CDR Format
Presenters — software development team
Participants
• Requirements definition team
• Quality assurance representatives from both teams
• Customer interfaces for both teams
• User representatives
• Representatives of interfacing systems
• System capacity/performance analysts
• CCB
Attendees should be familiar with the project background, requirements, and design.
Schedule — after the detailed design is completed and before implementation is begun
Agenda — selective presentation of the detailed design of the system. Emphasis should be given to changes to the high-level design, system operations, development plan, etc. since PDR.
Materials Distribution
• The detailed design report is distributed at least 10 days before the CDR.
• Hardcopy material is distributed a minimum of 3 days before the review

Figure 15.1

HARDCOPY MATERIAL FOR THE CDR

1. Agenda — outline of review material
2. Introduction — background of the project, purpose of the system, and an agenda outlining review materials to be presented
3. Design overview — major design changes since PDR (preliminary design review) (with justifications)
a. Design diagrams, showing products generated, interconnections among subsystems, external interfaces
b. Mapping of external interfaces to ICDs and ICD status (interface control document)
4. Results of prototyping efforts
5. Changes to system operation since PDR
a. Updated operations scenarios/scripts
b. System performance considerations
6. Changes to major software components since PDR (with justifications)
7. Requirements traceability matrix mapping requirements to major components
8. Software reuse strategy
a. Changes to the reuse proposal since PDR
b. New/revised reuse tradeoff analyses
c. Key points of the detailed reuse strategy, including software components to be reused in future projects
d. Summary of RSL contributions — what is used, what is not, reasons, statistics
9. Changes to testing strategy
a. How test data are to be obtained
b. Drivers/simulators to be built
c. Special considerations for Ada testing
10. Required resources — hardware required, internal storage requirements, disk space, impact on current computer usage, impacts of compiler
11. Changes to the SDMP (software development/management plan) since PDR
12. Implementation dependencies (Ada projects) — the order in which components should be implemented to optimize unit/package testing
13. Updated software size estimates
14. Milestones and schedules including a well-thought-out build plan
15. Issues, risks, problems, TBD items
a. Review of TBDs from PDR
b. Dates by which TBDs and other issues must be resolved

Figure 15.2

.

Review Checklists: Critical Design Review

What: A checklist to guide a Critical Design Review (CDR). This is held when a prototype is there to group around and constructively criticizes the prototype project. At this point a critical innovation has passed the "chicken test (an early “reality test” of the prototype)." The final CDR reviews the system “chicken test” results. A signoff form is included as part of the template.

Why: To convince everyone that a technical risk has been conquered. Or alternately, that it’s “time to go back to the drawing board …” This is a project decision point for management: Is this project ready to proceed?

How: The Design Review should be organized and lead by the project team leader. Use the checklist to focus on whether the known technical risk areas have been successfully reduced or eliminated, and if there are any new risk areas have been identified.

Definition of "Chicken Test": A reality test on a prototype that verifies the feasibility of the system, the critical parts, and major innovations of the design. Should be performed within 25-50% of the overall schedule. The name derives from tests done on jet engines for collisions of birds. It is done as part of a design “early warning” process to uncover flaws before investing the whole $250 required to build a new engine.

Critical Design Review Checklist Items

1. Has everybody done their preparation work?
2. Where do we stand regarding the innovative and risky parts of the design), and the other project risks on the Critical Issues and Dependencies Lists. How much has been done?
3. Can a demonstration of proof be shown? Review it.
4. What tests have been performed on these parts? What were the results? Did we pass the "chicken tests"?
5. Do we feel confident that the project can proceed as planned? Why?
    a) If not, what do we need to do to get there? When will that be?
    b) Have we run into "show stoppers" that value considering dropping the chosen design approaches, commencing other approaches, or even droppin the whole project?
6. Have new critical issues been identified? Discuss them, and add to the Critical Issues and Dependencies List. What will be done about them?
7. What does the progress to date indicate about how realistic the project schedule is? Do we have further tradeoffs to make, based on the results of our chicken testing?

CDR Follow-up:

1. Publish meeting minutes. Include action items assigned.

2. Reconvene if more research or tests are necessary to convince everyone that the risks have been shown to be acceptable for continuing as planned, and/or to follow up on the assigned action items.

3. If this CDR is for chicken test results of the system, get the appropriate Development executive to sign off on this critical project decision point.

4. Revise the project schedule if required.
5. Achieve the schedule.

Signoff Form For Critical Design Review

The chicken test of the system is a critical executive signoff point for the project. This form can be used to obtain the official "ok to proceed," once a Development executive has witnessed the system chicken test.

Project: _____
Date of System Chicken Test:_____

Brief description of the Chicken test of the system:

_____
_____
_____

Chicken test of the system results brief:

_____
_____
_____

Approvals to go ahead:

Project Leader: _____Date:_____
VP Development: _____Date:_____
Project Champion: _____ Date:_____

Structured Programming

This section reviews the elements of the structured programming theorem. Two graphical techniques for representing algorithms (flowcharts and box diagrams) appear in the figures. Students may need to see examples of representing algorithms using flowcharts and box diagrams, if they are not familiar with them from earlier coursework. Decision tables are discussed as an example of a tabular tool for representing the connections between conditions and actions. Students will benefit from the experience of developing decision tables for their own projects. Program design languages (PDL) are discussed in detail in this section. The tough part in getting students to express their designs using a PDL is getting them to stop trying to write directly in their favorite programming language (C++, Visual Basic, Cobol, etc.). Students frequently fail to distinguish among low level design and implementation. Requiring students to include pseudo code representations of their algorithms as part of the module design documentation seems to help them understand some of the differences between the two.

The foundations of component level design were formed in the early 1960s. Experts proposed the use of a set of constrained logical constructs from which any program could be produced. The constructs emphasized maintenance of functional domain. The constructs are sequence, condition and repetition. Sequence implements processing steps that are essential in the specification of any algorithm. Condition provides the facility for selected processing based on some logical occurrence and repetition allows looping. These three constructs are fundamental to structured programming a main component level design technique.

The structured constructs were proposed to control the procedural design of software to a small number of predictable operations. Complexity metrics indicate that the use of the structure construct reduces program complexity and there by enhances readability, testability and maintainability. The use of a limited number of logical constructs also contributes to a human perceptive process that psychologists call chunking. That means the structured constructs are logical chunks that allow a reader to recognize procedural elements of a module, rather than reading the design or code line. Understanding is improved when readily recognizable logical patterns are encountered. In short, any program, regardless of application area or technical complexity, can be designed and implemented using only the three structured constructs.

Graphical design notation comprises of a flow chart and the box diagram. A flow chart is quite simple pictorially. A box is used to indicate a processing step. A diamond depicts a logical condition and the flow of control shown using arrows.

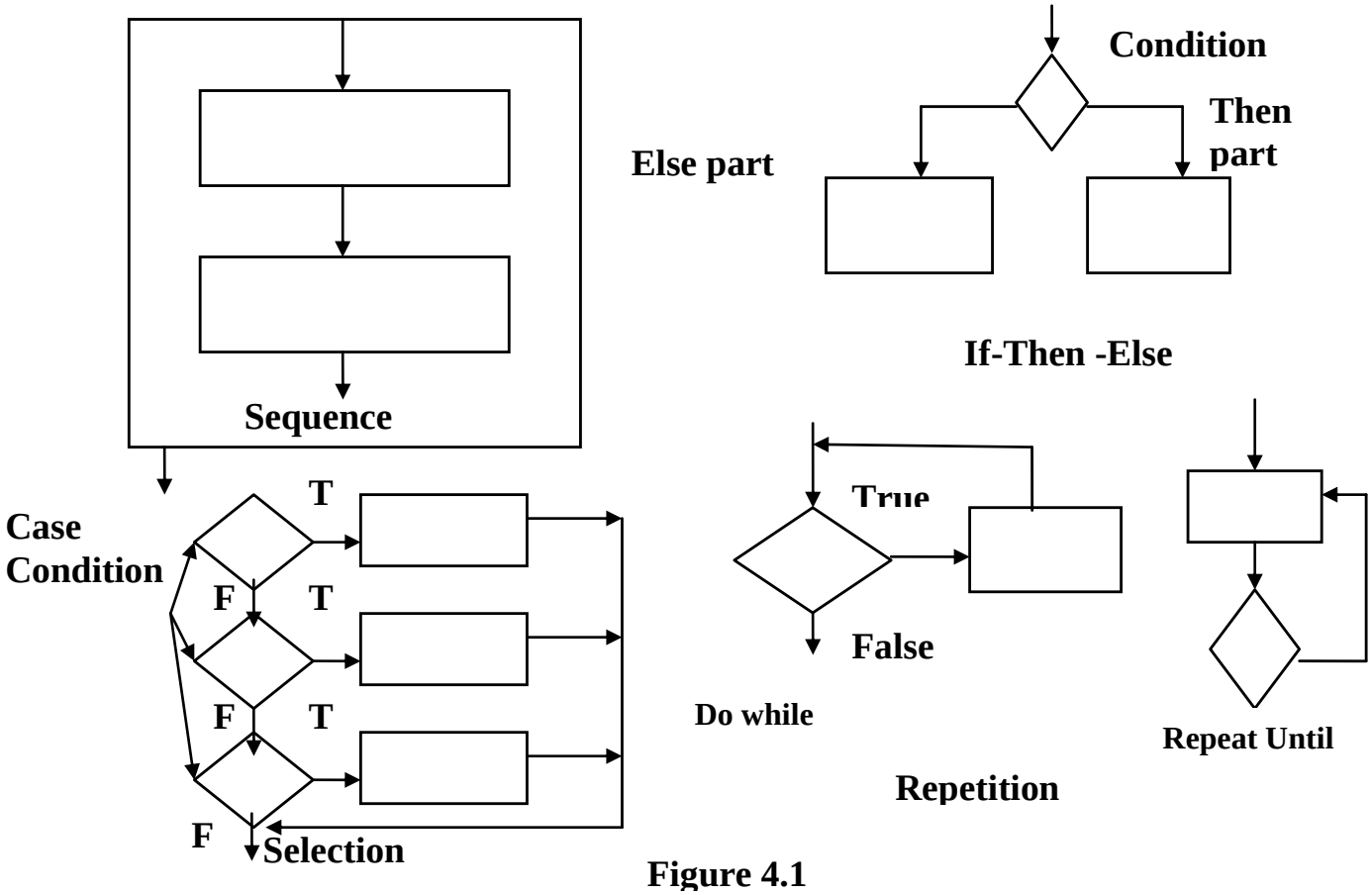


Figure 4.1

Figure 4.1 above shows three constructs. The sequence is represented as two processing boxes connected by line of control. Condition also called if-then-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. 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 the condition and repeats the task until the condition fails. The selection (or select case) construct shown in the figure A 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 box diagram invented from a desire to develop a procedural design represented that would not allow violation of the structured constructs. It has following characteristics a) functional domain (the scope of repetition or if-then-else) is well defined and clearly visible as a pictorial representation b) arbitrary transfer of control is impossible c) the scope of local and global data can be easily determined d) recursion is easy to represent. The graphical representation of structured constructs using the box diagram is shown in Figure 16.3. The sequence is represented using two boxes connected bottom to top. To represent if-then-else a condition box is followed by a then part and else part box. Repetition is shown with bounding pattern that encloses the process to be repeated. Selection is represented using the graphical form shown in Figure 4.2. Finally selection is represented using the graphical form shown at the bottom of the figure 4.2.

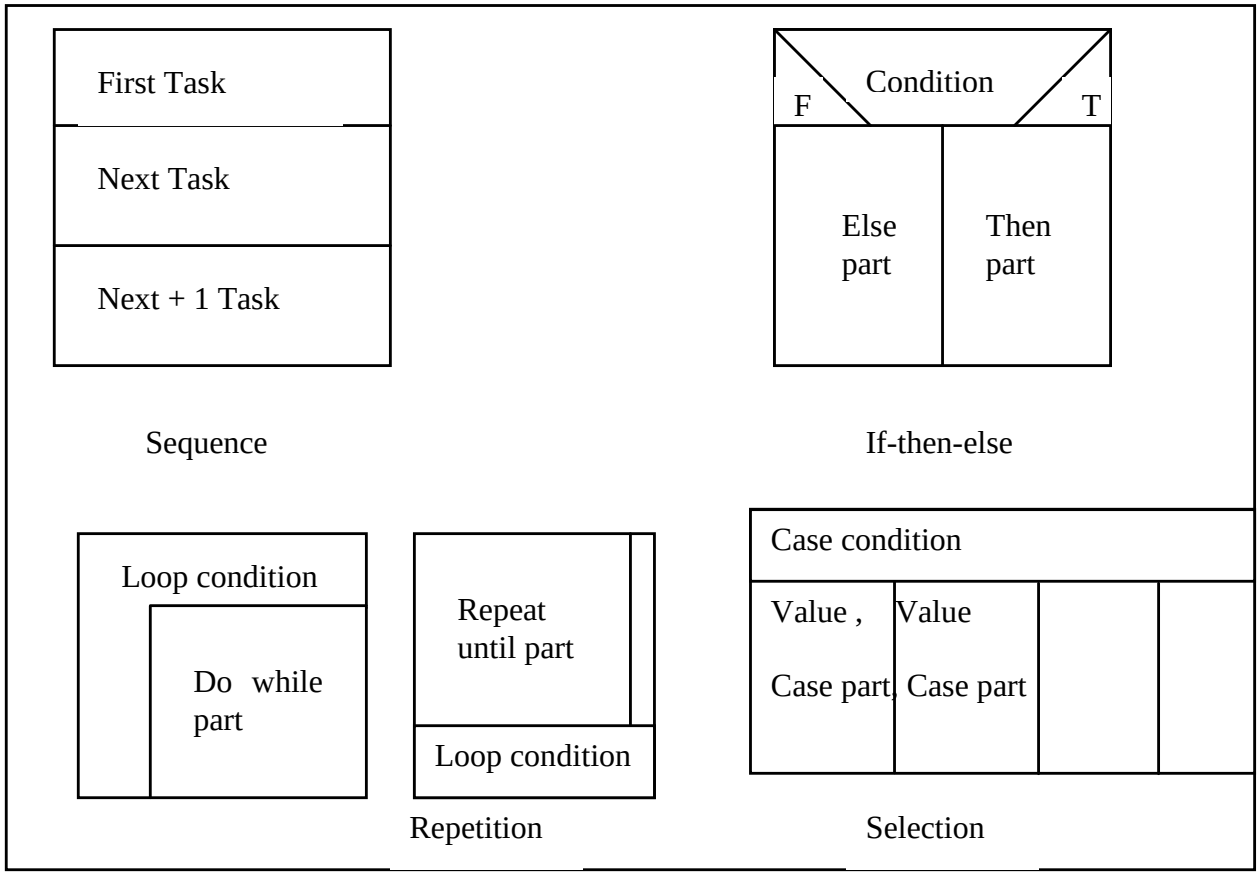


Figure 4.2

Internal Documentation –

Program documentation is of two types, one external which addresses information about the program. The other is internal which has to be close to the program, program statement and program block to explain it there itself. Internal documentation of the program is done through use of comments. All programming language provides means of writing comments in the program.

Commenting – It is a good practice followed by developer to add comments (code commenting) while programming. The comment is a text written for the user, reader or programmer to understand; it is not executed in any manner. A comment generally helps at the time of maintenance. It not only explains the program but also provides points on caution, condition of applicability and assumptions considered important for programmers to know before any action is taken for modification.

Generally in internal documentation, comments should provide information on the following format:-

- Functionality
- Parameters -Input ,Output and there role with usage
- Attributes of inputs for example assumption values, range , min and max. etc.
- Mention global variables
- Last modified
- Author name

Program Verification

Programs are written to translate the system design into executable instructions set using a programming language. This final phase affects the testing and maintenance of the program. The

program should be so constructed that its testing and maintenance will be easy. As far as possible quick fixes and patches of the program is to be avoided. The program is evaluated on the merits of readability, size, execution time, required memory, simplicity and clarity. Before a program is declared complete, it is good practice to put it through verification, not to test the merits of program but to check whether the program produces the desired output. Verification is intended not to confirm that design is right, but that the given design is completely translated into the program produce the desired result. Verification is conducted through two analysis methods one is static where the program/code is executed conceptually (code reading) without any data; in other method the program is executed with data and outputs are checked for quality and errors.

Static analysis – Code reading
In this step the code is read technically to check whether any discrepancies exist between design specification and actual implementation. It is a reverse process where the program component is read and corresponding design specification crosschecked. Program reading ensures that design specifications are completely translated into the program through mapping and tracking of program constructs to design specification.

Code analysis
The purpose of code/program analysis is to detect

- errors currently present
- potential errors
- To identify the scenario where documentation and understanding of program is difficult

Program compilers are used as a tool for program analysis. Mostly data flow analysis is used as a program analysis tool to detect errors.
The types of errors are appended below:-

- Data is specified but not used
- Redundant data is specified
- Data is missing
- Connections are missing which in turn leads to unreachable code segment.
- Variables defined but not used

Code/program analysis reveals errors

- When program constructs are distributed among different teams
- Mismatch errors in parameters used by the teams
- In interfacing two program(modules)
- In termination condition
- Open ends in the flow and in the loop
- In Coding standards not followed properly

Program simulation/ Symbolic execution
In this technique we execute the code using symbolic data, which can take different values. This helps to detect errors in path conditions, flow conditions, loops and soon. Success of this symbolic execution purely depends on the symbolic data used.

Programming Practice

Good programming practices helps to understand, debug, and maintain code. Inadequately written programs lead to confusion while debugging. The developer who has written the code might find himself in a situation where he cannot tell what the code is doing. Therefore, every developer should follow good programming practices.

Form design

The form design should be visually appealing. It should be simple and neat. Although Web pages use bright colors and lots of images to make them attractive, this type of design takes a longer time to load the WebPages. For that reason, while designing pages, you should keep the following guidelines in mind:

The controls which require user input should have the correct tab order and should be grouped and arranged in an order that makes good judgment while entering data. The controls should be properly aligned with respect to each other.

Variables and objects

While naming variables and objects, keep the following guidelines in mind:

- Use a proper naming notation, such as Hungarian or camel-casing notation, to name variables and objects. Hungarian notation enables you to identify the datatype of the variable from the name of the variable. So, a variable storing the first name of an employee will be declared as sFirstName. In camel-casing notation, the variable names take the form sfirstName, with the second part of the variable, which is a noun, capitalized.
- Name the variables and objects meaningfully. Meaningful names combined with Hungarian notation make the purpose and type of the variables clear. This results in a self-documented code, which is easy to understand and maintain.
- Declare the variables and objects in the beginning of a procedure. Declaration in the beginning makes the code execution more efficient, besides making it easy to understand by someone looking at the code text.
- Always initialize variables to definite default values before using them, to avoid any type conversion issues.
- Always rely on explicit conversion functions to eliminate confusion.

Programming logic

While implementing the programming logic, you should do a good chunking of the code. The chunking helps you to maintain the code and speed up debugging. Keep the following guidelines in mind:

- If you want to implement a programming logic that returns a single result, use a function.
- If you need multiple arguments to be passed without expecting a return value, use a procedure.
- If you want to create a reusable piece of code, use functions or Sub procedures or put the code in a separate class (if the code can be logically grouped).

Coding style

The program should be easy to read and to understand when you need to refer back to it. Follow these guidelines while coding:

- Always use "Option Explicit" to catch any undeclared or misspelled variables. Also, the use of "Option Explicit" makes the Web pages run fast. The "Option Explicit" option forces the explicit declaration of variables.
- Declare one variable per line. This avoids confusion about data types.
- Use comments wherever possible to document a difficult code section.
- Use blank lines in the code for clarity.

Programming style in detail

Every coder or programmer has a style evolved over a period. There are dos and don'ts to be observed in the style to make the program simple, readable easy to understand and maintain. However there are proven guiding principles that should be adhered to, to achieve the goal of a good program. The guiding principles are detailed as follows:-

- Naming – in a program you are required to name the module, process, variables and soon. Care should be taken that the naming style should not be cryptic and non representative. That means name should represent the entity completely without confusion. Avoid cryptic names, names totally unrelated to the entity. For example purchase order should be named as PO and not PRO, Vendor_PO and soon.
- Control constructs- control is best exercised when there is a single entity to process and single output to produce.
- Go to statements – this statement should be used sparingly and in a disciplined manner. In fact if other all alternatives fail then go to statement should be used. As far as possible with go to command the program should move forward not shift backward.
- Information hiding- only access functions to data should be made visible and data should be hidden behind these functions.
- User defined types- as far as possible user defined types should be used directly instead of coding them and using. For example type month = {Jan, Feb…Dec} should be used, rather than coding month as (01, 02, 03 …..12). when you use user defined types like month, days, product or scientific entities like debit, credit and so on, the code is automatically very clear to reader and easy to understand.
- Nesting in control constructs "if-then-else" are used extensively to construct a control, based on given conditions. If the condition is satisfied one action is proposed; if not, then another action is proposed. If this condition based nesting is too deep, the code becomes very complex. For example pricing a product for a customer, nesting in this control is as under:-
    - If customer type large then price P1
    - Else medium then price P2
    - Else small then price P3

Instead of this we can exercise the control in the following manner:-

- If customer type large then price P1
- If customer type medium then price P2
- If customer type small then price P3

In both cases control construct will produce the same result.

- Module size – A good module is the one that is cohesive within needs the least interaction outside the module. Modules normally are for functions. If the function generates a couple of sub function and sub-sub functions then one module size will be higher in terms of statements. To bring the module size to a smaller level, it is split into more than one module by sub function.

  For example in ERP system if the invoicing module has sales value calculation, excise value calculation and sales tax calculation to determine invoice construct the module will have more than 50 statements. To overcome this, the module can be split into three modules (sales, excise and sales tax). This will be self contained and easy to maintain. (ERP-Enterprise Resource Planning)
- Program layout – A good layout is the one that helps to read the program faster and to understand it better. The layout should be organized using proper indentation, blank spaces and parenthesis to enhance readability.
- Side effects – A module may have process, value, variable or any other element or entity and in a typical condition it may need modifications. Then such modification affects the module and affects the other modules too. This situation is unavoidable but can be handled by

documenting the effect for the knowledge of programmer so that it is taken care of during the course of modification.

- Crisis management- A well written program may face problems, if it meets with undefined or unacceptable input seeking a process not catered to in the program. This can happen because of incorrect input, incorrect input value, overflow of the data and its value and soon. The crisis should be handled not by crashing the program but by producing a meaningful error message and then systematically exiting the program by closing all earlier states of the program.

Summary

All build test plans are reviewed for correctness and completeness by the management team. When all coding, unit testing, unit and integration testing for the build are complete, selected members of the development team build the system from the source code and execute the tests specified in the build test plan. Both the management team and the development team review the test results to ensure that all discrepancies are identified and corrected. As build testing progresses, the development team begins to put together the user's guide and the system description documents. A draft of the user's guide must be completed by the end of the implementation phase so that it can be reconfirmed during system testing. Material from the detailed design document is updated for inclusion in the system description document, which is finished at the end of the system test phase. Before starting the next build, the development team conducts a build design review *(*BDR*)*. The formality of the BDR depends on the size of the system. Its purpose is to ensure that developers, managers, and customer representatives are aware of any specification modifications and design changes that may have been made since the previous review (CDR or BDR). Current plans for the remaining builds are presented, and any risks related with these builds are discussed. The plans for testing the completed system (or release) are also generated during the implementation phase. Application specialists from the development team prepare the system test plan, which is the basis for end-to-end testing during the next life cycle phase. At the same time, members of the independent acceptance test team prepare the test plan that they will use during the acceptance test phase. The implementation processes are repeated for each build and that a larger segment of the life cycle — extending from the detailed design phase through acceptance testing — is repeated for each release.

Structured programming technique generates codes that are easier to understand and test and maintain. While an insistence on the use of a structured construct is understandable, some unstructured constructs for example break statement are to continue as such statement helps to organize the program. Structured programming largely achieves Understandability, Testability, Maintainability and Readability and binds static behavior and dynamic behavior as one integrated unit. Structured programming is all about creating logical constructs from which a code can be built to represent a procedural design.

Structured programming and layered architecture makes it possible to increase the incidence of reusable components in the system. Efficient structured programming ensures that static structure and dynamic structure have close correspondence.

Self Test

61. The simplest type of construction language is a----------, in which software engineers choose from a limited set of predefined options to create new or custom software installations.
    a) configuration language
    b) program language

62. The purpose of the implementation phase is to build a----------, ----------software system from the "blueprint" provided in the detailed design document.
    a) complete, high-quality
    b) partial, low quality
63. There are --------general kinds of notation used for programming languages, namely:
    a) three
    b) two
    c) Linguistic
    d) Formal
    e) Visual
    f) a) and c),d),e)
    g) b)and c),d
64. Coding area is ----------with --------about the construction of the software components that are identified and described in the design documents
    a) concerned, knowledge
    b) not concerned, knowledge
65. Code Implementation unit is concerned with knowledge about ---------a software design into an ---------programming language.
    a) how to translate, implementation
    b) how to manage, use
66. Code Reuse unit is concerned with knowledge about developing code by ------of existing components and about developing --------code.
    a) Reuse, reusable
    b) Rewriting, reusable
67. Checking for adherence to ----------is certainly the most ----------of all features.
    a) coding standards, well-known
    b) coding style, not much known
68. A coding standard consists of
    a) a set of programming rules
    b) naming conventions
    c) layout specifications
    d) All of the above
    e) None of the above
69. In Coding activities of the Development Team are
    a) Code new units
    b) Read new and revised units
    c) Test each unit and module
    d) Plan and conduct build tests
    e) All of the above
    f) None of the above
    g) Only a)
    h) Only b)
70. Good programming practices make it easy to ----------, , and.
    a) understand
    b) debug
    c) maintain code
    d) All of the above
    e) None of the above
71. A good programming practices comprises of
    a) Form design
    b) variables and objects

c) programming logic
d) Coding style
e) All of the above
f) None of the above

72. While designing pages, you should keep the following guidelines in mind:
    a) The controls that need user input should have the correct tab order and should be grouped
    b) Arranged in an order that makes sense while entering data.
    c) The controls should be properly aligned.
    d) All of the above
    e) None of the above
73. If you want to implement a programming logic that returns a single result----------
    a) use a function
    b) use a procedure.
    c) use reusable code
74. The guiding principles for Programming style includes
    a) Naming
    b) Control construct
    c) Go to statements
    d) Information hiding
    e) User defined types
    f) All of the above
    g) None of the above
75. In structured programming,--------- graphical techniques for representing algorithms, ----------appear in the figures
    a) two
    b) flowcharts and box diagrams
    c) three
    d) charts and diagrams
76. In structured programming, the constructs are
    a) sequence,
    b) condition
    c) Repetition.
    d) All of the above
    e) None of the above
77. Program documentation is of two types, one external which addresses ---------
    a) Information about the program.
    b) Information about the process
78. Generally in internal documentation, comments should provide information on the following format:-
    a) Functionality
    b) Parameters -Input ,Output and there role with usage
    c) Attributes of inputs for example assumption values, range , min and max. etc.
    d) Mention global variables
    e) Last modified
    f) Author name
    g) All of the above
    h) None of the above
79. Verification is intended not to confirm that----------, but that the given design is completely translated into the program---------.
    a) design is right

b) produce the desired result
c) program is right
d) produce the result

80. The purpose of code/program analysis is to detect
    a) errors currently present
    b) potential errors
    c) To identify the situations where documentation and understanding of program is difficult
    d) All of the above
    e) None of the above

81. Classification of Algorithm design includes :-
    a) Divide and conquer
    b) Dynamic programming
    c) Only a)
    d) Only b)
    e) a), b)
82. A divide and conquer algorithm repeatedly ------an instance of a problem to -----or smaller instances of the same problem.
    a) reduces, one
    b) expands, two

Lesson 5

Software Testing and Maintenance

Overview

In this chapter, we are going to learn the fundamentals of testing: why testing is needed; its limitations, objectives and purpose; the principles behind testing; the process that testers follow; and some of the emotional factors that testers must consider in their work. By reading this chapter you' will understand the fundamentals of testing .

Learning Objectives

- To understand Testing Fundamentals
- To understand Test Oracles
- To understand Functional and Structural testing
- To understand Test case writing
- To understand Error/ Defects
- To understand Software Maintenance

Introduction –

Software Testing has achieved a importance in the System Development Life Cycle. A lot of people have worked on the subject and provided various techniques and methodologies for effective testing. Nowadays, we have many books and websites on Software Test Engineering, many people are misleading in understanding the basic concepts of the software testing.

Software Development has many phases. These phases include Requirements Engineering, Architecting, Design, Implementation, Testing, Software Deployment, and Maintenance. Maintenance is the last stage of the software life cycle. After the product has been released, the maintenance phase keeps the software up to date with environment changes and changing user requirements

Fundamentals of Testing

What is Software Testing?

.
The Software Crisis started in the 1960's when the basic cause for these circumstances was not following software engineering practices. There was a lot of interest in computers, a lot of code generated but no established standards. Then in early 70's a lot of computer programs started failing and people lost confidence and thus an industry crisis was declared. Various reasons leading to the crisis included:

- Hardware advancement unable to develop software for this kind of hardware.
- Software dependency started piling up

- Struggle to build reliable and high quality software
- Poor design and insufficient resources.

This crisis though identified in the early years, exists to date and we have examples of software failures around the world. Software is basically considered a failure if the project is terminated because of costs or overrun schedules, if the project has experienced overruns in excess of 50% of the original or if the software results in client lawsuits. Some examples of failures include failure of Aviation traffic control systems, failure of medical software, and failure in telecommunication software. The primary reason for these failures is due to bad software engineering practices followed. Some of the worst software practices include:

- No historical software-measurement data.
- Rejection of accurate cost estimates.
- Failure to use automated estimating and planning tools.
- Excessive, irrational schedule pressure and creep in user requirements.
- Failure to monitor progress and to perform risk management.
- Failure to use design reviews and code inspections.

To avoid these failures and thus improve the record, what is needed is a better understanding of the process, better estimation techniques for cost time and quality measures. But the question is, what is a process? Process transform inputs to outputs i.e. a product. A software process is a set of activities, practices and methods involving conversion that people use to develop and maintain software.
At present a large number of problems exist due to a chaotic software process and the occasional success depends on individual efforts. Therefore to be able to deliver successful software projects, a focus on the process is essential since a focus on the product alone is likely to miss the scalability issues, and improvements in the existing system. This focus would help in the predictability of outcomes, project trends, and project characteristics.
The process that has been defined and adopted needs to be managed well and thus process management comes into play. Process management is concerned with the knowledge and management of the software process, its technical aspects and also ensures that the processes are being followed as expected and improvements are shown.

From this we conclude that a set of distinct processes can possibly save us from software project failures. But it is nonetheless important to note that the process alone cannot help us avoid all the troubles/problems, because with varying circumstances the need varies and the process has to be adaptive to these changeable needs. Prime importance required to be given to the human aspect of software development which only can have a lot of impact on the results, and effective cost and time estimations may go totally waste if the human resources are not planned and managed effectively. Secondly, the reasons mentioned related to the software engineering principles may be resolved when the needs are properly recognized. Proper identification would then make it easier to identify the top practices that can be applied as one process that might be suitable for one organization may not be most suitable for another.
Therefore to make a successful product a combination of Process and Technicalities will be required under the umbrella of a well-defined process.

Having discussed about overall the Software process, it is important to identify and relate the responsibility software testing plays not only in producing quality software but also stage management the overall process.

Definition of testing is as follows: “Testing -- A verification method that applies a controlled set of conditions and arousing interest for the purpose of finding errors. This is the most desirable method

of verifying the functional and performance requirements. Test results are accepted proof that requirements were met and can be repeated. The resulting data is reviewed by all concerned for confirmation of capabilities."
There are many definitions of software testing, but it is good to start by defining testing and then progress on depending on the needs or requirements.

When Testing should occur?

Wrong Assumption

Testing is at times wrongly thought as an after the fact activity; performed after programming is over/finished for a product. But testing should be performed at every stage of the product development. Test data sets must be derived and their correctness and consistency should be monitored throughout the development process. If we divide the lifecycle of software development into "Requirements Analysis", "Design", "Programming/Construction" and "Operation and Maintenance", then testing should accompany each of the above phases. If testing is isolated as a single phase late in the cycle, errors in the problem statement or design may invite excessive costs. Not only must the original error be corrected, but the entire structure built upon it must also be changed. So testing should not be isolated as an inspection activity only. The testing should be carried out throughout the SDLC in order to get a good quality product.

Testing Activities in Each Phase

The under mentioned testing activities should be carried out during the phases

- Requirements Analysis - (1) Determine correctness (2) Create functional test data.
- Design - (1) Determine correctness and consistency (2) Create structural and functional test data.
- Programming/Construction - (1) Determine correctness and consistency (2) Create structural and functional test data (3) Apply test data (4) Refine test data.
- Operation and Maintenance - (1) Retest.

Requirements Analysis Phase

The following test activities should be performed during this stage.

- Devote in analysis at the beginning of the project - Having a clear, brief and proper statement of the requirements helps programming, communication, error analysis and test data generation.

The requirements statement should document the following decisions and information:

1. Program function - What the program must do? (expectation from program)
2. Input of the form, data types, format and units for.
3. Output of the form, data types, format and units .
4. How exceptions, errors and deviations are to be handled ?
5. For scientific computations, the numerical method or at least the required accuracy of the solution.
6. The hardware/software environment required or assumed (e.g. the machine, the operating system, and the implementation language).

Focusing the above issues is one of the activities connected to testing that should be followed during this stage.

- At the requirements analysis phase start developing the test set - Data should be generated that can be used to determine whether the requirements have been met. To do this, the input domain should be divided into classes of values that the program

will treat in a similar manner and for each class a representative element should be included in the test data. Also , following should also be included in the data set: (1) boundary values (2) any non-extreme input values that would require special handling.

- The output domain should be treated similarly.
- Invalid input requires the same analysis as valid input.
- The correctness, consistency and completeness of the requirements should also be analyzed - Consider whether the correct problem is being solved, check for conflicts and inconsistencies among the requirements and consider the possibility of missing cases.

Design Phase

The design document helps in programming, communication, and error analysis and test data generation. The the design document and requirements statement should collectively give the problem and the organization of the solution i.e. what the program will do and how it will be done.

The design document should contain:

- Principal data structures.
- Functions, algorithms, heuristics or special techniques used for processing.
- The program organization, how it will be modularized and categorized into external and internal interfaces.
- Any additional information.

Here the testing activities should consist of:

- Design Analysis to make sure its completeness and consistency - the whole process should be analyzed to decide that no steps or special cases have been ignored. Internal interfaces, data structures and I/O handling should specially be inspected for inconsistencies.
- Design Analysis to make sure that the requirements are satisfied - check that requirements and design docs contain the matching format, form, units used for input and output and all functions listed in the requirement document have been included in the design docs/manuscript. Selected test data which is generated during the requirements analysis phase should be manually simulated to determine whether the design will yield the expected values.
- Creation of test data based on the design - The tests generated should cover the structure as well as the internal functions of the design like the data structures, functions, algorithm, heuristics and common program structure etc. Standard extreme and special values should be included and expected output should be recorded in the test data.
- Reassessment and enhancement of the test data set generated at the requirements analysis phase.

The first two steps should also be performed by some peer/colleague as well as by the designer/developer.

Programming/Construction Phase

Here the main testing points are:

- Check the code for consistency with design - the areas to check include modular structure, module interfaces, data structures, functions, algorithms and I/O handling.
- Perform the Testing process in an planned and systematic manner with test runs dated, commented and saved. A plan or schedule can be used as a checklist to help the

programmer manage testing efforts. If errors are found and changes made to the program, all tests involving the erroneous part must be rerun and recorded.

- Asks some peer/colleague for help - Some independent party, other than the programmer of the specific part of the code, should analyze the development product at each phase. The programmer should explain the product to the customer/party who will then query the logic and search for errors with a checklist to guide the search. This is needed to locate errors the programmer has overlooked.
- Use existing tools - the programmer should be familiar with various compilers and interpreters available on the system for the execution language being used because they vary in their error analysis and code generation capabilities.

- Put Stress to the Program - Testing should work out and stress the program structure, the data structures, the internal functions and the externally visible functions or functionality. Both valid and invalid data should be included in the test set.

- Test one at a time - Pieces of code, individual modules and small collections of modules should be exercised separately before they are integrated into the total program, one by one. Errors are easier to isolate when the no. of potential interactions should be kept small. Instrumentation-insertion of some code into the program solely to measure various program characteristics – can be useful here. A tester should perform array bound checks, check loop control variables, determine whether key data values are within acceptable ranges, trace program execution, and count the no. of times a group of statements is executed.

Operations and Maintenance Phase

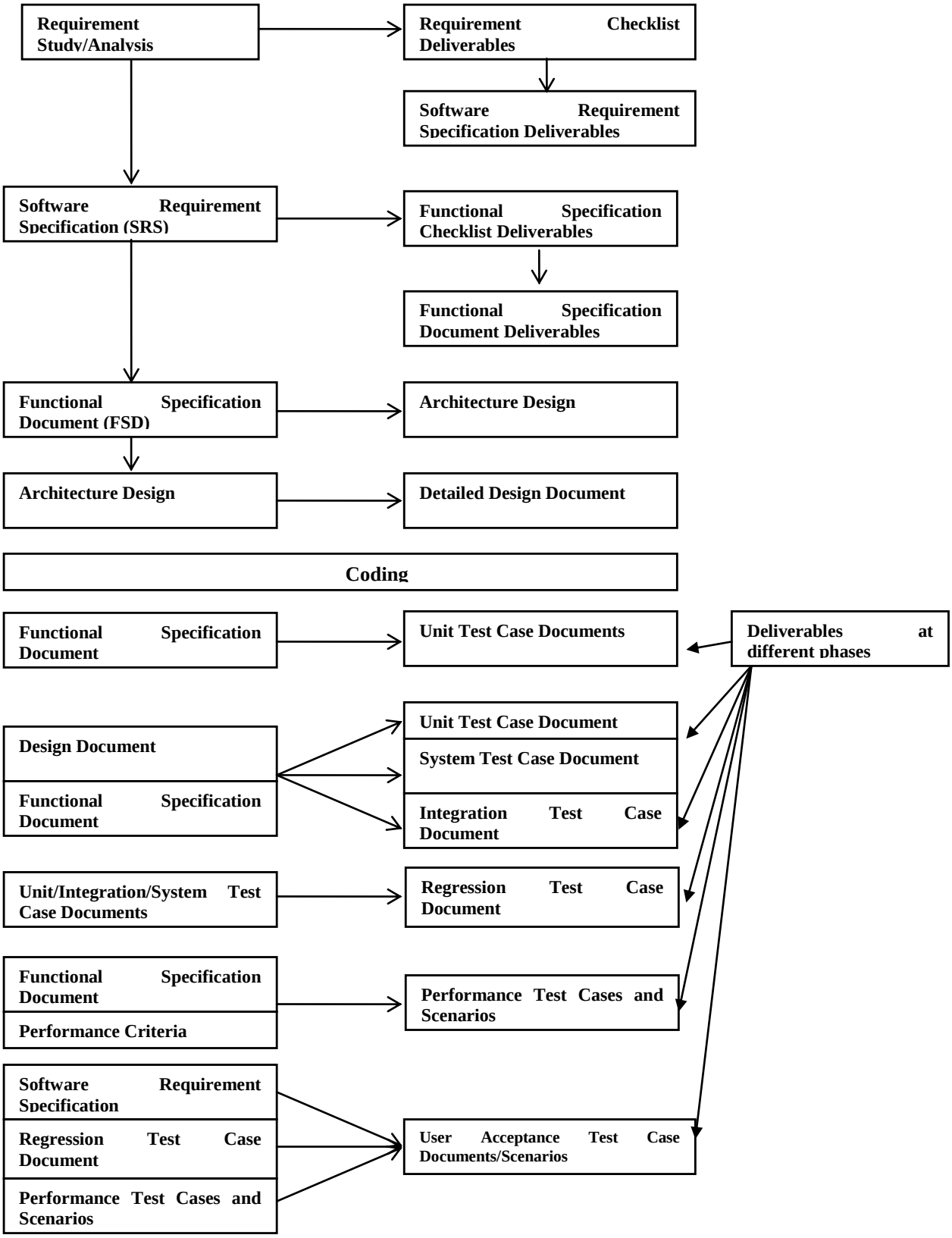
Corrections, modifications and extensions are bound to happen even for small programs and testing is required every time there is a change/modification. Testing during maintenance is termed regression testing. The test set, the test plan, and the test results for the unique program should exist. Modifications must be made to accommodate the program changes, and then all portions of the program affected by the modifications must be re-tested, that means tested again. After regression testing is complete, the program and test documentation must be updated with required changes.

Test Development Life Cycle (TDLC)

Usually, Testing is considered as a part of the System Development Life Cycle. With our practical experience, we framed this Test Development Life Cycle.
The diagram does not represent when and where you write your Test Plan and Strategy documents. But, it is understood that before you begin your testing activities these documents should be ready. Ideally, when the Project Plan and Project Strategy are being made, this is the time when the Test Plan and Test Strategy documents are also made.

Test Development Life Cycle (TDLC)

Diagram shows deliverables get ready and delivered at different stages. This diagram will help you to check deliverable/receivable at different stages..


Requirement Study/Analysis
Requirement Checklist Deliverables
Software Requirement Specification Deliverables
Software Requirement Specification (SRS)
Functional Specification Checklist Deliverables
Functional Specification Document Deliverables
Functional Specification Document (FSD)
Architecture Design
Architecture Design
Detailed Design Document
Coding
Functional Specification Document
Unit Test Case Documents
Deliverables at different phases
Design Document
Functional Specification Document
Unit Test Case Document
System Test Case Document
Integration Test Case Document
Unit/Integration/System Test Case Documents
Regression Test Case Document
Functional Specification Document
Performance Criteria
Performance Test Cases and Scenarios
Software Requirement Specification
Regression Test Case Document
Performance Test Cases and Scenarios
User Acceptance Test Case Documents/Scenarios

Testing Types and Techniques

Testing types

Testing types refer to different approaches towards testing a computer program, system or product. The testing has two types – one is white box testing and another is black box testing. We are going to discussed in depth in this lesson. One more type, termed as gray box testing or hybrid testing is budding presently and it combines the features of the black box and white box testing.

Testing Techniques

Testing techniques refer to different methods of testing particular features a computer program, system or product. Each testing type has its hold testing techniques while some techniques join the feature of both types.
A black box testing techniques are

- Error Guessing
- Boundary Value analysis
- Equivalence partitioning

A white box testing techniques are

- Control flow based testing
- Data Flow Testing

Difference between Testing Types and Testing Techniques

Testing types works with what feature of the computer software would be tested, while testing techniques works with how a specific part of the software would be tested.

That is, testing types signify whether we are testing the structure or the function of the software. That means, we should test each functionality of the software to see if it is functioning or we should test the internal parts/components of the software to check if its internal functioning are according to (SRS) software requirement specification,.

The ‘Testing technique’ means what methods/technique would be applied or calculations would be done to test a particular feature of software (viz. we test loops ,the interfaces, the segments)

How to decide a White Box Test or Black Box ?

The White box testing is related only with testing the software product; it is not sure that the complete specification has been implemented, that means meeting requirement cannot be assumed. Black box testing is related only with testing the specification; it is not sure that all components/parts of the implementation have been tested in that regard. Thus black box testing is testing against the specification and will discover errors of omission, indicating that part of the specification has not been fulfilled. White box testing is testing against the implementation and will discover errors of commission, indicating that part of the implementation is faulty. To test a software product completely both black and white box testing are required.

White box testing is much more expensive (In terms of time and resources) than black box testing. It requires the source code to be produced before the tests can be planned and is much more laborious in the determination of suitable input data and the determination if the software is or is not correct. It is advised to start test planning with a black box testing approach as soon as the specification is available. White box tests are to be planned as soon as the Low Level Design (LLD) is complete. The Low Level Design will deal with all the algorithms and coding style. The paths should then be checked against the black box test plan and any supplementary required test cases should be determined and applied.

The consequences of test failure at initiative/requirements stage are very expensive. A failure of a test case may result in a change, that triggers all black box testing to be repeated and the re-determination of the white box paths. The cheaper option is to regard the process of testing as one of quality assurance. The aim is that sufficient quality is put into all previous design and production stages so that it can be expected that testing will project the presence of very few faults, rather than testing being relied upon to discover any faults in the software, as in case of quality control. A combination of black box and white box test considerations is still not a completely adequate test rationale (basis)

White Box Testing

What is WBT?

White box testing involves looking at the structure of the code. When you know the internal structure of a product, tests can be conducted to ensure that the internal operations performed according to the specification. And all internal components have been adequately exercised. In other word WBT tends to involve the coverage of the specification in the code.

Code coverage is defined in two types as listed below.

- Control flow based testing
  - Segment coverage – Each segment of code b/w control structure is executed at least once.
  - Branch Coverage or Node Testing – Each branch in the code is taken in each possible direction at least once.
  - Compound Condition Coverage – When there are multiple conditions, you must test not only each direction but also each possible combinations of conditions, which is usually done by using a 'Truth Table'
  - Basis Path Testing – Each independent path throughout the code is taken in a preset order. This point will further be discussed in other section.

- Data Flow Testing (DFT) – In this approach you follow the particular variables through each possible calculation, thus defining the set of intermediate paths through the code i.e., those based on each piece of code chosen to be followed. Although the paths are said independent, dependencies across multiple paths are not really tested for by Data Flow Testing. DFT tends to reflect dependencies but it is mainly through sequences of data manipulation. This approach tends to uncover bugs like variables used but not initialize, or declared but not used, and so on.

What we do in White Box Testing?

In WBT, we use the control structure of the procedural design to derive test cases. Using WBT methods a tester can derive the test cases that

- Guarantee that all independent paths within a module have been exercised at least once.
- Exercise all logical decisions on their true and false values.
- Execute all loops at their boundaries and within their operational bounds
- Exercise internal data structures to ensure their validity.

the technicalities of Static Software Testing into one place and arriving at a common understanding. Verification helps in identifying not only presence of defects but also their location.

Unit Testing focuses an individual portion of code or component, usually a function or a subroutine or a screen, etc. it is normally the last step in writing a piece of code and treating unit as a whole system.

Verification Strategies

What is 'Verification'?

Verification is the process of evaluating a system or component to determine whether the products of a given development phase meets the conditions forced at the start of that phase.
What is the importance of the Verification Phase?
Verification helps in detecting defects early, and preventing their migrating un knowingly in the next phase. As a result, the high cost of later detection and rework is eliminated.

Review

A process or meeting during which a work product, or set of work products, is presented to project stake holders viz. managers, users, customers, and other interested parties for comment or approval.
The main aim of reviews is to find defects. Reviews are a good compliment to testing to help assure quality. A few purposes' of SQA reviews can be as appended below:

- Assure the quality of deliverable's before the project moves to the next stage.
- Once a deliverable has been reviewed, revised as required, and approved, it can be used as a basis for the next stage in the life cycle.

The types of reviews

Types of reviews are Management Reviews, Technical Reviews, Inspections, Walkthroughs and Audits.

Management Reviews

Management reviews are performed by those directly responsible for the system in order to monitor progress, determine status of plans and schedules, confirm requirements and their system allocation.

Therefore the main objectives of Management Reviews can be categorized as follows:

- Validate from a management perspective that the project is making progress according to the project plan.
- Ensure that deliverables are ready for management approvals.

- Resolve issues that require management's attention.
- Identify any project bottlenecks.
- Keeping project in Control.

Support decisions made during such reviews include Corrective actions, Changes in the allocation of resources or changes to the scope of the project
In management reviews the following Software products are reviewed:

Audit Reports

Contingency plans

Installation plans

Risk management plans

Software Q/A

The participants of the review play the roles of Decision-Maker, Review Leader, Recorder, Management Staff, and Technical Staff.

Technical Reviews

Technical reviews confirm that product Conforms to specifications, adheres to standards, regulations, guidelines, plans, changes are properly implemented, changes affect only those system areas identified by the change specification.

The main objectives of Technical Reviews can be separated as follows:-

- Make sure that the software confirms to the organization standards.
- Make sure that any changes in the development events (design, coding, testing) are implemented per the organization pre-defined standards.

In technical reviews, Software work products are reviewed as appended below:-

- Software requirements specification
- Software design description
- Software test documentation
- Software user documentation
- Installation procedure
- Release notes

The participants of the review play the roles of Decision-maker, Review leader, Recorder, Technical staff.

Requirement Review

A process or meeting during which the requirements for a system, hardware item, or software item are presented to project stake holders viz. managers, users, customers, or other interested

parties for comment or approval. Types include system requirements review, software requirements review.
In this review activity members from respected department take part and complete the review.

Input /Entry Criteria

Software requirement specification is the essential document for the review. A for review Requirement Review checklist can be used.

Output /Exit Criteria

Exit criteria include the filled & completed Requirement Review checklist with the reviewers' comments & suggestions and the re-verification whether they are incorporated in the documents.

Design Review

A process or meeting during which a system, hardware, or software design is presented to project stakeholder viz. managers, users, customers, or other interested parties for comment or approval. Types include critical design review, preliminary design review, and system design review.

Who participate in Design Review?

QA team member leads design review. Members from development team and QA team participate in the review.

Input / Entry Criteria

Design document is the essential document for the review. For the review a design review checklist can be used.

Output/Exit Criteria

Exit criteria include the filled & completed design review checklist with the reviewers' comments & suggestions and the re-verification whether they are incorporated in the documents.

What is Code Review?

A meeting at which software code is presented to project stakeholder's viz. managers, users, customers, or other interested parties for comment or approval.

Who is participate in Code Review?

QA team associate (In case the QA Team is only involved in Black Box Testing, then the Development team lead chairs the review team) controls code review. Members from development team and QA team participate in the review.

Input/Entry Criteria

The Coding Standards Document and the Source file are the essential documents for the review. For the review a checklist can be used.

Output/Exit Criteria

Exit criteria include the duly filled checklist with the reviewers' comments & suggestions and the re-verification whether they are incorporated in the documents.

Walkthrough

A static analysis technique in which a designer or programmer leads members of the development team and other interested parties through a segment of documentation or code, and the participants ask questions and make remarks about possible errors, violation of development/coding standards, and other problems.

In nutshell the objectives of Walkthrough are as follows:

- Detect errors early.
- Ensure (re)established standards are followed:
- Train and trade/exchange technical information flow among project teams which participate in the walkthrough.
- Increase the quality of the project, thereby improving morale of the team members.

The participants in Walkthroughs assume one or more of the following roles:

a) Walk-through leader

b) Recorder

c) Author

d) Team member

To think a review as a systematic walk-through, a team of at least two members shall be assembled. Roles may be shared among the team members. The walk-through leader or the author may serve as the recorder. The walk-through leader may be the author. Individuals holding management positions over any member of the walk-through team member should not participate in the walk-through.

Input to the walk-through shall include the following:

a) A statement of objectives for the walk-through

b) The software product under test/inspection

c) Standards that are in effect for the supply, acquisition, development, operation, and/or maintenance of the software product

Input to the walk-through may also comprise the following:

d) Any standards, regulations, guidelines, plans, and procedures against which the software product is to be inspected

e) Anomaly categories

The walk-through shall be considered complete when

a) The entire software product has been examined

b) Recommendations and required actions have been recorded

c) The walk-through output has been completed

Inspection

A thorough, word-by-word checking of a software artifacts (documents) with the intention of:

- Locating defects
- Confirming traceability of relevant requirements
- Checking for conformance to relevant standards and conventions

.

The participants in Inspections assume one or more of the following roles:

a) Inspection leader

b) Recorder

c) Reader

d) Author

e) Inspector

All participants in the review are inspectors. The author shall not act as inspection leader and should not act as reader or recorder. Other roles may be shared among the team participants. Individual participants may act in more than one role.
Individuals holding management positions over any member of the inspection team should not take part in the inspection activity.

Guidelines for Inspections*:*

- Plan the inspection.
- Allocate trained resources.
- Moderator / inspection leader circulates the product to be inspected and checklist to inspection team in advance.
- Product is clear-compiled, checked for spellings, grammar, formatting, etc.
- Inspection team consists of: moderator, author, domain expert, users (optional).
- Managers are not involved in inspections.
- Author reads the product, line by line.
- Concentrates on locating defects; avoids discussions on possible solutions.
- Classifies and records the defects.
- Products that are inspected: SRS, Design documents, Source Code files, User manual, Installation Manual, Help Files, Various plans, etc.

Participants:

- An Inspection Team vested with the responsibility.
- Concerned requirements: developers/designer(s)/ programmers(s).

Headed by:

- A senior member of the Inspection Team.

Advance Preparation:

- Relevant documentation circulated among all well in advance.
- Each Inspection Team member allocated specific part of the work to be inspected.
- Each member required to come prepared.

Presentations by:

- Presentation by each Inspection Team member, and not by developers/designers.

Target:

- Requirements, Design, Code, Documentation.

Inputs:

- Documents /Design/ Code documents.
- Checklists: generic / specific, Standards & Guidelines.

Output:

- Evaluation. Opinion. Suggestions. Error-List. Plan of action.

Validation Phase

The Validation Phase falls into picture after the software is ready or when the code is being written. There are various techniques and testing types that can be appropriately used while performing the testing activities. Let us examine a few of them.

Unit Testing

- Unit testing is the most 'micro' scale of testing.

- Unit Testing exercises an individual portion of code or component, usually a function or a subroutine or a screen, etc. it is normally the last step in writing a piece of code; first step – if one follows the more sensible approach of "extreme programming".

- To enable recursive and thorough testing, it is desirable to develop executable test programs for Unit Testing, prior to (preferably) or in parallel with development of the component.

- It is typically done by the programmer and not by testers, as it requires detailed knowledge of the internal program design and code.

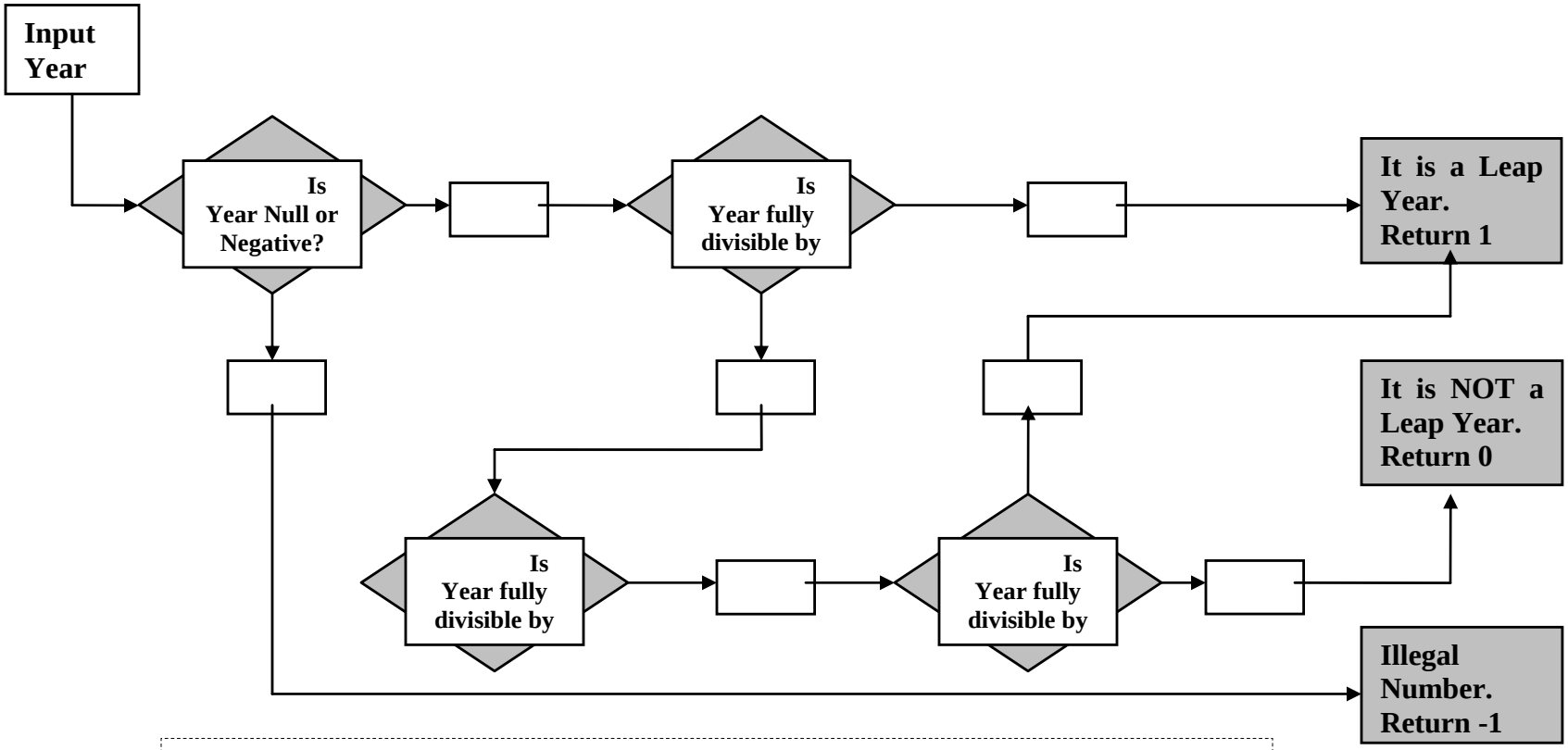
- The tests can be behavioral (black-box) or structural (white-box).

A Simple example for sake of illustrating the concept:

- Let us consider a function to test whether a year is a leap year.
- **Rather than choosing some complex, business domain specific function, requiring explanation on the business domain too, the leap-year function is chosen on account of its general familiarity**.
- Of course, it can be argued if one needs to do all this for a function which, once coded and tested, would not be subject to change.

- **The answer, as mentioned in the caption, is: it is only to illustrate a concept. Basically, the approach would be similar in many cases.**
- Like in “Extreme Programming”, even before we code, let us see how it would be tested.
- Essentially, the function would be called with “year” as an argument; and it should return a value indicating if the year is a leap year or not.

How to do black box testing- where internals/ code are not known? We must have sets of numbers/ years (1) that is leap years, (2) that are not leap years, (3) Those are illegal numbers.



Fully Divisible=Not Leaving a remainder. E.g. 2004/4, 1204/4, 1600/400, 2000/100

Component Testing

- Testing of a complete component, a unit, a sub-module that goes to make a module.
- It can be a generic component, usable across modules; or, it can be a component of a specific module.
- It can comprise one or more objects or classes or a set of functions and procedures.
- It can be tested through the various functional testing methods.

Examples:

A set/library of financial functions to compute interest, maturity value, installments, etc	**A set/library of date and time functions for various computations, and to return date/time in various formats**	**Error -handling. Routines to handle, display, communicate (email, SMS, fax…), Log Errors and take appropriate action.**	**Common sub-module or component to handle relevant aspects of TDS (tax deducted@ source) for various modules**

- Component testing follows Unit Testing. After each of the sub-components / methods / functions of the component are Unit Tested, component as a whole is tested.

- Components are tested for the services they provide as a unit.

- Unlike Unit testing, which is normally White-box testing done by the developers; Component testing would normally be Black-box or Grey-Box testing (although it could be while-box too).

- Services of well-made components would be accessible only through their standard, declared interfaces – making it that much more convenient to test and certify them. On the other hand, a badly designed component would have "n" number of entry points, making it much more cumbersome to both maintain and test – it could almost be like an aggregation of sub-components.

Module Testing

- "n" units = 1 component;
  "n" components = 1 module;
  "n" modules = an application.

- Testing of a complete module. It could be a generic Framework Module, or a specific Business Module.

  Examples:

Application	Business	Business
Security & Access	Transaction Engine	Term/Time
Error Handling	Audit Trails	Loans

- Depending on the nature of application, Module Testing may require testing over a period of time.

- For Example, in case of the banking modules above, the following are the requirements for reasonable test coverage:
  - Create a reasonable number of account masters.
  - Enter data in all necessary parameter and code tables.
  - Test transaction for a period of 15 to 18 months, to enable testing of all periodic processes for interest, service charges, renewals/roll-over, etc. – like begin and end of week or month and so on
  - You need not test for each day of the 15/18 months, to enable testing of all periodic processes for interest, service charges, renewals/ roll-overs, etc. – like begin and end of week or month and so on.
  - You need not test for each day of the 15/18 months period. You need to select 3 to 5 days each month, amounting to a total of say 60 days for testing of 15 months, @ an average of 4 days per month. Do begin-day, enter transaction for the first day, do end-day, roll forward to the next day, repeating begin/end-day, thus covering 60 days in the 15 months span.
  - Compare actual with the expected results for each day.

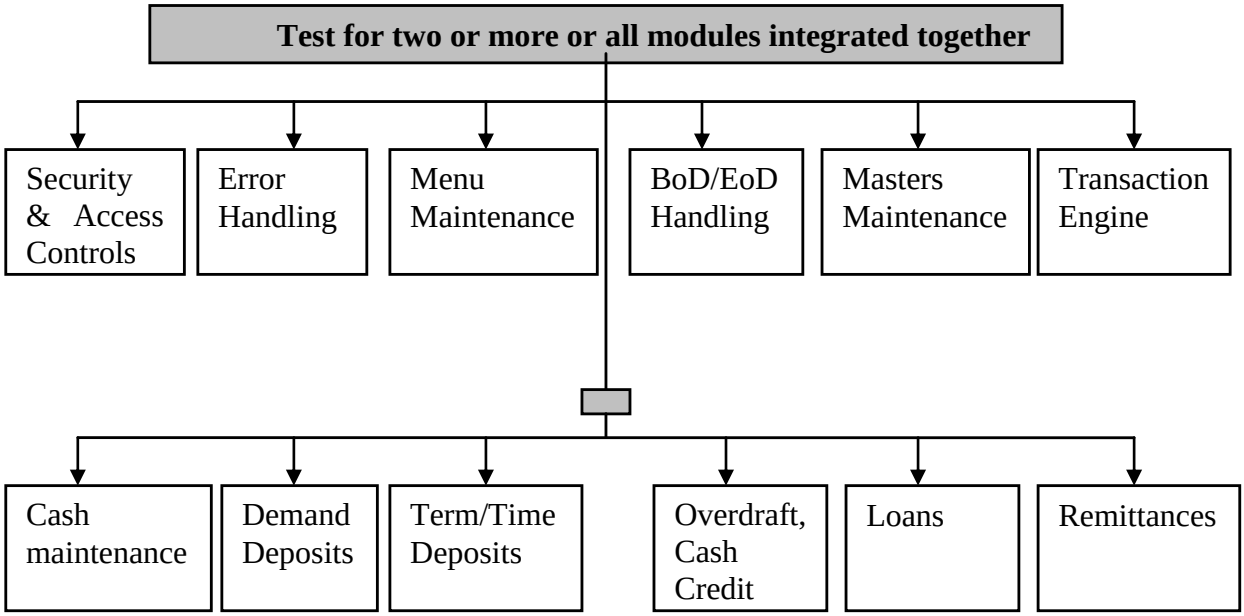
  However, in say Cash Module, such elaborate testing may not be required.

Integration Testing

- Put pieces together and test. Testing of combined parts (integrated parts) of an application to determine if they function together correctly. The 'parts' can be code modules, sub-modules, individual application, client-server application, web-application, etc. Individual components are first tested; they are then integrated with the application and re-tested. The intent is to expose faults in the interactions and interfaces between software modules and functions. It uses functional testing methodologies.

- **Incremental Integration Testing**: Testing of an application as incrementally modules, sub-modules, parts, components get added to it.
- **Big-bang Testing**: Opposite of incremental testing – integrate all and test.

Example



End-to-End testing / System Testing

- This is the most 'macro' end of the test scale.
- It involves testing of a complete application in a situation that mimics real-world use.
- Is it then functional testing, or, rather detailed or intensive functional testing? – No.
- Basically, the purpose of System Testing is to find the deficiency of the System in meeting its overall goals. And, overall goals are not just functional.
- System Testing must therefore check that both the functional and the non-functional goals are met. That is, apart from the fact that the System handles the business correctly, it must also be able to do so, say, with expected load in terms of number of users, transactions, size of data, etc. In other words, it should not be that while the System functions satisfactorily under the conditions tested, it begins to behave abnormally and malfunctions when the number of users grow to 1000 users, which is the expected load; or, when the database grows beyond 5 GB, although the expected size is 20 GB; or, when there are 300 transactions per second, although the expected number is 500, and so on.
- System Testing can therefore comprise "n" different tests; or, test for all aspects together as part of one test or a few tests. That is, it could mean:
    1) Integration Testing casing reasonable time-span to cover real-life situations.
    2) Usability / User interface Testing.
    3) Load Testing in terms of :
        1. Volume / Size.
        2. No. of Simultaneous Users.
        3. TPM/TPS: Transactions per Minute/Second.
    4) Stress Testing.
    5) Testing of Availability (24x7).

White box testing (WBT) is also called Structural or Glass box testing.

Why White box testing (WBT) ?

We do WBT because Black box testing is unlikely to uncover numerous sorts of defects in the program. These defects can be of the following nature:

- Logic errors and incorrect assumptions are inversely proportional to the probability that a program path will be executed. Error tend to crawl into our work product when we design and implement functions, conditions or controls that are out of the program
- The logical flow of the program is affected because our unaware assumptions about flow of control and data may lead to design errors that are uncovered only in path testing.
- Typographical errors are random, some of which will be uncovered by syntax checking mechanisms but others defects will pass undetected until testing starts.

Advantages of White Box Testing

- White box testing is testing against the implementation and will discover faults of commission, indicating that part of the implementation is faulty
- As we know the internal structure of a product, tests can be conducted to ensure that the internal operations performed according to the specification.

Disadvantages of White Box Testing

- White box testing is much more expensive (In terms of resources and time) than black box testing.
- White box testing is concerned only with testing the software product; it cannot guarantee that the complete specification has been implemented.
- It requires the source code.
- It is much more laborious in the determination of suitable input data and the determination if the software is or is not correct.

Black Box Testing

Black box is a test design method. Black box testing treats the system as a "black-box", so it doesn't explicitly use Knowledge of the internal structure/code. That means the test engineer not having knowledge of the internal working/language of the software system. It focuses on the functionality part of the module.
Some people like to call black box testing as behavioral, functional, opaque-box, and closed-box. While the term black box is most widely used, many people use the terms "behavioral" and "structural" for black box and white box respectively. Personally we feel that there is a trade off between the approaches used to test a product using white box and black box types. There are some bugs that cannot be found using only black box or only white box. If the test cases are extensive and the test inputs are also from a large sample space then it is always possible to find majority of the bugs through black box testing.

Error Guessing

Error Guessing comes with experience with the technology and the project. Error Guessing is the art of guessing where errors can be hidden. There are no specific tools and techniques for this, but you can easily write test cases depending on the scenario: Either when reading the functional documents or when you are testing and find an error that you have not documented.
For example : passing "null" value, using numeric data when alphabetic input required and vice a versa.

Boundary Value Analysis

Boundary Value Analysis (BVA) is a test data collection technique (Functional Testing technique) where the extreme values are chosen. Boundary values include maximum, minimum, just inside/outside boundaries, typical values, and error values. The hope is that, if a system works correctly for these special values then it will work correctly for all values in between.

- Extends equivalence partitioning
- Test both sides (min and max)of each boundary
- Considers output boundaries for test cases too
- Test min-1 ,min, min+1, max-1 ,max, max+1, typical values
- BVA focuses on the boundary of the input space to identify test cases
- Rational is that errors tend to occur near the extreme values of an input variable

Advantages of Boundary Value Analysis

1. Robustness Testing - Boundary Value Analysis plus values that go beyond the limits
2. Min - 1, Min, Min +1, Normal, Max -1, Max, Max +1
3. Forces attention to exception handling

Equivalence Partitioning

Equivalence partitioning is a black box testing method that divides the input domain of a program into classes of data from which test cases can be derived.

EP can be defined according to the following guidelines:

1. If an input condition specifies a range, one valid and two invalid classes are defined.

2. If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.

3. If an input condition specifies a member of a set, one valid and one invalid equivalence class is defined.

4. If an input condition is Boolean, one valid and one invalid class is defined.

Advantages of Black Box Testing

- Tester can be non-technical. (Knowledge of programming language not required)

- This testing is most likely to find those bugs as the user would find.
- Testing helps to identify the ambiguity and contradiction in functional specifications.
- Test cases can be designed as soon as the functional specifications are complete

Disadvantages of Black Box Testing

- Chances of having repetition of tests that are already done by programmer.
- The test inputs needs to be from large sample space.
- It is difficult to identify all possible inputs in limited testing time. So writing test cases is slow and difficult.
- Chances of having unidentified paths during this testing

Test Oracles

An oracle is a mechanism for determining whether the program has passed or failed a test.

A complete oracle would have three capabilities and would carry them out perfectly:

- A generator, to provide predicted or expected results for each test.
- A comparator, to compare predicted and obtained results.
- An evaluator, to determine whether the comparison results are sufficiently close to be a pass.

One of the key problems with oracles is that they can only address a small subset of the inputs and outputs actually associated with any test. The tester might intentionally set the values of some variables, but the entire program's other variables also have values. Configuration settings, amount of available memory, and program choice/options can also affect the test results. As a result, our evaluation of the test results in terms of the test inputs is based on incomplete data, and may be incorrect.

Any of the oracle capabilities may be automated. For example, we may generate predictions for a test from previous test results on this program, from the behavior of a previous release of this program or a competitor's program, from a standard function. We may generate these by hand, by a tool that feeds input to the reference program and captures output or by something that combines automated and manual testing. We may instead generate predictions from specifications, regulatory requirements or other sources of information that require a human to evaluate the information in order to generate the prediction.

The following example illustrates the use of oracles:

- (Oracles) Drag and Drop Images into Spreadsheets

Summary
In Microsoft Excel, a user can drag a picture from an Internet browser and drop it into a cell inside a spreadsheet. However, in OpenOffice.org *Calc*, when a user drags a picture into a cell, only a link to that picture is placed inside the spreadsheet.
Application Description
OpenOffice.org is a free office suite that includes a word processor, a spreadsheet creator, and a presentation creator. Calc is the spreadsheet component of OpenOffice.org and is used to create graphs , calculate statistics, merge data, and perform other math-data functions.

Microsoft Office 2003 is the most commonly used office suite. Excel is the spreadsheet component of Office 2003 and can be used to perform the same tasks as Calc.

Test Design

In Oracle-based testing, we compare the behavior of the program under test to the behavior of a source , which we believe accurate (the oracle).

One of the frequent tasks when testing a program is a survey of the program's capabilities. You walk through then entire product, trying out each feature to see what the product can do, what seems uncomfortable , what it does well, and what seems obviously unstable.

The tester doing the survey has to constantly evaluate the program. Is this behavior reasonable? Correct? In line with user expectations? A tester who is expert with this type of product will have no difficulty making these evaluations, but a newcomer needs a reference for assistance. An oracle is one such reference.

Open Office (OO) was designed to compete with Microsoft Office, so it makes sense to use MS Office as the reference point when surveying Open Office.

Results/Relevance

In both cases, dragging in a picture that was linked and dragging in one that was not, Excel showed us the picture in the spreadsheet and Calc did not. Instead, Calc gave us a link to the page.

We can't say that Calc is wrong. Some users will wish to store the link instead of the actual picture. And the user who wants to insert the picture can save the image locally, and insert the local copy of the graphic into the spreadsheet.

An Open Office tester may raise the issue that Open Office 's behavior varies from MS Office. If a person switches back and forth between Open Office and MS Office (a scenario in a company that has not yet accepted any of the Office applications as its special standard), small differences viz. this will lead to user errors and violated expectations.

Oracle-based testing allowed us to evaluate one application by comparing it to an industry standard. By doing this, we can spot what basic features the application should have, what it can expect that changeover users will want to see, and ways in which to improve the application.

Test Case Documents

Designing good test case is a composite skill. The complexity comes from three sources appended below:

1. Test cases help us discover information. Different types of tests are more effective for different module of information.
2. Test cases can be “good” in a variety of ways.
3. People tend to create test cases according to certain testing styles, such as risk-based testing or domain testing. Good domain tests are different from good risk-based tests.

What’s a test case?

“A test case specifies the pretest state of the AUT (application under test) and its environment, the test inputs or conditions, and the expected result. The expected result specifies what the AUT should produce from the test inputs. This specification includes messages created by the AUT, exceptions, returned values, and resultant state of the AUT and its environment. Test cases may also specify initial and resulting conditions for other objects that constitute the AUT and its environment.”

What’s a scenario?

A scenario is a flow of action, used to help a person think through a complex problem or system.

A Scenario

The primary objective of test case design is to derive a set of tests that have the highest approach of discovering defects in the software. Test cases are designed based on the analysis of requirements, use cases, and technical specifications, and they should be developed in parallel with the software development effort.

A test case describes a set of actions to be performed and the results that are expected. A test case should aim exact functionality or aim to work out a valid path through a use case. This should comprise unacceptable user actions and prohibited inputs that are not necessarily listed in the use case. A test case is described depends on several factors, e.g. the number of test cases, the frequency with which they change, the level of automation employed, the skill of the testers, the chosen testing methodology, staff turnover, and risk.

A generic format of the test cases appended below.

Test Case ID - The test case id must be unique in nature.
Test Case description - The test case description in brief.
Prerequisite of Test - The test pre-requisite clearly describes what should be present in the system, rather state of the system before the test can be executed.
Test Inputs - The test input is the test data prepared to be fed to the system.
Test steps/action - The test steps are the step-by-step/actions instructions on how to carry out the test.
Expected Result - The expected results means the system must give as output or how the system must react based on the test steps and input data.
Actual Result – The actual results means outputs of the action for the given input data or how the system reacts for the given input data.
Pass/Fail- The test case pass or fail
Bug ID- A unique bug/defect no.
Bug Date- A date on which defect found

For example consider a Tea/Coffee/Pepsi vending machine.

1. User should be able to insert Rs.5 coin. (We will be using simulator)
2. Rs. 5 coin should be recognized by system. (We will be using simulator)
3. Screen should be visible with 4 options viz Tea, Coffee, Pepsi, Abort.
4. Cursor should be on “Tea” by default.
5. User should be able to scroll using scroll button.
6. Cups should be available in stack or below the proper vent when user selects between the choices.
7. Selection of user choice should be done by second press button.
8. Select button should be disabled once it is pressed till user enters another Rs.5 coin.
9. User will not be able to enter second coin, till one serving is not completed.
10. Machine will remain active for 1 minute after entering the coin. After one minute, it should return coin.
11. When user selects “Abort”, Rs. 5/- coin is returned.

Tea serving

1. When user selects tea, 1 spoon tea should be released from tea pot, 1 cup water which is 100 ml is released, heated. User gets one cup of tea from T/C vent.
2. When user selects tea, tea and / or water is not available, heater is not working; it should give Rs. 5 back.
3. If cup is not available below T/C vent, it will return Rs.5/- coin back.

Coffee serving

4. When user selects Coffee, 1 spoon Coffee should be released from Coffee pot, 1 cup water which is 100 ml is released, heated. User gets one cup of Coffee from T/C vent.
5. When user selects Coffee, Coffee and / or water is not available, heater is not working, it should give Rs. 5 back.
6. If cup is not available below T/C vent, it will return Rs.5/- coin back.

Pepsi serving

7. When user selects Pepsi, 1 spoon Pepsi should be released from Pepsi pot, 1 cup water which is 100 ml is released, chilled. User gets one cup of Pepsi from P vent.
8. When user selects Pepsi, Pepsi and / or water is not available, chiller is not working, it should give Rs. 5 back.
9. If cup is not available below P vent, it will return Rs.5/- coin back.

Test cases for vending machine
Created by:
Reviewed by:
Created date:
Review date
Version:

Sr No	TS Ref	Test case title	Steps	Exp result	Actual result	Pass / Fail	Bug ID	Bug date
1	TS1	When user enters Rs. 5 coin, system shows menu.	Through simulator, enter Rs. 5/- coin.	Screen should be alive showing “Tea”, “Coffee”, “Pepsi”, “Abort”				
2	TS1	Cursor is on "Tea"	1. Through simulator, enter Rs. 5/- coin. 2. Screen should be alive showing “Tea”, “Coffee”, “Pepsi”, “Abort”	Cursor is on "Tea"				
3	TS1	"Scroll" button should be active.	1. Through simulator, enter Rs. 5/- coin. 2. Screen should be alive showing “Tea”, “Coffee”, “Pepsi”,	Cursor should move to "Coffee"				

			“Abort” 3. Cursor is on "Tea" 4. User presses "Scroll" button					
4	TS1	"Select" button should be active.	1. Through simulator, enter Rs. 5/- coin. 2. Screen should be alive showing “Tea”, “Coffee”, “Pepsi”, “Abort” 3. Cursor is on "Tea" 4. User presses "Scroll" button 5. Cursor moves to "Coffee" 6. User places a cup below "T/C" vent. 7. User presses "Select" button	Stub designed for the purpose should tell that coffee is served				
5	TS1	One spoon of coffee is served	1. Through simulator, enter Rs. 5/- coin. 2. Screen should be alive showing “Tea”, “Coffee”, “Pepsi”, “Abort” 3. Cursor is on "Tea" 4. User presses "Scroll" button 5. Cursor moves to "Coffee" 6. User places a cup below "T/C" vent. 7. User presses "Select" button	Stub designed for the purpose should tell that 1 spoon of coffee is released.				
6	TS1	100 ml of water is served	1. Through simulator, enter Rs. 5/- coin. 2. Screen should be alive showing “Tea”, “Coffee”, “Pepsi”, “Abort” 3. Cursor is on "Tea" 4. User presses "Scroll" button 5. Cursor moves to "Coffee" 6. User places a cup below "T/C" vent. 7. User presses "Select" button	Stub designed for the purpose should tell that 100 ml of coffee is released.				
7	TS1	water is heated and served	1. Through simulator, enter Rs. 5/- coin. 2. Screen should be alive showing “Tea”, “Coffee”, “Pepsi”, “Abort” 3. Cursor is on "Tea" 4. User presses "Scroll" button 5. Cursor moves to "Coffee"	Stub designed for the purpose should tell that water is heated and served.				

			6. User places a cup below "T/C" vent. 7. User presses "Select" button					
8	TS1	After serving Machine goes to original state	1. Through simulator, enter Rs. 5/- coin. 2. Screen should be alive showing “Tea”, “Coffee”, “Pepsi”, “Abort” 3. Cursor is on "Tea" 4. User presses "Scroll" button 5. Cursor moves to "Coffee" 6. User places a cup below "T/C" vent. 7. User presses "Select" button 8. Coffee is served	Machine should come to original stage.				
9	TS2	Rs. 5/- is returned when user places a cup below "P" vent and select "Tea" or "Coffee"	1. Through simulator, enter Rs. 5/- coin. 2. Screen should be alive showing “Tea”, “Coffee”, “Pepsi”, “Abort” 3. Cursor is on "Tea" 4. User presses "Scroll" button 5. Cursor moves to "Coffee" 6. User places a cup below "P" vent. 7. User presses "Select" button	Stub designed should tell that machine has returned Rs. 5/-				
10	TS3	Rs. 5/- is returned when coffee is not available and select "Coffee"	1. Through simulator, enter Rs. 5/- coin. 2. Screen should be alive showing “Tea”, “Coffee”, “Pepsi”, “Abort” 3. Cursor is on "Tea" 4. User presses "Scroll" button 5. Cursor moves to "Coffee" 6. User places a cup below "T/C" vent. 7. User presses "Select" button	Stub designed should tell that machine has returned Rs. 5/-				

11	TS4	Rs. 5/- is returned when water is not available and select "Coffee"	1. Through simulator, enter Rs. 5/- coin. 2. Screen should be alive showing “Tea”, “Coffee”, “Pepsi”, “Abort” 3. Cursor is on "Tea" 4. User presses "Scroll" button 5. Cursor moves to "Coffee" 6. User places a cup below "T/C" vent. 7. User presses "Select" button	Stub designed should tell that machine has returned Rs. 5/-				
12	TS5	Rs. 5/- is returned when heater is not working select "Coffee"	1. Through simulator, enter Rs. 5/- coin. 2. Screen should be alive showing “Tea”, “Coffee”, “Pepsi”, “Abort” 3. Cursor is on "Tea" 4. User presses "Scroll" button 5. Cursor moves to "Coffee" 6. User places a cup below "T/C" vent. 7. User presses "Select" button	Stub designed should tell that machine has returned Rs. 5/-				

Testing Tools

- HP Mercury
  - Winrunner
  - Quick Test Professional
  - Test Director
  - Quality Center
- IBM Rational
  - Rational Robot
  - Rational Performance Tester
  - Rational Functional Tester
  - Rational Test Manager
- Borland Segue Micro Focus
  - Silk Test
  - Silk Performer

Defect Analysis-

What is a Defect?

For a test engineer, a defect is following: -

- Any deviation from specification
- Anything that causes user dissatisfaction
- Incorrect output
- Software does not do what it intended to do.

Categories of Defects

- Variance from product specifications
- Variance from customer / user expectations
- These can be:
    - Wrong
    - Missing
    - Extra

Examples of Defects

- User gives wrong / incomplete requirements
- Analyst interprets requirement incorrectly
- Requirements are not recorded correctly
- Incorrect design specs
- Incorrect program specs
- Errors in coding
- Data entry errors
- Errors in testing: falsely detect an error / fail to detect existing errors
- Mistakes in error correction

Software Maintenance

Although software does not show aging or wear and tear with use, with upcoming advanced technologies and changing user requirements, software products get outdated or fail to support the changes in their environment for different reasons. What should stakeholder do with such software products on which a lot of money has been spent and which is now not suitable due to minor errors or incompatibilities with the environment.

Maintenance can only happen efficiently if the earlier phases are done properly. There are four major problems that can slow down the maintenance process: unstructured code, maintenance programmers having insufficient knowledge of the system, documentation not available, out of date, or at best insufficient, and software maintenance having a bad image. The success of the maintenance phase relies on these problems being fixed earlier in the life cycle.

Maintenance consists of four parts.

Corrective maintenance deals with fixing bugs in the code itself.

Adaptive maintenance deals with adapting the software to new surroundings.

Perfective maintenance deals with updating the software according to changes in user requirements.

Preventive maintenance 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.' As products age it becomes more difficult to keep them updated with new user requirements. Maintenance costs developers time, effort, and money. This requires that the maintenance phase be as efficient as possible. There are several steps in the software maintenance

phase. The first is to try to understand the design that already exists. The next step of maintenance is reverse engineering in which the design of the product is reexamined and restructured. The final step is to test and debug the product to make the new changes work properly.

Types of Software Maintenance

There are four types of maintenance corrective, adaptive, perfective, and preventive.

Corrective maintenance deals with the repair of faults or defects found. A defect can result from design errors, coding errors and logic errors. Design errors occur when, for example, changes made to the software are incorrect, incomplete, wrongly communicated or the change request is misunderstood. Logic errors result from invalid tests and conclusions, incorrect implementation of design specifications, faulty logic flow or incomplete test of data. Coding errors are caused by incorrect implementation of detailed logic design and incorrect use of the source code logic. Defects are also caused by data processing errors and system performance errors. All these errors, sometimes called 'residual errors' or 'bugs', prevent the software from conforming to its agreed specification. The need for corrective maintenance is usually initiated by bug reports drawn up by the end users. Examples of corrective maintenance include correcting a failure to test for all possible conditions or a failure to process the last record in a file.

Adaptive maintenance consists of adapting software to changes in the environment, such as the hardware or the operating system. The term environment in this context refers to the totality of all conditions and influences which act from outside upon the system, for example, government policies , business rule, work patterns, software and hardware operating platforms. The need for adaptive maintenance can only be recognized by monitoring the environment. An example of a government policy that can have an effect on a software system is the proposal to have a 'single European currency', the ECU. An accepting this modification will require that banks in the various member states, for example, make significant changes to their software systems to accommodate this currency. Other examples are an implementation of a database management system for an existing application system and an adjustment of two programs to make them use the same record structures.

Perfective maintenance mostly works with accommodating to new or changed user requirements. Perfective maintenance concerns functional enhancements to the system and activities to increase the system's performance or to enhance its user interface. A successful piece of software tends to be subjected to a succession of changes, resulting in an increase in the number of requirements. This is based on the premise that as the software becomes useful, the users tend to experiment with new cases beyond the scope for which it was initially developed. Examples of perfective maintenance include modifying the payroll program to incorporate a new union settlement, adding a new report in the sales analysis system, improving a terminal dialogue to make it more user-friendly, and adding an online HELP command.

Preventive maintenance activities intended at increasing the system's maintainability, such as updating documentation, adding comments, and improving the modular structure of the system. The long-term effect of corrective, adaptive and perfective changes increases the system's complexity. As a large program is continuously changed, its complexity, which reflects deteriorating structure, increases unless work is done to maintain or reduce it. This work is known as preventive change. The change is usually initiated from within the maintenance organization with the intention of making programs easier to understand and hence facilitating future maintenance work. Examples of preventive change include restructuring and optimizing code and updating documentation. Among these four types of maintenance, only corrective maintenance is 'traditional' maintenance. The other types can be considered software 'evolution'. Software evolution is now widely used in the software maintenance community.

Risks in Software Maintenance

As software maintenance leads to enhancement and corrections in the software, it also bears the major risk of partial or complex failure of the software. While correcting the residual errors in the software, modifications can in turn introduce new bugs in other linked parts too. Detecting this, needs rigorous testing of all the altered areas of the software and all its linked components. The cost effort and time consumption for these activities are usually very high leading to high maintenance cost.

Even after taking all precautions any alteration in the software carries the risk of high costs, slips of time schedules, non acceptance in the market and even complete or partial failure of the software. Hence any maintenance thread initialization needs highly mature analysis and conclusive decisions in tandem with established management and prime technical expertise.

Summary

You should be able to give reasons why both specification-based (black-box) and (white-box) structure-based approaches are useful, and list the common techniques for each of these approaches. You should be able to explain the characteristics and differences between different techniques like specification, structure and experience-based .

You should be able to write test cases from given software models using equivalence partitioning, boundary value analysis, decision tables and state transition diagrams. You should understand the main purpose of each of these four techniques, what level and type of testing could use each technique and how coverage can be measured for each of them.

You should be able to describe the concept and importance of code coverage. You should be able to explain the concepts of statement and decision coverage and understand that these concepts can also be used at test levels other than component testing (such as business procedures at system test level). you should be able to list the factors that influence the selection of the appropriate test design technique for a particular type of problem, such as the type of system, risk, customer requirements, requirements models or testing knowledge.

Maintenance clearly plays an important role in the life cycle of a software product. The focus of software maintenance is to increase the life and usability of the software. While “traditional maintenance” applies only to corrective maintenance – fixing bugs in the code, the maintenance phase also incorporates three other main aspects that are considered to be elements of software advancement. Adaptive maintenance serves to modify the software for use in new environments, such as a new hardware platform or interfacing with a different database system. Perfective maintenance concerns adding new functionality to a product, typically as a result of a request from customer. Finally, preventive maintenance increases the maintainability of a system, through updating documentation or changing the structure of the software.

Self Test

83. After regression testing is complete, the program and test documentation must be updated to reflect the changes. State whether this statement is true or false.
    a) True. b) False
84. Testing types refer to different approaches towards testing a computer program, system or product. These types are Black box and White box. State whether this statement is true or false.
    a) True. b) False
85. Testing activities should be performed during Requirements Analysis phase --------
    a) Determine correctness

b) Generate functional test data.
c) Only a) , b)
d) All of the above
e) None of the above

86. Advantages of Black Box Testing are ----------
a) Tester can be non-technical. (Knowledge of programming language not required)
b) This testing is most likely to find those bugs as the user would find.
c) Testing helps to identify the ambiguity and contradiction in functional specifications.
d) Test cases can be designed as soon as the functional specifications are complete
e) All of the above
f) None of the above

87. Testing activities should be performed during Design -
a) Determine correctness and consistency
b) Generate structural and functional test data.
c) Only a) , b)
d) All of the above
e) None of the above

88. Testing should be involved ----------the SDLC in order to bring out a quality product.
a) Throughout
b) In between
c) All of the above
d) None of the above

89. Boundary Value Analysis (BVA) is a test data --------technique (Functional Testing technique) where the ----------values are chosen.
a) selection
b) extreme
c) Pieces
d) Only a) , b)
e) All of the above

90. Equivalence partitioning is a black box testing method that --------the input domain of a program into ----------of data from which test cases can be derived.
a) Divides
b) Classes
c) Pieces
d) Only a) , b)
e) All of the above

91. A black box testing techniques are
a) Error Guessing
b) Boundary Value analysis
c) Equivalence partitioning
d) Control flow based testing
e) Data Flow Testing
f) Only a) , b) and c)
g) All of the above
h) None of the above

92. A white box testing techniques are
a) Control flow based testing
b) Data Flow Testing
c) Error Guessing

d) Boundary Value analysis
e) Equivalence partitioning
f) Only a) and b)
g) All of the above
h) None of the above

93. The system test must focus on ----------groups, rather than identifying ---------units.
a) functional
b) the program
c) structure
d) a) and b)
e) All of the above
f) None of the above

94. The system test cases meant to test the system as per----------; ----------
a) the requirements
b) end-to end
c) credible,
d) a) and b)
e) All of the above
f) None of the above

95. . Why are both specification-based (black box testing) and structure-based testing (white box testing) techniques useful?
a) They find different types of defect.
b) Using more techniques is always better.
c) Both find the same types of defect.
d) Because specifications tend to be unstructured.

96. A good scenario test has five key characteristics. It is ------, that is(b) --------(c)--------, (d) --------and (e) easy to evaluate.
a) a story
b) motivating,
c) credible,
d) complex,
e) All of the above
f) None of the above

97. For a test engineer a defect is -------- -
a) Any deviation from specification
b) Anything that causes user dissatisfaction
c) Incorrect output
d) Software does not do what it intended to do.
e) All of the above
f) None of the above

98. Defects means
a) Variance from product specifications
b) Variance from customer / user expectations
c) All of the above
d) None of the above

99. Categories of Defects can be
a) Wrong
b) Missing
c) Extra
d) All of the above
e) None of the above

100. The primary objective of test case design is to ----------that have the -------- attitude of discovering defects in the software
   a) derive a set of tests
   b) highest
   c) lowest
   d) a) and b)
   e) a) and c)
   f) None of the above

101. A complete Test Oracle would have following capabilities :-
   a) A *comparator*, to compare predicted and obtained results
   b) A *generator*, to provide predicted or expected results for each test.
   c) An *evaluator*, to determine whether the comparison results are sufficiently close to be a pass
   d) Only a) , b) and c)
   e) All of the above
   f) None of the above

102. Oracle-based testing allowed us to -----one application by --------it to an industry standard.
   a) evaluate
   b) comparing
   c) examine
   d) Only a) , b)
   e) All of the above

Lesson 6

Software Quality Assurance

Overview

"Of all men's miseries, the bitterest is this: to know so much and have control over nothing."
Herodotus

Software Quality Assurance (**SQA**) consists of a means of monitoring the software engineering processes and methods used to ensure quality. The methods by which this is accomplished are many and varied, and may include ensuring conformance to one or more standards, such as ISO 9000 or CMMI.

Quality Assurance is a professional competency whose focus is directed at critical processes used to build products and services. The profession is charged with the responsibility for tactical process improvement initiatives that are strategically aligned to the goals of the organization.

Learning Objectives

- To understand Software Quality Assurance
- To understand Attributes for Quality, Quality Standards
- To understand Check Sheet (Check List)
- To understand SEI CMM

Introduction - Software Quality Assurance

Concepts and Definitions

Software Quality Assurance (SQA) is defined as a planned and systematic approach to the evaluation of the quality of and adherence to software product standards, processes, and procedures. SQA includes the process of assuring that standards and procedures are established and are followed throughout the software acquisition life cycle. Compliance with agreed-upon standards and procedures is evaluated through process monitoring, product evaluation, and audits. Software development and control processes should include quality assurance approval points, where an SQA evaluation of the product may be done in relation to the applicable standards.

Quality Assurance (QA) is the application of planned, systematic quality activities to ensure that the project will utilize all processes needed to meet requirements. A quality assurance department, or similar organization, often oversees quality assurance activities. QA support, may be provided to the project team, the management of the performing organization, the customer or sponsor, and other stakeholders, which are not actively involved in the work of the project. QA also provides an umbrella for another important quality activity, continuous process improvement. Continuous process improvement provides an iterative means for improving the quality of all processes.

Continuous process improvement reduces waste and non-value-added activities, which allows processes to operate at increased levels of efficiency and effectiveness. Process improvement is distinguished by

its identification and review of organizational business processes. It may be applied to other processes within an organization as well, from micro processes, such as the coding of modules within a software program, to macro processes, such as the opening of new markets.

SQA processes provide assurance that the software products and processes in the project life cycle conform to their specified requirements by planning, enacting, and performing a set of activities to give enough confidence that quality is being built into the software product. This means ensuring that the problem is clearly and adequately stated and that the solution's requirements are properly defined and expressed. SQA seeks to maintain the quality throughout the development and maintenance of the product by the execution of a variety of activities at every phase which can result in early identification of problems, an almost inevitable feature of any complex activity. The role of SQA with respect to process is to ensure that planned processes are appropriate and later implemented according to plan, and that relevant measurement processes are provided to the appropriate organization.

Importance of Software Quality Assurance

Quality Assurance (QA) is an important activity for any business that produces products to be used by others. Software Quality Assurance (SQA) is to give management with suitable clarity into the processes being used by the software project and of the products being built . Quality Assurance (QA) assures that the requirements. are meeting and the application is developed as per the laid down procedure. It provides confidence to the management with the necessary data about the quality of product. The data provided by quality assurance identify the problem in the product then it is management's duty to address the problem and deploy required resources to solve the quality issues.
In addition, SQA also acts as a focal point for managing process improvement activities. Software Quality Assurance monitors process and product which assures that the software development and control processes described in the project's Management Plan are correctly carried out and that the project's procedures and standards are followed.

Software Quality Assurance Activities

Product evaluation and process monitoring are the SQA activities that assure the software development and control processes described in the project's Management Plan are correctly carried out and that the project's procedures and standards are followed. Products are monitored for conformance to standards and processes are monitored for conformance to procedures. Audits are a key technique used to perform product evaluation and process monitoring. Review of the Management Plan should ensure that appropriate SQA approval points are built into these processes.

Product evaluation is an SQA activity that confirms that standards are being followed. Ideally, the first products monitored by SQA should be the project's procedures and standards. SQA assures that achievable and clear standards exist and then evaluates compliance with the established standards of the software product. Product evaluation assures that the software product reflects the requirements of the applicable standard(s) as identified in the Management Plan.

Process monitoring is an SQA activity that confirms that suitable steps to carry out the process are being followed. SQA monitors processes by comparing the actual steps carried out with those in the documented procedures. The Assurance section of the Management Plan specifies the methods to be used by the SQA process monitoring activity.

A fundamental SQA technique is the audit, which looks at a process and/or a product in depth, comparing them to established procedures and standards. Audits are used to review management, technical, and assurance processes to provide an indication of the quality and status of the software product.

The purpose of an SQA audit is to confirms that appropriate control procedures are used, that required documentation is maintained, and that the developer's status reports accurately reflect the status of the activity. The SQA product is an audit report to management consisting of findings and recommendations to bring the development into meeting with standards and/or procedures.

The major impediments (obstacle) to QA come from management, which is typically results oriented, and sees little need for a function that emphasizes managing and controlling processes. Thus, many of the impediments to QA are associated with processes, and include the following:

- Management does not insist on compliance to processes
- Workers are not convinced of the value of processes
- Processes become obsolete
- Processes are difficult to use
- Workers lack training in processes
- Processes are not measurable
- Measurement can threaten employees
- Processes do not focus on critical aspects of products

Standards and Procedures

Establishing standards and procedures for software development is critical, since these provide the framework from which the software evolves. Standards are the established criteria to which the software products are compared. Procedures are the established criteria to which the development and control processes are compared. Standards and procedures establish the prescribed methods for developing software; the SQA role is to ensure their existence and adequacy. Proper documentation of standards and procedures is necessary since the SQA activities of process monitoring, product evaluation, and auditing rely upon unequivocal definitions to measure project compliance.

Types of Standards comprise:

Documentation Standards specify form and content for planning, control, and product documentation and provide consistency throughout a project. For example, the documentation standards are NASA's Data Item Descriptions (DIDs). Design Standards specify the form and content of the design product. They provide rules and methods for translating the software requirements into the software design and for representing it in the design documentation.

Code Standards specify the language in which the code is to be written and define any restrictions on use of language features. They describe legal language structures, style conventions, rules for data structures and interfaces, and internal code documentation. Procedures are explicit steps to be followed in carrying out a process. All processes should have documented procedures. Examples of processes for which procedures are needed are configuration management, nonconformance reporting and corrective action, testing, and formal inspections.

If developed according to the NASA DID, the Management Plan describes how to control the software development processes, such as configuration management, for which there have to be procedures, and contains a list of the product standards. Standards are to be documented according

to the Standards and Guidelines DID in the Product Specification. The planning activities required to assure that both products and processes comply with designated standards and procedures are described in the QA portion of the Management Plan.

Quality Attributes

Quality is a multifaceted concept driven by customer requirements. The level of quality can vary significantly from project to project and between organizations. In IT, the attributes of quality are examined in order to understand the components of quality and as a basis for measuring quality. Some of the commonly accepted quality attributes for an information system are described in Figure A.

Management needs to develop quantitative, measurable "standards" for each of these quality criteria for their development projects. For example, management must decide the degree of maintenance effort that is acceptable, the amount of time that it should take for a user to learn how to use the system, etc.

Attributes	Definition
Correctness	Extent to which a program satisfies its specifications and fulfills the user's mission objectives.
Reliability	Extent to which a program can be expected to perform its intended function with required precision.
Efficiency	The amount of computing resources and code required by a program to perform a function.
Integrity	Extent to which access to software or data by unauthorized persons can be controlled.
Usability	Effort required learning, operating, preparing input, and interpreting output of a program.
Maintainability	Effort required locating and fixing an error in an operational program.
Testability	Effort required testing a program to ensure that it performs its intended function.
Flexibility	Effort required modifying an operational program.
Reusability	Extent to which a program can be used in other applications – related to the packaging and scope of the functions that programs perform.
Interoperability	Effort required to couple one system with another.

Figure A Commonly Accepted Quality Attributes

Check Sheet

A check sheet (also called a checklist or tally sheet) of events or occurrences is a form used to gather and record data in an organized manner. This tool records the number of occurrences over a specified interval of time to determine the frequency of an event. The data is recorded to support or objectively validate the significance of the event. It may follow a Pareto analysis or cause-and effect diagram to validate or verify a problem, or it may be used to build Pareto charts or histograms. Figure B shows a sample check sheet.

(Daily System) Failures	Week of dd/mm/yy					
	Day 1	Day 2	Day 3	Day 4	Day 5	Total

Figure B. Check Sheet

Check sheets can be used to record the following types of information:

- Project review results, such as defect occurrences, location, or type
- Documentation defects by type or frequency
- Cycle times, such as requirements to design or design to implementation
- Conformance to standards
- End user complaints of all types
- End user surveys
- Late deliveries

To use a check sheet:

1. Clarify what must be collected objectively.
2. Establish a format for the data collection that is easily understood by the collector.
3. Ensure those involved understand the objectives so the collection process is accurate.
4. Establish the sample size and time frame of data collection.
5. Instruct or train data collectors for consistency.
6. Observe, record, and collect data.
7. Tally the results.
8. Depending on the purpose, build a Pareto chart or histogram or evaluate the results to determine whether the original analysis is supported.

Advantages of check sheets are that they pre-define areas to discuss, limit the scope, and provide a consistent, organized, and documented approach. Disadvantages might be their applicability or limiting of other questions.

Questions on check sheets should be organized by topic and tested prior to use. A response of "I don't know" should be allowed for, and bias should be avoided. The person using the check sheet should understand the reason for the questions and be able to anticipate a response.

Feasibility Analysis Checklists

Economic feasibility: An evaluation of development cost weighed against the ultimate income or benefit derived from the developed system or product.

Technical feasibility: A study of function, performance, and constraints that may affect the ability to achieve an acceptable system.

Legal feasibility: A determination of any infringement, violation, or liability that could result from development of the system.

Economic feasibility

Have the benefits associated with the product/system/service been identified explicitly?
Have the benefits been quantified in dollar terms?
Does the configuration represent the most profitable solution? Can it be marketed successfully? Will ultimate payoff justify development risk?
What is the risk associated with cost and schedule estimates?

Technical feasibility

Are all elements of the system configuration understood?
Can the configuration be built within already defined cost and schedule bounds?
Does the technology exist to develop all elements of the system?
Does the system rely on proven technologies?
Are all interfaces clearly defined?
Are function and performance assured?
Can the configuration be adequately maintained?
Do technical resources exist?
What is the risk associated with the technology?
Can quality assurance be adequately performed on all elements of the system?
Does the proposed configuration properly interface with the system's external environment?
Are machine to machine and human to machine communication handled in an intelligent manner?

Legal feasibility

Does this configuration introduce undue liability risk?
Can proprietary aspects be adequately protected?
Is there potential infringement?

MATURITY MODEL

When thinking about improvement process, we must remember that this applies not only to software development process, but also to the testing process. We will look at some of the most popular models and methods of software process enhancement/improvement.

Before we look in detail at the maturity models, to start with know what we indicate by the term 'maturity model'. A maturity model is basically a collection of elements that are structured in such a way that they can describe characteristics of processes and their effectiveness.

A maturity model can provide:

- A starting point
- A shared vision
- A structure for organizing actions
- Use of previous experience
- Determining real value of improvements

SEI CAPABILITY MATURITY MODEL (SEI CMMI)

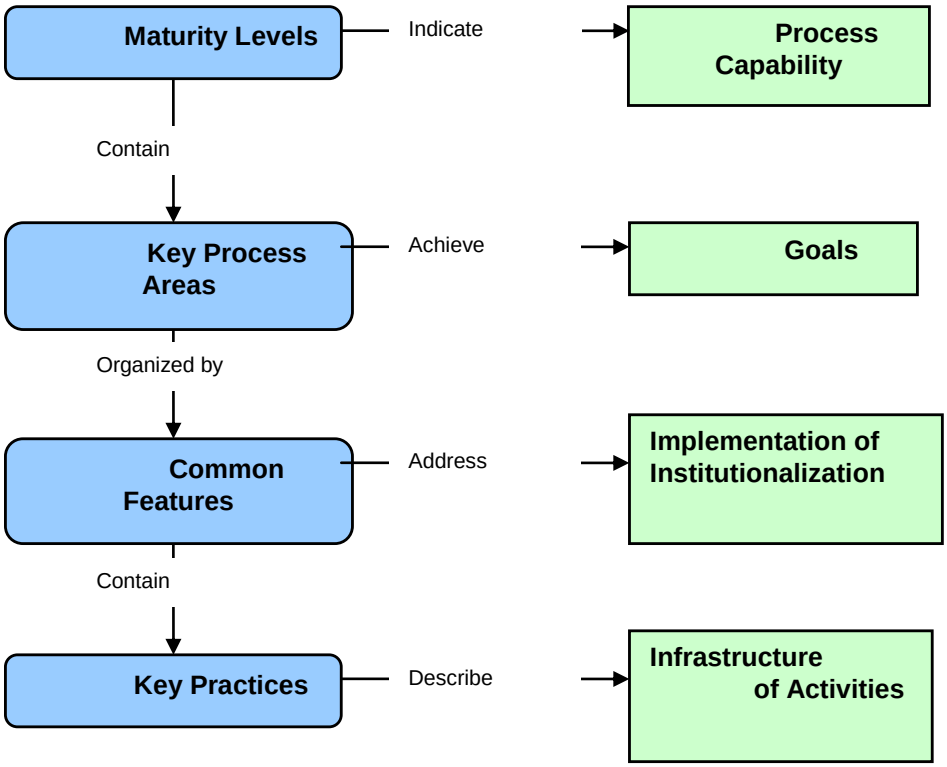
The Capability Maturity Model, , is a baseline of practices that should be implemented in order to develop or maintain a product. The product can be completely software, or just partially software.

The SW-CMM focuses on the software practices whereas with the CMMI (integration), you may find both software and systems practices.

The CMMI and SW-CMM were published by the SEI. The models are commonly used throughout the world and considered by many to be 'the' standard within this area. The Capability Maturity Model (or CMM) has been in use from 1991. This then became what is known as the CMMI, the 'I' stands for integration.

CMM Structure

The CMM is made up of five maturity levels. Apart from Level 1, every maturity level is comprises of multiple key process areas. Each key process area is separated into five sections, which are called common features. The common features specify the key practices that should accomplish the goals of the key process area.



Maturity Levels

A maturity level can be thought of as a defined evolutionary plateau that the software process is aiming to achieve. The CMM defines five maturity levels which form the top-level structure of the CMM itself. Each level is a foundation that can be built upon to improve the process in order. Starting with basic management practices and succeeding through following established levels.
When progressing through the path to improving a software process, the path will without doubt lead through each maturity level. The goal should be to achieve success in each maturity level, using what has been learned and implementing this within the software process.

1 – Initial

At this level the software development process is often not stable, and often any existing good practices are destroyed by bad planning. Commonly at this level when problems crop up in a project, short-cuts are taken, the original plan is abandoned, and with it any proper process structure. This leaves determining the software process capability a difficult task, as most of the defining qualities of the process are volatile. Some proof may exist at this level of a software process through individuals, but possibly not through the organizations capability.

2 – Managed

The software processes of a typical 'Level 2' organization would show that there is some discipline, as planning and tracking activities in a given project have been repeated elsewhere. Project related processes are under efficient control by using a management system.

3 – Defined

This level of organization will show that consistency exists within the software engineering activities and the management activities as the processes used are stable and repeatable. Defined roles and responsibilities exist within the software process, and an organization-wide understanding of the process exists.

4 – Quantitatively Managed

A typical 'level 4' organization will include predictability, as the process is measured and controlled by specified limits. This predictability has the advantage of giving organizations to foresee trends in process and quality within the specified limits. If the process strays out-of-bounds of the limit, then corrective action is taken. Products emerging from these types of organizations are typically of a high quality.

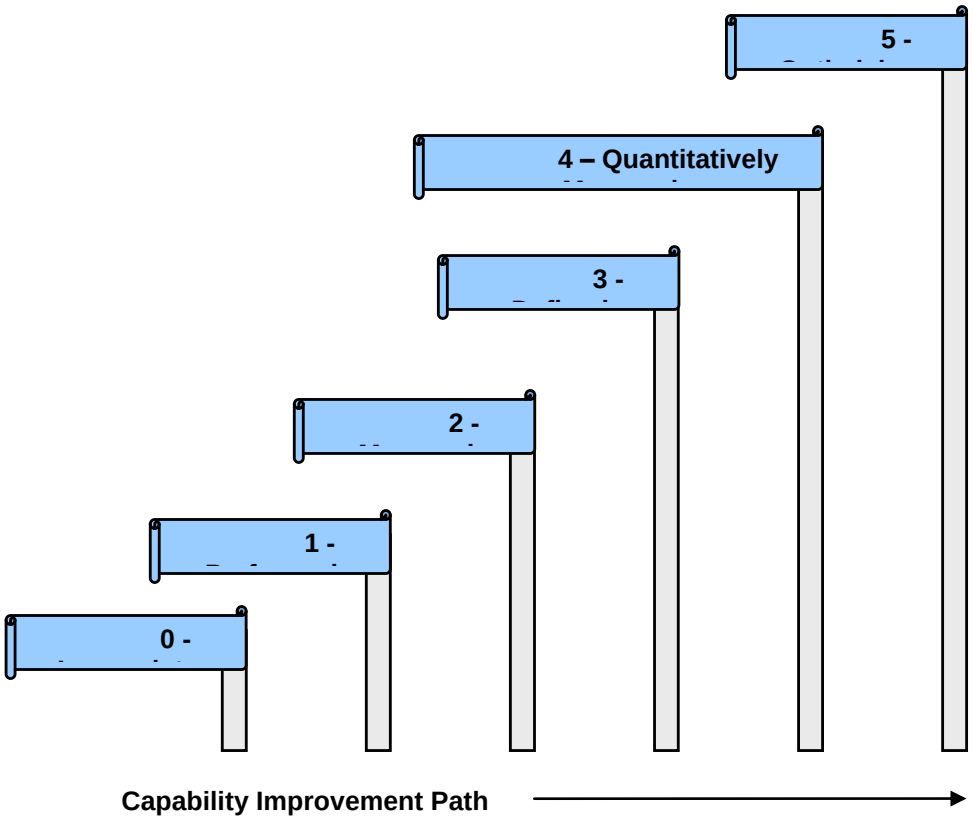
5 – Optimizing

This level of organization will explain that continuous improvement is a high priority. Their focus aims to expand the process capability, which has the effect of modifying project related process performance. New technologies and methods are welcomed here, and new processes and improvements to existing processes are their goal.

Process Capability

The software process capability determines what can be achieved by undertaking a specific software process. It achieves this by describing the range of expected results. It can be used to provide an idea of the outcome of future projects that use a specific process. There are six capability levels. Each level provides a foundation that can be built on for continuous process improvement. The levels are cumulative and so a top level would indicate that all lower levels have previously been satisfied.

Capability Levels:



Key Process Areas
Every maturity level is essentially made up of key process areas. The key processes themselves are each made up of a set of activities, which if all are performed correctly can fulfill a given goal.

Goals
The goals effectively form a summary of the key practices of a given key process area. They can be used as a yard-stick to ascertain whether or not the key process area has indeed been implemented correctly.

Common Features
Each key practice is sub-divided into five common feature areas:

- Commitment to Perform
- Ability to Perform
- Activities Performed
- Measurement and Analysis

- Verifying Implementation

The above common features can be used to show whether or not the implementation and institutionalization of a key process area is effective. The 'Activities Performed' common feature is associated with implementation, while the remaining four common features are associated with institutionalization.

Key Practices

A key process area can be thought of as individual key practices that assist in achieving the goal of a key process area. The key practices themselves detail the activities that will be the most important to the effectiveness of the implementation and institutionalization of the key process area.

Summary

Software QA involves the entire software development process - monitoring and improving the process, creation sure that any mutually accepted standards and procedures are followed, and ensuring that problems are found and dealt with. It is oriented to 'prevention'. Quality Assurance (QA) is the set of activities (viz. training, facilitation, measurement and analysis) needed to give enough confidence that processes are established and continuously improved in order to produce products or services that conform to requirements and are fit for use. QA is a staff function that prevents problems by heading them off, and by advising restraint and redirection at the proper time. It is also a catalytic function that should promote quality concepts, and encourage quality attitudes and discipline on the part of management and workers. Successful QA managers know how to make people quality conscious and to make them recognize the personal and organizational benefits of quality

Self Test

103. Software Quality Assurance (SQA) is defined as a -------and ------approach to the evaluation of the quality
    a) planned , systematic
    b) unplanned, systematic
    c) systematic ,planned,
    d) none of the above
104. Software Quality Assurance (**SQA**) consists of a ----------the software engineering processes and methods used to ensure quality.
    a) means of monitoring
    b) systematic
    c) planned,
    d) none of the above
105. Quality Assurance is a professional competency whose focus is directed at ------ processes.
    a) Critical
    b) Normal
    c) Abnormal,
    d) none of the above
106. SQA includes the process of assuring that ----------are established
    a) standards and procedures
    b) Normal and required process
    c) Abnormal,

d) none of the above

107. QA also provides an ----------for another important quality activity, continuous process improvement.
   a) Umbrella
   b) cover
   c) Abnormal,
   d) none of the above

108. Quality Assurance (QA) is the application of planned, systematic quality activities to ensure that the project will employ all ---------needed to ----------requirements
   a) processes , Meet
   b) cover, procedure
   c) Abnormal,
   d) none of the above

109. SQA assures that---------- and ----------standards exist
   a) clear , achievable
   b) cover, procedure
   c) Abnormal, achievable
   d) none of the above

110. A fundamental SQA technique is the ----------which looks at a--------.
   a) audit, process
   b) clear , achievable
   c) cover, procedure
   d) none of the above

111. The purpose of an SQA audit is to ---------that proper control ----------are being followed.
   a) Assure, procedures
   b) clear , achievable
   c) cover, procedure
   d) none of the above

112. Products are monitored for conformance to -------and processes are monitored for conformance to--------.
   a) Standards, procedures
   b) clear , achievable
   c) cover, procedure
   d) none of the above

113. Establishing ----------and ---------- for software development is critical.
   a) Standards, procedures
   b) clear , achievable
   c) cover, procedure
   d) none of the above

114. Standards are the established criteria to which the ----------are compared.
   a) software products
   b) clear , achievable
   c) cover, procedure
   d) none of the above

115. Procedures are the established criteria to which the ------ and -------are compared.
   a) development ,control processes
   b) clear , achievable
   c) cover, procedure
   d) none of the above

116. Documentation Standards specify form and content for ----------, --------- and product documentation and provide consistency throughout a project.

a) planning ,control
b) clear , achievable
c) cover, procedure
d) none of the above

117. Quality is a multifaceted ----------driven by customer ----------
a) concept , requirements.
b) clear , achievable
c) cover, procedure
d) none of the above

118. Management needs to develop----------, measurable ----------for each of these quality criteria for their development projects.
a) quantitative, standards
b) clear , achievable
c) cover, procedure
d) none of the above

119. A check sheet also called a ----------or----------
a) Checklist, tally sheet
b) clear , achievable
c) cover, procedure
d) none of the above

120. A check sheet of events or ----------is a form used to ----------and record data in an organized manner
a) Occurrences, gather
b) Checklist, tally sheet
c) clear , achievable
d) none of the above

121. A maturity model can provide:

a) A starting point ,A shared vision
b) Checklist, End point
c) clear , achievable
d) none of the above

122. Each key practice is sub-divided into --------common feature areas
a) Five
b) Four
c) Three
d) Two

Lesson 7

Software Configuration Management

Overview

Software Configuration Management, also known as Change Control Management

The dynamic nature of the majority business activities causes change in software or system. Changes require well-formulated and well-documented procedures to prevent the manipulation of programs for unauthorized purposes. The main objective of configuration management (or change control) is to get the right change installed at the right time. Change control concerns should be identified so that proper control mechanisms can be established to deal with the concerns.

Software configuration management is (SCM) is an umbrella activity that is applied throughout the software process. SCM activities are developed to identify change, control change, ensure that change is being properly implemented and report changes to others who are interested.

Learning Objectives

- To understand Software Configuration Management
- To understand What is Software Configuration Control
- To understand Baselines
- To understand What are the Configurable Items (artifacts)
- To understand Tools used for Software Configuration Management
- To understand Change Control Procedures
- To understand Template of Change Control Procedure /Process
- To understand Version Control, Software Configuration Audit, Status Reporting

What is Software Configuration Control *?*

Software Configuration Control is the systematic way of controlling the changes to the software items (artifacts)

The process is the primary responsibility of the software development staff. They must assure that the change requests are documented, that they are tracked through approval or rejection, and then incorporated into the developmental process. Many software development project teams group changes to be implemented into a new version of the system.

Benefits are

- Helps to store and retrieve the configurable items
- Allows to maintain customizations to the product in effective way
- Manages the versions along with change descriptions to help differentiate between the versions

What is the origin of these changes? (Identification)

- New business or market conditions dictates changes in product requirements
- New customer needs demand modification of data produced by information systems, functionality delivered by products
- Reorganization or business growth causes changes in project priorities or software engineering team structure
- Budgetary or scheduling constraints cause redefinition of the software system.

Baselines

- Customer wants to modify requirements. Developers want to modify the technical approach. Manager wants to modify the project strategy.

- Baseline means a product or specification (artifacts) that has been formally reviewed and agreed upon (mutually accepted) that there after serves as the basis for further development and that can be changed only through formal change control procedure.
- Baseline means milestone in the software development. For example Software requirement specification (SRS) have been documented, reviewed and corrected. That means in all respect SRS have been reviewed, corrected and approved, then SRS become a baseline document.
- Base lining is done when the product is ready to go to next phase
- Current state of product is frozen and further changes to the product are done along with change in version of the product.
- Defects are logged only for base lined items

What are the Configurable Items (artifacts*) ?*

- All types of Plans e.g. Project Plan, Test Plan, Quality Plan etc.
- Software Code
- Test Scripts and Test Case Documents
- Defect Log
- Test Reports
- User Documentation

Tools used for Software Configuration Management

There are many automated SCM tools are available to help /control SCM.

Main Frame Tools

- Change Man

Window Tools

- Microsoft's Visual Source Safe (VSS)
- IBM -Rational's Clear Case

Change Control Procedures

Several procedures are necessary to maintain control over program changes.

- The nature of the proposed change should be explained in writing, and formally approved by a responsible individual. Major changes should be approved by the systems-planning steering committee, commonly called the CCB or Configuration Control Board, in the same manner as for new systems. Minor changes may only require the joint approval of the IT manager and senior personnel in the user department.

Documenting the proposed change clears up any initial misunderstandings that may arise when only verbal requests are made. In addition, written proposals provide a history of changes in a particular system.

- Developers should make the program changes, not the operations group. Any change should be supported by adequate systems documentation. If the operators were authorized to make minor changes, it would greatly increase the difficulty of controlling versions and of maintaining up-to-date documentation.
- Someone independent of the person who designed and made the change should be responsible for testing the final revised program. The results should be recorded on program change registers and sent to the IT manager for approval. Operations should accept only properly approved changes.
- Finally, the documentation system should be updated with all change sheets or change registers and printouts.

Template of Change Control Procedure /Process

Change request from user

↓

Developer evaluation

↓

Change report document generated

↓

Change Control Board (CCB) checks for validity

↓

When it is valid, then assign individual (developer) to make change

↓

"Check out" the configurable item from SCM tool

↓

Make the change

↓

Review the change (over here audited that, change is made properly)

↓

"Check in" the configurable item in SCM tool (the configurable item becomes read only) gets the new version number

↓

Testing activity (purpose is to validate the change)

↓

Ready for promotion (inclusion in next release)

↓

Include changes in new version

↓

Release/ Distribute the new version

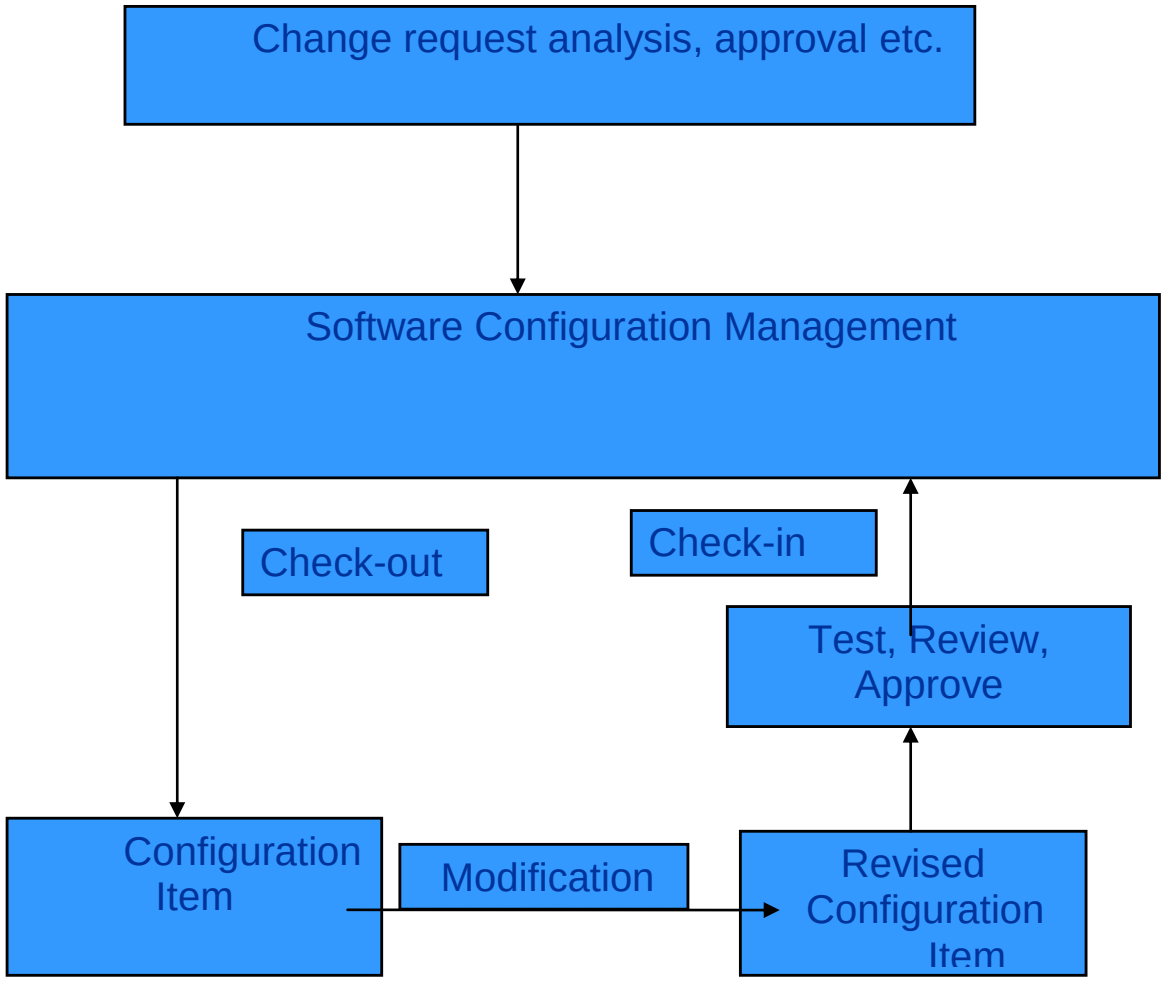


Figure A. Graphical View of SCM Process

Version Control

Version control combines process and SCM tools to manage different versions configuration objects that are created during the software development process.

SCM help the software developer to maintain different versions of the Software system.

For example. ABC Soft Solutions has developed banking domain software.

This is known as base product or core. This base product is generic in nature. This base product gets configured as per the requirement of HDFC bank, ICICI bank, and COSMOS bank. That means base product (source code, documents, data) get changed as per HDFC bank standards. To keep track of HDFC bank product new version number is given to modified base product.

Software Configuration Audit (SCA) and Status Reporting

Software Configuration Audit (SCA)

Identification, Version control, Change Control help the software developer to maintain order of software work product. (That means all latest versions of configurable items are readily available in project database) SCA is required to ensure that the change has been properly implemented.
The audit asks and answers the following questions.

- Has the change specified in the Change Control Procedures /Process been made?
- Has a formal technical review been conducted to assess technical correctness?
- Has the software process been followed and have software engineering standards been applied properly?
- Has SCM procedures for noting the change, recording it and reporting it been followed?

The SCA is conducted by the quality assurance group.

Status Reporting

Software Configuration status reporting gives the following information

- What happened?
- Who did it?
- When did it happen?
- What else will be affected?

Summary

Configuration Management is also known as Change Control Management. Configuration Control is the systematic way of controlling the changes to the software items. It helps to store and retrieve the configurable items, allows maintaining customizations to the product in effective way, and manages the versions along with change descriptions to help differentiate between the versions.
Base lining is done when the product is ready to go to next phase
Current state of product is frozen and further changes to the product are done along with change in version of the product. Defects are logged only for base lined items.
CM controls Build and Release Management.
The software is said to be built to release it to the customer.
The configurable items are together checked out for building the software.
Each configurable item could be having different version at the time of build.
Each build can be customized for different customers.
CM answers questions like the following:

- What is our current software configuration?
- What is its status?
- How do we control changes to our configuration?
- What changes have been made to our software?
- Do anyone else's changes affect our software?

Configuration and change control safeguard defined and approved baselines, prevent unnecessary or marginal changes, expedite worthwhile changes.

Configuration status accounting recording and tracking problem reports, change requests, change orders, etc

Self Test

123. Software Configuration Management, also known as ----------.
    a) Change Control Management b) Control Management c) Management

124. Software configuration management is (SCM) is an ---------- that is applied throughout the software process.
    a) umbrella activity b) activity c) planning activity
125. State the benefits of SCM.
126. The Configurable Items are ----------
    a) Project Plan b)Defect log c) Software Code d) All of the above
127. Explain the Base lining..
128. Explain SCM identification.
129. State the Tools used for Software Configuration Management
130. Explain the Change Control Procedures
131. Explain the Template of Change Control Procedure /Process.
132. Baseline means ----------in the software development.
    a) milestone b) activity c) planning activity
133. Base lining is done when the ----------is ready to go to next phase.
    a) Byproduct b) activity c) product
134. Defects are logged only for ----------items.
    a) Byproduct b) base lined c) standard
135. SCM manages the versions along with change descriptions to help ---------- between the versions.
    a) match b) differentiate c) standard
136. Major changes should be approved by the systems-planning steering committee, commonly called the----------
137. Developers should make the program changes. State whether this statement is true or false.
    a) True. b) False
138. SCM help the software developer to maintain different ----------of the Software system.
    a) Byproduct b) activity c) versions
139. SCA is required to ensure that the change has been properly----------.
    a) Byproduct b) incorporated c) implemented

140. The SCA is conducted by the quality assurance group. State whether this statement is true or false.
    a) True. b) False

141. State the importance of Check Out and Check In.
142. The configurable items are together checked out for building the software. State whether this statement is true or false.
    a) True. b) False

Lesson 8

Latest trends in Software Engineering

Overview

This chapter discusses the latest trends in Software Engineering. There are a number of areas where the advancement of software engineering is notable: The most important development was that new computers were coming out almost every year or two, rendering existing ones obsolete. Software people had to rewrite all their programs to run on these new machines. With the increasing demand for software in many smaller organizations, the need for inexpensive software solutions led to the growth of easy to operate, faster methodologies that created software, from requirements to deployment, quicker & easier. Using rapid-prototyping evolved to special methodologies, such as Extreme Programming (XP), which helped to simplify many areas of software engineering, including requirements gathering and reliability testing for the growing, vast number of small software systems. Very large software systems still used heavily-documented methodologies, with many volumes in the documentation set; however, smaller systems had a simpler, faster alternative approach to managing the development and maintenance of software calculations and algorithms, information storage/retrieval and display.

Learning Objectives

- To understand Web SE, Case Tools,
- To understand Agile programming, XP programming
- To understand UML
- To understand OOAD

Introduction - Web SE

The World Wide Web has become a major delivery platform for a variety of complex and sophisticated enterprise applications in several domains. In addition to their inherent multifaceted functionality, these Web applications exhibit complex behavior and place some unique demands on their usability, performance, security and ability to grow and evolve.

However, a huge majority of these applications continue to be developed in an unplanned way, contributing to problems of usability, maintainability, reliability and quality. While Web development can benefit from established practices from other related disciplines, it has definite distinctive characteristics that demand special considerations. Recently there have been some movements in solving these requirements and problems. **Web Engineering** is budding discipline which actively promotes systematic, quantifiable and disciplined approaches for successful creation of good quality, applications based on Web for everybody's use

In particular, Web engineering focuses on the methodologies, techniques and tools that are the foundation of Web application development and which support their design, development, evolution, and evaluation. Web application development has certain characteristics that make it different from traditional software, information system, or computer application development.

Web engineering is multidisciplinary and encompasses contributions from diverse areas: systems analysis and design, software engineering, hypermedia/hypertext engineering, requirements engineering, human-computer interaction, user interface, information engineering, information indexing and retrieval, testing, modeling and simulation, project management, and graphic design and presentation.

Web engineering is neither a replica nor a part of software engineering, even if both involve programming and software development. While Web Engineering uses software engineering principles, it encompasses new approaches, methodologies, tools, techniques, and guidelines to meet the unique requirements of Web-based applications.

Computer-Aided Software Engineering (CASE), in the field of Software Engineering is the scientific application of a set of tools and methods to software which is meant to result in high-quality, defect-free, and maintainable software work products. CASE tools refer to methods for the development of information systems among automated tools that can be used in the software development process.

The "Computer-aided software engineering" (CASE) is used for the automated development of systems software, i.e., computer code. The CASE functions comprise analysis, design, and coding. CASE tools automate methods for designing, documenting, and producing structured computer code in the desired programming language.

Two key ideas of Computer Aided Software Engineering are:

- The harboring of computer assistance in software development and or software maintenance processes, and
- An engineering approach to the software development and or maintenance.

Some typical CASE tools are:

- Configuration management tools
- Data modeling tools
- Model transformation tools
- Program transformation tools
- Refactoring tools
- Source code generation tools, and
- Unified Modeling Language

Many CASE tools not only output code but also generate other output typical of various systems analysis and design methodologies such as

- Data flow diagram
- Entity relationship diagram
- Logical schema
- Program specification
- User documentation

New trends in Software Development

Agile Software Development

Agile software development processes are built on the foundation of iterative development. To that foundation they add a lighter, more people-centric viewpoint than traditional approaches. Agile processes use feedback, rather than planning, as their primary control mechanism. The feedback is driven by regular tests and releases of the evolving software.

There are many specific agile development methods. Most promote development iterations, teamwork, collaboration, and process adaptability throughout the life-cycle of the project.

Agile methods break tasks into small increments with minimal planning, and don't directly involve long-term planning. Iterations are short time boxes that typically last from 1 week to 4 weeks. All iteration involves a team working through a full software development cycle including planning, requirements analysis, design, coding, unit testing, and acceptance testing when a working product is demonstrated to stakeholders. This helps minimize overall risk, and lets the project adapt to modification rapidly. Stakeholders produce documentation as required. Iteration may not add enough functionality to warrant a market release, but the goal is to have an available release (with minimal bugs) at the end of each, iteration. Multiple iterations may be required to release a product or new features.

Team composition in an agile project is usually cross-functional (different participant from different departments) and self-organizing without consideration for any existing corporate hierarchy or the corporate roles of team members. Team members normally take responsibility for tasks that deliver the functionality iteration requires. They decide individually how to meet iteration's requirements.

Agile methods emphasize face-to-face communication over written documents when the team is all in the same location. When a team works in different locations, they maintain daily contact through videoconferencing, voice, e-mail, etc. Most agile teams work in a single open office, which facilitates such communication. Team size is 5 to 9 members to help make team communication and team collaboration easier. Larger development hard work may be delivered by multiple teams working toward a common goal or different parts of hard work. This may also require a coordination of priorities among teams. No matter what development disciplines are required, each agile team will contain a customer representative. This person is appointed by stakeholders to act on their behalf and makes a personal commitment to being available for developers to answer mid-iteration problem-domain questions. At the end iteration, stakeholders and the customer representative review progress and re-evaluate priorities with a view to optimizing the return on investment and ensuring alignment with customer needs and company goals.

Most agile implementations use a routine and formal daily face-to-face communication among team members. This specifically includes the customer representative and any interested stakeholders as observers. In a concise meeting, team members report to each other what they did yesterday, what they intend to do today, and what their roadblocks are. This standing face-to-face communication prevents problems being hidden. Agile emphasizes working software as the primary measure of progress. This, combined with the preference for face-to-face communication, produces a lesser amount of written documentation than other methods—though, in an agile project, documentation and other artifacts/documents rank equally with a working product. The agile method boosts stakeholders to prioritize requirements/needs with other iteration outcomes based exclusively on business value perceived at the beginning of the iteration.

Specific tools and techniques such as continuous integration, automated or xUnit test, pair programming, test driven development, design patterns, domain-driven design, code refactoring and other techniques are often used to improve quality and enhance project agility.

XP: Extreme Programming

Extreme Programming (XP) is a software engineering methodology which is intended to improve software quality and responsiveness to changing customer requirements. As a type of agile software development, it advocates frequent "releases" in small development cycles, which is intended to improve productivity and launch checkpoints where new customer requirements can be adopted.

Other elements of Extreme Programming include: programming in pairs or doing extensive code review, unit testing of all code, avoiding programming of features until they are actually needed, a flat management structure, simplicity and clarity in code, expecting changes in the customer's requirements as time passes and the problem is better understood, and frequent communication with the customer and among programmers. The methodology takes its name from the idea that the beneficial elements of traditional software engineering practices are taken to "extreme" levels, on the theory that if some is good, more is better.

"Extreme Programming Explained" describes Extreme Programming as a software development regulation that organizes people to create high quality software more productively.

In conventional system development methods (such as Structured System Analysis and Design Methodology (SSADM) or the waterfall model) the requirements for the system are determined at the beginning of the development project and often fixed from that point on. This means that the cost of changing the requirements at a later stage (a common feature of software engineering projects) will be high. Like other agile software development methods, XP attempts to reduce the cost of change by having multiple short development cycles, rather than one long one. In this policy changes are a natural, inescapable and desirable aspect of software development projects, and should be planned for instead of attempting to define a stable set of requirements. Extreme Programming also introduces a number of basic values, principles and practices on top of the agile programming framework.

Extreme Programming Activities

XP describes four basic activities that are performed within the software development process.
Coding, Testing, Designing and Listening

Coding

The advocates of XP argue that the only truly important product of the system development process is code - software instructions a computer can understand. Without code, there is no work product. Coding can also be used to figure out the most suitable solution. For instance, XP would advocate that faced with several alternatives for a programming difficulty, one should simply code all solutions and determine with automated tests which solution is most appropriate. Coding can also help to communicate thoughts about programming problems. A programmer dealing with a complex programming problem and finding it hard to explain the solution to fellow programmers might code it and use the code to demonstrate what he or she means. Code, say the proponents of this position, is always clear and concise and cannot be interpreted in more than one way. Other programmers can give feedback on this code by also coding their thoughts.

Testing

One cannot be certain that a function works unless one tests it. Bugs and design errors are pervasive problems in software development. Extreme Programming's approach is that if a little testing can eliminate a few flaws, a lot of testing can eliminate many more flaws. Unit tests determine whether a given feature works as intended. A programmer writes as many automated tests as they can think of that might "break" the code; if all tests run effectively, then the coding is complete. Every piece of code that is written is tested before moving on to the next feature. Acceptance tests verify that the requirements as understood by the programmers satisfy the customer's actual requirements. These occur in the exploration phase of release planning.

Listening

Programmers must listen to what the customers need the system should do, what "business logic" is needed. They must be aware of these needs well enough to give the customer feedback about the technical aspects of how the problem might be solved, or cannot be solved, by understanding of his or her problem.

Designing

From the point of view of simplicity, one could say that system development doesn't need more than coding, testing and listening. If those activities are performed well, the result should always be a system that works. In practice, this will not work. One can come far away without designing but at a specified time one will get stuck up. The system becomes too complex and the dependencies within the system cease to be clear. One can stay away from this by creating a design structure that manages the logic in the system. High-quality design will avoid lots of dependencies within a system. That means altering one part of the system will not affect other parts of the system

Values

Extreme Programming having five values, as described below:-

Communication

Building software systems requires communicating system requirements to the developers of the system. In proper software development methodologies, this task is completed through documentation. Extreme Programming techniques can be viewed as methods for rapidly building and imparting institutional knowledge among members of a development team. The goal is to give all developers a shared view of the system which matches the view held by the users of the system. To this end, Extreme Programming special treatments simple designs, common descriptions, collaboration of users and programmers, frequent verbal communication, and feedback.

Simplicity

Extreme Programming promotes the easy solution. Afterwards Extra functionality can added. The difference between this approach and usual system development methods is the focus on designing and coding for the needs of today instead of those of tomorrow, next week, or next month. This is sometimes summed up as the "you're not going to need it" approach Proponents of XP acknowledge the disadvantage that this can sometimes entail more effort tomorrow to change the system; their claim is that this is more than compensated for by the advantage of not investing in possible upcoming requirements that might change before they become relevant. Coding and

designing for uncertain future requirements implies the risk of spending resources on something that might not be needed. Related to the "communication" value, simplicity in design and coding should improve the quality of communication. A simple design with very simple code could be easily understood by most programmers in the team.

Feedback

In Extreme Programming, feedback/view relates to different dimensions of the system development:

- Feedback from the system: by writing unit tests, or running periodic integration tests, the programmers have direct feedback from the state of the system after implementing changes.
- Feedback from the customer: The functional tests (acceptance tests) are written by the customer and the testers. They will get concrete feedback about the current state of their system. This review is planned once in every two or three weeks so the customer can easily steer the development.
- Feedback from the team: When customers come up with new requirements in the planning game the team directly gives an estimation of the time that it will take to implement. Feedback is closely related to communication and simplicity. Flaws in the system are easily communicated by writing a unit test that proves a certain piece of code will break.

The direct feedback from the system tells programmers to recode this part. A customer is able to test the system periodically according to the functional requirements, known as user stories.

Courage

Several practices embody courage. One is the commandment to always design and code for today and not for tomorrow. This is an effort to avoid getting bogged down in design and requiring a lot of effort to apply anything else. Courage enables developers to feel comfortable with refactoring their code when essential. This means reviewing the existing system and modifying it so that future changes can be executed more easily. Another example of courage is knowing when to throw code away: courage to remove source code that is obsolete, no matter how much effort was used to create that source code. Also, courage means persistence: A programmer might be stuck on a complex problem for an entire day, then solve the problem quickly the next day, if only they are persistent.

Respect

The respect value matters in several ways. In Extreme Programming, team members respect each other because programmers should never commit changes that break compilation, that make existing unit-tests fail, or that otherwise delay the work of their peers. Members respect their work by always striving for high quality and seeking for the best design for the solution.

Adopting the four earlier values leads to respect gained from others in the team. Nobody on the team should feel overlooked or unappreciated. This confirms high level of motivation and encourages loyalty toward the team, and the goal of the project. This value is very dependent upon the other values, and is very much oriented toward people in a team.

Unified Modeling Language (UML)

The UML is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems. The UML is a very important part of developing objects oriented software and the software development process. The UML generally uses graphical notations to express the design of software projects. Using the UML helps project teams communicate, explore potential designs, and validate the architectural design of the software.

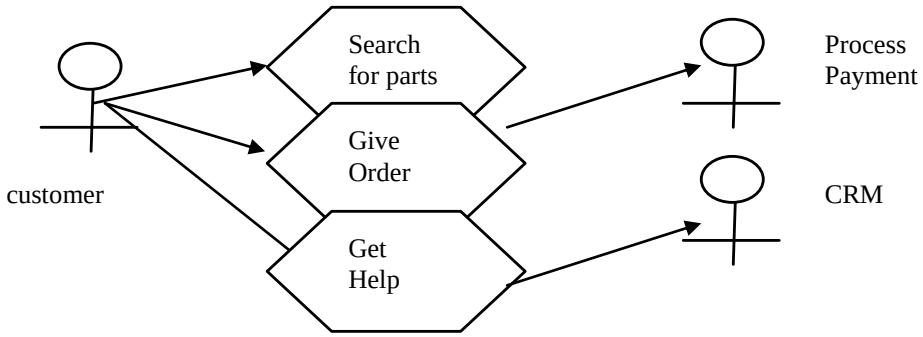
The primary goals in the design of the UML were:

- Provide users with a ready-to-use, meaningful visual modeling language so they can develop and exchange meaningful models.
- Provide extensibility and specialization mechanisms to extend the core concepts.
- Be independent of particular programming languages and development processes.
- Provide a formal basis for understanding the modeling language.
- Encourage the growth of the object oriented (OO) tools market.
- Support higher-level development concepts such as collaborations, frameworks, patterns and components.
- Integrate best practices.

Why Use UML?

As the strategic value of software increases for many companies, the industry looks for techniques to automate the production of software and to improve quality and reduce cost and time-to-market. These techniques include component technology, visual programming, patterns and frameworks. Businesses also seek techniques to manage the complexity of systems as they increase in scope and scale. In particular, they recognize the need to resolve persistent architectural problems, such as concurrency, physical distribution, replication, security, load balancing and fault tolerance. Additionally, the development for the Website , while making some things simpler, has increased the severity of these architectural problems. The Unified Modeling Language (UML) was designed to respond to these needs.

Figure 1. Using System boundary boxes to indicate releases

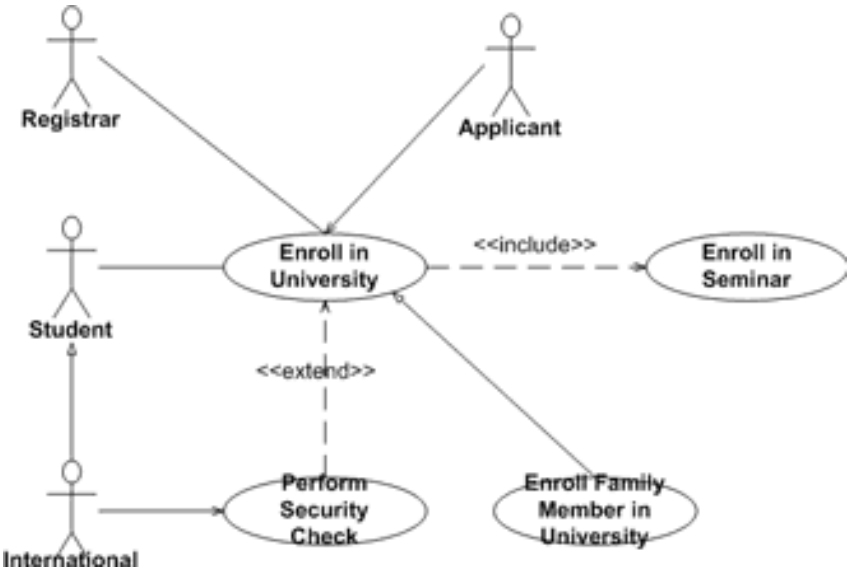


Creating Use Case Diagrams

You should ask how the actors interact with the system to identify an initial set of use cases. Then, on the diagram, you connect the actors with the use cases with which they are involved. If actor supplies information, initiates the use case, or receives any information as a result of the use case, then there should be an association between them.

The foregoing paragraph describes my common use case modeling style, an “actors first” approach. Others like to start by identifying one actor and the use cases that they’re involved with first and then evolve the model from there. Both approaches work. The important point is that different people take different approaches. Figure 2 shows the three types of relationships between use cases -- extends, includes, and inheritance -- as well as inheritance between actors. I like to think of extend relationships as the equivalent of a "hardware interrupt" because you don't know when or if the extending use case will be invoked (perhaps a better way to look at this is extending use cases are conditional). Include relationships as the equivalent of a procedure call. Inheritance is applied in the same way as you would on UML class diagrams -- to model specialization of use cases or actors in this case.

Figure 2. Use case with relationship



So how can you keep use case modeling responsive? Very first, focus on keeping it simple. Use simple, flexible tools to model with. I’ll typically create use case diagrams on a whiteboard, which is an example of an initial diagram that I would draw with my project stakeholders. The content is more important than representation, so it isn’t a big issue that the diagram is hand drawn, it’s just barely good enough and that’s all that we need. It’s also perfectly okay that the diagram isn’t complete, there’s clearly more to a university than what is depicted, because we can always modify the diagram as we need to.

Object-oriented analysis and design (OOAD)

Object-oriented analysis and design (OOAD) is a software engineering approach that models a system as a group of communicating objects. Each object represents some entity of interest in the system being modeled, and is characterized by its class, its data elements, and its behavior. Different models can be created to show the static structure, dynamic behavior, and run-time

deployment of these collaborating objects. There is different number of notations for representing these models, such as the UML.

Object-oriented analysis (OOA) applies object-modeling techniques to analyze the functional requirements for a system. Object-oriented design (OOD) elaborates the analysis models to produce implementation specifications. OOA focuses on what the system does, OOD on how the system does it.

Object-oriented systems

An object-oriented system is composed of objects. The behavior of the system results from the collaboration of those objects. Collaboration between objects involves those sending messages to each other. Sending a message differs from calling a function in that when a target object receives a message, it itself decides what function to carry out to service that message. The same message may be implemented by many different functions, the one selected depending on the state of the target object. The implementation of "message sending" changes depending on the architecture of the system being modeled, and the location of the objects.

Object-oriented analysis

Object-oriented analysis (OOA) looks at the problem domain, with the plan of producing a theoretical model of the information that exists in the area being scrutinized. Analysis models do not consider any implementation constraints that might exist, such as concurrency, distribution, persistence, or how the system is to be built. Implementation constraints are processed during object-oriented design (OOD). Analysis is done before the Design. The sources for the analysis can be a written requirements statement, a formal vision document, and interviews with stakeholders or other interested parties. A system may be divided into various domains, on behalf of different business, technological, or other areas of interest, each of which are analyzed separately.

The result of object-oriented analysis is a description of what the system is functionally required to do, in the form of a theoretical model. It will be presented as a set of use cases, one or more UML class diagrams, and a number of interaction diagrams. It may also comprise sort of user interface mock-up. The object oriented analysis is to develop a model that describe computer software as it works to satisfy, defined customer requirements..

Object-oriented design

(OOD) Object-oriented design converts the conceptual model produced in object-oriented analysis to take account of the constraints imposed by the chosen architecture and any non-functional – technological or environmental – constraints, such as transaction throughput, response time, run-time platform, development environment, or programming language. The concepts in the analysis model are mapped onto implementation classes and interfaces. The result is a model of the solution domain, a detailed description of *how* the system is to be built.

Summary

Web-based systems and applications (WebApps) deliver a complex array of content and functionality to a broad population of end-users. Web-engineering is the process that is used to create high-quality WebApps. Web engineering (WebE) is not a perfect clone of software engineering, but it borrows many of software engineering's fundamental concepts and principles, emphasizing the same technical and management activities. There are subtle differences in the way these activities are conducted, but an overriding philosophy that dictates a disciplined approach to the development of a computer-based system is identical.

Web applications are multidisciplinary. They are built in a constantly changing environment where requirements are unstable and the development teams typically small. The user community is wider than before and competition may be spread across the world. Quality Web applications need to be usable, functional, reliable, maintainable, scalable and secure. These demands on Web applications are radically different from those made on conventional applications. There is thus a strong need for Web Engineering. The increasing integration of software engineering and systems engineering

Extreme Programming (XP) is the best-known iterative process. In XP, the phases are carried out in extremely small (or "continuous") steps compared to the older, "batch" processes. The first pass through the steps might take a day or a week, rather than the months or years of each complete step in the Waterfall model. First, one writes automated tests, to provide concrete goals for development. Next is coding (by a pair of programmers), which is complete when all the tests pass, and the programmers can't think of any more tests that are needed. Design and architecture emerge out of refactoring, and come after coding. Design is done by the same people who do the coding. (Only the last feature - merging design and code - is common to all the other agile processes.) The incomplete but functional system is deployed or demonstrated for (some subset of) the users (at least one of which is on the development team). At this point, the practitioners start again on writing tests for the next most important part of the system.

Object-oriented analysis (OOA) applies object-modeling method to analyze the functional requirements for a system. Object-oriented design (OOD) elaborates the analysis models to produce implementation specifications. OOA focuses on what the system does, OOD on how the system does it.

In nutshell future trends will be as mentioned below:-

- An increased emphasis on users and end value
- Increasing criticality and need for dependability & security
- Increasingly rapid change
- Increasing SIS globalization and need for interoperability
- Increasingly complex systems of systems
- Increasing needs for COTS, reuse, and legacy SIS integration
- Computational plenty

Self Test

143. The World Wide Web has become a ----------delivery platform for a variety of ----------and sophisticated enterprise applications in several domains.
    a) Major , complex
    b) Minor, simple
    c) Major , simple
    d) Minor, complex
144. Web applications exhibit ----------behavior and place some ------demands on their usability, performance, security and ability to grow and evolve.
    a) Complex, unique
    b) Minor, unique
    c) unique , simple
    d) Minor, complex
145. Web engineering focuses on the ----------,----------and tools that are the foundation of Web application development.
    a) methodologies, techniques
    b) Complex, techniques
    c) techniques, unique
    d) unique , simple

146. **Computer-Aided Software Engineering** (CASE), in the field of Software Engineering is the ----------application of a set of tools and --------to software.
    a) Scientific, methods
    b) methodologies, techniques
    c) Complex, unique
    d) Minor, unique
147. **Some** typical CASE tools are:
    a) All mentioned below
    b) Configuration management tools
    c) Data modeling tools
    d) Model transformation tools

148. Agile software development processes are built on the foundation of ------- development.
    a) iterative
    b) methodologies
    c) Complex, unique
    d) Unique
149. Agile methods emphasize ----------communication over written documents when the team is all in the same location.
    a) face-to-face
    b) one-to-one
    c) one-to-all
    d) normal
150. Agile emphasizes working software as the ---------measure of progress.
    a) primary

b) iterative
c) secondary
d) unique

151. Agile methods break tasks into -------increments with --------planning, and don't directly involve long-term planning.
a) Small, minimal
b) Big, maximum
c) iterative , minimum
d) secondary, maximum

152. Team composition in an agile project is usually --------
a) cross-functional.
b) functional
c) iterative
d) unique

153. In Agile , at the end of each ---------stakeholders and the customer representative --------progress.
a) iteration, review
b) cross-functional., review
c) functional, review
d) incremental , review

154. **Extreme Programming (XP)** is a software engineering ---------which is intended to ---------software quality and responsiveness to changing customer requirements.
a) Methodology, improve
b) system, review
c) cross-functional., review
d) nonfunctional, review

155. **Extreme Programming (XP)** describes -------basic activities that are -----within the software development process.
a) four , performed
b) five, review
c) cross-functional., review
d) nonfunctional, review

156. **In Extreme Programming (XP),** Unit tests ---------whether a given feature works as intended.
a) determine
b) shows
c) fixes
d) dictates

157. In Extreme Programming, ---------relates to ---------dimensions of the system development:
a) Feedback, different
b) Determine, different
c) Shows, different
d) Information, different

158. Unified Modeling Language (UML) is a --------language for specifying, visualizing, constructing, and ---------the artifacts of software systems.
a) Standard, documenting
b) Common , documenting
c) Routine, documenting
d) cross-functional., documenting

159. The UML is a very ----------part of --------objects oriented software and the software development process.
   a) Important, developing
   b) Standard, developing
   c) Common , documenting
   d) Routine, documenting

160. The primary goals in the design of the UML were:
   a) Provide users with a ready-to-use, expressive visual modeling language
   b) Provide users with a ready, expressive visual language
   c) Provide users with a pure, expressive visual modeling language
   d) Provide users with a ready-to-use, expressive visual modeling

161. In UML, you should ask how the --------interact with the system to identify an initial set of----------.
   a) Actors, use cases
   b) creator, use cases
   c) Actors, system cases
   d) system, use cases

162. Extreme Programming (XP) is the best-known --------process.
   a) Iterative
   b) cross-functional.
   c) Functional
   d) Productive